第1章 概 述

1.1 引言

很多不同的厂家生产各种型号的计算机，它们运行完全不同的操作系统，但 TCP/IP协议族允许它们互相进行通信。这一点很让人感到吃惊，因为它的作用已远远超出了起初的设想。TCP/IP起源于60年代末美国政府资助的一个分组交换网络研究项目，到 90年代已发展成为计算机之间最常应用的组网形式。它是一个真正的开放系统，因为协议族的定义及其多种实现可以不用花钱或花很少的钱就可以公开地得到。它成为被称作“ 全球互联网 ”或“ 因特网(Internet) ”的基础，该广域网（WAN）已包含超过100万台遍布世界各地的计算机。

本章主要对TCP/IP协议族进行概述，其目的是为本书其余章节提供充分的背景知识。如果读者要从历史的角度了解有关 TCP/IP的早期发展情况，请参考文献 [Lynch 1993]。

1.2 分层

网络协议通常分不同层次进行开发，每一层分别负责不同的通信功能。一个协议族，比如 TCP/IP，是一组不同层次上的多个协议的组合。 TCP/IP通常被认为是一个四层协议系统，如图 1-1所示。

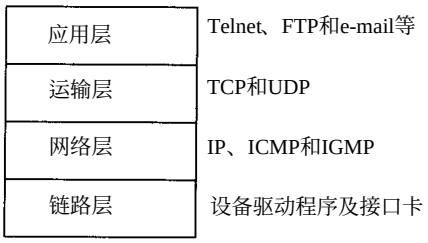


图1-1 TCP/IP协议族的四个层次

每一层负责不同的功能：

1) 链路层，有时也称作数据链路层或网络接口层，通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡。它们一起处理与电缆（或其他任何传输媒介）的物理接口细节。
2) 网络层，有时也称作互联网层，处理分组在网络中的活动，例如分组的选路。在TCP/IP协议族中，网络层协议包括 IP协议（网际协议），ICMP协议（Internet互联网控制报文协议），以及IGMP协议（Internet组管理协议）。
3) 运输层主要为两台主机上的应用程序提供端到端的通信。在 TCP/IP协议族中，有两个互不相同的传输协议：TCP（传输控制协议）和UDP（用户数据报协议）。
   TCP为两台主机提供高可靠性的数据通信。它所做的工作包括把应用程序交给它的数据分成合适的小块交给下面的网络层，确认接收到的分组，设置发送最后确认分组的超时时钟等。由于运输层提供了高可靠性的端到端的通信，因此应用层可以忽略所有这些细节。
   而另一方面， UDP则为应用层提供一种非常简单的服务。它只是把称作数据报的分组从一台主机发送到另一台主机，但并不保证该数据报能到达另一端。任何必需的可靠性必须由应用层来提供。
   这两种运输层协议分别在不同的应用程序中有不同的用途，这一点将在后面看到。
4) 应用层负责处理特定的应用程序细节。几乎各种不同的 TCP/IP实现都会提供下面这些通用的应用程序：

- Telnet 远程登录。
- FTP 文件传输协议。
- SMTP 简单邮件传送协议。
- SNMP 简单网络管理协议。

另外还有许多其他应用，在后面章节中将介绍其中的一部分。

假设在一个局域网（LAN）如以太网中有两台主机，二者都运行 FTP协议，图1-2列出了该过程所涉及到的所有协议。

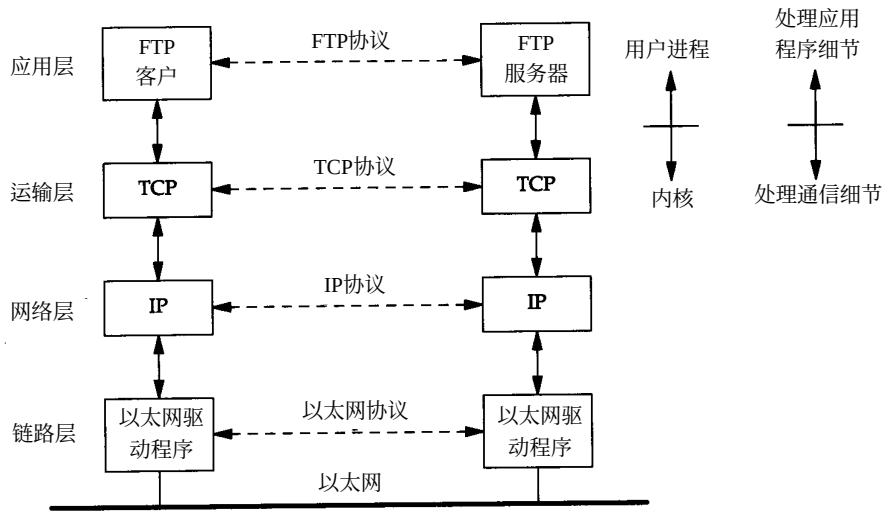


图1-2 局域网上运行FTP的两台主机

这里，我们列举了一个FTP客户程序和另一个FTP服务器程序。大多数的网络应用程序都被设计成客户—服务器模式。服务器为客户提供某种服务，在本例中就是访问服务器所在主机上的文件。在远程登录应用程序Telnet中，为客户提供的服务是登录到服务器主机上。

在同一层上，双方都有对应的一个或多个协议进行通信。例如，某个协议允许 TCP层进行通信，而另一个协议则允许两个IP层进行通信。

在图1-2的右边，我们注意到应用程序通常是一个用户进程，而下三层则一般在（操作系统）内核中执行。尽管这不是必需的，但通常都是这样处理的，例如 UNIX操作系统。

在图1-2中，顶层与下三层之间还有另一个关键的不同之处。应用层关心的是应用程序的细节，而不是数据在网络中的传输活动。下三层对应用程序一无所知，但它们要处理所有的通信细节。

在图1-2中列举了四种不同层次上的协议。 FTP是一种应用层协议， TCP是一种运输层协议， IP是一种网络层协议，而以太网协议则应用于链路层上。 TCP/IP协议族是一组不同的协议组合在一起构成的协议族。尽管通常称该协议族为 TCP/IP，但TCP和IP只是其中的两种协议而已（该协议族的另一个名字是Internet协议族(Internet Protocol Suite)）。

网络接口层和应用层的目的是很显然的 —— 前者处理有关通信媒介的细节（以太网、令牌环网等），而后者处理某个特定的用户应用程序（ FTP、Telnet等）。但是，从表面上看，网络层和运输层之间的区别不那么明显。为什么要把它们划分成两个不同的层次呢？为了理解这一点，我们必须把视野从单个网络扩展到一组网络。

在80年代，网络不断增长的原因之一是大家都意识到只有一台孤立的计算机构成的“孤岛”没有太大意义，于是就把这些孤立的系统组在一起形成网络。随着这样的发展，到了90年代，我们又逐渐认识到这种由单个网络构成的新的更大的“岛屿”同样没有太大的意义。于是，人们又把多个网络连在一起形成一个网络的网络，或称作互连网(internet)。一个互连网就是一组通过相同协议族互连在一起的网络。

构造互连网最简单的方法是把两个或多个网络通过路由器进行连接。它是一种特殊的用于网络互连的硬件盒。路由器的好处是为不同类型的物理网络提供连接：以太网、令牌环网、点对点的链接和FDDI（光纤分布式数据接口）等等。

这些盒子也称作IP路由器（IP Router），但我们这里使用路由器(Router)这个术语。从历史上说，这些盒子称作网关（gateway），在很多TCP/IP文献中都使用这个术语。现在网关这个术语只用来表示应用层网关：一个连接两种不同协议族的进程（例如，TCP/IP和IBM的SNA），它为某个特定的应用程序服务（常常是电子邮件或文件传输）。

图1-3是一个包含两个网络的互连网：一个以太网和一个令牌环网，通过一个路由器互相连接。尽管这里是两台主机通过路由器进行通信，实际上以太网中的任何主机都可以与令牌环网中的任何主机进行通信。

在图1-3中，我们可以划分出端系统（End system）（两边的两台主机）和中间系统（Intermediate system）（中间的路由器）。应用层和运输层使用端到端（End-to-end）协议。在图中，只有端系统需要这两层协议。但是，网络层提供的却是逐跳（Hop-by-hop）协议，两个端系统和每个中间系统都要使用它。

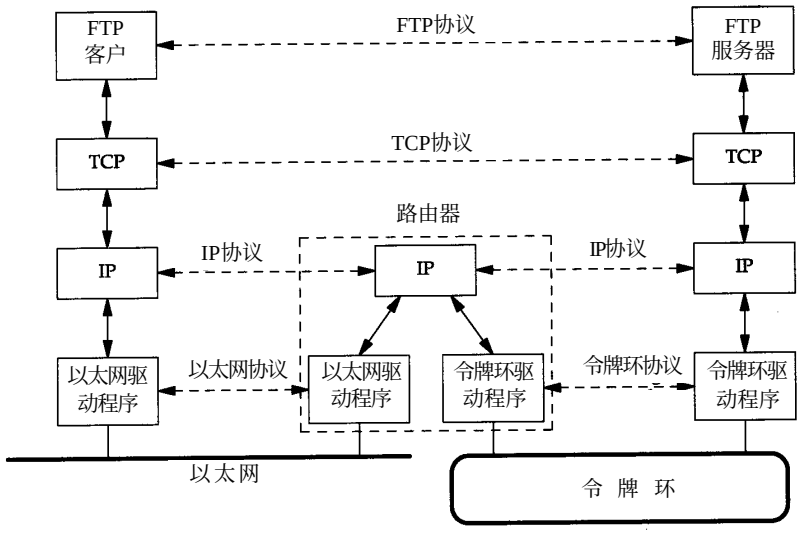


图1-3 通过路由器连接的两个网络

在TCP/IP协议族中，网络层IP提供的是一种不可靠的服务。也就是说，它只是尽可能快地把分组从源结点送到目的结点，但是并不提供任何可靠性保证。而另一方面，TCP在不可靠的IP层上提供了一个可靠的运输层。为了提供这种可靠的服务，TCP采用了超时重传、发送和接收端到端的确认分组等机制。由此可见，运输层和网络层分别负责不同的功能。

从定义上看，一个路由器具有两个或多个网络接口层（因为它连接了两个或多个网络）。

任何具有多个接口的系统，英文都称作是多接口的(multihomed)。一个主机也可以有多个接口，但一般不称作路由器，除非它的功能只是单纯地把分组从一个接口传送到另一个接口。同样，路由器并不一定指那种在互联网中用来转发分组的特殊硬件盒。大多数的 TCP/IP实现也允许一个多接口主机来担当路由器的功能，但是主机为此必须进行特殊的配置。在这种情况下，我们既可以称该系统为主机（当它运行某一应用程序时，如 FTP或Telnet），也可以称之为路由器（当它把分组从一个网络转发到另一个网络时）。在不同的场合下使用不同的术语。

互联网的目的之一是在应用程序中隐藏所有的物理细节。虽然这一点在图 1-3由两个网络组成的互联网中并不很明显，但是应用层不能关心（也不关心）一台主机是在以太网上，而另一台主机是在令牌环网上，它们通过路由器进行互连。随着增加不同类型的物理网络，可能会有20个路由器，但应用层仍然是一样的。物理细节的隐藏使得互联网功能非常强大，也非常有用。

连接网络的另一个途径是使用网桥。网桥是在链路层上对网络进行互连，而路由器则是在网络层上对网络进行互连。网桥使得多个局域网（ LAN）组合在一起，这样对上层来说就好像是一个局域网。

TCP /IP倾向于使用路由器而不是网桥来连接网络，因此我们将着重介绍路由器。文献[Perlman 1992]的第12章对路由器和网桥进行了比较。

1.3 TCP/IP的分层

在TCP/IP协议族中，有很多种协议。图1-4给出了本书将要讨论的其他协议。

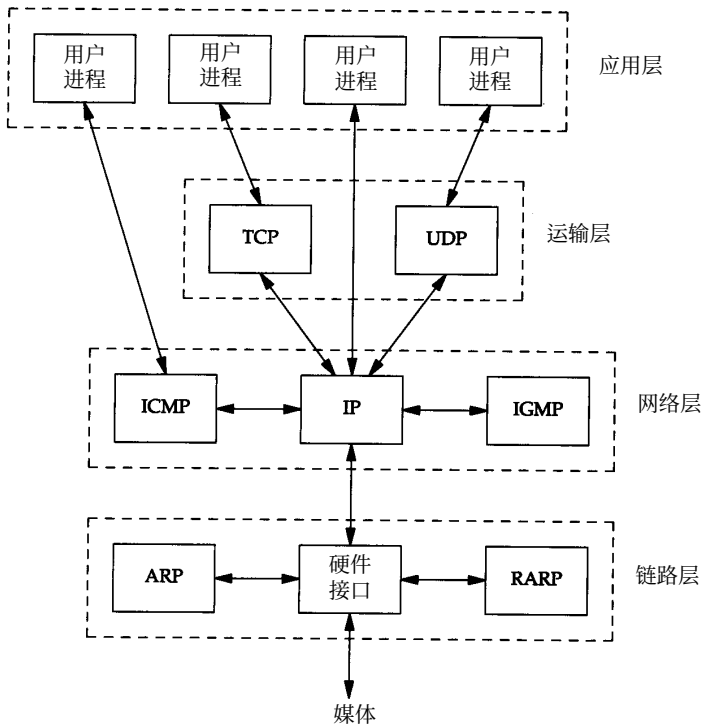


图1-4 TCP/IP协议族中不同层次的协议

TCP和UDP是两种最为著名的运输层协议，二者都使用IP作为网络层协议。

虽然TCP使用不可靠的IP服务，但它却提供一种可靠的运输层服务。本书第17～22章将详细讨论TCP的内部操作细节。然后，我们将介绍一些TCP的应用，如第26章中的Telnet和Rlogin、第27章中的FTP以及第28章中的SMTP等。这些应用通常都是用户进程。

UDP为应用程序发送和接收数据报。一个数据报是指从发送方传输到接收方的一个信息单元（例如，发送方指定的一定字节数的信息）。但是与TCP不同的是，UDP是不可靠的，它不能保证数据报能安全无误地到达最终目的。本书第11章将讨论UDP，然后在第14章（DNS: 域名系统），第15章（TFTP：简单文件传送协议），以及第16章（BOOTP：引导程序协议）介绍使用UDP的应用程序。SNMP也使用了UDP协议，但是由于它还要处理许多其他的协议，因此本书把它留到第25章再进行讨论。

IP是网络层上的主要协议，同时被TCP和UDP使用。TCP和UDP的每组数据都通过端系统和每个中间路由器中的IP层在互联网中进行传输。在图1-4中，我们给出了一个直接访问IP的应用程序。这是很少见的，但也是可能的（一些较老的选路协议就是以这种方式来实现的。当然新的运输层协议也有可能使用这种方式）。第3章主要讨论IP协议，但是为了使内容更加有针对性，一些细节将留在后面的章节中进行讨论。第9章和第10章讨论IP如何进行选路。

ICMP是IP协议的附属协议。IP层用它来与其他主机或路由器交换错误报文和其他重要信息。第6章对ICMP的有关细节进行讨论。尽管ICMP主要被IP使用，但应用程序也有可能访问它。我们将分析两个流行的诊断工具，Ping和Traceroute（第7章和第8章），它们都使用了ICMP。

IGMP是Internet组管理协议。它用来把一个UDP数据报多播到多个主机。我们在第12章中描述广播（把一个UDP数据报发送到某个指定网络上的所有主机）和多播的一般特性，然后在第13章中对IGMP协议本身进行描述。

ARP（地址解析协议）和RARP（逆地址解析协议）是某些网络接口（如以太网和令牌环网）使用的特殊协议，用来转换IP层和网络接口层使用的地址。我们分别在第4章和第5章对这两种协议进行分析和介绍。

1.4 互联网的地址

互联网上的每个接口必须有一个唯一的Internet地址（也称作IP地址）。IP地址长32 bit。Internet地址并不采用平面形式的地址空间，如1、2、3等。IP地址具有一定的结构，五类不同的互联网地址格式如图1-5所示。

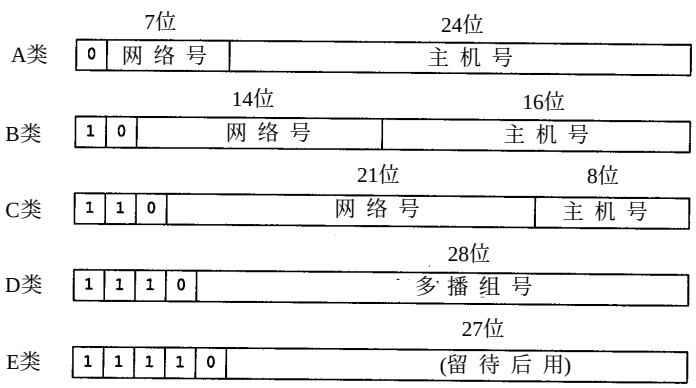


图1-5 五类互联网地址

这些32位的地址通常写成四个十进制的数，其中每个整数对应一个字节。这种表示方法称作“点分十进制表示法（Dotted decimal notation）”。例如，作者的系统就是一个B类地址，它表示为：140.252.13.33。

区分各类地址的最简单方法是看它的第一个十进制整数。图1-6列出了各类地址的起止范围，其中第一个十进制整数用加黑字体表示。

类型	范 围
A	**0**.0.0.0 到 **127**.255.255.255
B	**128**.0.0.0 到 **191**.255.255.255
C	**192**.0.0.0 到 **223**.255.255.255
D	**224**.0.0.0 到 **239**.255.255.255
E	**240**.0.0.0 到 **247**.255.255.255

图1-6 各类IP地址的范围

需要再次指出的是，多接口主机具有多个IP地址，其中每个接口都对应一个IP地址。

由于互联网上的每个接口必须有一个唯一的IP地址，因此必须要有一个管理机构为接入互联网的网络分配IP地址。这个管理机构就是互联网络信息中心（Internet Network Information Centre），称作InterNIC。InterNIC只分配网络号。主机号的分配由系统管理员来负责。

> Internet注册服务(IP地址和DNS域名)过去由NIC来负责，其网络地址是nic.ddn.mil。1993年4月1日，InterNIC成立。现在，NIC只负责处理国防数据网的注册请求，所有其他的Internet用户注册请求均由InterNIC负责处理，其网址是：rs.internic.net。
>
> 事实上InterNIC由三部分组成：注册服务（rs.internic.net），目录和数据库服务（ds.internic.net），以及信息服务（is.internic.net）。有关InterNIC的其他信息参见习题1.8。

有三类IP地址：单播地址（目的为单个主机）、广播地址（目的端为给定网络上的所有主机）以及多播地址（目的端为同一组内的所有主机）。第12章和第13章将分别讨论广播和多播的更多细节。

在3.4节中，我们在介绍IP选路以后将进一步介绍子网的概念。图3-9给出了几个特殊的IP地址：主机号和网络号为全0或全1。

1.5 域名系统

尽管通过IP地址可以识别主机上的网络接口，进而访问主机，但是人们最喜欢使用的还是主机名。在TCP/IP领域中，域名系统（DNS）是一个分布的数据库，由它来提供IP地址和主机名之间的映射信息。我们在第14章将详细讨论DNS。

现在，我们必须理解，任何应用程序都可以调用一个标准的库函数来查看给定名字的主机的IP地址。类似地，系统还提供一个逆函数——给定主机的IP地址，查看它所对应的主机名。

大多数使用主机名作为参数的应用程序也可以把IP地址作为参数。例如，在第4章中当我们用Telnet进行远程登录时，既可以指定一个主机名，也可以指定一个IP地址。

1.6 封装

当应用程序用TCP传送数据时，数据被送入协议栈中，然后逐个通过每一层直到被当作一串比特流送入网络。其中每一层对收到的数据都要增加一些首部信息（有时还要增加尾部信息），该过程如图1-7所示。TCP传给IP的数据单元称作TCP报文段或简称为TCP段（TCP segment）。IP传给网络接口层的数据单元称作IP数据报(IP datagram)。通过以太网传输的比特流称作帧(Frame)。

图1-7中帧头和帧尾下面所标注的数字是典型以太网帧首部的字节长度。在后面的章节中我们将详细讨论这些帧头的具体含义。

以太网数据帧的物理特性是其长度必须在 46～1500字节之间。我们将在4.5节遇到最小长度的数据帧，在2.8节中遇到最大长度的数据帧。

所有的Internet标准和大多数有关TCP/IP的书都使用octet这个术语来表示字节。使用这个过分雕琢的术语是有历史原因的，因为TCP/IP的很多工作都是在DEC-10系统上进行的，但是它并不使用8 bit的字节。由于现在几乎所有的计算机系统都采用8 bit的字节，因此我们在本书中使用字节（byte）这个术语。

更准确地说，图1-7中IP和网络接口层之间传送的数据单元应该是分组（packet）。分组既可以是一个IP数据报，也可以是IP数据报的一个片（fragment）。我们将在11.5节讨论IP数据报分片的详细情况。

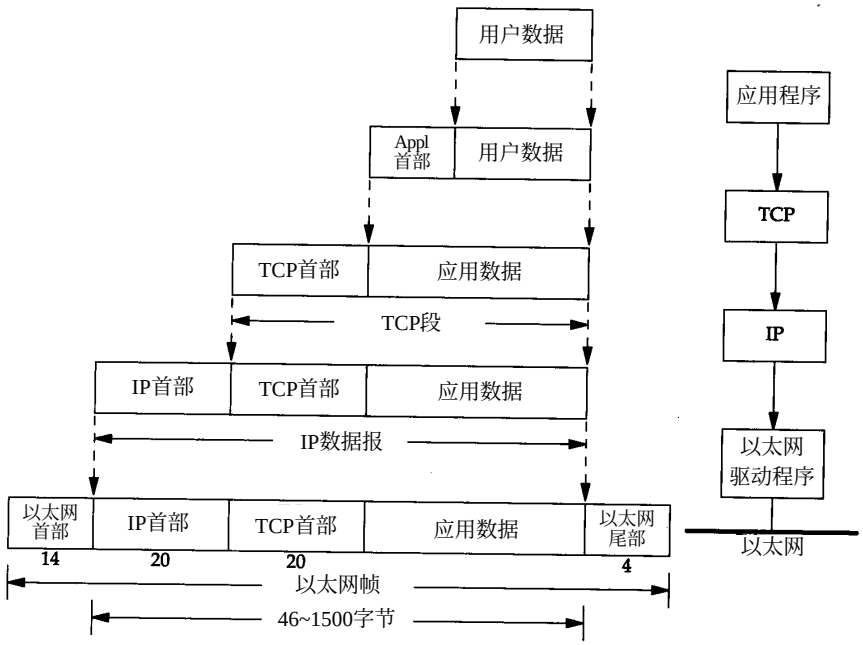


图1-7 数据进入协议栈时的封装过程

UDP数据与TCP数据基本一致。唯一的不同是 UDP传给IP的信息单元称作 UDP数据报（UDP datagram），而且UDP的首部长为8字节。

回想1.3节中的图1-4，由于TCP、UDP、ICMP和IGMP都要向IP传送数据，因此IP必须在生成的IP首部中加入某种标识，以表明数据属于哪一层。为此， IP在首部中存入一个长度为8bit的数值，称作协议域。 1表示为ICMP协议， 2表示为IGMP协议， 6表示为TCP协议， 17表示为UDP协议。

类似地，许多应用程序都可以使用 TCP或UDP来传送数据。运输层协议在生成报文首部时要存入一个应用程序的标识符。 TCP和UDP都用一个16bit的端口号来表示不同的应用程序。TCP和UDP把源端口号和目的端口号分别存入报文首部中。

网络接口分别要发送和接收IP、ARP和RARP数据，因此也必须在以太网的帧首部中加入

某种形式的标识，以指明生成数据的网络层协议。为此，以太网的帧首部也有一个 16 bit的帧类型域。

1.7 分用

当目的主机收到一个以太网数据帧时，数据就开始从协议栈中由底向上升，同时去掉各层协议加上的报文首部。每层协议盒都要去检查报文首部中的协议标识，以确定接收数据的上层协议。这个过程称作分用（Demultiplexing），图1-8显示了该过程是如何发生的。

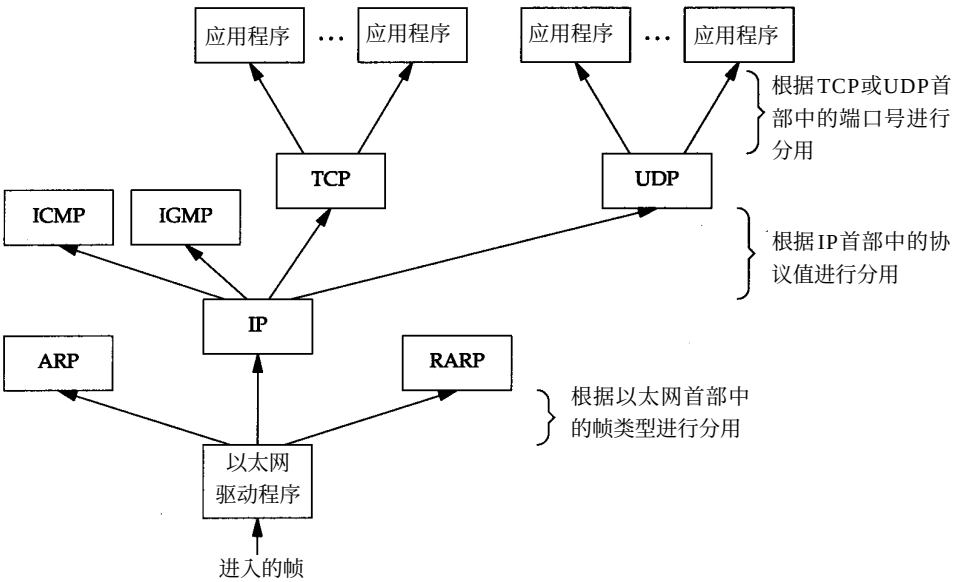


图1-8 以太网数据帧的分用过程

为协议ICMP和IGMP定位一直是一件很棘手的事情。在图1-4中，把它们与IP放在同一层上，那是因为事实上它们是IP的附属协议。但是在这里，我们又把它们放在IP层的上面，这是因为ICMP和IGMP报文都被封装在IP数据报中。

对于ARP和RARP，我们也遇到类似的难题。在这里把它们放在以太网设备驱动程序的上方，这是因为它们和IP数据报一样，都有各自的以太网数据帧类型。但在图2-4中，我们又把ARP作为以太网设备驱动程序的一部分，放在IP层的下面，其原因在逻辑上是合理的。

这些分层协议盒并不都是完美的。

当进一步描述TCP的细节时，我们将看到协议确实是通过目的端口号、源 IP地址和源端口号进行解包的。

1.8 客户-服务器模型

大部分网络应用程序在编写时都假设一端是客户，另一端是服务器，其目的是为了让服务器为客户提供一些特定的服务。

可以将这种服务分为两种类型：重复型或并发型。重复型服务器通过以下步骤进行交互：

I1. 等待一个客户请求的到来。

I2. 处理客户请求。

I3. 发送响应给发送请求的客户。

I4. 返回I1步。

重复型服务器主要的问题发生在I2状态。在这个时候，它不能为其他客户机提供服务。

相应地，并发型服务器采用以下步骤：

C1. 等待一个客户请求的到来。

C2. 启动一个新的服务器来处理这个客户的请求。在这期间可能生成一个新的进程、任务或线程，并依赖底层操作系统的支持。这个步骤如何进行取决于操作系统。生成的新服务器对客户的全部请求进行处理。处理结束后，终止这个新服务器。

C3. 返回C1步。

并发服务器的优点在于它是利用生成其他服务器的方法来处理客户的请求。也就是说，每个客户都有它自己对应的服务器。如果操作系统允许多任务，那么就可以同时为多个客户服务。

对服务器，而不是对客户进行分类的原因是因为对于一个客户来说，它通常并不能够辨别自己是与一个重复型服务器或并发型服务器进行对话。

一般来说，TCP服务器是并发的，而UDP服务器是重复的，但也存在一些例外。我们将在11.12节对UDP对其服务器产生的影响进行详细讨论，并在18.11节对TCP对其服务器的影响进行讨论。

1.9 端口号

前面已经指出过，TCP和UDP采用16 bit的端口号来识别应用程序。那么这些端口号是如何选择的呢？

服务器一般都是通过知名端口号来识别的。例如，对于每个TCP/IP实现来说，FTP服务器的TCP端口号都是21，每个Telnet服务器的TCP端口号都是23，每个TFTP(简单文件传送协议)服务器的UDP端口号都是69。任何TCP/IP实现所提供的服务都用知名的1～1023之间的端口号。这些知名端口号由Internet号分配机构（Internet Assigned Numbers Authority, IANA）来管理。

> 到1992年为止，知名端口号介于1～255之间。256～1023之间的端口号通常都是由Unix系统占用，以提供一些特定的Unix服务 —— 也就是说，提供一些只有Unix系统才有的、而其他操作系统可能不提供的服务。现在IANA管理1～1023之间所有的端口号。
>
> Internet扩展服务与Unix特定服务之间的一个差别就是Telnet和Rlogin。它们二者都允许通过计算机网络登录到其他主机上。Telnet是采用端口号为23的TCP/IP标准且几乎可以在所有操作系统上进行实现。相反，Rlogin最开始时只是为Unix系统设计的（尽管许多非Unix系统现在也提供该服务），因此在80年代初，它的有名端口号为513。

客户端通常对它所使用的端口号并不关心，只需保证该端口号在本机上是唯一的就可以了。客户端口号又称作临时端口号（即存在时间很短暂）。这是因为它通常只是在用户运行该客户程序时才存在，而服务器则只要主机开着的，其服务就运行。

大多数TCP/IP实现给临时端口分配1024～5000之间的端口号。大于5000的端口号是为其

他服务器预留的（Internet上并不常用的服务）。我们可以在后面看见许多这样的给临时端口分配端口号的例子。

Solaris 2.2是一个很有名的例外。通常TCP和UDP的缺省临时端口号从32768开始。在E.4节中，我们将详细描述系统管理员如何对配置选项进行修改以改变这些缺省项。

大多数Unix系统的文件/etc/services都包含了人们熟知的端口号。为了找到 Telnet服务器和域名系统的端口号，可以运行以下语句：

```
sun % grep telnet /etc/services
telnet     23/tcp     称它使用TCP端口号23
sun % grep domain /etc/services
domain     53/udp     称它使用UDP端口号53和TCP端口号53
domain     53/tcp
```

保留端口号

Unix系统有保留端口号的概念。只有具有超级用户特权的进程才允许给它自己分配一个保留端口号。

这些端口号介于1～1023之间，一些应用程序（如有名的 Rlogin，26.2节）将它作为客户与服务器之间身份认证的一部分。

1.10 标准化过程

究竟是谁控制着 TCP/IP协议族，又是谁在定义新的标准以及其他类似的事情？事实上，有四个小组在负责Internet技术。

1) Internet协会（ISOC，Internet Society）是一个推动、支持和促进Internet不断增长和发展的专业组织，它把Internet作为全球研究通信的基础设施。

2) Internet体系结构委员会（IAB，Internet Architecture Board）是一个技术监督和协调的机构。它由国际上来自不同专业的 15个志愿者组成，其职能是负责 Internet标准的最后编辑和技术审核。IAB隶属于ISOC。

3) Internet工程专门小组（IETF，Internet Engineering Task Force）是一个面向近期标准的组织，它分为9个领域（应用、寻径和寻址、安全等等）。IETF开发成为Internet标准的规范。为帮助IETF主席，又成立了Internet工程指导小组（IESG，Internet Engineering Steering Group）。

4) Internet研究专门小组（IRIF，Internet Research Task Force）主要对长远的项目进行研究。

IRTF和IETF都隶属于IAB。文献[Crocker 1993]提供了关于Internet内部标准化进程更为详细的信息，同时还介绍了它的早期历史。

1.11 RFC

所有关于Internet的正式标准都以RFC（Request for Comment）文档出版。另外，大量的RFC并不是正式的标准，出版的目的只是为了提供信息。 RFC的篇幅从1页到200页不等。每一项都用一个数字来标识，如 RFC 1122，数字越大说明RFC的内容越新。

所有的RFC都可以通过电子邮件或用 FTP从Internet上免费获取。如果发送下面这份电子邮件，就会收到一份获取RFC的方法清单：

```
To: rfc-info@ISI.EDU
Subject: getting rfcs
help: ways_to_get_rfcs
```

最新的RFC索引总是搜索信息的起点。这个索引列出了 RFC被替换或局部更新的时间。下面是一些重要的RFC文档：

1) 赋值RFC（Assigned Numbers RFC）列出了所有Internet协议中使用的数字和常数。至本书出版时为止，最新 RFC的编号是1340 [Reynolds和Postel 1992]。所有著名的Internet端口号都列在这里。

   当这个RFC被更新时(通常每年至少更新一次)，索引清单会列出RFC 1340被替换的时间。

2) Internet正式协议标准，目前是RFC 1600[Postel 1994]。这个RFC描述了各种Internet协议的标准化现状。每种协议都处于下面几种标准化状态之一：标准、草案标准、提议标准、实验标准、信息标准和历史标准。另外，对每种协议都有一个要求的层次、必需的、建议的、可选择的、限制使用的或者不推荐的。

   与赋值RFC一样，这个RFC也定期更新。请随时查看最新版本。

3) 主机需求RFC，1122和1123[Braden 1989a, 1989b]。RFC 1122针对链路层、网络层和运输层；RFC 1123针对应用层。这两个RFC对早期重要的RFC文档作了大量的纠正和解释。如果要查看有关协议更详细的细节内容，它们通常是一个入口点。它们列出了协议中关于“必须”、“应该”、“可以”、“不应该”或者“不能”等特性及其实现细节。

   文献[Borman 1993b]提供了有关这两个RFC的实用内容。RFC 1127[Braden 1989c]对工作小组开发主机需求RFC过程中的讨论内容和结论进行了非正式的总结。

4) 路由器需求RFC，目前正式版是RFC 1009[Braden and Postel 1987]，但一个新版已接近完成[Almquist 1993]。它与主机需求RFC类似，但是只单独描述了路由器的需求。

1.12 标准的简单服务

有一些标准的简单服务几乎每种实现都要提供。在本书中我们将使用其中的一些服务程序，而客户程序通常选择 Telnet。图1-9描述了这些服务。从该图可以看出，当使用 TCP和UDP提供相同的服务时，一般选择相同的端口号。

名 字	TCP端口号	UDP端口号	RFC	描 述
echo	7	7	862	服务器返回客户发送的所有内容
discard	9	9	863	服务器丢弃客户发送的所有内容
daytime	13	13	867	服务器以可读形式返回时间和日期
chargen	19	19	864	当客户发送一个数据报时，TCP服务器发送一串连续的字符流，直到客户中断连接。UDP服务器发送一个随机长度的数据报
time	37	37	868	服务器返回一个二进制形式的32 bit数，表示从UTC时间1900年1月1日午夜至今的秒数

图1-9 大多数实现都提供的标准的简单服务

如果仔细检查这些标准的简单服务以及其他标准的TCP/IP服务（如Telnet、FTP、SMTP等）的端口号时，我们发现它们都是奇数。这是有历史原因的，因为这些端口号都是从NCP端口号派生出来的（NCP，即网络控制协议，是ARPANET的运输层协议，是TCP的前身）。NCP是单工的，不是全双工的，因此每个应用程序需要两个连接，需预留一对奇数和偶数端口号。当TCP和UDP成为标准的运输层协议时，每个应用程序只需要一个端口号，因此就使用了NCP中的奇数。

1.13 互联网

在图1-3中，我们列举了一个由两个网络组成的互联网 —— 一个以太网和一个令牌环网。在1.4节和1.9节中，我们讨论了世界范围内的互联网 —— Internet，以及集中分配IP地址的需要（InterNIC），还讨论了知名端口号（IANA）。internet这个词第一个字母是否大写决定了它具有不同的含义。

internet意思是用一个共同的协议族把多个网络连接在一起。而 Internet指的是世界范围内通过TCP/IP互相通信的所有主机集合（超过100万台）。Internet是一个internet，但internet不等于Internet。

1.14 实现

既成事实标准的TCP/IP软件实现来自于位于伯克利的加利福尼亚大学的计算机系统研究小组。从历史上看，软件是随同4.x BSD系统（Berkeley Software Distribution）的网络版一起发布的。它的源代码是许多其他实现的基础。

图1-10列举了各种BSD版本发布的时间，并标注了重要的TCP/IP特性。列在左边的BSD网络版，其所有的网络源代码可以公开得到：包括协议本身以及许多应用程序和工具（如Telnet和FTP）。

在本书中，我们将使用“伯克利派生系统”来指SunOS 4.x、SVR4以及AIX 3.2等那些基于伯克利源代码开发的系统。这些系统有很多共同之处，经常包含相同的错误。

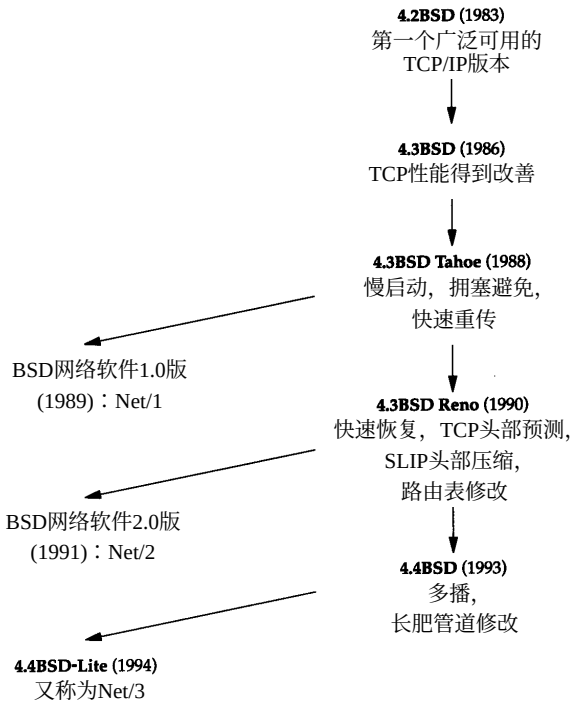


图1-10 不同的BSD版及其重要的TCP/IP特性

起初关于Internet的很多研究现在仍然在伯克利系统中应用 —— 新的拥塞控制算法（21.7节）、多播（12.4节）、“长肥管道”修改（24.3节）以及其他类似的研究。

1.15 应用编程接口

使用TCP/IP协议的应用程序通常采用两种应用编程接口（API）：socket和TLI（运输层接

口：Transport Layer Interface）。前者有时称作“ Berkeley socket ”，表明它是从伯克利版发展而来的。后者起初是由AT&T开发的，有时称作XTI（X/Open运输层接口），以承认X/Open这个自己定义标准的国际计算机生产商所做的工作。 XTI实际上是TLI的一个超集。

本书不是一本编程方面的书，但是偶尔会引用一些内容来说明 TCP/IP的特性，不管大多数的API（socket）是否提供它们。所有关于 socket和TLI的编程细节请参阅文献 [Stevens 1990]。

1.16 测试网络

图1-11是本书中所有的例子运行的测试网络。为阅读时参考方便，该图还复制在本书扉页前的插页中。

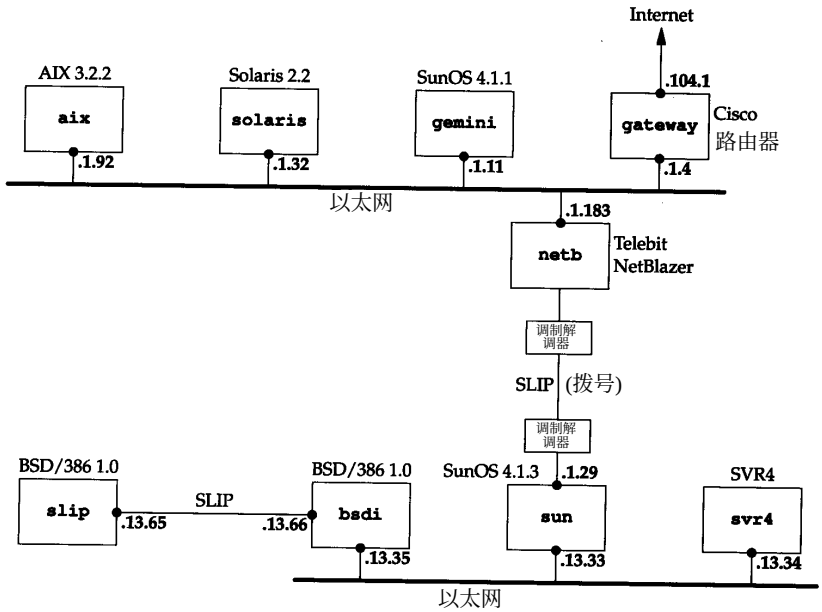


图1-11 本书中所有例子运行的测试网络，所有的IP地址均从140.252开始编址

在这个图中（作者的子网），大多数的例子都运行在下面四个系统中。图中所有的 IP地址属于B类地址，网络号为140.252。所有的主机名属于 `.tuc.noao.edu`这个域（`noao`代表National Optical Astronomy Observatories，`tuc`代表Tucson）。例如，右下方的系统有一个完整的名字：`svr4.tuc.noao.edu`，其IP地址是：140.252.13.34。每个方框上方的名称是该主机运行的操作系统。这一组系统和网络上的主机及路由器运行于不同的 TCP/IP实现。

需要指出的是，`noao.edu`这个域中的网络和主机要比图1-11中的多得多。这里列出来的只是本书中将要用到的系统。

在3.4节中，我们将描述这个网络所用到的子网形式。在 4.6节中将介绍`sun`与`netb`之间的拨号SLIP的有关细节。2.4节将详细讨论SLIP。

1.17 小结

本章快速地浏览了TCP/IP协议族，介绍了在后面的章节中将要详细讨论的许多术语和协议。

TCP/IP协议族分为四层：链路层、网络层、运输层和应用层，每一层各有不同的责任。在TCP/IP中，网络层和运输层之间的区别是最为关键的：网络层（ IP）提供点到点的服务，而运输层（TCP和UDP）提供端到端的服务。

一个互联网是网络的网络。构造互联网的共同基石是路由器，它们在 IP层把网络连在一起。第一个字母大写的Internet是指分布在世界各地的大型互联网，其中包括 1万多个网络和超过100万台主机。

在一个互联网上，每个接口都用 IP地址来标识，尽管用户习惯使用主机名而不是 IP地址。域名系统为主机名和 IP地址之间提供动态的映射。端口号用来标识互相通信的应用程序。服务器使用知名端口号，而客户使用临时设定的端口号。

习题

1.1 请计算最多有多少个A类、B类和C类网络号。

1.2 用匿名FTP（见27.3节）从主机`nic.merit.edu`上获取文件`nsfnet/statistics/history.netcount`。该文件包含在NSFNET网络上登记的国内和国外的网络数。画一坐标系，横坐标代表年，纵坐标代表网络总数的对数值。纵坐标的最大值是习题 1.1的结果。如果数据显示一个明显的趋势，请估计按照当前的编址体制推算，何时会用完所有的网络地址（3.10节讨论解决该难题的建议）。

1.3 获取一份主机需求RFC拷贝[Braden 1989a]，阅读有关应用于TCP/IP协议族每一层的稳健性原则。这个原则的参考对象是什么？

1.4 获取一份最新的赋值RFC拷贝。“ quote of the day ”协议的有名端口号是什么？哪个 RFC对该协议进行了定义？

1.5 如果你有一个接入TCP/IP互联网的主机帐号，它的主 IP地址是多少？这台主机是否接入了Internet？它是多接口主机吗？

1.6 获取一份RFC 1000的拷贝，了解RFC这个术语从何而来。

1.7 与Internet协会联系，`isoc@isoc.org`或者+1 703 648 9888，了解有关加入的情况。

1.8 用匿名FTP从主机`is.internic.net`处获取文件`about-internic/information-about-the-internic`。

第2章 链 路 层

2.1 引言

从图1-4中可以看出，在TCP/IP协议族中，链路层主要有三个目的：（1）为IP模块发送和接收IP数据报；（2）为ARP模块发送ARP请求和接收ARP应答；（3）为RARP发送RARP请求和接收RARP应答。TCP/IP支持多种不同的链路层协议，这取决于网络所使用的硬件，如以太网、令牌环网、FDDI（光纤分布式数据接口）及RS-232串行线路等。

在本章中，我们将详细讨论以太网链路层协议，两个串行接口链路层协议（SLIP和PPP），以及大多数实现都包含的环回（loopback）驱动程序。以太网和SLIP是本书中大多数例子使用的链路层。对MTU（最大传输单元）进行了介绍，这个概念在本书的后面章节中将多次遇到。我们还讨论了如何为串行线路选择MTU。

2.2 以太网和IEEE 802封装

以太网这个术语一般是指数字设备公司（Digital Equipment Corp.）、英特尔公司（Intel Corp.）和Xerox公司在1982年联合公布的一个标准。它是当今TCP/IP采用的主要的局域网技术。它采用一种称作CSMA/CD的媒体接入方法，其意思是带冲突检测的载波侦听多路接入（Carrier Sense, Multiple Access with Collision Detection）。它的速率为10 Mb/s，地址为48 bit。

几年后，IEEE（电子电气工程师协会）802委员会公布了一个稍有不同的标准集，其中802.3针对整个CSMA/CD网络，802.4针对令牌总线网络，802.5针对令牌环网络。这三者的共同特性由802.2标准来定义，那就是802网络共有的逻辑链路控制（LLC）。不幸的是，802.2和802.3定义了一个与以太网不同的帧格式。文献[Stallings 1987]对所有的IEEE 802标准进行了详细的介绍。

在TCP/IP世界中，以太网IP数据报的封装是在RFC 894[Hornig 1984]中定义的，IEEE 802网络的IP数据报封装是在RFC 1042[Postel and Reynolds 1988]中定义的。主机需求RFC要求每台Internet主机都与一个10 Mb/s的以太网电缆相连接：

1) 必须能发送和接收采用RFC 894（以太网）封装格式的分组。

2) 应该能接收与RFC 894混合的RFC 1042（IEEE 802）封装格式的分组。

3) 也许能够发送采用RFC 1042格式封装的分组。如果主机能同时发送两种类型的分组数据，那么发送的分组必须是可以设置的，而且默认条件下必须是RFC 894分组。

最常使用的封装格式是RFC 894定义的格式。图2-1显示了两种不同形式的封装格式。图中每个方框下面的数字是它们的字节长度。

两种帧格式都采用48 bit（6字节）的目的地址和源地址（802.3允许使用16 bit的地址，但一般是48 bit地址）。这就是我们在本书中所称的硬件地址。ARP和RARP协议（第4章和第5章）对32 bit的IP地址和48 bit的硬件地址进行映射。

接下来的2个字节在两种帧格式中互不相同。在802标准定义的帧格式中，长度字段是指

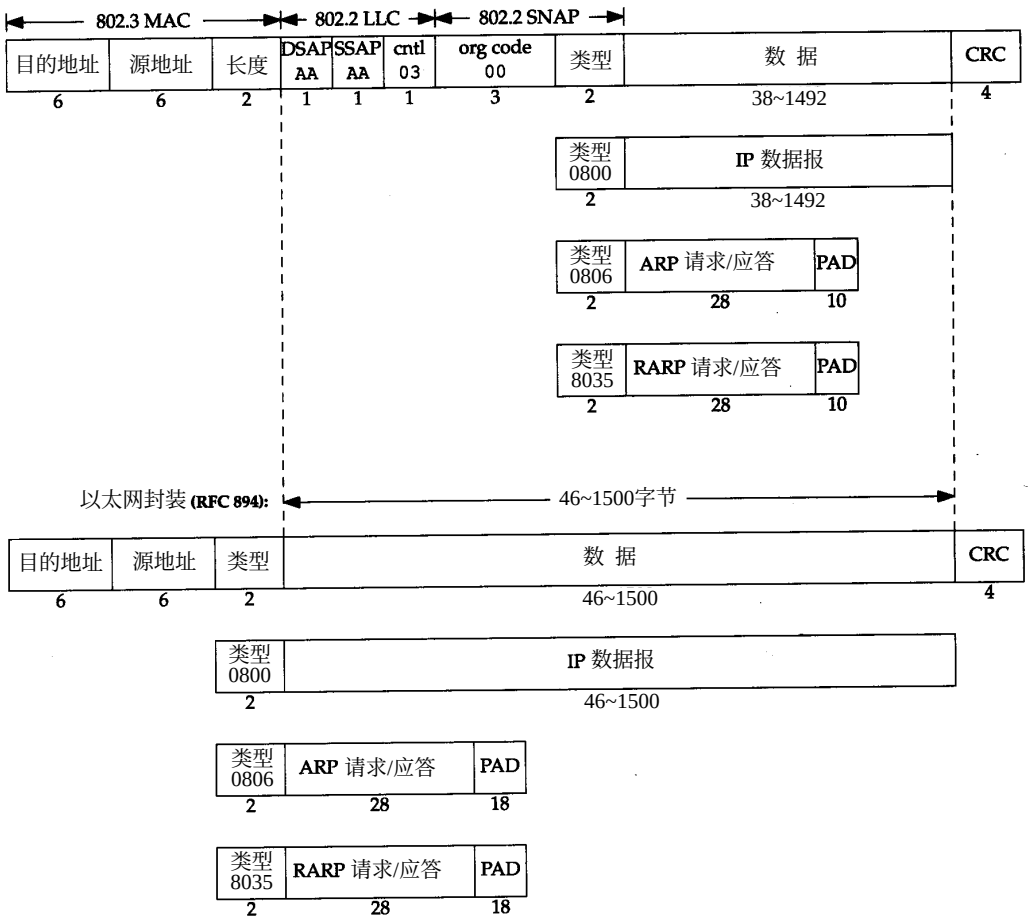


图2-1 IEEE 802.2/802.3（RFC 1042）和以太网的封装格式（RFC 894）

它后续数据的字节长度，但不包括 CRC检验码。以太网的类型字段定义了后续数据的类型。在802标准定义的帧格式中，类型字段则由后续的子网接入协议（ Sub-network Access Protocol，SNAP）的首部给出。幸运的是， 802定义的有效长度值与以太网的有效类型值无一相同，这样，就可以对两种帧格式进行区分。

在以太网帧格式中，类型字段之后就是数据；而在 802帧格式中，跟随在后面的是 3字节的802.2 LLC和5字节的802.2 SNAP。目的服务访问点（ Destination Service Access Point, DSAP）和源服务访问点（Source Service Access Point, SSAP）的值都设为0xaa。Ctrl字段的值设为3。随后的3个字节org code都置为0。再接下来的2个字节类型字段和以太网帧格式一样（其他类型字段值可以参见RFC 1340 [Reynolds and Postel 1992]）。

CRC字段用于帧内后续字节差错的循环冗余码检验（检验和）（它也被称为FCS或帧检验序列）。

802.3标准定义的帧和以太网的帧都有最小长度要求。 802.3规定数据部分必须至少为38字节，而对于以太网，则要求最少要有 46字节。为了保证这一点，必须在不足的空间插入填充（pad）字节。在开始观察线路上的分组时将遇到这种最小长度的情况。

在本书中，我们在需要的时候将给出以太网的封装格式，因为这是最为常见的封装格式。

2.3 尾部封装

RFC 893[Leffler and Karels 1984]描述了另一种用于以太网的封装格式，称作尾部封装（trailer encapsulation）。这是一个早期BSD系统在DEC VAX机上运行时的试验格式，它通过调整IP数据报中字段的次序来提高性能。在以太网数据帧中，开始的那部分是变长的字段（IP首部和TCP首部）。把它们移到尾部（在CRC之前），这样当把数据复制到内核时，就可以把数据帧中的数据部分映射到一个硬件页面，节省内存到内存的复制过程。 TCP数据报的长度是512字节的整数倍，正好可以用内核中的页表来处理。两台主机通过协商使用 ARP扩展协议对数据帧进行尾部封装。这些数据帧需定义不同的以太网帧类型值。

现在，尾部封装已遭到反对，因此我们不对它举任何例子。有兴趣的读者请参阅 RFC 893以及文献[Leffler et al. 1989]的11.8节。

2.4 SLIP：串行线路IP

SLIP的全称是Serial Line IP。它是一种在串行线路上对IP数据报进行封装的简单形式，在RFC 1055[Romkey 1988]中有详细描述。SLIP适用于家庭中每台计算机几乎都有的 RS-232串行端口和高速调制解调器接入Internet。

下面的规则描述了SLIP协议定义的帧格式：

1) IP数据报以一个称作END（0xc0）的特殊字符结束。同时，为了防止数据报到来之前的线路噪声被当成数据报内容，大多数实现在数据报的开始处也传一个 END字符（如果有线路噪声，那么END字符将结束这份错误的报文。这样当前的报文得以正确地传输，而前一个错误报文交给上层后，会发现其内容毫无意义而被丢弃）。

2) 如果IP报文中某个字符为 END，那么就要连续传输两个字节 0xdb和0xdc来取代它。0xdb这个特殊字符被称作SLIP的ESC字符，但是它的值与ASCII码的ESC字符（0x1b）不同。

3) 如果IP报文中某个字符为SLIP的ESC字符，那么就要连续传输两个字节 0xdb和0xdd来取代它。

图2-2中的例子就是含有一个END字符和一个ESC字符的IP报文。在这个例子中，在串行线路上传输的总字节数是原IP报文长度再加4个字节。

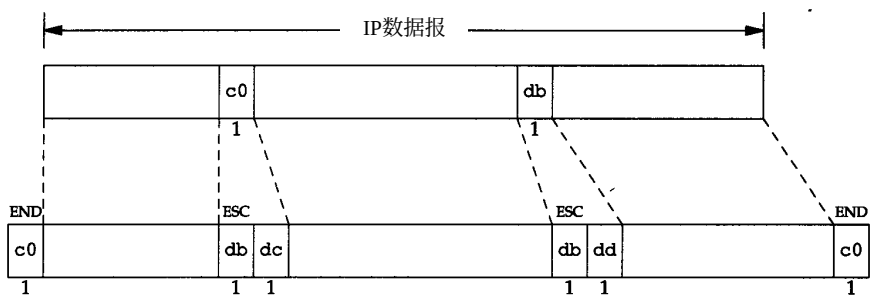


图2-2 SLIP报文的封装

SLIP是一种简单的帧封装方法，还有一些值得一提的缺陷：

1) 每一端必须知道对方的IP地址。没有办法把本端的IP地址通知给另一端。

2) 数据帧中没有类型字段（类似于以太网中的类型字段）。如果一条串行线路用于SLIP，那么它不能同时使用其他协议。

3) SLIP没有在数据帧中加上检验和（类似于以太网中的 CRC字段）。如果SLIP传输的报文被线路噪声影响而发生错误，只能通过上层协议来发现（另一种方法是，新型的调制解调器可以检测并纠正错误报文）。这样，上层协议提供某种形式的 CRC就显得很重要。在第 3章和第17章中，我们将看到IP首部和TCP首部及其数据始终都有检验和。在第 11章中，将看到UDP首部及其数据的检验和却是可选的。

尽管存在这些缺点， SLIP仍然是一种广泛使用的协议。

SLIP的历史要追溯到1984年，Rick Adams第一次在4.2BSD系统中实现。尽管它本身的描述是一种非标准的协议，但是随着调制解调器的速率和可靠性的提高， SLIP越来越流行。现在，它的许多产品可以公开获得，而且很多厂家都支持这种协议。

2.5 压缩的SLIP

由于串行线路的速率通常较低（ 19200 b/s或更低），而且通信经常是交互式的（如 Telnet和Rlogin，二者都使用TCP），因此在SLIP线路上有许多小的TCP分组进行交换。为了传送1个字节的数据需要20个字节的IP首部和20个字节的TCP首部，总数超过40个字节（19.2节描述了Rlogin会话过程中，当敲入一个简单命令时这些小报文传输的详细情况）。

既然承认这些性能上的缺陷，于是人们提出一个被称作 CSLIP（即压缩SLIP）的新协议，它在RFC 1144[Jacobson 1990a]中被详细描述。CSLIP一般能把上面的40个字节压缩到3或5个字节。它能在CSLIP的每一端维持多达16个TCP连接，并且知道其中每个连接的首部中的某些字段一般不会发生变化。对于那些发生变化的字段，大多数只是一些小的数字和的改变。这些被压缩的首部大大地缩短了交互响应时间。

现在大多数的SLIP产品都支持CSLIP。作者所在的子网（参见封面内页）中有两条SLIP链路，它们均是CSLIP链路。

2.6 PPP：点对点协议

PPP，点对点协议修改了SLIP协议中的所有缺陷。 PPP包括以下三个部分：

1) 在串行链路上封装 IP数据报的方法。 PPP既支持数据为 8位和无奇偶检验的异步模式（如大多数计算机上都普遍存在的串行接口），还支持面向比特的同步链接。

2) 建立、配置及测试数据链路的链路控制协议（ LCP：Link Control Protocol）。它允许通信双方进行协商，以确定不同的选项。

3) 针对不同网络层协议的网络控制协议（ NCP：Network Control Protocol）体系。当前RFC定义的网络层有IP、OSI网络层、DECnet以及AppleTalk。例如， IP NCP允许双方商定是否对报文首部进行压缩，类似于 CSLIP（缩写词NCP也可用在TCP的前面）。

RFC 1548[Simpson 1993]描述了报文封装的方法和链路控制协议。 RFC 1332[McGregor 1992]描述了针对IP的网络控制协议。

PPP数据帧的格式看上去很像 ISO的HDLC（高层数据链路控制）标准。图 2-3是PPP数据帧的格式。

每一帧都以标志字符0x7e开始和结束。紧接着是一个地址字节，值始终是 0xff，然后是一个值为0x03的控制字节。

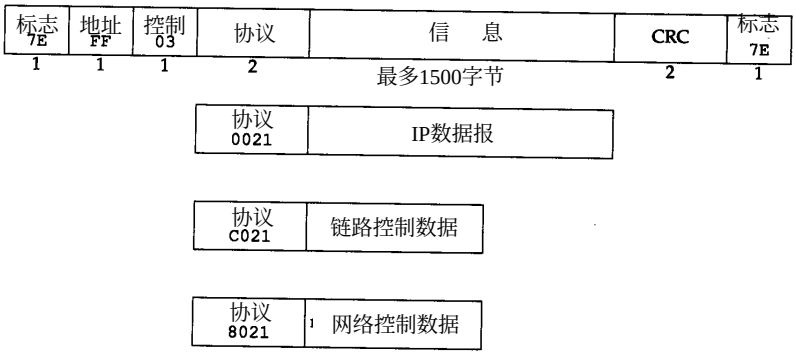


图2-3 PPP数据帧的格式

接下来是协议字段，类似于以太网中类型字段的功能。当它的值为 0x0021时，表示信息字段是一个IP数据报；值为0xc021时，表示信息字段是链路控制数据；值为 0x8021时，表示信息字段是网络控制数据。

CRC字段（或FCS，帧检验序列）是一个循环冗余检验码，以检测数据帧中的错误。

由于标志字符的值是 0x7e，因此当该字符出现在信息字段中时， PPP需要对它进行转义。在同步链路中，该过程是通过一种称作比特填充 (bit stuffing)的硬件技术来完成的[Tanenbaum 1989]。在异步链路中，特殊字符 0x7d用作转义字符。当它出现在 PPP数据帧中时，那么紧接着的字符的第6个比特要取其补码，具体实现过程如下：

1) 当遇到字符0x7e时，需连续传送两个字符：0x7d和0x5e，以实现标志字符的转义。

2) 当遇到转义字符0x7d时，需连续传送两个字符：0x7d和0x5d，以实现转义字符的转义。

3) 默认情况下，如果字符的值小于 0x20（比如，一个ASCII控制字符），一般都要进行转义。例如，遇到字符0x01时需连续传送0x7d和0x21两个字符（这时，第6个比特取补码后变为1，而前面两种情况均把它变为0）。

这样做的原因是防止它们出现在双方主机的串行接口驱动程序或调制解调器中，因为有时它们会把这些控制字符解释成特殊的含义。另一种可能是用链路控制协议来指定是否需要对这32个字符中的某一些值进行转义。默认情况下是对所有的 32个字符都进行转义。

与SLIP类似，由于PPP经常用于低速的串行链路，因此减少每一帧的字节数可以降低应用程序的交互时延。利用链路控制协议，大多数的产品通过协商可以省略标志符和地址字段，并且把协议字段由 2个字节减少到 1个字节。如果我们把 PPP的帧格式与前面的 SLIP的帧格式（图2-2）进行比较会发现， PPP只增加了 3个额外的字节： 1个字节留给协议字段，另 2个给CRC字段使用。另外，使用 IP网络控制协议，大多数的产品可以通过协商采用 Van Jacobson报文首部压缩方法（对应于CSLIP压缩），减小IP和TCP首部长度。

总的来说， PPP比SLIP具有下面这些优点：(1) PPP支持在单根串行线路上运行多种协议，不只是IP协议；(2) 每一帧都有循环冗余检验；(3) 通信双方可以进行 IP地址的动态协商(使用IP网络控制协议)；(4) 与CSLIP类似，对TCP和IP报文首部进行压缩；(5) 链路控制协议可以对多个数据链路选项进行设置。为这些优点付出的代价是在每一帧的首部增加 3个字节，当建立链路时要发送几帧协商数据，以及更为复杂的实现。

尽管PPP比SLIP有更多的优点，但是现在的SLIP用户仍然比PPP用户多。随着产品越来越多，产家也开始逐渐支持PPP，因此最终PPP应该取代SLIP。

2.7 环回接口

大多数的产品都支持环回接口（Loopback Interface），以允许运行在同一台主机上的客户程序和服务器程序通过TCP/IP进行通信。A类网络号127就是为环回接口预留的。根据惯例，大多数系统把IP地址127.0.0.1分配给这个接口，并命名为localhost。一个传给环回接口的IP数据报不能在任何网络上出现。

我们想象，一旦传输层检测到目的端地址是环回地址时，应该可以省略部分传输层和所有网络层的逻辑操作。但是大多数的产品还是照样完成传输层和网络层的所有过程，只是当IP数据报离开网络层时把它返回给自己。

图2-4是环回接口处理IP数据报的简单过程。

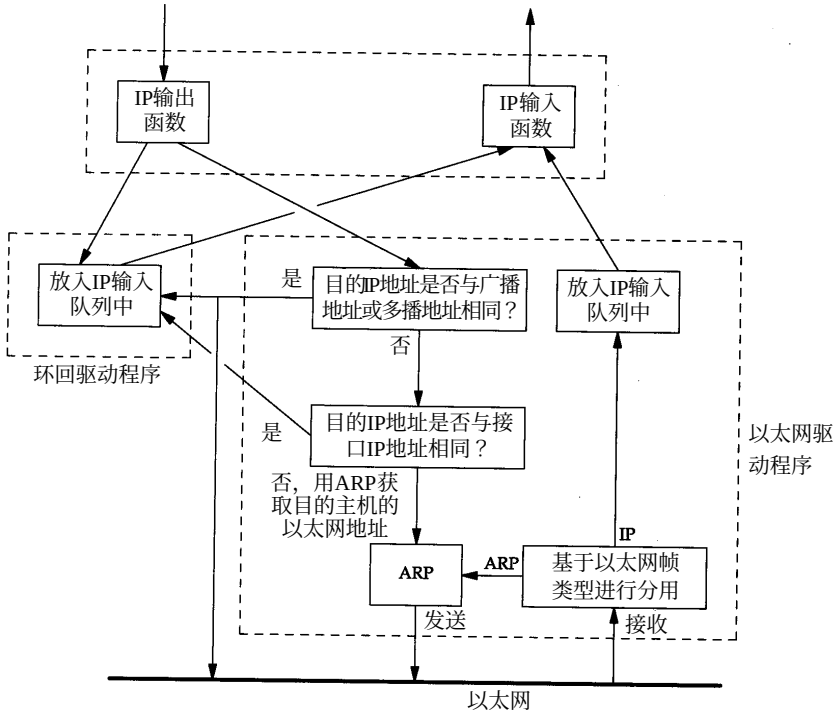


图2-4 环回接口处理IP数据报的过程

图中需要指出的关键点是：

1) 传给环回地址（一般是127.0.0.1）的任何数据均作为IP输入。

2) 传给广播地址或多播地址的数据报复制一份传给环回接口，然后送到以太网上。这是因为广播传送和多播传送的定义（第12章）包含主机本身。

3) 任何传给该主机IP地址的数据均送到环回接口。

看上去用传输层和IP层的方法来处理环回数据似乎效率不高，但它简化了设计，因为环回接口可以被看作是网络层下面的另一个链路层。网络层把一份数据报传送给环回接口，就像传给其他链路层一样，只不过环回接口把它返回到IP的输入队列中。

在图2-4中，另一个隐含的意思是送给主机本身IP地址的IP数据报一般不出现在相应的网络上。例如，在一个以太网上，分组一般不被传出去然后读回来。某些BSD以太网的设备驱动程序的注释说明，许多以太网接口卡不能读回它们自己发送出去的数据。由于一台主机必

须处理发送给自己的IP数据报，因此图2-4所示的过程是最为简单的处理办法。

4.4BSD系统定义了变量useloopback，并初始化为1。但是，如果这个变量置为0，以太网驱动程序就会把本地分组送到网络，而不是送到环回接口上。它也许不能工作，这取决于所使用的以太网接口卡和设备驱动程序。

2.8 最大传输单元MTU

正如在图2-1看到的那样，以太网和802.3对数据帧的长度都有一个限制，其最大值分别是1500和1492字节。链路层的这个特性称作MTU，最大传输单元。不同类型的网络大多数都有一个上限。

如果IP层有一个数据报要传，而且数据的长度比链路层的MTU还大，那么IP层就需要进行分片（fragmentation），把数据报分成若干片，这样每一片都小于MTU。我们将在11.5节讨论IP分片的过程。

网　　络	MTU字节
超通道	65535
16 Mb/s令牌环(IBM)	17914
4 Mb/s令牌环(IEEE 802.5)	4464
FDDI	4352
以太网	1500
IEEE 802.3/802.2	1492
X.25	576
点对点(低时延)	296

图2-5 几种常见的最大传输单元（MTU）

图2-5列出了一些典型的MTU值，它们摘自RFC 1191[Mogul and Deering 1990]。点到点的链路层（如SLIP和PPP）的MTU并非指的是网络媒体的物理特性。相反，它是一个逻辑限制，目的是为交互使用提供足够快的响应时间。在2.10节中，我们将看到这个限制值是如何计算出来的。

在3.9节中，我们将用netstat命令打印出网络接口的MTU。

2.9 路径MTU

当在同一个网络上的两台主机互相进行通信时，该网络的MTU是非常重要的。但是如果两台主机之间的通信要通过多个网络，那么每个网络的链路层就可能有不同的MTU。重要的不是两台主机所在网络的MTU的值，重要的是两台通信主机路径中的最小MTU。它被称作路径MTU。

两台主机之间的路径MTU不一定是个常数。它取决于当时所选择的路由。而选路不一定是对称的（从A到B的路由可能与从B到A的路由不同），因此路径MTU在两个方向上不一定是一致的。

RFC 1191[Mogul and Deering 1990]描述了路径MTU的发现机制，即在任何时候确定路径MTU的方法。我们在介绍了ICMP和IP分片方法以后再来看它是如何操作的。在11.6节中，我们将看到ICMP的不可到达错误就采用这种发现方法。在11.7节中，还会看到，traceroute程序也是用这个方法来确定到达目的节点的路径MTU。在11.8节和24.2节，将介绍当产品支持路径MTU的发现方法时，UDP和TCP是如何进行操作的。

2.10 串行线路吞吐量计算

如果线路速率是9600 b/s，而一个字节有8 bit，加上一个起始比特和一个停止比特，那么线路的速率就是960 B/s（字节/秒）。以这个速率传输一个1024字节的分组需要1066 ms。如果

用SLIP链接运行一个交互式应用程序，同时还运行另一个应用程序如 FTP发送或接收1024字节的数据，那么一般来说就必须等待一半的时间（ 533 ms）才能把交互式应用程序的分组数据发送出去。

假定交互分组数据可以在其他“ 大块 ”分组数据发送之前被发送出去。大多数的 SLIP实现确实提供这类服务排队方法，把交互数据放在大块的数据前面。交互通信一般有 Telnet、Rlogin以及FTP的控制部分（用户的命令，而不是数据）。

> 这种服务排队方法是不完善的。它不能影响已经进入下游（如串行驱动程序）队列的非交互数据。同时，新型的调制解调器具有很大的缓冲区，因此非交互数据可能已经进入该缓冲区了。

对于交互应用来说，等待 533 ms是不能接受的。关于人的有关研究表明，交互响应时间超过100～200 ms就被认为是不好的[Jacobson 1990a]。这是发送一份交互报文出去后，直到接收到响应信息（通常是出现一个回显字符）为止的往返时间。

把SLIP的MTU缩短到256就意味着链路传输一帧最长需要 266 ms，它的一半是133 ms（这是一般需要等待的时间）。这样情况会好一些，但仍然不完美。我们选择它的原因（与 64或128相比）是因为大块数据提供良好的线路利用率（如大文件传输）。假设CSLIP的报文首部是5个字节，数据帧总长为261个字节，256个字节的数据使线路的利用率为98.1%，帧头占了1.9%，这样的利用率是很不错的。如果把 MTU降到256以下，那么将降低传输大块数据的最大吞吐量。

在图2-5列出的MTU值中，点对点链路的MTU是296个字节。假设数据为256字节，TCP和IP首部占40个字节。由于MTU是IP向链路层查询的结果，因此该值必须包括通常的 TCP和IP首部。这样就会导致IP如何进行分片的决策。IP对于CSLIP的压缩情况一无所知。

我们对平均等待时间的计算（传输最大数据帧所需时间的一半）只适用于 SLIP链路（或PPP链路）在交互通信和大块数据传输这两种情况下。当只有交互通信时，如果线路速率是9600 b/s，那么任何方向上的1字节数据（假设有5个字节的压缩帧头）往返一次都大约需要12.5 ms。它比前面提到的100~200 ms要小得多。需要注意的是，由于帧头从40个字节压缩到5个字节，使得1字节数据往返时间从85 ms减到12.5 ms。

不幸的是，当使用新型的纠错和压缩调制解调器时，这样的计算就更难了。这些调制解调器所采用的压缩方法使得在线路上传输的字节数大大减少，但纠错机制又会增加传输的时间。不过，这些计算是我们进行合理决策的入口点。

在后面的章节中，我们将用这些串行线路吞吐量的计算来验证数据从串行线路上通过的时间。

2.11 小结

本章讨论了Internet协议族中的最底层协议，链路层协议。我们比较了以太网和 IEEE 802.2/802.3的封装格式，以及SLIP和PPP的封装格式。由于SLIP和PPP经常用于低速的链路，二者都提供了压缩不常变化的公共字段的方法。这使交互性能得到提高。

大多数的实现都提供环回接口。访问这个接口可以通过特殊的环回地址，一般为127.0.0.1。也可以通过发送IP数据报给主机所拥有的任一IP地址。当环回数据回到上层的协议栈中时，它已经过传输层和IP层完整的处理过程。

我们描述了很多链路都具有的一个重要特性， MTU，相关的一个概念是路径MTU。根据典型的串行线路MTU，对SLIP和CSLIP链路的传输时延进行了计算。

本章的内容只覆盖了当今TCP/IP所采用的部分数据链路公共技术。 TCP/IP成功的原因之一是它几乎能在任何数据链路技术上运行。

习题

2.1 如果你的系统支持`netstat`(1)命令（参见3.9节），那么请用它确定系统上的接口及其MTU。

第3章　IP：网际协议

3.1　引言

IP是TCP/IP协议族中最为核心的协议。所有的TCP、UDP、ICMP及IGMP数据都以IP数据报格式传输（见图1-4）。许多刚开始接触TCP/IP的人对IP提供不可靠、无连接的数据报传送服务感到很奇怪，特别是那些具有X.25或SNA背景知识的人。

不可靠（unreliable）的意思是它不能保证IP数据报能成功地到达目的地。IP仅提供最好的传输服务。如果发生某种错误时，如某个路由器暂时用完了缓冲区，IP有一个简单的错误处理算法：丢弃该数据报，然后发送ICMP消息报给信源端。任何要求的可靠性必须由上层来提供（如TCP）。

无连接（connectionless）这个术语的意思是IP并不维护任何关于后续数据报的状态信息。每个数据报的处理是相互独立的。这也说明，IP数据报可以不按发送顺序接收。如果一信源向相同的信宿发送两个连续的数据报（先是A，然后是B），每个数据报都是独立地进行路由选择，可能选择不同的路线，因此B可能在A到达之前先到达。

在本章，我们将简要介绍IP首部中的各个字段，讨论IP路由选择和子网的有关内容。还要介绍两个有用的命令：`ifconfig`和`netstat`。关于IP首部中一些字段的细节，将留在以后使用这些字段的时候再进行讨论。RFC 791[Postel 1981a]是IP的正式规范文件。

3.2　IP首部

IP数据报的格式如图3-1所示。普通的IP首部长为20个字节，除非含有选项字段。

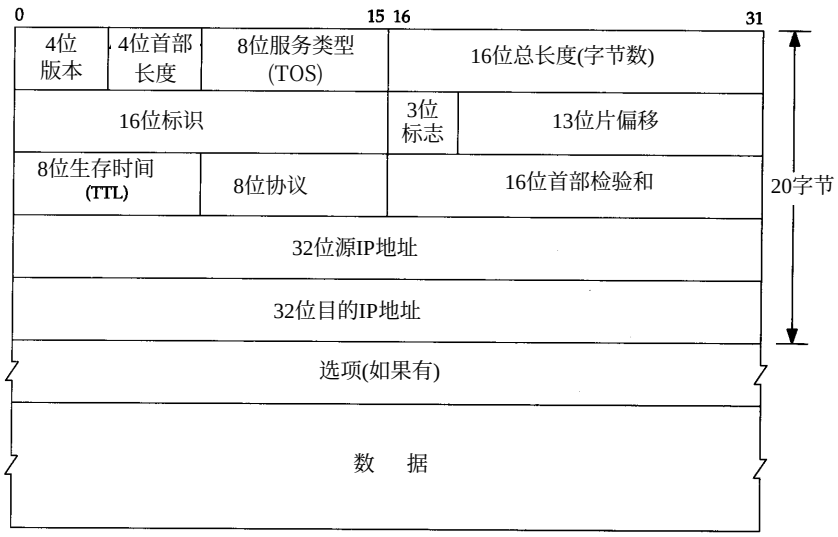


图3-1　IP数据报格式及首部中的各字段

分析图3-1中的首部。最高位在左边，记为0 bit；最低位在右边，记为31 bit。

4个字节的32 bit值以下面的次序传输：首先是0～7 bit，其次8～15 bit，然后16～23 bit，最后是24~31 bit。这种传输次序称作big endian字节序。由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。以其他形式存储二进制整数的机器，如little endian格式，则必须在传输数据之前把首部转换成网络字节序。

目前的协议版本号是4，因此IP有时也称作IPv4。3.10节将对一种新版的IP协议进行讨论。

首部长度指的是首部占32 bit字的数目，包括任何选项。由于它是一个 4比特字段，因此首部最长为60个字节。在第8章中，我们将看到这种限制使某些选项如路由记录选项在当今已没有什么用处。普通IP数据报（没有任何选择项）字段的值是5。

服务类型（TOS）字段包括一个3 bit的优先权子字段（现在已被忽略），4 bit的TOS子字段和1 bit未用位但必须置0。4 bit的TOS分别代表：最小时延、最大吞吐量、最高可靠性和最小费用。4 bit中只能置其中1 bit。如果所有4 bit均为0，那么就意味着是一般服务。RFC 1340 [Reynolds and Postel 1992]描述了所有的标准应用如何设置这些服务类型。 RFC 1349 [Almquist 1992]对该RFC进行了修正，更为详细地描述了TOS的特性。

图3-2列出了对不同应用建议的TOS值。在最后一列中给出的是十六进制值，因为这就是在后面将要看到的tcpdump命令输出。

应 用 程 序	最小时延	最大吞吐量	最高可靠性	最小费用	16进制值
Telnet/Rlogin	1	0	0	0	0x10
FTP					
控制	1	0	0	0	0x10
数据	0	1	0	0	0x08
任意块数据	0	1	0	0	0x08
TFTP	1	0	0	0	0x10
SMTP					
命令阶段	1	0	0	0	0x10
数据阶段	0	1	0	0	0x08
DNS					
UDP查询	1	0	0	0	0x10
TCP查询	0	0	0	0	0x00
区域传输	0	1	0	0	0x08
ICMP					
差错	0	0	0	0	0x00
查询	0	0	0	0	0x00
任何IGP	0	0	1	0	0x04
SNMP	0	0	1	0	0x04
BOOTP	0	0	0	0	0x00
NNTP	0	0	0	1	0x02

图3-2 服务类型字段推荐值

Telnet和Rlogin这两个交互应用要求最小的传输时延，因为人们主要用它们来传输少量的交互数据。另一方面， FTP文件传输则要求有最大的吞吐量。最高可靠性被指明给网络管理（SNMP）和路由选择协议。用户网络新闻（ Usenet news, NNTP）是唯一要求最小费用的应用。

现在大多数的TCP/IP实现都不支持TOS特性，但是自4.3BSD Reno以后的新版系统都对它进行了设置。另外，新的路由协议如OSPF和IS-IS都能根据这些字段的值进行路由决策。

在2.10节中，我们提到SLIP一般提供基于服务类型的排队方法，允许对交互通信

数据在处理大块数据之前进行处理。由于大多数的实现都不使用 TOS字段，因此这种排队机制由SLIP自己来判断和处理，驱动程序先查看协议字段（确定是否是一个 TCP段），然后检查TCP信源和信宿的端口号，以判断是否是一个交互服务。一个驱动程序的注释这样认为，这种“令人厌恶的处理方法”是必需的，因为大多数实现都不允许应用程序设置TOS字段。

总长度字段是指整个IP数据报的长度，以字节为单位。利用首部长度字段和总长度字段，就可以知道IP数据报中数据内容的起始位置和长度。由于该字段长 16比特，所以IP数据报最长可达65535字节（回忆图 2-5，超级通道的 MTU为65535。它的意思其实不是一个真正的MTU —— 它使用了最长的IP数据报）。当数据报被分片时，该字段的值也随着变化，这一点将在11.5节中进一步描述。

尽管可以传送一个长达 65535字节的IP数据报，但是大多数的链路层都会对它进行分片。而且，主机也要求不能接收超过 576字节的数据报。由于 TCP把用户数据分成若干片，因此一般来说这个限制不会影响 TCP。在后面的章节中将遇到大量使用 UDP的应用（RIP，TFTP，BOOTP，DNS，以及SNMP），它们都限制用户数据报长度为 512字节，小于576字节。但是，事实上现在大多数的实现（特别是那些支持网络文件系统 NFS的实现）允许超过 8192字节的IP数据报。

总长度字段是IP首部中必要的内容，因为一些数据链路（如以太网）需要填充一些数据以达到最小长度。尽管以太网的最小帧长为 46字节（见图2-1），但是IP数据可能会更短。如果没有总长度字段，那么IP层就不知道46字节中有多少是IP数据报的内容。

标识字段唯一地标识主机发送的每一份数据报。通常每发送一份报文它的值就会加 1。在11.5节介绍分片和重组时再详细讨论它。同样，在讨论分片时再来分析标志字段和片偏移字段。

RFC 791 [Postel 1981a]认为标识字段应该由让IP发送数据报的上层来选择。假设有两个连续的IP数据报，其中一个是由TCP生成的，而另一个是由UDP生成的，那么它们可能具有相同的标识字段。尽管这也可以照常工作（由重组算法来处理），但是在大多数从伯克利派生出来的系统中，每发送一个IP数据报，IP层都要把一个内核变量的值加1，不管交给IP的数据来自哪一层。内核变量的初始值根据系统引导时的时间来设置。

TTL（time-to-live）生存时间字段设置了数据报可以经过的最多路由器数。它指定了数据报的生存时间。TTL的初始值由源主机设置（通常为32或64），一旦经过一个处理它的路由器，它的值就减去1。当该字段的值为0时，数据报就被丢弃，并发送 ICMP报文通知源主机。第 8章我们讨论Traceroute程序时将再回来讨论该字段。

我们已经在第 1章讨论了协议字段，并在图 1-8中示出了它如何被 IP用来对数据报进行分用。根据它可以识别是哪个协议向 IP传送数据。

首部检验和字段是根据IP首部计算的检验和码。它不对首部后面的数据进行计算。 ICMP、IGMP、UDP和TCP在它们各自的首部中均含有同时覆盖首部和数据检验和码。

为了计算一份数据报的 IP检验和，首先把检验和字段置为 0。然后，对首部中每个 16 bit进行二进制反码求和（整个首部看成是由一串 16 bit的字组成），结果存在检验和字段中。当收到一份IP数据报后，同样对首部中每个 16 bit进行二进制反码的求和。由于接收方在计算过

程中包含了发送方存在首部中的检验和，因此，如果首部在传输过程中没有发生任何差错，那么接收方计算的结果应该为全1。如果结果不是全1（即检验和错误），那么IP就丢弃收到的数据报。但是不生成差错报文，由上层去发现丢失的数据报并进行重传。

ICMP、IGMP、UDP和TCP都采用相同的检验和算法，尽管TCP和UDP除了本身的首部和数据外，在IP首部中还包含不同的字段。在RFC 1071[Braden, Borman and Patridge 1988]中有关于如何计算Internet检验和的实现技术。由于路由器经常只修改TTL字段（减1），因此当路由器转发一份报文时可以增加它的检验和，而不需要对IP整个首部进行重新计算。RFC 1141[Mallory and Kullberg 1990]为此给出了一个很有效的方法。

但是，标准的BSD实现在转发数据报时并不是采用这种增加的办法。

每一份IP数据报都包含源IP地址和目的IP地址。我们在1.4节中说过，它们都是32 bit的值。

最后一个字段是任选项，是数据报中的一个可变长的可选信息。目前，这些任选项定义如下：

- 安全和处理限制（用于军事领域，详细内容参见RFC 1108[Kent 1991]）
- 记录路径（让每个路由器都记下它的IP地址，见7.3节）
- 时间戳（让每个路由器都记下它的IP地址和时间，见7.4节）
- 宽松的源站选路（为数据报指定一系列必须经过的IP地址，见8.5节）
- 严格的源站选路（与宽松的源站选路类似，但是要求只能经过指定的这些地址，不能经过其他的地址）。

这些选项很少被使用，并非所有的主机和路由器都支持这些选项。

选项字段一直都是以32 bit作为界限，在必要的时候插入值为0的填充字节。这样就保证IP首部始终是32 bit的整数倍（这是首部长度字段所要求的）。

3.3 IP路由选择

从概念上说，IP路由选择是简单的，特别对于主机来说。如果目的主机与源主机直接相连（如点对点链路）或都在一个共享网络上（以太网或令牌环网），那么IP数据报就直接送到目的主机上。否则，主机把数据报发往一默认的路由器上，由路由器来转发该数据报。大多数的主机都是采用这种简单机制。

在本节和第9章中，我们将讨论更一般的情况，即IP层既可以配置成路由器的功能，也可以配置成主机的功能。当今的大多数多用户系统，包括几乎所有的Unix系统，都可以配置成一个路由器。我们可以为它指定主机和路由器都可以使用的简单路由算法。本质上的区别在于主机从不把数据报从一个接口转发到另一个接口，而路由器则要转发数据报。内含路由器功能的主机应该从不转发数据报，除非它被设置成那样。在9.4小节中，我们将进一步讨论配置的有关问题。

在一般的体制中，IP可以从TCP、UDP、ICMP和IGMP接收数据报（即在本地生成的数据报）并进行发送，或者从一个网络接口接收数据报（待转发的数据报）并进行发送。IP层在内存中有一个路由表。当收到一份数据报并进行发送时，它都要对该表搜索一次。当数据报来自某个网络接口时，IP首先检查目的IP地址是否为本机的IP地址之一或者IP广播地址。如果确实是这样，数据报就被送到由IP首部协议字段所指定的协议模块进行处理。如果数据报的

目的不是这些地址，那么（1）如果IP层被设置为路由器的功能，那么就对数据报进行转发（也就是说，像下面对待发出的数据报一样处理）；否则（2）数据报被丢弃。

路由表中的每一项都包含下面这些信息：

- 目的IP地址。它既可以是一个完整的主机地址，也可以是一个网络地址，由该表目中的标志字段来指定（如下所述）。主机地址有一个非0的主机号（见图1-5），以指定某一特定的主机，而网络地址中的主机号为0，以指定网络中的所有主机（如以太网，令牌环网）。
- 下一站（或下一跳）路由器（next-hop router）的IP地址，或者有直接连接的网络IP地址。下一站路由器是指一个在直接相连网络上的路由器，通过它可以转发数据报。下一站路由器不是最终的目的，但是它可以把传送给它的数据报转发到最终目的。
- 标志。其中一个标志指明目的IP地址是网络地址还是主机地址，另一个标志指明下一站路由器是否为真正的下一站路由器，还是一个直接相连的接口（我们将在9.2节中详细介绍这些标志）。
- 为数据报的传输指定一个网络接口。

IP路由选择是逐跳地（hop-by-hop）进行的。从这个路由表信息可以看出，IP并不知道到达任何目的的完整路径（当然，除了那些与主机直接相连的目的）。所有的IP路由选择只为数据报传输提供下一站路由器的IP地址。它假定下一站路由器比发送数据报的主机更接近目的，而且下一站路由器与该主机是直接相连的。

IP路由选择主要完成以下这些功能：

1) 搜索路由表，寻找能与目的IP地址完全匹配的表目（网络号和主机号都要匹配）。如果找到，则把报文发送给该表目指定的下一站路由器或直接连接的网络接口（取决于标志字段的值）。
2) 搜索路由表，寻找能与目的网络号相匹配的表目。如果找到，则把报文发送给该表目指定的下一站路由器或直接连接的网络接口（取决于标志字段的值）。目的网络上的所有主机都可以通过这个表目来处置。例如，一个以太网上的所有主机都是通过这种表目进行寻径的。

   这种搜索网络的匹配方法必须考虑可能的子网掩码。关于这一点我们在下一节中进行讨论。
3) 搜索路由表，寻找标为“默认（default）”的表目。如果找到，则把报文发送给该表目指定的下一站路由器。

如果上面这些步骤都没有成功，那么该数据报就不能被传送。如果不能传送的数据报来自本机，那么一般会向生成数据报的应用程序返回一个“主机不可达”或“网络不可达”的错误。

完整主机地址匹配在网络号匹配之前执行。只有当它们都失败后才选择默认路由。默认路由，以及下一站路由器发送的ICMP间接报文（如果我们为数据报选择了错误的默认路由），是IP路由选择机制中功能强大的特性。我们在第9章对它们进行讨论。

为一个网络指定一个路由器，而不必为每个主机指定一个路由器，这是IP路由选择机制的另一个基本特性。这样做可以极大地缩小路由表的规模，比如Internet上的路由器有只有几千个表目，而不会是超过100万个表目。

举例

首先考虑一个简单的例子：我们的主机bsdi有一个IP数据报要发送给主机sun。双方都在

同一个以太网上（参见扉页前图）。数据报的传输过程如图3-3所示。

当IP从某个上层收到这份数据报后，它搜索路由表，发现目的IP地址（140.252.13.33）在一个直接相连的网络上（以太网140.252.13.0）。于是，在表中找到匹配网络地址（在下一节中，我们将看到，由于以太网的子网掩码的存在，实际的网络地址是140.252.13.32，但是这并不影响这里所讨论的路由选择）。

数据报被送到以太网驱动程序，然后作为一个以太网数据帧被送到sun主机上（见图2-1）。IP数据报中的目的地址是sun的IP地址（140.252.13.33），而在链路层首部中的目的地址是48 bit的sun主机的以太网接口地址。这个48 bit的以太网地址是用ARP协议获得的，我们将在下一章对此进行描述。

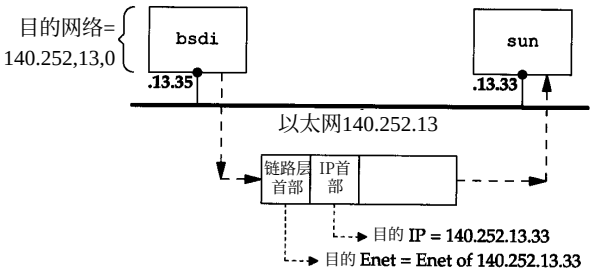


图3-3 数据报从主机bsdi到sun的传送过程

现在来看另一个例子：主机bsdi有一份IP数据报要传到ftp.uu.net主机上，它的IP地址是192.48.96.9。经过的前三个路由器如图3-4所示。首先，主机bsdi搜索路由表，但是没有找到与主机地址或网络地址相匹配的表目，因此只能用默认的表目，把数据报传给下一站路由器，即主机sun。当数据报从bsdi被传到sun主机上以后，目的IP地址是最终的信宿机地址（192.48.96.9），但是链路层地址却是sun主机的以太网接口地址。这与图3-3不同，在那里数据报中的目的IP地址和目的链路层地址都指的是相同的主机（sun）。

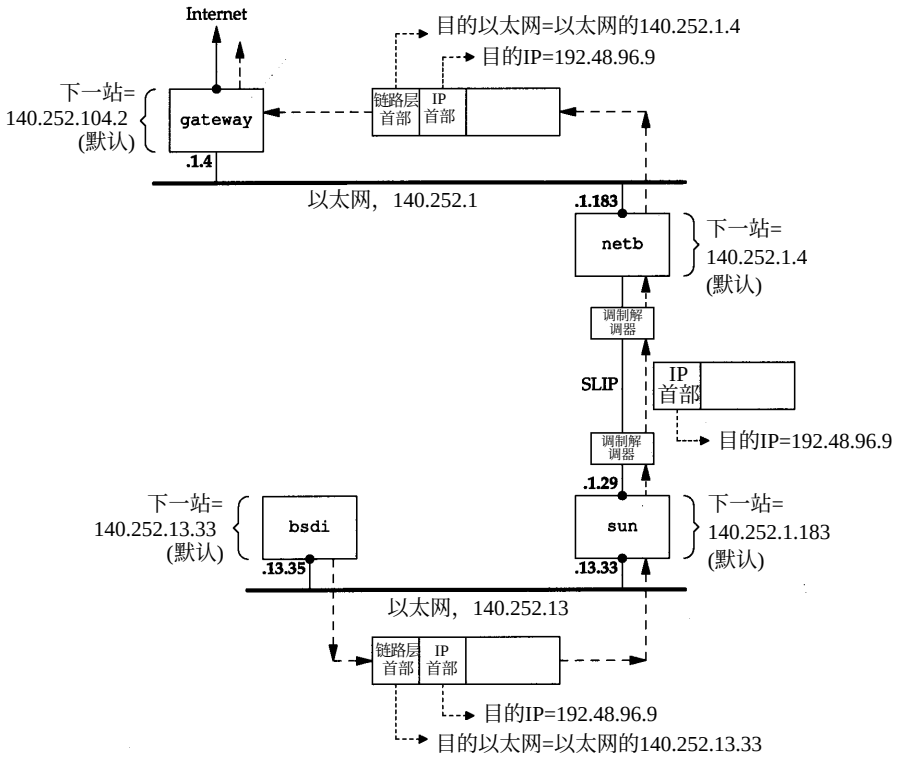


图3-4 从bsdi到ftp.uu.net (192.48.96.9)的初始路径

当sun收到数据报后，它发现数据报的目的IP地址并不是本机的任一地址，而sun已被设置成具有路由器的功能，因此它把数据报进行转发。经过搜索路由表，选用了默认表目。根据sun的默认表目，它把数据报转发到下一站路由器netb，该路由器的地址是140.252.1.183。数据报是经过点对点SLIP链路被传送的，采用了图2-2所示的最小封装格式。这里，我们没有给出像以太网链路层数据帧那样的首部，因为在SLIP链路中没有那样的首部。

当netb收到数据报后，它执行与sun主机相同的步骤：数据报的目的地址不是本机地址，而netb也被设置成具有路由器的功能，于是它也对该数据报进行转发。采用的也是默认路由表目，把数据报送到下一站路由器gateway（140.252.1.4）。位于以太网140.252.1上的主机netb用ARP获得对应于140.252.1.4的48 bit以太网地址。这个以太网地址就是链路层数据帧头上的目的地址。

路由器gateway也执行与前面两个路由器相同的步骤。它的默认路由表目所指定的下一站路由器IP地址是140.252.104.2（我们将在图8-4中证实，使用Traceroute程序时，它就是gateway使用的下一站路由器）。

对于这个例子需要指出一些关键点：

1) 该例子中的所有主机和路由器都使用了默认路由。事实上，大多数主机和一些路由器可以用默认路由来处理任何目的，除非它在本地局域网上。

2) 数据报中的目的IP地址始终不发生任何变化（在8.5节中，我们将看到，只有使用源路由选项时，目的IP地址才有可能被修改，但这种情况很少出现）。所有的路由选择决策都是基于这个目的IP地址。

3) 每个链路层可能具有不同的数据帧首部，而且链路层的目的地址（如果有的话）始终指的是下一站的链路层地址。在例子中，两个以太网封装了含有下一站以太网地址的链路层首部，但是SLIP链路没有这样做。以太网地址一般通过ARP获得。

在第9章，我们在描述了ICMP之后将再次讨论IP路由选择问题。我们将看到一些路由表的例子，以及如何用它们来进行路由决策的。

3.4 子网寻址

现在所有的主机都要求支持子网编址（RFC 950 [Mogul and Postel 1985]）。不是把IP地址看成由单纯的一个网络号和一个主机号组成，而是把主机号再分成一个子网号和一个主机号。

这样做的原因是因为A类和B类地址为主机号分配了太多的空间，可分别容纳的主机数为$2^{24}-2$和$2^{16}-2$。事实上，在一个网络中人们并不安排这么多的主机（各类IP地址的格式如图1-5所示）。由于全0或全1的主机号都是无效的，因此我们把总数减去2。

在InterNIC获得某类IP网络号后，就由当地的系统管理员来进行分配，由他（或她）来决定是否建立子网，以及分配多少比特给子网号和主机号。例如，这里有一个B类网络地址（140.252），在剩下的16 bit中，8 bit用于子网号，8 bit用于主机号，格式如图3-5所示。这样就允许有254个子网，每个子网可以有254台主机。

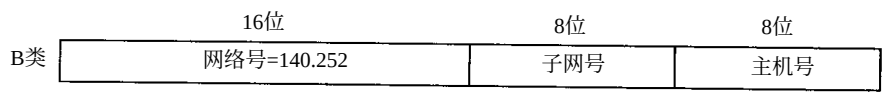


图3-5 B类地址的一种子网编址

许多管理员采用自然的划分方法，即把 B类地址中留给主机的 16 bit中的前8 bit作为子网地址，后8bit作为主机号。这样用点分十进制方法表示的 IP地址就可以比较容易确定子网号。但是，并不要求A类或B类地址的子网划分都要以字节为划分界限。

大多数的子网例子都是 B类地址。其实，子网还可用于 C类地址，只是它可用的比特数较少而已。很少出现 A类地址的子网例子是因为 A类地址本身就很少（但是，大多数 A类地址都是进行子网划分的）。

子网对外部路由器来说隐藏了内部网络组织（一个校园或公司内部）的细节。在我们的网络例子中，所有的IP地址都有一个B类网络号140.252。但是其中有超过30个子网，多于400台主机分布在这些子网中。由一台路由器提供了 Internet的接入，如图3-6所示。

在这个图中，我们把大多数的路由器编号为 R*n*，*n*是子网号。我们给出了连接这些子网的路由器，同时还包括了扉页前图中的九个系统。在图中，以太网用粗线表示，点对点链路用虚线表示。我们没有画出不同子网中的所有主机。例如，在子网 140.252.3上，就超过50台主机，而在子网140.252.1上则超过100台主机。

与30个C类地址相比，用一个包含30个子网的B类地址的好处是，它可以缩小 Internet路由表的规模。B类地址140.252被划分为若干子网的事实对于所有子网以外的 Internet路由器都是透明的。为了到达 IP地址开始部分为 140.252的主机，外部路由器只需要知道通往 IP地址140.252.104.1的路径。这就是说，对于网络 140.252只需一个路由表目，而如果采用 30个C类地址，则需要30个路由表目。因此，子网划分缩减了路由表的规模（在 10.8小节中，我们将介绍一种新技术，即使用C类地址也可以缩减路由表的规模）。

子网对于子网内部的路由器是不透明的。如图 3-6所示，一份来自 Internet的数据报到达 `gateway`，它的目的地址是140.252.57.1。路由器`gateway`需要知道子网号是57，然后把它送到`kpno`。同样，`kpno`必须把数据报送到R55，最后由R55把它送到R57。

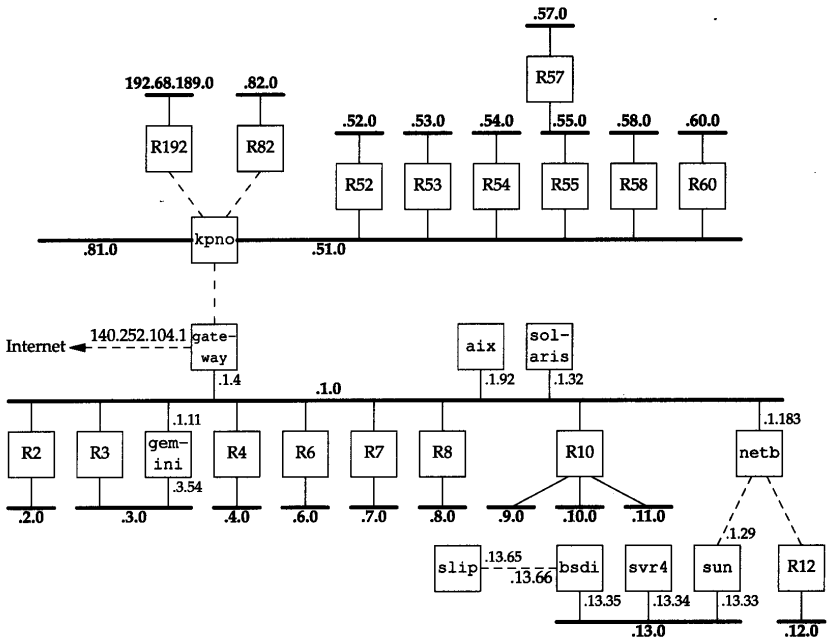


图3-6 网络noao.edu（140.252）中的大多数子网安排

3.5 子网掩码

任何主机在引导时进行的部分配置是指定主机 IP地址。大多数系统把IP地址存在一个磁盘文件里供引导时读用。在第5章我们将讨论一个无盘系统如何在引导时获得IP地址。

除了IP地址以外，主机还需要知道有多少比特用于子网号及多少比特用于主机号。这是在引导过程中通过子网掩码来确定的。这个掩码是一个 32 bit的值，其中值为1的比特留给网络号和子网号，为0的比特留给主机号。图3-7是一个B类地址的两种不同的子网掩码格式。第一个例子是noao.edu网络采用的子网划分方法，如图3-5所示，子网号和主机号都是8 bit宽。第二个例子是一个B类地址划分成10 bit的子网号和6 bit的主机号。

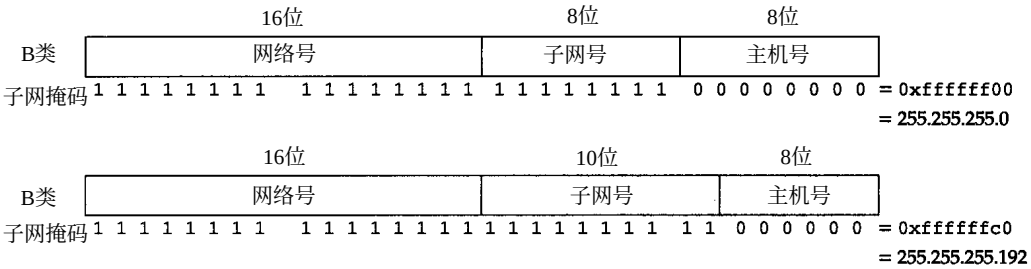


图3-7 两种不同的B类地址子网掩码的例子

尽管IP地址一般以点分十进制方法表示，但是子网掩码却经常用十六进制来表示，特别是当界限不是一个字节时，因为子网掩码是一个比特掩码。

给定IP地址和子网掩码以后，主机就可以确定 IP数据报的目的是：(1)本子网上的主机；(2)本网络中其他子网中的主机；(3)其他网络上的主机。如果知道本机的IP地址，那么就知道它是否为A类、B类或C类地址(从IP地址的高位可以得知)，也就知道网络号和子网号之间的分界线。而根据子网掩码就可知道子网号与主机号之间的分界线。

举例

假设我们的主机地址是140.252.1.1（一个B类地址），而子网掩码为255.255.255.0（其中8 bit为子网号，8 bit为主机号）。

- 如果目的IP地址是140.252.4.5，那么我们就知道B类网络号是相同的（140.252），但是子网号是不同的（1和4）。用子网掩码在两个IP地址之间的比较如图3-8所示。
- 如果目的IP地址是140.252.1.22，那么B类网络号还是一样的（140.252），而且子网号也是一样的（1），但是主机号是不同的。
- 如果目的IP地址是192.43.235.6（一个C类地址），那么网络号是不同的，因而进一步的比较就不用再进行了。

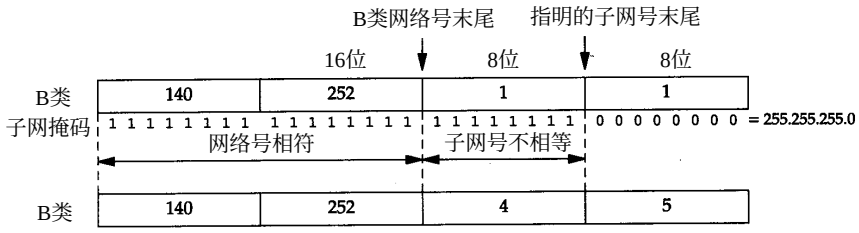


图3-8 使用子网掩码的两个B类地址之间的比较

给定两个IP地址和子网掩码后，IP路由选择功能一直进行这样的比较。

3.6 特殊情况的IP地址

经过子网划分的描述，现在介绍7个特殊的IP地址，如图3-9所示。在这个图中，0表示所有的比特位全为0；-1表示所有的比特位全为1；netid、subnetid和hostid分别表示不为全0或全1的对应字段。子网号栏为空表示该地址没有进行子网划分。

IP 地 址			可 以 为		描 述
网络号	子网号	主机号	源 端	目的端	
0		0	OK	不可能	网络上的主机（参见下面的限制）
0		主机号	OK	不可能	网络上的特定主机（参见下面的限制）
127		任何值	OK	OK	环回地址（2.7节）
-1		-1	不可能	OK	受限的广播（永远不被转发）
netid		-1	不可能	OK	以网络为目的向*netid*广播
netid	*subnetid*	-1	不可能	OK	以子网为目的向*netid*、*subnetid*广播
netid	-1	-1	不可能	OK	以所有子网为目的向*netid*广播

图3-9 特殊情况的IP地址

我们把这个表分成三个部分。表的头两项是特殊的源地址，中间项是特殊的环回地址，最后四项是广播地址。

表中的头两项，网络号为0，如主机使用BOOTP协议确定本机IP地址时只能作为初始化过程中的源地址出现。

在12.2节中，我们将进一步分析四类广播地址。

3.7 一个子网的例子

这个例子是本文中采用的子网，以及如何使用两个不同的子网掩码。具体安排如图 3-10所示。

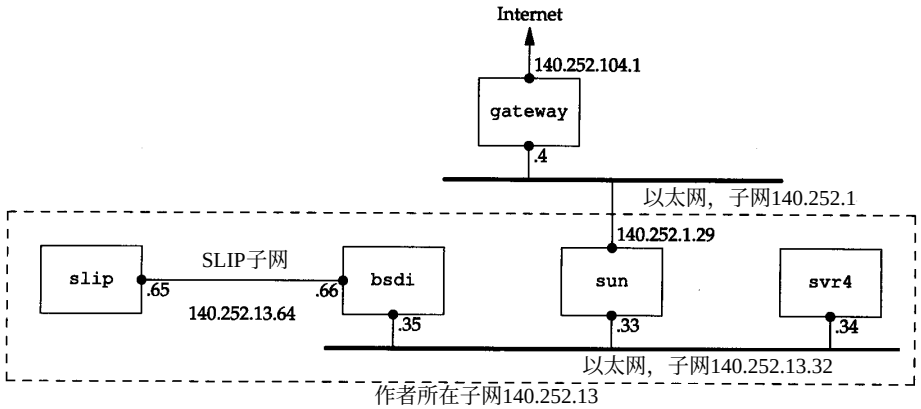


图3-10 作者所在子网中的主机和网络安排

如果把该图与扉页前图相比，就会发现在图 3-10中省略了从路由器sun到上面的以太网之间的连接细节，实际上它们之间的连接是拨号 SLIP。这个细节不影响本节中讨论的子网划分

问题。我们在4.6节讨论ARP代理时将再回头讨论这个细节。

问题是我们在子网13中有两个分离的网络：一个以太网和一个点对点链路（硬件连接的SLIP链路）（点对点链接始终会带来问题，因为它一般在两端都需要IP地址）。将来或许会有更多的主机和网络，但是为了不让主机跨越不同的网络就得使用不同的子网号。我们的解决方法是把子网号从8 bit扩充到11bit，把主机号从8 bit减为5 bit。这就叫作变长子网，因为140.252网络中的大多数子网都采用8 bit子网掩码，而我们的子网却采用11 bit的子网掩码。

RFC 1009[Braden and Postel 1987]允许一个含有子网的网络使用多个子网掩码。新的路由器需求RFC[Almquist 1993]则要求支持这一功能。

但是，问题在于并不是所有的路由选择协议在交换目的网络时也交换子网掩码。在第10章中，我们将看到RIP不支持变长子网，RIP第2版和OSPF则支持变长子网。在我们的例子中不存在这种问题，因为在我的子网中不要求使用RIP协议。

作者子网中的IP地址结构如图3-11所示，11位子网号中的前8 bit始终是13。在剩下的3 bit中，我们用二进制001表示以太网，010表示点对点SLIP链路。这个变长子网掩码在140.252网络中不会给其他主机和路由器带来问题——只要目的是子网140.252.13的所有数据报都传给路由器sun（IP地址是140.252.1.29），如图3-11所示。如果sun知道子网13中的主机有11 bit子网号，那么一切都好办了。

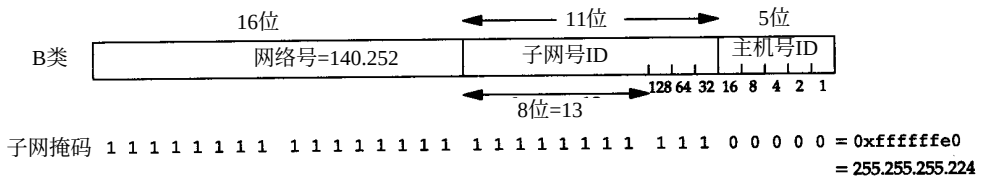


图3-11 变长子网

140.252.13子网中的所有接口的子网掩码是255.255.255.224，或0xfffffffe0。这表明最右边的5 bit留给主机号，左边的27 bit留给网络号和子网号。

图3-10中所有接口的IP地址和子网掩码的分配情况如图3-12所示。

主机	IP地址	子网掩码	网络号/子网号	主机号	注　　释
sun	140.252.1.29	255.255.255.0	140.252.1	29	在子网1上
	140.252.13.33	255.255.255.224	140.252.13.32	1	在作者所在子网上
svr4	140.252.13.34	255.255.255.224	140.252.13.32	2	
bsdi	140.252.13.35	255.255.255.224	140.252.13.32	3	在以太网上
	140.252.13.66	255.255.255.224	140.252.13.64	2	点对点
slip	140.252.13.65	255.255.255.224	140.252.13.64	1	点对点
	140.252.13.63	255.255.255.224	140.252.13.32	31	以太网上的广播地址

图3-12 作者子网的IP地址

第1栏标为是“主机”，但是sun和bsdi也具有路由器的功能，因为它们是多接口的，可以把分组数据从一个接口转发到另一个接口。

这个表中的最后一行是图3-10中的广播地址140.252.13.63：它是根据以太网子网号（140.252.13.32）和图3-11中的低5位置1（16＋8＋4＋2＋1＝31）得来的（我们在第12章中将看到，这个地址被称作以子网为目的的广播地址（subnet-directed broadcast address））。

3.8 ifconfig命令

到目前为止，我们已经讨论了链路层和IP层，现在可以介绍TCP/IP对网络接口进行配置和查询的命令了。ifconfig(8)命令一般在引导时运行，以配置主机上的每个接口。

由于拨号接口可能会经常接通和挂断（如SLIP链路），每次线路接通和挂断时，ifconfig都必须（以某种方法）运行。这个过程如何完成取决于使用的SLIP软件。

下面是作者子网接口的有关参数。请把它们与图3-12的值进行比较。

```
sun % /usr/etc/ifconfig -a              在所有接口报告的选项
le0: flags=63<UP,BROADCAST,NOTRAILERS,RUNNING>
        inet 140.252.13.33 netmask fffffe0 broadcast 140.252.13.63
sl0: flags=1051<UP,POINTOPOINT,RUNNING,LINK0>
        inet 140.252.1.29 --> 140.252.1.183 netmask ffffff00
lo0: flags=49<UP,LOOPBACK,RUNNING>
        inet 127.0.0.1 netmask ff000000
```

环回接口（2.7节）被认为是一个网络接口。它是一个A类地址，没有进行子网划分。

需要注意的是以太网没有采用尾部封装（2.3节），而且可以进行广播，而SLIP链路是一个点对点的链接。

SLIP接口的标志LINK0是一个允许压缩slip的数据（CSLIP，参见2.5节）的配置选项。其他的选项有LINK1（如果从另一端收到一份压缩报文，就允许采用CSLIP）和LINK2（所有外出的ICMP报文都被丢弃）。我们在4.6节中将讨论SLIP链接的目的地址。

> 安装指南中的注释对最后这个选项进行了解释：“一般它不应设置，但是由于一些不当的ping操作，可能会导致吞吐量降到0。”

bsdi是另一台路由器。由于-a参数是SunOS操作系统具有的功能，因此我们必须多次执行ifconfig，并指定接口名字参数：

```
bsdi % /sbin/ifconfig we0
we0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,SIMPLEX>
        inet 140.252.13.35 netmask fffffe0 broadcast 140.252.13.63
bsdi % /sbin/ifconfig sl0
sl0: flags=1011<UP,POINTOPOINT,LINK0>
        inet 140.252.13.66 --> 140.252.13.65 netmask fffffe0
```

这里，我们看到以太网接口（we0）的一个新选项：SIMPLEX。这个4.4BSD标志表明接口不能收到本机传送的数据。在BSD/386中所有的以太网都这样设置。一旦这样设置后，如果接口发送一帧数据到广播地址，那么就会为本机拷贝一份数据送到环回地址（在6.3小节我们将举例说明这一点）。

在主机slip中，SLIP接口的设置基本上与上面的bsdi一致，只是两端的IP地址进行了互换：

```
slip % /sbin/ifconfig sl0
sl0: flags=1011<UP,POINTOPOINT,LINK0>
        inet 140.252.13.65 --> 140.252.13.66 netmask fffffe0
```

最后一个接口是主机svr4上的以太网接口。它与前面的以太网接口类似，只是SVR4版的ifconfig没有打印RUNNING标志：

```
svr4 % /usr/sbin/ifconfig emd0
emd0: flags=23<UP,BROADCAST,NOTRAILERS>
        inet 140.252.13.34 netmask fffffe0 broadcast 140.252.13.63
```

ifconfig命令一般支持TCP/IP以外的其他协议族，而且有很多参数。关于这些细节可以查看系统说明书。

3.9 netstat命令

netstat(1)命令也提供系统上的接口信息。-i参数将打印出接口信息，-n参数则打印出IP地址，而不是主机名字。

```
sun % netstat -in
Name  Mtu  Net/Dest      Address         Ipkts  Ierrs Opkts  Oerrs Collis Queue
le0   1500 140.252.13.32 140.252.13.33  67719   0     92133   0     1      0
sl0   552  140.252.1.183 140.252.1.29   48035   0     54963   0     0      0
lo0   1536 127.0.0.0     127.0.0.1       15548   0     15548   0     0      0
```

这个命令打印出每个接口的MTU、输入分组数、输入错误、输出分组数、输出错误、冲突以及当前的输出队列长度。

在第9章将用netstat命令检查路由表，那时再回头讨论该命令。另外，在第13章将用它的一个改进版本来查看活动的广播组。

3.10 IP的未来

IP主要存在三个方面的问题。这是Internet在过去几年快速增长所造成的结果（参见习题1.2）。

1) 超过半数的B类地址已被分配。根据估计，它们大约在1995年耗尽。

2) 32 bit的IP地址从长期的Internet增长角度来看，一般是不够用的。

3) 当前的路由结构没有层次结构，属于平面型(flat)结构，每个网络都需要一个路由表目。随着网络数目的增长，一个具有多个网络的网站就必须分配多个C类地址，而不是一个B类地址，因此路由表的规模会不断增长。

无类别的域间路由选择CIDR（Classless Interdomain Routing）提出了一个可以解决第三个问题的建议，对当前版本的IP（IP版本4）进行扩充，以适应21世纪Internet的发展。对此我们将在10.8节进一步详细介绍。

对新版的IP，即下一代IP，经常称作IPng，主要有四个方面的建议。1993年5月发行的IEEE *Network* (vol.7, no.3)对前三个建议进行了综述，同时有一篇关于CIDR的论文。RFC 1454 [Dixon 1993]对前三个建议进行了比较。

1) SIP，简单Internet协议。它针对当前的IP提出了一个最小幅度的修改建议，采用64位地址和一个不同的首部格式（首部的前4比特仍然包含协议的版本号，其值不再是4）。

2) PIP。这个建议也采用了更大的、可变长度的和有层次结构的地址，而且首部格式也不相同。

3) TUBA，代表“TCP and UDP with Bigger Address”，它基于OSI的CLNP（Connectionless Network Protocol，无连接网络协议），一个与IP类似的OSI协议。它提供大得多的地址空间：可变长度，可达20个字节。由于CLNP是一个现有的协议，而SIP和PIP只是建议，因此关于CLNP的文档已经出现。RFC 1347[Callon 1992]提供了TUBA的有关细节。文献[Perlman 1992]的第7章对IPv4和CLNP进行了比较。许多路由器已经支持CLNP，但是很少有主机也提供支持。

4) TP/IX，由RFC 1475 [Ullmann 1993]对它进行了描述。虽然SIP采用了64 bit的址址，但是它还改变了TCP和UDP的格式：两个协议均为32 bit的端口号，64 bit的序列号，64 bit的确认号，以及TCP的32 bit窗口。

前三个建议基本上采用了相同版本的TCP和UDP作为传输层协议。

由于四个建议只能有一个被选为IPv4的替换者，而且在你读到此书时可能已经做出选择，因此我们对它们不进行过多评论。虽然CIDR即将实现以解决目前的短期问题，但是IPv4后继者的实现则需要经过许多年。

3.11 小结

本章开始描述了IP首部的格式，并简要讨论了首部中的各个字段。我们还介绍了IP路由选择，并指出主机的路由选择可以非常简单：如果目的主机在直接相连的网络上，那么就把数据报直接传给目的主机，否则传给默认路由器。

在进行路由选择决策时，主机和路由器都使用路由表。在表中有三种类型的路由：特定主机型、特定网络型和默认路由型。路由表中的表目具有一定的优先级。在选择路由时，主机路由优先于网络路由，最后在没有其他可选路由存在时才选择默认路由。

IP路由选择是通过逐跳来实现的。数据报在各站的传输过程中目的IP地址始终不变，但是封装和目的链路层地址在每一站都可以改变。大多数的主机和许多路由器对于非本地网络的数据报都使用默认的下一站路由器。

A类和B类地址一般都要进行子网划分。用于子网号的比特数通过子网掩码来指定。我们为此举了一个实例来详细说明，即作者所在的子网，并介绍了变长子网的概念。子网的划分缩小了Internet路由表的规模，因为许多网络经常可以通过单个表目就可以访问了。接口和网络的有关信息通过`ifconfig`和`netstat`命令可以获得，包括接口的IP地址、子网掩码、广播地址以及MTU等。

在本章的最后，我们对Internet协议族潜在的改进建议——下一代IP进行了讨论。

习题

3.1 环回地址必须是127.0.0.1吗？

3.2 在图3-6中指出有两个网络接口的路由器。

3.3 子网号为16 bit的A类地址与子网号为8 bit的B类地址的子网掩码有什么不同？

3.4 阅读RFC 1219 [Tsuchiya 1991]，学习分配子网号和主机号的有关推荐技术。

3.5 子网掩码255.255.0.255是否对A类地址有效？

3.6 你认为为什么3.9小节中打印出来的环回接口的MTU要设置为1536？

3.7 TCP/IP协议族是基于一种数据报的网络技术，即IP层，其他的协议族则基于面向连接的网络技术。阅读文献[Clark 1988]，找出数据报网络层提供的三个优点。

第4章 ARP：地址解析协议

4.1 引言

本章我们要讨论的问题是只对 TCP/IP协议簇有意义的IP地址。数据链路如以太网或令牌环网都有自己的寻址机制（常常为 48 bit地址），这是使用数据链路的任何网络层都必须遵从的。一个网络如以太网可以同时被不同的网络层使用。例如，一组使用 TCP/IP协议的主机和另一组使用某种PC网络软件的主机可以共享相同的电缆。

当一台主机把以太网数据帧发送到位于同一局域网上的另一台主机时，是根据 48 bit的以太网地址来确定目的接口的。设备驱动程序从不检查 IP数据报中的目的IP地址。

地址解析为这两种不同的地址形式提供映射： 32 bit的IP地址和数据链路层使用的任何类型的地址。 RFC 826 [Plummer 1982]是ARP规范描述文档。

本章及下一章我们要讨论的两种协议如图 4-1所示：ARP（地址解析协议）和RARP（逆地址解析协议）。

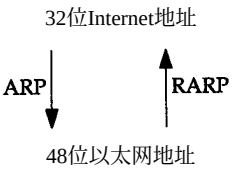


图4-1 地址解析协议：ARP和RARP

ARP为IP地址到对应的硬件地址之间提供动态映射。我们之所以用动态这个词是因为这个过程是自动完成的，一般应用程序用户或系统管理员不必关心。

RARP是被那些没有磁盘驱动器的系统使用（一般是无盘工作站或 X终端），它需要系统管理员进行手工设置。我们在第 5章对它进行讨论。

4.2 一个例子

任何时候我们敲入下面这个形式的命令：

```
% ftp bsdi
```

都会进行以下这些步骤。这些步骤的序号如图 4-2所示。

1) 应用程序FTP客户端调用函数gethostbyname(3)把主机名（bsdi）转换成32 bit的IP地址。这个函数在DNS（域名系统）中称作解析器，我们将在第 14章对它进行介绍。这个转换过程或者使用DNS，或者在较小网络中使用一个静态的主机文件（/etc/hosts）。
2) FTP客户端请求TCP用得到的IP地址建立连接。
3) TCP发送一个连接请求分段到远端的主机，即用上述 IP地址发送一份 IP数据报（在第18章我们将讨论完成这个过程的细节）。
4) 如果目的主机在本地网络上（如以太网、令牌环网或点对点链接的另一端），那么IP数据报可以直接送到目的主机上。如果目的主机在一个远程网络上，那么就通过 IP选路函数来确定位于本地网络上的下一站路由器地址，并让它转发 IP数据报。在这两种情况下，IP数据报都是被送到位于本地网络上的一台主机或路由器。
5) 假定是一个以太网，那么发送端主机必须把 32 bit的IP地址变换成48 bit的以太网地址。

从逻辑Internet地址到对应的物理硬件地址需要进行翻译。这就是 ARP的功能。

ARP本来是用于广播网络的，有许多主机或路由器连在同一个网络上。

6) ARP发送一份称作 ARP请求的以太网数据帧给以太网上的每个主机。这个过程称作广播，如图 4-2中的虚线所示。 ARP请求数据帧中包含目的主机的 IP地址（主机名为 bsdi），其意思是“ 如果你是这个IP地址的拥有者，请回答你的硬件地址。”

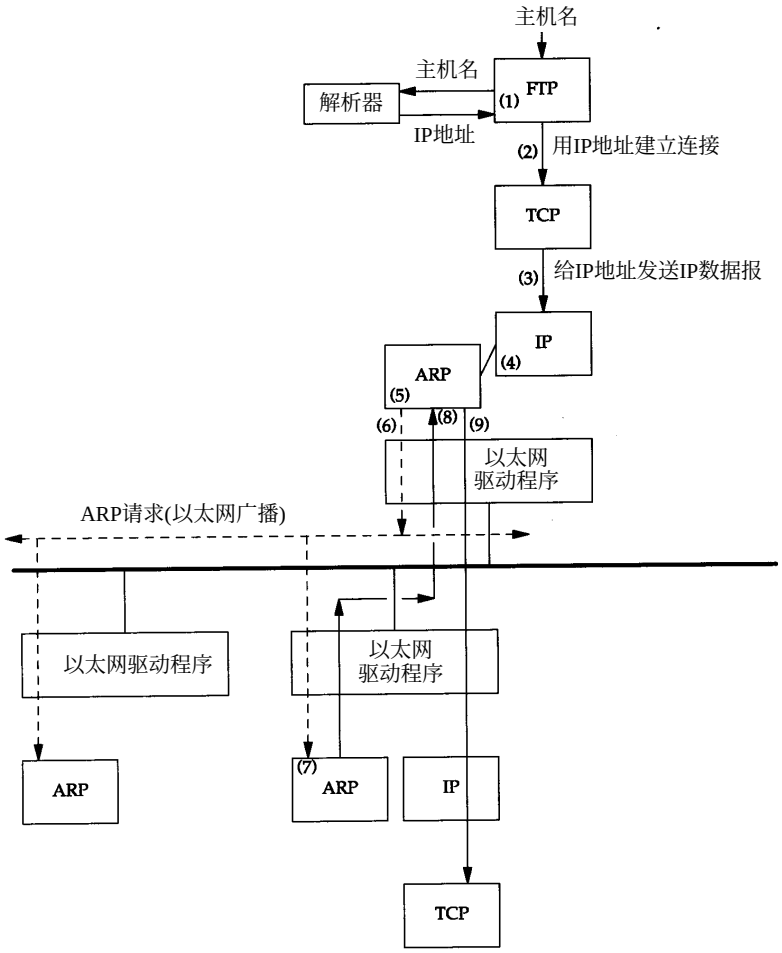


图4-2 当用户输入命令“ ftp 主机名 ”时ARP的操作

7) 目的主机的ARP层收到这份广播报文后，识别出这是发送端在寻问它的 IP地址，于是发送一个ARP应答。这个ARP应答包含IP地址及对应的硬件地址。

8) 收到ARP应答后，使ARP进行请求—应答交换的IP数据报现在就可以传送了。

9) 发送IP数据报到目的主机。

在ARP背后有一个基本概念，那就是网络接口有一个硬件地址（一个 48 bit的值，标识不同的以太网或令牌环网络接口）。在硬件层次上进行的数据帧交换必须有正确的接口地址。但是，TCP/IP有自己的地址：32 bit的IP地址。知道主机的IP地址并不能让内核发送一帧数据给主机。内核（如以太网驱动程序）必须知道目的端的硬件地址才能发送数据。 ARP的功能是在32 bit的IP地址和采用不同网络技术的硬件地址之间提供动态映射。

点对点链路不使用 ARP。当设置这些链路时（一般在引导过程进行），必须告知内核链路

每一端的IP地址。像以太网地址这样的硬件地址并不涉及。

4.3 ARP高速缓存

ARP高效运行的关键是由于每个主机上都有一个 ARP高速缓存。这个高速缓存存放了最近Internet地址到硬件地址之间的映射记录。高速缓存中每一项的生存时间一般为 20分钟，起始时间从被创建时开始算起。

我们可以用arp(8)命令来检查ARP高速缓存。参数-a的意思是显示高速缓存中所有的内容。

```
bsdi %arp -a
sun (140.252.13.33) at 8:0:20:3:f6:42
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
```

48 bit的以太网地址用6个十六进制的数来表示，中间以冒号隔开。在 4.8小节我们将讨论arp命令的其他功能。

4.4 ARP的分组格式

在以太网上解析IP地址时， ARP请求和应答分组的格式如图 4-3所示（ARP可以用于其他类型的网络，可以解析 IP地址以外的地址。紧跟着帧类型字段的前四个字段指定了最后四个字段的类型和长度）。

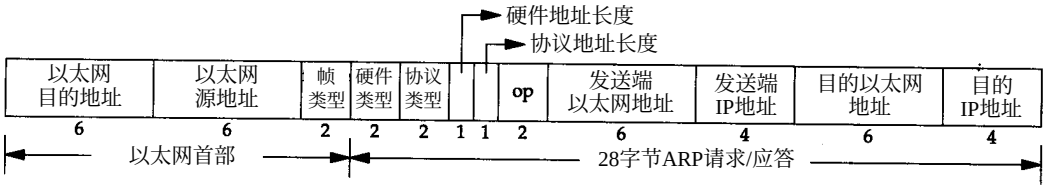


图4-3 用于以太网的ARP请求或应答分组格式

以太网报头中的前两个字段是以太网的源地址和目的地址。目的地址为全 1的特殊地址是广播地址。电缆上的所有以太网接口都要接收广播的数据帧。

两个字节长的以太网帧类型表示后面数据的类型。对于 ARP请求或应答来说，该字段的值为0x0806。

形容词hardware(硬件)和protocol(协议)用来描述ARP分组中的各个字段。例如，一个 ARP请求分组询问协议地址（这里是IP地址）对应的硬件地址（这里是以太网地址）。

硬件类型字段表示硬件地址的类型。它的值为 1即表示以太网地址。协议类型字段表示要映射的协议地址类型。它的值为 0x0800即表示IP地址。它的值与包含 IP数据报的以太网数据帧中的类型字段的值相同，这是有意设计的（参见图 2-1）。

接下来的两个1字节的字段，硬件地址长度和协议地址长度分别指出硬件地址和协议地址的长度，以字节为单位。对于以太网上 IP地址的ARP请求或应答来说，它们的值分别为 6和4。

操作字段指出四种操作类型，它们是 ARP请求（值为1）、ARP应答（值为2）、RARP请求（值为3）和RARP应答（值为4）（我们在第5章讨论RARP）。这个字段必需的，因为 ARP请求和ARP应答的帧类型字段值是相同的。

接下来的四个字段是发送端的硬件地址（在本例中是以太网地址）、发送端的协议地址（IP地址）、目的端的硬件地址和目的端的协议地址。注意，这里有一些重复信息：在以太网

的数据帧报头中和ARP请求数据帧中都有发送端的硬件地址。

对于一个ARP请求来说，除目的端硬件地址外的所有其他的字段都有填充值。当系统收到一份目的端为本机的ARP请求报文后，它就把硬件地址填进去，然后用两个目的端地址分别替换两个发送端地址，并把操作字段置为2，最后把它发送回去。

4.5 ARP举例

在本小节中，我们用tcpdump命令来看一看运行像Telnet这样的普通TCP工具软件时ARP会做些什么。附录A包含tcpdump命令的其他细节。

4.5.1 一般的例子

为了看清楚ARP的运作过程，我们执行telnet命令与无效的服务器连接。

```
bsdi % arp -a                          检验ARP高速缓存是空的
bsdi % telnet svr4 discard             连接无效的服务器
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
^]                                     键入Ctrl和右括号，使Telnet回到提示符并关闭
telnet> quit
Connection closed.
```

当我们在另一个系统（sun）上运行带有-e选项的tcpdump命令时，显示的是硬件地址（在我们的例子中是48 bit的以太网地址）。

图4-4中的tcpdump的原始输出如附录A中的图A-3所示。由于这是本书第一个tcpdump输出例子，你应该去查看附录中的原始输出，看看我们作了哪些修改。

```
1  0.0                  0:0:c0:6f:2d:40 ff:ff:ff:ff:ff:ff arp 60:
                        arp who-has svr4 tell bsdi
2  0.002174 (0.0022)    0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 arp 60:
                        arp reply svr4 is-at 0:0:c0:c2:9b:26
3  0.002831 (0.0007)    0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60:
                        bsdi.1030 > svr4.discard: S 596459521:596459521(0)
                        win 4096 <mss 1024> [tos 0x10]
4  0.007834 (0.0050)    0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 ip 60:
                        svr4.discard > bsdi.1030: S 3562228225:3562228225(0)
                        ack 596459522 win 4096 <mss 1024>
5  0.009615 (0.0018)    0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60:
                        bsdi.1030 > svr4.discard: . ack 1 win 4096 [tos 0x10]
```

图4-4 TCP连接请求产生的ARP请求和应答

我们删除了tcpdump命令输出的最后四行，因为它们是结束连接的信息（我们将在第18章进行讨论），与这里讨论的内容不相关。

在第1行中，源端主机（bsdi）的硬件地址是0:0:c0:6f:2d:40。目的端主机的硬件地址是ff:ff:ff:ff:ff:ff，这是一个以太网广播地址。电缆上的每个以太网接口都要接收这个数据帧并对它进行处理，如图4-2所示。

第1行中紧接着的一个输出字段是arp，表明帧类型字段的值是0x0806，说明此数据帧是一个ARP请求或回答。

在每行中，单词arp或ip后面的值60指的是以太网数据帧的长度。由于ARP请求或回答

的数据帧长都是42字节（28字节的ARP数据，14字节的以太网帧头），因此，每一帧都必须加入填充字符以达到以太网的最小长度要求：60字节。

请参见图1-7，这个最小长度60字节包含14字节的以太网帧头，但是不包括4个字节的以太网帧尾。有一些书把最小长度定为64字节，它包括以太网的帧尾。我们在图1-7中把最小长度定为46字节，是有意不包括14字节的帧首部，因为对应的最大长度（1500字节）指的是MTU——最大传输单元（见图2-5）。我们使用MTU经常是因为它对IP数据报的长度进行限制，但一般与最小长度无关。大多数的设备驱动程序或接口卡自动地用填充字符把以太网数据帧充满到最小长度。第3，4和5行中的IP数据报（包含TCP段）的长度都比最小长度短，因此都必须填充到60字节。

第1行中的下一个输出字段`arp who-has`表示作为ARP请求的这个数据帧中，目的IP地址是svr4的地址，发送端的IP地址是`bsdi`的地址。`tcpdump`打印出主机名对应的默认IP地址（在4.7节中，我们将用`-n`选项来查看ARP请求中真正的IP地址。）

从第2行中可以看到，尽管ARP请求是广播的，但是ARP应答的目的地址却是`bsdi`（0:0:c0:6f:2d:40）。ARP应答是直接送到请求端主机的，而是广播的。

`tcpdump`打印出`arp reply`的字样，同时打印出响应者的主机名和硬件地址。

第3行是第一个请求建立连接的TCP段。它的目的硬件地址是目的主机（`svr4`）。我们将在第18章讨论这个段的细节内容。

在每一行中，行号后面的数字表示`tcpdump`收到分组的时间（以秒为单位）。除第1行外，其他每行在括号中还包含了与上一行的时间差异（以秒为单位）。从这个图可以看出，发送ARP请求与收到ARP回答之间的延时是2.2 ms。而在0.7 ms之后发出第一段TCP报文。在本例中，用ARP进行动态地址解析的时间小于3 ms。

最后需要指出的一点，在`tcpdump`命令输出中，我们没有看到svr4在发出第一段TCP报文（第4行）之前发出的ARP请求。这是因为可能在`svr4`的ARP高速缓存中已经有`bsdi`的表项。一般情况下，当系统收到ARP请求或发送ARP应答时，都要把请求端的硬件地址和IP地址存入ARP高速缓存。在逻辑上可以假设，如果请求端要发送IP数据报，那么数据报的接收端将很可能会发送一个应答。

4.5.2 对不存在主机的ARP请求

如果查询的主机已关机或不存在会发生什么情况呢？为此我们指定一个并不存在的Internet地址——根据网络号和子网号所对应的网络确实存在，但是并不存在所指定的主机号。从图3-10可以看出，主机号从36到62的主机并不存在（主机号为63是广播地址）。这里，我们用主机号36来举例。

这次是Telnet的一个地址，而不是主机名

```
bsdi % date ; telnet 140.252.13.36 ; date
Sat Jan 30 06:46:33 MST 1993
Trying 140.252.13.36...
telnet: Unable to connect to remote host: Connection timed out
Sat Jan 30 06:47:49 MST 1993
```

在前一个日期输出后76秒

检查ARP高速缓存

```
bsdi % arp -a
? (140.252.13.36) at (incomplete)
```

tcpdump命令的输出如图4-5所示。

```
1    0.0                    arp who-has 140.252.13.36 tell bsdi
2    5.509069 ( 5.5091)     arp who-has 140.252.13.36 tell bsdi
3   29.509745 (24.0007)     arp who-has 140.252.13.36 tell bsdi
```

图4-5 对不存在主机的ARP请求

这一次，我们没有用-e选项，因为已经知道ARP请求是在网上广播的。

令人感兴趣的是看到多次进行ARP请求：第1次请求发生后5.5秒进行第2次请求，在24秒之后又进行第3次请求（在第21章我们将看到TCP的超时和重发算法的细节）。tcpdump命令输出的超时限制为29.5秒。但是，在telnet命令使用前后分别用date命令检查时间，可以发现Telnet客户端的连接请求似乎在大约75秒后才放弃。事实上，我们在后面将看到，大多数的BSD实现把完成TCP连接请求的时间限制设置为75秒。

在第18章中，当我们看到建立连接的TCP报文段序列时，会发现ARP请求对应于TCP试图发送的初始TCPSYN（同步）段。

注意，在线路上始终看不到TCP的报文段。我们能看到的是ARP请求。直到ARP回答返回时，TCP报文段才可以被发送，因为硬件地址到这时才可能知道。如果我们用过滤模式运行tcpdump命令，只查看TCP数据，那么将没有任何输出。

4.5.3 ARP高速缓存超时设置

在ARP高速缓存中的表项一般都要设置超时值（在4.8小节中，我们将看到管理员可以用arp命令把地址放入高速缓存中而不设置超时值）。从伯克利系统演变而来的系统一般对完整的表项设置超时值为20分钟，而对不完整的表项设置超时值为3分钟（在前面的例子中我们已见过一个不完整的表项，即在以太网上对一个不存在的主机发出ARP请求。）当这些表项再次使用时，这些实现一般都把超时值重新设为20分钟。

Host Requirements RFC表明即使表项正在使用时，超时值也应该启动，但是大多数从伯克利系统演变而来的系统没有这样做——它们每次都是在访问表项时重设超时值。

4.6 ARP代理

如果ARP请求是从一个网络的主机发往另一个网络上的主机，那么连接这两个网络的路由器就可以回答该请求，这个过程称作委托ARP或ARP代理(Proxy ARP)。这样可以欺骗发起ARP请求的发送端，使它误以为路由器就是目的主机，而事实上目的主机是在路由器的“另一边”。路由器的功能相当于目的主机的代理，把分组从其他主机转发给它。

举例是说明ARP代理的最好方法。如图3-10所示，系统sun与两个以太网相连。但是，我们也指出过，事实上并不是这样，请把它与封内图1进行比较。在sun和子网140.252.1之间实际存在一个路由器，就是这个具有ARP代理功能的路由器使得sun就好像在子网140.252.1上一样。具体安置如图4-6所示，路由器Telebit NetBlazer，取名为netb，在子网和主机sun之间。

当子网140.252.1（称作gemini）上的其他主机有一份IP数据报要传给地址为140.252.1.29的sun时，gemini比较网络号（140.252）和子网号（1），因为它们都是相同的，因而在图4-6上面的以太网中发送IP地址140.252.1.29的ARP请求。路由器netb识别出该IP地址属于它的一个拨号主机，于是把它的以太网接口地址140.252.1作为硬件地址来回答。主机gemini通过以太网发送IP数据报到netb，netb通过拨号SLIP链路把数据报转发到sun。这个过程对于所有

140.252.1子网上的主机来说都是透明的，主机sun实际上是在路由器netb后面进行配置的。

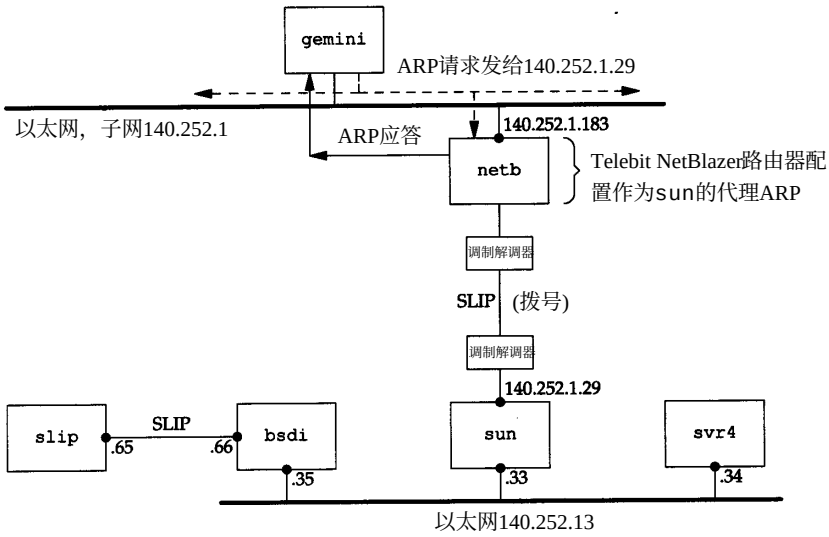


图4-6 ARP代理的例子

如果在主机gemini上执行arp命令，经过与主机sun通信以后，我们发现在同一个子网140.252.1上的netb和sun的IP地址映射的硬件地址是相同的。这通常是使用委托ARP的线索。

```
gemini %arp -a
                                    这里是子网140.252.1上其他主机的输出行
netb (140.252.1.183) at 0:80:ad:3:6a:80
sun (140.252.1.29) at 0:80:ad:3:6a:80
```

图4-6中的另一个需要解释的细节是在路由器netb的下方（SLIP链路）显然缺少一个IP地址。为什么在拨号SLIP链路的两端只拥有一个IP地址，而在bsdi和slip之间的两端却分别有一个IP地址？在3.8小节我们已经指出，用ifconfig命令可以显示拨号SLIP链路的目的地址，它是140.252.1.183。NetBlazer不需要知道拨号SLIP链路每一端的IP地址（这样做会用更多的IP地址）。相反，它通过分组到达的串行线路接口来确定发送分组的拨号主机，因此对于连接到路由器的每个拨号主机不需要用唯一的IP地址。所有的拨号主机使用同一个IP地址140.252.1.183作为SLIP链路的目的地址。

ARP代理可以把数据报传送到路由器sun上，但是子网140.252.13上的其他主机是如何处理的呢？选路必须使数据报能到达其他主机。这里需要特殊处理，选路表中的表项必须在网络140.252的某个地方制定，使所有数据报的目的端要么是子网140.252.13，要么是子网上的某个主机，这样都指向路由器netb。而路由器netb知道如何把数据报传到最终的目的端，即通过路由器sun。

ARP代理也称作混合ARP（promiscuousARP）或ARP出租(ARP hack)。这些名字来自于ARP代理的其他用途：通过两个物理网络之间的路由器可以互相隐藏物理网络。在这种情况下，两个物理网络可以使用相同的网络号，只要把中间的路由器设置成一个ARP代理，以响应一个网络到另一个网络主机的ARP请求。这种技术在过去用来隐藏一组在不同物理电缆上运行旧版TCP/IP的主机。分开这些旧主机有两个共同的理由，其一是它们不能处理子网划分，其二是它们使用旧的广播地址（所有比特值为0的主机号，而不是目前使用的所有比特值为1

的主机号）。

4.7 免费ARP

我们可以看到的另一个ARP特性称作免费ARP (gratuitous ARP)。它是指主机发送ARP查找自己的IP地址。通常，它发生在系统引导期间进行接口配置的时候。

在互联网中，如果我们引导主机bsdi并在主机sun上运行tcpdump命令，可以看到如图4-7所示的分组。

```
1  0.0                    0:0:c0:6f:2d:40 ff:ff:ff:ff:ff:ff arp 60:
                          arp who-has 140.252.13.35 tell 140.252.13.35
```

图4-7 免费ARP的例子

（我们用-n选项运行tcpdump命令，打印出点分十进制的地址，而不是主机名）。对于ARP请求中的各字段来说，发送端的协议地址和目的端的协议地址是一致的：即主机 bsdi的地址140.252.13.35。另外，以太网报头中的源地址0:0:c0:6f:2d:40，正如tcpdump命令显示的那样，等于发送端的硬件地址（见图4-4）。

免费ARP可以有两个方面的作用：

1) 一个主机可以通过它来确定另一个主机是否设置了相同的 IP地址。主机bsdi并不希望对此请求有一个回答。但是，如果收到一个回答，那么就会在终端日志上产生一个错误消息“以太网地址：a:b:c:d:e:f发送来重复的IP地址”。这样就可以警告系统管理员，某个系统有不正确的设置。

2) 如果发送免费ARP的主机正好改变了硬件地址（很可能是主机关机了，并换了一块接口卡，然后重新启动），那么这个分组就可以使其他主机高速缓存中旧的硬件地址进行相应的更新。一个比较著名的ARP协议事实[Plummer 1982]是，如果主机收到某个IP地址的ARP请求，而且它已经在接收者的高速缓存中，那么就要用 ARP请求中的发送端硬件地址（如以太网地址）对高速缓存中相应的内容进行更新。主机接收到任何 ARP请求都要完成这个操作（ARP请求是在网上广播的，因此每次发送ARP请求时网络上的所有主机都要这样做）。

文献[Bhide、Elnozahy和Morgan 1991]中有一个应用例子，通过发送含有备份硬件地址和故障服务器的IP地址的免费ARP请求，使得备份文件服务器可以顺利地接替故障服务器进行工作。这使得所有目的地为故障服务器的报文都被送到备份服务器那里，客户程序不用关心原来的服务器是否出了故障。

> 不幸的是，作者却反对这个做法，因为这取决于所有不同类型的客户端都要有正确的ARP协议实现。他们显然碰到过客户端的ARP协议实现与规范不一致的情况。
>
> 通过检查作者所在子网上的所有系统可以发现，SunOS 4.1.3和4.4BSD在引导时都发送免费ARP，但是SVR4却没有这样做。

4.8 arp命令

我们已经用过这个命令及参数-a来显示ARP高速缓存中的所有内容。这里介绍其他参数的功能。

超级用户可以用选项-d来删除ARP高速缓存中的某一项内容（这个命令格式可以在运行

一些例子之前使用，以让我们看清楚ARP的交换过程）。

另外，可以通过选项-s来增加高速缓存中的内容。这个参数需要主机名和以太网地址：对应于主机名的IP地址和以太网地址被增加到高速缓存中。新增加的内容是永久性的（比如，它没有超时值），除非在命令行的末尾附上关键字temp。

位于命令行末尾的关键字pub和-s选项一起，可以使系统起着主机ARP代理的作用。系统将回答与主机名对应的IP地址的ARP请求，并以指定的以太网地址作为应答。如果广播的地址是系统本身，那么系统就为指定的主机名起着委托ARP代理的作用。

4.9 小结

在大多数的TCP/IP实现中，ARP是一个基础协议，但是它的运行对于应用程序或系统管理员来说一般是透明的。ARP高速缓存在它的运行过程中非常关键，我们可以用arp命令对高速缓存进行检查和操作。高速缓存中的每一项内容都有一个定时器，根据它来删除不完整和完整的表项。arp命令可以显示和修改ARP高速缓存中的内容。

我们介绍了ARP的一般操作，同时也介绍了一些特殊的功能：委托ARP（当路由器对来自于另一个路由器接口的ARP请求进行应答时）和免费ARP（发送自己IP地址的ARP请求，一般发生在引导过程中）。

习题

4.1 当输入命令以生成类似图4-4那样的输出时，发现本地ARP快速缓存为空以后，输入命令

```
bsdi % rsh svr4 arp -a
```

如果发现目的主机上的ARP快速缓存也是空的，那将发生什么情况？（该命令将在svr4主机上运行arp -a命令）。

4.2 请描述如何判断一个给定主机是否能正确处理接收到的非必要的ARP请求的方法。

4.3 由于发送一个数据包后ARP将等待响应，因此4.2节所描述的步骤7可能会持续一段时间。你认为ARP将如何处理在这期间收到相同目的IP地址发来的多个数据包？

4.4 在4.5节的最后，我们指出Host Requirements RFC和伯克利派生系统在处理活动ARP表目的超时时存在差异。那么如果我们在一个由伯克利派生系统的客户端上，试图与一个正在更换以太网卡而处于关机状态的服务器主机联系，这时会发生什么情况？如果服务器在引导过程中广播一份免费ARP，这种情况是否会发生变化？

第5章 RARP：逆地址解析协议

5.1 引言

具有本地磁盘的系统引导时，一般是从磁盘上的配置文件中读取 IP地址。但是无盘机，如X终端或无盘工作站，则需要采用其他方法来获得 IP地址。

网络上的每个系统都具有唯一的硬件地址，它是由网络接口生产厂家配置的。无盘系统的RARP实现过程是从接口卡上读取唯一的硬件地址，然后发送一份 RARP请求（一帧在网络上广播的数据），请求某个主机响应该无盘系统的 IP地址（在RARP应答中）。

在概念上这个过程是很简单的，但是实现起来常常比 ARP要困难，其原因在本章后面介绍。RARP的正式规范是RFC 903 [Finlayson et al. 1984]。

5.2 RARP的分组格式

RARP分组的格式与 ARP分组基本一致（见图 4-3）。它们之间主要的差别是 RARP请求或应答的帧类型代码为0x8035，而且RARP请求的操作代码为3，应答操作代码为4。

对应于ARP，RARP请求以广播方式传送，而RARP应答一般是单播(unicast)传送的。

5.3 RARP举例

在互联网中，我们可以强制 sun主机从网络上引导，而不是从本地磁盘引导。如果在主机bsdi上运行RARP服务程序和tcpdump命令，就可以得到如图5-1那样的输出。用-e参数使得tcpdump命令打印出硬件地址：

```
1  0.0                  8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                        rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42

2  0.13 (0.13)          0:0:c0:6f:2d:40 8:0:20:3:f6:42 rarp 42:
                        rarp reply 8:0:20:3:f6:42 at sun

3  0.14 (0.01)          8:0:20:3:f6:42 0:0:c0:6f:2d:40 ip 65:
                        sun.26999 > bsdi.tftp: 23 RRQ "8CFC0D21.SUN4C"
```

图5-1 RARP请求和应答

RARP请求是广播方式（第 1行），而第2行的RARP应答是单播方式。第 2行的输出中at sun表示RARP应答包含主机sun的IP地址（140.252.13.33）。

在第3行中，我们可以看到，一旦 sun收到IP地址，它就发送一个TFTP读请求（RRQ）给文件8CFC0D21.SUN4C（TFTP表示简单文件传送协议。我们将在第 15章详细介绍）。文件名中的8个十六进制数字表求主机 sun的IP地址140.252.13.33。这个IP地址在RARP应答中返回。文件名的后缀SUN4C表示被引导系统的类型。

tcpdump在第3行中指出IP数据报的长度是65个字节，而不是一个UDP数据报（实际上是一个UDP数据报），因为我们运行tcpdump命令时带有 - e参数，以查看硬件层的地址。在图5-1中

需要指出的另一点是，第2行中的以太网数据帧长度比最小长度还要小（在4.5节中我们说过应该是60字节）。其原因是我们在发送该以太网数据帧的系统（bsdi）上运行tcpdump命令。应用程序rarpd写42字节到BSD分组过滤设备上（其中14字节为以太网数据帧的报头，剩下的28字节是RARP应答），这就是tcpdump收到的副本。但是以太网设备驱动程序要把这一短帧填充空白字符以达到最小传输长度（60）。如果我们在另一个系统上运行tcpdump命令，其长度将会是60。

从这个例子可以看出，当无盘系统从RARP应答中收到它的IP地址后，它将发送TFTP请求来读取引导映象。在这一点上我们将不再进一步详细讨论无盘系统是如何引导的（第16章将描述无盘X终端利用RARP、BOOTP以及TFTP进行引导的过程）。

当网络上没有RARP服务器时，其结果如图5-2所示。每个分组的目的地址都是以太网广播地址。在who- 后面的以太网地址是目的硬件地址，跟在tell后面的以太网地址是发送端的硬件地址。

请注意重发的频度。第一次重发是在6.55秒以后，然后增加到42.80秒，然后又减到5.34秒和6.55秒，然后又回到42.79秒。这种不确定的情况一直继续下去。如果计算一下两次重发之间的时间间隔，我们发现存在一种双倍的关系：从5.34到6.55是1.21秒，从6.55到8.97是2.42秒，从8.97到13.80是4.83秒，一直这样继续下去。当时间间隔达到某个阈值时（大于42.80秒），它又重新置为5.34秒。

```
1     0.0              8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
2     6.55 ( 6.55)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
3    15.52 ( 8.97)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
4    29.32 (13.80)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
5    52.78 (23.46)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
6    95.58 (42.80)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
7   100.92 ( 5.34)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
8   107.47 ( 6.55)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
9   116.44 ( 8.97)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
10  130.24 (13.80)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
11  153.70 (23.46)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
12  196.49 (42.79)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
```

图5-2 网络中没有RARP服务器的RARP请求

超时间隔采用这样的递增方法比每次都采用相同值的方法要好。在图6-8中，我们将看到一种错误的超时重发方法，以及在第21章中将看到TCP的超时重发机制。

5.4 RARP服务器的设计

虽然RARP在概念上很简单，但是一个RARP服务器的设计与系统相关而且比较复杂。相反，提供一个ARP服务器很简单，通常是TCP/IP在内核中实现的一部分。由于内核知道IP地

址和硬件地址，因此当它收到一个询问 IP地址的ARP请求时，只需用相应的硬件地址来提供应答就可以了。

5.4.1 作为用户进程的RARP服务器

RARP服务器的复杂性在于，服务器一般要为多个主机（网络上所有的无盘系统）提供硬件地址到IP地址的映射。该映射包含在一个磁盘文件中（在 Unix系统中一般位于/etc/ethers目录中）。由于内核一般不读取和分析磁盘文件，因此 RARP服务器的功能就由用户进程来提供，而不是作为内核的TCP/IP实现的一部分。

更为复杂的是，RARP请求是作为一个特殊类型的以太网数据帧来传送的（帧类型字段值为0x8035，如图2-1所示）。这说明RARP服务器必须能够发送和接收这种类型的以太网数据帧。在附录A中，我们描述了BSD分组过滤器、 Sun的网络接口栓以及 SVR4数据链路提供者接口都可用来接收这些数据帧。由于发送和接收这些数据帧与系统有关，因此 RARP服务器的实现是与系统捆绑在一起的。

5.4.2 每个网络有多个RARP服务器

RARP服务器实现的一个复杂因素是 RARP请求是在硬件层上进行广播的，如图 5-2所示。这意味着它们不经过路由器进行转发。为了让无盘系统在RARP服务器关机的状态下也能引导，通常在一个网络上（例如一根电缆）要提供多个 RARP服务器。

当服务器的数目增加时（以提供冗余备份），网络流量也随之增加，因为每个服务器对每个RARP请求都要发送RARP应答。发送RARP请求的无盘系统一般采用最先收到的 RARP应答（对于ARP，我们从来没有遇到这种情况，因为只有一台主机发送 ARP应答）。另外，还有一种可能发生的情况是每个RARP服务器同时应答，这样会增加以太网发生冲突的概率。

5.5 小结

RARP协议是许多无盘系统在引导时用来获取 IP地址的。RARP分组格式基本上与ARP分组一致。一个RARP请求在网络上进行广播，它在分组中标明发送端的硬件地址，以请求相应IP地址的响应。应答通常是单播传送的。

RARP带来的问题包括使用链路层广播，这样就阻止大多数路由器转发 RARP请求，只返回很少信息：只是系统的 IP地址。在第16章中，我们将看到BOOTP在无盘系统引导时会返回更多的信息：IP地址和引导主机的名字等。

虽然RARP在概念上很简单，但是 RARP服务器的实现却与系统相关。因此，并不是所有的TCP/IP实现都提供RARP服务器。

习题

5.1 RARP需要不同的帧类型字段吗？ ARP和RARP都使用相同的值0x0806吗？

5.2 在一个有多个RARP服务器的网络上，如何防止它们的响应发生冲突？

第6章　ICMP：Internet控制报文协议

6.1　引言

ICMP经常被认为是IP层的一个组成部分。它传递差错报文以及其他需要注意的信息。ICMP报文通常被IP层或更高层协议（TCP或UDP）使用。一些ICMP报文把差错报文返回给用户进程。

ICMP报文是在IP数据报内部被传输的，如图6-1所示。

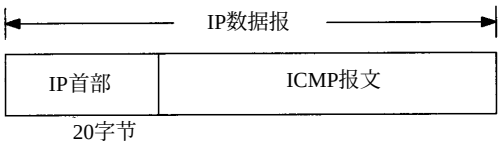


图6-1　ICMP封装在IP数据报内部

ICMP 的正式规范参见 RFC 792 [Posterl 1981b]。

ICMP报文的格式如图6-2所示。所有报文的前4个字节都是一样的，但是剩下的其他字节则互不相同。下面我们将逐个介绍各种报文格式。

类型字段可以有15个不同的值，以描述特定类型的ICMP报文。某些ICMP报文还使用代码字段的值来进一步描述不同的条件。

检验和字段覆盖整个ICMP报文。使用的算法与我们在3.2节中介绍的IP首部检验和算法相同。ICMP的检验和是必需的。

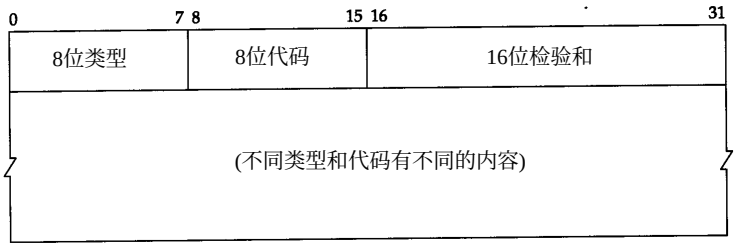


图6-2　ICMP报文

在本章中，我们将一般地讨论ICMP报文，并对其中一部分作详细介绍：地址掩码请求和应答、时间戳请求和应答以及不可达端口。我们将详细介绍第 27章Ping程序所使用的回应请求和应答报文和第9章处理IP路由的ICMP报文。

6.2　ICMP报文的类型

各种类型的ICMP报文如图6-3所示，不同类型由报文中的类型字段和代码字段来共同决定。

图中的最后两列表明ICMP报文是一份查询报文还是一份差错报文。因为对 ICMP差错报文有时需要作特殊处理，因此我们需要对它们进行区分。例如，在对 ICMP差错报文进行响应时，永远不会生成另一份ICMP差错报文（如果没有这个限制规则，可能会遇到一个差错产生另一个差错的情况，而差错再产生差错，这样会无休止地循环下去）。

当发送一份ICMP差错报文时，报文始终包含IP的首部和产生ICMP差错报文的IP数据报的前8个字节。这样，接收ICMP差错报文的模块就会把它与某个特定的协议（根据 IP数据报首

类 型	代 码	描 述	查 询	差 错
0	0	回显应答(Ping应答，第7章)	•	
3		目的不可达：		
	0	网络不可达（9.3节）		•
	1	主机不可达（9.3节）		•
	2	协议不可达		•
	3	端口不可达（6.5节）		•
	4	需要进行分片但设置了不分片比特（11.6节）		•
	5	源站选路失败（8.5节）		•
	6	目的网络不认识		•
	7	目的主机不认识		•
	8	源主机被隔离（作废不用）		•
	9	目的网络被强制禁止		•
	10	目的主机被强制禁止		•
	11	由于服务类型TOS，网络不可达（9.3节）		•
	12	由于服务类型TOS，主机不可达（9.3节）		•
	13	由于过滤，通信被强制禁止		•
	14	主机越权		•
	15	优先权中止生效		•
4	0	源端被关闭（基本流控制，11.11节）		•
5		重定向（9.5节）：		•
	0	对网络重定向		•
	1	对主机重定向		•
	2	对服务类型和网络重定向		•
	3	对服务类型和主机重定向		•
8	0	请求回显（Ping请求，第7章）	•	
9	0	路由器通告（9.6节）	•	
10	0	路由器请求（9.6节）	•	
11		超时：		
	0	传输期间生存时间为0（Traceroute, 第8章）		•
	1	在数据报组装期间生存时间为0（11.5节）		•
12		参数问题：		
	0	坏的IP首部（包括各种差错）		•
	1	缺少必需的选项		•
13	0	时间戳请求（6.4节）	•	
14	0	时间戳应答（6.4节）	•	
15	0	信息请求（作废不用）	•	
16	0	信息应答（作废不用）	•	
17	0	地址掩码请求（6.3节）	•	
18	0	地址掩码应答（6.3节）	•	

图6-3 ICMP报文类型

部中的协议字段来判断）和用户进程（根据包含在IP数据报前8个字节中的TCP或UDP报文首部中的TCP或UDP端口号来判断）联系起来。6.5节将举例来说明一点。

下面各种情况都不会导致产生ICMP差错报文：

1) ICMP差错报文（但是，ICMP查询报文可能会产生ICMP差错报文）。

2) 目的地址是广播地址（见图3-9）或多播地址（D类地址，见图1-5）的IP数据报。

3) 作为链路层广播的数据报。

4) 不是IP分片的第一片（将在11.5节介绍分片）。

5) 源地址不是单个主机的数据报。这就是说，源地址不能为零地址、环回地址、广播地址或多播地址。

这些规则是为了防止过去允许ICMP差错报文对广播分组响应所带来的广播风暴。

6.3 ICMP地址掩码请求与应答

ICMP地址掩码请求用于无盘系统在引导过程中获取自己的子网掩码（ 3.5节）。系统广播它的ICMP请求报文（这一过程与无盘系统在引导过程中用 RARP获取IP地址是类似的）。无盘系统获取子网掩码的另一个方法是 BOOTP协议，我们将在第 16章中介绍。 ICMP地址掩码请求和应答报文的格式如图6-4所示。

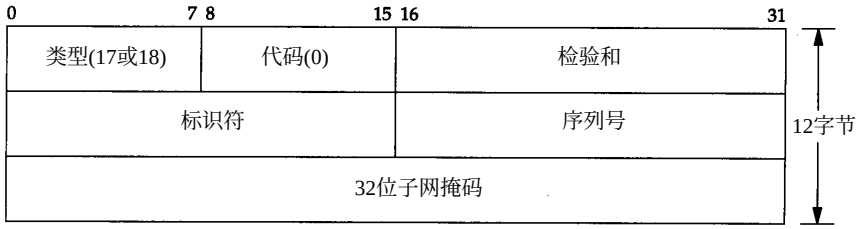


图6-4 ICMP地址掩码请求和应答报文

ICMP报文中的标识符和序列号字段由发送端任意选择设定，这些值在应答中将被返回。这样，发送端就可以把应答与请求进行匹配。

我们可以写一个简单的程序（取名为 icmpaddrmask），它发送一份ICMP地址掩码请求报文，然后打印出所有的应答。由于一般是把请求报文发往广播地址，因此这里我们也这样做。目的地址（140.252.13.63）是子网140.252.13.32的广播地址（见图3-12）。

```
sun %icmpaddrmask 140.252.13.63
received mask = ffffffe0, from 140.252.13.34 来自本机
received mask = ffffffe0, from 140.252.13.35 来自bsdi
received mask = ffff0000, from 140.252.13.33 来自svr4
```

在输出中我们首先注意到的是，从 svr4返回的子网掩码是错的。显然，尽管 svr4接口已经设置了正确的子网掩码，但是 SVR4还是返回了一个普通的 B类地址掩码，就好像子网并不存在一样。

```
svr4 %ifconfig emd0
emd0: flags=23<UP,BROADCAST,NOTRAILERS>
        inet 140.252.13.34 netmask ffffffe0 broadcast 140.252.13.63
```

SVR4处理ICMP地址掩码请求过程存在差错。

我们用tcpdump命令来查看主机bsdi上的情况，输出如图6-5所示。我们用 - e选项来查看硬件地址。

```
1  0.0                  8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff ip 60:
                        sun > 140.252.13.63: icmp: address mask request
2  0.00 (0.00)          0:0:c0:6f:2d:40 ff:ff:ff:ff:ff:ff ip 46:
                        bsdi > sun: icmp: address mask is 0xffffffe0
3  0.01 (0.01)          0:0:c0:c2:9b:26 8:0:20:3:f6:42 ip 60:
                        svr4 > sun: icmp: address mask is 0xffff0000
```

图6-5 发到广播地址的ICMP地址掩码请求

注意，尽管在线路上什么也看不见，但是发送主机 sun也能接收到ICMP应答（带有上面“来自本机”的输出行）。这是广播的一般特性：发送主机也能通过某种内部环回机制收到一份广播报文拷贝。由于术语“广播”的定义是指局域网上的所有主机，因此它必须包括发送

主机在内（参见图2-4，当以太网驱动程序识别出目的地址是广播地址后，它就把分组送到网络上，同时传一份拷贝到环回接口）。

接下来，bsdi广播应答，而svr4却只把应答传给请求主机。通常，应答地址必须是单播地址，除非请求端的源IP地址是0.0.0.0。本例不属于这种情况，因此，把应答发送到广播地址是BSD/386的一个内部差错。

> RFC规定，除非系统是地址掩码的授权代理，否则它不能发送地址掩码应答（为了成为授权代理，它必须进行特殊配置，以发送这些应答。参见附录E）。但是，正如我们从本例中看到的那样，大多数主机在收到请求时都发送一个应答，甚至有一些主机还发送差错的应答。

最后一点可以通过下面的例子来说明。我们向本机IP地址和环回地址分别发送地址掩码请求：

```
sun % icmpaddrmask sun
received mask= ff000000, from 140.252.13.33
sun % icmpaddrmask localhost
received mask= ff000000, from 127.0.0.1
```

上述两种情况下返回的地址掩码对应的都是环回地址，即A类地址127.0.0.1。还有，我们从图2-4可以看到，发送给本机IP地址的数据报（140.252.12.33）实际上是送到环回接口。ICMP地址掩码应答必须是收到请求接口的子网掩码（这是因为多接口主机每个接口有不同的子网掩码），因此两种情况下地址掩码请求都来自于环回接口。

6.4 ICMP时间戳请求与应答

ICMP时间戳请求允许系统向另一个系统查询当前的时间。返回的建议值是自午夜开始计算的毫秒数，协调的统一时间（Coordinated Universal Time, UTC）（早期的参考手册认为UTC是格林尼治时间）。这种ICMP报文的好处是它提供了毫秒级的分辨率，而利用其他方法从别的主机获取的时间（如某些Unix系统提供的rdate命令）只能提供秒级的分辨率。由于返回的时间是从午夜开始计算的，因此调用者必须通过其他方法获知当时的日期，这是它的一个缺陷。

ICMP时间戳请求和应答报文格式如图6-6所示。

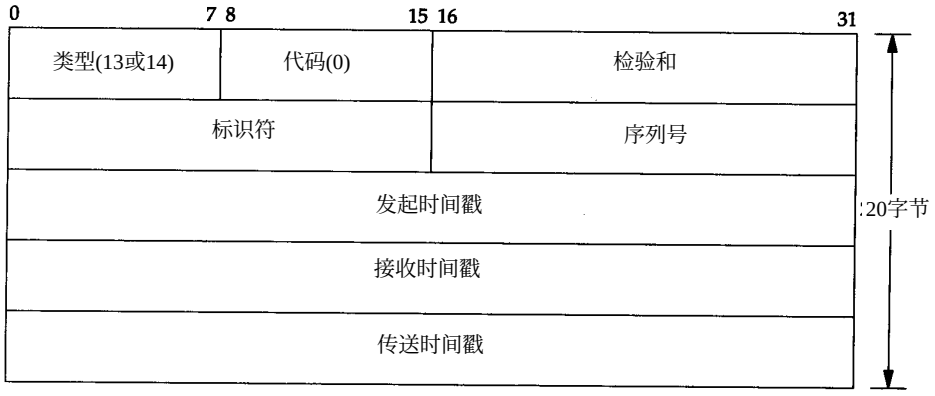


图6-6 ICMP时间戳请求和应答报文

请求端填写发起时间戳，然后发送报文。应答系统收到请求报文时填写接收时间戳，在发送应答时填写发送时间戳。但是，实际上，大多数的实现把后面两个字段都设成相同的值（提供三个字段的原因是可以让发送方分别计算发送请求的时间和发送应答的时间）。

6.4.1 举例

我们可以写一个简单程序（取名为 icmptime），给某个主机发送ICMP时间戳请求，并打印出返回的应答。它在我们的小互联网上运行结果如下：

```
sun % icmptime bsdi
orig = 83573336, recv = 83573330, xmit = 83573330, rtt = 2 ms
difference = -6 ms

sun % icmptime bsdi
orig = 83577987, recv = 83577980, xmit = 83577980, rtt = 2 ms
difference = -7 ms
```

程序打印出ICMP报文中的三个时间戳：发起时间戳（orig）、接收时间戳（recv）以及发送时间戳（xmit）。正如我们在这个例子以及下面的例子中所看到的那样，所有的主机把接收时间戳和发送时间戳都设成相同的值。

我们还能计算出往返时间（rtt），它的值是收到应答时的时间值减去发送请求时的时间值。difference的值是接收时间戳值减去发起时间戳值。这些值之间的关系如图6-7所示。

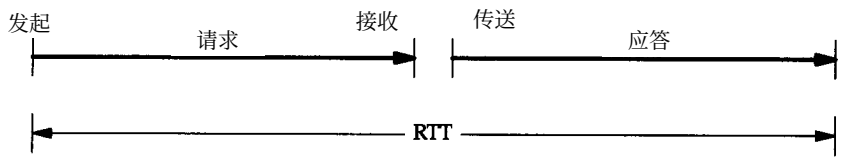


图6-7 icmptime 程序输出的值之间的关系

如果我们相信RTT的值，并且相信RTT的一半用于请求报文的传输，另一半用于应答报文的传输，那么为了使本机时钟与查询主机的时钟一致，本机时钟需要进行调整，调整值是difference减去RTT的一半。在前面的例子中，bsdi的时钟比sun的时钟要慢7 ms和8 ms。

由于时间戳的值是自午夜开始计算的毫秒数，即 UTC，因此它们的值始终小于86 400 000 (24×60×60×1000)。这些例子都是在下午4:00以前运行的，并且在一个比 UTC慢7个小时的时区，因此它们的值比82 800 000（2300小时）要大是有道理的。

如果对主机bsdi重复运行该程序数次，我们发现接收时间戳和发送时间戳的最后一位数总是0。这是因为该版本的软件（0.9.4版）只能提供10ms的时间分辨率（说明参见附录B）。

如果对主机svr4运行该程序两次，我们发现SVR4时间戳的最后三位数始终为0：

```
sun % icmptime svr4
orig = 83588210, recv = 83588000, xmit = 83588000, rtt = 4 ms
difference = -210 ms

sun % icmptime svr4
orig = 83591547, recv = 83591000, xmit = 83591000, rtt = 4 ms
difference = -547 ms
```

由于某种原因，SVR4在ICMP时间戳中不提供毫秒级的分辨率。这样，对秒以下的时间差调整将不起任何作用。

如果我们对子网 140.252.1上的其他主机运行该程序，结果表明其中一台主机的时钟与

sun相差3.7秒，而另一个主机时钟相差近75秒：

```
sun % icmptime gemini
orig = 83601883, recv = 83598140, xmit = 83598140, rtt = 247 ms
difference = -3743 ms

sun % icmptime aix
orig = 83606768, recv = 83532183, xmit = 83532183, rtt = 253 ms
difference = -74585 ms
```

另一个令人感兴趣的例子是路由器gateway（一个Cisco路由器）。它表明，当系统返回一个非标准时间戳值时（不是自午夜开始计算的毫秒数，UTC），它就用32 bit时间戳中的高位来表示。我们的程序证明了一点，在尖括号中打印出了接收和发送的时间戳值（在关闭高位之后）。另外，不能计算发起时间戳和接收时间戳之间的时间差，因为它们的单位不一致。

```
sun % icmptime gateway
orig = 83620811, recv = <4871036>, xmit = <4871036>, rtt = 220 ms

sun % icmptime gateway
orig = 83641007, recv = <4891232>, xmit = <4891232>, rtt = 213 ms
```

如果我们在这台主机上运行该程序数次，会发现时间戳值显然具有毫秒级的分辨率，而且是从某个起始点开始计算的毫秒数，但是起始点并不是午夜UTC（例如，可能是从路由器引导时开始计数的毫秒数）。

作为最后一个例子，我们来比较sun主机和另一个已知是准确的系统时钟——一个NTP stratum 1服务器（下面我们会更多地讨论NTP，网络时间协议）。

```
sun % icmptime clock.llnl.gov
orig = 83662791, recv = 83662919, xmit = 83662919, rtt = 359 ms
difference = 128 ms

sun % icmptime clock.llnl.gov
orig = 83670425, recv = 83670559, xmit = 83670559, rtt = 345 ms
difference = 134 ms
```

如果我们把difference的值减去RTT的一半，结果表明sun主机上的时钟要快38.5～51.5 ms。

6.4.2 另一种方法

还可以用另一种方法来获得时间和日期。

1) 在1.12节中描述了日期服务程序和时间服务程序。前者是以人们可读的格式返回当前的时间和日期，是一行ASCII字符。可以用telnet命令来验证这个服务：

```
sun % telnet bsdi daytime
Trying 140.252.13.35 ...
Connected to bsdi.
Escape character is '^]'.                前三行是Telnet客户的输出
Wed Feb  3 16:38:33 1993                 这是日期时间服务器的输出
Connection closed by foreign host.       这也是Telnet客户的输出
```

另一方面，时间服务程序返回的是一个32bit的二制进数值，表示自UTC，1900年1月1日午夜起算的秒数。这个程序是以秒为单位提供的日期和时间（前面我们提过的rdate命令使用的是TCP时间服务程序）。

2) 严格的计时器使用网络时间协议（NTP），该协议在RFC 1305中给出了描述[Mills 1992]。这个协议采用先进的技术来保证LAN或WAN上的一组系统的时钟误差在毫秒级以内。对计算机精确时间感兴趣的读者应该阅读这份RFC文档。

3) 开放软件基金会（OSF）的分布式计算环境（DCE）定义了分布式时间服务（DTS），

它也提供计算机之间的时钟同步。文献 [Rosenberg, Kenney and Fisher 1992]提供了该服务的其他细节描述。

4) 伯克利大学的Unix系统提供守护程序 `timed`(8)，来同步局域网上的系统时钟。不像NTP和DTS，`timed`不在广域网范围内工作。

6.5 ICMP端口不可达差错

最后两小节我们来讨论ICMP查询报文 —— 地址掩码和时间戳查询及应答。现在来分析一种ICMP差错报文，即端口不可达报文，它是 ICMP目的不可到达报文中的一种，以此来看一看ICMP差错报文中所附加的信息。使用UDP（见第11章）来查看它。

UDP的规则之一是，如果收到一份UDP数据报而目的端口与某个正在使用的进程不相符，那么UDP返回一个ICMP不可达报文。可以用TFTP来强制生成一个端口不可达报文（ TFTP将在第15章描述）。

对于TFTP服务器来说， UDP的公共端口号是 69。但是大多数的 TFTP客户程序允许用`connect`命令来指定一个不同的端口号。这里，我们就用它来指定 8888端口：

```
bsdi % tftp
tftp> connect svr4 8888                指定主机名和端口号
tftp> get temp.foo                     试图得到一个文件
Transfer timed out.                    大约25秒后
tftp> quit
```

`connect`命令首先指定要连接的主机名及其端口号，接着用 get命令来取文件。敲入 get命令后，一份UDP数据报就发送到主机svr4上的8888端口。tcpdump命令引起的报文交换结果如图6-8所示。

```
 1   0.0                  arp who-has svr4 tell bsdi
 2   0.002050 (0.0020)    arp reply svr4 is-at 0:0:c0:c2:9b:26

 3   0.002723 (0.0007)    bsdi.2924 > svr4.8888: udp 20
 4   0.006399 (0.0037)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

 5   5.000776 (4.9944)    bsdi.2924 > svr4.8888: udp 20
 6   5.004304 (0.0035)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

 7  10.000887 (4.9966)    bsdi.2924 > svr4.8888: udp 20
 8  10.004416 (0.0035)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

 9  15.001014 (4.9966)    bsdi.2924 > svr4.8888: udp 20
10  15.004574 (0.0036)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

11  20.001177 (4.9966)    bsdi.2924 > svr4.8888: udp 20
12  20.004759 (0.0036)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable
```

图6-8 由TFTP产生的ICMP端口不可达差错

在UDP数据报送到svr4之前，要先发送一份ARP请求来确定它的硬件地址（第1行）。接着返回ARP应答（第2行），然后才发送UDP数据报（第3行）（在`tcpdump`的输出中保留ARP请求和应答是为了提醒我们，这些报文交换可能在第一个 IP数据报从一个主机发送到另一个主机之前是必需的。在本书以后的章节中，如果这些报文与讨论的题目不相关，那么我们将省略它们）。

一个ICMP端口不可达差错是立刻返回的（第 4行）。但是，TFTP客户程序看上去似乎忽略了这个ICMP报文，而在5秒钟之后又发送了另一份 UDP数据报（第5行）。在客户程序放弃

之前重发了三次。

注意，ICMP报文是在主机之间交换的，而不用目的端口号，而每个 20字节的UDP数据报则是从一个特定端口（2924）发送到另一个特定端口（8888）。

跟在每个UDP后面的数字20指的是UDP数据报中的数据长度。在这个例子中， 20字节包括TFTP的2个字节的操作代码， 9个字节以空字符结束的文件名 `temp.foo`，以及9个字节以空字符结束的字符串`netascii`（TFTP报文的详细格式参见图15-1）。

如果用-e选项运行同样的例子，我们可以看到每个返回的 ICMP端口不可达报文的完整长度。这里的长度为70字节，各字段分配如图6-9所示。

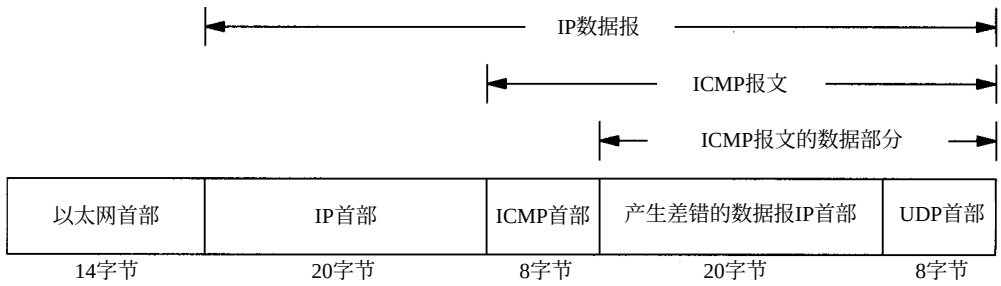


图6-9 “UDP端口不可达”例子中返回的ICMP报文

ICMP的一个规则是， ICMP差错报文（参见图6-3的最后一列）必须包括生成该差错报文的数据报IP首部（包含任何选项），还必须至少包括跟在该 IP首部后面的前8个字节。在我们的例子中，跟在IP首部后面的前8个字节包含UDP的首部（见图11-2）。

一个重要的事实是包含在 UDP首部中的内容是源端口号和目的端口号。就是由于目的端口号（8888）才导致产生了ICMP端口不可达的差错报文。接收 ICMP的系统可以根据源端口号（2924）来把差错报文与某个特定的用户进程相关联（在本例中是 TFTP客户程序）。

导致差错的数据报中的 IP首部要被送回的原因是因为 IP首部中包含了协议字段，使得ICMP可以知道如何解释后面的 8个字节（在本例中是 UDP首部）。如果我们来查看 TCP首部（图17-2）,可以发现源端口和目的端口被包含在 TCP首部的前8个字节中。

ICMP不可达报文的一般格式如图6-10所示。

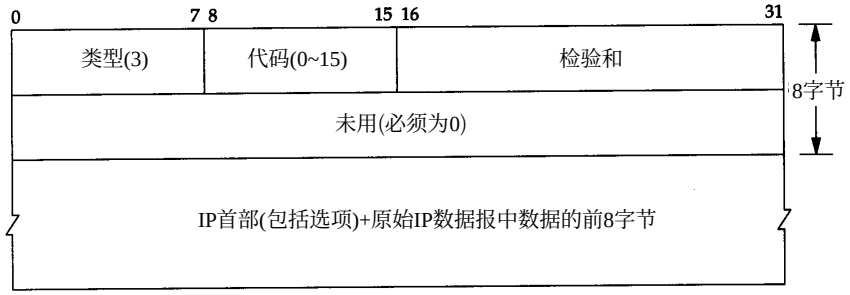


图6-10 ICMP不可达报文

在图6-3中，我们注意到有16种不同类型的ICMP不可达报文，代码分别从0到15。ICMP端口不可达差错代码是3。另外，尽管图6-10指出了在ICMP报文中的第二个32 bit字必须为0，但是当代码为4时（“需要分片但设置了不分片比特”），路径MTU发现机制（2.9节）却允许路由器把外

出接口的MTU填在这个32 bit字的低16 bit中。我们在11.6节中给出了一个这种差错的例子。

尽管ICMP规则允许系统返回多于8个字节的产生差错的IP数据报中的数据，但是大多数从伯克利派生出来的系统只返回 8个字节。 Solaris 2.2的ip_icmp_return_data_bytes选项默认条件下返回前64个字节（E.4节）。

tcpdump时间系列

在本书的后面章节中，我们还要以时间系列的格式给出tcpdump命令的输出，如图6-11所示。

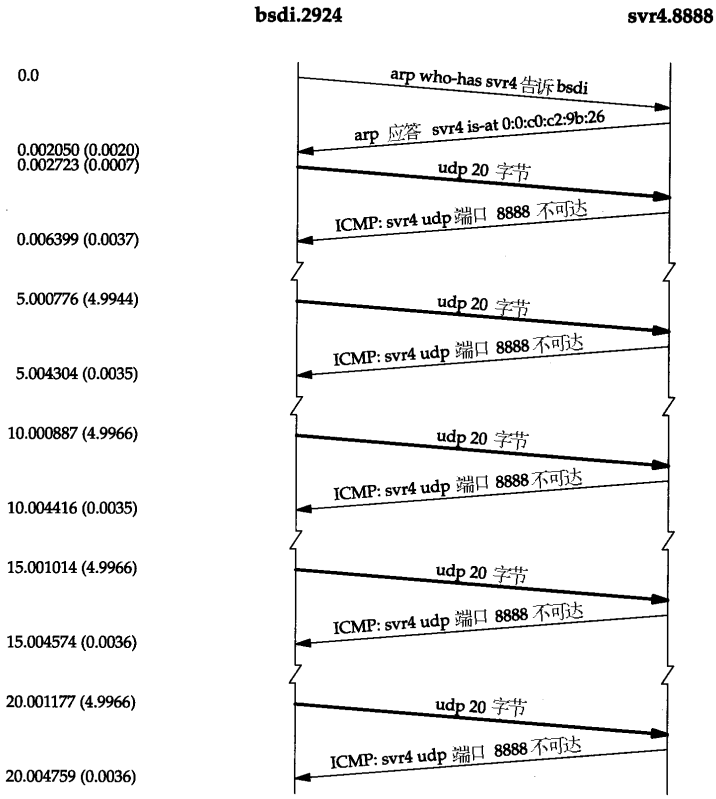


图6-11 发送到无效端口的TFTP请求的时间系列

时间随着向下而递增，在图左边的时间标记与 tcpdump命令的输出是相同的（见图 6-8）。位于图顶部的标记是通信双方的主机名和端口号。需要指出的是，随着页面向下的y坐标轴与真正的时间值不是成比例的。当出现一个有意义的时间段时，在本例中是每 5 秒之间的重发，我们就在时间系列的两侧作上标记。当UDP或TCP数据正在被传送时，我们用粗线的行来表示。

当ICMP报文返回时，为什么 TFTP客户程序还要继续重发请求呢？这是由于网络编程中的一个因素，即BSD系统不把从插口(socket)接收到的ICMP报文中的UDP数据通知用户进程，除非该进程已经发送了一个 connect命令给该插口。标准的 BSD TFTP客户程序并不发送connect命令，因此它永远也不会收到ICMP差错报文的通知。

这里需要注意的另一点是 TFTP客户程序所采用的不太好的超时重传算法。它只是假定 5秒是足够的，因此每隔5秒就重传一次，总共需要 25秒钟的时间。在后面我们将看到 TCP有一个较好的超时重发算法。

TFTP客户程序所采用的超时重传算法已被RFC所禁用。不过，在作者所在子网上的三个系统以及Solaris 2.2仍然在使用它。AIX 3.2.2采用一种指数退避方法来设置超时值，分别在0、5、15和35秒时重发报文，这正是所推荐的方法。我们将在第21章更详细地讨论超时问题。

最后需要指出的是，ICMP报文是在发送UDP数据报3.5 ms后返回的，这与第7章我们所看到的Ping应答的往返时间差不多。

6.6 ICMP报文的4.4BSD处理

由于ICMP覆盖的范围很广，从致命差错到信息差错，因此即使在一个给定的系统实现中，对每个ICMP报文的处理都是不相同的。图6-12的内容与图6-3相同，它显示的是4.4BSD系统对每个可能的ICMP报文的处理方法。

类 型	代 码	描 述	处 理 方 法
0	0	回显应答	用户进程
3		目的不可达：	
	0	网络不可达	“无路由到达主机”
	1	主机不可达	“无路由到达主机”
	2	协议不可达	“连接被拒绝”
	3	端口不可达	“连接被拒绝”
	4	需要进行分片但设置了不分片比特DF	“报文太长”
	5	源站选路失败	“无路由到达主机”
	6	目的网络不认识	“无路由到达主机”
	7	目的主机不认识	“无路由到达主机”
	8	源主机被隔离（作废不用）	“无路由到达主机”
	9	目的网络被强制禁止	“无路由到达主机”
	10	目的主机被强制禁止	“无路由到达主机”
	11	由于服务类型TOS，网络不可达	“无路由到达主机”
	12	由于服务类型TOS，主机不可达	“无路由到达主机”
	13	由于过滤，通信被强制禁止	（忽略）
	14	主机越权	（忽略）
	15	优先权中止生效	（忽略）
4	0	源站被抑制(quench)	TCP由内核处理，UDP则忽略
5		重定向	
	0	对网络重定向	内核更新路由表
	1	对主机重定向	内核更新路由表
	2	对服务类型和网络重定向	内核更新路由表
	3	对服务类型和主机重定向	内核更新路由表
8	0	回显请求	
9	0	路由器通告	用户进程
10	0	路由器请求	用户进程
11		超时：	
	0	传输期间生存时间为0	用户进程
	1	在数据报组装期间生存时间为0	用户进程
12		参数问题：	
	0	坏的IP首部（包括各种差错）	“协议不可用”
	1	缺少必需的选项	“协议不可用”
13	0	时间戳请求	内核产生应答
14	0	时间戳应答	用户进程
15	0	信息请求（作废不用）	（忽略）
16	0	信息应答（作废不用）	用户进程
17	0	地址掩码请求	内核产生应答
18	0	地址掩码应答	用户进程

图6-12 4.4BSD系统对ICMP报文的处理

如果最后一列标明是“内核”，那么ICMP就由内核来处理。如果最后一列指明是“用户进程”，那么报文就被传送到所有在内核中登记的用户进程，以读取收到的ICMP报文。如果不存在任何这样的用户进程，那么报文就悄悄地被丢弃（这些用户进程还会收到所有其他类型的ICMP报文的拷贝，虽然它们应该由内核来处理，当然用户进程只有在内核处理以后才能收到这些报文）。有一些报文完全被忽略。最后，如果最后一列标明的是引号内的一串字符，那么它就是对应的Unix差错。其中一些差错，如TCP对发送端关闭的处理等，我们将在以后的章节中对它们进行讨论。

6.7 小结

本章对每个系统都必须包括的Internet控制报文协议进行了讨论。图6-3列出了所有的ICMP报文类型，其中大多数都将在以后的章节中加以讨论。

我们详细讨论了ICMP地址掩码请求和应答以及时间戳请求和应答。这些是典型的请求—应答报文。二者在ICMP报文中都有标识符和序列号。发送端应用程序在标识字段内存入一个唯一的数值，以区别于其他进程的应答。序列号字段使得客户程序可以在应答和请求之间进行匹配。

我们还讨论了ICMP端口不可达差错，一种常见的ICMP差错。对返回的ICMP差错信息进行了分析：导致差错的IP数据报的首部及后续8个字节。这个信息对于ICMP差错的接收方来说是必要的，可以更多地了解导致差错的原因。这是因为TCP和UDP都在它们的首部前8个字节中存入源端口号和目的端口号。

最后，我们第一次给出了按时间先后的`tcpdump`输出，这种表示方式在本书后面的章节中会经常用到。

习题

6.1 在6.2节的末尾，我们列出了5种不发送ICMP差错报文的特殊条件。如果这些条件不满足而我们又在局域网上向一个似乎不存在的端口号发送一份广播UDP数据报，这时会发生什么样的情况？

6.2 阅读RFC [Braden 1989a]，注意生成一个ICMP端口不可达差错是否为“必须”，“应该”或者“可能”。这些信息所在的页码和章节是多少？

6.3 阅读RFC 1349 [Almquist 1992]，看看IP的服务类型字段（见图3-2）是如何被ICMP设置的？

6.4 如果你的系统提供`netstat`命令，请用它来查看接收和发送的ICMP报文类型。

第7章 Ping程序

7.1 引言

“ping”这个名字源于声纳定位操作。Ping程序由Mike Muuss编写，目的是为了测试另一台主机是否可达。该程序发送一份ICMP回显请求报文给主机，并等待返回ICMP回显应答（图6-3列出了所有的ICMP报文类型）。

一般来说，如果不能Ping到某台主机，那么就不能Telnet或者FTP到那台主机。反过来，如果不能Telnet到某台主机，那么通常可以用Ping程序来确定问题出在哪里。Ping程序还能测出到这台主机的往返时间，以表明该主机离我们有“多远”。

在本章中，我们将使用Ping程序作为诊断工具来深入剖析ICMP。Ping还给我们提供了检测IP记录路由和时间戳选项的机会。文献[Stevens 1990]的第11章提供了Ping程序的源代码。

> 几年前我们还可以作出这样没有限定的断言，如果不能Ping到某台主机，那么就不能Telnet或FTP到那台主机。随着Internet安全意识的增强，出现了提供访问控制清单的路由器和防火墙，那么像这样没有限定的断言就不再成立了。一台主机的可达性可能不只取决于IP层是否可达，还取决于使用何种协议以及端口号。Ping程序的运行结果可能显示某台主机不可达，但我们可以用Telnet远程登录到该台主机的25号端口（邮件服务器）。

7.2 Ping程序

我们称发送回显请求的ping程序为客户，而称被ping的主机为服务器。大多数的TCP/IP实现都在内核中直接支持Ping服务器——这种服务器不是一个用户进程（在第6章中描述的两种ICMP查询服务，地址掩码和时间戳请求，也都是直接在内核中进行处理的）。

ICMP回显请求和回显应答报文如图7-1所示。

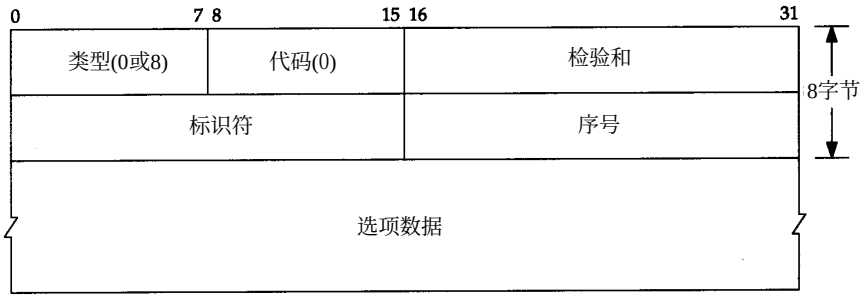


图7-1 ICMP回显请求和回显应答报文格式

对于其他类型的ICMP查询报文，服务器必须响应标识符和序列号字段。另外，客户发送的选项数据必须回显，假设客户对这些信息都会感兴趣。

Unix系统在实现ping程序时是把ICMP报文中的标识符字段置成发送进程的 ID号。这样即使在同一台主机上同时运行了多个ping程序实例，ping程序也可以识别出返回的信息。

序列号从0开始，每发送一次新的回显请求就加 1。ping程序打印出返回的每个分组的序列号，允许我们查看是否有分组丢失、失序或重复。 IP是一种最好的数据报传递服务，因此这三个条件都有可能发生。

旧版本的ping程序曾经以这种模式运行，即每秒发送一个回显请求，并打印出返回的每个回显应答。但是，新版本的实现需要加上- s选项才能以这种模式运行。默认情况下，新版本的ping程序只发送一个回显请求。如果收到回显应答，则输出“ host is alive ”；否则，在20秒内没有收到应答就输出“ no answer（没有回答）”。

7.2.1 LAN输出

在局域网上运行ping程序的结果输出一般有如下格式：

```
bsdi % ping svr4
PING svr4 (140.252.13.34): 56 data bytes
64 bytes from 140.252.13.34: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=2 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=3 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=4 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=5 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=6 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=7 ttl=255 time=0 ms
^?                                    键入中断键来停止显示
--- svr4 ping statistics ---
8 packets transmitted, 8 packets received, 0% packet loss
round-trip min/avg/max = 0/0/0 ms
```

当返回ICMP回显应答时，要打印出序列号和 TTL，并计算往返时间（TTL位于IP首部中的生存时间字段。当前的 BSD系统中的ping程序每次收到回显应答时都打印出收到的 TTL——有些系统并不这样做。我们将在第 8章中通过traceroute程序来介绍TTL的用法）。

从上面的输出中可以看出，回显应答是以发送的次序返回的（ 0，1，2等）。

ping程序通过在ICMP报文数据中存放发送请求的时间值来计算往返时间。当应答返回时，用当前时间减去存放在 ICMP报文中的时间值，即是往返时间。注意，在发送端 bsdi上，往返时间的计算结果都为 0 ms。这是因为程序使用的计时器分辨率低的原因。 BSD/386版本0.9.4系统只能提供10 ms级的计时器（在附录B中有更详细的介绍）。在后面的章节中，当我们在具有较高分辨率计时器的系统上（ Sun）查看tcpdump输出时会发现， ICMP回显请求和回显应答的时间差在4 ms以下。

输出的第一行包括目的主机的 IP地址，尽管指定的是它的名字（ svr4）。这说明名字已经经过解析器被转换成 IP地址了。我们将在第 14章介绍解析器和 DNS。现在，我们发现，如果敲入ping命令，几秒钟过后会在第 1行打印出IP地址， DNS就是利用这段时间来确定主机名所对应的IP地址。

本例中的tcpdump输出如图7-2所示。

从发送回显请求到收到回显应答，时间间隔始终为 3.7 ms。还可以看到，回显请求大约每隔1秒钟发送一次。

通常，第1个往返时间值要比其他的大。这是由于目的端的硬件地址不在 ARP高速缓存中

```
 1   0.0                 bsdi > svr4: icmp: echo request
 2   0.003733 (0.0037)   svr4 > bsdi: icmp: echo reply

 3   0.998045 (0.9943)   bsdi > svr4: icmp: echo request
 4   1.001747 (0.0037)   svr4 > bsdi: icmp: echo reply

 5   1.997818 (0.9961)   bsdi > svr4: icmp: echo request
 6   2.001542 (0.0037)   svr4 > bsdi: icmp: echo reply

 7   2.997610 (0.9961)   bsdi > svr4: icmp: echo request
 8   3.001311 (0.0037)   svr4 > bsdi: icmp: echo reply

 9   3.997390 (0.9961)   bsdi > svr4: icmp: echo request
10   4.001115 (0.0037)   svr4 > bsdi: icmp: echo reply

11   4.997201 (0.9961)   bsdi > svr4: icmp: echo request
12   5.000904 (0.0037)   svr4 > bsdi: icmp: echo reply

13   5.996977 (0.9961)   bsdi > svr4: icmp: echo request
14   6.000708 (0.0037)   svr4 > bsdi: icmp: echo reply

15   6.996764 (0.9961)   bsdi > svr4: icmp: echo request
16   7.000479 (0.0037)   svr4 > bsdi: icmp: echo reply
```

图7-2 在LAN上运行ping程序的结果

的缘故。正如我们在第4章中看到的那样，在发送第一个回显请求之前要发送一个 ARP请求并接收ARP应答，这需要花费几毫秒的时间。下面的例子说明了这一点：

```
sun % arp -a                                    保证ARP高速缓存是空的
sun % ping svr4
PING svr4: 56 data bytes
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=7. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=4. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=2. time=4. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=3. time=4. ms
^?                                              键入中断键来停止显示
----svr4 PING Statistics----
4 packets transmitted, 4 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 4/4/7
```

第1个RTT中多出的3 ms很可能就是因为发送ARP请求和接收ARP应答所花费的时间。

这个例子运行在sun主机上，它提供的是具有微秒级分辨率的计时器，但是 ping程序只能打印出毫秒级的往返时间。在前面运行于 BSD/386 0.9.4版上的例子中，打印出来的往返时间值为0 ms，这是因为计时器只能提供10 ms的误差。下面的例子是BSD/386 1.0版的输出，它提供的计时器也具有微秒级的分辨率，因此， ping程序的输出结果也具有较高的分辨率。

```
bsdi % ping svr4
PING svr4 (140.252.13.34): 56 data bytes
64 bytes from 140.252.13.34: icmp_seq=0 ttl=255 time=9.304 ms
64 bytes from 140.252.13.34: icmp_seq=1 ttl=255 time=6.089 ms
64 bytes from 140.252.13.34: icmp_seq=2 ttl=255 time=6.079 ms
64 bytes from 140.252.13.34: icmp_seq=3 ttl=255 time=6.096 ms
^?                                              键入中断键来停止显示
--- svr4 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 6.079/6.880/9.304 ms
```

7.2.2 WAN输出

在一个广域网上，结果会有很大的不同。下面的例子是在某个工作日的下午即 Internet具

有正常通信量时的运行结果：

```
gemini % ping vangogh.cs.berkeley.edu
PING vangogh.cs.berkeley.edu: 56 data bytes
64 bytes from (128.32.130.2): icmp_seq=0. time=660. ms
64 bytes from (128.32.130.2): icmp_seq=5. time=1780. ms
64 bytes from (128.32.130.2): icmp_seq=7. time=380. ms
64 bytes from (128.32.130.2): icmp_seq=8. time=420. ms
64 bytes from (128.32.130.2): icmp_seq=9. time=390. ms
64 bytes from (128.32.130.2): icmp_seq=14. time=110. ms
64 bytes from (128.32.130.2): icmp_seq=15. time=170. ms
64 bytes from (128.32.130.2): icmp_seq=16. time=100. ms
^?                                    键入中断来停止显示
----vangogh.CS.Berkeley.EDU PING Statistics----
17 packets transmitted, 8 packets received, 52% packet loss
round-trip (ms)  min/avg/max = 100/501/1780
```

这里，序列号为1、2、3、4、6、10、11、12和13的回显请求或回显应答在某个地方丢失了。另外，我们注意到往返时间发生了很大的变化（像 52%这样高的分组丢失率是不正常的。即使是在工作日的下午，对于Internet来说也是不正常的）。

通过广域网还有可能看到重复的分组（即相同序列号的分组被打印两次或更多次），失序的分组（序列号为$N+1$的分组在序列号为N的分组之前被打印）。

7.2.3 线路SLIP链接

让我们再来看看SLIP链路上的往返时间，因为它们经常运行于低速的异步方式，如 9600 b/s或更低。回想我们在 2.10节计算的串行线路吞吐量。针对这个例子，我们把主机 `bsdi`和`slip`之间的SLIP链路传输速率设置为1200 b/s。

下面我们可以来估计往返时间。首先，从前面的 Ping程序输出例子中可以注意到，默认情况下发送的ICMP报文有56个字节。再加上20个字节的IP首部和8个字节的ICMP首部，IP数据报的总长度为84字节（我们可以运行 `tcpdump -e`命令查看以太网数据帧来验证这一点）。另外，从2.4节可以知道，至少要增加两个额外的字节：在数据报的开始和结尾加上 END字符。此外，SLIP帧还有可能再增加一些字节，但这取决于数据报中每个字节的值。对于 1200 b/s这个速率来说，由于每个字节含有 8 bit数据、1 bit起始位和1 bit结束位，因此传输速率是每秒120个字节，或者说每个字节 8.33 ms。所以我们可以估计需要 1433（86×8.33×2）ms（乘2是因为我们计算的是往返时间）。

下面的输出证实了我们的计算：

```
svr4 % ping -s slip
PING slip: 56 data bytes
64 bytes from slip (140.252.13.65): icmp_seq=0. time=1480. ms
64 bytes from slip (140.252.13.65): icmp_seq=1. time=1480. ms
64 bytes from slip (140.252.13.65): icmp_seq=2. time=1480. ms
64 bytes from slip (140.252.13.65): icmp_seq=3. time=1480. ms
^?
----slip PING Statistics----
5 packets transmitted, 4 packets received, 20% packet loss
round-trip (ms)  min/avg/max = 1480/1480/1480
```

（对于SVR4来说，如果每秒钟发送一次请求则必须带 `-s`选项）。往返时间大约是 1.5秒，但是程序仍然每间隔 1秒钟发送一次 ICMP回显请求。这说明在第 1个回显应答返回之前（1.480秒时刻）就已经发送了两次回显请求（分别在 0秒和1秒时刻）。这就是为什么总结行指

出丢失了一个分组。实际上分组并未丢失，很可能仍然在返回的途中。

我们在第8章讨论traceroute程序时将回头再讨论这种低速的SLIP链路。

7.2.4 拨号SLIP链路

对于拨号SLIP链路来说，情况有些变化，因为在链路的两端增加了调制解调器。用在sun和netb系统之间的调制解调器提供的是V.32调制方式（9600 b/s）、V.42错误控制方式（也称作LAP-M）以及V.42bis数据压缩方式。这表明我们针对线路链路参数进行的简单计算不再准确了。

很多因素都有可能影响。调制解调器带来了时延。随着数据的压缩，分组长度可能会减小，但是由于使用了错误控制协议，分组长度又可能会增加。另外，接收端的调制解调器只能在验证了循环检验字符（检验和）后才能释放收到的数据。最后，我们还要处理每一端的计算机异步串行接口，许多操作系统只能在固定的时间间隔内，或者收到若干字符后才去读这些接口。

作为一个例子，我们在sun主机上ping主机gemini，输出结果如下：

```
sun % ping gemini
PING gemini: 56 data bytes
64 bytes from gemini (140.252.1.11): icmp_seq=0. time=373. ms
64 bytes from gemini (140.252.1.11): icmp_seq=1. time=360. ms
64 bytes from gemini (140.252.1.11): icmp_seq=2. time=340. ms
64 bytes from gemini (140.252.1.11): icmp_seq=3. time=320. ms
64 bytes from gemini (140.252.1.11): icmp_seq=4. time=330. ms
64 bytes from gemini (140.252.1.11): icmp_seq=5. time=310. ms
64 bytes from gemini (140.252.1.11): icmp_seq=6. time=290. ms
64 bytes from gemini (140.252.1.11): icmp_seq=7. time=300. ms
64 bytes from gemini (140.252.1.11): icmp_seq=8. time=280. ms
64 bytes from gemini (140.252.1.11): icmp_seq=9. time=290. ms
64 bytes from gemini (140.252.1.11): icmp_seq=10. time=300. ms
64 bytes from gemini (140.252.1.11): icmp_seq=11. time=280. ms
----gemini PING Statistics----
12 packets transmitted, 12 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 280/314/373
```

注意，第1个RTT不是10 ms的整数倍，但是其他行都是10 ms的整数倍。如果我们运行该程序若干次，发现每次结果都是这样（这并不是由sun主机上的时钟分辨率造成的结果，因为根据附录B中的测试结果可以知道它的时钟能提供毫秒级的分辨率）。

另外还要注意，第1个RTT要比其他的大，而且依次递减，然后徘徊在280～300 ms之间。我们让它运行1~2分钟，RTT一直处于这个范围，不会低于260 ms。如果我们以9600 b/s的速率计算RTT（习题7.2），那么观察到的值应该大约是估计值的1.5倍。

如果运行ping程序60秒钟并计算观察到的RTT的平均值，我们发现在V.42和V.42bis模式下平均值为277 ms（这比上个例子打印出来的平均值要好，因为运行时间较长，这样就把开始较长的时间平摊了）。如果我们关闭V.42bis数据压缩方式，平均值为330 ms。如果我们关闭V.42错误控制方式（它同时也关闭了V.42bis数据压缩方式），平均值为300 ms。这些调制解调器的参数对RTT的影响很大，使用错误控制和数据压缩方式似乎效果最好。

7.3 IP记录路由选项

ping程序为我们提供了查看IP记录路由（RR）选项的机会。大多数不同版本的ping程

序都提供-R选项，以提供记录路由的功能。它使得ping程序在发送出去的IP数据报中设置IP RR选项（该IP数据报包含ICMP回显请求报文）。这样，每个处理该数据报的路由器都把它的IP地址放入选项字段中。当数据报到达目的端时，IP地址清单应该复制到ICMP回显应答中，这样返回途中所经过的路由器地址也被加入清单中。当ping程序收到回显应答时，它就打印出这份IP地址清单。

这个过程听起来简单，但存在一些缺陷。源端主机生成RR选项，中间路由器对RR选项的处理，以及把ICMP回显请求中的RR清单复制到ICMP回显应答中，所有这些都是选项功能。幸运的是，现在的大多数系统都支持这些选项功能，只是有一些系统不把ICMP请求中的IP清单复制到ICMP应答中。

但是，最大的问题是IP首部中只有有限的空间来存放IP地址。我们从图3-1可以看到，IP首部中的首部长度字段只有4 bit，因此整个IP首部最长只能包括15个32 bit长的字（即60个字节）。由于IP首部固定长度为20字节，RR选项用去3个字节（下面我们再讨论），这样只剩下37个字节（60-20-3）来存放IP地址清单，也就是说只能存放9个IP地址。对于早期的ARPANET来说，9个IP地址似乎是很多了，但是现在看来是非常有限的（在第8章中，我们将用Traceroute工具来确定数据报的路由）。除了这些缺点，记录路由选项工作得很好，为详细查看如何处理IP选项提供了一个机会。

IP数据报中的RR选项的一般格式如图7-3所示。

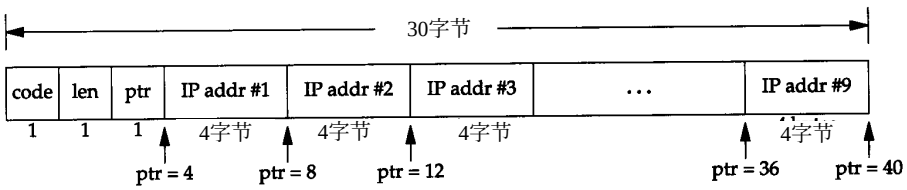


图7-3 IP首部中的记录路由选项的一般格式

code是一个字节，指明IP选项的类型。对于RR选项来说，它的值为7。len是RR选项总字节长度，在这种情况下为39（尽管可以为RR选项设置比最大长度小的长度，但是ping程序总是提供39字节的选项字段，最多可以记录9个IP地址。由于IP首部中留给选项的空间有限，它一般情况都设置成最大长度）。

ptr称作指针字段。它是一个基于1的指针，指向存放下一个IP地址的位置。它的最小值为4，指向存放第一个IP地址的位置。随着每个IP地址存入清单，ptr的值分别为8，12，16，最大到36。当记录下9个IP地址后，ptr的值为40，表示清单已满。

当路由器（根据定义应该是多穴的）在清单中记录IP地址时，它应该记录哪个地址呢？是入口地址还是出口地址？为此，RFC 791 [Postel 1981a]指定路由器记录出口IP地址。我们在后面将看到，当原始主机（运行ping程序的主机）收到带有RR选项的ICMP回显应答时，它也要把它的入口IP地址放入清单中。

7.3.1 通常的例子

我们举一个用RR选项运行ping程序的例子，在主机svr4上运行ping程序到主机slip。一个中间路由器(bsdi)将处理这个数据报。下面是svr4的输出结果：

```
svr4 % ping -R slip
PING slip (140.252.13.65): 56 data bytes
64 bytes from 140.252.13.65: icmp_seq=0 ttl=254 time=280 ms
RR:     bsdi (140.252.13.66)
        slip (140.252.13.65)
        bsdi (140.252.13.35)
        svr4 (140.252.13.34)
64 bytes from 140.252.13.65: icmp_seq=1 ttl=254 time=280 ms (same route)
64 bytes from 140.252.13.65: icmp_seq=2 ttl=254 time=270 ms (same route)
^?
--- slip ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 270/276/280 ms
```

分组所经过的四站如图7-4所示（每个方向各有两站），每一站都把自己的IP地址加入RR清单。

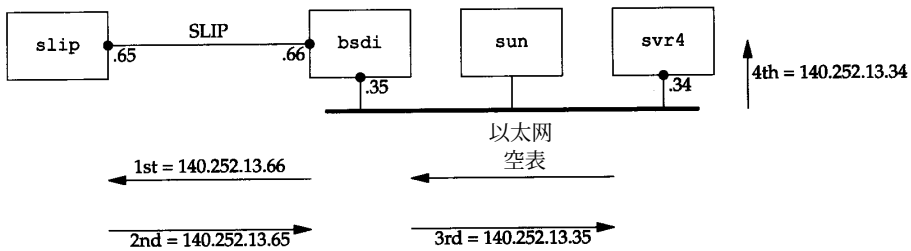


图7-4 带有记录路由选项的ping程序

路由器bsdi在不同方向上分别加入了不同的IP地址。它始终是把出口的IP地址加入清单。我们还可以看到，当ICMP回显应答到达原始系统（svr4）时，它把自己的入口IP地址也加入清单中。

还可以通过运行带有-v选项的tcpdump命令来查看主机sun上进行的分组交换（参见IP选项）。输出如图7-5所示。

```
1   0.0                  svr4 > slip: icmp: echo request (ttl 32, id 35835,
                         optlen=40 RR{39}= RR{#0.0.0.0/0.0.0.0/0.0.0.0/
                         0.0.0.0/ 0.0.0.0/0.0.0.0/0.0.0.0/0.0.0.0/0.0.0.0} EOL)

2   0.267746 (0.2677)    slip > svr4: icmp: echo reply (ttl 254, id 1976,
                         optlen=40 RR{39}= RR{140.252.13.66/140.252.13.65/
                         140.252.13.35/#0.0.0.0/0.0.0.0/0.0.0.0/0.0.0.0/
                         0.0.0.0/0.0.0.0} EOL)
```

图7-5 记录路由选项的tcpdump 输出

输出中optlen=40表示在IP首部中有40个字节的选项空间（IP首部长度必须为4字节的整数倍）。RR{39}的意思是记录路由选项已被设置，它的长度字段是39。然后是9个IP地址，符号“#”用来标记RR选项中的ptr字段所指向的IP地址。由于我们是在主机sun上观察这些分组（参见图7-4），因此所能看到ICMP回显请求中的IP地址清单是空的，而ICMP回显应答中有3个IP地址。我们省略了tcpdump输出中的其他行，因为它们与图7-5基本一致。

位于路由信息末尾的标记EOL表示IP选项“end of list（清单结束）”的值。EOL选项的值可以为0。这时表示39个字节的RR数据位于IP首部中的40字节空间中。由于在数据报发送之前空间选项被设置为0，因此跟在39个字节的RR数据之后的0字符就被解释为EOL。这正是我

们所希望的结果。如果在IP首部中的选项字段中有多个选项，在开始下一个选项之前必须填入空白字符，另外还可以用另一个值为1的特殊字符NOP（“no operation”）。

在图7-5中，SVR4把回显请求中的TTL字段设为32，BSD/386设为255（它打印出的值为254是因为路由器bsdi已经将其减去1）。新的系统都把ICMP报文中的TTL设为最大值（255）。

在作者使用的三个TCP/IP系统中，BSD/386和SVR4都支持记录路由选项。这就是说，当转发数据报时，它们都能正确地更新RR清单，而且能正确地把接收到的ICMP回显请求中的RR清单复制到出口ICMP回显应答中。虽然SunOS 4.1.3在转发一个数据报时能正确更新RR清单，但是不能复制RR清单。Solaris 2.x对这个问题已作了修改。

7.3.2 异常的输出

下面的例子是作者观察到的，把它作为第 9章讨论ICMP间接报文的起点。在子网140.252.1上ping主机aix（在主机sun上通过拨号SLIP连接可以访问），并带有记录路由选项。在slip主机上运行有如下输出结果：

```
slip % ping -R aix
PING aix (140.252.1.92): 56 data bytes
64 bytes from 140.252.1.92: icmp_seq=0 ttl=251 time=650 ms
RR:     bsdi (140.252.13.35)
        sun (140.252.1.29)
        netb (140.252.1.183)
        aix (140.252.1.92)
        gateway (140.252.1.4)          为什么用这个路由器？
        netb (140.252.1.183)
        sun (140.252.13.33)
        bsdi (140.252.13.66)
        slip (140.252.13.65)
64 bytes from aix: icmp_seq=1 ttl=251 time=610 ms (same route)
64 bytes from aix: icmp_seq=2 ttl=251 time=600 ms (same route)
^?
--- aix ping statistics ---
4 packets transmitted, 3 packets received, 25% packet loss
round-trip min/avg/max = 600/620/650 ms
```

我们已经在主机bsdi上运行过这个例子。现在选择slip来运行它，观察RR清单中所有的9个IP地址。

在输出中令人感到疑惑的是，为什么传出的数据报（ICMP回显请求）直接从netb传到aix，而返回的数据报（ICMP回显应答）却从aix开始经路由器gateway再到netb？这里看到的正是下面将要描述的IP选路的一个特点。数据报经过的路由如图7-6所示。

问题是aix不知道要把目的地为子网140.252.13的IP数据报发到主机netb上。相反，aix在它的路由表中有一个默认项，它指明当没有明确某个目的主机的路由时，就把所有的数据报发往默认项指定的路由器gateway。路由器gateway比子网140.252.1上的任何主机都具备更强的选路能力（在这个以太网上有超过150台主机，每台主机的路由表中都有一个默认项指向路由器gateway，这样就不用在每台主机上都运行一个选路守护程序）。

这里没有应答的一个问题是为什么gateway不直接发送ICMP报文重定向到aix（9.5节），以更新它的路由表？由于某种原因（很可能是由于数据报产生的重定向是一份ICMP回显请求报文），重定向并没有产生。但是如果我们用Telnet登录到aix上的daytime服务器，ICMP就会

产生重定向，因而它在aix上的路由表也随之更新。如果接着执行ping程序并带有记录路由选项，其路由显示表明数据报从netb到aix，然后返回netb，而不再经过路由器gateway。在9.5节中将更详细地讨论ICMP重定向的问题。

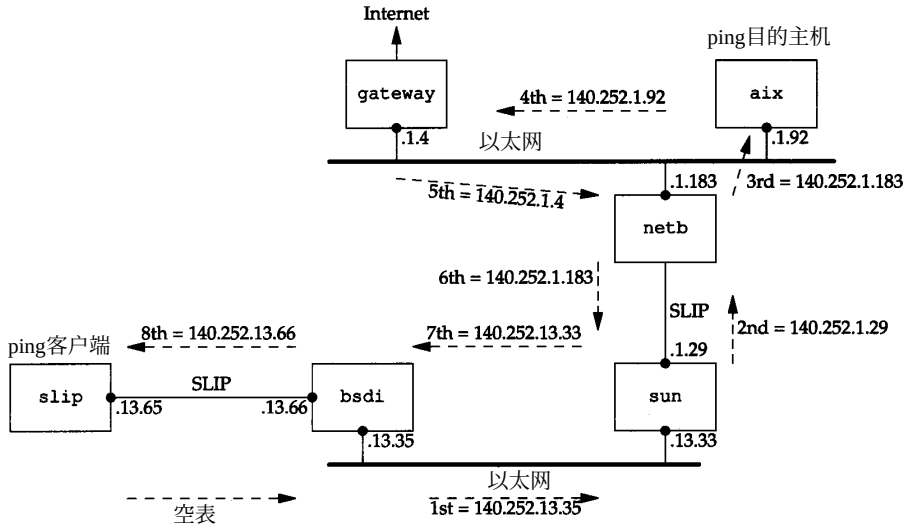


图7-6 运行带有记录路由选项的ping 程序，显示IP选路的特点

7.4 IP时间戳选项

IP时间戳选项与记录路由选项类似。IP时间戳选项的格式如图7-7所示（请与图7-3进行比较）。

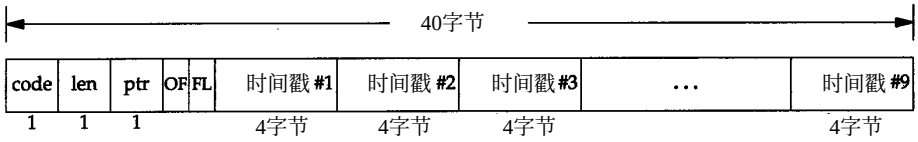


图7-7 IP首部中时间戳选项的一般格式

时间戳选项的代码为0x44。其他两个字段len和ptr与记录路由选项相同：选项的总长度（一般为36或40）和指向下一个可用空间的指针（5，9，13等）。

接下来的两个字段是4 bit的值：OF表示溢出字段，FL表示标志字段。时间戳选项的操作根据标志字段来进行，如图7-8所示。

标　志	描　述
0	只记录时间戳，正如我们在图7-7看到的那样
1	每台路由器都记录它的IP地址和时间戳。在选项列表中只有存放4对地址和时间戳的空间
3	发送端对选项列表进行初始化，存放了4个IP地址和4个取值为0的时间戳值。只有当列表中的下一个IP地址与当前路由器地址相匹配时，才记录它的时间戳

图7-8 时间戳选项不同标志字段值的意义

如果路由器由于没有空间而不能增加时间戳选项，那么它将增加溢出字段的值。

时间戳的取值一般为自UTC午夜开始计的毫秒数，与ICMP时间戳请求和应答相类似。如果路由器不使用这种格式，它就可以插入任何它使用的时间表示格式，但是必须打开时间戳中的高位以表明为非标准值。

与我们遇到的记录路由选项所受到的限制相比，时间戳选项遇到情况要更坏一些。如果我们要同时记录IP地址和时间戳（标志位为1），那么就可以同时存入其中的四对值。只记录时间戳是没有用处的，因为我们没有标明时间戳与路由器之间的对应关系（除非有一个永远不变的拓扑结构）。标志值取3会更好一些，因为我们可以插入时间戳的路由器。一个更为基本的问题是，很可能无法控制任何给定路由器上时间戳的正确性。这使得试图用 IP选项来计算路由器之间的跳站数是徒劳的。我们将看到（第8章）`traceroute`程序可以提供一种更好的方法来计算路由器之间的跳站数。

7.5 小结

`ping`程序是对两个TCP/IP系统连通性进行测试的基本工具。它只利用 ICMP回显请求和回显应答报文，而不用经过传输层（TCP/UDP）。Ping服务器一般在内核中实现ICMP的功能。

我们分析了在LAN、WAN以及SLIP链路（拨号和线路）上运行 `ping`程序的输出结果，并对串行线路上的SLIP链路吞吐量进行了计算。我们还讨论并使用了 `ping`程序的IP记录路由选项。利用该IP选项，可以看到它是如何频繁使用默认路由的。在第 9章我们将再次回到这个讨论主题。另外，还讨论了IP 时间戳选项，但它在实际使用时有所限制。

习题

7.1 请画出7.2节中`ping`输出的时间线。

7.2 若把`bsdi`和`slip`主机之间的SLIP链路设置为9600 b/s，请计算这时的RTT。假定默认的数据是56字节。

7.3 当前BSD版中的`ping`程序允许我们为ICMP报文的数据部分指定一种模式（数据部分的前8个字节不用来存放模式，因为它要存放发送报文的时间）。如果我们指定的模式为0xc0，请重新计算上一题中的答案（提示：阅读2.4节）。

7.4 使用压缩SLIP（CSLIP，见2.5节）是否会影响我们在7.2节中看到的`ping`输出中的时间值？

7.5 在图2-4中，`ping`环回地址与`ping`主机以太网地址会出现什么不同？

第8章 Traceroute程序

8.1 引言

由Van Jacobson编写的Traceroute程序是一个能更深入探索TCP/IP协议的方便可用的工具。尽管不能保证从源端发往目的端的两份连续的 IP数据报具有相同的路由，但是大多数情况下是这样的。Traceroute程序可以让我们看到IP数据报从一台主机传到另一台主机所经过的路由。Traceroute程序还可以让我们使用IP源路由选项。

使用手册上说：“程序由Steve Deering提议，由Van Jacobson实现，并由许多其他人根据C. Philip Wood, Tim Seaver 及Ken Adelman等人提出的令人信服的建议或补充意见进行调试。”

8.2 Traceroute程序的操作

在7.3节中，我们描述了IP记录路由选项（RR）。为什么不使用这个选项而另外开发一个新的应用程序？有三个方面的原因。首先，原先并不是所有的路由器都支持记录路由选项，因此该选项在某些路径上不能使用（Traceroute程序不需要中间路由器具备任何特殊的或可选的功能）。

其次，记录路由一般是单向的选项。发送端设置了该选项，那么接收端不得不从收到的 IP首部中提取出所有的信息，然后全部返回给发送端。在7.3节中，我们看到大多数Ping服务器的实现（内核中的ICMP回显应答功能）把接收到的RR清单返回，但是这样使得记录下来的IP地址翻了一番（一来一回）。这样做会受到一些限制，这一点我们在下一段讨论（Traceroute程序只需要目的端运行一个UDP模块——其他不需要任何特殊的服务器应用程序）。

最后一个原因也是最主要的原因是， IP首部中留给选项的空间有限，不能存放当前大多数的路径。在IP首部选项字段中最多只能存放9个IP地址。在原先的ARPANET中这是足够的，但是对现在来说是远远不够的。

Traceroute程序使用ICMP报文和IP首部中的TTL字段（生存周期）。TTL字段是由发送端初始设置一个8 bit字段。推荐的初始值由分配数字 RFC指定，当前值为64。较老版本的系统经常初始化为15或32。我们从第7章中的一些ping程序例子中可以看出，发送 ICMP回显应答时经常把TTL设为最大值255。

每个处理数据报的路由器都需要把 TTL的值减1或减去数据报在路由器中停留的秒数。由于大多数的路由器转发数据报的时延都小于 1秒钟，因此TTL最终成为一个跳站的计数器，所经过的每个路由器都将其值减1。

RFC 1009 [Braden and Postel 1987]指出，如果路由器转发数据报的时延超过1秒，那么它将把TTL值减去所消耗的时间（秒数）。但很少有路由器这么实现。新的路由器需求文档RFC [Almquist 1993]为此指定它为可选择功能，允许把ITL看成一个跳站计数器。

TTL字段的目的是防止数据报在选路时无休止地在网络中流动。例如，当路由器瘫痪或者两个路由器之间的连接丢失时，选路协议有时会去检测丢失的路由并一直进行下去。在这段时间内，数据报可能在循环回路被终止。 TTL字段就是在这些循环传递的数据报上加上一个生存上限。

当路由器收到一份IP数据报，如果其TTL字段是0或1，则路由器不转发该数据报（接收到这种数据报的目的主机可以将它交给应用程序，这是因为不需要转发该数据报。但是在通常情况下，系统不应该接收 TTL字段为0的数据报）。相反，路由器将该数据报丢弃，并给信源机发一份ICMP“超时”信息。Traceroute程序的关键在于包含这份ICMP信息的IP报文的信源地址是该路由器的IP地址。

我们现在可以猜想一下Traceroute程序的操作过程。它发送一份TTL字段为1的IP数据报给目的主机。处理这份数据报的第一个路由器将 TTL值减1，丢弃该数据报，并发回一份超时ICMP报文。这样就得到了该路径中的第一个路由器的地址。然后 Traceroute程序发送一份TTL值为2的数据报，这样我们就可以得到第二个路由器的地址。继续这个过程直至该数据报到达目的主机。但是目的主机哪怕接收到 TTL值为1的IP数据报，也不会丢弃该数据报并产生一份超时ICMP报文，这是因为数据报已经到达其最终目的地。那么我们该如何判断是否已经到达目的主机了呢？

Traceroute程序发送一份UDP数据报给目的主机，但它选择一个不可能的值作为 UDP端口号（大于30 000），使目的主机的任何一个应用程序都不可能使用该端口。因为，当该数据报到达时，将使目的主机的 UDP模块产生一份“端口不可达”错误（见 6.5节）的ICMP报文。这样，Traceroute程序所要做的就是区分接收到的 ICMP报文是超时还是端口不可达，以判断什么时候结束。

> Traceroute程序必须可以为发送的数据报设置TTL字段。并非所有与TCP/IP接口的程序都支持这项功能，同时并非所有的实现都支持这项能力，但目前大部分系统都支持这项功能，并可以运行Traceroute程序。这个程序界面通常要求用户具有超级用户权限，这意味着它可能需要特殊的权限以在你的主机上运行该程序。

8.3 局域网输出

现在已经做好运行Traceroute程序并观察其输出的准备了。我们将使用从 `svr4`到`slip`，经路由器bsdi的简单互联网（见内封面）。`bsdi`和`slip`之间是9600 b/s的SLIP链路。

```
svr4 % traceroute slip
traceroute to slip (140.252.13.65), 30 hops max, 40 byte packets
 1  bsdi (140.252.13.35)  20 ms  10 ms  10 ms
 2  slip (140.252.13.65)  120 ms  120 ms  120 ms
```

输出的第1个无标号行给出了目的主机名和其 IP地址，指出traceroute程序最大的TTL字段值为30。40字节的数据报包含20字节IP首部、8字节的UDP首部和12字节的用户数据（12字节的用户数据包含每发一个数据报就加 1的序列号，送出TTL的副本以及发送数据报的时间）。

输出的后面两行以TTL开始，接下来是主机或路由器名以及其IP地址。对于每个TTL值，发送3份数据报。每接收到一份ICMP报文，就计算并打印出往返时间。如果在5秒种内仍未收到3份数据报的任意一份的响应，则打印一个星号，并发送下一份数据报。在上述输出结果中，TTL字段为1的前3份数据报的ICMP报文分别在20 ms、10 ms和10 ms收到。TTL字段为2的3份数

据报的ICMP报文则在120 ms后收到。由于TTL字段为2到达最终目的主机，因此程序就此停止。

往返时间是由发送主机的traceroute程序计算的。它是指从traceroute程序到该路由器的总往返时间。如果我们对每段路径的时间感兴趣，可以用 TTL字段为N+1所打印出来的时间减去TTL字段为N的时间。

图8-1给出了tcpdump的运行输出结果。正如我们所预想的那样，第 1个发往bsdi的探测数据报的往返时间是 20 ms、而后面两个数据报往返时间是 10 ms的原因是发生了一次 ARP交换。tcpdump结果证实了确实是这种情况。

```
1  0.0                    arp who-has bsdi tell svr4
2  0.000586 (0.0006)      arp reply bsdi is-at 0:0:c0:6f:2d:40

3  0.003067 (0.0025)      svr4.42804 > slip.33435: udp 12 [ttl 1]
4  0.004325 (0.0013)      bsdi > svr4: icmp: time exceeded in-transit

5  0.069810 (0.0655)      svr4.42804 > slip.33436: udp 12 [ttl 1]
6  0.071149 (0.0013)      bsdi > svr4: icmp: time exceeded in-transit

7  0.085162 (0.0140)      svr4.42804 > slip.33437: udp 12 [ttl 1]
8  0.086375 (0.0012)      bsdi > svr4: icmp: time exceeded in-transit

9  0.118608 (0.0322)      svr4.42804 > slip.33438: udp 12
10 0.226464 (0.1079)      slip > svr4: icmp: slip udp port 33438 unreachable

11 0.287296 (0.0608)      svr4.42804 > slip.33439: udp 12
12 0.395230 (0.1079)      slip > svr4: icmp: slip udp port 33439 unreachable

13 0.409504 (0.0143)      svr4.42804 > slip.33440: udp 12
14 0.517430 (0.1079)      slip > svr4: icmp: slip udp port 33440 unreachable
```

图8-1 从svr4到slip的traceroute程序示例的tcpdump输出结果

目的主机UDP端口号最开始设置为 33435，且每发送一个数据报加 1。可以通过命令行选项来改变开始的端口号。 UDP数据报包含12个字节的用户数据，我们在前面 traceroute程序输出的40字节数据报中已经对其进行了描述。

后面tcpdump打印出了TTL字段为1的IP数据报的注释 [ttl 1]。当TTL值为0或1时，tcpdump打印出这条信息，以提示我们数据报中有些不太寻常之处。在这里可以预见到 TTL值为1；而在其他一些应用程序中，它可以警告我们数据报可能无法到达其最终目的主机。我们不可能看到路由器传送一个TTL值为0的数据报，除非发出该数据报的该路由器已经崩溃。

因为bsdi路由器将TTL值减到0，因此我们预计它将发回“传送超时”的 ICMP报文。即使这份被丢弃的IP报文发送往slip，路由器也会发回ICMP报文。

有两种不同的ICMP“超时”报文（见6.2节的图6-3），它们的ICMP报文中code字段不同。图8-2给出了这种ICMP差错报文的格式。

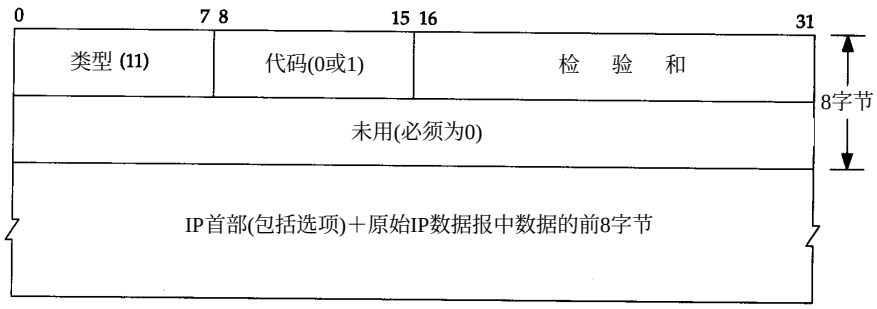


图8-2 ICMP超时报文

我们所讨论的ICMP报文是在TTL值等于0时产生的，其code字段为0。

主机在组装分片时可能发生超时，这时，它将发送一份“组装报文超时”的 ICMP报文（我们将在11.5节讨论分片和组装）。这种差错报文将code字段置1。

图8-1的第9~14行对应于TTL为2的3份数据报。这3份报文到达最终目的主机，并产生一份ICMP端口不可达报文。

计算出SLIP链路的往返时间是很有意义的，就象我们在 7.2节中所举的Ping例子，将链路值设置为1200b/s一样。发送出的UDP数据报共42个字节，包括12字节的数据、8字节UDP首部、20字节的IP首部以及（至少）2字节的SLIP帧（2.4节）。但是与Ping不一样的是，返回的数据报大小是变化的。从图 6-9可以看出，返回的ICMP报文包含发生差错的数据报的 IP首部以及紧随该IP首部的8字节数据（在traceroute程序中，即UDP首部）。这样，总共就是20 + 8 + 20 + 8 + 2，即58字节。在数据速率为960 b/s的情况下，预计的RTT就是（42 + 58/960），即104 ms。这个值与svr4上所估算出来的110 ms是吻合的。

图8-1中的源端口号（42804）看起来有些大。traceroute程序将其发送的UDP数据报的源端口号设置为 Unix进程号与 32768之间的逻辑或值。对于在同一台主机上多次运行traceroute程序的情况，每个进程都查看ICMP返回的UDP首部的源端口号，并且只处理那些对自己发送应答的报文。

关于traceroute程序，还有一些必须指出的事项。首先，并不能保证现在的路由也是将来所要采用的路由，甚至两份连续的 IP数据报都可能采用不同的路由。如果在运行程序时，路由发生改变，就会观察到这种变化，这是因为对于一个给定的 TTL，如果其路由发生变化，traceroute程序将打印出新的IP地址。

第二，不能保证ICMP报文的路由与traceroute程序发送的UDP数据报采用同一路由。这表明所打印出来的往返时间可能并不能真正体现数据报发出和返回的时间差（如果 UDP数据报从信源到路由器的时间是 1秒，而ICMP报文用另一条路由返回信源用了 3秒时间，则打印出来的往返时间是4秒）。

第三，返回的ICMP报文中的信源IP地址是UDP数据报到达的路由器接口的 IP地址。这与IP记录路由选项（7.3节）不同，记录的IP地址指的是发送接口地址。由于每个定义的路由器都有2个或更多的接口，因此，从 A主机到B主机上运行traceroute程序和从B主机到A主机上运行traceroute程序所得到的结果可能是不同的。事实上，如果我们从 slip主机到svr4上运行traceroute程序，其输出结果变成了：

```
slip % traceroute svr4
traceroute to svr4 (140.252.13.34), 30 hops max, 40 byte packets
 1  bsdi (140.252.13.66)  110 ms  110 ms  110 ms
 2  svr4 (140.252.13.34)  110 ms  120 ms  110 ms
```

这次打印出来的bsdi主机的IP地址是140.252.13.66，对应于SLIP接口；而上次的地址是140.252.13.35，是以太网接口地址。由于 traceroute程序同时也打印出与 IP地址相关的主机名，因而主机名也可能变化（在我们的例子中， bsdi上的两个接口都采用相同的名字）。

考虑图8-3的情况。它给出了两个局域网通过一个路由器相连的情况。两个路由器通过一个点对点的链路相连。如果我们在左边 LAN的一个主机上运行traceroute程序，那么它将发现路由器的IP地址为if1和if3。但在另一种情况下，就会发现打印出来的 IP地址为if4和if2。if2和if3有着同样的网络号，而另两个接口则有着不同的网络号。

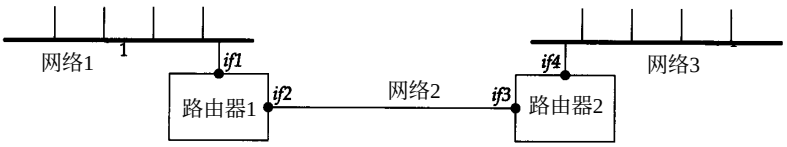


图8-3 traceroute 程序打印出的接口标识

最后，在广域网情况下，如果traceroute程序的输出是可读的域名形式，而不是IP地址形式，那么会更好理解一些。但是由于traceroute程序接收到ICMP报文时，它所获得的唯一信息就是IP地址，因此，在给定IP地址的情况下，它做一个“反向域名查看”工作来获得域名。这就需要路由器或主机的管理员正确配置其反向域名查看功能（并非所有的情况下都是如此）。我们将在14.5节描述如何使用DNS将一个IP地址转换成域名。

8.4 广域网输出

前面所给出的小互联网的输出例子对于查看协议运行过程来说是足够了，但对于像全球互联网这样的大互联网来说，应用traceroute程序就需要一些更为实际的东西。

图8-4是从sun主机到NIC (Network Information Center)的情况。

```
sun % traceroute nic.ddn.mil
traceroute to nic.ddn.mil (192.112.36.5), 30 hops max, 40 byte packets
 1  netb.tuc.noao.edu (140.252.1.183)  218 ms  227 ms  233 ms
 2  gateway.tuc.noao.edu (140.252.1.4)  233 ms  229 ms  204 ms
 3  butch.telcom.arizona.edu (140.252.104.2)  204 ms  228 ms  234 ms
 4  Gabby.Telcom.Arizona.EDU (128.196.128.1)  234 ms  228 ms  204 ms
 5  NSIgate.Telcom.Arizona.EDU (192.80.43.3)  233 ms  228 ms  234 ms
 6  JPL1.NSN.NASA.GOV (128.161.88.2)  234 ms  590 ms  262 ms
 7  JPL3.NSN.NASA.GOV (192.100.15.3)  238 ms  223 ms  234 ms
 8  GSFC3.NSN.NASA.GOV (128.161.3.33)  293 ms  318 ms  324 ms
 9  GSFC8.NSN.NASA.GOV (192.100.13.8)  294 ms  318 ms  294 ms
10  SURA2.NSN.NASA.GOV (128.161.166.2)  323 ms  319 ms  294 ms
11  nsn-FIX-pe.sura.net (192.80.214.253)  294 ms  318 ms  294 ms
12  GSI.NSN.NASA.GOV (128.161.252.2)  293 ms  318 ms  324 ms
13  NIC.DDN.MIL (192.112.36.5)  324 ms  321 ms  324 ms
```

图8-4 从sun主机到nic.ddn.mil 的traceroute 程序

由于运行的这个例子包含文本，非DDN站点（如，非军方站点）的NIC已经从nic.ddn.mil转移到rs.internic.net，即新的“InterNIC”。

一旦数据报离开tuc.noao.edu网，它们就进入了telcom.arizona.edu网络。然后这些数据报进入NASA Science Internet，nsn.nasa.gov。TTL字段为6和7的路由器位于JPL (Jet Propulsion Laboratory)上。TTL字段为11所输出的sura.net网络位于Southeastern Universities Research Association Network上。TTL字段为12的域名GSI是Government Systems, Inc., NIC的运营者。

TTL字段为6的第2个RTT（590）几乎是其他两个RTT值（234和262）的两倍。它表明IP路由的动态变化。在发送主机和这个路由器之间发生了使该数据报速度变慢的事件。同样，我们不能区分是发出的数据报还是返回的ICMP差错报文被拦截。

TTL字段为3的第1个RTT探测值（204）比TTL字段为2的第1个探测值（233）值还小。由

于每个打印出来的RTT值是从发送主机到路由器的总时间，因此这种情况是可能发生的。

图8-5的例子是从sun主机到作者出版商之间的运行例子。

```
sun % traceroute aw.com
traceroute to aw.com (192.207.117.2), 30 hops max, 40 byte packets

 1  netb.tuc.noao.edu (140.252.1.183)  227 ms  227 ms  234 ms
 2  gateway.tuc.noao.edu (140.252.1.4)  233 ms  229 ms  234 ms

 3  butch.telcom.arizona.edu (140.252.104.2)  233 ms  229 ms  234 ms
 4  Gabby.Telcom.Arizona.EDU (128.196.128.1)  264 ms  228 ms  234 ms
 5  Westgate.Telcom.Arizona.EDU (192.80.43.2)  234 ms  228 ms  234 ms

 6  uu-ua.AZ.westnet.net (192.31.39.233)  263 ms  258 ms  264 ms
 7  enss142.UT.westnet.net (192.31.39.21)  263 ms  258 ms  264 ms

 8  t3-2.Denver-cnss97.t3.ans.net (140.222.97.3)  293 ms  288 ms  275 ms
 9  t3-3.Denver-cnss96.t3.ans.net (140.222.96.4)  283 ms  263 ms  261 ms
10  t3-1.St-Louis-cnss80.t3.ans.net (140.222.80.2)  282 ms  288 ms  294 ms
11  t3-1.Chicago-cnss24.t3.ans.net (140.222.24.2)  293 ms  288 ms  294 ms
12  t3-2.Cleveland-cnss40.t3.ans.net (140.222.40.3)  294 ms  288 ms  294 ms
13  t3-1.New-York-cnss32.t3.ans.net (140.222.32.2)  323 ms  318 ms  324 ms
14  t3-1.Washington-DC-cnss56.t3.ans.net (140.222.56.2)  323 ms  318 ms  324 ms
15  t3-0.Washington-DC-cnss58.t3.ans.net (140.222.58.1)  324 ms  318 ms  324 ms
16  t3-0.enss136.t3.ans.net (140.222.136.1)  323 ms  318 ms  324 ms

17  Washington.DC.ALTER.NET (192.41.177.248)  323 ms  377 ms  324 ms
18  Boston.MA.ALTER.NET (137.39.12.2)  324 ms  347 ms  324 ms
19  AW-gw.ALTER.NET (137.39.62.2)  353 ms  378 ms  354 ms

20  aw.com (192.207.117.2)  354 ms  349 ms  354 ms
```

图8-5 从sun.tuc.noao.edu 主机到aw.com 的traceroute 程序

在这个例子中，数据报离开 telcom.arizona.edu网络后就进行了地区性的网络westnet.net (TTL字段值为6和7)。然后进行了由 Advanced Network & Services运营的NSFNET主干网，t3.ans.net，（T3是对于主干网采用的45 Mb/s电话线的一般缩写。）最后的网络是alter.net，即aw.com与互联网的连接点。

8.5 IP源站选路选项

通常IP路由是动态的，即每个路由器都要判断数据报下面该转发到哪个路由器。应用程序对此不进行控制，而且通常也并不关心路由。它采用类似 Traceroute程序的工具来发现实际的路由。

源站选路(source routing)的思想是由发送者指定路由。它可以采用以下两种形式：

- 严格的源路由选择。发送端指明IP数据报所必须采用的确切路由。如果一个路由器发现源路由所指定的下一个路由器不在其直接连接的网络上，那么它就返回一个“源站路由失败”的ICMP差错报文。
- 宽松的源站选路。发送端指明了一个数据报经过的 IP地址清单，但是数据报在清单上指明的任意两个地址之间可以通过其他路由器。

Traceroute程序提供了一个查看源站选路的方法，我们可以在选项中指明源站路由，然后检查其运行情况。

一些公开的Traceroute程序源代码包中包含指明宽松的源站选路的补丁。但是在标准版中通常并不包含此项。这些补丁的解释是“ Van Jacobson的原始Traceroute程序

（1988年春）支持该特性，但后来因为有人提出会使网关崩溃而将此功能去除。”对于本章中所给出的例子，作者将这些补丁安装上去，并将它们设置成允许宽松的源站选路和严格的源站选路。

图8-6给出了源站路由选项的格式。

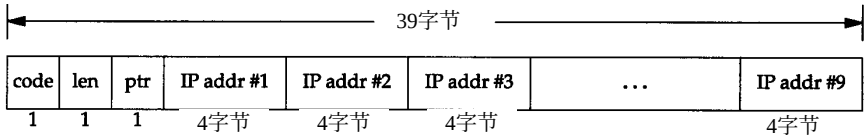


图8-6 IP首部源站路由选项的通用格式

这个格式与我们在图7-3中所示的记录路由选项格式基本一致。不同之处是，对于源站选路，我们必须在发送IP数据报前填充IP地址清单；而对于记录路由选项，我们需要为IP地址清单分配并清空一些空间，并让路由器填充该清单中的各项。同时，对于源站选路，只要为所需要的IP地址数分配空间并进行初始化，通常其数量小于9。而对于记录路由选项来说，必须尽可能地分配空间，以达到9个地址。

对于宽松的源站选路来说，code字段的值是0x83；而对于严格的源站选路，其值为0x89。len和ptr字段与7.3节中所描述的一样。

源站路由选项的实际称呼为“源站及记录路由”（对于宽松的源站选路和严格的源站选路，分别用LSRR和SSRR表示），这是因为在数据报沿路由发送过程中，对IP地址清单进行了更新。下面是其运行过程：

- 发送主机从应用程序接收源站路由清单，将第1个表项去掉（它是数据报的最终目的地址），将剩余的项移到1个项中（如图8-6所示），并将原来的目的地址作为清单的最后一项。指针仍然指向清单的第1项（即，指针的值为4）。
- 每个处理数据报的路由器检查其是否为数据报的最终地址。如果不是，则正常转发数据报（在这种情况下，必须指明宽松源站选路，否则就不能接收到该数据报）。
- 如果该路由器是最终目的，且指针不大于路径的长度，那么(1)由ptr所指定的清单中的下一个地址就是数据报的最终目的地址；(2)由外出接口(outgoing interface)相对应的IP地址取代刚才使用的源地址；(3)指针加4。

可以用下面这个例子很好地解释上述过程。在图8-7中，我们假设主机S上的发送应用程序发送一份数据报给*D*，指定源路由为*R*1，*R*2和*R*3。

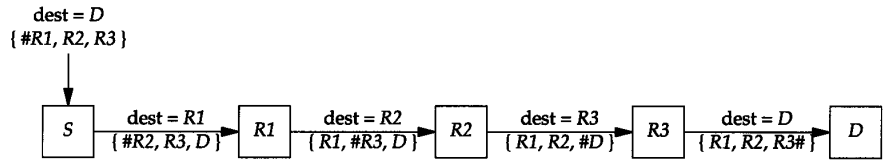


图8-7 IP源路由示例

在上图中，#表示指针字段，其值分别是4、8、12和16。长度字段恒为15（三个IP地址加上三个字节首部）。可以看出，每一跳IP数据报中的目的地址都发生改变。

当一个应用程序接收到由信源指定路由的数据时，在发送应答时，应该读出接收到的路由值，并提供反向路由。

Host Requirements RFC指明，TCP客户必须能指明源站选路，同时，TCP服务器必须能够接收源站选路，并且对于该TCP连接的所有报文段都能采用反向路由。如果TCP服务器下面接收到一个不同的源站选路，那么新的源站路由将取代旧的源站路由。

8.5.1 宽松的源站选路的traceroute程序示例

使用traceroute程序的-g选项，可以为宽松的源站选路指明一些中间路由器。采用该选项可以最多指定8个中间路由器（其个数是8而不是9的原因是，所使用的编程接口要求最后的表目是目的主机）。

在图8-4中，去往NIC，即nic.ddn.mil的路由经过NASA Science Internet。在图8-8中，我们通过指定路由器enss142.UT.westnet.net (192.31.39.21) 作为中间路由器来强制数据报通过NSFNET：

```
sun % traceroute -g 192.31.39.21  nic.ddn.mil
traceroute to nic.ddn.mil (192.112.36.5), 30 hops max, 40 byte packets
 1  netb.tuc.noao.edu (140.252.1.183)  259 ms  256 ms  235 ms
 2  butch.telcom.arizona.edu (140.252.104.2)  234 ms  228 ms  234 ms
 3  Gabby.Telcom.Arizona.EDU (128.196.128.1)  234 ms  257 ms  233 ms
 4  enss142.UT.westnet.net (192.31.39.21)  294 ms  288 ms  295 ms
 5  t3-2.Denver-cnss97.t3.ans.net (140.222.97.3)  294 ms  286 ms  293 ms
 6  t3-3.Denver-cnss96.t3.ans.net (140.222.96.4)  293 ms  288 ms  294 ms
 7  t3-1.St-Louis-cnss80.t3.ans.net (140.222.80.2)  294 ms  318 ms  294 ms
 8  * t3-1.Chicago-cnss24.t3.ans.net (140.222.24.2)  318 ms  295 ms
 9  t3-2.Cleveland-cnss40.t3.ans.net (140.222.40.3)  319 ms  318 ms  324 ms
10  t3-1.New-York-cnss32.t3.ans.net (140.222.32.2)  324 ms  318 ms  324 ms
11  t3-1.Washington-DC-cnss56.t3.ans.net (140.222.56.2)  353 ms  348 ms  325 ms
12  t3-0.Washington-DC-cnss58.t3.ans.net (140.222.58.1)  348 ms  347 ms  325 ms
13  t3-0.enss145.t3.ans.net (140.222.145.1)  353 ms  348 ms  325 ms
14  nsn-FIX-pe.sura.net (192.80.214.253)  353 ms  348 ms  325 ms
15  GSI.NSN.NASA.GOV (128.161.252.2)  353 ms  348 ms  354 ms
16  NIC.DDN.MIL (192.112.36.5)  354 ms  347 ms  354 ms
```

图8-8 采用宽松源站选路通过NSFNET到达nic.ddn.mil 的traceroute 程序

在这种情况下，看起来路径中共有16跳，其平均RTT大约是350 ms。而图8-4的通常选路则只有13跳，其平均RTT约为322 ms。默认路径看起来更好一些（在建立路径时，还需要考虑其他的一些因素。其中一些必须考虑的因素是所包含网络的组织及政治因素）。

前面我们说看起来有16跳，这是因为将其输出结果与前面的通过NSFNET（图8-5）的示例比较，发现在本例采用宽松源路由，选择了3个路由器（这可能是因为路由器对源站选路数据报产生ICMP超时差错报文上存在一些差错）。在netb和butch路由器之间的gateway.tuc.noao.edu路由器丢失了，同时，位于Gabby和enss142.UT.west.net之间的Westgate.Telcom.Arizona.edu和uu-ua.AZ.westnet.net两个路由器也丢失了。在这些丢失的路由器上可能发生了与接收到宽松的源站选路选项数据报有关的程序问题。实际上，当采用NSFNET时，信源和NIC之间的路径有19跳。本章习题8.5继续对这些丢失路由器进行讨论。

同时本例也指出了另一个问题。在命令行，我们必须指定路由器enss142.UT.westnet.net的点分十进制IP地址，而不能以其域名代替。这是因为，反向域名解析（14.5节中描述的通过IP

地址返回域名）将域名与IP地址相关联，但是前向解析（即给出域名返回IP地址）则无法做到。在DNS中，前向映射和反向映射是两个独立的文件，而并非所有的管理者都同时拥有这两个文件。因此，在一个方向是工作正常而另一个方向却失败的情况并不少见。

还有一种以前没有碰到过的情况是在TTL字段为8的情况下，对于第一个RTT，打印一个星号。这表明，发生超时，在5秒内未收到本次探查的应答信号。

将本图与图8-4相比较，还可以得出一个结论，即路由器nsn-FIX-pe.sura.net同时与NSFNET和NASA Science Internet相连。

8.5.2 严格的源站选路的traceroute程序示例

在作者的traceroute程序版本中，-G选项与前面所描述的-g选项是完全一样的，不过此时是严格的源站选路而不是宽松的源站选路。我们可以采用这个选项来观察在指明无效的严格的源站选路时其结果会是什么样的。从图8-5可以看出来，从作者的子网发往NSFNET的数据报的正常路由器顺序是netb，gateway，butch和gabby（为了便于查看，后面所有的输出结果中，均省略了域名后缀.tuc.noao.edu和.telcom.arizona.edu）。我们指定了一个严格源路由，使其试图将数据报从gateway直接发送到gabby，而省略了butch。我们可以猜测到其结果会是失败的，正如图8-9所给出的结果。

```
sun % traceroute -G netb -G gateway -G gabby westgate
traceroute to westgate (192.80.43.2), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  272 ms  257 ms  261 ms
 2  gateway (140.252.1.4)  263 ms  259 ms  234 ms
 3  gateway (140.252.1.4)  263 ms !S *  235 ms !S
```

图8-9 采用严格源站路由失败的traceroute程序

这里的关键是在于TTL字段为3的输出行中，RTT后面的!S。这表明traceroute程序接收到ICMP“源站路由失败”的差错报文：即图6-3中type字段为3，而code字段为5。TTL字段为3的第二个RTT位置的星号表示未收到这次探查的应答信号。这与我们所猜想的一样，gateway不可能直接发送数据报给gabby，这是因为它们之间没有直接的连接。

TTL字段为2和3的结果都来自于gateway，对于TTL字段为2的应答来自gateway，是因为gateway接收到TTL字段为1的数据报。在它查看到（无效的）严格的源站选路之前，就发现TTL已过期，因此发送回ICMP超时报文。TTL字段等于3的行，在进入gateway时其TTL字段为2，因此，它查看严格的源站选路，发现它是无效的，因此发送回ICMP源站选路失败的差错报文。

图8-10给出了与本例相对应的tcpdump输出结果。该输出结果是在sun和netb之间的SLIP链路上遇到的。我们必须在tcpdump中指定-v选项以显示出源站路由信息。这样，会输出一些像数据报ID这样我们并不需要的结果，我们在给出结果中将这些不需要的结果删除掉。同样，用SSRR表示“严格的源站及记录路由”。

首先注意到，sun所发送的每个UDP数据报的目的地址都是netb，而不是目的主机(westgate)。这一点可以用图8-7的例子来解释。类似地，-G选项所指定的另外两个路由器（gateway和gabby）以及最终目(westgate)成为第一跳的SSRR选项。

从这个输出结果中，还可以看出，traceroute程序所采用的定时时间（第15行和16行

之间的时间差）是5秒。

```
1   0.0                  sun.33593 > netb.33435: udp 12 [ttl 1]
                         (optlen=16 SSRR{#gateway gabby westgate} EOL)
2   0.270278 (0.2703)    netb > sun: icmp: time exceeded in-transit
3   0.284784 (0.0145)    sun.33593 > netb.33436: udp 12 [ttl 1]
                         (optlen=16 SSRR{#gateway gabby westgate} EOL)
4   0.540338 (0.2556)    netb > sun: icmp: time exceeded in-transit
5   0.550062 (0.0097)    sun.33593 > netb.33437: udp 12 [ttl 1]
                         (optlen=16 SSRR{#gateway gabby westgate} EOL)
6   0.810310 (0.2602)    netb > sun: icmp: time exceeded in-transit
7   0.818030 (0.0077)    sun.33593 > netb.33438: udp 12 (ttl 2,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
8   1.080337 (0.2623)    gateway > sun: icmp: time exceeded in-transit
9   1.092564 (0.0122)    sun.33593 > netb.33439: udp 12 (ttl 2,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
10  1.350322 (0.2578)    gateway > sun: icmp: time exceeded in-transit
11  1.357382 (0.0071)    sun.33593 > netb.33440: udp 12 (ttl 2,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
12  1.590586 (0.2332)    gateway > sun: icmp: time exceeded in-transit
13  1.598926 (0.0083)    sun.33593 > netb.33441: udp 12 (ttl 3,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
14  1.860341 (0.2614)    gateway > sun:
                         icmp: gateway unreachable - source route failed
15  1.875230 (0.0149)    sun.33593 > netb.33442: udp 12 (ttl 3,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
16  6.876579 (5.0013)    sun.33593 > netb.33443: udp 12 (ttl 3,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
17  7.110518 (0.2339)    gateway > sun:
                         icmp: gateway unreachable - source route failed
```

图8-10 失败的严格源站选路traceroute 程序的tcpdump 输出结果

8.5.3 宽松的源站选路traceroute程序的往返路由

我们在前面已经说过，从A到B的路径并不一定与从B到A的路径完全一样。除非同时在两个系统中登录并在每个终端上运行 traceroute程序，否则很难发现两条路径是否不同。但是，采用宽松的源站选路，就可以决定两个方向上的路径。

这里的窍门就在于指定一个宽松的源站路由，该路由的目的端和宽松路径一样，但发送端为目的主机。例如，在sun主机上，我们可以查看到发往以及来自bruno.cs.colorado.edu的结果如图8-11所示。

发出路径（TTL字段为1~11）的结果与返回路径（TTL字段为11~21）不同，这很好地说明了在Internet 上，选路可能是不对称的。

该输出同时还说明了我们在图 8-3中所讨论的问题。比较TTL字段为2和19的输出结果：它们都是路由器gateway.tuc.noao.edu，但两个IP地址却是不同的。由于traceroute程序以进入接口作为其标识，而我们从两条不同的方向经过该路由器，一条是发出路径（ TTL字段为2），另一条是返回路径（TTL字段为19），因此可以猜想到这个结果。通过比较 TTL字段为3和18、4和17的结果，可以看到同样的结果。

```
sun % traceroute -g bruno.cs.colorado.edu sun
traceroute to sun (140.252.13.33), 30 hops max, 40 byte packets
 1  netb.tuc.noao.edu (140.252.1.183)  230 ms  227 ms  233 ms
 2  gateway.tuc.noao.edu (140.252.1.4)  233 ms  229 ms  234 ms
 3  butch.telcom.arizona.edu (140.252.104.2)  234 ms  229 ms  234 ms
 4  Gabby.Telcom.Arizona.EDU (128.196.128.1)  233 ms  231 ms  234 ms
 5  NSIgate.Telcom.Arizona.EDU (192.80.43.3)  294 ms  258 ms  234 ms
 6  JPL1.NSN.NASA.GOV (128.161.88.2)  264 ms  258 ms  264 ms
 7  JPL2.NSN.NASA.GOV (192.100.15.2)  264 ms  258 ms  264 ms
 8  NCAR.NSN.NASA.GOV (128.161.97.2)  324 ms *  295 ms
 9  cu-gw.ucar.edu (192.43.244.4)  294 ms  318 ms  294 ms
10  engr-gw.Colorado.EDU (128.138.1.3)  294 ms  288 ms  294 ms
11  bruno.cs.colorado.edu (128.138.243.151)  293 ms  317 ms  294 ms
12  engr-gw-ot.cs.colorado.edu (128.138.204.1)  323 ms  317 ms  384 ms
13  cu-gw.Colorado.EDU (128.138.1.1)  294 ms  318 ms  294 ms
14  enss.ucar.edu (192.43.244.10)  323 ms  318 ms  294 ms
15  t3-1.Denver-cnss97.t3.ans.net (140.222.97.2)  294 ms  288 ms  384 ms
16  t3-0.enss142.t3.ans.net (140.222.142.1)  293 ms  288 ms  294 ms
17  Gabby.Telcom.Arizona.EDU (192.80.43.1)  294 ms  288 ms  294 ms
18  Butch.Telcom.Arizona.EDU (128.196.128.88)  293 ms  317 ms  294 ms
19  gateway.tuc.noao.edu (140.252.104.1)  294 ms  289 ms  294 ms
20  netb.tuc.noao.edu (140.252.1.183)  324 ms  321 ms  294 ms
21  sun.tuc.noao.edu (140.252.13.33)  534 ms  529 ms  564 ms
```

图8-11 显示非对称路径的traceroute 程序

8.6 小结

在一个TCP/IP网络中，traceroute程序是不可缺少的工具。其操作很简单：开始时发送一个TTL字段为1的UDP数据报，然后将TTL字段每次加1，以确定路径中的每个路由器。每个路由器在丢弃UDP数据报时都返回一个ICMP超时报文2，而最终目的主机则产生一个ICMP端口不可达的报文。

我们给出了在LAN和WAN上运行traceroute程序的例子，并用它来考察IP源站选路。我们用宽松的源站选路来检测发往目的主机的路由是否与从目的主机返回的路由一样。

习题

8.1 当IP将接收到的TTL字段减1，发现它为0时，将会发生什么结果？

8.2 traceroute程序是如何计算RTT的？将这种计算RTT的方法与ping相比较。

8.3 （本习题与下一道习题是基于开发traceroute程序过程中遇到的实际问题，它们来自于traceroute程序源代码注释）。假设源主机和目的主机之间有三个路由器（R1、R2和R3），而中间的路由器（R2）在进入TTL字段为1时，将TTL字段减1，但却错误地将该IP数据报发往下一个路由器。请描述会发生什么结果。在运行traceroute程序时会看到什么样的现象？

8.4 同样，假设源主机和目的主机之间有三个路由器。由于目的主机上存在错误，因此，它总是将进入TTL值作为外出ICMP报文的TTL值。请描述这将发生什么结果，你会看到什么现象。

8.5 在图8-8运行例子中，我们可以在sun和netb之间的SLIP链路上运行tcpdump程序。如果指定-v选项，就可以看到返回ICMP报文的TTL值。这样，我们可以看到进入netb、butch、Gabby和enss142.UT.westnet.net的TTL值分别为255、253、252和249。这是否为我们判断是否存在丢失路由器提供了额外的信息？

8.6 SunOS和SVR4都提供了带-l选项的ping版本，以提供松源选路。手册上说明，该选项可以与-R选项（指定记录路由选项）一起使用。如果已经进入到这些系统中，请尝试同时用这两个选项。其结果是什么？如果采用tcpdump来观测数据报，请描述其过程。

8.7 比较ping和traceroute程序在处理同一台主机上客户的多个实例的不同点。

8.8 比较ping和traceroute程序在计算往返时间上的不同点。

8.9 我们已经说过，traceroute程序选取开始UDP目的主机端口号为33453，每发送一个数据报将此数加1。在1.9节中，我们说过暂时端口号通常是1024~5000之间的值，因此traceroute程序的目的主机端口号不可能是目的主机上所使用的端口号。在Solaris2.2系统中的情况也是如此吗？（提示：查看E.4节）

8.10 RFC 1393 [Malkin 1993b]提出了另一种判断到目的主机路径的方法。请问其优缺点是什么？

第9章 IP选路

9.1 引言

选路是IP最重要的功能之一。图9-1是IP层处理过程的简单流程。需要进行选路的数据报可以由本地主机产生，也可以由其他主机产生。在后一种情况下，主机必须配置成一个路由器，否则通过网络接口接收到的数据报，如果目的地址不是本机就要被丢弃（例如，悄无声息地被丢弃）。

在图9-1中，我们还描述了一个路由守护程序（ daemon），通常这是一个用户进程。在Unix系统中，大多数普通的守护程序都是路由程序和网关程序（术语 daemon指的是运行在后台的进程，它代表整个系统执行某些操作。 daemon一般在系统引导时启动，在系统运行期间一直存在）。在某个给定主机上运行何种路由协议，如何在相邻路由器上交换选路信息，以及选路协议是如何工作的，所有这些问题都是非常复杂的，其本身就可以用整本书来加以讨论（有兴趣的读者可以参考文献[Perlman 1992]以获得更详细的信息）。在第10章中，我们将简单讨论动态选路和选路信息协议RIP（Routing Information Protocol）。在本章中，我们主要的目的是了解单个IP层如何作出路由决策。

图9-1所示的路由表经常被IP访问（在一个繁忙的主机上，一秒钟内可能要访问几百次），但是它被路由守护程序更新的频度却要低得多（可能大约 30秒种一次）。当接收到ICMP重定向，报文时，路由表也要被更新，这一点我们将在9.5节讨论route命令时加以介绍。在本章中，我们还将用`netstat`命令来显示路由表。

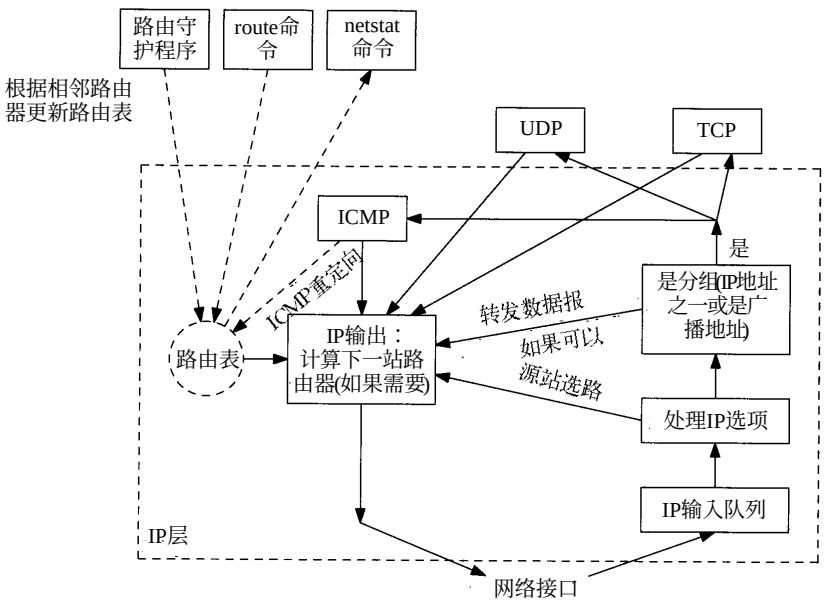


图9-1 IP层工作流程

9.2 选路的原理

开始讨论IP选路之前，首先要理解内核是如何维护路由表的。路由表中包含的信息决定了IP层所做的所有决策。

在3.3节中，我们列出了IP搜索路由表的几个步骤：

1) 搜索匹配的主机地址；

2) 搜索匹配的网络地址；

3) 搜索默认表项（默认表项一般在路由表中被指定为一个网络表项，其网络号为 0）。

匹配主机地址步骤始终发生在匹配网络地址步骤之前。

IP层进行的选路实际上是一种选路机制，它搜索路由表并决定向哪个网络接口发送分组。这区别于选路策略，它只是一组决定把哪些路由放入路由表的规则。 IP执行选路机制，而路由守护程序则一般提供选路策略。

9.2.1 简单路由表

首先来看一看一些典型的主机路由表。在主机svr4上，我们先执行带-r选项的netstat命令列出路由表，然后以-n选项再次执行该命令，以数字格式打印出IP地址（我们这样做是因为路由表中的一些表项是网络地址，而不是主机地址。如果没有- n选项，netstat命令将搜索文件/etc/networks并列出其中的网络名。这样会与另一种形式的名字——网络名加主机名相混淆）。

```
svr4 % netstat -rn
Routing tables
Destination        Gateway            Flags    Refcnt  Use      Interface
140.252.13.65      140.252.13.35      UGH      0       0        emd0
127.0.0.1          127.0.0.1          UH       1       0        lo0
default            140.252.13.33      UG       0       0        emd0
140.252.13.32      140.252.13.34      U        4       25043    emd0
```

第1行说明，如果目的地是140.252.13.65（slip主机），那么网关（路由器）将把分组转发给140.252.13.35（bsdi）。这正是我们所期望的，因为主机 slip通过SLIP链路与bsdi相连接，而bsdi与该主机在同一个以太网上。

对于一个给定的路由器，可以打印出五种不同的标志（ flag）：

U 该路由可以使用。

G 该路由是到一个网关（路由器）。如果没有设置该标志，说明目的地是直接相连的。

H 该路由是到一个主机，也就是说，目的地址是一个完整的主机地址。如果没有设置该标志，说明该路由是到一个网络，而目的地址是一个网络地址：一个网络号，或者网络号与子网号的组合。

D 该路由是由重定向报文创建的（9.5节）。

M 该路由已被重定向报文修改（9.5节）。

标志G是非常重要的，因为由它区分了间接路由和直接路由（对于直接路由来说是不设置标志G的）。其区别在于，发往直接路由的分组中不但具有指明目的端的 IP地址，还具有其链路层地址（见图 3-3）。当分组被发往一个间接路由时， IP地址指明的是最终的目的地，但是链路层地址指明的是网关（即下一站路由器）。我们在图3-4已看到这样的例子。在这个路由表例子中，有一个间接路由（设置了标志 G），因此采用这一项路由的分组其 IP地址是最终的目的地（140.252.13.65），但是其链路层地址必须对应于路由器 140.252.13.35。

理解G和H标志之间的区别是很重要的。 G标志区分了直接路由和间接路由，如上所述。但是H标志表明，目的地址（netstat命令输出第一行）是一个完整的主机地址。没有设置H标志说明目的地址是一个网络地址（主机号部分为 0）。当为某个目的IP地址搜索路由表时，主机地址项必须与目的地址完全匹配，而网络地址项只需要匹配目的地址的网络号和子网号就可以了。另外，大多数版本的 netstat命令首先打印出所有的主机路由表项，然后才是网络路由表项。

参考记数Refcnt（Reference count）列给出的是正在使用路由的活动进程个数。面向连接的协议如TCP在建立连接时要固定路由。如果在主机 svr4和slip之间建立Telnet连接，可以看到参考记数值变为1。建立另一个Telnet连接时，它的值将增加为2，依此类推。

下一列（“ use ”）显示的是通过该路由发送的分组数。如果我们是这个路由的唯一用户，那么运行ping程序发送5个分组后，它的值将变为 5。最后一列（interface）是本地接口的名字。

输出的第2行是环回接口（2.7节），它的名字始终为lo0。没有设置G标志，因为该路由不是一个网关。H标志说明目的地址（127.0.0.1）是一个主机地址，而不是一个网络地址。由于没有设置G标志，说明这是一个直接路由，网关列给出的是外出 IP地址。

输出的第3行是默认路由。每个主机都有一个或多个默认路由。这一项表明，如果在表中没有找到特定的路由，就把分组发送到路由器 140.252.13.33（sun主机）。这说明当前主机（svr4）利用这一个路由表项就可以通过 Internet经路由器sun（及其SLIP链路）访问其他的系统。建立默认路由是一个功能很强的概念。该路由标志（ UG）表明它是一个网关，这是我们所期望的。

这里，我们有意称sun为路由器而不是主机，因为它被当作默认路由器来使用，它发挥的是IP转发功能，而不是主机功能。

Host Requirements RFC文档特别说明，IP层必须支持多个默认路由。但是，许多实现系统并不支持这一点。当存在多个默认路由时，一般的技术就成为它们周围的知更鸟了，例如，Solaris 2.2就是这样做的。

输出中的最后一行是所在的以太网。 H标志没有设置，说明目的地址（ 140.252.13.32）是一个网络地址，其主机地址部分设为 0。事实上，是它的低 5位设为0（见图3-11）。由于这是一个直接路由（G标志没有被设置），网关列指出的IP地址是外出地址。

netstat命令输出的最后一项还隐含了另一个信息，那就是目的地址（ 140.252.13.32）的子网掩码。如果要把该目的地址与 140.252.13.33进行比较，那么在比较之前首先要把它与目的地址掩码（0xfffffe0，3.7节）进行逻辑与。由于内核知道每个路由表项对应的接口，而且每个接口都有一个对应的子网掩码，因此每个路由表项都有一个隐含的子网掩码。

主机路由表的复杂性取决于主机所在网络的拓扑结构。

1) 最简单的（也是最不令人感兴趣的）情况是主机根本没有与任何网络相连。 TCP/IP协议仍然能用于这样的主机，但是只能与自己本身通信！这种情况下的路由表只包含环回接口一项。

2) 接下来的情况是主机连在一个局域网上，只能访问局域网上的主机。这时路由表包含两项：一项是环回接口，另一项是局域网（如以太网）。

3) 如果主机能够通过单个路由器访问其他网络（如 Internet）时，那么就要进行下一步。

一般情况下增加一个默认表项指向该路由器。

4) 如果要新增其他的特定主机或网络路由，那么就要进行最后一步。在我们的例子中，到主机slip的路由要通过路由器bsdi就是这样的例子。

我们根据上述IP操作的步骤使用这个路由表为主机svr4上的一些分组例子选择路由。

1) 假定目的地址是主机sun，140.252.13.33。首先进行主机地址的匹配。路由表中的两个主机地址表项（slip和localhost）均不匹配，接着进行网络地址匹配。这一次匹配成功，找到表项140.252.13.32（网络号和子网号都相同），因此使用emd0接口。这是一个直接路由，因此链路层地址将是目的端的地址。

2) 假定目的地址是主机slip，140.252.13.65。首先在路由表搜索主机地址，并找到一个匹配地址。这是一个间接路由，因此目的端的 IP地址仍然是140.252.13.65，但是链路层地址必须是网关140.252.13.65的链路层地址，其接口名为emd0。

3) 这一次我们通过Internet给主机aw.com（192.207.117.2）发送一份数据报。首先在路由表中搜索主机地址，失败后进行网络地址匹配。最后成功地找到默认表项。该路由是一个间接路由，通过网关140.252.13.33，并使用接口名为emd0。

4) 在我们最后一个例子中，我们给本机发送一份数据报。有四种方法可以完成这件事，如用主机名、主机IP地址、环回名或者环回IP地址：

```
ftp svr4
ftp 140.252.13.34
ftp localhost
ftp 127.0.0.1
```

在前两种情况下，对路由表的第 2次搜索得到一个匹配的网络地址 140.252.13.32，并把IP报文传送给以太网驱动程序。正如图 2-4所示的那样， IP报文中的目的地址为本机IP地址，因此报文被送给环回驱动程序，然后由驱动程序把报文放入 IP输出队列中。

在后两种情况下，由于指定了环回接口的名字或 IP地址，第一次搜索就找到匹配的主机地址，因此报文直接被送给环回驱动程序，然后由驱动程序把报文放入 IP输出队列中。

上述四种情况报文都要被送给环回驱动程序，但是采用的两种路由决策是不相同的。

9.2.2 初始化路由表

我们从来没有说过这些路由表是如何被创建的。每当初始化一个接口时（通常是用ifconfig命令设置接口地址），就为接口自动创建一个直接路由。对于点对点链路和环回接口来说，路由是到达主机（例如，设置 H标志）。对于广播接口来说，如以太网，路由是到达网络。

到达主机或网络的路由如果不是直接相连的，那么就必须加入路由表。一个常用的方法是在系统引导时显式地在初始化文件中运行 route命令。在主机svr4上，我们运行下面两个命令来添加路由表中的表项：

```
route add default sun 1
route add slip bsdi 1
```

第3个参数（default和slip）代表目的端，第4个参数代表网关（路由器），最后一个参数代表路由的度量(metric)。route命令在度量值大于0时要为该路由设置G标志，否则，当耗费值为0时就不设置G标志。

不幸的是，几乎没有系统愿意在启动文件中包含route命令。在4.4BSD和BSD/386系统中，启动文件是 /etc/netstart；在SVR4系统中，启动文件是/etc/inet/rc.inet；在Solaris 2.x中，启动文件是/etc/rc2.d/S69inet；在SunOS 4.1.x中，启动文件是/etc/rc.local；而AIX 3.2.2则使用文件/etc/rc.net。

一些系统允许在某个文件中指定默认的路由器，如 /etc/defaultrouter。于是在每次重新启动系统时都要在路由表中加入该默认项。

初始化路由表的其他方法是运行路由守护程序（第 10章）或者用较新的路由器发现协议（9.6节）。

9.2.3 较复杂的路由表

在我们的子网上，主机 sun是所有主机的默认路由器，因为它有拨号 SLIP链路连接到Internet上（参见扉页前图）。

```
sun % netstat -rn
Routing tables
Destination       Gateway           Flags    Refcnt Use      Interface
140.252.13.65     140.252.13.35     UGH      0      171      le0
127.0.0.1         127.0.0.1         UH       1      766      lo0
140.252.1.183     140.252.1.29      UH       0      0        sl0
default           140.252.1.183     UG       1      2955     sl0
140.252.13.32     140.252.13.33     U        8      99551    le0
```

前两项与主机svr4的前两项一致：通过路由器bsdi到达slip的特定主机路由，以及环回路由。

第3行是新加的。这是一个直接到达主机的路由 (没有设置G标志，但设置了H标志)，对应于点对点的链路，即SLIP接口。如果我们把它与ifconfig命令的输出进行比较：

```
sun % ifconfig sl0
sl0: flags=1051<UP,POINTOPOINT,RUNNING>
      inet 140.252.1.29 --> 140.252.1.183 netmask ffffff00
```

可以发现路由表中的目的地址就是点对点链路的另一端 (即路由器netb)，网关地址为外出接口的本地IP地址(140.252.1.29)(前面已经说过，netstat为直接路由打印出来的网关地址就是本地接口所用的IP地址)。

默认的路由表项是一个到达网络的间接路由 (设置了G标志，但没有设置H标志)，这正是我们所希望的。网关地址是路由器的地址 (140.252.1.183，SLIP链路的另一端)，而不是SLIP链路的本地IP地址(140.252.1.29)。其原因还是因为是间接路由，不是直接路由。

还应该指出的是，netstat输出的第3和第4行(接口名为sl0)由SLIP软件在启动时创建，并在关闭时删除.

9.2.4 没有到达目的地的路由

我们所有的例子都假定对路由表的搜索能找到匹配的表项，即使匹配的是默认项。如果路由表中没有默认项，而又没有找到匹配项，这时会发生什么情况呢？

结果取决于该IP数据报是由主机产生的还是被转发的（例如，我们就充当一个路由器）。如果数据报是由本地主机产生的，那么就给发送该数据报的应用程序返回一个差错，或者是“主机不可达差错”或者是“网络不可达差错”。如果是被转发的数据报，那么就给原始发送

端发送一份ICMP主机不可达的差错报文。下一节将讨论这种差错。

9.3 ICMP主机与网络不可达差错

当路由器收到一份IP数据报但又不能转发时，就要发送一份 ICMP“主机不可达”差错报文（ICMP主机不可达报文的格式如图 6-10所示）。可以很容易发现，在我们的网络上把接在路由器sun上的拨号SLIP链路断开，然后试图通过该 SLIP链路发送分组给任何指定 sun为默认路由器的主机。

> 较老版本的BSD产生一个主机不可达或者网络不可达差错，这取决于目的端是否处于一个局域子网上。4.4 BSD只产生主机不可达差错。

我们在上一节通过在路由器 sun上运行netstat命令可以看到，当接通 SLIP链路启动时就要在路由表中增加一项使用 SLIP链路的表项，而当断开 SLIP链路时则删除该表项。这说明当SLIP链路断开时，sun的路由表中就没有默认项了。但是我们不想改变网络上其他主机的路由表，即同时删除它们的默认路由。相反，对于 sun不能转发的分组，我们对它产生的ICMP主机不可达差错报文进行计数。

在主机svr4上运行ping程序就可以看到这一点，它在拨号 SLIP链路的另一端（拨号链路已被断开）：

```
svr4 % ping gemini
ICMP Host Unreachable from gateway sun (140.252.13.33)
ICMP Host Unreachable from gateway sun (140.252.13.33)
^?                                          键入中断键停止显示
```

在主机bsdi上运行tcpdump命令的输出如图9-2所示。

```
1  0.0                  svr4 > gemini: icmp: echo request
2  0.00 (0.00)          sun > svr4: icmp: host gemini unreachable

3  0.99 (0.99)          svr4 > gemini: icmp: echo request
4  0.99 (0.00)          sun > svr4: icmp: host gemini unreachable
```

图9-2 响应ping 命令的ICMP主机不可达报文

当路由器sun发现找不到能到达主机gemini的路由时，它就响应一个主机不可达的回显请求报文。

如果把SLIP链路接到Internet上，然后试图ping一个与Internet没有连接的IP地址，那么应该会产生差错。但令人感兴趣的是，我们可以看到在返回差错报文之前，分组要在Internet上传送多远：

```
sun % ping 192.82.148.1
PING 192.82.148.1: 56 data bytes        该IP地址没有连接到Internet上
ICMP Host Unreachable from gateway enss142.UT.westnet.net (192.31.39.21)
 for icmp from sun (140.252.1.29) to 192.82.148.1
```

从图8-5可以看出，在发现该IP地址是无效的之前，该分组已通过了6个路由器。只有当它到达NSFNET骨干网的边界时才检测到差错。这说明，6个路由器之所以能转发分组是因为路由表中有默认项。只有当分组到达 NSFNET骨干网时，路由器才能知道每个连接到 Internet上的每个网络的信息。这说明许多路由器只能在局部范围内工作。

参考文献[Ford, Rekhter, and Braun 1993]定义了顶层选路域（top-level routing domain），由它来维护大多数Internet网站的路由信息，而不使用默认路由。他们指出，在 Internet上存在

5个这样的顶层选路域：NSFNET主干网、商业互联网交换（Commercial Internet Exchange: CIX）、NASA科学互联网（NASA Science Internet: NSI）、SprintLink以及欧洲IP主干网（EBONE）。

9.4 转发或不转发

前面我们已经提过几次，一般都假定主机不转发IP数据报，除非对它们进行特殊配置而作为路由器使用。如何进行这样的配置呢？

大多数伯克利派生出来的系统都有一个内核变量 `ipforwarding`，或其他类似的名字（参见附录E）。一些系统（如BSD/386和SVR4）只有在该变量值不为0的情况下才转发数据报。SunOS 4.1.x允许该变量可以有三个不同的值：-1表示始终不转发并且始终不改变它的值；0表示默认条件下不转发，但是当打开两个或更多个接口时就把该值设为 1；1表示始终转发。Solaris 2.x把这三个值改为0（始终不转发）、1（始终转发）和2（在打开两个或更多个接口时才转发）。

较早版本的4.2BSD主机在默认条件下可以转发数据报，这给没有进行正确配置的系统带来了许多问题。这就是内核选项为什么要设成默认的“始终不转发”的原因，除非系统管理员进行特殊设置。

9.5 ICMP重定向差错

当IP数据报应该被发送到另一个路由器时，收到数据报的路由器就要发送ICMP重定向差错报文给IP数据报的发送端。这在概念上是很简单的，正如图9-3所示的那样。只有当主机可以选择路由器发送分组的情况下，我们才可能看到ICMP重定向报文（回忆我们在图7-6中看过的例子）。

1) 我们假定主机发送一份IP数据报给R1。这种选路决策经常发生，因为R1是该主机的默认路由。

2) R1收到数据报并且检查它的路由表，发现R2是发送该数据报的下一站。当它把数据报发送给R2时，R1检测到它正在发送的接口与数据报到达接口是相同的（即主机和两个路由器所在的LAN）。这样就给路由器发送重定向报文给原始发送端提供了线索。

3) R1发送一份ICMP重定向报文给主机，告诉它以后把数据报发送给R2而不是R1。

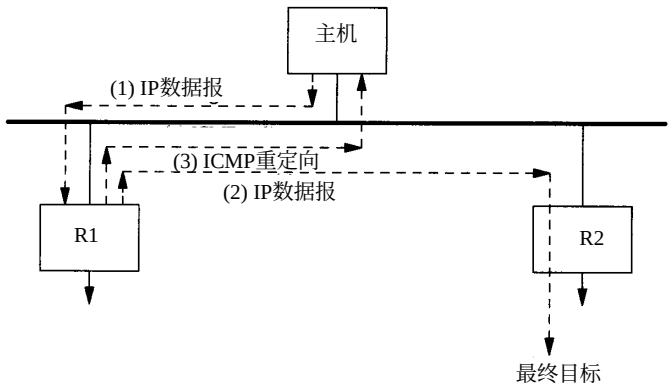


图9-3 ICMP重定向的例子

重定向一般用来让具有很少选路信息的主机逐渐建立更完善的路由表。主机启动时路由表中可以只有一个默认表项（在图 9-3所示的例子中，为 R1或R2）。一旦默认路由发生差错，默认路由器将通知它进行重定向，并允许主机对路由表作相应的改动。 ICMP重定向允许TCP/IP主机在进行选路时不需要具备智能特性，而把所有的智能特性放在路由器端。显然，在我们的例子中，R1和R2 必须知道有关相连网络的更多拓扑结构的信息，但是连在 LAN上的所有主机在启动时只需一个默认路由，通过接收重定向报文来逐步学习。

9.5.1 一个例子

可以在我们的网络上观察到 ICMP重定向的操作过程（见封二的图）。尽管在拓扑图中只画出了三台主机（aix,solaris和gemini）和两台路由器（gateway和netb），但是整个网络有超过150台主机和10台另外的路由器。大多数的主机都把 gateway指定为默认路由器，因为它提供了Internet的入口。

子网140.252.1上的主机是如何访问作者所在子网（图中底下的四台主机）的呢？首先，如果在SLIP链路的一端只有一台主机，那么就要使用代理 ARP（4.6节）。这意味着位于拓扑图顶部的子网（140.252.1）中的主机不需要其他特殊条件就可以访问主机 sun（140.252.1.29）。位于netb上的代理ARP软件处理这些事情。

但是，当网络位于 SLIP链路的另一端时，就要涉及到选路了。一个办法是让所有的主机和路由器都知道路由器 netb是网络140.252.13的网关。这可以在每个主机的路由表中设置静态路由，或者在每个主机上运行守护程序来实现。另一个更简单的办法（也是实际采用的方法）是利用ICMP重定向报文来实现。

在位于网络顶部的主机 solaris上运行ping程序到主机bsdi(140.252.13.35)。由于子网号不相同，代理 ARP不能使用。假定没有安装静态路由，发送的第一个分组将采用到路由器gateway的默认路由。下面是我们运行 ping程序之前的路由表：

```
solaris % netstat -rn
Routing Table:
  Destination        Gateway        Flags  Ref   Use   Interface
---------------- ---------------- ----- ----- ------ ---------
127.0.0.1        127.0.0.1         UH        0    848  lo0
140.252.1.0      140.252.1.32      U         3  15042  le0
224.0.0.0        140.252.1.32      U         3      0  le0
default          140.252.1.4       UG        0   5747
```

(224.0.0.0所在的表项是IP广播地址。我们将在第 12章讨论）。如果为ping程序指定 -v选项，可以看到主机接收到的任何ICMP报文。我们需要指定该选项以观察发送的重定向报文。

```
solaris % ping -sv bsdi
PING bsdi: 56 data bytes
ICMP Host redirect from gateway gateway (140.252.1.4)
 to netb (140.252.1.183) for bsdi (140.252.13.35)
64 bytes from bsdi (140.252.13.35): icmp_seq=0. time=383. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=1. time=364. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=2. time=353. ms
^?                                    键入中断键停止显示
----bsdi PING Statistics----
4 packets transmitted, 3 packets received, 25% packet loss
round-trip (ms)  min/avg/max = 353/366/383
```

在收到ping程序的第一个响应之前，主机先收到一份来自默认路由器 gateway发来的

ICMP重定向报文。如果这时查看路由表，就会发现已经插入了一个到主机 bsdi的新路由（该表项如以下黑体字所示）。

```
solaris % netstat -rn
Routing Table:
  Destination        Gateway        Flags  Ref    Use   Interface
---------------- ---------------- ----- ----- ------ ---------
127.0.0.1        127.0.0.1        UH        0    848  lo0
140.252.13.35    140.252.1.183    UGHD      0      2
140.252.1.0      140.252.1.32     U         3  15045  le0
224.0.0.0        140.252.1.32     U         3      0  le0
default          140.252.1.4      UG        0   5749
```

这是我们第一次看到D标志，表示该路由是被ICMP重定向报文创建的。G标志说明这是一份到达gateway(netb）的间接路由，H标志则说明这是一个主机路由（正如我们期望的那样），而不是一个网络路由。

由于这是一个被主机重定向报文增加的主机路由，因此它只处理到达主机 bsdi的报文。如果我们接着访问主机 svr4，那么就要产生另一个 ICMP重定向报文，创建另一个主机路由。类似地，访问主机 slip也创建另一个主机路由。位于子网上的三台主机（ bsdi,svr4和slip）还可以由一个指向路由器 sun的网络路由来进行处理。但是 ICMP重定向报文创建的是主机路由，而不是网络路由，这是因为在本例中，产生 ICMP重定向报文的路由器并不知道位于140.252.13网络上的子网信息。

9.5.2 更多的细节

ICMP重定向报文的格式如图9-4所示。

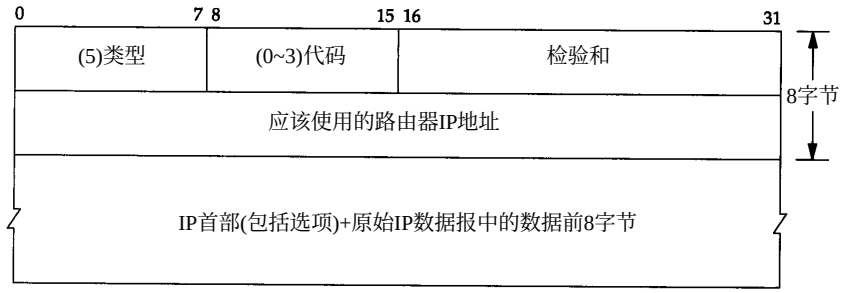


图9-4 ICMP重定向报文

有四种不同类型的重定向报文，有不同的代码值，如图 9-5所示。

ICMP重定向报文的接收者必须查看三个 IP地址：(1)导致重定向的IP地址（即ICMP重定向报文的数据位于IP数据报的首部）；(2)发送重定向报文的路由器的IP地址（包含重定向信息的 IP数据报中的源地址；(3)应该采用的路由器IP地址（在ICMP报文中的4~7字节）。

代码	描　述
0	网络重定向
1	主机重定向
2	服务类型和网络重定向
3	服务类型和主机重定向

图9-5 ICMP重定向报文的不同代码值

关于ICMP重定向报文有很多规则。首先，重定向报文只能由路由器生成，而不能由主机生成。另外，重定向报文是为主机而不是为路由器使用的。假定路由器和其他一些路由器共同参与某一种选路协议，则该协议就能消除重定向的需要（这意味着在图 9-1中的路由表应该

消除或者能被选路守护程序修改，或者能被重定向报文修改，但不能同时被二者修改）。

在4.4BSD系统中，当主机作为路由器使用时，要进行下列检查。在生成 ICMP重定向报文之前这些条件都要满足。

1) 出接口必须等于入接口。

2) 用于向外传送数据报的路由不能被 ICMP重定向报文创建或修改过，而且不能是路由器的默认路由。

3) 数据报不能用源站选路来转发。

4) 内核必须配置成可以发送重定向报文。

内核变量取名为ip_sendredirects或其他类似的名字（参见附录E）。大多数当前的系统（例如BSD、 SunOS 4.1.x、Solaris 2.x 及AIX 3.2.2）在默认条件下都设置该变量，使系统可以发送重定向报文。其他系统如SVR4则关闭了该项功能。

另外，一台4.4BSD主机收到ICMP重定向报文后，在修改路由表之前要作一些检查。这是为了防止路由器或主机的误操作，以及恶意用户的破坏，导致错误地修改系统路由表。

1) 新的路由器必须直接与网络相连接。

2) 重定向报文必须来自当前到目的地所选择的路由器。

3) 重定向报文不能让主机本身作为路由器。

4) 被修改的路由必须是一个间接路由。

关于重定向最后要指出的是，路由器应该发送的只是对主机的重定向（代码 1或3，如图9-5所示），而不是对网络的重定向。子网的存在使得难于准确指明何时应发送对网络的重定向而不是对主机的重定向。只当路由器发送了错误的类型时，一些主机才把收到的对网络的重定向当作对主机的重定向来处理。

9.6 ICMP路由器发现报文

在本章前面已提到过一种初始化路由表的方法，即在配置文件中指定静态路由。这种方法经常用来设置默认路由。另一种新的方法是利用 ICMP路由器通告和请求报文。

一般认为，主机在引导以后要广播或多播传送一份路由器请求报文。一台或更多台路由器响应一份路由器通告报文。另外，路由器定期地广播或多播传送它们的路由器通告报文，允许每个正在监听的主机相应地更新它们的路由表。

RFC 1256 [Deering 1991]确定了这两种ICMP报文的格式。ICMP路由器请求报文的格式如图9-6所示。ICMP路由器通告报文的格式如图 9-7所示。

路由器在一份报文中可以通告多个地址。地址数指的是报文中所含的地址数。地址项大小指的是每个路由器地址 32 bit字的数目，始终为 2。生存期指的是通告地址有效的时间（秒数）。

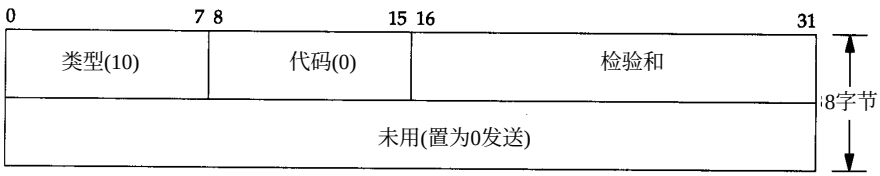


图9-6 ICMP路由器请求报文格式

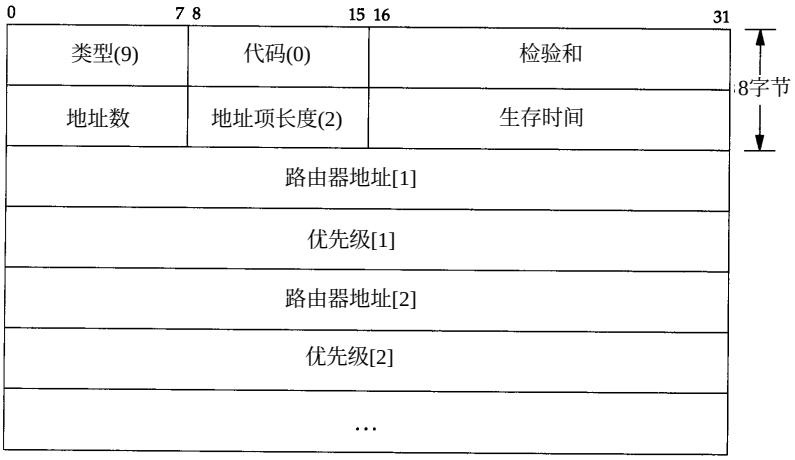


图9-7 ICMP路由器通告报文格式

接下来是一对或多对IP地址和优先级。IP地址必须是发送路由器的某个地址。优先级是一个有符号的32 bit整数，指出该IP地址作为默认路由器地址的优先等级，这是与子网上的其他路由器相比较而言的。值越大说明优先级越高。优先级为 0x80000000说明对应的地址不能作为默认路由器地址使用，尽管它也包含中通告报文中。优先级的默认值一般为 0。

9.6.1 路由器操作

当路由器启动时，它定期在所有广播或多播传送接口上发送通告报文。准确地说，这些通告报文不是定期发送的，而是随机传送的，以减小与子网上其他路由器发生冲突的概率。一般每两次通告间隔450秒和600秒。一份给定的通告报文默认生命周期是 30分钟。

使用生命周期域的另一个时机是当路由器上的某个接口被关闭时。在这种情况下，路由器可以在该接口上发送最后一份通告报文，并把生命周期值设为 0。

除了定期发送主动提供的通告报文以外，路由器还要监听来自主机的请求报文，并发送路由器通告报文以响应这些请求报文。

如果子网上有多台路由器，由系统管理员为每个路由器设置优先等级。例如，主默认路由器就要比备份路由器具有更高的优先级。

9.6.2 主机操作

主机在引导期间一般发送三份路由器请求报文，每三秒钟发送一次。一旦接收到一个有效的通告报文，就停止发送请求报文。

主机也监听来自相邻路由器的请求报文。这些通告报文可以改变主机的默认路由器。另外，如果没有接收到来自当前默认路由器的通告报文，那么默认路由器会超时。

只要有一般的默认路由器，该路由器就会每隔 10分钟发送通告报文，报文的生命周期是30分钟。这说明主机的默认表项是不会超时的，即使错过一份或两份通告报文。

9.6.3 实现

路由器发现报文一般由用户进程（守护程序）创建和处理。这样，在图 9-1中就有另一个

修改路由表的程序，尽管它只增加或删除默认表项。守护程序必须把它配置成一台路由器或主机来使用。

这两种ICMP报文是新加的，不是所有的系统都支持它们。在我们的网络中，只有Solaris 2.x支持这两种报文（`in.rdisc`守护程序）。尽管RFC建议尽可能用IP多播传送，但是路由器发现还可以利用广播报文来实现。

9.7 小结

IP路由操作对于运行TCP／IP的系统来说是最基本的，不管是主机还是路由器。路由表项的内容很简单，包括：5 bit标志、目的IP地址（主机、网络或默认）、下一站路由器的IP地址（间接路由）或者本地接口的IP地址（直接路由）及指向本地接口的指针。主机表项比网络表项具有更高的优先级，而网络表项比默认项具有更高的优先级。

系统产生的或转发的每份IP数据报都要搜索路由表，它可以被路由守护程序或ICMP重定向报文修改。系统在默认情况下不转发数据报，除非进行特殊的配置。用`route`命令可以进入静态路由，可以利用新ICMP路由器发现报文来初始化默认表项，并进行动态修改。主机在启动时只有一个简单的路由表，它可以被来自默认路由器的ICMP重定向报文动态修改。

在本章中，我们集中讨论了单个系统是如何利用路由表的。在下一章，我们将讨论路由器之间是如何交换路由信息的。

习题

9.1 为什么你认为存在两类ICMP重定向报文——网络和主机？

9.2 在9.4节开头列出的`svr4`主机上的路由表中，到主机`slip`（140.252.13.65）的特定路由是必需的吗？如果把这一项从路由表中删除会有什么变化？

9.3 考虑有一电缆连接4.2BSD主机和4.3BSD主机。假定网络号是140.1。4.2BSD主机把主机号为全0的地址识别为广播地址(140.1.0.0)，而4.3BSD通常使用全1的主机号（140.1.255.255）发送广播。另外，4.2BSD主机在默认条件下要尽力转发接收到的数据报，尽管它们只有一个接口。

请描述当4.2BSD主机收到一份目的地址为140.1.255.255的IP数据报时会发生什么事。

9.4 继续前一个习题，假定有人在子网140.1上的某个系统ARP高速缓存中增加了一项（用arp命令）内容，指定IP地址140.1.255.255对应的以太网地址为全1（以太网广播地址）。请描述此时发生的情况。

9.5 检查你所使用的系统上的路由表，并解释每一项内容。

第10章　动态选路协议

10.1　引言

在前面各章中，我们讨论了静态选路。在配置接口时，以默认方式生成路由表项（对于直接连接的接口），并通过route命令增加表项（通常从系统自引导程序文件），或是通过ICMP重定向生成表项（通常是在默认方式出错的情况下）。

在网络很小，且与其他网络只有单个连接点且没有多余路由时（若主路由失败，可以使用备用路由），采用这种方法是可行的。如果上述三种情况不能全部满足，通常使用动态选路。

本章讨论动态选路协议，它用于路由器间的通信。我们主要讨论　RIP，即选路信息协议(Routing Infromation Protocol)，大多数TCP/IP实现都提供这个应用广泛的协议。然后讨论两种新的选路协议，　OSPF和BGP。本章的最后研究一种名叫无分类域间选路的新的选路技术，现在Internet上正在开始采用该协议以保持B类网络的数量。

10.2　动态选路

当相邻路由器之间进行通信，以告知对方每个路由器当前所连接的网络，这时就出现了动态选路。路由器之间必须采用选路协议进行通信，这样的选路协议有很多种。路由器上有一个进程称为路由守护程序（ routing daemon），它运行选路协议，并与其相邻的一些路由器进行通信。正如图9-1所示，路由守护程序根据它从相邻路由器接收到的信息，更新内核中的路由表。

动态选路并不改变我们在 9.2节中所描述的内核在 IP层的选路方式。这种选路方式称为选路机制（routing mechanism）。内核搜索路由表，查找主机路由、网络路由以及默认路由的方式并没有改变。仅仅是放置到路由表中的信息改变了 —— 当路由随时间变化时，路由是由路由守护程序动态地增加或删除，而不是来自于自引导程序文件中的 route命令。

正如前面所描述的那样，路由守护程序将选路策略（ routing policy）加入到系统中，选择路由并加入到内核的路由表中。如果守护程序发现前往同一信宿存在多条路由，那么它（以某种方法）将选择最佳路由并加入内核路由表中。如果路由守护程序发现一条链路已经断开（可能是路由器崩溃或电话线路不好），它可以删除受影响的路由或增加另一条路由以绕过该问题。

在像Internet这样的系统中，目前采用了许多不同的选路协议。 Internet是以一组自治系统(AS，Autonomous System)的方式组织的，每个自治系统通常由单个实体管理。常常将一个公司或大学校园定义为一个自治系统。 NSFNET的Internet骨干网形成一个自治系统，这是因为骨干网中的所有路由器都在单个的管理控制之下。

每个自治系统可以选择该自治系统中各个路由器之间的选路协议。这种协议我们称之为内部网关协议IGP（Interior Gateway Protocol）或域内选路协议（intradomain routing protocol）。

最常用的IGP是选路信息协议 RIP。一种新的IGP是开放最短路径优先 OSPF（Open Shortest Path First）协议。它意在取代RIP。另一种1986年在原来NSFNET骨干网上使用的较早的IGP协议 —— HELLO，现在已经不用了。

新的RFC [Almquist 1993]规定，实现任何动态选路协议的路由器必须同时支持OSPF和RIP，还可以支持其他IGP协议。

外部网关协议EGP（Exterier Gateway Protocol）或域内选路协议的分隔选路协议用于不同自治系统之间的路由器。在历史上，（令人容易混淆）改进的EGP有着一个与它名称相同的协议：EGP。新EGP是当前在NSFNET骨干网和一些连接到骨干网的区域性网络上使用的是边界网关协议BGP（Border Gateway Protocol）。BGP意在取代EGP。

10.3 Unix选路守护程序

Unix系统上常常运行名为`routed`路由守护程序。几乎在所有的TCP/IP实现中都提供该程序。该程序只使用RIP进行通信，我们将在下一节中讨论该协议。这是一种用于小型到中型网络中的协议。

另一个程序是`gated`。IGP和EGP都支持它。[Fedor 1998]描述了早期开发的`gated`。图10-1对`routed`和两种不同版本的`gated`所支持的不同选路协议进行了比较。大多数运行路由守护程序的系统都可以运行`routed`，除非它们需要支持`gated`所支持的其他协议。

守护程序	内部网点协议			外部网点协议	
	HELLO	RIP	OSPF	EGP	BGP
routed		V1			
gated, 版本2	•	V1		•	V1
gated, 版本3	•	V1, V2	V2	•	V2, V3

图10-1 routed 和gated 所支持的选路协议

我们在下一节中描述 RIP 版本1，10.5节描述它与RIP版本2的不同点，10.6节描述OSPF，10.7节描述BGP。

10.4 RIP：选路信息协议

本节对RIP进行了描述，这是因为它是最广为使用（也是最受攻击）的选路协议。对于 RIP的正式描述文件是RFC 1058 [Hedrick 1988a]，但是该RFC是在该协议实现数年后才出现的。

10.4.1 报文格式

RIP报文包含中在UDP数据报中，如图10-2所示（在第11章中对UDP进行更为详细的描述）。

图10-3给出了使用IP地址时的RIP报文格式。

命令字段为1表示请求，2表示应答。还有两个舍弃不用的命令（3和4），两个非正式的命令：轮询（5）和轮询表项（6）。请求表示要求其他系统发送其全部或部分路由

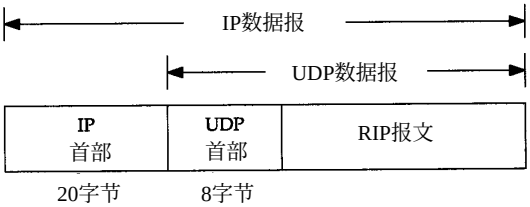


图10-2 封装在UDP数据报中的RIP报文

表。应答则包含发送者全部或部分路由表。

版本字段通常为1，而第2版RIP（10.5节）将此字段设置为2。

紧跟在后面的20字节指定地址系列（address family）（对于IP地址来说，其值是2）、IP地址以及相应的度量。在本节的后面可以看出， RIP的度量是以跳计数的。

采用这种20字节格式的RIP报文可以通告多达25条路由。上限25是用来保证RIP报文的总长度为20×25 + 4 = 504，小于512字节。由于每个报文最多携带 25个路由，因此为了发送整个路由表，经常需要多个报文。

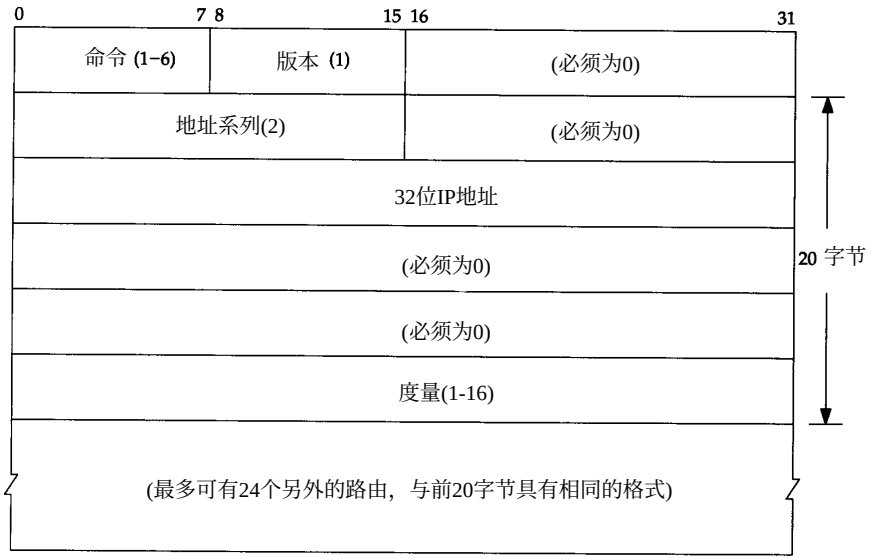


图 10-3

10.4.2 正常运行

让我们来看一下采用RIP协议的routed程序正常运行的结果。RIP常用的UDP端口号是520。

- 初始化：在启动一个路由守护程序时，它先判断启动了哪些接口，并在每个接口上发送一个请求报文，要求其他路由器发送完整路由表。在点对点链路中，该请求是发送给其他终点的。如果网络支持广播的话，这种请求是以广播形式发送的。目的 UDP端口号是520（这是其他路由器的路由守护程序端口号）。

  这种请求报文的命令字段为 1，但地址系列字段设置为 0，而度量字段设置为 16。这是一种要求另一端完整路由表的特殊请求报文。
- 接收到请求。如果这个请求是刚才提到的特殊请求，那么路由器就将完整的路由表发送给请求者。否则，就处理请求中的每一个表项：如果有连接到指明地址的路由，则将度量设置成我们的值，否则将度量置为 16（度量为16是一种称为“ 无穷大 ”的特殊值，它意味着没有到达目的的路由）。然后发回响应。
- 接收到响应。使响应生效，可能会更新路由表。可能会增加新表项，对已有的表项进行修改，或是将已有表项删除。
- 定期选路更新。每过30秒，所有或部分路由器会将其完整路由表发送给相邻路由器。发送路由表可以是广播形式的（如在以太网上），或是发送给点对点链路的其他终点的。

• 触发更新。每当一条路由的度量发生变化时，就对它进行更新。不需要发送完整路由表，而只需要发送那些发生变化的表项。

每条路由都有与之相关的定时器。如果运行 RIP的系统发现一条路由在 3分钟内未更新，就将该路由的度量设置成无穷大（ 16），并标注为删除。这意味着已经在 6个30秒更新时间里没收到通告该路由的路由器的更新了。再过 60秒，将从本地路由表中删除该路由，以保证该路由的失效已被传播开。

10.4.3 度量

RIP所使用的度量是以跳 (hop)计算的。所有直接连接接口的跳数为 1。考虑图10-4所示的路由器和网络。画出的 4条虚线是广播 RIP报文。

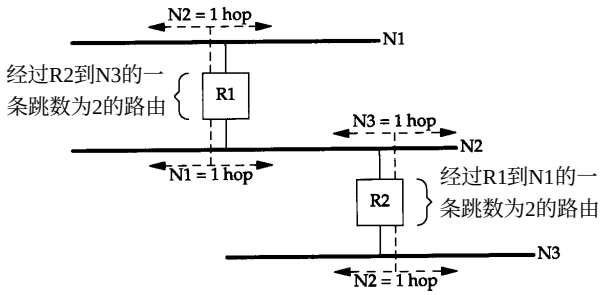


图10-4 路由器和网络示例

路由器R1通过发送广播到 N1通告它与N2之间的跳数是 1（发送给 N1的广播中通告它与N1之间的路由是无用的）。同时也通过发送广播给N2通告它与N1之间的跳数为1。同样， R2通告它与N2的度量为1，与N3的度量为1。

如果相邻路由器通告它与其他网络路由的跳数为1，那么我们与那个网络的度量就是 2，这是因为为了发送报文到该网络，我们必须经过那个路由器。在我们的例子中， R2到N1的度量是2，与R1到N3的度量一样。

由于每个路由器都发送其路由表给邻站，因此，可以判断在同一个自治系统 AS内到每个网络的路由。如果在该 AS内从一个路由器到一个网络有多条路由，那么路由器将选择跳数最小的路由，而忽略其他路由。

跳数的最大值是 15，这意味着 RIP只能用在主机间最大跳数值为 15的AS内。度量为16表示到无路由到达该IP地址。

10.4.4 问题

这种方法看起来很简单，但它有一些缺陷。首先， RIP没有子网地址的概念。例如，如果标准的B类地址中16 bit的主机号不为0，那么RIP无法区分非零部分是一个子网号，或者是一个主机地址。有一些实现中通过接收到的RIP信息，来使用接口的网络掩码，而这有可能出错。

其次，在路由器或链路发生故障后，需要很长的一段时间才能稳定下来。这段时间通常需要几分钟。在这段建立时间里，可能会发生路由环路。在实现 RIP时，必须采用很多微妙的措施来防止路由环路的出现，并使其尽快建立。 RFC 1058 [Hedrick 1988a]中指出了很多实现RIP的细节。

采用跳数作为路由度量忽略了其他一些应该考虑的因素。同时，度量最大值为 15则限制了可以使用RIP的网络的大小。

10.4.5 举例

我们将使用`ripquery`程序来查询一些路由器中的路由表，该程序可以从 `gated`中得到。

ripquery程序通过发送一个非正式请求（图10-3中命令字段为5的“poll”）给路由器，要求得到其完整的路由表。如果在5秒内未收到响应，则发送标准的RIP请求（command字段为1）（前面提到过的，将地址系列字段置为0，度量字段置为16的请求，要求其他路由器发送其完整路由表）。

图10-5给出了将从sun主机上查询其路由表的两个路由器。如果在主机sun上执行ripquery程序，以得到其下一站路由器netb的选路信息，那么可以得到下面的结果：

```
sun % ripquery -n netb
504 bytes from netb (140.252.1.183):      第一份报文包含504字节
                                          这里删除了许多行
      140.252.1.0, metric 1               图10-5中上面的以太网
      140.252.13.0, metric 1              图10-5中下面的以太网
244 bytes from netb (140.252.1.183):      第二份报文包含剩下的244字节下面删除了许多行
```

正如我们所猜想的那样，netb告诉我们子网的度量为1。另外，与netb相连的位于机端的以太网（140.252.1.0）的metric也是1（-n参数表示直接打印IP地址而不需要去查看其域名）。在本例中，将netb配置成认为所有位于140.252.13子网的主机都与其直接相连——即，netb并不知道哪些主机真正与140.252.13子网相连。由于与140.252.13子网只有一个连接点，因此，通告每个主机的度量实际上没有太大意义。

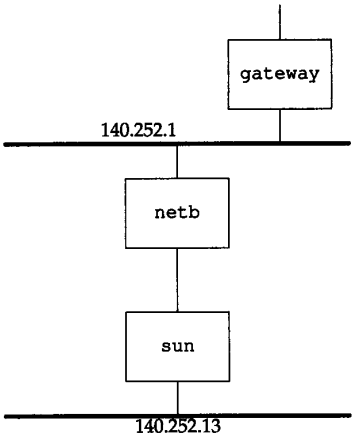


图10-5 查询其路由表内容的两个路由器netb和gateway

图10-6给出了使用tcpdump交换的报文。采用-i s10选项指定SLIP接口。

第1个请求发出一个RIP轮询命令（第1行）。这个请求在5秒后超时，发出一个常规的RIP请求（第2行）。第1行和第2行最后的24表示请求报文的长度：4个字节的RIP首部（包括命令和版本），然后是单个20字节的地址和度量。

第3行是第一个应答报文。该行最后的25表示包含了25个地址和度量对，我们在前面已经计算过，其字节数为504。这是上面的ripquery程序所打印出来的结果。我们为tcpdump程序指定-s600选项，以让它从网络中读取600个字节。这样，它可以接收整个UDP数据报（而不是报文的前半部），然后打印出RIP响应的内容。该输出结果省略了。

```
   sun % tcpdump -s600 -i sl0
1  0.0                    sun.2879 > netb.route: rip-poll 24
2  5.014702 (5.0147)      sun.2879 > netb.route: rip-req 24
3  5.560427 (0.5457)      netb.route > sun.2879: rip-resp 25:
4  5.710251 (0.1498)      netb.route > sun.2879: rip-resp 12:
```

图10-6 运行ripquery程序的tcpdump输出结果

第4行是来自路由器的第二个响应报文，它包含后面的12个地址和度量对。可以计算出该报文的长度为12×20 + 4=244，这正是ripquery程序所打印出来的结果。

如果越过netb路由器，到gateway，那么可以预测到我们子网（140.252.13.0）的度量为2。可以运行下面的命令来进行验证：

```
sun % ripquery -n gateway
504 bytes from gateway (140.252.1.4):
                                                    这里删除了许多行
      140.252.1.0, metric 1          图10-5上面的以太网
      140.252.13.0, metric 2                    图10-5下面的以太网
```

这里，位于图10-5上面的以太网（140.252.1.0）的度量依然是1，这是因为该以太网直接与gateway和netb相连。而我们的子网140.252.13.0正如预想的一样，其度量为2。

10.4.6 另一个例子

现在察看以太网上所有非主动请求的RIP更新，以看一看RIP定期给其邻站发送的信息。图10-7是noao.edu网络的多种排列情况。为了简化，我们不用本文其他地方所采用的路由器表示方式，而以Rn来代表路由器，其中n是子网号。以虚线表示点对点链路，并给出了这些链路对端的IP地址。

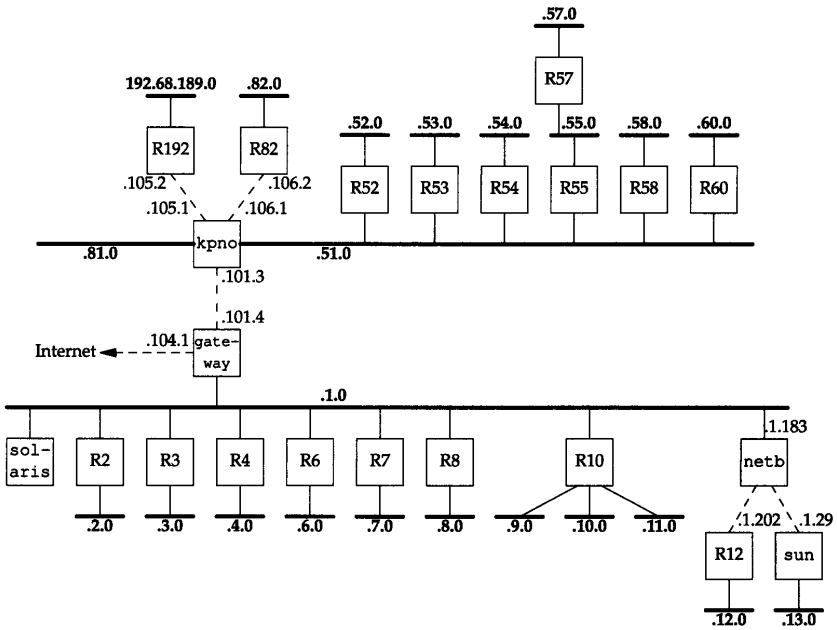


图10-7 noao.edu 140.252的多个网络

在主机solaris上运行Solaris 2.x的snoop程序，它与tcpdump相类似。我们可以在不需要超用户权限的条件下运行该程序，但它只捕获广播报文、多播报文以及发送给主机的报文。图10-8给出了在60秒内所捕获的报文。在这里，我们将大部分正式的主机名以Rn来表示。

-P标志以非混杂模式捕获报文，-tr打印出相应的时戳，而udp port 520捕获信源或信宿端口号为520的UDP数据报。

来自R6、R4、R2、R7、R8和R3的前6个报文，每个报文只通告一个网络。查看这些报文，可以发现R2通告前往140.252.6.0的跳数为1的一条路由，R4通告前往140.252.4.0的跳数为1的一条路由，等等。

但是，gateway路由器却通告了15条路由。我们可以通过运行snoop程序时加上-v参数来查看RIP报文的全部内容（这个标志输出全部报文的全部内容：以太网首部、IP首部、UDP首部以及RIP报文。我们只保留了RIP信息而删除了其他信息）。图10-9给出了输出结果。

```
solaris % snoop -P -tr udp port 520
  0.00000 R6.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
  4.49708 R4.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
  6.30506 R2.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 11.68317 R7.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 16.19790 R8.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 16.87131 R3.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 17.02187 gateway.tuc.noao.edu -> 140.252.1.255 RIP R (15 destinations)
 20.68009 R10.tuc.noao.edu -> BROADCAST    RIP R (4 destinations)

 29.87848 R6.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 34.50209 R4.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 36.32385 R2.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 41.34565 R7.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 46.19257 R8.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 46.52199 R3.tuc.noao.edu -> 140.252.1.255 RIP R (1 destinations)
 47.01870 gateway.tuc.noao.edu -> 140.252.1.255 RIP R (15 destinations)
 50.66453 R10.tuc.noao.edu -> BROADCAST    RIP R (4 destinations)
```

图10-8 solaris 在60秒内所捕获到的RIP广播报文

把这些子网140.252.1上通告报文经过的路由与图10-7中的拓扑结构进行比较。

使人迷惑不解的一个问题是为什么图 10-8输出结果中， R10通告其有4个网络而在图10-7中显示的只有3个。如果查看带snoop的RIP报文，就会得到以下通告路由：

```
RIP: Address         Metric
RIP:  140.251.0.0     16 (not reachable)
RIP:  140.252.9.0     1
RIP:  140.252.10.0    1
RIP:  140.252.11.0    1
```

前往B类网络140.251的路由是假的，不应该通告它（它属于其他机构而不是noao.edu）。

```
solaris % snoop -P -v -tr udp port 520 host gateway
                                        删去许多行
RIP:  Opcode = 2 (route response)
RIP:  Version = 1

RIP:  Address         Metric

RIP:  140.252.101.0   1
RIP:  140.252.104.0   1

RIP:  140.252.51.0    2
RIP:  140.252.81.0    2
RIP:  140.252.105.0   2
RIP:  140.252.106.0   2

RIP:  140.252.52.0    3
RIP:  140.252.53.0    3
RIP:  140.252.54.0    3
RIP:  140.252.55.0    3
RIP:  140.252.58.0    3
RIP:  140.252.60.0    3
RIP:  140.252.82.0    3
RIP:  192.68.189.0    3

RIP:  140.252.57.0    4
```

图10-9 来自gateway 的RIP响应

图10-8中，对于 R10发送的RIP报文， snoop输出“ BROADCAST ”符号，它表示目的IP地址是有限的广播地址255.255.255.255（12.2节），而不是其他路由器用来指向子网的广播地

址（140.252.1.255）。

10.5 RIP版本2

RFC 1388 [Malkin 1993a]中对RIP定义进行了扩充，通常称其结果为 RIP-2。这些扩充并不改变协议本身，而是利用图10-3中的一些标注为“必须为0”的字段来传递一些额外的信息。如果RIP忽略这些必须为0的字段，那么，RIP和RIP-2可以互操作。

图10-10重新给出了由RIP-2定义的图。对于RIP-2来说，其版本字段为2。

图10-10 RIP-2报文格式

选路域(routing domain)是一个选路守护程序的标识符，它指出了这个数据报的所有者。在一个Unix实现中，它可以是选路守护程序的进程号。该域允许管理者在单个路由器上运行多个RIP实例，每个实例在一个选路域内运行。

选路标记(routing tag)是为了支持外部网关协议而存在的。它携带着一个 EGP和BGP的自治系统号。

每个表项的子网掩码应用于相应的IP地址上。下一站IP地址指明发往目的IP地址的报文该发往哪里。该字段为0意味着发往目的地址的报文应该发给发送 RIP报文的系统。

RIP-2提供了一种简单的鉴别机制。可以指定 RIP报文的前20字节表项地址系列为 0xffff，路由标记为2。表项中的其余16字节包含一个明文口令。

最后，RIP-2除了广播（第12章）外，还支持多播。这可以减少不收听 RIP-2报文的主机的负载。

10.6 OSPF：开放最短路径优先

OSPF是除RIP外的另一个内部网关协议。它克服了 RIP的所有限制。 RFC 1247 [Moy 1991]中对第2版OSPF进行了描述。

与采用距离向量的 RIP协议不同的是， OSPF是一个链路状态协议。距离向量的意思是，RIP发送的报文包含一个距离向量（跳数）。每个路由器都根据它所接收到邻站的这些距离向

量来更新自己的路由表。

在一个链路状态协议中，路由器并不与其邻站交换距离信息。它采用的是每个路由器主动地测试与其邻站相连链路的状态，将这些信息发送给它的其他邻站，而邻站将这些信息在自治系统中传播出去。每个路由器接收这些链路状态信息，并建立起完整的路由表。

从实际角度来看，二者的不同点是链路状态协议总是比距离向量协议收敛更快。收敛的意思是在路由发生变化后，例如在路由器关闭或链路出故障后，可以稳定下来。 [Perlman 1992]的9.3节对这两种类型的选路协议的其他方面进行了比较。

OSPF与RIP（以及其他选路协议）的不同点在于， OSPF直接使用IP。也就是说，它并不使用UDP或TCP。对于IP首部的protocol字段，OSPF有其自己的值（图3-1）。

另外，作为一种链路状态协议而不是距离向量协议， OSPF还有着一些优于RIP的特点。

1) OSPF可以对每个IP服务类型（图3-2）计算各自的路由集。这意味着对于任何目的，可以有多个路由表表项，每个表项对应着一个IP服务类型。

2) 给每个接口指派一个无维数的费用。可以通过吞吐率、往返时间、可靠性或其他性能来进行指派。可以给每个IP服务类型指派一个单独的费用。

3) 当对同一个目的地址存在着多个相同费用的路由时，OSPF在这些路由上平均分配流量。我们称之为流量平衡。

4) OSPF支持子网：子网掩码与每个通告路由相连。这样就允许将一个任何类型的 IP地址分割成多个不同大小的子网（我们在 3.7节中给出了这样的一个例子，称之为变长度子网）。到一个主机的路由是通过全1子网掩码进行通告的。默认路由是以IP地址为0.0.0.0、网络掩码为全0进行通告的。

5) 路由器之间的点对点链路不需要每端都有一个 IP地址，我们称之为无编号网络。这样可以节省IP地址 —— 现在非常紧缺的一种资源。

6) 采用了一种简单鉴别机制。可以采用类似于 RIP-2机制（10.5节）的方法指定一个明文口令。

7) OSPF采用多播（第12章），而不是广播形式，以减少不参与OSPF的系统负载。

随着大部分厂商支持OSPF，在很多网络中OSPF将逐步取代RIP。

10.7 BGP：边界网关协议

BGP是一种不同自治系统的路由器之间进行通信的外部网关协议。 BGP是ARPANET所使用的老EGP的取代品。RFC1267 [Lougheed and Rekhter 1991] 对第3版的BGP进行了描述。

RFC 1268 [Rekhter and Gross 1991] 描述了如何在Internet中使用BGP。下面对于BGP的大部分描述都来自于这两个RFC文档。同时， 1993年开发第4版的BGP（见RFC 1467 [Topolcic 1993]），以支持我们将在10.8节描述的CIDR。

BGP系统与其他BGP系统之间交换网络可到达信息。这些信息包括数据到达这些网络所必须经过的自治系统AS中的所有路径。这些信息足以构造一幅自治系统连接图。然后，可以根据连接图删除选路环，制订选路策略。

首先，我们将一个自治系统中的IP数据报分成本地流量和通过流量。在自治系统中，本地流量是起始或终止于该自治系统的流量。也就是说，其信源 IP地址或信宿IP地址所指定的主机位于该自治系统中。其他的流量则称为通过流量。在 Internet中使用BGP的一个目的就是

减少通过流量。

可以将自治系统分为以下几种类型：

1) 残桩自治系统(stub AS)，它与其他自治系统只有单个连接。stub AS只有本地流量。

2) 多接口自治系统(multihomed AS)，它与其他自治系统有多个连接，但拒绝传送通过流量。

3) 转送自治系统(transit AS)，它与其他自治系统有多个连接，在一些策略准则之下，它可以传送本地流量和通过流量。

这样，可以将Internet的总拓扑结构看成是由一些残桩自治系统、多接口自治系统以及转送自治系统的任意互连。残桩自治系统和多接口自治系统不需要使用 BGP——它们通过运行EGP在自治系统之间交换可到达信息。

BGP允许使用基于策略的选路。由自治系统管理员制订策略，并通过配置文件将策略指定给BGP。制订策略并不是协议的一部分，但指定策略允许 BGP实现在存在多个可选路径时选择路径，并控制信息的重发送。选路策略与政治、安全或经济因素有关。

BGP与RIP和OSPF的不同之处在于BGP使用TCP作为其传输层协议。两个运行BGP的系统之间建立一条TCP连接，然后交换整个BGP路由表。从这个时候开始，在路由表发生变化时，再发送更新信号。

BGP是一个距离向量协议，但是与（通告到目的地址跳数的） RIP不同的是，BGP列举了到每个目的地址的路由（自治系统到达目的地址的序列号）。这样就排除了一些距离向量协议的问题。采用16 bit数字表示自治系统标识。

BGP通过定期发送keepalive报文给其邻站来检测TCP连接对端的链路或主机失败。两个报文之间的时间间隔建议值为 30秒。应用层的 keepalive报文与TCP的keepalive选项（第23章）是独立的。

10.8 CIDR：无类型域间选路

在第3章中，我们指出了B类地址的缺乏，因此现在的多个网络站点只能采用多个 C类网络号，而不采用单个B类网络号。尽管分配这些C类地址解决了一个问题（B类地址的缺乏），但它却带来了另一个问题：每个 C类网络都需要一个路由表表项。无类型域间选路（ CIDR）是一个防止Internet路由表膨胀的方法，它也称为超网（ supernetting）。在RFC 1518 [Rekhter and Li 1993] 和RFC 1519 [Fuller et al. 1993]中对它进行了描述，而[Ford, Rekhter, and Braun 1993]是它的综述。CIDR有一个Internet Architecture Board ’s blessing [Huitema 1993]。RFC 1467 [Topolcic 1993] 对Internet中CIDR的开发状况进行了小结。

CIDR的基本观点是采用一种分配多个 IP地址的方式，使其能够将路由表中的许多表项总和(summarization)成更少的数目。例如，如果给单个站点分配 16个C类地址，以一种可以用总和的方式来分配这 16个地址，这样，所有这 16个地址可以参照Internet上的单个路由表表项。同时，如果有8个不同的站点是通过同一个 Internet服务提供商的同一个连接点接入 Internet的，且这8个站点分配的8个不同IP地址可以进行总和，那么，对于这 8个站点，在Internet上，只需要单个路由表表项。

要使用这种总和，必须满足以下三种特性：

1) 为进行选路要对多个IP地址进行总和时，这些IP地址必须具有相同的高位地址比特。

2) 路由表和选路算法必须扩展成根据32 bit IP地址和32 bit掩码做出选路决策。

3) 必须扩展选路协议使其除了32 bit地址外，还要有32 bit掩码。OSPF（10.6节）和RIP-2（10.5节）都能够携带第4版BGP所提出的32 bit掩码。

例如，RFC 1466 [Gerich 1993] 建议欧洲新的C类地址的范围是194.0.0.0～195.255.255.255。以16进制表示，这些地址的范围是0xc2000000～0xc3ffffff。它代表了65536个不同的C类网络号，但它们地址的高7 bit是相同的。在欧洲以外的国家里，可以采用IP地址为0xc2000000和32 bit 0xfe000000 (254.0.0.0) 为掩码的单个路由表表项来对所有这些65536个C类网络号选路到单个点上。C类地址的后面各比特位（即在194或195后面各比特）也可以进行层次分配，例如以国家或服务提供商分配，以允许对在欧洲路由器之间使用除了这32 bit掩码的高7 bit外的其他比特进行概括。

CIDR同时还使用一种技术，使最佳匹配总是最长的匹配：即在32 bit掩码中，它具有最大值。我们继续采用上一段中所用的例子，欧洲的一个服务提供商可能会采用一个与其他欧洲服务提供商不同的接入点。如果给该提供商分配的地址组是从194.0.16.0到194.0.31.255 (16个C类网络号)，那么可能只有这些网络的路由表项的IP地址是194.0.16.0，掩码为255.255.240.0 (0xfffff000)。发往194.0.22.1地址的数据报将同时与这个路由表表项和其他欧洲C类地址的表项进行匹配。但是由于掩码255.255.240比254.0.0.0更“长”，因此将采用具有更长掩码的路由表表项。

“无类型”的意思是现在的选路决策是基于整个32 bit IP地址的掩码操作，而不管其IP地址是A类、B类或是C类，都没有什么区别。

CIDR最初是针对新的C类地址提出的。这种变化将使Internet路由表增长的速度缓慢下来，但对于现存的选路则没有任何帮助。这是一个短期解决方案。作为一个长期解决方案，如果将CIDR应用于所有IP地址，并根据各洲边界和服务提供商对已经存在的IP地址进行重新分配（且所有现有主机重新进行编址！），那么[Ford, Rekhter, and Braun 1993] 宣称，目前包含10 000网络表项的路由表将会减少成只有200个表项。

10.9 小结

有两种基本的选路协议，即用于同一自治系统各路由器之间的内部网关协议（IGP）和用于不同自治系统内路由器通信的外部网关协议（EGP）。

最常用的IGP是路由信息协议（RIP），而OSPF是一个正在得到广泛使用的新IGP。一种新近流行的EGP是边界网关协议（BGP）。在本章中，我们讨论了RIP及其交换的报文类型。第2版RIP是其最近的一个改进版，它支持子网，还有一些其他改进技术。同时也对OSPF、BGP和无类型域间选路（CIDR）进行了描述。CIDR是一种新技术，可以减小Internet路由表的大小。

你可能还会遇到一些其他的OSI选路协议。域间选路协议（IDRP）最开始时，是一个为了使用OSI地址而不是IP地址，而进行修改的BGP版本。Intermediate System to Intermediate System 协议（IS-IS）是OSI的标准IGP。可以用它来选路CLNP（无连接网络协议），这是一种与IP类似的OSI协议。IS-IS和OSPF相似。

动态选路仍然是一个网间互连的研究热点。对使用的选路协议和运行的路由守护程序进行选择，是一项复杂的工作。[Perlman 1992]提供了许多细节。

习题

10.1 在图10-9中哪些路由是从路由器kpno进入gateway的？

10.2 假设一个路由器要使用RIP通告30个路由，这需要一个包含25条路由和另一个包含5条路由的数据报。如果每过一个小时，第一个包含25条路由的数据报丢失一次，那么其结果如何？

10.3 OSPF报文格式中有一个检验和字段，而RIP报文则没有此项，这是为什么？

10.4 像OSPF这样的负载平衡，对于传输层的影响是什么？

10.5 查阅RFC1058 关于实现RIP的其他资料。在图10-8中，140.252.1网络的每个路由器只通告它所提供的路由，而它并不能通过其他路由器的广播中知道任何其他路由。这种技术的名称是什么？

10.6 在3.4节中，我们说过除了图10-7中所示的8个路由器外，140.252.1子网上还有超过100个主机。那么这100个主机是如何处理每30秒到达它们的8个广播信息呢（图10-8）？

第11章 UDP：用户数据报协议

11.1 引言

UDP是一个简单的面向数据报的运输层协议：进程的每个输出操作都正好产生一个 UDP数据报，并组装成一份待发送的 IP数据报。这与面向流字符的协议不同，如 TCP，应用程序产生的全体数据与真正发送的单个 IP数据报可能没有什么联系。

UDP数据报封装成一份 IP数据报的格式如图11-1所示。

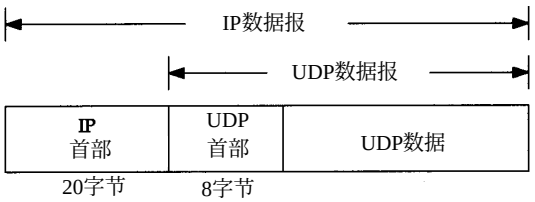


图11-1 UDP封装

RFC 768 [Postel 1980] 是UDP的正式规范。

UDP不提供可靠性：它把应用程序传给 IP层的数据发送出去，但是并不保证它们能到达目的地。由于缺乏可靠性，我们似乎觉得要避免使用 UDP而使用一种可靠协议如 TCP。我们在第17章讨论完TCP后将再回到这个话题，看看什么样的应用程序可以使用 UDP。

应用程序必须关心IP数据报的长度。如果它超过网络的 MTU（2.8节），那么就要对IP数据报进行分片。如果需要，源端到目的端之间的每个网络都要进行分片，并不只是发送端主机连接第一个网络才这样做（我们在 2.9节中已定义了路径 MTU的概念）。在11.5节中，我们将讨论IP分片机制。

11.2 UDP首部

UDP首部的各字段如图11-2所示。

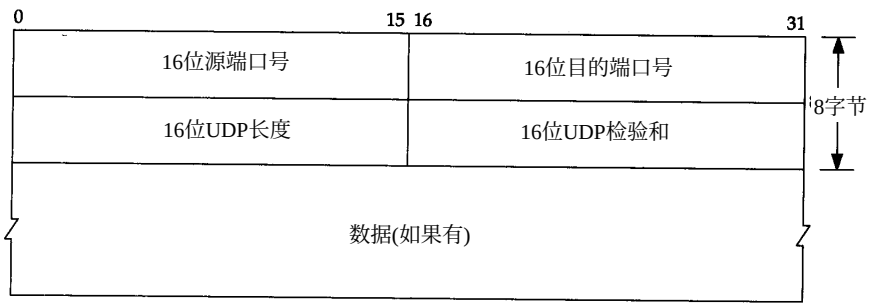


图11-2 UDP首部

端口号表示发送进程和接收进程。在图 1-8中，我们画出了TCP和UDP用目的端口号来分用来自IP层的数据的过程。由于IP层已经把IP数据报分配给TCP或UDP（根据IP首部中协议字段值），因此TCP端口号由TCP来查看，而UDP端口号由UDP来查看。TCP端口号与UDP端口号是相互独立的。

尽管相互独立，如果TCP和UDP同时提供某种知名服务，两个协议通常选择相同的端口号。这纯粹是为了使用方便，而不是协议本身的要求。

UDP长度字段指的是UDP首部和UDP数据的字节长度。该字段的最小值为8字节（发送一份0字节的UDP数据报是OK）。这个UDP长度是有冗余的。IP数据报长度指的是数据报全长（图3-1），因此UDP数据报长度是全长减去IP首部的长度（该值在首部长度字段中指定，如图3-1所示）。

11.3 UDP检验和

UDP检验和覆盖UDP首部和UDP数据。回想IP首部的检验和，它只覆盖IP的首部——并不覆盖IP数据报中的任何数据。

UDP和TCP在首部中都有覆盖它们首部和数据的检验和。UDP的检验和是可选的，而TCP的检验和是必需的。

尽管UDP检验和的基本计算方法与我们在3.2节中描述的IP首部检验和计算方法相类似（16 bit字的二进制反码和），但是它们之间存在不同的地方。首先，UDP数据报的长度可以为奇数字节，但是检验和算法是把若干个16 bit字相加。解决方法是必要时在最后增加填充字节0，这只是为了检验和的计算（也就是说，可能增加的填充字节不被传送）。

其次，UDP数据报和TCP段都包含一个12字节长的伪首部，它是为了计算检验和而设置的。伪首部包含IP首部一些字段。其目的是让UDP两次检查数据是否已经正确到达目的地（例如，IP没有接受地址不是本主机的数据报，以及IP没有把应传给另一高层的数据报传给UDP）。UDP数据报中的伪首部格式如图11-3所示。

图11-3 UDP检验和计算过程中使用的各个字段

在该图中，我们特地举了一个奇数长度的数据报例子，因而在计算检验和时需要加上填充字节。注意，UDP数据报的长度在检验和计算过程中出现两次。

如果检验和的计算结果为0，则存入的值为全1（65535），这在二进制反码计算中是等效的。如果传送的检验和为0，说明发送端没有计算检验和。

如果发送端没有计算检验和而接收端检测到检验和有差错，那么 UDP数据报就要被悄悄地丢弃。不产生任何差错报文（当IP层检测到IP首部检验和有差错时也这样做）。

UDP检验和是一个端到端的检验和。它由发送端计算，然后由接收端验证。其目的是为了发现UDP首部和数据在发送端到接收端之间发生的任何改动。

尽管UDP检验和是可选的，但是它们应该总是在用。在 80年代，一些计算机产商在默认条件下关闭UDP检验和的功能，以提高使用 UDP协议的NFS（Network File System）的速度。在单个局域网中这可能是可以接受的，但是在数据报通过路由器时，通过对链路层数据帧进行循环冗余检验（如以太网或令牌环数据帧）可以检测到大多数的差错，导致传输失败。不管相信与否，路由器中也存在软件和硬件差错，以致于修改数据报中的数据。如果关闭端到端的UDP检验和功能，那么这些差错在 UDP数据报中就不能被检测出来。另外，一些数据链路层协议（如SLIP）没有任何形式的数据链路检验和。

Host Requirements RFC声明，UDP检验和选项在默认条件下是打开的。它还声明，如果发送端已经计算了检验和，那么接收端必须检验接收到的检验和（如接收到检验和不为0）。但是，许多系统没有遵守这一点，只是在出口检验和选项被打开时才验证接收到的检验和。

11.3.1 tcpdump输出

很难知道某个特定系统是否打开了 UDP检验和选项。应用程序通常不可能得到接收到的UDP首部中的检验和。为了得到这一点，作者在 tcpdump程序中增加了一个选项，以打印出接收到的UDP检验和。如果打印出的值为0，说明发送端没有计算检验和。

测试网络上三个不同系统的输出如图 11-4所示（参见封面二）。运行我们自编的sock程序（附录C），发送一份包含9个字节数据的UDP数据报给标准回显服务器。

```
1   0.0                 sun.1900 > gemini.echo: udp 9 (UDP cksum=6e90)
2   0.303755 ( 0.3038)  gemini.echo > sun.1900: udp 9 (UDP cksum=0)

3  17.392480 (17.0887)  sun.1904 > aix.echo: udp 9 (UDP cksum=6e3b)
4  17.614371 ( 0.2219)  aix.echo > sun.1904: udp 9 (UDP cksum=6e3b)

5  32.092454 (14.4781)  sun.1907 > solaris.echo: udp 9 (UDP cksum=6e74)
6  32.314378 ( 0.2219)  solaris.echo > sun.1907: udp 9 (UDP cksum=6e74)
```

图11-4 tcpdump 输出，观察其他主机是否打开UDP检验和选项

从这里可以看出，三个系统中有两个打开了 UDP检验和选项。

还要注意的是，在这个简单例子中，送出的数据报与收到的数据报具有相同的检验和值（第3和第4行，第5和第6行）。从图11-3可以看出，两个IP地址进行了交换，正如两个端口号一样。伪首部和UDP首部中的其他字段都是相同的，就像数据回显一样。这再次表明 UDP检验和（事实上，TCP/IP协议簇中所有的检验和）是简单的16 bit和。它们检测不出交换两个16 bit的差错。

作者在14.2节中在8个域名服务器中各进行了一次DNS查询。DNS主要使用UDP，结果只有两台服务器打开了UDP检验和选项。

11.3.2 一些统计结果

文献[Mogul 1992]提供了在一个繁忙的NFS服务器上所发生的不同检验和差错的统计结果，

时间持续了40天。统计数字结果如图 11-5所示。

层　次	检验和差错数	近似总分组数
以太网	446	170 000 000
IP	14	170 000 000
UDP	5	140 000 000
TCP	350	30 000 000

图11-5　检测到不同检验和差错的分组统计结果

最后一列是每一行的大概总数，因为以太网和IP层还使用其他的协议。例如，不是所有的以太网数据帧都是 IP数据报，至少以太网还要使用 ARP协议。不是所有的 IP数据报都是UDP或TCP数据，因为ICMP也用IP传送数据。

注意，TCP发生检验和差错的比例与 UDP相比要高得多。这很可能是因为在该系统中的TCP连接经常是“远程”连接（经过许多路由器和网桥等中间设备），而UDP一般为本地通信。

从最后一行可以看出，不要完全相信数据链路（如以太网，令牌环等）的 CRC检验。应该始终打开端到端的检验和功能。而且，如果你的数据很有价值，也不要完全相信　UDP或TCP的检验和，因为这些都只是简单的检验和，不能检测出所有可能发生的差错。

11.4　一个简单的例子

用我们自己编写的sock程序生成一些可以通过tcpdump观察的UDP数据报：

```
bsdi %sock -v -u -i -n4 svr4 discard
connected on 140.252.13.35.1108 to 140.252.13.34.9
bsdi %sock -v -u -i -n4 -w0 svr4 discard
connected on 140.252.13.35.1110 to 140.252.13.34.9
```

第1次执行这个程序时，我们指定 verbose模式（-v）来观察ephemeral端口号，指定UDP（-u）而不是默认的TCP，并且指定源模式（-i）来发送数据，而不是读写标准的输入和输出。-n4选项指明输出4份数据报（默认条件下为1024），目的主机为svr4。在1.12节描述了丢弃服务。每次写操作的输出长度取默认值 1024。

第2次运行该程序时我们指定 -w0，意思是写长度为0的数据报。两个命令的 tcpdump输出结果如图11-6所示。

```
1   0.0                 bsdi.1108 > svr4.discard: udp 1024
2   0.002424 ( 0.0024)  bsdi.1108 > svr4.discard: udp 1024
3   0.006210 ( 0.0038)  bsdi.1108 > svr4.discard: udp 1024
4   0.010276 ( 0.0041)  bsdi.1108 > svr4.discard: udp 1024
5  41.720114 (41.7098)  bsdi.1110 > svr4.discard: udp 0
6  41.721072 ( 0.0010)  bsdi.1110 > svr4.discard: udp 0
7  41.722094 ( 0.0010)  bsdi.1110 > svr4.discard: udp 0
8  41.723070 ( 0.0010)  bsdi.1110 > svr4.discard: udp 0
```

图11-6　向一个方向发送UDP数据报时的tcpdump 输出

输出显示有四份1024字节的数据报，接着有四份长度为 0的数据报。每份数据报间隔几毫秒（输入第2个命令花了41秒的时间）。

在发送第1份数据报之前，发送端和接收端之间没有任何通信（在第 17章，我们将看到TCP在发送数据的第1个字节之前必须与另一端建立连接）。另外，当收到数据时，接收端没有任何确认。在这个例子中，发送端并不知道另一端是否已经收到这些数据报。

最后要指出的是，每次运行程序时，源端的 UDP端口号都发生变化。第一次是 1108，然后是110。在1.9节我们已经提过，客户程序使用 ephemeral端口号一般在1024～5000之间，正

如我们现在看到的这样。

11.5 IP分片

正如我们在2.8节描述的那样，物理网络层一般要限制每次发送数据帧的最大长度。任何时候IP层接收到一份要发送的IP数据报时，它要判断向本地哪个接口发送数据（选路），并查询该接口获得其MTU。IP把MTU与数据报长度进行比较，如果需要则进行分片。分片可以发生在原始发送端主机上，也可以发生在中间路由器上。

把一份IP数据报分片以后，只有到达目的地才进行重新组装（这里的重新组装与其他网络协议不同，它们要求在下一站就进行进行重新组装，而不是在最终的目的地）。重新组装由目的端的IP层来完成，其目的是使分片和重新组装过程对运输层（ TCP和UDP）是透明的，除了某些可能的越级操作外。已经分片过的数据报有可能会再次进行分片（可能不止一次）。IP首部中包含的数据为分片和重新组装提供了足够的信息。

回忆IP首部（图3-1），下面这些字段用于分片过程。对于发送端发送的每份IP数据报来说，其标识字段都包含一个唯一值。该值在数据报分片时被复制到每个片中（我们现在已经看到这个字段的用途）。标志字段用其中一个比特来表示“更多的片”。除了最后一片外，其他每个组成数据报的片都要把该比特置 1。片偏移字段指的是该片偏移原始数据报开始处的位置。另外，当数据报被分片后，每个片的总长度值要改为该片的长度值。

最后，标志字段中有一个比特称作“不分片”位。如果将这一比特置 1，IP将不对数据报进行分片。相反把数据报丢弃并发送一个 ICMP差错报文（“需要进行分片但设置了不分片比特”，见图6-3）给起始端。在下一节我们将看到出现这个差错的例子。

当IP数据报被分片后，每一片都成为一个分组，具有自己的 IP首部，并在选择路由时与其他分组独立。这样，当数据报的这些片到达目的端时有可能会失序，但是在 IP首部中有足够的信息让接收端能正确组装这些数据报片。

尽管IP分片过程看起来是透明的，但有一点让人不想使用它：即使只丢失一片数据也要重传整个数据报。为什么会发生这种情况呢？因为IP层本身没有超时重传的机制——由更高层来负责超时和重传（TCP有超时和重传机制，但UDP没有。一些UDP应用程序本身也执行超时和重传）。当来自TCP报文段的某一片丢失后，TCP在超时后会重发整个TCP报文段，该报文段对应于一份IP数据报。没有办法只重传数据报中的一个数据报片。事实上，如果对数据报分片的是中间路由器，而不是起始端系统，那么起始端系统就无法知道数据报是如何被分片的。就这个原因，经常要避免分片。文献[Kent and Mogul 1987]对避免分片进行了论述。

使用UDP很容易导致IP分片（在后面我们将看到，TCP试图避免分片，但对于应用程序来说几乎不可能强迫TCP发送一个需要进行分片的长报文段）。我们可以用sock程序来增加数据报的长度，直到分片发生。在一个以太网上，数据帧的最大长度是 1500字节（见图2-1），其中1472字节留给数据，假定IP首部为20字节，UDP首部为8字节。我们分别以数据长度为1471, 1472, 1473和1474字节运行sock程序。最后两次应该发生分片：

```
bsdi %sock -u -i -nl -w1471 svr4 discard
bsdi %sock -u -i -nl -w1472 svr4 discard
bsdi %sock -u -i -nl -w1473 svr4 discard
bsdi %sock -u -i -nl -w1474 svr4 discard
```

相应的tcpdump输出如图11-7所示。

```
1   0.0                 bsdi.1112 > svr4.discard: udp 1471
2   21.008303 (21.0083) bsdi.1114 > svr4.discard: udp 1472
3   50.449704 (29.4414) bsdi.1116 > svr4.discard: udp 1473 (frag 26304:1480@0+)
4   50.450040 ( 0.0003) bsdi > svr4: (frag 26304:1@1480)
5   75.328650 (24.8786) bsdi.1118 > svr4.discard: udp 1474 (frag 26313:1480@0+)
6   75.328982 ( 0.0003) bsdi > svr4: (frag 26313:2@1480)
```

图11-7 观察UDP数据报分片

前两份UDP数据报（第1行和第2行）能装入以太网数据帧，没有被分片。但是对应于写1473字节的IP数据报长度为1501，就必须进行分片（第3行和第4行）。同理，写1474字节产生的数据报长度为1502，它也需要进行分片（第5行和第6行）。

当IP数据报被分片后，`tcpdump`打印出其他的信息。首先，`frag 26304`（第3行和第4行）和`frag 26313`（第5行和第6行）指的是IP首部中标识字段的值。

分片信息中的下一个数字，即第3行中位于冒号和@号之间的1480，是除IP首部外的片长。两份数据报第一片的长度均为1480：UDP首部占8字节，用户数据占1472字节（加上IP首部的20字节分组长度正好为1500字节）。第1份数据报的第2片（第4行）只包含1字节数据——剩下的用户数据。第2份数据报的第2片（第6行）包含剩下的2字节用户数据。

在分片时，除最后一片外，其他每一片中的数据部分（除IP首部外的其余部分）必须是8字节的整数倍。在本例中，1480是8的整数倍。

位于@符号后的数字是从数据报开始处计算的片偏移值。两份数据报第1片的偏移值均为0(第3行和第5行)，第2片的偏移值为1480（第4行和第6行）。跟在偏移值后面的加号对应于IP首部中3 bit标志字段中的“更多片”比特。设置这一比特的目的是让接收端知道在什么时候完成所有的分片组装。

最后，注意第4行和第6行（不是第1片）省略了协议名（UDP）、源端口号和目的端口号。协议名是可以打印出来的，因为它在IP首部并被复制到各个片中。但是，端口号在UDP首部，只能在第1片中被发现。

发送的第3份数据报（用户数据为1473字节）分片情况如图11-8所示。需要重申的是，任何运输层首部只出现在第1片数据中。

另外需要解释几个术语：IP数据报是指IP层端到端的传输单元（在分片之前和重新组装之后），分组是指在IP层和链路层之间传送的数据单元。一个分组可以是一个完整的IP数据报，也可以是IP数据报的一个分片。

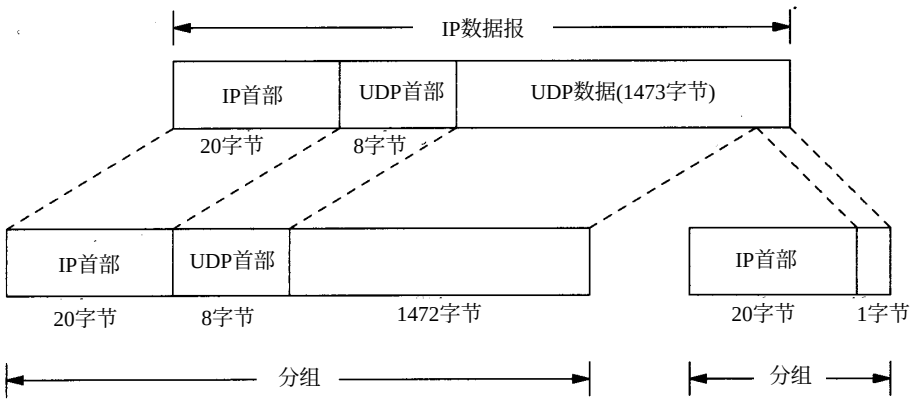


图11-8 UDP分片举例

11.6 ICMP不可达差错（需要分片）

发生ICMP不可达差错的另一种情况是，当路由器收到一份需要分片的数据报，而在IP首部又设置了不分片（DF）的标志比特。如果某个程序需要判断到达目的端的路途中最小MTU是多少——称作路径MTU发现机制（2.9节），那么这个差错就可以被该程序使用。

这种情况下的ICMP不可达差错报文格式如图11-9所示。这里的格式与图6-10不同，因为在第2个32 bit字中，16~31 bit可以提供下一站的MTU，而不再是0。

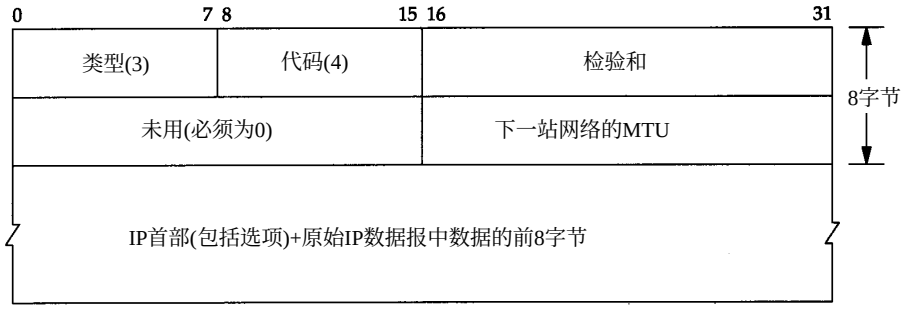


图11-9 需要分片但又设置不分片标志比特时的ICMP不可达差错报文格式

如果路由器没有提供这种新的ICMP差错报文格式，那么下一站的MTU就设为0。

新版的路由器需求RFC [Almquist 1993]声明，在发生这种ICMP不可达差错时，路由器必须生成这种新格式的报文。

例子

关于分片作者曾经遇到过一个问题，ICMP差错试图判断从路由器netb到主机sun之间的拨号SLIP链路的MTU。我们知道从sun到netb的链路的MTU：当SLIP被安装到主机sun时，这是SLIP配置过程中的一部分，加上在3.9节中已经通过netstat命令观察过。现在，我们想从另一个方向来判断它的MTU（在第25章，将讨论如何用SNMP来判断）。在点到点的链路中，不要求两个方向的MTU为相同值。

所采用的技术是在主机solaris上运行ping程序到主机bsdi，增加数据分组长度，直到看见进入的分组被分片为止。如图11-10所示。

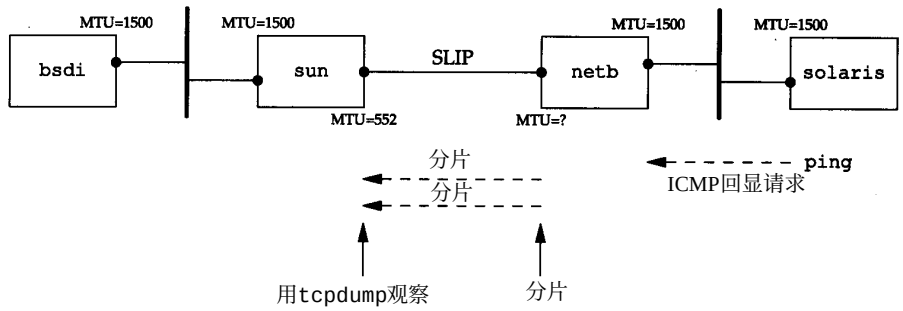


图11-10 用来判断从netb到sun的SLIP链路MTU的系统

在主机sun上运行tcpdump，观察SLIP链路，看什么时候发生分片。开始没有观察到分片，一切都很正常直到ping分组的数据长度从500增加到600字节。可以看到接收到的回显请

求（仍然没有分片），但不见回显应答。

为了跟踪下去，也在主机bsdi上运行tcpdump，观察它接收和发送的报文。输出如图11-11所示。

```
1  0.0                    solaris > bsdi: icmp: echo request (DF)
2  0.000000 (0.0000)      bsdi > solaris: icmp: echo reply (DF)
3  0.000000 (0.0000)      sun > bsdi: icmp: solaris unreachable -
                          need to frag, mtu = 0 (DF)
4  0.738400 (0.7384)      solaris > bsdi: icmp: echo request (DF)
5  0.748800 (0.0104)      bsdi > solaris: icmp: echo reply (DF)
6  0.748800 (0.0000)      sun > bsdi: icmp: solaris unreachable -
                          need to frag, mtu = 0 (DF)
```

图11-11 600字节的IP数据报从solaris 主机ping 到bsdi 主机时的tcpdump 输出

首先，每行中的标记（DF）说明在IP首部中设置了不分片比特。这意味着 Solaris 2.2 一般把不分片比特置1，作为实现路径MTU发现机制的一部分。

第1行显示的是回显请求通过路由器netb到达sun主机，没有进行分片，并设置了 DF比特，因此我们知道还没有达到netb的SLIP MTU。

接下来，在第2行注意到DF标志被复制到回显应答报文中。这就带来了问题。回显应答与回显请求报文长度相同（超过 600字节），但是sun外出的SLIP接口MTU为552。因此回显应答需要进行分片，但是DF标志比特又被设置了。这样， sun就产生一个ICMP不可达差错报文返回给bsdi（报文在bsdi处被丢弃）。

这就是我们在主机solaris上没有看到任何回显应答的原因。这些应答永远不能通过sun。分组的路径如图11-12所示。

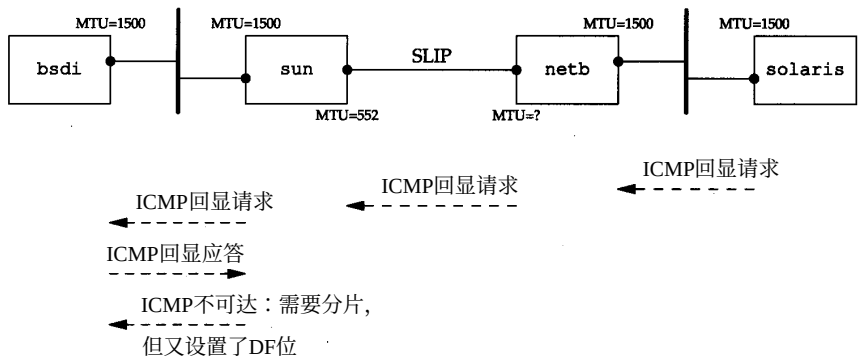


图11-12 例子中的分组交换

最后，在图11-11中的第3行和第6行中， mtu=0表示主机sun没有在ICMP不可达报文中返回出口MTU值，如图11-9所示（在25.9节中，将重新回到这个问题，用 SNMP判断netb上的SLIP接口MTU值为1500）。

11.7 用Traceroute确定路径MTU

尽管大多数的系统不支持路径 MTU发现功能，但可以很容易地修改 traceroute程序（第8章），用它来确定路径 MTU。要做的是发送分组，并设置“ 不分片 ”标志比特。发送的第一个分组的长度正好与出口 MTU相等，每次收到ICMP “ 不能分片 ”差错时（在上一节讨论

的）就减小分组的长度。如果路由器发送的ICMP差错报文是新格式，包含出口的MTU，那么就用该MTU值来发送，否则就用下一个最小的 MTU值来发送。正如RFC 1191 [Mogul and Deering 1990]声明的那样，MTU值的个数是有限的，因此在我们的程序中有一些由近似值构成的表，取下一个最小MTU值来发送。

首先，我们尝试判断从主机sun到主机slip的路径MTU，知道SLIP链路的MTU为296。

```
sun % traceroute.pmtu slip
traceroute to slip (140.252.13.65), 30 hops max
outgoing MTU = 1500
 1  bsdi (140.252.13.35)  15 ms  6 ms  6 ms
 2  bsdi (140.252.13.35)  6 ms
fragmentation required and DF set, trying new MTU = 1492
fragmentation required and DF set, trying new MTU = 1006
fragmentation required and DF set, trying new MTU = 576
fragmentation required and DF set, trying new MTU = 552
fragmentation required and DF set, trying new MTU = 544
fragmentation required and DF set, trying new MTU = 512
fragmentation required and DF set, trying new MTU = 508
fragmentation required and DF set, trying new MTU = 296
 2  slip (140.252.13.65)  377 ms  377 ms  377 ms
```

在这个例子中，路由器bsdi没有在ICMP差错报文中返回出口 MTU，因此我们选择另一个MTU近似值。TTL为2的第1行输出打印的主机名为bsdi，但这是因为它是返回ICMP差错报文的路由器。TTL为2的最后一行正是我们所要找的。

在bsdi上修改ICMP代码使它返回出口MTU值并不困难，如果那样做并再次运行该程序，得到如下输出结果：

```
sun % traceroute.pmtu slip
traceroute to slip (140.252.13.65), 30 hops max
outgoing MTU = 1500
 1  bsdi (140.252.13.35)  53 ms  6 ms  6 ms
 2  bsdi (140.252.13.35)  6 ms
fragmentation required and DF set, next hop MTU = 296
 2  slip (140.252.13.65)  377 ms  378 ms  377 ms
```

这时，在找到正确的MTU值之前，我们不用逐个尝试8个不同的MTU值——路由器返回了正确的MTU值。

全球互联网

作为一个实验，我们多次运行修改以后的traceroute程序，目的端为世界各地的主机。可以到达15个国家（包括南极洲），使用了多个跨大西洋和跨太平洋的链路。但是，在这样做之前，作者所在子网与路由器netb之间的拨号SLIP链路MTU（见图11-12）增加到1500，与以太网相同。

在18次运行当中，只有其中 2次发现的路径 MTU小于1500。其中一个跨大西洋的链路MTU值为572（其近似值甚至在 RFC 1191中也没有被列出），而路由器返回的是新格式的ICMP差错报文。另外一条链路，在日本的两个路由器之间，不能处理 1500字节的数据帧，并且路由器没有返回新格式的ICMP差错报文。把MTU值设成1006则可以正常工作。

从这个实验可以得出结论，现在许多但不是所有的广域网都可以处理大于 512字节的分组。利用路径MTU发现机制，应用程序就可以充分利用更大的 MTU来发送报文。

11.8 采用UDP的路径MTU发现

下面对使用UDP的应用程序与路径MTU发现机制之间的交互作用进行研究。看一看如果应用程序写了一个对于一些中间链路来说太长的数据报时会发生什么情况。

例子

由于我们所使用的支持路径MTU发现机制的唯一系统就是Solaris 2.x，因此，将采用它作为源站发送一份650字节数据报经slip。由于slip主机位于MTU为296的SLIP链路后，因此，任何长于268字节（296－20－8）且“不分片”比特置为1的UDP数据都会使bsdi路由器产生ICMP“不能分片”差错报文。图11-13给出了拓扑结构和MTU。

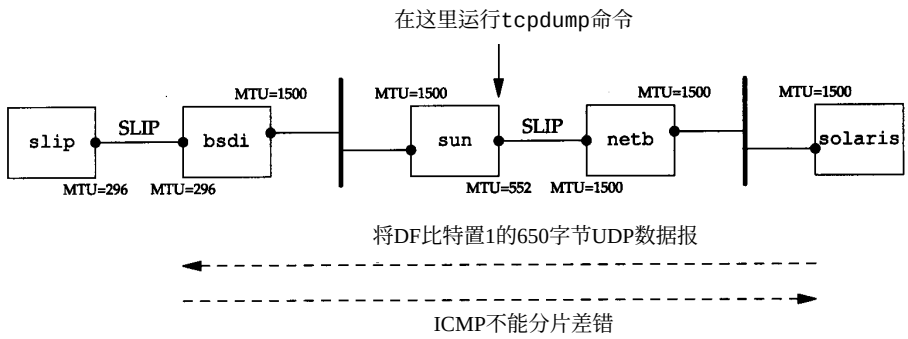


图11-13 使用UDP进行路径MTU发现的系统

可以用下面的命令行来产生650字节UDP数据报，每两个UDP数据报之间的间隔是5秒：

```
solaris %sock -u -i -n10 -w650 -p5 slip discard
```

图11-14是tcpdump的输出结果。在运行这个例子时，将bsdi设置成在ICMP“不能分片”差错中，不返回下一跳MTU信息。

在发送的第一个数据报中将DF比特置1（第1行），其结果是从bsdi路由器发回我们可以猜测的结果（第2行）。令人不解的是，发送一个DF比特置1的数据报（第3行），其结果是同样的ICMP差错（第4行）。我们预计这个数据报在发送时应该将DF比特置0。

第5行结果显示，IP已经知道了发往该目的地址的数据报不能将DF比特置1，因此，IP进而将数据报在源站主机上进行分片。这与前面的例子中，IP发送经过UDP的数据报，允许具有较小MTU的路由器（在本例中是bsdi）对它进行分片的情况不一样。由于ICMP“不能分片”报文并没有指出下一跳的MTU，因此，看来IP猜测MTU为576就行了。第一次分片（第5行）包含544字节的UDP数据、8字节UDP首部以及20字节IP首部，因此，总IP数据报长度是572字节。第2次分片（第6行）包含剩余的106字节UDP数据和20字节IP首部。

不幸的是，第7行的下一个数据报将其DF比特置1，因此bsdi将它丢弃并返回ICMP差错。这时发生了IP定时器超时，通知IP查看是不是因为路径MTU增大了而将DF比特再一次置1。我们可以从第19行和20行看出这个结果。将第7行与19行进行比较，可以看出IP每过30秒就将DF比特置1，以查看路径MTU是否增大了。

这个30秒的定时器值看来太短。RFC1191建议其值取10分钟。可以通过修改ip_ire_pathmtu_interval（E.4节）参数来改变该值。同时，Solaris 2.2无法对单个

UDP应用或所有UDP应用关闭该路径MTU发现。只能通过修改ip_path_mtu_discovery参数，在系统一级开放或关闭它。正如在这个例子里所能看到的那样，如果允许路径MTU发现，那么当UDP应用程序写入可能被分片数据报时，该数据报将被丢弃。

```
1   0.0                 solaris.38196 > slip.discard: udp 650 (DF)
2   0.004218 (0.0042)   bsdi > solaris: icmp:
                            slip unreachable - need to frag, mtu = 0 (DF)
3   4.980528 (4.9763)   solaris.38196 > slip.discard: udp 650 (DF)
4   4.984503 (0.0040)   bsdi > solaris: icmp:
                            slip unreachable - need to frag, mtu = 0 (DF)
5   9.870407 (4.8859)   solaris.38196 > slip.discard: udp 650 (frag 47942:552@0+)
6   9.960056 (0.0896)   solaris > slip: (frag 47942:106@552)
7   14.940338 (4.9803)  solaris.38196 > slip.discard: udp 650 (DF)
8   14.944466 (0.0041)  bsdi > solaris: icmp:
                            slip unreachable - need to frag, mtu = 0 (DF)
9   19.890015 (4.9455)  solaris.38196 > slip.discard: udp 650 (frag 47944:552@0+)
10  19.950463 (0.0604)  solaris > slip: (frag 47944:106@552)
11  24.870401 (4.9199)  solaris.38196 > slip.discard: udp 650 (frag 47945:552@0+)
12  24.960038 (0.0896)  solaris > slip: (frag 47945:106@552)
13  29.880182 (4.9201)  solaris.38196 > slip.discard: udp 650 (frag 47946:552@0+)
14  29.940498 (0.0603)  solaris > slip: (frag 47946:106@552)
15  34.860607 (4.9201)  solaris.38196 > slip.discard: udp 650 (frag 47947:552@0+)
16  34.950051 (0.0894)  solaris > slip: (frag 47947:106@552)
17  39.870216 (4.9202)  solaris.38196 > slip.discard: udp 650 (frag 47948:552@0+)
18  39.930443 (0.0602)  solaris > slip: (frag 47948:106@552)
19  44.940485 (5.0100)  solaris.38196 > slip.discard: udp 650 (DF)
20  44.944432 (0.0039)  bsdi > solaris: icmp:
                            slip unreachable - need to frag, mtu = 0 (DF)
```

图11-14 使用UDP路径MTU发现

solaris的IP层所假设的最大数据报长度（576字节）是不正确的。在图11-13中，我们看到，实际的MTU值是296字节。这意味着经solaris分片的数据报还将被bsdi分片。图11-15给出了在目的主机（slip）上所收集到的tcpdump对于第一个到达数据报的输出结果（图11-14的第5行和第6行）。

```
1  0.0                 solaris.38196 > slip.discard: udp 650 (frag 47942:272@0+)
2  0.304513 (0.3045)   solaris > slip: (frag 47942:272@272+)
3  0.334651 (0.0301)   solaris > slip: (frag 47942:8@544+)
4  0.466642 (0.1320)   solaris > slip: (frag 47942:106@552)
```

图11-15 从solaris到达slip的第一个数据报

在本例中，solaris不应该对外出数据报分片，它应该将DF比特置0，让具有最小MTU的路由器来完成分片工作。

现在我们运行同一个例子，只是对路由器bsdi进行修改使其在ICMP“不能分片”差错中返回下一跳MTU。图11-16给出了tcpdump输出结果的前6行。

与图11-14一样，前两个数据报同样是将DF比特置1后发送出去的。但是在知道了下一跳MTU后，只产生了3个数据报片，而图11-15中的bsdi路由器则产生了4个数据报片。

```
1  0.0                  solaris.37974 > slip.discard: udp 650 (DF)
2  0.004199 (0.0042)    bsdi > solaris: icmp:
                              slip unreachable - need to frag, mtu = 296 (DF)
3  4.950193 (4.9460)    solaris.37974 > slip.discard: udp 650 (DF)
4  4.954325 (0.0041)    bsdi > solaris: icmp:
                              slip unreachable - need to frag, mtu = 296 (DF)
5  9.779855 (4.8255)    solaris.37974 > slip.discard: udp 650 (frag 35278:272@0+)
6  9.930018 (0.1502)    solaris > slip: (frag 35278:272@272+)
7  9.990170 (0.0602)    solaris > slip: (frag 35278:114@544)
```

图11-16 使用UDP的路径MTU发现

11.9 UDP和ARP之间的交互作用

使用UDP，可以看到UDP与ARP典型实现之间的有趣的（而常常未被人提及）交互作用。

我们用sock程序来产生一个包含8192字节数据的UDP数据报。预测这将会在以太网上产生6个数据报片（见习题11.3）。同时也确保在运行该程序前，ARP缓存是清空的，这样，在发送第一个数据报片前必须交换ARP请求和应答。

```
bsdi %arp -a                                    验证ARP高速缓存是空的
bsdi %sock -u -i -nl -w8192 svr4 discard
```

预计在发送第一个数据报片前会先发送一个ARP请求。IP还会产生5个数据报片，这样就提出了我们必须用tcpdump来回答的两个问题：在接收到ARP回答前，其余数据报片是否已经做好了发送准备？如果是这样，那么在ARP等待应答时，它会如何处理发往给定目的的多个报文？图11-17给出了tcpdump的输出结果。

```
1   0.0                    arp who-has svr4 tell bsdi
2   0.001234 (0.0012)      arp who-has svr4 tell bsdi
3   0.001941 (0.0007)      arp who-has svr4 tell bsdi
4   0.002775 (0.0008)      arp who-has svr4 tell bsdi
5   0.003495 (0.0007)      arp who-has svr4 tell bsdi
6   0.004319 (0.0008)      arp who-has svr4 tell bsdi
7   0.008772 (0.0045)      arp reply svr4 is-at 0:0:c0:c2:9b:26
8   0.009911 (0.0011)      arp reply svr4 is-at 0:0:c0:c2:9b:26
9   0.011127 (0.0012)      bsdi > svr4: (frag 10863:800@7400)
10  0.011255 (0.0001)      arp reply svr4 is-at 0:0:c0:c2:9b:26
11  0.012562 (0.0013)      arp reply svr4 is-at 0:0:c0:c2:9b:26
12  0.013458 (0.0009)      arp reply svr4 is-at 0:0:c0:c2:9b:26
13  0.014526 (0.0011)      arp reply svr4 is-at 0:0:c0:c2:9b:26
14  0.015583 (0.0011)      arp reply svr4 is-at 0:0:c0:c2:9b:26
```

图11-17 在以太网上发送8192字节UDP数据报时的报文交换

在这个输出结果中有一些令人吃惊的结果。首先，在第一个ARP应答返回以前，总共产生了6个ARP请求。我们认为其原因是IP很快地产生了6个数据报片，而每个数据报片都引发了一个ARP请求。

第二，在接收到第一个ARP应答时（第7行），只发送最后一个数据报片（第9行）！看来似乎将前5个数据报片全都丢弃了。实际上，这是ARP的正常操作。在大多数的实现中，在等待一个ARP应答时，只将最后一个报文发送给特定目的主机。

Host Requirements RFC要求实现中必须防止这种类型的ARP洪泛（ARP flooding，

即以高速率重复发送到同一个IP地址的ARP请求)。建议最高速率是每秒一次。而这里却在4.3 ms内发出了6个ARP请求。

Host Requirements RFC规定，ARP应该保留至少一个报文，而这个报文必须是最后一个报文。这正是我们在这里所看到的结果。

另一个无法解释的不正常的现象是， svr4发回7个，而不是6个ARP应答。

最后要指出的是，在最后一个 ARP应答返回后，继续运行 tcpdump程序5分钟，以看看svr4是否会返回ICMP“组装超时”差错。并没有发送 ICMP差错（我们在图8-2中给出了该消息的格式。code字段为1表示在重新组装数据报时发生了超时)。

在第一个数据报片出现时， IP层必须启动一个定时器。这里“第一个”表示给定数据报的第一个到达数据报片，而不是第一个数据报片（数据报片偏移为 0)。正常的定时器值为30或60秒。如果定时器超时而该数据报的所有数据报片未能全部到达，那么将这些数据报片丢弃。如果不这么做，那些永远不会到达的数据报片（正如我们在本例中所看到的那样）迟早会引起接收端缓存满。

这里我们没看到ICMP消息的原因有两个。首先，大多数从 Berkeley派生的实现从不产生该差错！这些实现会设置定时器，也会在定时器溢出时将数据报片丢弃，但是不生成 ICMP差错。第二，并未接收到包含 UDP首部的偏移量为0的第一个数据报片（这是被 ARP所丢弃的5个报文的第1个)。除非接收到第一个数据报片，否则并不要求任何实现产生 ICMP差错。其原因是因为没有运输层首部，ICMP差错的接收者无法区分出是哪个进程所发送的数据报被丢弃。这里假设上层（TCP或使用UDP的应用程序）最终会超时并重传。

在本节中，我们使用 IP数据报片来查看 UDP与ARP之间的交互作用。如果发送端迅速发送多个UDP数据报，也可以看到这个交互过程。我们选择采用分片的方法，是因为 IP可以生成报文的速度，比一个用户进程生成多个数据报的速度更快。

尽管本例看来不太可能，但它确实经常发生。 NFS发送的UDP数据报长度超过8192字节。在以太网上，这些数据报以我们所指出的方式进行分片，如果适当的 ARP缓存入口发生超时，那么就可以看到这里所显示的现象。 NFS将超时并重传，但是由于ARP的有限队列，第一个IP数据报仍可能被丢弃。

11.10 最大UDP数据报长度

理论上，IP数据报的最大长度是65535字节，这是由IP首部（图3-1）16比特总长度字段所限制的。去除20字节的IP首部和8个字节的UDP首部， UDP数据报中用户数据的最长长度为65507字节。但是，大多数实现所提供的长度比这个最大值小。

我们将遇到两个限制因素。第一，应用程序可能会受到其程序接口的限制。 socket API提供了一个可供应用程序调用的函数，以设置接收和发送缓存的长度。对于 UDP socket，这个长度与应用程序可以读写的最大 UDP数据报的长度直接相关。现在的大部分系统都默认提供了可读写大于8192字节的UDP数据报（使用这个默认值是因为 8192是NFS读写用户数据数的默认值)。

第二个限制来自于TCP/IP的内核实现。可能存在一些实现特性（或差错），使IP数据报长度小于65535字节。

作者使用sock程序对不同UDP数据报长度进行了试验。在SunOS 4.1.3下使用环回

接口的最大IP数据报长度是32767字节。比它大的值都会发生差错。但是从BSD/386到SunOS 4.1.3的情况下，Sun所能接收到最大IP数据报长度为32786字节（即32758字节用户数据）。在Solaris 2.2下使用环回接口，最大可收发IP数据报长度为65535字节。从Solaris 2.2到AIX 3.2.2，发送的最大IP数据报长度可以是65535字节。很显然，这个限制与源端和目的端的实现有关。

我们在3.2节中提过，要求主机必须能够接收最短为576字节的IP数据报。在许多UDP应用程序的设计中，其应用程序数据被限制成512字节或更小，因此比这个限制值小。例如，我们在10.4节中看到，路径信息协议总是发送每份数据报小于 512字节的数据。我们还会在其他UDP应用程序如DNS（第14章）、TFTP（第15章）、BOOTP（第16章）以及SNMP（第25章）中遇到这个限制。

数据报截断

由于IP能够发送或接收特定长度的数据报并不意味着接收应用程序可以读取该长度的数据。因此，UDP编程接口允许应用程序指定每次返回的最大字节数。如果接收到的数据报长度大于应用程序所能处理的长度，那么会发生什么情况呢？

不幸的是，该问题的答案取决于编程接口和实现。

典型的Berkeley版socket API对数据报进行截断，并丢弃任何多余的数据。应用程序何时能够知道，则与版本有关（4.3BSD Reno及其后的版本可以通知应用程序数据报被截断）。

SVR4下的socket API(包括Solaris 2.x) 并不截断数据报。超出部分数据在后面的读取中返回。它也不通知应用程序从单个UDP数据报中多次进行读取操作。

TLI API不丢弃数据。相反，它返回一个标志表明可以获得更多的数据，而应用程序后面的读操作将返回数据报的其余部分。

在讨论TCP时，我们发现它为应用程序提供连续的字节流，而没有任何信息边界。 TCP以应用程序读操作时所要求的长度来传送数据，因此，在这个接口下，不会发生数据丢失。

11.11 ICMP源站抑制差错

我们同样也可以使用UDP产生ICMP“源站抑制(source quench)”差错。当一个系统（路由器或主机）接收数据报的速度比其处理速度快时，可能产生这个差错。注意限定词“可能”。即使一个系统已经没有缓存并丢弃数据报，也不要求它一定要发送源站抑制报文。

图11-18给出了ICMP源站抑制差错报文的格式。有一个很好的方案可以在我们的测试网络里产生该差错报文。可以从bsdi通过必须经过拨号SLIP链路的以太网，将数据报发送给路由器sun。由于SLIP链路的速度大约只有以太网的千分之一，因此，我们很容易就可以使其缓存用完。下面的命令行从主机bsdi通过路由器sun发送100个1024字节长数据报给solaris。我们将数据报发送给标准的丢弃服务，这样，这些数据报将被忽略：

```
bsdi % sock -u -i -w1024 -n100 solaris discard
```

图11-19给出了与此命令行相对应的tcpdump输出结果

在这个输出结果中，删除了很多行，这只是一个模型。接收前 26个数据报时未发生差

错；我们只给出了第一个数据报的结果。然而，从第 27个数据报开始，每发送一份数据报，就会接收到一份源站抑制差错报文。总共有 26 + （74×2）=174行输出结果。

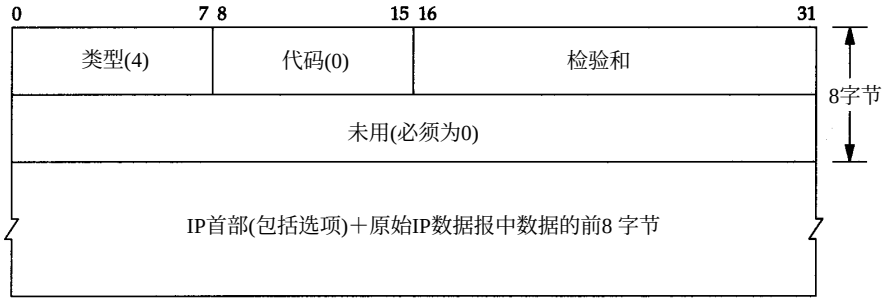


图11-18 ICMP源站抑制差错报文格式

```
 1  0.0                    bsdi.1403 > solaris.discard: udp 1024
                                  26 lines that we don't show
 27 0.10 (0.00)            bsdi.1403 > solaris.discard: udp 1024
 28 0.11 (0.01)            sun > bsdi: icmp: source quench

 29 0.11 (0.00)            bsdi.1403 > solaris.discard: udp 1024
 30 0.11 (0.00)            sun > bsdi: icmp: source quench
                                  142 lines that we don't show
173 0.71 (0.06)            bsdi.1403 > solaris.discard: udp 1024
174 0.71 (0.00)            sun > bsdi: icmp: source quench
```

图11-19 来自路由器sun 的ICMP源站抑制

从2.10节的并行线吞吐率计算结果可以知道，以 9600 b/s速率传送1024字节数据报只需要1秒时间（由于从sun到netb的SLIP链路的MTU为552字节，因此在我们的例子中，20 + 8 + 1024字节数据报将进行分片，因此，其时间会稍长一些）。但是我们可以从图 11-19的时间中看出，sun路由器在不到1秒时间内就处理完所有的 100个数据报，而这时，第一份数据报还未通过SLIP链路。因此我们用完其缓存就不足不奇了。

尽管RFC 1009 [Braden and Postel 1987] 要求路由器在没有缓存时产生源站抑制差错报文，但是新的Router Requirements RFC [Almquist 1993] 对此作了修改，提出路由器不应该产生源站抑制差错报文。由于源站抑制要消耗网络带宽，且对于拥塞来说是一种无效而不公平的调整，因此现在人们对于源站抑制差错的态度是不支持的。

在本例中，还需要指出的是，sock程序要么没有接收到源站抑制差错报文，要么接收到却将它们忽略了。结果是如果采用 UDP协议，那么BSD实现通常忽略其接收到的源站抑制报文（正如我们在21.10节所讨论的那样，TCP接受源站抑制差错报文，并将放慢在该连接上的数据传输速度）。其部分原因在于，在接收到源站抑制差错报文时，导致源站抑制的进程可能已经中止了。实际上，如果使用 Unix 的time程序来测定sock程序所运行的时间，其结果是它只运行了大约0.5秒时间。但是从图 11-19中可以看到，在发送第一份数据报过后 0.71秒才接收到一些源站抑制，而此时该进程已经中止。其原因是我们的程序写入了 100个数据报然后中止了。但是所有的100个数据报都已发送出去 —— 有一些数据报在输出队列中。

这个例子重申了UDP是一个非可靠的协议，它说明了端到端的流量控制。尽管 sock程序成功地将100个数据报写入其网络，但只有 26个数据报真正发送到了目的端。其他 74个数据报

可能被中间路由器丢弃。除非在应用程序中建立一些应答机制，否则发送端并不知道接收端是否收到了这些数据。

11.12 UDP服务器的设计

使用UDP的一些蕴含对于设计和实现服务器会产生影响。通常，客户端的设计和实现比服务器端的要容易一些，这就是我们为什么要讨论服务器的设计，而不是讨论客户端的设计的原因。典型的服务器与操作系统进行交互作用，而且大多数需要同时处理多个客户。

通常一个客户启动后直接与单个服务器通信，然后就结束了。而对于服务器来说，它启动后处于休眠状态，等待客户请求的到来。对于 UDP来说，当客户数据报到达时，服务器苏醒过来，数据报中可能包含来自客户的某种形式的请求消息。

在这里我们所感兴趣的并不是客户和服务器的编程方面（ [Stevens 1990]对这些方面的细节进行了讨论），而是UDP那些影响使用该协议的服务器的设计和实现方面的协议特性（我们在 18.11节中对TCP服务器的设计进行了描述）。尽管我们所描述的一些特性取决于所使用UDP的实现，但对于大多数实现来说，这些特性是公共的。

11.12.1 客户IP地址及端口号

来自客户的是UDP数据报。IP首部包含源端和目的端 IP地址，UDP首部包含了源端和目的端的UDP端口号。当一个应用程序接收到 UDP数据报时，操作系统必须告诉它是谁发送了这份消息，即源IP地址和端口号。

这个特性允许一个交互 UDP服务器对多个客户进行处理。给每个发送请求的客户发回应答。

11.12.2 目的IP地址

一些应用程序需要知道数据报是发送给谁的，即目的 IP地址。例如，Host Requirements RFC规定，TFTP服务器必须忽略接收到的发往广播地址的数据报（我们分别在第 12章和第15章对广播和TFTP进行描述）。

这要求操作系统从接收到的 UDP数据报中将目的IP地址交给应用程序。不幸的是，并非所有的实现都提供这个功能。

> socket API以IP_RECVDSTADDR socket选项提供了这个功能。对于本文中使用的系统，只有BSD/386、4.4BSD和AIX 3.2.2支持该选项。SVR4、SunOS 4.x和Solaris 2.x都不支持该选项。

11.12.3 UDP输入队列

我们在1.8节中说过，大多数UDP服务器是交互服务器。这意味着，单个服务器进程对单个UDP端口上（服务器上的名知端口）的所有客户请求进行处理。

通常程序所使用的每个 UDP端口都与一个有限大小的输入队列相联系。这意味着，来自不同客户的差不多同时到达的请求将由 UDP自动排队。接收到的UDP数据报以其接收顺序交给应用程序（在应用程序要求交送下一个数据报时）。

然而，排队溢出造成内核中的UDP模块丢弃数据报的可能性是存在的。可以进行以下试验。我们在作为UDP服务器的bsdi主机上运行sock程序：

```
bsdi %sock -s -u -v -E -R256 -P30 6666
from 140.252.13.33, to 140.252.13.63: 1111111111从sun发送到广播地址
from 140.252.13.34, to 140.252.13.35: 4444444444从svr4发送到单播地址
```

我们指明以下标志：-s表示作为服务器运行，-u表示UDP，-v表示打印客户的IP地址，-E表示打印目的IP地址（该系统支持这个功能）。另外，我们将这个端口的UDP接收缓存设置为256字节（-R），其每次应用程序读取的大小也是这个数（-r）。标志-P30表示创建UDP端口后，先暂停30秒后再读取第一个数据报。这样，我们就有时间在另两台主机上启动客户程序，发送一些数据报，以查看接收队列是如何工作的。

服务器一开始工作，处于其30秒的暂停时间内，我们就在sun主机上启动一个客户，并发送三个数据报：

```
sun % sock -u -v 140.252.13.63 6666                到以太网广播地址
connected on 140.252.13.33.1252 to 140.252.13.63.6666
1111111111                                         11字节的数据（新行）
222222222                                          10字节的数据（新行）
33333333333                                        12字节的数据（新行）
```

目的地址是广播地址（140.252.13.63）。我们同时也在主机svr4上启动第2个客户，并发送另外三个数据报：

```
svr4 %sock -u -v bsdi 6666
connected on 0.0.0.0.1042 to 140.252.13.35.6666
4444444444444                                      14字节的数据（新行）
555555555555555                                    16字节的数据（新行）
66666666                                           9字节的数据（新行）
```

首先，我们早些时候在bsdi上所看到的结果表明，应用程序只接收到2个数据报：来自sun的第一个全1报文，和来自svr4的第一个全4报文。其他4个数据报看来全被丢弃。

图11-20给出的tcpdump输出结果表明，所有6个数据报都发送给了目的主机。两个客户的数据报以交替顺序键入：第一个来自sun，然后是来自svr4的，以此类推。同时也可以看出，全部6个数据报大约在12秒内发送完毕，也就是在服务器休眠的30秒内完成的。

```
1   0.0                 sun.1252 > 140.252.13.63.6666: udp 11
2   2.499184 (2.4992)   svr4.1042 > bsdi.6666: udp 14
3   4.959166 (2.4600)   sun.1252 > 140.252.13.63.6666: udp 10
4   7.607149 (2.6480)   svr4.1042 > bsdi.6666: udp 16
5  10.079059 (2.4719)   sun.1252 > 140.252.13.63.6666: udp 12
6  12.415943 (2.3369)   svr4.1042 > bsdi.6666: udp 9
```

图11-20 两个客户发送UDP数据报的tcpdump 输出结果

我们还可以看到，服务器的-E选项使其可以知道每个数据报的目的IP地址。如果需要，它可以选择如何处理其接收到的第一个数据报，这个数据报的地址是广播地址。

我们可以从本例中看到以下几个要点。首先，应用程序并不知道其输入队列何时溢出。只是由UDP对超出数据报进行丢弃处理。同时，从tcpdump输出结果，我们看到，没有发回任何信息告诉客户其数据报被丢弃。这里不存在像ICMP源站抑制这样发回发送端的消息。最后，看来UDP输出队列是FIFO（先进先出）的，而我们在11.9节中所看到的ARP输入却是

LIFO（后进先出）的。

11.12.4 限制本地IP地址

大多数UDP服务器在创建UDP端点时都使其本地IP地址具有通配符(wildcard)的特点。这就表明进入的UDP数据报如果其目的地为服务器端口，那么在任何本地接口均可接收到它。例如，我们以端口号777启动一个UDP服务器：

```
sun % sock -u -s 7777
```

然后，用netstat命令观察端点的状态：

```
sun % netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address      Foreign Address      (state)
udp        0      0    *.7777               *.*
```

这里，我们删除了许多行，只保留了其中感兴趣的东西。 -a选项表示报告所有网络端点的状态。 -n选项表示以点数格式打印IP地址而不用DNS把地址转换成名字，打印数字端口号而不是服务名称。 -f inet选项表示只报告TCP和UDP端点。

本地地址以*.7777格式打印，星号表示任何本地IP地址。

当服务器创建端点时，它可以把其中一个主机本地 IP地址包括广播地址指定为端点的本地IP地址。只有当目的IP地址与指定的地址相匹配时，进入的 UDP数据报才能被送到这个端点。用我们的sock程序，如果在端口号之前指定一个 IP地址，那么该IP地址就成为该端点的本地IP地址。例如：

```
sun % sock -u -s 140.252.1.29 7777
```

就限制服务器在SLIP接口(140.252.1.29)处接收数据报。netstat输出结果显示如下：

```
Proto Recv-Q Send-Q    Local Address         Foreign Address      (state)
udp        0      0    140.252.1.29.7777          *.*
```

如果我们试图在以太网上的主机 bsdi以地址140.252.13.35向该服务器发送一份数据报，那么将返回一个ICMP端口不可达差错。服务器永远看不到这份数据报。这种情形如图 11-21所示。

```
1  0.0                  bsdi.1723 > sun.7777: udp 13
2  0.000822 (0.0008)    sun > bsdi: icmp: sun udp port 7777 unreachable
```

图11-21 服务器本地地址绑定导致拒绝接收UDP数据报

有可能在相同的端口上启动不同的服务器，每个服务器具有不同的本地 IP地址。但是，一般必须告诉系统应用程序重用相同的端口号没有问题。

使用sockets API时，必须指定SO_REUSEADDR socket选项。在sock程序中是通过-A选项来完成的。

在主机sun上，可以在同一个端口号（8888）上启动5个不同的服务器：

```
sun % sock -u -s 140.252.1.29 8888          对于SLIP链路
sun % sock -u -s -A 140.252.13.33 8888      对于以太网
sun % sock -u -s -A 127.0.0.1 8888          对于环回接口
sun % sock -u -s -A 140.252.13.63 8888      对于以太网广播
sun % sock -u -s -A 8888                    其他(IP地址通配)
```

除了第一个以外，其他的服务器都必须以-A选项启动，告诉系统可以重用同一个端口号。5个服务器的netstat输出结果如下所示：

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
udp        0      0  *.8888                 *.*
udp        0      0  140.252.13.63.8888     *.*
udp        0      0  127.0.0.1.8888         *.*
udp        0      0  140.252.13.33.8888     *.*
udp        0      0  140.252.1.29.8888      *.*
```

在这种情况下，到达服务器的数据报中，只有带星号的本地IP地址，其目的地址为140.252.1.255，因为其他4个服务器占用了其他所有可能的IP地址。

如果存在一个含星号的IP地址，那么就隐含了一种优先级关系。如果为端点指定了特定IP地址，那么在匹配目的地址时始终优先匹配该IP地址。只有在匹配不成功时才使用含星号的端点。

11.12.5 限制远端IP地址

在前面所有的netstat输出结果中，远端IP地址和远端端口号都显示为*.*，其意思是该端点将接受来自任何IP地址和任何端口号的UDP数据报。大多数系统允许UDP端点对远端地址进行限制。

这说明端点将只能接收特定IP地址和端口号的UDP数据报。sock程序用-f选项来指定远端IP地址和端口号：

```
sun % sock -u -s -f 140.252.13.35.4444 5555
```

这样就设置了远端IP地址140.252.13.35（即主机bsdi）和远端端口号4444。服务器的有名端口号为5555。如果运行netstat命令，我们发现本地IP地址也被设置了，尽管我们没有指定。

```
Proto Recv-Q Send-Q    Local Address          Foreign Address      (state)
udp        0      0   140.252.13.33.5555     140.252.13.35.4444
```

这是在伯克利派生系统中指定远端IP地址和端口号带来的副作用：如果在指定远端地址时没有选择本地地址，那么将自动选择本地地址。它的值就成为选择到达远端IP地址路由时将选择的接口IP地址。事实上，在这个例子中，sun在以太网上的IP地址与远端地址140.252.13.33相连。

图11-22总结了UDP服务器本身可以创建的三类地址绑定。

本地地址	远端地址	描　述
localIP.lport	foreignIP.fport	只限于一个客户
localIP.lport	*.*	限于到达一个本地接口的数据报：localIP
*.lport	*.*	接收发送到lport的所有数据报

图11-22 为UDP服务器指定本地和远端IP地址及端口号

在所有情况下，lport指的是服务器有名端口号，localIP必须是本地接口的IP地址。表中这三行的排序是UDP模块在判断用哪个端点接收数据报时所采用的顺序。最为确定的地址（第一行）首先被匹配，最不确定的地址（最后一行IP地址带有两个星号）最后进行匹配。

11.12.6 每个端口有多个接收者

尽管在RFC中没有指明，但大多数的系统在某一时刻只允许一个程序端点与某个本地IP

地址及UDP端口号相关联。当目的地为该IP地址及端口号的UDP数据报到达主机时，就复制一份传给该端点。端点的IP地址可以含星号，正如我们前面讨论的那样。

例如，在SunOS 4.1.3中，我们启动一个端口号为9999的服务器，本地IP地址含有星号：

```
sun % sock -u -s 9999
```

接着，如果启动另一个具有相同本地地址和端口号的服务器，那么它将不运行，尽管我们指定了-A选项：

```
sun % sock -u -s 9999        我们预计它会失败
can't bind local address: Address already in use
sun % sock -u -s -A 9999     因此，这次尝试-A参数
can't bind local address: Address already in use
```

在一个支持多播的系统上（第12章），这种情况将发生变化。多个端点可以使用同一个IP地址和UDP端口号，尽管应用程序通常必须告诉API是可行的（如，用-A标志来指明SO_REUSEADDR socket选项）。

> 4.4BSD支持多播传送，需要应用程序设置一个不同的socket选项（SO_REUSEPORT）以允许多个端点共享同一个端口。另外，每个端点必须指定这个选项，包括使用该端口的第一个端点。

当UDP数据报到达的目的IP地址为广播地址或多播地址，而且在目的IP地址和端口号处有多个端点时，就向每个端点传送一份数据报的复制（端点的本地IP地址可以含有星号，它可匹配任何目的IP地址）。但是，如果UDP数据报到达的是一个单播地址，那么只向其中一个端点传送一份数据报的复制。选择哪个端点传送数据取决于各个不同的系统实现。

11.13 小结

UDP是一个简单协议。它的正式规范是RFC 768 [Postel 1980]，只包含三页内容。它向用户进程提供的服务位于IP层之上，包括端口号和可选的检验和。我们用UDP来检查检验和，并观察分片是如何进行的。

接着，我们讨论了ICMP不可达差错，它是新的路径MTU发现功能中的一部分（2.9节）。用Traceroute和UDP来观察路径MTU发现过程。还查看了UDP和ARP之间的接口，大多数的ARP实现在等待ARP应答时只保留最近传送给目的端的数据报。

当系统接收IP数据报的速率超过这些数据报被处理的速率时，系统可能发送ICMP源站抑制差错报文。使用UDP时很容易产生这样的ICMP差错。

习题

11.1 在11.5节中，向UDP数据报中写入1473字节用户数据时导致以太网数据报片的发生。在采用以太网IEEE 802封装格式时，导致分片的最小用户数据长度为多少？

11.2 阅读RFC 791[Postel 1981a]，理解为什么除最后一片外，其他片中的数据长度均要求为8字节的整数倍？

11.3 假定有一个以太网和一份8192字节的UDP数据报，那么需要分成多少个数据报片，每个数据报片的偏移和长度为多少？

11.4 继续前一习题，假定这些数据报片要经过一条MTU为552的SLIP链路。必须记住每一个

数据报片中的数据（除IP首部外）为8字节的整数倍。那么又将分成多少个数据报片？每个数据报片的偏移和长度为多少？

11.5 一个用UDP发送数据报的应用程序，它把数据报分成 4个数据报片。假定第 1片和第2片到达目的端，而第3片和第4片丢失了。应用程序在10秒钟后超时重发该UDP数据报，并且被分成相同的4片（相同的偏移和长度）。假定这一次接收主机重新组装的时间为60秒，那么当重发的第3片和第4片到达目的端时，原先收到的第 1片和第2片还没有被丢弃。接收端能否把这4片数据重新组装成一份IP数据报？

11.6 你是如何知道图11-15中的片实际上与图11-14中第5行和第6行相对应？

11.7 主机gemini开机33天后，netstat程序显示48 000 000份IP数据报中由于首部检验和差错被丢弃129份，在30 000 000个TCP段中由于TCP检验和差错而被丢弃 20个。但是，在大约18 000 000份UDP数据报中，因为UDP检验和差错而被丢弃的数据报一份也没有。请说明两个方面的原因（提示：参见图 11-4）。

11.8 在讨论分片时没有提及任何关于 IP首部中的选项——它们是否也要被复制到每个数据报片中，或者只留在第一个数据报片中？我们已经讨论过下面这些 IP选项：记录路由（7.3节）、时间戳（7.4节）、严格和宽松的源站选路（ 8.5节）。你希望分片如何处理这些选项？对照RFC 791检查你的答案。

11.9 在图1-8中，我们说UDP数据报是根据目的UDP端口号进行分配的。这正确吗？

第12章　广播和多播

12.1　引言

在第1章中我们提到有三种IP地址：单播地址、广播地址和多播地址。本章将更详细地介绍广播和多播。

广播和多播仅应用于UDP，它们对需将报文同时传往多个接收者的应用来说十分重要。TCP是一个面向连接的协议，它意味着分别运行于两主机（由IP地址确定）内的两进程（由端口号确定）间存在一条连接。

考虑包含多个主机的共享信道网络如以太网。每个以太网帧包含源主机和目的主机的以太网地址（48bit）。通常每个以太网帧仅发往单个目的主机，目的地址指明单个接收接口，因而称为单播(unicast)。在这种方式下，任意两个主机的通信不会干扰网内其他主机（可能引起争夺共享信道的情况除外）。

然而，有时一个主机要向网上的所有其他主机发送帧，这就是广播。通过ARP和RARP可以看到这一过程。多播(multicast)处于单播和广播之间：帧仅传送给属于多播组的多个主机。

为了弄清广播和多播，需要了解主机对由信道传送过来帧的过滤过程。图12-1说明了这一过程。

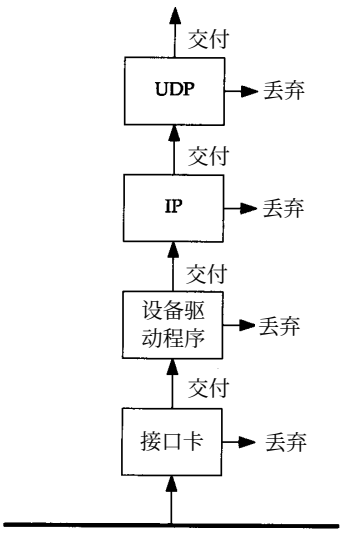


图12-1　协议栈各层对收到帧的过滤过程

首先，网卡查看由信道传送过来的帧，确定是否接收该帧，若接收后就将它传往设备驱动程序。通常网卡仅接收那些目的地址为网卡物理地址或广播地址的帧。另外，多数接口均被设置为混合模式，这种模式能接收每个帧的一个复制。作为一个例子，`tcpdump`使用这种模式。

目前，大多数的网卡经过配置都能接收目的地址为多播地址或某些子网多播地址的帧。对于以太网，当地址中最高字节的最低位设置为1时表示该地址是一个多播地址，用十六进制可表示为01:00:00:00:00:00（以太网广播地址ff:ff:ff:ff:ff:ff可看作是以太网多播地址的特例）。

如果网卡收到一个帧，这个帧将被传送给设备驱动程序（如果帧检验和错，网卡将丢弃该帧）。设备驱动程序将进行另外的帧过滤。首先，帧类型中必须指定要使用的协议（IP、ARP等等）。其次，进行多播过滤来检测该主机是否属于多播地址说明的多播组。

设备驱动程序随后将数据帧传送给下一层，比如，当帧类型指定为IP数据报时，就传往IP层。IP根据IP地址中的源地址和目的地址进行更多的过滤检测。如果正常，就将数据报传送给下一层（如TCP或UDP）。

每次UDP收到由IP传送来的数据报，就根据目的端口号，有时还有源端口号进行数据报

过滤。如果当前没有进程使用该目的端口号，就丢弃该数据报并产生一个 ICMP不可达报文（TCP根据它的端口号作相似的过滤）。如果UDP数据报存在检验和错，将被丢弃。

使用广播的问题在于它增加了对广播数据不感兴趣主机的处理负荷。拿一个使用 UDP广播应用作为例子。如果网内有 50个主机，但仅有 20个参与该应用，每次这 20个主机中的一个发送UDP广播数据时，其余 30个主机不得不处理这些广播数据报。一直到 UDP层，收到的UDP广播数据报才会被丢弃。这 30个主机丢弃UDP广播数据报是因为这些主机没有使用这个目的端口。

多播的出现减少了对应用不感兴趣主机的处理负荷。使用多播，主机可加入一个或多个多播组。这样，网卡将获悉该主机属于哪个多播组，然后仅接收主机所在多播组的那些多播帧。

12.2 广播

在图3-9中，我们知道了四种IP广播地址，下面对它们进行更详细的介绍。

12.2.1 受限的广播

受限的广播地址是 255.255.255.255。该地址用于主机配置过程中 IP数据报的目的地址，此时，主机可能还不知道它所在网络的网络掩码，甚至连它的 IP地址也不知道。

在任何情况下，路由器都不转发目的地址为受限的广播地址的数据报，这样的数据报仅出现在本地网络中。

一个未解的问题是：如果一个主机是多接口的，当一个进程向本网广播地址发送数据报时，为实现广播，是否应该将数据报发送到每个相连的接口上？如果不是这样，想对主机所有接口广播的应用必须确定主机中支持广播的所有接口，然后向每个接口发送一个数据报复制。

大多数BSD系统将255.255.255.255看作是配置后第一个接口的广播地址，并且不提供向所属具备广播能力的接口传送数据报的功能。不过， `routed`（见10.3节）和`rwhod`（BSD `rwho`客户的服务器）是向每个接口发送 UDP数据报的两个应用程序。这两个应用程序均用相似的启动过程来确定主机中的所有接口，并了解哪些接口具备广播能力。同时，将对应于那种接口的指向网络的广播地址作为发往该接口的数据报的目的地址。

> Host Requirements RFC没有进一步涉及多接口主机是否应当向其所有的接口发送受限的广播。

12.2.2 指向网络的广播

指向网络的广播地址是主机号为全 1的地址。A类网络广播地址为netid.255.255.255，其中netid为A类网络的网络号。

一个路由器必须转发指向网络的广播，但它也必须有一个不进行转发的选择。

12.2.3 指向子网的广播

指向子网的广播地址为主机号为全 1且有特定子网号的地址。作为子网直接广播地址的 IP地址需要了解子网的掩码。例如，如果路由器收到发往 128.1.2.255的数据报，当 B类网络

128.1的子网掩码为255.255.255.0时，该地址就是指向子网的广播地址；但如果该子网的掩码为255.255.254.0，该地址就不是指向子网的广播地址。

12.2.4 指向所有子网的广播

指向所有子网的广播也需要了解目的网络的子网掩码，以便与指向网络的广播地址区分开。指向所有子网的广播地址的子网号及主机号为全 1。例如，如果目的子网掩码为255.255.255.0，那么IP地址128.1.255.255是一个指向所有子网的广播地址。然而，如果网络没有划分子网，这就是一个指向网络的广播。

当前的看法[Almquist 1993]是这种广播是陈旧过时的，更好的方式是使用多播而不是对所有子网的广播。

> [Almquist 1993] 指出RFC 922要求将一个指向所有子网的广播传送给所有子网，但当前的路由器没有这么做。这很幸运，因为一个因错误配置而没有子网掩码的主机会把它的本地广播传送到所有子网。例如，如果IP地址为128.1.2.3的主机没有设置子网掩码，它的广播地址在正常情况下的默认值是128.1.255.255。但如果子网掩码被设置为255.255.255.0，那么由错误配置的主机发出的广播将指向所有的子网。
>
> 1983年问世的4.2BSD是第一个影响广泛的TCP/IP的实现，它使用主机号全0作为广播地址。一个最早提到广播IP地址的是IEN 212 [Gurwitz and Hinden 1982]，它提出用主机号中的1比特来表示IP广播地址（IENs 是互联网试验注释，基本上是RFC的前身）。RFC 894 [Hornig 1984]认为4.2BSD使用不标准的广播地址，但RFC 906 [Finlayson 1984]注意到对广播地址还没有Internet标准。RFC编辑在RFC 906中加了一个脚注承认缺少标准的广播地址，并强烈推荐将主机号全1作为广播地址。尽管1986年的4.3BSD采用主机号全1表示广播地址，但直到90年代早期，操作系统（著名的是SunOS 4.x）还继续使用非标准的广播地址。

12.3 广播的例子

广播是怎样传送的？路由器及主机又如何处理广播？很遗憾，这是难以回答的问题，因为它依赖于广播的类型、应用的类型、 TCP/IP实现方法以及有关路由器的配置。

首先，应用程序必须支持广播。如果执行

```
sun % ping 255.255.255.255
/usr/etc/ping: unknown host 255.255.255.255
```

打算在本地电缆上进行广播。但它无法进行，原因在于该应用程序（ ping）中存在一个程序设计上的问题。大多数应用程序收到点分十进制的 IP地址或主机名后，会调用函数inet_addr(3)来把它们转化为32 bit的二进制IP地址。假定要转化的是一个主机名，如果转化失败，该库函数将返回- 1来表明存在某种差错（例如是字符而不是数字或串中有小数点）。但本网广播地址（255.255.255.255）也被当作存在差错而返回- 1。大多数程序均假定接收到的字符串是主机名，然后查找DNS（第14章），失败后输出差错信息如“ 未知主机 ”。

如果我们修复ping程序中这个欠缺，结果也并不总是令人满意的。在 6个不同系统的测试中，仅有一个像预期的那样产生了一个本网广播数据报。大多数则在路由表中查找 IP地址255.255.255.255，而该地址被用作默认路由器地址，因此向默认路由器单播一个数据报。最

终该数据报被丢弃。

指向子网的广播是我们应该使用的。在6.3节中，我们向测试网络（见扉页前图）中IP地址为140.252.13.63 的以太网发送数据报，并接收以太网中所有主机的应答。与子网广播地址关联的每个接口是用于命令ifconfig（见3.8节）的值。如果我们ping那个地址，预期的结果是：

```
sun % arp -a                                        ARP高速缓存空

sun % ping 140.252.13.63
PING 140.252.13.63: 56 data bytes
64 bytes from sun (140.252.13.33): icmp_seq=0. time=4. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=0. time=172. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=192. ms

64 bytes from sun (140.252.13.33): icmp_seq=1. time=1. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=1. time=52. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=90. ms
^?                                                  键入中断以停止显示
----140.252.13.63 PING Statistics----
2 packets transmitted, 6 packets received, -200% packet loss
round-trip (ms)  min/avg/max = 1/85/192
                                                    再检验ARP缓存
sun % arp -a
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi (140.252.13.35) at 0:0:c0:6f:2d:40
```

IP通过目的地址（140.252.13.63）来确定，这是指向子网的广播地址，然后向链路层的广播地址发送该数据报。

在6.3节提到的这种广播类型的接收对象为局域网中包括发送主机在内的所有主机，因此可以看到除了收到网内其他主机的答复外，还收到来自发送主机（ sun）的答复。

在这个例子中，我们也显示了执行 ping广播地址前后ARP缓存的内容。这可以显示广播与ARP之间的相互作用。执行 ping命令前ARP缓存是空的，而执行后是满的（也就是说，对网内其他每个响应回显请求的主机在 ARP缓存中均有一个条目）。我们提到的该以太网数据帧被传送到链路层的广播地址（ 0xffffffff）是如何发生的呢？由 sun主机发送的数据帧不需要ARP。

如果使用tcpdump来观察ping的执行过程，可以看到广播数据帧的接收者在发送它的响应之前，首先产生一个对 sun主机的ARP请求，因为它的应答是单播的。在 4.5节我们介绍了一个ARP请求的接收者（该例中是sun）通常在发送ARP应答外，还将请求主机的IP地址和物理地址加入到ARP缓存中去。这基于这样一个假定：如果请求者向我们发送一个数据报，我们也很可能想向它发回什么。

我们使用的ping程序有些特殊，原因在于它使用的编程接口（在大多数 Unix实现中是低级插口(raw socket)）通常允许向一个广播地址发送数据报。如果使用不支持广播的应用如TFTP，情况又如何呢？（TFTP将在第15章详细介绍。）

```
bsdi %tftp                              启动客户程序
tftp> connect 140.252.13.63             说明服务器的IP地址
tftp> get temp.foo                      试图从服务器或获取一个文件
tftp: sendto: Permission denied
tftp> quit                              终止客户程序
```

在这个例子中，程序立即产生了一个差错，但不向网络发送任何信息。产生这一切的原因在于，插口提供的应用程序接口API只有在进程明确打算进行广播时才允许它向广播地址发送UDP

数据报。这主要是为了防止用户错误地采用了广播地址（正如此例）而应用程序却不打算广播。

在广播UDP数据报之前，使用插口中API的应用程序必须设置SO_BROADCAST插口选项。

并非所有系统均强制使用这个限制。某些系统中无需进程进行这个说明就能广播UDP数据报。而某些系统则有更多的限制，需要有超级用户权限的进程才能广播。

下一个问题是是否转发广播数据。有些系统内核和路由器有一选项来控制允许或禁止这一特性（见附录E）。

如果让路由器bsdi能够转发广播数据，然后在主机slip上运行ping程序，就能够观察到由路由器bsdi转发的子网广播数据报。转发广播数据报意味着路由器接收广播数据，确定该目的地址是对哪个接口的广播，然后用链路层广播向对应的网络转发数据报。

```
slip % ping 140.252.13.63
PING 140.252.13.63 (140.252.13.63): 56 data bytes
64 bytes from 140.252.13.35: icmp_seq=0 ttl=255 time=190 ms
64 bytes from 140.252.13.33: icmp_seq=0 ttl=254 time=280 ms (DUP!)
64 bytes from 140.252.13.34: icmp_seq=0 ttl=254 time=360 ms (DUP!)

64 bytes from 140.252.13.35: icmp_seq=1 ttl=255 time=180 ms
64 bytes from 140.252.13.33: icmp_seq=1 ttl=254 time=270 ms (DUP!)
64 bytes from 140.252.13.34: icmp_seq=1 ttl=254 time=360 ms (DUP!)
^?                                  键入中断以停止显示
--- 140.252.13.63 ping statistics ---
3 packets transmitted, 2 packets received, +4 duplicates, 33% packet loss
round-trip min/avg/max = 180/273/360 ms
```

我们观察到它的确正常工作了，同时也看到BSD系统中的ping程序检查重复的数据报序列号。如果出现重复序列号的数据报就显示DUP!，这意味着一个数据报已经在某处重复了，然而它正是我们所期望看到的，因为我们正向一个广播地址发送数据。

我们还可以从远离广播所指向的网络上的主机上来进行这个试验。在主机angogh.cx.berkeley.edu（和我们的网络距离14跳）上运行ping程序，如果路由器sun被设置为能够转发所指向的广播，它还能正常工作。在这种情况下，这个IP数据报（传送ICMP回显请求）被路径上的每个路由器像正常的数据报一样转发，它们均不知道传送的实际上是广播数据。接着最后一个路由器netb看到主机号为63，就将其转发给路由器sun。路由器sun觉察到该目的IP地址事实上是一个相连子网接口上的广播地址，就将该数据报以链路层广播传往相应网络。

广播是一种应该谨慎使用的功能。在许多情况下，IP多播被证明是一个更好的解决办法。

12.4 多播

IP多播提供两类服务：

1) 向多个目的地址传送数据。有许多向多个接收者传送信息的应用：例如交互式会议系统和向多个接收者分发邮件或新闻。如果不采用多播，目前这些应用大多采用TCP来完成（向每个目的地址传送一个单独的数据复制）。然而，即使使用多播，某些应用可能继续采用TCP来保证它的可靠性。

2) 客户对服务器的请求。例如，无盘工作站需要确定启动引导服务器。目前，这项服务是通过广播来提供的（正如第16章的BOOTP），但是使用多播可降低不提供这项服务主机的负担。

12.4.1 多播组地址

图12-2显示了D类IP地址的格式。

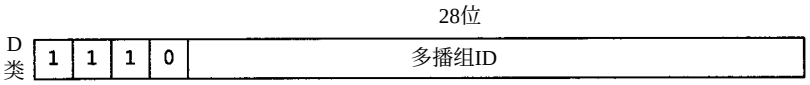


图12-2 D类IP地址格式

不像图1-5所示的其他三类IP地址（A、B和C），分配的28 bit均用作多播组号而不再表示其他。

多播组地址包括为1110的最高4 bit和多播组号。它们通常可表示为点分十进制数，范围从224.0.0.0到239.255.255.255。

能够接收发往一个特定多播组地址数据的主机集合称为主机组 (host group)。一个主机组可跨越多个网络。主机组中成员可随时加入或离开主机组。主机组中对主机的数量没有限制，同时不属于某一主机组的主机可以向该组发送信息。

一些多播组地址被IANA确定为知名地址。它们也被当作永久主机组，这和 TCP及UDP中的熟知端口相似。同样，这些知名多播地址在 RFC最新分配数字中列出。注意这些多播地址所代表的组是永久组，而它们的组成员却不是永久的。

例如，224.0.0.1代表“ 该子网内的所有系统组 ”，224.0.0.2代表“ 该子网内的所有路由器组 ”。多播地址224.0.1.1用作网络时间协议 NTP，224.0.0.9用作RIP-2(见10.5节)，224.0.1.2用作SGI公司的dogfight应用。

12.4.2 多播组地址到以太网地址的转换

IANA拥有一个以太网地址块，即高位 24 bit为00:00:5e（十六进制表示），这意味着该地址块所拥有的地址范围从00:00:5e:00:00:00到00:00:5e:ff:ff:ff。IANA将其中的一半分配为多播地址。为了指明一个多播地址，任何一个以太网地址的首字节必须是 01，这意味着与IP多播相对应的以太网地址范围从01:00:5e:00:00:00到01:00:5e:7f:ff:ff。

> 这里对CSMA/CD或令牌网使用的是Internet标准比特顺序，和在内存中出现的比特顺序一样。这也是大多数程序设计员和系统管理员采用的顺序。 IEEE文档采用了这种比特传输顺序。Assigned Numbers RFC给出了这些表示的差别。

这种地址分配将使以太网多播地址中的 23bit与IP多播组号对应起来，通过将多播组号中的低位23bit映射到以太网地址中的低位23bit实现，这个过程如图12-3所示。

由于多播组号中的最高 5 bit在映射过程中被忽略，因此每个以太网多播地址对应的多播组是不唯一的。 32个不同的多播组号被映射为一个以太网地址。例如，多播地址224.128.64.32（十六进制e0.80.40.20）和224.0.64.32（十六进制e0.00.40.20）都映射为同一以太网地址01:00:5e:00:40:20。

既然地址映射是不唯一的，那么设备驱动程序或 IP层（见图12-1）就必须对数据报进行过滤。因为网卡可能接收到主机不想接收的多播数据帧。另外，如果网卡不提供足够的多播数据帧过滤功能，设备驱动程序就必须接收所有多播数据帧，然后对它们进行过滤。

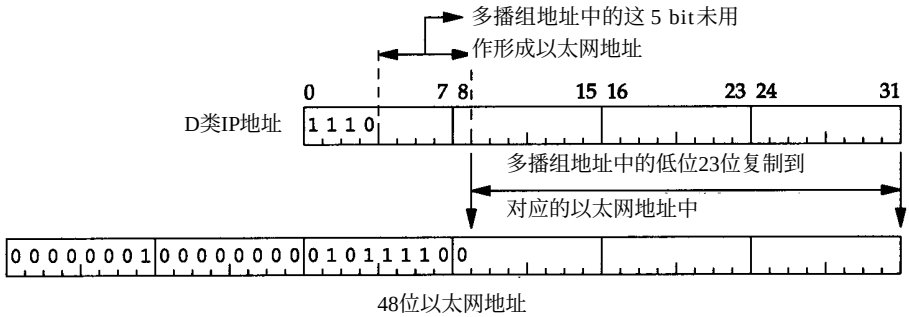


图12-3 D类IP地址到以太网多播地址的映射

局域网网卡趋向两种处理类型：一种是网卡根据对多播地址的散列值实行多播过滤，这意味仍会接收到不想接收的多播数据；另一种是网卡只接收一些固定数目的多播地址，这意味着当主机想接收超过网卡预先支持多播地址以外的多播地址时，必须将网卡设置为“多播混杂(multicast promiscuous)”模式。因此，这两种类型的网卡仍需要设备驱动程序检查收到的帧是否真是主机所需要的。

即使网卡实现了完美的多播过滤（基于48 bit的硬件地址），由于从D类IP地址到48 bit的硬件地址的映射不是一对一的，过滤过程仍是必要的。

尽管存在地址映射不完美和需要硬件过滤的不足，多播仍然比广播好。

单个物理网络的多播是简单的。多播进程将目的IP地址指明为多播地址，设备驱动程序将它转换为相应的以太网地址，然后把数据发送出去。这些接收进程必须通知它们的IP层，它们想接收的发往给定多播地址的数据报，并且设备驱动程序必须能够接收这些多播帧。这个过程就是“加入一个多播组”（使用“接收进程”复数形式的原因在于对一确定的多播信息，在同一主机或多个主机上存在多个接收者，这也是为什么要首先使用多播的原因）。当一个主机收到多播数据报时，它必须向属于那个多播组的每个进程均传送一个复制。这和单个进程收到单播UDP数据报的UDP不同。使用多播，一个主机上可能存在多个属于同一多播组的进程。

当把多播扩展到单个物理网络以外需要通过路由器转发多播数据时，复杂性就增加了。需要有一个协议让多播路由器了解确定网络中属于确定多播组的任何一个主机。这个协议就是Internet组管理协议（IGMP），也是下一章介绍的内容。

12.4.3 FDDI和令牌环网络中的多播

FDDI网络使用相同的D类IP地址到48 bit FDDI地址的映射过程[Katz 1990]。令牌环网络通常使用不同的地址映射方法，这是因为大多数令牌控制中的限制。

12.5 小结

广播是将数据报发送到网络中的所有主机（通常是本地相连的网络），而多播是将数据报发送到网络的一个主机组。这两个概念的基本点在于当收到送往上一个协议栈的数据帧时采用不同类型的过滤。每个协议层均可以因为不同的理由丢弃数据报。

目前有四种类型的广播地址：受限的广播、指向网络的广播、指向子网的广播和指向所有子网的广播。最常用的是指向子网的广播。受限的广播通常只在系统初始启动时才会用到。

试图通过路由器进行广播而发生的问题，常常是因为路由器不了解目的网络的子网掩码。结果与多种因素有关：广播地址类型、配置参数等等。

D类IP地址被称为多播组地址。通过将其低位 23 bit映射到相应以太网地址中便可实现多播组地址到以太网地址的转换。由于地址映射是不唯一的，因此需要其他的协议实现额外的数据报过滤。

习题

12.1 广播是否增加了网络通信量？

12.2 考虑一个拥有50台主机的以太网：20台运行TCP/IP，其他30台运行其他的协议族。主机如何处理来自运行另一个协议族主机的广播？

12.3 登录到一个过去从来没有用过的Unix系统，并且打算找出所有支持广播的接口的指向子网的广播地址。如何做到这点？

12.4 如果我们用ping程序向一个广播地址发送一个长的分组，如

```
sun % ping 140.252.13.63 1472
PING 140.252.13.63: 1472 data bytes
1480 bytes from sun (140.252.13.33): icmp_seq=0. time=6. ms
1480 bytes from svr4 (140.252.13.34): icmp_seq=0. time=84. ms
1480 bytes from bsdi (140.252.13.35): icmp_seq=0. time=128. ms
```

它正常工作，但将分组的长度再增加一个字节后出现如下差错：

```
sun % ping 140.252.13.63 1473
PING 140.252.13.63: 1473 data bytes
sendto: Message too long
```

究竟出了什么问题？

12.5 重做习题10.6，假定8个RIP报文是通过多播而不是广播（使用 RIP 版本2）。有什么变化？

第13章 IGMP：Internet组管理协议

13.1 引言

12.4节概述了IP多播给出，并介绍了D类IP地址到以太网地址的映射方式。也简要说明了在单个物理网络中的多播过程，但当涉及多个网络并且多播数据必须通过路由器转发时，情况会复杂得多。

本章将介绍用于支持主机和路由器进行多播的Internet组管理协议（IGMP）。它让一个物理网络上的所有系统知道主机当前所在的多播组。多播路由器需要这些信息以便知道多播数据报应该向哪些接口转发。 IGMP在RFC 1112中定义 [Deering 1989]。

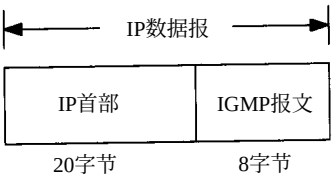


图13-1 IGMP报文封装在IP数据报中

正如ICMP一样， IGMP也被当作IP层的一部分。IGMP报文通过IP数据报进行传输。不像我们已经见到的其他协议， IGMP有固定的报文长度，没有可选数据。图13-1显示了IGMP报文如何封装在IP数据报中。

IGMP报文通过IP首部中协议字段值为2来指明。

13.2 IGMP报文

图13-2显示了长度为8字节的IGMP报文格式。

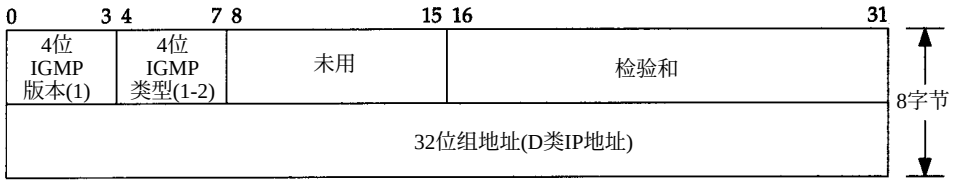


图13-2 IGMP报文的字段格式

这是版本为1的IGMP。IGMP类型为1说明是由多播路由器发出的查询报文，为2说明是主机发出的报告报文。检验和的计算和ICMP协议相同。

组地址为D类IP地址。在查询报文中组地址设置为0，在报告报文中组地址为要参加的组地址。在下一节中，当介绍IGMP如何操作时，我们将会更详细地了解它们。

13.3 IGMP 协议

13.3.1 加入一个多播组

多播的基础就是一个进程的概念（使用的术语进程是指操作系统执行的一个程序），该进程在一个主机的给定接口上加入了一个多播组。在一个给定接口上的多播组中的成员是动态

的——它随时因进程加入和离开多播组而变化。

这里所指的进程必须以某种方式在给定的接口上加入某个多播组。进程也能离开先前加入的多播组。这些是一个支持多播主机中任何 API所必需的部分。使用限定词“接口”是因为多播组中的成员是与接口相关联的。一个进程可以在多个接口上加入同一多播组。

Stanford大学伯克利版Unix中的IP 多播详细说明了有关socket API的变化，这些变化在Solaris 2.x和ip(7)的文档中也提供了。

这里暗示一个主机通过组地址和接口来识别一个多播组。主机必须保留一个表，此表中包含所有至少含有一个进程的多播组以及多播组中的进程数量。

13.3.2 IGMP 报告和查询

多播路由器使用IGMP报文来记录与该路由器相连网络中组成员的变化情况。使用规则如下：

1) 当第一个进程加入一个组时，主机就发送一个 IGMP报告。如果一个主机的多个进程加入同一组，只发送一个IGMP报告。这个报告被发送到进程加入组所在的同一接口上。

2) 进程离开一个组时，主机不发送 IGMP报告，即便是组中的最后一个进程离开。主机知道在确定的组中已不再有组成员后，在随后收到的 IGMP查询中就不再发送报告报文。

3) 多播路由器定时发送IGMP查询来了解是否还有任何主机包含有属于多播组的进程。多播路由器必须向每个接口发送一个 IGMP查询。因为路由器希望主机对它加入的每个多播组均发回一个报告，因此IGMP查询报文中的组地址被设置为0。

4) 主机通过发送IGMP报告来响应一个IGMP查询，对每个至少还包含一个进程的组均要发回IGMP报告。

使用这些查询和报告报文，多播路由器对每个接口保持一个表，表中记录接口上至少还包含一个主机的多播组。当路由器收到要转发的多播数据报时，它只将该数据报转发到（使用相应的多播链路层地址）还拥有属于那个组主机的接口上。

图13-3显示了两个IGMP报文，一个是主机发送的报告，另一个是路由器发送的查询。该路由器正在要求那个接口上的每个主机说明它加入的每个多播组。

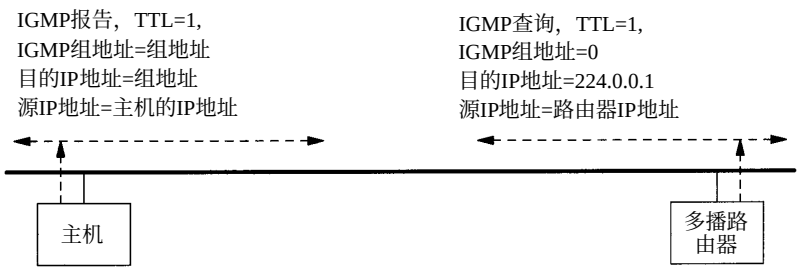


图13-3 IGMP的报告和查询

对TTL字段我们将在本节的后面介绍。

13.3.3 实现细节

为改善该协议的效率，有许多实现的细节要考虑。首先，当一个主机首次发送 IGMP报告

（当第一个进程加入一个多播组）时，并不保证该报告被可靠接收（因为使用的是IP交付服务）。下一个报告将在间隔一段时间后发送。这个时间间隔由主机在 0~10秒的范围内随机选择。

其次，当一个主机收到一个从路由器发出的查询后，并不立即响应，而是经过一定的时间间隔后才发出一些响应（采用“响应”的复数形式是因为该主机必须对它参加的每个组均发送一个响应）。既然参加同一多播组的多个主机均能发送一个报告，可将它们的发送间隔设置为随机时延。在一个物理网络中的所有主机将收到同组其他主机发送的所有报告，因为如图13-3所示的报告中的目的地址是那个组地址。这意味着如果一个主机在等待发送报告的过程中，却收到了发自其他主机的相同报告，则该主机的响应就可以不必发送了。因为多播路由器并不关心有多少主机属于该组，而只关心该组是否还至少拥有一个主机。的确，一个多播路由器甚至不关心哪个主机属于一个多播组。它仅仅想知道在给定的接口上的多播组中是否还至少有一个主机。

在没有任何多播路由器的单个物理网络中，仅有的 IGMP通信量就是在主机加入一个新的多播组时，支持IP多播的主机所发出的报告。

13.3.4 生存时间字段

在图13-3中，我们注意到IGMP报告和查询的生存时间 (TTL)均设置为1，这涉及到IP首部中的TTL字段。一个初始TTL为0的多播数据报将被限制在同一主机。在默认情况下，待传多播数据报的TTL被设置为1，这将使多播数据报仅局限在同一子网内传送。更大的 TTL值能被多播路由器转发。

回顾6.2节，对发往一个多播地址的数据报从不会产生 ICMP差错。当TTL值为0时，多播路由器也不产生ICMP“超时”差错。

> 在正常情况下，用户进程不关心传出数据报的TTL。然而，一个例外是Traceroute程序（第8章），它主要依据设置TTL值来完成。既然多播应用必须能够设置要传送数据报的TTL值，这意味着程序设计接口必须为用户进程提供这种能力。

通过增加TTL值的方法，一个应用程序可实现对一个特定服务器的扩展环搜索（expanding ring search)。第一个多播数据报以TTL等于1发送。如果没有响应，就尝试将 TTL设置为2，然后3，等等。在这种方式下，该应用能找到以跳数来度量的最近的服务器。

从224.0.0.0到224.0.0.255的特殊地址空间是打算用于多播范围不超过 1跳的应用。不管TTL值是多少，多播路由器均不转发目的地址为这些地址中的任何一个地址的数据报。

13.3.5 所有主机组

在图13-3中，我们看到了路由器的 IGMP查询被送到目的IP地址224.0.0.1。该地址被称为所有主机组地址。它涉及在一个物理网络中的所有具备多播能力的主机和路由器。当接口初始化后，所有具备多播能力接口上的主机均自动加入这个多播组。这个组的成员无需发送IGMP报告。

13.4 一个例子

现在我们已经了解了一些IP多播的细节，再来看看所包含的信息。我们使 sun主机能够支

持多播，并将采用一些多播软件所提供的测试程序来观察具体的过程。

首先，采用一个经过修改的netstat命令来报告每个接口上的多播组成员情况（在3.9节显示了netstat-ni命令的输出结果）。在下面的输出中，用黑体表示有关的多播组。

```
sun % netstat -nia
Name  Mtu   Network      Address             Ipkts Ierrs    Opkts Oerrs  Coll
le0   1500  140.252.13.  140.252.13.33        4370     0     4924     0     0
                         224.0.0.1
                         08:00:20:03:f6:42
                         01:00:5e:00:00:01
sl0   552   140.252.1    140.252.1.29        13587     0    15615     0     0
                         224.0.0.1
lo0   1536  127          127.0.0.1            1351     0     1351     0     0
                         224.0.0.1
```

其中，-n参数将以数字形式显示IP地址（而不是按名字来显示它们），-i参数将显示接口的统计结果，-a参数将显示所有配置的接口。

输出结果中的第2行le0（以太网）显示了这个接口属于主机组224.0.0.1（“所有主机”），和两行地址，后一行显示相应的以太网地址为：01:00:5e:00:00:01。这正是我们期望看到的以太网地址，和12.4节介绍的地址映射一致。我们还看到其他两个支持多播的接口：SLIP接口sl0和回送接口lo0，它们也属于所有主机组。

我们也必须显示IP路由表，用于多播的路由表同正常的路由表一样。黑体表项显示了所有传往224.0.0.0的数据报均被送往以太网：

```
sun % netstat -rn
Routing tables
Destination          Gateway             Flags    Refcnt Use      Interface
140.252.13.65        140.252.13.35       UGH      0      32       le0
127.0.0.1            127.0.0.1           UH       1      381      lo0
140.252.1.183        140.252.1.29        UH       0      6        sl0
default              140.252.1.183       UG       0      328      sl0
224.0.0.0            140.252.13.33       U        0      66       le0
140.252.13.32        140.252.13.33       U        8      5581     le0
```

如果将这个路由表与9.2节中sun路由器的路由表作比较，会发现只是多了有关多播的条目。

现在使用一个测试程序来让我们能在一个接口上加入一个多播组（不再显示使用这个测试程序的过程）。在以太网接口（140.252.13.33）上加入多播组224.1.2.3。执行netstat程序看到内核已加入这个组，并得到期望的以太网地址。用黑体字来突出显示和前面netstat输出的不同。

```
sun % netstat -nia
Name  Mtu   Network      Address             Ipkts Ierrs    Opkts Oerrs  Coll
le0   1500  140.252.13.  140.252.13.33        4374     0     4929     0     0
                         224.1.2.3
                         224.0.0.1
                         08:00:20:03:f6:42
                         01:00:5e:01:02:03
                         01:00:5e:00:00:01
sl0   552   140.252.1    140.252.1.29        13862     0    15943     0     0
                         224.0.0.1
lo0   1536  127          127.0.0.1            1360     0     1360     0     0
                         224.0.0.1
```

我们在输出中再次显示了其他两个接口：sl0和lo0，目的是为了重申加入多播组只发生在一个接口上。

图13-4显示了tcpdump对进程加入这个多播组的跟踪过程。

```
1  0.0                  8:0:20:3:f6:42 1:0:5e:1:2:3 ip 60:
                        sun > 224.1.2.3: igmp report 224.1.2.3 [ttl 1]

2  6.94 (6.94)          8:0:20:3:f6:42 1:0:5e:1:2:3 ip 60:
                        sun > 224.1.2.3: igmp report 224.1.2.3 [ttl 1]
```

图13-4 当一个主机加入1个多播组时tcpdump 的输出结果

当主机加入多播组时产生第1行的输出显示。第2行是经过时延后的IGMP报告，我们介绍过报告重发的时延是10秒内的随机时延。

在两行中显示硬件地址证实了以太网目的地址就是正确的多播地址。我们也看到了源 IP地址为相应的sun主机地址，而目的IP地址是多播组地址。同时，报告的地址和期望的多播组地址是一致的。

最后，我们注意到，正像指明的那样， TTL是1。当TTL的值为0或1时，tcpdump在打印时用方括号将它们括起来，这是因为 TTL在正常情况下均高于这些值。然而，使用多播我们期望看到许多TTL为1的IP数据报。

在这个输出中暗示了一个多播路由器必须接收在它所有接口上的所有多播数据报。路由器无法确定主机可能加入哪个多播组。

多播路由器的例子

继续前面的例子，但我们将在 sun主机中启动一个多播选路的守护程序。这里我们感兴趣的并不是多播选路协议，而是要研究所交换的 IGMP查询和报告。即使多播选路守护程序只运行在支持多播的主机（ sun）上，所有的查询和报告都将在那个以太网上进行多播，所以我们在该以太网中的其他系统中也能观察到它们。

在启动选路守护程序之前，加入另外一个多播组 224.9.9.9，图13-5显示了输出的结果。

```
1    0.0                sun > 224.0.0.4: igmp report 224.0.0.4

2    0.00 (  0.00)      sun > 224.0.0.1: igmp query
3    5.10 (  5.10)      sun > 224.9.9.9: igmp report 224.9.9.9

4    5.22 (  0.12)      sun > 224.0.0.1: igmp query
5    7.90 (  2.68)      sun > 224.1.2.3: igmp report 224.1.2.3
6    8.50 (  0.60)      sun > 224.0.0.4: igmp report 224.0.0.4
7   11.70 (  3.20)      sun > 224.9.9.9: igmp report 224.9.9.9

8  125.51 (113.81)      sun > 224.0.0.1: igmp query
9  125.70 (  0.19)      sun > 224.9.9.9: igmp report 224.9.9.9
10 128.50 (  2.80)      sun > 224.1.2.3: igmp report 224.1.2.3
11 129.10 (  0.60)      sun > 224.0.0.4: igmp report 224.0.0.4

12 247.82 (118.72)      sun > 224.0.0.1: igmp query
13 248.09 (  0.27)      sun > 224.1.2.3: igmp report 224.1.2.3
14 248.69 (  0.60)      sun > 224.0.0.4: igmp report 224.0.0.4
15 255.29 (  6.60)      sun > 224.9.9.9: igmp report 224.9.9.9
```

图13-5 当多播选路守护程序运行时tcpdump 的输出结果

在这个输出中没有包括以太网地址，因为已经证实了它们是正确的。也删去了 TTL等于1的说明，同样因为它们也是我们期望的那样。

当选路守护程序启动时，输出第1行。它发出一个已经加入了组224.0.0.4的报告。多播地址224.0.0.4是一个知名的地址，它被当前用于多播选路的距离向量多播选路协议 DVMRP

(Distance Vector Multicast Routing Protocol)所使用（DVMRP在RFC 1075中定义[Waitzman, Partridge, and Deering]）。

在该守护程序启动时，它也发送一个 IGMP查询（第2行）。该查询的目的IP地址为224.0.0.1（所有主机组），如图13-3所示。

第一个报告（第3行）大约在5秒后收到，报告给组224.9.9.9。这是在下一个查询发出之前（第4行）收到的唯一报告。当守护程序启动后，两次查询（第2行和第4行）发出的间隔很短，这是因为守护程序要将其多播路由表尽快建立起来。

第5、6和7行正是我们期望看到的：sun主机针对它所属的每个组发出一个报告。注意组224.0.0.4是被报告的，而其他两个组则是明确加入的，因为只要选路守护程序还在运行，它始终要属于组224.0.0.4。

下一个查询位于第8行，大约在前一个查询的2分钟后发出。它再次引发三个我们所期望的报告（第9、10和11行）。这些报告的时间顺序与前面不同，因为接收查询和发送报告的时间是随机的。

最后的查询在前一个查询的大约2分钟后发出，我们再次得到了期望的响应。

13.5 小结

多播是一种将报文发往多个接收者的通信方式。在许多应用中，它比广播更好，因为多播降低了不参与通信的主机的负担。简单的主机成员报告协议 (IGMP)是多播的基本模块。

在一个局域网中或跨越邻近局域网的多播需要使用本章介绍的技术。广播通常局限在单个局域网中，对目前许多使用广播的应用来说，可采用多播来替代广播。

然而，多播还未解决的一个问题是在广域网内的多播。[Deering and Cheriton 1990] 提出扩展目前的路由协议来支持多播。9.13节中的[Perlman 1992]讨论了广域网多播的一些问题。

[Casner and Deering 1992] 介绍了使用多播和一个称为MBONE（多播主干）的虚拟网络在整个Internet上传送IETF会议的情况。

习题

13.1 我们知道主机通过设置随机时延来调度 IGMP的发送。一个局域网中的主机采取什么措施才能避免两台主机产生相同的随机时延？

13.2 在[Casner and Deering 1992]中，他们提到UDP缺少两个通过MBONE传送音频采样数据的条件：分组失序检测和分组重复检测。你怎样在UDP上增加这些功能？

第14章 DNS：域名系统

14.1 引言

域名系统（DNS）是一种用于TCP/IP应用程序的分布式数据库，它提供主机名字和 IP地址之间的转换及有关电子邮件的选路信息。这里提到的分布式是指在 Internet上的单个站点不能拥有所有的信息。每个站点（如大学中的系、校园、公司或公司中的部门）保留它自己的信息数据库，并运行一个服务器程序供 Internet上的其他系统（客户程序）查询。 DNS提供了允许服务器和客户程序相互通信的协议。

从应用的角度上看，对 DNS的访问是通过一个地址解析器（ resolver）来完成的。在Unix主机中，该解析器主要是通过两个库函数 `gethostbyname`(3) 和`gethostbyaddr`(3)来访问的，它们在编译应用程序时与应用程序连接在一起。前者接收主机名字返回 IP地址，而后者接收IP地址来寻找主机名字。解析器通过一个或多个名字服务器来完成这种相互转换。

图4-2中指出了解析器通常是应用程序的一部分。解析器并不像 TCP/IP协议那样是操作系统的内核。该图指出的另一个基本概念就是：在一个应用程序请求 TCP打开一个连接或使用UDP发送一个数据报之前。心须将一个主机名转换为一个 IP地址。操作系统内核中的 TCP/IP协议族对于DNS一点都不知道。

本章我们将了解地址解析器如何使用 TCP/IP协议（主要是UDP）与名字服务器通信。我们不介绍运行名字服务器或有关可选参数的细节，这些技术细节的内容可以覆盖整整一本书。（见[Albitz and Liu 1992]标准Unix解析器和名字服务器介绍）。

RFC 1034 [Mockapetris 1987a] 说明了DNS的概念和功能， RFC 1035 [Mockapetris 1987b]详细说明了DNS 的规范和实现。DNS最常用的版本（包括解析器和名字服务器）是 BIND——伯克利Internet域名服务器。该服务器称作 named。[Danzig、Obraczka和Kumar 1992]分析了DNS 在广域网中产生的通信量。

14.2 DNS 基础

DNS的名字空间和Unix的文件系统相似，也具有层次结构。图 14-1 显示了这种层次的组织形式。

每个结点（图14-1中的圆圈）有一个至多 63个字符长的标识。这颗树的树根是没有任何标识的特殊结点。命名标识中一律不区分大写和小写。命名树上任何一个结点的域名就是将从该结点到最高层的域名串连起来，中间使用一个点“ . ”分隔这些域名（注意这和 Unix文件系统路径的形成不同，文件路径是由树根依次向下的形成的）。域名树中的每个结点必须有一个唯一的域名，但域名树中的不同结点可使用相同的标识。

以点“ . ”结尾的域名称为绝对域名或完全合格的域名 FQDN（Full Qualified Domain Name），例如`sun.tuc.noao.edu.`。如果一个域名不以点结尾，则认为该域名是不完全的。如何使域名完整依赖于使用的 DNS软件。如果不完整的域名由两个或两个以上的标号组成，

则认为它是完整的；或者在该域名的右边加入一个局部后缀。例如域名 sun通过加上局部后缀.tuc.noao.edu.成为完整的。

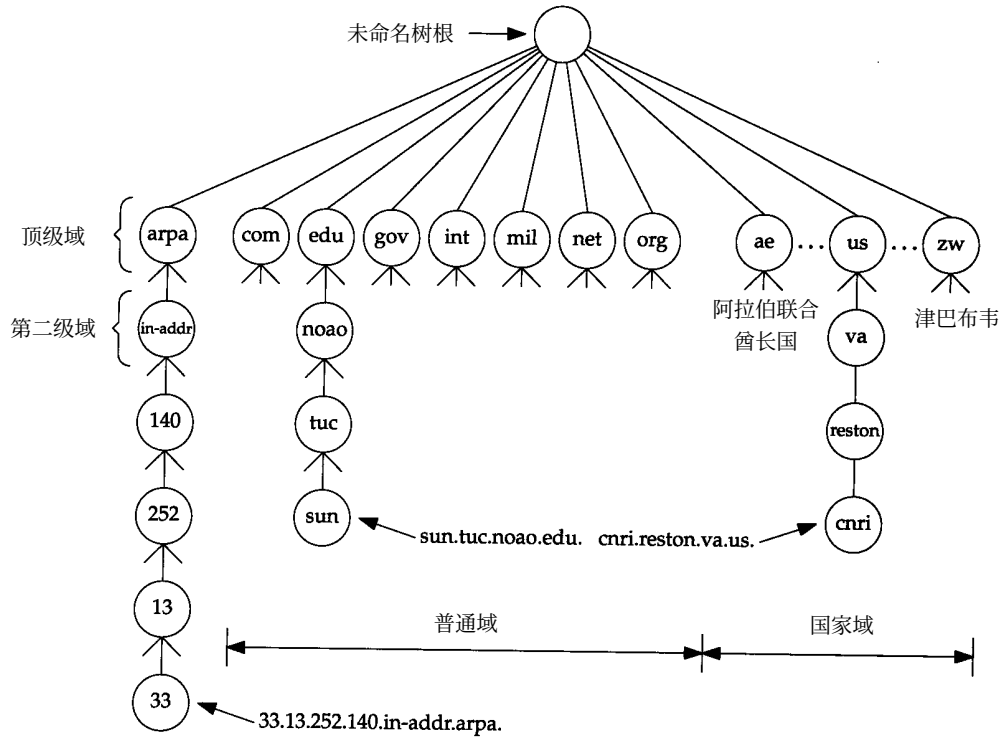


图14-1 DNS的层次组织

顶级域名被分为三个部分：

1) arpa是一个用作地址到名字转换的特殊域（我们将在 14.5节介绍）。

2) 7个3字符长的普通域。有些书也将这些域称为组织域。

3) 所有2字符长的域均是基于ISO3166中定义的国家代码，这些域被称为国家域，或地理域。

图14-2列出了7个普通域的正式划分。

域	描 述
com	商业组织
edu	教育机构
gov	其他美国政府部门
int	国际组织
mil	美国军事网点
net	网络
org	其他组织

图14-2 3字符长的普通域

在DNS中，通常认为3字符长的普通域仅用于美国的组织机构，2字符长的国家域则用于每个国家，但情况并不总是这样。许多非美国的组织机构仍然使用普通域，而一些美国的组织机构也使用 .us的国家域（ RFC 1480 [Cooper and Postel 1993] 详细描述了.us域）。普通域中只有.gov和.mil域局限于美国。

许多国家将它们的二级域组织成类似于普通域的结构：例如，.ac.uk是英国研究机构的二级域名，.co.uk则是英国商业机构的二级域名。

DNS的一个没在如图 14-1中表示出来的重要特征是 DNS中域名的授权。没有哪个机构来管理域名树中的每个标识，相反，只有一个机构，即网络信息中心 NIC负责分配顶级域和委派其他指定地区域的授权机构。

一个独立管理的DNS子树称为一个区域（zone）。一个常见的区域是一个二级域，如noao.edu。许多二级域将它们的区域划分成更小的区域。例如，大学可能根据不同的系来划分区域，公司可能根据不同的部门来划分区域。

如果你熟悉Unix的文件系统，会注意到DNS树中区域的划分同一个逻辑Unix文件系统到物理磁盘分区的划分很相似。正如无法确定图14-1中区域的具体位置，我们也不知道一个Unix文件系统中的目录位于哪个磁盘分区。

一旦一个区域的授权机构被委派后，由它负责向该区域提供多个名字服务器。当一个新系统加入到一个区域中时，该区域的DNS管理者为该新系统申请一个域名和一个IP地址，并将它们加到名字服务器的数据库中。这就是授权机构存在的必要性。例如，在一个小规模的大学，一个人就能完成每次新系统的加入。但对一个规模较大的大学来说，这一工作必须被专门委派的机构（可能是系）来完成，因为一个人已无法维持这一工作。

一个名字服务器负责一个或多个区域。一个区域的管理者必须为该区域提供一个主名字服务器和至少一个辅助名字服务器。主、辅名字服务器必须是独立和冗余的，以便当某个名字服务器发生故障时不会影响该区域的名字服务。

主、辅名字服务器的主要区别在于主名字服务器从磁盘文件中调入该区域的所有信息，而辅名字服务器则从主服务器调入所有信息。我们将辅名字服务器从主服务器调入信息称为区域传送。

当一个新主机加入一个区域时，区域管理者将适当的信息（最少包括名字和IP地址）加入到运行在主名字服务器上的一个磁盘文件中，然后通知主名字服务器重新调入它的配置文件。辅名字服务器定时（通常是每隔3小时）向主名字服务器询问是否有新数据。如果有新数据，则通过区域传送方式获得新数据。

当一个名字服务器没有请求的信息时，它将如何处理？它必须与其他的名字服务器联系。（这正是DNS的分布特性）。然而，并不是每个名字服务器都知道如何同其他名字服务器联系。相反，每个名字服务器必须知道如何同根的名字服务器联系。1993年4月时有8个根名字服务器，所有的主名字服务器都必须知道根服务器的IP地址（这些IP地址在主名字服务器的配置文件中，主服务器必须知道根服务器的IP地址，而不是它们的域名）。根服务器则知道所有二级域中的每个授权名字服务器的名字和位置（即IP地址）。这意味着这样一个反复的过程：正在处理请求的名字服务器与根服务器联系，根服务器告诉它与另一个名字服务器联系。在本章的后面我们将通过一些例子来详细了解这一过程。

你可以通过匿名的FTP获取当前的根服务器清单。具体是从ftp.rs.internic.net或nic.ddn.mil 获取文件netinfo/root-servers.txt。

DNS的一个基本特性是使用超高速缓存。即当一个名字服务器收到有关映射的信息（主机名字到IP地址）时，它会将该信息存放在高速缓存中。这样若以后遇到相同的映射请求，就能直接使用缓存中的结果而无需通过其他服务器查询。14.7节显示了一个使用高速缓存的例子。

14.3 DNS的报文格式

DNS定义了一个用于查询和响应的报文格式。图14-3显示这个报文的总体格式。

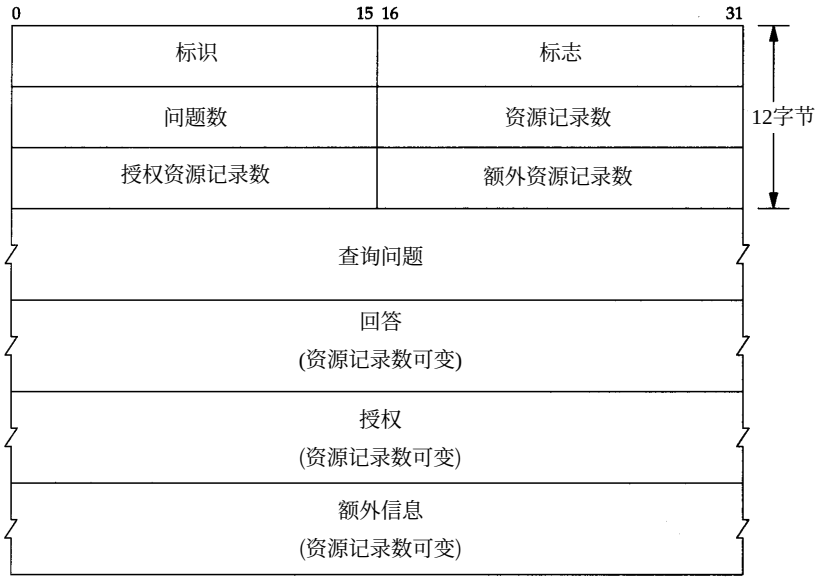


图14-3 DNS查询和响应的一般格式

这个报文由12字节长的首部和4个长度可变的字段组成。

标识字段由客户程序设置并由服务器返回结果。客户程序通过它来确定响应与查询是否匹配。

16 bit的标志字段被划分为若干子字段，如图 14-4所示。

图14-4 DNS报文首部中的标志字段

我们从最左位开始依次介绍各子字段：

- QR 是1 bit字段：0表示查询报文，1表示响应报文。
- opcode是一个4 bit字段：通常值为0（标准查询），其他值为1（反向查询）和2（服务器状态请求）。
- AA是1 bit标志，表示“授权回答 (authoritative answer)”。该名字服务器是授权于该域的。
- TC是1 bit字段，表示“可截断的 (truncated)”。使用UDP时，它表示当应答的总长度超过512字节时，只返回前512个字节。
- RD是1 bit字段表示“期望递归（recursion desired）”。该比特能在一个查询中设置，并在响应中返回。这个标志告诉名字服务器必须处理这个查询，也称为一个递归查询。如果该位为0，且被请求的名字服务器没有一个授权回答，它就返回一个能解答该查询的其他名字服务器列表，这称为迭代查询。在后面的例子中，我们将看到这两种类型查询的例子。
- RA是1 bit字段，表示“可用递归”。如果名字服务器支持递归查询，则在响应中将该比特设置为1。在后面的例子中可看到大多数名字服务器都提供递归查询，除了某些根服务器。

• 随后的3 bit字段必须为0。

• rcode是一个4 bit的返回码字段。通常的值为0（没有差错）和3（名字差错）。名字差错只有从一个授权名字服务器上返回，它表示在查询中制定的域名不存在。

随后的4个16 bit字段说明最后4个变长字段中包含的条目数。对于查询报文，问题(question)数通常是1，而其他3项则均为0。类似地，对于应答报文，回答数至少是1，剩下的两项可以是0或非0。

14.3.1 DNS查询报文中的问题部分

问题部分中每个问题的格式如图14-5所示，通常只有一个问题。

图14-5 DNS查询报文中问题部分的格式

查询名是要查找的名字，它是一个或多个标识符的序列。每个标识符以首字节的计数值来说明随后标识符的字节长度，每个名字以最后字节为0结束，长度为0的标识符是根标识符。计数字节的值必须是0~63的数，因为标识符的最大长度仅为63（在本节的后面我们将看到计数字节的最高两比特为1，即值192~255，将用于压缩格式）。不像我们已经看到的许多其他报文格式，该字段无需以整32 bit边界结束，即无需填充字节。

图14-6显示了如何存储域名gemini.tuc.noao.edu。

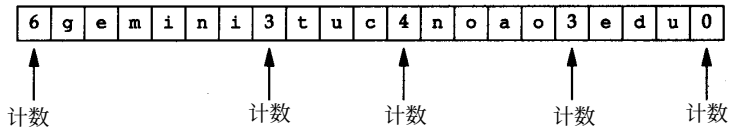


图14-6 域名gemini.tuc.noao.edu 的表示

每个问题有一个查询类型，而每个响应（也称一个资源记录，我们下面将谈到）也有一个类型。大约有20个不同的类型值，其中的一些目前已经过时。图14-7显示了其中的一些值。查询类型是类型的一个超集(superset)：图中显示的类型值中只有两个能用于查询类型。

名　字	数　值	描　述	类型?	查询类型
A	1	IP地址	•	•
NS	2	名字服务器	•	•
CNAME	5	规范名称	•	•
PTR	12	指针记录	•	•
HINFO	13	主机信息	•	•
MX	15	邮件交换记录	•	•
AXFR	252	对区域转换的请求		•
*或ANY	255	对所有记录的请求		•

图14-7 DNS问题和响应的类型值和查询类型值

最常用的查询类型是A类型，表示期望获得查询名的IP地址。一个PTR查询则请求获得一个IP地址对应的域名。这是一个指针查询，我们将在14.5节介绍。其他的查询类型将在14.6节介绍。

查询类通常是1，指互联网地址（某些站点也支持其他非IP地址）。

14.3.2 DNS响应报文中的资源记录部分

DNS报文中最后的三个字段，回答字段、授权字段和附加信息字段，均采用一种称为资源记录RR（Resource Record）的相同格式。图14-8显示了资源记录的格式。

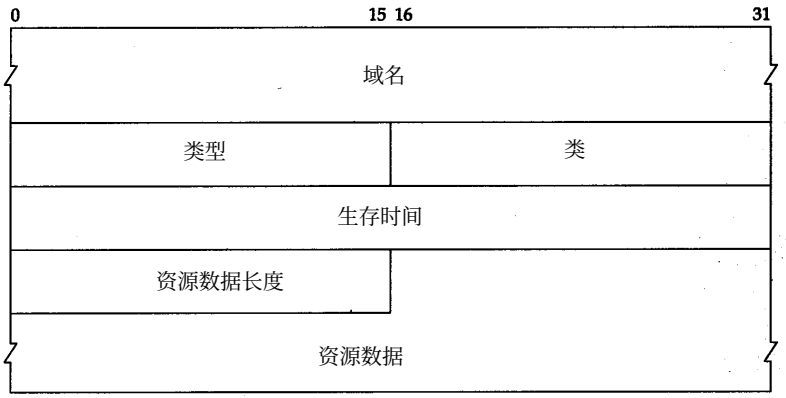


图14-8 DNS资源记录格式

域名是记录中资源数据对应的名字。它的格式和前面介绍的查询名字段格式（图 14-6）相同。

类型说明RR的类型码。它的值和前面介绍的查询类型值是一样的。类通常为 1，指Internet数据。

生存时间字段是客户程序保留该资源记录的秒数。资源记录通常的生存时间值为2天。

资源数据长度说明资源数据的数量。该数据的格式依赖于类型字段的值。对于类型 1（A记录）资源数据是4字节的IP地址。

现在已经介绍了DNS查询和响应的基本格式，我们将使用 tcpdump程序来观察具体的交换过程。

14.4 一个简单的例子

让我们从一个简单的例子来了解一个名字解析器与一个名字服务器之间的通信过程。在sun主机上运行Telnet客户程序远程登录到gemini主机上，并连接daytime服务器：

```
sun % telnet gemini daytime
Trying 140.252.1.11 ...                前3行的输出是从Telnet客户
Connected to gemini.tuc.noao.edu.
Escape character is '^]'.
Wed Mar 24 10:44:17 1993               这是从daytime服务器的输出
Connection closed by foreign host.     这是从Telnet客户的输出
```

在这个例子中，我们引导 sun主机（运行Telnet客户程序）上的名字解析器来使用位于noao.edu（140.252.1.54）的名字服务器。图14-9显示了这三个系统的排列情况。

和以前提到的一样，名字解析器是客户程序的一部分，并且在Telnet客户程序与daytime服务器建立TCP连接之前，名字解析器就能通过名字服务器获取IP地址。

在这个图中，省略了sun主机与140.252.1以太网的连接实际上是一个SLIP连接的细节（参见封2的插图），因为它不影响我们的讨论。通过在SLIP链路上运行tcpdump程序来了解名字解析器与名字服务器之间的分组交换。

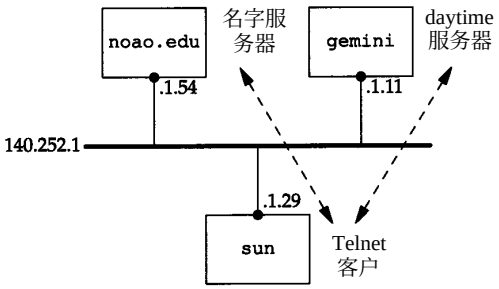


图14-9 用于简单DNS例子的系统

sun主机上的文件/etc/resolv.conf将告诉名字解析器作什么：

```
sun % cat /etc/resolv.conf
nameserver 140.252.1.54
domain tuc.noao.edu
```

第1行给出名字服务器——主机noao.edu的IP地址。最多可说明3个名字服务器行来提供足够的后备以防名字服务器故障或不可达。域名行说明默认域名。如果要查找的域名不是一个完全合格的域名（没有以句点结束），那末默认的域名.tuc.noao.edu将加到待查名后。

图14-10显示了名字解析器与名字服务器之间的分组交换。

```
1  0.0                    140.252.1.29.1447 > 140.252.1.54.53: 1+ A?
                          gemini.tuc.noao.edu. (37)

2  0.290820 (0.2908)      140.252.1.54.53 > 140.252.1.29.1447: 1* 2/0/0 A
                          140.252.1.11 (69)
```

图14-10 向名字服务器查询主机名gemini.tuc.noao.edu 的输出

让tcpdump程序不再显示每个IP数据报的源地址和目的地址。相反，它显示客户（resolver）的IP地址140.252.1.29和名字服务器的IP地址140.252.1.54。客户的临时端口号为1447，而名字服务器则使用熟知端口53。如果让tcpdump程序显示名字而不是IP地址，它可能会和同一个名字服务器联系（作指示查询），以致产生混乱的输出结果。

第1行中冒号后的字段（1+）表示标识字段为1，加号“+”表示RD标志（期望递归）为1。默认情况下，名字解析器要求递归查询方式。

下一个字段为A?，表示查询类型为A（我们需要一个IP地址），该问号指明它是一个查询（不是一个响应）。待查名字显示在后面：gemini.tuc.noao.edu.。名字解析器在待查名字后加上句点号指明它是一个绝对字段名。

在UDP数据报中的用户数据长度显示为37字节：12字节为固定长度的报文首部（图14-3）；21字节为查询名字（图14-6），以及用于查询类型和查询类的4个字节。在DNS报文中无需填充数据。

tcpdump程序的第2行显示的是从名字服务器发回的响应。1*是标识字段，星号表示设置AA标志（授权回答）（该服务器是noao.edu域的主域名服务器，其回答在该域内是可相信的。）

输出结果2/0/0表示在响应报文中最后3个变长字段的资源记录数：回答RR数为2，授权RR和附加信息RR数均为0。tcpdump仅显示第一个回答，回答类型为A（IP地址），值为140.252.1.11。

为什么我们的查询会得到两个回答？这是因为gemini是多接口主机，因此得到两个IP地址。事实上，另一个有用的DNS工具是一个称为host的公开程序，它能将查询传递给名字服务器，并显示返回的结果。如果使用这个程序，就能看到这个多地址主机的两个 IP地址：

```
sun % host gemini
gemini.tuc.noao.edu    A       140.252.1.11
gemini.tuc.noao.edu    A       140.252.3.54
```

图14-10中的第一个回答与host命令的第一行输出均是在同一子网（ 140.252.1）的IP地址。这不是偶然的。如果名字服务器和发出请求的主机位于相同的网络（或子网），那么BIND会排列显示的结果以便在相同网络的地址优先显示。

我们还可以使用其他的地址来访问gemini主机，但它可能不太有效。在这个例子中，使用traceroute显示出从子网140.252.1到140.252.3的正常路由不经过gemini主机，而是经过连接这两个网络的另一个路由器。因此在这种情况下，如果通过其他的IP地址（140.252.3.54）来访问gemini主机，所有分组均需经过额外的一跳。我们将在25.9节重新回到这个例子来探讨替换路由，那时可使用SNMP来查看一个路由器的路由表。

还有其他一些程序能很容易地对DNS进行交互访问。nslookup是大多数DNS实现中包含的程序。[Albitz and Liu 1992]的第10章详细介绍了该程序的使用方法。dig（“域名Internet搜索(Domain Internet Groper)”）程序是另一个查询DNS服务器的公开工具。doc（“域名模糊控制(Domain Obscenity Control)”）是一个使用dig的外壳脚本程序，它能向合适的名字服务器发送查询来诊断含义不清的域名，并对返回的查询结果进行简单的分析。附录F有如何获得这些程序的详细介绍。

在这个例子中要说明的最后一个问题是在查询结果中的 UDP数据长度：69字节。为说明这些字节需要知道以下两点：

1) 在返回的结果中包含查询问题。

2) 在返回的结果中会有许多重复的域名，因此使用压缩方式。在这个例子中，域名gemini.tuc.noao.edu出现了三次。

压缩方法很简单，当一个域名中的标识符是压缩的，它的单计数字节（范围由 0～63）中的最高两位将被设置为11。这表示它是一个16 bit指针而不再是8 bit的计数字节。指针中的剩下14 bit说明在该DNS报文中标识符所在的位置（起始位置由标识字段的第一字节起算）。我们明确说明只要一个标识符是压缩的，就可以使用这种指针，而不一定非要一个完整的域名压缩时才能使用。因为一个指针可能指向一个完整的域名，也可能只指向域名的结尾部分（这是因为给定域名的结尾标识符是相同的）。

图14-11显示了对应于图14-10的第2行的DNS应答的格式。我们也显示了IP首部和UDP首部来重申DNS报文被封装在UDP数据报中。还明确显示了在问题部分的域名中各标识符的计数字节。返回的两个回答除了返回的 IP地址不同外，其余都是一样的。在这个例子中，每个回答中的指针值为12，表示从DNS首部开始的偏移量。

在这个例子中最后要注意的是使用 telnet命令后输出的第2行，这里重复一下：

```
sun % telnet gemini daytime          我们只键入gemini
Trying 140.252.1.11 ...
Connected to gemini.tuc.noao.edu.  但Telnet客户输出FQDN
```

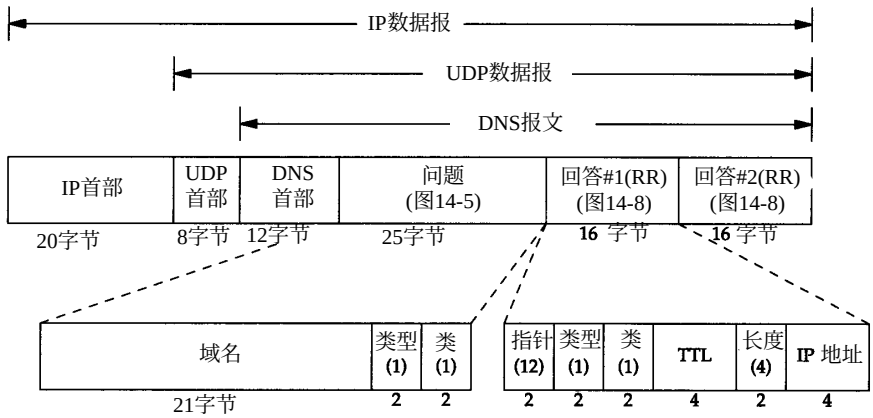


图14-11 对应于图14-10中第2行DNS应答的格式

我们仅仅输入了主机名(gemini)而不是FQDN，但Telnet客户程序部输出了FQDN。这是由于Telnet程序通过调用名字解析器（gethostbyname）对输入的名字进行查询，返回的结果包括IP地址和FQDN。Telnet程序就输出它试图与之建立TCP连接的IP地址，当连接建立后，它就输出FQDN。

如果在输入Telnet命令后间隔很长时间才显示IP地址，这个时延是由名字解析器和名字服务器在由域名到IP地址的解析所引起的。而显示Trying到显示Connected to的时延则是由客户与服务器建立TCP连接所引起的，与DNS无关。

14.5 指针查询

DNS中一直难于理解的部分就是指针查询方式，即给定一个IP地址，返回与该地址对应的域名。

首先回到图14-1，查看一下顶级域arpa，及它下面的in-addr域。当一个组织加入Internet，并获得DNS域名空间的授权，如noao.edu，则它们也获得了对应IP地址的in-addr.arpa域名空间的授权。在noao.edu这个例子中，它是网络号为140.252的B类网络。在DNS树中结点in-addr.arpa的下一级必须是该IP地址的第一字节（例中为140），再下一级为该IP地址的下一个字节（252），依此类推。但应牢记的是DNS名字是由DNS树的底部逐步向上书写的。这意味着对于IP地址为140.252.13.33的sun主机，它的DNS名字为33.13.252.140.in-addr.arpa。

必须写出4字节的IP地址，因为授权的代表是基于网络号：A类地址是第一字节，B类地址是第一、二字节，C类地址则是第一、二、三字节。IP地址的第一字节一定位于in-addr的下一级，但FQDN却是自树底往上书写的。如果FQDN由顶往下书写，则这个IP地址的DNS名字将是arpa.in-addr.140.252.13.33，而它所对应的域名将是edu.noao.tuc.sun。

如果DNS树中没有独立的分支来处理这种地址—名字的转换，将无法进行这种反向转换，除非从树根开始依次尝试每个顶级域。毫不夸张地说，这将需要数天或数周的时间。虽然反写IP地址和特殊的域名会造成某些混乱，但in-addr解决方案仍是一种最有效的方式。

只有在使用host程序或tcpdump程序直接同DNS打交道时，才会担心in-addr域和反写IP地址影响我们。从应用的角度上看，正常的名字解析器函数（gethostbyaddr）将接收一个IP地址并返回对应主机的有关信息。反转这些字节和添加in-addr.arpa域均由该函数自动完成。

14.5.1 举例

使用host程序完成一个指针查询，并使用tcpdump程序来观察这些分组。例子中的设置和图14-9相同，在sun主机上运行host程序，名字服务器在主机noao.edu上。我们指明svr4主机的IP地址：

```
sun % host 140.252.13.34
Name: svr4.tuc.noao.edu
Address: 140.252.13.34
```

既然IP地址是仅有的命令行参数，host程序将自动产生指针查询。图14-12显示了tcpdump的输出。

```
1  0.0                    140.252.1.29.1610 > 140.252.1.54.53: 1+ PTR?
                          34.13.252.140.in-addr.arpa. (44)

2  0.332288 (0.3323)      140.252.1.54.53 > 140.252.1.29.1610: 1* 1/0/0 PTR
                          svr4.tuc.noao.edu. (75)
```

图14-12 一个指针查询的tcpdump 输出

第1行显示标识符为1，期望递归标志设置为1（加号“+”），查询类型为PTR（应注意：问号“？”表示它是一个查询而不是响应）。44字节的数据包括12字节的DNS报文首部、28字节的域名标识符和4字节的查询类型和查询类。

查询结果包含一个回答RR，且为授权回答比特置1（带星号）。RR的类型是PTR，资源数据中包含该域名。

从名字解析器传递给名字服务器的指针查询不再是32 bit的IP地址，而是域名34.13.252.140.in-addr.arpa。

14.5.2 主机名检查

当一个IP数据报到达一个作为服务器的主机时，无论是UDP数据报还是TCP连接请求，服务器进程所能获得的是客户的IP地址和端口号（UDP或TCP）。某些服务器需要客户的IP地址来获得在DNS中的指针记录。在27.3节会看到这样的例子，从未知的IP地址使用匿名FTP访问服务器。

其他的一些服务器如Rlogin服务器（第26章）不但需要客户的IP地址来获得指针记录，还要向DNS询问该IP地址所对应的域名，并检查返回的地址中是否有地址与收到的数据报中的源IP地址匹配。该检查是因为.rhosts文件（见26.2节）中的条目仅包含主机名，而没有IP地址，因此主机需要证实该主机名是否对应源IP地址。

某些厂商将该项检查自动并入其名字解析器的例程中，特别是函数gethostbyaddr。这使得任何使用名字解析器的程序均可获得这种检查，而无需在应用中人为地进行这项检查。

来看一个使用SunOS 4.13名字解析器库的例子。我们编制了一个简单的程序通过调用函数gethostbyaddr来完成一个指针查询。我们已在文件/etc/resolv.conf中将名字服务器设置为noao.edu，sun主机通过SLIP链路与它相连。图14-13显示了当调用函数gethostbyaddr获取与IP地址140.252.1.29（sun主机）对应的名字时，tcpdump在SLIP链路上收到的内容。

第1行是预期的指针查询，第2行是预期的响应。但第3行显示了该名字解析器函数自动对第2行返回的名字发出一个IP地址查询。既然sun主机有两个IP地址，第4行的响应就包括两个

回答记录。如果这两个地址中没有与gethostbyaddr输入参数匹配的地址，函数会向系统的日志发送一条报文，并向应用程序返回差错。

```
1  0.0                    sun.1812 > noao.edu.domain: 1+ PTR?
                          29.1.252.140.in-addr.arpa. (43)
2  0.339091 (0.3391)      noao.edu.domain > sun.1812: 1* 1/0/0 PTR
                          sun.tuc.noao.edu. (73)

3  0.344348 (0.0053)      sun.1813 > noao.edu.domain: 2+ A?
                          sun.tuc.noao.edu. (33)
4  0.669022 (0.3247)      noao.edu.domain > sun.1813: 2* 2/0/0 A
                          140.252.1.29 (69)
```

图14-13 调用名字解析器函数执行指针查询

14.6 资源记录

至今我们已经见到了一些不同类型的资源记录（RR）：IP地址查询为A类型，指针查询为类型PTR。也已看到了由名字服务器返回的资源记录：回答RR、授权RR和附加信息RR。现有大约20种不同类型的资源记录，下面将介绍其中的一些。另外，随着时间的推移，会加入更多类型的RR。

A　一个A记录定义了一个IP地址，它存储32 bit的二进制数。

PTR　指针记录用于指针查询。IP地址被看作是in-addr.arpa域下的一个域名（标识符串）。

CNAME　这表示“规范名字(canonical name)”。它用来表示一个域名（标识符串），而有规范名字的域名通常被称为别名(alias)。某些FTP服务器使用它向其他的系统提供一个易于记忆的别名。

例如，gated服务器（10.3节提到）可通过匿名FTP从gated.cornell.edu获得，但这里并没有叫做gated的系统，这仅是为其他系统提供的别名。其他系统的规范名为gated.cornell.edu。

```
sun % host -t cname gated.cornell.edu
gated.cornell.edu  CNAM  COMET.CIT.CORNELL.EDU
```

这里使用的-t选项来指明它是特定的查询类型。

HINFO　表示主机信息：包括说明主机CPU和操作系统的两个字符串。并非所有的站点均提供它们系统的HINFO记录，并且提供的信息也可能不是最新的。

```
sun %host -t hinfo sun
sun.tuc.noao.edu  HINFO Sun-4/25        Sun4.1.3
```

MX　邮件交换记录，用于以下一些场合：(1) 一个没有连到Internet的站点能将一个连到Internet的站点作为它的邮件交换器。这两个站点能够用一种交替的方式交换到达的邮件，而通常使用的协议是UUCP协议。(2) MX记录提供了一种将无法到达其目的主机的邮件传送到一个替代主机的方式。(3) MX记录允许机构提供供他人发送邮件的虚拟主机，如cs.university.edu，即使这样的主机名根本不存在。(4) 防火墙网关能使用MX记录来限制外界与内部系统的连接。许多不能与Internet连接的站点通过UUCP链路与一个连接在Internet上的站点如UUNET相连接。通过MX记录能使用user@host这种邮件地址向那个站点发送电子邮件。例如，一个假想的域foo.com可能有下面的MX记录：

```
sun % host -t mx foo.com
foo.com              MX      relay1.UU.NET
foo.com              MX        relay2.UU.NET
```

MX记录能被连接在互联网主机中的邮件处理器使用。在这个例子中，其他的邮件处理器则被告知“如果有邮件要发往 user@foo.com，就将邮件送到relay1.uu.net或relay2.uu.net。”

每个MX记录被赋于一个16 bit的整数值，该值称为优先值。如果一个目的主机有多个MX记录，它们按优先值由小到大的顺序使用。

另一个MX记录的例子是处理主机脱机工作或不可达的情况。邮件处理器仅在无法使用TCP与目的主机连接时才使用MX记录。作者的主系统通过SLIP链路与互联网相连，它在大多数时间内是脱机工作的，我们有

```
sun % host -tv mx sun
Query about sun for record types MX
Trying sun within tuc.noao.edu ...
Query done, 2 answers, authoritative status: no error
sun.tuc.noao.edu     86400   IN     MX     0  sun.tuc.noao.edu
sun.tuc.noao.edu     86400   IN     MX     10 noao.edu
```

为了显示优先值，我们使用了-v选项（该选项也会导致其他字段的输出）。第二个字段，86400，是寿命值，单位为秒。因此该TTL值为24小时（24×60×60）。第3列，IN，是（Internet）类。我们看到直接传送给主机自身（第一个MX记录）有最低的优先值0。如果没有工作（即SLIP链路断开），会使用下一个更高优先值（10）的邮件记录，并试图向主机noao.edu传送。如果它仍没有成功，发送将超时并在以后重新发送。

NS　名字服务器记录。它说明一个域的授权名字服务器。它由域名表示（符号串）。在下节将看到这些类型的例子。

这些是RR的常用类型。将在后面的例子中遇到它们。

14.7 高速缓存

为了减少Internet上DNS的通信量，所有的名字服务器均使用高速缓存。在标准的Unix实现中，高速缓存是由名字服务器而不是由名字解析器维护的。既然名字解析器作为每个应用的一部分，而应用又不可能总处于工作状态，因此将高速缓存放在只要系统（名字服务器）处于工作状态就能起作用的程序中显得很重要。这样任何一个使用名字服务器的应用均可获得高速缓存。在该站点使用这个名字服务器的任何其他主机也能共享服务器的高速缓存。

在迄今为止（图14-9）所举例子的网络环境中，在sun主机上运行客户程序，通过主机noao.edu的SLIP链路访问名字服务器。现在将改变这种设置，在sun主机上运行名字服务器。在这种情况下，如果使用tcpdump监视在SLIP链路上的DNS通信量，将只能看到服务器因超出其高速缓存而不能处理的查询。

在默认情况下，名字解析器将在本地主机上（UDP端口号为53或TCP端口号为53）寻找名字服务器。从名字解析器文件中删除nameserver行，而留下domain行：

```
sun % cat  /etc/resolv.conf
domain tuc.noao.edu
```

在这个文件中缺少namerserver指示将导致名字解析器使用本地主机上的名字服务器。

使用host命令执行下列查询：

```
sun % host ftp.uu.net
ftp.uu.net              A           192.48.96.9
```

图14-14显示了这个查询的输出结果。

```
1   0.0                 sun.tuc.noao.edu.domain > NS.NIC.DDN.MIL.domain:
                        2 A? ftp.uu.net. (28)
2   0.559285 ( 0.5593)  NS.NIC.DDN.MIL.domain > sun.tuc.noao.edu.domain:
                        2- 0/5/5 (229)

3   0.564449 ( 0.0052)  sun.tuc.noao.edu.domain > ns.UU.NET.domain:
                        3+ A? ftp.uu.net. (28)
4   1.009476 ( 0.4450)  ns.UU.NET.domain > sun.tuc.noao.edu.domain:
                        3* 1/0/0 A ftp.UU.NET (44)
```

图14-14 执行host ftp.uu.net后的tcpdump 输出

这次在tcpdump中使用了新的选项。使用-w选项来收集进出UDP或TCP 53号端口的所有数据。将这些原始数据记录在一个文件中供以后处理，同时防止tcpdump试图调用名字解析器来显示与那个IP地址相对应的域名。执行查询后，终止tcpdump并使用-r选项再次运行它。它会读取含有原始数据的文件并产生正式的输出显示（如图14-14）。这个过程要花费几秒钟，因为tcpdump调用了它自己的名字解析器。

在tcpdump输出中要注意的第一点是标识符(identifier)是小整数（2和3）。这是因为我们关闭这个名字服务器，后又重新启动它来强制清空它的高速缓存。当名字服务器启动时，它将标识符初始化为1。

当键入查询，查找主机ftp.uu.net的IP地址，该名字服务器就同8个根名字服务器中的一个ns.nic.ddn.mil（第1行）取得联系。这是以前见到的正常的 A类型查询，但要注意的是它的期望递归表示没有说明（如果该标志被设置，在标识符 2的后边会跟着一个加号）。在以前的例子中，经常看到名字解析器设置期望递归标志，但这里的名字服务器在与某个根服务器联系时没有设置这个标志。这是因为不应该向根名字服务器发出期望递归的查询，它们仅用来寻找其他授权名字服务器的地址。

第2行显示返回的响应中没有回答资源记录，而包含 5个授权资源记录和5个附加信息资源记录。标识符2后的减号表示期望递归标志（RA）没有被设置。即使我们要求进行递归查询，这个根名字服务器也不会回答期望递归查询。

尽管tcpdump没有显示返回的10个资源记录，我们也能执行host命令来查看高速缓存的内容：

```
sun % host -v ftp.uu.net
Query about ftp.uu.net for record types A
Trying ftp.uu.net ...
Query done, 1 answer, status: no error
The following answer is not authoritative:
ftp.uu.net              19109   IN      A       192.48.96.9
Authoritative nameservers:
UU.NET                  170308  IN      NS      NS.UU.NET
UU.NET                  170308  IN      NS      UUNET.UU.NET
UU.NET                  170308  IN      NS      UUCP-GW-1.PA.DEC.COM
UU.NET                  170308  IN      NS      UUCP-GW-2.PA.DEC.COM
UU.NET                  170308  IN      NS      NS.EU.NET
Additional information:
NS.UU.NET               170347  IN      A       137.39.1.3
UUNET.UU.NET            170347  IN      A       192.48.96.2
UUCP-GW-1.PA.DEC.COM    170347  IN      A       16.1.0.18
UUCP-GW-2.PA.DEC.COM    170347  IN      A       16.1.0.19
NS.EU.NET               170347  IN      A       192.16.202.11
```

这次采用-v选项查看的不仅仅只是A记录。它显示出对于域uu.net有5个授权名字服务器，而由根名字服务器返回的5个附加信息资源记录中含有这5个名字服务器的IP地址。这避免了在查找其中的某个名字服务器的地址时，无需再次与根名字服务器联系。这是DNS中的另一个实现优化。

host命令指出这个回答不是授权的，这是因为这个回答来自名字服务器的高速缓存，而不是来自授权名字服务器。

回到图14-14中的第3行，我们的名字服务器与第一个授权名字服务器（ns.uu.net）询问同一个问题：ftp.uu.net的IP地址？这次我们的服务器设置了期望递归标志。返回的应答（第4行）包含一个回答资源记录。

而后我们再次执行host命令，询问相同的名字：

```
sun % host ftp.uu.net
ftp.uu.net                A           192.48.96.9
```

这时tcpdump没有输出，这正是我们所期望的，因为由host命令返回的回答来自于名字服务器的高速缓存。

再次执行host命令，查找ftp.ee.lbl.gov的地址：

```
sun % host ftp.ee.lbl.gov
ftp.ee.lbl.gov          CNAME      ee.lbl.gov
ee.lbl.gov              A          128.3.112.20
```

图14-15显示了这时的tcpdump输出。

```
1  18.664971 (17.6555)   sun.tuc.noao.edu.domain > c.nyser.net.domain:
                         4 A? ftp.ee.lbl.gov. (32)
2  19.429412 ( 0.7644)   c.nyser.net.domain > sun.tuc.noao.edu.domain:
                         4 0/4/4 (188)
3  19.432271 ( 0.0029)   sun.tuc.noao.edu.domain > ns1.lbl.gov.domain:
                         5+ A? ftp.ee.lbl.gov. (32)
4  19.909242 ( 0.4770)   ns1.lbl.gov.domain > sun.tuc.noao.edu.domain:
                         5* 2/0/0 CNAME ee.lbl.gov. (72)
```

图14-15 对ftp.ee.lbl.gov 主机的tcpdump 输出

这时第1行显示我们的服务器与另一个根名字服务器(c.nyser.net)联系。一个名字服务器通常轮询不同的根名字服务器来获得往返时间估计，然后选择往返时间最小的服务器。

既然我们的服务器向一个根服务器发出查询，那么期望递归标志不应被设置。正如我们在图14-14中所看到的该名字服务器并不清除期望递归标志（即便这样，一个名字服务器还是不应该向一个根名字服务器发出期望递归的查询）。

在第2行返回的响应中不包含回答资源记录，但含有4个授权记录和4个附加信息资源记录。正如我们所猜测的那样，4个授权资源记录是供主机ftp.ee.lbl.gov进行域名服务的名字服务器名，其他4个记录则是这4个服务器的IP地址。

第3行是向名字服务器nsl.lbl.gov（第2行中返回的4个名字服务器中的第一个）发出的查询请求。它的期望递归标志是被设置的。

第4行返回的响应和以往的响应不同。返回了两个回答资源记录，tcpdump指出其中的第一个是CNAME资源记录。ftp.ee.lbl.gov的规范名称是ee.lbl.gov。

这是CNAME记录常见的用法。LBL的FTP站点的名字通常是以ftp开始的，但它可能不时地从一个主机移到另一个主机。用户只需要知道ftp.ee.lbl.gov，必要时DNS会用它的规范名进行替换。

记得我们在运行host程序时，它显示了规范域名的CNAME和IP地址。这是因为响应（图14-15中的第4行）中含有两个回答资源记录，第一个是CNAME，而第二个是A记录。如果A记录没有随CNAME记录返回，我们的服务器将发出另一个查询请求，询问ee.lbl.gov的IP地址。这是另一个DNS的实现优化——在一个响应中同时返回一个规范域名的CNAME记录和A记录。

14.8 用UDP还是用TCP

注意到DNS名字服务器使用的熟知端口号无论对UDP还是TCP都是53。这意味着DNS均支持UDP和TCP访问，但我们使用tcpdump观察的所有例子都是采用UDP。那么这两种协议都在什么情况下采用以及采用的理由都是什么呢？

当名字解析器发出一个查询请求，并且返回响应中的TC（删减标志）比特被设置为1时，它就意味着响应的长度超过了512个字节，而仅返回前512个字节。在遇到这种情况时，名字解析器通常使用TCP重发原来的查询请求，它将允许返回的响应超过512个字节（回想在11.10节讨论的UDP数据报的最大长度）。既然TCP能将用户的数据流分为一些报文段，它就能用多个报文段来传送任意长度的用户数据。

此外，当一个域的辅助名字服务器在启动时，将从该域的主名字服务器执行区域传送。我们也说过辅助服务器将定时（通常是3小时）向主服务器进行查询以便了解主服务器数据是否发生变动。如果有变动，将执行一次区域传送。区域传送将使用TCP，因为这里传送的数据远比一个查询或响应多得多。

既然DNS主要使用UDP，无论是名字解析器还是名字服务器都必须自己处理超时和重传。此外，不像其他的使用UDP的Internet应用（TFTP、BOOTP和SNMP），大部分操作集中在局域网上，DNS查询和响应通常经过广域网。分组丢失率和往返时间的不确定性在广域网上比局域网上更大。这样对于DNS客户程序，一个好的重传和超时程序就显得更重要了。

14.9 另一个例子

让我们通过另一个例子将已经介绍的许多DNS特性作一个综合性回顾。先启动Rlogin客户程序，然后连接到一个位于其他域的Rlogin服务器。图14-16显示了发生的分组交换过程。下面发生的11个步骤都假定客户和服务器的高速缓存中没有任何信息。

1) 客户程序启动后，调用它的名字解析器函数将我们键入的主机名转换为一个IP地址。一个A类型的查询请求被送往一个根服务器。

2) 由根服务器返回的响应中包含为该服务器所在域服务的名字服务器名。

3) 客户端的名字解析器将向该服务器的名字服务器重发上述A类型查询，这个查询通常是将期望递归标志设置为1。

4) 返回的应答中包含Rlogin服务器的IP地址。

5) Rlogin客户和Rlogin服务器建立一个TCP连接（第18章将提供该步骤的细节）。客户和服务器的TCP模块间将交换3个分组。

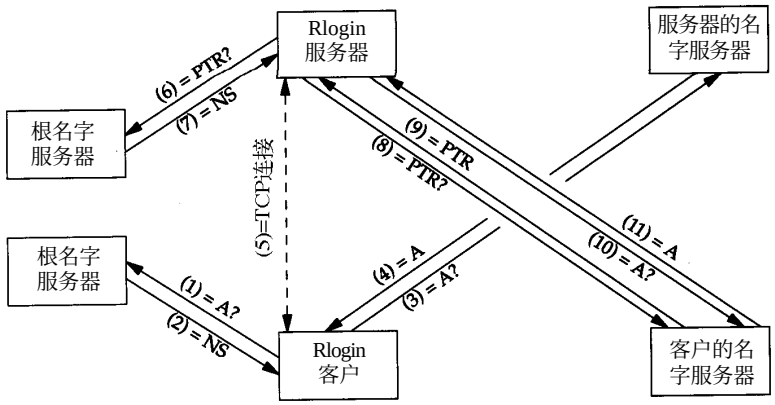


图14-16 启动Rlogin客户和服务器的分组交换过程

6) Rlogin服务器收到来自客户的连接请求后，调用它的名字解析器通过 TCP连接请求中的IP地址获得客户主机名。这是一个 PTR查询请求，由一个根名字服务器处理。这个根名字服务器可以不同于步骤1中客户使用的根名字服务器。

7) 这个根名字服务器的响应中含有为客户的 `in-addr.arpa`域的名字服务器。

8) 服务器上的名字解析器将向客户的名字服务器重传上述 PTR查询。

9) 返回的PTR应答中含有客户主机的FQDN。

10) 服务器的名字解析器向客户的名字服务器发送一个 A类型查询请求，查找前一步返回的名字对应的IP地址。这可能由服务器中的`gethostbyaddr`函数自动完成，正如我们在14.5节中介绍的那样，否则 Rlogin服务器将完成这一步。此外，客户的名字服务器常常就是客户的`in-addr.arpa`名字服务器，但这不是必需的。

11) 从客户的名字服务器返回的响应含有客户主机的 A记录。Rlogin服务器将客户的TCP连接请求中的IP地址与A记录作比较。

高速缓存将减少这个图中交换的分组数目。

14.10 小结

DNS是任何与Internet相连主机必不可少的一部分，同时它也广泛用于专用的互联网。层次树是组成DNS域名空间的基本组织形式。

应用程序通过名字解析器将一个主机名转换为一个 IP地址，也可将一个IP地址转换为与之对应的主机名。名字解析器将向一个本地名字服务器发出查询请求，这个名字服务器可能通过某个根名字服务器或其他名字服务器来完成这个查询。

所有的DNS查询和响应都有相同的报文格式。这个报文格式中包含查询请求和可能的回答资源记录、授权资源记录和附加资源记录。通过许多例子了解了名字解析器的配置文件以及DNS的优化措施：指向域名的指针（减少报文的长度）、查询结果的高速缓存、`in-addr.arpa`域（查找IP地址对应的域名）以及返回的附加资源记录（避免主机重发同一查询请求）。

习题

14.1 讨论一个DNS 名字解析器和一个DNS名字服务器作为客户程序、服务器或同时作为客

户和服务器的情况。

14.2 说明图14-12中构成响应的75个字节的含义。

14.3 在12.3节我们指出，一个既可接受点分十进制形式的 IP地址、也可接收主机名的应用程序，应先假定输入的是IP地址，如果失败，再假定是主机名。如果改变这个测试顺序会出现什么情况？

14.4 每个UDP数据报有一个相应的长度。一个接收 UDP数据报的进程将被告知这个长度。当名字解析器使用TCP而不是UDP来处理查询请求时，由于 TCP是没有任何记录标记的字节流，那么应用程序是如何知道有多少数据返回？注意在 DNS的报文首部（图14-3）中没有任何长度字段（提示：查阅 RFC 1035）

14.5 我们说一个名字服务器必须知道根名字服务器的 IP地址，这一信息可通过匿名 FTP获得。不幸的是当根名字服务器表发生变化时，并不是所有的系统管理员都会更新他们的 DNS配置文件（根名字服务表的确会发生变化，尽管不是经常的）你认为 DNS如何处理这个问题？

14.6 利用习题1.8指明的文件来确定谁应负责维护根名字服务器。名字服务器更新的频度是怎样的？

14.7 维护一个名字服务器和一个无状态的名字解析器高速缓存的问题分别是什么？

14.8 在图14-10的讨论中，我们指出名字服务器将对 A类型记录进行排序以便在公共网中的地址先出现。谁对A类型记录进行这种排序，是名字服务器还是名字解析器？

第15章 TFTP：简单文件传送协议

15.1 引言

TFTP(Trivial File Transfer Protocol)即简单文件传送协议，最初打算用于引导无盘系统(通常是工作站或X终端)。和将在第27章介绍的使用TCP的文件传送协议（FTP）不同，为了保持简单和短小，TFTP将使用UDP。TFTP的代码（和它所需要的UDP、IP和设备驱动程序）都能适合只读存储器。

本章对TFTP只作一般介绍，因为在下一章引导程序协议（Bootstrap Protocol）中还会遇到TFTP。在图5-1中，当从网络上引导sun主机时，也曾遇到过TFTP，sun主机通过RARP获得它的IP地址后，将发出一个TFTP请求。

RFC 1350 [Sollins 1992]是第2版TFTP的正式规范。第12章 [Stevens 1990] 提供了实现TFTP客户和服务器的全部源代码，并介绍了一些使用TFTP的编程技术。

15.2 协议

在开始工作时，TFTP的客户与服务器交换信息，客户发送一个读请求或写请求给服务器。在一个无盘系统进行系统引导的正常情况下，第一个请求是读请求（RRQ)。图15-1显示了5种TFTP报文格式（操作码为1和2的报文使用相同的格式)。

TFTP报文的头两个字节表示操作码。对于读请求和写请求（WRQ)，文件名字段说明客户要读或写的位于服务器上的文件。这个文件字段以0字节作为结束（见图15-1)。模式字段是一个ASCII码串netascii或octet（可大小写任意组合)，同样以0字节结束。netascii表示数据是以成行的ASCII码字符组成，以两个字节——回车字符后跟换行字符（称为CR/LF)作为行结束符。这两个行结束字符在这种格式和本地主机使用的行定界符之间进行转化。octet则将数据看作8 bit一组的字节流而不作任何解释。

每个数据分组包含一个块编号字段，它以后要在确认分组中使用。以读一个文件作为例子，TFTP客户需要发送一个读请求说明要读的文件名和文件模式 (mode)。如果这个文件能被这个客户读取，TFTP服务器就返回一个块编号为1的数据分组。TFTP客户又发送一个块编号为1的ACK。TFTP服务器随后发送块编号为2的数据。TFTP客户发回块编号为2的ACK。重复这个过程直到这个文件传送完。除了最后一个数据分组可含有不足512字节的数据，其他每个数据分组均含有512字节的数据。当TFTP客户收到一个不足512字节的数据分组，就知道它收到最后一个数据分组。

在写请求的情况下，TFTP 客户发送WRQ指明文件名和模式。如果该文件能被 该客户写，TFTP 服务器就返回块编号为0的ACK包。该客户就将文件的头512字节以块编号为1发出。服务器则返回块编号为1的ACK。

这种类型的数据传输称为停止等待协议。它只用在一些简单的协议如 TFTP中。在20.3节中将看到TCP提供了不同形式的确认，能提供更高的系统吞吐量。 TFTP的优点在于实现的简

单而不是高的系统吞吐量。

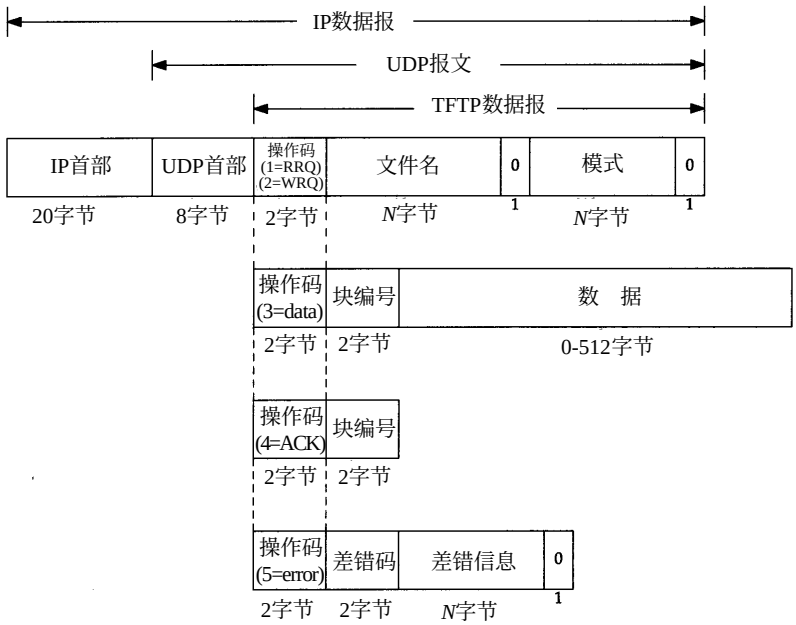


图15-1 5种TFTP报文格式

最后一种TFTP报文类型是差错报文，它的操作码为 5。它用于服务器不能处理读请求或写请求的情况。在文件传输过程中的读和写差错也会导致传送这种报文，接着停止传输。差错编号字段给出一个数字的差错码，跟着是一个 ASCII表示的差错报文字段，可能包含额外的操作系统说明的信息。

既然TFTP使用不可靠的UDP，TFTP就必须处理分组丢失和分组重复。分组丢失可通过发送方的超时与重传机制解决（注意存在一种称为“魔术新手综合症 (sorcerer’s apprentice syndrome)”的潜在问题，如果双方都超时与重传，就可能出现这个问题。 12.2 节 [Stevens 1990] 介绍了这个问题是如何发生的)。和许多UDP应用程序一样， TFTP报文中没有检验和，它假定任何数据差错都将被UDP的检验和检测到（参见11.3节）。

15.3 一个例子

让我们通过观察协议的工作情况来了解 TFTP。在bsdi主机上运行TFTP 客户程序，并从主机svr4读取一个文本文件：

```
bsdi % tftp svr4                          启动TFTP客户进程
tftp> get test1.c                         从服务器读取文件
Received 962 bytes in 0.3 seconds
tftp> quit                                结束

bsdi % ls -l test1.c                      查看我们读取的文件大小
-rw-r--r--  1 rstevens  staff  914 Mar 20 11:41 test1.c

bsdi % wc -l test1.c                      文件行数？
      48 test1.c
```

最先引起我们注意的是在Unix系统下接收的文件长度是914字节，而TFTP则传送了962个字节。使用wc程序我们看到文件共有 48行，因此48个Unix的换行符被转化成 48个CR/CF对，

因为默认情况下TFTP使用netascii模式传送。

图15-2显示了发生的分组交换过程。

```
1  0.0                  bsdi.1106 > svr4.tftp: 19 RRQ "test1.c"

2  0.287080 (0.2871)    svr4.1077 > bsdi.1106: udp 516
3  0.291178 (0.0041)    bsdi.1106 > svr4.1077: udp 4

4  0.299446 (0.0083)    svr4.1077 > bsdi.1106: udp 454
5  0.312320 (0.0129)    bsdi.1106 > svr4.1077: udp 4
```

图15-2 使用TFTP传输一个文件的分组交换过程

第1行显示了客户向服务器发送的读请求。由于目的 UDP端口是TFTP熟知端口（69），tcpdump将解释TFTP分组，并显示RRQ和文件名。19字节的UDP数据包括2字节的操作码，7字节的的文件名，1字节的0，8字节的netascii模式以及另1字节的0结束。

下一个分组由服务器发回（第2行），共包含516字节：2字节的操作码，2字节的数据块号和512字节的数据。第3行是这个数据块的确认，它包括2字节的操作码和2字节的数据块号。

最后的数据分组（第4行）包含450字节的数据。这450字节的数据加上第2行的512字节的数据就是向该客户传送的962字节的数据。注意tcpdump仅在第1行解释TFTP报文，而在2～5行都不显示任何 TFTP协议信息。这是因为服务器进程的端口在第 1行和第2行发生了变化。TFTP协议需要客户进程向服务器进程的UDP熟知端口（69）发送第一个分组（RRQ或WRQ）。之后服务器进程便向服务器主机申请一个尚未使用的端口（ 1077，见图15-2），服务器进程使用这个端口来进行请求客户进程与服务器进程间的其他数据交换。客户进程的端口号（在这个例子中为1106）没有变化。tcpdump无法知道主机srv4上的1077端口是一个TFTP服务器进程。

服务器进程端口变化的原因是服务器进程不能占用这个熟知端口来完成需一些时间的文件传输（可能是几十秒甚至数分钟）。相反，在传输当前文件的过程中，这个熟知端口要留出来供其他的TFTP客户进程发送它们的请求。

回顾图10-6，当RIP服务器向客户发送的数据超过 512字节，两个UDP数据报都使用服务器的熟知端口。在那个例子中，即使服务器进程必须写多个数据报以便将所有数据发回，服务器进程也是先写一个，再写一个，它们都使用它的熟知端口。然而， TFTP协议与它不同，因为客户与服务器间的连接需要持续一个较长的时间（可能是数秒或数分钟）。如果一个服务器进程使用熟知端口来进行文件传输，那么在文件传输期间，它要么拒绝任何来自其他客户的请求，要么一个服务器进程在同一端口（ 69）同时对多个客户进程进行多个文件传输。最简单的办法是让服务器进程在收到 RRQ或WRQ后，改用新的端口。当然，客户进程在收到第一个数据分组（图 15-2的第2行）后必须探测到这个新的端口，并将之后的所有确认（第 3行和第5行）发送到那个新的端口。

在16.3节我们将看到当X终端在进行系统引导时将使用 TFTP。

15.4 安全性

注意在TFTP分组（图15-1）中并不提供用户名和口令。这是 TFTP的一个特征（即“ 安全漏洞 ”）。由于TFTP是设计用于系统引导进程，它不可能提供用户名和口令。

TFTP的这一特性被许多解密高手用于获取 Unix口令文件的复制，然后来猜测用户口令。

为防止这种类型的访问，目前大多数 TFTP服务器提供了一个选项来限制只能访问特定目录下的文件（Unix系统中通常是`/tftpboot`）。这个目录中只包含无盘系统进行系统引导时所需的文件。

对其他的安全性，Unix系统下的TFTP服务器通常将它的用户 ID和组ID设置为不会赋给任何真正用户的值。这只允许访问具有读或写属性的文件。

15.5 小结

TFTP是一个简单的协议，适合于只读存储器，仅用于无盘系统进行系统引导。它只使用几种报文格式，是一种停止等待协议。

为了允许多个客户端同时进行系统引导， TFTP服务器必须提供一定形式的并发。因为UDP在一个客户与一个服务器之间并不提供唯一连接（ TCP也一样），TFTP服务器通过为每个客户提供一个新的 UDP端口来提供并发。这允许不同的客户输入数据报，然后由服务器中的UDP模块根据目的端口号进行区分，而不是由服务器本身来进行区分。

TFTP协议没有提供安全特性。大多数执行指望 TFTP服务器的系统管理员来限制客户的访问，只允许它们访问引导所必须的文件。

第27章介绍的文件传输协议（FTP）是设计用于一般目的的、高吞吐量的文件传输。

习题

15.1 阅读Host Requirements RFC，了解如果一个TFTP 服务器收到的请求的目的IP地址是一个广播地址，它将做什么。

15.2 当TFTP块号由65535跳回到0时，你认为会发生什么？ RFC 1350提到了如何处理这一问题吗？

15.3 TFTP发送方采用超时重发来处理分组丢失。当 TFTP作为引导进程的一部分时，这种方法对TFTP的使用有何影响？

15.4 使用TFTP时，影响传输文件所需时间的限制性因素是什么？

第16章 BOOTP：引导程序协议

16.1 引言

在第5章我们介绍了一个无盘系统，它在不知道自身 IP地址的情况下，在进行系统引导时能够通过RARP来获取它的IP地址。然而使用RARP有两个问题：（1）IP地址是返回的唯一结果；（2）既然RARP使用链路层广播，RARP请求就不会被路由器转发（迫使每个实际网络设置一个RARP 服务器）。本章将介绍一种用于无盘系统进行系统引导的替代方法，又称为引导程序协议，或BOOTP。

BOOTP使用UDP，且通常需与 TFTP（参见第 15章）协同工作。 RFC 951 [Croft and Gilmore 1985]是BOOTP的正式规范，RFC 1542 [Wimer 1993]则对它作了说明。

16.2 BOOTP 的分组格式

BOOTP 请求和应答均被封装在UDP数据报中，如图16-1所示。

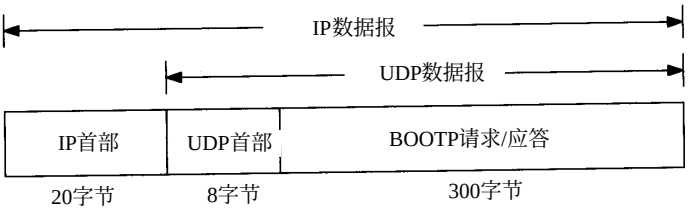


图16-1 BOOTP 请求和应答封装在一个UDP数据报内

图16-2显示了长度为300字节的BOOTP请求和应答的格式。

“操作码”字段为1表示请求，为2表示应答。硬件类型字段为 1表示10 Mb/s的以太网，这和ARP请求或应答（图 4-3）中同名字段表示的含义相同。类似地，对于以太网，硬件地址长度字段为6字节。

“跳数”字段由客户设置为0，但也能被一个代理服务器设置（参见 16.5节）。

“事务标识”字段是一个由客户设置并由服务器返回的 32 bit整数。客户用它对请求和应答进行匹配。对每个请求，客户应该将该字段设置为一个随机数。

客户开始进行引导时，将“秒数”字段设置为一个时间值。服务器能够看到这个时间值，备用服务器在等待时间超过这个时间值后才会响应客户的请求，这意味着主服务器没有启动。

如果该客户已经知道自身的 IP地址，它将写入“客户 IP地址”字段。否则，它将该字段设置为0。对于后面这种情况，服务器用该客户的 IP地址写入“你的IP地址”字段。“服务器IP地址”字段则由服务器填写。如果使用了某个代理服务器（见 16.5节），则该代理服务器就填写“网关IP地址”字段。

客户必须设置它的“客户硬件地址”字段。尽管这个值与以太网数据帧头中的值相同，UDP数据报中也设置这个字段，但任何接收这个数据报的用户进程能很容易地获得它（例如

一个BOOTP 服务器)。一个进程通过查看UDP数据报来确定以太网帧首部中的该字段通常是很困难的（或者说是不可能的)。

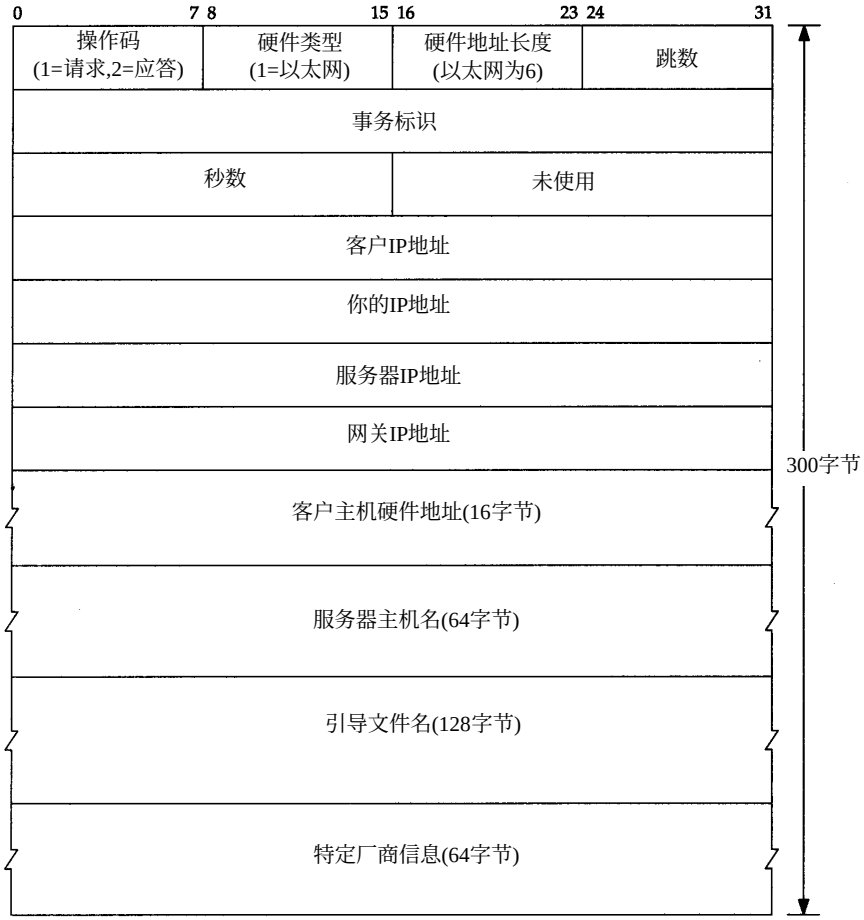


图16-2 BOOTP请求和应答的格式

“服务器主机名”字段是一个空值终止串，由服务器填写。服务器还将在“引导文件名字段”填入包括用于系统引导的文件名及其所在位置的路径全名。

“特定厂商区域”字段用于对BOOTP进行不同的扩展。16.6节将介绍这些扩展中的一些。

当一个客户使用BOOTP（操作码为1）进行系统引导时，引导请求通常是采用链路层广播，IP首部中的目的IP地址为255.255.255.255（受限的广播，12.2节)。源IP地址通常是0.0.0.0，因为此时客户还不知道它本身的IP地址。回顾图3-9，在系统进行自引导时，0.0.0.0是一个有效的IP地址。

端口号

BOOTP有两个熟知端口：BOOTP 服务器为67，BOOTP 客户为68。这意味着BOOTP 客户不会选择未用的临时端口，而只用端口 68。选择两个端口而不是仅选择一个端口为 BOOTP 服务器用的原因是：服务器的应答可以进行广播（但通常是不用广播的)。

如果服务器的应答是通过广播传送的，同时客户又选择未用的临时端口，那么这些广播

也能被其他的主机中碰巧使用相同临时端口的应用进程接收到。因此，采用随机端口（即临时端口）对广播来说是一个不好的选择。

如果客户也使用服务器的知名端口（67）作为它的端口，那么网络内的所有服务器会被唤醒来查看每个广播应答（如果所有的服务器都被唤醒，它们将检查操作码，如果是一个应答而不是请求，就不作处理）。因此可以让所有的客户使用与服务器知名端口不同的同一知名端口。

如果多个客户同时进行系统引导，并且服务器广播所有应答，这样每个客户都会收到其他客户的应答。客户可以通过BOOTP首部中的事务标识字段来确认应答是否与请求匹配，或者可以通过检查返回的客户硬件地址加以区分。

16.3 一个例子

让我们看一个用BOOTP引导一个X终端的例子。图16-3显示了tcpdump的输出结果（例中客户名为proteus，服务器名为mercury。这个tcpdump的输出是在不同的网络上获得的，这个应用程序是其他例子中一直使用的）。

```
 1   0.0                 0.0.0.0.68 > 255.255.255.255.bootp:
                         secs:100 ether 0:0:a7:0:62:7c
 2   0.355446 (0.3554)   mercury.bootp > proteus.68: secs:100 Y:proteus
                         S:mercury G:mercury ether 0:0:a7:0:62:7c
                         file "/local/var/bootfiles/Xncd19r"
 3   0.355447 (0.0000)   arp who-has proteus tell 0.0.0.0
 4   0.851508 (0.4961)   arp who-has proteus tell 0.0.0.0
 5   1.371070 (0.5196)   arp who-has proteus tell proteus
 6   1.863226 (0.4922)   proteus.68 > 255.255.255.255.bootp:
                         secs:100 ether 0:0:a7:0:62:7c
 7   1.871038 (0.0078)   mercury.bootp > proteus.68: secs:100 Y:proteus
                         S:mercury G:mercury ether 0:0:a7:0:62:7c
                         file "/local/var/bootfiles/Xncd19r"
 8   3.871038 (2.0000)   proteus.68 > 255.255.255.255.bootp:
                         secs:100 ether 0:0:a7:0:62:7c
 9   3.878850 (0.0078)   mercury.bootp > proteus.68: secs:100 Y:proteus
                         S:mercury G:mercury ether 0:0:a7:0:62:7c
                         file "/local/var/bootfiles/Xncd19r"
10   5.925786 (2.0469)   arp who-has mercury tell proteus
11   5.929692 (0.0039)   arp reply mercury is-at 8:0:2b:28:eb:1d
12   5.929694 (0.0000)   proteus.tftp > mercury.tftp: 37 RRQ
                         "/local/var/bootfiles/Xncd19r"
13   5.996094 (0.0664)   mercury.2352 > proteus.tftp: 516 DATA block 1
14   6.000000 (0.0039)   proteus.tftp > mercury.2352: 4 ACK
                                  这里删除了许多行
15  14.980472 (8.9805)   mercury.2352 > proteus.tftp: 516 DATA block 2463
16  14.984376 (0.0039)   proteus.tftp > mercury.2352: 4 ACK
17  14.984377 (0.0000)   mercury.2352 > proteus.tftp: 228 DATA block 2464
18  14.984378 (0.0000)   proteus.tftp > mercury.2352: 4 ACK
```

图16-3 用BOOTP引导一个X终端的例子

在第1行中，我们看到客户请求来自0.0.0.0.68，发送目的站是255.255.255.255.67。该客户已经填写的字段是秒数和自身的以太网地址。我们看到客户通常将秒数设置为100。tcpdump没有显示跳数和事务标识，因为它们均为0(事务标识为0表示该客户忽略这个字段，

因为如果打算对返回响应进行验证，它将把这个字段设置为一个随机数值)。

第2行是服务器返回的应答。由服务器填写的字段是该客户的 IP地址（tcpdump显示为名字proteus）、服务器的 IP地址（显示为名字 mercury）、网关的 IP地址（显示为名字 mercury）和引导文件名。

在收到BOOTP应答后，该客户立即发送一个ARP请求来了解网络中其他主机是否有IP地址。跟在who-has后的名字proteus对应目的IP地址（图4-3），发送者的IP地址被设置为0.0.0.0。它在0.5秒后再发一个相同的ARP请求，之后再过0.5秒又发一个。在第3个ARP请求（第5行）中，它将发送者的IP地址改变为它自己的IP地址。这是一个没有意义的ARP请求（见4.7节）。

第6行显示该客户在等待另一个 0.5秒后，广播另一个BOOTP请求。这个请求与第 1行的唯一不同是此时客户将它的IP地址写入IP首部中。它收到来自同一个服务器的相同应答（第7行）。该客户在等待 2秒后，又广播一个 BOOTP请求（第 8行），同样收到来自同一服务器的相同应答。

该客户等待2秒后，向它的服务器 mercury发送一个ARP请求（第10行）。收到这个ARP应答后，它立即发送一个 TFTP读请求，请求读取它的引导文件（第 12行）。文件传送过程包括2464个TFTP数据分组和确认，传送的数据量为 512×2463 + 224 = 1 261 280字节。这将操作系统调入X终端。我们已在图16-3中删除了大多数TFTP行。

当和图15-2比较TFTP的数据交换过程时，要注意的是这儿的客户在整个传输过程中使用TFTP的知名端口（69）。既然通信双方中的一方使用了端口 69，tcpdump就知道这些分组是TFTP报文，因此它能用TFTP协议来解释每个分组。这就是为什么图 16-3能指明哪些包含有数据，哪些包含有确认，以及每个分组的块编号。在图 15-2中我们并不能获得这些额外的信息，因为通信双方均没有使用 TFTP的知名端口进行数据传送。由于 TFTP 服务器作为一个多用户系统，且使用TFTP的知名端口，因此通常 TFTP客户不能使用那个端口。但这里的系统处于正被引导的过程中，无法提供一个 TFTP服务器，因此允许该客户在传输期间使用 TFTP的知名端口。这也暗示在mercury上的TFTP服务器并不关心客户的端口号是什么 —— 它只将数据传送到客户的端口上，而不管发生了什么。

从图16-3可以看出在9秒内共传送了1 261 280字节。数据速率大约为140 000 bps。这比大多数以FTP文件传送形式访问一个以太网要慢，但对于一个简单的停止等待协议如 TFTP来说已经很好了。

X终端系统引导后，还需使用 TFTP传送终端的字体文件、某些 DNS名字服务器查询，然后进行X协议的初始化。图 16-3中的所有步骤大概需要 15秒钟，其余的步骤需要 6秒钟，这样无盘X终端系统引导的总时间是21秒。

16.4 BOOTP服务器的设计

BOOTP客户通常固化在无盘系统只读存储器中，因此了解 BOOTP 服务器的实现将更有意义。

首先，BOOTP 服务器将从它的熟知端口（67）读取UDP数据报。这没有特别的地方。它不同于RARP 服务器 （5.4节），它必须读取类型字段为 “ RARP请求 ” 的以太网帧。 BOOTP协议通过将客户的硬件地址放入 BOOTP分组中，使得服务器很容易获取客户的硬件地址 (图16-2)。

这里出现了一个有趣的问题：TFTP 服务器如何能将一个响应直接送回 BOOTP 客户？这个响应是一个UDP数据报，而服务器知道该客户的 IP地址（可能通过读取服务器上的配置文件）。但如果这个客户向那个 IP地址发送一个UDP数据报（正常情况下会处理 UDP的输出），BOOTP 服务器的主机就可能向那个IP地址发送一个ARP请求。但这个客户不能响应这个 ARP请求，因为它还不知道它自己的IP地址！（这就是在RFC951中被称作“鸡和蛋”的问题。）

有两种解决办法：第一种，通常被Unix 服务器采用，是服务器发一个ioctl(2)请求给内核，为该客户在ARP高速缓存中设置一个条目（这就是命令 `arp -s`所做的工作，见4.8节）。服务器能一直这么做直到它知道客户的硬件地址和 IP地址。这意味着当服务器发送 UDP数据报（即BOOTP应答）时，服务器的ARP将在ARP 高速缓存中找到该客户的IP地址。

另一种可选的解决办法是服务器广播这个 BOOTP应答而不直接将应答发回该客户。既然通常期望网络广播越少越好，因此这种解决方案应该只在服务器无法在它的 ARP 高速缓存设置一个条目的情况下使用。通常只有拥有超级用户权限才能在 ARP高速缓存设置一个条目，如果没有这种权限就只能广播BOOTP应答。

16.5 BOOTP穿越路由器

我们在5.4节中提到RARP的一个缺点就是它使用链路层广播，这种广播通常不会由路由器转发。这就需要在每个物理网络内设置一个 RARP 服务器。如果路由器支持 BOOTP协议，那么BOOTP能够由路由器转发（绝大多数路由器厂商的产品都支持这个功能）。

这个功能主要用于无盘路由器，因为如果在磁盘的多用户系统被用作路由器，它就能够自己运行BOOTP 服务器。此外，常用的 Unix BOOTP服务器（附录F）支持这种中继模式（relay mode）。但如果在这个物理网络内运行一个BOOTP 服务器，通常没有必要将BOOTP请求转发到在另外网络中的另一个服务器。

研究一下当路由器（也称作“ BOOTP 中继代理”）在服务器的熟知端口（ 67）接收到BOOTP请求时将会发生什么。当收到一个 BOOTP请求时，中继代理将它的 IP地址填入收到BOOTP请求中的“网关IP地址字段”，然后将该请求发送到真正的 BOOTP服务器（由中继代理填入网关字段的地址是收到的 BOOTP请求接口的IP地址）。该代理中继还将跳数字段值加 1（这是为防止请求被无限地在网络内转发。 RFC 951认为如果跳数值到达3就可以丢弃该请求）。既然发出的请求是一个单播的数据报（与发起的客户的请求是广播的相反），它能按照一定的路由通过其他的路由器到达真正的 BOOTP服务器。真正的BOOTP服务器收到这个请求后，产生BOOTP应答，并将它发回中继代理，而不是请求的客户。既然请求网关字段不为零，真正的BOOTP服务器知道这个请求是经过转发的。中继代理收到应答后将它发给请求的客户。

16.6 特定厂商信息

在图16-2中我们看到了64字节的“特定厂商区域”。RFC 1533 [Alexander and Droms 1993] 定义了这个区域的格式。这个区域含有服务器返回客户的可选信息。

如果有信息要提供，这个区域的前 4个字节被设置为IP地址99.130.83.99。这可称作魔术甜饼(magic cookie)，表示该区域内包含信息。

这个区域的其余部分是一个条目表。每个条目的开始是 1字节标志字段。其中的两个条目仅有标志字段：标志为0的条目作为填充字节（为使后面的条目有更好的字节边界），标志为

255的条目表示结尾条目。第一个结尾条目后剩余的字节都应设置为这个数值（ 255）。

除了这两个1字节的条目，其他的条目还包含一个单字节的长度字段，后面是相应的信息。图16-4显示了厂商说明区域中一些条目的格式。

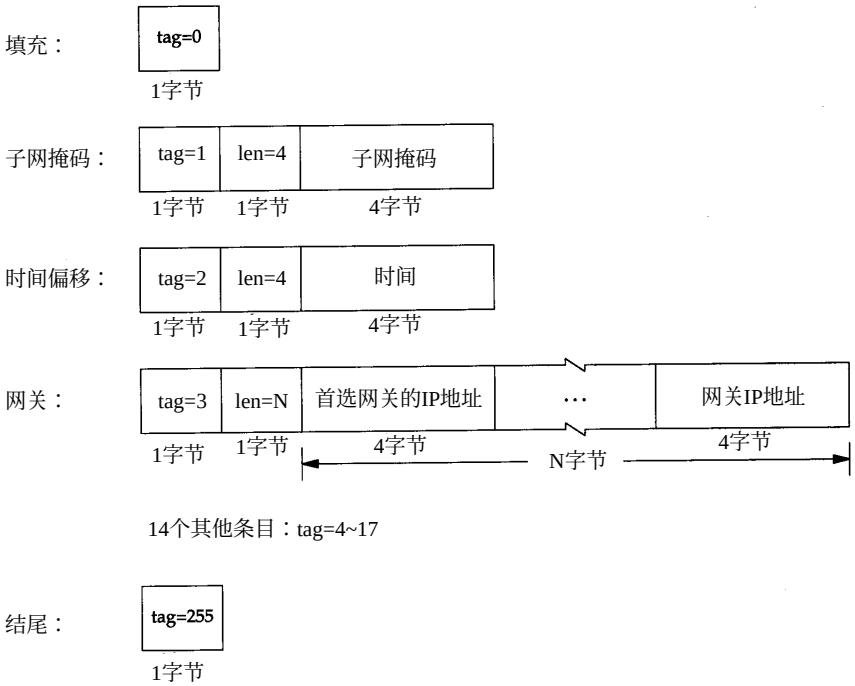


图16-4 厂商说明区域中一些条目的格式

子网掩码条目和时间值条目都是定长条目，因为它们的值总是占 4个字节。时间偏移值是从1900年1月1日0时以来的秒数（UTC）。

网关条目是变长条目。长度通常是4 的倍数，这个值是一个或多个供客户使用的网关（路由器）的IP地址。返回的第一个必须是首选的网关。

RFC 1533还定义了其他 14个条目。其中最重要的可能是 DNS名字服务器的IP地址条目，条目的志为6。其他的条目包括打印服务器、时间服务器等的 IP地址。详细情况可参考 RFC文档。

回到在图16-3中的例子，我们从未看到客户广播一个 ICMP地址掩码请求（6.3节）来获取它的子网掩码。尽管 tcpdump不能显示出来，但我们可认为客户所在网络的子网掩码在返回的BOOTP应答的厂商说明区域内。

Host Requirements RFC文档推荐一个系统使用BOOTP来获悉它的子网掩码，而不是采用ICMP。

厂商说明区域的大小被限制为 64字节。这对某些应用是个约束。一个新的称为动态主机配置协议DHCP（Dynamic Host Configuration Protocol）已经出现，但它不是替代BOOTP的。DHCP将这个区域的长度扩展到312字节，它在RFC 1541 [Droms 1993] 中定义。

16.7 小结

BOOTP使用UDP，它为引导无盘系统获得它的 IP地址提供了除RARP外的另外一种选择。

BOOTP还能返回其他的信息，如路由器的IP地址、客户的子网掩码和名字服务器的IP地址。

既然BOOTP用于系统引导过程，一个无盘系统需要下列协议才能在只读存储器中完成：BOOTP、TFTP、UDP、IP和一个局域网的驱动程序。

BOOTP服务器比RARP服务器更易于实现，因为BOOTP请求和应答是在UDP数据报中，而不是特殊的数据链路层帧。一个路由器还能作为真正BOOTP服务器的代理，向位于不同网络的真正BOOTP服务器转发客户的BOOTP请求。

习题

16.1 我们说BOOTP优于RARP的一个方面是BOOTP能穿越路由器，而RARP由于使用链路层广播则不能。在16.5节为使BOOTP穿越路由器，我们必须定义特殊的方式。如果在路由器中增加允许转发RARP请求的功能会发生什么？

16.2 我们说过，当有多个客户程序同时向一个服务器发出引导请求时，因为服务器要广播多个BOOTP应答，BOOTP客户就必须使用事务标识来使响应与请求相匹配。但在图16-3中，事务标识为0，表示这个客户不考虑事务标识。你认为这个客户将如何将这些响应与其请求匹配。

第17章 TCP：传输控制协议

17.1 引言

本章将介绍TCP为应用层提供的服务，以及TCP首部中的各个字段。随后的几章我们在了解TCP的工作过程中将对这些字段作详细介绍。

对TCP的介绍将由本章开始，并一直包括随后的 7章。第18章描述如何建立和终止一个TCP连接，第19和第20章将了解正常的数据传输过程，包括交互使用（远程登录）和批量数据传送（文件传输）。第21章提供TCP超时及重传的技术细节，第 22和第23章将介绍两种其他的定时器。最后，第24章概述TCP新的特性以及TCP的性能。

17.2 TCP的服务

尽管TCP和UDP都使用相同的网络层（IP），TCP却向应用层提供与UDP完全不同的服务。TCP提供一种面向连接的、可靠的字节流服务。

面向连接意味着两个使用 TCP的应用（通常是一个客户和一个服务器）在彼此交换数据之前必须先建立一个 TCP连接。这一过程与打电话很相似，先拨号振铃，等待对方摘机说“喂”，然后才说明是谁。在第 18章我们将看到一个TCP连接是如何建立的，以及当一方通信结束后如何断开连接。

在一个TCP连接中，仅有两方进行彼此通信。在第 12章介绍的广播和多播不能用于TCP。

TCP通过下列方式来提供可靠性：

- 应用数据被分割成TCP认为最适合发送的数据块。这和 UDP完全不同，应用程序产生的数据报长度将保持不变。由 TCP传递给IP的信息单位称为报文段或段（ segment）（参见图1-7）。在18.4节我们将看到TCP如何确定报文段的长度。
- 当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。在第 21章我们将了解TCP协议中自适应的超时及重传策略。
- 当TCP收到发自TCP连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒，这将在19.3节讨论。
- TCP将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错， TCP将丢弃这个报文段和不确认收到此报文段（希望发端超时并重发）。
- 既然TCP报文段作为IP数据报来传输，而IP数据报的到达可能会失序，因此 TCP报文段的到达也可能会失序。如果必要， TCP将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。
- 既然IP数据报会发生重复， TCP的接收端必须丢弃重复的数据。
- TCP还能提供流量控制。 TCP连接的每一方都有固定大小的缓冲空间。 TCP的接收端只

允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。

两个应用程序通过TCP连接交换8 bit字节构成的字节流。TCP不在字节流中插入记录标识符。我们将这称为字节流服务（byte stream service）。如果一方的应用程序先传10字节，又传20字节，再传50字节，连接的另一方将无法了解发方每次发送了多少字节。收方可以分4次接收这80个字节，每次接收20字节。一端将字节流放到TCP连接上，同样的字节流将出现在TCP连接的另一端。

另外，TCP对字节流的内容不作任何解释。TCP不知道传输的数据字节流是二进制数据，还是ASCII字符、EBCDIC字符或者其他类型数据。对字节流的解释由TCP连接双方的应用层解释。

这种对字节流的处理方式与Unix操作系统对文件的处理方式很相似。Unix的内核对一个应用读或写的内容不作任何解释，而是交给应用程序处理。对Unix的内核来说，它无法区分一个二进制文件与一个文本文件。

17.3 TCP的首部

TCP数据被封装在一个IP数据报中，如图17-1所示。

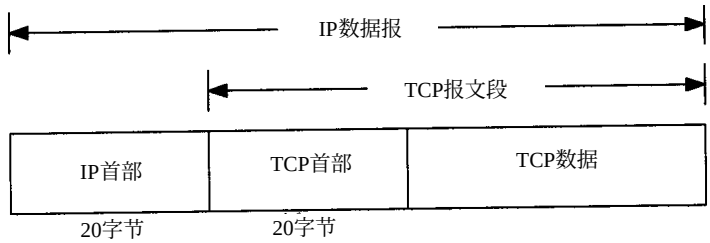


图17-1 TCP数据在IP数据报中的封装

图17-2显示TCP首部的数据格式。如果不计任选字段，它通常是20个字节。

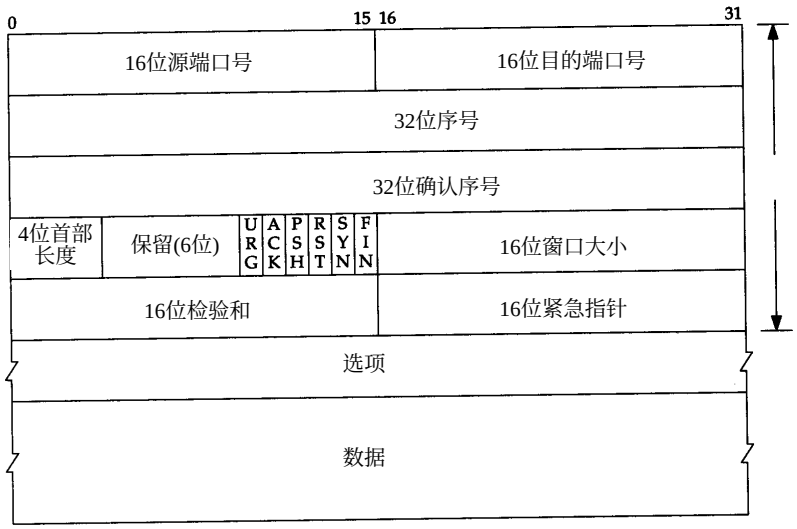


图17-2 TCP包首部

每个TCP段都包含源端和目的端的端口号，用于寻找发端和收端应用进程。这两个值加上IP首部中的源端IP地址和目的端IP地址唯一确定一个TCP连接。

有时，一个IP地址和一个端口号也称为一个插口（ socket）。这个术语出现在最早的 TCP规范（RFC793）中，后来它也作为表示伯克利版的编程接口（参见 1.15节）。插口对（socket pair）(包含客户IP地址、客户端口号、服务器 IP地址和服务器端口号的四元组)可唯一确定互联网络中每个TCP连接的双方。

序号用来标识从TCP发端向TCP收端发送的数据字节流，它表示在这个报文段中的的第一个数据字节。如果将字节流看作在两个应用程序间的单向流动，则 TCP用序号对每个字节进行计数。序号是32 bit的无符号数，序号到达$2^{32}-1$后又从0开始。

当建立一个新的连接时， SYN标志变1。序号字段包含由这个主机选择的该连接的初始序号ISN（Initial Sequence Number）。该主机要发送数据的第一个字节序号为这个 ISN加1，因为SYN标志消耗了一个序号（将在下章详细介绍如何建立和终止连接，届时我们将看到 FIN标志也要占用一个序号）。

既然每个传输的字节都被计数，确认序号包含发送确认的一端所期望收到的下一个序号。因此，确认序号应当是上次已成功收到数据字节序号加 1。只有ACK标志（下面介绍）为 1时确认序号字段才有效。

发送ACK无需任何代价，因为32 bit的确认序号字段和ACK标志一样，总是TCP首部的一部分。因此，我们看到一旦一个连接建立起来，这个字段总是被设置， ACK标志也总是被设置为1。

TCP为应用层提供全双工服务。这意味数据能在两个方向上独立地进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。

TCP可以表述为一个没有选择确认或否认的滑动窗口协议（滑动窗口协议用于数据传输将在20.3节介绍）。我们说TCP缺少选择确认是因为 TCP首部中的确认序号表示发方已成功收到字节，但还不包含确认序号所指的字节。当前还无法对数据流中选定的部分进行确认。例如，如果1～1024字节已经成功收到，下一报文段中包含序号从 2049～3072的字节，收端并不能确认这个新的报文段。它所能做的就是发回一个确认序号为 1025的ACK。它也无法对一个报文段进行否认。例如，如果收到包含 1025～2048字节的报文段，但它的检验和错， TCP接收端所能做的就是发回一个确认序号为 1025的ACK。在21.7节我们将看到重复的确认如何帮助确定分组已经丢失。

首部长度给出首部中 32 bit字的数目。需要这个值是因为任选字段的长度是可变的。这个字段占4 bit，因此TCP最多有60字节的首部。然而，没有任选字段，正常的长度是 20字节。

在TCP首部中有6个标志比特。它们中的多个可同时被设置为 1。我们在这儿简单介绍它们的用法，在随后的章节中有更详细的介绍。

URG	紧急指针（urgent pointer）有效（见20.8节）。
ACK	确认序号有效。
PSH	接收方应该尽快将这个报文段交给应用层。
RST	重建连接。
SYN	同步序号用来发起一个连接。这个标志和下一个标志将在第 18章介绍。
FIN	发端完成发送任务。

TCP的流量控制由连接的每一端通过声明的窗口大小来提供。窗口大小为字节数，起始于确认序号字段指明的值，这个值是接收端正期望接收的字节。窗口大小是一个 16 bit字段，因而窗口大小最大为65535字节。在24.4节我们将看到新的窗口刻度选项，它允许这个值按比例变化以提供更大的窗口。

检验和覆盖了整个的TCP报文段：TCP首部和TCP数据。这是一个强制性的字段，一定是由发端计算和存储，并由收端进行验证。 TCP检验和的计算和UDP检验和的计算相似，使用如11.3节所述的一个伪首部。

只有当URG标志置1时紧急指针才有效。紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。 TCP的紧急方式是发送端向另一端发送紧急数据的一种方式。我们将在20.8节介绍它。

最常见的可选字段是最长报文大小，又称为 MSS (Maximum Segment Size)。每个连接方通常都在通信的第一个报文段（为建立连接而设置 SYN标志的那个段）中指明这个选项。它指明本端所能接收的最大长度的报文段。我们将在 18.4节更详细地介绍MSS选项，TCP的其他选项中的一些将在第24章中介绍。

从图17-2中我们注意到TCP报文段中的数据部分是可选的。我们将在 18章中看到在一个连接建立和一个连接终止时，双方交换的报文段仅有 TCP首部。如果一方没有数据要发送，也使用没有任何数据的首部来确认收到的数据。在处理超时的许多情况中，也会发送不带任何数据的报文段。

17.4 小结

TCP提供了一种可靠的面向连接的字节流运输层服务。我们简单地介绍了 TCP首部中的各个字段，并在随后的几章里详细讨论它们。

TCP将用户数据打包构成报文段；它发送数据后启动一个定时器；另一端对收到的数据进行确认，对失序的数据重新排序，丢弃重复数据； TCP提供端到端的流量控制，并计算和验证一个强制性的端到端检验和。

许多流行的应用程序如Telnet、Rlogin、FTP和SMTP都使用TCP。

习题

17.1 我们已经介绍了以下几种分组格式： IP、ICMP、IGMP、UDP和TCP。每一种格式的首部中均包含一个检验和。对每种分组，说明检验和包括 IP数据报中的哪些部分，以及该检验和是强制的还是可选的。

17.2 为什么我们已经讨论的所有Internet协议（IP, ICMP, IGMP, UDP, TCP）收到有检验和错的分组都仅作丢弃处理？

17.3 TCP提供了一种字节流服务，而收发双方都不保持记录的边界。应用程序如何提供它们自己的记录标识？

17.4 为什么在TCP首部的开始便是源和目的的端口号？

17.5 为什么TCP首部有一个首部长度字段而UDP首部（图11-2）中却没有？

第18章 TCP连接的建立与终止

18.1 引言

TCP是一个面向连接的协议。无论哪一方向另一方发送数据之前，都必须先在双方之间建立一条连接。本章将详细讨论一个TCP连接是如何建立的以及通信结束后是如何终止的。

这种两端间连接的建立与无连接协议如UDP不同。我们在第11章看到一端使用UDP向另一端发送数据报时，无需任何预先的握手。

18.2 连接的建立与终止

为了了解一个TCP连接在建立及终止时发生了什么，我们在系统svr4上键入下列命令：

```
svr4 % telnet bsdi discard
Trying 140.252.13.35 ...
Connected to bsdi.
Escape character is '^]'.
^]
telnet> quit                                    键入Ctrl和右括号，使Telnet客户进程终止连接
Connection closed.
```

telnet命令在与丢弃(discard)服务（参见1.12节）对应的端口上与主机bsdi建立一条TCP连接。这服务类型正是我们需要观察的一条连接建立与终止的服务类型，而不需要服务器发起任何数据交换。

18.2.1 tcpdump的输出

图18-1显示了这条命令产生TCP报文段的tcpdump输出。

```
1  0.0                  svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                    win 4096 <mss 1024>

2  0.002402 (0.0024)    bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                    ack 1415531522 win 4096
                                                    <mss 1024>

3  0.007224 (0.0048)    svr4.1037 > bsdi.discard: . ack 1823083522 win 4096

4  4.155441 (4.1482)    svr4.1037 > bsdi.discard: F 1415531522:1415531522(0)
                                                    ack 1823083522 win 4096

5  4.156747 (0.0013)    bsdi.discard > svr4.1037: . ack 1415531523 win 4096

6  4.158144 (0.0014)    bsdi.discard > svr4.1037: F 1823083522:1823083522(0)
                                                    ack 1415531523 win 4096

7  4.180662 (0.0225)    svr4.1037 > bsdi.discard: . ack 1823083523 win 4096
```

图18-1 TCP连接建立与终止的tcpdump 输出显示

这7个TCP报文段仅包含TCP首部。没有任何数据。

对于TCP段，每个输出行开始按如下格式显示：

源 > 目的: 标志

这里的标志代表TCP首部（图17-2）中6个标志比特中的4个。图18-2显示了表示标志的5个字符的含义。

标志	3字符缩写	描 述
S	SYN	同步序号
F	FIN	发送方完成数据发送
R	RST	复位连接
P	PSH	尽可能快地将数据送往接收进程
.		以上四个标志比特均置0

图18-2 tcpdump 对TCP首部中部分标志比特的字符表示

在这个例子中，我们看到了S、F和句点“.”标志符。我们将在以后看到其他的两个标志（R和P）。TCP首部中的其他两个标志比特——ACK 和 URG——tcpdump将作特殊显示。

图18-2所示的4个标志比特中的多个可能同时出现在一个报文段中，但通常一次只见到一个。

RFC 1025 [Postel 1987]，“TCP and IP Bake Off”，将一种报文段称为Kamikaze分组[⊖]，在这样的报文段中有最大数量的标志比特同时被置为1（SYN, URG, PSH, FIN和1字节的数据）。这样的报文段也叫作nastygram, 圣诞树分组，灯测试报文段(lamp test segment)。

在第1行中，字段1415531521:1415531521(0)表示分组的序号是1415531521，而报文段中数据字节数为0。tcpdump显示这个字段的格式是开始的序号、一个冒号、隐含的结尾序号及圆括号内的数据字节数。显示序号和隐含结尾序号的优点是便于了解数据字节数大于 0时的隐含结尾序号。这个字段只有在满足条件（1）报文段中至少包含一个数据字节；或者（2）SYN、FIN或RST被设置为1时才显示。图18-1中的第1、2、4和6行是因为标志比特被置为1而显示这个字段的，在这个例子中通信双方没有交换任何数据。

在第2行中，字段ack 1415531522表示确认序号。它只有在首部中的 ACK标志比特被设置1时才显示。

每行显示的字段win 4096表示发端通告的窗口大小。在这些例子中，我们没有交换任何数据，窗口大小就维持默认情况下的 4096（我们将在20.4节中讨论TCP窗口大小）。

图18-1中的最后一个字段<mss 1024>表示由发端指明的最大报文段长度选项。发端将不接收超过这个长度的 TCP报文段。这通常是为了避免分段（见 11.5节）。我们将在18.4节讨论最大报文段长度，而在18.10节介绍不同TCP 选项的格式。

18.2.2 时间系列

图18-3显示了这些分组序列的时间系列（在图 6-11中已经首次介绍了这些时间系列的一些基本特性）。这个图显示出哪一端正在发送分组。我们也将对 tcpdump输出作一些扩展（例如，印出SYN而不是S）。在这个时间系列中也省略窗口大小的值，因为它和我们的讨论无关。

18.2.3 建立连接协议

现在让我们回到图18-3所示的TCP协议中来。为了建立一条TCP连接：

⊖ Kamikaze是神风队队员或神风队所使用的飞机。在第二次世界大战末期，日本空军的神风队队员驾驶满载炸弹的飞机去撞击轰炸目标，企图与之同归于尽。

1) 请求端（通常称为客户）发送一个 SYN段指明客户打算连接的服务器的端口，以及初始序号（ISN，在这个例子中为1415531521）。这个SYN段为报文段1。

2) 服务器发回包含服务器的初始序号的 SYN报文段（报文段2）作为应答。同时，将确认序号设置为客户的ISN加1以对客户的SYN报文段进行确认。一个SYN将占用一个序号。

3) 客户必须将确认序号设置为服务器的 ISN加1以对服务器的SYN报文段进行确认（报文段3）。

这三个报文段完成连接的建立。这个过程也称为三次握手（ three-way handshake）。

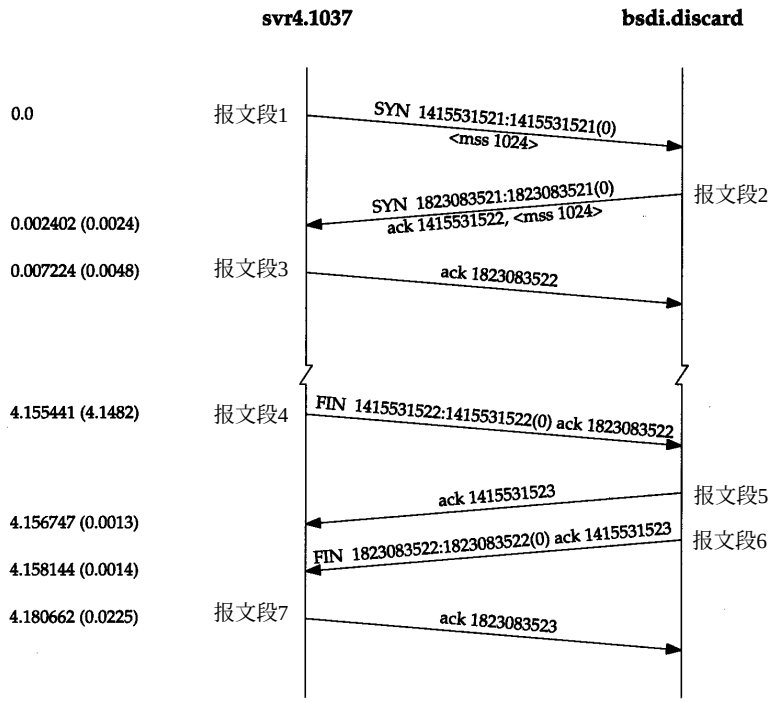


图18-3 连接建立与终止的时间系列

发送第一个SYN的一端将执行主动打开（active open）。接收这个SYN并发回下一个SYN的另一端执行被动打开（passive open）（在18.8节我们将介绍双方如何都执行主动打开）。

当一端为建立连接而发送它的 SYN时，它为连接选择一个初始序号。 ISN随时间而变化，因此每个连接都将具有不同的ISN。RFC 793 [Postel 1981c]指出ISN可看作是一个32比特的计数器，每4ms加1。这样选择序号的目的在于防止在网络中被延迟的分组在以后又被传送，而导致某个连接的一方对它作错误的解释。

如何进行序号选择？在4.4BSD（和多数的伯克利的实现版）中，系统初始化时初始的发送序号被初始化为1。这种方法违背了Host Requirements RFC（在这个代码中的一个注释确认这是一个错误）。这个变量每0.5秒增加64000，并每隔9.5小时又回到0（对应这个计数器每8 ms加1，而不是每4 ms加1）。另外，每次建立一个连接后，这个变量将增加64000。

报文段3与报文段4之间4.1秒的时间间隔是建立 TCP连接到向telnet键入quit命令来中止该连接的时间。

18.2.4 连接终止协议

建立一个连接需要三次握手，而终止一个连接要经过 4次握手。这由TCP的半关闭（half-close）造成的。既然一个TCP连接是全双工（即数据在两个方向上能同时传递），因此每个方向必须单独地进行关闭。这原则就是当一方完成它的数据发送任务后就能发送一个 FIN来终止这个方向连接。当一端收到一个 FIN，它必须通知应用层另一端几经终止了那个方向的数据传送。发送FIN通常是应用层进行关闭的结果。

收到一个FIN只意味着在这一方向上没有数据流动。一个 TCP连接在收到一个FIN后仍能发送数据。而这对利用半关闭的应用来说是可能的，尽管在实际应用中只有很少的 TCP应用程序这样做。正常关闭过程如图 18-3所示。我们将在18.5节中详细介绍半关闭。

首先进行关闭的一方（即发送第一个 FIN）将执行主动关闭，而另一方（收到这个 FIN）执行被动关闭。通常一方完成主动关闭而另一方完成被动关闭，但我们将在 18.9节看到双方如何都执行主动关闭。

图18-3中的报文段4发起终止连接，它由Telnet客户端关闭连接时发出。这在我们键入 quit命令后发生。它将导致TCP客户端发送一个FIN，用来关闭从客户到服务器的数据传送。

当服务器收到这个 FIN，它发回一个ACK，确认序号为收到的序号加 1（报文段5）。和SYN一样，一个FIN将占用一个序号。同时TCP服务器还向应用程序（即丢弃服务器）传送一个文件结束符。接着这个服务器程序就关闭它的连接，导致它的 TCP端发送一个FIN（报文段6），客户必须发回一个确认，并将确认序号设置为收到序号加1（报文段7）。

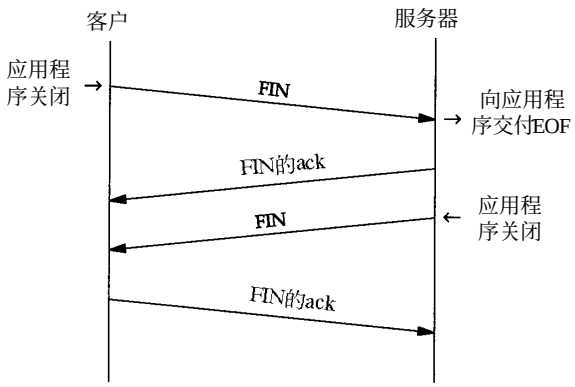


图18-4 连接终止期间报文段的正常交换

图18-4显示了终止一个连接的典型握手顺序。我们省略了序号。在这个图中，发送FIN将导致应用程序关闭它们的连接，这些FIN的ACK是由TCP软件自动产生的。

连接通常是由客户端发起的，这样第一个 SYN从客户传到服务器。每一端都能主动关闭这个连接（即首先发送 FIN）。然而，一般由客户端决定何时终止连接，因为客户进程通常由用户交互控制，用户会键入诸如“ quit ”一样的命令来终止进程。在图 18-4中，我们能改变上边的标识，将左方定为服务器，右方定为客户，一切仍将像显示的一样工作（例如在 14.4节中的第一个例子中就是由 daytime服务器关闭连接的）。

18.2.5 正常的tcpdump输出

对所有的数值很大的序号进行排序是很麻烦的，因此默认情况下 `tcpdump`只在显示SYN报文段时显示完整的序号，而对其后的序号则显示它们与初始序号的相对偏移值（为了得到图18-1的输出显示必须加上 `-S`选项）。对应于图18-1的正常`tcpdump`显示如图18-5所示：

除非我们需要显示完整的序号，否则将在以下的例子中使用这种形式的输出显示。

```
1  0.0                    svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                    win 4096 <mss 1024>
2  0.002402 (0.0024)      bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                    ack 1415531522
                                                    win 4096 <mss 1024>
3  0.007224 (0.0048)      svr4.1037 > bsdi.discard: . ack 1 win 4096
4  4.155441 (4.1482)      svr4.1037 > bsdi.discard: F 1:1(0) ack 1 win 4096
5  4.156747 (0.0013)      bsdi.discard > svr4.1037: . ack 2 win 4096
6  4.158144 (0.0014)      bsdi.discard > svr4.1037: F 1:1(0) ack 2 win 4096
7  4.180662 (0.0225)      svr4.1037 > bsdi.discard: . ack 2 win 4096
```

图18-5 连接建立与终止的正常tcpdump 输出

18.3 连接建立的超时

有很多情况导致无法建立连接。一种情况是服务器主机没有处于正常状态。为了模拟这种情况，我们断开服务器主机的电缆线，然后向它发出telnet命令。图18-6显示了tcpdump的输出。

```
1   0.0                   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
2   5.814797 ( 5.8148)    bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
3  29.815436 (24.0006)    bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
```

图18-6 建立连接超时的tcpdump 输出

在这个输出中有趣的一点是客户间隔多长时间发送一个 SYN，试图建立连接。第2个SYN与第1个的间隔是5.8秒，而第3个与第2个的间隔是24秒。

作为一个附注，这个例子运行38分钟后客户重新启动。这对应初始序号为291 008 001（约为38×60×64000×2）。我们曾经介绍过使用典型的伯克利实现版的系统将初始序号初始化为1，然后每隔0.5秒就增加64 000。

另外，因为这是系统启动后的第一个TCP连接，因此客户的端口号是1024。

图18-6中没有显示客户端在放弃建立连接尝试前进行 SYN重传的时间。为了了解它我们必须对telnet命令进行计时：

```
bsdi % date ; telnet svr4 discard ; date
Thu Sep 24 16:24:11 MST 1992
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection timed out
Thu Sep 24 16:25:27 MST 1992
```

时间差值是 76秒。大多数伯克利系统将建立一个新连接的最长时间限制为 75秒。我们将在21.4节看到由客户发出的第 3个分组大约在16:25:29超时， 客户在它第3个分组发出后48秒而不是75秒后放弃连接。

18.3.1 第一次超时时间

在图18-6中一个令人困惑的问题是第一次超时时间为 5.8秒，接近6秒，但不准确，相比之

下第二个超时时间几乎准确地为24秒。运行十多次测试，发现第一次超时时间在5.59秒~5.93秒之间变化。然而，第二次超时时间则总是24.00秒（精确到小数点后面两位）。

这是因为BSD版的TCP软件采用一种500 ms的定时器。这种500 ms的定时器用于确定本章中所有的各种各样的TCP超时。当我们键入telnet命令，将建立一个6秒的定时器（12个时钟滴答（tick）），但它可能在之后的5.5秒~6秒内的任意时刻超时。图18-7显示了这一发生过程。尽管定时器初始化为12个时钟滴答，但定时计数器会在设置后的第一个0~500 ms中的任意时刻减1。从那以后，定时计数器大约每隔500 ms减1，但在第1个500 ms内是可变的（我们使用限定词“大约”是因为在TCP每隔500 ms获得系统控制的瞬间，系统内核可能会优先处理其他中断）。

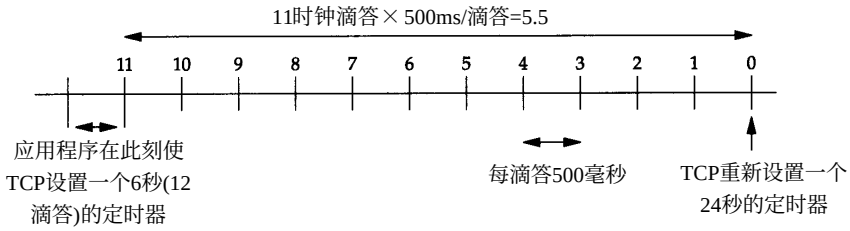


图18-7 TCP的500 ms定时器

当滴答计数器为0时，6秒的定时器便会超时（见图18-7），这个定时器会在以后的24秒（48个滴答）重新复位。之后的下一个定时器将更接近24秒，因为当TCP的500 ms定时器被内核调用时，它就会被修改一次。

18.3.2 服务类型字段

在图18-6中，出现了符号[tos 0x10]。这是IP数据报内的服务类型（TOS）字段（参见图3-2）。BSD/386中的Telnet客户进程将这个字段设置为最小时延。

18.4 最大报文段长度

最大报文段长度（MSS）表示TCP传往另一端的最大块数据的长度。当一个连接建立时，连接的双方都要通告各自的MSS。我们已经见过MSS都是1024。这导致IP数据报通常是40字节长：20字节的TCP首部和20字节的IP首部。

在有些书中，将它看作可“协商”选项。它并不是任何条件下都可协商。当建立一个连接时，每一方都有用于通告它期望接收的MSS选项（MSS选项只能出现在SYN报文段中）。如果一方不接收来自另一方的MSS值，则MSS就定为默认值536字节（这个默认值允许20字节的IP首部和20字节的TCP首部以适合576字节IP数据报）。

一般说来，如果没有分段发生，MSS还是越大越好（这也并不总是正确，参见图24-3和图24-4中的例子）。报文段越大允许每个报文段传送的数据就越多，相对IP和TCP首部有更高的网络利用率。当TCP发送一个SYN时，或者是因为一个本地应用进程想发起一个连接，或者是因为另一端的主机收到了一个连接请求，它能将MSS值设置为外出接口上的MTU长度减去固定的IP首部和TCP首部长度。对于一个以太网，MSS值可达1460字节。使用IEEE 802.3的封装（参见2.2节），它的MSS可达1452字节。

在本章见到的涉及BSD/386和SVR4的MSS为1024，这是因为许多BSD的实现版本需要

MSS为512的倍数。其他的系统，如SunOS 4.1.3、Solaris 2.2 和AIX 3.2.2，当双方都在一个本地以太网上时都规定MSS为1460。[Mogul 1993] 的比较显示了在以太网上1460的MSS在性能上比1024的MSS更好。

如果目的IP地址为“非本地的(nonlocal)”，MSS通常的默认值为536。而区分地址是本地还是非本地是简单的，如果目的IP地址的网络号与子网号都和我们的相同，则是本地的；如果目的IP地址的网络号与我们的完全不同，则是非本地的；如果目的IP地址的网络号与我们的相同而子网号与我们的不同，则可能是本地的，也可能是非本地的。大多数TCP实现版都提供了一个配置选项（附录E和图E-1），让系统管理员说明不同的子网是属于本地还是非本地。这个选项的设置将确定MSS可以选择尽可能的大（达到外出接口的MTU长度）或是默认值536。

MSS让主机限制另一端发送数据报的长度。加上主机也能控制它发送数据报的长度，这将使以较小MTU连接到一个网络上的主机避免分段。

考虑我们的主机slip，通过MTU为296的SLIP链路连接到路由器bsdi上。图18-8显示这些系统和主机sun。

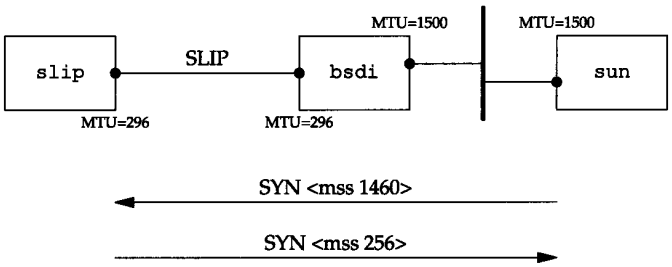


图18-8 显示sun与slip间TCP连接的MSS值

从sun向slip发起一个TCP连接，并使用tcpdump来观察报文段。图18-9显示这个连接的建立（省略了通告窗口大小）。

```
1  0.0                  sun.1093 > slip.discard: S 517312000:517312000(0)
                                                 <mss 1460>
2  0.10 (0.00)          slip.discard > sun.1093: S 509556225:509556225(0)
                                                 ack 517312001 <mss 256>
3  0.10 (0.00)          sun.1093 > slip.discard: . ack 1
```

图18-9 tcpdump显示了从sun向slip建立连接的过程

在这个例子中，sun发送的报文段不能超过256字节的数据，因为它收到的MSS选项值为256（第2行）。此外，由于slip知道它外出接口的MTU长度为296，即使sun已经通告它的MSS为1460，但为避免将数据分段，它不会发送超过256字节数据的报文段。系统允许发送的数据长度小于另一端的MSS值。

只有当一端的主机以小于576字节的MTU直接连接到一个网络中，避免这种分段才会有效。如果两端的主机都连接到以太网上，都采用536的MSS，但中间网络采用296的MTU，也将会出现分段。使用路径上的MTU发现机制（参见24.2节）是关于这个问题的唯一方法。

18.5 TCP的半关闭

TCP提供了连接的一端在结束它的发送后还能接收来自另一端数据的能力。这就是所谓

的半关闭。正如我们早些时候提到的只有很少的应用程序使用它。

为了使用这个特性，编程接口必须为应用程序提供一种方式来说明“我已经完成了数据传送，因此发送一个文件结束（FIN）给另一端，但我还想接收另一端发来的数据，直到它给我发来文件结束（FIN）”。

如果应用程序不调用close而调用shutdown，且第2个参数值为1，则插口的API支持半关闭。然而，大多数的应用程序通过调用close终止两个方向的连接。

图18-10显示了一个半关闭的典型例子。让左方的客户端开始半关闭，当然也可以由另一端开始。开始的两个报文段和图18-4是相同的：初始端发出的FIN，接着是另一端对这个FIN的ACK报文段。但后面就和图18-4不同，因为接收半关闭的一方仍能发送数据。我们只显示一个数据报文段和一个ACK报文段，但可能发送了许多数据报文段（将在第19章讨论数据报文段和确认报文段的交换）。当收到半关闭的一端在完成它的数据传送后，将发送一个FIN关闭这个方向的连接，这将传送一个文件结束符给发起这个半关闭的应用进程。当对第二个FIN进行确认后，这个连接便彻底关闭了。

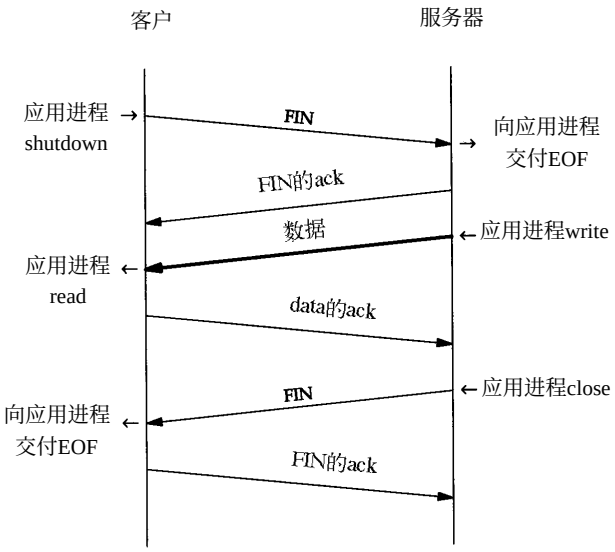


图18-10 TCP半关闭的例子

为什么要有半关闭？一个例子是Unix中的rsh(1)命令，它将完成在另一个系统上执行一个命令。命令

```
sun % rsh bsdi sort < datafile
```

将在主机bsdi上执行sort排序命令，rsh命令的标准输入来自文件datafile。rsh将在它与在另一主机上执行的程序间建立一个TCP连接。rsh的操作很简单：它将标准输入（datafile）复制给TCP连接，并将结果从TCP连接中复制给标准输出（我们的终端）。图18-11显示了这个建立过程（牢记TCP连接是全双工的）。

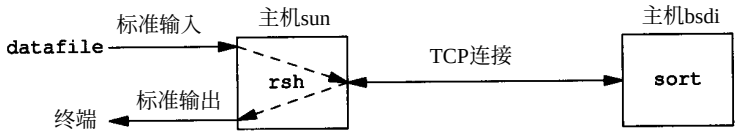


图18-11 命令：rsh bsdi sort < datafile

在远端主机bsdi上，rshd服务器将执行sort程序，它的标准输入和标准输出都是TCP连接。第14章的[Stevens 1990]详细介绍了有关Unix进程的结构，但这儿涉及的是使用TCP连接以及需要使用TCP的半关闭。

sort程序只有读取到所有输入数据后才能产生输出。所有的原始数据通过TCP连接从rsh客户端传送到sort服务器进行排序。当输入（datafile）到达文件尾时，rsh客户端

执行这个TCP连接的半关闭。接着sort服务器在它的标准输入（这个TCP连接）上收到一个文件结束符，对数据进行排序，并将结果写在它的标准输出上（ TCP连接）。rsh客户端继续接收来自TCP连接另一端的数据，并将排序的文件复制到它的标准输出上。

没有半关闭，需要其他的一些技术让客户通知服务器，客户端已经完成了它的数据传送，但仍要接收来自服务器的数据。使用两个TCP连接也可作为一个选择，但使用半关闭的单连接更好。

18.6 TCP的状态变迁图

我们已经介绍了许多有关发起和终止 TCP连接的规则。这些规则都能从图 18-12所示的状态变迁图中得出。

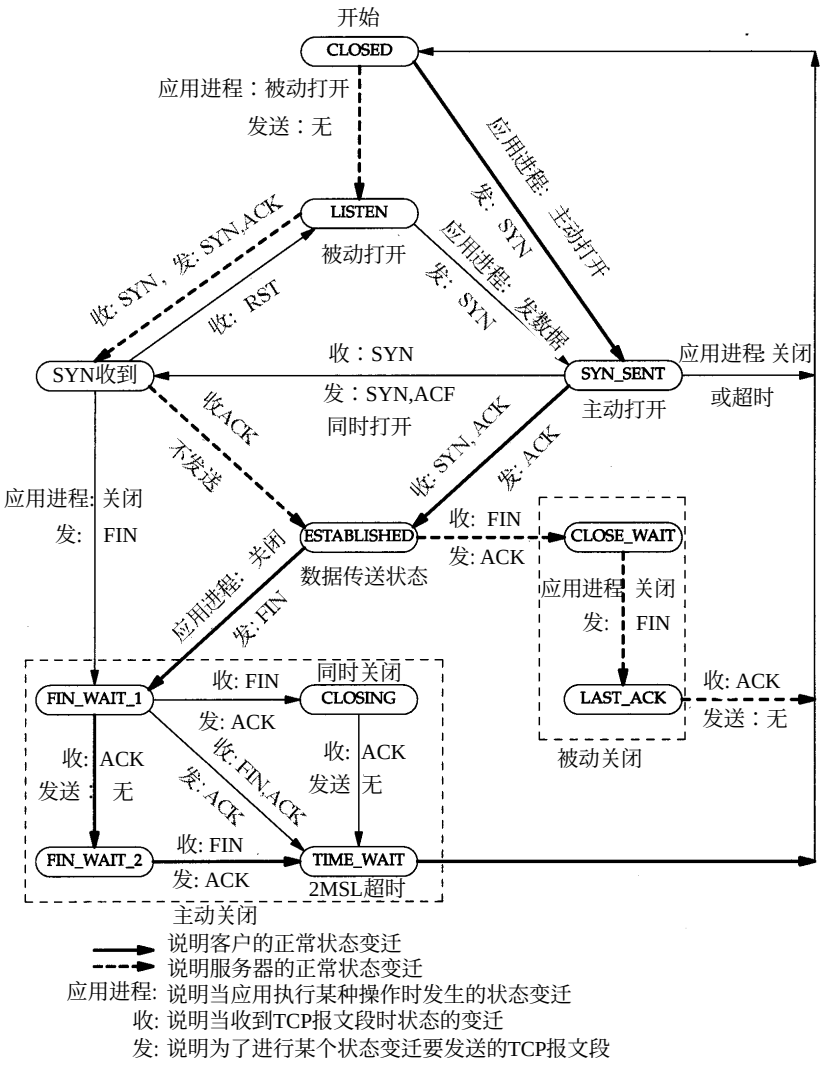


图18-12 TCP的状态变迁图

在这个图中要注意的第一点是一个状态变迁的子集是“ 典型的 ”。我们用粗的实线箭头表示正常的客户端状态变迁，用粗的虚线箭头表示正常的服务器状态变迁。

第二点是两个导致进入ESTABLISHED状态的变迁对应打开一个连接，而两个导致从ESTABLISHED状态离开的变迁对应关闭一个连接。ESTABLISHED状态是连接双方能够进行双向数据传递的状态。以后的章节将介绍这个状态。

将图中左下角4个状态放在一个虚线框内，并标为“主动关闭”。其他两个状态（CLOSE_WAIT和LAST_ACK）也用虚线框住，并标为“被动关闭”。

在这个图中11个状态的名称（CLOSED, LISTEN, SYN_SENT等）是有意与netstat命令显示的状态名称一致。netstat对状态的命名几乎与在RFC 793中的最初描述一致。CLOSED状态不是一个真正的状态，而是这个状态图的假想起点和终点。

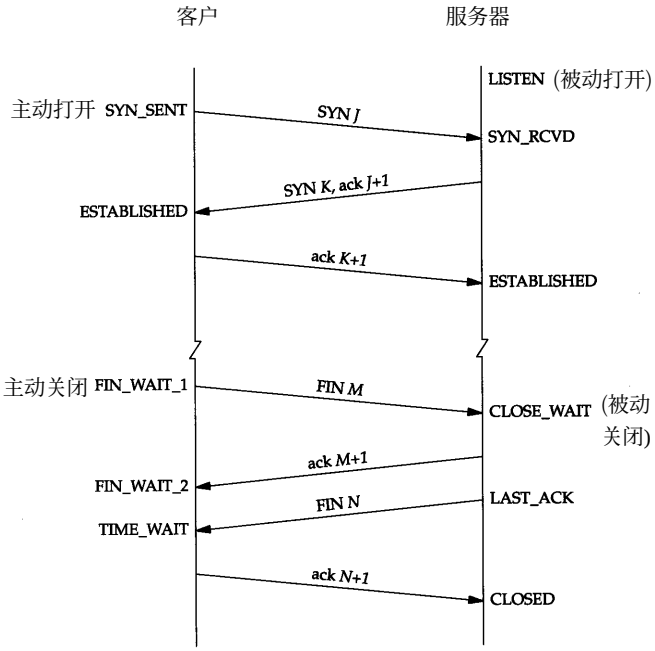


图18-13 TCP正常连接建立和终止所对应的状态

从LISTEN到SYN_SENT的变迁是正确的，但伯克利版的TCP软件并不支持它。

只有当SYN_RCVD状态是从LISTEN状态（正常情况）进入，而不是从SYN_SENT状态（同时打开）进入时，从SYN_RCVD回到LISTEN的状态变迁才是有效的。这意味着如果我们执行被动关闭（进入LISTEN），收到一个SYN，发送一个带ACK的SYN（进入SYN_RCVD），然后收到一个RST，而不是一个ACK，便又回到LISTEN状态并等待另一个连接请求的到来。

图18-13显示了在正常的TCP连接的建立与终止过程中，客户与服务器所经历的不同状态。它是图18-3的再现，不同的是仅显示了一些状态。

假定在图18-13中左边的客户执行主动打开，而右边的服务器执行被动打开。尽管图中显示出由客户端执行主动关闭，但和早前我们提到的一样，另一端也能执行主动关闭。

可以使用图18-12的状态图来跟踪图18-13的状态变化过程，以便明白每个状态的变化。

18.6.1 2MSL等待状态

TIME_WAIT状态也称为2MSL等待状态。每个具体TCP实现必须选择一个报文段最大生存时间MSL（Maximum Segment Lifetime）。它是任何报文段被丢弃前在网络内的最长时间。我们知道这个时间是有限的，因为TCP报文段以IP数据报在网络内传输，而IP数据报则有限制其生存时间的TTL字段。

> RFC 793 [Postel 1981c] 指出MSL为2分钟。然而，实现中的常用值是30秒，1分钟，或2分钟。

从第8章我们知道在实际应用中，对IP数据报TTL的限制是基于跳数，而不是定时器。

对一个具体实现所给定的MSL值，处理的原则是：当TCP执行一个主动关闭，并发回最

后一个ACK，该连接必须在TIME_WAIT状态停留的时间为2倍的MSL。这样可让TCP再次发送最后的ACK以防这个ACK丢失（另一端超时并重发最后的FIN）。

这种2MSL等待的另一个结果是这个TCP连接在2MSL等待期间，定义这个连接的插口（客户的IP地址和端口号，服务器的IP地址和端口号）不能再被使用。这个连接只能在2MSL结束后才能再被使用。

遗憾的是，大多数TCP实现（如伯克利版）强加了更为严格的限制。在2MSL等待期间，插口中使用的本地端口在默认情况下不能再被使用。我们将在下面看到这个限制的例子。

某些实现和API提供了一种避开这个限制的方法。使用插口API时，可说明其中的SO_REUSEADDR选项。它将让调用者对处于2MSL等待的本地端口进行赋值，但我们将看到TCP原则上仍将避免使用仍处于2MSL连接中的端口。

在连接处于2MSL等待时，任何迟到的报文段将被丢弃。因为处于2MSL等待的、由该插口对(socket pair)定义的连接在这段时间内不能被再用，因此当要建立一个有效的连接时，来自该连接的一个较早替身（incarnation）的迟到报文段作为新连接的一部分不可能不被曲解（一个连接由一个插口对来定义。一个连接的新的实例（instance）称为该连接的替身）。

我们说图18-13中客户执行主动关闭并进入TIME_WAIT是正常的。服务器通常执行被动关闭，不会进入TIME_WAIT状态。这暗示如果我们终止一个客户程序，并立即重新启动这个客户程序，则这个新客户程序将不能重用相同的本地端口。这不会带来什么问题，因为客户使用本地端口，而并不关心这个端口号是什么。

然而，对于服务器，情况就有所不同，因为服务器使用熟知端口。如果我们终止一个已经建立连接的服务器程序，并试图立即重新启动这个服务器程序，服务器程序将不能把它的这个熟知端口赋值给它的端点，因为那个端口是处于2MSL连接的一部分。在重新启动服务器程序前，它需要在1~4分钟。

可以通过sock程序看到这一切。我们启动服务器程序，从一个客户程序进行连接，然后停止这个服务器程序。

```
sun % sock -v -s 6666                    启动服务器进程，在端口6666监听(在bsdi上执行客
                                         户进程与该端口进行连接)
connection on 140.252.13.33.6666 from 140.252.13.35.1081
^?                                       键入中断键停止服务器进程
sun % sock -s 6666                       并立即在同一端口重启服务器进程
can't bind local address: Address already in use
sun % netstat                            检测连接状态
Active Internet connections
Proto Recv-Q Send-Q  Local Address   Foreign Address     (state)
tcp        0      0  sun.6666        bsdi.1081          TIME_WAIT
                                         删除了许多其他行
```

当重新启动服务器程序时，程序报告一个差错信息说明不能绑定它的熟知端口，因为该端口已被使用（即它处于2MSL等待）。

运行netstat程序来查看连接的状态，以证实它的确处于2MSL等待状态。

如果我们一直试图重新启动服务器程序，并测量它直到成功所需的时间，我们就能确定出2MSL值。对于SunOS 4.1.3、SVR4、BSD/386和AIX 3.2.2，它需要1分钟才能重新启动服务器程序，这意味着它们的MSL值为30秒。而对于Solaris 2.2，它需要4分

钟才能重新启动服务器程序，这表示它的MSL值为2分钟。

如果一个客户程序试图申请一个处于 2MSL等待的端口（客户程序通常不会这么做），就会出现同样的差错。

```
sun % sock -v bsdi echo                    启动客户进程，与回显服务器进程连接
connected on 140.252.13.33.1162 to 140.252.13.35.7
hello there                                键入这一行
hello there                                这一行应被服务器进程回显
^D                                         键入文件结束符终止客户进程

sun % sock -b1162 bsdi echo
can't bind local address: Address already in use
```

我们在第1次执行客户程序时采用 -v选项来查看它使用的本地端口为（ 1162）。第2次执行客户程序时则采用 - b选项来选择端口 1162为它的本地端口。正如我们所预料的那样，客户程序无法那么做，因为那个端口是一个还处于 2MSL等待连接的一部分。

需要再次强调2MSL等待的一个效果，因为我们将在第 27章的文件传输协议FTP中遇到它。和以前介绍的一样，一个插口对（即包含本地 IP地址、本地端口、远端 IP地址和远端端口的4元组）在它处于2MSL等待时，将不能再被使用。尽管许多具体的实现中允许一个进程重新使用仍处于2MSL等待的端口（通常是设置选项 SO_REUSEADDR），但TCP不能允许一个新的连接建立在相同的插口对上。可通过下面的试验来看到这一点：

```
sun % sock -v -s 6666                      启动服务器进程，在端口6666监听(在bsdi上执行
                                           客户进程与该端口进行连接)

connection on 140.252.13.33.6666 from 140.252.13.35.1098
^?                                         键入中断键停止服务器进程

sun % sock -b6666 bsdi 1098                尝试在本地端口6666启动客户进程
can't bind local address: Address already in use

sun % sock -A -b6666 bsdi 1098             再次尝试，加上 - A选项
active open error: Address already in use
```

在第1次运行sock程序中，我们将它作为服务器程序，端口号为 6666，并从主机bsdi上的一个客户程序与它连接，这个客户程序使用的端口为 1098。我们终止服务器程序，因此它将执行主动关闭。这将导致 4元组 140.252.13.33（本地 IP地址）、6666（本地端口号）、140.252.13.35（另一端IP地址）和1098（另一端的端口号）在服务器主机进入 2MSL等待。

在第2次运行sock程序时，我们将它作为客户程序，并试图将它的本地端口号指明为6666，同时与主机bsdi在端口1098上进行连接。但这个程序在试图将它的本地端口号赋值为6666时产生了一个差错，因为这个端口是处于 2MSL等待4元组的一部分。

为了避免这个差错，我们再次运行这个程序，并使用选项 - A来设置前面提到的SO_REUSEADDR。这将让sock程序能将它的本地端口号设置为 6666，但当我们试图进行主动打开时，又出现了一个差错。即使它能将它的本地端口设置为 6666，但它仍不能和主机bsdi在端口1098上进行连接，因为定义这个连接的插口对仍处于 2MSL等待状态。

如果我们试图从其他主机来建立这个连接会如何？首先我们必须在 sun上以 - A标记来重新启动服务器程序，因为它需要的端口（ 6666）是还处于2MSL等待连接的一部分。

```
sun % sock -A -s 6666          启动服务器程序，在端口 6666监听
```

接着，在2MSL等待结束前，我们在bsdi上启动客户程序：

```
bsdi %sock -b1098 sun 6666
```

```
connected on 140.252.13.35.1098 to 140.252.13.33.6666
```

不幸的是它成功了！这违反了TCP规范，但被大多数的伯克利版实现所支持。这些实现允许一个新的连接请求到达仍处于TIME_WAIT状态的连接，只要新的序号大于该连接前一个替身的最后序号。在这个例子中，新替身的ISN被设置为前一个替身最后序号与128 000的和。附录的RFC 1185 [Jacobsan、Braden和Zhang 1990] 指出了这项技术仍可能存在缺陷。

对于同一连接的前一个替身，这个具体实现中的特性让客户程序和服务器程序能连续地重用每一端的相同端口号，但这只有在服务器执行主动关闭才有效。我们将在图 27-8中使用FTP时看到这个2MSL等待条件的另一个例子。也见习题18.5。

18.6.2 平静时间的概念

对于来自某个连接的较早替身的迟到报文段， 2MSL等待可防止将它解释成使用相同插口对的新连接的一部分。但这只有在处于2MSL等待连接中的主机处于正常工作状态时才有效。

如果使用处于2MSL等待端口的主机出现故障，它会在MSL秒内重新启动，并立即使用故障前仍处于2MSL的插口对来建立一个新的连接吗？如果是这样，在故障前从这个连接发出而迟到的报文段会被错误地当作属于重启后新连接的报文段。无论如何选择重启后新连接的初始序号，都会发生这种情况。

为了防止这种情况， RFC 793指出TCP在重启动后的MSL秒内不能建立任何连接。这就称为平静时间(quiet time)。

只有极少的实现版遵守这一原则，因为大多数主机重启动的时间都比MSL秒要长。

18.6.3 FIN_WAIT_2 状态

在FIN_WAIT_2状态我们已经发出了FIN，并且另一端也已对它进行确认。除非我们在实行半关闭，否则将等待另一端的应用层意识到它已收到一个文件结束符说明，并向我们发一个FIN来关闭另一方向的连接。只有当另一端的进程完成这个关闭，我们这端才会从FIN_WAIT_2状态进入TIME_WAIT状态。

这意味着我们这端可能永远保持这个状态。另一端也将处于 CLOSE_WAIT状态，并一直保持这个状态直到应用层决定进行关闭。

许多伯克利实现采用如下方式来防止这种在FIN_WAIT_2状态的无限等待。如果执行主动关闭的应用层将进行全关闭，而不是半关闭来说明它还想接收数据，就设置一个定时器。如果这个连接空闲10分钟75秒，TCP将进入CLOSED状态。在实现代码的注释中确认这个实现代码违背协议的规范。

18.7 复位报文段

我们已经介绍了TCP首部中的RST比特是用于“复位”的。一般说来，无论何时一个报文段发往基准的连接（referenced connection）出现错误， TCP都会发出一个复位报文段（这里提到的“基准的连接”是指由目的IP地址和目的端口号以及源IP地址和源端口号指明的连接。这就是为什么RFC 793称之为插口）。

18.7.1 到不存在的端口的连接请求

产生复位的一种常见情况是当连接请求到达时，目的端口没有进程正在听。对于 UDP，我们在6.5节看到这种情况，当一个数据报到达目的端口时，该端口没在使用，它将产生一个ICMP端口不可达的信息。而TCP则使用复位。

产生这个例子也很容易，我们可使用 Telnet客户程序来指明一个目的端口没在使用的情况：

```
bsdi % telnet svr4 20000            端口20000未使用
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection refused
```

Telnet客户程序会立即显示这个差错信息。图 18-14显示了对应这个命令的分组交换过程。

```
1  0.0              bsdi.1087 > svr4.20000: S 297416193:297416193(0)
                                      win 4096 <mss 1024>
                                      [tos 0x10]
2  0.003771 (0.0038)    svr4.20000 > bsdi.1087: R 0:0(0) ack 297416194 win 0
```

图18-14 试图在不存在的端口上打开连接而产生的复位

在这个图中需要注意的值是复位报文段中的序号字段和确认序号字段。因为 ACK比特在到达的报文段中没有被设置为1，复位报文段中的序号被置为0，确认序号被置为进入的ISN加上数据字节数。尽管在到达的报文段中没有真正的数据，但 SYN比特从逻辑上占用了1字节的序号空间；因此，在这个例子中复位报文段中确认序号被置为 ISN与数据长度（0）、SYN比特所占的1的总和。

18.7.2 异常终止一个连接

我们在18.2节中看到终止一个连接的正常方式是一方发送 FIN。有时这也称为有序释放（orderly release），因为在所有排队数据都已发送之后才发送 FIN，正常情况下没有任何数据丢失。但也有可能发送一个复位报文段而不是 FIN来中途释放一个连接。有时称这为异常释放（abortive release）。

异常终止一个连接对应用程序来说有两个优点：（1）丢弃任何待发数据并立即发送复位报文段；（2）RST的接收方会区分另一端执行的是异常关闭还是正常关闭。应用程序使用的API必须提供产生异常关闭而不是正常关闭的手段。

使用sock程序能够观察这种异常关闭的过程。 Socket API通过“ linger on close ”选项（SO_LINGER）提供了这种异常关闭的能力。我们加上 -L选项并将停留时间设为 0。这将导致连接关闭时进行复位而不是正常的 FIN。我们连接到处于服务器上的 sock程序，并键入一输入行：

```
bsdi %sock -L0 svr4 8888      这是客户程序，服务器程序显示后面
hello, world                  键入一行输入，它被发往到另一端
^D                            键入文件结束符，终止客户程序
```

图18-15是这个例子的tcpdump输出显示（在这个图中我们已经删除了所有窗口大小的说明，因为它们与讨论无关）。

第1~3行显示出建立连接的正常过程。第 4行发送我们键入的数据行（ 12个字符和Unix换

行符），第5行是对收到数据的确认。

```
1  0.0                    bsdi.1099 > svr4.8888: S 671112193:671112193(0)
                                                  <mss 1024>
2  0.004975 (0.0050)      svr4.8888 > bsdi.1099: S 3224959489:3224959489(0)
                                                  ack 671112194 <mss 1024>
3  0.006656 (0.0017)      bsdi.1099 > svr4.8888: . ack 1

4  4.833073 (4.8264)      bsdi.1099 > svr4.8888: P 1:14(13) ack 1
5  5.026224 (0.1932)      svr4.8888 > bsdi.1099: . ack 14

6  9.527634 (4.5014)      bsdi.1099 > svr4.8888: R 14:14(0) ack 1
```

图18-15 使用复位（RST）而不是FIN来异常终止一个连接

第6行对应为终止客户程序而键入的文件结束符（Control_D）。由于我们指明使用异常关闭而不是正常关闭（命令行中的-L0选项），因此主机bsdi端的TCP发送一个RST而不是通常的FIN。RST报文段中包含一个序号和确认序号。需要注意的是RST报文段不会导致另一端产生任何响应，另一端根本不进行确认。收到RST的一方将终止该连接，并通知应用层连接复位。

我们在服务器上得到下面的差错信息：

```
svr4 %sock -s 8888                    作为服务器进程运行，在端口8888监听
hello, world                          这行是客户端发送的
read error: Connection reset by peer
```

这个服务器程序从网络中接收数据并将它接收的数据显示到其标准输出上。通常，从它的TCP上收到文件结束符后便将结束，但这里我们看到当收到RST时，它产生了一个差错。这个差错正是我们所期待的：连接被对方复位了。

18.7.3 检测半打开连接

如果一方已经关闭或异常终止连接而另一方却还不知道，我们将这样的TCP连接称为半打开（Half-Open）的。任何一端的主机异常都可能导致发生这种情况。只要不打算在半打开连接上传输数据，仍处于连接状态的一方就不会检测另一方已经出现异常。

半打开连接的另一个常见原因是当客户主机突然掉电而不是正常的结束客户应用程序后再关机。这可能发生在使用PC机作为Telnet的客户主机上，例如，用户在一天工作结束时关闭PC机的电源。当关闭PC机电源时，如果已不再有要向服务器发送的数据，服务器将永远不知道客户程序已经消失了。当用户在第二天到来时，打开PC机，并启动新的Telnet客户程序，在服务器主机上会启动一个新的服务器程序。这样会导致服务器主机中产生许多半打开的TCP连接（在第23章中我们将看到使用TCP的keepalive选项能使TCP的一端发现另一端已经消失）。

能很容易地建立半打开连接。在bsdi上运行Telnet客户程序，通过它和svr4上的丢弃服务器建立连接。我们键入一行字符，然后通过tcpdump进行观察，接着断开服务器主机与以太网的电缆，并重启服务器主机。这可以模拟服务器主机出现异常（在重启服务器之前断开以太网电缆是为了防止它向打开的连接发送FIN，某些TCP在关机时会这么做）。服务器主机重启后，我们重新接上电缆，并从客户向服务器发送另一行字符。由于服务器的TCP已经重新启动，它将丢失复位前连接的所有信息，因此它不知道数据报文段中提到的连接。TCP的处理原则是接收方以复位作为应答。

```
bsdi % telnet svr4 discard                    启动客户进程
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
hi there                                      运行已正确发送
                                              重新启动服务器主机
another line                                  导致连接复位
Connection closed by foreign host.
```

图18-16是这个例子的tcpdump输出显示（已从这个输出中删除了窗口大小的说明、服务类型信息和MSS声明，因为它们与讨论无关）。

```
1    0.0                  bsdi.1102 > svr4.discard: S 1591752193:1591752193(0)
2    0.004811 (  0.0048)  svr4.discard > bsdi.1102: S 26368001:26368001(0)
                                                     ack 1591752194
3    0.006516 (  0.0017)  bsdi.1102 > svr4.discard: . ack 1

4    5.167679 (  5.1612)  bsdi.1102 > svr4.discard: P 1:11(10) ack 1
5    5.201662 (  0.0340)  svr4.discard > bsdi.1102: . ack 11

6  194.909929 (189.7083)  bsdi.1102 > svr4.discard: P 11:25(14) ack 1
7  194.914957 (  0.0050)  arp who-has bsdi tell svr4
8  194.915678 (  0.0007)  arp reply bsdi is-at 0:0:c0:6f:2d:40
9  194.918225 (  0.0025)  svr4.discard > bsdi.1102: R 26368002:26368002(0)
```

图18-16 复位作为半打开连接上数据段的应答

第1~3行是正常的连接建立过程。第 4行向丢弃服务器发送字符行“ hithere ”，第5行是确认。

然后是断开svr4的以太网电缆，重新启动svr4，并重新接上电缆。这个过程几乎需要 190秒。接着从客户端输入下一行（即“ another line ”），当我们键入回车键后，这一行被发往服务器（图18-16的第6行）。这导致服务器产生一个响应，但要注意的是由于服务器主机经过重新启动，它的 ARP高速缓存为空，因此需要一个 ARP请求和应答（第 7、8行）。第9行表示RST被发送出去。客户收到复位报文段后显示连接已被另一端的主机终止（ Telnet客户程序发出的最后信息不再有什么价值）。

18.8 同时打开

两个应用程序同时彼此执行主动打开的情况是可能的，尽管发生的可能性极小。每一方必须发送一个SYN，且这些SYN必须传递给对方。这需要每一方使用一个对方熟知的端口作为本地端口。这又称为同时打开（simultaneous open）。

例如，主机A中的一个应用程序使用本地端口 7777，并与主机B的端口8888执行主动打开。主机B中的应用程序则使用本地端口 8888，并与主机A的端口7777执行主动打开。

这与下面的情况不同：主机A中的Telnet客户程序和主机B中Telnet的服务器程序建立连接，与此同时，主机B中的Telnet客户程序与主机A的Telnet服务器程序也建立连接。在这个 Telnet例子中，两个Telnet服务器都执行被动打开，而不是主动打开，并且 Telnet客户选择的本地端口不是另一端Telnet服务器进程所熟悉的端口。

TCP是特意设计为了可以处理同时打开，对于同时打开它仅建立一条连接而不是两条连接（其他的协议族，最突出的是OSI运输层，在这种情况下将建立两条连接而不是一条连接）。

当出现同时打开的情况时，状态变迁与图 18-13所示的不同。两端几乎在同时发送 SYN，并进入SYN_SENT状态。当每一端收到SYN时，状态变为SYN_RCVD（如图18-12），同时它

们都再发SYN并对收到的SYN进行确认。当双方都收到SYN及相应的ACK时，状态都变迁为ESTABLISHED。图18-17显示了这些状态变迁过程。

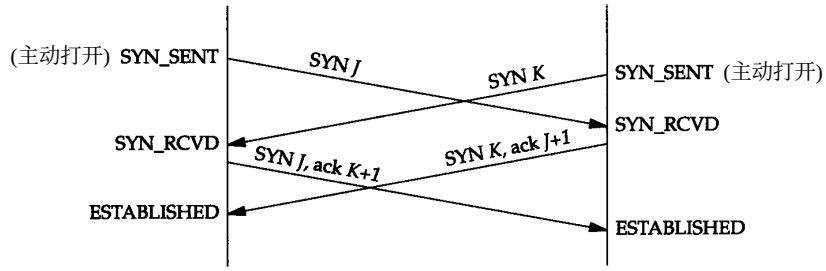


图18-17 同时打开期间报文段的交换

一个同时打开的连接需要交换4个报文段，比正常的三次握手多一个。此外，要注意的是我们没有将任何一端称为客户或服务器，因为每一端既是客户又是服务器。

一个例子

尽管很难，但仍有可能产生一个同时打开的连接。两端必须几乎在同时启动，以便收到彼此的SYN。只要两端有较长的往返时间就能保证这一点。这样我们将一端设置在主机 bsdi 上，另一端则设置在主机 vangogh.cs.berkeley.edu上。由于两端之间有一条拨号链路SLIP，它的往返时间对保证双方同步收到SYN是足够长的（几百毫秒）。

一端（bsdi）将本地端口设置为8888（使用命令行选项-b），并对另一端主机端口7777执行主动打开。

```
bsdi % sock -v -b8888 vangogh.cs.berkeley.edu 7777
connected on 140.252.13.35.8888 to 128.32.130.2.7777
TCP_MAXSEG = 512
hello, world                          键入该行
and hi there                          在另一端键入这一行
connection closed by peer             当收到FIN时的输出显示
```

另一端也几乎在同一时间将本地端口设置为7777，并对端口8888执行主动打开。

```
vangogh % sock -v -b7777 bsdi.tuc.noao.edu 8888
connected on 128.32.130.2.7777 to 140.252.13.35.8888
TCP_MAXSEG = 512
hello, world                          这是另一端键入的行
and hi there                          键入这行
^D                                    键入文件结束符EOF
```

我们指明带-v标志的sock程序来验证连接两端的IP地址和端口号。这个选项也显示每一端的MSS值。为证实两端确实在相互交谈，我们在每一端还输入一行字符，看它们是否会被送到另一端并显示出来。

图18-18显示了这个连接的段交换过程（我们删除了出现在来自 vangogh第一个SYN中的一些新的TCP选项，因为vangogh使用4.4BSD系统。将在18.10节介绍这些较新的选项）。注意两个SYN（第1~2行）后跟着两个带ACK的SYN（第3~4行）。它们将执行同时打开。

第5行显示了由bsdi发送给vangogh的输入行“ hello, world ”，第6行对此进行确认。第7~8行对应另一方向的输入行“ and hi there ”和确认。第9~12行显示正常的连接关闭。

许多伯克利版的TCP实现都不能正确地支持同时打开。在这些系统中，如果能够

进行SYN的同步接收，你将经历极多的报文段交换过程才能关闭它们。每个报文段交换过程包括每个方向上的一个 SYN和一个ACK。图18-12中从SYN_SENT到状态SYN_RCVD的变迁在许多TCP实现中很少测试过。

```
 1   0.0                 bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                   win 4096 <mss 512>
 2   0.213782 (0.2138)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                   win 8192 <mss 512>
 3   0.215399 (0.0016)   bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                   ack 1058199042 win 4096
                                                   <mss 512>
 4   0.340405 (0.1250)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                   ack 91904002 win 8192
                                                   <mss 512>
 5   5.633142 (5.2927)   bsdi.8888 > vangogh.7777: P 1:14(13) ack 1 win 4096
 6   6.100366 (0.4672)   vangogh.7777 > bsdi.8888: . ack 14 win 8192
 7   9.640214 (3.5398)   vangogh.7777 > bsdi.8888: P 1:14(13) ack 14 win 8192
 8   9.796417 (0.1562)   bsdi.8888 > vangogh.7777: . ack 14 win 4096
 9  13.060395 (3.2640)   vangogh.7777 > bsdi.8888: F 14:14(0) ack 14 win 8192
10  13.061828 (0.0014)   bsdi.8888 > vangogh.7777: . ack 15 win 4096
11  13.079769 (0.0179)   bsdi.8888 > vangogh.7777: F 14:14(0) ack 15 win 4096
12  13.299940 (0.2202)   vangogh.7777 > bsdi.8888: . ack 15 win 8192
```

图18-18 同时打开期间的报文段交换过程

18.9 同时关闭

我们在以前讨论过一方（通常但不总是客户方）发送第一个 FIN执行主动关闭。双方都执行主动关闭也是可能的， TCP协议也允许这样的同时关闭（simultaneous close）。

在图18-12中，当应用层发出关闭命令时，两端均从 ESTABLISHED变为FIN_WAIT_1。这将导致双方各发送一个 FIN，两个FIN经过网络传送后分别到达另一端。收到 FIN后，状态由FIN_WAIT_1变迁到CLOSING，并发送最后的 ACK。当收到最后的 ACK时，状态变化为TIME_WAIT。图18-19总结了这些状态的变化。

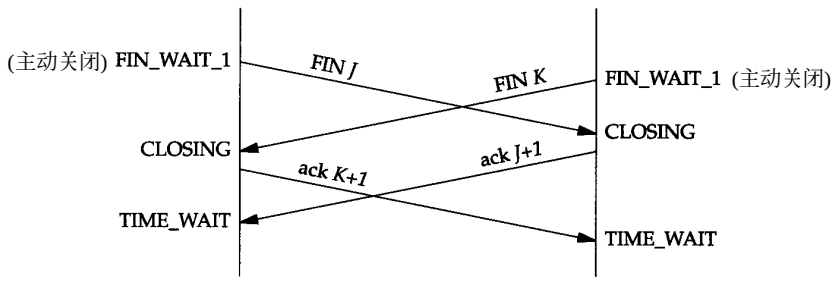


图18-19 同时关闭期间的报文段交换

同时关闭与正常关闭使用的段交换数目相同。

18.10 TCP 选项

TCP首部可以包含选项部分（图 17-2）。仅在最初的TCP规范中定义的选项是选项表结束、无操作和最大报文段长度。在我们的例子中，几乎每个 SYN报文段中我们都遇到过MSS选项。

新的RFC，主要是RFC 1323 [Jacobson, Braden和Borman 1992]，定义了新的TCP选项，

这些选项的大多数只在最新的 TCP实现中才能见到（我们将在第 24章介绍这些新选项）。图18-20显示了当前TCP选项的格式，这些选项的定义出自于 RFC 793和RFC 1323。

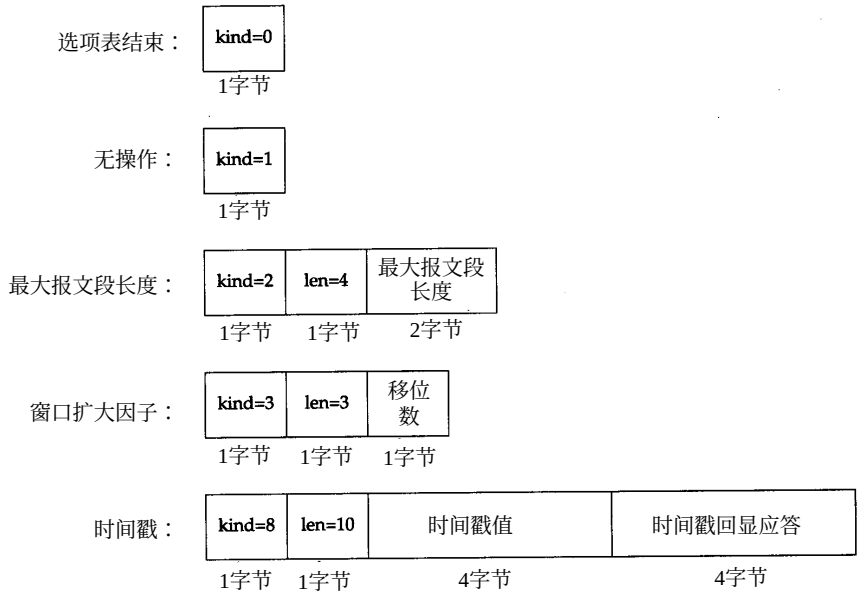


图18-20 TCP选项

每个选项的开始是1字节kind字段，说明选项的类型。kind字段为0和1的选项仅占1个字节。其他的选项在kind字节后还有len字节。它说明的长度是指总长度，包括 kind字节和len字节。

设置无操作选项的原因在于允许发方填充字段为 4字节的倍数。如果我们使用 4.4BSD系统进行初始化TCP连接，tcpdump将在初始的SYN上显示下面TCP选项：

<mss 512, nop, wscale 0, nop, nop, timestamp 146647 0>

MSS选项设置为512，后面是NOP，接着是窗口扩大选项。第一个 NOP用来将窗口扩大选项填充为4字节的边界。同样， 10字节的时间戳选项放在两个 NOP后，占12字节，同时使两个 4字节的时间戳满足4字节边界。

其他kind值为4、5、6和7的四个选项称为选择ACK及回显选项。由于回显选项已被时间戳选项取代，而目前定义的选择 ACK选项仍未定论，并未包括在RFC 1323中，因此图18-20没有将它们列出。另外，作为 TCP事务（第24.7节）的T/TCP建议也指明kind为11, 12和 13的三个选项。

18.11 TCP 服务器的设计

我们在1.8节说过大多数的TCP服务器进程是并发的。当一个新的连接请求到达服务器时，服务器接受这个请求，并调用一个新进程来处理这个新的客户请求。不同的操作系统使用不同的技术来调用新的服务器进程。在 Unix系统下，常用的技术是使用 `fork`函数来创建新的进程。如果系统支持，也可使用轻型进程，即线程（ thread）。

我们感兴趣的是 TCP与若干并发服务器的交互作用。需要回答下面的问题：当一个服务器进程接受一来自客户进程的服务请求时是如何处理端口的？如果多个连接请求几乎同时到

达会发生什么情况？

18.11.1 TCP服务器端口号

通过观察任何一个 TCP服务器，我们能了解 TCP如何处理端口号。我们使用 netstat 命令来观察 Telnet服务器。下面是在没有 Telnet连接时的显示（只留下显示 Telnet服务器的行）。

```
sun % netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  *.23                   *.*                    LISTEN
```

-a标志将显示网络中的所有主机端，而不仅仅是处于 ESTABLISHED的主机端。-n标志将以点分十进制的形式显示 IP地址，而不是通过 DNS将地址转化为主机名，同时还要求显示端口号（例如为 23）而不是服务名称（如 Telnet）。-f inet选项则仅要求显示使用 TCP或UDP的主机。

显示的本地地址为 *.23，星号通常又称为通配符。这表示传入的连接请求（即 SYN）将被任何一个本地接口所接收。如果该主机是多接口主机，我们将制定其中的一个 IP地址为本地IP地址，并且只接收来自这个接口的连接（在本节后面我们将看到这样的例子）。本地端口为23，这是Telnet的熟知端口号。

远端地址显示为 *.*，表示还不知道远端 IP地址和端口号，因为该端还处于 LISTEN状态，正等待连接请求的到达。

现在我们在主机 slip（140.252.13.65）启动一个 Telnet客户程序来连接这个 Telnet服务器。以下是netstat程序的输出行：

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23    140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

端口为23的第1行表示处于ESTABLISHED 状态的连接。另外还显示了这个连接的本地 IP地址、本地端口号、远端 IP地址和远端端口号。 本地IP地址为该连接请求到达的接口（以太网接口，140.252.13.33）。

处于LISTEN状态的服务器进程仍然存在。这个服务器进程是当前 Telnet服务器用于接收其他的连接请求。当传入的连接请求到达并被接收时，系统内核中的 TCP模块就创建一个处于ESTABLISHED状态的进程。另外，注意处于 ESTABLISHED状态的连接的端口不会变化：也是23，与处于LISTEN状态的进程相同。

现在我们在主机slip上启动另一个Telnet客户进程，并仍与这个 Telnet服务器进行连接。以下是netstat程序的输出行：

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23    140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23    140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

现在我们有两条从相同主机到相同服务器的处于 ESTABLISHED的连接。它们的本地端口号均为23。由于它们的远端端口号不同，这不会造成冲突。因为每个 Telnet客户进程要使用一个外

设端口，并且这个外设端口会选择为主机（slip）当前未曾使用的端口，因此它们的端口号肯定不同。

这个例子再次重申TCP使用由本地地址和远端地址组成的4元组：目的IP地址、目的端口号、源IP地址和源端口号来处理传入的多个连接请求。TCP仅通过目的端口号无法确定那个进程接收了一个连接请求。另外，在三个使用端口23的进程中，只有处于LISTEN的进程能够接收新的连接请求。处于ESTABLISHED的进程将不能接收SYN报文段，而处于LISTEN的进程将不能接收数据报文段。

下面我们从主机solaris上启动第3个Telnet客户进程，这个主机通过SLIP链路与主机sun相连，而不是以太网接口。

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.23       140.252.1.32.34603     ESTABLISHED
tcp        0      0  140.252.13.33.23      140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23      140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                  *.*                    LISTEN
```

现在第一个ESTABLISHED连接的本地IP地址对应多地址主机sun中的SLIP链路接口地址（140.252.1.29）。

18.11.2 限定的本地IP地址

我们来看看当服务器不能任选其本地IP地址而必须使用特定的IP地址时的情况。如果我们为sock程序指明一个IP地址（或主机名），并将它作为服务器，那么该IP地址就成为处于LISTEN服务器的本地IP地址。例如

```
sun % sock -s 140.252.1.29 8888
```

使这个服务器程序的连接仅局限于来自SLIP接口（140.252.1.29）。netstat的显示说明了这一点：

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

如果我们从主机solaris通过SLIP链路与这个服务器相连接，它将正常工作。

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     140.252.1.32.34614     ESTABLISHED
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

但如果我们试图从以太网（140.252.13）中的主机与这个服务器进行连接，连接请求将被TCP模块拒绝。如果使用tcpdump来观察这一切，对连接请求SYN的响应是一个如图18-21所示的RST。

```
1  0.0                    bsdi.1026 > sun.8888: S 3657920001:3657920001(0)
                                                  win 4096 <mss 1024>

2  0.000859 (0.0009)      sun.8888 > bsdi.1026: R 0:0(0) ack 3657920002 win 0
```

图18-21 具有限定本地IP地址服务器对连接请求的拒绝

这个连接请求将不会到达服务器的应用程序，因为它根据应用程序中指定的本地IP地址被内核中的TCP模块拒绝。

18.11.3 限定的远端IP地址

在11.12节，我们知道UDP服务器通常在指定IP本地地址和本地端口外，还能指定远端 IP地址和远端端口。RFC 793中显示的接口函数允许一个服务器在执行被动打开时，可指明远端插口（等待一个特定的客户执行主动打开），也可不指明远端插口（等待任何客户）。

遗憾的是，大多数API都不支持这么做。服务器必须不指明远端插口，而等待连接请求的到来，然后检查客户端的IP地址和端口号。

图18-22总结了TCP服务器进行连接时三种类型的地址绑定。在三种情况中， lport是服务器的熟知端口，而localIP必须是一个本地接口的IP地址。表中行的顺序正是TCP模块在收到一个连接请求时确定本地地址的顺序。最常使用的绑定（第 1行，如果支持的话）将最先尝试，最不常用的（最后一行两端的IP地址都没有制定）将最后尝试。

本地地址	远端地址	描　述
localIP.lport	*foreignIP.fport*	限制到一个客户进程(通常不支持)
localIP.lport	*.*	限制为到达一个本地接口：Local IP的连接
*.lport	*.*	接收发往Lport的所有连接

图18-22 TCP服务器本地和远端IP地址及端口号的规范

18.11.4 呼入连接请求队列

一个并发服务器调用一个新的进程来处理每个客户请求，因此处于被动连接请求的服务器应该始终准备处理下一个呼入的连接请求。那正是使用并发服务器的根本原因。但仍有可能出现当服务器在创建一个新的进程时，或操作系统正忙于处理优先级更高的进程时，到达多个连接请求。当服务器正处于忙时， TCP是如何处理这些呼入的连接请求？

在伯克利的TCP实现中采用以下规则：

1) 正等待连接请求的一端有一个固定长度的连接队列，该队列中的连接已被 TCP接受（即三次握手已经完成），但还没有被应用层所接受。

   注意区分TCP接受一个连接是将其放入这个队列，而应用层接受连接是将其从该队列中移出。

2) 应用层将指明该队列的最大长度，这个值通常称为积压值 (backlog)。它的取值范围是0~5之间的整数，包括0和5（大多数的应用程序都将这个值说明为 5）。

3) 当一个连接请求（即SYN）到达时， TCP使用一个算法，根据当前连接队列中的连接数来确定是否接收这个连接。我们期望应用层说明的积压值为这一端点所能允许接受连接的最大数目，但情况不是那么简单。图18-23显示了积压值与传统的伯克利系统和 Solaris 2.2所能允许的最大接受连接数之间的关系。

积压值	最大排队的连接数	
	传统的BSD	Solaris 2.2
0	1	0
1	2	1
2	4	2
3	5	3
4	7	4
5	8	5

图18-23 对正在听的端点所允许接受的最大连接数

注意，积压值说明的是 TCP监听的端点已被TCP接受而等待应用层接受的最大连接数。这个积压值对系统所允许的最大连接数，或者并发服务器所能并发处理的客户数，并无影响。

在这个图中，Solaris系统规定的值正如我们所期望的。而传统的BSD系统，将这个

值（由于某些原因）设置为积压值乘3除以2，再加1。

4) 如果对于新的连接请求，该 TCP监听的端点的连接队列中还有空间（基于图 18-23），TCP模块将对SYN进行确认并完成连接的建立。但应用层只有在三次握手中的第三个报文段收到后才会知道这个新连接时。另外，当客户进程的主动打开成功但服务器的应用层还不知道这个新的连接时，它可能会认为服务器进程已经准备好接收数据了（如果发生这种情况，服务器的TCP仅将接收的数据放入缓冲队列）。

5) 如果对于新的连接请求，连接队列中已没有空间， TCP将不理会收到的SYN。也不发回任何报文段（即不发回 RST）。如果应用层不能及时接受已被 TCP接受的连接，这些连接可能占满整个连接队列，客户的主动打开最终将超时。

通过sock程序能了解这种情况。我们调用它，并使用新的选项（ -O）。让它在创建一个新的服务器进程后而没有接受任何连接请求之前暂停下来。如果在它暂停期间又调用了多个客户进程，它将导致接受连接队列被填满，通过 tcpdump能够看到这一切。

```
bsdi % sock -s -v -q1 -O30 5555
```

-q1选项将服务器端的积压值置 1。在这种情况下，传统的 BSD系统中的队列允许接受两个连接请求（图 18-23）。-O30选项使程序在接受任何客户连接之前暂停 30秒。在这30秒内，我们可启动其他客户进程来填充这个队列。在主机 sun上启动4个客户进程。

图18-24显示了tcpdump的输出，首先是第 1个客户进程的第 1个SYN（省略窗口大小和MSS声明。当TCP连接建立时，将客户进程的端口号用粗体标出）。

端口为1090的第一个客户连接请求被 TCP接受（报文段1~3）。端口为1091的第2个客户连接请求也被 TCP接受（报文段 4~6）。而服务器的应用仍处于休眠状态，还未接受任何连接。目前的一切工作都由内核中的 TCP模块完成。另外，两个客户进程已经成功地完成了它们的主动打开，因为它们建立连接的三次握手已经完成。

```
 1   0.0                  sun.1090 > bsdi.7777: S 1617152000:1617152000(0)
 2   0.002310 ( 0.0023)   bsdi.7777 > sun.1090: S 4164096001:4164096001(0)
                                                  ack 1617152001
 3   0.003098 ( 0.0008)   sun.1090 > bsdi.7777: . ack 1

 4   4.291007 ( 4.2879)   sun.1091 > bsdi.7777: S 1617792000:1617792000(0)
 5   4.293349 ( 0.0023)   bsdi.7777 > sun.1091: S 4164672001:4164672001(0)
                                                  ack 1617792001
 6   4.294167 ( 0.0008)   sun.1091 > bsdi.7777: . ack 1

 7   7.131981 ( 2.8378)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
 8  10.556787 ( 3.4248)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
 9  12.695916 ( 2.1391)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
10  16.195772 ( 3.4999)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
11  24.695571 ( 8.4998)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)

12  28.195454 ( 3.4999)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
13  28.197810 ( 0.0024)   bsdi.7777 > sun.1093: S 4167808001:4167808001(0)
                                                  ack 1618688001
14  28.198639 ( 0.0008)   sun.1093 > bsdi.7777: . ack 1

15  48.694931 (20.4963)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
16  48.697292 ( 0.0024)   bsdi.7777 > sun.1092: S 4170496001:4170496001(0)
                                                  ack 1618176001
17  48.698145 ( 0.0009)   sun.1092 > bsdi.7777: . ack 1
```

图18-24 积压值例子的tcpdump 输出

我们接着在报文段7（端口1092）和报文段8（端口1093）启动第3和第4个客户进程。由于服务器的连接队列已满，TCP将不理会两个SYN。这两个客户进程在报文段9, 10, 11, 12, 15重发它们的SYN。第4个客户进程的第3个SYN重传被接受了，因为服务器程序的30秒休眠结束后，它将已接受的两个连接从队列中移出，使连接队列变空（服务器程序接收连接的时间是28.19，小于30的原因在于启动服务器程序后它需要几秒的时间来启动第1个客户进程(报文段1，显示的就是启动时间))。第3个客户进程的第4个SYN重传这时将被接受（报文段15~17)。服务器程序先接受第4个客户连接（端口1093）的原因是服务器程序30秒休眠与客户程序重传之间的定时交互作用。

我们期望接收连接队列按先进先出顺序传递给应用层。如TCP接受了端口为1090和1091的连接，我们希望应用层先接受端口为1090的连接，然后再接受端口为1091的连接。但许多伯克利的TCP实现都出现按后进先出的传递顺序，这个错误已存在了多年。产商最近已开始改正这个错误，但在如SunOS 4.13等系统中仍存在这个问题。

当队列已满时，TCP将不理会传入的SYN，也不发回RST作为应答，因为这是一个软错误，而不是一个硬错误。通常队列已满是由于应用程序或操作系统忙造成的，这样可防止应用程序对传入的连接进行服务。这个条件在一个很短的时间内可以改变。但如果服务器的TCP以系统复位作为响应，客户进程的主动打开将被废弃（如果服务器程序没有启动我们就会遇到)。由于不应答SYN，服务器程序迫使客户TCP随后重传SYN，以等待连接队列有空间接受新的连接。

这个例子中有一个巧妙之处，这在大多TCP/IP的具体实现中都能见到，就是如果服务器的连接队列未满时，TCP将接受传入的连接请求（即SYN)，但并不让应用层了解该连接源于何处（即不告知源IP地址和源端口)。这不是TCP所要求的，而只是共同的实现技术（如伯克利源代码通常都这么做)。如果一个API如TLI（见1.15节）向应用程序提供了解连接请求的到来的方法，并允许应用程序选择是否接受连接。当应用程序假定被告知连接请求已经到来时，TCP的三次握手已经结束！其他运输层的实现可能将连接请求的到达与接受分开（如OSI的运输层)，但TCP不是这样。

Solaris 2.2提供了一个选项使TCP只有在应用程序说可以接受（`tcp_eager_listeners`见E.4)，才允许接受传入的连接请求。

这种行为也意味着TCP服务器无法使客户进程的主动打开失效。当一个新的客户连接传递给服务器的应用程序时，TCP的三次握手就结束了，客户的主动打开已经完全成功。如果服务器的应用程序此时看到客户的IP地址和端口号，并决定是否为该客户进行服务，服务器所能做的就是关闭连接（发送FIN)，或者复位连接（发送RST)。无论哪种情况，客户进程都认为一切正常，因为它的主动打开已经完成，并且已经向服务器程序发送过请求。

18.12 小结

两个进程在使用TCP交换数据之前，它们之间必须建立一条连接。完成后，要关闭这个连接。本章已经详细介绍了如何使用三次握手来建立连接以及使用4个报文段来关闭连接。

我们用`tcpdump`程序显示了TCP首部中的各个字段。也了解了连接建立是如何超时，连

接复位是如何发送，使用半打开连接发生的情况以及 TCP是如何提供半关闭、同时打开和同时关闭。

弄清TCP操作的关键在于它的状态变迁图。我们跟踪了连接建立与关闭的步骤以及它们的状态变迁过程。还讨论了在设计TCP并发服务器时TCP连接建立的具体实现方法。

一个TCP连接由一个4元组唯一确定：本地IP地址、本地端口号、远端IP地址和远端端口号。无论何时关闭一个连接，一端必须保持这个连接，我们看到 TIME_WAIT状态将处理这个问题。处理的原则是执行主动打开的一端在进入这个状态时要保持的时间为 TCP实现中规定的MSL值的两倍。

习题

18.1 在18.2节我们说初始序号（ISN）正常情况下由1开始，并且每0.5秒增加64000，每次执行一个主动打开。这意味着 ISN的最低三位通常总是 001。但在图18-3中，两个方向上ISN中的最低三位都是521。究竟是怎么回事？

18.2 在图18-15中，我们键入12个字符，看到TCP发送了13个字节。在图18-16中我们键入8个字符，但TCP发送了10个字符。为什么在第1种情况下增加1个字节，而在第2种情况下增加2个字节？

18.3 半打开连接和半关闭连接的区别是什么？

18.4 如果启动sock程序作为一个服务器程序，然后终止它（还没有客户进程与它相连接），我们能立即重新启动这个服务器程序。这意味着它没有经历 2MSL等待状态。用状态变迁来解释这一切。

18.5 在18.6节我们知道一个客户进程不能重新使用同一个本地端口，如果该端口是仍处于2MSL等待连接的一部分。但如果 sock程序作为客户程序连续运行两次，并且连接到daytime服务器上，我们就能重新使用同一本地端口。另外，对一个仍处于 2MSL等待的连接，也能为它创建一个替身。这将如何做？

```
sun % sock -v bsdi daytime
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:54:51 1993
connection closed by peer
sun % sock -v -b1163 bsdi daytime      重用相同的本地端口号
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:55:01 1993
connection closed by peer
```

18.6 在18.6节的最后，我们介绍了 FIN_WAIT_2状态，提到如果应用程序仅过 11分钟后实行完全关闭（不是半关闭），许多具体的实现都将一个连接由这个状态转移到 CLOSED状态。如果另一端（处于CLOSE_WAIT状态）在宣布关闭（即发送 FIN）之前等待了12分钟，这一端的TCP将如何响应这个FIN？

18.7 对于一个电话交谈，哪一方是主动打开，哪一方是被动打开？是否允许同时打开？是否允许同时关闭？

18.8 在图18-6中，我们没有见到一个ARP请求或一个ARP应答。显然主机svr4的硬件地址一定在bsdi的ARP高速缓存中。如果这个ARP高速缓存不存在，这个图会有什么变化？

18.9 解释如下的tcpdump输出，并和图18-13进行比较。

```
1  0.0                 solaris.32990 > bsdi.discard: S 40140288:40140288(0)
                                                     win 8760 <mss 1460>
2  0.003295 (0.0033)   bsdi.discard > solaris.32990: S 4208081409:4208081409(0)
                                                     ack 40140289 win 4096
                                                     <mss 1024>
3  0.419991 (0.4167)   solaris.32990 > bsdi.discard: P 1:257(256) ack 1 win 9216
4  0.449852 (0.0299)   solaris.32990 > bsdi.discard: F 257:257(0) ack 1 win 9216
5  0.451965 (0.0021)   bsdi.discard > solaris.32990: . ack 258 win 3840
6  0.464569 (0.0126)   bsdi.discard > solaris.32990: F 1:1(0) ack 258 win 4096
7  0.720031 (0.2555)   solaris.32990 > bsdi.discard: . ack 2 win 9216
```

18.10 为什么图18-4中的服务器不将对客户FIN的ACK与自己的FIN合并，从而将报文段数减少为3个？

18.11 在图18-16中，RST的序号为什么是26368002?

18.12 TCP向链路层查询MTU是否违反分层的规则？

18.13 假定在图14.16中，每个DNS使用TCP而不是UDP进行查询，试问需要交换多少个报文段？

18.14 假定MSL为120秒，试问系统能够初始化一个新连接然后进行主动关闭的最大速率是多少？

18.15 阅读RFC 793，分析处于TIME_WAIT状态的主机收到使其进入此状态的重复的 FIN时所发生的情况。

18.16 阅读RFC 793，分析处于TIME_WAIT状态的主机收到一个RST时所发生的情况。

18.17 阅读Host Requirements RFC并找出半双工TCP关闭的定义。

18.18 在图1-8中，我们曾提到到来的TCP报文段可根据其目的端口号进行分用，请问这种说法是否正确？

第19章 TCP的交互数据流

19.1 引言

前一章我们介绍了TCP连接的建立与释放，现在来介绍使用TCP进行数据传输的有关问题。

一些有关TCP通信量的研究如[Caceres et al. 1991]发现，如果按照分组数量计算，约有一半的TCP报文段包含成块数据（如FTP、电子邮件和Usenet新闻），另一半则包含交互数据（如Telnet和Rlogin）。如果按字节计算，则成块数据与交互数据的比例约为90%和10%。这是因为成块数据的报文段基本上都是满长度（full-sized）的（通常为512字节的用户数据），而交互数据则小得多（上述研究表明Telnet和Rlogin分组中通常约90%左右的用户数据小于10个字节）。

很明显，TCP需要同时处理这两类数据，但使用的处理算法则有所不同。本章将以Rlogin应用为例来观察交互数据的传输过程。将揭示经受时延的确认是如何工作的以及Nagle算法怎样减少了通过广域网络传输的小分组的数目，这些算法也同样适用于Telnet应用。下一章我们将介绍成块数据的传输问题。

19.2 交互式输入

首先来观察在一个Rlogin连接上键入一个交互命令时所产生的数据流。许多TCP/IP的初学者很吃惊地发现通常每一个交互按键都会产生一个数据分组，也就是说，每次从客户传到服务器的是一个字节的按键（而不是每次一行）。而且，Rlogin需要远程系统（服务器）回显我们（客户）键入的字符。这样就会产生4个报文段：（1）来自客户的交互按键；（2）来自服务器的按键确认；（3）来自服务器的按键回显；（4）来自客户的按键回显确认。图19-1表示了这个数据流。

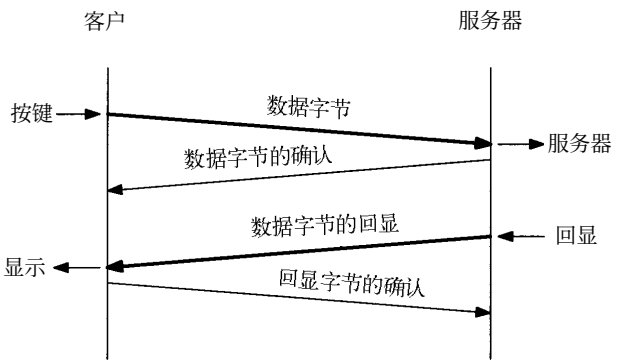


图19-1 一种可能的处理远程交互按键回显的方法

然而，我们一般可以将报文段2和3进行合并——按键确认与按键回显一起发送。下一节将描述这种合并的技术（称为经受时延的确认）。

本章我们特意使用Rlogin作为例子，因为它每次总是从客户发送一个字节到服务器。在第26章讲到Telnet的时候，将会发现它有一个选项允许客户发送一行到服务器，通过使用这个选项可以减少网络的负载。

图19-2显示的是当我们键入5个字符`date\n`时的数据流（我们没有显示连接建立的过程，并且去掉了所有的服务类型输出。BSD/386通过设置一个Rlogin连接的TOS来获得最小时延）。

第1行客户发送字符d到服务器。第2行是该字符的确认及回显（也就是图 19-1的中间两部分数据的合并）。第3行是回显字符的确认。与字符a有关的是第4~6行，与字符t有关的是第7~9行，第10~12行与字符e有关。第3~4、6~7、9~10和12~13行之间半秒左右的时间差是键入两个字符之间的时延。

注意到13~15行稍有不同。从客户发送到服务器的是一个字符（按下 RETURN键后产生的UNIX系统中的换行符），而回显的则是两个字符。这两个字符分别是回车和换行字符（CR/LF），它们的作用是将光标回移到左边并移动到下一行。

第16行是来自服务器的date命令的输出。这30个字节由28个字符与最后的CR/LF组成。紧接着从服务器发往客户的7个字符（第18行）是在服务器主机上的客户提示符： svr4 % 。第19行确认了这7个字符。

```
 1  0.0                bsdi.1023 > svr4.login: P 0:1(1) ack 1 win 4096
 2  0.016497 (0.0165)  svr4.login > bsdi.1023: P 1:2(1) ack 1 win 4096
 3  0.139955 (0.1235)  bsdi.1023 > svr4.login: . ack 2 win 4096

 4  0.458037 (0.3181)  bsdi.1023 > svr4.login: P 1:2(1) ack 2 win 4096
 5  0.474386 (0.0163)  svr4.login > bsdi.1023: P 2:3(1) ack 2 win 4096
 6  0.539943 (0.0656)  bsdi.1023 > svr4.login: . ack 3 win 4096

 7  0.814582 (0.2746)  bsdi.1023 > svr4.login: P 2:3(1) ack 3 win 4096
 8  0.831108 (0.0165)  svr4.login > bsdi.1023: P 3:4(1) ack 3 win 4096
 9  0.940112 (0.1090)  bsdi.1023 > svr4.login: . ack 4 win 4096

10  1.191287 (0.2512)  bsdi.1023 > svr4.login: P 3:4(1) ack 4 win 4096
11  1.207701 (0.0164)  svr4.login > bsdi.1023: P 4:5(1) ack 4 win 4096
12  1.339994 (0.1323)  bsdi.1023 > svr4.login: . ack 5 win 4096

13  1.680646 (0.3407)  bsdi.1023 > svr4.login: P 4:5(1) ack 5 win 4096
14  1.697977 (0.0173)  svr4.login > bsdi.1023: P 5:7(2) ack 5 win 4096
15  1.739974 (0.0420)  bsdi.1023 > svr4.login: . ack 7 win 4096

16  1.799841 (0.0599)  svr4.login > bsdi.1023: P 7:37(30) ack 5 win 4096
17  1.940176 (0.1403)  bsdi.1023 > svr4.login: . ack 37 win 4096
18  1.944338 (0.0042)  svr4.login > bsdi.1023: P 37:44(7) ack 5 win 4096
19  2.140110 (0.1958)  bsdi.1023 > svr4.login: . ack 44 win 4096
```

图19-2 当在Rlogin连接上键入date时的数据流

注意TCP是怎样进行确认的。第 1行以序号0发送数据字节，第 2行通过将确认序号设为 1，也就是最后成功收到的字节的序号加 1，来对其进行确认（也就是所谓的下一个期望数据的序号）。在第2行中服务器还向客户发送了一序号为 1的数据，客户在第 3行中通过设置确认序号为2来对该数据进行确认。

19.3 经受时延的确认

在图19-2中有一些与本节将要论及的时间有关的细微之处。图 19-3表示了图19-2中数据交换的时间系列（在该时间系列中，去掉了所有的窗口通告，并增加了一个记号来表明正在传输何种数据）。

把从bsdi发送到srv4的7个ACK标记为经受时延的ACK。通常TCP在接收到数据时并不立即发送ACK；相反，它推迟发送，以便将 ACK与需要沿该方向发送的数据一起发送（有时称这种现象为数据捎带 ACK）。绝大多数实现采用的时延为 200 ms，也就是说，TCP将以最大200 ms的时延等待是否有数据一起发送。

如果观察bsdi接收到数据和发送ACK之间的时间差，就会发现它们似乎是随机的： 123.5、

65.6、109.0、132.2、42.0、140.3和195.8 ms。相反，观察到发送ACK的实际时间（从0开始）为：139.9、539.3、940.1、1339.9、1739.9、1940.1和2140.1 ms（在图19-3中用星号标出）。这些时间之间的差则是200 ms的整数倍，这里所发生的情况是因为TCP使用了一个200 ms的定时器，该定时器以相对于内核引导的200 ms固定时间溢出。由于将要确认的数据是随机到达的（在时刻16.4, 474.3, 831.1等），TCP在内核的200 ms定时器的下一次溢出时得到通知。这有可能是将来1~200 ms中的任何一刻。

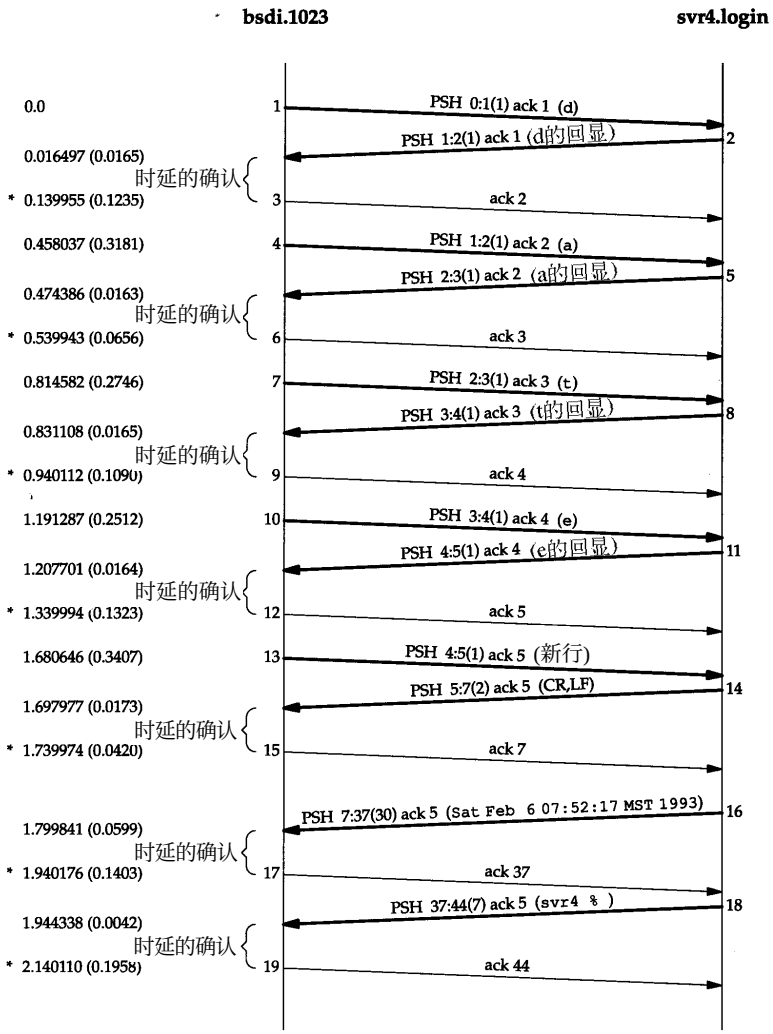


图19-3　在rlogin连接上键入date命令时的数据流时间系列

如果观察svr4为产生所收到的每个字符的回显所使用的时间，则这些时间分别为16.5、16.3、16.5、16.4和17.3 ms。由于这个时间小于200 ms，因此我们在另一端从来没有观察到一个经受时延的ACK。在经受时延的定时器溢出前总是有数据需要发送（如果有一个约为16 ms等待时间越过了内核的200 ms时钟滴答的边界，则仍可以看到一个经受时延的ACK。在本例中我们一个也没有看到）。

在图18-7中，当为检测超时而使用500 ms的TCP定时器时，我们会看到同样的情况。这两

个200 ms和500 ms的定时器都在相对于内核引导的时间处溢出。不论 TCP何时设置一个定时器，该定时器都可能在将来1~200 ms和1~500 ms的任一处溢出。

Host Requirements RFC声明TCP需要实现一个经受时延的ACK，但时延必须小于500 ms。

19.4 Nagle算法

在前一节我们看到，在一个Rlogin连接上客户一般每次发送一个字节到服务器，这就产生了一些41字节长的分组：20字节的IP首部、20字节的TCP首部和1个字节的数据。在局域网上，这些小分组（被称为微小分组（ tinygram））通常不会引起麻烦，因为局域网一般不会出现拥塞。但在广域网上，这些小分组则会增加拥塞出现的可能。一种简单和好的方法就是采用RFC 896 [Nagle 1984]中所建议的Nagle算法。

该算法要求一个TCP连接上最多只能有一个未被确认的未完成的小分组，在该分组的确认到达之前不能发送其他的小分组。相反， TCP收集这些少量的分组，并在确认到来时以一个分组的方式发出去。该算法的优越之处在于它是自适应的：确认到达得越快，数据也就发送得越快。而在希望减少微小分组数目的低速广域网上，则会发送更少的分组（我们将在22.3节看到“小”的含义是小于报文段的大小）。

在图19-3中可以看到，在以太网上一个字节被发送、确认和回显的平均往返时间约为 16 ms。为了产生比这个速度更快的数据，我们每秒键入的字符必须多于 60个。这表明在局域网环境下两个主机之间发送数据时很少使用这个算法。

但是，当往返时间（ RTT）增加时，如通过一个广域网，情况就会发生变化。看一下在主机`slip`和主机`vangogh.cs.berkeley.edu`之间的Rlogin连接工作的情况。为了从我们的网络中出去（参看原书封面内侧），需要使用两个SLIP链路和Internet。我们希望获得更长的往返时间。图19-4显示了当在客户端快速键入字符（像一个快速打字员一样）时一些数据流的时间系列（去掉了服务类型信息，但保留了窗口通告）。

比较图19-4与图19-3，我们首先注意到从`slip`到`vangogh`不存在经受时延的ACK。这是因为在时延定时器溢出之前总是有数据等待发送。

其次，注意到从左到右待发数据的长度是不同的，分别为： 1、1、2、1、2、2、3、1和3个字节。这是因为客户只有收到前一个数据的确认后才发送已经收集的数据。通过使用 Nagle算法，为发送16个字节的数据客户只需要使用9个报文段，而不再是16个。

报文段14和15看起来似乎是与Nagle算法相违背的，但我们需要通过检查序号来观察其中的真相。因为确认序号是 54，因此报文段14是报文段12中确认的应答。但客户在发送该报文段之前，接收到了来自服务器的报文段 13，报文段15中包含了对序号为56的报文段13的确认。因此即使我们看到从客户到服务器有两个连续返回的报文段，客户也是遵守了 Nagle算法的。

在图19-4中可以看到存在一个经受时延的 ACK，但该ACK是从服务器到客户的（报文段12），因为它不包含任何数据，因此我们可以假定这是经受时延的 ACK。服务器当时一定非常忙，因此无法在服务器的定时器溢出前及时处理所收到的字符。

最后看一下最后两个报文段中数据的数量以及相应的序号。客户发送 3个字节的数据（18, 19和20），然后服务器确认这3个字节（最后的报文段中的 ACK 21），但是只返回了一个字节（标号为59）。这是因为当服务器的TCP一旦正确收到这3个字节的数据，就会返回对该数据的确

认，但只有当Rlogin服务器发送回显数据时，它才能够发送这些数据的回显。这表明 TCP可以在应用读取并处理数据前发送所接收数据的确认。 TCP确认仅仅表明TCP已经正确接收了数据。最后一个报文段的窗口大小为8189而非8192，表明服务器进程尚未读取这三个收到的数据。

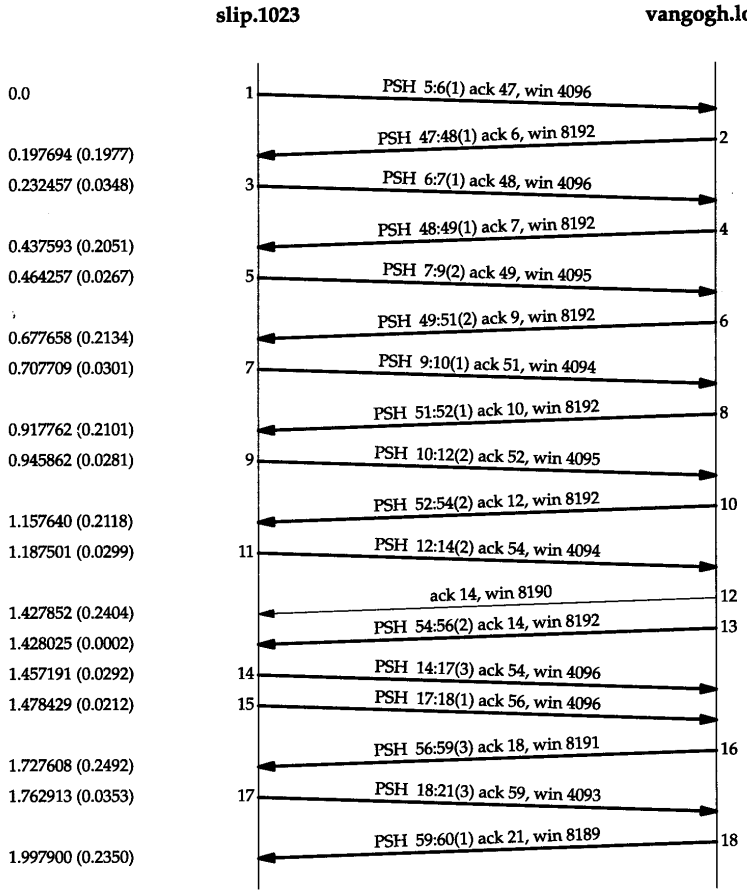


图19-4 在slip 和vangogh.cs.berkeley.edu 之间使用rlogin 时的数据流

19.4.1 关闭Nagle算法

有时我们也需要关闭Nagle算法。一个典型的例子是X窗口系统服务器（见30.5节）：小消息（鼠标移动）必须无时延地发送，以便为进行某种操作的交互用户提供实时的反馈。

这里将举另外一个更容易说明的例子——在一个交互注册过程中键入终端的一个特殊功能键。这个功能键通常可以产生多个字符序列，经常从 ASCII码的转义(escape)字符开始。如果TCP每次得到一个字符，它很可能会发送序列中的第一个字符（ ASCII码的ESC），然后缓存其他字符并等待对该字符的确认。但当服务器接收到该字符后，它并不发送确认，而是继续等待接收序列中的其他字符。这就会经常触发服务器的经受时延的确认算法，表示剩下的字符没有在200 ms内发送。对交互用户而言，这将产生明显的时延。

插口API用户可以使用TCP_NODELAY选项来关闭Nagle算法。

Host Requirements RFC声明TCP必须实现Nagle算法，但必须为应用提供一种方法来关闭该算法在某个连接上执行。

19.4.2 一个例子

可以在Nagle算法和产生多个字符的按键之间看到这种交互的情况。在主机 slip和主机vangogh.cs.berkeley.edu之间建立一个Rlogin连接，然后按下F1功能键，这将产生3个字节：一个escape、一个左括号和一个M。然后再按下F2功能键，这将产生另外3个字节。图19-5表示的是tcpdump的输出结果（我们去掉了其中的服务类型和窗口通告）。

```
                                按F1键
1   0.0                         slip.1023 > vangogh.login: P 1:2(1) ack 2
2   0.250520 (0.2505)           vangogh.login > slip.1023: P 2:4(2) ack 2
3   0.251709 (0.0012)           slip.1023 > vangogh.login: P 2:4(2) ack 4
4   0.490344 (0.2386)           vangogh.login > slip.1023: P 4:6(2) ack 4
5   0.588694 (0.0984)           slip.1023 > vangogh.login: . ack 6
                                按F2键
6   2.836830 (2.2481)           slip.1023 > vangogh.login: P 4:5(1) ack 6
7   3.132388 (0.2956)           vangogh.login > slip.1023: P 6:8(2) ack 5
8   3.133573 (0.0012)           slip.1023 > vangogh.login: P 5:7(2) ack 8
9   3.370346 (0.2368)           vangogh.login > slip.1023: P 8:10(2) ack 7
10  3.388692 (0.0183)           slip.1023 > vangogh.login: . ack 10
```

图19-5 当键入能够产生多个字节数据的字符时Nagle算法的观察情况

图19-6表示了这个交互过程的时间系列。在该图的下面部分我们给出了从客户发送到服务器的6个字节和它们的序号以及将要返回的8个字节的回显。

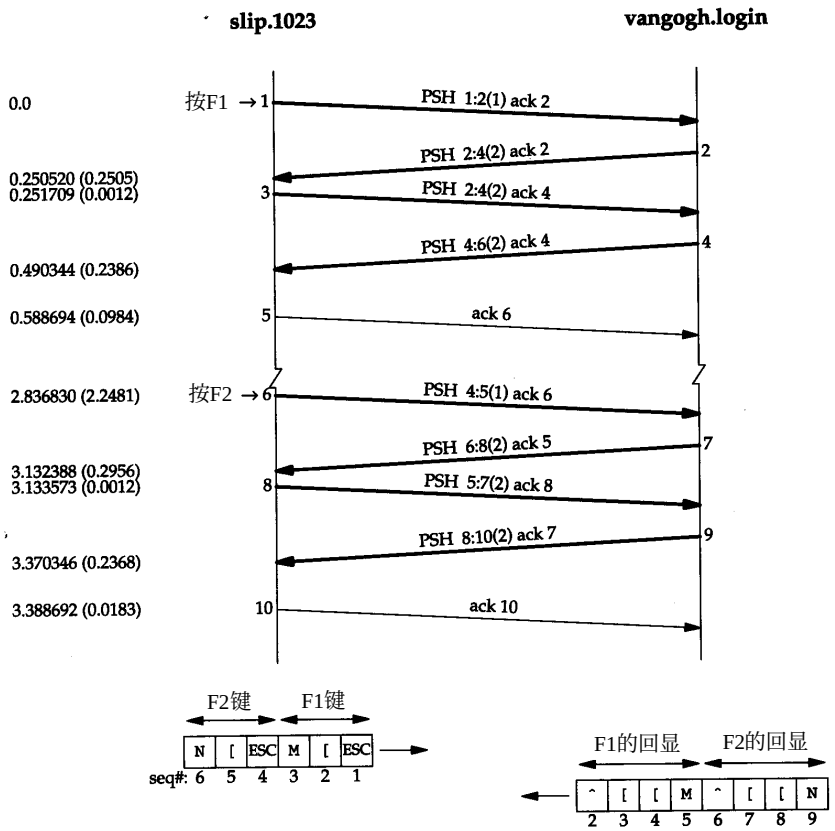


图19-6 图19-5的时间系列（Nagle算法的观察结果）

当rlogin客户读取到输入的第1个字节并向TCP写入时，该字节作为报文段1被发送。这是F1键所产生的3个字节中的第1个。它的回显在报文段2中被返回，此时剩余的2个字节才被发送（报文段3）。这两个字节的回显在报文段4被接收，而报文段5则是对它们的确认。

第1个字节的回显为2个字节（报文段2）的原因是因为在ASCII码中转义符的回显是2个字节：插入记号和一个左括号。剩下的两个输入字节：一个左括号和一个 M，分别以自身作为回显内容。

当按下下一个特殊功能键（报文段 6~10）时，也会发生同样的过程。正如我们希望的那样，在报文段5和10（slip发送回显的确认）之间的时间差是 200 ms的整数倍，因为这两个ACK被进行时延。

现在我们使用一个修改后关闭了 Nagle算法的rlogin版本重复同样的实验。图 19-7显示了tcpdump的输出结果（同样去掉了其中的服务类型和窗口通告）。

```
                                按F1键
 1  0.0                    slip.1023 > vangogh.login: P 1:2(1) ack 2
 2  0.002163 (0.0022)      slip.1023 > vangogh.login: P 2:3(1) ack 2
 3  0.004218 (0.0021)      slip.1023 > vangogh.login: P 3:4(1) ack 2
 4  0.280621 (0.2764)      vangogh.login > slip.1023: P 5:6(1) ack 4
 5  0.281738 (0.0011)      slip.1023 > vangogh.login: . ack 2
 6  2.477561 (2.1958)      vangogh.login > slip.1023: P 2:6(4) ack 4
 7  2.478735 (0.0012)      slip.1023 > vangogh.login: . ack 6
                                按F2键
 8  3.217023 (0.7383)      slip.1023 > vangogh.login: P 4:5(1) ack 6
 9  3.219165 (0.0021)      slip.1023 > vangogh.login: P 5:6(1) ack 6
10  3.221688 (0.0025)      slip.1023 > vangogh.login: P 6:7(1) ack 6
11  3.460626 (0.2389)      vangogh.login > slip.1023: P 6:8(2) ack 5
12  3.489414 (0.0288)      vangogh.login > slip.1023: P 8:10(2) ack 7
13  3.640356 (0.1509)      slip.1023 > vangogh.login: . ack 10
```

图19-7　在一个Rlogin会话中关闭Nagle算法

在已知某些报文段在网络上形成交叉的情况下，以该结果构造时间系列则更具有启发性和指导意义。这个例子同样也需要随着数据流对序号进行仔细的检查。在图 19-8中显示这个结果。用图19-7中tcpdump输出的号码对报文段进行了相应的编号。

我们注意到的第1个变化是当3个字节准备好时它们全部被发送（报文段 1、2和3）。没有时延发生 —— Nagle算法被禁止。

在tcpdump输出中的下一个分组（报文段 4）中带有来自服务器的第 5个字节及一个确认序号为4的ACK。这是不正确的，因为客户并不希望接收到第 5个字节，因此它立即发送一个确认序号为2而不是6的响应（没有被延迟）。看起来一个报文段丢失了，在图 19-8中我们用虚线表示。

如何知道这个丢失的报文段中包含第 2、3和4个字节，且其确认序号为 3呢？这是因为正如在报文段5中声明的那样，我们希望的下一个字节是第 2个字节（每当TCP接收到一个超出期望序号的失序数据时，它总是发送一个确认序号为其期望序号的确认）。也正是因为丢失的分组中包含第2、3和4个字节，表明服务器必定已经接收到报文段 2，因此丢失的报文段中的确认序号一定为3（服务器期望接收的下一个字节号）。最后，注意到重传的报文段6中包含有丢失的报文段中的数据和报文段 4，这被称为重新分组化。我们将在 22.11节对其进行更多的介绍。

现在回到禁止Nagle算法的讨论中来。可以观察到键入的下一个特殊功能键所产生的 3个字节分别作为单独的报文段（报文段8、9和10）被发送。这一次服务器首先回显了报文段8中的字节（报文段11），然后回显了报文段9和10中的字节（报文段12）。

在这个例子中，我们能够观察到的是在跨广域网运行一个交互应用的环境下，当进行多字节的按键输入时，默认使用Nagle算法会引起额外的时延。

在第21章我们将进行有关时延和重传方面的讨论。

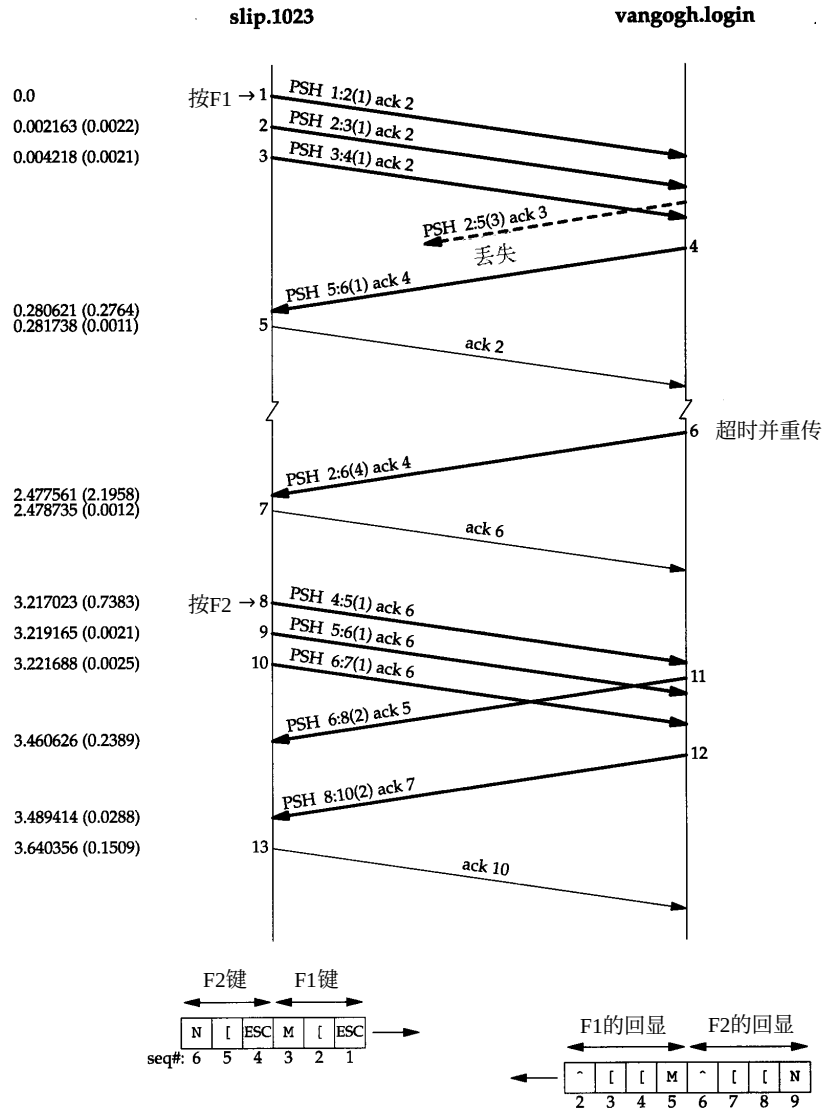


图19-8 图19-7的时间系列（关闭Nagle算法）

19.5 窗口大小通告

在图19-4中，我们可以观察到slip通告窗口大小为4096字节，而vangogh通告其窗口大小为8192个字节。该图中的大多数报文段都包含这两个值中的一个。

然而，报文段5通告的窗口大小为4095个字节，这意味着在TCP的缓冲区中仍然有一个字节等待应用程序（ Rlogin客户）读取。同样，来自客户的下一个报文段声明其窗口大小为4094个字节，这说明仍有两个字节等待读取。

服务器通常通告窗口大小为 8192个字节，这是因为服务器在读取并回显接收到的数据之前，其TCP没有数据发送。当服务器已经读取了来自客户的输入后，来自服务器的数据将被发送。

然而，在ACK到来时，客户的TCP总是有数据需要发送。这是因为它在等待 ACK的过程中缓存接收到的字符。当客户 TCP发送缓存的数据时， Rlogin客户没有机会读取来自服务器的数据，因此，客户通告的窗口大小总是小于 4096。

19.6 小结

交互数据总是以小于最大报文段长度的分组发送。在 Rlogin中通常只有一个字节从客户发送到服务器。Telnet允许一次发送一行输入数据，但是目前大多数实现仍然发送一个字节。

对于这些小的报文段，接收方使用经受时延的确认方法来判断确认是否可被推迟发送，以便与回送数据一起发送。这样通常会减少报文段的数目，尤其是对于需要回显用户输入字符的Rlogin会话。

在较慢的广域网环境中，通常使用 Nagle算法来减少这些小报文段的数目。这个算法限制发送者任何时候只能有一个发送的小报文段未被确认。但我们给出的一个例子也表明有时需要禁止Nagle算法的功能。

习题

19.1 考虑一个TCP客户应用程序，它发送一个小应用程序首部（ 8个字节）和一个小请求（12个字节），然后等待来自服务器的一个应答。比较以下两种方式发送请求时的处理情况：先发送8个字节再发送12个字节和一次发送20个字节。

19.2 图19-4中我们在路由器sun上运行tcpdump。这意味着从右至左的箭头中的数据也需要经过bsdi，同时从左至右的箭头中的数据已经流经 bsdi。当观察一个送往slip的报文段及下一个来自 slip的报文段时，我们发现它们之间的时间差分别为： 34.8、26.7、30.1、28.1、29.9和35.3 ms。现给定在sun和slip之间存在两条链路（一个以太链路和一个9600 b/s的CSLIP链路），试问这些时间差的含义（提示：重新阅读 2.10节）。

19.3 比较在使用Nagle算法（图19-6）和禁止Nagle算法（图19-8）的情况下发送一个特殊功能键并等待其应答所需要的时间。

第20章 TCP的成块数据流

20.1 引言

在第15章我们看到TFTP使用了停止等待协议。数据发送方在发送下一个数据块之前需要等待接收对已发送数据的确认。本章我们将介绍 TCP所使用的被称为滑动窗口协议的另一种形式的流量控制方法。该协议允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。

我们还将介绍TCP的PUSH标志，该标志在前面的许多例子中都出现过。此外，我们还要介绍慢启动，TCP使用该技术在一个连接上建立数据流，最后介绍成块数据流的吞吐量。

20.2 正常数据流

我们以从主机svr4单向传输8192个字节到主机bsdi开始。在bsdi上运行sock程序作为服务器：

```
bsdi %sock -i -s 7777
```

其中，标志-i和-s指示程序作为一个“吸收（sink）”服务器运行（从网络上读取并丢弃数据），服务器端口指明为7777。相应的客户程序运行为：

```
svr4 %sock -i -n8 bsdi 7777
```

该命令指示客户向网络发送8个1024字节的数据。图20-1显示了这个过程的时间系列。我们在输出的前3个报文段中显示了每一端MSS的值。

发送方首先传送3个数据报文段（4~6）。下一个报文段（7）仅确认了前两个数据报文段，这可以从其确认序号为2048而不是3073看出来。

报文段7的ACK的序号之所以是2048而不是3073是由以下原因造成的：当一个分组到达时，它首先被设备中断例程进行处理，然后放置到IP的输入队列中。三个报文段4、5和6依次到达并按接收顺序放到IP的输入队列。IP将按同样顺序将它们交给TCP。当TCP处理报文段4时，该连接被标记为产生一个经受时延的确认。TCP处理下一报文段（5），由于TCP现在有两个未完成的报文段需要确认，因此产生一个序号为2048的ACK（报文段7），并清除该连接产生经受时延的确认标志。TCP处理下一个报文段（6），而连接又被标志为产生一个经受时延的确认。在报文段9到来之前，由于时延定时器溢出，因此产生一个序号为3073的ACK（报文段8）。报文段8中的窗口大小为3072，表明在TCP的接收缓存中还有1024个字节的数据等待被应用程序读取。

报文段11~16说明了通常使用的“隔一个报文段确认”的策略。报文段11、12和13到达并被放入IP的接收队列。当报文段11被处理时，连接被标记为产生一个经受时延的确认。当报文段12被处理时，它们的ACK（报文段14）被产生且连接的经受时延的确认标志被清除。报文段13使得连接再次被标记为产生经受时延。但在时延定时器溢出之前，报文段15处理完毕，因此该确认立刻被发送。

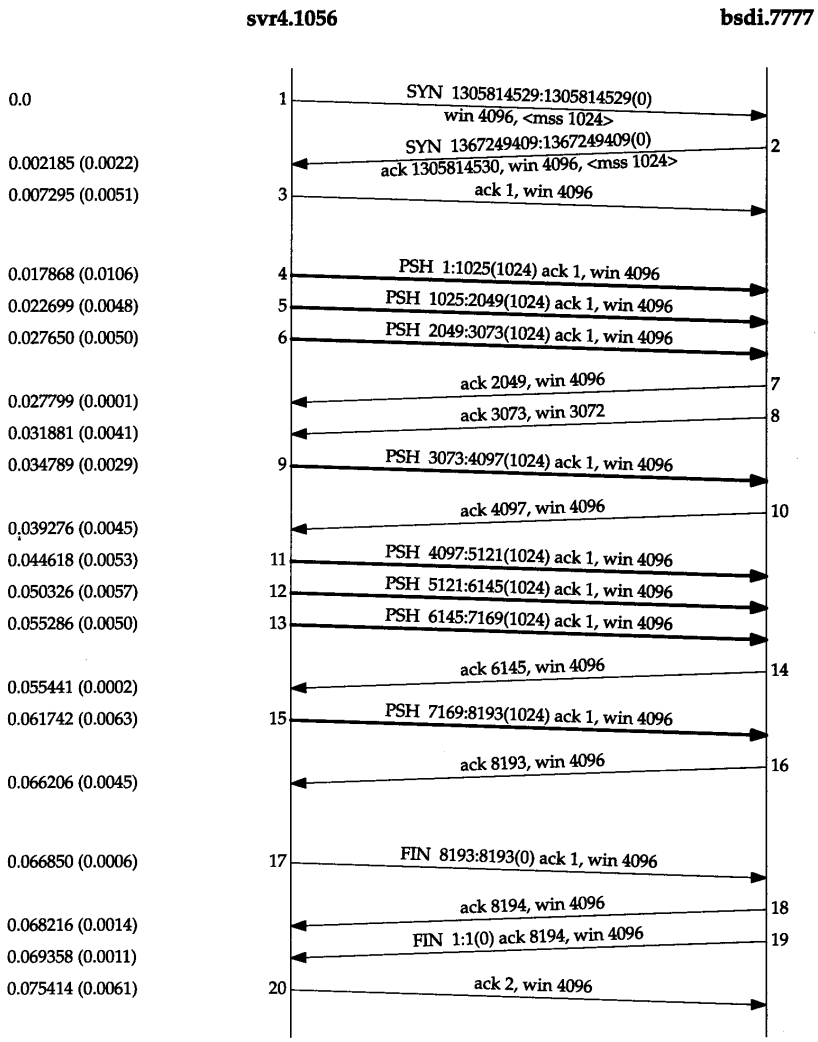


图20-1 从svr4 传输8192个字节到bsdi

注意到报文段7、14和16中的ACK确认了两个收到的报文段是很重要的。使用 TCP的滑动窗口协议时，接收方不必确认每一个收到的分组。在 TCP中，ACK是累积的——它们表示接收方已经正确收到了一直到确认序号减 1的所有字节。在本例中，三个确认的数据为 2048字节而两个确认的数据为1024字节（忽略了连接建立和终止中的确认）。

用tcpdump看到的是TCP的动态活动情况。我们在线路上看到的分组顺序依赖于许多无法控制的因素：发送方TCP的实现、接收方TCP的实现、接收进程读取数据（依赖于操作系统的调度）和网络的动态性（如以太网的冲突和退避等）。对这两个TCP而言，没有一种单一的、正确的方法来交换给定数量的数据。

为显示情况可能怎样变化，图 20-2显示了在同样两个主机之间交换同样数据时的另一个时间系列，它们是在图20-1所示的几分钟之后截获的。

一些情况发生了变化。这一次接收方没有发送一个序号为 3073的ACK，而是等待并发送序号为4097的ACK。接收方仅发送了4个ACK（报文段7、10、12和15）：三个确认了2048字

节，另一个确认了1024字节。最后1024字节数据的ACK出现在报文段17中，它与FIN的ACK一道发送（比较该图中的报文段17与图20-1中的报文段16和18）。

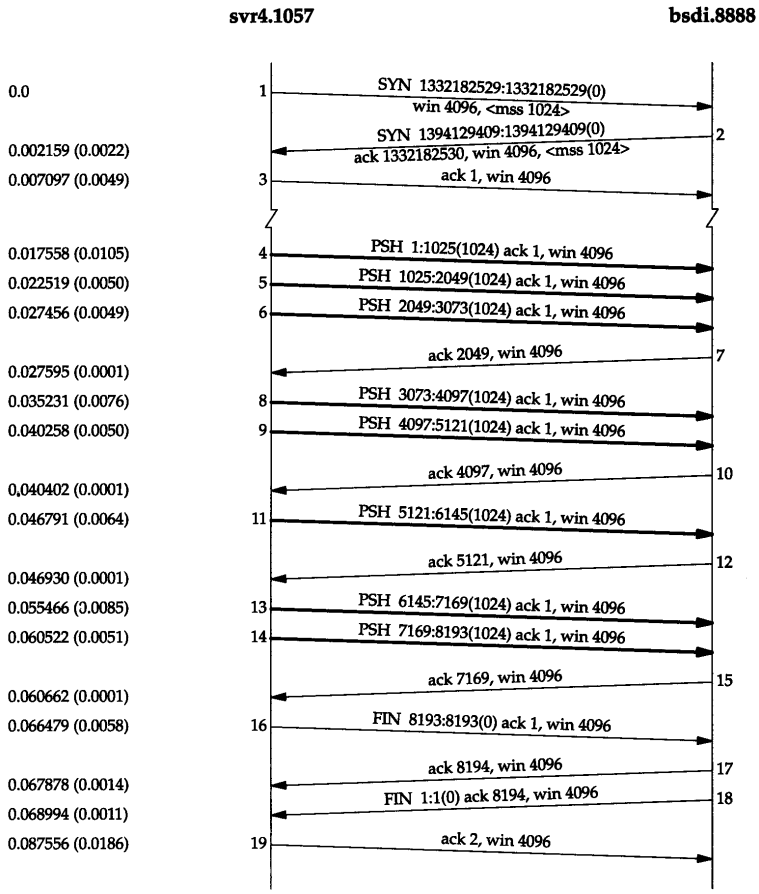


图20-2 从svr4 到bsdi 的另外8192字节数据的传输过程

快的发送方和慢的接收方

图20-3显示了另外一个时间系列。这次是从一个快的发送方（一个 Sparc工作站）到一个慢的接收方（配有慢速以太网卡的80386机器）。它的动态活动情况又有所不同。

发送方发送4个背靠背（back-to-back）的数据报文段去填充接收方的窗口，然后停下来等待一个ACK。接收方发送ACK（报文段8），但通告其窗口大小为0，这说明接收方已收到所有数据，但这些数据都在接收方的 TCP缓冲区，因为应用程序还没有机会读取这些数据。另一个ACK（称为窗口更新）在17.4 ms后发送，表明接收方现在可以接收另外的4096个字节的数据。虽然这看起来像一个 ACK，但由于它并不确认任何新数据，只是用来增加窗口的右边沿，因此被称为窗口更新。

发送方发送最后4个报文段（10~13），再次填充了接收方的窗口。注意到报文段13中包括两个比特标志：PUSH和FIN。随后从接收方传来另外两个ACK，它们确认了最后的4096字节的数据（从4097到8192字节）和FIN（标号为8192）。

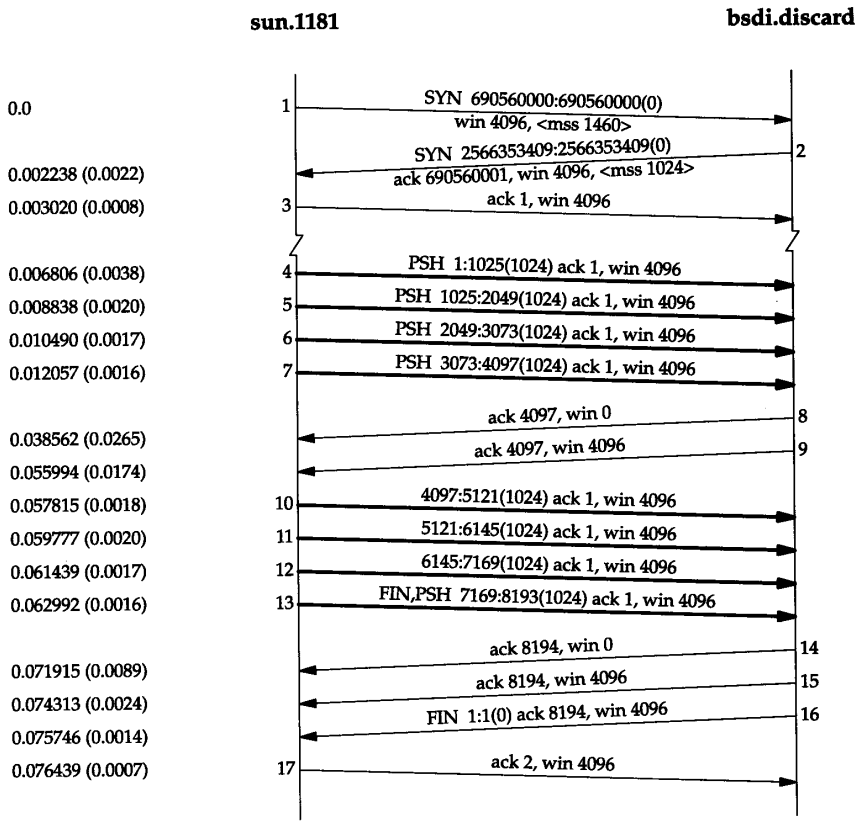


图20-3 从一个快发送方发送8192字节的数据到一个慢接收方

20.3 滑动窗口

图20-4用可视化的方法显示了我们在前一节观察到的滑动窗口协议。

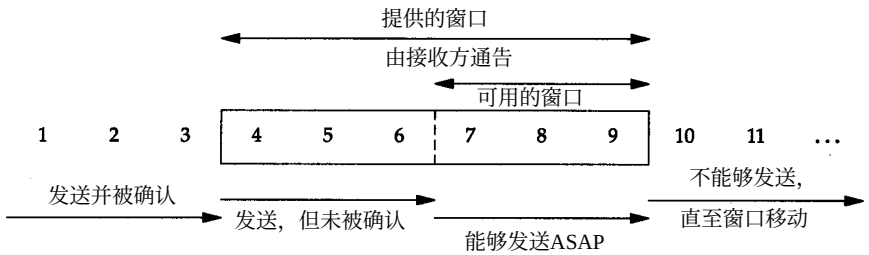


图20-4 TCP滑动窗口的可视化表示

在这个图中，我们将字节从1至11进行标号。接收方通告的窗口称为提出的窗口（offered window），它覆盖了从第4字节到第9字节的区域，表明接收方已经确认了包括第3字节在内的数据，且通告窗口大小为6。回顾第17章，我们知道窗口大小是与确认序号相对应的。发送方计算它的可用窗口，该窗口表明多少数据可以立即被发送。

当接收方确认数据后，这个滑动窗口不时地向右移动。窗口两个边沿的相对运动增加或减少了窗口的大小。我们使用三个术语来描述窗口左右边沿的运动：

1) 称窗口左边沿向右边沿靠近为窗口合拢。这种现象发生在数据被发送和确认时。

2) 当窗口右边沿向右移动时将允许发送更多的数据，我们称之为窗口张开。这种现象发生在另一端的接收进程读取已经确认的数据并释放了TCP的接收缓存时。

3) 当右边沿向左移动时，我们称之为窗口收缩。Host Requirements RFC强烈建议不要使用这种方式。但TCP必须能够在某一端产生这种情况时进行处理。第22.3节给出了这样的一个例子，一端希望向左移动右边沿来收缩窗口，但没能够这样做。

图20-5表示了这三种情况。因为窗口的左边沿受另一端发送的确认序号的控制，因此不可能向左边移动。如果接收到一个指示窗口左边沿向左移动的ACK，则它被认为是一个重复ACK，并被丢弃。

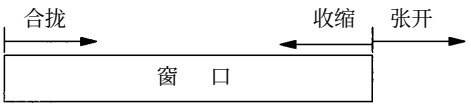


图20-5 窗口边沿的移动

如果左边沿到达右边沿，则称其为一个零窗口，此时发送方不能够发送任何数据。

一个例子

图20-6显示了在图20-1所示的数据传输过程中滑动窗口协议的动态性。

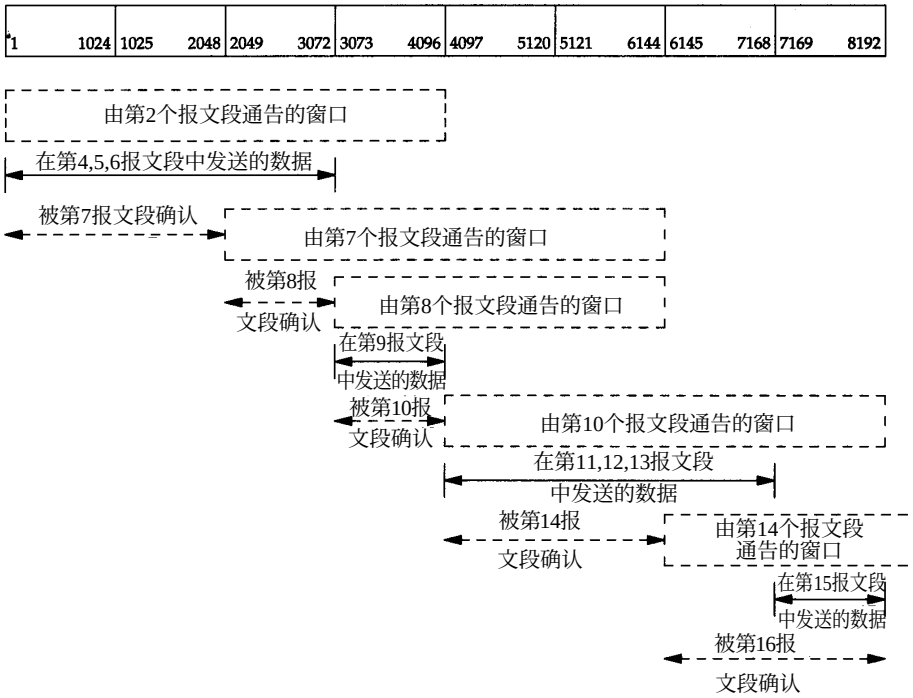


图20-6 图20-1的滑动窗口协议

以该图为例可以总结如下几点：

1) 发送方不必发送一个全窗口大小的数据。

2) 来自接收方的一个报文段确认数据并把窗口向右边滑动。这是因为窗口的大小是相对于确认序号的。

3) 正如从报文段7到报文段8中变化的那样，窗口的大小可以减小，但是窗口的右边沿却不能够向左移动。

4) 接收方在发送一个ACK前不必等待窗口被填满。在前面我们看到许多实现每收到两个报文段就会发送一个ACK。

下面我们可以看到更多的滑动窗口协议动态变化的例子。

20.4 窗口大小

由接收方提供的窗口的大小通常可以由接收进程控制，这将影响TCP的性能。

4.2BSD默认设置发送和接受缓冲区的大小为2048个字节。在4.3BSD中双方被增加为4096个字节。正如我们在本书中迄今为止所看到的例子一样，SunOS 4.1.3、BSD/386和SVR4仍然使用4096字节的默认大小。其他的系统，如Solaris 2.2、4.4BSD和AIX3.2则使用更大的默认缓存大小，如8192或16384等。

插口API允许进程设置发送和接收缓存的大小。接收缓存的大小是该连接上所能够通告的最大窗口大小。有一些应用程序通过修改插口缓存大小来增加性能。

[Mogul 1993]显示了在改变发送和接收缓存大小（在单向数据流的应用中，如文件传输，只需改变发送方的发送缓存和接收方的接收缓存大小）的情况下，位于以太网上的两个工作站之间进行文件传输时的一些结果。它表明对以太网而言，默认的4096字节并不是最理想的大小，将两个缓存增加到16384个字节可以增加约40%左右的吞吐量。在[Papadopoulos和Parulkar 1993]中也有相似的结果。

在20.7节中，我们将看到在给定通信媒体带宽和两端往返时间的情况下，如何计算最小的缓存大小。

一个例子

可以使用sock程序来控制这些缓存的大小。我们以如下方式调用服务器程序：

```
bsdi % sock -i -s -R6144 5555
```

该命令设置接收缓存为6144个字节（-R选项）。接着我们在主机sun上启动客户程序并使之发送8192个字节的数据：

```
sun % sock -i -n1 -w8192 bsdi 5555
```

图20-7显示了结果。

首先注意到的是在报文段2中提供的窗口大小为6144字节。由于这是一个较大的窗口，因此客户立即连续发送了6个报文段（4~9），然后停止。报文段10确认了所有的数据（从第1到6144字节），但提供的窗口大小却为2048，这很可能是接收程序没有机会读取多于2048字节的数据。报文段11和12完成了客户的数据传输，且最后一个报文段带有FIN标志。

报文段13包含与报文段10相同的确认序号，但通告了一个更大的窗口大小。报文段14确认了最后的2048字节的数据和FIN，报文段15和16仅用于通告一个更大的窗口大小。报文段17和18完成通常的关闭过程。

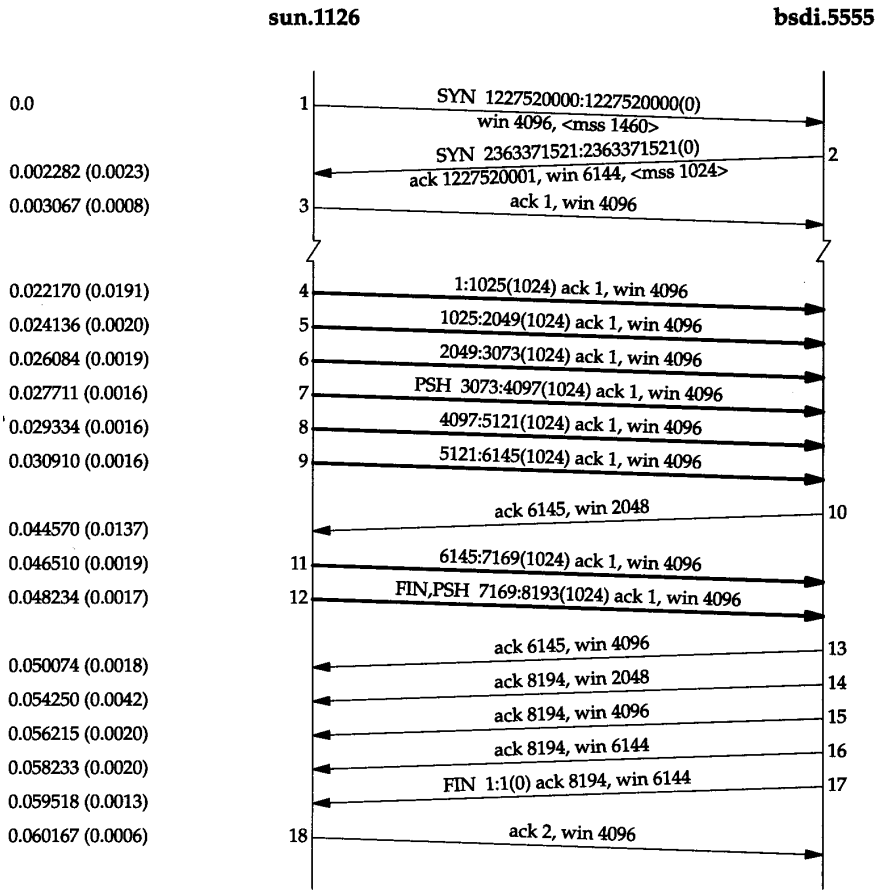


图20-7 接收方提供一个6144字节的接收窗口的情况下的数据传输

20.5 PUSH标志

在每一个TCP例子中，我们都看到了PUSH标志，但一直没有介绍它的用途。发送方使用该标志通知接收方将所收到的数据全部提交给接收进程。这里的数据包括与 PUSH一起传送的数据以及接收方TCP已经为接收进程收到的其他数据。

在最初的TCP规范中，一般假定编程接口允许发送进程告诉它的 TCP何时设置PUSH标志。例如，在一个交互程序中，当客户发送一个命令给服务器时，它设置 PUSH标志并停下来等待服务器的响应（在习题 19.1中我们假定当发送 12字节的请求时客户设置 PUSH标志）。通过允许客户应用程序通知其TCP设置PUSH标志，客户进程通知TCP在向服务器发送一个报文段时不要因等待额外数据而使已提交数据在缓存中滞留。类似地，当服务器的 TCP接收到一个设置了PUSH标志的报文段时，它需要立即将这些数据递交给服务器进程而不能等待判断是否还会有额外的数据到达。

然而，目前大多数的 API没有向应用程序提供通知其 TCP设置PUSH标志的方法。的确，许多实现程序认为PUSH标志已经过时，一个好的TCP实现能够自行决定何时设置这个标志。

如果待发送数据将清空发送缓存，则大多数的源于伯克利的实现能够自动设置 PUSH标志。这意味着我们能够观察到每个应用程序写的数据均被设置了 PUSH标志，因为数据在写的时候

就立即被发送。

代码中的注释表明该算法对那些只有在缓存被填满或收到一个PUSH标志时才向应用程序提交数据的TCP实现有效。

使用插口API通知TCP设置正在接收数据的PUSH标志或得到该数据是否被设置PUSH标志的信息是不可能的。

由于源于伯克利的实现一般从不将接收到的数据推迟交付给应用程序，因此它们忽略所接收的PUSH标志。

举例

在图20-1中我们观察到所有8个数据报文段（4~6、9、11~13和15）的PUSH标志均被置1，这是因为客户进行了8次1024字节数据的写操作，并且每次写操作均清空了发送缓存。

再次观察图20-7，我们预计报文段12中的PUSH标志被置1，因为它是最后一个报文段。为什么发送方知道有更多的数据需要发送还设置报文段 7中的PUSH标志呢？这是因为虽然我们指定写的是8192个字节的数据，但发送方的发送缓存却是4096个字节。

值得注意的另外一点是在图 20-7中的第14、15和16这三个连续的确认报文段。在图 20-3中我们也观察到了两个连续的ACK，但那是因为接收方已经通告其窗口为0（使发送方停止）。当窗口张开时，需要发送另一个窗口非0的ACK来使发送方重新启动。可是，在图20-7中，窗口的大小从来没有达到过0。然而，当窗口大小增加了2048个字节的时候，另一个ACK(报文段15和16)被发送以通知对方窗口被更新（在报文段 15和16中，这两个窗口更新是不需要的，因为已经收到了对方的FIN，表明它不会再发送任何数据）。许多TCP实现在窗口大小增加了两个最大报文段长度（本例中为2048字节，因为MSS为1024字节）或者最大可能窗口的50%（本例中为2048字节，因为最大窗口大小为4096字节）时发送这个窗口更新。在第22.3节详细考察糊涂窗口综合症的时候，我们还会看到这种现象。

作为PUSH标志的另一个例子，再次回到图20-3。我们之所以看到前4个报文段（4~7）的标志被设置，是因为它们每一个均使TCP产生了一个报文段并提交给IP层。但是随后，TCP停下来等待一个确认来移动4096字节的窗口。在此期间，TCP又得到了应用程序的最后4096个字节的数据。当窗口张开时（报文段9），发送方TCP知道它有4个可立即发送的报文段，因此它只设置了最后一个报文段（13）的PUSH标志。

20.6 慢启动

迄今为止，在本章所有的例子中，发送方一开始便向网络发送多个报文段，直至达到接收方通告的窗口大小为止。当发送方和接收方处于同一个局域网时，这种方式是可以的。但是如果在发送方和接收方之间存在多个路由器和速率较慢的链路时，就有可能出现一些问题。一些中间路由器必须缓存分组，并有可能耗尽存储器的空间。 [Jacobson 1988]证明了这种连接方式是如何严重降低了TCP连接的吞吐量的。

现在，TCP需要支持一种被称为“慢启动(slow start)”的算法。该算法通过观察到新分组进入网络的速率应该与另一端返回确认的速率相同而进行工作。

慢启动为发送方的TCP增加了另一个窗口：拥塞窗口(congestion window)，记为cwnd。当

与另一个网络的主机建立TCP连接时，拥塞窗口被初始化为1个报文段（即另一端通告的报文段大小）。每收到一个ACK，拥塞窗口就增加一个报文段（cwnd以字节为单位，但是慢启动以报文段大小为单位进行增加）。发送方取拥塞窗口与通告窗口中的最小值作为发送上限。拥塞窗口是发送方使用的流量控制，而通告窗口则是接收方使用的流量控制。

发送方开始时发送一个报文段，然后等待ACK。当收到该ACK时，拥塞窗口从1增加为2，即可以发送两个报文段。当收到这两个报文段的ACK时，拥塞窗口就增加为4。这是一种指数增加的关系。

在某些点上可能达到了互联网的容量，于是中间路由器开始丢弃分组。这就通知发送方它的拥塞窗口开得过大。当我们在下一章讨论TCP的超时和重传机制时，将会看到它们是怎样对拥塞窗口起作用的。现在，我们来观察一个实际中的慢启动。

一个例子

图20-8表示的是将从主机sun发送到主机vangogh.cs.berkeley.edu的数据。这些数据将通过一个慢的SLIP链路，该链路是TCP连接上的瓶颈（我们已经在时间系列上去掉了连接建立的过程）。

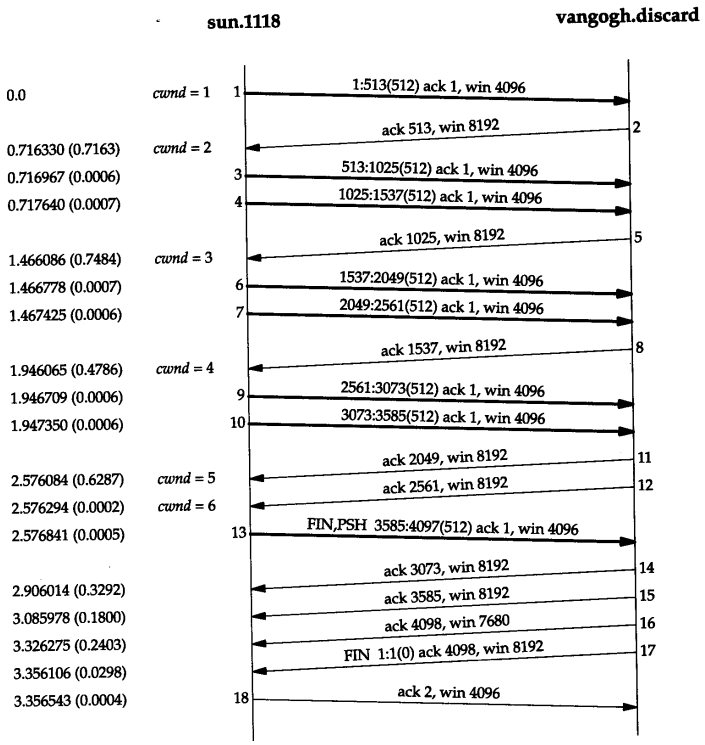


图20-8 慢启动的例子

我们观察到发送方发送一个长度为512字节的报文段，然后等待ACK。该ACK在716 ms后收到。这个时间是一个往返时间的指示。于是拥塞窗口增加了2个报文段，且又发送了两个报文段。当收到报文段5的ACK后，拥塞窗口增加为3。此时尽管可发送多达3个报文段，可是在下一个ACK收到之前，只发送了2个报文段。

在21.6节中我们将再次讨论慢启动，并介绍怎样采用另一种被称为“拥塞避免”的技术来

作为通常的实现。

20.7 成块数据的吞吐量

让我们看一看窗口大小、窗口流量控制以及慢启动对传输成块数据的 TCP连接的吞吐量的相互作用。

图20-9显示了左边的发送方和右边的接收方之间的一个 TCP连接上的时间系列，共显示了16个时间单元。为简单起见，本图只显示离散的时间单元。每个粗箭头线的上半部分显示的是从左到右的携带数据的报文段，标记为 1, 2, 3, 等等。在粗线箭头下面表示的是反向传输的ACK。我们把ACK用细箭头线表示，并标注了被确认的报文段号。

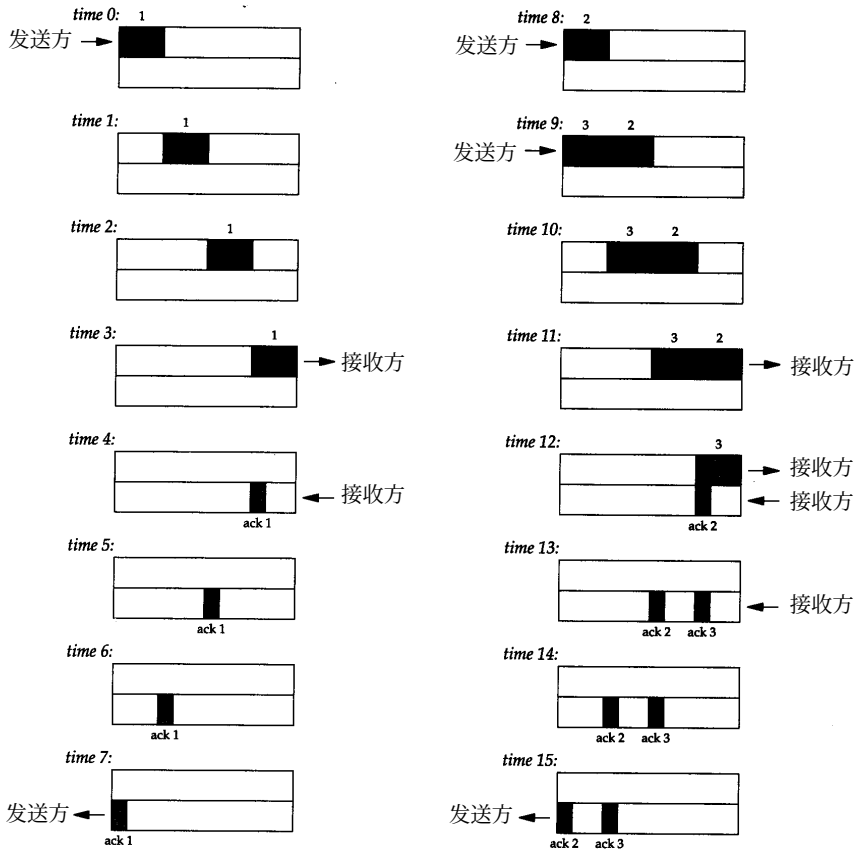


图20-9 时间0~15的成块数据吞吐量举例

在时间0，发送方发送了一个报文段。由于发送方处于慢启动中（其拥塞窗口为 1个报文段），因此在继续发送以前它必须等待该数据段的确认。

在时间1, 2和3，报文段从左向右移动一个时间单元。在时间 4接收方读取这个报文段并产生确认。经过时间 5、6和7，ACK移动到左边的发送方。我们有了一个 8个时间单元的往返时间RTT（Round-Trip Time）。

我们有意把 ACK报文段画得比数据报文段小，这是因为它通常只有一个 IP首部和一个TCP首部。这里显示仅仅是一个单向的数据流动，并且假定 ACK的移动速率与数据报文段的移动速率相等。实际上并不总是这样。

通常发送一个分组的时间取决于两个因素：传播时延（由光的有限速率、传输设备的等待时间等引起）和一个取决于媒体速率（即媒体每秒可传输的比特数）的发送时延。对于一个给定的两个接点之间的通路，传播时延一般是固定的，而发送时延则取决于分组的大小。在速率较慢的情况下发送时延起主要作用（例如，在习题7.2中我们甚至没有考虑传播时延），而在千兆比特速率下传播时延则占主要地位（见图24-6）。

当发送方收到ACK后，在时间8和9发送两个报文段（我们标记为2和3）。此时它的拥塞窗口为2个报文段。这两个报文段向右传送到接收方，在时间12和13接收方产生两个ACK。这两个返回到发送方的ACK之间的间隔与报文段之间的间隔一致，被称为TCP的自计时(self-clocking)行为。由于接收方只有在数据到达时才产生ACK，因此发送方接收到的ACK之间的间隔与数据到达接收方的间隔是一致的（然而在实际中，返回路径上的排队会改变ACK的到达率）。

图20-10表示的是后面16个时间单位。2个ACK的到达使得拥塞窗口从2个报文段增加为4个，而这4个报文段在时间16~19时被发送。第1个ACK在时间23到达。4个ACK的到达使得拥塞窗口从4个报文段增加为8个，并在时间24~31发送8个报文段。

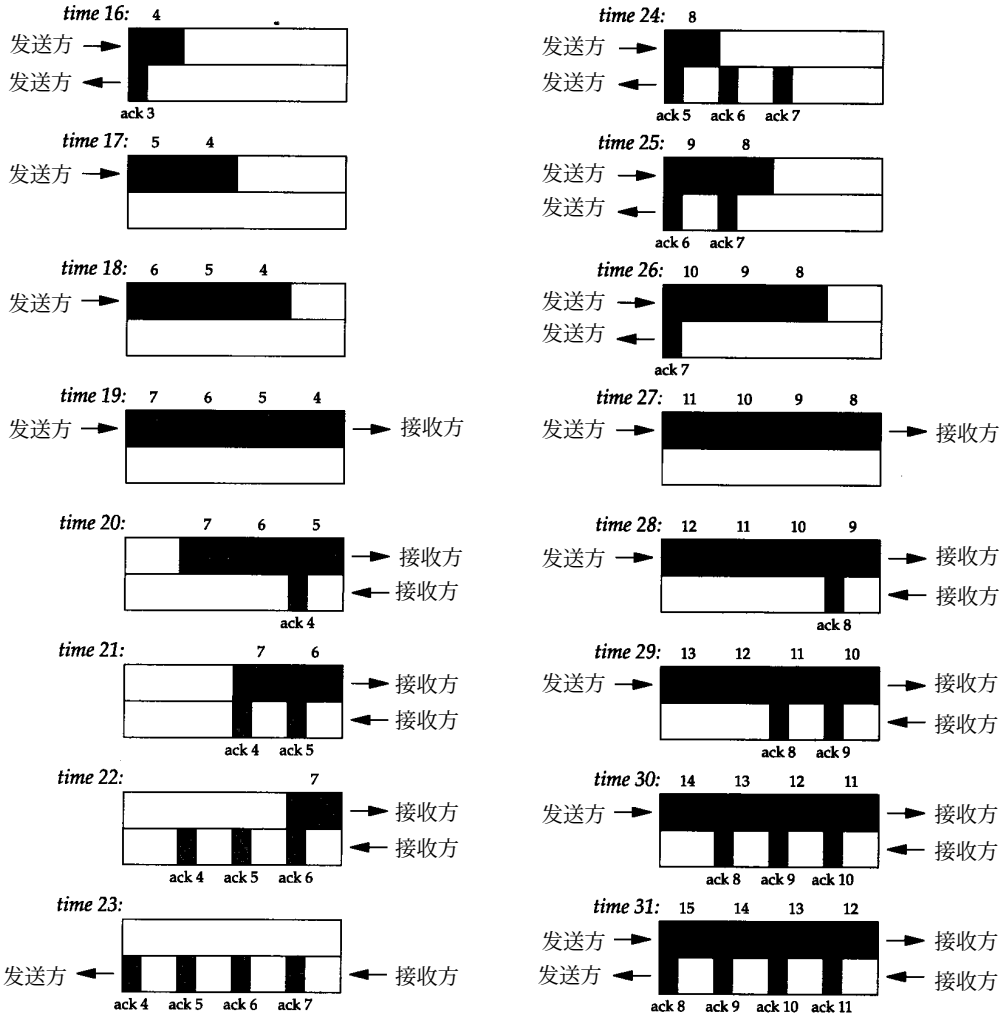


图20-10 时间16~31的成块数据吞吐量举例

在时间31及其后续时间，发送方和接收方之间的管道 (pipe)被填满。此时不论拥塞窗口和通告窗口是多少，它都不能再容纳更多的数据。每当接收方在某一个时间单位从网络上移去一个报文段，发送方就再发送一个报文段到网络上。但是不管有多少报文段填充了这个管道，返回路径上总是具有相同数目的 ACK。这就是连接的理想稳定状态。

20.7.1 带宽时延乘积

现在来回答窗口应该设置为多大的问题。在我们的例子中，作为最大的吞吐量，发送方在任何时候有8个已发送的报文段未被确认。接收方的通告窗口必须不小于这个数目，因为通告窗口限制了发送方能够发送的段的数目。

可以计算通道的容量为：

capacity (bit) = *bandwidth* (b/s) × *round-trip time* (s)

一般称之为带宽时延乘积。这个值依赖于网络速度和两端的 RTT，可以有很大的变动。例如，一条穿越美国（RTT约为60 ms）的T1的电话线路（1 544 000 b/s）的带宽时延乘积为11 580字节。对于20.4节中讨论的缓存大小而言，这个结果是合理的。但是一条穿越美国的 T3电话线路（45 000 000 b/s）的带宽时延乘积则为337 500字节，这个数值超过了最大所允许的TCP通告窗口的大小（65535字节）。在24.4节我们将讨论能够避免当前TCP限制的新的TCP窗口大小选项。

> T1电话线的1 544 000 b/s是原始比特率。由于每193个bit使用1个作为帧同步，因此实际数据率为1 536 000 b/s。一个T3电话线的原始比特率实际上是44 736 000 b/s，其数据率可达到44 210 000 b/s。在讨论中我们使用1.544 Mb/s和45 Mb/s。

不论是带宽还是时延均会影响发送方和接收方之间通路的容量。在图 20-11中我们显示了一个增加了一倍的RTT会使通路容量也增加一倍。

在图20-11底下的说明部分，通过使用一个较长的 RTT，这个管道能够容纳8个报文段而不是4个。

类似地，图20-12表示了增加一倍的带宽也可使该管道的容量增加一倍。

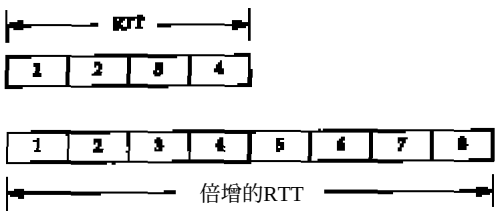


图20-11 RTT加倍可使管道容量增加一倍

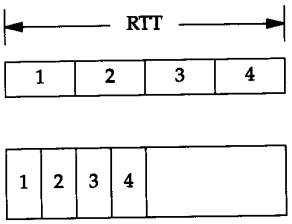


图20-12 带宽加倍可使管道容量增加一倍

在图20-12的下部，假定网络速率已经加倍，使得我们能够只使用上面一半的时间来发送4个报文段。这样，该管道的容量再次加倍（假定该图的上半部分与下半部分中的报文段具有同样大小，即具有相同的比特数）。

20.7.2 拥塞

当数据到达一个大的管道（如一个快速局域网）并向一个较小的管道（如一个较慢的广

域网）发送时便会发生拥塞。当多个输入流到达一个路由器，而路由器的输出流小于这些输入流的总和时也会发生拥塞。

图20-13显示了一个典型的大管道向小管道发送报文的情况。之所以说它典型，是因为大多数的主机都连接在局域网上，并通过一个路由器与速率相对较低的广域网相连（我们再次假定图中上半部分的报文段（9~20）都是相同的，而图中下半部分的ACK也都是相同的）。

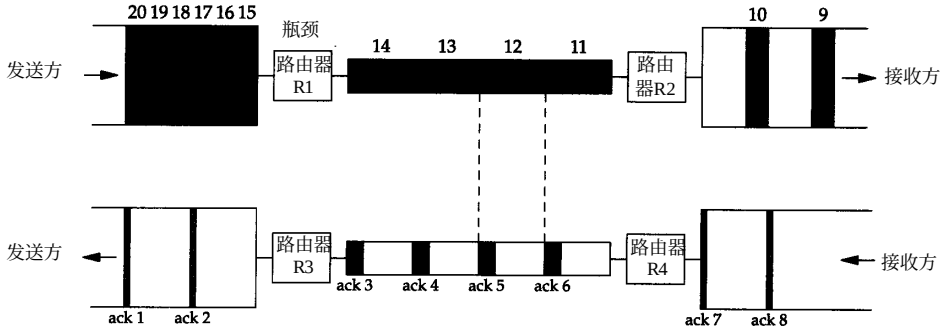


图20-13 从较大管道向较小管道发送分组引起的拥塞

在该图中，我们已经标记路由器 R1为“瓶颈”，因为它是拥塞发生的地方。它从左侧速率较高的局域网接收数据并向右侧速率较低的广域网发送（通常 R1与R3是同样的路由器，如同R2与R4一样。但这并不是必需的，有时也会使用不对称的路径）。当路由器R2将所接收到的分组发送到右侧的局域网时，这些分组之间维持与其左侧广域网上同样的间隔，尽管局域网具有更高的带宽。类似地，返回的确认之间的间隔也与其在路径中最慢的链路上的间隔一致。

在图20-13中已经假定发送方不使用慢启动，它按照局域网的带宽尽可能快地发送编号为1~20的报文段（假定接收方的通告窗口至少为 20个报文段）。正如我们看到的那样， ACK之间的间隔与在最慢链路上的一致。假定瓶颈路由器具有足够的容纳这 20个分组的缓存。如果这个不能保证，就会引起路由器丢弃分组。在 21.6节讨论避免拥塞时会看到怎样避免这种情况。

20.8 紧急方式

TCP提供了“紧急方式 (urgent mode)”，它使一端可以告诉另一端有些具有某种方式的“紧急数据”已经放置在普通的数据流中。另一端被通知这个紧急数据已被放置在普通数据流中，由接收方决定如何处理。

可以通过设置TCP首部（图17-2）中的两个字段来发出这种从一端到另一端的紧急数据已经被放置在数据流中的通知。 URG比特被置1，并且一个16bit的紧急指针被置为一个正的偏移量，该偏移量必须与 TCP首部中的序号字段相加，以便得出紧急数据的最后一个字节的序号。

仍有许多关于紧急指针是指向紧急数据的最后一个字节还是指向紧急数据最后一个字节的下一个字节的争论。最初的TCP规范给出了两种解释，但Host Requirements RFC确定指向最后一个字节是正确的。

然而，问题在于大多数的实现（包括源自伯克利的实现）继续使用错误的解释。

所有符合Host Requirements RFC的实现都是可兼容的，但很有可能无法与其他大多数主机正确通信。

TCP必须通知接收进程，何时已接收到一个紧急数据指针以及何时某个紧急数据指针还不在此连接上，或者紧急指针是否在数据流中向前移动。接着接收进程可以读取数据流，并必须能够被告知何时碰到了紧急数据指针。只要从接收方当前读取位置到紧急数据指针之间有数据存在，就认为应用程序处于“紧急方式”。在紧急指针通过之后，应用程序便转回到正常方式。

TCP本身对紧急数据知之甚少。没有办法指明紧急数据从数据流的何处开始。TCP通过连接传送的唯一信息就是紧急方式已经开始（TCP首部中的URG比特）和指向紧急数据最后一个字节的指针。其他的事情留给应用程序去处理。

不幸的是，许多实现不正确地称TCP的紧急方式为带外数据(out-of-band data)。如果一个应用程序确实需要一个独立的带外信道，第二个TCP连接是达到这个目的的最简单的方法（许多运输层确实提供许多人认为的那种真正的带外数据：使用同一个连接的独立的逻辑数据通道作为正常的数据通道。这是TCP所没有提供的）。

TCP的紧急方式与带外数据之间的混淆，也是因为主要的编程接口（插口API）将TCP的紧急方式映射为称为带外数据的插口。

紧急方式有什么作用呢？两个最常见的例子是Telnet和Rlogin。当交互用户键入中断键时，我们在第26章将看到使用紧急方式来完成这个功能的例子。另一个例子是FTP，当交互用户放弃一个文件的传输时，我们将在第27章看到这样的一个例子。

Telnet和Rlogin从服务器到客户使用紧急方式是因为在这个方向上的数据流很可能要被客户的TCP停止（也即，它通告了一个大小为0的窗口）。但是如果服务器进程进入了紧急方式，尽管它不能够发送任何数据，服务器TCP也会立即发送紧急指针和URG标志。当客户TCP接收到这个通知时就会通知客户进程，于是客户可以从服务器读取其输入、打开窗口并使数据流动。

如果在接收方处理第一个紧急指针之前，发送方多次进入紧急方式会发生什么情况呢？在数据流中的紧急指针会向前移动，而其在接收方的前一个位置将丢失。接收方只有一个紧急指针，每当对方有新的值到达时它将被覆盖。这意味着如果发送方进入紧急方式时所写的内容对接收方非常重要，那么这些字节数据必须被发送方用某种方式特别标记。我们将看到Telnet通过在数据流中加入一个值为255的字节作为前缀来标记它所有的命令。

一个例子

让我们观察一下即使是在接收方窗口关闭的情况下，TCP是如何发送紧急数据的。在主机bsdi上启动sock程序，并使之在连接建立后和从网络读取前暂停10秒种（通过使用-P选项），这将使另一端填满发送窗口：

```
bsdi %sock -i -s -P10 5555
```

接着我们在主机sun上启动客户，使之使用一个8192字节的发送缓存（使用-S选项）并进行6个向网络写1024字节数据的操作（使用-n选项）。还指明-U5选项，告知它向网络写第5个缓存之前要写1个字节的数据，并进入紧急数据方式。我们指明详细标志来观察写的顺序：

```
sun % sock -v -i -n6 -S8192 -U5 bsdi 5555
connected on 140.252.13.33.1305 to 140.252.13.35.5555
SO_SNDBUF = 8192
TCP_MAXSEG = 1024
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1 byte of urgent data
wrote 1024 bytes
wrote 1024 bytes
```

我们设置发送缓存为8192个字节，以便让发送应用程序能够立即写所有的数据。图 20-14显示了tcpdump输出的这个交换过程的结果（删去了连接建立的过程）。第1~5行表示发送方用4个1024字节的报文段去填充接收方的窗口。然后由于接收方的窗口被填满（第 4行的ACK确认了数据，但并没有移动窗口的右边沿），所以发送方停止发送。

在写了第4个正常数据之后，应用进程写了1个字节并进入紧急方式。第6行是该应用进程写的结果，紧急指针被设置为4098。尽管发送方不能发送任何数据，但紧急指针和URG标志一起被发送。

5个这样的ACK在13 ms内被发送（第6~10行）。第1个ACK在应用进程写1个字节并进入紧急方式时被发送，后面两个在应用进程写最后两个 1024字节的数据时被发送（尽管 TCP不能发送这2048个字节的数据，可每次当应用程序执行写操作的时候， TCP的输出功能被调用。当TCP看到正处于紧急方式时，它会发送其他的紧急通知）。第4个ACK在应用进程关闭其TCP连接时被发送（TCP的输出功能再次被调用）。发送应用程序在启动几毫秒后终止——在接收方应用进程已经发出其第一个写操作之前。 TCP将所有的数据进行排队，并在可能时发送出去（这就是为何指明发送缓存为 8192字节的原因，因此只有这样才能够把所有的数据都放置在缓存中）。第5个ACK很可能是在接收第 4行的ACK时产生的。发送 TCP很可能在这个ACK到达前便已将其第 4个报文段放入队列以便输出（第 5行）。另一端接收到这个 ACK也会引起TCP输出例程被调用。

```
 1  0.0                  sun.1305 > bsdi.5555: P 1:1025(1024) ack 1 win 4096
 2  0.073743 (0.0737)    sun.1305 > bsdi.5555: P 1025:2049(1024) ack 1 win 4096
 3  0.096969 (0.0232)    sun.1305 > bsdi.5555: P 2049:3073(1024) ack 1 win 4096
 4  0.157514 (0.0605)    bsdi.5555 > sun.1305: . ack 3073 win 1024
 5  0.164267 (0.0068)    sun.1305 > bsdi.5555: P 3073:4097(1024) ack 1 win 4096

 6  0.167961 (0.0037)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 7  0.171969 (0.0040)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 8  0.176196 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 9  0.180373 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
10  0.180768 (0.0004)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

11  0.367533 (0.1868)    bsdi.5555 > sun.1305: . ack 4097 win 0
12  0.368478 (0.0009)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

13  9.829712 (9.4612)    bsdi.5555 > sun.1305: . ack 4097 win 2048
14  9.831578 (0.0019)    sun.1305 > bsdi.5555: . 4097:5121(1024) ack 1 win 4096
                                                 urg 4098
15  9.833303 (0.0017)    sun.1305 > bsdi.5555: . 5121:6145(1024) ack 1 win 4096

16  9.835089 (0.0018)    bsdi.5555 > sun.1305: . ack 4097 win 4096
17  9.835913 (0.0008)    sun.1305 > bsdi.5555: FP 6145:6146(1) ack 1 win 4096
18  9.840264 (0.0044)    bsdi.5555 > sun.1305: . ack 6147 win 2048
19  9.842386 (0.0021)    bsdi.5555 > sun.1305: . ack 6147 win 4096
20  9.843622 (0.0012)    bsdi.5555 > sun.1305: F 1:1(0) ack 6147 win 4096
21  9.844320 (0.0007)    sun.1305 > bsdi.5555: . ack 2 win 4096
```

图20-14 tcpdump 对TCP紧急方式的输出结果

接着，接收方确认最后的1024字节的数据（第11行），但同时通告窗口为0。发送方用一个包含紧急通知的报文段进行了响应。

在第13行，当应用进程被唤醒、并从接收缓存读取一些数据时，接收方通告窗口为 2048字节。于是后面又发送了两个1024字节的报文段（第14和15行）。其中，由于紧急指针在第1个报文段的范围内，因此这个报文段被设置了紧急通知标志，而第 2个报文段则关闭了该标志。

当接收方再次打开窗口（第16行）时，发送方传输最后的数据（序号为6145）并发起正常的连接关闭。

图20-15显示了发送的6145个字节数据的序号。可以看到当进入紧急方式时所发送的字节的序号是4097，但在图20-14中紧急指针指向4098，这证明了该实现（SunOS 4.1.3）将紧急指针设置为紧急数据最后字节的下一个字节。

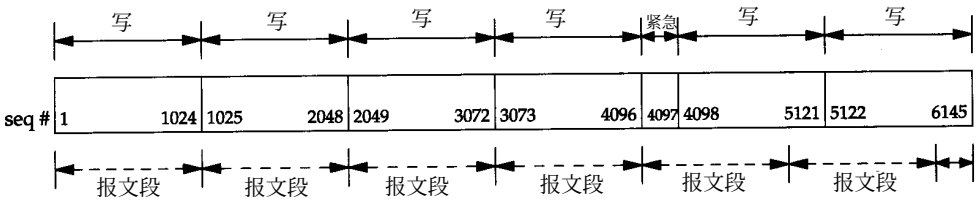


图20-15 紧急方式例子中，应用进程的写操作和TCP的一些报文段

该图还可以让我们观察TCP是如何对应用进程写的数据进行重新分组化的。当进入紧急方式时待输出的1个字节是与在缓存中的后面1023个字节一同发送的。下一个报文段也包含1024字节的数据，而最后一个报文段则只包含一个字节。

20.9 小结

正如我们在本章一开始时讲的那样，没有一种单一的方法可以使用 TCP进行成块数据的交换。这是一个依赖于许多因素的动态处理过程，有些因素我们可以控制（如发送和接收缓存的大小），而另一些我们则没有办法控制（如网络拥塞、与实现有关的特性等）。在本章，我们已经考察了许多TCP的传输过程，介绍了所有我们能够看到的特点和算法。

进行成块数据有效传输的最重要的方法是TCP的滑动窗口协议。我们考察了TCP为使发送方和接收方之间的管道充满来获得最可能快的传输速度而采用的方法。我们用带宽时延乘积衡量管道的容量，并分析了该乘积与窗口大小之间的关系。在24.8节介绍TCP性能的时候将再次涉及这个概念。

我们还介绍了TCP的PUSH标志，因为在跟踪结果中总是观察到它，但我们无法对它的设置与否进行控制。本章最后一个主题是TCP的紧急数据，人们常常错误地称其为“带外数据”。TCP的紧急方式只是一个从发送方到接收方的通知，该通知告诉接收方紧急数据已被发送，并提供该数据最后一个字节的序号。应用程序使用的有关紧急数据部分的编程接口常常都不是最佳的，从而导致更多的混乱。

习题

20.1 在图20-6中，我们可以看到一个序号为0的字节和一个序号为8193的字节，试问这两个

字节的含义是什么？

20.2 提前观察图22-1，并解释主机bsdi设置PUSH标志的含义。

20.3 在一个Usenet记录中，有人抱怨说美国和日本之间的一个128 ms时延、速率为256 000 b/s的链路吞吐量为120 000 b/s（利用率为47%），而当链路通过卫星时其吞吐量则为33 000 b/s（利用率为13%）。试问在这两种情况下窗口大小各为多少（假定卫星链路的时延为500 ms）？卫星链路的窗口大小应该如何调整？

20.4 如果API提供一种方法，使得发送方可以告诉其TCP打开PUSH标志，而接收方可以查询一个接收的报文段是否被设置了PUSH标志，试问该标志能否被用作一个记录标记？

20.5 在图20-3中为什么没有合并报文段15和16？

20.6 在图20-13中，我们假定对应数据报文段之间的间隔，返回的ACK之间的间隔被分隔得很好。如果在链路某处进行缓存并使许多ACK同时到达发送方，试问会发生什么情况？

第21章 TCP的超时与重传

21.1 引言

TCP提供可靠的运输层。它使用的方法之一就是确认从另一端收到的数据。但数据和确认都有可能会丢失。TCP通过在发送时设置一个定时器来解决这种问题。如果当定时器溢出时还没有收到确认，它就重传该数据。对任何实现而言，关键之处就在于超时和重传的策略，即怎样决定超时间隔和如何确定重传的频率。

我们已经看到过两个超时和重传的例子：(1) 在6.5节的ICMP端口不能到达的例子中，看到TFTP客户使用UDP实现了一个简单的超时和重传机制：假定5秒是一个适当的时间间隔，并每隔5秒进行重传；(2) 在向一个不存在的主机发送ARP的例子中（第4.5节），我们看到当TCP试图建立连接的时候，在每个重传之间使用一个较长的时延来重传SYN。

对每个连接，TCP管理4个不同的定时器。

1) 重传定时器使用于当希望收到另一端的确认。在本章我们将详细讨论这个定时器以及一些相关的问题，如拥塞避免。

2) 坚持(persist)定时器使窗口大小信息保持不断流动，即使另一端关闭了其接收窗口。第22章将讨论这个问题。

3) 保活(keepalive)定时器可检测到一个空闲连接的另一端何时崩溃或重启。第23章将描述这个定时器。

4) 2MSL定时器测量一个连接处于TIME_WAIT状态的时间。我们在18.6节对该状态进行了介绍。

本章以一个简单的TCP超时和重传的例子开始，然后转向一个更复杂的例子。该例子可以使我们观察到TCP时钟管理的所有细节。可以看到TCP的典型实现是怎样测量TCP报文段的往返时间以及TCP如何使用这些测量结果来为下一个将要传输的报文段建立重传超时时间。接着我们将研究TCP的拥塞避免——当分组丢失时TCP所采取的动作——并提供一个分组丢失的实际例子，我们还将介绍较新的快速重传和快速恢复算法，并介绍该算法如何使TCP检测分组丢失比等待时钟超时更快。

21.2 超时与重传的简单例子

首先观察TCP所使用的重传机制，我们将建立一个连接，发送一些分组来证明一切正常，然后拔掉电缆，发送更多的数据，再观察TCP的行为。

```
bsdi % telnet svr4 discard
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
hello, world                              正常发送本行
and hi                                    在发送本行前断连
Connection closed by foreign host.        9分钟后TCP放弃时输出
```

图21-1表示的是tcpdump的输出结果（已经去掉了bsdi设置的服务类型信息）。

```
 1    0.0                bsdi.1029 > svr4.discard: S 1747921409:1747921409(0)
                                                     win 4096 <mss 1024>
 2    0.004811 ( 0.0048) svr4.discard > bsdi.1029: S 3416685569:3416685569(0)
                                                     ack 1747921410
                                                     win 4096 <mss 1024>
 3    0.006441 ( 0.0016) bsdi.1029 > svr4.discard: . ack 1 win 4096

 4    6.102290 ( 6.0958) bsdi.1029 > svr4.discard: P 1:15(14) ack 1 win 4096
 5    6.259410 ( 0.1571) svr4.discard > bsdi.1029: . ack 15 win 4096

 6   24.480158 (18.2207) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
 7   25.493733 ( 1.0136) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
 8   28.493795 ( 3.0001) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
 9   34.493971 ( 6.0002) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
10   46.484427 (11.9905) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
11   70.485105 (24.0007) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
12  118.486408 (48.0013) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
13  182.488164 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
14  246.489921 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
15  310.491678 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
16  374.493431 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
17  438.495196 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
18  502.486941 (63.9917) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
19  566.488478 (64.0015) bsdi.1029 > svr4.discard: R 23:23(0) ack 1 win 4096
```

图21-1 TCP超时和重传的简单例子

第1、2和3行表示正常的TCP连接建立的过程，第4行是“ hello, world ”（12个字符加上回车和换行）的传输过程，第 5行是其确认。接着我们从 svr4拔掉了以太网电缆，第 6行表示“ and hi ”将被发送。第7~18行是这个报文段的12次重传过程，而第19行则是发送方的TCP最终放弃并发送一个复位信号的过程。

现在检查连续重传之间不同的时间差，它们取整后分别为 1、3、6、12、24、48和多个64秒。在本章的后面，我们将看到当第一次发送后所设置的超时时间实际上为 1.5秒（它在首次发送后的1.0136秒而不是精确的1.5秒后，发生的原因我们已在图 18-7中进行了解释），此后该时间在每次重传时增加1倍并直至64秒。

这个倍乘关系被称为“ 指数退避 (exponential backoff) ”。可以将该例子与 6.5节中的TFTP例子比较，在那里每次重传总是在前一次的 5秒后发生。

首次分组传输（第6行，24.480秒）与复位信号传输（第 19行，566.488秒）之间的时间差约为9分钟，该时间在目前的TCP实现中是不可变的。

对于大多数实现而言，这个总时间是不可调整的。Solaris 2.2允许管理者改变这个时间（E.4节中的tcp_ip_abort_interval变量），且其默认值为2分钟，而不是最常用的9分钟。

21.3 往返时间测量

TCP超时与重传中最重要的部分就是对一个给定连接的往返时间（ RTT）的测量。由于路由器和网络流量均会变化，因此我们认为这个时间可能经常会发生变化， TCP应该跟踪这些变化并相应地改变其超时时间。

首先TCP必须测量在发送一个带有特别序号的字节和接收到包含该字节的确认之间的RTT。在上一章中，我们曾提到在数据报文段和 ACK之间通常并没有一一对应的关系。在图

20.1中，这意味着发送方可以测量到的一个RTT，是在发送报文段4（第1~1024字节）和接收报文段7（对1~1024字节的ACK）之间的时间，用M表示所测量到的RTT。

最初的TCP规范使TCP使用低通过滤器来更新一个被平滑的RTT估计器（记为*O*）。

$$R \leftarrow \alpha R + (1 - \alpha)M$$

这里的α是一个推荐值为0.9的平滑因子。每次进行新测量的时候，这个被平滑的RTT将得到更新。每个新估计的90%来自前一个估计，而10%则取自新的测量。

该算法在给定这个随RTT的变化而变化的平滑因子的条件下，RFC 793推荐的重传超时时间*RTO*（Retransmission TimeOut）的值应该设置为

$$RTO = R\beta$$

这里的β是一个推荐值为2的时延离散因子。

[Jacobson 1988] 详细分析了在RTT变化范围很大时，使用这个方法无法跟上这种变化，从而引起不必要的重传。正如Jacobson记述的那样，当网络已经处于饱和状态时，不必要的重传会增加网络的负载，对网络而言这就像在火上浇油一样。

除了被平滑的RTT估计器，所需要做的还有跟踪RTT的方差。在往返时间变化起伏很大时，基于均值和方差来计算*RTO*，将比作为均值的常数倍数来计算*RTO*能提供更好的响应。在[Jacobson 1988]中的图5和图6中显示了根据RFC 793计算的某些实际往返时间的*RTO*和下面考虑了往返时间的方差所计算的*RTO*的比较结果。

正如Jacobson所描述的，均值偏差是对标准偏差的一种好的逼近，但却更容易进行计算（计算标准偏差需要一个平方根）。这就引出了下面用于每个RTT测量*M*的公式。

$$Err = M - A$$

$$A \leftarrow A + gErr$$

$$D \leftarrow D + h(|Err| - D)$$

$$RTO = A + 4D$$

这里的*A*是被平滑的RTT（均值的估计器）而*D*则是被平滑的均值偏差。*Err*是刚得到的测量结果与当前的RTT估计器之差。*A*和*D*均被用于计算下一个重传时间（*RTO*）。增量*g*起平均作用，取为1/8（0.125）。偏差的增益是*h*，取值为0.25。当RTT变化时，较大的偏差增益将使*RTO*快速上升。

[Jacobson 1988]指明在计算*RTO*时使用2D，但经过后来更深入的研究，[Jacobson1990c]将该值改为4D，也就是在BSD Net/1的实现中使用的那样。

Jacobson指明了一种使用整数运算来计算这些公式的方法，并被许多实现所采用（这也就是*g*, *h*和倍数4均是2的乘方的一个原因，这样一来计算均可只通过移位操作而不需要乘、除运算来完成）。

将Jacobson与最初的方法比较，我们发现被平滑的均值计算公式是类似的（α是1减去增益*g*），而增益可使用不同的值。而且Jacobson计算*RTO*的公式依赖于被平滑的RTT和被平滑的均值偏差，而最初的方法则使用了被平滑的RTT的一个倍数。

在看完下一节中的例子时，我们将看到这些估计器是如何被初始化的。

Karn算法

在一个分组重传时会产生这样一个问题：假定一个分组被发送。当超时发生时，*RTO*正

如21.2节中显示的那样进行退避，分组以更长的*RTO*进行重传，然后收到一个确认。那么这个ACK是针对第一个分组的还是针对第二个分组呢？这就是所谓的重传多义性问题。

[Karn and Partridge 1987]规定，当一个超时和重传发生时，在重传数据的确认最后到达之前，不能更新RTT估计器，因为我们并不知道ACK对应哪次传输（也许第一次传输被延迟而并没有被丢弃，也有可能第一次传输的ACK被延迟）。

并且，由于数据被重传，*RTO*已经得到了一个指数退避，我们在下一次传输时使用这个退避后的*RTO*。对一个没有被重传的报文段而言，除非收到了一个确认，否则不要计算新的*RTO*。

21.4 往返时间RTT的例子

在本章中，我们将使用以下这些例子来检查TCP的超时和重传、慢启动以及拥塞避免等方方面面的实现细节。

使用sock程序和如下的命令来将32768字节的数据从主机slip发送到主机vangogh.cs.berkeley.edu上的丢弃服务。

```
slip %sock -D -i -n32 vangogh.cs.berkeley.edu discard
```

在扉页前图中，可以看到slip通过两个SLIP链路与140.252.1以太网相连，并从这里通过Internet到达目的地。通过使用两个9600 b/s的SLIP链路，我们期望能够得到一些可测量的时延。

该命令执行32个写1024字节的操作。由于slip和bsdi之间的MTU为296字节，因此这些操作会产生128个报文段，每个报文段包含256字节的用户数据。整个传输过程的时间约为45秒，我们观察到了一个超时和三次重传。

当该传输过程进行时，我们在slip上使用tcpdump来截获所有的发送和接收的报文段，并通过使用-D选项来打开插口排错功能（见A.6节），这样便可以通过运行一个修改后的trpt(8)程序来打印出连接控制块中与RTT、慢启动及拥塞避免等有关的多个变量。

对于给出的跟踪结果，我们不能够完全进行显示，相反，我们将在介绍本章时看到它的各个部分。图21-2显示的是前5秒中的数据和确认的传输过程。与前面tcpdump的输出相比，我们已对其显示稍微进行了修改。虽然我们仅能够在运行tcpdump的主机上测量分组发送和接收的时间，但在本图中我们希望显示出分组正在网络中传输（它们确实存在，因为这个局域网连接与共享式的以太网并不一样）以及接收主机何时可能产生ACK（在本图中去掉了所有的窗口大小通告。主机slip总是通告窗口大小为4096，而vangogh则总是通告窗口大小为8192）。

还需要注意的是在本图中我们已经将报文段按照在主机slip上发送和接收的序号记为1~13和15。这与在这个主机上所收集的tcpdump的输出结果有关。

21.4.1 往返时间RTT的测量

在图21-2左边的时间轴上有三个括号，它们表明为进行RTT计算对哪些报文段进行了计时，并不是所有的报文段都被计时。

大多数源于伯克利的TCP实现在任何时候对每个连接仅测量一次RTT值。在发送一个报文段时，如果给定连接的定时器已经被使用，则该报文段不被计时。

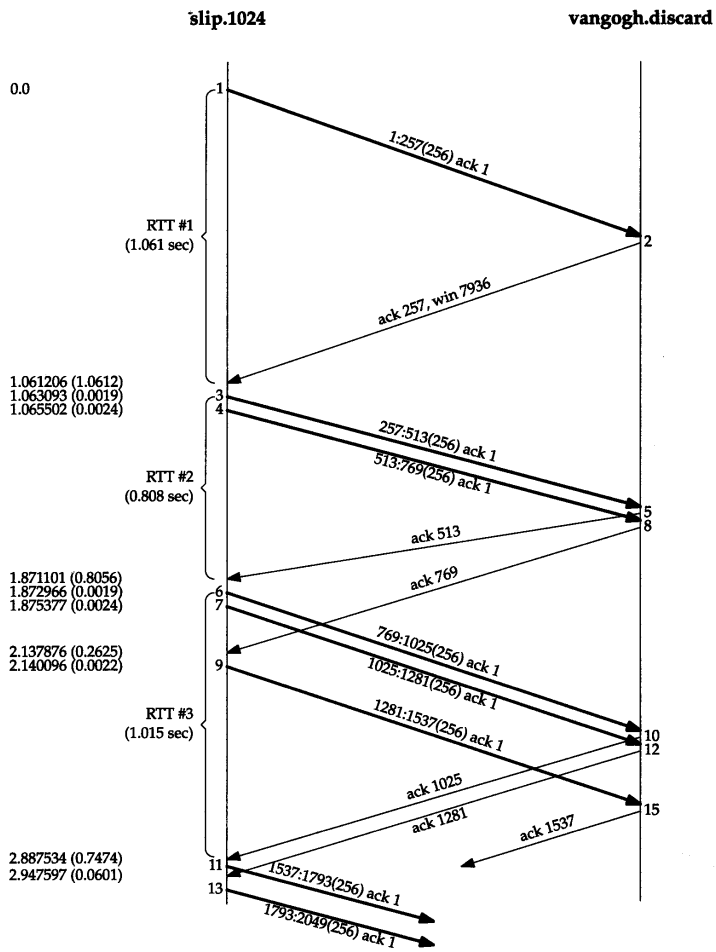


图21-2 分组交换和RTT测量

在每次调用500 ms的TCP的定时器例程时，就增加一个计数器来完成计时。这意味着，如果一个报文段的确认在它发送550 ms后到达，则该报文段的往返时间RTT将是1个滴答（即500 ms）或是2个滴答(即1000 ms)。

对每个连接而言，除了这个滴答计数器，报文段中数据的起始序号也被记录下来。当收到一个包含这个序号的确认后，该定时器就被关闭。如果ACK到达时数据没有被重传，则被平滑的RTT和被平滑的均值偏差将基于这个新测量进行更新。

图21-2中连接上的定时器在发送报文段1时启动，并在确认（报文段2）到达时终止。尽管它的RTT是1.061秒（tcpdump的输出），但插口排错的信息显示该过程经历了3个TCP时钟滴答，即RTT为1500 ms。

下一个被计时的是报文段3。当2.4 ms后传输报文段4时，由于连接的定时器已经被启动，因此该报文段不能被计时。当报文段5到达时，确认了正在被计时的数据。虽然我们从tcpdump的输出结果可以看到其RTT是0.808秒，但它的RTT被计算为1个滴答（500 ms）。

定时器在发送报文段6时再次被启动，并在1.015秒后接收到它的确认（报文段10）时终止。测量到的RTT是2个滴答。报文段7和9不能被计时，因为定时器已经被使用。而且，当收

到报文段8（第769字节的确认）时，由于该报文段不是正在计时的数据的确认，因此什么也没有进行更新。

图21-3显示了本例中通过tcpdump的输出所得到的实际RTT与时钟滴答计数之间的关系。

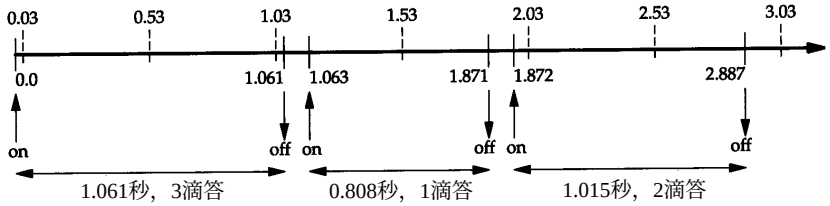


图21-3 RTT测量和时钟滴答

在图的上端表示间隔为500 ms的时钟滴答，图的下端表示tcpdump的输出时间及定时器何时被启动和关闭。在发送报文段 1和接收到报文段2之间经历了3个滴答，时间为1.061秒，因此假定第1个滴答发生在0.03秒处（第1个滴答一定在0~0.061秒之间）。接着该图表示了第2个被测量的RTT为什么被记为1个滴答，而第3个被记为2个滴答。

在这个完整的例子中， 128个报文段被传送，并收集了18个RTT采样。图21-4表示了测量的RTT（取自tcpdump的输出）和TCP为超时所使用的*RTO*（取自插口排错的输出）。在图21-2中，*x*轴从时间0开始，表示的是传输报文段1的时刻，而不是传输第1个SYN的时刻。

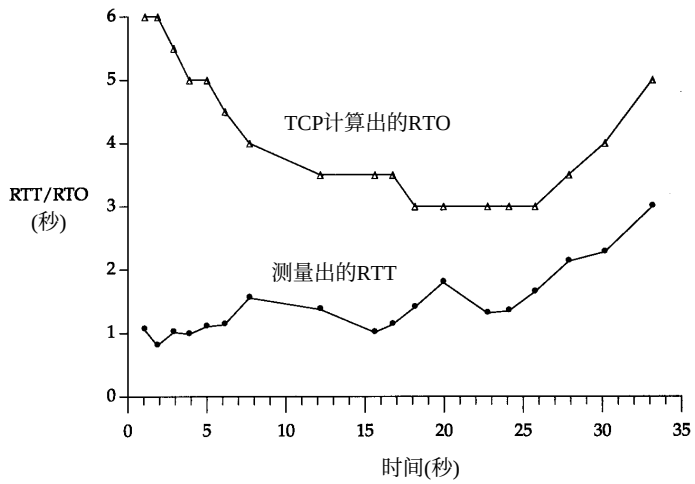


图21-4 测量出的RTT和TCP计算的*RTO*的例子

测量出RTT的前3个数据点对应图21-2所示的3个RTT。在时间10, 14和21处的间隔是由在这些时刻附近发生的重传（将在本章后面给出）引起的。 Karn算法在另一个报文段被发送和确认之前阻止我们更新估计器。同样注意到在这个实现中， TCP计算的*RTO*总是500 ms的倍数。

21.4.2 RTT估计器的计算

让我们来看一下RTT估计器（平滑的RTT和平滑的均值偏差）是如何被初始化和更新，以及每个重传超时是怎样计算的。

变量*A*和*D*分别被初始化为0和3秒。初始的重传超时使用下面的公式进行计算

$$RTO = A + 2D = 0 + 2 \times 3 = 6 \text{ s}$$

（因子2*D*只在这个初始化计算中使用。正如前面提到的，以后使用4*D*和*A*相加来计算*RTO*）。这就是传输初始SYN所使用的*RTO*。

结果是这个初始SYN丢失了，然后超时并引起了重传。图21-5给出了tcpdump输出文件中的前4行。

```
1  0.0                    slip.1024 > vangogh.discard: S 35648001:35648001(0)
                                                        win 4096 <mss 256>

2  5.802377 (5.8024)      slip.1024 > vangogh.discard: S 35648001:35648001(0)
                                                        win 4096 <mss 256>

3  6.269395 (0.4670)      vangogh.discard > slip.1024: S 1365512705:1365512705(0)
                                                        ack 35648002
                                                        win 8192 <mss 512>

4  6.270796 (0.0014)      slip.1024 > vangogh.discard: . ack 1 win 4096
```

图21-5 初始SYN的超时和重传

当超时在5.802秒后发生时，计算当前的*RTO*值为

$$RTO = A + 4D = 0 + 4 \times 3 = 12 \text{ s}$$

因此，应用于*RTO*的指数退避取为12。由于这是第1次超时，我们使用倍数2，因此下一个超时时间取值为24秒。再下一个超时时间的倍数为4，得出值为48秒（这些初始*RTO*，对于一个连接上的最初的SYN，取值为6秒，接下来为24秒，正是我们在图4-5中看到的）。

ACK在重传后467ms到达。*A*和*D*的值没有被更新，这是因为Karn算法对重传的处理比较模糊。下一个发送的报文段是第4行的ACK，但它只是一个ACK，所以没有被计时（只有数据报文段才会被计时）。

当发送第1个数据报文段时（图21-2中的报文段1），*RTO*没有改变，这同样是由于Karn算法。当前的24秒一直被使用，直到进行一个RTT测量。这意味着图21-4中时间0的*RTO*并不真的是24，但我们没有画出那个点。

当第1个数据报文段的ACK（图21-2中的报文段2）到达时，经历了3个时钟滴答，估计器被初始化为

$$A = M + 0.5 = 1.5 + 0.5 = 2$$

$$D = A/2 = 1$$

（因为经历3个时钟滴答，因此，*M*取值为1.5）。在前面，*A*和*D*初始化为0，*RTO*的初始计算值为3。这是使用第1个RTT的测量结果M对估计器进行首次计算的初始值。计算的*RTO*值为

$$RTO = A + 4D = 2 + 4 \times 1 = 6 \text{ s}$$

当第2个数据报文段的ACK（图21-2中的报文段5）到达时，经历了1个时钟滴答（0.5秒），估计器按如下更新：

$$Err = M - A = 0.5 - 2 = -1.5$$

$$A = A + gErr = 2 - 0.125 \times 1.5 = 1.8125$$

$$D = D + h(|Err| - D) = 1 + 0.25 \times (1.5 - 1) = 1.125$$

$$RTO = A + 4D = 1.8125 + 4 \times 1.125 = 6.3125$$

Err、*A*和*D*的定点表示与实际使用的定点计算（在简化浮点计算中表示过）有一些微小的差别。这些不同使*RTO*取值为6秒（而非6.3125秒），正如我们在图21-4中的时间1.871处所画的那样。

21.4.3 慢启动

我们在第20.6节介绍了慢启动算法，在图21-2中可再次看到它的工作过程。

连接上最初只允许传输一个报文段，然后在发送下一个报文段之前必须等待接收它的确认。当报文段2被接收后，就可以再发送两个报文段。

21.5 拥塞举例

现在观察一下数据报文段的传输过程。图 21-6显示了报文段中数据的起始序号与该报文段发送时间的对比图。它提供了一种较好的数据传输的可视化方法。通常代表数据的点将向上和向右移动，这些点的斜率就表示传输速率。当这些点向下和向右移动则表示发生了重传。

在21.4节开始时，我们曾提到整个传输的时间约为 45秒，但在本图中只显示了 35秒钟。这35秒只是数据报文段发送的时间。因为第 1个SYN看来是丢失了并被重传（见图 21-5），因此第1个数据报文段是在第1个SYN发送6.3秒后才发送的。而且，在发送最后一个数据报文段和FIN（图21-6中的34.1秒）之后，在接收方的FIN到达之前，又花费了另外的4.0秒接收来自接收方的最后14个ACK。

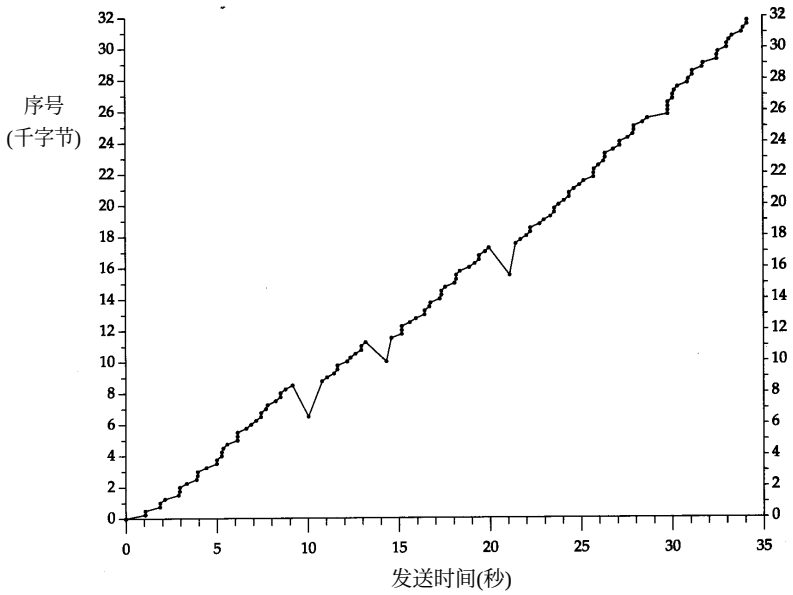


图21-6 从slip 发送32768个字节的数据到vangogh

可以立即看到图21-6中发生在时刻10，14和21附近的3个重传。我们还可以看到在这 3个点中只进行了一次报文段的重传，因为只有一个点下垂低于向上的斜率。

仔细检查一下这几个下垂点中的第 1个点（在10秒标记处的附近）。整理tcpdump的输出结果可以得到图21-7。

在这个图中，除了下面将要讨论的报文段 72，已经去掉了其他所有的窗口通告。主机slip总是通告窗口大小为4096，而主机vangogh则通告窗口为8192。该图中报文段的编号可以看作是图21-2的延续，在那里报文段的编号从 1开始。与图21-2一样，报文段根据在 slip上发送和接收的顺序进行编号， tcpdump在主机slip上运行。我们还去掉了一些与讨论无

关的段（第44, 47和49以及所有来自vangogh的ACK）。

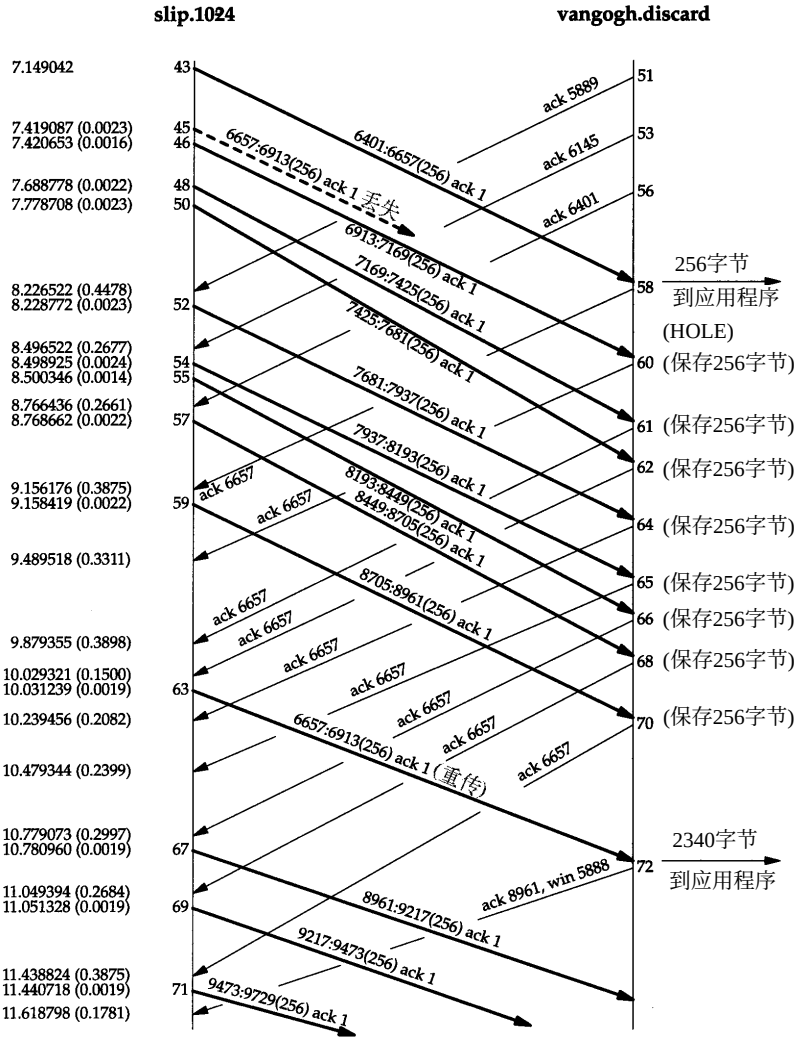


图21-7　10秒标记处附近重传的分组交换

看来报文段45丢失或损坏了，这一点无法从该输出上进行辨认。能够在主机 slip上看到的是对第6657字节（报文段58）以前数据的确认（不包括字节 6657在内）。紧接着的是带有相同序号的8个ACK。正是接收到报文段 62，也就是第3个重复 ACK，才引起自序号6657开始的数据报文段（报文段 63）进行重传。的确，源于伯克利的 TCP实现对收到的重复 ACK进行计数，当收到第3个时，就假定一个报文段已经丢失并重传自那个序号起的一个报文段。这就是Jacobson的快速重传算法，该算法通常与他的快速恢复算法一起配合使用。我们在第 21.7节中介绍这两个算法。

注意到在重传后（报文段 63），发送方继续正常的数据传输（报文段 67、69和71）。TCP不需要等待对方确认重传。

现在检查一下在接收端发生了什么。当按序收到正常数据（报文段 43）后，接收TCP将255个字节的数据交给用户进程。但下一个收到的报文段（报文段 46）是失序的：数据的开始

序号（6913）并不是下一个期望的序号（6657）。TCP保存256字节的数据，并返回一个已成功接收数据的最大序号加1（6657）的ACK。被vangogh接收到的后面7个报文段（48, 50, 52, 54, 55, 57和59）也是失序的，接收方TCP保存这些数据并产生重复ACK。

目前TCP尚无办法告诉对方缺少一个报文段，也无法确认失序数据。此时主机vangogh所能够做的就是继续发送确认序号为6657的ACK。

当缺少的报文段（报文段63）到达时，接收方TCP在其缓存中保存第6657~8960字节的数据，并将这2304字节的数据交给用户进程。所有这些数据在报文段72中进行确认。请注意此时该ACK通告窗口大小为5888（8192 - 2304），这是因为用户进程没有机会读取这些已准备好的2304字节的数据。

如果仔细检查图21-6中tcpdump的输出中第14和21秒附近的下垂点，我们会看到它们也是由于收到了3个重复ACK引起的，这表明一个分组已经丢失。在这些例子中只有一个分组被重传。

在介绍完拥塞避免算法后，将在第21.8节中继续讨论这个例子。

21.6 拥塞避免算法

在第20.6节介绍的慢启动算法是在一个连接上发起数据流的方法，但有时我们会达到中间路由器的极限，此时分组将被丢弃。拥塞避免算法是一种处理丢失分组的方法。该方法的具体描述见[Jacobson 1988]。

该算法假定由于分组受到损坏引起的丢失是非常少的（远小于1%），因此分组丢失就意味着在源主机和目的主机之间的某处网络上发生了拥塞。有两种分组丢失的指示：发生超时和接收到重复的确认（我们在21.5节看到这种现象。如果使用超时作为拥塞指示，则需要使用一个好的RTT算法，正如在21.3节中描述的那样）。

拥塞避免算法和慢启动算法是两个目的不同、独立的算法。但是当拥塞发生时，我们希望降低分组进入网络的传输速率，于是可以调用慢启动来作到这一点。在实际中这两个算法通常在一起实现。

拥塞避免算法和慢启动算法需要对每个连接维持两个变量：一个拥塞窗口*cwnd*和一个慢启动门限*ssthresh*。这样得到的算法的工作过程如下：

1) 对一个给定的连接，初始化*cwnd*为1个报文段，*ssthresh*为65535个字节。

2) TCP输出例程的输出不能超过*cwnd*和接收方通告窗口的大小。拥塞避免是发送方使用的流量控制，而通告窗口则是接收方进行的流量控制。前者是发送方感受到的网络拥塞的估计，而后者则与接收方在该连接上的可用缓存大小有关。

3) 当拥塞发生时（超时或收到重复确认），*ssthresh*被设置为当前窗口大小的一半（*cwnd*和接收方通告窗口大小的最小值，但最少为2个报文段）。此外，如果是超时引起了拥塞，则*cwnd*被设置为1个报文段（这就是慢启动）。

4) 当新的数据被对方确认时，就增加*cwnd*，但增加的方法依赖于我们是否正在进行慢启动或拥塞避免。如果*cwnd*小于或等于*ssthresh*，则正在进行慢启动，否则正在进行拥塞避免。慢启动一直持续到我们回到当拥塞发生时所处位置的半时候才停止（因为我们记录了在步骤2中给我们制造麻烦的窗口大小的一半），然后转为执行拥塞避免。

慢启动算法初始设置*cwnd*为1个报文段，此后每收到一个确认就加1。正如20.6节描述的

那样，这会使窗口按指数方式增长：发送1个报文段，然后是2个，接着是4个……。

拥塞避免算法要求每次收到一个确认时将*cwnd*增加1/*cwnd*。与慢启动的指数增加比起来，这是一种加性增长(additive increase)。我们希望在一个往返时间内最多为*cwnd*增加1个报文段（不管在这个RTT中收到了多少个ACK），然而慢启动将根据这个往返时间中所收到的确认的个数增加*cwnd*。

所有的4.3BSD版本和4.4BSD都在拥塞避免中将增加值不正确地设置为1个报文段的一小部分（即一个报文段的大小除以8），这是错误的，并在以后的版本中不再使用[Floyd 1994]。但是，为了和（不正确的）实现的结果对应，我们在将来的计算中给出了这个细节。

在[Leffler et al. 1989]中介绍的4.3BSD Tahoe版本仅在对方处于一个不同的网络上时才进行慢启动。而4.3BSD Reno版本改变了这种做法，因此，慢启动总是被执行。

图21-8是慢启动和拥塞避免的一个可视化描述。我们以段为单位来显示*cwnd*和*ssthresh*，但它们实际上都是以字节为单位进行维护的。

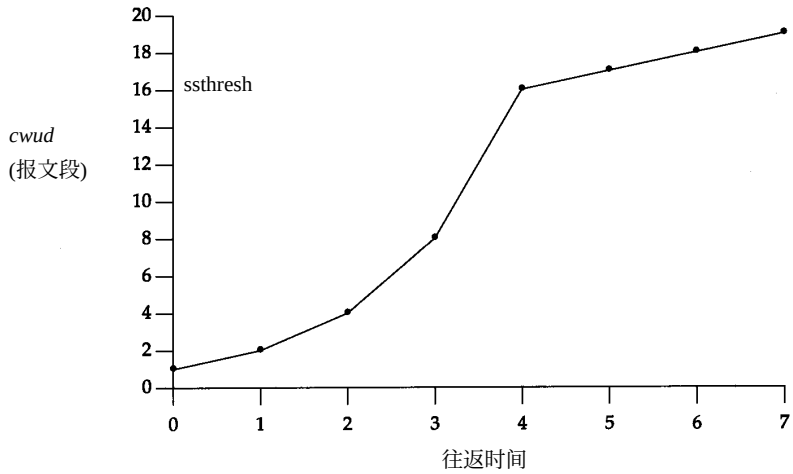


图21-8 慢启动和拥塞避免的可视化描述

在该图中，假定当*cwnd*为32个报文段时就会发生拥塞。于是设置*ssthresh*为16个报文段，而*cwnd*为1个报文段。在时刻0发送了一个报文段，并假定在时刻1接收到它的ACK，此时*cwnd*增加为2。接着发送了2个报文段，并假定在时刻2接收到它们的ACK，于是*cwnd*增加为4（对每个ACK增加1次）。这种指数增加算法一直进行到在时刻3和4之间收到8个ACK后*cwnd*等于*ssthresh*时才停止，从该时刻起，*cwnd*以线性方式增加，在每个往返时间内最多增加1个报文段。

正如我们在这个图中看到的那样，术语“慢启动”并不完全正确。它只是采用了比引起拥塞更慢些的分组传输速率，但在慢启动期间进入网络的分组数增加的速率仍然是在增加的。只有在达到*ssthresh*拥塞避免算法起作用时，这种增加的速率才会慢下来。

21.7 快速重传与快速恢复算法

拥塞避免算法的修改建议1990年提出[Jacobson 1990b]。在我们的例子（见21.5节）中已

经可以看到这些实施中的修改。

在介绍修改之前，我们认识到在收到一个失序的报文段时， TCP立即需要产生一个ACK（一个重复的ACK）。这个重复的ACK不应该被迟延。该重复的ACK的目的在于让对方知道收到一个失序的报文段，并告诉对方自己希望收到的序号。

由于我们不知道一个重复的ACK是由一个丢失的报文段引起的，还是由于仅仅出现了几个报文段的重新排序，因此我们等待少量重复的ACK到来。假如这只是一些报文段的重新排序，则在重新排序的报文段被处理并产生一个新的ACK之前，只可能产生1~2个重复的ACK。如果一连串收到3个或3个以上的重复ACK，就非常可能是一个报文段丢失了（我们在21.5节中见到过这种现象）。于是我们就重传丢失的数据报文段，而无需等待超时定时器溢出。这就是快速重传算法。接下来执行的不是慢启动算法而是拥塞避免算法。这就是快速恢复算法。

在图21-7中可以看到在收到3个重复的ACK之后没有执行慢启动。相反，发送方进行重传，接着在收到重传的ACK以前，发送了3个新的数据的报文段（报文段67, 69和71）。

在这种情况下没有执行慢启动的原因是由于收到重复的ACK不仅仅告诉我们一个分组丢失了。由于接收方只有在收到另一个报文段时才会产生重复的ACK，而该报文段已经离开了网络并进入了接收方的缓存。也就是说，在收发两端之间仍然有流动的数据，而我们不想执行慢启动来突然减少数据流。

这个算法通常按如下过程进行实现：

1) 当收到第3个重复的ACK时，将*ssthresh*设置为当前拥塞窗口*cwnd*的一半。重传丢失的报文段。设置*cwnd*为*ssthresh*加上3倍的报文段大小。

2) 每次收到另一个重复的ACK时，*cwnd*增加1个报文段大小并发送1个分组（如果新的*cwnd*允许发送）。

3) 当下一个确认新数据的ACK到达时，设置*cwnd*为*ssthresh*（在第1步中设置的值）。这个ACK应该是在进行重传后的一个往返时间内对步骤1中重传的确认。另外，这个ACK也应该是对丢失的分组和收到的第1个重复的ACK之间的所有中间报文段的确认。这一步采用的是拥塞避免，因为当分组丢失时我们将当前的速率减半。

在下一节中我们将看到变量*cwnd*和*ssthresh*的计算过程。

快速重传算法最早出现在4.3BSD Tahoe版本中，但它随后错误地使用了慢启动。快速恢复算法出现在4.3BSD Reno版本中。

21.8 拥塞举例(续)

通过使用`tcmdump`和插口排错选项（在第21.4节进行了介绍）来观察一个连接，就会在发送每一个报文段时看到*cwnd*和*ssthresh*的值。如果MSS为256字节，则*cwnd*和*ssthresh*的初始值分别为256和65535字节。每当收到一个ACK时，我们可以看到*cwnd*增加了一个MSS，取值分别为512, 768, 1024, 1280等。假定不会发生拥塞，则最终拥塞窗口将超过接收方的通告窗口，意味着通告窗口将对数据流进行限制。

一个更有趣的例子是观察在拥塞发生时的情况。使用与21.4节同样的例子。当这个例子运行时发生了4次拥塞。为建立连接而发送的初始SYN有一个因超时而引起的重传（见图21-5），接着在数据传输过程中有3个分组丢失（见图21-6）。

图21-9显示了当初始SYN重传并接着发送了前7个数据报文段时变量*cwnd*和*ssthresh*的值（在图21-2中显示了最初的数据报文段及其ACK之间的交换过程）。使用`tcpdump`的记号来表示数据字节：1:257(256)表示第1~256字节。

当SYN的超时发生时，*ssthresh*被置为其最小取值（512字节，在本例中表示2个报文段）。为进入慢启动阶段，*cwnd*被置为1个报文段（256字节，与当前值一致）。

当收到SYN和ACK时，没有对这两个变量做任何修改，因为新的数据还没有被确认。

当ACK 257到达时，因为*cwnd*小于等于*ssthresh*，因此仍然处于慢启动阶段，于是将*cwnd*增加256字节。当收到ACK 513时，进行同样的处理。

当ACK 769到达时，我们不再处于慢启动状态，而是进入了拥塞避免状态。新的*cwnd*值按以下方法计算：

$$cwnd \leftarrow cwnd + \frac{segsize \times segsize}{cwnd} + \frac{segsize}{8}$$

考虑到*cwnd*实际上以字节而非以报文段来维护，因此这就是我们前面提到的增加1/*cwnd*。在这个例子中我们计算

$$cwnd \leftarrow 768 + \frac{256 \times 256}{768} + \frac{256}{8}$$

为885字节（使用整数算法）。当下一个ACK 1025到达时，我们计算

$$cwnd \leftarrow 885 + \frac{256 \times 256}{885} + \frac{256}{8}$$

为991字节（在这些表达式中包括了不正确的256/8项来匹配实现计算的数值，正如我们在前面标注的那样）。

报文段号 (图21-2)	行为			变量	
	发送	接收	注释	*cwnd*	*ssthresh*
	SYN		初始化	256	65535
	SYN		超时重传	256	512
	ACK	SYN, ACK			
1	1:257(256)				
2		ACK 257	慢启动	512	512
3	257:513(256)				
4	513:769(256)				
5		ACK 513	慢启动	768	512
6	769:1025(256)				
7	1025:1281(256)				
8		ACK 769	cong. avoid	885	512
9	1281:1537(256)				
10		ACK 1025	cong. avoid	991	512
11	1537:1793(256)				
12		ACK 1281	cong. avoid	1089	512

图21-9 拥塞避免的例子

这个*cwnd*持续增加一直到在图21-6所示的发生在10秒左右的第1次重传。图21-10是使用与图21-6相同数据得到的图表，并给出了*cwnd*增加的数值。

本图中*cwnd*的前6个值就是我们为图21-9所计算的数值。在这个图中，要想直观分辨出在慢启动过程中的指数增加和在拥塞避免过程中的线性增加之间的区别是不可能的，因为慢启动的过程太快。

我们需要解释在重传的3个点上所发生的情况。回想起每个重传都是因为收到3个重复的ACK，表明1个分组丢失了。这就是21.7节的快速重传算法。*ssthresh*立即设置为当重传发生时正在起作用的窗口大小的一半，但是在接收到重复ACK的过程中*cwnd*允许保持增加，这是因为每个重复的ACK表示1个报文段已离开了网络（接收TCP已缓存了这个报文段，等待所缺数据的到达）。这就是快速恢复算法。

与图20-9类似，图21-10表示了*cwnd*和*ssthresh*的数值。第一列上的报文段编号与图21-7对应。

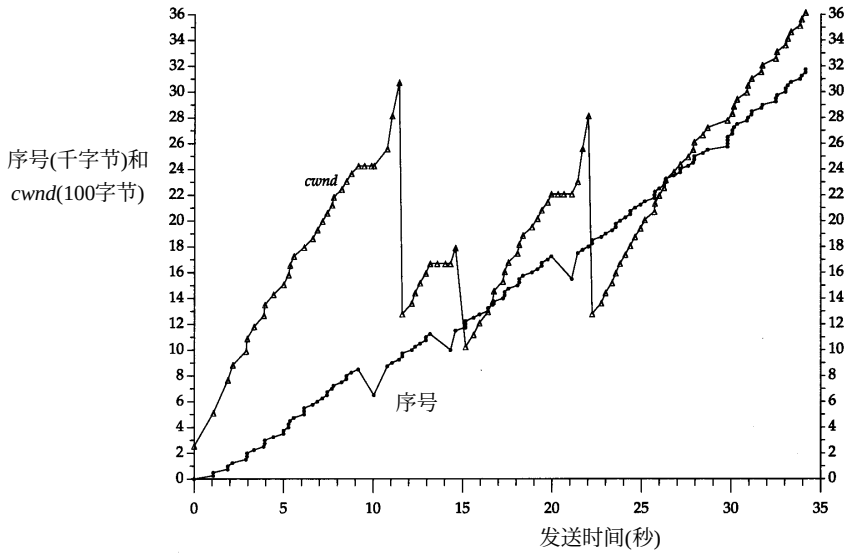


图21-10 当数据被发送时的发送序号和*cwnd*的取值

报文段号 (图21-7)	行为			变量	
	发送	接收	注释	*cwnd*	*ssthresh*
58		ACK 6657	新数据的确认	2426	512
59	8705:8961(256)				
60		ACK 6657	重复 ACK #1	2426	512
61		ACK 6657	重复 ACK #2	2426	512
62		ACK 6657	重复 ACK #3	1792	1024
63	6657:6913(256)		重传		
64		ACK 6657	重复 ACK #4	2048	1024
65		ACK 6657	重复 ACK #5	2304	1024
66		ACK 6657	重复 ACK #6	2560	1024
67	8961:9217(256)				
68		ACK 6657	重复 ACK #7	2816	1024
69	9217:9473(256)				
70		ACK 6657	重复 ACK #8	3072	1024
71	9473:9729(256)				
72		ACK 8961	新数据的确认	1280	1024

图21-11 拥塞避免的例子

*cwnd*的值一直持续增加，从图21-9中对应于报文段12的最终取值（1089）到图21-11中对应于报文段58的第一个取值（2426），而*ssthresh*的值则保持不变（512），这是因为在此过程中没有出现过重传。

当最初的2个重复的ACK（报文段60和61）到达时它们被计数，而*cwnd*保持不变（也就是图21-10中处理重传之前的平坦的一段）。然而，当第3个重复的ACK到达时，*ssthresh*被置

为*cwnd*的一半（四舍五入到报文段大小的下一个倍数），而*cwnd*被置为*ssthresh*加上所收到的重复的ACK数乘以报文段大小（也即1024加上3倍的256），然后发送重传数据。

又有5个重复的ACK到达（报文段64~66, 68和70），每次*cwnd*增加1个报文段长度。最后一个新的ACK（报文段72段）到达时，*cwnd*被置为*ssthresh*（1024）并进入正常的拥塞避免过程。由于*cwnd*小于等于*ssthresh*（现在相等），因此报文段的大小增加到 *cwnd*，取值为1280。当下一个新的ACK到达（没有在图21-11中表示出来）时，*cwnd*大于*ssthresh*，取值为1363。

在快速重传和快速恢复阶段，我们收到报文段 66、68和70中的重复的ACK后才发送新的数据，而不是在接收到报文段 64和65中重复的ACK之后就发送。这是*cwnd*的取值与未被确认的数据大小比较的结果。当报文段 65到达时，*cwnd*为2048，但未被确认的数据有 2304字节（9个报文段：46, 48, 50, 52, 54, 55, 57, 59和63），因此不能发送任何数据。当报文段65到达后，*cwnd*被置为2304，此时我们仍不能进行发送。但是当报文段 66到达时，*cwnd*为2560，所以我们可以发送1个新的数据报文段。类似地，当报文段 68到达时，*cwnd*等于2816，该数值大于未被确认的2560字节的数据大小，因此我们可以发送另 1个新的数据报文段。报文段 70到达时也进行了类似的处理。

在图21-10中的时刻14.3发生下一个重传，也是因为收到了 3个重复的ACK。因此当另一个ACK到达时，可以看到*cwnd*以同样的方式增长，之后降低到1024。

图21-10中的时刻21.1也是因为收到了重复的 ACK而引起了重传。在重传后收到了 3个重复的ACK，因此观察到*cwnd*增加3个，之后降低到1280。在传输的后面部分，*cwnd*以线性方式增加到最终值3615。

21.9 按每条路由进行度量

较新的TCP实现在路由表项中维持许多我们在本章已经介绍过的指标。当一个 TCP连接关闭时，如果已经发送了足够多的数据来获得有意义统计资料，且目的结点的路由表项不是一个默认的表项，那么下列信息就保存在路由表项中以备下次使用：被平滑的 RTT、被平滑的均值偏差以及慢启动门限。所谓“ 足够多的数据 ”是指 16个窗口的数据，这样就可得到 16个RTT采样，从而使被平滑的RTT过滤器能够集中在正确结果的5%以内。

而且，管理员可以使用 route(8)命令来设置给定路由的度量：前一段中给出的三个指标以及MT、输出的带宽时延乘积（见第 20.7节）和输入的带宽时延乘积。

当建立一个新的连接时，不论是主动还是被动，如果该连接将要使用的路由表项已经有这些度量的值，则用这些度量来对相应的变量进行初始化。

21.10 ICMP的差错

让我们来看一下TCP是怎样处理一个给定的连接返回的 ICMP的差错。TCP能够遇到的最常见的ICMP差错就是源站抑制、主机不可达和网络不可达。

当前基于伯克利的实现对这些错误的处理是：

- 一个接收到的源站抑制引起拥塞窗口 *cwnd*被置为1个报文段大小来发起慢启动，但是慢启动门限*ssthresh*没有变化，所以窗口将打开直至它或者开放了所有的通路（受窗口大小和往返时间的限制）或者发生了拥塞。
- 一个接收到的主机不可达或网络不可达实际上都被忽略，因为这两个差错都被认为是

短暂现象。这有可能是由于中间路由器被关闭而导致选路协议要花费数分钟才能稳定到另一个替换路由。在这个过程中就可能发生这两个 ICMP差错中的一个，但是连接并不必被关闭。相反， TCP试图发送引起该差错的数据，尽管最终有可能会超时（回想图21-1中TCP在9分钟内没有放弃的情况）。当前基于伯克利的实现记录发生的ICMP差错，如果连接超时， ICMP差错被转换为一个更合适的的差错码而不是“连接超时”。

早期的BSD实现在任何时候收到一个主机不可达或网络不可达的ICMP差错时会不正确的放弃连接。

一个例子

可以通过在连接中拨号SLIP链路的断开来观察一个ICMP主机不可达的差错是如何被处理的。建立一个从主机slip到主机aix的连接（从扉页前的图中可以看到这个连接经过了我们的拨号SLIP链路）。在建立连接并发送一些数据之后，在路由器 sun和netb之间的SLIP链路被断开，这引起sun上的默认路由表项（见9.2节）被移去。我们希望sun对目的为140.252.1以太网的IP数据报响应ICMP主机不可达。希望观察TCP如何处理这些ICMP差错。

下面是主机slip的交互会话：

```
slip % sock aix echo                    运行sock程序
test line                               键入本行
test line                               和它的回显
                                        此时挂断SLIP链路
another line                            然后键入本行并观察其行为
                                        SLIP链路此时重新建立，
another line                            该行及其回显被交换
line number 3
line number 3
the last line
                                        此时挂断SLIP链路，且没有重新建立
read error: No route to host            TCP最终放弃
```

图21-12显示了在路由器bsdi上截获的tcpdump的相应输出（去掉了连接建立和所有的窗口通告）。我们连接到在主机 aix上的回显服务器并键入“ test line ”（第1行），它被回显（第2行）且回显被确认（第3行），接着我们断开了SLIP链路。

我们键入“ another line ”（第3行之后）并希望看到TCP超时和重传报文。的确，这一行在收到应答前被发送了6次。第4~13行显示了第1次传输和接着的4次重传，每个都产生了一个来自路由器sun的ICMP主机不可达。这正是我们所希望的：从 slip来的IP数据报发往路由器bsdi（这是一个指向sun的默认路由器），并到达检测到链路中断的sun。

在发生这些重传时， SLIP链路又被连通，在第14行的重传被交付。第15行是来自aix的回显，而第16行是对这个回显的确认。

这表明TCP忽略ICMP主机不可达的差错并坚持重传。我们也可以观察到所预期的在每一次重传超时中的指数退避：第1次约为2.5秒，接着乘2（约5秒），乘4（约10秒），乘8（约20秒），乘14（约40秒）。

接着我们键入输入的第3行（“ line number 3 ”）并看到它在第17行被发送，在第18行回显，并在第19行对回显进行确认。

```
1     0.0                  slip.1035 > aix.echo: P 1:11(10) ack 1
2     0.212271 (  0.2123)  aix.echo > slip.1035: P 1:11(10) ack 11
3     0.310685 (  0.0984)  slip.1035 > aix.echo: . ack 11
```

SLIP链路此时被挂断

```
4   174.758100 (174.4474)  slip.1035 > aix.echo: P 11:24(13) ack 11
5   174.759017 (  0.0009)  sun > slip: icmp: host aix unreachable
6   177.150439 (  2.3914)  slip.1035 > aix.echo: P 11:24(13) ack 11
7   177.151271 (  0.0008)  sun > slip: icmp: host aix unreachable
8   182.150200 (  4.9989)  slip.1035 > aix.echo: P 11:24(13) ack 11
9   182.151189 (  0.0010)  sun > slip: icmp: host aix unreachable
10  192.149671 (  9.9985)  slip.1035 > aix.echo: P 11:24(13) ack 11
11  192.150608 (  0.0009)  sun > slip: icmp: host aix unreachable
12  212.148783 ( 19.9982)  slip.1035 > aix.echo: P 11:24(13) ack 11
13  212.149786 (  0.0010)  sun > slip: icmp: host aix unreachable
```

SLIP链路此时被建立

```
14  252.146774 ( 39.9970)  slip.1035 > aix.echo: P 11:24(13) ack 11
15  252.439257 (  0.2925)  aix.echo > slip.1035: P 11:24(13) ack 24
16  252.505331 (  0.0661)  slip.1035 > aix.echo: . ack 24
17  261.977246 (  9.4719)  slip.1035 > aix.echo: P 24:38(14) ack 24
18  262.158758 (  0.1815)  aix.echo > slip.1035: P 24:38(14) ack 38
19  262.305086 (  0.1463)  slip.1035 > aix.echo: . ack 38
```

SLIP链路此时被挂断

```
20  458.155330 (195.8502)  slip.1035 > aix.echo: P 38:52(14) ack 38
21  458.156163 (  0.0008)  sun > slip: icmp: host aix unreachable
22  461.136904 (  2.9807)  slip.1035 > aix.echo: P 38:52(14) ack 38
23  461.137826 (  0.0009)  sun > slip: icmp: host aix unreachable
24  467.136461 (  5.9986)  slip.1035 > aix.echo: P 38:52(14) ack 38
25  467.137385 (  0.0009)  sun > slip: icmp: host aix unreachable
26  479.135811 ( 11.9984)  slip.1035 > aix.echo: P 38:52(14) ack 38
27  479.136647 (  0.0008)  sun > slip: icmp: host aix unreachable
28  503.134816 ( 23.9982)  slip.1035 > aix.echo: P 38:52(14) ack 38
29  503.135740 (  0.0009)  sun > slip: icmp: host aix unreachable
```

在这里14行输出结果被删除

```
44  1000.219573 ( 64.0959)  slip.1035 > aix.echo: P 38:52(14) ack 38
45  1000.220503 (  0.0009)  sun > slip: icmp: host aix unreachable
46  1064.201281 ( 63.9808)  slip.1035 > aix.echo: R 52:52(0) ack 38
47  1064.202182 (  0.0009)  sun > slip: icmp: host aix unreachable
```

图21-12 TCP对接收到的ICMP主机不可达差错的处理

现在我们希望观察在接收到ICMP主机不可达后，TCP重传并放弃的情况。于是再次断开SLIP链路，之后键入“ the last line ”，并观察到在TCP放弃之前该行被发送了13次（我们已经从结果中删除了第30~43行，它们是额外的重传）。

然而，我们所观察到的现象是sock程序在最终放弃时打印出来的差错信息：“ 没有到达主机的路由 ”。这与Unix的ICMP主机不可达的差错类似（图6-12）。这表明TCP保存了它在连接上收到的ICMP差错，并在最终放弃时打印出该差错，而不是“ 连接超时 ”。

最后，注意到第22~46行与第6~14行不同的重传间隔。看起来我们键入的第3行在第17~19行被发送和确认时（无任何重传），TCP更新了它的估计器。最初的重传超时时间现在是3秒，后续取值为6, 12, 24, 48，直至上限64。

21.11 重新分组

当TCP超时并重传时，它不一定要重传同样的报文段。相反， TCP允许进行重新分组而发送一个较大的报文段，这将有助于提高性能（当然，这个较大的报文段不能够超过接收方声明的MSS）。在协议中这是允许的，因为TCP是使用字节序号而不是报文段序号来进行识别它所要发送的数据和进行确认。

在实际中，可以很容易地看到这一点。我们使用 sock程序连接到丢弃服务器并键入一行。接着拔掉以太网电缆并再键入一行。当这一行被重传时，键入第 3行。我们预期下一个重传包含第2次和第3次键入的数据。

```
bsdi %sock svr4 discard
hello there                          第一行发送成功
                                     接着我们断开以太网电缆
line number 2                        本行被重传
and 3                                在第2行发送成功之前键入本行
                                     接着重新连接以太网电缆
```

图21-13显示了tcpdump的输出（去掉了连接建立、连接终止以及所有的窗口通告）。

```
1   0.0                 bsdi.1032 > svr4.discard: P 1:13(12) ack 1
2   0.140489 ( 0.1405)  svr4.discard > bsdi.1032: . ack 13
                                  此时断开以太网电缆
3   26.407696 (26.2672) bsdi.1032 > svr4.discard: P 13:27(14) ack 1
4   27.639390 ( 1.2317) bsdi.1032 > svr4.discard: P 13:27(14) ack 1
5   30.639453 ( 3.0001) bsdi.1032 > svr4.discard: P 13:27(14) ack 1
                                  此时键入第3行
6   36.639653 ( 6.0002) bsdi.1032 > svr4.discard: P 13:33(20) ack 1
7   48.640131 (12.0005) bsdi.1032 > svr4.discard: P 13:33(20) ack 1
                                  此时重新连接以太网电缆
8   72.640768 (24.0006) bsdi.1032 > svr4.discard: P 13:33(20) ack 1
9   72.719091 ( 0.0783) svr4.discard > bsdi.1032: . ack 33
```

图21-13 TCP对数据的重新分组

第1行和第2行显示了头一行（“ hello there ”）被发送及其ACK。接着我们拔掉以太网电缆并键入“ line number 2 ”（14字节，包括换行）。这些数据在第3行被发送，并在第4和第5行被重传。

在第6行重传前，我们键入“ and 3 ”（6个字节，包括换行），并观察到这个重传包括20个字节：键入的两行。当ACK在第9行到达时，它确认了这20字节的数据。

21.12 小结

本章提供了对TCP超时和重传机制的详细研究。使用的第 1个例子是一个丢失的建立连接的SYN，并观察了在随后的重传和超时中怎样使用指数退避方式。

TCP计算往返时间并使用这些测量结果来维护一个被平滑的 RTT估计器和被平滑的均值偏差估计器。这两个估计器用来计算下一个重传时间。许多实现对每个窗口仅测量一次 RTT。Karn算法在分组丢失时可以不测量RTT就能解决重传的二义性问题。

详细例子包括3个丢失的分组，使我们看到TCP的许多实际算法：慢启动、拥塞避免、快速重传和快速恢复。我们也能够使用拥塞窗口和慢启动门限来手工计算 TCP RTT估计器，并将这些值与跟踪输出的实际数据进行比较。

以多种ICMP差错对TCP连接的影响以及TCP怎样允许对数据进行重新分组来结束本章。我们观察到“软”的ICMP差错没有引起TCP连接终止，但这些差错被保存以便在连接非正常中止时能够报告这些软差错。

习题

21.1 在图21-5中第1个超时时间计算为6秒而第2个为12秒。如果初始SYN的确认在12秒超时溢出时还没有到达，则下一次超时在什么时候发生？

21.2 在图21-5后面的讨论中，我们提到计算的超时间隔分别为图 4-5中表示的6、24和48秒。但是如果观察一个从SVR4系统到一个不存在的主机的连接，则超时间隔分别为 6, 12, 24和48秒。请问发生了什么情况？

21.3 按下面的描述比较TCP滑动窗口协议与TFTP的停止等待协议的性能。在本章中，我们在35秒（图21-6）内传输32768字节的数据，其中链路的平均RTT是1.5秒（图21-4）。计算在同样条件下TFTP需要多长时间？

21.4 在第21.7节，我们提到过收到一个重复的 ACK是因为一个报文段丢失或重新进行排序。在21.5节我们看到1个丢失的报文段产生一些重复的 ACK。请画图表示重新排序也会产生一些重复的ACK。

21.5 在图21-6中的时刻28.8和29.8之间有一个显而易见的点，请问这是不是一个重传？

21.6 在21.6节我们提到过，如果目的地址位于一个不同的网络上， 4.3BSD Tahoe版本只执行慢启动。你认为在这里“不同的网络”是由什么决定的？（提示：参看附录 E）。

21.7 在20.2节我们提到过，在正常情况下， TCP每隔一个报文段进行一次确认，但是在图 21-2中，我们看到接收方对每个报文段都进行了确认，请解释其中的原因？

21.8 如果默认路由占优势，那么每路由（per-route）的度量是否真的有用？

第22章 TCP的坚持定时器

22.1 引言

我们已经看到TCP通过让接收方指明希望从发送方接收的数据字节数（即窗口大小）来进行流量控制。如果窗口大小为0会发生什么情况呢？这将有效地阻止发送方传送数据，直到窗口变为非0为止。

可以在图20-3中看到这种情况。当发送方接收到报文段 9时，它打开被报文段 8关闭的窗口并立即开始发送数据。 TCP必须能够处理打开此窗口的ACK（报文段9）丢失的情况。 ACK的传输并不可靠，也就是说， TCP不对ACK报文段进行确认， TCP只确认那些包含有数据的ACK报文段。

如果一个确认丢失了，则双方就有可能因为等待对方而使连接终止：接收方等待接收数据（因为它已经向发送方通告了一个非 0的窗口），而发送方在等待允许它继续发送数据的窗口更新。为防止这种死锁情况的发生，发送方使用一个坚持定时器 (persist timer)来周期性地向接收方查询，以便发现窗口是否已增大。这些从发送方发出的报文段称为窗口探查 (window probe)。在本章中，我们将讨论窗口探查和坚持定时器，还将讨论与坚持定时器有关的糊涂窗口综合症。

22.2 一个例子

为了观察到实际中的坚持定时器，我们启动一个接收进程。它监听来自客户的连接请求，接受该连接请求，然后在从网上读取数据前休眠很长一段时间。

sock程序可以通过指定一个暂停选项 -P使服务器在接受连接和进行第一次读动作之间进入休眠。我们以这种方式调用服务器：

```
svr4 %sock -i -s -P100000 5555
```

该命令在从网络上读数据之前休眠 100 000秒（27.8小时）。客户运行在主机bsdi上，并向服务器的5555端口执行1024字节的写操作。图22-1给出了tcpdump的输出结果（我们已经在结果中去掉了连接的建立过程）。

报文段1~13显示的是从客户到服务器的正常的数据传输过程，有 9216个字节的数据填充了窗口。服务器通告窗口大小为 4096字节，且默认的插口缓存大小为 4096字节。但实际上它一共接收了9216字节的数据，这是在SVR4中TCP代码和流子系统(stream subsystem)之间某种形式交互的结果。

在报文段13中，服务器确认了前面 4个数据报文段，然后通告窗口为 0，从而使客户停止发送任何其他的数据。这就引起客户设置其坚持定时器。如果在该定时器时间到时客户还没有接收到一个窗口更新，它就探查这个空的窗口以决定窗口更新是否丢失。由于服务器进程处于休眠状态，所以TCP缓存9216字节的数据并等待应用进程读取。

请注意客户发出的窗口探查之间的时间间隔。在收到一个大小为 0的窗口通告后的第1个

（报文段14）间隔为4.949秒，下一个（报文段16）间隔是4.996秒，随后的间隔分别约为6, 12, 24, 48和60秒。

```
 1    0.0                bsdi.1027 > svr4.5555: P 1:1025(1024) ack 1 win 4096
 2    0.191961 ( 0.1920) svr4.5555 > bsdi.1027: . ack 1025 win 4096
 3    0.196950 ( 0.0050) bsdi.1027 > svr4.5555: . 1025:2049(1024) ack 1 win 4096
 4    0.200340 ( 0.0034) bsdi.1027 > svr4.5555: . 2049:3073(1024) ack 1 win 4096
 5    0.207506 ( 0.0072) svr4.5555 > bsdi.1027: . ack 3073 win 4096
 6    0.212676 ( 0.0052) bsdi.1027 > svr4.5555: . 3073:4097(1024) ack 1 win 4096
 7    0.216113 ( 0.0034) bsdi.1027 > svr4.5555: P 4097:5121(1024) ack 1 win 4096
 8    0.219997 ( 0.0039) bsdi.1027 > svr4.5555: P 5121:6145(1024) ack 1 win 4096
 9    0.227882 ( 0.0079) svr4.5555 > bsdi.1027: . ack 5121 win 4096
10    0.233012 ( 0.0051) bsdi.1027 > svr4.5555: P 6145:7169(1024) ack 1 win 4096
11    0.237014 ( 0.0040) bsdi.1027 > svr4.5555: P 7169:8193(1024) ack 1 win 4096
12    0.240961 ( 0.0039) bsdi.1027 > svr4.5555: P 8193:9217(1024) ack 1 win 4096
13    0.402143 ( 0.1612) svr4.5555 > bsdi.1027: . ack 9217 win 0

14    5.351561 ( 4.9494) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
15    5.355571 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

16   10.351714 ( 4.9961) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
17   10.355670 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

18   16.351881 ( 5.9962) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
19   16.355849 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

20   28.352213 (11.9964) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
21   28.356178 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

22   52.352874 (23.9967) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
23   52.356839 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

24  100.354224 (47.9974) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
25  100.358207 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

26  160.355914 (59.9977) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
27  160.359835 ( 0.0039) svr4.5555 > bsdi.1027: . ack 9217 win 0

28  220.357575 (59.9977) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
29  220.361668 ( 0.0041) svr4.5555 > bsdi.1027: . ack 9217 win 0

30  280.359254 (59.9976) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
31  280.363315 ( 0.0041) svr4.5555 > bsdi.1027: . ack 9217 win 0
```

图22-1 坚持定时器探查一个0大小窗口的例子

为什么这些间隔总是比5、6、12、24、48和60小一个零点几秒呢？因为这些探查被TCP的500 ms定时器超时例程所触发。当定时器时间到时，就发送窗口探查，并大约在4 ms之后收到一个应答。接收到应答使得定时器被重新启动，但到下一个时钟滴答之间的时间则约为500减4 ms。

计算坚持定时器时使用了普通的TCP指数退避。对一个典型的局域网连接，首次超时时间算出来是1.5秒，第2次的超时值增加一倍，为3秒，再下次乘以4为6秒，之后再乘以8为12秒等。但是坚持定时器总是在5~60秒之间，这与我们在图22-1中观察到的现象一致。

窗口探查包含一个字节的数据（序号为9217）。TCP总是允许在关闭连接前发送一个字节的数据。请注意，尽管如此，所返回的窗口为0的ACK并不是确认该字节（它们确认了包括9216在内的所有数据），因此这个字节被持续重传。

坚持状态与第21章中介绍的重传超时之间一个不同的特点就是TCP从不放弃发送窗口探查。这些探查每隔60秒发送一次，这个过程将持续到或者窗口被打开，或者应用进程使用的连接被终止。

22.3 糊涂窗口综合症

基于窗口的流量控制方案，如TCP所使用的，会导致一种被称为“糊涂窗口综合症SWS

(Silly Window Syndrome)”的状况。如果发生这种情况，则少量的数据将通过连接进行交换，而不是满长度的报文段[Clark 1982]。

该现象可发生在两端中的任何一端：接收方可以通告一个小的窗口（而不是一直等到有大的窗口时才通告），而发送方也可以发送少量的数据（而不是等待其他的数据以便发送一个大的报文段）。可以在任何一端采取措施避免出现糊涂窗口综合症的现象。

1) 接收方不通告小窗口。通常的算法是接收方不通告一个比当前窗口大的窗口(可以为0)，除非窗口可以增加一个报文段大小（也就是将要接收的 MSS）或者可以增加接收方缓存空间的一半，不论实际有多少。

2) 发送方避免出现糊涂窗口综合症的措施是只有以下条件之一满足时才发送数据： (a)可以发送一个满长度的报文段； (b)可以发送至少是接收方通告窗口大小一半的报文段； (c)可以发送任何数据并且不希望接收 ACK（也就是说，我们没有还未被确认的数据）或者该连接上不能使用Nagle算法（见第19.4节）。

条件(b)主要对付那些总是通告小窗口（也许比 1个报文段还小）的主机，条件 (c)使我们在有尚未被确认的数据（正在等待被确认）以及在不能使用 Nagle算法的情况下，避免发送小的报文段。如果应用进程在进行小数据的写操作（例如比该报文段还小），条件(c)可以避免出现糊涂窗口综合症。

这三个条件也可以让我们回答这样一个问题：在有尚未被确认数据的情况下，如果 Nagle算法阻止我们发送小的报文段，那么多小才算是小呢？从条件 (a)中可以看出所谓“小”就是指字节数小于报文段的大小。条件 (b)仅用来对付较老的、原始的主机。

步骤2中的条件(b)要求发送方始终监视另一方通告的最大窗口大小，这是一种发送方猜测对方接收缓存大小的企图。虽然在连接建立时接收缓存的大小可能会减小，但在实际中这种情况很少见。

一个例子

现在我们通过仔细查看一个详细的例子来观察实际避免出现糊涂窗口综合症的情况，该例子也包括了坚持定时器。我们将在发送主机 sun上运行sock程序，并向网络写 6个1024字节的数据。

```
sun % sock -i -n6 bsdi 7777
```

但是在主机bsdi的接收过程中我们加入一些暂停。在第 1次读数据前暂停4秒，之后每次读之前暂停2秒。而且，接收方进行的是256字节的读操作：

```
bsdi %sock -i -s -P4 -p2 -r256 7777
```

最初的暂停是为了让接收缓存被填满，迫使发送方停止发送。随后由于接收方从网络上进行了一些小数目的读取，我们预期能看到接收方采取的避免糊涂窗口综合症的措施。

图22-2是传输6144字节数据的时间系列（我们去掉了连接建立过程）。

我们还需要跟踪在每个时间点上读取数据时应用程序的运行情况、当前正在接收缓存中的数据的序号以及接收缓存中可用空间的大小。图 22-3显示了所发生的每件事情。

图22-3中的第1列是每个行为的相对时间点。那些带有3位小数点的时间是从tcpdump的输出结果（图22-2）中得到的，而小数点部分为 99的则是在接收服务器上产生行为的估计时间(使这些在接收方的估计时间包含一秒的99%仅与图22-2中的报文段20和22有关，它们是我们能

够从tcpdump的输出结果中看到的由接收主机超时引起的仅有的两个事件。而在主机 bsdi上观察到的其他分组，则是由接收到来自发送方的一个报文段所引起的。这同样是有意义的，因为这就使我们可以将最初的4秒暂停刚好放置在发送方发送第1个数据报文段的时间0前面。这是接收方在连接建立过程中收到它的SYN的ACK之后将要获得控制权的大致时间)。

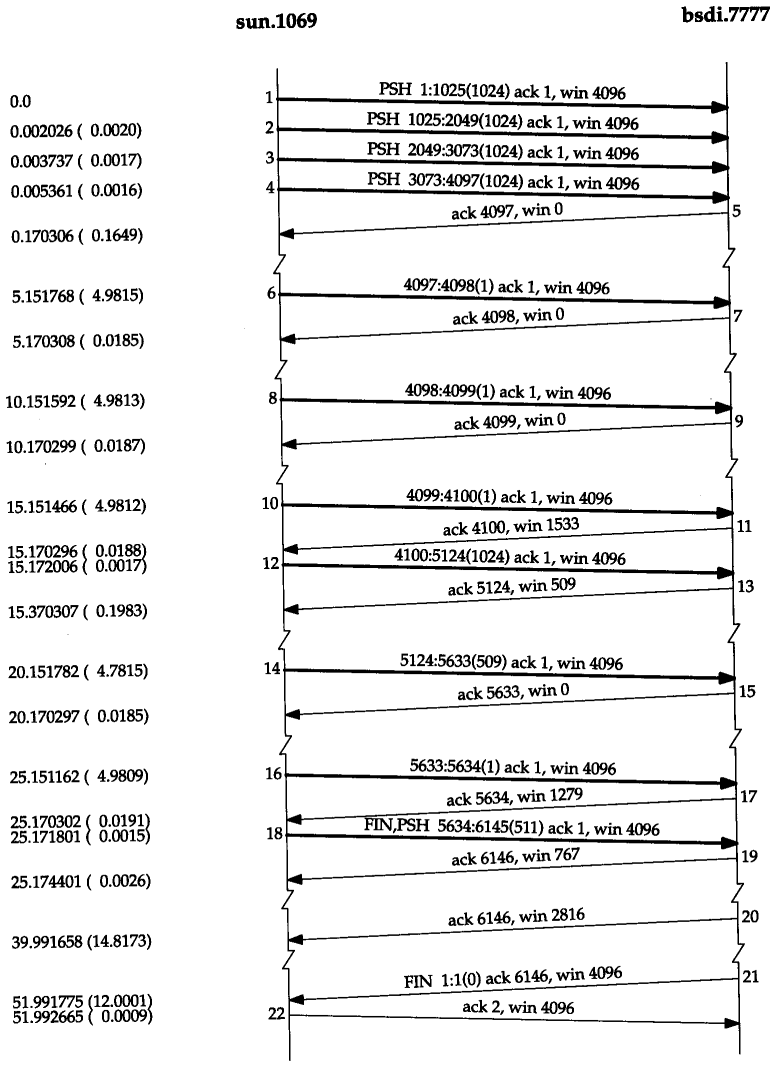


图22-2 显示接收方避免出现糊涂窗口综合症的时间系列

当接收到来自发送方的数据时，接收方缓存中的数据增加，而当应用进程从缓存中读取数据时，数据就减少。接下来我们关注的是接收方发给发送方的窗口通告以及这些窗口通告是什么。这样就可以使我们看到接收方是如何避免糊涂窗口综合症的。

前4个数据报文段及其ACK（报文段1~5）表示发送方正在填充接收方的缓存。在那个时刻发送方停止了发送，但仍然有更多的数据需要发送。它将自己的坚持定时器置为最小值5分钟。

当坚持定时器时间到时，就发送出1个字节的数据（报文段6）。接收的应用进程已经从接收缓存中读取了256字节的数据（在时刻3.99），因此这个字节被接受并被确认（报文段7段）。但是通告窗口仍为0，由于接收方仍然没有足够的空间来接收一个满长度的报文，或者不能腾

出缓存空间的一半。这就是接收方的糊涂窗口避免措施。

时间	报文段号 (图22-2)	行为			接收缓冲区	
		发送TCP	接收TCP	应用	数据	可用的
0.000	1	1:1025(1024)			1024	3072
0.002	2	1025:2049(1024)			2048	2048
0.003	3	2049:3073(1024)			3072	1024
0.005	4	3073:4097(1024)			4096	0
0.170	5		ACK 4097, win 0			
3.99				读取256	3840	256
5.151	6	4097:4098(1)			3841	255
5.17	7		ACK 4098, win 0			
5.99				读取256	3585	511
7.99				读取256	3329	767
9.99				读取256	3073	1023
10.151	8	4098:4099(1)			3074	1022
10.170	9		ACK 4099, win 0			
11.99				读取256	2818	1278
13.99				读取256	2562	1534
15.151	10	4099:4100(1)			2563	1533
15.170	11		ACK 4100, win 1533			
15.172	12	4100:5124(1024)			3587	509
15.370	13		ACK 5124, win 509			
15.99				读取256	3331	765
17.99				读取256	3075	1021
19.99				读取256	2819	1277
20.151	14	5124:5633(509)			3328	768
20.170	15		ACK 5633, win 0			
21.99				读取256	3072	1024
23.99				读取256	2816	1280
25.151	16	5633:5634(1)			2817	1279
25.170	17		ACK 5634, win 1279			
25.171	18	5634:6145(511)			3328	768
25.174	19		ACK 6146, win 767			
25.99				读取256	3072	1024
27.99				读取256	2816	1280
29.99				读取256	2560	1536
31.99				读取256	2304	1792
33.99				读取256	2048	2048
35.99				读取256	1792	2304
37.99				读取256	1536	2560
39.99				读取256	1280	2816
39.99	20		ACK 6146, win 2816			
41.99				读取256	1024	3072
43.99				读取256	768	3328
45.99				读取256	512	3584
47.99				读取256	256	3840
49.99				读取256	0	4096
51.99				读取256(EOF)	0	4096
51.991	21		ACK 6146, win 4096			
51.992	22	ACK 2				

图22-3 接收方避免出现糊涂窗口综合症的事件序列

发送方的坚持定时器被复位，并在5秒后再次到时（在时刻10.151）。然后又发送一个字节并被确认（报文段8和9），而接收方的缓存空间还不够用（1022字节），使得通告窗口为0。

发送方的坚持定时器在时刻 15.151再次时间到，又发送了另一个字节并被确认（报文段10和11）。这一次由于接收方有 1533字节的有效缓存空间，因此通告了一个非 0窗口。发送方立即利用这个窗口发送了1024字节的数据（报文段12）。对这1024字节数据的确认（报文段13）通告其窗口为509字节。这看起来与我们在前面看到的小窗口通告相抵触。

在这里之所以发生这种情况，是因为报文段 11段通告了一个大小为 1533字节的窗口，而发送方只使用了其中的1024字节。如果在报文段13中的ACK通告其窗口为0，就会违反窗口的右边沿不能向左边沿移动而导致窗口收缩的 TCP原则（见第20.3节）。这就是为什么必须通告

一个509字节的窗口的原因。

接下来我们看到发送方没有立即向这个小窗口发送数据。这就是发送方采取的糊涂窗口避免策略。相反，它等待另一个坚持定时器在时刻 20.151到时间，并在该时刻发送 509字节的数据。尽管它最终还是发送了一个长度为 509字节的小数据段，但在发送前它等待了 5秒钟，看是否会有一个ACK到达，以便可以将窗口开得更大。这 509字节的数据使得接收缓存仅剩下768字节的有效空间，因此接收方通告窗口为 0（报文段15）。

坚持定时器在时刻25.151再次到时间，发送方发送 1个字节，于是接收缓存中有 1279字节的可用空间，这就是在报文段 17所通告的窗口大小。

发送方只有另外的511个字节的数据需要发送，因此在收到 1279的窗口通告后立刻发送了这些数据（报文段 18）。这个报文段也带有 FIN标志。接收方确认数据和 FIN，并通告窗口大小为767（见习题22.2）。

由于发送应用进程在执行完 6个1024字节的写操作后发出关闭命令，发送方的连接从ESTABLISHED状态转变到FIN_WAIT_1状态，再到FIN_WAIT_2状态（见图18-12）。它一直处于这个状态，直到收到对方的 FIN。在这个状态上没有设置定时器（回忆我们在 18.6节结束时的讨论），因为它在报文段18中发送的FIN被报文段19确认。这就是为什么我们看到发送方直到接收到FIN（报文段21）为止没有发送其他任何数据的原因。

接收应用进程继续每隔 2秒从接收缓存区中读取 256个字节的数据。为什么在时刻 39.99发送ACK（报文段20）呢？这是因为应用进程在时刻 39.99读取数据时，接收缓存中的可用空间已经从原来通告的767（报文段19）变为2816，这相当于接收缓存中增加了额外的 2049字节的空间。回忆本节开始讲的第 1个规则，因为现在接收缓存已经增加了其空间的一半，因此接收方现在发送窗口更新。这意味着每次当应用进程从 TCP的接收缓存中读取数据时，接收的 TCP将检查是否需要更新发送窗口。

应用进程在时间51.99发出最后一个读操作，然后收到一个文件结束标志，因为缓存已经变空。这就导致了最后两个完成连接终止的报文段（报文段 21和22）的发送。

22.4 小结

在连接的一方需要发送数据但对方已通告窗口大小为0时，就需要设置TCP的坚持定时器。发送方使用与第21章类似的重传间隔时间，不断地探查已关闭的窗口。这个探查过程将一直持续下去。

当运行一个例子来观察坚持定时器时，我们还观察到了 TCP的避免出现糊涂窗口综合症的现象。这就是使 TCP避免通告小的窗口大小或发送小的报文段。在我们的例子中，可以观察到发送方和接收方为避免糊涂窗口综合症所使用的策略。

习题

22.1 在图22-3中注意到所有确认（报文段5、7、9、11、13、15和17）的发送时刻为：0.170, 5.170、10.170、15.170、20.170和25.170，还注意到在接收数据和发送ACK之间的时间差分别为：164.5、18.5、18.7、18.8、198.3、18.5和19.1 ms。试解释可能会发生的情况。

22.2 在图22-3中的时刻25.174，发送出一个767字节的通告窗口，而在接收缓存中有 768字节的可用空间。为什么相差1个字节？

第23章 TCP的保活定时器

23.1 引言

许多TCP/IP的初学者会很惊奇地发现可以没有任何数据流通过一个空闲的 TCP连接。也就是说，如果TCP连接的双方都没有向对方发送数据，则在两个TCP模块之间不交换任何信息。例如，没有可以在其他网络协议中发现的轮询。这意味着我们可以启动一个客户与服务器建立一个连接，然后离去数小时、数天、数个星期或者数月，而连接依然保持。中间路由器可以崩溃和重启，电话线可以被挂断再连通，但是只要两端的主机没有被重启，则连接依然保持建立。

这意味着两个应用进程 —— 客户进程或服务器进程 —— 都没有使用应用级的定时器来检测非活动状态，而这种非活动状态可以导致应用进程中的任何一个终止其活动。回想在第10.7节末尾曾提到过的BGP每隔30秒就向对端发送一个应用的探查，就是独立于 TCP的保活定时器之外的应用定时器。

然而，许多时候一个服务器希望知道客户主机是否崩溃并关机或者崩溃又重新启动。许多实现提供的保活定时器可以提供这种能力。

> 保活并不是TCP规范中的一部分。Host Requirements RFC提供了3个不使用保活定时器的理由：(1) 在出现短暂差错的情况下，这可能会使一个非常好的连接释放掉；(2) 它们耗费不必要的带宽；(3) 在按分组计费的情况下会在互联网上花掉更多的钱。然而，许多实现提供了保活定时器。

保活定时器是一个有争论的功能。许多人认为如果需要，这个功能不应该在 TCP中提供，而应该由应用程序来完成。这是应当认真对待的一些问题之一，因为在这个论题上有些人表达出了很大的热情。

在连接两个端系统的网络出现临时故障的时候，保活选项会引起一个实际上很好的连接终止。例如，如果在一个中间路由器崩溃并重新启动时发送保活探查，那么 TCP会认为客户的主机已经崩溃，而实际上所发生的并非如此。

保活功能主要是为服务器应用程序提供的。服务器应用程序希望知道客户主机是否崩溃，从而可以代表客户使用资源。许多版本的 Rlogin和Telnet服务器默认使用这个选项。

一个说明现在需要使用保活功能的常见例子是当个人计算机用户使用 TCP/IP向一个使用Telnet的主机注册时。如果在一天结束时，他们仅仅关闭了电源而没有注销，那么便会留下一个半开放的连接。在图18-16中，我们看到通过一个半开放连接发送数据会导致返回一个复位，但那是在来自正在发送数据的客户端。如果客户已经消失了，使得在服务器上留下一个半开放连接，而服务器又在等待来自客户的数据，则服务器将永远等待下去。保活功能就是试图在服务器端检测到这种半开放的连接。

23.2 描述

在这个描述中，我们称使用保活选项的一端为服务器，而另一端则为客户。并没有什么使客户不能使用这个选项，但通常都是服务器设置这个功能。如果双方都特别需要了解对方是否已经消失，则双方都可以使用这个选项（在29章我们将看到NFS使用TCP时，客户和服务器都设置了这个选项。但在第26章讲到Telnet和Rlogin时，只有服务器设置了这个选项，而客户则没有）。

如果一个给定的连接在两个小时之内没有任何动作，则服务器就向客户发送一个探查报文段（我们将在随后的例子中看到这个探查报文段看起来像什么）。客户主机必须处于以下4个状态之一。

1) 客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常工作的。服务器在两小时以后将保活定时器复位。如果在两个小时定时器到时间之前有应用程序的通信量通过此连接，则定时器在交换数据后的未来2小时再复位。

2) 客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务器将不能够收到对探查的响应，并在75秒后超时。服务器总共发送10个这样的探查，每个间隔75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。

3) 客户主机崩溃并已经重新启动。这时服务器将收到一个对其保活探查的响应，但是这个响应是一个复位，使得服务器终止这个连接。

4) 客户主机正常运行，但是从服务器不可达。这与状态2相同，因为TCP不能够区分状态4与状态2之间的区别，它所能发现的就是没有收到探查的响应。

服务器不用关注客户主机被关闭和重新启动的情况（这指的是一个操作员的关闭，而不是主机崩溃）。当系统被操作员关闭时，所有的应用进程也被终止（也就是客户进程），这会使客户的TCP在连接上发出一个FIN。接收到FIN将使服务器的TCP向服务器进程报告文件结束，使服务器可以检测到这个情况。

在第1种情况下，服务器的应用程序没有感觉到保活探查的发生。TCP层负责一切。这个过程对应用程序都是透明的，直至第2、3或4种情况发生。在这三种情况下，服务器应用程序将收到来自它的TCP的差错报告（通常服务器已经向网络发出了读操作请求，然后等待来自客户的数据。如果保活功能返回一个差错，则该差错将作为读操作的返回值返回给服务器）。在第2种情况下，差错是诸如“连接超时”之类的信息，而在第3种情况则为“连接被对方复位”。第4种情况看起来像是连接超时，也可根据是否收到与连接有关的ICMP差错来返回其他的差错。在下一节中我们将观察这4种情况。

一个被人们不断讨论的关于保活选项的问题就是两个小时的空闲时间是否可以改变。通常他们希望该数值可以小得多，处在分钟的数量级。正如我们在附录E看到的，这个值通常可以改变，但是在该附录所描述的所有系统中，保活间隔时间是系统级的变量，因此改变它会影响到所有使用该功能的用户。

Host Requirements RFC提到一个实现可提供保活的功能，但是除非应用程序指明要这样，否则就不能使用该功能。而且，保活间隔必须是可配置的，但是其默认值必须不小于两个小时。

23.3 保活举例

现在详细讨论前一节提到的第2、3和4种情况。我们将在使用这个选项的情况下检查所交换的分组。

23.3.1 另一端崩溃

首先观察另一端崩溃且没有重新启动的情况下所发生的现象。为模拟这种情况，我们采用如下步骤：

- 在客户（主机bsdi上运行的sock程序）和主机svr4上的标准回显服务器之间建立一个连接。客户使用-K选项使能保活功能。
- 验证数据可以通过该连接。
- 观察客户TCP每隔2小时发送保活分组，并观察被服务器的TCP确认。
- 将以太网电缆从服务器上拔掉直到这个例子完成，这会使客户认为服务器主机已经崩溃。
- 我们预期服务器在断定连接已中断前发送10个间隔为75秒的保活探查。

这里是客户端的交互输出结果：

```
bsdi %sock -K svr4 echo          -K是保活选项
hello, world                     开始时键入本行以验证连接有效
hello, world                     和看到回显
                                 4小时后断开以太网电缆
read error: Connection timed out这发生在启动后约6小时10分钟
```

图23-1显示的是tcpdump的输出结果（已经去掉了连接建立和窗口通告）。

```
 1      0.0                  bsdi.1055 > svr4.echo: P 1:14(13) ack 1
 2      0.006105 (   0.0061) svr4.echo > bsdi.1055: P 1:14(13) ack 14
 3      0.093140 (   0.0870) bsdi.1055 > svr4.echo: . ack 14

 4   7199.972793 (7199.8797) arp who-has svr4 tell bsdi
 5   7199.974878 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
 6   7199.975741 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
 7   7199.979843 (   0.0041) svr4.echo > bsdi.1055: . ack 14

 8  14400.134330 (7200.1545) arp who-has svr4 tell bsdi
 9  14400.136452 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
10  14400.137391 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
11  14400.141408 (   0.0040) svr4.echo > bsdi.1055: . ack 14

12  21600.318309 (7200.1769) arp who-has svr4 tell bsdi
13  21675.320373 (  75.0021) arp who-has svr4 tell bsdi
14  21750.322407 (  75.0020) arp who-has svr4 tell bsdi
15  21825.324460 (  75.0021) arp who-has svr4 tell bsdi
16  21900.436749 (  75.1123) arp who-has svr4 tell bsdi
17  21975.438787 (  75.0020) arp who-has svr4 tell bsdi
18  22050.440842 (  75.0021) arp who-has svr4 tell bsdi
19  22125.432883 (  74.9920) arp who-has svr4 tell bsdi
20  22200.434697 (  75.0018) arp who-has svr4 tell bsdi
21  22275.436788 (  75.0021) arp who-has svr4 tell bsdi
```

图23-1 决定一个主机已经崩溃的保活分组

客户在第1、2和3行向服务器发送“Hello, world”并得到回显。第4行是第一个保活探查，发生在两个小时以后（7200秒）。在第6行的TCP报文段能够发送之前，首先观察到的是一个ARP请求和一个ARP应答。第6行的保活探查引出来自另一端的响应（第7行）。两个小时以后，在第7和8行发生了同样的分组交换过程。

如果能够观察到第6和第10行的保活探查中的所有字段，我们就会发现序号字段比下一个将要发送的序号字段小1（在本例中，当下一个为14时，它就是13）。但是因为报文段中没有数据，tcpdump不能打印出序号字段（它仅能够打印出设置了SYN、FIN或RST标志的空数据的序号）。正是接收到这个不正确的序号，才导致服务器的TCP对保活探查进行响应。这个响应告诉客户，服务器下一个期望的序号是14。

一些基于4.2BSD的旧的实现不能够对这些保活探查进行响应，除非报文段中包含数据。某些系统可以配置成发送一个字节的无用数据来引出响应。这个无用数据是无害的，因为它不是所期望的数据（这是接收方前一次接收并确认的数据），因此它会被接收方丢弃。其他一些系统在探查的前半部分发送4.3BSD格式的报文段（不包含数据），如果没有收到响应，在后半部分则切换为4.2BSD格式的报文段。

接着我们拔掉电缆，并期望两个小时的再一次探查失败。当这下一个探查发生时，注意到从来没有看到电缆上出现TCP报文段，这是因为主机没有响应ARP请求。在放弃之前，我们仍可以观察到客户每隔75秒发送一个探查，一共发送了10次。从交互式脚本可以看到返回给客户进程的差错码被TCP转换为“连接超时”，这正是实际所发生的。

23.3.2 另一端崩溃并重新启动

在这个例子中，我们可以观察到当客户崩溃并重新启动时发生的情况。最初的环境与前一个例子相似，但是在我们验证连接有效之后，我们将服务器从以太网上断开，重新启动，然后再连接到网络上。我们希望看到下一个保活探查产生一个来自服务器的复位，因为现在服务器不知道关于这个连接的任何信息。这是交互会话的过程：

```
bsdi %sock -K svr4 echo              -K使保活选项有效
hi, there                            键入这行以验证连接有效
hi, there                            这是来自另一端的回显
                                     从以太网断连后，服务器这时重新启动
read error: Connection reset by peer
```

图23-2显示的是tcpdump的输出结果（已经去掉了连接建立和窗口通告）。

```
1     0.0                    bsdi.1057 > svr4.echo: P 1:10(9) ack 1
2     0.006406 (   0.0064)   svr4.echo > bsdi.1057: P 1:10(9) ack 10
3     0.176922 (   0.1705)   bsdi.1057 > svr4.echo: . ack 10

4  7200.067151 (7199.8902)   arp who-has svr4 tell bsdi
5  7200.069751 (   0.0026)   arp reply svr4 is-at 0:0:c0:c2:9b:26
6  7200.070468 (   0.0007)   bsdi.1057 > svr4.echo: . ack 10
7  7200.075050 (   0.0046)   svr4.echo > bsdi.1057: R 1135563275:1135563275(0)
```

图23-2 另一端崩溃并重启时保活的例子

我们建立了连接，并从客户发送9个字节的数据到服务器（第1~3行）。两个小时之后，客户发送第1个保活探查，其响应是一个来自服务器的复位。客户应用进程打印出“连接被对端复位”的差错，这是有意义的。

23.3.3 另一端不可达

在这个例子中，客户没有崩溃，但是在保活探查发送后的10分钟内无法到达，可能是一个中间路由器已经崩溃，或一条电话线临时出现故障，或发生了其他一些类似的情况。

为了仿真这个例子，我们从主机 slip经过一个拨号 SLIP链路与主机 vangogh.cs.berkeley.edu建立一个连接，然后断掉链路。这里是交互输出的结果：

```
bsdi %sock -K vangogh.cs.berkeley.edu echo
testing                                          我们键入这行
testing                                          看到这行的回显
                                                 在某个时刻这条SLIP链路被断开
read error: No route to host
```

图23-3显示了在路由器bsdi上收集到的tcpdump输出结果（已经去掉了连接建立和窗口通告）。

```
 1      0.0                  slip.1056 > vangogh.echo: P 1:9(8) ack 1
 2      0.277669 (   0.2777) vangogh.echo > slip.1056: P 1:9(8) ack 9
 3      0.424423 (   0.1468) slip.1056 > vangogh.echo: . ack 9

 4   7200.818081 (7200.3937) slip.1056 > vangogh.echo: . ack 9
 5   7201.243046 (   0.4250) vangogh.echo > slip.1056: . ack 9

 6  14400.688106 (7199.4451) slip.1056 > vangogh.echo: . ack 9
 7  14400.689261 (   0.0012) sun > slip: icmp: net vangogh unreachable

 8  14475.684360 (  74.9951) slip.1056 > vangogh.echo: . ack 9
 9  14475.685504 (   0.0011) sun > slip: icmp: net vangogh unreachable

                        删除14行

24  15075.759603 (  75.1008) slip.1056 > vangogh.echo: R 9:9(0) ack 9
25  15075.760761 (   0.0012) sun > slip: icmp: net vangogh unreachable
```

图23-3 当另一端不可达时的保活例子

我们与以前一样开始讨论这个例子：第 1~3行证实连接是有效的。两个小时之后的第 1个保活探查是正常的（第4、5行），但是在两个小时后发生下一个探查之前，我们断开在路由器sun和netb之间的SLIP连接（拓扑结构参见封）。

第6行的保活探查引发一个来自路由器 sun的ICMP网络不可达的差错。正如我们在第21.10节描述的那样，对于主机slip上接收的TCP而言，这只是一个软差错。它报告收到了一个ICMP差错，但是差错的接收者并没有终止这个连接。在发送主机最终放弃之前，一共发送了9个保活探查，间隔为 75秒。这时返回给应用进程的差错产生了一个不同的报文：“ 没有到达主机的路由 ”。我们在图6-12看到这对应于ICMP网络不可达的差错。

23.4 小结

正如我们在前面提到的，对保活功能是有争议的。协议专家继续在争论该功能是否应该归入运输层，或者应当完全由应用层来处理。

在连接空闲两个小时后，在一个连接上发送一个探查分组来完成保活功能。可能会发生 4种不同的情况：对端仍然运行正常、对端已经崩溃、对端已经崩溃并重新启动以及对端当前无法到达。我们使用一个例子来观察每一种情况，并观察到在最后三个条件下返回的不同差错。

在前两个例子中，如果没有提供这种功能，并且也没有应用层的定时器，则客户将永远无法知道对端已经崩溃或崩溃并重新启动。可是在最后一个例子中，两端都没有发生差错，只是它们之间的连接临时中断。我们在使用保活时必须关注这个限制。

习题

23.1 列出保活功能的一些优点。

23.2 列出保活功能的一些缺点。

第24章　TCP的未来和性能

24.1　引言

TCP已经在从1200 b/s的拨号SLIP链路到以太数据链路上运行了许多年。在80年代和90年代初期，以太网是运行TCP/IP最主要的数据链路方式。虽然TCP在比以太网速率高的环境（如T2电话线、FDDI及千兆比网络）中也能够正确运行，但在这些高速率环境下，TCP的某些限制就会暴露出来。

本章讨论TCP的一些修改建议，这些建议可以使TCP在高速率环境中获得最大的吞吐量。首先要讨论前面已经碰到过的路径MTU发现机制，本章主要关注它如何与TCP协同工作。这个机制通常可以使TCP为非本地的连接使用大于536字节的MTU，从而增加吞吐量。

接着介绍长肥管道(long fat pipe)，也就是那些具有很大的带宽时延乘积的网络，以及TCP在这些网络上所具有的局限性。为处理长肥管道，我们描述两个新的TCP选项：窗口扩大选项（用来增加TCP的最大窗口，使之超过65535字节）和时间戳选项。后面这个选项可以使TCP对报文段进行更加精确的RTT测量，还可以在高速率下对可能发生的序号回绕提供保护。这两个选项在RFC 1323 [Jacobson, Braden, and Borman 1992]中进行定义。

我们还将介绍建议的T/TCP，这是为增加事务功能而对TCP进行的修改。通信的事务模式以客户的请求将被服务器应答的响应为主要特征。这是客户服务器计算的常见模型。T/TCP的目的就是减少两端交换的报文段数量，避免三次握手和使用4个报文段进行连接的关闭，从而使客户可以在一个RTT和处理请求所必需的时间内收到服务器的应答。

这些新选项（路径MTU发现、窗口扩大选项、时间戳选项和T/TCP）中令人印象最深刻的就是它们与现有的TCP实现能够向后兼容，即包括这些新选项的系统仍然可以与原有的旧系统进行交互。除了在一个ICMP报文中为路径MTU发现增加了一个额外字段之外，这些新的选项只需要在那些需要使用它们的端系统中进行实现。

我们以介绍近来发表的有关TCP性能的图例作为本章的结束。

24.2　路径MTU发现

在2.9节我们描述了路径MTU的概念。这是当前在两个主机之间的路径上任何网络上的最小MTU。路径MTU发现在IP首部中继承并设置“不要分片（DF）”比特，来发现当前路径上的路由器是否需要对正在发送的IP数据报进行分片。在11.6节我们观察到如果一个待转发的IP数据报被设置DF比特，而其长度又超过了MTU，那么路由器将返回ICMP不可达的差错。在11.7节我们显示了某版本的`traceroute`程序使用该机制来决定目的地的路径MTU。在11.8节我们看到UDP是怎样处理路径MTU发现的。在本节我们将讨论这个机制是如何按照RFC 1191 [Mogul and Deering 1990]中规定的那样在TCP中进行使用的。

在本书的多种系统（参看序言）中只有Solaris 2.x支持路径MTU发现。

TCP的路径MTU发现按如下方式进行：在连接建立时， TCP使用输出接口或对端声明的MSS中的最小MTU作为起始的报文段大小。路径 MTU发现不允许TCP超过对端声明的MSS。如果对端没有指定一个MSS，则默认为536。一个实现也可以按21.9节中讲的那样为每个路由单独保存路径MTU信息。

一旦选定了起始的报文段大小，在该连接上的所有被 TCP发送的IP数据报都将被设置DF比特。如果某个中间路由器需要对一个设置了 DF标志的数据报进行分片，它就丢弃这个数据报，并产生一个我们在11.6节介绍的ICMP的“不能分片”差错。

如果收到这个ICMP差错， TCP就减少段大小并进行重传。如果路由器产生的是一个较新的该类ICMP差错，则报文段大小被设置为下一跳的 MTU减去IP和TCP的首部长度。如果是一个较旧的该类ICMP差错，则必须尝试下一个可能的最小 MTU（见图2-5）。当由这个ICMP差错引起的重传发生时，拥塞窗口不需要变化，但要启动慢启动。

由于路由可以动态变化，因此在最后一次减少路径 MTU的一段时间以后，可以尝试使用一个较大的值（直到等于对端声明的 MSS或输出接口MTU的最小值）。RFC 1191推荐这个时间间隔为10分钟（我们在11.8节看到Solaris 2.2使用一个30分钟的时间间隔）。

在对非本地目的地，默认的 MSS通常为536字节，路径MTU发现可以避免在通过 MTU小于576（这非常罕见）的中间链路时进行分片。对于本地目的主机，也可以避免在中间链路（如以太网）的MTU小于端点网络（如令牌环网）的情况下进行分片。但为了能使路径 MTU更加有用和充分利用MTU大于576的广域网，一个实现必须停止使用为非本地目的制定的 536的MTU默认值。MSS的一个较好的选择是输出接口的MTU（当然要减去IP和TCP的首部大小）（在附录E中，我们将看到大多数的实现都允许系统管理员改变这个默认的 MSS值）。

24.2.1 一个例子

在某个中间路由器的MTU比任一个端点接口MTU小的情况下，我们能够观察路径MTU发现是如何工作的。图24-1显示了这个例子的拓扑结构。

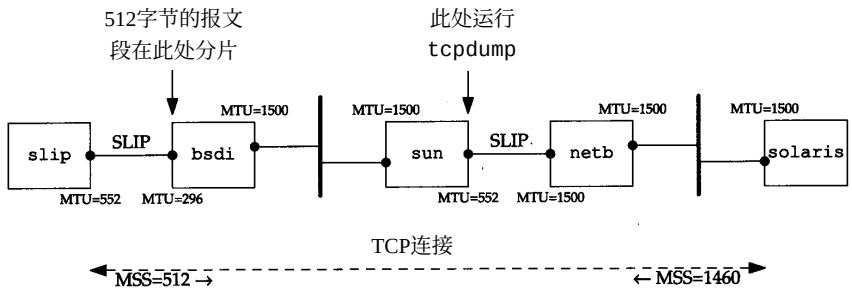


图24-1 路径MTU例子的拓扑结构

我们从主机solaris（支持路径MTU发现机制）到主机slip建立一个连接。这个建立过程与UDP的路径MTU发现（图11-13）中的一个例子相同，但在这里我们已经把 slip接口的MTU设置为552，而不是通常的296。这使得slip通告一个512的MSS。但是在bsdi上的SLIP链路上的 MTU为296，这就引起超过 256的TCP报文段被分片。于是就可以观察在solaris上的路径MTU发现是如何进行处理的。

我们在solaris上运行sock程序并向slip上的丢弃服务器进行一个512字节的写操作：

```
solaris %sock -i -n1 -w512 slip discard
```

图24-2是在主机sun的SLIP接口上收集的tcpdump的输出结果。

```
 1  0.0                 solaris.33016 > slip.discard: S 1171660288:1171660288(0)
                                                      win 8760 <mss 1460> (DF)
 2  0.101597 (0.1016)   slip.discard > solaris.33016: S 137984001:137984001(0)
                                                      ack 1171660289 win 4096
                                                      <mss 512>
 3  0.630609 (0.5290)   solaris.33016 > slip.discard: P 1:513(512)
                                                      ack 1 win 9216 (DF)
 4  0.634433 (0.0038)   bsdi > solaris: icmp:
                               slip unreachable - need to frag, mtu = 296 (DF)
 5  0.660331 (0.0259)   solaris.33016 > slip.discard: F 513:513(0)
                                                      ack 1 win 9216 (DF)
 6  0.752664 (0.0923)   slip.discard > solaris.33016: . ack 1 win 4096
 7  1.110342 (0.3577)   solaris.33016 > slip.discard: P 1:257(256)
                                                      ack 1 win 9216 (DF)
 8  1.439330 (0.3290)   slip.discard > solaris.33016: . ack 257 win 3840
 9  1.770154 (0.3308)   solaris.33016 > slip.discard: FP 257:513(256)
                                                       ack 1 win 9216 (DF)
10  2.095987 (0.3258)   slip.discard > solaris.33016: . ack 514 win 3840
11  2.138193 (0.0422)   slip.discard > solaris.33016: F 1:1(0) ack 514 win 4096
12  2.310103 (0.1719)   solaris.33016 > slip.discard: . ack 2 win 9216 (DF)
```

图24-2 路径MTU发现的tcpdump 输出结果

在第1和第2行的MSS值是我们所期望的。接着我们观察到 solaris发送一个包含512字节的数据和对SYN的确认报文段（第 3行）（在习题18.9中可以看到这种把 SYN的确认与第一个包含数据的报文段合并的情况）。这就在第4行产生了一个ICMP差错，我们看到路由器 bsdi产生较新的、包含输出接口MTU的ICMP差错。

看来在这个差错回到solaris之前，就发送了FIN（第5行）。由于slip从没有收到被路由器bsdi丢弃的512字节的数据，因此并不期望接收这个序号（ 513），所以在第6行用它期望的序号（1）进行了响应。

在这个时候， ICMP差错返回到了solaris，solaris用两个256字节的报文段（第 7和第9行）重传了512字节的数据。因为在bsdi后面可能还有具有更小的MTU的路由器，因此这两个报文段都设置了DF比特。

接着是一个较长的传输过程（持续了大约 15分钟），在最初的512字节变为256字节以后，solaris没有再尝试使用更大的报文段。

24.2.2 大分组还是小分组

常规知识告诉我们较大的分组比较好 [Mogul 1993, 15.2.8节]，因为发送较少的大分组比发送较多的小分组“ 花费 ”要少（假定分组的大小不足以引起分片，否则会引起其他方面的问题）。这些减少的花费与网络（分组首部负荷）、路由器（选路的决定）和主机（协议处理和设备中断）等有关。但并非所有的人都同意这种观点 [Bellovin 1993]。

考虑下面的例子。我们通过 4个路由器发送8192个字节，每个路由器与一个 T1电话线（1 544 000b/s）相连。首先我们使用两个4096字节的分组，如图24-3所示。

基本问题在于路由器是存储转发设备。它们通常接收整个输入分组，检验包含 IP检验和的IP首部，进行选路判决，然后开始发送输出分组。在这个图中，我们可以假定在理想情况下这些在路由器内部进行的操作不花费时间（水平点状线）。然而，从R1到R4它需要花费4个

单位时间来发送所有的8192字节。每一跳的时间为

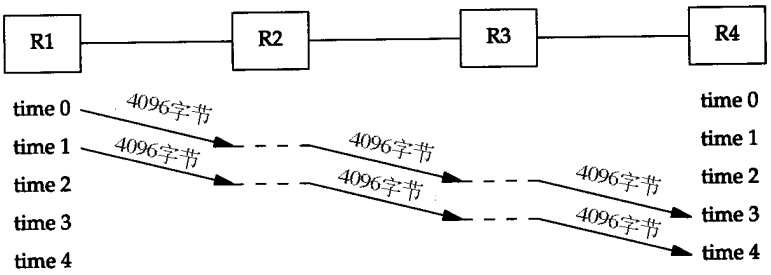


图24-3 通过4个路由器发送两个4096字节的分组

$$\frac{(4096\text{字节}+40\text{字节})\times 8\ \text{b/字节}}{1\ 544\ 000\ \text{b/s}} = 21.4\ \text{ms/跳}$$

(将TCP和IP的首部算为40字节)。发送数据的整个时间为分组个数加上跳数减 1，从图中可以看到是4个单位时间，或85.6秒。每个链路空闲2个单位时间，或42.8秒。

图24-4显示了当我们发送16个512字节的分组时所发生的情况。

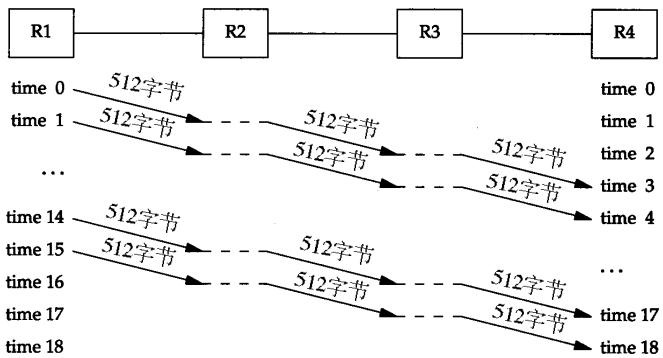


图24-4 通过4个路由器发送16个512字节的分组

这将花费更多的单位时间，但是由于发送的分组较短，因此每个单位时间较小。

$$\frac{(512\text{字节}+40\text{字节})\times 8\ \text{b/字节}}{1\ 544\ 000\ \text{b/s}} = 2.9\ \text{ms/跳}$$

现在总时间为（18×2.9）=52.2 ms。每个链路也空闲2个单位的时间，即5.8 ms。

在这个例子中，我们忽略了确认返回所需要的时间、连接建立和终止以及链路可能被其他流量共享等的影响。然而，在 [Bellovin 1993]中的测量表明，分组并不一定是越大越好。我们需要在更多的网络上对该领域进行更多的研究。

24.3 长肥管道

在20.7节，我们把一个连接的容量表示为

$$\textit{capacity}\ (\text{b}) = \textit{bandwidth}\ (\text{b/s}) \times \textit{round-trip time}\ (\text{s})$$

并称之为带宽时延乘积。也可称它为两端的管道大小。

当这个乘积变得越来越大时，TCP的某些局限性就会暴露出来。图 24-5显示了多种类型的网络的某些数值。

网 络	带宽(b/s)	RTT(ms)	带宽时延乘积(字节)
以太局域网	10 000 000	3	3 750
横跨大陆的T1电话线	1 544 000	60	11 580
卫星T1电话线	1 544 000	500	96 500
横跨大陆的T3电话线	45 000 000	60	337 500
横跨大陆的gigabit线路	1 000 000 000	60	7 500 000

图24-5 多种网络的带宽时延乘积

可以看到带宽时延乘积的单位是字节，这是因为我们用这个单位来测量每一端的缓存大小和窗口大小。

具有大的带宽时延乘积的网络被称为长肥网络（ Long Fat Network，即LFN，发音为“ elefan(t)s ”），而一个运行在LFN上的TCP连接被称为长肥管道。回顾图 20-11和图20-12，管道可以被水平拉长（一个长的 RTT），或被垂直拉高（较高的带宽），或向两个方向拉伸。使用长肥管道会遇到多种问题。

1) TCP首部中窗口大小为 16 bit，从而将窗口限制在 65535个字节内。但是从图 24-5的最后一列可以看到，现有的网络需要一个更大的窗口来提供最大的吞吐量。在 24.4节介绍的窗口扩大选项可以解决这个问题。

2) 在一个长肥网络 LFN内的分组丢失会使吞吐量急剧减少。如果只有一个报文段丢失，我们需要利用 21.7节介绍的快速重传和快速恢复算法来使管道避免耗尽。但是即使使用这些算法，在一个窗口内发生的多个分组丢失也会典型地使管道耗尽（如果管道耗尽了，慢启动会使它渐渐填满，但这个过程将需要经过多个 RTT）。

   在RFC 1072 [Jacobson and Braden 1988]中建议使用有选择的确认（SACK）来处理在一个窗口发生的多个分组丢失。但是这个功能在 RFC 1323中被忽略了，因为作者觉得在把它们纳入TCP之前需要先解决一些技术上的问题。

3) 我们在第21.4节看到许多TCP实现对每个窗口的RTT仅进行一次测量。它们并不对每个报文段进行RTT测量。在一个长肥网络LFN上需要更好的RTT测量机制。我们将在 24.5节介绍时间戳选项，它允许更多的报文段被计时，包括重传。

4) TCP对每个字节数据使用一个32 bit无符号的序号来进行标识。如果在网络中有一个被延迟一段时间的报文段，它所在的连接已被释放，而一个新的连接在这两个主机之间又建立了，怎样才能防止这样的报文段再次出现呢？首先回想起IP首部中的TTL为每个IP段规定了一个生存时间的上限——255跳或255秒，看哪一个上限先达到。在18.6节我们定义了最大的报文段生存时间（MSL）作为一个实现的参数来阻止这种情况的发生。推荐的MSL的值为2分钟（给出一个240秒的2MSL），但是我们在18.6节看到许多实现使用的MSL为30秒。

   在长肥网络LFN上，TCP的序号会碰到一个不同的问题。由于序号空间是有限的，在已经传输了4 294 967 296个字节以后序号会被重用。如果一个包含序号 *N*字节数据的报文段在网络上被迟延并在连接仍然有效时又出现，会发生什么情况呢？这仅仅是一个相同序号*N*在MSL期间是否被重用的问题，也就是说，网络是否足够快以至于在不到一个MSL的时候序号就发生了回绕。在一个以太网上要发送如此多的数据通常需要 60分钟左右，因此不会发生这种情况。但是在带宽增加时，这个时间将会减少：一个 T3的电话线（45 Mb/s）在12分钟内会发生回绕，FDDI（100 Mb/s）为5分钟，而一个千兆比网络（1000 Mb/s）则为34秒。这时问题不再是带宽时延乘积，而在于带宽本身。

在24.6节，我们将介绍一种对付这种情况的办法：使用 TCP的时间戳选项的 PAWS (Protection Against Wrapped Sequence numbers)算法（保护回绕的序号）。

4.4BSD包含了我们将要在下面介绍的所有选项和算法：窗口扩大选项、时间戳选项和保护回绕的序号。许多供应商也正在开始支持这些选项。

千兆比网络

当网络的速率达到千兆比的时候，情况就会发生变化。 [Partridge 1994]详细介绍了千兆比网络。在这里我们看一下在时延和带宽之间的差别 [Kleinrock 1992]。

考虑通过美国发送一个 100万字节的文件的情况，假定时延为 30 ms。图24-6显示了两种情况：上图显示了使用一个T1电话线（1 544 000 b/s）的情况，而下图则是使用一个1 Gb/s网络的情况。*x*轴显示的是时间，发送方在图的左侧，而接收方则在图的右侧， y轴为网络容量。两幅图中的阴影区域表示发送的100万字节。

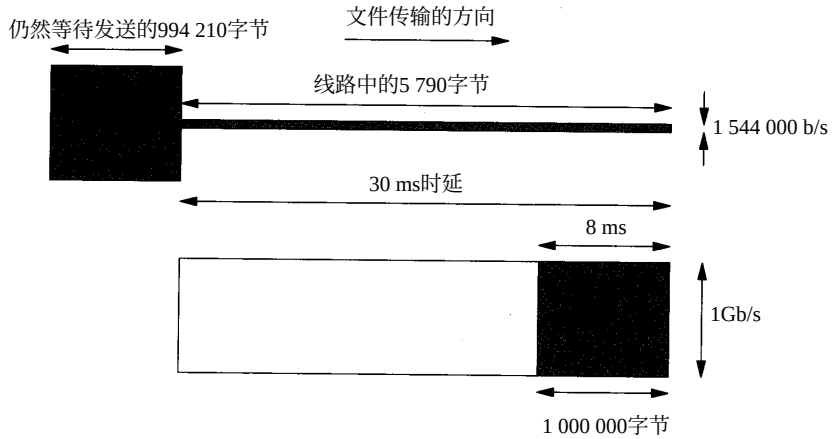


图24-6 以30 ms的延时通过网络发送100万字节的文件

图24-6显示了30 ms后这两个网络的状态。经过 30 ms（延时）以后数据的第1个比特都已到达对端。但对T1网络而言，由于管道容量仅为 5 790字节，因此发送方仍然有 994 210个字节等待发送。而千兆比网络的容量则为3 750 000字节，因此，整个文件仅使用了25%左右的带宽，此时文件的最后一个比特已经到达第1个字节后8 ms处。

经过T1网络传输文件的总时间为5.211秒。如果增加更多的带宽，使用一个T3网络(45 000 000 b/s)，则总时间减少到0.208秒。增加约29倍的带宽可以将总时间减小到约25分之一。

使用千兆比网络传输文件的总时间为0.038秒：30 ms的时延加上8 ms的真正传输文件的时间。假定能够将带宽增加为2000 Mb/s，我们只能够将总时间减小为0.304 ms：同样30 ms的时延和4ms的真正传输时间。现在使带宽加倍仅能够将时间减少约 10%。在千兆比速率下，时延限制占据了主要地位，而带宽不再成为限制。

时延主要是由光速引起的，而且不能够被减小（除非爱因斯坦是错误的）。当我们考虑到分组需要建立和终止一个连接时，这个固定时延起的作用就更糟糕了。千兆比网络会引起一些需要不同看待的连网观点。

24.4 窗口扩大选项

窗口扩大选项使TCP的窗口定义从16 bit增加为32 bit。这并不是通过修改TCP首部来实现的，TCP首部仍然使用16 bit，而是通过定义一个选项实现对16 bit的扩大操作(scaling operation)来完成的。于是TCP在内部将实际的窗口大小维持为32 bit的值。

在图18-20可以看到关于这个选项的例子。一个字节的移位记数器取值为0（没有扩大窗口的操作）和14。这个最大值14表示窗口大小为1 073 725 440字节（65535×2^{14}）。

这个选项只能够出现在一个SYN报文段中，因此当连接建立起来后，在每个方向的扩大因子是固定的。为了使用窗口扩大，两端必须在它们的SYN报文段中发送这个选项。主动建立连接的一方在其SYN中发送这个选项，但是被动建立连接的一方只能够在收到带有这个选项的SYN之后才可以发送这个选项。每个方向上的扩大因子可以不同。

如果主动连接的一方发送一个非零的扩大因子，但是没有从另一端收到一个窗口扩大选项，它就将发送和接收的移位记数器置为0。这就允许较新的系统能够与较旧的、不理解新选项的系统进行互操作。

> Host Requirements RFC要求TCP接受在任何报文段中的一个选项（只有前面定义的一个选项，即最大报文段大小，仅在SYN报文段中出现）。它还进一步要求TCP忽略任何它不理解的选项。这就使事情变得容易，因为所有新的选项都有一个长度字段（图8-20）。

假定我们正在使用窗口扩大选项，发送移位记数为S，而接收移位记数则为R。于是我们从另一端收到的每一个16 bit的通告窗口将被左移R位以获得实际的通告窗口大小。每次当我们向对方发送一个窗口通告的时候，我们将实际的32 bit窗口大小右移S比特，然后用它来替换TCP首部中的16 bit的值。

TCP根据接收缓存的大小自动选择移位计数。这个大小是由系统设置的，但是通常向应用进程提供了修改途径（我们在20.4节中讨论了这个缓存）。

一个例子

如果在4.4BSD的主机`vangogh.cs.berkeley.edu`上使用`sock`程序来初始化一个连接，我们可以观察到它的TCP计算窗口扩大因子的情况。下面的交互输出显示的是两个连续运行的程序，第1个指定接收缓存为128 000字节，而第2个的缓存则为220 000字节。

```
vangogh % sock -v -R128000 bsdi.tuc.noao.edu echo
SO_RCVBUF = 128000
connected on 128.32.130.2.4107 to 140.252.13.35.7
TCP_MAXSEG = 512
hello, world                          我们键入这一行
hello, world                          此处是它的回显
^D                                    键入文件结束字符以终止

vangogh % sock -v -R220000 bsdi.tuc.noao.edu echo
SO_RCVBUF = 220000
connected on 128.32.130.2.4108 to 140.252.13.35.7
TCP_MAXSEG = 512
bye, bye                              我们键入这一行
bye, bye                              此处是它的回显
^D                                    键入文件结束字符以终止
```

图24-7显示了这两个连接的`tcpdump`输出结果（去掉了第2个连接的最后8行，因为没有

什么新内容)。

```
1   0.0                 vangogh.4107 > bsdi.echo: S 462402561:462402561(0)
                          win 65535
                          <mss 512,nop,wscale 1,nop,nop,timestamp 995351 0>
2   0.003078 ( 0.0031)  bsdi.echo > vangogh.4107: S 177032705:177032705(0)
                          ack 462402562 win 4096 <mss 512>
3   0.300255 ( 0.2972)  vangogh.4107 > bsdi.echo: . ack 1 win 65535

4  16.920087 (16.6198)  vangogh.4107 > bsdi.echo: P 1:14(13) ack 1 win 65535
5  16.923063 ( 0.0030)  bsdi.echo > vangogh.4107: P 1:14(13) ack 14 win 4096
6  17.220114 ( 0.2971)  vangogh.4107 > bsdi.echo: . ack 14 win 65535

7  26.640335 ( 9.4202)  vangogh.4107 > bsdi.echo: F 14:14(0) ack 14 win 65535
8  26.642688 ( 0.0024)  bsdi.echo > vangogh.4107: . ack 15 win 4096
9  26.643964 ( 0.0013)  bsdi.echo > vangogh.4107: F 14:14(0) ack 15 win 4096
10 26.880274 ( 0.2363)  vangogh.4107 > bsdi.echo: . ack 15 win 65535

11 44.400239 (17.5200)  vangogh.4108 > bsdi.echo: S 468226561:468226561(0)
                          win 65535
                          <mss 512,nop,wscale 2,nop,nop,timestamp 995440 0>
12 44.403358 ( 0.0031)  bsdi.echo > vangogh.4108: S 182792705:182792705(0)
                          ack 468226562 win 4096 <mss 512>
13 44.700027 ( 0.2967)  vangogh.4108 > bsdi.echo: . ack 1 win 65535
```

该连接的其余部分被删除

图24-7 窗口扩大选项的例子

在第1行，vangogh通告一个65535的窗口，并通过设置移位计数为 1来指明窗口扩大选项。这个通告的窗口是比接收窗口（128 000）还小的一个最大可能取值，因为在一个 SYN报文段中的窗口字段从不进行扩大运算。

扩大因子为1表示vangogh发送窗口通告一直到131 070（65535×2^1）。这将调节我们的接收缓存的大小（12 8000）。因为bsdi在它的SYN（第2行）中没有发送窗口扩大选项，因此这个选项没有被使用。注意到vangogh在随后的连接阶段继续使用最大可能的窗口（65535）。

对于第2个连接vangogh请求的移位计数为 2，表明它希望发送窗口通告一直为 262 140（65535×2^2），这比我们的接收缓存（220 000）大。

24.5 时间戳选项

时间戳选项使发送方在每个报文段中放置一个时间戳值。接收方在确认中返回这个数值，从而允许发送方为每一个收到的 ACK计算RTT（我们必须说“每一个收到的 ACK”而不是“每一个报文段”，是因为TCP通常用一个ACK来确认多个报文段）。我们提到过目前许多实现为每一个窗口只计算一个 RTT，对于包含8个报文段的窗口而言这是正确的。然而，较大的窗口大小则需要进行更好的RTT计算。

RFC 1323的3.1节给出了需要为较大窗口进行更好的 RTT计算的信号处理的理由。通常RTT通过对一个数据信号（包含数据的报文段）以较低的频率（每个窗口一次）进行采样来进行计算，这就将别名引入了被估计的RTT中。当每个窗口中有8个报文段时，采样速率为数据率的1/8，这还是可以忍受的。但是如果每个窗口中有100个报文段时，采样速率则为数据速率的1/100，这将导致被估计的RTT不精确，从而引起不必要的重传。如果一个报文段被丢失，则会使情况变得更糟。

图18-20显示了时间戳选项的格式。发送方在第 1个字段中放置一个32 bit的值，接收方在应答字段中回显这个数值。包含这个选项的 TCP首部长度将从正常的20字节增加为32字节。

时间戳是一个单调递增的值。由于接收方只需要回显收到的内容，因此不需要关注时间戳单元是什么。这个选项不需要在两个主机之间进行任何形式的时钟同步。 RFC 1323推荐在1毫秒和1秒之间将时间戳的值加1。

4.4BSD在启动时将时间戳始终设置为0，然后每隔500 ms将时间戳时钟加1。

在图24-7中，如果观察在报文段1和报文段11的时间戳，它们之间的差（89个单元）对应于每个单元500 ms的规定，因为实际时间差为44.4秒。

在连接建立阶段，对这个选项的规定与前一节讲的窗口扩大选项类似。主动发起连接的一方在它的SYN中指定选项。只有在它从另一方的 SYN中收到了这个选项之后，该选项才会在以后的报文段中进行设置。

我们已经看到接收方 TCP不需要对每个包含数据的报文段进行确认，许多实现每两个报文段发送一个ACK。如果接收方发送一个确认了两个报文段的 ACK，那么哪一个收到的时间戳应当放入回显应答字段中来发回去呢？

为了减少任一端所维持的状态数量，对于每个连接只保持一个时间戳的数值。选择何时更新这个数值的算法非常简单：

1) TCP跟踪下一个ACK中将要发送的时间戳的值（一个名为 *tsrecent*的变量）以及最后发送的ACK中的确认序号（一个名为*lastack*的变量）。这个序号就是接收方期望的序号。

2) 当一个包含有字节号 *lastack*的报文段到达时，则该报文段中的时间戳被保存在 *tsrecent*中。

3) 无论何时发送一个时间戳选项， *tsrecent*就作为时间戳回显应答字段被发送，而序号字段被保存在*lastack*中。

这个算法能够处理下面两种情况：

1) 如果ACK被接收方迟延，则作为回显值的时间戳值应该对应于最早被确认的报文段。例如，如果两个包含 1~1024和1025~2048字节的报文段到达，每一个都带有一个时间戳选项，接收方产生一个ACK 2049来对它们进行确认。此时， ACK中的时间戳应该是包含字节 1~1024的第1个报文段中的时间戳。这种处理是正确的，因为发送方在进行重传超时时间的计算时，必须将迟延的ACK也考虑在内。

2) 如果一个收到的报文段虽然在窗口范围内但同时又是失序，这就表明前面的报文段已经丢失。当那个丢失的报文段到达时，它的时间戳（而不是失序的报文段的时间戳）将被回显。例如，假定有3个各包含1024字节数据的报文段，按如下顺序接收：包含字节 1~1024的报文段1，包含字节2049~4072的报文段3和包含字节1025~2048的报文段2。返回的ACK应该是带有报文段1的时间戳的ACK 1025（一个正常的所期望的对数据的 ACK）、带有报文段1的时间戳的ACK 1025（一个重复的、响应位于窗口内但却是失序的报文段的 ACK），然后是带有报文段2的时间戳的ACK 3073（不是报文段3中的较后的时间戳）。这与当报文段丢失时的对RTT估计过高具有同样的效果，但这比估计过低要好些。而且，如果最后的 ACK含有来自报文段3的时间戳，它可以包括重复的 ACK返回和报文段2被重传所需要的时间，或者可以包括发送方的报文段2的重传超时定时器到期的时间。无论在哪一种情况下，回显报文段 3的时间戳将引起发送方的RTT计算出现偏差。

尽管时间戳选项能够更好地计算 RTT，它还为发送方提供了一种方法，以避免接收到旧的报文段，并认为它们是现在的数据的一部分。下一节将对此进行描述。

24.6 PAWS：防止回绕的序号

考虑一个使用窗口扩大选项的TCP连接，其最大可能的窗口大小为1千兆字节（2^{30}）（最大的窗口是65535×2^{14}，而不是$2^{16}\times2^{14}$，但只比这个数值小一点点，并不影响这里的讨论）。还假定使用了时间戳选项，并且由发送方指定的时间戳对每个将要发送的窗口加1（这是保守的方法。通常时间戳比这种方式增加得快）。图24-8显示了在传输6千兆字节的数据时，在两个主机之间可能的数据流。为了避免使用许多10位的数字，我们使用G来表示1 073 741 824的倍数。我们还使用了tcpdump的记号，即用*J:K*来表示通过了*J*字节的数据，且包括字节*K*-1。

时间	发送字节	发送序号	发送时间戳	接　　收
A	0G:1G	0G:1G	1	正确
B	1G:2G	1G:2G	2	正确，但有一个段丢失并重发
C	2G:3G	2G:3G	3	正确
D	3G:4G	3G:4G	4	正确
E	4G:5G	0G:1G	5	正确
F	5G:6G	1G:2G	6	正确，但重发的段又出现了

图24-8 在6个1千兆字节的窗口中传输6千兆字节的数据

32 bit的序号在时间D和时间E之间发生了回绕。假定一个报文段在时间B丢失并被重传。还假定这个丢失的报文段在时间E重新出现。

这假定了在报文段丢失和重新出现之间的时间差小于MSL，否则这个报文段在它的TTL到期时会被某个路由器丢弃。正如我们前面提到的，这种情况只有在高速连接上才会发生，此时旧的报文段重新出现，并带有当前要传输的序号。

我们还可以从图24-8中观察到使用时间戳可以避免这种情况。接收方将时间戳视为序列号的一个32 bit的扩展。由于在时间E重新出现的报文段的时间戳为2，这比最近有效的时间戳小（5或6），因此PAWS算法将其丢弃。

PAWS算法不需要在发送方和接收方之间进行任何形式的时间同步。接收方所需要的就是时间戳的值应该单调递增，并且每个窗口至少增加1。

24.7 T/TCP：为事务用的TCP扩展

TCP提供的是一种虚电路方式的运输服务。一个连接的生存时间包括三个不同的阶段：建立、数据传输和终止。这种虚电路服务非常适合诸如远程注册和文件传输之类的应用。

但是，还有出现其他的应用进程被设计成使用事务服务。一个事务(transaction)就是符合下面这些特征的一个客户请求及其随后的服务器响应。

1) 应该避免连接建立和连接终止的开销，在可能的时候，发送一个请求分组并接收一个应答分组。

2) 等待时间应当减少到等于RTT与SPT之和。其中RTT (Round-Trip Time)为往返时间，而SPT (Server Processing Time)则是服务器处理请求的时间。

3) 服务器应当能够检测出重复的请求，并且当收到一个重复的请求时不重新处理事务（避免重新处理意味着服务器不必再次处理请求，而是返回保存的、与该请求对应的应答）。

我们已经看到的一个使用这种类型服务的应用就是域名服务（第14章），尽管DNS与服务器重新处理重复的请求无关。

如今一个应用程序设计人员面对的一种选择是使用TCP还是UDP。TCP提供了过多的事务特征，而UDP提供的则不够。通常应用程序使用UDP来构造（避免TCP连接的开销），而许多需要的特征（如动态超时和重传、拥塞避免等）被放置在应用层，一遍又一遍的重新设计和实现。

一个较好的解决方法是提供一个能够提供足够多的事务处理功能的运输层。我们在本节所介绍的事务协议被称为T/TCP。我们从它的定义，即RFC 1379 [Braden 1992b]和[Braden 1992c]，开始介绍。

大多数的TCP需要使用7个报文段来打开和关闭一个连接（见图18-13）。现在增加三个报文段：一个对应于请求，一个对应于应答和对请求的确认，第三个对应于对应答的确认。如果额外的控制比特被追加到报文段上——也就是，第1个报文段带有SYN、客户请求和一个FIN——客户仍然能够看到一个2倍的RTT与SPT之和的最小开销（与数据一起发送一个SYN和FIN是合法的；当前的TCP是否能够正确处理它们是另外一个问题）。

另一个与TCP有关的问题是TIME_WAIT状态和它需要的2MSL的等待时间。正如在习题18.14中看到的，这使两个主机之间的事务率降低到每秒268个。

TCP为处理事务而需要进行的两个改动是避免三次握手和缩短WAIT_TIME状态。T/TCP通过使用加速打开来避免三次握手：

1) 它为打开的连接指定一个32 bit的连接计数CC (Connection Count)，无论主动打开还是被动打开。一个主机的CC值从一个全局计数器中获得，该计数器每次被使用时加1。
2) 在两个使用T/TCP的主机之间的每一个报文段都包括一个新的TCP选项CC。这个选项的长度为6个字节，包含发送方在该连接上的32 bit的CC值。
3) 一个主机维持一个缓存，该缓存保留每个主机上一次的CC值，这些值从来自这个主机的一个可接受的SYN报文段中获得。
4) 当在一个开始的SYN中收到一个CC选项的时候，接收方比较收到的值与为该发送方缓存的CC值。如果接收到的CC比缓存的大，则该SYN是新的，报文段中的任何数据被传递给接收应用进程（服务器）。这个连接被称为半同步。
   如果接收的CC比缓存的小，或者接收主机上没有对应这个客户的缓存CC，则执行正常的TCP三次握手过程。
5) 为响应一个开始的SYN，带有SYN和ACK的报文段在另一个被称为CCECHO的选项中回显所接收到的CC值。
6) 在一个非SYN报文段中的CC值检测和拒绝来自同一个连接的前一个替身的任何重复的报文段。

这种“加速打开”避免了使用三次握手的要求，除非客户或者服务器已经崩溃并重新启动。这样做的代价是服务器必须记住从每个客户接收的最近的CC值。

基于在两个主机之间测量RTT来动态计算TIME_WAIT的延时，可以缩短TIME_WAIT状态。TIME_WAIT时延被设置为8倍的重传超时值*RTO*（见21.3节）。

通过使用这些特征，最小的事务序列是交换三个报文段：

1) 由一个主动打开引起的客户到服务器：客户的SYN、客户的数据（请求）、客户的FIN以及客户的CC。当被动的服务器TCP接收到这个报文段的时候，如果客户的CC比为这个客户缓存的CC要大，则客户的数据被传送给服务器应用程序进行处理。
2) 服务器到客户：服务器的SYN、服务器的数据（应答）、服务器的FIN、对客户的FIN

的ACK、服务器的CC以及客户的CC的CCECHO。由于TCP的确认是累积的，这个对客户的FIN的ACK也对客户的SYN、数据及FIN进行了确认。

当客户TCP接收到这个报文段，就将其传送给客户应用进程。

3) 客户到服务器：对服务器的FIN的ACK，它也确认了服务器的SYN、数据和FIN。

客户对它的请求的响应时间为RTT与SPT的和。

在参考资料中有许多关于实现这个TCP选项的很好的地方。我们在这里将它们归纳如下：

- 服务器的SYN和ACK（第2个报文段）必须被迟延，从而允许应答与它一起捎带发送（通常对SYN的ACK是不迟延的）。但它也不能迟延得太多，否则客户将超时并引起重传。
- 请求可以需要多个报文段，但是服务器必须对它们可能失序达到的情况进行处理（通常当数据在SYN之前到达时，该数据被丢弃并产生一个复位。通过使用 T/TCP，这些失序的数据将放入队列中处理）。
- API必须使服务器进程用一个单一的操作来发送数据和关闭连接，从而允许第二个报文段中的FIN与应答一起捎带发送（通常应用进程先写应答，从而引起发送一个数据报文段，然后关闭连接，引起发送FIN）。
- 在收到来自服务器的MSS通告之前，客户在第1个报文段中正在发送数据。为避免限制客户的MSS为536，一个给定主机的MSS应该与它的CC值一起缓存。
- 客户在没有接收到来自服务器的窗口通告之前也可以向服务器发送数据。 T/TCP建议默认的窗口为4096，并且也为服务器缓存拥塞门限。
- 使用最小3个报文段交换，在每个方向上只能计算一个 RTT。加上包括了服务器处理时间的客户测量RTT。这意味着被平滑的RTT及其方差的值也必须为服务器缓存起来，这与我们在21.9节描述的类似。

T/TCP的特征中吸引人的地方在于它对现有协议进行了最小的修改，同时又兼容了现有的实现。它还利用了TCP中现有的工程特征（动态超时和重传、拥塞避免等），而不是迫使应用进程来处理这些问题。

一个可作为替换的事务协议是通用报文事务协议 VMTP（Versatile Message Transaction Protocol），该协议在RFC 1045 [Cheriton 1988]中进行了描述。与T/TCP是现有协议的一个小的扩充不同，VMTP是使用IP的一个完整的运输层。VMTP处理差错检测、重传和重复压缩。它还支持多播通信。

24.8 TCP的性能

在80年代中期出版的数值显示出 TCP在一个以太网上的吞吐量在每秒 100 000~200 000字节之间（[Stevens 1990]的17.5节给出了参考文献）。从那时起事情已经发生了许多改变。现在通常使用的硬件（工作站和更快的个人电脑）每秒可以传输 800 000字节或者更快。

在10 Mb/s的以太网上计算我们能够观察到的理论上的TCP最大吞吐量是一件值得做的练习[Warnock

字 段	数据(字节)	ACK(字节)
以太网前导	8	8
以太网目的地址	6	6
以太网源地址	6	6
以太类型字段	2	2
IP首部	20	20
TCP首部	20	20
用户数据	1460	0
填充字符	0	6
以太网CRC检验	4	4
分组间隙(9.6ms)	12	12
总计	1538	84

图24-9 计算以太网理论上最大吞吐量的字段大小

1991]。我们可以在图24-9中看到这个计算的基础。这个图显示了满长度的数据报文段和一个ACK交换的全部的字节。

我们必须计及所有的开销：前同步码、加到确认上的填充字节、循环冗余检验 CRC以及分组之间的最小间隔（9.6ms，相当在10 Mb/s速率下的12个字节）。

首先假定发送方传输两个背对背、满长度的数据报文段，然后接收方为这两个报文段发送一个ACK。于是最大的吞吐量（用户数据）为：

$$throughput=\frac{2\times 1460\ B}{22\times 1538\ B+84\ B}\times\frac{10\ 000\ 000\ b/s}{8\ b/B}=1\ 555\ 063\ B/S$$

如果TCP窗口开到它的最大值（65535，不使用窗口扩大选项），这就允许一个窗口容纳44个1460字节的报文段。如果接收方每个报文段发送一个ACK，则计算变为：

$$throughput=\frac{22\times 1460\ B}{22\times 1538\ B+84\ B}\times\frac{10\ 000\ 000\ b/s}{8\ b/B}=1\ 183\ 667\ B/S$$

这就是理论上的限制，并做出某些假定：接收方发送的一个 ACK没有和发送方的报文段之一在以太网上发生冲突；发送方可按以太网的最小间隔时间来发送两个报文段；接收方可以在最小的以太网间隔时间内产生一个 ACK。不论在这些数字上多么乐观，[Warnock 1991]在一个以太网上使用标准的多用户工作站（即使是快的工作站）测量到了一个连续的 1 075 000字节/秒的速率，这个值在理论值的90%之内。

当移到更快的网络上时，如FDDI（100 Mb/s），[Schryver 1993]指出三个商业厂家已经演示了在FDDI上的TCP在80 Mb/s~90 Mb/s之间。即使在有更多带宽的环境下，[Borman 1992]报告说两个Gray Y-MP计算机在一个800 Mb/s的HIPPI通道上最大值为781 Mb/s，而运行在一个Gray Y-MP上的使用环回接口的两个进程间的速率为 907 Mb/s。

下面这些实际限制适用于任何的实际情况 [Borman 1991]。

1) 不能比最慢的链路运行得更快。

2) 不能比最慢的机器的内存运行得更快。这假定实现是只使用一遍数据。如果不是这样（也就是说，实现使用一遍数据是将它从用户空间复制到内核中，而使用另一遍数据是计算TCP的检验和），那么将运行得更慢。[Dalton et al. 1993]描述了将数据复制数目减少从而使一个标准伯克利源程序的性能得到改进。[Partridge and Pink 1993]将类似的“复制与检验和”的改变与其他性能改进措施一道应用于UDP，从而将UDP的性能提高了约30%。

3) 不能够比由接收方提供的窗口大小除以往返时间所得结果运行得更快（这就是带宽时延乘积公式，使用窗口大小作为带宽时延乘积，并解出带宽）。如果使用24.4节的最大窗口扩大因子14，则窗口大小为1.073 千兆字节，所以这除以RTT的结果就是带宽的极限。

所有这些数字的重要意义就是TCP的最高运行速率的真正上限是由TCP的窗口大小和光速决定的。正如[Partridge and Pink 1993]中计算的那样，许多协议性能问题在于实现中的缺陷而不是协议所固有的一些限制。

24.9 小结

本章已经讨论了五个新的TCP特征：路径MTU发现、窗口扩大选项、时间戳选项、序号回绕保护以及使用改进的 TCP事务处理。我们观察到中间的三个特征是为在长肥管道——具有大的带宽时延乘积的网络——上优化性能所需要的。

路径MTU发现在MTU较大时，对于非本地连接，允许 TCP使用比默认的536大的窗口。这样可以提高性能。

窗口扩大选项使最大的 TCP窗口从65535增加到1千兆字节以上。时间戳选项允许多个报文段被精确计时，并允许接收方提供序号回绕保护（ PAWS)。这对于高速连接是必须的。这些新的TCP选项在连接时进行协商，并被不理解它们的旧系统忽略，从而允许较新的系统与旧的系统进行交互。

为事务用的TCP扩展，即T/TCP，允许一个客户/服务器的请求-应答序列在通常的情况下只使用三个报文段来完成。它避免使用三次握手，并缩短了 TIME_WAIT状态，其方法是为每个主机高速缓存少量的信息，这些信息曾用来建立过一个连接。它还在包含数据报文段中使用SYN和FIN标志。

由于还有许多关于TCP能够运行多快的不精确的传闻，因此我们以对 TCP性能的分析来结束本章。对于一个使用本章介绍的较新特征、协调得非常好的实现而言， TCP的性能仅受最大的1千兆字节窗口和光速（也就是往返时间）的限制。

习题

24.1 当一个系统发送一个开始的SYN报文段，其窗口扩大因子为0，这是什么含义？

24.2 如果在图24-7中的主机bsdi支持窗口扩大选项，则来自 vangogh的报文段3的16 bit窗口大小字段中的期望值是多少？类似地，如果在该图的第 2个连接中也使用这个选项，那么报文段13中的窗口通告应该是多少？

24.3 与在建立连接时的固定窗口扩大因子不同，已经定义过的窗口扩大因子能否在扩大因子变化时也出现呢？

24.4 假定MSL为2分钟，那么在什么速率下序号回绕会成为一个问题呢？

24.5 PAWS被定义为只在一个单独的连接中进行。为了使 TCP将PAWS来替换2MSL等待(即TIME_WAIT状态)，需要进行什么改动？

24.6 在24.4节最后的例子中，为什么 sock程序在紧接着（具有IP地址和端口）后面的一行之前，将接收缓存的大小来输出呢？

24.7 假定MSS为1024，重新计算24.8节中的吞吐量。

24.8 时间戳选项是如何影响Karn算法（见21.3节）的？

24.9 如果主动建立连接的TCP发送带有SYN标志的报文段（没有使用我们在 24.7节介绍的扩展），那么接收TCP应该怎样处理这些数据呢？

24.10 在24.7节我们提到如果没有使用 T/TCP扩展，即使主动开启方发送带有 FIN的数据，客户在接收服务器的响应的时延仍然是两倍的 RTT再加上SPT。给出符合这种情况的报文段。

24.11 假定支持T/TCP，且源自伯克利系统的最小*RTO*为0.5秒，重做习题18.14。

24.12 如果我们实现了 T/TCP，并测量两个主机之间的事务时间，那么可以通过比较什么指标来确定它的有效性？

第25章 SNMP：简单网络管理协议

25.1 引言

随着网络技术的飞速发展，网络的数量也越来越多。而网络中的设备来自各个不同的厂家，如何管理这些设备就变得十分重要。本章的内容就是介绍管理这些设备的标准。

基于TCP/IP的网络管理包含两个部分：网络管理站（也叫管理进程， manager）和被管的网络单元（也叫被管设备）。被管设备种类繁多，例如：路由器、 X 终端、终端服务器和打印机等。这些被管设备的共同点就是都运行 TCP/IP协议。被管设备端和管理相关的软件叫做代理程序(agent)或代理进程。管理站一般都是带有彩色监视器的工作站，可以显示所有被管设备的状态(例如连接是否掉线、各种连接上的流量状况等)。

管理进程和代理进程之间的通信可以有两种方式。一种是管理进程向代理进程发出请求，询问一个具体的参数值（例如：你产生了多少个不可达的 ICMP端口？）。另外一种方式是代理进程主动向管理进程报告有某些重要的事件发生（例如：一个连接口掉线了）。当然，管理进程除了可以向代理进程询问某些参数值以外，它还可以按要求改变代理进程的参数值（例如：把默认的IP TTL值改为64）。

基于TCP/IP的网络管理包含3个组成部分：

1) 一个管理信息库MIB（Management Information Base）。管理信息库包含所有代理进程的所有可被查询和修改的参数。 RFC 1213 [McCloghrie and Rose 1991]定义了第二版的MIB，叫做MIB-II。

2) 关于MIB的一套公用的结构和表示符号。叫做管理信息结构 SMI（Structure of Management Information）。这个在RFC 1155 [Rose and McCloghrie 1990] 中定义。例如：SMI定义计数器是一个非负整数，它的计数范围是 0~4 294 967 295，当达到最大值时，又从0开始计数。

3) 管理进程和代理进程之间的通信协议，叫做简单网络管理协议 SNMP（Simple Network Management Protocol）。在RFC 1157 [Case et al. 1990]中定义。SNMP包括数据报交换的格式等。尽管可以在运输层采用各种各样的协议，但是在 SNMP中，用得最多的协议还是UDP。

上面提到的RFC所定义的SNMP叫做SNMP v1，或者就叫做SNMP，这也是本章的主要内容。到1993年为止，又有一些新的关于SNMP的 RFC发表。在这些RFC中定义的SNMP叫做第二版SNMP（SNMP v2），这将在25.12章节中讨论。

本章首先介绍管理进程和代理进程之间的协议，然后讨论参数的数据类型。在本章中将用到前面已经出现过的名词，如： IP、UDP和TCP等。我们在叙述中将举一些例子来帮助读者理解，这些例子和前面的某些章节相关。

25.2 协议

关于管理进程和代理进程之间的交互信息， SNMP定义了5种报文：

1) get-request操作：从代理进程处提取一个或多个参数值。

2) get-next-request操作：从代理进程处提取一个或多个参数的下一个参数值（关于“下一个（next）”的含义将在后面的章节中介绍）。

3) set-request操作：设置代理进程的一个或多个参数值。

4) get-response操作：返回的一个或多个参数值。这个操作是由代理进程发出的。它是前面3中操作的响应操作。

5) trap 操作：代理进程主动发出的报文，通知管理进程有某些事情发生。

前面的3个操作是由管理进程向代理进程发出的。后面两个是代理进程发给管理进程的（为简化起见，前面3个操作今后叫做get、get-next和set操作）。图25-1描述了这5种操作。

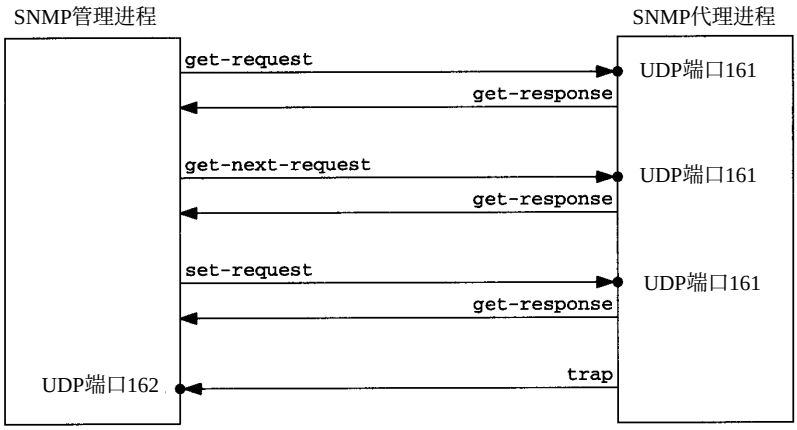


图25-1 SNMP的5种操作

既然这些操作中的前 4种操作是简单的请求 -应答方式（也就是管理进程发出请求，代理进程应答响应），而且在SNMP中往往使用UDP协议，所以可能发生管理进程和代理进程之间数据报丢失的情况。因此一定要有超时和重传机制。

管理进程发出的前面 3种操作采用UDP的161端口。代理进程发出的 Trap操作采用UDP的162端口。由于收发采用了不同的端口号，所以一个系统可以同时为管理进程和代理进程（参见习题25.1）。

图25-2是封装成UDP数据报的5种操作的SNMP报文格式。

在图中，我们仅仅对IP和UDP的首部长度进行了标注。这是由于：SNMP报文的编码采用了ASN.1和BER，这就使得报文的长度取决于变量的类型和值。关于 ASN.1和BER的内容将在后面介绍。在这里介绍各个字段的内容和作用。

版本字段是0。该字段的值是通过SNMP版本号减去1得到的。显然0代表SNMP v1。

图25-3显示各种PDU对应的值（PDU即协议数据单元，也就是分组）。

共同体字段是一个字符串。这是管理进程和代理进程之间的口令，是明文格式。默认的值是public。

对于get、get-next和set操作，请求标识由管理进程设置，然后由代理进程在 get-response中返回。这种类型的字段我们在其他UDP应用中曾经见过（回忆一下在图14-3中DNS的标识字段，或者是图16-2中的事务标识字段）。这个字段的作用是使客户进程（在目前情况下是管理进程）能够将服务器进程（即代理进程）发出的响应和客户进程发出的查询进行匹配。这个

字段允许管理进程对一个或多个代理进程发出多个请求，并且从返回的众多 应答中进行分类。

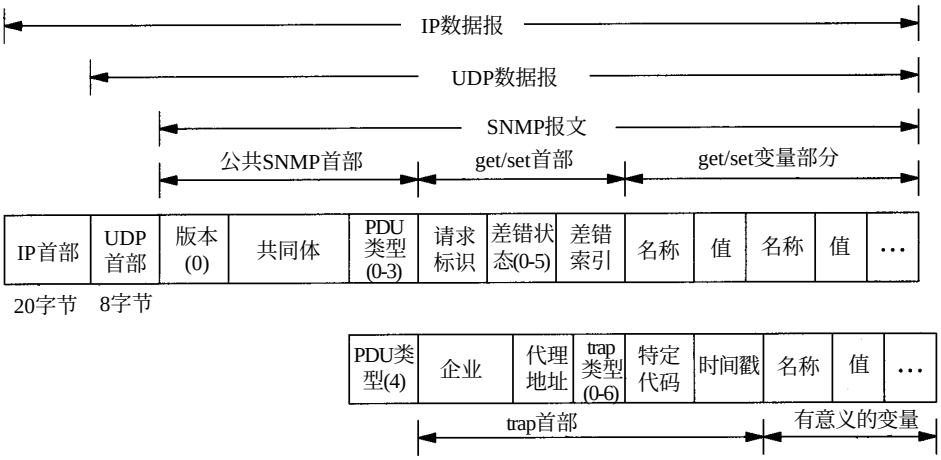


图25-2 SNMP报文的格式

差错状态字段是一个整数，它是由代理进程标注的，指明有差错发生。图25-4是参数值、名称和描述之间的对应关系。

差错索引字段是一个整数偏移量，指明当有差错发生时，差错发生在哪个参数。它是由代理进程标注的，并且只有在发生noSuchName、readOnly和badValue差错时才进行标注。

PDU类型	名　　称
0	get-request
1	get-next-request
2	get-response
3	set-request
4	trap

图25-3 SNMP报文中的PDU类型

差错状态	名　称	描　　述
0	noError	没有错误
1	tooBig	代理进程无法把响应放在一个SNMP消息中发送
2	noSuchName	操作一个不存在的变量
3	badValue	set操作的值或语义有错误
4	readOnly	管理进程试图修改一个只读变量
5	genErr	其他错误

图25-4 SNMP差错状态的值

在get、get-next和set的请求数据报中，包含变量名称和变量值的一张表。对于 get和get-next操作，变量值部分被忽略，也就是不需要填写。

对于trap操作符（PDU类型是4），SNMP报文格式有所变化。我们将在 25.10节中当讨论到trap时再详细讨论。

25.3 管理信息结构

SNMP中，数据类型并不多。在本节，我们就讨论这些数据类型，而不关心这些数据类型在实际中是如何编码的。

- INTEGER。一个变量虽然定义为整型，但也有多种形式。有些整型变量没有范围限制，有些整型变量定义为特定的数值（例如, IP的转发标志就只有允许转发时的1或者不允许转发时的2这两种），有些整型变量定义为一个特定的范围（例如, UDP和TCP的端口号就从0到65535）。
- OCTER STRING 0或多个8 bit字节，每个字节值在0~255之间。对于这种数据类型和

下一种数据类型的BER编码，字符串的字节个数要超过字符串本身的长度。这些字符串不是以NULL结尾的字符串。

- DisplayString。0或多个8 bit字节，但是每个字节必须是ASCII码（26.4中有ASCII字符集）。在MIB-II中，所有该类型的变量不能超过255个字符（0个字符是可以的）。
- OBJECT IDENTIFIER将在下一节中介绍。
- NULL。代表相关的变量没有值。例如，在get或get-next操作中，变量的值就是NULL，因为这些值还有待到代理进程处去取。
- IpAddress。4字节长度的OCTER STRING，以网络序表示的IP地址。每个字节代表IP地址的一个字段。
- PhysAddress。OCTER STRING类型，代表物理地址（例如以太网物理地址为6个字节长度）。
- Counter。非负的整数，可从0递增到 $2^{32}-1$（4 294 976 295）。达到最大值后归0。
- Gauge。非负的整数，取值范围为从0到4 294 976 295(或增或减)。达到最大值后锁定，直到复位。例如，MIB中的tcpCurrEstab就是这种类型的变量的一个例子，它代表目前在ESTABLISHED或CLOSE_WAIT状态的TCP连接数。
- TimeTicks。时间计数器，以0.01秒为单位递增，但是不同的变量可以有不同的递增幅度。所以在定义这种类型的变量的时候，必须指定递增幅度。例如，MIB中的sysUpTime变量就是这种类型的变量，代表代理进程从启动开始的时间长度，以多少个百分之一秒的数目来表示。
- SEQUENCE。这一数据类型与C程序设计语言中的“structure”类似。一个SEQUENCE包括0个或多个元素，每一个元素又是另一个ASN.1数据类型。例如，MIB中的UdpEntry就是这种类型的变量。它代表在代理进程侧目前“激活”的UDP数量（“激活”表示目前被应用程序所用）。在这个变量中包含两个元素：
  1) IpAddress类型中的udpLocalAddress，表示IP地址。
  2) INTEGER类型中的udpLocalPort，从0到65535，表示端口号。
- SEQUENDE OF。这是一个向量的定义，其所有元素具有相同的类型。如果每一个元素都具有简单的数据类型，例如是整数类型，那么我们就得到一个简单的向量（一个一维向量）。但是我们将看到，SNMP在使用这个数据类型时，其向量中的每一个元素是一个SEQUENCE（结构）。因而可以将它看成为一个二维数组或表。

例如，名为udpTable的UDP监听表(listener)就是这种类型的变量。它是一个二元的SEQUENCE变量。每个二元组就是一个UdpEntry。如图25-5所示。

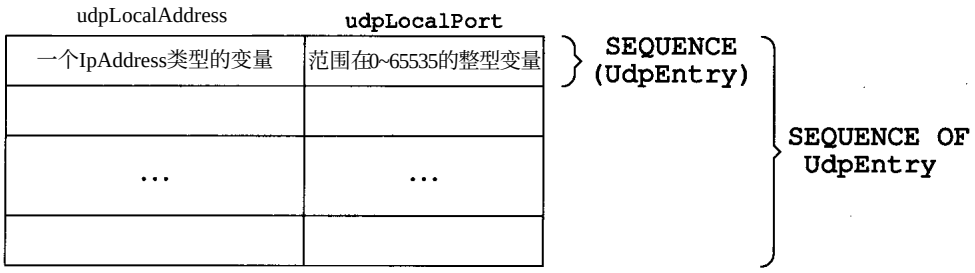


图25-5 表格形式的udpTable 变量

在SNMP中，对于这种类型的表格并没有标注它的列数。但在 25.7节中，我们将看到 get-next操作是如何判断已经操作到最后一列的情况。同时，在 25.6节中，我们还将介绍管理进程如何表示它对某一行数据进行 get或set操作。

25.4 对象标识符

对象标识是一种数据类型，它指明一种“ 授权 ”命名的对象。“ 授权 ”的意思就是这些标识不是随便分配的，它是由一些权威机构进行管理和分配的。

对象标识是一个整数序列，以点（“ . ”）分隔。这些整数构成一个树型结构，类似于 DNS（图14-1）或Unix的文件系统。对象标识从树的顶部开始，顶部没有标识，以 root表示（这和Unix中文件系统的树遍历方向非常类似）。

图25-6显示了在SNMP中用到的这种树型结构。所有的 MIB变量都从1.3.6.1.2.1这个标识开始。

树上的每个结点同时还有一个文字名。例如标识 1.3.6.1.2.1就和`iso.org.dod.internet.memt.mib`对应。这主要是为了人们阅读方便。在实际应用中，也就是说在管理进程和代理进程进行数据报交互时，MIB变量名是以对象标识来标识的，当然都是以1.3.6.1.2.1开头的。

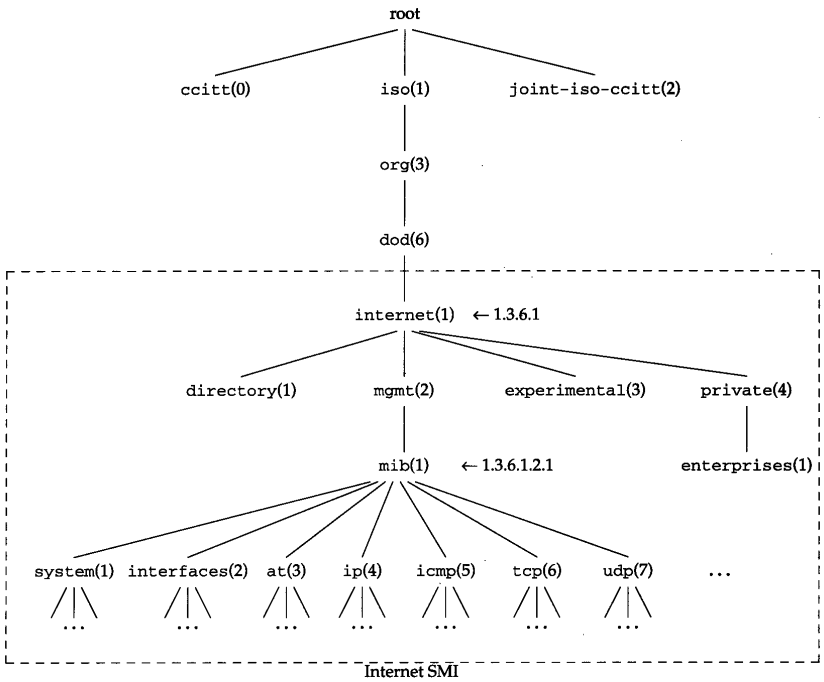


图25-6 管理信息库中的对象标识

在图25-6中，我们除了给出了`mib`对象标识外，还给出了`iso.org.dod.internet.private.enterprises`（1.3.6.1.4.1）这个标识。这是给厂家自定义而预留的。在 Assigned Number RFC中列出了在该结点下大约400个标识。

25.5 管理信息库介绍

所谓管理信息库，或者MIB，就是所有代理进程包含的、并且能够被管理进程进行查询和设

置的信息的集合。我们在前面已经提到了在RFC 1213 [McColghrie 和Rose 1991]中定义的MIB-II。

如图25-6所示，MIB被划分为若干个组，如system、interfaces、at（地址转换）和ip组等。

在本节，我们仅仅讨论UDP组中的变量。这个组比较简单，它包含几个变量和一个表格。在下一节，我们将以UDP组为例，详细讲解什么是实例标识（instance identification），什么是字典式排序(lexicographic ordering)以及和这些概念有关的一些简单例子。在这些例子之后，在25.8节我们继续回到MIB，描述MIB中的其他一些组。

在图25-6中我们画出了udp组在mib的下面。图25-7就显示了UDP组的结构。

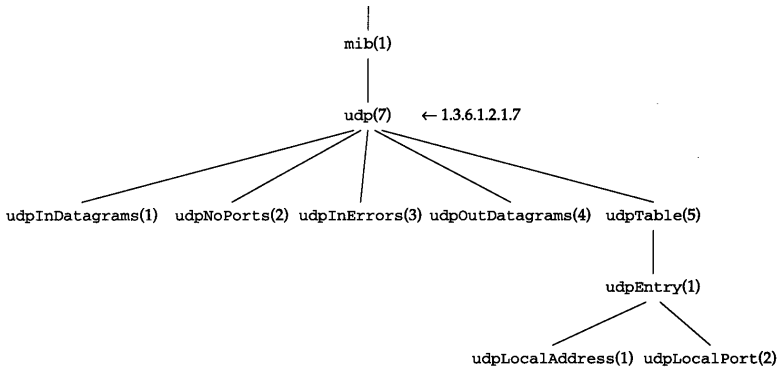


图25-7 UDP组的结构

在该组中，包含4个简单变量和1个由两个简单变量组成的表格。图 25-8描述了这4个简单变量。

名　　称	数据类型	R/W	描　　述
udpInDatagrams	Counter		UDP数据报输入数
udpNoPorts	Counter		没有发送到有效端口的UDP数据报个数
udpInErrors	Counter		接收到的有错误的UDP数据报个数(例如检验错误)
udpOutDatagrams	Counter		UDP数据报输出数

图25-8 UDP组下的简单变量

在本章中，我们就以图 25-8的格式来描述所有的MIB变量。“ R/W ”列如果为空，则代表该变量是只读的；如果变量是可读可写的，则以“ · ”符号来表示。哪怕整个组中的变量都是只读的，我们也将列出“ R/W ”列，以提示读者管理进程只能对这些变量进行查询操作(上图UDP组我们就是这样做的)。同样，如果变量类型是INTEGET类型并且有范围约束，我们也将标明它的下限和上限，就如我们在下图中描述 UDP端口号所做的一样。

图25-9描述了在udpTable中的两个简单变量。

UDP监听表，索引=<udpLocalAddress>.<udpLocalPort>			
名　　称	数据类型	R/W	描　　述
udpLocalAddress	IpAddress		监听进程的本地IP地址。0.0.0.0代表接收任何接口的数据报
udpLocalPort	[0..65535]		监听进程的本地端口号

图25-9 udpTable 中的变量

每次当我们以SNMP表格形式来描述MIB变量时，表格的第1行都表示索引的值，它是表

格中的每一列的参考。在下一节中读者将看到的一些例子也是这样做的。

Case图

在图25-8中，前3个计数器是有相互关系的。Case图真实地描述了一个给出的MIB组中变量之间的相互关系。图25-10就是UDP组的Case图。

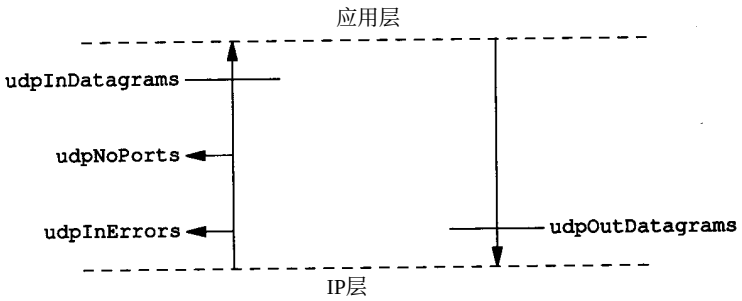


图25-10 UDP组的Case图

这张图表明，发送到应用层的UDP数据报的数量（udpInDatagrams）就是从IP层送到UDP层的UDP数据报的数量，当然udpInError和udpNoPorts也类似。同样，发送到IP层的UDP数据报的数量（udpOutDatagrams）就是从应用层发出的UDP数据报的数量。这表明udpInDatagram不包括udpInError和udpNoPorts。

在深入讲解MIB的时候，这些Case图被用来验证：分组的所有数据路径都是被计数的。[Rose 1994] 中显示了所有MIB组的Case图。

25.6 实例标识

当对MIB变量进行操作，如查询和设置变量的值时，必须对MIB的每个变量进行标识。首先，只有叶子结点是可操作的。SNMP没法处理表格的一整行或一整列。回到图25-7，在图25-8和图25-9中描述过的变量就是叶子结点，而mib、udp、udpTable和udpEntry就不是叶子结点。

25.6.1 简单变量

对于简单变量的处理方法是通过在其对象标识后面添加“ .0 ”来处理的。例如图25-8中的计数器udpInDatagrams，它的对象标识是1.3.6.1.2.1.7.1，它的实例标识是1.3.6.1.2.1.7.1.0，相对应的文字名称是iso.org.dod.internet.mgmt.mib.udp.udpInDatagrams.0。

虽然这个变量处理后通常可以缩写为 udpInDatagrams.0，但我们还是要提醒读者在SNMP报文中（图25-2）该变量的名称是其对象的标识1.3.6.1.2.1.7.1.0。

25.6.2 表格

表格的实例标识就要复杂得多。回顾一下图25-8中的UDP 监听表。

每个MIB中的表格都指明一个以上的索引。对于 UDP监听表来说， MIB定义了包含两个变量的联合索引，这两个变量是：udpLocalAddress，它是一个IP地址；udpLocalPort，它是一个整数（在图25-9中的第1行就显示了这个索引）。

假设在UDP监听表中有3行具体成员：第1行的IP地址是0.0.0.0，端口号是67；第2行的IP

地址是0.0.0.0，端口号是161；第3行的IP地址是0.0.0.0，端口号是520。如图25-11所示。

这意味着系统将从端口 67（BOOTP服务器）、端口161（SNMP）和端口520（RIP）接受来自任何接口的UDP数据报。表格中的这 3行经过处理后的结果在图 25-12中显示。

udpLocalAddress	udpLocalPort
0.0.0.0	67
0.0.0.0	161
0.0.0.0	520

图25-11 UDP监听表

25.6.3 字典式排序

MIB中按照对象标识进行排序时有一个隐含的排序规则。 MIB表格是根据其对象标识按照字典的顺序进行排序的。这就意味着图 25-12中的6个变量排序后的情况如图 25-13所示。从这种字典式排序中可以得出两个重要的结论。

行	对象标识	简　　称	值
1	1.3.6.1.2.1.7.5.1.1.0.0.0.0.67 1.3.6.1.2.1.7.5.1.2.0.0.0.0.67	udpLocalAddress.0.0.0.0.67 udpLocalPort.0.0.0.0.67	0.0.0.0 67
2	1.3.6.1.2.1.7.5.1.1.0.0.0.0.161 1.3.6.1.2.1.7.5.1.2.0.0.0.0.161	udpLocalAddress.0.0.0.0.161 udpLocalPort.0.0.0.0.161	0.0.0.0 161
3	1.3.6.1.2.1.7.5.1.1.0.0.0.0.520 1.3.6.1.2.1.7.5.1.2.0.0.0.0.520	udpLocalAddress.0.0.0.0.520 udpLocalPort.0.0.0.0.520	0.0.0.0 520

图25-12 UDP监听表中行的实例标识

列	对象标识(字典序)	简　　称	值
1	1.3.6.1.2.1.7.5.1.1.0.0.0.0.67 1.3.6.1.2.1.7.5.1.1.0.0.0.0.161 1.3.6.1.2.1.7.5.1.1.0.0.0.0.520	udpLocalAddress.0.0.0.0.67 udpLocalAddress.0.0.0.0.161 udpLocalAddress.0.0.0.0.520	0.0.0.0 0.0.0.0 0.0.0.0
2	1.3.6.1.2.1.7.5.1.2.0.0.0.0.67 1.3.6.1.2.1.7.5.1.2.0.0.0.0.161 1.3.6.1.2.1.7.5.1.2.0.0.0.0.520	udpLocalPort.0.0.0.0.67 udpLocalPort.0.0.0.0.161 udpLocalPort.0.0.0.0.520	67 161 520

图25-13 UDP监听表的字典式排序

1) 在表格中，一个给定变量（在这里指udpLocalAddress）的所有实例都在下个变量（这里指udpLocalPort）的所有实例之前显示。这暗示表格的操作顺序是“先列后行”的次序。这是由于对对象标识进行字典式排序所得到的，而不是按照人们的阅读习惯而排列的。

2) 表格中对行的排序和表格中索引的值有关。在图25-13中，67的字典序小于161，同样161的字典序小于520。

图25-14描述了例子中UDP监听表的这种“先列后行”的次序。

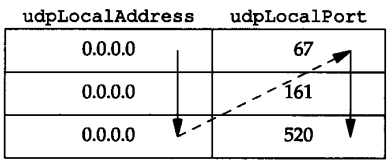


图25-14 按“先列后行”次序显示的UDP监听表

在下节中，讲述到get-next操作时，同样还会遇到这种“先列后行”的次序。

25.7 一些简单的例子

在本节中，我们将介绍如何从SNMP代理进程处获取变量的值。对代理进程进行查询的软件属于ISODE系统，叫做snmpi。两者在[Rose 1994]中有详细的介绍。

25.7.1 简单变量

对一个路由器取两个UDP组的简单变量值：

```
sun % snmpi -a gateway -c secret

snmpi> get udpInDatagrams.0 udpNoPorts.0
udpInDatagrams.0=616168
udpNoPorts.0=33

snmpi> quit
```

其中，-a选项代表要和之通信的代理进程名称，-c选项表示SNMP的共同体名。所谓共同体名，就是客户进程（在这里指snmpi）提供、同时能被服务器进程（这里指代理进程gateway）所识别的一个口令，共同体名称是管理进程请求的权限标志。代理进程允许客户进程用只读共同体名对变量进行读操作，用读写共同体名对变量进行读和写操作。

Snmpi程序的输出提示符是snmpi>，在后面可以键入如get这样的命令，该软件将把它转化为SNMP中的get-request报文。当结束时，键入quit就退出（在后面的例子中，我们将省略掉quit的操作）。

图25-15显示的是对于这个例子tcpdump的两行输出结果。

```
1  0.0                    sun.1024 > gateway.161: GetRequest(42)
                          1.3.6.1.2.1.7.1.0 1.3.6.1.2.1.7.2.0

2  0.348875 (0.3489)      gateway.161 > sun.1024: GetResponse(46)
                          1.3.6.1.2.1.7.1.0=616168
                          1.3.6.1.2.1.7.2.0=33
```

图25-15 简单SNMP查询操作tcpdump 的输出结果

对这两个变量的查询请求是封装在一个UDP数据报中的，而响应也在一个UDP数据报中。

显示的变量是以其对象标识的形式显示的，这是在SNMP报文中实际传输的内容。我们必须指定这两个变量的实例是0。注意，变量的名称（它的对象标识）同样也在响应中返回。在下面我们将看到对于get-next操作这是必需的。

25.7.2 get-next操作

get-next操作是基于MIB的字典式排序的。在下面的例子中，首先向代理进程询问UDP后的下一个对象标识（由于不是一个叶子对象，没有指定任何实例）。代理进程将返回UDP组中的第1个对象，然后我们继续向代理进程取该对象的下一个对象标识，这时候第 2个对象将被返回。重复上面的步骤直到取出所有的对象为止。

```
sun % snmpi -a gateway -c secret

snmpi> next udp
udpInDatagrams.0=616318

snmpi> next udpInDatagrams.0
udpNoPorts.0=33

snmpi> next udpNoPorts.0
udpInErrors.0=0
```

这个例子解释了为什么get-next操作总是返回变量的名称，这是因为我们向代理进程询问下一个变量，代理进程就把变量值和名称一起返回了。

采用这种方式进行get-next操作，我们可以想象管理进程只要做一个简单的循环程序，

就可以从MIB树的顶点开始，对代理进程一步步地进行查询，就可以得出代理进程处所有的变量值和标识。该方式的另外一个用处就是可以对表格进行遍历。

25.7.3 表格的访问

对于“先列后行”次序的UDP监听表，只要采用前面的简单查询程序一步一步地进行操作，就可以遍历整个表格。只要从询问代理进程 udpTable的下一个变量开始就可以了。由于udpTable不是叶子对象，我们不能指定一个实例，但是get-next操作依然能够返回表格中的下一个对象。然后就可以以返回的结果为基础进行下一步的操作，代理进程也会以“先列后行”的次序返回下一个变量，这样就可以遍历整个表格。我们可以看到返回的次序和图25-14相同。

```
sun % snmpi -a gateway -c secret

snmpi> next udpTable
udpLocalAddress.0.0.0.0.67=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.67
udpLocalAddress.0.0.0.0.161=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.161
udpLocalAddress.0.0.0.0.520=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.520
udpLocalPort.0.0.0.0.67=67

snmpi> next udpLocalPort.0.0.0.0.67
udpLocalPort.0.0.0.0.161=161

snmpi> next udpLocalPort.0.0.0.0.161
udpLocalPort.0.0.0.0.520=520

snmpi> next udpLocalPort.0.0.0.0.520
snmpInPkts.0=59
```

我们已完成了对UDP监听表的操作

但是管理进程如何知道已经到达表格的最后一行呢？既然get-next操作返回结果中包含表格中的下一个变量的值和名称，当返回的结果是超出表格之外的下一个变量时，管理进程就可以发现变量的名称发生了较大的变化。这样就可以判断出已经到达表格的最后一行。例如在我们的例子中，当返回的是snmpInPkts变量的时候就代表已经到了UDP监听表的最后一个变量了。

25.8 管理信息库（续）

现在继续讨论MIB。我们仅仅介绍下列MIB组：system（系统标识）、if（接口）、at（地址转换）、ip、icmp和tcp。

25.8.1 system组

system组非常简单，它包含7个简单变量（例如，没有表格）。图25-16列出了system组的名称、数据类型和描述。

可以对netb路由器查询一些简单变量：

```
sun % snmpi -a netb -c secret

snmpi> get sysDescr.0 sysObjectID.0 sysUpTime.0 sysServices.0
sysDescr.0="Epilogue Technology SNMP agent for Telebit NetBlazer"
sysObjectID.0=1.3.6.1.4.1.12.42.3.1
sysUpTime.0=22 days, 11 hours, 23 minutes, 2 seconds (194178200 timeticks)
sysServices.0=0xc<internet,transport>
```

名　称	数据类型	R/W	描　　述
sysDescr	DisplayString		系统的文字描述
sysObjectID	ObjectID		在子树1.3.6.1.4.1中的厂商标识
sysUpTime	TimeTicks		从系统的网管部分启动以来运行的时间（以百分之一秒为计算单位）
sysContact	DisplayString	•	联系人的名字及联系方式
sysName	DisplayString	•	结点的完全合格的域名(FQDN)
sysLocation	DisplayString	•	结点的物理位置
sysServices	[0…127]	•	指示结点提供的服务的值。该值是此结点所支持的OSI模型中层次的和。根据所提供的服务，将下面的一些值相加：0x01(物理层)、0x02（数据链路层）、0x04（互联网层）、0x08（端到端的运输层）和 0x40（应用层）

图25-16　system组中的简单变量

回到图25-6中，system的对象标识符在internet.private.enterprises组（1.3.6.1.4.1）中，在Assigned Numbers RFC文档中可以确定下一个对象标识符（12）肯定是指派给了厂家（Epilogue）。

同时还可以看出，sysServices变量的值是4与8的和，它支持网络层（例如选路）和运输层的应用（例如端到端）。

25.8.2　interface组

名　称	数据类型	R/W	描述
ifNumber	INTEGER		系统上的网络接口数

图25-17　if组中的简单变量

在本组中只定义了一个简单变量，那就是系统的接口数量，如图25-17所示。

在该组中，还有一个表格变量，有 22列。表格中的每行定义了接口的一些特征参数。如图25-18所示。

可以向主机sun查询所有这些接口的变量。如3.8节中所示，我们还是希望访问三个接口，如果SLIP接口已经启动：

```
sun % snmpi -a sun                          首先看第一个接口的接口索引

snmpi> next ifTable
ifIndex.1=1

snmpi> get ifDescr.1 ifType.1 ifMtu.1 ifSpeed.1 ifPhysAddress.1
ifDescr.1="le0"
ifType.1=ethernet-csmacd(6)
ifMtu.1=1500
ifSpeed.1=10000000
ifPhysAddress.1=0x08:00:20:03:f6:42

snmpi> next ifDescr.1 ifType.1 ifMtu.1 ifSpeed.1 ifPhysAddress.1
ifDescr.2="sl0"
ifType.2=propPointToPointSerial(22)
ifMtu.2=552
ifSpeed.2=0
ifPhysAddress.2=0x00:00:00:00:00:00

snmpi> next ifDescr.2 ifType.2 ifMtu.2 ifSpeed.2 ifPhysAddress.2
ifDescr.3="lo0"
ifType.3=softwareLoopback(24)
ifMtu.3=1536
ifSpeed.3=0
ifPhysAddress.3=0x00:00:00:00:00:00
```

接口表，索引＝<*IfIndex*>			
名称	数据类型	R/W	描　述
ifIndex	INTEGER		接口索引，介于1和ifNumber之间
ifDescr	DisplayString		接口的文字描述
ifType	INTEGER		类型，例如：6 = 以太网，7 = 802.3以太网，9 = 802.5令牌环，23 = PPP，28 = SLIP,还有其他一些值
ifMtu	INTEGER		接口的MTU
ifSpeed	Gauge		以b/s为单位的速率
ifPhysAddress	PhysAddress		物理地址，对无物理地址的接口，以一串0表示（例如，串行链路）
ifAdminStatus	[1…3]	•	所希望的接口状态：1= 工作，2 = 不工作，3 = 测试
ifOperStatus	[1…3]	•	当前接口的状态：1= 工作，2 = 不工作，3 = 测试
ifLastChange	TimeTicks		当接口进入目前运行状态时sysUpTime的值
ifInOctets	Counter		收到的字节总数，包括组帧字符
ifInUcastPkts	Counter		交付给高层的单播分组数
ifInNUcastPkts	Counter		交付给高层的非单播（例如，广播或多播）分组数
ifInDiscards	Counter		收到的被丢弃的分组数，即使在分组中无差错（例如，无缓存空间）
ifInErrors	Counter		收到的由于差错被丢弃的分组数
ifInUnknownProtos	Counter		收到的由于未知的协议被丢弃的分组数
ifOutOctets	Counter		发送的字节总数，包括组帧字符
ifOutUcastPkts	Counter		从高层接收到的单播分组数
ifOutNUcastPkts	Counter		从高层接收到的非单播（如广播或多播）分组数
ifOutDiscards	Counter		发出的被丢弃的分组数，即使在分组中无差错（如无缓存空间）
ifOutErrors	Counter		发出的由于差错被丢弃的分组数
ifOutQLen	Gauge		在输出队列中的分组数
ifSpecific	ObjectID		对这种特定媒体类型的MIB定义的引用

图25-18　在接口表中的变量：ifTable

对于第1个接口，采用get操作提取5个变量值，然后用get-next操作提取第2个接口的相同的5个参数。对于第3个接口，同样采用get-next操作。

对于SLIP链路的接口类型，所报告的是一个点到点的专用串行链路，而不是　SLIP链路。此外，SLIP链路的速率没有报告。

这个例子对我们理解get-next操作和“先列后行”次序之间的关系十分重要。如果我们键入命令“next ifDescr.1”，则系统返回的是表格中的下一行所对应的变量，而不是同一行中的下个变量。如果表格是按照“先行后列”次序存放，我们就不能通过一个给定变量来读取下一个变量。

25.8.3　at组

地址转换组对于所有的系统都是必需的，但是在 MIB-II中已经没有这个组。从 MIB-II开

始，每个网络协议组（如IP组）都包含它们各自的网络地址转换表。例如对于IP组，网络地址转换表就是ipNetToMediaTable。

在该组中，仅有一个由3列组成的表格变量。如图25-19所示。

我们将用snmpi程序中的一个新命令来转储(dump)整个表格。向一个叫做kinetics的路由器（该路由器连接了一个TCP/IP网络和一个AppleTalk网络）查询其整个ARP高速缓存。命令的输出是字典式排序的整个表格内容。

地址转换表，索引＝<*atIfIndex*>.1.<*atNetAddress*>			
名　称	数据类型	R/W	描　述
atIfIndex	INTEGER	•	接口数：ifIndex
atPhysAddress	PhysAddress	•	物理地址。若设置为长度为0的字符串，则表示无效表项
atNetAddress	NetworkAddress	•	IP地址

图25-19　网络地址转换表：atTable

```
sun % snmpi -a kinetics -c secret dump at

atIfIndex.1.1.140.252.1.4=1
atIfIndex.1.1.140.252.1.22=1
atIfIndex.1.1.140.252.1.183=1
atIfIndex.2.1.140.252.6.4=2
atIfIndex.2.1.140.252.6.6=2

atPhysAddress.1.1.140.252.1.4=0xaa:00:04:00:f4:14
atPhysAddress.1.1.140.252.1.22=0x08:00:20:0f:2d:38
atPhysAddress.1.1.140.252.1.183=0x00:80:ad:03:6a:80
atPhysAddress.2.1.140.252.6.4=0x00:02:16:48
atPhysAddress.2.1.140.252.6.6=0x00:02:3c:48

atNetAddress.1.1.140.252.1.4=140.252.1.4
atNetAddress.1.1.140.252.1.22=140.252.1.22
atNetAddress.1.1.140.252.1.183=140.252.1.183
atNetAddress.2.1.140.252.6.4=140.252.6.4
atNetAddress.2.1.140.252.6.6=140.252.6.6
```

让我们来分析一下用tcpdump命令时的分组交互情况。当snmpi要转储整个表格时，首先发出一条get-next命令以取得表格的名称（在本例中是at），该名称就是要获取的第一个表项。然后在屏幕上显示的同时生成另一条get-next命令。直到遍历完整个表格的内容后才终止。

atIfIndex	atPhysAddress	atNetAddress
1	0xaa:00:04:00:f4:14	140.252.1.4
1	0x08:00:20:0f:2d:38	140.252.1.22
1	0x00:80:ad:03:6a:80	140.252.1.183
2	0x00:02:16:48	140.252.6.4
2	0x00:02:3c:48	140.252.6.6

图25-20　at表举例（ARP高速缓存）

图25-20显示了在路由器中实际表格的内容。

注意图中，接口2的AppleTalk协议的物理地址是32 bit的数值，而不是我们所熟悉的以太网的48 bit物理地址。同时请注意，正如我们所希望的那样，在图中有一条记录和netb路由器（其IP地址是140.252.1.183）有关。这是因为netb路由器和kinetics路由器在同一个以太网中（140.252.1），而且kinetics路由器必需采用ARP来回送SNMP响应。

25.8.4　ip组

ip组定义了很多简单变量和3个表格变量。图25-21显示了所有的简单变量。

名称	数据类型	R/W	描 述
ipForwarding	[1…2]	•	1代表系统正在转发IP数据报，2则代表不在转发
ipDefaultTTL	INTEGER	•	当运输层不提供TTL值时的默认TTL值
ipInReceives	Counter		从所有接口收到的IP数据报的总数
ipInHdrErrors	Counter		由于首部差错被丢弃的数据报数（例如，检验和差错，版本不匹配，TTL超过等）
ipInAddrErrors	Counter		由于不正确的目的地址被丢弃的IP数据报数
ipForwDatagrams	Counter		曾进行过一次转发尝试的IP数据报数
ipInUnknownProtos	Counter		具有无效协议字段的发往本地的IP数据报数
ipInDiscards	Counter		由于缓存空间不足被丢弃的收到的数据报数
ipInDelivers	Counter		交付到适当的协议模块的IP数据报数
ipOutRequests	Counter		传递给IP层来传输的IP数据报总数。不包括已经在ipForwDatagrams中计入的那些
ipOutDiscards	Counter		由于缓存空间不足被丢弃的输出数据报数
ipOutNoRoutes	Counter		由于找不到路由被丢弃的数据报数
ipReasmTimeout	INTEGER		在等待重装时已收到的数据报片被保留的最大秒数
ipReasmReqds	Counter		收到的需要进行重装的IP数据报片的数目
ipReasmOKs	Counter		已成功重装的IP数据报数
ipReasmFails	Counter		IP重装算法失败次数
ipFragOKs	Counter		被成功分片的IP数据报数
ipFragFails	Counter		需要进行分片但由于设置了“不分片”标志而不能分片的IP数据报数
ipFragCreates	Counter		由分片而产生的IP数据报片的数目
ipRoutingDiscards	Counter		所选择的选路表项即使是有效的但也要丢弃的数目

图25-21 ip组中的简单变量

ip组中的第一个表格变量是IP地址表。系统的每个IP地址都对应该表格中的一行。每行中包含了5个变量，如图25-22所示。

IP地址表，索引＝<*ipAdEntAddr*>			
名 称	数据类型	R/W	描 述
ipAdEntAddr	IpAddress		这一行的IP地址
ipAdEntIfIndex	INTEGER		对应的接口数：ifIndex
ipAdEntNetMask	IpAddress		对这个IP地址的子网掩码
ipAdEntBcastAddr	[0…1]		IP广播地址中的最低位的值。通常为1
ipAdEntReasmMaxSize	[0…65535]		在这个接口上收到的、能够进行重装的、最长的IP数据报

图25-22 IP地址表：ipAddrTable

同样可以向主机sun查询整个IP地址表：

```
sun % snmpi -a sun dump ipAddrTable

ipAdEntAddr.127.0.0.1=127.0.0.1
ipAdEntAddr.140.252.1.29=140.252.1.29
ipAdEntAddr.140.252.13.33=140.252.13.33
```

```
ipAdEntIfIndex.127.0.0.1=3              环回接口
ipAdEntIfIndex.140.252.1.29=2           SLIP接口
ipAdEntIfIndex.140.252.13.33=1          以太网接口

ipAdEntNetMask.127.0.0.1=255.0.0.0
ipAdEntNetMask.140.252.1.29=255.255.255.0
ipAdEntNetMask.140.252.13.33=255.255.255.224

ipAdEntBcastAddr.127.0.0.1=1            所有这三个都使用一个比特进行广播
ipAdEntBcastAddr.140.252.1.29=1
ipAdEntBcastAddr.140.252.13.33=1

ipAdEntReasmMaxSize.127.0.0.1=65535
ipAdEntReasmMaxSize.140.252.1.29=65535
ipAdEntReasmMaxSize.140.252.13.33=65535
```

输出的接口号码可以和图25-18中的输出进行比较，同样IP地址和子网掩码可以和3.8节中采用ifconfig命令时的输出进行比较。

ip组中的第二个表是IP路由表（请回忆一下我们在9.2节中讲到的路由表），如图25-23所示。访问该表中每行记录的索引是目的IP地址。

名　称	数据类型	R/W	描　述
ipRouteDest	IpAddress	•	目的IP地址。值0.0.0.0表示一个默认的表项
ipRouteIfIndex	INTEGER	•	接口数：ifIndex
ipRouteMetric1	INTEGER	•	主要的选路度量。这个度量的意义取决于选路协议（ipRouteProto）。-1表示未使用
ipRouteMetric2	INTEGER	•	可选的选路度量
ipRouteMetric3	INTEGER	•	可选的选路度量
ipRouteMetric4	INTEGER	•	可选的选路度量
ipRouteNextHop	IpAddress	•	下一跳路由器的IP地址
ipRouteType	INTEGER	•	路由类型：1 = 其他，2 = 无效路由，3 = 直接，4= 间接
ipRouteProto	INTEGER		选路协议：1 = 其他，4 = ICMP重定向，8 = RIP, 13 = OSPF, 14 = BGP, 以及其他
ipRouteAge	INTEGER	•	自从路由上次被更新或确定是正确的以后所经历的秒数
ipRouteMask	IpAddress	•	在和ipRouteDest相比较之前，掩码要与目的IP地址进行逻辑“与”
ipRouteMetric5	INTEGER	•	其他的选路度量
ipRouteInfo	ObjectID	•	对这种特定选路协议的MIB定义的引用

图25-23　IP路由表：ipRouteTable

图25-24显示的是用snmpi程序采用dump ipRouteTable命令从主机sun得到的路由表。在这张表中，已经删除了所有5个路由度量，那是由于这5条记录的度量都是-1。在列的标题中，对每个变量名称已经删除了ipRoute这样的前缀。

Dest	IfIndex	NextHop	Type	Proto	Mask
0.0.0.0	2	140.252.1.183	间接(4)	其他(1)	0.0.0.0
127.0.0.1	3	127.0.0.1	直接(3)	其他(1)	255.255.255.255
140.252.1.183	2	140.252.1.29	直接(3)	其他(1)	255.255.255.255
140.252.13.32	1	140.252.13.33	直接(3)	其他(1)	255.255.0.0
140.252.13.65	1	140.252.13.35	间接(4)	其他(1)	255.255.255.255

图25-24　路由器sun上的IP路由表

为比较起见，下面的内容是我们用netstat命令（在9.2节中曾经讨论过）格式显示的路由表信息。图25-24是按字典序显示的，这和netstat命令显示格式不同：

```
sun % netstat -rn
Routing tables
Destination          Gateway              Flags    Refcnt Use        Interface
140.252.13.65        140.252.13.35        UGH      0      115        le0
127.0.0.1            127.0.0.1            UH       1      1107       lo0
140.252.1.183        140.252.1.29         UH       0      86         sl0
default              140.252.1.183        UG       2      1628       sl0
140.252.13.32        140.252.13.33        U        8      68359      le0
```

ip组的最后一个表是地址转换表，如图 25-25所示。正如我们前面所说的， at组已经被删除了，在这里已经用IP表来代替了。

IP地址转换表，索引＝<*ipNetToMediaIfIndex*>.<*ipNetToMediaNetAddress*>			
名　　称	数 据 类 型	R/W	描　　述
ipNetToMediaIfIndex	INTEGER	•	对应的接口：ifIndex
ipNetToMediaPhysAddress	PhysAddress	•	物理地址
ipNetToMediaNetAddress	IpAddress	•	IP地址
ipNetToMediaType	[1…4]	•	映射的类型：1 = 其他，2 = 无效的，3 = 动态的，4 = 静态的。

图25-25 IP地址转换表：ipNetToMediaTable

这里显示的是系统sun上的ARP高速缓存信息：

```
sun % arp -a
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi (140.252.13.35) at 0:0:c0:6f:2d:40
```

相应的SNMP输出：

```
sun % snmpi -a sun dump ipNetToMediaTable

ipNetToMediaIfIndex.1.140.252.13.34=1
ipNetToMediaIfIndex.1.140.252.13.35=1
ipNetToMediaPhysAddress.1.140.252.13.34=0x00:00:c0:c2:9b:26
ipNetToMediaPhysAddress.1.140.252.13.35=0x00:00:c0:6f:2d:40
ipNetToMediaNetAddress.1.140.252.13.34=140.252.13.34
ipNetToMediaNetAddress.1.140.252.13.35=140.252.13.35
ipNetToMediaType.1.140.252.13.34=dynamic(3)
ipNetToMediaType.1.140.252.13.35=dynamic(3)
```

25.8.5 icmp组

icmp组包含4个普通计数器变量（ICMP报文的输出和输入数量以及 ICMP差错报文的输入和输出数量）和22个其他ICMP报文数量的计数器： 11个是输出计数器，另外11个是输入计数器。如图25-26所示。

对于有附加代码的 ICMP报文（请回忆一下图 6-3中，有15种报文代表目的不可达），SNMP没有为它们定义专门的计数器。

25.8.6 tcp组

图25-27显示的是tcp组中的简单变量。其中的很多变量和图 18-12描述的TCP状态有关。

名　称	数据类型	R/W	描　述
icmpInMsgs	Counter		收到的ICMP报文总数
icmpInErrors	Counter		收到的有差错的ICMP报文数（例如，无效的ICMP检验和）
icmpInDestUnreachs	Counter		收到的ICMP目的站不可达报文数
icmpInTimeExcds	Counter		收到的ICMP超时报文数
icmpInParmProbs	Counter		收到的ICMP参数问题报文数
icmpInSrcQuenchs	Counter		收到的ICMP源站抑制报文数
icmpInRedirects	Counter		收到的ICMP重定向报文数
icmpInEchos	Counter		收到的ICMP回显请求报文数
icmpInEchosReps	Counter		收到的ICMP回显应答报文数
icmpInTimeStamps	Counter		收到的ICMP时间戳请求报文数
icmpInTimeStampReps	Counter		收到的ICMP时间戳应答报文数
icmpInAddrMasks	Counter		收到的ICMP地址掩码请求报文数
icmpInAddrMaskReps	Counter		收到的ICMP地址掩码应答报文数
icmpOutMsgs	Counter		输出的ICMP报文总数
icmpOutErrors	Counter		由于在ICMP报文中有一个问题（例如，缓存空间不足）而未发送的ICMP报文数
icmpOutDestUnreachs	Counter		发送的ICMP目的站不可达报文数
icmpOutTimeExcds	Counter		发送的ICMP超时报文数
icmpOutParmProbs	Counter		发送的ICMP参数问题报文数
icmpOutSrcQuenchs	Counter		发送的ICMP源站抑制报文数
icmpOutRedirects	Counter		发送的ICMP重定向报文数
icmpOutEchos	Counter		发送的ICMP回显请求报文数
icmpOutEchosReps	Counter		发送的ICMP回显应答报文数
icmpOutTimeStamps	Counter		发送的ICMP时间戳请求报文数
icmpOutTimeStampReps	Counter		发送的ICMP时间戳应答报文数
icmpOutAddrMasks	Counter		发送的ICMP地址掩码请求报文数
icmpOutAddrMaskReps	Counter		发送的ICMP地址掩码应答报文数

图25-26　icmp组中的简单变量

名　称	数据类型	R/W	描　述
tcpRtoAlgorithm	INTEGER		用来计算重传超时值的算法：1 =除下列值以外，2 = 固定的RTO, 3 = MIL-STD-1778附件B, 4 = Van Jacobson算法
tcpRtoMin	INTEGER		以毫秒计的最小重传超时值
tcpRtoMax	INTEGER		以毫秒计的最大重传超时值
tcpMaxConn	INTEGER		最大的TCP连接数。若为动态的，则值为- 1
tcpActiveOpens	Counter		从CLOSED到SYN_SENT的状态变迁数
tcpPassiveOpens	Counter		从LISTEN到SYN_RCVD的状态变迁数
tcpAttempFails	Counter		从SYN_SENT或SYN_RCVD到CLOSED的状态变迁数,加上从SYN_RCVD到LISTEN的状态变迁数
tcpEstabResets	Counter		从ESTABLISHED或CLOSE_WAIT状态到CLOSED的状态变迁数
tcpCurrEstab	Gauge		当前在ESTABLISHED或CLOSE_WAIT状态的连接数
tcpInSegs	Counter		收到的报文段的总数
tcpOutSegs	Counter		发送的报文段的总数，但将仅包含重传字节的除外
tcpRetransSegs			重传的报文段的总数
tcpInErrs	Counter		收到的具有一个差错(如无效的检验和)的报文段总数
tcpOutRsts	Counter		具有RST标志置位的报文段的总数

图25-27　tcp组中的简单变量

现在向系统sun查询一些tcp组变量：

```
sun % snmpi -a sun

snmpi> get tcpRtoAlgorithm.0 tcpRtoMin.0 tcpRtoMax.0 tcpMaxConn.0
tcpRtoAlgorithm.0=vanj(4)
tcpRtoMin.0=200
tcpRtoMax.0=12800
tcpMaxConn.0=-1
```

本系统（指SunOS4.1.3）使用的是Van Jacobson超时重传算法，超时定时器的范围在200 ms~12.8 s之间，并且对TCP连接数量没有特定的限制（这里的超时上限12.8 s恐怕有错，因为我们在21章中曾经介绍大多数应用的超时上限是64 s）。

tcp组还包括一个表格变量，即TCP连接表，如图25-28所示。对于每个TCP连接，都对应表格中的一条记录。每条记录包含5个变量：连接状态、本地IP地址、本地端口号、远端IP地址以及远端端口号。

索引＝<tcpConnLocalAddress>.<tcpConnLocalPort>.<tcpConnRemAddress>.<tcpConnRemPort>			
名　称	数据类型	R/W	描　述
tcpConnState	[1…12]	•	连接状态：1 = CLOSED, 2 = LISTEN, 3 = SYN_SENT, 4 = SYN_RCVD, 5 = ESTABLISHED, 6 = FIN_WAIT, 7 = FIN_WAIT, 8 = CLOSE_WAIT, 9 = LAST_ACK, 10 = CLOSING, 11 = TIME_WAIT, 12 = 删除TCB。管理进程对此变量可以设置的唯一值就是12（例如，立即终止此连接）
tcpConnLocalAddress	IpAddress		本地IP地址。0.0.0.0代表监听进程愿意在任何接口接受连接
tcpConnLocalPort	[1…65535]		本地端口号
tcpConnRemAddress	IpAddress		远程IP地址
tcpConnRemPort	[1…65535]		远程端口号

图25-28 TCP连接表：tcpConnTable

让我们看一看在系统sun上的这个表。由于有许多服务器进程在监听这些连接，所以我们只显示该表的一部分内容。在转储全部表格的变量之前，我们必需先建立两条TCP连接：

```
sun % rlogin gemini              gemini的IP地址是140.252.1.11
```

和

```
sun % rlogin localhost           IP地址应该是127.0.0.1
```

在所有的监听服务器进程中，我们仅仅列出了FTP服务器进程的情况，它使用21号端口。

```
sun % snmpi -a sun dump tcpConnTable

tcpConnState.0.0.0.0.21.0.0.0.0.0=listen(2)
tcpConnState.127.0.0.1.23.127.0.0.1.1415=established(5)
tcpConnState.127.0.0.1.1415.127.0.0.1.23=established(5)
tcpConnState.140.252.1.29.1023.140.252.1.11.513=established(5)

tcpConnLocalAddress.0.0.0.0.21.0.0.0.0.0=0.0.0.0
tcpConnLocalAddress.127.0.0.1.23.127.0.0.1.1415=127.0.0.1
tcpConnLocalAddress.127.0.0.1.1415.127.0.0.1.23=127.0.0.1
tcpConnLocalAddress.140.252.1.29.1023.140.252.1.11.513=140.252.1.29
```

```
tcpConnLocalPort.0.0.0.0.21.0.0.0.0.0=21
tcpConnLocalPort.127.0.0.1.23.127.0.0.1.1415=23
tcpConnLocalPort.127.0.0.1.1415.127.0.0.1.23=1415
tcpConnLocalPort.140.252.1.29.1023.140.252.1.11.513=1023

tcpConnRemAddress.0.0.0.0.21.0.0.0.0.0=0.0.0.0
tcpConnRemAddress.127.0.0.1.23.127.0.0.1.1415=127.0.0.1
tcpConnRemAddress.127.0.0.1.1415.127.0.0.1.23=127.0.0.1
tcpConnRemAddress.140.252.1.29.1023.140.252.1.11.513=140.252.1.11

tcpConnRemPort.0.0.0.0.21.0.0.0.0.0=0
tcpConnRemPort.127.0.0.1.23.127.0.0.1.1415=1415
tcpConnRemPort.127.0.0.1.1415.127.0.0.1.23=23
tcpConnRemPort.140.252.1.29.1023.140.252.1.11.513=513
```

对于rlogin到gemini，只显示一条记录，这是因为gemini是另外一个主机。而且我们仅仅能够看到连接的客户端信息（端口号是1023）。但是Telnet连接，客户端和服务器端都显示（客户端口号是1415，服务器端口号是23），这是因为这种连接通过环回接口。同时我们还可以看到，FTP监听服务器程序的本地IP地址是0.0.0.0。这表明它可以接受通过任何接口的连接。

25.9 其他一些例子

现在开始回答前面一些没有回答的问题，我们将用SNMP的知识进行解释。

25.9.1 接口MTU

回忆一下在11.6节的实验中，我们试图得出一条从netb到sun的SLIP连接的MTU。现在可以采用SNMP得到这个MTU。首先从IP路由表中取到SLIP连接（140.252.1.29）的接口号（ipRouteIfIndex），然后就可以用这个数值进入接口表并且取得想要的SLIP连接的MTU（通过SLIP的描述和数据类型）。

```
sun % snmpi -a netb -c secret

snmpi> get ipRouteIfIndex.140.252.1.29
ipRouteIfIndex.140.252.1.29=12
snmpi> get ifDescr.12 ifType.12 ifMtu.12
ifDescr.12="Telebit NetBlazer dynamic dial virtual interface"
ifType.12=other(1)
ifMtu.12=1500
```

可以看到，即使连接的类型是SLIP连接，但是MTU仍设置为以太网，其值为1500，目的可能是为了避免分片。

25.9.2 路由表

回忆一下在14.4节中，我们讨论了DNS如何进行地址排序的问题。当时我们介绍了从域名服务器返回的第1个IP地址是和客户有相同子网掩码的情况。还介绍了用其他的IP地址也会正常工作，但是效率比较低。现在我们从SNMP的角度来查阅路由表的入口，在这里将用到前面章节中和IP路由有关的很多相关知识。

路由器gemini是一个多接口主机，有两个以太网接口。首先确认一下两个接口都可以Telnet登录：

```
sun % telnet 140.252.1.11 daytime
Trying 140.252.1.11 ...
Connected to 140.252.1.11.
```

```
Escape character is '^]'.
Sat Mar 27 09:37:24 1993
Connection closed by foreign host.

sun % telnet 140.252.3.54 daytime
Trying 140.252.3.54 ...
Connected to 140.252.3.54.
Escape character is '^]'.
Sat Mar 27 09:37:35 1993
Connection closed by foreign host.
```

可以看出这两个地址的连接没有什么区别。现在我们采用 traceroute命令来看一下对于每个地址，是否有选路方面的不同：

```
sun % traceroute 140.252.1.11
traceroute to 140.252.1.11 (140.252.1.11), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  299 ms  234 ms  233 ms
 2  gemini (140.252.1.11)  233 ms  228 ms  234 ms

sun % traceroute 140.252.3.54
traceroute to 140.252.3.54 (140.252.3.54), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  245 ms  212 ms  234 ms
 2  swnrt (140.252.1.6)  233 ms  229 ms  234 ms
 3  gemini (140.252.3.54)  234 ms  233 ms  234 ms
```

可以看到：如果采用属于140.252.3子网的地址，就多了额外的一跳。下面解释造成这个额外一跳的原因。

图25-29是系统的连接关系图。从traceroute命令的输出结果可以看出主机gemini和路由器swnrt都连接了两个网段：140.252.3子网和140.252.1子网。

回忆一下在图4-6中，我们解释了路由器netb采用ARP代理进程，使得sun工作站好象是直接连接到140.252.1子网上的情况。我们忽略了sun和netb之间SLIP连接的调制解调器，因为这和我们这里的讨论不相关。

在图25-29中，我们用虚线箭头画出了当Telnet到140.252.3.54时的路径。返回的数据报怎么知道直接从gemini到netb，而不是从原路返回呢？我们采用在8.5节中介绍过的，带有宽松选路特性的traceroute版本来解释：

```
sun % traceroute -g 140.252.3.54 sun
traceroute to sun (140.252.13.33), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  244 ms  256 ms  234 ms
 2  * * *
 3  gemini (140.252.3.54)  285 ms  227 ms  234 ms
 4  netb (140.252.1.183)  263 ms  259 ms  294 ms
 5  sun (140.252.13.33)  534 ms  498 ms  504 ms
```

当在命令中指明是宽松源站选路时，swnrt路由器就不再有响应。看一下前面没有指明源站选路的traceroute命令输出，可以看出swnrt路由器是事实上的第2跳。超时数据必须这样设置的原因是：当数据报指定了宽松源站选路选项时，该路由器没有发生ICMP超时差错。所以在traceroute命令的输出中可以得出，返回路径是从gemini(TTL 3, 4和5)路由器直接到达netb路由器，而不通过swnrt路由器。

还剩下一个需要用SNMP来解释的问题就是：在netb路由器的路由表中，哪条信息代表寻径到140.252.3？该信息表示netb路由器把分组发送给swnrt而不是直接发送给gemini？用get命令来取下一跳路由器的值。

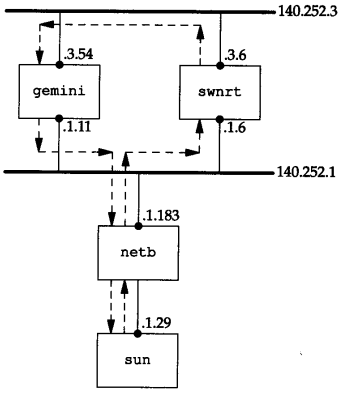


图25-29 例子中的网络拓扑结构

```
sun % snmpi -a netb -c secret get ipRouteNextHop.140.252.3.0
ipRouteNextHop.140.252.3.0=140.252.1.6
```

正如我们所看到发生的那样，路由表设置使得 netb路由器把分组发送到swnrt路由器。

为什么gemini路由器直接把分组回送给netb路由器？那是因为在gemini路由器端，它要回送的分组目的地址是140.252.1.29，而子网140.252.1是直接连接到gemini路由器上的。

从上面这个例子可以看出选路的策略。由于gemini是打算作一个多接口主机而不是路由器，所以默认的到140.253.3子网的路由器是swnrt。这是多接口主机和路由器之间差异的一个典型例子。

25.10 Trap

本章我们看到的例子都是从管理进程到代理进程的。当然代理进程也可以主动发送 trap到管理进程，以告诉管理进程在代理进程侧有某些管理进程所关心的事件发生，如图 25-1所示。trap发送到管理进程的162号端口。

在图25-2中，我们已经描述了 trap PDU的格式。在下面关于 tcpdump输出内容中我们将再一次用到这些字段。

现在已经定义了6种特定的trap类型，第7种trap类型是由供应商自己定义的特定类型。图25-30给出了trap报文中trap类型字段的内容。

trap类型	名　称	描　述
0	coldStart	代理进程对自己初始化
1	warmStart	代理进程对自己重新初始化
2	linkDown	一个接口已经从工作状态改变为故障状态（图 25-18）。报文中的第一个变量标识此接口
3	linkUp	一个接口已经从故障状态改变为工作状态（图 25-18）。报文中的第一个变量标识此接口
4	authenticationFailure	从SNMP管理进程收到无效共同体的报文
5	egpNeighborLoss	一个EGP邻站已变为故障状态。报文中的第一个变量包含此邻站的IP地址
6	enterpriseSpecific	在这个特定的代码字段中查找trap信息

图25-30　trap的类型

用tcpdump命令来看看trap的情况。我们在系统sun上启动SNMP代理进程，然后让它产生coldStart类型的trap（我们告诉代理进程把trap信息发送到bsdi主机。虽然在该主机上并没有运行处理trap的管理进程，但是可以用tcpdump来查看产生了什么样的分组。回忆一下在图25-1中，trap是从代理进程发送到管理进程的，而管理进程不需要给代理进程发送确认。所以我们不需要trap的处理程序）。然后我们用snmpi程序发送一个请求，但该请求的共同体名称是无效的。这将产生一个authenticationFailure类型的trap。图25-31显示了命令的输出结果。

```
1  0.0              sun.snmp > bsdi.snmp-trap: C=traps Trap(28)
                    E:unix.1.2.5 [140.252.13.33] coldStart 20

2  18.86 (18.86)    sun.snmp > bsdi.snmp-trap: C=traps Trap(29)
                    E:unix.1.2.5 [140.252.13.33] authenticationFailure 1907
```

图25-31　tcpdump 输出的由SNMP代理进程产生的trap

首先注意一下两个 UDP数据报都是从 SNMP代理进程（端口是 161，图中显示的名称是 snmp）发送到目的端口号是162的服务器进程上的（图中显示的名称是 snmp-trap）。

再注意一下C=traps是trap报文的共同体名称。这是ISODE SNMP代理进程的配置选项。

下一个要注意的是：第1行中的Trap（28）和第2行中的Trap（29）是PDU类型和长度。

两个输出行的第一项内容都是相同的 E:unix.1.2.5。这代表企业名字段和代理进程的sysObjiectID。它是图25-6中1.3.6.1.4.1（iso.org.dod.internet.private.enterparses）结点下面的某个结点，所以代理进程的对象标识是1.3.6.1.4.1.4.1.2.5。它的简称是：unix.agents.fourBSD-isode.5。最后一个数字“5”代表ISODE代理进程软件的版本号。这些企业名的值代表了产生trap的代理进程软件信息。

输出的下一项是代理进程的IP地址（140.252.13.33）。

在第1行中，trap的类型显示的是coldStart，第2行中，显示的是authenticationFailure。与之相对应的trap类型值是0和4（如图25-30所示）。由于这些都不是厂家自定义的trap，所以特定代码必须是0，在图中没有显示。

输出的最后分别是20和1907，这是时间戳字段。它是TimeTicks类型的值，表示从代理进程初始化开始到trap发生所经历了多少个百分之一秒。在冷启动 trap的情况下，在代理进程初始化后到trap的产生共经历了200 ms。同样，tcpdump的输出表明第2个trap在第1个trap产生的18.86s后出现，这对应于打印出的1907个百分之一秒减去200 ms所得到的值。

图25-2表明trap报文还包含很多代理进程发送给管理进程的变量，但在这些在例子中没有再讨论。

25.11 ASN.1和BER

在正式的SNMP规范中都是采用ASN.1（Abstract Syntax Notation 1）语法，并且在SNMP报文中比特的编码采用BER（Basic Encoding Rule）。和其他介绍SNMP的书不同，我们有目的地把ASN.1和BER的讨论放到最后。因为如果放在前面讨论，有可能使读者产生混淆而忽略了SNMP的真正目的是进行网络管理。在这里我们也只是对这两个概念简单地进行解释，[Rose 1990]的第8章详细讨论了ASN.1和BER。

ASN.1是一种描述数据和数据特征的正式语言。它和数据的存储及编码无关。 MIB和SNMP报文中的所有的字段都是用 ASN.1描述的。例如：对于 SMI中的ipAddress数据类型，ASN.1是这样描述的：

```
IpAddress ::=
    [APPLICATION 0]              -- in network-byte order
        IMPLICIT OCTET STRING (SIZE (4))
```

同样，在MIB中，简单变量的定义是这样描述的：

```
udpNoPorts OBJECT-TYPE
    SYNTAX  Counter
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "The total number of received UDP datagrams for which
            there was no application at the destination port."
    ::= { udp 2 }
```

用SEQUENCE和SEQUENCE OF来定义表格的描述更加复杂。

当有了这样的ASN.1定义，可以有多种编码方法把数据编码为传输的比特流。 SNMP使用的编码方法是BER。例如，对于一个简单的整数如64，在BER中需要用3个字节来表示。第一个字节说明类型是一个整数，下个字节说明用了多少个字节来存储该整数（在这里是 1），最后一个字节才是该整数的值。

幸运的是，ASN.1和BER这两个繁琐的概念仅仅在实现SNMP的时候才重要，对我们理解

网络管理的概念和流程并没有太大的关系。

25.12 SNMPv2

在1993年，发表了定义新版本SNMP的11个RFC。RFC 1441 [Case et al. 1993]是其中的第一个，它系统地介绍了SNMPv2。同样，有两本书 [Stallings 1993; Rose 1994]也对SNMPv2进行了介绍。现在已经有两个SNMPv2的基本模型（参见附录 B.3中的[Rose 1994]），但是厂家的实现到1994年才能广泛使用。

在本节中，我们主要介绍SNMPv1和SNMPv2之间的重要区别。

1) 在SNMPv2中定义了一个新的分组类型`get-bulk-request`，它高效率地从代理进程读取大块数据。

2) 另的一个新的分组类型是`inform-request`，它使一个管理进程可以向另一个管理进程发送信息。

3) 定义了两个新的MIB，它们是：SNMPv2 MIB和SNMPv2-M2M MIB（管理进程到管理进程的MIB）。

4) SNMPv2的安全性比SNMPv1大有提高。在SNMPv1中，从管理进程到代理进程的共同体名称是以明文方式传送的。而SNMP v2可以提供鉴别和加密。

厂家提供的设备支持SNMPv2的会越来越多，管理站将对两个版本的 SNMP代理进程进行管理。[Routhier 1993]中描述了如何将SNMPv1的实现扩展到支持SNMPv2。

25.13 小结

SNMP是一种简单的、SNMP管理进程和SNMP代理进程之间的请求-应答协议。MIB定义了所有代理进程所包含的、能够被管理进程查询和设置的变量，这些变量的数据类型并不多。

所有这些变量都以对象标识符进行标识，这些对象标识符构成了一个层次命名结构，由一长串的数字组成，但通常缩写成人们阅读方便的简单名字。一个变量的特定实例可以用附加在这个对象标识符后面的一个实例来标识。

很多SNMP变量是以表格形式体现的。它们有固定的栏目，但有多少条记录并不固定。对于SNMP来讲，重要的是对表格中的每一行如何进行标识（尤其当我们不知道表格中有多少条记录时）以及如何按字典方式进行排序（“先列后行”的次序）。最后要说明的一点是：SNMP的`get-next`操作符对任何SNMP管理进程来讲都是最基本的操作。

然后我们介绍了下列的SNMP变量组：system、interface、address translation、IP、ICMP、TCP和UDP。接着是两个例子，一个介绍如何确定一个接口的MTU，另外一个介绍如何获取路由器的路由信息。

在本章的后面介绍了SNMP的trap操作，它是当代理进程发生了某些重大事件后主动向管理进程报告的。最后我们简单介绍了 ASN.1和BER，这两个概念比较繁琐，但所幸的是，它对我们了解SNMP并不十分重要，仅仅在实现SNMP的时候才要用到。

习题

25.1 我们说过采用两个不同的UDP端口（161和162）可以使得一个系统既可以是管理进程，也可以是代理进程。如果对管理进程和代理进程采用一个同样的端口，会出现什么情况？

25.2 用`get-next`操作，如何列出一张路由表的完整信息？

第26章　Telnet和Rlogin：远程登录

26.1　引言

远程登录（Remote Login）是Internet上最广泛的应用之一。我们可以先登录（即注册）到一台主机然后再通过网络远程登录到任何其他一台网络主机上去，而不需要为每一台主机连接一个硬件终端（当然必须有登录帐号）。

在TCP/IP网络上，有两种应用提供远程登录功能。

1) Telnet是标准的提供远程登录功能的应用，几乎每个TCP/IP的实现都提供这个功能。它能够运行在不同操作系统的主机之间。Telnet通过客户进程和服务器进程之间的选项协商机制，从而确定通信双方可以提供的功能特性。

2) Rlogin起源于伯克利Unix，开始它只能工作在Unix系统之间，现在已经可以在其他操作系统上运行。

在本章中，我们将介绍Telnet和Rlogin。首先介绍Rlogin，因为Rlogin比较简单。

Telnet是一种最老的Internet应用，起源于1969年的ARPANET。它的名字是“电信网络协议（telecommunication network protocol）”的缩写词。

远程登录采用客户-服务器模式。图26-1显示的是一个Telnet客户和服务器的典型连接图（对于Rlogin的客户和服务器连接图，我们可以画得更加简单）。

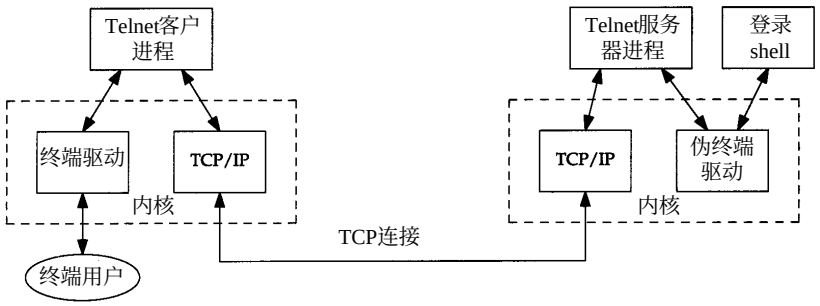


图26-1　客户-服务器模式的Telnet简图

在这张图中，有以下要点需要注意：

1) Telnet客户进程同时和终端用户和TCP/IP协议模块进行交互。通常我们所键入的任何信息的传输是通过TCP连接，连接的任何返回信息都输出到终端上。

2) Telnet服务器进程经常要和一种叫做“伪终端设备”（pseudo-terminal device）打交道，至少在Unix系统下是这样的。这就使得对于登录外壳(shell)进程来讲，它是被Telnet服务器进程直接调用的，而且任何运行在登录外壳进程处的程序都感觉是直接和一个终端进行交互。对于像满屏编辑器这样的应用来讲，就像直接在和终端打交道一样。实际上，如何对服务器进程的登录外壳进程进行处理，使得它好像在直接和终端交互，往往是编写远程登录服务器

进程程序中最困难的方面之一。

3) 仅仅使用了一条TCP连接。由于客户进程必须多次和服务器进程进行通信（反之亦然），这就必然需要某些方法，来描绘在连接上传输的命令和用户数据。我们在后面的内容中会介绍Telnet和Rlogin是如何处理这个问题的。

4) 注意在图 26-1中，我们用虚线框把终端驱动进程和伪终端驱动进程框了起来。在TCP/IP实现中，虚线框的内容一般是操作系统内核的一部分。 Telnet客户进程和服务器进程一般只是属于用户应用程序。

5) 把服务器进程的登录外壳进程画出来的目的是为了说明：当我们想登录到系统的时候，必须要有一个帐号， Telnet和Rlogin都是如此。

对于Telnet和Rlogin，如果比较一下它们客户进程和服务器进程源代码的数量，就可以知道这两者的复杂程度。图 26-2显示了伯克利不同版本的 Telnet和Rlogin客户进程和服务器进程源代码的数量。

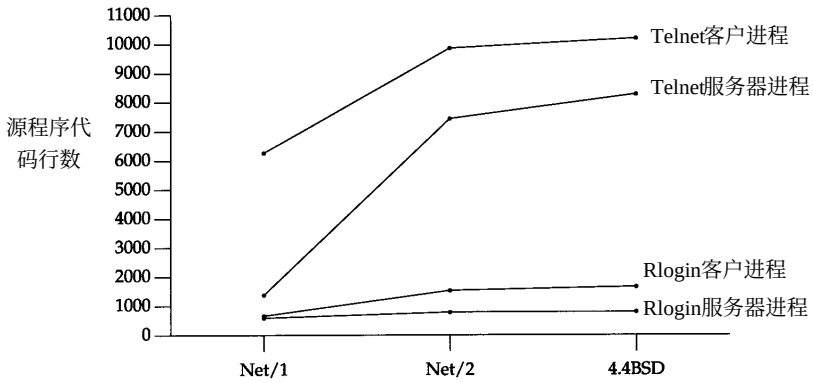


图26-2 Telnet/ Rlogin/客户进程/服务器进程的源代码数量比较

现在，不断有新的Telnet选项被添加到Telnet中去，这就使得Telnet实现的源代码数量大大增加，而Rlogin依然变化不大，还是比较简单。

远程登录不是那种有大量数据报传输的应用。正如我们前面讲到的一样，客户进程和服务器进程交互的分组大多比较小。 [Paxson 1993]发现客户进程发出的字节数（用户在终端上键入的信息）和服务器进程端发出的字节数的数量之比是 1:20。这是因为我们在终端上键入的一条短命令往往令服务器进程端产生很多输出。

26.2 Rlogin协议

Rlogin的第一次发布是在 4.2BSD中，当时它仅能实现 Unix主机之间的远程登录。这就使得Rlogin比Telnet简单。由于客户进程和服务器进程的操作系统预先都知道对方的操作系统类型，所以就不需要选项协商机制。在过去的几年中， Rlogin协议也派生出几种非 Unix环境的版本。

RFC 1282 [Kantor 1991]详细说明了Rlogin协议。类似于选路信息协议（RIP）的RFC，它是Rlogin用了许多年后才发布的。 [Stevens 1990]的第15章介绍了远程登录的客户进程及服务器进程端的编程，并且给出了 Rlogin的客户进程及服务器进程的完整源代码。 [Comer和Stevens 1993]的第25章和第26章给出了Telnet的客户进程的实现细节和源代码。

26.2.1 应用进程的启动

Rlogin的客户进程和服务器进程使用一个TCP连接。当普通的TCP连接建立完毕之后，客户进程和服务器进程之间将发生下面所述的动作。

1) 客户进程给服务器进程发送4个字符串：(a) 一个字节的0；(b) 用户登录进客户进程主机的登录名，以一个字节的0结束；(c) 登录服务器进程端主机的登录名，以一个字节的0结束；(d) 用户终端类型名，紧跟一个正斜杠“/”，然后是终端速率，以一个字节的0结束。在这里需要两个登录名字，这是因为用户登录客户和服务器的名称有可能不一样。
   由于大多满屏应用程序需要知道终端类型，所以终端类型也必须发送到服务器进程。发送终端速率的原因是因为有些应用随着速率的改变，它的操作也有所变化。例如vi编辑器，当速率比较小的时候，它的工作窗口也变小。所以它不能永远保持同样大小的窗口。
2) 服务器进程返回一个字节的0。
3) 服务器进程可以选择是否要求用户输入口令。这个步骤的数据交互没有什么特别的协议，而被当作是普通的数据进行传输。服务器进程给客户进程发送一个字符串（显示在客户进程的屏幕上），通常是password: 。如果在一定的限定时间内（通常是60秒）客户进程没有输入口令，服务器进程将关闭该连接。
   通常可以在服务器进程的主目录(home directory)下生成一个文件（通常叫 .rhosts），该文件的某些行记录了一个主机名和用户名。如果从该文件中已经记录的主机上用已经记录的用户名进行登录，服务器进程将不提示我们输入口令。但是很多关于安全性的文献，如[Curry 1992]，强烈建议不要采用这种方法，因为这存在安全漏洞。
   如果提示输入口令，那么我们输入的口令将以明文的形式发送到服务器进程。我们所键入的每个字符都是以明文的格式传输的。所以某人只要能够截取网络上的原始传输的分组，他就可以截获用户口令。针对这个问题，新版本的 Rlogin客户程序，例如4.4BSD版本的客户程序，第一次采用了Kerberos安全模型。Kerberos安全模型可以避免用户口令以明文的形式在网络上传输。当然，这要求服务器进程也支持 Kerberos ([Curry 1992]详细描述了Kerberos安全模型)。
4) 服务器进程通常要给客户进程发送请求，询问终端的窗口大小（将在后面解释）。

客户进程每次给服务器进程发送一个字节的内容，并且接收服务器进程的所有返回信息。这在19.2节中已经介绍过了。同样我们也采用了Nagle算法（在19.4节中曾经介绍），该算法可以保证在速率较低的网络上，若干输入字节以单个 TCP报文段传输。操作其实很简单：用户键入的所有东西被发送到服务器，服务器发送给客户的任何信息返回到用户的屏幕上。

另外，服务器和客户之间还可以互相发送命令。在介绍这些命令之前，先介绍需要用到这些命令的场合。

26.2.2 流量控制

默认情况下，流量控制是由 Rlogin的客户进程完成的。客户进程能够识别用户键入的STOP和START的ASCII字符（Control_S和Control_Q），并且终止或启动终端的输出。

如果不是这样，每次我们为终止终端输出而键入的 Control_S字符将沿网络传输到服务器进程，这时服务器进程将停止往网络上写数据。但是在写操作终止之前，服务器进程可能已经往网络上写了一窗口的输出数据。也就是说，在输出停止之前，成千上万的数据字节还将

在屏幕上显示。图26-3显示了这个情况。

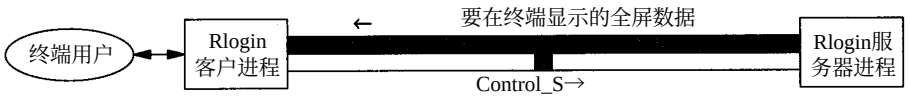


图26-3 服务器进程执行STOP/START的情况

对于一个交互式用户来讲，Control_S字符的响应延时是较大的。

有时候，服务器的应用程序需要解释输入的每个字节，但又不想让客户对它的输入内容进行处理，例如对控制字符如Control_S和Control_Q进行特殊处理（emacs编辑器就是这样的一个例子，它把Control_S和Control_C作为自己的命令）。解决这个问题的办法就是由服务器告诉客户是否要进行流量控制。

26.2.3 客户的中断键

当我们为中断服务器正在运行的进程而键入一个中断字符时（通常是DELETE或Control_C），会发生和流量控制相同的问题。这个情况和图26-3所示的类似，在一条TCP连接的管道上，从服务器进程向客户进程正在发送大量的数据，而客户进程同时在向服务器进程传输中断字符。而我们的本意是要中断字符尽快终止某个进程，使屏幕上不再有任何响应输出。

在流量控制和中断键这两种情况中，流量控制机制很少终止客户进程到服务器进程的数据流。这个方向仅仅包含我们键入的字符。所以对于从客户输出到服务器的特殊输入字符（Control_S和中断字符）不需要采用TCP的紧急方式（urgent mode）。

26.2.4 窗口大小的改变

如果是窗口风格的显示方式，当应用程序在运行的时候，我们还可以动态地改变窗口的大小。一些应用程序（典型的如那些操作整个窗口的应用程序，如全屏编辑器）需要知道窗口大小的变化。目前大多数Unix系统提供这种功能，可以告诉应用程序关于窗口大小的变化。

对于远程登录这种情况，窗口大小的变化发生在客户端，而运行在服务器端的应用程序需要知道窗口大小变化。所以Rlogin的客户需要采用某些方法来通知服务器窗口大小变化的情况以及新窗口的大小。

26.2.5 服务器到客户的命令

现在我们介绍通过TCP连接，Rlogin服务器进程可以发送给客户进程的4条命令。问题是只有一条TCP连接可供使用，所以服务器进程必须给这些命令字节做标记，使得客户进程可以从数据流中识别出这些是命令，而不是显示在终端上。所以我们将使用TCP的紧急方式（在20.8节中曾经介绍）。

当服务器要给客户发送命令时，服务器就进入紧急方式，并且把命令放在紧急数据的最后一个字节中。当客户进程收到这个紧急方式通知时，它从连接上读取数据并且保存起来，直到读到命令字节（即紧急数据的最后一个字节）。这时候客户进程根据读到的命令，再决定对于所读到并保存起来的数据是显示在终端上还是丢弃它。图26-4介绍了这4个命令。

采用TCP紧急方式发送这些命令的一个原因是第一个命令（“清仓输出(flush output)”）需要立即发送给客户，即使服务器到客户的数据流被窗口流量控制所终止。这种情况下，即服

务器到客户的输出被流量控制所终止的情况是经常发生的，这是因为运行在服务器的进程的输出速率通常大于客户终端的显示速率。另一方面，客户到服务器的数据流很少被流量控制所终止，因为这个方向的数据流仅仅包含用户所键入的字符。

字节	描　述
0x02	清仓输出。客户丢弃所有从服务器收到的数据，直到命令字节（紧急数据的最后一个字节）。客户还丢弃任何有可能被缓存的挂起输出 (pending output)。当服务器收到客户发出的中断命令时，就发送此命令
0x10	客户停止执行流量控制
0x20	客户继续进行流量控制处理
0x80	客户立即响应，将当前窗口大小发送给服务器，并在今后当窗口大小变化时通知服务器。通常，当连接建立后，服务器就立即发送这个命令

图26-4 服务器到客户的Rlogin命令

回忆一下图20-14中的例子，在那里我们介绍了即使窗口大小是 0时，紧急通知通过网络进行传输的情况（在下节中，我们还将介绍一个类似的例子）。其他的3个命令实时性并不特别强，但为了简单起见，也采用了和第一个命令相同的技术。

26.2.6 客户到服务器的命令

对于客户到服务器的命令，只定义了一条命令，那就是：将当前窗口大小发送给服务器。当客户的窗口大小发生变化时，客户并不立即向服务器报告，除非收到了服务器发来的 0x80命令（图26-4中有介绍）。

同样，由于只存在一条 TCP连接，客户必须对在连接上传输的该命令字节进行标注，使得服务器可以从数据流中识别出命令，而不是把它发送到上层的应用程序中去。处理的方法就是在两个字节的0xff后面紧跟着发送两个特殊的标志字节。

对于窗口大小命令，两个标志字节是 ASCII码的字符‘ s ’。之后是4个16 bit长的数据（按网络字节顺序），分别是：行数（例如， 25），每列的字符数（例如， 80），X方向的像素数量，Y方向的像素数量。通常情况下，后两个 16bit是0，因为在Rlogin服务器进程调用的应用程序中，通常是以字符为单位来度量屏幕的，而不是像素点。

上面我们介绍的从客户进程到服务器进程的命令采用带内信令（ in-band signaling），这是因为命令字节和其他的普通数据一起传输。选择 0xff字节来表示这个带内信令的原因是：一般用户的操作不会产生0xff这个字节。所以说Rlogin是不完备的，如果我们采用某种方法，使得通过键盘就可以产生两个连续的 0xff字节，而且正好在这之前是两个 ASCII的‘ s ’字符，那么下面的8个字节就会被误认为是窗口大小了。

图26-4中介绍的是从服务器到客户的 Rlogin命令，由于大多数的 API采用的技术叫做“ 带外数据（out-of-band data）”，所以我们就称它为带外信令 (out-of-band signaling)。但是回忆一下在20.8节中对TCP紧急方式的讨论，在那里我们说紧急方式数据不是带外数据，命令字节是按照普通数据流进行传输的，特殊之处是采用了紧急指针。

既然带内信令被用来传输从客户到服务器的命令，那么服务器进程必须检查从客户进程收到的每个字节，看看是否有两个连续的 0xff字节。但是对于采用带外信令的、从服务器传输到到客户的命令，客户进程不需要检查收到的每个字节，除非服务器进程进入了紧急方

式。即使在紧急方式下，客户进程也仅仅需要留意紧急指针所指向的字节。而且由于从客户进程到服务器的数据流量和相反方向的数据流量之比是 1:20，这就暗示带内信令适合于数据量比较小的情况（从客户到服务器），而带外信令适合于数据量比较大的情况（从服务器到客户）。

26.2.7 客户的转义符

通常情况下，我们向Rlogin客户进程键入的信息将传输到服务器进程。但是有些时候，我们并不需要把键入的信息传输到服务器，而是要和 Rlogin客户进程直接通信。方法是在一行的开头键入代字符(tilde)“ ~ ”，紧跟着是下列4个字符之一：

1) 以一个句号结束客户进程。

2) 以文件结束符（通常是Control_D）结束客户进程。

3) 以任务控制挂起符（通常是Control_Z）挂起客户进程。

4) 以任务控制延迟挂起符（通常是Control_Y）来挂起仅仅是客户进程的输入。这时，不管客户运行什么程序，键入的任何信息将由该程序进行解释，但是从服务器发送到客户的信息还是输出到终端上。这非常适合当我们需要在服务器上运行一个长时间程序的场合，我们既想知道该程序的输出结果，同时还想在客户上运行其他程序。

只有当客户进程的Unix系统支持任务控制时，后两个命令才有效。

26.3 Rlogin的例子

在这里举两个例子：第一个是当Rlogin会话建立的时候，客户和服务器的协议交互；从第二个例子可以看到，当用户键入中断键以取消正在服务器运行的程序时，服务器将产生很多输出。在图19-2中，我们给出了通常情况下， Rlogin会话上的数据流交互情况。

26.3.1 初始的客户-服务器协议

图26-5显示的是从主机bsdi到服务器svr4的Rlogin建立一个连接时的时间系列(在图中，去掉了通常的TCP连接的建立过程，窗口通告以及服务类型信息)。

上节介绍的协议对应图中的报文段 1~9。客户发送一个字节的 0（报文段1）之后发送3个字符串（报文段 3）。在本例中，这 3个字符串分别是： rstevens（客户的登录名）、rstevens（服务器的登录名）和 ibmpc3/9600（终端类型和速率）。当服务器确认了这些信息后回送一个字节的0（报文段5）。

然后服务器发送窗口请求命令（报文段 7）。这是采用TCP紧急方式发送的，我们又一次看到一个实现(SVR4)采用较老的但更普通的解释，即紧急指针指明的序号是紧急数据的最后一个字节加1。客户回送12字节的数据：2字节的0xff，2字节的‘ s ’，4个16 bit长度的窗口数据。

下面的4个报文段（ 10，12，14和16）是由服务器发送的，是从服务器操作系统的问候(greeting)。之后报文段18是一个7字节长度的外壳进程提示符“ svr4% ”。

客户输入的信息如图 19-2所示，每次发送一个字节。客户和服务器都可以主动中断该连接。如果我们输入一个命令，让服务器的外壳程序终止运行，那么服务器将中断该连接。如果我们给Rlogin客户键入一个转移符（通常是一个“ ~ ”），紧跟着一个句点或者是一个文件结束符号，那么客户将主动关闭该连接。

图26-5中，客户进程的端口号是1023，这是由IANA分配的（在1.9节中介绍）。Rlogin协议要求客户进程用小于1024的端口号，术语叫做保留端口。在Unix系统中，客户进程一般不能使用保留端口号，除非客户进程具有超级用户权限。这是客户进程和服务器进程相互鉴别的一部分，这种鉴别可以使得用户不需要口令而可以登录。[Stevens 1990]详细讨论了客户进程和服务器进程相互鉴别的过程和有关保留端口号的问题。

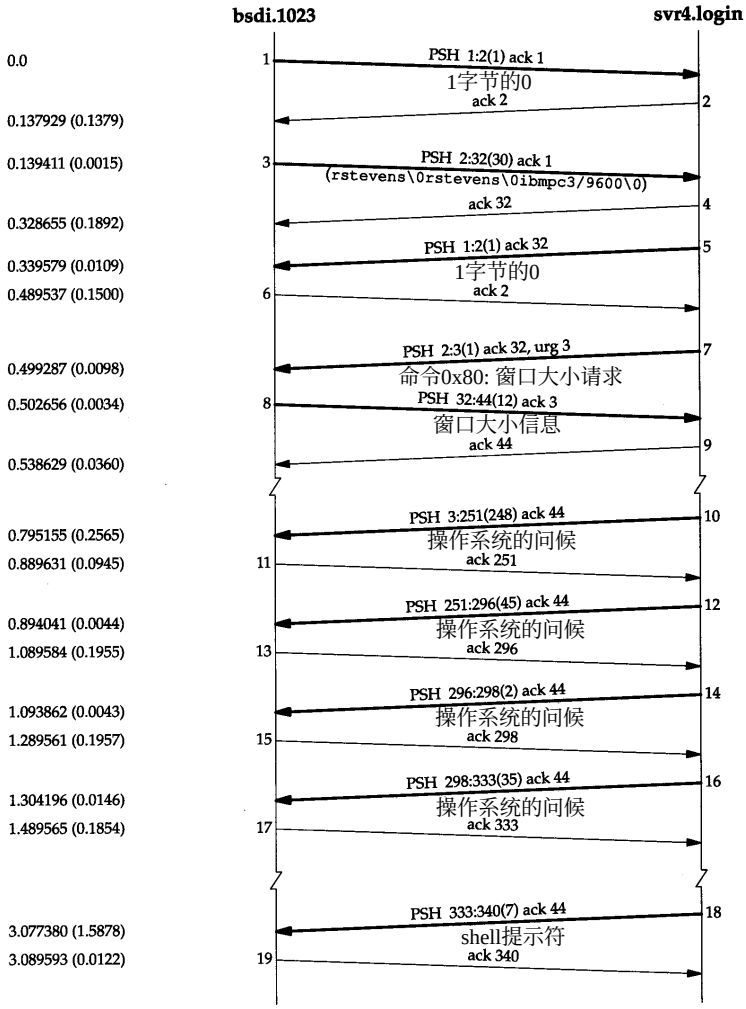


图26-5 Rlogin连接时间次序

26.3.2 客户中断键

让我们看一下另外一个例子，这个例子涉及到TCP的紧急方式。当数据流已经终止时，我们键入中断键。这个例子要用到前面讲到的很多TCP算法如：紧急方式、糊涂窗口避免技术、窗口流量控制和坚持计时器。在主机sun上运行客户进程。我们登录到主机bsdi，向终端输出一个大文本文件，然后键入Control_S中断输出。当输出停止时，我们键入中断键（DELETE）以异常方式中止该进程。

```
sun % rlogin bsdi
                                          所有操作系统的问候
bsdi % cat /usr/share/misc/termcap        向终端输出大文件
                                          大量的终端输出
                                          键入Control_S以中断输出,
                                          然后等待直到输出停止
^?                                        键入中断键，而这被回显了,
bsdi %                                    然后输出了提示符
```

下面这些要点是关于客户、服务器和连接的状态的概述：

1) 键入Control_S以停止终端的输出。

2) 用户终端的输出缓存很快被填满，所以Rlogin的客户向终端的写操作被阻塞。

3) 此时客户也不能从网络连接上读取数据，所以客户的TCP接收缓存也将被填满。

4) 当接收缓存已满时，客户进程的TCP会向服务器进程的TCP通告现在的接收窗口是0。

5) 当服务器收到客户的窗口为0时，将停止向客户发送数据，这样，服务器的发送缓存也将被填满。

6) 由于发送缓存已满，所以Rlogin服务器进程将停止。这样，Rlogin服务器将不能从服务器运行的应用程序（cat）处读取数据。

7) 当cat程序的输出缓存也被填满时，cat也将停止。

8) 然后我们用中断键来终止服务器上的cat程序。这个命令从客户的TCP传输到服务器的TCP，这是因为该方向的数据传输没有被流量控制所终止。

9) cat应用程序收到中断命令并且终止。这使得它的输出缓存（也就是Rlogin服务器进程读取数据的地方）被清空，这将唤醒Rlogin服务器进程。然后Rlogin服务器进程进入紧急方式，向客户进程发送“清仓输出”命令（0x02）。

图26-6概括了从服务器到客户的数据流（图中的序号就是下面将介绍的图中的时间系列）。

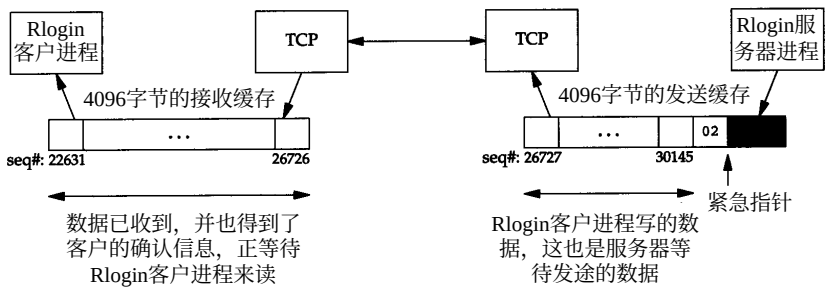


图26-6 Rlogin例子中，服务器进程到客户进程的数据流概述

发送缓存的阴影部分是4096字节的缓存中没有被使用的部分。图26-7是该例子的时间系列。

在报文段1~3，服务器进程向客户进程发送满长度（即1024字节）的TCP报文段。由于此时客户进程不能向终端写信息，客户进程也不能从网络上读数据，所以在报文段4中，客户进程向服务器进程发送ACK确认时，告诉服务器进程此时接收窗口是1024个字节。在报文段5中，服务器进程发送的数据长度就不再是满长度的了。同样，报文段6中客户进程的确认信号所带的接收窗口大小是此时接收缓存的空余字节长度。那么在报文段5中，客户进程ACK信号中为什么接收窗口大小是349而不是0呢？这是因为如果发送的是0（糊涂窗口避免技术），那么窗口指针将右边界移动到了左边界，而这是绝对不能发生的（见20.3节）。当服务器进程收到报

文段6的ACK信号后，它就不能再发送全长的数据报了，这时候它就采用糊涂窗口症避免技术，不发送任何东西，同时置一个5秒的坚持计时器。当计时器超时，服务器进程就发送一个349字节大小的数据（如报文段7）。由于此时客户进程依然不能输出接收缓存的信息，所以接收缓存将被填满，客户进程将发送ACK信号，此时接收窗口大小为0（如报文段8）。

这时候我们键入中断键并且以报文段9显示的那样传输。此时的接收窗口大小依然为0。当服务器进程接收到该中断键后，服务器进程把它发送给应用程序（cat），应用程序就终止。由于应用程序被终端中断键所终止，应用程序就清空它的输出缓存。服务器进程发现该变化后就通过TCP紧急方式向客户进程发送“清仓输出”命令，这如报文段10所示。注意命令字节0x02放在第30146字节中（紧急指针减1）。报文段10告诉客户进程在命令字节前还有3419个字节（从26727到30145）在服务器进程的发送缓存中等待发送。

报文段10采用紧急通知方式发送，包含了服务器进程向客户进程发送的下一个字节（序号是26727）。它不包含“清仓输出”命令字节。记得在22.2节中曾经介绍过，发送进程可以发送一个字节的数据来试探对方的接收窗口是否关闭。报文段10就是采用了这个原理。然后客户进程TCP就立即发送如报文段11所示的数据。虽然此时接收窗口还是0，但是在客户进程内部，由于客户进程的TCP收到了对方的紧急通知，它把该通知告诉客户进程，客户进程就知道服务器进程已经进入了紧急方式了。

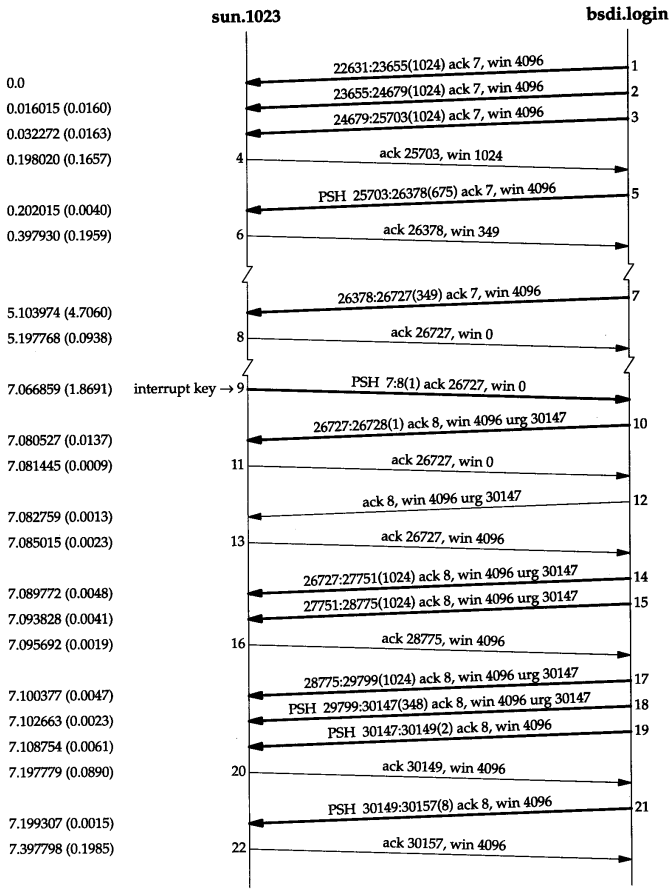


图26-7 Rlogin举例：当客户程停止输出然后终止服务器的应用程序的情况

当Rlogin客户进程从它的TCP收到了紧急通知，并且客户进程开始读取已经在输入缓存中等待被读取的数据时，接收窗口就会重新打开（报文段 13）。然后服务器进程就开始正常发送数据（报文段14, 15, 17和18）。注意报文段18的数据报中包含紧急数据的最后一个字节的数据（序号30146），该字节包含服务器进程发送给客户进程的命令字节。当客户进程收到该命令后，它就丢弃报文段14、15、17和18所收到的数据，并且清空终端的输出缓存。在报文段 19中的下两个字节是中断键的回显“ ^? ”。最后一个报文段（21）包含了客户进程的外壳提示符。

这个例子描述了当用户键入中断键后，连接的双方数据如何被存储的情况。如果这些动作仅仅丢弃在服务器的 3419个字节数据，而不丢弃已经在客户的 4096个字节的数据，那么这些已经在客户的终端输出缓存中的 4096字节数据将输出到终端上。

26.4 Telnet协议

Telnet协议可以工作在任何主机（例如，任何操作系统）或任何终端之间。 RFC 854 [Postel 和Reynolds 1983a]定义了该协议的规范，其中还定义了一种通用字符终端叫做网络虚拟终端NVT（Network Virtual Terminal）。NVT是虚拟设备，连接的双方，即客户机和服务器，都必须把它们的物理终端和 NVT进行相互转换。也就是说，不管客户进程终端是什么类型，操作系统必须把它转换为 NVT格式。同时，不管服务器进程的终端是什么类型，操作系统必须能够把NVT格式转换为终端所能够支持的格式。

NVT是带有键盘和打印机的字符设备。用户击键产生的数据被发送到服务器进程，服务器进程回送的响应则输出到打印机上。默认情况下，用户击键产生的数据是发送到打印机上的，但是我们可以看到这个选项是可以改变的。

26.4.1 NVT ASCII

术语NVT ASCII代表7比特的ASCII字符集，网间网协议族都使用 NVT ASCII。每个7比特的字符都以8比特格式发送，最高位比特为 0。

行结束符以两个字符 CR（回车）和紧接着的 LF（换行）这样的序列表示。以 \r\n来表示。单独的一个 CR也是以两个字符序列来表示，它们是 CR和紧接着的 NUL（字节 0），以 \r\0表示。

在下面的章节中可以看到， FTP, SMTP, Finger和Whois协议都以NVT ASCII来描述客户命令和服务器的响应。

26.4.2 Telnet命令

Telnet通信的两个方向都采用带内信令方式。字节 0xff（十进制的 255）叫做 IAC（interpret as command，意思是“ 作为命令来解释 ”）。该字节后面的一个字节才是命令字节。如果要发送数据255，就必须发送两个连续的字节 255（在前面一节中我们讲到数据流是 NVT ASCII，它们都是 7bit的格式，这就暗示着 255这个数据字节不能在 Telnet上传输。其实在Telnet中有一个二进制选项，在 RFC856[Postel和Reynolds 1983b]中有定义，关于这点我们没有讨论，该选项允许数据以 8bit进行传输）。图26-8列出了所有的Telnet命令。

由于这些命令中很多命令很少用到，所以对于一些重要的命令，如果在下面章节的例子或叙述中遇到，我们再做解释。

名称	代码(十进制)	描 述
EOF	236	文件结束符
SUSP	237	挂起当前进程（作业控制）
ABORT	238	异常中止进程
EOR	239	记录结束符
SE	240	子选项结束
NOP	241	无操作
DM	242	数据标记
BRK	243	中断
IP	244	中断进程
AO	245	异常中止输出
AYT	246	对方是否还在运行？
EC	247	转义字符
EL	248	删除行
GA	249	继续进行
SB	250	子选项开始
WILL	251	选项协商（图26-9）
WONT	252	选项协商
DO	253	选项协商
DONT	254	选项协商
IAC	255	数据字节255

图26-8 当前一个字节是IAC（255）时的Telnet命令集

26.4.3 选项协商

虽然我们可以认为Telnet连接的双方都是NVT，但是实际上Telnet连接双方首先进行交互的信息是选项协商数据。选项协商是对称的，也就是说任何一方都可以主动发送选项协商请求给对方。

对于任何给定的选项，连接的任何一方都可以发送下面4种请求的任意一个请求。

1) WILL：发送方本身将激活(enable)选项。

2) DO：发送方想叫接收端激活选项。

3) WONT：发送方本身想禁止选项。

4) DON'T：发送方想让接收端去禁止选项。

	发送方		接收方	描 述
1.	WILL	→		发送方想激活选项
		←	DO	接收方说同意
2.	WILL	→		发送方想激活选项
		←	DONT	接收方说不同意
3.	DO	→		发送方想让接收方激活选项
		←	WILL	接收方说同意
4.	DO	→		发送方想让接收方激活选项
	WONT	←		接收方说不同意
5.	WONT	→		发送方想禁止选项
		←	DONT	接收方必须说同意
6.	DONT	→		发送方想让接收方禁止选项
		←	WONT	接收方必须说同意

图26-9 Telnet选项协商的6种情况

由于Telnet规则规定，对于激活选项请求（如1和2），有权同意或者不同意。而对于使选项失效请求（如3和4），必须同意。这样，4种请求就会组合出6种情况，如图26-9所示。

选项协商需要3个字节：一个IAC字节，接着一个字节是WILL, DO, WONT和DONT这四

者之一，最后一个ID字节指明激活或禁止选项。现在，有40多个选项是可以协商的。Assigned Number RFC文档中指明选项字节的值，并且一些相关的RFC文档描述了这些选项。图26-10显示了在本章中会出现的选项代码。

Telnet的选项协商机制和Telnet协议的大部分内容一样，是对称的。连接的双方都可以发起选项协商请求。但我们知道，远程登录不是对称的应用。客户进程完成某些任务，而服务器进程则完成其他一些任务。下面我们将看到，某些Telnet选项仅仅适合于客户进程（例如要求激活行模式方式），某些选项则仅仅适合于服务器进程。

选项标识(十进制)	名　称	RFC
1	回显	857
3	抑制继续进行	858
5	状态	859
6	定时标记	860
24	终端类型	1091
31	窗口大小	1073
32	终端速率	1079
33	远程流量控制	1372
34	行方式	1184
36	环境变量	1408

图26-10　本章中将讨论的Telnet选项代码

26.4.4　子选项协商

有些选项不是仅仅用“激活”或“禁止”就能够表达的。指定终端类型就是一个例子，客户进程必须发送用一个ASCII字符串来表示终端类型。为了处理这种选项，我们必须定义子选项协商机制。

在RFC1091[VanBokkelen 1989]中定义了如何表示终端类型这样的子选项协商机制。首先连接的某一方（通常是客户进程）发送3个字节的字符序列来请求激活该选项。

<IAC, WILL, 24>

这里的24（十进制）是终端类型选项的ID号。如果收端（通常是服务器进程）同意，那么响应数据是：

<IAC, DO, 24>

然后服务器进程再发送如下的字符串：

<IAC, SB, 24, 1, IAC, SE>

该字符串询问客户进程的终端类型。其中SB是子选项协商的起始命令标志。下一个字节的“24”代表这是终端类型选项的子选项（通常SB后面的选项值就是子选项所要提交的内容）。下一个字节的“1”表示“发送你的终端类型”。子选项协商的结束命令标志也是IAC，就像SB是起始命令标志一样。如果终端类型是ibmpc，客户进程的响应命令将是：

＜IAC, SB, 24, 0‘I’, ‘B’, ‘M’, ‘P’, ‘C’, IAC, SE＞

第4个字节“0”代表“我的终端类型是”（在Assigned Numbers RFC文档中有正式的关于终端类型的数值定义，但是最起码在Unix系统之间，终端类型可以用任何对方可理解的数据进行表示。只要这些数据在termcap或terminfo数据库中有定义）。在Telnet子选项协商过程中，终端类型用大写表示，当服务器收到该字符串后会自动转换为小写字符。

26.4.5　半双工、一次一字符、一次一行或行方式

对于大多数Telnet的服务器进程和客户进程，共有4种操作方式。

1. 半双工

这是Telnet的默认方式，但现在却很少使用。NVT默认是一个半双工设备，在接收用户输入之前，它必须从服务器进程获得GO AHEAD（GA）命令。用户的输入在本地回显，方向是从NVT键盘到NVT打印机，所以客户进程到服务器进程只能发送整行的数据。

虽然该方式适用于所有类型的终端设备，但是它不能充分发挥目前大量使用的支持全双工通信的终端功能。RFC 857 [Postel 和Reynolds 1983c]定义了ECHO选项，RFC 858 [Postel和Reynolds 1983d]定义了SUPPRESS GO AHEAD（抑制继续进行）选项。如果联合使用这两个选项，就可以支持下面将讨论的方式：带远程回显的一次一个字符的方式。

2. 一次一个字符方式

这和前面的Rlogin工作方式类似。我们所键入的每个字符都单独发送到服务器进程。服务器进程回显大多数的字符，除非服务器进程端的应用程序去掉了回显功能。

该方式的缺点也是显而易见的。当网络速度很慢，而且网络流量比较大的时候，那么回显的速度也会很慢。虽然如此，但目前大多数Telnet实现都把这种方式作为默认方式。

我们将看到，如果要进入这种方式，只要激活服务器进程的SUPPRESS GO AHEAD选项即可。这可以通过由客户进程发送DO SUPPRESS GO AHEAD（请求激活服务器进程的选项）请求完成，也可以通过服务器进程给客户进程发送WILL SUPPRESS GO AHEAD（服务器进程激活选项）请求来完成。服务器进程通常还会跟着发送WILL ECHO，以使回显功能有效。

3. 一次一行方式

该方式通常叫做准行方式（kludge line mode），该方式的实现是遵照RFC 858的。该RFC规定：如果要实现带远程回显的一次一个字符方式，ECHO选项和SUPPRESS GO AHEAD选项必须同时有效。准行方式采用这种方式来表示当两个选项的其中之一无效时，Telnet就是工作在一次一行方式。在下节中我们将介绍一个例子，可以看到如何协商进入该方式，并且当程序需要接收每个击键时如何使该方式失效。

4. 行方式

我们用这个术语代表实行方式选项，这是在RFC 1184[Borman 1990]中定义的。这个选项也是通过客户进程和服务器进程进行协商而确定的，它纠正了准行方式的所有缺陷。目前比较新的Telnet实现支持这种方式。

图26-11是不同的Telnet客户进程和服务器进程之间默认的操作方式。“char”表示一次一个字符方式，“kludge”表示准行方式，“linemode”表示如RFC 1184定义的实行方式。

客户端	服务器端					
	SunOS 4.1.3	Solaris 2.2	SVR4	AIX 3.2.2	BSD/386	4.4BSD
SunOS 4.1.3	char	char	char	char	kludge	kludge
Solaris 2.2	char	char	char	char	kludge	kludge
SVR4	char	char	char	char	kludge	kludge
AIX 3.2.2	char	char	char	char	kludge	kludge
BSD/386	char	char	char	char	linemode	linemode
4.4BSD	char	char	char	char	linemode	linemode

图26-11 不同的Telnet客户进程和服务器进程之间默认的操作方式

从图中可以看出，只有当客户进程和服务器进程都是BSD/386或4.4BSD的时候才支持实行方式。当服务器进程的操作系统是这两者之一时，如果客户进程不支持实行方式，才会协商进入准行方式。从图中还可以看出，其实任何类型的客户进程和服务器进程都支持准行方式，但是一般都不把它作为默认方式，除非服务器进程指定。

26.4.6 同步信号

Telnet以Data Mark命令（即图26-8中的DM）作为同步信号，该同步信号是以TCP紧急数据形式发送的。DM命令是随数据流传输的同步标志，它告诉收端回到正常的处理过程上来。Telnet的双方都可以发送该命令。

当一端收到对方已经进入了紧急方式的通知后，它将开始读数据流，一边读一边丢弃所读的数据，直到读到Telnet命令为止。紧急数据的最后一个字节就是DM字节。采用TCP紧急方式的原因就是：即使TCP数据流已经被TCP流量控制所终止，Telnet命令也可以在连接上传输。

在下节中我们将看到Telnet同步信号的使用情况。

26.4.7 客户的转义符

和Rlogin的客户进程一样，Telnet客户进程也可以使用户直接和客户进程进行交互，而不是被发送到服务器进程。通常客户的转义字符是Control_]（control键和右中括号键，通常以“^]”表示）。这使得客户进程显示它的提示符，通常是“telnet>”。这时候有很多命令可供用户选择，以改变连接的特性或打印某些信息。大多数的Unix客户进程提供“help”命令，该命令将显示所有可用的命令。

在下节中我们将看到客户进程转义的例子，以及此时可以输入的命令。

26.5 Telnet举例

在这里我们将介绍在三种不同的操作方式下Telnet选项协商的情况。这些方式包括：单字符方式、实行方式和准行方式。同样我们还将讨论当用户在服务器端按了中断键退出了一个正在运行的进程后，系统的运行情况。

26.5.1 单字符方式

首先介绍基本的单字符方式，该方式类似于Rlogin。用户在终端输入的每个字符都将由终端发送到服务器进程，服务器进程的响应也将以字符方式回显到终端上。在这里运行的是一个新的客户进程BSD/386，它试图激活很多新的选项，服务器进程还是运行老的SVR4，我们将看到很多选项被服务器拒绝。

为了看到服务器和客户机之间选项协商的内容，我们将激活客户进程的一个选项来显示所有的选项协商。同样我们运行tcpdump来获得数据报交换的时间次序。图26-12显示了这个交互会话。

在图中，我们已经对由SENT或RCVD开头的选项协商的每一步都进行了标注。关于每一步的解释如下：

1) 客户发起SUPPRESS GO AHEAD选项协商。由于GO AHEAD命令通常是由服务器发送给客户的，而且客户希望服务器激活该选项，因此该选项的请求方式是DO（由于激活这一选项将会禁止GA命令的发送，上述过程很容易让人混淆）。在第10行可以看到服务器进程同意该选项。

2) 客户进程要按照在RFC 1091[VanBokkelen 1989]中的定义发送终端类型。这对Unix类型的客户进程来讲是很普通的。因为客户进程要激活本地的选项，所以该选项的请求方式是WILL。

```
bsdi % telnet                                   调用客户进程，不带任何命令行选项

telnet> toggle options                          告诉客户进程显示所有的选项协商过程
Will show option processing.

telnet> open svr4                               现在和服务器建立连接
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.

SENT DO SUPPRESS GO AHEAD                       1. (后面将讨论的行序号)
SENT WILL TERMINAL TYPE                         2.
SENT WILL NAWS                                  3.
SENT WILL TSPEED                                4.
SENT WILL LFLOW                                 5.
SENT WILL LINEMODE                              6.
SENT WILL ENVIRON                               7.
SENT DO STATUS                                  8.
RCVD DO TERMINAL TYPE                           9.
RCVD WILL SUPPRESS GO AHEAD                     10.
RCVD DONT NAWS                                  11.
RCVD DONT TSPEED                                12.
RCVD DONT LFLOW                                 13.
RCVD DONT LINEMODE                              14.
RCVD DONT ENVIRON                               15.
RCVD WONT STATUS                                16.
RCVD IAC SB TERMINAL-TYPE SEND                  17.
SENT IAC SB TERMINAL-TYPE IS "IBMPC3"           18.
RCVD WILL ECHO                                  19.
SENT DO ECHO                                    20.
RCVD DO ECHO                                    21.
SENT WONT ECHO                                  22.


UNIX(r) System V Release 4.0 (svr4)

RCVD DONT ECHO                                  23.
login: rstevens                                 我们键入用户和口令，服务进程不回显这些数据，
Password:                                       然后操作系统问候输出，然后是外壳提示符
```

图26-12 Telnet双方选项协商的初始化过程

3) NAWS的意思是“协商窗口大小”，它在RFC 1073 [Waitzman]中有定义。如果服务器进程同意该选项（实际上不同意，见11行），客户进程就要发送终端窗口的行、列大小的子选项。而且只要窗口大小发生变化，客户进程随时都将向服务器进程发送这一子选项（这和图26-4中Rlogin的0x80命令类似）。

4) TSPEED选项允许发送方（通常是客户进程）发送它的终端速率，这在 RFC 1079 [Hedrick 1988b]中有定义。如果服务器进程同意（实际上不同意，见12行），客户进程将发送其发送速率和接收速率的子选项。

5) LFLOW代表“本地流量控制”，这在RFC1371 [Hedrick 和Borman 1992]中定义。客户进程给服务器进程发送该选项，表示客户进程希望用命令方式激活或禁止流量控制。如果服务器进程同意（实际上不同意，见13行），只要Control_S和Control_Q进程需要在客户进程和服务器进程进行切换，客户进程都要向服务器进程发送子选项（这类似于图 26-4中Rlogin的0x10和0x20命令）。正如在关于Rlogin的讨论中我们所提到的那样，由客户进程进行流量控制的效果比由服务器进程来完成要好。

6) LINEMODE代表在26.4中所说的实行方式。所有终端字符的处理由 Telnet客户进程完成（例如回格，删除行等），然后整行发送给服务器进程。在本节后面，我们将介绍一个例子。该选项同样被服务器进程拒绝，如14行所示。

7) ENVIRON选项允许客户进程把环境变量发送给服务器进程，这在 RFC 1408 [Borman

1993a]中有定义。这样就可以把客户进程的用户环境变量自动传播到服务器进程。在 15行，服务器进程拒绝该选项（Unix中的环境变量通常是大写字母，紧跟一个等号，然后是一个字符串值，当然这只是一个惯例而已）。默认情况下，BSD/386 Telnet客户进程发送两个环境变量：DISPLAY和PRINTER，前提是这两个变量已经定义并且有效。Telnet用户可以定义其他一些要发送的环境变量。

8) STATUS选项（RFC 859 [Postel 和Reynolds 1983e]中定义）允许连接的一方询问对方对Telnet选项目前状态的理解。在这个例子中，客户进程要求对方激活选项（DO）。如果服务器进程同意（实际上不同意，见16行），客户进程就可以要求服务器进程以子选项的形式发送它的状态值。

9) 这是服务器进程的第一个响应。服务器进程同意激活终端类型选项（几乎所有的 Unix类型的服务器进程都支持该选项）。但现在客户进程还不能立即发送它的终端类型。它必须要等到服务器进程用子选项的形式询问终端类型的时候才能够发送（17行）。

10) 服务器进程同意抑制发送GO AHEAD命令。

11) 服务器进程不同意客户进程发送它的窗口大小。

12) 服务器进程不同意客户进程发送它的终端速率。

13) 服务器进程不同意客户进程实施流量控制。

14) 服务器进程不同意客户进程激活行方式选项。

15) 服务器进程不同意客户进程发送环境变量。

16) 服务器进程不发送状态信息。

17) 这是服务器进程要求客户进程发送终端类型的子选项。

18) 客户进程把终端类型“IBMPC3”以6字节的字符串形式发送给服务器进程。

19) 服务器进程要求客户进程发起请求，要求服务器进程激活回显选项。这是本例中服务器进程第一次主动发起选项协商。

20) 客户进程同意由服务器进程实现回显功能。

21) 服务器进程要求客户进程实现回显功能。这个命令是多余的，它只是将前两行进行了交换。这是目前大多数Unix的Telnet服务器进程判断客户进程是否运行 4.2BSD或更新的BSD版本时的一个方法。如果客户进程回送 WILL ECHO，就表明客户进程运行的是老版本的4.2BSD，不支持TCP的紧急方式（在这种情况下就不能采用TCP紧急方式）。

22) 客户进程回送WONT ECHO，表示它不是一台4.2BSD主机。

23) 对于客户进程回送的WONT ECHO，服务器进程以DONT ECHO作为响应。

图26-13显示的是本例中服务器进程和客户进程交互的时间系列（去掉了连接建立部分）。

报文段1包含了图26-12中的1~8行。该报文段中包含24个字节数据，每个选项占3个字节。这是客户进程发起的选项协商。该报文段显示多个Telnet选项可以打在一个TCP段中发送。

报文段3是图26-12中的第9行，即DO TERMINAL TYPE命令。报文段5包含下面的8个选项协商中服务器进程的响应，即图 26-12中的10~17行。该报文段的长度是27个字节，因为10~16行是常规选项，每个占3个字节，而17行的子选项部分占6个字节。报文段6包含12个字节，和18行对应，这是客户发送它的终端类型的子选项。

报文段8（53个字节）包含两个Telnet命令和47字节的输出数据。前面的两个服务器进程发送Telnet命令占6字节，：WILL ECHO和DO ECHO（19和21行）。后47个字节的数据是：

```
\r\n\r\nUNIX(r) System V Release 4.0 (svr4) \r\n\r\0\r\n\r\0
```

前面4个字节数据在字符串输出之前产生两个空行。两字节的字符序列“ \r\n ”在Telnet中被认为是换行命令，而两字节的字符序列“ \r\0 ”则被认为是回车命令。这表明数据和命令可以在一个数据段中传输。 Telnet服务器进程和客户进程必须扫描接收到的每个字符，寻找 IAC 字节并执行它后续的命令。

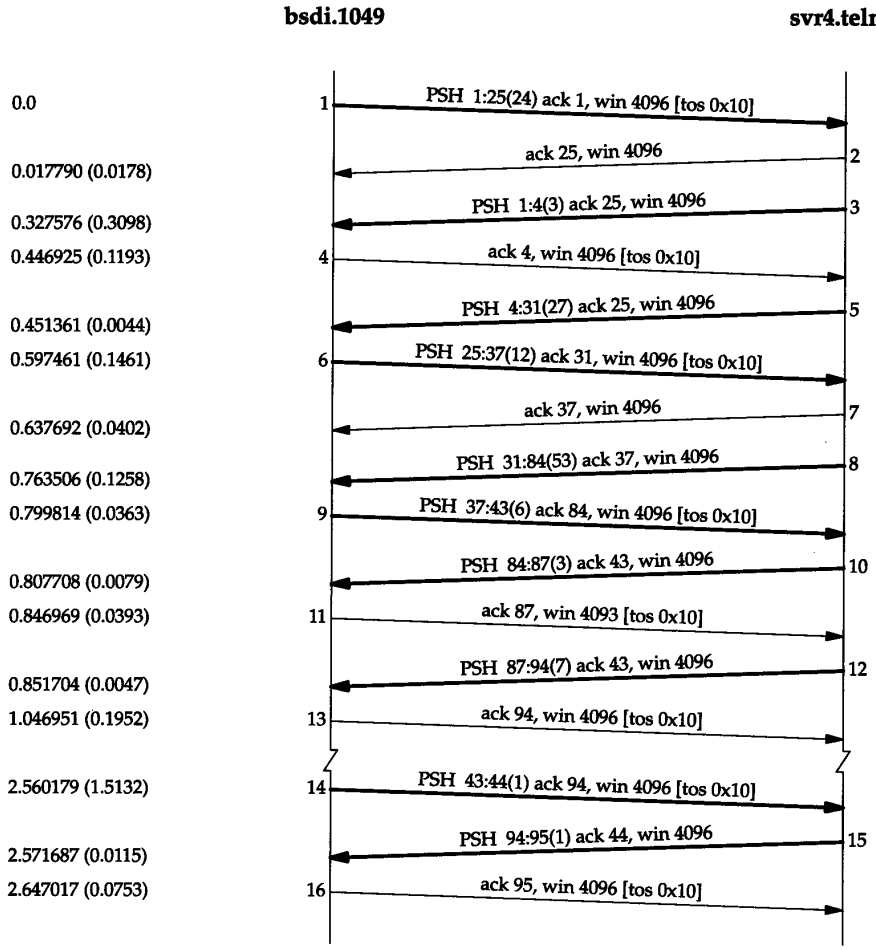


图26-13 Telnet服务器进程和客户进程选项协商初始化过程

报文段9包含客户进程发送的最后两个选项： 20和22行。23行是报文段10的响应，也是服务器进程发送的最后一个选项数据。

从现在开始，双方就可以交互数据了。当然在交互数据的过程中还可以进行选项协商，我们在该例子中就不多介绍了。报文段 12是服务器进程发送的提示符“ login: ”。报文段14是用户输入的登录用户名的第一个字符，它的回显在报文段 15中。这和我们在 19.2节中介绍的Rlogin交互类似：客户进程每次发送一个字符，服务器进程完成回显工作。

图26-12中的选项协商由客户进程初始化的，但是在本书中我们已经介绍了用Telnet客户进程连接某些标准服务器进程如：日间（daytime）服务器、回显(echo)服务器等情况。当然我们介绍这些的目的是为了描述 TCP的各种特性。但考察这些例子中

的分组交换，如图18-1，我们并没有看到客户进程发起的选项协商。为什么？这是因为在Unix系统中，除非使用标准的Telnet端口号23，否则客户进程不进行选项协商。这个特性使得Telnet客户进程可以使用标准的NVT同其他一些非Telnet服务器进程交换数据。我们已经在日间服务器、回显服务器和丢弃(discard)服务器中使用了这个特性，在后面章节介绍FTP和SMTP服务器的时候我们还将使用该特性。

26.5.2 行方式

为了描述Telnet的行方式选项协商过程，我们在主机 bsdi运行客户进程，服务器是位于`vangogh.vs.berkeley.edu`节点运行4.4BSD操作系统的一台主机。BSD/386和4.4BSD都支持这个选项。

我们不详细讨论所有的报文、选项和子选项协商过程，因为这个过程和前面的例子类似，而且对于行方式选项我们已经论述得比较清楚。下面我们仅仅讨论在选项协商中的一些区别。

1) 对于BSD/386希望协商的选项例如：窗口大小、本地流量控制、状态、环境变量和终端速率等，4.4BSD服务器进程都支持。

2) 4.4BSD服务器进程将协商一个 BSD/386客户进程不支持的新选项：鉴别（为避免以明文形式在网络上传输用户口令）。

3) 和上个例子一样，客户进程发送 WILL LINEMODE选项，由于服务器进程支持该选项，所以服务器进程发送DO LINEMODE。此时客户进程以子选项形式给服务器进程发送16个特定字符。这些字符是能影响客户进程的特定终端字符值：如中断字符，文件结束符等。服务器进程给客户进程发送一个子选项，让客户进程处理所有的输入，执行所有的编辑功能（删除字符，删除行等）。客户进程把除控制字符以外的字符以行的形式发送给服务器进程。服务器进程还要求客户进程把所有中断键和信号键转换为相应的 Telnet字符。例如中断键是Control_C，我们可以按Control_C来中断服务器端的某个进程。客户进程必须把Control_C转换为Telnet的IP命令（<IAC, IP>）传输给服务器进程。

4) 当用户输入口令时情况也有所不同。在 Rlogin和一次一字符方式的 Telnet中，都是由服务器进程负责回显，所以当服务器进程读到口令时，它并不回显这些字符。但在行方式中由客户进程负责回显。下面这些交互过程将处理这种情况：

   (a) 服务器进程发送WILL ECHO，以告诉客户进程：服务器进程将处理回显。

   (b) 客户进程回送DO ECHO。

   (c) 服务器进程向客户进程发送字符串`Password:`，客户进程把它发送到终端上。

   (d) 然后用户输入口令，当用户按下 RETURN键的时候，客户进程把口令发送给服务器进程。此时口令不回显，因为客户进程认为服务器进程将回显它。

   (e) 由于口令的结束符 RETURN没有回显，服务器进程发送两字节字符序列 CR和LF以移动光标。

   (f) 服务器进程发送WONT ECHO。

   (g) 客户进程回送DONT ECHO。然后继续由客户进程负责回显。

一旦登录完成，客户进程将把数据以整行的方式发送给服务器进程。这就是行方式选项的目的。行方式大大地减少了客户进程和服务器进程之间的数据交互数量，而且对于用户的

击键（也就是回显和编辑）提供更快的响应。图 26-14显示的是当我们输入命令时，在行方式连接下分组交换的情况。

```
Vangogh %date
```

（去掉了业务种类信息和窗口通告信息）。

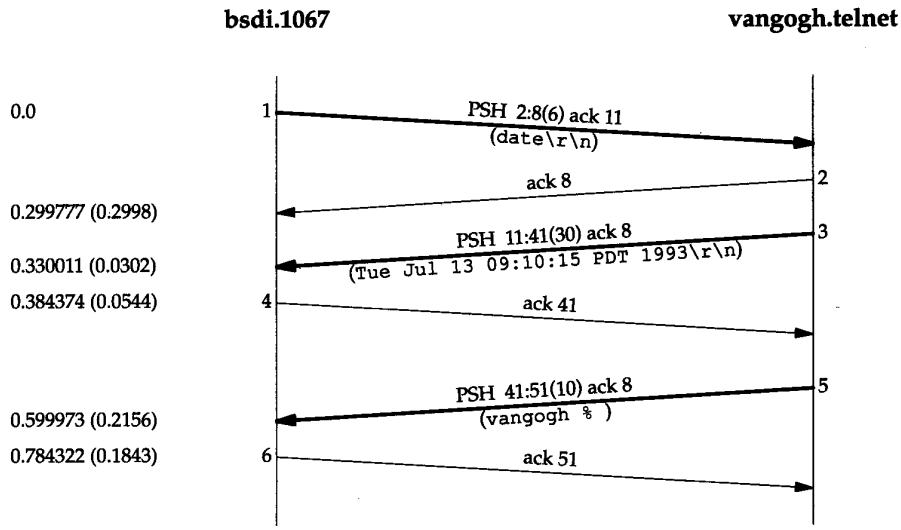


图26-14 Telnet行方式下客户进程向服务器进程发送命令的情况

把它和在Rlogin中输入同样命令（图 19-2）时的情况进行一下比较。我们看到在 Telnet行方式下只需要2个报文段（一个包含数据，另一个用于 ACK，连同IP和TCP首部共86字节），而在Rlogin中要发送15个报文段（5个有键入的数据， 5个有回显的数据， 5个是ACK，共611字节）。可见节省的数据量是非常可观的。

如果在服务器端运行一个需要进入单个字符方式的应用程序（例如 vi编辑器）会怎么样呢？实际上将发生如下的一些交互：

1) 当服务器的应用程序启动了，并改变其伪终端方式时， Telnet服务器进程被通告需要进入单个字符方式。然后服务器发送 WILL ECHO命令和行方式子选项，以告知客户不要再以行方式工作，转而进入单个字符方式。

2) 客户响应以DO ECHO，并确认行方式子选项。

3) 应用程序在服务器上运行。我们键入的每个字符将发送到服务器（当然要强制使用Nagle算法），此时服务器将处理必要的回显工作。

4) 当应用程序终止时，就恢复其伪终端方式，并通告 Telnet服务器。服务器将向客户发送WONT ECHO命令，同时发送行方式子选项，告诉客户恢复进入行方式。

5) 客户响应DONT ECHO，确认进入行方式。

上述情况同我们键入口令之间的区别表明：回显功能和单个字符方式与一次一行方式没有依赖关系。当我们键入口令时，回显功能必须失效，但一次一行方式有效。对于一个全屏应用来讲，例如编辑器，回显必须失效而单个字符方式必须有效。

图26-15概括了Rlogin和Telnet不同方式之间的差异。

应用程序	客户进程发送		客户进程回显？	例　子
	一次一字符	一次一行		
Rlogin	•		否	
Telnet	•		否	
Telnet，行方式		•	是	正常命令
Telnet，行方式		•	否	键入我们的口令
Telnet，行方式	•		否	vi编辑器

图26-15　Rlogin和不同方式的Telnet之间的比较

26.5.3　一次一行方式（准行方式）

从图26-11可以看出，如果客户不支持行方式，那么较新的服务器支持行方式选项，它也将转入准行方式(Kludge line mode)。我们同时指出所有的客户进程和服务器进程都支持准行方式，但它不是默认方式，必须由客户进程或用户特地激活它。让我们来看看如何用 Telnet选项激活准行方式。

首先介绍当客户进程不支持行方式时， BSD/386服务器进程如何协商进入该方式。

1) 当客户进程不同意服务器进程激活行方式的请求时，服务器进程发送　DO TIMING MARK选项。RFC 860 [Postel 和Reynolds 1983f]定义了这个Telnet选项。它的作用是让收发双方同步，关于这个问题将在本节的后面讲到用户键入中断键时讨论。该选项只是用来判断客户进程是否支持准行方式。

2) 客户响应WILL TIMING MARK，表明支持准行方式。

3) 服务器发送WONT SUPPRESS GO AHEAD和WONT ECHO选项，告诉客户它希望禁止这两个选项。我们在前面已经强调：单个字符方式下是假定　SUPPRESS GO AHEAD和ECHO选项同时有效的，所以禁止两个选项就进入了准行方式。

4) 客户响应DONT SUPPRESS GO AHEAD和DONT ECHO命令。

5) 服务器发送`login:`提示符，然后用户键入用户名。用户名是以整行的方式发送给服务器，回显由客户进程在本地处理。

6) 服务器发送`Password:`提示符和WILL ECHO命令。这将使客户进程的回显失效，因为此时客户进程认为服务器进程将处理回显工作，所以用户键入的口令就不回显到屏幕上。客户响应DO ECHO命令。

7) 我们键入口令。客户以整行方式发送到服务器。

8) 服务器发送WONT ECHO命令，使得客户重新激活回显功能，客户响应 DONT ECHO。

从此以后的普通命令处理过程就和行方式相似了。客户进程负责所有的编辑和回显，并以整行的方式发送给服务器进程。

在图26-11中，我们已经强调：所有标注为“ char ”的记录都支持准行方式，只不过默认是单个字符方式罢了。如果要客户进入行方式，我们就能很容易看到选项协商的过程：

```
svr4 %                                          客户是sun，服务器是svr 4
                                                键入Control]以和Telnet客户进程通信(无回显)
telnet> status                                  检验现在是否在一次一字符方式下
Connected to svr4.tuc.noao.edu
Operating in character-at-a-time mode.
Escape character is '^]'.
```

```
telnet> toggle options                    注意选项协商过程
Will show option processing.

telnet> mode line                         切换到准行方式
SENT dont SUPPRESS GO AHEAD               客户发送这两个选项
SENT dont ECHO
RCVD wont SUPPRESS GO AHEAD               服务器把WONT做为上述两个选项协商的响应
RCVD wont ECHO
```

这将使Telnet会话进入准行方式，此时SUPPRESS GO AHEAD和ECHO选项都是失效的。

如果在服务器端运行如vi编辑器这样的应用程序，同样会有行方式下遇到的问题。当要运行这样的应用程序时，服务器进程必须告诉客户进程从准行方式切换到单字符方式。当应用程序结束时，必须告诉客户进程返回到准行方式。下面是这个过程需要用到的技术要点。

1) 当应用程序改变其伪终端方式并通知服务器进程时，服务器进程将进入单字符方式。服务器进程向客户进程发送WILL SUPPRESS GO AHEAD和WILL ECHO，这将使客户进程进入单字符方式。

2) 客户进程回送DO SUPPRESS GO AHEAD和WILL ECHO。

3) 应用程序开始在服务器端运行。

4) 当应用程序结束并改变其伪终端方式时，服务器进程发送 WONT SUPPRESS GO AHEAD和WONT ECHO命令，使得客户进程返回准行方式。

5) 客户进程回送DONT SUPPRESS GO AHEAD和DONT ECHO命令，告诉服务器进程它已经回到了准行方式。

图26-16概括了单个字符方式及准行方式中不同的SUPPRESS GO AHEAD和ECHO选项设置。

方 式	SUPPRESS GO AHEAD	ECHO	举 例
一次一字符	√	√	准行方式下的vi编辑器
准行方式	×	×	正常命令
准行方式	×	√	键入我们的口令

图26-16 准行方式下Telnet选项的设置

26.5.4 行方式：客户中断键

看一下当用户键入中断键时 Telnet将发生什么情况。假定在客户主机 bsdi和服务器cangogh.cs.berkeley.edu之间建立了一个Telnet会话。图26-17显示了当用户键入中断键后的时间系列（去掉了窗口通告和服务类型）。

报文段1中显示的是中断键（通常是Control_C或DELETE）已经转换为Telnet的IP（中断进程）命令：<IAC, IP>。下面的3个字节：<IAC, DO, TM>，组成了Telnet的DO TIMING MARK选项。这个标志由客户进程发送，必须使用WILL或WONT响应。所有在响应前收到的数据都要丢弃（除非是Telnet命令）。这是服务器进程和客户机端的同步过程。报文段1没有采用TCP紧急方式。

Host Requirements RFC叙述了IP命令不能使用Telnet的同步信号来发送。如果可以的话，那么<IAC, IP>的后面将跟随<IAC, DM>，同时紧急指针指向DM字节。大多数的Unix Telnet 客户有一个选项来使用同步信号发送IP命令，但是这个选项默认是不用的（正如我们这里看到的）。

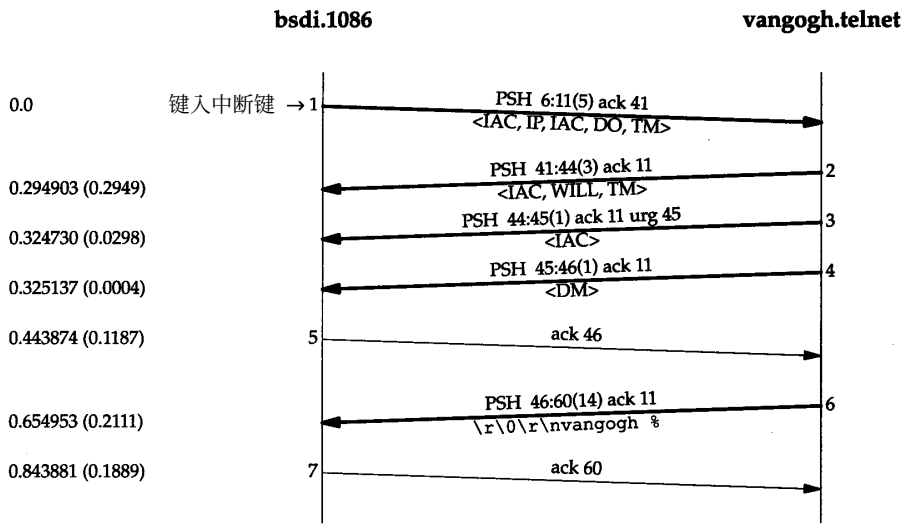


图26-17 行方式下键入中断键后的情况

报文段2是服务器进程对 DO TIMING MARK选项的响应。紧随其后的是报文段 3和4中Telnet的同步信号：<IAC, DM>。报文段3中的紧急指针指向将在报文段4中发送的DM字节。

如果服务器进程到客户进程的窗口已满，那么客户进程发送了如报文段 1中的IP命令后就丢弃收到的所有数据。即使服务器进程被 TCP流量控制所终止而不能发送如报文段 2、3和4中的数据，紧急指针仍然可以发送。这和图 26-7中的Rlogin类似。

为什么同步信号要分为两个数据段发送（3和4）？原因就是我们在20.8节中详细讨论TCP紧急指针时提到的情况。有关主机需求的 RFC中提到紧急指针应指向紧急数据的最后一个字节，而很多衍生于伯克利的系统中，紧急指针指向紧急数据的倒数第 2个字节（回忆一下在图26-6中，紧急指针指向命令字节的前一个字节）。Telnet服务器进程故意把同步信号的第 1个字节作为紧急数据，它知道紧急指针将指向下 1个字节（即DM字节），而IAC字节和紧急指针必须立即发送，在下一步才发送DM字节。

最后一个报文段6发送的是数据，它是服务器进程发生的提示符。

26.6 小结

本章我们介绍了 Rlogin和Telnet操作。两者都提供了从客户进程远程登录到服务器进程，是我们能够在服务器端运行程序的方法。

这两个应用是不同的。 Rlogin假定连接的双方都是 Unix系统，所以只提供一个选项，它是1个简单的协议。Telnet则不同，它用于在不同类型的主机之间建立连接。

为了支持这种多机环境， Telnet提供客户进程和服务器进程的选项协商机制。如果连接的双方都支持这些选项，则可以增强一些功能。对于比较简单的客户进程和服务器进程，它可以提供Telnet的基本功能，而当双方都支持某些选项时，它又可以充分利用双方的新特性。

我们介绍了Telnet的选项协商机制，也介绍了 3种数据传输的方式：单字符方式、准行方式和实行方式。现在的趋势是只要有可能，就尽量工作在准行方式下。这样可以减少网络上的数据量，同时为交互用户提供更好的行编辑和回显的响应。

图26-18概括并比较了Rlogin和Telnet的不同特性。

特　征	Rlogin	Telnet
运输协议	一个TCP连接。使用紧急方式	一个TCP连接。使用紧急方式。
分组方式	总是一次一字符，远程回显	通常的默认是一次一字符，远程回显。带客户回显的准行方式也支持带回显的实行模式。当服务器上的应用进程请求时，总是一次一字符的方式
流量控制	通常由客户完成，可以被服务器禁止	通常由服务器完成，选项允许客户来完成
终端类型	总是提供	选项，通常被支持
终端速率	总是提供	选项
窗口大小	大多数服务器支持此选项	选项
环境变量	不支持	选项
自动登录	默认。提示用户键入口令，口令以明文发送。较新的版本支持Kerberos方式的登录	默认是键入登录名和口令。口令以明文发送。较新的版本支持鉴别选项

图26-18　Rlogin和Telnet的不同特性

Rlogin服务器和Telnet服务器通常都将设置TCP的保活选项以检测客户主机是否崩溃（如果服务器的TCP实现支持，见第23章）。这两种应用都采用了TCP紧急方式，以便即使从服务器到客户的数据传输被流量控制所终止，服务器仍然可以向客户发送命令。

习题

26.1　在图26-5中，标出所有延迟的ACK。

26.2　在图26-7中，为什么要发送报文段12？

26.3　我们说过Rlogin客户进程必须使用保留端口号（见　1.9节）（通常Rlogin客户使用512~1023之间的保留端口）。这会给主机带来什么限制？有没有解决的办法？

26.4　阅读RFC 1097，它描述了Telnet的阈下报文(subliminal-message)选项。

第27章 FTP：文件传送协议

27.1 引言

FTP是另一个常见的应用程序。它是用于文件传输的 Internet标准。我们必须分清文件传送（file transfer）和文件存取 (file access) 之间的区别，前者是 FTP提供的，后者是如 NFS（Sun的网络文件系统，第 29章）等应用系统提供的。由 FTP提供的文件传送是将一个完整的文件从一个系统复制到另一个系统中。要使用 FTP，就需要有登录服务器的注册帐号，或者通过允许匿名FTP的服务器来使用（本章我们将给出这样的一个例子）。

与Telnet类似，FTP最早的设计是用于两台不同的主机，这两个主机可能运行在不同的操作系统下、使用不同的文件结构、并可能使用不同字符集。但不同的是，Telnet获得异构性是强制两端都采用同一个标准：使用7比特ASCII码的NVT。而FTP是采用另一种方法来处理不同系统间的差异。FTP支持有限数量的文件类型（ASCII，二进制，等等）和文件结构（面向字节流或记录）。

参考文献959 [Postel 和 Reynolds 1985] 是FTP的正式规范。该文献叙述了近年来文件传输的历史演变。

27.2 FTP协议

FTP与我们已描述的另一种应用不同，它采用两个 TCP连接来传输一个文件。

1) 控制连接以通常的客户服务器方式建立。服务器以被动方式打开众所周知的用于FTP的端口（21），等待客户的连接。客户则以主动方式打开 TCP端口21，来建立连接。控制连接始终等待客户与服务器之间的通信。该连接将命令从客户传给服务器，并传回服务器的应答。

   由于命令通常是由用户键入的，所以IP对控制连接的服务类型就是“最大限度地减小迟延”。

2) 每当一个文件在客户与服务器之间传输时，就创建一个数据连接。（其他时间也可以创建，后面我们将说到）。

   由于该连接用于传输目的，所以IP对数据连接的服务特点就是“最大限度提高吞吐量”。

图27-1描述了客户与服务器以及它们之间的连接情况

从图中可以看出，交互式用户通常不处理在控制连接中转换的命令和应答。这些细节均由两个协议解释器来完成。标有“用户接口”的方框功能是按用户所需提供各种交互界面（全屏幕菜单选择，逐行输入命令，等等），并把它们转换成在控制连接上发送的 FTP命令。类似地，从控制连接上传回的服务器应答也被转换成用户所需的交互格式。

从图中还可以看出，正是这两个协议解释器根据需要激活文件传送功能。

27.2.1 数据表示

FTP协议规范提供了控制文件传送与存储的多种选择。在以下四个方面中每一个方面都必须作出一个选择。

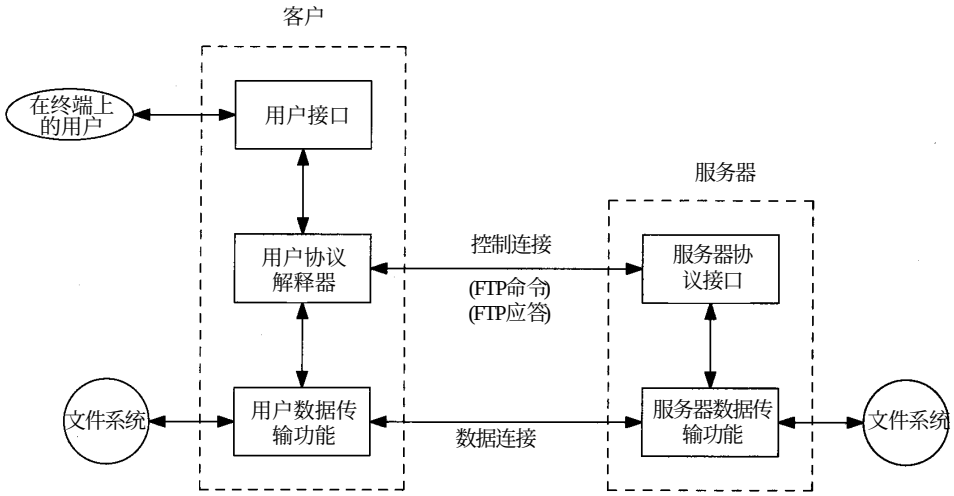


图27-1 文件传输中的处理过程

1. 文件类型

(a) ASCII码文件类型 （默认选择）文本文件以NVT ASCII码形式在数据连接中传输。这要求发方将本地文本文件转换成NVT ASCII码形式，而收方则将NVT ASCII码再还原成本地文本文件。其中，用NVT ASCII码传输的每行都带有一个回车，而后是一个换行。这意味着收方必须扫描每个字节，查找CR、LF对（我们在第15.2节见过的关于TFIP的ASCII码文件传输情况与此相同）。

(b) EBCDIC文件类型 该文本文件传输方式要求两端都是EBCDIC系统。

(c) 图像文件类型（也称为二进制文件类型） 数据发送呈现为一个连续的比特流。通常用于传输二进制文件。

(d) 本地文件类型 该方式在具有不同字节大小的主机间传输二进制文件。每一字节的比特数由发方规定。对使用8 bit字节的系统来说，本地文件以8 bit字节传输就等同于图像文件传输。

2. 格式控制

该选项只对ASCII和EBCDIC文件类型有效。

(a) 非打印 （默认选择）文件中不含有垂直格式信息。

(b) 远程登录格式控制 文件含有向打印机解释的远程登录垂直格式控制。

(c) Fortran 回车控制 每行首字符是Fortran格式控制符。

3. 结构

（a） 文件结构 （默认选择）文件被认为是一个连续的字节流。不存在内部的文件结构。

（b） 记录结构 该结构只用于文本文件（ASCII或EBCDIC）。

（c） 页结构 每页都带有页号发送，以便收方能随机地存储各页。该结构由 TOPS-20操作系统提供（主机需求RFC不提倡采用该结构）。

4. 传输方式

它规定文件在数据连接中如何传输。

（a） 流方式 （默认选择）文件以字节流的形式传输。对于文件结构，发方在文件尾提示关闭数据连接。对于记录结构，有专用的两字节序列码标志记录结束和文件结束。

（b） 块方式 文件以一系列块来传输，每块前面都带有一个或多个首部字节。

（c） 压缩方式 一个简单的全长编码压缩方法，压缩连续出现的相同字节。在文本文件

中常用来压缩空白串，在二进制文件中常用来压缩 0字节（这种方式很少使用，也不受支持。现在有一些更好的文件压缩方法来支持 FTP）。

如果算一下所有这些选择的排列组合数，那么对传输和存储一个文件来说就有 72种不同的方式。幸运的是，其中很多选择不是废弃了，就是不为多数实现环境所支持，所以我们可以忽略掉它们。

通常由Unix实现的FTP 客户和服务器把我们的选择限制如下：

- 类型：ASCII或图像。
- 格式控制：只允许非打印。
- 结构：只允许文件结构。
- 传输方式：只允许流方式。

这就限制我们只能取一、两种方式： ASCII或图像（二进制）。

该实现满足主机需求 RFC的最小需求（该 RFC也要求能支持记录结构，但只有操作系统支持它才行，而Unix不行）。

很多非Unix的实现提供了处理它们自己文件格式的 FTP功能。主机需求RFC指出“ FTP协议有很多特征，虽然其中一些通常不实现，但对 FTP中的每一个特征来说，都存在着至少一种实现 ”。

27.2.2 FTP命令

命令和应答在客户和服务器的控制连接上以 NVT ASCII码形式传送。这就要求在每行结尾都要返回CR、LF对（也就是每个命令或每个应答）。

从客户发向服务器的 Telnet命令（以IAC打头）只有中断进程（ <IAC,IP>）和Telnet的同步信号（紧急方式下 <IAC,DM>）。我们将看到这两条 Telnet命令被用来中止正在进行的文件传输，或在传输过程中查询服务器。另外，如果服务器接受了客户端的一个带选项的 Telnet命令（WILL，WONT，DO或DONT），它将以DONT 或WONT响应。

这些命令都是3或4个字节的大写ASCII字符，其中一些带选项参数。从客户向服务器发送的FTP命令超过30种。图27-2给出了一些常用命令，其中大部分将在本章再次遇到。

命　令	说　明
ABOR	放弃先前的FTP命令和数据传输
LIST *filelist*	列表显示文件或目录
PASS *password*	服务器上的口令
PORT *n1,n2,n3,n4,n5,n6*	客户端IP地址（*n1.n2.n3.n4*）和端口（*n5*×256+*n6*）
QUIT	从服务器注销
RETR *filename*	检索（取）一个文件
STOR *filename*	存储（放）一个文件
SYST	服务器返回系统类型
TYPE *type*	说明文件类型：A表示ASCII码，I表示图像
USER *username*	服务器上用户名

图27-2　常用的FTP命令

下节我们将通过一些例子看到，在用户交互类型和控制连接上传送的 FTP命令之间有时是一对一的。但也有些操作下，一个用户命令产生控制连接上多个 FTP命令。

27.2.3 FTP应答

应答都是ASCII码形式的3位数字，并跟有报文选项。其原因是软件系统需要根据数字代码来决定如何应答，而选项串是面向人工处理的。由于客户通常都要输出数字应答和报文串，一个可交互的用户可以通过阅读报文串（而不必记忆所有数字回答代码的含义）来确定应答的含义。

应答3位码中每一位数字都有不同的含义（我们将在第 28章看到简单邮件传送输协议，SMTP，使用相同的命令和应答约定）。

图27-3给出了应答代码第1位和第2位的含义。

应答	说　　明
1yz	肯定预备应答。它仅仅是在发送另一个命令前期待另一个应答时启动
2yz	肯定完成应答。一个新命令可以发送
3yz	肯定中介应答。该命令已被接受，但另一个命令必须被发送
4yz	暂态否定完成应答。请求的动作没有发生，但差错状态是暂时的，所以命令可以过后再发
5yz	永久性否定完成应答。命令不被接受，并且不再重试
x0z	语法错误
x1z	信息
x2z	连接。应答指控制或数据连接
x3z	鉴别和记帐。应答用于注册或记帐命令
x4z	未指明
x5z	文件系统状态

图27-3 应答代码3位数字中第1位和第2位的含义

第3位数字给出差错报文的附加含义。例如，这里是一些典型的应答，都带有一个可能的报文串。

- 125 数据连接已经打开；传输开始。
- 200 就绪命令。
- 214 帮助报文（面向用户）。
- 331 用户名就绪，要求输入口令。
- 425 不能打开数据连接。
- 452 错写文件。
- 500 语法错误（未认可的命令）。
- 501 语法错误（无效参数）。
- 502 未实现的MODE(方式命令)类型。

通常每个FTP命令都产生一行回答。例如， QUIT命令可以产生如下应答：

```
221 Goodbye.
```

如果需要产生一条多行应答，第1行在3位数字应答代码之后包含一个连字号，而不是空格，最后一行包含相同的3位数字应答代码，后跟一个空格符。例如，HELP命令可以产生如下应答：

```
214- The following commands are recognized (* =>'s unimplemented).
   USER    PORT    STOR    MSAM*   RNTO    NLST    MKD     CDUP
   PASS    PASV    APPE    MRSQ*   ABOR    SITE    XMKD    XCUP
   ACCT*   TYPE    MLFL*   MRCP*   DELE    SYST    RMD     STOU
   SMNT*   STRU    MAIL*   ALLO    CWD     STAT    XRMD    SIZE
```

```
   REIN*    MODE     MSND*    REST     XCWD     HELP     PWD      MDTM
   QUIT     RETR     MSOM*    RNFR     LIST     NOOP     XPWD
214 Direct comments to ftp-bugs@bsdi.tuc.noao.edu.
```

27.2.4 连接管理

数据连接有以下三大用途：

1) 从客户向服务器发送一个文件。

2) 从服务器向客户发送一个文件。

3) 从服务器向客户发送文件或目录列表。

FTP服务器把文件列表从数据连接上发回，而不是控制连接上的多行应答。这就避免了行的有限性对目录大小的限制，而且更易于客户将目录列表以文件形式保存，而不是把列表显示在终端上。

我们已说过，控制连接一直保持到客户-服务器连接的全过程，但数据连接可以根据需要随时来，随时走。那么需要怎样为数据连接选端口号，以及谁来负责主动打开和被动打开？

首先，我们前面说过通用传输方式（Unix环境下唯一的传输方式）是流方式，并且文件结尾是以关闭数据连接为标志。这意味着对每一个文件传输或目录列表来说都要建立一个全新的数据连接。其一般过程如下：

1) 正由于是客户发出命令要求建立数据连接，所以数据连接是在客户的控制下建立的。

2) 客户通常在客户端主机上为所在数据连接端选择一个临时端口号。客户从该端口发布一个被动的打开。

3) 客户使用PORT命令从控制连接上把端口号发向服务器。

4) 服务器在控制连接上接收端口号，并向客户端主机上的端口发布一个主动的打开。服务器的数据连接端一直使用端口20。

图27-4给出了第3步执行时的连接状态。假设客户用于控制连接的临时端口是1173，客户用于数据连接的临时端口是1174。客户发出的命令是PORT命令，其参数是6个ASCII中的十进制数字，它们之间由逗点隔开。前面4个数字指明客户上的IP地址，服务器将向它发出主动打开（本例中是140.252.13.34），而后两位指明16 bit端口地址。由于16 bit端口地址是从这两个数字中得来，所以其值在本例中就是4×256+150 = 1174。

图27-5给出了服务器向客户所在数据连接端发布主动打开时的连接状态。服务器的端点是端口20。

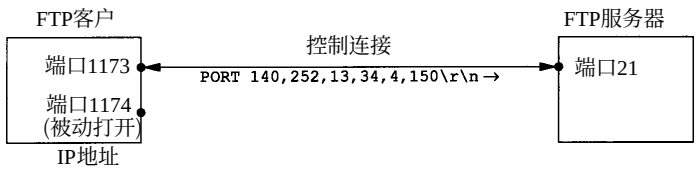


图27-4 在FTP控制连接上通过的PORT命令

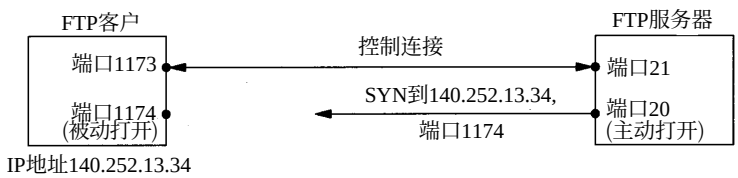


图27-5 主动打开数据连接的FTP服务器

服务器总是执行数据连接的主动打开。通常服务器也执行数据连接的主动关闭，除非当客户向服务器发送流形式的文件时，需要客户来关闭连接（它给服务器一个文件结束的通知）。

客户也有可能不发出PORT命令，而由服务器向正被客户使用的同一个端口号发出主动打开，来结束控制连接。这是可行的，因为服务器面向这两个连接的端口号是不同的：一个是20，另一个是21。不过，下节我们将看到为什么现有实现通常不这样做。

27.3 FTP的例子

现在看一些使用FTP的例子：它对数据连接的管理，采用NVT ASCII码的文本文件如何发送，FTP使用Telnet同步信号来中止进行中的文件传输，最后是常用的“匿名FTP”。

27.3.1 连接管理：临时数据端口

先看一下FTP的连接管理，它只在服务器上用简单FTP会话显示一个文件。我们用-d标志（debug）来运行svr4主机上的客户。这告诉它要打印控制连接上变换的命令和应答。所有前面冠以--->的行是从客户上发向服务器的，所有以3位数字开头的行都是服务器的应答。客户的交互提示是ftp>。

```
svr4 %ftp -d bsdi                              -d 选项用作排错输出
Connected to bsdi.                             客户执行控制连接的主动打开
220 bsdi FTP server (Version 5.60) ready       服务器响应就绪
Name (bsdi:rstevens):                          客户提示我们输入
---> USER rstevens                             键入RETURN，客户发送默认信息
331 Password required for rstevens.
Password:                                      键入口令；它不需要回显
---> PASS XXXXXXXX                                  客户以明文发送它
230 User rstevens logged in.
ftp> dir hello.c                               要求列出一个文件的目录
---> PORT 140,252,13,34,4,150                       见图27-4
200 PORT Command successful.
---> LIST hello.c
150 Opening ASCII mode data connection for /bin/ls.
-rw-r--r--   1 rstevens  staff  38 Jul 17 12:47 hello.c
226 Transfer complete.
remote: hello.c                                客户输出
56 bytes received in 0.03 seconds (1.8 Kbytes/s)
ftp> quit                                      我们已完成
---> QUIT
221 Goodbye
```

当FTP客户提示我们注册姓名时，它打印了默认值（我们在客户上的注册名）。当我们敲RETURN键时，默认值被发送出去。

对一个文件列出目录的要求引发一个数据连接的建立和使用。本例体现了我们在图27-4和图27-5中给出的程序。客户要求TCP为其数据连接的终端提供一个临时端口号，并用PORT命令发送这个端口号（1174）给服务器。我们也看到一个交互用户命令（dir）成为两个FTP命令（PORT和LIST）。

图27-6是控制连接上分组交换的时间系列（已除去了控制连接的建立和结束，以及所有窗口大小的通知）。我们关注该图中数据连接在哪儿被打开、使用和过后的关闭。

图27-7是数据连接的时间系列。图中的起始时间与图 27-6中的相同。已除去了所有窗口大小通知，但留下服务类型字段，以说明数据连接使用另一个服务类型（最大吞吐量），而不同于控制连接（最小时延）（服务类型(TOS)值在图3-2中）。

在时间系列上，FTP服务器执行数据连接的主动打开，从端口 20（称为ftp-data）到来自PORT命令的端口号（1174）。本例中还可以看到服务器在哪儿向数据连接上执行写操作，服务器对数据连接执行主动的关闭，这就告诉客户列表已完成。

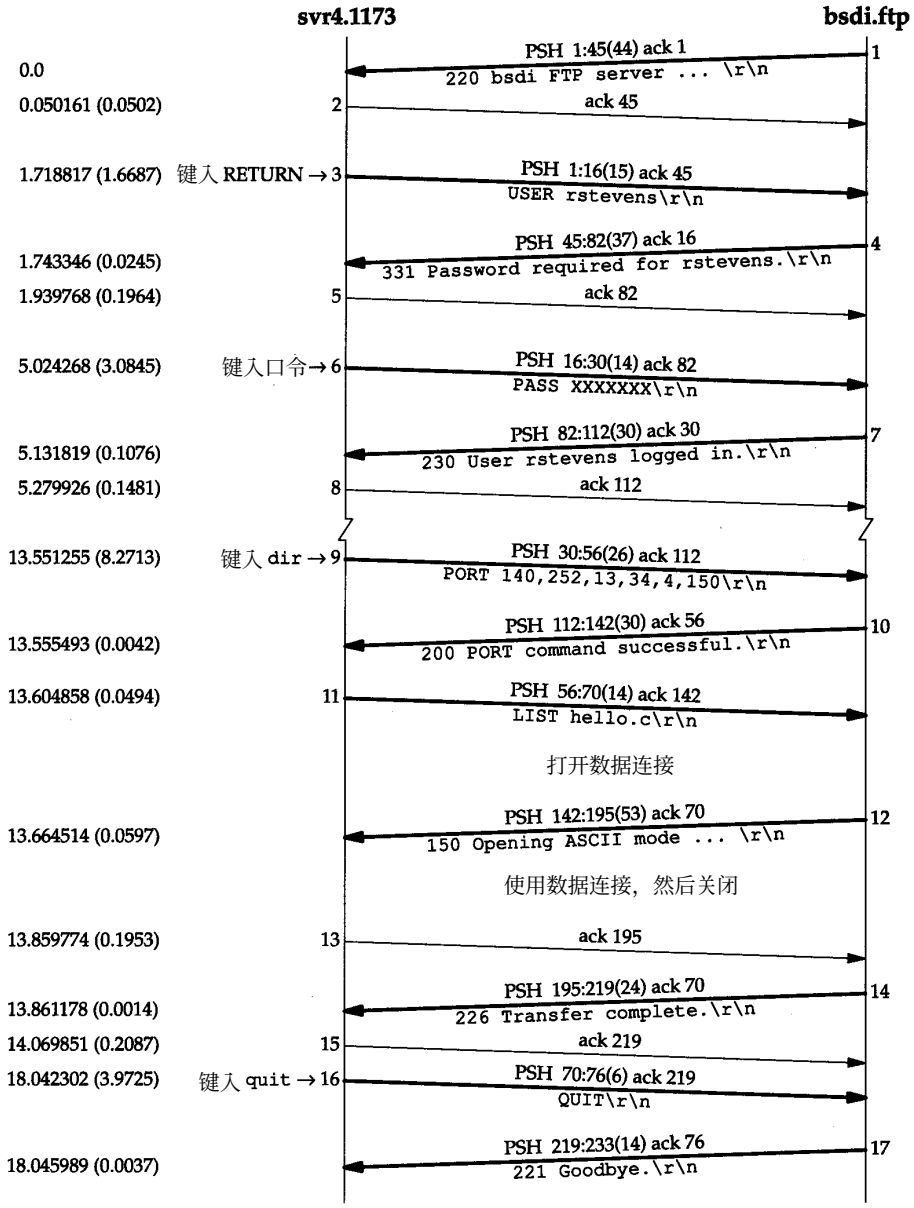


图27-6 FTP控制连接示例

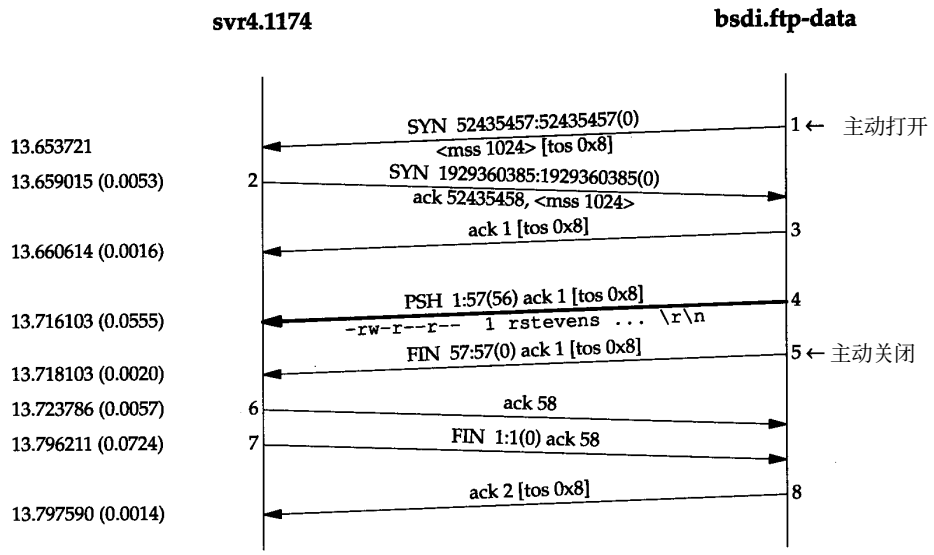


图27-7 FTP数据连接示例

27.3.2 连接管理：默认数据端口

如果客户没有向服务器发出PORT命令，来指明客户数据连接端的端口号，服务器就用与控制连接正在用的相同的端口号给数据连接。这会给使用流方式（ Unix FTP客户和服务器一直使用）的客户带来一些问题。正如下面所示：

> Host Requirements RFC建议使用流方式的FTP客户在每次使用数据连接前发一个PORT命令来启用一个非默认的端口号。

回到先前的例子（图27-6），如果我们要求在列出第1个目录后几秒钟再列出另一个目录，那该怎么办？客户将要求其内核选择另一个临时端口号（可能是 1175），下一个数据连接将建立在svr4端口1175和bsdi端口20之间。但在图27-7中服务器执行数据连接的主动打开，我们在18.6节说明了服务器将不把端口 20分配给新的数据连接，这是因为本地端口号已被更早的连接使用，而且还处于2MSL等待状态。

服务器通过指明我们在18.6节中提到的SO_REUSEADDR选项，来解决这个问题。这让它把端口20分配给新连接，而新连接将从处于 2MSL等待状态的端口（1174）处得到一个不一样的外部端口号（1175），这样一切都解决了。

如果客户不发送PORT命令，而在客户上指明一个临时端口号，那么情况将改变。我们可以通过执行用户命令sendport给FTP来使之发生。Unix FTP客户用这个命令在每个数据连接使用之前关闭向服务器发送PORT命令。

图27-8给出了用于两个连续LIST命令的数据连接时间系列。控制连接起自主机 svr4上的端口1176，所以在没有PORT命令的情况下，客户和服务器给数据连接使用相同的端口号（除去了窗口通知和服务类型值）。

事件序列如下：

1) 控制连接是建立在客户端口1176到服务器端口21上的（这里我们不展示）。

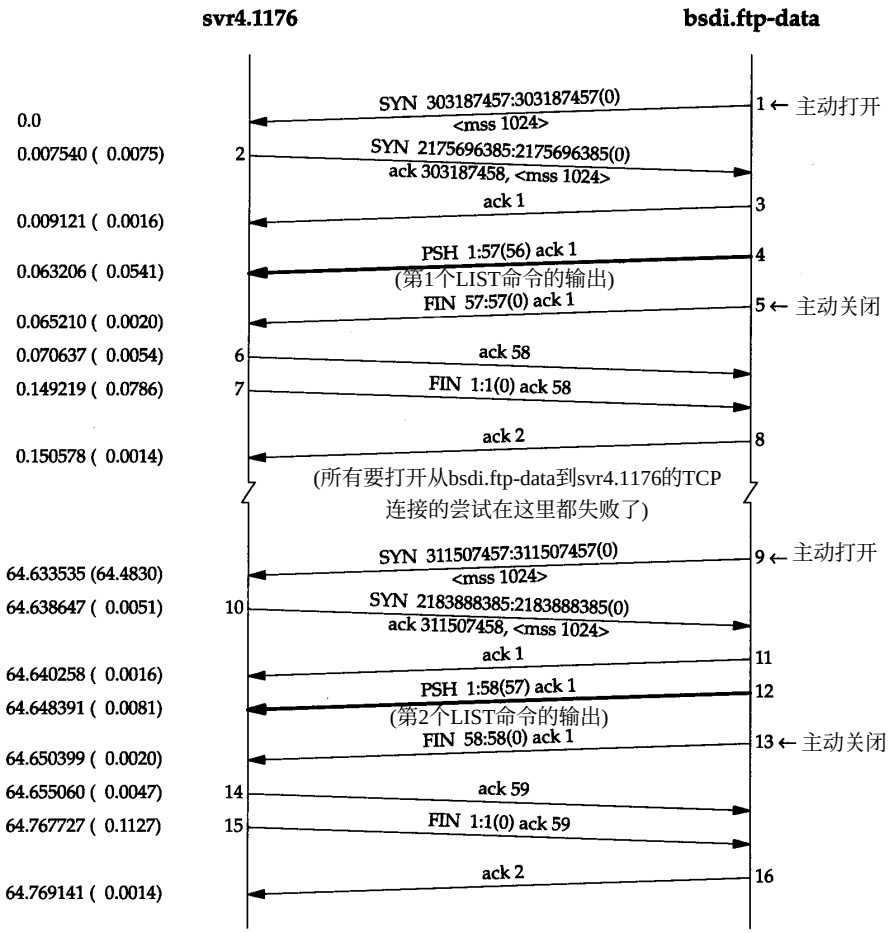


图27-8 两个连续LIST命令的数据连接

2) 当客户为端口1176上的数据连接做被动打开时，由于该端口已被客户上的控制连接使用，所以必须确定SO_REUSEADDR选项。

3) 服务器给端口20到端口1176的数据连接（报文段1）做主动打开。即便端口1176已在客户上被使用，客户仍会接受它（报文段2），这是因为下面这一对插口是不同的：

```
<svr4, 1176, bsdi, 21>
<svr4, 1176, bsdi, 20>
```

（在bsdi上的端口号是不同的）。TCP通过查看源IP地址、源端口号、目的IP地址、目的端口号分用各呼入报文段，只要这4个元素中的一个不同，就行。

4) 服务器对数据连接（报文段 5）做主动的关闭，即把这对插口置入服务器上的一个2MSL等待。

```
<svr4, 1176, bsdi, 20>
```

5) 客户在控制连接上发送另一个 LIST命令（这里我们不展示）。在此之前，客户在端口1176上为其数据连接端做一个被动打开。客户必须再一次指明 SO_REUSEADDR，这是因为端口号1176已在使用。

6) 服务器给从端口20到端口1176的数据连接发出一个主动打开。在此之前，服务器必须指明SO_REUSEADDR，这是因为本地端口（20）与处于2MSL等待状态的连接是相关联的，但从18.6节所示可知，该连接将不成功。其原由是这个连接用插口对 (socket pair)与步骤4中的仍处于2MSL等待状态的插口对相同。TCP规定禁止服务器发送同步信息（SYN）。这样就没办法让服务器跨过插口对的2MSL等待状态来重用相同的插口对。

在这一步伯克利软件分发（BSD）服务器每隔5秒就重试一次连接请求，直到满18次，总共90秒。我们看到报文段9将在大约1分钟后成功（我们在第18章提到过，SVR4使用一个30秒的MSL，以两个MSL来达到持续1分钟的等待）。我们没看到在这个时间系列上的这些失败有任何同步（SYN）信息，这是因为主动打开失败，服务器的TCP不再发送一个SYN。

Host Requirements RFC建议使用PORT命令的原因是在两个相继使用数据连接之间避免出现这个2MSL。通过不停地改变某一端的端口号，我们所说的这个问题就不会出现。

27.3.3 文本文件传输：NVT ASCII表示还是图像表示

让我们查证一下默认的文本文件传输使用 NVT ASCII码。这次不指定 -d标志，所以不看客户命令，但注意到客户还将打印服务器的响应：

```
sun % ftp bsdi
Connected to bsdi.
220 bsdi FTP server (Version 5.60) ready.
Name (bsdi:retevens);                                    键入RETURN
331 Password required for rstevens.
Passord :                                                键入口令
230 User rstevens logged in.
ftp> get hello.c                                         取一个文件
200 PORT command successful.
150 Opening ASCII mode data connection for hello.c (38 bytes).
226 Transfer complete.                                   服务器说明文件含有38字节
local: hello.c remote: hello.c                           由客户输出
42 bytes received in 0.0037 seconds (11 Kbytes/s)        字节传过数据连接
ftp> quit
221 Goodbye.
Sun % ls -l hello.c
-rw-rw-r--   1 rstevens     38 Jul 18   08:48 hello.c   但文件还含有38字节
sun % wc -l hello.c                                      在文件中记行数
    4 hello.c
```

因为文件有4行，所以从数据连接上传输42个字节。Unix下的每一新行符（\n）被服务器转换成NVT ASCII码的2字节行结尾序列（\r\n）来传输，然后再由客户转换成原先形式来存储。

新客户试图确定服务器是否是相同类型的系统，一旦相同，就可以用二进制码（图像文件类型）来传输文件，而不是ASCII码。这可以获得两个方面的好处：

1) 发方和收方不必查看每一字节（很大的节约）。

2) 如果主机操作系统使用比2字节的NVT ASCII码序列更少的字节来作行尾，就会传输更少的字节数（很小的节约）。

我们可以看到使用一个BSD/386客户和服务器的最优效果。启动排错（ debug）方式来看

客户FTP命令：

```
bsdi &ftp -d slip                                   指明 -d来看客户命令
Connected to slip.
220 slip FTP server (Version 5.60) ready.
Name (slip:rstevens):                               我们键入RETURN
---> USER rstevns
331 Password required for rstevens.
Password :                                          我们键入自己的口令
---> PASS XXXX
230 User rstevns logged in .
---> SYST                                           这由客户服务器的应答自动发送
215 UNIX Type: L8 Version : BSD-199103
Remote system type is UNIX.                         由客户发出的信息
Using binary mode to transfer files.                由客户发出的信息
ftp> get hello.c                                    取一个文件
---> TYPE I                                         由客户自动发送
200 Type set to I.
---> PORT  140,252,13,66,4,84                       端口号=4×256+84=1108
200 PORT command successful.
---> RETR hello.c
150 Opening BINARY mode data connection for hello.c (38 bytes) .
226 Transefer complete .
38 bytes received in 0.035 seconds (1.1 Kbytes/这时只有38个字节
ftp> quit
---> QUIT
221 Goodbye.
```

注册到服务器后，客户FTP自动发出SYST命令，服务器将用自己的系统类型来响应。如果应答起自字符串“ 215 UNIX Type : L8 ”，并且如果客户在每字节为8 bit的Unix系统上运行，那么二进制方式（图像）将被所有文件传输所使用，除非被用户改变。

当我们取文件hello.c时，客户自动发出命令TYPE I把文件类型定成图像。这样在数据连接上只有38字节被传输。

> Host Requirements RFC指出一个FTP服务器必须支持SYST命令（这曾是RFC 959中的一个选项）。但支持它的使用文本的系统（见封 2）仅仅是BSD/386和AIX 3.2.2。SunOS 4.1.3和Solaris 2.x 用500（不能理解的命令）来应答。SVR4采用极不大众化的应答行为500，并关闭控制连接！

27.3.4 异常中止一个文件的传输：Telnet 同步信号

现在看一下FTP客户是怎样异常中止一个来自服务器的文件传输。异常中止从客户传向服务器的文件很容易 —— 只要客户停止在数据连接上发送数据，并发出 ABOR命令到控制连接上的服务器即可。而异常中止接收就复杂多了，这是因为客户要告知服务器立即停止发送数据。我们前面提到要使用Telnet同步信号，下面的例子就是这样。

我们先发起一个接收，并在它开始后键入中断键。这里是交互会话，其中初始注册被略去：

```
ftp> get a.out                                  取一个大文件
---> TYPE I                                     客户和服务器都是8 bit字节的Unix系统
200 Type set to I.
---> PORT 140,252,13,66,4,99
200 PORT command successful.
---> RETR a.out
150 Opening BINARY mode data connection for a.out (28672 bytes).
^?                                              键入的中断键
receive aborted                                 由客户输出
waiting for remote to finish abort              由客户输出
426 Transfer aborted. Data connection closed.
226 Abort successful
1536 bytes received in 1.7 seconds (0.89 Kbytes/s)
```

在我们键入中断键之后，客户立即告知我们它将发起异常中止，并正在等待服务器完成。服务器发出两个应答：426和226。这两个应答都是由Unix 服务器在收到来自客户的紧急数据和ABOR命令时发出的。

图27-9和图27-10展示了会话时间系列。我们已把控制连接（实线）和数据连接（虚线）合在一起来说明它们之间的关系。

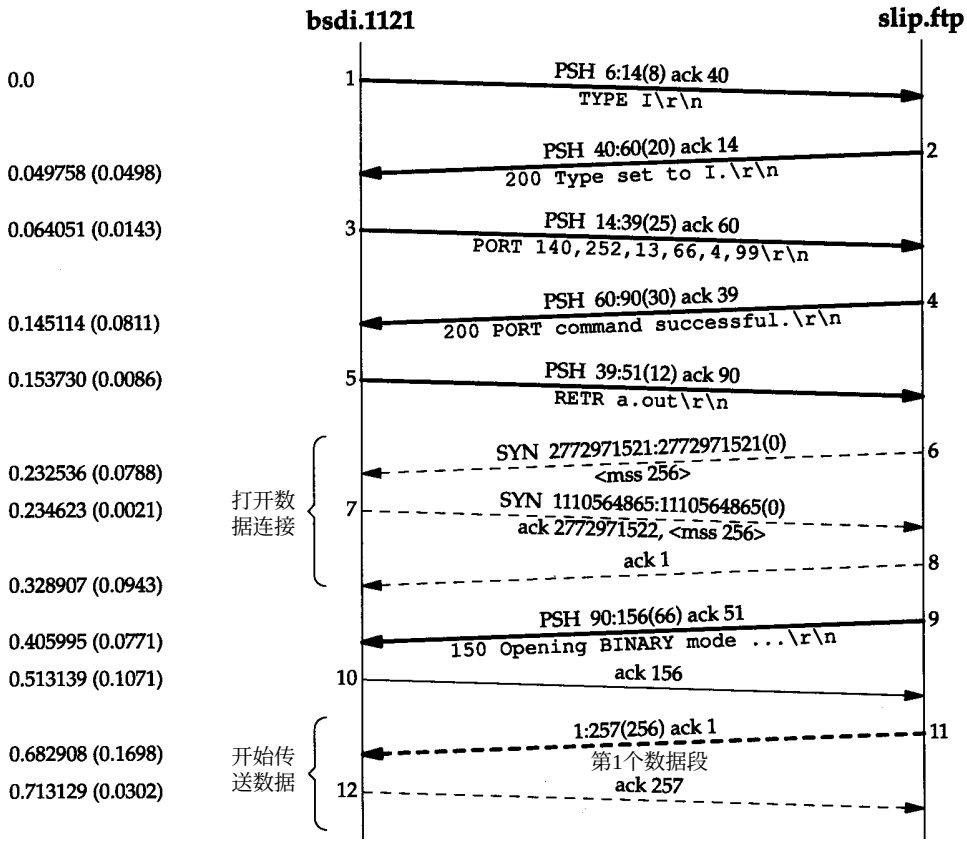


图27-9 异常中止一个文件的传输（前半部）

图27-9的前面12个报文段是我们所期望的。通过控制连接的命令和应答建立起文件传输，数据连接被打开，第1个报文段的数据从服务器发往客户。

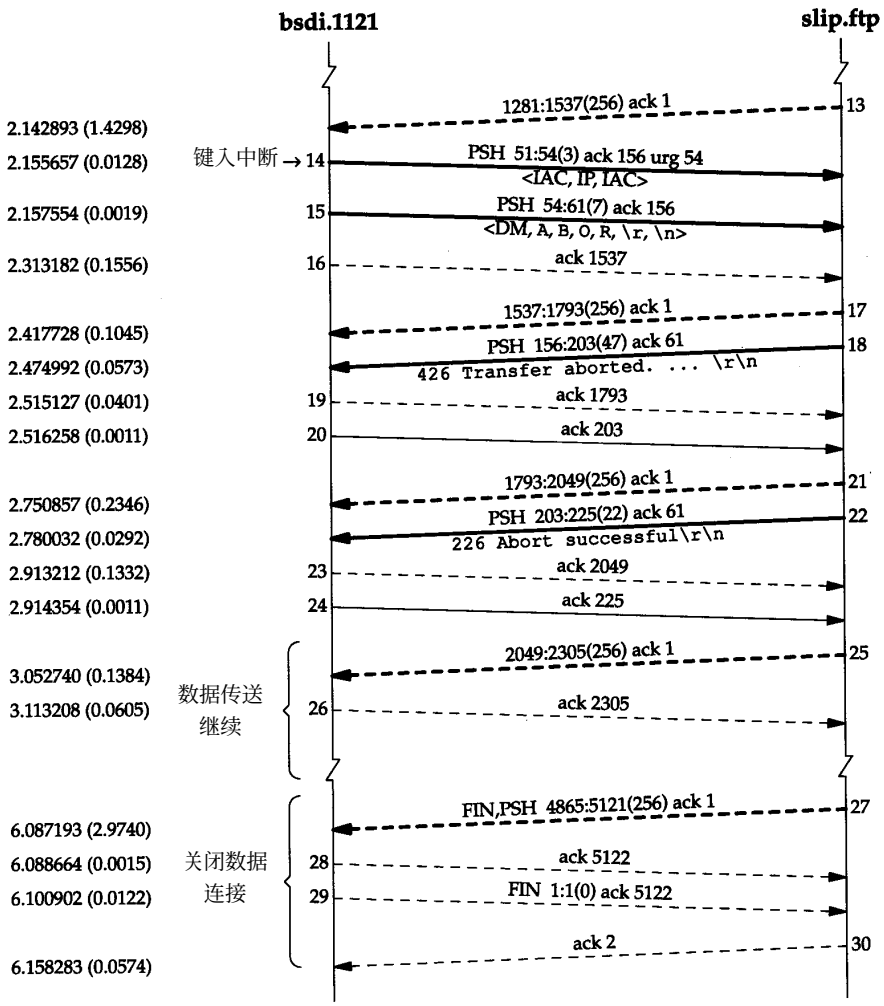


图27-10 异常中止一个文件的传输（后半部）

在图27-10中，报文段13是数据连接上来自服务器的第 6个数据报文段，后跟由我们键入的中断键产生的报文段14。客户发出10个字节来异常中止传输：

```
<IAC,IP,IAC,DM,A,B,O,R,\r,\n>
```

由于20.8节中详细讨论过这个问题，我们看到有两个报文段（ 14和15）涉及到TCP的紧急指针（我们在图26-17中看过对Telnet问题也做相同的处理）。Host Requirements RFC指出紧急指针应指向紧急数据的最后一个字节，而多数伯克利的派生实现使之指向紧急数据最后一个字节后面的第一个字节。了解到紧急指针将（错误地）指向下一个要写的字节（数据标志，DM。在序号为54处），FTP客户进程特意写前3个字节作为紧急数据。首先写下的 3字节紧急数据与紧急指针一起被立即发送，紧接着是后面 7个字节（BSD FTP 服务器不会出现由客户使用的紧急指针的解释问题。当服务器收到控制连接上的紧急数据时，它读下一个 FTP命令，寻找ABOR或STAT，忽略嵌入的Telnet命令）。

注意到尽管服务器指出传输被异常中止（报文段 18，在控制连接上），客户进程还要在数据连接上再接收14个报文段的数据（序列号是 1537~5120）。这些报文段可能在收到异常中止

时，还在服务器上的网络设备驱动器中排队，但客户打印“收到 1536字节”，意思是在发出异常中止后（报文段14和15），略去收到的所有数据报文段。

一旦Telnet用户键入中断键，我们在图 26-17中看到Unix客户在默认情况下不发出中断进程命令作为紧急数据。因为几乎没有机会用流控制来中止从客户进程到服务器进程的数据流，所以我们说这样就行了。FTP的客户进程也通过控制连接发送一个中断进程命令，因为两个连接正在被使用，因此没有机会用流控制来中止控制连接。为什么 FTP发送中断进程命令作为紧急数据而Telnet不呢？答案在于FTP使用两个连接，而 Telnet只使用一个，在某些操作系统上要求一个进程同时监控两个连接的输入是困难的。 FTP假设这些临界的操作系统至少提供紧急数据在控制连接上已到达的通知，而后让服务器从处理数据连接切换到控制连接上来。

27.3.5 匿名FTP

FTP的一种形式很常用，我们下面给出它的例子。它被称为匿名 FTP，当有服务器支持时，允许任何人注册并使用FTP来传输文件。使用这个技术可以提供大量的自由信息。

怎样找出你正在搜寻的站点是一个完全不同的问题。我们将在 30.4节简要介绍。

我们将把匿名FTP用在站点`ftp.uu.net`上（一个常用的匿名FTP站点）来取本书的勘误表文件。要使用匿名FTP，须使用“anonymous”（复习数遍就能正确地拼写）用户名来注册。当提示输入口令时，我们键入自己的电子邮箱地址。

```
sun % ftp ftp.uu.net
Connected to ftp.uu.net
220 ftp.UU.NET FTP server (Version 2.0WU(13) Fri Apr 9 20:44:32 EDT 1993) ready
Name (ftp.uu.net:rstevens): anonymous
331 Guest login ok, send your complete e-mail addraess as password.
Password :                                  键入rstevens@noao.edu；它没有回显
230-
230-                    Welcome to the UUNET archive.
230-    A service of UUNET Technologies Inc, Falls Church, Virginia
230-    For information about UUNET, call +1 703 204 8000, or see the files
230-    in /uunet-info
                                            还有一些问候行
230 Guest login ok, access restrictions apply.
ftp> cd published/books                     换成需要的目录
250 CWD command successful.
ftp> binary                                 我们将传送一个二进制文件
200 Type set to I.
ftp> get stevens.tcpipivl.errata.Z          取文件
200 PORT command successful.
150 Opening BINARY mode data connection for stevens.tcpipivl.errata.Z (150 bytes).
226 Transfer complete.                      (你可能得到一个不同的文件大小)
local: stevens.tcpipivl.errata.Z remote: stevens.tcpipivl.errata.Z
105 bytes received in 4.1 seconds (0.83 Kbytes/s)
ftp> quit
221 Goodbye.
sun % uncompress stevens.tcpipivl.errata.Z
sun % more stevens.tcpipivl.errata
```

不压缩是因为很多现行匿名FTP文件是用Unix compress(1)程序压缩的，这样导致文件带有.Z的扩展名。这些文件必须使用二进制文件类型来传输，而不是ASCII码文件类型。

27.3.6 来自一个未知IP地址的匿名FTP

可以把一些使用匿名FTP的域名系统（DNS）特征和选路特征结合在一起。在14.5节中我们谈到DNS中指针查询现象——取一个IP地址并返回其主机名。不幸的是并非所有系统管理员都能正确地创立涉及指针查询的名服务器。他们经常记得把新主机加入名字到地址匹配的文件中，却忘了把他们加入到地址到名字匹配的文件中。对此，可用traceroute经常看到这种现象，即它打印一个IP地址，而不是主机名。

有些匿名FTP服务器要求客户有一个有效域名。这就允许服务器来记录正在执行传输的主机域名。由于服务器在来自客户IP数据报中收到的关于客户的唯一标识是客户的IP地址，所以服务器能叫DNS来做指针查询，并获得客户的域名。如果负责客户主机的名服务器没有正确地创立，指针查询将失败。

要看清这个错误，我们来做以下诸步骤：

1) 把主机slip（见封2的图）的IP地址换成140.252.13.67。这是给作者子网的一个有效IP地址，但没有涉及到noao.edu域的域名服务器。

2) 把在bsdi上SLIP连接的目的IP地址换成140.252.13.67。

3) 把将数据报引向140.252.13.67的sun上的路由表入口加入路由器bsdi（回忆一下我们在9.2节中关于这个选路表的讨论）。

从Internet上仍然可以访问我们的主机slip，这是因为在10.4节中路由器gateway和netb正好把所有目的是子网140.252.13的所有数据报都发送给路由器sun。路由器sun知道利用我们在上述第3步建立的路由表入口来如何处理这些数据报。我们所创建的是拥有完整Internet连接性的主机，但没有有效的域名。结果，指针查询IP地址140.252.13.67将失败。

现在给一个我们所知的服务器使用匿名FTP，需要一个有效的域名：

```
slip % ftp ftp.uu.net
Connected to ftp.uu.net.

220 ftp.UU.NET FTP server (Version 2.0WU(13) Fri Apr 9 20:44:32 EDT 1993) ready.
Name (ftp.uu.net:rstevens): anonymous

530- Sorry, we're unable to map your IP address 140.252.13.67 to a hostname
530- in the DNS.  This is probably because your nameserver does not have a
530- PTR record for your address in its tables, or because your reverse
530- nameservers are not registered.  We refuse service to hosts whose
530- names we cannot resolve.  If this is simply because your nameserver is
530- hard to reach or slow to respond then try again in a minute or so, and
530- perhaps our nameserver will have your hostname in its cache by then.
530- If not, try reaching us from a host that is in the DNS or have your
530- system administrator fix your servers.
530 User anonymous access denied..

Login failed.
Remote system type is UNIX.
Using binary mode to transfer files.

ftp> quit
221 Goodbye.
```

来自服务器的出错应答是无需加以说明的。

27.4 小结

FTP是文件传输的Internet标准。与多数其他TCP应用不同，它在客户进程和服务器进程之间使用两个TCP连接——一个控制连接，它一直持续到客户进程与服务器进程之间的会话完成为止；另一个按需可以随时创建和撤消的数据连接。

FTP使用的关于数据连接的连接管理让我们更详细地了解TCP连接管理需求。我们看到TCP在不发出PORT命令的客户进程上对2MSL等待状态的作用。

FTP使用NVT ASCII码做跨越控制连接的所有远程登录命令和应答。数据传输的默认方式通常也是NVT ASCII码。我们看到较新的Unix客户进程会自动发送命令来查看服务器是否是8bit字节的Unix主机，并且如果是，那么就使用二进制方式来传输所有文件，那将带来更高的效率。

我们也展示了匿名FTP的一个例子，它是在Internet上分发软件的常用形式。

习题

27.1 图27-8中，如果客户对第2个数据连接做一次主动打开，而不是由服务器来做，那将发生什么变化？

27.2 在本章FTP客户例子中，我们加入诸如由客户输出行的行注释。如果不看源代码，我们如何确定这些不是来自服务器？

```
local: hello.c remote: hello.c
42 bytes received in 0.0037 seconds (11 Kbytes/s)
```

第28章 SMTP: 简单邮件传送协议

28.1 引言

电子邮件（e-mail）无疑是最流行的应用程序。[Caceres et al.1991]说明，所有TCP连接中大约一半是用于简单邮件传送协议SMTP (Simple Mail Transfer Protocol)的（以比特计算为基础，FTP连接传送更多的数据）。[Paxson 1993] 发现，平均每个邮件中包含大约1500字节的数据，但有的邮件中包含兆比特的数据，因为有时电子邮件也用于发送文件。

图28-1显示了一个用TCP/IP交换电子邮件的示意图。

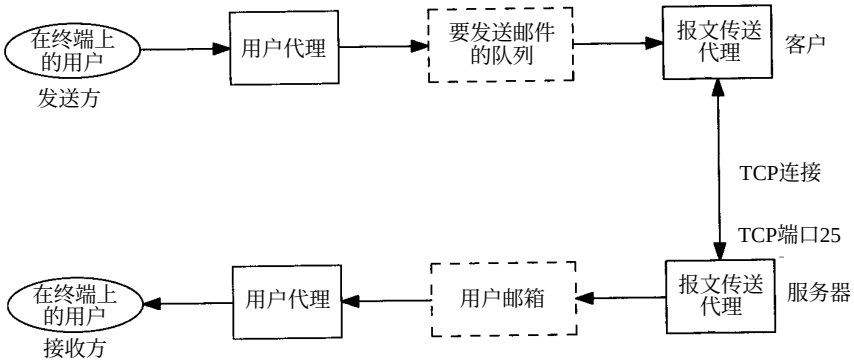


图28-1 Internet电子邮件示意图

用户与用户代理（user agent）打交道，可能会有多个用户代理可供选择。常用的Unix上的用户代理包括MH，Berkeley Mail, Elm和Mush。

用TCP进行的邮件交换是由报文传送代理MTA（Message Transfer Agent）完成的。最普通的Unix系统中的MTA是Sendmail。用户通常不和MTA打交道，由系统管理员负责设置本地的MTA。通常，用户可以选择它们自己的用户代理。

本章研究在两个MTA之间如何用TCP交换邮件。我们不考虑用户代理的运行或实现。

RFC 821 [Postel 1982] 规范了SMTP协议，指定了在一个简单TCP连接上，两个MTA如何进行通信。RFC 822 [Crocker 1982] 指定了在两个MTA之间用RFC 821发送的电子邮件报文的格式。

28.2 SMTP协议

两个MTA之间用NVT ASCII进行通信。客户向服务器发出命令，服务器用数字应答码和可选的人可读字符串进行响应。这与上一章的FTP类似。

客户只能向服务器发送很少的命令：不到12个（相比较而言，FTP超过40个）。我们用简单的例子说明发送邮件的工作过程，并不仔细描述每个命令。

28.2.1 简单例子

我们将发送一个只有一行的简单邮件，并观察SMTP连接。我们用 -v 标志调用用户代理，

它被传送给邮件传送代理（本例中是 Sendmail）。当设置该标志时，该 MTA显示在SMTP连接上发送和接收的内容。以 >>>开始的行是SMTP客户发出的命令，以 3位数字的应答码开始的行是从SMTP服务器来的。以下就是交互会话：

```
sun % mail -v rstevens@noao.edu                    调用我们的代理
To: rstevens@noao.edu                              这是用户代理的输出
Subject: testing                                   然后指示我们键入主题
                                                   用户代理在首部和正文之间加上一行空行
1, 2, 3.                                           这是我们键入的正文
.                                                  我们在一行上输入一个句点，说明完成了
Sending letter ... rstevens@noao.edu...            用户代理上详细的输出

Connecting to mailhost via ether...                以下是MTA(Sendmail)的输出
Trying 140.252.1.54...  connected.
220 noao.edu Sendmail 4.1/SAG-Noao.G89 ready at Mon, 19 Jul 93 12:47:34 MST

>>> HELO sun.tuc.noao.edu.
250 noao.edu Hello sun.tuc.noao.edu., pleased to meet you

>>> MAIL From:<rstevens@sun.tuc.noao.edu>
250 <rstevens@sun.tuc.noao.edu>... Sender ok
>>> RCPT To:<rstevens@noao.edu>
250 <rstevens@noao.edu>... Recipient ok

>>> DATA
354 Enter mail, end with "." on a line by itself

>>> .
250 Mail accepted

>>> QUIT
221 noao.edu delivering mail

rstevens@noao.edu... Sent
sent.                                              这是用户代理的输出
```

只有5个SMTP命令用于发送邮件：HELO，MAIL，RCTP，DATA和QUIT。

我们键入mail启动用户代理，然后键入主题（subject）的提示；键入后，再键入报文的正文。在一行上键入一个句点结束报文，用户代理把邮件传给 MTA，由MTA进行交付。

客户主动打开TCP端口25。返回时，客户等待从服务器来的问候报文（应答代码为 220）。该服务器的应答必须以服务器的完全合格的域名开始：本例中为 noao.edu（通常，跟在数字应答后面的文字是可选的。这里需要域名。以 Sendmail打头的文字是可选的）。

下一步客户用 HELO命令标识自己。参数必须是完全合格的的客户主机名： sun.tuc.noao.edu。

MAIL命令标识出报文的发起人。下一个命令， RCPT，标识接收方。如果有多个接收方，可以发多个RCPT命令。

邮件报文的内容由客户通过 DATA命令发送。报文的末尾由客户指定，是只有一个句点的一行。最后的命令QUIT，结束邮件的交换。

图28-2是在发送方SMTP（客户端）与接收方SMTP（服务器）之间的一个SMTP连接。

我们键入到用户代理的数据是一行报文（“ 1，2，3 ”），但在报文段12中共发送了393字节的数据。下面的12行组成了客户发送的393字节数据：

```
Received: by sun.tuc.noao.edu. (4.1/SMI-4.1)
        id AA00502; Mon, 19 Jul 93 12:47:32 MST
Message-Id: <9307191947.AA00502@sun.tuc.noao.edu.>
From: rstevens@sun.tuc.noao.edu (Richard Stevens)
Date: Mon, 19 Jul 1993 12:47:31 -0700
```

```
Reply-To: rstevens@noao.edu
X-Phone: +1 602 676 1676
X-Mailer: Mail User's Shell (7.2.5 10/14/92)
To: rstevens@noao.edu
Subject: testing

1, 2, 3.
```

前三行，`Received:`和`Message-Id:`由MTA加上；下一行由用户代理生成。

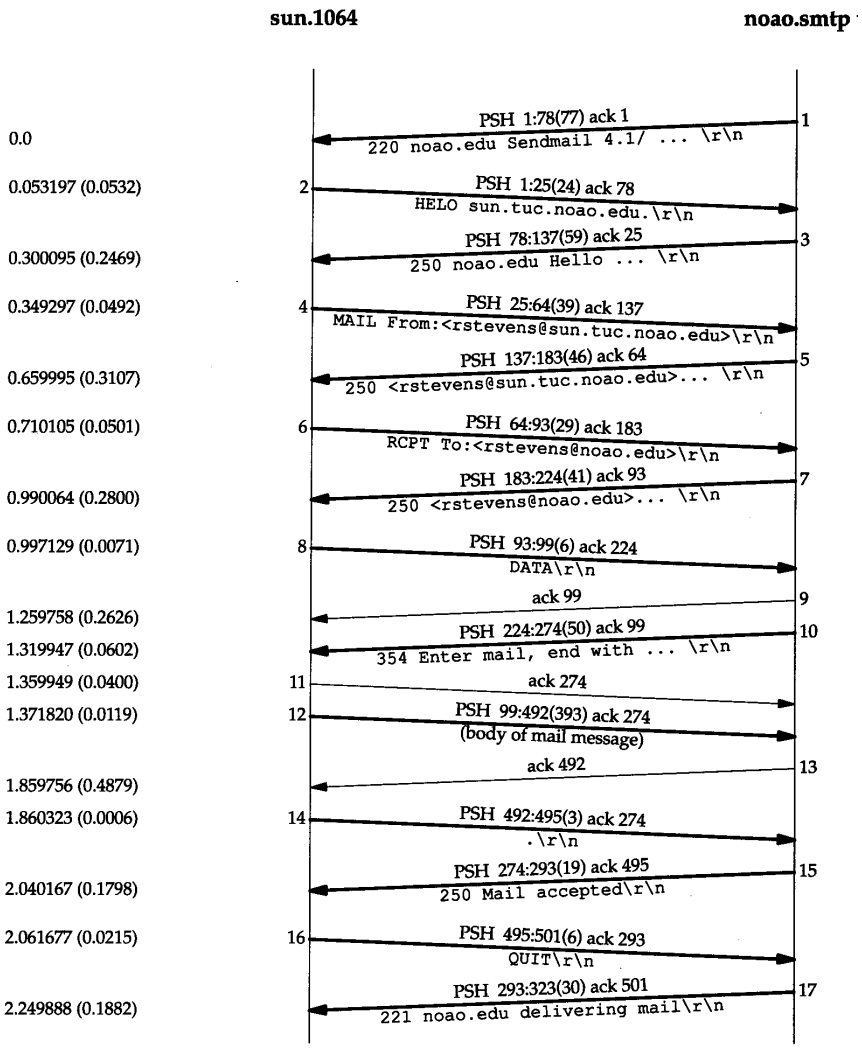


图28-2 基本SMTP邮件交付

28.2.2 SMTP命令

最小SMTP实现支持8种命令。我们在前面的例子中遇到5个：HELO，MAIL，RCPT，DATA和QUIT。

RSET命令异常中止当前的邮件事务并使两端复位。丢掉所有有关发送方、接收方或邮件的存储信息。

VRFY命令使客户能够询问发送方以验证接收方地址，而无需向接收方发送邮件。通常是系统管理员在查找邮件交付差错时手工使用的。我们将在下一节中给出这方面的例子。

NOOP命令除了强迫服务器响应一个OK应答码（200）外，不做任何事情。

还有附加和可选命令。EXPN扩充邮件表，与VRFY类似，通常是由系统管理员使用的。事实上，许多Sendmail的版本都把这两者等价地处理。

4.4BSD 中的Sendmail版本8不再将两者等同处理。VRFY不扩充别名也不接受.forward文件。

TURN命令使客户和服务器交换角色，无需拆除 TCP连接并建立新的连接就能以相反方向发送邮件（Sendmail不支持这个命令）。其他还有三个很少被实现的命令（SEND、SOML和SAML）取代MAIL命令。这三个命令允许邮件直接发送到客户终端（如果已注册）或发送到接收方的邮箱。

28.2.3 信封、首部和正文

电子邮件由三部分组成：

1) 信封（envelope）是MTA用来交付的。在我们的例子中信封由两个 SMTP命令指明：

```
MAIL From: <rstevens@sun.tuc.noao.edu>
RCPT To: <estevens@noao.edu>
```

RFC 821指明了信封的内容及其解释，以及在一个 TCP连接上用于交换邮件的协议。

2) 首部由用户代理使用。在我们的例子中可以看到 9个首部字段：`Received`、`Message-Id`、`From`、`Data`、`Reply-To`、`X-Phone`、`X-Mailer`、`To`和`Subject`。每个首部字段都包含一个名，紧跟一个冒号，接着是字段值。RFC 822指明了首部字段的格式的解释（以X-开始的首部字段是用户定义的字段，其他是由 RFC 822定义的）。长首部字段，如例子中的Received，被折在几行中，多余行以空格开头。

3) 正文（body）是发送用户发给接收用户报文的内容。RFC 822 指定正文为NVT ASCII文字行。当用DATA命令发送时，先发送首部，紧跟一个空行，然后是正文。用 DATA命令发送的各行都必须小于1000字节。

用户接收我们指定为正文的部分，加上一些首部字段，并把结果传到 MTA。MTA加上一些首部字段，加上信封，并把结果发送到另一个 MTA。

内容（content）通常用于描述首部和正文的结合。内容是客户用 DATA命令发送的。

28.2.4 中继代理

在我们的例子中本地MTA的信息输出的第1行是：“ Connecting to mailhost via ether ”（即“ 通过以太网连接到邮件主机 ”）。这是因为作者的系统已被配置成把所有非本地的向外的邮件发送到一台中继机上进行转发。

这样做的原因有两个。首先，简化了除中继系统 MTA外的其他所有MTA的配置（所有曾使用过Sendmail的人都能证明，配置一个MTA并不简单）。第二，它允许某个机构中的一个系统作为邮件集线器，从而可能把其他所有系统隐藏起来。

在这个例子中，中继系统在本地域（.tuc.noao.edu）中有一个mailhost的主机名，而其他所有系统都被配置成把它们的邮件发往该主机。我们可以执行 host命令来看看在DNS中这个名

是如何定义的：

```
sun % host mailhost
mailhost.tuc.noao.edu     CNAME     noao.edu          规范名
noao.edu                    A       140.252.1.54      它的真实IP地址
```

如果将来用于中继的主机改变了，只需改变它的 DNS名 —— 其他所有单个系统的邮箱配置都无需改变。

目前许多机构都采用中继系统。图 28-3是修改后的Internet邮件图（图28-2），考虑发送主机和最后的接收主机都可能使用中继主机。

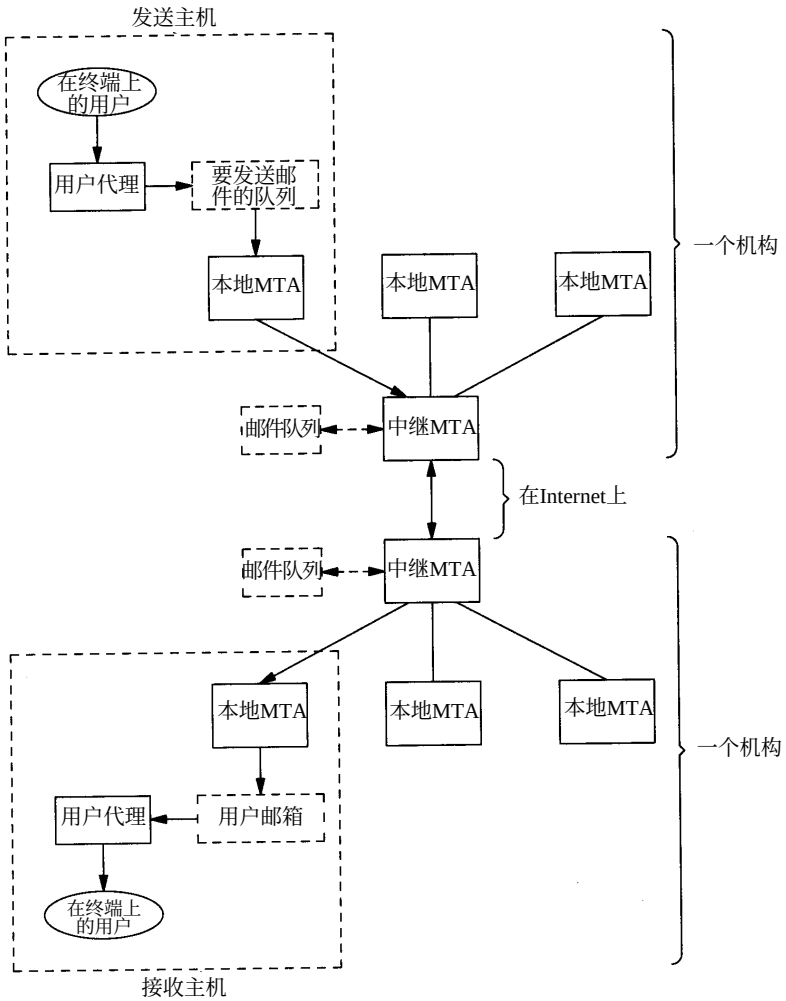


图28-3 在两端都有一个中继系统的Internet电子邮件

在这种情况下，在发送方和接收方之间有 4个MTA。发送方主机上的本地 MTA只把邮件交给它自己的中继 MTA（该中继 MTA可能在该机构的域中有一个 mailhost的主机名）。这个通信就在该机构的本地互联网上用 SMTP。然后，发送方机构的中继 MTA就在Internet上把邮件发送到接收方机构的中继 MTA上，而这个中继 MTA就通过与接收方主机上的本地 MTA通信，把邮件交给接收方主机。尽管可能存在其他协议，但这个例子中所有 MTA均使用SMTP协议。

28.2.5 NVT ASCII

SMTP的一个特色是它用NVT ASCII表示一切：信封、首部和正文。正如我们在 26.4节中谈到的，这是一个7 bit的字符码，以8 bit字节发送，高位比特被置为0。

在28.4节中，我们讨论了Internet邮件的一些新特性、允许发送和接收诸如音频和视频数据的扩充SMTP和多媒体邮件（MIME）。我们将看到，MIME和NVT ASCII一起表示信封、首部和正文，只需对用户代理作一些改变。

28.2.6 重试间隔

当用户把一个新的邮件报文传给它的 MTA时，通常立即试图交付。如果交付失败， MTA必须把该报文放入队列中以后再重试。

Host Requirements RFC推荐初始时间间隔至少为30分钟。发送方至少4~5天内不能放弃。而且，因为交付失败通常是透明的（接收方崩溃或临时网络连接中断），所以当报文在队列中等待的第1个小时内，尝试两次连接是有意义的。

28.3 SMTP的例子

上面我们说明了普通邮件发送，在这里我们将说明 MX记录如何用于邮件发送，以及VRFY和EXPN命令的用法。

28.3.1 MX记录：主机非直接连到Internet

在14.6节中我们提到DNS中的一种资源记录类型是邮件交换记录，称为 MX记录。在下面的例子中我们将说明如何用 MX记录向不直接连到 Internet的主机发送邮件。 RFC 974 [Partridge 1986] 描述了MTA对MX记录的处理。

主机mlfarm.com不是直接连到Internet的，但是有一个MX记录指向Internet上的一个邮件转发器。

```
sun % host -a -v -t mx mlfarm.com
The following answer is not authoritative:
mlfarm.com              86388    IN    MX    10 mercury.hsi.com
mlfarm.com              86388    IN    MX    15 hsi86.hsi.com
Additional information:
mercury.hsi.com         86388    IN     A    143.122.1.91
hsi86.hsi.com          172762    IN     A    143.122.1.6
```

有两个MX记录，各有不同的优先级。我们希望MTA从优先级数值低的开始。

```
sun % mail -v ron@mlfarm.com                    -v标志看MTA在做什么
To: ron@mlfarm.com
Subject: MX test message

                                                在这里键入报文的正文(没显示出来)
.                                               一行中的一个句号，结束报文

Sending letter ... ron@mlfarm.com...
Connecting to mlfarm.com via tcp...
mail exchanger is mercury.hsi.com               找到MX记录
Trying 143.122.1.91...  connected.             先试优先级低的那一个
220 mercury.hsi.com ...

                                                下面是正常的SMTP邮件传送
```

从输出中我们看到，MTA发现目的主机有一个MX记录，并使用具有低优先级数值的MX记录。

在主机sun运行这个例子之前，它被配置成不使用本地中继主机，所以我们会看到与目的主机的邮件交换。主机sun还被配置成可使用主机noao.edu（通过拨号SLIP链路）上的域名服务器，所以我们能用tcpdump捕获在SLIP链路上进行的邮件发送和DNS通信。图28-4显示了tcpdump输出的开始部分。

```
1  0.0                 sun.1624 > noao.edu.53: 2+ MX? mlfarm.com. (28)
2  0.445572 (0.4456)   noao.edu.53 > sun.1624: 2* 2/0/2 MX
                                                mercury.hsi.com. 10 (113)

3  0.505739 (0.0602)   sun.1143 > mercury.hsi.com.25: S 1617536000:1617536000(0)
                                                       win 4096
4  0.985428 (0.4797)   mercury.hsi.com.25 > sun.1143: S 1832064000:1832064000(0)
                                                       ack 1617536001 win 16384
5  0.986003 (0.0006)   sun.1143 > mercury.hsi.com.25: . ack 1 win 4096
6  1.735360 (0.7494)   mercury.hsi.com.25 > sun.1143: P 1:90(89) ack 1 win 16384
```

图28-4 向一个使用MX记录的主机发送邮件

在第1行，MTA向它的域名服务器查询mlfarm.com的MX记录。跟在2后面的加号“+”意思是设置要求递归的标志位。第2行的响应置位授权比特（跟在2后面的星号“*”），并包含两个回答RR（两个MX主机名），0个授权RR，以及两个附加的RR（两个主机的IP地址）。

第3~5行与主机mercury.hsi.com上的SMTP建立了一个TCP连接。服务器的初始响应220显示在第6行。

由于某种原因，主机mercury.hsi.com必须把这个邮件报文交付给目的地，mlfarm.com。对于没有连接到Internet上与它的MX站点交换邮件的系统，UUCP协议是一种常用的办法。

在这个例子中，MTA要求一个MX记录，得到一个肯定的结果，然后发送邮件。但不幸的是，MTA与DNS之间的交互随不同的实现而不同。RFC 974指定MTA必须首先要求MX记录，如果没有，就尝试提交给目的主机（也就是说，向DNS要主机的记录和IP地址）。MTA也必须处理DNS中的CNAM记录（规范的名）。

作为一个例子，如果我们从一个BSD/386主机上向rstevens@mailhost.tuc.noao.edu发送邮件，则MTA（Sendmail）执行以下步骤：

1) Sendmail向DNS询问主机mailhost.tuc.noao.edu的CNAME记录。我们看到存在一个CNAME记录：

```
sun % host -t cname mailhost.tuc.noao.edu
mailhost.tuc.noao.edu     CNAME    noao.edu
```

2) 发布一个要求noao.edu的CNAME记录的DNS查询，回答是不存在。

3) Sendmail向DNS寻求noao.edu的MX记录并得到一个记录：

```
sun % host -t mx noao.edu
noao.edu                  MX       noao.edu
```

4) Sendmail向DNS查询noao.edu的A记录（IP地址），并得到返回值140.252.1.54（这个A记录大概是由域名服务器为noao.edu返回的，作为第3步中MX应答的一个附加的RR）。

5) 启动一个到140.252.1.54的SMTP连接并发送邮件。

CNAME查询不是为MX记录（noao.edu）中返回的数据做的。MX记录中的数据不能是

别名——必须是具有一个A记录的主机名。

与只用DNS 的SunOS 4.1.3一起发布的Sendmail版本查询MX记录，并且如果没有找到MX记录就放弃。

28.3.2 MX记录：主机出故障

MX记录的另一个用途是在目的主机出故障时可提供另一个邮件接收器。如果看一下主机sun的DNS入口，我们就会看到它有两个MX记录：

```
sun % host -a -v -t mx sun.tuc.noao.edu
sun.tuc.noao.edu        86400     IN     MX       0 sun.tuc.noao.edu
sun.tuc.noao.edu        86400     IN     MX      10 noao.edu
Additional information:
sun.tuc.noao.edu        86400     IN      A      140.252.1.29
sun.tuc.noao.edu        86400     IN      A      140.252.13.33
noao.edu                86400     IN      A      140.252.1.54
```

最低优先级的MX记录表明应该首先尝试直接发送到主机本身，下一个优先级是把邮件发送到主机noao.edu。

在下面的描述中，在关掉目的SMTP服务器后，我们从主机vangogh.cs.berkeley.edu向位于主机sun.tuc.noao.edu的我们自己发送邮件。当端口25上的连接请求到达时，TCP应该响应一个RST，因为没有被动打开的进程为等待该端口而挂起。

```
vangogh % mail -v rstevens@sun.tuc.noao.edu
A test to a host that's down.
.
EOT
rstevens@sun.tuc.noao.edu... Connecting to sun.tuc.noao.edu. (smtp)...
rstevens@sun.tuc.noao.edu... Connecting to noao.edu. (smtp)...
220 noao.edu ...
```

下面是正常的*SMTP*邮件传送

我们看到MTA尝试联系sun.tuc.noao.edu，然后放弃，并转而联系noao.edu。

图28-5显示了TCP用一个RST向到来的SYN响应的tcpdump输出。

```
1 0.0                vangogh.3873 > 140.252.1.29.25: S 2358303745:2358303745(0) ...
2 0.000621 (0.0006) 140.252.1.29.25 > vangogh.3873: R 0:0(0) ack 2358303746 win 0
3 0.300203 (0.2996) vangogh.3874 > 140.252.13.33.25: S 2358367745:2358367745(0) ...
4 0.300620 (0.0004) 140.252.13.33.25 > vangogh.3874: R 0:0(0) ack 2358367746 win 0
```

图28-5 尝试连接一个不在运行的SMTP服务器

第1行vangogh向sun的第1个IP地址140.252.1.29 的端口25 发送一个SYN。在第2行它被拒绝。然后，vangogh上的SMTP客户尝试sun的第2个IP地址140.252.13.33（第3行），也产生一个RST的返回（第4行）。

SMTP客户不区分第1行它主动打开时所返回的不同差错，而这是导致它在第 2行尝试其他IP地址的原因。如果第1次的差错是类似“ host unreachable(主机不可达)”，那么第2次尝试或许可行。

如果SMTP客户的主动打开失败的原因是因为服务器主机出故障了，我们将看到客户会向IP地址140.252.1.29重传SYN总共75秒（类似于图18-6）。然后客户向IP地址140.252.13.33发送另一个75秒的其他3个SYN。150秒后客户会移到下一个具有更高优先级的 MX记录。

28.3.3 VRFY和EXPN命令

VRFY命令无需发送邮件而验证某个接收方地址是否OK。EXPN的目的是无需向邮件表发送邮件就可以扩充该表。许多SMTP实现（如Sendmail）把两者看成一个，但我们提到新的Sendmail区分这两者。

作为一个简单测试，我们可以连到一个新的Sendmail版本，并看到不同之处（已经删除了无关的Telnet客户输出）。

```
sun % telnet vangogh.cs.berkeley.edu 25
220-vangogh.CS.Berkeley.EDU Sendmail 8.1C/6.32 ready at Tue, 3 Aug 1993 14:
59:12 -0700
220 ESMTP spoken here
helo bsdi.tuc.noao.edu
250 vangogh.CS.Berkeley.EDU Hello sun.tuc.noao.edu [140.252.1.29], pleased
to meet you
vrfy nosuchname
550 nosuchname... User unknown
vrfy rstevens
250 Richard Stevens <rstevens@vangogh.CS.Berkeley.EDU>
expn rstevens
250 Richard Stevens <rstevens@noao.edu>
```

首先注意到我们故意在HELO命令中键入错误的主机名：bsdi，而不是sun。许多SMTP服务器得到客户的IP地址，完成一个DNS指针查询（14.5节）并比较主机名。这样允许服务器基于IP地址注册到客户的连接，而不是基于用户可能错误键入的名。某些服务器会用幽默的报文回答，如“你是一个骗子”，或“为什么叫你自己……”。在这个例子中我们看到，这个服务器通过指针查询只打印出我们的真实域名以及我们的IP地址。

然后我们用一个无效的名字键入VRFY命令，服务器就响应550差错。下一步我们键入一个有效的名字，服务器用本地主机上的用户名回答。然后我们试试EXPN命令，并得到一个不同的回答。EXPN命令决定到该用户的邮件是否被转发，并打印出转发的地址。

许多站点禁止VRFY和EXPN命令，有时是因为隐私，有时因为相信这是安全漏洞。例如，我们可以向白宫的SMTP服务器试试下面的命令：

```
sun % telnet whitehouse.gov 25
220 whitehouse.gov SMTP/smap Ready.
helo sun.tuc.noao.edu
250 (sun.tuc.noao.edu) pleased to meet you.
vrfy clinton
500 Command unrecognized
expn clinton
500 Command unrecognized
```

28.4 SMTP的未来

Internet邮件发生了很多改变。应当记得Internet邮件的三个组成部分：信封、首部和正文。新加入的SMTP命令影响了信封，首部中可以使用非ASCII字母，正文（MIME）中也加入了结构。本节中我们依次对这三部分的扩充进行讨论。

28.4.1 信封的变化：扩充的SMTP

RFC 1425 [Klensin等，1993a] 定义了扩充的SMTP的框架，其结果被称为扩充的 SMTP（ESMTP）。与其他我们已经讨论过的新特性一样，这些变化以向后兼容的方式被加入，所以不影响已有的实现。

如果客户想使用新的特性，首先通过发布一个 EHLO而不是HELO命令启动一个与服务器的会话。相兼容的服务器用 250应答码响应。这个应答通常有好几行，每行都包含一个关键字和一个可选的参数。这些关键字指定了该服务器支持的 SMTP扩充。新的扩充将在一个RFC中描述并以IANA注册（在一个多行应答中，各行数字应答码的后面都要有一个连字符。最后一行的数字应答码后面跟一个空行）。

我们将给出到4个SMTP服务器的初始连接，其中 3个支持扩充的SMTP。我们用Telnet和它们连接，但删掉了不必要的Telnet客户输出。

```
sun % telnet vangogh.cs.berkeley.edu 25
220-vangogh.CS.Berkeley.EDU Sendmail 8.1C/6.32 ready at Mon, 2 Aug 1993 15:
47:48 -0700
220 ESMTP spoken here

ehlo sun.tuc.noao.edu
250-vangogh.CS.Berkeley.EDU Hello sun.tuc.noao.edu [140.252.1.29], pleased
to meet you
250-EXPN
250-SIZE
250 HELP
```

这个服务器用一个多行220应答作为它的欢迎报文。对EHLO命令的250应答中列出的扩充命令是EXPN、SIZE和HELP。第一个和最后一个来自原来的 RFC 821规范，但它们是可选命令。ESMTP服务器说明除了新命令外，它们还支持哪些可选的 RFC 821命令。

这个服务器支持的SIZE关键字是在RFC 1427 [Klensin, Freed和Moore 1993] 中定义的。它让客户在MAIL FROM命令行中以字节的多少指定报文的大小，这样服务器就可以在客户开始发送该报文之前，验证它是否接收该长度的报文。增加这个命令的原因在于，随着对非 ASCII码（如图像、音频等）内容的支持， Internet邮件报文的长度在不断增大。

下一个主机也支持 ESMTP，注意250应答指明支持包含一个可选参数的 SIZE关键字。这表明该服务器将接受长度不超过461兆字节的报文。

```
sun % telnet ymir.claremont.edu 25
220 ymir.claremont.edu -- Server SMTP (PMDF V4.2-13 #4220)

ehlo sun.tuc.noao.edu
250-ymir.claremont.edu
250-8BITMIME
250-EXPN
250-HELP
250-XADR
250 SIZE 461544960
```

关键字8BITMIME来自于RFC 1426 [Klensin等，1993a]。它允许客户把关键字BODY加到MAIL FROM命令中，指定正文中是否包含NVT ASCII字符（默认的）或8 bit数据。除非客户收到服务器应答EHLO命令发来的8BITMIME关键字，否则禁止客户发送任何非 NVT ASCII字符（当我们在本节中谈到MIME时，我们将看到MIME不要求8 bit传送）。

该服务器也通告了XADR关键字。任何以X开头的关键字都指的是本地SMTP扩充。

另一个服务器也支持ESMTP，通知了我们已经看到的HELP和SIZE关键字。它也支持三个以X开头的本地扩充。

```
sun % telnet dbc.mtview.ca.us 25
220 dbc.mtview.ca.us Sendmail 5.65/3.1.090690, it's Mon, 2 Aug 93 15:48:50
-0700
ehlo sun.tuc.noao.edu
250-Hello sun.tuc.noao.edu, pleased to meet you
250-HELP
250-SIZE
250-XONE
250-XVRB
250 XQUE
```

最后，我们将看到当客户试图通过向一个不支持 EHLO的服务器发布EHLO命令来使用ESMTP时将发生什么。

```
sun % telnet relay1.uu.net 25
220 relay1.UU.NET Sendmail 5.61/UUNET-internet-primary ready at Mon, 2 Aug
93 18:50:27 -0400
ehlo sun.tuc.noao.edu
500 Command unrecognized
rset
250 Reset state
```

对EHLO命令，客户收到一个500应答而不是250应答。客户应发布RSET命令，并跟着一个HELO命令。

28.4.2 首部变化：非ASCII字符

RFC 1522 [Moore 1993] 指明了一个在RFC 822报文首部中如何发送非ASCII字符的方法。这样做的主要用途是为了允许在发送方名、接收方名以及主题中使用其他的字符。

首部字段中可以包含编码字(coded word)。它们具有以下格式：

=?*charset*?*encoding*?*encoded-text*?=

charset是字符集规范。有效值是两个字符串us-ascii和iso-8859-x，其中x是一个单个数字，例如在iso-8859-1中的数字“1”。

encoding是一个单个字符用来指定编码方法，支持两个值。

1) Q编码意思是引号中可打印的（quoted-printable），目的是用于拉丁字符集。大多数字符是作为NVT ASCII（当然最高位比特置0）发送的。任何要发送的字符若其第8比特置1则被作为3个字符发送：第1个是字符是“=”，跟着两个十六进制数。例如，字符é（它的二进制8 bit值为0xe9）作为三个字符发送：=E9。空格通常作为下划线或三个字符=20发送。这种编码的目的在于，某些文本中除了大多数ASCII字符外，还有几个特殊字符。

2) B意思是以64为基数的编码。文本中的3个连续字节（24bit）被编码成4个6 bit值。用于表示所有可能的6bit值的64个NVT ASCII字符如图28-6所示。当要编码的个数不是3的倍数时，等号符“=”被用作填充符。

下面两种编码方式的例子取自RFC 1522：

```
From: =?US-ASCII?Q?Keith_Moore?= <moore@cs.utk.edu>
To: =?ISO-8859-1?Q?Keld_J=F8rn_Simonsen?= <keld@dkuug.dk>
CC: =?ISO-8859-1?Q?Andr=E9_?= Pirard <PIRARD@vm1.ulg.ac.be>
Subject: =?ISO-8859-1?B?SWYgeW91IGNhbiByZWFkIHRoaXMgeW8=?=
 =?ISO-8859-2?B?dSB1bmRlcnN0YW5kIHRoZSBleGFtcGxlLg==?=
```

6 bit 值	ASCII 字符	6 bit 值	ASCII 字符	6 bit 值	ASCII 字符	6 bit 值	ASCII 字符
0	A	10	Q	20	g	30	w
1	B	11	R	21	h	31	x
2	C	12	S	22	i	32	y
3	D	13	T	23	j	33	z
4	E	14	U	24	k	34	0
5	F	15	V	25	l	35	1
6	G	16	W	26	m	36	2
7	H	17	X	27	n	37	3
8	I	18	Y	28	o	38	4
9	J	19	Z	29	p	39	5
a	K	1a	a	2a	q	3a	6
b	L	1b	b	2b	r	3b	7
c	M	1c	c	2c	s	3c	8
d	N	1d	d	2d	t	3d	9
e	O	1e	e	2e	u	3e	+
f	P	1f	f	2f	v	3f	/

图28-6 6 bit值的编码（以64为基数编码）

能处理这些首部的用户代理将输出：

```
From: Keith Moore <moore@cs.utk.edu>
To: Keld Jørn Simonsen <keld@dkuug.dk>
CC: André  Pirard <PIRARD@vm1.ulg.ac.be>
Subject: If you can read this you understand the example.
```

为说明以64为基数的编码方法是如何工作的，我们看一下主题行中前面 4个编码的字符：SWYg。按照图28-6写出这4个字符的6 bit值（S=0x12,W=0x16,Y=0x18以及g=0x20）的二进制码：

```
010010  010110  011000  100000
```

然后把这24 bit重新分组成3个8 bit字节：

```
01001001  01100110  00100000
 =0x49     =0x66     =0x20
```

它们是I、f和空格的ASCII表示。

28.4.3 正文变化：通用Internet邮件扩充

我们已经提到RFC 822指定正文是NVT ASCII文本行，没有结构。RFC 1521 [Borenstein和Freed 1993] 把扩充定义为允许把结构置入正文。这被称为MIME，即通用Internet邮件扩充。

MIME不要求任何扩充，我们在本节前面已作了说明（扩充的 SMTP或非ASCII标题）。MIME正好加入了一些告知收件者正文结构的新标题（与 RFC 822相一致）。正文仍可以用NVT ASCII码来发送，而不考虑邮件内容。虽然我们前面所述的一些扩充可能会和 MIME合在一起产生好的效果——扩充的SMTP SIZE命令，因为MIME报文能变得很长，以及非ASCII标题——这些扩充并不是MIME所要求的。与另一方交换MIME报文所需的一切，就是双方都要有一个能够理解MIME的用户代理。在任何一个MTA中不需要做任何改变。

MIME定义这5个新标题字段如下：

```
Mime-Version:
Content-Type:
Content-Transfer-Encoding:
```

```
Content-ID:
Content-Description:
```

作为例子，下面两个标题行可以出现在一个 Internet邮件报文中：

```
Mime-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
```

当前MIME版本是1.0，内容类型是无格式 ASCII码文本，即Internet邮件的默认选择。PLAIN这个字被认为是内容类型（ TEXT）的一个子类型，字符串 charset=US-ASCII是一个参数。

Text是MIME的7个被定义的内容类型之一。图 28-7总结了RFC 1521中定义的16个不同的内容类型和子类型。对具体的内容类型和子类型来说都有指定的很多参数。

内容类型	子类型	描　　述
text	plain richtext enriched	无格式文本 简单格式文本，如粗体、斜体或下划线等 richtext的简化和改进
multipart	mixed parallel digest altenative	多个正文部分，串行处理 多个正文部分，可并行处理 一个电子邮件的摘要 多个正文部分，具有相同的语义内容
message	rfc822 partial external-body	内容是另一个RFC 822邮件报文 内容是一个邮件报文的片断 内容是指向实际报文的指针
application	octet-stream postscript	任意二进制数据 一个PostScript程序
image	jpeg gif	ISO 10918格式 CompuServe的图形交换格式
audio	basic	用8 bit ISDN μ律格式编码
video	mpeg	ISO 11172格式

图28-7　MIME 内容类型和子类型

内容类型和用于内容的传送编码是相互独立的。前者由首部字段 Content-Type指明，后者由首部字段Content-Transfer-Encoding指明。在RFC 1521中定义了5种不同的编码格式。

1) 7bit，是默认的NVT ASCII；

2) quoted-printable，我们在前面的一个例子中看到有非 ASCII首部。当字符中只有很少一部分的第8 bit置1时非常有用；

3) base64，如图28-6所示；

4) 8bit，包含字符行，其中某些为非ASCII字符且第8bit置1；

5) binary编码，无需包含多行的8 bit数据。

对RFC 821 MTA，以上5种编码格式中只有前 3种是有效的。因为这 3种产生只包含NVT ASCII字符的正文。使用有8BITMIME支持的扩充SMTP允许使用8bit编码。

尽管内容类型和编码是独立的， RFC1521推荐有非ASCII数据的text使用quoted-printable，而image、audio、video和octet-stream application使用base64。这样允许与符合 RFC 821的MTA保持最大的互操作性。而且， multipart和message内容类型必须以7bit编码。

作为一个multipart内容类型的例子，图28-8显示了一个来自RFC发布清单的邮件报文。子类型是mixed，意思是各部分是顺序处理的，各部分的边界是字符串 NextPart，其前面是行首的两个连字符。

每个边界上可跟一行用于指明下一部分首部字段。忽略报文中第 1个边界之前和最后一个边界之后的所有内容。

因为在第一个边界后面跟着一个空行，而不是首部，所以在第 1个和第2个边界之间的数据的内容类型被假定为具有us-ascii字符集的text/plain。这是新RFC的文字描述。

但是第2个边界后面跟着首部字段。它指定了另一个multipart报文，具有边界OtherAccess。子类型为alternative，有两种不同的选择。第1种OtherAccess选项是用电子邮件获取RFC，第2种选项是用匿名FTP获取。MIME用户代理将列出这两种选项，允许我们选择一个，然后自动地用电子邮件或匿名FTP获取一份复制的RFC。

```
To: rfc-dist@nic.ddn.mil
Subject: RFC1479 on IDPR Protocol
Mime-Version: 1.0
Content-Type: Multipart/Mixed; Boundary="NextPart"
Date: Fri, 23 Jul 93 12:17:43 PDT
From: "Joyce K. Reynolds" <jkrey@isi.edu>


--NextPart                                  第1个边界

A new Request for Comments is now available in online RFC libraries.

. . .                                       这里的细节在新的RFC中

Below is the data which will enable a MIME compliant Mail Reader
implementation to automatically retrieve the ASCII version
of the RFCs.

--NextPart                                  第2个边界
Content-Type: Multipart/Alternative; Boundary="OtherAccess"
                                            一个具有新边界的嵌套的多部分报文

--OtherAccess
Content-Type:  Message/External-body;
        access-type="mail-server";
        server="mail-server@nisc.sri.com"

Content-Type: text/plain

SEND rfc1479.txt

--OtherAccess
Content-Type:   Message/External-body;
        name="rfc1479.txt";
        site="ds.internic.net";
        access-type="anon-ftp";
        directory="rfc"

Content-Type: text/plain


--OtherAccess--
--NextPart--                                最后的边界
```

图28-8 MIME multipart报文的例子

这一部分是MIME的一个简要概述。MIME的详细细节和例子，见RFC 1521和[Rose 1993]。

28.5 小结

电子邮件包括在两端（发送方和接收方）都有的一个用户代理以及两个或多个报文传送代理。可以把一个邮件报文分成三个部分：信封、首部和正文。我们已经看到这三个部分用SMTP和Internet标准是如何进行交换的。所有都作为NVT ASCII字符进行交换。

我们也看到了一些新的扩充：用于信封和非ASCII首部的扩充SMTP，以及使用MIME的正文增加了结构。MIME的结构和编码允许使用已有的7bit SMTP MTA交换任意二进制数据。

习题

28.1 读RFC 822，找到域文字（domain literal）的意思。试试用其中一个给自己发送邮件。

28.2 除了连接建立和终止外，要发送一个小的邮件报文的最小网络往返次数是多少？

28.3 TCP是一个全双工协议，但是SMTP用半双工的形式使用TCP。客户发送一个命令后停止等待应答。为什么客户不一次发送多个命令，如一行中包括HELO、MAIL、RCPT、DATA和QUIT命令（假定正文不是太大）？

28.4 当网络在接近其容量运行时，SMTP的这种半双工操作如何欺骗缓慢的启动机制？

28.5 当存在多个具有相同优先值的MX记录时，名服务器是否总能以相同的顺序返回它们？

第29章　网络文件系统

29.1　引言

本章中我们要讨论另一个常用的应用程序： NFS（网络文件系统），它为客户程序提供透明的文件访问。NFS的基础是Sun RPC：远程过程调用。我们首先必须描述一下 RPC。

客户程序使用 NFS不需要做什么特别的工作，当 NFS内核检测到被访问的文件位于一个NFS服务器时，就会自动产生一个访问该文件的 RPC调用。

我们对NFS如何访问文件的细节并不感兴趣，只对它如何使用 Internet的协议，尤其是UDP协议，感兴趣。

29.2　Sun远程过程调用

大多数的网络程序设计都是编写一些调用系统提供的函数来完成特定的网络操作的应用程序。例如，一个函数完成 TCP的主动打开，另一个完成 TCP的被动打开，一个函数在一个TCP连接上发送数据，另一个设置特定的协议选项（如激活 TCP的keepalive定时器）。在1.15节我们提到过两个常用的用于网络编程的函数集（ API）：插口(socket)和TLI。正像客户端和服务器端运行的操作系统可能会不相同一样，双方使用的 API也可能会不相同。由通信协议和应用协议决定一对客户和服务器是否可以彼此通信。如果两台主机连接在一个网络上，并且都有一个TCP/IP的实现，那么一台主机上的一个使用 C语言编写的、使用插口和TCP的Unix客户程序可以和另一台主机上的一个使用 COBOL语言编写的、使用其他 API和TCP的大型机服务器进行通信。

一般来说，客户发送命令给服务器，服务器向客户发送应答。目前为止，我们讨论过的所有应用程序 —— Ping，Traceroute，选路守护程序、以及 DNS、TFTP、BOOTP、SNMP、Telnet、FTP和SMTP的客户和服务器 —— 都是采用这种方式实现的。

远程过程调用RPC (Remote Procedure Call)是一种不同的网络程序设计方法。客户程序编写时只是调用了服务器程序提供的函数。这只是程序员所感觉到的，实际上发生了下面一些动作。

1) 当客户程序调用远程的过程时，它实际上只是调用了一个位于本机上的、由 RPC程序包生成的函数。这个函数被称为客户残桩（ stub）。客户残桩将过程的参数封装成一个网络报文，并且将这个报文发送给服务器程序。

2) 服务器主机上的一个服务器残桩负责接收这个网络报文。它从网络报文中提取参数，然后调用应用程序员编写的服务器过程。

3) 当服务器函数返回时，它返回到服务器残桩。服务器残桩提取返回值，把返回值封装成一个网络报文，然后将报文发送给客户残桩。

4) 客户残桩从接收到的网络报文中取出返回值，将其返回给客户程序。

网络程序设计是通过残桩和使用诸如插口或 TLI的某个API的RPC库例程来实现的，但是

用户程序——客户程序和被客户程序调用的服务器过程——不会和这个API打交道。客户应用程序只是调用服务器的过程，所有网络程序设计的细节都被 RPC程序包、客户残桩和服务器残桩所隐藏。

一个RPC程序包提供了很多好处。

1) 程序设计更加容易，因为很少或几乎没有涉及网络编程。应用程序设计员只需要编写一个客户程序和客户程序调用的服务器过程。

2) 如果使用了一个不可靠的协议，如 UDP，像超时和重传等细节就由 RPC程序包来处理。这就简化了用户应用程序。

3) RPC库为参数和返回值的传输提供任何需要的数据转换。例如，如果参数是由整数和浮点数组成的， RPC程序包处理整数和浮点数在客户机和服务器主机上存储的不同形式。这个功能简化了在异构环境中的客户和服务器的编码问题。

RPC程序设计的细节可以参看参考文献 [Stevens 1990]的第18章。两个常用的RPC程序包是Sun RPC和开放软件基金（OSF）分布式计算环境（DCE）的RPC程序包。我们对于RPC的兴趣在于想了解Sun RPC中过程调用和过程返回报文的形式，因为本章中讨论的网络文件系统使用了它们。Sun RPC的第2版定义在RFC 1057 [Sun Microsystems 1988a]中。

Sun RPC

Sun RPC有两个版本。一个版本建立在插口API基础上，和TCP和UDP打交道。另一个称为 TI-RPC的（独立于运输层），建立在TLI API基础上，可以和内核提供的任何运输层协议打交道。尽管本章中我们只讨论 TCP和UDP，从讨论的观点来看，两者是一样的。

图29-1显示的是使用 UDP时，一个RPC过程调用报文的格式。 IP首部和UDP首部是标准的首部，我们已经在图 3-1和图11-2中显示过。 UDP首部以下是 RPC程序包定义的部分。

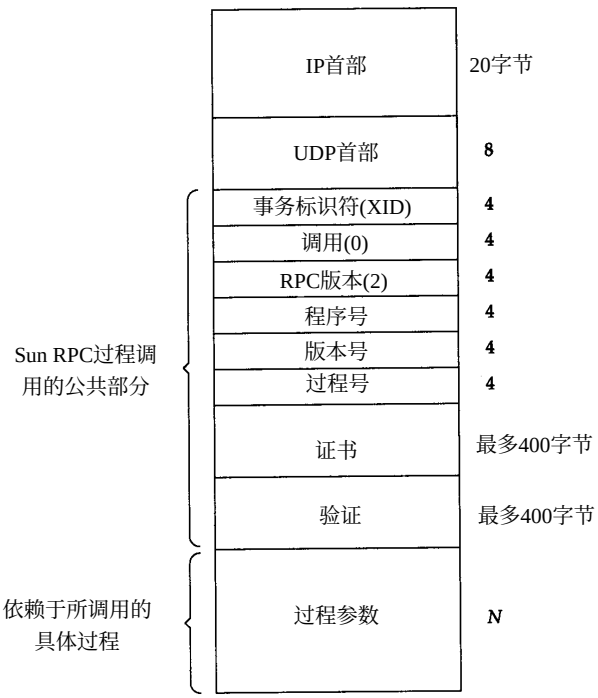


图29-1 RPC过程调用报文作为一个UDP数据报的格式

事务标识符（XID）由客户程序设置，由服务器程序返回。当客户收到一个应答，它将服务器返回的 XID与它发送的请求的XID相比较。如果不匹配，客户就放弃这个报文，等待从服务器返回的下一个报文。每次客户发出一个新的 RPC，它就会改变报文的XID。但是如果客户重传一个以前发送过的RPC（因为它没有收到服务器的一个应答），重传报文的XID不会修改。

调用(call)变量在过程调用报文中设置为 0，在应答报文中设置为 1。当前的RPC版本是2。接下来三个变量：程序号、版本号和过程号，标识了服务器上被调用的特定过程。

证书(credential)字段标识了客户。有些情况下，证书字段设置为空值；另外一些情况下，证书字段设置为数字形式的客户的用户号和组号。服务器可以查看证书字段以决定是否执行请求的过程。验证 (verifier)字段用于使用了DES加密的安全RPC。尽管证书字段和验证字段是可变长度的字段，它们的长度也作为字段的一部分被编码。

接下来是过程参数(procedure parameter)字段。参数的格式依赖于远程过程的定义。接收者（服务器残桩）如何知道参数字段的大小呢？既然使用的是UDP协议，UDP数据报的大小减去验证字段以上所有字段的长度就是参数的大小。如果使用的不是UDP而是TCP，因为TCP是一个字节流协议，没有记录边界，所以没有固定的长度。为了解决这个问题，在TCP首部和XID之间增加了一个4字节的长度字段，告诉接收者这个RPC调用由多少字节组成。这也使得一个RPC调用报文在必要时可以用多个TCP段来传输（DNS使用了类似的技术，参见习题14-4)。

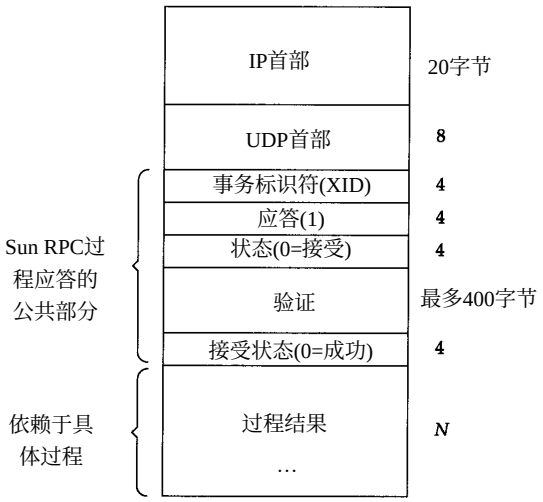


图29-2 RPC应答报文作为一个UDP数据报的格式

图29-2显示了一个RPC应答报文的格式。当远程过程返回时，服务器残桩将这个报文发送给客户残桩。

应答报文中的XID字段是从调用报文的XID字段复制而来。应答字段设置为 1，以区别于调用报文。如果调用报文被接受，状态字段设置为 0（如果RPC的版本号不为2，或者服务器不能鉴别客户的身份，调用报文可能被拒绝)。安全的RPC使用验证字段来标识服务器。

如果远程过程调用成功，接受状态字段置为 0。一个非零的值可能表示一个不合法的版本号或者一个不合法的过程号。如果使用的不是 UDP而是TCP，如同RPC调用报文一样，在TCP首部和XID字段之间插入一个4字节的长度字段。

29.3 XDR: 外部数据表示

外部数据表示XDR (eXternal Data Representation)是一个标准，用来对RPC调用报文和应答报文中的值进行编码。这些值包括 RPC首部字段（XID、程序号、接受状态等)、过程参数和过程结果。采用标准化的方法对这些值进行编码使得一个系统中的客户可以调用另一个不同架构的系统中的一个过程。 XDR在RFC 1014中定义[Sun Microsystems 1987]。

XDR定义了很多数据类型以及它们如何在一个 RPC报文中传输的具体形式（如比特顺序，字节顺序等)。发送者必须采用XDR格式构造一个RPC报文，然后接收者将XDR格式的报文转换为本机的表示形式。例如，在图29-1和图29-2中，我们显示的所有整数值（XID、调用字段、程序号等）都是4字节的整数。在XDR中，所有的整数的确占据 4个字节。XDR支持的其他数据类型包括无符号整数、布尔类型、浮点数、定长数组、可变长数组和结构。

29.4 端口映射器

包含远程过程的RPC服务器程序使用的是临时端口，而不是知名端口。这就需要某种形

式的“注册”程序来跟踪哪一个RPC程序使用了哪一个临时端口。在Sun RPC中，这个注册程序被称为端口映射器(port mapper)。

> “端口”这个词作为Internet协议族的一个特征，来自于TCP和UDP端口号。既然TI-RPC可以工作在任何运输层协议之上，而不仅仅是TCP和UDP，所以使用TI-RPC的系统中（如SVR4和Solaris 2.2），端口映射器的名字变成了rpcbind。下面我们继续使用更为常见的端口映射器的名字。

很自然地，端口映射器本身必须有一个知名端口：UDP端口111和TCP端口111。端口映射器也就是一个RPC服务器程序。它有一个程序号（100000）、一个版本号（2）、一个TCP端口111和一个UDP端口111。服务器程序使用RPC调用向端口映射器注册自身，客户程序使用RPC调用向端口映射器查询。端口映射器提供四个服务过程：

1) PMAPPROC_SET。一个RPC服务器启动时调用这个过程，注册一个程序号、版本号和带有一个端口号的协议。

2) PMAPPROC_UNSET。RPC服务器调用此过程来删除一个已经注册的映射。

3) PMAPPROC_GETPORT。一个RPC客户启动时调用此过程。根据一个给定的程序号、版本号和协议来获得注册的端口号。

4) PMAPPROC_DUMP。返回端口映射器数据库中所有的记录（每个记录包括程序号、版本号、协议和端口号）：

在一个RPC服务器程序启动，接着被一个RPC客户程序调用的过程中，进行了以下一些步骤：

1) 一般情况下，当系统引导时，端口映射器必须首先启动。它创建一个TCP端点，并且被动打开TCP端口111。它也创建一个UDP端点，并且在UDP端口111等待着UDP数据报的到来。

2) 当RPC服务器程序启动时，它为它所支持的程序的每一个版本创建一个TCP端点和一个UDP端点（一个给定的RPC程序可以支持多个版本。客户调用一个服务器过程时，说明它想要哪一个版本）。两个端点各自绑定一个临时端口（TCP端口号和UDP端口号是否一致无关紧要）。服务器通过RPC调用端口映射器的PMAPPROC_SET过程，注册每一个程序、版本、协议和端口号。

3) 当RPC客户程序启动时，它调用端口映射器的PMAPPROC_GETPORT过程来获得一个指定程序、版本和协议的临时端口号。

4) 客户发送一个RPC调用报文给第3步返回的端口号。如果使用的是UDP，客户只是发送一个包含RPC调用报文（见图29-1）的UDP数据报到服务器相应的UDP端口。服务器发送一个包含RPC应答报文（见图29-2）的UDP数据报到客户作为响应。

如果使用的是TCP，客户对服务器的TCP端口号做一个主动打开，然后在建立的TCP连接上发送一个RPC调用报文。服务器作为响应，在连接上发送一个RPC应答报文。

程序`rpcinfo(8)`打印了端口映射器中当前的映射记录（它调用了端口映射器的PMAPPROC_DUMP过程）。这里给出的是典型的输出：

```
sun % /usr/etc/rpcinfo -p
   program vers proto   port
    100005    1   tcp    702  mountd      NFS的安装守护程序
    100005    1   udp    699  mountd
    100005    2   tcp    702  mountd
    100005    2   udp    699  mountd
```

```
100003    2   udp   2049  nfs          NFS本身
100021    1   tcp    709  nlockmgr     NFS的加锁管理程序
100021    1   udp   1036  nlockmgr
100021    2   tcp    721  nlockmgr
100021    2   udp   1039  nlockmgr
100021    3   tcp    713  nlockmgr
100021    3   udp   1037  nlockmgr
```

可以看出一些程序确实支持多个版本。在端口映射器中，每一个程序号、版本号和协议的组合都有自己的端口号映射。

安装守护程序（mount daemon）的两个版本可以通过同样的TCP端口号（702）和同样的UDP端口号（699）来访问，而加锁管理程序（lock manager）的每个版本都有各自不同的端口号。

29.5 NFS协议

使用NFS，客户可以透明地访问服务器上的文件和文件系统。这不同于提供文件传输的FTP（第27章）。FTP会产生文件一个完整的副本。NFS只访问一个进程引用文件的那一部分，并且NFS的一个目的就是使得这种访问透明。这就意味着任何能够访问一个本地文件的客户程序不需要做任何修改，就应该能够访问一个NFS文件。

NFS是一个使用Sun RPC构造的客户服务器应用程序。NFS客户通过向一个NFS服务器发送RPC请求来访问其上的文件。尽管这一工作可以使用一般的用户进程来实现——即NFS客户可以是一个用户进程，对服务器进行显式调用。而服务器也可以是一个用户进程——因为两个理由，NFS一般不这样实现。首先，访问一个NFS文件必须对客户透明。因此，NFS的客户调用是由客户操作系统代表用户进程来完成的。第二，出于效率的考虑，NFS服务器在服务器操作系统中实现。如果NFS服务器是一个用户进程，每个客户请求和服务器应答（包括读和写的数据）将不得不在内核和用户进程之间进行切换，这个代价太大。

本节中，我们考察在RFC1094中说明的第2版的NFS [Sun Microsystems 1988b]。[X/Open 1991] 中给出了Sun RPC、XDR和NFS的一个更好的描述。[Stern 1991] 给出了使用和管理NFS的细节。第3版的NFS协议在1993年发布，我们在29.7节中对它做一个简单的描述。

图29-3显示了一个NFS客户和一个NFS服务器的典型配置，图中有很多地方需要注意。

1) 访问的是一个本地文件还是一个NFS文件对于客户来说是透明的。当文件被打开时，由内核决定这一点。文件被打开之后，内核将本地文件的所有引用传递给名为“本地文件访问”的框中，而将一个NFS文件的所有引用传递给名为“NFS客户”的框中。

2) NFS客户通过它的TCP/IP模块向NFS服务器发送RPC请求。NFS主要使用UDP，最新的实现也可以使用TCP。

3) NFS服务器在端口2049接收作为UDP数据报的客户请求。尽管NFS可以被实现成使用端口映射器，允许服务器使用一个临时端口，但是大多数的实现都是直接指定UDP端口2049。

4) 当NFS服务器收到一个客户请求时，它将这个请求传递给本地文件访问例程，后者访问服务器主机上的一个本地的磁盘文件。

5) NFS服务器需要花一定的时间来处理一个客户的请求。访问本地文件系统一般也需要一部分时间。在这段时间间隔内，服务器不应该阻止其他的客户请求得到服务。为了实现这一功能，大多数的NFS服务器都是多线程的——即服务器的内核中实际上有多个NFS服务器在

运行。具体怎么实现依赖于不同的操作系统。既然大多数的 Unix内核不是多线程的，一个共同的技术就是启动一个用户进程（常被称为 nfsd）的多个实例。这个实例执行一个系统调用，使自己作为一个内核进程保留在操作系统的内核中。

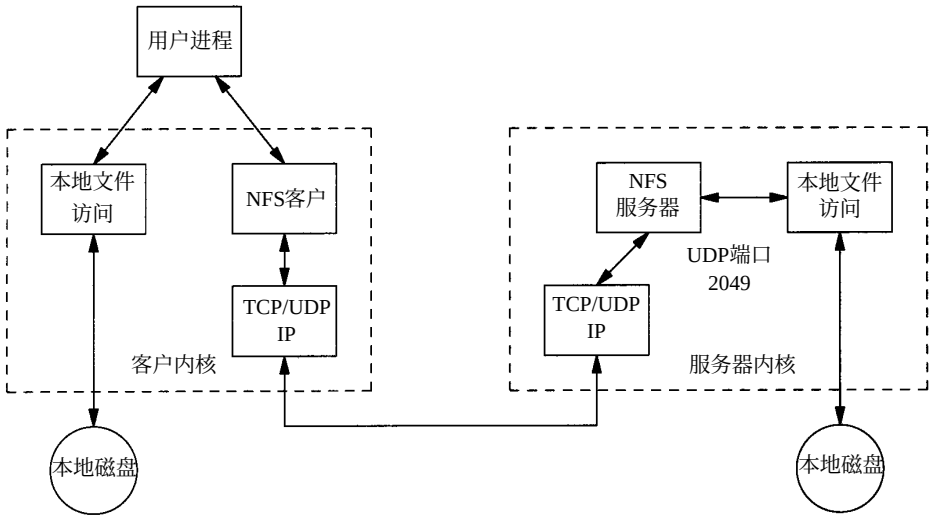


图29-3 NFS客户和NFS服务器的典型配置

6) 同样，在客户主机上， NFS客户需要花一定的时间来处理一个用户进程的请求。 NFS客户向服务器主机发出一个 RPC调用，然后等待服务器的应答。为了给使用 NFS的客户主机上的用户进程提供更多的并发性，在客户内核中一般运行着多个 NFS客户。同样，具体实现也依赖于操作系统。Unix系统经常使用类似于NFS服务器的技术：一个叫作biod的用户进程执行一个系统调用，作为一个内核进程保留在操作系统的内核中。

大多数的Unix主机可以作为一个NFS客户，一个NFS服务器，或者两者都是。大多数PC机的实现（MS-DOS）只提供了NFS客户实现。大多数的IBM大型机只提供了NFS服务器功能。

NFS实际上不仅仅由NFS协议组成。图29-4显示了NFS使用的不同RPC程序。

应用程序	程序号	版本号	过程数
端口映射器	100000	2	4
NFS	100003	2	15
安装程序	100005	1	5
加锁管理程序	100021	1, 2, 3	19
状态监视器	100024	1	6

图29-4 NFS使用的不同RPC程序

在这个图中，程序的版本是在SunOS 4.1.3中使用的。更新的实现提供了其中一些程序更新的版本。例如，Solaris 2.2还支持端口映射器的第3版和第4版，以及安装守护程序的第2版。SVR4支持第3版的端口映射器。

在客户能够访问服务器上的文件系统之前， NFS客户主机必须调用安装守护程序。我们在下面讨论安装守护程序。

加锁管理程序和状态监视器允许客户锁定一个 NFS服务器上文件的部分区域。这两个程

序独立于NFS协议，因为加锁需要知道客户和服务器的状态，而 NFS本身在服务器上是无状态的（下面我们对 NFS的无状态会介绍得更多）。[X/Open 1991] 的第9, 10和11章说明了使用加锁管理程序和状态监视器进行 NFS文件锁定的过程。

29.5.1 文件句柄

NFS中一个基本概念是文件句柄(file handle)。它是一个不透明(opaque)的对象，用来引用服务器上的一个文件或目录。不透明指的是服务器创建文件句柄，把它传递给客户，然后客户访问文件时，使用对应的文件句柄。客户不会查看文件句柄的内容 —— 它的内容只对服务器有意义。

每次一个客户进程打开一个实际上位于一个 NFS服务器上的文件时，NFS客户就会从NFS服务器那里获得该文件的一个文件句柄。每次 NFS客户为用户进程读或写文件时，文件句柄就会传给服务器以指定被访问的文件。

一般情况下，用户进程不会和文件句柄打交道 —— 只有NFS客户和NFS服务器将文件句柄传来传去。在第2版的NFS中，一个文件句柄占据32个字节，第3版中增加为64个字节。

> Unix服务器一般在文件句柄中存储下面的信息：文件系统标识符（文件系统最大和最小的设备号），i-node号（在一个文件系统中唯一的数值）和一个 i-node的生成码（每当一个i-node被一个不同的文件重用时就改变的数值）。

29.5.2 安装协议

客户必须在访问服务器上一个文件系统中的文件之前，使用安装协议安装那个文件系统。一般情况下，这是在客户主机引导时完成的。最后的结果就是客户获得服务器文件系统的一个文件句柄。

图29-5显示了一个Unix客户发出mount(8)命令所发生的情况，它说明一个NFS的安装过程。

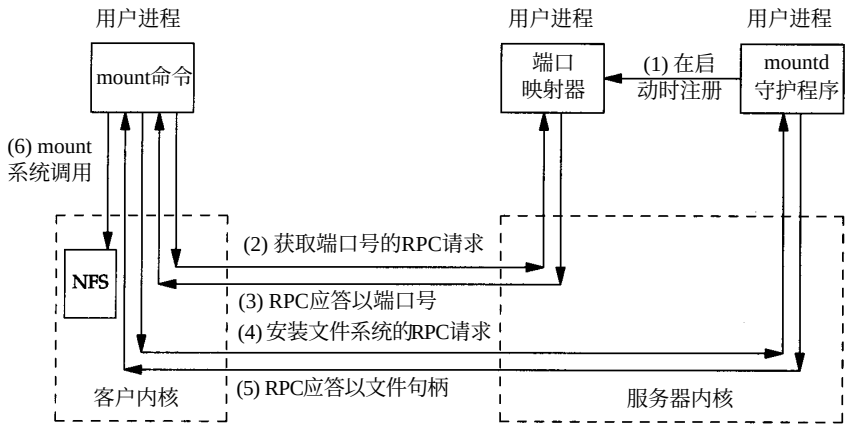


图29-5 使用Unix mount命令的安装协议

依次发生了下面的动作。

1) 服务器上的端口映射器一般在服务器主机引导时被启动。

2) 安装守护程序（mountd）在端口映射器之后被启动。它创建了一个 TCP端点和一个

UDP端点，并分别赋予一个临时的端口号。然后它在端口映射器中注册这些端口号。

3) 在客户机上执行mount命令，它向服务器上的端口映射器发出一个RPC调用来获得服务器上安装守护程序的端口号。客户和端口映射器交互既可以使用TCP也可以使用UDP，但一般使用UDP。

4) 端口映射器应答以安装守护程序的端口号。

5) mount命令向安装守护程序发出一个RPC调用来安装服务器上的一个文件系统。同样，既可以使用TCP也可以使用UDP，但一般使用UDP。服务器现在可以验证客户，使用客户的IP地址和端口号来判别是否允许客户安装指定的文件系统。

6) 安装守护程序应答以指定文件系统的文件句柄。

7) 客户机上的mount命令发出mount系统调用将第5步返回的文件句柄与客户机上的一个本地安装点联系起来。文件句柄被存储在NFS客户代码中，从现在开始，用户进程对于那个服务器文件系统的任何引用都将从使用这个文件句柄开始。

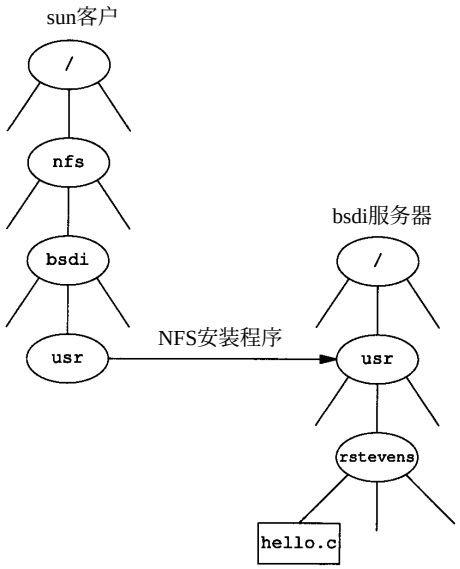


图29-6 将bsdi:/usr 目录安装成主机sun上的/nfs/bsdi/usr 目录

上述实现技术将所有的安装处理，除了客户机上的mount系统调用，都放在用户进程中，而不是放在内核中。我们显示的三个程序——mount命令、端口映射器和安装守护程序——都是用户进程。

作为一个例子，在我们的主机sun（一个NFS客户机）上执行：

```
sun  # mount -t nfs bsdi:/usr /nfs/bsdi/usr
```

这个命令将主机bsdi（一个NFS服务器）上的/usr目录安装成为本地文件系统/nfs/bsdi/usr。图29-6显示了结果。

当我们引用客户机sun上的/nfs/bsdi/usr/rstevens/hello.c文件时，实际上引用的是服务器bsdi上的文件/usr/rstevens/hello.c。

29.5.3 NFS过程

现在我们描述NFS服务器提供的15个过程（使用的个数与NFS过程的实际个数不一样，因为我们把它们按照功能分了组）。尽管NFS被设计成可以在不同的操作系统上工作，而不仅仅是Unix系统，但是一些提供Unix功能的过程可能不被其他操作系统支持（例如硬链接、符号链接、组的属主和执行权等）。[Stevens 1992]的第4章包含了Unix文件系统其他的一些信息，其中有些被NFS采用。

1) GETATTR。返回一个文件的属性：文件类型（一般文件，目录等）、访问权限、文件大小、文件的属主者及上次访问时间等信息。

2) SETATTR。设置一个文件的属性。只允许设置文件属性的一个子集：访问权限、文件

的属主、组的属主、文件大小、上次访问时间和上次修改时间。

3) STATFS。返回一个文件系统的状态：可用空间的大小、最佳传送大小等。例如 Unix的df命令使用此过程。

4) LOOKUP。查找一个文件。每当一个用户进程打开一个 NFS服务器上的一个文件时，NFS客户调用此过程。

5) READ。从一个文件中读数据。客户说明文件的句柄、读操作的开始位置和读数据的最大字节数（最多8192个字节）。

6) WRITE。对一个文件进行写操作。客户说明文件的句柄、开始位置、写数据的字节数和要写的数据。

7) CREATE。创建一个文件。

8) REMOVE。删除一个文件。

9) RENAME。重命名一个文件。

10) LINK。为一个文件构造一个硬链接。硬链接是一个 Unix的概念，指的是磁盘中的一个文件可以有任意多个目录项（即名字，也叫作硬链接）指向它。

11) SYMLINK。为一个文件创建一个符号链接。符号链接是一个包含另一个文件名字的文件。大多数引用符号链接的操作（例如，打开）实际上引用的是符号链接所指的文件。

12) READLINK。读一个符号链接。即返回符号链接所指的文件的名字。

13) MKDIR。创建一个目录。

14) RMDIR。删除一个目录。

15) READDIR。读一个目录。例如， Unix的ls命令使用此过程。

这些过程实际上有一个前缀NFSPROC_，我们把它省略了。

29.5.4 UDP还是TCP

NFS最初是用UDP写的，所有的厂商都提供了这种实现。最新的一些实现也支持 TCP。TCP支持主要用于广域网，它可以使文件操作更快。 NFS已经不再局限于局域网的使用。

当从LAN转换到WAN时，网络的动态特征变化得非常大。往返时间（round-trip time）变动范围大，拥塞经常发生。 WAN的这些特征使得我们考虑使用具有 TCP属性的算法 —— 慢启动，但是可以避免拥塞。既然 UDP没有提供任何类似的东西，那么在 NFS客户和服务器上加进同样的算法或者使用TCP。

29.5.5 TCP上的NFS

伯克利实现的Net/2 NFS支持UDP或者TCP。[Macklem 1991]描述了这个实现。让我们看一下使用TCP有什么不同。

1) 当服务器主机进行引导时，它启动一个 NFS服务器，后者被动打开 TCP端口2049，等待着客户的连接请求。这通常是另一个 NFS服务器，正常的NFS UDP服务器在UDP端口2049等待着进入的UDP数据报。

2) 当客户使用TCP安装服务器上的文件系统时，它对服务器上的 TCP端口2049做一个主动打开。这样就为这个文件系统在客户和服务器之间形成了一个 TCP连接。如果同样的客户安装同样服务器上的另一个文件系统，就会创建另一个 TCP连接。

3) 客户和服务器在它们连接的两端都要设置 TCP的keepalive选项，这样双方都能检测到对方主机崩溃，或者崩溃然后重启动。

4) 客户方所有使用这个服务器文件系统的应用程序共享这个 TCP连接。例如，在图 29-6中，如果在bsdi的/usr目录下还有另一个目录 smith，那么对两个目录 /nfs/bsdi/usr/rstevens和/nfs/bsdi/usr/smith下所有文件的引用将共享同样的 TCP连接。

5) 如果客户检测到服务器已经崩溃，或者崩溃然后重启动（通过收到一个 TCP差错“连接超时”或者“对方复位连接”），它尝试与服务器重新建立连接。客户做另一个主动打开，为同一个文件系统请求重新建立 TCP连接。在以前连接上超时的所有客户请求在新的连接上都会重新发出。

6) 如果客户机崩溃，那么当它崩溃时正在运行的应用程序也要崩溃。当客户机重新启动时，它很可能使用 TCP重新安装服务器的文件系统，这将导致和服务器的另一个连接。客户和服务器之间针对同一个文件系统的前一个连接现在打开了一半（服务器方认为它还开着），但是既然服务器设置了 keepalive选项，当服务器发出下一个 keepalive探查报文时，这个半开着的TCP连接就会被中止。

随着时间的流逝，另外一些厂商也计划支持 TCP上的NFS。

29.6 NFS实例

我们使用tcpdump来看一下在典型的文件操作中，客户调用了哪些 NFS过程。当tcpdump检测到一个包含 RPC调用（在图 29-1中调用字段等于 0）、目的端口是 2049的UDP数据报时，它把数据报按照一个 NFS请求进行解码。类似地，如果一个 UDP数据报是一个RPC应答（在图29-2中应答字段为1），源端口是2049，tcpdump就把此数据报作为一个NFS应答来解码。

29.6.1 简单的例子：读一个文件

第一个例子是使用cat(1)命令将位于一个NFS服务器上的一个文件复制到终端上：

```
sun % cat /nfs/bsdi/usr/rstevens/hello.c        把文件复制到终端
main()
{
    printf("hello, world\n");
}
```

如同图29-6所示，主机sun（NFS客户机）上的文件系统 /nfs/bsdi/usr 实际上是主机bsdi（NFS服务器）上的 /usr 文件系统。当cat打开这个文件时，sun上的内核检测到这一点，然后使用NFS去访问文件。图29-7显示了tcpdump的输出。

当tcpdump解析一个NFS请求或应答报文时，它打印客户的 XID字段，而不是端口号。第1行和第2行中的XID字段值是0x7aa6。

客户内核中的打开函数一次处理文件名 /nfs/bsdi/usr/rstevens/hello.c 中的一个成员。当处理到/nfs/bsdi/usr时，它发现这是指向一个已安装的 NFS文件系统的一个安装点。

在第1行中，客户调用 GETATTR过程取得客户已经安装的服务器目录的属性（ /usr）。这个RPC请求，除IP首部和UDP首部之外，包含 104个字节的数据。第 2行中的应答返回了一个OK值，除了IP首部和UDP首部之外，包含了 96个字节的数据。在这个图中，我们可以看出最小的NFS报文包含大约100个字节的数据。

```
1   0.0                 sun.7aa6 > bsdi.nfs: 104 getattr
2   0.003587 (0.0036)   bsdi.nfs > sun.7aa6: reply ok 96

3   0.005390 (0.0018)   sun.7aa7 > bsdi.nfs: 116 lookup "rstevens"
4   0.009570 (0.0042)   bsdi.nfs > sun.7aa7: reply ok 128

5   0.011413 (0.0018)   sun.7aa8 > bsdi.nfs: 116 lookup "hello.c"
6   0.015512 (0.0041)   bsdi.nfs > sun.7aa8: reply ok 128

7   0.018843 (0.0033)   sun.7aa9 > bsdi.nfs: 104 getattr
8   0.022377 (0.0035)   bsdi.nfs > sun.7aa9: reply ok 96

9   0.027621 (0.0052)   sun.7aaa > bsdi.nfs: 116 read 1024 bytes @ 0
10  0.032170 (0.0045)   bsdi.nfs > sun.7aaa: reply ok 140
```

图29-7 读一个文件的NFS操作

在第3行中，客户调用 LOOKUP过程来查看 rstevens文件。在第 4行中收到一个 OK应答。LOOKUP过程说明了文件名 rstevens和远程文件系统被安装时由内核保存的文件句柄。应答中包含了下一步要使用的一个新的文件句柄。

在第5行中，客户使用第4行中返回的文件句柄对 hello.c调用LOOKUP过程。在第6行返回了另一个文件句柄。新的文件句柄就是客户在第7行和第9行中引用文件/nfs/bsdi/usr/rstevens/hello.c所使用的文件句柄。我们看到客户对于正在打开的路径名的每个成员都调用了一次 LOOKUP过程。

在第7行中，客户又调用了一次 GETATTR过程，接着在第 9行中调用了 READ过程。客户请求从偏移 0开始的 1024个字节，但是接收到的没有这么多（减去 RPC字段和其他由 READ过程返回的值的大小，在第 10行中返回了 38个字节的数据。这是文件 hello.c的实际大小）。

在这个例子中，应用进程对于内核所做的这些 RPC请求和应答一点儿也不知道。应用进程只是调用了内核的 open函数，后者引起了 3个RPC请求和3个应答（1~6行），然后应用进程又调用了内核的 read函数，它引起了两个请求和两个应答（ 7~10行）。该文件位于一个 NFS文件服务器，这一点对客户应用进程来说是透明的。

29.6.2 简单的例子：创建一个目录

作为另一个简单的例子，我们将当前工作目录改变为一个 NFS服务器上的一个目录，然后创建一个新的目录：

```
sun  % cd  /nfs/bsdi/usr/rstevens          改变当前工作目录
sun  % mkdir  Mail                         并且创建一个目录
```

图29-8显示了tcpdump的输出。

```
1   0.0                 sun.7ad2 > bsdi.nfs: 104 getattr
2   0.004912 ( 0.0049)  bsdi.nfs > sun.7ad2: reply ok 96

3   0.007266 ( 0.0024)  sun.7ad3 > bsdi.nfs: 104 getattr
4   0.010846 ( 0.0036)  bsdi.nfs > sun.7ad3: reply ok 96

5   35.769875 (35.7590) sun.7ad4 > bsdi.nfs: 104 getattr
6   35.773432 ( 0.0036) bsdi.nfs > sun.7ad4: reply ok 96

7   35.775236 ( 0.0018) sun.7ad5 > bsdi.nfs: 112 lookup "Mail"
8   35.780914 ( 0.0057) bsdi.nfs > sun.7ad5: reply ok 28

9   35.782339 ( 0.0014) sun.7ad6 > bsdi.nfs: 144 mkdir "Mail"
10  35.992354 ( 0.2100) bsdi.nfs > sun.7ad6: reply ok 128
```

图29-8 NFS的操作：cd到NFS目录，然后mkdir

改变目录引起客户调用了两次GETATTR过程（1~4行）。当我们创建新的目录时，客户调用了GETATTR过程（5~6行），接着调用LOOKUP过程（7~8行，用来验证将创建的目录不存在），跟着调用了MKDIR过程来创建目录（9-10行）。在第8行中，应答OK并不表示目录存在。它只是表示过程返回了。 tcpdump并不理解NFS过程的返回值。它一般打印 OK和应答报文中数据的字节数。

29.6.3 无状态

NFS的一个特征（NFS的批评者称之为NFS的一个瑕疵，而不是一个特征）是 NFS服务器是无状态的(stateless)。服务器并不记录哪个客户正在访问哪个文件。请注意一下在前面给出的NFS过程中，没有一个 open操作和一个close操作。LOOKUP过程的功能与open操作有些类似，但是服务器永远也不会知道客户对一个文件调用了 LOOKUP过程之后是否会引用该文件。

无状态设计的理由是为了在服务器崩溃并且重启动时，简化服务器的崩溃恢复操作。

29.6.4 例子：服务器崩溃

在下面的例子中我们从一个崩溃然后重启动的 NFS服务器上读一个文件。这个例子演示了无状态的服务器是如何使得客户不知道服务器的崩溃。除了在服务器崩溃然后重启动时一个时间上的暂停外，客户并不知道发生的问题，客户应用进程没有受到影响。

在客户机sun上，我们对一个长文件（NFS服务器主机svr4上的文件/usr/share/lib/termcap）执行cat命令。在传送过程中把以太网的网线拔掉，关闭然后重启动服务器主机，再重新将网线连上。客户被配置成每个 NFS read过程读1024个字节。图29-9显示了tcpdump的输出。

1~10行对应于客户打开文件，操作类似于图 29-7所示。在第11行我们看到对文件的第一个READ操作，在12行返回了1024个字节的数据。这个操作一直继续到 129行（读1024个字节的数据，跟着一个OK应答）。

在第130行和第131行我们看到两个请求超时，并且分别在 132行和133行重传。第一个问题是这里为什么会有两个读请求，一个从偏移 65536开始读，另一个从偏移 73728开始读？答案是客户内核检测到客户应用进程正在进行顺序地读操作，所以试图预先取得数据块（大多数的Unix内核都采用了这种预读技术）。客户内核也正在运行多个 NFS块I/O守护程序，后者试图代表客户产生多个RPC请求。一个守护程序正在从偏移 65536处读8192个字节（以1024字节为一组数据块），而另一个正在从73728处预读8192个字节。

客户重传发生在130~168行。在第169行我们看到服务器已经重启动，在它对第 168行的客户NFS请求做出应答之前，它发送了一个 ARP请求。对168行的响应被发送在 171行。客户的READ操作继续进行下去。

除了从129行到171行5分钟的暂停，客户应用进程并不知道服务器崩溃然后又重启动了。这个服务器的崩溃对于客户是透明的。

为了研究这个例子中的超时和重传时间间隔，首先要意识到这儿有两个客户守护程序，分别有它们各自的超时。第 1个守护程序（在偏移65536处开始读）的间隔，四舍五入到两个十进制小数点，为0.68, 0.87, 1.74, 3.48, 6.96, 13.92, 20.0, 20.0, 20.0等等。第2个守护程序（在偏移73728处开始读）的间隔也是一样的（精确到两个小数点）。可以看出这些NFS客户使用了一个这样的超时定时器：间隔为0.875秒的倍数，上限为20秒。每次超时后，重传间隔翻倍：

0.875, 1.75, 3.5, 7.0和14.0。

```
 1    0.0                sun.7ade > svr4.nfs: 104 getattr
 2    0.007653 ( 0.0077) svr4.nfs > sun.7ade: reply ok 96

 3    0.009041 ( 0.0014) sun.7adf > svr4.nfs: 116 lookup "share"
 4    0.017237 ( 0.0082) svr4.nfs > sun.7adf: reply ok 128

 5    0.018518 ( 0.0013) sun.7ae0 > svr4.nfs: 112 lookup "lib"
 6    0.026802 ( 0.0083) svr4.nfs > sun.7ae0: reply ok 128

 7    0.028096 ( 0.0013) sun.7ae1 > svr4.nfs: 116 lookup "termcap"
 8    0.036434 ( 0.0083) svr4.nfs > sun.7ae1: reply ok 128

 9    0.038060 ( 0.0016) sun.7ae2 > svr4.nfs: 104 getattr
10    0.045821 ( 0.0078) svr4.nfs > sun.7ae2: reply ok 96

11    0.050984 ( 0.0052) sun.7ae3 > svr4.nfs: 116 read 1024 bytes @ 0
12    0.084995 ( 0.0340) svr4.nfs > sun.7ae3: reply ok 1124
```

连续地读

```
128    3.430313 ( 0.0013) sun.7b22 > svr4.nfs: 116 read 1024 bytes @ 64512
129    3.441828 ( 0.0115) svr4.nfs > sun.7b22: reply ok 1124

130    4.125031 ( 0.6832) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
131    4.868593 ( 0.7436) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

132    4.993021 ( 0.1244) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
133    5.732217 ( 0.7392) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

134    6.732084 ( 0.9999) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
135    7.472098 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

136   10.211964 ( 2.7399) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
137   10.951960 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

138   17.171767 ( 6.2198) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
139   17.911762 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

140   31.092136 (13.1804) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
141   31.831432 ( 0.7393) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

142   51.090854 (19.2594) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
143   51.830939 ( 0.7401) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

144   71.090305 (19.2594) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
145   71.830155 ( 0.7398) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
```

连续重传

```
167 291.824285 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
168 311.083676 (19.2594) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
```

服务器重启动

```
169 311.149476 ( 0.0658) arp who-has sun tell svr4
170 311.150004 ( 0.0005) arp reply sun is-at 8:0:20:3:f6:42

171 311.154852 ( 0.0048) svr4.nfs > sun.7b23: reply ok 1124

172 311.156671 ( 0.0018) sun.7b25 > svr4.nfs: 116 read 1024 bytes @ 66560
173 311.168926 ( 0.0123) svr4.nfs > sun.7b25: reply ok 1124
```

连续读

图29-9 当一个NFS服务器崩溃然后重启动时，客户正在读一个文件的过程

客户要重传多久呢？客户有两个与此有关的选项。首先，如果服务器文件系统是“硬”安装的，客户就会永远重传下去。但是如果服务器文件系统是“软”安装的，客户重传了固定数目的次数之后就会放弃。在“硬”安装的情况下，客户还有一个选项决定是否允许用户中断无限制的重传。如果客户主机安装服务器文件系统时说明了中断能力，并且如果我们不想在服务器崩溃之后等5分钟，等着服务器重启动，就可以键入一个中断键以终止客户应用

程序。

29.6.5 等幂过程

如果一个RPC过程被服务器执行多次仍然返回同样的结果，那么就把它叫作等幂过程（Idempotent Procedure）。例如，NFS的读过程是等幂的。正像我们在图29-9中看到的，客户只是重发一个特定的READ调用直到它得到一个响应。在我们的例子中，重传的原因是服务器崩溃了。如果服务器没有崩溃，而是RPC应答报文丢失了（既然UDP是不可靠的），客户只是重传请求，服务器再一次执行同样的READ过程。同一个文件的同一部分被重读一次，发送给客户。

这种方法行得通的原因在于每个READ请求指出了读操作开始的偏移位置。如果有一个NFS过程要求服务器读一个文件的下*N*个字节，这种方法就不行了。除非服务器被做成是有状态的（与无状态相反），如果一个应答丢失了，客户重发读下*N*个字节的READ请求，结果将是不一样的。这就是为什么NFS的READ和WRITE过程要求客户说明开始的偏移位置的原因。客户维护着状态（每个文件当前的偏移位置），而不是服务器。

不幸的是并不是所有的文件系统操作都是等幂的。例如，考虑下面的动作：客户NFS发出REMOVE请求来删除一个文件；服务器NFS删除了文件，并回答OK；服务器的回答丢失了；客户NFS超时，然后重传请求；服务器NFS找不到指定的文件，回答指出一个错误；客户应用程序接收到一个错误表示文件不存在。这个返回给客户应用程序的错误是不对的——该文件的确存在并且被删除了。

等幂的NFS过程是：GETATTR、STATES、LOOKUP、READ、WRITE、READLINK和READDIR。不是等幂的过程是：CREATE、REMOVE、RENAME、LINK、SYMLINK、MKDIR和RMDIR。SETATTR过程如果不用来截断文件，一般是等幂的。

既然使用UDP总会发生响应报文丢失的现象，NFS服务器需要一种方法来处理非等幂的操作。大多数的服务器实现了一个最近应答的高速缓存，用于存放非等幂操作最近的应答。每当服务器收到一个请求，它首先检查这个高速缓存，如果找到了一个匹配，就返回以前的应答而不再调用相应的NFS过程。[Juszczak 1989]提供了这种高速缓存的实现细节。

等幂服务器过程的概念可以应用于任何基于UDP的应用程序，而不仅仅是NFS。例如，DNS也提供了一个等幂服务。一个DNS的服务器可以任意多次地执行一个解析者的请求而没有任何不良的后果（如果不考虑网络资源浪费的话）。

29.7 第3版的NFS

1993年发布了第3版的NFS协议规范[Sun Microsystem 1994]。其实现有望在1994年成为可能。

我们总结一下第2版和第3版的主要区别。下面把两者分别称为V2和V3。

1) V2中的文件句柄是32字节的固定大小的数组。在V3中，它变成了一个最多为64个字节的可变长度的数组。在XDR中，一个可变长度的数组被编码为一个4字节的数组成员个数跟着实际的数组成员字节。这样在实现时减少了文件句柄的长度，例如Unix只需要12个字节，但又允许非Unix实现维护另外的信息。

2) V2将每个READ和WRITE RPC过程可以读写的数据限制为8192个字节。这个限制在V3

中取消了，这就意味着一个UDP上的实现只受到IP数据报大小的限制（65535字节）。这样允许在更快的网络上读写更大的分组。

3) 文件大小以及READ和WRITE过程开始偏移的字节从32字节扩充到64字节，允许读写更大的文件。

4) 每个影响文件属性值的调用都返回文件的属性。这样减少了客户调用GETATTR过程的次数。

5) WRITE过程可以是异步的，而在V2中要求同步的WRITE过程。这样可以提高WRITE过程的性能。

6) V3中删去了一个过程（STATFS），增加了七个过程：ACCESS（检查文件访问权限）、MKNOD（创建一个Unix特殊文件）、READDIRPLUS（返回一个目录中的文件名字和它们的属性）、FSINFO（返回一个文件系统的静态信息）、FSSTAT（返回一个文件系统的动态信息）、PATHCONF（返回一个文件的POSIX.1信息）和COMMIT（将以前的异步写操作提交到外存中）。

29.8 小结

RPC是构造客户-服务器应用程序的一种方式，使得看起来客户只是调用了服务器的过程。所有的网络操作细节都被隐藏在RPC程序包为一个应用程序生成的客户和服务器残桩以及RPC库的例程中。我们显示了RPC调用和应答报文的格式，并且提到了使用XDR对传输的值进行编码，使得RPC客户和服务器可以运行在不同架构的机器上。

最广泛使用的RPC应用之一就是Sun的NFS，一个在各种大小的主机上广泛实现的异构的文件访问协议。我们浏览了NFS和它使用UDP和TCP的方式。第2版的NFS协议定义了15个过程。

一个客户对一个NFS服务器的访问开始于安装协议，返回给客户一个文件句柄。客户接着可以使用那个文件句柄来访问服务器文件系统中的文件。在服务器上，一次检查文件名的一个成员，返回每个成员的一个新的文件句柄。最后的结果就是要引用的文件的一个文件句柄，它可以在随后的读写操作中被使用。

NFS试图把它的所用过程都做成等幂的，使得如果响应报文丢失了，客户只需要重发一个请求。我们看到了服务器崩溃然后又重启动时，一个客户读服务器上的一个文件的例子。

习题

29.1 在图29-7中，我们看到tcpdump将分组理解为NFS的请求和应答，打印了XID。tcpdump可以为任何的RPC请求或者应答这样做吗？

29.2 在一个Unix系统中，你认为为什么RPC服务器程序使用的是临时端口，而不是知名端口？

29.3 一个RPC客户调用了两个服务器过程。第1个服务器过程执行花了5秒钟的时间，第二个过程花了1秒钟。客户有一个4秒钟的超时。画出客户与服务器之间在时间轴上交互的信息（假定信息从客户传到服务器或者相反都不花时间）。

29.4 在图29-9的例子中，如果NFS服务器关机时，把它的以太网卡给换掉了，将会发生什么事情？

29.5 在图29-9中，当服务器重启动后，它处理了从偏移 65536开始的请求（168行和171行），然后处理了从偏移66560开始的下一个请求（172行和173行）。对于从偏移73728开始的请求怎么处理的呢？（167行）

29.6 当描述等幂NFS过程时，我们给出了一个REMOVE应答在网络中丢失的例子。在这种情况下，如果使用的是TCP而不是UDP会怎么样呢？

29.7 如果NFS服务器使用的是一个临时端口而不是 2049，那么当服务器崩溃然后又重启动时，一个NFS客户会发生什么情况呢？

29.8 每个主机最多只有1023个保留端口，所以保留端口是很缺乏的（ 1.9节）。如果一个NFS服务器要求它的客户拥有保留端口（公共的端口），一个NFS客户使用TCP安装了*N*个不同的服务器上的*N*个文件系统，那么客户对每个连接都需要一个不同的保留端口号吗？

第30章　其他的TCP/IP应用程序

30.1　引言

本章中我们描述了另外一些很多实现都支持的 TCP/IP应用程序。有些很简单，易于全面了解（Finger和Whois），而另一个则相当复杂（ X窗口系统）。我们只提供了这个复杂应用程序的一个简短的概述，集中介绍其对 TCP/IP协议的使用。

另外，我们提供一些 Internet上资源发现工具的概述。包括一组在 Internet上导航的工具，可以帮助寻找一些我们不知道确切位置和名字的信息。

30.2　Finger协议

Finger协议返回一个指定主机上一个或多个用户的信息。它常被用来检查某个人是否登录了，或者搞清一个人的登录名以便给他发送邮件。 RFC1288 [Zimmerman 1991] 指明了这个协议。

由于两个原因，很多站点不支持一个 Finger服务器。第一， Finger服务器的一个早期版本中的一个编程错误被 1988年声名狼藉的 Internet蠕虫病毒利用，作为进入点之一（ RFC1135 [Reynolds 1989] 和 [Curry 1992] 更详细地描述了蠕虫）。第二， Finger协议有可能会泄露一些很多管理员认为是有关用户的私有信息（登录名、电话号码，他们上次的登录时间，等等）。RFC1288的第3节给出了这个有关服务安全方面的细节。

从一个协议的角度来看， Finger服务器有一个知名的端口 79。客户对这个端口做一个主动打开，然后发送一个在线的请求。服务器处理这个请求，把输出发送回去，然后关闭连接。查询和响应都是采用NVT ASCII，类似于我们在FTP和SMTP协议中所看到的。

尽管大多数的Unix用户都是使用finger (1)客户来访问Finger服务器，我们将从使用 Telnet客户与Finger服务器直接相连开始，看看客户发出的每一条在线命令。如果客户的查询是一个空行（在NVT ASCII中，空行以一个回车符 CR跟着一个换行符 LF来传输），它就是一个请求查询所有在线用户信息的命令。

```
sun % telnet slip finger
Trying 140.252.13.65 ...
Connected to slip.
Escape character is '^]'.

Login    Name               Tty  Idle  Login Time   Office     Office Phone
rstevens Richard Stevens    *co    45  Jul 31 09:13
rstevens Richard Stevens    *c2    45  Aug  5 09:41
Connection closed by foreign host.
```

Telnet客户输出前三行

这儿我们键入回车作为Finger客户的命令

Telnet客户的输出

office和office phone的空白输出字段是从用户的口令 (password)文件记录的选项字段中取出的（在这个例子中，这两个字段的值没有提供）。

服务器必须在最后做一个主动的关闭操作，因为服务器返回的是一个可变长度的信息。

当客户收到文件结束字符时，就知道服务器的输出结束了。

当客户的请求由一个用户名组成时，服务器只以该用户的信息作为响应。下面是另一个例子，这个例子中删去了Telnet客户的输出：

```
sun % telnet vangogh.cs.berkeley.edu finger
rstevens                              这是我们键入的客户请求
Login: rstevens                       Name: Richard Stevens
Directory: /a/guest/rstevens          Shell: /bin/csh
Last login Thu Aug  5 09:55 (PDT) on ttyq2 from sun.tuc.noao.edu
Mail forwarded to: rstevens@noao.edu
No Plan.
```

当一个系统完全禁止了Finger服务时，因为没有进程被动打开端口79，所以客户的主动打开将从服务器接收到一个RST。

```
sun % finger @svr4
[svr4.tuc.noao.edu] connect: Connection refused
```

一些站点在端口79提供了一个服务器，但服务器只是向客户输出信息，而不理睬客户的任何请求：

```
sun % finger @att.com
[att.com]                  这一行是Finger客户输出的；其余行是服务器输出的
-------------------------------------------------------------
There are no user accounts on the AT&T Internet gateway.
To send email to an AT&T employee, send email to their name
separated by periods at att.com. If the employee has an email
address registered in the employee database, they will receive
email - otherwise, you'll receive a non-delivery notice.
For example: John.Q.Public@att.com
sun % finger clinton@whitehouse.gov
[whitehouse.gov]

        Finger service for arbitrary addresses on whitehouse.gov is not
supported. If you wish to send electronic mail, valid addresses are
"PRESIDENT@WHITEHOUSE.GOV", and "VICE-PRESIDENT@WHITEHOUSE.GOV".
```

对一个组织来说，另一种可能就是实现一个防火墙网关：在组织内部和Internet之间的一个路由器，负责过滤（也就是扔掉）特定的IP数据报（[Cheswick and Bellovin 1994]详细讨论了防火墙网关）。防火墙网关可以被配置成扔掉从Internet进来的这样一些数据报，这些数据报是目的端口为79的TCP报文段。

对于Finger的服务器和Unix的Finger客户还有其他的实现。欲知详情，请参考RFC1288和有关finger(1)的手册。

RFC1288指出提供了Finger服务器的、具有TCP/IP连接的自动售货机应该对客户的空行请求响应以现有产品的列表。对于由一个名字组成的客户请求，它们应该响应以一个数目或者与这个产品有关的可用项的列表。

30.3 Whois协议

Whois协议是另一种信息服务。尽管任何站点都可以提供一个Whois服务器，在InterNIC站点（rs.internic.net）的服务器是最常使用的。这个服务器维护着所有的DNS域和很多连接在Internet上的系统的系统管理员的信息（另一个可用的服务器在nic.ddn.mil，不过只包含了有关MILNET的信息）。不幸的是信息有可能是过期的或不完整的。RFC954 [Harrenstein, Stahl,

and Feinler 1985] 说明了Whois服务。

从协议的角度来看，Whois服务器有一个知名的TCP端口43。它接受客户的连接请求，客户向服务器发送一个在线的查询。服务器响应以任何可用的信息，然后关闭连接。请求和应答都以NVT ASCII来传输。除了请求和应答所包含的信息不一样，Whois服务器和Finger服务器几乎是一样的。

最常用的Unix客户程序是whois(1) 程序，尽管我们可以使用Telent自己手工键入命令。开始的命令是只包含一个问号的请求，服务器会返回所支持的客户请求的具体信息。

当NIC在1993年改变为InterNIC时，Whois服务器的站点也从 nic.ddn.mil移到了rs.internic.net。很多厂商仍然装载了采用nic.ddn.mil版本的whois客户程序。为了和正确的服务器联系上，你可能需要指明命令行参数 -h rs.internic.net。

另外，我们可以使用Telnet登录rs.internic.net站点，登录名采用whois。

我们将使用Whois服务器来查询一下本书的作者（已经删去了无关的 Telnet客户输出）。第一个请求是查询所有匹配“ stevens ”的名字。

```
sun % telnet rs.internic.net whois
stevens                                    这是我们键入的客户命令

                                           我们省略了其他25个“ stevens ”的信息
Stevens, W. Richard (WRS28)    stevens@kohala.com      +1 602 297 9416

The InterNIC Registration Services Host ONLY contains Internet
Information (Networks, ASN's, Domains, and POC's).
Please use the whois server at nic.ddn.mil for MILNET Information.
```

名字后面的括号中的三个大写字母跟着一个数字，（WRS28），是个人的NI句柄。下一个查询包含一个感叹号和一个NIC句柄，用于获得有关这个人的进一步信息。

```
sun % telnet rs.internic.net whois
!wrs28                                     我们键入的客户请求
Stevens, W. Richard (WRS28)                   stevens@kohala.com
   Kohala Software
   1202 E. Paseo del Zorro
   Tucson, AZ 85718
   +1 602 297 9416

   Record last updated on 11-Jan-91.
```

很多有关Internet变量的其他信息也可以查找。例如，请求 net 140.252将返回有关B类地址140.252的信息。

白页

使用SMTP的VRFY命令、Finger协议以及Whois协议在Internet上查找用户类似于使用电话号码簿的白页查找一个人的电话号码。在目前阶段，诸如上述的工具已经广泛可用了，为了提高这种服务的研究正在进行当中。

[Schwartz and Tsirigotis 1991] 包含了正在Internet上试验的不同白页服务的其他信息。一个叫作Netfind的特别工具可以通过使用 Telent，以netfind登录到bruno.cs.colorado或者ds.internic.net站点来访问。

RFC1309 [Weider, Reynolds, and Heker 1992]提供了对OSI目录服务X.500的概述，并且比较了它与当前的Internet技术（Finger和Whois）的相同点和不同点。

30.4 Archie、WAIS、Gopher、Veronica和WWW

前两节我们讨论的工具——Finger、Whois和一个白页服务——是用来查找人的信息的。还有一些工具是用来定位文件和文档的，本节中对这些工具给出了一个概述。我们只提供了一个概述，因为对每一个工具的细节的研究超出了本书的范围。我们给出了在Internet上找到这些工具的方法，鼓励你去试一试，找找看哪些工具可以帮助你。还有一些其他的工具正在被开发。[Obraczka, Danzig, and Li1993] 概述了在Internet上的资源发现服务。

30.4.1 Archie

本书中使用的很多资源都是使用匿名FTP得到的。问题是如何找到有我们想要的程序的FTP站点。有时候我们甚至不知道精确的文件名，但知道几个很可能在文件名中出现的关键字。

Archie提供了Internet上几千个FTP服务器的目录。我们可以通过登录进一个Archie服务器，搜索那些名字中包含了一个指定的常规表达式的文件。输出是一个与文件名匹配的FTP服务器的列表。然后我们可以使用匿名FTP去那个站点取得想要的文件。

全世界有很多Archie服务器。一个比较好的开始点是使用Telnet以archie名字登录进ds.internic.net，然后执行命令servers。这个命令的输出提供了所有Archie服务器以及它们的地址的一个列表。

30.4.2 WAIS

Archie帮助我们查找名字中包含关键字的文件，但有时候我们需要查找包含一个关键字的文件或数据库。即，想查找一个内容中包含一个关键字的文件，而不是文件名字中包含关键字。

WAIS (Wide Area Information Servers广域信息服务系统)知道几百个包含了有关计算机主题的和其他一般性主题信息的数据库。为了使用WAIS，我们要选择需要查找的数据库，指明关键字。尝试WAIS服务请使用Telnet，以wais名字登录quake.think.com站点。

30.4.3 Gopher

Gopher是其他Internet资源服务如Archie、WAIS和匿名FTP的一个菜单驱动的前端程序。Gopher是最容易使用的工具之一，因为不管它使用了哪个资源服务，它的用户界面都是一样的。

为了尝试Gopher，请使用Telnet，以gopher名字登录is.internic.net站点。

30.4.4 Veronica

就像Archie是一个匿名FTP服务器的索引一样，Veronica(Very Easy Rodent-Oriented Netwide Index to Computerized Archives)是一个Gopher标题的索引。一次Veronica搜索一般要查找几百个Gopher服务器。

我们必须通过一个Gopher客户来访问Veronica服务。选择Gopher的菜单项“Beyond InterNIC: Virtual Treasures of the Internet”，然后在下一个菜单中选择Veronica。

30.4.5 万维网WWW

万维网使用一个称为超文本的工具，使得我们可以浏览一个大的 /全球范围的服务和文档。信息和关键字一起显示，不过关键字被突出显示[⊖]。我们可以通过选择关键字得到更多的信息。

为了访问WWW，请使用Telnet登录info.cern.ch站点。

30.5 X窗口系统

X窗口系统(X Window System)，或简称为X，是一种客户-服务器应用程序。它可以使得多个客户（应用）使用由一个服务器管理的位映射显示器。服务器是一个软件，用来管理显示器、键盘和鼠标。客户是一个应用程序，它与服务器在同一台主机上或者在不同的主机上。在后一种情况下，客户与服务器之间通信的通用形式是 TCP，尽管也可以使用诸如 DECNET 的其他协议。在有些场合，服务器是与其他主机上客户通信的一个专门的硬件（一个 X终端）。在另一种场合，一个独立的工作站，客户与服务器位于同一台主机，使用那台主机上的进程间通信机制进行通信，而根本不涉及任何网络操作。在这两种极端情况之间，是一台既支持同一台主机上的客户又支持不同主机上的客户的工作站。

X需要一个诸如TCP的、可靠的、双向的流协议（ X不是为不可靠协议，如 UDP，而设计的）。客户与服务器的通信是由在连接上交换的 8 bit字节组成的。[Nye 1992] 给出了客户与服务器在它们的TCP连接上交换的150多个报文的格式。

在一个Unix系统中，当X客户和X服务器在同一台主机上时，一般使用 Unix系统的本地协议，而不使用 TCP协议，因为这样比使用 TCP的情况减少了协议处理时间。 Unix系统的本地协议是同一台主机上的客户和服务器之间可以使用的一种进程间通信的形式。回忆一下在图 2-4中，当使用TCP作为同一台主机上进程间的通信方式时，在 IP层以下发生了这个数据的环回(loopback)，隐含着所有的TCP和IP处理都发生了。

图30-1显示了三个客户使用一个显示器的可能的脚本。一个客户与服务器在同一台主机上，使用Unix系统的本地协议。另外两个位于不同的主机上，使用 TCP。一般来说，其中一个客户是一个窗口管理程序 (window manager)，它有权限管理显示器上窗口的布局。例如，窗口管理程序允许我们在屏幕上移动窗口，或者改变窗口的大小。

在这里客户和服务器这两个词猛一看含义相反了。对于 Telnet和FTP的应用，我们把客户看作是在键盘和显示器上的交互式用户。但是对于 X，键盘和显示器是属于服务器的。服务器被认为是提供服务的一方。 X提供的服务是对窗口、键盘和鼠标的访问。对于 Telnet，服务是登录远程的主机。对于FTP，服务是服务器上的文件系统。

当X终端或工作站引导时，一般启动 X服务器。服务器创建一个 TCP端点，在端口6000 + *n*上做一个被动打开，其中 *n*是显示器号（一般是 0）。大多数的 Unix服务器也使用名字 /tmp/.X11-unix/X*n*创建一个Unix系统的插口，其中*n*还是显示器的号。

当一个客户在另一台主机上启动时，它创建一个 TCP端点，对服务器上的端口 6000+*n*做一个主动打开。每个客户都得到了一个自己与服务器的连接。服务器负责对所有的客户请求进行复用。从这点开始，客户通过 TCP连接向服务器发送请求（例如，创建一个窗口），服务

⊖ 例如通过使用不同的颜色 —— 译者注。

器返回应答，服务器也发送事件给客户（鼠标按钮按下，键盘键按下，窗口暴露，窗口大小改变，等等）。

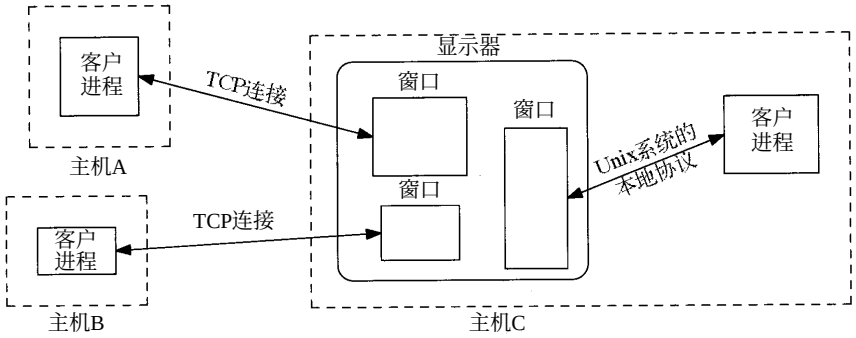


图30-1 使用一个显示器的三个X客户

图30-2将图30-1重新画，但强调了客户与X服务器进程间的通信，X服务器进程轮流管理着每个窗口。图中没有显示的是X服务器管理键盘和鼠标。

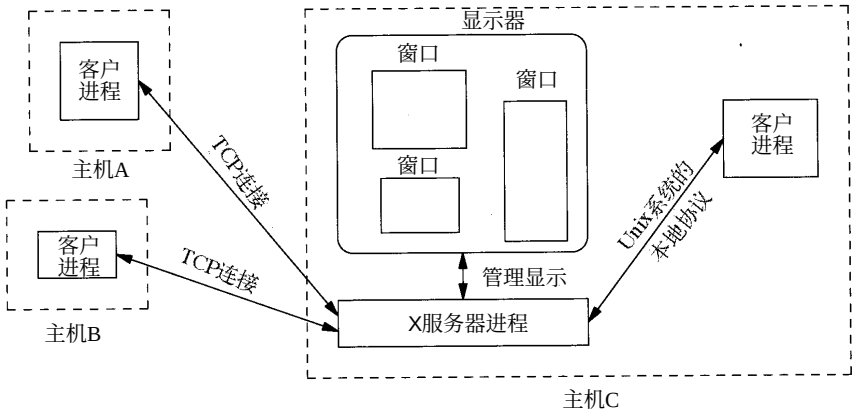


图30-2 使用一个显示器的三个客户

单个服务器处理多个客户请求的这种设计与我们在18.11节描述的正常的TCP并发服务器设计不同。例如，每次一个新的TCP连接请求到达，FTP和Telnet服务器都会产生一个新的进程，因此，每个客户都和一个不同的服务器进程通信。然而，对于X，运行在同一台主机或者在不同主机上的所有客户都和同一个服务器通信。

通过X客户和它的服务器之间的TCP连接可以交换很多数据。传输数据的数目依赖于特定的应用程序设计。例如，如果我们运行Xclock客户，Xclock在服务器的一个窗口中显示客户机当前的时间和日期。如果我们指定每隔1秒修改一次时间，那么每隔1秒，就会有一个X报文通过TCP连接从客户传输到服务器。如果我们运行X终端模拟程序，Xterm，我们敲的每一个键都会变成一个32字节的X报文（加上标准的IP和TCP首部就是72字节），在相反方向上的回送字符将是一个更大的X报文。[Droms and Dyksen 1990]检查了不同的X客户与一个特定的服务器之间的TCP流量。

30.5.1 Xscope程序

Xscope是检查X客户与它的服务器之间交换的信息的一个方便的程序。大多数的X窗口实

现都提供这个程序。它处在客户与服务器之间，双向传输所有的数据，同时解析所有的客户请求和服务器应答。图30-3显示了这种设置。

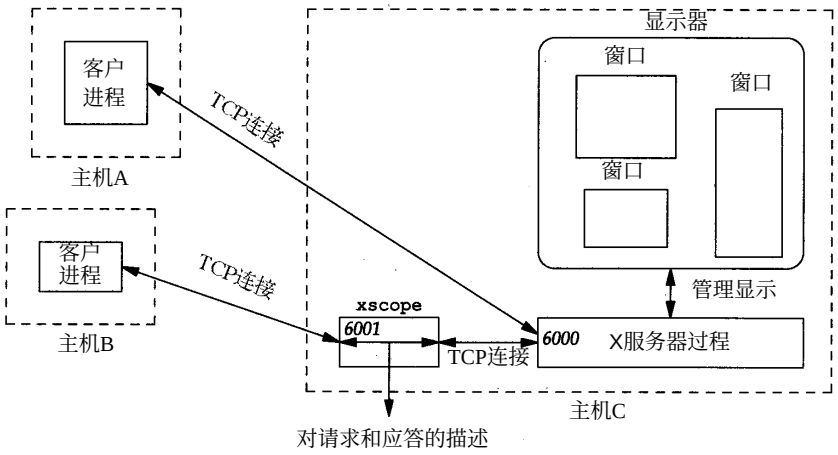


图30-3 使用xscope 监视一个X连接

首先，我们在服务器所在的主机上启动 xscope进程，但是xscope不是在端口6000而是在端口6001上监听TCP的连接请求。然后我们在另一台主机上启动一个客户，指明显示器号为1，而不是0，使得客户与xscope相连，而不直接与服务器相连。当客户的连接请求到达时，xscope创建与端口6000上的真正的服务器的一个 TCP连接，在客户与服务器之间复制所有的数据，同时生成请求与应答的一个可读的描述。

我们将在sun主机上启动xscope，然后在主机svr4上运行xclock客户。

```
svr4  %  DISPLAY=sun:1 xclock  -digital  -update 5
```

这条命令在主机sun的一个窗口中以数字形式显示时间和日期。我们指明了一个每 5秒的更新时间。

```
Thu  Sep  9  10:32:55  1993
```

我们对xscope指明一个 - q选项以产生最小的输出。为了看到每个报文的所有字段，可以使用不同的冗长级别。下面的输出显示了前三个请求和应答。

```
sun % xscope -q
0.00: Client --> 12 bytes
0.02:                                152 bytes <-- X11 Server
0.03: Client --> 48 bytes
        ............REQUEST: CreateGC
        ............REQUEST: GetProperty
0.20:                                396 bytes <-- X11 Server
                                     ..............REPLY: GetProperty
0.30: Client -->  8 bytes
0.38: Client --> 20 bytes
        ............REQUEST: InternAtom
0.43:                                 32 bytes <-- X11 Server
                                     ..............REPLY: InternAtom
```

客户的第1个在时刻0.00的报文和服务器在时刻 0.02的响应是客户与服务器之间标准的连接建立过程。客户标识它的字节顺序以及它希望的服务器版本。服务器响应以有关自己的不

同的信息。

下一个在时刻0.03的报文包含了两个客户请求。第1个请求在服务器上创建一个客户可以在其中画的图形上下文。第2个请求从服务器上得到一个属性（RESOURCE_MANAGER属性）。属性可以用于客户之间的通信，经常是在一个应用程序和窗口管理程序之间。服务器在时刻0.20的应答包含了这个属性。

下面两个在时刻0.30和0.38的客户报文形成了返回一个原子的单个请求（每个属性具有一个唯一的整型标识符称为原子）。服务器在时刻0.43的应答包含了这个原子。

如果不提供有关X窗口系统更多的细节是不可能进一步理解这个例子的，但这又不是本节的目的。在这个例子中，在窗口被显示之前，客户总共发送了1668个字节组成的12个报文段，服务器总共发送了1120个字节组成的10个报文段。耗费的时间为3.17秒。从这以后客户每5秒发送一个平均44个字节的小请求，请求更新窗口。这样一直持续到客户被终止。

30.5.2 LBX：低带宽X

为了将X用于局域网，对X协议使用的编码进行了优化，因为在局域网中花在对数据进行编码和解码的时间比最小化传输的数据更重要。尽管这种推断对以太网是适用的，但对于低速的串行线，如SLIP和PPP链路，就存在问题了（2.4节和2.6节）。

定义一个称为低带宽X（LBX）的标准的工作正在进行当中，它使用了下面的技术来减少网络流量的数目：快速缓存、只发送与前面分组的不同部分以及压缩技术。标准的规范和在第6版的X窗口系统中的一个样本实现应该会在1994年的早些时候完成。

30.6 小结

我们介绍的前两个应用，Finger和Whois，是用来获得用户信息的。Finger客户查询一个服务器，经常是为了找到某个人的登录名（以便给他们发电子邮件），或者去看一下某个人是否登录了。Whois客户一般与InterNIC运行的服务器联系，查找关于一个人、机构、域或网络号的信息。

我们简单描述了其他一些Internet资源发现服务：Archie、WAIS、Gopher、Veronica和WWW，帮助我们在Internet上定位文件和文档。还有一些资源发现工具正在被开发。

本章的最后简单浏览了另一个TCP/IP的重要客户程序，X窗口系统。我们看到X服务器管理一个显示器上的多个窗口，处理客户与其窗口的通信。每个客户都有它自己的与服务器的TCP连接，一个单个的服务器为一个给定的显示器管理着所有的客户。通过Xscope程序，我们看到怎样把一个程序放在一个客户与服务器之间，输出有关两者之间交换的报文的信息。

习题

30.1 试用Whois找到网络号为88的A类网络的拥有者。

30.2 试用Whois找到管理`whitehouse.gov`域的DNS服务器。这个应答与DNS给出的答案相匹配吗？

30.3 在图30-3中，你认为`xscope`进程必须和X服务器运行在同一台主机上吗？

附录A　tcpdump程序

tcpdump程序是由Van Jacobson、Craig Leres和Steven McCanne编写的，他们都来自加利福尼亚大学伯克利分校的劳伦斯伯克利实验室。本书中使用的是 2.2.1版（1992年6月）。

tcpdump通过将网络接口卡设置为混杂模式（ promiscuous mode）来截获经过网络接口的每一个分组。正常情况下，用于诸如以太网媒体的接口卡只截获送往特定接口地址或广播地址的链路层的帧（2.2节）。

底层的操作系统必须允许将一个接口设置成混杂模式，并且允许一个用户进程截获帧。下列的操作系统可以支持 tcpdump，或者可以加入对 tcpdump的支持：4.4BSD、BSD/386、SunOS、Ultrix和HP-UX。参考一下随着 tcpdump发布的README文件，了解它支持哪些操作系统以及哪些版本。

除了tcpdump还有其他一些选择。在图10-8中，我们使用了Solaris 2.2的程序snoop来查看一些分组。AIX 3.2.2提供了iptrace程序，该程序也提供了类似的功能。

A.1　BSD 分组过滤器

当前由BSD演变而来的 Unix内核提供了 BSD 分组过滤器 BPF (BSD Packet Filter)，tcpdump用它来截获和过滤来自一个被置为混杂模式的网络接口卡的分组。 BPF也可以工作在点对点的链路上，如 SLIP（2.4节），不需要什么特别的处理就可以截获所有通过接口的分组。BPF还可以工作在环回接口上（2.7节）。

> BPF有一个很长的历史。1980年卡耐基梅隆大学的Mike Accetta和Rick Rashid创造了Enet分组过滤程序。斯坦福的Jeffrey Mogul将代码移植到BSD，从1983年开始继续开发。从那以后，它演变为DEC的Ultrix分组过滤器、SunOS 4.1下的一个STREAMS NIT模块和BPF。劳伦斯伯克利实验室的Steven McCanne在1990年的夏天实现了BPF。其中很多设计来自于Van Jacobson。[McCanne and Jacobson 1993] 给出了最新版本的细节以及与Sun的NIT的一个比较。

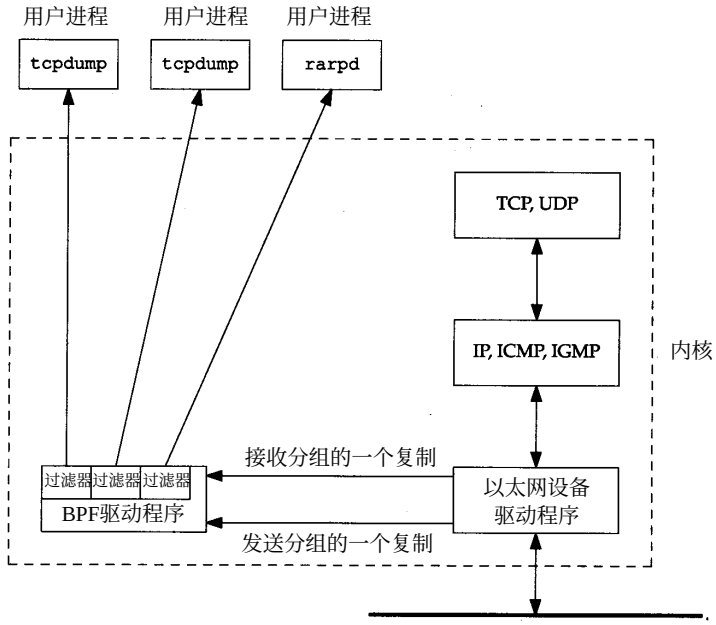
图A-1显示了用于以太网的BPF的特征。

BPF将以太网设备驱动程序设置为混杂模式，然后从驱动程序那里接收每一个收到的分组和传输的分组。这些分组要通过一个用户指明的过滤器，使得只有那些用户进程感兴趣的分组才会传递给用户进程。

多个进程可以同时监视一个接口，每个进程指明了一个自己的过滤器。图　A-1显示了tcpdump的两个实例进程和一个RARP守护进程（5.4节）监视同样的以太网接口。 tcpdump的每个实例指明了一个自己的过滤器。 tcpdump的过滤器可以由用户在命令行指明，而rarpd总是使用只截获RARP请求的过滤器。

除了指明一个过滤器， BPF的每个用户还指明了一个超时定时器的值。因为网络的数据传输率可以很容易地超过 CPU的处理能力，而且一个用户进程从内核中只读小块数据的代价昂

贵，因此，BPF试图将多个帧装载进一个读缓存，只有缓存满了或者用户指明的超时到期才将读缓存保存的帧返回。tcpdump将超时定时器置为1秒，因为它一般从BPF收到很多数据。而RARP守护进程收到的帧很少，所以rarpd将超时置为0（收到一个帧就返回）。



图A-1 BSD分组过滤器

用户指明的过滤器告诉BPF用户进程对什么帧感兴趣，过滤器是对一个假想机器的一组指令。这些指令被内核中的BPF过滤器解释。在内核中过滤，而不在用户进程中，减少了必须从内核传递到用户进程的数据量。RARP守护进程总是使用绑定在程序里的、同样的过滤程序。另一方面，tcpdump在每次运行时，让用户在命令行指明一个过滤表达式。tcpdump将用户指明的表达式转换为相应的BPF的指令序列。tcpdump表达式的例子如下：

```
% tcpdump tcp port 25
% tcpdump'icmp [0] != 8 and icmp[0] != 0
```

第一个只打印源端口和目的端口为25的TCP报文段。第二个只打印不是回送请求和回送应答的ICMP报文（也就是非ping的分组）。这个表达式指明了ICMP报文的第一个字节，图6-2中的type字段，不等于8或0，即图6-3中的回送请求和回送应答。正像你所看到的，设计过滤器需要有底层分组结构的知识。第二个例子中的表达式被放在一对单引号中，防止Unix外壳程序解释特殊字符。

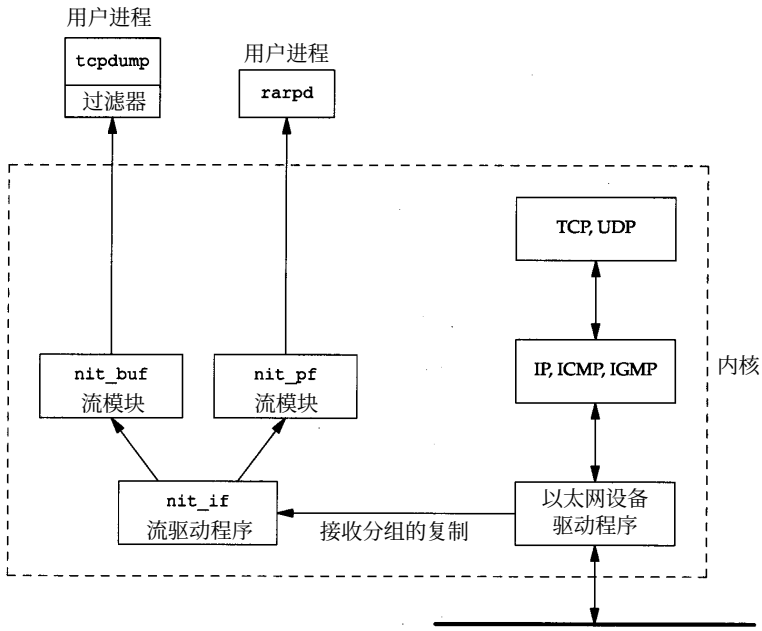
参考tcpdump (1)的手册，了解用户可以指明的表达式的全部细节。bpf(4)的手册详细描述了BPF使用的假想机器指令。[McCanne and Jacobson 1993]比较了这个假想机器方法与其他方法的设计与性能。

A.2 SunOS的网络接口分接头

SunOS 4.1.x提供了一个STREAMS伪设备驱动程序(pseudo-device driver)，称为网络接口分接头(Network Interface Tap)或者NIT（[Rago 1993]包含了流设备驱动程序的其他细节。我

们把这种特征叫作“流”)。NIT类似于BSD分组过滤器，但不如后者功能强大和效率高。图A-2显示了使用NIT所用到的流模块。这个图与图A-1之间的一个不同点在于BPF可以截获网络接口收到的和传送的分组，而NIT只能截获接口收到的分组。将tcpdump与NIT结合起来意味着我们只能看见由网络中其他主机发送来的分组——即根本不可能看见我们自己主机发送的分组（尽管BPF可以工作在SunOS 4.1.x上，但它需要对以太网设备驱动程序的源代码进行改变，大多数的用户没有权限访问源代码，因而这是不可能的)。

当设备/dev/nit被打开时，流驱动程序nit_if就会被打开。既然NIT是使用流来构造的，处理模块可以放在nit_if驱动程序之上。tcpdump将模块nit_buf放在STREAM之上。这个模块将多个网络帧聚集在一个读缓存中，允许用户进程指明一个超时的值。这种情况类似于我们在BPF中所描述的。RARP守护进程没有把这个模块放在它的流之上，因为它只处理了一小部分分组。



图A-2 SunOS的网络接口分接头

用户指明的过滤由流模块nit_pf处理。在图A-2中，我们注意到这个模块被RARP守护进程所用，但没有被tcpdump使用。在SunOS操作系统中，tcpdump代之以在用户进程中完成自己的过滤操作。这么做的理由是nit_pf使用的假想机器的指令与BPF所支持的指令不同(不如BPF所支持的功能强大)。这就意味着当用户对tcpdump指明了一个过滤表达式时，与BPF相比较，使用NIT就会有更多的数据在内核与用户进程之间交换。

A.3 SVR4数据链路提供者接口

SVR4支持数据链路提供者接口DLPI (Data Link Provider Interface)，它是OSI数据链路服务定义的一个流实现。SVR4的大多数版本支持第1版的DLPI，SVR4.2同时支持第1版和第2版，Sun的Solaris 2.x支持第2版，但是增强了一些功能。

像tcpdump的网络监视程序必须使用DLPI来直接访问数据链路设备驱动程序。在Solaris 2.x中，分组过滤的流模块被改名为pfmod，缓存模块被改名为bufmod。

尽管Solaris 2.x还很新，tcpdump在其上的一个实现有一天也会出现。Sun还提供了一个叫作snoop的程序，完成的功能类似于tcpdump（snoop代替了SunOS 4.x的程序etherfind）。作者还不知道tcpdump到vanilla SVR4上的任何端口实现。

A.4 tcpdump的输出

tcpdump的输出是“原始的”。在本书中包含它的输出时，我们对它进行了修改以便阅读。

首先，它总是输出它正在监听的网络接口的名字。我们把这一行给删去了。

其次，tcpdump输出的时间戳在一个微秒精度的系统中采用如同09:11:22.642008的格式，在一个10ms时钟精度的系统中则如同09:11:22.64一样（在附录B中，我们更多地讨论了计算机时钟的精度）。在任何一种情况下，HH:MM:SS的格式都不是我们想要的。我们感兴趣的是每个分组与开始监听的相对时间以及与下面分组的时间差。我们修改了输出以显示这两个时间差。第1个差值在微秒精度的系统中打印到十进制小数点后面6位（对于只有10 ms精度的系统打印到小数点后面2位），第2个差值打印到十进制小数点后面4位或2位（依赖于时钟精度）。

本书中大多数tcpdump的输出都是在sun主机上收集的，它提供了微秒精度。一些输出来自于运行0.9.4版BSD/386操作系统的主机bsdi，它只提供了10 ms的精度（如图5-1所示）。一些输出收集于当bsdi主机运行1.0版BSD/386时，后者提供了微秒级的精度。

tcpdump总是打印发送主机的名字，接着一个大于号，然后是目的主机的名字。这样显示很难追踪两个主机之间的分组流。尽管tcpdump输出仍然显示了数据流的方向，但我们经常把这条输出删掉，代替以产生一条时间线（在本书中的第一次出现是在图6-11）。在我们的时间线上，一个主机在左边，另一个在右边。这样很容易看出哪一边发送分组，哪一边接收分组。

我们给tcpdump的每条输出增加了行号，使得我们可以在书中引用特定的行。还在某些行之间增加了额外的空白，以区别一些不同的分组交换。

最后，tcpdump的输出可能会超出一页的宽度。我们在太长行的适当地方进行了换行。

作为一个例子，相应于图4-4的tcpdump的原始输出显示在图A-3中。这里假设了一个80列的终端窗口。

没有显示我们键入的中断键（用于中止tcpdump），也没有显示接收到的和漏掉的分组的个数（漏掉的分组是那些到达得太快，tcpdump来不及处理的分组。因为本文中的例子经常运行在另外一个空闲网络上，所以漏掉的分组个数总是0）。

```
sun % tcpdump -e
tcpdump: listening on le0
09:11:22.642008 0:0:c0:6f:2d:40 ff:ff:f f:ff:ff:ff arp 60: arp who-has svr4 tell
bsdi
09:11:22 .644182 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 arp 60: arp reply svr4 is-at 0:0
:c0:c2:9b:26
09:11:22.644839 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26  ip 60: bsdi.1030 > svr4.discard:
 S 596459521:596459521(0) win 4096 <mss 1024> [tos 0x10]
09:11:22. 649842 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 ip 60: svr4.discard > bsdi.1030:
 S 3562228225:3562228225(0) ack 596459522 win 4096 <mss 1024>
09:1 1:22.651623 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60: bsdi.1030 > svr4.discard:
 . ack 1 win 4096 [tos 0x10]
                                        我们没有显示其他4个分组
^?                                      键入中断字符来中断显示
9 packets received by filter
0 packets dropped by kernel
```

图A-3 图4-4的tcpdump 的输出

A.5 安全性考虑

很明显，截获网络中传输的数据流使我们可以看到很多不应该看到的东西。例如，Telnet和FTP用户输入的口令在网络中传输的内容和用户输入的一样（与口令的加密表示相比，这称为口令的明文表示。在Unix口令文件中，一般是/etc/passwd或/etc/shadow，存储的是加密的表示）。然而，很多时候一个网络管理员需要使用一个类似于 tcpdump的工具来分析网络中出现的问题。

我们是把tcpdump作为一个学习的工具，用来查看网络中实际传输的东西。对 tcpdump以及其他厂商提供的类似工具的访问权限依赖于具体系统。例如，在 SunOS，对NIT设备的访问只限于超级用户。 BSD的分组过滤器使用了一种不同的技术：通过对 /dev/bpfXX设备的授权来控制访问。一般来说，只有属主才能读写这些设备（属主应该是超级用户），对于同组用户是可读的（经常是系统管理组）。这就是说如果系统管理员不对程序设置用户的 ID，一般的用户是不能运行类似于tcpdump的程序的。

A.6 插口排错选项

查看一个TCP连接上发生的事情的另一种方法是使能插口排错选项，当然是在支持这一特征的系统中。这个特征只能工作在 TCP上（其他协议都不行），并且需要应用程序支持（当应用程序启动时，使能一个插口排错选项）。

大多数伯克利演变的实现都支持这个特征，包括SunOS、4.4BSD和SVR4。

程序使能了一个插口选项，内核就会保留在那个连接上发生的事情的一个痕迹记录。在这之后，所有记录的信息都可以使用 trpt(8)程序打印出来。使能一个插口排错选项不需要特别的许可，但是因为trpt程序访问了内核的内存，所以运行 trpt需要特别的权限。

sock程序（附录C）的-D选项支持这个特征，但是输出的信息比相应的 tcpdump的输出更难解析和理解。然而，我们在 21.4节确实使用它查看了TCP连接上tcpdump不能访问的内核变量。

附录B　计算机时钟

既然本书中的大多数的例子都需要测量一个时间间隔，我们需要更仔细地介绍一下当前Unix系统所采用的记录时间的方法。下面的描述适用于本书中例子所使用的系统，也适用于大多数的Unix系统。[Leffler et al. 1989]的3.4节和3.5节给出了另外的细节。

硬件按照一定的频率产生一个时钟中断。对于Sun SPARC和Intel 80386，时钟中断每10 ms产生一次。

应该注意到大多数的计算机使用一种无补偿的晶体振荡器来生成这些时钟中断。正如RFC1305 [Mills 1992] 的表7指出的，你不要想知道这种振荡器一天的偏差有多少。这就意味着几乎没有计算机能维持精确的时间（即，中断并不是精确地每 10 ms发生一次）。一个0.01%的误差就会产生一个每天 8.64秒的差错。为了得到更好的时间测量需要：（1）一个更好的振荡器；（2）一个外部的更精确的时间资源（如，全球定位卫星提供的时间资源）；或者（ 3）通过因特网访问一个具有更精确时钟的系统。后者通过 RFC1305定义的网络时间协议实现，对该协议的描述超出了本书的范围。

Unix系统中引起时间差错的另一个公共的原因是 10 ms的中断只是引起内核给一个记录时间的变量增 1。如果内核丢失了一个中断（也就是说两个连续中断之间间隔 10 ms对于内核来说太快了），时钟将失去 10 ms。丢失这种类型的中断经常引起 Unix系统丢失时间。

尽管时间中断近似于每10 ms到达一次，更新的系统，如SPARC，提供了一个更高精度的定时器来测量时间差异。通过NIT驱动程序，`tcpdump`（在附录A中描述）已经访问了这个更高精度的定时器。在 SPARC上，这个定时器提供了微秒级的精度。用户进程也可以通过`gettimeofday`(2)函数来访问这个更高精度的定时器。

作者做了下面的试验。我们运行了一个程序，这个程序在一个循环里调用了10 000次`gettimeofday`函数，并将每次的返回值保存在一个数组中。在循环结束后，打印了9999个时间差。对于一个SPARC ELC，时间差的分布如图B-1所示。

微　秒	次数
36	4 914
37	4 831
38	167
39	8
其他	79

图B-1　在SPARC ELC上调用`gettimeofday`函数10 000次所需要的时间分布

在一个空闲的系统中，运行这个程序所花的时钟时间为 `0.38`秒。根据这一点，我们可以说进程调用`gettimeofday`所花的时间大约`37`微秒。既然`ELC`的速度是`21 MIPS`（`MIPS`表示每秒`100`万指令），`37`微秒相应于大约`800`个指令。这些指令对于内核处理一个用户进程的调用、执行系统调用、复制 8个字节的结果及返回给用户进程看起来是合理的（ `MIPS`速度是不可靠的，很难测量当前系统的指令时间。我们试图做的只是得到一个粗略的估计来评价一下上面的值是否有意义）。

从这个简单的试验，我们可以说`gettimeofday`返回的值确实包含了微秒级的精度。

如果我们在SVR4/386上进行类似的测试，结果是不同的。这是因为很多 386 Unix系统，如SVR4，只计数10 ms的时钟中断，而没有提供更高的精度。图 B-2是运行在 25 MHz 80386上的SVR4中9999个时间差的分布。

微 秒	次 数
0	9 871
10 000	128

图B-2 在SVR4/386上调用gettimeofday函数10000次所需要的时间

这些值是无意义的，因为时间差一般小于 10 ms，都被认为是0了。在这些系统中，我们所能做的就是在一个空闲的系统上测量时钟时间，除以循环的次数。这个结果提供了一个上界，因为它包含了调用 `printf`函数9999次的时间和将结果写入一个文件的时间（在 SPARC的情况，图B-1，时间差没有包括`printf`的时间，因为所有10 000个值都是首先获得的，然后才打印结果）。在SVR4的时钟时间为3.15秒，每个系统调用消耗了315微秒。这个大约比SPARC慢8.5倍的系统调用时间看来是正确的。

BSD/386 1.0版提供了类似于SPARC的微秒级的精度。它读8253时钟寄存器，计算从上次时钟中断以来的微秒次数。调用`gettimeofday`的进程和内核模块，如BSD 分组过滤器，可以使用这个精度。

和`tcpdump`联系起来，这些数字意味着我们可以相信在 SPARC和BSD/386系统上打印的毫秒和亚毫秒(submillisecond)的值。而在SVR4/386上，`tcpdump`打印的值总是10 ms的倍数。对于其他打印往返时间的程序，如 `ping`（第7章）和`traceroute`（第8章），在SPARC和BSD/386系统上，我们可以相信它们输出的毫秒值，但在 SVR4/386上，打印的值总是10的倍数。为了在LAN上测量某个`ping`的时间，在第 7章中我们显示的时间是 3 ms，所以需要在SPARC和BSD/386系统上运行`ping`程序。

本书中的一些例子是运行在BSD/386 0.9.4版上，它类似于SVR4，只提供了10 ms的时钟精度。在我们显示这个系统的`tcpdump`输出时，只显示到小数点后面两位，因为这就是所提供的精度。

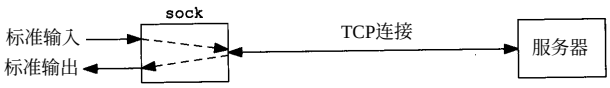
附录C　sock程序

在本书中一直使用一个称为sock的小测试程序，用来生成TCP和UDP数据。它既可以用作一个客户进程，也可以用作一个服务器进程。有这样一个可以从外壳程序执行的测试程序，使我们避免了为每一个我们想要研究的特征编写新的客户和服务器　C程序。因为本书的目的是了解网络互联协议，而不是网络编程，所以在这个附录中我们只描述这个程序和它不同的选项。

有很多与sock功能类似的程序。Juergen Nickelsen写了一个称为socket的程序，Dave Yost写了一个称为sockio的程序。两者都包含了很多类似的特征。sock程序的某些部分也受到了Mike Muuss和Terry Slattery所写的公开域ttcp程序的启发。

sock程序运行在以下四种模式之一：

1) 交互式客户：默认模式。程序和一个服务器相连，然后将标准输入的数据传给服务器，再将从服务器那里接收到的数据复制到标准输出。如图C-1所示。



图C-1　sock 程序作为交互式客户的默认操作

我们必须指明服务器主机的名字和想要连接的服务的名字。主机可指明为点分十进制数，服务可指明为一个整数的端口号。从sun到bsdi与标准的echo服务器（1.12节）相连，回显我们键入的每一个字符：

```
sun % sock bsdi echo
    a test line                    我们键入这一行
    a test line                    echo服务器返回一个复制行
    ^D                             键入文件结束符来中止
```

2) 交互式服务器：指明-s选项。需要指明服务名字（或端口号）：

```
sun % sock -s 5555                 作为一个在端口5555监听的服务器
```

程序等待一个客户的连接请求，然后将标准输入复制给客户，将从客户接收到的东西复制到标准输出。在命令行中，端口号之前可以有一个因特网地址，用来指明接收哪一个本地接口上的连接：

```
sun % sock -s 140.252.13.33 5555        只接受来自以太网的连接
```

默认的模式是接受任何一个本地接口上的连接请求。

3) 源客户：指明-i选项。在默认情况下，将一个1024字节的缓存写到网络中，写1024次。-n选项和-w选项可以改变默认值。例如，

```
sun % sock -i -n12 -w4096 bsdi discard
```

把12个缓存，每个包含4096字节的数据，送给主机bsdi上的discard服务器。

4) 接收器服务器：指明-i选项和-s选项。从网络中读数据然后扔掉。

这些例子都使用了TCP（默认情况），-u选项指明使用UDP。

sock程序有许多选项，用于对程序的运行提供更好的控制。我们需要使用这些选项来产

生本书中用到的所有测试条件。

-b	*n*	将*n*绑定为客户的本地端口号（在默认情况下，系统给客户分配一个临时的端口号）。
-c		将从标准输入读入的新行字符转换为一个回车符和一个换行符。类似地，当从网络中读数据时，将〈回车，换行〉序列转换为新行字符。很多因特网应用需要NVT ASCII（26.4节），它使用回车和换行来终止每一行。
-f	*a.b.c.d.p*	为一个UDP端点指明远端的IP地址（*a.b.c.d*）和远端的端口号（*p*）。
-h		实现TCP的半关闭机制（18.5节）。即，当在标准输入中读到一个文件结束符时并不终止。而是在TCP连接上发送一个半关闭报文，继续从网络中读报文直到对方关闭连接。
-i		源客户或接收器服务器。向网络写数据（默认），或者如果和 -s 选项一起用，从网络读数据。 -n选项可以指明写（或读）的缓存的数目， -w选项可以指明每次写的大小， -r 选项可以指明每次读的大小。
-n	*n*	当和-i 选项一起使用时， *n*指明了读或写的缓存的数目。 *n*的默认值是1024。
-p	*n*	指明每个读或写之间暂停的秒数。这个选项可以和源客户（- i）或接收器服务器（-is）一起使用作为每次对网络读写时的延迟。参考- P选项，实现在第1次读或写之前暂停。
-q	*n*	为TCP服务器指明挂起的连接队列的大小：TCP将为之进行排队的、已经接受的连接的数目（图18-23）。默认值是5。
-r	*n*	和-is选项一起使用， *n*指明每次从网络中读数据的大小。默认是每次读1024字节。
-s		作为一个服务器，而不是一个客户。
-u		使用UDP，而不是TCP。
-v		详细模式。在标准差错上打印附加的细节信息（如客户和服务器的临时端口号）。
-w	*n*	和-i选项一起使用， *n*指明每次从网络中写数据的大小。默认值是每次写1024字节。
-A		使能 SO_REUSEADDR插口选项。对于TCP，这个选项允许进程给自己分配一个处于2MSL等待的连接的端口号。对于UDP，这个选项支持多播，它允许多个进程使用同一个本地端口来接收广播或多播的数据报。
-B		使能SO_BROADCAST插口选项，允许向一个广播 IP地址发送UDP数据报。
-D		使能SO_DEBUG插口选项。这个选项使得内核为这个 TCP连接维护另外的调试信息（A.6节）。以后可以运行trpt(8)程序输出这个信息。
-E		如果实现支持，使能 IP_RECVDSTADDR插口选项。这个选项用于

UDP服务器，用来打印接收到的UDP数据报的目的IP地址。

-F 指明一个并发的TCP服务器。即，服务器使用fork函数为每一个客户连接创建一个新的进程。

-K 使能TCP的SO_KEEPALIVE插口选项（第23章）。

-L *n* 把一个TCP端点的拖延时间(linger time)（SO_LINGER）设置为*n*。一个为0的拖延时间意味着当网络连接关闭时，正在排队等着发送的任何数据都被丢弃，向对方发送一个重置报文（ 18.7节）。一个正的拖延时间（百分之一秒）是关闭网络连接必须等待的将所有正在排队等着发送的数据发送完并收到确认的时间。关闭网络连接时，如果这个拖延定时器超时，挂起的数据没有全部发送完并收到确认，关闭操作将返回一个差错信息。

-N 设置TCP_NODELAY插口选项来禁止Nagle算法（19.4节）。

-O *n* 指明一个TCP服务器在接受第一个客户连接之前暂停的秒数。

-P *n* 指明在第一次对网络进行读或写之前暂停的秒数。这个选项可以和接收器服务器（- is）一起使用，完成在接受了客户的连接请求之后但在执行从网络中第一次读之前的延迟。和接收源（- i）一起使用时，完成连接建立之后但第一次向网络写之前的延迟。参看- p选项，实现在接下来的每一次读或写之间进行暂停。

-Q *n* 指明当一个TCP客户或服务器收到了另一端发来的一个文件结束符，在它关闭自己这一端的连接之前需要暂停的秒数。

-R *n* 把插口的接收缓存（SO_RCVBUF插口选项）设置为*n*。这可以直接影响TCP通告的接收窗口的大小。对于 UDP，这个选项指明了可以接收的最大的UDP数据报。

-S *n* 把插口的发送缓存（SO_SNDBUF插口选项）设置为*n*。对于UDP，这个选项指明了可以发送的最大的UDP数据报。

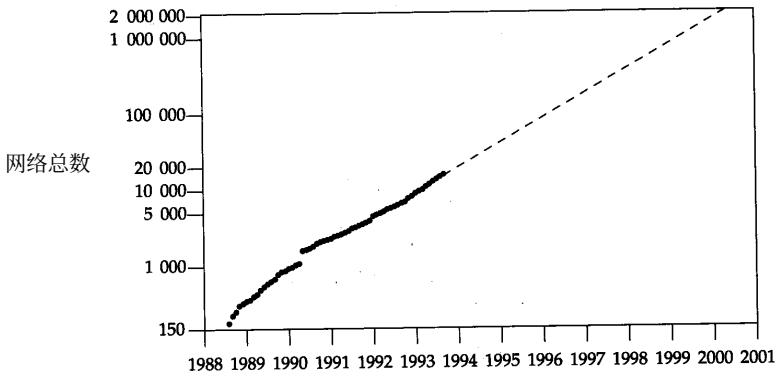
-U *n* 在向网络写了数字*n*后进入TCP的紧急模式。写一个字节的数据以启动紧急模式（20.8节）。

附录D　部分习题的解答

第1章

1.1　答案是：2^7-2（126）＋$2^{14}-2$（16 382）＋$2^{21}-2$（2 097 150）= 2 113 658。每一部分都减去2是因为全0或全1网络ID是非法的。

1.2　图D-1显示了直到1993年8月的有关数据。



图D-1　宣布加入NSFNET的网络数

如果网络数继续呈指数增长的话，虚线估计了2000年可能达到的最大的网络数。

1.3　“自由地接收，保守地发送。”

第3章

3.1　不，任何网络ID为127的A类地址都是可行的，尽管大多数系统使用了127.0.0.1。

3.2　kpno有5个接口：3个点对点链路和2个以太网接口。R10有4个以太网接口。gateway有3个接口：2个点对点链路和1个以太网接口。最后，netb有1个以太网接口和2个点对点链路。

3.3　没有区别：作为一个没有再区分子网的C类地址，它们都有一个255.255.255.0的子网掩码。

3.5　它是合法的，被称为非连续的子网掩码，因为其用于子网掩码的16位是不连续的。但是RFC建议反对使用非连续的子网掩码。

3.6　这是一个历史遗留问题。值是1024＋512，但是打印的MTU值包含了所有需要的首部字节数。Solaris 2.2将回环接口的MTU设置为8232（8192＋40），其中包含了8192字节的用户数据加上20字节的IP首部和20字节的TCP首部。

3.7　第一，数据报降低了路由器中对于连接状态的需求。第二，数据报提供了基本的构件，在它的上面可以构造不可靠的（UDP）和可靠的（TCP）的运输层。第三，数据报代表

了最小的网络层假定，使得可以使用很大范围的数据链路层服务。

第4章

4.1 发出一条rsh命令与另一台主机建立一个TCP连接。这样做引起在两个主机之间交换IP数据报。为此，在那台主机的ARP缓存中必须有我们这台主机的登记项。因此，即使在执行rsh命令之前，ARP缓存是空的，当rsh服务器执行arp命令时，必须保证ARP缓存中登记有我们这台主机。

4.2 保证你的主机上的ARP缓存中没有登记以太网上的某个叫作foo的主机。保证foo引导时发送一个免费ARP请求，也许是在foo引导时，在那台主机上运行tcpdump。然后关闭主机foo，使用说明了temp选项的arp命令，在你的系统的ARP缓存中为foo输入一个不正确的登记项。引导foo并在它启动好之后，察看主机的ARP缓存，看看不正确的登记项是不是已经被更正了。

4.3 阅读主机需求（Host Requirement）RFC的2.3.2.2节和本书中的11.9节。

4.4 假设当服务器关闭时，客户机保存了关于服务器的一个完整的ARP登记项。如果我们继续试图与（已关闭的）服务器联系，过了20分钟以后，ARP将超时。最后，当服务器以另一个新的硬件地址重启动。如果它没有发出一个免费ARP，旧的、不再正确的ARP登记项仍然存在于客户机上。我们将无法和在新硬件地址上的服务器联系直到我们手工删除这个ARP登记项，或者在20分钟内停止与服务器联系的尝试。

第5章

5.1 一个单独的帧类型并不是必需的，因为图4-3中的op字段对于所有的四个操作（ARP请求、ARP应答、RARP请求和RARP应答）都有一个不同的值。但是实现一个RARP服务器，独立于内核中的ARP服务器，更容易处理不同的帧类型字段。

5.2 每个RARP服务器在发送一个响应之前可以延迟一个小的随机时间。

作为一个优化，可以指定一个RARP服务器为主服务器，其他的为次服务器。主服务器发出响应不需要延迟，而次服务器发出响应则需要一个随机的延迟。

作为另一个优化，也是指定一个主RARP服务器，其他为次服务器。次服务器只对在一个短时间段内发生的重复请求进行响应。这里假设出现重复请求的原因是由于主服务器停机了。

第6章

6.1 如果在局域网线上有一百个主机，每个都可能在同一时刻发送一个ICMP端口不可达的报文。很多报文的传输都可能发生冲突（如果使用的是以太网），这将导致1秒或2秒的时间里网络不可用。

6.2 它是一个“should”。

6.3 如我们在图3-2所指出的，发送一个ICMP差错总是将TOS置为0。发送一个ICMP查询请求可以将TOS置为任何值，但是发送相应的应答必须将TOS置为相同的值。

6.4 netstat -s是查看每个协议统计数据的常用方法。在一台收到了4800万个IP数据报的SunOS 4.1.1主机（gemini）上，ICMP的统计为：

```
Output histogram:
        echo reply: 1757
        destination unreachable: 700
        time stamp reply: 1
Input histogram:
        echo reply: 211
        destination unreachable: 3071
        source quench: 249
        routing redirect: 2789
        echo: 1757
        #10: 21
        time exceeded: 56
        time stamp: 1
```

21个类型为10的报文是SunOS 4.1.1不支持的路由器的请求。

也可以使用 SNMP（图25-26），有些系统，如 Solaris 2.2，可以生成使用 SNMP变量名的netstat-s的输出。

第7章

7.2 86字节除以960字节/秒，乘以2得到179.2 ms。当以这个速率运行ping时，打印的值为180 ms。

7.3 （86＋48）除以960字节/秒，乘以2得到279.2 ms。另外的48字节是因为56字节的数据部分的最后48字节必须忽略：0xc0是SLIP END字符。

7.4 CSLIP只压缩了TCP报文段的TCP首部和IP首部。它对ping使用的ICMP报文没有作用。

7.5 在一个SPARC工作站 ELC上，对回环地址的ping操作产生一个1.310 ms的RTT，而对一个主机的以太网地址的ping操作产生一个1.460 ms的RTT。这个差值是由于以太网驱动程序需要时间来判定这个数据报的目的地址是一个本地的主机。需要一个产生微秒级输出的ping来验证这一点。

第8章

8.1 如果一个输入数据报的 TTL为0，做减一操作然后测试会将把 TTL设置为255，并且让数据报继续传输。尽管一个路由器永远不会收到一个 TTL为0的数据报，但这种情况确实会发生。

8.2 我们注意到traceroute在UDP数据报的数据部分存储了 12个字节，其中包含了数据报发送的时间。然而，从图6-9可以看出ICMP只返回了出错的IP数据报的头8个字节，实际上这是8个字节的UDP首部。因此，ICMP的差错报文没有返回traceroute存储的时间值。traceroute保存了它发送分组的时间，当收到一个ICMP应答时，取出当时的时间，把两个值相减就可以得出RTT。

回忆一下第7章中，ping在输出的ICMP回显请求中存储了时间，这个值被服务器回显了回来。这样即使分组返回时失序，ping也能打印出正确的RTT。

8.3 第1行输出是正确的，并且标识了 R1。下一个探测分组启动时将 TTL置为2，并且这个值被R1减1。当R2收到这个分组时，把 TTL从1减为0，但是错误地将它传递给了 R3。R3看见进入的TTL是0就将超时的分组发送回来。这就意味着第2行输出（TTL为2）标识了R3，而不是R2。第3行输出正确地标识了 R3。这个错误所表现出来的线索就是两个连续的输

出行标识了同一个路由器。

8.4 在这种情况下，TTL为1标识了R1，TTL为2标识了R2，TTL为3标识了R3，但是当TTL为4时，UDP数据报到达了目的地，其输入的TTL为1。ICMP端口不可达报文生成了，但它的TTL是1（错误地从进入的TTL复制而来）。这个ICMP报文到了R3，在那儿TTL被减1，报文被丢弃。没有生成一个ICMP超时报文，因为被丢弃的数据报是一个ICMP的差错报文（端口不可达）。类似的现象也出现在TTL为5的探测分组，但这次输出的端口不可达报文以TTL为2开始（进入的TTL），这个报文被传给R2，在那儿被丢弃。对应于TTL为6探测分组的端口不可达报文被传递给R1，在那儿被丢弃。最后，对应于TTL为7探测分组的端口不可达报文被送回了原地，到达时它的TTL为1（traceroute认为一个TTL为0或1的到达ICMP报文是有问题的，因此它在RTT后面打印了一个惊叹号）。总之，TTL为1、2和3的行正确地标识了R1、R2和R3，接下来的三行每个都包含三个超时，再接下来的TTL为7的行标识了目的地。

8.5 它表明这些路由器都将一个ICMP报文的输出TTL设置为255，这是共同的。从netb输入的255的TTL值是我们想要的，而从butch输入的253的TTL值表明在butch和netb之间可能有一个未觉察的路由器。否则，在这个点上我们应该看到一个TTL值为254的输入报文。类似地，我们希望看到一个值为252而不是249的、来自enss142.UT.westnet.net的报文。这表明这些未觉察的路由器没有正确地处理向外输出的UDP数据报，但它们都对返回的ICMP报文正确地进行了TTL减1操作。

我们必须在查看输入的TTL时非常小心，因为有时候一个和我们想要的不同的值可能是由于返回的ICMP报文采用了一条与输出UDP数据报不同的路径。但是，在这个例子中证实了我们所怀疑的——当使用松源路由选项时，确实存在traceroute没有发现的未察觉的路由器。

8.7 ping的客户把ICMP回显请求报文（图7-1）的标识符字段设置为它的进程ID。ICMP回显应答报文包含同样值的标识符字段。每个客户都要查看这个返回的标识符字段，并且只处理那些它发送过的报文。

traceroute客户将它的UDP源端口号设置为它的进程ID和32768的逻辑或。因为返回的ICMP报文总是包含产生错误的IP数据报的前8个字节（图6-9），这8个字节包括了完整的UDP首部，所以这个源端口号在ICMP差错报文中被返回。

8.8 ping客户将ICMP回显请求报文的可选数据部分设置为分组发送的时间。这个可选的数据必须在ICMP回显应答中返回。这样使得即使分组返回时失序，客户也能计算出精确的回环时间。

traceroute客户不能这样操作，因为在ICMP差错报文中返回的只是UDP首部（图6-9），没有UDP数据。因此，traceroute必须记住它发送一个请求的时间，等待应答，然后计算两者的时间差。

这里显示了ping和traceroute的另一个不同点：ping每秒发送一个分组，而不管是否收到任何应答；traceroute发送一个请求，然后在发送下一个请求前等待一个应答或者一个超时。

8.9 因为默认情况下Solaris 2.2从32768开始使用临时的UDP端口，所以目的主机上的目的端口已经被使用的机会更大。

第9章

9.1 当ICMP标准第1次发布时，RFC 792 [Postel 1981b]所述的划分子网技术还没有使用。另外，使用一个网络重定向而不是N个主机重定向（对于目的网络中的所有*N*个主机）也节省了路由表的空间。

9.2 这一项并不需要，但是如果把它删除了，所有到slip的IP数据报将被发送到默认的路由器（sun），后者又将把它们送到路由器bsdi。既然sun将数据报从与接收数据报相同的接口转发出去，它把一个ICMP重定向到svr4。这样在svr4中又创建了我们删除过的同样的路由表项，尽管这一次是因为重定向而创建的，而不是在引导时增加的。

9.3 当那个4.2BSD主机收到目的地址是140.1.255.255的数据报，发现它有一个通往该网络（140.1）的路由，因此就试图转发数据报。它发送一个ARP广播来寻找140.1.255.255。这个ARP请求没有收到任何应答，所以这个数据报最终被丢弃。如果在网线上有很多这样的4.2BSD主机，每一个都在差不多同一时刻发送ARP这个广播，将会暂时地阻塞网络。

9.4 这次，每一个ARP请求都收到一个应答，告诉每个4.2BSD主机向一个指定的硬件地址（以太网广播）发送数据报。如果网线上有*k*个这样的4.2BSD主机，全部收到了它们自己的ARP应答，使得每一个生成了另一个广播。每个主机都收到了每一个目的地址为140.1.255.255的广播IP数据报，既然现在每个主机都有一个ARP缓存项，这个数据报又被转发给了广播地址。这个过程继续下去，就会产生一次以太网的熔毁（Ethernet meltdown）。[Manber 1990] 描述了网络中另一种形式的链式反应。

第10章

10.1 路由表中有13条来自于kpno：除了140.252.101.0和140.252.104.0之外的所有gateway直接相连的其他网络。

10.2 丢失的数据报中通告的25条路由需要60秒才能得到更新。这不成问题，因为一般来说一条路由如果连续3分钟没有得到更新，RIP才会声明它失效。

10.3 RIP运行在UDP上，而UDP提供了UDP数据报中数据部分的一个可选的检验和（11.3节）。然而，OSPF运行在IP上，IP的检验和只覆盖了IP首部，所以OSPF必须增加它自己的检验和字段。

10.4 负载平衡增加了分组被失序交付的机会，并且很可能使得运输层计算的环回时间出错。

10.5 这叫作简单的分裂范围（split horizon）。

10.6 在图12-1中，我们显示了100个主机的每一个都通过设备驱动程序、IP层和UDP层来处理这个广播的UDP数据报。当它们发现UDP端口520没有被使用时，这个广播数据报才最终被丢弃。

第11章

11.1 因为使用IEEE 802封装时，存在8个额外的首部字节，所以1465个字节的用户数据是引起分片的最小长度。

11.3 对于IP来说有8200字节的数据需要发送，8192字节的用户数据和8个字节的UDP首部。

采用tcpdump记号，第1个分片是1480@0+（1480字节的数据，偏移为0，将“更多片”比特置1）。第2个是1480@1480+，第3个是1480@2960+，第4个是1480@4440+，第5个是1480@5920+，第6个是800@7400。1480×5 + 800 = 8200，正好是要发送的字节。

11.4 每个1480字节的数据报片被分成三小片：两个528字节和一个424字节。小于532（552-20）的8的最大倍数是528。800字节的数据报片被分成两小片：一个528字节和一个272字节。这样，原来8192字节的数据报变成了SLIP链路上的17个帧。

11.5 不。问题是当应用程序超时重传时，重传产生的IP数据报有一个新的标识字段。而重新装配只针对那些具有相同标识字段的分段。

11.6 IP首部中的标识字段（47942）是一样的。

11.7 第一，从图11-4我们看到gemini没有使能输出UDP的检验和。如果输出UDP的检验和没有被使能，这个主机上的操作系统（SunOS 4.1.1）就不会验证一个进入UDP的检验和。第二，大多数的UDP通信量都是本地的，而不是WAN的，因此没有服从所有的WAN特征。

11.8 不严格的和严格的源站选路选项被复制到每一个数据报片中。时间戳选项和记录路由选项没有被复制到每一个数据报片中——它们只出现在第1个数据报片中。

11.9 不。在11.12节中，我们看到很多实现可以根据目的IP地址、源IP地址和源端口号来过滤送往一个给定UDP端口号的输入数据报。

第12章

12.1 广播本身不会增加网络通信量，但它增加了额外的主机处理时间。如果接收主机不正确地响应了诸如ICMP端口不可达之类的差错，那么广播也可能导致额外的网络通信量。路由器一般不转发广播分组，而网桥一般转发，所以在一个桥接网络上的广播分组可能比在一个路由网络上走得更远。

12.2 每个主机都收到了所有广播分组的一个副本。接口层收到了帧，把它传递给设备驱动程序。如果类型字段指的是其他协议，设备驱动程序就会丢弃该帧。

12.3 首先执行netstat -r来看一下路由表，结果显示了所有接口的名字。然后对每个接口执行ifconfig（3.8节）：标志指出了一个接口是否支持广播，如果支持，相应的广播地址也会被输出。

12.4 伯克利演变的实现不允许对一个广播数据报进行分片。当我们说明了1472字节的长度，产生的IP数据报将是1500字节，正好是以太网的MTU。不允许分片一个广播数据报是一个策略上的决定——没有技术上的原因（并不是想要减少广播分组的数目）。

12.5 依赖于100个主机上不同的以太网接口卡的多播支持，多播数据报可能被接口卡忽略，或者被设备驱动程序丢弃。

第13章

13.1 生成随机数时要使用对于主机唯一的值。IP地址和链路层地址是每个主机都应该不一样的两个值。日期时间是一个不好的选择，尤其是在所有的主机都运行了一个类似于NTP的协议来同步它们的时钟的情况下。

13.2 他们增加了一个包括一个序号和一个时间戳的应用协议首部。

第14章

14.1 一个解析器总是一个客户，但一个名字服务器既是一个客户又是一个服务器。

14.2 问题被返回，它占用了前44个字节。一个回答占用了剩下来的31个字节：2个字节指向域名的指针（即，指向问题中域名的一个指针），10字节固定长度的字段（类型、种类、TTL和资源长度），19字节的资源数据（一个域名）。注意到资源数据中的域名（svr4.tuc.noao.edu.）没有共享问题（34.13.252.140.in-addr.arpa.）中域名的后缀，所以不能使用一个指针。

14.3 将顺序颠倒意味着首先使用DNS，如果使用DNS失败，然后才将参数翻转过来作为一个点分十进制数。这就是说每次说明一个点分十进制数，都要使用DNS，涉及一个名字服务器。这是对资源的一种浪费。

14.4 RFC 1035的4.2.2节说明了在实际的DNS报文之前的两个字节长度的字段。

14.5 当一个名字服务器启动时，它一般从一个磁盘文件中读出一个根服务器列表（可能已经过时了）。然后尝试和这些根服务器中的一个联系，请求根域的名字服务器记录（一个NS的查询类型）。这个请求返回了当前最新的根服务器列表。启动磁盘文件中根服务器项中至少需要一个是有效的。

14.6 InterNIC的注册服务每一周更新三次根服务器。

14.7 就像应用是不定的一样，解析器也是不定的。如果系统配置成使用多个名字服务器，而且解析器是无状态的，那么解析器就不能记住不同的名字服务器的往返时间。这样定时太短的解析器将会超时，引起不必要的重传。

14.8 对A记录的排序应该由解析器来执行，而不是名字服务器，因为解析器一般比服务器了解更多的客户的网络拓扑（更新版本的BIND提供了解析器对A记录排序的功能）。

第15章

15.1 送往广播地址的TFTP请求应该被忽略。正像Host Requirements RFC所描述的，对一个广播请求的响应可能产生一个非常严重的安全漏洞。但是，问题是并不是所有的实现和API都对接收一个UDP数据报的进程提供了该数据报的目的地址（11.12节）。因为这个原因，很多TFTP服务器没有严格遵守这个限制。

15.2 不幸的是，RFC没有提到这个块数目环绕问题。具体实现时应该能够传输最大为33 553 920（65535×512）字节的文件。但是当文件的长度超过16 776 704（32767×512）时，很多实现都会失败，因为它们将块数目错误地表示为一个有符号的16位整数，而不是一个无符号的整数。

15.3 这样简化了编写一个适合于只读内存的TFTP客户的工作，因为服务器是引导文件的发送者，所以服务器必须实现超时和重传机制。

15.4 利用它的停止等待协议，TFTP可以在每一次客户与服务器的往返过程中最多传输512字节的数据。TFTP的最大吞吐量就是512字节除以客户与服务器之间的往返时间。在以太网上，假设一个往返时间为3 ms，那么最大的吞吐量就是大约170 000字节/秒。

第16章

16.1 一个路由器可以转发一个RARP请求到路由器连接的其他网络上的任何一台主机上。但

是发送应答就成问题了，路由器还必须转发RARP应答。

BOOTP没有这个应答问题，因为应答的地址是路由器知道如何转发的一般IP地址。问题是RARP只使用了链路层地址，路由器一般不知道在其他的、没有连接在路由器的网络上主机的链路层地址。

16.2 它可能使用了自己的硬件地址。该地址应该是唯一的，在请求报文中设置，在应答中返回。

第17章

17.1 除了UDP的检验和，其他都是必需的。IP检验和只覆盖了IP首部，而其他字段都紧接着IP首部开始。

17.2 源IP地址、源端口号或者协议字段可能被破坏了。

17.3 很多Internet应用使用一个回车和换行来标记每个应用记录的结束。这是NVT ASCII采用的编码（26.4节）。另外一种技术是在每个记录之前加上一个记录的字节计数，DNS（习题14.4）和Sun RPC（29.2节）采用了这种技术。

17.4 就像我们在6.5节所看到的，一个ICMP差错报文必须至少返回引起差错的IP数据报中除了IP首部的前8个字节。当TCP收到一个ICMP差错报文时，它需要检查两个端口号以决定差错对应于哪个连接。因此，端口号必须包含在TCP首部的前8个字节里。

17.5 TCP首部的最后有一些选项，但UDP首部中没有选项。

第18章

18.1 ISN是一个32 bit的计数器，它在系统引导大约9.5小时后从4 294 912 000环绕到8704。再过大约9.5小时它将环绕到17 408，然后再过9.5小时是26 112，如此继续下去。因为系统引导时ISN从1开始，并且因为最低次序的数字在4、8、2、6和0之间循环，所以ISN应该总是一个奇数。

18.2 在第1种情况下，我们使用了sock程序。默认情况下它把Unix的新行字符不作改变地进行传输——单个ASCII字符012（八进制）。在第2种情况下，我们使用了Telnet客户，它把Unix的新行字符转变为两个ASCII字符——一个回车符（八进制015）跟着一个换行符（八进制012）。

18.3 在一个半关闭的连接上，一个端点已经发送了一个FIN，正等待另一端的数据或者一个FIN。一个半打开的连接是当一个端点崩溃了，而另一端还不知道的情况。

18.4 一个连接只有经过了已建立状态才能进入2MSL等待状态。

18.5 首先，日期服务器在将时间和日期写给客户之后对TCP连接做一个主动关闭。这可以通过sock程序打印的消息："connection closed by peer."表现出来。连接的客户端经历了被动关闭的状态。这样就把一对插口置于服务器端的TIME_WAIT状态，而不是在客户端。

其次，正如在18.6节所示，大多数伯克利演变的实现都允许一个新的连接请求到达一个正处于TIME_WAIT状态的一对插口，这也就是这里所发生的情况。

18.6 因为在一个已经关闭的连接上到达了一个FIN，所以相应于这个FIN发送了一个复位。

18.7 拨号的一方做主动打开，电话振铃的一方做被动打开。不允许同时打开，但允许同时关

闭。

18.8 我们将只看到ARP请求，而不是TCP SYN报文段，但ARP请求将和图中具有相同的计时。

18.9 客户在主机solaris上，服务器在主机bsdi上。客户对服务器SYN的确认（ACK）和客户的第一个数据段结合在一起（第3行）。这种处理非常符合TCP的规则，尽管大多数的实现都没有这么做。接着，客户在等待它的数据的确认之前发送了它的 FIN（第4行）。这样使得服务器可以在第5行同时确认客户数据和它的FIN。

这次交互（将一个报文段从客户发送到服务器）需要 7个报文段，而正常的连接建立和终止（图18-13），以及一个数据段和它的确认，需要9个报文段。

18.10 首先，服务器对客户的 FIN的确认一般不会被延迟（我们在 19.3节讨论延迟的确认），而是在FIN到达后立即发送。应用进程需要一些时间来接收 EOF，告诉它的TCP关闭它这一端的连接。第二，服务器收到客户的FIN后，并不一定要关闭它这一端的连接。就像我们在18.5节中看到的，仍然可以发送数据。

18.11 如果一个产生RST的到达报文段有一个 ACK字段，那么RST的序号就是到达的ACK字段。第6行中值为1的ACK是相对于第2行中的26368001的ISN。

18.12 参见 [Crowcroft et al. 1992]中对分层的评论。

18.13 发出了5个查询。假设有3个分组用于建立连接， 1个用于查询，1个用于确认查询，1个用于响应， 1个用于确认响应， 4个用于终止连接。这就是说每次查询需要 11个分组，总共需要55个分组。使用UDP则可以减少到10个分组。

如果对查询的确认和响应结合在一起，每个查询需要的分组可以减少到 10个（19.3节）。

18.14 限制是每秒268个连接：最大数目的TCP端口号（65536 - 1024 = 64512，忽略知名端口）除以TIME_WAIT状态的2MSL。

18.15 重复的FIN会得到确认， 2MSL定时器重新开始。

18.16 在TIME_WAIT状态中收到一个RST引起状态过早地终止。这就叫作 TIME_WAIT断开(assassination)。RFC1337 [Braden 1992a]仔细讨论了这个现象，并显示了潜在的问题。这个RFC提出的简单的修改就是在TIME_WAIT状态时忽略RST段。

18.17 它是在具体实现不支持半关闭连接的时候。一旦应用进程引起发送一个 FIN，应用进程就不能再从这个连接读数据了。

18.18 不。输入的数据段通过源 IP地址、源端口号、目的 IP地址和目的端口号进行区分。在18.11节中我们看到对于输入的连接请求，一个 TCP服务器一般可以通过目的 IP地址来拒绝接收。

第19章

19.1 应用程序的两个写操作，跟着一个读操作，引起了迟延，因为 Nagle算法很可能被激活。第一个报文段（包含8个字节的数据）被发送后，在发送后面 12个字节的数据之前必须等待第一个报文段的确认。如果服务器实现了延迟的确认，在收到这个确认之前，可能会有一个达到200 ms的时延（加上RTT）。

19.2 假设5个字节的CSLIP首部（IP和TCP）和两个字节的数据，这些段通过 SLIP链路的RTT大约是14.5 ms。我们要加上通过以太网的 RTT（一般是5~10 ms），加上在sun和bsdi

上的选路时间。因此，观察到的时间看起来是正确的。

19.3 在图19-6中，第6和第9报文段之间的时间是533ms。在图19-8中，第8和第12报文段之间的时间是272ms（我们测量了F2键的时间，而不是F1键，因为F1键的第1个回显在第2个图中丢掉了）。

第20章

20.1 字节号0是SYN，字节号8193是FIN。SYN和FIN在序号空间里各占用了一个字节。

20.2 应用程序的第1个写操作引起置位 PUSH标志发送第1个报文段。因为 BSD/386总是使用慢启动，它在发送更多数据之前等待第1个确认。在这段时间里，下面三个写操作发生了，发送TCP把要发送的数据缓存了起来。因为在缓存中有更多的数据要发送，下面的三个报文段没有包含PUSH标志。最后，慢启动跟上了应用程序的写操作，每个应用程序的写操作引起了发送一个报文段，并且因为那个报文段是缓存中最后的一个，所以设置PUSH标志。

20.3 为容量求解带宽迟延方程式，第一种情况是 1920字节，卫星的情况是2062字节。看起来TCP只声明了一个2048字节的窗口。

一个大于16000字节的窗口应该能够使卫星链路饱和。

20.4 不，因为TCP超时之后可能重新对数据进行分组，就像我们将在 21.11节中看到的。

20.5 作为应用进程读数据的结果，第 15报文段是TCP模块自动发送的窗口更新，它引起窗口的打开。这类似于图中的第 9报文段。但是，第 16报文段是应用进程关闭它这一端连接的结果。

20.6 这可能引起发送者以比网络实际能够处理的更快的速率向网络发送分组。这叫作确认压缩(ACK compression)或确认粉碎(ACK smashing) [Mogul 1993, 15.8.13节]。这个引用显示了在Internet上发生的ACK压缩，尽管它很少会导致拥塞。

第21章

21.1 下一个超时设定是48秒：0＋4×12。因子4是指数退避的下一个乘数。

21.2 看起来SVR4在计算*RTO*时仍然使用了因子2*D*，而不是4*D*。

21.3 TFTP使用的停止等待协议被限制为每个往返过程传送 512字节的数据。32768/512×1.5是96秒。

21.4 显示了4个报文段，编号为1、2、3和4。假设接收的顺序是1、3、2和4，接收者产生的确认将是ACK 1（一个正常的确认），ACK 1（当收到了报文段3，失序后一个重复的确认），当收到了报文段2后的ACK 3（对报文段2和报文段3的确认），然后是ACK 4。这儿产生了一个重复的确认。如果接收的顺序是1、3、4和2，将会产生两个重复的确认。

21.5 不，因为斜率仍然是向上和向右，不是向下。

21.6 参见图E-1。

21.7 在图21-2中，报文段包含256个字节的数据，在`slip`和`bsdi`之间的9600 b/s的CSLIP链路传输大约需要250 ms。假定数据段没有在`bsdi`和`vangogh`之间的某个地方排队，它们分别需要大约250 ms到达`vangogh`。因为这个时间超过了延迟确认定时器的 200 ms时间，当下一个延迟确认定时器超时时，每个报文段得到确认。

第22章

22.1 主机bsdi上的确认很可能都要被延迟，因为没有理由立即发送它们。这就是为什么相对次数有一个0.170和0.370的小数部分。看起来bsdi上200 ms的定时器在sun上同样的定时器之后18 ms才开始计时。

22.2 FIN标志，和SYN标志一样，在序号空间占据了 1个字节。通知的窗口看起来小了一个字节，因为TCP允许FIN标志在序号空间占用1个字节的空间。

第23章

23.1 通常激活keepalive选项比显式地编写应用程序探测报文更容易； keepalive探测报文比应用程序探测报文占用更少的网络带宽（因为 keepalive探测报文和应答不包含任何数据)；如果连接不是空闲的，就不会发送探测报文。

23.2 keepalive选项可能会由于一个临时性的网络中断而引起一个非常好的连接断开；发送探测报文的间隔（2小时）一般不可以根据应用程序进行配置。

第24章

24.1 它意味着发送TCP支持窗口扩缩选项，但这个连接并不需要扩缩它的窗口。另一端（接收这个SYN的）可以说明一个窗口扩缩因子（可以是 0或非0）。

24.2 64 000：接收缓存大小（128 000）向右移1位。55 000：接收缓存（220 000）向右移2位。

24.3 不。问题是确认没有可靠地交付（除非它们被数据捎带在一起发送），因此，一个确认上的扩缩改变可能会丢失。

24.4 $2^{32} \times 8/120$等于286 Mb/s，2.86倍于FDDI数据率。

24.5 每个TCP将不得不记住从每个主机的任何一个连接上收到的上一个时间戳。阅读 RFC 1323的附录B.2以了解进一步的细节。

24.6 应用程序必须在和另一端建立连接之前设置接收缓存的大小，因为窗口扩缩选项在初始的SYN段中发送。

24.7 如果接收者每次确认第2个数据报文段，吞吐量是 1 118 881字节/秒。若使用一个62个报文段的窗口，每31个报文段确认一次，则数值是1 158 675。

24.8 使用这个选项，确认中回显的时间戳总是来自于引起确认的报文段。对哪个重传的报文段进行确认没有疑问，但仍然需要Karn算法的另一部分，即处理重传的指数退避。

24.9 接收TCP对数据进行排队，但只有完成了三次握手后，数据才能传递给应用程序：当接收TCP进入了ESTABLISH状态。

24.10 交换了5个报文段：

1) 客户到服务器：SYN，数据（请求）和FIN。服务器必须像上个习题所述的一样对数据进行排队。

2) 服务器到客户：SYN和对客户SYN的确认。

3) 客户到服务器：对服务器的 SYN的确认和客户的FIN（再次）。这样使得服务器进入已建立状态，并且来自报文段1的排队数据被传递给服务器应用程序。

4) 服务器到客户：客户 FIN的确认（它也确认了客户的数据）、数据（服务器的应答）

和服务器的FIN。这里假定了SPT足够短以允许这个延迟的确认。当客户 TCP收到这个报文段，就将应答传递给客户端的应用程序，但是整个时间是 RTT加上SPT的两倍。

5) 客户到服务器：对服务器FIN的确认。

24.11 每秒16 128次交互（64 512除以4）。

24.12 使用T/TCP的交互时间不可能比两个主机之间交换一个 UDP数据报所需的时间短。因为T/TCP涉及了UDP没有做的状态处理，所以T/TCP总是要花更多的时间。

第25章

25.1 如果一个系统运行了一个管理进程和一个代理进程，它们很可能是不同的进程。管理进程在UDP端口162监听trap告警，代理进程在 UDP端口161等待请求。如果 trap告警和SNMP请求使用了同样的端口，将很难区分管理进程和代理进程。

25.2 参考25.7节中的“ 表访问（Table Access）”部分。

第26章

26.1 我们预期从服务器来的第 2、4和9报文段将被延迟。第 2和第4报文段之间的时间差是190.7 ms，第2和第9报文段之间的时间差是400.7 ms。

从客户到服务器的所有确认看起来都被延迟：第 6、11、13、15、17和19报文段。从第6报文段开始的最后5个时间差是400.0、600.0、800.0、1000.0和2.600 ms。

26.2 如果连接的一个端点处于TCP的紧急模式，每次收到一个报文段，就会发送一个报文段。这个报文段没有告诉接收者任何新的东西（例如，它不是对新数据的确认），它也不包含数据，它只是重申进入了紧急模式。

26.3 只有512个这样的保留端口（ 512~1023），限制了一个主机只能有 512个远程登录的（Rlogin）客户。在实际生活中，这个限制一般小于 512个，因为在这个范围中的一些端口号被不同的服务器，如远程登录服务器，用作了知名端口。

TCP的限制是一对插口定义的一个连接（ 4元组）必须是唯一的。因为 Rlogin服务器总是使用了同样的知名端口（513），一台主机上多个Rlogin客户只有在它们和不同的服务器主机相连接时才可以使用相同的保留端口。然而， Rlogin客户没有采用重用保留端口的技术。如果使用了这种技术，理论限制就是在同一时刻最多有 512个Rlogin客户和同一个的服务器主机相连。

第27章

27.1 当一对插口处于 2MSL等待另一端状态时，理论上不能建立连接。然而，实际上，在18.6节中我们看到大多数伯克利演变的实现确实为一个处于 TIME_WAIT状态的连接接受了一个新的SYN。

27.2 这些行不是以3个数字作为应答代码开始的服务器应答的一部分，因此它们不可能来自于服务器。

第28章

28.1 一个域文字 (domain literal) 是在一对方括号里的点分十进制 IP地址。例如： `mail`

rstevens@[140.252.1.54]。

28.2 6个来回：HELO命令、MAIL、RCPT、DATA报文主体和QUIT。

28.3 这是合法的，称为流水线技术 (pipelining) [Rose 1993，4.4.4节]。不幸的是，有一些脑子坏了（brain-damaged）的SMTP接收者实现，每处理完一条命令就要清除它们的输入缓存，使得这种技术不可用。如果使用了这种技术，客户自然不能丢弃报文直到所有的应答都已检查过，确信报文已被服务器接受了。

28.4 考虑习题28.2的前5个网络上的往返。每个都是一个小命令（很可能是只有一个报文段），对网络几乎没有负载。如果所有 5个都没有重传地送到了服务器，当报文主体发送时，拥塞窗口可能是6个报文段。如果报文主体很大，客户可能一次发送前 6个报文段，造成网络可能来不及处理。

28.5 更新版本的BIND使用同样的值来正移MX记录，作为平衡负载的一种方式。

第29章

29.1 不，因为tcpdump不能从其他UDP数据报中区别RPC请求或应答。它只有在源端口号或目的端口号为2049时，才将UDP数据报的内容理解为NFS分组。随机的RPC请求和应答可以使用两个端点上的一个临时的端口号。

29.2 回忆一下1.9节中，一个进程必须有超级用户权限才能给自己分配一个小于 1024的端口号（一个知名端口）。尽管这对于系统提供的服务器没问题，如 Telnet服务器、FTP服务器、和端口映射器，但我们不想给所有的 RPC服务器提供这个权限。

29.3 这个例子中的两个概念是客户忽略那些服务器应答，如果这些应答不具有客户正在等待的XID；UDP对收到的数据报进行排队（队列长度有一个上限），直到应用进程读取了数据报。另外，XID不会在超时和重传时改变，它只在调用另一个服务器过程时改变。

客户stub执行的事件为：时刻 0：发送请求1；时刻4：超时并重传请求 1；时刻5：接收服务器应答1，将应答返回给应用程序；时刻 5：发送请求2；时刻9：超时并重传请求2；时刻10：收到服务器的应答 1，但因为我们在等待应答 2，所以忽略它；时刻 11：收到服务器的应答2，将应答返回给应用程序。

服务器的事件如下：时刻0：收到请求1，启动操作；时刻5：发送应答1；时刻5：收到请求1（来自于客户在时刻 4的重传），启动操作；时刻 10：发送应答1；时刻10：收到请求2（来自于客户在时刻 5的重传），启动操作；时刻 11：发送应答2；时刻11：收到请求2（来自于客户在时刻 9的重传），启动操作；时刻 12：发送应答2。这个最后的服务器应答仅仅被客户的 UDP排队，用于客户的下一次接收操作。当客户读了这个应答时，XID将是错误的，客户将忽略它。

29.4 改变服务器的以太网卡就改变了它的物理地址。即使我们注意到在 4.7节中SVR4没有在引导时发送一个免费 ARP，它仍然必须在能够应答 sun的NFS请求之前，向sun发送一个请求sun的物理地址的ARP请求。因为sun已经有了svr4的一个ARP登记项，它从这个ARP请求中根据发送者的（新）硬件地址更新这个登记项。

29.5 客户块I/O守护进程的第2个（在偏移 73728处读的）与第1个失去同步大约 0.74秒。即在行131~145，第2个守护进程在第 1个之后超时0.74秒。看来服务器没有看到在 167行的请求，但它看到了 168行的请求。第2个块 I/O守护进程只会在 168行之后0.74秒才会重传。

同时，第1个块I/O守护进程继续发送请求。

29.6 如果使用的是TCP，包含着服务器应答的TCP报文段在网络中丢失了，当服务器的TCP模块没有从客户的TCP模块收到一个确认时，它将超时，并重传应答。最终，这个报文段将到达客户的TCP。这里不同的是两个TCP模块完成超时和重传，而不是NFS客户和服务器（当使用UDP时，NFS客户代码完成超时和重传）。因此，NFS客户并不知道应答丢失了，需要被重传。

29.7 NFS服务器在重启动之后获得一个不同的端口号是可能的。这将使客户变得很复杂，因为它需要知道服务器崩溃了，并且在服务器重启动之后与服务器主机的端口映射器联系以找到NFS服务器的新的端口号。

这种情况，即服务器主机崩溃然后重启动时，一个服务器的RPC应用程序获得一个新的临时性端口，可能发生在任何一个没有使用知名端口的RPC应用程序上。

29.8 不。NFS客户可以为不同的服务器主机使用相同的本地的保留端口号。TCP要求由本地IP地址、本地端口、远端IP地址和远端端口组成的4元组是唯一的，对于每个服务器主机来说，远端的IP地址是不同的。

第30章

30.1 键入`whois "net 88"`。A类网络号64~95是保留的。

30.2 键入`whois whitehouse.gov`可以使用`host`命令或者`nslookup`查询DNS。

30.3 不，`xscope`可以与服务器运行在不同的主机上。如果主机不同，`xscope`也可以使用TCP端口6000作为它的呼入连接。

附录E 配置选项

我们已经看到了许多冠以“依赖于具体配置”的TCP/IP特征。典型的例子包括是否使能UDP的检验和（11.3节），具有同样的网络号但不同的子网号的目的IP地址是本地的还是非本地的（18.4节）以及是否转发直接的广播（12.3节）。实际上，一个特定的TCP/IP实现的许多操作特征都可以被系统管理员修改。

这个附录列举了本书中用到的一些不同的TCP/IP实现可以配置的选项。就像你可能想到的，每个厂商都提供了与其他实现不同的方案。不过，这个附录给出的是不同的实现可以修改的参数类型。一些与实现联系紧密的选项，如内存缓存池的低水平线，没有描述。

这些描述的变量只用于报告的目的。在不同的实现版本中，它们的名字、默认值、或含义都可以改变。所以你必须检查你的厂商的文档（或向他们要更充分的文档）来了解这些变量实际使用的单词。

这个附录没有覆盖每次系统引导时发生的初始化工作：对每个网络接口使用`ifconfig`进行初始化（设置IP地址、子网掩码等等）、往路由表中输入静态路由等等。这个附录集中描述了影响TCP/IP操作的那些配置选项。

E.1 BSD/386版本1.0

这个系统是自从4.2BSD以来使用的“经典”BSD配置的一个例子。因为源代码是和系统一起发布的，所以管理员可以指明配置选项，内核也可重编译。存在两种类型的选项：在内核配置文件中定义的常量（参见`config`(8)手册）和在不同的C源文件中的变量初始化。大胆而又经验丰富的管理员也可以使用排错工具修改正在运行的内核或者内核的磁盘映像中这些变量的值，以避免重新构造内核。

下面列出的是在内核配置文件中可以修改的常量。

IPFORWARDING

这个常量的值初始化内核变量`ipforwarding`。如果值为0（默认），就不转发IP数据报。如果是1，就总是使能转发功能。

GATEWAY

如果定义了这个常量，就使得IPFORWARDING的值被置为1。另外，定义这个常量还使得特定的系统表格（ARP快速缓存表和路由表）更大。

SUBNETSARELOCAL

这个常量的值初始化内核变量`subnetsarelocal`。如果值为1（默认），一个和发送主机具有同样网络号、但不同子网号的目的IP地址被认为是本地的。如果是0，只有在同一个子

网的目的IP地址才认为是本地的。图E-1总结了上述规律。

网络标识符	子网标识符	subnetsarelocal		注释
		1	0	
相同	相同	本地	本地	总是本地的
相同	不同	本地	非本地	依赖于配置
不同		非本地	非本地	总是非本地的

图E-1 对subnetarelocal内核变量的理解

这个变量的值影响了TCP选择的MSS。当给一个本地的目的地址发送报文时，TCP选择的是基于输出接口的MTU的MSS。而发送给一个非本地的地址时，TCP使用变量 `tcp_mssdflt` 作为MSS。

IPSENDREDIRECTS

这个常量的值初始化内核变量 `ipsendredirects`。如果值为1（默认），主机在转发IP数据报时，将发送ICMP重定向。如果是0，不发送ICMP重定向。

DIRECTED_BROADCAST

如果值为1（默认），如果收到的数据报的目的地址是主机的一个接口的直接广播地址，就将它作为一个链路层的广播来转发。如果是0，这些数据报就会被丢弃。

下面的变量也可以改变，它们在目录 /usr/src/sys/netinet中的不同文件中定义。

`tcprexmtthresh`

引起快速重传和快速恢复算法的连续ACK的数目。默认值是3。

`tcp_ttl`

TCP段的TTL字段的默认值。默认值是60。

`tcp_mssdflt`

用于非本地目的地址的默认的TCP MSS。默认值是512。

tcp_keepidle

在发送一个`keepalive`探测报文之前必须等待的500 ms时钟间隔的次数。默认值是14 400（2个小时）。

`tcp_keepintvl`

如果没有收到响应，在两个连续的`keepalive`探测报文之间等待的500 ms时钟间隔的次数。默认值是150（75秒）。

`tcp_sendspace`

TCP发送缓存的默认大小。默认值是4096。

`tcp_recvspace`

TCP接收缓存的默认大小。这个值影响了提供的窗口大小。默认值是4096。

`udpcksum`

如果非0，对输出的UDP数据报计算UDP检验和，并且对于包含了非0检验和的输入UDP数据报要验证它们的检验和。如果值为0，不计算输出的UDP数据报的检验和，也不验证输入UDP数据报的检验和，即使发送者计算了一个检验和。默认值是1。

udp_ttl
UDP数据报TTL字段的默认值。默认值是30。
udp_sendspace
UDP发送缓存的默认大小。定义了可以发送最大的UDP数据报。默认值是9126。
udp_recvspace
UDP接收缓存的默认大小。默认值是41 600，允许40个1024字节的数据报。

E.2 SunOS 4.1.3

SunOS 4.1.3使用的方法类似于我们在BSD/386中看到的。因为大部分的内核源代码都没有发布，所以所有的C变量初始化都包含在一个提供的C源文件中。

管理员的内核配置文件（参见config(8)手册）可以定义下面的变量。修改了配置文件之后，需要构造一个新的内核，然后重启动。

IPFORWARDING

这个常量的值初始化内核变量ip_forwarding。如果值为-1，就不转发IP数据报，而且变量的值不能再改变。如果是0（默认），不转发IP数据报，但是如果多个接口都工作，变量的值可以修改为1。如果是1，就总是能转发IP数据报。

SUBNETSARELOCAL

这个常量的值初始化内核变量ip_subnetsarelocal。如果值为1（默认），一个和发送主机具有同样网络号，但不同子网号的目的IP地址被认为是本地的。如果是0，只有在同一个子网的目的IP地址才认为是本地的。图E-1总结了上述规律。当给一个本地的目的地址发送报文时，TCP选择的是基于输出接口的MTU的MSS，而发送给一个非本地的地址时，TCP使用变量tcp_default_mss作为MSS。

IPSENDREDIRECTS

这个常量的值初始化内核变量ip_sendredirects。如果值为1（默认），主机在转发IP数据报时，将发送ICMP重定向。如果是0，不发送ICMP重定向。

DIRECTED_BROADCAST

这个常量的值初始化内核变量ip_dirbroadcast。如果值为1（默认），如果收到的数据报的目的地址是主机的一个接口的直接广播地址，就将它作为一个链路层的广播来转发。如果是0，这些数据报就会被丢弃。

文件/usr/kvm/sys/netinet/in_proto.c定义了下面一些可以修改的变量。一旦修改了这些变量，必须构造一个新的内核，然后重启动。

tcp_default_mss
用于非本地地址的默认TCP MSS。默认值是512。
tcp_sendspace
TCP发送缓存的默认大小。默认值是4096。

`tcp_recvspace`

TCP接收缓存的默认大小。这个值影响了提供的窗口大小。默认值是 4096。

`tcp_keeplen`

一个发往4.2BSD主机的keepalive探测报文必须包含一个字节的数据来得到一个响应。把这个变量的值设置为1是为了兼容于以前的实现。默认值是 1。

`tcp_ttl`

TCP段的TTL字段的默认值。默认值是60。

`tcp_nodelack`

如果非0，对ACK不做延迟。默认值是0。

`tcp_keepidle`

在发送一个keepalive探测报文之前必须等待的 500 ms时钟间隔的次数。默认值是 14 400（2个小时）。

`tcp_keepintvl`

如果没有收到响应，在两个连续的 keepalive探测报文之间等待的500 ms时钟间隔的次数。默认值是150（75秒）。

`udp_cksum`

如果非0，对输出的UDP数据报计算UDP检验和，并且对于包含了非 0检验和的输入UDP数据报要验证它们的检验和。如果值为 0，不计算输出UDP数据报的检验和，也不验证输入UDP数据报的检验和，即使发送者计算了一个检验和。默认值是 0。

`udp_ttl`

UDP数据报TTL字段的默认值。默认值是60。

`udp_sendspace`

UDP发送缓存的默认大小。定义了可以发送最大的 UDP数据报。默认值是9000。

`udp_recvspace`

UDP接收缓存的默认大小。默认值是 18 000，允许两个9000字节的数据报。

E.3 SRV4

SVR4的TCP/IP配置类似于前两个系统，但可用的选项更少。在文件`/etc/conf/pack.d/ip/space.c`5可以定义两个常量，然后必须重新构造内核并且重启动。

IPFORWARDING

这个常量的值初始化内核变量`ipforwarding`。如果是0（默认），不转发IP数据报。如果是1，就总是能转发IP数据报。

IPSENDREDIRECTS

这个常量的值初始化内核变量`ipsendredirects`。如果值为1（默认），主机在转发IP数据报时，将发送ICMP重定向。如果是0，不发送ICMP重定向。

前两节中，我们描述的许多变量在内核中都有定义，但必须修补内核来改变它们。例如，存在一个名为`tcp_keepidle`的变量，它的值是14 400。

E.4 Solaris 2.2

Solaris 2.2是较新的Unix系统的典型代表，它为管理员提供了一个可以改变 TCP/IP系统配置选项的程序。这样可以不必通过修改源文件和重新构造内核来进行配置。

配置程序是ndd(1)。我们可以运行程序，看看在UDP模块中可以检验和修改的参数：

```
solaris % ndd /dev/udp \?
udp_wroff_extra                 读、写
udp_def_ttl                     读、写
udp_first_anon_port             读、写
udp_trust_optlen                读、写
udp_do_checksum                 读、写
udp_status                      只读
```

我们可以指明 5个模块：/dev/ip、/dev/icmp、/dev/arp、/dev/udp和/dev/tcp。问号参数（为了防止外壳程序解释问号，我们在它前面加了一个反斜线）告诉ndd程序列出那个模块的所有参数。查询一个变量的值的例子是：

```
solaris %ndd /dev/tcp tcp_mss_def
536
```

为了修改一个变量的值，我们需要有超级用户的权限，输入：

```
solaris #ndd -set /dev/ip ip_forwarding 0
```

这些变量可以划分为三种类型：

1) 系统管理员可以修改的配置变量（如， ip_forwarding）。

2) 只能显示的状态变量（如， ARP快速缓存）。这个信息一般通过命令 ifconfig，netstat和arp以一种更好理解的格式提供。

3) 用于内核源代码的排错变量。使能一些这种变量可以在运行时产生内核的排错输出，当然这会降低系统的性能。

现在我们可以描述每个模块的参数了。所有的参数如果没有注明“（只读）”，就是可读写的。只读的参数是上面第 2种情况的状态变量。我们对于第 3种情况的变量注明了“（排错）”。如果不另外说明，所有的计时变量都以毫秒指明，这和其他系统不同，其他系统一般以 500 ms时钟间隔的次数来指明时间。

/dev/ip

ip_cksum_choice

（排错）在IP检验和算法的两个独立实现之中选择一个。

ip_debug

（排错）如果大于0，使能内核打印排错信息功能。值越大输出的信息越多。默认为 0。

ip_def_ttl

如果运输层没有指明，指定输出IP数据报默认的TTL。默认值是255。

ip_forward_directed_broadcasts

如果值为1（默认），如果收到的数据报的目的地址是主机的一个接口的直接广播地址，就将它作为一个链路层的广播来转发。如果是 0，这些数据报就会被丢弃。

ip_forward_src_routed

如果为1（默认），就转发包含一个源路由选项的接收数据报。如果为 0，这些数据报将被

丢弃。

ip_forwarding

指明系统是否转发进入的IP数据报：0表示不转发，1表示总是转发，2（默认）表示只有当两个或两个以上接口都工作时才转发。

ip_icmp_return_data_bytes

一个ICMP差错返回的除了IP首部以外的数据字节的数目，默认是64。

ip_ignore_delete_time

（排错）一个IP路由表项（IRE）最小的生命期。默认是30秒（这个参数以秒记，不是毫秒）。

ip_ill_status

（只读）显示每个IP下层数据结构的状态。每个接口存在一个下层数据结构。

ip_ipif_status

（只读）显示每个IP接口数据结构的状态（IP地址、子网掩码等等）。每个接口存在一个这种结构。

ip_ire_cleanup_interval

（排错）扫描IP路由表，删除过时表项的时间间隔。默认是30 000 ms（30秒）。

ip_ire_flush_interval

从IP路由表中无条件地刷新ARP信息的间隔。默认是1200 000 ms（20分钟）。

ip_ire_pathmtu_interval

路径MTU发现算法尝试增加MTU的间隔。默认是30 000 ms（30秒）。

ip_ire_redirect_interval

来自ICMP重定向的IP路由表项被删除的间隔。默认是60 000 ms（60秒）。

ip_ire_status

（只读）显示所有的IP路由表项。

ip_local_cksum

如果为0（默认），IP不为通过环回接口发送和接收的数据报计算IP检验和或者更高层的检验和（即TCP、UDP、ICMP或IGMP）。如果为1，就要计算这些检验和。

ip_mrtdebug

（排错）如果为1，使能内核打印多播路由的排错输出。默认是0。

ip_path_mtu_discovery

如果为1（默认），IP执行路径MTU发现。如果是0，IP不会在输出的数据报中设置“不分片”比特。

ip_respond_to_address_mask

如果为0（默认），主机不响应ICMP的地址掩码请求。如果为1，主机则响应。

ip_respond_to_echo_broadcast

如果为1（默认），主机响应发往一个广播地址的ICMP回显请求。如果为0，则不响应。

ip_respond_to_timestamp

如果为0（默认），主机不响应ICMP的时间戳请求。如果为1，则响应。

ip_respond_to_timestamp_broadcast

如果为0（默认），主机不响应发往一个广播地址的ICMP时间戳请求。如果为1，则响应。

ip_rput_pullups

（排错）来自于网络接口驱动程序的缓存数目的计数，它需要增长以访问整个 IP首部。引导时它被初始化为0，并且可以被复位为0。

ip_send_redirects

如果为1（默认），当主机作为一个路由器时，它发送ICMP重定向。如果为0，则不发送。

ip_send_source_quench

如果为1（默认），当输入的数据报被丢弃时，主机生成 ICMP源抑制差错。如果为 0，则不生成这种差错。

ip_wroff_extra

（排错）在缓存中为IP首部分配的额外空间的字节数。默认是 32。

/dev/icmp

icmp_bsd_compat

（排错）如果为1（默认），收到的数据报的IP首部的长度字段的值被调整为不包括 IP首部的长度。这和伯克利演变的实现是一致的，用于读原始的 IP或原始的ICMP分组的应用程序。如果为0，则不改变长度字段的值。

icmp_def_ttl

输出ICMP报文的默认的TTL。默认值为255。

icmp_wroff_extra

（排错）在缓存中为IP选项和数据链路首部所分配的额外空间的字节数。默认是 32。

/dev/arp

arp_cache_report

（只读）ARP的快速缓存。

arp_cleanup_interval

ARP登记项从ARP快速缓存中被删除的时间间隔。默认是 300 000 ms（5分钟）（IP为完成的ARP传输维护着它自己的快速缓存；参见 ip_ire_flush_interval）。

arp_debug

（排错）如果为1，使能打印ARP驱动程序的排错输出。默认是 0。

/dev/udp

udp_def_ttl

输出UDP数据报的默认的TTL。默认值是255。

udp_do_checksum

如果为1（默认），为输出的UDP数据报计算UDP检验和。如果为0，输出的UDP数据报不包含一个检验和（和其他大多数的实现不一样，这个 UDP检验和标志并不影响进入的数据报。如果一个接收到的数据报有一个非 0的检验和，它总是要被验证）。

udp_largest_anon_port

可以为UDP临时端口分配的最大端口号。默认是 65535。

`udp_smallest_anon_port`
可以为UDP临时端口分配的最小端口号。默认是32768。

`udp_smallest_nonpriv_port`
一个进程需要超级用户的权限才能给自己分配一个小于这个值的端口号。默认是 1024。

`udp_status`
（只读）所有本地的UDP端点的状态：本地IP地址和端口，远端IP地址和端口。

`udp_trust_optlen`
（排错）不再使用。

`udp_wroff_extra`
（排错）在缓存中为IP选项和数据链路首部所分配的额外空间的字节数。默认是 32。

/dev/tcp

`tcp_close_wait_interval`
2MSL的值：在TIME_WAIT状态花费的时间。默认是240 000 ms（4分钟）。

`tcp_conn_grace_period`
（排错）当发送一个SYN时，在定时器间隔上附加的时间。默认是 500 ms。

`tcp_conn_req_max`
在一个监听的端口上挂起的连接请求的最大数目。默认是 5。

`tcp_cwnd_max`
拥塞窗口的最大值。默认是32768。

`tcp_debug`
（排错）如果为1，使能打印TCP的排错输出。默认是0。

`tcp_deferred_ack_interval`
在发送一个延迟的ACK之前等待的时间。默认是50 ms。

`tcp_dupack_fast_retransmit`
引起快速重传、快速恢复算法的连续的重复 ACK的数目。默认是3。

`tcp_eager_listeners`
（排错）如果为1（默认），TCP在将一个新的连接返回给一个挂起的被动打开的应用程序之前需要进行三次握手。这是大多数的 TCP实现采用的方式。如果为 0，TCP将呼入连接请求（收到的SYN）传递给应用程序，并不完成三次握手直到该应用程序接受了这个连接（把这个值置为0可能引起很多已经存在的应用程序不能用）。

`tcp_ignore_path_mtu`
（排错）如果为1，路径MTU发现算法忽略接收到的需要 ICMP分段的报文。如果为0（默认），使能TCP的路径MTU发现。

`tcp_ip_abort_cinterval`
当TCP进行一个主动打开时，整个重传超时的值。默认是 240 000 ms（4分钟）。

`tcp_ip_abort_interval`
一个TCP连接建立以后，整个重传超时的值。默认是 120 000 ms（2分钟）。

`tcp_ip_notify_cinterval`

当TCP正在进行一个主动打开时，TCP通知IP去寻找一条新路由超时的值。默认是10 000 ms（10秒）。

`tcp_ip_notify_interval`

TCP为一个已经建立的连接通知IP去寻找一条新路由超时的值。默认是10 000 ms（10秒）。

`tcp_ip_ttl`

用于输出TCP段的TTL。默认为255。

`tcp_keepalive_interval`

在发出一个keepalive探测报文之前，一个连接保持空闲状态的时间。默认为 7200000 ms（2小时）。

`tcp_largest_anon_port`

为TCP临时端口分配的最大端口号。默认为 65535。

`tcp_maxpsz_multiplier`

（排错）指明了报文流首部将应用程序写的数据分装成几个 MSS。默认是1。

`tcp_mss_def`

非本地的目的地址的默认的MSS。默认是536。

`tcp_mss_max`

最大的MSS。默认为65495。

`tcp_mss_min`

最小的MSS。默认为1。

`tcp_naglim_def`

（排错）每个连接的Nagle算法阈值的最大值。默认是 65535。每个连接的值以MSS的最小值或这个值开始。TCP_NODELAY插口选项将每个连接的值设置为 1，以禁止Nagle算法。

`tcp_old_urp_interpretation`

（排错）如果为1（默认），采用紧急指针的一个以前的（但更常见的） BSD的理解：它指向紧急数据最后一个字节后的一个字节。如果为 0，采用主机需求RFC理解：它指向紧急数据的最后一个字节。

`tcp_rcv_push_wait`

（排错）在把接收数据传递给应用程序之前，可以缓存的没有设置 PUSH标志的数据的最大字节数。默认是16384。

`tcp_rexmit_interval_initial`

（排错）初始的重传超时间隔。默认是 500 ms。

`tcp_rexmit_interval_max`

（排错）最大的重传超时间隔。默认是 60 000 ms（60秒）。

`tcp_rexmit_interval_min`

（排错）最小的重传超时间隔。默认是 200 ms。

`tcp_rwin_credit_pct`

（排错）在对每个接收的段进行流量控制检查之前，必须达到的接收缓存窗口的百分比。默认是50%。

`tcp_smallest_anon_port`

分配给TCP临时端口的开始端口号。默认是32768。

`tcp_smallest_nonpriv_port`

一个进程需要有超级用户的权限才能给自己分配一个小于这个值的端口号。默认是1024。

`tcp_snd_lowat_fraction`

（排错）如果非0，发送缓存的低水平线是发送缓存的大小除以这个值。默认是0（禁止）。

`tcp_status`

（只读）所有TCP连接的信息。

`tcp_sth_rcv_hiwat`

（排错）如果非0，把报文流首部的高水平线设置为这个值。默认为0。

`tcp_sth_rcv_lowat`

（排错）如果非0，把报文流首部的低水平线设置为这个值。默认为0。

`tcp_wroff_xtra`

（排错）在缓存中为IP选项和数据链路首部所分配的额外空间的字节数。默认是32。

E.5 AIX 3.2.2

AIX3.2.2允许在运行时使用no命令设置网络选项。它可以显示一个选项的值，设置一个选项的值，或者将一个选项的值设置为默认。例如，显示一个选项，我们键入：

```
aix % no -o udp_ttl
udp_ttl = 30
```

下面的选项可以被修改。

`arpt_killc`

在删除一个不活动的、完成的ARP项之前等待的时间（以分钟计）。默认是20。

`ipforwarding`

如果为1（默认），总是转发IP数据报。如果为0，则禁止转发。

`ipfragttl`

等待重新装配的IP数据报片的生存时间（单位为秒）。默认是60。

`ipsendredirects`

如果为1（默认），当转发IP数据报时，主机将发送ICMP重定向。如果为0，则不发送ICMP重定向。

`loop_check_sum`

如果为1（默认），对通过环回接口发送的数据报计算IP检验和。如果为0，则不计算这个检验和。

`nonlocsrcroute`

如果为1（默认），就转发包含一个源路由选项的接收数据报。如果为0，就丢弃这些数据报。

`subnetsarelocal`

如果值为1（默认），一个和发送主机具有同样网络号，但不同子网号的目的IP地址被认为是本地的。如果是0，只有在同一个子网的目的IP地址才认为是本地的。图E-1总结了上述规律。当给一个本地的目的地址发送报文时，TCP选择的是基于输出接口的MTU的MSS。当发送给一个非本地的地址时，TCP使用默认（536）作为MSS。

`tcp_keepidle`

在发送一个keepalive探测报文之前等待的500 ms时间段的倍数。默认值是14 400（2小时）。

`tcp_keepintvl`

如果没有收到响应，在发送下一个 keepalive探测报文之前等待的 500 ms时间段的倍数。默认值是150（75秒）。

`tcp_recvspace`

TCP接收缓存的默认大小。它影响了提供的窗口大小。默认值是 16 384。

`tcp_sendspace`

TCP发送缓存的默认大小。默认是 16 384。

`tcp_ttl`

TCP报文段TTL字段的默认值。默认值是60。

`udp_recvspace`

UDP接收缓存的默认大小。默认值是 41 600，允许40个1024字节数据报。

`udp_sendspace`

UDP发送缓存的默认大小。定义了可以发送的最大的 UDP数据报。默认是9216。

`udp_ttl`

UDP数据报TTL字段的默认值。默认值是30。

E.6 4.4BSD

4.4BSD是第一个为多个内核参数提供动态配置的伯克利版本。使用 sysctl（8）命令。参数的名字看起来就像SNMP的MIB变量的名字。查看一个参数，我们键入：

```
vangogh %sysctl net.inet.ip.forwarding
net.inet.ip.forwarding = 1
```

要修改一个参数的值需要有超级用户的权限，键入：

```
vangogh #sysctl -w net.inet.ip.ttl=128
```

可以修改下面的参数。

`net.inet.ip.forwarding`

如果为0（默认），就不转发IP数据报。如果为1，则使能转发功能。

`net.inet.ip.redirect`

如果为1（默认），当转发 IP数据报时，主机将发送 ICMP重定向。如果为 0，则不发送ICMP重定向。

`net.inet.ip.ttl`

TCP和UDP默认的TTL。默认值是64。

`net.inet.icmp.maskrepl`

如果为0（默认），主机不响应ICMP地址掩码请求。如果为1，则响应。

`net.inet.udp.checksum`

如果为1（默认），对输出的UDP数据报计算UDP检验和，并且对于包含了非0检验和的呼入UDP数据报要校验它们的检验和。如果值为 0，不计算输出UDP数据报的检验和，也不验证输入UDP数据报的检验和，即使发送者计算了一个检验和。

另外，在本附录前面部分描述的许多变量分散在几个不同的源文件中（`tcp_keepidle`、`subnetarelocal`等等），它们也可以被修改。

附录F　可以免费获得的源代码

本书中使用了很多共享软件包。本附录提供了一些如何获得这些软件的细节。

用来获得共享软件的技术称为匿名 FTP，FTP是标准的Internet文件传输协议（第 27章）。27.3节显示了一个匿名FTP的例子。如果想了解Internet资源的一般背景和匿名FTP的知识，请参考任何一本关于Internet的书，如[LaQuey 1993] 或 [Krol 1992]。

这里列出的主机被认为是提供共享软件的主要站点。其他许多站点也提供这些软件。可以通过Internet的Archie服务查找其他的软件版本。下面列出的软件版本是本书中使用到的版本。

当你读到这本书时，可能已经发布了更新版本的软件。

你应该使用FTP的dir命令来看一下在指定的主机上是否存在更新的版本。

本附录按照资源在本书中出现的章节号进行排序。

RFC（1.11节）

1.11节提供了请求得到RFC站点的电子邮件地址。应答中包含了许多可以使用电子邮件或匿名FTP获得RFC的站点。

记住你要先得到一个当前的索引，在索引中查找想要的 RFC。这个RFC项中还告诉你这个RFC是否废弃了或是被一个更新的RFC替代了。

BSD Net/2 源代码（1.14节）

BSD Net/2 源代码，其中包括了TCP/IP协议的内核实现和标准的应用程序（Telnet客户和服务器、FTP客户和服务器等），从主机`ftp.uu.net`的目录树中以`system/unix/bsd-source`开始的位置可以得到。

SLIP（2.4节）

本书中使用的SLIP的版本来自于`ftp.ee.lbl.gov`。文件名以`cslip`开始，因为它提供了压缩的SLIP（2.5节）。

icmpaddrmask程序（6.3节）

参见本附录的最后一项。

icmptime程序（6.4节）

参见本附录的最后一项。

ping程序（第7章）

BSD版本的`ping`程序一般比其他厂商提供的版本具有更多的选项和特征。主机`ftp.uu.net`

的文件system/unix/bsd-sources/sbin/ping中包含了最新版本的ping程序。

traceroute程序（第8章）

traceroute程序来自于主机ftp.ee.lbl.gov。本附录的最后一项提供了8.5节使用的允许不严格的和严格的源站选路的版本。

路由器发现守护程序（9.6节）

可用的一个程序为路由器发现报文提供了主机支持和路由器支持。主机是gregorio.stanford.edu，文件是gw-discovery/nordmark-rdisc.tar。这个程序是Sun微系统公司开发的一个共享软件。

gated守护程序（10.3节）

在10.3节提到的gated路由选择守护程序在主机gated.cornell.edu上可以得到。

traceroute.pmtu程序（11.7节）

参见本附录的最后一项。

IP多播软件（第13章）

为SunOS 4.x和Ultrix提供IP多播的修改程序在主机gregorio.stanford.edu的目录vmtp-ip中可以得到。这个目录中还包含了为伯克利Unix系统提供IP多播的修改源程序。

BIND名字服务器（第14章）

BIND名字服务器，即named守护程序，在主机ftp.uu.net的文件networking/ip/dns/bind/bind.4.8.3.tar.z中。

一个更新的版本，即4.9版，从主机gatekeeper.dec.com的目录pub/BSD/bind/4.9中可以得到。

host程序（第14章）

host程序在主机nikhefh.nikhef.nl的文件host.tar.z中提供。

dig和doc程序（第14章）

第14章中提到的dig程序和doc程序在主机isi.edu的文件dig.2.0.tar.Z和doc.2.0.tar.Z中提供。

BOOTP服务器（第26章）

常用的Unix BOOTP服务器的不同版本在主机lancaster.andrew.cmu.edu的pub目录中提供。

TCP快速扩充（第24章）

TCP窗口扩缩选项、时间戳选项和PAWS算法的一个共享源代码作为BSD Net/2版的一组

修改程序在主机uxc.cso.uiuc.edu的文件pub/tcplw.shar.z中提供。

ISODE SNMP管理进程和代理进程（第25章）

25.7节中描述的SNMP管理者进程和代理进程是ISODE 8.0版本的一部分。很多站点都可以得到它，如ftp.uu.net上的目录networking/osi/isode中。

MIME软件和实例（28.4节）

一个名为MetaMail的程序为很多不同的用户代理提供了MIME的能力。这个程序可以从主机thumper.bellcore.com的pub/nsb目录中得到。在这个目录中还有MIME的其他信息。

Sun RPC（29.2节）

RPC 4.0版本的源代码（使用插口API）在主机ftp.uu.net的目录systems/sun/sextape/rpc4.0中提供。TI-RPC版本的源代码（使用了TLI API）在主机ftp.uu.net的目录networking/rpc中提供。

Sun NFS（第29章）

一个NFS客户和服务器的共享软件的实现作为BSD Net/2源代码的一部分提供，在本附录的前面部分介绍了。

tcpdump程序（附录A）

本书中使用的tcpdump的版本来自于主机ftp.ee.lbl.gov上的文件tcpdump-2.2.1.tar.z。

BSD分组过滤器（A.1节）

BSD分组过滤器是tcpdump发布的一部分。

sock程序（附录C）

参见本附录的最后一项。

ttcp程序

（这个程序本书中没有用到，但它是一个读者应该注意的常用的工具）。ttcp是测量两个系统之间TCP和UDP性能的基准工具。它是美国军事弹道研究实验室开发的，是一个共享软件。它的副本可以从很多匿名FTP站点获得，但一个增强的版本可以从主机ftp.sgi.com的目录sgi/src/ttcp中获得。

作者编写的软件

本书中用到的作者编写的软件在主机ftp.uu.net的文件published/books/stevens.tcpipiv1.tar.z中提供。

参 考 文 献

所有的RFC都可以如同在1.11节描述的那样在Internet上通过电子邮件或使用匿名 FTP免费获得。

Albitz, P., and Liu, C. 1992. *DNS and BIND*. O'Reilly & Associates, Sebastopol, Calif.
配置和运行一个名字服务器所需要的管理任务的许多细节。

Alexander, S., and Droms, R. 1993. "DHCP Operations and BOOTP Vendor Extensions," RFC1533, 30 pages(Oct.).

Almquist, P. 1992. "Type of Service in the Internet Protocol Suite," RFC1349, 28 pages (July).
怎样使用IP首部的服务类型字段。

Almquist, P., ed. 1993. "Requirements for IP Routers," Internet Draft (Mar.).
这是代替RFC 1009[Braden and Postel 1987]的一个RFC的草稿。新的RFC将很可能以四卷出现。卷1：Internet的结构、术语和一般性内容。卷 2：链路层、互联网层、运输层和应用层。卷3：转发和选路协议。卷4：操作、维护和网络管理。
这个草稿可以通过匿名 FTP从主机jessica.stanford.edu的目录rreq中获得。一旦最终的RFC发布就要忽略这个草稿。

Bellovin, S. M. 1993. Private Communication.

Bhide, A., Elnozahy, E. N., and Morgan, S. P. 1991. "A Highly Available Network File Server," *Proceeding of the 1991 Winter USENIX Conference*, pp. 199-205, Dallas, Tex.
描述了一个对免费ARP的使用（4.7节）。

Borenstein, N., and Freed, N. 1993. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," RFC 1521, 81 pages (Sept.).
这个RFC废止了早期的RFC 1341。这个RFC的附录H列出了与RFC 1341的不同点。

Borman, D. A., ed. 1990. "Telnet Linemode Option," RFC 1184, 23 pages (Oct.).

Borman, D. A. 1991. "IP Bandwidth Limits," Message-ID 91011437.AA17276@berserkly.cray.com>, Usenet, comp.protocols.tcp-ip Newsgroup (Jan.).
说明了我们在24.8节的最后部分列出的有关 TCP性能的三个限制。

Borman, D. A. 1992. "TCP/IP Performance at Cray Research," *Proceeding of the Twenty-third Internet Engineering Task Force*, pp. 492 - 493 (Mar.), San Diego Supercomputer Center, San Diego, Calif.

Borman, D. A., ed. 1993a. "Telnet Environment Option," RFC 1408, 7 pages (Jan.).
从客户向服务器传递环境变量的 Telnet选项。

Borman, D. A. 1993b. " A Practical Perspective on Host Networking," in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 309-367. Addison-Wesley, Reading, Mass.
对于Host Requirements（主机需求）RFC（1122和1123）的一个实际的审视。

Braden, R. T., ed. 1989a. “Requirements for Internet Hosts —— Communication Layers,” RFC1122, 116 pages (Oct.).
主机需求RFC的前半部分。这部分覆盖了链路层， IP，TCP和UDP。
Braden, R. T., ed. 1989b. “Requirements for Internet Hosts —— Application and Support,” RFC1123, 98 pages (Oct.).
主机需求RFC的后半部分。这部分覆盖了Telnet，FTP，TFTP，SMTP和DNS。
Braden, R. T. 1989c. “Perspective on the Host Requirements RFCs,” RFC 1127, 20 pages (Oct.).
对于开发主机需求RFC的IETF工作组的讨论和结果的一个非正式的总结。
Braden, R. T. 1992a. “TIME-WAIT Assassination Hazards in TCP,” RFC1337, 11 pages (May).
显示处于TIME_WAIT状态时，接收一个RST如何导致问题。
Braden, R. T. 1993b. “Extending TCP for Transactions —— Concepts,” RFC 1379, 38 pages (Nov.).
开发T/TCP背后的概念和历史。
Braden, R. T. 1992c. “Extending TCP for Transactions —— Functional Specification,” Internet Draft, 32 pages (Dec.).
T/TCP的功能说明和有关实现的讨论。
Braden, R. T., Borman, D. A., and Partridge, C. 1988. “Computing the Internet Checksum,” RFC 1071, 24 pages (Sept.).
提供计算IP、ICMP、IGMP、UDP和TCP的检验和的技术和算法。
Braden, R. T., and Postel, J. B. 1987. “Requirements for Internet Gateways,” RFC1009, 55 pages (June).
等效于路由器的主机需求RFC。这个RFC已经被代替了；见 [Almquist 1993]。
Caceres, R., Danzig, P. B., Jamin, S., and Mitzel, D. J. 1991. “Characteristics of Wide-Area TCP/IP Conversations,” *Computer Communication Review*, vol. 21, no. 4, pp. 101-112 (Sept.).
Callon, R. 1992. “TCP and UDP with Bigger Addresses (TUBA), A Simple Proposal for Internet Addressing and Routing,” RFC 1347, 9 pages (June).
Case, J. D., Fedor, M. S., Schoffstall, M. L., and Davin, C. 1990. “Simple Network Management (SNMP),” RFC 1157, 36 pages (May).
SNMP的协议规范。
Case, J. D., McCloghrie, K., Rose, M. T., and Waldbusser, S. 1993. “An Introduction to Version 2 of the Internet-Standard Network Management Framework,” RFC 1441, 13 pages (Apr.).
一个SNMPv2的介绍和其他11个定义SNMPv2的RFC的引用。
Case, J. D., and Partridge, C. 1989. “Case Diagrams: A First Step to Diagrammed Management Information Bases,” *Computer Communication Review*, vol. 19, no. 1, pp. 13-16 (Jan.).
定义了用来可视化一个给定模块中SNMP变量之间关系的图形表示。
Casner, S., and Deering, S. E. 1992. “First IETF Internet Audiocast,” *Computer Communication Review*, vol. 22, no. 3, pp. 92-97 (July).

描述了怎样使用Internet的多播技术实时传递一个IETF会议的声音信息。这篇文章的一个PostScript的 副本可以通过匿名FTP从主机venera.isi.edu上的文件pub/ietf-autocast-article.ps中获得。另外，那台主机上的文件 mbone/faq.txt包含了有关Internet多播骨干（MBONE）常问的问题。

Cheriton, D. P. 1988. “VMTP: Versatile Message Transaction Protocol,” RFC 1045, 123 pages (Feb.).

Cheswick, W. R., and Bellovin, S. M. 1994. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, Reading, Mass.

描述了怎样建立和管理一个防火墙网关以及涉及的安全问题。

Clark, D. D. 1982. “Window and Acknowledgement Strategy in TCP,” RFC 813, 22 pages (July).

标识了糊涂窗口综合症以及如何避免它的原始RFC。

Clark, D. D. 1988. “ The Design Philosophy of the DARPA Internet Protocols,” *Computer Communication Review*, vol. 18, no. 4, pp. 106-114 (Aug.).

描述了影响Internet协议的早期的推理技术。

Comer, D. E., and Stevens, D. L. 1993. *Internetworking with TCP/IP: Vol. III: Client-Server Programming and Applications, BSD Socket Version*. Prentice-Hall, Englewood Cliffs, N.J.

Cooper, A. W., and Postel, J. B. 1993. “ The US Doman,” RFC 1480, 47 pages (June).

描述了DNS的.us域。

Crocker, D. H. 1982. “Standard for the Format of ARPA Internet Text Messages,” RFC 822, 47 pages (Aug.).

定义了使用SMTP传输的电子邮件的格式。

Crocker, D. H. 1993. “Evolving the System,” in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 41-76. Addison-Wesley, Reading, Mass.

讲述了开发ARPANET标准的一些历史和Internet技术共同体当前结构的细节，还定义了当前Internet标准形成的过程。

Croft, W., and Gilmore, J. 1985. “Bootstrap Protocol (BOOTP),” RFC 951, 12 pages (Sept.).

Crowcroft, J., Wakeman, I., Wang, Z., and Sirovica, D. 1992. “Is Layering Harmful?,” *IEEE Network*, vol. 6, no. 1, pp. 20-24 (Jan.).

这篇文章中漏掉的7张图刊登在下一期，卷6，第2期（三月）。

Curry, D. A. 1992. *UN LX System Security: A Guide for Users and System Administrators*. Addison-Wesley, Reading, Mass.

一本关于Unix安全性的书。第4章和第5章讲述网络安全。

Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., and Lumley, J. 1993. “After-burner,” *IEEE Network*, vol. 7, no. 4, pp. 36-43 (July).

描述了怎样通过减少数据复制的次数来加快TCP，以及一个支持这种设计的专用接口卡。

Danzig, P. B., Obraczka, K., and Kumar, A. 1992. “An Analysis of Wide-Area Name Server Traffic,” *Computer Communication Review*, vol. 22, no. 4, pp. 281-292 (Oct.).

对根名字服务器之一进行的两个 24小时流量监视的分析。显示来自于一个故障实现的

DNS流量如何消耗了20倍于正常的WAN带宽。这篇文章的一个PostScript副本通过匿名FTP在主机**caldera.usc.edu**的文件**pub/danzig/dns.ps.z**中提供。

Deering, S. E. 1989. “Host Extensions for IP Multicasting,” RFC 1112, 17 pages (Aug.).
IP多播和IGMP的规范。

Deering, S. E., ed. 1991. “ICMP Router Discovery Messages,” RFC 1256, 19 pages (Sept.).

Deering, S. E., and Cheriton, D. P. 1990. “Multicast Routing in Datagram Internetworks and Extended LANs,” *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85-110 (May).
对于常用路由技术的支持多播传输的扩充。

Dixon, T. 1993. “Comparison of Proposals for Next Version of IP,” RFC 1454, 15 pages (May).
SIP, PIP 和TUBA的比较和总结。

Droms, R. 1993. “Dynamic Host Configuration Protocol,” RFC 1541, 39 pages (Oct.).

Droms, R., and Dyksen, W. R. 1990. “Performance Measurements of the X Window System Communication Protocol,” *Software Practice & Experience*, vol. 20, pp. 119-136 (Oct.).
使用不同的X客户时涉及到的TCP通信的测量。

Fedor, M. S. 1988. “GATED: A Multi-routing Protocol Daemon for Unix,” *Proceeding of the 1988 Summer USENIX Conference*, pp. 365-376, San Francisco, Calif.

Finlayson, R. 1984. “Bootstrap Loading using TFTP,” RFC 906, 4 pages (June).

Finlayson, R., Mann, T., Mogul, J. C., and Theimer, M. 1984. “ A Reverse Address Resolution Protocol,” RFC 903, 4 pages (June).

Floyd, S. 1994. Private Communication.

Ford, P. S., Rekhter, Y., and Braun, H-W. 1993. “Improving the Routing and Addressing of IP,” *IEEE Network*, vol. 7, no. 3, pp. 10-15 (May).
对于CIDR（无类型域间路由）的描述。

Fuller, V., Li, T., Yu, J.Y., and Varadhan, K. 1993. “Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy,” RFC 1519, 24 pages (Sept.).
CIDR（无类型域间选路）的规范。

Gerich, E. 1993. “Guidelines for Management of IP Address Space,” RFC 1466, 10 pages (May).
将来怎样分配IP地址的说明（即B类地址将很难获得，作为替代手段，一般将分配一组C类地址）。

Gurwitz, R., and Hinden, R. 1982. “IP —— Local Area Network Addressing Issues,” IEN 212, 11 pages (Sept.)。
对IP广播地址最早的引用之一。

Harrenstein, K., Stahl, M. K., and Feinler, E. J. 1985. “NICNAME/WHOIS,” RFC 954, 4 pages (OCT.).

Hedrick, C. L. 1988a. “Routing Information Protocol,” RFC 1058, 33 pages (June).

Hedrick, C. L. 1988b. “Telnet Terminal Speed Option,” RFC 1079, 3 pages (Dec.).

Hedrick, C. L., and Borman, D. A. 1992. “Telnet Remote Flow Control Option,” RFC 1372, 6 pages (Oct.).

Hornig, C. 1984. “Standard for the Transmission of IP Datagrams over Ethernet Networks,” RFC 894, 3 pages (Apr.).

Huitema, C. 1993. “IAB Recommendation for an Intermediate Strategy to Address the Issue of Scaling,” RFC 1481, 2 pages (July).
实现CIDR的IAB建议。

Jacobson, V. 1988. “Congrestion Avoidance and Control,” *Computer Communication Review*, vol. 18, no. 4, pp. 314-329 (Aug.).
描述TCP的慢启动和拥塞避免算法的一篇经典文章。这篇文章的一个 PostScript副本通过匿名FTP在主机`ftp.ee.lbl.gov`的文件`congavoid.ps.z`中提供。

Jacobson, V. 1990a. “Compressing TCP/IP Headers for Low-Speed Serial Links,” RFC 1144, 43 pages (Feb.).
描述CSLIP, 一个压缩TCP和IP首部的SLIP版本。

Jacobson, V. 199b. “Modified TCP Congestion Avoidance Algorithm,” April 30, 1990, end2end-interest mailing list (Apr.).
描述了快速重传和快速恢复算法。

Jacobson, V. 1990c. “Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno,” *Proceeding of the Eighteenth Internet Engineering Task Force*, p. 365 (Sept.), University of British Columbia, Vancouver, B.C.

Jacobson, V., and Braden, R. T. 1988. “TCP Extensions for Long-Delay Paths,” RFC 1072, 16 pages (Oct.).
描述了TCP的选择确认选项，在后来的RFC 1323中这个选项已经被删去了。

Jacobson, V. Braden, R. T., and Borman, D. A. 1992. “TCP Extensions for High Performance,” RFC 1323, 37 pages (May).
描述了窗口缩放选项、时间戳选项和PAWS算法，以及需要这些改变的理由。

Jacobson, V., Braden, R. T., and Zhang, L. 1990. “TCP Extensions for High-Speed Paths,” RFC 1185, 21 pages (Oct.).
尽管这个RFC已经被RFC 1323废止了，但有关防止TCP旧的重复的报文段的附录仍然值得一读。

Juszczak, C. 1989. “Improving the Performance and Correctness of an NFS Server,” *Proceedings of the 1989 Winter USENIX Conference*, pp. 53-63, San Diego, Calif.
提供了NFS服务器快速缓存的实现细节。

Kantor, B. 1991. “BSD Rlogin,” RFC 1282, 5 pages (Dec.).
Rlogin协议的规范。

Karn, P., and Partridge, C. 1987. “Improving Round-Trip Time Estimates in Reliable Transport Protocols,” *Computer Communication Review*, vol. 17, no. 5, pp. 2-7 (Aug.).
处理已经重传报文段的重传超时的 Karn算法的细节。这篇文章的一个 PostScript副本通过匿名FTP在主机sics.se的文件pub/craig/karn-partridge.ps中提供。

Katz, D. 1990. “Proposed Standard for the Transmission of IP Datagrams Over FDDI Networks,” RFC 1188, 11 pages (Oct.).

说明了在FDDI网络上如何封装IP数据报和ARP请求和应答，包括多播技术。

Kent, C. A., and Mogul, J. C. 1987. “Fragmentation Considered Harmful,” *Computer Communication Review*, vol. 17, no. 5, pp. 390-401 (Aug.).

Kent, S. T. 1991. “ U.S. Department of Defense Security Options for the Internet Protocol,” RFC 1108, 17 pages (Nov.).

Kleinrock, L. 1992. “The Latency / Bandwidth Tradeoff in Gigabit Networks,” *IEEE Communications Magazine*, vol. 30, no. 4, pp. 36-40 (Apr.).

Klensin, J., Freed, N., and Moore, K. 1993. “SMTP Service Extension for Message Size Declaration,” RFC 1427, 8 pages (Feb.).

Klensin, J., Freed, N., Rose, M. T., Stefferud, E. A., and Crocker, D. 1993a. “SMTP Service Extensions,” RFC 1425, 10 pages (Feb.).

Klensin, J., Freed, N., Rose, M. T., Stefferud, E. A., and Crocker, D. 1993b. “SMTP Service Extension for 8bit-MIME Transport,” RFC 1426, 6 pages (Feb.).

Krol, E. 1992. *The Whole Internet*. O’Reilly & Associates, Sebastopol, Calif.

关于Internet介绍的入门书。

LaQuey, T. 1993. *The Internet Companion: A Beginner ’s Guide to Global Networking*. Addison-Wesley, Reading, Mass.

关于Internet介绍的入门书。

Leffler, S. J., and Karels, M. J. 1984. “Trailer Encapsulations,” RFC 893, 3 pages (Apr.).

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Mass.

一本完整介绍4.3BSD Unix系统的书。这本书描述了Tahoe版的4.3BSD。

Lougheed, K., and Rekhter, Y. 1991. “A Border Gateway Protocol 3 (BGP-3),” RFC 1267, 35 pages (Oct.).

Lynch, D. C. 1993. “Historical Perspective,” *in Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 3-14. Addison-Wesley, Reading, Mass.

描述了Internet的前身：ARPANET。

Macklem, R. 1991. “Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol,” *Proceedings of the 1991 Winter USENIX Conference*, pp. 53-64, Dallas, Tex.

描述了一个同时使用TCP和UDP的NFS实现。

Malkin, G. S. 1993a. “RIP Version 2: Carrying Additional Information.” RFC 1388, 7 pages (Jan.).

Malkin, G. S. 1993b. “Traceroute Using an IP Option,” RFC 1393, 7 pages (Jan.).

建议用于traceroute新版本的ICMP修改。

Mallory, T., and Kullberg, A. 1990. “Incremental Updating of the Internet Checksum,” RFC 1141, 2 pages (Jan.).

描述了递增改变Internet检验和的实现技术。

Manber, U. 1990. “Chain Reactions in Networks,” *IEEE Computer*, vol. 23, no. 10, pp. 57-63 (Oct.).

描述了广播风暴和网络崩溃的类型，类似于习题 9.3和9.4所显示的。

McCanne, S., and Jacobson, V. 1993. “The BSD Packet Filter: A New Architecture for User-Level Packet Capture,” *Proceeding of the 1993 Winter USENIX Conference*. Pp. 259-269, San Diego, Calif.

BSD分组过滤器（BPF）的具体描述以及与 Sun的网络接口栓（NIT）的比较。这篇文章的一个PostScript副本通过匿名FTP在主机ftp.ee.lbl.gov的文件papers/bpf-usenix93.ps.Z中提供。

McCloghrie, K., and Rose, M. T. 1991. “Management Information Base for Network Management of TCP/IP-based Internets: MIB-II,” RFC 1213 (Mar.).

McGregor, G. 1992. “PPP Internet Protocol Control Protocol (IPCP),” RFC 1332, 12 pages (May).

用于TCP/IP的PPP的NCP的描述。

Mills, D. L. 1992. “Network Time Protocol (Version 3): Specification, Implementation, and Analysis,” RFC 1305, 113 pages (Mar.).

Mockapetris, P. V. 1987a. “Domain Names: Concepts and Facilities,” RFC 1034, 55 pages (Nov.).

一个DNS的介绍。

Mockapetris, P. V. 1987b. “Domain Names: Implementation and Specification,” RFC 1035, 55 pages (Nov.).

DNS规范。

Mogul, J. C. 1990. “Efficient Use of Workstations for Passive Monitoring of Local Area Networks,” *Computer Communication Review*, vol. 20, no. 4, pp. 253-263 (Sept.).

描述了如何使用工作站监视局域网，而不购买专门的网络分析硬件产品。

Mogul, J. C. 1992. “Holy Turbocharger Batman, (evil cheating), NFS async writes,” Message-ID 1992Mar2.191711.9935@PA.dec.com, Usenet, comp.protocols.nfs Newsgroup (Mar.).

在一个繁忙的NFS服务器上收集的，40天内Internet检验和错误的一些有趣的统计数据。

Mogul, J. C. 1993. “IP Network Performance,” in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 575-675. Addison-Wesley, Reading, Mass.

包含了许多TCP/IP协议栈的主题，它们可以用来获得最优的性能。

Mogul, J. C., and Deering, S. E. 1990. “Path MTU Discovery,” RFC 1191, 19 pages (Apr.).

Mogul, J. C., and Postel, J. B. 1985. “Internet Standard Subnetting Procedure,” RFC 950, 18 pages (Aug.).

Moore, K. 1993. “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text,” RFC 1522, 10 pages (Sept.).

描述了使用7 bit ASCII在RFC 822邮件首部发送非ASCII字符的一种方法。

Moy, J. 1991. “OSPF Version 2,” RFC 1247, 189 pages (July).

Nagle, J. 1984. “Congestion Control in IP/TCP Internetworks,” RFC 896, 9 pages(Jan.).

Nagle算法的描述。

Nye, A., ed. 1992. *The X Window System, Volume 0: X Protocol Reference Manual, Third*

Edition. O'Reilly & Associates, Sebastopol, Calif.

Obracezka, K., Danzig, P. B., and Li, S. 1993. "Internet Resource Discovery Services," *IEEE Computer*, vol. 26, no. 9, pp. 8-22 (Sept.).

对当前Internet上资源发现工具的一个综述：Alex、Archie、Gopher、Indie、Knowbot信息服务、Netfind、Prospero、WAIS、WWW和X.500。这篇文章的一个PostScript副本通过匿名FTP在主机`caldera.usc.edu`的文件`/pub/kobraczk/ieeecomputer.ps.z`中提供。

Papadopoulos, C., and Parulkar, G. M. 1993. "Experimental Evaluation of SunOS IPC and TCP/IP Protocol Implementation," *IEEE/ACM Transactions on Networking*, vol. 1, no. 2, pp. 199-216 (Apr.).

测量当发送数据和接收数据时在协议族的不同层中引入的开销。

Partridge, C. 1986. "Mail Routing and the Domain System," RFC 974, 7 pages (Jan.).

怎样使用DNS的MX记录进行邮件的选路。

Partridge, C. 1994. *Gigabit Networking*. Addison-Wesley, Reading, Mass.

描述当网络速率超过1 Gb/s时产生的问题。

Partridge, C., and Pink, S. 1993. "A Faster UDP." *IEEE/ACM Transactions on Networking*, Vol. 1, no. 4, pp. 429-440 (Aug.).

描述了对伯克利实现的改进，可以将UDP的性能提高30%。

Paxson, V. 1993. "Empirically-Derived Analytic Models of Wide-Area TCP Connections: Extended Report," LBL-34086, Lawrence Berkeley Laboratory and EECS Division, University of California, Berkeley (June).

包含了在14个广域流量跟踪中250万个TCP连接的分析。这个报告的一个PostScript副本通过匿名FTP在主机`ftp.ee.lbl.gov`的文件`WAN-TCP-models.1.ps.Z`和`WAN-TCP-models.2.ps.Z`中提供。

Perlman, R. 1992. *Interconnections: Bridges and Routers*. Addison-Wesley, Reading, Mass.

一本包含了许多详细的网络互连方法（桥和路由器）和不同的路由算法的书。

Plummer, D. C. 1982. "An Ethernet Address Resolution Protocol," RFC 826, 10 pages (Nov.).

Postel, J. B. 1980. "User Datagram Protocol," RFC 768, 3 pages (Aug.).

Postel, J. B., ed. 1981a. "Internet Protocol," RFC 791, 45 pages (Sept.).

Postel, J. B. 1981b. "Internet Control Message Protocol," RFC 792, 21 pages (Sept.).

Postel, J. B., ed. 1981c. "Transmission Control Protocol," RFC 793, 85 pages (Sept.).

Postel, J. B. 1982. "Simple Mail Transfer Protocol," RFC 821, 68 pages (Aug.).

Postel, J. B. 1987. "TCP and IP Bake Off," RFC 1025, 6 pages (Sept.).

描述了在早期开发过程中，为了测试互连性在TCP/IP的不同实现之间进行的一些测试过程和记录。

Postel, J. B., ed. 1994. "Internet Official Protocol Standards," RFC 1600, 36 pages (Mar.).

所有的Internet协议的状态。这个RFC定期改变——查看最近的RFC索引以了解当前的版本。

Postel, J. B., and Reynolds, J. K. 1983a. "Telnet Protocol Specification," RFC 854, 15 pages (May).

基本的Telnet协议规范。很多以后的RFC描述了特定的Telnet选项。

Postel, J. B., and Reynolds, J. K. 1983b. “Telnet Binary Transmission,” RFC 856, 4 pages (May).

Postel, J. B., and Reynolds, J. K. 1983c. “Telnet Echo Option,” RFC 857, 5 pages (May).

Postel, J. B., and Reynolds, J. K. 1983d. “Telnet Suppress Go Ahead Option,” RFC 858, 3 pages (May).

Postel, J. B., and Reynolds, J. K. 1983e. “Telnet Status Option,” RFC 859, 3 pages (May).

Postel, J. B., and Reynolds, J. K. 1983f. “Telnet Timing Mark Option,” RFC 860, 4 pages (May).

Postel, J. B., and Reynolds, J. K. 1985. “File Transfer Protocol (FTP),” RFC 959, 69 pages (Oct.).

Postel, J. B., and Reynolds, J. K. 1988. “Standard for the Transmission of IP Datagrams over IEEE 802 Networks,” RFC 1042, 15 pages (Apr.).

在IEEE 802网络上封装IP数据报和ARP请求和应答的规范。

Pusateri, T. 1993. “IP Multicast Over Token-Ring Local Area Networks,” RFC 1469, 4 pages (June).

Rago, S. A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, Mass.

一本关于TLI和流子系统的书。

Rekhter, Y. and Gross, P. 1991. “Application of the Border Gateway Protocol in the Internet,” RFC 1268, 13 pages (Oct.).

Rekhter, Y., and Li, T. 1993. “An Architecture for IP Address Allocation with CIDR,” RFC 1518, 27 pages (Sept.).

Reynolds, J. K. 1989. “The Helminthiasis of the Internet,” RFC 1135, 33 pages (Dec.).

包含了1988年Internet蠕虫的详细讨论。

Reynolds, J. K., and Postel, J. B. 1992. “Assigned Numbers,” RFC 1340, 138 pages (July).

Internet协议族所有的编码。这个RFC定期改变——查看最近的RFC索引以了解当前的版本。

Romkey, J. L. 1988. “A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP,” RFC 1055, 6 pages (June).

Rose, M. T. 1990. *The Open Book: A Practical Perspective on OSI*. Prentice-Hall, Englewood Cliffs, N.J.

一本关于OSI协议的书。第8章提供了ASN.1和BER的细节。

Rose, M. T. 1993. *The Internet Message: Closing the Book with Electronic Mail*. Prentice-Hall, Englewood Cliffs, N.J.

一本关于Internet邮件的书，包含MIME的细节。

Rose, M. T. 1994. *The Simple Book: An Introduction to Internet Management, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.

一本关于SNMPv2的书。这本书的第1版介绍了SNMPv1。

Rose, M. T., and McCloghrie, K. 1990. “Structure and Identification of Management

Information for TCP/IP-based Internets,” RFC 1155, 22 pages (May).
定义了SNMPv1的SMI。

Rosenberg, W., Kenney, D., and Fisher, G. 1992. *Understanding DCE*. O’Reilly & Associates, Sebastopol, Calif.
提供了OSF的分布式计算环境的概述。

Routhier, S. A. 1993.“Implementation Experience for SNMPv2,” *The Simple Times*, vol. 2, no. 4, pp. 1-4 (July-Aug.).
描述了对SNMPv1的修改以支持SNMPv2。
这个期刊是免费电子发行的。以主题“ help ”向st-subscriptions@simple-times.org发送一个电子邮件以获得订阅信息。

Schryver, V. J. 1993.“Info on High Speed Transport Protocols Requested,” Message-ID i0imr8g@rhyolite.wpd.sgi.com, Usenet, comp.protocols.tcp-ip Newsgroup (May).
提供了一些FDDI实现上的TCP性能数字。

Schwartz, M. F., and Tsirigotis, P. G. 1991.“Experience with a Semantically Cognizant Internet White Pages Directory Tool,” *Journal of Internetworking Research and Experience*, vol. 2, no. 1, pp. 23-50 (Mar.).
也可以通过匿名 FTP在主机 ftp.cs.colorado.edu的文件 pub/cs/techreports/schwartz/PostScript/ White.Pages.ps.Z中得到。

Simpson, W. A. 1993.“The Point-to-Point Protocol (PPP),” RFC 1548, 53 pages (Dec.).
定义了PPP和它的链路控制协议。

Sollins, K. R. 1992.“The TFTP Protocol (Revision 2),” RFC 1350, 11 pages (July).

Stallings, W. 1987. *Handbook of Computer-Communications Standards, Volume* 2: *Local Network Standards*. Macmillan, New York.
包含了IEEE 802 局域网标准的细节。

Stallings, W. 1993. *SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards*. Addison-Wesley, Reading, Mass.
描述了SNMPv1和SNMPv2的差别。

Stern, H. 1991. *Managing NFS and NIS*. O’Reilly & Associates, Sebastopol, Calif.
包含了许多安装、使用和管理NFS的细节。

Stevens, W. R. 1990. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, N.J.
一本详细介绍在Unix上使用插口和TLI进行网络程序设计的书。

Stevens, W. R. 1992. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, Mass.
一本详细介绍Unix程序设计的书。

Sun Microsystems. 1987.“XDR: External Data Representation Standard,” RFC 1014, 20 pages (June).

Sun Microsystems. 1988a.“RFC: Remote Procedure Call, Protocol Specification, Version 2,” RFC 1057, 25 pages (June).

Sun Microsystems. 1988b.“NFS: Network File System Protocol Specification,” RFC 1094, 27 pages

(Mar.).
第2版Sun NFS的规范。

Sun Microsystems. 1994. *NFS: Network File System Version 3 Protocol Specification*. Sun Microsystems, Mountain View, Calif.
这个文档的一个 PostScript副本通过匿名 FTP在主机 ftp.uu.net的文件 networking/ip/nfs/NFS3.spec.ps.Z中提供。

Tanenbaum, A. S. 1989. *Computer Networks, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.
一本关于计算机网络的通用书。

Topolcic, C. 1993. “Status of CIDR Deployment in the Internet,” RFC 1467, 9 pages (Aug.).

Tsuchiya, P. F. 1991. “On the Assignment of Subnet Numbers,” RFC 1219, 13 pages (Apr.).
建议从最高位往下分配子网 ID，从最低位往上分配主机 ID。这样使得以后在某个点上改变子网掩码时不需要对所有的系统重新编号。

Ullmann, R. 1993. “TP/IX: The Next Internet,” RFC 1475, 35 pages (June).
下一代Internet协议的另一个建议。

VanBokkelen, J. 1989. “Telnet Terminal-Type Option,” RFC 1091, 7 pages (Feb.).

Waitzman, D. 1988. “Telnet Window Size Option,” RFC 1073, 4 pages (Oct.).

Waitzman, D., Partridge, C., and Deering, S. E. 1988. “Distance Vector Multicast Routing Protocol,” RFC 1075, 24 pages (Nov.).

Warnock, R. P. 1991. “Need Help Selecting Ethernet Cards for Very High Performance Throughput Rates,” Message-ID< lbhal10@sgi.sgi.com> Usenet, comp.protocols.tcp-ip Newsgroup (Sept.).
提供了我们从图24-9计算的TCP的性能数据。

Weider, C., Reynolds, J. K., and Heker, S. 1992. “Technical Overview of Directory Services Using the X.500 Protocol,” RFC 1309, 16 pages (Mar.).

Wimer, W. 1993. “Clarifications and Extensions for the Bootstrap Protocol,” RFC 1542, 23 pages (Oct.).

X/Open. 1991. *Protocols for X/Open Internetworking: XNFS*. X/Open, Reading, Berkshire, U.K.
对Sun RPC、XDR和NFS的一个更好的描述。还包含了 NFS锁定管理程序和状态监视协议的描述。附录中详细讨论了使用 NFS和使用本地文件访问之间的语义差别。 X/Open文档编号为XO/CAE/91/030。

Zimmerman, D. P. 1991. “Finger User Information Protocol,” RFC 1288, 12 pages (Dec.)

缩 略 语

ACK (ACKnowledgment) TCP首部中的确认标志
API (Application Programming Interface) 应用编程接口
ARP (Address Resolution Protocol) 地址解析协议
ARPANET
(Defense Advanced Research Project Agency NETwork) (美国)国防部远景研究规划局
AS (Autonomous System) 自治系统
ASCII (American Standard Code for Information Interchange) 美国信息交换标准码
ASN.1 (Abstract Syntax Notation One) 抽象语法记法1
BER (Basic Encoding Rule) 基本编码规则
BGP (Border Gateway Protocol) 边界网关协议
BIND (Berkeley Internet Name Domain) 伯克利Internet域名
BOOTP (BOOTstrap Protocol) 引导程序协议
BPF (BSD Packet Filter) BSD 分组过滤器
CIDR (Classless InterDomain Routing) 无类型域间选路
CIX (Commercial Internet Exchange) 商业互联网交换
CLNP (ConnectionLess Network Protocol) 无连接网络协议
CRC (Cyclic Redundancy Check) 循环冗余检验
CSLIP (Compressed SLIP) 压缩的SLIP
CSMA (Carrier Sense Multiple Access) 载波侦听多路存取
DCE (Data Circuit-terminating Equipment) 数据电路端接设备
DDN (Defense Data Network) 国防数据网
DF (Don't Fragment) IP首部中的不分片标志
DHCP (Dynamic Host Configuration Protocol) 动态主机配置协议
DLPI (Data Link Provider Interface) 数据链路提供者接口
DNS (Domain Name System) 域名系统
DSAP (Destination Service Access Point) 目的服务访问点
DSLAM (DSL Access Multiplexer) 数字用户线接入复用器
DSSS (Direct Sequence Spread Spectrum) 直接序列扩频
DTS (Distributed Time Service) 分布式时间服务
DVMRP (Distance Vector Multicast Routing Protocol) 距离向量多播选路协议
EBONE (European IP BackbONE) 欧洲IP主干网
EOL (End of Option List) 选项清单结束
EGP (External Gateway Protocol) 外部网关协议
EIA (Electronic Industries Association) 美国电子工业协会

FCS (Frame Check Sequence) 帧检验序列
FDDI (Fiber Distributed Data Interface) 光纤分布式数据接口
FIFO (First In, First Out) 先进先出
FIN (FINish) TCP首部中的结束标志
FQDN (Full Qualified Domain Name) 完全合格的域名
FTP (File Transfer Protocol) 文件传送协议
HDLC (High-level Data Link Control) 高级数据链路控制
HELLO 选路协议
IAB (Internet Architecture Board) Internet体系结构委员会
IANA (Internet Assigned Numbers Authority) Internet号分配机构
ICMP (Internet Control Message Protocol) Internet控制报文协议
IDRP (InterDomain Routing Protocol) 域间选路协议
IEEE (Institute of Electrical and Electronics Engineering) （美国）电气与电子工程师协会
IEN (Internet Experiment Notes) 互联网试验注释
IESG (Internet Engineering Steering Group) Internet工程指导小组
IETF (Internet Engineering Task Force) Internet工程专门小组
IGMP (Internet Group Management Protocol) Internet组管理协议
IGP (Interior Gateway Protocol) 内部网关协议
IMAP (Internet Message Access Protocol) Internet报文存取协议
IP (Internet Protocol) 网际协议
IRTF (Internet Research Task Force) Internet研究专门小组
IS-IS (Intermediate System to Intermediate System Protocol) 中间系统到中间系统协议
ISN (Initial Sequence Number) 初始序号
ISO (International Organization for Standardization) 国际标准化组织
ISOC (Internet SOCiety) Internet协会
LAN (Local Area Network) 局域网
LBX (Low Bandwidth X) 低带宽X
LCP (Link Control Protocol) 链路控制协议
LFN (Long Fat Net) 长肥网络
LIFO (Last In, First Out) 后进先出
LLC (Logical Link Control) 逻辑链路控制
LSRR (Loose Source and Record Route) 宽松的源站及记录路由
MBONE (Multicast Backbone On the InterNEt) Internet上的多播主干网
MIB (Management Information Base) 管理信息库
MILNET (MILitary NETwork) 军用网
MIME (Multipurpose Internet Mail Extensions) 通用Internet邮件扩充
MSL (Maximum Segment Lifetime) 报文段最大生存时间
MSS (Maximum Segment Size) 最大报文段长度
MTA (Message Transfer Agent) 报文传送代理

MTU (Maximum Transmission Unit) 最大传输单元
NCP (Network Control Protocol) 网络控制协议
NFS (Network File System) 网络文件系统
NIC (Network Information Center) 网络信息中心
NIT (Network Interface Tap) 网络接口栓（Sun公司的一个程序）
NNTP (Network News Transfer Protocol) 网络新闻传送协议
NOAO (National Optical Astronomy Observatories) 国家光学天文台
NOP (No Operation) 无操作
NSFNET (National Science Foundation NETwork) 国家科学基金网络
NSI (NASA Science Internet) （美国）国家宇航局Internet
NTP (Network Time Protocol) 网络时间协议
NVT (Network Virtual Terminal) 网络虚拟终端
OSF (Open Software Foudation) 开放软件基金
OSI (Open Systems Interconnection) 开放系统互连
OSPF (Open Shortest Path First) 开放最短通路优先
PAWS (Protection Against Wrapped Sequence number) 防止回绕的序号
PDU (Protocol Data Unit) 协议数据单元
POSIX (Portable Operating System Interface) 可移植操作系统接口
PPP (Point-to-Point Protocol) 点对点协议
PSH (PuSH) TCP首部中的急迫标志
RARP (Reverse Address Resolution Protocol) 逆地址解析协议
RFC (Request For Comments) Internet的文档，其中的少部分成为标准文档
RIP (Routing Information Protocol) 路由信息协议
RPC (Remote Procedure Call) 远程过程调用
RR (Resource Record) 资源记录
RST (ReSeT) TCP首部中的复位标志
RTO (Retransmission Time Out) 重传超时
RTT (Round-Trip Time) 往返时间
SACK (Selective ACKnowledgment) 有选择的确认
SLIP (Serial Line Internet Protocol) 串行线路Internet协议
SMI (Structure of Management Information) 管理信息结构
SMTP (Simple Mail Transfer Protocol) 简单邮件传送协议
SNMP (Simple Network Management Protocol) 简单网络管理协议
SSAP (Source Service Access Point) 源服务访问点
SSRR (Strict Source and Record Route) 严格的源站及记录路由
SWS (Silly Window Syndrome) 糊涂窗口综合症
SYN (SYNchronous) TCP首部中的同步序号标志
TCP (Transmission Control Protocol) 传输控制协议
TFTP (Trivial File Transfer Protocol) 简单文件传送协议

TLI (Transport Layer Interface) 运输层接口
TTL (Time-To-Live) 生存时间或寿命
TUBA (TCP and UDP with Bigger Addresses) 具有更长地址的TCP和UDP
Telnet 远程终端协议
UA (User Agent) 用户代理
UDP (User Datagram Protocol) 用户数据报协议
URG (URGent) TCP首部中的紧急指针标志
UTC (Coordinated Universal Time) 协调的统一时间
UUCP (Unix-to-Unix CoPy) Unix到Unix的复制
WAN (Wide Area Network) 广域网
WWW (World Wide Web) 万维网
XDR (eXternal Data Representation) 外部数据表示
XID (transaction ID) 事务标识符
XTI (X/Open Transport Layer Interface) X/Open运输层接口

第1章 概 述

1.1 引言

本章介绍伯克利(Berkeley)联网程序代码。开始我们先看一段源代码并介绍一些通篇要用的印刷约定。对各种不同代码版本的简单历史回顾让我们可以看到本书中的源代码处于什么位置。接下来介绍了两种主要的编程接口，它们在Unix与非Unix系统中用于编写TCP/IP协议。

然后我们介绍一个简单的用户程序，它发送一个UDP数据报给一个位于另一主机上的日期/时间服务器，服务器返回一个UDP数据报，其中包含服务器上日期和时间的ASCII码字符串。这个进程发送的数据报经过所有的协议栈到达设备驱动器，来自服务器的应答从下向上经过所有协议栈到达这个进程。通过这个例子的这些细节介绍了很多核心数据结构和概念，这些数据结构和概念在后面的章节中还要详细说明。

本章的最后介绍了在本书中各源代码的组织，并显示了联网代码在整个组织中的位置。

1.2 源代码表示

不考虑主题，列举15 000行源代码本身就是一件难事。下面是所有源代码都使用的文本格式：

tcp_subr.c

```
381 void
382 tcp_quench(inp, errno)
383 struct inpcb *inp;
384 int     errno;
385 {
386     struct tcpcb *tp = intotcpcb(inp);
387     if (tp)
388         tp->snd_cwnd = tp->t_maxseg;
389 }
```

tcp_subr.c

1.2.1 将初始窗口设置为1

387-388 这是文件tcp_subr.c中的函数tcp_quench。这些源文件名引用4.4BSD-Lite发布的文件。4.4BSD在1.13节中讨论。每个非空白行都有编号。正文所描述的代码的起始和结束位置的行号记于行开始处，如本段所示。有时在段前有一个简短的描述性题头，对所描述的代码提供一个概述。

这些源代码同4.4BSD-Lite发行版一样，偶尔也包含一些错误，在遇到时我们会提出来并加以讨论，偶尔还包括一些原作者的编者评论。这些代码已通过了GNU缩进程序的运行，使它们从版面上看起来具有一致性。制表符的位置被设置成4个栏的界线使得这些行在一个页面中显示得很合适。在定义常量时，有些#ifdef语句和它们的对应语句#endif被删去(如：GATEWAY和MROUTING，因为我们假设系统被作为一个路由器或多播路由器)。所有register说

明符被删去。有些地方加了一些注释，并且一些注释中的印刷错误被修改了，但代码的其他部分被保留下来。

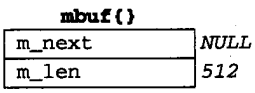
这些函数大小不一，从几行 (如前面的tcp_quench)到最大1100行(tcp_input)。超过大约40行的函数一般被分成段，一段一段地显示。虽然尽量使代码和相应的描述文字放在同一页或对开的两页上，但为了节约版面，不可能完全做到。

本书中有很多对其他函数的交叉引用。为了避免给每个引用都添加一个图号和页码，书封底内页中有一个本书中描述的所有函数和宏的字母交叉引用表和描述的起始页码。因为本书的源代码来自公开的 4.4BSD_Lite版，因此很容易获得它的一个拷贝：附录 B详细说明了各种方法。当你阅读文章时，有时它会帮助你搜索一个在线拷贝 [例如Unix程序grep (1)]。

描述一个源代码模块的各章通常以所讨论的源文件的列表开始，接着是全局变量、代码维护的相关统计以及一个实际系统的一些例子统计，最后是与所描述协议相关的 SNMP变量。全局变量的定义通常跨越各种源文件和头文件，因此我们将它们集中到的一个表中以便于参考。这样显示所有的统计，简化了后面当统计更新时对代码的讨论。卷 1的第25章提供了SNMP的所有细节。我们在本文中关心的是由内核中的 TCP/IP例程维护的、支持在系统上运行的SNMP代理的信息。

1.2.2 印刷约定

通篇的图中，我们使用一个等宽字体表示变量名和结构成员名 (m_next)，用斜体等宽字体表示定义常量 *(NULL)*或常量的值 *(512)*的名称，用带花括号的粗体等宽字体表示结构名称 (mbuf{})。这里有一个例子：



在表中，我们使用等宽字体表示变量名称和结构成员名称，用斜体等宽字体表示定义的常量。这里有一个例子：

m_flags	说 明
M_BCAST	以链路层广播发送/接收

通常用这种方式显示所有的 #define符号。如果必要，我们显示符号的值 (M_BCAST的值无关紧要)并且所列符号按字母排序，除非对顺序有特殊要求。

通篇我们会使用像这样的缩进的附加说明来描述历史的观点或实现的细节。

我们用有一个数字在圆括号里的命令名称来表示 Unix命令，如grep(1)。圆括号中的数字是4.4BSD手册“ manual page ”中此命令的节号，在那里可以找到其他的信息。

1.3 历史

本书讨论在伯克利的加利福尼亚大学计算机系统研究组的 TCP/IP实现的常用引用。历史上，它曾以4.x BSD系统(伯克利软件发行)和“ BSD联网版本 ”发行。这个源代码是很多其他实现的起点，不论是Unix或非Unix操作系统。

图1-1显示了各种BSD版本的年表，包括重要的 TCP/IP特征。显示在左边的版本是公开可

用源代码版，它包括所有联网代码：协议本身、联网接口的内核例程及很多应用和实用程序(如Telnet和FTP)。

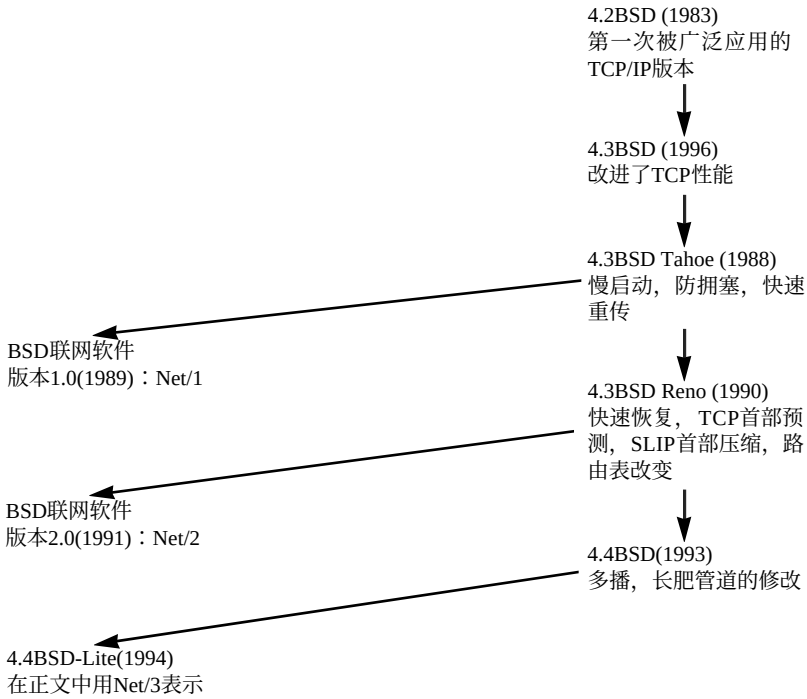


图1-1 带有重要TCP/IP特征的各种BSD版本

虽然本文描述的软件的官方名称为4.4BSD-Lite发行软件，但我们简单地称它为Net/3。

虽然源代码由U. C. Berkeley发行并被称为伯克利软件发行，但TCP/IP代码确实是各种研究者的工作的融合，包括伯克利和其他地区的研究人员。

通篇我们会使用术语源于伯克利的实现来谈及各厂商的实现，如SunOS 4.x、系统V版本4(SVR4)和AIX 3.2，它们的TCP/IP代码最初都是从伯克利源代码发展而来的。这些实现有很多共同之处，通常包括同样的错误！

在图1-1中没有显示的伯克利联网代码的第1版实际上是1982年的4.1cBSD，但是广泛发布的是1983年的版本4.2BSD。

在4.1cBSD之前的BSD版本使用的一个TCP/IP实现，是由Bolt Beranek and Newman(BBN)的Rob Gurwitz和Jack Haverty开发的。[Salus 1994]的第18章提供了另外一些合并到4.2BSD中的BBN代码细节。其他对伯克利TCP/IP代码有影响的实现是由Ballistics研究室的Mike Muuss为PDP-11开发的TCP/IP实现。

描述联网代码从一个版本到下一个版本的变化的文档有限。[Karels and McKusick 1986]描述了从4.2BSD到4.3BSD的变化，并且[Jacobson 1990d]描述了从4.3BSD Tahoe到4.3BSD Reno的变化。

1.4 应用编程接口

在互联网协议中两种常用的应用编程接口(API)是插口(socket)和TLI(运输层接口)。前者

有时称为伯克利插口(Berkeley socket)，因为它被广泛地发布于4.2BSD系统中(见图1-1)。但它已被移植到很多非BSD Unix系统和很多非Unix系统中。后者最初是由AT&T开发的，由于被X/Open承认，有时叫作XTI(X/Open传输接口)。X/Open是一个计算机厂商的国际组织，它制定自己的标准。XTI是TLI的一个有效超集。

虽然本文不是一本程序设计书，但既然在Net/3(和所有BSD版本)中应用编程用插口来访问TCP/IP，我们还是说明一下插口。在各种非Unix系统中也实现了插口。插口和TLI的编程细节在[Stevens 1990]中可以找到。

系统V版本4(SVR4)也为应用编程提供了一组插口API，在实现上与本文中列举的有所不同。在SVR4中的插口基于“流”子系统，在[Rago 1993]中有所说明。

1.5 程序示例

在本章我们用一个简单的C程序(图1-2)来介绍一些BSD网络实现的很多特点。

```
 1 /*
 2  * Send a UDP datagram to the daytime server on some other host,
 3  * read the reply, and print the time and date on the server.
 4  */

 5 #include    <sys/types.h>
 6 #include    <sys/socket.h>
 7 #include    <netinet/in.h>
 8 #include    <arpa/inet.h>
 9 #include    <stdio.h>
10 #include    <stdlib.h>
11 #include    <string.h>

12 #define BUFFSIZE    150         /* arbitrary size */
13 int
14 main()
15 {
16     struct sockaddr_in serv;
17     char    buff[BUFFSIZE];
18     int     sockfd, n;

19     if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
20         err_sys("socket error");

21     bzero((char *) &serv, sizeof(serv));
22     serv.sin_family = AF_INET;
23     serv.sin_addr.s_addr = inet_addr("140.252.1.32");
24     serv.sin_port = htons(13);

25     if (sendto(sockfd, buff, BUFFSIZE, 0,
26                (struct sockaddr *) &serv, sizeof(serv)) != BUFFSIZE)
27         err_sys("sendto error");

28     if ((n = recvfrom(sockfd, buff, BUFFSIZE, 0,
29                       (struct sockaddr *) NULL, (int *) NULL)) < 2)
30         err_sys("recvfrom error");
31     buff[n - 2] = 0;            /* null terminate */
32     printf("%s\n", buff);

33     exit(0);
34 }
```

图1-2 程序示例：发送一个数据报给UDP日期/时间服务器并读取一个应答

1. 创建一个数据报插口

19-20 socket函数创建了一个UDP 插口，并且给进程返回一个保存在变量sockfd中的描述符。差错处理函数err_sys在[Stevens 1992]的附录B.2中给出。它接收任意数量的参数，并用vsprintf对它们格式化，将系统调用产生的 errno值对应的Unix错误信息打印出来，并中断进程。

我们在不同的地方使用术语插口：(1)为4.2BSD开发的程序用来访问网络协议的API通常叫插口API或者就叫插口接口；(2) socket是插口API中的一个函数的名字；(3)我们把调用socket创建的端点叫做一个插口，如评注“创建一个数据报插口”。

但是这里还有一些地方也使用术语插口：(4) socket函数的返回值叫一个插口描述符或者就叫一个插口；(5)在内核中的伯克利联网协议实现叫插口实现，相比较其他系统如：系统V的流实现。(6)一个IP地址和一个端口号的组合叫一个插口，IP地址和端口号对叫一个插口对。所幸的是引用哪一种术语是很明显的。

2. 将服务器地址放到结构sockaddr_in中

21-24 在一个互联网插口地址结构中存放日期/时间服务器的IP地址(140.252.1.32)和端口号(13)。大多数TCP/IP实现都提供标准的日期/时间服务器，它的端口号为13 [Stevens 1994，图1-9]。我们对服务器主机的选择是随意的——直接选择了提供此服务的本地主机(图1-17)。

函数inet_addr将一个点分十进制表示的IP地址的ASCII字符串转换成网络字节序的32 bit二进制整数。(Internet协议族的网络字节序是高字节在后)。函数htons把一个主机字节序的短整数(可能是低字节在后)转换成网络字节序(高字节在后)。在Sparc这种系统中，整数是高字节在后的格式，htons典型地是一个什么也不做的宏。但是在低字节在后的 80386上的BSD/386系统中，htons可能是一个宏或者是一个函数，来完成一个16 bit整数中的两个字节的交换。

3. 发送数据报给服务器

25-27 程序调用sendto发送一个150字节的数据报给服务器。因为是运行时栈中分配的未初始化数组，150字节的缓存内容是不确定的。但没有关系，因为服务器根本就不看它收到的报文的内容。当服务器收到一个报文时，就发送一个应答给客户端。应答中包含服务器以可读格式表示的当前时间和日期。

我们选择的150字节的客户数据报是随意的。我们有意选择一个报文长度在 100~208之间的值，来说明在本章的后面要提到的 mbuf链表的使用。为了避免拥塞，在以太网中，我们希望长度要小于1472。

4. 读取从服务器返回的数据报

28-32 程序通过调用recvfrom来读取从服务器发回的数据报。Unix服务器典型地发回一个如下格式的26字节字符串

```
Sat Dec 11 11:28:05 1993\r\n
```

\r是一个ASCII回车符，\n是ASCII换行符。我们的程序将回车符替换成一个空字节，然后调用printf输出结果。

在本章和下一章我们在分析函数socket、sendto和recvfrom的实现时，要进入这个例子的一些细节部分。

1.6 系统调用和库函数

所有的操作系统都提供服务访问点，程序可以通过它们请求内核中的服务。各种 Unix都提供精心定义的有限个内核入口点，即系统调用。我们不能改变系统调用，除非我们有内核的源代码。Unix第7版提供大约50个系统调用，4.4BSD提供大约135个，而SVR4大约有120个。

在《Unix程序员手册》第2节中有系统调用接口的文档。它是以C语言定义的，在任何给定的系统中无需考虑系统调用是如何被调用的。

在各种Unix系统中，每个系统调用在标准C函数库中都有一个相同名字的函数。一个应用程序用标准C的调用序列来调用此函数。这个函数再调用相应的内核服务，所使用的技术依赖于所在系统。例如，函数可能把一个或多个C参数放到通用寄存器中，并执行几条机器指令产生一个软件中断进入内核。对于我们来说，可以把系统调用看成C函数。

在《Unix程序员手册》的第3节中为程序员定义了一般用途的函数。虽然这些函数可能调用一个或多个内核系统调用但没有进入内核的入口点。如函数printf可能调用了系统调用write去执行输出，而函数strcpy(复制一个串)和atoi(将ASCII码转换成整数)完全不涉及操作系统。

从实现者的角度来看，一个系统调用和库函数有着根本的区别。但在用户看来区别并不严重。例如，在4.4BSD中我们运行图1-2中的程序。程序调用了三个函数：`socket`、`sendto`和`recvfrom`，每个函数最终调用了一个内核中同样名称的函数。在本书的后面我们可以看到这三个系统调用的BSD内核实现。

如果我们在SVR4中运行这个程序，在那里，用户库中的插口函数调用“流”子系统，那么三个函数同内核的相互作用是完全不同的。在SVR4中对`socket`的调用最终调用内核`open`系统调用，操作文件`/dev/udp`并将流模块`sockmod`放置到结果流。调用`sendto`导致一个`putmsg`系统调用，而调用`recvfrom`导致一个`getmsg`系统调用。这些SVR 4的细节在本书中并不重要，我们仅仅想指出的是：实现可能不同但都提供相同的API给应用程序。

最后，从一个版本到下一个版本的实现技术可能会改变。例如，在Net/1中，send和sendto是分别用内核系统调用实现的。但在Net/3中，send是一个调用系统调用sendto的库函数：

```
send(int s, char *msg, int len, int flags)
{
    return(sendto(s, msg, len, flags, (struct sockaddr *) NULL, 0));
}
```

用库函数实现`send`的好处是仅调用`sendto`，减少了系统调用的个数和内核代码的长度。缺点是由于多调用了一个函数，增加了进程调用send的开销。

因为本书是说明TCP/IP的伯克利实现的，大多数进程调用的函数（`socket`、`bind`、`connect`等)是直接由内核系统调用来实现。

1.7 网络实现概述

Net/3通过同时对多种通信协议的支持来提供通用的底层基础服务。的确，4.4BSD支持四种不同的通信协议族：

1) TCP/IP(互联网协议族)，本书的主题。

2) XNS(Xerox网络系统)，一个与TCP/IP相似的协议族；在80年代中期它被广泛应用于连

接Xerox设备(如打印机和文件服务器)，通常使用的是以太网。虽然Net/3仍然发布它的代码，但今天已很少使用这个协议了，并且很多使用伯克利 TCP/IP代码的厂商把XNS代码删去了(这样他们就不需要支持它了)。

3) OSI协议[Rose 1990；Piscitello and Chapin 1993]。这些协议是在80年代作为开放系统技术的最终目标而设计的，来代替所有其他通信协议。在 90年代初它没有什么吸引力，以致于在真正的网络中很少被使用。它的历史地位有待进一步确定。

4) Unix域协议。从通信协议是用来在不同的系统之间交换信息的意义上来说，它还不算是一套真正的协议，但它提供了一种进程间通信 (IPC)的形式。

相对于其他IPC，例如系统V消息队列，在同一主机上两个进程间的 IPC使用Unix域协议的好处是Unix域协议用与其他三种协议同样的 API(插口)访问。另一方面，消息队列和大多数其他形式IPC的API与插口和TLI完全不同。在同一主机上的两进程间的 IPC使用网络API，更容易将一个客户/服务器应用程序从一台主机移植到多台主机上。在 Unix域中提供两个不同的协议 —— 一个是可靠的，面向连接的，与 TCP相似的字节流协议；一个是不可靠的，无连接的，与UDP相似的数据报协议。

虽然Unix域协议可以作为一种同一主机上两进程间的 IPC，但也可以用TCP/IP来完成它们之间的通信。进程间通信并不要求使用在不同的主机上的互联网协议。

内核中的联网代码组织成三层，如图1-3所示。在图的右侧我们注明了 OSI参考模型[Piscitello和Chapin 1994]的七层分别对应到BSD组织的哪里。

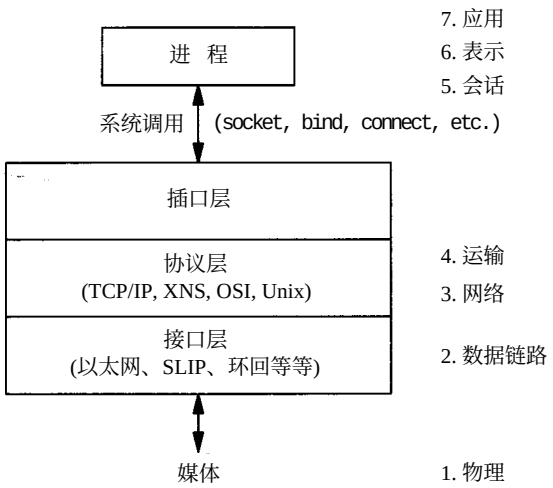


图1-3 Net/3联网代码的大概组织

1) 插口层是一个到下面协议相关层的协议无关接口。所有系统调用从协议无关的插口层开始。例如：在插口层中的 bind系统调用的协议无关代码包含几十行代码，它们验证的第一个参数是一个有效的插口描述符，并且第二个参数是一个进程中的有效指针。然后调用下层的协议相关代码，协议相关代码可能包含几百行代码。

2) 协议层包括我们前面提到的四种协议族(TCP/IP，XNS，OSI和Unix域)的实现。每个协议族可能包含自己的内部结构，在图 1-3中我们没有显示出来。例如，在 Internet协议族中，IP(网络层)是最低层，TCP和UDP两运输层在IP的上面。

3) 接口层包括同网络设备通信的设备驱动程序。

1.8 描述符

图1-2中，一开始调用`socket`，这要求定义插口类型。 Internet协议族(`PF_INET`)和数据报插口(`SOCK_DGRAM`)组合成一个UDP协议插口。

`socket`的返回值是一个描述符，它具有其他 `Unix`描述符的所有特性：可以用这个描述符调用`read`和`write`；可以用`dup`复制它，在调用了 `fork`后，父进程和子进程可以共享

它；可以调用fcntl来改变它的属性，可以调用colse来关闭它，等等。在我们的例子中可以看到插口描述符是函数 sendto和recvfrom的第一个参数。当程序终止时（通过调用exit），所有打开的描述符，包括插口描述符都会被内核关闭。

我们现在介绍在进程调用socket时被内核创建的数据结构。在后面的几章中会更详细地描述这些数据结构。

首先从进程的进程表表项开始。在每个进程的生存期内都会有一个对应的进程表表项存在。

一个描述符是进程的进程表表项中的一个数组的下标。这个数组项指向一个打开文件表的结构，这个结构又指向一个描述此文件的i-node或v-node结构。图1-4说明了这种关系。

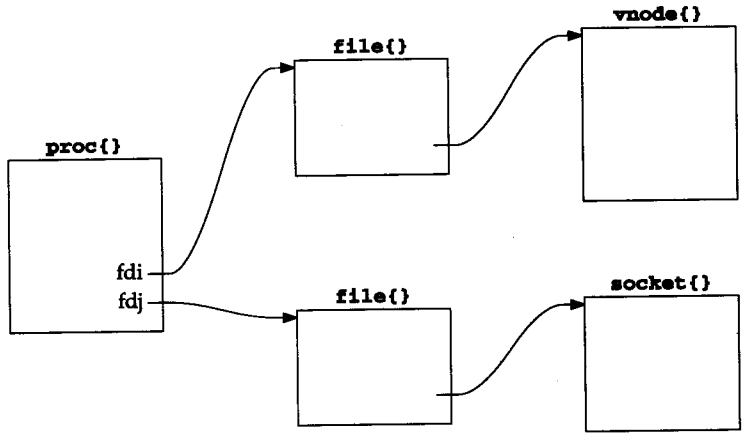


图1-4 从一个描述符开始的内核数据结构的基本关系

在这个图中，我们还显示了一个涉及插口的描述符，它是本书的焦点。由于进程表表项是由以下C语言定义的，我们把记号proc{}放在进程表项的上面。并且在本书所有的图中都用它来标注这个结构。

```
struct proc {
    ...
}
```

[Stevens 1992，3.10节]显示了当进程调用dup和fork时，描述符、文件表结构和i-node或v-node之间的关系是如何改变的。这三种数据结构的关系存在于所有版本的 Unix中，但不同的实现细节有所变化。在本书中我们感兴趣的是 socket结构和它所指向的Internet专用数据结构。但是既然插口系统调用以一个描述符开始，我们就需要理解如何从一个描述符导出一个socket结构。

如果程序如此执行

```
a.out
```

不重定向标准输入(描述符0)、标准输出(描述符1)和标准错误处理(描述符2)，图1-5显示了程序示例中的Net/3数据结构的更多细节。在这个例子中，描述符 0、1和2连接到我们的终端，并且当socket被调用时未用描述符的最小编号是3。

当进程执行了一个系统调用，如 socket，内核就访问进程表结构。在这个结构中的项p_fd指向进程的 filedesc结构。在这个结构中有两个我们现在关心的成员：一个是

fd_ofileflags，它是一个字符数组指针（每个描述符有一个描述符标志）；一个是fd_ofiles，它是一个指向文件表结构的指针数组的指针。描述符标志有 8 bit，只有两位可为任何描述符设置：close-on-exec标志和mapped-from-device标志。在这里我们显示的所有标志都是0。

由于Unix描述符与很多东西有关，除了文件外，还有：插口、管道、目录、设

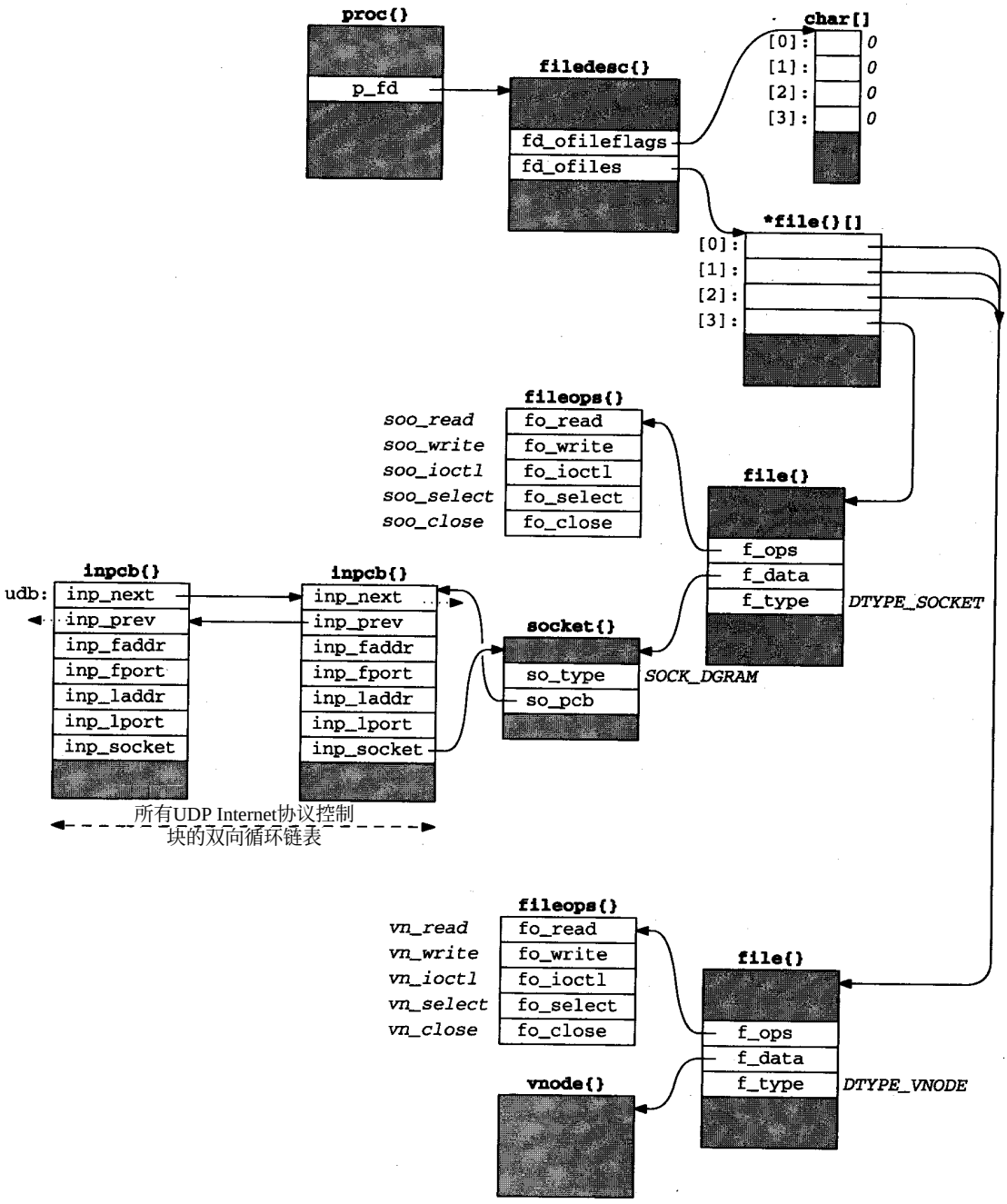


图1-5 在程序示例中调用socket后的内核数据结构

备等等，因此，我们有意把本节叫做“描述符”而不是“文件描述符”。但是很多Unix文献在谈到描述符时总是加上“文件”这个修饰词，其实没有必要。虽然我们要说明的是插口描述符，但这个内核数据结构叫 filedesc{}。我们尽可能地使用描述符这个未加修饰的术语。

项fd_ofiles指向的数据结构用*file{}[]来表示。它是一个指向file结构的指针数组。这个数组及描述符标志数组的下标就是描述符本身： 0、1、2等等，是非负整数。在图1-5中我们可以看到描述符 0、1、2对应的项指向图底部的同一个 file结构(由于这三个描述符都对应终端设备)。描述符3对应的项指向另外一个file结构。

结构file的成员f_type指示描述符的类型是 DTYPE_SOCKET和DTYPE_VNODE。v-node是一个通用机制，允许内核支持不同类型的文件系统 —— 磁盘文件系统、网络文件系统(如NFS)、CD-ROM文件系统、基于存储器的文件系统等等。在本书中关心的不是 v-node，因为TCP/IP插口的类型总是DTYPE_SOCKET。

结构file的成员f_data指向一个socket结构或者一个vnode结构，根据描述符类型而定。成员 f_ops指向一个有 5个函数指针的向量。这些函数指针用在 read、readv、write、writev、ioctl、select和close系统调用中，这些系统调用需要一个插口描述符或非插口描述符。这些系统调用每次被调用时都要查看 f_type的值，然后做出相应的跳转，实现者选择了直接通过fileops结构的相应项来跳转的方式。

我们用一个等宽字体 (fo_read)来醒目地表示一个结构成员的名称，用斜体等宽字体(soo_read)来表示一个结构成员的内容。注意，有时我们用一个箭头指向一个结构的左上角(如结构filedesc)，有时用一个箭头指向右上角 (如结构file和fileops)。我们用这些方法来简化图例。

下面我们来查看结构socket，当描述符的类型是DTYPE_SOCKET时，结构file指向结构socket。在我们的例子中， socket的类型(数据报插口的类型是SOCK_DGRAM)保存在成员so_type中。还分配了一个Internet协议控制块(PCB)：一个inpcb结构。结构socket的成员so_pcb指向inpcb，并且结构inpcb的成员inp_socket指向结构socket。对于一个给定插口的操作可能来自两个方向：“上”或“下”，因此需要有指针来互相指向。

1) 当进程执行一个系统调用时，如 sendto，内核从描述符值开始，使用 fd_ofiles索引到file结构指针向量，直到描述符所对应的 file结构。结构file指向socket结构，结构socket带有指向结构inpcb的指针。

2) 当一个UDP数据报到达一个网络接口时，内核搜索所有 UDP协议控制块，寻找一个合适的，至少要根据目标 UDP端口号，可能还要根据目标 IP地址、源IP地址和源端口号。一旦定位所找的inpcb，内核就能通过inp_socket指针来找到相应的socket结构。

成员inp_faddr和inp_laddr包含远地和本地 IP地址，而成员 inp_fport和inp_lport包含远地和本地端口号。 IP地址和端口号的组合经常叫做一个插口。

在图1-5的左边，我们用名称udb来标注另一个inpcb结构。这是一个全局结构，它是所有UDP PCB组成的链表表头。我们可以看到两个成员 inp_next和inp_prev把所有的UDP PCB组成了一个双向环型链表。为了简化此图，我们用两条平行的水平箭头来表示两条链，而不是用箭头指向PCB的顶角。右边的inpcb结构的成员inp_prev指向结构udb，而不是它的成员inp_prev。来自udb.inp_prev和另一个PCB成员inp_next的虚线箭头表示这里

还有其他PCB在这个双链表上，但我们没有画出。

在本章，我们已看了不少内核数据结构，大多数还要在后续章节中说明。现在要理解的关键是：

1) 我们的进程调用socket，最后分配了最小未用的描述符(在我们的例子中是3)。在后面，所有针对此socket的系统调用都要用这个描述符。

2) 以下内核数据结构是一起被分配和链接起来的：一个DTYPE_SOCKET类型file结构、一个socket结构和一个inpcb结构。这些结构的很多初始化过程我们并没有说明：file结构的读写标志(因为调用socket总是返回一个可读或可写的描述符)；默认的输入和输出缓存大小被设置在socket结构中，等等。

3) 我们显示了标准输入、输出和标准错误处理的非socket描述符的目的是为了说明所有描述符最后都对应一个file结构，虽然socket描述符和其他描述符之间有所不同。

1.9 mbuf与输出处理

在伯克利联网代码设计中的一个基本概念就是存储器缓存，称作一个mbuf，在整个联网代码中用于存储各种信息。通过我们的简单例子(图1-2)分析一些mbuf的典型用法。在第2章中我们会更详细地说明mbuf。

1.9.1 包含插口地址结构的mbuf

在sendto调用中，第5个参数指向一个Internet插口地址结构(叫serv)，第6个参数指示它的长度(后面我们将要看到是16个字节)。插口层为这个系统调用做的第一件事就是验证这些参数是有效的(即这个指针指向进程地址空间的一段存储器)，并且将插口地址结构复制到一个mbuf中。图1-6所示的是这个所得到的mbuf。

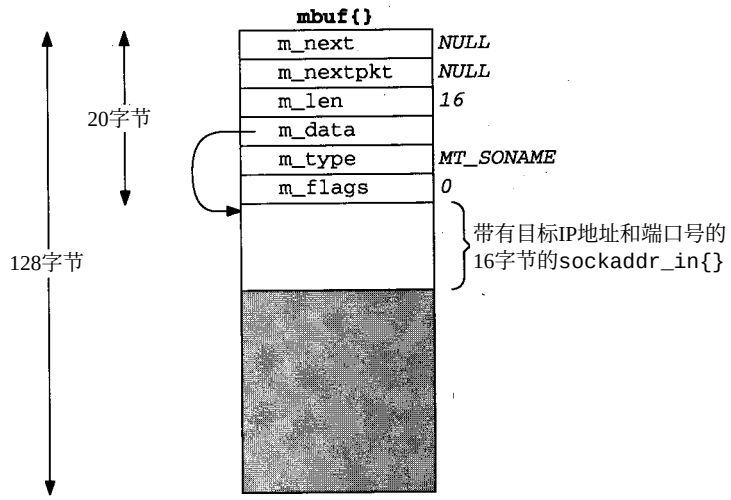


图1-6 mbuf中针对sendto的目的地址

mbuf的前20个字节是首部，它包含关于这个mbuf的一些信息。这20个字节的首部包括四个4字节字段和两个2字节字段。mbuf的总长为128个字节。

稍后我们会看到，mbuf可以用成员m_next和m_nextpkt链接起来。在这个例子中都是

空指针，它是一个独立的mbuf。

成员m_data指向mbuf中的数据，成员m_len指示它的长度。对于这个例子， m_data指向mbuf中数据的第一个字节(紧接着mbuf首部)。mbuf后面的92个字节(108-16)没有用(图1-6的阴影部分)。

成员m_type指示包含在mbuf中数据的类型，在本例中是MT_SONAME(插口名称)。首部的最后一个成员m_flags，在本例中是零。

1.9.2 包含数据的mbuf

下面继续讨论我们的例子，插口层将sendto调用中指定的数据缓存中的数据复制到一个或多个mbuf中。sendto的第二个参数指示了数据缓存(buff)的开始位置，第三个参数是它的大小(150字节)。图1-7显示了150字节的数据是如何存储在两个 mbuf 中的。

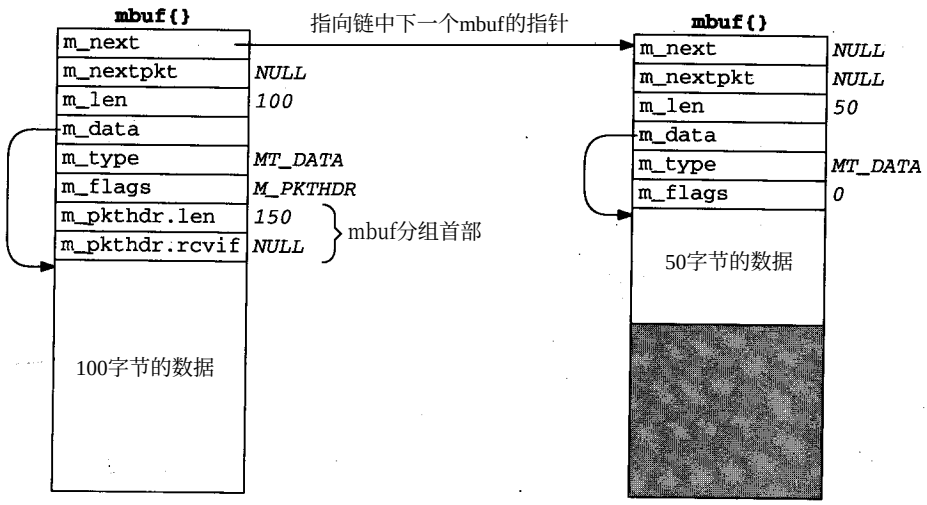


图1-7 用两个mbuf来存储150字节的数据

这种安排叫做mbuf链表。在每个mbuf中的成员m_next把链表中所有的mbuf都链接在一起。

我们看到的另一个变化是链表中第一个mbuf的mbuf首部的另外两个成员：m_pkthdr.len和m_pkthdr.rcvif。这两个成员组成了分组首部并且只用在链表的第一个 mbuf中。成员m_flags的值是M_PKTHDR，指示这个mbuf包含一个分组首部。分组首部结构的成员len包含了整个mbuf链表的总长度(在本例中是150)，下一个成员rcvif在后面我们会看到，它包含了一个指向接收分组的接收接口结构的指针。

因为mbuf总是128个字节，在链表的第一个mbuf中提供了100字节的数据存储能力，而后面所有的mbuf有108字节的存储空间。在本例中的两个 mbuf需要存储150字节的数据。我们稍后会看到当数据超过208字节时，就需要3个或更多的mbuf。有一种不同的技术叫“簇”，一种大缓存，典型的有1024或2048字节。

在链表的第一个mbuf中维护一个带有总长度的分组首部的原因是，当需要总长度时可以避免查看所有mbuf中的m_len来求和。

1.9.3 添加IP和UDP首部

在插口层将目标插口地址结构复制到一个 mbuf中，并把数据复制到mbuf链中后，与此插口描述符(一个UDP描述符)对应的协议层被调用。明确地说， UDP输出例程被调用，指向mbuf的指针被作为一个参数传递。这个例程要在这 150字节数据的前面添加一个IP首部和一个UDP首部，然后将这些mbuf传递给IP输出例程。

在图1-7中的mbuf链表中添加这些数据的方法是分配另外一个 mbuf，把它放在链首，并将分组首部从带有100字节数据的mbuf复制到这个mbuf。在图1-8中显示了这三个mbuf。

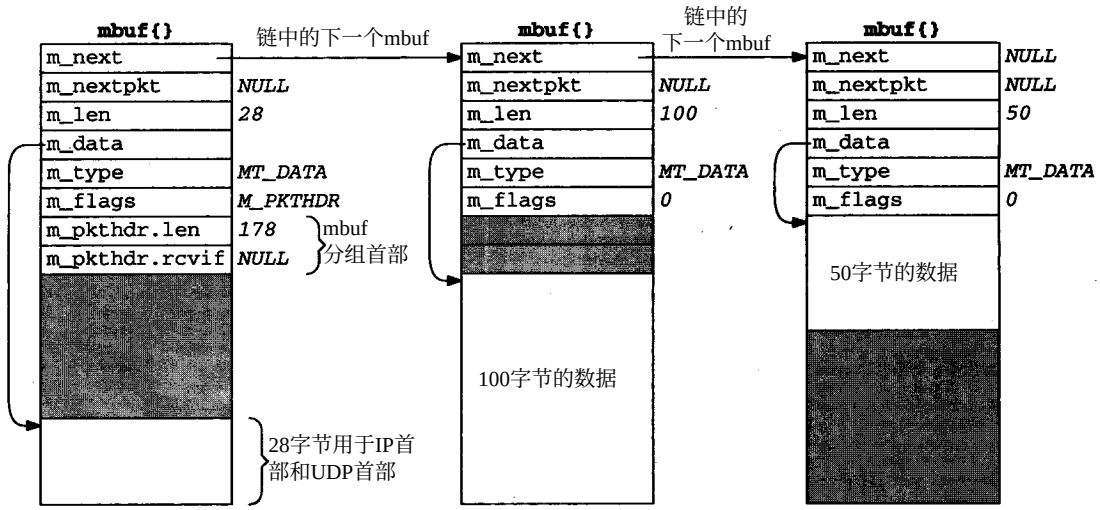


图1-8 在图1-7中的mbuf链表中添加另一个带有IP和UDP首部的mbuf

IP首部和UDP首部被放置在新 mbuf的最后，这个新 mbuf就成了整个链表的首部。如果需要，它允许任何其他低层协议 (例如接口层)在IP首部前添加自己的首部，而不需要再复制IP和UDP首部。在第一个 mbuf中的m_data指针指向这两个首部的起始位置， m_len的值是28。在分组首部和 IP首部之间有 72字节的未用空间留给以后的首部，通过适当地修改m_data指针和m_len添加在IP首部的前面。稍后我们会看见以太网首部就是用这种方法建立的。

注意，分组首部已从带有 100字节数据的mbuf中移到新mbuf中去了。分组首部必须放在mbuf链表的第一个mbuf中。在移动分组首部的同时，在第一个 mbuf设置M_PKTHDR标志并且在第二个mbuf中清除此标志。在第二个 mbuf中分组首部占用的空间现在未用。最后，在此分组首部中的长度成员由于增加了 28 字节而变成了178。

然后UDP输出例程填写 UDP首部和IP首部中它们所能填写的部分。例如， IP首部中的目标地址可以被设置，但IP检验和要留给IP输出例程来计算和存放。

UDP检验和计算后存储在UDP首部中。注意，这要求遍历存储在 mbuf链表中的所有150字节的数据。这样，内核要对这 150字节的用户数据做两次遍历：一次是把用户缓存中的数据复制到内核中的mbuf中，而现在是计算UDP检验和。对整个数据的额外遍历会降低协议的性能，在后续章节中我们会介绍另一种可选的实现技术，它可以避免不必要的遍历。

接着，UDP输出例程调用IP输出例程，并把此mbuf链表的指针传递给IP输出例程。

1.9.4 IP输出

IP输出例程要填写IP首部中剩余的字段，包括IP检验和；确定数据报应发到哪个输出接口(这是IP路由功能)；必要时，对IP报文分片；以及调用接口输出函数。

假设输出接口是一个以太网接口，再次把此 mbuf链表的指针作为一个参数，调用一个通用的以太网输出函数。

1.9.5 以太网输出

以太网输出函数的第一个功能就是把 32位IP地址转换成相应的48位以太网地址。在使用ARP(地址解析协议)时会使用这个功能，并且会在以太网上发送一个 ARP请求并等待一个ARP应答。此时，要输出的mbuf链表已得到，并等待应答。

然后以太网输出例程把一个 14字节的以太网首部添加到链表的第一个 mbuf中，紧接在IP首部的前面(图1-8)。以太网首部包括6字节以太网目标地址、 6字节以太网源地址和2字节以太网帧类型。

之后此mbuf链表被加到此接口的输出队列队尾。如果接口不忙，接口的“开始输出”例程立即被调用。若接口忙，在它处理完输出队列中的其他缓存后，它的输出例程会处理队列中的这个新mbuf。

当接口处理它输出队列中的一个 mbuf时，它把数据复制到它的传输缓存中，并且开始输出。在我们的例子中， 192字节被复制到传输缓存中： 14字节以太网首部、 20字节IP首部、 8字节UDP首部及150字节用户数据。这是内核第三次遍历这些数据。一旦数据从 mbuf链表被复制到设备传输缓存， mbuf链表就被以太网设备驱动程序释放。这三个 mbuf被放回到内核的自由缓存池中。

1.9.6 UDP输出小结

我们在图1-9中给出了一个进程调用 sendto传输一个UDP数据报时的大致处理过程。在图中我们说明的处理过程与三层内核代码 (图1-3)的关系也显示出来了。

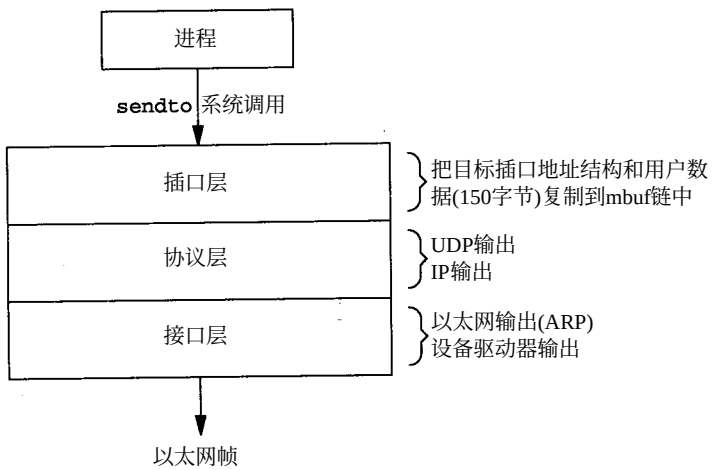


图1-9 三层处理一个简单UDP输出的执行过程

函数调用控制从插口层到UDP输出例程，到IP输出例程，然后到以太网输出例程。每个函数调用传递一个指向要输出的mbuf的指针。在最低层，设备驱动程序层，mbuf链表被放置到设备输出队列并启动设备。函数调用按调用的相反顺序返回，最后系统调用返回给进程。注意，直到UDP数据报到达设备驱动程序前，UDP数据没有排队。高层仅仅添加它们的协议首部并把mbuf传递给下一层。

这时，在我们的程序示例中调用recvfrom去读取服务器的应答。因为该插口的输入队列是空的(假设应答还没有到达)，进程就进入睡眠状态。

1.10 输入处理

输入处理与刚讲过的输出处理不同，因为输入是异步的。就是说，它是通过一个接收完成中断驱动以太网设备驱动程序来接收一个输入分组，而不是通过进程的系统调用。内核处理这个设备中断，并调度设备驱动程序进入运行状态。

1.10.1 以太网输入

以太网设备驱动程序处理这个中断，假定它表示一个正常的接收已完成，数据从设备读到一个mbuf链表中。在我们的例子中，接收了54字节的数据并复制到一个mbuf中：20字节IP首部、8字节UDP首部及26字节数据(服务器的时间与日期)。图1-10所示的是这个mbuf的格式。

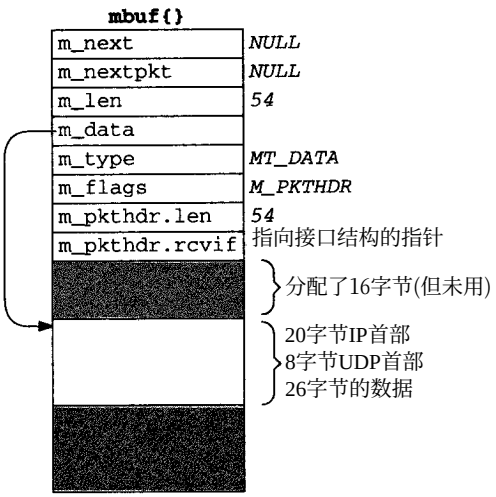


图1-10 用一个mbuf存储输入的以太网数据

这个mbuf是一个分组首部(m_flags被设置成M_PKTHDR)，它是一个数据记录的第一个mbuf。分组首部的成员len包含数据的总长度，成员rcvif包含一个指针，它指向接收数据的接口的接口结构(第3章)。我们可以看到成员rcvif用于接收分组而不是输出分组(图1-7和图1-8)。

mbuf的前16字节数据空间被分配给一个接口层首部，但没有使用。数据就存储在这个mbuf中，54字节的数据存储在剩余的84字节的空间中。

设备驱动程序把mbuf传给一个通用以太网输入例程，它通过以太网帧中的类型字段来确定哪个协议层来接收此分组。在这个例子中，类型字段标识一个IP数据报，从而mbuf被加入到IP输入队列中。另外，会产生一个软中断来执行IP输入例程。这样，这个设备中断处理就完成了。

1.10.2 IP输入

IP输入是异步的，并且通过一个软中断来执行。当接口层在系统的一个接口上收到一个IP数据报时，它就设置这个软中断。当IP输入例程执行它时，循环处理在它的输入队列中的每一个IP数据报，并在整个队列被处理完后返回。

IP输入例程处理每个接收到的IP数据报。它验证IP首部检验和，处理IP选项，验证数据报

被传递到正确的主机(通过比较数据报的目标IP地址与主机IP地址)，并当系统被配置为一个路由器，且数据报被表注为其他的IP地址时，转发此数据报。如果IP数据报到达它的最终目标，调用IP首部中标识的协议的输入例程： ICMP，IGMP，TCP或UDP。在我们的例子中，调用UDP输入例程去处理UDP数据报。

1.10.3 UDP输入

UDP输入例程验证UDP首部中的各字段(长度与可选的检验和)，然后确定是否一个进程应该接收此数据报。在第23章我们要详细讨论这个检查是如何进行的。一个进程可以接收到一指定UDP端口的所有数据报，或让内核根据源与目标 IP地址及源与目标端口号来限制数据报的接收。

在我们的例子中， UDP输入例程从一个全局变量udb(图1-5)开始，查看所有UDP协议控制块链表，寻找一个本地端口号(inp_lport)与接收的UDP数据报的目标端口号匹配的协议控制块。这个PCB是由我们调用socket创建的，它的成员inp_socket指向相应插口结构，并允许接收的数据在此插口排队。

> 在程序示例中，我们从未为应用程序指定本地端口号。在习题 23.3中，我们会看到在写第一个UDP程序时创建一个插口而不绑定一个本地端口号会导致内核自动地给此插口分配一个本地端口号 (称为短期端口)。这就是为什么插口的 PCB成员inp_lport不是一个空值的原因。

因为这个UDP数据报要传递给我们的进程，发送方的 IP地址和UDP端口号被放置到一个mbuf中，这个mbuf和数据(在我们的例子中是26字节)被追加到此插口的接收队列中。图 1-11所示的是被追加到这个插口的接收队列中的这两个 mbuf。

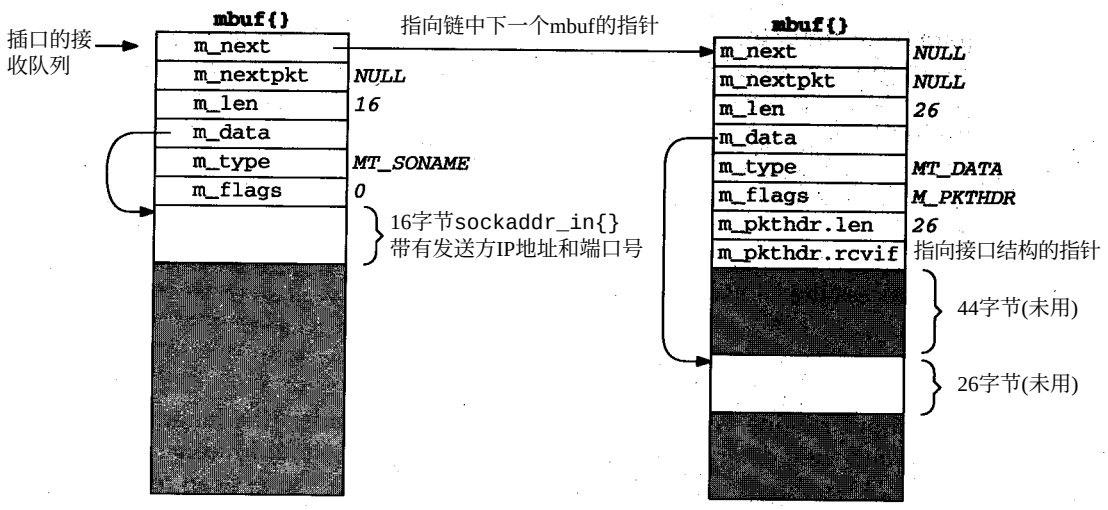


图1-11 发送方地址和数据

比较这个链表中的第二个 mbuf(MT_DATA类型)与图1-10中的mbuf，成员m_len和m_pkthdr.len都减小了28字节(20字节的IP首部和8字节UDP首部)，并且指针m_data也减小了28字节。这有效地将IP和UDP首部删去，只保留了26字节数据追加到插口接收队列。

在链表的第一个mbuf中包括一个16字节Internet插口地址结构，它带有发送方IP地址和UDP端口号。它的类型是MT_SONAME，与图1-6中的mbuf类似。这个mbuf是插口层创建的，将这些信息返回给通过调用系统调用recvform或recvmsg的调用进程。即使在这个链表的第二个mbuf中有空间(16字节)存储这个插口地址结构，它也必须存放到它自己的mbuf中，因为它们的类型不同(一个是MT_SONAME，一个是MT_DATA)。

然后接收进程被唤醒。如果进程处于睡眠状态等待数据的到达(我们例子中的情况)，进程被标志为可运行状态等待内核的调度。也可以通过select系统调用或SIGIO信号来通知进程数据的到达。

1.10.4 进程输入

我们的进程调用recvfrom时被阻塞，在内核中处于睡眠状态，现在进程被唤醒。UDP层追加到插口接收队列中的26字节的数据(接收的数据报)被内核从mbuf复制到我们程序的缓存中。

注意，我们的程序把recvform的第5，第6个参数设置为空指针，告诉系统在接收过程中不关心发送方的IP地址和UDP端口号。这使得系统调用recvfrom时，略过链表中的第一个mbuf(图1-11)，仅返回第二个mbuf中的26字节的数据。然后内核的recvfrom代码释放图1-11中的两个mbuf，并把它们放回到自由mbuf池中。

1.11 网络实现概述(续)

图1-12总结了在各层间为网络输入输出而进行的通信。图1-12是对图1-3进行了重画，它只考虑Internet协议，并且强调层间的通信。符号splnet与splimp在下一节讨论。

我们使用复数术语插口队列(socket queues)和接口队列(interface queues)，因为每个插口

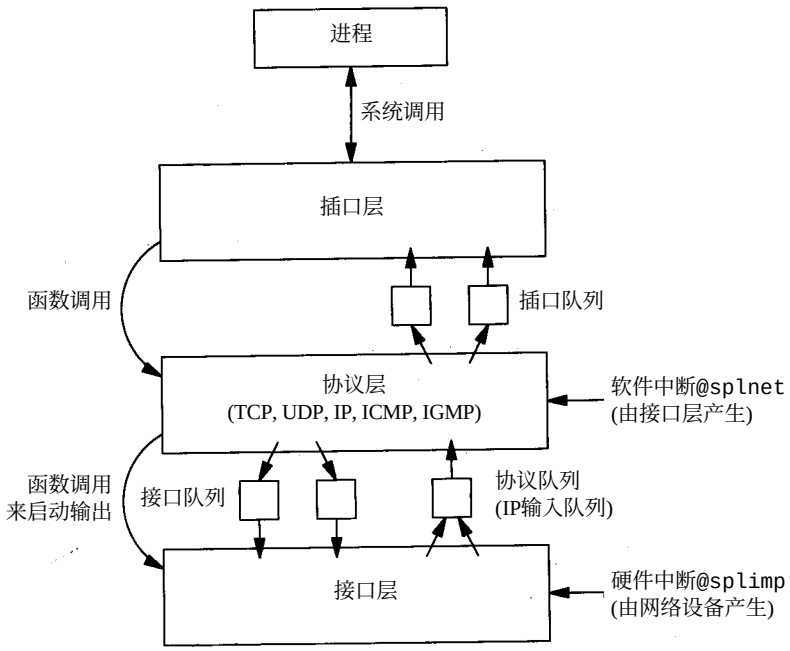


图1-12 网络输入输出的层间通信

和每个接口 (以太网、环回、 SLIP、PPP等)都有一个队列，但我们使用单数术语协议队列(protocol queue)，因为只有一个IP输入队列。如果考虑其他协议层，我们就会有一个队列用于XNS协议，一个队列用于OSI协议。

1.12 中断级别与并发

我们在1.10节看到网络代码处理输入分组用的是异步和中断驱动的方式。首先，一个设备中断引发接口层代码执行，然后它产生一个软中断引发协议层代码执行。当内核完成这些级别的中断后，执行插口代码。

在这里给每个硬件和软件中断分配一个优先级。图 1-13所示的是8个优先级别的顺序，从最低级别(不阻塞中断)到最高级别(阻塞所有中断)。

函　数	说　明
spl0	正常操作方式，不阻塞中断　(最低优先级)
Splsoftclock	低优先级时钟处理
splnet	网络协议处理
spltty	终端输入输出
splbio	磁盘与磁带输入输出
splimp	网络设备输入输出
splclock	高优先级时钟处理
splhigh	阻塞所有中断　(最高优先级)
splx(s)	(见正文)

图1-13　阻塞所选中断的内核函数

[Leffler et al. 1989]的表4-5显示了用于VAX实现的优先级别。386的Net/3的实现使用图1-13所示的8个函数，但splsoftclock与splnet在同一级别，splclock与splhigh也在同一级别。

用于网络接口级的名称 *imp*来自于缩写IMP(接口报文处理器)，它是在ARPANET中使用的路由器的最初类型。

不同优先级的顺序意味着高优先级中断可以抢占一个低优先级中断。看图 1-14所示的事

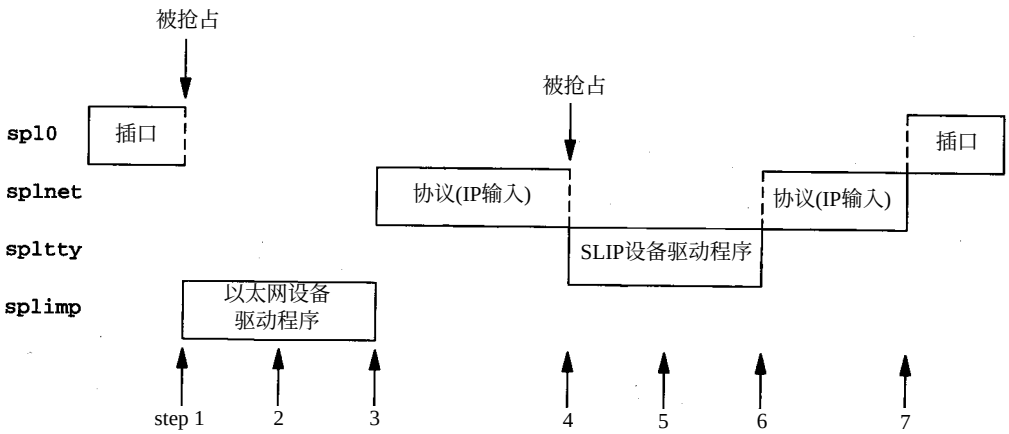


图1-14　优先级示例与内核处理

件顺序。

1) 当插口层以级别spl0执行时，一个以太网设备驱动程序中断发生，使接口层以级别splimp执行。这个中断抢占了插口层代码的执行。这就是异步执行接口输入例程。

2) 当以太网设备驱动程序在运行时，它把一个接收的分组放置到 IP输出队列中并调度一个splnet级别的软中断。软中断不会立即有效，因为内核正在一个更高的优先级 (splimp)上运行。

3) 当以太网设备驱动程序完成后，协议层以级别splnet执行。这就是异步执行IP输入例程。

4) 一个终端设备中断发生 (完成一个SLIP分组)，它立即被处理，抢占协议层，因为终端输入/输出(spltty)优先级比图1-13中的协议层(splnet)更高。

5) SLIP驱动程序把接收的分组放到IP输入队列中并为协议层调度另一个软中断。

6) 当SLIP驱动程序结束时，被抢占的协议层继续以级别 splnet执行，处理完从以太网设备驱动程序收到的分组后，处理从 SLIP驱动程序接收的分组。仅当没有其他输入分组要处理时，它会把控制权交还给被它抢占的进程 (在本例中是插口层)。

7) 插口层从它被中断的地方继续执行。

对于这些不同优先级，一个要关心的问题就是如何处理那些在不同级别的进程间共享的数据结构。在图1-2中显示了三种在不同优先级进程间共享的数据结构 —— 插口队列、接口队列和协议队列。例如，当 IP输入例程正在从它的输入队列中取出一个接收的分组时，一个设备中断发生，抢占了协议层，并且那个设备驱动程序可能添加另一个分组到 IP输入队列。这些共享的数据结构(本例中的IP输入队列，它共享于协议层和接口层)，如果不协调对它们的访问，可能会破坏数据的完整性。

Net/3的代码经常调用函数splimp和splnet。这两个调用总是与splx成对出现，splx使处理器返回到原来的优先级。例如下面这段代码，被协议层 IP输入函数执行，去检查是否有其他分组在它的输入队列中等待处理：

```
struct mbuf *m;
int s;

s = splimp ();
IF_DEQUEUE (&ipintrq, m);
splx(s);
if (m == 0)
   return;
```

调用splimp把CPU的优先级升高到网络设备驱动程序级，防止任何网络设备驱动程序中断发生。原来的优先级作为函数的返回值存储到变量 s中。然后执行宏IF_DEQUEUE把IP输入队列(ipintrq)头部的第二个分组删去，并把指向此 mbuf链表的指针放到变量 m中。最后，通过调用带有参数 s(其保存着前面调用 splimp的返回值)的splx，CPU的优先级恢复到调用splimp前的级别。

由于在调用splimp和splx之间所有的网络设备驱动程序的中断被禁止，在这两个调用间的代码应尽可能的少。如果中断被禁止过长的时间，其他设备会被忽略，数据会被丢失。因此，对变量m的测试(看是否有其他分组要处理)被放在调用splx之后而不是之前。

当以太网输出例程把一个要输出的分组放到一个接口队列，并测试接口当前是否忙时，若接口不忙则启动接口，这时例程需要调用这些 spl调用。

```
struct mbuf  *m;
int  s;

s = splimp();
/*
 * Queue message on interface, and start output if interface not active.
 */
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);     /* queue is full, drop packet */
    splx(s);
    error = ENOBUFS;
    goto bad;
}

IF_ENQUEUE(&ifp->if_snd, m);  /* add the packet to interface queue */

if ((ifp->if_flags & IFF_OACTIVE) == 0)
    (*ifp->if_start)(ifp);     /* start interface */

splx(s);
```

在这个例子中，设备中断被禁止的原因是防止在协议层正在往队列添加分组时，设备驱动程序从它的发送队列中取走下一个分组。设备发送队列是一个在协议层和接口层共享的数据结构。

在整个源代码中到处都会看到spl函数。

1.13 源代码组织

图1-15所示的是Net/3网络源代码的组织，假设它位于目录/usr/src/sys。

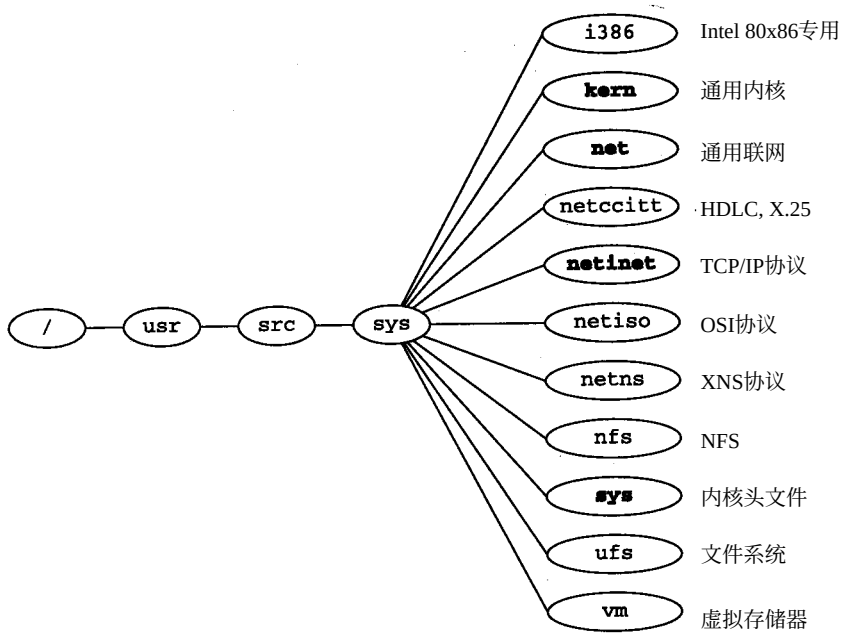


图1-15 Net/3源代码组织

本书的重点在目录netinet，它包含所有TCP/IP源代码。在目录kern和net中我们也可找到一些文件。前者是协议无关的插口代码，而后者是一些通用联网函数，用于TCP/IP例程，如路由代码。

包含在每个目录中的文件简要地列于下面：

- i386：因特80x86专用目录。例如，目录i386/isa包含专用于ISA总线的设备驱动程序。目录i386/stand包含单机引导程序代码。
- kern：通用的内核文件，不属于其他目录。例如，处理系统调用 fork和exec的内核文件在这个目录。在这个目录中，我们只考察少数几个文件——用于插口系统调用的文件(插口层在图1-3)。
- net：通用联网文件，例如，通用联网接口函数， BPF(BSD分组过滤器)代码、SLIP驱动程序和路由代码。在这个目录中我们考察一些文件。
- netccitt：OSI协议接口代码，包括HDLC(高级数据链路控制)和X.25驱动程序。
- netinet：Internet协议代码：IP，ICMP，IGMP，TCP和UDP。本书的重点集中在这个目录中的文件。
- netiso：OSI协议。
- netns：施乐(Xerox)XNS协议。
- nfs：SUN公司的网络文件系统代码。
- sys：系统头文件。在这个目录中我们考察几个头文件。这个目录中的文件还出现在目录/usr/include/sys中。
- ufs：Unix文件系统的代码，有时叫伯克利快速文件系统。它是标准磁盘文件系统。
- vm：虚拟存储器系统代码。

图1-16所示的是源代码组织的另一种表现形式，它映射到我们的三个内核层。忽略netimp和nfs这样的目录，在本书中我们不关心它们。

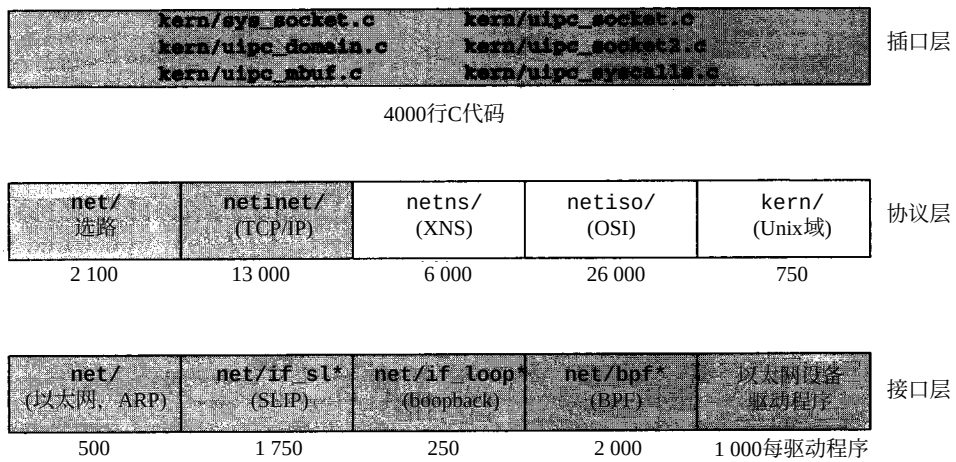


图1-16 映射到三个内核层的Net/3源代码组织

在每个表格框底下的数字是对应功能的C代码的近似行数，包括源文件中的所有注释。

我们不考察图中所有的源代码。显示目录 netns与netiso是为了与Internet协议比较。我们仅考虑有阴影的表格框。

1.14 测试网络

图1-17所示的测试网络用于本书中所有的例子。除了在图顶部的主机 vangogh，所有的

IP地址属于B类网络地址140.252，并且所有主机名属于域.tuc.noao.edu (noao代表“国家光学天文台”，tuc代表Tucson)。例如，在右下角的系统的主机全名是svr4.tuc.noao.edu，IP地址是140.252.13.34。在每个框图顶上的记号是运行在此系统上操作系统的名称。

在图顶的主机的全名是vangogh.cs.berkeley.edu，其他主机通过Internet可以连接到它。

这个图与卷1中的测试网络几乎一样，有一些操作系统升级了，在sun与netb之间的拨号链路现在用PPP取代了SLIP。另外，我们用Net/3网络代码代替了BSD/386 V1.1提供的Net/2网络代码。

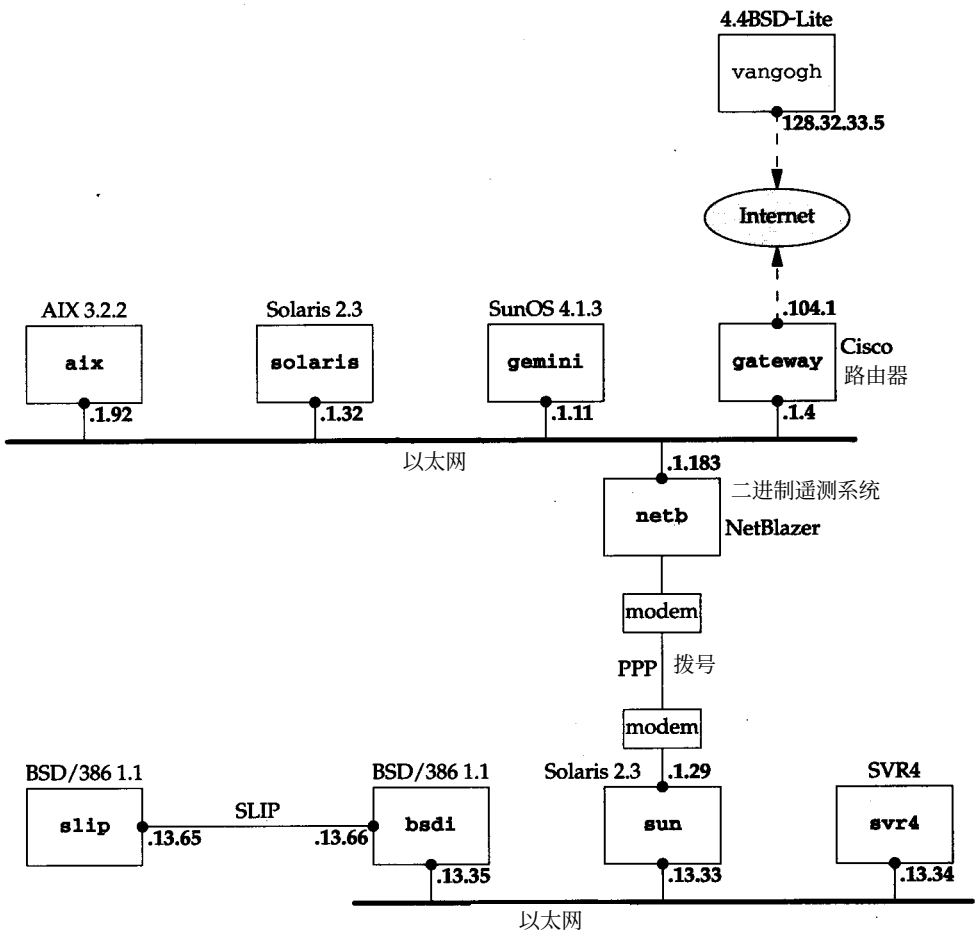


图1-17 用于本书中所有例子的测试网络

1.15 小结

本章是对Net/3网络代码的概述。通过一个简单的程序示例 (图1-2) —— 发送一个UDP数据报给一个日期时间服务器并接收应答，我们分析了通过内核进行输入输出的过程。 mbuf中保存要输出的信息和接收的IP数据报。下一章我们要查看mbuf的更多细节。

当进程执行sendto系统调用时，产生UDP输出，而IP输入是异步的。当一个设备驱动程

序接收了一个IP数据报，数据报被放到IP输入队列中并且产生一个软中断使IP输入函数执行。我们考察了在内核中用于联网代码的不同中断级别。由于很多联网数据结构被不同的层所共享，而这些层在不同的中断级别上执行，因此当访问或修改这些共享结构时要特别小心。几乎所有我们要查看的函数中都会遇到spl函数。

本章结束时我们查看了Net/3源代码的整个组织结构，及本书关注的代码。

习题

1.1 输入程序示例(图1-2)并在你的系统上运行。如果你的系统有系统调用跟踪能力，如`trace` (SunOS 4.x)、`truss` (SVR4)或`ktrace` (4.4BSD)，用它检测本例中调用的系统调用。

1.2 在1.12节调用`IF_DEQUEUE`的例子中，我们注意到调用`splimp`来防止网络设备驱动程序的中断。当以太网驱动程序以这个级别执行时， SLIP驱动程序会发生什么？

第2章 mbuf：存储器缓存

2.1 引言

网络协议对内核的存储器管理能力提出了很多要求。这些要求包括能方便地操作可变长缓存，能在缓存头部和尾部添加数据 (如低层封装来自高层的数据)，能从缓存中移去数据 (如，当数据分组向上经过协议栈时要去掉首部)，并能尽量减少为这些操作所做的数据复制。内核中的存储器管理调度直接关系到联网协议的性能。

在第1章我们介绍了普遍应用于Net/3内核中的存储器缓存：mbuf，它是“ memory buffer ”的缩写。在本章，我们要查看 mbuf和内核中用于操作它们的函数的更多的细节，在本书中几乎每一页我们都会遇到mbuf。要理解本书的其他部分必须先要理解 mbuf。

mbuf的主要用途是保存在进程和网络接口间互相传递的用户数据。但 mbuf也用于保存其他各种数据：源与目标地址、插口选项等等。

图2-1显示了我们要遇到的四种不同类型的 mbuf，它们依据在成员 `m_flags`中填写的不同标志`M_PKTHDR`和`M_EXT`而不同。图2-1中四个mbuf的区别从左到右罗列如下：

1) 如果`m_flags`等于0，mbuf只包含数据。在mbuf中有108字节的数据空间(`m_dat`数组)。指针`m_data`指向这108字节缓存中的某个位置。我们所示的 `m_data`指向缓存的起始，但它能指向缓存中的任意位置。成员 `m_len`指示了从`m_data`开始的数据的字节数。图1-6是这类mbuf的一个例子。

   在图2-1中，结构`m_hdr`中有六个成员，它的总长是 20字节。当我们查看此结构的 C语言定义时，会看见前四个成员每个占用 4字节而后两个成员每个占用 2字节。在图2-1中我们没有区分4字节成员和2字节成员。

2) 第二类mbuf的`m_flags`值是`M_PKTHDR`，它指示这是一个分组首部，描述一个分组数据的第一个 mbuf。数据仍然保存在这个 mbuf中，但是由于分组首部占用了 8字节，只有100字节的数据可存储在这个 mbuf中(在`m_pktdat`数组中)。图1-10是这种mbuf的一个例子。

   成员`m_pkthdr.len`的值是这个分组的 mbuf链表中所有数据的总长度：即所有通过`m_next`指针链接的 mbuf的`m_len`值的和，如图 1-8所示。输出分组没有使用成员`m_pkthdr.rcvif`，但对于接收的分组，它包含一个指向接收接口 `ifnet`结构(图3-6)的指针。

3) 下一种mbuf不包含分组首部(没有设置`K_PKTHDR`)，但包含超过208字节的数据，这时用到一个叫“ 簇 ”的外部缓存 (设置`M_EXT`)。在此mbuf中仍然为分组首部结构分配了空间，但没有用 —— 在图2-1中，我们用阴影显示出来。 Net/3分配一个大小为 1024或2048字节的簇，而不是使用多个 mbuf来保存数据(第一个带有100字节数据，其余的每个带有108字节数据)。在这个mbuf中，指针`m_data`指向这个簇中的某个位置。

   Net/3版本支持七种不同的结构。定义了四种 1024字节的簇(惯例值)，三种2048字节的

簇。传统上用1024字节的原因是为了节约存储器：如果簇的大小是 2048，对于以太网分组(最大1500字节)，每个簇大约有四分之一没有用。在 27.5节中我们会看到 Net/3 TCP发送的每个 TCP报文段从来不超过一簇大小，因为当簇的大小为 1024时，每个1500字节的以太网帧几乎三分之一未用。但是 [Mogul 1993，图15-15]显示了当在以太网中发送最大帧而不是1024字节的帧时能明显提高以太网的性能。这就是一种性能 /存储器互换。老的系统使用 1024字节簇来节约存储器，而拥有廉价存储器的新系统用2048字节的簇来提高性能。在本书中我们假定一簇的大小是 2048字节。

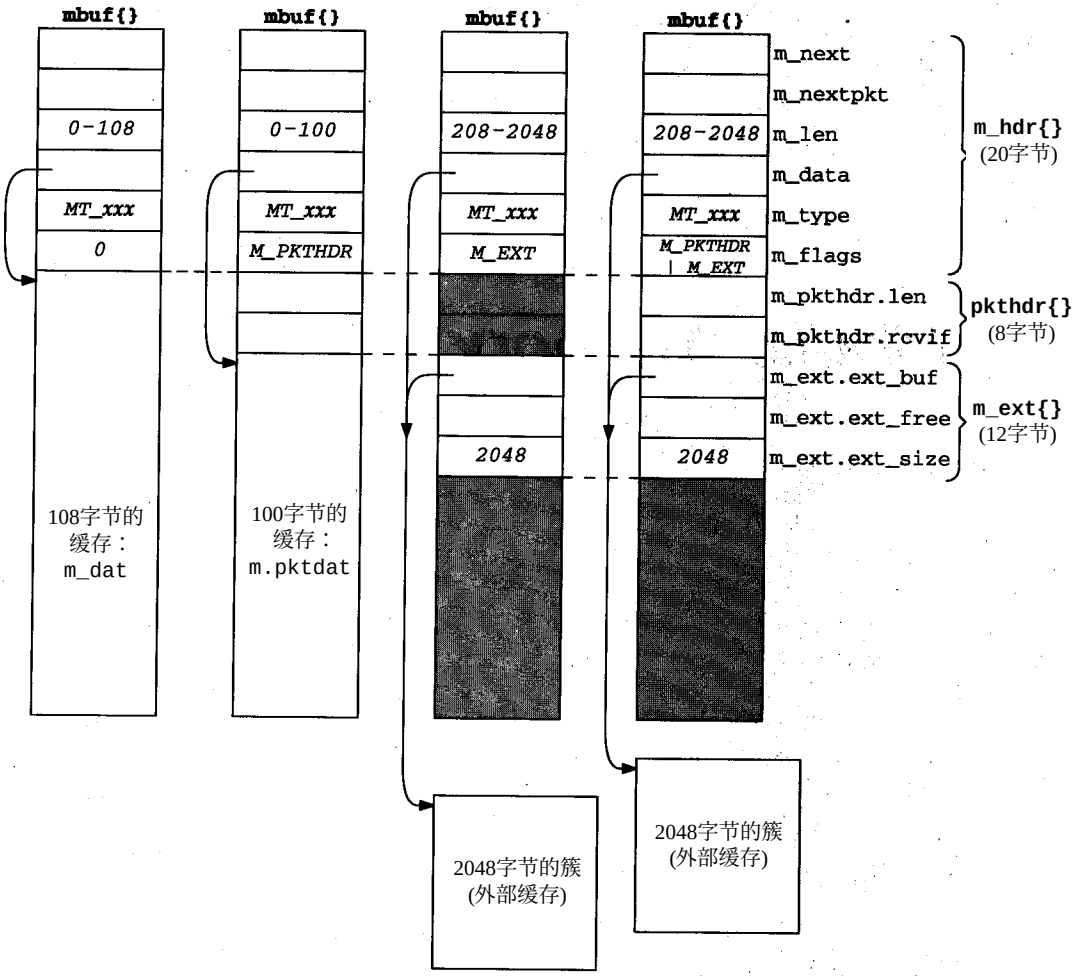


图2-1 根据不同m_flags 值的四种不同类型的mbuf

不幸的是，我们所说的“簇(cluster)”用过不同的名字。常量MCLBYTES是这些缓存(1024或2048)的大小，操作这些缓存的宏的名字是 MCLGET、MCLALLOC和MCLFREE。这就是为什么称它们为“簇”的原因。但我们还看到 mbuf的标志是M_EXT，它代表“外部的”缓存。最后，[Leffler et al. 1989]称它们为映射页(mapped page)。这后一种称法来源于它们的实现，在 2.9节我们会看到当要求一个副本时，这些簇是可以共享的。

我们可能会希望这种类型的 mbuf的m_len的最小值是209而不是我们在图中所示

的208。这是指，208字节数据的记录是可以存放在两个 mbuf中的，第一个mbuf存放100字节，第二个mbuf存放108字节。但在源代码中有一个差错：若超过或等于 208就分配一个簇。

4) 最后一类 mbuf包含一个分组首部，并包含超过 208字节的数据。同时设置了标志M_PKTHDR和M_EXT。

对于图2-1，我们还有另外几点需要说明：

- mbuf结构的大小总是128字节。这意味着图 2-1右边两个mbuf在结构m_ext后面的未用空间为88字节(128 - 20 - 8 - 12)。
- 既然有些协议(例如UDP)允许零长记录，当然就可以有m_len为0的数据缓存。
- 在每个mbuf中的成员m_data指向相应缓存的开始(mbuf缓存本身或一个簇)。这个指针能指向相应缓存的任意位置，不一定是起始。

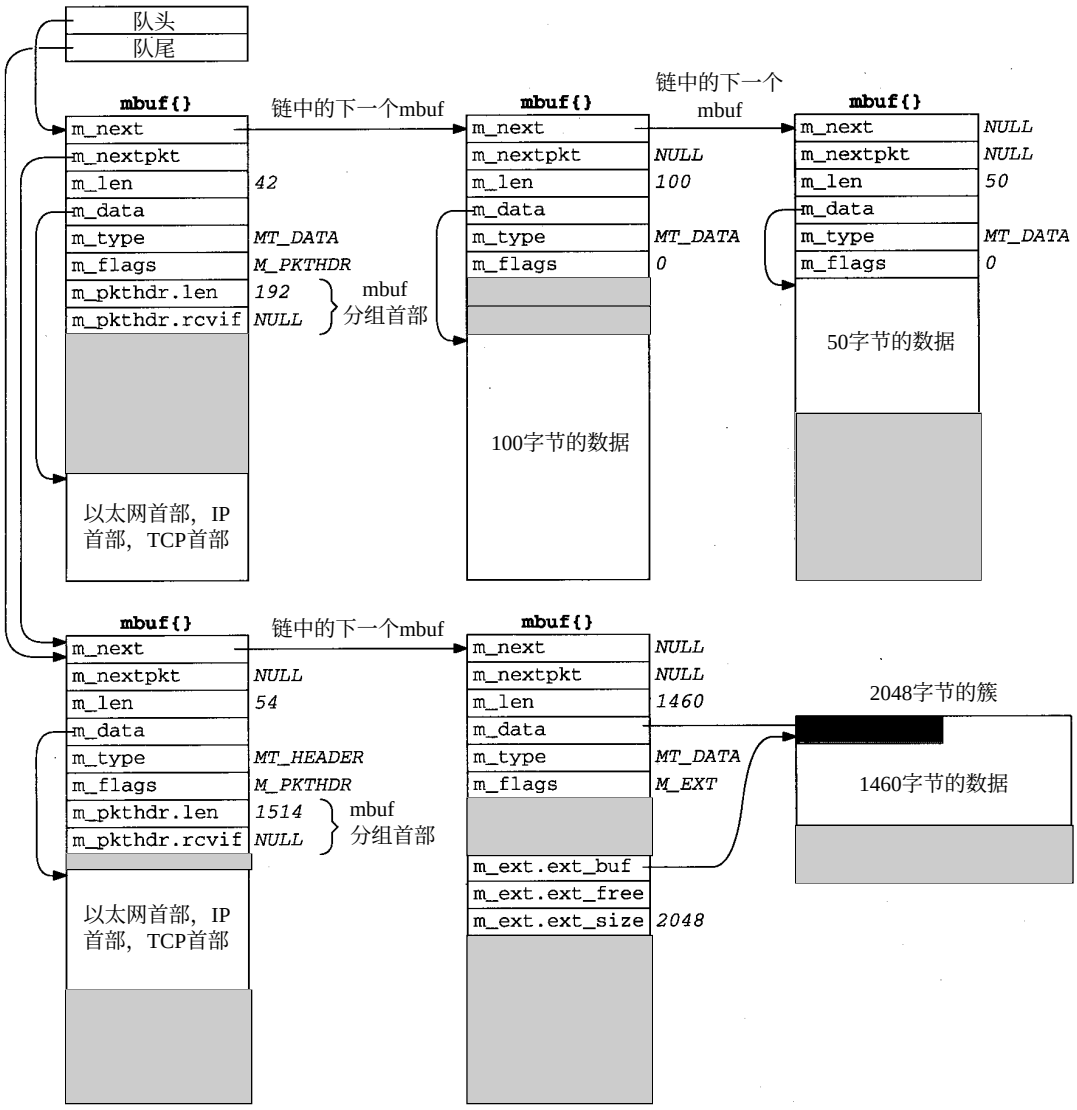


图2-2 在一个队列中的两个分组：第一个带有 192字节数据，第二个带有1514字节数据

- 带有簇的mbuf总是包含缓存的起始地址(m_ext.ext_buf)和它的大小(m_ext.ext_size)。我们在本书采用的大小为2048。成员m_data和m_ext.ext_buf值是不同的(如我们所示)，除非m_data也指向缓存的第一个字节。结构 m_ext的第三个成员 ext_free，Net/3当前未用。
- 指针m_next把mbuf链接在一起，把一个分组(记录)形成一条mbuf链表，如图1-8所示。
- 指针m_nextpkt把多个分组(记录)链接成一个mbuf链表队列。在队列中的每个分组可以是一个单独的mbuf，也可以是一个mbuf链表。每个分组的第一个mbuf包含一个分组首部。如果多个mbuf定义一个分组，只有第一个mbuf的成员m_nextpkt被使用——链表中其他mbuf的成员m_nextpkt全是空指针。

图2-2所示的是在一个队列中的两个分组的例子。它是图 1-8的一个修改版。我们已经把UDP数据报放到接口输出队列中(显示出14字节的以太网首部已经添加到链表中第一个 mbuf的IP首部前面)，并且第二个分组已经被添加到队列中： TCP段包含1460字节的用户数据。 TCP数据包含在一个簇中，并且有一个 mbuf包含了它的以太网、 IP与TCP首部。通过这个簇，我们可以看到指向簇的数据指针 (m_data)不需要指向簇的起始位置。我们所示的队列有一个头指针和一个尾指针。这就是Net/3处理接口输出队列的方法。我们给有 M_EXT标志的mbuf还添加了一个m_ext结构，并且用阴影表示这个mbuf中未用的pkthdr结构。

带有UDP数据报分组首部的第一个 mbuf的类型是MT_DATA，但带有TCP报文段分组首部的第一个mbuf的类型是MT_HEADER。这是由于UDP和TCP采用了不同的方式往数据中添加首部造成的，但没有什么不同。这两种类型的 mbuf本质上一样。链表中第一个mbuf的m_flags的值M_PKTHDR指示了它是一个分组首部。

仔细的读者可能会注意到我们显示一个 mbuf的图(Net/3 mbuf，图2-1)与显示一个Net/1 mbuf的图[Leffler et al. 1989，p.290]的区别。这个变化是在Net/2中造成的：添加了成员 m_flags，把指针 m_act改名为 m_nextpkt，并把这个指针移到这个mbuf的前面。

在第一个mbuf中，UDP与TCP协议首部位置的不同是由于 UDP调用M_PREPEND(图23-15和习题23.1)而TCP调用MGETHDR(图26-25)造成的。

2.2 代码介绍

mbuf函数在一个单独的C文件中，并且mbuf宏与各种mbuf定义都在一个单独的头文件中，如图2-3所示。

文　件	说　明
sys/mbuf.h	mbuf结构、mbuf宏与定义
kern/uipc_mbuf.c	mbuf函数

图2-3 本章讨论的文件

2.2.1 全局变量

在本章中有一个全局变量要介绍，如图 2-4所示。

变 量	数 据 类 型	说 明
mbstat	struct mbstat	mbuf的统计信息(图2-5)

图2-4 本章介绍的全局变量

2.2.2 统计

在全局结构mbstat中维护的各种统计，如图2-5所示。

mbstat成员	说 明
m_clfree	自由簇
m_clusters	从页池中获得的簇
m_drain	调用协议的drain函数来回收空间的次数
m_drops	寻找空间(未用)失败的次数
m_mbufs	从页池(未用)中获得的mbuf数
m_mtypes[256]	当前mbuf的分配数：MT_*xxx*索引
m_spare	剩余空间(未用)
m_wait	等待空间(未用)的次数

图2-5 在结构mbstat 中维护的mbuf统计

这个结构能被命令netstat -m检测；图2-6所示的是一些输出示例。关于所用映射页的数量的两个值是：m_clusters(34)减m_clfree (32)——当前使用的簇数(2)和m_clusters(34)。

分配给网络的存储器的千字节数是 mbuf存储器(99×128字节)加上簇存储器(34×2048字节)再除以1024。使用百分比是mbuf存储器(99×128字节)加上所用簇的存储器(2×2048字节)除以网络存储器总数(80千字节)，再乘100。

netstat -m output	mbstat member
99 mbufs in use:	
1 mbufs allocated to data	m_mtypes[MT_DATA]
43 mbufs allocated to packet headers	m_mtypes[MT_HEADER]
17 mbufs allocated to protocol control blocks	m_mtypes[MT_PCB]
20 mbufs allocated to socket names and addresses	m_mtypes[MT_SONAME]
18 mbufs allocated to socket options	m_mtypes[MT_SOOPTS]
2/34 mapped pages in use	(see text)
80 Kbytes allocated to network (20% in use)	(see text)
0 requests for memory denied	m_drops
0 requests for memory delayed	m_wait
0 calls to protocol drain routines	m_drain

图2-6 mbuf统计例子

2.2.3 内核统计

mbuf统计显示了在Net/3源代码中的一种通用技术。内核在一个全局变量 (在本例中是结构mbstat)中保持对某些统计信息的跟踪。当内核在运行时，一个进程 (在本例中是netstat程序)可以检查这些统计。

不是提供系统调用来获取由内核维护的统计，而是进程通过读取链接编辑器在内核建立时保存的信息来获得所关心的数据结构在内核中的地址。然后进程调用函数 kvm(3)，通过使用特殊文件/dev/mem读取在内核存储器中的相应位置。如果内核数据结构从一个版本改变

为下一版本，任何读取这个结构的程序也必须改变。

2.3 mbuf的定义

处理mbuf时，我们会反复遇到几个常量。它们的值显示在图 2-7中。除了MCLBYTES定义在文件/usr/include/machine/param.h中外，其他所有常量都定义在文件mbuf.h中。

常　量	值(字节数)	说　明
MCLBYTES	2048	一个mbuf簇(外部缓存)的大小
MHLEN	100	带分组首部的mbuf的最大数据量
MINCLSIZE	208	存储到簇中的最小数据量
MLEN	108	在正常mbuf中的最大数据量
MSIZE	128	每个mbuf的大小

图2-7 mbuf.h 中的mbuf常量

2.4 mbuf结构

图2-8所示的是mbuf结构的定义。

mbuf.h

```
60 /* header at beginning of each mbuf: */
61 struct m_hdr {
62     struct mbuf *mh_next;        /* next buffer in chain */
63     struct mbuf *mh_nextpkt;     /* next chain in queue/record */
64     int      mh_len;             /* amount of data in this mbuf */
65     caddr_t mh_data;             /* pointer to data */
66     short    mh_type;            /* type of data (Figure 2.10) */
67     short    mh_flags;           /* flags (Figure 2.9) */
68 };

69 /* record/packet header in first mbuf of chain; valid if M_PKTHDR set */
70 struct pkthdr {
71     int      len;                /* total packet length */
72     struct ifnet *rcvif;         /* receive interface */
73 };

74 /* description of external storage mapped into mbuf, valid if M_EXT set */
75 struct m_ext {
76     caddr_t ext_buf;             /* start of buffer */
77     void    (*ext_free) ();      /* free routine if not the usual */
78     u_int   ext_size;            /* size of buffer, for ext_free */
79 };

80 struct mbuf {
81     struct m_hdr m_hdr;
82     union {
83         struct {
84             struct pkthdr MH_pkthdr;     /* M_PKTHDR set */
85             union {
86                 struct m_ext MH_ext;     /* M_EXT set */
87                 char     MH_databuf[MHLEN];
88             } MH_dat;
89         } MH;
90         char     M_databuf[MLEN];     /* !M_PKTHDR, !M_EXT */
91     } M_dat;
92 };
```

图2-8 mbuf 结构

```
 93 #define m_next       m_hdr.mh_next
 94 #define m_len        m_hdr.mh_len
 95 #define m_data       m_hdr.mh_data
 96 #define m_type       m_hdr.mh_type
 97 #define m_flags      m_hdr.mh_flags
 98 #define m_nextpkt    m_hdr.mh_nextpkt
 99 #define m_act        m_nextpkt
100 #define m_pkthdr     M_dat.MH.MH_pkthdr
101 #define m_ext        M_dat.MH.MH_dat.MH_ext
102 #define m_pktdat     M_dat.MH.MH_dat.MH_databuf
103 #define m_dat        M_dat.M_databuf
```

mbuf.h

图2-8 (续)

结构mbuf是用一个m_hdr结构跟着一个联合来定义的。如注释所示，联合的内容依赖于标志M_PKTHDR和M_EXT。

93-103 这11个#define语句简化了对mbuf结构中的结构与联合的成员的访问。我们会看到这种技术普遍应用于Net/3源代码中，只要是一个结构包含其他结构或联合这种情况。

我们在前面说明了在结构 mbuf中前两个成员的目的：指针 m_next把mbuf链接成一个mbuf链表，而指针m_nextpkt把mbuf链表链接成一个mbuf队列。

图1-8显示了每个mbuf的成员m_len与分组首部中的成员 m_pkthdr.len的区别。后者是链表中所有mbuf的成员m_len的和。

图2-9所示的是成员m_flags的五个独立的值。

m_flags	说 明
M_BCAST	作为链路层广播发送/接收
M_EOR	记录结束
M_EXT	此mubf带有簇(外部缓存)
M_MCAST	作为链路层多播发送/接收
M_PKTHDR	形成一个分组(记录)的第一个mbuf
M_COPYFLAGS	*M_PKTHDR/M_EOR/M_BCAST/M_MCAST*

图2-9 m_flags 值

我们已经说明了标志 M_EXT和M_PKTHDR。M_EOR在一个包含记录尾的 mbuf中设置。Internet协议(例如TCP)从来不设置这个标志，因为TCP提供一个无记录边界的字节流服务。但是OSI与XNS运输层要用这个标志。在插口层我们会遇到这个标志，因为这一层是协议无关的，并且它要处理来自或发往所有运输层的数据。

当要往一个链路层广播地址或多播地址发送分组，或者要从一个链路层广播地址或多播地址接收一个分组时，在这个 mubf中要设置接下来的两个标志 M_BCAST和M_MCAST。这两个常量是协议层与接口层之间的标志(图1-3)。

对于最后一个标志值 M_COPYFLAGS，当一个mbuf包含一个分组首部的副本时，这个标志表明这些标志是复制的。

图2-10所示的常量MT_*xxx*用于成员m_type，指示存储在mbuf中的数据的类型。虽然我们总认为一个mbuf是用来存放要发送或接收的用户数据，但 mbuf可以存储各种不同的数据结构。回忆图1-6中的一个mbuf被用来存放一个插口地址结构，其中的目标地址用于系统调用sendto。它的m_type成员被设置为MT_SONAME。

不是图2-10中所有的mbuf类型值都用于Net/3。有些已不再使用(MT_HTABLE)，还有一些不用于TCP/IP代码中，但用于内核的其他地方。例如， MT_OOBDATA用于OSI和XNS协议，但是TCP用不同方法来处理带外 (out-of-band)数据(我们在29.7节说明)。当我们在本书的后面遇到其他mbuf类型时会说明它们的用法。

Mbuf m_type	用于Net/3 TCP/IP代码	说　明	存储类型
MT_CONTROL	•	外部数据协议报文	M_MBUF
MT_DATA	•	动态数据分配	M_MBUF
MT_FREE		应在自由列表中	M_FREE
MT_FTABLE	•	分片重组首部	M_FTABLE
MT_HEADER	•	分组首部	M_MBUF
MT_HTABLE		IMP主机表	M_HTABLE
MT_IFADDR		接口地址	M_IFADDR
MT_OOBDATA		加速(带外)数据	M_MBUF
MT_PCB		协议控制块	M_PCB
MT_RIGHTS		访问权限	M_MBUF
MT_RTABLE		路由表	M_RTABLE
MT_SONAME	•	插口名称	M_MBUF
MT_SOOPTS	•	插口选项	M_SOOPTS
MT_SOCKET		插口结构	M_SOCKET

图2-10 成员m_type 的值

本图的最后一列所示的M_*xxx*值与内核为不同类型mbuf分配的存储器片有关。这里有大约60个可能的M_*xxx*值指派给由内核函数malloc和宏MALLOC分配的不同类型的存储器空间。图2-6所示的是来源于命令netstat -m的mbuf分配统计信息，它包括每种MT_*xxx*类型的统计。命令vmstat -m显示了内核的存储分配统计，包括每个M_*xxx*类型的统计。

由于mbuf有一个固定长度(128字节)，因此对于mbuf的使用有一个限制 —— 包含的数据不能超过108字节。Net/3用一个mbuf来存储一个TCP协议控制块(在第24章我们会涉及到)，这个mbuf的类型为MT_PCB。但是4.4BSD把这个结构的大小从108字节增加到140字节，并为这个结构使用一种不同的内核存储器分配类型。

仔细的读者会注意到图2-10中我们表明未使用MT_PCB类型的mbuf，而图2-6显示这个类型的计数不为零。Unix域协议使用这种类型的mbuf，并且mbuf的统计功能用于所有协议，而不只是Internet协议，记住这一点很重要。

2.5 简单的mbuf宏和函数

有超过两打的宏和函数来处理mbuf(分配一个mbuf，释放一个mbuf，等等)。让我们来查看几个宏与函数的源代码，看看它们是如何实现的。

有些操作既提供了宏也提供了函数。宏版本的名称是以M开头的大写字母名称，而函数是以m_开始的小写字母名称。两者的区别是一种典型的时间-空间互换。宏版本在每个被用到的地方都被C预处理器展开(要求更多的代码空间)，但是它在执行时更快，因为它不需要执行函数调用(对于有些体系结构，这是费时的)。而对于函数版本，它在每个被调用的地方变成了一些指令(参数压栈，调用函数等)，要求较少的代码空间，但会花费更多的执行时间。

2.5.1 m_get函数

让我们先看一下图2-11中分配mbuf的函数：m_get。这个函数仅仅就是宏MGET的展开。

uipc_mbuf.c

```
134 struct mbuf *
135 m_get(nowait, type)
136 int     nowait, type;
137 {
138     struct mbuf *m;

139     MGET(m, nowait, type);
140     return (m);
141 }
```

uipc_mbuf.c

图2-11 m_get 函数：分配一个mbuf

注意，Net/3代码不使用ANSI C参数声明。但是，如果使用一个ANSI C编译器，所有Net/3系统头文件为所有的内核函数都提供了ANSI C函数原型。例如，<sys/mbuf.h>头文件中包含这样的行：

```
struct mbuf *m_get(int, int);
```

这些函数原型为所有内核函数的调用提供编译期间的参数与返回值的检查。

这个调用表明参数nowait的值为M_WAIT或M_DONTWAIT，它取决于在存储器不可用时是否要求等待。例如，当插口层请求分配一个 mbuf来存储sendto系统调用(图1-6)的目标地址时，它指定M_WAIT，因为在此阻塞是没有问题的。但是当以太网设备驱动程序请求分配一个mbuf来存储一个接收的帧时 (图1-10)，它指定M_DONTWAIT，因为它是作为一个设备中断处理来执行的，不能进入睡眠状态来等待一个 mbuf。在这种情况下，若存储器不可用，设备驱动程序丢弃这个帧比较好。

2.5.2 MGET宏

图2-12所示的是MGET宏。调用MGET来分配存储sendto系统调用(图1-6)的目标地址的mbuf如下所示：

```
MGET(m, M_WAIT, MT_SONAME);
if (m == NULL)
   return(ENOBUFS);
```

mbuf.h

```
154 #define MGET(m, how, type) { \
155     MALLOC((m), struct mbuf *, MSIZE, mbtypes[type], (how)); \
156     if (m) { \
157         (m)->m_type = (type); \
158         MBUFLOCK(mbstat.m_mtypes[type]++;) \
159         (m)->m_next = (struct mbuf *)NULL; \
160         (m)->m_nextpkt = (struct mbuf *)NULL; \
161         (m)->m_data = (m)->m_dat; \
162         (m)->m_flags = 0; \
163     } else \
164         (m) = m_retry((how), (type)); \
165 }
```

mbuf.h

图2-12 MGET 宏

虽然调用指定了M_WAIT，但返回值仍然要检查，因为，如图2-13所示，等待一个mbuf并不保证它是可用的。

154-157 MGET一开始调用内核宏MALLOC，它是通用内核存储器分配器进行的。数组mbtypes把mbuf的MT_*xxx*值转换成相应的M_*xxx*值(图2-10)。若存储器被分配，成员m_type被设置为参数中的值。

158 用于跟踪统计每种mbuf类型的内核结构加1(mbstat)。当执行这句时，宏MBUFLOCK把它作为参数来改变处理器优先级(图1-13)，然后把优先级恢复为原值。这防止在执行语句mbstat.m_mtypes[type]++；时被网络设备中断，因为mbuf可能在内核中的各层中被分配。考虑这样一个系统，它用三步来实现一个C中的++运算：(1)把当前值装入一个寄存器；(2)寄存器加1；(3)把寄存器值存入存储器。假设计数器值为77并且MGET在插口层执行。假设执行了步骤1和2(寄存器值为78)，并且一个设备中断发生。若设备驱动也执行MGET来获得同种类型的mbuf，在存储器中取值(77)，加1(78)，并存回在存储器。当被中断执行的MGET的步骤3继续执行时，它将寄存器的值(78)存入存储器。但是计数器应为79，而不是78，这样计数器就被破坏了。

159-160 两个mbuf指针，m_next和m_nextpkt，被设置为空指针。若必要，由调用者把这个mbuf加入到一个链或队列。

161-162 最后，数据指针被设置为指向108字节的mbuf缓存的起始，而标志被设置为0。

163-164 若内核的存储器分配调用失败，调用m_retry(图2-13)。第一个参数是M_WAIT或M_DONTWAIT。

2.5.3 m_retry函数

图2-13所示的是m_retry函数。

uipc_mbuf.c

```
 92 struct mbuf *
 93 m_retry(i, t)
 94 int     i, t;
 95 {
 96     struct mbuf *m;

 97     m_reclaim();
 98 #define m_retry(i, t)    (struct mbuf *)0
 99     MGET(m, i, t);
100 #undef m_retry
101     return (m);
102 }
```

uipc_mbuf.c

图2-13 m_retry 函数

92-97 被m_retry调用的第一个函数是m_reclaim。在7.4节我们会看到每个协议都能定义一个“drain”函数，在系统缺乏可用存储器时能被m_reclaim调用。在图10-32中我们还会发现当IP的drain函数被调用时，所有等待重新组成IP数据报的IP分片被丢弃。TCP的drain函数什么都不做，而UDP甚至就没有定义一个drain函数。

98-102 因为在调用了m_reclaim后有可能有机会得到更多的存储器，因此再次调用宏MGET，试图获得mbuf。在展开宏MGET(图2-12)之前，m_retry被定义为一个空指针。这可以防止当存储器仍然不可用时的无休止的循环：这个MGET展开会把m设置为空指针而不是调用m_retry函数。在MGET展开以后，这个m_retry的临时定义就被取消了，以防在此之后

有对MGET的其他引用。

2.5.4 mbuf锁

在本节中我们所讨论的函数和宏并不调用 spl函数，而是调用图2-12中的MBUFLOCK来保护这些函数和宏不被中断。但在宏 MALLOC的开始包含一个 splimp，在结束时有一个 splx。宏MFREE中包含同样的保护机制。由于 mbuf在内核的所有层中被分配和释放，因此内核必须保护那些用于存储器分配的数据结构。

另外，用于分配和释放 mbuf簇的宏MCLALLOC与MCLFREE要用一个splimp和一个splx包括起来，因为它们修改的是一个可用簇链。

因为存储器分配与释放及簇分配与释放的宏被保护起来防止被中断，我们通常在 MGET和m_get这样的函数和宏的前后不再调用 spl函数。

2.6 m_devget和m_pullup函数

我们在讨论IP、ICMP、IGMP、UDP和TCP的代码时会遇到函数m_pullup。它用来保证指定数目的字节 (相应协议首部的大小)在链表的第一个 mbuf中紧挨着存放；即这些指定数目的字节被复制到一个新的 mbuf并紧接着存放。为了理解 m_pullup的用法，必须查看它的实现及相关的函数 m_devget和宏mtod与dtom。在分析这些问题的同时我们还可以再次领会Net/3中mbuf的用法。

2.6.1 m_devget函数

当接收到一个以太网帧时，设备驱动程序调用函数 m_devget来创建一个mbuf链表，并把设备中的帧复制到这个链表中。根据所接收的帧的长度 (不包括以太网首部)，可能导致4种不同的mbuf链表。图2-14所示的是前两种。

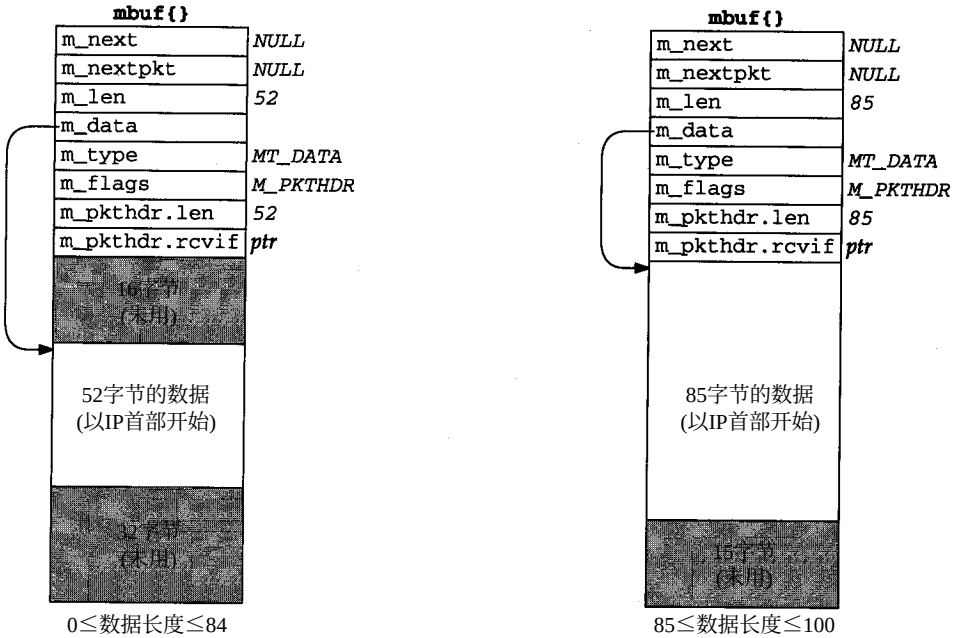


图2-14 m_devget 创建的前两种类型的mbuf

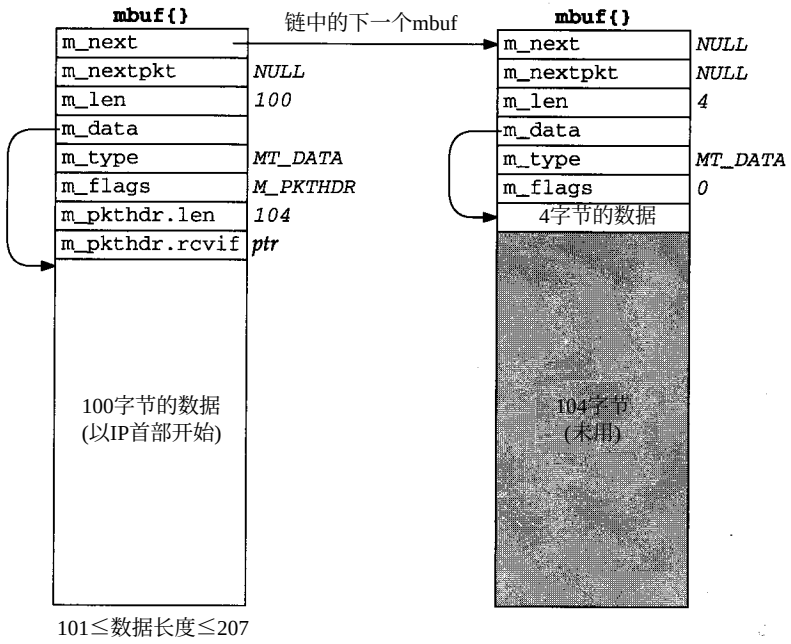


图2-15 m_devget 创建的第3种mbuf

1) 图2-14左边的mbuf用于数据的长度在0~84字节之间的情况。在这个图中，我们假定有 52 字节的数据：一个20字节的IP首部和一个32字节的TCP首部(标准的20字节的TCP首部加上12字节的TCP选项)，但不包括TCP数据。既然m_devget返回的mbuf数据从IP首部开始，m_len的实际最小值是28：20字节的IP首部，8字节的UDP首部和一个0长度的UDP数据报。m_devget在这个mbuf的开始保留了16字节未用。虽然14字节的以太网首部不存放在这里，还是分配了一个 14字节的用于输出的以太网首部，这是同一个 mbuf，用于输出。我们会遇到两个函数： icmp_reflect和tcp_respond，它们通过把接收到的 mbuf 作为输出mbuf来产生一个应答。在这两种情况中，接收的数据报应该少于 84 字节，因此很容易在前面保留 16 字节的空间，这样在建立输出数据报时可以节省时间。分配 16 字节而不是14字节的原因是为了在 mbuf中用长字对准方式存储IP首部。

2) 如果数据在85~100字节之间，就仍然存放在一个分组首部mbuf中，但在开始没有16字节

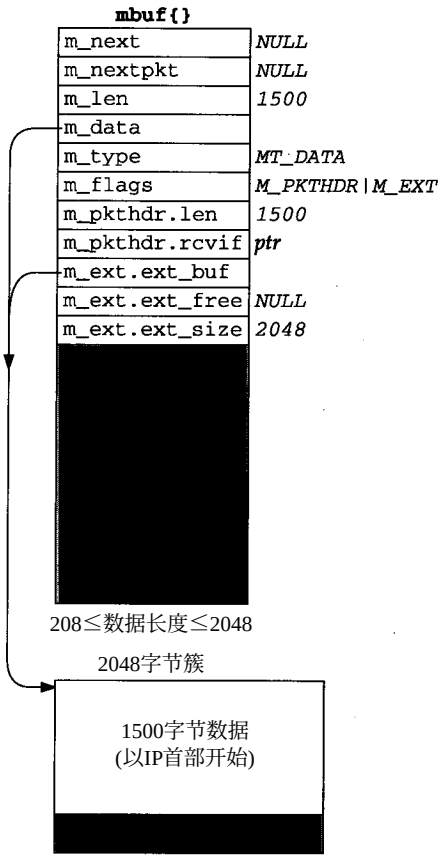


图2-16 m_devget 创建的第4种mbuf

的空间。数据存储在数组m_pktdat的开始，并且任何未用的空间放在这个数组的后面。例如在图2-14的右边的mbuf显示的就是这个例子，假设有85字节数据。

3) 图2-15所示的是m_devget创建的第3种mbuf。当数据在101~207字节之间时，要求有两个mbuf。前100字节存放在第一个mbuf中(有分组首部的mbuf)，而剩下的存放在第二个mbuf中。在此例中，我们显示的是一个104字节的数据报。在第一个mbuf的开始没有保留16字节的空间。

4) 图2-16所示的是m_devget创建的第四种mbuf。如果数据超过或等于208字节(MINCLBYTES)，要用一个或多个簇。图中的例子假设了一个1500字节的以太网帧。如果使用1024字节的簇，本例子需要两个mbuf，每个mbuf都有标志M_EXT，和指向一个簇的指针。

2.6.2 mtod和dtom宏

宏mtod和dtom也定义在文件mbuf.h中。它们简化了复杂的mbuf结构表达式。

```
#define  mtod(m,t)   ((t)((m)->m_data))
#define  dtom(x)     ((struct mbuf *)((int)(x) & ~(MSIZE-1)))
```

mtod(“mbuf到数据”)返回一个指向mbuf数据的指针，并把指针声名为指定类型。例如代码

```
struct mbuf *m;
struct ip *ip;

ip = mtod(m, struct ip *);
ip->ip_v = IPVERSION;
```

存储在mbuf的数据(m_data)指针ip中。C编译器要求进行类型转换，然后代码用指针ip引用IP首部。我们可以看到当一个C结构(通常是一个协议首部)存储在一个mbuf中时会用到这个宏。当数据存放在mbuf本身(图2-14和图2-15)或存放在一个簇中(图2-16)时，可以用这个宏。

宏dtom (“数据到mbuf”)取得一个存放在一个mbuf中任意位置的数据的指针，并返回这个mbuf结构本身的一个指针。例如，若我们知道ip指向一个mbuf的数据区，下面的语句序列

```
struct mbuf *m;
struct ip *ip;

m = dtom(ip);
```

把指向这个mbuf开始的指针存放到m中。我们知道MSIZE(128)是2的幂，并且内核存储器分配器总是为mbuf分配连续的MSIZE字节的存储块，dtom仅仅是清除参数中指针的低位来发现这个mbuf的起始位置。

宏dtom有一个问题：当它的参数指向一个簇，或在一个簇内，如图2-16时，它不能正确执行。因为那里没有指针从簇内指回mbuf结构，dtom不能被使用。这导致了下一个函数：m_pullup。

2.6.3 m_pullup函数和连续的协议首部

函数m_pullup有两个目的。第一个是当一个协议(IP、ICMP、IGMP、UDP或TCP)发现在第一个mbuf的数据量(m_len)小于协议首部的最小长度(例如：IP是20，UDP是8，TCP是20)时。调用m_pullup是基于假定协议首部的剩余部分存放在链表中的下一个mbuf。

m_pullup重新安排mbuf链表，使得前N字节的数据被连续地存放在链表的第一个 mbuf中。N是这个函数的一个参数，它必须小于或等于 100(MHLEN)。如果前N字节连续存放在第一个mbuf中，则可以使用宏mtod和dtom。

例如，我们在IP输入例程中会遇到下面这样的代码：

```
if (m->m_len < sizeof(struct ip) &&
    (m = m_pullup(m, sizeof(struct ip))) == 0) {
        ipstat.ips_toosmall++;
        goto next;
}
ip = mtod(m, struct ip *);
```

如果第一个mbuf中的数据少于20(标准IP首部的大小)，m_pullup被调用。函数m_pullup有两个原因会失败：(1)如果它需要其他mbuf并且调用MGET失败；或者(2)如果整个mbuf链表中的数据总数少于要求的连续字节数 (即我们所说的N，在本例中是20)。通常，失败是因为第二个原因。在此例中，如果m_pullup失败，一个IP计数器加1，并且此IP数据报被丢弃。注意，这段代码假设失败的原因是mbuf 链表中数据少于20字节。

实际上，在这种情况下，m_pullup很少能被调用(注意，C语言的&&操作符仅当mbuf长度小于期待值时才调用它)，并且当它被调用时，它通常会失败。通过查看图 2-14~图2-16，我们可以找到它的原因：在第一个mbuf中，或在簇中，从IP首部开始有至少100字节的连续字节。这允许60字节的最大IP首部，并且后面跟着40字节的TCP首部(其他协议——ICMP，IGMP和UDP——它们的协议首部不到40字节)。如果mbuf链表中的数据可用(分组不小于协议要求的最小值)，则所要求的字节数总能连续地存放在第一个 mbuf中。但是，如果接收的分组太小(m_len小于期待的最小值)，则m_pullup被调用，并且它返回一个差错，因为在 mbuf链表中没有所要求数目的可用数据。

> 源于伯克利的内核维护一个叫MPFail的变量，每次m_pullup失败时，它都加1。在一个Net/3系统中曾经接收了超过2700万的IP数据报，而MPFail只有9。计数器ipstat.ips_toosmall也是9，并且所有其他协议计数器(ICMP、IGMP、UDP和TCP等)所计的m_pullup失败次数为0。这证实了我们的断言：大多数 m_pullup的失败是因为接收的IP数据报太小。

2.6.4 m_pullup和IP的分片与重组

使用m_pullup的第二个用途涉及到IP和TCP的重组。假定IP接收到一个长度为296的分组，这个分组是一个大的IP数据报的一个分片。这个从设备驱动程序传到 IP输入的mbuf看起来像我们在图2-16中所示的一个mbuf：296字节的数据存放在一个簇中。我们将这显示在图 2-17中。

问题在于，IP的分片算法将各分片都存放在一个双向链表中，使用 IP首部中的源与目标IP地址来存放向前与向后链表指针 (当然，这两个IP地址要保存在这个链表的表头中，因为它们还要放回到重组的数据报中。我们在第 10章讨论这个问题)。但是如果这个IP首部在一个簇中，如图2-17所示，这些链表指针会存放在这个簇中，并且当以后遍历链表时，指向 IP首部的指针(即指向这个簇的起始的指针)不能被转换成指向mbuf的指针。这是我们在本节前面提到的问题：如果m_data指向一个簇时不能使用宏dtom，因为没有从簇指回mbuf的指针。IP分片

不能如图2-17所示的把链指针存储在簇中。

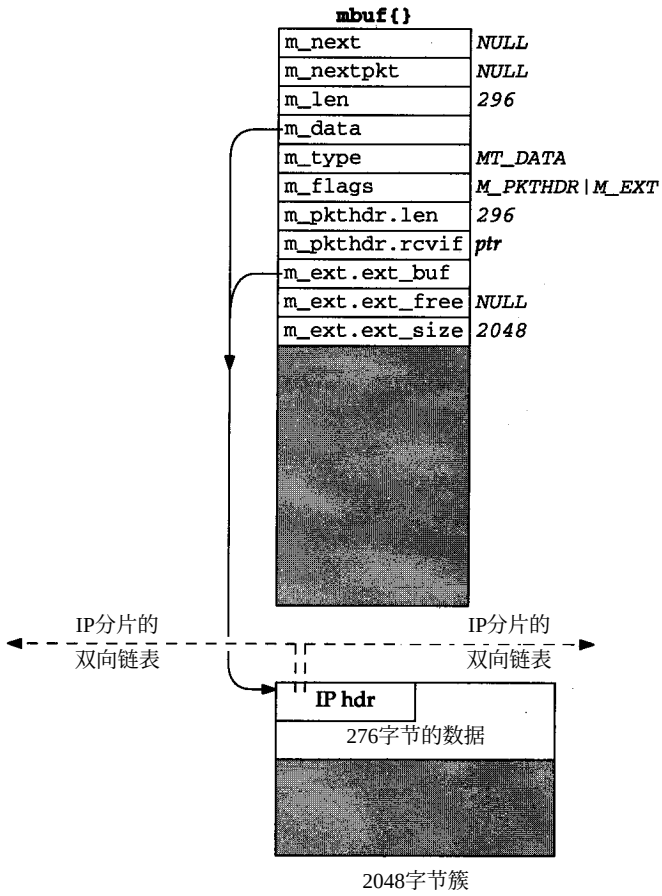


图2-17 一个长度为296的IP分片

为解决这个问题，当接收到一个分片时，若分片存放在一个簇中， IP分片例程总是调用m_pullup。它强行将20字节的IP首部放到它自己的mbuf中。代码如下：

```
if (m->m_flags & M_EXT) {
    if ((m = m_pullup(m, sizeof(struct ip))) == 0) {
        ipstat.ips_toosmall++;
        goto next;
    }
    ip = mtod(m, struct ip *);
}
```

图2-18所示的是在调用了m_pullup后得到的mbuf链表。m_pullup分配了一个新的mubf，挂在链表的前面，并从簇中取走40字节放入到这个新的mbuf中。之所以取40字节而不是仅要求的20字节，是为了保证以后在IP把数据报传给一个高层协议(例如：ICMP，IGMP，UDP或TCP)时，高层协议能正确处理。采用不可思议的40(图7-17中的max_protohdr)是因为最大协议首部通常是一个20字节的IP首部和20字节的TCP首部的组合(这假设其他协议族，例如OSI协议，并不编译到内核中)。

在图2-18中，IP分片算法在左边的mbuf中保存了一个指向IP首部的指针，并且可以用dtom将这个指针转换成一个指向mbuf本身的指针。

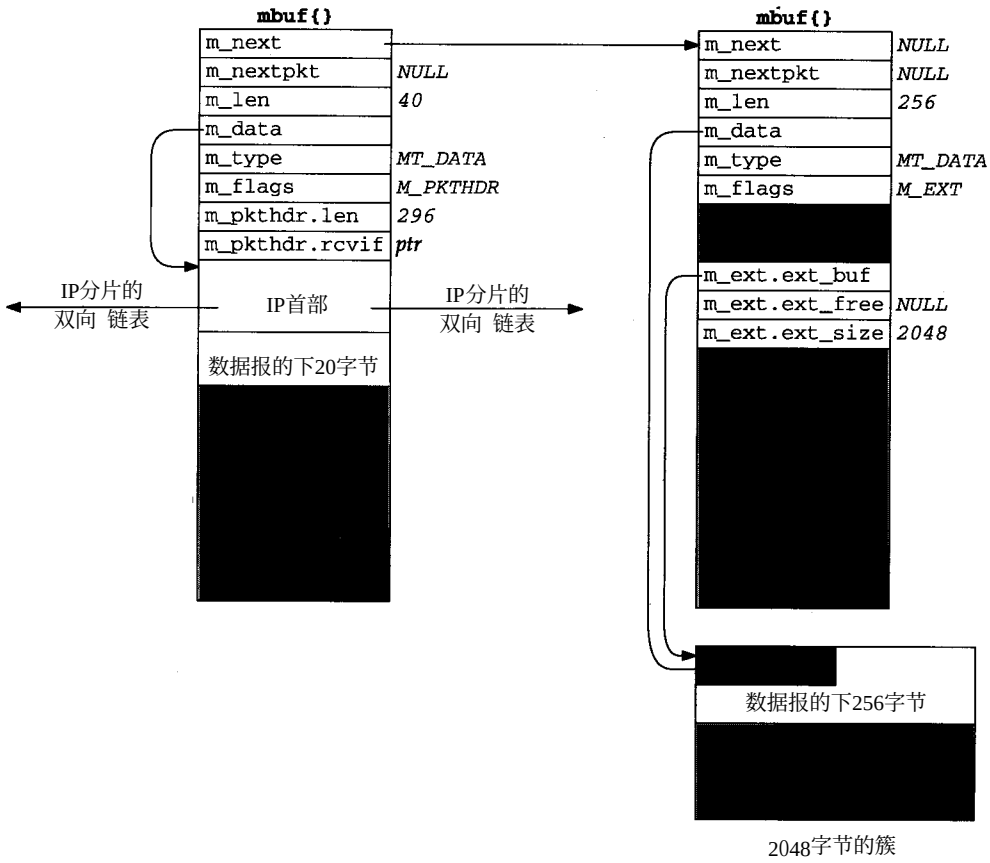


图2-18 调用m_pullup 后的长度为296的IP分片

2.6.5 TCP重组避免调用m_pullup

重组TCP报文段使用一个不同的技术而不是调用 m_pullup。这是因为m_pullup开销较大：分配存储器并且数据从一个簇复制到一个 mbuf中。TCP试图尽可能地避免数据的复制。

卷1的第19章提到大约有一半的 TCP数据是批量数据 (通常每个报文段有 512或更多字节的数据)，并且另一半是交互式数据 (这里面有大约90%的报文段包含不到 10字节的数据)。因此，当TCP从IP接收报文段时，它们通常是如图 2-14左边所示的格式 (一个小量的交互数据，存储在mbuf本身)或图2-16所示的格式 (批量数据，存储在一个簇中)。当TCP报文段失序到达时，它们被TCP存储到一个双向链表中。如IP分片一样，在IP首部的字段用于存放链表的指针，既然这些字段在TCP接收了IP数据报后不再需要，这完全可行。但当 IP首部存放在一个簇中，要将一个链表指针转换成一个相应的 mbuf指针时，会引起同样的问题 (图2-17)。

为解决这个问题，在27.9节中我们会看到TCP把mbuf指针存放在TCP首部中的一些未用的字段中，提供一个从簇指回 mbuf的指针，来避免对每个失序的报文段调用 m_pullup。如果IP首部包含在mbuf的数据区(图2-18)，则这个回指指针是无用的，因为宏 dtom对这个链表指针会正常工作。但如果IP首部包含在一个簇中，这个回指指针将被使用。当我们在讨论 27.9节的tcp_reass时，会研究实现这种技术的源代码。

2.6.6 m_pullup使用总结

我们已经讨论了关于使用m_pullup的三种情况：

- 大多数设备驱动程序不把一个IP数据报的第一部分分割到几个mbuf中。假设协议首部都紧挨着存放，则在每个协议(IP、ICMP、IGMP、UDP和TCP)中调用m_pullup的可能性很小。如果调用m_pullup，通常是因为IP数据报太小，并且m_pullup返回一个差错，这时数据报被丢弃，并且差错计数器加1。
- 对于每个接收到的IP分片，当IP数据报被存放在一个簇中时，m_pullup被调用。这意味着，几乎对于每个接收的分片都要调用m_pullup，因为大多数分片的长度大于208字节。
- 只要TCP报文段不被IP分片，接收一个TCP报文段，不论是否失序，都不需调用m_pullup。这是避免IP对TCP分片的一个原因。

2.7 mbuf宏和函数的小结

在操作mbuf的代码中，我们会遇到图2-19中所列的宏和图2-20中所列的函数。图2-19中的宏以函数原型的形式显示，而不是以#define形式来显示参数的类型。由于这些宏和函数主要用于处理mbuf数据结构并且不涉及联网问题，因此我们不查看实现它们的源代码。还有另外一些mbuf宏和函数用于Net/3源代码的其他地方，但由于我们在本书中不会遇到它们，因此没有把它们列于图中。

宏	描　述
MCLGET	获得一个簇(一个外部缓存)并将*m*指向的mbuf中的数据指针(m_data)设置为指向这个簇。如果存储器不可用，返回时不设置mbuf中的M_EXT标志 `void MCLGET(struct mbuf *m, int nowait);`
MFREE	释放一个*m*指向的mbuf。若*m*指向一个簇(设置了M_EXT)，这个簇的引用计数器减1，但这个簇并不被释放，直到它的引用计数器降为0(如2.9节所述)。返回*m*的后继(由m->m_next指向，可以为空)存放在*n*中 `void MFREE(struct mbuf *m, struct mbuf *n);`
MGETHDR	分配一个mbuf，并把它初始化为一个分组首部。这个宏与MGET(图2-12)相似，但设置了标志M_PKTHDR，并且数据指针(m_data)指向紧接分组首部后的100字节的缓存 `void MGETHDR(struct mbuf *m, int nowait, int type);`
MH_ALIGN	设置包含一个分组首部的mbuf的m_data，在这个mbuf数据区的尾部为一个长度为len字节的对象提供空间。这个数据指针也是长字对准方式的 `void MH_ALIGN(struct mbuf *m, int len);`
M_PREPEND	在*m*指向的mbuf中的数据的前面添加*len*字节的数据。如果mbuf有空间，则仅把指针(m_data)减len字节，并将长度(m_len)增加*len*字节。如果没有足够的空间，就分配一个新的mbuf，它的m_next指针被设置为*m*。一个新mbuf的指针存放在*m*中。并且新mbuf的数据指针被设置，这样*len*字节的数据放置到这个mbuf的尾部(例如，调用MH_ALIGN)。如果一个新mbuf被分配，并且原来的mbuf的分组首部标志被设置，则分组首部从老mbuf中移到新mbuf中 `void M_PREPEND(struct mbuf *m, int len, int nowait);`
dtom	将指向一个mbuf数据区中某个位置的指针*x*转换成一个指向这个mbuf的起始的指针。 `struct mbuf *dtom(void *x);`
mtod	将*m*指向的mbuf的数据区指针的类型转换成*type*类型 `type mtod(struct mbuf *m, type);`

图2-19　我们在本书中会遇到的mbuf宏

函 数	说 明
m_adj	从*m*指向的mbuf中移走*len*字节的数据。如果*len*是正数，则所操作的是紧排在这个mbuf的开始的*len*字节数据；否则是紧排在这个mbuf的尾部的*len*绝对值字节数据 `void m_adj(struct mbuf *m, int len);`
m_cat	把由*n*指向的mbuf链表链接到由*m*指向的mbuf链表的尾部。当我们讨论IP重组时(第10章)会遇到这个函数 `void m_cat(struct mbuf *m, struct mbuf *n);`
m_copy	这是m_copym的三参数版本，它隐含的第4个参数的值为M_DONTWAIT `struct mbuf * m_copy(struct mbuf *m, int offset, int len);`
m_copydata	从*m*指向的mbuf链表中复制*len*字节数据到由*cp*指向的缓存。从mbuf链表数据区起始的*offset*字节开始复制 `void m_copydata(struct mbuf *m, int offset, int len, caddr_t cp);`
m_copyback	从*cp*指向的缓存复制*len*字节的数据到由*m*指向的mbuf，数据存储在mbuf链表起始*offset*字节后。必要时，mbuf链表可以用其他mbuf来扩充 `void m_copyback(struct mbuf *m, int offset, int len, caddr_t cp);`
m_copym	创建一个新的mbuf链表，并从*m*指向的mbuf链表的开始*offset*处复制*len*字节的数据。一个新mbuf链表的指针作为此函数的返回值。如果*len*等于常量M_COPYALL，则从这个mbuf链表的*offset*开始的所有数据都将被复制。在2.9节中，我们会更详细地介绍这个函数 `struct mbuf *m_copym(struct mbuf *m, int offset, int len, int nowait);`
m_devget	创建一个带分组首部的mbuf链表，并返回指向这个链表的指针。这个分组首部的len和rcvif字段被设置为*len*和*ifp*。调用函数*copy*从设备接口(由*buf*指向)将数据复制到mbuf中。如果*copy*是一个空指针，调用函数bcopy。由于尾部协议不再被支持，*off*为0。我们在2.6节讨论了这个函数 `struct mbuf *m_devget(char *buf, int len, int off, struct ifnet *ifp, void (*copy)(const void *, void *, u_int));`
m_free	宏MFREE的函数版本 `struct mbuf *m_free(struct mbuf *m);`
m_freem	释放*m*指向的链表中的所有mbuf `void m_freem(struct mbuf *m);`
m_get	宏MGET的函数版本。我们在图2-12中显示过此函数 `struct mbuf *m_get(int nowait, int type);`
m_getclr	此函数调用宏MGET来得到一个mbuf，并把108字节的缓存清零 `struct mbuf *m_getclr(int nowait, int type);`
m_gethdr	宏MGETHDR的函数版本 `struct mbuf *m_gethdr(int nowait, int type);`
m_pullup	重新排列由*m*指向的mbuf中的数据，使得前*len*字节的数据连续地存储在链表中的第一个mbuf中。如果这个函数成功，则宏mtod能返回一个正好指向这个大小为*len*的结构。我们在2.6节讨论了这个函数 `struct mbuf *m_pullup(struct mbuf *m, int len);`

图2-20 在本书中我们要遇到的mbuf函数

所有原型的参数*nowait*是M_WAIT或M_DONTWAIT，参数*type*是图2-10中所示的MT_*xxx*中的一个。

M_PREPEND的一个例子是，从图1-7转换到图1-8的过程中，当IP和UDP首部被添加到数据的前面时要调用这个宏，因为另一个 mbuf要被分配。但当这个宏再次被调用 (从图1-8转换成图2-2)来添加以太网首部时，在那个mbuf中已有存放这个首部的空间。

M_copydata的最后一个参数的类型是caddr_t，它代表“内核地址”。这个数据类型通常定义在<sys/types.h>中，为char *。它最初在内核中使用，但被某些系统调用使用时被外露出来。例如， mmap系统调用，不论是4.4BSD或SVR4都把caddr_t作为第一个参数的类型并作为返回值类型。

2.8 Net/3联网数据结构小结

本节总结我们在Net/3联网代码中要遇到的数据结构类型。在 Net/3内核中用到其他数据结构(感兴趣的读者可以查看头文件<sys/queue.h>)，但下面这些是我们在本书中要遇到的。

1) 一个mbuf链：一个通过m_next指针链接的mbuf链表。我们已经看过几个这样的例子。

2) 只有一个头指针的 mbuf链的链表。 mbuf链通过每个链的第一个 mbuf中的m_nextpkt指针链接起来。

图2-21所示的就是这种链表。这种数据结构的例子是一个插口发送缓存和接收缓存。

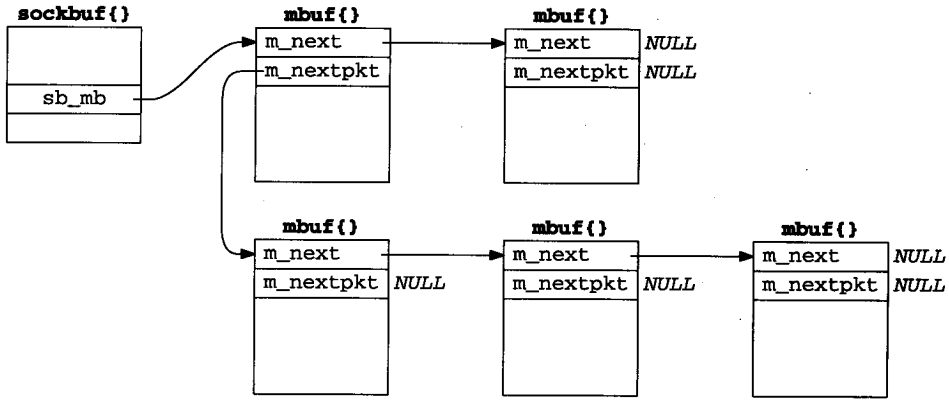


图2-21 只有头指针的mbuf链的链表

顶部的两个mbuf形成这个队列中的第一个记录，底下三个 mbuf形成这个队列的第二个记录。对于一个基于记录的协议，如 UDP，我们在每个队列中能遇到多个记录。但对于像TCP这样的协议，它没有记录的边界，每个队列我们只能发现一个记录 (一个mbuf链可能包含多个mbuf)。

把一个mbuf追加到队列的第一个记录中要遍历所有第一个记录的 mbuf，直到遇到m_next为空的mbuf。而追加一个包含新记录的 mbuf链到这个队列中，要查找所有记录直到遇到m_nextpkt为空的记录。

3) 一个有头指针和尾指针的mbuf链的链表。

图2-22显示的是这种类型的链表。我们在接口队列中会遇到它 (图3-13)，并且在图2-2中已显示过它的一个例子。

在图2-21中仅有一点改变：增加了一个尾指针，来简化增加一个新记录的操作。

4) 双向循环链表。

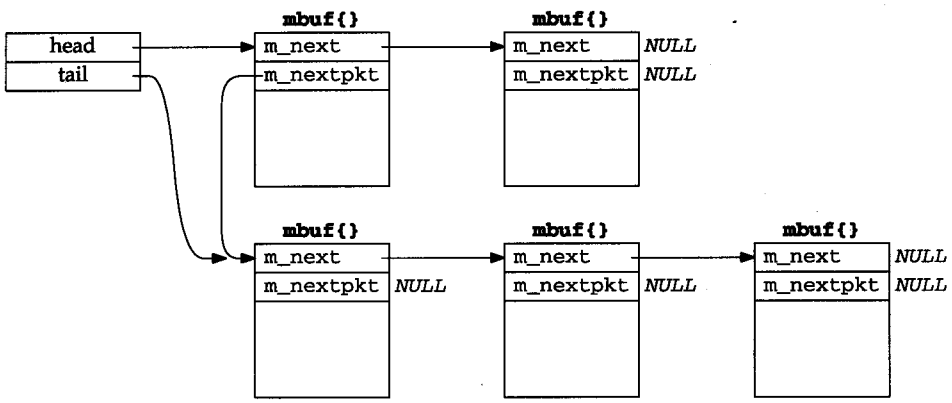


图2-22 有头指针和尾指针的链表

图2-23所示的是这种类型的链表，我们在IP分片与重装(第10章)、协议控制块(第22章)及TCP失序报文段队列(第27.9节)中会遇到这种数据结构。

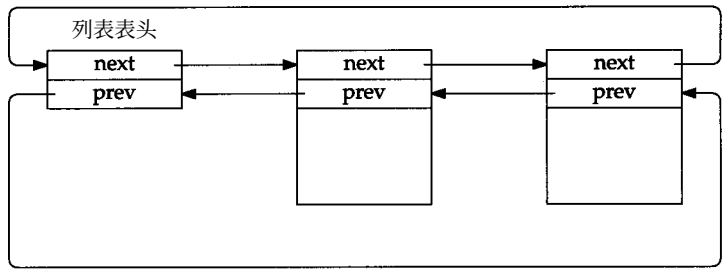


图2-23 双向循环链表

在这个链表中的元素不是mbuf——它们是一些定义了两个相邻的指针的结构：一个next指针跟着一个previous指针。两个指针必须在结构的起始处。如果链表为空，表头的next和previous指针都指向这个表头本身。

在图中我们简单地把向后指针指向另一个向后指针。显然所有的指针应包含它所指向的结构的地址，即向前指针的地址(因为向前和向后指针总是放在结构的起始处)。

这种类型的数据结构能方便地向前向后遍历，并允许方便地在链表中任何位置进行插入与删除。

函数`insque`和`remque`(图10-20)被调用来对这个链表进行插入和删除。

2.9 m_copy和簇引用计数

使用簇的一个明显的好处就是在要求包含大量数据时能减少mbuf的数目。例如，如果不使用簇，要有10个mbuf才能包含1024字节的数据：第一个mbuf带有100字节的数据，后面8个每个存放108字节数据，最后一个存放60字节数据。分配并链接10个mbuf比分配一个包含1024字节簇的mbuf开销要大。簇的一个潜在缺点是浪费空间。在我们的例子中使用一个簇(2048 + 128)要2176字节，而1280字节不到1簇(10×128)。

簇的另外一个好处是在多个mbuf间可以共享一个簇。在TCP输出和`m_copy`函数中我们遇到过这种情况，但现在我们要更详细地说明这个问题。

例如，假设应用程序执行一个write，把4096字节写到TCP插口中。假设插口发送缓存原来是空的，接收窗口至少有4096，则会发生以下操作。插口层把前2048字节的数据放在一个簇中，并且调用协议的发送例程。TCP发送例程把这个mbuf追加到它的发送缓存后，如图2-24所示，并调用tcp_output。结构socket中包含sockbuf结构，这个结构中存储着发送缓存mbuf链的链表的表头：so_snd.sb_mb。

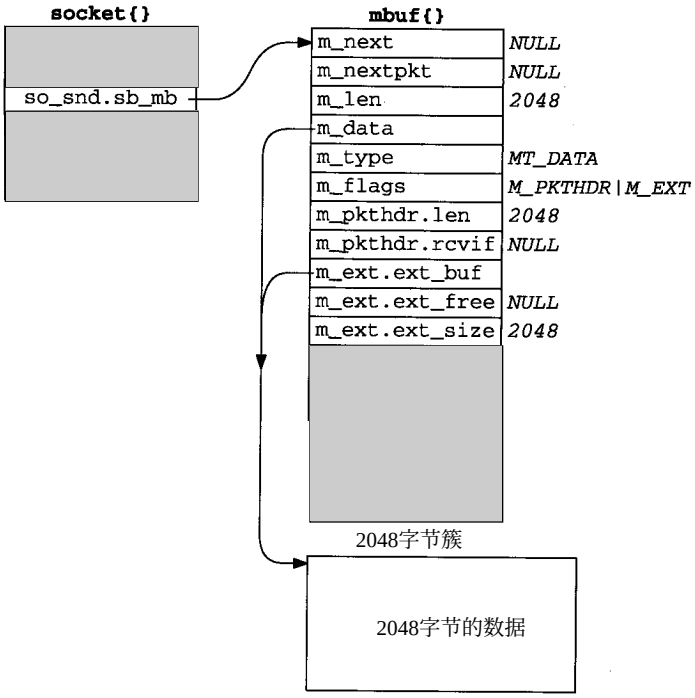


图2-24 包含2048字节数据的TCP插口发送缓存

假设这个连接(典型的是以太网)的一个TCP最大报文段大小(MSS)为1460，tcp_output建立一个报文段来发送包含前1460字节的数据。它还建立一个包含IP和TCP首部的mbuf，为链路层首部(16字节)预留了空间，并将这个mbuf链传给IP输出。在接口输出队列尾部的mbuf链显示在图2-25中。

在1.9节的UDP例子中，UDP用mbuf链来存放数据报，在前面添加一个mbuf来存放协议首部，并把此链传给IP输出。UDP并不把这个mbuf保存在它的发送缓存中。而TCP不能这样做，因为TCP是一个可靠协议，并且它必须维护一个发送数据的副本，直到数据被对方确认。

在这个例子中，tcp_output调用函数m_copy，请求复制1460字节的数据，从发送缓存起始位置开始。但由于数据被存放在一个簇中，m_copy创建一个mbuf(图2-25的右下侧)并且对它初始化，将它指向那个已存在的簇的正确位置(此例中是簇的起始处)。这个mbuf的长度是1460，虽然有另外588字节的数据在簇中。我们所示的这个mbuf链的长度是1514，包括以太网首部、IP首部和TCP首部。

在图2-25的右下侧我们还显示了这个mbuf包含一个分组首部，但它不是链中的第一个mbuf。当m_copy复制一个包含一个分组首部的mbuf并且从原来mbuf的起始地址开始复制时，分组首部也被复制下来。因为这个mbuf不是链中的第一个mbuf，

这个额外的分组首部被忽略。而在这个额外的分组首部中的 m_pkthdr.len的值2048也被忽略。

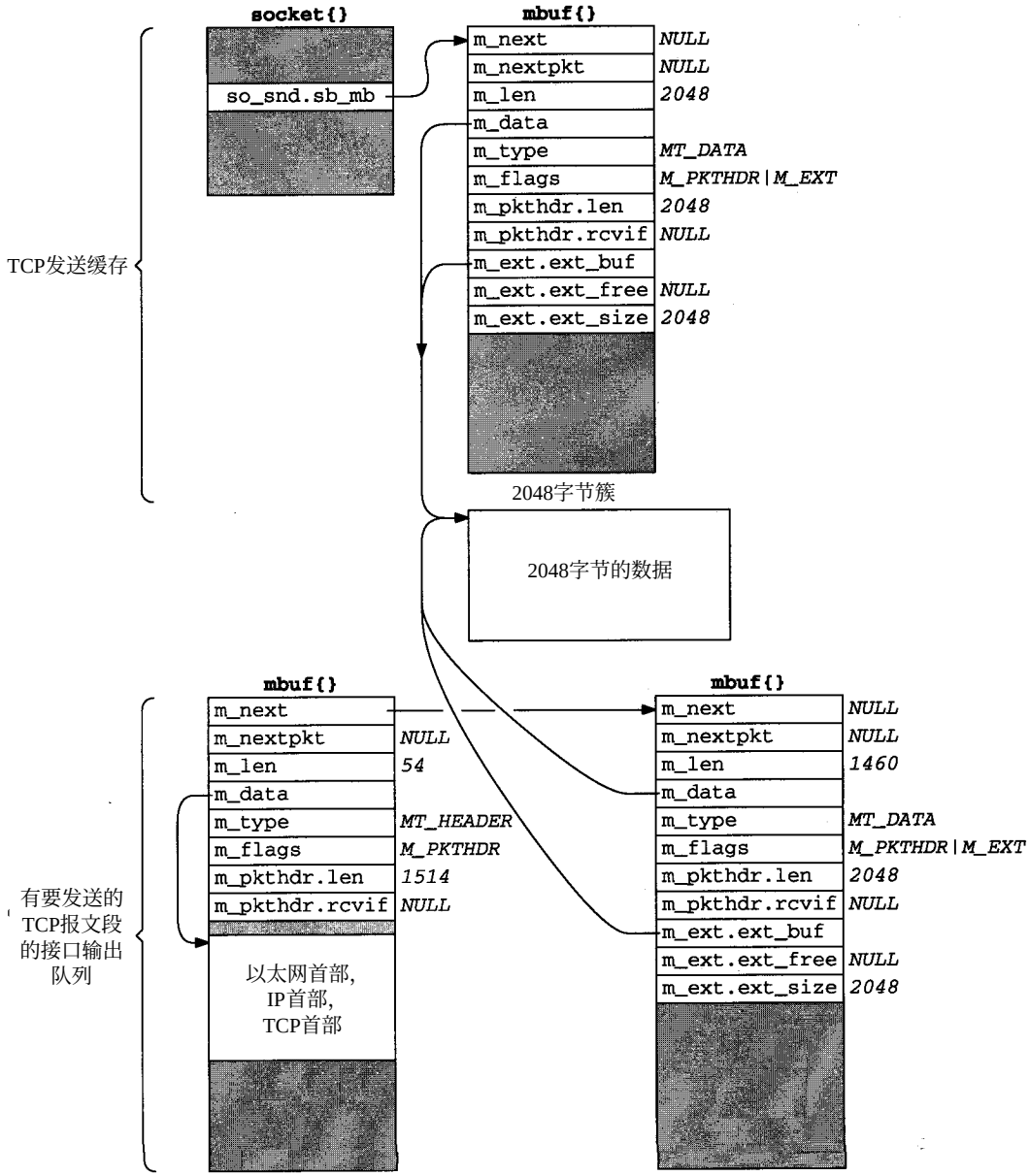


图2-25 TCP插口发送缓存和接口输出队列中的报文段

这个共享的簇避免了内核将数据从一个 mbuf复制到另一个 mbuf中 —— 这节约了很多开销。它是通过为每个簇提供一个引用计数来实现的，每次另一个 mbuf指向这个簇时计数加 1，当一个簇释放时计数减 1。仅当引用计数到达 0时，被这个簇占用的存储器才能被其他程序使用(见习题2.4)。

例如，当图2-25底部的mbuf链到达以太网设备驱动程序并且它的内容已被复制给这个设备时，驱动程序调用 m_freem。这个函数释放带有协议首部的第一个 mbuf，并注意到链中第

二个mbuf指向一个簇。簇引用计数减 1，但由于它的值变成了 1，它仍然保存在存储器中。它不能被释放，因为它仍在TCP发送缓存中。

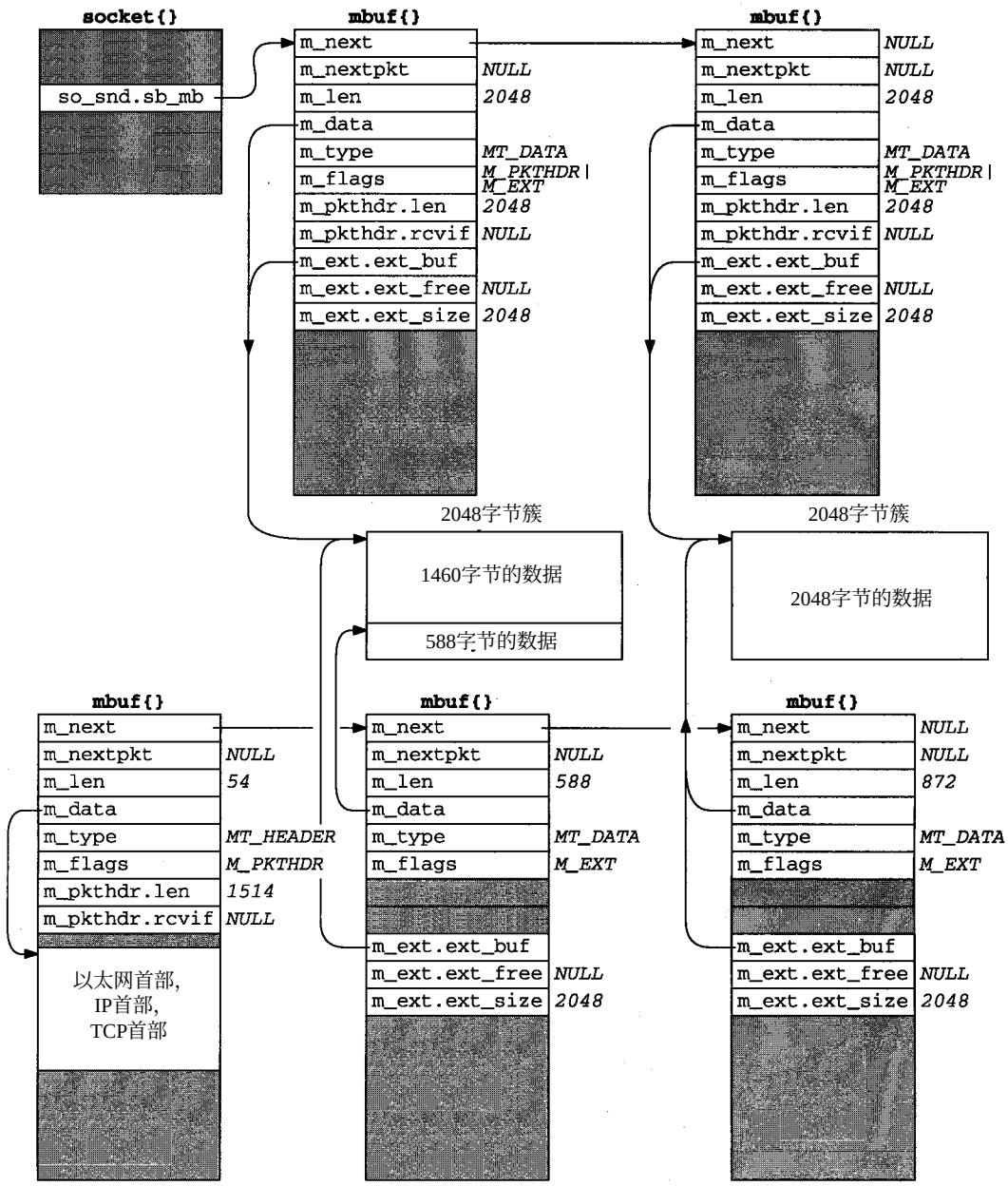


图2-26 用于发送1460字节TCP报文段的mbuf链

继续我们的例子，由于在发送缓存中剩余的 588字节不能组成一个报文段，tcp_output在把1460字节的报文段传给IP后返回(在第26章我们要详细说明在这种条件下 tcp_output发送数据的细节)。插口层继续处理来自应用程序的数据：剩下的 2048字节被存放到一个带有一个簇的mbuf中，TCP发送例程再次被调用，并且新的 mbuf被追加到插口发送缓存中。因为能发送一个完整的报文段，tcp_output建立另一个带有协议首部和1460字节数据的mbuf链表。

m_copy的参数指定了1460字节的数据在发送缓存中的起始位移和长度 (1460字节)。这显示在图2-26中，并假设这个mbuf链在接口输出队列中(这个链中的第一个mbuf的长度反映了以太网首部、IP首部及TCP首部)。

这次1460字节的数据来自两个簇：前 588字节来自发送缓存的第一个簇而后面的 872字节来自发送缓存的第二个簇。它用两个 mbuf来存放1460字节，但m_copy还是不复制这1460字节的数据 —— 它引用已存在的簇。

这次我们没有在图 2-26右下侧的任何 mbuf中显示一个分组首部。原因是调用m_copy的起始位移为零。但在插口发送缓存中的第二个 mbuf包含一个分组首部，而不是链中的第一个mbuf。这是函数sosend的特点，这个额外的分组首部被简单地忽略了。

我们在通篇中会多次遇到函数m_copy。虽然这个名字隐含着对数据进行物理复制，但如果数据被包含在一个簇中，却是仅引用这个簇而不是复制。

2.10 其他选择

mbuf远非完美，并且时常遭到批评。但不管怎样，它们形成了所有今天正使用着的伯克利联网代码的基础。

一种由Van Jacobson [Partridge 1993]完成的Internet协议的研究实现，它废除了支持大量连续缓存的复杂的mbuf数据结构。[Jacobson 1993]提出了一种速度能提高一到两个数量级的改进方案，还包括其他改进，及废除mbuf。

这个mbuf的复杂性是一种权衡，以避免分配大的固定长度的缓存，这样的大缓存很少能被装满。而在这种情况下， mbuf要进行设计，一个VAX-11/780有4兆存储器，是一个大系统，并且存储器是昂贵的资源，需要仔细分配。今天存储器已不昂贵了，而焦点已经转向更高的性能和代码的简单性。

mbuf的性能基于存放在mbuf中数据量。[Hutchinson and Peterson 1991]显示了处理mbuf的时间与数据量不是线性关系。

2.11 小结

在本书几乎所有的函数中我们都会遇到 mbuf。它们的主要用途是在进程和网络接口之间传递用户数据时用来存放用户数据，但 mbuf还用于保存其他各种数据：源地址和目标地址、插口选项等等。

根据M_PKTHDR和M_EXT标志是否被设置，这里有4种类型的mbuf：

- 无分组首部， mbuf本身带有0~108字节数据；
- 有分组首部， mbuf本身带有0~100字节数据；
- 无分组首部，数据在簇(外部缓存)中；
- 有分组首部，数据在簇(外部缓存)中。

我们查看了几个mbuf宏和函数的源代码，但不是所有的mbuf例程源代码。图2-19和图2-20提供了所有我们在本书中遇到的mbuf例程的函数原型和说明。

查看了我们要遇到的两个函数的操作： m_devget，很多网络设备驱动程序调用它来存

储一个收到的帧；m_pullup，所有输入例程调用它把协议首部连续放置在一个mbuf中。

由一个mbuf指向的簇(外部缓存)能通过m_copy被共享。例如，用于TCP输出，因为一个被传输的数据的副本要被发送端保存，直到数据被对方确认。比起进行物理复制来说，通过引用计数，共享簇提高了性能。

习题

2.1 在图2-9中定义了M_COPYFLAGS。为什么不复制标志M_EXT？

2.2 在2.6节中，我们列出了两个m_pullup失败的原因。实际上有三个原因。查看这个函数的源代码(附录B)，并发现另外一个原因。

2.3 为避免宏dtom遇到在2.6节中我们所讨论的问题——当数据在簇中时，为什么不仅仅给每个簇加一个指向mbuf的回指指针？

2.4 既然一个mbuf簇的大小是2的幂(典型的是1024或2048)，簇内的空间不能用于引用计数。查看Net/3的源代码(附录B)，并确定这些引用计数存储在什么地方。

2.5 在图2-5中，我们注意到两个计数器m_drops和m_wait现在没有实现。修改mbuf例程增加这些计数器。

第3章 接 口 层

3.1 引言

本章开始讨论Net/3在协议栈底部的接口层，它包括在本地网上发送和接收分组的硬件与软件。

我们使用术语设备驱动程序来表示与硬件及网络接口 (或仅仅是接口)通信的软件，网络接口是指在一个特定网络上硬件与设备驱动器之间的接口。

Net/3接口层试图在网络协议和连接到一个系统的网络设备的驱动器间提供一个与硬件无关的编程接口。这个接口层为所有的设备提供以下支持：

- 一套精心定义的接口函数；
- 一套标准的统计与控制标志；
- 一个与设备无关的存储协议地址的方法；
- 一个标准的输出分组的排队方法。

这里不要求接口层提供可靠的分组传输，仅要求提供最大努力 (best-effort)的服务。更高协议层必须弥补这种可靠性缺陷。本章说明为所有网络接口维护的通用数据结构。为了说明相关数据结构和算法，我们参考 Net/3中三种特定的网络接口：

1) 一个AMD 7990 LANCE以太网接口：一个能广播局域网的例子。
2) 一个串行线IP(SLIP)接口：一个在异步串行线上的点对点网络的例子。
3) 一个环回接口：一个逻辑网络把所有输出分组作为输入返回。

3.2 代码介绍

通用接口结构和初始化代码可在三个头文件和两个 C文件中找到。在本章说明的设备专用初始化代码可在另外三个C文件中找到。所有的8个文件都列于图3-1中。

文 件	说 明
sys/socket.h	地址结构定义
net/if.h	接口结构定义
net/if_dl.h	链路层结构定义
kern/init_main.c	系统和接口初始化
net/if.c	通用接口代码
net/if_loop.c	环回设备驱动程序
net/if_sl.c	SLIP设备驱动程序
hp300/dev/if_le.c	LANCE以太网设备驱动程序

图3-1 本章讨论的文件

3.2.1 全局变量

在本章中介绍的全局变量列于图 3-2中。

变　量	数据类型	说　明
pdevinit	struct pdevinit[]	伪设备如SLIP和环回接口的初始化参数数组
ifnet	struct ifnet *	ifnet结构的列表的表头
ifnet_addrs	struct ifaddr **	指向链路层接口地址的指针数组
if_indexlim	int	数组ifnet_addrs的大小
if_index	int	上一个配置接口的索引
ifqmaxlen	int	接口输出队列的最大值
hz	int	这个系统的时钟频率(次/秒)

图3-2　本章中介绍的全局变量

3.2.2　SNMP变量

Net/3内核收集了大量的各种联网统计。在大多数章节中，我们都要总结这些统计并说明它们与定义在简单网络管理协议信息库 (SNMP MIB-II)中的标准TCP/IP信息和统计之间的关系。RFC 1213 [McCloghrie and Rose 1991]说明了SNMP MIB-II，它组织成如图3-3所示的10个不同的信息组。

SNMP组	说　明
System	系统通用信息
Interfaces	网络接口信息
Address Translation	网络地址到硬件地址的映射表(不推荐使用)
IP	IP协议信息
ICMP	ICMP协议信息
TCP	TCP协议信息
UDP	UDP协议信息
EGP	EGP协议信息
Transmission	媒体专用信息
SNMP	SNMP协议信息

图3-3　MIB-II中的SNMP组

Net/3并不包括一个SNMP代理。一个针对Net/3的SNMP代理是作为一个进程来实现的，它根据SNMP的要求通过2.2节描述的机制来访问这些内核统计。

Net/3收集大多数MIB-II变量并且能被SNMP代理直接访问，而其他的变量则要通过间接的方式来获得。MIB-II变量分为三类：(1)简单变量，例如一个整数值、一个时间戳或一个字节串；(2)简单变量的列表，例如一个单独的路由项或一个接口描述项；(3)表的列表，例如整个路由表和所有接口实体的列表。

ISODE包含有一个Net/3 SNMP代理例子。ISODE的信息见附录B。

图3-4所示的是一个为SNMP接口组维护的简单变量。我们在后面的图 4-7中描述SNMP接口表。

SNMP变量	Net/3变量	说　明
ifNumber	if_index + 1	if_index是系统中最后一个接口的索引值，并且起始为0；加1来获得系统中接口个数ifNumber

图3-4　在接口组中的一个简单的SNMP变量

3.3 ifnet结构

结构ifnet中包含所有接口的通用信息。在系统初始化期间，分别为每个网络设备分配一个独立的ifnet结构。每个ifnet结构有一个列表，它包含这个设备的一个或多个协议地址。图3-5说明了一个接口和它的地址之间的关系。

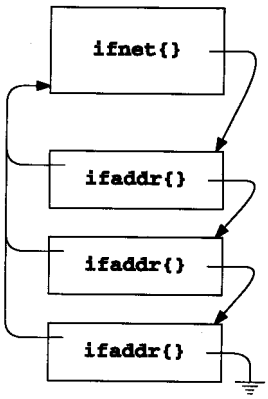


图3-5 每个ifnet结构有一个ifaddr结构的列表

在图3-5中的接口显示了3个存放在ifaddr结构中的协议地址。虽然一些网络接口，例如SLIP，仅支持一个协议；而其他接口，如以太网，支持多个协议并需要多个地址。例如，一个系统可能使用一个以太网接口同时用于Internet和OSI两个协议。一个类型字段标识每个以太网帧的内容，并且因为Internet和OSI协议使用不同的编址方式，以太网接口必须有一个Internet地址和一个OSI地址。所有地址用一个链表链接起来(图3-5右侧的箭头)，并且每个结构包含一个回指指针指向相关的ifnet结构(图3-5左侧的箭头)。

可能一个网络接口支持同一协议的多个地址。例如，在Net/3中可能为一个以太网接口分配两个Internet地址。

> 这个特点第一次是出现在Net/2中。当为一个网络重编地址时，一个接口有两个IP地址是有用的。在过渡期间，接口可以接收老地址和新地址的分组。

结构ifnet比较大，我们分五个部分来说明：

- 实现信息
- 硬件信息
- 接口统计
- 函数指针
- 输出队列

图3-6所示的是包含在结构ifnet中的实现信息。

if.h

```
80 struct ifnet {
81     struct ifnet *if_next;       /* all struct ifnets are chained */
82     struct ifaddr *if_addrlist; /* linked list of addresses per if */
83     char    *if_name;            /* name, e.g. 'le' or 'lo' */
84     short    if_unit;            /* sub-unit for lower level driver */
85     u_short if_index;            /* numeric abbreviation for this if  */
86     short    if_flags;           /* Figure 3.7 */
87     short    if_timer;           /* time 'til if_watchdog called */
88     int      if_pcount;          /* number of promiscuous listeners */
89     caddr_t if_bpf;              /* packet filter structure */
```

if.h

图3-6 ifnet结构：实现信息

80-82 if_next把所有接口的ifnet结构链接成一个链表。函数if_attach在系统初始化期间构造这个链表。if_addrlist指向这个接口的ifaddr结构列表(图3-16)。每个ifaddr结构存储一个要用这个接口通信的协议的地址信息。

1. 通用接口信息

83-86 if_name是一个短字符串，用于标识接口的类型，而if_unit标识多个相同类型的实例。例如，一个系统有两个SLIP接口，每个都有一个if_name，包含两字节的“ s1 ”和一个if_unit。对第一个接口，if_unit为0；对第二个接口为1。if_index在内核中唯一地标识这个接口，这在sysctl系统调用(见19.14节)以及路由域中要用到。

有时一个接口并不被一个协议地址唯一地标识。例如，几个 SLIP连接可以有同样的本地IP地址。在这种情况下，if_index明确地指明这个接口。

if_flags表明接口的操作状态和属性。一个进程能检查所有的标志，但不能改变在图 3-7中“ 内核专用 ”列中作了记号的标志。这些标志用 4.4节讨论的命令SIOCGIFFLAGS和SIOCSIFFLAGS来访问。

if_flags	内核专用	说 明
IFF_BROADCAST	•	接口用于广播网
IFF_MULTICAST	•	接口支持多播
IFF_POINTOPOINT	•	接口用于点对点网络
IFF_LOOPBACK		接口用于环回网络
IFF_OACTIVE	•	正在传输数据
IFF_RUNNING	•	资源已分配给这个接口
IFF_SIMPLEX	•	接口不能接收它自己发送的数据
IFF_LINK0	见正文	由设备驱动程序定义
IFF_LINK1	见正文	由设备驱动程序定义
IFF_LINK2	见正文	由设备驱动程序定义
IFF_ALLMULTI		接口正接收所有多播分组
IFF_DEBUG		这个接口允许调试
IFF_NOARP		在这个接口上不使用ARP协议
IFF_NOTRAILERS		避免使用尾部封装
IFF_PROMISC		接口接收所有网络分组
IFF_UP		接口正在工作

图3-7 if_flags 值

IFF_BROADCAST和IFF_POINTOPOINT标志是互斥的。

宏IFF_CANTCHANGE是对所有在“ 内核专用 ”列中作了记号的标志进行按位“ 或 ”操作。

设备专用标志(IFF_LINK*x*)对于一个依赖这个设备的进程可能是可修改的，也可能是不可修改的。例如，图3-29显示了这些标志是如何被SLIP驱动程序定义的。

2. 接口时钟

87 if_timer以秒为单位记录时间，直到内核为此接口调用函数if_watchdog为止。这个函数用于设备驱动程序定时收集接口统计，或用于复位运行不正确的硬件。

3. BSD分组过滤器

88-89 下面两个成员，if_pcount和if_bpf，支持BSD分组过滤器(BPF)。通过BPF，一个进程能接收由此接口传输或接收的分组的备份。当我们讨论设备驱动程序时，还要讨论分组是如何通过BPF的。BPF在第31章讨论。

ifnet结构的下一个部分显示在图3-8中，它用来描述接口的硬件特性。

if.h

```
90      struct if_data {
91 /* generic interface information */
92          u_char  ifi_type;       /* Figure 3.9 */
93          u_char  ifi_addrlen;    /* media address length */
94          u_char  ifi_hdrlen;     /* media header length */
95          u_long  ifi_mtu;        /* maximum transmission unit */
96          u_long  ifi_metric;     /* routing metric (external only) */
97          u_long  ifi_baudrate;   /* linespeed */
                /* other ifnet members */
138 #define if_mtu      if_data.ifi_mtu
139 #define if_type     if_data.ifi_type
140 #define if_addrlen  if_data.ifi_addrlen
141 #define if_hdrlen   if_data.ifi_hdrlen
142 #define if_metric   if_data.ifi_metric
143 #define if_baudrate if_data.ifi_baudrate
```

if.h

图3-8 ifnet 结构：接口特性

Net/3和本书使用第138行~143行的#define语句定义的短语来表示ifnet的成员。

4. 接口特性

90-92 if_type指明接口支持的硬件地址类型。图3-9列出了net/if_types.h中几个公共的if_type值。

if_type	说 明
IFT_OTHER	未指明
IFT_ETHER	以太网
IFT_ISO88023	IEEE 802.3以太网(CSMA/CD)
IFT_ISO88025	IEEE 802.5令牌环
IFT_FDDI	光纤分布式数据接口
IFT_LOOP	环回接口
IFT_SLIP	串行线IP

图3-9 if_type ：数据链路类型

93-94 if_addrlen是数据链路地址的长度，而if_hdrlen是由硬件附加给任何分组的首部的长度。例如，以太网有一个长度为6字节的地址和一个长度为14字节的首部(图4-8)。

95 if_mtu是接口传输单元的最大值：接口在一次输出操作中能传输的最大数据单元的字节数。这是控制网络和传输协议创建分组大小的重要参数。对于以太网来说，这个值是 1500。

96-97 if_metric通常是0；其他更大的值不利于路由通过此接口。 if_baudrate指定接口的传输速率，只有SLIP接口才设置它。

接口统计由图3-10中显示的下一组ifnet接口成员来收集。

5. 接口统计

98-111 这些统计大多数是不言自明的。当分组传输被共享媒体上其他传输中断时，if_collisions加1。if_noproto统计由于协议不被系统或接口支持而不能处理的分组数

(例如：仅支持IP的系统接收到一个OSI分组)。如果一个非IP分组到达一个SLIP接口的输出队列时，if_noproto加1。

if.h

```
 98 /* volatile statistics */
 99         u_long  ifi_ipackets;   /* #packets received on interface */
100         u_long  ifi_ierrors;    /* #input errors on interface */
101         u_long  ifi_opackets;   /* #packets sent on interface */
102         u_long  ifi_oerrors;    /* #output errors on interface */
103         u_long  ifi_collisions; /* #collisions on csma interfaces */
104         u_long  ifi_ibytes;     /* #bytes received */
105         u_long  ifi_obytes;     /* #bytes sent */
106         u_long  ifi_imcasts;    /* #packets received via multicast */
107         u_long  ifi_omcasts;    /* #packets sent via multicast */
108         u_long  ifi_iqdrops;    /* #packets dropped on input, for this
109                                    interface */
110         u_long  ifi_noproto;    /* #packets destined for unsupported
111                                    protocol */
112         struct timeval ifi_lastchange;  /* last updated */
113     } if_data;

                        /* other ifnet members */

144 #define if_ipackets if_data.ifi_ipackets
145 #define if_ierrors  if_data.ifi_ierrors
146 #define if_opackets if_data.ifi_opackets
147 #define if_oerrors  if_data.ifi_oerrors
148 #define if_collisions   if_data.ifi_collisions
149 #define if_ibytes   if_data.ifi_ibytes
150 #define if_obytes   if_data.ifi_obytes
151 #define if_imcasts  if_data.ifi_imcasts
152 #define if_omcasts  if_data.ifi_omcasts
153 #define if_iqdrops  if_data.ifi_iqdrops
154 #define if_noproto  if_data.ifi_noproto
155 #define if_lastchange   if_data.ifi_lastchange
```

if.h

图3-10 结构ifnet ：接口统计

这些统计在Net/1中不是ifnet结构的一部分。它们被加入来支持接口的标准SNMP MIB-II变量。

if_iqdrops仅被SLIP设备驱动程序访问。当IF_DROP被调用时，SLIP和其他网络驱动程序把if_snd.ifq_drops (图3-13)加1。在SNMP统计加入前，ifq_drops就已经存在于BSD软件中了。ISODE SNMP代理忽略if_iqdrops而使用if_snd.ifq_drops。

6. 改变时间戳

112-113 if_lastchange记录任何统计改变的最近时间。

Net/3和本书又一次用从144行到155行的#define语句定义的短名来指明ifnet的成员。

结构ifnet的下一个部分，显示在图3-11中，它包含指向标准接口层函数的指针，它们把设备专用的细节从网络层分离出来。每个网络接口实现这些适用于特定设备的函数。

if.h

```
114 /* procedure handles */
115     int     (*if_init)          /* init routine */
116             (int);
117     int     (*if_output)        /* output routine (enqueue) */
118             (struct ifnet *, struct mbuf *, struct sockaddr *,
119              struct rtentry *);
120     int     (*if_start)         /* initiate output routine */
121             (struct ifnet *);
122     int     (*if_done)          /* output complete routine */
123             (struct ifnet *);   /* (XXX not used; fake prototype) */
124     int     (*if_ioctl)         /* ioctl routine */
125             (struct ifnet *, int, caddr_t);
126     int     (*if_reset)
127             (int);              /* new autoconfig will permit removal */
128     int     (*if_watchdog)      /* timer routine */
129             (int);
```

if.h

图3-11 结构ifnet ：接口过程

7. 接口函数

114-129 在系统初始化时，每个设备驱动程序初始化它自己的 ifnet结构，包括7个函数指针。图3-12说明了这些通用函数。

我们在Net/3中常会看到注释/* XXX */它提醒读者这段代码是易混淆的，包括不明确的副作用，或是一个更难问题的快速解决方案。在这里，它指示 if_done 不在Net/3中使用。

函 数	说 明
if_init	初始化接口
if_output	对要传输的输出分组进行排队
if_start	启动分组的传输
if_done	传输完成后的清除(未用)
if_ioctl	处理I/O控制命令
if_reset	复位接口设备
if_watchdog	周期性接口例程

图3-12 结构ifnet ：函数指针

在第4章我们要查看以太网、SLIP和环回接口的设备专用函数，内核通过 ifnet结构中的这些指针直接调用它们。例如，如果 ifp指向一个ifnet结构，

```
(*ifp->if_start)(ifp)
```

调用这个接口的设备驱动程序的 if_start函数。

结构ifnet中剩下的最后一个成员是接口的输出队列，如图 3-13所示。

130-137 if_snd是接口输出分组队列，每个接口有它自己的 ifnet结构，即它自己的输出队列。ifq_head指向队列的第一个分组(下一个要输出的分组)，ifq_tail指向队列最后一个分组，if_len是当前队列中分组的数目，而 ifq_maxlen是队列中允许的缓存最大个数。除非驱动程序修改它，这个最大值被设置为 50(来源于全局整数ifqmaxlen，它在编译期间根据IFQ_MAXLEN初始化而来)。队列作为一个mbuf链的链表来实现。ifq_drops统计

因为队列满而丢弃的分组数。图3-14列出了那些访问队列的宏和函数。

if.h

```
130     struct ifqueue {
131         struct mbuf *ifq_head;
132         struct mbuf *ifq_tail;
133         int     ifq_len;        /* current length of queue */
134         int     ifq_maxlen;     /* maximum length of queue */
135         int     ifq_drops;      /* packets dropped because of full queue */
136     } if_snd;                   /* output queue */
137 };
```

if.h

图3-13 结构ifnet ：输出队列

函 数	说 明
IF_QFULL	*ifq*是否满 int **IF_QFULL**(struct ifqueue *ifq);
IF_DROP	IF_DROP仅将与*ifq*关联的ifq_drops加1。这个名字会引起误导：调用者丢弃这个分组 void **IF_DROP**(struct ifqueue *ifq);
IF_ENQUEUE	把分组*m*追加到*ifq*队列的后面。分组通过mbuf首部中的m_nextpkt链接在一起 void **IF_ENQUEUE**(struct ifqueue *ifq, struct mbuf *m);
IF_PREPEND	把分组*m*插入到*ifq*队列的前面 void **IF_PREPEND**(struct ifqueue *ifq, struct mbuf *m);
IF_DEQUEUE	从*ifq*队列中取走第一个分组。*m*指向取走的分组，若队列为空，则*m*为空值 void **IF_DEQUEUE**(struct ifqueue *ifq, struct mbuf *m);
if_qflush	丢弃队列*ifq*中的所有分组，例如，当一个接口被关闭了 void **if_qflush**(struct ifqueue *ifq);

图3-14 fiqueue 例程

前5个例程是定义在net/if.h中的宏，最后一个例程if_qflush是定义在net/if.c中的一个函数。这些宏经常出现在下面这样的程序语句中：

```
s = splimp();
if (IF_QFULL(inq)) {
    IF_DROP(inq);           /* queue is full, drop new packet */
    m_freem(m);
} else
    IF_ENQUEUE(inq, m);     /* there is room, add to end of queue */
splx(s);
```

这段代码试图把一个分组加到队列中。如果队列满， IF_DROP把ifq_drops加1，并且分组被丢弃。可靠协议如TCP会重传丢弃的分组。使用不可靠协议(如UDP)的应用程序必须自己检测和处理重传。

访问队列的语句被splimp和splx括起来，阻止网络中断，并且防止在不确定状态时网络中断服务例程访问此队列。

在splx之前调用m_freem，是因为这段mbuf代码有一个临界区运行在splimp级别上。若在m_freem前调用splx，在m_freem中进入另一个临界区(2.5节)是浪费效率的。

3.4 ifaddr结构

我们要看的下一个结构是接口地址结构，ifaddr，它显示在图3-15中。每个接口维护一个ifaddr结构的链表，因为一些数据链路，如以太网，支持多于一个的协议。一个单独的ifaddr结构描述每个分配给接口的地址，通常每个协议一个地址。支持多地址的另一个原因是很多协议，包括TCP/IP，支持为单个物理接口指派多个地址。虽然 Net/3支持这个特性，但很多TCP/IP实现并不支持。

if.h

```
217 struct ifaddr {
218         struct  ifaddr *ifa_next;       /* next address for interface */
219         struct  ifnet *ifa_ifp;         /* back-pointer to interface */
220         struct  sockaddr *ifa_addr;     /* address of interface */
221         struct  sockaddr *ifa_dstaddr;  /* other end of p-to-p link */
222 #define ifa_broadaddr   ifa_dstaddr     /* broadcast address interface */
223         struct  sockaddr *ifa_netmask;  /* used to determine subnet */
224         void    (*ifa_rtrequest)();     /* check or clean routes */
225         u_short ifa_flags;              /* mostly rt_flags for cloning */
226         short   ifa_refcnt;             /* references to this structure */
227         int     ifa_metric;             /* cost for this interface */
228 };
```

if.h

图3-15 结构ifaddr

217-219 结构ifaddr通过ifa_next把分配给一个接口的所有地址链接起来，它还包括一个指回接口的ifnet结构的指针ifa_ifp。图3-16显示了结构ifnet与ifaddr之间的关系。

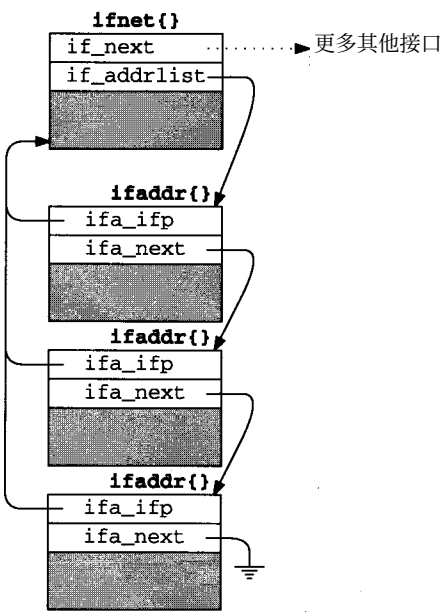


图3-16 结构ifnet 和ifaddr

220 ifa_addr指向接口的一个协议地址，而ifa_netmask指向一个位掩码，它用于选择ifa_addr中的网络部分。地址中表示网络部分的比特在掩码中被设置为 1，地址中表示主机的部分被设置为0。两个地址都存放在 sockaddr结构中(3.5节)。图3-38显示了一个地址及其掩码结构。对于IP地址，掩码选择IP地址中的网络和子网部分。

221-223 ifa_dstaddr(或它的别名ifa_broadaddr)指向一个点对点链路上的另一端的接口协议地址或指向一个广播网中分配给接口的广播地址(如以太网)。接口的ifnet结构中互斥的两个标志IFF_BROADCAST和IFF_POINTOPOINT(图3-7)指示接口的类型。

224-228 ifa_rtrequest、ifa_flags和ifa_metric支持接口的路由查找。

ifa_refcnt统计对结构ifaddr的引用。宏IFAFREE仅在引用计数降到0时才释放这个结构，例如，当地址被命令SIOCDIFADDRioctl删除时。结构ifaddr使用引用计数是因为接口和路由数据结构共享这些结构。

如果有其他对ifaddr的引用，IFAFREE将计数器加1并返回。这是一个通用的方法，除了最后一个引用外，它避免了每次都调用一个函数的开销。如果是最后一个引用，IFAFREE调用函数ifafree，来释放这个结构。

3.5 sockaddr结构

一个接口的编址信息不仅仅只包括一个主机地址。 Net/3在通用的sockaddr结构中维护主机地址、广播地址和网络掩码。通过使用一个通用的结构，将硬件与协议专用的地址细节相对于接口层隐藏起来。

图3-17显示的是这个结构的当前定义及早期 BSD版的定义 —— 结构osockaddr。图3-18说明了这些结构的组织。

socket.h

```
120 struct sockaddr {
121     u_char  sa_len;             /* total length */
122     u_char  sa_family;          /* address family (Figure 3.19) */
123     char    sa_data[14];        /* actually longer; address value */
124 };

271 struct osockaddr {
272     u_short sa_family;          /* address family (Figure 3.19) */
273     char    sa_data[14];        /* up to 14 bytes of direct address */
274 };
```

socket.h

图3-17 结构sockaddr 和osockaddr

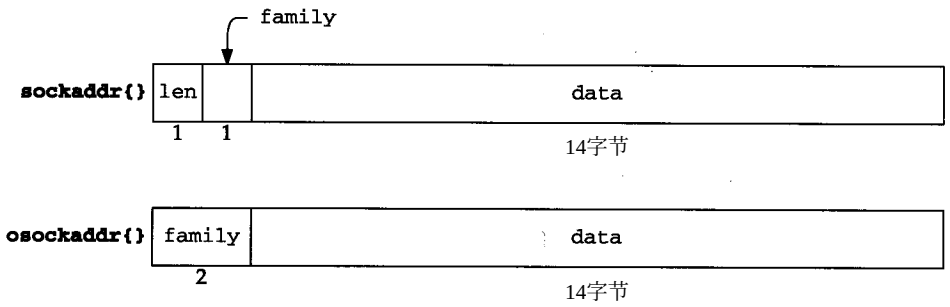


图3-18 结构sockaddr 和osockaddr(省略了前缀sa_)

在很多图中，我们省略了成员名中的公共前缀。在这里，我们省略了 sa_前缀。

1. Sockaddr结构

120-124 每个协议有它自己的地址格式。 Net/3在一个sockaddr结构中处理通用的地址。sa_len指示地址的长度(OSI和Unix域协议有不同长度的地址), sa_family指示地址的类型。图3-19列出了地址族(address family)常量，其中包括我们遇到的。

当指明为AF_UNSPEC时，一个sockaddr的内容要根据情况而定。大多数情况下，它包含一个以太网硬件地址。

成员sa_len和sa_family允许协议无关代码操作来自多个协议的变长的 sockaddr结构。剩下的成员sa_data，包含一个协议相关格式的地址。 sa_data定义为一个14字节的数组，但当sockaddr结构覆盖更大的内存空间时， sa_data可能会扩展到253字节。sa_len

仅有一个字节，因此整个地址，包括 sa_len和sa_family必须不超过256字节。

这是C语言的一种通用技术，它允许程序员把一个结构中的最后一个成员看成是可变长的。

sa_family	协 议
AF_INET	Internet
AF_ISO,AF_OSI	OSI
AF_UNIX	Unix
AF_ROUTE	路由表
AF_LINK	数据链路
AF_UNSPEC	(见正文)

图3-19 sa_family 常量

每个协议定义一个专用的sockaddr结构，该结构复制成员sa_len和sa_family，但按那个协议的要求来定义成员sa_data。存储在sa_data中的地址是一个传输地址；它包含足够的信息来标识同一主机上的多个通信端点。在第 6章我们要查看Internet地址结构sockaddr_in，它包含了一个IP地址和一个端口号。

2. Osockaddr结构

271-274 结构osockaddr是4.3BSD Reno版本以前的sockaddr定义。因为在这个定义中一个地址的长度不是显式地可用，所以它不能用来写处理可变长地址的协议无关代码。 OSI协议使用可变长地址，为了包括 OSI协议，使得在Net/3的sockaddr定义中有了我们所见的改变。结构osockaddr是为了支持对以前编译的程序的二进制兼容。

在本书中我们省略了二进制兼容代码。

3.6 ifnet与ifaddr的专用化

结构ifnet和ifaddr包含适用于所有网络接口和协议地址的通用信息。为了容纳其他设备和协议专用信息，每个设备定义了并且每个协议分配了一个专用化版本的 ifnet和ifaddr结构。这些专用化的结构总是包含一个 ifnet或ifaddr结构作为它们的第一个成员，这样无须考虑其他专用信息就能访问这些公共信息。

多数设备驱动程序通过分配一个专用化的 ifnet结构的数组来处理同一类型的多个接口，但其他设备(例如环回设备)仅处理一个接口。图3-20所示的是我们的例子接口的专用化 ifnet结构的组织。

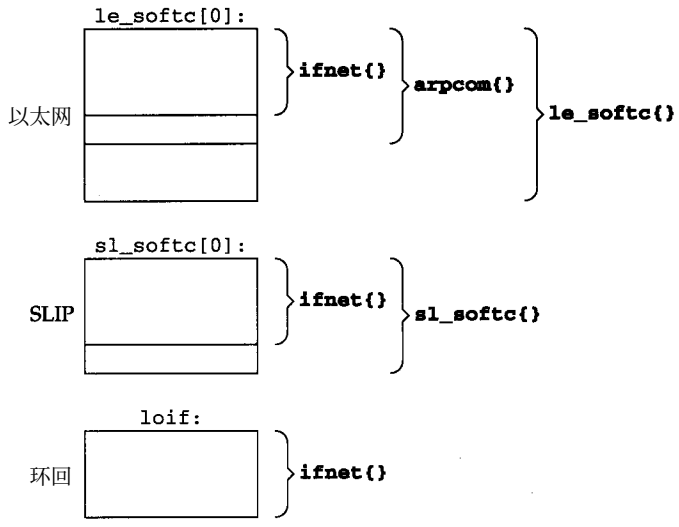


图3-20 设备相关的结构中的 ifnet 结构的组织

注意，每个设备的结构以一个ifnet开始，接下来全是设备相关的数据。环回接口只声名了一个ifnet结构，因为它不要求任何设备相关的数据。在图3-20中，我们显示的以太网和SLIP驱动程序的结构softc带有数组下标0，因为两个设备都支持多个接口。任何给定类型的接口的最大个数由内核建立时的配置参数来限制。

结构arpcom(图3-26)对于所有以太网设备是通用的，并且包含地址解析协议(ARP)和以太网多播信息。结构le_softc(图3-25)包含专用于LANCE以太网设备驱动器的其他信息。

每个协议把每个接口的地址信息存储在一个专用化的ifaddr结构的列表中。以太网协议使用一个in_ifaddr结构(6.5节)，而OSI协议使用一个iso_ifaddr结构。另外，当接口被初始化时，内核为每个接口分配了一个链路层地址，它在内核中标识这个接口。

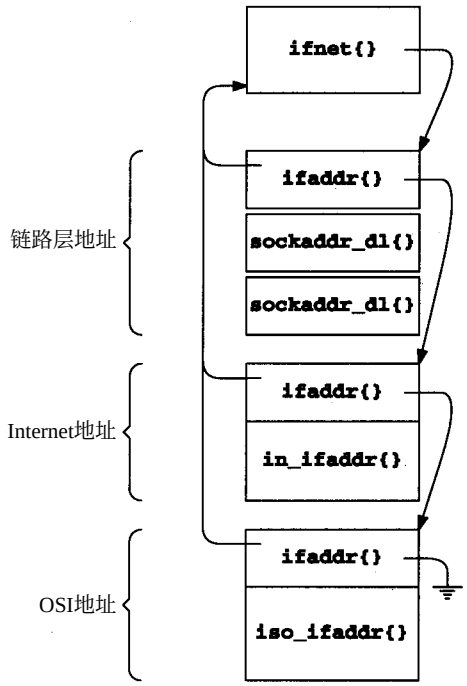


图3-21 一个包含链路层地址、Internet地址和OSI地址的接口地址列表

内核通过分配一个ifaddr结构和两个sockaddr_dl结构(一个是链路层地址本身，一个是地址掩码)来构造一个链路层地址。结构sockaddr_dl可被OSI、ARP和路由算法访问。图3-21显示的是一个带有一个链路层地址、一个Internet地址和一个OSI地址的以太网接口。3.11节说明了链路层地址(ifaddr和两个sockaddr_ld结构)的构造和初始化。

3.7 网络初始化概述

所有我们说明的结构是在内核初始化时分配和互相链接起来的。在本节我们大致概述一下初始化的步骤。在后面的章节，我们说明特定设备的初始化步骤和特定协议的初始化步骤。

有些设备，例如SLIP和环回接口，完全用软件来实现。这些伪设备用存储在全局pdevinit数组中的一个pdevinit结构来表示(图3-22)。在内核配置期间构造这个数组。例如：

```
struct pdevinit pdevinit[] = {
    { slattach, 1 },
    { loopattach, 1 },
    { 0, 0 }
};
```

device.h

```
120 struct pdevinit {
121     void    (*pdev_attach) (int);   /* attach function */
122     int     pdev_count;             /* number of devices */
123 };
```

device.h

图3-22 结构pdevinit

120-123 对于SLIP和环回接口，在结构pdevinit中，pdev_attach分别被设置为

slattach和loopattach。当调用这个attach函数时，pdev_count作为传递的唯一参数，它指定创建的设备个数。只有一个环回设备被创建，但如果管理员适当配置 SLIP项可能有多个SLIP设备被创建。

网络初始化函数从main开始显示在图3-23中。

init_main.c

```
70 main(framep)
71 void   *framep;
72 {
                        /* nonnetwork code */
96      cpu_startup();              /* locate and initialize devices */
                        /* nonnetwork code */
172     /* Attach pseudo-devices. (e.g., SLIP and loopback interfaces) */
173     for (pdev = pdevinit; pdev->pdev_attach != NULL; pdev++)
174         (*pdev->pdev_attach) (pdev->pdev_count);
175     /*
176      * Initialize protocols.  Block reception of incoming packets
177      * until everything is ready.
178      */
179     s = splimp();
180     ifinit();                   /* initialize network interfaces */
181     domaininit();               /* initialize protocol domains */
182     splx(s);
                        /* nonnetwork code */
231     /* The scheduler is an infinite loop. */
232     scheduler();
233     /* NOTREACHED */
234 }
```

init_main.c

图3-23 main 函数：网络初始化

70-96 cpu_startup查找并初始化所有连接到系统的硬件设备，包括任何网络接口。

97-174 在内核初始化硬件设备后，它调用包含在 pdevinit数组中的每个pdev_attach函数。

175-234 ifinit和domaininit完成网络接口和协议的初始化，并且 scheduler开始内核进程调度。ifinit和domaininit在第7章讨论。

在下面几节中，我们说明以太网、 SLIP和环回接口的初始化。

3.8 以太网初始化

作为cpu_startup的一部分，内核查找任何连接的网络设备。这个进程的细节超出了本书的范围。一旦一个设备被识别，一个设备专用的初始化函数就被调用。图 3-24显示的是我们的3个例子接口的初始化函数。

每个设备驱动程序为一个网络接口初始化一个专用化的ifnet结构，并调用if_attach把这个结构插入到接口链表中。显示在图 3-25中的结构le_softc是我们的例子以太网驱动程序的专用化 ifnet结构(图3-20)。

设　备	初始化函数
LANCE以太网	leattach
SLIP	slattach
环回	loopattach

图3-24　网络接口初始化函数

1. le_softc结构

69-95　在if_le.c中声明了一个le_softc结构(有NLE成员)的数组。每个结构的第一个成员是 sc_ac，一个arpcom结构，它对于所有以太网接口都是通用的，接下来是设备专用成员。宏 sc_if和sc_addr简化了对结构ifnet及存储在结构arpcom(sc_ac)中的以太网地址的访问，如图3-26所示。

if_le.c

```
69 struct le_softc {
70     struct arpcom sc_ac;          /* common Ethernet structures */
71 #define sc_if   sc_ac.ac_if       /* network-visible interface */
72 #define sc_addr sc_ac.ac_enaddr /* hardware Ethernet address */

                    /* device-specific members */

95 } le_softc[NLE];
```

if_le.c

图3-25　结构le_softc

if_ether.h

```
95 struct arpcom {
96     struct ifnet ac_if;           /* network-visible interface */
97     u_char  ac_enaddr[6];         /* ethernet hardware address */
98     struct in_addr ac_ipaddr;     /* copy of ip address - XXX  */
99     struct ether_multi *ac_multiaddrs;  /* list of ether multicast addrs */
100    int     ac_multicnt;          /* length of ac_multiaddrs list */
101 };
```

if_ether.h

图3-26　结构arpcom

2. arpcom结构

95-101　结构arpcom的第一个成员ac_if是一个ifnet结构，如图3-20所示。ac_enaddr是以太网硬件地址，它是在 cpu_startup期间内核检测设备时由 LANCE设备驱动程序从硬件上复制的。对于我们的例子驱动程序，这发生在函数 leattach中(图3-27)。ac_ipaddr是上一个分配给此设备的 IP地址。我们在6.6节讨论地址的分配，可以看到一个接口可以有多个IP地址。见习题6.3。ac_multiaddrs是一个用结构ether_multi表示的以太网多播地址的列表。ac_multicnt统计这个列表的项数。多播列表在第 12章讨论。

图3-27所示的是LANCE以太网驱动程序的初始化代码。

106-115　内核在系统中每发现一个LANCE卡都调用一次leattach。

只有一个指向一个hp_device结构的参数，它包含了 HP专用信息，因为它是专为HP工作站编写的驱动程序。

le指向此卡的专用化ifnet结构(图3-20)，ifp指向这个结构的第一个成员 sc_if，一个通用的ifnet结构。图3-27并不包括设备专用初始化代码，它在本书中不予讨论。

if_le.c

```
106 leattach(hd)
107 struct hp_device *hd;
108 {
109     struct lereg0 *ler0;
110     struct lereg2 *ler2;
111     struct lereg2 *lemem = 0;
112     struct le_softc *le = &le_softc[hd->hp_unit];
113     struct ifnet *ifp = &le->sc_if;
114     char   *cp;
115     int     i;

                    /* device-specific code */

126     /*
127      * Read the ethernet address off the board, one nibble at a time.
128      */
129     cp = (char *) (lestd[3] + (int) hd->hp_addr);
130     for (i = 0; i < sizeof(le->sc_addr); i++) {
131         le->sc_addr[i] = (*++cp & 0xF) << 4;
132         cp++;
133         le->sc_addr[i] |= *++cp & 0xF;
134         cp++;
135     }
136     printf("le%d: hardware address %s\n", hd->hp_unit,
137            ether_sprintf(le->sc_addr));

                    /* device-specific code */

150     ifp->if_unit = hd->hp_unit;
151     ifp->if_name = "le";
152     ifp->if_mtu = ETHERMTU;
153     ifp->if_init = leinit;
154     ifp->if_reset = lereset;
155     ifp->if_ioctl = leioctl;
156     ifp->if_output = ether_output;
157     ifp->if_start = lestart;
158     ifp->if_flags = IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST;
159     bpfattach(&ifp->if_bpf, ifp, DLT_EN10MB, sizeof(struct ether_header));
160     if_attach(ifp);
161     return (1);
162 }
```

if_le.c

图3-27 函数leattach

3. 从设备复制硬件地址

126-137 对于LANCE设备，由厂商指派的以太网地址在这个循环中以每次半个字节 (4 bit) 从设备复制到sc_addr(即sc_ac.ac_enaddr —— 见图3-26)。

lestd是一个设备专用的位移表，用于定位hp_addr的相关信息，hp_addr指向LANCE专用信息。

通过printf语句将完整的地址输出到控制台，来指示此设备存在并且可操作。

4. 初始化ifnet结构

150-157 leattach从hp_device结构把设备单元号复制到if_unit来标识同类型的多

个接口。这个设备的if_name是“ le ”；if_mtu为1500字节(ETHERMTU)，以太网的最大传输单元；if_init、if_reset、if_ioctl、if_output和it_start都指向控制网络接口的通用函数的设备专用实现。4.1节说明这些函数。

158 所有的以太网设备都支持IFF_BROADCAST。LANCE设备不接收它自己发送的数据，因此被设置为IFF_SIMPLEX。支持多播的设备和硬件还要设置IFF_MULTICAST。

159-162 bpfattach登记有BPF的接口，在图31-8中说明。函数if_attach把初始化了的ifnet结构插入到接口的链表中(3.11节)。

3.9 SLIP初始化

依赖标准异步串行设备的SLIP接口在调用cpu_startup时初始化。当main直接通过SLIP的pdevinit结构中的指针pdev_attach调用slattach时，SLIP伪设备被初始化。

每个SLIP接口由图3-28中的一个sl_softc结构来描述。

if_slvar.h

```
43 struct sl_softc {
44     struct ifnet sc_if;          /* network-visible interface */
45     struct ifqueue sc_fastq;     /* interactive output queue */
46     struct tty *sc_ttyp;         /* pointer to tty structure */
47     u_char *sc_mp;               /* pointer to next available buf char */
48     u_char *sc_ep;               /* pointer to last available buf char */
49     u_char *sc_buf;              /* input buffer */
50     u_int   sc_flags;            /* Figure 3.29 */
51     u_int   sc_escape;           /* =1 if last char input was FRAME_ESCAPE */
52     struct slcompress sc_comp;   /* tcp compression data */
53     caddr_t sc_bpf;              /* BPF data */
54 };
```

if_slvar.h

图3-28 结构sl_softc

43-54 与所有接口结构一样，sl_softc有一个ifnet结构并且后面跟着设备专用信息。

除了在ifnet结构中的输出队列外，一个SLIP设备还维护另一个队列sc_fastq，它用于要求低时延服务的分组——典型地由交互应用产生。

sc_ttyp指向关联的终端设备。指针sc_buf和sc_ep分别指向一个接收SLIP分组的缓存的第一个字节和最后一个字节。sc_mp指向下一个接收字节的地址，并在另一个字节到达时向前移动。

SLIP定义的4个标志显示在图3-29中。

常 量	sc_softc成员	说 明
SC_COMPRESS	sc_if.if_flags	IFF_LINK0；压缩TCP通信
SC_NOICMP	sc_if.if_flags	IFF_LINK1；禁止ICMP通信
SC_AUTOCOMP	sc_if.if_flags	IFF_LINK2；允许TCP自动压缩
SC_ERROR	sc_flags	检测到错误；丢弃接收帧

图3-29 SLIP的if_flags 和sc_flags 值

SLIP在ifnet结构中定义了3个接口标志预留给设备驱动程序，另一个标志定义在结构sl_softc中。

sc_escape用于串行线的IP封装机制(5.3节)，而TCP首部压缩信息(29.13节)保留在

sc_comp中。

指针sc_bpf指向SLIP设备的BPF信息。

结构sl_softc由slattach初始化，如图3-30所示。

135-152 不像leattach一次仅初始化一个接口，内核只调用一次 slattach，并且slattach初始化所有的SLIP接口。硬件设备在内核执行 cpu_startup被发现时初始化，而伪设备都是在 main为这个设备调用 pdev_attach函数时被初始化的。一个 SLIP设备的if_mtu为296字节(SLMTU)。这包括标准的20字节IP首部、标准的20字节TCP首部和256字节的用户数据(5.3节)。

if_sl.c

```
135 void
136 slattach()
137 {
138     struct sl_softc *sc;
139     int     i = 0;

140     for (sc = sl_softc; i < NSL; sc++) {
141         sc->sc_if.if_name = "sl";
142         sc->sc_if.if_next = NULL;
143         sc->sc_if.if_unit = i++;
144         sc->sc_if.if_mtu = SLMTU;
145         sc->sc_if.if_flags =
146             IFF_POINTOPOINT | SC_AUTOCOMP | IFF_MULTICAST;
147         sc->sc_if.if_type = IFT_SLIP;
148         sc->sc_if.if_ioctl = slioctl;
149         sc->sc_if.if_output = sloutput;
150         sc->sc_if.if_snd.ifq_maxlen = 50;
151         sc->sc_fastq.ifq_maxlen = 32;
152         if_attach(&sc->sc_if);
153         bpfattach(&sc->sc_bpf, &sc->sc_if, DLT_SLIP, SLIP_HDRLEN);
154     }
155 }
```

if_sl.c

图3-30 函数slattach

一个SLIP网络由位于一个串行通信线两端的两个接口组成。 slattach在if_flags中设置IFF_POINTOPOINT、SC_AUTOCOMP和IFF_MULTICAST。

SLIP接口限制它的输出分组队列if_snd的长度为50，并且它自己的接口队列sc_fastq的长度为32。图3-42显示if_snd队列的长度默认为50(ifqmaxlen)，因此，如果设备驱动程序选择一个长度，这里的初始化是多余的。

以太网设备驱动程序不显式地设置它的输出队列的长度，它依赖于 ifinit (图3-42)把它设置为系统的默认值。

if_attach需要一个指向一个ifnet结构的指针，因此slattach将sc_if的地址传递给if_attach，sc_if是一个第一个成员为结构sl_softc的ifnet结构。

专用程序slattach在内核初始化后运行(从初始化文件/etc/netstart)，并通过打开串行设备和执行ioctl命令(5.3节)添加SLIP接口和一个异步串行设备。

153-155 对于每个SLIP设备，slattach调用bpfattach来登记有BPF的接口。

3.10 环回初始化

最后显示环回接口的初始化。环回接口把输出分组放回到相应的输入队列中。接口没有

相关联的硬件设备。环回伪设备在main通过环回接口的pdevinit结构中的pdev_attach指针直接调用loopattach时初始化。图3-31所示的是函数loopattach。

if_loop.c

```
41 void
42 loopattach(n)
43 int     n;
44 {
45     struct ifnet *ifp = &loif;
46     ifp->if_name = "lo";
47     ifp->if_mtu = LOMTU;
48     ifp->if_flags = IFF_LOOPBACK | IFF_MULTICAST;
49     ifp->if_ioctl = loioctl;
50     ifp->if_output = looutput;
51     ifp->if_type = IFT_LOOP;
52     ifp->if_hdrlen = 0;
53     ifp->if_addrlen = 0;
54     if_attach(ifp);
55     bpfattach(&ifp->if_bpf, ifp, DLT_NULL, sizeof(u_int));
56 }
```

if_loop.c

图3-31 环回接口初始化

41-56 环回if_mtu被设置为1536字节(LOMTU)。在if_flags中设置IFF_LOOPBACK和IFF_MULTICAST。一个环回接口没有链路首部和硬件地址，因此if_hdrlen和if_addrlen被设置为0。if_attach完成ifnet结构的初始化并且bpfattach登记有BPF的环回接口。

环回MTU至少有1576(40 + 3×512)留给一个标准的TCP/IP首部。例如Solaris 2.3环回MTU设置为8232(40 + 8×1024)。这些计算基于Internet协议；而其他协议可能有大于40字节的默认首部。

3.11 if_attach函数

前面显示的三个接口初始化函数都调用 if_attach来完成接口的ifnet结构的初始化，并把这个结构插入到先前配置的接口的列表中。在 if_attach中，内核也为每个接口初始化并分配一个链路层地址。图3-32说明了由if_attach构造的数据结构。

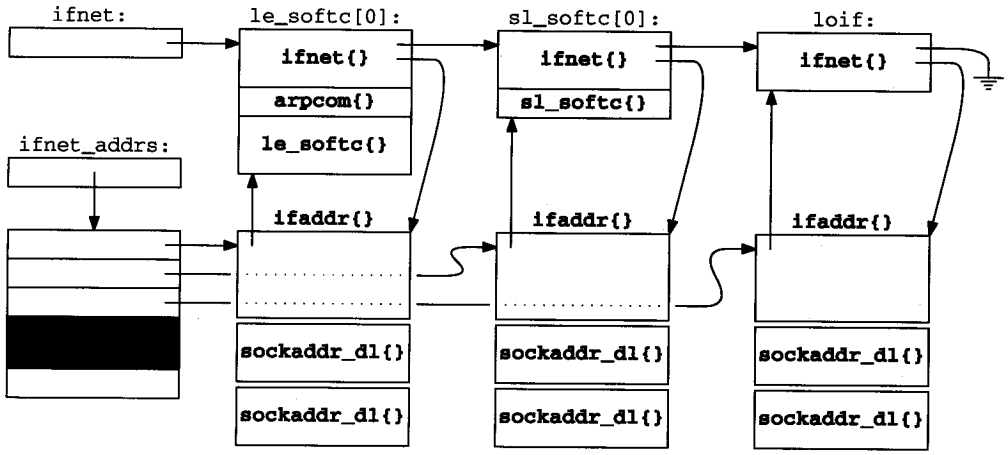


图3-32 ifnet 列表

在图3-32中，if_attach被调用了三次：以一个le_softc结构为参数从leattach调用，以一个sl_softc结构为参数从slattach调用，以一个通用ifnet结构为参数从loopattach调用。每次调用时，它向ifnet列表中添加一个新的ifnet结构，为这个接口创建一个链路层ifaddr结构(包含两个sockaddr_dl结构，图3-33)，并且初始化ifnet_addrs数组中的一项。

if_dl.h

```
55 struct sockaddr_dl {
56     u_char  sdl_len;            /* Total length of sockaddr */
57     u_char  sdl_family;         /* AF_LINK */
58     u_short sdl_index;          /* if != 0, system given index for
59                                    interface */
60     u_char  sdl_type;           /* interface type (Figure 3.9) */
61     u_char  sdl_nlen;           /* interface name length, no trailing 0
62                                    reqd. */
63     u_char  sdl_alen;           /* link level address length */
64     u_char  sdl_slen;           /* link layer selector length */
65     char    sdl_data[12];       /* minimum work area, can be larger;
66                                    contains both if name and ll address */
67 };

68 #define LLADDR(s) ((caddr_t)((s)->sdl_data + (s)->sdl_nlen))
```

if_dl.h

图3-33 结构sockaddr_dl

图3-20显示了包含在le_softc[0]和sl_softc[0]中嵌套的结构。

初始化以后，接口仅配置链路层地址。例如， IP地址直到后面讨论的ifconfig程序才配置(6.6节)。

链路层地址包含接口的一个逻辑地址和一个硬件地址 (如果网络支持，例如le0的一个48 bit以太网地址)。在ARP和OSI协议中要用到这个硬件地址，而一个sockaddr_dl中的逻辑地址包含一个名称和这个接口在内核中的索引数值，它支持用于在接口索引和关联ifaddr结构(ifa_ifwithnet，图6-32)间相互转换的表查找。

结构sockaddr_dl显示在图3-33中。

55-57 回忆图3-18，sdl_len指明了整个地址的长度，而sdl_family指明了地址族类，在此例中为AF_LINK。

58 sdl_index在内核中标识接口。图3-32中的以太网接口会有一个为1的索引，SLIP接口的索引为2，而环回接口的索引为3。全局整数变量if_index包含的是内核最近分配的一个索引值。

60 sdl_type根据这个数据链路地址的ifnet结构的成员if_type进行初始化。

61-68 除了一个数字索引，每个接口有一个由结构ifnet的成员if_name和if_unit组成的文本名称。例如，第一个SLIP接口叫“sl0”，而第二个叫“sl1”。文本名称存储在数组sdl_data的前面，并且sdl_nlen为这个名称的字节长度(在我们的SLIP例子中为3)。

数据链路地址也存储在这个结构中。宏LLADDR将一个指向sockaddr_dl结构的指针转换成一个指向这个文本名称的第一个字节的指针。 sdl_alen是硬件地址的长度。对于一个以太网设备，48 bit硬件地址位于结构sockaddr_dl的这个文本名称的前面。图3-38所示的是一个初始化了的sockaddr_dl结构。

Net/3不使用sdl_slen。

if_attach更新两个全局变量。第一个是if_index，它存放系统中的最后一个接口的索引；第二个是ifnet_addrs，它指向一个ifaddr指针的数组。这个数组的每项都指向一个接口的链路层地址。这个数组提供对系统中每个接口的链路层地址的快速访问。

函数if_attach较长，并且有几个奇怪的赋值语句。从图 3-34开始，我们分5个部分讨论这个函数。

59-74 if_attach有一个参数：ifp，这是一个指向ifnet结构的指针，由网络设备驱动程序初始化。Net/3在一个链表中维护所有这些ifnet结构，全局指针ifnet指向这个链表的首部。while循环查找链表的尾部，并将链表尾部的空指针的地址存储到 P中。在循环后，新ifnet结构被接到这个 ifnet链表的尾部， if_index加1，并且将新索引值赋给 ifp ->if_index。

if.c

```
59 void
60 if_attach(ifp)
61 struct ifnet *ifp;
62 {
63     unsigned socksize, ifasize;
64     int     namelen, unitlen, masklen, ether_output();
65     char    workbuf[12], *unitname;
66     struct ifnet **p = &ifnet;  /* head of interface list */
67     struct sockaddr_dl *sdl;
68     struct ifaddr *ifa;
69     static int if_indexlim = 8; /* size of ifnet_addrs array */
70     extern void link_rtrequest();

71     while (*p)                  /* find end of interface list */
72         p = &((*p)->if_next);
73     *p = ifp;
74     ifp->if_index = ++if_index; /* assign next index */

75     /* resize ifnet_addrs array if necessary */
76     if (ifnet_addrs == 0 || if_index >= if_indexlim) {
77         unsigned n = (if_indexlim <<= 1) * sizeof(ifa);
78         struct ifaddr **q = (struct ifaddr **)
79                     malloc(n, M_IFADDR, M_WAITOK);

80         if (ifnet_addrs) {
81             bcopy((caddr_t) ifnet_addrs, (caddr_t) q, n / 2);
82             free((caddr_t) ifnet_addrs, M_IFADDR);
83         }
84         ifnet_addrs = q;
85     }
```

if.c

图3-34 函数if_attach ：分配接口索引

1. 必要时调整ifnet_addrs数组的大小

75-85 第一次调用if_attach时，数组ifnet_addrs不存在，因此要分配16(16=8<<1)项的空间。当数组满时，一个两倍大的新数组被分配，并且老数组中的项被复制到新的数组中。

if_indexlim是if_attach私有的一个静态变量。if_indexlim通过<<=操作符来更新。

图3-34中的函数malloc和free不是同名的标准C库函数。内核版的第二个参数指明一个

类型，内核中可选的诊断代码用它来检测程序错误。如果malloc的第三个参数为M_WAITOK，且函数需要等待释放的可用内存，则阻塞调用进程。如果第三个参数为 M_DONTWAIT，则当内存不可用时，函数不阻塞并返回一个空指针。

函数if_attach的下一部分显示在图3-35中，它为接口准备一个文本名称并计算链路层地址的长度。

if.c

```
86      /* create a Link Level name for this device */
87      unitname = sprint_d((u_int) ifp->if_unit, workbuf, sizeof(workbuf));
88      namelen = strlen(ifp->if_name);
89      unitlen = strlen(unitname);

90      /* compute size of sockaddr_dl structure for this device */
91  #define _offsetof(t, m) ((int)((caddr_t)&((t *)0)->m))
92      masklen = _offsetof(struct sockaddr_dl, sdl_data[0]) +
93                unitlen + namelen;
94      socksize = masklen + ifp->if_addrlen;
95  #define ROUNDUP(a) (1 + (((a) - 1) | (sizeof(long) - 1)))
96      socksize = ROUNDUP(socksize);
97      if (socksize < sizeof(*sdl))
98          socksize = sizeof(*sdl);
99      ifasize = sizeof(*ifa) + 2 * socksize;
```

if.c

图3-35 if_attach 函数：计算链路层地址大小

2. 创建链路层名称并计算链路层地址的长度

86-99 if_attach用if_unit和if_name组装接口的名称。函数sprint_d将if_unit的数值转换成一个串并存储到workbuf中。masklen是sockaddr_dl数组中sdl_data前面的信息所占用的字节数加上这个接口的文本名称的大小(namelen + unitlen)。ROUNDUP函数对socksize进行上舍入，socksize是masklen加上硬件地址长度(if_addrlen)，上舍入为一个长整型(ROUNDUP)。如果它小于一个sockaddr_dl结构的长度，就使用标准的sockaddr_dl结构。ifasize是一个ifaddr结构的大小加上两倍的 socksize，因此，它能容纳结构sockaddr_dl。

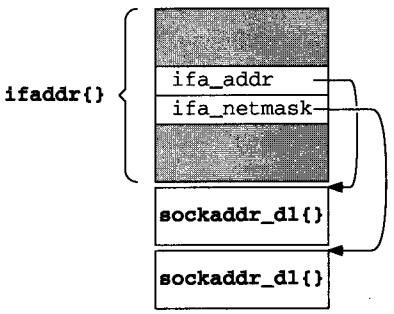


图3-36 在if_attach 中分配的链路层地址和掩码

在函数的下一个部分中，if_attach分配结构并将结构连接起来，如图3-36所示。

图3-36中，在ifaddr结构与两个sockaddr_dl结构间有一个空隙来说明它们分配在一个连续的内存中但没有定义在一个C结构中。

像图3-36所示的组织还出现在结构in_ifaddr中；这个结构的通用ifaddr部分中的指针指向在这个结构的设备专用部分中的专用化sockaddr结构，在本例中是结构sockaddr_dl。图3-37所示的是这些结构的初始化。

3. 地址

100-116 如果有足够的内存可用，bzero把新结构清零，并且sdl指向紧接着ifnet结构的第一个sockaddr_dl。若没有可用内存，代码被忽略。

sdl_len被设置为结构sockaddr_dl的长度，并且sdl_family被设置为AF_LINK。

用if_name和unitname组成的文本名称存放在 sdl_data中，而它的长度存放在sdl_nlen中。接口的索引被复制到sdl_index中，而接口的类型被复制到sdl_type中。分配的结构被插入到数组ifnet_addrs中，并通过ifa_ifp和ifa_addrlist链接到结构ifnet。最后，结构sockaddr_dl用ifa_addr连接到ifnet结构。以太网接口用arp_rtrequest取代默认函数link_rtrequest。环回接口装入函数loop_rtrequest。我们在第19章和第21章讨论ifa_rtrequest和arp_rtrequest。而linkrtrequest和loop_rtrequest留给读者自己去研究。以上完成了第一个sockaddr_dl结构的初始化。

if.c

```
100     if (ifa = (struct ifaddr *) malloc(ifasize, M_IFADDR, M_WAITOK)) {
101         bzero((caddr_t) ifa, ifasize);
102         /* First: initialize the sockaddr_dl address */
103         sdl = (struct sockaddr_dl *) (ifa + 1);
104         sdl->sdl_len = socksize;
105         sdl->sdl_family = AF_LINK;
106         bcopy(ifp->if_name, sdl->sdl_data, namelen);
107         bcopy(unitname, namelen + (caddr_t) sdl->sdl_data, unitlen);
108         sdl->sdl_nlen = (namelen += unitlen);
109         sdl->sdl_index = ifp->if_index;
110         sdl->sdl_type = ifp->if_type;
111         ifnet_addrs[if_index - 1] = ifa;
112         ifa->ifa_ifp = ifp;
113         ifa->ifa_next = ifp->if_addrlist;
114         ifa->ifa_rtrequest = link_rtrequest;
115         ifp->if_addrlist = ifa;
116         ifa->ifa_addr = (struct sockaddr *) sdl;
117         /* Second: initialize the sockaddr_dl mask */
118         sdl = (struct sockaddr_dl *) (socksize + (caddr_t) sdl);
119         ifa->ifa_netmask = (struct sockaddr *) sdl;
120         sdl->sdl_len = masklen;
121         while (namelen != 0)
122             sdl->sdl_data[--namelen] = 0xff;
123     }
```

if.c

图3-37 函数if_attach ：分配并初始化链路层地址

4. 掩码

117-123 第二个sockaddr_dl结构是一个比特掩码，用来选择出现在第一个结构中的文本名称。ifa_netmask从结构ifaddr指向掩码结构(在这里是选择接口文本名称而不是网络掩码)。while循环把与名称对应的那些字节的每个比特都置为1。

图3-38所示的是我们以太网接口例子的两个初始化了的 sockaddr_dl结构。它的if_name为“ le ”，if_unit为0，if_index为1。

图3-38中所示的是ether_ifattach对这个结构初始化后的地址(图3-41)。

图3-39所示的是第一个接口被if_attach连接后的结构。

在if_attach的最后，以太网设备的函数ether_ifattach被调用，如图3-40所示。

124-127 开始不调用ether_ifattach (例如：从leattach)，是因为它要把以太网硬件地址复制到if_attach分配的sockaddr_dl中。

XXX注释表示作者发现在此处插入代码比修改所有的以太网驱动程序要容易。

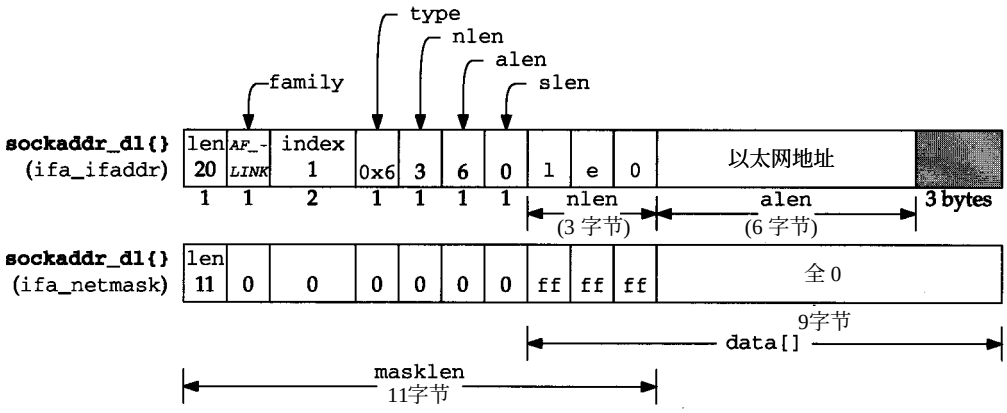


图3-38 初始化了的以太网 sockaddr_dl 结构(省略了前缀 sdl_)

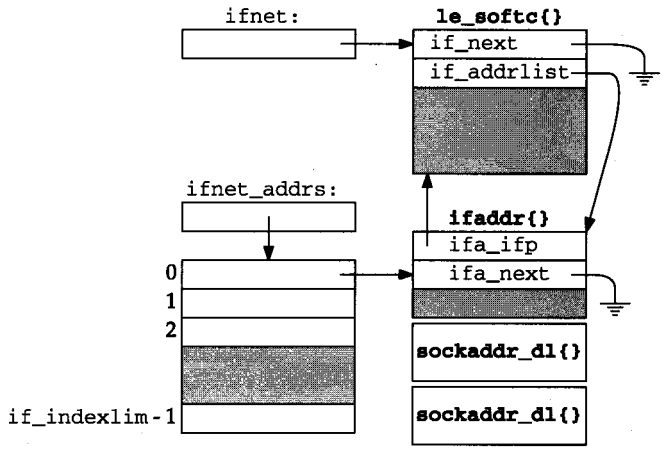


图3-39 第一次调用 if_attach 后的 ifnet 和 sockaddr_dl 结构

if.c

```
124     /* XXX -- Temporary fix before changing 10 ethernet drivers */
125     if (ifp->if_output == ether_output)
126         ether_ifattach(ifp);
127 }
```

if.c

图3-40 函数 if_attach ：以太网初始化

if_ethersubr.c

```
338 void
339 ether_ifattach(ifp)
340 struct ifnet *ifp;
341 {
342     struct ifaddr *ifa;
343     struct sockaddr_dl *sdl;

344     ifp->if_type = IFT_ETHER;
345     ifp->if_addrlen = 6;
346     ifp->if_hdrlen = 14;
347     ifp->if_mtu = ETHERMTU;
348     for (ifa = ifp->if_addrlist; ifa; ifa = ifa->ifa_next)
349         if ((sdl = (struct sockaddr_dl *) ifa->ifa_addr) &&
```

图3-41 函数 ether_ifattach

```
350             sdl->sdl_family == AF_LINK) {
351             sdl->sdl_type = IFT_ETHER;
352             sdl->sdl_alen = ifp->if_addrlen;
353             bcopy((caddr_t) ((struct arpcom *) ifp)->ac_enaddr,
354                   LLADDR(sdl), ifp->if_addrlen);
355             break;
356         }
357 }
```

if_ethersubr.c

图3-41 (续)

5. ether_ifattach函数

函数ether_ifattach执行对所有以太网设备通用的ifnet结构的初始化。

338-357 对于一个以太网设备，if_type为IFT_ETHER，硬件地址为6字节长，整个以太网首部有14字节，而以太网MTU为1500 (ETHERMTU)。

leattach已经指派了MTU，但其他以太网设备驱动程序可能没有执行这个初始化。

4.3节讨论以太网帧组织的更多细节。for循环定位接口的链路层地址，然后初始化结构sockaddr_dl中的以太网硬件地址信息。在系统初始化时，以太网地址被复制到结构arpcom中，现在被复制到链路层地址中。

3.12 ifinit函数

接口结构被初始化并链接到一起后，main (图3-23)调用ifinit，如图3-42所示。

if.c

```
43 void
44 ifinit()
45 {
46     struct ifnet *ifp;

47     for (ifp = ifnet; ifp; ifp = ifp->if_next)
48         if (ifp->if_snd.ifq_maxlen == 0)
49             ifp->if_snd.ifq_maxlen = ifqmaxlen;     /* set default length */
50     if_slowtimo(0);
51 }
```

if.c

图3-42 函数ifinit

43-51 for循环遍历接口列表，并把没有被接口的attach函数设置的每个接口输出队列的最大长度设置为50 (ifqmaxlen)。

输出队列的大小关键要考虑的是发送最大长度数据报的分组的个数。例如以太网，若一个进程调用sendto发送65 507字节的数据，它被分片为45个数据报片，并且每个数据报片被放进接口的输出队列。若队列非常小，由于队列没有空间，进程可能不能发送大的数据报。

if_slowtimo启动接口的监视计时器。当一个接口时钟到期，内核会调用这个接口的把关定时器函数。一个接口可以提前重设时钟来阻止把关定时器函数的调用，或者，若不需要把关定时器函数，则可以把if_timer设置为0。图3-43所示的是函数if_slowtimo。

338-343 if_slowtimo函数有一个参数arg没有使用，但慢超时函数的原型(7.4节)要求有这个参数。

if.c

```
338 void
339 if_slowtimo(arg)
340 void   *arg;
341 {
342     struct ifnet *ifp;
343     int     s = splimp();

344     for (ifp = ifnet; ifp; ifp = ifp->if_next) {
345         if (ifp->if_timer == 0 || --ifp->if_timer)
346             continue;
347         if (ifp->if_watchdog)
348             (*ifp->if_watchdog) (ifp->if_unit);
349     }
350     splx(s);
351     timeout(if_slowtimo, (void *) 0, hz / IFNET_SLOWHZ);
352 }
```

if.c

图3-43 函数if_slowtimo

344-352 if_slowtimo忽略if_timer为0的接口；若if_timer不等于0，if_slowtimo把if_timer减1，并在这个时钟到达0时调用这个接口关联的if_watchdog函数。在调用if_slowtimo时，分组处理进程被splimp阻塞。返回前，ip_slowtimo调用timeout，来以hz/IFNET_SLOWHZ时钟频率调度对它自己的调用。hz是1秒钟内时钟滴答数(通常是100)。它在系统初始化时设置，并保持不变。因为IFNET_SLOWHZ被定义为1，因此内核每赫兹调用一次if_slowtimo，即每秒一次。

函数timeout调度的函数被内核的函数callout回调。详见[Leffler er al. 1989]。

3.13 小结

在本章我们研究了结构ifnet和ifaddr，它们被分配给在系统初始化时发现的每一个网络接口。结构ifnet链接成ifnet链表。每个接口的链路层地址被初始化，并被加到ifnet结构的地址链表中，还存放到数组if_addrs中。

我们讨论了通用sockaddr结构及其成员sa_family和sa_len，它们标识每个地址的类型和长度。我们还查看了一个链路层地址的sockaddr_dl结构的初始化。

在本章中，我们还介绍了在全书中要用到的三个网络接口例子。

习题

3.1 很多Unix系统中的netstat程序列出网络接口及其配置信息。在你接触的系统中试一下命令netstat -i。那个网络接口的名称(if_name)是什么？传输单元的最大长度(if_mtu)是多少？

3.2 在if_slowtimo (图3-43)中，调用splimp和splx出现在循环的外面。与把这些调用放到循环内部相比，这样安排有何优缺点？

3.3 为什么SLIP的交互队列比它的标准输出队列要短？

3.4 为什么if_hdrlen和if_addrlen不在slattach中初始化？

3.5 为SLIP和环回设备画一个与图3-38类似的图。

第4章　接口：以太网

4.1　引言

在第3章中，我们讨论了所有接口要用到的数据结构及对这些数据结构的初始化。在本章中，我们说明以太网设备驱动程序在初始化后是如何接收和传输帧的。本章的后半部分介绍配置网络设备的通用`ioctl`命令。第5章是SLIP和环回驱动程序。

我们不准备查看整个以太网驱动程序的源代码，因为它有大约 1 000行C代码(其中有一半是一个特定接口卡的硬件细节)，但要研究与设备无关的以太网代码部分，及驱动程序是如何与内核其他部分交互的。

如果读者对一个驱动程序的源代码感兴趣， Net/3版本包括很多不同接口的源代码。要想研究接口的技术规范，就要求能理解设备专用的命令。图 4-1所示的是Net/3提供的各种驱动程序，包括在本章我们要讨论的LANCE驱动程序。

网络设备驱动程序通过`ifnet`结构(图3-6)中的7个函数指针来访问。图 4-2列出了指向我们的三个例子驱动程序的入口点。

输入函数不包含在图4-2中，因为它们是网络设备中断驱动的。中断服务例程的配置与硬件相关，并且超出了本书的范围。我们要识别处理设备中断的函数，但不是这些函数被调用的机制。

设　备	文　件
DEC DEUNA接口	`vax/if/if_de.c`
3Com以太网接口	`vax/if/if_ec.c`
Excelan EXOS 204接口	`vax/if/if_ex.c`
Interlan以太网通信控制器	`vax/if/if_il.c`
Interlan NP100以太网通信控制器	`vax/if/if_ix.c`
Digital Q-BUS to NI适配器	`vax/if/if_qe.c`
CMC ENP-20以太网控制器	`tahoe/if/if_enp.c`
Excelan EXOS 202 (VME) & 203 (QBUS)	`tahoe/if/if_ex.c`
ACC VERSAbus以太网控制器	`tahoe/if/if_ace.c`
AMD 7990 LANCE接口	**`hp300/dev/if_le.c`**
NE2000以太网	`i386/isa/if_ne.c`
Western Digital 8003以太网适配器	`i386/isa/if_we.c`

图4-1　Net/3中可用的以太网驱动程序

ifnet	以　太　网	SLIP	环　回	说　明
`if_init`	`leinit`			硬件初始化
`if_output`	`ether_output`	`slouput`	`looutput`	接收并对传输的帧进行排队
`if_start`	`lestart`			开始传输帧
`if_done`				输出完成 (未用)
`if_ioctl`	`leioctl`	`slioctl`	`lcioctl`	处理来自一个进程的`ioctl`命令
`if_reset`	`lereset`			把设备复位到已知的状态
`if_watchdog`				监视设备故障或收集统计信息

图4-2　例子驱动程序的接口函数

只有函数if_output和if_ioctl被经常地调用。而if_init、if_done和if_reset从来不被调用或仅从设备专用代码调用（例如：leinit直接被leioctl调用）。函数if_start仅被函数ether_output调用。

4.2　代码介绍

以太网设备驱动程序和通用接口ioctl的代码包含在两个头文件和三个C文件中，它们列于图4-3中。

文　件	说　明
net/if_ether.h	以太网结构
net/if.h	ioctl命令定义
net/if_ethersubr.c	通用以太网函数
hp300/dev/if_le.c	LANCE以太网驱动程序
net/if.c	ioctl处理

图4-3　在本章讨论的文件

4.2.1　全局变量

显示在图4-4中的全局变量包括协议输入队列、LANCE接口结构和以太网广播地址。

变　量	数据类型	说　明
arpintrq	struct ifqueue	ARP输入队列
clnlintrq	struct ifqueue	CLNP输入队列
ipintrq	struct ifqueue	IP输入队列
le_softc	struct le_softc[]	LANCE以太网接口
etherbraodcastaddr	u_char[]	以太网广播地址

图4-4　本章介绍的全局变量

le_softc是一个数组，因为这里可以有多个以太网接口。

4.2.2　统计量

结构ifnet中为每个接口收集的统计量如图4-5所示。

图4-6显示了netstat命令的一些输出例子，包括ifnet结构中的一些统计信息。

第1列包含显示为一个字符串的if_name和if_unit。若接口是关闭的（不设置IFF_UP），一个星号显示在这个名字的旁边。在图4-6中，sl0、sl2和sl3是关闭的。

第2列显示的是if_mtu。在表头“Network”和“Address”底下的输出依赖于地址的类型。对于链路层地址，显示了结构sockaddr_dl的sdl_data的内容。对于IP地址，显示了子网和单播地址。其余的列是if_ipackets、if_ierrors、if_opackets、if_oerrors和if_collisions。

- 在输出中冲突的分组大约有3% (942 798 / 23 234 729 = 3%)。
- 这个机器的SLIP输出队列从未满过，因为SLIP接口的输出没有差错。
- 在传输中，LANCE硬件检测到12个以太网的输出差错。其中有些差错可能被视为冲突。

ifnet成员	说　明	用于SNMP
if_collisions	在CSMA接口的冲突数	
if_ibytes	接收到的字节总数	•
if_ierrors	接收到的有输入差错分组数	•
if_imcasts	接收到的多播分组数	•
if_ipackets	在接口接收到的分组数	•
if_iqdrops	被此接口丢失的输入分组数	•
if_lastchange	上一次改变统计的时间	•
if_noproto	指定为不支持协议的分组数	•
if_obytes	发送的字节总数	•
if_oerrors	接口上输出的差错数	•
if_omcasts	发送的多播分组数	•
if_opackets	接口上发送的分组数	•
if_snd.ifq_drops	在输出期间丢失的分组数	•
if_snd.ifq_len	输出队列中的分组数	

图4-5　结构ifnet 中维护的统计

netstat -i output

Name	Mtu	Network	Address	Ipkts	Ierrs	Opkts	Oerrs	Coll
le0	1500	<Link>8.0.9.13.d.33		28680519	814	29234729	12	942798
le0	1500	128.32.33	128.32.33.5	28680519	814	29234729	12	942798
sl0*	296	<Link>		54036	0	45402	0	0
sl0*	296	128.32.33	128.32.33.5	54036	0	45402	0	0
sl1	296	<Link>		40397	0	33544	0	0
sl1	296	128.32.33	128.32.33.5	40397	0	33544	0	0
sl2*	296	<Link>		0	0	0	0	0
sl3*	296	<Link>		0	0	0	0	0
lo0	1536	<Link>		493599	0	493599	0	0
lo0	1536	127	127.0.0.1	493599	0	493599	0	0

图4-6　接口统计的样本

• 硬件检测出814个以太网的输入差错，例如分组太短或错误的检验和。

4.2.3　SNMP变量

图4-7所示的是SNMP接口表(ifTable)中的一个接口项对象(ifEntry)，它包含在每个接口的ifnet结构中。

ISODE SNMP代理从if_type获得ifSpeed，并为ifAdminStatus维护一个内部变量。代理的ifLastChange基于结构ifnet中的if_lastchange，但与代理的启动时间相关，而不是与系统的启动时间相关。代理为ifSpecific返回一个空变量。

接口表，索引=<*ifIndex*>		
SNMP变量	ifnet成员	说　明
ifIndex	if_index	唯一地标识接口
ifDescr	if_name	接口的文本名称
ifType	if_type	接口的类型(例如以太网、SLIP等等)

图4-7　接口表ifTable 的变量

接口表，索引=<*ifIndex*>		
SNMP变量	ifnet成员	说 明
ifMtu	if_mtu	接口的MTU(字节)
ifSpeed	(看正文)	接口的正常速率(每秒比特)
ifPhysAddress	ac_enaddr	媒体地址(来自结构arpcom)
ifAdminStatus	(看正文)	接口的期望状态(IFF_UP标志)
ifOperStatus	if_flags	接口的操作状态(IFF_UP标志)
ifLastChange	(看正文)	上一次统计改变时间
ifInOctets	if_ibytes	输入的字节总数
ifInUcastPkts	if_ipackets -if_imcasts	输入的单播分组数
ifInNUcastPkts	if_imcasts	输入的广播或多播分组数
ifInDiscards	if_iqdrops	因为实现的限制而丢弃的分组数
ifInErrors	if_ierrors	差错的分组数
ifInUnknownProtos	if_noproto	指定为未知协议的分组数
ifOutOctets	if_obytes	输出字节数
ifOutUcastPkts	if_opackets-if_omcasts	输出的单播分组数
ifOutNUcastPkts	if_omcasts	输出的广播或多播分组数
ifOutDiscards	if_snd.ifq_drops	因为实现的限制而丢失的输出分组数
ifOutErrors	if_oerrors	因为差错而丢失的输出分组数
ifOutQLen	if_snd.ifq_len	输出队列长度
ifSpecific	n/a	媒体专用信息的SNMP对象ID(未实现)

图4-7 (续)

4.3 以太网接口

Net/3以太网设备驱动程序都遵循同样的设计。对于大多数 Unix设备驱动程序来说，都是这样，因为写一个新接口卡的驱动程序总是在一个已有的驱动程序的基础上修改而来的。在本节，我们简要地概述一下以太网的标准和一个以太网驱动程序的设计。我们用 LANCE驱动程序来说明这个设计。

图4-8说明了一个IP分组的以太网封装。

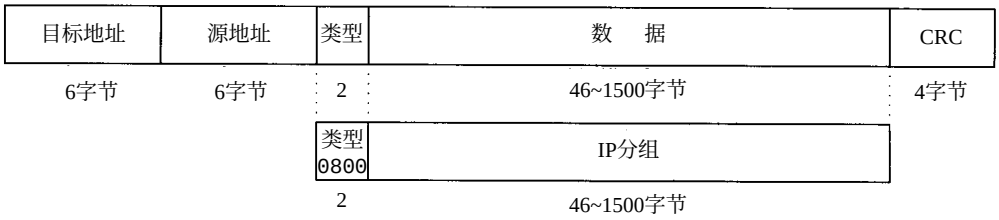


图4-8 一个IP分组的以太网封装

以太网帧包括48 bit的目标地址和源地址，接下来是一个 16 bit的类型字段，它标识这个帧所携带的数据的格式。对于 IP分组，类型是0x0800(2048)。帧的最后是一个32 bit的CRC(循环冗余检验)，它用来检查帧中的差错。

我们所讨论的最初的以太网组帧的标准在1982年由Digital设备公司、Intel公司及施乐公司发布，并作为今天在TCP/IP网络中最常用的格式。另一个可选的格式是IEEE(电气电子工程师协会)规定的802.2和802.3标准。更多的IEEE标准详见[Stallings 1987]。

对于以太网，IP分组的封装由RFC 894[Hornig 1984]规定，而对于802.3网，却由RFC1042[Postel和Reynolds 1988]规定。

我们用48 bit的以太网地址作为硬件地址。IP地址到硬件地址之间的转换用ARP协议(RFC 826 [Plummer 1982])，这个协议在第21章讨论。而硬件地址到IP地址的转换用RARP协议(RFC 903 [Finlayson et al. 1984])。以太网地址有两种类型：单播和多播。一个单播地址描述一个单一的以太网接口，而一个多播地址描述一组以太网接口。一个以太网广播是一个所有接口都接收的多播。以太网单播地址由设备的厂商分配，也有一些设备的地址允许用软件改变。

一些DECNET协议要求标识一个多接口主机的硬件地址，因此DECNET必须能改变一个设备的以太网单播地址。

图4-9列举了以太网接口的数据结构和函数。

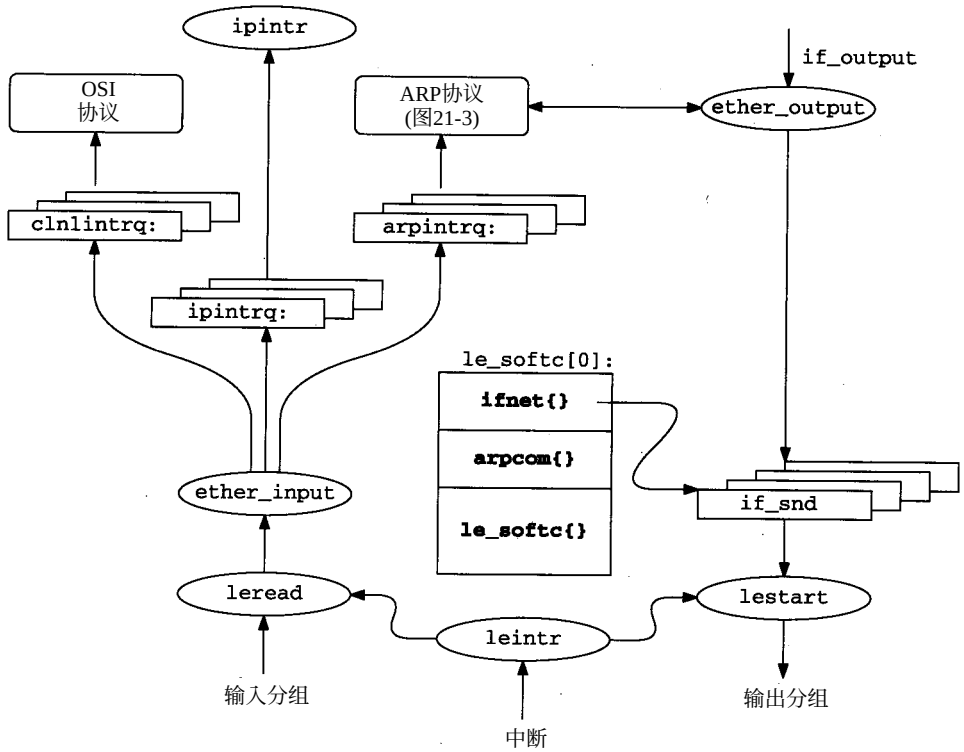


图4-9 以太网设备驱动程序

在图中：用一个椭圆标识一个函数（leintr）、用一个方框标识数据结构(le_softc[0])、le_softc及用圆角方框标识一组函数(ARP协议)。

图4-9左上角显示的是OSI无连接网络层(clnl)协议、IP和ARP的输入队列。对于clnlintrq，我们不打算讲更多，将它包含进来是为了强调ether_input要将以太网帧分用到多个协议队列中。

在技术上，OSI使用无连接网络协议(CLNP而不是CLNL)，但我们使用的是Net/3中的术语。CLNP的官方标准是ISO 8473。[Stallings 1993]对这个标准作了概述。

接口结构le_softc在图4-9的中间。我们感兴趣的是这个结构中的 ifnet和arpcom，其他是LANCE硬件的专用部分。我们在图 3-6中显示了结构ifnet，在图3-26中显示了结构arpcom。

4.3.1 leintr函数

我们从以太网帧的接收开始。现在，假设硬件已初始化并且系统已完成配置，当接口产生一个中断时，leintr被调用。在正常操作中，一个以太网接口接收发送到它的单播地址和以太网广播地址的帧。当一个完整的帧可用时，接口就产生一个中断，并且内核调用 leintr。

在第12章中，我们会看见可能要配置多个以太网接口来接收以太网多播帧 (不同于广播)。

有些接口可以配置为运行在混杂方式。在这种方式下，接口接收所有出现在网络上的帧。在卷1中讨论的tcpdump程序可以使用BPF (BSD分组过滤程序)来利用这种特性。

leintr检测硬件，并且如果有一个帧到达，就调用 leread把这个帧从接口转移到一个mbuf链中(用m_devget)。如果硬件报告一个帧已传输完或发现一个差错 (如一个有错误的检验和)，则leintr更新相应的接口统计，复位这个硬件，并调用 lestart来传输另一个帧。

所有以太网设备驱动程序将它们接收到的帧传给 ether_input做进一步的处理。设备驱动程序构造的 mbuf链不包括以太网首部，以太网首部作为一个独立的参数传递给ether_input。结构ether_header显示在图4-10中。

38-42 以太网CRC并不总是可用。它由接口硬件来计算与检验，接口硬件丢弃到达的 CRC差错帧。以太网设备驱动程序负责 ether_type的网络和主机字节序间的转换。在驱动程序外，它总是主机字节序。

if_ether.h

```
38 struct ether_header {
39     u_char  ether_dhost[6];     /* Ethernet destination address */
40     u_char  ether_shost[6];     /* Ethernet source address */
41     u_short ether_type;         /* Ethernet frame type */
42 };
```

if_ether.h

图4-10 结构ether_header

4.3.2 leread函数

函数leread(图4-11)的开始是由leintr传给它的一个连续的内存缓冲区，并且构造了一个ether_header结构和一个mbuf链。这个链表存储来自以太网帧的数据。 leread还将输入帧传给BPF。

if_le.c

```
528 leread(unit, buf, len)
529 int     unit;
530 char    *buf;
531 int     len;
532 {
533     struct le_softc *le = &le_softc[unit];
```

图4-11 函数leread

```
534     struct ether_header *et;
535     struct mbuf *m;
536     int     off, resid, flags;

537     le->sc_if.if_ipackets++;
538     et = (struct ether_header *) buf;
539     et->ether_type = ntohs((u_short) et->ether_type);
540     /* adjust input length to account for header and CRC */
541     len = len - sizeof(struct ether_header) - 4;
542     off = 0;

543     if (len <= 0) {
544         if (ledebug)
545             log(LOG_WARNING,
546                 "le%d: ierror(runt packet): from %s: len=%d\n",
547                 unit, ether_sprintf(et->ether_shost), len);
548         le->sc_runt++;
549         le->sc_if.if_ierrors++;
550         return;
551     }
552     flags = 0;
553     if (bcmp((caddr_t) etherbroadcastaddr,
554              (caddr_t) et->ether_dhost, sizeof(etherbroadcastaddr)) == 0)
555         flags |= M_BCAST;
556     if (et->ether_dhost[0] & 1)
557         flags |= M_MCAST;

558     /*
559      * Check if there's a bpf filter listening on this interface.
560      * If so, hand off the raw packet to enet.
561      */
562     if (le->sc_if.if_bpf) {
563         bpf_tap(le->sc_if.if_bpf, buf, len + sizeof(struct ether_header));

564         /*
565          * Keep the packet if it's a broadcast or has our
566          * physical ethernet address (or if we support
567          * multicast and it's one).
568          */

569         if ((flags & (M_BCAST | M_MCAST)) == 0 &&
570             bcmp(et->ether_dhost, le->sc_addr,
571                  sizeof(et->ether_dhost)) != 0)
572             return;
573     }
574     /*
575      * Pull packet off interface.  Off is nonzero if packet
576      * has trailing header; m_devget will then force this header
577      * information to be at the front, but we still have to drop
578      * the type and length which are at the front of any trailer data.
579      */
580     m = m_devget((char *) (et + 1), len, off, &le->sc_if, 0);
581     if (m == 0)
582         return;
583     m->m_flags |= flags;
584     ether_input(&le->sc_if, et, m);
585 }
```

if_le.c

图4-11 (续)

528-539 函数leintr给leread传了三个参数：unit，它标识接收到此帧的特定接口

卡；buf，它指向接收到的帧；len，它是帧的字节数(包括首部和CRC)。

函数将et指向这个缓存的开始，并且将以太网字节序转换成主机字节序，来构造结构ether_header。

540-551 将len减去以太网首部和CRC的大小得到数据的字节数。短分组(runt packet)是一个长度太短的非法以太网帧，它被记录、统计，并被丢弃。

552-557 接下来，目标地址被检测，并判断是不是以太网广播或多播地址。以太网广播地址是一个以太网多播地址的特例；它的每一比特都被设置了。etherbroadcastaddr是一个数组，定义如下：

```
u_char etherbroadcastaddr[6] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
```

> 这是C语言中定义一个48 bit值的简便方法。这项技术仅在我们假设字符是 8 bit值时才起作用——ANSI C并不保证这一点。

bcmp比较etherbroadcastaddr和ether_dhost，若相同，则设置标志M_BCAST。一个以太网多播地址由这个地址的首字节的低位比特来标识，如图 4-12所示。

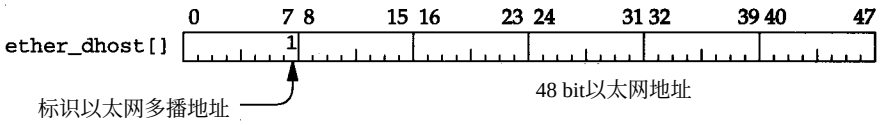


图4-12 检测一个以太网多播地址

在第12章中，我们会看到并不是所有以太网多播帧都是 IP多播数据报，并且IP必须进一步检测这个分组。

如果这个地址的多播比特被置位，在 mbuf首部中设置M_MCAST。检测的顺序是重要的：首先ether_input将整个48 bit地址和以太网广播地址比较，若不同，则检测标识以太网多播地址的首字节的低位比特(习题4.1)。

558-573 如果接口带有BPF，调用bpf_tap把这个帧直接传给BPF。我们会看见对于SLIP和环回接口，要构造一个特定的BPF帧，因为这些网络没有一个链路层首部(不像以太网)。

当一个接口带有BPF时，它可以配置为运行在混淆模式，并且接收网络上出现的所有以太网帧，而不是通常由硬件接收的帧的子集。如果分组发送给一个不与此接口地址匹配的单播地址，则被leread丢弃。

574-585 m_devget (2.6节)将数据从传给leread的缓存中复制到一个它分配的mbuf链中。传给m_devget的第一个参数指向以太网首部后的第一个字节，它是此帧中的第一个数据字节。如果m_devget内存用完，leread立即返回。另外广播和多播标志被设置在链表中的第一个mbuf中，ether_input处理这个分组。

4.3.3 ether_input函数

函数ether_input显示在图4-13中，它检查结构ether_header来判断接收到的数据的类型，并将接收到的分组加入到队列中等待处理。

1. 广播和多播的识别

196-209 传给ether_input的参数有：ifp，一个指向接收此分组的接口的ifnet结构的指针；eh，一个指向接收分组的以太网首部的指针；m，一个指向接收分组的指针(不包括以太网首部)。

任何到达不工作接口的分组将被丢弃。可能没有为接口配置一个协议地址，或者接口可能被程序ifconfig(8) (6.6节)显式地禁用了。

if_ethersubr.c

```
196 void
197 ether_input(ifp, eh, m)
198 struct ifnet *ifp;
199 struct ether_header *eh;
200 struct mbuf *m;
201 {
202     struct ifqueue *inq;
203     struct llc *l;
204     struct arpcom *ac = (struct arpcom *) ifp;
205     int     s;

206     if ((ifp->if_flags & IFF_UP) == 0) {
207         m_freem(m);
208         return;
209     }
210     ifp->if_lastchange = time;
211     ifp->if_ibytes += m->m_pkthdr.len + sizeof(*eh);
212     if (bcmp((caddr_t) etherbroadcastaddr, (caddr_t) eh->ether_dhost,
213              sizeof(etherbroadcastaddr)) == 0)
214         m->m_flags |= M_BCAST;
215     else if (eh->ether_dhost[0] & 1)
216         m->m_flags |= M_MCAST;
217     if (m->m_flags & (M_BCAST | M_MCAST))
218         ifp->if_imcasts++;

219     switch (eh->ether_type) {
220     case ETHERTYPE_IP:
221         schednetisr(NETISR_IP);
222         inq = &ipintrq;
223         break;

224     case ETHERTYPE_ARP:
225         schednetisr(NETISR_ARP);
226         inq = &arpintrq;
227         break;

228     default:
229         if (eh->ether_type > ETHERMTU) {
230             m_freem(m);
231             return;
232         }

                    /* OSI code */

307     }

308     s = splimp();
309     if (IF_QFULL(inq)) {
310         IF_DROP(inq);
311         m_freem(m);
312     } else
313         IF_ENQUEUE(inq, m);
314     splx(s);
315 }
```

if_ethersubr.c

图4-13 函数ether_input

210-218 变量time是一个全局的timeval结构，内核用它维护当前时间和日期，它是从Unix新纪元(1970年1月1日00:00:00，协调通用时间[UTC])开始的秒和微秒数。在[Itano and Ramsey 1993]中可以找到对UTC的简要讨论。我们在Net/3源代码中会经常遇到结构timeval：

```
struct timeval {
  long tv_sec;     /* seconds */
  long tv_usec;    /* and microseconds */
};
```

ether_input用当前时间更新if_lastchange，并且把if_ibytes加上输入分组的长度(分组长度加上14字节的以太网首部)。

然后，ether_input再次用leread去判断分组是否为一个广播或多播分组。

有些内核编译时可能没有包括BPF代码，因此测试必须在ether_input中进行。

2. 链路层分用

219-227 ether_input根据以太网类型字段来跳转。对于一个IP分组，schednetisr调度一个IP软件中断，并选择IP输入队列，ipintrq。对于一个ARP分组，调度ARP软件中断，并选择arpintrq。

一个isr是一个中断服务例程。

在原先的BSD版本中，当处于网络中断级别时，ARP分组通过调用arpinput立即被处理。通过分组排队，它们可以在软件中断级别被处理。

如果要处理其他以太网类型，一个内核程序员应在此增加其他情况的处理。或者，一个进程能用BPF接收其他以太网类型。例如，在Net/3中，RARP服务通常用BPF实现。

228-307 默认情况处理不识别以太网类型或按802.3标准(例如OSI无连接传输)封装的分组。以太网的type字段和802.3的length字段在一个以太网帧中占用同一位置。两种封装能够分辨出来，因为一个以太网封装的类型范围和802.3封装的长度范围是不同的(图4-14)。我们跳过OSI代码，在[Stallings 1993]中有对OSI链路层协议的说明。

范　围	说　明
0~1500	IEEE 802.3 *length*字段
1501~65535	以太网*type*字段：
2048	IP分组
2045	ARP分组

图4-14 以太网的*type*字段和802.3的*length*字段

有很多其他以太网类型值分配给各种协议；我们没有在图4-14中显示。在RFC 1700 [Reynolds and Postel 1994]中有一个有更多通用类型的列表。

3. 分组排队

308-315 最后，ether_input把分组放置到选择的队列中，若队列为空，则丢弃此分组。我们在图7-23和图21-16中会看到IP和ARP队列的默认限制为每个50个(ipqmaxlen)分组。

当ether_input返回时，设备驱动程序通知硬件它已准备接收下一分组，这时下一分组可能已存在于设备中。当schednetisr调度的软件中断发生时，处理分组输入队列(1.12节)。准确地说，调用ipintr来处理IP输入队列中的分组，调用arpintr来处理ARP输入队列中的分组。

4.3.4 ether_output函数

我们现在查看以太网帧的输出，当一个网络层协议，如 IP，调用此接口ifnet结构中指定的函数if_output时，开始处理输出。所有以太网设备的 if_output是ether_output (图4-2)。ether_output用14字节以太网首部封装一个以太网帧的数据部分，并将它放置到接口的发送队列中。这个函数比较大，我们分 4个部分来说明：

- 验证；
- 特定协议处理；
- 构造帧；
- 接口排队。

图4-15包括这个函数的第一个部分。

if_ethersubr.c

```
49 int
50 ether_output(ifp, m0, dst, rt0)
51 struct ifnet *ifp;
52 struct mbuf *m0;
53 struct sockaddr *dst;
54 struct rtentry *rt0;
55 {
56     short   type;
57     int     s, error = 0;
58     u_char  edst[6];
59     struct mbuf *m = m0;
60     struct rtentry *rt;
61     struct mbuf *mcopy = (struct mbuf *) 0;
62     struct ether_header *eh;
63     int     off, len = m->m_pkthdr.len;
64     struct arpcom *ac = (struct arpcom *) ifp;

65     if ((ifp->if_flags & (IFF_UP | IFF_RUNNING)) != (IFF_UP | IFF_RUNNING))
66         senderr(ENETDOWN);
67     ifp->if_lastchange = time;
68     if (rt = rt0) {
69         if ((rt->rt_flags & RTF_UP) == 0) {
70             if (rt0 = rt = rtalloc1(dst, 1))
71                 rt->rt_refcnt--;
72             else
73                 senderr(EHOSTUNREACH);
74         }
75         if (rt->rt_flags & RTF_GATEWAY) {
76             if (rt->rt_gwroute == 0)
77                 goto lookup;
78             if (((rt = rt->rt_gwroute)->rt_flags & RTF_UP) == 0) {
79                 rtfree(rt);
80                 rt = rt0;
81     lookup:     rt->rt_gwroute = rtalloc1(rt->rt_gateway, 1);

82                 if ((rt = rt->rt_gwroute) == 0)
83                     senderr(EHOSTUNREACH);
84             }
85         }
86         if (rt->rt_flags & RTF_REJECT)
87             if (rt->rt_rmx.rmx_expire == 0 ||
88                 time.tv_sec < rt->rt_rmx.rmx_expire)
89                 senderr(rt == rt0 ? EHOSTDOWN : EHOSTUNREACH);
90     }
```

if_ethersubr.c

图4-15 函数ether_output ：验证

49-64 ether_output的参数有：ifp，它指向输出接口的ifnet结构；m0，要发送的分组；dst，分组的目标地址；rt0，路由信息。

65-67 在ether_output中多次调用宏senderr。

```
#define senderr(e) { error = (e); goto bad;}
```

senderr保存差错码，并跳到函数的尾部 bad，在那里分组被丢弃，并且 ether_output返回error。

如果接口启动并在运行， ether_output更新接口的上次更改时间。否则，返回ENETDOWN。

1. 主机路由

68-74 rt0指向ip_output找到的路由项，并传递给 ether_output。如果从BPF调用ether_output，rt0可以为空。在这种情况下，控制转给图 4-16中的代码。否则，验证路由。如果路由无效，参考路由表，并且当路由不能被找到时，返回 EHOSTUNREACH。这时，rt0和rt指向一个到下一跳目的地的有效路由。

if_ethersubr.c

```
 91      switch (dst->sa_family) {

 92      case AF_INET:
 93          if (!arpresolve(ac, rt, m, dst, edst))
 94              return (0);         /* if not yet resolved */
 95          /* If broadcasting on a simplex interface, loopback a copy */
 96          if ((m->m_flags & M_BCAST) && (ifp->if_flags & IFF_SIMPLEX))
 97              mcopy = m_copy(m, 0, (int) M_COPYALL);
 98          off = m->m_pkthdr.len - m->m_len;
 99          type = ETHERTYPE_IP;
100          break;
101      case AF_ISO:
                                    /* OSI code */
142      case AF_UNSPEC:
143          eh = (struct ether_header *) dst->sa_data;
144          bcopy((caddr_t) eh->ether_dhost, (caddr_t) edst, sizeof(edst));
145          type = eh->ether_type;
146          break;

147      default:
148          printf("%s%d: can't handle af%d\n", ifp->if_name, ifp->if_unit,
149                 dst->sa_family);
150          senderr(EAFNOSUPPORT);
151      }
```

if_ethersubr.c

图4-16 函数ether_output ：网络协议处理

2. 网关路由

75-85 如果分组的下一跳是一个网关 (而不是最终目的)，找到一个到此网关的路由，并且rt指向它。如果不能发现一个网关路由，则返回 EHOSTUNREACH。这时， rt指向下一跳目的地的路由。下一跳可能是一个网关或最终目标地址。

3. 避免ARP泛洪

86-90 当目标方不准备响应 ARP请求时， ARP代码设置标志RTF_REJECT来丢弃到达目标

方的分组。这在图21-24中描述。

ether_output根据此分组的目标地址继续处理。因为以太网设备仅响应以太网地址，要发送一个分组，ether_output必须发现下一跳目的地的IP地址所对应的以太网地址。ARP协议(第21章)用来实现这个转换。图4-16显示了驱动程序是如何访问ARP协议的。

4. IP输出

91-101 ether_output根据目标地址中的sa_family进行跳转。我们在图4-16中仅显示了case为AF_INET、AF_ISO和AF_UNSPEC的代码，而略过了case AF_ISO的代码。

case AF_INET调用arpresolve来决定与目标IP地址相对应的以太网地址。如果以太网地址已存在于ARP高速缓存中，则arpresolve返回1，并且ether_output继续执行。否则，这个IP分组由ARP控制，并且ARP判断地址，从函数in_arpinput调用ether_output。

假设ARP高速缓存包含硬件地址，ether_output检查是否分组要广播，并且接口是否是单向的(例如，它不能接收自己发送的分组)。如果都成立，则m_copy复制这个分组。在执行switch后，这个复制的分组同到达以太网接口的分组一样进行排队。这是广播定义的要求，发送主机必须接收这个分组的一个备份。

我们在第12章会看到多播分组可能会环回到输出接口而被接收。

5. 显式以太网输出

142-146 有些协议，如ARP，需要显式地指定以太网目的地和类型。地址族类常量AF_UNSPEC指出：dst指向一个以太网首部。bcopy复制edst中的目标地址，并把以太网类型设为type。它不必调用arpresolve(如AF_INET)，因为以太网目标地址已由调用者显式地提供了。

6. 未识别的地址族类

147-151 未识别的地址族类产生一个控制台消息，并且ether_output返回EAFNOSUPPORT。

图4-17所示的是ether_output的下一部分：构造以太网帧。

if_ethersubr.c

```
152     if (mcopy)
153         (void) looutput(ifp, mcopy, dst, rt);
154     /*
155      * Add local net header.  If no space in first mbuf,
156      * allocate another.
157      */
158     M_PREPEND(m, sizeof(struct ether_header), M_DONTWAIT);
159     if (m == 0)
160         senderr(ENOBUFS);
161     eh = mtod(m, struct ether_header *);
162     type = htons((u_short) type);
163     bcopy((caddr_t) &type, (caddr_t) &eh->ether_type,
164         sizeof(eh->ether_type));
165     bcopy((caddr_t)edst, (caddr_t)eh->ether_dhost, sizeof (edst));
166     bcopy((caddr_t)ac->ac_enaddr, (caddr_t)eh->ether_shost,
167         sizeof(eh->ether_shost));
```

if_ethersubr.c

图4-17 函数ether_output：构造以太网帧

7. 以太网首部

152-167 如果在switch中的代码复制了这个分组，这个分组副本同在输出接口上接收到

的分组一样通过调用looutput来处理。环回接口和looutput在5.4节讨论。

M_PREPEND确保在分组的前面有14字节的空间。

大多数协议要在mbuf链表的前面留一些空间，因此，M_PREPEND仅需要调整一些指针(例如，16.7节中UDP输出的sosend和13.6节的igmp_sendreport)。

ether_output用type、edst和ac_enaddr(图3-26)构成以太网首部。ac_enaddr是与此输出接口关联的以太网单播地址，并且是所有从此接口传输的帧的源地址。ether_header用ac_enaddr重写调用者可能在ether_header结构中指定的源地址。这使得伪造一个以太网帧的源地址变得更难。

这时，mbuf包含一个除32 bit CRC以外的完整以太网帧，CRC由以太网硬件在传输时计算。图4-18所示的代码对设备要传送的帧进行排队。

if_ethersubr.c

```
168     s = splimp();
169     /*
170      * Queue message on interface, and start output if interface
171      * not yet active.
172      */
173     if (IF_QFULL(&ifp->if_snd)) {
174         IF_DROP(&ifp->if_snd);
175         splx(s);
176         senderr(ENOBUFS);
177     }
178     IF_ENQUEUE(&ifp->if_snd, m);
179     if ((ifp->if_flags & IFF_OACTIVE) == 0)
180         (*ifp->if_start) (ifp);
181     splx(s);
182     ifp->if_obytes += len + sizeof(struct ether_header);
183     if (m->m_flags & M_MCAST)
184         ifp->if_omcasts++;
185     return (error);

186   bad:
187     if (m)
188         m_freem(m);
189     return (error);
190 }
```

if_ethersubr.c

图4-18 函数ether_output：输出排队

168-185 如果输出队列为空，ether_output丢弃此帧，并返回ENOBUFS。如果输出队列不为空，这个帧放置到接口的发送队列中，并且若接口未激活，接口的if_start函数传输下一帧。

186-190 宏senderr跳到bad，在这里帧被丢弃，并返回一个差错码。

4.3.5 lestart函数

函数lestart从接口输出队列中取出排队的帧，并交给LANCE以太网卡发送。如果设备空闲，调用此函数开始发送帧。ether_output(图4-18)的最后是一个例子，直接通过接口的if_start函数调用lestart。

如果设备忙，当它完成了当前帧的传输时产生一个中断。设备调用lestart来退队并传输下一帧。一旦开始，协议层不再用调用lestart来排队帧，因为驱动程序不断退队并传输

帧，直到队列为空为止。

图4-19所示的是函数lestart。lestart假设已调用splimp来阻塞所有设备中断。

if_le.c

```
325 lestart(ifp)
326 struct ifnet *ifp;
327 {
328     struct le_softc *le = &le_softc[ifp->if_unit];
329     struct letmd *tmd;
330     struct mbuf *m;
331     int      len;

332     if ((le->sc_if.if_flags & IFF_RUNNING) == 0)
333         return (0);

                /* device-specific code */

335     do {

                /* device-specific code */

340         IF_DEQUEUE(&le->sc_if.if_snd, m);
341         if (m == 0)
342             return (0);
343         len = leput(le->sc_r2->ler2_tbuf[le->sc_tmd], m);
344         /*
345          * If bpf is listening on this interface, let it
346          * see the packet before we commit it to the wire.
347          */
348         if (ifp->if_bpf)
349             bpf_tap(ifp->if_bpf, le->sc_r2->ler2_tbuf[le->sc_tmd],
350                     len);

                /* device-specific code */

359     } while (++le->sc_txcnt < LETBUF);
360     le->sc_if.if_flags |= IFF_OACTIVE;
361     return (0);
362 }
```

if_le.c

图4-19 函数lestart

1. 接口必须初始化

325-333 如果接口没有初始化，lestart立即返回。

2. 将帧从输出队列中退队

335-342 如果接口已初始化，下一帧从队列中移去。如果接口输出队列为空，则lestart返回。

3. 传输帧并传递给BPF

343-350 leput将m中的帧复制到leput第一个参数所指向的硬件缓存中。如果接口带有BPF，将帧传给bpf_tap。我们跳过硬件缓存中帧传输的设备专用初始化代码。

4. 如果设备准备好，重复发送多帧

359 当le->sc_txcnt等于LETBUF时，lestart停止给设备传送帧。有些以太网接口能排队多个以太网输出帧。对于LANCE驱动器，LETBUF是此驱动器硬件传输缓存的可用个数，并且le->sc_txcnt保持跟踪有多少个缓存被使用。

5. 将设备标记为忙

360-362 最后，lestart在ifnet结构中设置IFF_OACTIVE来标识这个设备忙于传输帧。

在设备中将多个要传输的帧进行排队有一个负面影响。根据 [Jacobson 1998a]，LANCE芯片能够在两个帧间以很小的时延传输排队的帧。不幸的是，有些 (差的)以太网设备会丢失帧，因为它们不能足够快地处理输入的数据。

在一个应用如NFS中，这会很糟糕地互相影响。 NFS发送大的UDP数据报(经常是超过8192字节)，数据报被IP分片，并在LANCE设备中作为多个以太网帧排队。分片在接收方丢失，当NFS重传整个UDP数据报时，会导致很多未完成的数据报极大的时延。

Jacobson提出Sun的LANCE驱动器一次只排队一个帧就可能避免这一问题。

4.4 ioctl系统调用

ioctl系统调用提供一个通用命令接口，一个进程用它来访问一个设备的标准系统调用所不支持的特性。ioctl的原型为：

```
int ioctl(int fd, unsigned long com,…);
```

fd是一个描述符，通常是一个设备或网络连接。每种类型的描述符都支持它自己的一套ioctl命令，这套命令由第二个参数com来指定。第三个参数在原型中显示为“ ...”，因为它是依赖于被调用的ioctl命令的类型的指针。如果命令要取回信息，第三个参数必须是指向一个足够保存数据的缓存的指针。在本书中，我们仅讨论用于插口描述符的 ioctl命令。

我们显示的系统调用的原型是一个进程进行系统调用的原型。在第 15章中我们会看见在内核中的这个函数还有一个不同的原型。

我们在第17章讨论系统调用ioctl的实现，但在本书的各个部分讨论 ioctl单个命令的实现。

我们讨论的第一个ioctl命令提供对讨论过的网络接口结构的访问。我们总结的本书中所有的ioctl命令如图4-20所示。

命 令	第三个参数	函 数	说 明
SIOCGIFCONF	struct ifconf *	ifconf	获取接口配置清单
SIOCGIFFLAGS	struct ifreq *	ifioctl	获得接口标志
SIOCGIFMETRIC	struct ifreq *	ifioctl	获得接口度量
SIOCSIFFLAGS	struct ifreq *	ifioctl	设置接口标志
SIOCSIFMETRIC	struct ifreq *	ifioctl	设置接口度量

图4-20 接口ioctl 的命令

第一列显示的符号常量标识 ioctl命令(第二个参数， com)。第二列显示传递给第一列所显示的命令的系统调用的第三个参数的类型。第三列是实现这个命令的函数的名称。

图4-21显示处理ioctl命令的各种函数的组织。带阴影的函数我们在本章中说明。其余

的函数在其他章说明。

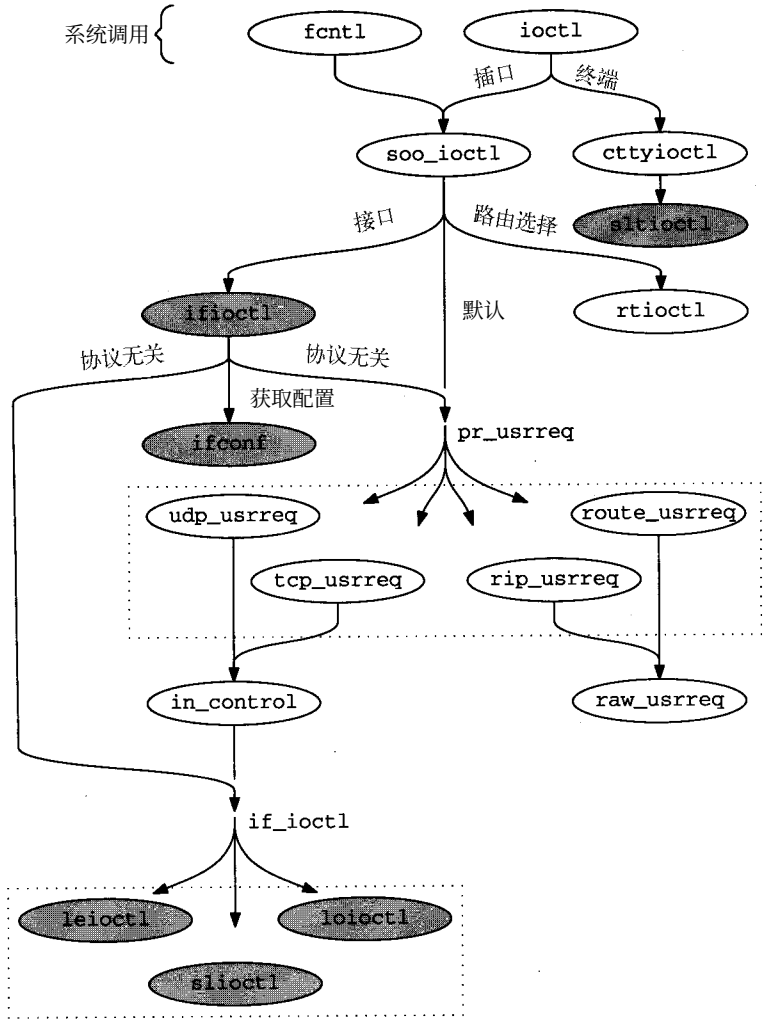


图4-21 在本章说明的ioctl 函数

4.4.1 ifioctl函数

系统调用ioctl将图4-20所列的5种命令传递给图4-22所示的ifioctl函数。

394-405 对于命令SIOCGIFCONF，ifioctl调用ifconf来构造一个可变长ifreq结构的表。

406-410 对于其他ioctl命令，数据参数是指向一个 ifreq结构的指针。 ifunit在ifnet列表中查找名称为进程在ifr->ifr_name中提供的文本名称(例如：“sl0”，“le1”或“lo0”)的接口。如果没有匹配的接口， ifioctl返回ENXIO。剩下的代码依赖于cmd，它们在图4-29中说明。

447-454 如果接口ioctl命令不能被识别，ifioctl把命令发送给与所请求插口关联的协议的用户要求函数。对于IP，这些命令以一个UDP插口发送并调用udp_usrreq。这一类命

令在图6-10中描述。23.10节将详细讨论函数udp_usrreq。

如果控制到达switch语句外，返回0。

if.c

```
394 int
395 ifioctl(so, cmd, data, p)
396 struct socket *so;
397 int     cmd;
398 caddr_t data;
399 struct proc *p;
400 {
401     struct ifnet *ifp;
402     struct ifreq *ifr;
403     int     error;

404     if (cmd == SIOCGIFCONF)
405         return (ifconf(cmd, data));

406     ifr = (struct ifreq *) data;
407     ifp = ifunit(ifr->ifr_name);
408     if (ifp == 0)
409         return (ENXIO);
410     switch (cmd) {

        /* other interface ioctl commands (figures 4.29 and 12.11) */

447     default:
448         if (so->so_proto == 0)
449             return (EOPNOTSUPP);
450         return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
451                                             cmd, data, ifp));
452     }
453     return (0);
454 }
```

if.c

图4-22 函数ifioctl ：综述与SIOCGIFCONF

4.4.2 ifconf函数

ifconf为进程提供一个标准的方法来发现一个系统中的接口和配置的地址。由结构ifreq和ifconf表示的接口信息如图4-23和图4-24所示。

262-279 一个ifreq结构包含在ifr_name中一个接口的名称。在联合中的其他成员被各种ioctl命令访问。通常，用宏来简化对联合的成员的访问语法。

292-300 在结构ifconf中，ifc_len是ifc_buf指向的缓存的字节数。这个缓存由一个进程分配，但由ifconf用一个具有可变长ifreq结构的数组来填充。对于函数ifconf，ifr_addr是结构ifreq中联合的相关成员。每个ifreq结构有一个可变长度，因为ifr_addr(一个sockaddr结构)的长度根据地址的类型而变。必须用结构sockaddr的成员sa_len来定位每项的结束。图4-25说明了ifconf所维护的数据结构。

在图4-25中，左边的数据在内核中，而右边的数据在一个进程中。我们用这个图来讨论图4-26中所示的ifconf函数。

462-474 ifconf的两个参数是：cmd，它被忽略；data，它指向此进程指定的ifconf结构的一个副本。

if.h

```
262 struct  ifreq {
263 #define IFNAMSIZ    16
264     char    ifr_name[IFNAMSIZ];                 /* if name, e.g. "en0" */
265     union {
266         struct  sockaddr ifru_addr;
267         struct  sockaddr ifru_dstaddr;
268         struct  sockaddr ifru_broadaddr;
269         short   ifru_flags;
270         int ifru_metric;
271         caddr_t ifru_data;
272     } ifr_ifru;
273 #define ifr_addr    ifr_ifru.ifru_addr          /* address */
274 #define ifr_dstaddr ifr_ifru.ifru_dstaddr       /* other end of p-to-p link */
275 #define ifr_broadaddr   ifr_ifru.ifru_broadaddr /* broadcast address */
276 #define ifr_flags   ifr_ifru.ifru_flags         /* flags */
277 #define ifr_metric  ifr_ifru.ifru_metric        /* metric */
278 #define ifr_data    ifr_ifru.ifru_data          /* for use by interface */
279 };
```

if.h

图4-23 结构ifreq

if.h

```
292 struct  ifconf {
293     int ifc_len;                    /* size of associated buffer */
294     union {
295         caddr_t ifcu_buf;
296         struct  ifreq *ifcu_req;
297     } ifc_ifcu;
298 #define ifc_buf ifc_ifcu.ifcu_buf   /* buffer address */
299 #define ifc_req ifc_ifcu.ifcu_req   /* array of structures returned */
300 };
```

if.h

图4-24 结构ifconf

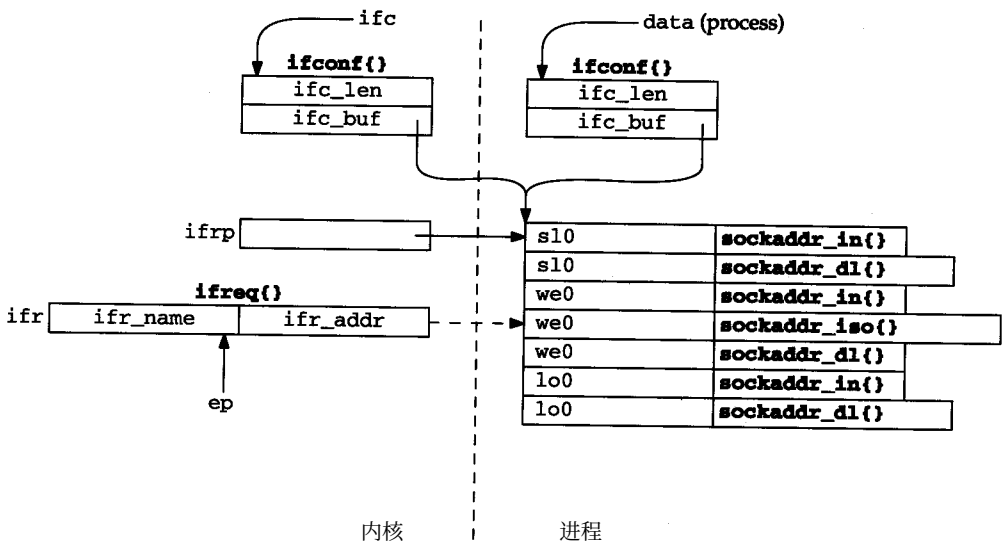


图4-25 ifconf 数据结构

if.c

```
462 int
463 ifconf(cmd, data)
464 int     cmd;
465 caddr_t data;
466 {
467     struct ifconf *ifc = (struct ifconf *) data;
468     struct ifnet *ifp = ifnet;
469     struct ifaddr *ifa;
470     char    *cp, *ep;
471     struct ifreq ifr, *ifrp;
472     int     space = ifc->ifc_len, error = 0;
473     ifrp = ifc->ifc_req;
474     ep = ifr.ifr_name + sizeof(ifr.ifr_name) - 2;
475     for (; space > sizeof(ifr) && ifp; ifp = ifp->if_next) {
476         strncpy(ifr.ifr_name, ifp->if_name, sizeof(ifr.ifr_name) - 2);
477         for (cp = ifr.ifr_name; cp < ep && *cp; cp++)
478             continue;
479         *cp++ = '0' + ifp->if_unit;
480         *cp = '\0';
481         if ((ifa = ifp->if_addrlist) == 0) {
482             bzero((caddr_t) & ifr.ifr_addr, sizeof(ifr.ifr_addr));
483             error = copyout((caddr_t) & ifr, (caddr_t) ifrp,
484                             sizeof(ifr));
485             if (error)
486                 break;
487             space -= sizeof(ifr), ifrp++;
488         } else
489             for (; space > sizeof(ifr) && ifa; ifa = ifa->ifa_next) {
490                 struct sockaddr *sa = ifa->ifa_addr;
491                 if (sa->sa_len <= sizeof(*sa)) {
492                     ifr.ifr_addr = *sa;
493                     error = copyout((caddr_t) & ifr, (caddr_t) ifrp,
494                                     sizeof(ifr));
495                     ifrp++;
496                 } else {
497                     space -= sa->sa_len - sizeof(*sa);
498                     if (space < sizeof(ifr))
499                         break;
500                     error = copyout((caddr_t) & ifr, (caddr_t) ifrp,
501                                     sizeof(ifr.ifr_name));
502                     if (error == 0)
503                         error = copyout((caddr_t) sa,
504                                         (caddr_t) & ifrp->ifr_addr, sa->sa_len);
505                     ifrp = (struct ifreq *)
506                         (sa->sa_len + (caddr_t) & ifrp->ifr_addr);
507                 }
508                 if (error)
509                     break;
510                 space -= sizeof(ifr);
511             }
512     }
513     ifc->ifc_len -= space;
514     return (error);
515 }
```

if.c

图4-26 函数ifconf

ifc是强制为一个ifconf结构指针的data。ifp从ifnet(列表头)开始遍历接口列表,

而ifa遍历每个接口的地址列表。 cp和ep控制构造在ifr中的接口文本名称， ifr是一个ifreq结构，它在接口名称和地址复制到进程的缓存前保存接口名称和地址。 ifrq指向这个缓存，并且在每个地址被复制后指向下一个。 space是进程缓存中剩余字节的个数， cp用来搜寻名称的结尾，而ep标志接口名称数字部分最后的可能位置。

475-488 for循环遍历接口列表。对于每个接口，文本名称被复制到 ifr_name，在ifr_name的后面跟着if_unit数的文本表示。如果没有给接口分配地址，一个全 0的地址被构造，所得的ifreq结构被复制到进程中，并减小space，增加ifrp。

489-515 如果接口有一个或多个地址，用 for循环来处理每个地址。地址加到 ifr中的接口名称中，然后ifr被复制到进程中。长度超过标准 sockaddr结构的地址不放到ifr中，并且直接复制到进程。在复制完每个地址后，调整 space和ifrp的值。所有接口处理完后，更新缓存长度 (ifc->ifc_len)，并且ifconf返回。系统调用 ioctl负责将结构ifconf中新的内容复制回进程中的结构 ifconf。

4.4.3 举例

图4-27显示了以太网、SLIP和环回接口被初始化后的接口结构的配置。

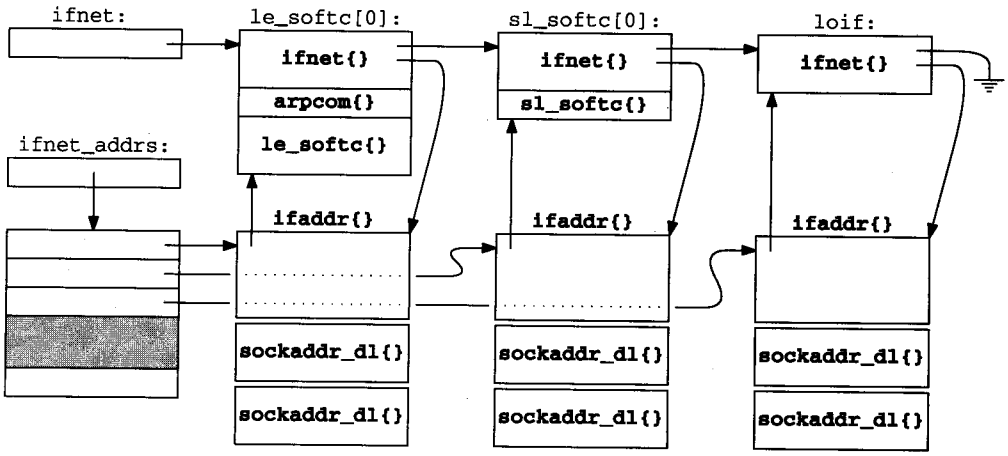


图4-27 接口和地址数据结构

图4-28显示了以下代码执行后的ifc和buffer的内容。

```
struct ifconf ifc;      /* SIOCGIFCONF adjusts this */
char buffer[144];       /* contains interface addresses when ioctl returns */
int s;                  /* any socket */

ifc.ifc_len = 144;
ifc.ifc_buf = buffer;
if (ioctl(s, SIOCGIFCONF, &ifc) < 0 ) {
    perror("ioctl failed");
    exit(1);
}
```

这里对命令SIOCGIFCONF操作的插口的类型没有限制，如我们所看到的，这个命令返回所有协议族类的地址。

在图4-28中，因为在缓存中返回的三个地址仅占用 108 (3×36)字节， ioctl将ifc_len

由144改为108。返回三个sockaddr_dl地址，并且这个缓存后面的36字节未用。每项的前16个字节包含接口的文本名称。在这里，这16字节中只有3个字节被使用。

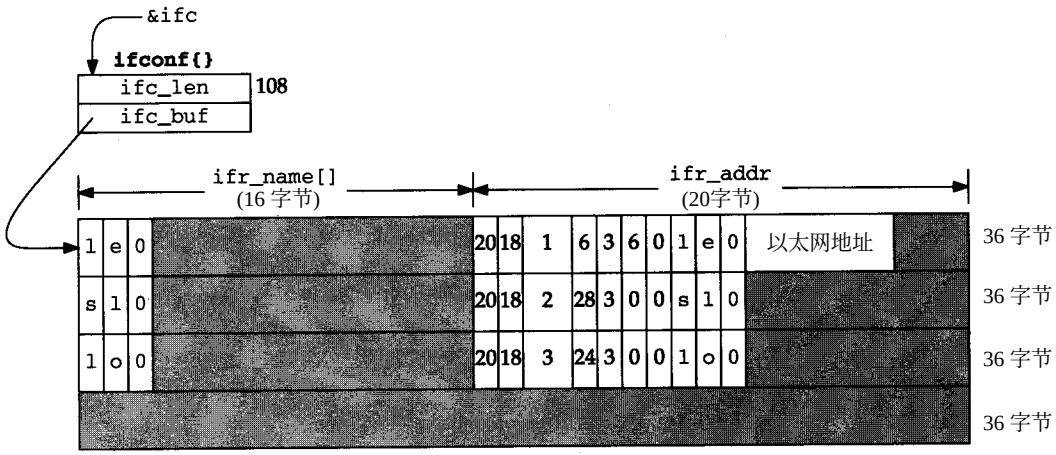


图4-28 SIOCGIFCONF 命令返回的数据

ifr_addr为一个sockaddr结构的形式，因此第一个值为长度(20字节)，且第二个值为地址的类型(18，AF_LINK)。接下来的一个值为sdl_index，与sdl_type一样，对于每个接口，它是不同的(与IFT_ETHER、IFT_SLIP和IFT_LOOP相对应的值为6、28和24)。

下面三个值为sa_nlen(文本名称的长度)、sa_alen(硬件地址的长度)及sa_slen(未用)。对于所有三项，sa_nlen都为3。以太网地址的sa_alen为6，而SLIP和环回接口的sa_alen为0。sa_slen总是为0。

最后，是接口的文本名称，其后面是硬件地址(仅对于以太网)。SLIP和环回接口在sockaddr_dl结构中不存放一个硬件级地址。

在此例中，仅返回sockaddr_dl地址(因为在图4-27中没有配置其他地址类型)，因此缓存中的每项大小一样。如果为每个接口配置其他地址(例如：IP或OSI地址)，它们会同sockaddr_dl地址一起返回，并且每项的大小根据返回的地址类型的不同而不同。

4.4.4 通用接口ioctl命令

图4-20中剩下的四个接口命令(SIOCGIFFLAGS、SIOCGIFMETRIC、SIOCSIFFLAGS和SIOCSIFMETRIC)由函数ifioctl处理。图4-29所示的是处理这些命令的case语句。

1. SIOCGIFLAGS和SIOCGIFMETRIC

410-416 对于两个SIOCG*xxx*命令，ifioctl将每个接口的if_flags或if_metric值复制到ifreq结构中。对于标志，使用联合的成员ifr_flags；而对于度量，使用成员ifr_metric(图4-23)。

2. SIOCSIFFLAGS

417-429 为改变接口的标志，调用进程必须有超级用户权限。如果进程正在关闭一个运行的接口或启动一个未运行的接口，分别调用if_down和if_up。

3. 忽略标志IFF_CANTCHAGE

430-434 回忆图3-7，有些接口标志不能被进程改变。表达式(ifp->if_flags&

IFF_CANTCHANGE)清除能被进程改变的接口标志，而表达式 (ifr->ifr_flags~IFF_CANTCHANGE)清除在请求中不被进程改变的标志。这两个表达式进行或运算并作为新值保存在ifp->if_flags中。在返回前，请求被传递给与设备相关联的 if_ioctl函数(例如：LANCE驱动器的leioctl —— 图4-31)。

if.c

```
410     switch (cmd) {
411     case SIOCGIFFLAGS:
412         ifr->ifr_flags = ifp->if_flags;
413         break;

414     case SIOCGIFMETRIC:
415         ifr->ifr_metric = ifp->if_metric;
416         break;

417     case SIOCSIFFLAGS:
418         if (error = suser(p->p_ucred, &p->p_acflag))
419             return (error);
420         if (ifp->if_flags & IFF_UP && (ifr->ifr_flags & IFF_UP) == 0) {
421             int     s = splimp();
422             if_down(ifp);
423             splx(s);
424         }
425         if (ifr->ifr_flags & IFF_UP && (ifp->if_flags & IFF_UP) == 0) {
426             int     s = splimp();
427             if_up(ifp);
428             splx(s);
429         }
430         ifp->if_flags = (ifp->if_flags & IFF_CANTCHANGE) |
431             (ifr->ifr_flags & ~IFF_CANTCHANGE);
432         if (ifp->if_ioctl)
433             (void) (*ifp->if_ioctl) (ifp, cmd, data);
434         break;

435     case SIOCSIFMETRIC:
436         if (error = suser(p->p_ucred, &p->p_acflag))
437             return (error);
438         ifp->if_metric = ifr->ifr_metric;
439         break;
```

if.c

图4-29 函数ifioctl ：标志和度量

4. SIOCSIFMETRIC

435-439 改变接口的度量要容易些；进程同样要有超级用户权限， ifioctl将接口新的度量复制到if_metric中。

4.4.5 if_down和if_up函数

利用程序ifconfig，一个管理员可以通过命令 SIOCSIFFLAGS设置或清除标志IFF_UP来启用或禁用一个接口。图4-30显示了函数if_down和if_up的代码。

292-302 当一个接口被关闭时， IFF_UP标志被清除并且对与接口关联的每个地址用pfctlinput(7.7节)发送命令PRC_IFDOWN。这给每个协议一个机会来响应被关闭的接口。有些协议，如OSI，要使用接口来终止连接。对于IP，如果可能，要通过其他接口为连接进行重新路由。TCP和UDP忽略失效的接口，并依赖路由协议去发现分组的可选路径。

if_qflush忽略接口的任何排队分组。rt_ifmsg通知路由系统发生的变化。TCP自动重传丢失的分组；UDP应用必须自己显式地检测这种情况，并对此作出响应。

308-315 当一个接口被启用时，IFF_UP标志被设置，并且rt_ifmsg通知路由系统接口状态发生变化。

if.c

```
292 void
293 if_down(ifp)
294 struct ifnet *ifp;
295 {
296     struct ifaddr *ifa;

297     ifp->if_flags &= ~IFF_UP;
298     for (ifa = ifp->if_addrlist; ifa; ifa = ifa->ifa_next)
299         pfctlinput(PRC_IFDOWN, ifa->ifa_addr);
300     if_qflush(&ifp->if_snd);
301     rt_ifmsg(ifp);
302 }

308 void
309 if_up(ifp)
310 struct ifnet *ifp;
311 {
312     struct ifaddr *ifa;

313     ifp->if_flags |= IFF_UP;
314     rt_ifmsg(ifp);
315 }
```

if.c

图4-30 函数if_down 和if_up

4.4.6 以太网、SLIP和环回

我们看图4-29中处理SIOCSIFFLAGS命令的代码，ifioctl调用接口的if_ioctl函数。在我们的三个例子接口中，函数 slioctl和loioctl为这个被ifioctl忽略的命令返回EINVAL。图4-31显示了函数leioctl及LANCE以太网驱动程序的SIOCSIFFLAGS命令的处理。

if_le.c

```
614 leioctl(ifp, cmd, data)
615 struct ifnet *ifp;
616 int     cmd;
617 caddr_t data;
618 {
619     struct ifaddr *ifa = (struct ifaddr *) data;
620     struct le_softc *le = &le_softc[ifp->if_unit];
621     struct lereg1 *ler1 = le->sc_r1;
622     int     s = splimp(), error = 0;

623     switch (cmd) {

                /* SIOCSIFADDR code (Figure 6.28) */

638     case SIOCSIFFLAGS:
```

图4-31 函数leioctl ：SIOCSIFFLAGS

```
639         if ((ifp->if_flags & IFF_UP) == 0 &&
640             ifp->if_flags & IFF_RUNNING) {
641             LERDWR(le->sc_r0, LE_STOP, ler1->ler1_rdp);
642             ifp->if_flags &= ~IFF_RUNNING;
643         } else if (ifp->if_flags & IFF_UP &&
644                   (ifp->if_flags & IFF_RUNNING) == 0)
645             leinit(ifp->if_unit);
646         /*
647          * If the state of the promiscuous bit changes, the interface
648          * must be reset to effect the change.
649          */
650         if (((ifp->if_flags ^ le->sc_iflags) & IFF_PROMISC) &&
651             (ifp->if_flags & IFF_RUNNING)) {
652             le->sc_iflags = ifp->if_flags;
653             lereset(ifp->if_unit);
654             lestart(ifp);
655         }
656         break;

            /* SIOCADDMULTI and SIOCDELMULTI code (Figure 12.31) */

672     default:
673         error = EINVAL;
674     }
675     splx(s);
676     return (error);
677 }
```

if_le.c

图4-31 (续)

614-623 leioctl把第三个参数data转换为一个ifaddr结构的指针，并保存在ifa中。le指针引用下标为ifp->if_unit的le_softc结构。基于cmd的switch语句构成了这个函数的主体。

638-656 在图4-31中仅显示了case SIOCSIFFLAGS。这次ifioctl调用leioctl，接口标志被改变。显示的代码强制物理接口进入标志所配置的状态。如果要关闭接口 (没有设置IFF_UP)，但接口正在工作，则关闭接口。若要启动未操作的接口，接口被初始化并重启。

如果混淆比特被改变，那么就关闭接口，复位，并重启来实现这种变化。

仅当要求改变IFF_PROMISC比特时包含异或和IFF_PROMISC的表达式才为真。

672-677 处理未识别命令的default情况分支发送EINVAL，并在函数的结尾将它返回。

4.5 小结

在本章中，我们说明了LANCE以太网设备驱动程序的实现，这个驱动程序在全书中多处引用。我们还看到了以太网驱动程序如何检测输入中的广播地址和多播地址，如何检测以太网和802.3封装，以及如何将输入的帧分用到相应的协议队列中。在第 21章中我们会看到IP地址(单播、广播和多播)是如何在输出转换成正确的以太网地址。

最后，我们讨论了协议专用的ioctl命令，它用来访问接口层数据结构。

习题

4.1 在leread中，当接收到一个广播分组时，总是设置标志 M_MCAST(除了M_BCAST外)。与ether_input的代码比较，为什么在leread和ether_input中设置此标志？它至关重要吗？哪个正确？

4.2 在ether_input(图4-13)中，如果交换广播地址和多播地址检测次序会发生什么情况？如果在检测多播地址的if语句前加上一个else会发生什么情况？

第5章　接口：SLIP和环回

5.1　引言

在第4章中，我们查看了以太网接口。在本章中，我们讨论　SLIP和环回接口，同样用`ioctl`命令来配置所有网络接口。 SLIP驱动程序使用的TCP压缩算法在29.13节讨论。环回驱动程序比较简单，在这里我们要对它进行完整地讨论。

像图4-2一样，图5-1列出了针对我们三个示例驱动程序的入口点。

ifnet	以 太 网	SLIP	环　回	说　明
if_init	leinit			初始化硬件
if_output	ether_output	sloutput	looutput	接收并将要传输的分组进行排队
if_start	lestart			开始传输帧
if_done				输出完成(未用)
if_ioctl	leioctl	slioctl	loioctl	从一个进程处理ioctl命令
if_reset	lereset			将设备重新设置为一已知状态
if_watchdog				监视设备的故障或采集统计信息

图5-1　例子驱动程序的接口函数

5.2　代码介绍

SLIP和环回驱动程序的代码文件列于图5-2中。

文　件	说　明
net/if_slvar.h	SLIP定义
net/if_sl.c	SLIP驱动程序函数
net/if_loop.c	环回驱动程序

图5-2　本章讨论的文件

5.2.1　全局变量

在本章讨论SLIP和环回接口结构。全局变量见图5-3。

变　量	数据类型	说　明
sl_softc	struct sl_softc []	SLIP接口
loif	struct ifnet	环回接口

图5-3　本章中介绍的全局变量

`sl_softc`是一个数组，因为可能有很多SLIP接口。`loif`不是一个数组，因为只可能有一个环回接口。

5.2.2 统计量

在第4章讨论的ifnet结构的统计也会被SLIP和环回驱动程序更新。采集的另一个统计量(它不在ifnet结构中)显示在图5-4中。

变 量	说 明	被SNMP使用
tk_nin	被任何串行接口(被SLIP驱动程序更新)接收的字节数	

图5-4 变量tk_nin

5.3 SLIP接口

一个SLIP接口通过一个标准的异步串行线与一个远程系统通信。像以太网一样， SLIP定义了一个标准的方法对传输在串行线上的 IP分组进行组帧。图5-5显示了将一个包含SLIP保留字符的IP分组封装到一个SLIP帧中。

分组用SLIP END字符0xc0来分割开。如果END字符出现在IP分组中，则在它前面填充SLIP ESC字符0xdb，并且在传输时将它替换为 0xdc。当ESC字符出现在IP分组中时，就在它前面填充ESC字符0xdb，并在传输时将它替换为0xdd。

因为在SLIP帧(与以太网比较)中没有类型字段， SLIP仅适用于传输IP分组。

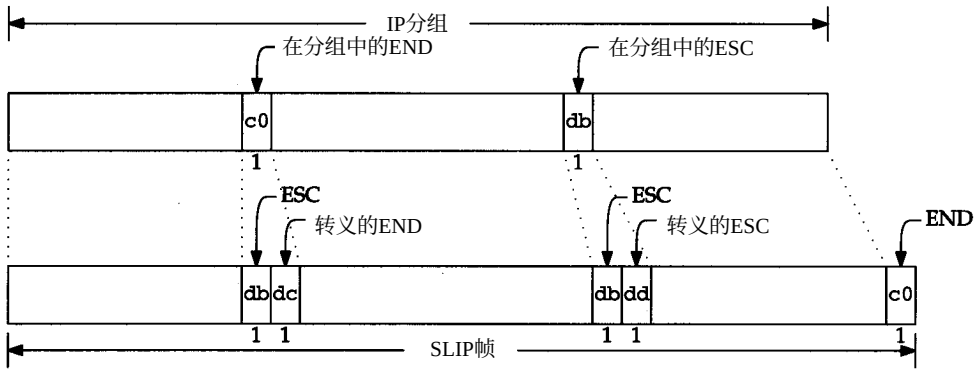


图5-5 将一个IP分组进行SLIP封装

在RFC 1055 [Romkey 1988]中讨论了SLIP，陈述了它的很多弱点和非标准情况。卷1中包含了SLIP封装的详细讨论。

点对点协议(PPP)被设计用来解决SLIP的问题，并提供一个标准方法来通过一个串行链路传输帧。PPP在RFC 1332 [McGregor 1992]和RFC 1548 [Simpson 1993]中定义。Net/3不包含一个PPP的实现，因此我们不在本书中讨论它。关于 PPP的更多信息见卷1的2.6节。附录B讨论在哪里获得一个PPP实现的参考。

5.3.1 SLIP线路规程：SLIPDISC

在Net/3中， SLIP接口依靠一个异步串行设备驱动器来发送和接收数据。传统上，这些设备驱动器称为TTY(电传机)。Net/3 TTY子系统包括一个线路规程(Line discipline)的概念，这个线路规程作为一个在物理设备和 I/O系统调用(如read和write)之间的过滤器。一个线路规

程实现以下特性：如行编辑、换行和回车处理、制表符扩展等等。 SLIP接口作为TTY子系统的一个线路规程，但它不把输入数据传给从设备读数据的进程，也不接受来自向设备写数据的进程的输出数据。SLIP接口将输入分组传给IP输入队列，并通过SLIP的ifnet结构中的函数if_output来获得要输出的分组。内核通过一个整数常量来标识线路规程，对于 SLIP，该常量是SLIPDISC。

图5-6左边显示的是传统的线路规程，右边是 SLIP规程。我们在右边用slattach显示进程，因为它是初始化SLIP接口的程序。TTY子系统和线路规程的细节超出了本书的范围。我们仅介绍理解SLIP代码工作的相关信息。对于更多关于 TTY子系统的信息见 [Leffler et al. 1989]。图5-7列出了实现SLIP驱动程序的函数。中间的列指示函数是否实现线路规程特性和(或)网络接口特性。

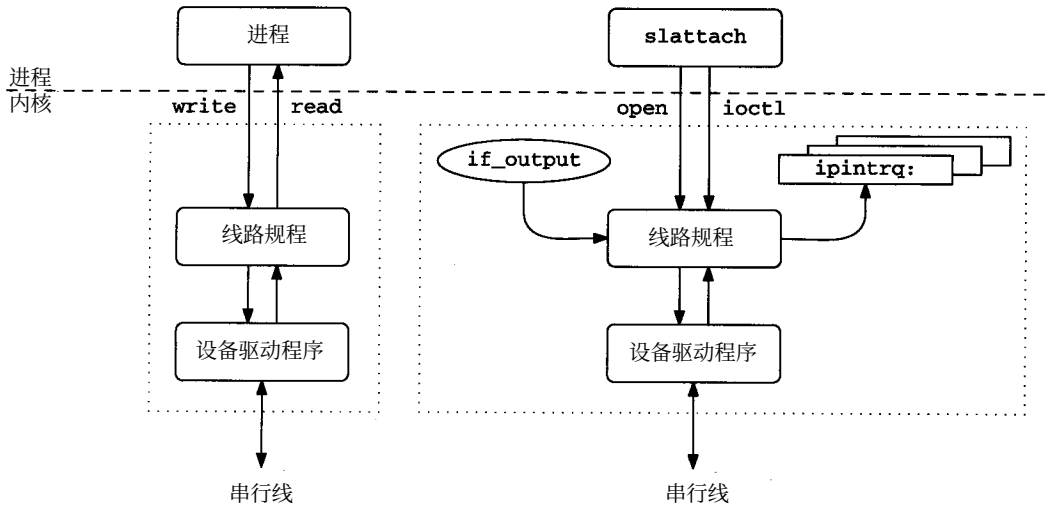


图5-6 SLIP接口作为一个线路规程

函 数	网络接口	线路规程	说 明
slattach	•		初始化sl_softc结构，并将它连接到ifnet列表
slinit	•		初始化SLIP数据结构
sloutput	•		对相关TTY设备上要传输的输出分组进行排队
slioctl	•		处理插口ioctl请求
sl_btom	•		将一个设备缓存转换成一个mbuf链表
slopen		•	将sl_softc结构连接到TTY设备，并初始化驱动程序
slclose		•	取消TTY设备与sl_softc结构的连接，标记接口为关闭，并释放存储器
sltioctl		•	处理TTY ioctl命令
slstart	•	•	从队列中取分组，并开始在TTY设备上传输数据
slinput	•	•	处理从TTY设备输入的字节，如果整个帧被接收，就排列输入的分组

图5-7 SLIP设备驱动程序的函数

在Net/3中的SLIP驱动程序通过支持TCP分组首部压缩来得到更好的吞吐量。我们在 29.13节讨论分组首部压缩，因此，图 5-7跳过实现这些特性的函数。

Net/3 SLIP接口还支持一种转义序列。当接收方检测到这个序列时，就终止 SLIP

的处理，并将对设备的控制返回给标准线路规程。我们这里的讨论忽略这个处理。

图5-8显示了作为一个线路规程的SLIP和作为一个网络接口的SLIP间的复杂关系。

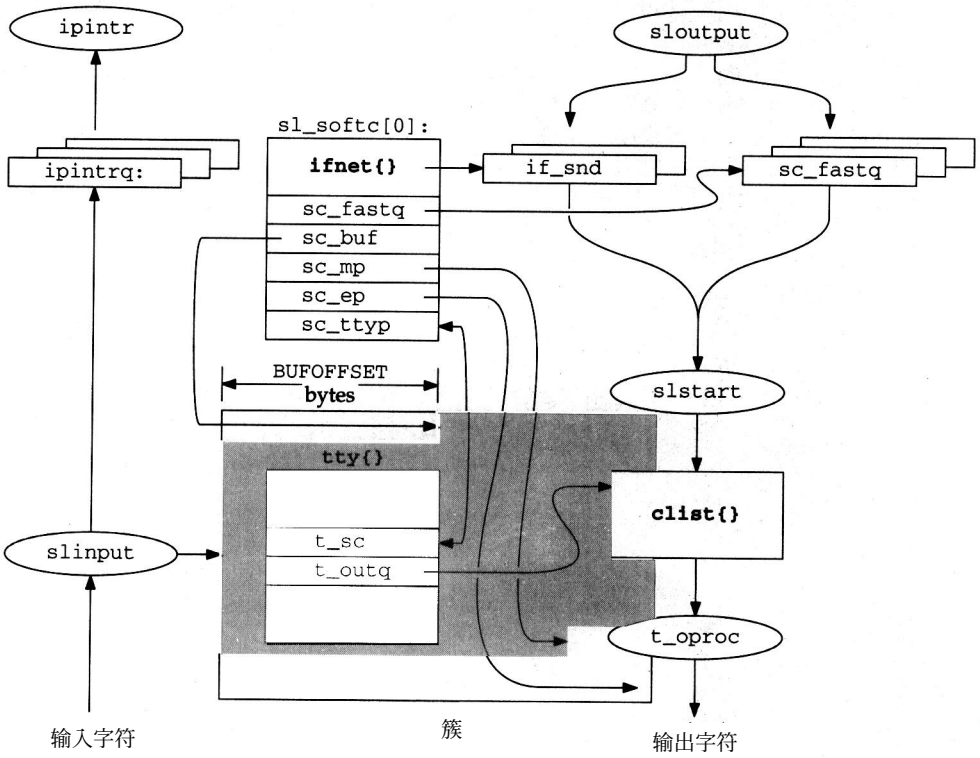


图5-8 SLIP设备驱动程序

在Net/3中，sc_ttyp和t_sc指向tty结构和sl_softc[0]结构。由于使用两个箭头会使图显得较乱，我们用一对相反的箭头表示两个指针来说明结构间的双链。

在图5-8中包含很多信息：

- 结构sl_softc表示的网络接口和结构tty表示的TTY设备。
- 输入字节存放在簇中(显示在结构tyy后面)。当一个完整的SLIP帧被接收时，封装的IP分组被slinput放到ipintrq中。
- 输出分组从if_snd或sc_fastq退队，转换成SLIP帧，并被slstart传给TTY设备。TTY缓存将字节输出到结构clist。函数t_oproc取完，并传输在clist结构中的字节。

5.3.2 SLIP初始化：slopen和slinit

我们在3.7节讨论了slattach是如何初始化sl_softc结构的。接口虽然被初始化，但还不能操作，直到一个程序(通常是slattach)打开一个TTY设备(例如：/dev/tty01)，并发送一个ioctl命令用SLIP规程代替标准的线路规程才能操作。这时， TTY子系统调用线路规程的打开函数(在此是slopen)，此函数在一个特定TTY设备和一个特定SLIP接口间建立关联。slopen显示在图5-9中。

if_sl.c

```
181 int
182 slopen(dev, tp)
183 dev_t   dev;
184 struct tty *tp;
185 {
186     struct proc *p = curproc;    /* XXX */
187     struct sl_softc *sc;
188     int     nsl;
189     int     error;

190     if (error = suser(p->p_ucred, &p->p_acflag))
191         return (error);

192     if (tp->t_line == SLIPDISC)
193         return (0);

194     for (nsl = NSL, sc = sl_softc; --nsl >= 0; sc++)
195         if (sc->sc_ttyp == NULL) {
196             if (slinit(sc) == 0)
197                 return (ENOBUFS);
198             tp->t_sc = (caddr_t) sc;
199             sc->sc_ttyp = tp;
200             sc->sc_if.if_baudrate = tp->t_ospeed;
201             ttyflush(tp, FREAD | FWRITE);
202             return (0);
203         }
204     return (ENXIO);
205 }
```

if_sl.c

图5-9 函数slopen

181-193 传递给slopen的两个参数为：dev，一个内核设备标识，slopen未用此参数；tp，一个指向此TTY设备相关tty结构的指针。最开始是一些预防处理：若进程没有超级用户权限，或TTY的线路规程已经被设置为SLIPDISC，则slopen立即返回。

194-205 for循环在sl_softc结构数组中查找第一个未用的项，调用slinit (5.10节)，通过t_sc和sc_ttyp加进结构tty和sl_softc，并将TTY输出速率(t_ospeed)复制到SLIP接口。ttyflush丢弃任何在TTY队列中追加的输入输出数据。如果一个SLIP接口结构不可用，slopen返回ENXIO。若成功，返回0。

注意，第一个变量sl_softc结构与TTY设备相关。如果系统有多个SLIP线路，在TTY设备和SLIP接口间不需要固定的映射。实际上，这个映射依赖于slattach打开和关闭TTY设备的次序。

显示在图5-10中的函数slinit初始化结构sl_softc。

156-175 函数slinit分配一个mbuf簇，并将它用三个指针连接到结构sl_softc。当一个完整的SLIP帧被接收后，输入字节存储在这个簇中。sc_buf总是指向簇中的这个分组的起始位置，sc_mp指向要接收的下一个字节的位置，并且sc_ep指向这个簇的结束。sl_compress_init为此链路初始化TCP首部的压缩状态(29.13节)。

在图5-8中，我们看到sc_buf不指向簇的第一个字节。slinit保留了148字节(BUFOFFSET)的空间，因为输入分组可能含有一个压缩了的首部，它会扩展来填充这个空间。在簇中已接收的字节用阴影表示。我们看到sc_mp指向接收的最后一个字节的下一个字节，并且sc_ep指向这个簇的结尾。图5-11显示了在几个SLIP常量间的关系。

使这个接口能运行，剩下的要做的工作就是给它分配一个 IP地址。同以太网驱动程序一样，我们将地址分配的讨论推迟到6.6节。

if_sl.c

```
156 static int
157 slinit(sc)
158 struct sl_softc *sc;
159 {
160     caddr_t p;

161     if (sc->sc_ep == (u_char *) 0) {
162         MCLALLOC(p, M_WAIT);
163         if (p)
164             sc->sc_ep = (u_char *) p + SLBUFSIZE;
165         else {
166             printf("sl%d: can't allocate buffer\n", sc - sl_softc);
167             sc->sc_if.if_flags &= ~IFF_UP;
168             return (0);
169         }
170     }
171     sc->sc_buf = sc->sc_ep - SLMAX;
172     sc->sc_mp = sc->sc_buf;
173     sl_compress_init(&sc->sc_comp);
174     return (1);
175 }
```

if_sl.c

图5-10 函数slinit

常 量	值	说 明
MCLBYTES	2048	一个mbuf簇的大小
SLBUFSIZE	2048	一个未压缩的SLIP分组的最大长度 —— 包括一个BPF首部
SLIP_HDRLEN	16	SLIP BPF首部的大小
BUFOFFSET	148	一个扩展的TCP/IP首部的最大长度加上一个BPF首部的大小
SLMAX	1900	一个存储在簇中的压缩SLIP分组的最大长度
SLMTU	296	SLIP分组的最佳长度；导致最小的时延，同时还有较高的批量吞吐量
SLIP_HIWAT	100	在TTY输出队列中排队的最大字节数
BUFOFFSET+SLMAX=SLBUFSIZE=MCLBYTES		

图5-11 SLIP常量

5.3.3 SLIP输入处理：slinput

TTY设备驱动程序每次调用slinput，都将输入字符传给SLIP线路规程。图5-12显示了函数slinput，但跳过了帧结束的处理，对于它我们分开讨论。

527-545 传递给slinput的参数为：c，下一个输入字符；tp，一个指向设备tty结构的指针。全局整数tk_nin计算所有TTY设备的输入字符数。slinput将tp->t_sc转换成sc，sc是指向一个sl_softc结构的指针。如果这个TTY设备没有相关联的接口，slinput立即返回。

slinput的第一个参数是一个整数。除了接收的字符，c还包含从TTY设备驱动程序以高位在前的比特序发送的控制字符。如果在c中指示了一个差错，或调制解调器控制线禁用并且不应该被忽略，则SC_ERROR被置位，并且slinput返回。之后，当slinput处理END字符时，此帧被丢弃。标志CLOCAL指示系统应该把这个线路视为一个本地线路 (即不是一个拨号线路)，并且不应该看到调制解调器的控制信号。

if_sl.c

```
527 void
528 slinput(c, tp)
529 int     c;
530 struct tty *tp;
531 {
532     struct sl_softc *sc;
533     struct mbuf *m;
534     int     len;
535     int     s;
536     u_char  chdr[CHDR_LEN];

537     tk_nin++;
538     sc = (struct sl_softc *) tp->t_sc;
539     if (sc == NULL)
540         return;
541     if (c & TTY_ERRORMASK || ((tp->t_state & TS_CARR_ON) == 0 &&
542                               (tp->t_cflag & CLOCAL) == 0)) {
543         sc->sc_flags |= SC_ERROR;
544         return;
545     }
546     c &= TTY_CHARMASK;

547     ++sc->sc_if.if_ibytes;

548     switch (c) {

549     case TRANS_FRAME_ESCAPE:
550         if (sc->sc_escape)
551             c = FRAME_ESCAPE;
552         break;
553     case TRANS_FRAME_END:
554         if (sc->sc_escape)
555             c = FRAME_END;
556         break;

557     case FRAME_ESCAPE:
558         sc->sc_escape = 1;
559         return;

560     case FRAME_END:
            /* FRAME_END code (Figure 5.13) */
636     }
637     if (sc->sc_mp < sc->sc_ep) {
638         *sc->sc_mp++ = c;
639         sc->sc_escape = 0;
640         return;
641     }
642     /* can't put lower; would miss an extra frame */
643     sc->sc_flags |= SC_ERROR;

644 error:
645     sc->sc_if.if_ierrors++;
646 newpack:
647     sc->sc_mp = sc->sc_buf = sc->sc_ep - SLMAX;
648     sc->sc_escape = 0;
649 }
```

if_sl.c

图5-12 函数slinput

546-636 slinput丢弃c中的控制比特，并用TTY_CHARMASK来屏蔽掉，更新接口上接收字节数的计数，同时跳过接收到的字符：

- 如果c是一个转义的ESC字符，并且前一字符为ESC，则slinput用一个ESC字符替代c。
- 如果c是一个转义的END字符，并且前一字符为ESC，则slinput用一个END字符代替c。
- 如果c是SLIP ESC字符，则将sc_escape置位，并且slinput立即返回(即，ESC字符被丢弃)。
- 如果c是SLIP END字符，则将分组放到IP输入队列。处理SLIP帧结束字符的代码显示在图5-13中。

if_sl.c

```
560     case FRAME_END:
561         if (sc->sc_flags & SC_ERROR) {
562             sc->sc_flags &= ~SC_ERROR;
563             goto newpack;
564         }
565         len = sc->sc_mp - sc->sc_buf;
566         if (len < 3)
567             /* less than min length packet - ignore */
568             goto newpack;

569         if (sc->sc_bpf) {
570             /*
571              * Save the compressed header, so we
572              * can tack it on later.  Note that we
573              * will end up copying garbage in some
574              * cases but this is okay.  We remember
575              * where the buffer started so we can
576              * compute the new header length.
577              */
578             bcopy(sc->sc_buf, chdr, CHDR_LEN);
579         }
580         if ((c = (*sc->sc_buf & 0xf0)) != (IPVERSION << 4)) {
581             if (c & 0x80)
582                 c = TYPE_COMPRESSED_TCP;
583             else if (c == TYPE_UNCOMPRESSED_TCP)
584                 *sc->sc_buf &= 0x4f;     /* XXX */
585             /*
586              * We've got something that's not an IP packet.
587              * If compression is enabled, try to decompress it.
588              * Otherwise, if auto-enable compression is on and
589              * it's a reasonable packet, decompress it and then
590              * enable compression.  Otherwise, drop it.
591              */
592             if (sc->sc_if.if_flags & SC_COMPRESS) {
593                 len = sl_uncompress_tcp(&sc->sc_buf, len,
594                                         (u_int) c, &sc->sc_comp);
595                 if (len <= 0)
596                     goto error;
597             } else if ((sc->sc_if.if_flags & SC_AUTOCOMP) &&
598                        c == TYPE_UNCOMPRESSED_TCP && len >= 40) {
599                 len = sl_uncompress_tcp(&sc->sc_buf, len,
600                                         (u_int) c, &sc->sc_comp);
601                 if (len <= 0)
602                     goto error;
603                 sc->sc_if.if_flags |= SC_COMPRESS;
604             } else
```

图5-13 函数slinput：帧结束处理

```
605                 goto error;
606         }
607         if (sc->sc_bpf) {
608             /*
609              * Put the SLIP pseudo-"link header" in place.
610              * We couldn't do this any earlier since
611              * decompression probably moved the buffer
612              * pointer.  Then, invoke BPF.
613              */
614             u_char *hp = sc->sc_buf - SLIP_HDRLEN;

615             hp[SLX_DIR] = SLIPDIR_IN;
616             bcopy(chdr, &hp[SLX_CHDR], CHDR_LEN);
617             bpf_tap(sc->sc_bpf, hp, len + SLIP_HDRLEN);
618         }
619         m = sl_btom(sc, len);
620         if (m == NULL)
621             goto error;

622         sc->sc_if.if_ipackets++;
623         sc->sc_if.if_lastchange = time;
624         s = splimp();
625         if (IF_QFULL(&ipintrq)) {
626             IF_DROP(&ipintrq);
627             sc->sc_if.if_ierrors++;
628             sc->sc_if.if_iqdrops++;
629             m_freem(m);
630         } else {
631             IF_ENQUEUE(&ipintrq, m);
632             schednetisr(NETISR_IP);
633         }
634         splx(s);
635         goto newpack;
```

if_sl.c

图5-13 (续)

通过这个switch语句的普通控制流会落到switch外(这里没有default情况)。大多数字节是数据，并且不与这4种情况中的任何一种匹配。前两个case的控制也会落到这个switch外。

637-649 如果控制落到switch外，接收的字符为IP分组中的一部分。这个字符被存储到簇中(如果还有空间)，指针增加，sc_escape被清除，并且slinput返回。

如果簇满，字符被丢弃，并且slinput设置SC_ERROR。如果簇满或在处理帧结束时检测到一个差错，则控制跳到error。程序在newpack为一个新的分组重设簇指针，sc_escape被清除，并且slinput返回。

图5-13显示了图5-12中跳过的FRAME_END代码。

560-579 如果SC_ERROR被设置，同时正在接收分组或如果分组长度小于3字节(记住，分组可能被压缩)，则slinput立即丢弃此输入SLIP分组。

如果SLIP接口带有BPF，slinput在chdr数组中保存这个首部的一个备份(可能被压缩)。

580-606 通过检查分组的第一个字节，slinput判断它是一个未压缩的IP分组，还是一个压缩的TCP分段，或者一个未压缩的TCP分段。类型存放在c中，并且类型信息从数据的第一个字节中移去(29.13节)。如果分组以压缩形式出现，并且允许压缩，sl_uncompress_tcp对分组进行解压缩。如果禁止压缩，自动允许压缩被设置，并且如果分组足够大，则仍然调

用sl_uncompress_tcp。如果是一个压缩的TCP分组，则设置压缩标志。

若分组不被识别，slinput跳到error，丢弃此分组。29.13节详细讨论了首部压缩技术。现在簇中包含一个完整的未压缩分组。

607-618 SLIP解压缩分组后，首部和数据传给BPF。图5-14显示了slinput构造的缓存格式。

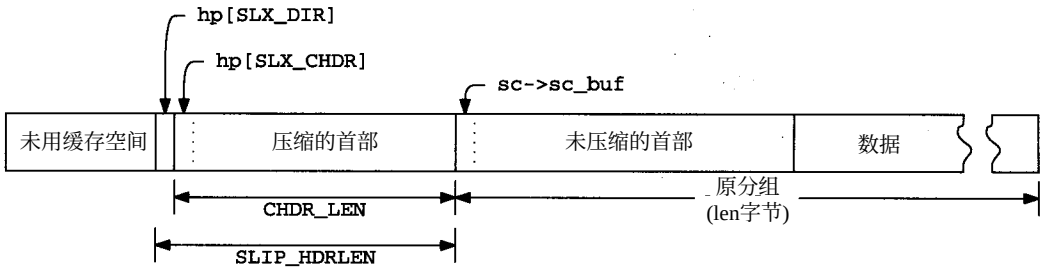


图5-14 BPF格式的SLIP分组

BPF首部的第一个字节是分组方向的编码，在此例中是输入(SLIPDIR_IN)。接下来的15字节包含压缩的首部。整个分组被传给bpf_tap。

619-635 sl_btom将簇转换为一个mbuf链表。如果分组足够小，能放到一个单独的mbuf中，sl_btom就将分组从簇复制到一个新分配的mbuf的分组首部；否则sl_btom将这个簇连接到一个mbuf，并为这个接口分配一个新簇。这样比从一个簇复制到另一个簇要快。我们在本书中不显示sl_btom的代码。

因为在SLIP接口上只能传输IP分组，slinput不必选择协议队列(如以太网驱动程序所做)。分组在ipintrq中排队，一个IP软件中断被调度，并且slinput跳到newpack，更新簇的分组指针，并清除sc_escape。

> 如果分组不能在ipintrq上排队，SLIP驱动程序增加if_ierrors，而在这种情况下，以太网或环回驱动程序都不增加这个统计量。

即使在spltty调用slinput，访问IP输入队列必须用splimp保护。回忆图1-14，一个splimp中断能抢占spltty进程。

5.3.4 SLIP输出处理：sloutput

如所有的网络接口，当一个网络层协议调用接口的if_output函数时，开始处理输出。对于以太网驱动程序，此函数是ether_output。而对于SLIP，此函数是sloutput(图5-15)。

259-289 sloutput的4个参数为：ifp，指向SLIP ifnet结构(在此例中是一个sl_softc结构)的指针；m，指向排队等待输出的分组的指针；dst，分组下一跳的目标地址；rtp，指向一个路由表项的指针。sloutput未用第4个参数，但却是要求的，因为sloutput必须匹配在ifnet结构中的if_output函数原型。

sloutput确认dst是一个IP地址，接口被连接到一个TTY设备，并且这个TTY设备是正在运行的(即有载波信号，或应忽略它)。如果任何检测失败，则返回差错。

290-291 SLIP为输出分组维护两个队列。默认选择标准队列if_snd。

292-295 如果输出分组包含一个ICMP报文，并且接口的SC_NOICMP被置位，则丢弃此分组。这防止一个SLIP链路被一个恶意用户发送的无关ICMP分组(例如ECHO分组)所淹没(第11章)。

if_sl.c

```
259 int
260 sloutput(ifp, m, dst, rtp)
261 struct ifnet *ifp;
262 struct mbuf *m;
263 struct sockaddr *dst;
264 struct rtentry *rtp;
265 {
266     struct sl_softc *sc = &sl_softc[ifp->if_unit];
267     struct ip *ip;
268     struct ifqueue *ifq;
269     int     s;
270     /*
271      * Cannot happen (see slioctl).  Someday we will extend
272      * the line protocol to support other address families.
273      */
274     if (dst->sa_family != AF_INET) {
275         printf("sl%d: af%d not supported\n", sc->sc_if.if_unit,
276                dst->sa_family);
277         m_freem(m);
278         sc->sc_if.if_noproto++;
279         return (EAFNOSUPPORT);
280     }
281     if (sc->sc_ttyp == NULL) {
282         m_freem(m);
283         return (ENETDOWN);      /* sort of */
284     }
285     if ((sc->sc_ttyp->t_state & TS_CARR_ON) == 0 &&
286         (sc->sc_ttyp->t_cflag & CLOCAL) == 0) {
287         m_freem(m);
288         return (EHOSTUNREACH);
289     }
290     ifq = &sc->sc_if.if_snd;
291     ip = mtod(m, struct ip *);
292     if (sc->sc_if.if_flags & SC_NOICMP && ip->ip_p == IPPROTO_ICMP) {
293         m_freem(m);
294         return (ENETRESET);     /* XXX ? */
295     }
296     if (ip->ip_tos & IPTOS_LOWDELAY)
297         ifq = &sc->sc_fastq;
298     s = splimp();
299     if (IF_QFULL(ifq)) {
300         IF_DROP(ifq);
301         m_freem(m);
302         splx(s);
303         sc->sc_if.if_oerrors++;
304         return (ENOBUFS);
305     }
306     IF_ENQUEUE(ifq, m);
307     sc->sc_if.if_lastchange = time;
308     if (sc->sc_ttyp->t_outq.c_cc == 0)
309         slstart(sc->sc_ttyp);
310     splx(s);
311     return (0);
312 }
```

if_sl.c

图5-15 函数sloutput

差错码ENETRESET指示分组因决策而被丢弃 (相对于网络故障)。我们在第11章会看到除

了在本地产生一个ICMP报文外，此差错简单地被忽略，在这种情况下，一个差错返回给发送此报文的进程。

Net/2在这种情况返回一个0。对于一个诊断工具，如ping或traceroute，会出现这种情况：好像这个分组消失了，因为输出操作会报告成功完成。

通常，ICMP报文可以被丢弃。对于正确的操作，它们并不必要，但丢弃它们会造成更多的麻烦，可能导致不佳的路由决定和较差的性能，并且会浪费网络资源。

296-297 如果在输出分组的TOS字段指明低时延服务(IPTOS_LOWDELAY)，则输出队列改为sc_fastq。

RFC 1700和RFC 1349 [Almquist 1992]规定了标准协议的TOS设置。为Telnet、Rlogin、FTP(控制)、TFTP、SMTP(命令阶段)和DNS(UDP查询)指明了低时延服务。更多细节见卷1的3.2节。

在以前的BSD版本中，ip_tos不由应用程序设置。SLIP驱动程序通过检查在IP分组中的传输首部来实现TOS排队。如果发现FTP(命令)、Telnet或Rlogin端口的TCP分组，分组就如指明了IPTOS_LOWDELAY一样被排队。很多路由器仍然这样，因为很多这些交互服务的实现仍然不设置ip_tos。

298-312 现在分组被放到所选择的队列中，接口统计被更新，并且 (如果TTY输出队列为空) sloutput调用slstart来发起对此分组的传输。

如果接口队列满，则SLIP增加if_oerrors；而对于ether_output，则不是这样做的。

不像以太网输出函数(ether_output)，sloutput不为输出分组构造一个数据链路首部。因为在SLIP网络上的另一系统在串行链路的另一端，所以不需要硬件地址或一个协议 (如ARP)在IP地址和硬件地址间进行转换。协议标识符 (如以太网类型字段)也是多余的，因为一个SLIP链路仅承载IP分组。

5.3.5 slstart函数

除了被sloutput调用外，当TTY取完它的输出队列并要传输更多的字节时， TTY设备调用slstart。TTY子系统通过一个 clist结构管理它的队列。在图 5-8中，输出 clist t_outq显示在slstart下面和设备的t_oproc函数的上面。slstart把字节添加到队列中，而t_oproc将队列取完并传输这些字节。

函数slstart显示在图5-16中。

318-358 当slstart函数被调用时，tp指向设备的tty结构。slstart的主体由一个for循环构成。如果输出队列t_outq不空，slstart调用设备的输出函数t_oproc，此函数传输设备所能接收的字节数。如果 TTY输出队列中剩余的字节超过 100字节(SLIP_HIWAT)，则slstart返回而不是将另一分组的字节添加到队列中。当传输完所有字节，输出设备产生一个中断，并且当输出列表为空时， TTY子系统调用slstart。

如果TTY输出队列为空，则一个分组从sc_fastq中退队，或者，若sc_fastq为空，则从if_snd队列中退队，这样在其他分组前传输所有交互的分组。

没有标准的SNMP变量来统计根据TOS字段排队的分组。在353行的XXX注释表示SLIP驱动程序在if_omcasts中统计低时延分组数，而不是多播分组数。

359-383 如果SLIP接口带有BPF，slstart在任何首部压缩前为输出分组产生一个备份。这个备份存储在bpfbuf数组的栈中。

384-388 如果允许压缩，并且分组包含一个TCP报文段，则sloutput调用sl_compress_tcp来压缩这个分组。得到的分组类型被返回，并与IP首部的第一个字节(29.13节)进行逻辑或运算。

389-398 压缩的首部现在复制到BPF首部，并且方向标记为SLIPDIR_OUT。完整的BPF分组传给bpf_tap。

483-484 如果for循环终止，则slstart返回。

if_sl.c

```
318 void
319 slstart(tp)
320 struct tty *tp;
321 {
322     struct sl_softc *sc = (struct sl_softc *) tp->t_sc;
323     struct mbuf *m;
324     u_char *cp;
325     struct ip *ip;
326     int     s;
327     struct mbuf *m2;
328     u_char  bpfbuf[SLMTU + SLIP_HDRLEN];
329     int     len;
330     extern int cfreecount;

331     for (;;) {
332         /*
333          * If there is more in the output queue, just send it now.
334          * We are being called in lieu of ttstart and must do what
335          * it would.
336          */
337         if (tp->t_outq.c_cc != 0) {
338             (*tp->t_oproc) (tp);
339             if (tp->t_outq.c_cc > SLIP_HIWAT)
340                 return;
341         }
342         /*
343          * This happens briefly when the line shuts down.
344          */
345         if (sc == NULL)
346             return;

347         /*
348          * Get a packet and send it to the interface.
349          */
350         s = splimp();
351         IF_DEQUEUE(&sc->sc_fastq, m);
352         if (m)
353             sc->sc_if.if_omcasts++;     /* XXX */
354         else
355             IF_DEQUEUE(&sc->sc_if.if_snd, m);
356         splx(s);
357         if (m == NULL)
358             return;
```

图5-16 函数slstart：分组退队

```
359         /*
360          * We do the header compression here rather than in sloutput
361          * because the packets will be out of order if we are using TOS
362          * queueing, and the connection id compression will get
363          * munged when this happens.
364          */
365         if (sc->sc_bpf) {
366             /*
367              * We need to save the TCP/IP header before it's
368              * compressed.  To avoid complicated code, we just
369              * copy the entire packet into a stack buffer (since
370              * this is a serial line, packets should be short
371              * and/or the copy should be negligible cost compared
372              * to the packet transmission time).
373              */
374             struct mbuf *m1 = m;
375             u_char *cp = bpfbuf + SLIP_HDRLEN;
376             len = 0;
377             do {
378                 int     mlen = m1->m_len;
379                 bcopy(mtod(m1, caddr_t), cp, mlen);
380                 cp += mlen;
381                 len += mlen;
382             } while (m1 = m1->m_next);
383         }
384         if ((ip = mtod(m, struct ip *))->ip_p == IPPROTO_TCP) {
385             if (sc->sc_if.if_flags & SC_COMPRESS)
386                 *mtod(m, u_char *) |= sl_compress_tcp(m, ip,
387                                                       &sc->sc_comp, 1);
388         }
389         if (sc->sc_bpf) {
390             /*
391              * Put the SLIP pseudo-"link header" in place.  The
392              * compressed header is now at the beginning of the
393              * mbuf.
394              */
395             bpfbuf[SLX_DIR] = SLIPDIR_OUT;
396             bpfcopy(mtod(m, caddr_t), &bpfbuf[SLX_CHDR], CHDR_LEN);
397             bpf_tap(sc->sc_bpf, bpfbuf, len + SLIP_HDRLEN);
398         }

                    /* packet output code */

483     }
484 }
```

if_sl.c

图5-16 (续)

slstart的下一部分(图5-17)在系统存储器容量不足时丢弃分组，并且采用一种简单的技术来丢弃由于串行线上的噪声产生的数据。这些代码在图 5-16中忽略了。

399-409 如果系统缺少clist结构，则分组被丢弃，并且作为一个冲突被统计。通过不断地循环而不是返回，slstart快速地丢弃所有剩余的排队输出的分组。由于设备仍然有太多字节为输出排队，每次迭代都要丢弃一个分组。高层协议必须检测丢失的分组并重传它们。

410-418 如果TTY输出队列为空，则通信线路可能有一段时间空闲，并且接收方在另一端

可能接收了线路噪声产生的无关数据。slstart在输出队列中放置一个额外的 SLIP END字符。一个长度为0的帧或一个由线路噪声产生的帧应该被接收方 SLIP接口或IP协议丢弃。

if_sl.c

```
399         sc->sc_if.if_lastchange = time;
400         /*
401          * If system is getting low on clists, just flush our
402          * output queue (if the stuff was important, it'll get
403          * retransmitted).
404          */
405         if (cfreecount < CLISTRESERVE + SLMTU) {
406             m_freem(m);
407             sc->sc_if.if_collisions++;
408             continue;
409         }
410         /*
411          * The extra FRAME_END will start up a new packet, and thus
412          * will flush any accumulated garbage.  We do this whenever
413          * the line may have been idle for some time.
414          */
415         if (tp->t_outq.c_cc == 0) {
416             ++sc->sc_if.if_obytes;
417             (void) putc(FRAME_END, &tp->t_outq);
418         }
```

if_sl.c

图5-17 函数slstart ：资源缺乏和线路噪声

图5-18说明了这个丢弃线路噪声的技术，它来源于由 Phil Karn撰写的RFC 1055。在图5-18中，传输第二个帧结束符 (END)，因为线路空闲了一段时间。由噪声产生的无效帧和这个END字节被接收系统丢弃。

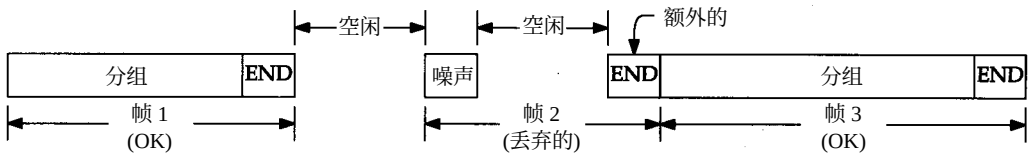


图5-18 Karn的丢弃SLIP线路噪声的方法

在图5-19中，线路上没有噪声并且0长度帧被接收系统丢弃。

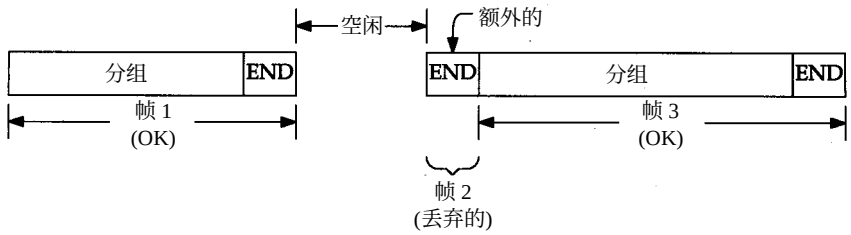


图5-19 无噪声的Karn方法

slstart的下一部分(图5-20)将数据从一个mbuf传给TTY设备的输出队列。

419-467 在这部分的外部while循环对链表中的每个mbuf执行一次。中间的while循环将数据从每个mbuf传给输出设备。内部的while循环不断递增cp，直到它找到一个END或ESC字符。b_to_q传输bp到cp之间的数据。END和ESC字符被转义，并且两次通过调用 putc放

入队列。中间的循环直到 mbuf的所有字节都传给 TTY设备输出队列才停止。图 5-21说明了对包含了一个SLIP END字符和一个SLIP ESC字符的mbuf的处理。

bp标记用b_to_q传输的mbuf的第一部分的开始， cp标记这个部分的结束。 ep标记这个mbuf中数据的结束位置。

if_sl.c

```
419         while (m) {
420             u_char *ep;

421             cp = mtod(m, u_char *);
422             ep = cp + m->m_len;
423             while (cp < ep) {
424                 /*
425                  * Find out how many bytes in the string we can
426                  * handle without doing something special.
427                  */
428                 u_char *bp = cp;

429                 while (cp < ep) {
430                     switch (*cp++) {
431                     case FRAME_ESCAPE:
432                     case FRAME_END:
433                         --cp;
434                         goto out;
435                     }
436                 }
437               out:
438                 if (cp > bp) {
439                     /*
440                      * Put n characters at once
441                      * into the tty output queue.
442                      */
443                     if (b_to_q((char *) bp, cp - bp,
444                                &tp->t_outq))
445                         break;
446                     sc->sc_if.if_obytes += cp - bp;
447                 }
448                 /*
449                  * If there are characters left in the mbuf,
450                  * the first one must be special..
451                  * Put it out in a different form.
452                  */
453                 if (cp < ep) {
454                     if (putc(FRAME_ESCAPE, &tp->t_outq))
455                         break;
456                     if (putc(*cp++ == FRAME_ESCAPE ?
457                              TRANS_FRAME_ESCAPE : TRANS_FRAME_END,
458                              &tp->t_outq)) {
459                         (void) unputc(&tp->t_outq);
460                         break;
461                     }
462                     sc->sc_if.if_obytes += 2;
463                 }
464             }
465             MFREE(m, m2);
466             m = m2;
467         }
```

if_sl.c

图5-20 函数slstart ：传输分组

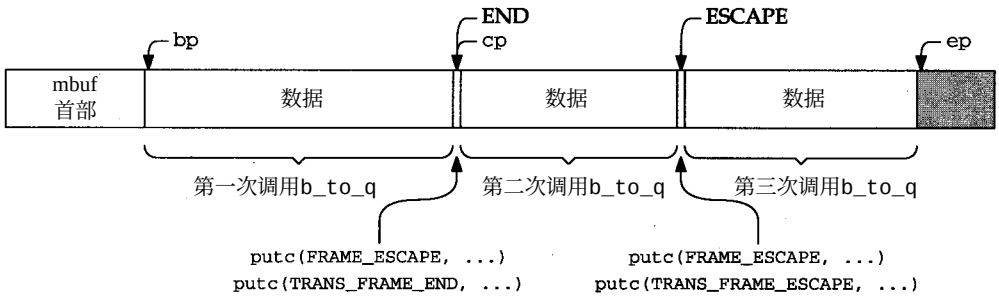


图5-21 单个mbuf的SLIP传输

如果b_to_q或putc失败(即，数据不能在TTY设备排队)，则break导致slstart退出内部while循环。这种失败表示内核 clist资源用完。在每个 mbuf被复制到TTY设备后，或者当一个差错发生时， mbuf被释放， m增加到链表的下一个 mbuf，并且外部while循环继续执行直到链表中所有mbuf被处理。

图5-22显示了slstart完成输出帧的处理。

if_sl.c

```
468         if (putc(FRAME_END, &tp->t_outq)) {
469             /*
470              * Not enough room.  Remove a char to make room
471              * and end the packet normally.
472              * If you get many collisions (more than one or two
473              * a day) you probably do not have enough clists
474              * and you should increase "nclist" in param.c.
475              */
476             (void) unputc(&tp->t_outq);
477             (void) putc(FRAME_END, &tp->t_outq);
478             sc->sc_if.if_collisions++;
479         } else {
480             ++sc->sc_if.if_obytes;
481             sc->sc_if.if_opackets++;
482         }
```

if_sl.c

图5-22 函数slstart ：帧结束处理

468-482 当外部while循环处理完对输出队列中的字节排队时，控制到达这段代码。驱动程序发送一个SLIP END字符，来终止这个帧。

如果这些字节在排队时发生差错，则输出帧无效，并会因为“无效的检验和”或“无效的长度”被接收系统检测出来。

无论这个帧是不是因为一个差错而终止，如果 END字符没有填充到输出队列中，队列的最后一个字符就要被丢弃，并且 slstart将使这个帧结束。这保证传输了一个 END字符。这个无效帧在目标站被丢弃。

5.3.6 SLIP分组丢失

SLIP接口提供了一个尽最大努力服务的好例子。如果 TTY超载，则SLIP丢弃分组；在分组开始传输后，如果资源不可用，则它截断分组，并且为了检测和丢弃线路噪声插入无关的空分组。对以上的每一种情况都不产生差错报文。 SLIP依靠IP层和运输层来检测损坏的和丢失的分组。

在一个路由器上从一个高速接口例如以太网，发送帧到一个低速的 SLIP线路上。如果发送方不能意识到瓶颈并相应调节数据速率，则会有大比例的分组被丢弃。在 25.11节我们会看到TCP是如何检测并对此响应的。应用程序使用一个无流量控制的协议，如 UDP，必须自己识别和响应这种情况(习题5.8)。

5.3.7 SLIP性能考虑

一个SLIP帧的MTU(SLMTU)、clist高水位标记(high-water mark)(SLIP_HIWAT)和SLIP的TOS排队策略都是用来设计交互通信的低速串行链，使得固有的时延最小。

1) 一个小的MTU能够改进交互数据的时延 (如敲键和回显)，但有损批量数据传输的吞吐量。一个大的MTU能改进批量数据的吞吐量，但增加了交互时延。 SLIP链路的另一个问题是键入一个字符就要有 40字节的开销来写入 TCP首部和IP首部的信息，这就增加了通信的时延。

   解决办法是挑选一个足够大的 MTU来提供好的交互响应时间和适当的批量数据吞吐量，并压缩TCP/IP首部来减小每个分组的负荷。 RFC 1144 [Jacobson 1990a]描述了一个压缩方案和时间计算，它为一个典型的 9600 b/s异步SLIP链路选择了一个数值为 296的MTU。我们在29.13节讨论压缩的SLIP(CSLIP)。卷1的2.10节和7.2节总结了这种定时考虑，并说明了在SLIP链路上的时延。

2) 如果有太多的字节缓存在clist中(因为SLIP_HIWAT设置得太高)，TOS排队会受到阻碍，因为新的交互式通信等在大量缓存数据的后面。如果 SLIP一次传给TTY驱动程序一个字节(因为SLIP_HIWAT设置得太低)，设备为每个字节调用 slstart，并在每个字节传输后线路空闲一段时间。把SLIP_HIWAT设置为100可使在设备排队的数据量最小化，并且减小了TTY子系统调用slstart的频率，大约每100字符必须调用slstart一次。

3) 如前所述， SLIP驱动程序提供了TOS排队，其策略是先从 sc_fastq队列中发送交互式通信数据，然后在标准接口队列 if_snd中发送其他的通信数据。

5.3.8 slclose函数

为了完整性，我们显示函数 slclose。当slattach程序关闭SLIP的TTY设备，并且中断对远程系统的连接时，调用它。

if_sl.c

```
210 void
211 slclose(tp)
212 struct tty *tp;
213 {
214     struct sl_softc *sc;
215     int     s;

216     ttywflush(tp);
217     s = splimp();               /* actually, max(spltty, splnet) */
218     tp->t_line = 0;
219     sc = (struct sl_softc *) tp->t_sc;
220     if (sc != NULL) {
221         if_down(&sc->sc_if);
222         sc->sc_ttyp = NULL;
223         tp->t_sc = NULL;
```

图5-23 函数slclose

```
224             MCLFREE((caddr_t) (sc->sc_ep - SLBUFSIZE));
225             sc->sc_ep = 0;
226             sc->sc_mp = 0;
227             sc->sc_buf = 0;
228         }
229         splx(s);
230 }
```

if_sl.c

图5-23 (续)

210-230 tp指向要关闭的TTY设备。slclose清除任何残留在串行设备中的数据，中断TTY和网络处理，并且将TTY复位到默认的线路规程。如果TTY设备被连接到一个SLIP接口，则关闭这个接口，在这两个结构间的链接被切断，与此接口关联的mbuf簇被释放，并且指向现在被丢弃的簇的指针被复位。最后，splx重新允许TTY中断和网络中断。

5.3.9 sltioctl函数

回忆一下，SLIP在内核中有两种作用：

- 作为一个网络接口；
- 作为一个TTY线路规程。

图5-7显示了slioctl处理通过一个插口描述符发送给一个SLIP接口的ioctl命令。在4.4节中，我们显示了ifioctl是如何调用slioctl的。我们会看到一个处理ioctl命令的相似模型，并且在后面的章节中会讨论到。

图5-7还表示了sltioctl处理发送给与一个SLIP网络接口关联的TTY设备的ioctl命令。这个被sltioctl识别的命令显示在图5-24中。

命 令	参 数	函 数	说 明
SLIOCGUNIT	int *	sltioctl	返回与TTY设备关联的接口联合

图5-24 sltioctl 命令

函数sltioctl显示在图5-25中。

```
236 int
237 sltioctl(tp, cmd, data, flag)
238 struct tty *tp;
239 int     cmd;
240 caddr_t data;
241 int     flag;
242 {
243     struct sl_softc *sc = (struct sl_softc *) tp->t_sc;

244     switch (cmd) {
245     case SLIOCGUNIT:
246         *(int *) data = sc->sc_if.if_unit;
247         break;

248     default:
249         return (-1);
250     }
251     return (0);
252 }
```

if_sl.c

图5-25 函数sltioctl

236-252 tty结构的t_sc指针指向关联的sl_softc结构。这个SLIP接口的设备号从if_unit被复制到*data，它最后返回给进程(17.5节)。

当系统被初始化时，slattach初始化if_unit，并且当slattach程序为此TTY设备选择SLIP线路规程时，slopen初始化t_sc。因为一个TTY设备和一个SLIP sl_softc结构间的关系是在运行时建立的，一个进程能通过SLIOCGUNIT命令发现所选择的接口结构。

5.4 环回接口

任何发送给环回接口(图5-26)的分组立即排入输入队列。接口完全用软件实现。

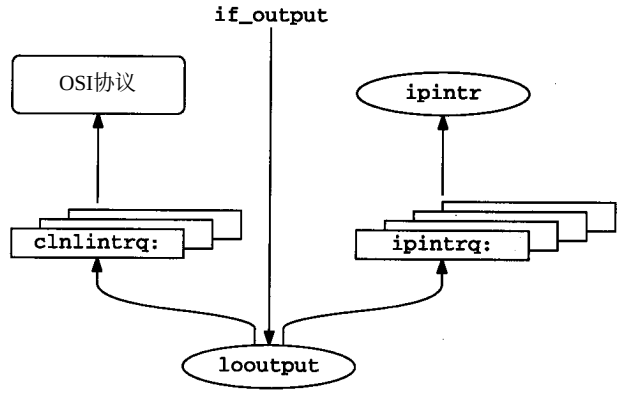


图5-26 环回设备驱动程序

环回接口的if_output指向的函数looutput，将输出分组放置到分组的目的地址指明的协议的输入队列中。

我们已经看到当设备被设置为IFF_SIMPLEX时，ether_output会调用looutput来排队一个输出广播分组。在第 12章中，我们会看到多播分组也可能以这种方式环回。looutput显示在图5-27中。

if_loop.c

```
57 int
58 looutput(ifp, m, dst, rt)
59 struct ifnet *ifp;
60 struct mbuf *m;
61 struct sockaddr *dst;
62 struct rtentry *rt;
63 {
64     int     s, isr;
65     struct ifqueue *ifq = 0;

66     if ((m->m_flags & M_PKTHDR) == 0)
67         panic("looutput no HDR");
68     ifp->if_lastchange = time;
69     if (loif.if_bpf) {
70         /*
71          * We need to prepend the address family as
72          * a four byte field.  Cons up a dummy header
73          * to pacify bpf.  This is safe because bpf
74          * will only read from the mbuf (i.e., it won't
75          * try to free it or keep a pointer a to it).
76          */
```

图5-27 函数looutput

```
77          struct mbuf m0;
78          u_int   af = dst->sa_family;

79          m0.m_next = m;
80          m0.m_len = 4;
81          m0.m_data = (char *) &af;

82          bpf_mtap(loif.if_bpf, &m0);
83      }
84      m->m_pkthdr.rcvif = ifp;

85      if (rt && rt->rt_flags & (RTF_REJECT | RTF_BLACKHOLE)) {
86          m_freem(m);
87          return (rt->rt_flags & RTF_BLACKHOLE ? 0 :
88                  rt->rt_flags & RTF_HOST ? EHOSTUNREACH : ENETUNREACH);
89      }
90      ifp->if_opackets++;
91      ifp->if_obytes += m->m_pkthdr.len;
92      switch (dst->sa_family) {
93      case AF_INET:
94          ifq = &ipintrq;
95          isr = NETISR_IP;
96          break;

97      case AF_ISO:
98          ifq = &clnlintrq;
99          isr = NETISR_ISO;
100         break;

101     default:
102         printf("lo%d: can't handle af%d\n", ifp->if_unit,
103                dst->sa_family);
104         m_freem(m);
105         return (EAFNOSUPPORT);
106     }
107     s = splimp();
108     if (IF_QFULL(ifq)) {
109         IF_DROP(ifq);
110         m_freem(m);
111         splx(s);
112         return (ENOBUFS);
113     }
114     IF_ENQUEUE(ifq, m);
115     schednetisr(isr);
116     ifp->if_ipackets++;
117     ifp->if_ibytes += m->m_pkthdr.len;
118     splx(s);
119     return (0);
120 }
```

if_loop.c

图5-27 (续)

57-66 looutput的参数同ether_output一样，因为都是通过它们的 ifnet结构中的if_output指针直接调用的。 ifp，指向输出接口的 ifnet结构的指针； m，要发送的分组；dst，分组的目的地址； rt，路由信息。如果链表中的第一个 mbuf不包含一个分组，looutput调用panic。

图5-28所示的是一个BPF环回分组的逻辑格式。

69-83 驱动程序在堆栈上的m0中构造BPF环回分组，并且把m0连接到包含原始分组的mbuf

链表中。注意m0的声明不同往常。它是一个mbuf，而不是一个mbuf指针。m0的m_data指向af，它也分配在这个堆栈中。图5-29显示了这种安排。

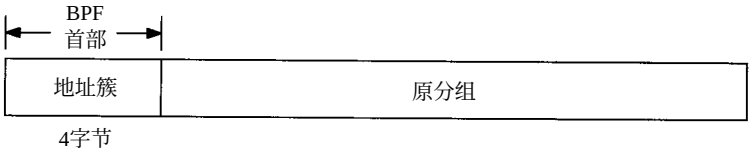


图5-28 BPF环回分组：逻辑格式

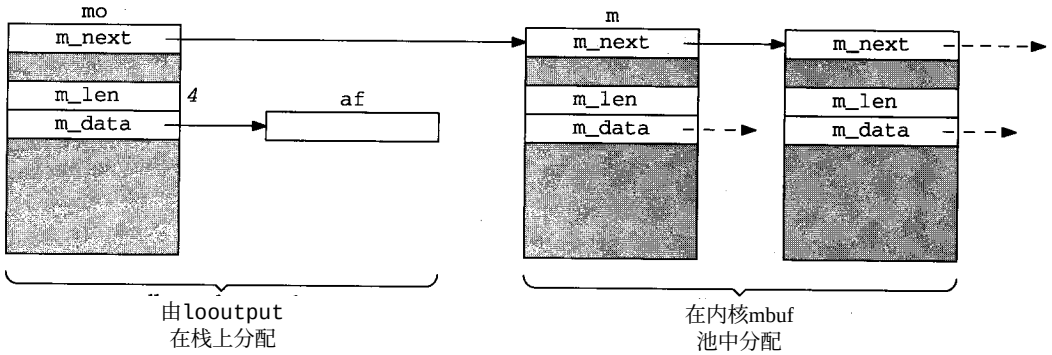


图5-29 BPF环回分组：mbuf格式

looutput将目的地址族复制到af，并且将新mbuf链表传递给bpf_mtap，去处理这个分组。与bpf_tap相比，它在一个单独的连续缓存中接收这个分组而不是在一个 mbuf链表中。如图中注释所示，BPF从来不释放一个链表中的mbuf，因此将m0 (它指向栈中的一个mbuf)传给bpf_mtap是安全的。

84-89 looutput剩下的代码包含input对此分组的处理。虽然这是一个输出函数，但分组被环回到输入。首先，m->m_pkthdr.rcvif设置为指向接收接口。如果调用方提供一个路由项，looutput检查是否它指示此分组应该被拒绝 (RTF_REJECT)或直接被丢弃(RTF_BLACKHOLE)。通过丢弃mbuf并返回0来实现一个黑洞。从调用者看来就好像分组已经被传输了。要拒绝一个分组，如果路由是一个主机，则 looutput返回EHOSTUNREACH；如果路由是一个网络则返回ENETUNREACH。

各种RTF_*xxx*标志在图18-25中描述。

90-120 然后looutput通过检查分组目的地址中的sa_family来选择合适的协议输入队列和软件中断。接着把识别的分组进行排队，并用 schednetisr来调度一个软件中断。

5.5 小结

我们讨论了两个剩下的接口，它们在书中多次引用：sl0，一个SLIP接口；lo0，标准的环回接口。

我们显示了在SLIP接口和SLIP线路规程之间的关系，讨论了 SLIP封装方法，并且讨论了TOS处理交互式通信和SLIP驱动程序的其他性能考虑。

我们显示了环回接口是如何按目的地址分用输出分组及将分组放到相应的输入队列中去。

习题

5.1 为什么环回接口没有输入函数？

5.2 你认为为什么图5-27中的m0要分配在堆栈中？

5.3 分析一个19 200 b/s的串行线的SLIP特性。对于这个线路，SLIP MTU应该改变吗？

5.4 导出一个根据串行线速率选择SLIP MTU的公式。

5.5 如果一个分组对于SLIP输入缓存太大，会发生什么情况？

5.6 一个`slinput`的早期版本，当一个分组在输入缓存溢出时，不将`SC_ERROR`置位。在这种情况下如何检测这种差错？

5.7 在图4-31中`le`被下标为`ifp->if_unit`的`le_softc`数组项初始化。你能想出另一种初始化`le`的方法吗？

5.8 当分组因为网络瓶颈被丢弃时，一个UDP应用程序如何知道？

第6章 IP 编 址

6.1 引言

本章讨论Net/3如何管理IP地址信息。我们从`in_ifaddr`和`sockaddr_in`结构开始，它们基于通用的`ifaddr`和`sockaddr`结构。

本章其余部分讨论IP地址的指派和几个查询接口数据结构与维护 IP地址的实用函数。

6.1.1 IP地址

虽然我们假设读者熟悉基本的Internet编址系统，仍然有几点值得指出。

在IP模型中，地址是指派给一个系统 (一个主机或路由器)中的网络接口而不是系统本身。在系统有多个接口的情况下，系统有多重初始地址，并有多个 IP地址。一个路由器被定义为有多重初始地址。如我们所看到的，这个体系特点有几个小分支。

IP地址定义了5类。A、B和C类地址支持单播通信。 D类地址支持IP多播。在一个多播通信中，一个单独的源方发送一个数据报给多个目标方。 D类地址和多播协议在第 12章说明。 E类地址是试验用的。接收的E类地址分组被不参与试验的主机丢弃。

我们强调IP多播和硬件多播间的区别是重要的。硬件多播的特点是数据链路硬件用来将帧传输给多个硬件接口。有些网络硬件，如以太网，支持数据链路多播。其他硬件可能不支持。

IP多播是一个在IP系统内实现的软件特性，将分组传输给多个可能在 Internet中任何位置的IP地址。

我们假设读者熟悉IP网络的子网划分 (RFC 950 [Mogul and Postel 1985]和卷1的第3章)。我们会看到每个网络接口有一个相关的子网掩码，它是判断一个分组是否到达它最后的目的地或还需要被转发的关键。通常，当提及一个 IP地址的网络部分时，我们包括任何可能定义的子网。当需要区分网络和子网时，我们就要明确地指出来。

环回网络，127.0.0.0，是一个特殊的A类网络。这种格式的地址是不会出现在一个主机的外部的。发送到这个网络的分组被环回并被这个主机接收。

> RFC 1122要求所有在环回网络中的地址被正确地处理。因为环回接口必须指派一个地址，很多系统选择 127.0.0.1作为环回地址。如果系统不能正确识别，像127.0.0.2这样的地址可能不能被路由到环回接口而被传输到一个连接的网络，这是不允许的。有些系统可能正确地路由这个到环回接口的分组，但由于目标地址与地址127.0.0.1不匹配，分组被丢弃。

图18-2显示了一个Net/3系统配置为拒绝接收发送到一个不是 127.0.0.1的环回地址的分组。

6.1.2 IP地址的印刷规定

我们通常以点分十进制数表示法来显示一个 IP地址。图6-1列出了每类IP地址的范围。

地址类	范　围	类　型
A	0.0.0.0到127.255.255.255	单播
B	128.0.0.0到191.255.255.255	
C	192.0.0.0到223.255.255.255	
D	224.0.0.0到239.255.255.255	多播
E	240.0.0.0到247.255.255.255	试验性

图6-1　不同IP地址类的范围

对于我们的有些例子，子网字段不按一个字节对齐 (即，一个网络/子网/主机在一个B类网络中分为16/11/5)。从点分十进制数表示法很难表示这样的地址，因此我们还是用方块图来说明IP地址的内容。我们用三个部分显示每个地址：网络、子网和主机。每个部分的阴影指示它的内容。图6-2用我们网络示例(1.14节)中的主机sun的以太网接口来同时说明块表示法和点分十进制数表示法。

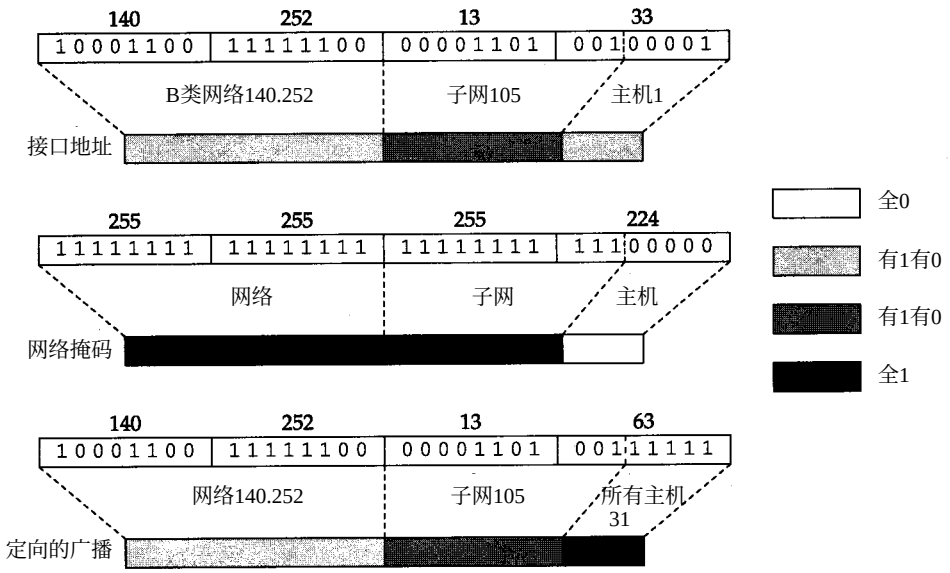


图6-2　可选的IP地址表示法

当地址的一个部分不是全为0或1时，我们使用两个中等程度的阴影。有两种中等程度的阴影，这样我们就能区分网络和子网部分或用来显示如图 6-31所示的地址组合。

6.1.3　主机和路由器

在一个Internet上的系统通常能划分为两类：主机和路由器。一个主机通常有一个网络接口，并且是一个IP分组的源或目标方。一个路由器有多个网络接口，当分组向它的目标方移动时将分组从一个网络转发到下一个网络。为执行这个功能，路由器用各种专用路由协议来交换关于网络拓扑的信息。IP路由问题比较复杂，在第18章开始讨论它们。

如果一个有多个网络接口的系统不在网络接口间路由分组，仍然叫一个主机。一个系统可能既是一个主机又是一个路由器。这种情况经常发生在当一个路由器提供运输层服务如用

于配置的Telnet访问，或用于网络管理的SNMP时。当区分一个主机和路由器间的意义并不重要时，我们使用术语系统。

不谨慎地配置一个路由器会干扰一个网络的正常运转，因此 RFC 1122规定一个系统必须默认为一个主机来操作，并且必须显式地由一个管理员来配置作为一个路由器操作。这样做是不鼓励管理员将通用主机作为路由器来操作而没有仔细地配置。在 Net/3中，如果全局整数`ipforwarding`不为0，则一个系统作为一个路由器；如果`ipforwarding`为0(默认)，则系统作为一个主机。

在Net/3中，一个路由器通常称为网关，虽然术语网关现在更多的是与一个提供应用层路由的系统相关，如一个电子邮件网关，而不是转发 IP分组的系统。我们在本书中使用术语路由器，并假设`ipforwarding`非0。在编译Net/3内核期间，当GATEWAY被定义时，我们还有条件地包括所有代码，它们将`ipforwarding`定义为1。

6.2 代码介绍

图6-3所列的两个头文件和两个C文件包含本章中讨论的结构定义和实用函数。

文 件	说 明
`netinet/in.h`	Internet地址定义
`netinet/in_var.h`	Internet接口定义
`netinet/in.c`	Internet初始化和实用函数
`netinet/if.c`	Internet接口实用函数

图6-3 本章讨论的文件

全局变量

图6-4所列的是本章中介绍的两个全局变量。

变 量	数据类型	说 明
`in_ifaddr`	`struct in_ifaddr *`	`in_ifaddr`结构列表的首部
`in_interfaces`	`int`	有IP能力的接口个数

图6-4 在本章中介绍的全局变量

6.3 接口和地址小结

在本章讨论的所有接口和地址结构的一个例子配置如图 6-5所示。

图6-5显示了我们的三个接口例子：以太网接口、 SLIP接口和环回接口。它们都有一个链路层地址作为地址列表中的第一个结点。显示的以太网接口有两个 IP地址， SLIP接口有一个IP地址，并且环回接口有一个IP地址和一个OSI地址。

注意所有的IP地址被链接到`in_ifaddr`列表中，并且所有链路层地址能从`ifnet_addrs`数组访问。

为了清楚起见，图6-5没有画出每个`ifaddr`结构中的指针`ifa_ifp`。这些指针回指包含此`ifaddr`结构的列表的首部`ifnet`结构。

接下来的部分讨论图6-5中的数据结构及用来查看和修改这些结构的 IP专用`ioctl`命令。

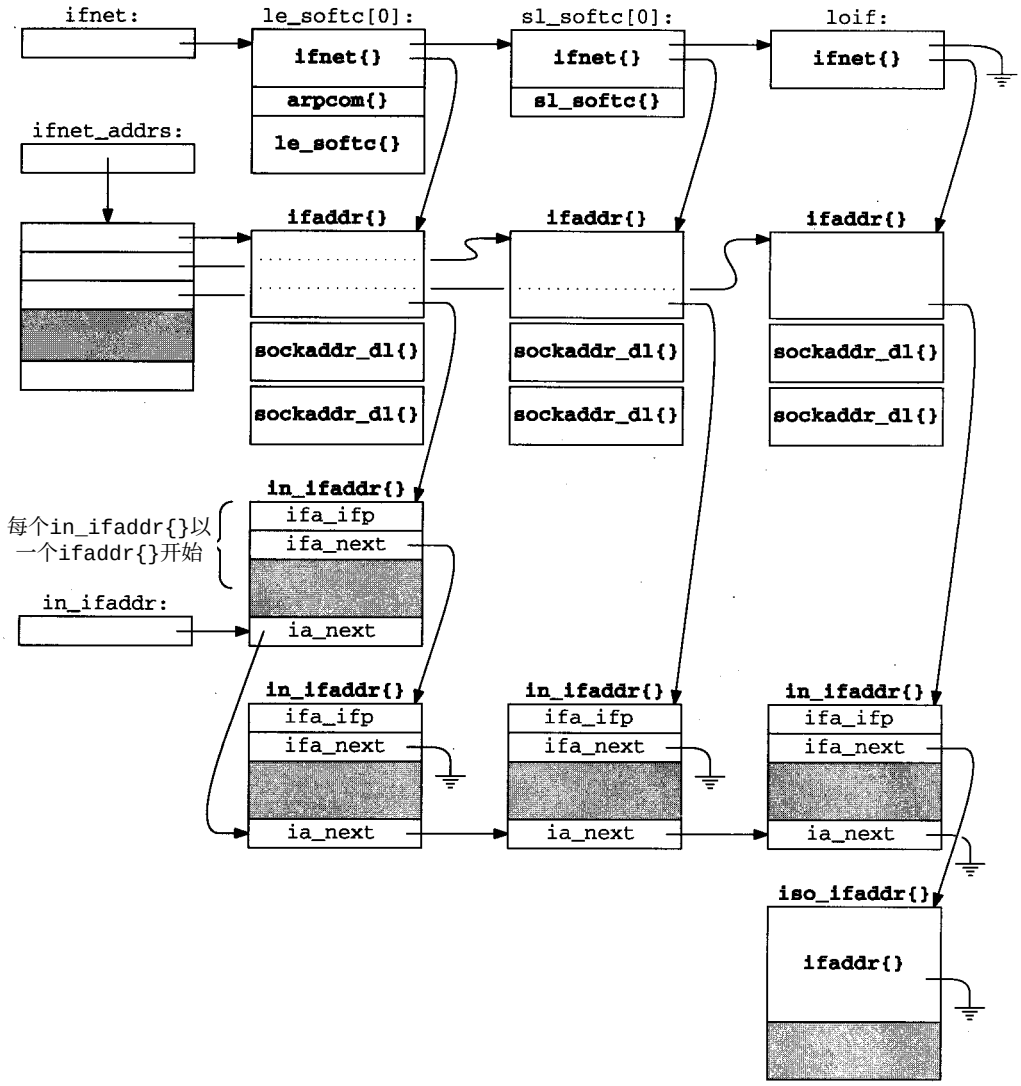


图6-5 接口和地址数据结构

6.4 sockaddr_in结构

我们在第3章讨论了通用的sockaddr和ifaddr结构。现在我们显示IP专用的结构：sockaddr_in和in_ifaddr。在Internet域中的地址存放在一个sockaddr_in结构：

68-70 由于历史原因，Net/3以网络字节序将Internet地址存储在一个in_addr结构中。这个结构只有一个成员s_addr，它包含这个地址。虽然这是多余和混乱的，但在Net/3中一直保持这种组织方式。

106-112 sin_len总是16(结构sockaddr_in的大小)，并且sin_family为AF_INET。sin_port是一个网络字节序(不是主机字节序)的16 bit值，用来分用运输层报文。sin_addr标识一个32 bit Internet地址。

图6-6显示了sockaddr_in的成员sin_port、sin_addr和sin_zero覆盖sockaddr

的成员sa_data。在Internet域中，sin_zero未用，但必须由全0字节组成(2.7节)。将它追加到sockaddr_in结构后面，以得到与一个sockaddr结构一样的长度。

in.h

```
 68 struct in_addr {
 69     u_long  s_addr;              /* 32-bit IP address, net byte order */
 70 };

106 struct sockaddr_in {
107     u_char  sin_len;             /* sizeof (struct sockaddr_in) = 16 */
108     u_char  sin_family;          /* AF_INET */
109     u_short sin_port;            /* 16-bit port number, net byte order */
110     struct in_addr sin_addr;
111     char    sin_zero[8];         /* unused */
112 };
```

in.h

图6-6 结构sockaddr_in

通常，当一个Internet地址存储在一个u_long中时，它以主机字节序存储，以便于地址的压缩和位操作。在in_addr结构(图6-7)中的s_addr是一个值得注意的例外。

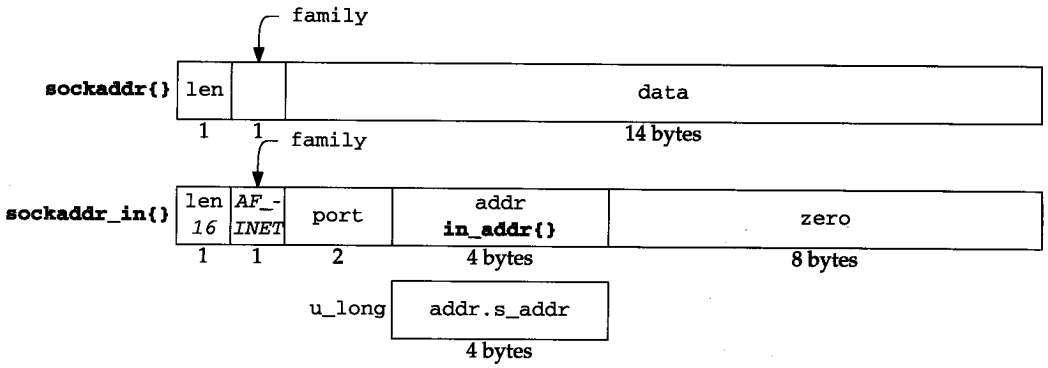


图6-7 一个sockaddr_in 结构(省略sin_)的组织

6.5 in_ifaddr结构

图6-8显示了为Internet协议定义的接口地址结构。对于每个指派给一个接口的 IP地址，分配了一个in_ifaddr结构，并且添加到接口地址列表中和IP地址全局列表中(图6-5)。

41-45 in_ifaddr开始是一个通用接口地址结构ia_ifa，跟着是IP专用成员。ifaddr结构显示在图3-15中。两个宏ia_ifp和ia_flags简化了对存储在通用ifaddr结构中的接口指针和接口地址标志的访问。ia_next维护指派给任意接口的所有Internet地址的链接列表。这个列表独立于每个接口关联的链路层ifaddr结构列表，并且通过全局列表in_ifaddr来访问。

46-54 其余的成员(除了ia_multiaddrs)显示在图6-9中，它显示了在我们的B类网络例子中sun的三个接口的相应值。地址按主机字节序以 u_long变量存储；变量in_addr和sockaddr_in按照网络字节序存储。sun有一个PPP接口，但显示在本表中的信息对于一个PPP或SLIP接口是一样的。

55-56 结构in_ifaddr的最后一个成员指向一个in_multi结构的列表(12.6节)，其中每项包含与此接口有关的一个IP多播地址。

in_var.h

```
41 struct in_ifaddr {
42         struct  ifaddr ia_ifa;          /* protocol-independent info    */
43 #define ia_ifp          ia_ifa.ifa_ifp
44 #define ia_flags        ia_ifa.ifa_flags
45         struct  in_ifaddr *ia_next;     /* next internet addresses list */
46         u_long  ia_net;                 /* network number of interface  */
47         u_long  ia_netmask;             /* mask of net part             */
48         u_long  ia_subnet;              /* subnet number, including net */
49         u_long  ia_subnetmask;          /* mask of subnet part          */
50         struct  in_addr ia_netbroadcast; /* to recognize net broadcasts */
51         struct  sockaddr_in ia_addr;    /* space for interface name     */
52         struct  sockaddr_in ia_dstaddr; /* space for broadcast addr     */
53 #define ia_broadaddr    ia_dstaddr
54         struct  sockaddr_in ia_sockmask; /* space for general netmask   */
55         struct  in_multi *ia_multiaddrs; /* list of multicast addresses */
56 };
```

in_var.h

图6-8 结构in_ifaddr

变量	类型	以太网	PPP	环回	说明
ia_addr	sockaddr_in	140.252.13.33	140.252.1.29	127.0.0.1	网络，子网和主机号
ia_net	u_long	140.252.0.0	140.252.0.0	127.0.0.0	网络号
ia_netmask	u_long	255.255.0.0	255.255.0.0	255.0.0.0	网络号掩码
ia_subnet	u_long	140.252.13.32	140.252.1.0	127.0.0.0	网络和子网号
ia_subnetmask	u_long	255.255.255.224	255.255.255.0	255.0.0.0	网络和子网掩码
ia_netbroadcast	in_addr	140.252.255.255	140.252.255.255	127.255.255.255	网络广播地址
ia_broadaddr	sockaddr_in	140.252.13.63			定向广播地址
ia_dstaddr	sockaddr_in		140.252.1.183	127.0.0.1	目的地址
ia_sockmask	sockaddr_in	255.255.255.224	255.255.255.0	255.0.0.0	像ia_subnetmask但是用网络字节序

图6-9 sun上的以太网、PPP和环回in_ifaddr 结构

6.6 地址指派

在第4章中，我们显示了当接口结构在系统初始化期间被识别时的初始化。在Internet协议能通过这个接口进行通信前，必须指派一个IP地址。一旦Net/3内核运行，程序ifconfig就配置这些接口，ifconfig通过在某个插口上的ioctl系统调用来发送配置命令。这通常通过/etc/netstart shell脚本来实现，这个脚本在系统引导时执行。

图6-10显示了本章中讨论的ioctl命令。命令相关的地址必须是此命令指定插口所支持的地址族类(即，你不能通过一个UDP插口配置一个OSI地址)。对于IP地址，ioctl命令在一

个UDP插口上发送。

命 令	参 数	函 数	说 明
SIOCGIFADDR	struct ifreq *	in_control	获得接口地址
SIOCGIFNETMASK	struct ifreq *	in_control	获得接口网络掩码
SIOCGIFDSTADDR	struct ifreq *	in_control	获得接口目标地址
SIOCGIFBRDADDR	struct ifreq *	in_control	获得接口广播地址
SIOCSIFADDR	struct ifreq *	in_control	设置接口地址
SIOCSIFNETMASK	struct ifreq *	in_control	设置接口网络掩码
SIOCSIFDSTADDR	struct ifreq *	in_control	设置接口目标地址
SIOCSIFBRDADDR	struct ifreq *	in_control	设置接口广播地址
SIOCDIFADDR	struct ifreq *	in_control	删除接口地址
SIOCAIFADDR	struct in_aliasreq *	in_control	添加接口地址

图6-10 接口ioctl 命令

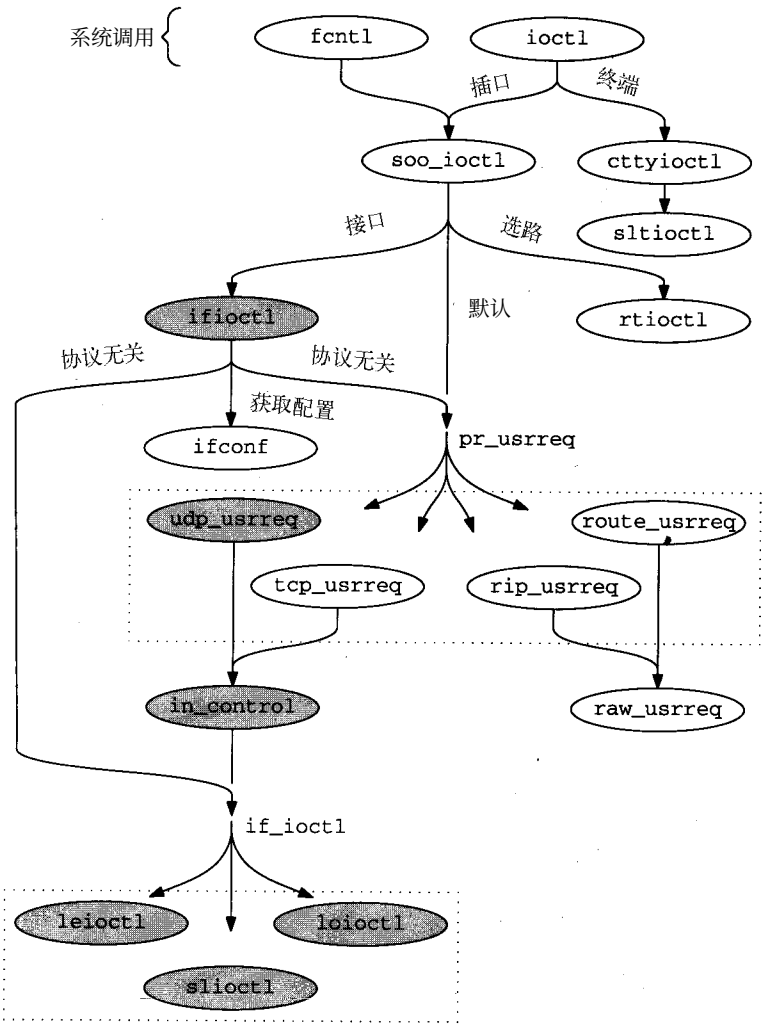


图6-11 本章中说明的ioctl 函数

获得地址信息的命令从SIOCG开始，设置地址信息的命令从SIOCS开始。SIOC代表socket ioctl，G代表get，而S代表set。

在第4章中，我们看到了5个与协议无关的ioctl命令。图6-10中的命令修改一个接口的相关地址信息。由于地址是特定协议使用的，因此，命令处理是与协议相关的。图 6-11强调了与这些命令关联的ioctl相关函数。

6.6.1 ifioctl函数

如图6-11所示，ifioctl将协议无关的ioctl命令传递给此插口关联协议的pr_usrreq函数。将控制交给udp_usrreq，并且又立即传给in_control，在in_control中进行大部分的处理。如果在一个TCP插口上发送同样的命令，控制最后也会到达in_control。图6-12再次显示了ifioctl中的default代码，第一次显示在图4-22中。

if.c

```
447     default:
448         if (so->so_proto == 0)
449             return (EOPNOTSUPP);
450         return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
451                                             cmd, data, ifp));
452     }
453     return (0);
454 }
```

if.c

图6-12 函数ifioctl ：特定协议的命令

447-454 函数将图6-10中所列ioctl命令的所有相关数据传给与请求指定的插口相关联的协议的用户请求函数。对于一个UDP插口，调用udp_usrreq。23.10节讨论udp_usrreq函数的细节。现在，我们仅需要查看udp_usrreq中的PRU_CONTROL代码：

```
if (req == PRU_CONTROL)
    return (in_control(so, (int)m, (caddr_t)addr, (struct ifnet *)control));
```

6.6.2 in_control函数

图6-11显示了通过soo_ioctl中的default或ifioctl中的与协议相关的情况，控制能到达in_control。在这两种情况中， udp_usrreq调用in_control，并返回in_control的返回值。图6-13显示了in_control。

132-145 so指向这个ioctl命令(由第二个参数cmd标识)指定的插口。第三个参数data指向命令所用或返回的数据(图6-10的第二列)。最后一个参数ifp为空(来自soo_ioctl的无接口ioctl)或指向结构ifreq或in_aliasreq中命名的接口(来自ifioctl的接口ioctl)。in_control初始化ifr和ifra来访问作为一个ifreq或in_aliasreq结构的data。

146-152 如果ifp指向一个ifnet结构，这个for循环找到与此接口关联的Internet地址列表中的第一个地址。如果发现一个地址， ia指向它的in_ifaddr结构；否则ia为空。

若ifp为空，cmd就不会匹配第一个switch中的任何情况；或第二个switch中任何非默认情况。在第二个switch中的default情况中，当ifp为空时，返回EOPNOTSUPP。

153-330 in_control中的第一个switch确保在第二个switch处理命令之前每个命令的前提条件都满足。在后面的章节会单独说明各个情况。

in.c

```
132 in_control(so, cmd, data, ifp)
133 struct socket *so;
134 int     cmd;
135 caddr_t data;
136 struct ifnet *ifp;
137 {
138     struct ifreq *ifr = (struct ifreq *) data;
139     struct in_ifaddr *ia = 0;
140     struct ifaddr *ifa;
141     struct in_ifaddr *oia;
142     struct in_aliasreq *ifra = (struct in_aliasreq *) data;
143     struct sockaddr_in oldaddr;
144     int     error, hostIsNew, maskIsNew;
145     u_long  i;

146     /*
147      * Find address for this interface, if it exists.
148      */
149     if (ifp)
150         for (ia = in_ifaddr; ia; ia = ia->ia_next)
151             if (ia->ia_ifp == ifp)
152                 break;

153     switch (cmd) {

                    /* establish preconditions for commands */

218     }
219     switch (cmd) {

                    /* perform the commands */

326     default:
327         if (ifp == 0 || ifp->if_ioctl == 0)
328             return (EOPNOTSUPP);
329         return ((*ifp->if_ioctl) (ifp, cmd, data));
330     }
331     return (0);
332 }
```

in.c

图6-13 函数in_control

如果在第二个switch中的default情况被执行，ifp指向一个接口结构；并且如果接口有一个if_ioctl函数，则in_control将ioctl命令传给这个接口进行设备的特定处理。

Net/3不定义任何会被default情况处理的接口命令。但是，一个特定设备的驱动程序可能会定义它自己的接口ioctl命令，并通过这个*case*来处理它们。

331-332 我们会看到这个switch语句中的很多情况都直接返回了。如果控制落到两个switch语句外，则in_control返回0。第二个switch中有几个case执行了跳出语句。

我们按照下面的顺序查看这个接口ioctl命令：

- 指派一个地址、网络掩码或目标地址；
- 指派一个广播地址；

- 取回一个地址、网络掩码、目标地址或广播地址；
- 给一个接口指派多播地址；
- 删除一个地址。

对于每组命令，在第一个switch语句中进行前提条件处理，然后在第二个switch语句中处理命令。

6.6.3 前提条件：SIOCSIFADDR、SIOCSIFNETMASK和SIOCSIFDSTADDR

图6-14显示了对SIOCSIFADDR、SIOCSIFNETMASK和SIOCSIFDSTADDR的前提条件检验。

in.c

```
166     case SIOCSIFADDR:
167     case SIOCSIFNETMASK:
168     case SIOCSIFDSTADDR:
169         if ((so->so_state & SS_PRIV) == 0)
170             return (EPERM);

171         if (ifp == 0)
172             panic("in_control");
173         if (ia == (struct in_ifaddr *) 0) {
174             oia = (struct in_ifaddr *)
175                 malloc(sizeof *oia, M_IFADDR, M_WAITOK);
176             if (oia == (struct in_ifaddr *) NULL)
177                 return (ENOBUFS);
178             bzero((caddr_t) oia, sizeof *oia);
179             if (ia = in_ifaddr) {
180                 for (; ia->ia_next; ia = ia->ia_next)
181                     continue;
182                 ia->ia_next = oia;
183             } else
184                 in_ifaddr = oia;
185             ia = oia;
186             if (ifa = ifp->if_addrlist) {
187                 for (; ifa->ifa_next; ifa = ifa->ifa_next)
188                     continue;
189                 ifa->ifa_next = (struct ifaddr *) ia;
190             } else
191                 ifp->if_addrlist = (struct ifaddr *) ia;

192             ia->ia_ifa.ifa_addr = (struct sockaddr *) &ia->ia_addr;
193             ia->ia_ifa.ifa_dstaddr
194                 = (struct sockaddr *) &ia->ia_dstaddr;
195             ia->ia_ifa.ifa_netmask
196                 = (struct sockaddr *) &ia->ia_sockmask;
197             ia->ia_sockmask.sin_len = 8;
198             if (ifp->if_flags & IFF_BROADCAST) {
199                 ia->ia_broadaddr.sin_len = sizeof(ia->ia_addr);
200                 ia->ia_broadaddr.sin_family = AF_INET;
201             }
202             ia->ia_ifp = ifp;
203             if (ifp != &loif)
204                 in_interfaces++;
205         }
206         break;
```

in.c

图6-14 函数in_control ：地址指派

1. 仅用于超级用户

166-172 如果这个插口不是由一个超级用户进程创建的，这些命令被禁止，并且 in_control返回EPERM。如果此请求没有关联的接口，内核调用 panic。由于如果ifioctl不能找到一个接口，它就返回(图4-22)，因此，panic从来不会被调用。

当一个超级用户进程创建一个插口时，socreate(图15-16)设置标志SS_PRIV。因为这里的检验是针对标志而不是有效的进程用户 ID的，所以一个设置用户 ID的根进程能创建一个插口，并且放弃它的超级用户权限，但仍然能发送有特权的 ioctl 命令。

2. 分配结构

173-191 如果ia为空，命令请求一个新的地址。in_control分配一个in_ifaddr结构，用bzero清除它，并且将它链接到系统的in_ifaddr列表中和此接口的if_addrlist列表中。

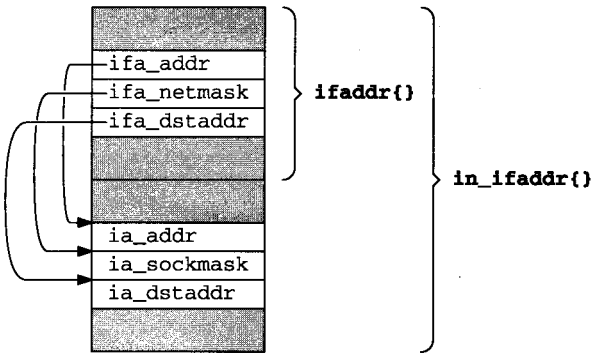


图6-15 被in_control 初始化后的一个in_ifaddr 结构

3. 初始化结构

192-206 代码的下一部分初始化in_ifaddr结构。首先，在此结构的ifaddr部分的通用指针被初始化为指向结构in_ifaddr中的结构sockaddr_in。必要时，此函数还初始化结构ia_sockmask和ia_broadaddr。图6-15说明了初始化后的结构in_ifaddr。

202-206 最后，in_control建立从in_ifaddr到此接口的ifnet结构的回指指针。

Net/3在in_interfaces中只统计非环回接口。

6.6.4 地址指派：SIOCSIFADDR

前提条件处理代码保证ia指向一个要被SIOCSIFADDR命令修改的in_ifaddr结构。图6-16显示了in_control第二个switch中处理这个命令的执行代码。

in.c

```
259     case SIOCSIFADDR:
260         return (in_ifinit(ifp, ia,
261                           (struct sockaddr_in *) &ifr->ifr_addr, 1));
```

in.c

图6-16 函数in_control ：地址指派

159-261 in_ifinit完成所有的工作。IP地址包含在ifreq结构(ifr_addr)里传递给in_ifinit。

6.6.5 in_ifinit函数

in_ifinit的主要步骤是：

- 将地址复制到此结构并将此变化通知硬件；

- 忽略原地址配置的任何路由；
- 为这个地址建立一个子网掩码；
- 建立一个默认路由到连接的网络(或主机)；
- 将此接口加入到所有主机组。

从图6-17开始分三个部分讨论这段代码。

353-357 in_ifinit的四个参数为：ifp，指向接口结构的指针；ia，指向要改变的in_ifaddr结构的指针；sin，指向请求的IP地址的指针；scrub，指示这个接口如果存在路由应该被忽略。i保存主机字节序的IP地址。

in.c

```
353 in_ifinit(ifp, ia, sin, scrub)
354 struct ifnet *ifp;
355 struct in_ifaddr *ia;
356 struct sockaddr_in *sin;
357 int     scrub;
358 {
359     u_long  i = ntohl(sin->sin_addr.s_addr);
360     struct sockaddr_in oldaddr;
361     int     s = splimp(), flags = RTF_UP, error, ether_output();

362     oldaddr = ia->ia_addr;
363     ia->ia_addr = *sin;
364     /*
365      * Give the interface a chance to initialize
366      * if this is its first address,
367      * and to validate the address if necessary.
368      */
369     if (ifp->if_ioctl &&
370         (error = (*ifp->if_ioctl) (ifp, SIOCSIFADDR, (caddr_t) ia))) {
371         splx(s);
372         ia->ia_addr = oldaddr;
373         return (error);
374     }
375     if (ifp->if_output == ether_output) {    /* XXX: Another Kludge */
376         ia->ia_ifa.ifa_rtrequest = arp_rtrequest;
377         ia->ia_ifa.ifa_flags |= RTF_CLONING;
378     }
379     splx(s);
380     if (scrub) {
381         ia->ia_ifa.ifa_addr = (struct sockaddr *) &oldaddr;
382         in_ifscrub(ifp, ia);
383         ia->ia_ifa.ifa_addr = (struct sockaddr *) &ia->ia_addr;
384     }
```

in.c

图6-17 函数in_ifinit：地址指派和路由初始化

1. 指派地址并通知硬件

358-374 in_control将原来的地址保存在oldaddr中，万一发生差错时，必须恢复它。如果接口定义了一个if_ioctl函数，则in_control调用它。相同接口的三个函数leioctl、slioctl和loioctl在下一节讨论。如果发生差错，恢复原来的地址，并且in_control返回。

2. 以太网配置

375-378 对于以太网设备，arp_rtrequest作为链路层路由函数被选择，并且设置RTF_CLONING标志。arp_rtrequest在21.13节讨论，而RTF_CLONING在19.4节的最后

讨论。如XXX注释所建议，在此加入代码以避免改变所有以太网驱动程序。

3. 忽略原来的路由

379-384 如果调用者要求已存在的路由被清除，原地址被重新连接到 ifa_addr，同时in_ifscrub找到并废除任何基于老地址的路由。in_ifscrub返回后，新地址被恢复。

in_ifinit显示在图6-18中的部分构造网络和子网掩码。

in.c

```
385     if (IN_CLASSA(i))
386         ia->ia_netmask = IN_CLASSA_NET;
387     else if (IN_CLASSB(i))
388         ia->ia_netmask = IN_CLASSB_NET;
389     else
390         ia->ia_netmask = IN_CLASSC_NET;
391     /*
392      * The subnet mask usually includes at least the standard network part,
393      * but may may be smaller in the case of supernetting.
394      * If it is set, we believe it.
395      */
396     if (ia->ia_subnetmask == 0) {
397         ia->ia_subnetmask = ia->ia_netmask;
398         ia->ia_sockmask.sin_addr.s_addr = htonl(ia->ia_subnetmask);
399     } else
400         ia->ia_netmask &= ia->ia_subnetmask;
401     ia->ia_net = i & ia->ia_netmask;
402     ia->ia_subnet = i & ia->ia_subnetmask;
403     in_socktrim(&ia->ia_sockmask);
```

in.c

图6-18 函数in_ifinit ：网络和子网掩码

4. 构造网络掩码和默认子网掩码

385-400 根据地址是一个A类、B类或C类地址，在ia_netmask中构造了一个尝试性网络掩码。如果这个地址没有子网掩码， ia_subnetmask和ia_sockmask被初始化为ia_netmask中的尝试性掩码。

如果指定了一个子网，in_ifinit将这个尝试性网络掩码和这个已存在的子网掩码进行逻辑与运算来获得一个新的网络掩码。这个操作可能会清除该尝试性网络掩码的一些 1 bit(它从来不设置0 bit，因为0逻辑与任何值都得到0)。在这种情况下，网络掩码比所考虑的地址类型所期望的要少一些1 bit。

这叫作超级联网，它在RFC 1519 [Fuller et al. 1993]中作了描述。一个超级网络是几个A类、B类或C类网络的一个群组。卷1的10.8节也讨论了超级联网。

一个接口默认配置为不划分子网 (即，网络和子网的掩码相同)。一个显式请求 (用SIOCSIFNETMASK或SIOCAIFADDR)用来允许子网划分(或超级联网)。

5. 构造网络和子网数量

401-403 网络和子网数量通过网络和子网掩码从新地址中获得。函数in_socktrim通过查找掩码中包含1 bit的最后一个字节来设置in_sockmask(是一个sockaddr_in结构)的长度。

图6-19显示了in_ifinit的最后一部分，它为接口添加了一个路由，并加入所有主机多播组。

6. 为主机或网络建立路由

404-422 下一步是为新地址所指定的网络创建一个路由。in_control从接口将路由度量

复制到结构in_ifaddr中。如果接口支持广播，则构造广播地址，并且把目的地址强制为分配给环回接口的地址。如果一个点对点接口没有一个指派给链路另一端的IP地址，则in_control在试图为这个无效地址建立路由前返回。

in_ifinit将flags初始化为RTF_UP，并与环回和点对点接口的RTF_HOST进行逻辑或。rtinit为此接口给这个网络(不设置RTF_HOST)或主机(设置RTF_HOST)安装一个路由。若rtinit安装成功，则设置ia_flags中的标志IFA_ROUTE，指示已给此地址安装了一个路由。

in.c

```
404     /*
405      * Add route for the network.
406      */
407     ia->ia_ifa.ifa_metric = ifp->if_metric;
408     if (ifp->if_flags & IFF_BROADCAST) {
409         ia->ia_broadaddr.sin_addr.s_addr =
410             htonl(ia->ia_subnet | ~ia->ia_subnetmask);
411         ia->ia_netbroadcast.s_addr =
412             htonl(ia->ia_net | ~ia->ia_netmask);
413     } else if (ifp->if_flags & IFF_LOOPBACK) {
414         ia->ia_ifa.ifa_dstaddr = ia->ia_ifa.ifa_addr;
415         flags |= RTF_HOST;
416     } else if (ifp->if_flags & IFF_POINTOPOINT) {
417         if (ia->ia_dstaddr.sin_family != AF_INET)
418             return (0);
419         flags |= RTF_HOST;
420     }
421     if ((error = rtinit(&(ia->ia_ifa), (int) RTM_ADD, flags)) == 0)
422         ia->ia_flags |= IFA_ROUTE;
423     /*
424      * If the interface supports multicast, join the "all hosts"
425      * multicast group on that interface.
426      */
427     if (ifp->if_flags & IFF_MULTICAST) {
428         struct in_addr addr;

429         addr.s_addr = htonl(INADDR_ALLHOSTS_GROUP);
430         in_addmulti(&addr, ifp);
431     }
432     return (error);
433 }
```

in.c

图6-19 函数in_ifinit：路由和多播组

7. 加入所有主机组

423-433 最后，一个有多播能力的接口当它被初始化时必须加入所有主机多播组。in_addmulti完成此工作，并在12.11节讨论。

6.6.6 网络掩码指派：SIOCSIFNETMASK

图6-20显示了网络掩码命令的处理。

in.c

```
262     case SIOCSIFNETMASK:
263         i = ifra->ifra_addr.sin_addr.s_addr;
264         ia->ia_subnetmask = ntohl(ia->ia_sockmask.sin_addr.s_addr = i);
265         break;
```

in.c

图6-20 函数in_control：网络掩码指派

262-265 in_control从ifreq结构中获取网络掩码，并将它以网络字节序保存在ia_sockmask中，以主机字节序保存在ia_subnetmask中。

6.6.7 目的地址指派：SIOCSIFDSTADDR

对于点对点接口，在链路另一端的系统的地址用SIOCSIFDSTADDR命令指定。图6-14显示了图6-21中的代码的前提条件处理。

in.c

```
236     case SIOCSIFDSTADDR:
237         if ((ifp->if_flags & IFF_POINTOPOINT) == 0)
238             return (EINVAL);
239         oldaddr = ia->ia_dstaddr;
240         ia->ia_dstaddr = *(struct sockaddr_in *) &ifr->ifr_dstaddr;
241         if (ifp->if_ioctl && (error = (*ifp->if_ioctl)
242                               (ifp, SIOCSIFDSTADDR, (caddr_t) ia))) {
243             ia->ia_dstaddr = oldaddr;
244             return (error);
245         }
246         if (ia->ia_flags & IFA_ROUTE) {
247             ia->ia_ifa.ifa_dstaddr = (struct sockaddr *) &oldaddr;
248             rtinit(&(ia->ia_ifa), (int) RTM_DELETE, RTF_HOST);
249             ia->ia_ifa.ifa_dstaddr =
250                 (struct sockaddr *) &ia->ia_dstaddr;
251             rtinit(&(ia->ia_ifa), (int) RTM_ADD, RTF_HOST | RTF_UP);
252         }
253         break;
```

in.c

图6-21 函数in_control：目的地址指派

236-245 只有点对点网络才有目的地址，因此对于其他网络，in_control返回EINVAL。将当前目的地址保存在oldaddr后，代码设置新地址，并通过函数if_ioctl通知硬件。如果发生差错，则恢复原地址。

246-253 如果地址原来有一个关联的路由，首先调用rtinit删除这个路由，并再次调用rtinit为新地址安装一个新路由。

6.6.8 获取接口信息

图6-22显示了命令SIOCSIFBRDADDR的前提条件处理，它同将接口信息返回给调用进程的ioctl命令一样。

in.c

```
207     case SIOCSIFBRDADDR:
208         if ((so->so_state & SS_PRIV) == 0)
209             return (EPERM);
210         /* FALLTHROUGH */

211     case SIOCGIFADDR:
212     case SIOCGIFNETMASK:
213     case SIOCGIFDSTADDR:
214     case SIOCGIFBRDADDR:
215         if (ia == (struct in_ifaddr *) 0)
216             return (EADDRNOTAVAIL);
217         break;
```

in.c

图6-22 函数in_control：前提条件处理

207-217 广播地址只能通过一个超级用户进程创建的插口来设置。命令SIOCSIFBRDADDR和4个SIOCG*xxx*命令仅当已经为此接口定义了一个地址时才起作用，在这种情况下，ia不会为空(ia被in_control设置，图6-13)。如果ia为空，返回EADDRNOTAVAIL。

这5个命令(4个*get*命令和一个*set*命令)的处理显示在图6-23中。

in.c

```
220     case SIOCGIFADDR:
221         *((struct sockaddr_in *) &ifr->ifr_addr) = ia->ia_addr;
222         break;

223     case SIOCGIFBRDADDR:
224         if ((ifp->if_flags & IFF_BROADCAST) == 0)
225             return (EINVAL);
226         *((struct sockaddr_in *) &ifr->ifr_dstaddr) = ia->ia_broadaddr;
227         break;

228     case SIOCGIFDSTADDR:
229         if ((ifp->if_flags & IFF_POINTOPOINT) == 0)
230             return (EINVAL);
231         *((struct sockaddr_in *) &ifr->ifr_dstaddr) = ia->ia_dstaddr;
232         break;

233     case SIOCGIFNETMASK:
234         *((struct sockaddr_in *) &ifr->ifr_addr) = ia->ia_sockmask;
235         break;

        /* processing for SIOCSIFDSTADDR command (Figure 6.21) */

254     case SIOCSIFBRDADDR:
255         if ((ifp->if_flags & IFF_BROADCAST) == 0)
256             return (EINVAL);
257         ia->ia_broadaddr = *(struct sockaddr_in *) &ifr->ifr_broadaddr;
258         break;
```

in.c

图6-23 函数in_control：处理

220-235 将单播地址、广播地址、目的地址或者网络掩码复制到ifreq结构。只有网络接口支持广播，广播地址才有效；并且只有点对点接口，目的地址才有效。

254-258 仅当接口支持广播，才从结构ifreq中复制广播地址。

6.6.9 每个接口多个IP地址

SIOCG*xxx*和SIOCS*xxx*命令只操作与一个接口关联的第一个IP地址——在in_control开头的循环找到的第一个地址(图6-25)。为支持每个接口的多个IP地址，必须用SIOCAIFADDR命令指派和配置其他的地址。实际上，SIOCAIFADDR能完成所有SIOCG*xxx*和SIOCS*xxx*命令能完成的操作。程序ifconfig使用SIOCAIFADDR来配置一个接口的所有地址信息。

如前所述，每个接口有多个地址便于在主机或网络改号时过渡。一个容错软件系统可能使用这个特性来准许一个备份系统充当一个故障系统的IP地址。

Net/3的ifconfig程序的-alias选项将存放在一个in_aliasreq中的其他地址的相关信息传递给内核，如图6-24所示。

in_var.h

```
59 struct in_aliasreq {
60     char    ifra_name[IFNAMSIZ];    /* interface name, e.g. "en0" */
61     struct sockaddr_in ifra_addr;
62     struct sockaddr_in ifra_broadaddr;
63 #define ifra_dstaddr ifra_broadaddr
64     struct sockaddr_in ifra_mask;
65 };
```

in_var.h

图6-24 结构in_aliasreq

59-65 注意，不像结构ifreq，在结构in_aliasreq中没有定义联合。在一个单独的ioctl调用中可以为SIOCAIFADDR指定地址、广播地址和掩码。

SIOCAIFADDR增加一个新地址或修改一个已存在地址的相关信息。SIOCDIFADDR删除匹配的IP地址的in_ifaddr结构。图6-25显示命令SIOCAIFADDR和SIOCDIFADDR的前提条件处理，它假设在in_control(图6-13)开头的循环已经将ia设置为指向与ifra_name(如果存在)指定的接口关联的第一个IP地址。

in.c

```
154     case SIOCAIFADDR:
155     case SIOCDIFADDR:
156         if (ifra->ifra_addr.sin_family == AF_INET)
157             for (oia = ia; ia; ia = ia->ia_next) {
158                 if (ia->ia_ifp == ifp &&
159                     ia->ia_addr.sin_addr.s_addr ==
160                     ifra->ifra_addr.sin_addr.s_addr)
161                     break;
162             }
163         if (cmd == SIOCDIFADDR && ia == 0)
164             return (EADDRNOTAVAIL);
165         /* FALLTHROUGH to Figure 6.14 */
```

in.c

图6-25 函数in_control：添加和删除地址

154-165 因为SIOCDIFADDR代码只查看*ifra的前两个成员，图6-25所示的代码用于处理SIOCAIFADDR(当ifra指向一个in_aliasreq结构时)和SIOCDIFADDR(当ifra指向一个ifreq结构时)。结构in_aliasreq和ifreq的前两个成员是一样的。

对于这两个命令，in_control开头的循环启动for循环不断地查找与ifra->ifra_addr指定的IP地址相同的in_ifaddr结构。对于删除命令，如果地址没有找到，则返回EADDRNOTAVAIL。

在这个处理删除命令的循环和检验后，控制落到我们在图6-14中讨论的代码之外。对于添加命令，图6-14的代码若找不到一个与in_aliasreq结构中地址匹配的地址，就分配一个新in_ifaddr结构。

6.6.10 附加IP地址：SIOCAIFADDR

这时ia指向一个新的in_ifaddr结构或一个包含与请求地址匹配的IP地址的旧in_ifaddr结构。SIOCAIFADDR的处理显示在图6-26中。

266-277 因为SIOCAIFADDR能创建一个新地址或修改一个已存在地址的相关信息，标志maskIsNew和hostIsNew跟踪变化的情况。这样，在这个函数结束时，如果必要，能更新路由。

in.c

```
266     case SIOCAIFADDR:
267         maskIsNew = 0;
268         hostIsNew = 1;
269         error = 0;
270         if (ia->ia_addr.sin_family == AF_INET) {
271             if (ifra->ifra_addr.sin_len == 0) {
272                 ifra->ifra_addr = ia->ia_addr;
273                 hostIsNew = 0;
274             } else if (ifra->ifra_addr.sin_addr.s_addr ==
275                         ia->ia_addr.sin_addr.s_addr)
276                 hostIsNew = 0;
277         }
278         if (ifra->ifra_mask.sin_len) {
279             in_ifscrub(ifp, ia);
280             ia->ia_sockmask = ifra->ifra_mask;
281             ia->ia_subnetmask =
282                 ntohl(ia->ia_sockmask.sin_addr.s_addr);
283             maskIsNew = 1;
284         }
285         if ((ifp->if_flags & IFF_POINTOPOINT) &&
286             (ifra->ifra_dstaddr.sin_family == AF_INET)) {
287             in_ifscrub(ifp, ia);
288             ia->ia_dstaddr = ifra->ifra_dstaddr;
289             maskIsNew = 1;      /* We lie; but the effect's the same */
290         }
291         if (ifra->ifra_addr.sin_family == AF_INET &&
292             (hostIsNew || maskIsNew))
293             error = in_ifinit(ifp, ia, &ifra->ifra_addr, 0);
294         if ((ifp->if_flags & IFF_BROADCAST) &&
295             (ifra->ifra_broadaddr.sin_family == AF_INET))
296             ia->ia_broadaddr = ifra->ifra_broadaddr;
297         return (error);
```

in.c

图6-26 函数in_control：SIOCAIFADDR处理

代码在默认方式下取一个新的IP地址指派给接口(hostIsNew以1开始)。如果新地址的长度为0，in_control将当前地址复制到请求中，并将hostIsNew修改为0。如果长度不是0，并且新地址与老地址匹配，则这个请求不包含一个新地址，并且hostIsNew被设置为0。

278-284 如果在这个请求中指定一个网络掩码，则任何使用此当前地址的路由被忽略，并且in_control安装此新掩码。

285-290 如果接口是一个点对点接口，并且此请求包括一个新目的地址，则in_scrub忽略任何使用此地址的路由，新目的地址被安装，并且maskIsNew被设置为1，以强制调用in_ifinit来重配置接口。

291-297 如果配置了一个新地址或指派了一个新掩码，则in_ifinit作适当的修改来支持新的配置(图6-17)。注意，in_ifinit的最后一个参数为0。这表示已注意到这一点，不必刷新所有路由。最后，如果接口支持广播，则从in_aliasreq结构复制广播地址。

6.6.11 删除IP地址：SIOCDIFADDR

命令SIOCDIFADDR从一个接口删除IP地址，如图6-27所示。记住，ia指向要被删除的in_ifaddr结构(即，与请求匹配的)。

in.c

```
298     case SIOCDIFADDR:
299         in_ifscrub(ifp, ia);
300         if ((ifa = ifp->if_addrlist) == (struct ifaddr *) ia)
301             /* ia is the first address in the list */
302             ifp->if_addrlist = ifa->ifa_next;
303         else {
304             /* ia is *not* the first address in the list */
305             while (ifa->ifa_next &&
306                    (ifa->ifa_next != (struct ifaddr *) ia))
307                 ifa = ifa->ifa_next;
308             if (ifa->ifa_next)
309                 ifa->ifa_next = ((struct ifaddr *) ia)->ifa_next;
310             else
311                 printf("Couldn't unlink inifaddr from ifp\n");
312         }
313         oia = ia;
314         if (oia == (ia = in_ifaddr))
315             in_ifaddr = ia->ia_next;
316         else {
317             while (ia->ia_next && (ia->ia_next != oia))
318                 ia = ia->ia_next;
319             if (ia->ia_next)
320                 ia->ia_next = oia->ia_next;
321             else
322                 printf("Didn't unlink inifadr from list\n");
323         }
324         IFAFREE((&oia->ia_ifa));
325         break;
```

in.c

图6-27 in_control 函数：删除地址

298-323 前提条件处理代码将ia指向要删除的地址。in_ifscrub删除任何与此地址关联的路由。第一个 if删除接口地址列表的结构。第二个 if删除来自 Internet地址列表(in_ifaddr)的结构。

324-325 IFAFREE只在引用计数降到0时才释放此结构。

其他引用可能来自路由表中的各项。

6.7 接口ioctl处理

我们现在查看当一个地址被分配给接口时的专用 ioctl处理，对于我们的每个例子接口，这个处理分别在函数leioctl、slioctl和loioctl中。

in_ifinit通过图6-16中的SIOCSIFADDR代码和图6-26中的SIOCAIFADDR代码调用。in_ifinit总是通过接口的if_ioctl函数发送SIOCSIFADDR命令(图6-17)。

6.7.1 leioctl函数

图4-31显示了 LANCE驱动程序的 SIOCSIFFLAGS命令的处理。图 6-28显示了SIOCSIFADDR命令的处理。

614-637 在处理命令前，data转换成一个ifaddr结构指针，并且ifp->if_unit为此请求选择相应的le_softc结构。

leinit将接口标志为启动并初始化硬件。对于 Internet地址，IP地址保存在arpcom结构

中，并且为此地址发送一个免费ARP。免费ARP在21.5节和卷1的4.7节中讨论。

未识别命令

627-677 对于未识别命令，返回EINVAL。

if_le.c

```
614 leioctl(ifp, cmd, data)
615 struct ifnet *ifp;
616 int     cmd;
617 caddr_t data;
618 {
619     struct ifaddr *ifa = (struct ifaddr *) data;
620     struct le_softc *le = &le_softc[ifp->if_unit];
621     struct lereg1 *ler1 = le->sc_r1;
622     int     s = splimp(), error = 0;

623     switch (cmd) {
624     case SIOCSIFADDR:
625         ifp->if_flags |= IFF_UP;
626         switch (ifa->ifa_addr->sa_family) {
627         case AF_INET:
628             leinit(ifp->if_unit);   /* before arpwhohas */
629             ((struct arpcom *) ifp)->ac_ipaddr =
630                 IA_SIN(ifa)->sin_addr;
631             arpwhohas((struct arpcom *) ifp, &IA_SIN(ifa)->sin_addr);
632             break;
633         default:
634             leinit(ifp->if_unit);
635             break;
636         }
637         break;

                /* SIOCSIFFLAGS command (Figure 4.31) */
        /* SIOCADDMULTI and SIOCDELMULTI commands (Figure 12.31) */

672     default:
673         error = EINVAL;
674     }
675     splx(s);
676     return (error);
677 }
```

if_le.c

图6-28 函数leioctl

6.7.2 slioctl函数

函数slioctl(图6-29)为SLIP设备驱动器处理命令SIOCSIFADDR和SIOCSIFDSTADDR。

if_sl.c

```
653 int
654 slioctl(ifp, cmd, data)
655 struct ifnet *ifp;
656 int     cmd;
657 caddr_t data;
658 {
659     struct ifaddr *ifa = (struct ifaddr *) data;
660     struct ifreq *ifr;
661     int     s = splimp(), error = 0;

662     switch (cmd) {
```

图6-29 函数slioctld ：命令SIOCSIFADDR 和SIOCSIFDSTADDR

```
663     case SIOCSIFADDR:
664         if (ifa->ifa_addr->sa_family == AF_INET)
665             ifp->if_flags |= IFF_UP;
666         else
667             error = EAFNOSUPPORT;
668         break;

669     case SIOCSIFDSTADDR:
670         if (ifa->ifa_addr->sa_family != AF_INET)
671             error = EAFNOSUPPORT;
672         break;

            /* SIOCADDMULTI and SIOCDELMULTI commands (Figure 12.29)*/

688     default:
689         error = EINVAL;
690     }
691     splx(s);
692     return (error);
693 }
```

if_sl.c

图6-29 (续)

663-672 对于这两个命令，如果地址不是一个 IP地址，则返回 EAFNOSUPPORT。SIOCSIFADDR命令设置IFF_UP。

未识别命令

688-693 对于未识别命令，返回EINVAL。

6.7.3 loioctl函数

函数loioctl和它的SIOCSIFADDR命令的实现显示在图6-30中。

if_loop.c

```
135 int
136 loioctl(ifp, cmd, data)
137 struct ifnet *ifp;
138 int     cmd;
139 caddr_t data;
140 {
141     struct ifaddr *ifa;
142     struct ifreq *ifr;
143     int     error = 0;

144     switch (cmd) {
145     case SIOCSIFADDR:
146         ifp->if_flags |= IFF_UP;
147         ifa = (struct ifaddr *) data;
148         /*
149          * Everything else is done at a higher level.
150          */
151         break;

          /* SIOCADDMULTI and SIOCDELMULTI commands (Figure 12.30) */
```

图6-30 函数loioctl ：命令SIOCSIFADDR

```
167     default:
168         error = EINVAL;
169     }
170     return (error);
171 }
```

if_loop.c

图6-30 (续)

135-151 对于Internet地址，loioctl设置IFF_UP，并立即返回。

未识别命令

167-171 对于未识别命令，返回EINVAL。

注意，对于所有这三个例子驱动程序，指派一个地址会导致接口被标记为启用 (IFF_UP)。

6.8 Internet实用函数

图6-31列出了几个操作Internet地址或依赖于图6-5中ifnet结构的函数，它们通常用于发现不能单从32 bit IP地址中获得的子网信息。这些函数的实现主要包括数据结构的转换和操作位掩码。读者在netinet/in.c中可以找到这些函数。

函 数	说 明
in_netof	返回*in*中的网络和子网部分。主机比特被设置为0。对于D类地址，返回D类首标比特和用于多播组的0比特 `u_long in_netof(struct in_addr in);`
in_canforward	如果地址为*in*的IP分组有资格转发，则返回真。D类和E类地址、环回网络地址和有一个为0网络号的地址不能转发 `int in_canforward(struct in_addr in);`
in_localaddr	如果主机*in*被定位在一个直接连接的网络，则返回真。如果全局变量subnetsarelocal非0，则所有直接连接的网络的子网也被认为是本地的 `int in_localaddr(struct in_addr in);`
in_broadcast	如果*in*是一个由*ifp*指向的接口所关联的广播地址，则返回真 `int in_broadcast(struct in_addr in, struct ifnet *ifp);`

图6-31 Internet地址函数

Net/2在in_canforward中有一个错误：它允许转发环回地址。因为大多数Net/2系统被配置为只承认一个环回地址，如 127.0.0.1，Net/2系统常沿着默认路由在环回网络中转发其他地址(例如127.0.0.2)。

一个到127.0.0.2的telnet可能不是你所希望的！(习题6.6)

6.9 ifnet实用函数

几个查找数据结构的函数显示在图 6-5中。列于图6-32的函数接受任何协议族类的地址，因为它们的参数是指向一个 sockaddr结构的指针，这个结构中包含有地址族类。与图 6-31中的函数比较，在那里的每个函数将 32 bit的IP地址作为一个参数。这些函数定义在文件net/if.c中。

函 数	说 明
ifa_ifwithaddr	在ifnet列表中查找有一个单播或广播地址*addr*的接口。返回一个指向这个匹配的ifaddr结构的指针；或者若没有找到，则返回一个空指针 `struct ifaddr *ifa_ifwithaddr(struct sockaddr *addr);`
ifa_ifwithdstaddr	在ifnet列表中查找目的地址为*addr*的接口。返回一个指向这个匹配的ifaddr结构的指针；或者若没有找到，则返回一个空指针 `struct ifaddr *ifa_ifwithdstaddr(struct sockaddr *addr);`
ifa_ifwithnet	在ifnet列表中查找与*addr*同一网络的地址。返回匹配的ifaddr结构的指针；或者若没有找到，则返回一个空指针 `struct ifaddr *ifa_ifwithnet(struct sockaddr *addr);`
ifa_ifwithaf	在ifnet列表中查找与*addr*具有相同地址族类的第一个地址。返回匹配的ifaddr结构的指针；或者若没有找到，则返回一个空指针 `struct ifaddr *ifa_ifwithaf(struct sockaddr *addr);`
ifaof_ifpforaddr	在*ifp*列表中查找与*addr*匹配的地址。用于精确匹配的引用次序为：一个点对点链路上的目的地址、一个同一网络上的地址和一个在同一地址族类的地址，则返回匹配的ifaddr结构的指针；或者若没有找到，则返回一个空指针 `struct ifaddr * ifaof_ifpforaddr(struct sockaddr *addr, struct ifnet *ifp);`
ifa_ifwithroute	返回目的地址(dst)和网关地址(gateway)指定的相应的本地接口的ifaddr结构的指针 `struct ifaddr * ifa_ifwithroute(int flags, struct sockaddr *dst, struct sockaddr *gateway);`
ifunit	返回与*name*关联的ifnet结构的指针 `struct ifnet *ifunit(char *name);`

图6-32 ifnet 实用函数

6.10 小结

在本章中，我们概述了IP编址机制，并且说明了IP专用的接口地址结构和协议地址结构：结构in_ifaddr和sockaddr_in。

我们讨论了如何通过程序ifconfig和ioctl接口命令来配置接口的IP专用信息。

最后，我们总结了几个操作IP地址和查找接口数据结构的实用函数。

习题

6.1 你认为为什么在结构sockaddr_in中的sin_addr最初定义为一个结构？

6.2 ifunit("sl0")返回的指针指向图6-5中的哪个结构？

6.3 当IP地址已经包含在接口的地址列表中的一个 ifaddr结构中时，为什么还要在ac_ipaddr中备份？

6.4 你认为为什么IP接口地址要通过一个UDP插口而不是一个原始的IP插口来访问？

6.5 为什么 in_socktrim要修改 sin_len来匹配掩码的长度，而不使用一个sockaddr_in结构的标准长度？

6.6 当一个telnet 127.0.0.2命令中的连接请求部分被一个 Net/2系统错误地转发，并且最后被认可，同时还被默认路由上的一个系统所接收时，会发生什么情况？

第7章 域和协议

7.1 引言

在本章中，我们讨论支持同时操作多个网络协议的 Net/3数据结构。用Internet协议来说明在系统初始化时这些数据结构的构造和初始化。本章为我们讨论 IP协议处理层提供必要的背景资料，IP协议处理层在第8章讨论。

Net/3组把协议关联到一个域中，并且用一个协议族常量来标识每个域。 Net/3还通过所使用的编址方法将协议分组。回忆图 3-19，地址族也有标识常量。当前，在一个域中的每个协议使用同类地址，并且每种地址只被一个域使用。作为结果，一个域能通过它的协议族或地址族常量唯一标识。图7-1列出了我们讨论的协议和常量。

协 议 族	地 址 族	协 议
PF_INET	*AF_INET*	Internet
PF_OSI, PF_ISO	*AF_OSI, AF_ISO*	OSI
PF_LOCAL, PF_UNIX	*AF_LOCAL, AF_UNIX*	本地IPC(Unix)
PF_ROUTE	*AF_ROUTE*	路由表
n/a	*AF_LINK*	链路层 (例如以太网)

图7-1 公共的协议和地址族常量

PF_LOCAL和AF_LOCAL是支持同一主机上进程间通信的协议的主要标识，它们是POSIX.12标准的一部分。在 Net/3以前，用PF_UNIX和AF_UNIX标识这些协议。在Net/3中保留UNIX常量，用于向后兼容，并且要在本书中讨论。

PF_UNIX域支持在一个单独的 Unix主机上的进程间通信。细节见 [Stevens 1990]。PF_ROUTE域支持在一个进程和内核中路由软件间的通信 (第18章)。我们偶尔引用PF_OSI协议，它作为 Net/3特性仅支持 OSI协议，但我们不讨论它们的细节。大多数讨论是关于PF_INET协议的。

7.2 代码介绍

本章涉及两个头文件和两个C文件。图7-2描述了这4个文件。

文 件	说 明
netinet/domain.h	domain结构定义
netinet/protosw.h	protosw结构定义
netinet/in_proto.c	IP domain和protosw结构
kern/uipc_domain.c	初始化和查找函数

图7-2 在本章中讨论的文件

7.2.1 全局变量

图7-3描述了几个重要的全局数据结构和系统参数，它们在本章中讨论，并经常在 Net/3中引用。

变 量	数据类型	说 明
domain	struct domain *	链接的域列表
inetdomain	struct domain	Internet协议的domain结构
inetsw	struct protosw[]	Internet协议的protosw结构数组
max_linkhdr	int	见图7-17
max_protohdr	int	见图7-17
max_hdr	int	见图7-17
max_datalen	int	见图7-17

图7-3 在本章中介绍的全局变量

7.2.2 统计量

除了图7-4显示的由函数ip_init分配和初始化的统计量表，本章讨论的代码没有收集其他统计量。通过一个内核调试工具是查看这个表的唯一方法。

变 量	数据类型	说 明
ip_ifmatrix	int[][]	二维数组，用来统计在任意两个接口间传送的分组数

图7-4 在本章中收集的统计量

7.3 domain结构

一个协议域由一个图7-5所示的domain结构来表示。

domain.h

```
42 struct domain {
43     int     dom_family;          /* AF_xxx */
44     char   *dom_name;
45     void    (*dom_init)          /* initialize domain data structures */
46             (void);
47     int     (*dom_externalize)   /* externalize access rights */
48             (struct mbuf *);
49     int     (*dom_dispose)       /* dispose of internalized rights */
50             (struct mbuf *);
51     struct protosw *dom_protosw, *dom_protoswNPROTOSW;
52     struct domain *dom_next;
53     int     (*dom_rtattach)      /* initialize routing table */
54             (void **, int);
55     int     dom_rtoffset;        /* an arg to rtattach, in bits */
56     int     dom_maxrtkey;        /* for routing layer */
57 };
```

domain.h

图7-5 结构domain 的定义

42-57 dom_family是一个地址族常量(例如AF_INET)，它指示在此域中协议使用的编址方式。dom_name是此域的一个文本名称(例如“internet”)。

除了程序fstat (1)在它格式化插口信息时使用dom_name外，成员dom_name不被Net/3内核的任何部分访问。

dom_init指向初始化此域的函数。dom_externalize和dom_dispose指向那些管理通过此域内通信路径发送的访问权限的函数。Unix域实现这个特性在进程间传递文件描述符。Internet域不实现访问权限。

dom_protosw和dom_protoswNPROTOSW指向一个protosw结构的数组的起始和结束。dom_next指向在一个内核支持的域链表中的下一个域。包含所有域的链表通过全局指针domains来访问。

接下来的三个成员，dom_rtattach、dom_rtoffset和dom_maxrtkey保存此域的路由信息。它们在第18章讨论。

图7-6显示了一个domains列表的例子。

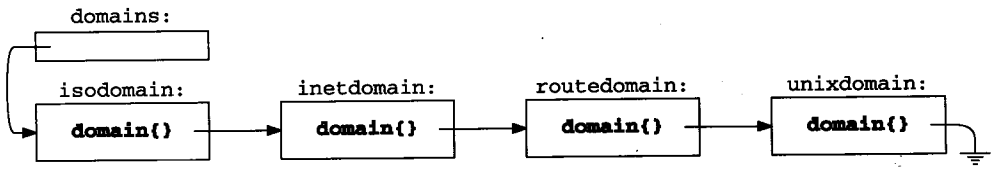


图7-6 domains 列表

7.4 protosw结构

在编译期间，Net/3为内核中的每个协议分配一个protosw结构并初始化，同时将在一个域中的所有协议的这个结构组织到一个数组中。每个domain结构引用相应的protosw结构数组。一个内核可以通过提供多个protosw项为同一协议提供多个接口。Protosw结构的定义见图7-7。

protosw.h

```
57 struct protosw {
58     short   pr_type;            /* see (Figure 7.8) */
59     struct domain *pr_domain;   /* domain protocol a member of */
60     short   pr_protocol;        /* protocol number */
61     short   pr_flags;           /* see Figure 7.9 */
62 /* protocol-protocol hooks */
63     void    (*pr_input) ();     /* input to protocol (from below) */
64     int     (*pr_output) ();    /* output to protocol (from above) */
65     void    (*pr_ctlinput) ();  /* control input (from below) */
66     int     (*pr_ctloutput) (); /* control output (from above) */
67 /* user-protocol hook */
68     int     (*pr_usrreq) ();    /* user request from process */
69 /* utility hooks */
70     void    (*pr_init) ();      /* initialization hook */
71     void    (*pr_fasttimo) ();  /* fast timeout (200ms) */
72     void    (*pr_slowtimo) ();  /* slow timeout (500ms) */
73     void    (*pr_drain) ();     /* flush any excess space possible */
74     int     (*pr_sysctl) ();    /* sysctl for protocol */
75 };
```

protosw.h

图7-7 protosw 结构的定义

57-61 此结构中的前4个成员用来标识和表征协议。pr_type指示协议的通信语义。图 7-8

列出了pr_type可能的值和对应的Internet协议。

pr_type	协 议 语 义	Internet协议
SOCK_STREAM	可靠的双向字节流服务	TCP
SOCK_DGRAM	最好的运输层数据报服务	UDP
SOCK_RAW	最好的网络层数据报服务	ICMP，IGMP，原始IP
SOCK_RDM	可靠的数据报服务(未实现)	n/a
SOCK_SEQPACKET	可靠的双向记录流服务	n/a

图7-8 pr_type 指明协议的语义

pr_domain指向相关的domain结构，pr_protocol为域中协议的编号，pr_flags标识协议的附加特征。图7-9列出了pr_flags的可能值。

pr_flags	说 明
PR_ATOMIC	每个进程请求映射为一个单个的协议请求
PR_ADDR	协议在每个数据报中都传递地址
PR_CONNREQUIRED	协议是面向连接的
PR_WANTRCVD	当一个进程接收到数据时通知协议
PR_RIGHTS	协议支持访问权限

图7-9 pr_flags 的值

如果一个协议支持 PR_ADDR，必须也支持 PR_ATOMIC。PR_ADDR和PR_CONNREQUIRED是互斥的。

当设置了PR_WANTRCVD，并当插口层将插口接收缓存中的数据传递给一个进程时(即当在接收缓存中有更多空间可用时)，它通知协议层。

PR_RIGHTS指示访问权限控制报文能通过连接来传递。访问权限要求内核中的其他支持来确保在接收进程没有销毁报文时能完成正确的清除工作。仅 Unix域支持访问权限，在那里它们用来在进程间传递描述符。

图7-10所示的是协议类型、协议标志和协议语义间的关系。

pr_type	PR_			是否有记录边界	可靠否	举 例	
	ADDR	ATOMIC	CONNREQUIRED			Internet	其他
SOCK_STREAM			•	否	•	TCP	SPP
SOCK_SEQPACKET		•	• •	显式 隐式	• •		TP4 SPP
SOCK_RDM		•	•	隐式	见正文		RDP
SOCK_DGRAM *SOCK_RAW*	• •	• •		隐式 隐式		UDP ICMP	

图7-10 协议特征和举例

图7-10不包括标志PR_WANTRCVD和PR_RIGHTS。对于可靠的面向连接的协议，PR_WANTRCVD总是被设置。

为了理解 Net/3中一个protosw项的通信语义，我们必须要一起考虑 PR*xxx*标志和pr_type。在图7-10中，我们用两列(“是否有记录边界”和“可靠否”)来描述由pr_type

隐式指示的语义。图7-10显示了可靠协议的三种类型：

- 面向连接的字节流协议，如TCP和SPP(源于XNS协议族)。这些协议用SOCK_STREAM标识。
- 有记录边界的面向连接的流协议用 SOCK_SEQPACKET标识。在这种协议类型中，PR_ATOMIC指示记录是否由每个输出请求隐式地指定，或者显式地通过在输出中设置标志MSG_EOR来指定。

  SPP同时支持语义SOCK_STREAM和SOCK_SEQPACKET。

- 第三种可靠协议提供一个有隐式记录边界的面向连接服务，它由 SOCK_RDM标识。RDP不保证按照记录发送的顺序接收记录。 RDP在[Partridge 1987]中讨论并在RFC 115 [Partridge and Hinden 1990]中被描述。

两种不可靠协议显示在图7-10中：

- 一个运输层数据报协议，如UDP，它包括复用和检验和，由SOCK_DGRAM指定。
- 一个网络层数据报协议，如ICMP，它由SOCK_RAW指定。在Net/3中，只有超级用户进程才能创建一个SOCK_RAW插口(图15-8)。

62-68 接着的5个成员是函数指针，用来提供从其他协议对此协议的访问。 pr_input处理从一个低层协议输入的数据， pr_output处理从一个高层协议输出的数据， pr_ctlinput处理来自下层的控制信息，而 pr_ctloutput处理来自上层的控制信息。 pr_usrreq处理来自进程的所有通信请求。如图7-11所示。

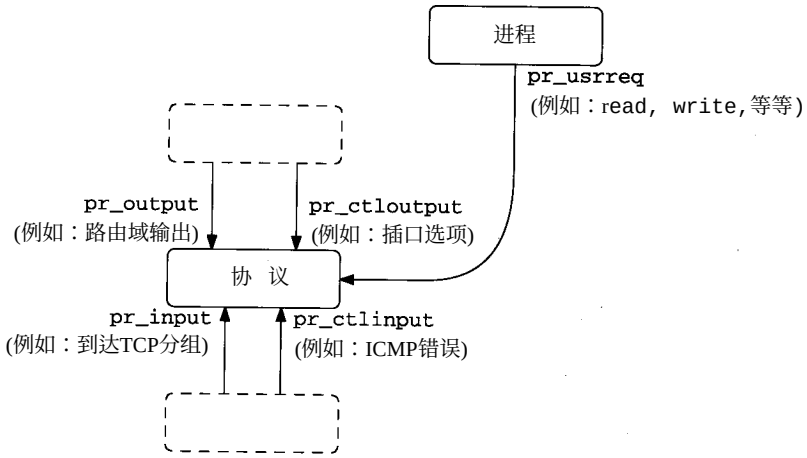


图7-11 一个协议的5个主要入口点

69-75 剩下的5个成员是协议的实用函数。 pr_init处理初始化。 pr_fasttimo和pr_slowtimo分别每200 ms和500 ms被调用来执行周期性的协议函数，如更新重传定时器。pr_drain在内存缺乏时被m_reclaim调用(图2-13)。它请求协议释放尽可能多的内存。pr_sysctl为sysctl(8)命令提供一个接口，一种修改系统范围的参数的方式，如允许转发分组或UDP检验和计算。

7.5 IP的domain和protosw结构

声明所有协议的结构 domain和protosw，并进行静态初始化。对于 Internet协议，inetsw数组包含protosw结构。图7-12总结了在数组inetsw中的协议信息。图7-13显示了

Internet协议的数组定义和domain结构的定义。

inetsw[]	pr_protocol	pr_type	说 明	缩 写
0	*0*	*0*	Internet协议	IP
1	*IPPROTO_UDP*	*SOCK_DGRAM*	用户数据报协议	UDP
2	*IPPROTO_TCP*	*SOCK_STREAM*	传输控制协议	TCP
3	*IPPROTO_RAW*	*SOCK_RAW*	Internet协议(原始)	IP(原始)
4	*IPPROTO_ICMP*	*SOCK_RAW*	Internet控制报文协议	ICMP
5	*IPPROTO_IGMP*	*SOCK_RAW*	Internet组管理协议	IGMP
6	*0*	*SOCK_RAW*	Internet协议(原始、默认)	IP(原始)

图7-12 Internet域协议

in_proto.c

```
39 struct protosw inetsw[] =
40 {
41     {0, &inetdomain, 0, 0,
42      0, ip_output, 0, 0,
43      0,
44      ip_init, 0, ip_slowtimo, ip_drain, ip_sysctl
45     },
46     {SOCK_DGRAM, &inetdomain, IPPROTO_UDP, PR_ATOMIC | PR_ADDR,
47      udp_input, 0, udp_ctlinput, ip_ctloutput,
48      udp_usrreq,
49      udp_init, 0, 0, 0, udp_sysctl
50     },
51     {SOCK_STREAM, &inetdomain, IPPROTO_TCP, PR_CONNREQUIRED | PR_WANTRCVD,
52      tcp_input, 0, tcp_ctlinput, tcp_ctloutput,
53      tcp_usrreq,
54      tcp_init, tcp_fasttimo, tcp_slowtimo, tcp_drain,
55     },
56     {SOCK_RAW, &inetdomain, IPPROTO_RAW, PR_ATOMIC | PR_ADDR,
57      rip_input, rip_output, 0, rip_ctloutput,
58      rip_usrreq,
59      0, 0, 0, 0,
60     },
61     {SOCK_RAW, &inetdomain, IPPROTO_ICMP, PR_ATOMIC | PR_ADDR,
62      icmp_input, rip_output, 0, rip_ctloutput,
63      rip_usrreq,
64      0, 0, 0, 0, icmp_sysctl
65     },
66     {SOCK_RAW, &inetdomain, IPPROTO_IGMP, PR_ATOMIC | PR_ADDR,
67      igmp_input, rip_output, 0, rip_ctloutput,
68      rip_usrreq,
69      igmp_init, igmp_fasttimo, 0, 0,
70     },
71     /* raw wildcard */
72     {SOCK_RAW, &inetdomain, 0, PR_ATOMIC | PR_ADDR,
73      rip_input, rip_output, 0, rip_ctloutput,
74      rip_usrreq,
75      rip_init, 0, 0, 0,
76     },
77 };

78 struct domain inetdomain =
79 {AF_INET, "internet", 0, 0, 0,
80  inetsw, &inetsw[sizeof(inetsw) / sizeof(inetsw[0])], 0,
81  rn_inithead, 32, sizeof(struct sockaddr_in)};
```

in_proto.c

图7-13 Internet 的domain 和protosw 结构

39-77 在inetsw数组中的3个protosw结构提供对IP的访问。第一个：inetsw[0]，标识IP的管理函数并且只能由内核访问。其他两项： inetsw[3]和inetsw[6]，除了pr_protocol值以外它们是一样的，都提供到IP的一个原始接口。inetsw[3]处理接收到的任何未识别协议的分组。inetsw[6]是默认的原始协议，当没有找到其他可匹配的项时，这个结构由函数pffindproto返回(7.6节)。

在Net/3以前的版本中，通过inetsw[3]传输不带IP首部的分组，由进程负责构造正确的首部。由内核通过inetsw[6]传输带IP首部的分组。4.3BSD Reno引入了IP_HDRINCL插口选项(32.8节)，这样在inetsw[3]和inetsw[6]之间的区别就不再重要了。

原始接口允许一个进程发送和接收不涉及运输层的IP分组。原始接口的一个用途是实现内核外的传输协议。一旦这个协议稳定下来，就能移植到内核中改进它的性能和对其他进程的可用性。另一个用途就是作为诊断工具，如traceroute，它使用原始IP接口来直接访问IP。第32章讨论原始IP接口。图7-14总结了IP protosw结构。

protosw	inetsw[0]	inetsw[3和6]	说 明
pr_type	*0*	*SOCK_RAW*	IP提供原始分组服务
pr_domain	*&inetdomain*	*&inetdomain*	两协议都是Internet域的一部分
pr_protocol	*0*	*IPPROTO_RAW或0*	IPPROTO_RAW(255)和0都是预留的(RFC 1700)，并且不应在一个IP数据报中出现
pr_flags	*0*	*PR_ATOMIC/PR_ADDR*	插口层标志，IP不使用
pr_input	*null*	*rip_input*	从IP、ICMP或IGMP接收未识别的数据报
pr_output	*ip_output*	*rip_output*	分别准备并发送数据报到IP和硬件层
pr_ctlinput	*null*	*null*	IP不使用
pr_ctloutput	*null*	*rip_ctloutput*	响应来自进程的配置请求
pr_usrreq	*null*	*rip_usrreq*	响应来自进程的协议请求
pr_init	*ip_init*	*null或rip_init*	ip_init完成所有初始化
pr_fasttimo	*null*	*null*	IP不使用
pr_slowtimo	*ip_slowtimo*	*null*	用于IP重装算法的慢超时
pr_drain	*ip_drain*	*null*	如果可能，释放内存
pr_sysctl	*ip_sysctl*	*null*	修改系统范围参数

图7-14 IP inetsw 的条目

78-81 Internet协议的domain结构显示在图7-13的下部。Internet域使用AF_INET风格编址，文本名称为“internet”，没有初始化和控制报文函数，它的protosw结构在inetsw数组中。

Internet协议的路由初始化函数是rn_inithead。一个IP地址的最大有效位数为32，并且一个Internet选路键的大小为一个sockaddr_in结构的大小(16字节)。

inetsw[3]和inetsw[6]的唯一区别是它们的pr_protocol号和初始化函数rip_init，它仅在inetsw[6]中定义，因此只在初始化期间被调用一次。

domaininit函数

在系统初始化期间(图3-23)，内核调用domaininit来链接结构domain和protosw。domaininit显示在图7-15中。

uipc_domain.c

```
37 /* simplifies code in domaininit */
38 #define ADDDOMAIN(x)    { \
39     extern struct domain __CONCAT(x,domain); \
40     __CONCAT(x,domain.dom_next) = domains; \
41     domains = &__CONCAT(x,domain); \
42 }
43 domaininit()
44 {
45     struct domain *dp;
46     struct protosw *pr;
47     /* The C compiler usually defines unix. We don't want to get
48      * confused with the unix argument to ADDDOMAIN
49      */
50 #undef unix
51     ADDDOMAIN(unix);
52     ADDDOMAIN(route);
53     ADDDOMAIN(inet);
54     ADDDOMAIN(iso);
55     for (dp = domains; dp; dp = dp->dom_next) {
56         if (dp->dom_init)
57             (*dp->dom_init) ();
58         for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
59             if (pr->pr_init)
60                 (*pr->pr_init) ();
61     }
62     if (max_linkhdr < 16)       /* XXX */
63         max_linkhdr = 16;
64     max_hdr = max_linkhdr + max_protohdr;
65     max_datalen = MHLEN - max_hdr;
66     timeout(pffasttimo, (void *) 0, 1);
67     timeout(pfslowtimo, (void *) 0, 1);
68 }
```

uipc_domain.c

图7-15 函数domaininit

37-42 ADDDOMAIN宏声明并链接一个domain结构。例如，ADDDOMAIN(unix)展开为：

```
extern struct domain unixdomain;
unixdomain.dom_next = domains;
domains = &unixdomain;
```

宏_CONCAT定义在sys/defs.h中，并且连接两个符号名。例如 _CONCAT(unix, domain)产生unixdomain。

43-54 domaininit通过调用ADDDOMAIN为每个支持的域构造域列表。

因为符号unix常常被C预处理器预定义，因此，Net/3在这里显式地取消它的定义，使ADDDOMAIN能正确工作。

图7-16显示了链接的结构domain和protosw，它们配置在内核中来支持Internet、Unix和OSI协议族。

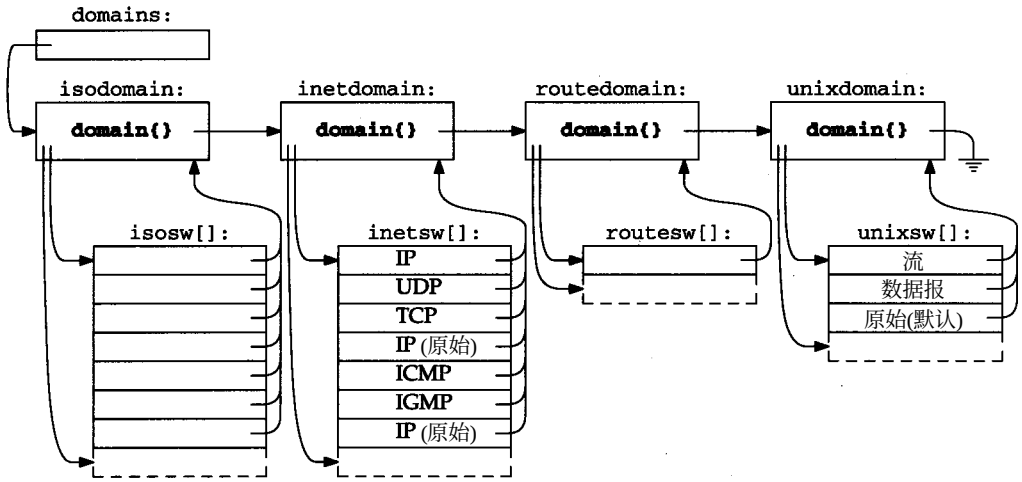


图7-16 初始化后的domain 链表和protosw 数组

55-61 两个嵌套的for循环查找内核中的每个域和协议，并且若定义了初始化函数dom_init和pr_init，则调用它们。对于Internet协议，调用下面的函数 (图7-13)：ip_init、udp_init、tcp_init、igmp_init和rip_init。

62-65 在domaininit中计算这些参数，用来控制mbuf中分组的格式，以避免对数据的额外复制。在协议初始化期间设置了max_linkhdr和max_protohdr。这里，domaininit将max_linkhdr强制设置为一个下限16。16字节用于给带有4字节边界的14字节以太网首部留出空间。图7-17和图7-18列出了这些参数和典型的取值。

变　量	值	说　明
max_linkhdr	16	由链路层添加的最大字节数
max_protohdr	40	由网络和运输层添加的最大字节数
max_hdr	56	max_linkhdr + max_protohdr
max_datalen	44	在计算了链路和协议首部后的分组首部mubf中的可用数据字节数

图7-17 用来减少协议数据复制的参数

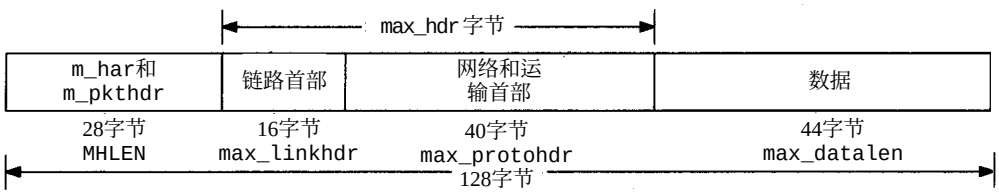


图7-18 mbuf和相关的最大首部长度

max_protohdr是一个软限制，估算预期的协议首部大小。在Internet域中，IP和TCP首部长度通常为20字节，但都可达到60字节。长度超过max_protohdr的代价是花时间将数据向后移动，以留出比预期的协议首部更大的空间。

66-68 domaininit通过调用timeout启动pfslowtimo和pffasttimo。第3个参数指明何时内核应该调用这个函数，在这里是在1个时钟滴答内。两个函数都显示在图7-19中。

uipc_domain.c

```
153 void
154 pfslowtimo(arg)
155 void   *arg;
156 {
157     struct domain *dp;
158     struct protosw *pr;

159     for (dp = domains; dp; dp = dp->dom_next)
160         for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
161             if (pr->pr_slowtimo)
162                 (*pr->pr_slowtimo) ();
163     timeout(pfslowtimo, (void *) 0, hz / 2);
164 }

165 void
166 pffasttimo(arg)
167 void   *arg;
168 {
169     struct domain *dp;
170     struct protosw *pr;

171     for (dp = domains; dp; dp = dp->dom_next)
172         for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
173             if (pr->pr_fasttimo)
174                 (*pr->pr_fasttimo) ();
175     timeout(pffasttimo, (void *) 0, hz / 5);
176 }
```

uipc_domain.c

图7-19 函数pfslowtimo 和pffasttimo

153-176 这两个相近的函数用两个 for循环分别为每个协议调用函数 pr_slowtimo和pr_fasttimo，前提是如果定义了这两个函数。这两个函数每 500 ms和200 ms通过调用timeout调度自己一次，timeout在图3-43中讨论过。

7.6 pffindproto和pffindtype函数

如图7-20所示，函数pffindproto和pffindtype通过编号(例如IPPROTO_TCP)或类型(例如SOCK_STREAM)来查找一个协议。如我们在第15章要看到的，当进程创建一个插口时，这两个函数被调用来查找相应的protosw项。

69-84 pffindtype线性搜索domains，查找指定的族，然后在域内搜索第一个为此指定类型的协议。

85-107 pffindproto像pffindtype一样搜索domains，查找由调用者指定的族、类型和协议。如果pffindproto在指定的协议族中没有发现一个 (protocol，type)匹配项，并且type为SOCK_RAW，而此域有一个默认的原始协议 (pr_protocol等于0)，则pffindproto选择默认的原始协议而不是完全失败。例如，一个调用如下：

```
pffindproto(PF_INET, 27, SOCK_RAW);
```

它返回一个指向inetsw[6]的指针，默认的原始IP协议，因为Net/3不包括对协议27的支持。通过访问原始IP，一个进程可以使用内核来管理 IP分组的发送和接收，从而自己实现协议 27

服务。

协议27预留给可靠的数据报协议(RFC 1151)。

两个函数都返回一个所选协议的protosw结构的指针；或者，如果没有找到匹配项，则返回一个空指针。

uipc_domain.c

```
 69 struct protosw *
 70 pffindtype(family, type)
 71 int     family, type;
 72 {
 73     struct domain *dp;
 74     struct protosw *pr;

 75     for (dp = domains; dp; dp = dp->dom_next)
 76         if (dp->dom_family == family)
 77             goto found;
 78     return (0);
 79   found:
 80     for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
 81         if (pr->pr_type && pr->pr_type == type)
 82             return (pr);
 83     return (0);
 84 }

 85 struct protosw *
 86 pffindproto(family, protocol, type)
 87 int     family, protocol, type;
 88 {
 89     struct domain *dp;
 90     struct protosw *pr;
 91     struct protosw *maybe = 0;

 92     if (family == 0)
 93         return (0);
 94     for (dp = domains; dp; dp = dp->dom_next)
 95         if (dp->dom_family == family)
 96             goto found;
 97     return (0);
 98   found:
 99     for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++) {
100         if ((pr->pr_protocol == protocol) && (pr->pr_type == type))
101             return (pr);

102         if (type == SOCK_RAW && pr->pr_type == SOCK_RAW &&
103             pr->pr_protocol == 0 && maybe == (struct protosw *) 0)
104             maybe = pr;
105     }
106     return (maybe);
107 }
```

uipc_domain.c

图7-20 函数pffindproto 和pffindtype

举例

我们在15.6节中会看到，当一个应用程序进行下面的调用时：

```
socket(PF_INET, SOCK_STREAM, 0);    /* TCP 插口 */
```

pffindtype被调用如下：

```
pffindtype(PF_INET, SOCK_STREAM);
```

图7-12显示pffindtype会返回一个指向inetsw[2]的指针，因为TCP是此数组中第一个SOCK_STREAM协议。同样，

```
socket(PF_INET, SOCK_DGRAM, 0);    /* UCP 插口 */
```

会导致

```
pffindtype(PF_INET, SOCK_DGRAM);
```

它返回一个指向inetsw[1]中UDP的指针。

7.7 pfctlinput函数

函数pfctlinput给每个域中的每个协议发送一个控制请求 (图7-21)。当可能影响每个协议的事件发生时，使用这个函数，例如一个接口被关闭，或路由表发生改变。当一个 ICMP重定向报文到达时，ICMP调用pfctlinput (图11-14)，因为重定向会影响所有Internet协议(例如UDP和TCP)。

uipc_domain.c

```
142 pfctlinput(cmd, sa)
143 int     cmd;
144 struct sockaddr *sa;
145 {
146     struct domain *dp;
147     struct protosw *pr;

148     for (dp = domains; dp; dp = dp->dom_next)
149         for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
150             if (pr->pr_ctlinput)
151                 (*pr->pr_ctlinput) (cmd, sa, (caddr_t) 0);
152 }
```

uipc_domain.c

图7-21 函数pfctlinput

142-152 两个嵌套的for循环查找每个域中的每个协议。pfctlinput通过调用每个协议的pr_ctlinput函数来发送由cmd指定的协议控制命令。对于UDP，调用udp_ctlinput；而对于TCP，调用tcp_ctlinput。

7.8 IP初始化

如图7-13所示，Internet域没有一个初始化函数，但单个Internet协议有。现在，我们仅查看IP初始化函数ip_init。在第23章和第24章中，我们讨论UDP和TCP初始化函数。在讨论这些代码前，需要说明一下数组ip_protox。

7.8.1 Internet传输分用

一个网络层协议像IP必须分用输入数据报，并将它们传递到相应的运输层协议。为了完成这些，相应的protosw结构必须通过一个在数据报中出现的协议编号得到。对于 Internet协议，这由数组ip_protox来完成，如图7-22所示。

数组ip_protox的下标是来自IP首部的协议值(ip_p，图8-8)。被选项是inetsw数组中处理此数据报的协议的下标。例如，一个协议编号为 6的数据报由inetsw[2]处理，协议为TCP。内核在协议初始化时构造ip_protox，如图7-23所示。

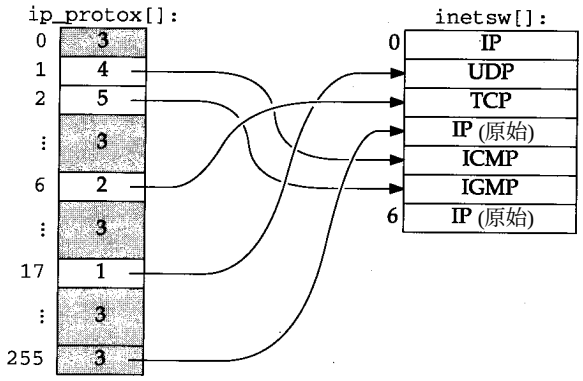


图7-22 数组ip_protox 将协议编号映射到数组inetsw 中的一项

ip_input.c

```
71 void
72 ip_init()
73 {
74     struct protosw *pr;
75     int     i;

76     pr = pffindproto(PF_INET, IPPROTO_RAW, SOCK_RAW);
77     if (pr == 0)
78         panic("ip_init");
79     for (i = 0; i < IPPROTO_MAX; i++)
80         ip_protox[i] = pr - inetsw;
81     for (pr = inetdomain.dom_protosw;
82          pr < inetdomain.dom_protoswNPROTOSW; pr++)
83         if (pr->pr_domain->dom_family == PF_INET &&
84             pr->pr_protocol && pr->pr_protocol != IPPROTO_RAW)
85             ip_protox[pr->pr_protocol] = pr - inetsw;
86     ipq.next = ipq.prev = &ipq;
87     ip_id = time.tv_sec & 0xffff;
88     ipintrq.ifq_maxlen = ipqmaxlen;
89     i = (if_index + 1) * (if_index + 1) * sizeof(u_long);
90     ip_ifmatrix = (u_long *) malloc(i, M_RTABLE, M_WAITOK);
91     bzero((char *) ip_ifmatrix, i);
92 }
```

ip_input.c

图7-23 函数ip_init

7.8.2 ip_init函数

domaininit (图7-15)在系统初始化期间调用函数ip_init。

71-78 pffindproto返回一个指向原始协议(inetsw[3]，图7-14)的指针。如果找不到原始协议，Net/3就调用panic，因为这是内核要求的部分。如果找不到原始协议，内核一定被错误配置了。IP将传输到一个未知传输协议的到达分组传递给此协议，在那里它们由内核外部的一个进程来处理。

79-85 接着的两个循环初始化数组ip_protox。第一个循环将数组中的每项设置为pr，即默认协议的下标(图7-22中为3)。第二个循环检查inetsw中的每个协议(而不是协议编号为0或IPPROTO_RAW的项)，并且将ip_protox中的匹配项设置为引用相应的inetsw项。为此，每个protosw结构中的pr_protocol必须是期望出现在输入数据报中的协议编号。

86-92 ip_init初始化IP重装队列ipq(10.6节)，用系统时钟植入ip_id，并将IP输入队列(ipintrq)的最大长度设置为50(ipqmaxlen)。ip_id用系统时钟设置，为数据报标识符提供一个随机起点(10.6节)。最后，ip_init分配一个两维数组ip_ifmatrix，统计在系统接口之间路由的分组数。

在Net/3中，有很多变量可以被一个系统管理员修改。为了允许在运行时改变这些变量而不需重新编译内核，一个常量(在此例中是IFQ_MAXLEN)表示的默认值在编译时指派给一个变量(ipqmaxlen)。一个系统管理员能使用一个内核调试器如adb，来修改ipqmaxlen，并用新值重启内核。如果图7-23直接使用IFQ_MAXLEN，它会要求内核重新编译来改变这个限制。

7.9 sysctl系统调用

系统调用sysctl访问并修改Net/3系统范围参数。系统管理员通过程序sysctl(8)修改这些参数。每个参数由一个分层的整数列表来标识，并有一个相应的类型。此系统调用的原型为：

```
int sysctl(int * name, u_int namelen, void *old, size_t * oldlenp, void *new, size_t newlen);
```

*name指向一个包含namelen个整数的数组。*old指向在此范围内返回的旧值，*new指向在此范围内传递的新值。

图7-24总结了关于联网名称的组织。

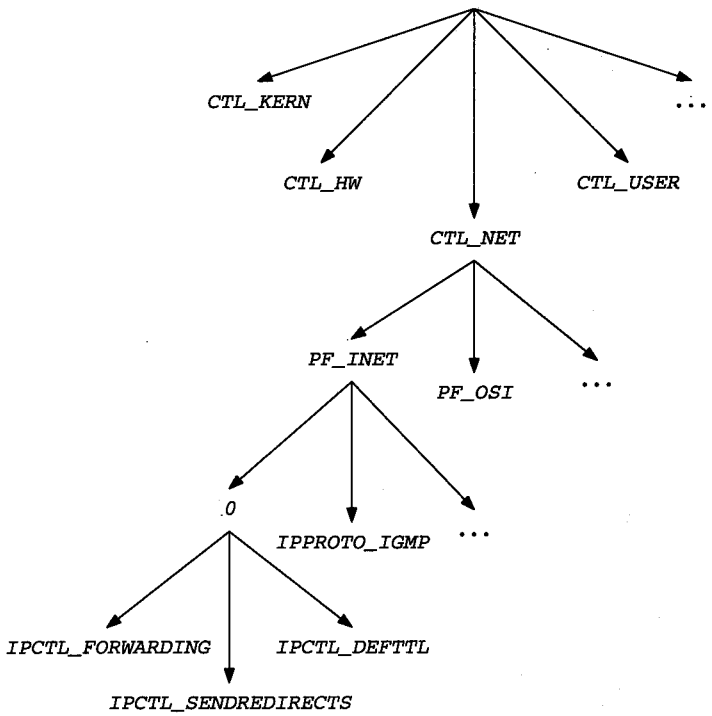


图7-24 sysctl 的名称组织

在图7-24中，IP转发标志的全名为

CTL_NET、PF_INET、0、IPCTL_FORWARDING

用4个整数存储在一个数组中。

net_sysctl函数

每层的sysctl命名方案通过不同函数处理。图7-25显示了处理这些Internet参数的函数。

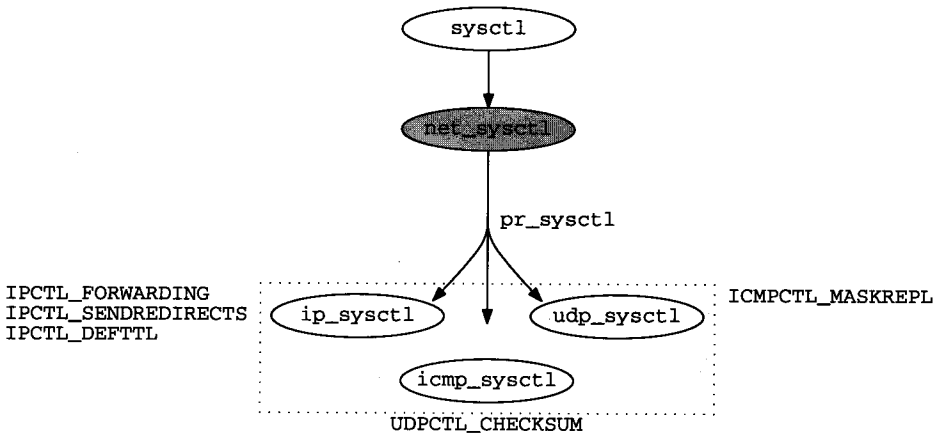


图7-25 处理Internet 参数的sysctl 函数

顶层名称由sysctl处理。网络层名称由net_sysctl处理，它根据族和协议将控制转给此协议的protosw项指定的pr_sysctl函数。

sysctl在内核中通过_sysctl函数实现，函数_sysctl不在本书中讨论。它包含将sysctl参数传给内核和从内核取出sysctl参数的代码及一个switch语句，这个switch语句选择相应的函数来处理这些参数，在这里是net_sysctl。

图7-26所示的是函数net_sysctl。

uipc_domain.c

```
108 net_sysctl(name, namelen, oldp, oldlenp, newp, newlen, p)
109 int    *name;
110 u_int   namelen;
111 void   *oldp;
112 size_t *oldlenp;
113 void   *newp;
114 size_t  newlen;
115 struct proc *p;
116 {
117     struct domain *dp;
118     struct protosw *pr;
119     int     family, protocol;

120     /*
121      * All sysctl names at this level are nonterminal;
122      * next two components are protocol family and protocol number,
123      * then at least one additional component.
124      */
125     if (namelen < 3)
126         return (EISDIR);        /* overloaded */
127     family = name[0];
128     protocol = name[1];

129     if (family == 0)
```

图7-26 函数net_sysctl

```
130         return (0);
131     for (dp = domains; dp; dp = dp->dom_next)
132         if (dp->dom_family == family)
133             goto found;
134     return (ENOPROTOOPT);
135   found:
136     for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
137         if (pr->pr_protocol == protocol && pr->pr_sysctl)
138             return ((*pr->pr_sysctl) (name + 2, namelen - 2,
139                                       oldp, oldlenp, newp, newlen));
140     return (ENOPROTOOPT);
141 }
```

uipc_domain.c

图7-26 (续)

108-119 net_sysctl的参数除了增加了p外，同系统调用sysctl一样，p指向当前进程结构。

120-134 在名称中接下来的两个整数被认为是在结构domain和protosw中指定的协议族和协议编号成员的值。如果没有指定族，则返回0。如果指定了族，for循环在域列表中查找一个匹配的族。如果没有找到，则返回ENOPROTOOPT。

135-141 如果找到匹配域，第二个for循环查找第一个定义了函数pr_sysctl的匹配协议。当找到匹配项，将请求传递给此协议的pr_sysctl函数。注意，把(name+2)指向的整数传递给下一级。如果没有找到匹配的协议，返回ENOPROTOOPT。

图7-27所示的是为Internet协议定义的pr_sysctl函数。

pr_protocol	inetsw[]	pr_sysctl	说 明	参 考
0	0	*ip_sysctl*	IP	8.9节
IPPROTO_UDP	1	*udp_sysctl*	UDP	23.11节
IPPROTO_ICMP	4	*icmp_sysctl*	ICMP	11.14节

图7-27 Internet协议族的pr_sysctl 函数

在路由选择域中，pr_sysctl指向函数sysctl_rtable，它在第19章中讨论。

7.10 小结

本章从说明结构domain和protosw开始，这两个结构在Net/3内核中描述及组织协议。我们看到一个域的所有protosw结构在编译时分配在一个数组中，inetdomain和数组inetsw描述Internet协议。我们仔细查看了三个描述IP协议的inetsw项：一个用于内核访问IP，其他两个用于进程访问IP。

在系统初始化时，domainint将域链接到domains列表中，调用域和协议初始化函数，并调用快速和慢速超时函数。

两个函数pffindproto和pffindtype通过协议号或类型搜索域和协议列表。pfctlinput发送一个控制命令给所有协议。

最后，我们说明了IP初始化程序，它通过数组ip_protox完成传输分用。

习题

7.1 由谁调用pfsfindproto会返回一个指向inetsw[6]指针？

第8章　IP：网际协议

8.1　引言

本章我们介绍IP分组的结构和基本的IP处理过程，包括输入、转发和输出。假定读者熟悉IP协议的基本操作，其他IP的背景知识见卷1的第3、9和12章。RFC 791 [Postel 1981a] 是IP的官方规范，RFC 1122 [Braden 1989a] 中有RFC 791的说明。

第9章将讨论选项的处理，第10章讨论分片和重装。图8-1显示了IP层常见的组织形式。

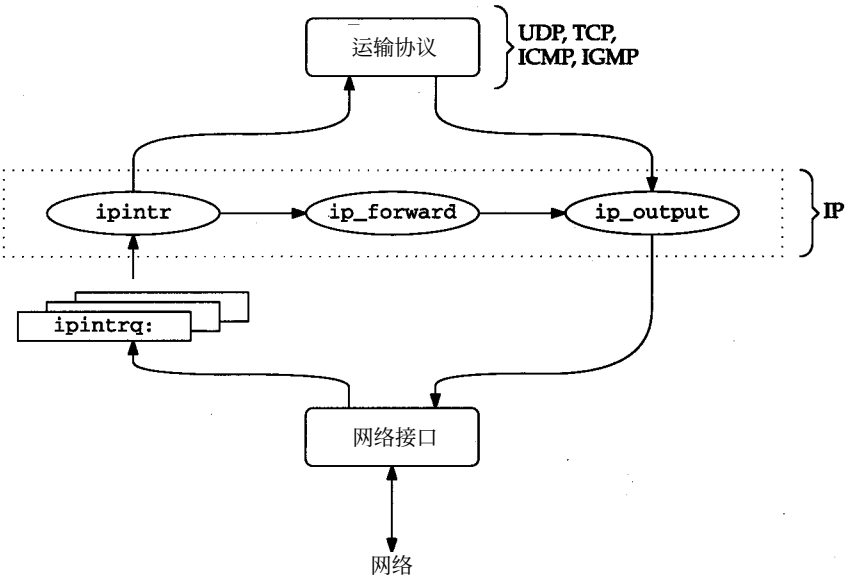


图8-1　IP层的处理

在第4章中，我们看到网络接口如何把到达的IP分组放到IP输入队列ipintrq中去,并如何调用一个软件中断。因为硬件中断的优先级比软件中断的要高，所以在发生一次软件中断之前，有的分组可能会被放到队列中。在软件中断处理中，ipintr函数不断从ipintrq中移走和处理分组，直到队列为空。在最终的目的地，IP把分组重装为数据报,并通过函数调用把该数据报直接传给适当的运输层协议。如果分组没有到达最后的目的地，并且如果主机被配置成一个路由器，则IP把分组传给ip_forward。传输协议和ip_forward把要输出的分组传给ip_output，由ip_output完成IP首部、选择输出接口以及在必要时对分组分片。最终的分组被传给合适的网络接口输出函数。

当产生差错时，IP丢弃该分组，并在某些条件下向分组的源站发出一个差错报文。这些报文是ICMP(第11章)的一部分。Net/3通过调用icmp_error发出ICMP差错报文，icmp_error接收一个mbuf，其中包含差错分组、发现的差错类型以及一个选项码，提供依赖于差错类型的附加信息。

本章我们讨论IP为什么以及何时发送ICMP报文，至于有关ICMP本身的详细讨论将在第11章进行。

8.2 代码介绍

本章讨论两个头文件和三个C文件。如图8-2所示。

文 件	描 述
net/route.h	路由入口
netinet/ip.h	IP首部结构
netinet/ip_ input.c	IP输入处理
netinet/ip_ output.c	IP输出处理
netinet/ip_ cksum.c	Internet检验和算法

图8-2 本章描述的文件

8.2.1 全局变量

在IP处理代码中出现了几个全局变量，见图 8-3。

变 量	数据类型	描 述
in_ifaddr	struct in_ ifaddr *	IP地址清单
ip_defttl	int	IP分组的默认TTL
ip_id	int	赋给输出的IP分组的上一个ID
ip_protox	int[]	IP分组的分路矩阵
ipforwarding	int	系统是否转发IP分组？
ipforward_rt	struct route	大多数最近转发的路由的缓存
ipintrq	struct ifqueue	IP输入队列
ipqmaxlen	int	IP输入队列的最大长度
ipsendredirects	int	系统是否发送ICMP重定向?
ipstat	struct ipstat	IP统计

图8-3 本章中引入的全局变量

8.2.2 统计量

IP收集的所有统计量都放在图 8-4描述的ipstat结构中。图8-5显示了由netstat-s命令得到的一些统计输出样本。统计是在主机启动 30天后收集的。

ipstat成员	描 述	SNMP使用的
ips_badhlen	IP首部长度无效的分组数	•
ips_badlen	IP首部和IP数据长度不一致的分组数	•
ips_badoptions	在选项处理中发现差错的分组数	•
ips_badsum	检验和差错的分组数	•
ips_badvers	IP版本不是4的分组数	•
ips_cantforward	目的站不可到达的分组数	•
ips_delivered	向高层交付的数据报数	•

图8-4 本章收集的统计

ipstat成员	描　述	SNMP使用的
ips_forward	转发的分组数	•
ips_fragdropped	分片丢失数(副本或空间不足)	•
ips_fragments	收到分片数	•
ips_fragtimeout	超时的分片数	•
ips_noproto	具有未知或不支持的协议的分组数	•
ips_reassembled	重装的数据报数	•
ips_tooshort	具有无效数据长度的分组数	•
ips_toosmall	无法包含IP分组的太小的分组数	•
ips_total	全部接收到的分组数	•
ips_cantfrag	由于不分片比特而丢弃的分组数	•
ips_fragmented	成功分片的数据报数	•
ips_localout	系统生成的数据报数(即没有转发的)	•
ips_noroute	丢弃的分组数 —— 到目的地没有路由	•
ips_odropped	由于资源不足丢掉的分组数	•
ips_ofragments	为输出产生的分片数	•
ips_rawout	全部生成的原始ip分组数	
ips_redirectsent	已发送的重定向报文数	

图8-4 (续)

netstat -s 输出	ipstat 成员
27,881,978 total packets received	ips_total
6 bad header checksums	ips_badsum
9 with size smaller than minimum	ips_tooshort
14 with data size < data length	ips_toosmall
0 with header length < data size	ips_badhlen
0 with data length < header length	ips_badlen
0 with bad options	ips_badoptions
0 with incorrect version number	ips_badvers
72,786 fragments received	ips_fragments
0 fragments dropped (dup or out of space)	ips_fragdropped
349 fragments dropped after timeout	ips_fragtimeout
16,557 packets reassembled ok	ips_reassembled
27,390,665 packets for this host	ips_delivered
330,882 packets for unknown/unsupported protocol	ips_noproto
97,939 packets forwarded	ips_forward
6,228 packets not forwardable	ips_cantforward
0 redirects sent	ips_redirectsent
29,447,726 packets sent from this host	ips_localout
769 packets sent with fabricated ip header	ips_rawout
0 output packets dropped due to no bufs, etc.	ips_odropped
0 output packets discarded due to no route	ips_noroute
260,484 output datagrams fragmented	ips_fragmented
796,084 fragments created	ips_ofragments
0 datagrams that can't be fragmented	ips_cantfrag

图8-5 IP统计样本

ips_noproto的值很高，因为当没有进程准备接收报文时，它能对 ICMP主机不可达报文进行计数。见第32.5节的详细讨论。

8.2.3 SNMP变量

图8-6显示了IP组和Net/3收集的统计中的SNMP变量之间的关系。

SNMP变量	Ipstat成员	描 述
ipDefaultTTL ipForwarding ipReasmTimeout	ip_defttl ipforwarding IPFRAGTTL	数据报的默认TTL(64跳) 系统是路由器吗？ 分片的重装超时(30秒)
ipInReceives	ips_total	收到的全部IP分组数
ipInHdrErrors	ips_badsum+ ips_tooshort+ ips_toosmall+ ips_badhlen+ ips_badlen+ ips_badoptions+ ips_badvers	IP首部出错的分组数
ipInAddrErrors ipForwDatagrams ipReasmReqds ipReasmFails ipReasmOKs ipInDiscards ipInUnknownProtos ipInDelivers	ips_cantforward ips_forward ips_fragments ips_fragdropped+ ips_fragtimeout ips_reassembled (未实现) ips_noproto ips_delivered	由于错误交付而丢弃的IP分组数(ip_output也失败) 转发的IP分组数 收到的分片数 丢失的分片数 成功地重装的数据报数 由于资源限制而丢弃的数据报数 具有未知或不支持的协议的数据报数 交付到运输层的数据报数
ipOutRequests ipFragOKs iPFragFails ipFragCreates ipOutDiscards ipOutRoutes	ips_localout ips_fragmented ips_cantfrag ips_ofragments ips_odropped ips_noroute	由运输层产生的数据报数 分片成功的数据报数 由于不分片比特丢弃的IP分组数 为输出产生的分片数 由于资源短缺丢失的IP分组数 由于没有路由丢弃的IP分组数

图8-6 IP组中SNMP变量的例子

8.3 IP分组

为了更准确地讨论 Internet协议处理，我们必须定义一些名词。图 8-7显示了在不同的Internet层之间传递数据时用来描述数据的名词。

我们把传输协议交给 IP的数据称为报文。典型的报文包含一个运输层首部和应用程序数据。图8-7所示的传输协议是 UDP。IP在报文的首部前加上它自己的首部形成一个数据报。如果在选定的网络中，数据报的长度太大， IP就把数据报分裂成几个分片，每个分片中含有它自己的IP首部和一段原来数据报的数据。图 8-7显示了一个数据报被分成三个分片。

当提交给数据链路层进行传送时，一个 IP分片或一个很小的无需分片的 IP数据报称为分组。数据链路层在分组前面加上它自己的首部，并发送得到的帧。

IP只考虑它自己加上的 IP首部，对报文本身既不检查也不修改 (除非进行分片)。图8-8显示了IP首部的结构。

图8-8包括ip结构(如图8-9)中各成员的名字， Net/3通过该结构访问IP首部。

47-67 因为在存储器中，比特字段的物理顺序依机器和编译器的不同而不同，所以由 #ifs保证编译器按照IP标准排列结构成员。从而，当 Net/3把一个ip结构覆盖到存储器中的一个 IP

分组上时，结构成员能够访问到分组中正确的比特。

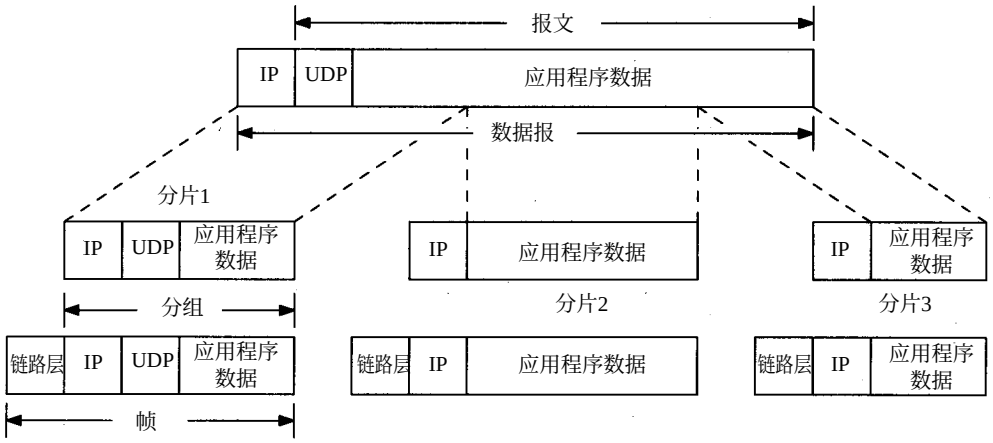


图8-7 帧、分组、分片、数据报和报文

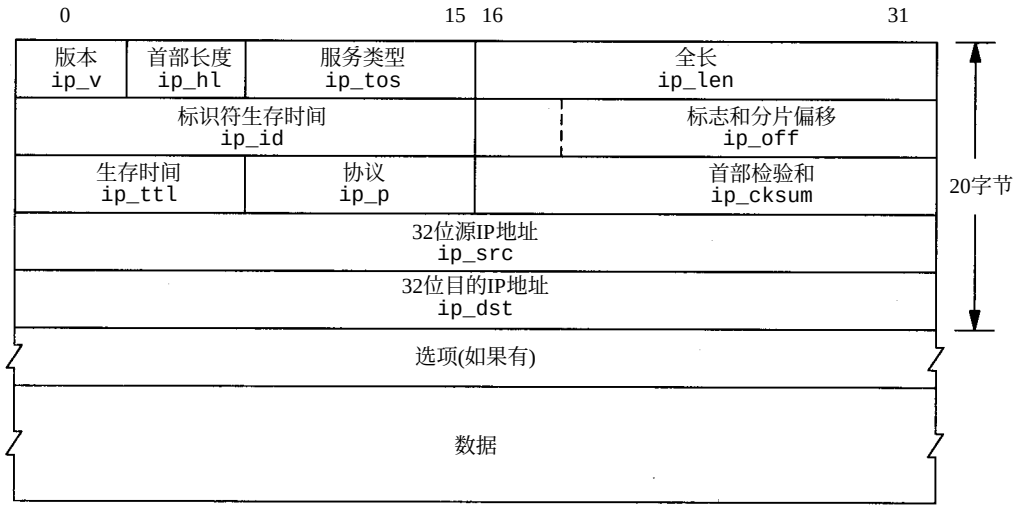


图8-8 IP数据报，包括ip结构名

ip.h

```
40 /*
41  * Structure of an internet header, naked of options.
42  *
43  * We declare ip_len and ip_off to be short, rather than u_short
44  * pragmatically since otherwise unsigned comparisons can result
45  * against negative integers quite easily, and fail in subtle ways.
46  */
47 struct ip {
48 #if BYTE_ORDER == LITTLE_ENDIAN
49     u_char  ip_hl:4,            /* header length */
50             ip_v:4;             /* version */
51 #endif
52 #if BYTE_ORDER == BIG_ENDIAN
53     u_char  ip_v:4,             /* version */
54             ip_hl:4;            /* header length */
55 #endif
```

图8-9 ip结构

```
56      u_char  ip_tos;             /* type of service */
57      short   ip_len;             /* total length */
58      u_short ip_id;              /* identification */
59      short   ip_off;             /* fragment offset field */
60  #define IP_DF 0x4000            /* dont fragment flag */
61  #define IP_MF 0x2000            /* more fragments flag */
62  #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
63      u_char  ip_ttl;             /* time to live */
64      u_char  ip_p;               /* protocol */
65      u_short ip_sum;             /* checksum */
66      struct in_addr ip_src, ip_dst;  /* source and dest address */
67  };
```

ip.h

图8-9 (续)

IP首部中包含IP分组格式、内容、寻址、路由选择以及分片的信息。

IP分组的格式由版本ip_v指定，通常为4；首部长度ip_hl，通常以4字节单元度量；分组长度ip_len以字节为单位度量；传输协议ip_p生成分组内数据；ip_sum是检验和，检测在发送中首部的变化。

标准的IP首部长度是20个字节，所以ip_hl必须大于或等于5。大于5表示IP选项紧跟在标准首部后。如ip_hl的最大值为15 (2^4-1)，允许最多40个字节的选项(20+40=60)。IP数据报的最大长度为65535 ($2^{16}-1$)字节，因为ip_len是一个16 bit的字段。图8-10是整个构成。

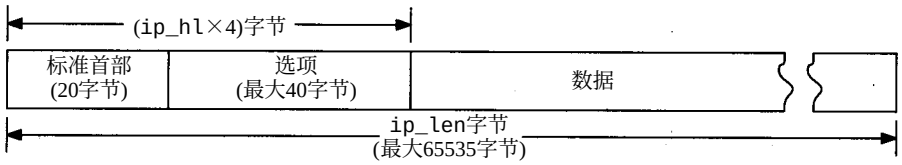


图8-10 有选项的IP分组构成

因为ip_hl是以4字节为单元计算的，所以IP选项必须常常被填充成4字节的倍数。

8.4 输入处理：ipintr函数

在第3、第4和第5章中，我们描述了示例的网络接口如何对到达的数据报排队以等待协议处理：

1) 以太网接口用以太网首部中的类型字段分路到达帧 (见4.3节)；

2) SLIP接口只处理IP分组，所以无需分用(见5.3节)；

3) 环回接口在looutput函数中结合输入和输出处理，用目的地址中的sa_family成员对数据报分用(见5.4节)。

在以上情况中，当接口把分组放到ipintrq上排队后，通过schednetisr调用一个软中断。当该软中断发生时，如果IP处理过程已经由schednetisr调度，则内核调用ipintr。在调用ipintr之前，CPU的优先级被改变成splnet。

8.4.1 ipintr概观

ipintr是一个大函数，我们将在4个部分中讨论：(1)对到达分组验证；(2)选项处理及转发；(3)分组重装；(4)分用。在ipintr中发生分组的重装，但比较复杂，我们将单独放在第

10章中讨论。图8-11显示了ipintr的整体结构。

ip_input.c

```
100 void
101 ipintr()
102 {
103     struct ip *ip;
104     struct mbuf *m;
105     struct ipq *fp;
106     struct in_ifaddr *ia;
107     int     hlen, s;

108   next:
109     /*
110      * Get next datagram off input queue and get IP header
111      * in first mbuf.
112      */
113     s = splimp();
114     IF_DEQUEUE(&ipintrq, m);
115     splx(s);
116     if (m == 0)
117         return;

                /* input packet processing */
            /* Figures 8.12, 8.13, 8.15, 10.11, and 11.16 */

332     goto next;
333   bad:
334     m_freem(m);
335     goto next;
336 }
```

ip_input.c

图8-11 ipintr 函数

100-117 标号next标识主要的分组处理循环的开始。ipintr从ipintrq中移走分组，并对之加以处理直到整个队列空为止。如果到函数最后控制失败，goto把控制权传回给next中最上面的函数。ipintr把分组阻塞在splimp内，避免当它访问队列时，运行网络的中断程序(例如slinput和ether_input)。

332-336 标号bad标识由于释放相关mbuf并返回到next中处理循环的开始而自动丢弃的分组。在整个ipintr中，都是跳到bad来处理差错。

8.4.2 验证

我们从图8-12开始：把分组从ipintrq中取出，验证它们的内容。损坏和有差错的分组被自动丢弃。

ip_input.c

```
118     /*
119      * If no IP addresses have been set yet but the interfaces
120      * are receiving, can't do anything with incoming packets yet.
121      */
122     if (in_ifaddr == NULL)
123         goto bad;
124     ipstat.ips_total++;
125     if (m->m_len < sizeof(struct ip) &&
```

图8-12 ipintr 函数

```
126                 (m = m_pullup(m, sizeof(struct ip))) == 0) {
127         ipstat.ips_toosmall++;
128         goto next;
129     }
130     ip = mtod(m, struct ip *);
131     if (ip->ip_v != IPVERSION) {
132         ipstat.ips_badvers++;
133         goto bad;
134     }
135     hlen = ip->ip_hl << 2;
136     if (hlen < sizeof(struct ip)) {     /* minimum header length */
137         ipstat.ips_badhlen++;
138         goto bad;
139     }
140     if (hlen > m->m_len) {
141         if ((m = m_pullup(m, hlen)) == 0) {
142             ipstat.ips_badhlen++;
143             goto next;
144         }
145         ip = mtod(m, struct ip *);
146     }
147     if (ip->ip_sum = in_cksum(m, hlen)) {
148         ipstat.ips_badsum++;
149         goto bad;
150     }
151     /*
152      * Convert fields to host representation.
153      */
154     NTOHS(ip->ip_len);
155     if (ip->ip_len < hlen) {
156         ipstat.ips_badlen++;
157         goto bad;
158     }
159     NTOHS(ip->ip_id);
160     NTOHS(ip->ip_off);
161     /*
162      * Check that the amount of data in the buffers
163      * is as at least much as the IP header would have us expect.
164      * Trim mbufs if longer than we expect.
165      * Drop packet if shorter than we expect.
166      */
167     if (m->m_pkthdr.len < ip->ip_len) {
168         ipstat.ips_tooshort++;
169         goto bad;
170     }
171     if (m->m_pkthdr.len > ip->ip_len) {
172         if (m->m_len == m->m_pkthdr.len) {
173             m->m_len = ip->ip_len;
174             m->m_pkthdr.len = ip->ip_len;
175         } else
176             m_adj(m, ip->ip_len - m->m_pkthdr.len);
177     }
```

ip_input.c

图8-12 (续)

1. IP版本

118-134 如果in_ifaddr表(见第6.5节)为空，则该网络接口没有指派IP地址，ipintr必须丢掉所有的IP分组；没有地址，ipintr就无法决定该分组是否要到该系统。通常这是一种

暂时情况，是当系统启动时，接口正在运行但还没有配置好时发生的。我们在 6.3节中介绍了地址是如何分配的问题。

在ipintr访问任何IP首部字段之前，它必须证实ip_v是4(IPVERSION)。RFC 1122要求某种实现丢弃那些具有无法识别版本号的分组而不回显信息。

Net/2不检查ip_v。目前大多数正在使用的IP实现，包括Net/2，都是在IP的版本4之后产生的，因此无需区分不同 IP版本的分组。因为目前正在对 IP进行修正，所以不久将来的实现都将检查ip_v。

IEN 119 [Forgie 1979] 和RFC 1190 [Topolcic 1990] 描述了使用IP版本5和6的实验协议。版本6还被选为下一个正式的IP标准(IPv6)。保留版本0和15，其他的没有赋值。

在C中，处理位于一个无类型存储区域中数据的最简单的方法是：在该存储区域上覆盖一个结构，转而处理该结构中的各个成员，而不再对原始的字节进行操作。如第 2章所言， mbuf链把一个字节的逻辑序列，例如一个 IP分组，储存在多个物理 mbuf中，各mbuf相互连接在一个链表上。因为覆盖技术也可用于 IP分组的首部，所以首部必须驻留在一段连续的存储区内(也就是说，不能把首部分开放在不同的存储器缓存区)。

135-146 下面的步骤保证IP首部(包括选项)位于一段连续的存储器缓存区上：

- 如果在第一个mbuf中的数据小于一个标准的 IP首部(20字节)， m_pullup会重新把该标准首部放到一个连续的存储器缓存区上去。

链路层不太可能会把最大的 (60字节) IP首部分在两个 mbuf中从而使用上面的m_pullup。

- ip_hl通过乘以4得到首部字节长度，并将其保存在hlen中。
- 如果IP分组首部的字节数长度hlen小于标准首部(20字节)，将是无效的并被丢弃。
- 如果整个首部仍然不在第一个 mbuf中(也就是说，分组包含了 IP选项)，则由m_pullup完成其任务。

同样，这不一定是必需的。

检验和计算是所有 Internet协议的重要组成。所有的协议均使用相同的算法 (由函数in_cksum完成)，但应用于分组的不同部分。对 IP来说，检验和只保证 IP的首部(以及选项，如果有的话)。对传输协议，如UDP或TCP，检验和覆盖了分组的数据部分和运输层首部。

2. IP检验和

147-150 ipintr把由in_cksum计算出来的检验和保存首部的ip_sum字段中。一个未被破坏的首部应该具有0检验和。

正如我们将在8.7节中看到的，在计算到达分组的检验和之前，必须对 ip_sum清零。通过把in_cksum中的结果储存在ip_sum中，就为分组转发作好了准备 (尽管还没有减小TTL)。ip_output函数不依赖这项操作；它为转发的分组重新计算检验和。

如果结果非零，则该分组被自动丢弃。我们将在 8.7节中详细讨论in_cksum。

3. 字节顺序

151-160 Internet标准在指定协议首部中多字节整数值的字节顺序时非常小心。NTOHS把IP首部中所有16 bit的值从网络字节序转换成主机字节序：分组长度(ip_len)、数据报标识符(ip_id)和分片偏移(ip_off)。如果两种格式相同，则NTOHS是一个空的宏。在这里就转换成主机字节序，以避免Net/3每次检查该字段时必须进行一次转换。

4. 分组长度

161-177 如果分组的逻辑长度(ip_len)比储存在mbuf中的数据量(m_pkthdr.len)大，并且有些字节被丢失了，就必须丢弃该分组。如果mbuf比分组大，则去掉多余的字节。

丢失字节的一个常见原因是因为数据到达某个没有或只有很少缓存的串口设备，例如许多个人计算机。设备丢弃到达的字节，而IP丢弃最后的分组。

多余的字节可能产生，如在某个以太网设备上，当一个IP分组的大小比以太网要求的最小长度还小时。发送该帧时加上的多余字节就在这里被丢掉。这就是为什么IP分组的长度被保存在首部的原因之一；IP允许链路层填充分组。

现在，有了完整的IP首部，分组的逻辑长度和物理长度相同，检验和表明分组的首部无损地到达。

8.4.3 转发或不转发

图8-13显示了ipintr的下一部分，调用ip_dooptions(见第9章)来处理IP选项，然后决定分组是否到达它最后的目的地。如果分组没有到达最后目的地，Net/3会尝试转发该分组(如果系统被配置成路由器)。如果分组到达最后目的地，就被交付给合适的运输层协议。

ip_input.c

```
178     /*
179      * Process options and, if not destined for us,
180      * ship it on.  ip_dooptions returns 1 when an
181      * error was detected (causing an icmp message
182      * to be sent and the original packet to be freed).
183      */
184     ip_nhops = 0;               /* for source routed packets */
185     if (hlen > sizeof(struct ip) && ip_dooptions(m))
186                 goto next;

187     /*
188      * Check our list of addresses, to see if the packet is for us.
189      */
190     for (ia = in_ifaddr; ia; ia = ia->ia_next) {
191 #define satosin(sa) ((struct sockaddr_in *)(sa))

192         if (IA_SIN(ia)->sin_addr.s_addr == ip->ip_dst.s_addr)
193             goto ours;

194         /* Only examine broadcast addresses for the receiving interface */
195         if (ia->ia_ifp == m->m_pkthdr.rcvif &&
196             (ia->ia_ifp->if_flags & IFF_BROADCAST)) {
197             u_long  t;

198             if (satosin(&ia->ia_broadaddr)->sin_addr.s_addr ==
199                 ip->ip_dst.s_addr)
200                 goto ours;
201             if (ip->ip_dst.s_addr == ia->ia_netbroadcast.s_addr)
```

图8-13 续ipintr

```
202                 goto ours;
203             /*
204              * Look for all-0's host part (old broadcast addr),
205              * either for subnet or net.
206              */
207             t = ntohl(ip->ip_dst.s_addr);
208             if (t == ia->ia_subnet)
209                 goto ours;
210             if (t == ia->ia_net)
211                 goto ours;
212         }
213     }
```

```
/* multicast code (Figure 12.39) */
```

```
258     if (ip->ip_dst.s_addr == (u_long) INADDR_BROADCAST)
259         goto ours;
260     if (ip->ip_dst.s_addr == INADDR_ANY)
261         goto ours;
262     /*
263      * Not for us; forward if possible and desirable.
264      */
265     if (ipforwarding == 0) {
266         ipstat.ips_cantforward++;
267         m_freem(m);
268     } else
269         ip_forward(m, 0);
270     goto next;
271 ours:
```

ip_input.c

图8-13 (续)

1. 选项处理

178-186 通过对ip_nhops(见9.6节)清零，丢掉前一个分组的原路由。如果分组首部大于默认首部，它必然包含由 ip_dooptions处理的选项。如果 ip_dooptions返回0，ipintr将继续处理该分组；否则， ip_dooptions通过转发或丢弃分组完成对该分组的处理，ipintr可以处理输入队列中的下一个分组。我们把对选项的进一步讨论放到第 9章进行。

处理完选项后，ipintr通过把IP首部内的ip_dst与配置的所有本地接口的IP地址比较，以决定分组是否已到达最终目的地。 ipintr必须考虑与接口相关的几个广播地址、一个或多个单播地址以及任意个多播地址。

2. 最终目的地

187-261 ipintr通过遍历in_ifaddr(图6-5)，配置好的Internet地址表，来决定是否有与分组的目的地址的匹配。对清单中的每个 in_ifaddr结构进行一系列的比较。要考虑 4种常见的情况：

- 与某个接口地址的完全匹配(图8-14中的第一行)，
- 与某个与接收接口相关的广播地址的匹配(图8-14的中间4行)，
- 与某个与接收接口相关的多播组之一的匹配(图12-39)，或
- 与两个受限的广播地址之一的匹配(图8-14的最后一行)。

图8-14显示的是当分组到达我们的示例网络里的主机 sun上的以太网接口时要测试的地址，将在第12章中讨论的多播地址除外。

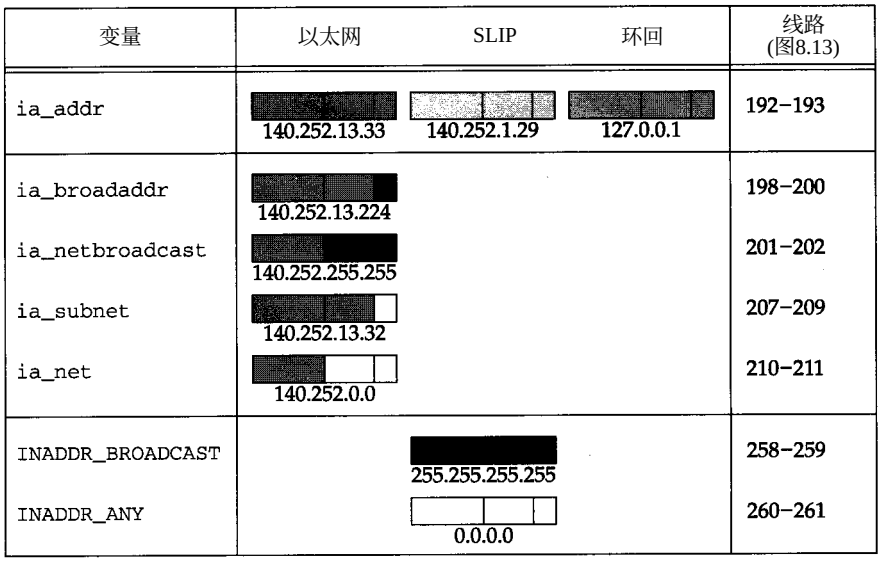


图8-14 为判定分组是否到达最终目的地进行的比较

对ia_subnet、ia_net和INADDR_ANY的测试不是必需的，因为它们表示的是4.2BSD使用的已经过时的广播地址。但不幸的是，许多 TCP/IP实现是从4.2BSD派生而来的，所以，在某些网络中能够识别出这些旧广播地址可能十分重要。

3. 转发

262-271 如果ip_dst与所有地址都不匹配，分组还没有到达最终目的地。如果还没有设置ipforwarding，就丢弃分组。否则，ip_forward尝试把分组路由到它的最终目的地。

当分组到达的某个地址不是目的地址指定的接口时，主机会丢掉该分组。在这种情况下，Net/3将搜索整个in_ifaddr表；只考虑那些分配给接收接口的地址。RFC 1122 称此为强端系统(strong end system)模型。

对多主主机而言，很少出现分组到达接口地址与其目的地址不对应的情况，除非配置了特定的主机路由。主机路由强迫相邻的路由器把多主主机作为分组的下一跳路由器。弱端系统(weak end system)模型要求该主机接收这些分组。实现人员可以随意选择两种模型。Net/3实现弱端系统模型。

8.4.4 重装和分用

最后，我们来看ipintr的最后一部分(图8-15)，在这里进行重装和分用。我们略去了重装的代码，推迟到第10章讨论。当无法重装完全的数据报时，略去的代码将把指针 ip设成空。否则，ip指向一个已经到达目的地的完整数据报。

运输分用

325-332 数据报中指定的协议被ip_p用ip_protox数组(图7-22)映射到inetsw数组的下

标。ipintr调用选定的protosw结构中的pr_input函数来处理数据报包含的运输报文。当pr_input返回时，ipintr继续处理ipintrq中的下一个分组。

注意，运输层对分组的处理发生在ipintr处理循环的内部。在IP和传输协议之间没有到达分组的排队，这与TCP/IP中SVR4流实现的排队不同。

ip_input.c

```
325     /*
326      * If control reaches here, ip points to a complete datagram.
327      * Otherwise, the reassembly code jumps back to next (Figure 8.11)
328      * Switch out to protocol's input routine.
329      */
330     ipstat.ips_delivered++;
331     (*inetsw[ip_protox[ip->ip_p]].pr_input) (m, hlen);
332     goto next;
```

ip_input.c

图8-15 续ipintr

8.5 转发：ip_forward函数

到达非最终目的地系统的分组需要被转发。只有当 ipforwarding非零(6.1节)或当分组中包含源路由(9.6节)时，ipintr才调用实现转发算法的 ip_forward函数。当分组中包含源路由时，ip_dooptions调用ip_forward，并且第2个参数srcrt设为1。

ip_forward通过图8-16中显示的route结构与路由表接口。

route.h

```
46 struct route {
47     struct rtentry *ro_rt;      /* pointer to struct with information */
48     struct sockaddr ro_dst;     /* destination of this route */
49 };
```

route.h

图8-16 route 结构

46-49 route结构有两个成员： ro_rt，指向rtentry结构的指针；ro_dst，一个sockaddr结构，指定与ro_rt所指的路由项相关的目的地。目的地是在内核的路由表中用来查找路由信息的关键字。第18章对rtentry结构和路由表有详细的描述。

我们分两部分讨论ip_forward。第一部分确定允许系统转发分组，修改IP首部，并为分组选择路由。第二部分处理ICMP重定向报文，并把分组交给ip_output进行发送。见图8-17。

1. 分组适合转发吗

867-871 ip_froward的第1个参数是指向一个mbuf链的指针，该mbuf中包含了要被转发的分组。如果第2个参数srcrt为非零，则分组由于源路由选项(见9.6节)正在被转发。

879-884 if语句识别并丢弃以下分组。

• 链路层广播

任何支持广播的网络接口驱动器必须为收到的广播分组把 M_BCAST标志置位。如果分组寻址是到以太网广播地址，则ether_input(图4-13)就把M_BCAST置位。不转发链路层的广播分组。

RFC 1122不允许以链路层广播的方式发送一个寻址到单播 IP地址的分组，并在这里将该分组丢掉。

• 环回分组

对寻址到环回网络的分组， in_canforward返回0。这些分组将被ipintr提交给ip_forward，因为没有正确配置反馈接口。

• 网络0和E类地址

对这些分组， in_canforward返回0。 这些目的地址是无效的，而且因为没有主机接收这些分组，所以它们不应该继续在网络中流动。

• D类地址

寻址到D类地址的分组应该由多播函数 ip_mforward而不是由ip_forward处理。in_canforward拒绝D类(多播)地址。

RFC 791 规定处理分组的所有系统都必须把生存时间 (TTL)字段至少减去1，即使TTL是以秒计算的。由于这个要求， TTL通常被认为是对IP分组在被丢掉之前能经过的跳的个数的界限。从技术角度说，如果路由器持有分组超过 1秒，就必须把ip_ttl减去多于1。

ip_input.c

```
867 void
868 ip_forward(m, srcrt)
869 struct mbuf *m;
870 int     srcrt;
871 {
872     struct ip *ip = mtod(m, struct ip *);
873     struct sockaddr_in *sin;
874     struct rtentry *rt;
875     int     error, type = 0, code;
876     struct mbuf *mcopy;
877     n_long  dest;
878     struct ifnet *destifp;

879     dest = 0;
880     if (m->m_flags & M_BCAST || in_canforward(ip->ip_dst) == 0) {
881         ipstat.ips_cantforward++;
882         m_freem(m);
883         return;
884     }
885     HTONS(ip->ip_id);
886     if (ip->ip_ttl <= IPTTLDEC) {
887         icmp_error(m, ICMP_TIMXCEED, ICMP_TIMXCEED_INTRANS, dest, 0);
888         return;
889     }
890     ip->ip_ttl -= IPTTLDEC;

891     sin = (struct sockaddr_in *) &ipforward_rt.ro_dst;
892     if ((rt = ipforward_rt.ro_rt) == 0 ||
893         ip->ip_dst.s_addr != sin->sin_addr.s_addr) {
894         if (ipforward_rt.ro_rt) {
895             RTFREE(ipforward_rt.ro_rt);
896             ipforward_rt.ro_rt = 0;
897         }
898         sin->sin_family = AF_INET;
899         sin->sin_len = sizeof(*sin);
900         sin->sin_addr = ip->ip_dst;

901         rtalloc(&ipforward_rt);
```

图8-17 ip_forward 函数：路由选择

```
902             if (ipforward_rt.ro_rt == 0) {
903                 icmp_error(m, ICMP_UNREACH, ICMP_UNREACH_HOST, dest, 0);
904                 return;
905             }
906             rt = ipforward_rt.ro_rt;
907         }
908         /*
909          * Save at most 64 bytes of the packet in case
910          * we need to generate an ICMP message to the src.
911          */
912         mcopy = m_copy(m, 0, imin((int) ip->ip_len, 64));
913         ip_ifmatrix[rt->rt_ifp->if_index +
914                     if_index * m->m_pkthdr.rcvif->if_index]++;
```

ip_input.c

图8-17 (续)

这就产生了一个问题：在Internet上，最长的路径有多长？这个度量称为网络的直径(diameter)。除了通过实验外无法知道直径的大小。[Olivier 1994]中有37跳的路径。

2. 减小TTL

885-890 由于转发时不再需要分组的标识符，所以标识符又被转换回网络字节序。但是当ip_forward发送包含无效IP首部的ICMP差错报文时，分组的标识符又应该是正确的顺序。

Net/3漏做了对已被ipintr转换成主机字节序的ip_len的转换。作者注意到在大头机器上，这不会产生问题，因为从未对字节进行过转换。但在小头机器如386上，这个小的漏洞允许交换了字节的值在ICMP差错中的IP首部中。返回从运行在386上的SVR4(可能是Net/1码)和AIX3.2(4.3BSD Reno码)返回的ICMP分组中可以观察到这个小的漏洞。

如果ip_ttl达到1(IPTTLDEC)，则向发送方返回一个ICMP超时报文，并丢掉该分组。否则，ip_forward把ip_ttl减去IPTTLDEC。

系统不接受TTL为0的IP数据报，但Net/3在即使出现这种情况时也能生成正确的ICMP差错，因为p_ttl是在分组被认为是在本地交付之后和被转发之前检测的。

3. 定位下一跳

891-907 IP转发算法把最近的路由缓存在全局route结构的ipforward_rt中，在可能时应用于当前分组。研究表明连续分组趋向于同一目的地址([Jain和Routhier 1986]和[Mogul 1991])，所以这种向后一个(one-behind)的缓存使路由查询的次数最少。如果缓存为空(ipforward_rt)或当前分组的目的地不是ipforward_rt中的路由，就取消前面的路由，ro_dst被初始化成新的目的地，rtalloc为当前分组的目的地找一个新路由。如果找不到路由，则返回一个ICMP主机不可达差错，并丢掉该分组。

908-914 由于在产生差错时，ip_output要丢掉分组，所以m_copy复制分组的前64个字节，以便ip_forward发送ICMP差错报文。如果调用m_copy失败，ip_forward并不终止。在这种情况下，不发送差错报文。ip_ifmatrix记录在接口之间进行路由选择的分组的个数。具有接收和发送接口索引的计数器是递增的。

重定向报文

当主机错误地选择某个路由器作为分组的第一跳路由器时，该路由器向源主机返回一个ICMP重定向报文。IP网络互连模型假定主机相对地并不知道整个互联网的拓扑结构，把维护正确路由选择的责任交给路由器。路由器发出重定向报文是向主机表明它为分组选择了一个不正确的路由。我们用图8-18说明重定向报文。

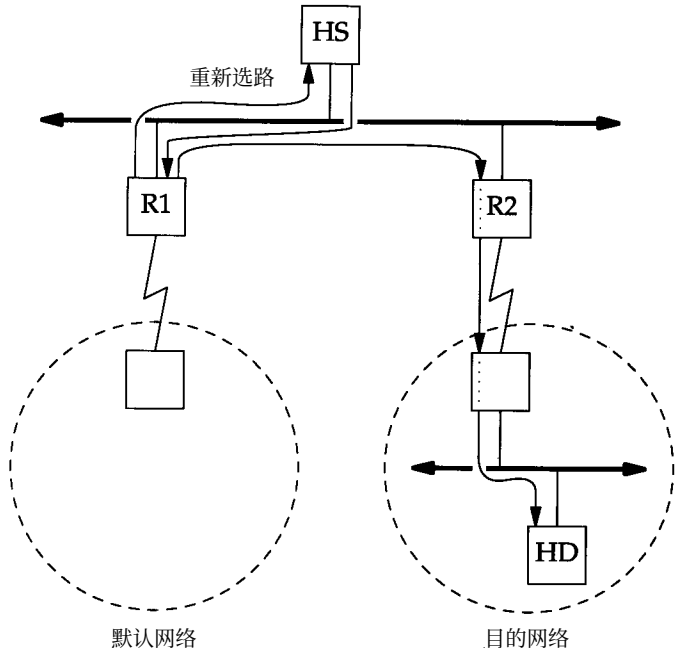


图8-18 路由器R1重定向主机HS使用路由器R2到达HD

通常，管理员对主机的配置是：把到远程网络的分组发送到某个默认路由器上。在图 8-18中，主机HS上R1被配置成它的默认路由器。当 HS首次向HD发送分组时，它不知道R2是合适的选择，而把分组发给 R1。R1识别出差错，就把分组转发给 R2，并向HS发回一个重定向报文。接收到重定向报文后， HS更新它的路由表，下一次发往HD的分组就直接发给R2。

RFC 1122推荐只有路由器才发重定向报文，而主机在接收到 ICMP重定向报文后必须更新它们的路由表 (11.8节)。因为Net/3只在系统被配置成路由器时才调用 `ip_forward`，所以Net/3采用RFC 1122的推荐。

在图8-19中，`ip_forward`决定是否发重定向报文。

1. 在接收接口上离开吗

915-929 路由器识别重定向情况的规则很复杂。首先，只有在同一接口 (`rt_ifp`和`rcvif`)上接收或重发分组时，才能应用重定向。其次，被选择的路由本身必须没有被 ICMP重定向报文创建或修改过 (`RTF_DYNAMIC|RTF_MODIFIED`)，而且该路由也不能是到默认目的地的(0.0.0.0)。这就保证系统在未授权时不会生成路由选择信息，并且不与其他系统共享自己的默认路由。

通常，路由选择协议使用特殊目的地址 0.0.0.0定位默认路由。当到某目的地的某个路由不能使用时，与目的地 0.0.0.0相关的路由就把分组定向到一个默认路由器上。

第18章对默认路由有详细的讨论。

全局整数 ipsendredirects指定系统是否被授权发送重定向 (第8.9节)，ipsendredirects的默认值为1。当传给ip_forward的参数srcrt指明系统是对分组路由选择的源时，禁止系统重定向，因为假定源主机要覆盖中间路由器的选择。

ip_input.c

```
915     /*
916      * If forwarding packet is using same interface that it came in on,
917      * perhaps should send a redirect to sender to shortcut a hop.
918      * Only send redirect if source is sending directly to us,
919      * and if packet was not source routed (or has any options).
920      * Also, don't send redirect if forwarding using a default route
921      * or a route modified by a redirect.
922      */
923 #define satosin(sa) ((struct sockaddr_in *)(sa))
924     if (rt->rt_ifp == m->m_pkthdr.rcvif &&
925         (rt->rt_flags & (RTF_DYNAMIC | RTF_MODIFIED)) == 0 &&
926         satosin(rt_key(rt))->sin_addr.s_addr != 0 &&
927         ipsendredirects && !srcrt) {
928 #define RTA(rt) ((struct in_ifaddr *)(rt->rt_ifa))
929         u_long  src = ntohl(ip->ip_src.s_addr);

930         if (RTA(rt) &&
931             (src & RTA(rt)->ia_subnetmask) == RTA(rt)->ia_subnet) {
932             if (rt->rt_flags & RTF_GATEWAY)
933                 dest = satosin(rt->rt_gateway)->sin_addr.s_addr;
934             else
935                 dest = ip->ip_dst.s_addr;
936             /* Router requirements says to only send host redirects */
937             type = ICMP_REDIRECT;
938             code = ICMP_REDIRECT_HOST;
939         }
940     }
```

ip_input.c

图8-19 ip_forward (续)

2. 发送重定向吗

930-931 这个测试决定分组是否产生于本地子网。如果源地址的子网掩码位和输出接口的地址相同，则两个地址位于同一 IP网络中。如果源接口和输出的接口位于同一网络中，则该系统就不应该接收这个分组，因为源站可能已经把分组发给正确的第一跳路由器了。 ICMP重定向报文告诉主机正确的第一跳目的地。如果分组产生于其他子网，则前一系统是个路由器，这个系统就不应该发重定向报文；差错由路由选择协议纠正。

在任何情况下，都要求路由器忽略重定向报文。尽管如此，当 ipforwarding 被置位时(也就是说，当它被配置成路由器时)，Net/3并不丢掉重定向报文。

3. 选择合适的路由器

932-940 ICMP重定向报文中包含正确的下一个系统的地址，如果目的主机不在直接相连的网络上时，该地址是一个路由器的地址；当目的主机在直接相连的网络中时，该地址是主机地址。

RFC 792描述了重定向报文的 4种类型：(1)网络；(2)主机；(3)TOS和网络；(4)TOS和主机。RFC 1009推荐在任何时候都不发送网络重定向报文，因为无法保证接收到重定向报文的主机能为目的网络找到合适的子网掩码。 RFC 1122推荐主机把网络重定向看作是主机重定向，

以避免二义性。Net/3只发送主机重定向报文，并省略所有对 TOS的考虑。在图 8-20中，ipintr把分组和所有的ICMP报文提交给链路层。

ip_input.c

```
941     error = ip_output(m, (struct mbuf *) 0, &ipforward_rt,
942                       IP_FORWARDING | IP_ALLOWBROADCAST, 0);
943     if (error)
944         ipstat.ips_cantforward++;
945     else {
946         ipstat.ips_forward++;
947         if (type)
948             ipstat.ips_redirectsent++;
949         else {
950             if (mcopy)
951                 m_freem(mcopy);
952             return;
953         }
954     }
955     if (mcopy == NULL)
956         return;
957     destifp = NULL;

958     switch (error) {

959     case 0:                 /* forwarded, but need redirect */
960         /* type, code set above */
961         break;

962     case ENETUNREACH:           /* shouldn't happen, checked above */
963     case EHOSTUNREACH:
964     case ENETDOWN:
965     case EHOSTDOWN:
966     default:
967         type = ICMP_UNREACH;
968         code = ICMP_UNREACH_HOST;
969         break;

970     case EMSGSIZE:
971         type = ICMP_UNREACH;
972         code = ICMP_UNREACH_NEEDFRAG;
973         if (ipforward_rt.ro_rt)
974             destifp = ipforward_rt.ro_rt->rt_ifp;
975         ipstat.ips_cantfrag++;
976         break;

977     case ENOBUFS:
978         type = ICMP_SOURCEQUENCH;
979         code = 0;
980         break;
981     }
982     icmp_error(mcopy, type, code, dest, destifp);
983 }
```

ip_input.c

图8-20 ip_forward(续)

重定向报文的标准化是在子网化之前，在一个非子网化的互联网中，网络重定向很有用，但在一个子网化的互联网中，由于重定向报文中没有有关子网掩码的信息，所以容易产生二义性。

4. 转发分组

941-954 现在，ip_forward有一个路由，并决定是否需要ICMP重定向报文。Ip_output把分组发送到路由ipforward_rt所指定的下一跳。IP_ALLOWBROADCAST标志位允许被转发分组是个到某局域网的广播。如果ip_output成功，并且不需要发送任何重定向报文，则丢掉分组的前64字节，ip_forward返回。

5. 发送ICMP差错报文？

955-983 ip_forward可能会由于ip_output失败或重定向而发送ICMP报文。如果没有原始分组的复制(可能当要复制时，曾经缓存不足)，则无法发送重定向报文，ip_forward返回。如果有重定向，type和code以前又被置位，但如果ip_output失败，switch语句基于从ip_output返回的值重新设置新的ICMP类型和码值。icmp_error发送该报文。来自失败的ip_outputICMP报文将覆盖任何重定向报文。

处理来自ip_output的差错的switch语句非常重要。它把本地差错翻译成适当的ICMP差错报文，并返回给分组的源站。图8-21对差错作了总结。第11章更详细地描述了ICMP报文。

当ip_output返回ENOBUFS时，Net/3通常生成ICMP源站抑制报文。Router Requirements(路由器需求)RFC [Almquist和Kastenholz 1994]不赞成源站抑制并要求路由器不产生这种报文。

来自ip_output的差错码	生成的ICMP报文	描　述
EMSGSIZE	ICMP_UNREACH_NEEDFRAG	对所选的接口来说，发出的分组太大，并且禁止分片(第10章)
ENOBUFS	ICMP_SOURCEQUENCH	接口队列满或内核运行内存不足。本报文向源主机指示降低数据率
EHOSTUNREACH		找不到到主机的路由
ENETDOWN		路由指明的输出接口没在运行
EHOSTDOWN	ICMP_UNREACH_HOST	接口无法把分组发给选定的主机
default		所有不识别的差错均作为ICMP_UNREACH_HOST差错报告

图8-21 来自ip_output 的差错

8.6 输出处理：ip_output函数

IP输出代码从两处接收分组：ip_forward和运输协议(图8-1)。让inetsw[0].pr_output能访问到IP输出操作似乎很有道理，但事实并非如此。标准的Internet传输协议(ICMP、IGMP、UDP和TCP)直接调用ip_output，而不查询inetsw表。对标准Internet传输协议而言，protosw结构不必具有一般性，因为调用函数并不是在与协议无关的情况下接入IP的。在第20章中，我们将看到与协议无关的路由选择插口调用pr_output接入IP。

我们分三个部分描述ip_ output：

- 首部初始化；
- 路由选择；和
- 源地址选择和分片。

8.6.1 首部初始化

图8-22显示了ip_output的第一部分，把选项与外出的分组合并，完成传输协议提交 (不是ip_forward提交的)的分组首部。

44-59 传给ip_output的参数包括：m0，要发送的分组；opt，包含的IP选项；ro，缓存的到目的地的路由；flags，见图8-23；imo，指向多播选项的指针，见第12章。

IP_FORWARDING被ip_forward和ip_mforward(多播分组转发)设置，并禁止ip_output重新设置任何IP首部字段。

ip_output.c

```
44 int
45 ip_output(m0, opt, ro, flags, imo)
46 struct mbuf *m0;
47 struct mbuf *opt;
48 struct route *ro;
49 int     flags;
50 struct ip_moptions *imo;
51 {
52     struct ip *ip, *mhip;
53     struct ifnet *ifp;
54     struct mbuf *m = m0;
55     int     hlen = sizeof(struct ip);
56     int     len, off, error = 0;
57     struct route iproute;
58     struct sockaddr_in *dst;
59     struct in_ifaddr *ia;

60     if (opt) {
61         m = ip_insertoptions(m, opt, &len);
62         hlen = len;
63     }
64     ip = mtod(m, struct ip *);
65     /*
66      * Fill in IP header.
67      */
68     if ((flags & (IP_FORWARDING | IP_RAWOUTPUT)) == 0) {
69         ip->ip_v = IPVERSION;
70         ip->ip_off &= IP_DF;
71         ip->ip_id = htons(ip_id++);
72         ip->ip_hl = hlen >> 2;
73         ipstat.ips_localout++;
74     } else {
75         hlen = ip->ip_hl << 2;
76     }
```

ip_output.c

图8-22 函数ip_output

标 志	描 述
IP_FORWARDING	这是一个转发过的分组
IP_ROUTETOIF	忽略路由表，直接路由到接口
IP_ALLOWBROADCAST	允许发送广播分组
IP_RAWOUTPUT	包含一个预构IP首部的分组

图8-23 ip_output ：flag 值

send、sendto和sendmsg的MSG_DONTROUTE标志使IP_ROUTETOIF有效，并进行一次写操作(见16.4)，而SO_DONTROUTE插口选项使IP_ROUTETOIF有效，并在某个特定插口上进行任意的写操作(见8.8节)。该标志被传输协议传给ip_ output。

IP_ALLOWBROADCAST标志可以被SO_BROADCAST插口选项(见8.8节)设置，但只被UDP提交。原来的 IP默认地设置 IP_ALLOWBROADCAST。TCP不支持广播，所以 IP_ALLOWBROADCAST不能被TCP提交给ip_output。不存在广播的预请求标志。

1. 构造IP首部

60-73 如果调用程序提供任何IP选项，它们将被ip_insertoptions(见9.8节)与分组合并，并返回新的首部长度。

我们将在8.8节中看到，进程可以设置 IP_OPTIONS插口选项来为一个插口指定 IP选项。插口的运输层(TCP或UDP)总是把这些选项提交给ip_output。

被转发分组(IP_FORWARDING)或有预构首部(IP_RAWOUTPUT)分组的IP首部不能被ip_output修改。任何其他分组(例如，产生于这个主机的UDP或TCP分组)需要有几个IP首部字段被初始化。ip_output把ip_v设置成4(IPVERSION)，把DF位需要的ip_off清零，并设置成调用程序提供的值(见第10章)，给来自全局整数的ip->ip_id赋一个唯一的标识符，把ip_id加1。ip_id是在协议初始化时由系统时钟设置的(见7.8节)。ip_hl被设置成用32 bit字度量的首部长度。

IP首部的其他字段——长度、偏移、TTL、协议、TOS和目的地址——已经被传输协议初始化了。源地址可能没被设置，因为是在确定了到目的地的路由后选择的(图8-25)。

2. 分组已经包括首部

74-76 对一个已转发的分组(或一个有首部的原始IP分组)，首部长度(以字节数度量)被保存在hlen中，留给将来分片算法使用。

8.6.2 路由选择

在完成IP首部后，ip_output的下一个任务就是确定一条到目的地的路由。见图 8-24所示。

ip_output.c

```
77      /*
78       * Route packet.
79       */
80      if (ro == 0) {
81          ro = &iproute;
82          bzero((caddr_t) ro, sizeof(*ro));
83      }
84      dst = (struct sockaddr_in *) &ro->ro_dst;
85      /*
86       * If there is a cached route,
87       * check that it is to the same destination
88       * and is still up.  If not, free it and try again.
89       */
90      if (ro->ro_rt && ((ro->ro_rt->rt_flags & RTF_UP) == 0 ||
91                        dst->sin_addr.s_addr != ip->ip_dst.s_addr)) {
92          RTFREE(ro->ro_rt);
93          ro->ro_rt = (struct rtentry *) 0;
94      }
```

图8-24 ip_output (续)

```
 95      if (ro->ro_rt == 0) {
 96          dst->sin_family = AF_INET;
 97          dst->sin_len = sizeof(*dst);
 98          dst->sin_addr = ip->ip_dst;
 99      }
100      /*
101       * If routing to interface only,
102       * short circuit routing lookup.
103       */
104 #define ifatoia(ifa)    ((struct in_ifaddr *)(ifa))
105 #define sintosa(sin)    ((struct sockaddr *)(sin))
106      if (flags & IP_ROUTETOIF) {
107          if ((ia = ifatoia(ifa_ifwithdstaddr(sintosa(dst)))) == 0 &&
108              (ia = ifatoia(ifa_ifwithnet(sintosa(dst)))) == 0) {
109              ipstat.ips_noroute++;
110              error = ENETUNREACH;
111              goto bad;
112          }
113          ifp = ia->ia_ifp;
114          ip->ip_ttl = 1;
115      } else {
116          if (ro->ro_rt == 0)
117              rtalloc(ro);
118          if (ro->ro_rt == 0) {
119              ipstat.ips_noroute++;
120              error = EHOSTUNREACH;
121              goto bad;
122          }
123          ia = ifatoia(ro->ro_rt->rt_ifa);
124          ifp = ro->ro_rt->rt_ifp;
125          ro->ro_rt->rt_use++;
126          if (ro->ro_rt->rt_flags & RTF_GATEWAY)
127              dst = (struct sockaddr_in *) ro->ro_rt->rt_gateway;
128      }
```

[redacted]

ip_output.c

图8-24 (续)

1. 验证高速缓存中的路由

77-99 ip_output可能把一条在高速缓存中的路由作为ro参数来提供。在第24章中，我们将看到UDP和TCP维护一个与各插口相关的路由缓存。如果没有路由，则ip_output把ro设置成指向临时route结构iproute。

如果高速缓存中的目的地不是去当前分组的目的地，就把该路由丢掉，新的目的地址放在dst中。

2. 旁路路由选择

100-114 调用方可通过设置IP_ROUTETOI标志(见8.8节)禁止对分组进行路由选择。ip_output必须找到一个与分组中指定目的地网络直接相连的接口。ifa_ifwithdstaddr搜索点到点接口，而in_ifwithnet搜索其他接口。如果任一函数找到与目的网络相连的接口，就返回ENETUNREACH；否则，ifp指向选定的接口。

这个选项允许路由选择协议绕过本地路由表，并使分组通过某特定接口退出系

统。通过这个方法，即使本地路由表不正确，也可以与其他路由器交换路由选择信息。

3. 本地路由

115-122 如果分组正被路由选择(IP_ROUTETOIF为关状态)，并且没有其他缓存的路由，则rtalloc找到一条到dst指定地址的路由。如果rtalloc没找到路由，则ip_ output返回EHOSTUNREACH。如果ip_forward调用ip_output，就把EHOSTUNREACH转换成ICMP差错。如果某个传输协议调用ip_output，就把差错传回给进程(图8-21)。

123-128 ia被设成指向选定接口的地址(ifaddr结构)，而ifp指向接口的ifnet结构。如果下一跳不是分组的最终目的地，则把 dst改成下一跳路由器地址，而不再是分组最终目的地址。IP首部内的目的地址不变，但接口层必须把分组提交给 dst，即下一跳路由器。

8.6.3 源地址选择和分片

ip_output的最后一部分如图 8-25所示，保证IP首部有一个有效源地址，然后把分组提交给与路由相关的接口。如果分组比接口的 MTU大，就必须对分组分片，然后一片一片地发送。像前面的重装代码一样，我们省略了分片代码，并推迟到第 10章再讨论。

ip_output.c

```
212     /*
213      * If source address not specified yet, use address
214      * of outgoing interface.
215      */
216     if (ip->ip_src.s_addr == INADDR_ANY)
217         ip->ip_src = IA_SIN(ia)->sin_addr;
218     /*
219      * Look for broadcast address and
220      * verify user is allowed to send
221      * such a packet.
222      */
223     if (in_broadcast(dst->sin_addr, ifp)) {
224         if ((ifp->if_flags & IFF_BROADCAST) == 0) {      /* interface check */
225             error = EADDRNOTAVAIL;
226             goto bad;
227         }
228         if ((flags & IP_ALLOWBROADCAST) == 0) {      /* application check */
229             error = EACCES;
230             goto bad;
231         }
232         /* don't allow broadcast messages to be fragmented */
233         if ((u_short) ip->ip_len > ifp->if_mtu) {
234             error = EMSGSIZE;
235             goto bad;
236         }
237         m->m_flags |= M_BCAST;
238     } else
239         m->m_flags &= ~M_BCAST;

240 sendit:
241     /*
242      * If small enough for interface, can just send directly.
243      */
244     if ((u_short) ip->ip_len <= ifp->if_mtu) {
```

图8-25 ip_output(续)

```
245         ip->ip_len = htons((u_short) ip->ip_len);
246         ip->ip_off = htons((u_short) ip->ip_off);
247         ip->ip_sum = 0;
248         ip->ip_sum = in_cksum(m, hlen);
249         error = (*ifp->if_output) (ifp, m,
250                                    (struct sockaddr *) dst, ro->ro_rt);
251         goto done;
252     }
```

[redacted]

```
339   done:
340     if (ro == &iproute && (flags & IP_ROUTETOIF) == 0 && ro->ro_rt)
341         RTFREE(ro->ro_rt);
342     return (error);
343   bad:
344     m_freem(m0);
345     goto done;
346 }
```

ip_output.c

图8-25 (续)

1. 选择源地址

212-239 如果没有指定ip_src，则ip_output选择输出接口的IP地址ia作为源地址。这不能在早期填充其他IP首部字段时做，因为那时还没有选定路由。转发的分组通常都有一个源地址，但是，如果发送进程没有明确指定源地址，产生于本地主机的分组可能没有源地址。

如果目的IP地址是一个广播地址，则接口必须支持广播 (IFF_BROADCAST，图3-7)，调用方必须明确使能广播(IP_ALLOWBROADCAST，图8-23)，而分组必须足够小，无需分片。

> 最后的测试是一个策略决定。IP协议规范中没有明确禁止对广播分组的分片。但是，要求分组适合接口的MTU，就增加了广播分组被每个接口接收的机会，因为接收一个未损坏的分组的机会要远大于接收两个或多个未损坏分组的机会。

如果这些条件都不满足，就扔掉该分组，把EADDRNOTAVAIL、EACCES和EMSGSIZE返回给调用方。否则，设置输出分组的M_BCAST，告诉接口输出函数把该分组作为链路级广播发送。21.20节中，我们将看到arpresolve把IP广播地址翻译成以太网广播地址。

如果目的地址不是广播地址，则ip_output把M_BCAST清零。

> 如果M_BCAST没有清零，则对一个作为广播到达的请求分组的应答将可能作为一个广播被返回。我们将在第11章中看到，ICMP应答将以这种方式作为TCP RST分组(见26.9节)在请求分组内构造。

2. 发送分组

240-252 如果分组对所选择的接口足够小，ip_len和ip_off被转换成网络字节序，IP检验和与in_cksum(见8.7节)一起计算，把分组提交给所选接口的if_output函数。

3. 分片分组

253-338 大分组在被发送之前必须分片。这里我们省略这段代码，推迟到第10章讨论。

4. 清零

339-346 对每一路由入口都有一个引用计数。我们提到过，如果参数ro为空，ip_

output可能会使用一个临时的route结构(iproute)。如果需要，RTFREE发布iproute内的路由入口，并把引用计数减1。Bad处的代码在返回前扔掉当前分组。

引用计数是一个存储器管理技术。程序员必须对一个数据结构的外部引用计数；当计数返回为0时，就可以安全地把存储器返回给空存储器池。引用计数要求程序员遵守一些规定，在恰当的时机增加或减小引用计数。

8.7 Internet检验和：in_cksum函数

有两个操作占据了处理分组的主要时间：复制数据和计算检验和 ([Kay和Pasquale 1993])。mbuf数据结构的灵活性是Net/3中减少复制操作的主要方法。由于对硬件的依赖，所以检验和的有效计算相对较难。Net/3中有几种in_cksum的实现(图8-26)。

版　本	源 文 件
portable C	sys/netinet/in_cksum.c
SPARC	net3/sparc/sparc/in_cksum.c
68k	net3/luna68k/luna68k/in_cksum.c
VAX	sys/vax/vax/in_cksum.c
Tahoe	sys/tahoe/tahoe/in_cksum.c
HP 3000	sys/hp300/hp300/in_cksum.c
Intel 80386	sys/i386/i386/in_cksum.c

图8-26 在Net/3中的几个in_cksum 版本

即使是可移植C实现也已经被相当好地优化了。RFC 1071 [Braden、Borman和Partridge 1988] 和RFC 1141 [Mallory和Kullberg 1990]讨论了Internet检验和函数的设计和实现。RFC 1141被RFC 1624 [Rijsinghani 1994] 修正。从RFC 1071：

1) 把被检验的相邻字节成对配成16 bit整数，就形成了这些整数的二进制反码的和。

2) 为生成检验和，把检验和字段本身清零，把16 bit的二进制反码的和以及这个和的二进制反码放到检验和字段。

3) 为检验检验和，对同一组字节计算它们的二进制反码的和。如果结果为全 1(在二进制反码运算中- 0，见下面的解释)，则检验成功。

简而言之，当对用二进制反码表示的整数进行加法运算时，把两个整数相加后再加上进位就得到加法的结果。在二进制反码运算中，只要把每一位求补就得到一个数的反。所以在二进制反码运算中，0有两种表示方法：全0，和全1。有关二进制反码的运算和表示的详细讨论见 [Mano 1982]。

检验和算法在发送分组之前计算出要放在 IP首部检验和字段的值。为了计算这个值，先把首部的检验和字段设为0，然后计算整个首部(包括选项)的二进制反码的和。把首部作为一个16 bit整数数组来处理。让我们把这个计算结果称为 a。因为检验和字段被明确设为0，所以 a是除了检验和字段外所有IP首部字段的和。a的二进制反码，用- a表示，被放在检验和字段中，发送该分组。

如果在传输过程中没有比特位被改变，则在目的地计算的检验和应该等于 (a+ - a)的二进制反码。在二进制反码运算中 (a+ - a)的和是- 0(全1)，而它的二进制反码应该等于0(全0)。所以在目的地，一个没有损坏分组计算出来的检验和应该总是为 0。这就是我们在图8-12中看到

的。下面的C代码(不是Net/3的内容)是这个算法的一种原始的实现：

```
 1 unsigned short
 2 cksum(struct ip *ip, int len)
 3 {
 4     long    sum = 0;            /* assume 32 bit long, 16 bit short */

 5     while (len > 1) {
 6         sum += *((unsigned short *) ip)++;
 7         if (sum & 0x80000000)   /* if high-order bit set, fold */
 8             sum = (sum & 0xFFFF) + (sum >> 16);
 9         len -= 2;
10     }

11     if (len)                    /* take care of left over byte */
12         sum += (unsigned short) *(unsigned char *) ip;

13     while (sum >> 16)
14         sum = (sum & 0xFFFF) + (sum >> 16);

15     return ~sum;
16 }
```

图8-27 IP检验和计算的一种原始的实现

1-16 这里唯一提高性能之处在于累计sum高16 bit的进位。当循环结束时，累计的进位被加在低16 bit上，直到没有其他进位发生。RFC 1071称此为延迟进位(deferred carries)。在没有有进位加法指令或检测进位代价很大的机器上，这个技术非常有效。

现在我们显示Net/3的可移植C版本。它使用了延迟进位技术，作用于存储在一个mbuf链中的分组。

42-140 我们的新检验和实现假定所有被检验字节存储在一个连续缓存而不是mbuf中。这个版本的检验和计算采用相同的底层算法来正确地处理mbuf：用32bit整数的延迟进位对16 bit字作加法。对奇数个字节的mbuf，多出来的一个字节被保存起来，并与下一个mbuf的第一个字节配对。因为在大多数体系结构中，对16 bit字的不对齐访问是无效的，甚至会产生严重差错，所以不对齐字节将被保存，in_cksum继续加上下一个对齐的字。当这种情况发生时，in_cksum总是很小心地交换字节，保证位于奇数和偶数位置的字节被放在单独的和字节中，以满足检验和算法的要求。

循环展开

93-115 函数中的三个while循环在每次迭代中分别在和中加上16个字、4个字和1个字。展开的循环减小了循环的耗费，在某些体系结构中可能比一个直接循环要快得多。但代价是代码长度和复杂性增大。

in_cksum.c

```
42 #define ADDCARRY(x)  (x > 65535 ? x -= 65535 : x)
43 #define REDUCE {l_util.l = sum; sum = l_util.s[0] + l_util.s[1]; ADDCARRY(sum);}

44 int
45 in_cksum(m, len)
46 struct mbuf *m;
47 int     len;
48 {
49     u_short *w;
50     int     sum = 0;
```

图8-28 IP检验和计算的一个优化的可移植C程序

```
51      int     mlen = 0;
52      int     byte_swapped = 0;
53      union {
54          char    c[2];
55          u_short s;
56      } s_util;
57      union {
58          u_short s[2];
59          long    l;
60      } l_util;
61      for (; m && len; m = m->m_next) {
62          if (m->m_len == 0)
63              continue;
64          w = mtod(m, u_short *);
65          if (mlen == -1) {
66              /*
67               * The first byte of this mbuf is the continuation of a
68               * word spanning between this mbuf and the last mbuf.
69               *
70               * s_util.c[0] is already saved when scanning previous mbuf.
71               */
72              s_util.c[1] = *(char *) w;
73              sum += s_util.s;
74              w = (u_short *) ((char *) w + 1);
75              mlen = m->m_len - 1;
76              len--;
77          } else
78              mlen = m->m_len;
79          if (len < mlen)
80              mlen = len;
81          len -= mlen;
82          /*
83           * Force to even boundary.
84           */
85          if ((1 & (int) w) && (mlen > 0)) {
86              REDUCE;
87              sum <<= 8;
88              s_util.c[0] = *(u_char *) w;
89              w = (u_short *) ((char *) w + 1);
90              mlen--;
91              byte_swapped = 1;
92          }
93          /*
94           * Unroll the loop to make overhead from
95           * branches &c small.
96           */
97          while ((mlen -= 32) >= 0) {
98              sum += w[0]; sum += w[1]; sum += w[2]; sum += w[3];
99              sum += w[4]; sum += w[5]; sum += w[6]; sum += w[7];
100             sum += w[8]; sum += w[9]; sum += w[10]; sum += w[11];
101             sum += w[12]; sum += w[13]; sum += w[14]; sum += w[15];
102             w += 16;
103         }
104         mlen += 32;
105         while ((mlen -= 8) >= 0) {
106             sum += w[0]; sum += w[1]; sum += w[2]; sum += w[3];
107             w += 4;
```

图8-28 (续)

```
108         }
109         mlen += 8;
110         if (mlen == 0 && byte_swapped == 0)
111             continue;
112         REDUCE;
113         while ((mlen -= 2) >= 0) {
114             sum += *w++;
115         }
116         if (byte_swapped) {
117             REDUCE;
118             sum <<= 8;
119             byte_swapped = 0;
120             if (mlen == -1) {
121                 s_util.c[1] = *(char *) w;
122                 sum += s_util.s;
123                 mlen = 0;
124             } else
125                 mlen = -1;
126         } else if (mlen == -1)
127             s_util.c[0] = *(char *) w;
128     }
129     if (len)
130         printf("cksum: out of data\n");
131     if (mlen == -1) {
132         /* The last mbuf has odd # of bytes. Follow the standard (the odd
133            byte may be shifted left by 8 bits or not as determined by
134            endian-ness of the machine) */
135         s_util.c[1] = 0;
136         sum += s_util.s;
137     }
138     REDUCE;
139     return (~sum & 0xffff);
140 }
```

in_cksum.c

图8-28 (续)

其他优化

RFC 1071提到两个在Net/3中没有出现的优化：联合的有检验和的复制操作和递增的检验和更新。对IP首部检验和来说，把复制和检验和操作结合起来并不像对 TCP和UDP那么重要，因为后者覆盖了更多的字节。在 23.12节中对这个合并的操作进行了讨论。 [Partridge和Pink 1993]报告了IP首部检验和的一个内联版本比调用更一般的 `in_cksum`函数要快得多，只需6~8个汇编指令就可以完成(标准的20字节IP首部)。

检验和算法设计允许改变分组，并在不重新检查所有字节的情况下更新检验和。 RFC 1071对该问题进行简明的讨论。 RFC 1141和1624中有更详细的讨论。该技术的一个典型应用是在分组转发的过程中。通常情况下，当分组没有选项时，转发过程中只有 TTL字段发生变化。在这种情况下，可以只用一次循环进位，重新计算检验和。

为了进一步提高效率，递增的检验和也有助于检测到被有差错的软件破坏的首部。如果递增地计算检验和，则下一个系统可以检测到被破坏的首部。但是如果不是递增计算检验和，那么检验和中就包含了差错的字节，检测不到有问题的首部。 UDP和TCP使用的检验和算法在最终目的主机检测到该差错。我们将在第 23和25章看到UDP和TCP检验和包含了IP首部的几个部分。

使用硬件有进位加法指令一次性计算32 bit检验和的检验和函数，可参见 ./sys/vax/vax/in_cksum.c文件中VAX实现的in_cksum。

8.8 setsockopt和getsockopt系统调用

Net/3提供setsockopt和getsockopt两个系统调用来访问一些网络互连的性质。这两个系统调用支持一个动态接口，进程可用该动态接口来访问某种网络互连协议的一些性质，而标准系统调用通常不支持该协议。这两个调用的原型是：

```
int setsockopt(int s, int level, int optname, void *optval, int optlen);
int getsockopt(int s, int level, int optname, const void *optval, int optlen);
```

大多数插口选项只影响它们在其上发布的插口。与sysctl参数相比，后者影响整个系统。与多播相关的插口选项是一个明显的例外，将在第12章中讨论。

setsockopt和getsockopt设置和获取通信栈所有层上的选项。Net/3按照与*s*相关的协议和由*level*指定的标识符处理选项。图8-29列出了在我们讨论的协议中*level*可能取得的值。

在第17章中，我们描述了setsockopt和getsockopt的实现，但在其他适当章节中讨论有关选项的实现。本章讨论访问IP性质的选项。

字段	协议	*level*	函　数	参　考
任意	任意	*SOL_SOCKET*	sosetopt和sogetopt	图17-5和图17-11
IP	UDP	*IPPROTO_IP*	ip_ctloutput	图8-31
	TCP	*IPPROTO_TCP* *IPPROTO_IP*	tcp_ctloutput ip_ctloutput	30.6节 图8-31
	原始IP ICMP IGMP	*IPPROTO_IP*	rip_ctloutput和 ip_ctloutput	32.8节

图8-29 sosetopt和sogetopt参数

我们把本书中出现的所有插口选项总结在图8-30中。该图显示了IPPROTO_IP级的选项。选项出现在第1列，*optval*指向变量的数据类型出现在第2列，第3列显示的是处理该选项的函数。

选　项　名	Optval类型	函　数	描　述
IP_OPTIONS	void*	in_pcbopts	设置或获取发出的数据报中的IP选项
IP_TOS	int	ip_ctloutput	设置或获取发出的数据报中的IP TOS
IP_TTL	int	ip_ctloutput	设置或获取发出的数据报中的IP TTL
TP_RECVDSTADDR	int	ip_ctloutput	使能或禁止IP目的地址(只有UDP)的排队
IP_RECVOPTS	int	ip_ctloutput	使能或禁止对到达IP选项作为控制信息的排队(只对UDP；还没有实现)
IP_RECVRETOPTS	int	ip_ctloutput	使能或禁止与到达数据报相关的逆源路由(只对UDP；还没有实现)

图8-30 插口选项：SOCK_RAW、SOCK_DGRAM和SOCK_STREAMR插口的IPPROTO_IP级

图8-31显示了用于处理大部分IPPROTO_IP选项的ip_ctloutput函数的整个结构。在32.8节中我们给出与SOCK_RAW插口一起使用的IPPROTO_IP选项。

ip_output.c

```
431 int
432 ip_ctloutput(op, so, level, optname, mp)
433 int     op;
434 struct socket *so;
435 int     level, optname;
436 struct mbuf **mp;
437 {
438     struct inpcb *inp = sotoinpcb(so);
439     struct mbuf *m = *mp;
440     int     optval;
441     int     error = 0;

442     if (level != IPPROTO_IP) {
443         error = EINVAL;
444         if (op == PRCO_SETOPT && *mp)
445             (void) m_free(*mp);
446     } else
447         switch (op) {

448         case PRCO_SETOPT:
449             switch (optname) {

                    /* PRCO_SETOPT processing (Figures 8.32 and 12.17) */

493               freeit:
494             default:
495                 error = EINVAL;
496                 break;
497             }
498             if (m)
499                 (void) m_free(m);
500             break;

501         case PRCO_GETOPT:
502             switch (optname) {

                    /* PRCO_SETOPT processing (Figures 8.33 and 12.17) */

546             default:
547                 error = ENOPROTOOPT;
548                 break;
549             }
550             break;
551         }
552     return (error);
553 }
```

ip_output.c

图8-31 ip_ctloutput 函数：概貌

431-447 ip_ctloutput的第一个参数op，可以是PRCO_SETOPT或者PRCO_GETOPT。第二个参数so，指向向其发布请求的插口。level必须是IPPROTO_IP。Optname是要改变或要检索的选项，mp间接地指向一个含有与该选项相关数据的 mbuf，m被初始化为指向由 *mp引用的mbuf。

448-500 如果在调用setsockopt时指定了一个无法识别的选项(因此，在switch中调用PRCO_SETOPT语句)，ip_ctloutput释放掉所有调用方传来的缓存，并返回 EINVAL。

501-553 getsockopt传来的无法识别的选项导致ip_ctloutput返回ENOPROTOOPT。在这种情况下，调用方释放mbuf。

8.8.1 PRCO_SETOPT的处理

对PRCO_SETOPT的处理如图8-32所示。

ip_output.c

```
450             case IP_OPTIONS:
451                 return (ip_pcbopts(&inp->inp_options, m));

452             case IP_TOS:
453             case IP_TTL:
454             case IP_RECVOPTS:
455             case IP_RECVRETOPTS:
456             case IP_RECVDSTADDR:
457                 if (m->m_len != sizeof(int))
458                         error = EINVAL;
459                 else {
460                     optval = *mtod(m, int *);
461                     switch (optname) {

462                     case IP_TOS:
463                         inp->inp_ip.ip_tos = optval;
464                         break;

465                     case IP_TTL:
466                         inp->inp_ip.ip_ttl = optval;
467                         break;
468 #define OPTSET(bit) \
469     if (optval) \
470         inp->inp_flags |= bit; \
471     else \
472         inp->inp_flags &= ~bit;

473                     case IP_RECVOPTS:
474                         OPTSET(INP_RECVOPTS);
475                         break;

476                     case IP_RECVRETOPTS:
477                         OPTSET(INP_RECVRETOPTS);
478                         break;

479                     case IP_RECVDSTADDR:
480                         OPTSET(INP_RECVDSTADDR);
481                         break;
482                     }
483                 }
484                 break;
```

ip_output.c

图8-32 ip_ctloutput 函数：处理PRCO_SETOPT

450-451 IP_OPTIONS是由ip_pcbopts处理的(图9-32)。

452-484 IP_TOS、IP_TTL、IP_RECVOPTS、IP_RECVERTOPTS以及IP_RECVDSTADDR选项都需要在由m指向的mbuf中有一个整数。该整数储存在optval中，用来改变与插口有关的ip_tos和ip_ttl的值，或者用来设置或复位与插口相关的INP_RECVOPTS、INP_RECVERTOPTS和INP_RECVDSTADDR标志位。如果optval是非零(或0)，则宏OPTSET设置(或复位)指定的比特。

图8-30中显示没有实现IP_RECVOPTS和IP_RECVERTOPTS。在第23章中，我

们将看到UDP忽略了这些选项的设置。

8.8.2 PRCO_GETOPT的处理

图8-33显示的一段代码完成了当指定PRCO_GETOPT时对IP选项的检索。

ip_output.c

```
503             case IP_OPTIONS:
504                 *mp = m = m_get(M_WAIT, MT_SOOPTS);
505                 if (inp->inp_options) {
506                     m->m_len = inp->inp_options->m_len;
507                     bcopy(mtod(inp->inp_options, caddr_t),
508                           mtod(m, caddr_t), (unsigned) m->m_len);
509                 } else
510                     m->m_len = 0;
511                 break;
512             case IP_TOS:
513             case IP_TTL:
514             case IP_RECVOPTS:
515             case IP_RECVRETOPTS:
516             case IP_RECVDSTADDR:
517                 *mp = m = m_get(M_WAIT, MT_SOOPTS);
518                 m->m_len = sizeof(int);
519                 switch (optname) {
520                 case IP_TOS:
521                     optval = inp->inp_ip.ip_tos;
522                     break;
523                 case IP_TTL:
524                     optval = inp->inp_ip.ip_ttl;
525                     break;
526 #define OPTBIT(bit) (inp->inp_flags & bit ? 1 : 0)
527                 case IP_RECVOPTS:
528                     optval = OPTBIT(INP_RECVOPTS);
529                     break;
530                 case IP_RECVRETOPTS:
531                     optval = OPTBIT(INP_RECVRETOPTS);
532                     break;
533                 case IP_RECVDSTADDR:
534                     optval = OPTBIT(INP_RECVDSTADDR);
535                     break;
536                 }
537                 *mtod(m, int *) = optval;
538                 break;
```

ip_output.c

图8-33 ip_ctloutput 函数：PRCO_GETOPT 的处理

503-538 对IP_OPTIONS，ip_ctloutput返回一个缓存，该缓存中包含了与该插口相关的选项的备份。对其他选项，ip_ctloutput返回ip_tos和ip_ttl的值，或与该选项相关的标志的状态。返回的值放在由 *m*指向的mbuf中。如果在inp_flags中的bit是打开(或关闭)的，则宏OPTBIT将返回1(或0)。

8.9 ip_sysctl函数

图7-27显示，在调用sysctl中，当协议和协议族的标识符是0时，就调用ip_sysctl函

数。图8-34显示了ip_sysctl支持的三个函数。

sysctl常量	Net/3变量	描 述
IPCTL_FORWARDING	ipforwarding	系统是否转发IP分组？
IPCTL_SENDREDIRECTS	ipsendredirects	系统是否发ICMP重定向？
IPCTL_DEFTTL	ip_defttl	IP分组的默认TTL

图8-34 sysctl 参数

图8-35显示了ip_sysctl函数。

ip_input.c

```
984 int
985 ip_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
986 int    *name;
987 u_int   namelen;
988 void   *oldp;
989 size_t *oldlenp;
990 void   *newp;
991 size_t  newlen;
992 {
993     /* All sysctl names at this level are terminal. */
994     if (namelen != 1)
995         return (ENOTDIR);

996     switch (name[0]) {
997     case IPCTL_FORWARDING:
998         return (sysctl_int(oldp, oldlenp, newp, newlen, &ipforwarding));
999     case IPCTL_SENDREDIRECTS:
1000        return (sysctl_int(oldp, oldlenp, newp, newlen,
1001                           &ipsendredirects));
1002    case IPCTL_DEFTTL:
1003        return (sysctl_int(oldp, oldlenp, newp, newlen, &ip_defttl));
1004    default:
1005        return (EOPNOTSUPP);
1006    }
1007    /* NOTREACHED */
1008 }
```

ip_input.c

图8-35 ip_sysctl 函数

因为ip_sysctl并不把sysctl请求转发给其他函数，所以在name中只能有一个成员。否则返回ENOTDIR。

Switch语句选择恰当的调用 systl_int，它访问或修改 ipforwarding、ipsendredirects或ip_defttl。对无法识别的选项返回EOPNOTSUPP。

8.10 小结

IP是一个最佳的数据报服务，它为所有其他 Internet协议提供交付机制。标准IP首部长度为20字节，但可跟最多40字节的选项。IP可以把大的数据报分片发送，并在目的地重装分片。对选项处理的讨论放在第9章和第10章讨论分片和重装。

ipintr保证IP首部到达时未经破坏，通过把目的地址与系统接口地址及其他几个广播地址比较来确定它们是否到达最终目的地。 ipintr把到达最终目的地的数据报传给分组内指定的运输层协议。如果系统被配置成路由器，就把还没有到达最终目的地的分组发给

ip_forward转发到最终目的地。分组有一个受限的生命期。如果 TTL字段变成0，则ip_forward就丢掉该分组。

许多Internet协议都使用Internet检验和函数，Net/3用in_cksum实现。IP检验和只覆盖首部(和选项)，不覆盖数据，数据必须由传输协议级的检验和保护。作为IP中最耗时的操作，检验和函数通常要对不同的平台进行优化。

习题

8.1 当没有为任何接口分配IP地址时，IP是否该接收广播分组？

8.2 修改ip_forward和ip_output，当转发一个没有选项的分组时，对IP检验和进行递增的更新。

8.3 当拒绝转发分组时，为什么需要检测链路级广播(某缓存中的M_BCAST标志)和IP级广播(in_canforward)？在何种情况下，把一个具有IP单播目的地的分组作为一个链路层广播接收？

8.4 当一个IP分组到达时有检验和差错，为什么不向发送方返回一个差错信息？

8.5 假定一个多接口主机上的某个进程为它发出的分组选择了一个明确的源地址。而且，假定是通过一个接口而不是作为分组源地址所选择的地址到达的。当第一跳路由器发现分组应该到另一个路由器时，会发生什么情况？会向主机发送重定向报文吗？

8.6 一个新的主机被连到一个已划分子网的网络中，并被配置成完成路由选择的功能(ipforwarding等于1)，但它的网络接口没有分配子网掩码。当该主机接收一个子网广播分组时会出现什么情况？

8.7 图8-17中，在检测ip_ttl后(与之前相比)，为什么需要把它减1？

8.8 如果两个路由器都认为对方是分组的最佳下一跳目的地，将发生什么情况？

8.9 图8-14中，对一个到达SLIP接口的分组，不检测哪些地址？有没有其他在图8-14中没有列出的地址被检测？

8.10 ip_forward在调用icmp_error之前，把分片的id从主机字节序转换成网络字节序。为什么它不对分片的偏移进行转换？

第9章　IP选项处理

9.1　引言

第8章中提到，IP输入函数(ipintr)将在验证分组格式(检验和，长度等)之后，确定分组是否到达目的地之前，对选项进行处理。这表明，分组所遇到的每个路由器以及最终的目的主机都要对分组的选项进行处理。

RFC 791和1122指定了IP选项和处理规则。本章将讨论大多数IP选项的格式和处理。我们也将显示运输协议如何指定IP数据报内的IP选项。

IP分组内可以包含某些在分组被转发或被接收之前处理的可选字段。 IP实现可以用任意顺序处理选项；Net/3按照选项在分组中出现的顺序处理选项。图 9-1显示，标准IP首部之后最多可跟40字节的选项。

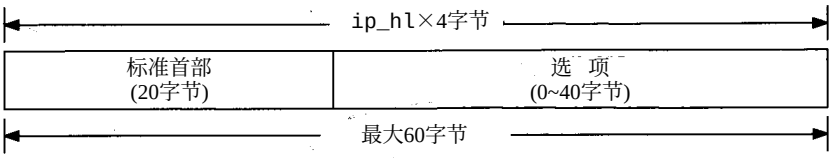


图9-1　一个IP首部可以有0~40字节的IP选项

9.2　代码介绍

两个首部描述了IP选项的数据结构。选项处理的代码出现在两个 C文件中。图9-2列出了相关的文件。

文　件	描　述
netinet/ip.h	ip_timestamp结构
netinet/ip_var.h	ipoption结构
netinet/ip_input.c	选项处理
netinet/ip_output.c	ip_insertoptions函数

图9-2　本章讨论的文件

9.2.1　全局变量

图9-3描述了两个全局变量支持源路由的逆 (reversal)。

变　量	数据类型	描　述
ip_nhops	int	以前的源路由跳计数
ip_srcrt	struct ip_srcrt	以前的源路由

图9-3　本章引入的全局变量

9.2.2 统计量

选项处理代码更新的唯一的统计量是ipstat结构中的ips_badoptions，如图8-4所示。

9.3 选项格式

IP选项字段可能包含0个或多个单独选项。选项有两种类型，单字节和多字节，如图 9-4中所示。

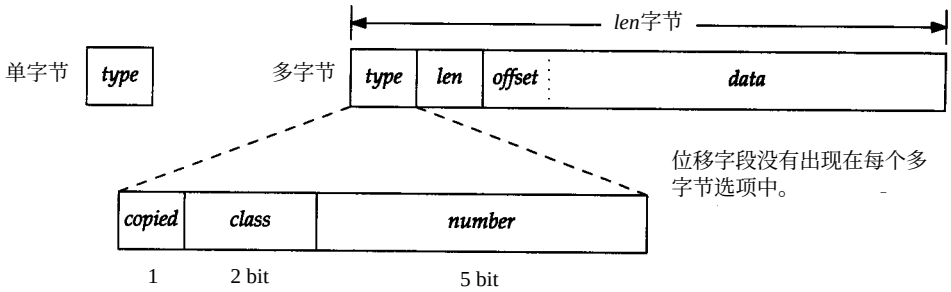


图9-4 单字节和多字节IP选项的结构

所有选项都以1字节类型(*type*)字段开始。在多字节选项中，类型字段后面紧接着一个长度(*len*)字段，其他的字节是数据 (*data*)。许多选项数据字段的第一个字节是 1字节的位移(*offset*)字段，指向数据字段内的某个字节。长度字节的计算覆盖了类型、长度和数据字段。类型被继续分成三个子字段： 1 bit备份(*copied*)标志、 2 bit类(*class*)字段和 5 bit数字(*number*)字段。图9-5列出了目前定义的 IP选项。前两个选项是单字节选项；其他的是多字节选项。

常 量	类 型		长度	Net/3	描 述
	十进制	二进制	(字节)		
IPOPT_EOL	0-0-0 0	0-00-00000	1	•	选项表的结尾(EOL)
IPOPT_NOP	0-0-1 1	0-00-00001	1	•	无操作(NOP)
IPOPT_RR	0-0-7 7	0-00-00111	可变	•	记录路由
IPOPT_TS	0-2-4 68	0-10-00100	可变	•	时戳
IPOPT_SECURITY	1-0-2 130	1-00-00010	11		基本的安全
IPOPT_LSRR	1-0-3 131	1-00-00011	可变	•	宽松源路由和记录路由(LSRR)
	1-0-5 133	1-00-00101	可变		扩展的安全
IPOPT_SATID	1-0-8 136	1-00-01000	4		流标识符
IPOPT_SSRR	1-0-9 137	1-00-01001	可变	•	严格源路由和记录路由(SSRR)

图9-5 RFC 791定义的IP选项

第1列显示了Net/3的选项常量，第2列和第3列是该类型的十进制和二进制值，第4列是选项的长度。Net/3列显示的是在Net/3中由ip_dooptions实现的选项。IP必须自动忽略所有它不识别的选项。我们不描述 Net/3没有实现的选项：安全和流 ID。流ID选项是过时的，安全选项主要只由美国军方使用。 RFC 791中有更多的讨论。

class	描 述
0	控制
1	保留
2	查错和措施
3	保留

图9-6 IP选项内的*class*字段

当Net/3对一个有选项的分组进行分片时 (10.4节)，它将检查*copied*标志位。该标志位指出是否把所有选项都备份到每个分片的 IP首部。*class*字段把相关的

选项按如图9-6所示进行分组。图9-5中，除时戳选项具有*class*为2外，所有选项都是*class*为0。

9.4 ip_dooptions函数

在图8-13中，我们看到ipintr在检测分组的目的地址之前调用ip_dooptions。ip_dooptions被传给一个指针m，该指针指向某个分组，ip_dooptions处理分组中它所知道的选项。如果ip_dooptions转发该分组，如在处理LSRR和SSRR选项时，或由于某个差错而丢掉该分组时，它返回1。如果它不转发分组，ip_dooptions返回0，由ipintr继续处理该分组。

ip_dooptions是一个长函数，所以我们分步地显示。第一部分初始化一个for循环，处理首部中的各选项。

当处理每个选项时，cp指向选项的第一个字节。图9-7显示，当可用时，如何从cp的常量位移访问*type*、*length*和*offset*字段。

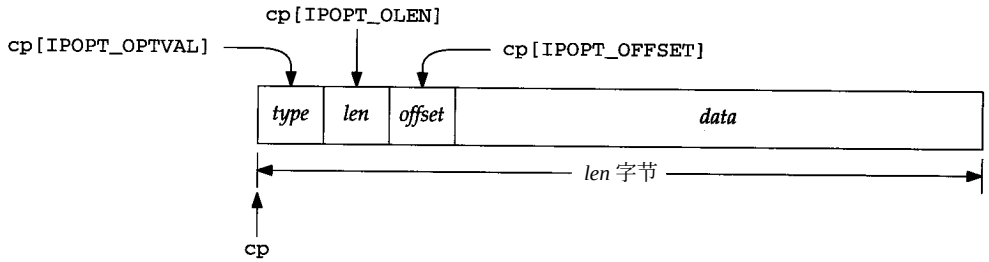


图9-7 用常量位移访问IP选项字段

RFC把位移(*offset*)字段描述作指针(*pointer*)，指针比位移的描述性略强一些。*offset*的值是某个字节在该选项内的序号(从*type*开始，序号为1)，不是从*type*开始的、且以零开始的计数。位移的最小值是4(IPOPT_MINOFF),它指向的是多字节选项中数据字段的第一个字节。

图9-8显示了ip_dooptions函数的整体结构。

555-566 ip_dooptions把ICMP差错类型type初始化为ICMP_PARAMPROB，对任何没有特定差错类型的差错，这是一个一般值。对于ICMP_PARAMPROB，code指的是出错字节在分组内的位移。这是默认的ICMP差错报文。某些选项将改变这些值。

> ip指向一个20字节大小的ip结构，所以ip+1指向的是跟在IP首部后面的下一个ip结构。因为ip_dooptions需要IP首部后面字节的地址，所以就把结果指针转换成为指向一个无符号字节(u_char)的指针。因此，cp指向标准IP首部以外的第一个字节，就是IP选项的第一个字节。

1. EOL和NOP过程

567-582 for循环按照每个选项在分组中出现的顺序分别对它们进行处理。EOL选项以及一个无效的选项长度(也即选项长度表明选项数据超过了IP首部)都将终止该循环。当出现NOP选项时，忽略它。switch语句的default情况隐含要求系统忽略未知的选项。

下面的内容描述了switch语句处理的每个选项。如果ip_dooptions在处理分组选项时没有出错，就把控制交给switch下面的代码。

2. 源路由转发

719-724 如果分组需要被转发，SSRR或LSRR选项处理代码就把forward置位。分组被传给ip_forward，并且第2个参数为1，表明分组是按源路由选择的。

ip_input.c

```
553 int
554 ip_dooptions(m)
555 struct mbuf *m;
556 {
557     struct ip *ip = mtod(m, struct ip *);
558     u_char *cp;
559     struct ip_timestamp *ipt;
560     struct in_ifaddr *ia;
561     int     opt, optlen, cnt, off, code, type = ICMP_PARAMPROB, forward = 0;
562     struct in_addr *sin, dst;
563     n_time  ntime;

564     dst = ip->ip_dst;
565     cp = (u_char *) (ip + 1);
566     cnt = (ip->ip_hl << 2) - sizeof(struct ip);
567     for (; cnt > 0; cnt -= optlen, cp += optlen) {
568         opt = cp[IPOPT_OPTVAL];
569         if (opt == IPOPT_EOL)
570             break;
571         if (opt == IPOPT_NOP)
572             optlen = 1;
573         else {
574             optlen = cp[IPOPT_OLEN];
575             if (optlen <= 0 || optlen > cnt) {
576                 code = &cp[IPOPT_OLEN] - (u_char *) ip;
577                 goto bad;
578             }
579         }
580         switch (opt) {

581         default:
582             break;

                        /* option processing */

719     }
720     if (forward) {
721         ip_forward(m, 1);
722         return (1);
723     }
724     return (0);

725 bad:
726     ip->ip_len -= ip->ip_hl << 2;   /* XXX icmp_error adds in hdr length */
727     icmp_error(m, type, code, 0, 0);
728     ipstat.ips_badoptions++;
729     return (1);
730 }
```

ip_input.c

图9-8 ip_dooptions 函数

我们在8.5节中讲到，并不为源路由选择分组生成ICMP重定向——这就是为什么在传给ip_forward时设置第2个参数的原因。

如果转发了分组，则 ip_dooptions返回1。如果分组中没有源路由，则返回 0给ipintr，表明需要对该数据报进一步处理。注意，只有当系统被配置成路由器时(ipforwarding等于1)，才发生源路由转发。

从某种程度上说，这是一个有些矛盾的策略，但却是 RFC 1122的书面要求。RFC 1127 [Braden 1989c]把它作为一个公开问题加以阐述。

3. 差错处理

725-730 如果在switch语句里出现了错误，ip_dooptions就跳到bad。从分组长度中把IP首部长度减去，因为icmp_error假设首部长度不包含在分组长度里。icmp_error发出适当的差错报文，ip_dooptions返回1，避免ipintr处理被丢弃的分组。

下一节描述Net/3处理的所有选项。

9.5 记录路由选项

记录路由选项使得分组在穿过互联网时所经过的路由被记录在分组内部。项的大小是源主机在构造时确定的，必须足够保存所有预期的地址。我们记得在 IP分组的首部，选项最多只能有40字节。记录路由选项可以有 3个字节的开销，后面紧跟地址的列表 (每个地址4字节)。如果该选项是唯一的选项，则最多可以有 9个(3＋4×9＝39)地址出现。一旦分配给该选项的空间被填满，就按通常的情况对分组进行转发，中间的系统就不再记录地址。

图9-9说明了一个记录路由选项的格式，图 9-10是其源程序。

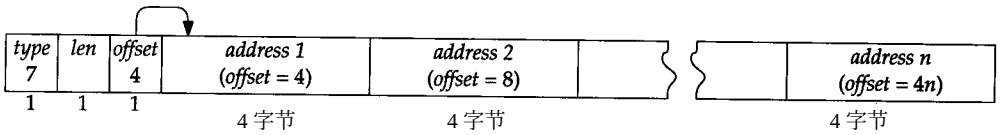


图9-9 记录路由选项，其中n必须≤9

ip_input.c

```
647             case IPOPT_RR:
648                 if ((off = cp[IPOPT_OFFSET]) < IPOPT_MINOFF) {
649                     code = &cp[IPOPT_OFFSET] - (u_char *) ip;
650                     goto bad;
651                 }
652                 /*
653                  * If no space remains, ignore.
654                  */
655                 off--;              /* 0 origin */
656                 if (off > optlen - sizeof(struct in_addr))
657                             break;
658                 bcopy((caddr_t) (&ip->ip_dst), (caddr_t) & ipaddr.sin_addr,
659                       sizeof(ipaddr.sin_addr));
660                 /*
661                  * locate outgoing interface; if we're the destination,
662                  * use the incoming interface (should be same).
663                  */
664                 if ((ia = (INA) ifa_ifwithaddr((SA) & ipaddr)) == 0 &&
665                     (ia = ip_rtaddr(ipaddr.sin_addr)) == 0) {
666                     type = ICMP_UNREACH;
667                     code = ICMP_UNREACH_HOST;
```

图9-10 函数ip_dooptions ：记录路由选项的处理

```
668                 goto bad;
669             }
670             bcopy((caddr_t) & (IA_SIN(ia)->sin_addr),
671                   (caddr_t) (cp + off), sizeof(struct in_addr));
672             cp[IPOPT_OFFSET] += sizeof(struct in_addr);
673             break;
```

ip_input.c

图9-10 (续)

647-657 如果位移选项太小，则ip_dooptions就发送一个ICMP参数问题差错。如果变量code被设置成分组内无效选项的字节位移量，并且 bad标号(图9-8)语句的执行产生错误，则发出的ICMP参数问题差错报文中就具有该 code值。如果选项中没有附加地址的空间，则忽略该选项，并继续处理下一个选项。

记录地址

658-673 如果ip_dst是某个系统地址(分组已到达目的地)，则把接收接口的地址记录在选项中；否则把ip_rtaddr提供的外出接口的地址记录下来。把位移更新为选项中下一个可用地址位置。如果ip_rtaddr无法找到到目的地的路由，就发送一个ICMP主机不可达差错报文。

卷1的7.3节举了一些记录路由选项的例子。

ip_rtaddr函数

函数ip_rtaddr查询路由缓存，必要时查询完整的路由表，来找到到给定 IP地址的路由。它返回一个指向in_ifaddr结构的指针，该指针与该路由的外出接口有关。图 9-11显示了该函数。

ip_input.c

```
735 struct in_ifaddr *
736 ip_rtaddr(dst)
737 struct in_addr dst;
738 {
739     struct sockaddr_in *sin;

740     sin = (struct sockaddr_in *) &ipforward_rt.ro_dst;

741     if (ipforward_rt.ro_rt == 0 || dst.s_addr != sin->sin_addr.s_addr) {
742         if (ipforward_rt.ro_rt) {
743             RTFREE(ipforward_rt.ro_rt);
744             ipforward_rt.ro_rt = 0;
745         }
746         sin->sin_family = AF_INET;
747         sin->sin_len = sizeof(*sin);
748         sin->sin_addr = dst;

749         rtalloc(&ipforward_rt);
750     }
751     if (ipforward_rt.ro_rt == 0)
752         return ((struct in_ifaddr *) 0);
753     return ((struct in_ifaddr *) ipforward_rt.ro_rt->rt_ifa);
754 }
```

ip_input.c

图9-11 函数ip_rtaddr ：寻找外出的接口

1. 检查IP转发缓存

735-741 如果路由缓存为空，或者如果 ip_rtaddr的唯一参数dest与路由缓存中的目的

地不匹配，则必须查询路由表选择一个外出的接口。

2. 确定路由

742-750 旧的路由 (如果有的话)被丢弃，并把新的路由储存在 *sin(这是转发缓存的ro_dst成员)。rtalloc搜索路由表，寻找到目的地的路由。

3. 返回路由信息

751-754 如果没有路由可用，就返回一个空指针；否则，就返回一个指针，指向与所选路由相关联的接口地址结构。

9.6 源站和记录路由选项

通常是在中间路由器所选择的路径上转发分组。源站和记录路由选项允许源站明确指定一条到目的地的路由，覆盖掉中间路由器的路由选择决定。而且，在分组到达目的地的过程中，把该路由记录下来。

严格路由包含了源站和目的站之间的每个中间路由器的地址；宽松路由只指定某些中间路由器的地址。在宽松路由中，路由器可以自由选择两个系统之间的任何路径；而在严格路由中，则不允许路由器这样做。我们用图 9-12说明源路由处理.

A、*B*和*C*是路由器，而HS和HD是源和目的主机。因为每个接口都有自己的 IP地址，所以我们看到路由器 A有三个地址： A_1，A_2和A_3。同样，路由器 *B*和*C*也有多个地址。图 9-13显示了源站和记录路由选项的格式。

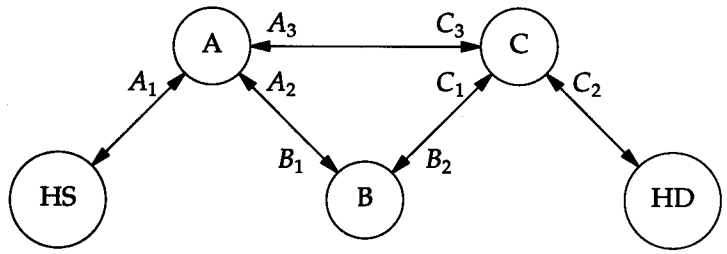


图9-12 源路由举例

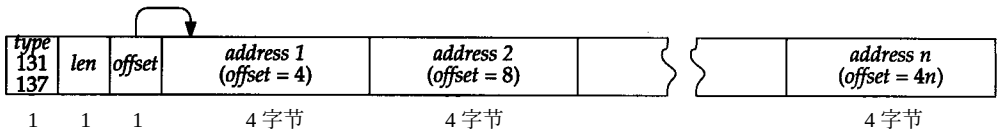


图9-13 严格和宽松源路由选项

IP首部的源和目的地址以及在选项中列出的位移和地址表，指定了路由以及分组目前在该路由中所处的位置。图 9-14显示，当分组按照这个宽松源路由从 HS经A、B、C到HD时，信息是如何改变的。每行代表当分组被第 1列显示的系统发送时的状态。最后一行显示分组被HD接收。图9-15显示了相关的代码。

符号“ • ”表示位移与路由中地址的相对位置。注意，每个系统都把出接口的地址放到选项去。特别地，原来的路由指定 A_3为第一跳目的地，但是外出接口 A_2被记录在路由中。通过这种方法，分组所采用的路由被记录在选项中。被记录的路由将被目的地系统倒转过来放到所有应答分组上，让它们沿着原始的路由的逆方向发送。

除了UDP，Net/3在应答时总是把收到的源路由逆转过来。

系统	IP首部		源路由选项	
	ip_src	ip_dst	位移	地址
HS	HS	A_3	4	• B_1 C_1 HD
A	HS	B_1	8	A_2 • C_1 HD
B	HS	C_1	12	A_2 B_2 • HD
C	HS	HD	16	A_2 B_2 C_2 •
HD	HS	HD	16	A_2 B_2 C_2 •

图9-14 当分组通过该路由时，源路由选项被修改。

ip_input.c

```
583             /*
584              * Source routing with record.
585              * Find interface with current destination address.
586              * If none on this machine then drop if strictly routed,
587              * or do nothing if loosely routed.
588              * Record interface address and bring up next address
589              * component.  If strictly routed make sure next
590              * address is on directly accessible net.
591              */
592         case IPOPT_LSRR:
593         case IPOPT_SSRR:
594             if ((off = cp[IPOPT_OFFSET]) < IPOPT_MINOFF) {
595                 code = &cp[IPOPT_OFFSET] - (u_char *) ip;
596                 goto bad;
597             }
598             ipaddr.sin_addr = ip->ip_dst;
599             ia = (struct in_ifaddr *)
600                 ifa_ifwithaddr((struct sockaddr *) &ipaddr);
601             if (ia == 0) {
602                 if (opt == IPOPT_SSRR) {
603                     type = ICMP_UNREACH;
604                     code = ICMP_UNREACH_SRCFAIL;
605                     goto bad;
606                 }
607                 /*
608                  * Loose routing, and not at next destination
609                  * yet; nothing to do except forward.
610                  */
611                 break;
612             }
613             off--;              /* 0 origin */
614             if (off > optlen - sizeof(struct in_addr)) {
615                 /*
616                  * End of source route.  Should be for us.
617                  */
618                 save_rte(cp, ip->ip_src);
619                 break;
620             }
621             /*
622              * locate outgoing interface
623              */
624             bcopy((caddr_t) (cp + off), (caddr_t) & ipaddr.sin_addr,
625                   sizeof(ipaddr.sin_addr));
626             if (opt == IPOPT_SSRR) {
627 #define INA struct in_ifaddr *
628 #define SA  struct sockaddr *
629                 if ((ia = (INA) ifa_ifwithdstaddr((SA) & ipaddr)) == 0)
```

图9-15 函数ip_dooptions ：LSRR和SSRR选项处理

```
630                     ia = (INA) ifa_ifwithnet((SA) & ipaddr);
631             } else
632                 ia = ip_rtaddr(ipaddr.sin_addr);
633             if (ia == 0) {
634                 type = ICMP_UNREACH;
635                 code = ICMP_UNREACH_SRCFAIL;
636                 goto bad;
637             }
638             ip->ip_dst = ipaddr.sin_addr;
639             bcopy((caddr_t) & (IA_SIN(ia)->sin_addr),
640                   (caddr_t) (cp + off), sizeof(struct in_addr));
641             cp[IPOPT_OFFSET] += sizeof(struct in_addr);
642             /*
643              * Let ip_intr's mcast routing check handle mcast pkts
644              */
645             forward = !IN_MULTICAST(ntohl(ip->ip_dst.s_addr));
646             break;
```

ip_input.c

图9-15 (续)

583-612 如果选项位移小于4(IPOPT_MINOFF)，则Net/3发送一个ICMP参数问题差错，并带上相应的code值。如果分组的目的地址与本地地址没有一个匹配，且选项是严格源路由(IPOPT_SSRR)，则发送一个源路由失败差错。如果本地地址不在路由中，则上一个系统把分组发送到错误的主机上了。对宽松路由(IPOPT_LSRR)来说，这不是错误；仅意味着IP必须把分组转发到目的地。

1. 源路由的结束

613-620 减小off，把它转换成从选项开始的字节位移。如果IP首部的ip_dst是某个本地地址，并且off所指向的超过了源路由的末尾，源路由中没有地址了，则分组已经到达了目的地。save_rte复制在静态结构ip_srcrt中的路由，并保存在全局ip_nhops(图9-18)里路由中的地址个数。

ip_srcrt被定义成为一个外部静态结构，因为它只能被在ip_input.c中定义的函数访问。

2. 为下一跳更新分组

621-637 如果ip_dst是一个本地地址，并且offset指向选项内的一个地址，则该系统是源路由中指定的一个中间系统，分组也没有到达目的地。在严格路由中，下一个系统必须位于某个直接相连的网络上。ifa_ifwithdst和ifa_ifwithnet通过在配置的接口中搜索匹配的目的地址(一个点到点的接口)或匹配的网络地址(广播接口)来寻找一条到下一个系统的路由。而在宽松路由中，ip_rtaddr(图9-11)通过查询路由表来寻找到下一个系统的路由。如果没有找到到下一系统的接口或路由，就发送一个ICMP源路由失败差错报文。

638-644 如果找到一个接口或一条路由，则ip_dooptions把ip_dst设置成off指向的IP地址。在源路由选项内，用外出接口的地址代替中间系统的地址，把位移增加，指向路由中的下一个地址。

3. 多播目的地

645-646 如果新的目的地址不是多播地址，就将forward设置成1，表明在处理完所有选项后，应该把分组转发而不是返回给ipintr。

源路由中的多播地址允许两个多播路由器通过不支持多播的中间路由器进行通信。第14章详细描述了这一技术。

卷1的8.5节有更多的源路由选项的例子。

9.6.1 save_rte函数

RFC 1122要求，在最终目的地，运输协议必须能够使用分组中被记录下来的路由。运输协议必须把该路由倒过来并附在所有应答的分组上。图 9-18中显示的save_rte函数，把源路由保存在如图9-16所示的ip_srcrt结构中。

ip_input.c

```
57 int      ip_nhops = 0;
58 static struct ip_srcrt {
59      struct in_addr dst;          /* final destination */
60      char    nop;                 /* one NOP to align */
61      char    srcopt[IPOPT_OFFSET + 1];    /* OPTVAL, OLEN and OFFSET */
62      struct in_addr route[MAX_IPOPTLEN / sizeof(struct in_addr)];
63 } ip_srcrt;
```

ip_input.c

图9-16 结构ip_srcrt

Route的声明是不正确的，尽管这不是个恶性错误。应该是

```
Struct in_addr route[(MAX_IPOPTLEN - 3)/sizeof(struct in_addr)];
```

对图9-26和图9-27的讨论详细地说明了这个问题。

57-63 该代码定义了ip_srcrt结构,并声明了静态变量 ip_srcrt。只有两个函数访问ip_srcrt：save_rte，把到达分组中的源路由复制到ip_srcrt中；ip_srcroute，创建一个与源路由方向相逆的路由。图 9-17说明了源路由处理过程。

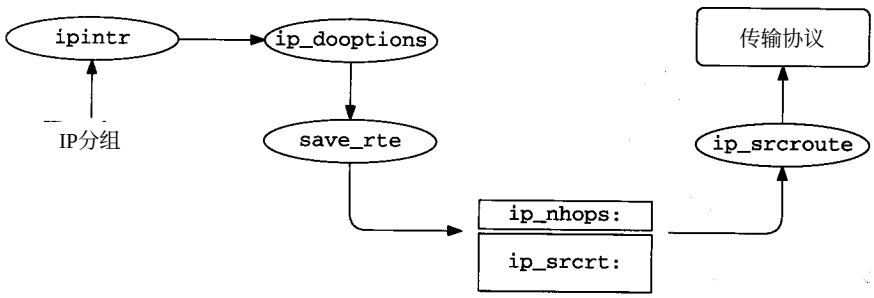


图9-17 对求逆后的源路由的处理

ip_input.c

```
759 void
760 save_rte(option, dst)
761 u_char *option;
762 struct in_addr dst;
763 {
764     unsigned olen;

765     olen = option[IPOPT_OLEN];
766     if (olen > sizeof(ip_srcrt) - (1 + sizeof(dst)))
767         return;
768     bcopy((caddr_t) option, (caddr_t) ip_srcrt.srcopt, olen);
769     ip_nhops = (olen - IPOPT_OFFSET - 1) / sizeof(struct in_addr);
770     ip_srcrt.dst = dst;
771 }
```

ip_input.c

图9-18 函数save_rte

759-771 当一个源路由选择的分组到达目的地时，ip_dooptions调用save_rte。option是一个指向分组的源路由选项的指针，dst是从分组首部来的ip_src (也就是，返回路由的目的地，图9-12中的HS)。如果选项的长度超过ip_srcrt结构，save_rte立即返回。

永远也不可能发生这种情况，因为ip_srcrt结构比最大选项长度(40字节)要大。

save_rte把该选项复制到ip_srcrt，计算并保存ip_nhops中源路由的跳数，把返回路由的目的地保存在dst中。

9.6.2 ip_srcroute函数

当响应某个分组时，ICMP和标准的运输层协议必须把分组带的任意源路由逆转。逆转源路由是通过ip_srcroute保存的路由构造的，如图9-19所示。

ip_input.c

```
777 struct mbuf *
778 ip_srcroute()
779 {
780     struct in_addr *p, *q;
781     struct mbuf *m;

782     if (ip_nhops == 0)
783         return ((struct mbuf *) 0);
784     m = m_get(M_DONTWAIT, MT_SOOPTS);
785     if (m == 0)
786         return ((struct mbuf *) 0);

787 #define OPTSIZ  (sizeof(ip_srcrt.nop) + sizeof(ip_srcrt.srcopt))

788     /* length is (nhops+1)*sizeof(addr) + sizeof(nop + srcrt header) */
789     m->m_len = ip_nhops * sizeof(struct in_addr) + sizeof(struct in_addr) +
790             OPTSIZ;

791     /*
792      * First save first hop for return route
793      */
794     p = &ip_srcrt.route[ip_nhops - 1];
795     *(mtod(m, struct in_addr *)) = *p--;

796     /*
797      * Copy option fields and padding (nop) to mbuf.
798      */
799     ip_srcrt.nop = IPOPT_NOP;
800     ip_srcrt.srcopt[IPOPT_OFFSET] = IPOPT_MINOFF;
801     bcopy((caddr_t) & ip_srcrt.nop,
802           mtod(m, caddr_t) + sizeof(struct in_addr), OPTSIZ);
803     q = (struct in_addr *) (mtod(m, caddr_t) +
804                             sizeof(struct in_addr) + OPTSIZ);
805 #undef OPTSIZ
806     /*
807      * Record return path as an IP source route,
808      * reversing the path (pointers are now aligned).
809      */
810     while (p >= ip_srcrt.route) {
811         *q++ = *p--;
812     }
813     /*
814      * Last hop goes to final destination.
```

图9-19 ip_srcroute 函数

```
815       */
816      *q = ip_srcrt.dst;
817      return (m);
818 }
```

ip_input.c

图9-19 (续)

777-783 ip_srcroute把保存在ip_srcrt结构中的源路由逆转后，返回与ipoption结构(图9-26)格式类似的结果。如果 ip_nhops是0，则没有保存的路由，所以ip_srcroute返回一个指针。

记得在图8-13中，当一个无效分组到达时，ipintr把ip_nhops清零。运输层协议必须调用ip_srcroute，并在下一个分组到达之前自己保存逆转后的路由。正如以前我们注意到的，这样做是正确的，因为 ipintr在处理分组时，在IP输入队列的下一个分组被处理之前都会调用运输层(TCP或UDP)的。

为源路由分配存储器缓存

784-786 如果ip_nhops非0，ip_srcroute就分配一个mbuf，并把m_len设置成足够大，以便包含第一跳目的地、选项首部信息(OPTSIZ)以及逆转后的路由。如果分配失败，则返回一个空指针，跟没有源路由一样。

p被初始化为指向到达路由的末尾，ip_srcroute把最后记录的地址复制到mbuf的前面，在这里它为外出的第一跳目的地开始逆转后的路由。然后该函数把一份 NOP选项(习题9.4)和源路由信息复制到mbuf中。

805-818 While循环把其余的IP 地址从源路由中以相反的顺序复制到 mbuf中。路由的最后一个地址被设置成到达分组中被 save_rte放在ip_srcrt.dst中的源站地址。返回一个指向mbuf的指针。图9-20说明了对图9-12的路由如何构造逆转的路由。

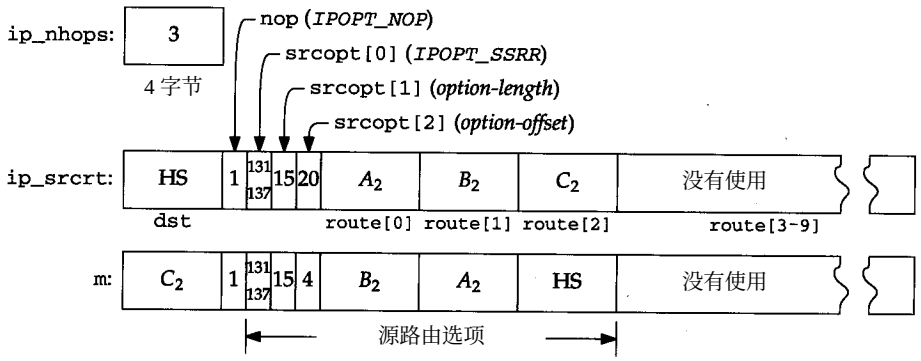


图9-20 ip_srcroute 逆转ip_srcrt 中的路由

9.7 时间戳选项

当分组穿过一个互联网时，时间戳选项使各个系统把它当前的时间表示记录在分组的选项内。时间是以从UTC的午夜开始计算的毫秒计，被记录在一个 32 bit的字段里。

如果系统没有准确的UTC(几分钟以内)或没有每秒更新至少15次，就不把它作为标准时间。非标准时间必须把时间戳字段的高比特位置位。

有三种时间戳选项类型，Net/3通过如图9-22所示的ip_timestamp结构访问。

114-133 如同ip结构(图8-10)一样，#ifs保证比特字段访问选项中正确的比特位。图9-21中列出了由ipt_flg指定的三种时戳选项类型。

ipt_flg	值	描 述
IPOOPT_TS_TSONLY	0	记录时间戳
IPOPT_TS_TSANDADDR	1	记录地址和时间戳
	2	保留
IPOPT_TS_PRESPEC	3	只在预先指定的系统记录时间戳
	4-15	保留

图9-21 ipt_flg 可能的值

ip.h

```
114 struct ip_timestamp {
115     u_char  ipt_code;           /* IPOPT_TS */
116     u_char  ipt_len;            /* size of structure (variable) */
117     u_char  ipt_ptr;            /* index of current entry */
118 #if BYTE_ORDER == LITTLE_ENDIAN
119     u_char  ipt_flg:4,          /* flags, see below */
120             ipt_oflw:4;         /* overflow counter */
121 #endif
122 #if BYTE_ORDER == BIG_ENDIAN
123     u_char  ipt_oflw:4,         /* overflow counter */
124             ipt_flg:4;          /* flags, see below */
125 #endif
126     union ipt_timestamp {
127         n_long  ipt_time[1];
128         struct ipt_ta {
129             struct in_addr ipt_addr;
130             n_long  ipt_time;
131         } ipt_ta[1];
132     } ipt_timestamp;
133 };
```

ip.h

图9-22 ip_timestamp 结构和常量

初始主机必须构造一个具有足够大的数据区存放可能的时间戳和地址的时间戳选项。对于ipt_flg为3的时间戳选项，初始主机在构造该选项时，填写要记录时间戳的系统的地址。图9-23显示了三种时间戳选项的结构。

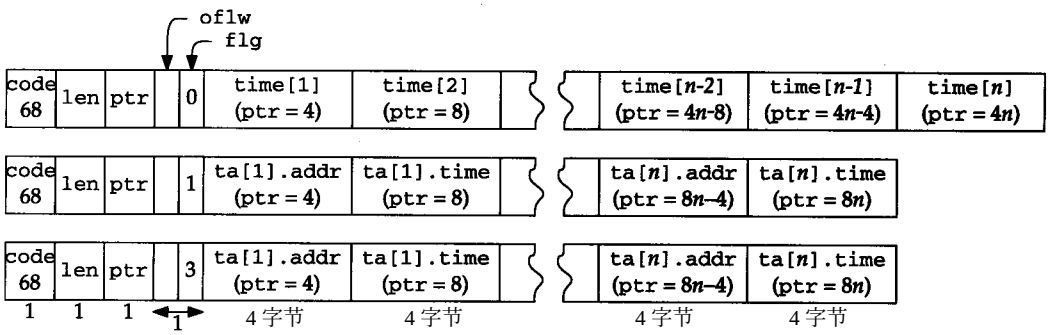


图9-23 三种时间戳选项(省略ipt_)

因为IP选项只能有40个字节，所以时戳选项限制只能有9个时戳(ipt_flg等于0)或4个地

址和时间戳对(ipt_flg等于1或3)。图9-24显示了对三种不同的时戳选项类型的处理。

674-684 如果选项长度小于5个字节(时戳选项的最小长度)，则ip_dooptions发出一个ICMP参数问题差错报文。oflw字段统计由于选项数据区满而无法登记时戳的系统个数。如果数据区满，则oflw加1；当它本身超过16(它是一个4 bit的字段)而溢出时，发出一个ICMP参数问题差错报文。

ip_input.c

```
674         case IPOPT_TS:
675             code = cp - (u_char *) ip;
676             ipt = (struct ip_timestamp *) cp;
677             if (ipt->ipt_len < 5)
678                 goto bad;
679             if (ipt->ipt_ptr > ipt->ipt_len - sizeof(long)) {
680                 if (++ipt->ipt_oflw == 0)
681                     goto bad;
682                 break;
683             }
684             sin = (struct in_addr *) (cp + ipt->ipt_ptr - 1);
685             switch (ipt->ipt_flg) {

686             case IPOPT_TS_TSONLY:
687                 break;

688             case IPOPT_TS_TSANDADDR:
689                 if (ipt->ipt_ptr + sizeof(n_time) +
690                     sizeof(struct in_addr) > ipt->ipt_len)
691                         goto bad;
692                 ipaddr.sin_addr = dst;
693                 ia = (INA) ifaof_ifpforaddr((SA) & ipaddr,
694                                             m->m_pkthdr.rcvif);
695                 if (ia == 0)
696                     continue;
697                 bcopy((caddr_t) & IA_SIN(ia)->sin_addr,
698                       (caddr_t) sin, sizeof(struct in_addr));
699                 ipt->ipt_ptr += sizeof(struct in_addr);
700                 break;

701             case IPOPT_TS_PRESPEC:
702                 if (ipt->ipt_ptr + sizeof(n_time) +
703                     sizeof(struct in_addr) > ipt->ipt_len)
704                         goto bad;
705                 bcopy((caddr_t) sin, (caddr_t) & ipaddr.sin_addr,
706                       sizeof(struct in_addr));
707                 if (ifa_ifwithaddr((SA) & ipaddr) == 0)
708                     continue;
709                 ipt->ipt_ptr += sizeof(struct in_addr);
710                 break;

711             default:
712                 goto bad;
713             }
714             ntime = iptime();
715             bcopy((caddr_t) & ntime, (caddr_t) cp + ipt->ipt_ptr - 1,
716                   sizeof(n_time));
717             ipt->ipt_ptr += sizeof(n_time);
718         }
719     }
```

ip_input.c

图9-24 函数ip_dooptions：时间戳选项处理

1. 只有时间戳

685-687 对于ipt_flg为0的时间戳选项(IPOPT_TS_TSONLY)，所有的工作都在switch语句之后再做。

2. 时间戳和地址

688-700 对于ipt_flg为1的时间戳选项(IPOPT_TS_TSANDADDR)，接收接口的地址被记录下来(如果数据区还有空间)，选项的指针前进一步。因为Net/3支持一个接收接口上的多IP地址，所以ip_dooptions调用ifaof_ifpforaddr选择与分组的初始目的地址(也就是在任何源路由选择发生之前的目的地)最匹配的地址。如果没有匹配，则跳过时间戳选项(INA和SA定义如图9-15所示)。

3. 预定地址上的时间戳

701-710 如果ipt_flg为3(IPOPT_TS_PRESPEC)，ifa_ifwithaddr确定选项中指定的下一个地址是否与系统的某个地址匹配。如果不匹配，该选项要求在这个系统上不处理；continue使ip_dooptions继续处理下一个选项。如果下一个地址与系统的某个地址匹配，则选项的指针前进到下一个位置，控制交给switch的后面。

4. 插入时间戳

711-713 default截获无效的ipt_flg值，并把控制传递到bad。

714-719 时间戳用switch语句后面的代码写入到选项中。iptime返回自从UTC午夜起到现在的毫秒数，ip_dooptions记录此时间戳，并增加此选项相对于下一个位置的偏移。

iptime函数

图9-25显示了iptime的实现。

ip_icmp.c

```
458 n_time
459 iptime()
460 {
461     struct timeval atv;
462     u_long  t;

463     microtime(&atv);
464     t = (atv.tv_sec % (24 * 60 * 60)) * 1000 + atv.tv_usec / 1000;
465     return (htonl(t));
466 }
```

ip_icmp.c

图9-25 函数iptime

458-466 microtime返回从UTC1970年1月1日午夜以来的时间，放在timeval结构中。从午夜以来的毫秒数用atv计算，并以网络字节序返回。

卷1的7.4节有几个时间戳选项的例子。

9.8 ip_insertoptions函数

我们在8.6节看到，ip_output函数接收一个分组和选项。当ip_forward调用该函数时，选项已经是分组的一部分，所以ip_forward总是把一个空选项指针传给ip_output。但是，运输层协议可能会把由ip_insertoptions(由图8-22中的ip_output调用)合并到分组中的选项传递给ip_forward。

ip_insertoptions希望选项在ipoption结构中被格式化，如图9-26所示。

ip_var.h
```
92 struct ipoption {
93     struct in_addr ipopt_dst;   /* first-hop dst if source routed */
94     char    ipopt_list[MAX_IPOPTLEN];   /* options proper */
95 };
```
ip_var.h

图9-26 结构ipoption

92-95 该结构只有两个成员：ipopt_dst，如果选项表中有源路由，则其中有第一跳目的地，ipopt_list，是一个最多40(MAX_IPOPTLEN)字节的选项矩阵，其格式我们在本章中已做了描述。如果选项表中没有源路由，则ipopt_dst全为0。

注意，ip_srcrt结构(图9-16)和由ip_srcroute(图9-19)返回的mbuf都符合由ipoption结构所指定的格式。图9-27把结构ip_srcrt和ipoption作了比较。

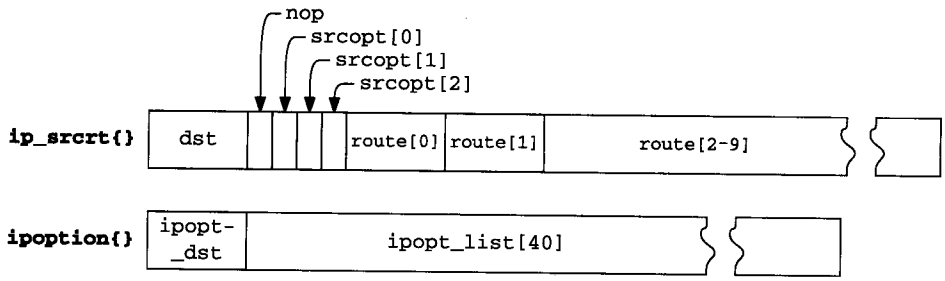


图9-27 结构ip_srcrt 和ipoption

结构ip_srcrt比ipoption多4个字节。路由矩阵的最后一个入口(route[9])永远都不会填上，因为这样的话，源路由选项将会有 44字节长，比IP首部所能容纳的要大(图9-16)。

函数ip_insertoptions如图9-28所示。

ip_output.c
```
352 static struct mbuf *
353 ip_insertoptions(m, opt, phlen)
354 struct mbuf *m;
355 struct mbuf *opt;
356 int    *phlen;
357 {
358     struct ipoption *p = mtod(opt, struct ipoption *);
359     struct mbuf *n;
360     struct ip *ip = mtod(m, struct ip *);
361     unsigned optlen;

362     optlen = opt->m_len - sizeof(p->ipopt_dst);
363     if (optlen + (u_short) ip->ip_len > IP_MAXPACKET)
364         return (m);             /* XXX should fail */
365     if (p->ipopt_dst.s_addr)
366         ip->ip_dst = p->ipopt_dst;
367     if (m->m_flags & M_EXT || m->m_data - optlen < m->m_pktdat) {
368         MGETHDR(n, M_DONTWAIT, MT_HEADER);
369         if (n == 0)
```

图9-28 函数ip_insertoptions

```
370             return (m);
371         n->m_pkthdr.len = m->m_pkthdr.len + optlen;
372         m->m_len -= sizeof(struct ip);
373         m->m_data += sizeof(struct ip);
374         n->m_next = m;
375         m = n;
376         m->m_len = optlen + sizeof(struct ip);
377         m->m_data += max_linkhdr;
378         bcopy((caddr_t) ip, mtod(m, caddr_t), sizeof(struct ip));
379     } else {
380         m->m_data -= optlen;
381         m->m_len += optlen;
382         m->m_pkthdr.len += optlen;
383         ovbcopy((caddr_t) ip, mtod(m, caddr_t), sizeof(struct ip));
384     }
385     ip = mtod(m, struct ip *);
386     bcopy((caddr_t) p->ipopt_list, (caddr_t) (ip + 1), (unsigned) optlen);
387     *phlen = sizeof(struct ip) + optlen;
388     ip->ip_len += optlen;
389     return (m);
390 }
```

ip_output.c

图9-28 (续)

352-364 ip_insertoptions有三个参数：m，外出的分组；opt，在结构中格式化的选项；phlen，一个指向整数的指针，在这里返回新首部的长度(在插入选项之后)。如果插入选项分组长度超过最大分组长度 65 535(IP_MAXPACKET)字节，则自动将选项丢弃。ip_dooptions认为ip_insertoptions永远都不会失败，所以无法报告差错。幸好，很少有应用程序会试图发送最大长度的数据报，更别说选项了。

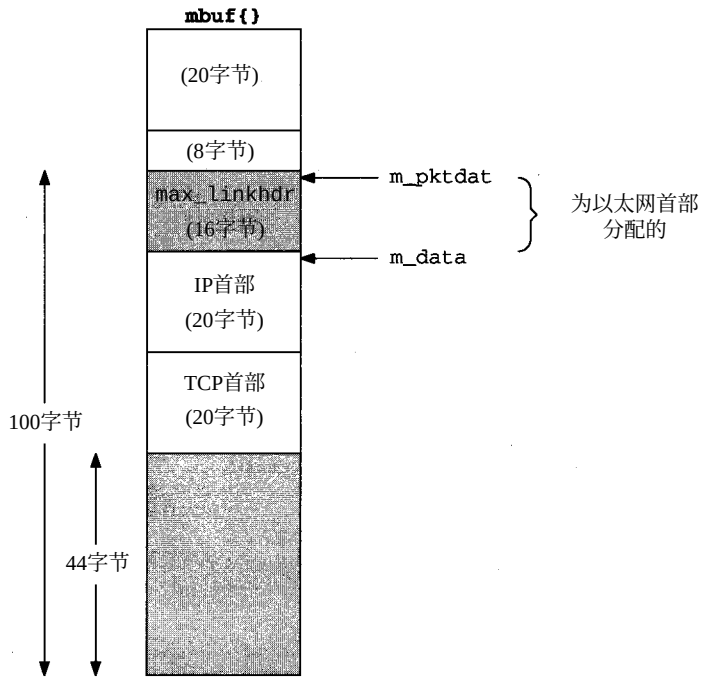


图9-29 函数ip_insertoptions ：TCP报文段

365-366 如果ipopt_dst.s_addr指定了一个非零地址，则选项中包括了源路由，并且分组首部的ip_dst被源路由中的第一跳目的地代替。

在26.2节中，我们将看到TCP调用MGETHDR为IP和TCP首部分配一个单独的mbuf。图9-29显示了在第367~378行代码执行之前，一个TCP报文段的mbuf结构。

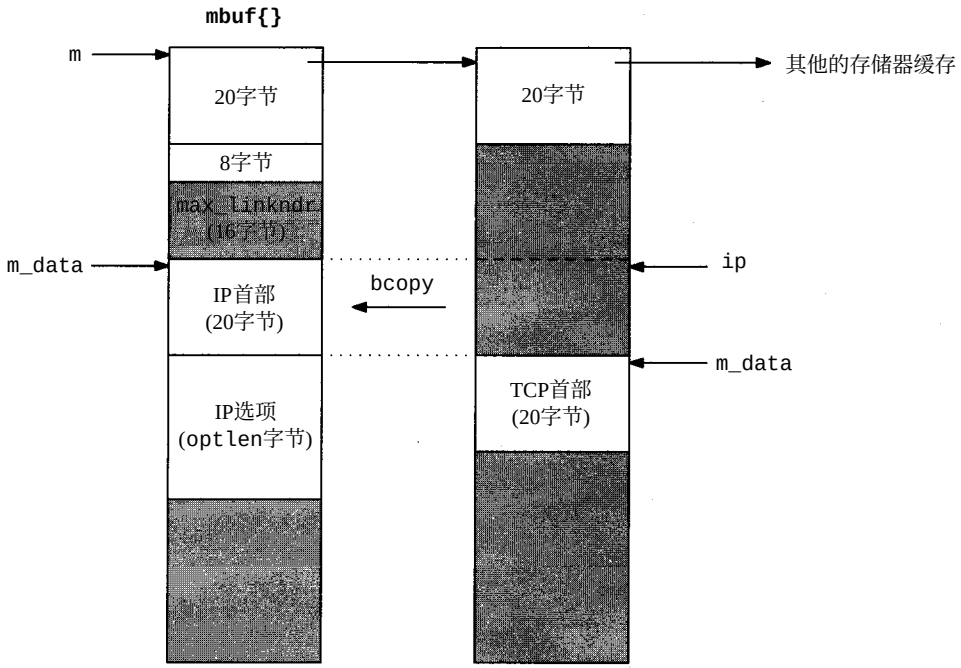


图9-30 函数ip_insertoptions ：在选项被复制后的TCP报文段

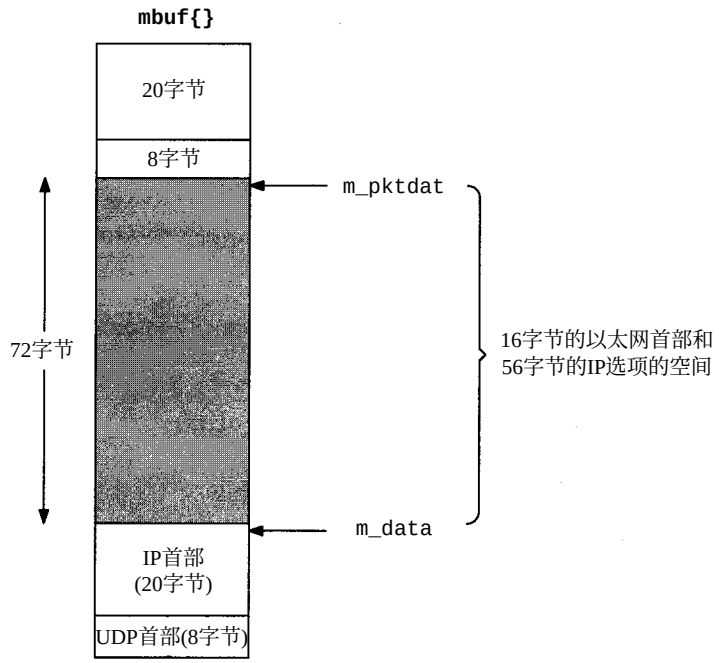


图9-31 函数ip_insertoptions ：UDP报文段

如果被插入的选项占据了多于16的字节数，则第367行的测试为真，并调用MGETHDR分配另一个mbuf。图9-30显示了选项被复制到新的mbuf后，该缓存的结构。

367-378 如果分组首部被存放在一簇，或者第一个缓存中没有多余选项的空间，则ip_insertoptions分配一个新的分组首部mbuf，初始化它的长度，从旧的缓存中把该IP首部截取下来，并把该首部从旧缓存中移动到新缓存中。

如23.6节中所述，UDP使用M_PREPEND把UDP和IP首部放置到缓存的最后，与数据分离。如图9-31所示。因为首部是放在缓存的最后，所以在缓存中总有空间存放选项，对UDP来说，第367行的条件总为假。

379-384 如果分组在缓存数据区的开始部分有存放选项的空间，则修改m_data和m_len，以包含optlen更多的字节。并且当前的IP首部被ovbcopy(能够处理源站和目的站的重叠问题)移走，为选项腾出位置。

385-390 ip_insertoptions现在可以把ipoption结构的成员ipopt_list直接复制到紧接在IP首部后面的缓存中。把新的首部长度存放在*phlen中，修改数据报长度(ip_len)，并返回一个指向分组首部缓存的指针。

9.9 ip_pcbopts函数

函数ip_pcbopts把IP选项表及IP_OPTIONS插口选项转换成ip_output希望的格式：ipoption结构。如图9-32所示。

ip_output.c

```
559 int
560 ip_pcbopts(pcbopt, m)
561 struct mbuf **pcbopt;
562 struct mbuf *m;
563 {
564     cnt, optlen;
565     u_char *cp;
566     u_char  opt;

567     /* turn off any old options */
568     if (*pcbopt)
569         (void) m_free(*pcbopt);
570     *pcbopt = 0;
571     if (m == (struct mbuf *) 0 || m->m_len == 0) {
572         /*
573          * Only turning off any previous options.
574          */
575         if (m)
576             (void) m_free(m);
577         return (0);
578     }
579     if (m->m_len % sizeof(long))
580                 goto bad;
581     /*
582      * IP first-hop destination address will be stored before
583      * actual options; move other options back
584      * and clear it when none present.
585      */
586     if (m->m_data + m->m_len + sizeof(struct in_addr) >= &m->m_dat[MLEN])
587                 goto bad;
588     cnt = m->m_len;
589     m->m_len += sizeof(struct in_addr);
```

图9-32 函数ip_pcbopts

```
590     cp = mtod(m, u_char *) + sizeof(struct in_addr);
591     ovbcopy(mtod(m, caddr_t), (caddr_t) cp, (unsigned) cnt);
592     bzero(mtod(m, caddr_t), sizeof(struct in_addr));
593     for (; cnt > 0; cnt -= optlen, cp += optlen) {
594         opt = cp[IPOPT_OPTVAL];
595         if (opt == IPOPT_EOL)
596             break;
597         if (opt == IPOPT_NOP)
598             optlen = 1;
599         else {
600             optlen = cp[IPOPT_OLEN];
601             if (optlen <= IPOPT_OLEN || optlen > cnt)
602                 goto bad;
603         }
604         switch (opt) {
605         default:
606             break;
607         case IPOPT_LSRR:
608         case IPOPT_SSRR:
609             /*
610              * user process specifies route as:
611              *  ->A->B->C->D
612              * D must be our final destination (but we can't
613              * check that since we may not have connected yet).
614              * A is first hop destination, which doesn't appear in
615              * actual IP option, but is stored before the options.
616              */
617             if (optlen < IPOPT_MINOFF - 1 + sizeof(struct in_addr))
618                         goto bad;
619             m->m_len -= sizeof(struct in_addr);
620             cnt -= sizeof(struct in_addr);
621             optlen -= sizeof(struct in_addr);
622             cp[IPOPT_OLEN] = optlen;
623             /*
624              * Move first hop before start of options.
625              */
626             bcopy((caddr_t) &cp[IPOPT_OFFSET + 1], mtod(m, caddr_t),
627                   sizeof(struct in_addr));
628             /*
629              * Then copy rest of options back
630              * to close up the deleted entry.
631              */
632             ovbcopy((caddr_t) (&cp[IPOPT_OFFSET + 1] +
633                                sizeof(struct in_addr)),
634                             (caddr_t) &cp[IPOPT_OFFSET + 1],
635                             (unsigned) cnt + sizeof(struct in_addr));
636             break;
637         }
638     }
639     if (m->m_len > MAX_IPOPTLEN + sizeof(struct in_addr))
640                 goto bad;
641     *pcbopt = m;
642     return (0);
643   bad:
644     (void) m_free(m);
645     return (EINVAL);
646 }
```

ip_output.c

图9-32 (续)

559-562 第一个参数，pcbopt引用指向当前选项表的指针。然后该函数用一个指向新的选项表的指针来代替该指针，这个新选项表是由第二个参数 m指向的缓存链所指定的选项构造而来。该过程所准备的选项表，将被包含在 IP_OPTIONS插口选项中，除了LSRR和SSRR选项的格式外，看起来象一个标准的 IP选项表。对这些选项，第一跳目的地址是作为路由的第一个地址出现的。图 9-14显示，在外出的分组中，第一跳目的地址是作为目的地址出现的，而不是路由的第一个地址。

1. 扔掉以前的选项

563-580 所有被m_free和*pcbopt扔掉的选项都被清除。如果该过程传来一个空缓存或者根本不传递缓存，则该函数不安装任何新的选项，并立即返回。

如果新选项表没有填充到 4 bit的边界，则 ip_pcbopts跳到bad，扔掉该表，并返回EINVAL。

该函数的其余部分重新安排该表，使其看起来象一个 ipoption结构。图9-33显示了这个过程。

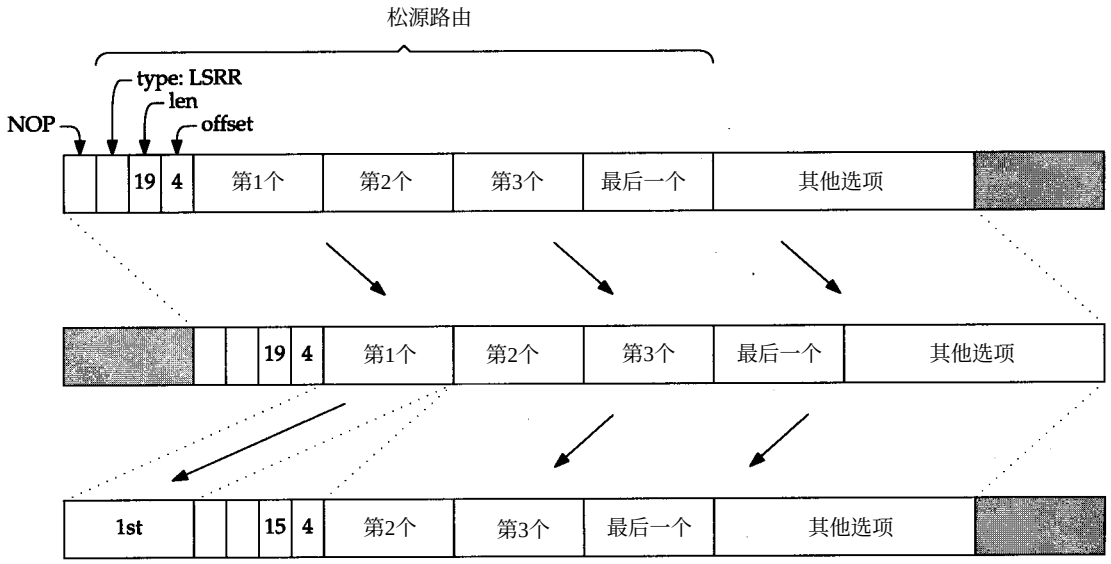


图9-33 ip_pcbopts 选项表处理

2. 为第一跳目的地腾出位置

581-592 如果缓存中没有位置，则把所有的数据都向缓存的末尾移动 4个字节(是一个in_addr结构的大小)。ovbcopy完成复制。bzero清除缓存开始的4个字节。

3. 扫描选项表

593-606 for循环扫描选项表，寻找LSRR和SSRR选项。对多字节选项，该循环也验证选项的长度是否合理。

4. 重新安排LSRR和SSRR选项

607-638 当该循环找到一个LSRR或SSRR选项时，它把缓存的大小、循环变量和选项长度减去4，因为选项的每一个地址将被移走，并被移到缓存的前面。

bcopy把第一个地址移走， ovbcopy把选项的其他部分移动 4个字节，来填补第一个地

址留下的空隙。

5. 清除

639-646 循环结束后，选项表的大小(包括第一跳地址)必须不超过44 (MAX_IPOPTLEN +4)字节。更长的选项表无法放入IP分组的首部。该表被保存在*pcbopt中，函数返回。

9.10 一些限制

除了管理和诊断工具生成的IP数据报外，很少出现选项。卷1讨论了两个最常用的工具，ping和traceroute。使用IP选项的应用程序很难写。编程接口的文档很少，也没有很好地标准化。许多厂商提供的应用程序，比如Telnet和FTP，并没有为用户提供方法，来指定如源路由等的选项。

在大的互联网上，记录路由、时间戳和源路由选项的用途被IP首部的最大长度所限制。许多路由含有的跳数远多于40选项字节所能表示的。当多选项在同一分组中出现时，所能得到的空间是几乎没有用的。IPv6用一个更为灵活的选项首部设计强调了这个问题。

在分片过程中，IP只把某些选项复制到非初始片上，因为重组时会扔掉非初始片上的选项。在目的主机上，运输层协议只能用到初始片上的选项(10.6节)。但有些选项，如源路由，即使在目的主机上，被非初始片丢弃，依然必须被复制到每个分片。

9.11 小结

本章中我们显示了IP选项的格式和处理过程。我们没有讨论安全和流ID选项，因为Net/3没有实现它们。

我们看到，多字节选项的大小是由源主机在构造它们时确定的。最大选项首部长度只有40字节，这严格限制了IP选项的使用。

源路由选项要求最多的支持。到达的源路由被save_rte保存，并保留在ip_srcroute中。通常不转发分组的主机可能转发源路由选择的分组，但是RFC 1122默认要求不允许这种功能。Net/3没有对这种特性的判断，总是转发源路由选择的分组。

最后，我们看到ip_insertoptions是如何把选项插入到一个外出的分组中去的。

习题

9.1 如果一个分组中有两个不同的源路由选项会发生什么情况？

9.2 一些商用路由器可以被配置成按照分组的IP目的地址扔掉它们。通过种方式，可以把一台或一组主机通过路由器隔离在更大的互联网之外。请描述源路由选择的分组如何绕过这个机制。假定网络中至少有一个主机，路由器没有阻塞，并转发源路由选择的数据报。

9.3 某些主机可能没有被配置成默认路由。这样主机就无法路由选择到其他与它直接相连的网络。请描述源路由如何与这种类型的主机通信。

9.4 为什么NOP采用如图9-16所示的ip_srcrt结构？

9.5 时间戳选项中非标准时间值会和标准时间值混淆吗？

9.6 ip_dooptions在处理其他选项之前要把分组的目的地址保存在dest中(图9-8)。为什么？

第10章　IP的分片与重装

10.1　引言

我们将第8章的IP的分片与重装处理问题推迟到本章来讨论。

IP具有一种重要功能，就是当分组过大而不适合在所选硬件接口上发送时，能够对分组进行分片。过大的分组被分成两个或多个大小适合在所选定网络上发送的 IP分片。而在去目的主机的路途中，分片还可能被中间的路由器继续分片。因此，在目的主机上，一个 IP数据报可能放在一个IP分组内，或者，如果在发送时被分片，就放在多个 IP分组内。因为各个分片可能以不同的路径到达目的主机，所以只有目的主机才有机会看到所有分片。因此，也只有目的主机才能把所有分片重装成一个完整的数据报，提交给合适的运输层协议。

图8-5显示在被接收的分组中， 0.3%(72 786/27 881 978)是分片， 0.12% (264 484/(29 447 726-796 084))的数据报是被分片后发送的。在 `world.std.com`上，被接收分组的9.5%是被分片的。`world`有更多的NFS活动，这是IP分片的主要来源。

IP首部内有三个字段实现分片和重装：标识字段 (`ip_id`)、标志字段(`ip_off`的3个高位比特)和偏移字段(`ip_off`的13个低位比特)。标志字段由三个1 bit标志组成。比特0是保留的，必须为0；比特1是“ 不分片 ”(DF)标志；比特2是“ 更多分片 ”(MF)标志。Net/3中，标志和偏移字段结合起来，由`ip_off`访问，如图10-1所示。

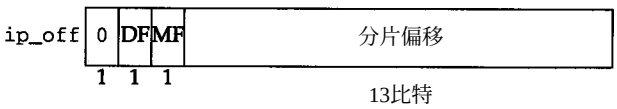


图10-1　`ip_off` 控制IP分组的分片

Net/3通过用IP_DF和 IP_MF掩去`ip_off`来访问DF和MF。IP实现必须允许应用程序请求在输出的数据报中设置DF比特。

> 当使用UDP或TCP时，Net/3并不提供对DF比特的应用程序级的控制。
>
> 进程可以用原始IP接口(第32章)构造和发送它自己的IP首部。运输层必须直接设置DF比特。例如，当TCP运行“ 路径MTU发现(*path* MTU *discovery*) ”时。

`ip_off`的其他13 bit指出在原始数据报内分片的位置，以8字节为单元计算。因而，除最后一个分片外，其他每个分片都希望是一个 8字节倍数的数据，从而使后面的分片从 8字节边界开始。图10-2显示了在原始数据报内的字节偏移关系，以及在分片的 IP首部内分片的偏移(`ip_off`的低位13 bit)。

图10-2显示了把一个最大的IP数据报分成8190个分片，除最后一个分片包含 3个字节外，其他每个分片都包含8个字节。图中还显示，除最后一个分片外，设置了其余分片的 MF比特。这是一个不太理想的例子，但它说明了一些实现中存在的问题。

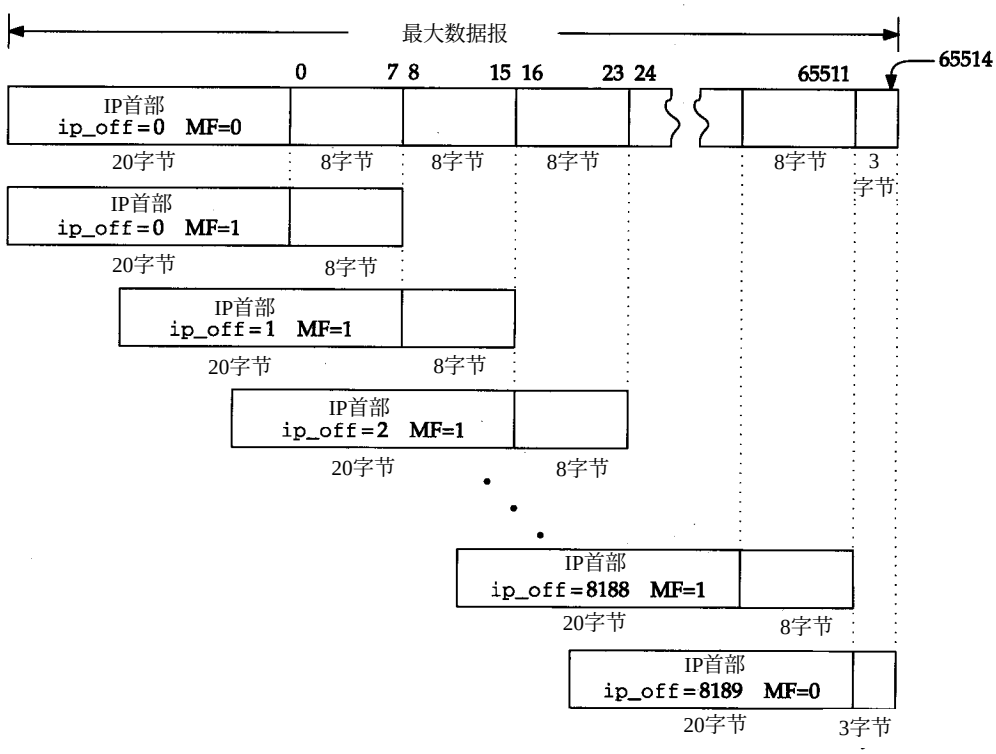


图10-2 65535字节的数据报的分片

原始数据报上面的数字是该数据部分在数据报内的字节偏移。分片偏移 (ip_off)是从数据报的数据部分开始计算的。分片不可能含有偏移超过 65514的字节，因为如果这样的话，重装的数据报会大于65535字节 —— 这是ip_len字段的最大值。这就限制了ip_off的最大值为8189(8189×8=65512)，只为最后一个分片留下3字节空间。如果有IP选项，则偏移还要小些。

因为IP互联网是无连接的，所以，在目的主机上，来自一个数据报的分片必然会与来自其他数据报的分片交错。ip_id唯一地标识某个特定数据报的分片。源系统用相同的源地址(ip_src)、目的地址(ip_dst)和协议(ip_p)值，作为数据报在互联网上生命期的值，把每个数据报的ip_id设置成一个唯一的值。

总而言之，ip_id标识了特定数据报的分片，ip_off确定了分片在原始数据报内的位置，除最后一个分片外，MF标识每个分片。

10.2 代码介绍

重装数据结构出现在一个头文件里。两个 C文件中有重装和分片处理的代码。这三个文件列在图10-3中。

文　件	描　述
netinet/ip_var.h	重装数据结构
netinet/ip_output.c	分片代码
netinet/ip_input.c	重装代码

图10-3 本章讨论的文件

10.2.1 全局变量

本章中只有一个全局变量，ipq。如图10-4所示。

变 量	类 型	描 述
ipq	struct ipq *	重装表

图10-4 本章介绍的全局变量

10.2.2 统计量

分片和重装代码修改的统计量如图 10-5所示。它们是图 8-4的ipstat结构中所包含统计量的子集。

ipstat成员	描 述
ips_cantfrag	要求分片但被DF比特禁止而没有发送的数据报数
ips_odropped	因为内存不够而被丢弃的分组数
ips_ofragments	被发送的分片数
ips_fragmented	为输出分片的分组数

图10-5 本章收集的统计量

10.3 分片

我们现在返回到ip_output，分析分片代码。记得在图 8-25中，如果分组正好适合选定出接口的MTU，就在一个链路级帧中发送它。否则，必须对分组分片，并在多个帧中将其发送。分组可以是一个完整的数据报或者它自己也是前边系统创建的分片。我们分三个部分讨论分片代码：

- 确定分片大小(图10-6)；
- 构造分片表(图10-7)；以及
- 构造第一个分片并发送分片 (图10-8)。

ip_output.c

```
253     /*
254      * Too large for interface; fragment if possible.
255      * Must be able to put at least 8 bytes per fragment.
256      */
257     if (ip->ip_off & IP_DF) {
258         error = EMSGSIZE;
259         ipstat.ips_cantfrag++;
260         goto bad;
261     }
262     len = (ifp->if_mtu - hlen) & ~7;
263     if (len < 8) {
264         error = EMSGSIZE;
265         goto bad;
266     }
```

ip_output.c

图10-6 函数ip_output ：确定分片大小

253-261 分片算法很简单，但由于对 mbuf结构和链的操作使实现非常复杂。如果 DF比特禁

止分片，则ip_output丢弃该分组，并返回EMSGSIZE。如果该数据报是在本地生成的，则运输层协议把该错误传回该进程；但如果分组是被转发的，则 ip_forward生成一个ICMP目的地不可达差错报文，并指出不分片就无法转发该分组 (图8-21)。

Net/3没有实现“路径MTU发现”算法，该算法用来搜索到目的主机的路径，并发现所有中间网络支持的最大传送单元。卷1的11.8节和24.2节讨论了UDP和TCP的路径MTU发现。

262-266 每个分片中的数据字节数，len的计算是用接口的MTU减去分组首部的长度后，舍去低位的3个比特(&~7)。后成为8字节倍数。如果MTU太小，使每个分片中无法含有8字节的数据，则ip_output返回EMSGSIZE。

每个新的分片中都包含：一个IP首部、某些原始分组中的选项以及最多 len长度的数据。

图10-7中的代码，以一个C的复合语句开始，构造了从第 2个分片开始的分片表。在表生成后(图10-8)，原来的分组被转换成第一个分片。

ip_output.c

```
267     {
268         int     mhlen, firstlen = len;
269         struct mbuf **mnext = &m->m_nextpkt;
270         /*
271          * Loop through length of segment after first fragment,
272          * make new header and copy data of each part and link onto chain.
273          */
274         m0 = m;
275         mhlen = sizeof(struct ip);
276         for (off = hlen + len; off < (u_short) ip->ip_len; off += len) {
277             MGETHDR(m, M_DONTWAIT, MT_HEADER);
278             if (m == 0) {
279                 error = ENOBUFS;
280                 ipstat.ips_odropped++;
281                 goto sendorfree;
282             }
283             m->m_data += max_linkhdr;
284             mhip = mtod(m, struct ip *);
285             *mhip = *ip;
286             if (hlen > sizeof(struct ip)) {
287                 mhlen = ip_optcopy(ip, mhip) + sizeof(struct ip);
288                 mhip->ip_hl = mhlen >> 2;
289             }
290             m->m_len = mhlen;
291             mhip->ip_off = ((off - hlen) >> 3) + (ip->ip_off & ~IP_MF);
292             if (ip->ip_off & IP_MF)
293                 mhip->ip_off |= IP_MF;
294             if (off + len >= (u_short) ip->ip_len)
295                 len = (u_short) ip->ip_len - off;
296             else
297                 mhip->ip_off |= IP_MF;
298             mhip->ip_len = htons((u_short) (len + mhlen));
299             m->m_next = m_copy(m0, off, len);
300             if (m->m_next == 0) {
301                 (void) m_free(m);
302                 error = ENOBUFS;    /* ??? */
303                 ipstat.ips_odropped++;
304                 goto sendorfree;
305             }
306             m->m_pkthdr.len = mhlen + len;
```

图10-7 函数ip_output ：构造分片表

```
307             m->m_pkthdr.rcvif = (struct ifnet *) 0;
308             mhip->ip_off = htons((u_short) mhip->ip_off);
309             mhip->ip_sum = 0;
310             mhip->ip_sum = in_cksum(m, mhlen);
311             *mnext = m;
312             mnext = &m->m_nextpkt;
313             ipstat.ips_ofragments++;
314         }
```

ip_output.c

图10-7 (续)

ip_output.c

```
315         /*
316          * Update first fragment by trimming what's been copied out
317          * and updating header, then send each fragment (in order).
318          */
319         m = m0;
320         m_adj(m, hlen + firstlen - (u_short) ip->ip_len);
321         m->m_pkthdr.len = hlen + firstlen;
322         ip->ip_len = htons((u_short) m->m_pkthdr.len);
323         ip->ip_off = htons((u_short) (ip->ip_off | IP_MF));
324         ip->ip_sum = 0;
325         ip->ip_sum = in_cksum(m, hlen);
326       sendorfree:
327         for (m = m0; m; m = m0) {
328             m0 = m->m_nextpkt;
329             m->m_nextpkt = 0;
330             if (error == 0)
331                 error = (*ifp->if_output) (ifp, m,
332                                            (struct sockaddr *) dst, ro->ro_rt);
333             else
334                 m_freem(m);
335         }
336         if (error == 0)
337             ipstat.ips_fragmented++;
338     }
```

ip_output.c

图10-8 函数ip_output ：发送分片

267-269 外部块允许在函数中离使用点更近一点的地方定义mhlen、firstlen和mnext。这些变量的范围一直到块的末尾，它们隐藏其他在块外定义的有相同名字的变量。

270-276 因为原来的缓存链现在成了第一个分片，所以for循环从第2个分片的偏移开始：hlen+len。对每个分片，ip_output采取以下动作：

- *277-284* 分配一个新的分组缓存，调整m_data指针，为一个16字节链路层首部(max_linkhdr)腾出空间。如果ip_output不这么做，则网络接口驱动器就必须再分配一个mbuf来存放链路层首部或移动数据。两种工作都很耗时，在这里就很容易避免。
- *285-290* 从原来的分组中把IP首部和IP选项复制到新的分组中。前者复制在一个结构中。ip_optcopy只复制那些将被复制到每个分片中的选项(10.4节)。
- *291-297* 设置分片包括MF比特的偏移字段(ip_off)。如果原来分组中已设置了MF比特，则在所有分片中都把MF置位。如果原来分组中没有设置MF比特，则除了最后一个分片外，其他所有分片中的MF都置位。
- *298* 为分片设置长度，解决首部小一些(ip_optcopy可能没有复制所有选项)，以及最后一个分片的数据区小一些的问题。以网络字节序存储长度。

- *299-305* 从原始分组中把数据复制到该分片中。如果必要，m_copy会再分配一个mbuf。如果m_copy失败，则发出ENOBUFS。sendorfree丢弃所有已被分配的缓存。
- *306-314* 调整新创建的分片的mbuf分组首部，使其具有正确的全长。把新分片的接口指针清零，把ip_off转换成网络字节序，计算新分片的检验和。通过 m_nextpkt把该分片与前面的分片链接起来。

在图10-8中，ip_output构造了第一个分片，并把每个分片传递到接口层。

315-325 把末尾多余的数据截断后，原来的分组就被转换成第一个分片，同时设置MF比特，把ip_len和ip_off转换成网络字节序，计算新的检验和。在这个分片中，保留所有的 IP选项。在目的主机重装时，只保留数据报的第一个分片的 IP选项(图10-28)。某些选项，如源路由选项，必须被复制到每个分片中，即使在重装时都被丢弃了。

326-338 此时，ip_output可能有一个完整的分片表，或者已经产生了错误，都必须把生成的那部分分片表丢弃。 for循环遍历该表，发送分片或者由于 error而丢弃分片。在发送期间遇到的所有错误都会使后面的分片被丢弃。

10.4 ip_optcopy函数

在分片过程中，ip_optcopy(图10-9)复制到达分组(如果分组是被转发的)或者原始数据报中(如果该数据报是在本地生成的)中的选项到外出的分片中。

ip_output.c

```
395 int
396 ip_optcopy(ip, jp)
397 struct ip *ip, *jp;
398 {
399     u_char *cp, *dp;
400     int     opt, optlen, cnt;

401     cp = (u_char *) (ip + 1);
402     dp = (u_char *) (jp + 1);
403     cnt = (ip->ip_hl << 2) - sizeof(struct ip);
404     for (; cnt > 0; cnt -= optlen, cp += optlen) {
405         opt = cp[0];
406         if (opt == IPOPT_EOL)
407             break;
408         if (opt == IPOPT_NOP) {
409             /* Preserve for IP mcast tunnel's LSRR alignment. */
410             *dp++ = IPOPT_NOP;
411             optlen = 1;
412             continue;
413         } else
414             optlen = cp[IPOPT_OLEN];
415         /* bogus lengths should have been caught by ip_dooptions */
416         if (optlen > cnt)
417             optlen = cnt;
418         if (IPOPT_COPIED(opt)) {
419             bcopy((caddr_t) cp, (caddr_t) dp, (unsigned) optlen);
420             dp += optlen;
421         }
422     }
423     for (optlen = dp - (u_char *) (jp + 1); optlen & 0x3; optlen++)
424         *dp++ = IPOPT_EOL;
425     return (optlen);
426 }
```

ip_output.c

图10-9 函数：确定分片大小

395-422 ip_optcopy的参数是：ip，一个指向外出分组的IP首部的指针；jp，一个指向新生成的分片的IP首部的指针；ip_optcopy初始化cp和dp指向每个分组的第一个选项，并在处理每个选项时把cp和dp向前移动。第一个for循环在每次重复时复制一个选项，当它遇到EOL选项或已经检查完所有选项时。NOP选项被复制，用来维持后继选项的对齐限制。

Net/2版本废除了NOP选项。

如果IPOPT_COPIED指示*copied*比特被置位，则ip_optcopy把选项复制到新片中。图9-5显示了哪些选项的*copied*比特是被置位的。如果某个选项的长度太大，就被截断；ip_dooptions应该已经发现这种错误了。

423-426 第2个for循环把选项表填充到4字节的边界。由于分组首部长度(ip_hlen)是以4字节为单位计算的，所以需要这个操作。这也保证了后面跟着的运输层首部与4字节边界对齐。这样会提高性能，因为在许多运输层协议的设计中，如果运输层首部从一个32 bit边界开始，那么32 bit首部字段将按照32 bit边界对齐。在某些机器上，CPU访问32 bit对齐的字有困难，这时，这种字节安排就提高了CPU的性能。

图10-10显示了ip_optcopy的运行。

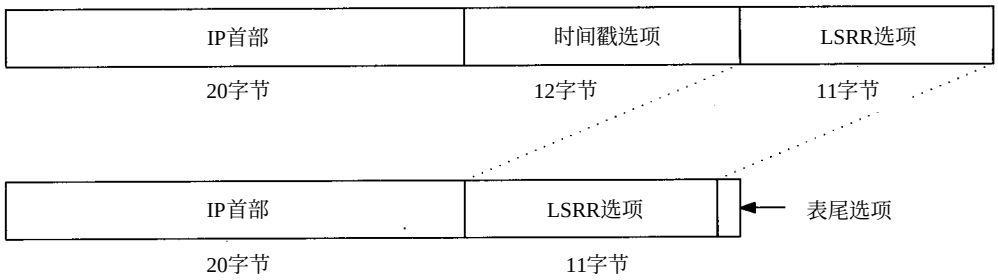


图10-10 在分片中并不复制所有选项

在图10-10中，我们看到ip_optcopy不复制时间戳选项(它的*copied*比特为0)，但却复制LSRR选项(它的*copied*比特为1)。为了把新选项与4字节边界对齐，ip_optcopy也增加了一个EOL选项。

10.5 重装

到目前为止，我们已经讨论了数据报(或片)的分片，现在再回到ipintr讨论重装过程。在图8-15中，我们把ipintr中的重装代码省略了，并推迟对它的讨论。ipintr可以把数据报整个地交给运输层处理。ipintr接收的分片被传给ip_reass，由它尝试把分片重装成一个完整的数据报。图10-11显示了ipintr的代码。

271-279 我们知道ip_off包含DF比特、MF比特以及分片偏移。如果MF比特或分片偏移非零，则DF就被掩盖掉了，分组就是一个必须被重装的分片。如果两者都为零，则分组就是一个完整的数据报。跳过重装代码，执行图10-11中最后的else语句，它从全部数据报长度中排除了首部长度。

280-286 m_pullup把位于外部簇上的数据移动到mbuf的数据区。我们记得，如果一个缓存区无法容纳外部簇上的一个IP分组，则SLIP接口(5.3节)可能会把该分组整个返回。m_devget也会全部返回外部簇上的某个IP分组(2.6节)。在mtod宏(2.6节)开始工作之前，

m_pullup必须把IP首部从外部簇上移到mbuf的数据区中去。

ip_input.c

```
271   ours:
272     /*
273      * If offset or IP_MF are set, must reassemble.
274      * Otherwise, nothing need be done.
275      * (We could look in the reassembly queue to see
276      * if the packet was previously fragmented,
277      * but it's not worth the time; just let them time out.)
278      */
279     if (ip->ip_off & ~IP_DF) {
280         if (m->m_flags & M_EXT) {    /* XXX */
281             if ((m = m_pullup(m, sizeof(struct ip))) == 0) {
282                 ipstat.ips_toosmall++;
283                 goto next;
284             }
285             ip = mtod(m, struct ip *);
286         }
287         /*
288          * Look for queue of fragments
289          * of this datagram.
290          */
291         for (fp = ipq.next; fp != &ipq; fp = fp->next)
292             if (ip->ip_id == fp->ipq_id &&
293                 ip->ip_src.s_addr == fp->ipq_src.s_addr &&
294                 ip->ip_dst.s_addr == fp->ipq_dst.s_addr &&
295                 ip->ip_p == fp->ipq_p)
296                 goto found;
297         fp = 0;
298       found:
299         /*
300          * Adjust ip_len to not reflect header,
301          * set ip_mff if more fragments are expected,
302          * convert offset of this to bytes.
303          */
304         ip->ip_len -= hlen;
305         ((struct ipasfrag *) ip)->ipf_mff &= ~1;
306         if (ip->ip_off & IP_MF)
307             ((struct ipasfrag *) ip)->ipf_mff |= 1;
308         ip->ip_off <<= 3;
309         /*
310          * If datagram marked as having more fragments
311          * or if this is not the first fragment,
312          * attempt reassembly; if it succeeds, proceed.
313          */
314         if (((struct ipasfrag *) ip)->ipf_mff & 1 || ip->ip_off) {
315             ipstat.ips_fragments++;
316             ip = ip_reass((struct ipasfrag *) ip, fp);
317             if (ip == 0)
318                 goto next;
319             ipstat.ips_reassembled++;
320             m = dtom(ip);
321         } else if (fp)
322             ip_freef(fp);
323     } else
324         ip->ip_len -= hlen;
```

ip_input.c

图10-11 函数ipintr：分片处理

287-297 Net/3在一个全局双向链表ipq上记录不完整的数据报。这个名字可能容易产生误解，因为这个数据结构并不是一个队列。也就是说，可以在表的任何地方插入和删除，并不限制一定要在末尾。我们将用名词“表(*list*)”来强调这个事实。

ipintr对表进行线性搜索，为当前分片找到合适的数据报。记住分片是由4元组{ip_id、ip_src、ip_dst和ip_p}唯一标识的。ipq的每个入口是一个分片表，如果ipintr找到一个匹配，则fp指向匹配的表。

Net/3采用线性搜索来访问它的许多数据结构。尽管简单，但是当主机支持大量网络连接时，这种方法就成为瓶颈。

298-303 在found语句，ipintr为方便重装，修改了分组：

- *304* ipintr修改ip_len，从中减去标准IP首部和任何选项。我们必须牢记这一点，以免混淆对标准ip_len解释的理解。标准ip_len中包含了标准首部、选项和数据。如果跳过重装代码，ip_len也会被改变，因为这个分组不是一个分片。
- *305-307* ipintr把MF标志复制到ipf_mff的低位，把ip_tos覆盖掉(&=~1只清除低位)。注意，在ipf_mff成为一个有效成员之前，必须把ip指一个ipasfrag结构。10.6节和图10-14描述了ipasfrag结构。

尽管RFC 1122要求IP层提供某种机制，允许运输层为每个外出的数据报设置ip_tos比特。但它只推荐，在目的主机，IP层把ip_tos值传给运输层。因为TOS字段的低位字节必须总是0，所以当重装算法使用ip_off(通常在这里找到MF比特)时，可以得到MF比特。

现在，可以把ip_off作为一个16 bit偏移，而不是3个标志比特和一个13 bit偏移来访问了。

- *308* 用8乘ip_off，把它从以8字节为单元转换成以1字节为单元。

ipf_mff和ip_off决定ipintr是否应该重装。图10-12描述了不同的情况及相应的动作，其中fp指向的是系统以前为该数据报接收的分片表。许多工作是由ip_reass做的。

309-322 如果ip_reass通过把当前分片与以前收到的分片组合在一起，能重装成一个完整的数据报，它就返回指向该重装好的数据报的指针。如果没有重装好，则ip_reass保存该分片，ipintr跳到next去处理下一个分片(图8-12)。

ip_off	ipf_mff	fp	描　述	动　作
0	假	空	完整数据报	没有要求重装
0	假	非空	完整数据报	丢弃前面的分片
任意	真	空	新数据报的分片	用这个分片初始化新的分片表
任意	真	非空	不完整新数据报的分片	插入到已有的分片表中，尝试重装
非零	假	空	新数据报的末尾分片	初始化新的分片表
非零	假	非空	不完整新数据报的末尾分片	插入到已有的分片表中，尝试重装

图10-12 ipintr和ip_reass中的IP分片处理

323-324 当到达一个完整的数据报时，就选择这个else分支，并按照前面的叙述修改ip_hlen。这是一个普通的流，因为收到的大多数数据报都不是分片。

如果重装处理产生一个完整的数据报，ipintr就把这个完整的数据报上传给合适的运输

层协议(图8-15)：

```
(*inetsw[ip_protox[ip->ip_p]].pr_input)(m,hlen);
```

10.6 ip_reass函数

ipintr把一个要处理的分片和一个指针传给 ip_reass，其中指针指向的是 ipq中匹配的重装首部。ip_reass可能重装成功并返回一个完整的数据报，可能把该分片链接到数据报的重装链表上，等待其他分片到达后重装。每个重装链表的表头是一个 ipq结构，如图10-13所示。

ip_var.h

```
52 struct ipq {
53     struct ipq *next, *prev;     /* to other reass headers */
54     u_char  ipq_ttl;             /* time for reass q to live */
55     u_char  ipq_p;               /* protocol of this fragment */
56     u_short ipq_id;              /* sequence id for reassembly */
57     struct ipasfrag *ipq_next, *ipq_prev;
58     /* to ip headers of fragments */
59     struct in_addr ipq_src, ipq_dst;
60 };
```

ip_var.h

图10-13 ipq 结构

52-60 用来标识一个数据报分片的四个字段，ip_id、ip_src、ip_dst和ip_p，被保存在每个重装链表表头的 ipq结构中。Net/3用next和prev构造数据报链表，用 ipq_next和ipq_prev构造分片的链表。

到达分组的IP首部在被放在重装链表之前，首先被转换成一个 ipasfrag结构(图10-14)。

ip_var.h

```
66 struct  ipasfrag {
67 #if BYTE_ORDER == LITTLE_ENDIAN
68     u_char  ip_hl:4,
69         ip_v:4;
70 #endif
71 #if BYTE_ORDER == BIG_ENDIAN
72     u_char  ip_v:4,
73         ip_hl:4;
74 #endif
75     u_char  ipf_mff;        /* XXX overlays ip_tos: use low bit
76                              * to avoid destroying tos;
77                              * copied from (ip_off&IP_MF) */
78     short   ip_len;
79     u_short ip_id;
80     short   ip_off;
81     u_char  ip_ttl;
82     u_char  ip_p;
83     u_short ip_sum;
84     struct  ipasfrag *ipf_next; /* next fragment */
85     struct  ipasfrag *ipf_prev; /* previous fragment */
86 };
```

ip_var.h

图10-14 ipasfrag 结构

66-86 ip_reass在一个由ipf_next和ipf_prev链接起来的双向循环链表上，收集某个数据报的分片。这些指针覆盖了 IP首部的源地址和目的地址。ipf_mff成员覆盖ip结构中的

ip_tos。其他成员是相同的。

图10-15显示了分片首部链表(ipq)和分片(ipasfrag)之间的关系。

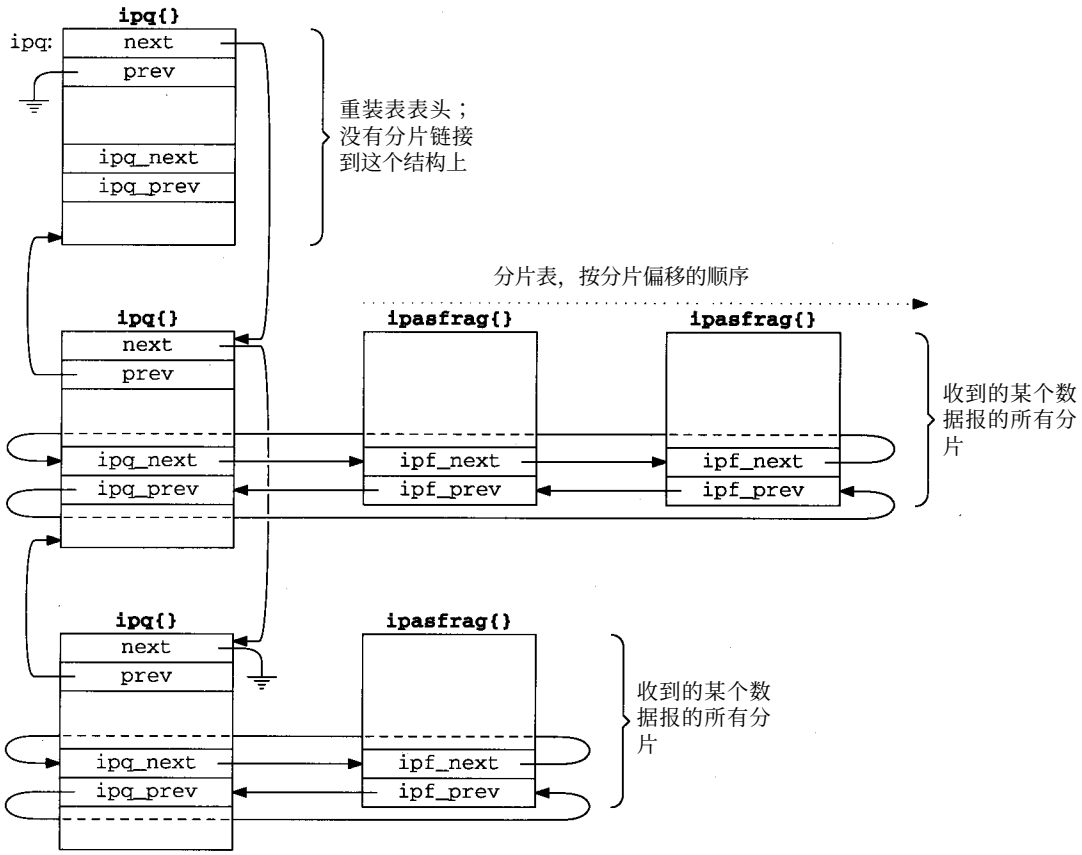


图10-15 分片首部链表ipq和分片

图10-15的左下部是重装首部的链表。表中第一个节点是全局ipq结构，ipq。它永远不会有自己的相关分片表。ipq表是一个双向链表，用于支持快速插入和删除。next和prev指针指向前一个和后一个ipq结构，用终止结构的角上的箭头表示。

图10-15仍然没有显示重装结构的所有复杂性。重装代码很难跟踪，因为它完全依靠把指针指向底层mbuf上的三个不同的结构。我们已经接触过这个技术了，例如，当一个ip结构覆盖某个缓存的数据部分时。

图10-16显示了mbuf、ipq结构、ipasfrag结构和ip结构之间的关系。

图10-16中含有大量信息：

- 所有结构都放在一个mbuf的数据区内。
- ipq链表由next和prev链接起来的ipq结构组成。每个ipq结构保存了唯一标识一个IP数据报的四个字段(图10-16中的阴影部分)。
- 当作为分片链表的头访问时，每个ipq结构被看成是一个ipasfrag结构。这些分片由ipf_next和ipf_prev链接起来，分别覆盖了ipq结构的ipq_next和ipq_prev成员。
- 每个ipasfrag结构都覆盖了到达分片的ip结构，与分片一起到达的数据在缓存中跟在该结构之后。ipasfrag结构的阴影部分的成员的含义与其在ip结构中不太相同。

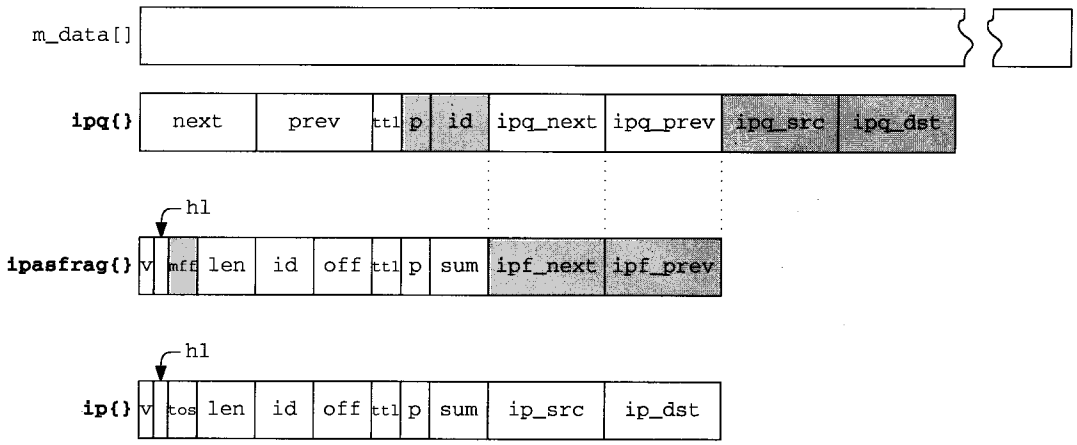


图10-16 可通过多种结构访问的一段内存区

图10-15显示了这些重装结构之间的物理连接，图10-16显示了ip_reass使用的覆盖技术。图10-17从逻辑的观点说明重装结构：该图显示了三个数据报的重装，以及 ipq链表和ipasfrag结构之间的关系。

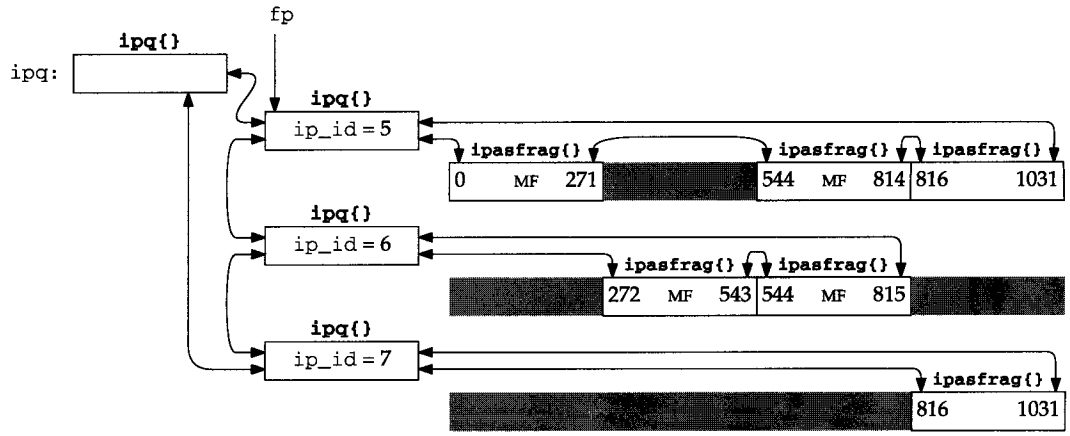


图10-17 三个IP数据报的重装

每个重装链表的表头包含原始数据报的标识符、协议、源和目的地址。图中只显示了ip_id字段。分片表通过偏移字段排序，如果 MF比特被置位，则用 MF标志分片，缺少的分片出现在阴影里。每个分片内的数字显示了该分片的开始和结束字节相对于原始数据报数据区的相对偏移，而不是相对于原始数据报的 IP首部。

这个例子是用来说明三个没有IP选项的UDP数据报，其中每个数据报都有1024字节的数据。每个数据报的全长是1052(20+8+1024)字节，正好适合1500字节以太网MTU。在到目的主机的途中，这些数据报会遇到一个 SLIP链路，该链路上的路由器对数据报分片，使其大小适于放在典型的296字节的SLIP MTU中。每个数据报分4个分片到达。第1个分片中包含一个标准的20字节IP首部，8字节UDP首部和264字节数据。第2个和第3个分片中包含一个20字节IP首部和272字节数据。最后一个分片中有一个20字节首部和216字节数据(1032=272×3+216)。

在图10-17中，数据报 5缺少一个包含 272~543字节的分片。数据报 6缺少第一个分片，

0~271字节，以及最后一个从偏移816开始的分片。数据报7缺少前三个分片，0~815。

图10-18列出了ip_reass。前面讲到，当目的地是本主机的某个IP分片到达时，在处理完所有选项后，ipintr会调用ip_reass。

ip_input.c

```
337 /*
338  * Take incoming datagram fragment and try to
339  * reassemble it into whole datagram.  If a chain for
340  * reassembly of this datagram already exists, then it
341  * is given as fp; otherwise have to make a chain.
342  */
343 struct ip *
344 ip_reass(ip, fp)
345 struct ipasfrag *ip;
346 struct ipq *fp;
347 {
348     struct mbuf *m = dtom(ip);
349     struct ipasfrag *q;
350     struct mbuf *t;
351     int     hlen = ip->ip_hl << 2;
352     int     i, next;
353     /*
354      * Presence of header sizes in mbufs
355      * would confuse code below.
356      */
357     m->m_data += hlen;
358     m->m_len -= hlen;
                [redacted]
465   dropfrag:
466     ipstat.ips_fragdropped++;
467     m_freem(m);
468     return (0);
469 }
```

ip_input.c

图10-18 函数ip_reass：数据报重装

343-358 当调用ip_reass时，ip指向分片fp或者指向匹配的ipq结构或者为空。因为重装只涉及每个分片的数据部分，所以ip_reass调整含有该分片的mbuf的m_data和m_len，减去每个分片的IP首部。

465-469 在重装过程中，如果产生错误，该函数就跳到dropfrag。dropfrag增加ips_fragdropped，丢弃该分片，并返回一个空指针。

在运输层丢弃分片通常会严重降低性能，因为必须重传整个数据报。TCP谨慎地避免分片，但是UDP应用程序必须采取步骤以避免对自己分片。[Kent和Mogul 1987] 解释了为什么应该避免分片。

所有IP实现必须能够重装最多576字节的数据报。没有通用的方法来确定远程主机能重装的最大数据报的大小。我们将在27.5节中看到TCP提供了一个机制，可以确定远程主机所能处理的最大数据报的大小。UDP没有这样的机制，所以许多基于UDP的协议(例如，RIP、TFTP、BOOTP、SNMP和DNS)，都限制在576字节左右。

我们将分7个部分显示重装代码，从图10-19开始。

ip_input.c

```
359     /*
360      * If first fragment to arrive, create a reassembly queue.
361      */
362     if (fp == 0) {
363         if ((t = m_get(M_DONTWAIT, MT_FTABLE)) == NULL)
364             goto dropfrag;
365         fp = mtod(t, struct ipq *);
366         insque(fp, &ipq);
367         fp->ipq_ttl = IPFRAGTTL;
368         fp->ipq_p = ip->ip_p;
369         fp->ipq_id = ip->ip_id;
370         fp->ipq_next = fp->ipq_prev = (struct ipasfrag *) fp;
371         fp->ipq_src = ((struct ip *) ip)->ip_src;
372         fp->ipq_dst = ((struct ip *) ip)->ip_dst;
373         q = (struct ipasfrag *) fp;
374         goto insert;
375     }
```

ip_input.c

图10-19 函数ip_reass ：创建重装表

1. 创建重装表

359-366 当fp为空时，ip_reass用新的数据报的第一个分片创建一个重装表。它分配一个mbuf来存放新表的头(一个ipq结构)，并调用insque，把该结构插入到重装表的链表中。

图10-20列出了操作数据报和分片链表的函数。

Net/3的386版本是在machdep.c文件中定义insque和remque函数的。每台机器都有自己的文件，在其中定义核心函数，通常是为提高性能。该文件也包括与机器体系结构有关的函数，包括中断处理支持、 CPU和设备配置以及内存管理函数。

insque和remque的存在主要是为了维护内核执行队列。 Net/3可把它们用于数据报重装链表，因为链表具有下一个和前一个两个指针，分别作为各自节点结构的前两个成员。对任何类型结构的链表，这些函数同样可以用，尽管编译器可能会发出一些警告。这也是另一个通过两个不同结构访问内存的例子。

在所有内核结构里，下一个指针通常位于前一个指针的前面 (例如，图10-14)。这是因为insque和remque最早是在VAX上用insque和remque硬件指令实现的，这些指令要求前向和后向指针具有这种顺序。

分片表不是用ipasfrag结构的前两个成员链接起来的 (图10-14)，所以Net/3调用ip_deq和ip_enq而不是insque和remque。

函 数	描 述
insque	紧接在*prev*后面插入*node* void **insque**(*void *node*, void* *prev*);
Remque	把*node*从表中移走 void **remque**(void *node);
ip_enq	紧接在分片*prev*后面插入分片*p* void **ip_enq**(struct ipasfrag*p*,* Struct ipasfrag *prev*);
ip_deq	移走分片*p* void **ip_deq**(struct ipasfrag*p*)*;

图10-20 ip_reass 采用的队列函数

2. 重装超时

367 RFC 1122要求有生命期字段(ipq_ttl)，并限制Net/3等待分片以完成一个数据报的时间。这与IP首部的TTL字段是不同的，IP首部的TTL字段是为了限制分组在互联网中循环的时间。重用IP首部的TTL字段作为重装超时的原因在于，一旦分片到达它的最终目的地，就不再需要首部TTL。

在Net/3中，重装超时的初始值设为60 (IPFRAGTTL)。因为每次内核调用ip_slowtimo时，ipq_ttl就减去1，而内核每500 ms调用ip_slowtimo一次。如果系统在接收到数据报的任一分片30秒后，还没有组装好一个完整的IP数据报，那么系统就丢弃该IP重装链表。重装定时器在链表被创建后的第一次调用ip_slowtimo时开始计时。

RFC 1122推荐重装时间在60~120秒内，并且当收到数据报的第一个分片且定时器超时时，向源主机发出一个ICMP超时差错报文。重装后，总是丢弃其他分片的首部和选项，并且在ICMP差错报文中必须包含出错数据报的前64 bit(或者，如果该数据报短于8字节，就可以少一些)。所以，如果内核还没有接收到分片0，它就不能发ICMP报文。

Net/3的定时器要短一些，并且当丢弃分片时，Net/3不发送ICMP报文。要求返回数据报的前64 bit保证含有运输层首部的前端，这样就可以把差错报文返回给发生错误的应用程序。注意，因为这个原因，TCP和UDP有意把它们的端口号放在首部的前8个字节。

3. 数据报标识符

368-375 ip_reass在分配给该数据报的ipq结构中保存ip_p、ip_id、ip_src和ip_dst，让ipq_next和ipq_prev指针指向该ipq结构(也就是说，它用一个节点构造一个循环链表)，让q指向这个结构，并跳到insert(图10-25)，把第一个分片ip插入到新的重装表中去。

ip_reass的下一个部分(图10-21)是当fp不空，并已当前表中为新的分片找到正确位置后执行的。

ip_input.c

```
376     /*
377      * Find a fragment which begins after this one does.
378      */
379     for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = q->ipf_next)
380         if (q->ip_off > ip->ip_off)
381             break;
```

ip_input.c

图10-21 函数ip_reass：在重装链表中找位置

376-381 因为fp不空，所以for循环搜索数据报的分片链表，找到一个偏移大于ip_off的分片。

在目的主机上，分片包含的字节范围可能会相互覆盖。发生这种情况的原因是，当一个运输层协议重传某个数据报时，采用与原来数据报不同的路由；而且，分片的模式也可能不同，这就导致在目的主机上的相互覆盖。传输协议必须能强制IP使用原来的ID字段，确保目的主机识别出该数据报是重传的。

Net/3并不为运输层协议提供机制保证在重传的数据报中重用IP ID字段。在准备

新数据报时，ip_output通过增加全局整数ip_id来赋一个新值(图8-22)。尽管如此，Net/3系统也能从让运输层用相同ID字段重传IP数据报的系统上接收重叠的分片。

图10-22说明分片可能会以不同的方式与已经到达的分片重叠。分片是按照它们到达目的主机的顺序编号的。重装的分片在图10-22底部显示，分片的阴影部分是被丢弃的多余字节。

在下面的讨论中，早到(*earlier*)分片是指先前到达主机的分片。

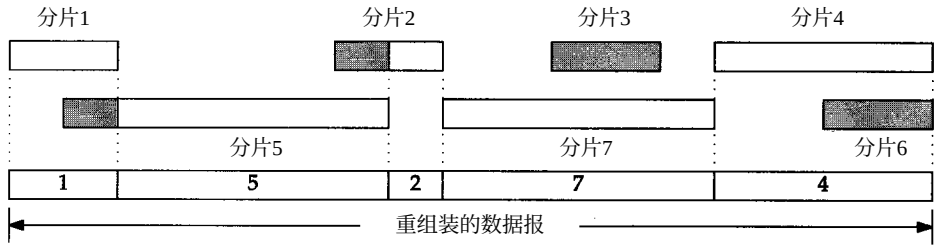


图10-22 可能会在目的主机重叠的分片的字节范围

图10-23中代码截断或丢弃到达的分片。

382-396 ip_reass把新片中与早到分片末尾重叠的字节丢弃，截断重复的部分(图10-22中分片5的前部)，或者，如果新分片的所有字节已经在早先的分片中 (分片4)出现，就丢弃整个新分片(分片6)。

ip_input.c

```
382     /*
383      * If there is a preceding fragment, it may provide some of
384      * our data already.  If so, drop the data from the incoming
385      * fragment.  If it provides all of our data, drop us.
386      */
387     if (q->ipf_prev != (struct ipasfrag *) fp) {
388         i = q->ipf_prev->ip_off + q->ipf_prev->ip_len - ip->ip_off;
389         if (i > 0) {
390             if (i >= ip->ip_len)
391                 goto dropfrag;
392             m_adj(dtom(ip), i);
393             ip->ip_off += i;
394             ip->ip_len -= i;
395         }
396     }
```

ip_input.c

图10-23 函数ip_reass ：截断到达分组

图10-24中的代码截断或丢弃已有的分片。

397-412 如果当前分片部分地与早到分片的前端部分重叠，就把早到分片中重复的数据截掉(图10-22中分片2的前部)。丢弃所有与当前分片完全重叠的早到分片 (分片3)。

图10-25中，到达分片被插入到重装链表。

413-426 在截断后，ip_enq把该分片插入链表，并扫描整个链表，确定是否所有分片全部到达。如果还缺少分片，或链表最后一个分片的ipf_mff被置位，ip_reass就返回0，并等待更多的分片。

当目前的分片完成一个数据报后，整个链表被图10-26所示的代码转换成一个mbuf链。

427-440 如果某个数据报的所有分片都被接收下来，while循环用m_cat把分片重新构造

成数据报。

ip_input.c

```
397     /*
398      * While we overlap succeeding fragments trim them or,
399      * if they are completely covered, dequeue them.
400      */
401     while (q !=(struct ipasfrag *) fp && ip->ip_off+ip->ip_len > q->ip_off){
402         i = (ip->ip_off + ip->ip_len) - q->ip_off;
403         if (i < q->ip_len) {
404             q->ip_len -= i;
405             q->ip_off += i;
406             m_adj(dtom(q), i);
407             break;
408         }
409         q = q->ipf_next;
410         m_freem(dtom(q->ipf_prev));
411         ip_deq(q->ipf_prev);
412     }
```

ip_input.c

图10-24 函数ip_reass ：截断已有分组

ip_input.c

```
413   insert:
414     /*
415      * Stick new fragment in its place;
416      * check for complete reassembly.
417      */
418     ip_enq(ip, q->ipf_prev);
419     next = 0;
420     for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = q->ipf_next) {
421         if (q->ip_off != next)
422             return (0);
423         next += q->ip_len;
424     }
425     if (q->ipf_prev->ipf_mff & 1)
426         return (0);
```

ip_input.c

图10-25 函数ip_reass ：插入分组

ip_input.c

```
427     /*
428      * Reassembly is complete; concatenate fragments.
429      */
430     q = fp->ipq_next;
431     m = dtom(q);
432     t = m->m_next;
433     m->m_next = 0;
434     m_cat(m, t);
435     q = q->ipf_next;
436     while (q != (struct ipasfrag *) fp) {
437         t = dtom(q);
438         q = q->ipf_next;
439         m_cat(m, t);
440     }
```

ip_input.c

图10-26 函数ip_reass ：重装数据报

图10-27显示了一个有三个分片的数据报的mbuf和ipq结构之间的关系。

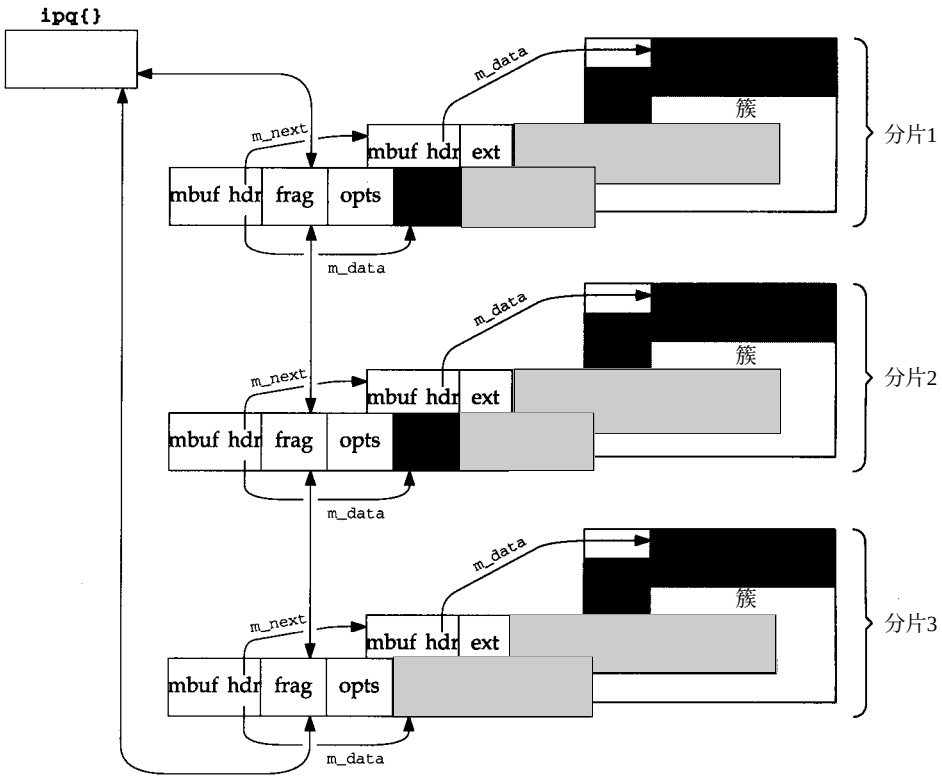


图10-27 m_cat 重装缓存内的分片

图中最暗的区域是分组的数据部分，稍淡的阴影部分是 mbuf中未用的部分。有三个分片，每个分片都被存放在一个有两个 mbuf的链上：一个分组首部和一个簇。每个分片的第一个缓存上的m_data指针指向分组数据，而不是分组的首部。因此，由 m_cat构造的缓存链只包含分片的数据部分。

当一个分片含有多于208字节的数据时，情况通常是这样的(2.6节)。缓存的“ frag ”部分是分片的IP首部。由于图10-18中的代码，各缓存链第一个缓存的m_data指针指向“ opts ”之后。

图10-28显示了用所有分片的缓存重装的数据报。注意，分片 2和3的IP部分和选项不在重装的数据报里。

第一个分片的首部仍然被用作 ipasfrag结构。它被图10-29中的代码恢复成一个有效的IP数据报首部。

4. 重建数据报首部

441-456 ip_reass把ip指向链表的第一个分片，将ipasfrag结构恢复成ip结构：把数据报长度恢复成 ip_len，源站地址恢复成 ip_src，目的地址恢复成 ip_dst；并把ipf_mff的低位清零(从图10-14可以知道， ipasfrag结构的ipf_mff覆盖了ip结构的ipf_tos)。

ip_reass用remque把整个分组从重装链表中移走，丢弃链表头 ipq结构，调整第一个缓存中的m_len和m_data，将前面被隐藏起来的第一个分片的首部和选项包含进来。

5. 计算分组长度

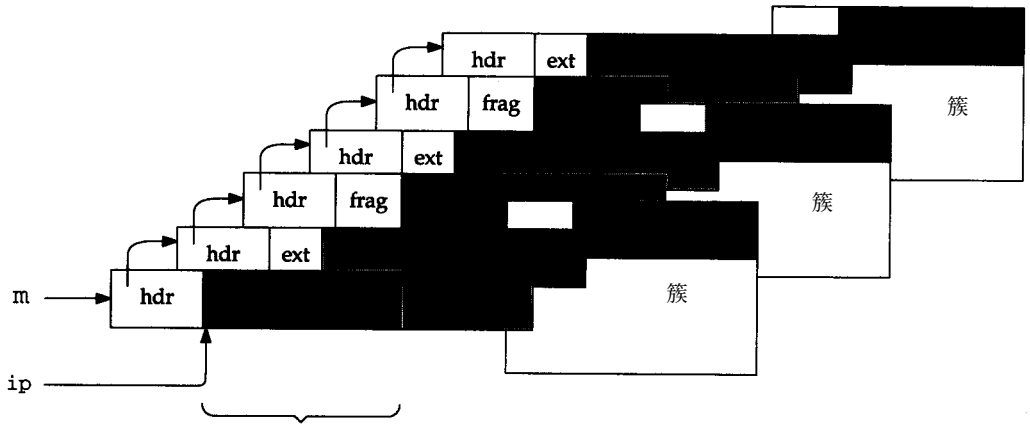


图10-28 重装的数据报

ip_input.c

```
441     /*
442      * Create header for new ip packet by
443      * modifying header of first packet;
444      * dequeue and discard fragment reassembly header.
445      * Make header visible.
446      */
447     ip = fp->ipq_next;
448     ip->ip_len = next;
449     ip->ipf_mff &= ~1;
450     ((struct ip *) ip)->ip_src = fp->ipq_src;
451     ((struct ip *) ip)->ip_dst = fp->ipq_dst;
452     remque(fp);
453     (void) m_free(dtom(fp));
454     m = dtom(ip);
455     m->m_len += (ip->ip_hl << 2);
456     m->m_data -= (ip->ip_hl << 2);
457     /* some debugging cruft by sklower, below, will go away soon */
458     if (m->m_flags & M_PKTHDR) {    /* XXX this should be done elsewhere */
459         int     plen = 0;
460         for (t = m; m; m = m->m_next)
461             plen += m->m_len;
462         t->m_pkthdr.len = plen;
463     }
464     return ((struct ip *) ip);
```

ip_input.c

图10-29 函数ip_reass ：数据报重装

457-464 此处的代码总被执行，因为数据报的第一个缓存总是一个分组首部。 for循环计算缓存链中数据的字节数，并把值保存在m_pkthdr.len中。

在选项类型字段中，*copied*比特的意义现在应该很明白了。因为目的主机只保留那些出现在第一个分片中的选项，而且只有那些在分组去往目的主机的途中控制分组处理的选项才被复制下来。不复制那些在传送过程中收集信息的选项，因为当分组在目的主机上被重装时，所有收集的信息都被丢弃了。

10.7 ip_slowtimo函数

如7.4节所述，Net/3的各项协议可能指定每500 ms调用一个函数。对IP而言，这个函数是ip_slowtimo，如图10-30所示，为重装链表上的分片计时。

ip_input.c

```
515 void
516 ip_slowtimo(void)
517 {
518     struct ipq *fp;
519     int     s = splnet();
520     fp = ipq.next;
521     if (fp == 0) {
522         splx(s);
523         return;
524     }
525     while (fp != &ipq) {
526         --fp->ipq_ttl;
527         fp = fp->next;
528         if (fp->prev->ipq_ttl == 0) {
529             ipstat.ips_fragtimeout++;
530             ip_freef(fp->prev);
531         }
532     }
533     splx(s);
534 }
```

ip_input.c

图10-30 ip_slowtimo 函数

515-534 ip_slowtimo遍历部分数据报的链表，减少重装TTL字段。当该字段减为0时，就调用ip_freef，把与该数据报相关的分片都丢弃。在splnet处运行ip_slowtimo，避免到达分组修改链表。

ip_freef显示如图10-31。

470-486 ip_freef移走并释放链表上fp指向的各分片，然后释放链表本身。

ip_input.c

```
474 void
475 ip_freef(fp)
476 struct ipq *fp;
477 {
478     struct ipasfrag *q, *p;
479     for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = p) {
480         p = q->ipf_next;
481         ip_deq(q);
482         m_freem(dtom(q));
483     }
484     remque(fp);
485     (void) m_free(dtom(fp));
486 }
```

ip_input.c

图10-31 ip_freef 函数

ip_drain函数

在图7-14中，我们讲到IP把ip_drain定义成一个当内核需要更多内存时才调用的函数。

这种情况通常发生在我们讨论过的(图2-13)分配缓存时。ip_drain显示如图10-32。

ip_input.c

```
538 void
539 ip_drain()
540 {
541     while (ipq.next != &ipq) {
542         ipstat.ips_fragdropped++;
543         ip_freef(ipq.next);
544     }
545 }
```

ip_input.c

图10-32 ip_drain 函数

538-545 IP释放内存的最简单办法就是丢弃重装链表上的所有IP分片。对属于某个TCP报文段的分片，TCP最终会重传该数据。属于UDP数据报的IP分片就丢失了，基于UDP的协议必须在应用程序层处理这种情况。

10.8 小结

本章展示了当一个外出的数据报过大而不适于在选定网络上传送时，ip_output如何对数据报分片。由于分片在向目的地传送的途中可能会被继续分片，也有可能走不同的路径，所以只有目的主机才能组装原来的数据报。

ip_reass接收到达分片，并试图重装数据报。如果重装成功，就把数据报传回ipintr，然后提交给恰当的运输层协议。所有IP实现必须能够重装最多576字节的数据报。Net/3的唯一限制就是可以得到的mbuf的个数。如果在一段合理的时间内，没有接收完数据报的所有分片，ip_slowtimo就丢弃不完整的数据报。

习题

10.1 修改ip_slowtimo，当它丢弃一个不完整数据报时(图11-1)，发出一个ICMP超时差错报文。

10.2 在分片的数据报中，各分片记录的路由可能互不相同。在目的主机上重装某个数据报时，返回给运输层的哪一个路由？

10.3 画一个图说明图10-17中ID为7的分片的ipq结构所涉及的mbuf和相关的分片链表。

10.4 [Auerbach 1994] 建议在对数据报分片后，应该首先传送最后的分片。如果接收系统先收到最后的分片，它就可以利用偏移值为数据报分配一个大小合适的缓冲区。请修改ip_output，首先发送最后的分片。

[Auerbach 1994] 注意到某些商用TCP/IP实现如果先收到最后的分片，就会出现崩溃。

10.5 用图8-5中的统计回答下面的问题。什么是每个重装的数据报的平均分片数？当一个外出的数据报被分片时，创建的平均分片数是多少？

10.6 如果ip_off的保留比特被置位，分组会发生什么情况？

第11章　ICMP：Internet控制报文协议

11.1 引言

ICMP在IP系统间传递差错和管理报文，是任何IP实现必需和要求的组成部分。ICMP的规范见RFC 792 [Postel 1981b]。RFC 950 [Mogul和Postel 1985]和RFC 1256 [Deering 1991a]定义了更多的ICMP报文类型。RFC 792 [Braden 1989a]提供了重要的ICMP细节。

ICMP有自己的传输协议号(1)，允许ICMP报文在IP数据报内携带。应用程序可以直接从第32章讨论的原始IP接口发送或接收ICMP报文。

我们可把ICMP报文分成两类：差错和查询。查询报文是用一对请求和回答定义的。ICMP差错报文通常包含了引起错误的IP数据报的第一个分片的IP首部(和选项)，加上该分片数据部分的前8个字节。标准假定这8个字节包含了该分组运输层首部的所有分用信息，这样运输层协议可以向正确的进程提交ICMP差错报文。

TCP和UDP端口号在它们首部的前8个字节内出现。

图11-1显示了所有目前定义的ICMP报文。双线上面的是ICMP请求和回答报文；双线下面的是ICMP差错报文。

type和code	描　述	PRC_
ICMP_ECHO	回显请求	
ICMP_ECHOREPLY	回显回答	
ICMP_TSAMP	时间戳请求	
ICMP_TSTAMPREPLY	时间戳回答	
ICMP_MASKREQ	地址掩码请求	
ICMP_MASKREPLY	地址掩码回答	
ICMP_IREQ	信息请求(过时的)	
ICMP_IREQREPLY	信息回答(过时的)	
ICMP_ROUTERADVERT	路由器通告	
ICMP_ROUTESOLICIT	路由器请求	
ICMP_REDIRECT	有更好的路由	
ICMP_REDIRECT_NET	网络有更好的路由	PRC_REDIRECT_HOST
ICMP_REDIRECT_HOST	主机有更好的路由	PRC_REDIRECT_HOST
ICMP_REDIRECT_TOSNET	TOS和网络有更好的路由	PRC_REDIRECT_HOST
ICMP_REDIRECT_TOSHOST	TOS和主机有更好的路由	PRC_REDIRECT_HOST
其他	不识别码	
ICMP_UNREACH	目的主机不可达	
ICMP_UNREACH_NET	网络不可达	PRC_UNREACH_NET
ICMP_UNREACH_HOST	主机不可达	PRC_UNREACH_HOST

图11-1　ICMP报文类型和代码

type和code	描　述	PRC_
ICMP_UNREACH_PROTOCOL	目的主机上协议不能用	PRC_UNREACH_PROTOCOL
ICMP_UNREACH_PORT	目的主机上端口没有被激活	PRC_UNREACH_PORT
ICMP_UNREACH_SRCFAIL	源路由失败	PRC_UNREACH_SRCFAIL
ICMP_UNREACH_NEEDFRAG	需要分片并设置DF比特	PRC_MSGSIZE
ICMP_UNREACH_NET_UNKNOWN	目的网络未知	PRC_UNREACH_NET
ICMP_UNREACH_HOST_UNKNOWN	目的主机未知	PRC_UNREACH_HOST
ICMP_UNREACH_ISOLATED	源主机被隔离	PRC_UNREACH_HOST
ICMP_UNREACH_NET_PROHIB	从管理上禁止与目的网络通信	PRC_UNREACH_NET
ICMP_UNREACH_HOST_PROHIB	从管理上禁止与目的主机通信	PRC_UNREACH_HOST
ICMP_UNREACH_TOSNET	对服务类型，网络不可达	PRC_UNREACH_NET
ICMP_UNREACH_TOSHOST	对服务类型，主机不可达	PRC_UNREACH_HOST
13	用过滤从管理上禁止通信	
14	主机优先违规	
15	事实上优先切断	
其他	不识别码	
ICMP_TIMXCEED	超时	
ICMP_TIMXCEED_INTRANS	传送过程中IP生存期到期	PRC_TIMXCEED_INTRANS
ICMP_TIMXCEED_REASS	重装生存期到期	PRC_TIMXCEED_REASS
其他	不识别码	
ICMP_PRRAMPROB	IP首部的问题	
0	未指明首部差错	PRC_PARAMPROB
ICMP_PRRAMPROB_OPTABSENT	丢失需要的选项	PRC_PARAMPROB
其他	无效字节的字节偏移	
ICMP_SOURCEQUENCH	要求放慢发送	PRC_QUENCH
其他	不识别类型	

图11-1 (续)

图11-1和图11-2中含有大量信息：

- PRC_栏显示了Net/3处理的与协议无关的差错码(11.6节)和ICMP报文之间的映射。对请求和回答，这一列是空的。因为在这种情况下不会产生差错。如果对一个 ICMP差错，这一行为空，说明Net/3不识别该码，并自动丢弃该差错报文。
- 图11-3显示了我们讨论图11-2所列函数的位置。
- icmp_input栏是icmp_input为每个ICMP报文调用的函数。
- UDP栏是为UDP插口处理ICMP报文的函数。
- TCP栏是为TCP插口处理ICMP报文的函数。注意，是tcp_quench处理ICMP源站抑制差错，而不是tcp_notify。
- 如果errno栏为空，内核不向进程报告ICMP报文。
- 表的最后一行显示，在用于接收 ICMP报文的进程的接收点上，不识别的 ICMP报文被提交给原来的IP协议。

在Net/3中，ICMP是作为IP之上的一个运输层协议实现的，它不产生差错或请求；它代表

其他协议格式化并发送报文。ICMP传递到达的差错，并向适当的传输协议或等待ICMP报文的进程发出回答。另一方面，ICMP用一个合适的ICMP回答响应大多数ICMP请求。图11-4对此作了总结。

type 和 code	icmp_input	UDP	TCP	errno
ICMP_ECHO	icmp_reflect			
ICMP_ECHOREPLY	rip_input			
ICMP_TSTAMP	icmp_reflect			
ICMP_TSTAMPREPLY	rip_input			
ICMP_MASKREQ	icmp_reflect			
ICMP_MASKREPLY	rip_input			
ICMP_IREQ	rip_input			
ICMP_IREQREPLY	rip_input			
ICMP_ROUTERADVERT	rip_input			
ICMP_ROUTERSOLICIT	rip_input			
ICMP_REDIRECT				
ICMP_REDIRECT_NET	pfctlinput	in_rtchange	in_rtchange	
ICMP_REDIRECT_HOST	pfctlinput	in_rtchange	in_rtchange	
ICMP_REDIRECT_TOSNET	pfctlinput	in_rtchange	in_rtchange	
ICMP_REDIRECT_TOSHOST	pfctlinput	in_rtchange	in_rtchange	
其他	rip_input			
ICMP_UNREACH				
ICMP_UNREACH_NET	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_HOST	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_PROTOCOL	pr_ctlinput	udp_notify	tcp_notify	ECONNREFUSED
ICMP_UNREACH_PORT	pr_ctlinput	udp_notify	tcp_notify	ECONNREFUSED
ICMP_UNREACH_SRCFAIL	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_NEEDFRAG	pr_ctlinput	udp_notify	tcp_notify	EMSGSIZE
ICMP_UNREACH_NET_UNKNOWN	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_HOST_UNKNOWN	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_ISOLATED	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_NET_PROHIB	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_HOST_PROHIB	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_TOSNET	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_TOSHOST	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
13	rip_input			
14	rip_input			
15	rip_input			
其他	rip_input			
ICMP_TIMXCEED				
ICMP_TIMXCEED_INTRANS	pr_ctlinput	udp_notify	tcp_notify	
ICMP_TIMXCEED_REASS	pr_ctlinput	udp_notify	tcp_notify	
其他	rip_input			
ICMP_PARAMPROB				
0	pr_ctlinput	udp_notify	tcp_notify	ENOPROTOOPT
ICMP_PARAMPROB_OPTABSENT	pr_ctlinput	udp_notify	tcp_notify	ENOPROTOOPT
其他	rip_input			
ICMP_SOURCEQUENCH	pr_ctlinput	udp_notify	tcp_quench	
其他	rip_input			

图11-2 ICMP报文类型和代码(续)

函　数	描　述	引　用
icmp_reflect	为ICMP生成回答	11.12节
in_rtchange	更新IP路由表	图22-34
pfctlinput	向所有协议报告差错	7.7节
pr_ctlinput	向与插口有关的协议报告差错	7.4节
rip_input	进程不识别的ICMP报文	32.5节
tcp_notify	向进程报告差错或忽略	图27-12
tcp_quench	放慢输出	图27-13
udp_notify	向进程报告差错	图23-31

图11-3　ICMP输入处理时调用的函数

ICMP报文类型	到　达	输　出
请求	向ICMP请求生成回答	由某个进程生成
回答	传给原始IP	由内核生成
差错	传给传输协议和原始IP	由IP或传输协议生成
未知	传给原始IP	由某个进程生成

图11-4　ICMP报文处理

11.2　代码介绍

图11-5的两个文件中有本章讨论的ICMP数据结构、统计量和处理的程序。

文　件	描　述
netinet/ip_icmp.h	ICMP结构定义
netinet/ip_icmp.c	ICMP处理

图11-5　本章定义的文件

11.2.1　全局变量

本章介绍的全局变量如图11-6所示。

变　量	类　型	描　述
icmpmaskrepl	int	使ICMP地址掩码回答的返回有效
icmpstat	struct icmpstat	ICMP统计量(图11-7)

图11-6　本章介绍的全局变量

11.2.2　统计量

统计量是由图11-7所示的icmpstat结构的成员收集的。

icmpstat成员	描　述	SNMP使用的
icps_oldicmp	因为数据报是一个ICMP报文而丢弃的差错数	•
icps_oldshort	因为IP数据报太短而丢弃的差错数	•

图11-7　本章收集的统计信息

icmpstat成员	描　述	SNMP使用的
icps_badcode	由于无效码而丢弃的ICMP报文数	•
icps_badlen	由于无效的ICMP体而丢弃的ICMP报文数	•
icps_checksum	由于坏的ICMP检验和而丢弃的ICMP报文数	•
icps_tooshort	由于ICMP首部太短而丢弃的报文数	•
icps_outhist[]	输出计数器数组;每种ICMP类型对应一个	•
icps_inhist[]	输入计数器数组;每种ICMP类型对应一个	•
icps_error	icmp_error的调用(重定向除外)数	
icps_reflect	内核反映的ICMP报文数	

图11-7 (续)

在分析程序时，我们会看到计数器是递增的。

图11-8显示的是执行netstat -s命令输出的统计信息的例子。

netstat -s 输出	icmpstat 成员
84124 calls to icmp_error	icps_error
0 errors not generated 'cuz old message was icmp	icps_oldicmp
Output histogram:	icps_outhist[]
echo reply: 11770	ICMP_ECHOREPLY
destination unreachable: 84118	ICMP_UNREACH
time exceeded: 6	ICMP_TIMXCEED
6 messages with bad code fields	icps_badcode
0 messages < minimum length	icps_badlen
0 bad checksums	icps_checksum
143 messages with bad length	icps_tooshort
Input histogram:	icps_inhist[]
echo reply: 793	ICMP_ECHOREPLY
destination unreachable: 305869	ICMP_UNREACH
source quench: 621	ICMP_SOURCEQUENCH
routing redirect: 103	ICMP_REDIRECT
echo: 11770	ICMP_ECHO
time exceeded: 25296	ICMP_TIMXCEED
11770 message responses generated	icps_reflect

图11-8 ICMP统计信息示例

11.2.3 SNMP变量

图11-9显示了SNMP ICMP组的变量与Net/3收集的统计量之间的关系。

SNMP变量	icmpstat成员	描　述
icmpInMsgs	见正文	
icmpInErrors	icps_badcode + icps_badlen + icps_checksum + icps_tooshort	收到的ICMP报文数 由于错误丢弃的ICMP报文数
icmpInDestUnreachs icmpInTimeExcds icmpInParmProbs icmpInSrcQuenchs icmpInRedirects icmpInEchos	icps_inhist[]计数器	每一类收到的ICMP报文数

图11-9 ICMP组内的简单SNMP变量

icmpInEchoReps icmpInTimestamps icmpInTimestampReps icmpInAddrMasks icmpInAddrMaskReps		
icmpOutMsgs icmpOutErrors	见正文 icps_oldicmp + icps_oldshort	发送的ICMP报文数 由于一个错误而没有发送的ICMP错误数
icmpOutDestUnreachs icmpOutTimeExcds icmpOutParmProbs icmpOutSrcQuenchs icmpOutRedirects icmpOutEchos icmpOutEchoReps icmpOutTimestamps icmpOutTimestampReps icmpOutAddrMasks icmpOutAddrMaskReps	icps_outhist[]计数器	每一类发送的ICMP报文数

图11-9 (续)

icmpInMsgs是icps_inhist数组和icmpInErrors中的计数之和，icmpOutMsgs是icps_outist数组和icmpOutErrors中的计数之和。

11.3 icmp结构

Net/3通过图11-10中的icmp结构访问某个ICMP报文。

42-45 icmp_type标识特定报文，icmp_code进一步指定该报文(图11-1的第1栏)。计算icmp_cksum的算法与IP首部检验和相同，保护整个ICMP 报文(像IP一样，不仅仅保护首部)。

46-79 联合icmp_hun(首部联合)和icmp_dun(数据联合)按照icmp_type和icmp_code访问多种ICMP报文。每种ICMP报文都使用icmp_hun；只有一部分报文用 icmp_dun。没有使用的字段必须设置为0。

80-86 我们已经看到，利用其他嵌套的结构(例如 mbuf、le_softc和ether_arp)，#define宏可以简化对结构成员的访问。

图11-11显示了ICMP报文的整体结构，并再次强调 ICMP报文是封装在IP数据报里的。我们将在分析程序时，分析所遇报文的特定结构。

ip_icmp.h

```
42 struct icmp {
43     u_char  icmp_type;          /* type of message, see below */
44     u_char  icmp_code;          /* type sub code */
45     u_short icmp_cksum;         /* ones complement cksum of struct */
46     union {
47         u_char  ih_pptr;        /* ICMP_PARAMPROB */
48         struct in_addr ih_gwaddr;   /* ICMP_REDIRECT */
49         struct ih_idseq {
50             n_short icd_id;
51             n_short icd_seq;
52         } ih_idseq;
53         int     ih_void;

54         /* ICMP_UNREACH_NEEDFRAG -- Path MTU Discovery (RFC1191) */
55         struct ih_pmtu {
```

图11-10 icmp 结构

```
56              n_short ipm_void;
57              n_short ipm_nextmtu;
58          } ih_pmtu;
59      } icmp_hun;
60  #define icmp_pptr    icmp_hun.ih_pptr
61  #define icmp_gwaddr  icmp_hun.ih_gwaddr
62  #define icmp_id      icmp_hun.ih_idseq.icd_id
63  #define icmp_seq     icmp_hun.ih_idseq.icd_seq
64  #define icmp_void    icmp_hun.ih_void
65  #define icmp_pmvoid  icmp_hun.ih_pmtu.ipm_void
66  #define icmp_nextmtu     icmp_hun.ih_pmtu.ipm_nextmtu
67      union {
68          struct id_ts {
69              n_time  its_otime;
70              n_time  its_rtime;
71              n_time  its_ttime;
72          } id_ts;
73          struct id_ip {
74              struct ip idi_ip;
75              /* options and then 64 bits of data */
76          } id_ip;
77          u_long  id_mask;
78          char    id_data[1];
79      } icmp_dun;
80  #define icmp_otime   icmp_dun.id_ts.its_otime
81  #define icmp_rtime   icmp_dun.id_ts.its_rtime
82  #define icmp_ttime   icmp_dun.id_ts.its_ttime
83  #define icmp_ip      icmp_dun.id_ip.idi_ip
84  #define icmp_mask    icmp_dun.id_mask
85  #define icmp_data    icmp_dun.id_data
86  };
```

ip_icmp.h

图11-10 (续)

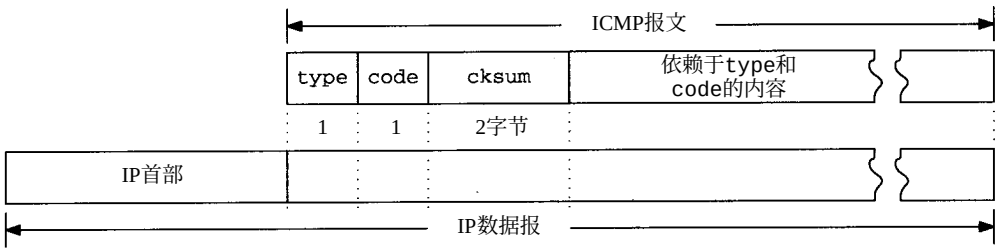


图11-11 一个ICMP报文(省略icmp_)

11.4 ICMP 的protosw结构

inetsw[4](图11-13)的protosw结构描述了ICMP，并支持内核和进程对协议的访问。图11-12显示了该结构。在内核里，icmp_input处理到达的ICMP报文，进程产生的外出ICMP报文由rip_output处理。以rip_开头的三个函数将在第32章中讨论。

成 员	inetsw[4]	描 述
pr_type	*SOCK_RAW*	ICMP提供原始分组服务
pr_domain	*&inetdomain*	ICMP是Internet域的一部分

图11-12 ICMP 的inetsw 项

成 员	inetsw[4]	描 述
pr_protocol	*IPPROTO_ICMP (1)*	出现在IP首部的ip_p字段中
pr_flags	*PR_ATOMIC|PR_ADDR*	插口层标志，ICMP不使用
pr_input	*icmp_input*	从IP层接收ICMP报文
pr_output	*rip_output*	将ICMP报文发送到IP层
pr_ctlinput	*0*	ICMP不使用
pr_ctloutput	*rip_ctloutput*	响应来自一个进程的管理请求
pr_usrreq	*rip_usrreq*	响应来自一个进程的通信请求
pr_init	*0*	ICMP不使用
pr_fasttimo	*0*	ICMP不使用
pr_slowtimo	*0*	ICMP不使用
pr_drain	*0*	ICMP不使用
pr_sysctl	*0*	ICMP不使用

图11-12 (续)

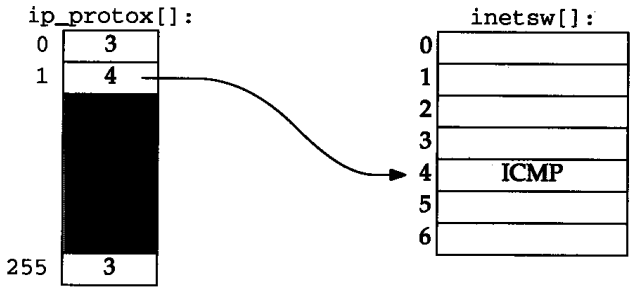


图11-13 数值为1的ip_p 选择了inetsw[4]

11.5 输入处理：icmp_input函数

回想起ipintr对数据报进行分用是根据IP首部中的传输协议编号ip_p。对于ICMP报文，ip_p是1，并通过ip_protox选择inetsw[4]。

当一个ICMP报文到达时，IP层通过inetsw[4]的pr_input函数，间接调用icmp_input(图10-11)。

我们将看到，在icmp_input中，每一个ICMP报文要被处理3次：被icmp_input处理一次；被与ICMP差错报文中的IP分组相关联的传输协议处理一次；被记录收到ICMP报文的进程处理一次。ICMP输入处理过程的总的构成情况如图11-14所示。

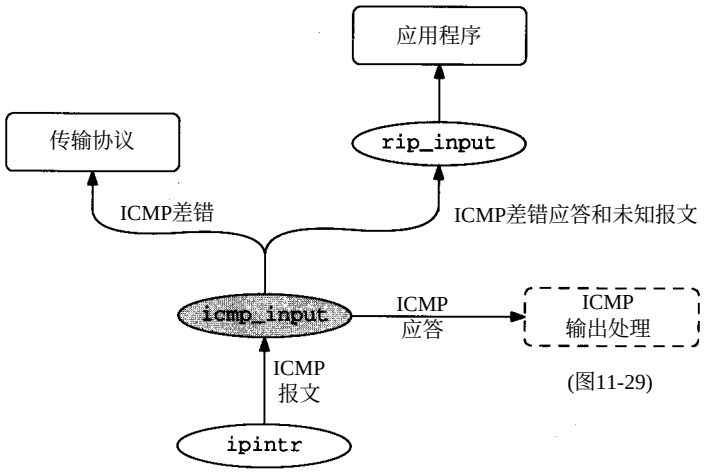


图11-14 ICMP的输入处理过程

我们将在以下5节(11.6～11.10)讨论icmp_input：(1) 验证收到的报文；(2) ICMP差错报文；(3) ICMP请求报文；(4) ICMP重定向报文；(5) ICMP回答报文。icmp_input函数的第一部分如图11-15所示。

ip_icmp.c

```
131 static struct sockaddr_in icmpsrc = { sizeof (struct sockaddr_in), AF_INET };
132 static struct sockaddr_in icmpdst = { sizeof (struct sockaddr_in), AF_INET };
133 static struct sockaddr_in icmpgw = { sizeof (struct sockaddr_in), AF_INET };
134 struct sockaddr_in icmpmask = { 8, 0 };

135 void
136 icmp_input(m, hlen)
137 struct mbuf *m;
138 int     hlen;
139 {
140     struct icmp *icp;
141     struct ip *ip = mtod(m, struct ip *);
142     int     icmplen = ip->ip_len;
143     int     i;
144     struct in_ifaddr *ia;
145     void    (*ctlfunc) (int, struct sockaddr *, struct ip *);
146     int     code;
147     extern u_char ip_protox[];

148     /*
149      * Locate icmp structure in mbuf, and check
150      * that not corrupted and of at least minimum length.
151      */
152     if (icmplen < ICMP_MINLEN) {
153         icmpstat.icps_tooshort++;
154         goto freeit;
155     }
156     i = hlen + min(icmplen, ICMP_ADVLENMIN);
157     if (m->m_len < i && (m = m_pullup(m, i)) == 0) {
158         icmpstat.icps_tooshort++;
159         return;
160     }
161     ip = mtod(m, struct ip *);
162     m->m_len -= hlen;
163     m->m_data += hlen;
164     icp = mtod(m, struct icmp *);
165     if (in_cksum(m, icmplen)) {
166         icmpstat.icps_checksum++;
167         goto freeit;
168     }
169     m->m_len += hlen;
170     m->m_data -= hlen;

171     if (icp->icmp_type > ICMP_MAXTYPE)
172         goto raw;
173     icmpstat.icps_inhist[icp->icmp_type]++;
174     code = icp->icmp_code;
175     switch (icp->icmp_type) {
```

图11-15 icmp_input 函数

```
317     default:
318         break;
319     }
320   raw:
321     rip_input(m);
322     return;

323   freeit:
324     m_freem(m);
325 }
```

ip_icmp.c

图11-15 (续)

1. 静态结构

131-134 因为icmp_input是在中断时调用的，此时堆栈的大小是有限的。所以，为了在每次调用icmp_input时，避免动态分配造成的延迟，以及使堆栈最小，这4个结构是动态分配的。icmp_input把这4个结构用作临时变量。

icmpsrc的命名容易引起误解，因为icmp_input把它用作临时sockaddr_in变量，而它也从未包含过源站地址。在Net/2版本的icmp_input中，在报文被raw_input函数提交给原始IP之前，报文的源站地址在函数的最后被复制到icmpsrc中。而Net/3调用只需要一个指向该分组的指针的rip_input，而不是raw_input。虽然有这个改变，但是icmpsrc仍然保留了在Net/2中的名字。

2. 确认报文

135-139 icmp_input希望收到的ICMP报文(m)中含有一个指向该数据报的指针，以及该数据报IP首部的字节长度(hlen)。图11-16列出了几个在icmp_input里用于简化检测无效ICMP报文的常量。

常量／宏	值	描　述
ICMP_MINLEN	8	ICMP报文大小的最小值
ICMP_TSLEN	20	ICMP时间戳报文大小
ICMP_MASKLEN	12	ICMP地址掩码报文大小
ICMP_ADVLENMIN	36	ICMP差错(建议)报文大小的最小值 (*IP* + *ICMP* + *BADIP* = 20 + 8 + 8 = 36)
ICMP_ADVLEN(p)	36 + *optsize*	ICMP差错报文的大小，包含无效分组p的IP选项的*optsize*字节

图11-16 ICMP引用的用来验证报文的常量

140-160 icmp_input从ip_len取出ICMP报文的大小，并把它存放在icmplen中。第8章讲过，ipintr从ip_len中排除了IP首部的长度。如果报文长度太短，不是有效报文，就生成icps_tooshort，并丢弃该报文。如果在第一个mbuf中，ICMP首部和IP首部不是连续的，则由m_pullup保证ICMP首部以及封闭的IP分组的IP首部在同一个mbuf中。

3. 验证检验和

161-170 icmp_input隐藏mbuf中的IP首部，并用in_cksum验证ICMP的检验和。如果报文被破坏，就增加icps_checksum，并丢弃该报文。

4. 验证类型

171-175 如果报文类型(icmp_type)不在识别范围内，icmp_input就跳过switch执行

raw语句(图11-9)。如果在识别范围内，icmp_input复制icmp_code，switch按照icmp_type处理该报文。

在ICMP switch语句处理完后，icmp_input向rip_input发送ICMP报文，后者把ICMP报文发布给准备接收的进程。只有那些被破坏的报文(长度或检验和出错)以及只由内核处理的ICMP请求报文才不传给rip_input。在这两种情况下，icmp_input都立即返回，并跳过raw处的源程序。

5. 原始ICMP输入

317-325 icmp_input把到达的报文传给rip_input，rip_input依据报文里含有的协议及源站和目的站地址信息(32章)，把报文发布给正在监听的进程。

原始IP机制允许进程直接发送和接收ICMP报文，这样做有几个原因：

- 新ICMP报文可由进程处理而无需修改内核(例如，路由器通告，图11-28)。
- 可以用进程而无需用内核模块来实现发送ICMP请求和处理回答的机制(ping和traceroute)。
- 进程可以增加对报文的内核处理。与此类似，内核在更新完它的路由表后，会把ICMP重定向报文传给一个路由守护程序。

11.6 差错处理

我们首先考虑ICMP差错报文。当主机发出的数据报无法成功地提交给目的主机时，它就接收这些报文。目的主机或中间的路由器生成这些报文，并将它们返回到原来的系统。图11-17显示了多种ICMP差错报文的格式。

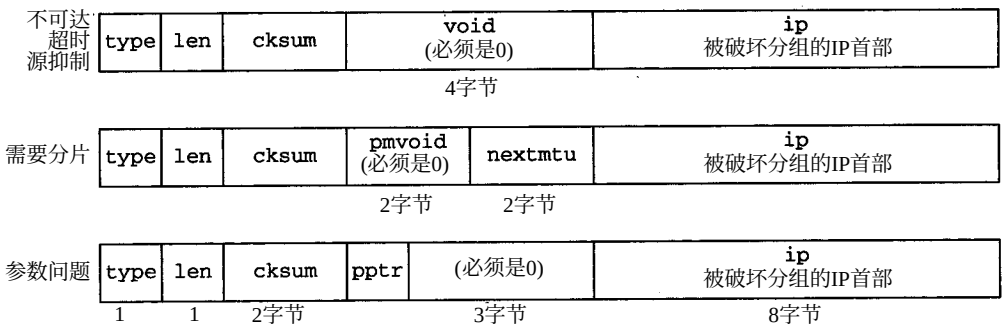


图11-17 ICMP差错报文(省略icmp_)

图11-18中的源程序来自图11-15中的switch语句。

ip_icmp.c

```
176     case ICMP_UNREACH:
177         switch (code) {
178         case ICMP_UNREACH_NET:
179         case ICMP_UNREACH_HOST:
180         case ICMP_UNREACH_PROTOCOL:
181         case ICMP_UNREACH_PORT:
182         case ICMP_UNREACH_SRCFAIL:
183             code += PRC_UNREACH_NET;
184             break;
185         case ICMP_UNREACH_NEEDFRAG:
```

图11-18 icmp_input 函数：差错报文

```
186             code = PRC_MSGSIZE;
187             break;

188         case ICMP_UNREACH_NET_UNKNOWN:
189         case ICMP_UNREACH_NET_PROHIB:
190         case ICMP_UNREACH_TOSNET:
191             code = PRC_UNREACH_NET;
192             break;

193         case ICMP_UNREACH_HOST_UNKNOWN:
194         case ICMP_UNREACH_ISOLATED:
195         case ICMP_UNREACH_HOST_PROHIB:
196         case ICMP_UNREACH_TOSHOST:
197             code = PRC_UNREACH_HOST;
198             break;

199         default:
200             goto badcode;
201         }
202         goto deliver;

203     case ICMP_TIMXCEED:
204         if (code > 1)
205             goto badcode;
206         code += PRC_TIMXCEED_INTRANS;
207         goto deliver;
208     case ICMP_PARAMPROB:
209         if (code > 1)
210             goto badcode;
211         code = PRC_PARAMPROB;
212         goto deliver;

213     case ICMP_SOURCEQUENCH:
214         if (code)
215             goto badcode;
216         code = PRC_QUENCH;

217       deliver:
218         /*
219          * Problem with datagram; advise higher level routines.
220          */
221         if (icmplen < ICMP_ADVLENMIN || icmplen < ICMP_ADVLEN(icp) ||
222             icp->icmp_ip.ip_hl < (sizeof(struct ip) >> 2)) {
223             icmpstat.icps_badlen++;
224             goto freeit;
225         }
226         NTOHS(icp->icmp_ip.ip_len);
227         icmpsrc.sin_addr = icp->icmp_ip.ip_dst;
228         if (ctlfunc = inetsw[ip_protox[icp->icmp_ip.ip_p]].pr_ctlinput)
229             (*ctlfunc) (code, (struct sockaddr *) &icmpsrc,
230                         &icp->icmp_ip);
231         break;

232       badcode:
233         icmpstat.icps_badcode++;
234         break;
```

ip_icmp.c

图11-18 (续)

176-216 对ICMP差错的处理是最少的，因为这主要是运输层协议的责任。imcp_input把icmp_type和icmp_code映射到一个与协议无关的差错码集上，该差错码是由PRC_常量

(图11-19)表示的。PRC_常量有一个隐含的顺序，正好与ICMP的code相对应。这就解释了为什么code是按一个PRC_常量递增的。

217-225 如果识别出类型和码，icmp_input就跳到deliver。如果没有识别出来，icmp_input就跳到badcode。

如果对所报告的差错而言，报文长度不正确，icps_badlen的值就加1，并丢弃该报文。Net/3总是丢弃无效的ICMP报文，也不生成有关该无效报文的ICMP差错。这样，就避免在两个有缺陷的实现之间形成无限的差错报文序列。

226-231 icmp_input调用运输层协议的pr_ctlinput函数，该函数根据原始数据报的ip_p，把到达分组分用到正确的协议，从而构造出原始的IP数据报。差错码(code)、原始IP数据报的目的地址(icmpsrc)以及一个指向无效数据报的指针(icmp_ip)被传给pr_ctlinput(如果是为该协议定义的)。图23-31和图27-12讨论这些差错。

232-234 最后，icps_badcode的值增加1，并终止switch语句的执行。

常 量	描 述
PRC_HOSTDEAD	主机似乎已关闭
PRC_IFDOWN	网络接口关闭
PRC_MSGSIZE	无效报文大小
PRC_PRRAMPROB	首部不正确
PRC_QUENCH	某人说要放慢
PRC_QUENCH2	阻塞比特要求放慢
PRC_REDIRECT_HOST	主机路由选择重定向
PRC_REDIRECT_NET	网络路由选择重定向
PRC_REDIRECT_TOSHOST	TOS和主机的重定向
PRC_REDIRECT_TOSNET	TOS和网络的重定向
PRC_ROUTEDEAD	如果可能，选择新的路由
PRC_TIMXCEED_INTRANS	传送过程中分组生命期到期
PRC_TIMXCEED_REASS	分片在重装过程中生命期到期
PRC_UNREACH_HOST	没有到主机的路由
PRC_UNREACH_NET	没有到网络的路由
PRC_UNREACH_PORT	目的主机称端口未激活
PRC_UNREACH_PROTOCOL	目的主机称协议不可用
PRC_UNREACH_SRCFAIL	源路由失败

图11-19 与协议无关的差错码

尽管PRC_常量表面上与协议无关，但它们主要还是基于Internet协议族。其结果是，当某个Internet协议族以外的协议把自己的差错映射到PRC_常量时，会失去可指定性。

11.7 请求处理

Net/3响应具有正确格式的ICMP请求报文，但把无效ICMP请求报文传给rip_input。第32章讨论了应用程序如何生成ICMP请求报文。

除路由器通告报文外，大多数Net/3所接收的ICMP请求报文都生成回答报文。为避免为回答报文分配新的mbuf，icmp_input把请求的缓存转换成回答的缓存，并返回给发送方。我

们将分别讨论各个请求。

11.7.1 回显询问：ICMP_ECHO和ICMP_ECHOREPLY

尽管ICMP非常简单，但是ICMP回显请求和回答却是网络管理员最有力的诊断工具。发出ICMP回显请求称为“ ping ”一个主机，也就是调用 ping程序一次。许多系统提供该程序来手工发送ICMP回显请求。卷1的第7章详细讨论了ping。

ping程序的名字依照了声纳脉冲 (soar ping)，用其他物体对声纳脉冲的反射所产生的回声确定它们的位置。卷 1把这个名字解释成Packet InterNet Groper，是不正确的。

图11-20是ICMP回显请求和回答报文的结构。

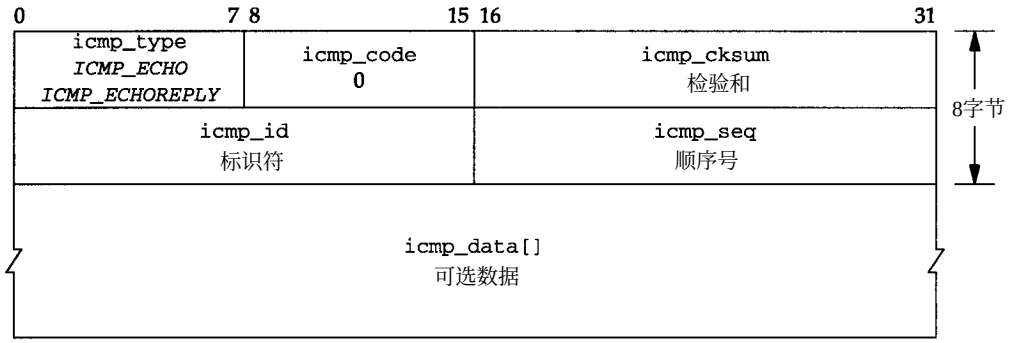


图11-20 ICMP回显请求和回答

icmp_code总是0。 icmp_id和icmp_seq设置成请求的发送方，回答中也不做修改。源系统可以用这些字段匹配请求和回答。 icmp_data中到达的所有数据也被反射。图 11-21是ICMP回显处理和icmp_input实现反射ICMP请求的源程序。

ip_icmp.c

```
235     case ICMP_ECHO:
236         icp->icmp_type = ICMP_ECHOREPLY;
237         goto reflect;

                    /* other ICMP request processing */

277       reflect:
278         ip->ip_len += hlen;     /* since ip_input deducts this */
279         icmpstat.icps_reflect++;
280         icmpstat.icps_outhist[icp->icmp_type]++;
281         icmp_reflect(m);
282         return;
```

ip_icmp.c

图11-21 icmp_input 函数：回显请求和回答

235-237 通过把icmp_type变成ICMP_ECHOREPLY，并跳转到reflect发送回答，icmp_input把回显请求转换成了回显回答。

277-282 在为每个ICMP 请求构造完回答之后，icmp_input执行reflect处的程序。在这里，存储数据报正确的长度被恢复，在 icps_reflect和icps_outhist[]中分别计算请求的数量和ICMP报文的类型。icps_reflect(11.12节)把回答发回给请求方。

11.7.2 时间戳询问：ICMP_TSTAMP和ICMP_TSTAMPREPLY

ICMP时间戳报文如图11-22所示。

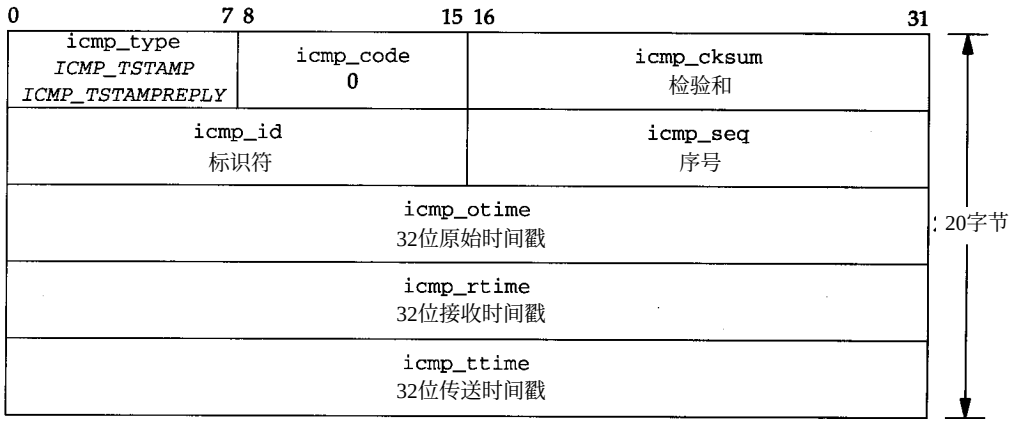


图11-22 ICMP时间戳请求和回答

icmp_code总是0。icmp_id和icmp_seq的作用与它们在ICMP回显报文中的一样。请求的发送方设置icmp_otime(发出请求的时间)；icmp_rtime(收到请求的时间)和icmp_ttime(发出回答的时间)由回答的发送方设置。所有时间都是从UTC午夜开始的毫秒数。如果时间值没有以标准单位记录，就把高位置位，与IP时间戳选项一样。

图11-23是实现时间戳报文的程序。

ip_icmp.c

```
238     case ICMP_TSTAMP:
239         if (icmplen < ICMP_TSLEN) {
240             icmpstat.icps_badlen++;
241             break;
242         }
243         icp->icmp_type = ICMP_TSTAMPREPLY;
244         icp->icmp_rtime = iptime();
245         icp->icmp_ttime = icp->icmp_rtime;  /* bogus, do later! */
246         goto reflect;
```

ip_icmp.c

图11-23 icmp_input 函数：时间戳请求和回答

238-246 icmp_input对ICMP的响应，包括：把icmp_type改成ICMP_TSTAMPREPLY，记录当前icmp_rtime和icmp_ttime，并跳转到reflect发送回答。

很难精确地设置icmp_rtime和icmp_ttime。当系统执行这段程序时，报文可能已经在IP输入队列中等待处理，这时设置icmp_rtime已经太晚了。类似地，数据报也可能在要求处理时在网络接口的传输队列中被延迟，这时设置icmp_ttime又太早了。为了把时间戳设置得更接近真实的接收和发送时间，必须修改每个网络的接口驱动程序，使其能理解ICMP报文(习题11.8)。

11.7.3 地址掩码询问：ICMP_MASKREQ和ICMP_MASKREPLY

ICMP 地址掩码请求和回答如图11-24所示。

RFC 950 [Mogul和Postel 1985]在原来的ICMP规范说明中增加了地址掩码报文，使系统能发现某个网络上使用的子网掩码。

除非系统被明确地配置成地址掩码的授权代理，否则， RFC 1122禁止向其发送掩码回答。这样，就避免系统与所有向它发出请求的系统共享不正确的地址掩码。如果没有管理员授权回答，系统也要忽略地址掩码请求。

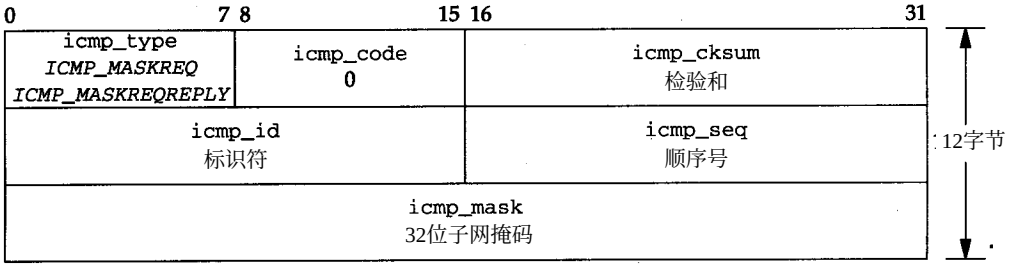


图11-24 ICMP地址掩码请求和回答

如果全局整数icmpmaskrepl非零， Net/3会响应地址掩码请求。 icmpmaskrepl的默认值是0， icmp_sysctl可以通过systctl(8)程序(11.14节)修改它。

Net/2系统中没有控制回答地址掩码请求的机制。其结果是，必须非常正确地配置Net/2接口的地址掩码；该信息是与网络上所有发出地址掩码请求的系统共享的。

地址掩码报文的处理如图11-25所示。

ip_icmp.c

```
247     case ICMP_MASKREQ:
248 #define satosin(sa) ((struct sockaddr_in *)(sa))
249         if (icmpmaskrepl == 0)
250             break;
251         /*
252          * We are not able to respond with all ones broadcast
253          * unless we receive it over a point-to-point interface.
254          */
255         if (icmplen < ICMP_MASKLEN)
256             break;
257         switch (ip->ip_dst.s_addr) {

258         case INADDR_BROADCAST:
259         case INADDR_ANY:
260             icmpdst.sin_addr = ip->ip_src;
261             break;

262         default:
263             icmpdst.sin_addr = ip->ip_dst;
264         }
265         ia = (struct in_ifaddr *) ifaof_ifpforaddr(
266                         (struct sockaddr *) &icmpdst, m->m_pkthdr.rcvif);
267         if (ia == 0)
268             break;
269         icp->icmp_type = ICMP_MASKREPLY;
270         icp->icmp_mask = ia->ia_sockmask.sin_addr.s_addr;
271         if (ip->ip_src.s_addr == 0) {
272             if (ia->ia_ifp->if_flags & IFF_BROADCAST)
273                 ip->ip_src = satosin(&ia->ia_broadaddr)->sin_addr;
274             else if (ia->ia_ifp->if_flags & IFF_POINTOPOINT)
275                 ip->ip_src = satosin(&ia->ia_dstaddr)->sin_addr;
276         }
```

ip_icmp.c

图11-25 icmp_input 函数：地址掩码请求和回答

247-256 如果没有配置响应掩码请求，或者该请求太短，这段程序就中止switch的执行，并把报文传给rip_input(图11-15)。

在这里Net/3无法增加icps_badlen。对其他ICMP长度差错，它却增加icps_badlen。

1. 选择子网掩码

257-267 如果地址掩码请求被发到0.0.0.0或255.255.255.255，源地址被保存在icmpdst中。在这里，ifaof_offoraddr把icmpdst作为源站地址，在同一网络上查找in_ofaddr结构。如果源站地址是0.0.0.0或255.255.255.255，ifaof_offoraddr返回一个指针，该指针指向与接收接口相关的第一个IP地址。

default情况(针对单播或有向广播)为ifaof_ifpforaddr保存目的地址。

2. 转换成回答

269-270 通过改变icmp_type，并把所选子网掩码ia_sockmask复制到icmp_mask，就完成了把请求转换成回答的工作。

3. 选择目的地址

271-276 如果请求的源站地址全0(“该网络上的这台主机”，只在引导时用作源站地址，RFC 1122)，并且源站不知道自己的地址，Net/3必须广播这个回答，使源站系统接收到这个报文。在这种情况下，如果接收接口位于某个广播或点到点网络上，该回答的目的地址将分别是ia_broadaddr和ia_dstaddr。icmp_input把回答的目的地址放在ip_src里，因为reflect处的程序(图11-21)会把源站和目的站地址倒过来。不改变单播请求的地址。

11.7.4 信息询问：ICMP_IREQ和ICMP_IREQREPLY

ICMP信息报文已经过时了。它们企图广播一个源和目的站地址字段的网络部分为全0的请求，使系统发现连接的IP网络的数量。响应该请求的主机将返回一个填好网络号的报文。主机还需要其他办法找到地址的主机部分。

RFC 1122推荐主机不要实现ICMP信息报文，因为RARP(RFC 903 [Finlayson et al., 1984])和BOOTP(RFC 951 [Croft和Gilmore 1985])更适于发现地址。RFC 1541 [Droms 1993]描述的一个新协议，动态主机配置协议(Dynamic Host Configuration Protocol，DHCP)，可能会取代或增强BOOTP的功能。它现在是一个建议的标准。

Net/2响应ICMP信息请求报文。但是，Net/3把它们传给rip_input。

11.7.5 路由器发现：ICMP_ROUTERADVERT和ICMP_ROUTERSOLICIT

RFC 1256定义了ICMP路由器发现报文。Net/3内核不直接处理这些报文，而由rip_input把它们传给一个用户级守护程序，由它发送和响应这种报文。

卷1的9.6节讨论了这种报文的设计和运行。

11.8 重定向处理

图11-26显示了ICMP重定向报文的格式。

icmp_input中要讨论的最后一个case是ICMP_REDIRECT。如8.5节的讨论，当分组

被发给错误的路由器时，产生重定向报文。该路由器把分组转发给正确的路由器，并发回一个ICMP重定向报文，系统把信息记入它自己的路由表。

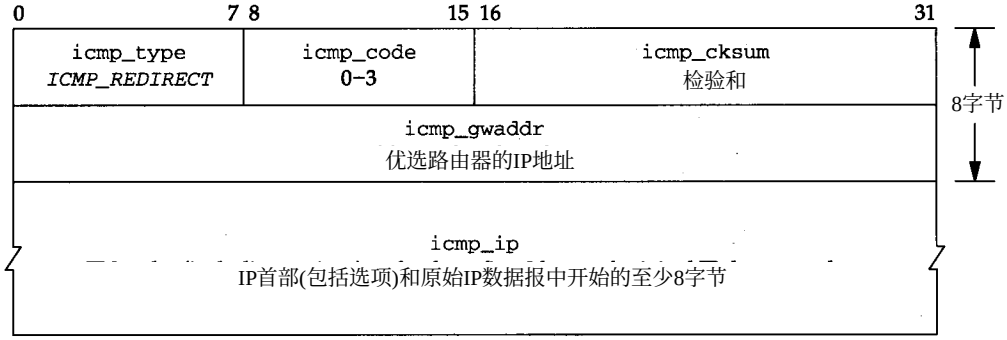


图11-26 ICMP重定向报文

图11-27显示了icmp_input用来处理重定向报文的程序。

ip_icmp.c

```
283     case ICMP_REDIRECT:
284         if (code > 3)
285             goto badcode;
286         if (icmplen < ICMP_ADVLENMIN || icmplen < ICMP_ADVLEN(icp) ||
287             icp->icmp_ip.ip_hl < (sizeof(struct ip) >> 2)) {
288             icmpstat.icps_badlen++;
289             break;
290         }
291         /*
292          * Short circuit routing redirects to force
293          * immediate change in the kernel's routing
294          * tables.  The message is also handed to anyone
295          * listening on a raw socket (e.g. the routing
296          * daemon for use in updating its tables).
297          */
298         icmpgw.sin_addr = ip->ip_src;
299         icmpdst.sin_addr = icp->icmp_gwaddr;
300         icmpsrc.sin_addr = icp->icmp_ip.ip_dst;
301         rtredirect((struct sockaddr *) &icmpsrc,
302                    (struct sockaddr *) &icmpdst,
303                    (struct sockaddr *) 0, RTF_GATEWAY | RTF_HOST,
304                    (struct sockaddr *) &icmpgw, (struct rtentry **) 0);
305         pfctlinput(PRC_REDIRECT_HOST, (struct sockaddr *) &icmpsrc);
306         break;
```

ip_icmp.c

图11-27 icmp_input 函数：重定向报文

1. 验证

283-290 如果重定向报文中含有未识别的ICMP 码，icmp_input就跳到badcode(图11-18的232行)；如果报文具有无效长度或封闭的IP分组具有无效首部长度，则中止switch。图11-16显示了ICMP差错报文的最小长度是36(ICMP_ADVLENMIN)。ICMP_ADVLEN(icp)是当icp所指向的分组有IP选项时，ICMP差错报文的最小长度。

291-300 icmp_input分别把重定向报文的源站地址(发送该报文的网关)、为原始分组推荐的路由器(第一跳目的地)和原始分组的最终目的地址分配给 icmpgw、icmpdst和icmpsrc。

这里，icmpsrc并不包含源站地址 —— 这是方便存放目的地址的位置，无需再定义一个sockaddr结构。

2. 更新路由

301-306 Net/3按照RFC 1122的推荐，等价地对待网络重定向和主机重定向。重定向信息被传给rtredirect，由这个函数更新路由表。重定向的目的地址(保存在 icmpsrc)被传给pfctlinput，由它通告重定向的所有协议域(7.3节)，使协议有机会把缓存的到目的站的路由作废。

按照RFC 1122，应该把网络重定向作为主机重定向对待，因为当目的网络划分了子网时，它们会提供不正确的路由信息。事实上， RFC 1009要求，在网络划分子网的情况下，不发送网络重定向。不幸的是，许多路由器违背了这个要求。 Net/3从不发重定向报文。

ICMP重定向报文是IP路由选择体系结构的基本组成部分。尽管被划分到差错报文类，但它却是在任何有多个路由器的网络正常运行时出现的。第 18章更详细讨论了IP路由选择问题。

11.9 回答处理

内核不处理任何ICMP回答报文。ICMP请求由进程产生，内核从不产生请求。所以，内核把它接收的所有回答传给等待 ICMP报文的进程。另外， ICMP路由器发现报文被传给rip_input。

ip_icmp.c

```
307         /*
308          * No kernel processing for the following;
309          * just fall through to send to raw listener.
310          */
311     case ICMP_ECHOREPLY:
312     case ICMP_ROUTERADVERT:
313     case ICMP_ROUTERSOLICIT:
314     case ICMP_TSTAMPREPLY:
315     case ICMP_IREQREPLY:
316     case ICMP_MASKREPLY:
317     default:
318         break;
319     }
320   raw:
321     rip_input(m);
322     return;
```

ip_icmp.c

图11-28 icmp_input 函数：回答报文

307-322 内核无需对ICMP回答报文做出任何反应，所以在raw处的switch语句后继续执行(图11-15)。注意，switch语句的default情况(未识别的ICMP报文)也把控制传给在raw处的代码。

11.10 输出处理

有几种方法产生外出的ICMP报文。第8章讲到IP调用icmp_error来产生和发送ICMP 差错报文。icmp_reflect发送ICMP 回答报文，同时，进程也可能通过原始 ICMP 协议生成ICMP 报文。图11-29显示了这些函数与ICMP外出处理之间的关系。

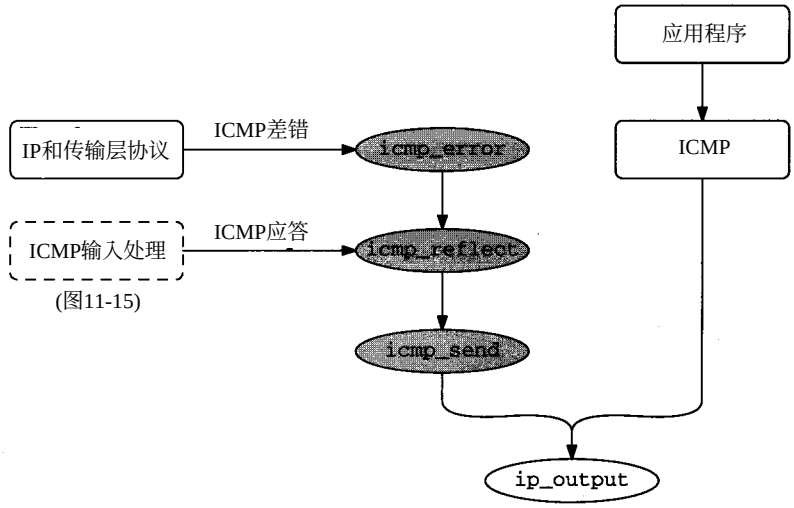


图11-29 ICMP外出处理

11.11 icmp_error函数

icmp_error在IP或运输层协议的请求下，构造一个ICMP差错请求报文，并把它传给icmp_reflect，在那里该报文被返回无效数据报的源站。我们分三部分分析这个函数：

ip_icmp.c

```
46 void
47 icmp_error(n, type, code, dest, destifp)
48 struct mbuf *n;
49 int     type, code;
50 n_long  dest;
51 struct ifnet *destifp;
52 {
53     struct ip *oip = mtod(n, struct ip *), *nip;
54     unsigned oiplen = oip->ip_hl << 2;
55     struct icmp *icp;
56     struct mbuf *m;
57     unsigned icmplen;

58     if (type != ICMP_REDIRECT)
59         icmpstat.icps_error++;
60     /*
61      * Don't send error if not the first fragment of message.
62      * Don't error if the old packet protocol was ICMP
63      * error message, only known informational types.
64      */
65     if (oip->ip_off & ~(IP_MF | IP_DF))
66         goto freeit;
67     if (oip->ip_p == IPPROTO_ICMP && type != ICMP_REDIRECT &&
68         n->m_len >= oiplen + ICMP_MINLEN &&
69         !ICMP_INFOTYPE(((struct icmp *)((caddr_t) oip + oiplen))->icmp_type)){
70         icmpstat.icps_oldicmp++;
71         goto freeit;
72     }
73     /* Don't send error in response to a multicast or broadcast packet */
74     if (n->m_flags & (M_BCAST | M_MCAST))
75         goto freeit;
```

ip_icmp.c

图11-30 icmp_error 函数：验证

- 确认该报文(图11-30)；
- 构造首部(图11-32)；并
- 把原来的数据报包含进来(图11-33)。

46-57 参数是：n，指向包含无效数据报缓存链的指针；type和code，ICMP差错类型和代码；dest，ICMP重定向报文中的下一跳路由器地址；以及destifp，指向原始IP分组外出接口的指针。mtod把缓存链指针n转换成oip，oip是指向缓存中ip结构的指针。原始IP分组的字节长度保存在ioplen中。

58-75 icps_error统计除重定向报文外的所有ICMP差错。Net/3不把重定向报文看作错误，而且icps_error也不是一个SNMP变量。

icmp_error丢弃无效数据报oip，并且在以下情况下，不发送差错报文：

- 除IP_MF和IP_DF外，ip_off的某些位非零(习题11.10)。这表明oip不是数据报的第一个分片，而且ICMP决不能为跟踪数据报的分片而生成差错报文。
- 无效数据报本身是一个ICMP差错报文。如果icmp_type是ICMP请求或响应类型，则ICMP_INFOTYPE返回真；如果icmp_type是一个差错类型，则ICMP_INFOTYPE返回假。

Net/3不考虑ICMP重定向报文差错，尽管RFC 1122要求考虑。

- 数据报作为链路层广播或多播到达(由M_BCAST和M_MCAST标志表明)。

在以下两种其他情况下，不能发送ICMP差错报文：

- 该数据报是发给IP广播和IP多播地址的。
- 数据报的源站地址不是单播IP地址(也即，这个源站地址是一个全零地址、环回地址、广播地址、多播地址或E类地址)。

Net/3无法检查第一种情况。icmp_reflect函数强调了第二种情况(11.12节)。

有趣的是，Net/2的Deering多播扩展并不丢弃第一种类型的数据报。因为Net/3的多播程序来自Deering多播扩展，所以，检测似乎被删去了。

这些限制的目的是为了避免有错的广播数据报触发网络上所有主机都发出ICMP差错报文。当网络上所有主机同时要发送差错报文时，产生的广播风暴会使整个网络的通信崩溃。

这些规则适用于ICMP差错报文，但不适用于ICMP回答。如RFC 1122和RFC 1127的讨论，允许响应广播请求，但既不推荐也不鼓励。Net/3只响应具有单播源地址的广播请求，因为ip_output会把返回到广播地址的ICMP报文丢弃(图11-39)。

图11-31是ICMP差错报文的构造。

图11-32的程序构造差错报文。

76-106 icmp_error以下面的方式构造ICMP差错报文的首部：

- m_gethdr分配一个新的分组首部缓存。MH_ALIGN定位缓存的数据指针，使无效数据报的ICMP首部、IP首部(和选项)和最多8字节的数据被放在缓存的最后。
- icmp_type、icmp_code、icmp_gwaddr(用于重定向)、icmp_pptr(用于参数问题)和icmp_nextmtu(用于要求分片报文)被初始化。icmp_nextmtu字段实现了RFC 1191中描述的要求分片报文的扩展。卷1的24.2节描述的“路径MTU发现算法”依赖于这个报文。

一旦构造好ICMP首部，就必须把原始数据报的一部分附到首部上，如图11-33所示。

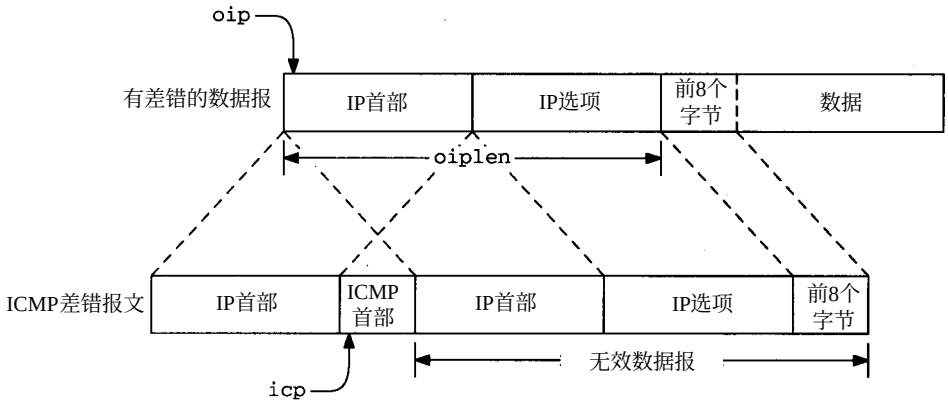


图11-31 ICMP差错报文的构造

ip_icmp.c

```
 76      /*
 77       * First, formulate icmp message
 78       */
 79      m = m_gethdr(M_DONTWAIT, MT_HEADER);
 80      if (m == NULL)
 81          goto freeit;
 82      icmplen = oiplen + min(8, oip->ip_len);
 83      m->m_len = icmplen + ICMP_MINLEN;
 84      MH_ALIGN(m, m->m_len);
 85      icp = mtod(m, struct icmp *);
 86      if ((u_int) type > ICMP_MAXTYPE)
 87          panic("icmp_error");
 88      icmpstat.icps_outhist[type]++;
 89      icp->icmp_type = type;
 90      if (type == ICMP_REDIRECT)
 91          icp->icmp_gwaddr.s_addr = dest;
 92      else {
 93          icp->icmp_void = 0;
 94          /*
 95           * The following assignments assume an overlay with the
 96           * zeroed icmp_void field.
 97           */
 98          if (type == ICMP_PARAMPROB) {
 99              icp->icmp_pptr = code;
100              code = 0;
101          } else if (type == ICMP_UNREACH &&
102                     code == ICMP_UNREACH_NEEDFRAG && destifp) {
103              icp->icmp_nextmtu = htons(destifp->if_mtu);
104          }
105      }
106      icp->icmp_code = code;
```

ip_icmp.c

图11-32 icmp_error 函数：报文首部构造

107-125 无效数据报的IP首部、选项和数据(一共是icmplen个字节)被复制到ICMP差错报文中。同时，首部的长度被加回无效数据报的ip_len中。

在udp_usrreq中，UDP也把首部长度加回到无效数据报的ip_len。其结果是一个ICMP报文，该报文具有无效分组 IP首部内的不正确的数据报长度。作者发现，许多基于Net/2程序的系统都有这个错误，Net/1系统没有这个问题。

ip_icmp.c

```
107     bcopy((caddr_t) oip, (caddr_t) & icp->icmp_ip, icmplen);
108     nip = &icp->icmp_ip;
109     nip->ip_len = htons((u_short) (nip->ip_len + oiplen));

110     /*
111      * Now, copy old ip header (without options)
112      * in front of icmp message.
113      */
114     if (m->m_data - sizeof(struct ip) < m->m_pktdat)
115                 panic("icmp len");
116     m->m_data -= sizeof(struct ip);
117     m->m_len += sizeof(struct ip);
118     m->m_pkthdr.len = m->m_len;
119     m->m_pkthdr.rcvif = n->m_pkthdr.rcvif;
120     nip = mtod(m, struct ip *);
121     bcopy((caddr_t) oip, (caddr_t) nip, sizeof(struct ip));
122     nip->ip_len = m->m_len;
123     nip->ip_hl = sizeof(struct ip) >> 2;
124     nip->ip_p = IPPROTO_ICMP;
125     nip->ip_tos = 0;
126     icmp_reflect(m);

127   freeit:
128     m_freem(n);
129 }
```

ip_icmp.c

图11-33 icmp_error 函数：包含原始数据报

因为MH_ALIGN把ICMP报文分配在缓存的最后，所以缓存的前面应该有足够的空间存放IP首部。无效数据报的IP首部(除选项外)被复制到ICMP报文的前面。

Net/2版本的这部分有一个错误：函数的最后一个 bcopy移动oiplen个字节，其中包括无效数据报的选项。应该只复制没有选项的标准首部。

在恢复正确的数据报长度(ip_len)、首部长度(ip_hl)和协议(ip_p)后，IP首部就完整了。TOS字段(ip_tos)被清除。

RFC 792和RFC 1122推荐在ICMP报文中，把TOS字段设为0。

126-129 完整的报文被传给icmp_reflect，由icmp_reflect把它发回源主机。丢掉无效数据报。

11.12 icmp_reflect函数

icmp_reflect把ICMP回答或差错发回给请求或无效数据报的源站。必须牢记，icmp_reflect在发送数据报之前，把它的源站地址和目的地址倒过来。与 ICMP 报文的源站和目的站地址有关的规则非常复杂，图 11-34对其中几个函数的作用作了小结。

我们分三部分讨论icmp_reflect函数：源站和目的站地址选择、选项构造及组装和发送。图11-35显示了该函数的第一部分。

1. 设置目的地址

329-345 icmp_reflect一开始，就复制 ip_dst，并把请求或差错报文的源站地址ip_src移到ip_dst。icmp_error和icmp_reflect保证：ip_src对差错报文而言是有效的目的地址。ip_output丢掉所有发往广播地址的分组。

函 数	小 结
icmp_input	在地址掩码请求中，用接收接口的广播或目的地址代替全0地址
icmp_error	把作为链路级广播或多播发送的数据报引起的差错报文丢弃。应该丢弃(但没有)发往IP广播或多播地址的数据报引起的报文
icmp_reflect	丢弃报文，而不是把它返回给多播或实验地址 把非单播目的地址转换成接收接口的地址，对返回的报文来说，目的地址就是一个有效的源地址 交换源站和目的站的地址
ip_output	按照ICMP的请求丢弃输出的广播(也就是说，丢弃由发往广播地址的分组产生的差错报文)

图11-34 ICMP丢弃和地址小结

2. 选择源站地址

346-371 icmp_reflect在in_ifaddr中找到具有单播或广播地址的接口，该接口地址与原始数据报的目的地址匹配，这样，icmp_reflect就为报文选好了源地址。在多接口主机上，匹配的接口可能不是接收该数据报的接口。如果没有匹配，就选择正在接收的接口的in_ifaddr结构，或者in_ifaddr中的第一个地址(如果该接口没有被配置成IP可用的)。该函数把ip_src设成所选的地址，并把ip_ttl改为255(MAXTTL)，因为这是一个新的数据报。

RFC 1700推荐把所有IP分组的TTL字段设成64。但是现在，许多系统把ICMP报文的TTL设成255。

TTL的取值有一个拆衷。小的TTL避免分组在路由回路里面循环，但也有可能使分组无法到达远一点的节点(有很多跳)。大的TTL允许分组到达远距离的主机，但却让分组在路由回路里循环较长时间。

ip_icmp.c

```
329 void
330 icmp_reflect(m)
331 struct mbuf *m;
332 {
333     struct ip *ip = mtod(m, struct ip *);
334     struct in_ifaddr *ia;
335     struct in_addr t;
336     struct mbuf *opts = 0, *ip_srcroute();
337     int     optlen = (ip->ip_hl << 2) - sizeof(struct ip);

338     if (!in_canforward(ip->ip_src) &&
339         ((ntohl(ip->ip_src.s_addr) & IN_CLASSA_NET) !=
340          (IN_LOOPBACKNET << IN_CLASSA_NSHIFT))) {
341         m_freem(m);             /* Bad return address */
342         goto done;              /* Ip_output() will check for broadcast */
343     }
344     t = ip->ip_dst;
345     ip->ip_dst = ip->ip_src;
346     /*
347      * If the incoming packet was addressed directly to us,
348      * use dst as the src for the reply.  Otherwise (broadcast
349      * or anonymous), use the address which corresponds
350      * to the incoming interface.
351      */
352     for (ia = in_ifaddr; ia; ia = ia->ia_next) {
```

图11-35 icmp_reflect 函数：地址选择

```
353         if (t.s_addr == IA_SIN(ia)->sin_addr.s_addr)
354             break;
355         if ((ia->ia_ifp->if_flags & IFF_BROADCAST) &&
356             t.s_addr == satosin(&ia->ia_broadaddr)->sin_addr.s_addr)
357             break;
358     }
359     icmpdst.sin_addr = t;
360     if (ia == (struct in_ifaddr *) 0)
361         ia = (struct in_ifaddr *) ifaof_ifpforaddr(
362                         (struct sockaddr *) &icmpdst, m->m_pkthdr.rcvif);
363     /*
364      * The following happens if the packet was not addressed to us,
365      * and was received on an interface with no IP address.
366      */
367     if (ia == (struct in_ifaddr *) 0)
368         ia = in_ifaddr;
369     t = IA_SIN(ia)->sin_addr;
370     ip->ip_src = t;
371     ip->ip_ttl = MAXTTL;
```

ip_icmp.c

图11-35 (续)

RFC 1122提出，对到达的回显请求或时间戳请求，要求把其中的源路由选项及记录路由和时间戳选项的建议，附到回答报文中。在这个过程期间，源路由必须被逆转过来。 RFC 1122没有涉及在其他ICMP回答报文中如何处理这些选项。 Net/3把这些规则应用于地址掩码请求，因为它在构造地址掩码回答后调用了`icmp_reflect`(图11-21)。

程序的下一部分(图11-36)为ICMP报文构造选项。

ip_icmp.c

```
372     if (optlen > 0) {
373         u_char *cp;
374         int     opt, cnt;
375         u_int   len;
376         /*
377          * Retrieve any source routing from the incoming packet;
378          * add on any record-route or timestamp options.
379          */
380         cp = (u_char *) (ip + 1);
381         if ((opts = ip_srcroute()) == 0 &&
382             (opts = m_gethdr(M_DONTWAIT, MT_HEADER))) {
383             opts->m_len = sizeof(struct in_addr);
384             mtod(opts, struct in_addr *)->s_addr = 0;
385         }
386         if (opts) {
387             for (cnt = optlen; cnt > 0; cnt -= len, cp += len) {
388                 opt = cp[IPOPT_OPTVAL];
389                 if (opt == IPOPT_EOL)
390                     break;
391                 if (opt == IPOPT_NOP)
392                     len = 1;
393                 else {
394                     len = cp[IPOPT_OLEN];
395                     if (len <= 0 || len > cnt)
396                         break;
397                 }
398                 /*
```

图11-36 `icmp_reflect` 函数：选项构造

```
399                  * Should check for overflow, but it "can't happen"
400                  */
401                 if (opt == IPOPT_RR || opt == IPOPT_TS ||
402                     opt == IPOPT_SECURITY) {
403                     bcopy((caddr_t) cp,
404                           mtod(opts, caddr_t) + opts->m_len, len);
405                     opts->m_len += len;
406                 }
407             }
408             /* Terminate & pad, if necessary */
409             if (cnt = opts->m_len % 4) {
410                 for (; cnt < 4; cnt++) {
411                     *(mtod(opts, caddr_t) + opts->m_len) =
412                         IPOPT_EOL;
413                     opts->m_len++;
414                 }
415             }
416         }
```

ip_icmp.c

图11-36 (续)

3. 取得逆转后的源路由

372-385 如果到达的数据报没有选项，控制被传给 430行(图11-37)。icmp_error传给icmp_reflect的差错报文从来没有IP选项，所以后面的程序只用于那些被转换成回答并直接传给icmp_reflect的ICMP请求。

cp指向回答的选项的开始。ip_srcroute逆转并返回所有在ipintr处理数据报时保存下来的源路由选项。如果ip_srcoute返回0，即请求中没有源路由选项，icmp_reflect分配并初始化一个mbuf，作为空的ipoption结构。

4. 加上记录路由和时间戳选项

386-416 如果opts指向某个缓存，for循环搜索原始IP首部的选项，在ip_srcroute返回的源路由后面加上记录路由和时间戳选项。

在ICMP报文发送之前必须移走原始首部里的选项。这由图 11-37中的程序完成。

ip_icmp.c

```
417         /*
418          * Now strip out original options by copying rest of first
419          * mbuf's data back, and adjust the IP length.
420          */
421         ip->ip_len -= optlen;
422         ip->ip_hl = sizeof(struct ip) >> 2;
423         m->m_len -= optlen;
424         if (m->m_flags & M_PKTHDR)
425             m->m_pkthdr.len -= optlen;
426         optlen += sizeof(struct ip);
427         bcopy((caddr_t) ip + optlen, (caddr_t) (ip + 1),
428               (unsigned) (m->m_len - sizeof(struct ip)));
429     }
430     m->m_flags &= ~(M_BCAST | M_MCAST);
431     icmp_send(m, opts);
432 done:
433     if (opts)
434         (void) m_free(opts);
435 }
```

ip_icmp.c

图11-37 icmp_reflect 函数：最后的组装

5. 移走原始选项

417-429 icmp_reflect把ICMP报文移到IP首部的后面，这样就从原始请求中移走了选项。如图11-38所示。新选项在opts所指向的mbuf里，被ip_output再次插入。

6. 发送报文和清除

430-435 在报文和选项被传给icmp_send之前，要明确地清除广播和多播标志位。此后释放掉存放选项的缓存。

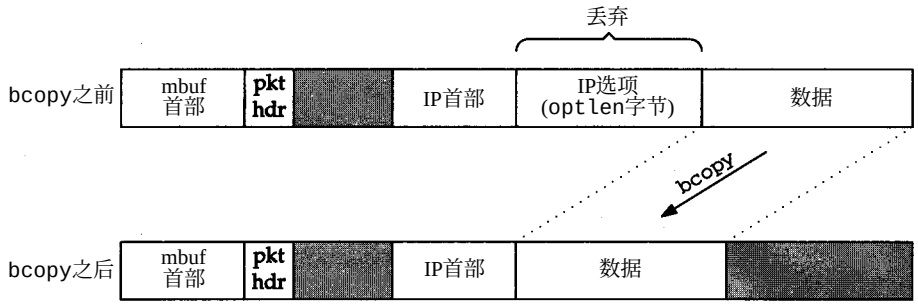


图11-38 icmp_reflect ：移走选项

11.13 icmp_send函数

icmp_send(图11-39)处理所有输出的ICMP报文，并在把它们传给IP层之前计算ICMP检验和。

ip_icmp.c

```
440 void
441 icmp_send(m, opts)
442 struct mbuf *m;
443 struct mbuf *opts;
444 {
445     struct ip *ip = mtod(m, struct ip *);
446     int     hlen;
447     struct icmp *icp;

448     hlen = ip->ip_hl << 2;
449     m->m_data += hlen;
450     m->m_len -= hlen;
451     icp = mtod(m, struct icmp *);
452     icp->icmp_cksum = 0;
453     icp->icmp_cksum = in_cksum(m, ip->ip_len - hlen);
454     m->m_data -= hlen;
455     m->m_len += hlen;
456     (void) ip_output(m, opts, NULL, 0, NULL);
457 }
```

ip_icmp.c

图11-39 icmp_send 函数

440-457 与icmp_input检测ICMP检验和一样， Net/3调整缓存的数据指针和长度，隐藏IP首部，让in_cksum只看到ICMP报文。计算好的检验和放在首部的 icmp_cksum，然后把数据报和所有选项传给ip_output。ICMP层并不维护路由高速缓存，所以 icmp_send只传给ip_output一个空指针(第4个参数)，而不是控制标志。特别是不传 IP_ALLOWBROADCAST，所以ip_output丢弃所有具有广播目的地址的 ICMP报文(也就是说，到达原始数据报的具有无效的源地址)。

11.14 icmp_sysctl函数

IP的icmp_sysctl函数只支持图11-40列出的选项。系统管理员可以用sysctl程序修改该选项。

Sysctl常量	Net/3变量	描 述
ICMPCTL_MASKREPL	icmpmaskrepl	系统是否响应ICMP地址掩码请求

图11-40 icmp_sysctl 参数

图11-41显示了icmp_sysctl函数。

ip_icmp.c

```
467 int
468 icmp_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
469 int    *name;
470 u_int   namelen;
471 void   *oldp;
472 size_t *oldlenp;
473 void   *newp;
474 size_t  newlen;
475 {

476     /* All sysctl names at this level are terminal. */
477     if (namelen != 1)
478         return (ENOTDIR);

479     switch (name[0]) {
480     case ICMPCTL_MASKREPL:
481         return (sysctl_int(oldp, oldlenp, newp, newlen, &icmpmaskrepl));
482     default:
483         return (ENOPROTOOPT);
484     }
485     /* NOTREACHED */
486 }
```

ip_icmp.c

图11-41 icmp_sysctl 函数

467-478 如果缺少所要求的ICMP sysctl名，就返回ENOTDIR。

479-486 ICMP级以下没有选项，所以，如果不识别选项，该函数就调用sysctl_int修改icmpmaskrepl或返回ENOPROTOOPT。

11.15 小结

ICMP协议是作为IP上面的运输层实现的，但它与IP层紧密结合一起。我们看到，内核直接响应ICMP请求报文，但把差错与回答传给合适的运输层协议或应用程序处理。当一个ICMP重定向报文到达时，内核立刻重定向表，并且也把重定向传给所有等待的进程，比如典型地传给一个路由守护程序。

我们将在23.9和27.6节看到UDP和TCP协议如何响应ICMP差错报文，在第32章看到进程如何产生ICMP请求。

习题

11.1 一个目的地址是0.0.0.0的请求所产生的ICMP地址掩码回答报文的源地址是什么？

11.2 试描述一个具有假的单播源地址的分组在链路级的广播会如何影响网络上另一个主机的运行。

11.3 RFC 1122建议，如果新的第一跳路由器与旧的第一跳路由器位于不同的子网，或者如果发送报文的路由器不是报文最终目的地的当前第一跳路由器，那么主机应该丢弃ICMP重定向报文。为什么要采纳这个建议？

11.4 如果ICMP信息请求是过时的，为什么`icmp_inout`要把它传给`rip_input`而不是丢弃它呢？

11.5 我们指出，Net/3在把IP分组放入一个ICMP差错报文之前，并不把它的偏移和长度字段转换成网络字节序。为什么这对IP位移字段来说是无关紧要的？

11.6 描述某种情况，使图11-25的`ifaof_ifpforaddr`返回一个空指针。

11.7 在一次时间戳询问中，时间戳后面的数据会怎么样？

11.8 实现以下改变，改进ICMP时间戳程序：
在缓存分组首部加上一个时间戳字段，让设备驱动程序把接收分组的确切时间记录在这个字段内，并用ICMP 时间戳程序把该值复制到`icmp_rtime`字段。
在输出端，让ICMP时间戳程序保存分组中的某个字节偏移，该位置用于保存时间戳里的当前时间。修改设备驱动程序，在发送分组之前插入时间戳。

11.9 修改`icmp_error`，使ICMP差错报文中返回最多64字节(像Solaris 2.x一样)的原始数据。

11.10 图11-30中，`ip_off`的高位被置位的分组会发生什么情况？

11.11 为什么图11-39中丢弃了`ip_output`返回的值？

第12章 IP 多 播

12.1 引言

第8章讲到，D类IP地址(224.0.0.0到239.255.255.255)不识别互联网内的单个接口，但识别接口组。因为这个原因，D类地址被称为多播组(multicast group)。具有D类目的地址的数据报被提交给互联网内所有加入相应多播组的各个接口。

Internet上利用多播的实验性应用程序包括：音频和视频会议应用程序、资源发现工具和共享白板等。

多播组的成员由于接口加入或离开组而动态地变化，这是根据各系统上运行的进程的请求决定的。因为多播组成员与接口有关，所以多接口主机可能针对每个接口，都有不同的多播组成员关系表。我们称一个特定接口上的组成员关系为一对 {接口，多播组}。

单个网络上的组成员利用 IGMP协议(第13章)在系统之间通信。多播路由器用多播选路协议(第14章)，如DVMRP(Distance Vector Multicast Routing Protocol，距离向量多播路由选择协议)传播成员信息。标准IP路由器可能支持多播选路，或者用一专用路由器处理多播选路。

如以太网、令牌环和FDDI一类的网络直接支持硬件多播。在 Net/3中，如果某个接口支持多播，那么在接口的`ifnet`结构(图3-7)中的`if_flags`标志的`IFF_MULTICAST`比特就被打开。因为以太网被广泛使用，并且 Net/3有以太网驱动器程序，所以我们将以以太网为例说明硬件支持的IP多播。多播业务通常在如SLIP和环回接口等的点到点网络上实现。

如果本地网络不支持硬件级多播，那么在某个特定接口上就得不到 IP多播业务。RFC 1122并不反对接口层提供软件级的多播业务，只要它对 IP是透明的。

RFC 1112 [Deering 1989] 描述了多播对主机的要求。分三个级别：

0级：主机不能发送或接收IP多播。

这种主机应该自动丢弃它收到的具有 D类目的地址的分组。

1级：主机能发送但不能接收IP多播。

在向某个IP多播组发送数据报之前，并不要求主机加入该组。多播数据报的发送方式与单播一样，除了多播数据报的目的地址是 IP多播组之外。网络驱动器必须能够识别出这个地址，把在本地网络上多播数据报。

2级：主机能发送和接收IP多播。

为了接收IP多播，主机必须能够加入或离开多播组，而且必须支持IGMP，能够在至少一个接口上交换组成员信息。多接口主机必须支持在它的接口的一个子网上的多播。Net/3符合2级主机要求，可以完成多播路由器的工作。与单播 IP选路一样，我们假定所描述的系统是一个多播路由器，并加上了 Net/3多播选路的程序。

知名的IP多播组

和UDP、TCP的端口号一样，互联网号授权机构 IANA(Internet Assigned Numbers

Authority)维护着一个注册的IP多播组表。当前的表可以在RFC 1700中查到。有关IANA的其他信息可以在RFC 1700中找到。图12-1只给出了一些知名的多播组。

组	描 述	Net/3常量
224.0.0.0	预留	*INADDR_UNSPEC_GROUP*
224.0.0.1	这个子网上的所有系统	*INADDR_ALLHOSTS_GROUP*
224.0.0.2	这个子网上的所有路由器	
224.0.0.3	没有分配	*INADDR_MAX_LOCAL_GROUP*
224.0.0.4	DVMRP路由器	
224.0.0.255	没有分配	
224.0.1.1	NTP网络时间协议	
224.0.1.2	SGI-Dogfight	

图12-1 一些注册的IP多播组

前256个组(224.0.0.0到224.0.0.255)是为实现IP单播和多播选路机制的协议预留的。不管发给其中任意一个组的数据报内IP首部的TTL值如何变化，多播路由器都不会把它转发出本地网络。

RFC 1075只对224.0.0.0组和224.0.0.1组有这个要求，但最常见的多播选路实现mrouted限制这里讨论的其他组。组224.0.0.0(INADDR_UNSPEC_GROUP)被预留，组224.0.0.255(INADDR_MAX_LOCAL_GROUP)标志着本地最后一个多播组。

对于符合2级的系统，要求其在系统初始化时(图6-17)，在所有的多播接口上加入224.0.0.1组(INADDR_ALLHOSTS_GROUP)，并且保持为该组成员，直到系统关闭。在一个互联网上，没有多播组与每个接口都对应。

想像一下，如果你的语音邮件系统有一个选项，可以向公司里的所有语音邮箱发一个消息。可能你就有这个选项。你发现它有用吗？对更大的公司适用吗？是否有人能向“所有邮箱”组发邮件，或者是否限制这么做？

单播和多播路由可能会加入224.0.0.2组进行互相通信。ICMP路由器请求报文和路由器通告报文可能被分别发往224.0.0.2(“所有路由器”组)和224.0.0.1(“所有主机”组)，而不是受限的广播地址(255.255.255.255)。

224.0.0.4组支持在实现DVMRP的多播路由器之间的通信。本地多播组范围内的其他组被类似地指派给其他路由选择协议。

除了前256个组外，其他组(224.0.1.0~239.255.255.255)或者被分配给多个多播应用程序协议，或者仍然没有被分配。图12-1中有两个例子，网络时间协议(224.0.1.1)和SGI-Dogfight(224.0.1.2)。

在本章中，我们注意到，是主机上的运输层发送和接收多播分组。尽管多播程序并不知道具体是哪个传输协议发送和接收多播数据报，但唯一支持多播的Internet传输协议是UDP。

12.2 代码介绍

本章中讨论的基本多播程序与标准IP程序在相同的文件里。图12-2列出了我们研究的文件。

文　件	描　述
net/if_either.h	以太网多播数据结构和宏定义
netinet/in.h	其他Internet多播数据结构
netinet/in_var.h	Internet多播数据结构和宏定义
netinet/ip_var.h	IP多播数据结构
net/if_ethersubr.c	以太网多播函数
netinet/in.c	组成员函数
netinet/ip_input.c	输入多播处理
netinet/ip_output.c	输出多播处理

图12-2　本章讨论的文件

12.2.1　全局变量

本章介绍了三个新的全局变量(图12-3)。

变　量	数据类型	描　述
ether_ipmulticast_min	u_char []	为IP预留的最小以太网多播地址
ether_ipmulticast_max	u_char []	为IP预留的最大以太网多播地址
ip_mrouter	struct socket *	多播选路守护程序创建的指向插口的指针

图12-3　本章引入的全局变量

12.2.2　统计量

本章讨论的程序更新全局ipstat结构中的几个计数器。

ipstat成员	描　述
ips_forward	被这个系统转发的分组数
ips_cantforward	不能被系统转发的分组数 —— 系统不是一个路由器
ips_noroute	由于无法访问到路由器而无法转发的分组数

图12-4　多播处理统计量

链路级多播统计放在ifnet结构中(图4-5)，还可能统计除IP以外的其他协议的多播。

12.3　以太网多播地址

IP多播的高效实现要求IP充分利用硬件级多播，因为如果没有硬件级多播，就不得不在网络上广播每个多播IP数据报，而每台主机也不得不检查每个数据报，把那些不是给它的丢掉。硬件在数据报到达IP层之前，就把没有用的过滤掉了。

为了保证硬件过滤器能正常工作，网络接口必须把IP多播组目的地址转换成网络硬件识别的链路级多播地址。在点到点网络上，如SLIP和环回接口，必须明确给出地址映射，因为只能有一个目的地址。在其他网络上，如以太网，也需要有一个明确地完成映射地址的函数。以太网的标准映射适用于任何使用802.3寻址方式的网络。

图4-12显示了以太网单播和多播地址的区别：如果以太网地址的高位字节的最低位是1，则它是一个多播地址；否则，它是一个单播地址。单播以太网地址由接口制造商分配，多播

地址由网络协议动态分配。

IP到以太网地址映射

因为以太网支持多种协议，所以要采取措施分配多播地址，避免冲突。IEEE管理以太网多播地址分配。IEEE把一块以太网多播地址分给IANA以支持IP多播。块的地址都以01:00:5e开头。

以00:00:5e开头的以太网单播也被分配给IANA，但为将来使用预留。

图12-5显示了从一个D类IP地址构造出一个以太网多播地址。

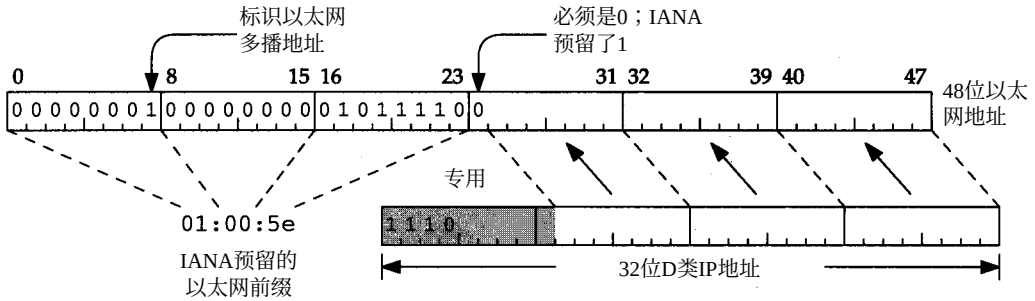


图12-5 IP和以太网地址之间的映射

图12-5显示的映射是一个多到一的映射。在构造以太网地址时，没有使用 D类IP地址的高位9比特。32个 IP多播组映射到一个以太网多播地址 (习题12.3)。我们将在12.14节看到这将如何影响输入的处理。图12-6显示了Net/3中实现这个映射的宏。

if_ether.h

```
61 #define ETHER_MAP_IP_MULTICAST(ipaddr, enaddr) \
62     /* struct in_addr *ipaddr; */ \
63     /* u_char enaddr[6];       */ \
64 { \
65     (enaddr)[0] = 0x01; \
66     (enaddr)[1] = 0x00; \
67     (enaddr)[2] = 0x5e; \
68     (enaddr)[3] = ((u_char *)ipaddr)[1] & 0x7f; \
69     (enaddr)[4] = ((u_char *)ipaddr)[2]; \
70     (enaddr)[5] = ((u_char *)ipaddr)[3]; \
71 }
```

if_ether.h

图12-6 ETHER_MAP_IP_MULTICAST 宏

IP到以太网多播映射

61-71 ETHER_MAP_IP_MULTICAST实现图12-5所示的映射。ipaddr指向D类多播地址，enaddr构造匹配的以太网地址，用 6字节的数组表示。该以太网多播地址的前 3个字节是0x01，0x00和0x5e，后面跟着0比特，然后是D类IP地址的低23位。

12.4 ether_multi结构

Net/3为每个以太网接口维护一个该硬件接收的以太网多播地址范围表。这个表定义了该设备要实现的多播过滤。因为大多数以太网设备能选择地接收的地址是有限的，所以 IP层必须要准备丢弃那些通过了硬件过滤的数据报。地址范围被保存在ether_multi结构中(图12-7)：

if_ether.h

```
147 struct ether_multi {
148     u_char  enm_addrlo[6];      /* low  or only address of range */
149     u_char  enm_addrhi[6];      /* high or only address of range */
150     struct arpcom *enm_ac;      /* back pointer to arpcom */
151     u_int   enm_refcount;       /* no. claims to this addr/range */
152     struct ether_multi *enm_next;   /* ptr to next ether_multi */
153 };
```

if_ether.h

图12-7 ether_multi 结构

1. 以太网多播地址

147-153 enm_addrlo和enm_addrhi指定需要被接收的以太网多播地址的范围。当enm_addrlo和enm_addrhi相同时，就指定一个以太网地址。ether_multi的完整列表附在每个以太网接口的arpcom结构中(图3-26)。以太网多播独立于ARP——使用arpcom结构只是为了方便，因为该结构已经存在于所有以太网接口结构中。

我们将看到，这个范围的开头和结尾总是相同的，因为在Net/3中，进程无法指定地址范围。

enm_ac指回相关接口的arpcom结构，enm_refcount跟踪对ether_multi结构的使用。当引用计数变成0时，就释放arpcom结构。enm_next把单个接口的ether_multi结构做成链表。图12-8显示出，有三个ether_multi结构的链表附在le_softc[0]上，这是我们以太网接口示例的ifnet结构。

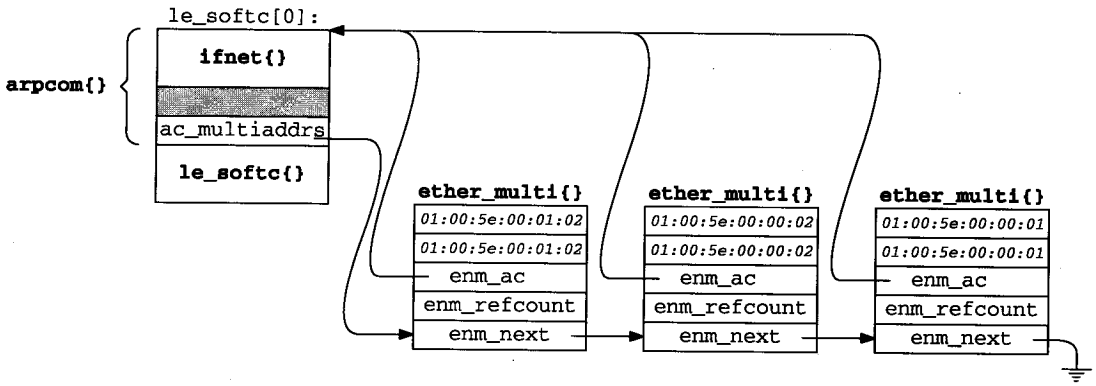


图12-8 有三个ether_multi 结构的LANCE接口

在图12-8中，我们看到：

- 接口已经加入了三个组。很有可能是224.0.0.1(所有主机)、224.0.0.2(所有路由器)和224.0.1.2(SGI-dogfight)。因为以太网到IP地址的映射是一到多的，所以只看到以太网多播地址的结果，无法确定确切的IP多播地址。比如，接口可能已经加入了225.0.0.1、225.0.0.2和226.0.1.2组。
- 有了enm_ac后向指针，就很容易找到链表的开始，释放某个ehter_multi结构，无需再实现双向链表。
- ether_multi只适用于以太网设备。其他多播设备可能有其他实现。

图12-9中的ETHER_LOOKUP_MULTI宏，搜索某个ether_multi结构，找到地址范围。

2. 以太网多播查找

166-177 addrlo和addrhi指定搜索的范围，ac指向包含了要搜索链表的arpcom结构。

for循环完成线性搜索，在表的最后结束，或者当enm_addrlo和enm_addrhi都分别与和所提供的addrlo和addrhi匹配时结束。当循环终止时，enm为空或者指向某个匹配的ether_multi结构。

if_ether.h

```
166 #define ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm) \
167     /* u_char addrlo[6]; */ \
168     /* u_char addrhi[6]; */ \
169     /* struct arpcom *ac; */ \
170     /* struct ether_multi *enm; */ \
171 { \
172     for ((enm) = (ac)->ac_multiaddrs; \
173         (enm) != NULL && \
174         (bcmp((enm)->enm_addrlo, (addrlo), 6) != 0 || \
175          bcmp((enm)->enm_addrhi, (addrhi), 6) != 0); \
176         (enm) = (enm)->enm_next); \
177 }
```

if_ether.h

图12-9 ETHER_LOOKUP_MULTI 宏

12.5 以太网多播接收

从本节以后，本章只讨论IP多播。但是，在Net/3中，也有可能把系统配置成接收所有以太网多播分组。虽然对IP协议族没有用，但内核的其他协议族可能准备接收这些多播分组。发出图12-10中的ioctl命令，就可以明确地进行多播配置。

命 令	参 数	函 数	描 述
SIOCADDMULTI	struct ifreq *	ifioctl	在接收表里加上多播地址
SIOCDELMULTI	struct ifreq *	ifioctl	从接收表里删去多播地址

图12-10 多播ioctl 命令

这两个命令被ifioctl(图12-11)直接传给ifreq结构(图6-12)中所指定的接口的设备驱动程序。

if.c

```
440     case SIOCADDMULTI:
441     case SIOCDELMULTI:
442         if (error = suser(p->p_ucred, &p->p_acflag))
443             return (error);
444         if (ifp->if_ioctl == NULL)
445             return (EOPNOTSUPP);
446         return ((*ifp->if_ioctl) (ifp, cmd, data));
```

if.c

图12-11 ifioctl 函数：多播命令

440-446 如果该进程没有超级用户权限，或者如果接口没有if_ioctl结构，则ifioctl返回一个错误；否则，把请求直接传给该设备驱动程序。

12.6 in_multi结构

12.4节描述的以太网多播数据结构并不专用于IP；它们必须支持所有内核支持的任意协议族的多播活动。在网络级，IP维护着一个与接口相关的IP多播组表。

为了实现方便，把这个IP多播表附在与该接口有关的in_ifaddr结构中。6.5节讲到，这

个结构中包含了该接口的单播地址。除了它们都与同一个接口相关以外，这个单播地址与所附的多播组表之间没有任何关系。

这是Net/3实现的产品。也可以在一个不接收IP单播分组的接口上，支持IP多播组。

图12-12中的in_multi结构描述了每个IP多播{接口，组}对。

in_var.h

```
111 struct in_multi {
112     struct in_addr inm_addr;     /* IP multicast address */
113     struct ifnet *inm_ifp;       /* back pointer to ifnet */
114     struct in_ifaddr *inm_ia;    /* back pointer to in_ifaddr */
115     u_int    inm_refcount;       /* no. membership claims by sockets */
116     u_int    inm_timer;          /* IGMP membership report timer */
117     struct in_multi *inm_next;   /* ptr to next multicast address */
118 };
```

in_var.h

图12-12 in_multi 结构

1. IP多播地址

111-118 inm_addr是一个D类多播地址(如224.0.0.1，所有主机组)。inm_ifp指回相关接口的ifnet结构，而inm_ia指回接口的in_ifaddr结构。

只有当系统中的某个进程通知内核，它要在某个特定的 {接口，组}对上接收多播数据报时，才存在一个in_multi结构。由于可能会有多个进程要求接收发往同一个对上的数据报，所以inm_refcount跟踪对该对的引用次数。当没有进程对某个特定的对感兴趣时，inm_refcount就变成0，in_multi结构就被释放掉。这个动作可能会引起相关的ether_multi结构也被释放，如果此时它的引用计数也变成了0。

inm_timer是第13章描述的IGMP协议实现的一部分，最后，inm_next指向表中的下一个in_multi结构。

图12-13用接口示例le_softc[0]显示了接口，即它的单播地址和它的 IP多播组表之间的关系。

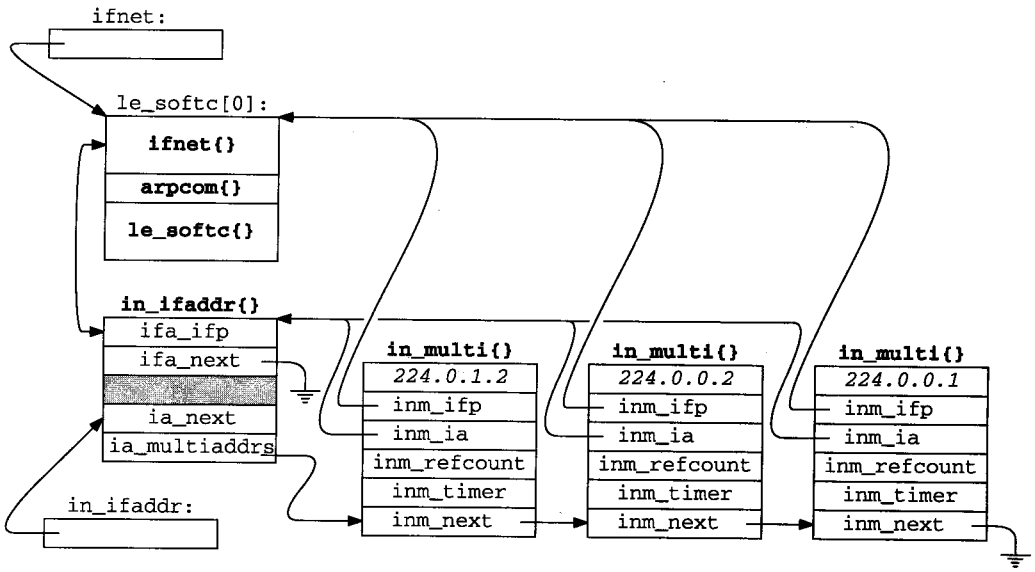


图12-13 le接口的一个IP多播组表

为了清楚起见，我们已经省略了对应的 ether_multi结构(图12-34)。如果系统有两个以太网网卡，第二个可能由 le_softc[1]管理，还可能有它自己的附在 arpcom结构的多播组表。IN_LOOKUP_MULTI宏(图12-14)搜索IP多播表寻找某个特定多播组。

2. IP多播查找

131-146 IN_LOOKUP_MULTI在与接口 ifp相关的多播组表中查找多播组 addr。IFP_TO_IA搜索Internet地址表in_ifaddr，寻找与接口ifp相关的in_ifaddr结构。如果IFP_TO_IA找到一个接口，则for循环搜索它的IP多播表。循环结束后，inm为空或指向匹配的in_multi结构。

in_var.h

```
131 #define IN_LOOKUP_MULTI(addr, ifp, inm) \
132     /* struct in_addr addr; */ \
133     /* struct ifnet *ifp; */ \
134     /* struct in_multi *inm; */ \
135 { \
136      struct in_ifaddr *ia; \
137 \
138     IFP_TO_IA((ifp), ia); \
139     if (ia == NULL) \
140         (inm) = NULL; \
141     else \
142         for ((inm) = ia->ia_multiaddrs; \
143             (inm) != NULL && (inm)->inm_addr.s_addr != (addr).s_addr; \
144              (inm) = inm->inm_next) \
145             continue; \
146 }
```

in_var.h

图12-14 IN_LOOKUP_MULTI 宏

12.7 ip_moptions结构

运输层通过 ip_moptions结构包含的多播选项控制多播输出处理。例如， UDP调用ip_output是：

```
error = ip_output(m, inp->inp_options, &inp->inp_route,
                  inp->inp_socket->so_options & (SO_DONTROUTE|SO_BROADCAST),
                  inp->inp_moptions);
```

在第22章中我们将看到， inp指向某个Internet协议控制块(PCB)，并且UDP为每个由进程创建的socket关联一个PCB。在PCB内，inp_moptions是指向某个ip_moptions结构的指针。这里我们看到，对每个输出的数据报，都可以给 ip_output传一个不同的ip_moptions结构。图12-15是ip_moptions结构的定义。

ip_var.h

```
100 struct ip_moptions {
101     struct  ifnet *imo_multicast_ifp; /* ifp for outgoing multicasts */
102     u_char  imo_multicast_ttl;        /* TTL for outgoing multicasts */
103     u_char  imo_multicast_loop;       /* 1 => hear sends if a member */
104     u_short imo_num_memberships;      /* no. memberships this socket */
105     struct  in_multi *imo_membership[IP_MAX_MEMBERSHIPS];
106 };
```

ip_var.h

图12-15 ip_moptions 结构

多播选项

100-106 ip_output通过imo_multicast_ifp指向的接口对输出的多播数据报进行选路。如果imo_multicast_ifp为空，就通过目的站多播组的默认接口(第14章)。

imo_multicast_ttl为外出的多播数据报指定初始的IP TTL。默认值是1，把多播数据报保留在本地网络内。

如果imo_multicast_loop是0，就不回送数据报，也不把数据报提交给正在发送的接口，即使该接口是多播组的成员。如果imo_multicast_loop是1，并且如果正在发送的接口是多播组的成员，就把多播数据报回送给该接口。

最后，整数imo_num_memberships和数组imo_membership维护与该结构相关的{接口，组}对。所有对该表的改变都转告给IP，由IP在所连到的本地网络上宣布成员的变化。imo_membership数组的每个入口都是指向一个in_multi结构的指针，该in_multi结构附在适当接口的in_ifaddr结构上。

12.8 多播的插口选项

图12-16显示了几个IP级的插口选项，提供对ip_moptions结构的进程级访问。

命 令	参 数	函 数	描 述
IP_MULTICAST_IF	struct in_addr	ip_ctloutput	为外出的多播选择默认接口
IP_MULTICAST_TTL	u_char	ip_ctloutput	为外出的多播选择默认的TTL
IP_MULTICAST_LOOP	u_char	ip_ctloutput	允许或使能回送外出的多播
IP_ADD_MEMBERSHIP	struct ip_mreq	ip_ctloutput	加入一个多播组
IP_DROP_MEMBERSHIP	struct ip_mreq	ip_ctloutput	离开一个多播组

图12-16 多播插口选项

我们在图8-31中看到ip_ctloutput函数的整体结构。图12-17显示了与改变和检索多播选项有关的情况语句。

ip_output.c

```
448         case PRCO_SETOPT:
449             switch (optname) {
                    /* other set cases */
486             case IP_MULTICAST_IF:
487             case IP_MULTICAST_TTL:
488             case IP_MULTICAST_LOOP:
489             case IP_ADD_MEMBERSHIP:
490             case IP_DROP_MEMBERSHIP:
491                 error = ip_setmoptions(optname, &inp->inp_moptions, m);
492                 break;
493               freeit:
494             default:
495                 error = EINVAL;
496                 break;
497             }
498             if (m)
```

图12-17 ip_ctloutput 函数：多播选项

```
499                 (void) m_free(m);
500             break;

501         case PRCO_GETOPT:
502             switch (optname) {

                                /* other get cases */

539             case IP_MULTICAST_IF:
540             case IP_MULTICAST_TTL:
541             case IP_MULTICAST_LOOP:
542             case IP_ADD_MEMBERSHIP:
543             case IP_DROP_MEMBERSHIP:
544                 error = ip_getmoptions(optname, inp->inp_moptions, mp);
545                 break;

546             default:
547                 error = ENOPROTOOPT;
548                 break;
549             }
```

ip_output.c

图12-17 (续)

486-491 所有多播选项都由ip_setmoptions和ip_getmoptions函数处理。ip_moptions结构由引用传给

539-549 ip_getmoptions和ip_setmoptions，该结构与发布ioctl命令的那个插口关联。

对于PRCO_SETOPT和PRCO_GETOPT两种情况，选项不识别时返回的差错码是不一样的。ENOPROTOOPT是更合理的选择。

12.9 多播的TTL值

多播的TTL值难以理解，因为它们有两个作用。TTL值的基本作用，如IP分组一样，是限制分组在互联网内的生存期，避免它在网络内部无限地循环。第二个作用是，把分组限制在管理边界所指定的互联网的某个区域内。管理区域是由一些主观的词语指定的，如“这个结点”，“这个公司”，“这个州”等，并与分组开始的地方有关。与多播分组有关的区域叫做它的辖域(scope)。

RFC 1122的标准实现把生存期和辖域这两个概念合并在IP首部的一个TTL值里。当IP TTL变成0时，除了丢弃该分组外，多播路由器还给每个接口关联了一个TTL阈值，限制在该接口上的多播传输。一个要在该接口上传输的分组必须具有大于或等于该接口阈值的TTL。由于这个原因，多播分组可能会在它的TTL到0之前就被丢弃了。

阈值是管理员在配置多播路由器时分配的，这些值确定了多播分组的辖域。管理员使用的阈值策略以及数据报的源站与多播接口之间的距离定义多播数据报的初始TTL值的意义。

图12-18显示了多种应用程序的推荐TTL值和推荐的阈值。

第一栏是IP首部中的ip_ttl初始值。第二栏是应用程序专用阈值([Casner 1993])。第三栏是与该TTL值相关的推荐的辖域。

例如，一个要与本地结点外的网络通信的接口，多播阈值要被配置成32。所有开始时

TTL为32(或小于32)的数据报到达该接口时，TTL都小于32(假定源站和路由器之间至少有一跳)，所以它们在被转发到外部网络之前，都被丢弃了——即使TTL远大于0。

TTL初始值是128的多播数据报可以通过阈值为32的结点接口(只要它以少于128-32=96跳到达接口)，但将被阈值为128的洲际接口丢弃。

ip_ttl	应用程序	辖域
0		同一接口
1		同一子网
31	本地事件视频	
32		同一地点
63	本地事件音频	
64		同一区域
95	IETF频道2视频	
127	IETF频道1视频	
128		同一州
159	IETF频道2音频	
191	IETF频道1音频	
223	IETF频道2低速率音频	
255	IETF频道1低速率音频，辖域不受限	

图12-18 IP多播数据报的TTL值

12.9.1 MBONE

Internet上有一个路由器子网支持IP多播选路。这个多播骨干网称为MBONE,[Casner 1993]对其作了描述。它是为了支持用IP多播的实验——尤其是用音频和视频数据流的实验。在MBONE里，阈值限制了多种数据流传播的距离。在图12-18中，我们看到本地事件视频分组总是以TTL 31开始。阈值为32的接口总是阻止本地事件视频。另外，IETF频道1低速率音频，只受到IP TTL固有的最大255跳的限制。它能传播通过整个MBONE。MBONE内的路由器的管理员可以选择阈值，有选择地接受或丢弃MBONE数据流。

12.9.2 扩展环搜索

多播TTL的另一种用处是，只要改变探测数据报的初始TTL值，就能在互联网上探测资源。这个技术叫做扩展环搜索(expanding-ring search，[Boggs 1982])。初始TTL为0的数据报只能到达与外出接口相关的本地网络上的一个资源；TTL为1，则到达本地子网(如果存在)上的资源；TTL为2，则到达相距2跳的资源。应用程序指数地增加TTL的值，迅速地在大的互联网上探测资源。

RFC 1546 [Partridge、Mendez和Milliken 1993]描述了一种相关业务的任播(*anycasting*)。任播依赖一组显著的IP地址来表示更像多播的多个主机的组。与多播地址不同，网络必须传播所有任播的分组，直到它被至少一个主机接收。这样简化了应用程序的实现，不再进行扩展环搜索。

12.10 ip_setmoptions函数

ip_setmoptions函数块包括一个用来处理各选项的switch语句。图12-19是

ip_setmoptions的开始和结束。下面几节讨论switch的语句体。

ip_output.c

```
650 int
651 ip_setmoptions(optname, imop, m)
652 int     optname;
653 struct ip_moptions **imop;
654 struct mbuf *m;
655 {
656     int     error = 0;
657     u_char  loop;
658     int     i;
659     struct in_addr addr;
660     struct ip_mreq *mreq;
661     struct ifnet *ifp;
662     struct ip_moptions *imo = *imop;
663     struct route ro;
664     struct sockaddr_in *dst;
665     if (imo == NULL) {
666         /*
667          * No multicast option buffer attached to the pcb;
668          * allocate one and initialize to default values.
669          */
670         imo = (struct ip_moptions *) malloc(sizeof(*imo), M_IPMOPTS,
671                                             M_WAITOK);
672         if (imo == NULL)
673             return (ENOBUFS);
674         *imop = imo;
675         imo->imo_multicast_ifp = NULL;
676         imo->imo_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;
677         imo->imo_multicast_loop = IP_DEFAULT_MULTICAST_LOOP;
678         imo->imo_num_memberships = 0;
679     }
680     switch (optname) {

            /* switch cases */

857     default:
858         error = EOPNOTSUPP;
859         break;
860     }
861     /*
862      * If all options have default values, no need to keep the mbuf.
863      */
864     if (imo->imo_multicast_ifp == NULL &&
865         imo->imo_multicast_ttl == IP_DEFAULT_MULTICAST_TTL &&
866         imo->imo_multicast_loop == IP_DEFAULT_MULTICAST_LOOP &&
867         imo->imo_num_memberships == 0) {
868         free(*imop, M_IPMOPTS);
869         *imop = NULL;
870     }
871     return (error);
872 }
```

ip_output.c

图12-19 ip_setmoptions 函数

650-664 第一个参数，optname，指明正在改变哪个多播参数。第二个参数，imop，是指向某个ip_motions结构的指针。如果*imop不空，ip_setmoptions修改它所指向的

结构。否则，ip_setmoptions分配一个新的ip_moptions结构，并把它的地址保存在*imop里。如果没有内存了，ip_setmoptions立即返回ENOBUFS。后面的所有错误都通告error，error在函数的最后被返回给调用方。第三个参数，m，指向存放要改变选项数据的mbuf(图12-16的第二栏)。

1. 构造默认值

665-679 当分配一个新的ip_moptions结构时，ip_setmoptions把默认的多播接口指针初始化为空，把默认TTL初始化为1(IP_DEFAULT_MULTICAST_TTL)，使能多播数据报的回送，并清除组成员表。有了这些默认值后，ip_output查询路由表选择一个输出的接口，多播被限制在本地网络中，并且，如果输出的接口是目的多播组的成员，则系统将接收它自己的多播发送。

2. 进程选项

680-860 ip_setmoptions体由一个switch语句组成，其中对每种选项都有一个case语句。default情况(对未知选项)把error设成EOPNOTSUPP。

3. 如果默认值是OK，丢弃结构

861-872 switch语句之后，ip_setmoptions检查ip_moptions结构。如果所有多播选项与它们对应的默认值匹配，就不再需要该结构，将其释放。ip_setmoptions返回0或公布的差错码。

12.10.1 选择一个明确的多播接口：IP_MULTICAST_IF

当optname是IP_MULTICAST_IF时，传给ip_setmoptions的mbuf中就包含了多播接口的单播地址，该地址指定了在这个插口上发送的多播所使用的特定接口。图12-20是这个选项的程序。

ip_output.c

```
681     case IP_MULTICAST_IF:
682         /*
683          * Select the interface for outgoing multicast packets.
684          */
685         if (m == NULL || m->m_len != sizeof(struct in_addr)) {
686             error = EINVAL;
687             break;
688         }
689         addr = *(mtod(m, struct in_addr *));
690         /*
691          * INADDR_ANY is used to remove a previous selection.
692          * When no interface is selected, a default one is
693          * chosen every time a multicast packet is sent.
694          */
695         if (addr.s_addr == INADDR_ANY) {
696             imo->imo_multicast_ifp = NULL;
697             break;
698         }
699         /*
700          * The selected interface is identified by its local
701          * IP address.  Find the interface and confirm that
702          * it supports multicasting.
703          */
```

图12-20 ip_setmoptions 函数：选择多播输出接口

```
704         INADDR_TO_IFP(addr, ifp);
705         if (ifp == NULL || (ifp->if_flags & IFF_MULTICAST) == 0) {
706             error = EADDRNOTAVAIL;
707             break;
708         }
709         imo->imo_multicast_ifp = ifp;
710         break;
```
ip_output.c

图12-20 (续)

1. 验证

681-698 如果没有提供 mbuf，或者 mbuf中的数不是一个 in_addr结构的大小，则 ip_setmoptions通告一个EINVAL差错；否则把数据复制到 addr。如果接口地址是 INADDR_ANY，则丢弃所有前面选定的接口。对后面用这个 ip_moptions结构的多播，将根据它们的目的多播组进行选路，而不再通过一个明确命名的接口 (图12-40)。

2. 选择默认接口

699-710 如果addr中有地址，就由INADDR_TO_IFP找到匹配接口的位置。如果找不到匹配或接口不支持多播，就发布 EADDRNOTAVAIL。否则，匹配接口 ifp成为与这个 ip_moptions结构相关的输出请求的多播接口。

12.10.2 选择明确的多播TTL：IP_MULTICAST_TTL

当optname是IP_MULTICAST_TTL时，缓存中有一个字节指定输出多播的 IP TTL。这个TTL是ip_output在每个发往相关插口的多播数据报中插入的。图 12-21是该选项的程序。

```
711     case IP_MULTICAST_TTL:
712         /*
713          * Set the IP time-to-live for outgoing multicast packets.
714          */
715         if (m == NULL || m->m_len != 1) {
716             error = EINVAL;
717             break;
718         }
719         imo->imo_multicast_ttl = *(mtod(m, u_char *));
720         break;
```
ip_output.c

图12-21 ip_setmoptions 函数：选择明确的多播TTL

验证和选项默认的TTL

711-720 如果缓存中有一个字节的数据，就把它复制到 imo_multicast_ttl。否则，发布EINVAL。

12.10.3 选择多播环回：IP_MULTICAST_LOOP

通常，多播应用程序有两种形式：

- 一个系统内一个发送方和多个远程接收方的应用程序。这种配置中，只有一个本地进程向多播组发送数据报，所以无需回送输出的多播。这样的例子有多播选路守护进程和会议系统。
- 一个系统内的多个发送方和接收方。必须回送数据报，确保每个进程接收到系统其他发

送方的传送。

IP_MULTICAST_LOOP选项(图12-22)为ip_moptions结构选择回送策略。

ip_output.c

```
721     case IP_MULTICAST_LOOP:
722         /*
723          * Set the loopback flag for outgoing multicast packets.
724          * Must be zero or one.
725          */
726         if (m == NULL || m->m_len != 1 ||
727             (loop = *(mtod(m, u_char *))) > 1) {
728             error = EINVAL;
729             break;
730         }
731         imo->imo_multicast_loop = loop;
732         break;
```

ip_output.c

图12-22 ip_setmoptions 函数：选择多播环回

验证和选择环回策略

721-732 如果m为空，或者没有1字节数据，或者该字节不是0或1，就发布EINVAL。否则，把该字节复制到imo_multicast_loop。0指明不要把数据报回送，1允许环回机制。

图12-23显示了多播数据报的最大辖域值之间的关系： imo_multicast_ttl和imo_multicast_loop。

imo_multicast-		Recipients			
_loop	_ttl	Outgoing Interface?	Local Network?	Remote Networks?	Other Interfaces?
1	0	•			
1	1	•	•		
1	>1	•	•	•	see text

图12-23 环回和TTL对多播辖域的影响

图12-23显示了根据发送的环回策略，指定的 TTL值接收多播分组的接口的设置。如果硬件接收自己的发送，则不管采用什么环回策略，都接收分组。数据报可能通过选路穿过该网络，并到达与系统相连的其他接口 (习题12.6)。如果发送系统本身是一个多播路由器，输出的分组可能被转发到其他接口，但是，只有一个接口接受它们进行输入处理 (第14章)。

12.11 加入一个IP多播组

除了内核自动加入(图6-17)的IP所有主机组外，其他组成员是由进程明确发出请求产生的。加入(或离开)多播组选项比其他选项更多使用。必须修改接口的 in_multi表以及其他链路层多播结构，如我们在以太网中讨论的ether_multi。

当optname是IP_ADDMEMBERSHIP时，mbuf中的数据是一个如图 12-24所示的ip_mreq结构。

in.h

```
148 struct ip_mreq {
149     struct in_addr imr_multiaddr;   /* IP multicast address of group */
150     struct in_addr imr_interface;   /* local IP address of interface */
151 };
```

in.h

图12-24 ip_mreq 结构

148-151 imr_multiaddr指定多播组，imr_interface用相关的单播IP地址指定接口。ip_mreq结构指定{接口，组}对表示成员的变化。

图12-25显示了加入和离开与我们的以太网接口例子相关的多播组时，所调用的函数。

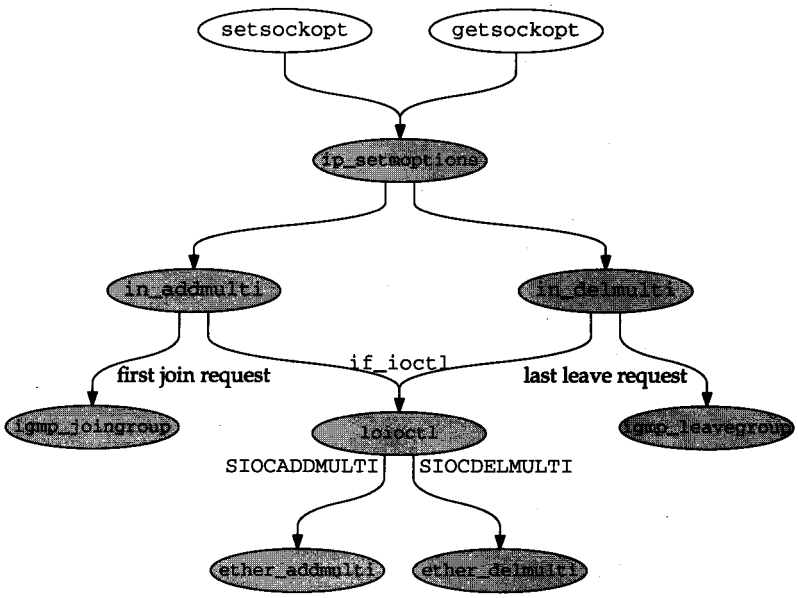


图12-25 加入和离开一个多播组

我们从ip_setmoptions(图12-26)的IP_ADD_MEMBERSHIP情况开始，在这里修改ip_moptions结构。然后我们跟踪请求通过IP层、以太网驱动程序，一直到物理设备——在这里，是LANCE以太网网卡。

ip_output.c

```
733     case IP_ADD_MEMBERSHIP:
734         /*
735          * Add a multicast group membership.
736          * Group must be a valid IP multicast address.
737          */
738         if (m == NULL || m->m_len != sizeof(struct ip_mreq)) {
739             error = EINVAL;
740             break;
741         }
742         mreq = mtod(m, struct ip_mreq *);
743         if (!IN_MULTICAST(ntohl(mreq->imr_multiaddr.s_addr))) {
744             error = EINVAL;
745             break;
746         }
747         /*
748          * If no interface address was provided, use the interface of
749          * the route to the given multicast address.
750          */
751         if (mreq->imr_interface.s_addr == INADDR_ANY) {
752             ro.ro_rt = NULL;
753             dst = (struct sockaddr_in *) &ro.ro_dst;
754             dst->sin_len = sizeof(*dst);
755             dst->sin_family = AF_INET;
```

图12-26 ip_setmoptions 函数：加入一个多播组

```
756             dst->sin_addr = mreq->imr_multiaddr;
757             rtalloc(&ro);
758             if (ro.ro_rt == NULL) {
759                 error = EADDRNOTAVAIL;
760                 break;
761             }
762             ifp = ro.ro_rt->rt_ifp;
763             rtfree(ro.ro_rt);
764         } else {
765             INADDR_TO_IFP(mreq->imr_interface, ifp);
766         }
767         /*
768          * See if we found an interface, and confirm that it
769          * supports multicast.
770          */
771         if (ifp == NULL || (ifp->if_flags & IFF_MULTICAST) == 0) {
772             error = EADDRNOTAVAIL;
773             break;
774         }
775         /*
776          * See if the membership already exists or if all the
777          * membership slots are full.
778          */
779         for (i = 0; i < imo->imo_num_memberships; ++i) {
780             if (imo->imo_membership[i]->inm_ifp == ifp &&
781                 imo->imo_membership[i]->inm_addr.s_addr
782                 == mreq->imr_multiaddr.s_addr)
783                 break;
784         }
785         if (i < imo->imo_num_memberships) {
786             error = EADDRINUSE;
787             break;
788         }
789         if (i == IP_MAX_MEMBERSHIPS) {
790             error = ETOOMANYREFS;
791             break;
792         }
793         /*
794          * Everything looks good; add a new record to the multicast
795          * address list for the given interface.
796          */
797         if ((imo->imo_membership[i] =
798              in_addmulti(&mreq->imr_multiaddr, ifp)) == NULL) {
799             error = ENOBUFS;
800             break;
801         }
802         ++imo->imo_num_memberships;
803         break;
```

ip_output.c

图12-26 (续)

1. 验证

733-746 ip_setmoptions从验证该请求开始。如果没有传给mbuf，或缓存的大小不对，或结构的地址(imr_multiaddr)不是一个多播组地址，则ip_setmoptions发布ENIVAL。Mreq指向有效ip_mreq地址。

2. 找到接口

747-774 如果接口的单播地址(imr_interface)是INADDR_ANY，则ip_setmoptions

必须找到指定组的默认接口。该多播组构造一个 route结构，作为目的地址，并传给rtalloc，由rtalloc为多播组找到一个路由器。如果没有路由器可用，则请求失败，产生错误EADDRNOTAVAIL。如果找到路由器，则在ifp中保存指向路由器外出接口的指针，而不再需要路由器入口，将其释放。

如果imr_interface不是INADDR_ANY，则请求一个明确的接口。INADDR_TO_IFP宏用请求的单播地址搜索接口。如果没有找到接口或者它支持多播，则请求失败，产生错误EADDRNOTAVAIL。

8.5节描述了route结构，19.2节描述了rtalloc函数，第14章描述了用路由选择表选择多播接口。

3. 已经是成员了？

775-792 对请求做的最后检查是检查imo_membership数组，看看所选接口是否已经是请求组的成员。如果 for循环找到一个匹配，或者成员数组为空，则发布 EADDRINUSE或ETOOMANYREFS，并终止对这个选项的处理。

4. 加入多播组

793-803 此时，请求似乎是合理的了。in_addmulti安排IP开始接收该组的多播数据报。in_addmulti返回的指针指向一个新的或已存在的 in_multi结构(图12-12)，该结构位于接口的多播组表中。这个结构被保存在成员数组中，并且把数组的大小加 1。

12.11.1 in_addmulti函数

in_addmulti和相应的in_delmulti(图12-27和图12-36)维护接口已加入多播组的表。加入请求或者在接口表中增加一个新的 in_multi结构，或者增加对某个已有结构的引用次数。

in.c

```
469 struct in_multi *
470 in_addmulti(ap, ifp)
471 struct in_addr *ap;
472 struct ifnet *ifp;
473 {
474     struct in_multi *inm;
475     struct ifreq ifr;
476     struct in_ifaddr *ia;
477     int      s = splnet();
478     /*
479      * See if address already in list.
480      */
481     IN_LOOKUP_MULTI(*ap, ifp, inm);
482     if (inm != NULL) {
483         /*
484          * Found it; just increment the reference count.
485          */
486         ++inm->inm_refcount;
487     } else {
```

in.c

图12-27 in_addmulti 函数：前半部分

1. 已经是一个成员了

469-487 ip_setmoptions已经证实ap指向一个D类多播地址，ifp指向一个能够多播的

接口。IN_LOOKUP_MULTI(图12-14)确定接口是否已经是该组的一个成员。如果是，则in_addmulti更新引用计数后返回。

如果接口还不是该组的成员，则执行图12-28中的程序。

in.c

```
487     } else {
488         /*
489          * New address; allocate a new multicast record
490          * and link it into the interface's multicast list.
491          */
492         inm = (struct in_multi *) malloc(sizeof(*inm),
493                                          M_IPMADDR, M_NOWAIT);
494         if (inm == NULL) {
495             splx(s);
496             return (NULL);
497         }
498         inm->inm_addr = *ap;
499         inm->inm_ifp = ifp;
500         inm->inm_refcount = 1;
501         IFP_TO_IA(ifp, ia);
502         if (ia == NULL) {
503             free(inm, M_IPMADDR);
504             splx(s);
505             return (NULL);
506         }
507         inm->inm_ia = ia;
508         inm->inm_next = ia->ia_multiaddrs;
509         ia->ia_multiaddrs = inm;
510         /*
511          * Ask the network driver to update its multicast reception
512          * filter appropriately for the new address.
513          */
514         ((struct sockaddr_in *) &ifr.ifr_addr)->sin_family = AF_INET;
515         ((struct sockaddr_in *) &ifr.ifr_addr)->sin_addr = *ap;
516         if ((ifp->if_ioctl == NULL) ||
517             (*ifp->if_ioctl) (ifp, SIOCADDMULTI, (caddr_t) & ifr) != 0) {
518             ia->ia_multiaddrs = inm->inm_next;
519             free(inm, M_IPMADDR);
520             splx(s);
521             return (NULL);
522         }
523         /*
524          * Let IGMP know that we have joined a new IP multicast group.
525          */
526         igmp_joingroup(inm);
527     }
528     splx(s);
529     return (inm);
530 }
```

in.c

图12-28 in_addmulti 函数：后半部分

2. 更新in_multi表

487-509 如果接口还不是成员，则in_addmulti分配并初始化一个新的in_multi结构，把该结构插到接口的in_ifaddr(图12-13)结构中ia_multiaddrs表的前端。

3. 更新接口，通告变化

510-530 如果接口驱动程序已经定义了一个if_ioctl函数，则in_addmulti构造一个

包含了该组地址的ifreq结构(图4-23)，并把SIOCADDMULTI请求传给接口。如果接口拒绝该请求，则把in_multi结构从链表中断开，释放掉。最后，in_addmulti调用igmp_joingroup，把成员变化信息传播给其他主机和路由器。

in_addmulti返回一个指针，该指针指向in_multi结构，或者如果出错，则为空。

12.11.2 slioctl和loioctl函数：SIOCADDMULTI和SIOCDELMULTI

SLIP和环回接口的多播组处理很简单：除了检查差错外，不做其他事情。图 12-29显示了SLIP处理。

if_sl.c

```
673     case SIOCADDMULTI:
674     case SIOCDELMULTI:
675         ifr = (struct ifreq *) data;
676         if (ifr == 0) {
677             error = EAFNOSUPPORT;    /* XXX */
678             break;
679         }
680         switch (ifr->ifr_addr.sa_family) {

681         case AF_INET:
682             break;

683         default:
684             error = EAFNOSUPPORT;
685             break;
686         }
687         break;
```

if_sl.c

图12-29 slioctl 函数：多播处理

673-687 不管请求为空还是不适用于AF_INET协议族，都返回EAFNOSUPPORT。

图12-30显示了环回处理。

if_loop.c

```
152     case SIOCADDMULTI:
153     case SIOCDELMULTI:
154         ifr = (struct ifreq *) data;
155         if (ifr == 0) {
156             error = EAFNOSUPPORT;    /* XXX */
157             break;
158         }
159         switch (ifr->ifr_addr.sa_family) {

160         case AF_INET:
161             break;

162         default:
163             error = EAFNOSUPPORT;
164             break;
165         }
166         break;
```

if_loop.c

图12-30 lioctl 函数：多播处理

152-166 环回接口的处理等价于图 12-29中SLIP的程序。不管请求为空还是不适用于AF_INET协议族，都返回EAFNOSUPPORT。

12.11.3 leioctl函数：SIOCADDMULTI和SIOCDELMULTI

在图4-2中，我们讲到LANCE以太网驱动程序的leioctl和if_ioctl函数。图12-31是处理SIOCADDMULTI和SIOCDELMULTI的程序。

if_le.c

```
657     case SIOCADDMULTI:
658     case SIOCDELMULTI:
659         /* Update our multicast list  */
660         error = (cmd == SIOCADDMULTI) ?
661             ether_addmulti((struct ifreq *) data, &le->sc_ac) :
662             ether_delmulti((struct ifreq *) data, &le->sc_ac);
663         if (error == ENETRESET) {
664             /*
665              * Multicast list has changed; set the hardware
666              * filter accordingly.
667              */
668             lereset(ifp->if_unit);
669             error = 0;
670         }
671         break;
```

if_le.c

图12-31 leioctl 函数：多播处理

657-671 leioctl把增加和删除请求直接传给ether_addmulti或ether_delmulti函数。如果请求改变了该物理硬件必须接收的 IP多播地址集，则两个函数都返回 ENETRESET。如果发生了这种情况，则 leioctl调用lereset，用新的多播接收表重新初始化该硬件。

我们没有显示 lereset，因为它是LANCE以太网硬件专用的。对多播来说，lereset安排硬件接收所有寻址到ether_multi中与该接口相关的多播地址的帧。如果多播表中的每个入口是一个地址，则 LANCE驱动程序采用散列机制。散列程序使硬件可以有选择地接收分组。如果驱动程序发现某个入口是一个地址范围，它废除散列策略，配置硬件接收所有多播分组。如果驱动程序必须回到接收所有以太网多播地址的状态，lereset就在返回时把IFP_ALLMULTI标志位置位。

12.11.4 ether_addmulti函数

所有以太网驱动程序都调用 ether_addmulti函数处理SIOCADDMULTI请求。这个函数把IP D类地址映射到合适的以太网多播地址 (图12-5)上，并更新ether_multi表。图12-32是ether_multi函数的前半部。

1. 初始化地址范围

366-399 首先，ether_addmulti初始化addrlo和addrhi(两者都是六个无符号字符)中的多播地址范围。如果所请求的地址来自 AF_UNSPEC族，ether_addmulti假定该地址是一个明确的以太网多播地址，并把它复制到 addrlo和addrhi中。如果地址属于AF_INET族，并且是 INADDR_ANY (0.0.0.0)，ether_addmulti把 addrlo初始化成ether_ipmulticast_min，把addrhi初始化成ether_ipmulticast_max。这两个以太网地址常量定义为：

```
u_char ether_ipmulticast_min[6] = { 0x01, 0x00, 0x5e, 0x00, 0x00, 0x00 };
u_char ether_ipmulticast_max[6] = { 0x01, 0x00, 0x5e, 0x7f, 0xff, 0xff };
```

if_ethersubr.c

```
366 int
367 ether_addmulti(ifr, ac)
368 struct ifreq *ifr;
369 struct arpcom *ac;
370 {
371     struct ether_multi *enm;
372     struct sockaddr_in *sin;
373     u_char  addrlo[6];
374     u_char  addrhi[6];
375     int     s = splimp();

376     switch (ifr->ifr_addr.sa_family) {

377     case AF_UNSPEC:
378         bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
379         bcopy(addrlo, addrhi, 6);
380         break;

381     case AF_INET:
382         sin = (struct sockaddr_in *) &(ifr->ifr_addr);
383         if (sin->sin_addr.s_addr == INADDR_ANY) {
384             /*
385              * An IP address of INADDR_ANY means listen to all
386              * of the Ethernet multicast addresses used for IP.
387              * (This is for the sake of IP multicast routers.)
388              */
389             bcopy(ether_ipmulticast_min, addrlo, 6);
390             bcopy(ether_ipmulticast_max, addrhi, 6);
391         } else {
392             ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
393             bcopy(addrlo, addrhi, 6);
394         }
395         break;

396     default:
397         splx(s);
398         return (EAFNOSUPPORT);
399     }
```

if_ethersubr.c

图12-32 ether_addmulti 函数：前一半

与etherbroadcastaddr(4.3节)一样，这是一个很方便地定义一个48 bit常量的方法。

IP多播路由器必须监听所有IP多播。把组指定为INADDR_ANY，被认为是请求加入所有IP多播组。在这种情况下，所选择的以太网地址范围跨越了分配给 IANA的整个IP多播地址块。

当mrouted (8)守护程序开始对到多播接口的分组进行路选时，它用INADDR_ANY发布一个SIOCADDMULTI请求。

ETHER_MAP_IP_MULTICAST把其他特定的IP多播组映射到合适的以太网多播地址。当发生EAFNOSUPPORT错误时，将拒绝对其他地址族的请求。

尽管以太网多播表支持地址范围，但是除了列举出所有地址外，进程或内核无法对某个特定范围提出请求，因为总是把addrlo和addrhi设成同一值。

ether_addmulti的第二部分，显示如图12-33，证实地址范围，并且，如果该地址是

新的，就把它加入表中。

if_ethersubr.c

```
400     /*
401      * Verify that we have valid Ethernet multicast addresses.
402      */
403     if ((addrlo[0] & 0x01) != 1 || (addrhi[0] & 0x01) != 1) {
404         splx(s);
405         return (EINVAL);
406     }
407     /*
408      * See if the address range is already in the list.
409      */
410     ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm);
411     if (enm != NULL) {
412         /*
413          * Found it; just increment the reference count.
414          */
415         ++enm->enm_refcount;
416         splx(s);
417         return (0);
418     }
419     /*
420      * New address or range; malloc a new multicast record
421      * and link it into the interface's multicast list.
422      */
423     enm = (struct ether_multi *) malloc(sizeof(*enm), M_IFMADDR, M_NOWAIT);
424     if (enm == NULL) {
425         splx(s);
426         return (ENOBUFS);
427     }
428     bcopy(addrlo, enm->enm_addrlo, 6);
429     bcopy(addrhi, enm->enm_addrhi, 6);
430     enm->enm_ac = ac;
431     enm->enm_refcount = 1;
432     enm->enm_next = ac->ac_multiaddrs;
433     ac->ac_multiaddrs = enm;
434     ac->ac_multicnt++;
435     splx(s);
436     /*
437      * Return ENETRESET to inform the driver that the list has changed
438      * and its reception filter should be adjusted accordingly.
439      */
440     return (ENETRESET);
441 }
```

if_ethersubr.c

图12-33 ether_addmulti 函数：后一半

2. 已经在接收

400-418 ether_addmulti检查高地址和低地址的多播比特位(图4-12)，保证它们是真正的以太网多播地址。ETHER_LOOKUP_MULTI(图12-9)确定硬件是否已经对指定的地址开始监听。如果是，则增加匹配的 ether_multi结构中的引用计数 (enm_refcount)，并且ether_addmulti返回0。

3. 更新ether_multi表

419-441 如果这是一个新的地址范围，则分配并初始化一个新的 ether_multi结构，把它链到接口arpcom结构(图12-8)中的ac_multiaddrs表上。如果ether_addmulti返回

ENETRESET，则调用它的设备驱动程序就知道多播表被改变了，必须更新硬件接收过滤器。

图12-34显示在LANCE以太网接口加入所有主机组后，ip_moptions、in_multi和ether_multi结构之间的关系。

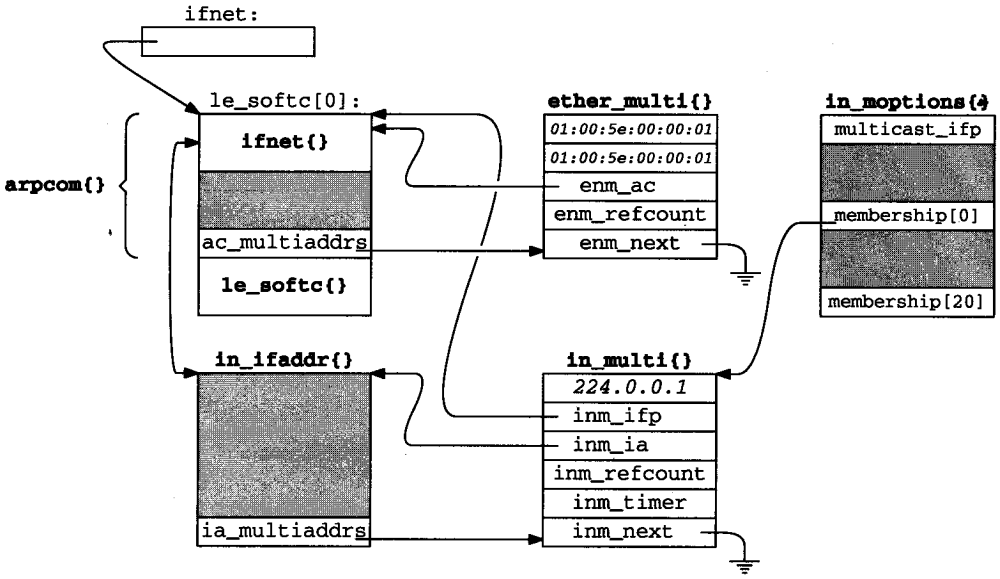


图12-34 多播数据结构的整体图

12.12 离开一个IP多播组

通常情况下，离开一个多播组的步骤是加入一个多播组的步骤的反序。更新ip_moptions结构中的成员表、IP接口的in_multi表和设备的ether_multi表。首先，我们回到ip_setmoptions中的IP_DROP_MEMBERSHIP情况语句，如图12-35所示。

ip_output.c

```
804     case IP_DROP_MEMBERSHIP:
805         /*
806          * Drop a multicast group membership.
807          * Group must be a valid IP multicast address.
808          */
809         if (m == NULL || m->m_len != sizeof(struct ip_mreq)) {
810             error = EINVAL;
811             break;
812         }
813         mreq = mtod(m, struct ip_mreq *);
814         if (!IN_MULTICAST(ntohl(mreq->imr_multiaddr.s_addr))) {
815             error = EINVAL;
816             break;
817         }
818         /*
819          * If an interface address was specified, get a pointer
820          * to its ifnet structure.
821          */
822         if (mreq->imr_interface.s_addr == INADDR_ANY)
823             ifp = NULL;
824         else {
825             INADDR_TO_IFP(mreq->imr_interface, ifp);
```

图12-35 ip_setmoptions 函数：离开一个多播组

```
826             if (ifp == NULL) {
827                 error = EADDRNOTAVAIL;
828                 break;
829             }
830         }
831         /*
832          * Find the membership in the membership array.
833          */
834         for (i = 0; i < imo->imo_num_memberships; ++i) {
835             if ((ifp == NULL ||
836                  imo->imo_membership[i]->inm_ifp == ifp) &&
837                 imo->imo_membership[i]->inm_addr.s_addr ==
838                 mreq->imr_multiaddr.s_addr)
839                 break;
840         }
841         if (i == imo->imo_num_memberships) {
842             error = EADDRNOTAVAIL;
843             break;
844         }
845         /*
846          * Give up the multicast address record to which the
847          * membership points.
848          */
849         in_delmulti(imo->imo_membership[i]);
850         /*
851          * Remove the gap in the membership array.
852          */
853         for (++i; i < imo->imo_num_memberships; ++i)
854             imo->imo_membership[i - 1] = imo->imo_membership[i];
855         --imo->imo_num_memberships;
856         break;
```

ip_output.c

图12-35 (续)

1. 验证

804-830 存储器缓存中必然包含一个ip_mreq结构，其中的imr_multiaddr必须是一个多播组，而且必须有一个接口与单播地址imr_interface相关。如果这些条件不满足，则发布EINVAL和EADDRNOTAVAIL错误信息，继续到该switch语句的最后进行处理。

2. 删除成员引用

831-856 for循环用请求的{接口，组}对在组成员表中寻找一个in_multi结构。如果没有找到，则发布EADDRNOTAVAIL错误信息。如果找到了，则in_delmulti更新in_multi表，并且第二个for循环把成员数组中不用的入口删去，把后面的入口向前移动。数组的大小也被相应更新。

12.12.1 in_delmulti函数

因为可能会有多个进程接收多播数据报，所以调用in_delmulti(图12-36)的结果是，当对in_multi结构没有引用时，只离开指定的多播组。

更新in_multi结构

534-567 in_delmulti一开始就减少in_multi结构的引用计数，如果该计数非零，则返回。如果该计数减为0，则表明在指定的{接口，组}对上，没有其他进程等待多播数据报。调用igmp_leavegroup，但该函数不做任何事情，我们将在13.8节中看到。

for循环遍历in_multi结构的链表，找到匹配的结构。

in.c

```
534 int
535 in_delmulti(inm)
536 struct in_multi *inm;
537 {
538     struct in_multi **p;
539     struct ifreq ifr;
540     int     s = splnet();

541     if (--inm->inm_refcount == 0) {
542         /*
543          * No remaining claims to this record; let IGMP know that
544          * we are leaving the multicast group.
545          */
546         igmp_leavegroup(inm);
547         /*
548          * Unlink from list.
549          */
550         for (p = &inm->inm_ia->ia_multiaddrs;
551              *p != inm;
552              p = &(*p)->inm_next)
553             continue;
554         *p = (*p)->inm_next;
555         /*
556          * Notify the network driver to update its multicast reception
557          * filter.
558          */
559         ((struct sockaddr_in *) &(ifr.ifr_addr))->sin_family = AF_INET;
560         ((struct sockaddr_in *) &(ifr.ifr_addr))->sin_addr =
561             inm->inm_addr;
562         (*inm->inm_ifp->if_ioctl) (inm->inm_ifp, SIOCDELMULTI,
563                                    (caddr_t) & ifr);
564         free(inm, M_IPMADDR);
565     }
566     splx(s);
567 }
```

in.c

图12-36 in_delmulti 函数

for循环体只包含一个continue语句。但所有工作都由循环上面的表达式做了，并不需要continue语句，只是因为它比只有一个分号更清楚一些。

图12-9中的宏ETHER_LOOKUP_MULTI不用continue语句，仅有一个分号几乎是不可检测的。

循环结束后，把匹配的 in_multi结构从链表上断开， in_delmulti向接口发布SIOCDELMULTI请求，以便更新所有设备专用的数据结构。对以太网接口来说，这意味着更新ether_multi表。最后释放in_multi结构。

LANCE驱动程序的SIOCDELMULTI情况语句包括在图12-31中，这里我们也讨论了SIOCADDRMULTI情况。

12.12.2 ether_delmulti函数

当IP释放与某个以太网设备相关的 in_multi结构时，该设备也可能释放匹配的ether_multi结构。我们说“可能”是因为 IP忽略其他监听IP多播的软件。当 ether_

multi结构的引用计数变成0时，就释放该结构。图12-37是ether_delmulti函数。

if_ethersubr.c

```
445 int
446 ether_delmulti(ifr, ac)
447 struct ifreq *ifr;
448 struct arpcom *ac;
449 {
450     struct ether_multi *enm;
451     struct ether_multi **p;
452     struct sockaddr_in *sin;
453     u_char  addrlo[6];
454     u_char  addrhi[6];
455     int     s = splimp();

456     switch (ifr->ifr_addr.sa_family) {

457     case AF_UNSPEC:
458         bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
459         bcopy(addrlo, addrhi, 6);
460         break;

461     case AF_INET:
462         sin = (struct sockaddr_in *) &(ifr->ifr_addr);
463         if (sin->sin_addr.s_addr == INADDR_ANY) {
464             /*
465              * An IP address of INADDR_ANY means stop listening
466              * to the range of Ethernet multicast addresses used
467              * for IP.
468              */
469             bcopy(ether_ipmulticast_min, addrlo, 6);
470             bcopy(ether_ipmulticast_max, addrhi, 6);
471         } else {
472             ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
473             bcopy(addrlo, addrhi, 6);
474         }
475         break;

476     default:
477         splx(s);
478         return (EAFNOSUPPORT);
479     }

480     /*
481      * Look up the address in our list.
482      */
483     ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm);
484     if (enm == NULL) {
485         splx(s);
486         return (ENXIO);
487     }
488     if (--enm->enm_refcount != 0) {
489         /*
490          * Still some claims to this record.
491          */
492         splx(s);
493         return (0);
494     }
495     /*
496      * No remaining claims to this record; unlink and free it.
497      */
```

图12-37 ether_delmulti 函数

```
498     for (p = &enm->enm_ac->ac_multiaddrs;
499          *p != enm;
500          p = &(*p)->enm_next)
501         continue;
502     *p = (*p)->enm_next;
503     free(enm, M_IFMADDR);
504     ac->ac_multicnt--;
505     splx(s);
506     /*
507      * Return ENETRESET to inform the driver that the list has changed
508      * and its reception filter should be adjusted accordingly.
509      */
510     return (ENETRESET);
511 }
```

if_ethersubr.c

图12-37 (续)

445-479 ether_delmulti函数用ether_addrmulti函数采用的同一方法初始化addrlo和addrhi数组。

1. 寻找ether_multi结构

480-494 ETHER_LOOKUP_MULTI寻找匹配的ether_multi结构。如果没有找到，则返回ENXIO。如果找到匹配的结构，则把引用计数减去 1。如果此时引用计数非零，ether_delmulti立即返回。在这种情况下，可能会由于其他协议也要接收相同的多播分组而释放该结构。

2. 删除ether_multi结构

495-511 for循环搜索ether_multi表，寻找匹配的地址范围，并从链表中断开匹配的结构，将它释放掉。最后，更新链表的长度，返回 ENETRESET，使设备驱动程序可以更新它的硬件接收过滤器。

12.13 ip_getmoptions函数

取得当前的选项设置比设置它们要容易。ip_getmoptions完成所有的工作，如图12-38所示。

复制选项数据和返回

876-914 ip_getmoptions的三个参数是： optname，要取得的选项； imo，ip_moptions结构；mp，一个指向mbuf的指针。m_get分配一个mbuf存放该选项数据。这三个选项的指针(分别是addr、ttl和loop)被初始化为指向mbuf的数据域，而mbuf的长度被设成选项数据的长度。

对IP_MULTICAST_IF，返回IFP_TO_IA发现的单播地址，或者如果没有选择明确的多播接口，则返回INADDR_ANY。

对IP_MULTICAST_TTL，返回imo_multicast_ttl，或者如果没有选择明确的TTL，则返回1(IP_DEFAULT_MULTICAST_TTL)。

对IP_MULTICAST_LOOP，返回imo_multicast_loop，或者如果没有选择明确的多播环回策略，则返回1(IP_DEFAULT_MULTICAST_LOOP)。

最后，如果不识别该选项，则返回EOPNOTSUPP。

ip_output.c

```
876 int
877 ip_getmoptions(optname, imo, mp)
878 int     optname;
879 struct ip_moptions *imo;
880 struct mbuf **mp;
881 {
882     u_char *ttl;
883     u_char *loop;
884     struct in_addr *addr;
885     struct in_ifaddr *ia;

886     *mp = m_get(M_WAIT, MT_SOOPTS);

887     switch (optname) {

888     case IP_MULTICAST_IF:
889         addr = mtod(*mp, struct in_addr *);
890         (*mp)->m_len = sizeof(struct in_addr);
891         if (imo == NULL || imo->imo_multicast_ifp == NULL)
892             addr->s_addr = INADDR_ANY;
893         else {
894             IFP_TO_IA(imo->imo_multicast_ifp, ia);
895             addr->s_addr = (ia == NULL) ? INADDR_ANY
896                 : IA_SIN(ia)->sin_addr.s_addr;
897         }
898         return (0);

899     case IP_MULTICAST_TTL:
900         ttl = mtod(*mp, u_char *);
901         (*mp)->m_len = 1;
902         *ttl = (imo == NULL) ? IP_DEFAULT_MULTICAST_TTL
903             : imo->imo_multicast_ttl;
904         return (0);

905     case IP_MULTICAST_LOOP:
906         loop = mtod(*mp, u_char *);
907         (*mp)->m_len = 1;
908         *loop = (imo == NULL) ? IP_DEFAULT_MULTICAST_LOOP
909             : imo->imo_multicast_loop;
910         return (0);

911     default:
912         return (EOPNOTSUPP);
913     }
914 }
```

ip_output.c

图12-38 ip_getmoptions 函数

12.14 多播输入处理：ipintr函数

到目前为止，我们已经讨论了多播选路，组成员关系，以及多种与 IP和以太网多播有关的数据结构，现在转入讨论对多播数据报的处理。

在图4-13中，我们看到ether_input检测到达的以太网多播分组，在把一个 IP分组放到IP输入队列之前(ipintrq)，把mbuf首部的M_MCAST标志位置位。 ipintr函数按顺序处理每个分组。我们在ipintr中省略的多播处理程序如图 12-39所示。

该段代码来自于ipintr程序，用来确定分组是寻址到本地网络还是应该被转发。此时，已经检测到分组中的错误，并且已经处理完分组的所有选项。 ip指向分组内的IP首部。

如果被配置成多播路由器，就转发分组

214-245 如果目的地址不是一个IP多播组，则跳过整个这部分代码。如果地址是一个多播组，并且系统被配置成 IP多播路由器(ip_mrouter)，就把ip_id转换成网络字节序(ip_mforward希望的格式)，并把分组传给ip_mforward。如果出现错误或者分组是通过一个多播隧道(multicast tunnel)到达的，则ip_mforward返回一个非零值。分组被丢弃，且ips_cantforward的值加1。

我们在第14章中描述了多播隧道。它们在两个被标准IP路由器隔开的多播路由器之间传递分组。通过隧道到达的分组必须由ip_mforward处理，而不是由ipintr处理。

如果ip_mforward返回0，则把ip_id转换回主机字节序，由ipintr继续处理分组。

如果ip指向一个IGMP分组，则接受该分组，并在ours处(图10-11的ipintr)继续执行。不管到达接口的每个目的组或组成员是什么，多播路由器必须接受所有 IGMP分组。IGMP分组中有组成员变化的信息。

246-257 根据系统是否被配置成多播路由器来确定是否执行图 12-39中的其余程序。IN_LOOKUP_MULTI搜索接口加入的多播组表。如果没有找到匹配，则丢弃该分组。当硬件过滤器接受不需要的分组时，或者当与接口相关的多播组与分组中的目的多播地址映射到同一个以太网地址时，才出现这种情况。

如果接受了该分组，就继续执行ipintr(图10-11)的ours标号处的语句。

ip_input.c

```
214     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {
215         struct in_multi *inm;
216         extern struct socket *ip_mrouter;

217         if (ip_mrouter) {
218             /*
219              * If we are acting as a multicast router, all
220              * incoming multicast packets are passed to the
221              * kernel-level multicast forwarding function.
222              * The packet is returned (relatively) intact; if
223              * ip_mforward() returns a non-zero value, the packet
224              * must be discarded, else it may be accepted below.
225              *
226              * (The IP ident field is put in the same byte order
227              * as expected when ip_mforward() is called from
228              * ip_output().)
229              */
230             ip->ip_id = htons(ip->ip_id);
231             if (ip_mforward(m, m->m_pkthdr.rcvif) != 0) {
232                 ipstat.ips_cantforward++;
233                 m_freem(m);
234                 goto next;
235             }
236             ip->ip_id = ntohs(ip->ip_id);

237             /*
238              * The process-level routing demon needs to receive
239              * all multicast IGMP packets, whether or not this
240              * host belongs to their destination groups.
241              */
```

图12-39 ipintr 函数：多播输入处理

```
242             if (ip->ip_p == IPPROTO_IGMP)
243                 goto ours;
244             ipstat.ips_forward++;
245         }
246         /*
247          * See if we belong to the destination multicast group on the
248          * arrival interface.
249          */
250         IN_LOOKUP_MULTI(ip->ip_dst, m->m_pkthdr.rcvif, inm);
251         if (inm == NULL) {
252             ipstat.ips_cantforward++;
253             m_freem(m);
254             goto next;
255         }
256         goto ours;
257     }
```

ip_input.c

图12-39 (续)

12.15 多播输出处理：ip_output函数

当我们在第8章讨论ip_output时，推迟了对ip_output的mp参数和多播处理程序的讨论。在ip_output中，如果mp指向一个ip_moptions结构，它就覆盖多播输出处理的默认值。ip_output中省略的程序在图12-40和图12-41中显示。ip指向输出的分组，m指向包含该分组的mbuf，ifp指向路由表为目的多播组选择的接口。

ip_output.c

```
129     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {
130         struct in_multi *inm;
131         extern struct ifnet loif;
132         m->m_flags |= M_MCAST;
133         /*
134          * IP destination address is multicast.  Make sure "dst"
135          * still points to the address in "ro".  (It may have been
136          * changed to point to a gateway address, above.)
137          */
138         dst = (struct sockaddr_in *) &ro->ro_dst;
139         /*
140          * See if the caller provided any multicast options
141          */
142         if (imo != NULL) {
143             ip->ip_ttl = imo->imo_multicast_ttl;
144             if (imo->imo_multicast_ifp != NULL)
145                 ifp = imo->imo_multicast_ifp;
146         } else
147             ip->ip_ttl = IP_DEFAULT_MULTICAST_TTL;
148         /*
149          * Confirm that the outgoing interface supports multicast.
150          */
151         if ((ifp->if_flags & IFF_MULTICAST) == 0) {
152             ipstat.ips_noroute++;
153             error = ENETUNREACH;
154             goto bad;
155         }
156         /*
```

图12-40 ip_output 函数：默认和源地址

```
157          * If source address not specified yet, use address
158          * of outgoing interface.
159          */
160         if (ip->ip_src.s_addr == INADDR_ANY) {
161             struct in_ifaddr *ia;
162             for (ia = in_ifaddr; ia; ia = ia->ia_next)
163                 if (ia->ia_ifp == ifp) {
164                     ip->ip_src = IA_SIN(ia)->sin_addr;
165                     break;
166                 }
167         }
```

ip_output.c

图12-40 (续)

ip_output.c

```
168         IN_LOOKUP_MULTI(ip->ip_dst, ifp, inm);
169         if (inm != NULL &&
170             (imo == NULL || imo->imo_multicast_loop)) {
171             /*
172              * If we belong to the destination multicast group
173              * on the outgoing interface, and the caller did not
174              * forbid loopback, loop back a copy.
175              */
176             ip_mloopback(ifp, m, dst);
177         } else {
178             /*
179              * If we are acting as a multicast router, perform
180              * multicast forwarding as if the packet had just
181              * arrived on the interface to which we are about
182              * to send.  The multicast forwarding function
183              * recursively calls this function, using the
184              * IP_FORWARDING flag to prevent infinite recursion.
185              *
186              * Multicasts that are looped back by ip_mloopback(),
187              * above, will be forwarded by the ip_input() routine,
188              * if necessary.
189              */
190             extern struct socket *ip_mrouter;
191             if (ip_mrouter && (flags & IP_FORWARDING) == 0) {
192                 if (ip_mforward(m, ifp) != 0) {
193                     m_freem(m);
194                     goto done;
195                 }
196             }
197         }
198         /*
199          * Multicasts with a time-to-live of zero may be looped-
200          * back, above, but must not be transmitted on a network.
201          * Also, multicasts addressed to the loopback interface
202          * are not sent -- the above call to ip_mloopback() will
203          * loop back a copy if this host actually belongs to the
204          * destination group on the loopback interface.
205          */
206         if (ip->ip_ttl == 0 || ifp == &loif) {
207             m_freem(m);
208             goto done;
209         }
210         goto sendit;
211     }
```

ip_output.c

图12-41 ip_output 函数：环回、转发和发送

1. 建立默认值

129-155 只有分组是到一个多播组时，才执行图 12-40中的程序。此时，ip_output把mbuf中的M_MCAST置位，并把dst重设成最终目的地址，因为ip_output可能曾把它设成下一跳路由器(图8-24)。

如果传递了一个ip_moptions结构，则相应地改变ip_ttl和ifp。否则，把ip_ttl设成1(IP_DEFAULT_MULTICAST_TTL)，避免多播分组到达某个远程网络。查询路由表或ip_moptions结构所得到的接口必须支持多播。如果不支持，则 ip_output丢弃该分组，并返回ENETUNREACH。

2. 选择源地址

156-167 如果没有指定源地址，则由 for循环找到与输出接口相关的单播地址，并填入 IP首部的ip_src。

与单播分组不同，如果系统被配置成一个多播路由器，则必须在一个以上的接口上发送输出的多播分组。即使系统不是一个多播路由器，输出的接口也可能是目的多播组的一个成员，也会需要接收该分组。最后，我们需要考虑一下多播环回策略和环回接口本身。把所有这些都考虑进去，共有三个问题：

- 是否要在输出的接口上接收该分组？
- 是否向其他接口转发该分组？
- 是否在出去的接口发送该分组？

图12-41显示了ip_output中解决这三个问题的程序。

3. 是否环回？

168-176 如果 IN_LOOKUP_MULTI确定输出的接口是目的多播组的成员，而且imo_multicast_loop非零，则分组被ip_mloopback放到输出接口上排队，等待输入。在这种情况下，不考虑转发原始分组，因为在输入过程中如果需要，分组的复制会被转发的。

4. 是否转发？

178-197 如果分组不是环回的，但系统被配置成一个多播路由器，并且分组符合转发的条件，则ip_mforward向其他多播接口分发该分组的备份。如果 ip_mforward没有返回0，则ip_output丢弃该分组，不发送它。这表明分组中有错误。

为了避免 ip_mforward和ip_output之间的无限循环， ip_mforward在调用ip_output之前，总是把IP_FORWARDING打开。在本系统上产生的数据报是符合转发条件的，因为运输层不打开IF_FORWARDING。

5. 是否发送？

198-209 TTL是0的分组可能被环回，但从不转发它们(ip_mforward丢弃它们)，也从不被发送。如果TTL是0或者如果输出接口是环回接口，则 ip_output丢弃该分组，因为TTL超时，或者分组已经被ip_mloopback环回了。

6. 发送分组

210-211 到这个时候，分组应该已经从物理上在输出接口上被发送了。sendit(ip_output，图8-25)处的程序在把分组传给接口的if_output函数之前可能已经把它分片了。我们将在21.10节中看到，以太网输出函数ether_output调用arpresolve，

arpresolve又调用ETHER_MAP_MULTICAST，由ETHER_MAP_MULTICAST根据IP多播目的地址构造一个以太网多播目的地址。

ip_mloopback函数

ip_mloopback依靠looutput(图5-27)完成它的工作。ip_mloopback传递的looutput不是指向环回接口的指针，而是指向输出多播接口的指针。图 12-42显示了ip_mloopback函数。

ip_output.c

```
935 static void
936 ip_mloopback(ifp, m, dst)
937 struct ifnet *ifp;
938 struct mbuf *m;
939 struct sockaddr_in *dst;
940 {
941     struct ip *ip;
942     struct mbuf *copym;

943     copym = m_copy(m, 0, M_COPYALL);
944     if (copym != NULL) {
945         /*
946          * We don't bother to fragment if the IP length is greater
947          * than the interface's MTU.  Can this possibly matter?
948          */
949         ip = mtod(copym, struct ip *);
950         ip->ip_len = htons((u_short) ip->ip_len);
951         ip->ip_off = htons((u_short) ip->ip_off);
952         ip->ip_sum = 0;
953         ip->ip_sum = in_cksum(copym, ip->ip_hl << 2);
954         (void) looutput(ifp, copym, (struct sockaddr *) dst, NULL);
955     }
956 }
```

ip_output.c

图12-42 ip_mloopback 函数

复制并把分组放到队列中

929-956 仅仅复制分组是不够的；必须看起来分组已经被输出接口接收了，所以ip_mloopback把ip_len和ip_off转换成网络字节序，并计算分组的检验和。looutput把分组放到IP输入队列。

12.16 性能的考虑

Net/3的多播实现有几个潜在的性能瓶颈。因为许多以太网网卡并不能完美地实现对多播地址的过滤，所以操作系统必须能够丢弃那些通过硬件过滤器的分组。在最坏的情况下，以太网网卡可能会接收所有分组，而其中大部分可能会被 ipintr发现不具有合法的IP多播组地址。

IP用简单的线性表和线性搜索过滤到达的 IP数据报。如果表增长到一定长度后，某些高速缓存技术，如移动最近接收地址到表的最前面，将有助于提高性能。

12.17 小结

本章我们讨论了一个主机如何处理 IP多播数据报。我们看到，在IP的D类地址和以太网多

播地址的格式及它们之间的映射关系。

我们讨论了in-multi和ether_multi结构，每个IP多播接口都维护一个它自己的组成员表，而每个以太网接口都维护一个以太网多播地址。

在输入处理中，只有到达接口是目的多播组的成员时，该 IP多播才被接受下来。尽管如果系统被配置成多播路由器，它们也可能被继续转发到其他接口。

被配置成多播路由器的系统必须接受所有接口上的所有多播分组。只要为 INADDR_ANY地址发布SIOCADDMULTI命令，就可以迅速做到这一点。

ip_moptions结构是多播输出处理的基础。它控制对输出接口的选择、多播数据报 TTL辖域值的设置以及环回策略。它也控制对 in_multi结构的引用计数，从而决定接口加入或离开某个IP多播组的时机。

我们也讨论了多播TTL值实现的两个概念：分组生存期和分组辖域。

习题

12.1 发送IP广播分组到255.255.255.255和发送IP多播给所有主机组 224.0.0.1的区别是什么？

12.2 为什么用多播代码中的IP单播地址标识接口？如果接口能发送和接收多播地址，但没有一个单播IP地址，必须做什么改动？

12.3 在12.3节中，我们讲到32个IP组地址被映射到同一个以太网地址上。因为 32 bit地址中的9 bit不在映射中。为什么我们不说 512(2^9)个IP 组被映射到一个以太网地址上？

12.4 你认为为什么把IP_MAX_MEMBERSHIPS设成20？能被设得更大一些吗？提示：考虑ip_moptions结构(图12-15)的大小。

12.5 当一个多播数据报被 IP环回并且被发送它的硬件接口接收 (即一个非单工接口)时，会发生什么情况？

12.6 画一个有一个多接口主机的网络图，即使该主机没有被配置成多播路由器，其他接口也能接收到在某个接口上发送的多播分组。

12.7 通过SLIP和环回接口而不是以太网接口跟踪成员增加请求。

12.8 进程如何请求内核加入多于IP_MAX_MEMBERSHIPS个组？

12.9 计算环回分组的检验和是多余的。设计一个方法，避免计算环回分组的检验和。

12.10 接口在不重用以太网地址的情况下，最多可加入多少个 IP多播组中？

12.11 细心的读者可能已经注意到 in_delmulti在发布SIOCDELMULTI请求时，假定接口已经定义了ioctl函数。为什么这样不会出错？

12.12 如果请求一个未识别的选项，则 ip_getmoptions中分配的mbuf将会发生什么情况？

12.13 为什么把组成员机制与用于接收单播和广播数据报的绑定机制分离开来？

第13章　IGMP：Internet组管理协议

13.1　引言

IGMP在本地网络上的主机和路由器之间传达组成员信息。路由器定时向“所有主机组”多播IGMP查询。主机多播IGMP报告报文以响应查询。IGMP规范在RFC 1112中。卷1的第13章讨论了IGMP的规范，并给出了一些例子。

从体系结构的观点来看，IGMP是位于IP上面的运输层协议。它有一个协议号(2)，它的报文是由IP数据报运载的(与ICMP一样)。与ICMP一样，进程通常不直接访问IGMP，但进程可以通过IGMP插口发送或接收IGMP报文。这个特性使得能够把多播选路守护程序作为用户级进程实现。

图13-1显示了Net/3中IGMP协议的整体结构。

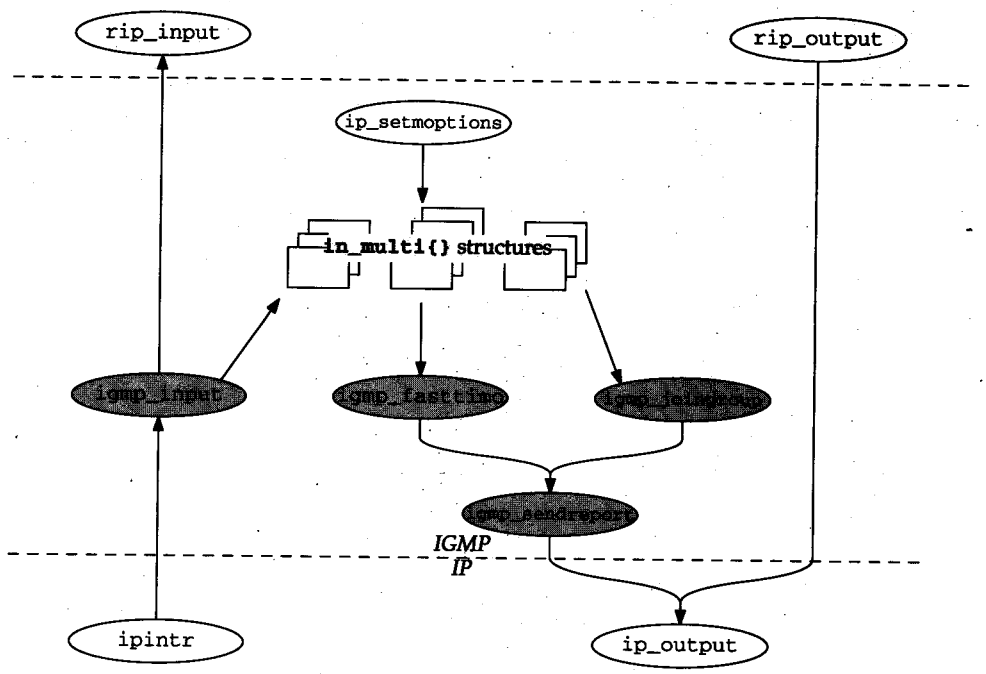


图13-1　IGMP处理概要

IGMP处理的关键是一组在图 13-1中心显示的 `in_multi`结构。到达的IGMP查询使`igmp_input`为每个`in_multi`结构初始化一个递减定时器。该定时器由 `igmp_fasttimo`更新，当每个定时器超时时，`igmp_fasttimo`调用`igmp_sendreport`。

我们在第12章中看到，当创建一个新的 `in_multi`结构时，`ip_setmoptions`调用`igmp_joingroup`。`igmp_joingroup`调用`igmp_sendreport`来发布新的组成员信息，使组的定时器能够在短时间内安排第二次通告。`igmp_sendreport`完成对IGMP报文的格式

化，并把它传给ip_output。

在图13-1的左边和右边，我们看到一个原始插口可以直接发送和接收 IGMP报文。

13.2 代码介绍

图13-2中列出了实现IGMP协议的4个文件。

文　件	描　述
netinet/igmp.h	IGMP协议定义
netinet/igmp_var.h	IGMP实现定义
netinet/in_var.h	IP多播数据结构
netinet/igmp.c	IGMP协议实现

图13-2　本章讨论的文件

13.2.1 全局变量

本章中介绍的新的全局变量显示在图 13-3中。

变　量	数据类型	描　述
igmp_all_hosts_group	u_long	网络字节序的“所有主机组”地址
igmp_timer_are_running	int	如果所有IGMP定时器都有效，则为真；否则为假
igmp_stat	struct igmpstat	IGMP统计(图13-4)

图13-3　本章介绍的全局变量

13.2.2 统计量

IGMP统计信息是在图13-4的igmpstat变量中维护的。

Igmpstat成员	描　述
igps_rcv_badqueries	作为无效查询接收的报文数
igps_rcv_badreports	作为无效报告接收的报文数
igps_rcv_badsum	接收的报文检验和错误数
igps_rcv_ourreports	作为逻辑组的报告接收的报文数
igps_rcv_queries	作为成员关系查询接收的报文数
igps_rcv_reports	作为成员关系报告接收的报文数
igps_rcv_tooshort	字节数太少的报文数
igps_rcv_total	接收的全部报文数
igps snd reports	作为成员关系报告发送的报文数

图13-4　IGMP统计

图13-5是在vangogh.cs.berkeley.edu上执行netstat -p igmp命令后，输出的统计信息。

在图13-5中，我们看到vangogh是连到一个使用IGMP的网络上的，但是vangogh没有加入任何多播组，因为igps_snd_reports是0。

netstat -p igmp 输出	igmpstat 成员
18774 messages received	igps_rcv_total
0 messages received with too few bytes	igps_rcv_tooshort
0 messages received with bad checksum	igps_rcv_badsum
18774 membership queries received	igps_rcv_queries
0 membership queries received with invalid field(s)	igps_rcv_badqueries
0 membership reports received	igps_rcv_reports
0 membership reports received with invalid field(s)	igps_rcv_badreports
0 membership reports received for groups to which we belong	igps_rcv_ourreports
0 membership reports sent	igps_snd_reports

图13-5 IGMP统计示例

13.2.3 SNMP变量

IGMP没有标准的SNMP MIB，但 [McCloghrie Farinacci 1994a]描述了一个IGMP的实验MIB。

13.3 igmp结构

IGMP报文只有8字节长。图13-6显示了Net/3使用的igmp结构。

igmp.h

```
43 struct igmp {
44     u_char  igmp_type;          /* version & type of IGMP message  */
45     u_char  igmp_code;          /* unused, should be zero          */
46     u_short igmp_cksum;         /* IP-style checksum               */
47     struct in_addr igmp_group;  /* group address being reported    */
48 };                              /* (zero for queries)              */
```

igmp.h

图13-6 igmp结构

igmp_type包括一个4 bit的版本码和一个4 bit的类型码。图13-7显示了标准值。

版本	类型	igmp_type	描 述
1	1	0x11(IGMP_HOST_MEMBERSHIP_QUERY)	成员关系查询
1	2	0x11 (IGMP_HOST_MEMBERSHIP_REPORT)	成员关系报告
1	3	0x13	DVMRP报文(第14章)

图13-7 IGMP报文类型

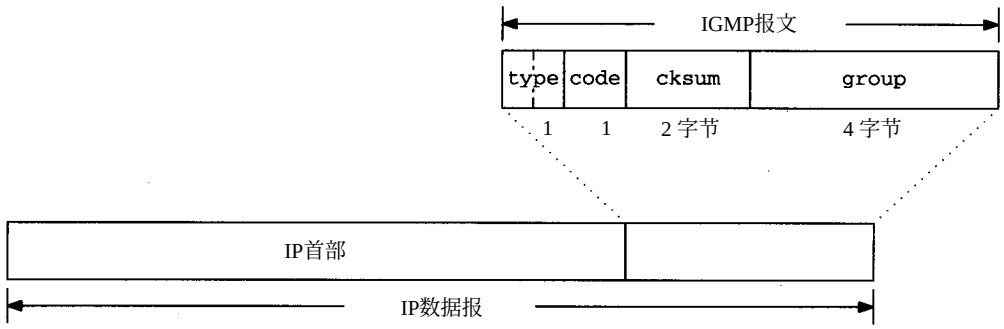


图13-8 IGMP 报文(省略igmp_)

43-44 Net/3只使用版本1的报文。多播路由器发送1类报文(IGMP_HOST_MEMBERSHIP_QUERY)向本地网络上所有主机请求成员关系报告。对1类IGMP报文的响应是主机的一个2类报文(IGMP_HOST_MEMBERSHIP_REPORT)，报告它们的多播成员信息。3类报文在路由器之间传输多播选路信息(第14章)。主机不处理3类报文。本章后面部分只讨论1类和2类报文。

45-46 在IGMP版本1中没有使用igmp_code。igmp_cksum与IP类似，计算IGMP报文的所有8个字节。

47-48 对查询，igmp_group是0。对回答，它包括报告的多播组。

图13-8是相对于IP数据报的IGMP报文结构。

13.4 IGMP的protosw的结构

图13-9是IGMP的protosw结构。

成 员	Inetsw[5]	描 述
pr_type	*SOCK_RAW*	IGMP提供原始分组服务
pr_domain	*&inetdomain*	IGMP是Internet域的一部分
pr_protocol	*IPROTO_IGMP (2)*	显示在IP首部的ip_p字段
pr_flags	*PR_ATOMIC|PR_ADDR*	插口层标志，协议处理不使用
pr_input	*igmp_input*	从IP层接收报文
pr_output	*rip_output*	向IP层发送IGMP报文
pr_ctlinput	*0*	IGMP没有使用
pr_ctloutput	*rip_ctloutput*	响应来自进程的管理请求
pr_usrreq	*rip_usrreq*	响应来自进程的通信请求
pr_init	*igmp_init*	为IGMP初始化
pr_fasttimo	*igmp_fasttino*	进程挂起成员关系报告
pr_slowtimo	*0*	IGMP没有使用
pr_drain	*0*	IGMP没有使用
pr_sysctl	*0*	IGMP没有使用

图13-9 IGMP protosw 的结构

尽管进程有可能通过IGMP protosw入口发送原始IP分组，但在本章，我们只考虑内核如何处理IGMP报文。第32章讨论进程如何用原始插口访问IGMP。

三种事件触发IGMP处理：

- 一个本地接口加入一个新的多播组(13.5节)；
- 某个IGMP定时器超时(13.6节)；和
- 收到一个IGMP查询(13.7节)。

还有两种事件也触发本地IGMP处理，但结果不发送任何报文：

- 收到一个IGMP报告(13.7节)；和
- 某个本地接口离开一个多播组(13.8节)。

下一节将讨论这五种事件。

13.5 加入一个组：igmp_joingroup函数

在第12章中我们看到，当一个新的 in_multi结构被创建时， in_addmulti调用igmp_joingroup。后面加入同一多播组的请求只增加 in_multi结构里的引用计数；不调

用igmp_joingroup。igmp_joingroup如图13-10所示。

igmp.c

```
164 void
165 igmp_joingroup(inm)
166 struct in_multi *inm;
167 {
168     int     s = splnet();
169     if (inm->inm_addr.s_addr == igmp_all_hosts_group ||
170         inm->inm_ifp == &loif)
171         inm->inm_timer = 0;
172     else {
173         igmp_sendreport(inm);
174         inm->inm_timer = IGMP_RANDOM_DELAY(inm->inm_addr);
175         igmp_timers_are_running = 1;
176     }
177     splx(s);
178 }
```

igmp.c

图13-10 igmp_joingroup 函数

164-178 inm指向组的新in_multi结构。如果新的组是“所有主机组”，或成员关系请求是环回接口的，则inm_timer被禁止，igmp_joingroup返回。不报告“所有主机组”的成员关系，因为假定每个多播主机都是该组的成员。没必要向环回接口发送组成员报告，因为本地主机是在回路网络上的唯一系统，它已经知道它的成员状态了。

在其他情况下，新组的报告被立即发送，并根据组的情况为组定时器选择一个随机的值。全局标志位 igmp_timers_are_running被设置，表明至少使能一个定时器。igmp_fasttimo (13.6节)检查这个变量，避免不必要的处理。

59-73 当新组的定时器超时，就发布第2次成员关系报告。复制报告是无害的，当第一次报告丢失或被破坏时，有了它就保险了。IGMP_RANDOM_DELAY (13-11图)计算报告时延。

igmp_var.h

```
59 /*
60  * Macro to compute a random timer value between 1 and (IGMP_MAX_REPORTING_
61  * DELAY * countdown frequency).  We generate a "random" number by adding
62  * the total number of IP packets received, our primary IP address, and the
63  * multicast address being timed-out.  The 4.3 random() routine really
64  * ought to be available in the kernel!
65  */
66 #define IGMP_RANDOM_DELAY(multiaddr) \
67     /* struct in_addr multiaddr; */ \
68     ( (ipstat.ips_total + \
69        ntohl(IA_SIN(in_ifaddr)->sin_addr.s_addr) + \
70        ntohl((multiaddr).s_addr) \
71       ) \
72       % (IGMP_MAX_HOST_REPORT_DELAY * PR_FASTHZ) + 1 \
73     )
```

igmp_var.h

图13-11 IGMP_RANDOM_DELAY 函数

根据RFC 1122，报告定时器必须设成 0到10之间的随机秒数 (IGMP_MAX_HOST_REPORT_DELAY)。因为IGMP 定时器每秒被减去5次(PR_FASTHZ)，所以IGMP_RANDOM_DELAY必须选择一个在1~50之间的随机数。如果r是把接到的所有IP分组数、主机的原始地址和多播组相加后得到的随机数，则

$$0 \le (r \bmod 50) \le 49$$

且

$$1 \le (r \bmod 50) + 1 \le 50$$

要避免为0，因为这会禁止定时器，并且不发送任何报告。

13.6 igmp_fasttimo函数

在讨论igmp_fasttino之前，我们需要描述一下遍历in_multi结构的机制。

为找到各个in_multi结构，Net/3必须遍历每个接口的in_multi表。在遍历过程中，in_multistep结构(如图13-12所示)记录位置。

in_var.h

```
123 struct in_multistep {
124     struct in_ifaddr *i_ia;
125     struct in_multi *i_inm;
126 };
```

in_var.h

图13-12 in_multistep 函数

123-126 i_ia指向下一个in_ifaddr接口结构，i_inm指向当前接口的in_multi结构。

IN_FIRST_MULTI和IN_NEXT_MULTI宏(显示如图13-13)遍历该表。

in_var.h

```
147 /*
148  * Macro to step through all of the in_multi records, one at a time.
149  * The current position is remembered in "step", which the caller must
150  * provide.  IN_FIRST_MULTI(), below, must be called to initialize "step"
151  * and get the first record.  Both macros return a NULL "inm" when there
152  * are no remaining records.
153  */
154 #define IN_NEXT_MULTI(step, inm) \
155     /* struct in_multistep  step; */ \
156     /* struct in_multi *inm; */ \
157 { \
158     if (((inm) = (step).i_inm) != NULL) \
159         (step).i_inm = (inm)->inm_next; \
160     else \
161         while ((step).i_ia != NULL) { \
162             (inm) = (step).i_ia->ia_multiaddrs; \
163             (step).i_ia = (step).i_ia->ia_next; \
164             if ((inm) != NULL) { \
165                 (step).i_inm = (inm)->inm_next; \
166                 break; \
167             } \
168         } \
169 }

170 #define IN_FIRST_MULTI(step, inm) \
171     /* struct in_multistep step; */ \
172     /* struct in_multi *inm; */ \
173 { \
174     (step).i_ia = in_ifaddr; \
175     (step).i_inm = NULL; \
176     IN_NEXT_MULTI((step), (inm)); \
177 }
```

in_var.h

图13-13 IN_FIRST_MULTI 和IN_NEXT_MULTI 结构

154-169 如果in_multi表有多个入口，i_inm就前进到下一个入口。当IN_NEXT_MULTI到达多播表的最后时，i_ia就指向下一个接口，i_inm指向与该接口相关的第一个in_multi结构。如果该接口没有多播结构，while循环继续遍历整个接口表，直到搜索完所有接口。

170-177 in_multistep数组初始化时，指向in_ifaddr表的第一个in_ifaddr结构，i_inm设成空。IN_NEXT_MULTI找到第一个in_multi结构。

从图13-9我们知道，igmp_fasttimo是IGMP的快速超时函数，每秒被调用5次。igmp_fasttimo(如图13-14)递减多播报告定时器，并在定时器超时时发送一个报告。

igmp.c

```
187 void
188 igmp_fasttimo()
189 {
190     struct in_multi *inm;
191     int     s;
192     struct in_multistep step;
193     /*
194      * Quick check to see if any work needs to be done, in order
195      * to minimize the overhead of fasttimo processing.
196      */
197     if (!igmp_timers_are_running)
198         return;
199     s = splnet();
200     igmp_timers_are_running = 0;
201     IN_FIRST_MULTI(step, inm);
202     while (inm != NULL) {
203         if (inm->inm_timer == 0) {
204             /* do nothing */
205         } else if (--inm->inm_timer == 0) {
206             igmp_sendreport(inm);
207         } else {
208             igmp_timers_are_running = 1;
209         }
210         IN_NEXT_MULTI(step, inm);
211     }
212     splx(s);
213 }
```

igmp.c

图13-14 igmp_fasttino 结构

187-198 如果igmp_timers_are_running为假，igmp_fasttimo立即返回，不再浪费时间检查各个定时器。

199-213 igmp_fasttimo重新设置运行标志位，用IN_FIRST_MULTI初始化step和inm。igmp_fasttimo函数用while循环找到各个in_multi结构和IN_NEXT_MULTI宏。对每个结构：

- 如果定时器是0，什么都不做。
- 如果定时器不是0，则将其递减。如果到达0，则发送一个IGMP组成员关系报告。
- 如果定时器还不是0，则至少还有一个定时器在运行，所以把igmp_timers_are_running设成1。

igmp_sendreport函数

igmp_sendreport函数(图13-15)为一个多播组构造和发送IGMP报告报文。

igmp.c

```
214 static void
215 igmp_sendreport(inm)
216 struct in_multi *inm;
217 {
218     struct mbuf *m;
219     struct igmp *igmp;
220     struct ip *ip;
221     struct ip_moptions *imo;
222     struct ip_moptions simo;

223     MGETHDR(m, M_DONTWAIT, MT_HEADER);
224     if (m == NULL)
225         return;
226     /*
227      * Assume max_linkhdr + sizeof(struct ip) + IGMP_MINLEN
228      * is smaller than mbuf size returned by MGETHDR.
229      */
230     m->m_data += max_linkhdr;
231     m->m_len = sizeof(struct ip) + IGMP_MINLEN;
232     m->m_pkthdr.len = sizeof(struct ip) + IGMP_MINLEN;

233     ip = mtod(m, struct ip *);
234     ip->ip_tos = 0;
235     ip->ip_len = sizeof(struct ip) + IGMP_MINLEN;
236     ip->ip_off = 0;
237     ip->ip_p = IPPROTO_IGMP;
238     ip->ip_src.s_addr = INADDR_ANY;
239     ip->ip_dst = inm->inm_addr;

240     igmp = (struct igmp *) (ip + 1);
241     igmp->igmp_type = IGMP_HOST_MEMBERSHIP_REPORT;
242     igmp->igmp_code = 0;
243     igmp->igmp_group = inm->inm_addr;
244     igmp->igmp_cksum = 0;
245     igmp->igmp_cksum = in_cksum(m, IGMP_MINLEN);

246     imo = &simo;
247     bzero((caddr_t) imo, sizeof(*imo));
248     imo->imo_multicast_ifp = inm->inm_ifp;
249     imo->imo_multicast_ttl = 1;

250     /*
251      * Request loopback of the report if we are acting as a multicast
252      * router, so that the process-level routing demon can hear it.
253      */
254     {
255         extern struct socket *ip_mrouter;
256         imo->imo_multicast_loop = (ip_mrouter != NULL);
257     }
258     ip_output(m, NULL, NULL, 0, imo);

259     ++igmpstat.igps_snd_reports;
260 }
```

igmp.c

图13-15 igmp_sentreport 函数

214-232 唯一的参数inm指向被报告组的in_multi结构。igmp_sendreport分配一个新的mbuf，准备存放一个IGMP报文。igmp_sendreport为链路层首部留下空间，把mbuf

的长度和分组的长度设成IGMP报文的长度。

233-245 每次构造IP首部和IGMP报文的一个字段。数据报的源地址设成INADDR_ANY，目的地址是被报告的多播组。ip_output用输出接口的单播地址替换INADDR_ANY。每个组成员和所有多播路由器都接收报告(因为路由器接收所有IP多播)。

246-260 最后，igmp_sentreport构造一个ip_moptions结构，并把它与报文一起传给ip_output。与in_multi结构相关的接口被选做输出的接口；TTL被设成1，使报告只在本地网络上；如果本地系统被配置成路由器，则允许这个请求的多播环回。

进程级的多播路由器必须监听成员关系报告。在12.14节中我们看到，当系统被配置成多播路由器时，总是接收IGMP数据报。通过普通的运输层分用程序把报文传给IGMP的igmp_input和pr_input函数(图13-9)。

13.7 输入处理：igmp_input函数

在12.14节中，我们描述了ipintr的多播处理部分。我们看到，多播路由器接受所有IGMP报文，但多播主机只接受那些到达接口是目的多播组成员的IGMP报文(也即，那些接收它们的接口是组成员的查询和成员关系报告)。

标准协议分用机制把接受的报文传给igmp_input。igmp_input的开始和结束如图13-16所示。下面几节描述每种IGMP报文类型码。

igmp.c

```
52 void
53 igmp_input(m, iphlen)
54 struct mbuf *m;
55 int     iphlen;
56 {
57     struct igmp *igmp;
58     struct ip *ip;
59     int     igmplen;
60     struct ifnet *ifp = m->m_pkthdr.rcvif;
61     int     minlen;
62     struct in_multi *inm;
63     struct in_ifaddr *ia;
64     struct in_multistep step;

65     ++igmpstat.igps_rcv_total;

66     ip = mtod(m, struct ip *);
67     igmplen = ip->ip_len;

68     /*
69      * Validate lengths
70      */
71     if (igmplen < IGMP_MINLEN) {
72         ++igmpstat.igps_rcv_tooshort;
73         m_freem(m);
74         return;
75     }
76     minlen = iphlen + IGMP_MINLEN;
77     if ((m->m_flags & M_EXT || m->m_len < minlen) &&
78         (m = m_pullup(m, minlen)) == 0) {
79         ++igmpstat.igps_rcv_tooshort;
80         return;
81     }
```

图13-16 igmp_input 函数

```
82      /*
83       * Validate checksum
84       */
85      m->m_data += iphlen;
86      m->m_len -= iphlen;
87      igmp = mtod(m, struct igmp *);
88      if (in_cksum(m, igmplen)) {
89          ++igmpstat.igps_rcv_badsum;
90          m_freem(m);
91          return;
92      }
93      m->m_data -= iphlen;
94      m->m_len += iphlen;
95      ip = mtod(m, struct ip *);

96      switch (igmp->igmp_type) {

                [redacted]

157     }
158     /*
159      * Pass all valid IGMP packets up to any process(es) listening
160      * on a raw IGMP socket.
161      */
162     rip_input(m);
163 }
```

igmp.c

图13-16 (续)

1. 验证IGMP报文

52-96 函数ipintr传递一个指向接受分组(存放在一个mbuf中)的指针m，和数据报IP首部的大小iphlen。

数据报的长度必须足够容纳一个 IGMP报文(IGMP_MIN_LEN)，并能被放在一个标准的mbuf首部中(m_pullup)，而且还必须有正确的IGMP检验和。如果发现有任何错误，统计错误的个数，并自动丢弃该数据报，igmp_input返回。

igmp_input进程体根据igmp_type内的代码处理无效报文。记得在图 13-6中，igmp_type包含一个版本码和一个类型码。switch语句基于igmp_type(图13-7)中两个值的结合。下面几节分别讨论几种情况。

2. 把IGMP报文传给原始IP

157-163 这个switch语句没有default情况。所有有效报文(也就是，格式正确的报文)被传给rip_input，在rip_input里被提交给所有监听IGMP报文的进程。监听进程可以自由处理或丢弃那些具有内核不识别的版本或类型的 IGMP报文。

mrouted依靠对rip_input的调用接收成员关系查询和报告。

13.7.1 成员关系查询：IGMP_HOST_MEMBERSHIP_QUERY

RFC 1075推荐多播路由器每 120秒至少发布一次 IGMP成员关系查询。把查询发到224.0.0.1组(“所有主机组”)。图13-17显示了主机如何处理报文。

97-122 到达环回接口上的查询报文被自动丢弃 (习题13.1)。查询报文被定义成发给“所有

主机组”，到达其他地址的查询报文由igps_rcv_badqueries统计数量，并被丢弃。

igmp.c

```
 97     case IGMP_HOST_MEMBERSHIP_QUERY:
 98         ++igmpstat.igps_rcv_queries;

 99         if (ifp == &loif)
100             break;

101         if (ip->ip_dst.s_addr != igmp_all_hosts_group) {
102             ++igmpstat.igps_rcv_badqueries;
103             m_freem(m);
104             return;
105         }
106         /*
107          * Start the timers in all of our membership records for
108          * the interface on which the query arrived, except those
109          * that are already running and those that belong to the
110          * "all-hosts" group.
111          */
112         IN_FIRST_MULTI(step, inm);
113         while (inm != NULL) {
114             if (inm->inm_ifp == ifp && inm->inm_timer == 0 &&
115                 inm->inm_addr.s_addr != igmp_all_hosts_group) {
116                 inm->inm_timer =
117                     IGMP_RANDOM_DELAY(inm->inm_addr);
118                 igmp_timers_are_running = 1;
119             }
120             IN_NEXT_MULTI(step, inm);
121         }

122         break;
```

igmp.c

图13-17 IGMP查询报文的输入处理

接受查询报文并不会立即引起IGMP成员报告。相反，igmp_input为与接收查询的接口相关的各个组定时器设置一个随机的值 IGMP_RANDOM_DELAY。当某组的定时器超时，则igmp_fasttimo发送一个成员关系报告，与此同时，其他所有收到查询的主机也进行同一动作。一旦某个主机上的某个特定组的随机定时器超时，就向该组多播一个报告。这个报告将取消其他主机上的定时器，保证只有一个报告在网络上多播。路由器与其他组成员一样，接收该报告。

这个情况的一个例外就是“所有主机组”。这个组不设定时器，也不发送报告。

13.7.2 成员关系报告：IGMP_HOST_MEMBERSHIP_REPORT

接收一个IGMP成员关系报告是我们在13.1节中提到的不会产生IGMP报文的两种事件之一。该报文的效果限于接收它的接口本地。图13-18显示了报文处理。

123-146 和发送到不正确多播组的成员关系报告一样，发到环回接口上的报告被丢弃。也就是说，报文必须寻址到报文内标识的组。

不完整地初始化的主机的源地址中可能没有网络号或主机号 (或两者都没有)。igmp_report查看地址的A类网络部分，如果地址的网络或子网部分是0，这部分一定为0。如果是这种情况，则把源地址设成子网地址，其中包含正在接收接口的网络标识符和子网标识符。这样做的唯一原因是为了通知子网号所标识的正在接收接口上的某个进程级守护程序。

如果接收接口属于被报告的组，就把相关的报告定时器重新设成0。从而使发给该组的第一个报告能够制止其他主机发布报告。路由器只需知道网络上至少有一个接口是组的成员，

就无需维护一个明确的组成员表或计数器。

igmp.c

```
123     case IGMP_HOST_MEMBERSHIP_REPORT:
124         ++igmpstat.igps_rcv_reports;

125         if (ifp == &loif)
126             break;

127         if (!IN_MULTICAST(ntohl(igmp->igmp_group.s_addr)) ||
128             igmp->igmp_group.s_addr != ip->ip_dst.s_addr) {
129             ++igmpstat.igps_rcv_badreports;
130             m_freem(m);
131             return;
132         }
133         /*
134          * KLUDGE: if the IP source address of the report has an
135          * unspecified (i.e., zero) subnet number, as is allowed for
136          * a booting host, replace it with the correct subnet number
137          * so that a process-level multicast routing demon can
138          * determine which subnet it arrived from.  This is necessary
139          * to compensate for the lack of any way for a process to
140          * determine the arrival interface of an incoming packet.
141          */
142         if ((ntohl(ip->ip_src.s_addr) & IN_CLASSA_NET) == 0) {
143             IFP_TO_IA(ifp, ia);
144             if (ia)
145                 ip->ip_src.s_addr = htonl(ia->ia_subnet);
146         }
147         /*
148          * If we belong to the group being reported, stop
149          * our timer for that group.
150          */
151         IN_LOOKUP_MULTI(igmp->igmp_group, ifp, inm);
152         if (inm != NULL) {
153             inm->inm_timer = 0;
154             ++igmpstat.igps_rcv_ourreports;
155         }
156         break;
```

igmp.c

图13-18 IGMP报告报文的输入处理

13.8 离开一个组：igmp_leavegroup函数

我们在12章中看到，当in_multi结构中的引用计数器跳到0时，in_delmulti调用igmp_leavegroup。如图13-19所示。

igmp.c

```
179 void
180 igmp_leavegroup(inm)
181 struct in_multi *inm;
182 {
183     /*
184      * No action required on leaving a group.
185      */
186 }
```

igmp.c

图13-19 igmp_leavegroup 函数

179-186 当一个接口离开一个组时，IGMP没有采取任何动作。不发明确的通知——下一次多播路由器发布IGMP查询时，接口不为该组生成IGMP报告。如果没有为某个组生成报告，则多播路由器就假定所有接口已经离开该组，并停止把到该组的分组在网络上多播。

如果当一个报告被挂起时，接口离开了该组(就是说，此时组的报告定时器正在计时)，就不再发送该报告，因为当`icmp_leavegroup`返回时，`in_delmulti`(图12-36)已经把组的定时器及其相关的`in_multi`结构丢掉了。

13.9 小结

本章我们讲述了IGMP，IGMP在一个网络上的主机和路由器之间传递IP多播成员信息。当一个接口加入一个组时，或按照多播路由器发布的IGMP报告查询报文的要求，生成IGMP成员关系报告。

设计IGMP使交换成员信息所需要的报文数最少：

- 当主机加入一个组时，宣布它们的成员关系；
- 对成员关系查询的响应被推迟一个随机的时间，而且第一个响应抑制了其他的响应；
- 当主机离开一个组时，不发通知报文；
- 每分钟发的成员查询不超过一次。

多播路由器与其他路由器共享自己收集的IGMP信息(第14章)，以便于把多播数据报传给多播目的组的远程成员。

习题

13.1 为什么不需要响应在环回接口上到达的IGMP查询？

13.2 验证图13-15中226到229行的假设。

13.3 对在点到点网络接口上到达的成员关系查询，是否有必要设置随机的延迟时间？

第14章　IP多播选路

14.1　引言

前面两章讨论了在一个网络上的多播。本章我们讨论在整个互联网上的多播。我们将讨论mrouted程序的执行，该程序计算多播路由表，以及在网络之间转发多播数据报的内核函数。

从技术上说，多播分组(packet)被转发。本章我们假定每个多播分组中都包含一个完整数据报(也就是说，没有分片)，所以我们只用名词数据报(datagram)。Net/3转发IP分片，也转发IP数据报。

图14-1是mrouted的几个版本及它们和BSD版本的对应关系。mrouted版本包括用户级守护程序和内核级多播程序。

mrouted版本	描　述
1.2	修改4.3 BSD Tahoe版本
2.0	包括在4.4 BSD和Net/3中
3.3	修改SunOS 4.1.3

图14-1　mrouted 和IP多播版本

IP多播技术是一个活跃的研究和开发领域。本章讨论包括在Net/3中的多播软件的2.0版，但被认为已经过时了。3.3版的发行还有一段时间，因此无法在本书中完整地讨论，但我们在整个过程中将指出3.3版本的一些特点。

因为还没有广泛安装商用多播路由器，所以常用多播隧道连接标准IP单播互联网上的两个多播路由器，构造多播网络。Net/3支持多播隧道，并采用宽松源站记录路由(LSRR，Loose Source Record Route)选项(9.6节)构造多播隧道。一种更好的隧道技术把IP多播数据报封装在一个单播数据报里，3.3版的多播程序支持这一技术，但Net/3不支持。

与第12章一样，我们用通常名称运输层协议代指发送和接收多播数据报的协议，但UDP是唯一支持多播的Internet协议。

14.2　代码介绍

本章讨论的三个文件显示在图14-2中。

文　件	描　述
netinet/ip_mroute.h	多播结构定义
netinet/ ip_mroute.c	多播选路函数
netinet/raw_ip.c	多播选路选项

图14-2　本章讨论的文件

14.2.1　全局变量

多播选路程序所使用的全局变量显示在图14-3中。

变 量	数据类型	描 述
cached_mrt	struct mrt	多播选路的“后面一个”高速缓存
cached_origin	u_long	“后面一个” 高速缓存的多播组
cached_originmask	u_long	“后面一个” 高速缓存的多播组的掩码
mrtstat	struct mrtstat	多播选路统计
mrttable	struct mrt *[]	指向多播路由器的指针的散列表
numvifs	vifi_t	允许的多播接口数
viftable	struct vif[]	虚拟多播接口的数组

图14-3 本章介绍的全局变量

14.2.2 统计量

多播选路程序收集的所有统计信息都放在图 14-4的mrtstat结构中。图14-5是在执行netstat -gs命令后，输出的统计信息。

mrtstat成员	描 述	SNMP使用的
mrts_mrt_lookups	查找的多播路由数	
mrts_mrt_misses	高速缓存丢失的多播路由数	
mrts_grp_lookups	查找的组地址数	
mrts_grp_misses	高速缓存丢失的组地址数	
mrts_no_route	查找失败的多播路由数	
mrts_bad_tunnel	有错误的隧道选项的分组数	
mrts_cant_tunnel	没有空间存放隧道选项的分组数	

图14-4 本章收集的统计量

netstat -gs 输出	mrtstat 成员
multicast routing:	
329569328 multicast route lookups	mrts_mrt_lookups
9377023 multicast route cache misses	mrts_mrt_misses
242754062 group address lookups	mrts_grp_lookups
159317788 group address cache misses	mrts_grp_misses
65648 datagrams with no route for origin	mrts_no_route
0 datagrams with malformed tunnel options	mrts_bad_tunnel
0 datagrams with no room for tunnel options	mrts_cant_tunnel

图14-5 IP多播路由选择统计的例子

这些统计信息来自一个有两个物理接口和一个隧道接口的系统。它们说明， 98%的时间，在高速缓存中发现多播路由。组地址高速缓存的效率稍低一些，最高只有 34%。图14-34描述了路由缓存，图14-21描述了组地址高速缓存。

14.2.3 SNMP变量

多播选路没有标准的SNMP MIB，但 [McCloghrie和Farinacci 1994a] 和 [McCloghrie和Farinacci 1994b] 描述一些多播路由器的实验MIB。

14.3 多播输出处理(续)

12.15节讲到如何为输出的多播数据报选择接口。我们看到在 ip_moptions结构中

ip_output被传给一个明确的接口，或者ip_output在路由表中查找目的组，并使用在路由入口中返回的接口。

如果在选择了输出的接口后，ip_output回送该数据报，就把它放在所选输出接口等待输入处理，当ipintr处理它时，把它当作是要转发的数据报。图14-6显示了这个过程。

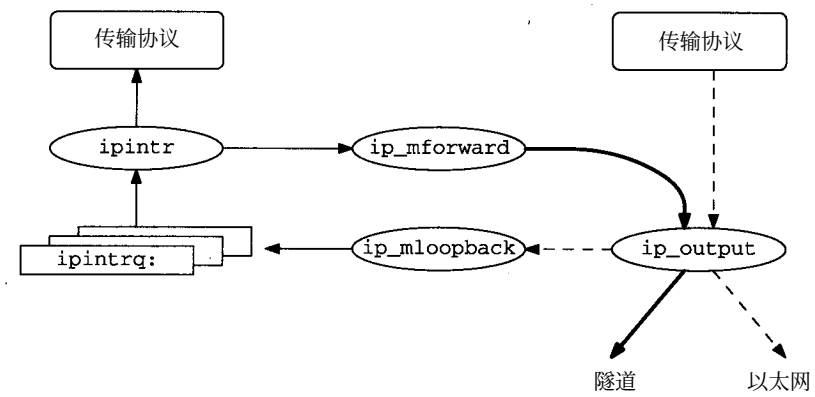


图14-6 有环回的多播输出处理

在图14-6中，虚线箭头代表原始输出的数据报，本例是本地以太网上的多播。ip_mloopback创建的备份由带箭头的细线表示；并作为输入被传给运输层协议。当ip_mforward决定通过系统上的另一个接口转发该数据报时，就产生第三个备份。图14-6中最粗的箭头代表第三个备份，在多播隧道上发送。

如果数据报不是回送的，则ip_output把它直接传给ip_mforward，ip_mforward复制并处理该数据报，就像它是从ip_output选定的接口上收到的一样。图14-7显示了这个过程。

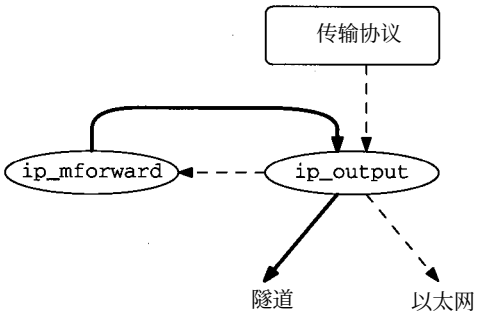


图14-7 没有环回的多播输出处理

一旦ip_mforward调用ip_output发送多播数据报，它就把IP_FORWARDING置位，这样，ip_output就不再把数据报传回给ip_mforward，以免导致无限循环。

图12-42显示了ip_mloopback。14.8节描述了ip_mforward。

14.4 mrouted守护程序

用户级进程mrouted守护程序允许和管理多播路由选择。mrouted实现IGMP协议的路由部分，并与其他多播路由器通信，实现网络间的多播路由选择。路由算法在mrouted上实现，但内核维护多播路由选择表，并转发数据报。

本书中我们只讨论支持mrouted的内核数据结构和函数——不讨论mrouted本身。我们讨论用于为数据报选择路由的截断逆向路径广播TRPB(Truncated Reverse Path Broadcast)算法[Deering和Cheriton 1990]，以及用于在多播路由器之间传递信息的距离向量多播选路协议DVMRP。我们力求使读者了解内核多播程序的工作原理。

RFC 1075 [Waitzman、Partidge 和Deering1988] 是DVMRP的一个老版本。mrouted实现了一个新的 DVMRP，还没有用 RFC文档写出来。目前，该算法和协议的最好的文档是mrouted发布的源代码。附录B指出在哪里能找到到源代码。

mrouted守护程序通过在一个IGMP插口上设置选项与内核通信 (第32章)。这些选项总结在图14-8中。

optname	optval类型	函 数	描 述
DVMRP_INIT		ip_mrouter_init	mrouted开始
DVMRP_DONE		ip_mrouter_done	mrouted被关闭
DVMRP_ADD_VIF	struct vifctl	add_vif	增加虚拟接口
DVMRP_DEL_VIF	vifi_t	del_vif	删除虚拟接口
DVMRP_ADD_LGRP	struct lgrplctl	add_lgrp	为某个接口增加多播组入口
DVMRP_DEL_LGRP	struct lgrplctl	del_lgrp	为某个接口删除多播组入口
DVMRP_ADD_MRT	struct mrtctl	add_mrt	增加多播路由
DVMRP_DEL_MRT	struct in addr	del_mrt	删除多播路由

图14-8 多播路由插口选项

图14-8显示的插口选项被setsockopt系统调用传给rip_ctloutput(32.8节)。图14-9显示了处理DVMRP_*xxx* 选项的rip_ctloutput部分。

raw_ip.c

```
173     case DVMRP_INIT:
174     case DVMRP_DONE:
175     case DVMRP_ADD_VIF:
176     case DVMRP_DEL_VIF:
177     case DVMRP_ADD_LGRP:
178     case DVMRP_DEL_LGRP:
179     case DVMRP_ADD_MRT:
180     case DVMRP_DEL_MRT:
181         if (op == PRCO_SETOPT) {
182             error = ip_mrouter_cmd(optname, so, *m);
183             if (*m)
184                 (void) m_free(*m);
185         } else
186             error = EINVAL;
187         return (error);
```

raw_ip.c

图14-9 rip_ctloutput 函数：DVMRP_ *xxx* 插口选项

173-187 当调用setsockopt时，op等于PRCO_SETOPT，而且所有选项都被传给ip_mrouter_cmd函数。对于getsockopt系统调用，op等于PRCO_GETOPT；对所有选项都返回EINVAL。

图14-10显示了ip_mrouter_cmd函数。

ip_mroute.c

```
84 int
85 ip_mrouter_cmd(cmd, so, m)
86 int     cmd;
87 struct socket *so;
88 struct mbuf *m;
89 {
90     int     error = 0;
```

图14-10 ip_mrouter_cmd 函数

```
 91      if (cmd != DVMRP_INIT && so != ip_mrouter)
 92          error = EACCES;
 93      else
 94          switch (cmd) {
 95          case DVMRP_INIT:
 96              error = ip_mrouter_init(so);
 97              break;
 98          case DVMRP_DONE:
 99              error = ip_mrouter_done();
100              break;
101          case DVMRP_ADD_VIF:
102              if (m == NULL || m->m_len < sizeof(struct vifctl))
103                      error = EINVAL;
104              else
105                  error = add_vif(mtod(m, struct vifctl *));
106              break;
107          case DVMRP_DEL_VIF:
108              if (m == NULL || m->m_len < sizeof(short))
109                      error = EINVAL;
110              else
111                  error = del_vif(mtod(m, vifi_t *));
112              break;
113          case DVMRP_ADD_LGRP:
114              if (m == NULL || m->m_len < sizeof(struct lgrplctl))
115                      error = EINVAL;
116              else
117                  error = add_lgrp(mtod(m, struct lgrplctl *));
118              break;
119          case DVMRP_DEL_LGRP:
120              if (m == NULL || m->m_len < sizeof(struct lgrplctl))
121                      error = EINVAL;
122              else
123                  error = del_lgrp(mtod(m, struct lgrplctl *));
124              break;
125          case DVMRP_ADD_MRT:
126              if (m == NULL || m->m_len < sizeof(struct mrtctl))
127                      error = EINVAL;
128              else
129                  error = add_mrt(mtod(m, struct mrtctl *));
130              break;
131          case DVMRP_DEL_MRT:
132              if (m == NULL || m->m_len < sizeof(struct in_addr))
133                      error = EINVAL;
134              else
135                  error = del_mrt(mtod(m, struct in_addr *));
136              break;
137          default:
138              error = EOPNOTSUPP;
139              break;
140          }
141      return (error);
142  }
```

ip_mroute.c

图14-10 (续)

这些“选项”更像命令，因为它们引起内核更新多个数据结构。本章后面我们将使用命令(command)一词强调这个事实。

84-92 mrouted发布的第一个命令必须是 DVMRP_INIT。后续命令必须来自发布DVMRP_INIT的同一插口。当在其他插口上发布其他命令时，返回 EACCES。

94-142 switch语句的每个case语句检查每条命令中的数据量是否正确，然后调用匹配函数。如果不能识别该命令，则返回 EOPNOTSUPP。任何从匹配函数返回的错误都在 error中发布，并在函数的最后返回。

初始化时，mrouted发布DVMRP_INIT命令，调用图14-11显示的ip_mrouter_init。

ip_mroute.c

```
146 static int
147 ip_mrouter_init(so)
148 struct socket *so;
149 {
150     if (so->so_type != SOCK_RAW ||
151         so->so_proto->pr_protocol != IPPROTO_IGMP)
152         return (EOPNOTSUPP);
153     if (ip_mrouter != NULL)
154         return (EADDRINUSE);
155     ip_mrouter = so;
156     return (0);
157 }
```

ip_mroute.c

图14-11 ip_mrouter_init 函数：DVMRP_INIT 命令

146-157 如果不是在某个原始IGMP插口上发布命令，或者如果DVMRP_INIT已经被置位，则分别返回EOPNOTSUPP和EADDRINUSE。全局变量ip_mrouter保存指向某个插口的指针，初始化命令就是在该插口上发布的。必须在该插口上发布后续命令。以避免多个 mrouted进程的并行操作。

下面几节讨论其他DVMRP_*xxx*命令。

14.5 虚拟接口

当作为多播路由器运行时， Net/3接收到达的多播数据报，复制它们，并在一个或多个接口上转发备份。通过这种方式，数据报被转发给互联网上的其他多播路由器。

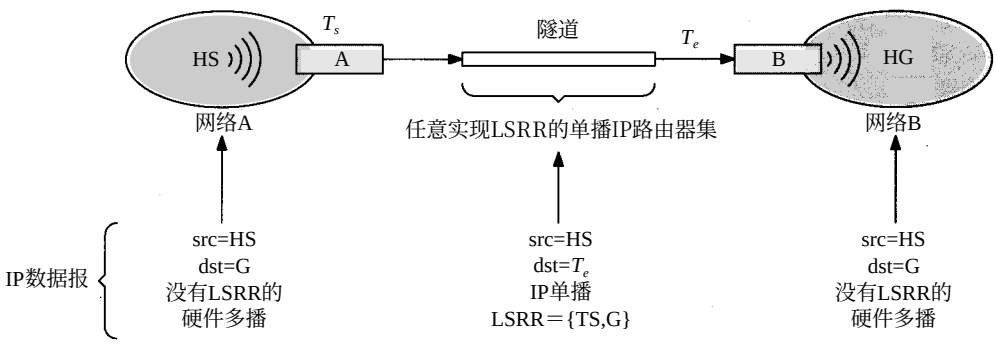


图14-12 多播隧道

输出的接口可以是一个物理接口，也可以是一个多播隧道。多播隧道的两端都与一个多播路由器上的某个物理接口相关。多播隧道使两个多播路由器，即使被不能转发多播数据报的路由器分隔，也能够交换多播数据报。图14-12是一个多播隧道连接的两个多播路由器。

图14-12中，网络A上的源主机HS正在向组G多播数据报。组G的唯一成员在网络B上，并通过一个多播隧道连接到网络A。路由器A接收多播(因为多播路由器接收所有多播)，查询它的多播路由选择表，并通过多播隧道转发该数据报。

隧道的开始是路由器A上的一个物理接口，以IP单播地址T_s标识。隧道的结束是网络B上的一个物理接口，以IP单播地址T_e标识。隧道本身是一个任意复杂的网络，由实现LSRR选项的IP单播路由器连接起来。图14-13显示IP LSRR选项如何实现多播隧道。

系统	IP首部		源路由选项		描　述
	ip_src	ip_dst	偏移	地　址	
HS	HS	G			在网络A上
T_s	HS	T_e	8	T_s • G	在隧道上
T_e	HS	G	12	T_s　见正文	在路由器B上ip_dooptions之后
T_e	HS	G			在路由器B上ip_mforward之后

图14-13　LSRR多播隧道选项

图14-13的第一行是HS在网络A上发送的多播数据报。路由器A全部接收，因为多播路由器接收本地连接的网络上的所有数据报。

为通过隧道发送数据报，路由器A在IP首部插入一个LSRR选项。第二行是在隧道上离开A时的数据报。LSRR选项的第一个地址是隧道的源地址，第二个地址是目的多播组地址。数据报的目的地址是T_e——隧道的另一端。LSRR偏移指向目的组。

经过隧道的数据报被转发，通过互联网，直到它到达路由器B上的隧道的另一端。

该图中的第三行是被路由器B上的ip_dooptions处理之后的数据报。记得第9章中讲到，ip_dooptions在ipintr检查数据报的目的地址之前处理LSRR选项。因为数据报的目的地址(T_e)和路由器B上的一个接口匹配，所以ip_dooptions把由选项偏移(本例中是G)标识的地址复制到IP首部的目的地址字段。在选项内，G被ip_rtaddr返回的地址取代，ip_rtaddr通常根据IP目的地址(本例中是G)为数据报选择输出的接口。这个地址是不相关的，因为ip_mforward将丢弃整个选项。最后，ip_dooptions把选项偏移向前移动。

图14-13的第四行是ipintr调用ip_mforward之后的数据报。在那里，LSRR选项被识别，并从数据报首部中移走。得到的数据报看起来就象原始多播数据报，由ip_mforward处理它，把它作为多播数据报在网络B上转发，并被HG收到。

用LSRR构造的多播隧道已经过时了。因为1993年3月发布了mrouted程序，该程序通过在IP多播数据报的首部前面加上另一个IP首部来构造隧道。新IP首部的协议设置为4，表明分组的内容是另一个IP分组。有关这个值的文档在RFC 1700——“IP中的IP”协议中。新版本的mrouted程序为了向后兼容，也支持LSRR隧道。

14.5.1 虚拟接口表

无论物理接口还是隧道接口，内核都为其在虚拟接口(virtual interface)表中维护一个入口，其中包含了只有多播使用的信息。每个虚拟接口都用一个vif结构表示(图14-14)。全局变量

viftable是一个这种结构的数组。数组的下标保存在无符号短整数 vifi_t变量中。

ip_mroute.h

```
105 struct vif {
106     u_char  v_flags;            /* VIFF_ flags */
107     u_char  v_threshold;        /* min ttl required to forward on vif */
108     struct in_addr v_lcl_addr;  /* local interface address */
109     struct in_addr v_rmt_addr;  /* remote address (tunnels only) */
110     struct ifnet *v_ifp;        /* pointer to interface */
111     struct in_addr *v_lcl_grps; /* list of local grps (phyints only) */
112     int     v_lcl_grps_max;     /* malloc'ed number of v_lcl_grps */
113     int     v_lcl_grps_n;       /* used number of v_lcl_grps */
114     u_long  v_cached_group;     /* last grp looked-up (phyints only) */
115     int     v_cached_result;    /* last look-up result (phyints only) */
116 };
```

ip_mroute.h

图14-14 vif 结构

105-110 为v_flags定义的唯一的标志位是VIFF_TUNNEL。被置位时，该接口是一个到远程多播路由器的隧道。没有置位时，接口是在本地系统上的一个物理接口。 v_threshold是我们在12.9节描述的多播阈值。v_lcl_addr是与这个虚拟接口相关的本地接口的 IP地址。v_rmt_addr是一个IP多播隧道远端的单播IP地址。v_lcl_addr或者v_rmt_addr为非零，但不会两者都为非零。对物理接口， v_ifp非空，并指向本地接口的 ifnet结构。对隧道，v_ifp是空的。

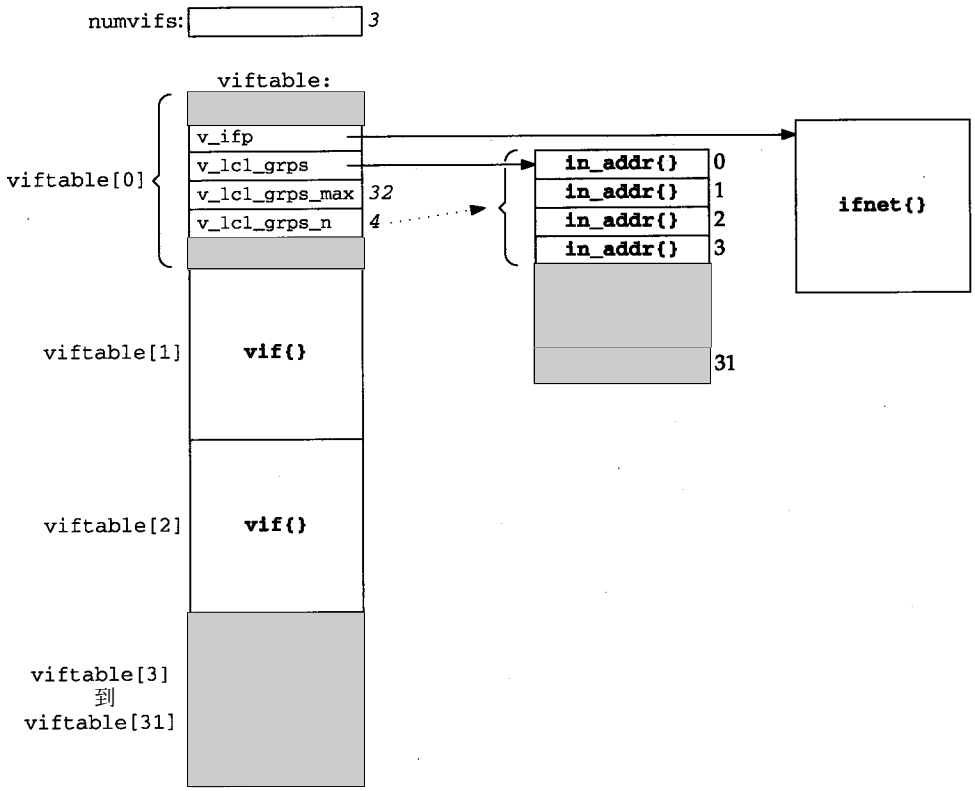


图14-15 viftable 数组

111-116 v_lcl_grps指向一个IP 多播组地址数组，这个数组记录了在连到的接口上的成员组列表。对隧道来说，v_lcl_grps总是空的。数组的大小保存在v_lcl_grps_max中，被使用的入口数保存在 v_lcl_grps_n中。数组随着组成员关系表的增大而增长。v_cached_group和v_cached_result实现“一个入口”高速缓存，其中记录的是最近一次查找得到的组。

图14-15说明了viftable，它最多有32个(MAXVIFS)入口。viftable[2]是正在使用的最后一个入口，所以numvifs是3。编译内核时固定了表的大小。图中还显示了表的第一个入口的vif结构的几个成员。v_ifp指向一个ifnet结构，v_lcl_grps指向in_addr结构中的一个数组。数组有32(v_lcl_grps_max)个入口，其中只用了4个(v_lcl_grps_n)。

mrouted通过DVMRP_ADD_VIF和DVMRP_DEL_VIF命令维护viftable。通常，当mrouted开始运行时，会把本地系统上有多播能力的接口加入表中。当 mrouted阅读自己的配置文件，通常是/etc/mrouted.conf时，会把多播隧道加入表中。这个文件中的命令也可能从虚拟接口表中删除物理接口，或者改变与接口有关的多播信息。

mrouted用DVMRP_ADD_VIF命令把ctl结构(图14-16)传给内核。它指示内核在虚拟接口表中加入一个接口项。

ip_mroute.h

```
76 struct vifctl {
77     vifi_t  vifc_vifi;          /* the index of the vif to be added */
78     u_char  vifc_flags;         /* VIFF_ flags (Figure 14.14) */
79     u_char  vifc_threshold;     /* min ttl required to forward on vif */
80     struct in_addr vifc_lcl_addr;   /* local interface address */
81     struct in_addr vifc_rmt_addr;   /* remote address (tunnels only) */
82 };
```

ip_mroute.h

图14-16 vifctl 结构

78-82 vifc_vifi识别viftable中虚拟接口的下标。其他 4个成员，vifc_flags、vifc_threshold、vifc_lcl_addr和vifc_rmt_addr，被add_vif函数复制到vif函数中。

14.5.2 add_vif函数

图14-17是add_vif函数。

ip_mroute.c

```
202 static int
203 add_vif(vifcp)
204 struct vifctl *vifcp;
205 {
206     struct vif *vifp = viftable + vifcp->vifc_vifi;
207     struct ifaddr *ifa;
208     struct ifnet *ifp;
209     struct ifreq ifr;
210     int     error, s;
211     static struct sockaddr_in sin =
212     {sizeof(sin), AF_INET};

213     if (vifcp->vifc_vifi >= MAXVIFS)
214         return (EINVAL);
215     if (vifp->v_lcl_addr.s_addr != 0)
216         return (EADDRINUSE);
```

图14-17 add_vif 函数：DVMRP_ADD_VIF 命令

```
217     /* Find the interface with an address in AF_INET family */
218     sin.sin_addr = vifcp->vifc_lcl_addr;
219     ifa = ifa_ifwithaddr((struct sockaddr *) &sin);
220     if (ifa == 0)
221         return (EADDRNOTAVAIL);
222     s = splnet();
223     if (vifcp->vifc_flags & VIFF_TUNNEL)
224         vifp->v_rmt_addr = vifcp->vifc_rmt_addr;
225     else {
226         /* Make sure the interface supports multicast */
227         ifp = ifa->ifa_ifp;
228         if ((ifp->if_flags & IFF_MULTICAST) == 0) {
229             splx(s);
230             return (EOPNOTSUPP);
231         }
232         /*
233          * Enable promiscuous reception of all IP multicasts
234          * from the interface.
235          */
236         satosin(&ifr.ifr_addr)->sin_family = AF_INET;
237         satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
238         error = (*ifp->if_ioctl) (ifp, SIOCADDMULTI, (caddr_t) & ifr);
239         if (error) {
240             splx(s);
241             return (error);
242         }
243     }
244     vifp->v_flags = vifcp->vifc_flags;
245     vifp->v_threshold = vifcp->vifc_threshold;
246     vifp->v_lcl_addr = vifcp->vifc_lcl_addr;
247     vifp->v_ifp = ifa->ifa_ifp;
248     /* Adjust numvifs up if the vifi is higher than numvifs */
249     if (numvifs <= vifcp->vifc_vifi)
250         numvifs = vifcp->vifc_vifi + 1;
251     splx(s);
252     return (0);
253 }
```

ip_mroute.c

图14-17 (续)

1. 验证下标

202-216 如果mrouted指定的vifc_vifi中的下标太大，或者该表入口已经被使用，则分别返回EINVAL或EADDRINUSE。

2. 本地物理接口

217-221 ifa_ifwithaddr取得vifc_lcl_addr中的单播IP地址，并返回一个指向相关ifnet结构的指针。这就标识出这个虚拟接口要用的物理接口。如果没有匹配的接口，返回EADDRNOTAVAIL。

3. 配置隧道接口

222-224 对于隧道，它的远端地址被从vifctl结构中复制到接口表的vif结构中。

4. 配置物理接口

225-243 对于物理接口，链路级驱动程序必须支持多播。 SIOCADDMULTI命令与

INADDR_ANY一起配置接口，开始接收所有IP多播数据报(图12-32)，因为它是一个多播路由器。当ipintr把到达数据报传给ip_mforward时，被ip_mforward转发。

5. 保存多播信息

244-253 其他接口信息被从vifctl结构复制到vif结构。如果需要，更新numvifs，记录正在使用的虚拟接口数。

14.5.3 del_vif函数

图14-18显示的del_vif函数从虚拟接口表中删除表项。当 mrouted设置DVMRP_DEL_VIF命令时，调用该函数。

1. 验证下标

257-268 如果传给del_vif的下标大于正在使用的最大下标，或者指向一个没有使用的入口，则分别返回EINVAL和EADDRNOTAVAIL。

ip_mroute.c

```
257 static int
258 del_vif(vifip)
259 vifi_t *vifip;
260 {
261     struct vif *vifp = viftable + *vifip;
262     struct ifnet *ifp;
263     int     i, s;
264     struct ifreq ifr;

265     if (*vifip >= numvifs)
266         return (EINVAL);
267     if (vifp->v_lcl_addr.s_addr == 0)
268         return (EADDRNOTAVAIL);

269     s = splnet();

270     if (!(vifp->v_flags & VIFF_TUNNEL)) {
271         if (vifp->v_lcl_grps)
272             free(vifp->v_lcl_grps, M_MRTABLE);
273         satosin(&ifr.ifr_addr)->sin_family = AF_INET;
274         satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
275         ifp = vifp->v_ifp;
276         (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);
277     }
278     bzero((caddr_t) vifp, sizeof(*vifp));

279     /* Adjust numvifs down */
280     for (i = numvifs - 1; i >= 0; i--)
281         if (viftable[i].v_lcl_addr.s_addr != 0)
282             break;
283     numvifs = i + 1;

284     splx(s);
285     return (0);
286 }
```

ip_mroute.c

图14-18 del_vif 函数：DVMRP_DEL_VIF 命令

2. 删除接口

269-278 对于物理接口，释放本地多播组表， SIOCADDMULTI禁止接收所有多播数据报，bzero对viftable的入口清零。

3. 调整接口计数

279-286 for循环从以前活动的最大入口开始向后直到第一个入口为止，搜索出第一个活动的入口。对没有使用的入口，v_lcl_addr(一个in_addr结构)的成员s_addr是0。相应地更新numvifs，函数返回。

14.6 IGMP(续)

第13章侧重于IGMP协议的主机部分，mrouted实现了这个协议的路由器部分。mrouted必须为每个物理接口记录哪个多播组有成员在连到的网络上。 mrouted每120秒多播一个IGMP_HOST_MEMBERSHIP_QUERY数据报，并把IGMP_HOST_MEMBERSHIP_REPORT的结果汇编到与每个网络相关的成员关系数组中。这个数组不是我们在第13章讲的成员关系表。

mrouted根据收集到的信息构造多播路由选择表。多播组表也提供信息，用来抑制向没有目的组成员的多播互联网区进行多播。

只为物理接口维护这样的成员关系数组。对其他多播路由器来说，隧道是点到点接口，所以无需组成员关系信息。

我们在图14-14中看到，v_lcl_grps指向一个IP多播组数组。mrouted用DVMRP_ADD_LGRP和DVMRP_DEL_LGRP命令维护这个表。两个命令都带了一个 lgrpctl结构(图14-19)。

ip_mroute.h

```
87 struct lgrplctl {
88     vifi_t  lgc_vifi;
89     struct in_addr lgc_gaddr;
90 };
```

ip_mroute.h

图14-19 lgrpctl 结构

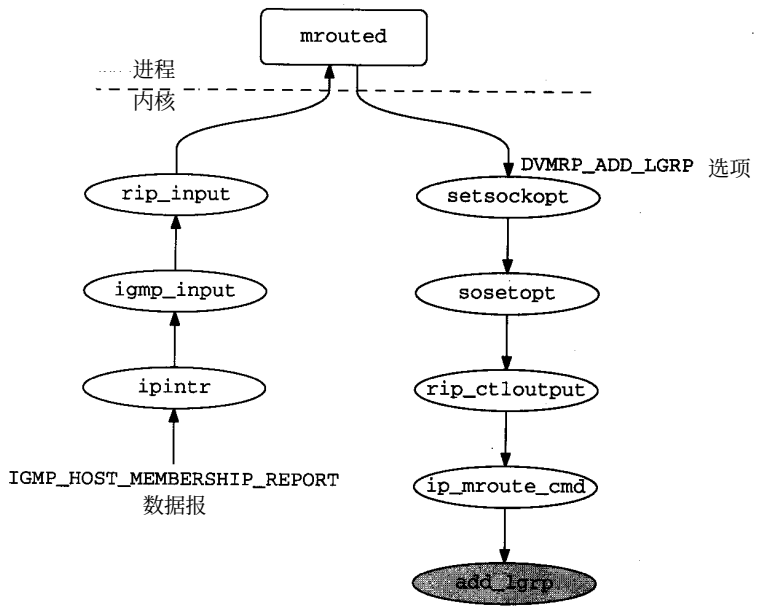


图14-20 IGMP报告处理

87-90 lgc_vifi和lgc_gaddr标识{接口，组}对。接口下标(无符号短整数lgc_vifi)标识一个虚拟接口，而不是物理接口。

当收到一个IGMP_HOST_MEMBERSHIP_REPORT时，调用图14-20所示的函数。

14.6.1 add_lgrp函数

mrouted检查到达IGMP报告的源地址，确定是哪个子网，从而确定报告是哪个接口接

ip_mroute.c

```
291 static int
292 add_lgrp(gcp)
293 struct lgrplctl *gcp;
294 {
295     struct vif *vifp;
296     int     s;

297     if (gcp->lgc_vifi >= numvifs)
298         return (EINVAL);

299     vifp = viftable + gcp->lgc_vifi;
300     if (vifp->v_lcl_addr.s_addr == 0 || (vifp->v_flags & VIFF_TUNNEL))
301         return (EADDRNOTAVAIL);

302     /* If not enough space in existing list, allocate a larger one */
303     s = splnet();
304     if (vifp->v_lcl_grps_n + 1 >= vifp->v_lcl_grps_max) {
305         int     num;
306         struct in_addr *ip;

307         num = vifp->v_lcl_grps_max;
308         if (num <= 0)
309             num = 32;           /* initial number */
310         else
311             num += num;         /* double last number */
312         ip = (struct in_addr *) malloc(num * sizeof(*ip),
313                                        M_MRTABLE, M_NOWAIT);
314         if (ip == NULL) {
315             splx(s);
316             return (ENOBUFS);
317         }
318         bzero((caddr_t) ip, num * sizeof(*ip));     /* XXX paranoid */
319         bcopy((caddr_t) vifp->v_lcl_grps, (caddr_t) ip,
320               vifp->v_lcl_grps_n * sizeof(*ip));

321         vifp->v_lcl_grps_max = num;
322         if (vifp->v_lcl_grps)
323             free(vifp->v_lcl_grps, M_MRTABLE);
324         vifp->v_lcl_grps = ip;

325         splx(s);
326     }
327     vifp->v_lcl_grps[vifp->v_lcl_grps_n++] = gcp->lgc_gaddr;

328     if (gcp->lgc_gaddr.s_addr == vifp->v_cached_group)
329         vifp->v_cached_result = 1;

330     splx(s);
331     return (0);
332 }
```

ip_mroute.c

图14-21 add_lgrp 函数：DVMRP_ADD_GLRP 命令

收的。根据这个信息，mrouted为该接口设置DVMRP_ADD_LGRP命令，更新内核中的成员关系表。这个信息也被送到多播路由选择算法，更新路由选择表。图 14-21显示了add_lgrp函数。

1. 验证增加请求

291-301 如果该请求标识了一个无效接口，就返回 EINVAL。如果没有使用该接口或它是一个隧道，则返回EADDRNOTAVAIL。

2. 如果需要，扩展组数组

302-326 如果新组无法放在当前的组数组中，就分配一个新的数组。第一次为接口调用add_lgrp函数时，分配一个能装32个组的数组。

每次数组被填满后，add_lgrp就分配一个两倍于前面数组大小的新数组。 Malloc负责分配， bzero负责清零， bcopy把旧数组中的内容复制到新数组中。更新最大入口数v_lcl_grps_max，释放旧数组(如果有的话)，把新数组和v_lcl_grps连接到vif入口。

“偏执狂(paranoid)”评论指出，无法保证malloc分配的内存全部是0。

3. 增加新的组

327-332 新组被复制到下一个可用的入口，如果高速缓存中已经存放了新组，就把高速缓存标记为有效。

查找高速缓存中包含一个地址 v_cached_group，以及一个高速缓存的查找结果v_cached_result。grplst_member函数在搜索成员关系数组之前，总是先查一下这个高速缓存。如果给定的组与v_cached_group匹配，就返回高速缓存的查找结果；否则，搜索成员关系数组。

14.6.2 del_lgrp函数

如果在270秒内，没有收到该组任何成员关系的报告，则每个接口的组信息超时。mrouted维护适当的定时器，并当信息超时后，发布 DVMRP_DEL_LGRP命令。图14-22显示了del_lgrp。

1. 验证接口下标

337-347 如果请求标识无效接口，叫返回 EINVAL。如果该接口没有使用或是一个隧道，则返回EADDRNOTAVAIL。

2. 更新查找高速缓存

348-350 如果要删除的组在高速缓存里，就把查找结果设成 0(假)。

3. 删除组

351-364 如果在成员关系表中没有找到该组，则在 error中发布EADDRNOTAVAIL。for循环搜索与该接口相关的成员关系数组。如果 same(是一个宏，用bcmp比较两个地址)为真，则清除error，把组计数器加1。bcopy移动后续的数组入口，删除该组， del_lgrp跳出该循环。

如果循环结束，没有找到匹配，则返回 EADDRNOTAVAIL；否则返回0。

ip_mroute.c

```
337 static int
338 del_lgrp(gcp)
339 struct lgrplctl *gcp;
340 {
341     struct vif *vifp;
342     int     i, error, s;

343     if (gcp->lgc_vifi >= numvifs)
344         return (EINVAL);
345     vifp = viftable + gcp->lgc_vifi;
346     if (vifp->v_lcl_addr.s_addr == 0 || (vifp->v_flags & VIFF_TUNNEL))
347         return (EADDRNOTAVAIL);

348     s = splnet();

349     if (gcp->lgc_gaddr.s_addr == vifp->v_cached_group)
350         vifp->v_cached_result = 0;

351     error = EADDRNOTAVAIL;
352     for (i = 0; i < vifp->v_lcl_grps_n; ++i)
353         if (same(&gcp->lgc_gaddr, &vifp->v_lcl_grps[i])) {
354             error = 0;
355             vifp->v_lcl_grps_n--;
356             bcopy((caddr_t) & vifp->v_lcl_grps[i + 1],
357                   (caddr_t) & vifp->v_lcl_grps[i],
358                   (vifp->v_lcl_grps_n - i) * sizeof(struct in_addr));
359             error = 0;
360             break;
361         }
362     splx(s);
363     return (error);
364 }
```

ip_mroute.c

图14-22 del_lgrp 函数：DVMRP_ DEL_LGRP命令

14.6.3 grplst_member函数

在转发多播时，查询成员关系数组，以免把数据报发到没有目的组成员的网络上。图 14-23显示的grplst_member函数，搜索整个表，寻找给定组地址。

ip_mroute.c

```
368 static int
369 grplst_member(vifp, gaddr)
370 struct vif *vifp;
371 struct in_addr gaddr;
372 {
373     int     i, s;
374     u_long  addr;

375     mrtstat.mrts_grp_lookups++;

376     addr = gaddr.s_addr;
377     if (addr == vifp->v_cached_group)
378         return (vifp->v_cached_result);

379     mrtstat.mrts_grp_misses++;

380     for (i = 0; i < vifp->v_lcl_grps_n; ++i)
```

图14-23 grplst_member 函数

```
381         if (addr == vifp->v_lcl_grps[i].s_addr) {
382             s = splnet();
383             vifp->v_cached_group = addr;
384             vifp->v_cached_result = 1;
385             splx(s);
386             return (1);
387         }
388     s = splnet();
389     vifp->v_cached_group = addr;
390     vifp->v_cached_result = 0;
391     splx(s);
392     return (0);
393 }
```

ip_mroute.c

图14-23 (续)

1. 检查高速缓存

368-379 如果请求的组在高速缓存中，则返回高速缓存的结果，不搜索成员关系数组。

2. 搜索成员关系数组

380-390 对数组进行线性搜索，确定组是否在其中。如果找到，就更新高速缓存以记录匹配的值，并返回1；如果没有找到，就更新高速缓存记录丢失的，并返回0。

14.7 多播选路

正如在本章开始提到的，我们不给出 mrouted实现的TRPB算法，但给出一个有关该机制的综述，描述内核的多播路由选择表和多播路由选择函数。图 14-24显示了一个我们用于解释该算法的示例多播网络。

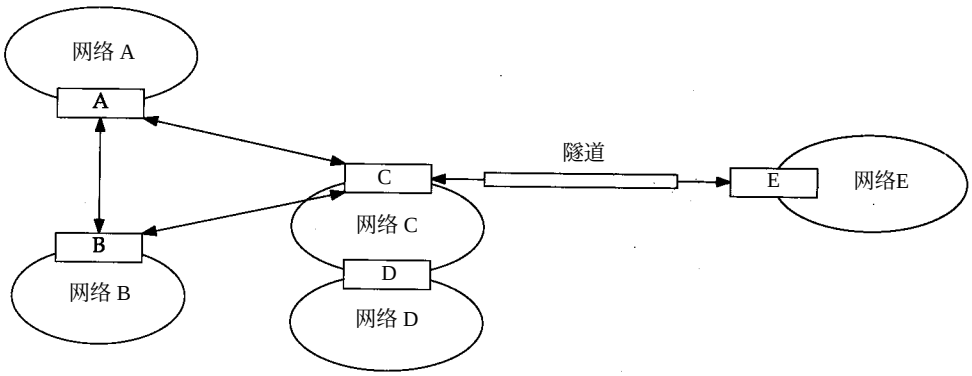


图14-24 多播网络示例

图14-24中，方框代表路由器，椭圆代表连接到路由器的多播网络。例如，路由器 D可以在网络D和网络C上多播。路由器C可以向网络C多播，通过点到点接口向路由器 A和B多播，并可以通过一个多播隧道向路由器E多播。

最简单的路由选择办法是，从互联网拓扑中选出一个子网，形成一个生成树。如果每个路由器都沿着生成树转发多播，则各路由器最终会收到数据报。图 14-25显示了示例网络的一个生成树。其中，网络A上的主机S是多播数据报的源。

有关生成树的讨论，参见 [Tanenbaum 1989] 或 [Perlman 1992]。

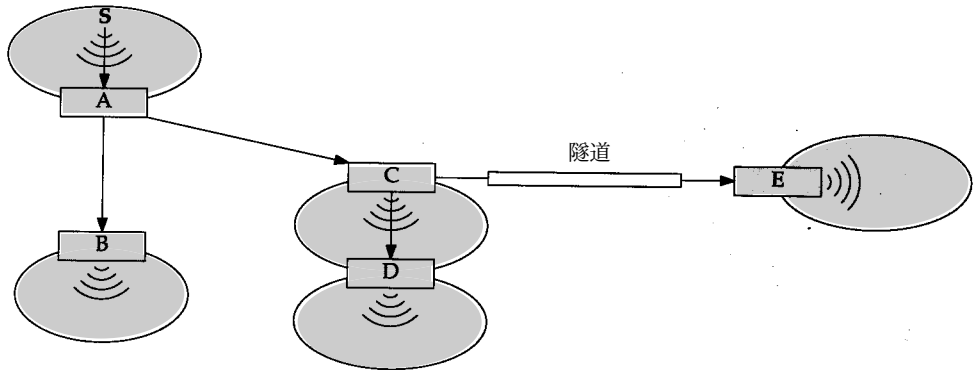


图14-25 网络A的生成树

这个生成树是根据从各网络回到网络A上的源站的最短逆向路径(*reverse path*)构造的。图14-25的生成树中，省略了路由器B和C之间的线路。源站和路由器A之间的箭头，以及路由器B和C之间的箭头，强调了多播网络是生成树的一部分。

如果用同一生成树转发来自网络C的数据报，为了在网络B上收到，数据报经过的转发路径将大于需要的长度。RFC 1075提出的算法为每个潜在的源站计算了一个单独的生成树，以避免这种情况。路由选择表为每条路由记录了一个网络号和子网掩码，所以一条路由可以应用到源子网内的任意主机。

因为构造生成树是为了给源站的数据报提供最短逆向路径，而每个网络都接收所有多播数据报，所以这个过程称为逆向路径广播(*reverse path broadcast*)即RPB。

RPB没有任何多播组成员信息，使许多数据报被不必要地转发到没有目的组成员的网络上。如果，除了计算生成树外，该路由选择算法还能记录哪些网络是叶子，注意到每个网络上的组成员关系，那么，连到叶子网络的路由器就可以避免把数据报转发到没有目的组成员的网络上去。这称为截断逆向路径广播(TRPB)，2.0版的`mrouted`在IGMP帮助下记录叶子网络上的成员关系，从而实现这一算法。

图14-26显示了TRPB算法的应用。多播来自网络C上的源站，并在网络B上有一个目的组成员。

我们用图14-26说明Net/3多播路由选择表中使用的名词。在这个例子中，有阴影的网络和

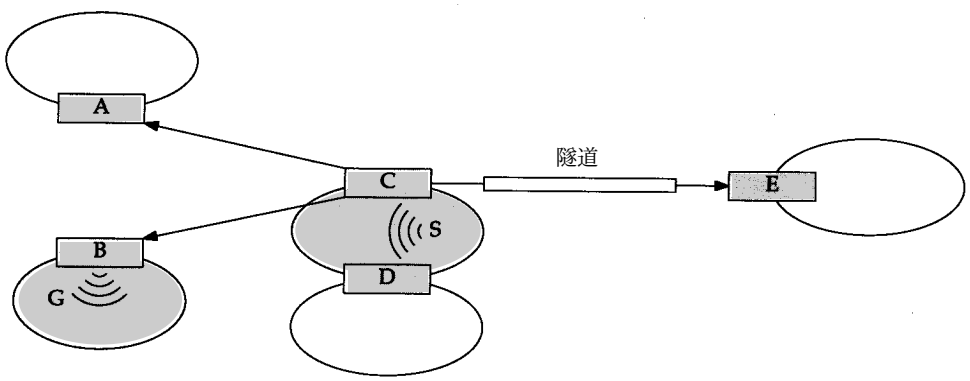


图14-26 网络C的TRPB路由选择

路由器收到来自网络C上源站的数据报。A和B之间的线路不属于生成树，C与D之间没有连接，因为C和D直接收到源站发送的多播。

在这个图中，网络A、B、D和E是叶子网络。路由器C接收多播，并通过连到路由器A、B和E的接口将其转发——尽管把它发给A和E都是浪费。这是TRPB算法的缺点。

路由器C上与网络C相关的接口叫做父亲，因为路由器C期望用它接收来自网络C的多播。从路由器C到路由器A、B和E的接口叫做儿子接口。对路由器A来说，点到点接口是来自C的源分组的父亲，到网络A的接口是儿子。接口相对于数据报的源站，被标识为父亲和儿子。只在相关的儿子接口上转发多播数据报，不在父亲接口上转发多播。

继续我们的例子，因为网络A、D和E是叶子网络，并且没有目的组成员，所以它们没有阴影。在路由器处截断生成树，也不把数据报转发到这些网络上去。路由器B把数据报转发到网络B上，因为B上有一个目的组成员。为实现截断算法，接收数据报的所有路由器都在自己的viftable中查询与每个虚拟接口相关的组表。

对该多播路由选择算法的最后一个改进叫做逆向路径多播 (reverse path multicasting，RPM)。RPM的目的是修剪(prune)各生成树，避免在没有目的组成员的分支上发送数据报。在图14-26中，RPM可以避免路由器C向A和E发送数据报，因为在这两个分支上没有目的多播组的成员。3.3版的mrouted实现了RPM。

图14-27是我们的示例网络，但这一次，只有那些RPM算法选路数据报能到达的路由器和网络才有阴影。

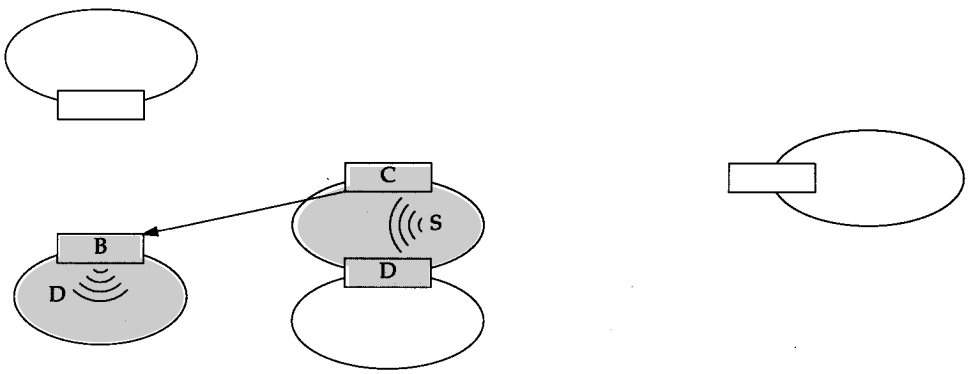


图14-27 网络C的RMP路由选择

为了计算生成树对应的路由表，多播路由器和邻近的多播路由器通信，发现多播互联网拓扑和多播组成员的位置。在Net/3中，用DVMRP进行这种通信。DVMRP作为IGMP数据报传送，发给224.0.0.4组，该组是给DVMRP通信保留的(图12-1)。

在图12-39中，我们看到，多播路由器总是接受到达的IGMP分组，把它们传给igmp_input和rip_input，然后mrouted在一个原始IGMP插口上读它们。mrouted把DVMRP报文发送到同一原始IGMP插口上的其他多播路由器。

实现这些算法需要的有关RPB、TRPB、RPM以及DVMRP报文的其他细节参见[Deering和Cheriton 1990]和mrouted的源代码。

Internet上还使用了其他多播路由选择协议。Proteon路由器实现了RFC 1584 [Moy 1994]提出的MOSPF协议。Cisco从操作软件的10.2版开始实现了PIM(Protocol Independent

Multicasting)。[Deering et al1994]描述了PIM。

14.7.1 多播选路表

现在我们描述Net/3中实现的多播路由选择。内核的多播路由选择表是作为一个有 64个入口的散列表实现的 (MRTHASHIZ)。该表保存在全局数组 mrttable中，每个入口指向一个mrt结构的链表，如图14-28所示。

ip_mroute.h

```
120 struct mrt {
121     struct in_addr mrt_origin;  /* subnet origin of multicasts */
122     struct in_addr mrt_originmask;  /* subnet mask for origin */
123     vifi_t  mrt_parent;         /* incoming vif */
124     vifbitmap_t mrt_children;   /* outgoing children vifs */
125     vifbitmap_t mrt_leaves;     /* subset of outgoing children vifs */
126     struct mrt *mrt_next;       /* forward link */
127 };
```

ip_mroute.h

图14-28 mrt 结构

120-127 mrtc_origin和mrtc_originmask标识表中的一个入口。mrtc_parent是虚拟接口的下标，该虚拟接口上预期有来自起点的所有多播数据报。 mrtc_children是一个位图，标识外出的接口。 mrtc_leaves也是一个位图，里面标识多播路由选择树中也是叶子的外出接口。当多条路由散列到同一个数组入口时，最后一个成员 mrt_next实现该入口的一个链表。

图14-29是多播选路表的整体结构。各 mrt结构都放在一个散列链上，该散列链与nethash(图14-31)函数返回的值对应。

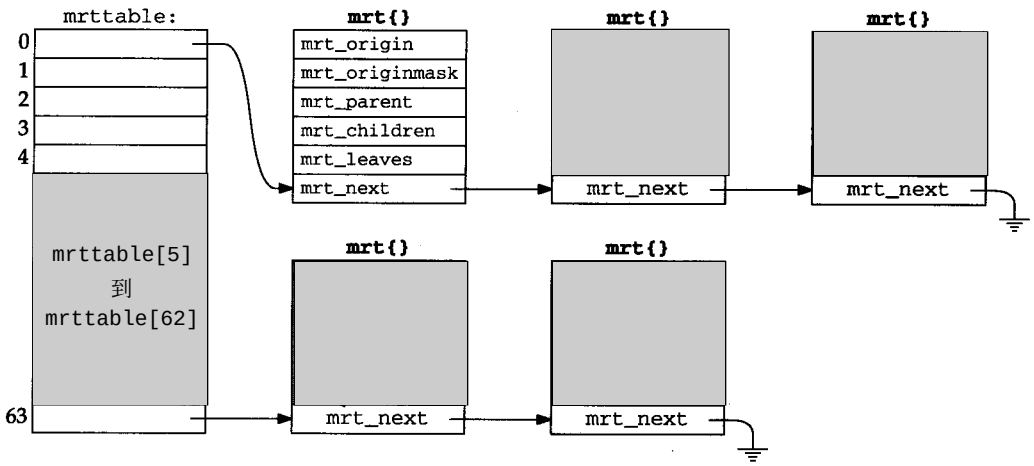


图14-29 多播选路表

内核维护的多播选路表是mrouted维护的多播选路表的一个子集，其中的信息足够内核支持多播转发。发送内核表更新和 DVMRP_ADD_MRT命令，其中包含图 14-30显示的mrtctl结构。

95-101 mrtctl结构的5个成员携带了我们谈到的mrouted和内核之间的信息(图14-28)。

多播选路表的键值是多播数据报的源 IP地址。nethash(图14-31)实现该用于该表的散列

算法。它接受源IP地址，并返回0~63之间的一个值(MRTHASHSIZ-1)。

ip_mroute.h

```
 95 struct mrtctl {
 96     struct in_addr mrtc_origin; /* subnet origin of multicasts */
 97     struct in_addr mrtc_originmask;     /* subnet mask for origin */
 98     vifi_t  mrtc_parent;        /* incoming vif */
 99     vifbitmap_t mrtc_children;  /* outgoing children vifs */
100     vifbitmap_t mrtc_leaves;    /* subset of outgoing children vifs */
101 };
```

ip_mroute.h

图14-30 mrtctl 结构

ip_mroute.c

```
398 static  u_long
399 nethash(in)
400 struct in_addr in;
401 {
402     u_long  n;

403     n = in_netof(in);
404     while ((n & 0xff) == 0)
405         n >>= 8;
406     return (MRTHASHMOD(n));
407 }
```

ip_mroute.c

图14-31 nethash 结构

398-407 in_netof返回in，主机部分设置为全0，在n中仅留下发送主机的A、B和C类网络。右移结果，直到低8位非零为止。MRTHASHMOD是

```
#define MRTHASHMOD(h) ((h) & (MRTHASHSIZ - 1))
```

把低8位与63进行逻辑与运算，留下低6位，这是0~63之间的一个整数。

用两个函数调用(nethash和in_netof)计算散列值，作为散列32 bit地址值太过昂贵了。

14.7.2 del_mrt函数

mrouted守护程序通过DVMRP_ADD_MRT和DVMRP_DEL_MRT命令在内核的多播选路表中增加或删除表项。图14-32显示了del_mrt函数。

ip_mroute.c

```
451 static int
452 del_mrt(origin)
453 struct in_addr *origin;
454 {
455     struct mrt *rt, *prev_rt;
456     u_long  hash = nethash(*origin);
457     int     s;

458     for (prev_rt = rt = mrttable[hash]; rt; prev_rt = rt, rt = rt->mrt_next)
459         if (origin->s_addr == rt->mrt_origin.s_addr)
460             break;
461     if (!rt)
462         return (ESRCH);

463     s = splnet();
```

图14-32 del_mrt 函数：DVMRP_DEL_MRT 命令

```
464     if (rt == cached_mrt)
465         cached_mrt = NULL;
466     if (prev_rt == rt)
467         mrttable[hash] = rt->mrt_next;
468     else
469         prev_rt->mrt_next = rt->mrt_next;
470     free(rt, M_MRTABLE);
471     splx(s);
472     return (0);
473 }
```

ip_mroute.c

图14-32 (续)

1. 找到路由入口

451-462 for循环从hash标识的入口开始(在nethash中定义时初始化)。如果没有找到入口，则返回ESRCH。

2. 删除路由入口

463-473 如果该入口在高速缓存中，则高速缓存也无效了。从散列链上把该入口断开，并且释放。当匹配入口在表的最前面时，需要if语句处理这一特殊情况。

14.7.3 add_mrt函数

add_mrt函数如图14-33所示。

ip_mroute.c

```
411 static int
412 add_mrt(mrtcp)
413 struct mrtctl *mrtcp;
414 {
415     struct mrt *rt;
416     u_long  hash;
417     int     s;
418     if (rt = mrtfind(mrtcp->mrtc_origin)) {
419         /* Just update the route */
420         s = splnet();
421         rt->mrt_parent = mrtcp->mrtc_parent;
422         VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
423         VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
424         splx(s);
425         return (0);
426     }
427     s = splnet();
428     rt = (struct mrt *) malloc(sizeof(*rt), M_MRTABLE, M_NOWAIT);
429     if (rt == NULL) {
430         splx(s);
431         return (ENOBUFS);
432     }
433     /*
434      * insert new entry at head of hash chain
435      */
436     rt->mrt_origin = mrtcp->mrtc_origin;
437     rt->mrt_originmask = mrtcp->mrtc_originmask;
438     rt->mrt_parent = mrtcp->mrtc_parent;
```

图14-33 add_mrt 函数：处理DVMRP_ADD_MRT 命令

```
439     VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
440     VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
441     /* link into table */
442     hash = nethash(mrtcp->mrtc_origin);
443     rt->mrt_next = mrttable[hash];
444     mrttable[hash] = rt;

445     splx(s);
446     return (0);
447 }
```

ip_mroute.c

图14-33 (续)

1. 更新存在的路由

411-427 如果请求的路由已经在路由表中，则把新的信息复制到该路由中，add_mrt返回。

2. 分配新路由

428-447 在新分配的mbuf中，根据增加请求传递的mrtctl结构，构造一个mrt结构。从mrtc_origin计算出散列下标，并把新路由插入散列链的第一个入口。

14.7.4 mrtfind函数

mrtfind函数负责搜索多播选路表。如图14-34所示。把数据报的源站地址传给mrtfind，mrtfind返回一个指向匹配mrt结构的指针；如果没有匹配，则返回一个空指针。

1. 检查路由查询高速缓存

477-488 把给定的源IP地址(orgin)与高速缓存中的原始掩码做逻辑与运算。如果结果与cached_origin匹配，则返回高速缓存的入口。

ip_mroute.c

```
477 static struct mrt *
478 mrtfind(origin)
479 struct in_addr origin;
480 {
481     struct mrt *rt;
482     u_int   hash;
483     int     s;

484     mrtstat.mrts_mrt_lookups++;

485     if (cached_mrt != NULL &&
486         (origin.s_addr & cached_originmask) == cached_origin)
487         return (cached_mrt);

488     mrtstat.mrts_mrt_misses++;

489     hash = nethash(origin);
490     for (rt = mrttable[hash]; rt; rt = rt->mrt_next)
491         if ((origin.s_addr & rt->mrt_originmask.s_addr) ==
492             rt->mrt_origin.s_addr) {
493             s = splnet();
494             cached_mrt = rt;
495             cached_origin = rt->mrt_origin.s_addr;
496             cached_originmask = rt->mrt_originmask.s_addr;
497             splx(s);
498             return (rt);
499         }
500     return (NULL);
501 }
```

ip_mroute.c

图14-34 mrtfind 函数

2. 检查散列表

489-501 nethash返回该路由入口的散列下标。for循环搜索散列链找到匹配的路由。当找到一个匹配时，更新高速缓存，返回一个指向该路由的指针。如果没有找到匹配，则返回一个空指针。

14.8 多播转发：ip_mforward函数

内核实现了整个多播转发。我们在图 12-39中看到，当ip_mrouter非空时，也就是mrouted在运行时，ipintr把到达数据报传给ip_mforward。

我们在图 12-40中看到，ip_output可以把本地主机产生的多播数据报传给 ip_mforward，由ip_mforward为这些数据报选路到除ip_output选定的接口以外的其他接口上去。

与单播转发不同，每当多播数据报被转发到某个接口上时，就为该数据报产生一个备份。例如，如果本地主机是一个多播路由器，并且连接到三个不同的网络，则系统产生的多播数据报被分别复制三份，在三个接口上等待输出。另外，如果应用程序设置了多播环回标志位，或者任何输出的接口也接收它自己的传送，则数据报也将被复制，等待输入。

图14-35显示了一个到达某个物理接口的多播数据报。

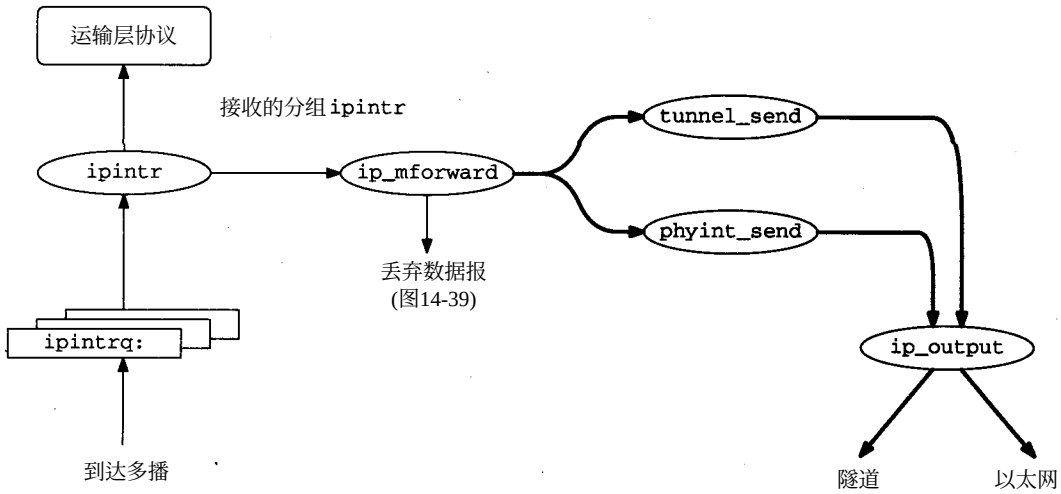


图14-35 到达某个物理接口的多播数据报

在图14-35中，数据报到达的接口是目的多播组的一个成员，所以数据报被传给运输层协议等待输入处理。该数据报也被传给 ip_mforward，在这里它被复制和转发到一个物理接口和一个隧道上(带粗线的箭头)，这两个必须都不和接收接口相同。

图14-36显示了一个到达某隧道的多播数据报。

在图14-36中，用带虚线的箭头表示与该隧道的本地端有关的物理接口，数据报就在这一接口上到达。数据报被传给ip_mforward，我们将在图14-37看到，因为分组到达一个隧道，所以ip_mforward返回一个非零值。这导致ipintr不再把该分组传给运输层协议。

ip_mforward从分组中取出隧道选项，查询多播选路表，并且，在本例中，还把分组转发到另一个隧道以及到达的物理接口上去，用带细线的箭头表示。这是可行的，因为多播选

路表是根据虚拟接口，而不是物理接口。

在图14-36中，我们假定物理接口是目的多播组的成员，所以ip_output把该数据报传给ip_mloopback，ip_mloopback把它送到队列中等待ipintr的处理(带粗线的箭头)。然后，分组又被传给ip_mforward，并被这个函数丢弃（练习14.4)。这一次，ip_mforward返回0(因为分组是在物理接口上到达的)，所以ipintr接受该数据报，并进行输入处理。

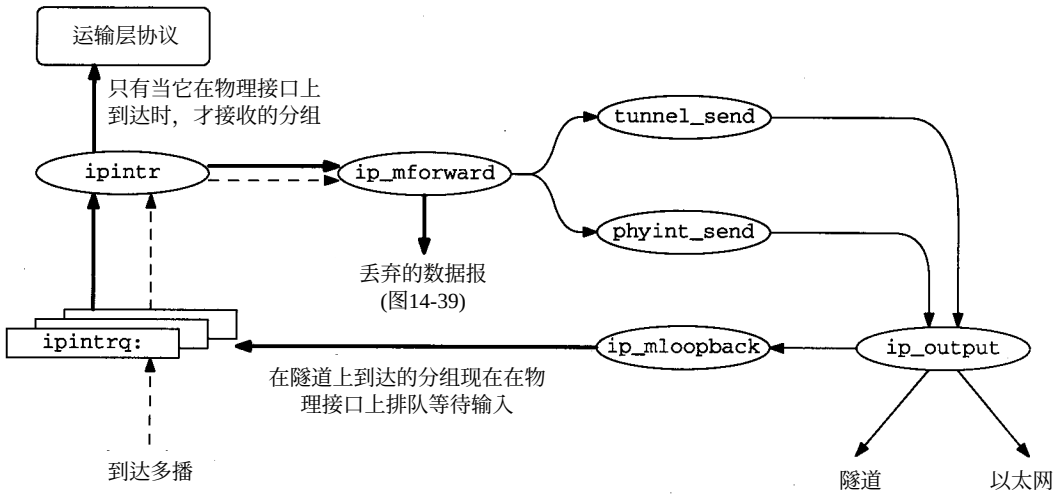


图14-36 到达某个多播隧道的多播数据报

我们分三部分说明多播转发程序：

- 隧道输入处理(图14-37)；
- 转发条件合格(图14-39)；和
- 转发到出去的接口上(图14-40)。

ip_mroute.c

```
516 int
517 ip_mforward(m, ifp)
518 struct mbuf *m;
519 struct ifnet *ifp;
520 {
521     struct ip *ip = mtod(m, struct ip *);
522     struct mrt *rt;
523     struct vif *vifp;
524     int     vifi;
525     u_char *ipoptions;
526     u_long  tunnel_src;

527     if (ip->ip_hl < (IP_HDR_LEN + TUNNEL_LEN) >> 2 ||
528         (ipoptions = (u_char *) (ip + 1))[1] != IPOPT_LSRR) {
529         /* Packet arrived via a physical interface. */
530         tunnel_src = 0;
531     } else {
532         /*
533          * Packet arrived through a tunnel.
534          * A tunneled packet has a single NOP option and a
```

图14-37 ip_mforward 函数：到达隧道

```
535          * two-element loose-source-and-record-route (LSRR)
536          * option immediately following the fixed-size part of
537          * the IP header.  At this point in processing, the IP
538          * header should contain the following IP addresses:
539          *
540          * original source          - in the source address field
541          * destination group        - in the destination address field
542          * remote tunnel end-point  - in the first  element of LSRR
543          * one of this host's addrs - in the second element of LSRR
544          *
545          * NOTE: RFC-1075 would have the original source and
546          * remote tunnel end-point addresses swapped.  However,
547          * that could cause delivery of ICMP error messages to
548          * innocent applications on intermediate routing
549          * hosts!  Therefore, we hereby change the spec.
550          */
551         /* Verify that the tunnel options are well-formed.  */
552         if (ipoptions[0] != IPOPT_NOP ||
553             ipoptions[2] != 11 ||    /* LSRR option length    */
554             ipoptions[3] != 12 ||    /* LSRR address pointer */
555             (tunnel_src = *(u_long *) (&ipoptions[4])) == 0) {
556             mrtstat.mrts_bad_tunnel++;
557             return (1);
558         }
559         /* Delete the tunnel options from the packet. */
560         ovbcopy((caddr_t) (ipoptions + TUNNEL_LEN), (caddr_t) ipoptions,
561                 (unsigned) (m->m_len - (IP_HDR_LEN + TUNNEL_LEN)));
562         m->m_len -= TUNNEL_LEN;
563         ip->ip_len -= TUNNEL_LEN;
564         ip->ip_hl -= TUNNEL_LEN >> 2;
565     }
```

ip_mroute.c

图14-37 (续)

516-526 ip_mforward的两个参数是：一个指向包含该数据报的 mbuf链的指针；另一个是指向接收接口ifnet结构的指针。

1. 到达物理接口

527-530 为了区分在同一物理接口上到达的多播数据报是否经过隧道，要检查 IP首部的特征LSRR选项。如果首部太小，无法包含该选项；或者该选项不是以一个后面跟着一个 LSRR选项的NOP开始，就假定该数据报是在一个物理接口上到达的，并把 tunnel_src设为0。

2. 到达隧道

531-558 如果数据报看起来像是从隧道上到达的，就检查选项，验证格式是否正确。如果选项的格式不符合多播隧道，则 ip_mforward返回1，指示应该把该数据报丢弃。图 14-38是隧道选项的结构。

在图14-38中，我们假定数据报里没有其他选项，但不是必须这样的。任何其他IP选项都可能出现在 LSRR选项的后面，因为隧道开始端的多播路由器总是把 LSRR选项插在所有其他选项之前。

3. 删除隧道选项

559-565 如果选项正确，就把后面的选项和数据向前移动，调整 mbuf首部的m_len和IP 首部的ip_len和ip_hl的值，然后删除隧道选项(图14-38)。

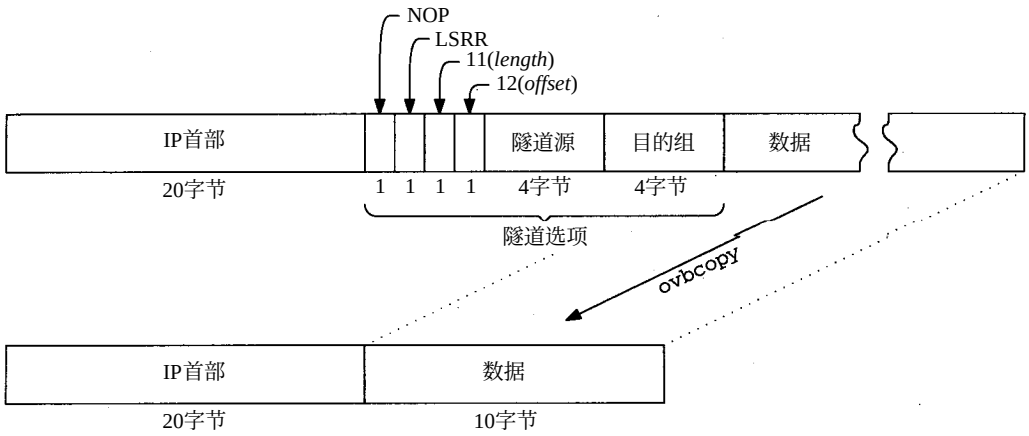


图14-38 多播隧道选项

ip_mforward经常把tunnel_source作为返回值。当数据报从隧道上到达时，这个值只能是非零的。当ip_mforward返回非零值时，它的调用方就丢弃该数据报。对 ipintr来说，这意味着在隧道上到达的一个数据报被传给 ip_mforward，并且被ipintr丢弃。转发程序取出隧道信息，复制数据报，用 ip_output将其发送出去；如果接口是目的多播组的成员，则ip_output调用ip_mloopback。

ip_mforward的下一部分显示在图 14-39中，在这部分程序中，如果数据报不符合转发的条件，就丢弃它。

ip_mroute.c

```
566     /*
567      * Don't forward a packet with time-to-live of zero or one,
568      * or a packet destined to a local-only group.
569      */
570     if (ip->ip_ttl <= 1 ||
571         ntohl(ip->ip_dst.s_addr) <= INADDR_MAX_LOCAL_GROUP)
572         return ((int) tunnel_src);

573     /*
574      * Don't forward if we don't have a route for the packet's origin.
575      */
576     if (!(rt = mrtfind(ip->ip_src))) {
577         mrtstat.mrts_no_route++;
578         return ((int) tunnel_src);
579     }
580     /*
581      * Don't forward if it didn't arrive from the parent vif for its origin.
582      */
583     vifi = rt->mrt_parent;
584     if (tunnel_src == 0) {
585         if ((viftable[vifi].v_flags & VIFF_TUNNEL) ||
586             viftable[vifi].v_ifp != ifp)
587             return ((int) tunnel_src);
588     } else {
589         if (!(viftable[vifi].v_flags & VIFF_TUNNEL) ||
590             viftable[vifi].v_rmt_addr.s_addr != tunnel_src)
591             return ((int) tunnel_src);
592     }
```

ip_mroute.c

图14-39 ip_mforward 函数：转发可行性检查

4. 超时的TTL或本地多播

566-572 如果ip_ttl是0或1，那么数据报已经到了生存期的最后，不再转发它。如果目的组小于或等于INADDR_MAX_LOCAL_GROUP(几个224.0.0.x组，图12-1)，则不允许数据报离开本地网络，也不转发它。在两种情况下，都把tunnel_src返回给调用方。

3.3版的mrouted支持对某些目的多播组的管理辖域。可把接口配置成丢弃所有寻址到这些组的数据报，与224.0.0.x组的自动辖域类似。

5. 没有路由可用

573-579 如果mrtfind无法根据数据报中的源地址找到一条路由，则函数返回。没有路由，多播路由器无法确定把数据报转发到哪个接口上去。这种情况可能发生在，比如，多播数据报在mrouted更新多播选路表之前到达。

6. 在没有想到的接口上到达

580-592 如果数据报到达某个物理接口，但系统本来预想它应该到达某个隧道或其他物理接口，则ip_mforward返回；如果数据报到达某个隧道，但系统本来预想它应该在某个物理接口或其他隧道上到达，则ip_mforward也返回。产生这些情况的原因是，当组成员关系或网络的物理拓扑发生变化后，正在更新选路表时，数据报到达。

ip_mforward的最后一部分(图14-40)把该数据报在多播路由入口所指定的每个输出接口上发送。

ip_mroute.c

```
593     /*
594      * For each vif, decide if a copy of the packet should be forwarded.
595      * Forward if:
596      *       - the ttl exceeds the vif's threshold AND
597      *       - the vif is a child in the origin's route AND
598      *       - ( the vif is not a leaf in the origin's route OR
599      *           the destination group has members on the vif )
600      *
601      * (This might be speeded up with some sort of cache -- someday.)
602      */
603     for (vifp = viftable, vifi = 0; vifi < numvifs; vifp++, vifi++) {
604         if (ip->ip_ttl > vifp->v_threshold &&
605             VIFM_ISSET(vifi, rt->mrt_children) &&
606             (!VIFM_ISSET(vifi, rt->mrt_leaves) ||
607              grplst_member(vifp, ip->ip_dst))) {
608             if (vifp->v_flags & VIFF_TUNNEL)
609                 tunnel_send(m, vifp);
610             else
611                 phyint_send(m, vifp);
612         }
613     }
614     return ((int) tunnel_src);
615 }
```

ip_mroute.c

图14-40 ip_mforward 函数：转发

593-615 对viftable中的每个接口，如果以下条件满足，则在该接口上发送数据报：

- 数据报的TTL大于接口的多播阈值；
- 接口是该路由的子接口；以及
- 接口没有和某个叶子网络相连。

如果该接口是一个叶子，那么只有当网络上有目的多播组成员时(也就是说，grplst_member返回一个非零值)，才输出该数据报。

tunnel_send在隧道接口上转发该数据报；用phyint_send在物理接口上转发。

14.8.1 phyint_send函数

为在物理接口上发送多播数据报，phyint_send(图14-41)在它传给ip_output的ip_moptions结构中，明确指定了输出接口。

ip_mroute.c

```
616 static void
617 phyint_send(m, vifp)
618 struct mbuf *m;
619 struct vif *vifp;
620 {
621     struct ip *ip = mtod(m, struct ip *);
622     struct mbuf *mb_copy;
623     struct ip_moptions *imo;
624     int     error;
625     struct ip_moptions simo;

626     mb_copy = m_copy(m, 0, M_COPYALL);
627     if (mb_copy == NULL)
628         return;

629     imo = &simo;
630     imo->imo_multicast_ifp = vifp->v_ifp;
631     imo->imo_multicast_ttl = ip->ip_ttl - 1;
632     imo->imo_multicast_loop = 1;

633     error = ip_output(mb_copy, NULL, NULL, IP_FORWARDING, imo);
634 }
```

ip_mroute.c

图14-41 phyint_send 函数

616-634 m_copy复制输出的数据报。ip_moptions结构设置为强制在选定的接口上传送该数据报。递减TTL，允许多播环回。

数据报被传给ip_output。IP_FORWARDING标志位避免产生无限回路，使ip_output再次调用ip_mforward。

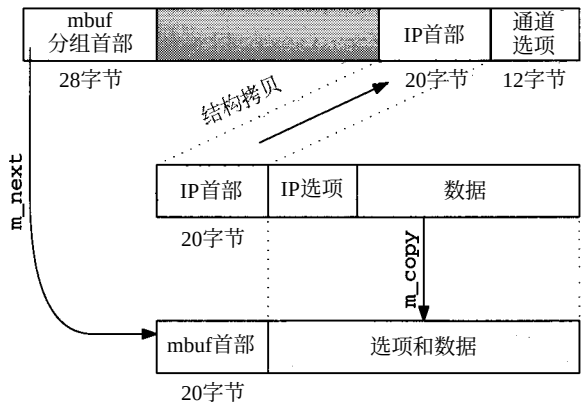


图14-42 插入隧道选项

14.8.2 tunnel_send函数

为了在隧道上发送数据报，tunnel_send(图14-43)必须构造合适的隧道选项，并将其插到输出数据报的首部。图14-42显示了tunnel_send如何为隧道准备分组。

ip_mroute.c

```
635 static void
636 tunnel_send(m, vifp)
637 struct mbuf *m;
638 struct vif *vifp;
639 {
640     struct ip *ip = mtod(m, struct ip *);
641     struct mbuf *mb_copy, *mb_opts;
642     struct ip *ip_copy;
643     int     error;
644     u_char *cp;

645     /*
646      * Make sure that adding the tunnel options won't exceed the
647      * maximum allowed number of option bytes.
648      */
649     if (ip->ip_hl > (60 - TUNNEL_LEN) >> 2) {
650         mrtstat.mrts_cant_tunnel++;
651         return;
652     }
653     /*
654      * Get a private copy of the IP header so that changes to some
655      * of the IP fields don't damage the original header, which is
656      * examined later in ip_input.c.
657      */
658     mb_copy = m_copy(m, IP_HDR_LEN, M_COPYALL);
659     if (mb_copy == NULL)
660         return;
661     MGETHDR(mb_opts, M_DONTWAIT, MT_HEADER);
662     if (mb_opts == NULL) {
663         m_freem(mb_copy);
664         return;
665     }
666     /*
667      * Make mb_opts be the new head of the packet chain.
668      * Any options of the packet were left in the old packet chain head
669      */
670     mb_opts->m_next = mb_copy;
671     mb_opts->m_len = IP_HDR_LEN + TUNNEL_LEN;
672     mb_opts->m_data += MSIZE - mb_opts->m_len;
```

ip_mroute.c

图14-43 tunnel_send 函数：验证和分配新首部

1. 隧道选项合适吗

635-652 如果IP首部内没有隧道选项的空间，tunnel_send立即返回，不再在隧道上转发该数据报。可能在其他接口上转发。

2. 复制数据报，为新首部和隧道选项分配mbuf

653-672 在调用m_copy时，复制的开始偏移是20(IP_HDR_LEN)。产生的mbuf链中包含了数据报的选项和数据报，但没有IP首部。mb_opts指向MGETHDR分配的一个新的数据报首部，这个新的数据报首部被放在mb_copy的前面。然后调整m_len和m_data的值，以容纳IP首部和隧道选项。

tunnel_send的第二部分，如图14-44所示，修改输出分组的首部，并发送该分组。

ip_mroute.c

```
673     ip_copy = mtod(mb_opts, struct ip *);
674     /*
675      * Copy the base ip header to the new head mbuf.
676      */
677     *ip_copy = *ip;
678     ip_copy->ip_ttl--;
679     ip_copy->ip_dst = vifp->v_rmt_addr;     /* remote tunnel end-point */
680     /*
681      * Adjust the ip header length to account for the tunnel options.
682      */
683     ip_copy->ip_hl += TUNNEL_LEN >> 2;
684     ip_copy->ip_len += TUNNEL_LEN;
685     /*
686      * Add the NOP and LSRR after the base ip header
687      */
688     cp = (u_char *) (ip_copy + 1);
689     *cp++ = IPOPT_NOP;
690     *cp++ = IPOPT_LSRR;
691     *cp++ = 11;                 /* LSRR option length */
692     *cp++ = 8;                  /* LSSR pointer to second element */
693     *(u_long *) cp = vifp->v_lcl_addr.s_addr;   /* local tunnel end-point */
694     cp += 4;
695     *(u_long *) cp = ip->ip_dst.s_addr;     /* destination group */

696     error = ip_output(mb_opts, NULL, NULL, IP_FORWARDING, NULL);
697 }
```

ip_mroute.c

图14-44 tunnel_send 函数：构造首部和发送

3. 修改IP首部

673-679 从原始mbuf链中把原始IP 首部复制到新分配的mbuf首部中。减少该首部的TTL，把目的地址改成隧道另一端的接口地址。

4. 构造隧道选项

680-664 调整ip_hl和ip_len的值以容纳隧道选项。隧道选项紧跟在IP 首部的后面：一个NOP，后面是LSRR码，LSRR选项的长度(11字节)，以及一个指向选项第二个地址的指针 (8字节)。源路由包括了本地隧道端点和后面的目的多播组地址 (图14-13)。

5. 发送经过隧道处理的数据报

665-697 现在，这个数据报看起来像一个有LSRR选项的单播数据报，因为它的目的地址是隧道另一端的单播地址。ip_output发送该数据报。当数据报到达隧道的另一端时，隧道选项被剥离，另一端可能会通过其他隧道将数据报继续转发。

14.9 清理：ip_mrouter_done函数

当mrouted结束时，它发布DVMRP_DONE命令，ip_mrouter_done函数(图14-45)处理这个命令。

161-186 这个函数在splnet上运行，避免与多播转发代码的任何交互。对每个物理多播接口，释放本地组表，并发布 SIOCDELMULTI命令，阻止接收多播数据报 (练习14.3) 。bzero清零整个viftable数组，并把numvifs设置成0。

187-198 释放多播选路表中的所有活动入口，bzero清零整个表，清零缓存，置位ip_mrouter。

多播选路表中的每个入口都可能是入口链表的第一个。这段代码只释放表的第一个入口，引起内存泄露。

ip_mroute.c

```
161 int
162 ip_mrouter_done()
163 {
164     vifi_t  vifi;
165     int     i;
166     struct ifnet *ifp;
167     int     s;
168     struct ifreq ifr;
169     s = splnet();
170     /*
171      * For each phyint in use, free its local group list and
172      * disable promiscuous reception of all IP multicasts.
173      */
174     for (vifi = 0; vifi < numvifs; vifi++) {
175         if (viftable[vifi].v_lcl_addr.s_addr != 0 &&
176             !(viftable[vifi].v_flags & VIFF_TUNNEL)) {
177             if (viftable[vifi].v_lcl_grps)
178                 free(viftable[vifi].v_lcl_grps, M_MRTABLE);
179             satosin(&ifr.ifr_addr)->sin_family = AF_INET;
180             satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
181             ifp = viftable[vifi].v_ifp;
182             (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);
183         }
184     }
185     bzero((caddr_t) viftable, sizeof(viftable));
186     numvifs = 0;
187     /*
188      * Free any multicast route entries.
189      */
190     for (i = 0; i < MRTHASHSIZ; i++)
191         if (mrttable[i])
192             free(mrttable[i], M_MRTABLE);
193     bzero((caddr_t) mrttable, sizeof(mrttable));
194     cached_mrt = NULL;
195     ip_mrouter = NULL;
196     splx(s);
197     return (0);
198 }
```

ip_mroute.c

图14-45 ip_mrouter_done 函数：DVMRP_DONE 命令

14.10 小结

本章我们描述了网际多播的一般概念和支持它的 Net/3内核中心专用函数。我们没有讨论mrouted的实现，有兴趣的读者可以得到源代码。

我们描述了虚拟接口表，讨论了物理接口和隧道之间的区别，以及 Net/3中用于实现隧道的LSRR选项。

我们说明了RPB、TRPB和RPM算法，描述了根据TRPB转发多播数据报的内核表，还讨论了父网络和叶子网络。

习题

14.1 在图14-25中，需要多少多播路由？

14.2 为什么splnet和splx保护对图14-23中组成员关系高速缓存的更新？

14.3 当某个接口用IP_ADD_MEMBERSHIP选项明确加入一个多播组后，如果向它发布SIOCDELMULTI，会发生什么？

14.4 当某个上隧道上到达一个数据报，并被ip_mforward接收后，可能会在转发到某个物理接口时，被ip_output环回。为什么当环回分组到达该物理接口时，ip_mforward会丢弃它呢？

14.5 重新设计组地址高速缓存，提高它的效率。

第15章 插 口 层

15.1 引言

本书共有三章介绍 Net/3的插口层代码，本章是第一章。插口概念最早出现于 1983年的4.2BSD版本中，它的主要目的是提供一个统一的访问网络和进程间通信协议的接口。这里讨论的Net/3版基于4.3BSD Reno版，该版本与大多数Unix供应商使用的早期的4.2版有些细小的差别。

如第1.7节所介绍的，插口层的主要功能是将进程发送的与协议有关的请求映射到产生插口时指定的与协议有关的实现。

为了允许标准的Unix I/O系统调用，如`read`和`write`，也能读写网络连接，在BSD版本中将文件系统和网络功能集成在系统调用级。与通过一个描述符访问一个打开的文件一样，进程也是通过一个描述符(一个小整数)来访问插口上的网络连接。这个特点使得标准的文件系统调用，如`read`和`write`，以及与网络有关的系统调用，如`sendmsg`和`recvmsg`，都能通过描述符来处理插口。

我们的重点是插口及相关的系统调用的实现而不是讨论如何使用插口层来实现网络应用。关于进程级的插口接口和如何编写网络应用的详细讨论，请参考 [Stevens 1990]和[Rago 1990]。

图15-1说明了进程中的插口接口与内核中的协议实现之间的层次关系。

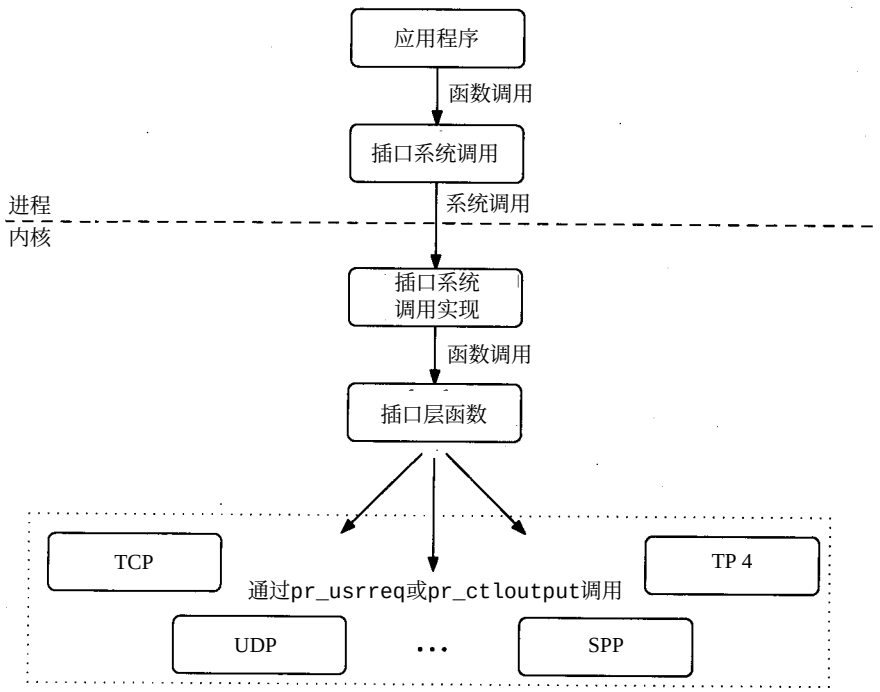


图15-1 插口层将一般的请求转换为指定的协议操作

splnet处理

插口包含很多对splnet和splx的成对调用。正如第1.12节中介绍的，这些调用保护访问在插口层和协议处理层间共享的数据结构的代码。如果不使用splnet，初始化协议处理和改变共享的数据结构的软件中断将使得插口层代码恢复执行时出现混乱。

我们假定读者理解了这些调用，因而在以后讨论中一般不再特别说明它们。

15.2 代码介绍

本章讨论涉及的三个文件在图15-2中列出。

文 件	描 述
sys/socketvar.h	socket结构定义
kern/uipc_syscalls.c kern/uipc_socket.c	系统调用实现 插口层函数

图15-2 本章讨论涉及的源文件

全局变量

本章讨论涉及到的两个全局变量如图15-3所示。

变 量	数据类型	描 述
socketops	struct fileops	I/O系统调用的socket实现
sysent	struct sysent[]	系统调用入口数组

图15-3 本章介绍的全局变量

15.3 socket结构

插口代表一条通信链路的一端，存储或指向与链路有关的所有信息。这些信息包括：使用的协议、协议的状态信息(包括源和目的地址)、到达的连接队列、数据缓存和可选标志。图15-5中给出了插口和与插口相关的缓存的定义。

41-42 so_type由产生插口的进程来指定，它指明插口和相关协议支持的通信语义。so_type的值等于图7-8所示的pr_type。对于UDP，so_type等于SOCK_DGRAM，而对于TCP，so_type则等于SOCK_STREAM。

43 so_options是一组改变插口行为的标志。图15-4列出了这些标志。

> 通过getsockopt和setsockopt系统调用进程能修改除SO_ACCEPTCONN外所有的插口选项。当在插口上发送listen系统调用时，SO_ACCEPTCONN被内核设置。

44 so_linger等于当关闭一条连接时插口继续发送数据的时间间隔 (单位为一个时钟滴答)(第15.15节)。

45 so_state表示插口的内部状态和一些其他的特点。图15-6列出了so_state可能的取值。

so_options	仅用于内核	描　述
SO_ACCEPTCONN	•	插口接受进入的连接
SO_BROADCAST		插口能够发送广播报文
SO_DEBUG		插口记录排错信息
SO_DONTROUTE		输出操作旁路选路表
SO_KEEPALIVE		插口查询空闲的连接
SO_OOBINLINE		插口将带外数据同正常数据存放在一起
SO_REUSEADDR		插口能重新使用一个本地地址
SO_REUSEPORT		插口能重新使用一个本地地址和端口
SO_USELOOPBACK		仅针对选路域插口；发送进程收到它自己的选路请求

图15-4 so_options 的值

socketvar.h

```
41 struct socket {
42     short   so_type;            /* generic type, Figure 7.8 */
43     short   so_options;         /* from socket call, Figure 15.5 */
44     short   so_linger;          /* time to linger while closing */
45     short   so_state;           /* internal state flags, Figure 15.6 */
46     caddr_t so_pcb;             /* protocol control block */
47     struct protosw *so_proto;   /* protocol handle */
48 /*
49  * Variables for connection queueing.
50  * Socket where accepts occur is so_head in all subsidiary sockets.
51  * If so_head is 0, socket is not related to an accept.
52  * For head socket so_q0 queues partially completed connections,
53  * while so_q is a queue of connections ready to be accepted.
54  * If a connection is aborted and it has so_head set, then
55  * it has to be pulled out of either so_q0 or so_q.
56  * We allow connections to queue up based on current queue lengths
57  * and limit on number of queued connections for this socket.
58  */
59     struct socket *so_head;     /* back pointer to accept socket */
60     struct socket *so_q0;       /* queue of partial connections */
61     struct socket *so_q;        /* queue of incoming connections */
62     short   so_q0len;           /* partials on so_q0 */
63     short   so_qlen;            /* number of connections on so_q */
64     short   so_qlimit;          /* max number queued connections */
65     short   so_timeo;           /* connection timeout */
66     u_short so_error;           /* error affecting connection */
67     pid_t   so_pgid;            /* pgid for signals */
68     u_long  so_oobmark;         /* chars to oob mark */
69 /*
70  * Variables for socket buffering.
71  */
72     struct sockbuf {
73         u_long  sb_cc;          /* actual chars in buffer */
74         u_long  sb_hiwat;       /* max actual char count */
75         u_long  sb_mbcnt;       /* chars of mbufs used */
76         u_long  sb_mbmax;       /* max chars of mbufs to use */
77         long    sb_lowat;       /* low water mark */
78         struct mbuf *sb_mb;     /* the mbuf chain */
79         struct selinfo sb_sel;  /* process selecting read/write */
80         short   sb_flags;       /* Figure 16.5 */
81         short   sb_timeo;       /* timeout for read/write */
82     } so_rcv, so_snd;
83     caddr_t so_tpcb;            /* Wisc. protocol control block XXX */
```

图15-5 struct socket定义

```
84      void    (*so_upcall) (struct socket * so, caddr_t arg, int waitf);
85      caddr_t so_upcallarg;       /* Arg for above */
86 };
```

socketvar.h

图15-5 (续)

so_state	仅用于内核	描 述
SS_ASYNC		插口应该I/O事件的异步通知
SS_NBIO		插口操作不能阻塞进程
SS_CANTRCVMORE	•	插口不能再从对方接收数据
SS_CANTSENDMORE	•	插口不能再发送数据给对方
SS_ISCONFIRMING	•	插口正在协商一个连接请求
SS_ISCONNECTED	•	插口被连接到外部插口
SS_ISCONNECTING	•	插口正在连接一个外部插口
SS_ISDISCONNECTING	•	插口正在同对方断连
SS_NOFDREF	•	插口没有同任何描述符相连
SS_PRIV	•	插口由拥有超级用户权限的进程所产生
SS_RCVATMARK	•	在最近的带外数据到达之前，插口已处理完所有收到的数据

图15-6 so_state 的值

从图15-6的第二列中可以看出，进程可以通过 fcntl和ioctl系统调用直接修改SS_ASYNC和SS_NBIO。对于其他的标志，进程只能在系统调用的执行过程中间接修改。例如，如果进程调用connect，当连接被建立时，SS_ISCONNECTED标志就会被内核设置。

SS_NBIO和SS_ASYNC标志

在默认情况下，进程在发出I/O请求后会等待资源。例如，对一个插口发 read系统调用，如果当前没有网络上来的数据，则 read系统调用就会被阻塞。同样，当一个进程调用 write系统调用时，如果内核中没有缓存来存储发送的数据，则内核将阻塞进程。如果设置了SS_NBIO，在对插口执行I/O操作且请求的资源不能得到时，内核并不阻塞进程，而是返回EWOULDBLOCK。

如果设置了 SS_ASYNC，当因为下列情况之一而使插口状态发生变化时，内核发送SIGIO信号给so_pgid标识的进程或进程组：

- 连接请求已完成；
- 断连请求已被启动；
- 断连请求已完成；
- 连接的一个通道已被关闭；
- 插口上有数据到达；
- 数据已被发送(即，输出缓存中有闲置空间)；或
- UDP或TCP插口上出现了一个异步差错。

46 so_pcb指向协议控制块，协议控制块包含与协议有关的状态信息和插口参数。每一种协议都定义了自己的控制块结构，所以 so_pcb被定义成一个通用的指针。图 15-7列出了我们讨论的控制块结构。

so_pcb从来不直接指向tcpcb结构；参考图22-1。

协　议	控 制 块	参考章节
UDP	struct inpcb	第22.3节
TCP	struct inpcb struct tcpcb	第22.3节 第24.5节
ICMP、IGMP和原始IP	struct inpcb	第22.3节
路由	struct rawcb	第20.3节

图15-7　协议控制块

47　so_proto指向进程在socket系统调用(第7.4节)中选择的协议的protosw结构。

48-64　设置了SO_ACCEPTCONN标志的插口维护两个连接队列。还没有完全建立的连接 (如TCP的三次握手还没完成)被放在队列so_q0中。已经建立的连接或将被接受的连接 (例如，TCP的三次握手已完成)被放入队列so_q中。队列的长度分别为so_q0len和so_qlen。每一个被排队的连接由它自己的插口来表示。在每一个被排队的插口中，so_head指向设置了SO_ACCEPTCONN的源插口。

插口上可排队的连接数通过so_qlimit来控制，进程可以通过listen系统调用来设置so_qlimit。内核隐含设置的上限为5 (SOMAXCONN，图15-24)和下限为0。图15-29中显示的有点晦涩的公式使用so_qlimit来控制排队的连接数。

图15-8说明了有三个连接将被接受、一个连接已被建立的情况下的队列内容。

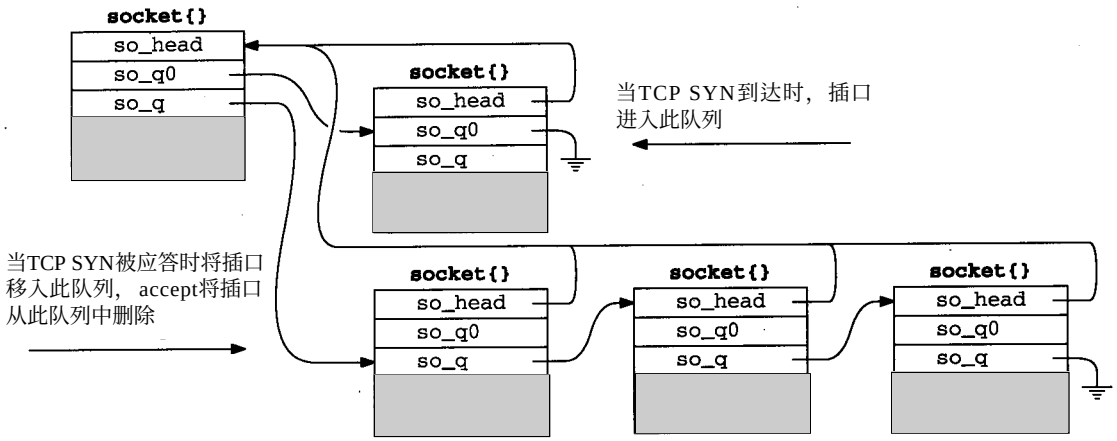


图15-8　插口连接队列

65　so_timeo用作accept、connet和close处理期间的等待通道(*wait channel*，第15.10节)。

66　so_error保存差错代码，直到在引用该插口的下一个系统调用期间差错代码能送给进程。

67　如果插口的SS_ASYNC被设置，则SIGIO信号被发送给进程(如果so_pgid大于0)或进程组(如果so_pgid小于0)。可以通过ioctl的SIOCSPGRP和SIOCGPGRP命令来修改或检查so_pgid的值。关于进程组的更详细信息请参考 [Stevens 1992]。

68　so_oobmark标识在输入数据流中最近收到的带外数据的开始点。第16.11节将讨论插口对带外数据的支持，第29.7节将讨论TCP中的带外数据的语义。

69-82 每一个插口包括两个数据缓存，so_rcv和so_snd，分别用来缓存接收或发送的数据。so_rcv和so_snd是包含在插口结构中的结构而不是指向结构的指针。我们将在第16章中描述插口缓存的结构和使用。

83-86 在Net/3中不使用so_tpcb。so_upcall和so_upcallarg也仅用于Net/3中的NFS软件。

NFS与通常的软件不太一样。在很大程度上它是一个进程级的应用但却在内核中运行。当数据到达接收缓存时，通过so_upcall来触发NFS的输入处理。在这种情况下，tsleep和wakeup机制是不合适的，因为NFS协议是在内核中运行而不是作为一个普通进程。

文件socketvar.h和uipc_socket2.c定义了几个简化插口层代码的宏和函数。图15-9对它们进行了描述。

名 称	描 述
sosendallatonce	*so*中指定的协议要求每一个发送系统调用产生一个协议请求吗？ int **sosendallatonce**(struct socket *so);
soisconnecting	将插口状态设置为SO_ISCONNECTING int **soisconnecting**(struct socket *so);
soisconnected	参考图15-30
soreadable	插口*so*上的读调用不阻塞就返回信息吗？ int **soreadable**(struct socket *so);
sowriteable	插口*so*上的写调用不阻塞就返回吗？ int **sowriteable**(struct socket *so);
socantsendmore	设置插口标志SO_CANTSENDMORE。唤醒所有等待在发送缓存上的进程 int **socantsendmore**(struct socket *so);
socantrcvmore	设置插口标志SO_CANTRCVMORE。唤醒所有等待在接收缓存上的进程 int **socantrcvmore**(struct socket *so);
soisdisconnecting	清除SS_ISCONNECTING标志。设置SS_ISDISCONG、SS_CANTRCVMORE和SS_CANTSENDMORE标志。唤醒所有等待在插口上的进程 int **soisdisconnecting**(struct socket *so);
soisdisconnected	清除SS_ISCONNECTING、SS_ISCONNECTED和SS_ISDISCONNECTING标志。设置SS_CANTRCVMORE和SS_CANTSENDMORE标志。唤醒所有等待在插口上的进程或等待close完成的进程 int **soisdisconnected**(struct socket *so);
soqinsque	将so插入*head*指向的队列中。如果*q*等于0，插口被插到存放未完成的连接的so_q0队列的后面。否则，插口被插到存放准备接受的连接的队列so_q的后面。Net/1错误地将插口插到队列的前面 int **soqinsque**(struct socket * *head*, struct socket * *so*, int *q*);
soqremque	从队列*q*中删除*so*。通过*so->so_head*来定位插口队列 int **soqremque**(struct socket *so, int *q*);

图15-9 插口的宏和函数

15.4 系统调用

进程同内核交互是通过一组定义好的函数来进行的，这些函数称为系统调用。在讨论支持网络的系统调用之前，我们先来看看系统调用机制的本身。

从进程到内核中的受保护的环境的转换是与机器和实现相关的。在下面的讨论中，我们使用Net/3在386上的实现来说明如何实现有关的操作。

在BSD内核中，每一个系统调用均被编号，当进程执行一个系统调用时，硬件被配置成仅传送控制给一个内核函数。将标识系统调用的整数作为参数传给该内核函数。在386实现中，这个内核函数为syscall。利用系统调用的编号，syscall在表中找到请求的系统调用的sysent结构。表中的每一个单元均为一个sysent结构。

```
struct sysent {
      int sy_narg;        /* number of arguments */
      int (*sy_call) ();  /* implementing function */
};                        /* system call table entry */
```

表中有几个项是从sysent数组中来的，该数组是在kern/init_sysent.c中定义的。

```
struct sysent sysent[] = {
       /*...*/
       { 3, recvmsg },                /* 27 = recvmsg */
       { 3, sendmsg },                /* 28 = sendmsg */
       { 6, recvfrom },               /* 29 = recvfrom */
       {3, accept },                  /* 30 = accept */
       {3, getpeername },             /* 31=getpeername */
       {3, getsockname },             /* 32 = getsockname */
       /* ...*/
}
```

例如，recvmsg系统调用在系统调用表中的第27个项，它有两个参数，利用内核中的recvmsg函数实现。

syscall将参数从调用进程复制到内核中，并且分配一个数组来保存系统调用的结果。然后，当系统调用执行完成后，syscall将结果返回给进程。syscall将控制交给与系统调用相对应的内核函数。在386实现中，调用有点像：

```
struct sysent *callp;
error = (*callp->sy_call) (p, args, rval);
```

这里指针callp指向相关的sysent结构；指针p则指向调用系统调用的进程的进程表项；args作为参数传给系统调用，它是一个32 bit长的字数组；而rval则是一个用来保存系统调用的返回结果的数组，数组有两个元素，每个元素是一个32 bit长的字。当我们用“系统调用”这个词时，我们指的是被syscall调用的内核中的函数，而不是应用调用的进程中的函数。

syscall期望系统调用函数(即sy_call指向的函数)在没有差错时返回0，否则返回非0的差错代码。如果没有差错出现，内核将rval中的值作为系统调用(应用调用的)的返回值传送给进程。如果有差错，syscall忽略rval中的值，并以与机器相关的方式返回差错代码给进程，使得进程能从外部变量errno中得到差错代码。应用调用的函数则返回-1或一个空指针表示应用应该查看errno获得差错信息。

在386上的实现，设置进位比特(carry bit)来表示syscall的返回值是一个差错代码。进程中的系统调用残桩将差错代码赋给errno，并返回-1或空指针给应用。如果没有设置进位

比特，则将syscall返回的值返回给进程中的系统调用的残桩。

总之，实现系统调用的函数“返回”两个值：一个给 syscall函数；在没有差错的情况下，syscall将另一个(在rval中)返回给调用进程。

15.4.1 举例

socket系统调用的原型是：

```
int socket(int domain, int type, int protocol);
```

实现socket系统调用的内核函数的原型是：

```
struct socket_args {
    int domain;
    int type;
    int protocol;
};
socket(struct proc *p, struct socket_args *uap, int *retval);
```

当一个应用调用socket时，进程用系统调用机制将三个独立的整数传给内核。 syscall将参数复制到32bit值的数组中，并将数组指针作为第二个参数传给 socket的内核版。内核版的socket将第二个参数作为指向socket_args结构的指针。图15-10显示了上述过程。

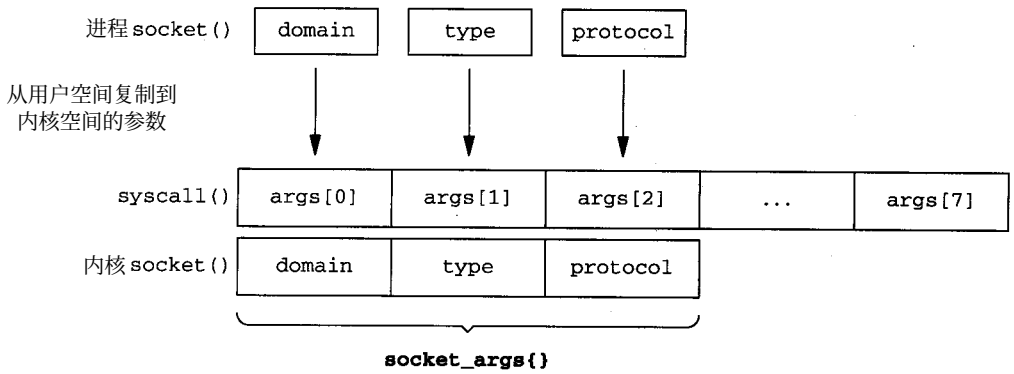


图15-10 socket 参数处理

同socket类似，每一个实现系统调用的内核函数将 args说明成一个与系统调用有关的结构指针，而不是一个指向32 bit的字的数组的指针。

当原型无效时，隐式的类型转换仅在传统的 K&R C中或ANSI C中是合法的。如果原型是有效的，则编译器将产生一个警告。

syscall在执行内核系统调用函数之前将返回值置为0。如果没有差错出现，系统调用函数直接返回而不需清除 *retval，syscall返回0给进程。

15.4.2 系统调用小结

图15-11对与网络有关的系统调用进行了小结。

我们将在本章中讨论建立、服务器、客户和终止类系统调用。输入、输出类系统调用将在第16章中介绍，管理类系统调用将在第17章中介绍。

类　别	名　称	功　能
建　立	socket	在指明的通信域内产生一个未命名的插口
	bind	分配一个本地地址给插口
服务器	listen	使插口准备接收连接请求
	accept	等待并接受连接
客　户	connect	同外部插口建立连接
输　入	read	接收数据到一个缓存中
	readv	接收数据到多个缓存中
	recv	指明选项接收数据
	recvfrom	接收数据和发送者的地址
	recvmsg	接收数据到多个缓存中，接收控制信息和发送者地址；指明接收选项
输　出	write	发送一个缓存中的数据
	writev	发送多个缓存中的数据
	send	指明选项发送数据
	sendto	发送数据到指明的地址
	sendmsg	从多个缓存发送数据和控制信息到指明的地址；指明发送选项
I/O	select	等待I/O事件
终　止	shutdown	终止一个或两个方向上的连接
	close	终止连接并释放插口
管　理	fcntl	修改I/O语义
	ioctl	各类插口操作
	setsockopt	设置插口或协议选项
	getsockopt	得到插口或协议选项
	getsockname	得到分配给插口的本地地址
	getpeername	得到分配给插口的远端地址

图15-11　Net/3中的网络系统调用

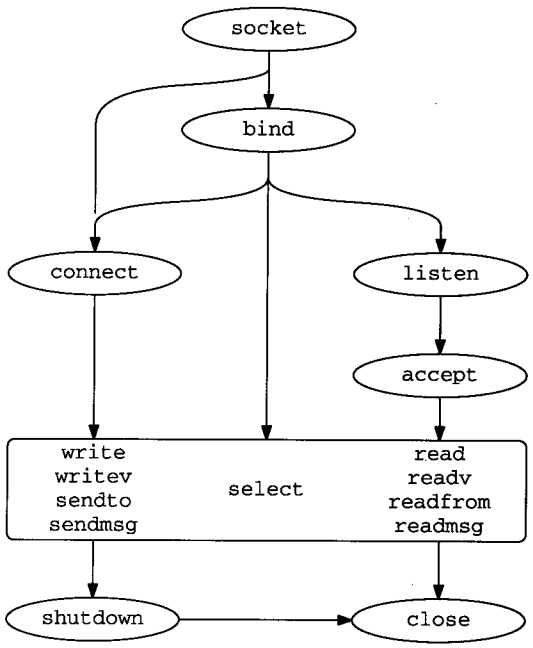


图15-12　网络系统调用流程图

图15-12画出了应用使用这些系统调用的顺序。大方块中的I/O系统调用可以在任何时候调用。该图不是一个完整的状态流程图，因为一些正确的转换在本图中没有画出；仅显示了一些常见的转换。

15.5 进程、描述符和插口

在描述插口系统调用之前，我们需要介绍将进程、描述符和插口联系在一起的数据结构。图15-13给出了这些结构以及与我们的讨论有关的结构成员。关于文件结构的更复杂的解释请参考[Leffer ea al. 1989]。

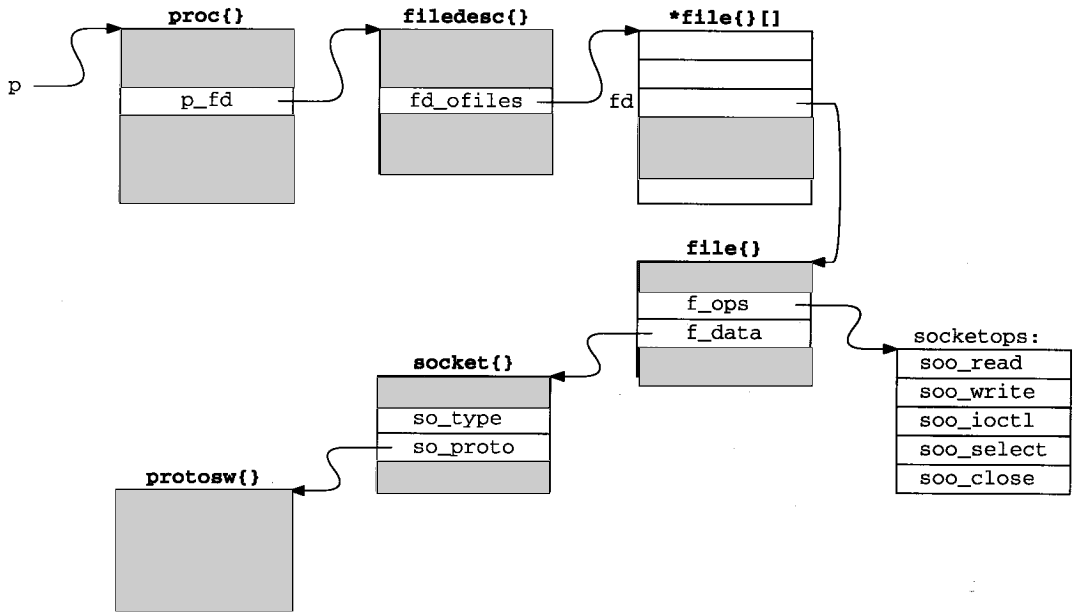


图15-13 进程、文件和插口结构

实现系统调用的函数的第一个参数总为p，即指向调用进程的proc结构的指针。内核利用proc结构记录进程的有关信息。在proc结构中，p_fd指向filedesc结构，该结构的主要功能是管理fd_ofiles指向的描述符表。描述符表的大小是动态变化的，由一个指向file结构的指针数组组成。每一个file结构描述一个打开的文件，该结构可被多个进程共享。

图15-13仅显示了一个file结构。通过p->p_fd->fd_ofiles[fd]访问到该结构。在file结构中，有两个结构成员是我们感兴趣的：f_ops和f_data。I/O系统调用(如read和write)的实现因描述符中的I/O对象类型的不同而不同。f_ops指向fileops结构，该结构包含一张实现read、write、ioctl、select和close系统调用的函数指针表。图15-13显示f_ops指向一个全局的fileops结构，即socketops，该结构包含指向插口用的函数的指针。

f_data指向相关I/O对象的专用数据。对于插口而言，f_data指向与描述符相关的socket结构。最后，socket结构中的so_proto指向产生插口时选中的协议的protosw结构。回想一下，每一个protosw结构是由与该协议关联的所有插口共享的。

下面我们开始讨论系统调用。

15.6 socket系统调用

socket系统调用产生一个新的插口，并将插口同进程在参数 domain、type和protocol中指定的协议联系起来。该函数 (如图15-14所示)分配一个新的描述符，用来在后续的系统调用中标识插口，并将描述符返回给进程。

uipc_syscalls.c

```
42 struct socket_args {
43     int     domain;
44     int     type;
45     int     protocol;
46 };

47 socket(p, uap, retval)
48 struct proc *p;
49 struct socket_args *uap;
50 int    *retval;
51 {
52     struct filedesc *fdp = p->p_fd;
53     struct socket *so;
54     struct file *fp;
55     int     fd, error;

56     if (error = falloc(p, &fp, &fd))
57         return (error);
58     fp->f_flag = FREAD | FWRITE;
59     fp->f_type = DTYPE_SOCKET;
60     fp->f_ops = &socketops;
61     if (error = socreate(uap->domain, &so, uap->type, uap->protocol)) {
62         fdp->fd_ofiles[fd] = 0;
63         ffree(fp);
64     } else {
65         fp->f_data = (caddr_t) so;
66         *retval = fd;
67     }
68     return (error);
69 }
```

uipc_syscalls.c

图15-14 socket 系统调用

42-55 在每一个系统调用的前面，都定义了一个描述进程传递给内核的参数的结构。在这种情况下，参数是通过socket_args传入的。所有插口层系统调用都有三个参数：p，指向调用进程的proc结构；uap，指向包含进程传送给系统调用的参数的结构；retval，用来接收系统调用的返回值。在通常情况下，忽略参数 p和retval，引用uap所指的结构中的内容。

56-60 falloc分配一个新的file结构和fd_ofiles数组(图15-13)中的一个元素。fp指向新分配的结构，fd则为结构在数组fd_ofiles中的索引。socket将file结构设置成可读、可写，并且作为一个插口。将所有插口共享的全局 fileops结构socketops连接到f_ops指向的file结构中。socketops变量在编译时被初始化，如图15-15所示。

成 员	值
fo_read	*soo_read*
fo_write	*soo_write*
fo_ioctl	*soo_ioctl*
fo_select	*soo_select*
fo_close	*soo_close*

图15-15 socketops ：插口用全局fileops 结构

60-69 socreate分配并初始化一个 socket结构。如果socreate执行失败，将差错代码赋给 error，释放file结

构，清除存放描述符的数组元素。如果 socreate执行成功，将f_data指向socket结构，建立插口和描述符之间的联系。通过 *retval将fd返回给进程。 socket返回0或返回由socreate返回的差错代码。

15.6.1 socreate函数

大多数插口系统调用至少被分成两个函数，与 socket和socreate类似。第一个函数从进程那里获取需要的数据，调用第二个函数 so*xxx*来完成功能处理，然后返回结果给进程。这种分成多个函数的做法是为了第二个函数能直接被基于内核的网络协议调用，如 NFS。socreate的代码如图15-16所示。

uipc_socket.c

```
43 socreate(dom, aso, type, proto)
44 int     dom;
45 struct socket **aso;
46 int     type;
47 int     proto;
48 {
49     struct proc *p = curproc;   /* XXX */
50     struct protosw *prp;
51     struct socket *so;
52     int     error;

53     if (proto)
54         prp = pffindproto(dom, proto, type);
55     else
56         prp = pffindtype(dom, type);
57     if (prp == 0 || prp->pr_usrreq == 0)
58         return (EPROTONOSUPPORT);
59     if (prp->pr_type != type)
60         return (EPROTOTYPE);
61     MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_WAIT);
62     bzero((caddr_t) so, sizeof(*so));
63     so->so_type = type;
64     if (p->p_ucred->cr_uid == 0)
65         so->so_state = SS_PRIV;
66     so->so_proto = prp;
67     error =
68         (*prp->pr_usrreq) (so, PRU_ATTACH,
69             (struct mbuf *) 0, (struct mbuf *) proto, (struct mbuf *) 0);
70     if (error) {
71         so->so_state |= SS_NOFDREF;
72         sofree(so);
73         return (error);
74     }
75     *aso = so;
76     return (0);
77 }
```

uipc_socket.c

图15-16 socreate 函数

43-52 socreate共有四个参数：dom，请求的协议域(如，PF_INET)；aso，保存指向一个新的socket结构的指针；type，请求的插口类型(如，SOCK_STREAM)；proto，请求的协议。

1. 发现协议交换表

53-60 如果proto等于非0值，pffindproto查找进程请求的协议。如果proto等于0，pffindtype用由type指定的语义在指定域中查找一种协议。这两个函数均返回一个指向匹配协议的protosw结构的指针或空指针(参考第7.6节)。

2. 分配并初始化socket结构

61-66 socreate分配一个新的socket结构，并将结构内容全清成0，记录下type。如果调用进程有超级用户权限，则设置插口结构中的SS_PRIV标志。

3. PRU_ATTACH请求

67-69 在与协议无关的插口层中发送与协议有关的请求的第一个例子出现在socreate中。回想在第7.4节和图15-13中，so->so_proto->pr_usrreq是一个指向与插口so相关联的协议的用户请求函数指针。每一个协议均提供了一个这样的函数来处理从插口层来的通信请求。函数原型是：

```
int pr_usrreq(struct socket *so, int req, struct mbuf *mo, *m1, *m2);
```

第一个参数是一个指向相关插口的指针，*req*是一个标识请求的常数。后三个参数(*m*0，*m*1，*m*2)因请求不同而异。它们总是被作为一个mbuf结构指针传递，即使它们是其他的类型。在必要的时候，进行类似转换以避免编译器的警告。

图15-17列出了pr_usrreq函数提供的通信请求。每一个请求的语义起决于服务请求的协议。

请　求	参　数			描　述
	*m*0	*m*1	*m*2	
PRU_ABORT				异常终止每一个存在的连接
PRU_ACCEPT		*address*		等待并接受连接
PRU_ATTACH		*protocol*		产生了一个新的插口
PRU_BIND		*address*		绑定地址到插口
PRU_CONNECT		*address*		同地址建立关联或连接
PRU_CONNECT2		*socket2*		将两个插口连在一起
PRU_DETACH				插口被关闭
PRU_DISCONNECT				切断插口和另一地址间的关联
PRU_LISTEN				开始监听连接请求
PRU_PEERADDR		*buffer*		返回与插口关联的对方地址
PRU_RCVD		*flags*		进程已收到一些数据
PRU_RCVOOB	*buffer*	*flags*		接收OOB数据
PRU_SEND	*data*	*address*	*control*	发送正常数据
PRU_SENDOOB	*data*	*address*	*control*	发送OOB数据
PRU_SHUTDOWN				结束同另一地址的通信
PRU_SOCKADDR		*buffer*		返回与插口相关联的本地地址

图15-17 pr_usrreq 函数

PRU_CONNECT2请求只用于Unix域，它的功能是将两个本地插口连接起来。Unix的管道(pipe)就是通过这种方式来实现的。

4. 退出处理

70-77 回到socreate，函数将协议交换表连接到插口，发送PRU_ATTACH请求通知协议

已建立一个新的连接端点。该请求引起大多数协议，如 TCP和UDP，分配并初始化所有支持新的连接端点的数据结构。

15.6.2 超级用户特权

图15-18列出了要求超级用户权限的网络操作。

函 数	超级用户		描 述	参考图
	进程	插口		
in_control		•	分配接口地址、网络掩码、目的地址	图6-14
in_control		•	分配广播地址	图6-22
in_pcbbind	•		绑定到一个小于1024的Internet端口	图22-22
ifioctl	•		改变接口配置	图4-29
ifioctl	•		配置多播地址(见下面的说明)	图12-11
rip_usrreq	•		产生一个ICMP、IGMP或原始 IP插口	图32-10
slopen	•		将一个SLIP设备与一个tty设备联系起来	图5-9

图15-18 Net/3中的超级用户特权

当多播ioctl命令(SIOCADDMULTI和SIOCDELMULTI)是被IP_ADD_MEMBERSHIP和IP_DROP_MEMBERSHIP插口选项间接激活时，它可以被非超级用户访问。

在图15-18中，“进程”栏表示请求必须由超级用户进程来发起，“插口”栏表示请求必须是针对由超级用户产生的插口(也就是说，进程不需要超级用户权限，而只需有访问插口的权限，习题15.1)。在Net/3中，suser函数用来判断调用进程是否有超级用户权限，通过SS_PRIV标志来判断一个插口是否由超级用户进程产生。

因为rip_usrreq在用socreate产生插口后立即检查SS_PRIV标志，所以我们认为只有超级用户进程才能访问这个函数。

15.7 getsock和sockargs函数

这两个函数重复出现在插口系统调用中。getsock的功能是将描述符映射到一个文件表项中，sockargs将进程传入的参数复制到内核中的一个新分配的 mbuf中。这两个函数都要检查参数的正确性，如果参数不合法，则返回相应的非0差错代码。

图15-19列出了getsock函数的代码。

754-767 getsock函数利用fdp查找描述符fdes指定的文件表项，fdp是指向filedesc结构的指针。getsock将打开的文件结构指针赋给 fpp，并返回，或者当出现下列情况时返回差错代码：描述符的值超过了范围而不是指向一个打开的文件；描述符没有同插口建立联系。

图15-20列出了sockargs函数的代码。

768-783 如图15-4中所描述的，sockargs将进程传给系统调用的参数的指针从进程复制到内核而不是复制指针指向的数据，这样做是因为每一个参数的语义只有相对应的系统调用才知道，而不是针对所有的系统调用。多个系统调用在调用 sockargs复制参数指针后，将指针指向的数据从进程复制到内核中新分配的 mbuf中。例如，sockargs将bind的第二个参

数指向的本地插口地址从进程复制到一个mbuf中。

uipc_syscalls.c

```
754 getsock(fdp, fdes, fpp)
755 struct filedesc *fdp;
756 int     fdes;
757 struct file **fpp;
758 {
759     struct file *fp;

760     if ((unsigned) fdes >= fdp->fd_nfiles ||
761         (fp = fdp->fd_ofiles[fdes]) == NULL)
762         return (EBADF);
763     if (fp->f_type != DTYPE_SOCKET)
764         return (ENOTSOCK);
765     *fpp = fp;
766     return (0);
767 }
```

uipc_syscalls.c

图15-19 getsock 函数

uipc_syscalls.c

```
768 sockargs(mp, buf, buflen, type)
769 struct mbuf **mp;
770 caddr_t buf;
771 int     buflen, type;
772 {
773     struct sockaddr *sa;
774     struct mbuf *m;
775     int     error;

776     if ((u_int) buflen > MLEN) {
777         return (EINVAL);
778     }
779     m = m_get(M_WAIT, type);
780     if (m == NULL)
781         return (ENOBUFS);
782     m->m_len = buflen;
783     error = copyin(buf, mtod(m, caddr_t), (u_int) buflen);
784     if (error)
785         (void) m_free(m);
786     else {
787         *mp = m;
788         if (type == MT_SONAME) {
789             sa = mtod(m, struct sockaddr *);
790             sa->sa_len = buflen;
791         }
792     }
793     return (error);
794 }
```

uipc_syscalls.c

图15-20 sockargs 函数

如果数据不能存入一个mbuf中或无法分配mbuf，则sockargs返回EINVAL或ENOBUFS。注意，这里使用的是标准的mbuf而不是分组首部的mbuf。copyin的功能是将数据从进程复制到mbuf中。copyin返回的最常见的差错是EACCES，它表示进程提供的地址不正确。

784-785 当出现差错时，丢弃mbuf，并返回差错代码。如果没有差错，通过mp返回指向mbuf的指针，sockargs返回0。

786-794 如果type等于MT_SONAME，则进程传入的是一个sockaddr结构。sockargs

将刚复制的参数的长度赋给内部长度变量 sa_len。这一点确保即使进程没有正确地初始化结构，结构内的大小也是正确的。

Net/3确实包含了一段代码来支持在 pre-4.3BSD Reno系统上编译的应用，这些应用的sockaddr结构中并没有sa_len字段，但是图15-20中没有显示这段代码。

15.8 bind系统调用

bind系统调用将一个本地的网络运输层地址和插口联系起来。一般来说，作为客户的进程并不关心它的本地地址是什么。在这种情况下，进程在进行通信之前没有必要调用 bind；内核会自动为其选择一个本地地址。

服务器进程则总是需要绑定到一个已知的地址上。所以，进程在接受连接 (TCP)或接收数据报(UDP)之前必须调用bind，因为客户进程需要同已知的地址建立连接或发送数据报到已知的地址。

插口的外部地址由connect指定或由允许指定外部地址的写调用 (sendto或sendmsg)指定。

图15-21列出了bind调用的代码。

uipc_syscalls.c

```
70 struct bind_args {
71     int     s;
72     caddr_t name;
73     int     namelen;
74 };

75 bind(p, uap, retval)
76 struct proc *p;
77 struct bind_args *uap;
78 int    *retval;
79 {
80     struct file *fp;
81     struct mbuf *nam;
82     int     error;

83     if (error = getsock(p->p_fd, uap->s, &fp))
84         return (error);
85     if (error = sockargs(&nam, uap->name, uap->namelen, MT_SONAME))
86         return (error);
87     error = sobind((struct socket *) fp->f_data, nam);
88     m_freem(nam);
89     return (error);
90 }
```

uipc_syscalls.c

图15-21 bind 函数

70-82 bind调用的参数有(在bind_args结构中)：s，插口描述符；name，包含传输地址(如，sockaddr_in结构)的缓存指针；和namelen，缓存大小。

83-90 getsock返回描述符的file结构，sockargs将本地地址复制到mbuf中，sobind将进程指定的地址同插口联系起来。在bind返回sobind的结果之前，释放保存地址的mbuf。

从技术上讲，一个描述符，如 s，标识一个同socket结构相关联的 file结构，而它本身并不是一个socket结构。将这种描述符看作插口是为了简化我们的讨论。

我们将在下面的讨论中经常看到这种模式：进程指定的参数被复制到 mbuf，必要时还要

进行处理，然后在系统调用返回之前释放 mbuf。虽然mbuf是为方便处理网络数据分组而设计的，但是将它们用作一般的动态内存分配机制也是有效的。

bind说明的另一种模式是：许多系统调用不使用 retval。在第15.4节中我们已提到过，在syscall将控制交给相应的系统调用之前总是将 retval清0。如果0不是合适的返回值，系统调用并不需要修改retval。

sobind函数

如图15-22所示，sobind是一个封装器，它给与插口相关联的协议发送 PRU_BIND请求。

uipc_socket.c

```
78 sobind(so, nam)
79 struct socket *so;
80 struct mbuf *nam;
81 {
82     int     s = splnet();
83     int     error;

84     error =
85         (*so->so_proto->pr_usrreq) (so, PRU_BIND,
86                                     (struct mbuf *) 0, nam, (struct mbuf *) 0);
87     splx(s);
88     return (error);
89 }
```

uipc_socket.c

图15-22 sobind 函数

78-89 sobind发送PRU_BIND请求。如果请求成功，将本地地址 nam同插口联系起来；否则，返回差错代码。

15.9 listen系统调用

listen系统调用的功能是通知协议进程准备接收插口上的连接请求，如图 15-23所示。它同时也指定插口上可以排队等待的连接数的门限值。超过门限值时，插口层将拒绝进入的连接请求排队等待。当这种情况出现时， TCP将忽略进入的连接请求。进程可以通过调用 accept (第15.11节)来得到队列中的连接。

uipc_syscalls.c

```
91 struct listen_args {
92     int     s;
93     int     backlog;
94 };

95 listen(p, uap, retval)
96 struct proc *p;
97 struct listen_args *uap;
98 int    *retval;
99 {
100    struct file *fp;
101    int     error;

102    if (error = getsock(p->p_fd, uap->s, &fp))
103        return (error);
104    return (solisten((struct socket *) fp->f_data, uap->backlog));
105 }
```

uipc_syscalls.c

图15-23 listen 系统调用

91-98 listen系统调用有两个参数：一个指定插口描述符；另一个指定连接队列门限值。
99-105 getsock返回描述符s的file结构，solisten将请求传递给协议层。

solisten函数

solisten函数发送PRU_LISTEN请求，并使插口准备接收连接，如图15-24所示。
90-109 在solisten发送PRU_LISTEN请求且pr_usrreq返回后，标识插口处于准备接收连接状态。如果当pr_usrreq返回时有连接正在连接队列中，则不设置SS_ACCEPTCONN标志。

计算存放进入连接的队列的最大值，并赋给 so_qlimit。Net/3默认设置下限为0，上限为5(SOMAXCONN)条连接。

uipc_socket.c

```
 90 solisten(so, backlog)
 91 struct socket *so;
 92 int     backlog;
 93 {
 94     int     s = splnet(), error;
 95     error =
 96         (*so->so_proto->pr_usrreq) (so, PRU_LISTEN,
 97                 (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0);
 98     if (error) {
 99         splx(s);
100         return (error);
101     }
102     if (so->so_q == 0)
103         so->so_options |= SO_ACCEPTCONN;
104     if (backlog < 0)
105         backlog = 0;
106     so->so_qlimit = min(backlog, SOMAXCONN);
107     splx(s);
108     return (0);
109 }
```

uipc_socket.c

图15-24 solisten 函数

15.10 tsleep和wakeup函数

当一个在内核中执行的进程因为得不到内核资源而不能继续执行时，它就调用 tsleep等待。tsleep的原型是：

```
int tsleep(caddr_t chan, int pri, char *mesg, int timeo);
```

tsleep的第一个参数*chan*，被称之为等待通道。它标志进程等待的特定资源或事件。许多进程能同时在同一个等待通道上睡眠。当资源可用或事件出现时，内核调用 wakeup，并将等待通道作为唯一的参数传入。wakeup的原型是：

```
void wakeup(caddr_t chan);
```

所有等待在该通道上的的进程均被唤醒，并被设置成运行状态。当每一个进程均恢复执行时，内核安排tsleep返回。

当进程被唤醒时，tsleep的第二个参数*pri*指定被唤醒进程的优先级。*pri*中还包括几个用于tsleep的可选的控制标志。通过设置 *pri*中的PCATCH标志，当一个信号出现时，tsleep也返回。*mesg*是一个说明调用tsleep的字符串，它将被放在调用报文或ps的输出中。

*timeo*设置睡眠间隔的上限值，其单位是时钟滴答。

图15-25列出了tsleep的返回值。

因为所有等待在同一等待通道上的进程均被wakeup唤醒，所以我们总是看到在一个循环中调用tsleep。每一个被唤醒的进程在继续执行之前必须检查等待的资源是否可得到，因为另一个被唤醒的进程可能已经先一步得到了资源。如果仍然得不到资源，进程再调用tsleep等待。

tsleep()	描　述
0	进程被一个匹配的wakeup唤醒
EWOULDBLOCK	进程在睡眠*timeo*个时钟滴答后，在匹配的wakeup调用之前被唤醒
ERESTART	在睡眠期间信号被进程处理，应重新启动挂起的系统调用
EINTR	在睡眠期间信号被进程处理，挂起的系统调用失败

图15-25　tsleep 的返回值

多个进程在一个插口上睡眠等待的情况是不多见的，所以，通常情况下，一次调用wakeup只有一个进程被内核唤醒。

关于睡眠和唤醒机制的详细讨论请参考 [Leffler et al. 1989]。

举例

多个进程在同一个等待通道上睡眠的一个例子是：让多个服务器进程读同一个 UDP插口。每一个服务器都调用recvfrom，并且只要没有数据可读就在tsleep中等待。当一个数据报到达插口时，插口层调用wakeup，所有等待进程均被放入运行队列。第一个运行的服务器读取了数据报而其他的服务器则再次调用tsleep。在这种情况下，不需要每一个数据报启动一个新的进程，就可将进入的数据报分发到多个服务器。这种技术同样可以用来处理 TCP的连接请求，只需让多个进程在同一个插口上调用accept。这种技术在[Comer and Stevens 1993]中描述。

15.11 accept系统调用

调用listen后，进程调用accept等待连接请求。accept返回一个新的描述符，指向一个连接到客户的新的插口。原来的插口 s仍然是未连接的，并准备接收下一个连接。如果name指向一个正确的缓存，accept就会返回对方的地址。

处理连接的细节由与插口相关联的协议来完成。对于TCP而言，当一条连接已经被建立(即，三次握手已经完成)时，就通知插口层。对于其他的协议，如 OSI的TP4，只要一个连接请求到达，tsleep就返回。当进程通过在插口上发送或接收数据来显式证实连接后，连接则算完成。

图15-26说明accept的实现。

uipc_syscalls.c

```
106 struct accept_args {
107     int     s;
108     caddr_t name;
109     int     *anamelen;
110 };

111 accept(p, uap, retval)
112 struct proc *p;
113 struct accept_args *uap;
114 int     *retval;
```

图15-26　accept 系统调用

```
115 {
116     struct file *fp;
117     struct mbuf *nam;
118     int     namelen, error, s;
119     struct socket *so;

120     if (uap->name && (error = copyin((caddr_t) uap->anamelen,
121                                      (caddr_t) & namelen, sizeof(namelen))))
122         return (error);
123     if (error = getsock(p->p_fd, uap->s, &fp))
124         return (error);
125     s = splnet();
126     so = (struct socket *) fp->f_data;
127     if ((so->so_options & SO_ACCEPTCONN) == 0) {
128         splx(s);
129         return (EINVAL);
130     }
131     if ((so->so_state & SS_NBIO) && so->so_qlen == 0) {
132         splx(s);
133         return (EWOULDBLOCK);
134     }
135     while (so->so_qlen == 0 && so->so_error == 0) {
136         if (so->so_state & SS_CANTRCVMORE) {
137             so->so_error = ECONNABORTED;
138             break;
139         }
140         if (error = tsleep((caddr_t) & so->so_timeo, PSOCK | PCATCH,
141                            netcon, 0)) {
142             splx(s);
143             return (error);
144         }
145     }
146     if (so->so_error) {
147         error = so->so_error;
148         so->so_error = 0;
149         splx(s);
150         return (error);
151     }
152     if (error = falloc(p, &fp, retval)) {
153         splx(s);
154         return (error);
155     }
156     { struct socket *aso = so->so_q;
157       if (soqremque(aso, 1) == 0)
158         panic("accept");
159       so = aso;
160     }

161     fp->f_type = DTYPE_SOCKET;
162     fp->f_flag = FREAD | FWRITE;
163     fp->f_ops = &socketops;
164     fp->f_data = (caddr_t) so;
165     nam = m_get(M_WAIT, MT_SONAME);
166     (void) soaccept(so, nam);
167     if (uap->name) {
168         if (namelen > nam->m_len)
169             namelen = nam->m_len;
170         /* SHOULD COPY OUT A CHAIN HERE */
171         if ((error = copyout(mtod(nam, caddr_t), (caddr_t) uap->name,
172                              (u_int) namelen)) == 0)
```

图15-26 (续)

```
173             error = copyout((caddr_t) & namelen,
174                             (caddr_t) uap->anamelen, sizeof(*uap->anamelen));
175     }
176     m_freem(nam);
177     splx(s);
178     return (error);
179 }
```

uipc_syscalls.c

图15-26 (续)

106-114 accept有三个参数：s为插口描述符；name为缓存指针，accept将把外部主机的运输地址填入该缓存；anamelen是一个保存缓存大小的指针。

1. 验证参数

116-134 accept将缓存大小(*anamelen)赋给namelen，getsock返回插口的file结构。如果插口还没有准备好接收连接 (即，还没有调用listen)，或已经请求了非阻塞的I/O，且没有连接被送入队列，则分别返回EINVAL或EWOULDBLOCK。

2. 等待连接

135-145 当出现下列情况时，while循环退出：有一条连接到达；出现差错；或插口不能再接收数据。当信号被捕获之后 (tsleep返回EINTR)，accept并不自动重新启动。当协议层通过sonewconn将一条连接插入队列后，唤醒进程。

在循环内，进程在tsleep中等待，当有连接到达时，tsleep返回0。如果tsleep被信号中断或插口被设置成非阻塞，则accept返回EINTR或EWOULDBLOCK(图15-25)。

3. 异步差错

146-151 如果进程在睡眠期间出现差错，则将插口中的差错代码赋给accept中的返回码，清除插口中的差错码后，accept返回。

异步事件改变插口状态是比较常见的。协议处理层通过设置so_error或唤醒在插口上等待的所有进程来通知插口层插口状态的改变。因为这一点，插口层必须在每次被唤醒后检查so_error，查看是否在进程睡眠期间有差错出现。

4. 将插口同描述符相关联

152-164 falloc为新的连接分配一个描述符；调用soqremque将插口从接收队列中删除，放到描述符的file结构中。习题15.4讨论调用panic。

5. 协议处理

167-179 accept分配一个新的mbuf来保存外部地址，并调用soaccept来完成协议处理。在连接处理期间产生的新的插口的分配和排队在第15.12中描述。如果进程提供了一个缓存来接收外部地址，copyout将地址和地址长度分别从nam和namelen中复制给进程。如果有必要，copyout还可能将地址截掉，如果进程提供的缓存不够大。最后，释放mbuf，使能协议处理，accept返回。

因为仅仅分配了一个mbuf来存放外部地址，运输地址必须能放入一个mbuf中。因为Unix域地址是文件系统中的路径名 (最长可达1023个字节)，所以要受到这个限制，但这对Internet域中的16字节长的sockaddr_in地址没有影响。第170行的注释说明可以通过分配和复制一个mbuf链的方式来去掉这个限制。

soaccept函数

soaccept函数通过协议层获得新的连接的客户地址，如图15-27所示。

uipc_socket.c

```
184 soaccept(so, nam)
185 struct socket *so;
186 struct mbuf *nam;
187 {
188     int     s = splnet();
189     int     error;

190     if ((so->so_state & SS_NOFDREF) == 0)
191         panic("soaccept: !NOFDREF");
192     so->so_state &= ~SS_NOFDREF;
193     error = (*so->so_proto->pr_usrreq) (so, PRU_ACCEPT,
194                                         (struct mbuf *) 0, nam, (struct mbuf *) 0);
195     splx(s);
196     return (error);
197 }
```

uipc_socket.c

图15-27 soaccept 函数

184-197 soaccept确保插口与一个描述符相连，并发送 PRU_ACCEPT请求给协议。pr_usrreq返回后，nam中包含外部插口的名字。

15.12 sonewconn和soisconnected函数

从图15-26中可以看出，accept等待协议层处理进入的连接请求，并且将它们放入 so_q 中。图15-28利用TCP来说明这个过程。

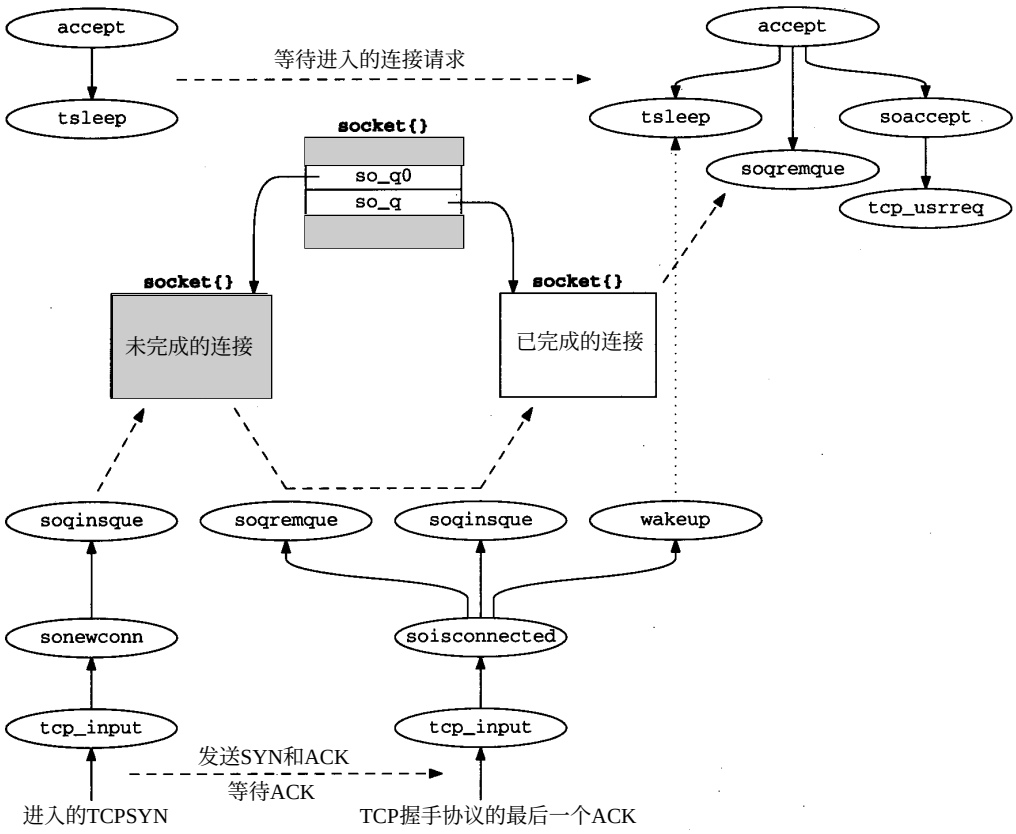


图15-28 处理进入的TCP连接

在图15-28的左上角，accept调用tsleep等待进入的连接。在左下角，tcp_input调用sonewconn为新的连接产生一个插口来处理进入的TCP SYN(图28-7)。sonewconn将产生的插口放入so_q0排队，因为三次握手还没有完成。

当TCP握手协议的最后一个ACK到达时，tcp_input调用soisconnected(图29-2)来更新产生的插口，并将它从so_q0中移到so_q中，唤醒所有调用accept等待进入的连接的进程。

图的右上角说明我们在图15-26中描述的函数。当tsleep返回时，accept从so_q中得到连接，发送PRU_ATTACH请求。插口同一个新的文件描述符建立了联系，accept也返回到调用进程。

图15-29显示了sonewconn函数。

uipc_socket2.c

```
123 struct socket *
124 sonewconn(head, connstatus)
125 struct socket *head;
126 int     connstatus;
127 {
128     struct socket *so;
129     int     soqueue = connstatus ? 1 : 0;
130     if (head->so_qlen + head->so_q0len > 3 * head->so_qlimit / 2)
131         return ((struct socket *) 0);
132     MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_DONTWAIT);
133     if (so == NULL)
134         return ((struct socket *) 0);
135     bzero((caddr_t) so, sizeof(*so));
136     so->so_type = head->so_type;
137     so->so_options = head->so_options & ~SO_ACCEPTCONN;
138     so->so_linger = head->so_linger;
139     so->so_state = head->so_state | SS_NOFDREF;
140     so->so_proto = head->so_proto;
141     so->so_timeo = head->so_timeo;
142     so->so_pgid = head->so_pgid;
143     (void) soreserve(so, head->so_snd.sb_hiwat, head->so_rcv.sb_hiwat);
144     soqinsque(head, so, soqueue);
145     if ((*so->so_proto->pr_usrreq) (so, PRU_ATTACH,
146                 (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0)) {
147         (void) soqremque(so, soqueue);
148         (void) free((caddr_t) so, M_SOCKET);
149         return ((struct socket *) 0);
150     }
151     if (connstatus) {
152         sorwakeup(head);
153         wakeup((caddr_t) & head->so_timeo);
154         so->so_state |= connstatus;
155     }
156     return (so);
157 }
```

uipc_socket2.c

图15-29 sonewconn 函数

123-129 协议层将head(指向正在接收连接的插口的指针)和connstatus(指示新连接的状态的标志)传给sonewconn。对于TCP而言，connstatus总是等于0。

对于TP4，connstatus总是等于SS_ISCONFIRMING。当一个进程开始从插口上接收或发送数据时隐式证实连接。

1. 限制进入的连接

130-131 当下面的不等式成立时，sonewconn不再接收任何连接：

$$so\_qlen + so\_q0len > \frac{3 \times so\_qlimit}{2}$$

这个不等式为一直没有完成的连接提供了一个令人费解的因子，且该不等式确保listen(fd, 0)允许一条连接。有关这个不等式的详细情况请参考卷1的图18-23。

2. 分配一个新的插口

132-143 一个新的socket结构被分配和初始化。如果进程对处理接收连接状态的插口调用了setsockopt，则新产生的socket继承好几个插口选项，因为so_options、so_linger、so_pgid和sb_hiwat的值被复制到新的socket结构中。

3. 排队连接

144 在第129行的代码中，根据connstatus的值设置soqueue。如果soqueue为0 (如，TCP连接)，则将新的插口插入到so_q0中；若connstatus等于非0值，则将其插入到so_q中(如，TP4连接)。

4. 协议处理

145-150 发送PRU_ATTACH请求，启动协议层对新的连接的处理。如果处理失败，则将插口从队列中删除并丢弃，然后sonewconn返回一个空指针。

5. 唤醒进程

151-157 如果connstatus等于非0值，所有在accept中睡眠或查询插口的可读性的进程均被唤醒。将connstatus对so_state执行或操作。TCP协议从来不会执行这段代码，因为对TCP而言，connstatus总是等于0。

某些将进入的连接首先插入so_q0队列中的协议在连接建立阶段完成时调用soisconnected，如TCP。对于TCP，当第二个SYN被应答时，就出现这种情况。

图15-30显示了soisconnected的代码。

kern/uipc_socket2.c

```
78 soisconnected(so)
79 struct socket *so;
80 {
81     struct socket *head = so->so_head;

82     so->so_state &= ~(SS_ISCONNECTING | SS_ISDISCONNECTING | SS_ISCONFIRMING);
83     so->so_state |= SS_ISCONNECTED;
84     if (head && soqremque(so, 0)) {
85         soqinsque(head, so, 1);
86         sorwakeup(head);
87         wakeup((caddr_t) & head->so_timeo);
88     } else {
89         wakeup((caddr_t) & so->so_timeo);
90         sorwakeup(so);
91         sowwakeup(so);
92     }
93 }
```

kern/uipc_socket2.c

图15-30 soisconnected 函数

6. 排队未完成的连接

78-87 通过修改插口的状态来表明连接已经完成。当对进入的连接调用soisconnected

(即，本地进程正在调用accept)时，head为非空。

如果soqremque返回1，就将插口放入so_q排队，sorwakeup唤醒通过调用select测试插口的可读性来监控插口上连接到达的进程。如果进程在accept中因等待连接而阻塞，则wakeup使得相应的tsleep返回。

7. 唤醒等待新连接的进程

88-93 如果head为空，就不需要调用soqremque，因为进程用connect系统调用初始化连接，且插口不在队列中。如果head非空，且soqremque返回0，则插口已经在so_q队列中。在某些协议中，如TP4，就出现这种情况，因为在TP4中，连接完成之前就已插入到so_q队列中。wakeup唤醒所有阻塞在connect中的进程，sorwakeup和sowwakeup负责唤醒那些调用select等待连接完成的进程。

15.13 connect系统调用

服务器进程调用listen和accept系统调用等待远程进程初始化连接。如果进程想自己初始化一条连接(即客户端)，则调用connect。

对于面向连接的协议如TCP，connect建立一条与指定的外部地址的连接。如果进程没有调用bind来绑定地址，则内核选择并且隐式地绑定一个地址到插口。

对于无连接协议如UDP或ICMP，connect记录外部地址，以便发送数据报时使用。任何以前的外部地址均被新的地址所代替。

图15-31显示了UDP或TCP调用connect时涉及到的函数。

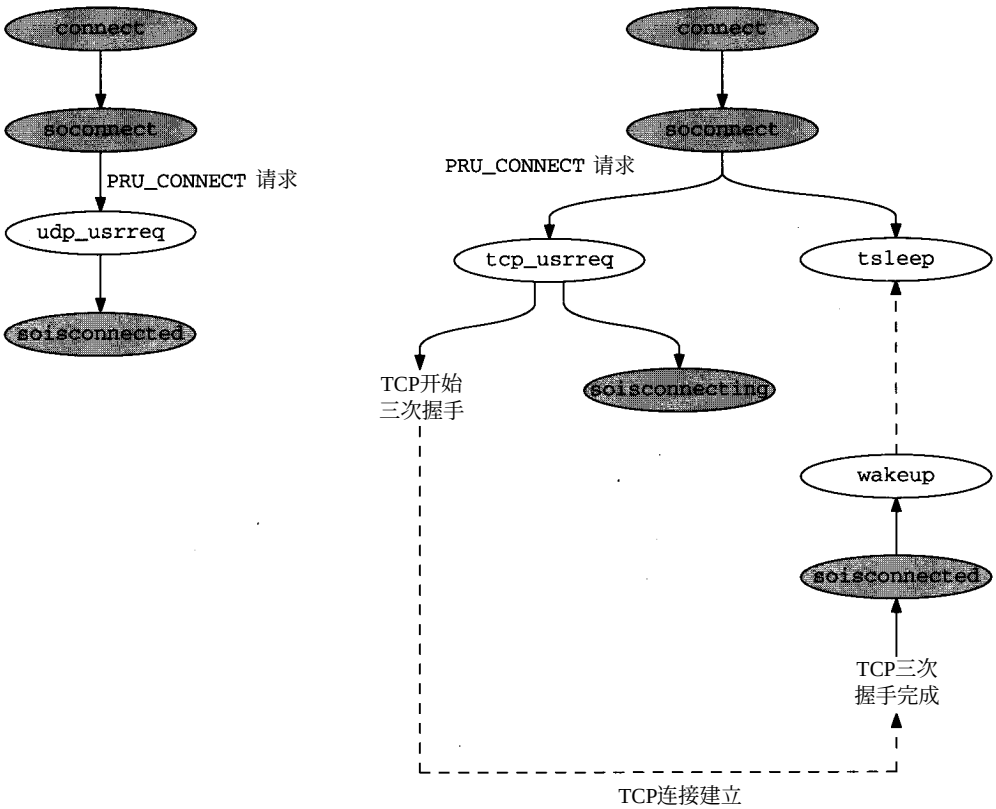


图15-31 connect 处理过程

图的左边说明connect如何处理无连接协议，如 UDP。在这种情况下，协议层调用soisconnected后connect系统调用立即返回。

图的右边说明connect如何处理面向连接的协议，如 TCP。在这种情况下，协议层开始建立连接，调用 soisconnecting指示连接将在某个时候完成。如果插口是非阻塞的，soconnect调用tsleep等待连接完成。对于 TCP，当三次握手完成时，协议层调用soisconnected将插口标识为已连接，然后调用 wakeup唤醒等待的进程，从而完成connect系统调用。

图15-32列出了connect系统调用的代码。

uipc_syscalls.c

```
180 struct connect_args {
181     int     s;
182     caddr_t name;
183     int     namelen;
184 };

185 connect(p, uap, retval)
186 struct proc *p;
187 struct connect_args *uap;
188 int    *retval;
189 {
190     struct file *fp;
191     struct socket *so;
192     struct mbuf *nam;
193     int     error, s;

194     if (error = getsock(p->p_fd, uap->s, &fp))
195         return (error);
196     so = (struct socket *) fp->f_data;
197     if ((so->so_state & SS_NBIO) && (so->so_state & SS_ISCONNECTING))
198         return (EALREADY);
199     if (error = sockargs(&nam, uap->name, uap->namelen, MT_SONAME))
200         return (error);

201     error = soconnect(so, nam);
202     if (error)
203         goto bad;
204     if ((so->so_state & SS_NBIO) && (so->so_state & SS_ISCONNECTING)) {
205         m_freem(nam);
206         return (EINPROGRESS);
207     }
208     s = splnet();
209     while ((so->so_state & SS_ISCONNECTING) && so->so_error == 0)
210         if (error = tsleep((caddr_t) & so->so_timeo, PSOCK | PCATCH,
211                            netcon, 0))
212             break;
213     if (error == 0) {
214         error = so->so_error;
215         so->so_error = 0;
216     }
217     splx(s);
218   bad:
219     so->so_state &= ~SS_ISCONNECTING;
220     m_freem(nam);
221     if (error == ERESTART)
222         error = EINTR;
223     return (error);
224 }
```

uipc_syscalls.c

图15-32 connect 系统调用

180-188 connect的三个参数(在connect_args结构中)是：s为插口描述符；name是一个指针，指向存放外部地址的缓存；namelen为缓存的长度。

189-200 getsock获取插口描述符对应的file结构。可能已有连接请求在非阻塞的插口上，若出现这种情况，则返回EALREADY。函数sockargs将外部地址从进程复制到内核。

1. 开始连接处理

201-208 连接是从调用soconnect开始的。如果soconnect报告差错出现，connect跳转到bad。如果soconnect返回时连接还没有完成且使能了非阻塞的 I/O，则立即返回EINPROGRESS以免等待连接完成。因为通常情况下，建立连接要涉及同远程系统交换几个分组，因而这个过程可能需要一些时间才能完成。如果连接没完成，则下次对 connect调用就返回EALREADY。当连接完成时，soconnect返回EISCONN。

2. 等待连接建立

208-217 while循环直到连接已建立或出现差错时才退出。 splnet防止connect在测试插口状态和调用tsleep之间错过wakeup。循环完成后，error包含0、tsleep中的差错代码或插口中的差错代码。

218-224 清除SS_ISCONNECTING标志，因为连接已完成或连接请求已失败。释放存储外部地址的mbuf，返回差错代码。

15.13.1 soconnect函数

soconnect函数确保插口处于正确的连接状态。如果插口没有连接或连接没有被挂起，则连接请求总是正确的。如果插口已经连接或连接正等待处理，则新的连接请求将被面向连接的协议(如TCP)拒绝。对于无连接协议，如UDP，多个连接是允许的，但是每一个新的请求

uipc_socket.c

```
198 soconnect(so, nam)
199 struct socket *so;
200 struct mbuf *nam;
201 {
202     int     s;
203     int     error;

204     if (so->so_options & SO_ACCEPTCONN)
205         return (EOPNOTSUPP);
206     s = splnet();
207     /*
208      * If protocol is connection-based, can only connect once.
209      * Otherwise, if connected, try to disconnect first.
210      * This allows user to disconnect by connecting to, e.g.,
211      * a null address.
212      */
213     if (so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING) &&
214         ((so->so_proto->pr_flags & PR_CONNREQUIRED) ||
215          (error = sodisconnect(so))))
216         error = EISCONN;
217     else
218         error = (*so->so_proto->pr_usrreq) (so, PRU_CONNECT,
219                                 (struct mbuf *) 0, nam, (struct mbuf *) 0);
220     splx(s);
221     return (error);
222 }
```

uipc_socket.c

图15-33 soconnect 函数

中的外部地址会取代原来的外部地址。

图15-33列出了soconnect函数的代码。

198-222 如果插口被标识准备接收连接，则soconnect返回EOPNOTSUPP，因为如果已经对插口调用了listen，则进程不能再初始化连接。如果协议是面向连接的，且一条连接已经被初始化，则返回EISCONN。对于无连接协议，任何已有的同外部地址的联系都被sodisconnect切断。

PRU_CONNECT请求启动相应的协议处理来建立连接或关联。

15.13.2 切断无连接插口和外部地址的关联

对于无连接协议，可以通过调用connect，并传入一个不正确的name参数，如指向内容为全0的结构指针或大小不对的结构，来丢弃同插口相关联的外部地址。sodisconnect删除同插口相关联的外部地址，PRU_CONNECT返回差错代码，如EAFNOSUPPORT或EADDRNOTAVAIL，留下没有外部地址的插口。这种方式虽然有点晦涩，但却是一种比较有用的断连方式，在无连接插口和外部地址之间断连，而不是替换。

15.14 shutdown系统调用

shutdown系统调用关闭连接的读通道、写通道或读写通道，如图15-34所示。对于读通道，shutdown丢弃所有进程还没有读走的数据以及调用shutdown之后到达的数据。对于写通道，shutdown使协议作相应的处理。对于TCP，所有剩余的数据将被发送，发送完成后发送FIN。这就是TCP的半关闭特点(参考卷1的第18.5节)。

为了删除插口和释放描述符，必须调用close。可以在没有调用shutdown的情况下，直接调用close。同所有描述符一样，当进程结束时，内核将调用close，关闭所有还没有被关闭的插口。

uipc_syscalls.c

```
550 struct shutdown_args {
551     int     s;
552     int     how;
553 };

554 shutdown(p, uap, retval)
555 struct proc *p;
556 struct shutdown_args *uap;
557 int    *retval;
558 {
559     struct file *fp;
560     int     error;

561     if (error = getsock(p->p_fd, uap->s, &fp))
562         return (error);
563     return (soshutdown((struct socket *) fp->f_data, uap->how));
564 }
```

uipc_syscalls.c

图15-34 shutdown 系统调用

550-557 在shutdown_args结构中，s为插口描述符，how指明关闭连接的方式。图15-35列出了how和how++(在图15-36中用到的)的期望值。

how	how++	描 述
0	*FREAD*	关闭连接的读通道
1	*FWRITE*	关闭连接的写通道
2	*FREAD*|*FWRITE*	关闭连接的读写通道

图15-35 shutdown 系统调用选项

注意，在how和常数FREAD和FWRITE之间有一种隐含的数值关系。

558-564 shutdown是函数soshutdown的包装函数(wrapper function)。由getsock返回与描述符相关联的插口，调用soshutdown，并返回其值。

soshutdown和sorflush函数

关闭连接的读通道是由插口层调用 sorflush处理的，写通道的关闭是由协议层的PRU_SHUTDOWN请求处理的。soshutdown函数如图15-36所示。

uipc_socket.c

```
720 soshutdown(so, how)
721 struct socket *so;
722 int     how;
723 {
724     struct protosw *pr = so->so_proto;

725     how++;
726     if (how & FREAD)
727         sorflush(so);
728     if (how & FWRITE)
729         return ((*pr->pr_usrreq) (so, PRU_SHUTDOWN,
730                 (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0));
731     return (0);
732 }
```

uipc_socket.c

图15-36 soshutdown 函数

720-732 如果是关闭插口的读通道，则 sorflush丢弃插口接收缓存中的数据，禁止读连接(如图15-37所示)。如果是关闭插口的写通道，则给协议发送 PRU_SHUTDOWN请求。

733-747 进程等待给接收缓存加锁。因为 SB_NOINTR被设置，所以当中断出现时，sblock并不返回。在修改插口状态时， splimp阻塞网络中断和协议处理，因为协议层在接收到进入的分组时可能要访问接收缓存。

socantrcvmore标识插口拒绝接收进入的分组。将 sockbuf结构保存在asb中，当splx恢复中断后，要使用asb。调用bzero清除原始的sockbuf结构，使得接收队列为空。

释放控制mbuf

748-751 当shutdown被调用时，存储在接收队列中的控制信息可能引用了一些内核资源。通过sockbuf结构的副本中的sb_mb仍然可以访问mbuf链。

如果协议支持访问权限，且注册了一个dom_dispose函数，则调用该函数来释放这些资源。

在Unix域中，用控制报文在进程间传递描述符是可能的。这些报文包含一些引用计数的数据结构的指针。dom_dispose函数负责去掉这些引用，如果必要，还释放相关的数据缓存以避免产生一些未引用的结构和导致内存漏洞。有关在 Unix域内传递文件描述符的细节请参考 [Stevens 1990]和[Leffler et al. 1989]。

uipc_socket.c

```
733 sorflush(so)
734 struct socket *so;
735 {
736     struct sockbuf *sb = &so->so_rcv;
737     struct protosw *pr = so->so_proto;
738     int     s;
739     struct sockbuf asb;

740     sb->sb_flags |= SB_NOINTR;
741     (void) sblock(sb, M_WAITOK);
742     s = splimp();
743     socantrcvmore(so);
744     sbunlock(sb);
745     asb = *sb;
746     bzero((caddr_t) sb, sizeof(*sb));
747     splx(s);

748     if (pr->pr_flags & PR_RIGHTS && pr->pr_domain->dom_dispose)
749         (*pr->pr_domain->dom_dispose) (asb.sb_mb);
750     sbrelease(&asb);
751 }
```

uipc_socket.c

图15-37 sorflush 函数

当sbrelease释放接收队列中的所有mbuf时，丢弃所有调用shutdown时还没有被处理的数据。

注意，连接的读通道的关闭完全由插口层来处理 (习题15.6)，连接的写通道的关闭通过发送PRU_SHUTDOWN请求交由协议处理。TCP协议收到PRU_SHUTDOWN请求后，发送所有排队的数据，然后发送一个FIN来关闭TCP连接的写通道。

15.15 close系统调用

close系统调用能用来关闭各类描述符。当 fd是引用对象的最后的描述符时，与对象有关的close函数被调用：

```
error = (*fp->f_ops->fo_close)(fp,p);
```

如图15-13所示，插口的fp->f_ops->fo_close是soo_close函数。

15.15.1 soo_close函数

soo_close函数是soclose函数的封装器，如图15-38所示。

sys_socket.c

```
152 soo_close(fp, p)
153 struct file *fp;
154 struct proc *p;
155 {
156     int     error = 0;

157     if (fp->f_data)
158         error = soclose((struct socket *) fp->f_data);
159     fp->f_data = 0;
160     return (error);
161 }
```

sys_socket.c

图15-38 soo_close 函数

152-161 如果socket结构与file相关联，则调用soclose，清除f_data，返回已出现的差错。

15.15.2 soclose函数

soclose函数取消插口上所有未完成的连接 (即，还没有完全被进程接受的连接)，等待数据被传输到外部系统，释放不需要的数据结构。

soclose函数的代码如图15-39所示。

```
130 struct socket *so;
131 {
132     int     s = splnet();       /* conservative */
133     int     error = 0;

134     if (so->so_options & SO_ACCEPTCONN) {
135         while (so->so_q0)
136             (void) soabort(so->so_q0);
137         while (so->so_q)
138             (void) soabort(so->so_q);
139     }
140     if (so->so_pcb == 0)
141         goto discard;
142     if (so->so_state & SS_ISCONNECTED) {
143         if ((so->so_state & SS_ISDISCONNECTING) == 0) {
144             error = sodisconnect(so);
145             if (error)
146                 goto drop;
147         }
148         if (so->so_options & SO_LINGER) {
149             if ((so->so_state & SS_ISDISCONNECTING) &&
150                 (so->so_state & SS_NBIO))
151                 goto drop;
152             while (so->so_state & SS_ISCONNECTED)
153                 if (error = tsleep((caddr_t) & so->so_timeo,
154                                    PSOCK | PCATCH, netcls, so->so_linger))
155                     break;
156         }
157     }
158   drop:
159     if (so->so_pcb) {
160         int     error2 =
161         (*so->so_proto->pr_usrreq) (so, PRU_DETACH,
162                 (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0);
163         if (error == 0)
164             error = error2;
165     }
166   discard:
167     if (so->so_state & SS_NOFDREF)
168         panic("soclose: NOFDREF");
169     so->so_state |= SS_NOFDREF;
170     sofree(so);
171     splx(s);
172     return (error);
173 }
```

uipc_socket.c

图15-39 soclose 函数

1. 丢弃未完成的连接

129-141 如果插口正在接收连接，soclose遍历两个连接队列，并且调用soabort取消每一个挂起的连接。如果协议控制块为空，则协议已同插口分离，soclose跳转到discard进行退出处理。

soabort发送PRU_ABORT请求给协议，并返回结果。本书中没有介绍soabort的代码。图23-38和图30-7讨论了UDP和TCP如何处理PRU_ABORT请求。

2. 断开已建立的连接或关联

142-157 如果插口没有同任何外部地址相连接，则跳转到drop处继续执行。否则，必须断开插口与对等地址之间的连接。如果断连没有开始，则sodisconnect启动断连进程。如果设置了SO_LINGER插口选项，soclose可能要等到断连完成后才返回。对于一个非阻塞的插口，从来不需要等待断连完成，所以在这种情况下，soclose立即跳转到drop。否则，连接终止正在进行且SO_LINGER选项指示soclose必须等待一段时间才能完成操作。直到出现下列情况时while才退出：断连完成；拖延时间(so_linger)到；或进程收到了一个信号。

如果滞留时间被设为0，tsleep仅当断连完成(也许因为一个差错)或收到一个信号时才返回。

3. 释放数据结构

158-173 如果插口仍然同协议相连，则发送PRU_DETACH请求断开插口与协议的联系。最后，插口被标记为同任何描述符没有关联，这意味着可以调用sofree释放插口。

sofree函数代码如图15-40所示。

uipc_socket.c

```
110 sofree(so)
111 struct socket *so;
112 {
113     if (so->so_pcb || (so->so_state & SS_NOFDREF) == 0)
114         return;
115     if (so->so_head) {
116         if (!soqremque(so, 0) && !soqremque(so, 1))
117             panic("sofree dq");
118         so->so_head = 0;
119     }
120     sbrelease(&so->so_snd);
121     sorflush(so);
122     FREE(so, M_SOCKET);
123 }
```

uipc_socket.c

图15-40 sofree 函数

4. 如果插口仍在用则返回

110-114 如果仍然有协议同插口相连，或如果插口仍然同描述符相连，则sofree立即返回。

5. 从连接队列中删除插口

115-119 如果插口仍在连接队列上(so_head非空)，则插口的队列应该为空。如果不空，则插口代码和内核panic中有差错。如果队列为空，清除so_head。

6. 释放发送和接收队列中的缓存

120-123 sorelease释放发送队列中的所有缓存，sorflush释放接收队列中的所有缓存。最后，释放插口本身。

15.16 小结

本章中我们讨论了所有与网络操作有关的系统调用。描述了系统调用机制，并且跟踪系统调用直到它们通过pr_usrreq函数进入协议处理层。

在讨论插口层时，我们避免涉及地址格式、协议语义或协议实现等问题。在接下来的章节中，我们将通过协议处理层中的Internet协议的实现将链路层处理和插口层处理联系在一起。

习题

15.1 一个没有超级用户权限的进程怎样才能获取对超级用户进程产生的插口的访问权？

15.2 一个进程怎样才能判断它提供给accept的sockaddr缓存是不是太小以至不能存放调用返回的外部地址？

15.3 IPv6的插口有一个特点：使accept和recvfrom返回一个128 bit的IPv6地址的数组作为源路由，而不是仅返回一个对等地址。因为数组不能存放在一个mbuf中，所以修改accept和recvfrom，使得它们能够处理协议层来的mbuf链而不是仅仅一个mbuf。如果协议在mbuf簇中返回一个数组而不是一个mbuf链，已有的代码仍然能正常工作吗？

15.4 为什么在图15-26中当soqremque返回一个空指针时要调用panic？

15.5 为什么sorflush要复制接收缓存？

15.6 在sorflush将插口的接收缓存清0后，如果还有数据到达会出现什么现象？在做这个习题之前请阅读第16章的内容。

第16章 插 口 I/O

16.1 引言

本章讨论有关从网络连接上读写数据的系统调用，分三部分介绍。

第一部分介绍四个用来发送数据的系统调用：write、writev、sendto和sendmsg。第二部分介绍四个用来接收数据的系统调用：read、readv、recvfrom和recvmsg。第三部分介绍select系统调用，select调用的作用是监控通用描述符和特殊描述符(插口)的状态。

插口层的核心是两个函数：sosend和soreceive。这两个函数负责处理所有插口层和协议层之间的I/O操作。在后续的章节中我们将看到，因为这两个函数要处理插口层和各种类型的协议之间的I/O操作，使得这两个函数特别长和复杂。

16.2 代码介绍

图16-1中列出了本章后续章节要用到的三个头文件和四个 C源文件。

文 件 名	说 明
sys/socket.h	插口API中的结构和宏定义
sys/socketvar.h	socket结构和宏定义
sys/uio.h	uio结构定义
kern/uipc_syscalls.c	socket系统调用
kern/uipc_socket.c	插口层处理
kern/sys_generic.c	select系统调用
kern/sys_socket.c	select对插口的处理

图16-1 本章涉及的头文件和C源文件

全局变量

图16-2列出了三个全局变量。前两个变量由 select系统调用使用，第三个变量控制分配给插口的存储器大小。

变 量	数据类型	说 明
selwait	int	select调用的等待通道
nselcoll	int	避免select调用中出现竞争的标志
sb_max	u_long	插口发送或接收缓存的最大字节数

图16-2 本章涉及的全局变量

16.3 插口缓存

从第15.3节我们已经知道，每一个插口都有一个发送缓存和一个接收缓存。缓存的类型为

sockbuf。图16-3中列出了sockbuf结构的定义(重复图15-5)。

socketvar.h

```
72      struct sockbuf {
73          u_long   sb_cc;          /* actual chars in buffer */
74          u_long   sb_hiwat;       /* max actual char count */
75          u_long   sb_mbcnt;       /* chars of mbufs used */
76          u_long   sb_mbmax;       /* max chars of mbufs to use */
77          long     sb_lowat;       /* low water mark */
78          struct mbuf *sb_mb;      /* the mbuf chain */
79          struct selinfo sb_sel;   /* process selecting read/write */
80          short    sb_flags;       /* Figure 16.5 */
81          short    sb_timeo;       /* timeout for read/write */
82      } so_rcv, so_snd;
```

socketvar.h

图16-3 sockbuf 结构

72-78 每一个缓存均包含控制信息和指向存储数据的mbuf链的指针。sb_mb指向mbuf链的第一个mbuf，sb_cc的值等于存储在mbuf链中的数据字节数。sb_hiwat和sb_lowat用来调整插口的流控算法。sb_mbcnt等于分配给缓存中的所有mbuf的存储器数量。

在前面的章节中提到过每一个 mbuf可存储0~2048个字节的数据 (如果使用了外部簇)。sb_mbmax是分配给插口mbuf缓存的存储器数量的上限。默认的上限在socket系统调用中发送PRU_ATTACH请求时由协议设置。只要内核要求的每个插口缓存的大小不超过 262,144个字节的限制(sb_max)，进程就可以修改缓存的上限和下限。流控算法将在 16.4节和16.8节中讨论。图16-4显示了Internet协议的默认设置。

协 议	so_snd			so_rcv		
	sb_hiwat	sb_lowat	sb_mbmax	sb_hiwat	sb_lowat	sb_mbmax
UDP	9×1024	2048 (忽略)	2×sb_hiwat	40×(1024+16)	1	2×sb_hiwat
TCP	8×1024	2048	2×sb_hiwat	8×1024	1	2×sb_hiwat
原始IP ICMP IGMP	8×1024	2048 (忽略)	2×sb_hiwat	8×1024	1	2×sb_hiwat

图16-4 Internet协议的默认的插口缓存限制

因为每一个进入的UDP报文的源地址同数据一起排队，所以 UDP协议的sb_hiwat的默认值设置为能容纳40个1K字节长的数据报和相应的sockaddr_in结构(每个16字节)。

79 sb_sel是一个用来实现select系统调用的selinfo结构(16.13节)。

80 图16-5列出了sb_flags的所有可能的值。

sb-flags	说 明
SB_LOCK	一个进程已经锁定了插口缓存
SB_WANT	一个进程正在等待给插口缓存加锁
SB_WAIT	一个进程正在等待接收数据或发送数据所需的缓存
SB_SEL	一个或多个进程正在选择这个缓存
SB_ASYNC	为这个缓存产生异步I/O信号
SB_NOINTR	信号不取消加锁请求
SB_NOTIFY	(SB_WAIT|SB_AEL|SB_ASYNC) 一个进程正在等待缓存的变化，如果缓存发生任何改变，用 wakeup通知该进程

图16-5 sb_flags 的值

81-82 sb_timeo用来限制一个进程在读写调用中被阻塞的时间，单位为时钟滴答（tick）。默认值为0，表示进程无限期的等待。SO_SNDTIMEO和SO_RCVTIMEO插口选项可以改变或读取sb_timeo的值。

插口宏和函数

有许多宏和函数用来管理插口的发送和接收缓存。图 16-6中列出了与缓存加锁和同步有关的宏和函数。

名 称	说 明
sblock	申请给*sb*加锁，如果*wf*等于M_WAITOK，则进程睡眠等待加锁；否则，如果不能立即给缓存加锁，就返回 EWOULDBLOCK。如果进程睡眠被一个信号中断，则返回 EINTR或ERESTART；否则返回0 `int sblock(struct sockbuf *sb, int wf);`
sbunlock	释放加在*sb*上的锁。所有等待给*sb*加锁的进程被唤醒 `void sbunlock(struct sockbuf *sb);`
sbwait	调用tsleep等待*sb*上的协议动作。返回tsleep返回的结果 `int sbwait(struct sockbuf *sb);`
sowakeup	通知插口有协议动作出现。唤醒所有匹配的调用 sbwait的进程或在*sb*上调用tsleep的进程 `void sowakeup(struct socket *so, struct sockbuf *sb);`
sorwakeup	唤醒等待*sb*上的读事件的进程，如果进程请求了I/O事件的异步通知，则还应给该进程发送SIGIO信号 `void sorwakeup(struct socket *so);`
sowwakeup	唤醒等待*sb*上的写事件的进程，如果进程请求了I/O事件的异步通知，则还应给该进程发送SIGIO信号 `void sowwakeup(struct socket *so);`

图16-6 与缓存加锁和同步有关的宏和函数

图16-7显示了设置插口资源限制、往缓存中写数据和从缓存中删除数据的宏和函数。在该表中，*m*、*m0*、*n*和*control*都是指向mbuf链的指针。*sb*指向插口的发送或接收缓存。

名 称	说 明

sbspace	*sb*中可用的空间（字节数）：
	`min(sb_hiwat - sb_cc), (sb_mbmax - sb_mbcont)`
	`long sbspace(struct sockbuf *sb);`

sballoc	将*m*加到*sb*中，同时修改*sb*中的sb_cc和sb_mbcnt `void sballoc(struct sockbuf *sb, struct mbuf *m);`
sbfree	从*sb*中删除*m*，同时修改*sb*中的sb_cc和sb_mbcnt `int sbfree(struct sockbuf *sb, struct mbuf *m);`
sbappend	将m中的mbuf加到*sb*的最后面 `int sbappend(struct sockbuf *sb, struct mbuf *m);`
sbappendrecord	将*m0*中的记录加到*sb*的最后面。调用sbcompress `int sbappendrecord(struct sockbuf *sb, struct mbuf *m0);`

图16-7 与插口缓存分配与操作有关的宏和函数

名 称	说 明
sbappendaddr	将*asa*的地址放入一个mbuf。将地址、*control*和*m0*连接成一个mbuf链，并将该链放在*sb*的最后面 `int abappendaddr(struct sb, struct sockaddr asa, struct mbuf m0, struct mbuf control);`
sbappendcontrol	将*control*和*m0*连接成一个mbuf链，并将该链放在*sb*的最后面 `int abappendcontrol(struct sb, struct mbuf m0, struct mbuf control);`
sbinsertoob	将*m0*插在没有带外数据的*sb*的第一个记录的前面 `int abinsertoob(struct sockbuf sb, struct mbuf m0);`
sbcompress	将*m*合并到*n*中并压缩没用的空间 `void abcompress(struct sockbuf sb, struct mbuf m, struct mbuf n);`
sbdrop	删除*sb*的前*len*个字节 `void sbdrop(struct sockbuf sb, int len);`
sbdroprecord	删除*sb*的第一个记录，将下一个记录移作第一个记录 `void sbdroprecord(struct sockbuf sb);`
sbrelease	调用sbflush释放*sb*中所有的mbuf。并将sb_hiwat和sb_mbmax清0 `void sbrelease(struct sockbuf sb);`
sbflush	释放*sb*中的所有mbuf `void sbflush(struct sockbuf sb);`
soreserve	设置插口缓存高、低水位标记(high-water and low-water marks)于发送缓存，调用sbreserve并传入参数*sndcc*。对于接收缓存，调用sbreserve并传入参数rcvcc。将发送缓存和接收缓存的sb_lowat初始化成默认值(图16-4)。如果超过系统限制，则返回ENOBUFS `int soreserve(struct socket so, int sndcc, int rcvcc);`
sbreserve	将*sb*的高水位标记设置成*cc*。同时将低水位标记降到*cc*。本函数不分配存储器 `int sbreserve(struct sockbuf sb, int cc);`

图16-7 (续)

16.4 write、writev、sendto和sendmsg系统调用

我们将write、writev、sendto和sendmsg四个系统调用统称为写系统调用，它们的作用是往网络连接上发送数据。相对于最一般的调用sendmsg而言，前三个系统调用是比较简单的接口。

所有的写系统调用都要直接或间接地调用sosend。sosend的功能是将进程来的数据复制到内核，并将数据传递给与插口相关的协议。图16-8给出了sosend的工作流程。

在下面的章节中，我们将讨论图16-8中带阴影的函数。其余的四个系统调用和soo_write留给读者自己去了解。

图16-9说明了这四个系统调用和一个相关的库函数(send)的特点。

在Net/3中，send被实现成一个调用sendto的库函数。为了与以前编译的程序二进制兼容，内核将旧的send调用映射成函数osend，该函数不在本书中讨论。

从图16-9的第二栏中可以看出，write和writev系统调用适用于任何描述符，而其他的系统调用只适用于插口描述符。

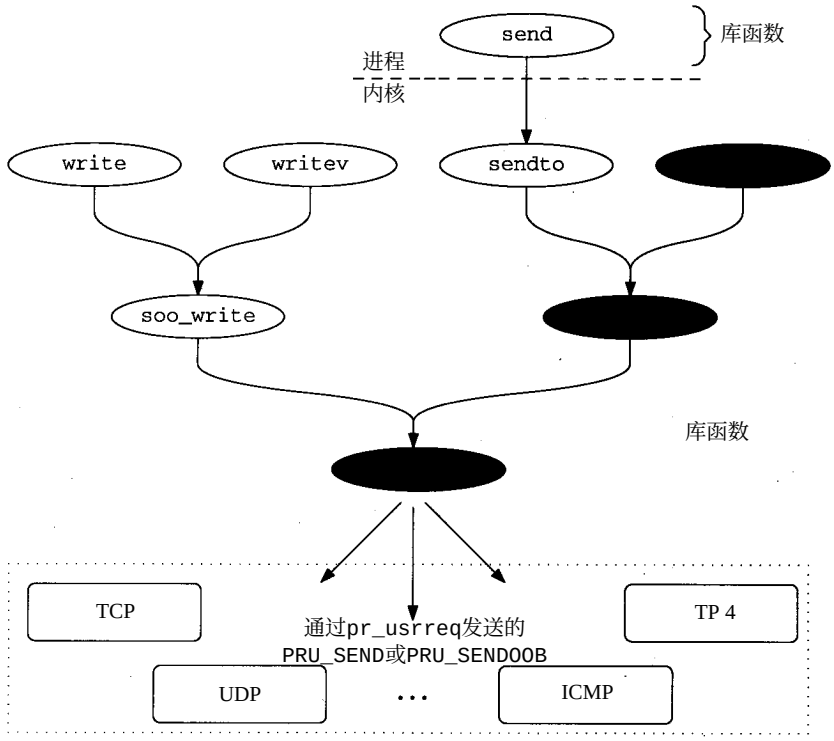


图16-8 所有的插口输出均由 sosend 处理

函 数	描述符类型	缓存数量	是否指明目的地址	标 志?	控制信息?
write	任何类型	1			
writev	任何类型	[1..UIO_MAXIOV]			
send	插口	1		•	
sendto	插口	1	•	•	
sendmsg	插口	[1..UIO-MAXIOV]	•	•	•

图16-9 写系统调用

从图16-9的第三栏中可以看出，writev和sendmsg系统调用可以接收从多个缓存中来的数据。从多个缓存中写数据称为收集 (gathering)，同它相对应的读操作称为分散(scattering)。执行收集操作时，内核按序接收类型为 iovec的数组中指定的缓存中的数据。数组最多有UIO_MAXIOV个单元。图16-10显示了类型iovec的结构。

uio.h

```
41 struct iovec {
42     char   *iov_base;          /* Base address */
43     size_t  iov_len;           /* Length */
44 };
```

uio.h

图16-10 iovec 结构

41-44 在图16-10中，iov_base指向长度为iov_len个字节的缓存的开始。

如果没有这种接口，一个进程将不得不将多个缓存复制到一个大的缓存中，或调用多个

写系统调用来发送从多个缓存来的数据。相对于用一个系统调用传送类型为 iovec的数组，这两种方法的效率更低。对于数据报协议而言，调用一次 writev就是发送一个数据报，数据报的发送不能用多个写动作来实现。

图16-11说明了iovec结构在writev系统调用中的应用，图中iovp指向数组的第一个元素，iovcnt等于数组的大小。

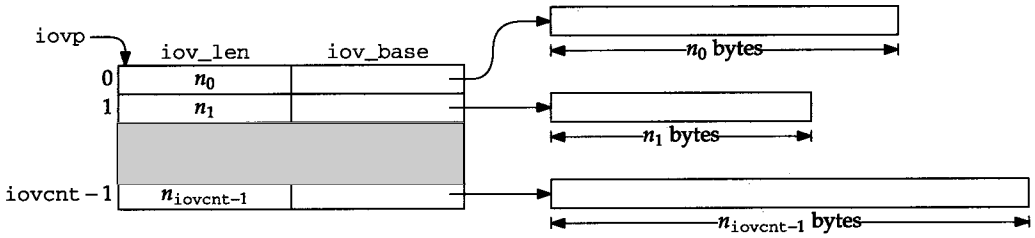


图16-11 writev 系统调用中的iovec 参数

数据报协议要求每一个写调用必须指定一个目的地址。因为 write、writev和send调用接口不支持对目的地址的指定，因此这些调用只能在调用 connect将目的地址同一个无连接的插口联系起来后才能被调用。调用 sendto或sendmsg时必须提供目的地址，或在调用它们之前调用connect来指定目的地址。

图16-9的第五栏显示send *xxx*系统调用接收一个可选的控制标志，这些标志在图 16-12中定义。

flags	描　述	参　考
MSG_DONTROUTE	发送本报文时，不查路由表	图16-23
MSG_DONTWAIT	发送本报文时，不等待资源	图16-22
MSG_EOR	标志一个逻辑记录的结束	图16-25
MSG_OOB	发送带外数据	图16-26

图16-12 send *xxx*系统调用：flags 值

如图16-9的最后一栏所示，只有 sendmsg系统调用支持控制信息。控制信息和另外几个参数是通过结构msghdr(图16-13)一次传递给sendmsg，而不是分别传递。

socket.h

```
228 struct msghdr {
229     caddr_t msg_name;           /* optional address */
230     u_int   msg_namelen;        /* size of address */
231     struct iovec *msg_iov;      /* scatter/gather array */
232     u_int   msg_iovlen;         /* # elements in msg_iov */
233     caddr_t msg_control;        /* ancillary data, see below */
234     u_int   msg_controllen;     /* ancillary data buffer len */
235     int     msg_flags;          /* Figure 16.33 */
236 };
```

socket.h

图16-13 msghdr 结构

msg_name应该被说明成一个指向sockaddr结构的指针，因为它包含网络地址。

228-236 msghdr结构包含一个目的地址(msg_name和msg_namelen)、一个分散/收集数组(msg_iov和msg_iovlen)、控制信息(msg_control和msg_controllen)和接收标志

(msg_flags)。控制信息的类型为cmsghdr结构，如图16-14所示。

socket.h

```
251 struct cmsghdr {
252     u_int    cmsg_len;          /* data byte count, including hdr */
253     int      cmsg_level;        /* originating protocol */
254     int      cmsg_type;         /* protocol-specific type */
255 /* followed by  u_char  cmsg_data[]; */
256 };
```

socket.h

图16-14 cmsghdr 结构

251-256 插口层并不解释控制信息，但是报文的类型被置为 cmsg_type，且报文长度为cmsg_len。多个控制报文可能出现在控制信息缓存中。

举例

图16-15说明了在调用sendmsg时msghdr的结构。

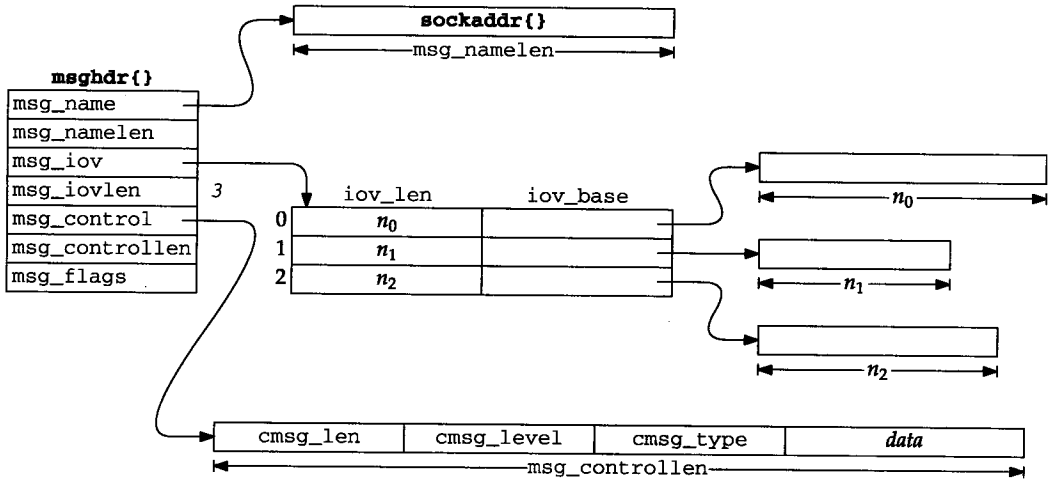


图16-15 sendmsg 系统调用的msghdr 结构

16.5 sendmsg系统调用

只有通过sendmsg系统调用才能访问到与插口 API的输出有关的所有功能。 sendmsg和sendit函数准备sosend系统调用所需的数据结构，然后由 sosend调用将报文发送给相应的协议。对SOCK_DGRAM协议而言，报文就是数据报。对 SOCK_STREAM协议而言，报文是一串字节流。对于SOCK_SEQPACKET协议而言，报文可能是一个完整的记录 (隐含的记录边界)或一个大的记录的一部分 (显式的记录边界)。对于SOCK_PDM协议而言，报文总是一个完整的记录(隐含的记录边界)。

即使一般的 sosend代码处理SOCK_SEQPACKET和SOCK_PDK协议，但是在Internet域中没有这样的协议。

图16-16显示了sendmsg系统调用的源代码。

307-319 sendmsg有三个参数：插口描述符；指向 msghdr结构的指针；几个控制标志。函数copyin将msghdr结构从用户空间复制到内核。

uipc_syscalls.c

```
307 struct sendmsg_args {
308     int     s;
309     caddr_t msg;
310     int     flags;
311 };

312 sendmsg(p, uap, retval)
313 struct proc *p;
314 struct sendmsg_args *uap;
315 int    *retval;
316 {
317     struct msghdr msg;
318     struct iovec aiov[UIO_SMALLIOV], *iov;
319     int     error;

320     if (error = copyin(uap->msg, (caddr_t) & msg, sizeof(msg)))
321         return (error);
322     if ((u_int) msg.msg_iovlen >= UIO_SMALLIOV) {
323         if ((u_int) msg.msg_iovlen >= UIO_MAXIOV)
324             return (EMSGSIZE);
325         MALLOC(iov, struct iovec *,
326                sizeof(struct iovec) * (u_int) msg.msg_iovlen, M_IOV,
327                M_WAITOK);
328     } else
329         iov = aiov;
330     if (msg.msg_iovlen &&
331         (error = copyin((caddr_t) msg.msg_iov, (caddr_t) iov,
332                         (unsigned) (msg.msg_iovlen * sizeof(struct iovec)))))
333                 goto done;
334     msg.msg_iov = iov;
335     error = sendit(p, uap->s, &msg, uap->flags, retval);
336   done:
337     if (iov != aiov)
338         FREE(iov, M_IOV);
339     return (error);
340 }
```

uipc_syscalls.c

图16-16 sendmsg 系统调用

1. 复制iov数组

320-334 一个有8个元素(UIO_SMALLIOV)的iovec数组从栈中自动分配。如果分配的数组不够大，sendmsg将调用MALLOC分配更大的数组。如果进程指定的数组单元大于1024(UIO_MAXIOV)，则返回EMSGSIZE。copyin将iovec数组从用户空间复制到栈中的数组或一个更大的动态分配的数组中。

这种技术避免了调用malloc带来的高代价，因为大多数情况下，数组的单元数小于等于8。

2. sendit和cleanup

335-340 如果sendit返回，则表明数据已经发送给相应的协议或出现差错。sendmsg释放iovec数组(如果它是动态分配的),并且返回sendit调用返回的结果。

16.6 sendit函数

sendit函数是被sendto和sendmsg调用的公共函数。sendit初始化一个uio结构，

将控制和地址信息从进程空间复制到内核。在讨论 sosend之前，我们必须先解释 uiomove函数和uio结构。

16.6.1 uiomove函数

uiomove函数的原型为：

```
int uiomove(caddr_t cp, int n, struct uio *uio);
```

uiomove函数的功能是在由*cp*指向的缓存与*uio*指向的类型为*iovec*的数组中的多个缓存之间传送*n*个字节。图16-7说明了uio结构的定义，该结构控制和记录uiomove的行为。

uio.h

```
45 enum uio_rw {
46     UIO_READ, UIO_WRITE
47 };

48 enum uio_seg {                    /* Segment flag values */
49     UIO_USERSPACE,                /* from user data space */
50     UIO_SYSSPACE,                 /* from system space */
51     UIO_USERISPACE                /* from user instruction space */
52 };

53 struct uio {
54     struct iovec *uio_iov;        /* an array of iovec structures */
55     int      uio_iovcnt;          /* size of iovec array */
56     off_t    uio_offset;          /* starting position of transfer */
57     int      uio_resid;           /* remaining bytes to transfer */
58     enum uio_seg uio_segflg;      /* location of buffers */
59     enum uio_rw uio_rw;           /* direction of transfer */
60     struct proc *uio_procp;       /* the associated process */
61 };
```

uio.h

图16-17 uio 结构

45-61 在uio结构中，uio_iov指向类型为iovec结构的数组，uio_offset记录uiomove传送的字节数，uio_resid记录剩余的字节数。每次调用 uiomove，uio_offset增加*n*，uio_resid减去*n*。同时，uiomove根据传送的字节数调整uio_iov数组中的基指针和缓存长度，从而从缓存中删除每次调用时传送的字节。最后，每当从uio_iov中传送一块缓存，uio_iov数组的每个单元就向前进一个数组单元。uio_segflg指向uio_iov数组的基指针指向的缓存的位置。uio_rw指定数据传送的方向。缓存可能在用户数据空间，用户指令空间或内核数据空间。图 16-18对uiomove函数的操作进行了小结。图中对操作的描述用到了uiomove函数原型中的参数名。

uio_segflg	uio_rw	描 述
UIO_USERSPACE	*UIO-READ*	从内核缓存*cp*中分散*n*个字节到进程缓存
UIO_USERISPACE		
UIO_USERSPACE	*UIO-WRITE*	从进程缓存中收集*n*个字节到内核缓存*cp*
UIO_USERISPACE		
UIO_SYSSPACE	*UIO-READ*	从内核缓存*cp*中分散*n*个字节到多个内核缓存
	UIO-WRITE	从多个内核缓存中收集*n*个字节到内核缓存*cp*中

图16-18 uiomove 操作

16.6.2 举例

图16-19显示了一个调用uiomove之前的uio结构。

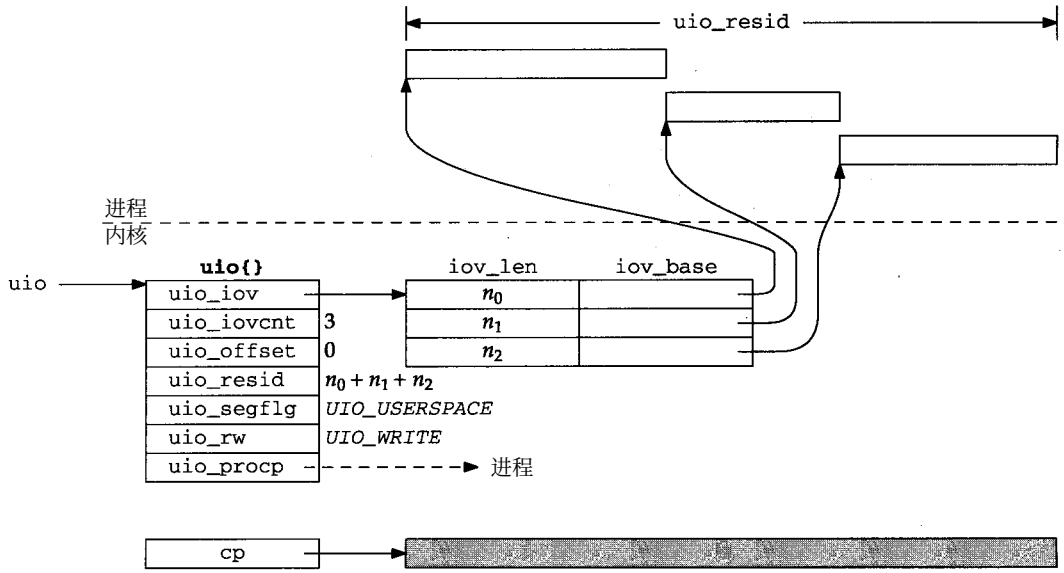


图16-19 调用uiomove 前的uio 结构

uio_iov指向iovec数组的第一个单元。iov_base指针数组的每一个单元分别指向它们在进程地址空间中的缓存的起始地址。uio_offset等于0，uio_resid等于三块缓存的总的大小。cp指向内核中的一块缓存，一般来说，这块缓存是一个 mbuf的数据区。图16-20显示了调用uiomove之后同一个uio结构的内容。

```
uiomove(cp, n, uio);
```

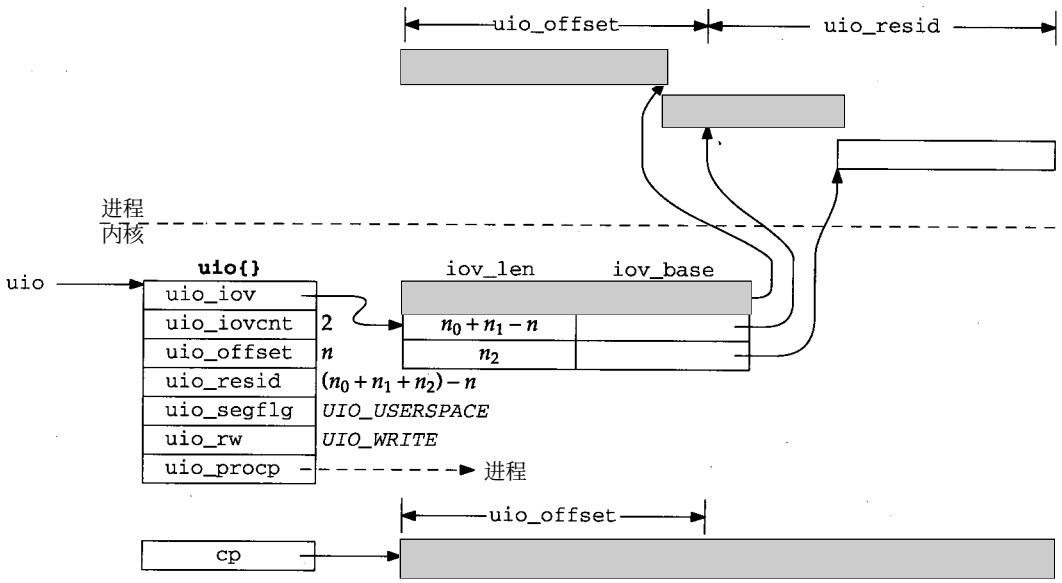


图16-20 调用uiomove 后的uio 结构

在上述调用中， n包括第一块缓存中的所有字节和第二块缓存中的部分字节 (即，$n_0<n<n_0+n_1$)。

调用uiomove后，第一块缓存的长度变为0，且它的基指针指向缓存的末端。uio_iov现在指向iovec数组的下一个单元。单元指针也前进了一个单元，长度也减少了，减少的字节数等于缓存中被传送的字节数。同时，uio_offset增加了n，uio_resid减少了n。数据已经从进程中的缓存传送到内核缓存，因为uio_rw等于UIO_WRITE.。

16.6.3 sendit代码

现在开始讨论sendit的代码，如图16-21所示。

1. 初始化auio

341-368 sendit调用getsock函数获取描述符对应的file结构，初始化uio结构，并将进程指定的输出缓存中的数据收集到内核缓存中。传送的数据的长度通过一个 for循环来计算，并将结果保存在uio_resid。循环内的第一个if保证缓存的长度非负。第二个if保证uio_resid不溢出，因为uio_resid是一个有符号的整数，且iov_len要求非负。

2. 从进程复制地址和控制信息

369-385 如果进程提供了地址和控制信息,则sockargs将地址和控制信息复制到内核缓存中。

uipc_syscalls.c

```
341 sendit(p, s, mp, flags, retsize)
342 struct proc *p;
343 int     s;
344 struct msghdr *mp;
345 int     flags, *retsize;
346 {
347     struct file *fp;
348     struct uio auio;
349     struct iovec *iov;
350     int     i;
351     struct mbuf *to, *control;
352     int     len, error;
353     if (error = getsock(p->p_fd, s, &fp))
354         return (error);
355     auio.uio_iov = mp->msg_iov;
356     auio.uio_iovcnt = mp->msg_iovlen;
357     auio.uio_segflg = UIO_USERSPACE;
358     auio.uio_rw = UIO_WRITE;
359     auio.uio_procp = p;
360     auio.uio_offset = 0;        /* XXX */
361     auio.uio_resid = 0;
362     iov = mp->msg_iov;
363     for (i = 0; i < mp->msg_iovlen; i++, iov++) {
364         if (iov->iov_len < 0)
365             return (EINVAL);
366         if ((auio.uio_resid += iov->iov_len) < 0)
367             return (EINVAL);
368     }
369     if (mp->msg_name) {
370         if (error = sockargs(&to, mp->msg_name, mp->msg_namelen,
371                              MT_SONAME))
```

图16-21 sendit 函数

```
372             return (error);
373     } else
374         to = 0;
375     if (mp->msg_control) {
376         if (mp->msg_controllen < sizeof(struct cmsghdr)
377         ) {
378             error = EINVAL;
379             goto bad;
380         }
381         if (error = sockargs(&control, mp->msg_control,
382                              mp->msg_controllen, MT_CONTROL))
383             goto bad;
384     } else
385         control = 0;
386     len = auio.uio_resid;
387     if (error = sosend((struct socket *) fp->f_data, to, &auio,
388                        (struct mbuf *) 0, control, flags)) {
389         if (auio.uio_resid != len && (error == ERESTART ||
390                                       error == EINTR || error == EWOULDBLOCK))
391             error = 0;
392         if (error == EPIPE)
393             psignal(p, SIGPIPE);
394     }
395     if (error == 0)
396         *retsize = len - auio.uio_resid;
397 bad:
398     if (to)
399         m_freem(to);
400     return (error);
401 }
```

uipc_syscalls.c

图16-21 (续)

3. 发送数据和清除缓存

386-401 为了防止sosend不接受所有数据而无法计算传送的字节数，将uio_resid的值保存在len中。将插口、目的地址、uio结构、控制信息和标志全部传给函数sosend。当sosend返回后，sendit响应如下：

- 如果sosend传送了部分数据后，传送被信号或阻塞条件所中断，差错被丢弃，报告传送了部分数据。
- 如果sosend返回EPIPE，则发送信号SIGPIPE给进程。error设置成非0，所以如果进程捕捉到了该信号，并且从信号处理程序中返回，或进程忽略信号，写调用返回EPIPE。
- 如果没有差错出现(或差错被丢弃)，则计算传送的字节数，并将其保存在*retsize中。如果sendit返回0，syscall(第15.4节)返回*retsize给进程而不是返回差错代码。
- 如果任何其他类型的差错出现，返回相应差错代码给进程。

在返回之前，sendit释放包含目的地址的缓存。sosend负责释放control缓存。

16.7 sosend函数

sosend是插口层中最复杂的函数之一。在图16-8中已提到过所有五个写系统调用最终都要调用sosend。sosend的功能就是：根据插口指明的协议支持的语义和缓存的限制，将数

据和控制信息传递给插口指明的协议的pr_usrreq函数。sosend从不将数据放在发送缓存中；存储和移走数据应由协议来完成。

sosend对发送缓存的sb_hiwat和sb_lowat值的解释，取决于对应的协议是否实现可靠或不可靠的数据传送功能。

16.7.1 可靠的协议缓存

对于提供可靠的数据传送协议，发送缓存保存了还没有发送的数据和已经发送但还没有被确认的数据。sb_cc等于发送缓存的数据的字节数，且0≤sb_cc≤sb_hiwat。

如果有带外数据发送，则sb_cc有可能暂时超过sb_hiwat。

sosend应该确保在通过pr_usrreq函数将数据传递给协议层之前有足够的发送缓存。协议层将数据放到发送缓存中。sosend通过下面两种方式之一将数据传送给协议层：

- 如果设置了PR_ATOMIC，sosend就必须保护进程和协议层之间的边界。在这种情况下，sosend等待得到足够的缓存来存储整个报文。当获取到足够的缓存后，构造存储整个报文的mbuf链，并用pr_usrreq函数一次性传送给协议层。RDP和SPP就是这种类型的协议。
- 如果没有设置PR_ATOMIC，sosend每次传送一个存有报文的mbuf给协议，可能传送部分mbuf给协议层以防止超过上限。这种方法在SOCK_STREAM类协议如TCP中和SOCK_SEQPACKET类协议如TP4中被采用。在TP4中，记录边界通过MSG_EOR标志(图16-12)来显式指定，所以sosend没有必要保护报文边界。

TCP应用程序对外出的TCP报文段的大小没有控制。例如，在TCP插口上发送一个长度为4096字节的报文，假定发送缓存中有足够的缓存，则插口层将该报文分成两部分，每一部分长度为2048个字节，分别存放在一个带外部簇的mbuf中。然后，在协议处理时，TCP将根据连接上的最大报文段大小将数据分段，通常情况下，最大报文段大小为2048个字节。

当一个报文因为太大而没有足够的缓存时，协议允许报文被分成多段，但sosend仍然不将数据传送给协议层直到缓存中的闲置空间大小大于sb_lowat。对于TCP而言，sb_lowat的默认值为2048 (图16-4)，从而阻止插口层在发送缓存快满时用小块数据干扰TCP。

16.7.2 不可靠的协议缓存

对于提供不可靠的数据传输的协议(如UDP)而言，发送缓存不需保存任何数据，也不等待任何确认。每一个报文一旦被排队等待发送到相应的网络设备，插口层立即将它传送到协议。在这种情况下，sb_cc总是等于0，sb_hiwat指定每一次写的最大长度，间接指明数据报的最大长度。

图16-4显示了UDP协议的sb_hiwat的默认值为9216(9×1024)。如果进程没有通过SO_SNDBUF插口选项改变sb_hiwat的值，则发送长度大于9216个字节的数据报将导致差错。不仅如此，其他的协议限制也可能不允许一个进程发送大的数据报报文。卷 1的第11.10节中已讨论了在其他的TCP/IP实现中的这些选项和限制。

对于NFS写而言，9216已足够大，NFS写的数据加上协议首部的长度一般默认为8192个字节。

图16-22显示了sosend函数的概况。下面分别讨论图中四个带阴影的部分。

uipc_socket.c

```
271 sosend(so, addr, uio, top, control, flags)
272 struct socket *so;
273 struct mbuf *addr;
274 struct uio *uio;
275 struct mbuf *top;
276 struct mbuf *control;
277 int     flags;
278 {
                    /* initialization (Figure 16.23) */
305   restart:
306     if (error = sblock(&so->so_snd, SBLOCKWAIT(flags)))
307         goto out;
308     do {                        /* main loop, until resid == 0 */
                /* wait for space in send buffer (Figure 16.24) */
342         do {
343             if (uio == NULL) {
344                 /*
345                  * Data is prepackaged in "top".
346                  */
347                 resid = 0;
348                 if (flags & MSG_EOR)
349                     top->m_flags |= M_EOR;
350             } else
351                 do {
                        /* fill a single mbuf or an mbuf chain (Figure 16.25) */
396                 } while (space > 0 && atomic);
                    /* pass mbuf chain to protocol (Figure 16.26) */
412         } while (resid && space > 0);
413     } while (resid);

414   release:
415     sbunlock(&so->so_snd);
416   out:
417     if (top)
418         m_freem(top);
419     if (control)
420         m_freem(control);
421     return (error);
422 }
```

uipc_socket.c

图16-22 sosend 函数：概述

271-278 sosend的参数有如下几个：so，指向相应插口的指针；addr，指向目的地址的指针；uio，指向描述用户空间的 I/O缓存的uio结构；top，保存将要发送的数据的 mbuf链；control，保存将要发送的控制信息的mbuf链；flags，包含本次写调用的一些选项。

正常情况下，进程通过uio机制将数据提供给插口层， top为空。当内核本身正在使用插口层时(如NFS)，数据将作为一个mbuf链传送给sosend， top指向该mbuf链，而uio为空。

279-304 初始化代码分别如下所述。

1. 给发送缓存加锁

305-308 sosend的主循环从restart开始，在循环的开始调用sblock给发送缓存加锁。通过加锁确保多个进程按序互斥访问插口缓存。

如果在flags中MSG_DONTWAIT被设置，则SBLOCKWAIT将返回M_NOWAIT。M_NOWAIT告知sblock，如果不能立即加锁，则返回EWOULDBLOCK。

MSG_DONTWAIT仅用于Net/3中的NFS。

主循环直到将所有数据都传送给协议 (即resid=0)后才退出。

2. 检查空间

309-341 在传送数据给协议之前，需要对各种差错情况进行检查，并且 sosend实现前面讨论的流控和资源控制算法。如果 sosend阻塞等待输出缓存中的更多的空间，则它跳回restart等待。

3. 使用top中的数据

342-350 一旦有了足够的空间并且sosend也获得了发送缓存上的锁，则准备传送给协议的数据。如果uio等于空(即数据在top指向的mbuf链中)，则sosend检查MSG_EOR，并且在链中设置M_EOR来标志逻辑记录的结束。 mbuf链是准备发送给协议层的。

4. 从进程复制数据

351-396 如果uio不空，则sosend必须从进程间复制数据。当PR_ATOMIC被设置时(例如，UDP)，循环继续，直到所有数据都被复制到一个 mbuf链中。当sosend从进程得到所有数据后，通过循环中的break(图16-22中没有显示这个break)跳出循环。跳出循环后， sosend将整个数据链一次传送给相应协议。

5. 传送数据给协议

395-414 对于PR_ATOMIC协议，当整个数据链被传送给协议后， resid总是等于0，并且控制跳出两个循环后至release处。如果PR_ATOMIC没有被置位，且当还有数据要发送并有缓存空间时，则 sosend继续往mbuf中写数据。如果缓存中没有闲置空间，但仍然有数据要发送，则 sosend回到循环开始，等待闲置空间来写下一个 mbuf。如果所有数据都发送完，则两个循环结束。

6. 释放缓存

414-422 当所有数据都传送给协议后，给插口缓存解锁，释放多余的 mbuf缓存，然后返回。

sosend的详细情况将分四个部分来描述：

- 初始化(图16-23)
- 差错和资源检查(图16-24)
- 数据传送(图16-25)
- 协议处理(图16-26)

sosend的第一部分初始化变量，如图 16-23所示。

7. 计算传送大小和语义

279-284 如果sosendallatonce等于true(任何设置了PR_ATOMIC的协议)或数据已经

通过top中的mbuf链传送给sosend，则将设置atomic。这个标志控制数据是作为一个mbuf链还是作为多个独立的mbuf传送给协议。

285-297 resid等于iovec缓存中的数据字节数或top中的mbuf链中的数据字节数。习题16.1讨论为什么resid可能等于负数的问题。

uipc_socket.c

```
279     struct proc *p = curproc;    /* XXX */
280     struct mbuf **mp;
281     struct mbuf *m;
282     long     space, len, resid;
283     int      clen = 0, error, s, dontroute, mlen;
284     int      atomic = sosendallatonce(so) || top;
285     if (uio)
286         resid = uio->uio_resid;
287     else
288         resid = top->m_pkthdr.len;
289     /*
290      * In theory resid should be unsigned.
291      * However, space must be signed, as it might be less than 0
292      * if we over-committed, and we must use a signed comparison
293      * of space and resid.  On the other hand, a negative resid
294      * causes us to loop sending 0-length segments to the protocol.
295      */
296     if (resid < 0)
297         return (EINVAL);
298     dontroute =
299         (flags & MSG_DONTROUTE) && (so->so_options & SO_DONTROUTE) == 0 &&
300         (so->so_proto->pr_flags & PR_ATOMIC);
301     p->p_stats->p_ru.ru_msgsnd++;
302     if (control)
303         clen = control->m_len;
304 #define snderr(errno)    { error = errno; splx(s); goto release; }
```

uipc_socket.c

图16-23 sosend 函数：初始化

8. 关闭路由

298-303 如果仅仅要求对这个报文不通过路由表进行路由选择，则设置 dontroute。clen等于在可选的控制缓存中的字节数。

304 宏snderr传送差错代码，重新使能协议处理，控制跳转到 out执行解锁和释放缓存的工作。这个宏简化函数内的差错处理工作。

图16-24显示的sosend代码功能是检查差错条件和等待发送缓存中的闲置空间。

309 当检查差错情况时，为防止缓存发生改变，协议处理被挂起。在每一次数据传送之前，sosend要检查以下几种差错情况：

310-311 • 如果插口输出被禁止(即，TCP连接的写道通已经被关闭)，则返回EPIPE。

312-313 • 如果插口正处于差错状态(例如，前一个数据报可能已经产生了一个ICMP不可达的差错)，则返回so_error。如果差错出现之前数据已经被收到，则 sendit忽略这个差错(图16-21的第389行)。

314-318 • 如果协议请求连接且连接还没有建立或连接请求还没有启动，则返回ENOTCONN。sosend允许只有控制信息但没有数据的写操作，即使连接还没有建立。

Internet协议并不使用这个特点，但TP4用它在连接请求中发送数据，证实连接请

求，在断连请求中发送数据。

319-321 • 如果在无连接协议中没有指定目的地址 (例如，进程调用 send但并没有用connect建立目的地址)，则返回EDESTADDREQ。

uipc_socket.c

```
309         s = splnet();
310         if (so->so_state & SS_CANTSENDMORE)
311             snderr(EPIPE);
312         if (so->so_error)
313             snderr(so->so_error);
314         if ((so->so_state & SS_ISCONNECTED) == 0) {
315             if (so->so_proto->pr_flags & PR_CONNREQUIRED) {
316                 if ((so->so_state & SS_ISCONFIRMING) == 0 &&
317                     !(resid == 0 && clen != 0))
318                     snderr(ENOTCONN);
319             } else if (addr == 0)
320                 snderr(EDESTADDRREQ);
321         }
322         space = sbspace(&so->so_snd);
323         if (flags & MSG_OOB)
324             space += 1024;
325         if (atomic && resid > so->so_snd.sb_hiwat ||
326             clen > so->so_snd.sb_hiwat)
327             snderr(EMSGSIZE);
328         if (space < resid + clen && uio &&
329             (atomic || space < so->so_snd.sb_lowat || space < clen)) {
330             if (so->so_state & SS_NBIO)
331                 snderr(EWOULDBLOCK);
332             sbunlock(&so->so_snd);
333             error = sbwait(&so->so_snd);
334             splx(s);
335             if (error)
336                 goto out;
337             goto restart;
338         }
339         splx(s);
340         mp = &top;
341         space -= clen;
```

uipc_socket.c

图16-24 sosend 函数：差错和资源检查

9. 计算可用空间

322-324 sbspace函数计算发送缓存中剩余的闲置空间字节数。这是一个基于缓存高水位标记的管理上的限制，但也是sb_mbmax对它的限制，其目的是为了防止太多的小报文消耗太多的mbuf缓存(图16-6)。sosend通过放宽缓存限制到1024个字节来给予带外数据更高的优先级。

10. 强制实施报文大小限制

325-327 如果atomic被置位，并且报文大于高水位标记(high-watermark)，则返回EMSGSIZE；报文因为太大而不被协议接受，即使缓存是空的。如果控制信息的长度大于高水位标记，同样返回EMSGSIZE。这是限制数据或记录大小的测试代码。

11. 等待更多的空间吗？

328-329 如果发送缓存中的空间不够，数据来源于进程 (而不是来源于内核中的top)，并且下列条件之一成立，则sosend必须等待更多的空间：

• 报文必须一次传送给协议(atomic为真)；或
• 报文可以分段传送，但闲置空间大小低于低水位标记；或
• 报文可以分段传送，但可用空间存放不下控制信息。

当数据通过top传送给sosend (即，uio为空)时，数据已经在mbuf缓存中。因此，sosend忽略缓存高、低水位标记限制，因为不需要附加的缓存来保存数据。

如果在测试中，忽略发送缓存的低水位标记，在插口层和运输层之间将出现一种有趣的交互过程，它将导致性能下降。[Crowcroft et al. 1992]提供了有关这个问题的详细情况。

12. 等待空间

330-338 如果sosend必须等待缓存且插口是非阻塞的，则返回EWOULDBLOCK。同时，缓存锁被释放，sosend调用sbwait等待，直到缓存状态发生变化。当sbwait返回后，sosend重新使能协议处理，并且跳转到restart获取缓存锁，检查差错和缓存空间。如果条件满足，则继续执行。

默认情况下，sbwait阻塞直到可以发送数据。通过SO_SNDTIMEO插口选项改变缓存中的sb_timeo，进程可以设置等待时间的上限。如果定时器超时，则返回EWOULDBLOCK。回想一下图16-21，如果数据已经被成功发送给协议，则sendit忽略这个差错。这个定时器并不限制整个调用的时间，而仅仅是限制写两个mbuf缓存之间的不活动时间。

339-341 在这点上，sosend已经知道一些数据已传送给协议。splx使能中断，因为sosend从进程复制数据到内核相对较长的时间间隔内不应该被阻塞。mp包含一个指针，用来构造mbuf链。在sosend从进程复制任何数据之前，可用缓存的数量需减去控制信息的大小(clen)。

图16-25显示了sosend从进程复制数据到一个或多个内核中的mbuf中的代码段。

13. 分配分组首部或标准mbuf

351-360 当atomic被置位时，这段代码在第一次循环时分配一个分组首部，随后分配标准的mbuf缓存。如果atomic没有被置位，则这段代码总是分配一个分组首部，因为进入循环之前，top总是被清除。

14. 尽可能用簇

361-371 如果报文足够大使得为其分配一个簇是值得的，并且space大于或等于MCLBYTES，则调用MCLGET分配一个簇同mbuf连在一起。当space小于MCLBYTES时，额外的2048个字节将超过缓存分配限制，因为即使resid小于MCLBYTES，整个簇也将被分配。

如果调用MCLGET失败，sosend跳转到nopages，用一个标准的mbuf代替一个外部簇。

对MINCLSIZE的测试应该用>，而不是≥，因为208(MINCLSIZE)个字节的写操作只适合小于两个mbuf的情况。

如果atomic被设置(例如，UDP)，则mbuf链表示一个数据报或记录，并且在第一个簇的前面为协议首部保留max_hdr个字节。而后续的簇因为是同一条链的一部分，所以不需要再为协议首部保留空间。

如果atomic没有被置位(如，TCP)，则不需要保留空间，因为sosend不知道协议如何将发送的数据进行分段。

需要注意的是，space由簇大小(2048个字节)而不是len来决定，len等于放在簇中的数据的字节数(习题16-2)。

uipc_socket.c

```
351                 do {
352                     if (top == 0) {
353                         MGETHDR(m, M_WAIT, MT_DATA);
354                         mlen = MHLEN;
355                         m->m_pkthdr.len = 0;
356                         m->m_pkthdr.rcvif = (struct ifnet *) 0;
357                     } else {
358                         MGET(m, M_WAIT, MT_DATA);
359                         mlen = MLEN;
360                     }
361                     if (resid >= MINCLSIZE && space >= MCLBYTES) {
362                         MCLGET(m, M_WAIT);
363                         if ((m->m_flags & M_EXT) == 0)
364                             goto nopages;
365                         mlen = MCLBYTES;
366                         if (atomic && top == 0) {
367                             len = min(MCLBYTES - max_hdr, resid);
368                             m->m_data += max_hdr;
369                         } else
370                             len = min(MCLBYTES, resid);
371                         space -= MCLBYTES;
372                     } else {
373                       nopages:
374                         len = min(min(mlen, resid), space);
375                         space -= len;
376                         /*
377                          * For datagram protocols, leave room
378                          * for protocol headers in first mbuf.
379                          */
380                         if (atomic && top == 0 && len < mlen)
381                             MH_ALIGN(m, len);
382                     }
383                     error = uiomove(mtod(m, caddr_t), (int) len, uio);
384                     resid = uio->uio_resid;
385                     m->m_len = len;
386                     *mp = m;
387                     top->m_pkthdr.len += len;
388                     if (error)
389                         goto release;
390                     mp = &m->m_next;
391                     if (resid <= 0) {
392                         if (flags & MSG_EOR)
393                             top->m_flags |= M_EOR;
394                         break;
395                     }
396                 } while (space > 0 && atomic);
```

uipc_socket.c

图16-25 sosend 函数：数据传送

15. 准备mbuf

372-382 如果不用簇，存储在 mbuf中的字节数受下面三个量中最小一个量的限制： (1) mbuf中的可用空间；(2) 报文的字节数；(3) 缓存的空间。

如果atomic被置位，则利用MH_ALIGN可知数据在链中的第一个缓存的尾部。如果数据占居整个mbuf，则忽略MH_ALIGN。这一点可能导致没有足够的空间来存放协议首部，主要取决于有多少数据存放在mbuf中。如果atomic没有被置位，则没有为协议首部保留空间。

16. 从进程复制数据

383-395 uiomove从进程复制len个字节的数据到mbuf。传送完成后，更新mbuf的长度，前面的mbuf连接到新的mbuf(或top指向第一个mbuf)，更新mbuf链的长度。如果在传送过程中发生差错，则sosend跳转到release。

一旦最后一个字节传送完毕，如果进程设置了MSG_EOR，则设置分组中的M_EOR，然后sosend跳出循环。

MSG_EOR仅用于有显式的记录边界的协议，如OSI协议簇中的TP4。TCP不支持逻辑记录因而忽略MSG_EOR标志。

17. 写另一个缓存吗?

396 如果设置了atomic，sosend回到循环开始，写另一个mbuf。

> 对space>0的测试好像无关紧要。当atomic没有被设置时，space也是无关紧要的，因为一次只传送一个mbuf给协议。如果设置了atomic，只有当有足够的缓存空间来存放整个报文时才进入这个循环。参考习题16-2。

sosend的最后一段代码的功能是传送数据和控制mbuf给插口指定的协议，如图16-26所示。

uipc_socket.c

```
397             if (dontroute)
398                 so->so_options |= SO_DONTROUTE;
399             s = splnet();       /* XXX */
400             error = (*so->so_proto->pr_usrreq) (so,
401                                 (flags & MSG_OOB) ? PRU_SENDOOB : PRU_SEND,
402                                                 top, addr, control);
403             splx(s);
404             if (dontroute)
405                 so->so_options &= ~SO_DONTROUTE;
406             clen = 0;
407             control = 0;
408             top = 0;
409             mp = &top;
410             if (error)
411                 goto release;
412         } while (resid && space > 0);
413     } while (resid);
```

uipc_socket.c

图16-26 sosend 函数：协议分散

397-405 在传送数据到协议层的前后，可能通过SO_DONTROUTE选项选择是否利用路由表为这个报文选择路由。这是唯一的一个针对单个报文的选项，如图16-23所示，在写期间通过MSG_DONTROUTE标志来控制路由选择。

为了防止协议在处理报文期间pr_usrreq阻塞中断，pr_usrreq被放在splnet函数和splx函数之间执行。一些协议(如UDP)可能在进行输出处理期间并不阻塞中断，但插口层得不到这些信息。

如果进程传送的是带外数据，则sosend发送PRU_SENDOOB请求；否则，它发送PRU_SEND请求。同时将地址和控制mbuf传送给协议。

406-413 因为控制信息只需传送给协议一次，所以将clen、control、top和mp初始化，然后为传送报文的下一部分构造新的mbuf链。只有atomic没有被设置时(如TCP)，resid才

可能等于非0。在这种情况下，如果缓存中仍然有空间，则 sosend回到循环开始，继续写另一个mbuf。如果没有可用空间，则sosend回到循环开始，等待可用空间(图16-24)。

在第23章我们将了解到不可靠的协议，如 UDP，立即将数据排队等待发送。第 26章描述可靠的协议，如TCP，将数据放到插口发送缓存直到数据被发送和确认。

16.7.3 sosend函数小结

sosend是一个比较复杂的函数。它共有 142行，包含3个嵌套的循环，一个利用 goto实现的循环，两个基于是否设置 PR_ATOMIC的代码分支，两个并行锁。像许多其他软件一样，复杂性是多年积累的结果。 NFS加入MSG_DONTWAIT功能以及从mbuf链接收数据而不是从进程那里接收数据。 SS_ISCONFIRMING状态和MSG_EOR标志是为处理OSI协议连接和记录功能而加入的。

比较好的做法是为每一种协议实现一个独立的 sosend函数，通过分散指针 pr_send给protosw入口来实现。[Partridge and Pink 1993]中提出并实现了这种方法。

16.7.4 性能问题

如图16-25所描述的， sosend尽可能地以mbuf为单位将报文传送到协议层。与将一个报文用一个mbuf链的形式一次建立并传送给协议层的方法相比，这种做法导致了更多的调用，但是[Jacobson 1998a]说明了这种做法增加了并行性，因而获得了较好的性能。

一次传送一个 mbuf(2048个字节)允许CPU在网络硬件传输数据的同时准备一个分组。同发送一个大的 mbuf链相比：构造一个大的 mbuf链的同时，网络和接收系统是空闲的。在[Jacobson 1998a]描述的系统中，这种改变导致了网络吞吐量增加 20%。

有一点非常重要，即确保发送缓存的大小总是大于连接的带宽和时延的乘积 (卷1的第20.7节)。例如，如果TCP认为一条连接在收到确认之前能保留 20个报文段，那么发送缓存必须大到足够存储20个未被确认的报文段。如果发送缓存太小， TCP在收到第一个确认之前将用完数据，连接将在一段时间内是空闲的。

16.8 read、readv、recvfrom和recvmsg系统调用

我们将read、readv、recvfrom和recvmsg系统调用统称为读系统调用，从网络连接上接收数据。同recvmsg相比，前三个系统调用比较简单。 recvmsg因为比较通用而复杂得多。图16-27给出了这四个系统调用和一个库函数 (recv)的特点。

函 数	描述符类型	缓存数量	返回发送者的地址吗?	标 志?	返回控制信息?
read	任何类型	1			
readv	任何类型	[1..UIO_MAXIOV]			
recv	插口	1		•	
recvfrom	插口	1	•	•	
recvmsg	插口	[1..UIO-MAXIOV]	•	•	•

图16-27 读系统调用

在Net/3中，recv是一个库函数，通过调用 recvfrom来实现的。为了同以前编

译的程序二进制兼容，内核将旧的recv系统调用映射到函数orecv。我们仅仅讨论recvfrom的内核实现。

只有read和readv系统调用适用于各类描述符，其他的调用只适用于插口描述符。

同写调用一样，通过iovec结构数组来指定多个缓存。对数据报协议，recvfrom和recvmsg返回每一个收到的数据报的源地址。对于面向连接的协议，getpeername返回连接对方的地址。与接收调用相关的标志参考第16.11节。

同写调用一样，读调用利用一个公共函数soreceive来做所有工作。图16-28说明读系统调用的流程。

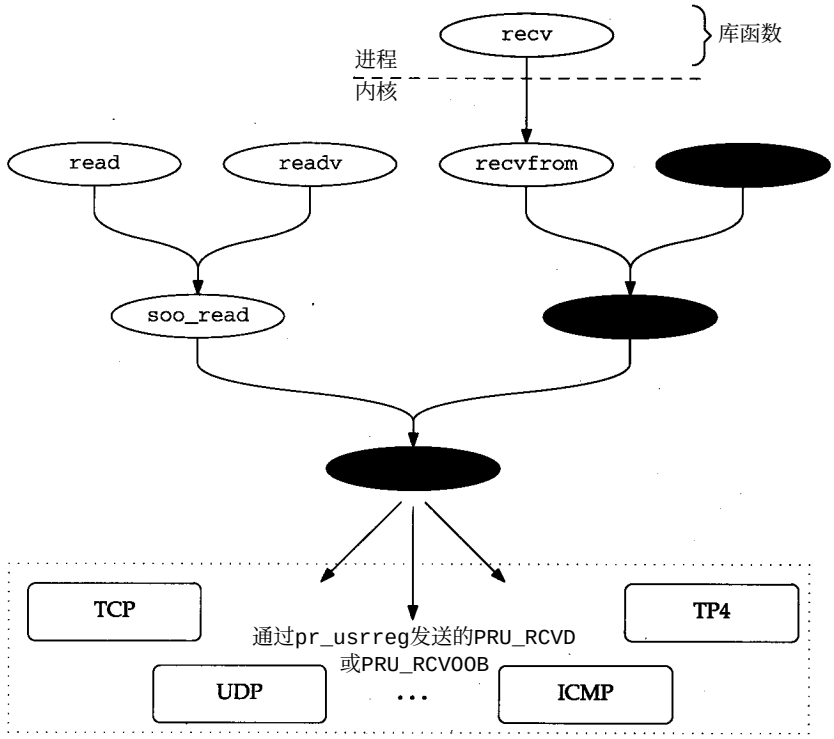


图16-28 所有插口输入都由soreceive处理

我们仅仅讨论图16-28中的带阴影的函数。其余的函数读者可以自己查阅有关资料。

16.9 recvmsg系统调用

recvmsg函数是最通用的读系统调用。如果一个进程使用任何一个其他的读系统调用，且地址、控制信息和接收标志的值还未定，则系统可能在没有任何通知的情况下丢弃它们。图16-29显示了recvmsg函数。

433-445 recvmsg的三个参数是：插口描述符；类型为msghdr的结构指针，几个控制标志。

1. 复制iov数组

446-461 同sendmsg一样，recvmsg将msghdr结构复制到内核，如果自动分配的数组

aiov太小，则分配一个更大的iovec数组，并且将数组单元从进程复制到由iov指向的内核数组(第16.4节)。将第三个参数复制到msghdr结构中。

uipc_syscalls.c

```
433 struct recvmsg_args {
434     int     s;
435     struct msghdr *msg;
436     int     flags;
437 };

438 recvmsg(p, uap, retval)
439 struct proc *p;
440 struct recvmsg_args *uap;
441 int    *retval;
442 {
443     struct msghdr msg;
444     struct iovec aiov[UIO_SMALLIOV], *uiov, *iov;
445     int     error;

446     if (error = copyin((caddr_t) uap->msg, (caddr_t) & msg, sizeof(msg)))
447         return (error);
448     if ((u_int) msg.msg_iovlen >= UIO_SMALLIOV) {
449         if ((u_int) msg.msg_iovlen >= UIO_MAXIOV)
450             return (EMSGSIZE);
451         MALLOC(iov, struct iovec *,
452                sizeof(struct iovec) * (u_int) msg.msg_iovlen, M_IOV,
453                M_WAITOK);
454     } else
455         iov = aiov;
456     msg.msg_flags = uap->flags;
457     uiov = msg.msg_iov;
458     msg.msg_iov = iov;
459     if (error = copyin((caddr_t) uiov, (caddr_t) iov,
460                        (unsigned) (msg.msg_iovlen * sizeof(struct iovec))))
461         goto done;
462     if ((error = recvit(p, uap->s, &msg, (caddr_t) 0, retval)) == 0) {
463         msg.msg_iov = uiov;
464         error = copyout((caddr_t) & msg, (caddr_t) uap->msg, sizeof(msg));
465     }
466   done:
467     if (iov != aiov)
468         FREE(iov, M_IOV);
469     return (error);
470 }
```

uipc_syscalls.c

图16-29 recvmsg 系统调用

2. recvit和释放缓存

462-470 recvit收完数据后，将更新过的缓存长度和标志的msghdr结构再复制到进程。如果分配了一个更大的iovec结构，则返回之前释放它。

16.10 recvit函数

recvit函数被recv、recvfrom和recvmsg调用，如图16-30所示。基于recv *xxx*调用提供的msghdr结构，recvit函数为soreceive的处理准备了一个uio结构。

471-500 getsock为描述符s返回一个file结构，然后recvit初始化uio结构，该结构描述从内核到进程之间的一次数据传送。通过对 iovec数组中的msg_iovlen字段求和得到

传送的字节数。结果保留在uio_resid中的len中。

uipc_syscalls.c

```
471 recvit(p, s, mp, namelenp, retsize)
472 struct proc *p;
473 int     s;
474 struct msghdr *mp;
475 caddr_t namelenp;
476 int    *retsize;
477 {
478     struct file *fp;
479     struct uio auio;
480     struct iovec *iov;
481     int     i;
482     int     len, error;
483     struct mbuf *from = 0, *control = 0;
484     if (error = getsock(p->p_fd, s, &fp))
485         return (error);
486     auio.uio_iov = mp->msg_iov;
487     auio.uio_iovcnt = mp->msg_iovlen;
488     auio.uio_segflg = UIO_USERSPACE;
489     auio.uio_rw = UIO_READ;
490     auio.uio_procp = p;
491     auio.uio_offset = 0;        /* XXX */
492     auio.uio_resid = 0;
493     iov = mp->msg_iov;
494     for (i = 0; i < mp->msg_iovlen; i++, iov++) {
495         if (iov->iov_len < 0)
496             return (EINVAL);
497         if ((auio.uio_resid += iov->iov_len) < 0)
498             return (EINVAL);
499     }
500     len = auio.uio_resid;
```

uipc_syscalls.c

图16-30 recvit 函数：初始化uio结构

recvit的第二部分调用soreceive，并且将结果复制到进程，如图16-31所示。

uipc_syscalls.c

```
501     if (error = soreceive((struct socket *) fp->f_data, &from, &auio,
502       (struct mbuf **) 0, mp->msg_control ? &control : (struct mbuf **) 0,
503                             &mp->msg_flags)) {
504         if (auio.uio_resid != len && (error == ERESTART ||
505                                       error == EINTR || error == EWOULDBLOCK))
506             error = 0;
507     }
508     if (error)
509         goto out;
510     *retsize = len - auio.uio_resid;
511     if (mp->msg_name) {
512         len = mp->msg_namelen;
513         if (len <= 0 || from == 0)
514             len = 0;
515         else {
516             if (len > from->m_len)
517                 len = from->m_len;
518             /* else if len < from->m_len ??? */
519             if (error = copyout(mtod(from, caddr_t),
```

图16-31 recvit 函数：返回结果

```
520                                     (caddr_t) mp->msg_name, (unsigned) len))
521                 goto out;
522         }
523         mp->msg_namelen = len;
524         if (namelenp &&
525             (error = copyout((caddr_t) & len, namelenp, sizeof(int)))) {
526             goto out;
527         }
528     }
529     if (mp->msg_control) {
530         len = mp->msg_controllen;
531         if (len <= 0 || control == 0)
532             len = 0;
533         else {
534             if (len >= control->m_len)
535                 len = control->m_len;
536             else
537                 mp->msg_flags |= MSG_CTRUNC;
538             error = copyout((caddr_t) mtod(control, caddr_t),
539                             (caddr_t) mp->msg_control, (unsigned) len);
540         }
541         mp->msg_controllen = len;
542     }
543   out:
544     if (from)
545         m_freem(from);
546     if (control)
547         m_freem(control);
548     return (error);
549 }
```

uipc_syscalls.c

图16-31 (续)

1. 调用soreceive

501-510 soreceive实现从插口缓存中接收数据的最复杂的功能。传送的字节数保存在*retsize中，并且返回给进程。如果有些数据已经被复制到进程后信号出现或阻塞出现(len不等于uio_resid)，则忽略差错，并返回已经传送的字节。

2. 将地址和控制信息复制到进程

511-542 如果进程传入了一个存放地址或控制信息或两者都有的缓存，则 recvit将结果写入该缓存，并且根据soreceive返回的结果调整它们的长度。如果缓存太小，则地址信息可能被截掉。如果进程在发送读调用之前保留缓存的长度，将该长度同内核返回的namelenp变量(或sockaddr结构的长度域)相比较就可以发现这个差错。通过设置msg_flags中的MSG_CTRUNC标志来报告这种差错，参考习题16-7。

3. 释放缓存

543-549 从out开始，释放存储源地址和控制信息的mbuf缓存。

16.11 soreceive函数

soreceive函数将数据从插口的接收缓存传送到进程指定的缓存。某些协议还提供发送者的地址，地址可以同可能的附加控制信息一起返回。在讨论它的代码之前，先来讨论接收操作，带外数据和插口接收缓存的组织的含义。

图16-32 列出了在执行soreceive期间内核知道的一些标志。

flags	描 述	参 考
MSG_DONTWAIT	在调用期间不等待资源	图16-38
MSG_OOB	接收带外数据而不是正常的数据	图16-39
MSG_PEEK	接收数据的副本而不取走数据	图16-43
MSG_WAITALL	在返回之前等待数据写缓存	图16-50

图16-32 recv *xxx*系统调用：传递给内核的标志值

recvmsg是唯一返回标志字段给进程的读系统调用。在其他的系统调用中，控制返回给进程之前，这些信息被内核丢弃。图16-33列出了在msghdr中recvmsg能设置的标志。

msg_flags	描 述	参 考
MSG_CTRUNC	控制信息的长度大于提供的缓存长度	图16-31
MSG_EOR	收到的数据标志一个逻辑记录的结束	图16-48
MSG_OOB	缓存中包含带外数据	图16-45
MSG_TRUNC	收到的报文的长度大于提供的缓存长度	图16-51

图16-33 recvmsg 系统调用：内核返回的msg_flag 值

16.11.1 带外数据

带外数据(OOB)在不同的协议中有不同的含义。一般来说，协议利用已建立的通信连接来发送OOB数据。OOB数据可能与已发送的正常数据同序。插口层支持两种与协议无关的机制来实现对OOB数据的处理：标记和同步。本章讨论插口层实现的抽象的 OOB机制。UDP不支持OOB数据。TCP的紧急数据机制与插口层的OOB数据之间的关系在TCP一章中描述。

发送进程通过在send*xxx*调用中设置MSG_OOB标志将数据标记为OOB数据。sosend将这个信息传递给插口协议，插口层收到这个信息后，对数据进行特殊处理，如加快发送数据或使用另一种排队策略。

当一个协议收到OOB数据后，并不将它放进插口的接收缓存而是放在其他地方。进程通过设置recv*xxx*调用中的MSG_OOB标志来接收到达的 OOB数据。另一种方法是，通过设置SO_OOBINLINE插口选项(见第17.3节)，接收进程可以要求协议将OOB数据放在正常的数据之内。当SO_OOBINLINE被设置时，协议将收到的OOB数据放进正常数据的接收缓存。在这种情况下，MSG_OOB不用来接收OOB数据。读调用要么返回所有的正常数据，要么返回所有的OOB数据。两种类型的数据从来不会在一个输入调用的输入缓存中混淆。进程使用 recvmsg来接收数据时，可以通过检查MSG_OOB标志来决定返回的数据是正常数据还是OOB数据。

插口层支持OOB数据和正常数据的同步接收，采用的方法是允许协议在正常数据流中标记OOB数据起始点。接收者可以在每一个读系统调用的后面，通过 SIOCATMARKioctl命令来检查是否已经达到 OOB数据的起始点。当接收正常的数据时，插口层确保在一个报文中只有在标记前的正常数据才会收到，使得接收者接收的数据不会超过标记。如果在接收者到达标记之前收到一些附加的OOB数据，标记就自动向前移。

16.11.2 举例

图16-34说明两种接收带外数据的方法。在两个例了中，字节 A~I作为正常数据接收，字

节J作为带外数据接收，字节K~L作为正常数据接收。接收进程已经接收了A之前(不包括A)的所有数据。

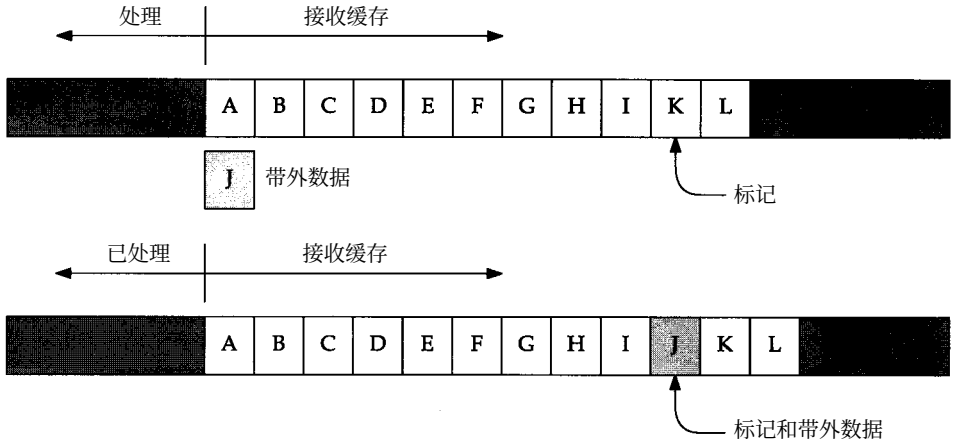


图16-34 接收带外数据

在第一个例子中，进程能够正确读出字节A~I，或者如果设置MSG_OOB，也能读出字节J。即使读请求的长度大于9个字节(A~I)，插口层也只返回9个字节，以免超过带外数据的同步标记。当读出字节I后，SIOCATMARK为真；对于到达带外数据标记的进程，不必读出字节J。

在第二个例了中，在SIOCATMARK为真时只能读字节A~I。第二次调用读字节J~L。

在图16-34中，字节J不是TCP的紧急数据指针指示的字节。在本例中，紧急指针指向的是字节K。有关细节请参考第29.7节。

16.11.3 其他的接收操作选项

进程能够通过设置标志MSG_PEEK来查看是否有数据到达。而数据仍然留在接收队列中，被下一个不设置MSG_PEEK的读调用读出。

标志MSG_WAITALL指示读调用只有在读到指定数量的数据后才返回。即使soreceive中有一些数据可以返回给进程，但它仍然要等到收到剩余的数据后才返回。

当标志MSG_WAITALL被设置后，soreceive只有在下列情况下可以在没有读完指定长度的数据时返回：

- 连接的读通道被关闭；
- 插口的接收缓存小于所读数据的大小；
- 在进程等待剩余的数据时差错出现；
- 带外数据到达；或
- 在读缓存被写满之前，一个逻辑记录的结尾出现。

NFS是Net/3中唯一使用MSG_WAITALL和MSG_DONTWAIT标志的软件。进程可以不通过ioctl或fcntl来选择非阻塞的I/O操作而是设置MSG_DONTWAIT标志来实现非阻塞的读系统调用。

16.11.4 接收缓存的组织：报文边界

对于支持报文边界的协议，每一个报文存放在一个 mbuf链中。接收缓存中的多个报文通

过m_nextpkt指针链接成一个mbuf队列(图2-21)。协议处理层加数据到接收队列，插口层从接收队列中移走数据。接收缓存的高水位标记限制了存储在缓存中的数据量。

如果PR_ATOMIC没有被置位，协议层尽可能多地在缓存中存放数据，丢弃输入数据中的不合要求的部分。对于 TCP，这就意味着到达的任何数据如果在接收窗口之外都将被丢弃。如果PR_ATOMIC被置位，缓存必须能够容纳整个报文，否则协议层将丢弃整个报文。对于UDP而言，如果接收缓存已满，则进入的数据报都将被丢弃，缓存满的原因可能是进程读数据报的速度不够快。

PR_ADDR被置位的协议使用sbappendaddr构造一个mbuf链，并将其加入到接收队列。缓存链包含一个存放报文源地址的 mbuf，0个或更多的控制mbuf，后面跟着0个或更多的包含数据的mbuf。

对于SOCK_SEQPACKET和SOCK_RDM协议，它们为每一个记录建立一个 mbuf链。如果PR_ATOMIC被置位，则调用 sbappendrecord，将记录加到接收缓存的尾部。如果PR_ATOMIC没有被置位(OSI的TP4)，则用sbappendrecord产生一个新的记录，其余的数据用sbappend加到这个记录中。

> 假定PR_ATOMIC就是表示缓存的组织结构是不正确的。例如， TP4中并没有PR_ATOMIC，而是用M_EOR标志来支持记录边界。

图16-35说明了由三个mbuf链(即三个数据报)组成的UDP接收缓存的结构。每一个 mbuf中都标有m_type的值。

在图16-35中，第三个数据报中有一些控制信息。三个 UDP插口选项能够导致控制信息被存入接收缓存。详细情况参考图 22-5和图23-7。

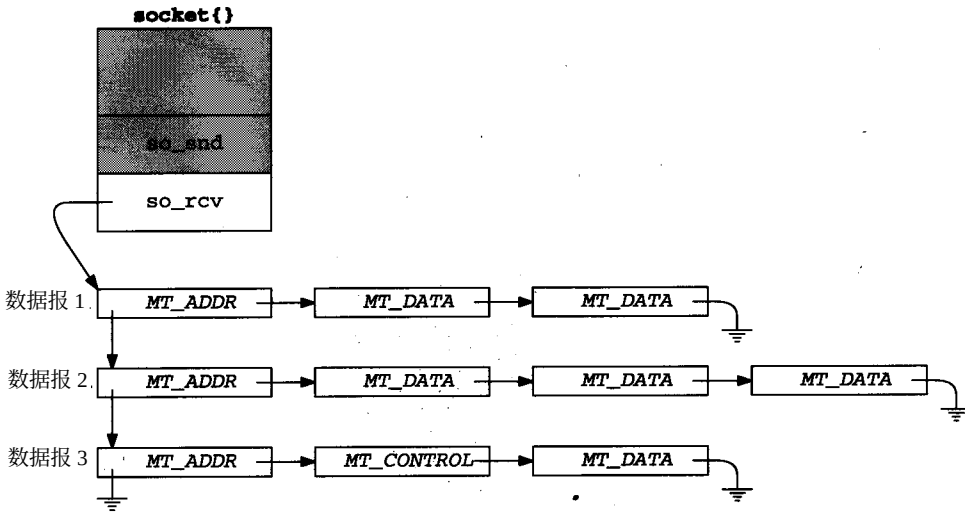


图16-35 包含三个数据报的UDP接收缓存

对于PR_ATOMIC协议，当收到数据时， sb_lowat被忽略。当没有设置PR_ATOMIC时，sb_lowat的值等于读系统调用返回的最小的字节数。但也有一些例外，如图 16-41所示。

16.11.5 接收缓存的组织：没有报文边界

当协议不需维护报文边界 (即SOCK_STREAM协议，如TCP)时，通过sbappend将进入的

数据加到缓存中的最后一个 mbuf链的尾部。如果进入的数据长度大于缓存的长度，则数据将被截掉，sb_lowat为一个读系统调用返回的字节数设置了一个下限。

图16-36说明了仅仅包含正常数据的TCP接收缓存的结构。

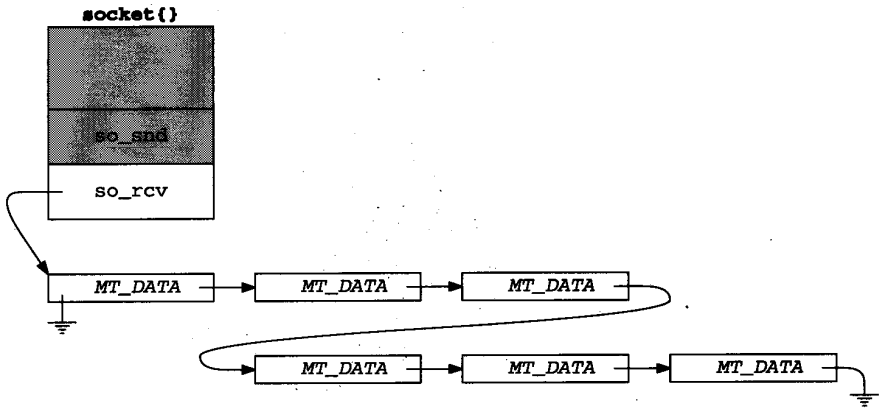


图16-36 TCP的so_rcv 缓存

16.11.6 控制信息和带外数据

不像TCP，一些流协议支持控制信息，并且调用 sbappendcontrol将控制信息和相关数据作为一个新的mbuf链加入接收缓存。如果协议支持内含OOB数据，则调用sbinsertoob插入一个新的mbuf链到任何包含OOB数据的mbuf链之后，但在任何包含正常数据的mbuf链之前。这一点确保进入的OOB数据总是排在正常数据之前。

图16-37说明包含控制信息和OOB数据的接收缓存的结构。

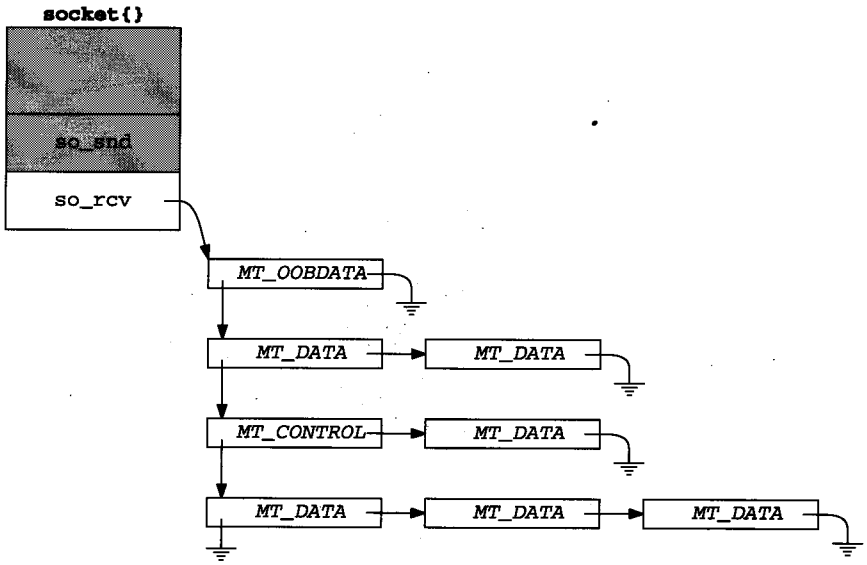


图16-37 带有控制信息和OOB数据的so_rcv 缓存

Unix域流协议支持控制信息，OSI TP4协议支持MT_OOBDATA mbuf。TCP既不支持控制信息，也不支持MT_OOBDATA形式的带外数据。如果TCP的紧急指针指向的字节存储在数据内(SO_OOBINLINE被设置)，那么该字节是正常数据而不是 OOB数据。TCP对紧急指针和相

关数据的处理在第29.7节中讨论。

16.12 soreceive代码

我们现在有足够的背景信息来详细讨论 soreceive函数。在接收数据时， soreceive必须检查报文边界，处理地址和控制信息以及读标志所指定的任何特殊操作 (图16-32)。一般来说， soreceive的一次调用只处理一个记录，并且尽可能返回要求读的字节数。图 16-38显示了soreceive函数的大概情况。

uipc_socket.c

```
439 soreceive(so, paddr, uio, mp0, controlp, flagsp)
440 struct socket *so;
441 struct mbuf **paddr;
442 struct uio *uio;
443 struct mbuf **mp0;
444 struct mbuf **controlp;
445 int    *flagsp;
446 {
447     struct mbuf *m, **mp;
448     int     flags, len, error, s, offset;
449     struct protosw *pr = so->so_proto;
450     struct mbuf *nextrecord;
451     int     moff, type;
452     int     orig_resid = uio->uio_resid;

453     mp = mp0;
454     if (paddr)
455         *paddr = 0;
456     if (controlp)
457         *controlp = 0;
458     if (flagsp)
459         flags = *flagsp & ~MSG_EOR;
460     else
461         flags = 0;

                    /* MSG_OOB processing and */
                /* implicit connection confirmation */

483   restart:
484     if (error = sblock(&so->so_rcv, SBLOCKWAIT(flags)))
485         return (error);
486     s = splnet();
487     m = so->so_rcv.sb_mb;

                /* if necessary, wait for data to arrive */

542   dontblock:
543     if (uio->uio_procp)
544         uio->uio_procp->p_stats->p_ru.ru_msgrcv++;
545     nextrecord = m->m_nextpkt;

                /* process address and control information */

591     if (m) {
```

图16-38 soreceive 函数：概述

```
592             if ((flags & MSG_PEEK) == 0)
593                 m->m_nextpkt = nextrecord;
594             type = m->m_type;
595             if (type == MT_OOBDATA)
596                 flags |= MSG_OOB;
597         }
                                    /* process data */
693         } while (m && pr->pr_flags & PR_ATOMIC && error == 0 && uio->uio_resid > 0);
                                    /* cleanup */
715     release:
716         sbunlock(&so->so_rcv);
717         splx(s);
718         return (error);
719 }
```

uipc_socket.c

图16-38 (续)

439-446 soreceive有六个参数。指向插口的so指针。指向存放接收地址信息的mbuf缓存的指针*paddr。如果mp0指向一个mbuf链，则soreceive将接收缓存中的数据传送到*mp0指向的mbuf缓存链。在这种情况下，uio结构中只有用来记数的uio_resid字段是有意义的。如果mp0为空，则soreceive将数据传送到uio结构中指定的缓存。*controlp指向包含控制信息的mbuf缓存。soreceive将图16-33中描述的标志存放在*flagsp。

447-453 soreceive一开始将pr指向插口协议的交换结构，并将uio_resid(接收请求的大小)保存在orig_resid。如果将控制或地址信息从内核复制到进程，则将orig_resid清0。如果复制的是数据，则更新uio_resid。不管哪一种情况，orig_resid都不可能等于uio_resid。soreceive函数的最后处理要利用这一事实(图16-51)。

454-461 在这一段代码中，首先将*paddr和*controlp置空。在将MSG_EOR标志清0后，将传给soreceive的*flagsp的值保存在flags中(习题16.8)。flagsp是一个用来返回结果的参数，但是只有recvmsg系统调用才能收到结果。如果flagsp为空，则将flags清0。

483-487 在访问接收缓存之前，调用sblock给缓存加锁。如果flags中没有设置MSG_DONTWAIT标志，则soreceive必须等待加锁成功。

支持在内核中从NFS发调用到插口层带来了另一个负作用。

挂起协议处理，使得在检查缓存过程中soreceive不被中断。m是接收缓存中的第一个mbuf链上的第一个mbuf。

1. 如果需要，等待数据

488-541 soreceive要检查几种情况，并且如果需要，它可能要等待接收更多的数据才继续往下执行。如果soreceive在这里进入睡眠状态，则在它醒来后跳转到restart查看是否有足够的数据到达。这个过程一直继续，直到收到足够的数据为止。

542-545 当soreceive已收到足够的数据来满足读请求所要求的数据量时，就跳转到dontblock。并将指向接收缓存中的第二个mbuf链的指针保存在nextrecord中。

2. 处理地址和控制信息

542-545 在传送数据之前，首先处理地址信息和控制信息。

3. 建立数据传送

591-597 因为只有OOB数据或正常数据是在一次soreceive调用中传送，这段代码的功能就是记住队列前端的数据的类型，这样在类型改变时，soreceive能够停止传送。

4. 传送数据循环

598-692 只要缓存中还有mbuf (m不空)，请求的数据还没有传送完毕(uio_resid>0)，且没有差错出现，本循环就不会退出。

退出处理

693-719 剩余的代码主要是更新指针、标志和偏移；释放插口缓存锁；使能协议处理并返回。

图16-39说明soreceive对OOB数据的处理。

uipc_socket.c

```
462     if (flags & MSG_OOB) {
463         m = m_get(M_WAIT, MT_DATA);
464         error = (*pr->pr_usrreq) (so, PRU_RCVOOB,
465                 m, (struct mbuf *) (flags & MSG_PEEK), (struct mbuf *) 0);
466         if (error)
467             goto bad;
468         do {
469             error = uiomove(mtod(m, caddr_t),
470                             (int) min(uio->uio_resid, m->m_len), uio);
471             m = m_free(m);
472         } while (uio->uio_resid && error == 0 && m);
473       bad:
474         if (m)
475             m_freem(m);
476         return (error);
477     }
```

uipc_socket.c

图16-39 soreceive 函数：带外数据

5. 接收OOB数据

462-477 因为OOB数据不存放在接收缓存中，所以soreceive为其分配一块标准的mbuf，并给协议发送PRU_RCVOOB请求。while循环将协议返回的数据复制到uio指定的缓存中。复制完成后，soreceive返回0或差错代码。

对于PRU_RCVOOB请求，UDP协议总是返回EOPNOTSUPP。关于TCP的紧急数据的处理的详细情况参考第30.2节。图16-40说明soreceive对连接信息的处理。

uipc_socket.c

```
478     if (mp)
479         *mp = (struct mbuf *) 0;
480     if (so->so_state & SS_ISCONFIRMING && uio->uio_resid)
481         (*pr->pr_usrreq) (so, PRU_RCVD, (struct mbuf *) 0,
482                           (struct mbuf *) 0, (struct mbuf *) 0);
```

uipc_socket.c

图16-40 soreceive 函数：连接信息

6. 连接证实

478-482 如果返回的数据存放在 mbuf链中，则将 *mp初始化成空。如果插口处于

SO_ISCONFIRMING状态，PRU_RCVD请求告知协议进程想要接收数据。

SO_ISCONFIRMING状态仅用于OSI的流协议，TP4。在TP4中，直到一个用户级进程通过发送或接收数据的方式来证实连接，该连接才被认为已完全建立。在通过调用getpeername来获取对方的身份后，进程可能调用 shutdown或close来拒绝连接。

图16-38显示了图16-41中的代码在检查接收缓存时，接收缓存被加锁。 soreceive的这部分代码的功能是查看接收缓存中的数据是否能满足读系统调用的要求。

uipc_socket.c

```
488     /*
489      * If we have less data than requested, block awaiting more
490      * (subject to any timeout) if:
491      *   1. the current count is less than the low water mark, or
492      *   2. MSG_WAITALL is set, and it is possible to do the entire
493      *  receive operation at once if we block (resid <= hiwat).
494      *   3. MSG_DONTWAIT is not set
495      *
496      * If MSG_WAITALL is set but resid is larger than the receive buffer,
497      * we have to do the receive in sections, and thus risk returning
498      * a short count if a timeout or signal occurs after we start.
499      */
500     if (m == 0 || ((flags & MSG_DONTWAIT) == 0 &&
501                    so->so_rcv.sb_cc < uio->uio_resid) &&
502         (so->so_rcv.sb_cc < so->so_rcv.sb_lowat ||
503       ((flags & MSG_WAITALL) && uio->uio_resid <= so->so_rcv.sb_hiwat)) &&
504         m->m_nextpkt == 0 && (pr->pr_flags & PR_ATOMIC) == 0) {
```

uipc_socket.c

图16-41 soreceive 函数：数据够吗？

7. 读调用的请求能满足吗？

488-504 一般情况下，soreceive要等待直到接收缓存中有足够的数据来满足整个读请求。但是，有几种情况可能导致差错或返回比读请求要求少的数据。

- 接收缓存没有数据(m等于0)。
- 缓存中的数据不能满足读请求要求的数量 (sb_cc<uio_resid)并且没有设置MSG_DONTWAIT标志，最少的数据也得不到(sb_cc<sb_lowat)，且当该链到达时更多的数据能够加到链的后面(m_nextpkt等于0，且没有设置PR_ATOMIC)。
- 缓存中的数据不能满足读请求要求的数量，能得到最少的数据量，数据能够加到链中来，但是MSG_WAITALL指示soreceive必须等待直到缓存中的数据能满足读请求。

如果最后一种情况的条件能够满足，但是因为读请求的数据太大以至如果不阻塞等待就不能满足(uio_resid≤sb_hiwat)，soreceive就不等待而是继续往下执行。

如果接收缓存有数据，并且设置了MSG_DONTWAIT，则sorecevie不等待更多的数据。

有几种原因使得等待更多的数据是不合适的。在图 16-42中，soreceive要么检查三种情况，然后返回；要么等待更多的数据到达。

8. 等待更多的数据吗？

505-534 在此处，soreceive已经决定等待更多的数据来满足读请求。在等待之前，它需要检查以下几种情况：

505-512 • 如果插口处于差错状态，且缓存为空(m为空)，则soreceive返回差错代码。如

果有差错，但是接收缓存中有数据 (m非空)，则返回缓存的数据；当下一个读调用来时，如果没有数据，就返回差错。如果设置了 MSG_PEEK，就不清除差错，因为设置了 MSG_PEEK的读调用不能改变插口的状态。

513-518 • 如果连接的读通道已经被关闭并且数据仍在接收缓存中，则 sosend不等待而是将数据返回给进程 (在dontblock的情况下)。如果接收缓存为空，则 soreceive跳转到release，读系统调用返回0，表示连接的读通道已经被关闭。

519-523 • 如果接收缓存中包含带外数据或出现一个逻辑记录的结尾，则 soreceive不等待，而是跳转到dontblock。

524-528 • 如果协议请求中的连接不存在，则设置差错代码为 ENOTCONN，函数跳转到release。

529-534 • 如果读请求读0字节或插口是非阻塞的，则函数跳转到 release，并返回0或EWOULDBLOCK(后一种情况)。

uipc_socket.c

```
505         if (so->so_error) {
506             if (m)
507                 goto dontblock;
508             error = so->so_error;
509             if ((flags & MSG_PEEK) == 0)
510                 so->so_error = 0;
511             goto release;
512         }
513         if (so->so_state & SS_CANTRCVMORE) {
514             if (m)
515                 goto dontblock;
516             else
517                 goto release;
518         }
519         for (; m; m = m->m_next)
520             if (m->m_type == MT_OOBDATA || (m->m_flags & M_EOR)) {
521                 m = so->so_rcv.sb_mb;
522                 goto dontblock;
523             }
524         if ((so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING)) == 0 &&
525             (so->so_proto->pr_flags & PR_CONNREQUIRED)) {
526             error = ENOTCONN;
527             goto release;
528         }
529         if (uio->uio_resid == 0)
530             goto release;
531         if ((so->so_state & SS_NBIO) || (flags & MSG_DONTWAIT)) {
532             error = EWOULDBLOCK;
533             goto release;
534         }
535         sbunlock(&so->so_rcv);
536         error = sbwait(&so->so_rcv);
537         splx(s);
538         if (error)
539             return (error);
540         goto restart;
541     }
```

uipc_socket.c

图16-42 soreceive 函数：等待更多的数据吗？

9. 是，等待更多的数据

535-541 此处soreceive已决定等待更多的数据，并且有理由这么做 (即，将有数据到达)。在进程调用sbwait进入睡眠期间，缓存被解锁。如果因为差错或信号出现使得 sbwait返回，则soreceive返回相应的差错；否则 soreceive跳转到restart，查看接收缓存中的数据是否能够满足读请求。

同sosend中一样，进程能够利用 SO_RCVTIMEO插口选项为sbwait设置一个接收定时器。如果在数据到达之前定时器超时，则 sbwait返回EWOULDBLOCK。

定时器并不能总令人满意。因为当插口上有活动时，定时器每次都被重置。如果在一个超时间隔内至少有一个字节到达，则定时器从来不会超时，一直到设置了更长的超时值的读系统调用返回。 sb_timeo是一个不活动定时器，并不要求超时值上限，但为了满足读系统调用，超时值的上限可能是必要的。

在此处， soreceive准备从接收缓存中传送数据。图 16-43说明了地址信息的传送。

uipc_socket.c

```
542   dontblock:
543     if (uio->uio_procp)
544         uio->uio_procp->p_stats->p_ru.ru_msgrcv++;
545     nextrecord = m->m_nextpkt;
546     if (pr->pr_flags & PR_ADDR) {
547         orig_resid = 0;
548         if (flags & MSG_PEEK) {
549             if (paddr)
550                 *paddr = m_copy(m, 0, m->m_len);
551             m = m->m_next;
552         } else {
553             sbfree(&so->so_rcv, m);
554             if (paddr) {
555                 *paddr = m;
556                 so->so_rcv.sb_mb = m->m_next;
557                 m->m_next = 0;
558                 m = so->so_rcv.sb_mb;
559             } else {
560                 MFREE(m, so->so_rcv.sb_mb);
561                 m = so->so_rcv.sb_mb;
562             }
563         }
564     }
```

uipc_socket.c

图16-43 soreceive 函数：返回地址信息

10. dontblock

542-545 nextrecord指向接收缓存中的下一条记录。在 soreceive的后面，当第一个链被丢弃后，该指针被用来将剩余的 mbuf放入插口缓存。

11. 返回地址信息

546-564 如果协议提供地址信息，如UDP，则将从mbuf链中删除包含地址的mbuf，并通过*paddr返回。如果paddr为空，则地址被丢弃。

在soreceive中，如果设置了MSG_PEEK，则数据仍留在缓存中。

图16-44中的代码处理缓存中的控制 mbuf。

12. 返回控制信息

565-590 每一个包含控制信息的mbuf都将从缓存中删除(如果设置了MSG_PEEK，则不删除而是复制)，并连到*controlp。如果controlp为空，则丢弃控制信息。

uipc_socket.c

```
565     while (m && m->m_type == MT_CONTROL && error == 0) {
566         if (flags & MSG_PEEK) {
567             if (controlp)
568                 *controlp = m_copy(m, 0, m->m_len);
569             m = m->m_next;
570         } else {
571             sbfree(&so->so_rcv, m);
572             if (controlp) {
573                 if (pr->pr_domain->dom_externalize &&
574                     mtod(m, struct cmsghdr *)->cmsg_type ==
575                     SCM_RIGHTS)
576                         error = (*pr->pr_domain->dom_externalize) (m);
577                 *controlp = m;
578                 so->so_rcv.sb_mb = m->m_next;
579                 m->m_next = 0;
580                 m = so->so_rcv.sb_mb;
581             } else {
582                 MFREE(m, so->so_rcv.sb_mb);
583                 m = so->so_rcv.sb_mb;
584             }
585         }
586         if (controlp) {
587             orig_resid = 0;
588             controlp = &(*controlp)->m_next;
589         }
590     }
```

uipc_socket.c

图16-44 soreceive 函数：处理控制信息

如果进程准备接收控制信息，则协议定义了一个dom_externalize函数，一旦控制信息mbuf中包含SCM_RIGHTS(访问权限)，就调用dom_externalize函数。该函数执行内核中所有接收访问权限的操作。只有Unix域协议支持访问权限，有关细节在第7.3节已讨论过。如果进程不准备接收控制信息(controlp为空)，则丢弃控制mbuf。

直到处理完所有包含控制信息的mbuf或出现差错时，循环才退出。

对于Unix协议域，dom_externalize函数通过修改接收进程的文件描述符表来实现文件描述符的传送。

处理完所有的控制mbuf后，m指向链中的下一个mbuf。如果在地址或控制信息的后面，链中没有其他的mbuf，则m为空。例如，当一个长度为0的数据报进入接收缓存时就会出现这种情况。图16-45说明了soreceive准备从mbuf链中传送数据。

uipc_socket.c

```
591     if (m) {
592         if ((flags & MSG_PEEK) == 0)
593             m->m_nextpkt = nextrecord;
594         type = m->m_type;
595         if (type == MT_OOBDATA)
596             flags |= MSG_OOB;
597     }
```

uipc_socket.c

图16-45 soreceive 函数：准备传送mbuf

13. 准备传送数据

591-597 处理完控制mbuf后，链中应该只剩下正常数据、带外数据 mbuf或没有任何mbuf。如果m为空，则soreceive完成处理，控制跳到while循环的底部。如果m不空，所有剩余的mbuf链(nextrecord)都将重新连接到m，并将下一个mbuf的类型赋给type。如果下一个mbuf包含OOB数据，则设置flags中的MSG_OOB标志，并在最后返回给进程。因为 TCP不支持MT_OOBDATA形式的带外数据，所以MSG_OOB不会返回给TCP插口上的读调用。

图16-47显示了传送mbuf循环的第一部分。图16-46列出了循环中更新的变量。

变 量	描 述
moff	当MSG_PEEK被置位时，将被传送的下一个字节的偏移位置
offset	当MSG_PEEK被置位时，OOB标记的偏移位置
uio_resid	还未传送的字节数
len	从本mbuf中将要传送的字节数；如果uio_resid比较小或靠OOB标记比较近，则len可能小于m_len。

图16-46 soreceive 函数：循环内的变量

uipc_socket.c

```
598     moff = 0;
599     offset = 0;
600     while (m && uio->uio_resid > 0 && error == 0) {
601         if (m->m_type == MT_OOBDATA) {
602             if (type != MT_OOBDATA)
603                 break;
604         } else if (type == MT_OOBDATA)
605             break;
606         so->so_state &= ~SS_RCVATMARK;
607         len = uio->uio_resid;
608         if (so->so_oobmark && len > so->so_oobmark - offset)
609             len = so->so_oobmark - offset;
610         if (len > m->m_len - moff)
611             len = m->m_len - moff;
612         /*
613          * If mp is set, just pass back the mbufs.
614          * Otherwise copy them out via the uio, then free.
615          * Sockbuf must be consistent here (points to current mbuf,
616          * it points to next record) when we drop priority;
617          * we must note any additions to the sockbuf when we
618          * block interrupts again.
619          */
620         if (mp == 0) {
621             splx(s);
622             error = uiomove(mtod(m, caddr_t) + moff, (int) len, uio);
623             s = splnet();
624         } else
625             uio->uio_resid -= len;
```

uipc_socket.c

图16-47 soreceive 函数：uiomove

598-600 while循环的每一次循环中，一个 mbuf中的数据被传送到输出链或 uio缓存中。一旦链中没有mbuf或进程的缓存已满或出现差错，就退出循环。

14. 检查OOB和正常数据之前的变换

600-605 如果在处理mbuf链的过程中，mbuf的类型发生变化，则立即停止传送，以确保正常数据和带外数据不会混合在一个返回的报文中。但是，这种检查不适用于 TCP。

15. 更新OOB标记

606-611 计算当前字节到oobmark之间的长度来限制传送的大小，所以oobmark的前一个字节为传送的最后一个字节。传送的大小同时还要受 mbuf大小的限制。这段代码同样适用于TCP。

612-625 如果将数据传送到uio缓存，则调用uiomove。如果数据是作为一个mbuf链返回的，则更新uio_resid的值，使其等于传送的字节数。

为了避免在传送数据过程中协议处理挂起的时间太长，在调用 uiomove过程中使能协议处理。所以，在uiomove运行的过程中，接收缓存中可能会出现新的数据。

图16-48中描述的代码说明调整指针和偏移准备传送下一个 mbuf。

uipc_socket.c

```
626         if (len == m->m_len - moff) {
627             if (m->m_flags & M_EOR)
628                 flags |= MSG_EOR;
629             if (flags & MSG_PEEK) {
630                 m = m->m_next;
631                 moff = 0;
632             } else {
633                 nextrecord = m->m_nextpkt;
634                 sbfree(&so->so_rcv, m);
635                 if (mp) {
636                     *mp = m;
637                     mp = &m->m_next;
638                     so->so_rcv.sb_mb = m = m->m_next;
639                     *mp = (struct mbuf *) 0;
640                 } else {
641                     MFREE(m, so->so_rcv.sb_mb);
642                     m = so->so_rcv.sb_mb;
643                 }
644                 if (m)
645                     m->m_nextpkt = nextrecord;
646             }
647         } else {
648             if (flags & MSG_PEEK)
649                 moff += len;
650             else {
651                 if (mp)
652                     *mp = m_copym(m, 0, len, M_WAIT);
653                 m->m_data += len;
654                 m->m_len -= len;
655                 so->so_rcv.sb_cc -= len;
656             }
657         }
```

uipc_socket.c

图16-48 soreceive 函数：更新缓存

16. mbuf处理完毕了吗

626-646 如果mbuf中的所有字节都已传送完毕，则必须丢弃 mbuf或将指针向前移。如果mbuf中包含了一个逻辑记录的结尾，还应设置 MSG_EOR。如果将 MSG_PEEK置位，则so_receive跳到下一个缓存。在没有将 MSG_PEEK置位的情况下，如果数据已通过uiomove复制完成，则丢弃这块缓存；或者如果数据是作为一个 mbuf链返回，则将缓存添加到mp中。

图16-49包含处理OOB偏移和MSG_EOR的代码段。

uipc_socket.c

```
658         if (so->so_oobmark) {
659             if ((flags & MSG_PEEK) == 0) {
660                 so->so_oobmark -= len;
661                 if (so->so_oobmark == 0) {
662                     so->so_state |= SS_RCVATMARK;
663                     break;
664                 }
665             } else {
666                 offset += len;
667                 if (offset == so->so_oobmark)
668                     break;
669             }
670         }
671         if (flags & MSG_EOR)
672             break;
```

uipc_socket.c

图16-49 soreceive 函数：带外数据标记

17. 更新OOB标记

658-670 如果带外数据标志等于非0，则将其减去已传送的字节数。如果已到达标记处，则将SS_RCVATMARK置位，soreceive跳出while循环。如果没有将MSG_PEEK置位，则更新offset，而不是so_oobmark。

18. 逻辑记录结束

671-672 如果已到达一个逻辑记录的结尾，则 soreceive跳出mbuf处理循环，因而不会将下一个逻辑记录也作为这个报文的一部分返回。

在图16-50中，当设置了MSG_WAITALL标志 ，并且读请求还没有完成，则循环将等待更多的数据到达。

uipc_socket.c

```
673         /*
674          * If the MSG_WAITALL flag is set (for non-atomic socket),
675          * we must not quit until "uio->uio_resid == 0" or an error
676          * termination.  If a signal/timeout occurs, return
677          * with a short count but without error.
678          * Keep sockbuf locked against other readers.
679          */
680         while (flags & MSG_WAITALL && m == 0 && uio->uio_resid > 0 &&
681                !sosendallatonce(so) && !nextrecord) {
682             if (so->so_error || so->so_state & SS_CANTRCVMORE)
683                 break;
684             error = sbwait(&so->so_rcv);
685             if (error) {
686                 sbunlock(&so->so_rcv);
687                 splx(s);
688                 return (0);
689             }
690             if (m = so->so_rcv.sb_mb)
691                 nextrecord = m->m_nextpkt;
692         }
693     }                           /* while more data and more space to fill */
```

uipc_socket.c

图16-50 soreceive 函数：MSG_WAITALL 处理

19. MSG_WAITALL

673-681 如果将MSG_WAITALL置位，而缓存中没有数据(m等于0)，调用者需要更多的数据，sosendallatonce为假，并且这是接收缓存中的最后一个记录（nextrecord为空），则soreceive必须等待新的数据。

20. 差错或没有数据到达

682-683 如果差错出现或连接被关闭，则退出循环。

21. 等待数据到达

684-689 当接收缓存被协议层改变时sbwait返回。如果sbwait是被信号中断(error非0)，则soreceive立即返回。

22. 用接收缓存同步m和nextrecord

690-692 更新m和nextrecord，因为接收缓存被协议层修改了。如果数据到达mbuf，则m等于非0，while循环结束。

23. 处理下一个mbuf

693 本行是mbuf处理循环的结尾。控制返回到循环开始的第600行(图16-47)。一旦接收缓存中有数据，有新的缓存空间，没有差错出现，则循环继续。

如果soreceive停止复制数据，则执行图16-51所示的代码段。

uipc_socket.c

```
694     if (m && pr->pr_flags & PR_ATOMIC) {
695         flags |= MSG_TRUNC;
696         if ((flags & MSG_PEEK) == 0)
697             (void) sbdroprecord(&so->so_rcv);
698     }
699     if ((flags & MSG_PEEK) == 0) {
700         if (m == 0)
701             so->so_rcv.sb_mb = nextrecord;
702         if (pr->pr_flags & PR_WANTRCVD && so->so_pcb)
703             (*pr->pr_usrreq) (so, PRU_RCVD, (struct mbuf *) 0,
704                               (struct mbuf *) flags, (struct mbuf *) 0,
705                               (struct mbuf *) 0);
706     }
707     if (orig_resid == uio->uio_resid && orig_resid &&
708         (flags & MSG_EOR) == 0 && (so->so_state & SS_CANTRCVMORE) == 0) {
709         sbunlock(&so->so_rcv);
710         splx(s);
711         goto restart;
712     }
713     if (flagsp)
714         *flagsp |= flags;
```

uipc_socket.c

图16-51 soreceive 函数：退出处理

24. 被截断的报文

694-698 如果因为进程的接收缓存太小而收到一个被截断的报文（数据报或记录），则插口层将这种情况通过设置MSG_TRUNC来通知进程，报文的被截断部分被丢弃。同其他接收标志一样，进程只有通过recvmsg系统调用才能获得MSG_TRUNC，即使soreceive总是设置这个标志。

25. 记录结尾的处理

699-706 如果没有将MSG_PEEK置位，则下一个mbuf链将被连接到接收缓存，并且如果发送了PRU_RCVD协议请求，则通知协议接收操作已经完成。 TCP通过这种机制来完成对连接

接收窗口的更新。

26. 没有传送数据

707-712 如果soreceive运行完成，没有传送任何数据，没有到达记录的结尾，且连接的读通道是活动的，则将接收缓存解锁，soreceive跳回到restart继续等待数据。

713-714 soreceive中设置的任何标志都在*flagsp中返回，缓存被解锁，soreceive返回。

讨论

soreceive是一个复杂的函数。导致其复杂性的主要原因是繁锁的指针操作及对多种类型的数据(带外数据、地址、控制信息和正常数据)和多目标(进程缓存，mbuf链)的处理。

同sosend类似，soreceive的复杂性是多年积累的结果。为每一种协议编写一个特殊的接收函数将会模糊插口层和协议层之间的边界，但是可以大大简化代码。

[Partridge and Pink 1993]描述了一个专门为UDP编写的soreceive函数，其功能是将数据报从接收缓存复制到进程缓存中时给数据报求检验和。他们给出的结论是：修改通用的soreceive函数来支持这一功能将“使本来已经很复杂的插口子程序变得更加复杂。”

16.13 select系统调用

在下面的讨论中，我们假定读者熟悉 select调用的基本操作和含义。关于 select的应用接口的详细描述参考[Stevens 1992]。

图16-52列出了select能够监控的插口状态。

描 述	select监控的操作		
	读	写	例外
有数据可读	•		
连接的读通道被关闭	•		
listen插口已经将连接排队	•		
插口差错未处理	•		
缓存可供写操作用，且一个连接存在或还没有连接请求		•	
连接的写通道被关闭		•	
插口差错未处理		•	
OOB同步标记未处理			•

图16-52 select 系统调用：插口事件

我们从select系统调用的第一部分开始讨论，如图 16-53所示。

1. 验证和初始化

390-410 在堆栈中分配两个数组：ibits和obits，每个数组有三个单元，每个单元为一个描述符集合。用bzero将它们清0。第一个参数，nd，必须不大于进程的描述符的最大数量。如果nd大于当前分配给进程的描述符个数，将其减少到当前分配给进程的描述符的个数。ni等于用来存放nd个比特(1个描述符占1个比特)的比特掩码所需的字节数。例如，假设最多有256个描述符(FD_SETSIZE)，falset表示一个32 bit的整型(NFDBITS)数组，且nd等于65，那么：

$$ni=howmany(65,32)\times 4=3\times 4=12$$

在上面的公式中，howmany (x，y)返回存储x比特所需要的长度为y比特的对象的数量。

sys_generic.c

```
390 struct select_args {
391     u_int   nd;
392     fd_set *in, *ou, *ex;
393     struct timeval *tv;
394 };

395 select(p, uap, retval)
396 struct proc *p;
397 struct select_args *uap;
398 int    *retval;
399 {
400     fd_set  ibits[3], obits[3];
401     struct timeval atv;
402     int     s, ncoll, error = 0, timo;
403     u_int   ni;

404     bzero((caddr_t) ibits, sizeof(ibits));
405     bzero((caddr_t) obits, sizeof(obits));
406     if (uap->nd > FD_SETSIZE)
407         return (EINVAL);
408     if (uap->nd > p->p_fd->fd_nfiles)
409         uap->nd = p->p_fd->fd_nfiles;   /* forgiving; slightly wrong */
410     ni = howmany(uap->nd, NFDBITS) * sizeof(fd_mask);

411 #define getbits(name, x) \
412     if (uap->name && \
413         (error = copyin((caddr_t)uap->name, (caddr_t)&ibits[x], ni))) \
414         goto done;
415     getbits(in, 0);
416     getbits(ou, 1);
417     getbits(ex, 2);
418 #undef  getbits

419     if (uap->tv) {
420         error = copyin((caddr_t) uap->tv, (caddr_t) & atv,
421                        sizeof(atv));
422         if (error)
423             goto done;
424         if (itimerfix(&atv)) {
425             error = EINVAL;
426             goto done;
427         }
428         s = splclock();
429         timevaladd(&atv, (struct timeval *) &time);
430         timo = hzto(&atv);
431         /*
432          * Avoid inadvertently sleeping forever.
433          */
434         if (timo == 0)
435             timo = 1;
436         splx(s);
437     } else
438         timo = 0;
```

sys_generic.c

图16-53 Select 函数：初始化

2. 从进程复制文件描述符集

411-418 getbits宏用copyin从进程那里将文件描述符集合传送到 ibits中的三个描述符集合。如果描述符集合指针为空，则不需复制。

3. 设置超时值

419-438 如果tv为空，则将timeo置成0，select将无限期等待。如果tv非空，则将超时值复制到内核，并调用itimerfix将超时值按硬件时钟的分辨率取整。调用timevaladd将当前时间加到超时值中。调用hzto计算从启动到超时之间的时钟滴答数，并保存在timo中。如果计算出来的结果为0，将timeo置1，从而防止select阻塞，实现利用全0的timeval结构来实现非阻塞操作。

select的第二部分代码，如图16-54所示。其作用是扫描进程指示的文件描述符，当一个或多个描述符处于就绪状态或定时器超时或信号出现时返回。

sys_generic.c

```
439   retry:
440     ncoll = nselcoll;
441     p->p_flag |= P_SELECT;
442     error = selscan(p, ibits, obits, uap->nd, retval);
443     if (error || *retval)
444         goto done;
445     s = splhigh();
446     /* this should be timercmp(&time, &atv, >=) */
447     if (uap->tv && (time.tv_sec > atv.tv_sec ||
448                 time.tv_sec == atv.tv_sec && time.tv_usec >= atv.tv_usec)) {
449         splx(s);
450         goto done;
451     }
452     if ((p->p_flag & P_SELECT) == 0 || nselcoll != ncoll) {
453         splx(s);
454         goto retry;
455     }
456     p->p_flag &= ~P_SELECT;
457     error = tsleep((caddr_t) & selwait, PSOCK | PCATCH, "select", timo);
458     splx(s);
459     if (error == 0)
460         goto retry;
461   done:
462     p->p_flag &= ~P_SELECT;
463     /* select is not restarted after signals... */
464     if (error == ERESTART)
465         error = EINTR;
466     if (error == EWOULDBLOCK)
467         error = 0;
468 #define putbits(name, x) \
469     if (uap->name && \
470         (error2 = copyout((caddr_t)&obits[x], (caddr_t)uap->name, ni))) \
471         error = error2;
472     if (error == 0) {
473         int     error2;
474         putbits(in, 0);
475         putbits(ou, 1);
476         putbits(ex, 2);
477 #undef putbits
478     }
479     return (error);
480 }
```

sys_generic.c

图16-54 select 函数：第二部分

4. 扫描文件描述符

439-442 从retry开始的循环直到select能够返回时退出。在调用进程的控制块中保存

全局整数nselcoll的当前值和P_SELECT标志。如果在selscan(图16-55)扫描文件描述符期间出现任何一种变化，则这种变化表明描述符的状态因为中断处理而发生改变， select必须重新扫描文件描述符。selscan查看三个输入的描述符集合中的每一个描述符集合，如果描述符处于就绪状态，则在输出的描述符集合中设置匹配的描述符。

sys_generic.c

```
481 selscan(p, ibits, obits, nfd, retval)
482 struct proc *p;
483 fd_set *ibits, *obits;
484 int     nfd, *retval;
485 {
486     struct filedesc *fdp = p->p_fd;
487     int     msk, i, j, fd;
488     fd_mask bits;
489     struct file *fp;
490     int     n = 0;
491     static int flag[3] =
492     {FREAD, FWRITE, 0};

493     for (msk = 0; msk < 3; msk++) {
494         for (i = 0; i < nfd; i += NFDBITS) {
495             bits = ibits[msk].fds_bits[i / NFDBITS];
496             while ((j = ffs(bits)) && (fd = i + --j) < nfd) {
497                 bits &= ~(1 << j);
498                 fp = fdp->fd_ofiles[fd];
499                 if (fp == NULL)
500                     return (EBADF);
501                 if ((*fp->f_ops->fo_select) (fp, flag[msk], p)) {
502                     FD_SET(fd, &obits[msk]);
503                     n++;
504                 }
505             }
506         }
507     }
508     *retval = n;
509     return (0);
510 }
```

sys_generic.c

图16-55 selscan 函数

5. 差错或一些描述符准备就绪

443-444 如果差错出现或描述符处于就绪状态，就立即返回。

6. 超时了吗

445-451 如果进程提供的时间限制和当前时间已经超过了超时值，则立即返回。

7. 在执行selscan期间状态发生变化

452-455 selscan可以被协议处理中断。如果在中断期间插口状态改变，则将 P_SELECT和nselcoll置位，且selscan必须重新扫描所有描述符。

8. 等待缓存发生变化

456-460 所有调用select的进程均在调用tsleep时用selwait作为等待通道。如图16-60所示，这种做法在多个进程等待同一个插口缓存的情况下将导致效率降低。如果 tsleep正确返回，则select跳转到retry，重新扫描所有描述符。

9. 准备返回

461-480 在done处清除P_SELECT，如果差错代码为ERESTART，则修改为EINTR；如果差错代码为EWOULDBLOCK，则将差错代码置成0。这些改变确保在select调用期间若信号出现时能返回EINTR；若超时，则返回0。

16.13.1 selscan函数

select函数的核心是图16-55所示的selscan函数。对于任意一个描述符集合中设置的每一个比特，selscan找出同它相关联的描述符，并且将控制分散给与描述符相关联的so_select函数。对于插口而言，就是soo_select函数。

1. 定位被监视的描述符

481-496 第一个for循环依次查看三个描述符集合：读，写和例外。第二个 for循环在每个描述符集合内部循环，这个循环在集合中每隔 32 bit(NFDBITS)循环一次。

最里面的while循环检查所有被32 bit 的掩码标记的描述符，该掩码从当前描述符集合中获取并保存在bits中。函数ffs返回bits中的第一个被设置的比特的位置，从最低位开始。例如，如果bits等于1000 (省略了前面的28个0)，则ffs (bits)等于4。

2. 轮询描述符

497-500 从i到ffs函数的返回值，计算与比特相关的描述符，并保存在 fd中。在bits中(而不是在输入描述符集合中)清除比特，找到与描述符相对应的 file结构，调用fo_select。

fo_select的第二个参数是flag数组中的一个元素。msk是外层的for循环的循环变量。所以，第一次循环时，第二个参数等于 FREAD，第二次循环时等于 FWRITE，第三次循环时等于0。如果描述符不正确，则返回EBADF。

3. 描述符准备就绪

501-504 当发现某个描述符的状态为准备就绪时，设置输出描述符集合中相对应的比特位。并将n (状态就绪的描述符的个数)加1。

505-510 循环继续直到轮询完所有描述符。状态就绪的描述符的个数通过 *retval返回。

16.13.2 soo_select函数

对于selscan在输入描述符集合中发现的每一个状态就绪的描述符，selscan调用与描述符相关的fileops结构(参考第15.5节)中的fo_select指针引用的函数。在本书中，我们只对插口描述符和图16-56所示的soo_select函数感兴趣。

sys_socket.c

```
105 soo_select(fp, which, p)
106 struct file *fp;
107 int     which;
108 struct proc *p;
109 {
110     struct socket *so = (struct socket *) fp->f_data;
111     int     s = splnet();
112     switch (which) {
113     case FREAD:
114         if (soreadable(so)) {
```

图16-56 soo_select 函数

```
115             splx(s);
116             return (1);
117         }
118         selrecord(p, &so->so_rcv.sb_sel);
119         so->so_rcv.sb_flags |= SB_SEL;
120         break;
121     case FWRITE:
122         if (sowriteable(so)) {
123             splx(s);
124             return (1);
125         }
126         selrecord(p, &so->so_snd.sb_sel);
127         so->so_snd.sb_flags |= SB_SEL;
128         break;
129     case 0:
130         if (so->so_oobmark || (so->so_state & SS_RCVATMARK)) {
131             splx(s);
132             return (1);
133         }
134         selrecord(p, &so->so_rcv.sb_sel);
135         so->so_rcv.sb_flags |= SB_SEL;
136         break;
137     }
138     splx(s);
139     return (0);
140 }
```

sys_socket.c

图16-56 (续)

105-112 soo_select每次被调用时，它只检查一个描述符的状态。如果相对于which中指定的条件，描述符处于就绪状态，则立即返回1。如果描述符没有处于就绪状态，就用selrecord标记插口的接收缓存或发送缓存，指示进程正在选择该缓存，然后soo_select返回0。

图16-52显示了插口的读、写和例外情况。我们将看到soo_select使用了soreadable和sowriteable宏，这些宏在sys/socketvar.h中定义。

1. 插口可读吗

113-120 soreadable宏的定义如下：

```
#define soreadable(so) \
    ((so)->so_rcv.sb_cc >= (so)->so_rcv.sb_lowat || \
    ((so)->so_state & SS_CANTRCVMORE) || \
    (so)->so_qlen || (so)->so_error)
```

因为UDP和TCP的接收下限默认值为1 (图16-4)，下列情况表示插口是可读的：接收缓存中有数据，连接的读通道被关闭，可以接受任何连接或有挂起的差错。

2. 插口可写吗

121-128 sowriteable宏的定义如下：

```
#define sowriteable(so) \
    (sbspace(&(so)->so_snd) >= (so)->so_snd.sb_lowat && \
    (((so)->so_state&SS_ISCONNECTED) || \
    ((so)->so_proto->pr_flags&PR_CONNREQUIRED)==0) || \
    ((so)->so_state & SS_CANTSENDMORE) || \
    (so)->so_error)
```

TCP和UDP默认的发送低水位标记是2048。对于UDP而言，sowriteable总是为真，因

为sbspace总是等于sb_hiwat，当然也总是大于或等于so_lowat，且不要求连接。

对于TCP而言，当发送缓存中的可用空间小于 2048个字节时，插口不可写。其他的情况在图16-52中讨论。

3. 还有挂起的例外情况吗

129-140 对于例外情况，需检查标志so_oobmark和SS_RECVATMARK。直到进程读完数据流中的同步标记后，例外情况才可能存在。

16.13.3 selrecord函数

图16-57显示同发送和接收缓存存储在一起的selinfo结构的定义(图16-3中的sb_sel成员)。

select.h

```
41 struct selinfo {
42     pid_t   si_pid;             /* process to be notified */
43     short   si_flags;           /* 0 or SI_COLL */
44 };
```

select.h

图16-57 selinfo 结构

41-44 当只有一个进程对某一给定的插口缓存调用 select时，sl_pid等于等待进程的进程标志符。当其他的进程对同一缓存调用 select时，设置sl_flags中的SI_COLL标志。将这种情况称为冲突。这个标志是目前 sl_flags中唯一已定义的标志。

当soo_select发现描述符不在就绪状态时就调用 selrecord函数，如图 16-58所示。该函数记录了足够的信息，使得缓存内容发生变化时协议处理层能够唤醒进程。

sys_generic.c

```
522 void
523 selrecord(selector, sip)
524 struct proc *selector;
525 struct selinfo *sip;
526 {
527     struct proc *p;
528     pid_t   mypid;

529     mypid = selector->p_pid;
530     if (sip->si_pid == mypid)
531         return;
532     if (sip->si_pid && (p = pfind(sip->si_pid)) &&
533         p->p_wchan == (caddr_t) & selwait)
534         sip->si_flags |= SI_COLL;
535     else
536         sip->si_pid = mypid;
537 }
```

sys_generic.c

图16-58 selrecord 函数

1. 重复选择描述符

522-531 selrecord的第一个参数指向调用 select进程的proc结构。第二个参数指向selinfo记录，该记录的 so_snd.sb_sel和so_rcv.sb_sel可能会被修改。如果selinfo中已记录了该进程的信息，则立即返回。例如，进程对同一个描述符调用 select查询读和例外情况时，函数就立即返回。

2. 同另一个进程的操作冲突？

532-534 如果另一个进程已经对同一插口缓存执行select操作，则设置SI_COLL。

3. 没有冲突

535-537 如果调用没有发生冲突，则 si_pid等于0，将当前进程的进程标志符赋给si_pid。

16.13.4 selwakeup函数

当协议处理改变插口缓存的状态，并且只有一个进程选择了该缓存时，Net/3就能根据selinfo结构中记录的信息立即将该进程放入运行队列。

当插口缓存发生变化但是有多个进程选择同一插口缓存时（设置了SI_COLL），Net/3就无法确定哪些进程对这种缓存变化感兴趣。我们在讨论图16-54中的代码段时就已经指出，每一个调用select的进程在调用tsleep时使用selwait作为等待通道。这意味着对应的wakeup将唤醒所有阻塞在select上的进程——甚至是对缓存的变化不关心的进程。

图16-59说明如何调用selwakeup。

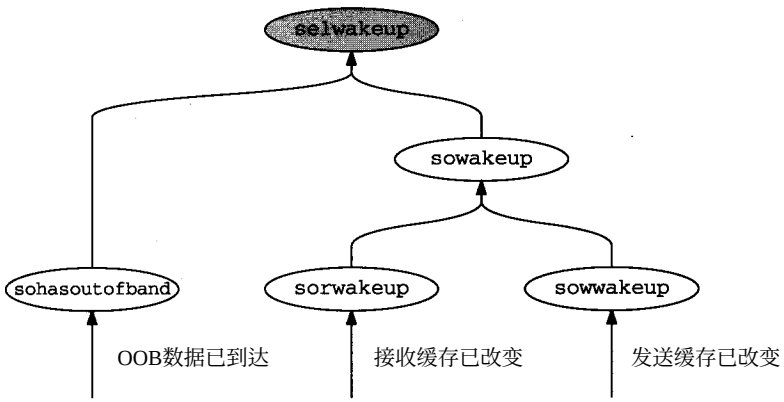


图16-59 selwakeup处理

当改变插口状态的事件出现时，协议处理层负责调用图16-59的底部列出的一个函数来通知插口层。图16-59底部显示的三个函数都将导致selwakeup被调用，在插口上选择的任何进程将被调度运行。

selwakeup函数如图16-60所示。

541-548 如果si_pid等于0，表明没有进程对该缓存执行select操作，函数立即返回。

在冲突中唤醒所有进程

549-553 如果多个进程对同一插口执行select操作，将nselcoll加1，清除冲突标志，唤醒所有阻塞在select上的进程。正如图16-54中讨论的，进程在tsleep中阻塞之前若缓存发生改变，nselcoll能使select重新扫描描述符(习题16.9)。

554-567 如果si_pid标识的进程正在selwait中等待，则调度该进程运行。如果进程是在其他等待通道中，则清除P_SELECT标志。如果对一个正确的描述符执行selrecord，则调用进程可能正在其他的等待通道中等待，然后，selscan在描述符集合中发现一个差错的文件描述符，并返回EBADF，不清除以前修改的selinfo记录。到selwakeup运行时，selwakup

可能会发现sel_pid标识的进程不再在插口缓存等待，从而忽略selinfo中的信息。

如果没有出现多个进程共享同一个描述符的情况 (也就是同一块插口缓存)，当然这种情况很少，则只有一个进程被selwakeup唤醒。在作者使用的机器上，nselcoll总是等于0，这说明select冲突是很少发生的。

sys_generic.c

```
541 void
542 selwakeup(sip)
543 struct selinfo *sip;
544 {
545     struct proc *p;
546     int     s;

547     if (sip->si_pid == 0)
548         return;
549     if (sip->si_flags & SI_COLL) {
550         nselcoll++;
551         sip->si_flags &= ~SI_COLL;
552         wakeup((caddr_t) & selwait);
553     }
554     p = pfind(sip->si_pid);
555     sip->si_pid = 0;
556     if (p != NULL) {
557         s = splhigh();
558         if (p->p_wchan == (caddr_t) & selwait) {
559             if (p->p_stat == SSLEEP)
560                 setrunnable(p);
561             else
562                 unsleep(p);
563         } else if (p->p_flag & P_SELECT)
564             p->p_flag &= ~P_SELECT;
565         splx(s);
566     }
567 }
```

sys_generic.c

图16-60 selwakeup 函数

16.14 小结

本章介绍了插口的读、写和选择系统调用。

我们了解到sosend处理插口层与协议处理层之间的所有输出，而 soreceive处理所有输入。

本章还介绍了发送缓存和接收缓存的组织结构，以及缓存的高、低水位标记的默认值和含义。

本章的最后一部分介绍了select系统调用。从这部分内容中我们了解到，当只有一个进程对描述符执行select调用时，协议处理层仅仅唤醒 selinfo结构中标识的那个进程。当有多个进程对同一个描述符执行select操作而发生冲突时，协议层只能唤醒所有等待在该描述符上的进程。

习题

16.1 当将一个大于最大的正的有符号整数的无符号整数传给 write系统调用时，

sosend中的resid如何变化？

16.2 当sosend将小于MCLBYTES个字节的数据放入簇中时，space被减去MCLBYTES，可能会成为一个负数，这会导致为atomic协议填写mbuf的循环结束。这种结果是正常的吗？

16.3 数据报和流协议有着不同的语义。将sosend和soreceive函数分别分成两个函数，一个用来处理报文，另一个用来处理流。除了使得代码清晰外，这样做还有什么好处？

16.4 对于PR_ATOMIC协议，每一个写调用都指定了一个隐含的报文边界。插口层将这个报文作为一个整体交给协议。MSG_EOR标志允许进程显式指定报文边界。为什么仅有隐含的报文边界是不够的？

16.5 如果插口描述符没有标记为非阻塞，且进程也没有指定MSG_DONTWAIT，当sosend不能立即获取发送缓存上的锁时，结果如何？

16.6 在什么情况下，虽然sb_cc<sb_hiwat，但sbspace仍然报告没有闲置空间？为什么在这种情况下进程应该被阻塞？

16.7 为什么recvit不将控制报文的长度而是将名字长度返回给进程？

16.8 为什么soreceive要清除MSG_EOR？

16.9 如果将nselcoll代码从select和selwakeup中删除，会有什么问题？

16.10 修改select系统调用，使得select返回时返回定时器的剩余时间。

第17章 插口选项

17.1 引言

本章讨论修改插口行为的几个系统调用，以此来结束插口层的介绍。

setsockopt和getsockopt系统调用已在第8.8节中介绍过，主要描述访问IP特点的选项。在本章中，我们将介绍这两个系统调用的实现以及通过它们来控制的插口级选项。

ioctl函数在第4.4节中已介绍过，在第4.4节中，我们描述了用于网络接口配置的与协议无关的ioctl命令。在第6.7节中，我们描述了用来分配网络掩码以及单播、多播和目的地址的IP专用的ioctl命令。本章我们将介绍ioctl的实现和fcntl函数的相关特点。

最后，我们介绍getsockname和getpeername系统调用，它们用来返回插口和连接的地址信息。

图17-1列出了实现插口选项系统调用的函数。本章描述带阴影的函数。

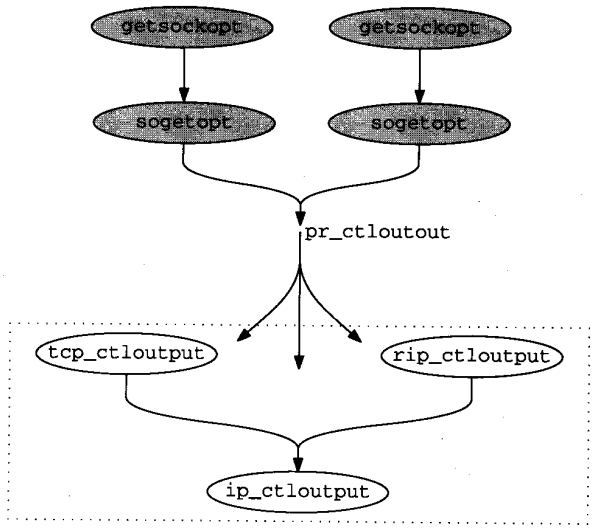


图17-1 setsockopt 和getsockopt 系统调用

17.2 代码介绍

本章中涉及的源代码来自于图17-2中列出的四个文件。

文 件 名	说 明
kern/kern_descrip.c	fcntl系统调用
kern/uipc_syscalls.c	setsockopt、getsockopt、getsockname和getpeername系统调用
kern/uipc_socket.c	插口层对setsockopt和getsockopt的处理
kern/sys_socket.c	ioctl系统调用对插口的处理

图17-2 本章讨论的源文件

全局变量和统计量

本章中描述的系统调用没有定义新的全局变量，也没有收集任何统计量。

17.3 setsockopt系统调用

图8-29列出了函数setsockopt (和getsockopt)能够访问的各种不同的协议层。本章主要集中在SOL_SOCKET级的选项，这些选项在图17-3中列出。

optname	optval 类型	变 量	说 明
SO_SNDBUF	int	so_snd.sb_hiwat	发送缓存高水位标记
SO_RCVBUF	int	so_rcv.sb_hiwat	接收缓存高水位标记
SO_SNDLOWAT	int	so_snd.sb_lowat	发送缓存低水位标记
SO_RCVLOWAT	int	so_rcv.sb_lowat	接收缓存低水位标记
SO_SNDTIMEO	struct timeval	so_snd.sb_timeo	发送超时值
SO_RCVTIMEO	struct timeval	so_snd.sb_timeo	接收超时值
SO_DEBUG	int	so_options	记录插口调试信息
SO_REUSEADDR	int	so_options	插口能重新使用一个本地地址
SO_REUSEPORT	int	so_options	插口能重新使用一个本地端口
SO_KEEPALIVE	int	so_options	协议查询空闲的连接
SO_DONTROUTE	int	so_options	旁路路由表
SO_BROADCAST	int	so_options	插口支持广播报文
SO_USELOOPBACK	int	so_options	仅用于选路域插口；发送进程接收自己的选路报文
SO_OOBINLINE	int	so_options	协议排队内联的带外数据
SO_LINGER	struct linger	so_linger	插口关闭但仍发送剩余数据
SO_ERROR	int	so_error	获取差错状态并清除；只用于getsockopt
SO_TYPE	int	so_type	获取插口类型；只用于getsockopt
其他			返回ENOPROTOOPT

图17-3 setsockopt 和getsockopt 选项

setsockopt函数原型为：

```
int setsockopt(int s, int level, int optname, void *optval, int optlen);
```

图17-4显示了setsockopt调用的源代码。

565-597 getsock返回插口描述符的file结构。如果val非空，则将valsize个字节的数据从进程复制到用m_get分配的mbuf中。与选项对应的数据长度不能超过MLEN个字节，所以，如果valsize大于MLEN，则返回EINVAL。调用sosetopt，并返回其值。

uipc_syscalls.c

```
565 struct setsockopt_args {
566     int     s;
567     int     level;
568     int     name;
569     caddr_t val;
570     int     valsize;
571 };

572 setsockopt(p, uap, retval)
573 struct proc *p;
574 struct setsockopt_args *uap;
575 int    *retval;
```

图17-4 setsockopt 系统调用

```
576 {
577     struct file *fp;
578     struct mbuf *m = NULL;
579     int     error;

580     if (error = getsock(p->p_fd, uap->s, &fp))
581         return (error);
582     if (uap->valsize > MLEN)
583         return (EINVAL);
584     if (uap->val) {
585         m = m_get(M_WAIT, MT_SOOPTS);
586         if (m == NULL)
587             return (ENOBUFS);
588         if (error = copyin(uap->val, mtod(m, caddr_t),
589                            (u_int) uap->valsize)) {
590             (void) m_free(m);
591             return (error);
592         }
593         m->m_len = uap->valsize;
594     }
595     return (sosetopt((struct socket *) fp->f_data, uap->level,
596                      uap->name, m));
597 }
```

uipc_syscalls.c

图17-4 (续)

sosetopt函数

sosetopt函数处理所有插口级的选项，并将其他的选项传给与插口关联的协议的pr_ctloutput函数。图17-5列出了sosetopt函数的部分代码。

uipc_socket.c

```
752 sosetopt(so, level, optname, m0)
753 struct socket *so;
754 int     level, optname;
755 struct mbuf *m0;
756 {
757     int     error = 0;
758     struct mbuf *m = m0;

759     if (level != SOL_SOCKET) {
760         if (so->so_proto && so->so_proto->pr_ctloutput)
761             return ((*so->so_proto->pr_ctloutput)
762                     (PRCO_SETOPT, so, level, optname, &m0));
763         error = ENOPROTOOPT;
764     } else {
765         switch (optname) {

                    /* socket option processing */

841         default:
842             error = ENOPROTOOPT;
843             break;
844         }
845         if (error == 0 && so->so_proto && so->so_proto->pr_ctloutput) {
846             (void) ((*so->so_proto->pr_ctloutput)
847                     (PRCO_SETOPT, so, level, optname, &m0));
```

图17-5 sosetopt 函数

```
848             m = NULL;           /* freed by protocol */
849         }
850     }
851   bad:
852     if (m)
853         (void) m_free(m);
854     return (error);
855 }
```

uipc_socket.c

图17-5 (续)

752-764 如果选项不是插口级的(SOL_SOCKET)选项，则给底层协议发送PRCO_SETOPT请求。注意：调用的是协议的pr_ctloutput函数，而不是它的pr_usrreq函数。图17-6说明了Internet协议调用的pr_ctloutput函数。

协　议	pr_ctloutput函数	参　考
UDP TCP	ip_ctloutput tcp_ctloutput	第8.8节 第30.6节
ICMP IGMP 原始IP	rip_ctloutput和ip_ctloutput	第8.8节和第32.8节

图17-6 pr_ctloutput 函数

765 switch语句处理插口级的选项。

841-844 对于不认识的选项，在保存它的mbuf被释放后返回ENOPROTOOPT。

845-855 如果没有出现差错，则控制总是会执行到switch。在switch语句中，如果协议层需要响应请求或插口层，则将选项传送给相应的协议。 Internet协议中没有一个预期处理插口级的选项。

注意，如果协议收到不预期的选项，则直接将其pr_ctloutput函数的返回值丢弃。并将m置空，以免调用m_free，因为协议负责释放缓存。

图17-7说明了linger选项和在插口结构中设置单一标志的选项。

766-772 linger选项要求进程传入linger结构：

```
struct linger {
   int l_onoff;   /* option on/off */
   int l_linger;  /* linger time in seconds */
};
```

确保进程已传入长度为 linger结构大小的数据后，将结构成员 l_linger复制到so_linger中。在下一组case语句后决定是使能还是关闭该选项。 so_linger和close系统调用在第15.15节中已介绍过。

773-789 当进程传入一个非0值时，设置选项对应的布尔标志；当进程传入的是 0时，将对应标志清除。第一次检查确保一个整数大小 (或更大)的对象在mbuf中，然后设置或清除对应的选项。

图17-8显示了插口缓存选项的处理。

790-815 这组选项改变插口的发送和接收缓存的大小。第一个 if语句确保提供给四个选项的变量是整型。对于SO_SNDBUF和SO_RCVBUF，sbreserve只调整缓存的高水位标记而不

分配缓存。对于SO_SNDLOWAT和SO_RCVLOWAT，调整缓存的低水位标记。

uipc_socket.c

```
766         case SO_LINGER:
767             if (m == NULL || m->m_len != sizeof(struct linger)) {
768                 error = EINVAL;
769                 goto bad;
770             }
771             so->so_linger = mtod(m, struct linger *)->l_linger;
772             /* fall thru... */

773         case SO_DEBUG:
774         case SO_KEEPALIVE:
775         case SO_DONTROUTE:
776         case SO_USELOOPBACK:
777         case SO_BROADCAST:
778         case SO_REUSEADDR:
779         case SO_REUSEPORT:
780         case SO_OOBINLINE:
781             if (m == NULL || m->m_len < sizeof(int)) {
782                 error = EINVAL;
783                 goto bad;
784             }
785             if (*mtod(m, int *))
786                 so->so_options |= optname;
787             else
788                 so->so_options &= ~optname;
789             break;
```

uipc_socket.c

图17-7 sosetopt 函数：linger 和标志选项

uipc_socket.c

```
790         case SO_SNDBUF:
791         case SO_RCVBUF:
792         case SO_SNDLOWAT:
793         case SO_RCVLOWAT:
794             if (m == NULL || m->m_len < sizeof(int)) {
795                 error = EINVAL;
796                 goto bad;
797             }
798             switch (optname) {

799             case SO_SNDBUF:
800             case SO_RCVBUF:
801                 if (sbreserve(optname == SO_SNDBUF ?
802                             &so->so_snd : &so->so_rcv,
803                             (u_long) * mtod(m, int *)) == 0) {
804                     error = ENOBUFS;
805                     goto bad;
806                 }
807                 break;

808             case SO_SNDLOWAT:
809                 so->so_snd.sb_lowat = *mtod(m, int *);
810                 break;
811             case SO_RCVLOWAT:
812                 so->so_rcv.sb_lowat = *mtod(m, int *);
813                 break;
814             }
815             break;
```

uipc_socket.c

图17-8 sosetopt 函数：插口缓存选项

图17-9说明超时选项。

uipc_socket.c

```
816         case SO_SNDTIMEO:
817         case SO_RCVTIMEO:
818             {
819                 struct timeval *tv;
820                 short   val;

821                 if (m == NULL || m->m_len < sizeof(*tv)) {
822                     error = EINVAL;
823                     goto bad;
824                 }
825                 tv = mtod(m, struct timeval *);
826                 if (tv->tv_sec > SHRT_MAX / hz - hz) {
827                     error = EDOM;
828                     goto bad;
829                 }
830                 val = tv->tv_sec * hz + tv->tv_usec / tick;

831                 switch (optname) {

832                 case SO_SNDTIMEO:
833                     so->so_snd.sb_timeo = val;
834                     break;
835                 case SO_RCVTIMEO:
836                     so->so_rcv.sb_timeo = val;
837                     break;
838                 }
839                 break;
840             }
```

uipc_socket.c

图17-9 sosetopt 函数：超时选项

816-824 进程在timeval结构中设置SO_SNDTIMEO和SO_RCVTIMEO选项的超时值。如果传入的数值不正确，则返回EINVAL。

825-830 存储在timeval结构中的时间间隔值不能太大，因为sb_timeo是一个短整数，当时间间隔值的单位为一个时钟滴答时，时间间隔值的大小就不能超过一个短整数的最大值。

第826行代码是不正确的。在下列条件下，时间间隔不能表示为一个短整数：

$$\text{tv\_sec} \times \text{hz} + \frac{\text{tv\_usec}}{\text{tick}} > \text{SHRT\_MAX}$$

其中，ticck=1 000 000/和SHRT_MAX=32767

所以，如果下列不等式成立，则返回。

$$\text{tv\_sec} > \frac{\text{SHRT\_MAX}}{\text{hz}} - \frac{\text{tv\_usec}}{\text{tick} \times \text{hz}} = \frac{\text{SHRT\_MAX}}{\text{hz}} - \frac{\text{tv\_usec}}{100000}$$

等式的最后一项不是代码指明的hz。正确的测试代码应该是：

```
if (tv->tv_sec * hz+tv->tv_usec/tick>SHRT_MAX)
error=EDOM;
```

习题17.3中有更详细的讨论。

831-840 将转换后的时间，val，保存在请求的发送或接收缓存中。sb_timeo限制了进程等待接收缓存中的数据或发送缓存中的闲置空间的时间。详细讨论参考第 16.7和16.11节。

超时值是传给tsleep的最后一个参数，因为tsleep要求超时值为一个整数，所以进程最多只能等待65535个时钟滴答。假设时钟频率为100 Hz，则等待时间小于11分钟。

17.4 getsockopt系统调用

getsockopt返回进程请求的插口和协议选项。函数原型是：

```
int getsockopt(int s, int level, int name, caddr_t val, int *valsize);
```

该调用的源代码如图17-10所示。

uipc_syscalls.c

```
598 struct getsockopt_args {
599     int     s;
600     int     level;
601     int     name;
602     caddr_t val;
603     int    *avalsize;
604 };

605 getsockopt(p, uap, retval)
606 struct proc *p;
607 struct getsockopt_args *uap;
608 int    *retval;
609 {
610     struct file *fp;
611     struct mbuf *m = NULL;
612     int     valsize, error;

613     if (error = getsock(p->p_fd, uap->s, &fp))
614         return (error);
615     if (uap->val) {
616         if (error = copyin((caddr_t) uap->avalsize, (caddr_t) & valsize,
617                            sizeof(valsize)))
618             return (error);
619     } else
620         valsize = 0;
621     if ((error = sogetopt((struct socket *) fp->f_data, uap->level,
622                 uap->name, &m)) == 0 && uap->val && valsize && m != NULL) {
623         if (valsize > m->m_len)
624             valsize = m->m_len;
625         error = copyout(mtod(m, caddr_t), uap->val, (u_int) valsize);
626         if (error == 0)
627             error = copyout((caddr_t) & valsize,
628                             (caddr_t) uap->avalsize, sizeof(valsize));
629     }
630     if (m != NULL)
631         (void) m_free(m);
632     return (error);
633 }
```

uipc_syscalls.c

图17-10 getsockopt 系统调用

598-633 这段代码现在看上去应该很熟悉了。getsock获取插口的file结构，将选项缓存的大小复制到内核，调用sogetopt来获取选项的值。将sogetopt返回的数据复制到进程提供的缓存，可能还需修改缓存长度。如果进程提供的缓存不够大，则返回的数据可能会被截掉。通常情况下，存储选项数据的mbuf在函数返回后被释放。

sogetopt函数

同sosetopt一样，sogetopt函数处理所有插口级的选项，并将其他的选项传给与插口关联的协议。图17-11列出了sogetopt函数的开始和结束部分的代码。

uipc_socket.c

```
856 sogetopt(so, level, optname, mp)
857 struct socket *so;
858 int     level, optname;
859 struct mbuf **mp;
860 {
861     struct mbuf *m;

862     if (level != SOL_SOCKET) {
863         if (so->so_proto && so->so_proto->pr_ctloutput) {
864             return ((*so->so_proto->pr_ctloutput)
865                     (PRCO_GETOPT, so, level, optname, mp));
866         } else
867             return (ENOPROTOOPT);
868     } else {
869         m = m_get(M_WAIT, MT_SOOPTS);
870         m->m_len = sizeof(int);

871         switch (optname) {

                        /* socket option processing */

918         default:
919             (void) m_free(m);
920             return (ENOPROTOOPT);
921         }
922         *mp = m;
923         return (0);
924     }
925 }
```

uipc_socket.c

图17-11 sogetopt 函数：概述

856-871 同sosetopt一样，函数将那些与插口级选项无关的选项立即通过PRCO_GETOPT协议请求传递给相应的协议级。协议将被请求的选项保存在mp指向的mbuf中。

对于插口级的选项，分配一块标准的mbuf缓存来保存选项值，选项值通常是一个整数，所以将m_len设成整数大小。相应的选项值通过switch语句复制到mbuf中。

918-925 如果执行的是switch中的default情况下的语句，则释放mbuf，并返回ENOPROTOOPT。否则，switch语句执行完成后，将指向mbuf的指针赋给*mp。当函数返回后，getsockopt从该mbuf中将数据复制到进程提供的缓存，并释放mbuf。

图17-12说明对SO_LINGER选项和作为布尔型标志实现的选项的处理。

872-877 SO_LINGER选项请求返回两个值：一个是标志值，赋给l_onoff；另一个是拖延时间，赋给l_linger。

878-887 其余的选项作为布尔标志实现。将so_options和optname执行逻辑与操作，如果选项被打开，则与操作的结果为非0值；反之则结果为0。注意：标志被打开并不表示返回值等于1。

sogetopt的下一部分代码(图17-13)将整型值选项的值复制到mbuf中。

888-906 将每一个选项作为一个整数复制到mbuf中。注意：有些选项在内核中是作为一个短整数存储的(如缓存高低水位标记)，但是作为整数返回。一旦将so_error复制到mbuf中后，即清除so_error，这是唯一的一次getsockopt调用修改插口状态。

uipc_socket.c

```
872         case SO_LINGER:
873             m->m_len = sizeof(struct linger);
874             mtod(m, struct linger *)->l_onoff =
875                     so->so_options & SO_LINGER;
876             mtod(m, struct linger *)->l_linger = so->so_linger;
877             break;

878         case SO_USELOOPBACK:
879         case SO_DONTROUTE:
880         case SO_DEBUG:
881         case SO_KEEPALIVE:
882         case SO_REUSEADDR:
883         case SO_REUSEPORT:
884         case SO_BROADCAST:
885         case SO_OOBINLINE:
886             *mtod(m, int *) = so->so_options & optname;
887             break;
```

uipc_socket.c

图17-12 sogetopt 选项：SO_LINGER 选项和布尔选项

uipc_socket.c

```
888         case SO_TYPE:
889             *mtod(m, int *) = so->so_type;
890             break;

891         case SO_ERROR:
892             *mtod(m, int *) = so->so_error;
893             so->so_error = 0;
894             break;

895         case SO_SNDBUF:
896             *mtod(m, int *) = so->so_snd.sb_hiwat;
897             break;

898         case SO_RCVBUF:
899             *mtod(m, int *) = so->so_rcv.sb_hiwat;
900             break;
901         case SO_SNDLOWAT:
902             *mtod(m, int *) = so->so_snd.sb_lowat;
903             break;

904         case SO_RCVLOWAT:
905             *mtod(m, int *) = so->so_rcv.sb_lowat;
906             break;
```

uipc_socket.c

图17-13 sogetopt 函数：整型值选项

uipc_socket.c

```
907         case SO_SNDTIMEO:
908         case SO_RCVTIMEO:
909             {
910                 int     val = (optname == SO_SNDTIMEO ?
911                                 so->so_snd.sb_timeo : so->so_rcv.sb_timeo);

912                 m->m_len = sizeof(struct timeval);
913                 mtod(m, struct timeval *)->tv_sec = val / hz;
914                 mtod(m, struct timeval *)->tv_usec =
915                         (val % hz) / tick;
916                 break;
917             }
```

uipc_socket.c

图17-14 sogetopt 函数：超时选项

图17-14列出了sogetopt的第三和第四部分代码，它们的作用分别是处理SO_SNDTIMEO和SO_RCVTIMEO选项。

907-917 将发送或接收缓存中的sb_timeo值赋给var。基于val中的时钟滴答数，在mbuf中构造一个timeval结构。

计算tv_usec的代码有一个差错。表达式应该为："(val % hz) * tick"。

17.5 fcntl和ioctl系统调用

因为历史的原因而非有意这么做，插口API的几个特点既能通过ioctl也能通过fcntl来访问。关于ioctl命令，我们已经讨论了很多。我们也几次提到fcntl。

图17-15显示了本章描述的函数。

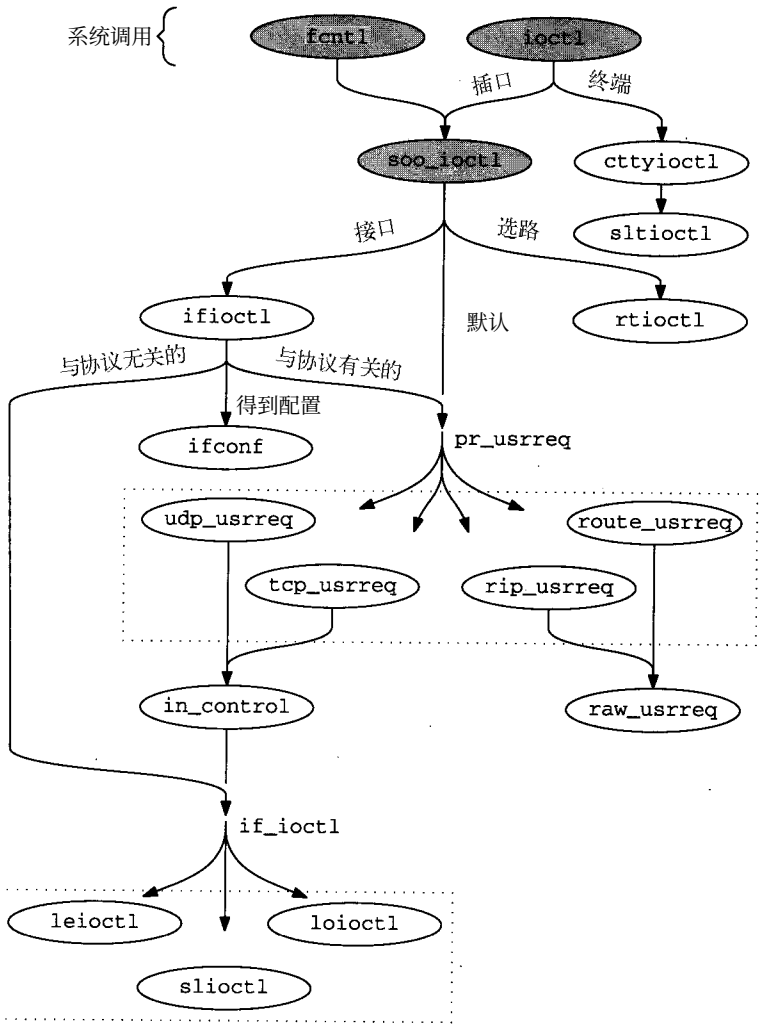


图17-15 fcntl和ioctl函数

ioctl和fcntl的原型分别为：

```
int ioctl(int fd, unsigned long result, char *argp);
int fcntl(int fd, int cmd,... /* int arg */);
```

图17-16总结了这两个系统调用与插口有关的特点。我们在图17-16中还列出了一些传统的常数，因为它们出现在代码中。考虑与Posix的兼容性，可以用O_NONBLOCK来代替FNONBLOCK，用O_ASYNC来代替FASYNC。

描　述	fcntl	ioctl
通过打开或关闭so_state中的SS_NBIO来使能或禁止非阻塞功能	FNONBLOCK文件状态标志	FIONBIO命令
通过打开或关闭sb_flags中的SB_ASYNC来使能或禁止异步通知功能	FASYNC文件状态标志	FIOASYNC命令
设置或得到so_pgid，它是SIGIOG和SIGURG信号的目标进程或进程组	F_SETOWN或F_GETOWN	SIOCSPGRP或SIOCGPGRP命令
得到接收缓存中的字节数；返回so_rcv.sb_cc		FIONREAD
返回OOB同步标记；即so_state中的SS_RCVATMARK标志		SIOCATMARK

图17-16　fcntl 和ioctl 命令

17.5.1 fcntl代码

图17-17列出了fcntl函数的部分代码。

kern_descrip.c

```
133 struct fcntl_args {
134     int     fd;
135     int     cmd;
136     int     arg;
137 };
138 /* ARGSUSED */
139 fcntl(p, uap, retval)
140 struct proc *p;
141 struct fcntl_args *uap;
142 int    *retval;
143 {
144     struct filedesc *fdp = p->p_fd;
145     struct file *fp;
146     struct vnode *vp;
147     int     i, tmp, error, flg = F_POSIX;
148     struct flock fl;
149     u_int   newmin;
150     if ((unsigned) uap->fd >= fdp->fd_nfiles ||
151         (fp = fdp->fd_ofiles[uap->fd]) == NULL)
152         return (EBADF);
153     switch (uap->cmd) {
                /* command processing */
253     default:
254         return (EINVAL);
255     }
256     /* NOTREACHED */
257 }
```

kern_descrip.c

图17-17　fcntl 系统调用：概况

133-153 验证完指向打开文件的描述符的正确性后，switch语句处理请求的命令。

253-257 对于不认识的命令，fcntl返回EINVAL。

图17-18仅显示fcntl中与插口有关的代码。

kern_descrip.c

```
168     case F_GETFL:
169         *retval = OFLAGS(fp->f_flag);
170         return (0);
171     case F_SETFL:
172         fp->f_flag &= ~FCNTLFLAGS;
173         fp->f_flag |= FFLAGS(uap->arg) & FCNTLFLAGS;
174         tmp = fp->f_flag & FNONBLOCK;
175         error = (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) & tmp, p);
176         if (error)
177             return (error);
178         tmp = fp->f_flag & FASYNC;
179         error = (*fp->f_ops->fo_ioctl) (fp, FIOASYNC, (caddr_t) & tmp, p);
180         if (!error)
181             return (0);
182         fp->f_flag &= ~FNONBLOCK;
183         tmp = 0;
184         (void) (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) & tmp, p);
185         return (error);
186     case F_GETOWN:
187         if (fp->f_type == DTYPE_SOCKET) {
188             *retval = ((struct socket *) fp->f_data)->so_pgid;
189             return (0);
190         }
191         error = (*fp->f_ops->fo_ioctl)
192             (fp, (int) TIOCGPGRP, (caddr_t) retval, p);
193         *retval = -*retval;
194         return (error);
195     case F_SETOWN:
196         if (fp->f_type == DTYPE_SOCKET) {
197             ((struct socket *) fp->f_data)->so_pgid = uap->arg;
198             return (0);
199         }
200         if (uap->arg <= 0) {
201             uap->arg = -uap->arg;
202         } else {
203             struct proc *p1 = pfind(uap->arg);
204             if (p1 == 0)
205                 return (ESRCH);
206             uap->arg = p1->p_pgrp->pg_id;
207         }
208         return ((*fp->f_ops->fo_ioctl)
209                 (fp, (int) TIOCSPGRP, (caddr_t) & uap->arg, p));
```

kern_descrip.c

图17-18 fcntl 系统调用：插口处理

168-185 F_GETFL返回与描述符相关的当前文件状态标志，F_SETFL设置状态标志。通过调用fo_ioctl将FNONBLOCK和FASYNC的新设置传递给对应的插口，而插口的新设置是通过图17-20中描述的soo_ioctl函数来传递的。只有在第二个 fo_ioctl调用失败后，才第三次调用fo_ioctl。该调用的功能是清除FNONBLOCK标志，但是应该改为将这个标志恢复

到原来的值。

186-194 F_GETOWN返回与插口相关联的进程或进程组的标识符，so_pgid。对于非插口描述符，将TIOCGPGRP ioctl命令传给对应的fo_ioctl函数。F_SETOWN的功能是给so_pgid赋一个新值。

17.5.2 ioctl代码

我们跳过ioctl系统调用本身而先从soo_ioctl开始讨论，如图17-20所示，因为ioctl的代码中的大部分是从图17-17所示的代码中复制的。我们已经说过，soo_ioctl函数将选路命令发送给rtioctl，接口命令发送给ifioctl，任何其他的命令发送给底层协议的pr_usrreq函数。

55-68 有几个命令是由soo_ioctl直接处理的。如果*data非空，则FIONBIO打开非阻塞方式，否则关闭非阻塞方式。正于我们已经了解的，这个标志将影响到accept、connect和close系统调用，也包括其他的读和写系统调用。

69-79 FIOASYNC使能或禁止异步I/O通知功能。如果设置了SS_ASYNC，则无论什么时候插口上有活动，就调用sowakeup，将信号SIGIO发送给相应的进程或进程组。

80-88 FIONREAD返回接收缓存中的可读字节数。SIOCSPGRP设置与插口相关的进程组，SIOCGPGRP则是得到它。so_pgid作为我们刚讨论过的SIGIO信号的目标进程或进程组，当有带外数据到达插口时，则作为SIGURG信号的目标进程或进程组。

89-92 如果插口正处于带外数据的同步标记，则SIOCATMARK返回真；否则返回假。

ioctl命令，FIO*xxx*和SIO*xxx*常量，有一个内部结构，如图17-19所示。

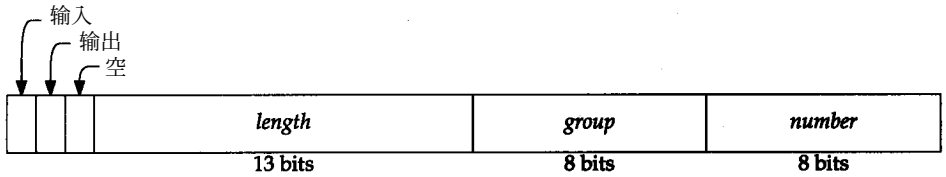


图17-19 ioctl 命令的内部结构

sys_socket.c

```
55 soo_ioctl(fp, cmd, data, p)
56 struct file *fp;
57 int     cmd;
58 caddr_t data;
59 struct proc *p;
60 {
61     struct socket *so = (struct socket *) fp->f_data;
62     switch (cmd) {
63     case FIONBIO:
64         if (*(int *) data)
65             so->so_state |= SS_NBIO;
66         else
67             so->so_state &= ~SS_NBIO;
68         return (0);
69     case FIOASYNC:
70         if (*(int *) data) {
71             so->so_state |= SS_ASYNC;
72             so->so_rcv.sb_flags |= SB_ASYNC;
```

图17-20 soo_ioctl 函数

```
 73              so->so_snd.sb_flags |= SB_ASYNC;
 74          } else {
 75              so->so_state &= ~SS_ASYNC;
 76              so->so_rcv.sb_flags &= ~SB_ASYNC;
 77              so->so_snd.sb_flags &= ~SB_ASYNC;
 78          }
 79          return (0);
 80      case FIONREAD:
 81          *(int *) data = so->so_rcv.sb_cc;
 82          return (0);
 83      case SIOCSPGRP:
 84          so->so_pgid = *(int *) data;
 85          return (0);
 86      case SIOCGPGRP:
 87          *(int *) data = so->so_pgid;
 88          return (0);
 89      case SIOCATMARK:
 90          *(int *) data = (so->so_state & SS_RCVATMARK) != 0;
 91          return (0);
 92      }
 93      /*
 94       * Interface/routing/protocol specific ioctls:
 95       * interface and routing ioctls should have a
 96       * different entry since a socket's unnecessary
 97       */
 98      if (IOCGROUP(cmd) == 'i')
 99          return (ifioctl(so, cmd, data, p));
100      if (IOCGROUP(cmd) == 'r')
101          return (rtioctl(cmd, data, p));
102      return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
103              (struct mbuf *) cmd, (struct mbuf *) data, (struct mbuf *) 0));
104  }
```

sys_socket.c

图17-20 (续)

如果将ioctl的第三个参数作为输入，则设置*input*。如果该参数作为输出，则*output*被置位。如果不用该参数，则*void*被置位。*length*是参数的大小(字节)。相关的命令在同一个*group*中但每一个命令在组中都有各自的*number*。图17-21中的宏用来解析ioctl命令中的元素。

宏	描　述
IOCPARM_LEN(cmd)	返回cmd中的*length*
IOCBASECMD(cmd)	*length*设为0的命令
IOCGROUP(cmd)	返回*cmd*中*的group*

图17-21 ioctl 命令宏

93-104 宏IOCGROUP从命令中得到8 bit的*group*。接口命令由ifioctl处理。选路命令由rtioctl处理。通过PRU_CONTROL请求将所有其他的命令传递给插口协议。

正如我们在第19章中描述的，Net/2定义了一个新的访问路由选择表接口，在该接口中，报文是通过一个在PF_ROUTE域中产生的插口传递给路由选择子系统。用这种方法来代替这里讨论的ioctl。在不兼容的内核中，rtioctl总是返回ENOTSUPP。

17.6 getsockname系统调用

getsockname系统调用的原型是：

```
int getsockname(int fd, caddr_t asa, int * alen);
```

getsockname得到绑定在插口*fd*上的本地地址，并将它存入*asa*指向的缓存中。当在一个隐式的绑定中内核选择了一个地址，或在一个显式的bind调用中进程指定了一个通配符地址(2.2.5节)时，该函数就很有用。getsockname系统调用如图17-22所示。

uipc_syscalls.c

```
682 struct getsockname_args {
683     int     fdes;
684     caddr_t asa;
685     int    *alen;
686 };

687 getsockname(p, uap, retval)
688 struct proc *p;
689 struct getsockname_args *uap;
690 int    *retval;
691 {
692     struct file *fp;
693     struct socket *so;
694     struct mbuf *m;
695     int     len, error;

696     if (error = getsock(p->p_fd, uap->fdes, &fp))
697         return (error);
698     if (error = copyin((caddr_t) uap->alen, (caddr_t) & len, sizeof(len)))
699         return (error);
700     so = (struct socket *) fp->f_data;
701     m = m_getclr(M_WAIT, MT_SONAME);
702     if (m == NULL)
703         return (ENOBUFS);
704     if (error = (*so->so_proto->pr_usrreq) (so, PRU_SOCKADDR, 0, m, 0))
705         goto bad;
706     if (len > m->m_len)
707         len = m->m_len;
708     error = copyout(mtod(m, caddr_t), (caddr_t) uap->asa, (u_int) len);
709     if (error == 0)
710         error = copyout((caddr_t) & len, (caddr_t) uap->alen,
711                         sizeof(len));
712   bad:
713     m_freem(m);
714     return (error);
715 }
```

uipc_syscalls.c

图17-22 getsockname 系统调用

682-715 getsock返回描述符的file结构。将进程指定的缓存的长度赋给len。这是我们第一次看到对m_getclr的调用：该函数分配一个标准的mbuf，并调用bzero清零。当协议收到PRU_SOCKADDR请求时，协议处理层负责将本地地址存入m。

如果地址长度大于进程提供的缓存的长度，则返回的地址将被截掉。 *alen等于实际返回的字节数。最后，释放mbuf，并返回。

17.7 getpeername系统调用

getpeername系统调用的原型是：

```
int getpeername(int fd, caddr_t asa, int *alen);
```

getpeername系统调用返回指定插口上连接的远端地址。当一个调用 accept的进程通过fork和exec启动一个服务器时(即，任何被inetd启动的服务器)，经常要调用这个函数。服务器不能得到accept返回的远端地址，而只能调用 getpeername。通常，要在应用的访问地址表查找返回地址，如果返回地址不在访问表中，则连接将被关闭。

某些协议，如 TP4，利用这个函数来确定是否拒绝或证实一个进入的连接。在 TP4中，accept返回的插口上的连接是不完整的，必须经证实之后才算连接成功。基于getpeername返回的地址，服务器能够关闭连接或通过发送或接收数据来间接证实连接。这一特点与TCP 无关，因为TCP必须在三次握手完成之后，accept才能建立连接。图17-23列出了getpeername函数的代码。

uipc_syscalls.c

```
719 struct getpeername_args {
720     int     fdes;
721     caddr_t asa;
722     int    *alen;
723 };

724 getpeername(p, uap, retval)
725 struct proc *p;
726 struct getpeername_args *uap;
727 int    *retval;
728 {
729     struct file *fp;
730     struct socket *so;
731     struct mbuf *m;
732     int     len, error;

733     if (error = getsock(p->p_fd, uap->fdes, &fp))
734         return (error);
735     so = (struct socket *) fp->f_data;
736     if ((so->so_state & (SS_ISCONNECTED | SS_ISCONFIRMING)) == 0)
737         return (ENOTCONN);
738     if (error = copyin((caddr_t) uap->alen, (caddr_t) & len, sizeof(len)))
739         return (error);
740     m = m_getclr(M_WAIT, MT_SONAME);
741     if (m == NULL)
742         return (ENOBUFS);
743     if (error = (*so->so_proto->pr_usrreq) (so, PRU_PEERADDR, 0, m, 0))
744         goto bad;
745     if (len > m->m_len)
746         len = m->m_len;
747     if (error = copyout(mtod(m, caddr_t), (caddr_t) uap->asa, (u_int) len))
748         goto bad;
749     error = copyout((caddr_t) & len, (caddr_t) uap->alen, sizeof(len));
750   bad:
751     m_freem(m);
752     return (error);
753 }
```

uipc_syscalls.c

图17-23 getpeername 系统调用

719-753 图中列出的代码与getsockname的代码是一样的。getsock获取插口对应的file结构，如果插口还没有同对方建立连接或连接还没有证实 (如，TP4)，则返回ENOTCONN。如果已建立连接，则从进程那里得到缓存的大小，并分配一块 mbuf来存储地址。发送PRU_PEERADDR请求给协议层来获取远端地址。将地址和地址的长度从内核的 mbuf中复制到

进程提供的缓存中。释放mbuf，并返回。

17.8 小结

本章中，我们讨论了六个修改插口功能的函数。插口选项由setsockopt和getsockopt函数处理。其他的选项，其中有些不仅仅用于插口，由 fcntl和ioctl处理。最后，通过getsockname和getpeername来获取连接信息。

习题

17.1 为什么选项受标准mbuf大小(MHLEN, 128个字节)的限制?

17.2 为什么图17-7中的最后一段代码能处理SO_LINGER选项?

17.3 图17-9中用来测试timeval结构的代码有些问题，因为tv->tv_sec * hz可能会溢出。请对这段代码作些修改来解决这个问题。

第18章　Radix树路由表

18.1　引言

由IP完成的路由选择是一种选路机制，它通过搜索路由表来确定从哪个接口把分组发送出去。它与选路策略(routing policy)不一样，选路策略是一组规则的集合，这些规则用来确定哪些路由可以编入到路由表中。　Net/3内核实现选路机制，而选路守护进程，典型地如routed或gated，实现选路策略。由于分组转发是频繁发生的 (一个繁忙的系统每秒要转发成百上千个分组)，相对而言，选路策略的变化要少些，因此路由表的结构必须能够适应这种情况。

关于路由选择的详细情况，我们分三章进行讨论：

- 本章将讨论Net/3分组转发代码所使用的Radix树路由表的结构。每次发送或转发分组时，IP都将查看该表(发送时分组需要查看该表，是因为IP必须决定哪个本地接口将接收该分组)。
- 第19章着重讨论内核与Radix树之间的接口函数以及内核与选路进程 (通常指实现选路策略的选路守护进程)之间交换的选路消息。进程可以通过这些消息来修改内核的路由表(添加路由、删除路由等)，并且当发生了一个异步事件，可能影响到路由策略 (如收到重定向，接口断开等)时，内核也通过这些消息来通知守护进程。
- 第20章给出了内核与进程之间交换选路消息时使用的选路插口。

18.2　路由表结构

在讨论Net/3路由表的内部结构之前，我们需要了解一下路由表中包含的信息类型。图 18-1是图1-17(作者以太网中的四个系统)的下半部分。

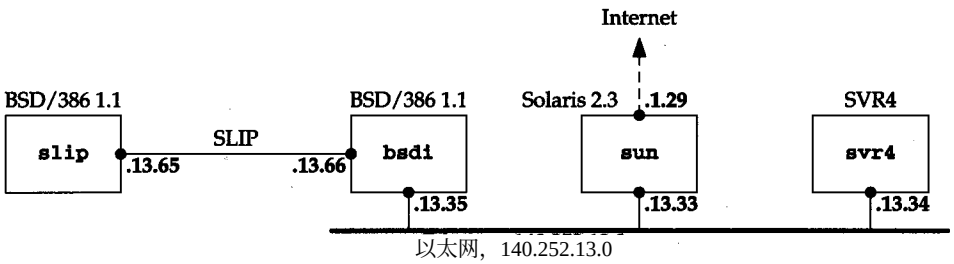


图18-1　路由表例子中使用子网

图18-2给出了图18-1中bsdi上的路由表。

为了能够更容易地看出每个表项中所设置的标志，我们已经对 netstat输出的“ Flags ”列进行了修改。

该表中的路由是按照下列过程添加的。其中，第 1、3、5、8和第9步是在系统的初始化阶段执行/etc/netstart shell脚本时完成的。

```
bsdi $ netstat -rn
Routing tables

Internet:
Destination       Gateway            Flags      Refs      Use   Interface
default           140.252.13.33      UG S          0        3   le0
127               127.0.0.1          UG S   R      0        2   lo0
127.0.0.1         127.0.0.1          U H            1       55   lo0
128.32.33.5       140.252.13.33      UGHS           2       16   le0
140.252.13.32     link#1             U    C         0        0   le0
140.252.13.33     8:0:20:3:f6:42     U H  L        11    55146   le0
140.252.13.34     0:0:c0:c2:9b:26    U H  L         0        3   le0
140.252.13.35     0:0:c0:6f:2d:40    U H  L         1       12   lo0
140.252.13.65     140.252.13.66      U H            0       41   sl0
224               link#1             U    C         0        0   le0
224.0.0.1         link#1             U H  L         0        5   le0
```

图18-2 主机bsdi 上的路由表

1) 默认路由是由route命令添加的。该路由通往主机sun(140 252 13 33)，主机sun拥有一条到Internet的PPP链路。

2) 到网络127的路由表项通常是由选路守护进程 (如gated)创建的，也可以通过/etc/netstart文件中的route命令将其添加到路由表中。该表项使得所有发往该网络的分组都将被环回驱动器 (图5-27)拒绝，但发往主机127.0.0.1的分组除外，因为对于该类分组，在下一步中添加的一条更特殊的路由将屏蔽本路由表项的作用。

3) 到环回接口(127.0.0.1)的表项是由ifconfig命令配置的。

4) 到vangogh.cs.berkeley.edu(128.32.33.5)的表项是用route命令手工创建的。该路由指定的路由器与默认路由所指定的相同 (都是140.252.13.33)。但是在拥有一条替代默认路由的通往特定主机的路由之后，我们就能把路由度量存储在该路由表项中。这些度量能够可以由管理者选择设置。每次 TCP建立一条到达目的主机的连接时都使用该度量，并且在连接关闭时，由TCP对其进行更新。我们将在图27-3中详细描述这些度量。

5) 接口le0的初始化是由 ifconfig命令完成的。该命令会在路由表中增加一条到140.252.13.32网络的表项。

6) 到以太网上另两台主机sun(140.252.13.33)和svr4(140.252.13.34)的路由表项是由ARP创建的，见第21章。它们都是临时路由，经过一段时间后，如果还未被使用，它们就会被自动删除。

7) 到本机(140.252.13.35)的表项是在第一次引用本机IP地址时创建的。该接口是一个环回，也就是说，任何发往本机IP地址的数据报将从内部反送回来。 4.4BSD中包含了自动创建该路由的新功能，见第21.13节。

8) 到主机140.252.13.65的表项是在ifconfig配置SLIP接口时创建的。

9) 通过以太网接口到达网络244的路由是由route命令添加的。

10) 到多播组224.0.0.1(所有主机的组， all-host group)的表项是Ping程序在连接224.0.0.1即“ Ping 224.0.0.1 ”时创建的。它也是一条临时路由，如果在一段时间内未被使用，就会被自动删除。

图18-2中的“ Flags ”列需要简单地说明一下。图18-25列出了所有可能的标志。

U 该路由存在。

G 该路由通向一个网关(路由器)。这种路由被称为间接路由。如果没有设置本标志，则表明路由的目的地与本机直接相连，称为直接路由。

H 该路由通往一台主机，也就是说，目的地址是一个完整的主机地址。如果没有设置本标志，则路由通往一个网络，目的地址是一个网络地址：一个网络号，或一个网络号与子网号的组合。netstat命令并不区分这一点，但每一条网络路由中都包含一个网络掩码，而主机路由中则隐含了一个全1的掩码。

S 该路由是静态的。图18-2中route命令创建的三个路由表项是静态的。

C 该路由可被克隆(clone)以产生新的路由。在本路由表中有两条路由设置了这个标志：一条是到本地以太网(140.252.13.32)的路由，ARP通过克隆该路由创建到以太网中其他特定主机的路由；另一条是到多播组224的路由，克隆该路由可以创建到特定多播组(如224.0.0.1)的路由。

L 该路由含有链路层地址。本标志应用于单播地址和多播地址。由ARP从以太网路由克隆而得到的所有主机路由都设置了本标志。

R 环回驱动器(为设有本标志的路由而设计的普通接口)将拒绝所有使用该路由的数据报。

添加带有拒绝标志的路由的功能由NET/2提供。它提供了一种简单的方法，来防止主机向外发送以网络127为目的地的数据报。参见习题6.6。

在4.3BSD Reno之前，内核将为IP地址维护两个不同的路由表：一个针对主机路由，另一个针对网络路由。对于给定的路由，将根据它的类型添加到相应的路由表中。默认路由被存储在网络路由表中，其目的地址是0.0.0.0。查找过程隐含了这样一种层次关系：首先查看主机路由表；如果找不到，则查找网络路由表；如果仍找不到，则查找默认路由。仅当三次查找都失败时，才认为目的地不可达。[Leffler et al. 1998]的第11.5节描述了一种带链表结构的hash表，该hash表同时用于Net/1中的主机路由表和网络路由表。

4.3BSD Reno [Sklower 1991]的变化主要与路由表的内部表示有关。这些变化允许相同的路由表函数访问不同协议栈的路由表，如OSI协议，它的地址是变长的，这一点与长度固定为32位的IP地址不同。为了提高查询速度，路由表的内部结构也做了变动。

Net/3路由表采用Patricia树结构[Sedgewick 1990]来表示主机地址和网络地址(Patricia支持“从文字数字的编码中提取信息的Patricia算法”)。待查找的地址和树中的地址都被看成比特序列。这样就可以用相同的函数来查找和维护不同类型的树，如：含有32 bit定长IP地址的树、含有48 bit定长XNS地址的树以及一棵含有变长OSI地址的树。

使用Patricia树构造路由表的思想应归功于Van Jacobson的[Sklower 1991]。

举个例子就可以很容易地描述出这个算法。查找路由表的目标就是为了找到一个最能匹配给定目标的特定地址。我们称这个给定的目标为查找键(search key)。所谓最能匹配的地址，也就是说，一个能够匹配的主机地址要优于一个能够匹配的网络地址；而一个能够匹配的网络地址要优于默认地址。

每条路由表项都有一个对应的网络掩码，尽管在主机路由中没有存储掩码，但它隐含了一个全1比特的掩码。我们对查找键和路由表项的掩码进行逻辑与运算，如果得到的值与该路由表项的目的地址相同，则称该路由表项是匹配的。对于某个给定的查找键，可能会从路由表中找到多条这样的匹配路由，所以在单个表同时包含网络路由和主机路由的情况下，我们

必须有效地组织该表，使得总能先找到那个更能匹配的路由。

让我们来讨论图18-3给出的例子。图中给出了两个查找键，分别是 127.0.0.1和127.0.0.2。为了更容易地说明逻辑与运算，图中同时给出了它们的十六进制值。图中给出的两个路由表项分别是主机路由127.0.0.1(它隐含了一个全1的掩码0xffffffff)和网络路由127.0.0.0(它的掩码是0xff000000)。

		查找键=127.0.0.1		查找键=127.0.0.2	
		主机路由	**网络路由**	**主机路由**	**网络路由**
1	查找键	7f000001	7f000001	7f000002	7f000002
2	路由表键	7f000001	7f000001	7f000001	7f000000
3	路由表掩码	ffffffff	ff000000	ffffffff	ff000000
4	1和3的逻辑与	7f000001	7f000000	7f000002	7f000000
	2和4相等？	相等	相等	不等	相等

图18-3 分别以127.0.0.1和127.0.0.2为查找键的路由表查找示例

其中两个路由表项都能够匹配查找键 127.0.0.1，这时路由表的结构必须确保能够先找到更能匹配该查找键的表项(127.0.0.1)。

图18-4给出了对应于图18-2的Net/3路由表的内部表示。执行带-A标志的netstat命令可以导出路由表的树型结构，图18-4就是根据导出的结果而建立的。

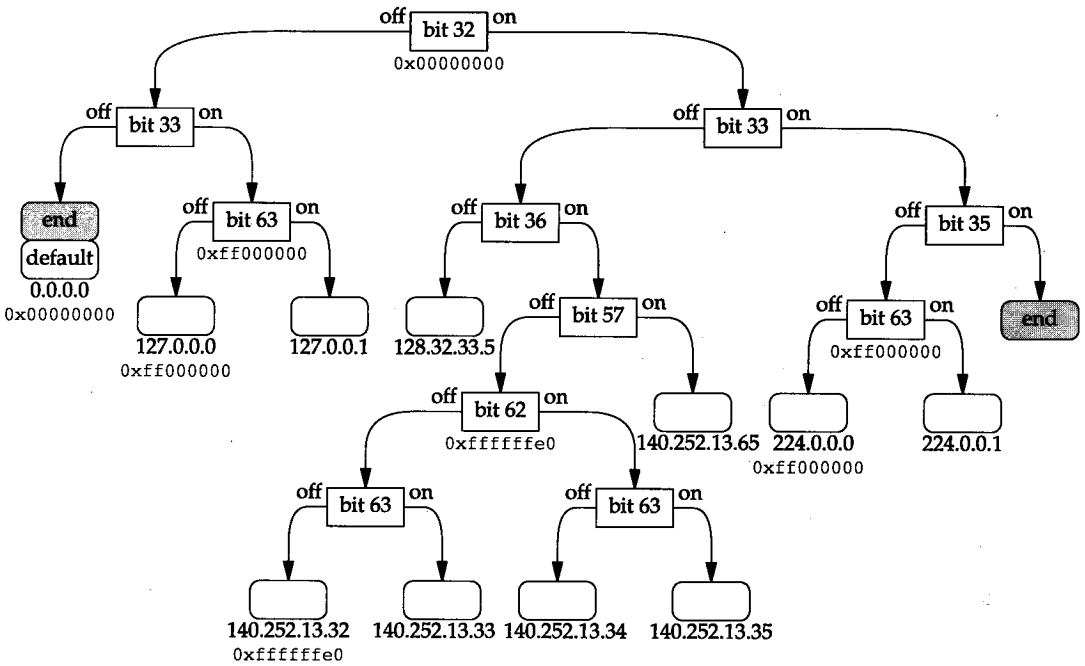


图18-4 对应于图18-2的Net/3路由表

标有“ end ”的两个阴影框是该树结构中带有特殊标志的叶结点，该标志代表树的端点。左边的那个拥有一个全0键，而右边的拥有一个全1键。左边的两个标有“ end ”和“ default ”的框垒在一起，这两个框有特殊的意义，它们与重复键有关，具体内容可参考 18.9节。

方角框被称为内部结点 (internal node)或简称为结点(node)，圆角框被称为叶子。每一个

内部结点对应于测试查找键的一个比特位，其左右各有一个分枝。每一个叶子对应于一个主机地址或者对应于一个网络地址。如果在叶子下面有一个十六进制数，那么这个叶子就对应于一个网络地址，该十六进制数就是叶子的网络掩码。如果在叶子下面没有十六进制的掩码，那么这个叶子就是一个主机地址，其隐含的掩码是0xffffffff。

有一些内部结点也含有网络掩码，在后面的学习中，我们将会了解这些掩码在回溯过程中是如何使用的。图中的每一个结点还包含了一个指向其父结点的指针 (没有在图中表示出来)，它能使树结构的回溯、删除及非递归操作更加方便。

比特比较是运用在插口地址结构上的，因此，在图 18-4中给出的比特位置是从插口地址结构中的起始位置开始算的。图 18-5给出了sockaddr_in结构中的比特位置。

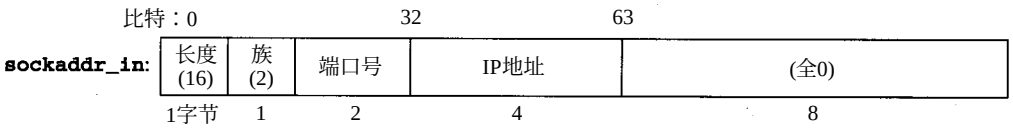


图18-5 Internet插口地址结构的比特位置

IP地址的最高位比特是比特 32，最低位是比特 63。此外还列出了长度是 16，地址族为2(AF_INET)，这两个数值在我们所列举的例子中将会遇到。

为了解释这些例子，还需要给出树中不同IP地址的比特表示形式。它们都被列在图18-6中，该图还给出了下面例子中要用到的一些其他 IP地址的比特表示形式。该图采用了加粗的字体来表示图18-4中分支点所对应的比特位置。

现在我们举一些特定的例子来说明路由表的查找过程是如何完成的。

1. 与主机地址匹配的例子

假定主机地址127.0.0.1是查找键 —— 待查找的目的地址。比特 32为0，因此，沿树顶点向左分支继续查找，到下一个结点。比特 33为1，因此，从该结点右分支继续查找，到下一个结点。比特63为1，因此，从右分支继续查找，到下一个结点。而下一个结点是个叶子，此时查找键(127.0.0.1)与叶子中的地址 (127.0.0.1)相比较。它们完全匹配，这样查找函数就可以返回该路由表项。

	32 bit IP地址								点分割表示
比特位置	**3333 2345**	**3333 6789**	4444 0123	4444 4567	4455 8901	5555 2345	5555 6789	**6666 0123**	
	0000	1010	0000	0001	0000	0010	0000	0011	**10.1.2.3**
	0111	0000	0000	0000	0000	0000	0000	0001	**112.0.0.1**
	0111	1111	0000	0000	0000	0000	0000	0000	**127.0.0.0**
	0111	1111	0000	0000	0000	0000	0000	0001	**127.0.0.1**
	0111	1111	0000	0000	0000	0000	0000	0011	**127.0.0.3**
	1000	0000	0010	0000	0010	0001	0000	0101	**128.32.33.5**
	1000	0000	0010	0000	0010	0001	0000	0110	**128.32.33.6**
	1000	1100	1111	1100	0000	1101	0010	0000	**140.252.13.32**
	1000	1100	1111	1100	0000	1101	0010	0001	**140.252.13.33**
	1000	1100	1111	1100	0000	1101	0010	0010	**140.252.13.34**
	1000	1100	1111	1100	0000	1101	0010	0011	**140.252.13.35**
	1000	1100	1111	1100	0000	1101	0100	0001	**140.252.13.65**
	1110	0000	0000	0000	0000	0000	0000	0000	**224.0.0.0**
	1110	0000	0000	0000	0000	0000	0000	0001	**224.0.0.1**

图18-6 图18-2和图18-4中IP地址的比特表示形式

2. 与主机地址匹配的例子

再假定查找键是地址140.252.13.35。比特32为1，因此，沿树顶点向右分支继续查找。比特33为0，比特36为1，比特57为0，比特62为1，比特63为1，因此，查找在底部标有140.252.13.35的叶子处终止。查找键与路由表键完全匹配。

3. 与网络地址匹配的例子

假定查找键是127.0.0.2。比特32为0，比特33为1，比特63为0，因此，查找在标有127.0.0.0的叶子处终止。查找键和路由表键并没有完全匹配，因此，需要进一步看它是不是一个能够匹配的网络地址。对查找键和网络掩码(0xff000000)进行逻辑与运算，得到的结果与该路由表键相同，即认为该路由表项能够匹配。

4. 与默认地址匹配的例子

假定查找键是10.1.2.3。比特32为0，比特33为0，因此，查找在标有“ end ”和“ default ”并带有重复键的叶子处终止。在这两个叶子中重复的路由表键是 0.0.0.0。查找键与路由表键值没有完全匹配，因此，需要进一步看它是不是一个能够匹配的网络地址。这种匹配运算要对每个含网络掩码的重复键都试一遍。第一个键 (标有end)没有网络掩码，可以跳过不查。第二个键(默认表项)有一个0x00000000的掩码。查找键和这个掩码进行逻辑与运算，所得结果和路由表键(0)相等，即认为该路由表项能够匹配。这样默认路由就被用做匹配路由。

5. 带回溯过程的与网络地址匹配的例子

假定查找键是127.0.0.3。比特32为0，比特33为1，比特63为1，因此，查找在标有127.0.0.1的叶子处终止。查找键和路由表键没有完全匹配。由于这个叶子没有网络掩码，无法进行网络掩码匹配的尝试。此时就要进行回溯。

回溯算法在树中向上移动，每次移动一层。如果遇到的内部结点含有掩码，则对查找关键字和该掩码进行逻辑与运算，得到一个键值，然后以这个键值作为新的查找键，在含该掩码的内部结点为开始的子树中进行另一次查找，看是否能找到匹配的结点。如果找不到，则回溯过程继续沿树上移，直到到达树的顶点。

在这个例子中，查找上移一层到达比特 63对应的结点，该结点含有一个掩码。于是对查找键和掩码(0xff000000)进行逻辑与运算，得到一个新的查找键，其值为 127.0.0.0。然后从该结点开始查找127.0.0.0。比特63为0，因此，沿左分支到达标有 127.0.0.0的叶子上。用新的查找键与路由表键相比较，它们是相等的，因此认为这个叶子是匹配的。

6. 多层回溯的例子

假定查找键是 112.0.0.1。比特32为0，比特33为1，比特63为1，因此，查找在标有127.0.0.1的叶子处终止。该键与查找键不相等，并且路由表项中没有网络掩码，因此需要进行回溯。

查找过程向上移动一层，到达比特 63对应的结点，该结点含有一个掩码。对查找关键字和该掩码(0xff000000)进行逻辑与运算，然后再从这个结点开始进一步查找。在新的查找键中比特63为0，因此，沿左分支到达标有127.0.0.0的叶子。比较之后发现逻辑与运算得到的查找键(112.0.0.0)和路由表键并不相等。

因此继续向上回溯一层，从比特63对应的结点上移到比特33对应的结点。但这个结点没有掩码，再继续向上回溯。到达的下一层是树的顶点 (比特32)，它有一个掩码。对查找键(112.0.0.1)和该掩码(0x00000000)进行逻辑与运算后，从该点开始一个新的查找。在新的查找

键中，比特32为0，比特33也为0，因此，查找在标有“ end ”和“ default ”的叶子处结束。通过与重复键列表中的每一项进行比较，发现默认键与新的查找键相匹配，因此采用默认路由。

从这个例子中可以知道，如果在路由表中存在默认路由，那么当回溯最终到达树的顶点时，它的掩码为全0比特，这使得查找向树中最左边叶子的方向进行搜索，最终与默认路由相匹配。

7. 带回溯和克隆过程、并与主机地址相匹配的例子

假定查找键是224.0.0.5。比特32为1，比特331，比特35为0，比特63为1，因此，查找在标有224.0.0.1的叶子处结束。路由表的键值和查找关键字并不相等，并且该路由表项不包含网络掩码，因此要进行回溯。

回溯向上移动一层，到达比特 63对应的结点。这个结点含有掩码 `0xff000000`，因此，对查找键和该掩码进行逻辑与运算，产生一个新的查找键，即 224.0.0.0。再从这个结点开始一次新的查找。在新的查找键中比特 63为0，于是沿左分支到达标有 224.0.0.0的叶子。这个路由表键和逻辑与运算得到的查找键相匹配，因此这个路由表项是匹配的。

该路由上设置了“ 克隆 ”标志 (见图18-2)，因此，以224.0.0.5为地址创建一个新的叶子。新的路由表项是：

```
Destination    Gateway      Flags      Refs      Use    Interface
224.0.0.5      link#1       UHL         0         0        le0
```

图18-7从比特35对应的结点开始，给出了图 18-4中路由表树右边部分的新的排列。注意，无论何时向树中添加新的叶子，都需要两个结点：一个作为叶子，另一个作为测试某一位比特的内部结点。

新创建的表项就被返回给查找 224.0.0.5的调用者。

8. 大图

图18-8是一张比较大的图，它描述了所有涉及到的数据结构。该图的底部来自于图 3-32。

现在我们将解释图中的几个要点，在后面，本章还将给出详细的阐述。

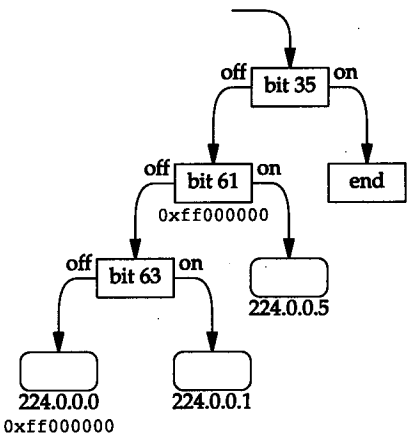


图18-7 插入224.0.0.5路由表项后图18-6的改动

- `rt_tables`是指向`radix_node_head`结构的指针数组。每一个地址族都有一个数组单元与之对应。`rt_tables[AF_INET]`指向Internet路由表树的顶点。
- `radix_node_head`结构包含三个`radix_node`结构。这三个结构是在初始化路由树时创建的，中间的是树的顶点。它对应于图 18-4中最上端标有“ bit 32 ”的结点框。三个`radix_node`结构中的第一个是图 18-4中最左边的叶子(与默认路由共享的重复)，第三个结构是最右边的叶子。在一个空的路由表中，就只包含这三个 `radix_node`结构，我们将会看到`rn_inithead`函数是如何构建它们的。
- 全局变量`mask_rnhead`也指向一个`radix_node_head`结构。它是包含了所有掩码的一棵独立树的首部结构。观察图 18-4中给出的八个掩码可知，有一个掩码重复了四次，有两个掩码重复了一次。通过把掩码放在一棵单独的树中，可以做到对每一个掩码只需要维护它的一个备份即可。

• 路由表树是用 rtentry结构创建的，在图 18-8中，有两个 rtentry结构。每一个 rtentry结构包含两个radix_node结构，因为每次向树中插入一个新的路由时，都需要两个结点：一个是内部结点，对应于某一位测试比特；另一个是叶子，对应于一个主机路由或一个网络路由。在每一个 rtentry结构中，给出了内部结点对应的要测试的那位比特以及叶子中所包含的地址。

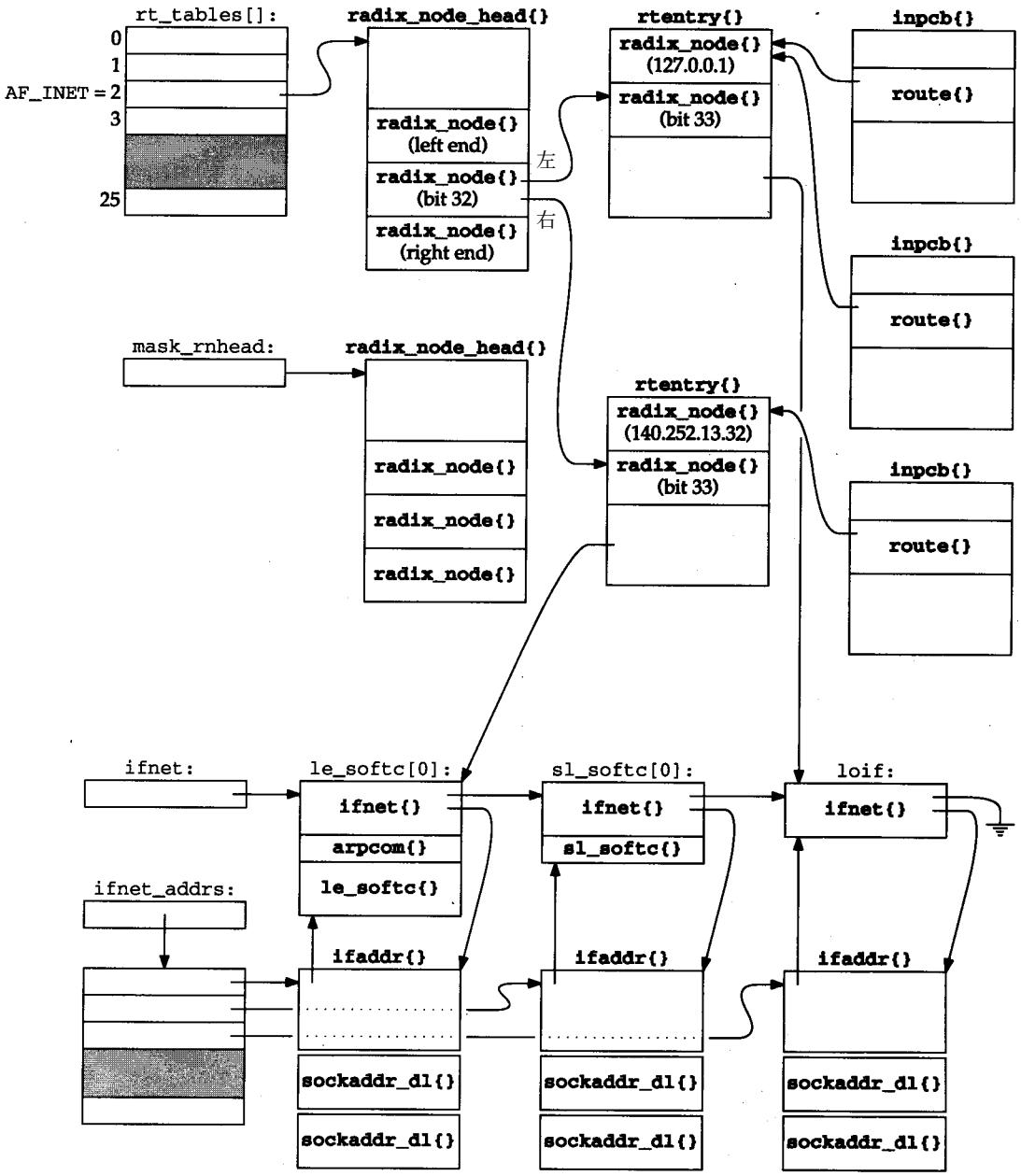


图18-8 路由表中涉及的数据结构

rtentry结构中的其余部分是该路由的一些其他重要信息。虽然我们只给出了该结构中的一个指向ifnet结构的指针，但在这个结构中还包含了指向 ifaddr结构的指针、该路由的

标志、指向另一个rtentry结构的指针(如果该路由是一个非直接路由)和该路由的度量等等。

- 存在于每一个UDP和TCP插口(图22-1)中的协议控制块PCB(见第22章)中包含了一个指向rtentry结构的route结构。每次发送一个IP数据报时，UDP和TCP输出函数都传递一个指向PCB中route结构的指针，作为调用ip_output的第三个参数。使用相同路由的PCB都指向相同的路由表项。

18.3 选路插口

在4.3BSD Reno的路由表做了变动后，路由子系统和进程间的交互过程也要做出变动，这就引出了选路插口(routing socket)的概念。在4.3BSD Reno之前，是由进程(如route命令)通过发出定长的ioctl来修改路由表的。4.3BSD Reno采用新的PF_ROUTE域把它改变成一种更为通用的消息传递模式。进程在PF_ROUTE域中创建一个原始插口(raw socket)，就能够向内核发送选路消息，以及从内核接收选路消息(如重定向或来自于内核的其他的异步通知)。

图18-9给出了12种不同类型的选路消息。消息类型位于rt_msghdr结构(图19-16)中的rtm_type字段。进程只能发送其中的5种消息(写入到选路插口中)，但可以接收全部的12种消息。

我们将在第19章给出这些选路消息的详细讨论。

rtm_type	发往内核？	从内核发出？	描　述	结构类型
RTM_ADD	•	•	添加路由	rt-msghdr
RTM_CHANGE	•	•	改变网关、度量或标志	rt-msghdr
RTM_DELADDR		•	从接口中删除地址	ifa-msghdr
RTM_DELETE	•	•	删除路由	rt-msghdr
RTM_GET	•	•	报告度量及其他路由信息	rt-msghdr
RTM_IFINFO		•	接口打开或关闭等	rt-msghdr
RTM_LOCK	•	•	锁定指明的度量	rt-msghdr
RTM_LOSING		•	内核怀疑某路由无效	rt-msghdr
RTM_MISS		•	查找这个地址失败	rt-msghdr
RTM_NEWADDR		•	接口中添加了地址	ifa-msghdr
RTM_REDIRECT		•	内核得知要使用不同的路由	rt-msghdr
RTM_RESOLVE		•	请求将目的地址解析成链路层地址	rt-msghdr

图18-9 通过选路插口交换的消息类型

18.4 代码介绍

路由选择中使用的各种结构和函数是通过五个C文件和三个头文件来定义的。图18-10列出了这些文件。

通常，前缀rn_表示radix结点函数，这些函数可以对Patricia树进行查找和操作，前缀raw_表示路由控制块函数，rout_、rt_和rt这三个前缀表示常用的选路函数。

尽管有的文件和函数以raw为前缀，但在所有的选路章节中我们仍使用术语选路控制块(routing control block)而不是原始控制块。这是为了防止与第32章中讨论的原始IP控制块及其函数相混淆。虽然原始控制块及相关函数不仅仅用于Net/3中的选路插口(使用这些结构和函数的原始OSI协议之一)，但是本书中我们只用做PF_ROUTE域中的选路插口。

文　件	描　述
net/radix.h	radix结点定义
net/raw_cb.h	选路控制块定义
net/route.h	选路结构
net/radix.c	radix结点(Patricia树)函数
net/raw_cb.c	选路控制块函数
net/raw_usrreq.c	选路控制块函数
net/route.c	选路函数
net/rtsock.c	选路插口函数

图18-10　本章中讨论的文件

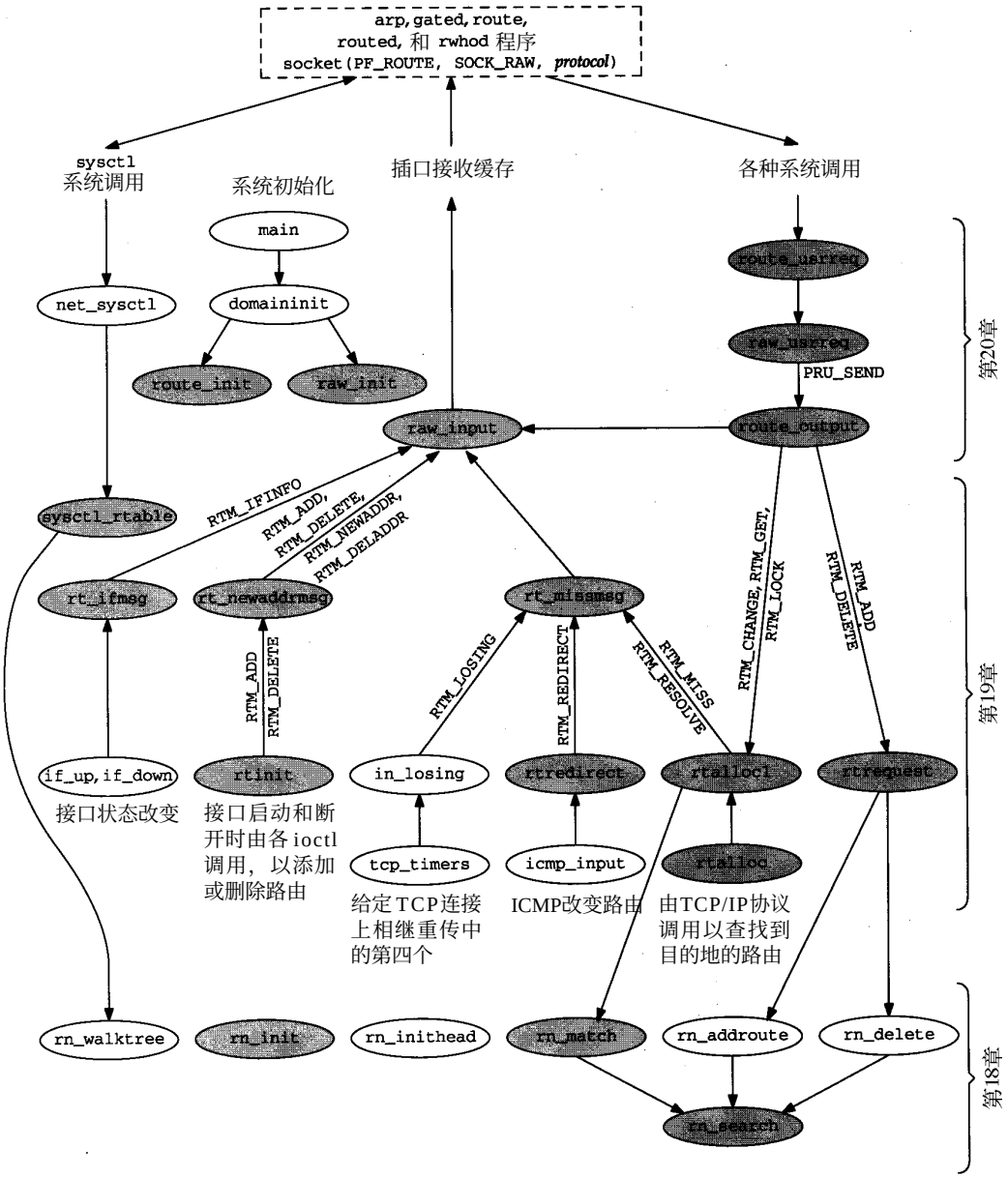


图18-11　各选路函数之间的关系

图18-11给出了一些基本的选路函数，并表示了它们之间的相互关系。其中带阴影的椭圆是在本章和下面两章中要涉及的内容。在图中，我们还给出了每种类型的选路消息 (共12种)的产生之处。

rtalloc函数是由 Internet协议调用的，用于查找到达指定目的地的路由。在ip_rtaddr、ip_forward、ip_output和ip_setmoptions函数中都已出现过rtalloc，在后面介绍的in_pcbconnect和tcp_mss函数中也将会遇到它。

图18-11还给出了在选路域中创建插口的五个典型程序：

- arp处理ARP高速缓存，该ARP高速缓存被存储在Net/3的IP路由表中(见第21章)；
- gated和routed是选路守护进程，它们与其他路由器进行通信。当选路环境发生变化时(指路由器及链路断开或连通)，对内核的路由表进行操作；
- route通常是由启动脚本或系统管理员执行的一个程序，用于添加或删除路由；
- rwhod在启动时会调用一个选路sysctl来测定连接的接口。

当然，任何进程(具有超级用户的权限)都能打开一个选路插口向选路子系统发送或从中接收消息；在图18-11中，我们只给出了一些常用的系统程序。

18.4.1 全局变量

图18-12列出了在三个有关路由选择的章节中介绍的全局变量。

变　量	数据类型	描　述
rt_tables	struct radix_node_head *[]	路由表表头指针的数组
mask_rnhead	struct radix_node_head *	指向掩码表表头的指针
rn_mkfreelist	struct radix_mask *	可用radix_mask结构的链表表头
max_keylen	int	以字节为单位的路由表键值的最大长度
rn_zeros	char *	长为max_keylen、值为全零比特的数组
rn_ones	char *	长为max_keylen、值为全1比特的数组
maskedKey	char *	长为max_keylen、掩码过的查找键数组
rtstat	struct rtstat	路由选择统计(图18-13)
rttrash	int	未释放的非表中路由的数目
rawcb	struct rawcb	选路控制块双向链表表头
raw_recvspace	u_long	选路插口接收缓冲区的默认大小，8192字节
raw_sendspace	u_long	选路插口发送缓冲区的默认大小，8192字节
route_cb	struct route_cb	选路插口监听器的数目，每个协议的数目及总的数目
route_dst	struct sockaddr	保存选路消息中目的地址的临时变量
route_src	struct sockaddr	保存选路消息中源地址的临时变量
route_proto	struct sockproto	保存选路消息中协议的临时变量

图18-12　在三个有关选路的章节中介绍的全局变量

18.4.2 统计量

图18-13列举了一些路由选择统计量，它们是在全局结构 rtstat中维护的。

在代码的处理中，我们可以发现计数器是怎样增加的。这些计数器在 SNMP中并未使用。

图18-14给出了netstat -rs命令输出的一些统计数据的样例，该命令用于显示rtstat结构。

rtstat成员	描 述	在SNMP中的使用
rts_badredirect	无效重定向调用的数目	
rts_dynamic	由重定向创建的路由数目	
rts_newgateway	由重定向修改的路由数目	
rts_unreach	查找失败的次数	
rts_wildcard	由通配符匹配的查找次数(从未使用)	

图18-13 在rtstat 结构中维护的路由选择统计数据

netstat-rs的输出	rtstat 成员
1029 bad routing redirects	rts_badredirect
0 dynamically created routes	rts_dynamic
0 new gateways due to redirects	rts_newgateway
0 destinations found unreachable	rts_unreach
0 uses of a wildcard route	rts-wildcard

图18-14 路由选择统计数据样例

18.4.3 SNMP变量

图18-15给出了名为ipRouteTable的IP路由表以及相应的内核变量。

IP路由表, index = <*ipRouteDest*>		
SNMP变量	变 量	描 述
ipRouteDest	rt_key	IP目的地址。值为0.0.0.0时，代表默认路由
ipRouteIfIndex	rt_ifp, if_index	接口号：ifIndex
ipRouteMetric1	-1	基本的路由度量。其含义取决于选路协议的值 (ipRouteProto)。值为-1，表示没有使用
ipRouteMetric2	-1	可选的路由度量
ipRouteMetric3	-1	可选的路由度量
ipRouteMetric4	-1	可选的路由度量
ipRouteNextHop	rt_gateway	下一跳路由器的IP地址
ipRouteType	(见正文)	路由类型：1=其他，2=无效路由，3=直接的，4=间接的
ipRouteProto	(见正文)	路由协议：1=其他，4=ICMP重定向，8=RIP, 13=OSPF, 14= BGP 等
ipRouteAge	(未实现)	从路由最后一次被修改或被确定为正确时起的秒数
ipRouteMask	rt_mask	在和ipRouteDest比较前，与目的主机IP地址进行逻辑与运算的掩码
ipRouteMetric5	-1	可选的路由度量
ipRouteInfo	NULL	本选路协议特定的MIB定义的引用

图18-15 IP路由表：ipRouteTable

如果在rt_flags中将标志RTF_GATEWAY置位，则该路由就是远程的，ipRouteType等于4；否则该路由就是直达的，ipRouteType值为3。对于ipRouteProto，如果将标志RTF_DYNAMIC或RTF_MODIFIED置位，则该路由就是由ICMP来创建或修改的，值为4，否则为其他情况，其值为1。最后，如果rt_mask指针为空，则返回的掩码就是全1(即主机路由)。

18.5 Radix结点数据结构

在图18-8中可以发现每一个路由表的表头都是一个radix_node_head结构，而选路树中所有的结点(包括内部结点和叶子)都是radix_node结构。radix_node_head结构如图18-16所示。

radix.h

```
91 struct radix_node_head {
92     struct radix_node *rnh_treetop;
93     int     rnh_addrsize;        /* (not currently used) */
94     int     rnh_pktsize;         /* (not currently used) */
95     struct radix_node *(*rnh_addaddr)    /* add based on sockaddr */
96             (void *v, void *mask,
97              struct radix_node_head * head, struct radix_node nodes[]);
98     struct radix_node *(*rnh_addpkt)     /* add based on packet hdr */
99             (void *v, void *mask,
100             struct radix_node_head * head, struct radix_node nodes[]);
101    struct radix_node *(*rnh_deladdr)    /* remove based on sockaddr */
102            (void *v, void *mask, struct radix_node_head * head);
103    struct radix_node *(*rnh_delpkt)     /* remove based on packet hdr */
104            (void *v, void *mask, struct radix_node_head * head);
105    struct radix_node *(*rnh_matchaddr)      /* locate based on sockaddr */
106            (void *v, struct radix_node_head * head);
107    struct radix_node *(*rnh_matchpkt)   /* locate based on packet hdr */
108            (void *v, struct radix_node_head * head);
109    int     (*rnh_walktree)      /* traverse tree */
110            (struct radix_node_head * head, int (*f) (), void *w);

111    struct radix_node rnh_nodes[3];      /* top and end nodes */
112 };
```

radix.h

图18-16 radix_node_head 结构：每棵选路树的顶点

92 rnh_treetop指向路由树顶端的radix_node结构。可以看到radix_ node_head结构的最后一项分配了三个radix_node结构，其中中间的那个被初始化成树的顶点 (图18-8)。

93-94 rnh_addrsize和rnh_pktsize目前未被使用。

> rnh_addrsize是为了能够方便地将路由表代码导入到系统中去，因为系统的插口地址结构中没有标识其长度的字节。rnh_pktsize是为了能够利用radix结点机制直接检查分组头结构中的地址，而无需把该地址拷贝到某个插口地址结构中去。

95-110 从rnh_addaddr到rnh_walktree是七个函数指针，它们所指向的函数将被调用以完成对树的操作。如图 18-17所示，rn_inithead仅初始化了其中的四个指针，剩下的三个指针在Net/3中未被使用。

成 员	被rn_inithead初始化为
rnh_addaddr	*rn_addroute*
rnh_addpkt	*NULL*
rnh_deladdr	*rn_delete*
rnh_delpkt	*NULL*
rnh_matchaddr	*rn_match*
rnh_matchpkt	*NULL*
rnh_walktree	*rn_walktree*

图18-17 在radix_node_head 结构中的七个函数指针

111-112 图18-18给出了组成树中结点的radix_node结构。在图18-8中，我们可以发现，在radix_node_head中分配了三个这样的radix_node结构，而在每一个rtentry结构中分配了两个radix_node结构。

41-45 前五个成员是内部结点和叶子都有的成员。后面是一个 union：如果结点是叶子，那么它定义了三个成员；如果是内部结

点，那么它定义了另外不同的三个成员。由于在 Net/3代码中经常使用union中的这些成员，因此，用一组#define语句定义它们的简写形式。

41-42 rn_mklist是该结点掩码链表的表头。我们将在18.9节中描述该字段。rn_p指向该结点的父结点。

43 如果rn_b值大于或者等于零，那么该结点为内部结点；否则就是叶子。对于内部结点来说，rn_b就是要测试的比特位置：例如，在图 18-4中，树的顶结点的rn_b值为32。对于叶子来说，rn_b是负的，它的值等于- 1减去网络掩码索引(index of the network mask)。该索引是指掩码中出现的第一个零的比特位置。图 18-19给出了图18-4中掩码的索引。

radix.h

```
40 struct radix_node {
41     struct radix_mask *rn_mklist;   /* list of masks contained in subtree */
42     struct radix_node *rn_p;     /* parent pointer */
43     short    rn_b;               /* bit offset; -1-index(netmask) */
44     char     rn_bmask;           /* node: mask for bit test */
45     u_char   rn_flags;           /* Figure 18.20 */
46     union {
47         struct {                 /* leaf only data: rn_b < 0 */
48             caddr_t rn_Key;      /* object of search */
49             caddr_t rn_Mask;     /* netmask, if present */
50             struct radix_node *rn_Dupedkey;
51         } rn_leaf;
52         struct {                 /* node only data: rn_b >= 0 */
53             int     rn_Off;      /* where to start compare */
54             struct radix_node *rn_L;     /* left pointer */
55             struct radix_node *rn_R;     /* right pointer */
56         } rn_node;
57     } rn_u;
58 };

59 #define rn_dupedkey rn_u.rn_leaf.rn_Dupedkey
60 #define rn_key      rn_u.rn_leaf.rn_Key
61 #define rn_mask     rn_u.rn_leaf.rn_Mask
62 #define rn_off      rn_u.rn_node.rn_Off
63 #define rn_l        rn_u.rn_node.rn_L
64 #define rn_r        rn_u.rn_node.rn_R
```

radix.h

图18-18 radix_node 结构：路由树的结点

	32 bit IP掩码(比特32 - 36)								索引	rn_b
	3333 2345	3333 6789	4444 0123	4444 4567	4455 8901	5555 2345	5555 6789	6666 0123		
00000000:	0000	0000	0000	0000	0000	0000	0000	0000	0	–1
ff000000:	1111	1111	0000	0000	0000	0000	0000	0000	40	–41
fffffffe0:	1111	1111	1111	1111	1111	1111	1110	0000	59	–60

图18-19 掩码索引的例子

我们可以发现，其中的全0掩码的索引是特殊处理的：它的索引是 0，而不是32。

44 内部结点的rn_bmask是个单字节的掩码，用于检测相应的比特位是 0还是1。在叶子中它的值为0。很快我们将会看到如何运用成员rn_bmask和成员rn_off。

45 图18-20给出了成员rn_flags的三个值。

常 量	描 述
RNF_ACTIVE	该结点是活的(alive)(针对rtfree)
RNF_NORMAL	该叶子含有正常路由(目前未被使用)
RNF_ROOT	该叶子是树的根叶子

图18-20 rn_flags 的值

RNF_ROOT标志只有在radix_node_head结构中的三个radix结点(树的顶结点、左端结点和右端结点)中才能设置。这三个结点不能从路由树中删除。

48-49 对于叶子而言，rn_key指向插口地址结构，rn_mask指向保存掩码的插口地址结构。如果rn_mask为空，则其掩码为隐含的全 1值(即，该路由指向某个主机而不是某个网络)。

图18-21例举了一个与图18-4中的叶子140.252.13.32相对应的radix_node结构的例子。

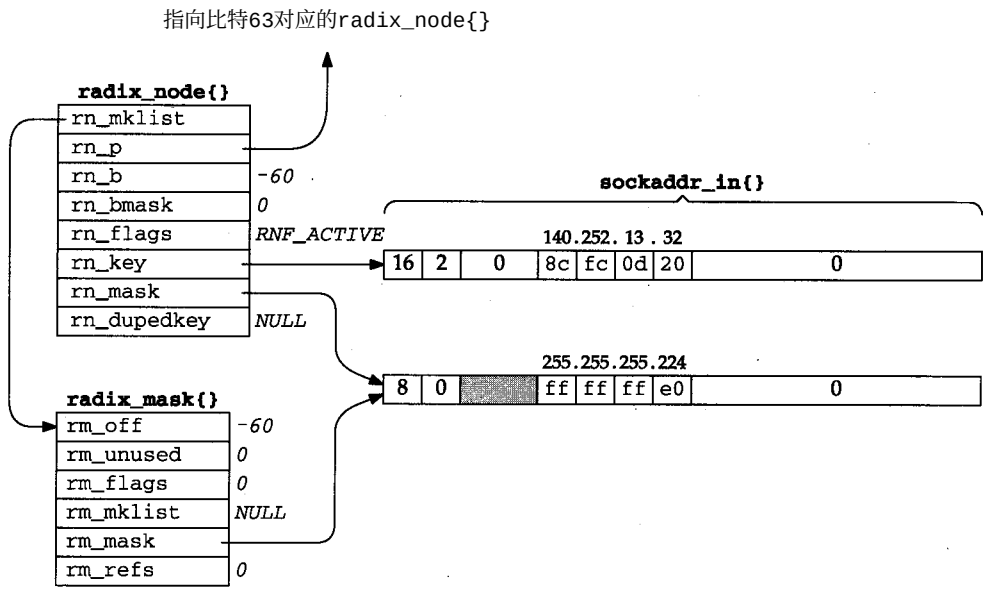


图18-21 与图18-4中的叶子140.252.13.32相对应的radix_node 结构

该例子中还给出了图18-22中描述的radix_mask结构。我们把它的宽度略微缩小了一些，以区分于radix_node结构；这两种结构在后面的很多图例中都会遇到。有关 radix_mask结构的作用将在18.9节中阐述。

值为-60的rn_b相对应的索引为59。rn_key指向一个sockaddr_in结构，它的长度值为16，地址族值为2(AF_INET)。由rn_mask和rm_mask指向的掩码结构所含的长度值为 8，地址族值为0(该族为AF_UNSPEC，但我们从未使用它)。

50-51 当有多个叶子的键值相同时，使用rn_dupedkey指针。具体内容将在18.9节中阐述。

52-58 有关rn_off的内容将在18.8节中阐述。rn_l和rn_r是该内部结点的左、右指针。

图18-22给出了radix_mask结构的定义。

76-83 该结构中包含了一个指向其掩码的指针：rm_mask，实际上是一个保存掩码的插口地址结构的指针。每一个 radix_node结构对应一个radix_mask结构的链表，这就允许每

个结点包含多个掩码：成员 rn_mklist指向链表的第一个结点，然后每个结构的成员rm_mklist指向链表的下一个结点。该结构的定义同时声名了全局变量 rn_mkfreelist，它是可用的radix_mask结构链表的表头。

radix.h

```
76 extern struct radix_mask {
77     short    rm_b;              /* bit offset; -1-index(netmask) */
78     char     rm_unused;         /* cf. rn_bmask */
79     u_char   rm_flags;          /* cf. rn_flags */
80     struct radix_mask *rm_mklist;   /* more masks to try */
81     caddr_t rm_mask;            /* the mask */
82     int      rm_refs;           /* # of references to this struct */
83 }      *rn_mkfreelist;
```

radix.h

图18-22 radix_mask 结构

18.6 选路结构

访问内核路由信息的关键之处是：

1) rtalloc函数，用于查找通往目的地的路由；

2) route结构，它的值由rtalloc函数填写；

3) route结构所指向的rtentry结构。

图18-8给出了UDP和TCP(参见第22章)中使用的协议控制块(PCB)，其中包含一个route结构，见图18-23。

route.h

```
46 struct route {
47     struct rtentry *ro_rt;      /* pointer to struct with information */
48     struct sockaddr ro_dst;     /* destination of this route */
49 };
```

route.h

图18-23 route 结构

ro_dst被定义成一个一般的插口地址结构，但对于 Internet协议而言，它就是一个sockaddr_in结构。注意，对这种结构类型的绝大多数引用都是一个指针，而 ro_dst是该结构的一个实例而非指针。

这里，我们有必要回顾一下图8-24。从该图可以得知，每次发送IP数据报时，这些路由是如何使用的。

- 如果调用者传递了一个route结构的指针，那么就使用该结构。否则，就要用一个局部route结构，其值设置为0(设置ro_rt为空指针)。UDP和TCP把指向它们的PCB中route结构的指针传递给ip_output。
- 如果route结构指向一个rtentry结构(ro_rt指针为非空)，同时所引用的接口仍然有效；而且如果route结构中的目的地址与IP数据报中的目的地址相等，那么该路由就被使用。否则，目的主机的 IP地址将会设置在插口地址结构 so_dst中，并且调用rtalloc来查找一条通向该目的主机的路由。在TCP链接中，数据报的目的地址始终是路由的目的地址，不会发生变化，但是 UDP应用可以通过sendto每次都把数据报发送到不同的目的地。
- 如果rtalloc返回的ro_rt是个空指针，则表明找不到路由，并且 ip_output返回一

个差错。

- 如果在rtentry结构中设有RTF_GATEWAY标志，那么该路由为非直接路由(参见图18-2中的G标志)。接口输出函数的目的地址(dst)就变成网关的IP地址，即rt_gateway成员，而不是IP数据报的目的地地址。

图18-24给出了rtentry结构的定义。

route.h

```
83 struct rtentry {
84     struct radix_node rt_nodes[2];  /* a leaf and an internal node */
85     struct sockaddr *rt_gateway;    /* value associated with rn_key */
86     short   rt_flags;           /* Figure 18.25 */
87     short   rt_refcnt;          /* #held references */
88     u_long  rt_use;             /* raw #packets sent */
89     struct ifnet *rt_ifp;       /* interface to use */
90     struct ifaddr *rt_ifa;      /* interface address to use */
91     struct sockaddr *rt_genmask;    /* for generation of cloned routes */
92     caddr_t rt_llinfo;          /* pointer to link level info cache */
93     struct rt_metrics rt_rmx;   /* metrics: Figure 18.26 */
94     struct rtentry *rt_gwroute; /* implied entry for gatewayed routes */
95 };
96 #define rt_key(r)    ((struct sockaddr *)((r)->rt_nodes->rn_key))
97 #define rt_mask(r)   ((struct sockaddr *)((r)->rt_nodes->rn_mask))
```

route.h

图18-24 rtentry 结构

83-84 在该结构中包含有两个radix_node结构。正如我们在图18-7的例子中所提到的，每次向路由树添加一个新叶子的同时也要添加一个新的内部结点。rt_nodes[0]为叶子，rt_nodes[1]为内部结点。在图18-24最后的两个#define语句提供了访问该叶结点的键和掩码的简写形式。

86 图18-25给出了储存在rt_flags中的各种常量以及在图18-2的“Flags”列中由netstat输出的相应字符。

常 量	netstat标志	描 述
RTF_BLACKHOLE		无差错的丢弃分组(环回驱动器:图5-27)
RTF_CLONING	C	使用中产生新的路由(由ARP使用)
RTF_DONE	d	内核的证实，表示消息处理完毕
RTF_DYNAMIC	D	(由重定向)动态创建
RTF_GATEWAY	G	目的主机是一个网关(非直接路由)
RTF_HOST	H	主机路由(否则，为网络路由)
RTF_LLINFO	L	当rt_llinfo指针无效时，由ARP设置
RTF_MASK	m	子网掩码存在(未使用)
RTF_MODIFIED	M	(由重定向)动态修改
RTF_PROTO1	1	协议专用的路由标志
RTF_PROTO2	2	协议专用的路由标志(ARP使用)
RTF_REJECT	R	有差错的丢弃分组(环回驱动器:图5-27)
RTF_STATIC	S	人工添加的路由(route程序)
RTF_UP	U	可用路由
RTF_XRESOLVE	X	由外部守护进程解析名字(用于X.25)

图18-25 rt_flags 的值

netstat不输出RTF_BLACKHOLE标志。两个标志为小写字符的常量，RTF_DONE和RTF_MASK，在路由消息中使用，但通常并不储存在路由表项中。

85 如果设置了RTF_GATEWAY标志，那么rt_gateway所含的插口地址结构的指针就指向网关的地址(即网关的IP地址)。同样，rt_gwroute就指向该网关的rtentry。后一个指针在ether_output(图4-15)中用到。

87 rt_refcnt是一个计数器，保存正在使用该结构的引用数目。在19.3节的最后部分将具体描述该计数器。在图18-2中，该计数器在“Refs”列输出。

88 当分配该结构存储空间时，rt_use被初始化为0。在图8-24中，可发现每次利用该路由输出一份IP数据报时，其值会随之递增。该计数器的值在图18-2的“Use”栏中输出。

89-90 rt_ifp和rt_ifa分别指接口结构和接口地址结构。在图6-5中曾指出一个给定的接口可以有多个地址，因此，rt_ifa是必需的。

92 rt_llinfo指针允许链路层协议在路由表项中储存该协议专用的结构指针。该指针通常与RTF_LLINFO标志一起使用。图21-1描述了ARP如何使用该指针。

route.h

```
54 struct rt_metrics {
55      u_long  rmx_locks;          /* bitmask for values kernel leaves alone */
56      u_long  rmx_mtu;            /* MTU for this path */
57      u_long  rmx_hopcount;       /* max hops expected */
58      u_long  rmx_expire;         /* lifetime for route, e.g. redirect */
59      u_long  rmx_recvpipe;       /* inbound delay-bandwith product */
60      u_long  rmx_sendpipe;       /* outbound delay-bandwith product */
61      u_long  rmx_ssthresh;       /* outbound gateway buffer limit */
62      u_long  rmx_rtt;            /* estimated round trip time */
63      u_long  rmx_rttvar;         /* estimated RTT variance */
64      u_long  rmx_pksent;         /* #packets sent using this route */
65 };
```

route.h

图18-26 rt_metrics 结构

93 图18-26给出了rt_metrics结构，rtentry结构含有该结构。图27-3显示了TCP使用了该结构的六个成员。

54-65 rmx_locks是一个比特掩码，由它告诉内核后面的八个度量中的哪些禁止修改。该比特掩码的值在图20-13中给出。

ARP(参见第21章)把rmx_expire用作每一个ARP路由项的定时器。与rmx_expire的注释不同的是，rm_expire不是用作重定向的。

图18-28概括了我们上面所阐述的各种结构和这些结构之间的关系，以及所引用的各种插口地址结构。图中给出的rtentry是图18-2中到128.32.33.5的路由。包含在rtentry中的另一个radix_node对应于图18-4中位于该结点正上方的测试比特36的内部结点。第一个ifaddr所指的两个sockaddr_dl结构如图3-38所示。另外，从图6-5中也可注意到ifnet结构包含在le_softc结构中，第二个ifaddr结构包含在in_ifaddr结构中。

18.7 初始化：route_init和rtable_init函数

路由表的初始化过程并非是一目了然的，我们需要回顾一下第7章中的domain结构。在描述这些函数调用之前，图18-27给出了各协议族中与domain结构相关的字段。

成　员	OSI值	Internet值	选路值	Unix值	XNS值	注　释
dom_family	*AF_ISO*	*AF_INET*	*PF_ROUTE*	*AF_UNIX*	*AF_NS*	
dom_init	*0*	*0*	*route_init*	*0*	*0*	
dom_rtattach	*rn_inithead*	*rn_inithead*	*0*	*0*	*rn_inithead*	比特
dom_rtoffset	*48*	*32*	*0*	*0*	*16*	字节
dom_maxrtkey	*32*	*16*	*0*	*0*	*16*	

图18-27　domain 结构中与路由选择有关的成员

PF_route域是唯一具有初始化函数的域。同样，只有那些需要路由表的域才有dom_rtattach函数，并且该函数总是rn_inithead。选路域和Unix域并不需要路由表。

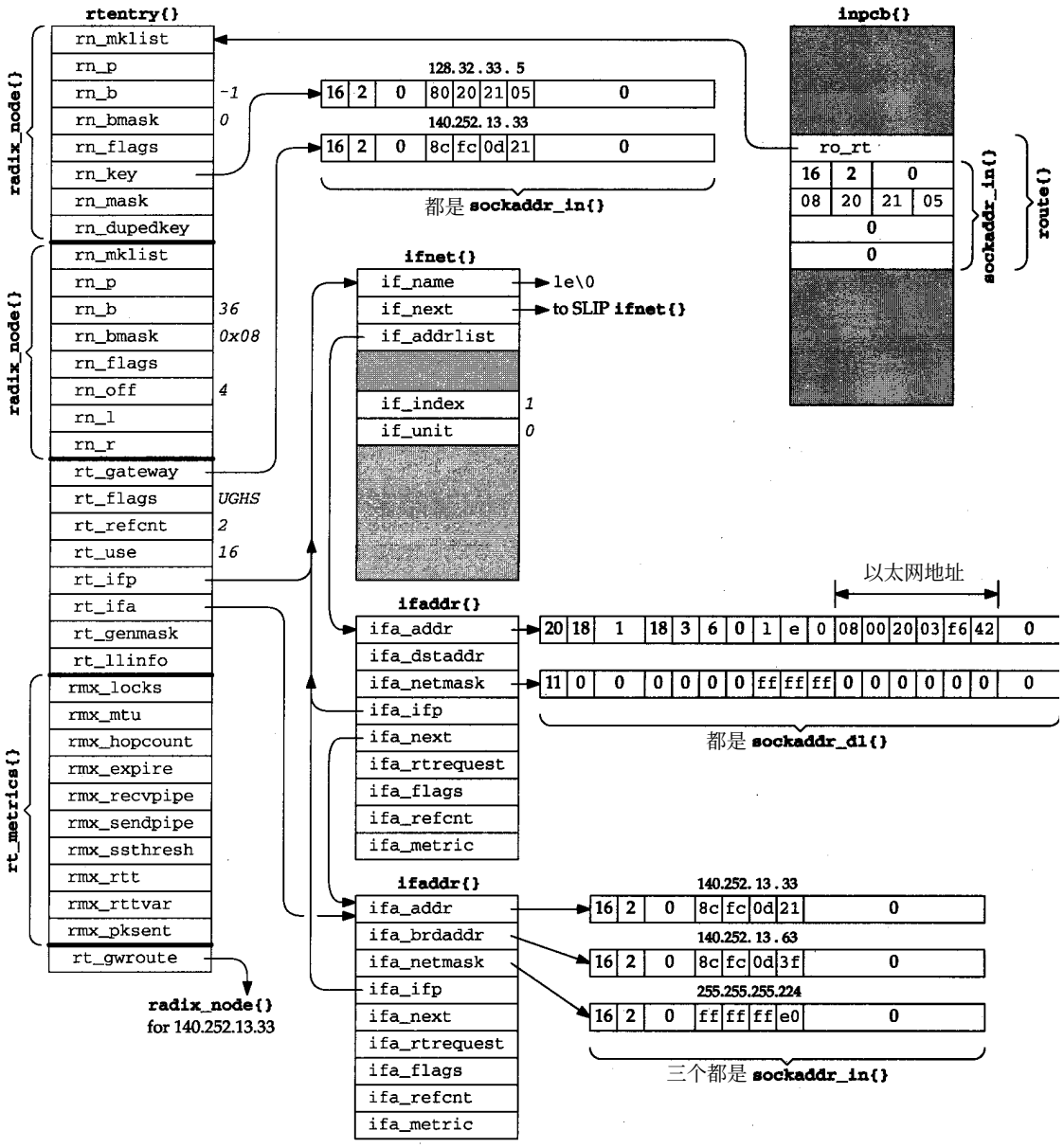


图18-28　选路结构小结

dom_rtoffset成员是以比特为单位的选路过程中被检测的第一个比特的偏移量 (从域的插口地址结构的起始处开始计算)。dom_maxrtkey给出了该结构的字节长度。在本章的前一部分的内容中，我们已经知道， sockaddr_in结构中的 IP地址是从比特 32开始的。dom_maxrtkey成员是协议的插口地址结构的字节长度： sockaddr_in的字节长度为16。

图18-29列出了路由表初始化过程所包含的步骤。

```
main()              /* kernel initialization */
{
    ...
    ifinit();
    domaininit();
    ...
}

domaininit()        /* Figure 7.15 */
{
    ...
    ADDDOMAIN(unix);
    ADDDOMAIN(route);
    ADDDOMAIN(inet);
    ADDDOMAIN(osi);
    ...
    for ( dp = all domains ) {
        (*dp->dom_init)();
        for ( pr = all protocols for this domain )
            (*pr->pr_init)();
    }
}

raw_init()          /* pr_init() function for SOCK_RAW/PF_ROUTE protocol */
{
                    初始化选路协议控制块的首部
}

route_init()        /* dom_init() function for PF_ROUTE domain */
{
    rn_init();
    rtable_init();
}

rn_init()
{
    for ( dp = all domains )
        if (dp->dom_maxrtkey > max_keylen)
            max_keylen = dp->dom_maxrtkey;
                分配并初始化 rn_zeros, rn_ones, masked_key;
    rn_inithead(&mask_rnhead);  /* allocate and init tree for masks */
}

rtable_init()
{
    for ( dp = all domains )
        (*dp->dom_rtattach)(&rt_tables[dp->dom_family]);
}

rn_inithead()       /* dom_attach() function for all protocol families */
{
                分配并初始化一个 radix_node_head 结构 ;
}
```

图18-29 初始化路由表时包含的步骤

在系统初始化时，内核的 main函数将调用一次domaininit函数。ADDDOMAIN宏用于

创建一个domain结构的链表，并调用每个域的dom_init函数(如果定义了该函数)。正如图18-27所示，route_init是唯一的一个dom_init函数，其代码如图18-30所示。

route.c

```
49 void
50 route_init()
51 {
52     rn_init();  /* initialize all zeros, all ones, mask table */
53     rtable_init((void **) rt_tables);
54 }
```

route.c

图18-30 rout_init 函数

在图18-32中的函数rn_init只被调用一次。

在图18-31中的函数rtable_init也只被调用一次。它接着调用所有域的dom_rtattach函数，这些函数为各自所属的域初始化一张路由表。

route.c

```
39 void
40 rtable_init(table)
41 void  **table;
42 {
43     struct domain *dom;
44     for (dom = domains; dom; dom = dom->dom_next)
45         if (dom->dom_rtattach)
46             dom->dom_rtattach(&table[dom->dom_family],
47                               dom->dom_rtoffset);
48 }
```

route.c

图18-31 rtable_init 函数：调用每一个域的dom_rtattach 函数

从图18-27中可知，rn_inithead是唯一的一个dom_rtattach函数，关于rn_inithead函数将在下一节中介绍。

18.8 初始化：rn_init和rn_inithead函数

图18-32中的函数rn_init只被route_init调用一次，用于初始化radix函数使用的一些全局变量。

1. 确定max_keylen

750-761 检查所有domain结构，并将全局变量max_keylen设置为最大的dom_maxrtkey值。在图18-27中最大值是32(对应于AF_ISO)，但是，在一个常用的不含OSI和XNS协议的系统中，max_key为16，即sockaddr_in结构的大小。

2. 分配并初始化rn_zeros、rn_ones和maskedKey

762-769 先分配了一个大小为max_keylen的三倍的缓存，并在全局变量rn_zeros中储存该缓存的指针。R_Malloc是一个调用内核的malloc函数的宏，它指定了M_RTABLE和M_DONTWAIT的类型。我们还会遇到Bcmp、Bcopy、Bzero和Free这些宏，它们对参数进行适当分类，并调用名称相似的内核函数。

该缓存被分解成三个部分，每一部分被初始化成如图18-33所示。

rn_zeros是一个全0比特的数组，rn_ones是一个全1比特的数组，maskedKey数组用于存放被掩码过的查找键的临时副本。

radix.c

```
750 void
751 rn_init()
752 {
753     char    *cp, *cplim;
754     struct domain *dom;
755     for (dom = domains; dom; dom = dom->dom_next)
756         if (dom->dom_maxrtkey > max_keylen)
757             max_keylen = dom->dom_maxrtkey;
758     if (max_keylen == 0) {
759         printf("rn_init: radix functions require max_keylen be set\n");
760         return;
761     }
762     R_Malloc(rn_zeros, char *, 3 * max_keylen);
763     if (rn_zeros == NULL)
764         panic("rn_init");
765     Bzero(rn_zeros, 3 * max_keylen);
766     rn_ones = cp = rn_zeros + max_keylen;
767     maskedKey = cplim = rn_ones + max_keylen;
768     while (cp < cplim)
769         *cp++ = -1;
770     if (rn_inithead((void **) &mask_rnhead, 0) == 0)
771         panic("rn_init 2");
772 }
```

radix.c

图18-32 rn_init 函数

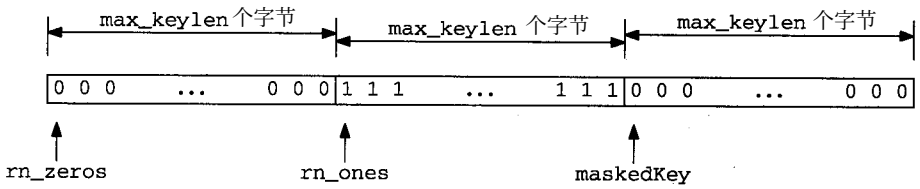


图18-33 rn_zeros 、rn_ones 和maskedKey 数组

3. 初始化掩码树

770-772 调用rn_inithead，初始化地址掩码路由树的首部；并使图 18-8中全局变量mask_rnhead指向该radix_node_head结构。

从图18-27可知，对于所有需要路由表的协议，rn_inithead也是它们的dom_attach函数。图 18-34给出的不是该函数的源代码，而是该函数为Internet协议创建的radix_node_head结构。

这三个radix_node结构组成了一棵树：中间的那个结构是树的顶点 (由rnh_treetop指向它)，第一个结构是树的最左边的叶子，最后一个结构是树的最右边的叶子。这三个结点的父指针(rn_p)都指向中间的那个结点。

rnh_nodes[1].rn_b的值32是待测试的比特位置。它来自于 Internet的domain结构中的dom_rtoffset成员(图18-27)。它的字节偏移量及字节掩码被预先计算出来，这样就不需要在处理过程中完成移位和掩码。其中，字节偏移量从插口地址结构起始处开始计算，它被存放在radix_node结构的rn_off成员中(在这个例子中它的值为 4)；字节掩码存放在rn_bmask成员中(在这个例子中为0x80)。无论何时往树中添加radix_node结构，都要计

算这些值，以便于在转发过程中加快比较的速度。其他的例子有：在图 18-4中检测比特33的两个结点的偏移量和字节掩码分别为 4和0x40；检测比特63的两个结点的偏移量和字节掩码分别为7和0x01。

两个叶子中的rn_b成员的值 - 33是由 - 1减去该叶子的索引而得到的。

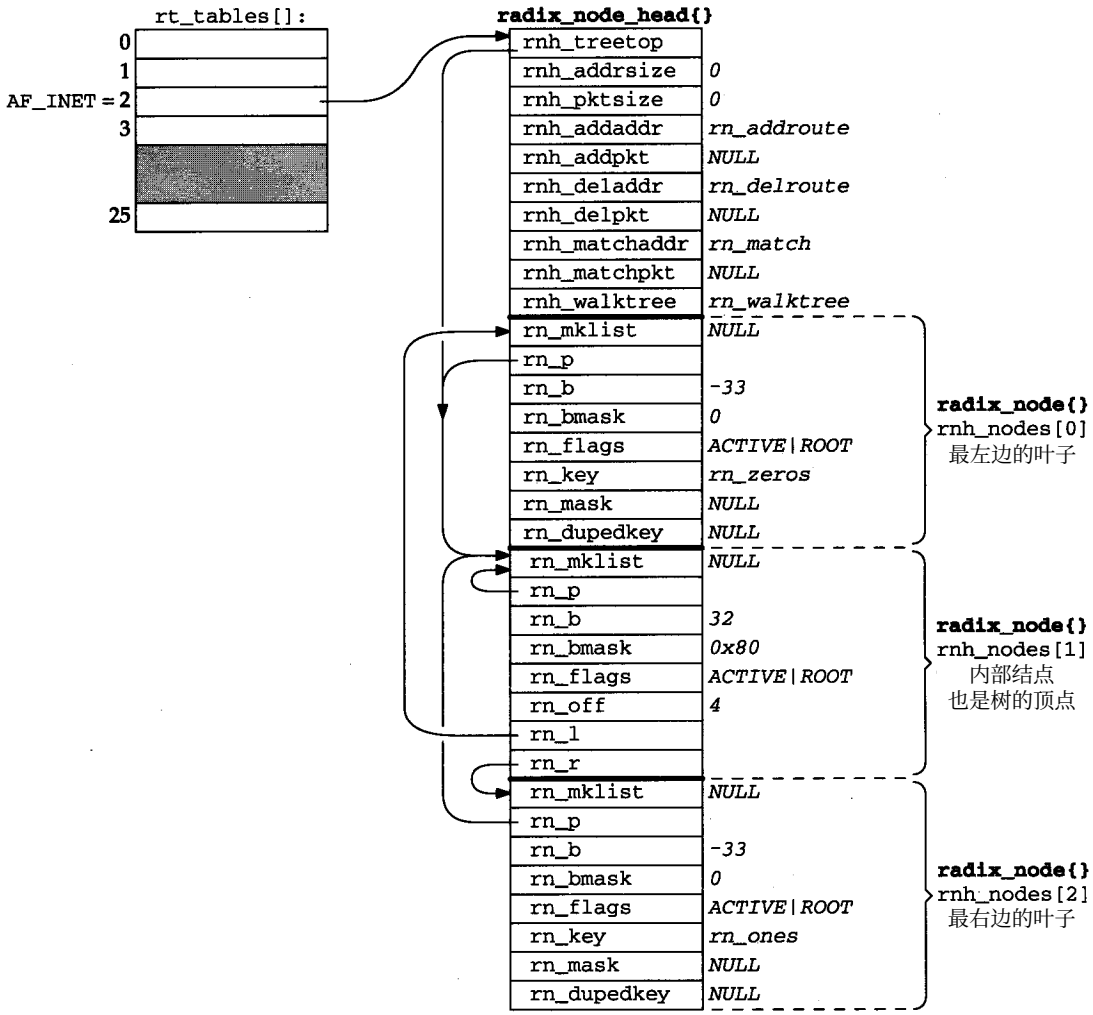


图18-34 rn_inithead 为Internet协议创建的 radix_node_head 结构

最左边结点的键是全0(rn_zeros)，最右边结点的键是全1(rn_ones)。

这三个结点都设置了 RNF_ROOT标志(我们省略了前缀 RNF_)。这说明它们都是构成树的原始结点之一。它们也是唯一具有该标志的结点。

有一个细节我们没有提到，就是网络文件系统 (NFS)也使用路由表函数。对于本地主机的每一个装配点 (mount point)都分配一个radix_node_head结构，并且具有一个指向这些结构的指针数组 (利用协议族检索)，与rt_tables数组相似。每次输出装配点时，针对该装配点，把能装配该文件系统的主机的协议地址添加到适当的树中去。

18.9 重复键和掩码列表

在介绍查找路由表项的源代码之前，必须先理解 radix_node结构中的两个字段：一个是rn_dupedkey，它构成了附加的含重复键的 radix_node结构链表；另一个是rn_mklist，它是含网络掩码的radix_mask结构链表的开始。

先看一下图18-4中树的最左边标有“ end ”和“ default ”的两个框。这些就是重复键。最左边设有RNF_ROOT标志的结点(在图18-34中的rnh_nodes[0])有一个为全0比特的键，但是它和默认路由的键相同。如果创建一个255.255.255.255的路由表项(但该地址是受限的广播地址，不会在路由树中出现)，则我们会在树的最右端结点 (该结点有一个值为全1比特的键)遇到同样的问题。总的来说，如果每次都有不同的掩码，那么Net/3中的radix结点函数就允许重复任何键。

图18-35给出了两个具有全 0比特重复键的结点。在这个图中，我们去掉了 rn_flags中的RNF_前缀，并且省略了非空父指针、左指针和右指针，因为它们与要讨论的内容无关。

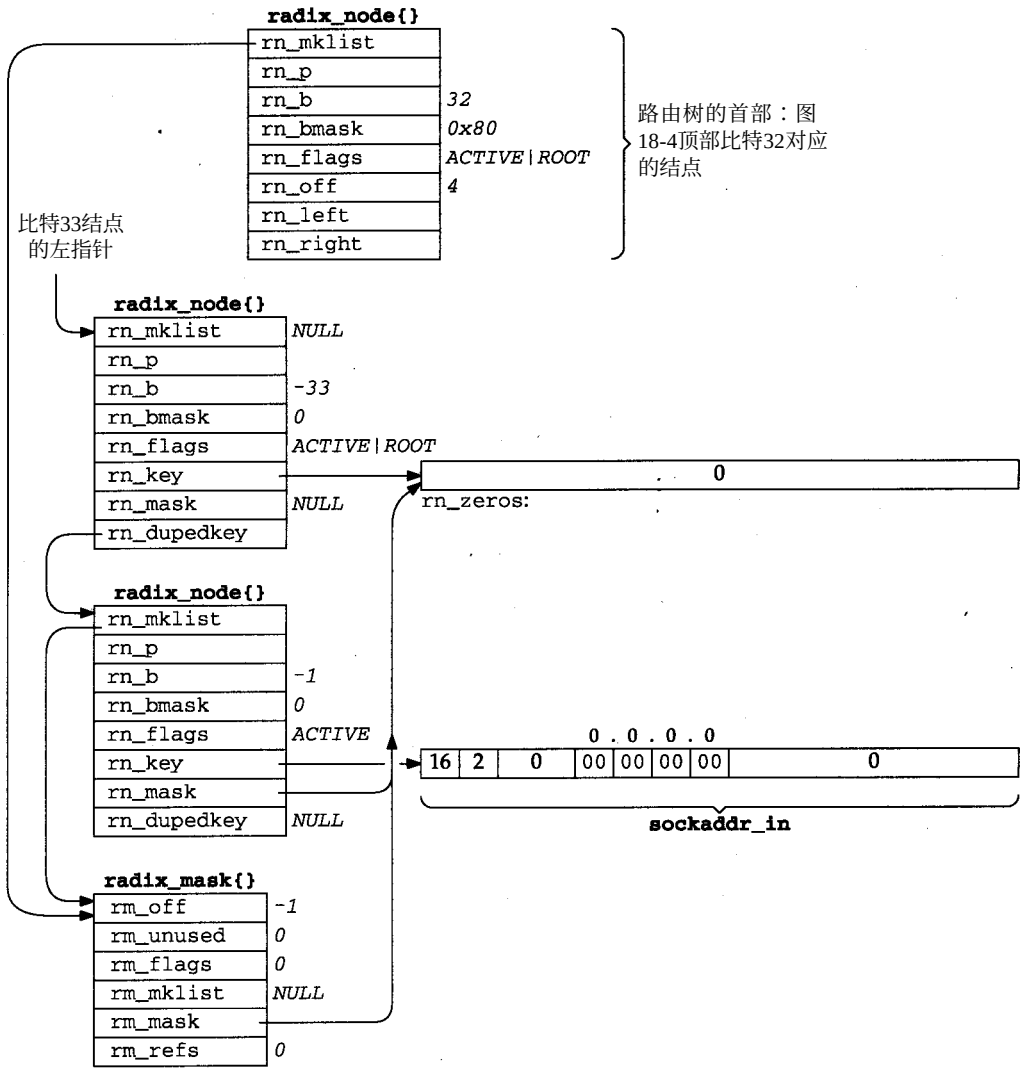


图18-35 值为全0的键的重复结点

图中最上面的结点即为路由树的顶点——图18-4中顶部比特32对应的结点。接下来的两个结点是叶子(它们的rn_b为负值)，其中第一个叶子的rn_dupedkey成员指向第二个结点。第一个叶子是图18-34中的rnh_nodes[0]结构，该结构是树的左边标有“ end ”的结点——它设有RNF_ROOT标志。它的键被rn_inithead设为rn_zeros。

第二个叶子是默认路由的表项。它的 rn_key指向值为0.0.0.0的sockaddr_in结构，并具有一个全0的掩码。由于掩码表中相同的掩码是共享的，因此，该叶子的 rn_mask也指向rn_zeros。

通常，键是不共享的，更不会与掩码共享。由于两个标有“ end ”的结点的rn_key指针(具有RNF_ROOT标志)是由rn_inithead(图18-34)创建的，因此这两个指针例外。左边标有end的结点的键指向rn_zeros，右边标有“ end ”的结点的键指向rn_ones。

最后一个是radix_mask结构，树的顶结点和默认路由对应的叶子都指向这个结构。这个列表是树的顶结点的掩码列表，在查找网络掩码时，回溯算法将使用它。 radix_mask结构列表和内部结点一起确定了运用于从该结点开始的子树的掩码。在重复键的例子中，掩码列表和叶子出现在一起，跟着的这个例子也是这样的。

现在我们给出一个特意添加到选路树中的重复键和所得到的掩码列表。在图 18-4中有一个主机路由 127.0.0.1和一个网络路由 127.0.0.0。图中采用了 A类网络路由的默认掩码，即0xff000000。如果我们把跟在A类网络号之后的24 bit分解成一个16 bit子网号和一个8 bit主机号，就可以为子网127.0.0添加一个掩码为0xffffff00的路由：

```
bsdi $ route add 127.0.0.0 -netmask 0xffffff00 140 252 13 33
```

虽然在这种情况下使用网络127没什么实际意义，但是我们感兴趣的是所得到的路由表结构。虽然重复键在Internet协议中并不常见(除了前面例子中的默认路由之外)，但是仍需要利用重复键来为所有网络的0号子网提供路由。

在网络号127的这三个路由表项中存在一个隐含的优先规则。如果查找键是 127.0.0.1，则它和这三个路由表项都匹配，但是只选择主机路由，因为它是最匹配的：其掩码(0xffffffff)含有最多的1。如果查找键是127.0.0.2，它与两个网络路由匹配，但是掩码为0xffffff00的子网0的路由比掩码为0xff0000的路由更匹配。如果查找键为 127.0.2.3，那么只与掩码为0xff000000的路由表项匹配。

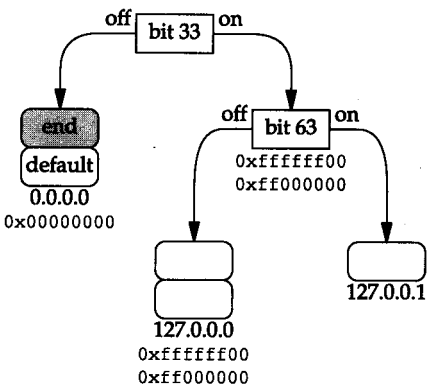


图18-36 反映重复键127.0.0.0的路由树

图18-36给出了添加路由之后得到的树结构，从图18-4中对应比特33的内部结点处开始。由于这个重复键有两个叶子，我们用两个框来表示键值为 127.0.0.0的路由表项。

图18-37给出了所得到的radix_nod和radix_mask结构。

首先看一下每一个radix_node的radix_mask结构的链表。最上端结点 (比特63)的掩码列表由0xffffff00及其后的0xff000000组成。在列表中首先遇到的是更匹配的掩码，这样它就能够更早地被测试到。第二个 radix_node(rn_b值为 - 57的那个)的掩码列表与第一个相同。但是第三个radix_node的掩码列表仅由值为0xff000000的掩码构成。

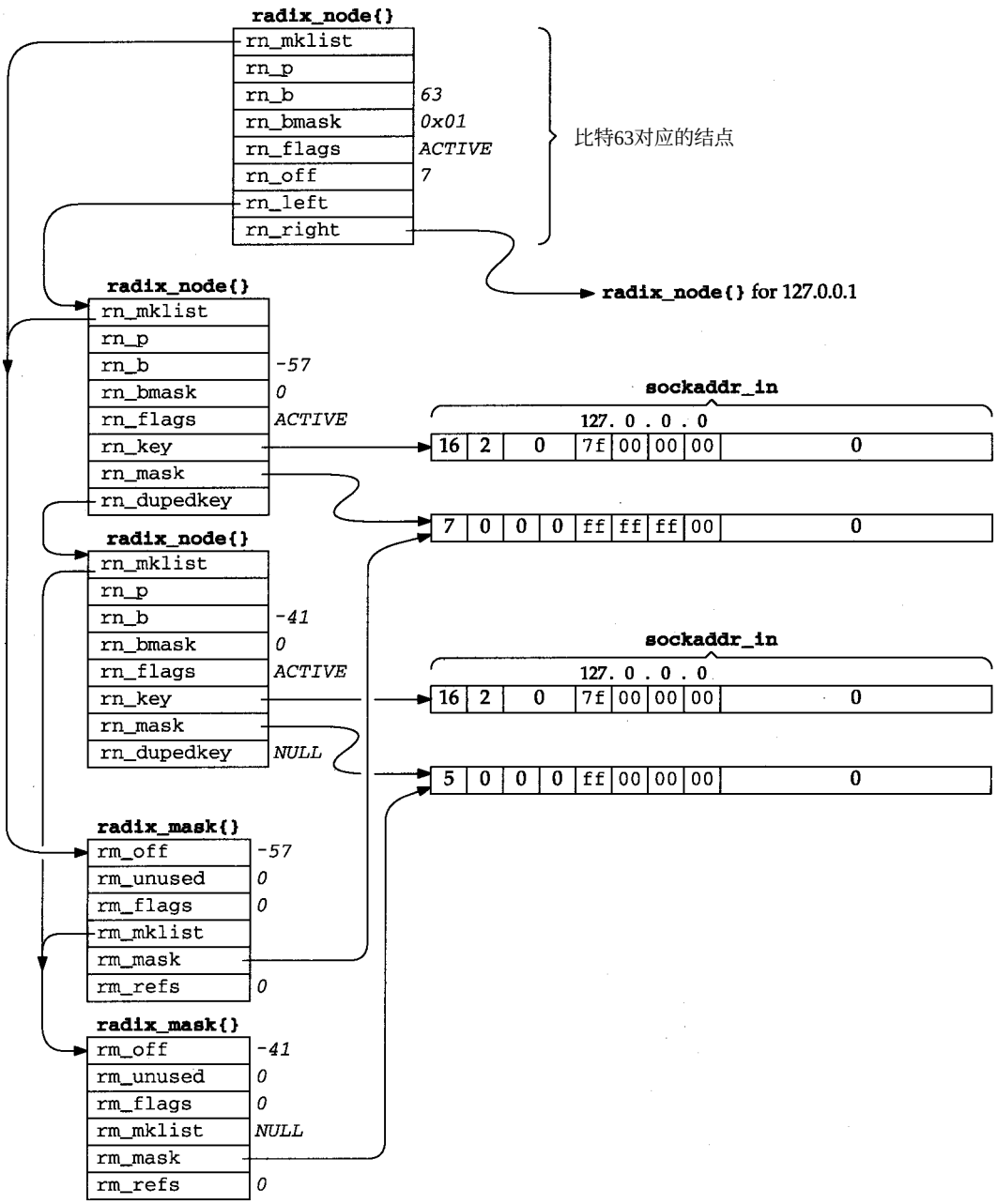


图18-37 网络127.0.0.0的重复键的路由表结构举例

应注意的是，具有相同值的掩码之间可以共享，但是具有相同值的键之间不能共享。这是因为掩码被保存在它们自己的路由树中，可以显式地被共享，而且值相同的掩码经常出现(例如，每个C类网络路由都有相同的掩码`0xffffff00`)，但是值相同的键却不常见。

18.10 rn_match函数

现在我们介绍`rn_match`函数，在Internet协议中，它被称为`rnh_matchaddr`函数。在

后面学习中，我们可以看到它将被rtallocl函数调用(而rtalloc1函数将被rtalloc函数调用)。具体算法如下：

1) 从树的顶端开始搜索，直到到达与查找键的比特相应的叶子。检测该叶子，看能否得到一个精确的匹配(图18-38)。

2) 检测该叶结点，看是否能得到匹配的网络地址。

3) 回溯(图18-43)。

图18-38给出了rn_match的第一部分。

radix.c

```
135 struct radix_node *
136 rn_match(v_arg, head)
137 void    *v_arg;
138 struct radix_node_head *head;
139 {
140     caddr_t v = v_arg;
141     struct radix_node *t = head->rnh_treetop, *x;
142     caddr_t cp = v, cp2, cp3;
143     caddr_t cplim, mstart;
144     struct radix_node *saved_t, *top = t;
145     int     off = t->rn_off, vlen = *(u_char *) cp, matched_off;
146     /*
147      * Open code rn_search(v, top) to avoid overhead of extra
148      * subroutine call.
149      */
150     for (; t->rn_b >= 0;) {
151         if (t->rn_bmask & cp[t->rn_off])
152             t = t->rn_r;        /* right if bit on */
153         else
154             t = t->rn_l;        /* left if bit off */
155     }
156     /*
157      * See if we match exactly as a host destination
158      */
159     cp += off;
160     cp2 = t->rn_key + off;
161     cplim = v + vlen;
162     for (; cp < cplim; cp++, cp2++)
163         if (*cp != *cp2)
164             goto on1;
165     /*
166      * This extra grot is in case we are explicitly asked
167      * to look up the default.  Ugh!
168      */
169     if ((t->rn_flags & RNF_ROOT) && t->rn_dupedkey)
170         t = t->rn_dupedkey;
171     return t;
172 on1:
```

radix.c

图18-38 rn_match 函数：沿着树向下搜索，查找严格匹配的主机地址

135-145 第一个参数v_arg是一个插口地址结构的指针，第二个参数head是该协议的指向radix_node_head结构的指针。所有协议都可调用这个函数 (图18-17)，但调用时使用不同的head参数。

在变量声明中，off是树的顶结点的rn_off成员(对Internet地址，其值为4，见图18-34)，

vlen是查找键插口地址结构中的长度字段(对Internet地址，其值为16)。

1. 沿着树向下搜索到相应的叶子

146-155 这个循环从树的顶结点开始，然后沿树的左右分支搜索，直到遇到一个叶子为止(rn_b小于0)。每次测试相应比特时，都利用了事先计算好的rn_bmask中的字节掩码和事先计算好的rn_off中的偏移量。对于Internet地址而言，rn_off为4、5、6或7。

2. 检测是否精确匹配

156-164 当遇到叶子时，首先检测能否精确匹配。比较插口地址结构中从协议族的rn_off值开始的所有字节。图18-39给出了Internet插口地址结构的比较情况。

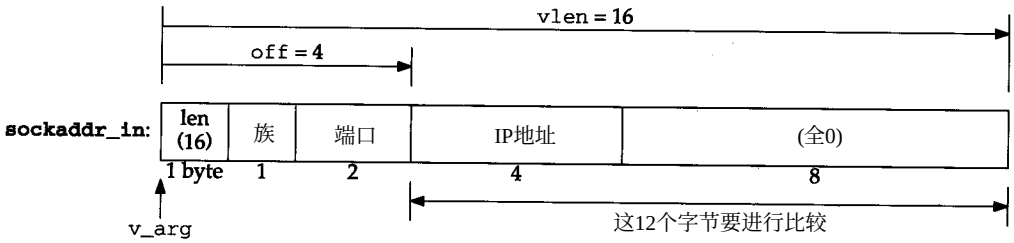


图18-39 比较sockaddr_in 结构时的各种变量

如果发现匹配不成功，就立刻跳到on1。

通常，sockaddr_in的最后8个字节为0，但是地址解析协议代理(proxy ARP)(21.12节)会设置其中的一个为非零。这就允许一个给定的IP地址有两个路由表项：一个对应于正常IP地址(最后8个字节为0)，另一个对应于相同IP地址的地址解析协议代理(最后8个字节中有一个为非零)。

图18-39中的长度字节在函数的一开始时就赋值给了vlen，并且我们还会看到rtallocl将利用family成员来选择路由表进行搜索。选路函数未使用port成员。

3. 显示地检测默认地址

165-172 图18-35给出了存储在键为0的重复叶子中的默认路由。第一个重复的叶子设有RNF_ROOT标志。因此，如果在匹配的结点中设有RNF_ROOT标志，并且该叶子含有重复键，那么就返回指针rn_dupedkey的值(即图18-35中含默认路由的结点的指针)。如果路由树中没有默认路由，则查找过程匹配左边标有“end”的叶子(键为全0比特)；或者如果查找时遇到右边标有“end”的叶子(键值为全1比特)，那么返回指针t，它指向一个设有RNF_ROOT标志的结点。我们将看到rtallocl会显式地检查匹配结点是否设有这个标志，并判断匹配是否失败。

程序执行到此时，rn_match函数已经到达了某个叶子上，但是它并不是查找键的精确匹配。函数的下一部分将检测该叶子是否为匹配的网络地址，如图18-40所示。

173-174 cp指向该查找键中那个不相等的字节。matched_off被赋值为该字节在插口地址结构中的位置偏移量。

175-183 do while循环反复与所有重复叶子中的每一个具有网络掩码的叶子进行比较。下面我们通过一个例子来看这段代码。假定我们要在图18-4所示的路由表中查找IP地址140.252.13.60。查找会在标有140.252.13.32(比特62和63都为0)的结点处终止，该结点包含一个网络掩码。图18-41给出了图18-40中的for循环开始执行时的结构。

radix.c

```
173     matched_off = cp - v;
174     saved_t = t;
175     do {
176         if (t->rn_mask) {
177             /*
178              * Even if we don't match exactly as a host;
179              * we may match if the leaf we wound up at is
180              * a route to a net.
181              */
182             cp3 = matched_off + t->rn_mask;
183             cp2 = matched_off + t->rn_key;
184             for (; cp < cplim; cp++)
185                 if ((*cp2++ ^ *cp) & *cp3++)
186                     break;
187             if (cp == cplim)
188                 return t;
189             cp = matched_off + v;
190         }
191     } while (t = t->rn_dupedkey);
192     t = saved_t;
```

radix.c

图18-40 rn_match 函数：检测是否为匹配的网络地址

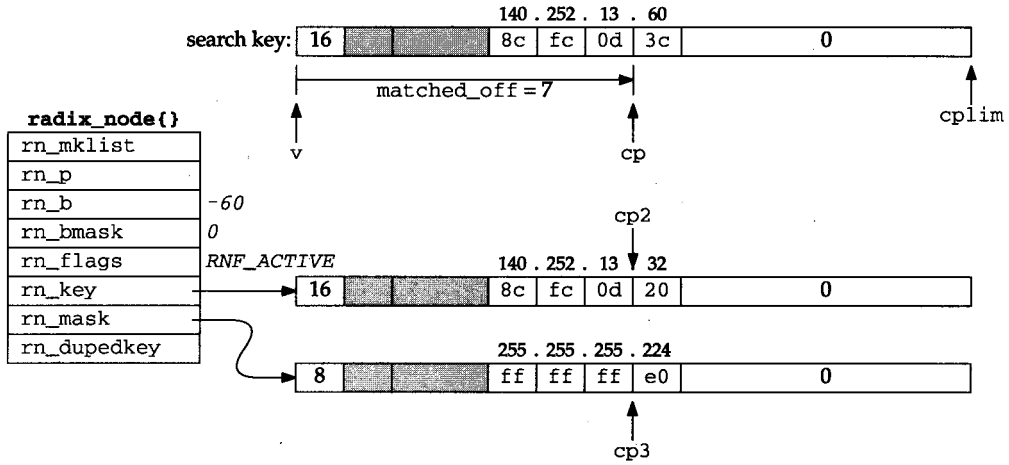


图18-41 比较网络掩码的例子

虽然查找键和路由表键都是sockaddr_in结构，但是掩码的长度并不相同。该掩码长度是非零字节的最小数目。从该点之后直到max_keylen之间的所有字节都为0。

184-190 逐个字节地对查找关键字和路由表键进行异或运算，并将结果同网络掩码进行逻辑与运算。如果所得到的字节出现非零值，就会由于不匹配而终止循环 (习题18.1)。如果循环正常终止，那么与网络掩码进行逻辑与运算后的查找键就和路由表项相匹配。程序将返回指向该路由表项的指针。

查看IP地址的第四个字节，我们可以从图 18-42中看出本例子是如何匹配成功的，以及 IP地址140.252.13.188是如何匹配失败的。采用这两个地址，是因为它们中的比特 57、62和63都为0，查找都在图18-41给出的结点上终止。

第一个例子(140.252.13.60)匹配成功是因为逻辑与运算的结果为 0(并且地址、键和掩码中所有剩余的字节全都为0)。另一个例子匹配不成功是因为逻辑与运算的结果为非零。

	查找键 = 140.252.13.60	查找键 = 140.252.13.188
查找键字节(*cp): 路由表键字节(*cp2):	0011 1100 = 3c 0010 0000 = 20	1011 1100 = bc 0010 0000 = 20
异或: 网络掩码字节(*cp3):	0001 1100 1110 0000 = e0	1001 1100 1110 0000 = e0
逻辑与:	0000 0000	1000 0000

图18-42 用网络掩码进行关键字匹配的例子

191 如果路由表项含有重复键，那么对每一个键都要执行一次该循环体。

rn_match的最后一部分，如图18-43所示，沿路由树向上回溯，以查找匹配的网络地址或默认地址。

radix.c

```
193     /* start searching up the tree */
194     do {
195         struct radix_mask *m;
196         t = t->rn_p;
197         if (m = t->rn_mklist) {
198             /*
199              * After doing measurements here, it may
200              * turn out to be faster to open code
201              * rn_search_m here instead of always
202              * copying and masking.
203              */
204             off = min(t->rn_off, matched_off);
205             mstart = maskedKey + off;
206             do {
207                 cp2 = mstart;
208                 cp3 = m->rm_mask + off;
209                 for (cp = v + off; cp < cplim;)
210                     *cp2++ = *cp++ & *cp3++;
211                 x = rn_search(maskedKey, t);
212                 while (x && x->rn_mask != m->rm_mask)
213                     x = x->rn_dupedkey;
214                 if (x &&
215                     (Bcmp(mstart, x->rn_key + off,
216                         vlen - off) == 0))
217                     return x;
218             } while (m = m->rm_mklist);
219         }
220     } while (t != top);
221     return 0;
222 };
```

radix.c

图18-43 rn_match 函数：沿树向上回溯

193-195 do while 循环沿着路由树一直向上，检测每一层的结点，直至检测到树的顶端为止。

196 指向父结点的指针的值被赋给了指针 t，即向上移动了一层。可见，在每一个结点中包含一个父指针能够简化回溯操作。

197-210 对于回溯到的每一层，只要内部结点的掩码列表非空，就对该层进行检测。rn_mklist是指向radix_node结构的链表的指针，链表中的每一个 radix_node结构都包含一个掩码，这些掩码将应用于从该结点开始的子树。程序中的内部 do while循环将遍历每一个radix_mask结构。

利用前面的例子，140.252.13.188，图18-44给出了在最内层的for循环开始时的各种数据结构。这个循环对每个掩码中的字节和对应的查找键的字节进行逻辑与操作，并将结果保存在全局变量maskedKey中。该掩码值为0xfffffe0，搜索会从图18-4中的叶结点140.252.13.32处回溯两层，到达测试比特62的结点。

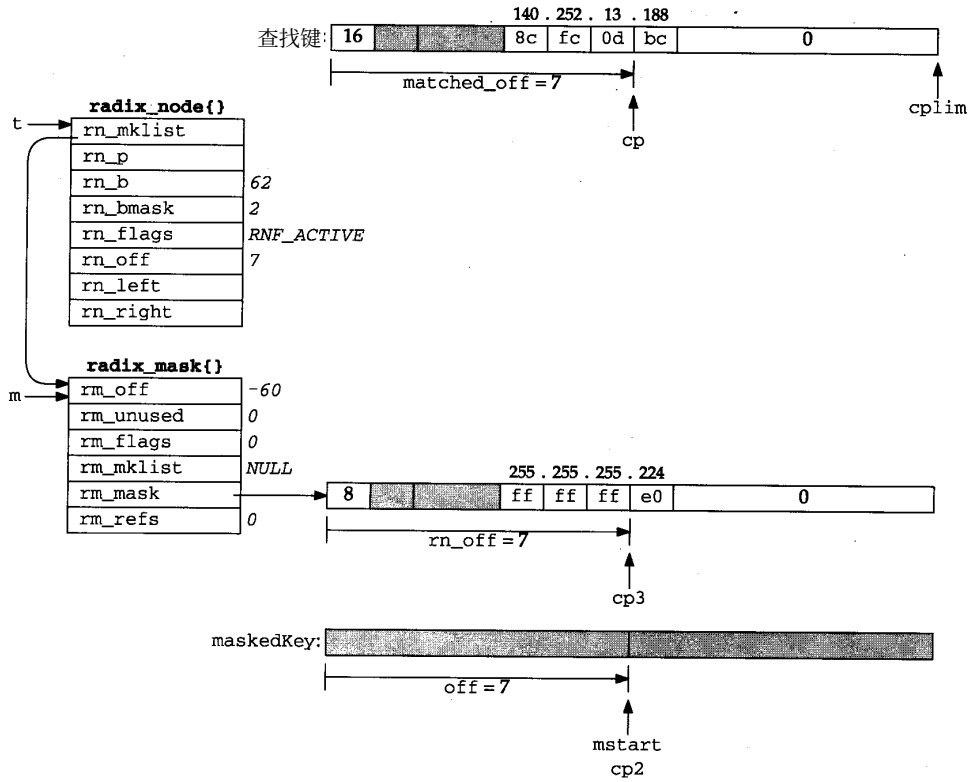


图18-44 利用掩码过的查找键进行再次搜索的准备

for循环完成后，掩码过程也就完成了，再调用rn_search(如图18-48所示)，其调用参数以maskedKey为查找键，以指针t为查找子树的顶点。图18-45给出了我们所举例子中的maskedKey的值。

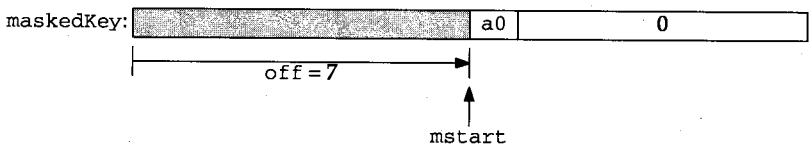


图18-45 调用rn_search 时的maskedKey

字节0xa0是0xbc(188，查找键)和0xe0(掩码)逻辑与运算的结果。

211 rn_search从起点开始沿着树往下搜索，根据查找键来确定沿向左或向右的分支进行搜索，直到到达某个叶子。在这个例子中，查找键有9个字节，如图18-45所示，所到达的是图18-4中标有140.252.13.32的那个叶子，这是因为在字节0xa0中比特62和63都为0。图18-46给出了调用Bcmp检验是否匹配时的数据结构。

由于这两个9字节的字符串不相同，所以这次比较失败。

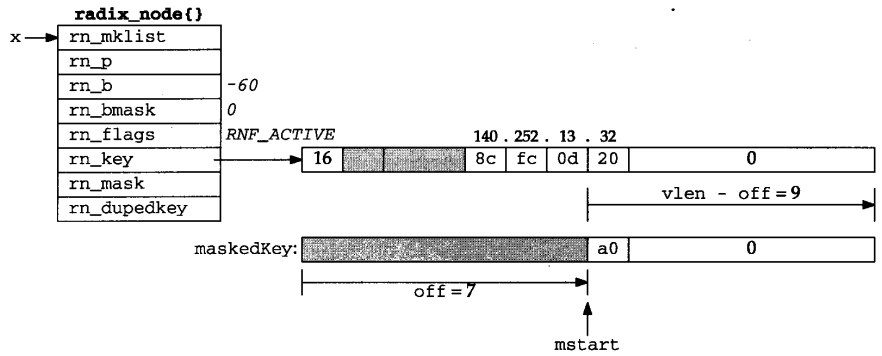


图18-46 maskedKey 和新叶结点之间的比较

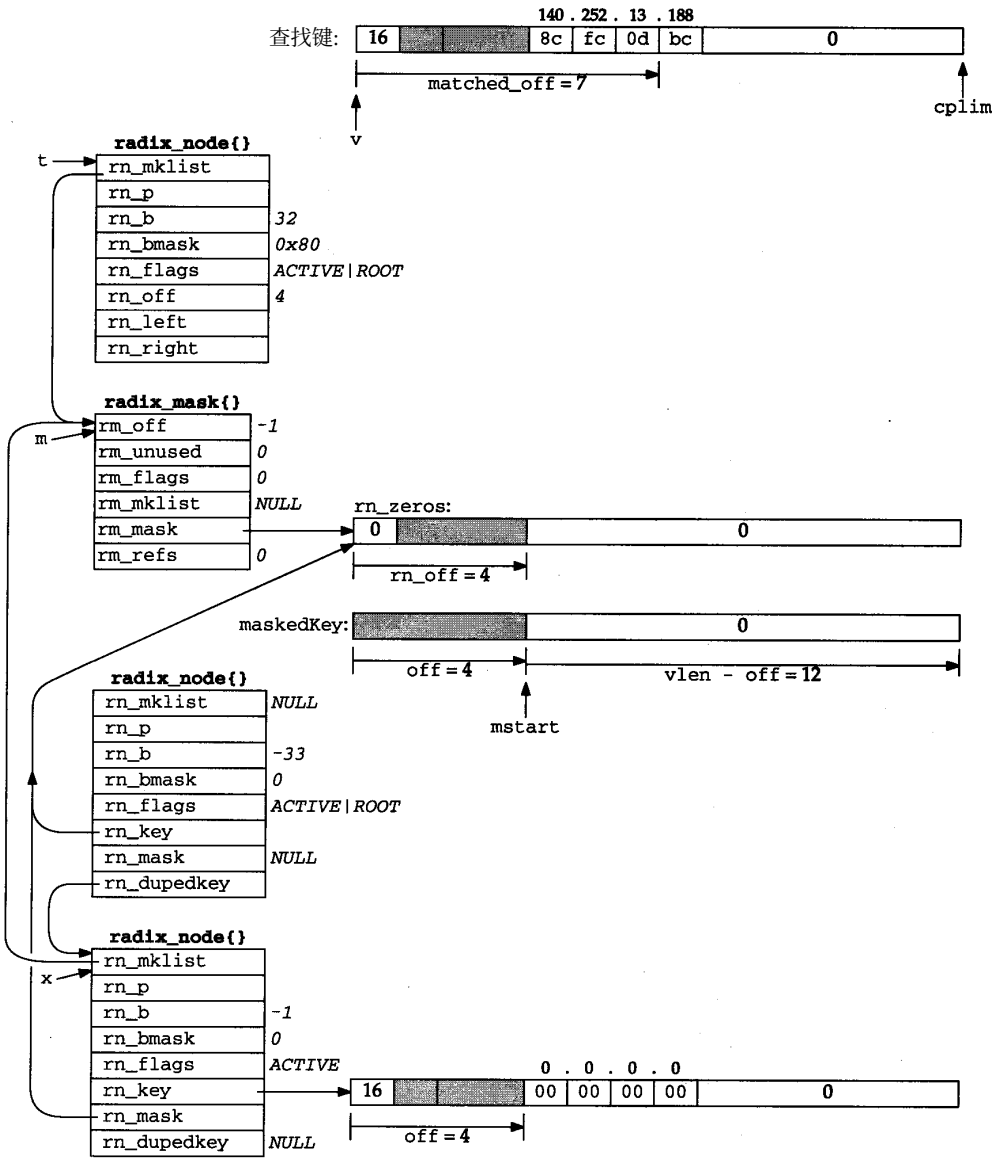


图18-47 回溯到路由树的顶端和查找默认叶子的 rn_search

212-221 该while循环处理各重复键，且处理每个重复键时的掩码不同。唯一被比较的重复键是那个rn_mask指针与m->rm_mask相等的键。下面以图18-36和图18-37为例进行说明。如果查找从比特63的结点处开始，第一次内部 do while循环中，m指向radix_mask结构0xffffff00。当rn_search返回指向第一个重复叶子 127.0.0.0的指针时，该叶子的rm_mask等于m->rm_mask，因此，就调用Bcmp。如果比较失败，m的值就被设置成指向列表中的下一个radix_mask结构(具有掩码0xff000000)的指针，并且对新掩码再次执行 do while循环体。rn_search再一次返回指向第一个重复叶子 127.0.0.0的指针，但是它的rn_mask并不等于m->rm_mask。While继续进行到下一个重复叶子，它的 rn_mask与m->rm_mask恰好相等。

现在回到查找键为140.252.13.188的例子中，由于从检测比特62的结点处开始的搜索失败，因此，沿着树向上继续回溯，直到到达树的顶点，该顶点就是沿树向上的下一个 rn_mklist为非空的结点。

图18-47给出了到达树的顶结点时的数据结构。此时，计算 maskedKey(为全0)，并且rn_search从这个结点(树的顶结点)处开始，继续沿着树的左分支向下两层到达图 18-4中标有“ default ”的叶子。

当rn_search返回时，x指向rn_b值为-33的radix_node，这是从树的顶端开始沿两个左分支向下之后遇到的第一个叶子。但是 x->rn_mask(为空)与m->rm_mask不等，因此，将x->rn_dupedkey赋给x。用于测试的while循环再次执行，但是，此时 x-rn_mask等于m->rm_mask，因此该while循环终止。Bcmp对从mstart开始的12个值为0的字节和从x->rn_key加4开始的12个值为0的字节进行比较，结果相等，函数返回指针 x，该指针指向默认路由的路由项。

18.11 rn_search函数

在前面一节中，我们已经知道 rn_match调用了rn_search来搜索路由表的子树。如图18-48所示。

radix.c

```
79 struct radix_node *
80 rn_search(v_arg, head)
81 void    *v_arg;
82 struct radix_node *head;
83 {
84     struct radix_node *x;
85     caddr_t v;

86     for (x = head, v = v_arg; x->rn_b >= 0;) {
87         if (x->rn_bmask & v[x->rn_off])
88             x = x->rn_r;          /* right if bit on */
89         else
90             x = x->rn_l;          /* left if bit off */
91     }
92     return (x);
93 };
```

radix.c

图18-48 rn_search 函数

这个循环和图18-38中的相似。它在每一个结点上比较查找键中的一位比特，如果该比特

为0，就通向左边的分支，如果该比特为1，就通向右边的分支。在遇到一个叶子时终止搜索，并返回指向该叶子的指针。

18.12 小结

每一个路由表项都由一个键来标识：在IP协议中就是目的IP地址，该IP地址可以是一个主机地址或者是一个具有相应网络掩码的网络地址。一旦键的搜索确定了路由表项，在该表项中的其他信息就会指定一个路由器的IP地址，到目的地址的数据报就会发往该指定地址，还会指明要用到的接口的指针、度量等等。

由Internet协议维护的信息是`route`结构，该`route`结构只有两个成员构成：指向路由表项的指针和目的地址。在UDP、TCP和原始IP使用的每个Internet协议控制块中，我们都会遇到由Internet协议维护的`route`结构。

Patricia树数据结构非常适合于路由表。由于路由表的查找要比添加或者删除路由频繁得多，因此从性能的角度来看，在路由表中使用Patricia树就更加有意义。Patricia树虽然不利于添加和删除这些附加工作，但是加快了查找的速度。[Sklower 1991]给出的radix树方法和Net/1散列表的比较结果表明，用radix树方法构造测试树用比Net/1散列表法快一倍，搜索速度快三倍。

习题

18.1 我们说过，在图18-3中，查找键与路由表项匹配的一般条件是，它和路由表掩码的逻辑与运算的结果等于路由表键。但是在图18-40中采用了不同的测试方法。请建立一个逻辑真值表以证明这两种方法等价。

18.2 假设某个Net/3系统中的路由表需要20 000个表项(IP地址)。在不考虑掩码的情况下，请估算大约需要多大的存储器？

18.3 `radix_node`结构对路由表键的长度限制是多少？

第19章　选路请求和选路消息

19.1　引言

内核的各种协议并不直接使用前一章提供的函数来访问选路树，而是调用本章提供的几个函数：rtalloc和rtalloc1是完成路由表查询的两个函数；rtrequest函数用于添加和删除路由表项；另外大多数接口在接口连接或断开时都会调用函数 rtinit。

选路消息在两个方向上传递信息。进程(如route命令)或守护进程(routed或gated)把选路消息写入选路插口，以使内核添加路由、删除路由或修改现有的路由。当有事件发生时，如接口断开、收到重定向等，内核也会发送选路消息。进程通过选路插口来读取它们感兴趣的内容。在本章中，我们将讨论这些选路消息的格式及其含义，关于选路插口的讨论将在下一章进行。

内核还提供了另一种访问路由表的接口，即系统的 sysctl调用，我们将在本章的结尾部分阐述。该系统调用允许进程读取整个路由表或所有已配置的接口及接口地址。

19.2　rtalloc和rtalloc1函数

通常，路由表的查找是通过调用 rtalloc和rtalloc1函数来实现的。图 19-1给出了rtalloc。

route.c

```
58 void
59 rtalloc(ro)
60 struct route *ro;
61 {
62     if (ro->ro_rt && ro->ro_rt->rt_ifp && (ro->ro_rt->rt_flags & RTF_UP))
63         return;                 /* XXX */
64     ro->ro_rt = rtalloc1(&ro->ro_dst, 1);
65 }
```

route.c

图19-1　rtalloc 函数

58-65　参数ro是一个指针，它指向TCP或UDP所使用的Internet PCB(第22章)中的route结构。如果ro已经指向了某个rtentry结构(即ro_rt非空)，而该结构指向一个接口结构且路由有效，则函数立即返回。否则， rtalloc1将被调用，调用的第二个参数为 1。我们很快会看到该参数的用途。

如图19-2所示，rtalloc1调用了rnh_matchaddr函数，对于Internet地址来说，该函数就是rn_match数(图18-17)。

66-76　第一个参数是一个指针，它指向一个含有待查找地址的插口地址结构。 sa_family成员用于选择所查找的路由表。

1. 调用rn_match

77-78　如果符合下列三个条件，则查找成功。

1) 存在该协议族的路由表；

2) rn_match返回一个非空指针；并且

3) 匹配的radix_node结构没有设置RNF_ROOT标志。

注意，树中标有end的两个叶子都设有RNF_ROOT标志。

route.c

```
66 struct rtentry *
67 rtalloc1(dst, report)
68 struct sockaddr *dst;
69 int     report;
70 {
71     struct radix_node_head *rnh = rt_tables[dst->sa_family];
72     struct rtentry *rt;
73     struct radix_node *rn;
74     struct rtentry *newrt = 0;
75     struct rt_addrinfo info;
76     int     s = splnet(), err = 0, msgtype = RTM_MISS;
77     if (rnh && (rn = rnh->rnh_matchaddr((caddr_t) dst, rnh)) &&
78         ((rn->rn_flags & RNF_ROOT) == 0)) {
79         newrt = rt = (struct rtentry *) rn;
80         if (report && (rt->rt_flags & RTF_CLONING)) {
81             err = rtrequest(RTM_RESOLVE, dst, SA(0),
82                             SA(0), 0, &newrt);
83             if (err) {
84                 newrt = rt;
85                 rt->rt_refcnt++;
86                 goto miss;
87             }
88             if ((rt = newrt) && (rt->rt_flags & RTF_XRESOLVE)) {
89                 msgtype = RTM_RESOLVE;
90                 goto miss;
91             }
92         } else
93             rt->rt_refcnt++;
94     } else {
95         rtstat.rts_unreach++;
96       miss:if (report) {
97             bzero((caddr_t) & info, sizeof(info));
98             info.rti_info[RTAX_DST] = dst;
99             rt_missmsg(msgtype, &info, 0, err);
100        }
101    }
102    splx(s);
103    return (newrt);
104 }
```

route.c

图19-2　rtalloc1 函数

2. 查找失败

94-101　在这三个条件中只要有一个条件没有得到满足，查找就会失败，并且统计值rts_unreach也要递增。这时，如果调用rtallco1的第二个参数(report)为1，就会产生一个选路消息。任何感兴趣的进程都可以通过选路插口读取该消息。选路消息的类型为RTM_MISS，并且函数返回一个空指针。

79　如果三个条件都满足，则查找成功。指向匹配的 radix_node结构的指针保存在rt和newrt中。注意，在rtentry结构的定义中(图18-24)，两个radix_node结构在开头的位置处，如图18-8所示，其中第一个代表一个叶结点。因此， rn_match返回的radix_node结构的指针事实上是一个指向rtentry结构的指针，该rtentry结构是一个匹配的叶结点。

3. 创建克隆表项

80-82 如果调用的第二个参数非零，而且匹配的路由表项设有RTF_CLONING标志，则调用rtrequest函数发送RTM_RESOLVE命令来创建一个新的rtentry结构，该结构是查询结果的克隆。ARP将针对多播地址利用这一机制。

4. 克隆失败

83-87 如果rtrequest返回一个差错，newrt就被重新设置成rn_match所返回的表项，并增加它的引用计数。然后程序跳转到miss处，产生一条RTM_MISS消息。

5. 检查是否需要外部转换

88-91 如果rtrequest成功，并且新克隆的表项设有RTF_XRESOLVE标志，则程序跳至miss处，但这次产生的是RTM_RESOLVE消息。该消息的目的是为了把路由创建的时间通知给用户进程，在IP地址到X.121地址的转换过程中会用到它。

6. 为正常的成功查找递增引用计数

92-93 当查找成功但没有设置RTF_CLONING标志时，该语句将递增路由表项的引用计数。这是本函数正常情况下的处理流程，之后程序返回一个非空的指针。

虽然是这样小的一段程序，但是在rtalloc1的处理过程中有很多选择。该函数有7个不同的流程，如图19-3所示。

	report参数	RTF_CLONING标志	RTM_RESOLVE返回	RTF_XRESOLVE标志	产生的路由消息	rt_refcnt	返回值
查找失败	0						空
	1				RTM_MISS		空
查找成功		0				++	ptr
	0					++	ptr
	1	1	OK	0		++	ptr
	1	1	OK	1	RTM_RESOLVE	++	ptr
	1	1	差错		RTM_MISS	++	ptr

图19-3 rtalloc1 处理过程小结

需要解释的是，如果存在默认路由，前两行(找不到路由表项的流程)是不可能出现的。还有，在第5、第6两行中的rt_refcnt也做了递增，因为这两行在调用rtrequest时使用了RTM_RESOLVE参数，递增在rtrequest中完成。

19.3 宏RTFREE和rtfree函数

宏RTFREE，如图19-4所示，仅在引用计数小于等于1时才调用rtfree函数；否则，它仅完成引用计数的递减。

209-213 rtfree函数如图19-5所示。当不存在对rtentry结构的引用时，函数就释放该结构。例如，在图22-7中，当释放一个协议控制块时，如果它指向一个路由表项，则需要调用rtfree。

105-115 首先递减路由表项的引用计数，如果它小于等于0并且该路由不可用，则该表项可以被释放。如果该表项设有RNF_ACTIVE或RNF_ROOT标志，那么这是一个内部差错。因为，如果设有RNF_ACTIVE，那么该结构仍是路由表的一部分；如果设有RNF_ROOT，那么它是一个由rn_inithead创建的标有end的结构。

route.h

```
209 #define RTFREE(rt) \
210     if ((rt)->rt_refcnt <= 1) \
211         rtfree(rt); \
212     else \
213         (rt)->rt_refcnt--;      /* no need for function call */
```

route.h

图19-4 宏RTFREE

route.c

```
105 void
106 rtfree(rt)
107 struct rtentry *rt;
108 {
109     struct ifaddr *ifa;

110     if (rt == 0)
111         panic("rtfree");
112     rt->rt_refcnt--;
113     if (rt->rt_refcnt <= 0 && (rt->rt_flags & RTF_UP) == 0) {
114         if (rt->rt_nodes->rn_flags & (RNF_ACTIVE | RNF_ROOT))
115             panic("rtfree 2");
116         rttrash--;
117         if (rt->rt_refcnt < 0) {
118             printf("rtfree: %x not freed (neg refs)\n", rt);
119             return;
120         }
121         ifa = rt->rt_ifa;
122         IFAFREE(ifa);
123         Free(rt_key(rt));
124         Free(rt);
125     }
126 }
```

route.c

图19-5 rtfree 函数：释放一个rtentry 结构

116 rttrash是一个用于调试的计数器，记录那些不在选路树中但仍未释放的路由表项的数目。当rtrequest开始删除路由时，它被递增，然后在这儿递减。正常情况下，它的值应该是0。

1. 释放接口引用

117-122 先查看引用计数。确认引用计数非负后，IFAFREE将递减ifaddr结构的引用计数。当计数值递减为零时，调用ifafree释放它。

2. 释放选路存储器

123-124 释放由路由表项关键字及其网关所占的存储器。我们会看到rt_setgate把它们分配在存储器的同一个连着的块中。因此，只调用一个Free就可以同时把它们释放。最后还要释放rtentry结构。

路由表引用计数

路由表引用计数(rt_refcnt)的处理与其他许多引用计数的处理不同。我们看到，在图18-2中，大多数路由的引用计数为0，而这些没有引用的路由表项并没有被删除。原因就在rtfree中：只有当RTF_UP标志被删除时，引用计数为0的表项才会被删除。而仅当从选路树中删除路由时，该标志才会被rtrequest删除。

大多数路由是按如下方式使用的。

- 如果到某接口的路由是在配置该接口时自动创建的（典型的，例如以太网接口的配置），则rtinit用命令参数RTM_ADD来调用rtquest，以创建新的路由表项，并设置它的引用计数为1。然后，rtinit在退出前把该引用计数递减成0。

  对于点到点接口的处理过程也是类似的，所以路由的引用计数也是从0开始。

  如果路由是由route命令手工创建的，或者是由选路守护进程创建的，处理过程同样是类似的。route_output用命令参数RTM_ADD来调用rtrequest，并设置新路由的引用计数为1。在退出前，route_output把该引用计数递减到0。

  因此，所有新创建的路由都是从引用计数0开始的。

- 当TCP或UDP在插口上发送IP数据包时，ip_output调用rtalloc，rtalloc再调用rtalloc1。如图19-3所示，如果找到了路由，rtalloc1就会递增其引用计数。

  所查找到的路由称为被持路由(held route)，因为协议持有指向路由表项的指针，该指针通常被包含在协议控制块中的route结构里。一个被其他协议持有的rtentry结构是不能被删除的。所以，在rtfree中，当引用计数为0时，rtentry结构才能被删除。

- 协议通过调用RTFREE或rtfree来释放被持路由。在图8-24中，当ip_output检测到目的地址改变时，我们已经使用了这种处理。在第22章中，释放持有路由的协议控制块时，我们还会遇到这种处理。

在后面的代码中可能会引起混淆的是，rtalloc1经常被调用以判断路由是否存在，而调用者并非试图持有该路由。因为rtalloc1递增了该路由的引用计数器，所以调用者就立即递减该计数器。

考虑一个被rtrequest删除的路由。它的RTF_UP标志被清除，并且，如果没有被持有(它的引用计数为0)，就要调用rtfree。但rtfree认为引用计数小于0是错的，所以rtrequest查看它的引用计数，如果小于等于0，就递增该计数值并调用rtfree。通常，这将使引用计数变成1，之后，rtfree把引用计数递减到0，并删除该路由。

19.4 rtrequest函数

rtrequest函数是添加和删除路由表项的关键点。图19-6给出了调用它的一些其他函数。

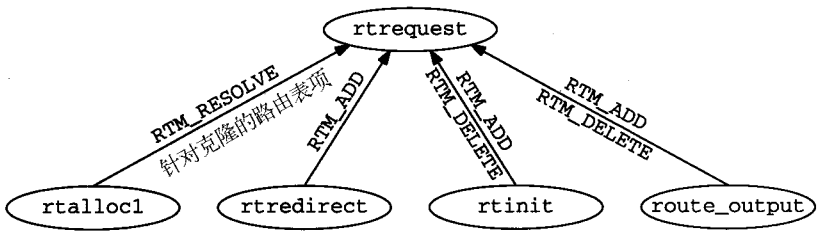


图19-6 调用rtrequest 的函数

rtrequest是一个switch语句，每个case对应一个命令：RTM_ADD、RTM_DELETE和RTM_RESOLVE。图19-7给出了该函数的开头一段以及RTM_DELETE命令的处理。

290-307 第二个参数，dst，是一个插口地址结构，它指定在路由表中添加或删除的表项。

表项中的sa_family用于选择路由表。如果flags参数指出该路由是主机路由(而不是到某个网络的路由)，则设置netmask指针为空，忽略调用者设置的任何值。

route.c

```
290 int
291 rtrequest(req, dst, gateway, netmask, flags, ret_nrt)
292 int     req, flags;
293 struct sockaddr *dst, *gateway, *netmask;
294 struct rtentry **ret_nrt;
295 {
296     int     s = splnet();
297     int     error = 0;
298     struct rtentry *rt;
299     struct radix_node *rn;
300     struct radix_node_head *rnh;
301     struct ifaddr *ifa;
302     struct sockaddr *ndst;
303 #define senderr(x) { error = x ; goto bad; }
304     if ((rnh = rt_tables[dst->sa_family]) == 0)
305         senderr(ESRCH);
306     if (flags & RTF_HOST)
307         netmask = 0:
308     switch (req) {
309     case RTM_DELETE:
310         if ((rn = rnh->rnh_deladdr(dst, netmask, rnh)) == 0)
311             senderr(ESRCH);
312         if (rn->rn_flags & (RNF_ACTIVE | RNF_ROOT))
313             panic("rtrequest delete");
314         rt = (struct rtentry *) rn;
315         rt->rt_flags &= ~RTF_UP;
316         if (rt->rt_gwroute) {
317             rt = rt->rt_gwroute;
318             RTFREE(rt);
319             (rt = (struct rtentry *) rn)->rt_gwroute = 0;
320         }
321         if ((ifa = rt->rt_ifa) && ifa->ifa_rtrequest)
322             ifa->ifa_rtrequest(RTM_DELETE, rt, SA(0));
323         rttrash++;
324         if (ret_nrt)
325             *ret_nrt = rt;
326         else if (rt->rt_refcnt <= 0) {
327             rt->rt_refcnt++;
328             rtfree(rt);
329         }
330         break;
```

route.c

图19-7 rtrequest 函数：RTM_DELETE 命令

1. 从选路树中删除路由

309-315 rnh_deladdr函数(图18-17中的rn_delete)从选路树中删除表项，返回相应rtentry结构的指针，并清除RTF_UP标志。

2. 删除对网关路由表项的引用

316-320 如果该表项是一个经过某网关的非直接路由，则RTFREE递减该网关路由表项的引用计数。如它的引用计数被减为0，则删除它。设置rt_gwroute指针为空，并将rt设置成原来要删除的表项。

3. 调用接口请求函数

321-322 如果该表项定义了ifa_rtrequest函数，就调用该函数。ARP会使用该函数，例如，在第21章中用它来删除对应的ARP表项。

4. 返回指针或删除引用

323-330 因为该表项在接着的代码里不一定被删除，所以递增全局变量rttrash。如果调用者需要选路树中被删除的rtentry结构的指针(即如果ret_nrt非空)，则返回该指针，但此时不能释放该表项：调用者必须在使用完该表项后调用rtfree来删除它。如果ret_nrt为空，则该表项被释放：如果它的引用计数小于等于0，则递增该计数值，并调用rtfree。break语句将使函数退出。

图19-8给出了函数的下一部分，用于处理RTM_RESOLVE命令。只有rtalloc1能够携带此命令参数调用本函数。也只有在从一个设有RTF_CLONING标志的表项中克隆一个新的表项时，rtalloc1才这样用。

route.c

```
331     case RTM_RESOLVE:
332         if (ret_nrt == 0 || (rt = *ret_nrt) == 0)
333             senderr(EINVAL);
334         ifa = rt->rt_ifa;
335         flags = rt->rt_flags & ~RTF_CLONING;
336         gateway = rt->rt_gateway;
337         if ((netmask = rt->rt_genmask) == 0)
338             flags |= RTF_HOST;
339         goto makeroute;
```

route.c

图19-8 rtrequest 函数：RTM_RESOLVE 命令

331-339 最后一个参数，ret_nrt，在这个命令里的用途不同：它是一个设有RTF_CLONING标志的路由表项的指针(图19-2)。新的表项具有相同的rt_ifa指针、相同的rt_gateway和相同的标志(RTF_CLONING标志被清除)。如果被克隆表项的rt_genmask指针为空，则新表项是一个主机路由，因此要设置它的RTF_HOST标志；否则新表项为网络路由，其网络掩码通过复制rt_genmask得到。在本节的结尾部分，我们给出了克隆带网络掩码的路由的一个例子。这个case将跳转至下个图中的makeroute标记处继续进行。

图19-9给出了RTM_ADD命令的代码。

5. 定位相应的接口

340-342 函数ifa_ifwithroute为目的(dst)查找适当的本地接口，并返回指向该接口的ifaddr结构的指针。

6. 为路由表项分配存储器

343-348 分配了一个rtentry结构。在前一章中我们知道，该结构包含了两个选路树的radix_node结构及其他路由信息。该结构被清零，之后，其标志rt_flags被设置成调用本函数的flags参数，同时再设置RTF_UP标志。

7. 分配并复制网关地址

349-352 rt_gateway函数(图19-11)为路由表(dst)及其gateway分配了存储器，然后将gateway复制到新分配的存储器中，并设置指针rt_key、rt_gateway和rt_gwroute。

8. 复制目的地址

route.c

```
340     case RTM_ADD:
341         if ((ifa = ifa_ifwithroute(flags, dst, gateway)) == 0)
342             senderr(ENETUNREACH);
343       makeroute:
344         R_Malloc(rt, struct rtentry *, sizeof(*rt));
345         if (rt == 0)
346             senderr(ENOBUFS);
347         Bzero(rt, sizeof(*rt));
348         rt->rt_flags = RTF_UP | flags;
349         if (rt_setgate(rt, dst, gateway)) {
350             Free(rt);
351             senderr(ENOBUFS);
352         }
353         ndst = rt_key(rt);
354         if (netmask) {
355             rt_maskedcopy(dst, ndst, netmask);
356         } else
357             Bcopy(dst, ndst, dst->sa_len);
358         rn = rnh->rnh_addaddr((caddr_t) ndst, (caddr_t) netmask,
359                               rnh, rt->rt_nodes);
360         if (rn == 0) {
361             if (rt->rt_gwroute)
362                 rtfree(rt->rt_gwroute);
363             Free(rt_key(rt));
364             Free(rt);
365             senderr(EEXIST);
366         }
367         ifa->ifa_refcnt++;
368         rt->rt_ifa = ifa;
369         rt->rt_ifp = ifa->ifa_ifp;
370         if (req == RTM_RESOLVE)
371             rt->rt_rmx = (*ret_nrt)->rt_rmx;    /* copy metrics */
372         if (ifa->ifa_rtrequest)
373             ifa->ifa_rtrequest(req, rt, SA(ret_nrt ? *ret_nrt : 0));
374         if (ret_nrt) {
375             *ret_nrt = rt;
376             rt->rt_refcnt++;
377         }
378         break;
379     }
380   bad:
381     splx(s);
382     return (error);
383 }
```

route.c

图19-9 rtrequest 函数：RTM_ADD 命令

353-357 把目的地址(路由表表项dst)复制到rn_key所指向的存储器中。如果提供了网络掩码，则rt_maskedcopy对dst和netmask进行逻辑与运算，得到新的表项。否则，dst就会被复制成新的表项。对dst和netmask进行逻辑与运算是为了确保表中的表项已经和它的掩码进行了与运算。这样，查找表项与表中的表项进行比较时，只需要另外对查找表项和掩码进行逻辑与运算就可以了。例如，下面的这个命令向以太网接口le0添加了另一个IP地址(一个别名)，其子网为12而不是13。

```
bsdi $ ifconfig le0 inet 140.252.12.63 netmask 0xfffffe0 alias
```

该例子中存在的一个问题是，我们所指定的全 1的主机号是错误的。不过，该表项存入路由表后，我们用netstat验证可知该地址已经和掩码进行过逻辑与运算了。

```
Destination     Gateway         Flags  Refs    Use  Interface
140.252.12.32   link#1          U C       0      0       le0
```

9. 往选路树中添加表项

358-366 rnh_addaddr函数(图18-17中的rn_addroute)向选路树中添加这个rtentry结构，其中附带了它的目的地址和掩码。如果有差错产生，则释放该结构，并返回EEXIST(即，该表项已经存在于路由表中了)。

10. 保存接口指针

367-369 递增ifaddr结构的引用计数，并保存ifaddr和ifnet结构的指针。

11. 为新克隆的路由复制度量

370-371 如果命令是RTM_RESOLVE(不是RTM_ADD)，则把被克隆的表项中的整个度量结构复制到新的表项里。如果命令是RTM_ADD，则调用者可在函数返回后设置该度量值。

12. 调用接口请求函数

372-373 如果为该表项定义了ifa_rtrequest函数，则调用该函数。对于RTM_ADD和RTM_RESOLVE命令，ARP都要用该函数来完成一些额外的处理。

13. 返回指针并递增引用计数

374-378 如果调用者需要该新结构的指针，则通过ret_nrt返回该指针，并将引用计数值从0递增到1。

例：克隆的带网络掩码的路由

仅当rtrequest的RTM_RESOLVE命令创建克隆路由时，才使用rt_genmask的值。如果rt_genmask指针非空，则它指向的插口地址结构就成了新创建路由的网络掩码。在我们的路由表中，即图 18-2，克隆的路由是针对本地以太网和多播地址的。下面的例子引自[Sklower 1991]，它提供了克隆路由的不同用法。另外一个例子见习题 19.2。

考虑一个B类网络，如128.1，它在点到点链路之外。子网掩码是0xffffff00，其中含8比特的子网号和8比特的主机号。我们要为所有可能的 254个子网提供路由表项，这些表项的网关是与本机直接相连的路由器，该路由器知道如何到达与 128.1网络相连的链路。

假设该网关不是我们的默认路由器，则最简单的方法就是创建单个表项，该表项以128.1.0.0为目的、以0xffff0000为掩码。可是，假设128.1网络的拓扑使所有可能的254个子网中的每一个都有不同的运营特性： RTTs、MTUs和时延等。那么如果每个子网都有单独的路由表项，我们就能够看到，无论何时连接断开后， TCP都会刷新该路由表项的统计值，如路由的RTT、RTT变量等(图27-3)。尽管我们可以用route命令手工地为254个子网中的每一个子网都添加路由表项，但更好的方法是采用克隆机制。

由系统管理员先创建一个以 128.1.0.0为目的地，以0xffff0000为网络掩码的表项。再设置其RTF_CLONING标志，并设置genmask为0xffffff00(与网络掩码不同)。这时，如果在路由表中查找 128.1.2.3，而路由表中没有子网 128.1.2的表项，那么具有掩码0xffff0000的网络128.1的表项为最佳匹配。因为该表项设有RTF_CLONING标志，所以要创建一个新的表项，新表项以128.1.2为目的地，以0xffffff00(genmask的值)为网络掩码。

这样，下一次引用该子网内的主机时，如128.1.2.88，最佳匹配就是新创建的表项。

19.5 rt_setgate函数

选路树中的每个叶子都有一个表项（rt_key，也就是在rtentry结构开头的radix_node结构的rn_key成员)和一个相关联的网关(rt_gateway)。在创建路由表项时，它们都被指定为插口地址结构。rt_setgate为这两个结构分配存储器，如图19-10所示。

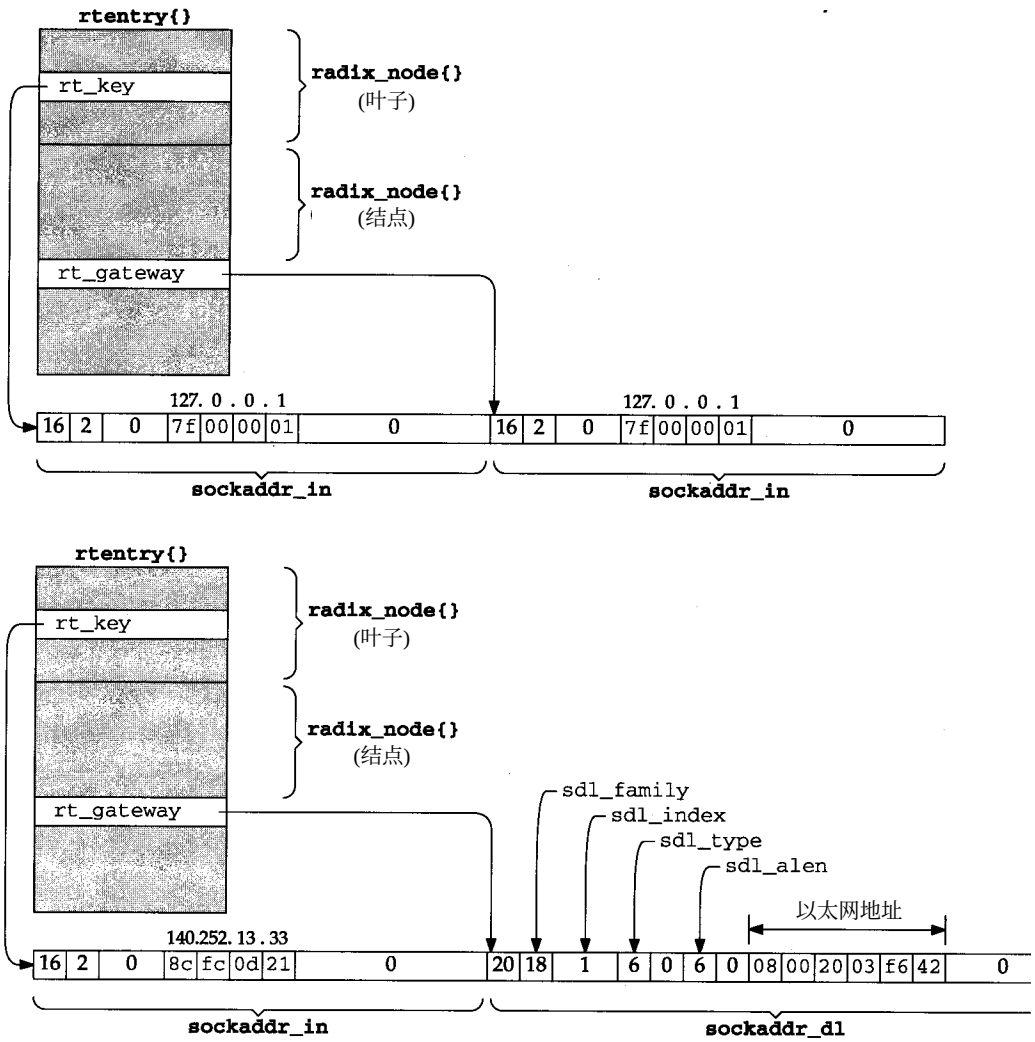


图19-10 路由表表项和相关网关示例

这个例子给出了图18-2中的两个表项，它们的表项分别是127.0.0.1和140.252.13.33。前一个的网关成员指向一个Internet插口地址结构，后一个的网关成员指向一个含以太网地址的数据链路插口地址。前一个是在系统初始化时，由route系统将其添加到路由表中的；后一个是由ARP创建的。

在图19-11中，我们有意识地把两个由rt_key指向的结构紧挨着画在一起，因为它们是由rt_setgate一起分配的。

route.c

```
384 int
385 rt_setgate(rt0, dst, gate)
386 struct rtentry *rt0;
387 struct sockaddr *dst, *gate;
388 {
389     caddr_t new, old;
390     int     dlen = ROUNDUP(dst->sa_len), glen = ROUNDUP(gate->sa_len);
391     struct rtentry *rt = rt0;

392     if (rt->rt_gateway == 0 || glen > ROUNDUP(rt->rt_gateway->sa_len)) {
393         old = (caddr_t) rt_key(rt);
394         R_Malloc(new, caddr_t, dlen + glen);
395         if (new == 0)
396             return 1;
397         rt->rt_nodes->rn_key = new;
398     } else {
399         new = rt->rt_nodes->rn_key;
400         old = 0;
401     }
402     Bcopy(gate, (rt->rt_gateway = (struct sockaddr *) (new + dlen)), glen);
403     if (old) {
404         Bcopy(dst, new, dlen);
405         Free(old);
406     }
407     if (rt->rt_gwroute) {
408         rt = rt->rt_gwroute;
409         RTFREE(rt);
410         rt = rt0;
411         rt->rt_gwroute = 0;
412     }
413     if (rt->rt_flags & RTF_GATEWAY) {
414         rt->rt_gwroute = rtalloc1(gate, 1);
415     }
416     return 0;
417 }
```

route.c

图19-11 rt_setgate 函数

1. 依据插口地址结构设置长度

384-391 dlen是目的插口地址结构的长度，glen是网关插口地址结构的长度。ROUNDUP宏把数值上舍入成4的倍数个字节，但大多数插口地址结构的长度本身就是4的倍数。

2. 分配存储器

392-401 如果还没有给该路由表表项和网关分配存储器，或glen大于当前rt_gateway所指向的结构的长度，则分配一片新的存储器，并使rn_key指向新分配的存储器。

3. 使用分配给表项和网关的存储器

398-401 由于已经给表项和网关分配了一片足够大小的存储器，因此，直接将new指向这个已经存在的存储器。

4. 复制新网关

402 复制新的网关结构，并且设置rt_gateway，使其指向插口地址结构。

5. 从原有的存储器中将表项复制到新存储器中

403-406 如果分配了新的存储器，则在网关字段被复制前，先复制路由表表项dst，并释放原有的存储器片。

6. 释放网关路由指针

407-412 如果该路由表项含有非空的rt_gwroute指针，则用RTFREE释放该指针所指向的结构，并设置rt_gwroute为空。

7. 查找并保存新的网关路由指针

413-415 如果路由表项是一个非直接路由，则rtalloc1查找新网关的路由表项，并将它保存在rt_gwroute中。如果非直接路由指定的网关无效，则rt_setgate并不返回任何差错，但rt_gwroute会是一个空指针。

19.6 rtinit函数

Internet协议添加或删除相关接口的路由时，对rtinit的调用有四个。

- 在设置点到点接口的目的地址时，in_control调用rtinit两次。第一次调用指定RTM_DELETE命令，以删除所有现存的到该目的地址的路由(图6-21)；第二次调用指定RTM_ADD命令，以添加新路由。
- in_ifinit调用rtinit为广播网络添加一条网络路由或为点到点链路(图6-19)添加一条主机路由。如果是给以太网接口添加的路由，则in_ifinit自动设置其RTF_CLONING标志。
- in_ifscrub调用rtinit，以删除一个接口现存的路由。

图19-12给出了rtinit函数的第一部分。cmd参数只能是RTM_ADD或RTM_DELETE。

route.c

```
441 int
442 rtinit(ifa, cmd, flags)
443 struct ifaddr *ifa;
444 int     cmd, flags;
445 {
446     struct rtentry *rt;
447     struct sockaddr *dst;
448     struct sockaddr *deldst;
449     struct mbuf *m = 0;
450     struct rtentry *nrt = 0;
451     int     error;

452     dst = flags & RTF_HOST ? ifa->ifa_dstaddr : ifa->ifa_addr;
453     if (cmd == RTM_DELETE) {
454         if ((flags & RTF_HOST) == 0 && ifa->ifa_netmask) {
455             m = m_get(M_WAIT, MT_SONAME);
456             deldst = mtod(m, struct sockaddr *);
457             rt_maskedcopy(dst, deldst, ifa->ifa_netmask);
458             dst = deldst;
459         }
460         if (rt = rtalloc1(dst, 0)) {
461             rt->rt_refcnt--;
462             if (rt->rt_ifa != ifa) {
463                 if (m)
464                     (void) m_free(m);
465                 return (flags & RTF_HOST ? EHOSTUNREACH
466                         : ENETUNREACH);
467             }
468         }
469     }
470     error = rtrequest(cmd, dst, ifa->ifa_addr, ifa->ifa_netmask,
471                       flags | ifa->ifa_flags, &nrt);
472     if (m)
473         (void) m_free(m);
```

route.c

图19-12 rt_init 函数：调用 rtrequest 处理命令

1. 为路由获取目的地址

452 如果是一个到达某主机的路由，则目的地址是点到点链路的另一端。否则，我们处理的就是一个网络路由，其目的地址是接口的单播地址 (经ifa_netmask掩码过的)。

2. 用网络掩码给网络地址掩码

453-459 如果要删除路由，则必须在路由表中查找该目的地址，并得到它的路由表项。如果要删除的是一个网络路由且接口拥有相关联的网络掩码，则分配一个 mbuf，用rt_maskedcopy对目的地址和调用参数中的掩码地址进行逻辑与运算，并将结果复制到mbuf中。令dst指向mbuf中掩码过的复制值，它就是下一步要查找的目的地址。

3. 查找路由表项

460-469 rtalloc1在路由表中查找目的地址，如果能找到，则先递减该表项的引用计数 (因为rtalloc1递增了该引用计数)。如果路由表中该接口的 ifaddr指针不等于调用者的参数，则返回一个差错。

4. 处理请求

470-473 rt_request执行RTM_ADD或RTM_DELETE命令。当rt_request返回时，如果之前分配了mbuf，则释放它。

图19-13给出了rtinit的后半部分。

route.c

```
474     if (cmd == RTM_DELETE && error == 0 && (rt = nrt)) {
475         rt_newaddrmsg(cmd, ifa, error, nrt);
476         if (rt->rt_refcnt <= 0) {
477             rt->rt_refcnt++;
478             rtfree(rt);
479         }
480     }
481     if (cmd == RTM_ADD && error == 0 && (rt = nrt)) {
482         rt->rt_refcnt--;
483         if (rt->rt_ifa != ifa) {
484             printf("rtinit: wrong ifa (%x) was (%x)\n", ifa,
485                    rt->rt_ifa);
486             if (rt->rt_ifa->ifa_rtrequest)
487                 rt->rt_ifa->ifa_rtrequest(RTM_DELETE, rt, SA(0));
488             IFAFREE(rt->rt_ifa);
489             rt->rt_ifa = ifa;
490             rt->rt_ifp = ifa->ifa_ifp;
491             ifa->ifa_refcnt++;
492             if (ifa->ifa_rtrequest)
493                 ifa->ifa_rtrequest(RTM_ADD, rt, SA(0));
494         }
495         rt_newaddrmsg(cmd, ifa, error, nrt);
496     }
497     return (error);
498 }
```

route.c

图19-13 rtint 函数：后半部分

5. 删除成功时产生一个选路消息

474-480 如果删除了一个路由，并且 rtrequest返回0和被删除的rtentry结构的指针 (nrt中)，就用rt_newaddrmsg产生一个选路插口消息。如果引用计数小于等于 0，则递增该引用计数，并用rtfree释放该路由。

6. 成功添加

481-482 如果添加了一个路由，并且rtrequest返回了0和被添加的rtentry结构的指针(nrt)，就递减其引用计数(因为rtrequest递增了该引用计数)。

7. 不正确的接口

483-494 如果新路由表项中接口的ifaddr指针不等于调用参数，则表明有差错产生。利用rtrequest，通过调用ifa_ifwithroute来测定新路由中的ifa指针(rtrequest函数如图19-9所示)。产生这个差错后，做如下步骤：向控制台输出一条出错消息；如果定义了ifa_rtrequest函数，就以RTM_DELETE为参数调用它；释放ifaddr结构；设置rt_ifa指针为调用者指定的值；递增接口的引用计数；如果定义了 ifa_rtrequest函数，就以RTM_ADD为参数调用它。

8. 产生选路消息

495 用rt_newaddrmsg为RTM_ADD命令产生一个选路插口消息。

19.7 rtredirect函数

当收到一个ICMP重定向后，icmp_input调用rtredirect及pfctlinput(图11-27)。后一个函数又调用udp_ctlinput和tcp_ctlinput，这两个函数遍历所有的UDP和TCP协议控制块(PCB)。如果PCB连接到一个外部地址，而到该外部地址的方向已经被改变，并且该PCB持有到那个外部地址的路由，则调用 rtfree释放该路由。下一次使用这些控制块发送到该外部地址的IP数据报时，就会调用 rtalloc，并在路由表中查找该目的地址，很可能会找到一条新(改变过方向的)路由。

rtredirect函数的作用是验证重定向中的信息，并立即更新路由表，产生选路插口消息。图19-14给出了rtredirect函数的前半部分。

147-157 函数的参数包括：dst，导致重定向的数据报的目的 IP地址(图8-18中的HD)；gateway，路由器的IP地址，用作该目的的新网关字段 (图8-18中的R2)；netmask，空指针；flags，设置了RTF_GATEWAY标志和RTF_HOST标志；src，发送重定向的路由器的IP地址(图8-18中的R1)；rtp，空指针。需要指出的是， icmp_input调用本函数时，参数netmask和rtp是空指针，但是其他协议调用本函数时，这两个参数未必为空指针。

1. 新路由必须直接相连

158-162 新的网关必须是直接相连的，否则该重定向无效。

2. 查找目的地址的路由表项并验证重定向

163-177 调用rtalloc1在路由表中查找到目的地址的路由。验证重定向时，下列条件必须为真，否则该重定向无效，并且函数返回一个差错。要注意的一点是， icmp_input会忽略从rtredirect返回的任何差错。ICMP也会忽略它，即不会为一个无效的重定向而产生一个差错信息。

- 必须未设置RTF_DONE标志；
- rtalloc必须已找到一个到dst的路由表项；
- 发送重定向的路由器的地址(src)必须等于当前为目的地址设置的rt_gateway；
- 新网关的接口(由ifa_ifwithnet返回的ifa结构)必须等于当前为目的地址设置的接口(rt_ifa)，也就是说，新网关必须和当前网关在同一个网络上；并且

• 新网关不能把到这个主机的路由改变为到它自己，也就是说，不能存在与 gateway相等的带有单播地址或广播地址的连接着的接口。

route.c

```
147 int
148 rtredirect(dst, gateway, netmask, flags, src, rtp)
149 struct sockaddr *dst, *gateway, *netmask, *src;
150 int     flags;
151 struct rtentry **rtp;
152 {
153     struct rtentry *rt;
154     int     error = 0;
155     short  *stat = 0;
156     struct rt_addrinfo info;
157     struct ifaddr *ifa;

158     /* verify the gateway is directly reachable */
159     if ((ifa = ifa_ifwithnet(gateway)) == 0) {
160         error = ENETUNREACH;
161         goto out;
162     }
163     rt = rtalloc1(dst, 0);
164     /*
165      * If the redirect isn't from our current router for this dst,
166      * it's either old or wrong.  If it redirects us to ourselves,
167      * we have a routing loop, perhaps as a result of an interface
168      * going down recently.
169      */
170 #define equal(a1, a2) (bcmp((caddr_t)(a1), (caddr_t)(a2), (a1)->sa_len) == 0)
171     if (!(flags & RTF_DONE) && rt &&
172         (!equal(src, rt->rt_gateway) || rt->rt_ifa != ifa))
173         error = EINVAL;
174     else if (ifa_ifwithaddr(gateway))
175         error = EHOSTUNREACH;
176     if (error)
177         goto done;
178     /*
179      * Create a new entry if we just got back a wildcard entry
180      * or if the lookup failed.  This is necessary for hosts
181      * which use routing redirects generated by smart gateways
182      * to dynamically build the routing tables.
183      */
184     if ((rt == 0) || (rt_mask(rt) && rt_mask(rt)->sa_len < 2))
185         goto create;
```

route.c

图19-14 rtredirect 函数：验证收到的重定向

3. 必须创建一个新路由

178-185 如果到达目的地址的路由没有找到，或找到的路由表项是默认路由，则为该目的地址创建一个新的路由。如程序注释所述，对于可访问多个路由器的主机来说，当默认路由器出错时，它可以利用这种机制来获悉正确的路由器。判断是否为默认路由的测试方法是查看该路由表项是否具有相关的掩码以及掩码的长度字段是否小于 2，因为默认路由的掩码是rn_zeros(图18-35)。

图19-15给出了rtredirect函数的后半部分。

4. 创建新的主机路由

186-195 如果到达目的地址的当前路由是一个网络路由，并且重定向是主机重定向而不是

网络重定向，那么就为该目的地址建立一个主机路由，而不必去管现存的网络路由。我们要提示的是，flags参数总是指明RTF_HOST标志，因为Net/3 ICMP把所有收到的重定向都看成主机重定向。

route.c

```
186     /*
187      * Don't listen to the redirect if it's
188      * for a route to an interface.
189      */
190     if (rt->rt_flags & RTF_GATEWAY) {
191         if (((rt->rt_flags & RTF_HOST) == 0) && (flags & RTF_HOST)) {
192             /*
193              * Changing from route to net => route to host.
194              * Create new route, rather than smashing route to net.
195              */
196           create:
197             flags |= RTF_GATEWAY | RTF_DYNAMIC;
198             error = rtrequest((int) RTM_ADD, dst, gateway,
199                               netmask, flags,
200                               (struct rtentry **) 0);
201             stat = &rtstat.rts_dynamic;
202         } else {
203             /*
204              * Smash the current notion of the gateway to
205              * this destination.  Should check about netmask!!!
206              */
207             rt->rt_flags |= RTF_MODIFIED;
208             flags |= RTF_MODIFIED;
209             stat = &rtstat.rts_newgateway;
210             rt_setgate(rt, rt_key(rt), gateway);
211         }
212     } else
213         error = EHOSTUNREACH;
214   done:
215     if (rt) {
216         if (rtp && !error)
217             *rtp = rt;
218         else
219             rtfree(rt);
220     }
221   out:
222     if (error)
223         rtstat.rts_badredirect++;
224     else if (stat != NULL)
225         (*stat)++;
226     bzero((caddr_t) & info, sizeof(info));
227     info.rti_info[RTAX_DST] = dst;
228     info.rti_info[RTAX_GATEWAY] = gateway;
229     info.rti_info[RTAX_NETMASK] = netmask;
230     info.rti_info[RTAX_AUTHOR] = src;
231     rt_missmsg(RTM_REDIRECT, &info, flags, error);
232 }
```

route.c

图19-15 rtrequest 函数：后半部分

5. 创建路由

196-201 rtrequest创建一个新路由，并将标志RTF_GATEWAY和RTF_DYNAMIC置位。参数netmask是一个空指针，这是因为新路由是一个主机路由，它的掩码是隐含的全 1比特。

stat指向一个计数器，它在后面的程序里递增。

6. 改变现存的主机路由

202-211 当到达目的地址的当前路由已经是一个主机路由时，才执行这段代码。此时，不需要创建新的表项，但需要修改现存的表项：设置RTF_MODIFIED标志，并调用rt_setgate来修改路由表项的rt_gateway字段，使其指向新的网关地址。

7. 如果目的地址直接相连，则忽略

212-213 如果到达目的地址的当前路由是一个直接路由 (没有设置RTF_GATEWAY标志)，那么该重定向针对的是一个已直接连接的目的地址。此时，函数返回 EHOSTUNREACH。

8. 返回指针并递增统计值

214-225 如果找到了路由表项，那么该表项被返回 (如果rtp非空且没有出错)或者用rtfree释放它。相关的统计值被递增。

9. 产生选路消息

226-232 rt_addrinfo结构清零，并由 rt_missmsg产生一个选路插口消息。raw_input把该消息发送到所有对重定向感兴趣的进程。

19.8 选路消息的结构

选路消息由一个定长的首部和至多 8个插口地址结构组成。该定长首部是下列三种结构中的一个：

- rt_msghdr
- if_msghdr
- ifa_msghdr

图18-11给出了产生不同消息的函数的概观图，图 18-9给出了每种消息类型所使用的结构。选路消息三种首部结构的前三个成员的数据类型及其含义是相同的，分别为：消息的长度、版本和类型。这样，消息接受者就可以对消息进行解码了。而且，每种结构都各有一个成员来编码首部之后8个可能的插口地址结构：rtm_addrs、ifm_addrs和ifam_addrs成员，它们都是一个比特掩码。

图19-16给出了最常用的结构，即 rt_msghdr。RTM_IFINFO消息使用了图 19-17中的if_msghdr结构。RTM_NEWADDR和RTM_DELADDR消息使用图19-18中的ifa_msghdr结构。

route.h

```
139 struct rt_msghdr {
140     u_short rtm_msglen;         /* to skip over non-understood messages */
141     u_char  rtm_version;        /* future binary compatibility */
142     u_char  rtm_type;           /* message type */
143     u_short rtm_index;          /* index for associated ifp */
144     int     rtm_flags;          /* flags, incl. kern & message, e.g. DONE */
145     int     rtm_addrs;          /* bitmask identifying sockaddrs in msg */
146     pid_t   rtm_pid;            /* identify sender */
147     int     rtm_seq;            /* for sender to identify action */
148     int     rtm_errno;          /* why failed */
149     int     rtm_use;            /* from rtentry */
150     u_long  rtm_inits;          /* which metrics we are initializing */
151     struct rt_metrics rtm_rmx;  /* metrics themselves */
152 };
```

route.h

图19-16 rt_msghdr 结构

if.h

```
235 struct if_msghdr {
236     u_short ifm_msglen;         /* to skip over non-understood messages */
237     u_char  ifm_version;        /* future binary compatability */
238     u_char  ifm_type;           /* message type */

239     int     ifm_addrs;          /* like rtm_addrs */
240     int     ifm_flags;          /* value of if_flags */
241     u_short ifm_index;          /* index for associated ifp */
242     struct if_data ifm_data;    /* statistics and other data about if */
243 };
```

if.h

图19-17 if_msghdr 结构

if.h

```
248 struct ifa_msghdr {
249     u_short ifam_msglen;        /* to skip over non-understood messages */
250     u_char  ifam_version;       /* future binary compatability */
251     u_char  ifam_type;          /* message type */

252     int     ifam_addrs;         /* like rtm_addrs */
253     int     ifam_flags;         /* value of ifa_flags */
254     u_short ifam_index;         /* index for associated ifp */
255     int     ifam_metric;        /* value of ifa_metric */
256 };
```

if.h

图19-18 ifa_msghdr 结构

注意，这三种不同结构的前三个成员具有相同的数据结构和含义。

三个变量rtm_addrs、ifm_addrs和ifam_addrs都是比特掩码，它们定义了首部之后的插口地址结构。图19-19给出了比特掩码用到的一些常量。

比特掩码		数组索引		rtsock.c	描述
常量	值	常量	值	中的名字	
RTA_DST	0x01	*RTAX_DST*	0	dst	目的插口地址结构
RTA_GATEWAY	0x02	*RTAX_GATEWAY*	1	gate	网关插口地址结构
RTA_NETMASK	0x04	*RTAX_NETMASK*	2	netmask	网络掩码插口地址结构
RTA_GENMASK	0x08	*RTAX_GENMASK*	3	genmask	克隆掩码插口地址结构
RTA_IFP	0x10	*RTAX_IFP*	4	ifpaddr	接口名称插口地址结构
RTA_IFA	0x20	*RTAX_IFA*	5	ifaaddr	接口地址插口地址结构
RTA_AUTHOR	0x40	*RTAX_AUTHOR*	6		重定向产生者的插口地址结构
RTA_BRD	0x80	*RTAX_BRD*	7	brdaddr	广播或点到点的目的地址
		RTAX_MAX	8		rti-into[]数组的元素个数

图19-19 用来引用rti_info 数组成员的常量

比特掩码的值可以用常数1左移数组下标指定的位数而得到。例如，0x20(RTA_IFA)是1左移五位(RTAX_IFA)。我们会在代码中看到这个过程。

插口地址结构总是按照数组下标递增的次序，一个接一个地出现的。例如，如果掩码是0x87，则第一个插口地址结构的内容为目的地址，接着是网关地址，网络掩码，最后是广播地址。

内核用图19-19中的数组下标来引用rt_addrinfo结构，如图19-20所示。该结构具有与我们所述相同的比特掩码，以表示哪些地址存在。它的另一个成员指向那些插口地址结构。

route.h

```
199 struct rt_addrinfo {
200     int     rti_addrs;          /* bitmask, same as rtm_addrs */
201     struct sockaddr *rti_info[RTAX_MAX];
202 };
```

route.h

图19-20 rt_addrinfo 结构：表示哪些地址存在的掩码和指向这些地址的指针

例如，如果rti_addrs成员中设置了RTA_GATEWAY比特，则rti_info[RTA_GATEWAY]成员就是含网关地址的插口地址结构的指针。对于Internet协议，该插口地址结构就是含网关的IP地址的sockaddr_in结构。

图19-19中的第五栏给出了文件rtsock.c中rti_info数组成员相应的名称。它们的定义形式如下：

```
#define dst info.rti_info[RTAX_DST]
```

在本章的很多源文件中我们都将遇到这些名称。元素RTAX_AUTHOR没有命名，因为进程不会向内核传递该元素。

我们已经有两次遇到过rt_addrinfo结构：在函数rtalloc1(图19-2)和rtredirect中(图19-14)。图19-21给出了rtalloc1在路由表查找失败后调用rt_missmsg时创建的该结构的格式。

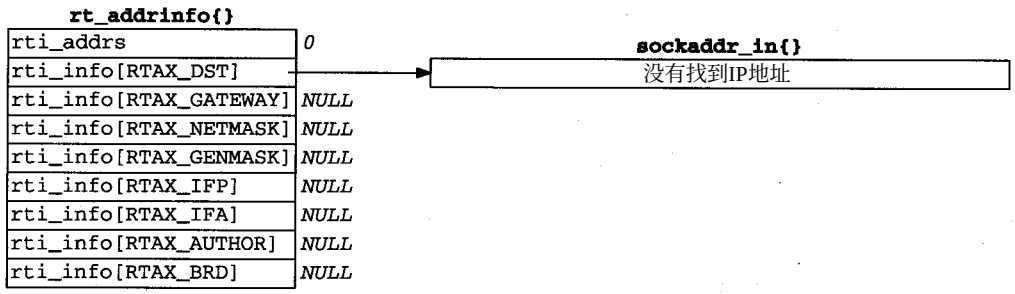


图19-21 rtalloc1 传递给rt_missmsg 的rt_addrinfo 结构

所有未使用的指针都是空指针，因为该结构在使用前被设置成 0。还要指出的是，rti_addrs成员没有被初始化成相应的比特掩码，因为在内核使用该结构时，rti_info数组中的指针为空就代表不存在该插口地址结构。仅在进程和内核之间传递的消息里该掩码才是必不可少的。

图19-22给出了rtredirect调用rt_missmsg时创建的该结构的格式。

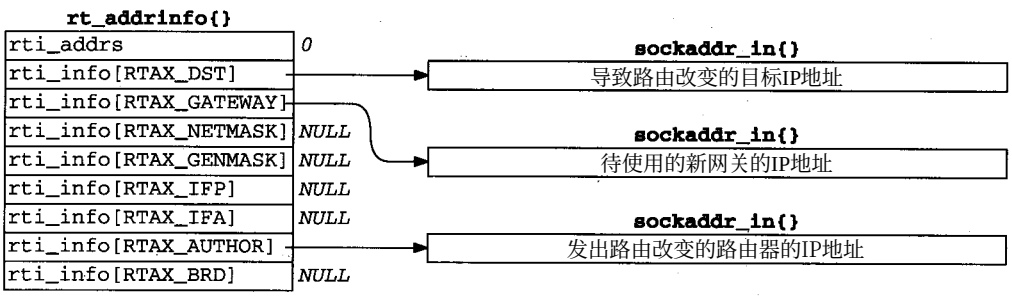


图19-22 rtredirect 传递给rt_missmsg 的rt_addrinfo 结构

下一节中将介绍该结构是如何被放置在发送到其他进程的消息里的。

图19-23给出了route_cb结构，我们会在下一节中遇到。该结构由四个计数器组成，前三个分别针对IP、XNS和OSI协议，最后一个为“任意”计数器。每个计数器分别记录相应的域中当前存在的选路插口的数目。

203-208 内核跟踪选路插口监听器的数目。这样，当不存在等待消息的进程时，内核就可以避免创建选路消息以及调用raw_input发送该消息。

route.h

```
203 struct route_cb {
204     int     ip_count;           /* IP */
205     int     ns_count;           /* XNS */
206     int     iso_count;          /* ISO */
207     int     any_count;          /* sum of above three counters */
208 };
```

route.h

图19-23 route_cb 结构：选路插口监听器的计数器

19.9 rt_missmsg函数

如图19-24所示，rt_missmsg函数使用了图19-21和图19-22中的结构，并调用rt_msg1在mbuf链中为进程创建了相应的变长消息，之后调用 raw_input将该mbuf链传递给所有相关的选路插口。

rtsock.c

```
516 void
517 rt_missmsg(type, rtinfo, flags, error)
518 int     type, flags, error;
519 struct rt_addrinfo *rtinfo;
520 {
521     struct rt_msghdr *rtm;
522     struct mbuf *m;
523     struct sockaddr *sa = rtinfo->rti_info[RTAX_DST];
524     if (route_cb.any_count == 0)
525         return;
526     m = rt_msg1(type, rtinfo);
527     if (m == 0)
528         return;
529     rtm = mtod(m, struct rt_msghdr *);
530     rtm->rtm_flags = RTF_DONE | flags;
531     rtm->rtm_errno = error;
532     rtm->rtm_addrs = rtinfo->rti_addrs;
533     route_proto.sp_protocol = sa ? sa->sa_family : 0;
534     raw_input(m, &route_proto, &route_src, &route_dst);
535 }
```

rtsock.c

图19-24 rt_missmsg 函数

516-525 如果没有任何选路插口监听器，则函数立即退出。

1. 在mbuf链中创建消息

526-528 rt_msg1(19.12节)在mbuf链中创建相应的消息，并返回该链的指针。利用图19-22中的rt_addrinfo结构，图19-25给出了所得到的mbuf链的一个例子。这些信息之所以要放在一个mbuf链中，是因为raw_input要调用sbappendaddr把该mbuf链添加到插口接收

缓存的尾部。

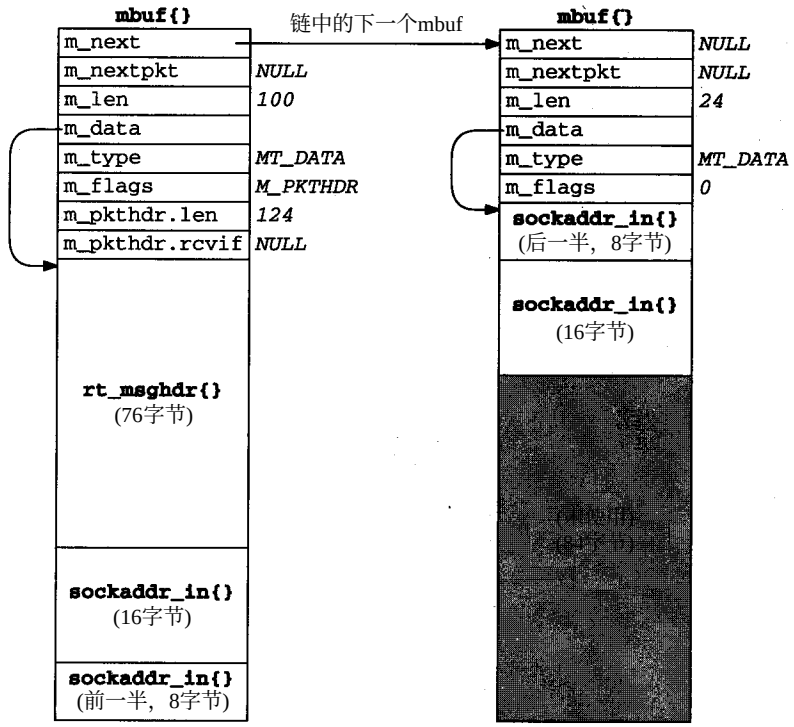


图19-25 由rt_msg1 创建的对应于图19-22的mbuf链

2. 完成消息的创建

529-532 成员rtm_flags和rtm_errno被设置成调用者传递的值。成员rtm_addrs的值是由rti_addrs复制而得到的。在图19-21和图19-22中，我们给出的rti_addrs值为0，因此，rt_msg1依据rti_info数组中的指针是否为空，来计算并保存相应的比特掩码。

3. 设置消息的协议，调用raw_input

533-534 raw_input的后三个参数指定了选路消息的协议、源和目的。这三个结构被初始化成：

```
struct sockaddr   route_dst = { 2, PF_ROUTE, };
struct sockaddr   route_src = { 2, PF_ROUTE, };
struct sockproto  route_proto = { PF_ROUTE };
```

内核不会修改前两个结构。第三个结构 sockproto，我们第一次遇到。图19-26给出了它的定义。

socket.h

```
128 struct sockproto {
129     u_short sp_family;          /* address family */
130     u_short sp_protocol;        /* protocol */
131 };
```

socket.h

图19-26 sockproto 结构

该结构的协议族成员一直保持了它的初始值 PF_ROUTE，但其协议成员的值在每一次调

用raw_input时都要进行设置。当进程调用 socket创建选路插口时，第三个参数定义了该进程所感兴趣的协议。 raw_input的调用者再把route_proto结构的sp_protocol成员设置成选路消息的协议。在 rt_missmsg这种情况下，该成员被设置成目的插口地址结构的sa_family(如果调用者指定了该值)，在图19-21和图19-22中，其值为AF_INET。

19.10 rt_ifmsg函数

在图4-30中我们可以看出， if_up和if_dowm都调用了图19-27中的rt_ifmsg。在接口连接或断开时，该函数被用来产生一个选路插口消息。

rtsock.c

```
540 void
541 rt_ifmsg(ifp)
542 struct ifnet *ifp;
543 {
544     struct if_msghdr *ifm;
545     struct mbuf *m;
546     struct rt_addrinfo info;

547     if (route_cb.any_count == 0)
548         return;

549     bzero((caddr_t) & info, sizeof(info));
550     m = rt_msg1(RTM_IFINFO, &info);
551     if (m == 0)
552         return;

553     ifm = mtod(m, struct if_msghdr *);
554     ifm->ifm_index = ifp->if_index;
555     ifm->ifm_flags = ifp->if_flags;
556     ifm->ifm_data = ifp->if_data;    /* structure assignment */
557     ifm->ifm_addrs = 0;

558     route_proto.sp_protocol = 0;
559     raw_input(m, &route_proto, &route_src, &route_dst);
560 }
```

rtsock.c

图19-27 rt_ifmsg 函数

547-548 如果没有选路插口监听器，函数立即退出。

1. 在mbuf链中创建消息

549-552 rt_addrinfo结构被设置成0。rt_msg1在一个mbuf链中创建相应的消息。需要注意的是， rt_addrinfo结构中的所有指针都为空，选路消息仅由定长的 if_msghdr结构组成，而不含任何地址。

2. 完成消息的创建

553-557 把接口的索引、标志和if_data结构复制到mbuf中的报文里。把ifm_addrs比特掩码设置成0。

3. 设置消息的协议，调用raw_input

558-559 因为该消息可应用于所有的协议，所以其协议被设置成 0。并且该消息是关于某接口的，而不是针对特定的目的地。 raw_input把该消息传递给相应的监听器。

19.11 rt_newaddrmsg函数

从图19-13中可以看到，在接口上添加或从中删除一个地址时，rtinit要以RTM_ADD或RTM_DELETE为参数调用rt_newaddrmsg。图19-28给出了rt_newaddrmsg函数的前半部分。

rtsock.c

```
569 void
570 rt_newaddrmsg(cmd, ifa, error, rt)
571 int     cmd, error;
572 struct ifaddr *ifa;
573 struct rtentry *rt;
574 {
575     struct rt_addrinfo info;
576     struct sockaddr *sa;
577     int     pass;
578     struct mbuf *m;
579     struct ifnet *ifp = ifa->ifa_ifp;
580     if (route_cb.any_count == 0)
581         return;
582     for (pass = 1; pass < 3; pass++) {
583         bzero((caddr_t) & info, sizeof(info));
584         if ((cmd == RTM_ADD && pass == 1) ||
585             (cmd == RTM_DELETE && pass == 2)) {
586             struct ifa_msghdr *ifam;
587             int     ncmd = cmd == RTM_ADD ? RTM_NEWADDR : RTM_DELADDR;
588             ifaaddr = sa = ifa->ifa_addr;
589             ifpaddr = ifp->if_addrlist->ifa_addr;
590             netmask = ifa->ifa_netmask;
591             brdaddr = ifa->ifa_dstaddr;
592             if ((m = rt_msg1(ncmd, &info)) == NULL)
593                 continue;
594             ifam = mtod(m, struct ifa_msghdr *);
595             ifam->ifam_index = ifp->if_index;
596             ifam->ifam_metric = ifa->ifa_metric;
597             ifam->ifam_flags = ifa->ifa_flags;
598             ifam->ifam_addrs = info.rti_addrs;
599         }
```

rtsock.c

图19-28 rt_newaddrmsg 函数的前半部分：创建ifa_msghdr

580-581 如果没有选路插口监听器，函数立即退出。

1. 产生两个选路消息

582 本函数要产生两个选路报文，一个用于提供有关接口的信息，另一个提供有关地址的信息。因此，for循环执行两次，每次产生一个消息。如果命令是RTM_ADD，则第一个消息的类型是RTM_NEWADDR，第二个消息的类型是RTM_ADD；如果命令是RTM_DELETE，则第一个消息的类型是RTM_DELETE，第二个消息的类型是RTM_DELADDR。RTM_NEWADDR和RTM_DELADDR消息的首部为ifa_msghdr结构，而RTM_ADD和RTM_DELETE消息的首部为rt_msghdr结构。

583 rt_addrinfo结构被设置为0。

2. 产生至多含四个地址的消息

588-591 这四个插口地址结构包含的是有关被添加或删除的接口地址的信息，它们的指针存储于rti_info数组中。其中ifaaddr、ifpaddr、netmask和brdaddr引用的是名为info的rti_info数组中的成员，如图19-19所示。rt_msg1在mbuf链中创建了相应的消息。注意，sa也设置成指向ifa_addr结构的指针，我们将在函数的尾部看到选路消息的协议被设置成该插口地址结构的族。

把ifa_msghdr结构的其他成员设置成接口的索引、度量和标志。其比特掩码由rt_msg1设置。

图19-29给出了rt_newaddrmsg的后半部分。该部分用于创建 rt_msghdr消息，该消息中包含了有关被添加或删除的路由表项的信息。

3. 创建消息

600-609 rt_mask、rt_key和rt_gateway这三个地址结构的指针存放在rti_info数组中。sa被设置成目的地址的指针，它的族将成为选路消息的协议。 rt_msg1在mbuf链中创建相应的消息。

设置其余的rt_msghdr结构成员。其中，比特掩码由rt_msg1设置。

4. 设置消息的协议，调用raw_input

616-619 设置消息的协议，并由raw_input把消息发送给相应的监听器。函数在完成了两次循环后返回。

rtsock.c

```
600         if ((cmd == RTM_ADD && pass == 2) ||
601             (cmd == RTM_DELETE && pass == 1)) {
602             struct rt_msghdr *rtm;
603             if (rt == 0)
604                 continue;
605             netmask = rt_mask(rt);
606             dst = sa = rt_key(rt);
607             gate = rt->rt_gateway;
608             if ((m = rt_msg1(cmd, &info)) == NULL)
609                 continue;
610             rtm = mtod(m, struct rt_msghdr *);
611             rtm->rtm_index = ifp->if_index;
612             rtm->rtm_flags |= rt->rt_flags;
613             rtm->rtm_errno = error;
614             rtm->rtm_addrs = info.rti_addrs;
615         }
616         route_proto.sp_protocol = sa ? sa->sa_family : 0;
617         raw_input(m, &route_proto, &route_src, &route_dst);
618     }
619 }
```

rtsock.c

图19-29 rt_newaddrmsg 函数的后半部分：创建rt_msghdr 消息

19.12 rt_msg1函数

前三节的函数都调用 rt_msg1函数来创建一个相应的选路消息。图 19-25还给出了一个mbuf链，该链是由rt_msg1按照图19-22中的rt_msghdr和rt_addrinfo结构创建的。图19-30给出了本函数的代码。

rtsock.c

```
399 static struct mbuf *
400 rt_msg1(type, rtinfo)
401 int     type;
402 struct rt_addrinfo *rtinfo;
403 {
404     struct rt_msghdr *rtm;
405     struct mbuf *m;
406     int     i;
407     struct sockaddr *sa;
408     int     len, dlen;

409     m = m_gethdr(M_DONTWAIT, MT_DATA);
410     if (m == 0)
411         return (m);
412     switch (type) {

413     case RTM_DELADDR:
414     case RTM_NEWADDR:
415         len = sizeof(struct ifa_msghdr);
416         break;

417     case RTM_IFINFO:
418         len = sizeof(struct if_msghdr);
419         break;

420     default:
421         len = sizeof(struct rt_msghdr);
422     }
423     if (len > MHLEN)
424         panic("rt_msg1");
425     m->m_pkthdr.len = m->m_len = len;
426     m->m_pkthdr.rcvif = 0;
427     rtm = mtod(m, struct rt_msghdr *);
428     bzero((caddr_t) rtm, len);

429     for (i = 0; i < RTAX_MAX; i++) {
430         if ((sa = rtinfo->rti_info[i]) == NULL)
431             continue;
432         rtinfo->rti_addrs |= (1 << i);
433         dlen = ROUNDUP(sa->sa_len);
434         m_copyback(m, len, dlen, (caddr_t) sa);
435         len += dlen;
436     }
437     if (m->m_pkthdr.len != len) {
438         m_freem(m);
439         return (NULL);
440     }
441     rtm->rtm_msglen = len;
442     rtm->rtm_version = RTM_VERSION;
443     rtm->rtm_type = type;
444     return (m);
445 }
```

rtsock.c

图19-30 rt_msg1 函数：获取并初始化mbuf

1. 得到mbuf并确定消息首部的长度

399-422 获得一个含分组首部的mbuf，并将分组消息定长部分的长度存入len中。图18-9中各种类型的消息里，有两个使用ifa_msghdr结构，有一个使用if_msghdr结构，其余的九个使用rt_msghdr结构。

2. 验证结构是否适合mbuf

423-424 定长结构的大小必须完全适合分组首部 mbuf的数据部分，因为该 mbuf指针将被mtod转换成一个结构指针，之后通过指针来引用该结构。三个结构中最大的为 if_msghdr，其长度为84，小于MHLEN(100)。

3. 初始化mbuf分组首部并使结构清零

425-428 初始化分组首部的两个字段，并将mbuf中的结构设置成0。

4. 将插口地址结构复制到mbuf链中

429-436 调用者传递了一个rt_addrinfo结构的指针。与rti_info中所有非空指针相对应的插口地址结构都被m_copyback复制到mbuf里。将数值1左移下标RTX_*xxx*对应的位数就可以得到相应的 RTA_*xxx*比特掩码 (图19-19)。将每个比特掩码用逻辑或添加到rti_addrs成员中去，调用者在函数返回时可将它保存为相应的报文结构成员。 ROUNDUP宏将每个插口地址结构的大小上舍入成下一个 4的倍数个字节。

437-440 在循环结束时，如果mbuf分组首部的长度不等于 len，则表明函数m_copyback没能获得所需的mbuf。

5. 保存长度、版本和类型

441-445 把长度、版本和报文类型保存到报文结构的前三个成员中。再次说明一下，因为所有的三种 *xxx*_msghdr结构都以相同的成员开始，所以尽管代码中的指针 rtm是一个rt_msghdr结构的指针，但它可以处理所有这三种情况。

19.13 rt_msg2函数

rt_msg1在mbuf链中创建一个选路消息，调用它的三个函数接着又调用 raw_input，从而把mbuf结构附加到一个或多个插口的接收缓存中去。与 rt_msg1不同，rt_msg2在存储器缓存中创建选路消息，而不是在 mbuf链中创建。并且rt_msg2有一个walkarg结构的参数，在选路域中处理 sysctl系统调用时，有两个函数使用该参数调用了 rt_msg2。以下是两种调用rt_msg2的情况：

1) route_output调用它处理RTM_GET命令

2) sysctl_dumpentry和sysctl_iflist调用它处理sysctl系统调用。

在给出rt_msg2的代码之前，图19-31给出了在第2种情况下使用的walkarg结构的定义。我们在遇到它的各成员时再一一介绍。

rtsock.c

```
41 struct walkarg {
42     int     w_op;              /* NET_RT_xxx */
43     int     w_arg;             /* RTF_xxx for FLAGS, if_index for IFLIST */
44     int     w_given;           /* size of process' buffer */
45     int     w_needed;          /* #bytes actually needed (at end) */
46     int     w_tmemsize;        /* size of buffer pointed to by w_tmem */
47     caddr_t w_where;           /* ptr to process' buffer (maybe null) */
48     caddr_t w_tmem;            /* ptr to our malloc'ed buffer */
49 };
```

rtsock.c

图19-31 walkarg 结构：选路域内sysctl 系统调用中使用

图19-32给出了rt_msg2函数的前半部分，它与rt_msg1的前半部分类似。

rtsock.c

```
446 static int
447 rt_msg2(type, rtinfo, cp, w)
448 int     type;
449 struct rt_addrinfo *rtinfo;
450 caddr_t cp;
451 struct walkarg *w;
452 {
453     int     i;
454     int     len, dlen, second_time = 0;
455     caddr_t cp0;
456     rtinfo->rti_addrs = 0;
457   again:
458     switch (type) {
459     case RTM_DELADDR:
460     case RTM_NEWADDR:
461         len = sizeof(struct ifa_msghdr);
462         break;
463     case RTM_IFINFO:
464         len = sizeof(struct if_msghdr);
465         break;
466     default:
467         len = sizeof(struct rt_msghdr);
468     }
469     if (cp0 = cp)
470         cp += len;
471     for (i = 0; i < RTAX_MAX; i++) {
472         struct sockaddr *sa;
473         if ((sa = rtinfo->rti_info[i]) == 0)
474             continue;
475         rtinfo->rti_addrs |= (1 << i);
476         dlen = ROUNDUP(sa->sa_len);
477         if (cp) {
478             bcopy((caddr_t) sa, cp, (unsigned) dlen);
479             cp += dlen;
480         }
481         len += dlen;
482     }
```

rtsock.c

图19-32 rt_msg2 函数：复制插口地址结构

446-455 本函数将选路消息保存在一个存储器缓存中，调用者用 cp参数指定该缓存的起始位置。调用者必须保证缓存足够长，以容纳所产生的选路消息。当 cp参数为空时，rt_msg2不保存任何结果而是处理输入参数，并返回保存结果所需要的字节总数。这样可以帮助调用者确定缓存的大小。我们可以看到 route_output就利用了这一机制，它调用本函数两次：第一次确定缓存的大小；在获得了大小无误的缓存后，再次调用本函数以保存结果。route_output调用本函数时，最后一个参数为空，但如果是作为 sysctl系统调用处理的一部分被调用时，该参数就不是空指针了。

1. 确定结构的大小

458-470 定长消息结构的大小是根据消息类型来确定的。如果 cp指针非空，则把大小等于定长消息结构长度的偏移量添加到cp指针上去。

2. 复制插口地址结构

471-482 for循环查看rti_info数组的每个元素。遇到非空指针时，设置rti_addrs比特掩码中的相应比特，并将该插口地址结构复制到cp中(如果cp指针非空)，并修改长度变量。

图19-33给出了rt_msg2函数的后半部分。其代码用于处理可选参数walkarg结构。

rtsock.c

```
483     if (cp == 0 && w != NULL && !second_time) {
484         struct walkarg *rw = w;
485         rw->w_needed += len;
486         if (rw->w_needed <= 0 && rw->w_where) {
487             if (rw->w_tmemsize < len) {
488                 if (rw->w_tmem)
489                     free(rw->w_tmem, M_RTABLE);
490                 if (rw->w_tmem = (caddr_t)
491                     malloc(len, M_RTABLE, M_NOWAIT))
492                     rw->w_tmemsize = len;
493             }
494             if (rw->w_tmem) {
495                 cp = rw->w_tmem;
496                 second_time = 1;
497                 goto again;
498             } else
499                 rw->w_where = 0;
500         }
501     }
502     if (cp) {
503         struct rt_msghdr *rtm = (struct rt_msghdr *) cp0;
504         rtm->rtm_version = RTM_VERSION;
505         rtm->rtm_type = type;
506         rtm->rtm_msglen = len;
507     }
508     return (len);
509 }
```

rtsock.c

图19-33 rt_msg2 函数：处理可选参数walkarg

483-484 当调用者传递了一个非空的walkarg结构指针且函数第一次执行到这里时，该if语句的判断条件才为真。变量second_time被初始化成0，它将在本if语句中被设置成1，然后程序往回跳转到图19-32中的标号again处。cp为空指针的测试是不必要的，因为当w指针非空时，cp指针一定是空，反之亦然。

3. 检查是否要保存数据

485-486 w_needed将增大，其增量为报文长度的值。该变量的初始值为0减去sysctl函数中用户缓存的长度。例如，如果该缓存为500比特，则w_needed的初始值为-500。只要该变量为负值，则表明缓存内还有剩余空间。在调用进程中，w_where是指向该缓存的指针。w_where为空表明调用进程不想要函数的处理结果，而仅想获得systcl处理结果的大小。因此，当w_where为空时，rt_msg2没有必要将数据复制给进程，也就是返回给调用者。同样，rt_msg2也没有必要为保存结构而申请缓存，也不需要返回到标号again处再次执行，因为再次执行是为了把结果填入到缓存里。本函数的处理实际上只有五种情况，如图19-34所示。

4. 第一次调用时或消息长度增加时分配缓存

487-493 w_tmemsize是w_tmem所指向的缓存的长度。它被sysctl_rtable初始化成0。

因此，对于给定的sysctl请求，在第一次调用rt_msg2时，必须为它分配一个缓存。同样，当产生的结果的长度增加时，必须释放原有的缓存，并重新分配一个更大的缓存。

调 用 者	cp	w	w.w_where	second_time	描 述
route_output	空	空			希望返回长度
	非空	空			希望返回结果
sysctl_rtable	空	非空	空	0	进程希望返回长度
	空	非空	非空	0	第一次执行，计算长度
	非空	非空	非空	1	第二次执行，保存结果

图19-34 rt_msg2 函数的五种执行情况

5. 返回再执行一次并保存结果

494-499 如果w_tmemsize非空，则表明该缓存已经存在或刚被分配。设置cp指向该缓存，把second_time设置成1，跳转至标号again处。因为second_time的值为1，所以第二次执行到本图的第一个语句时， if语句的判断不再为真。如果 w_tmem为空，则表明调用malloc失败，因此，把进程中的缓存指针设置成空指针以阻止返回任何结果。

6. 保存长度、版本和类型

502-509 如果cp非空，则保存消息首部的前三个成员。函数返回报文的长度。

19.14 sysctl_rtable函数

本函数处理选路插口的sysctl系统调用。如图18-11所示，net_sysctl函数调用了该函数。

在解释其源代码之前，图19-35给出了该系统调用关于路由表的一种典型的用法。这个例子来自于arp程序。

```
int       mib[6];
size_t    needed;
char      *buf, *lim, *next;
struct rt_msghdr  *rtm;

mib[0] = CTL_NET;
mib[1] = PF_ROUTE;
mib[2] = 0;
mib[3] = AF_INET;          /* address family; can be 0 */
mib[4] = NET_RT_FLAGS;     /* operation */
mib[5] = RTF_LLINFO;       /* flags; can be 0 */

if (sysctl(mib, 6, NULL, &needed, NULL, 0) < 0)
    quit("sysctl error, estimate");

if ( (buf = malloc(needed)) == NULL)
    quit("malloc");

if (sysctl(mib, 6, buf, &needed, NULL, 0) < 0)
    quit("sysctl error, retrieval");

lim = buf + needed;
for (next = buf; next < lim; next += rtm->rtm_msglen) {
    rtm = (struct rt_msghdr *)next;
    ...  /* do whatever */
}
```

图 19-35

mib数组的前三个元素引导内核调用sysctl_rtable，以处理其余的元素。

mib[4]用于指定操作的类型，共支持3种操作类型。

1) NET_RT_DUMP：返回mib[3]指定的地址族所对应的路由表。如果地址族为0，则返回所有的路由表。

针对每一个路由表项，程序都将返回一个RTM_GET选路消息。每个消息可能包含两个、三个或四个插口地址结构。这些地址结构由指针rt_key、rt_gateway、rt_netmask和rt_genmask所指向。其中最后两个指针可能为空。

2) NET_RT_FLAGS：与前一个命令相同，但mib[5]指定了一个RTF_*xxx*标志(图18-25)，程序仅返回那些设置了该标志的表项。

3) NET_RT_IFLIST：返回所有已配置接口的信息。如果mib[5]的值不是零，则程序仅返回if_index为相应值的接口。否则，返回所有ifnet链表上的接口。

针对每个接口，将返回一个RTM_IFINFO消息，该消息传递了有关接口本身的一些信息。之后用一个RTM_NEWADDR消息传递接口的if_addrlist上的每个ifaddr结构。如果mib[3]的值为非0，则RTM_NEWADDR消息仅返回那些地址族与mib[3]相匹配的地址。否则，mib[3]为0，将返回所有地址的信息。

这个操作是为了替代SIOCGIFCONF ioctl(图4-26)

与该系统调用有关的一个问题是，该系统返回信息的数量是可变的，这种变化取决于路由表表项的数目和接口的数目。因此，第一次调用sysctl所指定的第三个参数通常是个空指针，也就是说，不需要返回任何选路信息，只要把该信息所占的比特数目返回即可。从图19-35中我们可以看出，进程第一次调用sysctl之后调用了malloc，接着再调用sysctl来获取信息。第二次调用时，通过第四个参数，sysctl又返回了该比特数目(该数目与上次相比可能会有变化)。通过该数目我们可以得到指针lim，它指向的位置位于返回的最后一个字节之后。进程接着就遍历缓存中的每个消息，利用rtm_msglen可找到下一个消息。

图19-36给出了不同的Net/3程序访问路由表和接口列表时指定的这六个mib变量的值。

mib[]	arp	route	netstat	routed	gated	rwhod
0	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET
1	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE
2	0	0	0	0	0	0
3	AF_INET	0	0	AF_INET	0	AF_INET
4	NET_RT_FLAGS	NET_RT_DUMP	NET_RT_DUMP	NET_RT_IFLIST	NET_RT_IFLIST	NET_RT_IFLIST
5	RTF_LLINFO	0	0	0	0	0

图19-36 调用sysctl获取路由表和接口列表的程序举例

前三个程序从路由表中提取路由表项，后三个从接口列表中提取数据。route程序仅支持Internet选路协议，所以它指定mib[3]的值为AF_INET，而gated还支持其他协议，所以它对应的mib[3]的值为0。

图19-37画出了三个sysctl_*xxx*函数的结构，在后面的几节中会逐个予以阐述。

图19-38给出了sysctl_rtable函数。

1. 验证参数

705-719 当进程调用sysctl来设置一个路由表中不支持的变量时，使用new参数。因此该参数必须是一个空指针。

720-721 namelen必须是3，因为系统调用处理到这儿时，name数组中有三个元素：name[0]，地址族(进程中它被指定为mib[3])；name[1]，操作(mib[4])；以及name[2]，标志(mib[5])。

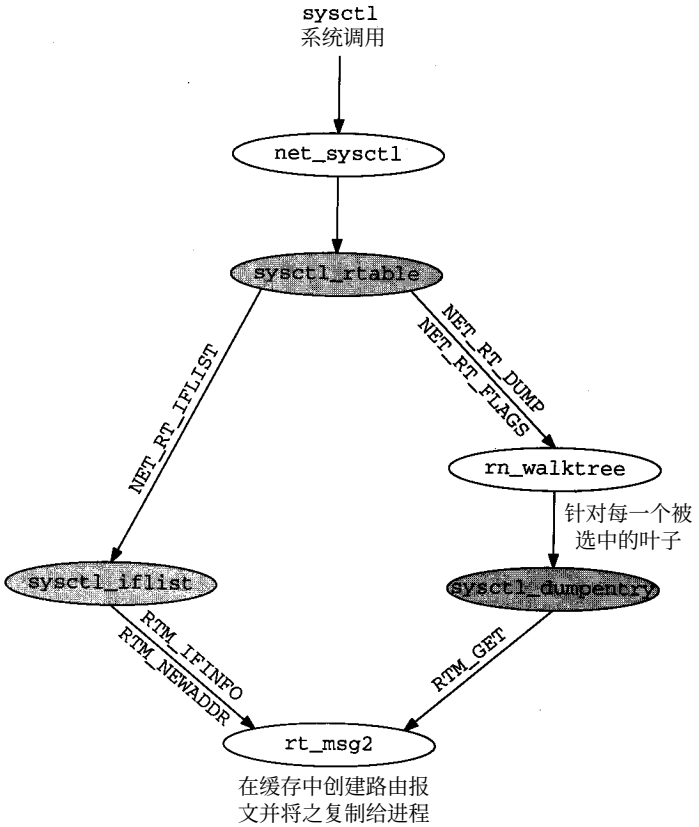


图19-37 支持针对选路插口的sysctl 系统调用的函数

rtsock.c

```
705 int
706 sysctl_rtable(name, namelen, where, given, new, newlen)
707 int    *name;
708 int     namelen;
709 caddr_t where;
710 size_t *given;
711 caddr_t *new;
712 size_t  newlen;
713 {
714     struct radix_node_head *rnh;
715     int     i, s, error = EINVAL;
716     u_char  af;
717     struct walkarg w;

718     if (new)
719         return (EPERM);
720     if (namelen != 3)
721         return (EINVAL);
722     af = name[0];
723     Bzero(&w, sizeof(w));
```

图19-38 sysctl_rtable 函数：处理sysctl 系统调用请求

```
724     w.w_where = where;
725     w.w_given = *given;
726     w.w_needed = 0 - w.w_given;
727     w.w_op = name[1];
728     w.w_arg = name[2];

729     s = splnet();
730     switch (w.w_op) {

731     case NET_RT_DUMP:
732     case NET_RT_FLAGS:
733         for (i = 1; i <= AF_MAX; i++)
734             if ((rnh = rt_tables[i]) && (af == 0 || af == i) &&
735                 (error = rnh->rnh_walktree(rnh,
736                                            sysctl_dumpentry, &w)))
737                 break;
738         break;

739     case NET_RT_IFLIST:
740         error = sysctl_iflist(af, &w);
741     }
742     splx(s);
743     if (w.w_tmem)
744         free(w.w_tmem, M_RTABLE);
745     w.w_needed += w.w_given;
746     if (where) {
747         *given = w.w_where - where;
748         if (*given < w.w_needed)
749             return (ENOMEM);
750     } else {
751         *given = (11 * w.w_needed) / 10;
752     }
753     return (error);
754 }
```

rtsock.c

图19-38 (续)

2. 初始化walkarg结构

723-728 把walkarg结构(图19-31)设置成0，并初始化下列成员：把w_where设置成调用进程中为结果准备的缓存地址；w_given是该缓存的比特数目(当w_where为空指针时，作为输入参数，该成员没有实际含义，但在输出时它必须被设置成将要返回的结果的长度)；w_needed被设置成缓存的大小的负数；w_op指明操作类型(值为NET_RT_*xxx*)；w_arg被设置成标志值。

3. 导出路由表

731-738 NET_RT_DUMP和NET_RT_FLAGS操作的处理是相同的：利用一个循环语句遍历所有的路由表(rt_table数组)，如果系统使用了该路由表，并且地址族调用参数为0或地址族调用参数与该路由表的族相匹配，则调用rnh_walktree函数来处理整个路由表。图18-17所给出的rnh_walktree函数是通常使用的rn_walktree函数。该函数的第二个参数的值是另一个函数的地址，这个函数(sysctl_dumpentry)将被调用以处理选路树的每一个叶子。rn_walktree的第三个参数是个任意类型的指针，该指针将传递给sysctl_dumpentry函数。在这里，它指向一个walkarg结构，该结构包含了有关sysctl调用的所有信息。

4. 返回接口列表

739-740 NET_RT_IFLIST操作调用sysctl_iflist函数来处理所有的ifnet结构。

5. 释放缓存

743-744 如果由rt_msg2分配的缓存里含有选路消息，则释放该缓存。

6. 更新w_needed

745 rt_msg2函数把每个消息的长度都加入到 w_needed中。而该变量被我们初始化成w_given的负数，所以它的值可以表示成：

```
w_needed = 0 - w_given + totalbytes
```

式中，totalbytes表示由rt_msg2函数添加的所有消息的长度总和。通过往 w_needed中加入w_given的值，我们就能得到所有消息的字节总数：

```
w_needed = 0 - w_given + totalbytes + w_given
         = totalbytes
```

因为等式中两个w_given的值最终相互抵消，所以当进程所指定的 w_where是个空指针时，就没有必要初始化w_given的值。事实上，图19-35中的变量needed就没有被初始化。

7. 返回报文的实际长度

746-749 如果where指针非空，则通过given指针返回保存在缓存中的字节数。如果返回的数值小于进程指定的缓存的大小，则返回一个差错，因为返回信息被截短了。

8. 返回报文长度的估算值

750-752 当where指针为空时，进程只想获得要返回的字节总数。为了防止在两次sysctl调用之间相应的表被增大，我们将该字节总数扩大了 10%。10%这个增量的确定没有特定的理由。

19.15 sysctl_dumpentry函数

在前一节中我们阐述了被sysctl_rtable调用的rn_walktree是如何调用本函数的。图19-39给出了本函数的代码。

623-630 每次调用本函数时，第一个参数指向一个 radix_node结构，同时它也是一个rtentry结构的指针，第二个参数指向一个由 sysctl_rtable初始化了的walkarg结构。

1. 检测路由表项的标志

631-632 如果进程指定了标志值(mib[5])，则忽略那些rt_flags成员中没有设置该标志的表项。在图19-36中，我们可以看到arp程序使用这种方法来选择那些设有RTF_LLINFO标志的表项，因为ARP仅对这些表项感兴趣。

2. 构造选路消息

633-638 rti_info数组中的下列四个指针是从路由表项中复制而得的： dst、gate、netmask和genmask。前两个总是非空的，但另外两个可以是空指针。调用 rt_msg2是为了构造一个RTM_GET消息。

3. 复制消息回传给进程

639-651 如果进程需要返回一个报文，并且rt_msg2分配了一个缓存，则将选路报文中的其余部分填写到w_tmem所指向的缓存中去，并调用 copyout复制消息回传给进程。如果复制成功，就增大w_where，增加的数目等于所复制的字节的数目。

rtsock.c

```
623 int
624 sysctl_dumpentry(rn, w)
625 struct radix_node *rn;
626 struct walkarg *w;
627 {
628     struct rtentry *rt = (struct rtentry *) rn;
629     int     error = 0, size;
630     struct rt_addrinfo info;

631     if (w->w_op == NET_RT_FLAGS && !(rt->rt_flags & w->w_arg))
632         return 0;
633     bzero((caddr_t) & info, sizeof(info));
634     dst = rt_key(rt);
635     gate = rt->rt_gateway;
636     netmask = rt_mask(rt);
637     genmask = rt->rt_genmask;
638     size = rt_msg2(RTM_GET, &info, 0, w);
639     if (w->w_where && w->w_tmem) {
640         struct rt_msghdr *rtm = (struct rt_msghdr *) w->w_tmem;

641         rtm->rtm_flags = rt->rt_flags;
642         rtm->rtm_use = rt->rt_use;
643         rtm->rtm_rmx = rt->rt_rmx;
644         rtm->rtm_index = rt->rt_ifp->if_index;
645         rtm->rtm_errno = rtm->rtm_pid = rtm->rtm_seq = 0;
646         rtm->rtm_addrs = info.rti_addrs;
647         if (error = copyout((caddr_t) rtm, w->w_where, size))
648             w->w_where = NULL;
649         else
650             w->w_where += size;
651     }
652     return (error);
653 }
```

rtsock.c

图19-39 sysctl_dumpentry 函数：处理一个路由表项

19.16 sysctl_iflist函数

图19-40给出了本函数的代码。本函数由sysctl_rtable直接调用，用来把接口列表返回给进程。

本函数由一个for循环组成，该循环从指针ifnet开始，针对每个接口重复执行。接着用while循环处理每个接口的ifaddr结构链表。函数将针对每个接口产生一个RTM_IFINFO选路消息，并且针对每个地址产生一个RTM_NEWADDR消息。

1. 检测接口索引

654-666 进程可以指定一个非零的标志参数（图19-36中的mib[5]）。函数用接口的if_index值与之比较，只有匹配时，才进行处理。

2. 创建选路消息

667-670 在RTM_IFINFO消息中只返回了唯一的一个插口地址结构，即ifpaddr。该RTM_IFINFO消息是由rt_msg2创建的。info结构中的ifpaddr指针被设置成0，因为该info结构还要用来产生后面的RTM_NEWADDR消息。

3. 复制消息回传给进程

671-681 如果进程需要返回消息，则填入if_msghdr结构的其余部分，用copyout给进

程复制该缓存，并增大w_where。

rtsock.c

```
654 int
655 sysctl_iflist(af, w)
656 int     af;
657 struct walkarg *w;
658 {
659     struct ifnet *ifp;
660     struct ifaddr *ifa;
661     struct rt_addrinfo info;
662     int     len, error = 0;
663     bzero((caddr_t) & info, sizeof(info));
664     for (ifp = ifnet; ifp; ifp = ifp->if_next) {
665         if (w->w_arg && w->w_arg != ifp->if_index)
666             continue;
667         ifa = ifp->if_addrlist;
668         ifpaddr = ifa->ifa_addr;
669         len = rt_msg2(RTM_IFINFO, &info, (caddr_t) 0, w);
670         ifpaddr = 0;
671         if (w->w_where && w->w_tmem) {
672             struct if_msghdr *ifm;
673             ifm = (struct if_msghdr *) w->w_tmem;
674             ifm->ifm_index = ifp->if_index;
675             ifm->ifm_flags = ifp->if_flags;
676             ifm->ifm_data = ifp->if_data;
677             ifm->ifm_addrs = info.rti_addrs;
678             if (error = copyout((caddr_t) ifm, w->w_where, len))
679                 return (error);
680             w->w_where += len;
681         }
682         while (ifa = ifa->ifa_next) {
683             if (af && af != ifa->ifa_addr->sa_family)
684                 continue;
685             ifaaddr = ifa->ifa_addr;
686             netmask = ifa->ifa_netmask;
687             brdaddr = ifa->ifa_dstaddr;
688             len = rt_msg2(RTM_NEWADDR, &info, 0, w);
689             if (w->w_where && w->w_tmem) {
690                 struct ifa_msghdr *ifam;
691                 ifam = (struct ifa_msghdr *) w->w_tmem;
692                 ifam->ifam_index = ifa->ifa_ifp->if_index;
693                 ifam->ifam_flags = ifa->ifa_flags;
694                 ifam->ifam_metric = ifa->ifa_metric;
695                 ifam->ifam_addrs = info.rti_addrs;
696                 if (error = copyout(w->w_tmem, w->w_where, len))
697                     return (error);
698                 w->w_where += len;
699             }
700         }
701         ifaaddr = netmask = brdaddr = 0;
702     }
703     return (0);
704 }
```

rtsock.c

图19-40 sysctl_iflist 函数：返回接口列表及其地址

4. 循环处理每一个地址结构，检测其地址族

682-684 处理接口的每一个ifaddr结构。进程也可以指定一个非零地址族 (图19-36中的

mib[3])来选择仅处理那些指定族的接口地址。

5. 创建选路消息

685-688 rt_msg2创建的每个RTM_NEWADDR消息中最多可以返回三个插口地址结构：ifaddr、netmask和brdaddr。

6. 复制消息回传给进程

689-699 如果进程需要返回消息，则填入ifa_msghdr结构的其余部分，用copyout给进程复制缓存，并增大w_where。

701 因为info数组还要在下一个接口消息中使用，所以程序将其中的这三个指针设置成0。

19.17 小结

所有选路消息的格式都是相同的——一个定长的结构，后面跟着若干个插口地址结构。共有三种不同类型的消息，各自具有相应的定长结构，每种定长结构的前三个元素都分别标识消息的长度、版本和类型。每种结构中的比特掩码指定哪些插口地址结构跟在定长结构之后。

这些消息以两种方式在内核与进程之间传递。消息可以在任意一个方向上传递，并且都是通过选路插口每次读或写一个消息。这就使得一个超级用户进程对内核路由表的访问具有完全的读写能力。选路守护进程(如routed和gated)就是这样实现其期望的选路策略的。

另外，任何一个进程都可以用sysctl系统调用来读取内核路由表的内容。这种方法不需要涉及选路插口，也不需要特别的权限。最终的结果通常包含许多选路消息，该结果作为系统调用的一部分被返回。由于进程不知道结果的大小，因此，为系统调用提供了一种方法来返回结果的大小而不返回结果本身。

习题

19.1 RTF_DYNAMIC和RTF_MODIFIED标志之间有什么区别？对于一个给定的路由表项，它们可以同时设置吗？

19.2 当用下列命令添加默认路由时会有什么情况发生？

```
bsdi $ route add default -cloning -genmask 255.255.255.255 sun
```

19.3 某路由表包含了15个ARP表项和20个路由，试估算用sysctl导出该路由表时需要多少空间。

第20章　选 路 插 口

20.1　引言

一个进程使用选路域(routing domain)中的一个插口来发送和接收前一章所描述的选路报文。socket系统调用需要指定一个PF_ROUTE的族类型和一个SOCK_RAW的插口类型。

接着，该进程可以向内核发送以下五种选路报文：

1) RTM_ADD：增加一条新路由。

2) RTM_DELETE：删除一条已经存在的路由。

3) RTM_GET：取得有关一条路由的所有信息。

4) RTM_CHANGE：改变一条已经存在路由的网关、接口或者度量。

5) RTM_LOCK：说明内核不应该修改哪个度量。

除此之外，该进程可以接收其他七个选路报文，这些报文是在发生某些事件时，如接口下线和收到重定向报文等等，由内核生成的。

本章简介选路域、为每个选路插口创建的选路控制块、处理进程产生的报文的函数(route_output)、发送选路报文给一个或多个进程的函数(raw_input)、以及不同的支持一个选路插口上所有插口操作的函数。

20.2　routedomain和protosw结构

在描述选路插口函数之前，我们需要讨论有关选路域的其他一些细节；在选路域中支持的SOCK_RAW协议；以及每个选路插口所附带的选路控制块。

图20-1列出了称为routedomain的PF_ROUTE域的domain结构。

成　员	值	描　述
dom_family	*PF_ROUTE*	域的协议族
dom_name	*route*	名字
dom_init	*route_init*	域的初始化，图18-30
dom_externalize	*0*	在选路域中不使用
dom_dispose	*0*	在选路域中不使用
dom_protosw	*routesw*	协议交换结构，图20-2
dom_protoswNPROTOSW		指向协议交换结构之后的指针
dom_next		由domaininit填入，图7-15
dom_rtattch	*0*	在选路域中不使用
dom_rtoffset	*0*	在选路域中不使用
dom_maxrtkey	*0*	在选路域中不使用

图20-1　routedomain　结构

与支持多个协议（TCP、UDP和ICMP等)的Internet域不一样，在选路域中只支持SOCK_RAW类型的一种协议。图20-2列出了PF_ROUTE域的协议转换项。

成 员	routesw[0]	描 述
pr_type	*SOCK_RAW*	原始插口
pr_domain	*&routedomain*	选路域部分
pr_protocol	*0*	
pr_flags	*PR_ATOMI|PR_ADDR*	插口层标志，协议处理时不使用
pr_input	*raw_input*	不使用这项，raw_input直接调用
pr_output	*route_output*	PRU_SEND请求所调用
pr_ctlinput	*raw_ctlinput*	控制输入函数
pr_ctloutput	*0*	不使用
pr_usrreq	*route_usrreq*	对一个进程通信请求的响应
pr_init	*raw_init*	初始化
pr_fasttimo	*0*	不使用
pr_slowtimo	*0*	不使用
pr_drain	*0*	不使用
pr_sysctl	*sysctl_rtable*	用于sysctl(8)系统调用

图20-2 选路协议protosw 的结构

20.3 选路控制块

每当采用如下形式的调用创建一个选路插口时，

```
socket(PF_ROUTE, SOCK_RAW, protocol);
```

对协议的用户请求函数(route_usrreq)的一个对应的PRU_ATTACH请求分配一个选路控制块，并且将它链接到插口结构上。 *protocol*参数可以将发送给这个插口上的进程的报文类型限制为一个特定族。例如，如果将 *protocol*参数说明为AF_INET，只有包含了Internet地址的选路报文将被发送给这个进程。 *protocol*参数为0将使得来自内核的所有选路报文都发送给这个插口。

记住我们把这些结构称为选路控制块，而不是原始控制块 (raw control block)，是为了避免与第32章中的原始IP控制块相混淆。

图20-3显示了rawcb结构的定义。

raw_cb.h

```
39 struct rawcb {
40     struct rawcb *rcb_next;     /* doubly linked list */
41     struct rawcb *rcb_prev;
42     struct socket *rcb_socket;  /* back pointer to socket */
43     struct sockaddr *rcb_faddr; /* destination address */
44     struct sockaddr *rcb_laddr; /* socket's address */
45     struct sockproto rcb_proto; /* protocol family, protocol */
46 };

47 #define sotorawcb(so)       ((struct rawcb *)(so)->so_pcb)
```

raw_cb.h

图20-3 rawcb 结构

另外，分配了一个相同名字的全局结构， rawcb，作为这个双向链表的头。图 20-4显示了这种情况。

39-47 我们在图19-26中显示了sockproto的结构。它的sp_family成员变量被设置为PF_ROUTE，sp_protocol成员变量被设置为 socket系统调用的第三个参数。

rcb_faddr成员变量被永久性地设置为指向 route_src的指针，我们在图 19-26中描述了route_src。rcb_laddr总是一个空指针。

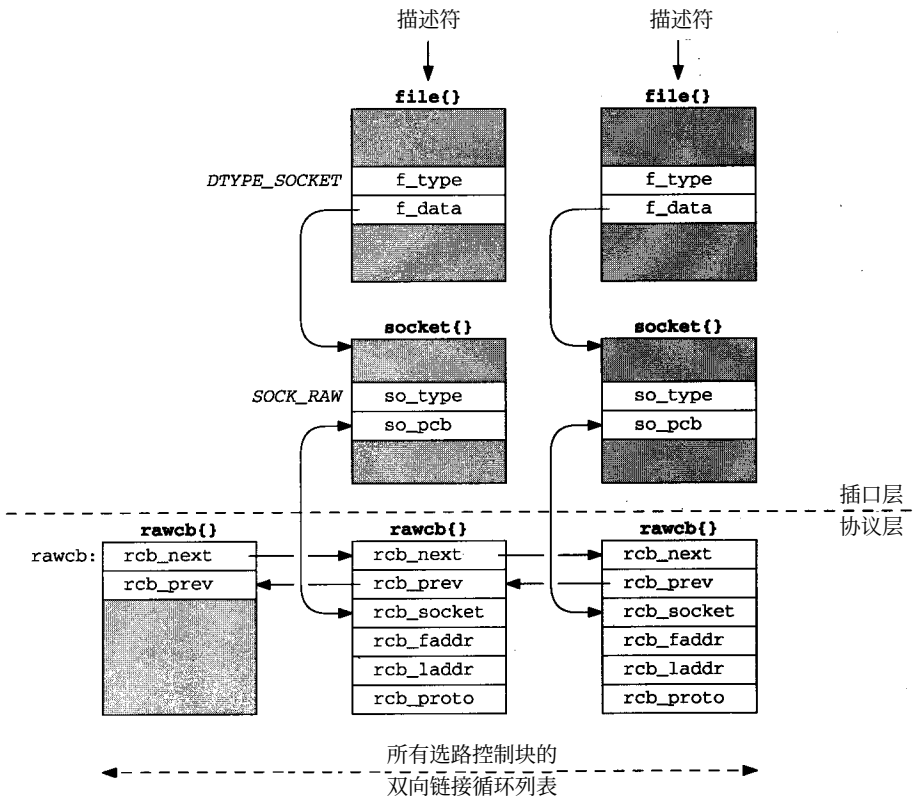


图20-4 原始协议控制块与其他数据结构的关系

20.4 raw_init函数

在图20-5中显示的raw_init函数是图20-2中的protosw结构的协议初始化函数。我们在图18-29中描述了选路域的完整初始化过程。

38-42 这个函数将头结构的下一个和前一个指针设置为指向自身来对这个双向链表进行初始化。

raw_usrreq.c

```
38 void
39 raw_init()
40 {
41     rawcb.rcb_next = rawcb.rcb_prev = &rawcb;
42 }
```

raw_usrreq.c

图20-5 raw_init 函数：初始化选路控制块的双向链表

20.5 route_output函数

如同我们在图18-11所显示的，当给协议的用户请求函数发送 PRU_SEND请求时，就会调

用route_output，这是一个进程向一个选路插口进行写操作所引起的。在图 18-9中，我们给出了内核接受的、由进程发出的五种不同类型的选路报文。

因为这个函数是由一个进程的写操作激活的，来自于该进程的数据 (发送给进程的选路报文)被放在一个由sosend开始的mbuf链中。图20-6显示了大概的处理步骤，假定进程发送了一个RTM_ADD命令，说明三个地址：目的地址、它的网关和一个网络掩码 (因此，这是一个网络路由，而不是一个主机路由)。

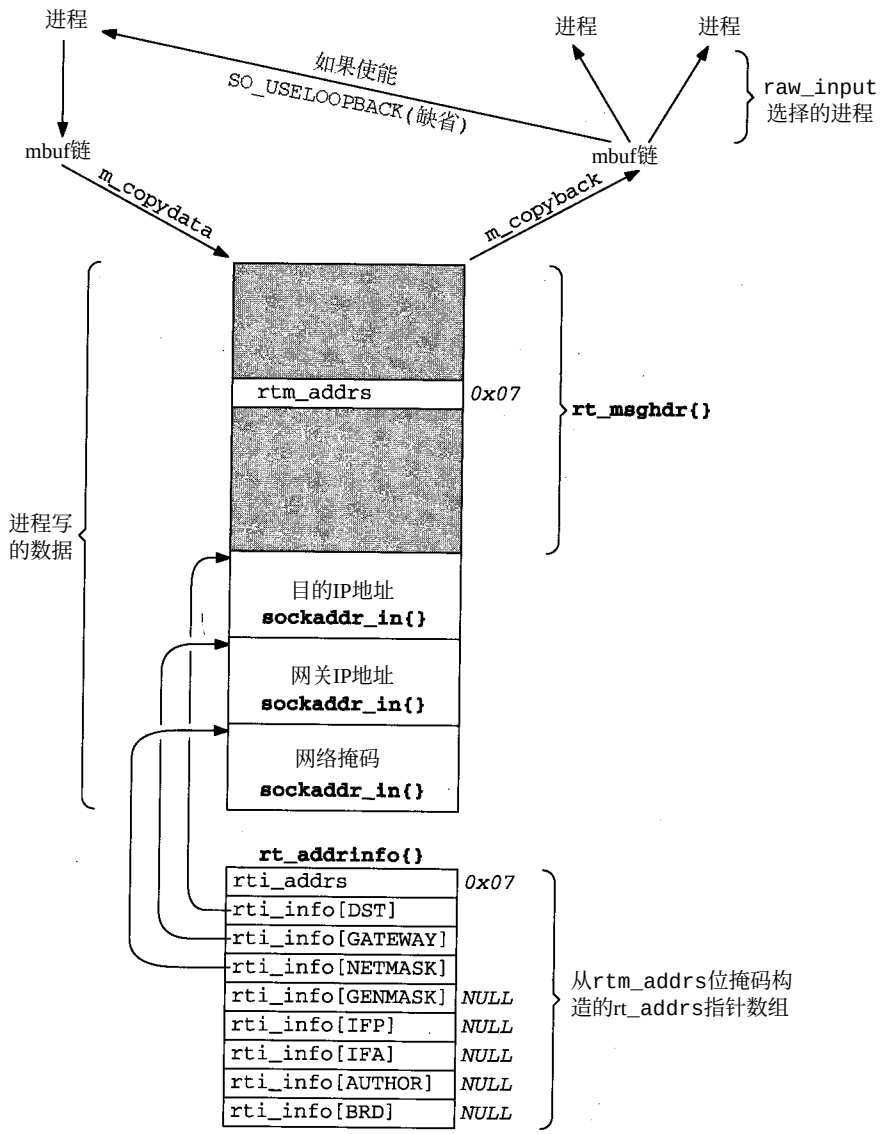


图20-6 一个进程发出的RTM_ADD 命令的处理过程示例

在这个图中有几点需要引起注意，我们在介绍 route_output的源代码时讨论了这里需要注意的大多数情况。另外，为了节省空间，我们省略了 rt_addrinfo结构中每个数组下标的RTAX_前缀。

• 进程通过设置比特掩码rtm_addrs来说明在定长的rt_msghdr结构之后有哪些插口地

址结构。我们显示了一个值为0x07的比特掩码，表示有一个目的地址、一个网关地址和一个网络掩码(图19-19)。RTM_ADD命令需要前两个地址；第三个地址是可选的。另一个可选的地址，genmask说明了用来产生克隆路由的掩码。

- write系统调用(sosend函数)将来自进程的一个缓存数据复制到内核的一个mbuf链中。
- m_copydata将mbuf链中数据复制到route_output使用malloc获得的一个缓存中。访问存储在单个连续缓存中的结构以及结构后面的若干插口地址结构，比访问一个mbuf链更容易。
- route_output调用rt_xaddrs函数取得比特掩码，并且构造一个指向缓存的rt_addrinfo结构。route_output中的代码使用图19-19中第五栏显示的名字来引用这些结构。比特掩码也要复制到rti_addrs成员中。
- route_output一般要修改rt_msghdr结构。如果发生了一个错误，相应的errno值被返回到rtm_errno中(例如，如果路由已经存在，则返回EEXIST)；否则，RTF_DONE标志与进程提供的rtm_flags执行逻辑或操作。
- rt_msghdr结构以及接着的地址成为0个或多个正在读选路插口的进程的输入。首先使用m_copyback将缓存转换为一个mbuf链。raw_input经过所有的选路PCB，并且传递一个复制给对应的进程。我们还显示了一个带有选路插口的进程，如果该进程没有禁止SO_USELOOPBACK插口选项，就会收到它写给那个插口的每个报文的一个复制。

为了避免收到它们自己的选路报文的一个复制，有些程序，如route，将第二个参数置为0来调用shutdown，以防止从选路插口上收到任何数据。

我们分成七个部分分析route_output的源代码。图20-7显示了这个函数的大概流程。

```
int
route_output()
{
    R_Malloc() to allocate buffer;
    m_copydata() to copy from mbuf chain into buffer;
    rt_xaddrs() to build rt_addrinfo{};

    switch (message type) {
    case RTM_ADD:
        rtrequest(RTM_ADD);
        rt_setmetrics();
        break;
    case RTM_DELETE:
        rtrequest(RTM_DELETE);
        break;

    case RTM_GET:
    case RTM_CHANGE:
    case RTM_LOCK:
        rtalloc1();

        switch (message type) {
        case RTM_GET:
            rt_msg2(RTM_GET);
            break;

        case RTM_CHANGE:
```

图20-7 route_output 处理步骤小结

```
            change appropriate fields;
            /* fall through */

        case RTM_LOCK:
            set rmx_locks;
            break;
        }
        break;
    }

    set rtm_error if error, else set RTF_DONE flag;

    m_copyback() to copy from buffer into mbuf chain;

    raw_input();    /* mbuf chain to appropriate processes */
}
```

图20-7 (续)

route_output的第一部分显示在图20-8中。

rtsock.c

```
113 int
114 route_output(m, so)
115 struct mbuf *m;
116 struct socket *so;
117 {
118     struct rt_msghdr *rtm = 0;
119     struct rtentry *rt = 0;
120     struct rtentry *saved_nrt = 0;
121     struct rt_addrinfo info;
122     int     len, error = 0;
123     struct ifnet *ifp = 0;
124     struct ifaddr *ifa = 0;

125 #define senderr(e) { error = e; goto flush;}
126     if (m == 0 || ((m->m_len < sizeof(long)) &&
127                    (m = m_pullup(m, sizeof(long))) == 0))
128                 return (ENOBUFS);
129     if ((m->m_flags & M_PKTHDR) == 0)
130         panic("route_output");
131     len = m->m_pkthdr.len;
132     if (len < sizeof(*rtm) ||
133         len != mtod(m, struct rt_msghdr *)->rtm_msglen) {
134         dst = 0;
135         senderr(EINVAL);
136     }
137     R_Malloc(rtm, struct rt_msghdr *, len);
138     if (rtm == 0) {
139         dst = 0;
140         senderr(ENOBUFS);
141     }
142     m_copydata(m, 0, len, (caddr_t) rtm);
143     if (rtm->rtm_version != RTM_VERSION) {
144         dst = 0;
145         senderr(EPROTONOSUPPORT);
146     }
147     rtm->rtm_pid = curproc->p_pid;

148     info.rti_addrs = rtm->rtm_addrs;
149     rt_xaddrs((caddr_t) (rtm + 1), len + (caddr_t) rtm, &info);
```

图20-8 route_output 函数：初始化处理，从mbuf链中复制报文

```
150     if (dst == 0)
151         senderr(EINVAL);
152     if (genmask) {
153         struct radix_node *t;
154         t = rn_addmask((caddr_t) genmask, 1, 2);
155         if (t && Bcmp(genmask, t->rn_key, *(u_char *) genmask) == 0)
156             genmask = (struct sockaddr *) (t->rn_key);
157         else
158             senderr(ENOBUFS);
159     }
```

rtsock.c

图20-8 (续)

1. 检查mbuf的合法性

113-136 检查mbuf的合法性：它的长度必须至少是一个rt_msghdr结构的大小。从mbuf的数据部分取出第一个长字，里面包含了rtm_msglen的值。

2. 分配缓存

137-142 分配一个缓存来存放整个报文，m_copydata将报文从mbuf链复制到缓存。

3. 检查版本号

143-146 检查报文的版本号。如果将来引入了新版本的选路报文，这个成员变量可以用来对早期版本提供支持。

147-149 进程的ID被复制到rtm_pid，进程提供的比特掩码被复制到该函数的一个内部结构info.rti_addrs。函数rt_xaddrs(在下一节显示)填入info结构的第8个插口地址指针来指示当前包含报文的缓存。

4. 需要的目的地址

150-151 所有的命令都需要一个目的地址。如果info.rti_info[RTAX_DST]项是一个空指针，就需要一个EINVAL。记住dst引用了这个数组成员(图19-19)。

5. 处理可选的genmask

152-159 genmask是可选的，它是在设置了RTF_CLONING标志后(图19-8)，用作所创建路由的网络掩码。rn_addmask将这个掩码加入到掩码树中，并首先在掩码树中查找是否存在与这个掩码相同的条目，如果找到，就引用那个条目。如果在掩码树中找到或者将这个掩码加入到掩码树中，还要再检查一下掩码树中的那个条目是否真等于genmask的值，如果等于，则genmask指针就被替代为掩码树中那个掩码的指针。

图20-9显示了route_output函数处理RTM_ADD和RTM_DELETE的下一部分。

162-163 RTM_ADD命令要求进程说明一个网关。

164-165 rtrequest处理该请求。如果输入的路由是一个主机路由，则netmask指针可以为空。如果一切OK，则saved_nrt返回新的路由表项的指针。

166-172 将rt_metrics结构从调用者缓存中复制到路由表项中。引用计数减1，并且保存genmask指针(可能是一个空指针)。

173-176 处理RTM_DELETE命令非常简单，因为所有的工作都由rtrequest来完成。既然最后一个参数是一个空指针，如果引用计数为0，rtrequest就调用rtfree从路由表中删除指定的项(图19-7)。

下一步的处理过程显示在图20-10中，它显示了RTM_GET、RTM_CHANGE和RTM_LOCK命

令的公共代码。

rtsock.c

```
160     switch (rtm->rtm_type) {

161     case RTM_ADD:
162         if (gate == 0)
163             senderr(EINVAL);
164         error = rtrequest(RTM_ADD, dst, gate, netmask,
165                           rtm->rtm_flags, &saved_nrt);
166         if (error == 0 && saved_nrt) {
167             rt_setmetrics(rtm->rtm_inits,
168                           &rtm->rtm_rmx, &saved_nrt->rt_rmx);
169             saved_nrt->rt_refcnt--;
170             saved_nrt->rt_genmask = genmask;
171         }
172         break;

173     case RTM_DELETE:
174         error = rtrequest(RTM_DELETE, dst, gate, netmask,
175                           rtm->rtm_flags, (struct rtentry **) 0);
176         break;
```

rtsock.c

图20-9 route_output 函数：进程RTM_ADD 和RTM_DELETE 命令

rtsock.c

```
177     case RTM_GET:
178     case RTM_CHANGE:
179     case RTM_LOCK:
180         rt = rtalloc1(dst, 0);
181         if (rt == 0)
182             senderr(ESRCH);
183         if (rtm->rtm_type != RTM_GET) {     /* XXX: too grotty */
184             struct radix_node *rn;
185             extern struct radix_node_head *mask_rnhead;

186             if (Bcmp(dst, rt_key(rt), dst->sa_len) != 0)
187                 senderr(ESRCH);
188             if (netmask && (rn = rn_search(netmask,
189                                            mask_rnhead->rnh_treetop)))
190                 netmask = (struct sockaddr *) rn->rn_key;
191             for (rn = rt->rt_nodes; rn; rn = rn->rn_dupedkey)
192                 if (netmask == (struct sockaddr *) rn->rn_mask)
193                     break;
194             if (rn == 0)
195                 senderr(ETOOMANYREFS);
196             rt = (struct rtentry *) rn;
197         }
```

rtsock.c

图20-10 route_output 函数：RTM_GET 、RTM_CHANGE 和RTM_LOCK 的公共处理部分

6. 查找已经存在的项

177-182 因为三个命令都引用了一个已经存在的项，所以用 rtalloc1函数来查找这个项。如果没有找到，则返回一个ESRCH。

7. 不允许网络匹配

183-187 对于RTM_CHANGE和RTM_LOCK命令，一个网络匹配是不合适的：需要一个路由表关键字的精确匹配。因此，如果 dst参数不等于路由表关键字，这个匹配就是一个网络匹配，返回一个ESRCH。

8. 使用网络掩码来查找正确的项

188-193 即使是一个精确的匹配，如果存在网络掩码不同的重复表项，仍然必须查找正确的项。如果提供了一个netmask参数，就要在掩码表中查找它(mask_rnhead)。如果找到了，netmask指针就被替代为掩码树中相应掩码的指针。检查重复表项列表的每个叶结点，查找一个rn_mask指针等于netmask的项。这个测试只是比较指针，而不是指针所指向的结构。这是因为所有的掩码都出现在掩码树中，并且每个不同的掩码只有一个副本出现在这个掩码树中。大多数情况下，表项不会重复，因此 for循环只执行一次。如果一个主机路由项被修改了，不应该提供一个网络掩码，因此， netmask和rn_mask都是空指针(两者是相等的)。但是，如果有一个附带掩码的项被修改了，那个掩码必须作为 netmask参数提供。

194-195 如果for循环终止时没有找到一个匹配的网络掩码，就返回 ETOOMANYREFS。

注释XXX表示这个函数必须做所有的工作来找到需要的项。所有这些细节在其他一些类似rtalloc1的函数中都应该被隐藏，rtalloc1检测网络匹配，并且处理掩码参数。

这个函数的下一部分继续处理 RTM_GET命令，显示在图 20-11中。这个命令与

rtsock.c

```
198         switch (rtm->rtm_type) {

199         case RTM_GET:
200             dst = rt_key(rt);
201             gate = rt->rt_gateway;
202             netmask = rt_mask(rt);
203             genmask = rt->rt_genmask;
204             if (rtm->rtm_addrs & (RTA_IFP | RTA_IFA)) {
205                 if (ifp = rt->rt_ifp) {
206                     ifpaddr = ifp->if_addrlist->ifa_addr;
207                     ifaaddr = rt->rt_ifa->ifa_addr;
208                     rtm->rtm_index = ifp->if_index;
209                 } else {
210                     ifpaddr = 0;
211                     ifaaddr = 0;
212                 }
213             }
214             len = rt_msg2(RTM_GET, &info, (caddr_t) 0,
215                           (struct walkarg *) 0);
216             if (len > rtm->rtm_msglen) {
217                 struct rt_msghdr *new_rtm;
218                 R_Malloc(new_rtm, struct rt_msghdr *, len);
219                 if (new_rtm == 0)
220                     senderr(ENOBUFS);
221                 Bcopy(rtm, new_rtm, rtm->rtm_msglen);
222                 Free(rtm);
223                 rtm = new_rtm;
224             }
225             (void) rt_msg2(RTM_GET, &info, (caddr_t) rtm,
226                           (struct walkarg *) 0);
227             rtm->rtm_flags = rt->rt_flags;
228             rtm->rtm_rmx = rt->rt_rmx;
229             rtm->rtm_addrs = info.rti_addrs;
230             break;
```

rtsock.c

图20-11 route_output 函数：RTM_GET 处理过程

route_output支持的其他命令的区别在于它能够返回比传递给它的更多的数据。例如，只需要输入一个插口地址结构，即目的地址，但至少返回两个插口地址结构，即目的地址和它的网关。参看图20-6，这就意味着为m_copydata复制数据所分配的缓存可能需要扩充大小。

9. 返回目的地址、网关和掩码

198-203 rti_info数组中存储了四个指针：dst、gate、netmask和genmask。后两个可能是空指针。info结构中的这些指针指向进程将要返回的各个插口地址结构。

10. 返回接口信息

204-213 进程可以在rtm_flags比特掩码中设置RTA_IFP和RTA_IFA掩码。如果设置了一项或两项，就表示进程想要接收这个路由表项所指示的 ifaddr结构：接口的链路层地址(由rt_ifp->addrlist指向)以及这个路由项的协议地址(由rt_ifa->ifa_addr指向)的内容。接口索引也会被返回。

11. 构造应答报文

214-224 将第三个指针置为空，调用rt_msg2来计算相应于RTM_GET的选路报文和info结构所指向的地址的长度。如果结果报文的长度超过了输入报文的长度，就会分配一个新的缓存，输入报文被复制到新的缓存中，老的缓存被释放， rtm被重新设置为指向新缓存。

225-230 再次调用rt_msg2，这次调用时第三个指针非空，因为在缓存中已经构造了一个结果报文。这次调用填入rt_msghdr结构的最后三个成员项。

图20-12显示了RTM_CHANGE和RTM_LOCK命令的处理过程。

12. 改变网关

231-233 如果进程传递了一个gate地址，rt_setgate就被调用来改变这个路由表项的网关。

13. 查找新的接口

234-244 新的网关(如果被改变)可能也需要rt_ifp和rt_ifa指针。进程可以通过传递一个ifpaddr插口地址结构或者一个ifaaddr插口地址结构来说明这些新的值。先看第一个，然后再看第二个。如果进程两个结构都没传递， rt_ifp和rt_ifa指针就被忽略。

14. 检验接口是否改变

245-256 如果找到了一个接口(ifa非空)，则该路由的现有rt_ifa指针要和新的值进行比较。如果数值已经改变了，则两个针对 rt_ifp和rt_ifa的新值需要存储到路由表的表项中去。在这样做之前，先要用 RTM_DELETE命令调用该接口的请求函数 (如果定义了该函数的话)。这个删除动作是必须的，因为从一种类型的网络到另一种类型的网络，它们的链路层信息可能会有很大的差别(比如说从一个X.25网络改变成以太网的路由)，同时我们还必须通知输出例程。

15. 更新度量

257-258 rt_setmetrics修改路由表项的度量。

16. 调用接口请求函数

259-260 如果定义了一个接口请求函数，它就会和 RTM_ADD命令一起被调用。

17. 保存克隆生成的掩码

261-262 如果进程指定了 genmask参数，就将在图 20-8中获得的掩码的指针保存在rt_genmask中。

18. 修改加锁度量的比特掩码

266-270 RTM_LOCK命令修改保存在rt_rmx.rmx_locks中的比特掩码。图20-13显示了这个比特掩码中不同比特的值，每个度量一个值。

rtsock.c

```
231         case RTM_CHANGE:
232             if (gate && rt_setgate(rt, rt_key(rt), gate))
233                 senderr(EDQUOT);
234             /* new gateway could require new ifaddr, ifp; flags may also be
235                different; ifp may be specified by ll sockaddr when protocol
236                address is ambiguous */
237             if (ifpaddr && (ifa = ifa_ifwithnet(ifpaddr)) &&
238                 (ifp = ifa->ifa_ifp))
239                 ifa = ifaof_ifpforaddr(ifaaddr ? ifaaddr : gate,
240                                        ifp);
241             else if ((ifaaddr && (ifa = ifa_ifwithaddr(ifaaddr))) ||
242                      (ifa = ifa_ifwithroute(rt->rt_flags,
243                                             rt_key(rt), gate)))
244                 ifp = ifa->ifa_ifp;
245             if (ifa) {
246                 struct ifaddr *oifa = rt->rt_ifa;
247                 if (oifa != ifa) {
248                     if (oifa && oifa->ifa_rtrequest)
249                         oifa->ifa_rtrequest(RTM_DELETE,
250                                             rt, gate);
251                     IFAFREE(rt->rt_ifa);
252                     rt->rt_ifa = ifa;
253                     ifa->ifa_refcnt++;
254                     rt->rt_ifp = ifp;
255                 }
256             }
257             rt_setmetrics(rtm->rtm_inits, &rtm->rtm_rmx,
258                           &rt->rt_rmx);
259             if (rt->rt_ifa && rt->rt_ifa->ifa_rtrequest)
260                 rt->rt_ifa->ifa_rtrequest(RTM_ADD, rt, gate);
261             if (genmask)
262                 rt->rt_genmask = genmask;
263             /*
264              * Fall into
265              */
266         case RTM_LOCK:
267             rt->rt_rmx.rmx_locks &= ~(rtm->rtm_inits);
268             rt->rt_rmx.rmx_locks |=
269                 (rtm->rtm_inits & rtm->rtm_rmx.rmx_locks);
270             break;
271         }
272         break;
273     default:
274         senderr(EOPNOTSUPP);
275     }
```

rtsock.c

图20-12 route_output 函数：RTM_CHANGE 和RTM_LOCK 处理过程

路由表项中rt_metrics结构的rmx_locks成员是告诉内核哪些度量不要管的比特掩码。即，rmx_locks指定的那些度量内核不能修改。内核惟一能使用这些度量的地方是和TCP一起，如图27-3所示。rmx_pksent度量不能被初始化或加锁，但是内核也从来没有引用或修改过这个成员。

进程发出的报文中的 rtm_inits值是一个比特掩码，指出哪些度量刚刚被

常 量	值	描 述
RTV_MTU	0x01	初始化或者锁住rmx_mtu
RTV_HOPCOUNT	0x02	初始化或者锁住rmx_hopcount
RTV_EXPIRE	0x04	初始化或者锁住rmx_expire
RTV_RPIPE	0x08	初始化或者锁住rmx_recvpipe
RTV_SPIPE	0x10	初始化或者锁住rmx_sendpipe
RTV_SSTHRESH	0x20	初始化或者锁住rmx_ssthresh
RTV_RTT	0x40	初始化或者锁住rmx_rtt
RTV_RTTVAR	0x80	初始化或者锁住rmx_rttvar

图20-13 对度量初始化或加锁的常量

rt_setmetrics初始化过。报文中的rtm_rmx.rmx_locks值是一个指出哪些度量现在应该加锁的比特掩码。rt_rmx.rmx_locks的值是一个指出路由表中哪些度量当前被加锁的比特掩码。首先，任何将要初始化的比特(rtm_inits)都要解锁。任何既被初始化(rtm_inits)又被加锁(rtm_rmx.rmx_locks)的比特都必须加锁。

273-275 这个default是用于图20-9开始的switch语句，用来处理进程发出的报文中除了所支持的五个命令以外的其他选路命令。

route_output的最后一部分显示在图20-14中，用来发送应答给raw_input。

rtsock.c

```
276   flush:
277     if (rtm) {
278         if (error)
279             rtm->rtm_errno = error;
280         else
281             rtm->rtm_flags |= RTF_DONE;
282     }
283     if (rt)
284         rtfree(rt);
285     {
286         struct rawcb *rp = 0;
287         /*
288          * Check to see if we don't want our own messages.
289          */
290         if ((so->so_options & SO_USELOOPBACK) == 0) {
291             if (route_cb.any_count <= 1) {
292                 if (rtm)
293                     Free(rtm);
294                 m_freem(m);
295                 return (error);
296             }
297             /* There is another listener, so construct message */
298             rp = sotorawcb(so);
299         }
300         if (rtm) {
301             m_copyback(m, 0, rtm->rtm_msglen, (caddr_t) rtm);
302             Free(rtm);
303         }
304         if (rp)
305             rp->rcb_proto.sp_family = 0;     /* Avoid us */
306         if (dst)
307             route_proto.sp_protocol = dst->sa_family;
308         raw_input(m, &route_proto, &route_src, &route_dst);
```

图20-14 route_output 函数：将结果传递给raw_input

```
309         if (rp)
310             rp->rcb_proto.sp_family = PF_ROUTE;
311     }
312     return (error);
313 }
```
rtsock.c

图20-14 (续)

19. 返回错误或OK

276-282 flush是该函数开头定义的senderr宏所跳转的标号。如果发生了一个错误，错误就在rtm_errno成员中返回；否则，就设置RTF_DONE标志。

20. 释放拥有的路由

283-284 如果拥有一条路由，就要被释放。如果找到，在图 20-10的开始位置对rtalloc1的调用拥有这条路由。

21. 没有进程接收报文

285-296 SO_USELOOPBACK插口选项的默认值为真，表示发送进程将会收到它发送给选路插口的每个选路报文的一个复制(如果发送者不接收一个复制的报文，它就不能收到RTM_GET返回的任何信息)。如果没有设置这个选项，并且选路插口的总数小于或等于 1，就没有其他进程接收报文，并且发送者不想要一个复制报文。缓存和mbuf链都会被释放，该函数返回。

22. 没有环回复制报文的其他监听者

297-299 至少有一个其他的监听者而不是发送进程不想要一个复制报文。指针 rp，默认是空，被设置成指向发送者的选路控制块，它也用来作为发送者不想要复制报文的一个标志。

23. 将缓存转换成mbuf链

300-303 缓存被转换成一个mbuf链(图20-6)，然后释放缓存。

24. 避免环回复制

304-305 如果设置了rp，则某个其他的进程可能想要报文，但是发送者不想要一个复制。发送者的选路控制块的sp_family成员被临时设置为0，但是报文的sp_family(route_proto结构，显示在图19-26中)有一个PF_ROUTE的族。这个技巧防止raw_input将结果的一个复制传递给发送进程，因为raw_input不会将一个复制传递给sp_family为0的任何插口。

25. 设置选路报文的地址族

306-308 如果dst是一个非空的指针，则那个插口地址结构的地址族成为选路报文的协议。对于Internet协议，这个值将是PF_INET。通过raw_input，一个复制被传递给合适的监听者。

309-313 如果调用进程的sp_family成员被临时设置为0，它就被复位成正常值，PF_ROUTE。

20.6 rt_xaddrs函数

在将来自进程的选路报文从 mbuf链复制到一个缓存以及将来自进程的比特掩码(rtm_addrs)复制到rt_addrinfo结构的rti_info成员之后，只从route_output中调用一次rt_xaddrs函数(图20-8)。rt_xaddrs的目的是获取这个比特掩码，并且设置rti_info数组的指针，使之指向缓存中相应的地址。图 20-15显示了这个函数。

330-340 指针数组被设置成0，因此，所有在比特掩码中不出现的地址结构的指针都将是空。

341-347 测试比特掩码中8个(RTM_MAX)可能比特的每一个(如果设置)，将相应于插口地址结构的一个指针存到rti_info数组中。ADVANCE宏以插口地址结构的sa_len字段为参数，

上舍入为4个字节的倍数，相应地增加指针cp。

rtsock.c

```
330 #define ROUNDUP(a) \
331     ((a) > 0 ? (1 + (((a) - 1) | (sizeof(long) - 1))) : sizeof(long))
332 #define ADVANCE(x, n) (x += ROUNDUP((n)->sa_len))
333 static void
334 rt_xaddrs(cp, cplim, rtinfo)
335 caddr_t cp, cplim;
336 struct rt_addrinfo *rtinfo;
337 {
338     struct sockaddr *sa;
339     int     i;
340     bzero(rtinfo->rti_info, sizeof(rtinfo->rti_info));
341     for (i = 0; (i < RTAX_MAX) && (cp < cplim); i++) {
342         if ((rtinfo->rti_addrs & (1 << i)) == 0)
343             continue;
344         rtinfo->rti_info[i] = sa = (struct sockaddr *) cp;
345         ADVANCE(cp, sa);
346     }
347 }
```

rtsock.c

图20-15 rt_xaddrs 函数：将指针填入rti_info 数组

20.7 rt_setmetrics函数

这个函数在route_output中调用了两次：增加一条新路由时和改变一条已经存在的路由时。来自进程的选路报文的rtm_inits成员说明了进程想要初始化rtm_rmx数组中的哪些度量。比特掩码中的比特的值显示在图20-13中。

请注意，rtm_addrs和rtm_inits都是来自进程的报文中的比特掩码，前者说明了接下来的插口地址结构，而后者说明哪些度量将被初始化。为了节省空间，在rtm_addrs中没有设置比特的插口地址结构也不会出现在选路报文中。但是整个rt_metrics总是以定长的rt_msghdr结构的形式出现——在rtm_inits中没有设置比特的数组成员将被忽略。

图20-16显示了rt_setmetrics函数。

rtsock.c

```
314 void
315 rt_setmetrics(which, in, out)
316 u_long  which;
317 struct rt_metrics *in, *out;
318 {
319 #define metric(f, e) if (which & (f)) out->e = in->e;
320     metric(RTV_RPIPE, rmx_recvpipe);
321     metric(RTV_SPIPE, rmx_sendpipe);
322     metric(RTV_SSTHRESH, rmx_ssthresh);
323     metric(RTV_RTT, rmx_rtt);
324     metric(RTV_RTTVAR, rmx_rttvar);
325     metric(RTV_HOPCOUNT, rmx_hopcount);
326     metric(RTV_MTU, rmx_mtu);
327     metric(RTV_EXPIRE, rmx_expire);
328 #undef metric
329 }
```

rtsock.c

图20-16 rt_setmetrics 函数：设置rt_metrics 结构中的成员

314-318 which参数总是进程的选路报文的 rtm_inits成员。 in指向进程的 rt_metrics结构，而out指向将要创建或修改的路由表项的rt_metrics结构。

319-329 测试比特掩码中8比特的每一比特，如果该比特被设置，就复制相应的度量。请注意当使用RTM_ADD创建一个新的路由表项时， route_output调用了rtrequest，后者将整个路由表项设置为0(图19-9)。因此，在选路报文中，进程没有说明的任何度量，其默认值都是0。

20.8 raw_input函数

向一个进程发送的所有选路报文 —— 包括由内核产生的和由进程产生的 —— 都被传递给raw_input，后者选择接收这个报文的进程。图 18-11总结了调用raw_input的四个函数。

当创建一个选路插口时，族总是PF_ROUTE；而协议， socket的第三个参数，可能为0，表示进程想要接收所有的选路报文；或者是一个如同 AF_INET的值，限制插口只接收包含指定协议族地址的报文。为每个选路插口创建一个选路控制块 (20.3节)，这两个值分别存储在rcb_proto结构的sp_family和sp_protocol成员中。

图20-17显示了raw_input函数。

raw_usrreq.c

```
51 void
52 raw_input(m0, proto, src, dst)
53 struct mbuf *m0;
54 struct sockproto *proto;
55 struct sockaddr *src, *dst;
56 {
57     struct rawcb *rp;
58     struct mbuf *m = m0;
59     int     sockets = 0;
60     struct socket *last;
61     last = 0;
62     for (rp = rawcb.rcb_next; rp != &rawcb; rp = rp->rcb_next) {
63         if (rp->rcb_proto.sp_family != proto->sp_family)
64             continue;
65         if (rp->rcb_proto.sp_protocol &&
66             rp->rcb_proto.sp_protocol != proto->sp_protocol)
67             continue;
68         /*
69          * We assume the lower level routines have
70          * placed the address in a canonical format
71          * suitable for a structure comparison.
72          *
73          * Note that if the lengths are not the same
74          * the comparison will fail at the first byte.
75          */
76 #define equal(a1, a2) \
77   (bcmp((caddr_t)(a1), (caddr_t)(a2), a1->sa_len) == 0)
78         if (rp->rcb_laddr && !equal(rp->rcb_laddr, dst))
79             continue;
80         if (rp->rcb_faddr && !equal(rp->rcb_faddr, src))
81             continue;
82         if (last) {
83             struct mbuf *n;
84             if (n = m_copy(m, 0, (int) M_COPYALL)) {
85                 if (sbappendaddr(&last->so_rcv, src,
86                                  n, (struct mbuf *) 0) == 0)
```

图20-17 raw_input 函数：将选路报文传递给0个或多个进程

```
87                      /* should notify about lost packet */
88                      m_freem(n);
89                  else {
90                      sorwakeup(last);
91                      sockets++;
92                  }
93              }
94          }
95          last = rp->rcb_socket;
96      }
97      if (last) {
98          if (sbappendaddr(&last->so_rcv, src,
99                           m, (struct mbuf *) 0) == 0)
100             m_freem(m);
101         else {
102             sorwakeup(last);
103             sockets++;
104         }
105     } else
106         m_freem(m);
107 }
```

raw_usrreq.c

图20-17 (续)

51-61 在我们所看到的四个对raw_input的调用中，proto、src和dst参数指向三个全局变量route_proto、route_src和route_dst，这些变量都如同图19-26所示的那样被声明和初始化。

1. 比较地址族和协议

62-67 for循环遍历每个选路控制块来查找一个匹配。控制块里的族(一般是PF_ROUTE)必须与sockproto结构的族相匹配，否则这个控制块就被略过。接下来，如果控制块里的协议(socket的第三个参数)非空，它必须匹配sockproto结构的族；否则，这个报文被略去。因此，以0协议创建了一个选路插口的进程将收到所有的选路报文。

2. 比较本地的和外部的地址

68-81 如果指定了的话，这两个测试比较了控制块里的本地地址和外部地址。目前，进程不能设置控制块的rcb_laddr或者rcb_faddr成员。一般来说，进程使用bind设置前者，用connect设置后者，但对于Net/3中的选路插口这是不可能的。作为替代，我们将看到route_usrreq将插口固定地连接到route_src插口地址结构，这是可行的，因为它总是这个函数的src参数。

3. 将报文添加到插口的接收缓存中

82-107 如果last非空，它指向最近看到的应该接收这个报文的socket结构。如果这个变量非空，就使用m_copy和sbappendaddr将这个报文的一个复制添加到那个插口的接收缓存中，并且在这个接收缓存等待的任何进程都会被唤醒。然后，last被设置成指向在以前的测试中刚刚匹配的插口。使用last是为了在只有一个进程接收报文的情况下避免调用m_copy(一个代价昂贵的操作)。

如果有N个进程接收报文，那么前$N-1$个接收一个复制报文，最后一个进程收到的是这个报文本身。

在这个函数里递增的socket变量并没有被用到。因为只有当报文被传递给一个进程后它才会被递增，所以，如果在函数的结尾这个变量的值是0，就表示没有进程接收该报文(但是

变量值没有在任何地方保存)。

20.9 route_usrreq函数

route_usrreq是选路协议的用户请求函数。它被不同的操作调用。图 20-18显示了这个函数。

rtsock.c

```
 64 int
 65 route_usrreq(so, req, m, nam, control)
 66 struct socket *so;
 67 int     req;
 68 struct mbuf *m, *nam, *control;
 69 {
 70     int     error = 0;
 71     struct rawcb *rp = sotorawcb(so);
 72     int     s;

 73     if (req == PRU_ATTACH) {
 74         MALLOC(rp, struct rawcb *, sizeof(*rp), M_PCB, M_WAITOK);
 75         if (so->so_pcb = (caddr_t) rp)
 76             bzero(so->so_pcb, sizeof(*rp));
 77     }
 78     if (req == PRU_DETACH && rp) {
 79         int     af = rp->rcb_proto.sp_protocol;
 80         if (af == AF_INET)
 81             route_cb.ip_count--;
 82         else if (af == AF_NS)
 83             route_cb.ns_count--;
 84         else if (af == AF_ISO)
 85             route_cb.iso_count--;
 86         route_cb.any_count--;
 87     }
 88     s = splnet();
 89     error = raw_usrreq(so, req, m, nam, control);
 90     rp = sotorawcb(so);
 91     if (req == PRU_ATTACH && rp) {
 92         int     af = rp->rcb_proto.sp_protocol;
 93         if (error) {
 94             free((caddr_t) rp, M_PCB);
 95             splx(s);
 96             return (error);
 97         }
 98         if (af == AF_INET)
 99             route_cb.ip_count++;
100         else if (af == AF_NS)
101             route_cb.ns_count++;
102         else if (af == AF_ISO)
103             route_cb.iso_count++;
104         route_cb.any_count++;

105         rp->rcb_faddr = &route_src;
106         soisconnected(so);
107         so->so_options |= SO_USELOOPBACK;
108     }
109     splx(s);
110     return (error);
111 }
```

rtsock.c

图20-18 route_usrreq 函数：处理PRU_xxx请求

1. PRU_ATTACH：分配控制块

64-77 当进程调用socket时，就会发出PRU_ATTACH请求。为一个选路控制块分配内存。MALLOC返回的指针保存在socket结构的so_pcb成员中。如果分配了内存，rawcb结构被设置成0。

2. PRU_DETACH：计数器递减

78-87 close系统调用发出PRU_DETACH请求。如果socket结构指向一个协议控制块，route_cb结构的计数器中有两个被减1：一个是any_count；另一个是基于该协议的计数器。

3. 处理请求

88-90 函数raw_usrreq被调用来进一步处理PRU_*xxx*请求。

4. 计数器递增

91-104 如果请求是PRU_ATTACH，并且插口指向一个选路控制块，就要检查raw_usrreq是否返回一个错误。然后，route_cb结构的计数器中的两个被递增：一个是any_count，另一个是基于该协议的计数器。

5. 连接插口

105-106 选路控制块里的外部地址被设置成route_src。这将永久地连接到新的插口来接收PF_ROUTE族的选路报文。

6. 默认情况下使能SO_USELOOPBACK

107-111 使能SO_USELOOPBACK插口选项。这是一个默认使能的插口选项——其他所有的选项默认都被禁止。

20.10 raw_usrreq函数

raw_usrreq完成在选路域中用户请求处理的大部分工作。在上一节中它被route_usrreq函数所调用。用户请求的处理被划分成这两个函数，是因为其他的一些协议(例如OSI CLNP)调用raw_usrreq而不是route_usrreq。raw_usrreq并不是想要成为一个协议的pr_usrreq函数，相反，它是一个被不同的pr_usrreq函数调用的公共的子例程。

图20-19显示了raw_usrreq函数的开始和结尾。其中的switch语句体在该图后面的图中单独讨论。

1. PRU_CONTROL请求是不合法的

119-129 PRU_CONTROL请求来自于ioctl系统调用，在路由选择域中不被支持。

2. 控制信息不合法

130-133 如果进程传递控制信息(使用sendmsg系统调用)，就会返回一个错误，因为路由选择域中不使用这个可选的信息。

3. 插口必须有一个控制块

134-137 如果socket结构没有指向一个选路控制块，就返回一个错误。如果创建了一个新的插口，调用者(即route_usrreq)有责任在调用这个函数之前分配这个控制块，并且将指针保存在so_pcb成员中。

262-269 这个switch语句的default子句处理case子句没有处理的两个请求：PRU_BIND

和PRU_CONNECT。这两个请求的代码是提供的，但在Net/3中被注释掉了。因此，如果在一个选路插口上发出bind或connect系统调用，就会引起一个内核的告警(panic)。这是一个程序错误(bug)。幸运的是创建这种类型的插口需要有超级用户的权限。

raw_usrreq.c

```
119 int
120 raw_usrreq(so, req, m, nam, control)
121 struct socket *so;
122 int     req;
123 struct mbuf *m, *nam, *control;
124 {
125     struct rawcb *rp = sotorawcb(so);
126     int     error = 0;
127     int     len;

128     if (req == PRU_CONTROL)
129         return (EOPNOTSUPP);
130     if (control && control->m_len) {
131         error = EOPNOTSUPP;
132         goto release;
133     }
134     if (rp == 0) {
135         error = EINVAL;
136         goto release;
137     }
138     switch (req) {

                /* switch cases */

262     default:
263         panic("raw_usrreq");
264     }
265 release:
266     if (m != NULL)
267         m_freem(m);
268     return (error);
269 }
```

raw_usrreq.c

图20-19 raw_usrreg 函数体

我们现在讨论单个的case语句。图20-20显示了对PRU_ATTACH和PRU_DETACH请求的处理。

139-148 PRU_ATTACH请求是socket系统调用的一个结果。一个选路插口只能由一个超级用户的进程创建。

149-150 函数raw_attach(图20-24)将控制块链接到双向链接列表中。 nam参数是socket的第三个参数，被存储在控制块中。

151-159 PRU_DETACH是由close系统调用发出的请求。对一个空的rp指针的测试是多余的，因为在switch语句之前已经进行过这个测试了。

160-161 raw_detach(图20-25)从双向链接表中删除这个控制块。

图20-21显示了PRU_CONNECT2、PRU_DISCONNECT和PRU_SHUTDOWN请求的处理。

186-188 PRU_CONNECT2请求来自于socketpair系统调用，在路由选择域中不被支持。

189-196 因为一个选路插口总是连接的(图20-18)，所以PRU_DISCONNECT请求在PRU_DETACH请

raw_usrreq.c

```
139         /*
140          * Allocate a raw control block and fill in the
141          * necessary info to allow packets to be routed to
142          * the appropriate raw interface routine.
143          */
144     case PRU_ATTACH:
145         if ((so->so_state & SS_PRIV) == 0) {
146             error = EACCES;
147             break;
148         }
149         error = raw_attach(so, (int) nam);
150         break;

151         /*
152          * Destroy state just before socket deallocation.
153          * Flush data or not depending on the options.
154          */
155     case PRU_DETACH:
156         if (rp == 0) {
157             error = ENOTCONN;
158             break;
159         }
160         raw_detach(rp);
161         break;
```

raw_usrreq.c

图20-20 raw_usrreq 函数：PRU_ATTACH 和PRU_DETACH 请求

求之前由close发出。插口必须已经和一个外部地址相连接，这对于一个选路插口来说总是成立的。raw_disconnect和soisdisconnected完成这个处理。

197-202 当参数指定在这个插口上没有更多的写操作时，shutdown系统调用发出PRU_SHUTDOWN请求。socantsendmore禁止以后的写操作。

raw_usrreq.c

```
186     case PRU_CONNECT2:
187         error = EOPNOTSUPP;
188         goto release;

189     case PRU_DISCONNECT:
190         if (rp->rcb_faddr == 0) {
191             error = ENOTCONN;
192             break;
193         }
194         raw_disconnect(rp);
195         soisdisconnected(so);
196         break;

197         /*
198          * Mark the connection as being incapable of further input.
199          */
200     case PRU_SHUTDOWN:
201         socantsendmore(so);
202         break;
```

raw_usrreq.c

图20-21 raw_usrreq 函数：PRU_CONNECT2 、PRU_DISCONNECT 和PRU_SHUTDOWN 请求

对一个选路插口最常见的请求：PRU_SEND、PRU_ABORT和PRU_SENSE显示在图20-22中。

203-217 当进程向插口写时，sosend发出了PRU_SEND请求。如果指定了一个nam参数，

raw_usrreq.c

```
203         /*
204          * Ship a packet out.  The appropriate raw output
205          * routine handles any massaging necessary.
206          */
207     case PRU_SEND:
208         if (nam) {
209             if (rp->rcb_faddr) {
210                 error = EISCONN;
211                 break;
212             }
213             rp->rcb_faddr = mtod(nam, struct sockaddr *);
214         } else if (rp->rcb_faddr == 0) {
215             error = ENOTCONN;
216             break;
217         }
218         error = (*so->so_proto->pr_output) (m, so);
219         m = NULL;
220         if (nam)
221             rp->rcb_faddr = 0;
222         break;

223     case PRU_ABORT:
224         raw_disconnect(rp);
225         sofree(so);
226         soisdisconnected(so);
227         break;

228     case PRU_SENSE:
229         /*
230          * stat: don't bother with a blocksize.
231          */
232         return (0);
```

raw_usrreq.c

图20-22 raw_usrreq 函数：PRU_SEND 、PRU_ABORT 和PRU_SENSE 请求

即进程使用 sendto或者sendmsg指定了一个目的地址，就会返回一个错误，因为route_usrreq总是为一个选路插口设置rcb_faddr。

218-222 m指向的mbuf链中的信息被传递给协议的pr_output函数，也就是route_output。

223-227 如果发出了一个PRU_ABORT请求，则该控制块被断开连接，插口被释放，然后被断开连接。

228-232 fstat系统调用发出PRU_SENSE请求。函数返回OK。

图20-23显示了剩下的PRU_*xxx*请求。

raw_usrreq.c

```
233         /*
234          * Not supported.
235          */
236     case PRU_RCVOOB:
237     case PRU_RCVD:
238         return (EOPNOTSUPP);

239     case PRU_LISTEN:
240     case PRU_ACCEPT:
241     case PRU_SENDOOB:
242         error = EOPNOTSUPP;
```

图20-23 raw_usrreq 函数：最后部分

```
243         break;
244     case PRU_SOCKADDR:
245         if (rp->rcb_laddr == 0) {
246             error = EINVAL;
247             break;
248         }
249         len = rp->rcb_laddr->sa_len;
250         bcopy((caddr_t) rp->rcb_laddr, mtod(nam, caddr_t), (unsigned) len);
251         nam->m_len = len;
252         break;
253     case PRU_PEERADDR:
254         if (rp->rcb_faddr == 0) {
255             error = ENOTCONN;
256             break;
257         }
258         len = rp->rcb_faddr->sa_len;
259         bcopy((caddr_t) rp->rcb_faddr, mtod(nam, caddr_t), (unsigned) len);
260         nam->m_len = len;
261         break;
```

raw_usrreq.c

图20-23 (续)

233-243 这五个请求不被支持。

244-261 PRU_SOCKADDR和PRU_PEERADDR请求分别来自于getsockname和getpeername系统调用。前者总是返回一个错误，因为设置本地地址的bind系统调用在路由选择域中不被支持。后者总是返回插口地址结构route_src的内容，这个内容是由route_usrreq作为外部地址设置的。

20.11 raw_attach、raw_detach和raw_disconnect函数

raw_attach函数，显示在图20-24中，被raw_input调用来完成PRU_ATTACH请求的处理。

raw_cb.c

```
49 int
50 raw_attach(so, proto)
51 struct socket *so;
52 int     proto;
53 {
54     struct rawcb *rp = sotorawcb(so);
55     int     error;
56     /*
57      * It is assumed that raw_attach is called
58      * after space has been allocated for the
59      * rawcb.
60      */
61     if (rp == 0)
62         return (ENOBUFS);
63     if (error = soreserve(so, raw_sendspace, raw_recvspace))
64         return (error);
65     rp->rcb_socket = so;
66     rp->rcb_proto.sp_family = so->so_proto->pr_domain->dom_family;
67     rp->rcb_proto.sp_protocol = proto;
68     insque(rp, &rawcb);
69     return (0);
70 }
```

raw_cb.c

图20-24 raw_attach 函数

49-64 调用者必须已经分配了原始的协议控制块。soreserve将发送和接收缓存的高水位标记设置为8192。这对于选路报文应该是绰绰有余了。

65-67 socket结构的一个指针和dom_family(即图20-1中用于选路域的PF_ROUTE)以及proto参数(socket调用的第三个参数)一起被存储到协议控制块中。

68-70 insque将这个控制块加入到由全局变量rawcb作为头指针的双向链接表的前面。

raw_detach函数，显示在图20-25中，被raw_input调用来完成PRU_DETACH请求的处理。

raw_cb.c

```
75 void
76 raw_detach(rp)
77 struct rawcb *rp;
78 {
79     struct socket *so = rp->rcb_socket;

80     so->so_pcb = 0;
81     sofree(so);
82     remque(rp);
83     free((caddr_t) (rp), M_PCB);
84 }
```

raw_cb.c

图20-25 raw_detach 函数

75-84 socket结构中的so_pcb指针被设置成空，然后释放这个插口。使用remque从双向链接表中删除该控制块，使用free来释放被控制块占用的内存。

raw_disconnect函数，显示在图 20-26中，被raw_input调用来完成 PRU_DISCONNECT和PRU_ABORT请求的处理。

88-94 如果该插口没有引用一个描述符，raw_detach释放该插口和控制块。

raw_cb.c

```
88 void
89 raw_disconnect(rp)
90 struct rawcb *rp;
91 {

92     if (rp->rcb_socket->so_state & SS_NOFDREF)
93         raw_detach(rp);
94 }
```

raw_cb.c

图20-26 raw_disconnect 函数

20.12 小结

一个选路插口是PF_ROUTE域中的一个原始插口。选路插口只能被一个超级用户进程创建。如果一个没有权限的进程想要读内核包含的选路信息，可以使用选路域所支持的 sysctl 系统调用(我们在前一章中描述过)。

在本章中，我们第一次碰到了与插口相联系的协议控制块 (PCB)。在选路域中，一个专门的rawcb包含了有关选路插口的信息：本地和外部的地址、地址族和协议。我们将在第 22章中看到用于UDP、TCP和原始IP插口的更大的Internet协议控制块(inpcb)。然而概念是相同的：socket结构被插口层使用，而 PCB，一个rawcb或一个inpcb，被协议层使用。socket结构指向该PCB，后者也指向前者。

route_output函数处理进程可以发出的五个请求。依赖于协议和地址族，raw_input将一个选路报文发送给一个或多个选路插口。对一个选路插口的不同的 PRU_*xxx*请求由raw_usrreq和route_usrreq处理。在后面的章节中，我们将碰到另外的 *xxx*_usrreq函数，每个协议(UDP、TCP和原始IP)对应一个，每个函数都由一个 switch语句组成用来处理每一个请求。

习题

20.1 当进程向一个选路插口写一个报文时，列出两种进程可以从 route_output收到返回值的方法。哪种方法更可靠？

20.2 因为routesw结构的pr_protocol成员为0，所以当进程对socket系统调用指定了一个非0的*protocol*参数时，会发生什么情况？

20.3 路由表中的路由(和ARP项不同)永远不会超时。试在路由上实现一个超时机制。

第21章 ARP：地址解析协议

21.1 介绍

地址解析协议(ARP)用于实现IP地址到网络接口硬件地址的映射。常见的以太网网络接口硬件地址长度为48 bit。ARP同时也可以工作在其他类型的数据链路下，但在本章中，我们只考虑将IP地址映射到48 bit的以太网地址。ARP在RFC 826［Plummer 1982]中定义。

当某主机要向以太网中另一台主机发送IP数据时，它首先根据目的主机的IP地址在ARP高速缓存中查询相应的以太网地址， ARP高速缓存是主机维护的一个IP地址到相应以太网地址的映射表。如果查到匹配的结点，则相应的以太网地址被写入以太网帧首部，数据报被加入到输出队列等候发送。如果查询失败， ARP会先保留待发送的IP数据报，然后广播一个询问目的主机硬件地址的ARP报文，等收到回答后再将IP数据报发送出去。

以上只是简要描述了ARP协议的基本工作过程，下面我们将结合Net/3中的ARP实现来详细描述其具体细节。卷1的第4章包含了ARP的例子。

21.2 ARP和路由表

Net/3中ARP的实现是和路由表紧密关联的，这也是为什么我们要在描述路由表结构之后再来讲解ARP的原因。图21-1显示了本章中我们描述ARP要用到的一个例子。整个图是与本书中用到的网络实例相对应的，它显示了bsdi主机上当前ARP缓存的相关结构。其中Ifnet、ifaddr和in_ifaddr结构是由图3-32和图6-5简化而来的，所以在这里忽略了在第3章和第6章中描述过的这三个结构中的某些细节。例如，图中没有画出在两个 ifaddr结构之后的sockaddr_dl结构——而仅仅是概述了这两个结构中的相应信息。同样，我们也仅仅是概述了三个in_ifaddr结构中的信息。

下面，我们简要概述图中的有关要点。细节部分将随着本章的进行而详细展开。

1) llinfo_arp结构的双向链表包含了每一个ARP已知的硬件地址的少量信息。同名全局变量llinfo_arp是该链表的头结点，图中没有画出第一位的la_prev指针指向最后一项，最后一项的la_next指针指向第一项。该链表由ARP时钟函数每隔5分钟处理一次。

2) 每一个已知硬件地址的IP地址都对应一个路由表结点(rtentry结构)。llinfo_arp结构的 la_rt指针成员用来指向相应的 rtentry结构，同样地， rtentry结构的rt_llinfo指针成员指向 llinfo_arp结构。图中对应主机 sun(140.252.13.33)、svr4(140.252.13.34)和bsdi(140.252.13.35)的三个路由表结点各自具有相应的 llinfo_arp结构。如图18-2所示。

3) 而在图的最左边第四个路由表结点则没有对应的 llinfo_arp结构，该结点对应于本地以太网(140.252.13.32)的路由项。该结点的rt_flags中设置了C比特，表明该结点是被用来复制形成其他结点的。设置接口 IP 地址功能的in_ifinit函数(图6-19)通过调用rtinit函数来创建该结点。其他三个结点是主机路由结点 (H标志)，并由bsdi向其他机器发送数据

时通过ARP间接调用路由相关函数产生的(L标志)。

4) rtentry结构中的rt_gateway指针成员指向一个sockaddr_dl结构变量。如果保存物理地址长度的结构sdl_alen成员为6，那么sockaddr_dl结构就包含相应的硬件地址信息。

5) 路由结点变量的rt_ifp成员的相应指针成员指向对应网络设备接口的ifnet结构。中间的两个路由结点对应的是以太网上的其他主机，这两个结点都指向le_softc[0]。而右边的路由结点对应的是bsdi，指向环回结构loif。因为rt_ifp.if_output指向输出函数，所以目的为本机的数据报被路由至环回接口。

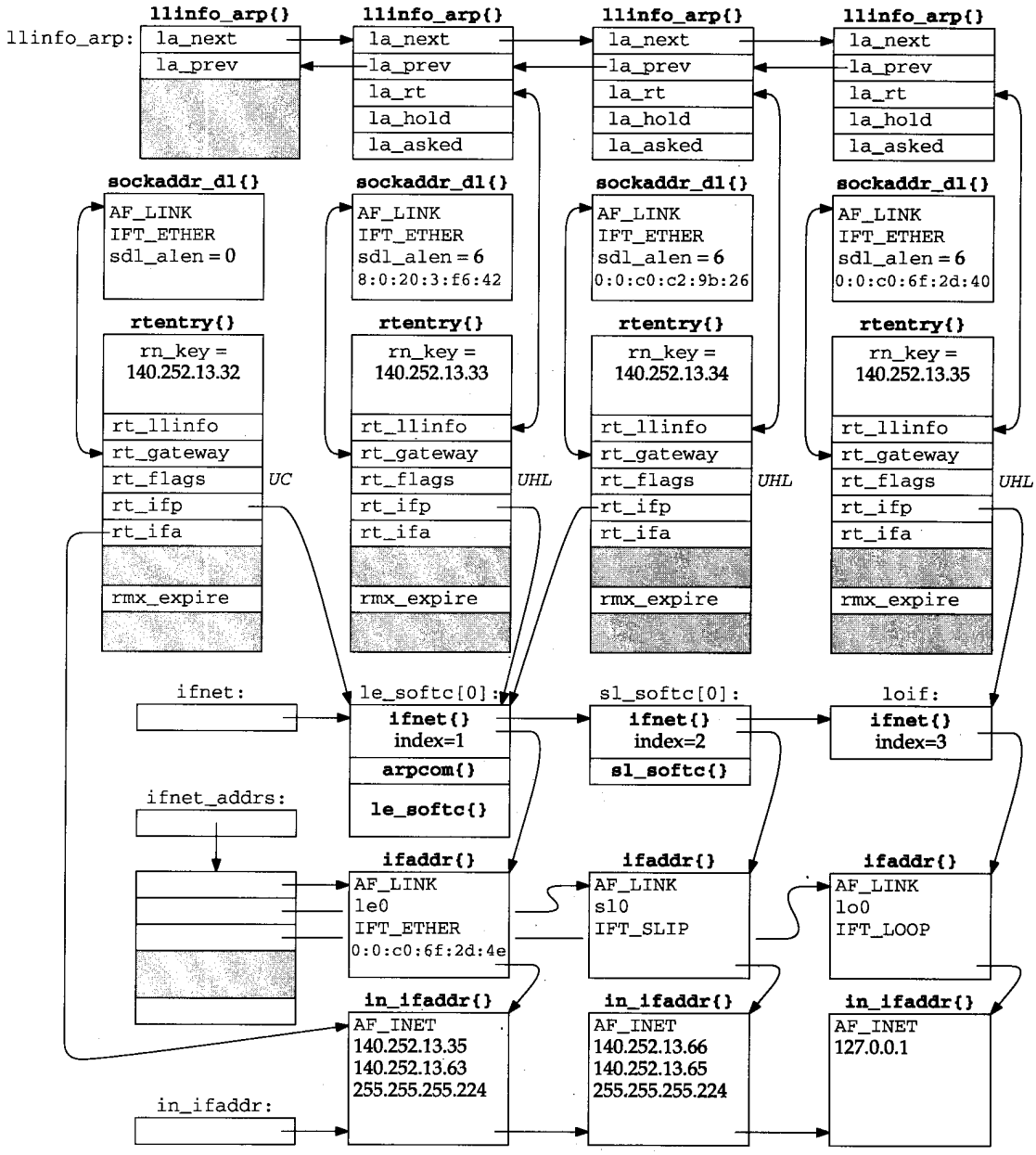


图21-1 ARP与路由表和接口结构的关系

6) 每一个路由结点还有指向相应的 in_ifaddr结构的指针变量 (图6-8中指出了in_ifaddr结构内的第一个成员是一个ifaddr结构，因此，rt_ifa同样是指向了ifaddr结构变量)。在本图中，我们只显示一个路由结点的相应指向，其余的路由结点具有同样的性质。而一个接口如le0，可以同时设置多个IP地址，每个IP地址都有对应的in_ifaddr结构，这就是为什么除了rt_ifp之外还需要rt_ifa的原因。

7) la_hold成员是指向mbuf链表的指针。当要向某个IP传送数据报时，就需要广播一个ARP请求。当内核等待ARP回答时，存放该待发数据报的 mbuf链的头结点的地址信息就存放在la_hold里。当收到ARP回答后，la_hold指向的mbuf链表中的IP数据被发送出去。

8) 路由表结点中rt_metric结构的变量rmx_expire存放的是与对应的ARP结点相关的定时信息，用来实现删除超时 (通常20分钟)的ARP结点。

在4.3BSD Reno中，路由表结构定义有了很大的变化，但 4.3BSD Reno和Net/2. 4.4BSD中依然定义有 ARP缓存，只是去除了作为单独结构的 ARP缓存链表，而把ARP信息放在了路由表结点里。

在Net/2中，ARP表是一个结构数组，其中每个元素包含有以下成员： IP地址、以太网地址、定时器、标志和一个指向 mbuf的指针(类似于图21-1中的la_hold成员)。在Net/3中，我们可以看到，这些信息被分散到多个相互链接的结构里。

21.3 代码介绍

如图21-2所示，共有包含9个ARP函数的一个C文件和两个头文件。

文　件	描　述
net/if_arp.h	arphdr结构的定义
netinet/if_ether.h	多个结构和常量的定义
netinet/if_ether.d	ARP函数

图21-2 本章中讨论的文件

图21-3显示了ARP函数与其他内核函数的关系。该图中还说明了 ARP函数与第19章中某些子函数的关系，下面将逐步解释这些关系。

21.3.1 全局变量

本章中将介绍10个全局变量，如图21-4所示。

21.3.2 统计量

保存ARP的统计量有两个全局变量：arp_inuse和arp_allocated，如图21-4所示。前者用来记录当前正在使用的 ARP结点数，后者用来记录在系统初始化时分配的 ARP结点数。两个统计数都不能由netstat程序输出，但可以通过调试器来查看。

可以使用命令arp -a来显示当前ARP缓存的信息，该命令使用 sysctl系统调用，参数如图19-36所示。图21-5显示该命令的一个输出结果。

由于图18-2中对应多播组224.0.0.1的相应路由表项设置了 L标志，而同时由于arp程序查询带有RTF_LLINFO标志位的ARP结点，所以该程序也输出多播地址。后面我们将解释为什么该表项标识为“ incomplete ”，而在它上面的表项是“ permanent ”。

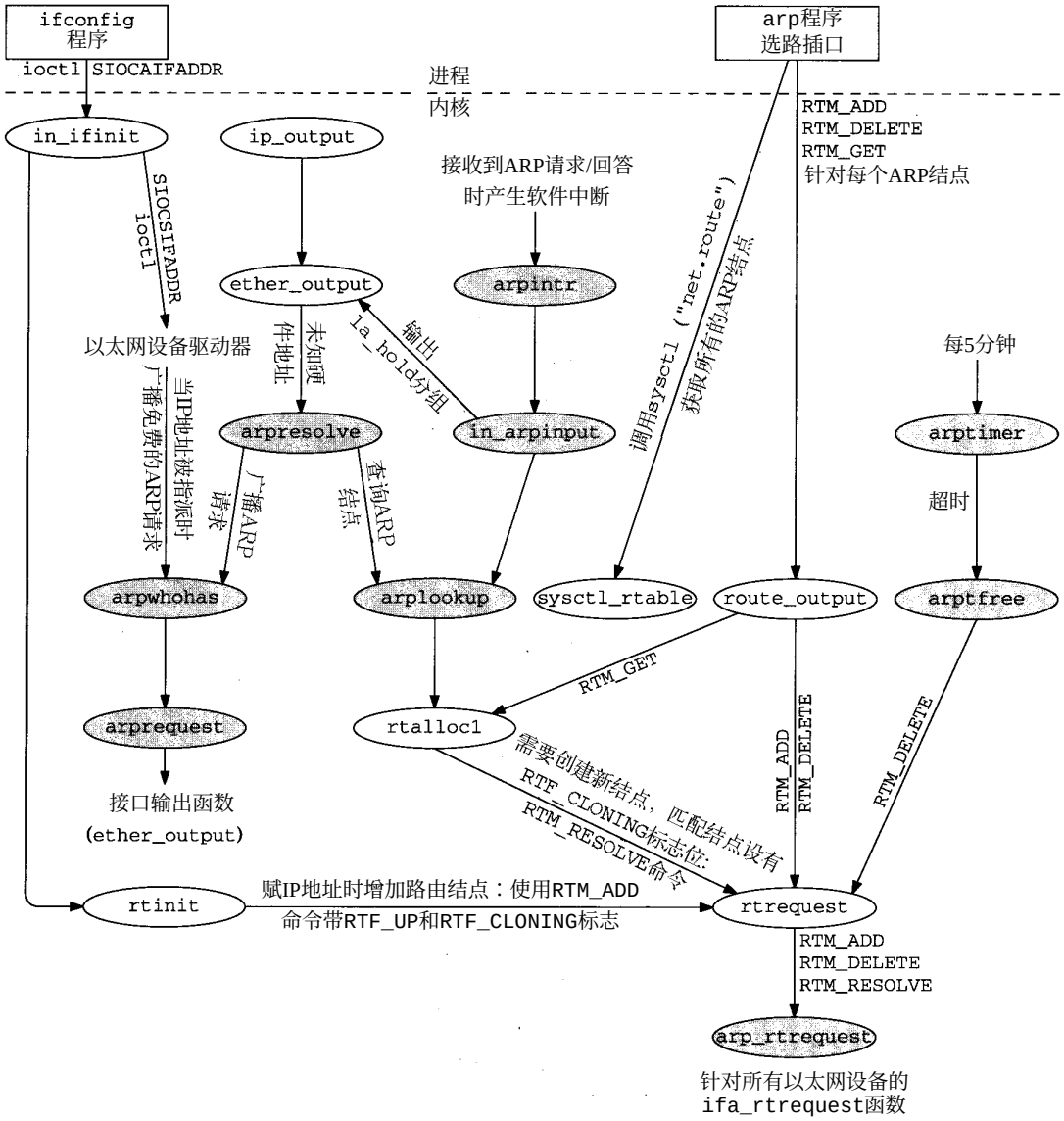


图21-3 ARP函数和内核中其他函数的关系

变 量	数据类型	描 述
llinfo_arp	struct llinfo_arp	双向链接表的表头
arpintrq	struct ifqueue	来自以太网设备驱动程序的ARP输入队列
arpt_prune	Int	检查ARP链表的时间间隔的分钟数(5)
arpt_keep	Int	ARP结点的有效时间的分钟数(20)
arpt_down	Int	ARP洪泛算法的时间间隔的秒数(20)
arp_inuse	Int	正在使用的ARP结点数
arp_allocated	Int	已经分配的ARP结点数
arp_maxtries	Int	对一个IP地址发送ARP请求的重试次数(5)
arpinit_done	Int	初始化标志
uselookback	int	对本机使用环回(默认)

图21-4 本章介绍的全局变量

```
bsdi $ arp -a
sun.tuc.noao.edu (140.252.13.33) at 8:0:20:3:f6:42
svr4.tuc.noao.edu (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi.tuc.noao.edu (140.252.13.35) at 0:0:c0:6f:2d:40 permanent
ALL-SYSTEMS.MCAST.NET (224.0.0.1) at (incomplete)
```

图21-5 与图18-2相应的arp -a命令的输出

21.3.3 SNMP变量

在卷1的25.8节中我们讲过，最初的SNMP MIB定义了一个地址映射组，该组对应的是系统的当前ARP缓存信息。在MIB-II中不再使用该组，而用各个网络协议组(如IP组)分别包含地址映射表来替代。注意，从Net/2到Net/3，将单独结构的ARP缓存演化为在路由表中集成的ARP信息是与SNMP的变化并行的。

IP地址映射表， index = <*ipNetToMediaIfIndex*>.<*ipNetToMediaNetAddress*>		
名　称	成　员	描　述
ipNetToMediaIfIndex	if_index	相应接口：ifIndex
ipNetToMediaPhysAddress	rt_gateway	硬件地址
ipNetToMediaNetAddress	rt_key	IP地址
ipNetToMediaType	rt_flags	映射类型：1=其他，2=失效，3=动态，4=静态(见正文)

图21-6 IP地址映射表：ipNetToMediaTable

图21-6所示的是MIB-II中的一个IP地址映射表，ipNetToMediaTable，该表保存的值来自于路由表结点和相应的ifnet结构。

如果路由表结点的生存期为0，则被认为是永久的，也即静态的。否则就是动态的。

21.4 ARP 结构

在以太网中传送的ARP分组的格式图21-7所示。

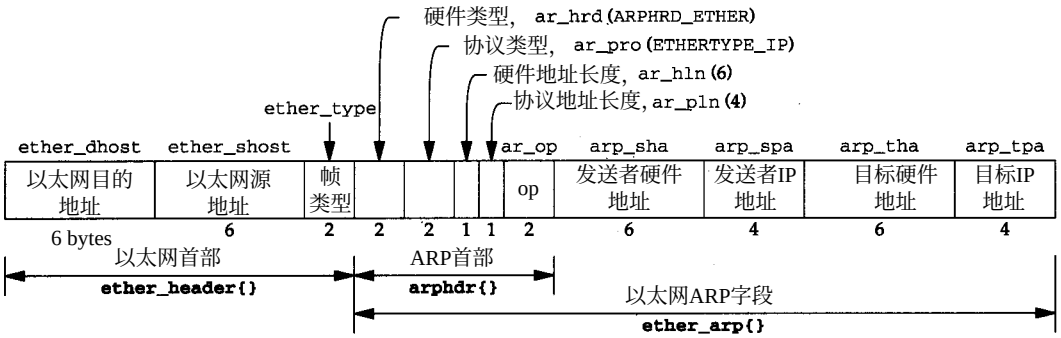


图21-7 在以太网上使用时ARP请求或回答的格式

结构ether_header定义了以太网帧首部；结构arphdr定义了其后的5个字段，其信息用于在任何类型的介质上传送ARP请求和回答；ether_arp结构除了包含arphdr结构外，还包含源主机和目的主机的地址。

结构arphdr的定义如图21-8所示。图21-7显示了该结构中的前4个字段。

if_arp.h

```
45 struct arphdr {
46     u_short ar_hrd;             /* format of hardware address */
47     u_short ar_pro;             /* format of protocol address */
48     u_char  ar_hln;             /* length of hardware address */
49     u_char  ar_pln;             /* length of protocol address */
50     u_short ar_op;              /* ARP/RARP operation, Figure 21.15 */
51 };
```

if_arp.h

图21-8 arphdr 结构：通用的ARP请求/回答数据首部

图21-9显示了ether_arp结构的定义，其中包含了arphdr结构、源主机和目的主机的IP地址和硬件地址。注意，ARP用硬件地址来表示48 bit以太网地址，用协议地址来表示32 bit IP地址。

if_ether.h

```
79 struct ether_arp {
80     struct arphdr ea_hdr;       /* fixed-size header */
81     u_char  arp_sha[6];         /* sender hardware address */
82     u_char  arp_spa[4];         /* sender protocol address */
83     u_char  arp_tha[6];         /* target hardware address */
84     u_char  arp_tpa[4];         /* target protocol address */
85 };

86 #define arp_hrd ea_hdr.ar_hrd
87 #define arp_pro ea_hdr.ar_pro
88 #define arp_hln ea_hdr.ar_hln
89 #define arp_pln ea_hdr.ar_pln
90 #define arp_op  ea_hdr.ar_op
```

if_ether.h

图21-9 ether_arp 结构

每个ARP结点中，都有一个llinfo_arp结构，如图21-10所示。所有这些结构组成的链接表的头结点是作为全局变量分配的。我们经常把该链接表称为ARP高速缓存，因为在图21-1中，只有该数据结构是与ARP结点一一对应的。

if_ether.h

```
103 struct llinfo_arp {
104     struct llinfo_arp *la_next;
105     struct llinfo_arp *la_prev;
106     struct rtentry *la_rt;
107     struct mbuf *la_hold;       /* last packet until resolved/timeout */
108     long    la_asked;           /* #times we've queried for this addr */
109 };

110 #define la_timer la_rt->rt_rmx.rmx_expire   /* deletion time in seconds */
```

if_ether.h

图21-10 llinfo_arp结构

在Net/2及以前的系统中，很容易识别作为ARP高速缓存的数据结构，因为每一个ARP结点的信息都存放在单一的结构中。而Net/3则把ARP信息存放在多个结构中，没有哪个数据结构被称为ARP高速缓存。但是为了讨论方便，我们依然用ARP高速缓存的概念来表示一个ARP结点的信息。

104-106 该双向链接表的前两项由insque和remque两个函数更新。la_rt指向相关的路

由表结点，该路由表结点的rt_llinfo成员指向la_rt。

107 当ARP接收到一个要发往其他主机的IP数据报，且不知道相应硬件地址时，必须发送一个ARP请求，并等待回答。在等待ARP回答时，指向待发数据报的指针存放在la_hold中。收到回答后，la_hold所指的数据报被发送出去。

108-109 la_asked记录了连续为某个IP地址发送请求而没有收到回答的次数。在图21-24中，我们可以看到，当这个数值达到某个限定值时，我们就认为该主机是关闭的，并在其后一段时间内不再发送该主机的ARP请求。

110 这个定义使用路由结点中rt_metrics结构的rmx_expire成员作为ARP定时器。当值为0时，ARP项被认为是永久的；当为非零时，值为当结点到期时算起的秒数。

21.5 arpwhohas函数

arpwhohas函数通常由arpresolve调用，用于广播一个ARP请求。如图21-11所示。它还可由每个以太网设备驱动程序调用，在将IP地址赋予该设备接口时主动发送一个地址联编信息(图6-28中的SIOCSIFADDRioctl)。主动发送地址联编信息不但可以检测在以太网中是否存在IP地址冲突，并且可以使其他机器更新其相应信息。arpwhohas只是简单调用下一部分将要介绍的arprequest函数。

if_ether.c

```
196 void
197 arpwhohas(ac, addr)
198 struct arpcom *ac;
199 struct in_addr *addr;
200 {
201     arprequest(ac, &ac->ac_ipaddr.s_addr, &addr->s_addr, ac->ac_enaddr);
202 }
```

if_ether.c

图21-11 arpwhohas函数：广播一个ARP请求

196-202 arpcom结构(图3-26)对所有以太网设备是通用的，是le_softc结构(图3-20)的一部分。ac_ipaddr成员是接口的IP地址的复制，当SIOCSIFADDRioctl执行时由驱动程序填写(图6-28)。ac_enaddr是该设备的以太网地址。

该函数的第二个参数addr，是ARP请求的目的IP地址。在主动发送动态联编信息时，addr等于ac_ipaddr，所以arprequest的第二和第三个参数是一样的，即发送IP地址和目的IP地址在主动发送动态联编信息时是一样的。

21.6 arprequest函数

arprequest函数由arpwhohas函数调用，用于广播一个ARP请求。该函数建立一个ARP请求分组，并将它传送到接口的输出函数。

在分析代码之前，我们先来看一下该函数建立的数据结构。传送ARP请求需要调用以太网设备的接口输出函数ether_output。ether_output的一个参数是mbuf，它包含待发送数据，即图21-7中以太网类型字段后的所有内容。另外一个参数包含目的地址的端口地址结构。通常情况下，该目的地址是IP地址(例如，在图21-3中，ip_output调用ether_output)。特殊情况下，端口地址的sa_family被设为AF_UNSPEC，即告知ether_output它所带的是一个已填充的以太网帧首部，包含了目的主机的硬件地址，这就

防止了ether_output去调用arpreslove而导致死循环。图21-3中没有显示这种循环，在arprequest下面的接口输出函数是 ether_output。如果ether_output再去调用arpresolve，将导致死循环。

图21-12显示了该函数建立的两个数据结构 mbuf和sockaddr。另外还有两个函数中用到的指针eh和ea。

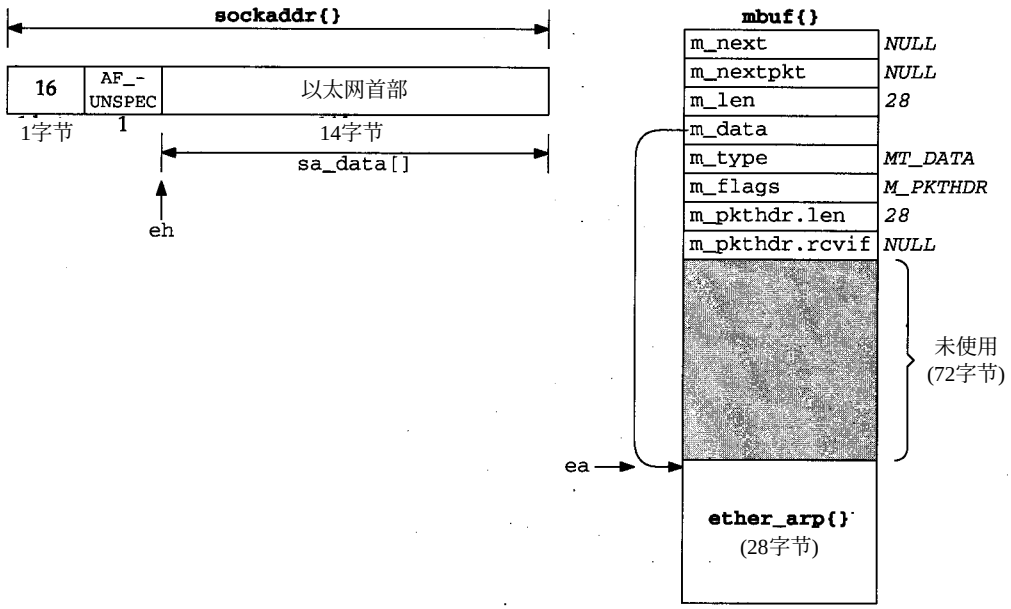


图21-12 arprequest 建立的sockaddr 和mbuf

图21-13给出了arprequest函数的源代码。

if_ether.c

```
209 static void
210 arprequest(ac, sip, tip, enaddr)
211 struct arpcom *ac;
212 u_long *sip, *tip;
213 u_char *enaddr;
214 {
215     struct mbuf *m;
216     struct ether_header *eh;
217     struct ether_arp *ea;
218     struct sockaddr sa;

219     if ((m = m_gethdr(M_DONTWAIT, MT_DATA)) == NULL)
220         return;
221     m->m_len = sizeof(*ea);
222     m->m_pkthdr.len = sizeof(*ea);
223     MH_ALIGN(m, sizeof(*ea));

224     ea = mtod(m, struct ether_arp *);
225     eh = (struct ether_header *) sa.sa_data;
226     bzero((caddr_t) ea, sizeof(*ea));

227     bcopy((caddr_t) etherbroadcastaddr, (caddr_t) eh->ether_dhost,
228           sizeof(eh->ether_dhost));
```

图21-13 arprequest 函数：创建一个ARP请求并发送

```
229     eh->ether_type = ETHERTYPE_ARP;      /* if_output() will swap */

230     ea->arp_hrd = htons(ARPHRD_ETHER);
231     ea->arp_pro = htons(ETHERTYPE_IP);
232     ea->arp_hln = sizeof(ea->arp_sha);   /* hardware address length */
233     ea->arp_pln = sizeof(ea->arp_spa);   /* protocol address length */
234     ea->arp_op = htons(ARPOP_REQUEST);
235     bcopy((caddr_t) enaddr, (caddr_t) ea->arp_sha, sizeof(ea->arp_sha));
236     bcopy((caddr_t) sip, (caddr_t) ea->arp_spa, sizeof(ea->arp_spa));
237     bcopy((caddr_t) tip, (caddr_t) ea->arp_tpa, sizeof(ea->arp_tpa));

238     sa.sa_family = AF_UNSPEC;
239     sa.sa_len = sizeof(sa);

240     (*ac->ac_if.if_output) (&ac->ac_if, m, &sa, (struct rtentry *) 0);
241 }
```

if_ether.c

图21-13 (续)

1. 分配和初始化mbuf

209-223 分配一个分组数据首部的mbuf，并对两个长度字段赋值。MH_ALIGN将28字节的ether_arp结构置于mbuf的尾部，并相应地设置m_data指针的值。将该数据结构置于mbuf尾部，是为了允许ether_output预先考虑将14字节的以太网帧首部置于同一mbuf中。

2. 初始化指针

224-226 给ea和eh两个指针赋值，并将ether_arp结构的值赋为0。bzero的惟一目的是将目的硬件地址置0，该结构中其余8个字段已被设成相应的值。

3. 填充以太网帧首部

227-229 目的以太网地址设为以太网广播地址，并将以太网帧类型设为ETHERTYPE_ARP。注意代码中的注释，接口输出函数将该字段从主机字节序转化为网络字节序，该函数还将填充本机的以太网地址。图21-14显示了不同以太网帧类型字段的常量值。

常 量	值	描 述
ETHERTYPE_IP	0x0800	IP帧
ETHERTYPE_ARP	0x0806	ARP帧
ETHERTYPE_REVARP	0x8035	逆ARP帧
ETHERTYPE_IPTRAILERS	0x1000	尾部封装(已废弃)

图21-14 以太网帧类型字段

RARP 将硬件地址映射成IP地址，通常在无盘工作站系统引导时使用。一般来说， RARP部分不属于内核TCP/IP实现，所以本书将不作描述，卷1的第5章讲述了RARP的概念。

4. 填充ARP字段

230-237 填充了ether_arp的所有字段，除了ARP请求所要询问的目的硬件地址。常量ARPHRD_ETHER的值为1时，表示硬件地址的格式是6字节的以太网地址。为了表示协议地址是4字节的IP地址，arp_pro的值设为图21-14中所指的IP协议地址类型(0x800)。图21-15显示了不同的ARP操作码。本章中，我们将看到前两种。后两种在 RARP中使用。

5. 填充sockaddr，并调用接口输出函数

238-241 接口地址结构的sa_family成员的值设为AP_UNSPEC，sa_member成员的值设为16。调用接口输出函数ether_output。

常 量	值	描 述
ARPOP_REQUEST	1	解析协议地址的ARP请求
ARPOP_REPLY	2	回答ARP请求
ARPOP_REVREQUEST	3	解析硬件地址的RARP请求
ARPOP_REVREPLY	4	回答RARP请求

图21-15 ARP 操作码

21.7 arpintr函数

在图4-13中，当ether_input函数接收到帧类型字段为ETHERTYPE_ARP的以太网帧时，产生优先级为NETISR_ARP的软件中断，并将该帧挂在 ARP输入队列arpintrq的后面。当内核处理该软件中断时，调用arpintr函数，如图21-16所示。

if_ether.c

```
319 void
320 arpintr()
321 {
322     struct mbuf *m;
323     struct arphdr *ar;
324     int     s;

325     while (arpintrq.ifq_head) {
326         s = splimp();
327         IF_DEQUEUE(&arpintrq, m);
328         splx(s);
329         if (m == 0 || (m->m_flags & M_PKTHDR) == 0)
330             panic("arpintr");

331         if (m->m_len >= sizeof(struct arphdr) &&
332             (ar = mtod(m, struct arphdr *)) &&
333             ntohs(ar->ar_hrd) == ARPHRD_ETHER &&
334             m->m_len >= sizeof(struct arphdr) + 2*ar->ar_hln + 2*ar->ar_pln)

335                 switch (ntohs(ar->ar_pro)) {
336                 case ETHERTYPE_IP:
337                 case ETHERTYPE_IPTRAILERS:
338                     in_arpinput(m);
339                     continue;
340                 }

341         m_freem(m);
342     }
343 }
```

if_ether.c

图21-16 arpintr 函数：处理包含ARP请求/回答的以太网帧

319-343 while循环一次处理一个以太网帧，直到处理完队列中的所有帧为止。只有当帧的硬件类型指明为以太网地址，并且帧的长度大于或等于 arphdr结构的长度加上两个硬件地址和两个协议地址的长度时，该帧才能被处理。如果协议地址的类型是 ETHERTYPE_IP或ETHERTYPE_IPTRAILERS时，调用in_arpinput函数，否则该帧将被丢弃。

注意if语句中对条件的检测顺序。共两次检查帧的长度。首先，当帧长大于或等于arphdr结构的长度时，才去检查帧结构中的其他字段；然后，利用 arphdr中的两个长度字段再次检查帧长。

21.8 in_arpinput函数

该函数由arpintr调用，用于处理接收到的ARP请求/回答。ARP本身的概念比较简单，但是加上许多规则后，实现就比较复杂，下面先来看一下两种典型情况：

1) 如果收到一个针对本机IP地址的请求，则发送一个回答。这是一种普通情况。很显然，我们将继续从那个主机收到数据报，随后也会向它回送报文。所以，如果我们还没有对应它的ARP结点，就应该添加一个ARP结点，因为这时我们已经知道了对方的IP地址和硬件地址。这会优化其后与该主机的通信。

2) 如果收到一个ARP回答，那么此时ARP结点是完整的，因此就知道了对方的硬件地址。该地址存放在sockaddr_dl结构中，所有发往该地址的数据报将被发送。
ARP请求是被广播发送的，所以以太网上的所有主机都将看到该请求，当然包括那些非目的主机。回想一下arprequest函数，在发送ARP请求时，帧中包含着请求方的IP地址和硬件地址，这就产生了下面的情况：

3) 如果其他主机发送了一个ARP请求或回答，其中发送方的IP地址与本机相同，那么肯定有一个主机配置有误。Net/3将检测到该差错，并向管理员登记一个报文(这里我们不分请求或回答，因为in_arpinout不检查操作类型，但是ARP回答将被单播，只有目的主机才能收到信息)。

4) 如果主机收到来自其他主机的请求或回答，对应的ARP结点早已存在，但硬件地址发生了变化，那么ARP结点将被更新。这种情况是这样发生的：其他主机以不同的硬件地址重新启动，而本机的对应ARP结点还未失效。这样，根据机器重启动时主动发送动态联编信息，可以使主机不至于因其他主机重启动后导致的ARP结点失效而不能通信。

5).主机可以被配置为代理ARP服务器。这种情况下，主机可以代其他主机响应ARP请求，在回答中提供其他主机的硬件地址。代理ARP回答中对应目的硬件地址的主机必须能够把IP数据报转发至ARP请求中指定的目的主机。卷1的4.6节讨论了代理ARP。

一个Net/3系统可以配置成代理ARP服务器。这些ARP结点可以通过arp命令添加，该命令中指定IP地址、硬件地址并使用关键词pub。我们将在图21-20中看到该实现，并在21-12节讨论其实现细节。

将in_arpinput的分析分为四部分，图21-17显示了第一部分。

358-375 ether_arp结构的长度由调用者(arp_intr函数)验证，所以ea指针指向接收到的分组。ARP操作码(请求或回答)被拷贝至op字段，但具体值要到后面来验证。发送方和目的方的IP地址拷贝到isaddr和itaddr。

1. 查找匹配的接口和IP地址

376-382 搜索本机的Internet地址链表(in_ifaddr结构的链表，图6-5)。要记住一个接口可以有多个IP地址。收到的数据报中有指向接收接口ifnet结构的指针(在mbuf数据报的首部)，for循环只考虑与接收接口相关的IP地址。如果查询到有IP地址等于目的方IP地址或发送方IP地址，则退出循环。

383-384 如果循环退出时，变量maybe_ia的值为0，说明已经搜索了配置的IP地址整个链表而没有找到相关项。函数跳至out(图21-19)，丢弃mbuf，并返回。这种情况只发生在收到ARP请求的接口虽然已初始化但还没有分配IP地址时。

385 如果退出循环时，maybe_ia值不为0，即找到了一个接收端接口，但没有一个IP地址与目的方IP地址或发送方IP地址匹配，则myaddr的值设为该接口的最后一个IP地址；否则(正常情况)，myaddr包含与目的方或发送方的IP地址匹配的本地IP地址。

if_ether.c

```
358 static void
359 in_arpinput(m)
360 struct mbuf *m;
361 {
362     struct ether_arp *ea;
363     struct arpcom *ac = (struct arpcom *) m->m_pkthdr.rcvif;
364     struct ether_header *eh;
365     struct llinfo_arp *la = 0;
366     struct rtentry *rt;
367     struct in_ifaddr *ia, *maybe_ia = 0;
368     struct sockaddr_dl *sdl;
369     struct sockaddr sa;
370     struct in_addr isaddr, itaddr, myaddr;
371     int     op;

372     ea = mtod(m, struct ether_arp *);
373     op = ntohs(ea->arp_op);
374     bcopy((caddr_t) ea->arp_spa, (caddr_t) & isaddr, sizeof(isaddr));
375     bcopy((caddr_t) ea->arp_tpa, (caddr_t) & itaddr, sizeof(itaddr));

376     for (ia = in_ifaddr; ia; ia = ia->ia_next)
377         if (ia->ia_ifp == &ac->ac_if) {
378             maybe_ia = ia;
379             if ((itaddr.s_addr == ia->ia_addr.sin_addr.s_addr) ||
380                 (isaddr.s_addr == ia->ia_addr.sin_addr.s_addr))
381                 break;
382         }
383     if (maybe_ia == 0)
384         goto out;
385     myaddr = ia ? ia->ia_addr.sin_addr : maybe_ia->ia_addr.sin_addr;
```

if_ether.c

图21-17 in_arpinput 函数：查找匹配接口

图21-18显示了in_arpinput函数的第二部分，执行分组的验证。

if_ether.c

```
386     if (!bcmp((caddr_t) ea->arp_sha, (caddr_t) ac->ac_enaddr,
387               sizeof(ea->arp_sha)))
388         goto out;               /* it's from me, ignore it. */
389     if (!bcmp((caddr_t) ea->arp_sha, (caddr_t) etherbroadcastaddr,
390               sizeof(ea->arp_sha))) {
391         log(LOG_ERR,
392             "arp: ether address is broadcast for IP address %x!\n",
393             ntohl(isaddr.s_addr));
394         goto out;
395     }
396     if (isaddr.s_addr == myaddr.s_addr) {
397         log(LOG_ERR,
398             "duplicate IP address %x!! sent from ethernet address: %s\n",
399             ntohl(isaddr.s_addr), ether_sprintf(ea->arp_sha));
400         itaddr = myaddr;
401         goto reply;
402     }
```

if_ether.c

图21-18 in_arpinput 函数：验证接收到的分组

2. 验证发送方的硬件地址

386-388 如果发送方的硬件地址等于本机接口的硬件地址，那是因为收到了本机发出的请求，忽略该分组。

389-395 如果发送方的硬件地址等于以太网的广播地址，说明出了差错。记录该差错，并丢弃该分组。

3. 检查发送方IP地址

396-402 如果发送方的IP地址等于myaddr，说明发送方和本机正在使用同一个IP地址。这也是一个差错——要么是发送方，要么是本机系统配置出了差错。记录该差错，在将目的IP地址设为myaddr后，程序转至reply(图21-19)。注意该ARP分组本来要送往以太网中其他主机的——该分组本来不是要送给本机的。但是，如果这种形式的IP地址欺骗被检测到，应记录差错，并产生回答。

图21-19显示了in_arpinput函数的第三部分。

if_ether.c

```
403     la = arplookup(isaddr.s_addr, itaddr.s_addr == myaddr.s_addr, 0);
404     if (la && (rt = la->la_rt) && (sdl = SDL(rt->rt_gateway))) {
405         if (sdl->sdl_alen &&
406             bcmp((caddr_t) ea->arp_sha, LLADDR(sdl), sdl->sdl_alen))
407             log(LOG_INFO, "arp info overwritten for %x by %s\n",
408                 isaddr.s_addr, ether_sprintf(ea->arp_sha));
409         bcopy((caddr_t) ea->arp_sha, LLADDR(sdl),
410               sdl->sdl_alen = sizeof(ea->arp_sha));
411         if (rt->rt_expire)
412             rt->rt_expire = time.tv_sec + arpt_keep;
413         rt->rt_flags &= ~RTF_REJECT;
414         la->la_asked = 0;
415         if (la->la_hold) {
416             (*ac->ac_if.if_output) (&ac->ac_if, la->la_hold,
417                                     rt_key(rt), rt);
418             la->la_hold = 0;
419         }
420     }

421   reply:
422     if (op != ARPOP_REQUEST) {
423       out:
424         m_freem(m);
425         return;
426     }
```

if_ether.c

图21-19 in_arpinput 函数：创建新的ARP结点或更新已有的ARP结点

4. 在路由表中搜索与发送方IP地址匹配的结点

403 arplookup在ARP高速缓存中查找符合发送方的IP地址(isaddr)。当ARP分组中的目的IP地址等于本机IP时，如果要创建新的ARP结点，那么第二个参数是1，如果不需要创建新的ARP结点，那么第二个参数是0。如果本机就是目的主机，总是要创建ARP结点的，除非一个查找其他主机的广播分组，这种情况下只是在已有的ARP结点中查询。正如前面提到的，如果主机收到一个对应它自己的ARP请求，则说明以太网中有其他主机将要与它通信，所以应该创建一个对应该主机的ARP结点。

第三个参数是0，意味着不去查找代理ARP结点(后面要证明)。返回值是指向llinfo_

arp结构的指针；如果查不到或没有创建，返回值就是空。

5. 更新已有结点或填充新的结点

404 只有当以下三个条件为真时if语句才执行：

1) 找到一个已有的ARP结点或成功创建一个新的ARP结点(即la非空)；

2) ARP结点指向一个路由表结点(rt)；

3) 路由表结点的re_gateway字段指向一个sockaddr_dl结构。

对于每一个目的并非本机的广播 ARP请求，如果发送方的IP地址不在路由表，则第一个条件为假。

6. 检查发送方的硬件地址是否已改变

405-408 如果链路层地址长度(sdl_alen)非0，说明引用的路由表结点是现存的而非新创建的，则比较链路层地址和发送方的硬件地址。如果不同，则说明发送方的硬件地址已经改变，这是因为发送方以不同的以太网地址重新启动了系统，而本机的 ARP结点还未超时。这种情况虽然很少出现，但也必须考虑到。记录差错信息后，程序继续往下执行，更新 ARP结点的硬件地址。

在这个记录报文中，发送方的IP地址必须转换为主机字节序，这是一个错误。

7. 记录发送方硬件地址

409-410 将发送方的硬件地址写入路由表结点中rt_gateway成员指向的sockaddr_dl结构。sockaddr_dl结构的链路层地址长度(sdl_alen)也被设为6。该赋值对于最近创建的ARP结点是需要的(习题21-3)。

8. 更新最近解析的ARP结点

411-412 在解析了发送方的硬件地址后，执行以下步骤。如果时限是非零的，则将被复位成20分钟(arpt_keep)。arp命令可以创建永久的ARP结点，即该结点永远不会超时。这些ARP结点的时限值置为0。在图21-24中我们将看到，在发送ARP请求(非永久性ARP结点)时，时限被设为本地时间，它是非0的。

413-414 清除RTF_REJECT标志，la_asked计数器设为0。我们将看到，在arpresolve中使用最后两个步骤是为了防止ARP洪泛。

415-420 如果ARP中保持有正在等待ARP解析该目的方硬件地址的mbuf，那么将mbuf送至接口输出函数(如图21-1所示)。由于该mbuf是由ARP保持的，即目的地址肯定是在以太网上，所以接口输出函数应该是ether_outout。该函数也调用arpresolve，但这时硬件地址已被填充，所以允许mbuf加入实际的设备输出队列。

9. 如果是ARP回答分组，则返回

421-426 如果该ARP操作不是请求，那么丢弃接收到的分组，并返回。

in_arpinput的剩下部分如图21-20所示，产生一个对应于ARP请求的回答。只有当以下两种情况时才会产生ARP回答：

1) 本机就是该请求所要查找的目的主机；

2) 本机是该请求所要查找的目的主机的ARP代理服务器。

函数执行到这个时刻，已经接收了 ARP请求，但ARP请求是广播发送的，所以目的主机可能是以太网上的任何主机。

10. 本机就是所要查找的目的主机

427-432 如果目的IP地址等于myaddr，那么本机就是所要查找的目的主机。将发送方硬件地址拷贝到目的硬件地址字段(发送方现在变成了目的主机)，arpcom结构中的接口以太网地址拷贝到源硬件地址字段。ARP回答中的其余部分在else语句后处理。

if_ether.c

```
427     if (itaddr.s_addr == myaddr.s_addr) {
428         /* I am the target */
429         bcopy((caddr_t) ea->arp_sha, (caddr_t) ea->arp_tha,
430               sizeof(ea->arp_sha));
431         bcopy((caddr_t) ac->ac_enaddr, (caddr_t) ea->arp_sha,
432               sizeof(ea->arp_sha));
433     } else {
434         la = arplookup(itaddr.s_addr, 0, SIN_PROXY);
435         if (la == NULL)
436             goto out;
437         rt = la->la_rt;
438         bcopy((caddr_t) ea->arp_sha, (caddr_t) ea->arp_tha,
439               sizeof(ea->arp_sha));
440         sdl = SDL(rt->rt_gateway);
441         bcopy(LLADDR(sdl), (caddr_t) ea->arp_sha, sizeof(ea->arp_sha));
442     }

443     bcopy((caddr_t) ea->arp_spa, (caddr_t) ea->arp_tpa, sizeof(ea->arp_spa));
444     bcopy((caddr_t) & itaddr, (caddr_t) ea->arp_spa, sizeof(ea->arp_spa));
445     ea->arp_op = htons(ARPOP_REPLY);
446     ea->arp_pro = htons(ETHERTYPE_IP);  /* let's be sure! */
447     eh = (struct ether_header *) sa.sa_data;
448     bcopy((caddr_t) ea->arp_tha, (caddr_t) eh->ether_dhost,
449           sizeof(eh->ether_dhost));
450     eh->ether_type = ETHERTYPE_ARP;
451     sa.sa_family = AF_UNSPEC;
452     sa.sa_len = sizeof(sa);
453     (*ac->ac_if.if_output) (&ac->ac_if, m, &sa, (struct rtentry *) 0);
454     return;
455 }
```

if_ether.c

图21-20 in_arpinput函数：形成ARP回答，并发送出去

11. 检测本机是否目的主机的ARP代理服务器

433-437 即使本机不是所要查找的目的主机，也可能被配置为目的主机的ARP代理服务器。再次调用arplookup函数，将第二个参数设为0，第三个参数设为SIN_PROXY，这将在路由表中查找SIN_PROXY标志为1的结点。如果查找不到(这是通常情况，本机收到了以太网上其他ARP请求的拷贝)，out处的代码将丢弃mbuf，并返回。

12. 产生代理回答

437-442 处理代理ARP请求时，发送方的硬件地址变成目的硬件地址，ARP结点中的以太网地址拷贝到发送方硬件地址。该ARP结点中的硬件地址可以是以太网中任一台主机的硬件地址，只要它可以向目的主机转发IP数据报。通常，提供代理ARP服务的主机会填入自己的硬件地址，当然这不是要求的。代理ARP结点是由系统管理员用arp命令带关键字pub创建的，标明目的IP地址(这是路由表项的关键值)和在ARP回答中返回的以太网地址。

13. 完成构造ARP回答分组

443-444 继续完成ARP回答分组的构建。发送方和目标的硬件地址已经填充好了，现在交换发送方和目标的IP地址。目的IP地址在itaddr中，如果发现以太网中有其他主机使用同一

IP地址，则该值已经被填充了(见图21-18)。

445-446 ARP操作码字段设为ARPOP_REPLY，协议地址类型设为ETHERTYPE_IP。旁边加了注释“你需要确定”，是因为当协议地址类型为ETHERTYPE_IPTRAILERS时arpintr也会调用该函数，但现在跟踪封装(trailer encapsulation)已不再使用了。

14. 用以太网帧首部填充sockaddr

447-452 sockaddr结构用14字节的以太网帧首部填充，如图21-12所示。目的硬件地址变成了以太网目的地址。

453-455 将ARP回答传送至接口输出函数，并返回。

21.9 ARP定时器函数

ARP结点一般是动态的——需要时创建，超时时自动删除。也允许管理员创建永久性结点，前面我们讨论的代理结点就是永久性的。回忆一下图21-1和图21-10中最后的#define语句，路由度量结构中的rmx_expire成员就是用作ARP定时器的。

21.9.1 arptimer函数

如图21-21所示，该函数每5分钟被调用一次。它查看所有ARP结点是否超时。

if_ether.c

```
74 static void
75 arptimer(ignored_arg)
76 void   *ignored_arg;
77 {
78     int     s = splnet();
79     struct llinfo_arp *la = llinfo_arp.la_next;

80     timeout(arptimer, (caddr_t) 0, arpt_prune * hz);
81     while (la != &llinfo_arp) {
82         struct rtentry *rt = la->la_rt;
83         la = la->la_next;
84         if (rt->rt_expire && rt->rt_expire <= time.tv_sec)
85             arptfree(la->la_prev);  /* timer has expired, clear */
86     }
87     splx(s);
88 }
```

if_ether.c

图21-21 arptimer 函数：每5分钟查看所有ARP定时器

1. 设置下一个时限

80 arp_rtrequest函数使arptimer函数第一次被调用，随后arptimer每隔5分钟(arpt_prune)使自己被调用一次。

2. 查看所有ARP结点

81-86 查看ARP结点链表中的每一个结点。如果定时器值是非零的(不是一个永久结点)，而且时间已经超时，那么arptfree就删除该结点。如果rt_expire是非零的，它的值是从结点超时起到现在的秒数。

21.9.2 arptfree函数

如图21-22所示，arptfree函数由arptimer函数调用，用于从链接llinfo_dl表项的

列表中删除一个超时的ARP结点。

1. 使正在使用的结点无效(不删除)

467-473 如果路由表引用计数器值大于0，而且rt_gatewary成员指向一个sockaddr_dl结构，则arptfree执行以下步骤：

1) 将链路层地址长度设为0；

2) 将la_asked计数器值设为0；

3) 清除RTF_REJECT标志。

随后函数返回。因为路由表引用计数器值非零，所以该路由结点不能删除。但是将sdl_alen值设为0，该结点也就无效了。下次要使用该结点时，还将产生一个ARP请求。

if_ether.c

```
459 static void
460 arptfree(la)
461 struct llinfo_arp *la;
462 {
463     struct rtentry *rt = la->la_rt;
464     struct sockaddr_dl *sdl;
465     if (rt == 0)
466         panic("arptfree");
467     if (rt->rt_refcnt > 0 && (sdl = SDL(rt->rt_gateway)) &&
468         sdl->sdl_family == AF_LINK) {
469         sdl->sdl_alen = 0;
470         la->la_asked = 0;
471         rt->rt_flags &= ~RTF_REJECT;
472         return;
473     }
474     rtrequest(RTM_DELETE, rt_key(rt), (struct sockaddr *) 0, rt_mask(rt),
475               0, (struct rtentry **) 0);
476 }
```

if_ether.c

图21-22 arptfree 函数：删除或使一个ARP结点无效

2. 删除没有被引用的结点

474-475 rtrequest删除路由结点，在 21.13节中，我们将看到它调用了 arp_rtrequest。arp_rtrequest函数释放所有该ARP结点保持的mbuf(由la_hold指针所指向)，并删除相应的llinfo_arp结点。

21.10 arpresolve函数

在图4-16中，ether_output函数调用arpresolve函数以获得对应某个IP地址的以太网地址。如果已知该以太网地址，则 arpreslove返回值为1，允许将待发IP数据报挂在接口输出队列上。如果不知道该以太网地址，则 arpresolve返回值为0，arpsolve函数利用llinfo_arp结构的la_hold成员指针“保持(held)”待发IP数据报，并发送一个ARP请求。收到ARP回答后，再将保持的IP数据报发送出去。

arpresolve应避免ARP洪泛，也就是说，它不应在尚未收到 ARP回答时高速重复发送ARP请求。出现这种情况主要有两个原因，第一，有多个 IP数据报要发往同一个尚未解析硬件地址的主机；第二，一个 IP数据报的每个分片都会作为独立分组调用 ether_output。11.9节讨论了一个由分片引起的 ARP洪泛的例子及相关的问题。图 21-23显示了arpresolve的前半部分。

252-261 dst是一个指向sockaddr_in的指针，它包含目的IP地址和对应的以太网地址(一个6字节的数组)。

if_ether.c

```
252 int
253 arpresolve(ac, rt, m, dst, desten)
254 struct arpcom *ac;
255 struct rtentry *rt;
256 struct mbuf *m;
257 struct sockaddr *dst;
258 u_char *desten;
259 {
260     struct llinfo_arp *la;
261     struct sockaddr_dl *sdl;

262     if (m->m_flags & M_BCAST) { /* broadcast */
263         bcopy((caddr_t) etherbroadcastaddr, (caddr_t) desten,
264               sizeof(etherbroadcastaddr));
265         return (1);
266     }
267     if (m->m_flags & M_MCAST) { /* multicast */
268         ETHER_MAP_IP_MULTICAST(&SIN(dst)->sin_addr, desten);
269         return (1);
270     }
271     if (rt)
272         la = (struct llinfo_arp *) rt->rt_llinfo;
273     else {
274         if (la = arplookup(SIN(dst)->sin_addr.s_addr, 1, 0))
275             rt = la->la_rt;
276     }
277     if (la == 0 || rt == 0) {
278         log(LOG_DEBUG, "arpresolve: can't allocate llinfo");
279         m_freem(m);
280         return (0);
281     }
```

if_ether.c

图21-23 arpresolve 函数：查找所需的ARP结点

1. 处理广播和多播地址

262-270 如果mbuf的M_BCAST标志置位，则用以太网广播地址填充目的硬件地址字段，函数返回1。如果M_MCAST标志置位，则宏ETHER_MAP_IP_MULTICAST(图12-6)将D类地址映射为相应的以太网地址。

2. 得到指向llinfo_arp结构的指针

271-276 目的地址是单播地址。如果调用者传输了一个指向路由表结点的指针，则将la设置为相应的llinfo_arp结构。否则，arplookup根据给定IP的地址搜索路由表。第二个参数是1，告诉arplookup如果搜索不到相应的ARP结点就创建一个新的；第三个参数是0，即意味着不去查找代理ARP结点。

277-281 如果rt或la中有一个是空指针，说明刚才请求分配内存时失败，因为即使不存在已有结点，arplookup也已经创建了一个，rt和la都不应是空值。记录一个差错报文，释放分组，函数返回0。

图21-24显示了arpresolve的后半部分。它检查ARP结点是否有效，如无效，则发送一个ARP请求。

if_ether.c

```
282     sdl = SDL(rt->rt_gateway);
283     /*
284      * Check the address family and length is valid, the address
285      * is resolved; otherwise, try to resolve.
286      */
287     if ((rt->rt_expire == 0 || rt->rt_expire > time.tv_sec) &&
288         sdl->sdl_family == AF_LINK && sdl->sdl_alen != 0) {
289         bcopy(LLADDR(sdl), desten, sdl->sdl_alen);
290         return 1;
291     }
292     /*
293      * There is an arptab entry, but no ethernet address
294      * response yet.  Replace the held mbuf with this
295      * latest one.
296      */
297     if (la->la_hold)
298         m_freem(la->la_hold);
299     la->la_hold = m;

300     if (rt->rt_expire) {
301         rt->rt_flags &= ~RTF_REJECT;
302         if (la->la_asked == 0 || rt->rt_expire != time.tv_sec) {
303             rt->rt_expire = time.tv_sec;
304             if (la->la_asked++ < arp_maxtries)
305                 arpwhohas(ac, &(SIN(dst)->sin_addr));
306             else {
307                 rt->rt_flags |= RTF_REJECT;
308                 rt->rt_expire += arpt_down;
309                 la->la_asked = 0;
310             }
311         }
312     }
313     return (0);
314 }
```

if_ether.c

图21-24 arpresolve 函数：检查ARP结点是否有效，如无效，则发送一个ARP请求

3. 检查ARP结点的有效性

282-291 即使找到了一个ARP结点，还需检查其有效性。如以下条件成立，则ARP结点是有效的：

1) 结点是永久有效的(时限值为0)，或尚未超时；

2) 由rt_gateway指向的插口地址结构的sdl_family字段为AF_LINK；

3) 链路层地址长度值(sdl_alen)不等于0。

arptfree使一个仍被引用的ARP结点失效的方法是将sdl_alen值置0。如果结点是有效的，则将sockaddr_dl中的以太网地址拷贝到desten，函数返回1。

4. 只保持最近的IP数据报

292-299 此时，已经有了ARP结点，但它没有一个有效的以太网地址，因此，必须发送一个ARP请求。将la_hold指针指向mbuf，同时也就释放了刚才la_hold所指的内容。这意味着，在发送ARP请求到收到ARP回答之间，如果有多个发往同一目的地的IP数据报要发送，只有最近的一个IP数据报才被la_hold保留，之前的全部丢弃。NFS就是这样的一个例子，如果NFS要传送一个8500字节的IP数据报，需要将其分割成6个分片。如果每个分片都在发送ARP请求到收到ARP回答之间由ip_output送往ether_output，那么前5个分片将被丢弃，

当收到ARP回答时，只有最后一个分片被保留了下来。这会使 NFS超时，并重发这6个分片。

5. 发送ARP请求，但避免ARP洪泛

300-314 RFC 1122要求ARP避免在收到ARP回答之前以过高的速度对一个以太网地址重发ARP请求。Net/3采用以下方法来避免ARP洪泛：

- Net/3不在同一秒钟内发送多个对应同一目的地的 ARP请求；
- 如果在连续5个ARP请求(也就是5秒钟)后还没有收到回答，路由结点的 RTF_REJECT标志置1，时限设为往后的20秒。这会使ether_output在20秒内拒绝发往该目的地址的IP数据报，并返回EHOSTDOWN或EHOSTUNREACH(如图4-15所示)。
- 20秒钟后，arpresolve会继续发送该目的主机的ARP请求。

如果时限值不等于0(非永久性结点)，则清除RTF_REJECT标志，该标志是在早些时候为避免ARP洪泛而设置的。计数器 la_asked记录的是连续发往该目的地址的 ARP请求数。如果计数器值为0或时限值不等于当前时钟 (只需看一下当前时钟的秒钟部分)，那么需要再发送一个ARP请求。这就避免了在同一秒钟内发送多个 ARP请求。然后将时限值设为当前时钟的秒钟部分(也就是微秒部分，time_tv_usec被忽略)。

将la_asked所含计数器值与限定值5(arp_maxtries)比较，然后加1。如果小于5，则arpwhohas发送ARP请求；如果等于5，则ARP已经达到了限定值：将RTF_REJECT标志置1，时限值置为往后的20秒钟，la_asked计数器值复位为0。

图21-25显示了一个例子，进一步解释了 arpresolve和ether_output为了避免ARP洪泛所采用的算法。

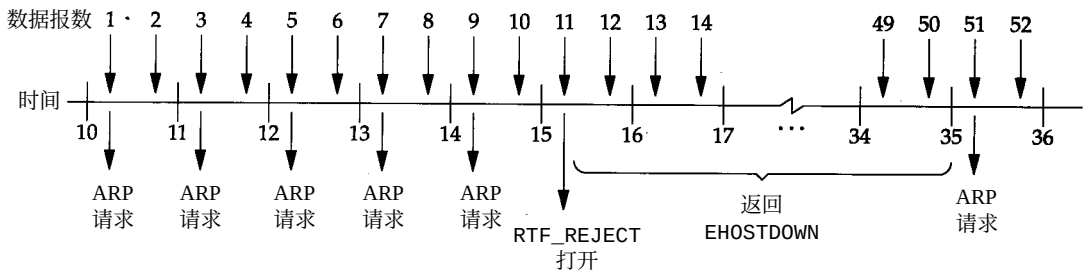


图21-25 避免ARP洪泛所采用的算法

图中总共显示了26秒的时间，从10到36。我们假定有一个进程每隔 0.5秒发送一个IP数据报，也就是说，一秒钟内有两个数据报等待发送。数据报依次被标号为 1~52。我们还假定目的主机已经关闭，所以收不到ARP回答。ARP将采取以下行动：

- 假定当进程写数据报1时la_asked的值为0。la_hold设为指向数据报1，rt_expire值设为当前时钟(10)，la_asked值变为1，发送ARP请求。函数返回0。
- 进程写数据报2时，丢弃数据报1，la_hold指向数据报2。由于rt_expire值等于当前时钟(10)，所以不发送ARP请求，函数返回，返回值为0。
- 进程写数据报3时，丢弃数据报2，la_hold指向数据报3。由于当前时钟(11)不等于rt_expire(10)，所以将rt_expire设为11。la_asked值为1，小于5，所以发送ARP请求，并将la_asked值置为2。
- 进程写数据报4时，丢弃数据报3，la_hold指向数据报4。由于rt_expire值等于当

前时钟(11)，所以无须其他动作，函数返回0。

- 对于数据报5~10，情况都是一样的。在数据报9到达后，发送ARP请求后，la_asked值被设为5；
- 进程写数据报11时，丢弃数据报10，la_hold指向数据报11。当前时钟(15)不等于rt_expire(14)，所以将rt_expire的值设为15。此时la_asked的值不再小于5，ARP避免洪泛的算法开始作用：RTF_REJECT标志位置1，rt_expire的值被设为35(即往后20秒)，la_asked的值设为0，函数返回0。
- 进程写数据报12时，ether_output注意到RTF_REJECT标志位为1，而且当前时钟小于rt_expire(35)，因此，返回EHOSTDOWN给发送者(通常是ip_output)。
- 从数据报13到50，都返回EHOSTDOWN给发送者。
- 当进程写数据报51时，尽管此时的RTF_REJECT标志位仍然为1，但当前时钟的值(35)不再小于rt_expire(35)，因此不会返回出错信息。调用arpresolve，整个过程重新开始，5秒钟内发送5个ARP请求，然后是20秒钟的等待，直到发送者放弃或目的主机响应ARP请求。

21.11 arplookup函数

arplookup函数调用选路函数rtallocl在Internet路由表中查找ARP结点。我们已经看到过3次调用arplookup的情况：

1) 在in_arpinput中，在接收到ARP分组后，对应源IP地址查找或创建一个ARP结点。

2) 在in_arpinput中，接收到ARP请求后，查看是否存在目的硬件地址的代理ARP结点。

3) 在arpresolve中，查找或创建一个对应待发送数据报IP地址的ARP结点。

如果arplookup执行成功，则返回一个指向相应llinfo_arp结构的指针，否则返回一个空指针。

arplookup带有三个参数，第一个参数是目的IP地址；第二个参数是个标志，为真时表示若找不到相应结点就创建一个新的结点；第三个参数也是一个标志，为真时表示查找或创建代理ARP结点。

代理ARP结点通过定义一个不同形式的Internet插口地址结构来处理，即sockaddr_inarp结构，如图21-26所示。该结构只在ARP中使用。

if_ether.h

```
111 struct sockaddr_inarp {
112     u_char  sin_len;            /* sizeof(struct sockaddr_inarp) = 16 */
113     u_char  sin_family;         /* AF_INET */
114     u_short sin_port;
115     struct in_addr sin_addr;    /* IP address */
116     struct in_addr sin_srcaddr; /* not used */
117     u_short sin_tos;            /* not used */
118     u_short sin_other;          /* 0 or SIN_PROXY */
119 };
```

if_ether.h

图21-26 sockaddr_inarp 结构

111-119 前面8个字节与sockaddr_in结构相同，sin_family被设为AF_INET。最后8

个字节有所不同：sin_srcaddr、sin_tos和sin_other成员。当结点作为代理结点时，只用到sin_other成员，并将其设为SIN_PROXY(1)。

图21-27显示了arplookup函数。

if_ether.c

```
480 static struct llinfo_arp *
481 arplookup(addr, create, proxy)
482 u_long  addr;
483 int     create, proxy;
484 {
485     struct rtentry *rt;
486     static struct sockaddr_inarp sin =
487     {sizeof(sin), AF_INET};
488     sin.sin_addr.s_addr = addr;
489     sin.sin_other = proxy ? SIN_PROXY : 0;
490     rt = rtalloc1((struct sockaddr *) &sin, create);
491     if (rt == 0)
492         return (0);
493     rt->rt_refcnt--;
494     if ((rt->rt_flags & RTF_GATEWAY) || (rt->rt_flags & RTF_LLINFO) == 0 ||
495         rt->rt_gateway->sa_family != AF_LINK) {
496         if (create)
497             log(LOG_DEBUG, "arptnew failed on %x\n", ntohl(addr));
498         return (0);
499     }
500     return ((struct llinfo_arp *) rt->rt_llinfo);
501 }
```

if_ether.c

图21-27 arplookup 函数：在路由表中查找ARP结点

1. 初始化sockaddr_inarp结构，准备查找

480-489 sin_addr成员设为将要查找的IP地址。如果proxy参数值不为0，则sin_other成员设为SIN_PROXY；否则设为0。

2. 路由表中查找结点

490-492 rtallocl在Internet路由表中查找IP地址，如果create参数值不为0，就创建一个新的结点。如果找不到结点，则函数返回值为0(空指针)。

3. 减少路由表结点的引用计数值

493 如果找到了结点，则减少路由表结点的引用计数。因为，此时ARP不再被认为像运输层一样“持有”路由表结点，因此，路由表查找时对rt_refcnt计数的递增，应在这里由ARP取消。

494-499 如果将标志RTF_GATEWAY置位，或者标志RTF_LLINFO没有置位，或者由rt_gateway指向的插口地址结构的地址族字段值不是AF_LINK，说明出了某些差错，返回一个空指针。如果结点是这样创建的，应创建一个记录报文。

记录报文中对arptnew的注释是针对老版本Net/2中创建ARP结点的。

如果rtallocl由于匹配结点的RTF_CLONING标志置位而创建一个新的结点，那么函数arp_rtrequest (21.13节)也要被rtrequest调用。

21.12 代理ARP

Net/3支持代理ARP，有两种不同类型的代理ARP结点，可以通过arp命令及pub选项将

它们加入到路由表中。添加代理ARP选项会使arp_rtrequest主动发送动态联编信息(如图21-28所示)，因为在创建结点时RTF_ANNOUNCE标志位被置1。

代理ARP结点的第一种类型：它允许将网络内的某一主机的IP地址填入到ARP高速缓存内。硬件地址可以设为任意值。这种结点加入到路由表中时使用了直接的掩码0xffffffff。加掩码的目的是即使插口地址的SIN_PROXY标志位为1，在调用图21-27中的rtallocl时能与该结点匹配。于是在调用图21-20中的arplookup时也能与该结点匹配，目的地址的SIN_PROXY置位。

如果本网中的主机H1不能实现ARP，那么可以使用这种类型的代理ARP结点。作为代理的主机代替H1回答所有的ARP请求，同时提供创建代理ARP结点时设定的硬件地址(比如可以是H1的以太网地址)。这种类型的结点可以通过arp -a命令查看，它带有“published”符号。

第二种类型的代理ARP结点用于已经存有路由表结点的主机。内核为该目的地址创建另外一个路由表结点，在这个新的结点中含有链路层的信息(如以太网地址)。该新结点中sockaddr_inarp结构(图21-26)的sin_other成员的SIN_PROXY标志置位。回想一下，搜索路由表时是比较12字节的Internet插口地址(图18-39)。只有当该结构的最后8字节非零时，才会用到SIX_PROXY标志位。当arplookup指定送往rtallocl的结构中的sin_other成员中SIN_PROXY的值时，只有路由表中那些匹配的结点的SIN_PROXY标志置位。

这种类型的代理ARP结点通常指明了作为代理ARP服务器的以太网地址。如果某代理ARP结点是为主机HD创建的，一般有以下步骤：

1) 代理服务器收到来自主机HS的查找HD硬件地址的广播ARP请求，主机HS认为HD在本地网上；

2) 代理服务器回答请求，并提供本机的以太网地址；

3) HS将发往HD的数据报发送给代理服务器；

4) 收到发往HD的数据报后，代理服务器利用路由表中关于HD的信息将数据报转发给HD。

路由器netb使用这种类型的代理ARP结点，见卷1 4.6节中的例子。可以通过命令arp -a来查看这些带有“published (proxy only)”的结点。

21.13 arp_rtrequest函数

图21-3简要显示了ARP函数和选路函数之间的关系。在ARP中，我们将调用两个路由表函数：

1) arplookup调用rtallocl查找ARP结点，如果找不到匹配结点，则创建一个新的ARP结点。

如果在路由表中找到了匹配结点，且该结点的RTF_CLONING标志位没有置位(即该结点就是目的主机的结点)，则返回该结点。如果RTF_CLONING标志位被置位，rtallocl以RTM_RESOLVE命令为参数调用rtrequest。图18-2中的140.252.13.33和140.252.13.34结点就是这么创建的，它们是从140.252.13.32的结点复制而来的。

2) arptfree以RTM_DELETE命令为参数调用rtrequest，删除对应ARP结点的路由表结点。

此外，arp命令通过发送和接收路由插口上的路由报文来操纵ARP高速缓存。arp以命令

RTM_RESOLVE、RTM_DELETE和RT_GET为参数发布路由信息。前两个参数用于调用rtrequest，第三个参数用于调用rtallocl。

最后，当以太网设备驱动程序获得了赋予该接口的IP地址后，rtinit增加一个网络路由。于是rtrequest函数被调用，参数是RTM_ADD，标志位是RTF_UP和RTF_CLONING。图18-2中140 252 13 32结点就是这么创建的。

在第19章中我们讲过，每一个ifaddr结构都有一个指向函数(ifa_rtrequest成员)的指针，该函数在创建或删除一个路由表结点时被自动调用。在图6-17中，对于所有以太网设备，in_ifinit将该指针指向arp_rtrequest函数。因此，当调用路由函数为ARP创建或删除路由表结点时，总会调用arp_rtrequest。当任意路由表函数被调用时，arp_rtrequest函数的作用是做各种初始化或退出处理所需的工作。例如：当创建新的ARP结点时，arp_rtrequest内要为llinfo_arp结构分配内存。同样，当路由函数处理完一个RTM_DELETE命令后，arp_rtrequest的工作是删除llinfo_arp结构。

图21-28显示了arp_rtrequest函数的第一部分。

if_ether.c

```
 92 void
 93 arp_rtrequest(req, rt, sa)
 94 int     req;
 95 struct rtentry *rt;
 96 struct sockaddr *sa;
 97 {
 98     struct sockaddr *gate = rt->rt_gateway;
 99     struct llinfo_arp *la = (struct llinfo_arp *) rt->rt_llinfo;
100     static struct sockaddr_dl null_sdl =
101     {sizeof(null_sdl), AF_LINK};

102     if (!arpinit_done) {
103         arpinit_done = 1;
104         timeout(arptimer, (caddr_t) 0, hz);
105     }
106     if (rt->rt_flags & RTF_GATEWAY)
107         return;
108     switch (req) {

109     case RTM_ADD:
110         /*
111          * XXX: If this is a manually added route to interface
112          * such as older version of routed or gated might provide,
113          * restore cloning bit.
114          */
115         if ((rt->rt_flags & RTF_HOST) == 0 &&
116             SIN(rt_mask(rt))->sin_addr.s_addr != 0xffffffff)
117             rt->rt_flags |= RTF_CLONING;
118         if (rt->rt_flags & RTF_CLONING) {
119             /*
120              * Case 1: This route should come from a route to iface.
121              */
122             rt_setgate(rt, rt_key(rt),
123                        (struct sockaddr *) &null_sdl);
124             gate = rt->rt_gateway;
125             SDL(gate)->sdl_type = rt->rt_ifp->if_type;
126             SDL(gate)->sdl_index = rt->rt_ifp->if_index;
127             rt->rt_expire = time.tv_sec;
```

图21-28 arp_rtrequest 函数：RTM_ADD 命令

```
128             break;
129         }
130         /* Announce a new entry if requested. */
131         if (rt->rt_flags & RTF_ANNOUNCE)
132             arprequest((struct arpcom *) rt->rt_ifp,
133                        &SIN(rt_key(rt))->sin_addr.s_addr,
134                        &SIN(rt_key(rt))->sin_addr.s_addr,
135                        (u_char *) LLADDR(SDL(gate)));
136         /* FALLTHROUGH */
```

if_ether.c

图21-28 (续)

1. 初始化ARP timeou函数

92-105 第一次调用arp_rtrequest函数时(系统初始化阶段，在对第一个以太网接口赋IP地址时)，timeout函数在一个时钟滴答内调用arptimer函数。此后，ARP定时器代码每5分钟运行一次，因为arptimer总是要调用timeout的。

2. 忽略间接路由

106-107 如果将标志RTF_GATEWAY置位，则函数返回。RTF_GATEWAY标志表明该路由表结点是间接的，而所有ARP结点都是直接的。

108 一个带有三种可能的switch语句：RTM_ADD、RTM_RESOLVE和RTM_DELETE(后两种在后面的图中显示)。

3. RTM_ADD命令

109 RTM_ADD命令出现在以下两种情况中：执行arp命令手工创建ARP结点或者rtinit函数对以太网接口赋IP地址(图21-3)。

4. 向后兼容

110-117 若标志RTF_HOST没有置位，说明该路由表结点与一个掩码相关(也就是说是网络路由，而非主机路由)。如果掩码不是全1，那么该结点确实是某一接口的路由，因此，将标志RTF_CLONING置位。如注释中所述，这是为了与某些旧版本的路由守护程序兼容。此外，/etc/netstart中的命令：

```
route add -net 224.0.0.0 -interface bsdi
```

为图18-2所示网络创建带有RTF_CLONING标志的路由表结点。

5. 初始化到接口的网络路由结点

118-126 若标志RTF_CLONING(in_ifinit为所有以太网接口设置该标志)置位，那么该路由表结点是由rtinit添加的。rt_setgate为sockaddr_dl结构分配空间，该结构由rt_gateway指针所指。与图21-1中140.252.13.32的路由表结点相关的就是该数据链路插口地址结构。sdl_family和sdl_len成员的值是根据静态定义的null_sd而初始化的，sdl_type(可能是IFT_ETHER)和sdl_index成员的值来自接口的ifnet结构。该结构不包含以太网地址，sdl_alen成员的值为0。

127-128 最后将时限值设为当前时间，也就是结点的创建时间，执行break后返回。对于在系统初始化时创建的结点，它们的rmx_expire值为系统启动的时间。注意，图21-1中该路由表结点没有相应的llinfo_arp结构，所以它不会被arptimer处理。但是要用它的sockaddr_dl结构，对于以太网中特定主机的路由结点来说，要复制的是rt_gateway结构，用RTM_RESOLVE命令参数创建路由表结点时，rtrequest复制该结构。此外，

netstat程序将sdl_index的值输出为link#n，见图18-2。

6. 发送免费ARP请求

130-135 若将标志RTF_ANNOUNCE置位，则该结点是由arp命令带pub选项创建的。该选项有两个分支：(1) sockaddr_inarp结构中sin_other成员的SIN_PROXY标志被置位；(2)标志RTF_ANNOUNCE被置位。因为标志RTF_ANNOUNCE被置位，所以arprequest广播免费ARP请求。注意，第二个和第三个参数是相同的，即该 ARP请求中，发送方IP地址和目的方IP地址是一样的。

136 继续执行针对RTM_RESOLVE命令的case语句。

图21-29显示了arp_rtrequest函数的第二部分，处理 RTM_RESOLVE命令。当rtallocl找到一个RTF_CLONING标志位置位的路由表结点且 rtallocl的第二个参数值(arplookup的create参数)不为0时，调用该命令。需要分配一个新的 llinfo_arp结构，并将其初始化。

if_ether.c

```
137     case RTM_RESOLVE:
138         if (gate->sa_family != AF_LINK ||
139             gate->sa_len < sizeof(null_sdl)) {
140             log(LOG_DEBUG, "arp_rtrequest: bad gateway value");
141             break;
142         }
143         SDL(gate)->sdl_type = rt->rt_ifp->if_type;
144         SDL(gate)->sdl_index = rt->rt_ifp->if_index;
145         if (la != 0)
146             break;              /* This happens on a route change */
147         /*
148          * Case 2:  This route may come from cloning, or a manual route
149          * add with a LL address.
150          */
151         R_Malloc(la, struct llinfo_arp *, sizeof(*la));
152         rt->rt_llinfo = (caddr_t) la;
153         if (la == 0) {
154             log(LOG_DEBUG, "arp_rtrequest: malloc failed\n");
155             break;
156         }
157         arp_inuse++, arp_allocated++;
158         Bzero(la, sizeof(*la));

159         la->la_rt = rt;
160         rt->rt_flags |= RTF_LLINFO;
161         insque(la, &llinfo_arp);

162         if (SIN(rt_key(rt))->sin_addr.s_addr ==
163             (IA_SIN(rt->rt_ifa))->sin_addr.s_addr) {
164             /*
165              * This test used to be
166              *  if (loif.if_flags & IFF_UP)
167              * It allowed local traffic to be forced
168              * through the hardware by configuring the loopback down.
169              * However, it causes problems during network configuration
170              * for boards that can't receive packets they send.
171              * It is now necessary to clear "useloopback" and remove
172              * the route to force traffic out to the hardware.
173              */
174             rt->rt_expire = 0;
```

图21-29 arp_rtrequest 函数：RTM_RESOLVE 命令

```
175             Bcopy(((struct arpcom *) rt->rt_ifp)->ac_enaddr,
176                   LLADDR(SDL(gate)), SDL(gate)->sdl_alen = 6);
177             if (useloopback)
178                 rt->rt_ifp = &loif;

179         }
180         break;
```

if_ether.c

图21-29 (续)

7. 验证sockaddr_dl结构

137-144 验证rt_gateway指针所指的sockaddr_dl结构的sa_family和sa_len成员的值。接口类型(可能是IFT_ETHER)和索引值填入新的sockaddr_dl结构。

8. 处理路由变化

145-146 正常情况下，该路由表结点是新创建的，并没有指向一个 llinfo_arp结构。如果la指针非空，则在路由已发生了变化时调用 arp_rtrequest。此时llinfo_arp已经分配，执行break，函数返回。

9. 初始化llinfo_arp结构

147-158 分配一个llinfo_arp结构，rt_llinfo中存有指向该结构的指针。统计值变量arp_inuse和arp_allocated各加1，llinfo_arp结构置0。将la_hold指针置空，la_asked值置0。

159-161 将rt指针存储于llinfo_arp结构中，置RTF_LLINFO标志位。如图18-2所示，ARP创建的三个结点140.252.13.33、140.252.13.34和140.252.13.35都有L标志，和240.0.0.1一样。arp程序只检查该标志(图19-36)。最后insque将llinfo_arp加入到链接表的首部。

就这样创建了一个ARP结点：rtrequest创建路由表结点(经常为以太网克隆一个特定网络的结点)，arp_rtrequest分配和初始化llinfo_arp结构。剩下只需广播一个ARP请求，在收到回答后填充主机的以太网地址。事件发生的一般次序是： arpresolve调用arplookup，于是arp_rtrequest被调用(中间可能跟有函数调用，见图 21-3)。当控制返回到arpresolve时，发送ARP广播请求。

10. 处理发给本机的特例情况

162-173 这是4.4BSD新增的测试特例部分(注释是老版本留下的)。它创建了图21-1中最右边的路由表结点，该结点包含了本机的IP地址(140.252.13.35)。if语句检测它是否等于本机IP地址，如等于，那么这个刚创建的结点代表的是本机。

11. 将结点置为永久性，并设置以太网地址

174-176 时限值设为0，意味着该结点是永久有效的——永远不会超时。从接口的arpcom结构中将硬件地址拷贝至rt_gateway所指的sockaddr_dl结构中。

12. 将接口指针指向环回接口

177-178 若全局变量usrloopback值不为0(默认为1)，则将路由表结点内的接口指针指向环回接口。这意味着，如果有数据报发给自己，就送往环回接口。在 4.4BSD以前的版本中，可以通过/etc/netstart文件中的命令：

```
route add 140.252.13.35 127.0.0.1
```

来建立从本机IP地址到环回接口的路由。4.4BSD仍然支持这种方式，但已不是必需的了。当

第一次有数据报发给本机IP地址时，我们刚才看到的代码会自动创建一个这样的路由。此外，这些代码对于一个接口只会执行一次。一旦路由表结点和永久性 ARP结点创建好后，它们就不会超时，所以不会再次出现对本机IP地址的RTM_RESOLLVE命令。

arp_rtrequest函数的最后部分如图21-30所示，处理RTM_DELETE请求。从图21-3中，我们可以看到，该命令是由 arp命令产生的，用于手工删除一个结点；或者在一个 ARP结点超时时由arptfree产生。

if_ether.c

```
181     case RTM_DELETE:
182         if (la == 0)
183             break;
184         arp_inuse--;
185         remque(la);
186         rt->rt_llinfo = 0;
187         rt->rt_flags &= ~RTF_LLINFO;
188         if (la->la_hold)
189             m_freem(la->la_hold);
190         Free((caddr_t) la);
191     }
192 }
```

if_ether.c

图21-30 arp_rtrequest函数：RTM_DELETE 命令

13. 验证la指针

182-183 la指针应该是非空的，也就是说路由表结点必须指向一个 llinfo_arp结构；否则，执行break，函数返回。

14. 删除llinfo_arp结构

184-190 统计值变量 arp_inuse减1，remque从链表中删除 llinfo_arp结构。rt_llinfo指针置0，清除RTF_LLINFO标志。如果该ARP结点保持有mbuf(即该ARP请求未收到回答)，则将mbuf释放。最后释放llinfo_arp结构。

注意，switch语句中没有包含default情况，也没有考虑 RTM_GET命令。这是因为arp程序产生的RTM_GET命令全部由route_output函数处理，并不调用 rtrequest。此外，见图21-3，在RTM_GET命令产生的对 rtallocl调用中，指定第二个参数是 0，所以rtallocl并不调用rtrequest。

21.14 ARP和多播

如果一个IP数据报要采用多播方式发送，ip_output检测进程是否已将某个特定的接口赋予插口(见图12-40)。如果已经赋值，则将数据报发往该接口，否则，ip_output利用路由表选择输出接口(见图8-24)。因此，对于具有多个多播发送接口的系统来说，IP路由表应指定每个多播组的默认接口。

在图18-2中我们看到，路由表中有一个结点是为网络224.0.0.0创建的，该结点具有“flag”标志。所有以224开头的多播组都以该结点指定的接口 (le0)为默认接口。对于其他的多播组(以225~239开头)，可以分别创建新的路由表结点，也可以对某个指定多播组创建一个路由表结点。例如，可以为224.0.11(网络定时协议)创建一个与224.0.0.0不同的路由表结点。如果路由表中没有对应某个多播组的结点，同时进程没有用 IP_MULTICAST_IF插口选项指明接口，那么该组的默认接口成为路由表中默认路由的接口。其实图 18-2中对应224.0.0.0的路由表结

点并不是必要的，因为默认接口就是le0。

如果选定的接口是以太网接口，则调用arpresolve将多播组地址映射为相应的以太网地址。在图21-23中，映射通过调用宏ETHER_MAP_IP_MULTICAST来完成。该宏所做的就是将该多播组地址的低23位与一个常量逻辑或(图12-6)，映射不需要ARP请求和回答，也不需要进入ARP高速缓存。每次需要映射时，调用该宏。

如果多播组是从另外一个结点复制得来的，那么多播组地址会出现在ARP缓存里，如图21-5所示。因为这些结点将RTF_LLINFO标志置位。它们不会有ARP请求和回答，所以说不是真正的ARP结点。它们也没有相应的链路层地址，宏ETHER_MAP_IP_MULTICAST就可以完成映射。

这些多播组的ARP结点的时效与正常的ARP结点不同。在为某个多播组创建一个路由表结点时，如图18-2中的224.0.0.1，rtrequest从被克隆的结点中复制rt_metrics结构(图19-9)。图21-28中，网络路由结点的rmx_expire值被设为RTM_ADD命令执行的时间，也即系统初始化的时间。为224.0.0.1设置的结点也设置为同样的时间。

这就意味着在下次arptimer执行时，对应多播组224.0.0.1的ARP结点总是超时的。所以，当下一次在路由表中查找时就需重新创建该结点。

21.15 小结

ARP提供了IP地址到硬件地址的映射，本章讲述了如何实现这种映射。

Net/3实现与以往的BSD版本有很大不同。ARP信息被存放在多个结构里面：路由表、数据链路插口地址结构和llinfo_arp结构。图21-1显示了这些结构之间的关系。

发送一个ARP请求是很简单的：正确填充相关字段后，将请求广播发送出去就行了。处理请求就要复杂一些，因为每个主机都收到了广播的ARP请求。除了响应请求外，in_arpinput还要检测是否有其他主机正与它使用同一个IP地址。因为每一个ARP请求中包含发送方的IP和硬件地址，所以网络上的所有主机都可以通过它来更新自己的ARP结点。

在局域网中，ARP洪泛将是一个问题，Net/3是第一个考虑这种问题的BSD版本。对于同一个目的地，一秒钟内只可发送一个ARP请求，如果连续5个请求都没有收到回答，必须暂停20秒钟才可再发送去往该目的地的ARP请求。

习题

21.1 图21-17中给局部变量ac赋值时，做过什么假设？

21.2 如果我们先ping本地以太网的广播地址，之后执行arp -a，就可以发现几乎所有本地以太网上的其他主机的表项都填入到了ARP高速缓存中。这是为什么？

21.3 查看代码并解释为什么图21-19中需要把sdl_alen的值赋为6。

21.4 在Net/2中有一个独立于路由表而存在的ARP表，每次调用arpresolve时，都要在该ARP表中查找。试与Net/3的方法比较，哪个更有效？

21.5 Net/2中的ARP代码显式地设置ARP高速缓存中非完整表项的超时为3分钟，非完整表项是指正在等待ARP回答的表项。但我们从没有提过Net/3如何处理该超时，那么Net/3何时才认为非完整表项超时？

21.6 当Net/3系统作为一个路由器并且导致洪泛的分组来自其他主机时，为避免ARP洪

泛要做哪些变动？

21.7 图21-1中给出的四个rmx_expire变量的值是什么？代码在何处设置该值？

21.8 对广播ARP请求的每个主机，本章中引起要创建一个 ARP结点的代码需要做哪些变动？

21.9 为了验证图21-25中的例子，作者运行了卷 1附录C的sock程序，每隔500 ms向本地以太网上一个不存在的主机发送一个 UDP数据报(程序的 -p选项改为等待的毫秒数)。但是在返回第一个EHOSTDOWN差错之前，仅无差错地发送了10个UDP数据报，而不是图21-25所示的11个，这是为什么？

21.10 修改ARP，使得它在等待 ARP回答时持有到目的主机的所有分组，而不是持有最近的一个。如何实现这种改变？是否像每个接口的输出队列一样，需要一个限制？是否需要改变数据结构？

第22章　协议控制块

22.1　引言

协议层使用协议控制块(PCB)存放各UDP和TCP插口所要求的多个信息片。Internet协议维护Internet 协议控制块(Internet protocol control block)和TCP控制块(TCP control block)。因为UDP是无连接的，所以一个端结点需要的所有信息都可以在 Internet PCB中找到；不存在UDP控制块。

Internet PCB含有所有UDP和TCP端结点共有的信息：外部和本地 IP地址、外部和本地端号、IP首部原型、该端结点使用的IP选项以及一个指向该端结点目的地址选路表入口的指针。TCP控制块包含了TCP为各连接维护的所有结点信息：两个方向的序号、窗口大小、重传次数等等。

本章我们描述Net/3所用的Internet PCB，在详细讨论TCP时再探讨TCP控制块。我们将研究几个操作Internet PCB的函数，会在描述UDP和TCP时遇到它们。大多数的函数以 `in_pcb` 开头。

图22-1总结了协议控制块以及它们与 `file`和`socket`结构之间的关系。该图中有几点要考虑：

- 当`socket`或`accept`创建一个插口后，插口层生成一个 `file`结构和一个`socket`结构。文件类型是`DTYPE_SOCKET`，UDP端结点的插口类型是 `SOCK_DGRAM`，TCP端结点的插口类型是`SOCK_STREAM`。
- 然后调用协议层。UDP创建一个Internet PCB(一个`inpcb`结构)，并把它链接到`socket`结构上：`so_pcb`成员指向`inpcb`结构，`inp_socket`成员指向`socket`结构。
- TCP做同样的工作，也创建它自己的控制块 (一个`tcpcb`结构)，并用指针`inp_ppcb`和`t_inpcb`把它链接到`inpcb`上。在两个UDP `inpcb`中，`inp_ppcb`成员是一个空指针，因为UDP不负责维护它自己的控制块。
- 我们显示的其他四个`inpcb`结构的成员，从`inp_faddr`到`inp_lport`，形成了该端结点的插口对：外部IP地址和端口号，以及本地IP地址和端口号。
- UDP和TCP用指针`inp_next`和`inp_prev`维护一个所有Internet PCB的双向链表。它们在表头分配一个全局 `inpcb`结构(命名为`udb`和`tcb`)，在该结构中只使用三个成员：下一个和前一个指针，以及本地端口号。后一个成员中包含了该协议使用的下一个临时端口号。

Internet PCB是一个传输层数据结构。TCP、UDP和原始IP使用它，但IP、ICMP或ICMP不用它。

我们还没有讲过原始IP，但它也用Internet PCB。与TCP和UDP不同，原始IP在PCB中不用端口号成员，原始 IP只用本章中提到的两个函数： `in_pcballoc`分配PCB，`in_pcbdetach`释放PCB。第32章将讨论原始IP。

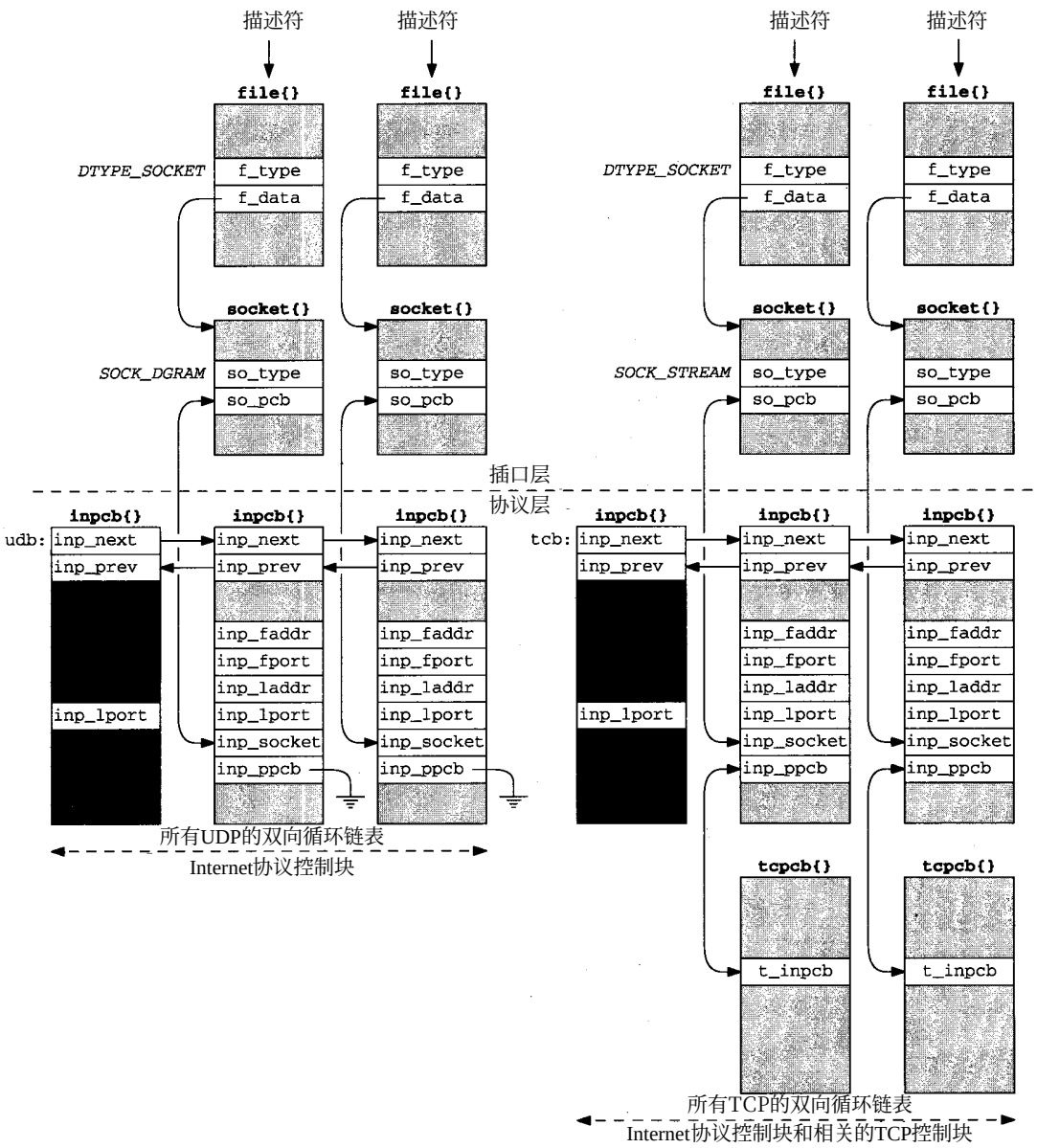


图22-1 Internet协议控制块以及与其他结构之间的关系

22.2 代码介绍

所有PCB函数都在一个C文件和一个包含定义的头文件中，如图 22-2所示。

文 件	描 述
netinet/in_pcb.h	in_pcb结构定义
netinet/in_pcb.c	PCB函数

图22-2 本章中讨论的文件

22.2.1 全局变量

本章只引入一个全局变量，如图22-3所示。

变 量	数据类型	描 述
zeroin_addr	struct in_addr	32 bit全零IP地址

图22-3 本章中引入的全局变量

22.2.2 统计量

Internet PCB和TCP PCB都是内核的malloc函数分配的M_PCB类型。这只是内核分配的大约60种不同类型内存的一种。例如，mbuf的类型是M_BUF，socket结构分配的类型是M_SOCKET。

因为内核保持所分配的不同类型内存缓存的计数器，所以维护着几个PCB数量的统计量。vmstat -m命令显示内核的内存分配统计信息，netstat -m命令显示的是mbuf分配统计信息。

22.3 inpcb的结构

图22-4是inpcb结构的定义。这不是一个大结构，只占84个字节。

in_pcb.h

```
42 struct inpcb {
43     struct inpcb *inp_next, *inp_prev;  /* doubly linked list */
44     struct inpcb *inp_head;     /* pointer back to chain of inpcb's for
45                                    this protocol */
46     struct in_addr inp_faddr;   /* foreign IP address */
47     u_short inp_fport;          /* foreign port# */
48     struct in_addr inp_laddr;   /* local IP address */
49     u_short inp_lport;          /* local port# */
50     struct socket *inp_socket;  /* back pointer to socket */
51     caddr_t inp_ppcb;           /* pointer to per-protocol PCB */
52     struct route inp_route;     /* placeholder for routing entry */
53     int     inp_flags;          /* generic IP/datagram flags */
54     struct ip inp_ip;           /* header prototype; should have more */
55     struct mbuf *inp_options;   /* IP options */
56     struct ip_moptions *inp_moptions;   /* IP multicast options */
57 };
```

in_pcb.h

图22-4 inpcb 结构

43-45 inp_next和inp_prev为UDP和TCP的所有PCB形成一个双向链表。另外，每个PCB都有一个指向协议链表表头的指针(inp_head)。对UDP表上的PCB，inp_head总是指向udb(图22-1)；对TCP表上的PCB，这个指针总是指向tcb。

46-49 下面四个成员：inp_faddr、inp_fport、inp_laddr和inp_lport，包含了这个IP端结点的插口对：外部IP地址和端口号，以及本地IP地址和端口号。PCB中以网络字节序而不是以主机字节序维护这四个值。

运输层的TCP和UDP都使用Internet PCB。尽管在这个结构里保存本地和外部IP地址很有意义，但端口号并不属于这里。端口号及其大小的定义是由各运输层协议

指定的，不同的运输层可以指定不同的值。[Partridge 1987] 提出了这个问题，其中版本1的RDP采用8 bit的端口号，需要用 8 bit的端口号重新实现几个标准内核程序。版本2的RDP [Partridge和Hinden 1990] 采用16 bit端口号。实际上，端口号属于运输层专用控制块，例如 TCP的tcpcb。可能会要求采用一种新的 UDP专用的PCB。尽管这个方案可行，但却可能使我们马上要讨论的几个程序复杂化。

50-51 inp_socket是一个指向该PCB的socket结构的指针，inp_ppcb是一个指针，它指向这个PCB的可选运输层专用控制块。我们在图 22-1中看到，inp_ppcb和TCP一起指向对应的tcpcb，但UDP不用它。socket和inpcb之间的链接是双向的，因为有时内核从插口层开始，需要对应的 Internet PCB(如用户输出)，而有时内核从 PCB开始，需要找到对应的socket结构(如处理收到的IP数据报)。

52 如果IP有一个到外部地址的路由，则它被保存在 ipp_route入口处。我们将看到，当收到一个ICMP重定向报文时，将扫描所有 Internet PCB，找到那些外部IP地址与重定向IP地址匹配的PCB，将其inp_route入口标记成无效。当再次将该 PCB用于输出时，迫使IP重新找一条到该外部地址的新路由。

53 inp_flags成员中存放了几个标志。图22-5显示了各标志。

inp_flags	描　述
INP_HDRINCL	进程提供整个IP首部(只有原始插口)
INP_RECVOPTS	把到达IP选项作为控制信息接收(只有UDP，还没有实现)
INP_RECVRETOPTS	把回答的IP选项作为控制信息接收(只有UDP，还没有实现)
INP_RECVDSTADDR	把IP目的地址作为控制信息接收(只有UDP)
INP_CONTROLOPTS	*INP_RECVOPTS|INP-RECVRETOPTS|INP_RECVDSTADDR*

图22-5 inp_flags 值

54 PCB中维护一个IP首部的备份，但它只使用其中的两个成员，TOS和TTL。TOS被初始化为0(普通业务)，TTL被运输层初始化。我们将看到，TCP和UDP都把TTL的默认值设为 64。进程可以用IP_TOS或IP_TTL插口选项改变这些默认值，新的值记录在 inpcb-inp_ip结构中。以后，TCP和UDP在发送IP数据报时，却把该结构用作原型 IP首部。

55-56 进程也可以用IP_OPTIONS插口选项设置外出数据报的 IP选项。函数ip_pcbopts把调用方选项的备份存放在一个 mbuf中，inp_options成员是一个指向该 mbuf的指针。每次TCP和UDP调用ip_output函数时，就把一个指向这些 IP首部的指针传给 IP，IP将其插到出去的IP数据报中。类似地，inp_moptions成员是一个指向用户 IP多播选项备份的指针。

22.4 in_pcballoc和in_pcbdetach函数

在创建插口时，TCP、UDP和原始IP会分配一个 Internet PCB。系统调用 socket发布PRU_ATTACH请求。在UDP情况下，我们将在图23-33中看到，产生的调用是

```
struct socket *so;
int error;

error = in_pcballoc(so, &udb);
```

图22-6是in_pcballoc函数。

1. 分配PCB，初始化为零

in_pcb.c

```
36 int
37 in_pcballoc(so, head)
38 struct socket *so;
39 struct inpcb *head;
40 {
41     struct inpcb *inp;

42     MALLOC(inp, struct inpcb *, sizeof(*inp), M_PCB, M_WAITOK);
43     if (inp == NULL)
44         return (ENOBUFS);
45     bzero((caddr_t) inp, sizeof(*inp));

46     inp->inp_head = head;
47     inp->inp_socket = so;
48     insque(inp, head);
49     so->so_pcb = (caddr_t) inp;
50     return (0);
51 }
```

in_pcb.c

图22-6 in_pcballoc 函数：分配一个Internet PCB

36-45 in_pcballoc使用宏MALLOC调用内核的内存分配器。因为这些PCB总是作为系统调用的结果分配的，所以总能等到一个。

Net/2和早期的伯克利版本把Internet PCB和TCP PCB都保存在mbuf中。它们的大小分别是80和108字节。Net/3版本中的大小变成了84和140字节，所以TCP控制块不再适合存放在mbuf中。Net/3使用内核的内存分配器而不是mbuf分配两种控制块。

细心的读者会注意到图2-6的例子中，为PCB分配了17个mbuf，而我们刚刚讲到Net/3不再用mbuf存放Internet PCB和TCP PCB。但是，Net/3的确用mbuf存放Unix域的PCB，这就是计数器所指的。netstat输出的mbuf统计信息是针对内核为所有协议族分配的mbuf，而不仅仅是Internet协议族。

bzero把PCB设成0。这非常重要，因为PCB 中的IP地址和端口号必须被初始化成0。

in_pcb.c

```
252 int
253 in_pcbdetach(inp)
254 struct inpcb *inp;
255 {
256     struct socket *so = inp->inp_socket;

257     so->so_pcb = 0;
258     sofree(so);
259     if (inp->inp_options)
260         (void) m_free(inp->inp_options);
261     if (inp->inp_route.ro_rt)
262         rtfree(inp->inp_route.ro_rt);
263     ip_freemoptions(inp->inp_moptions);
264     remque(inp);
265     FREE(inp, M_PCB);
266 }
```

in_pcb.c

图22-7 in_pcbdetach 函数：释放一个Internet PCB

2. 把结构链接起来

46-49 in_head成员指向协议的PCB表头(udb或tcp)，inp_socket成员指向socket结

构，新的PCB结构被加到协议的双向链表上(insque)，socket结构指向该PCB。insque函数把新的PCB放到协议表的表头里。

在发布PRU_DETACH请求后，释放一个Internet PCB，这是在关闭插口时发生的。图22-7显示了in_pcbdetach函数，最后将调用它。

252-263 socket结构中的PCB指针被设成0，sofree释放该结构。如果给这个PCB分配的是一个有IP选项的mbuf，则由m_free将其释放。如果该PCB 中有一个路由，则由rtfree将其释放。所有多播选项都由ip_freemoptions释放。

264-265 remque把该PCB从协议的双向链表中移走，该PCB使用的内存被返回给内核。

22.5 绑定、连接和分用

在研究绑定插口、连接插口和分用进入的数据报的内核函数之前，我们先来看一下内核对这些动作施加的限制规则。

1. 绑定本地IP地址和端口号

图22-8是进程在调用bind时可以指定的本地IP地址和本地端口号的六种组合。

前三行通常是服务器的——它们绑定某个特定端口，称为服务器的知名端口(well-known port)，客户都知道这些端口的值。后三行通常是客户的——它们不考虑本地的端口，称为临时端口(ephemeral port)，只要它在客户主机上是唯一的。

大多数服务器和客户在调用bind时，都指定通配IP地址。如图22-8所示，在第3行和第6行中，用*表示。

本地IP地址	本地端口	描　述
单播或广播	非零	一个本地接口，特定端口
多播	非零	一个本地多播组，特定端口
*	非零	任何本地接口或多播组，特定端口
单播或广播	0	一个本地接口，内核选择端口
多播	0	一个多播组，内核选择端口
*	0	任何本地接口，内核选择端口

图22-8 bind的本地IP地址和本地端口号的组合

如果服务器把某个特定IP地址绑定到某个插口上(也就是说，不是通配地址)，那么进入的IP数据报中，只有那些以该特定IP地址作为目的IP地址的IP数据报——不管是单播、广播或多播——都被交付给该进程。自然地，当进程把某个特定单播或广播 IP地址绑定到某个插口上时，内核验证该IP地址与一个本地接口对应。

尽管可能，但很少出现客户程序绑定某个特定 IP地址的情况(图22-8中的第4行和第5行)。通常客户绑定通配IP地址(图22-8中的最后一行)，让内核根据自己选择的到服务器的路由来选择外出的接口。

图22-8没有显示如果客户程序试图绑定一个已经被其他插口使用的本地端口时会发生什么情况。默认情况下，如果一个端口已经被使用，进程是不能绑定它的。如果发生这种情况，则返回EADDRINUSE差错(地址正在被使用)。正在被使用(in use)的定义很简单，就是只要存在一个PCB，就把该端口作为它的本地端口。“正在被使用”的概念是相对于绑定协议的：TCP或UDP，因为TCP端口号与UDP端口号无关。

Net/3允许进程指定以下两个插口选项来改变这个默认行为：

SO_REUSEADDR 允许进程绑定一个正在被使用的端口号，但被绑定的IP地址(包括通配地址)必须没有被绑定到同一个端口。

例如，如果连到的接口的IP地址是140.252.1.29，则一个插口可以被绑定到140.252.1.29，端口5555；另一个插口可绑定到127.0.0.1，端口5555；还有一个插口可以绑定到通配IP地址，端口5555。在第二种和第三种情况下调用bind之前，必须先调用setsockopt，设置SO_REUSEADDR选项。

SO_REUSEPORT 允许进程重用IP地址和端口号，但是包括第一个在内的各个IP地址和端口号，必须指定这个插口选项。和SO_REUSEADDR一样，第一次绑定端口号时要指定插口选项。

例如，如果连到的接口具有140.252.1.29的IP地址，并且某个插口绑定到140.252.1.29，端口6666，并指定SO_REUSEPORT插口选项，则另一个插口也可以指定同一个插口选项，并绑定140.252.1.29，端口6666。

本节的后面将讨论在后一个例子中，当到达一个目的地址是140.252.1.29，目的端口是6666的IP数据报时，会发生什么情况。因为这两个插口都被绑定到该端结点上。

SO_REUSEPORT是Net/3新加上的，在4.4BSD中是为支持多播而引入的。在这个版本之前，两个插口是不可能绑定到同一个IP地址和同一个端口号的。

不幸的是，SO_REUSEPORT不是原来的标准多播源程序的内容，所以对它的支持并不广泛。其他支持多播的系统，如Solaris 2.x，让进程指定SO_REUSEADDR来表明允许把多个端口绑定到同一IP地址和相同的端口号。

2. 连接一个UDP插口

我们通常把connect系统调用和TCP客户联系起来，但是UDP客户或UDP服务器也可能调用connect，为插口指定外部IP地址和外部端口号。这就限制插口必须只与某个特定对方交换UDP数据报。

当连接UDP插口时，会有一个副作用：本地IP地址，如果在调用bind时没有指定，会自动被connect设置。它被设成由IP选路指定对方所选择的本地接口地址。

图22-9显示了UDP插口的三种不同的状态，以及函数为终止各状态调用的伪代码。

本地插口	外部插口	描　述
localIP.lport	*foreignIP.fport*	限制到一个对方： socket(), bind(*.*lport*), connect(*foreignIP, fport*) socket(), bind(*localIP, lport*), connect(*foreignIP, fport*)
localIP.lport	*.*	限制在本地接口上到达的数据报：*localIP* socket(), bind(*localIP, lport*)
**.lport*	*.*	接收所有发到*lport*的数据报： socket(),bind(*, *lport*)

图22-9 UDP插口的本地和外部IP地址和端口号规范

前三个状态叫做已连接的UDP插口(connected UDP socket)，后两个叫做未连接的UDP插口(unconnected UDP socket)。两个没有连接上的UDP插口的区别在于，第一个具有一个完全

指定的本地地址，而第二个具有一个通配本地 IP地址。

3. 分用TCP接收的IP数据报

图22-10显示了主机sun上的三个Telnet服务器的状态。前两个插口处于 LISTEN状态，等待进入的连接请求，加三个连接到 IP地址是140 252 111的主机上的端口1500。第一个监听插口处理在接口140.252.129上到达的连接请求，第二个监听插口将处理所有其他接口 (因为它的本地IP地址是通配地址)。

本地地址	本地端口	外部地址	外部端口	TCP状态
140.252.1.29	23	*	*	LISTEN
*	23	*	*	LISTEN
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED

图22-10 本地端口是23的三个TCP插口

两个具有未指定的外部 IP地址和端口号的监听插口都显示了出来，因为插口 API不允许TCP服务器限制任何一个值。 TCP服务器必须accept客户的连接，并在连接建立完成之后(也就是说，当TCP的三次握手结束之后)被告知客户的IP地址和端口号。只有到这个时候，如果服务器不喜欢客户的 IP地址和端口号，才能关闭连接。这并不是对 TCP要求的特性，这只是插口API通常的工作方式。

当TCP收到一个目的端口是 23的报文段时，它调用 in_pcblookup，搜索它的整个Internet PCB表，找到一个匹配。马上我们会研究这个函数，将看到它有优先权，因为它的通配匹配(wildcard match)数最少。为了确定通配匹配数，我们只考虑本地和外部的 IP地址，不考虑外部端口号。本地端口号必须匹配，否则我们甚至不考虑 PCB。通配匹配数可以是 0、1(本地IP地址或外部IP地址)或2(本地和外部IP地址)。

例如，假定到达报文段来自 140.252.1.11，端口1500，目的地是140.252.1.29，端口23。图22-11是图22-10中三个插口的通配匹配数。

本地地址	本地端口	外部地址	外部端口	TCP状态	通配匹配数
140.252.1.29	23	*	*	LISTEN	1
*	23	*	*	LISTEN	2
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED	0

图22-11 从{140.252.1.11, 1500}到{140.252.1.29, 23}的到达报文段

第一个插口匹配这四个值，但有一个通配匹配 (外部IP地址)。第二个插口也和到达报文段匹配，但有两个通配匹配 (本地和外部IP地址)。第三个插口是一个没有通配匹配的完全匹配。Net/3使用第三个插口，它具有最小通配匹配数。

继续这个例子，假定到达报文段来自 140.252.1.11，端口1501，目的地是140.252.1.29，端口23。图22-12显示了通配匹配数。

本地地址	本地端口	外部地址	外部端口	TCP状态	通配匹配数
140.252.1.29	23	*	*	LISTEN	1
*	23	*	*	LISTEN	2
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED	

图22-12 从{140.252.1.11, 1501}到{140.252.1.29, 23}的到达报文段

第一个插口匹配有一个通配匹配；第二个插口匹配有两个通配匹配；第三个插口根本不匹配，因为外部端口号不相等 (只有当PCB中的外部IP地址不是通配地址时，才比较外部端口号)。所以选择第一个插口。

在这两个例子中，我们没有提到到达 TCP报文段的类型：假定图 22-11中的报文段包含数据或对一个已经建立的连接的确认，因为它是发送到一个已经建立的插口上的。我们还假定，图22-12中的报文段是一个到达的连接请求 (一个SYN)，因为它是发送给一个正在监听的插口的。但是in_pcblookup的分用代码并不关心这些。如果 TCP报文段对交付的插口来说是错误的类型，我们将在后面看到， TCP会处理这种情况。现在，重要的是，分用代码只把 IP数据报中的源和目的插口对的值与 PCB中的值进行比较。

4. 分用UDP接收的IP数据报

UDP数据报的交付比我们刚才研究的 TCP的例子要复杂得多，因为可以把 UDP数据报发送到一个广播或多播地址。因为 Net/3(以及大多数支持多播的系统)允许多个插口有相同的本地IP地址和端口，所以如何处理多个接收方的情况呢？ Net/3的规则是：

1) 把目的地是广播 IP地址或多播IP地址的到达UDP数据报交付给所有匹配的插口。这里没有“ 最好的 ”匹配的概念(也就是具有最小通配匹配数的匹配)。

2) 把目的地是单播 IP地址的到达UDP数据报只交付给一个匹配的插口，就是具有最小通配匹配数的插口。如果有多个插口具有相同的“ 最小 ”通配匹配数，那么具体由哪个插口来接收到达数据报依赖于不同的实现。

图22-13显示了四个我们将在后面例子中使用的 UDP插口。要使四个UDP插口具有相同的本地端口号需要使用SO_REUSEADDR或SO_REUSEPORT。前两个插口已经被连接到一个外部IP地址和端口号，后面两个没有任何连接。

本地地址	本地端口	外部地址	外部端口	说　明
140.252.1.29	577	140.252.1.11	1500	已连接，本地IP = 单播
140.252.13.63	577	140.252.13.35	1500	已连接，本地IP = 广播
140.252.13.63	577	*	*	未连接，本地IP = 广播
*	577	*	*	未连接，本地IP = 通配地址

图22-13　四个本地端口为577的UDP插口

考虑目的地是 140.252.13.63(位于子网 140.252.13上的广播地址)，端口 577，来自140.252.13.34，端口1500。图22-14显示它被交付给第三和第四个插口。

本地地址	本地端口	外部地址	外部端口	交　付？
140.252.1.29	577	140.252.1.11	1500	不，本地和外部IP不匹配
140.252.13.63	577	140.252.13.35	1500	不，外部IP不匹配
140.252.13.63	577	*	*	交付
*	577	*	*	交付

图22-14　接收从{140.252.13.34, 1500}到{140.252.13.63, 577}的数据报

广播数据报不交付给第一个插口，因为本地 IP地址和目的IP地址不匹配，外部IP地址和源IP地址也不匹配。也不把它交付给第二个插口，因为外部 IP地址和源IP地址不匹配。

对于下一个例子，考虑目的地是140.252.129(一个单播地址)，端口577，来自140 252 1.111，

端口1500。图22-15显示了把该数据报交付给哪个端口。

本地地址	本地端口	外部地址	外部端口	交 付？
140.252.1.29	577	140.252.1.11	1500	交付，0个通配匹配
140.252.13.63	577	140.252.13.35	1500	不，本地和外部IP不匹配
140.252.13.63	577	*	*	不，本地IP不匹配
*	577	*	*	不，2个通配匹配

图22-15 接收从{140.252.1.11, 1500}到{140.252.1.29, 577}的数据报

该数据报和第一个插口匹配，且没有通配匹配；也和第四个插口匹配，但有两个通配匹配。所以，它被交付给第一个插口，最好的匹配。

22.6 in_pcblookup函数

in_pcblookup函数有几个作用：

1) 当TCP或UDP收到一个IP数据报时，in_pcblookup扫描协议的Internet PCB表，寻找一个匹配的PCB，来接收该数据报。这是运输层对收到数据报的分用。

2) 当进程执行bind系统调用，为某个插口分配一个本地IP地址和本地端口号时，协议调用in_pcbbind，验证请求的本地地址对没有被使用。

3) 当进程执行bind系统调用，请求给它的插口分配一个临时端口时，内核选了一个临时端口，并调用in_pcbbind检查该端口是否正在被使用。如果正在被使用，就试下一个端口号，以此类推，直到找到一个没有被使用的端口号。

4) 当进程显式或隐式地执行connect系统调用时，in_pcbbind验证请求的插口对是唯一的(当在一个没有连接上的插口上发送一个UDP数据报时，会隐式地调用connect，我们将在第23章看到这种情况)。

在第2种、第3种和第4种情况下，in_pcbbind调用in_pcblookup。两个选项使该函数的逻辑显得有些混乱。首先，进程可以指定SO_REUSEADDR或SO_REUSEPORT插口选项，表明允许复制本地地址。

其次，有时通配匹配也是允许的(例如，一个到达UDP数据报可以和一个自己的本地IP地址有通配符的PCB匹配，意味着该插口将接收在任何本地接口上到达的UDP数据报)，而其他情况下，一个通配匹配是禁止的(例如，当连接到一个外部IP地址和端口号时)。

> 在原始的标准IP多播代码中，出现了这样的注释“ in_pcblookup的逻辑比较模糊，也没有一点说明…… ”。形容词模糊比较保守。
>
> 公开的IP多播码是BSD／386的，是由Craig Leres从4.4BSD派生而来的。他修改了该函数过载的语义，只对上面的第 1种情况使用in_pcblookup。第2种和第4种情况由一个新函数in_pcbconflict处理。情况3由新函数in_uniqueport处理。把原来的功能分成几个独立的函数就显得更清楚了，但在我们描述的 Net/3版本中，整个逻辑还是结合在一个函数in_pcblookup中。

图22-16显示了in_pcblookuup函数。

该函数从协议的PCB表的表头开始，并可能会遍历表中的每个 PCB。变量match记录了到目前为止最佳匹配的指针入口，matchwild记录在该匹配中的通配匹配数。后者被初始化

成3，比可能遇到的最大通配匹配数还大(任何大于2的值都可以)。每次循环时，wildcard从0开始，计数每个PCB的通配匹配数。

1. 比较本地端口号

第一个比较的是本地端口号。如果PCB的本地端口和lport参数不匹配，则忽略该PCB。

in_pcb.c

```
405 struct inpcb *
406 in_pcblookup(head, faddr, fport_arg, laddr, lport_arg, flags)
407 struct inpcb *head;
408 struct in_addr faddr, laddr;
409 u_int   fport_arg, lport_arg;
410 int     flags;
411 {
412     struct inpcb *inp, *match = 0;
413     int     matchwild = 3, wildcard;
414     u_short fport = fport_arg, lport = lport_arg;

415     for (inp = head->inp_next; inp != head; inp = inp->inp_next) {
416         if (inp->inp_lport != lport)
417             continue;           /* ignore if local ports are unequal */

418         wildcard = 0;

419         if (inp->inp_laddr.s_addr != INADDR_ANY) {
420             if (laddr.s_addr == INADDR_ANY)
421                 wildcard++;
422             else if (inp->inp_laddr.s_addr != laddr.s_addr)
423                 continue;
424         } else {
425             if (laddr.s_addr != INADDR_ANY)
426                 wildcard++;
427         }

428         if (inp->inp_faddr.s_addr != INADDR_ANY) {
429             if (faddr.s_addr == INADDR_ANY)
430                 wildcard++;
431             else if (inp->inp_faddr.s_addr != faddr.s_addr ||
432                      inp->inp_fport != fport)
433                 continue;
434         } else {
435             if (faddr.s_addr != INADDR_ANY)
436                 wildcard++;
437         }

438         if (wildcard && (flags & INPLOOKUP_WILDCARD) == 0)
439             continue;           /* wildcard match not allowed */

440         if (wildcard < matchwild) {
441             match = inp;
442             matchwild = wildcard;
443             if (matchwild == 0)
444                 break;          /* exact match, all done */
445         }
446     }
447     return (match);
448 }
```

in_pcb.c

图22-16 in_pcblookuup 函数：搜索所有PCB寻找匹配

2. 比较本地地址

419—427 in_pcblookup比较PCB内的本地地址和laddr参数。如果有一个是通配地址，另一个不是，则wildcard计数器加1。如果都不是通配地址，则它们必须一样；否则忽略这个PCB 。如果都是通配地址，则什么也不改变：它们不可比，也不增加 wildcard计数器。图22-17对四种不同的情况做了小结。

PCB本地IP	laddr参数	描 述
不是*	*	wildcard++
不是*	不是*	比较IP地址，如果不相等，则略过PCB
*	*	不能比较
*	不是*	wildcard++

图22-17 in_pcblookup 做的四种IP地址比较

3. 比较外部地址和外部端口号

428—437 这几行完成与我们刚才讲的同样的检查，但是用外部地址而不是本地地址。而且，如果两个外部地址都不是通配地址，则不仅两个 IP地址必须相等，而且两个外部端口也必须相等。图22-18对外部IP地址的比较作了总结。

PCB外部IP	faddr参数	描 述
不是*	*	wildcard++
不是*	不是*	比较IP地址和端口，如果不相等，则略过PCB
*	*	不能比较
*	不是*	wildcard++

图22-18 in_pcblookup 做的四种外部IP地址比较

可以对图22-18中的第二行进行另外的外部端口号比较，因为一个 PCB不可能具有非通配外部地址，且外部端口号为 0。这个限制是由connect加上的，我们马上就会看到，该函数要求一个非通配外部 IP地址和一个非零外部端口。但是，也可能，并且通常都是具有一个通配本地地址和一个非零本地端口。我们在图 22-10和图22-13看到过这种情况。

4. 检查是否允许通配匹配

438—439 参数flags可以被设成INPLOOKUP_WILDCARD，意味着允许匹配中包含通配匹配。如果在匹配中有通配匹配(wildcard非零)，并且调用方没有指定这个标志位，则忽略这个PCB。当TCP和UDP调用这个函数分用一个到达数据报时，总是把 INPLOOKUP_WILDCARD置位，因为允许通配匹配(记住我们用图22-10和图22-13所作的例子)。但是，当这个函数作为connect系统调用的一部分而调用时，为了验证一个插口对没有被使用，把flags参数设成0。

5. 记录最佳匹配，如果找到确切匹配，则返回

440-447 这些语句记录到目前为止找到的最佳匹配。重复一下，最佳匹配是具有最小通配匹配数的匹配。如果一个匹配有一个或两个通配匹配，则记录该匹配，循环继续。但是，如果找到一个确切的匹配(wildcard是0)，则循环终止，返回一个指向该确切匹配 PCB的指针。

例子——分用收到的TCP报文段

图22-19取自我们在图 22-11中的TCP的例子。假定 in_pcblookup正在分用一个从140.252.1.11即端口1500到140.252.1.29即端口23的数据报。还假定PCB的顺序是图中行的顺序。

laddr是目的IP地址，lport是目的TCP端口，faddr是源IP地址，fport是源TCP端口。

PCB值				wildcard
本地地址	本地端口	外部地址	外部端口	
140.252.1.29	23	*	*	1
*	23	*	*	2
140.252.1.29	23	140.252.1.11	1500	0

图22-19 laddr =140.252.1.29, lport= 23, faddr = 140.252.1.11, fport= 1500

当把第一行和到达报文段比较时，wildcard是1(外部IP地址)，flags被设成INPLOOKUP_WILDCARD，所以把match设成指向该PCB，matchwild设为1。因为还没有找到确切的匹配，所以循环继续。下一次循环中，wildcard是2(本地和外部IP地址)，因为比matchwild大，所以不记录该入口，循环继续。再次循环时，wildcard是0，比matchwild(1)小，所以把这个入口记录在match中。因为已经找到了一个确切的地址，所以终止循环，把指向该PCB的指针返回给调用方。

如果TCP和UDP只用in_pcblookup来分用到达数据报，就可以对它进行简化。首先，没有必要检查faddr或laddr是否是通配地址，因为它们是收到数据报的源和目的IP地址。参数flags以及与相应的检测也可以不要，因为允许通配匹配。

这一节讨论了in_pcblookup函数的机制。我们在讨论in_pcbbind和in_pcbconnect如何调用这个函数后，将继续回来讨论它的意义。

22.7 in_pcbbind函数

下一个函数in_pcbbind，把一个本地地址和端口号绑定到一个插口上。从五个函数中调用它：

1) bind为某个TCP插口调用(通常绑定到服务器的一个知名端口上)；

2) bind为某个UDP插口调用(绑定到服务器的一个知名端口上，或者绑定到客户插口的一个临时端口上)；

3) connect为某个TCP插口调用，如果该插口还没有绑定到一个非零端口上(对TCP客户来说，这是一种典型情况)；

4) listen为某个TCP插口调用，如果该插口还没有绑定到一个非零端口(这很少见，因为是TCP服务器调用listen，TCP服务器通常绑定到一个知名端口上，而不是临时端口)；

5) 从in_pcbconnect(22.8节)调用，如果本地IP地址和本地端口号被置位(当为一个UDP插口调用connect，或为一个未连接UDP插口调用sendto时，这种情况比较典型)。

在第3种、第4种和第5种情形下，把一个临时端口号绑定到该插口上，不改变本地IP地址(在它已经被置位的情况下)。

称情形1和情形2为显式绑定(explicit bind)，情形3、4和5为隐式绑定(implicit bind)。我们也注意到，尽管在情形2时，服务器绑定到一个知名端口是很正常的，但那些用远程过程调用(RPC)启动的服务器也常常绑定到临时端口上，然后用其他程序注册它们的临时端口，该程序维护在该服务器的RPC程序号与其临时端口之间的映射(例如，卷1的29.4节描述的Sun端口映射器)。

我们分三部分显示in_pcbbind函数。图22-20是第一部分。

in_pcb.c

```
52 int
53 in_pcbbind(inp, nam)
54 struct inpcb *inp;
55 struct mbuf *nam;
56 {
57     struct socket *so = inp->inp_socket;
58     struct inpcb *head = inp->inp_head;
59     struct sockaddr_in *sin;
60     struct proc *p = curproc;    /* XXX */
61     u_short lport = 0;
62     int     wild = 0, reuseport = (so->so_options & SO_REUSEPORT);
63     int     error;

64     if (in_ifaddr == 0)
65         return (EADDRNOTAVAIL);
66     if (inp->inp_lport || inp->inp_laddr.s_addr != INADDR_ANY)
67         return (EINVAL);

68     if ((so->so_options & (SO_REUSEADDR | SO_REUSEPORT)) == 0 &&
69         ((so->so_proto->pr_flags & PR_CONNREQUIRED) == 0 ||
70          (so->so_options & SO_ACCEPTCONN) == 0))
71         wild = INPLOOKUP_WILDCARD;
```

in_pcb.c

图22-20 in_pcbbind 函数：绑定本地地址和端口号

64-67 前两个测试验证至少有一个接口已经被分配了一个IP地址，且该插口还没有绑定。不能两次绑定一个插口。

68-71 这个if语句有点令人疑问。总的结果是如果SO_REUSEADDR和SO_REUSEPORT都没有置位，就把wild设置成INPLOOKUP_WILDCARD。

对UDP来说，第二个测试为真，因为PR_CONNREQUIRED对无连接插口为假，对面向连接的插口为真。

第三个测试就是疑问所在[Torek 1992]。插口标志SO_ACCEPTCONN只被系统调用listen置位(15.9节)，该值只对面向连接的服务器有效。在正常情况下，一个TCP服务器调用socket、bind，然后调用listen。因而，当in_pcbbind被bind调用时，就清除了这个插口标志位。即使进程调用完socket后就调用listen，而不调用bind，TCP的PRU_LISTEN请求还是调用in_pcbbind，在插口层设置SO_ACCEPTCONN标志位之前，给插口分配一个临时端口。这意味着if语句中的第三个测试，测试SO_ACCEPTCONN是否没有置位，总是为真。因此if语句等价于

```
if ((so->so_options & (SO_REUSEADDR|SO_REUSEPORT)) == 0 &&
   ((so->so_proto->pr_flags & PR_CONNREQUIRED)==0||1)
        wild = INPLOOKUP_WILDCARD;
```

因为任何与1作逻辑或运算的结果都为真，所以这等价于

```
if ((so->so_options & (SO_REUSEADDR|SO_REUSEPORT)) == 0 )
         wild = INPLOOKUP_WILDCARD;
```

这样简单且容易理解：如果任何一个REUSE插口选项被置位，wild就是0。如果没有REUSE选项被置位，则把wild设成INPLOOKUP_WILDCARD。换言之，当函数在后面调用in_pcblookup时，只有在没有REUSE选项处于开状态时，才允许通配匹配。

in_pcbbind的下一部分，显示在图22-22中，函数处理可选nam参数。

72-75 只有当进程显式调用 bind时，nam参数才是一个非零指针。对一个隐式的绑定(connect、listen或in_pcbconnect的副作用，本节开始的情形3、4和5)，nam是一个空指针。当指定了该参数时，它是一个含有 sockaddr_in结构的mbuf。图22-21显示了非空参数nam的四种情形。

nam参数		PCB成员被设成：		说　明
localIP	*lport*	inp_laddr	inp_iport	
不是*	0	*localIP*	临时端口	*localIP*必须是本地接口
不是*	非零	*localIP*	*lport*	交付给in_pcblookup
*	0	*	临时端口	
*	非零	*	*lport*	交付给 in_pcblookup

图22-21 in_pcbbind 的nam 参数的四种情形

76-83 对正确的地址族的测试被注释掉了，但在函数 in_pcbconnect(图22-25)中执行了等价的测试。我们希望两者或者都有或者都没有。

85-94 Net/3测试被绑定的IP地址是否是一个多播组。如果是，则SO_REUSEADDR选项被认为与SO_REUSEPORT等价。

95-99 否则，如果调用方绑定的本地地址不是通配地址，则 ifa_ifwithaddr验证该地址与一个本地接口对应。

注释“ yech ”可能是因为插口地址结构中的端口号必须是 0，因为 ifa_ifwithaddr对整个结构作二进制比较，而不仅仅比较 IP地址。

这是进程在调用系统调用之前必须把插口地址结构全部置零的几种情况之一。如果调用 bind，并且插口地址结构 (sin_zero[8])的最后 8个字节非零，则ifa_ifwithaddr将找不到请求的接口，in_pcbbind会返回一个错误。

100-105 当调用方绑定了一个非零端口时，也就是说，进程要绑定一个特殊端口号 (图22-21中的第2种和第4种情形)，就执行下一个if语句。如果请求的端口小于1024(IPPORT_RESERVED)，则进程必须具有超级用户的优先权限。这不是 Internet协议的一部分，而是伯克利的习惯。使用小于 1024的端口号，我们称之为保留端口 (reserved port)，例如，rcmd函数 [Stevens 1990]使用的端口，rlogin和rsh客户程序又调用该函数，作为服务器对它们身份认证的一部分。

106-109 然后调用函数in_pcblookup(图22-16)，检测是否已经存在一个具有相同本地 IP地址和本地端口号的 PCB。第二个参数是通配 IP地址(外部IP地址)，第三个参数是一个为 0的端口号(外部端口号)。第二个参数的通配值导致 in_pcblookup忽略该PCB的外部IP地址和外部端口 —— 只把本地IP 地址和本地端口号分别和 sin->sin_addr及lport进行比较。我们前面提到，只有当所有REUSE插口选项都没有被设置时，才把wild设成INPLOOKUP_WILDCARD。

111 调用方的本地IP地址值存放在PCB中。如果调用方指定，它可以是通配地址。在这种情况下，由内核选择本地 IP地址，但要等到晚些时候插口连接上时。这就是为什么说本地 IP地址是根据外部IP地址，由IP路由选择决定。

in_pcb.c

```
72      if (nam) {
73          sin = mtod(nam, struct sockaddr_in *);
74          if (nam->m_len != sizeof(*sin))
75              return (EINVAL);
76  #ifdef notdef
77          /*
78           * We should check the family, but old programs
79           * incorrectly fail to initialize it.
80           */
81          if (sin->sin_family != AF_INET)
82              return (EAFNOSUPPORT);
83  #endif
84          lport = sin->sin_port;  /* might be 0 */
85          if (IN_MULTICAST(ntohl(sin->sin_addr.s_addr))) {
86              /*
87               * Treat SO_REUSEADDR as SO_REUSEPORT for multicast;
88               * allow complete duplication of binding if
89               * SO_REUSEPORT is set, or if SO_REUSEADDR is set
90               * and a multicast address is bound on both
91               * new and duplicated sockets.
92               */
93              if (so->so_options & SO_REUSEADDR)
94                  reuseport = SO_REUSEADDR | SO_REUSEPORT;
95          } else if (sin->sin_addr.s_addr != INADDR_ANY) {
96              sin->sin_port = 0;  /* yech... */
97              if (ifa_ifwithaddr((struct sockaddr *) sin) == 0)
98                  return (EADDRNOTAVAIL);
99          }
100         if (lport) {
101             struct inpcb *t;
102             /* GROSS */
103             if (ntohs(lport) < IPPORT_RESERVED &&
104                 (error = suser(p->p_ucred, &p->p_acflag)))
105                 return (error);
106             t = in_pcblookup(head, zeroin_addr, 0,
107                              sin->sin_addr, lport, wild);
108             if (t && (reuseport & t->inp_socket->so_options) == 0)
109                 return (EADDRINUSE);
110         }
111         inp->inp_laddr = sin->sin_addr;     /* might be wildcard */
112     }
```

in_pcb.c

图22-22 in_pcbbind 函数：处理可选的nam参数

当调用方显式绑定端口0，或nam参数是一个空指针(隐式绑定)时，in_pcbbind的最后一部分处理分配一个临时端口。

113-122 这个协议(TCP或UDP)使用的下一个临时端口号被维护在该协议的PCB表的head：tcb或udb。除了协议的head PCB中的inp_next和inp_back指针外，inpcb结构另一个唯一被使用的元素是本地端口号。令人迷惑的是，这个本地端口在head PCB中是主机字节序，而在表中其他PCB上，却是网络字节序！使用从1024开始的临时端口号(IPPORT_RESERVED)，每次加1，直到5000(IPPORT_USERRESERVED)，然后又从1024重新开始循环。该循环一直执行到in_pcbbind找不到匹配为止。

1. SO_REUSEADDR举例

让我们通过一些普通的例子，来了解一下in_pcbbind与in_pcblookup及两个REUSE

插口选项之间的交互。

in_pcb.c

```
113     if (lport == 0)
114         do {
115             if (head->inp_lport++ < IPPORT_RESERVED ||
116                 head->inp_lport > IPPORT_USERRESERVED)
117                 head->inp_lport = IPPORT_RESERVED;
118             lport = htons(head->inp_lport);
119         } while (in_pcblookup(head,
120                               zeroin_addr, 0, inp->inp_laddr, lport, wild));
121     inp->inp_lport = lport;
122     return (0);
123 }
```

in_pcb.c

图22-23 in_pcbbind 函数：选择一个临时端口

1) TCP或UDP通常以调用socket和bind开始。假定一个调用bind的TCP服务器，指定了通配IP 地址和它的非零知名端口23(Telnet服务器)。还假定该服务器还没有运行，进程没有设置SO_REUSEADDR插口选项。
in_pcbbind把INPLOOKUP_WILDCARD作为最后一个参数，调用in_pcblookup。in_pcblookup中的循环没有找到匹配的PCB，就假定没有其他进程使用服务器的知名TCP端口，返回一个空指针。一切正常，in_pcbbind，返回0。
2) 假定和上面相同的情况，但当再次试图启动服务器时，该服务器已经开始运行。
当调用in_pcblookup时，它发现了本地插口为{*, 23}的PCB。因为wildcard计数器是0，所以in_pcblookup返回指向这个入口的指针。因为reuseport是0，所以in_pcbbind返回EADDRINUSE。
3) 假定与上面相同的情况，但当第二次试图启动服务器时，指定了SO_REUSEADDR插口选项。
因为指定了这个插口选项，所以in_pcbbind在调用in_pcblookup时，最后一个参数为0。但本地插口为{*, 23}的PCB仍然匹配，因为in_pcblookup无法比较两个通配地址(图22-17)，所以wildcard为0。in_pcbbind又返回EADDRINUSE，避免启动两个具有相同本地插口的服务器例程，不管是否指定了SO_REUSEADDR。
4) 假定有一个Telnet服务器已经以本地插口{*, 23}开始运行，而我们试图以另一个本地插口{140.252.13.35，23}启动另一个服务器。
假定没有指定SO_REUSEADDR，调用in_pcblookup时，最后一个参数为INPLOOKUP_WILDCARD。当它与含有*.23的PCB比较时，wildcard计数器被设为1。因为允许通配匹配，所以在扫描完所有 TCP PCB后，就把这个匹配作为最佳匹配。in_pcbbind返回EADDRINUSE。
5) 这个例子与上一个相同，但为第二个试图绑定本地插口{140.252.13.35, 23}的服务器指定了SO_REUSEADDR插口选项。
现在，in_pcblookup的最后一个参数是0，因为指定了插口选项。当与本地插口为{*, 23}的PCB比较时，wildcard计数器为1，但因为最后的flags参数是0，所以跳过这个入口，不把它记作匹配。在比较完所有 TCP PCB后，函数返回一个空指针，in_pcbbind返回0。

6) 假定当我们试图以本地插口{*, 23}启动第二个服务器时，第一个Telnet服务器以本地插口{140.252.13.35, 23}启动。与前面的例子一样，但这一次我们以相反的顺序启动服务器。第一个服务器的启动没有问题，假定没有其他插口绑定到端口 23。当我们启动第二个服务器时，in_pcblookup的最后一个参数是INPLOOKUP_WILDCARD，假定没有指定SO_REUSEADDR插口选项。当和具有本地插口 {140.252.13.35, 23}的PCB比较时，wildcard被设成1，记录这个入口。在比较完所有 TCP PCB后，返回指向这个入口的指针。导致in_pcbbind返回EADDRINUSE。

7) 如果我们启动同一个服务器的两个例程，并且都是非通配本地 IP地址，会发生什么情况？假定我们以本地插口 {140.252.13.35, 23}启动第一个Telnet服务器，然后试图用本地插口{127.0.0.1, 23}启动第二个服务器，且不指定SO_REUSEADDR。

当第二个服务器调用 in_pcbbind时，它调用 in_pcblookup，最后一个参数是INPLOOKUP_WILDCARD。当比较具有本地插口{140.252.13.35, 23}的PCB时，因为本地IP地址不相等，所以跳过它。in_pcblookup返回一个空指针，in_pcbbind返回0。

从这个例子中我们看到，SO_REUSEADDR插口选项对非通配IP地址没有影响。事实上，只有当wildcard大于0 时，也就是说，当PCB入口具有一个通配IP地址，或者绑定的IP地址是一个通配地址时，才检查in_pcblookup中的INPLOOKUP_WILDCARD标志位。

8) 作为最后一个例子，假定我们试图启动同一服务器的两个例程，具有相同的非通配本地IP 地址127.0.0.1。

启动第二个服务器时，in_pcblookup总是返回具有相同本地插口的匹配PCB。不管是否指定SO_REUSEADDR插口选项，都发生这种情况，因为对这种比较，wildcard计数器总是0。因为in_pcblookup返回一个非空指针，所以in_pcbbind返回EADDRINUSE。

从这些例子中，我们可以指出本地 IP地址和SO_REUSEADDR插口选项的绑定规则。这些规则如图22-24所示。假定*localIP*1和*localP*2是在本地主机上有效的两个不同的单播或广播 IP地址，*localmcastIP*是一个多播组。我们还假定进程要绑定到一个已经绑定到某个已存在 PCB的非零端口号。

我们需要区分单播或多播地址和一个多播地址，因为我们看到，in_pcbbind认为对多播地址，SO_REUSEADDR与SO_REUSEPORT是一样。

存在PCB	试图绑定	SO_REUSEADDR		描 述
		关	开	
*localIP*1	*localIP*1	错误	错误	每个IP地址和端口一个服务器
*localIP*1	*localIP*2	正确	正确	每个本地接口一个服务器
*localIP*1	*	错误	正确	一个接口一个服务器，其他接口一个服务器
*	*localIP*1	错误	正确	一个接口一个服务器，其他接口一个服务器
*	*	错误	错误	不能复制本地插口(和第一个例子一样)
*localIP*1	*localIP*1	错误	正确	多个多播接收方

图22-24 SO_REUSEADDR 插口选项对绑定本地IP地址的影响

2. SO_REUSEPORT插口选项

Net/3中对SO_REUSEPORT的处理改变了in_pcbbind的逻辑，只要指定了SO_REUSEPORT，就允许复制本地插口。换言之，所有服务器都必须同意共享同一本地端口。

22.8 in_pcbconnect函数

函数in_pcbconnect为插口指定IP地址和外部端口号。有四个函数调用它：

1) connect为某个TCP插口(某个TCP客户的请求)调用；

2) connect为某个UDP插口(对UDP客户是可选的，UDP服务器很少见)调用；

3) 当在一个没有连接上的UDP插口(普通)上输出数据报时从sendto调用；

4) 当一个连接请求(一个SYN报文段)到达一个处于LISTEN状态(对TCP服务器是标准的)的TCP插口时，tcp_input调用。

在以上四种情况下，当调用in_pcbconnect时，通常，但不要求，不指定本地IP地址和本地端口。因此，在没有指定的情形下，由in_pcbconnect的一个函数给它们赋一个本地的值。

我们将分四个部分讨论in_pcbconnect函数。图22-25显示了第一部分。

in_pcb.c

```
130 int
131 in_pcbconnect(inp, nam)
132 struct inpcb *inp;
133 struct mbuf *nam;
134 {
135     struct in_ifaddr *ia;
136     struct sockaddr_in *ifaddr;
137     struct sockaddr_in *sin = mtod(nam, struct sockaddr_in *);
138     if (nam->m_len != sizeof(*sin))
139         return (EINVAL);
140     if (sin->sin_family != AF_INET)
141         return (EAFNOSUPPORT);
142     if (sin->sin_port == 0)
143         return (EADDRNOTAVAIL);
144     if (in_ifaddr) {
145         /*
146          * If the destination address is INADDR_ANY,
147          * use the primary local address.
148          * If the supplied address is INADDR_BROADCAST,
149          * and the primary interface supports broadcast,
150          * choose the broadcast address for that interface.
151          */
152 #define satosin(sa)     ((struct sockaddr_in *)(sa))
153 #define sintosa(sin)    ((struct sockaddr *)(sin))
154 #define ifatoia(ifa)    ((struct in_ifaddr *)(ifa))
155         if (sin->sin_addr.s_addr == INADDR_ANY)
156             sin->sin_addr = IA_SIN(in_ifaddr)->sin_addr;
157         else if (sin->sin_addr.s_addr == (u_long) INADDR_BROADCAST &&
158                  (in_ifaddr->ia_ifp->if_flags & IFF_BROADCAST))
159             sin->sin_addr = satosin(&in_ifaddr->ia_broadaddr)->sin_addr;
160     }
```

in_pcb.c

图22-25 in_pcbconnect 函数：验证参数，检查外部IP地址

1. 确认参数

130-143 nam参数指向一个包含sockaddr_in结构以及外部IP地址和端口号的mbuf。这些行确认参数并验证调用方不打算连接到端口号为0的端口上。

2. 特别处理到0.0.0.0和255.255.255.255的连接

134-160 对全局变量in_ifaddr的检查证实已配置了一个IP接口。如果外部地址是0.0.0.0(INADDR_ANY)，则用最初的IP接口的IP地址代替0.0.0.0。这就是说，调用进程是连接到这个主机上的一个对等实体的。如果外部IP地址是255.255.255.255 (INADDR_BROADCAST),

而且原来的接口支持广播，则用原来接口的广播地址代替 255.255.255.255。这样，UDP应用程序无需计算它的 IP 地址，就可以在原来的接口上广播 —— 它可以简单地把数据报发送给255.255.255.255，由内核把这个地址转换成该接口合适的 IP地址。

下一部分代码，如图 22-26所示，处理没有指定本地地址的情况。对 TCP和UDP客户程序来说，本节开始的表中的情形 1、2和3是非常普遍的。

in_pcb.c

```
161     if (inp->inp_laddr.s_addr == INADDR_ANY) {
162         struct route *ro;
163         ia = (struct in_ifaddr *) 0;
164         /*
165          * If route is known or can be allocated now,
166          * our src addr is taken from the i/f, else punt.
167          */
168         ro = &inp->inp_route;
169         if (ro->ro_rt &&
170             (satosin(&ro->ro_dst)->sin_addr.s_addr !=
171              sin->sin_addr.s_addr ||
172              inp->inp_socket->so_options & SO_DONTROUTE)) {
173             RTFREE(ro->ro_rt);
174             ro->ro_rt = (struct rtentry *) 0;
175         }
176         if ((inp->inp_socket->so_options & SO_DONTROUTE) == 0 &&     /* XXX */
177             (ro->ro_rt == (struct rtentry *) 0 ||
178              ro->ro_rt->rt_ifp == (struct ifnet *) 0)) {
179             /* No route yet, so try to acquire one */
180             ro->ro_dst.sa_family = AF_INET;
181             ro->ro_dst.sa_len = sizeof(struct sockaddr_in);
182             ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
183                 sin->sin_addr;
184             rtalloc(ro);
185         }
186         /*
187          * If we found a route, use the address
188          * corresponding to the outgoing interface
189          * unless it is the loopback (in case a route
190          * to our address on another net goes to loopback).
191          */
192         if (ro->ro_rt && !(ro->ro_rt->rt_ifp->if_flags & IFF_LOOPBACK))
193             ia = ifatoia(ro->ro_rt->rt_ifa);
194         if (ia == 0) {
195             u_short fport = sin->sin_port;
196             sin->sin_port = 0;
197             ia = ifatoia(ifa_ifwithdstaddr(sintosa(sin)));
198             if (ia == 0)
199                 ia = ifatoia(ifa_ifwithnet(sintosa(sin)));
200             sin->sin_port = fport;
201             if (ia == 0)
202                 ia = in_ifaddr;
203             if (ia == 0)
204                 return (EADDRNOTAVAIL);
205         }
```

in_pcb.c

图22-26 in_pcbconnect 函数：没有指定本地IP地址

3. 如果路由不再有效，则释放该路由

164-175 如果PCB中含有一条路由，但该路由的目的地址和已经连接上的外部地址不同，或者SO_DONTROUTE插口选项被置位，则放弃该路由。

为了理解为什么一个PCB会含有一条相关路由，考虑本节开始的表中的情形3：每次在一个未连接上的插口上发送UDP数据报时，就调用in_pcbconnect。每次进程调用sendto时，UDP输出函数调用in_pcbconnect、ip_output和in_pcbdisconnect。如果在该插口上发送的所有数据报都具有相同的目的IP地址，则第一次通过in_pcbconnect时，就分配了一条路由，从此时开始可以使用该路由。但是，因为UDP应用程序可能在每次调用sendto时，都向不同的IP地址发送数据报，所以必须比较目的地址和保存的路由。当目的地址改变时，就放弃该路由。ip_output也作同样的检查，这看起来似乎是多余的。

SO_DONTROUTE插口选项告诉内核旁路掉正常的选路决策，把该IP数据报发到本地连接的接口，该接口的IP网络地址和目的地址的网络部分匹配。

4. 获取路由

176-185 如果没有置位SO_DONTROUTE插口选项，则PCB中没有到目的地的路由，就要调用rtalloc获取一条路由。

5. 确定外出的接口

186-205 这一节代码的意图是让ia指向一个接口地址结构(in_ifaddr，6.5节)，该结构中包含了该接口的IP地址。如果PCB中的路由仍然有效，或者如果rtalloc找到一条路由，并且该路由不是到回环接口的，则使用相应的接口。否则，调用ifa_withdstaddr和ifa_withnet检查该外部IP地址是否在一个点到点链路的另一端，或者位于一个连到的网络上。两个函数都要求插口地址结构中的端口号为0，以便在调用期间保存在fport中。如果失败，就用原来的IP地址(in_ifaddr)，如果没有配置接口(in_ifaddr为0)，则返回错误。

图22-27显示了in_pcbconnect的下一部分，处理目的地址是多播地址的情况。

206-223 如果目的地址是一个多播地址，且进程指定了多播分组的外出接口(用IP_MULTICAST_IF插口选项)，则该接口的IP地址被用作本地地址。搜索所有IP接口，找到与插口选项所指定接口的匹配。如果该接口不存在，则返回错误。

224-225 图22-26的开头是处理通配本地地址情形的完整代码。指向本地接口ia的sockaddr_in结构的指针保存在ifaddr中。

in_pcblookup的最后一部分显示在图22-28中。

6. 验证插口对是唯一的

227-233 in_pcblookup验证插口对是唯一的。外部地址和外部端口号是指定给in_pcbconnect的参数的值。本地地址是已经绑定到该插口的值，或者是ifaddr中我们刚刚介绍的代码计算出来的值。本地端口可以是0，对TCP客户程序来说这是典型的。我们将在这部分代码的后面看到，为本地端口选择了一个临时端口。

这个测试避免从相同的本地地址和本地端口上建立两个到同一外部地址和外部端口的TCP连接。例如，如果我们与主机sun上的回显服务器建立了一个TCP连接，然后试图从同一本地端口(8888，用-b选项指定)建立另一条到同一服务器的连接，调用in_pcblookup后返回一个匹配，导致connect返回差错EADDRINUSE(我们用卷1附录C的sock程序)。

```
bsdi S sock -b 8888 sun echo &      启动后台的第一个
bsdi S sock -A  -b 8888 sun echo    然后再试一次
connect() error: Address already in use
```

in_pcb.c

```
206         /*
207          * If the destination address is multicast and an outgoing
208          * interface has been set as a multicast option, use the
209          * address of that interface as our source address.
210          */
211         if (IN_MULTICAST(ntohl(sin->sin_addr.s_addr)) &&
212             inp->inp_moptions != NULL) {
213             struct ip_moptions *imo;
214             struct ifnet *ifp;

215             imo = inp->inp_moptions;
216             if (imo->imo_multicast_ifp != NULL) {
217                 ifp = imo->imo_multicast_ifp;
218                 for (ia = in_ifaddr; ia; ia = ia->ia_next)
219                     if (ia->ia_ifp == ifp)
220                         break;
221                 if (ia == 0)
222                     return (EADDRNOTAVAIL);
223             }
224         }
225         ifaddr = (struct sockaddr_in *) &ia->ia_addr;
226     }
```

in_pcb.c

图22-27 in_pcbconnect 函数：目的地址是一个多播地址

in_pcb.c

```
227     if (in_pcblookup(inp->inp_head,
228                      sin->sin_addr,
229                      sin->sin_port,
230                  inp->inp_laddr.s_addr ? inp->inp_laddr : ifaddr->sin_addr,
231                      inp->inp_lport,
232                      0))
233         return (EADDRINUSE);

234     if (inp->inp_laddr.s_addr == INADDR_ANY) {
235         if (inp->inp_lport == 0)
236             (void) in_pcbbind(inp, (struct mbuf *) 0);
237         inp->inp_laddr = ifaddr->sin_addr;
238     }
239     inp->inp_faddr = sin->sin_addr;
240     inp->inp_fport = sin->sin_port;
241     return (0);
242 }
```

in_pcb.c

图22-28 in_pcbconnect 函数：验证插口对是唯一的

我们指定-A选项，设置SO_REUSEADDR插口选项，使bind成功，但是connect不成功。这是一个人为的例子，因为我们显式地把两个插口都绑定到同一本地端口上(8888)。在正常情形下，主机bsdi上的两个不同客户程序连接到sun的回显服务器上，当第二个客户程序调用图22-28中的in_pcblookup函数时，本地端口将是0。

这个测试也避免了两个UDP插口从相同的本地端口上连接到同一个外部地址。但这个测试不能避免两个UDP插口从同一个本地端口上交替地向同一个外部地址发送数据报，只要它们都不调用connect。因为UDP插口在sendto系统调用的过程中，只是临时连接到一个对等实体上。

7. 隐式绑定和分配临时端口

234-238 如果插口的本地地址仍然是通配匹配的，则把它设置成ifaddr中保存的值。这是一个隐式绑定：22.7节开始时讲的情形3、4和5。首先，检查本地端口是否已经被绑定，如果没有，in_pcbbind就把该插口绑定到一个临时端口。调用in_pcbbind和给inp_laddr赋值的顺序很重要，因为如果本地地址不是通配地址，则in_pcbbind会失败。

8. 把外部地址和外部端口存放在PCB中

239-240 这个函数的最后一步设置PCB的外部IP地址和外部端口号成员。如果这个函数成功返回，我们就能保证PCB中的插口对——本地的和外部的——都有了特定的值。

IP源地址与外出接口地址

在IP数据报的源地址和用来发送该数据报接口的IP地址之间有些微妙的差别。

TCP和UDP把PCB成员inp_laddr用作该IP数据报的源地址。它可由进程设成任何被bind配置的接口的IP地址(在in_pcbbind中调用ifa_ifwithaddr验证应用程序想要的本地地址)。只有当本地地址是一个通配地址时，in_pcbconnect才给它赋值。而当这种情况发生时，本地地址是根据外出接口分配的(因为目的地址已知)。

但是，外出接口也是根据目的IP地址，由ip_output确定的。在多接口主机上，当进程显式绑定一个不同于外出接口的本地地址时，源地址有可能是一个本地接口的IP地址，且该接口不是外出的接口。这种情况是允许的，因为Net/3选择了弱端系统模式(8.4节)。

22.9 in_pcbdisconnect函数

ip_pcbdisconnect把UDP插口断连。把外部IP地址设成全0(INADDR_ANY)，外部端口号设成0，就把外部相关内容删除了。

这是在已经在一个未连接上的UDP插口上发送了一个数据报后，在一个连接上的UDP插口上调用connect时做的。在第一种情况下，调用sendto的次序是：UDP调用in_pcbconnect把插口临时连接到目的地，udp_output发送数据报，然后in_pcbdisconnect删除临时连接。

当关闭插口时，不调用in_pcbdisconnect，因为in_pcbdetach处理释放PCB。只有当一个不同的地址或端口号要求重用该PCB时，才断连。

图22-29显示了in_pcbdisconnect函数。

in_pcb.c

```
243 int
244 in_pcbdisconnect(inp)
245 struct inpcb *inp;
246 {

247     inp->inp_faddr.s_addr = INADDR_ANY;
248     inp->inp_fport = 0;
249     if (inp->inp_socket->so_state & SS_NOFDREF)
250         in_pcbdetach(inp);
251 }
```

in_pcb.c

图22-29 in_pcbdisconnect函数：与外部地址和端口号断连

如果该PCB不再有文件表引用(SS_NOFDREF置位)，则in_pcbdetach(图22-7)释放该PCB。

22.10 in_setsockaddr和in_setpeeraddr函数

getsockname系统调用返回插口的本地协议地址 (例如, Internet插口的IP 地址和端口号), getpeername系统调用返回外部协议地址。两个系统调用终止时, 都发布一个PRU_SOCKADDR或PRU_PEERADDR请求。然后协议调用 in_setsockaddr或in_setpeeraddr。图22-30显示了以上的第一种情况。

in_pcb.c

```
267 int
268 in_setsockaddr(inp, nam)
269 struct inpcb *inp;
270 struct mbuf *nam;
271 {
272     struct sockaddr_in *sin;
273     nam->m_len = sizeof(*sin);
274     sin = mtod(nam, struct sockaddr_in *);
275     bzero((caddr_t) sin, sizeof(*sin));
276     sin->sin_family = AF_INET;
277     sin->sin_len = sizeof(*sin);
278     sin->sin_port = inp->inp_lport;
279     sin->sin_addr = inp->inp_laddr;
280 }
```

in_pcb.c

图22-30 in_setsockaddr 函数：返回本地地址和端口号

参数nam是一个指针, 该指针指向一个用来存放结果的 mbuf：一个sockaddr_in结构, 系统调用复制给进程的备份。该代码填写插口地址结构的内容, 并把 IP 地址和端口号从Internet PCB拷贝到sin_addr和sin_port成员中。

图22-31显示了in_setpeeraddr函数。它基本上等同于图22-30中的代码, 但从PCB中拷贝了外部IP地址和端口号。

in_pcb.c

```
281 int
282 in_setpeeraddr(inp, nam)
283 struct inpcb *inp;
284 struct mbuf *nam;
285 {
286     struct sockaddr_in *sin;
287     nam->m_len = sizeof(*sin);
288     sin = mtod(nam, struct sockaddr_in *);
289     bzero((caddr_t) sin, sizeof(*sin));
290     sin->sin_family = AF_INET;
291     sin->sin_len = sizeof(*sin);
292     sin->sin_port = inp->inp_fport;
293     sin->sin_addr = inp->inp_faddr;
294 }
```

in_pcb.c

图22-31 in_setpeeraddr 函数：返回外部地址和端口号

22.11 in_pcbnotify、in_rtchange和in_losing函数

当收到一个ICMP差错时, 调用in_pcbnotify函数, 把差错通知给合适的进程。通过对所有的PCB搜索一个协议 (TCP或UDP), 并把本地和外部IP地址及端口号与ICMP差错返回的值进行比较, 找到“合适的进程”。例如, 当因为一些路由器丢掉了某个 TCP报文段而收到

ICMP源抑制差错时，TCP必须找到产生该差错的连接的PCB，放慢在该连接上的传输速度。

在显示该函数之前，我们必须回顾一下它是怎样被调用的。图 22-32总结了处理ICMP差错时调用的函数。两个有阴影的椭圆是本节描述的函数。

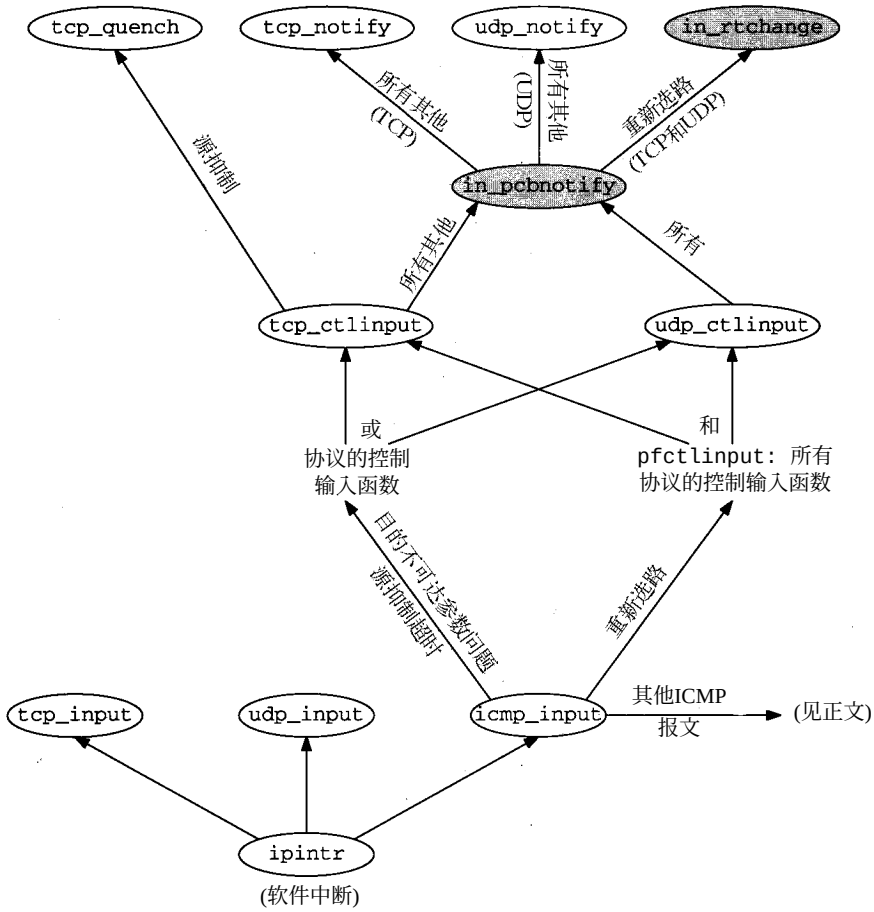


图22-32 ICMP差错处理总结

当收到一个ICMP报文时，调用`icmp_input`。ICMP的五种报文按差错来划分 (图11-1和图11-2)：

- 目的主机不可达；
- 参数问题；
- 重定向；
- 源抑制；
- 超时。

重定向的处理不同于其他四个差错。所有其他的 ICMP报文(查询)的处理见第11章。

每个协议都定义了它的控制输入函数，即`protosw`结构(7.4节)中的`pr_ctlinput`入口。对TCP和UDP，它们分别称为`tcp_ctlinput`和`udp_ctlinput`，我们将在后面几章给出它们的代码。因为收到的ICMP差错中包含了引起差错的数据报的 IP首部，所以引起该差错的协议(TCP或UDP)是已知的。这五个 ICMP差错中的四个将引起对协议的控制输入函数的调用。

重定向的处理不同：调用函数 pfctlinput，它继续调用协议族(Internet)中所有协议的控制输入函数。TCP和UDP是Internet协议族中仅有的两个具有控制输入函数的协议。

重定向的处理是特殊的，因为它们不仅影响产生重定向的数据报，还将影响所有到该目的地的IP数据报。另一方面，其他四个差错只需由产生差错的协议进行处理。

in_pcb.c

```
306 int
307 in_pcbnotify(head, dst, fport_arg, laddr, lport_arg, cmd, notify)
308 struct inpcb *head;
309 struct sockaddr *dst;
310 u_int   fport_arg, lport_arg;
311 struct in_addr laddr;
312 int     cmd;
313 void    (*notify) (struct inpcb *, int);
314 {
315     extern u_char inetctlerrmap[];
316     struct inpcb *inp, *oinp;
317     struct in_addr faddr;
318     u_short fport = fport_arg, lport = lport_arg;
319     int     errno;

320     if ((unsigned) cmd > PRC_NCMDS || dst->sa_family != AF_INET)
321         return;
322     faddr = ((struct sockaddr_in *) dst)->sin_addr;
323     if (faddr.s_addr == INADDR_ANY)
324         return;

325     /*
326      * Redirects go to all references to the destination,
327      * and use in_rtchange to invalidate the route cache.
328      * Dead host indications: notify all references to the destination.
329      * Otherwise, if we have knowledge of the local port and address,
330      * deliver only to that socket.
331      */
332     if (PRC_IS_REDIRECT(cmd) || cmd == PRC_HOSTDEAD) {
333         fport = 0;
334         lport = 0;
335         laddr.s_addr = 0;
336         if (cmd != PRC_HOSTDEAD)
337             notify = in_rtchange;
338     }
339     errno = inetctlerrmap[cmd];
340     for (inp = head->inp_next; inp != head;) {
341         if (inp->inp_faddr.s_addr != faddr.s_addr ||
342             inp->inp_socket == 0 ||
343             (lport && inp->inp_lport != lport) ||
344             (laddr.s_addr && inp->inp_laddr.s_addr != laddr.s_addr) ||
345             (fport && inp->inp_fport != fport)) {
346             inp = inp->inp_next;
347             continue;           /* skip this PCB */
348         }
349         oinp = inp;
350         inp = inp->inp_next;
351         if (notify)
352             (*notify) (oinp, errno);
353     }
354 }
```

in_pcb.c

图22-33 in_spcbnotify 函数：把差错通知传给进程

有关图22-32我们要做的最后一点说明是，TCP在处理源抑制差错时，与其他差错的处理不同，而重定向由in_pcbnotify特别处理：不管引起差错的是什么协议，都调用in_rtchange函数。

图22-33显示了in_pcbnotify函数。当TCP调用它时，第一个参数是tcb的地址，最后一个参数是函数tcp_notify的地址。对UDP来说，这两个参数分别是udb的地址和函数udp_notify的地址。

1. 验证参数

306-324 验证cmd参数和目的地址族。检测外部地址，保证它不是0.0.0.0。

2. 特殊处理重定向

325-338 如果差错是重定向，则对它的处理是特殊的(差错PRC_HOSTDEAD是一种旧的差错，由IMP产生。目前的系统再也看不到这种差错了——它是一个历史产物)。外部端口、本地端口和本地地址都被设成全0，这样后面的for循环就不会比较它们了。对于重定向，我们需要该循环只根据外部IP地址选出接收通知的PCB，因为主机是在这个IP地址上接收到重定向的。而且，为重定向调用的函数是in_rtchange(图22-34)，而不是调用方指定的notify参数。

339 全局数组inetctlerrmap把协议无关差错码(图11-19中的PRC_*xxx*值)映射到它对应的Unix的errno值(图11-1的最后一栏)。

3. 为所选的PCB调用通知函数

341-353 这个循环选择要通知的PCB。可以通知多个PCB——该循环在找到匹配后仍然继续。第一个if语句结合了五个检测，如果这五个中有任一个为真，则跳过该PCB：(1)如果外部地址不相等；(2)如果该PCB没有对应的socket结构；(3)如果本地端口不相等；(4)如果本地地址不相等；或(5)如果外部端口不相等。外部地址必须匹配，但只有当对应的参数非零时，才比较其他三个外部和本地参数。当找到一个匹配时，调用notify函数。

22.11.1 in_rtchange函数

我们看到，当ICMP差错是一个重定向时，in_pcbnotify调用in_rtchange函数。对所有外部地址与已重定向的IP地址匹配的PCB，都调用该函数。图22-34显示了in_rtchange函数。

in_pcb.c

```
391 void
392 in_rtchange(inp, errno)
393 struct inpcb *inp;
394 int     errno;
395 {
396     if (inp->inp_route.ro_rt) {
397         rtfree(inp->inp_route.ro_rt);
398         inp->inp_route.ro_rt = 0;
399         /*
400          * A new route can be allocated the next time
401          * output is attempted.
402          */
403     }
404 }
```

in_pcb.c

图22-34 in_rtchange函数：使路由无效

如果该PCB中有路由，则rtfree释放该路由，且该PCB 成员被标记为空。此时，我们不用重定向返回的路由来更新路由。当这个 PCB被再次使用时，ip_output会根据内核的选路表重新分配新的路由，而该选路表是在调用 pfctlinput之前，由重定向报文更新的。

22.11.2 重定向和原始插口

让我们来研究一下重定向、原始插口和缓存在 PCB中的路由之间的交互。如果我们运行Ping程序，该程序使用一个原始插口，收到来自被 ping的IP地址发来的ICMP重定向差错。Ping程序继续使用原来的路由，而不是已重定向的路由。我们可以从以下过程来看。

我们从位于1 402 521网络上的gemini主机 ping位于14 025 213网络上的svr4主机。gemini的默认路由是gateway，但分组应该被发送到路由器netb。图22-35显示了这个安排。

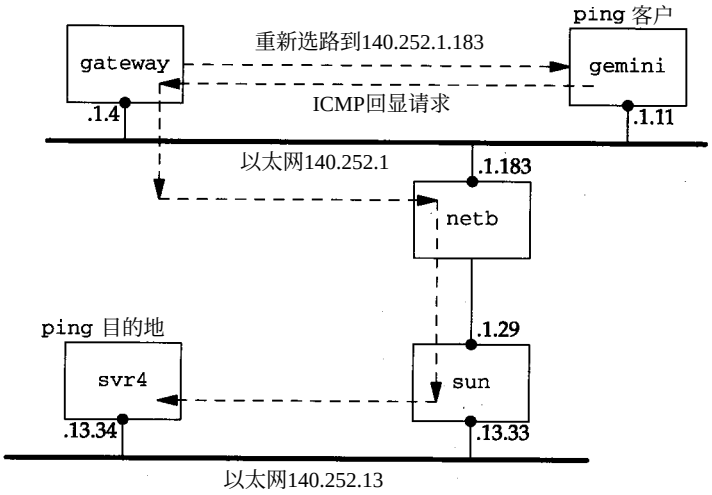


图22-35 ICMP重定向举例

我们希望gateway在收到第一个ICMP回显请求时，发一个重定向。

```
gemini $ ping -sv svr4
PING 140.252.13.34: 56 data bytes
ICMP Host redirect from gateway 140.252.1.4
  to netb (140.252.1.183) for svr4 (140.252.13.34)
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=572. ms
ICMP Host redirect from gateway 140.252.1.4
  to netb (140.252.1.183) for svr4 (140.252.13.34)
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=392. ms
```

选项-s使每隔一秒发送一次ICMP回显请求，选项-v打印每个收到的ICMP报文(不仅仅是ICMP回显回答)。

每个ICMP回显请求引出一个重定向，但 ping使用的原始插口从来不通知重定向改变它正在使用的路由。第一次计算出来并被保存在 PCB中的路由，使 IP数据报被发送到路由器gateway{140.252.1.4}，应该更新它，使数据报能被发送到路由器 netb{140.252.1.183}上。我们看到，gemini上的内核接收ICMP重定向，但它们被略过了。

如果我们终止ping程序，并重新运行它，我们就再也看不到重定向了：

```
gemini $ ping -sv svr4
```

```
PING 140.252.13.34: 56 data bytes
64 bytes from svr4 (140.252.13.34): icmp_seq=0,time=388. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=363. ms
```

这个不正常的原因是原始IP插口代码(第32章)没有控制输入函数。只有TCP和UDP有控制输入函数。当收到重定向时，ICMP更新内核的选路表，调用pfctlinput(图22-32)。但是因为原始IP协议没有控制输入函数，所以不释放与Ping的原始插口相关的PCB中高速缓存的路由。但是，当我们第二次运行ping程序时，根据内核更新后的选路表分配路由，所以我们看不到重定向了。

22.11.3 ICMP差错和UDP插口

插口API令人迷惑的一部分是，不把在UDP插口上收到的ICMP差错传给应用程序，除非该应用程序在该插口上发布connect，限制该插口的外部IP地址和端口号。现在我们来看一下in_pcbnotify是如何实施这一限制的。

考虑某个ICMP插口不可达，这大概是UDP插口上最普通的一种ICMP差错了。in_pcbnotify的dst参数内的外部IP地址和外部端口号是引起ICMP差错的IP地址和端口号。但是，如果该进程已经在该插口上发布connect命令，则PCB的inp_faddr和inp_fport成员都是0，避免in_pcbnotify在该插口上调用notify函数。图22-33中的for循环将跳过每个UDP PCB。

产生这个限制的原因有两个。首先，如果正在发送的进程有一个未连接上的UDP插口，则该插口对中唯一的非零元素是本地端口(假定该进程不调用bind)。这是in_pcbnotify在分用进入的ICMP差错，并把它传给正确进程时，唯一可用的值。尽管很少发生，但也可能有多个进程都绑定到相同的本地端口上，所以具体由哪个进程接收ICMP差错就不明确了。还有一种可能就是，发送引起ICMP差错数据报的进程已经终止了，而另一个进程又开始运行并使用同一本地端口。这也不太可能，因为临时端口是从1024到5000按顺序分配的，只有循环一遍以后才可能重用同一端口号(图22-23)。

这个限制的第二个原因是，内核给进程的差错通知——一个errno值——是不够的。考虑某个进程连续三次在一个未连接上的UDP插口上调用sendto函数，向三个不同的目的地发送一个UDP数据报，然后用recvfrom等待回答。如果其中一个数据报生成一个ICMP端口不可达差错，且内核将向该进程发布的recvfrom返回对应的差错(ECONNREFUSED)，那么，errno值并没有告诉进程是哪个数据报产生了该差错。内核具有ICMP差错所要求的所有信息，但是插口API并不提供手段把这些信息返回给该进程。

因此，如果进程想要得到在某个UDP插口上的这些ICMP差错通知，在设计时就必须决定插口只能连接到一个对等实体上。如果在该连接上的插口返回ECONNREFUSED差错，毫无疑问就是该对等实体产生的差错。

还有一种远程可能性，会把ICMP差错交付给错误的进程。假设某个进程发送了一个UDP数据报，引起一个ICMP差错，但它在收到该差错之前终止了。另一个进程在收到该差错之前开始运行，并且绑定到同一个本地端口，连接到相同的外部地址和外部端口上，导致这个新进程接收到前面的ICMP差错。由于UDP缺少内存，所以无法避免这种情况的发生。我们将看到TCP用它的TIME_WAIT状态处理这个问题。

在我们前面的例子中，应用程序绕开这个限制的一个办法是使用三个连接上的UDP插口，

而不是一个未连接上的插口，并在其中任意一个有收到的数据报或差错要读写时，调用select函数来确定。

这里我们有一种情形是内核有足够的信息而API(插口)的信息不足。大多数Unix系统V及其他常见的API(TLI)，其逆为真：TLI函数t_rcvuderr可以返回对等实体的IP地址、端口号以及一个差错值。但大多数TCP/IP的SVR4流实现都不为ICMP提供手段，把差错传递给一个未连接上的UDP端节点。

在理想情况下，in_pcbnotify把ICMP差错交付给所有匹配的UDP插口，即使唯一的非通配匹配是本地端口。返回给进程的差错将包括产生差错的目的IP地址和目的UDP端口，允许进程确定该差错是否是它发送的数据报产生的。

22.11.4 in_losing函数

处理PCB的最后一个函数图22-36的in_losing。当TCP的某个连接的重传定时器连续第三次超时时，调用该函数。

in_pcb.c

```
361 int
362 in_losing(inp)
363 struct inpcb *inp;
364 {
365     struct rtentry *rt;
366     struct rt_addrinfo info;
367     if ((rt = inp->inp_route.ro_rt)) {
368         inp->inp_route.ro_rt = 0;
369         bzero((caddr_t) & info, sizeof(info));
370         info.rti_info[RTAX_DST] =
371             (struct sockaddr *) &inp->inp_route.ro_dst;
372         info.rti_info[RTAX_GATEWAY] = rt->rt_gateway;
373         info.rti_info[RTAX_NETMASK] = rt_mask(rt);
374         rt_missmsg(RTM_LOSING, &info, rt->rt_flags, 0);
375         if (rt->rt_flags & RTF_DYNAMIC)
376             (void) rtrequest(RTM_DELETE, rt_key(rt),
377                              rt->rt_gateway, rt_mask(rt), rt->rt_flags,
378                              (struct rtentry **) 0);
379         else
380             /*
381              * A new route can be allocated
382              * the next time output is attempted.
383              */
384             rtfree(rt);
385     }
386 }
```

in_pcb.c

图22-36 in_losing 函数：使高速缓存路由信息无效

1. 产生选路报文

361-374 如果PCB中有一个路由，则丢掉该路由。用要失效的高速缓存路由的有关信息填充一个rt_addrinfo结构。然后调用rt_missmsg函数，从RTM_LOSING类型的选路插口中生成一个报文，指明有关该路由的问题。

2. 删除或释放路由

375-384 如果高速缓存路由是由一个重定向生成的（RTF_DYNAMIC置位），则用请求RTM_DELETE调用rtrequest，删除该路由。否则释放高速缓存的路由，这样，当该插口上有下一个输出时，为它重新分配一条到目的地的路由——希望是一条更好的路由。

22.12 实现求精

毫无疑问，这一章我们遇到的最耗时的算法是in_pcblookup做的对PCB的线性搜索。22.6节一开始，我们就注意到有四种情况会调用这个函数。可以忽略对bind和connect的调用，因为TCP和UDP在分用每个收到的IP数据报时，调用它们的次数比调用in_pcblookup的少得多。

后面几章我们将看到，TCP和UDP试图帮助这个线性搜索，它们都维护一个指向该协议引用的最后一个PCB的指针：一个单入口高速缓存。如果高速缓存的PCB的本地地址、本地端口、外部地址和外部端口与收到的数据报的值匹配，则协议根本就不调用in_pcblookup。如果协议数据适合分组列模型[Jain和Routhier 1986]，这个简单的高速缓存效果很好。但是，如果数据不适合这个模型，例如，看起来象联机交易处理系统的数据入口，则单入口高速缓存的效率很低[McKenney和Dove 1992]。

一个稍好一点的PCB安排的建议是，当引用某个PCB时，把它移到该PCB表的最前面([McKenney和Dove 1992]把这个想法给了Jon Crowcroft；[Partridge和Pink 1993]把它给了Gary Delp)。移动PCB很容易，因为该表是一个双向链表，而且in_pcblookup的第一个参数是一个指向该表表头的指针。

[McKenney和Dove 1992]把原始的Net/1实现(没有高速缓存)，一种提高的单入口发送-接收高速缓存，“移到最前面”启发算法，以及他们自己的使用散列链的算法做了比较。他们指出，在散列链上维护一个PCB的线性表比其他算法的性能提高了一个数量级。散列链的唯一耗费是需要内存存放散列链的链头，以及计算散列函数。他们也考虑把“移到最前面”启发算法与他们的散列链算法结合，结论是只增加一些散列链，更为简单。

BSD线性搜索和散列表搜索的另一个比较是在[Hutchinson和Peterson 1991]中。他们指出，随着散列表中插口数量的增加，分用一个进入的UDP数据报所需要的时间是常量，但线性搜索所需要的时间随插口数量的增加而增加。

22.13 小结

每个Internet插口都有一个相关的Internet PCB：TCP、UDP和原始IP。它包含了Internet插口的一般信息：本地和外部IP地址，指向一个路由结构的指针等等。给定协议的所有PCB都放在该协议维护的一个双向链表上。

本章中，我们研究了多个操作PCB的函数，对其中的三个作了详细的讨论：

1) TCP和UDP调用in_pcblookup分用每个进入的数据报。它选择接收数据报的插口，考虑通配匹配。

in_pcbbind也调用这个函数来验证本地地址和本地进程是唯一的；in_pcbconnect调用这个函数验证本地地址、本地进程、外部地址和外部进程的组合是唯一的。

2) in_pcbbind显式或隐式地把一个本地地址和本地端口号绑定到一个插口。当进程调用bind时，发生显式绑定；当一个TCP客户程序调用connect而不调用bind时，或当一个

UDP进程调用sendto或connect而不调用bind时，发生隐式绑定。

3) in_pcbconnect设置外部地址和外部进程。如果进程还没有设置本地地址，计算一条到外部地址的路由，结果的本地接口成为本地地址。如果进程还没有设置本地端口，in_pcbbind为插口选择一个临时端口。

图22-37对多种TCP和UDP应用程序以及存放在PCB中的本地地址，本地端口、外部地址和外部端口的值做了总结。我们还没有讨论完图 22-37中TCP和UDP进程的所有动作，将在后面的章节中继续讨论。

应用程序	本地地址：inp_laddr	本地端口：inp_lport	外部地址：inp_faddr	外部端口：inp_fport
TCP客户程序：connect(*foreignIP, fport*)	in_pcbconnect调用rtalloc为*foreignIP*分配路由。本地地址是本地接口	in_pcbconnect调用in_pcbbind选择临时端口。	*foreignIP*	*fport*
TCP客户程序：bind(*localIP, lport*) connect(*foreignIP,fport*)	*localIP*	*lport*	*foreignIP*	*fport*
TCP客户程序：bind(**, lport*) connect(*foreignIP, fport*)	in_pcbconnect调用rtalloc为*foreignIP*分配路由。本地地址是本地接口	*lport*	*fport*	*fport*
TCP客户程序:bind(*localIP, 0*) connect(*foreignIP, fport*)	*localIP*	in_pcbbind选择临时端口	*foreignIP*	*fport*
TCP服务器程序：bind(*localIP, lport*) listen()accept()	*localIP*	*lport*	IP首部内的源地址	TCP首部内的源端口地址
TCP服务器程序：bind(**, lport*) listen() accept()	IP首部里的目的地址	*lport*	*foreignIP*。发送完数据报后，置位为0.0.0.0	TCP首部内的源端口地址
UDP客户程序：sendto(*foreignIP, fport*)	in_pcbconnect调用rtalloc为*foreignIP*分配路由。本地地址是本地接口。在发送完数据报之后，置位为0.0.0.0	in_pcbconnect调用in_pcbbind选择临时端口。后面调用sendto时不改变	*foreignIP*	*fport*。发送完数据报后，置位为0
UDP客户程序：connect(*foreignIP, fport*) write()	in_pcbconnect调用rtalloc为*foreignIP*分配路由。本地地址是本地接口。后面调用write时不改变	in_pcbconnect调用in_pcbbind选择临时端口。后面调用write时不改变	*foreignIP*	*fport*

图22-37 in_pcbbind 和in_pcbconnect 的总结

习题

22.1 在图22-23中，当进程请求一个临时端口，而所有临时端口都被使用时，会发生什么情况？

22.2 在图22-10中，我们显示了两个有正在监听插口的 Telnet服务器：一个具有特定本地IP地址；另一个的本地 IP地址是通配地址。你的系统的 Telnet守护程序允许你指定本地IP地址吗？如果允许，如何指定？

22.3 假定某个插口被绑定到本地插口 {140.252.1.29, 8888}，且这是唯一使用本地插口8888的插口。 (1)当有另一个插口绑定到 {140.252.1.29，8888}时，请执行in_pcbbind的所有步骤，假定没有任何插口选项。 (2)当有另一个插口绑定到通配IP地址，端口8888时，执行in_pcbbind的所有步骤，假定没有任何插口选项。 (3)当有另一个插口绑定到通配 IP地址，端口 8888时，且设定了插口选项SO_REUSEADDR，执行in_pcbbind的所有步骤。

22.4 UDP分配的第一个临时端口号是什么？

22.5 当进程调用bind时，必须填充sockaddr_in结构中的哪一个元素？

22.6 如果进程要bind一个本地广播地址时，会发生什么情况？如果进程要 bind受限广播地址(255.255.255.255)时，会发生什么情况？

第23章 UDP：用户数据报协议

23.1 引言

用户数据报协议，即UDP，是一个面向数据报的简单运输层协议：进程的每次输出操作只产生一个UDP数据报，从而发送一个IP数据报。

进程通过创建一个Internet域内的SOCK_DGRAM类型的插口，来访问UDP。该类插口默认地称为无连接的(unconnected)。每次进程发送数据报时，必须指定目的IP地址和端口号。每次从插口上接收数据报时，进程可以从数据报中收到源IP地址和端口号。

我们在22.5节中提到，UDP插口也可以被连接到一个特殊的IP地址和端口号。这样，所有写到该插口上的数据报都被发往该目的地，而且只有来自该IP地址和端口号的数据报才被传给该进程。

本章讨论UDP的实现。

23.2 代码介绍

9个UDP函数在一个C文件中，2个UDP定义的头文件，如图23-1所示。

图23-2显示了6个主要的UDP函数与其他内核函数之间的关系。带阴影的椭圆是本章我们讨论的6个函数，另外还有其他3个函数是这6个函数经常调用的。

文　件	描　述
netinet/udp.h netinet/udp_var.h	udphdr结构定义 其他UDP定义
netinet/udp_usrreq.c	UDP函数

图23-1 本章中讨论的文件

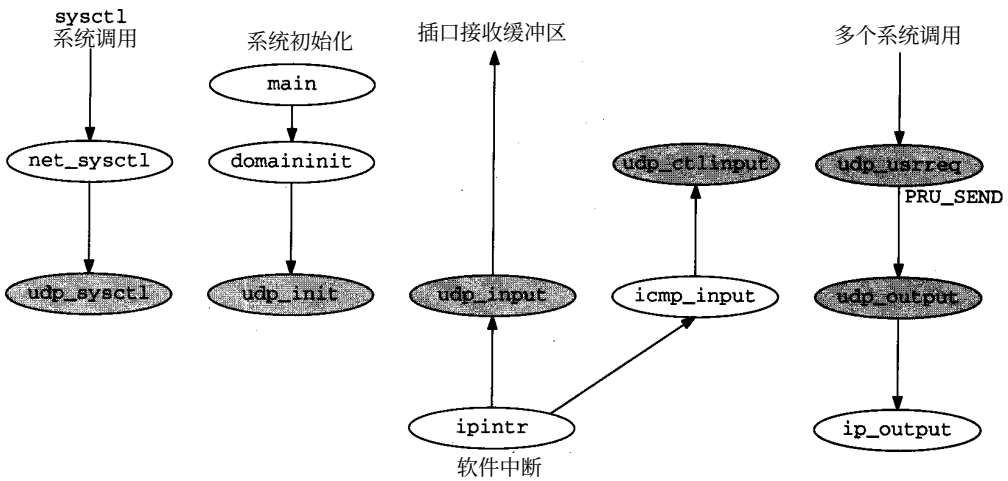


图23-2 UDP函数与内核其他函数之间的关系

23.2.1 全局变量

本章引入的全局变量，如图22-3所示。

变 量	数据类型	描 述
udb	Struct inpcb	UDP PCB表的表头
udp_last_inpcb	Struct inpcb *	指向最近收到的数据报的指针：“向后一个”高速缓存
udpcksum	Int	用于计算和验证UDP检验和的标志位
udp_in	Struct sockaddr_in	在输入时存放发送方的IP地址
udpstat	Struct udpstat	UDP统计(图23-4)
udp_recvspace	u_long	插口接收缓存的默认大小，41 600字节
udp_sendspace	u_long	插口发送缓存的默认大小，9 216字节

图23-3 本章中引入的全局变量

23.2.2 统计量

全局结构udpstat维护多种UDP统计量，如图23-4所示。讨论代码的过程中，我们会看到何时增加这些计数器的值。

udpstat成员	描 述	SNMP使用的
udps_badlen	收到所有数据长度大于分组的数据报个数	•
udps_badsum	收到有检验和错误的数据报个数	•
udps_fullsock	收到由于输入插口已满而没有提交的数据报个数	
udps_hdrops	收到分组小于首部的数据报个数	•
udps_ipackets	所有收到的数据报个数	•
udps_noport	收到在目的端口没有进程的数据报个数	•
udps_noportbcast	收到在目的端口没有进程的广播／多播数据报个数	•
udps_opackets	全部输出数据报的个数	•
udps_pcbcachemiss	收到的丢失pcb高速缓存的输入数据报个数	

图23-4 在udpstat 结构中维持的UDP统计

图23-5显示了执行netstat -s后输出的统计信息。

netstat -s 输出	udpstat 成员
18,575,142 datagrams received	udps_ipackets
0 with incomplete header	udps_hdrops
18 with bad data length field	udps_badlen
58 with bad checksum	udps_badsum
84,079 dropped due to no socket	udps_noport
446 broadcast/multicast datagrams dropped due to no socket	udps_noportbcast
5,356 dropped due to full socket buffers	udps_fullsock
18,485,185 delivered	(见正文)
18,676,277 datagrams output	udps_opackets

图23-5 UDP统计样本

提交的UDP数据报的个数(输出的倒数第二行)是收到的数据报总数(udps_ipackets)减去图23-5中它前面的6个变量。

23.2.3 SNMP变量

图23-6显示了UDP组中的四个简单SNMP变量，这四个变量在实现该变量的udpstat结构中计数。

图23-7显示了UDP监听器表，称为udpTable。SNMP为这个表返回的值是取自UDP PCB，而不是udpstat结构。

SNMP变量	udpstat成员	描　述
udpInDatagrams udpInErrors	udps_ipackets udps_hdrops + udps_badsum+ udps_badlen	收到的所有提交给进程的数据报个数 收到的由于某些原因不可提交的 UDP数据报个数，这些原因中不包括在目的端口没有应用程序的原因 (例如，UDP检验和差错)
udpNoPorts	udps_noport + udps_noportbcast	收到的所有目的端口没有应用进程的数据报
udpOutDatagrams	udps_opackets	发送的数据报的个数

图23-6 udp组中的简单SNMP变量

UDP监听器表，索引=<*udpLocalAddress*>.<*udpLocalPort*>		
SNMP变量	PCB变量	描　述
udpLocalAddress udpLocalPort	inp_laddr inp_lport	这个监听器的本地IP 这个监听器的本地端口号

图23-7 UDP监听器表：udpTable

23.3 UDP的protosw结构

图23-8显示了UDP的协议交换入口

成　员	inetsw[1]	描　述
pr_type	*SOCK_DGRAM*	UDP提供数据报分组服务
pr_domain	*&inetdomain*	UDP是Internet域的一部分
pr_protocol	*IPPROTO_UDP (17)*	出现在IP首部的ip_p字段
pr_flags	*PR_ATOMIC|PR_ADDR*	插口层标志，协议处理没有使用
pr_input	*Udp_input*	从IP层接收报文
pr_output	*0*	UDP没有使用
pr_ctlinput	*udp_ctlinput*	ICMP差错的控制输入函数
pr_ctloutput	*ip_ctloutput*	响应来自进程的管理请求
pr_usrreq	*udp_usrreq*	响应来自进程的通信请求
pr_init	*udp_init*	初始化UDP
pr_fasttimo	*0*	UDP没有使用
pr_slowtimo	*0*	UDP没有使用
pr_drain	*0*	UDP没有使用
pr_sysctl	*udp_sysctl*	对sysctl(8)系统调用

图23-8 UDP的protosw 结构

本章我们描述五个以udp_开头的函数。另外我们还要介绍第6个函数udp_output，它

不在协议交换入口，但当输出一个UDP数据报时，udp_usrreq会调用它。

23.4 UDP的首部

UDP首部定义成一个udphdr结构。图23-9是C结构，图23-10是UDP首部的图。

udp.h

```
39 struct udphdr {
40     u_short uh_sport;          /* source port */
41     u_short uh_dport;          /* destination port */
42     short   uh_ulen;           /* udp length */
43     u_short uh_sum;            /* udp checksum */
44 };
```

udp.h

图23-9 udphdr 结构

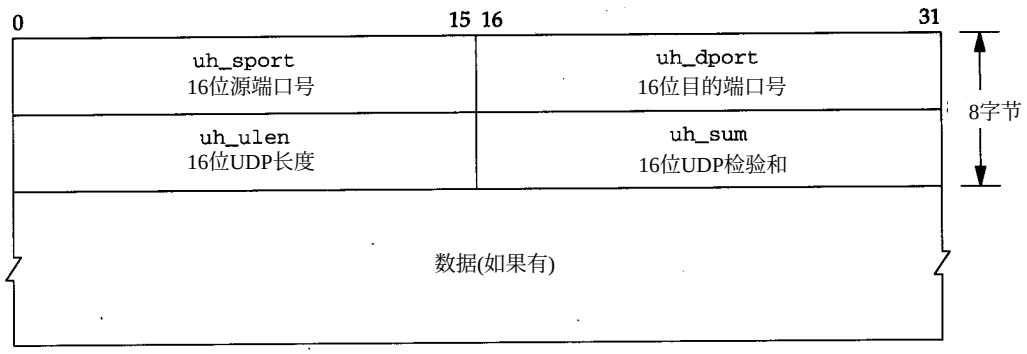


图23-10 UDP首部和可选数据

在源代码中，通常把UDP首部作为一个紧跟着UDP首部的IP首部来引用。这就是udp_input如何处理收到的IP数据报，以及udp_output如何构造外出的IP数据报。这种联合的IP/UDP首部是一个udpiphdr结构，如图23-11所示。

udp_var.h

```
38 struct udpiphdr {
39     struct ipovly ui_i;        /* overlaid ip structure */
40     struct udphdr ui_u;        /* udp header */
41 };
42 #define ui_next     ui_i.ih_next
43 #define ui_prev     ui_i.ih_prev
44 #define ui_x1       ui_i.ih_x1
45 #define ui_pr       ui_i.ih_pr
46 #define ui_len      ui_i.ih_len
47 #define ui_src      ui_i.ih_src
48 #define ui_dst      ui_i.ih_dst
49 #define ui_sport    ui_u.uh_sport
50 #define ui_dport    ui_u.uh_dport
51 #define ui_ulen     ui_u.uh_ulen
52 #define ui_sum      ui_u.uh_sum
```

udp_var.h

图23-11 udpiphdr 结构：联合的IP/UDP首部

20字节的IP首部定义成一个ipovly结构，如图23-12所示。

不幸的是，这个结构并不是一个真正的如图8-8所示的IP首部。大小相同(20字节)，但字

段不同。我们将在23.6节讲UDP检验和的计算时回来讨论这个不同之处。

ip_var.h

```
38 struct ipovly {
39     caddr_t ih_next, ih_prev;    /* for protocol sequence q's */
40     u_char  ih_x1;               /* (unused) */
41     u_char  ih_pr;               /* protocol */
42     short   ih_len;              /* protocol length */
43     struct in_addr ih_src;       /* source internet address */
44     struct in_addr ih_dst;       /* destination internet address */
45 };
```

ip_var.h

图23-12 ipovly 结构

23.5 udp_init函数

domaininit函数在系统初始化时调用UDP的初始化函数(udp_init，图23-13)。

这个函数所做的唯一的工作是把头部PCB(udb)的向前和向后指针指向它自己。这是一个双向链表。

udb PCB的其他部分都被初始化成 0，尽管在这个头部 PCB中唯一使用的字段是inp_lport，它是要分配的下一个UDP临时端口号。在解习题22.4时，我们提到，因为这个本地端口号被初始化成0，所以第一个临时端口号将是1024。

udp_usrreq.c

```
50 void
51 udp_init()
52 {
53     udb.inp_next = udb.inp_prev = &udb;
54 }
```

udp_usrreq.c

图23-13 udp_init 函数

23.6 udp_output函数

当应用程序调用以下五个写函数中的任意一个时，发生UDP 输出。这五个函数是：send、sendto、sendmsg、write或writev。如果插口已连接上的，则可任意调用这五个函数，尽管用sendto或sendmsg不能指定目的地址。如果插口没有连接上，则只能调用 sendto和sendmsg，并且必须指定一个目的地址。图23-14总结了这五个函数，它们在终止时，都调用udp_output，该函数再调用ip_output。

五个函数终止调用sosend，并把一个指向msghdr结构的指针作为参数传给该函数。要输出的数据被分装在一个mbuf链上，sosend把一个可选的目的地址和可选的控制信息放到mbuf中，并发布PRU_SEND请求。

UDP调用函数udp_output，该函数的第一部分如图23-15所示。四个参数分别是：inp，指向插口Internet PCB的指针；m，指向输出mbuf链的指针；addr，一个可选指针，指向某个mbuf，存放分装在一个sockaddr_in结构中的目的地址；control，一个可选指针，指向一个mbuf，其中存放着sendmsg中的控制信息。

1. 丢掉可选控制信息

333-344 m_freem丢弃可选的控制信息，不产生差错。UDP输出不使用任何控制信息。

注释*xxx*是因为忽略控制信息且不产生错误。其他协议如路由选择域和 TCP，当进程传递控制信息时，都会产生一个错误。

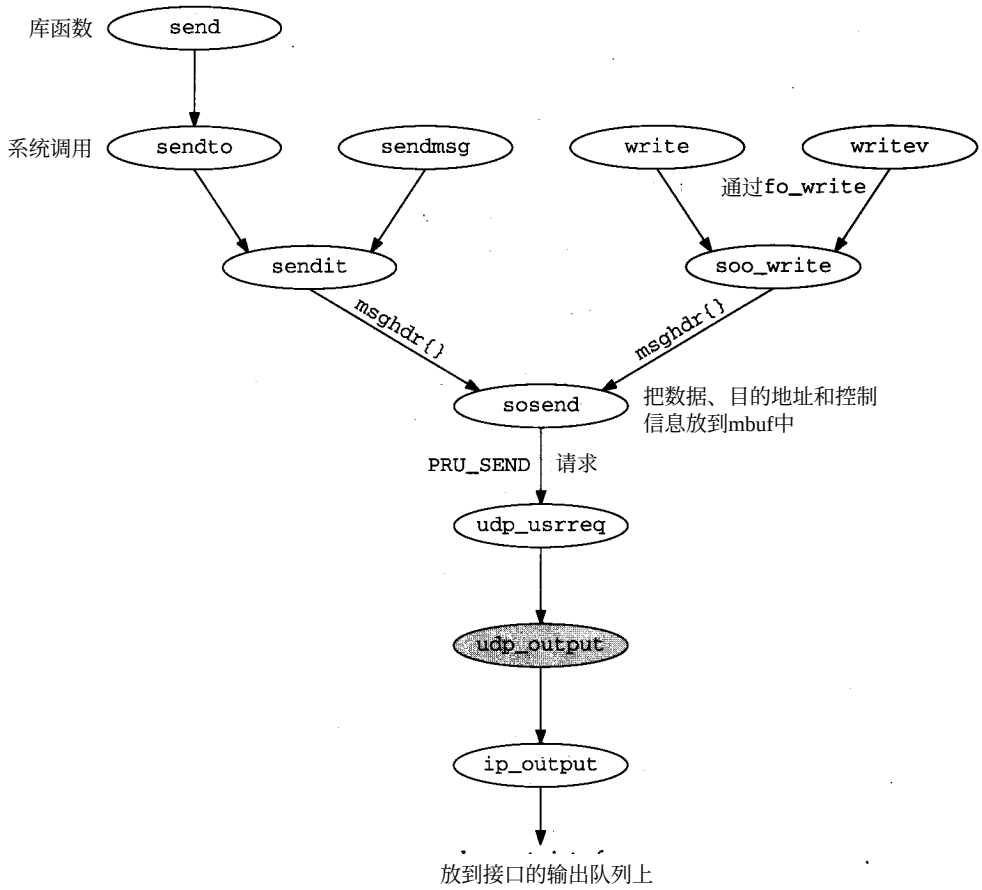


图23-14 五个写函数如何终止调用udp_output

2. 临时连接一个未连接上的插口

345-359 如果调用方为UDP数据报指定了目的地址(addr非空)，则插口是由in_pcbconnect临时连接到该目的地址的，并在该函数的最后被断连。在连接之前，要作一个检测，判断插口是否已经连接上。如果已连接上，则返回错误 EISCONN。这就是为什么sendto在已连接上的插口上指定目的地址时，会返回错误。

在临时连接插口之前，splnet停止IP的输入处理。这样做的原因是因为，临时连接将改变插口PCB中的外部地址、外部端口以及本地地址。如果在临时连接该 PCB的过程中处理某个收到的UDP数据报，可能把该数据报提交给错误的进程。把处理器设置成比 splnet优先，只能阻止软件中断引发执行IP输入程序(图1-12)，它不能阻止接口层接收进入的分组，并把它们放到IP的输入队列中。

[Partridge和Pink 1993] 注意到临时连接插口的这个操作开销很大，用去每个UDP传送将近三分之一的时间。

在临时连接之前，PCB的本地地址被保存在laddr中，因为如果它是通配地址，它将被

in_pcbconnect在调用in_pcbbind时改变。

如果进程调用了connect，则应用于目的地址的同一规则也将适用，因为两种情况都将调用in_pcbconnect。

udp_usrreq.c

```
333 int
334 udp_output(inp, m, addr, control)
335 struct inpcb *inp;
336 struct mbuf *m;
337 struct mbuf *addr, *control;
338 {
339     struct udpiphdr *ui;
340     int     len = m->m_pkthdr.len;
341     struct in_addr laddr;
342     int     s, error = 0;
343     if (control)
344         m_freem(control);       /* XXX */
345     if (addr) {
346         laddr = inp->inp_laddr;
347         if (inp->inp_faddr.s_addr != INADDR_ANY) {
348             error = EISCONN;
349             goto release;
350         }
351         /*
352          * Must block input while temporarily connected.
353          */
354         s = splnet();
355         error = in_pcbconnect(inp, addr);
356         if (error) {
357             splx(s);
358             goto release;
359         }
360     } else {
361         if (inp->inp_faddr.s_addr == INADDR_ANY) {
362             error = ENOTCONN;
363             goto release;
364         }
365     }
366     /*
367      * Calculate data length and get an mbuf for UDP and IP headers.
368      */
369     M_PREPEND(m, sizeof(struct udpiphdr), M_DONTWAIT);
370     if (m == 0) {
371         error = ENOBUFS;
372         goto release;
373     }
        /* remainder of function shown in Figure 23.20 */
409   release:
410     m_freem(m);
411     return (error);
412 }
```

udp_usrreq.c

图23-15 udp_output 函数：临时连接一个未连接上的插口

360-364 如果进程没有指定目的地址，并且插口没有连接上，则返回 ENOTCONN。

3. 在前面加上IP/UDP首部

366-373 M_PREPEND在数据的前面为IP和UDP首部分配空间。图 1-8是一种情况，假定mbuf链上的第一个 mbuf已经没有空间存放首部的 28个字节。习题23.1详细给出了其他情况。需要指定标志位 M_DONTWAIT，因为如果插口是临时连接的，则 IP处理被阻塞，所以M_PREPEND也应被阻塞。

早期的伯克利版本不正确地指定了这里的 M_WAIT。

23.6.1 在前面加上IP/UDP首部和mbuf簇

在M_PREPEND宏和mbuf簇之间有一个微妙的交互。如果 sosend把用户数据放到一个簇中，则该簇的最前面的 56个字节(max_hdr，图7-17)没有使用，这就为以太网、 IP和UDP首部提供了空间。避免 M_PREPEND仅仅为存放这些首部而另外再分配一个 mbuf。M_PREPEND调用M_LEADINGSPACE来计算在mbuf的前面有多大的空间可以使用：

```
#define M_LEADINGSPACE(m) \
   ((m)->m_flags & M_EXT ? /* (m)->m_data - (m)->m_ext_buf */ 0 : \
        (m)->m_flags & M_PKTHDR ? (m)->m_data - (m)->m_pktdat : \
        (m)->m_data - (m)->m_dat)
```

正确地计算出簇前面可用空间大小的代码被注释掉了，如果数据在簇内，该宏总是返回 0。这意味着，当用户数据也在某个簇中时， M_PREPEND总是为协议首部分配一个新的mbuf，而不再使用sosend分配的用于存放首部的空间。

M_LEADINGSPACE中注释掉正确代码的原因是因为该簇可能被共享 (2.9节)，而且，如果它被共享，使用簇中数据报前面的空间可能会擦掉其他数据。

UDP数据不共享簇，因为 udp_output不保存数据的备份。但是 TCP在它的发送缓存内保存数据备份(等待对该数据的确认)，而且如果数据不在簇内，则说明它是共享的。但tcp_output不调用M_LEADINGSPACE，因为sosend只为数据报协议在簇前面留56个字节，所以，tcp_output总是调用MGETHDR为协议首部分配一个mbuf。

23.6.2 UDP检验和计算和伪首部

在讨论udp_output的后一部分之前，我们描述一下 UDP如何填充IP/UDP首部的某些字段，如何计算UDP检验和，以及如何传递 IP/UDP首部及数据给IP输出的。这些工作很巧妙地使用了ipovly结构。

图23-16显示了udp_output在由m指向的mbuf链的第一个存储器上构造的28字节IP/UDP首部。没有阴影的字段是udp_output填充的，有阴影的字段是ip_output填充的。这个图显示了首部在线路上的格式。

UDP检验和的计算覆盖了三个区域： (1)一个12字节的伪首部，其中包含 IP首部的字段；(2)8 字节UDP首部，和(3)UDP数据。图23-17显示了用于检验和计算的 12字节伪首部，以及UDP首部。用于计算检验和的UDP首部等价于出现在线路上的UDP首部(图23-16)。

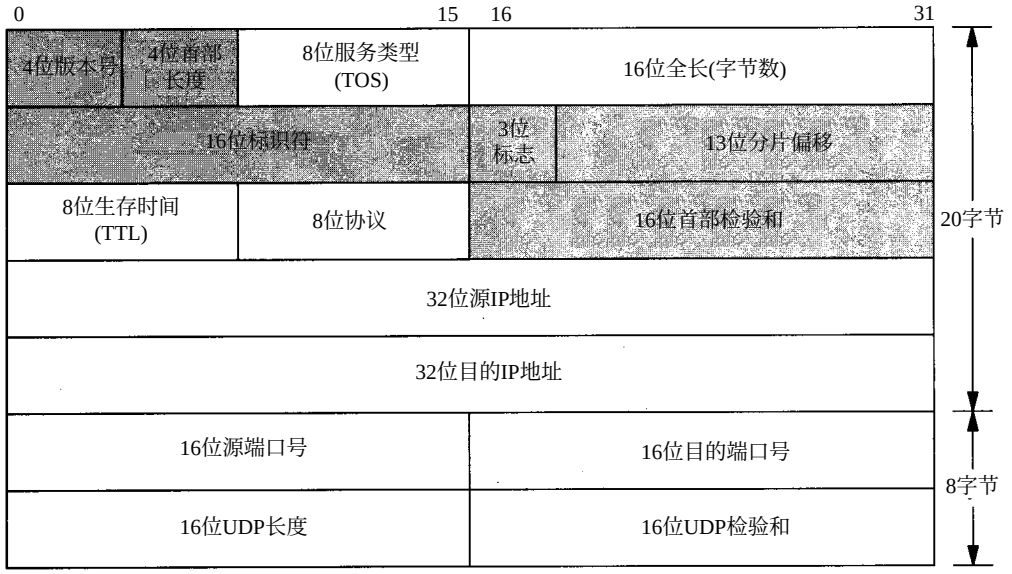


图23-16 IP/UDP首部：UDP填充没有阴影的字段，IP填充有阴影的字段

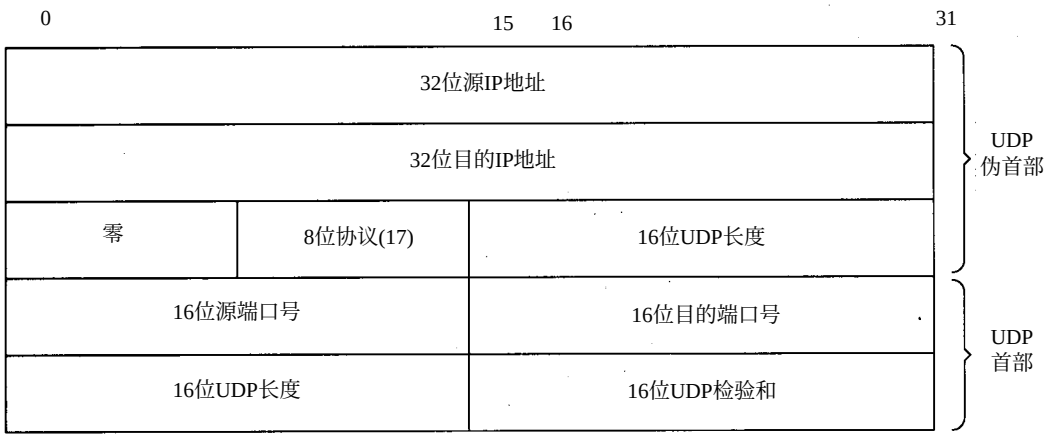


图23-17 检验和计算所使用的伪首部和UDP首部

在计算UDP检验和时使用以下三个事实：(1)在伪首部(图23-17)中的第三个32 bit字看起来与IP首部(图23-16)中的第三个32 bit字类似：两个8 bit值和一个16 bit值。(2)伪首部中三个32 bit值的顺序是无关的。事实上，Internet检验和的计算不依赖于所使用的16 bit值的顺序(8.7节)。(3)在检验和计算中另外再加上一个全0的32 bit字没有任何影响。

`udp_output`利用这三个事实，填充`udpiphdr`结构(图23-11)的字段，如图23-18所示。该结构包含在由`m`指向的mbuf链的第一个mbuf中。

在20字节IP首部(5个成员：`ui_x1`、`ui_pr`、`ui_len`、`ui_src`和`ui_dst`)中的最后三个32 bit字被用作检验和计算的伪首部。IP首部的前两个32 bit字(`ui_next`和`ui_prev`)也用在检验和计算中，但它们被初始化成0，所以不影响最后的检验和。

图23-19总结了我们描述的操作：

1) 图23-19中最上面的图是伪首部的协议定义，与图23-17对应。

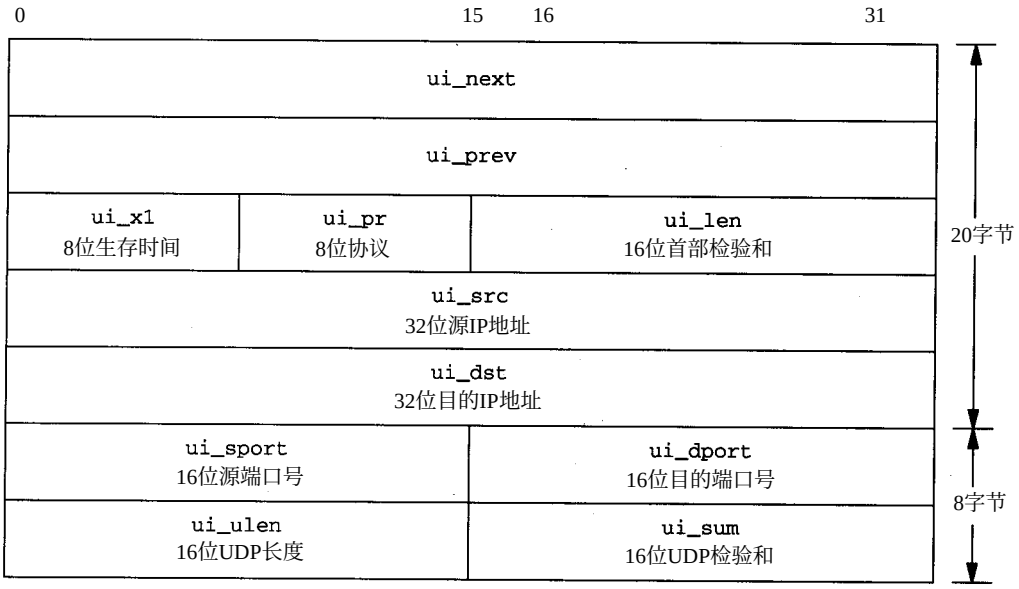


图23-18 udp_output 填充的udpiphdr

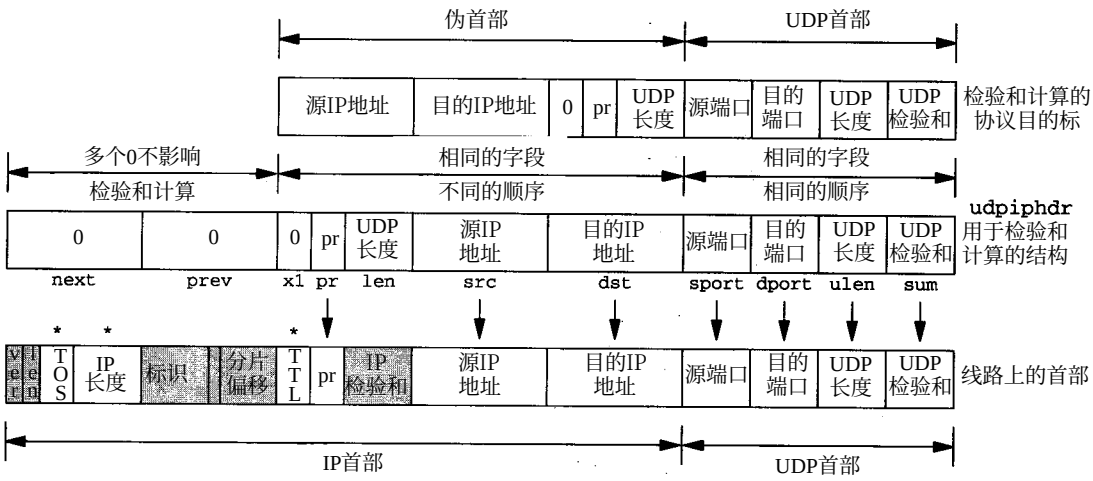


图23-19 填充IP/UDP首部和计算UDP检验和的操作

2) 中间的图是源代码使用的udpiphdr结构，与图23-11对应(为图的可读性，省略了所有成员的前缀ui_)。这是udp_output在mbuf链上的第一个mbuf中构造的结构，然后被用于计算UDP检验和。

3) 下面的图是出现在线路上的IP/UDP首部，与图23-16对应。上面有箭头的7个字段是udp_output在检验和计算之前填充的。上面有星号的3个字段是udp_output在检验和计算之后填充的。其他6个有阴影的字段是ip_output填充的。

图23-20是udp_output函数的后半部分。

1. 为检验和计算准备伪首部

374-387 把udpiphdr结构(图23-18)的所有成员设置成它们相应的值。PCB中的本地和外部插口已经是网络字节序，但必须把UDP的长度转换成网络字节序。UDP的长度是数据报的

字节数(len，可以是0)加上UDP 首部的大小(8)。UDP 长度字段在UDP 检验和计算中出现了两次：ui_len和ui_ulen。有一个是冗余的。

udp_usrreq.c

```
374     /*
375      * Fill in mbuf with extended UDP header
376      * and addresses and length put into network format.
377      */
378     ui = mtod(m, struct udpiphdr *);
379     ui->ui_next = ui->ui_prev = 0;
380     ui->ui_x1 = 0;
381     ui->ui_pr = IPPROTO_UDP;
382     ui->ui_len = htons((u_short) len + sizeof(struct udphdr));
383     ui->ui_src = inp->inp_laddr;
384     ui->ui_dst = inp->inp_faddr;
385     ui->ui_sport = inp->inp_lport;
386     ui->ui_dport = inp->inp_fport;
387     ui->ui_ulen = ui->ui_len;

388     /*
389      * Stuff checksum and output datagram.
390      */
391     ui->ui_sum = 0;
392     if (udpcksum) {
393         if ((ui->ui_sum = in_cksum(m, sizeof(struct udpiphdr) + len)) == 0)
394                 ui->ui_sum = 0xffff;
395     }
396     ((struct ip *) ui)->ip_len = sizeof(struct udpiphdr) + len;
397     ((struct ip *) ui)->ip_ttl = inp->inp_ip.ip_ttl;     /* XXX */
398     ((struct ip *) ui)->ip_tos = inp->inp_ip.ip_tos;     /* XXX */
399     udpstat.udps_opackets++;
400     error = ip_output(m, inp->inp_options, &inp->inp_route,
401             inp->inp_socket->so_options & (SO_DONTROUTE | SO_BROADCAST),
402                     inp->inp_moptions);

403     if (addr) {
404         in_pcbdisconnect(inp);
405         inp->inp_laddr = laddr;
406         splx(s);
407     }
408     return (error);
```

udp_usrreq.c

图23-20 udp_output 函数：填充首部、计算检验和并传给IP

2. 计算检验和

388-395 计算检验和时，首先把它设成0，然后调用in_cksum。如果UDP检验和是禁止的(一个坏的想法 —— 见卷1的11.3节)，则检验和的结果是0。如果计算的检验和是0，则在首部中保存16个1，而不是0(在求补运算中，全1和全0都是0)。这样，接收方就可以区分没有检验和的UDP分组(检验和字段为0)和有检验和的UDP分组了，后者的检验和值为0(16位的检验和是16个1)。

变量udpcksum(图23-3)通常默认值为1，使能UDP检验和。对内核的编译可对4.2BSD兼容，把udpcksum初始化为0。

3. 填充UDP长度、TTL和TOS

396-398 指针ui指向一个指向某个标准IP首部的指针(ip)，UDP设置IP首部内的三个字段。IP长度字段等于UDP数据报中数据的个数加上IP/UDP首部大小28。注意，IP首部的这个字段

以主机字节序保存，不象首部其他多字节字段，是以网络字节序保存的。 ip_output在发送之前，把它转换成网络字节序。

把IP首部里的TTL和TOS字段的值设成插口PCB中的值。在创建插口时， UDP设置这些默认值，进程可用setsockopt改变它们。因为这三个字段 —— IP长度、TTL和TOS —— 不是伪首部的内容， UDP检验和计算时也没有用到它们，所以，在计算检验和之后，调用ip_output之前，必须设置它们。

4. 发送数据报

400-402 ip_output发送数据报。第二个参数inp_options，是进程可用setsockopt设置的IP选项。这些IP选项是ip_output放置到IP首部中的。第三个参数是一个指向高速缓存在PCB中的路由的指针，第四个参数是插口选项。传给 ip_output的唯一插口选项是SO_DONTROUTE(旁路选路表)和SO_BROADCAST(允许广播)。最后一个参数是一个指向该插口的多播选项的指针。

5. 断连临时连接的插口

403-407 如果插口是临时连接上的，则 in_pcbdisconnect断连插口，本地IP地址在PCB中恢复，恢复中断级别到保存的值。

23.7 udp_input函数

进程调用五个写函数之一来驱动UDP 输出。图23-14显示的函数都作为系统调用的组成部分被直接调用。另一方面，当 IP在它的协议字段指定为 UDP的输入队列上收到一个 IP数据报时，才发生 UDP的输入。 IP通过协议交换表 (图8-15)中的pr_input函数调用函数udp_input。因为IP的输入是在软件中断级，所以 udp_input也在这一级上执行。udp_input的目标是把UDP数据报放置到合适的插口的缓存内，唤醒该插口上因输入阻塞的所有进程。

我们对udp_input函数的讨论分三个部分：

1) UDP对收到的数据报完成一般性的确认；

2) 处理目的地是单播地址的UDP数据报：找到合适的PCB，把数据报放到插口的缓存内，

3) 处理目的地是广播或多播地址的UDP数据报：必须把数据报提交给多个插口。

最后一步是新的，是为了在Net/3中支持多播，但占用了大约三分之一的代码。

23.7.1 对收到的UDP数据报的一般确认

图23-21是UDP输入的第一部分。

55-65 udp_input的两个参数是： m，一个指向包含了该 IP数据报的mbuf链的指针；iphlen，IP首部的长度(包括可能的IP选项)。

1. 丢弃IP选项

67-76 如果有IP选项，则ip_stripoptions丢弃它们。正如注释中表明的， UDP 应该保存IP选项的一个备份，使接收进程可以通过 IP_RECVOPTS插口选项访问到它们，但这个还没有实现。

77-88 如果 mbuf链上的第一个 mbuf小于 28字节 (IP首部加上 UDP首部的大小)，则m_pullup重新安排mbuf链，使至少有28个字节连续地存放在第一个mbuf中。

udp_usrreq.c

```
55 void
56 udp_input(m, iphlen)
57 struct mbuf *m;
58 int     iphlen;
59 {
60     struct ip *ip;
61     struct udphdr *uh;
62     struct inpcb *inp;
63     struct mbuf *opts = 0;
64     int     len;
65     struct ip save_ip;
66     udpstat.udps_ipackets++;
67     /*
68      * Strip IP options, if any; should skip this,
69      * make available to user, and use on returned packets,
70      * but we don't yet have a way to check the checksum
71      * with options still present.
72      */
73     if (iphlen > sizeof(struct ip)) {
74         ip_stripoptions(m, (struct mbuf *) 0);
75         iphlen = sizeof(struct ip);
76     }
77     /*
78      * Get IP and UDP header together in first mbuf.
79      */
80     ip = mtod(m, struct ip *);
81     if (m->m_len < iphlen + sizeof(struct udphdr)) {
82         if ((m = m_pullup(m, iphlen + sizeof(struct udphdr))) == 0) {
83             udpstat.udps_hdrops++;
84             return;
85         }
86         ip = mtod(m, struct ip *);
87     }
88     uh = (struct udphdr *) ((caddr_t) ip + iphlen);
89     /*
90      * Make mbuf data length reflect UDP length.
91      * If not enough data to reflect UDP length, drop.
92      */
93     len = ntohs((u_short) uh->uh_ulen);
94     if (ip->ip_len != len) {
95         if (len > ip->ip_len) {
96             udpstat.udps_badlen++;
97             goto bad;
98         }
99         m_adj(m, len - ip->ip_len);
100        /* ip->ip_len = len; */
101    }
102    /*
103     * Save a copy of the IP header in case we want to restore
104     * it for sending an ICMP error message in response.
105     */
106    save_ip = *ip;
107    /*
108     * Checksum extended UDP header and data.
109     */
110    if (udpcksum && uh->uh_sum) {
```

图23-21 udp_input 函数：对收到的UDP数据报的一般确认

```
111         ((struct ipovly *) ip)->ih_next = 0;
112         ((struct ipovly *) ip)->ih_prev = 0;
113         ((struct ipovly *) ip)->ih_x1 = 0;
114         ((struct ipovly *) ip)->ih_len = uh->uh_ulen;
115         if (uh->uh_sum = in_cksum(m, len + sizeof(struct ip))) {
116             udpstat.udps_badsum++;
117             m_freem(m);
118             return;
119         }
120     }
```

udp_usrreq.c

图23-21 (续)

2. 验证UDP长度

333-344 与UDP 数据报相关的两个长度是：IP首部的长度字段(ip_len)和UDP首部的长度字段(uh_ulen)。前面讲到，ipintr在调用udp_input之前，从ip_len中抽取出IP首部的长度(图10-11)。比较这两个长度，可能有三种可能性：

1) ip_len等于uh_ulen。这是通常情况。

2) ip_len大于uh_ulen。IP首部太大，如图23-22所示。代码相信两个长度中小的那个(UDP首部长度)，m_adj从数据报的最后移走多余的数据字节。m_adj的第二个参数是负数，在图2-20中我们说，从mbuf链的最后截断数据。在这种情况下，UDP的长度字段出现冲突。如果是这样，假定发送方计算了UDP的检验和，则不久检验和会检测到这个错误，接收方也会验证检验和，从而丢弃该数据报。IP长度字段必须正确，因为IP根据接口上收到的数据量验证它，而强制的IP首部检验和覆盖了IP首部的长度字段。

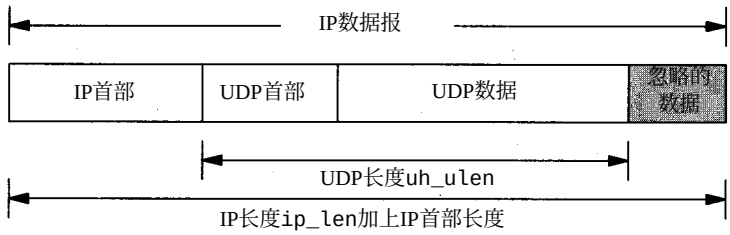


图23-22 UDP长度太小

3) ip_len小于uh_ulen。当UDP 首部的长度给定时，IP数据报比可能的小。图23-23显示了这种情况。这说明数据报有错误，必须丢弃，没有其他的选择：如果 UDP长度字段被破坏，用UDP检验和是无法检测到的。需要用正确的UDP长度来计算UDP检验和。

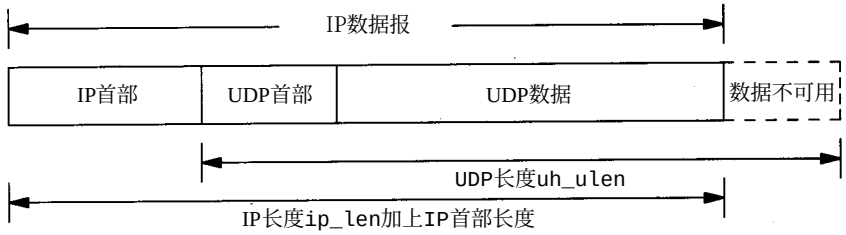


图23-23 UDP长度太大

正如我们提到的，UDP长度是冗余的。在第28章中我们将看到，TCP自己的首部内没有长度字段 —— 它用IP长度字段减去IP和TCP首部的长度，以此确定数据报内数

据的数量。为什么存在 UDP长度字段呢？可能是为了加上少量的差错检测，因为UDP检验和是可选的。

3. 保存IP首部的备份，验证UDP检验和

102-106 udp_input在验证检验和之前保存IP首部的备份，因为检验和计算会擦去原始 IP首部的一些字段。

110 只有当的UDP检验和(udpcksum)是内核允许的，并且发送方也计算了 UDP检验和(收到的检验和不为0)时，才验证检验和。

这个检测是不正确的。如果发送方计算了一个检验和，就应该验证它，不管外出的检验和是否被计算。变量 udpcksum应该只指定是否计算外出的检验和。不幸的是，许多厂商都复制了这个检测，尽管厂商已经改变它们产品的内核，却默认地允许UDP检验和。

111-120 在计算检验和之前， IP首部作为ipovly结构(图23-18)引用，所有字段的初始化都是udp_output在计算UDP检验和(上一节)时初始化的。

此时，如果数据报的目的地是一个广播或多播 IP地址，将执行特别的代码。我们把这段代码推迟到本节最后。

23.7.2 分用单播数据报

假定数据报的目的地是一个单播地址，图 23-24显示了执行的代码。

udp_usrreq.c

```
        /* demultiplex broadcast & multicast datagrams (Figure 23.26) */
206     /*
207      * Locate pcb for unicast datagram.
208      */
209     inp = udp_last_inpcb;
210     if (inp->inp_lport != uh->uh_dport ||
211         inp->inp_fport != uh->uh_sport ||
212         inp->inp_faddr.s_addr != ip->ip_src.s_addr ||
213         inp->inp_laddr.s_addr != ip->ip_dst.s_addr) {
214         inp = in_pcblookup(&udb, ip->ip_src, uh->uh_sport,
215                            ip->ip_dst, uh->uh_dport, INPLOOKUP_WILDCARD);
216         if (inp)
217             udp_last_inpcb = inp;
218         udpstat.udpps_pcbcachemiss++;
219     }
220     if (inp == 0) {
221         udpstat.udps_noport++;
222         if (m->m_flags & (M_BCAST | M_MCAST)) {
223             udpstat.udps_noportbcast++;
224             goto bad;
225         }
226         *ip = save_ip;
227         ip->ip_len += iphlen;
228         icmp_error(m, ICMP_UNREACH, ICMP_UNREACH_PORT, 0, 0);
229         return;
230     }
```

udp_usrreq.c

图23-24 udp_input 函数：分用单播数据报

1. 检查“向后一个”高速缓存

206-209 UDP维护一个指针，该指针指向最后在其上接收数据报的Internet PCB，udp_last_inpcb。在调用in_pcblookup之前，可能必须搜索UDP表上的PCB，把最近一次接收PCB的外部和本地地址以及端口号和收到数据报的进行比较。这称为“向后一个”高速缓存(one-behind cache)[Partridge和Pink 1993]。它是根据这样一个假设，即收到的数据报极有可能要发往上一个数据报发往的同一端口[Mogul 1991]。这个高速缓存技术是在4.3BSD Tahoe版本中引入的。

210-213 高速缓存的PCB和收到数据报之间的四个比较的次序是故意安排的。如果PCB不匹配，则应尽快结束比较。最大的可能性是目的端口号不相同——这就是为什么第一个检测它。不匹配的可能性最小的是本地地址，尤其在只有一个接口的主机，所以它是最后一个检测。

不幸的是，这种“向后一个”高速缓存技术代码，在实际中毫无用处[Partridge和Pink 1993]。最普通的UDP服务器类型只绑定它的知名端口，它的本地地址、外部地址和外部端口都是通配地址。最普通的UDP客户程序类型并不连接它的UDP插口；它用sendto指定每个数据报的目的地址。因此，大多数时间PCB内的inp_laddr、inp_faddr和inp_fport都是通配的。在高速缓存的比较中，收到数据报的这四个值永远都不是通配的，这意味着只有当指定PCB的四个本地和外部值是非通配时，高速缓存入口与收到数据报的比较才可能相等。这种情况只在连接上的UDP插口上发生。

在系统bsdi上，udpps_pcbcachemiss计数器是41 253，udps_ipackets计数器是42 485。小于3%缓存命中率。

netstat-s命令打印出udpstat结构(图23-5)的大多数字段。不幸的是，Net/3版本，以及多数厂家的版本都不打印udpps_pcbcachemiss。如果你想看它们的值，用调试器检查在运行的内核中的变量。

2. 搜索所有UDP的PCB

214-218 假定与高速缓存的比较失败，则in_pcblookup寻找一个匹配。指定INPLOOKUP_WILDCARD标志，允许通配匹配。如果找到一个匹配，则把指向该PCB的指针保存在udp_last_inpcb中，我们说它高速缓存了最后收到的UDP数据报的PCB。

3. 生成ICMP端口不可达差错

220-230 如果没找到匹配的PCB，UDP通常产生一个ICMP端口不可达差错。首先检测收到的mbuf链的m_flags，看看该数据报是否是要发送到一个链路级广播或多播地址。有可能会收到一个发送到链路级广播或多播地址的IP数据报，具有单播地址，此时不应该产生ICMP端口不可达差错。如果成功产生ICMP差错，则把IP首部恢复成收到它时的值(save_ip)，也把IP长度设置成它原来的值。

链路级广播或多播地址的检测是冗余的。icmp_error也做这个检测。这个冗余检测的唯一好处是，在udps_noportbcast计数器之外，还维护了udps_noport计数器。

把iphlen改回ip_len是一个错误。icmp_error也会做这项工作，使得ICMP差错返回的IP首部的IP长度字段是20字节，这太大了。可以在Traceroute程

序(卷1的第8章)中加上几行新程序，在最终到达目的主机后，打印出 ICMP端口不可达差错报文中的这个字段，可以测试系统是否有这个错误。

图23-25是处理单播数据报的代码，把数据报提交给与目的 PCB对应的插口。

udp_usrreq.c

```
231     /*
232      * Construct sockaddr format source address.
233      * Stuff source address and datagram in user buffer.
234      */
235     udp_in.sin_port = uh->uh_sport;
236     udp_in.sin_addr = ip->ip_src;

237     if (inp->inp_flags & INP_CONTROLOPTS) {
238         struct mbuf **mp = &opts;

239         if (inp->inp_flags & INP_RECVDSTADDR) {
240             *mp = udp_saveopt((caddr_t) & ip->ip_dst,
241                               sizeof(struct in_addr), IP_RECVDSTADDR);
242             if (*mp)
243                 mp = &(*mp)->m_next;
244         }
245 #ifdef notyet
246         /* IP options were tossed above */
247         if (inp->inp_flags & INP_RECVOPTS) {
248             *mp = udp_saveopt((caddr_t) opts_deleted_above,
249                               sizeof(struct in_addr), IP_RECVOPTS);
250             if (*mp)
251                 mp = &(*mp)->m_next;
252         }
253         /* ip_srcroute doesn't do what we want here, need to fix */
254         if (inp->inp_flags & INP_RECVRETOPTS) {
255             *mp = udp_saveopt((caddr_t) ip_srcroute(),
256                               sizeof(struct in_addr), IP_RECVRETOPTS);
257             if (*mp)
258                 mp = &(*mp)->m_next;
259         }
260 #endif
261     }
262     iphlen += sizeof(struct udphdr);
263     m->m_len -= iphlen;
264     m->m_pkthdr.len -= iphlen;
265     m->m_data += iphlen;
266     if (sbappendaddr(&inp->inp_socket->so_rcv, (struct sockaddr *) &udp_in,
267                      m, opts) == 0) {
268         udpstat.udps_fullsock++;
269         goto bad;
270     }
271     sorwakeup(inp->inp_socket);
272     return;

273   bad:
274     m_freem(m);
275     if (opts)
276         m_freem(opts);
277 }
```

udp_usrreq.c

图23-25 udp_input 函数：把单播数据报提交给插口

4. 返回源站IP地址和源站端口

231-236 收到的IP数据报的源站IP地址和源站端口被保存在全局 sockaddr_in结构中的

udp_in。在函数的后面，该结构作为参数传给了sbappendaddr。

采用全局变量保存IP地址和端口号不出现问题的原因是，udp_input是单线程的。当ipintr调用它时，它在返回之前完整地处理了收到的数据报。而且，sbappendaddr还把该插口结构从全局变量复制一个mbuf中。

5. IP_RECVDSTADDR插口选项

337-244 常数INP_CONTROLOPTS是三个插口选项的结合，进程可以设置这三个插口选项，通过系统调用recvmsg返回插口的控制信息(图22-5)。IP_RECVDSTADDR把收到的UDP数据报中的目的IP地址作为控制信息返回。函数udp_saveopt分配一个MT_CONTROL类型的mbuf，并把4字节的目的IP地址存放在该缓存中。我们在23.8节中介绍这个函数。

该插口选项与4.3BSD Reno一起出现，是为一般文件传输协议TFTP的应用程序设计的，它们不响应发给广播地址的客户程序请求。不幸的是，即使接收应用程序使用这个选项，也很难确定目的IP地址是否是一个广播地址(习题23.6)。

当4.4BSD中加上了多播功能后，这个代码只对目的地是单播地址的数据报有效。我们将在图23-26看到，对发给多播地址的广播数据报还没有实现这个选项，根本无法达到该选项的目的。

6. 未实现的插口选项

245-260 这段代码被注释掉了，因为它们不起作用。IP_RECVOPTS插口选项的原意是把收到数据报的IP选项作为控制信息返回，而IP_RECVRETOPTS插口选项返回源路由信息。三个IP_RECV插口选项对mp变量的操作构造了一个最多有三个mbuf的链表，该链表由sbappendaddr放置到插口的缓存。图23-25显示的代码只把一个选项作为控制信息返回，所以指向该mbuf的m_next总是一个空指针。

7. 把数据加到插口的接收队列中

262-272 此时，已经准备好把收到的数据报(m指向的mbuf链)以及一个表示发送方IP地址和端口的插口地址结构(udp_in)和一些可选的控制信息(opts指向的mbuf，目的IP地址)放到插口的接收队列中。这个工作由sbappendaddr完成。但在调用这个函数之前，要修正指针和缓存链上的第一个mbuf，忽略掉UDP和IP首部。返回之前，调用sorwakeup唤醒插口接收队列中的所有睡眠进程。

8. 返回差错

273-276 如果在UDP输入处理的过程中遇到错误，udp_input会跳转到bad标号语句，释放所有包含该数据报以及控制信息(如果有的话)的mbuf链。

23.7.3 分用多播和广播数据报

现在返回到udp_input处理发给广播或多播IP地址数据报的这部分代码。如图23-26所示。

121-138 正如注释所表明的，这些数据报被提交给匹配的所有插口，而不仅仅是一个插口。我们提到的UDP接口不够指的是除非连接上插口，否则进程没有能力在UDP插口上接收异步差错(特别是ICMP端口不可达差错)。我们22-11节讨论这个问题。

139-145 源站的IP地址和端口号被保存在全局sockaddr_in结构的udp_in中，传给sbappendaddr。更新mbuf链的长度和数据指针，忽略UDP和IP首部。

udp_usrreq.c

```
121     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr)) ||
122         in_broadcast(ip->ip_dst, m->m_pkthdr.rcvif)) {
123         struct socket *last;
124         /*
125          * Deliver a multicast or broadcast datagram to *all* sockets
126          * for which the local and remote addresses and ports match
127          * those of the incoming datagram.  This allows more than
128          * one process to receive multi/broadcasts on the same port.
129          * (This really ought to be done for unicast datagrams as
130          * well, but that would cause problems with existing
131          * applications that open both address-specific sockets and
132          * a wildcard socket listening to the same port -- they would
133          * end up receiving duplicates of every unicast datagram.
134          * Those applications open the multiple sockets to overcome an
135          * inadequacy of the UDP socket interface, but for backwards
136          * compatibility we avoid the problem here rather than
137          * fixing the interface.  Maybe 4.5BSD will remedy this?)
138          */
139         /*
140          * Construct sockaddr format source address.
141          */
142         udp_in.sin_port = uh->uh_sport;
143         udp_in.sin_addr = ip->ip_src;
144         m->m_len -= sizeof(struct udpiphdr);
145         m->m_data += sizeof(struct udpiphdr);
146         /*
147          * Locate pcb(s) for datagram.
148          * (Algorithm copied from raw_intr().)
149          */
150         last = NULL;
151         for (inp = udb.inp_next; inp != &udb; inp = inp->inp_next) {
152             if (inp->inp_lport != uh->uh_dport)
153                 continue;
154             if (inp->inp_laddr.s_addr != INADDR_ANY) {
155                 if (inp->inp_laddr.s_addr !=
156                     ip->ip_dst.s_addr)
157                     continue;
158             }
159             if (inp->inp_faddr.s_addr != INADDR_ANY) {
160                 if (inp->inp_faddr.s_addr !=
161                     ip->ip_src.s_addr ||
162                     inp->inp_fport != uh->uh_sport)
163                     continue;
164             }
165             if (last != NULL) {
166                 struct mbuf *n;
167                 if ((n = m_copy(m, 0, M_COPYALL)) != NULL) {
168                     if (sbappendaddr(&last->so_rcv,
169                                      (struct sockaddr *) &udp_in,
170                                      n, (struct mbuf *) 0) == 0) {
171                         m_freem(n);
172                         udpstat.udps_fullsock++;
173                     } else
174                         sorwakeup(last);
175                 }
176             }
177             last = inp->inp_socket;
```

图23-26 udp_input 函数：分用广播或多播数据报

```
178             /*
179              * Don't look for additional matches if this one does
180              * not have either the SO_REUSEPORT or SO_REUSEADDR
181              * socket options set.  This heuristic avoids searching
182              * through all pcbs in the common case of a non-shared
183              * port.  It assumes that an application will never
184              * clear these options after setting them.
185              */
186             if ((last->so_options & (SO_REUSEPORT | SO_REUSEADDR) == 0))
187                 break;
188         }
189         if (last == NULL) {
190             /*
191              * No matching pcb found; discard datagram.
192              * (No need to send an ICMP Port Unreachable
193              * for a broadcast or multicast datgram.)
194              */
195             udpstat.udps_noportbcast++;
196             goto bad;
197         }
198         if (sbappendaddr(&last->so_rcv, (struct sockaddr *) &udp_in,
199                          m, (struct mbuf *) 0) == 0) {
200             udpstat.udps_fullsock++;
201             goto bad;
202         }
203         sorwakeup(last);
204         return;
205     }
```

udp_usrreq.c

图23-26 (续)

146-164 大的for循环扫描每个 UDP PCB，寻找所有匹配 PCB。这种分用不调用in_pcblookup，因为它只返回一个PCB，而广播或多播数据报可能需要提交给多个 PCB。

如果PCB的本地端口和收到数据报的本地端口不匹配，则忽略该入口。如果 PCB的本地端口不是通配地址，则把它和目的 IP地址比较，如果不相等则跳过该入口。如果 PCB内的外部地址不是通配地址，就把它和源站 IP地址比较，如果不匹配，则外部端口也必须和源站端口匹配。最后一个检测假定，如果插口连接到某个外部 IP地址，则它也必须连接到一个外部端口，反之亦然。这与in_pcblookup函数的逻辑相同。

165-177 如果这不是第一个匹配(last非空)，则把该数据报放到上一个匹配的接收队列中。因为当sbappendaddr完成后要释放mbuf链，所以m_copy先要做个备份。sorwakeup唤醒所有等待这个数据的进程，last保存指向匹配的socket结构的指针。

使用变量last避免调用m_copy函数(因为要复制整个mbuf链，所以耗费很大)，除非有多个接收方接收该数据报。在通常只有一个接收方的情况下，for循环必须把last设成指向一个匹配PCB，当循环终止时，sbappendaddr把mbuf链放到插口的接收队列中——不做备份。

178-188 如果匹配的插口没有设置SO_REUSEPORT或SO_REUSEADDR插口选项，则没必要再找其他匹配，终止该循环。在循环的外部，调用 sbappendaddr把数据报放到这个插口的接收队列中。

189-197 如果在循环的最后，last为空，没找到匹配，则并不产生 ICMP差错，因为该数据报是发给广播或多播IP地址。

198-204 最后的匹配入口(可能是唯一的匹配入口)把原来的数据报(m)放到它的接收队列中。在调用sorwakeup后，udp_input返回，因为完成了对广播或多播数据报的处理。

函数的其他部分(图23-24)处理单播数据报。

23.7.4 连接上的UDP插口和多接口主机

在使用连接上的UDP插口与多接口主机上的一个进程交换数据报时，有一个微妙的问题。来自对等实体的数据报可能到达时具有不同的源站 IP地址，不能提交给连接上的插口。

考虑图23-27所示的例子。

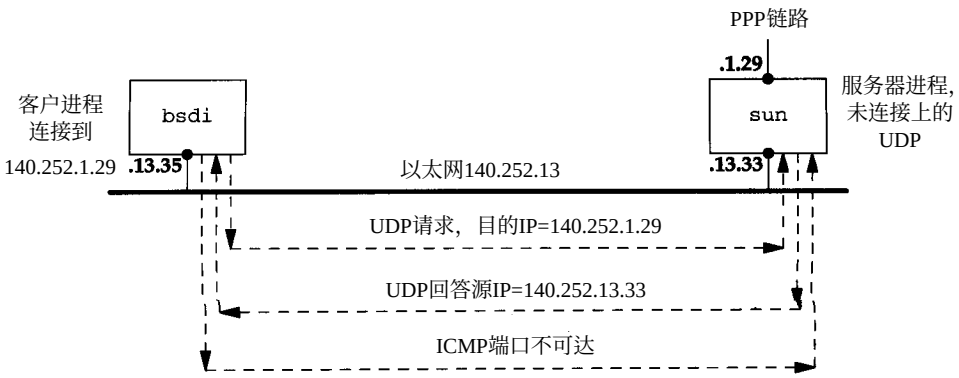


图23-27 连接上的UDP插口向一个多接口主机发送数据报的例子

有三个步骤：

1) bsdi上的客户程序创建一个UDP插口，并把它连接到140.252.1.29，这是sun上的PPP接口，而不是以太网接口。客户程序在插口上把数据报发给服务器。

Sun上的服务器接收并收下该数据报，即使到达接口与目的 IP地址不同(sun是一个路由器，所以不管它实现的是弱端系统模型或强端系统模型都没有关系)。数据报被提交给在未连接上的UDP插口上等待客户请求的服务器。

2) 服务器发一个回答，因为是在一个未连接上的 UDP插口上发送的，所以由内核根据外出的接口(140.252.13.33)选择回答的目的IP地址。请求的目的IP地址不作为回答的源站地址。

bsdi收到回答时，因为 IP地址不匹配，所以不把它提交给客户程序的连接上的 UDP接口。

3) 因为无法分用回答，所以 bsdi产生一个ICMP端口不可达差错(假定在bsdi上没有其他进程符合接收该数据报的条件)。

这个例子的问题在于，服务器并不把请求中的目的 IP地址作为回答的源站 IP地址。如果它这样做，就不存在这个问题了，但这个办法并不简单 —— 见习题23.10。我们将在图28-16中看到，如果一个 TCP服务器没有明确地把一个本地 IP地址绑定它的插口上，它就把来自客户的目的IP地址用作来自它自己的源IP地址。

23.8 udp_saveopt函数

如果进程指定了IP_RECVDSTADDR插口选项，则udp_input调用udp_saveopt，从收到的数据报中接收目的IP地址：

```
*mp = udp_saveopt((caddr_t) & ip_dst, sizeof(struct in_addr),
                  IP_RECVDSTADDR);
```

图23-28显示了这个函数。

udp_usrreq.c

```
278 /*
279  * Create a "control" mbuf containing the specified data
280  * with the specified type for presentation with a datagram.
281  */
282 struct mbuf *
283 udp_saveopt(p, size, type)
284 caddr_t p;
285 int     size;
286 int     type;
287 {
288     struct cmsghdr *cp;
289     struct mbuf *m;

290     if ((m = m_get(M_DONTWAIT, MT_CONTROL)) == NULL)
291         return ((struct mbuf *) NULL);
292     cp = (struct cmsghdr *) mtod(m, struct cmsghdr *);
293     bcopy(p, CMSG_DATA(cp), size);
294     size += sizeof(*cp);
295     m->m_len = size;
296     cp->cmsg_len = size;
297     cp->cmsg_level = IPPROTO_IP;
298     cp->cmsg_type = type;
299     return (m);
300 }
```

udp_usrreq.c

图23-28 udp_saveopt 函数：用控制信息创建mbuf

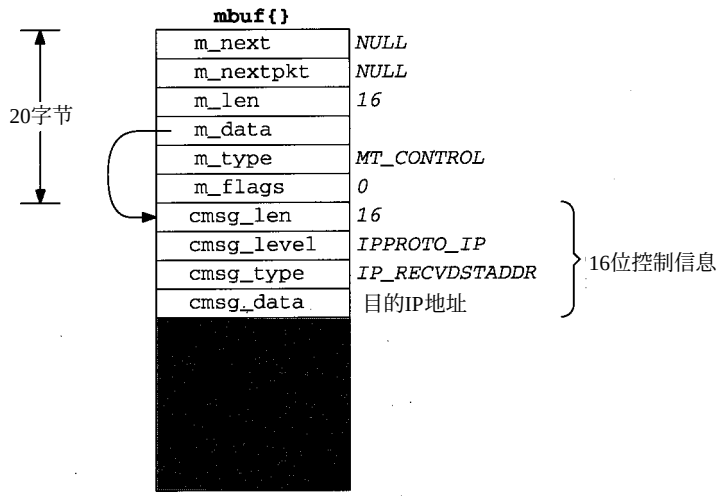


图23-29 把收到的数据报的目的地址作为控制信息保存的 mbuf

276-286 参数包括p，一个指向存储在mbuf中的信息的指针(收到的数据报的目的IP地址)；size，字节数大小(在这个例子中是4，IP地址的大小)；以及type，控制信息的类型(IP_RECVDSTADDR)。

290-299 分配一个mbuf，并且因为是在软件中断级执行代码，所以指定M_DONTWAIT。指针cp指向mbuf的数据部分，是一个指向cmsghdr结构(图16-14)的指针。bcopy把IP首部中

的IP地址复制到cmsghdr结构的数据部分。然后设置紧跟在cmsghdr结构后面的mbuf的长度(在本例中设成16)。图23-29是mbuf的最后一个状态。

cmsg_len字段包含了cmsghdr的长度(12)加上cmsg_data字段的长度(本例中是4)。如果应用程序调用recvmsg接收控制信息，则它必须检查cmsghdr结构，确定cmsg_data字段的类型和长度。

23.9 udp_ctlinput函数

当icmp_input收到一个ICMP差错(目的主机不可达、参数问题、重定向、源站抑制和超时)时，调用相应协议的pr_ctlinput函数：

```
if (ctlfunc = inetsw[ ip_protox[icp->icmp_ip.ip_p] ].pr_ctlinput)
    (*ctlfunc)(code, (struct sockaddr *)&icmpsrc, &icp->icmp_ip);
```

对于UDP，调用图22-32显示的函数udp_ctlinput。我们将在图23-30中给出这个函数。

314-322 参数包括cmd，图11-19的一个PRC_*xxx*常数；sa，一个指向sockaddr_in结构的指针，该结构含有ICMP报文的源站IP地址；ip，一个指向引起差错的IP首部的指针。对于目的站不可达、参数问题、源站抑制和超时差错，ip指向引起差错的IP首部。但当pfctlinput为重定向(图22-32)调用udp_ctlinput时，sa指向一个sockaddr_in结构，该结构中包含要被重定向的目的地址，ip是一个空指针。最后一种情况没有信息丢失，因为我们在22.11节看到，重定向应用于所有连接到目的地址的TCP和UDP插口。但对其他差错，如端口不可达，需要非空的第三个参数，因为协议跟在IP首部后面的协议首部包含了不可达端口。

323-325 如果差错不是重定向，并且PRC_*xxx*的值太大或全局数组inetctlerrmap中没有差错码，则忽略该ICMP差错。为理解这个检测，我们来看一下对收到的ICMP所做的处理：

1) icmp_input把ICMP类型和码转换成一个PRC_*xxx*差错码。

2) 把PRC_*xxx*差错码传给协议的控制输入函数。

3) Internet PCB协议(TCP和UDP)用inetctlerrmap把PRC_*xxx*差错码映射到一个Unix的errno值，这个值被返回给进程。

udp_usrreq.c

```
314 void
315 udp_ctlinput(cmd, sa, ip)
316 int     cmd;
317 struct sockaddr *sa;
318 struct ip *ip;
319 {
320     struct udphdr *uh;
321     extern struct in_addr zeroin_addr;
322     extern u_char inetctlerrmap[];

323     if (!PRC_IS_REDIRECT(cmd) &&
324         ((unsigned) cmd >= PRC_NCMDS || inetctlerrmap[cmd] == 0))
325         return;
326     if (ip) {
327         uh = (struct udphdr *) ((caddr_t) ip + (ip->ip_hl << 2));
328         in_pcbnotify(&udb, sa, uh->uh_dport, ip->ip_src, uh->uh_sport,
329                      cmd, udp_notify);
330     } else
331         in_pcbnotify(&udb, sa, 0, zeroin_addr, 0, cmd, udp_notify);
332 }
```

udp_usrreq.c

图23-30 udp_ctlinput 函数：处理收到的ICMP差错

图11-1和图11-2总结了ICMP报文的处理。

回到图 23-30，我们可以看到如何处理响应 UDP数据报的 ICMP源站抑制报文。icmp_input把ICMP报文转换成差错PRC_QUENCH，并调用udp_ctlinput。但因为在图11-2中，这个ICMP差错的errno行是空白，所以忽略该差错。

326-331 in_pcbnotify函数把该ICMP差错通知给恰当的PCB。如果udp_ctlinput的第三个参数非空，则把引起差错数据报的源和目的UDP端口以及源IP地址，传给in_pcbnotify。

udp_notify函数

in_pcbnotify函数的最后一个参数是一个指向函数的指针，in_pcbnotify为每个准备接收差错的PCB调用该函数。对UDP，该函数是udp_notify，如图23-31所示。

301-313 该函数的第二个参数errno保存在插口的so_error变量中。通过设置这个插口变量，使插口变成可读，并且如果进程调用 select，插口也可写。然后唤醒插口上所有正在等待接收或发送的进程接收该差错。

udp_usrreq.c

```
305 static void
306 udp_notify(inp, errno)
307 struct inpcb *inp;
308 int     errno;
309 {
310     inp->inp_socket->so_error = errno;
311     sorwakeup(inp->inp_socket);
312     sowwakeup(inp->inp_socket);
313 }
```

udp_usrreq.c

图23-31 udp_notify函数：通知进程一个异步差错

23.10 udp_usrreq函数

许多操作都要调用协议的用户请求函数。从图 23-14我们看到，在某个UDP插口上调用五个写函数中的任意一个，都以请求 PRU_SEND调用UDP 的用户请求函数结束。

图23-32显示了udp_usrreq的开始和结束。switch单独在后面的图中给出。图 15-17显示了该函数的参数。

udp_usrreq.c

```
417 int
418 udp_usrreq(so, req, m, addr, control)
419 struct socket *so;
420 int     req;
421 struct mbuf *m, *addr, *control;
422 {
423     struct inpcb *inp = sotoinpcb(so);
424     int     error = 0;
425     int     s;

426     if (req == PRU_CONTROL)
427         return (in_control(so, (int) m, (caddr_t) addr,
428                            (struct ifnet *) control));
429     if (inp == NULL && req != PRU_ATTACH) {
430         error = EINVAL;
431         goto release;
432     }
```

图23-32 udp_usrreq 函数体

```
433     /*
434      * Note: need to block udp_input while changing
435      * the udp pcb queue and/or pcb addresses.
436      */
437     switch (req) {

                        /* switch cases */

522     default:
523         panic("udp_usrreq");
524     }

525   release:
526     if (control) {
527         printf("udp control data unexpectedly retained\n");
528         m_freem(control);
529     }
530     if (m)
531         m_freem(m);
532     return (error);
533 }
```

udp_usrreq.c

图23-32 (续)

417-428 PRU_CONTROL请求来自ioctl系统调用。函数in_control完整地处理该请求。

429-432 在函数的开头定义inp时，把插口指针转换成PCB指针。唯一允许PCB指针为空的时候是创建新插口时(PRU_ATTACH)。

433-436 注释表明，只要在UDP PCB表中增加或删除表项，代码必须由splnet保护起来。这是因为udp_usrreq是作为系统调用的一部分来调用的，在它修改PCB的双重链表时，不能被UDP输入中断(被IP输入作为软件中断调用)。在修改PCB的本地或外部地址或端口时，也必须阻塞UDP输入，避免in_pcblookup不正确地提交收到的UDP数据报。

我们现在讨论每个case语句。图23-33语句中的PRU_ATTACH请求，来自socket系统调用。

udp_usrreq.c

```
438     case PRU_ATTACH:
439         if (inp != NULL) {
440             error = EINVAL;
441             break;
442         }
443         s = splnet();
444         error = in_pcballoc(so, &udb);
445         splx(s);
446         if (error)
447             break;
448         error = soreserve(so, udp_sendspace, udp_recvspace);
449         if (error)
450             break;
451         ((struct inpcb *) so->so_pcb)->inp_ip.ip_ttl = ip_defttl;
452         break;

453     case PRU_DETACH:
454         udp_detach(inp);
455         break;
```

udp_usrreq.c

图23-33 udp_usrreq 函数：PRU_ATTACH 和PRU_DETACH 请求

438-447 如果插口结构已经指向一个PCB，则返回EINVAL。in_pcballoc分配一个新的PCB，把它加到UDP PCB表的前面，把插口结构和PCB链接到一起。

448-450 soreserve为插口的发送和接收缓存保留缓存空间。如图 16-7所示，soreserve只是实施系统的限制，并没有真正分配缓存空间。发送和接收缓存的默认大小分别是9216字节(udp_sendspace)和41 600字节(udp_recvspace)。前者允许最大9200字节的数据报(在NFS分组中，有8 KB的数据)，加上16字节目的地址的sockaddr_in结构。后者允许插口上一次最多有40个1024字节的数据报排队。进程可调用setsockopt改变这些值。

451-452 进程通过setsockopt函数可以改变PCB中原型IP首部的两个字段：TTL和TOS。TTL默认值是64(ip_defttl)，TOS的默认值是0(普通服务)，因为in_pcballoc把PCB初始化为0。

453-455 close系统调用发布PRU_DETACH请求，调用图23-34所示的udp_detach函数。本节后面的PRU_ABORT请求也调用这个函数。

udp_usrreq.c

```
534 static void
535 udp_detach(inp)
536 struct inpcb *inp;
537 {
538     int     s = splnet();

539     if (inp == udp_last_inpcb)
540         udp_last_inpcb = &udb;
541     in_pcbdetach(inp);
542     splx(s);
543 }
```

udp_usrreq.c

图23-34 udp_detach 函数：删除一个UDP PCB

如果最后收到的PCB指针(“向后一个”缓存)指向一个已分离的PCB，则把缓存的指针设成指向UDP 表的表头(udb)。函数in_pcbdetach从UDP 表中移走PCB，并释放该PCB。

回到udp_usrreq，PRU_BIND请求是系统调用bind的结果，而PRU_LISTEN请求是系统调用listen的结果。如图23-35所示。

456-460 in_pcbbind完成所有PRU_BIND请求的工作。

461-463 对无连接协议来说，PRU_LISTEN请求是无效的——只有面向连接的协议才使用它。

udp_usrreq.c

```
456     case PRU_BIND:
457         s = splnet();
458         error = in_pcbbind(inp, addr);
459         splx(s);
460         break;

461     case PRU_LISTEN:
462         error = EOPNOTSUPP;
463         break;
```

udp_usrreq.c

图23-35 udp_usrreq 函数：PRU_BIND 和PRU_LISTEN 请求

前面提到，一个UDP应用程序，客户或服务器 (通常是客户)，可以调用connect。它修改插口发送或接收的外部IP地址和端口号。图 23-6显示了PRU_CONNECT、PRU_CONNECT2和PRU_ACCEPT请求。

464-474 如果插口已经连接上，则返回EISCONN。在这个时候，不应该连接上插口，因为在一个已经连接上的UDP插口上调用connect，会在生成PRU_CONNECT请求之前生成PRU_DISCONNECT请求。否则，由in_pcbconnect完成所有工作。如果没有遇到任何错误，soisconnected就把该插口结构标记成已经连接上的。

475-477 socketpair系统调用发布PRU_CONNECT2请求，只适用于Unix域的协议。

478-480 PRU_ACCEPT请求来自系统调用accept，只适用于面向连接的协议。

udp_usrreq.c

```
464     case PRU_CONNECT:
465         if (inp->inp_faddr.s_addr != INADDR_ANY) {
466             error = EISCONN;
467             break;
468         }
469         s = splnet();
470         error = in_pcbconnect(inp, addr);
471         splx(s);
472         if (error == 0)
473             soisconnected(so);
474         break;

475     case PRU_CONNECT2:
476         error = EOPNOTSUPP;
477         break;

478     case PRU_ACCEPT:
479         error = EOPNOTSUPP;
480         break;
```

udp_usrreq.c

图23-36 udp_usrreq 函数：PRU__CONNECT 、PRU__CONNECT2 和PRU__ACCEPT 请求

对于UDP插口，有两种情况会产生PRU_DISCONNECT请求：

1) 当关闭了一个连接上的UDP插口时，在PRU_DETACH之前调用PRU_DISCONNECT。

2) 当在一个已经连接上的UDP插口上发布connect时，soconnect在PRU_CONNECT请求之前发布PRU_DISCONNECT请求。

PRU_DISCONNECT请求如图23-37所示。

udp_usrreq.c

```
481     case PRU_DISCONNECT:
482         if (inp->inp_faddr.s_addr == INADDR_ANY) {
483             error = ENOTCONN;
484             break;
485         }
486         s = splnet();
487         in_pcbdisconnect(inp);
488         inp->inp_laddr.s_addr = INADDR_ANY;
489         splx(s);
490         so->so_state &= ~SS_ISCONNECTED;    /* XXX */
491         break;
```

udp_usrreq.c

图23-37 udp_usrreq 函数：PRU_DISCONNECT 请求

如果插口没有连接上，则返回ENOTCONN。否则，in_pcbdisconnect把外部IP地址设成0.0.0.0，把外部地址设成0。本地地址也被设成0.0.0.0，因为connect可能已经设置了这个PCB变量。

调用shutdown说明进程数据发送结束，产生PRU_SHUTDOWN请求，尽管对UDP插口来

说，很少有进程发布这个系统调用。图 23-38显示了PRU_SHUTDOWN、PRU_SEND和PRU_ABORT请求。

492-494 socantsendmore设置插口的标志，阻止其他更多输出。

495-496 图23-14显示了五个写函数如何调用udp_surreq，发布PRU_SEND请求。udp_output发送该数据报，udp_usrreq返回，避免执行release标号语句(图23-32)，因为还不能释放包含数据的mbuf链(m)。IP输出把这个mbuf链加到合适的接口输出队列中，当发送完数据后，由设备驱动器释放mbuf链。

udp_usrreq.c

```
492     case PRU_SHUTDOWN:
493         socantsendmore(so);
494         break;

495     case PRU_SEND:
496         return (udp_output(inp, m, addr, control));

497     case PRU_ABORT:
498         soisdisconnected(so);
499         udp_detach(inp);
500         break;
```

udp_usrreq.c

图23-38 udp_usrreq 函数体：PRU_SHUTDOWN 、PRU_SEND 和PRU_ABORT 请求

内核中UDP输出的唯一缓冲是在接口的输出队列中。如果插口的发送缓存内有存放数据报和目的地址的空间，则sosend调用udp_usrreq，该函数调用udp_output。图23-20显示，udp_output继续调用ip_output，ip_output为以太网调用ether_output，把数据报放到接口的输出队列中(如果有空间)。如果进程调用sendto的动作比接口快，就可以发送该数据报，ether_output返回ENOBUFS，并被返回给进程。

497-500 在UDP插口上从不发布PRU_ABORT请求。但如果发布，则断连插口，分离PCB。

PRU_SOCKADDR和PRU_PEERADDR请求分别来自系统调用 getsockname和getpeername。这两个请求和PRU_SENSE请求一起，如图23-39所示。

udp_usrreq.c

```
501     case PRU_SOCKADDR:
502         in_setsockaddr(inp, addr);
503         break;

504     case PRU_PEERADDR:
505         in_setpeeraddr(inp, addr);
506         break;

507     case PRU_SENSE:
508         /*
509          * fstat: don't bother with a blocksize.
510          */
511         return (0);
```

udp_usrreq.c

图23-39 udp_usrreq 函数体：PRU_SOCKADDR 、PRU_PEERADDR 和PRU_SENSE 请求

501-506 函数in_setsockaddr和in_setpeeraddr从PCB中取得信息，并把结果保存在addr参数中。

507-511 系统调用fstat产生PRU_SENSE请求。该函数返回OK，但并不返回其他信息。我们将在后面看到，TCP把发送缓存的大小作为stat结构的st_blksize元素返回。

图23-40显示了其他7个PRU_*xxx*请求，UDP插口不支持。

对最后两个请求的处理略微有些不同，因为PRU_RCVD不把指向mbuf的指针(m是一个非空指针)作为参数传递，而PRU_RCVOOB则传递指向协议mbuf的指针来填充。两种情况下，立即返回错误，不终止switch语句的执行，释放mbuf链。调用方用PRU_RCVOOB释放它分配的mbuf。

udp_usrreq.c

```
512     case PRU_SENDOOB:
513     case PRU_FASTTIMO:
514     case PRU_SLOWTIMO:
515     case PRU_PROTORCV:
516     case PRU_PROTOSEND:
517         error = EOPNOTSUPP;
518         break;

519     case PRU_RCVD:
520     case PRU_RCVOOB:
521         return (EOPNOTSUPP);    /* do not free mbuf's */
```

udp_usrreq.c

图23-40 udp_usrreq 函数体：不支持的7个请求

23.11 udp_sysctl函数

UDP的sysctl函数只支持一个选项， UDP检验和标志位。系统管理员可以禁止用sysctl(8)程序使能或禁止UDP检验和。图23-41显示了udp_sysctl函数。该函数调用sysctl_int取得或设置整数udpcksum的值。

udp_usrreq.c

```
547 udp_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
548 int    *name;
549 u_int   namelen;
550 void   *oldp;
551 size_t *oldlenp;
552 void   *newp;
553 size_t  newlen;
554 {
555     /* All sysctl names at this level are terminal. */
556     if (namelen != 1)
557         return (ENOTDIR);

558     switch (name[0]) {
559     case UDPCTL_CHECKSUM:
560         return (sysctl_int(oldp, oldlenp, newp, newlen, &udpcksum));
561     default:
562         return (ENOPROTOOPT);
563     }
564     /* NOTREACHED */
565 }
```

udp_usrreq.c

图23-41 udp_sysctl 函数

23.12 实现求精

23.12.1 UDP PCB高速缓存

在22.12节中，我们讲到PCB搜索的一般性质，以及代码是如何线性搜索协议的 PCB表的。现在我们把它和图23-24中UDP使用的“向后一个”高速缓存结合起来。

“向后一个”高速缓存的问题发生在当高速缓存的 PCB中有通配值时(本地地址，外部地址

或外部端口)：高速缓存的值永远不和收到的数据报匹配。[Partridge和Pink 1993] 测试的一个解决办法是，修改高速缓存，不比较通配值。也就是说，不再把 PCB中的外部地址和数据报的源地址进行比较，而是只有当PCB中的外部地址不是通配地址时，才比较这两个值。

这个办法有一个微妙的问题[Partridge和Pink 1993]。假定有两个插口绑定到本地端口555上。其中一个有三个通配成份，而另一个已经连接到外部地址 128.1.2.3，外部端口1600。如果我们高速缓存第一个PCB，且有一个数据报来自128.1.2.3，端口1600，则不能仅仅因为高速缓存的值具有通配外部地址就不比较外部地址。这叫做高速缓存隐藏 (cache hiding)。在这个例子中，高速缓存的PCB隐藏了另一个更好匹配的PCB。

为解决高速缓存隐藏，当在高速缓存加上或删除一个入口时，要做更多的工作。不能高速缓存那些可能隐藏其他 PCB的PCB。但这很简单，因为普通情形是每个本地端口都有一个插口。刚才我们给的例子中，两个插口都绑定到本地端口 555，尽管可能(尤其在一个多接口主机上)，但很少见。

[Partridge和Pink 1993]的另一个提高测试的也是记录最后发送的数据报的 PCB。这是[Mogul 1991]提出的，指出在所有收到的数据报中，一半都是对最后发送的数据报的回答。在这里高速缓存隐藏也是个问题，所以不高速缓存那些可能隐藏其他 PCB的PCB。

在通用系统上测试[Partridge和Pink 1993] 的两种高速缓存结果是，100 000个左右收到的UDP数据报显示出57%命中最近收到PCB高速缓存，30%命中最近发送PCB高速缓存。相比于没有高速缓存的版本，`udp_input`使用的CPU时间减少了一半还多。

这两种高速缓存还在某种程度上依赖于位置：刚刚到达的 UDP数据报极大可能来自与最近收到或发送UDP数据报相同的对等实体上。后者对发送一个数据报并等待回答的请求—应答应用程序很典型。[McKenney和Dove 1992] 显示某些应用程序，如联机交易处理(OLTP)系统的数据入口，没有产生 [Partridge和Pink 1993] 观察到的很高的命中率。正如我们在22.12节中提到的，对于具有上千个OLTP连接的系统来说，把PCB放在哈希链上，相对于最近收到和最近发送高速缓存而言，性能提高了一个数量级。

23.12.2 UDP检验和

提高实现性能的下一个领域是把进程和内核之间的数据复制与检验和计算结合起来。Net/3中，在输出操作中，每个数据都被处理两遍：一次是从进程复制到mbuf中(`uiomove`函数，被`sosend`调用)；另一次是计算UDP检验和(函数`in_cksum`被`udp_output`调用)。输入跟输出一样。

[Partridge和Pink1993] 修改了图23-14的UDP输出处理，调用一个 UDP专有函数`udp_sosend`，而不是`sosend`。这个新函数计算UDP 首部和内嵌的伪首部的检验和(不调用通用的`in_cksum`函数)，然后用特殊函数`in_uiomove`把数据从进程复制到一个mbuf链上(不是通用函数`uiomove`)，由这个新函数复制数据，更新检验和。采用这个技术，花在复制数据和计算检验和的时间减少了40%到45%。

在接收方情况就不同了。UDP 计算UDP首部和伪首部的检验和，移走UDP首部，把数据报在合适的插口上排队。当应用程序读取数据报时，`soreceive`的一个特殊版本(`udp_soreceive`)在把数据复制到用户高速缓存的同时，计算检验和。但是，如果检验和不正确，在整个数据报被复制到用户高速缓存之前，检测不到错误。对于普通的阻塞插口来说，`udp_soreceive`仅仅等待下一个数据报的到达。但是若插口是无阻塞的，且下一个数据报还没有准备好传给进程，就必须返回差错 `EWOULDBLOCK`。对于无阻塞读的UDP插口来说，

这意味着插口接口的两个变化：

1) select函数可以指示无阻塞UDP 插口可读，但如果检验和失败，其中一个读函数依然要返回错误EWOULDBLOCK。

2) 因为是在数据报被复制到用户高速缓存之后检测到检验和错误，所以即使读没有返回数据，应用程序的高速缓存也被改变了。

即使是阻塞插口，如果有检验和错误的数据报包含了100字节的数据，而下一个没有错误的数据报包含40字节的数据，则recvfrom的返回长度是40，但跟在用户高速缓存后面的60字节没有改变。

[Partridge和Pink1993] 在六台不同计算机上，对单纯复制和有检验和的复制的计时作了比较。结果显示，在许多体系结构的机器上，在复制操作中计算检验和无需额外时间。这种情况是在内存访问速度和CPU处理速度正确匹配的系统上的，目前许多RISC处理器都符合条件。

23.13 小结

UDP 是一个无连接的简单协议，这是我们为什么在 TCP之前讨论它的原因。UDP输出很简单：IP和UDP首部放在用户数据的前面，尽可能填满首部，把结果传递给 ip_output。唯一复杂的是UDP检验和计算，包括只为计算 UDP检验和而加上一个伪首部。我们将在第 26章遇到用于计算TCP检验和的伪首部。

当udp_input收到一个数据报时，它首先完成一个常规确认 (长度和检验和)；然后的处理根据目的IP地址是单播地址、广播或多播地址而不同。最多把单播数据报提交给一个进程，但多播或广播数据报可能会被提交给多个进程。“ 向后一个 ” 高速缓存适用于单播，其中维护着一个指向在其上接收数据报的最近 Internet PCB的指针。但是，我们也看到，由于 UDP应用程序普遍使用通配地址，所以这个高速缓存技术实际上毫无用处。

调用udp_ctlinput函数处理收到的ICMP报文，udp_usrreq函数处理来自插口层的PRU_*xxx*请求。

习题

23.1 列出udp_output传给ip_output的mbuf链的五种类型(提示：看看sosend)。

23.2 当进程为外出的数据报指定了IP选项时，上一题会是什么答案？

23.3 UDP 客户需要调用bind吗？为什么？

23.4 如果插口没有连接上，并且图23-15中调用M_PREPEND失败，那么在udp_output里，处理器优先级会发生什么变化？

23.5 udp_output不检测目的端口0。它可能发送一个具有目的端口0的UDP数据报吗？

23.6 假定当把一个数据报发送到一个广播地址时， IP_RECVDSTADDR插口选项有效，你如何确定这个地址是否是一个广播地址？

23.7 谁释放udp_saveopt(图23-38)分配的mbuf？

23.8 进程如何断连连接上的 UDP插口？也就是说，进程调用 connect并与对等实体交换数据报，然后进程要断连插口。允许它调用 sendto，并向其他主机发送数据报。

23.9 我们在图22-25的讨论中，注意到一个用外部IP地址255.255.255.255调用connect的UDP应用程序，在接口上发送时，是把该接口对应的广播地址作为目的IP地址。如果UDP应用使用未连接的插口，用目的地址255.255.255.255调用sendto，会发生什么情况？

第24章　TCP：传输控制协议

24.1　引言

传输控制协议，即TCP，是一种面向连接的传输协议，为两端的应用程序提供可靠的端到端的数据流传输服务。它完全不同于无连接的、提供不可靠的数据报传输服务的 UDP协议。

我们在第23章中详细讨论了UDP的实现，有9个函数、约800行C代码。我们将要讨论的TCP实现包括28个函数、约4500行C代码，因此，我们将TCP的实现分成7章来讨论。

这几章中不包括对TCP概念的介绍，假定读者已阅读过卷 1的第17章~第24章，熟悉TCP的操作。

24.2　代码介绍

TCP实现代码包括7个头文件，其中定义了大量的TCP结构和常量和6个C文件，包含TCP函数的具体实现代码。文件如图24-1所示。

文　件	描　述
netinet/tcp.h	tcphdr结构定义
netinet/tcp_debug.h	tcp_debug结构定义
netinet/tcp_fsm.h	TCP有限状态机定义
netinet/tcp_seg.h	实现TCP序号比较的宏定义
netinet/tcp_timer.h	TCP定时器定义
netinet/tcp_var.h	tcpcb(控制块)和tcpstat(统计)结构定义
netinet/tcpip.h	TCP+IP首部定义
netinet/tcp_debug.c	支持SO_DEBUG协议端口号调试(第27.10节)
netinet/tcp_input.c	tcp_input及其辅助函数(第28和第29章)
netinet/tcp_output.c	tcp_output及其辅助函数(第26章)
netinet/tcp_subr.c	各种TCP子函数(第27章)
netinet/tcp_timer.c	TCP定时器处理(第25章)
netinet/tcp_usrreq.c	PRU_*xxx*请求处理(第30章)

图24-1　TCP各章中将讨论的文件

图24-2描述了各TCP函数与其他内核函数之间的关系。带阴影的椭圆分别表示我们将要讨论的9个主要的TCP函数，其中8个出现在protosw结构中(图24-8)，第9个是tcp_output。

24.2.1　全局变量

图24-3列出了TCP函数中用到的全局变量。

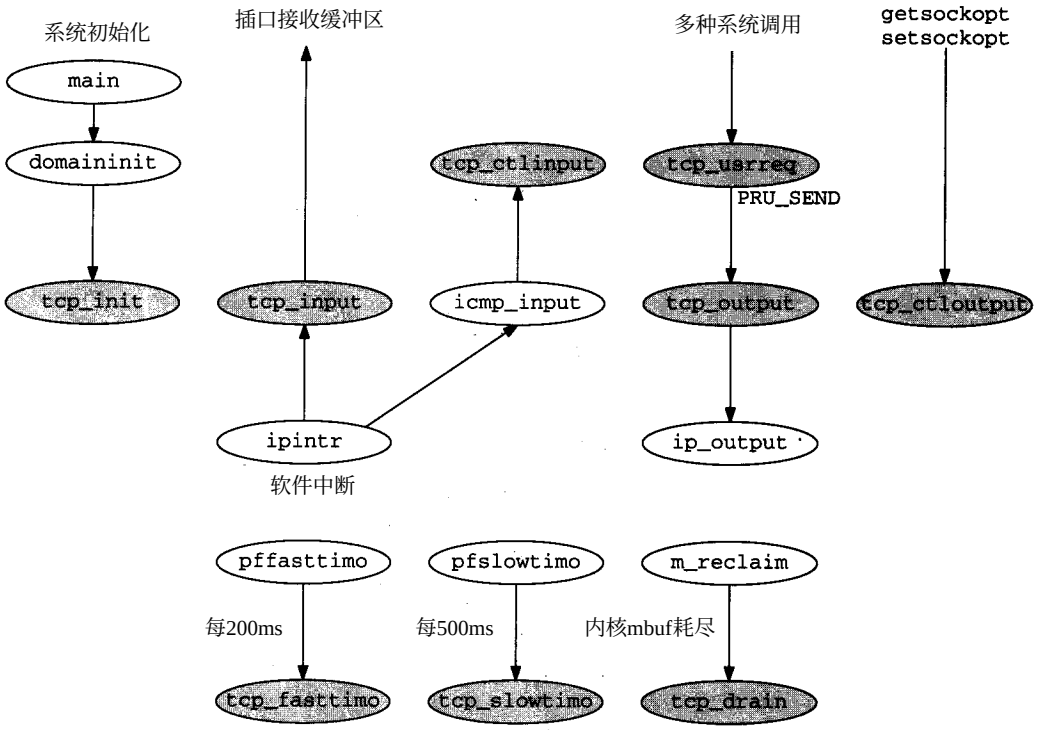


图24-2 TCP函数与其他内核函数间的关系

变 量	数据类型	描 述
tcb	struct inpcb	TCP Internet PCB表头
tcp_last_inpcb	struct inpcb *	指向最后收到报文段的PCB的指针：“后面一个”高速缓存
tcpstat	struct tcpstat	TCP统计数据(图24-4)
tcp_outflags	u_char	输出标志数组，索引为连接状态(图24-16)
tcp_recvspace	u_long	端口接收缓存大小默认值(8192字节)
tcp_sendspace	u_long	端口发送缓存大小默认值(8192字节)
tcp_iss	tcp_seq	TCP发送初始序号(ISS)
tcprexmtthresh	int	ACK重复次数的门限值(3)，触发快速重传
tcp_mssdflt	int	默认MSS值(512字节)
tcp_rttdflt	int	没有数据时RTT的默认值(3秒)
tcp_do_rfrc1323	int	如果为真(默认值)，请求窗口大小和时间戳选项
tcp_now	u_long	用于RFC 1323时间戳实现的500 ms计数器
tcp_keepidle	int	保活：第一次探测前的空闲时间(2小时)
tcp_keepintvl	int	保活：无响应时两次探测的间隔时间(75秒)
tcp_maxidle	int	保活：探测之后、放弃之前的时间(10分钟)

图24-3 后续章节中将介绍的全局变量

24.2.2 统计量

全局结构变量tcpstat中保存了各种TCP统计量，图24-4描述了各统计量的具体含义。在接下来的代码分析过程中，读者会了解到这些计数器数值的变化过程。

tcpstat成员	描　述	SNMP使用
tcps_accepts	被动打开的连接数	•
tcps_closed	关闭的连接数(包括意外丢失的连接)	
tcps_connattempt	试图建立连接的次数(调用connect)	•
tcps_conndrops	在连接建立阶段失败的连接次数(SYN收到之前)	•
tcps_connects	主动打开的连接次数(调用connect成功)	
tcps_delack	延迟发送的ACK数	
tcps_drops	意外丢失的连接数(收到SYN之后)	•
tcps_keepdrops	在保活阶段丢失的连接数(已建立或正等待SYN)	
tcps_keepprobe	保活探测指针发送次数	
tcps_keeptimeo	保活定时器或连接建立定时器超时次数	
tcps_pawsdrop	由于PSWS而丢失的报文段数	
tcps_pcbcachemiss	PCB高速缓存匹配失败次数	
tcps_persisttimeo	持续定时器超时次数	
tcps_predack	对ACK报文首部预测的正确次数	
tcps_preddat	对数据报文首部预测的正确次数	
tcps_rcvackbyte	由收到的ACK报文确认的发送字节数	
tcps_rcvackpack	收到的ACK报文数	
tcps_rcvacktoomuch	收到的对未发送数据进行确认的ACK报文数	
tcps_rcvafterclose	连接关闭后收到的报文数	
tcps_rcvbadoff	收到的首部长度无效的报文数	•
tcps_rcvbadsum	收到的检验和错误的报文数	•
tcps_rcvbyte	连续收到的字节数	
tcps_rcvbyteafterwin	在滑动窗口已满时收到的字节数	
tcps_rcvdupack	收到的重复ACK报文的次数	
tcps_rcvdupbyte	在完全重复报文中收到的字节数	
tcps_rcvduppack	内容完全一致的报文数	
tcps_rcvoobyte	收到失序的字节数	
tcps_rcvoopack	收到失序的报文数	
tcps_rcvpack	顺序接收的报文数	
tcps_rcvpackafterwin	携带数据超出滑动窗口通告值的报文数	
tcps_rcvpartdupbyte	部分内容重复的报文中的重复字节数	
tcps_rcvpartduppack	部分数据重复的报文数	
tcps_rcvshort	长度过短的报文数	•
tcps_rcvtotal	收到的报文总数	•
tcps_rcvwinprobe	收到的窗口探测报文数	
tcps_rcvwinupd	收到的窗口更新报文数	
tcps_rexmttimeo	重传超时次数	
tcps_rttupdated	RTT估算值更新次数	
tcps_segstimed	可用于RTT测算的报文数	
tcps_sndacks	发送的纯ACK报文数(数据长度=0)	
tcps_sndbyte	发送的字节数	
tcps_sndctrl	发送的控制(SYN、FIN、RST)报文数(数据长度=0)	
tcps_sndpack	发送的数据报文数(数据长度>0)	
tcps_sndprobe	发送的窗口探测次数(等待定时器强行加入1字节数据)	
tcps_sndrexmitbyte	重传的数据字节数	•
tcps_sndrexmitpack	重传的报文数	•
tcps_sndtotal	发送的报文总数	•
tcps_sndurg	只携带URG标志的报文数(数据长度=0)	
tcps_sndwinup	只携带窗口更新信息的报文数(数据长度=0)	
tcps_timeoutdrop	由于重传超时而丢失的连接数	

图24-4　tcpstat 结构变量中保存的TCP统计量

在命令行输入netstat -s，系统将输出当前TCP的统计值。图24-5的例子显示了主机连续运行30天后，各统计计数器的值。由于某些统计量互相关联——一个保存数据分组数目，另一个保存相应的字节数——图中做了一些简化。例如，表中第二行tcps_snd(pack,byte)实际表示了两个统计量，tcps_sndpack和tcps_sndbyte。

tcps_sndbyte值应为3 722 884 824字节，而不是-22 194 928字节，平均每个数据分组有450字节。类似的，tcps_rcvackbyte值应为3 738 811 552字节，而不是-21 264 360字节(平均每个数据分组565字节)。这些数据之所以被错误地显示，是因为netstat程序中调用printf语句时使用了%d(符号整型)，而非%lu(无符号长整型)。所有统计量均定义为无符号长整型，上面两个统计量的值已接近无符号32位长整型的上限($2^{32}-1$=4 294 967 295)。

netstat -s 输出	tcpstat 成员
10,655,999 packets sent 9,177,823 data packets (-22,194,928 bytes) 257,295 data packets (81,075,086 bytes) retransmitted 862,900 ack-only packets (531,285 delayed) 229 URG-only packets 3,453 window probe packets 74,925 window update packets 279,387 control packets	tcps_sndtotal tcps_snd{pack,byte} tcps_sndrexmit{pack,byte} tcps_sndacks,tcps_delack tcps_sndurg tcps_sndprobe tcps_sndwinup tcps_sndctrl
8,801,953 packets received 6,617,079 acks (for -21,264,360 bytes) 235,311 duplicate acks 0 acks for unsent data 4,670,615 packets (324,965,351 bytes) rcvd in-sequence 46,953 completely duplicate packets (1,549,785 bytes) 22 old duplicate packets 3,442 packets with some dup. data (54,483 bytes duped) 77,114 out-of-order packets (13,938,456 bytes) 1,892 packets (1,755 bytes) of data after window 1,755 window probes 175,476 window update packets 1,017 packets received after close 60,370 discarded for bad checksums 279 discarded for bad header offset fields 0 discarded because packet too short	tcps_rcvtotal tcps_rcvack{pack,byte} tcps_rcvdupack tcps_rcvacktoomuch tcps_rcv{pack,byte} tcps_rcvdup{pack,byte} tcps_pawsdrop tcps_rcvpartdup{pack,byte} tcps_rcvoo{pack,byte} tcps_rcv{pack,byte}afterwin tcps_rcvwinprobe tcps_rcvwindup tcps_rcvafterclose tcps_rcvbadsum tcps_rcvbadoff tcps_rcvshort
144,020 connection requests 92,595 connection accepts 126,820 connections established (including accepts) 237,743 connections closed (including 1,061 drops) 110,016 embryonic connections dropped	tcps_connattempt tcps_accepts tcps_connects tcps_closed,tcps_drops tcps_conndrops
6,363,546 segments updated rtt (of 6,444,667 attempts) 114,797 retransmit timeouts 86 connection dropped by rexmit timeout 1,173 persist timeouts 16,419 keepalive timeouts 6,899 keepalive probes sent 3,219 connections dropped by keepalive	tcps_{rttupdated,segstimed} tcps_rexmttimeo tcps_timeoutdrop tcps_persisttimeo tcps_keeptimeo tcps_keepprobe tcps_keepdrops
733,130 correct ACK header predictions 1,266,889 correct data packet header predictions 1,851,557 cache misses	tcps_predack tcps_preddat tcps_pcbcachemiss

图24-5 TCP统计量样本

24.2.3 SNMP变量

图24-6列出了SNMP TCP组中定义的14个SNMP简单变量，以及与它们相对应的tcpstat结构中的统计量。前四项的常量值在Net/3中定义，计数器tcpCurrEstab用于保存TCP PCB表中Internet PCB的数目。

图24-7列出了tcpTable，即TCP监听表(listener table)。

SNMP变量	tcpstat成员或常量	描　述
tcpRtoAlgorithm	4	用于计算重传定时时限的算法： 1=其他； 2=RTO为固定值； 3=MIL-STD-1778附录B； 4=Van Jacobson的算法；
tcpRtoMin	1000	最小重传定时时限，以毫秒为单位
tcpRtoMax	64000	最大重传定时时限，以毫秒为单位
tcpMaxConn	-1	可支持的最大TCP连接数(-1表示动态设置)
tcpActiveOpens	tcps_connattempt	从CLOSED转换到SYN SENT的次数
tcpPassiveOpens	tcps_accepts	从LISTEN转换到SYN RCVD的次数
tcpAttemptFails	tcps_conndrops	从SYN_SENT或SYN_RCVD转换到CLOSED的次数+从SYN_RCVD转换到LISTEN的次数
tcpEstabResets	tcps_drops	从ESTABLISHED或CLOSE_WAIT转换到CLOSED的次数
tcpCurrEstab	(见正文)	当前位于ESTABLISHED或CLOSE_WAIT状态的连接数
tcpInSegs	tcps_rcvtotal	收到的报文总数
tcpOutSegs	tcps_sndtotal - tcps_sndrexmitpack	发送的报文总数，减去重传报文数
tcpRetransSegs	tcps_sndrexmitpack	重传的报文总数
tcpInErrs	tcps_rcvbadsum + tcps_rcvbadoff + tcps_rcvshort	收到的出错报文总数
tcpOutRsts	(未实现)	RST标志置位的发送报文数

图24-6　tcp组中的简单SNMP变量

index = <tcpConnLocalAddress>.<tcpConnLocalPort>.<tcpConnRemAddress>.<tcpConnRemPort>		
SNMP变量	PCB变量	描　述
tcpConnState	t_state	连接状态：1 = CLOSED, 2=LISTEN, 3 = SYN_SENT, 4 = SYN_RCVD, 5 = ESTABLISHED,6 = FIN_WAIT, 7 = FIN_WAIT_2, 8 = CLOSE_WAIT, 9 = LAST_ACK, 10 = CLOSING, 11 = TIME_WAIT,12 = 删除TCP控制块
tcpConnLocalAddress	inp_laddr	本地IP地址
tcpConnLocalPort	inp_lport	本地端口号
tcpConnRemAddress	inp_faddr	远端IP地址
tcpConnRemPort	inp_fport	远端端口号

图24-7　TCP监听表：tcpTable 中的变量

第一个PCB变量(t_state)来自TCP控制块(图24-13)，其他四个变量来自 Internet PCB (图22-4)。

24.3 TCP 的protosw结构

图24-8列出了TCP protosw结构的成员变量，它定义了TCP协议与系统内其他协议间的交互接口。

成员变量	inetsw[2]	描　述
pr_type	*SOCK_STREAM*	TCP提供字节流传输服务
pr_domain	*&inetdomain*	TCP属于Internet协议族
pr_ptotocol	*IPPROTO_TCP(6)*	填充IP首部的ip_p字段
pr_flags	*PR_CONNREQUIRED|PR_WANTRCVD*	插口层标志，协议处理中忽略
pr_input	*tcp_input*	从IP层接收消息
pr_output	*0*	TCP协议忽略该成员变量
pr_ctlinput	*tcp_ctlinput*	处理ICMP错误的控制输入函数
pr_ctloutput	*tcp_ctloutput*	在进程中响应管理请求
pr_usrreg	*tcp_usrreq*	在进程中响应通信请求
pr_init	*tcp_init*	TCP初始化
pr_fasttimo	*tcp_fasttimo*	快超时函数，每200 ms调用一次
pr_slowtimo	*tcp_slowtimo*	慢超时函数，每500 ms调用一次
pr_drain	*tcp_drain*	内核mbuf耗尽时调用
pr_sysctl	*0*	TCP协议忽略该成员变量

图24-8 TCP protosw 结构

24.4 TCP的首部

tcphdr结构定义了TCP首部。图24-9给出了tcphdr结构的定义，图24-10描述了TCP首部。

tcp.h

```
40 struct tcphdr {
41     u_short th_sport;          /* source port */
42     u_short th_dport;          /* destination port */
43     tcp_seq th_seq;            /* sequence number */
44     tcp_seq th_ack;            /* acknowledgement number */
45 #if BYTE_ORDER == LITTLE_ENDIAN
46     u_char  th_x2:4,           /* (unused) */
47             th_off:4;          /* data offset */
48 #endif
49 #if BYTE_ORDER == BIG_ENDIAN
50     u_char  th_off:4,          /* data offset */
51             th_x2:4;           /* (unused) */
52 #endif
53     u_char  th_flags;          /* ACK, FIN, PUSH, RST, SYN, URG */
54     u_short th_win;            /* advertised window */
55     u_short th_sum;            /* checksum */
56     u_short th_urp;            /* urgent offset */
57 };
```

tcp.h

图24-9 tcphdr 结构

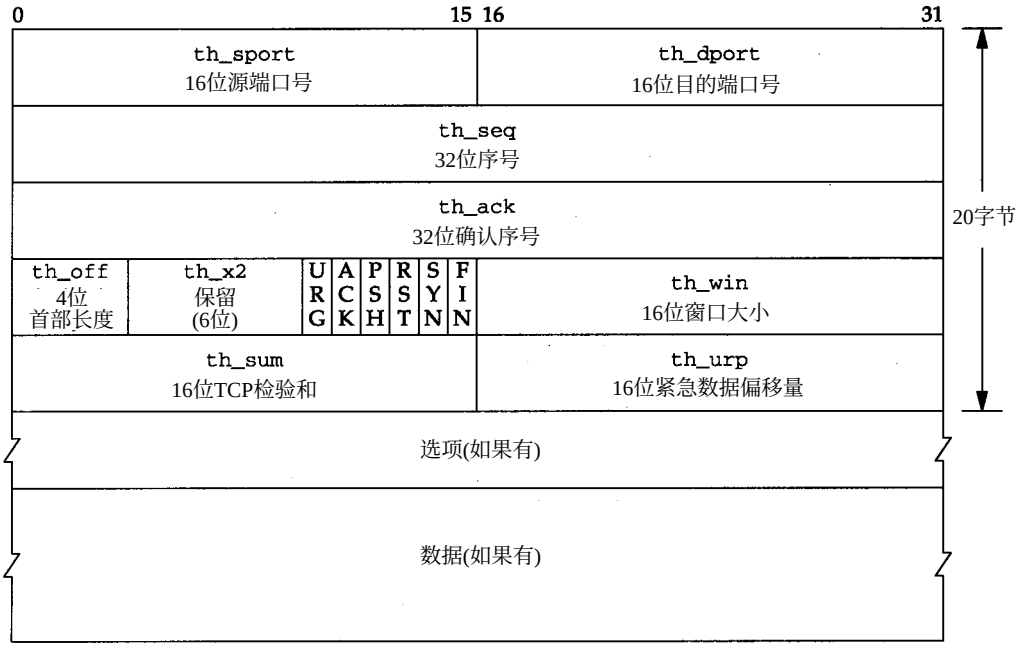


图24-10 TCP首部及可选数据

大多数RFC文档，相关书籍 (包括卷1)和接下来要讨论的TCP实现代码，都把th_urp称为“紧急指针 (urgent pointer)”。更准确的名称应该是“紧急数据偏移量 (urgent offset)”，因为这个字段给出的16 bit无符号整数值，与th_seq序号字段相加后，得到发送的紧急数据最后一个八位组的32 bit序号(关于该序号应该是紧急数据最后一个字节的序号，或者是紧急数据结束后的第一个字节的序号，一直存在着争议。但就我们目前的讨论而言，这一点无关紧要)。图24-13中，TCP代码把保存紧急数据最后一个八位组的32 bit序号的snd_up正确地称为“紧急数据发送指针”。如果将TCP首部的16 bit偏移量也称为“指针”，容易引起误解。在练习26.6中，我们重申了“紧急指针”和“紧急数据偏移量”间的区别。

TCP首部中4 bit的首部长度、接着的6 bit的保留字段和6 bit的码元标志，在C结构中定义为两个4 bit的比特字段，和紧跟的一个8 bit字节。为了处理两个比特字段在8 bit字节中的存放次序，C代码根据不同的主机字节存储顺序使用了#ifdef语句。

还请注意，TCP中称4 bit的th_off为“首部长度”，而C代码中称之为“数据偏移量”。两种名称都正确，因为它表示TCP首部的长度，包括可选项，以32 bit为单位，也就是指向用户数据第一个字节的偏移量。

th_flags成员变量包括6个码元标志比特，通过图24-11中定义的名称读写。

Net/3中，TCP首部通常意味着“IP首部 + TCP首部”。tcp_input处理收到的IP数据报和tcp_output构造待发送的IP数据报时都采用了这一思想。图24-12中给出了tcpiphdr结构的定义，形式化地描述了组合的IP/TCP首部。

38-58 图23-19给出的ipovly结构定义了20字节长度的IP首部。通过前面章节的讨论可知，尽管长度相同(20字节)，但这个结构并不是一个真正的IP首部。

th_flags	描 述
TH_ACK	确认序号(th_ack)有效
TH_FIN	发送方字节流结束
TH_PUSH	接收方应该立即将数据提交给应用程序
TH_RST	连接复位
TH_SYN	序号同步(建立连接)
TH_URG	紧急数据偏移量(th_urp)有效

图24-11 th_flags 值

tcpip.h

```
38 struct tcpiphdr {
39     struct ipovly ti_i;          /* overlaid ip structure */
40     struct tcphdr ti_t;          /* tcp header */
41 };

42 #define ti_next     ti_i.ih_next
43 #define ti_prev     ti_i.ih_prev
44 #define ti_x1       ti_i.ih_x1
45 #define ti_pr       ti_i.ih_pr
46 #define ti_len      ti_i.ih_len
47 #define ti_src      ti_i.ih_src
48 #define ti_dst      ti_i.ih_dst
49 #define ti_sport    ti_t.th_sport
50 #define ti_dport    ti_t.th_dport
51 #define ti_seq      ti_t.th_seq
52 #define ti_ack      ti_t.th_ack
53 #define ti_x2       ti_t.th_x2
54 #define ti_off      ti_t.th_off
55 #define ti_flags    ti_t.th_flags
56 #define ti_win      ti_t.th_win
57 #define ti_sum      ti_t.th_sum
58 #define ti_urp      ti_t.th_urp
```

tcpip.h

图24-12 tcpiphdr 结构定义：组合的IP/TCP首部

24.5 TCP的控制块

在图22-1中我们看到，除了标准的Internet PCB外，TCP还有自己专用的控制块，tcpcb结构，而UDP则不需要专用控制块，它的全部控制信息都已包含在Internet PCB中。

TCP控制块较大，需占用140字节。从图22-1中可看到，Internet PCB与TCP控制块彼此对应，都带有指向对方的指针。图24-13给出了TCP控制块的定义。

tcp_var.h

```
41 struct tcpcb {
42     struct tcpiphdr *seg_next;   /* reassembly queue of received segments */
43     struct tcpiphdr *seg_prev;   /* reassembly queue of received segments */
44     short   t_state;             /* connection state (Figure 24.16) */
45     short   t_timer[TCPT_NTIMERS];  /* tcp timers (Chapter 25) */
46     short   t_rxtshift;          /* log(2) of rexmt exp. backoff */
47     short   t_rxtcur;            /* current retransmission timeout (#ticks) */
48     short   t_dupacks;           /* #consecutive duplicate ACKs received */
49     u_short t_maxseg;            /* maximum segment size to send */
50     char    t_force;             /* 1 if forcing out a byte (persist/OOB) */
51     u_short t_flags;             /* (Figure 24.14) */
```

图24-13 tcpcb 结构：TCP控制块

```
52      struct tcpiphdr *t_template;     /* skeletal packet for transmit */
53      struct inpcb *t_inpcb;       /* back pointer to internet PCB */
54  /*
55   * The following fields are used as in the protocol specification.
56   * See RFC783, Dec. 1981, page 21.
57   */
58  /* send sequence variables */
59      tcp_seq snd_una;             /* send unacknowledged */
60      tcp_seq snd_nxt;             /* send next */
61      tcp_seq snd_up;              /* send urgent pointer */
62      tcp_seq snd_wl1;             /* window update seg seq number */
63      tcp_seq snd_wl2;             /* window update seg ack number */
64      tcp_seq iss;                 /* initial send sequence number */
65      u_long  snd_wnd;             /* send window */
66  /* receive sequence variables */
67      u_long  rcv_wnd;             /* receive window */
68      tcp_seq rcv_nxt;             /* receive next */
69      tcp_seq rcv_up;              /* receive urgent pointer */
70      tcp_seq irs;                 /* initial receive sequence number */
71  /*
72   * Additional variables for this implementation.
73   */
74  /* receive variables */
75      tcp_seq rcv_adv;             /* advertised window by other end */
76  /* retransmit variables */
77      tcp_seq snd_max;             /* highest sequence number sent;
78                                    * used to recognize retransmits */
79  /* congestion control (slow start, source quench, retransmit after loss) */
80      u_long  snd_cwnd;            /* congestion-controlled window */
81      u_long  snd_ssthresh;        /* snd_cwnd size threshhold for slow start
82                                    * exponential to linear switch */
83  /*
84   * transmit timing stuff.  See below for scale of srtt and rttvar.
85   * "Variance" is actually smoothed difference.
86   */
87      short   t_idle;              /* inactivity time */
88      short   t_rtt;               /* round-trip time */
89      tcp_seq t_rtseq;             /* sequence number being timed */
90      short   t_srtt;              /* smoothed round-trip time */
91      short   t_rttvar;            /* variance in round-trip time */
92      u_short t_rttmin;            /* minimum rtt allowed */
93      u_long  max_sndwnd;          /* largest window peer has offered */
94  /* out-of-band data */
95      char    t_oobflags;          /* TCPOOB_HAVEDATA, TCPOOB_HADDATA */
96      char    t_iobc;              /* input character, if not SO_OOBINLINE */
97      short   t_softerror;         /* possible error not yet reported */
98  /* RFC 1323 variables */
99      u_char  snd_scale;           /* scaling for send window (0-14) */
100     u_char  rcv_scale;           /* scaling for receive window (0-14) */
101     u_char  request_r_scale;     /* our pending window scale */
102     u_char  requested_s_scale;   /* peer's pending window scale */
103     u_long  ts_recent;           /* timestamp echo data */
104     u_long  ts_recent_age;       /* when last updated */
105     tcp_seq last_ack_sent;       /* sequence number of last ack field */
106 };
107 #define intotcpcb(ip)    ((struct tcpcb *)(ip)->inp_ppcb)
108 #define sototcpcb(so)    (intotcpcb(sotoinpcb(so)))
```

tcp_var.h

图24-13 (续)

现在暂不讨论上述成员变量的具体含义，在后续代码中遇到时再详细分析。

图24-14列出了t_flags变量的可选值。

t_flags	描　述
TF_ACKNOW	立即发送ACK
TF_DELACK	延迟发送ACK
TF_NODELAY	立即发送用户数据，不等待形成最大报文段 (禁止Nagle算法)
TF_NOOPT	不使用TCP选项(永不填充TCP选项字段)
TF_SENTFIN	FIN已发送
TF_RCVD_SCALE	对端在SYN报文中发送窗口变化选项时置位
TF_RCVD_TSTMP	对端在SYN报文中发送时间戳选项时置位
TF_REQ_SCALE	已经/将要在SYN报文中请求窗口变化选项
TF_REQ_TSTMP	已以/将要在SYN中请求时间戳选项

图24-14 t_flags 取值

24.6 TCP的状态变迁图

TCP协议根据连接上到达报文的不同类型，采取相应动作，协议规程可抽象为图 24-15所示的有限状态变迁图。读者在本书的扉页前也可找到这张图，以便在阅读有关TCP的章节时参考。

图中的各种状态变迁组成了 TCP有限状态机。尽管TCP协议允许从LISTEN状态直接变迁到SYN_SENT状态，但使用SOCKET API编程时这种变迁不可实现 (调用listen后不可以调用connect)。

TCP控制块的成员变量t_state保存一个连接的当前状态，可选值如图 24-16所示。

图中还定义了tcp_outflags数组，保存了处于对应连接状态时 tcp_output将使用的输出标志。

图24-16还列出了与符号常量相对应的数值，因为在代码中将利用它们之间的数值关系。例如，有下面两个宏定义：

```
#define TCPS_HAVERCVDSYN(s) ((s)>=TCPS_SYN_RECEIVED)
#define TCPS_HAVERCVDFIN(s) ((s)>=TCPS_TIME_WAIT)
```

类似地，连接未建立时，即t_state小于TCPS_ESTABLISHED时，tcp_notify处理ICMP差错的方式也不同。

> TCPS_HAVERCVDSYN的命名是正确的，但TCPS_HAVERCVDFIN则可能引起误解，因为在 CLOSE_WAIT、CLOSING和LAST_ACK状态也会收到FIN。我们将在第29章中遇到该宏。

半关闭

当进程调用shutdown且第二个参数设为1时，称为“半关闭”。TCP发送FIN，但允许进程在同一端口上继续接收数据(卷1的18.5节中举例介绍了TCP的半关闭)。

例如，尽管图24-15中只在ESTABLISHED状态标注了“数据传输”，但如果进程执行了“半关闭”，则连接变迁到FIN_WAIT_1状态和其后的FIN_WAIT_2状态，在这两个特定状态中，进程仍然可以接收数据。

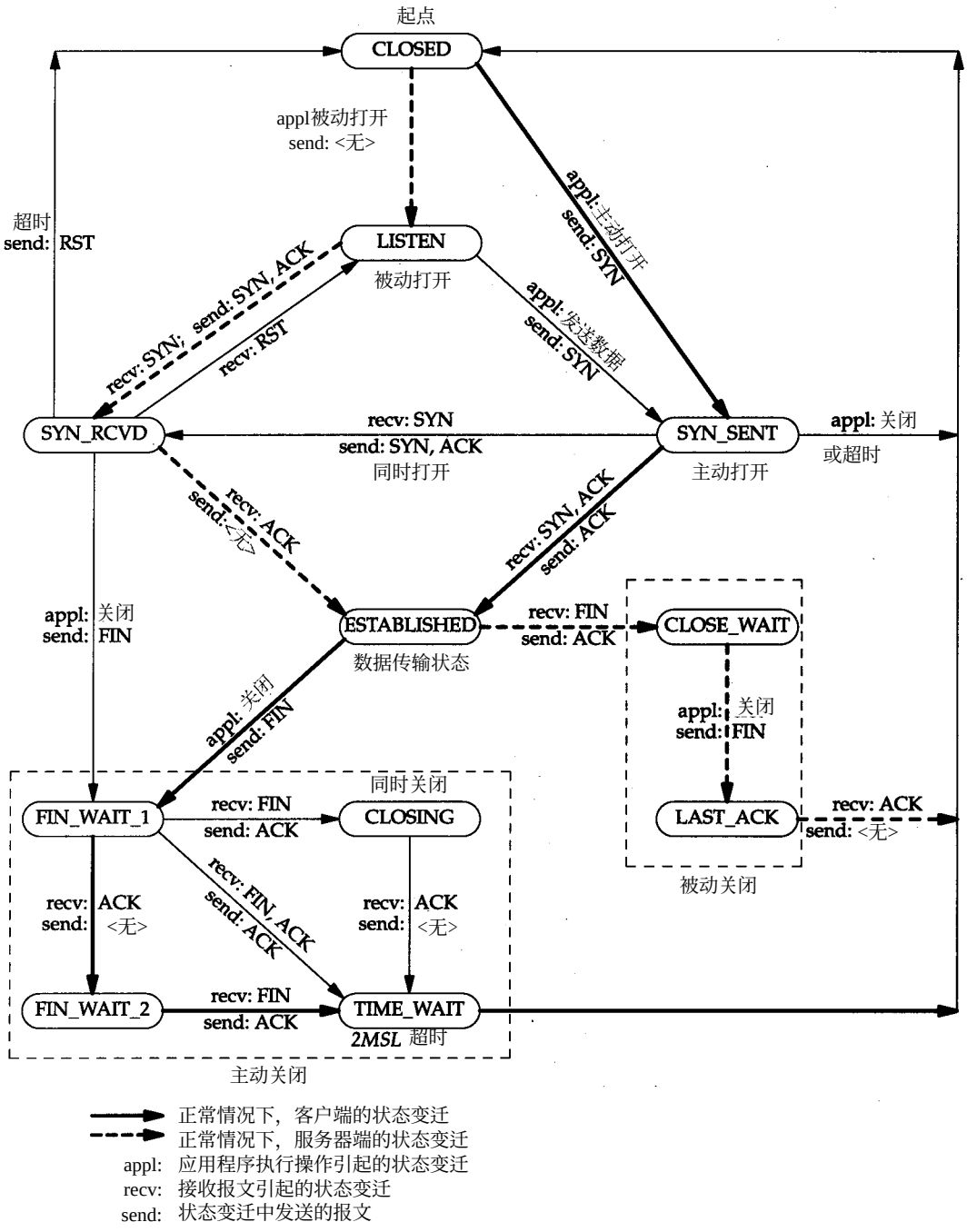


图24-15 TCP状态变迁图

24.7 TCP的序号

TCP连接上传输的每个数据字节，以及SYN、FIN等控制报文都被赋予一个32 bit的序号。TCP首部的序号字段(图24-10)填充了报文段第一个数据字节的32 bit的序号，确认号字段填充

了发送方希望接收的下一序号，确认已正确接收了所有序号小于等于确认号减 1的数据字节。换言之，确认号是ACK发送方等待接收的下一序号。只有当报文首部的 ACK标志置位时，确认序号才有效。读者将看到，除了在主动打开首次发送 SYN时(SYN_SENT状态，参见图24-16中的tcp_outflags[2])或在某些RST报文段中，ACK标志总是被置位的。

t_state	值	描 述	tcp_outflags[]
TCPS_CLOSED	0	关闭	*TH_RST* | *TH_ACK*
TCPS_LISTEN	1	监听连接请求(被动打开)	*0*
TCPS_SYN_SENT	2	已发送SYN(主动打开)	*TH_SYN*
TCPS_SYN_RECEIVED	3	已发送并接收SYN；等待ACK	*TH_SYN* | *TH_ACK*
TCPS_ESTABLISHED	4	连接建立(数据传输)	*TH_ACK*
TCPS_CLOSE_WAIT	5	已收到FIN，等待应用程序关闭	*TH_ACK*
TCPS_FIN_WAIT_1	6	已关闭，发送FIN；等待ACK和FIN	*TH_FIN* | *TH_ACK*
TCPS_CLOSING	7	同时关闭；等待ACK	*TH_FIN* | *TH_ACK*
TCPS_LAST_ACK	8	收到的FIN已关闭；等待ACK	*TH_FIN* | *TH_ACK*
TCPS_FIN_WAIT_2	9	已关闭，等待FIN	*TH_ACK*
TCPS_TIME_WAIT	10	主动关闭后2MSL等待状态	*TH_ACK*

图24-16 t_state 取值

由于TCP连接是全双工的，每一端都必须为两个方向上的数据流维护序号。 TCP控制块中(图24-13)有13个序号：8个用于数据发送(发送序号空间)，5个用于数据接收(接收序号空间)。

图24-17给出了发送序号空间中4个变量间的关系：snd_wnd、snd_una、snd_nxt和snd_max。这个例子列出了数据流的第1~第11字节。

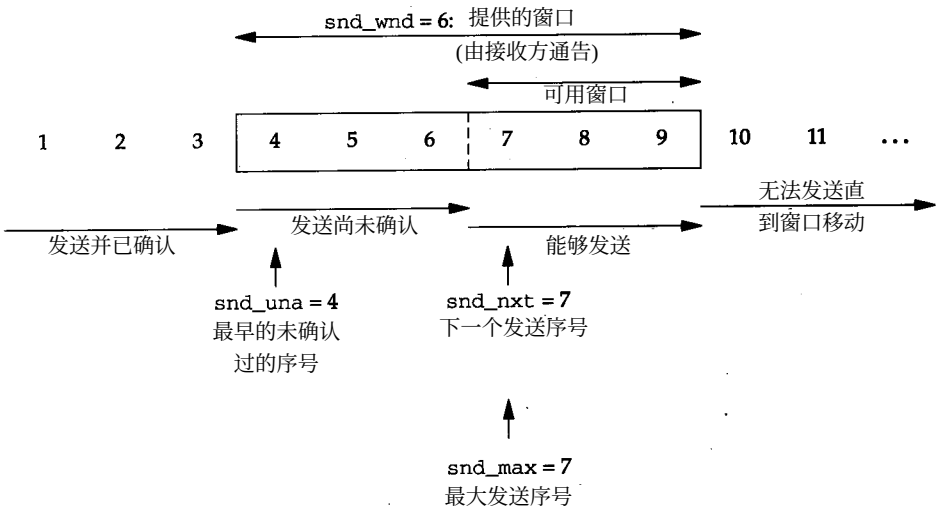


图24-17 发送序号空间举例

一个有效的ACK序号必须满足：

```
snd_una < 确认序号 <= snd_max
```

图24-17的例子中，一个有效 ACK的确认号必须是 5、6或7。如果确认号小于或等于snd_una，则是一个重复的ACK。它确认了已确认过的八位组，否则 snd_una不会递增超过那些序号。

tcp_output中有多处用到下面的测试，如果正发送的是重传数据，则表达式为真：

```
snd_nxt < snd_max
```

图24-18给出了图24-17中连接的另一端：接收序号空间，图中假定还未收到序号为 4、5、6的报文，标出了三个变量rcv_nxt、rcv_wnd和rcv_adv。

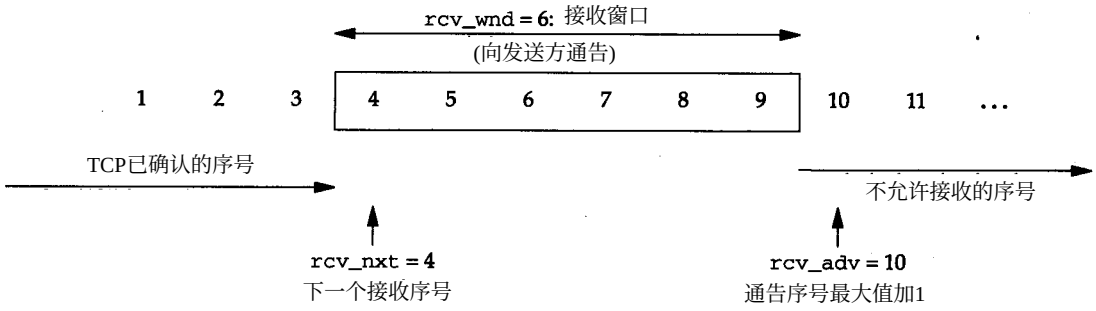


图24-18 接收序号空间举例

如果接收报文段中携带的数据落在接收窗口内，则该报文段是一个有效报文段。换言之，下面两个不等式中至少要有一个为真。

```
rcv_nxt <= 报文段起始序号  < rcv_nxt + rcv_wnd
rcv_nxt <=报文段终止序号 <  rcv_nxt + rcv_wnd
```

报文段起始序号就是TCP首部的序号字段，ti_seq。终止序号是序号字段加上 TCP数据长度后减1。

例如，图24-19中的TCP报文段，携带了图24-17中发送的三个字节，序号分别是4、5和6。

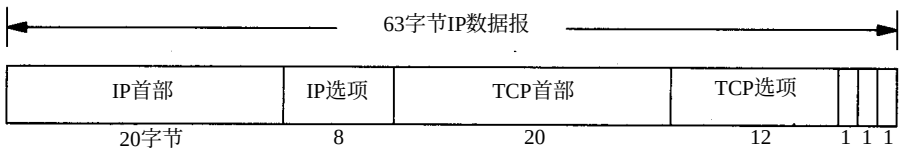


图24-19 TCP报文段在IP数据报中传输

假定IP数据报中有8字节的IP任选项和12字节的TCP任选项。图12-20列出了各有关变量的取值。

变　量	值	描　述
ip_hl	7	IP首部+IP任选项长度，以32 bit为单位(=28字节)
ip_len	63	IP数据报长度，以字节为单位 (20+8 +20+12+3)
ti_off	8	TCP首部+TCP任选项长度，以32 bit为单位(=32字节)
ti_seq	4	用户数据第一个字节的序号
ti_len	3	TCP数据的字节数：ip_len-(ip_hl×4)-(ti_off×4)
	6	用户数据最后一个字节的序号：ti_seq+ti_len-1

图24-20 图24-19中各变量的取值

ti_len并非TCP首部的字段，而是在对接收到的首部计算检验和及完成验证之后，根据图24-20中的算式得到的结果，存储到外加的 IP结构中(图24-12)。图中最后一个值并不存储到变量中，而是在需要时直接从其他值中通过计算得到。

1. 序号取模运算

TCP必须处理的一个问题是序号来自有限的32位取值空间：0~4 294 967 295。如果某个TCP连接传输的数据量超过2^{32}字节，序号从4 294 967 295回绕到0，将出现重复序号。

即使传输数据量小于2^{32}字节，仍可能遇到同样的问题，因为连接的初始序号并不一定从0开始。各数据流方向上的初始序号可以是0~4 294 967 295之间的任何值。这个问题使序号复杂化。例如，序号1可能大于序号4 294 967 295。

在tcp.h中，TCP序号定义为unsigned long

```
typedef u_long tcp_seq;
```

图24-21定义了4个用于序号比较的宏。

tcp_seq.h

```
40 #define SEQ_LT(a,b)     ((int)((a)-(b)) < 0)
41 #define SEQ_LEQ(a,b)    ((int)((a)-(b)) <= 0)
42 #define SEQ_GT(a,b)     ((int)((a)-(b)) > 0)
43 #define SEQ_GEQ(a,b)    ((int)((a)-(b)) >= 0)
```

tcp_seq.h

图24-21 TCP序号比较宏

2. 举例——序号比较

下面这个例子说明了TCP序号的操作方式。假定序号只有3 bit，0~7。图24-22列出了全部8个序号和相应的二进制补码(为求二进制补码，将二进制码中的所有0变为1，所有1变为0，最后再加1)。给出补码形式，是因为a-b =a+(b的补码)。

x	二进制码	二进制补码	0-x	1-x	2-x
0	000	000	000	001	010
1	001	111	111	000	001
2	010	110	110	111	000
3	011	101	101	110	111
4	100	100	100	101	110
5	101	011	011	100	101
6	110	010	010	011	100
7	111	001	001	010	011

图24-22 3 bit序号举例

表中最后三栏分别是0-x、1-x和2-x。在这三栏中，如果定义计算结果是带符号整数(注意图24-21中的四个宏，计算结果全部强制转换为int)，那么最高位为1表示值小于0(SEQ_LT宏)，最高位为0且值不为0表示大于0(SEQ_GT宏)。最后三栏中以横线分隔开四个负值和四个非负值。

请注意图24-22中的第四栏(标注“0-x”)，可看出0小于1、2、3和4(最高位比特为1)，而0大于5、6和7(最高位比特为0且结果非0)。图24-23显示了这种关系。

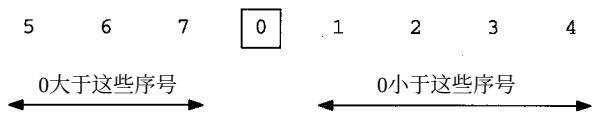


图24-23 3 bit的TCP序号的比较

图24-22中的第五栏(1 - x)也存在类似的关系，如图24-24所示。

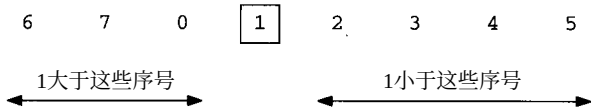


图24-24 3 bit的TCP序号的比较

图24-25是上面两图的另一种表示形式，使用圆环强调了序号的回绕现象。

图24-25 图24-23和图24-24的另一种表示形式

就TCP而言，通过序号比较来确定给定序号是新序号或重传序号。例如，在图 24-24的例子中，如果TCP正等待的序号为1，但到达序号为6，通过前面介绍的计算可知 6小于1，从而判定这是重传的数据，可予以丢弃。但如果到达序号为 5，因为5大于1，TCP判定这是新数据，予以保存，并继续等待序号为2、3和4的八位组(假定序号为5的数据字节落在接收窗口内)。

图24-26扩展了图24-25中左边的圆环，用TCP 32 bit的序号替代了3 bit的序号。

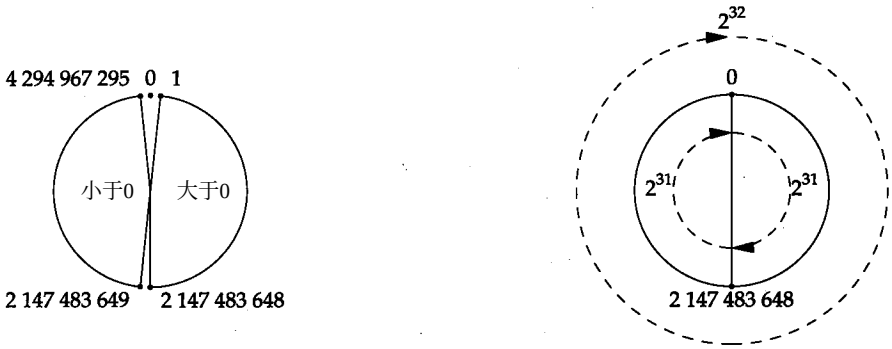


图24-26 与序号0比较：采用32 bit序号

图24-26右边的圆环强调了32 bit序号空间的一半有2^{31}个可用数字。

24.8 tcp_init函数

系统初始化时，domaininit函数调用TCP的初始化函数：tcp_init (图24-27)。

1. 设定初始发送序号

初始发送序号(ISS)，tcp_iss，被初始化为1。请注意，代码注释指出，这是错误的。后面讨论TCP的“平静时间(quite time)”时，将简单介绍这一选择的原因。请读者自行与图 7-23中IP标识符的初始化做比较，后者使用了当天的时钟。

tcp_subr.c

```
43 void
44 tcp_init()
45 {
46     tcp_iss = 1;                    /* wrong */
47     tcb.inp_next = tcb.inp_prev = &tcb;

48     if (max_protohdr < sizeof(struct tcpiphdr))
49                 max_protohdr = sizeof(struct tcpiphdr);
50     if (max_linkhdr + sizeof(struct tcpiphdr) > MHLEN)
51                 panic("tcp_init");
52 }
```

tcp_subr.c

图24-27 tcp_init 函数

2. TCP Internet PCB链表初始化

PCB首部(tcb)的previous指针和next指针都指向自己，这是一个空的双向链表。tcb PCB的其余成员均初始化为0(所有未明确初始化的全局变量均设为0)。事实上，除链表外，在该PCB首部中只用了一个字段inp_lport：下一个分配的TCP临时端口号。TCP使用的第一个临时端口号应为1024，练习22.4的解答中给出了原因。

3. 计算最大协议首部长度

到目前为止，讨论过的协议首部的长度最大不超过40字节，max_protohdr设为40(组合的IP/TCP首部长度，不带任何可选项)。图7-17定义了该变量。如果max_linkhdr (通常为16)加40后大于放入单个mbuf中带首部的数据报的数据长度(100字节，图2-7中的MHLEN)，内核将告警。

MSL和平静时间的概念

TCP协议要求如果主机崩溃，且没能保存打开TCP连接上最后使用的序号，则重启后在一个MSL(2分钟，平静时间)内，不能发送任何TCP报文段。目前，基本没有TCP实现能够在系统崩溃或操作员关机时保存这些信息。

MSL是最大报文段生存时间(maximum segment lifetime)，指任何报文段被丢弃前在网络中能够存在的最大时间。不同的实现可选择不同的MSL。连接主动关闭后，将在CLOSE_WAIT状态等待2个MSL时间(图24-15)。

RFC 793(Postel 1981c)建议MSL设定为2分钟，但Net/3实现中MSL设为30秒(图25-3中定义的常量TCPTV_MSL)。

如果报文段在网络中出现延迟，协议会出现问题(RFC 793称之为漫游重复(wandering duplicate)。假定Net/3系统启动时tcp_iss置为1(图24-27)，经过一段时间，在序号刚刚回绕时系统崩溃。后面25.5节中将介绍，tcp_iss每秒增加128 000，即重启后需经过9.3小时序号才会回绕。此外，每发送一个connect，tcp_iss将增加64 000，因此序号回绕时间必然早于9.3小时。下面的例子说明了老的报文段怎样被错误地发送到现在的连接上。

1) 一个客户和服务器建立了一个连接。客户的端口号是1024，发送了一个序号为2的报文段。该报文段在传送途中陷入路径循环，未能到达服务器。这个报文段成为“漫游重复”报文段。

2) 客户重发该报文段，序号依旧为2。重发报文段到达服务器。

3) 客户关闭连接。

4) 客户主机崩溃。

5) 客户主机在崩溃后40秒重启，TCP初始化`tcp_iss`为1。

6) 同一客户和同一服务器之间立即建立了一条新的连接，使用了同样的端口号：客户端口号为1024，服务器方依然是其预知的端口号。客户发送的SYN中初始序号置为1。这条新的使用同样端口对的连接称为原有连接的化身(incarnation)。

7) 步聚1中的漫游重复报文段最终到达服务器，并被认为是新建连接中的合法报文段，尽管它实际上属于原有连接。

图24-28列出了上述步骤发生的时间顺序。

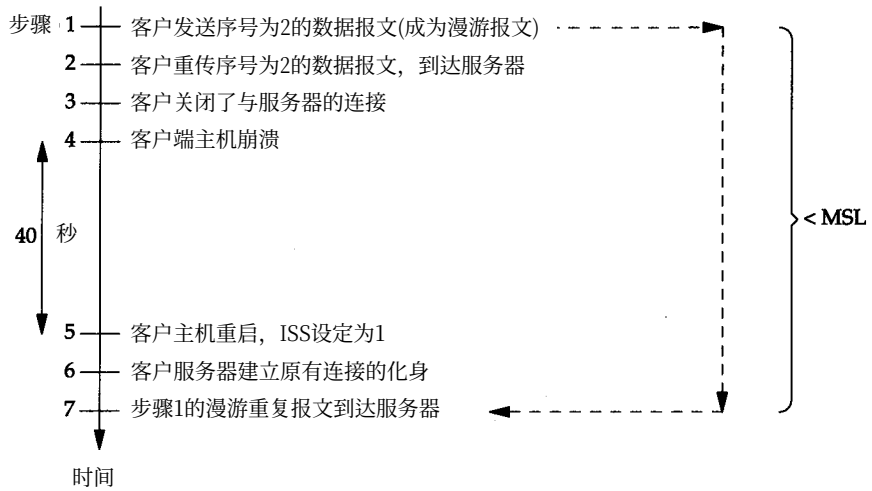


图24-28 示例：旧报文段到达原有连接的化身

即使系统重启后，TCP通过当前时钟计算ISS，问题同样存在。无论原有连接的ISS设为多少，由于序号会回绕，完全有可能重启后新建连接的ISS接近等于重启前原有连接最后使用的序号。

除了保存重启前所有已建连接的序号，解决这个问题的唯一方法就是重启后TCP在MSL内保持平静(不发送任何报文段)。尽管问题有可能出现，但绝大多数TCP中并未实现相应的解决方法，因为多数主机仅重启时间就要长于MSL。

24.9 小结

本章概要介绍了接下来的6章中将要讨论的TCP源代码。TCP为每条连接建立自己的控制块，保存该连接的所有变量和状态信息。

定义了TCP的状态变迁图，TCP在哪些条件下从一个状态变迁到另一个状态，每次变迁过程中发送和接收了哪些报文段。状态变迁图还显示了连接建立和终止的过程。在后续TCP讨论中会经常引用该图。

TCP连接上传输的每个数据字节都有相应的序号，TCP在连接控制块中维护多个序号：有些用于发送，有些用于接收(TCP工作于全双工方式)。由于序号来自有限的32 bit空间，会从

最大值回绕到0。本章解释了如何使用小于和大于测试来比较序号，在后续的 TCP代码中将不断遇到序号的比较。

最后介绍了最简单的 TCP函数，`tcp_init`，完成对Internet PCB的TCP链表的初始化。此外，还讨论了初始发送序号的选取问题。

习题

24.1 研究图24-5中的统计数据，计算每条连接上发送和接收的平均字节数。

24.2 在`tcp_init`中，内核告警是否合理？

24.3 执行`netstat -a`了解你的系统当前有多少个活跃的 TCP端点。

第25章　TCP的定时器

25.1　引言

从本章起，我们开始详细讨论TCP的实现代码，首先熟悉一下在绝大多数TCP函数里都会遇到的各种定时器。

TCP为每条连接建立了七个定时器。按照它们在一条连接生存期内出现的次序，简要介绍如下。

1)“连接建立(connection establishment)”定时器在发送SYN报文段建立一条新连接时启动。如果没有在75秒内收到响应，连接建立将中止。

2)“重传(retransmission)”定时器在TCP发送数据时设定。如果定时器已超时而对端的确认还未到达，TCP将重传数据。重传定时器的值(即TCP等待对端确认的时间)是动态计算的，取决于TCP为该连接测量的往返时间和该报文段已被重传的次数。

3)“延迟ACK(delayed ACK)”定时器在TCP收到必须被确认但无需马上发出确认的数据时设定。TCP等待200 ms后发送确认响应。如果，在这200 ms内，有数据要在该连接上发送，延迟的ACK响应就可随着数据一起发送回对端，称为捎带确认。

4)“持续(persist)”定时器在连接对端通告接收窗口为0，阻止TCP继续发送数据时设定。由于连接对端发送的窗口通告不可靠(只有数据才会被确认，ACK不会被确认)，允许TCP继续发送数据的后续窗口更新有可能丢失。因此，如果TCP有数据要发送，但对端通告接收窗口为0，则持续定时器启动，超时后向对端发送1字节的数据，判定对端接收窗口是否已打开。与重传定时器类似，持续定时器的值也是动态计算的，取决于连接的往返时间，在5秒到60秒之间取值。

5)“保活(keepalive)”定时器在应用进程选取了插口的SO_KEEPALIVE选项时生效。如果连接的连续空闲时间超过2小时，保活定时器超时，向对端发送连接探测报文段，强迫对端响应。如果收到了期待的响应，TCP可确定对端主机工作正常，在该连接再次空闲超过2小时之前，TCP不会再进行保活测试。如果收到的是其他响应，TCP可确定对端主机已重启。如果连续若干次保活测试都未收到响应，TCP就假定对端主机已崩溃，尽管它无法区分是主机故障(例如，系统崩溃而尚未重启)，还是连接故障(例如，中间的路由器发生故障或电话线断了)。

6) FIN_WAIT_2定时器。当某个连接从FIN_WAIT_1状态变迁到FIN_WAIT_2状态(图24-15)，并且不能再接收任何新数据时(意味着应用进程调用了close，而非shutdown，没有利用TCP的半关闭功能)，FIN_WAIT_2定时器启动，设为10分钟。定时器超时后，重新设为75秒，第二次超时后连接被关闭。加入这个定时器的目的是为了避免如果对端一直不发送FIN，某个连接会永远滞留在FIN_WAIT_2状态。

7) TIME_WAIT定时器，一般也称为2MSL定时器。2MSL指两倍的MSL，24.8节定义的最大报文段生存时间。当连接转移到TIME_WAIT状态，即连接主动关闭时，定时器启动。卷1

的18.6节详细说明了需要2MSL等待状态的原因。连接进入TIME_WAIT状态时，定时器设定为1分钟(Net/3选用30秒的MSL)，超时后，TCP控制块和Internet PCB被删除，端口号可重新使用。

TCP包括两个定时器函数：一个函数每200 ms调用一次(快速定时器)；另一个函数每500 ms调用一次(慢速定时器)。延迟ACK定时器与其他6个定时器有所不同：如果某个连接上设定了延迟ACK定时器，那么下一次200 ms定时器超时后，延迟的ACK必须被发送(ACK的延迟时间必须在0~200 ms之间)。其他的定时器每500 ms递减一次，计数器减为0时，就触发相应的动作。

25.2 代码介绍

当某个连接的TCP控制块中的TF_DELACK标志(图24-14)置位时，允许该连接使用延迟ACK定时器。TCP控制块中的t_timer数组包括4个(TCPT_NTIMERS)计数器，用于实现其他的6个定时器。图25-1列出了数组的索引。下面简单地介绍这6个计数器是如何实现除延迟ACK定时器外的其余6个定时器的。

常 量	值	描 述
TCPT_REXMT	0	重传定时器
TCPT_PERSIST	1	持续定时器
TCPT_KEEP	2	保活定时器或连接建立定时器
TCPT_2MSL	3	2MSL定时器或FIN_WAIT_2定时器

图25-1 t_timer 数组索引

t_timer中的每条记录，保存了定时器的剩余值，以500 ms为计时单位。如果等于零，则说明对应的定时器没有设定。由于每个定时器都是短整型，所以定时器的最大值只能设定为16 383.5秒，约为4.5小时。

	建连定时器	重传定时器	延迟ACK定时器	持续定时器	保活定时器	FIN_WAIT_2	2MSL
t_timer[TCPT_REXMT]		•					
t_timer[TCPT_PERSIST]				•			
t_timer[TCPT_KEEP]	•				•		
t_timer[TCPT_2MSL]						•	•
t_flags & TF_DELACK			•				
tcp_keepidle (2小时)					•		
tcp_keepintvl (75秒)					•	•	
tcp_maxidle (10分钟)					•	•	
2 * TCPTV_MSL (60秒)							•
TCPTV_KEEP_INIT (75秒)	•						

图25-2 七个TCP定时器的实现

请注意，图25-1中利用4个“定时计数器”实现了6个TCP“定时器”，这是因为有些定时器彼此间是互斥的。下面我们首先区分一下计数器与定时器。TCPT_KEEP计数器同时实现了保活定时器和连接建立定时器，因为这两个定时器永远不会同时出现在同一条连接上。类似地，2MSL定时器和FIN_WAIT_2定时器都由TCPT_2MSL计数器实现，因为一条连接在同一

时间内只可能处于其中的一种状态。图 25-2的第一行小结了7个TCP定时器的实现方式，第二行和第三行列出了其中4个定时器初始化时用到的3个全局变量(图24-3)和2个常量(图25-3)。注意，有2个全局变量同时被多个定时器使用。前面已讨论过，延迟 ACK定时器直接受控于TCP的200 ms定时器，在本章后续部分将讨论其他 2个定时器的时间长度是如何设定的。

图25-3列出了Net/3实现中基本的定时器取值。

常　量	500ms的时钟滴答数	秒　数	描　述
TCPTV_MSL	60	30	MSL，最大报文段生存时间
TCPTV_MIN	2	1	重传定时器最小值
TCPTV_REXMTMAX	128	64	重传定时器最大值
TCPTV_PERSMIN	10	5	持续定时器最小值
TCPTV_PERSMAX	120	60	持续定时器最大值
TCPTV_KEEP_INIT	150	75	连接建立定时器取值
TCPTV_KEEP_IDLE	14400	7200	第一次保活测试前连接的空闲时间 (2小时)
TCPTV_KEEPINTVL	150	75	对端无响应时保活测试间的间隔时间
TCPTV_SRTTBASE	0		特殊取值，意味着目前无连接 RTT样本
TCPTV_SRTTDFLT	6	3	连接无RTT样本时的默认值

图25-3　TCP实现中基本的定时器取值

图25-4列出了在代码中会遇到的其他定时器常量。

常　量	值	描　述
TCP_LINGERTIME	120	用于SO_LINGER插口选项的最大时间，以秒为单位
TCP_MAXRXTSHIFT	12	等待某个ACK的最大重传次数
TCPTV_KEEPCNT	8	对端无响应时，最大保活测试次数

图25-4　定时器常量

图25-5中定义的TCPT_RANGESET宏，给定时器设定一个给定值，并确认该值在指定范围内。

tcp_timer.h

```
102 #define TCPT_RANGESET(tv, value, tvmin, tvmax) { \
103     (tv) = (value); \
104     if ((tv) < (tvmin)) \
105         (tv) = (tvmin); \
106     else if ((tv) > (tvmax)) \
107         (tv) = (tvmax); \
108 }
```

tcp_timer.h

图25-5　TCPT_RANGESET　宏

从图25-3可知，重传定时器和持续定时器都有最大值和最小值限制，因为它们的取值都是基于测量的往返时间动态计算得到的，其他定时器均设为常值。

本章中将不讨论图25-4中列出的一个特殊定时器：插口的拖延定时器 (linger timer)，这是由插口选项SO_LINGER设置的。这是一个插口级的定时器，由系统函数close使用(15.15节)。在图30-12中读者将看到，插口关闭时， TCP会首先检查该选项是否置位，拖延时间是否为 0。

如果上述条件满足，将不采用TCP正常的关闭过程，连接直接被复位。

25.3 tcp_canceltimers函数

图25-6中定义了tcp_canceltimers函数。连接进入TIME_WAIT状态时，tcp_input在设定2MSL定时器之前，调用该函数。4个定时计数器清零，相应地关闭了重传定时器、持续定时器、保活定时器和FIN_WAIT_2定时器。

tcp_timer.c

```
107 void
108 tcp_canceltimers(tp)
109 struct tcpcb *tp;
110 {
111     int     i;

112     for (i = 0; i < TCPT_NTIMERS; i++)
113         tp->t_timer[i] = 0;
114 }
```

tcp_timer.c

图25-6 tcp_canceltimers 函数

25.4 tcp_fasttimo函数

图25-7定义了tcp_fasttimo函数。该函数每隔200 ms被pr_fasttimo调用一次，用于操作延迟ACK定时器。

tcp_timer.c

```
41 void
42 tcp_fasttimo()
43 {
44     struct inpcb *inp;
45     struct tcpcb *tp;
46     int     s = splnet();

47     inp = tcb.inp_next;
48     if (inp)
49         for (; inp != &tcb; inp = inp->inp_next)
50             if ((tp = (struct tcpcb *) inp->inp_ppcb) &&
51                 (tp->t_flags & TF_DELACK)) {
52                 tp->t_flags &= ~TF_DELACK;
53                 tp->t_flags |= TF_ACKNOW;
54                 tcpstat.tcps_delack++;
55                 (void) tcp_output(tp);
56             }
57     splx(s);
58 }
```

tcp_timer.c

图25-7 tcp_fasttimo 函数，每200 ms调用一次

函数检查TCP链表中每个具有对应TCP控制块的Internet PCB。如果TCP_DELACK标志置位，清除该标志，并置位TF_ACKNOW标志。调用tcp_output，由于TF_ACKNOW标志已置位，ACK被发送。

为什么TCP的PCB链表中的某个Internet PCB会没有相应的TCP控制块(第50行的判断)？读者将在图30-11中看到，创建插口时(PRU_ATTACH请求响应socket系统调用)，首先创建

Inertnet PCB，之后才创建TCP控制块。两个操作间有可能会插入高优先级的时钟中断(图1-13)，该中断有可能调用tcp_fasttimo函数。

25.5 tcp_slowtimo函数

图25-8定义了tcp_slowtimo函数，每隔500ms被pr_slowtimo调用一次。它操作其他6个定时器：连接建立定时器、重传定时器、持续定时器、保活定时器、 FIN_WAIT_2定时器和2MSL定时器。

tcp_timer.c

```
64 void
65 tcp_slowtimo()
66 {
67     struct inpcb *ip, *ipnxt;
68     struct tcpcb *tp;
69     int     s = splnet();
70     int     i;

71     tcp_maxidle = TCPTV_KEEPCNT * tcp_keepintvl;
72     /*
73      * Search through tcb's and update active timers.
74      */
75     ip = tcb.inp_next;
76     if (ip == 0) {
77         splx(s);
78         return;
79     }
80     for (; ip != &tcb; ip = ipnxt) {
81         ipnxt = ip->inp_next;
82         tp = intotcpcb(ip);
83         if (tp == 0)
84             continue;
85         for (i = 0; i < TCPT_NTIMERS; i++) {
86             if (tp->t_timer[i] && --tp->t_timer[i] == 0) {
87                 (void) tcp_usrreq(tp->t_inpcb->inp_socket,
88                                   PRU_SLOWTIMO, (struct mbuf *) 0,
89                                   (struct mbuf *) i, (struct mbuf *) 0);
90                 if (ipnxt->inp_prev != ip)
91                     goto tpgone;
92             }
93         }
94         tp->t_idle++;
95         if (tp->t_rtt)
96             tp->t_rtt++;
97       tpgone:
98         ;
99     }
100    tcp_iss += TCP_ISSINCR / PR_SLOWHZ;     /* increment iss */
101    tcp_now++;                  /* for timestamps */
102    splx(s);
103 }
```

tcp_timer.c

图25-8 tcp_slowtimo 函数，每隔500 ms调用一次

71 tcp_maxidle初始化为10分钟，这是TCP向对端发送连接探测报文段后，收到对端主机响应前的最长等待时间。如图25-6所示，FIN_WAIT_2定时器也使用了这一变量。它的初始化语句可放到tcp_init中，因为其值可在系统初启时设定(见习题25.2)。

1. 检查所有TCP控制块中的所有定时器

72-89 检查TCP链表中每个具有对应TCP控制块的Internet PCB，测试每个连接的所有定时计数器，如果非0，计数器减1。如果减为0，则发送PRU_SLOWTIMO请求。后面会介绍该请求将调用tcp_timers函数。

tcp_usrreq的第四个入口参数是指向mbuf的指针。不过，在不需要mbuf指针的场合，这个参数实际被用于完成其他功能。tcp_slowtimo函数中利用它传递索引i，指出超时的是哪一个时钟。代码中把i强制转换为mbuf指针是为了避免编译错误。

2. 检查TCP控制块是否已被删除

90-93 在检查控制块中的定时器之前，先将指向下一个Internet PCB的指针保存在ipnxt中。每次PRU_SLOWTIMO请求返回后，tcp_slowtimo会检查TCP链表中的下一个PCB是否仍指向当前正处理的PCB。如果不是，则意味着控制块已被删除——也许2MSL定时器超时或重传定时器超时，并且TCP已放弃当前连接——控制转到tpgone，跳过当前控制块的其余定时器，并移至下一个PCB。

3. 计算空闲时间

94 当一个报文段到达当前连接，tcp_input清零控制块中的t_idle。从连接收到最后一个报文段起，每隔500ms t_idle递增一次。空闲时间统计主要有三个目的：(1)TCP在连接空闲2小时后发送连接探测报文段；(2)如果连接位于FIN_WAIT_2状态，且空闲10分钟后又空闲75秒，TCP将关闭该连接；(3)连接空闲一段时间后，tcp_output将返回慢启动状态。

4. 增加RTT计数器

95-96 如果需要测量某个报文段的RTT，tcp_output在发送该报文段时，初始化t_rtt计数器为1。它每500 ms递增一次，直至收到该报文段的确认。在tcp_slowtimo函数中，如果连接正对某个报文段计时，即t_rtt计数器非零，则递增t_rtt。

5. 递增初始发送序号

100 tcp_iss在tcp_init中初始化为1。每500 ms tcp_iss增加64 000: 128 000 (TCP_ISSINCR)除以2 (PR_SLOWHZ)。尽管看上去tcp_iss每秒钟仅递增两次，但实际速率可达每8微秒增加1。后面将介绍，无论主动打开或被动打开，只要建立了一条连接，tcp_iss就会增加64 000。

> RFC 793规定初始发送序号应该约每4微秒增加一次，或每秒钟250 000次。Net/3实现的增加速率只有规定的一半。

6. 递增RFC 1323规定的时间戳值

101 tcp_now在系统重启时初始化为0，每500 ms递增一次，用于实现RFC 1323中定义的时间戳[Jacobson, Barden和Borman 1992]。26.6节中将详细介绍这一功能。

75-79 请注意，如果主机上没有打开的连接(tcb.inp_next为空)，则tcp_iss和则tcp_now的递增将停止。这种状况只可能发生在系统初启时，因为在一个联网的UNIX系统中几乎不可能没有若干活跃的TCP服务器。

25.6 tcp_timers函数

tcp_timers函数在4个TCP定时计数器中的任何一个减为0时由TCP的PRU_SLOWTIMO请求处理代码调用(图30-10)：

```
case PRU_SLOWTIMO:
     tp = tcp_timers(tp, (int)nam);
```

整个函数的结构是一个switch语句，每个定时器对应一个case语句，如图25-9所示。

tcp_timer.c

```
120 struct tcpcb *
121 tcp_timers(tp, timer)
122 struct tcpcb *tp;
123 int     timer;
124 {
125     int     rexmt;
126     switch (timer) {

                /* switch cases */

256     }
257     return (tp);
258 }
```

tcp_timer.c

图25-9 tcp_timers 函数：总体框架

下面我们介绍其中3个定时计数器(5个TCP定时器)，重传定时器留待25.11节中再讨论。

25.6.1 FIN_WAIT_2和2MSL定时器

TCP的TCP2_2MSL定时计数器实现了两种定时器。

1) FIN_WAIT_2定时器。当tcp_input从FIN_WAIT_1状态变迁到FIN_WAIT_2状态，并且插口不再接收任何新数据(意味着应用进程调用了close，而不是shutdown，从而无法利用TCP的半关闭功能)时，FIN_WAIT_2定时器设定为10分钟(tcp_maxidle)。这样可以防止连接永远停留在FIN_WAIT_2状态。

2) 2MSL定时器。当TCP转移到TIME_WAIT状态，2MSL定时器设定为60秒。

图25-10列出了处理2MSL定时器的case语句——在该定时器减为0时执行。

tcp_timer.c

```
127         /*
128          * 2 MSL timeout in shutdown went off.  If we're closed but
129          * still waiting for peer to close and connection has been idle
130          * too long, or if 2MSL time is up from TIME_WAIT, delete connection
131          * control block.  Otherwise, check again in a bit.
132          */
133     case TCPT_2MSL:
134         if (tp->t_state != TCPS_TIME_WAIT &&
135             tp->t_idle <= tcp_maxidle)
136             tp->t_timer[TCPT_2MSL] = tcp_keepintvl;
137         else
138             tp = tcp_close(tp);
139         break;
```

tcp_timer.c

图25-10 tcp_timers 函数：2MSL定时器超时

1. 2MSL定时器

127-139 图25-10中的条件判断逻辑较为复杂，因为TCPT_2MSL计数器的两种不同用法混在了一起(习题25.4)。首先看TIME_WAIT状态，定时器60秒超时后，将调用tcp_close并释

放控制块。图25-11给出了典型的时间顺序，列出了2MSL定时器超时后的一系列函数调用。从图中可看出，如果某个定时器被设定为N秒($2\times N$滴答)，由于定时计数器的第一次递减将发生在其后的0~500 ms之间，定时器将在其后$2\times N-1$和$2\times N$个滴答之间的某个时刻超时。

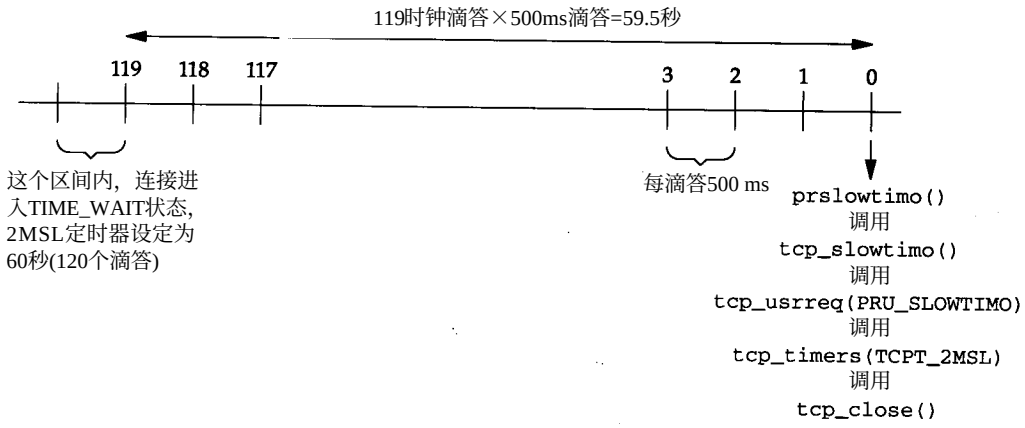


图25-11 TIME_WAIT状态下2MSL定时器的设定与超时

2. FIN_WAIT_2定时器

127-139 如果连接状态不是TIME_WAIT，TCPT_2MSL计数器表示FIN_WAIT_2定时器。只要连接的空闲时间超过10分钟(tcp_maxidle)，连接就会被关闭。但如果连接的空闲时间小于或等于10分钟，FIN_WAIT_2定时器将被设为75秒。图25-12给出了典型的时间顺序。

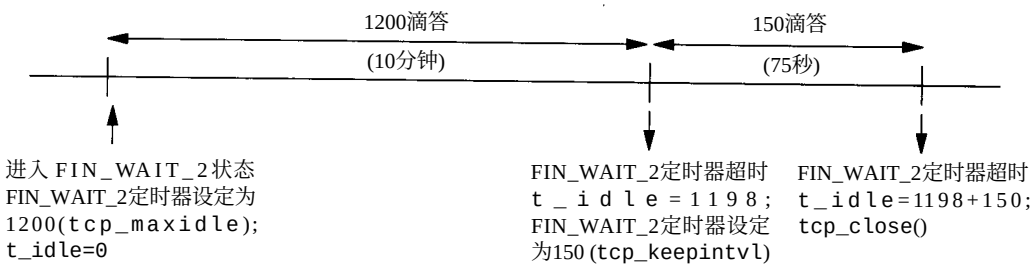


图25-12 FIN_WAIT_2定时器，避免永久滞留于FIN_WAIT_2状态

连接接收到一个ACK后，从FIN_WAIT_1状态变迁到FIN_WAIT_2状态(图24-15)，t_idle被置为0，FIN_WAIT_2定时器设为1200(tcp_maxidle)。图25-12中，向上的箭头指着10分钟定时起始时刻的右侧，强调定时计数器的第一次递减发生在定时器设定后的0~500 ms之间。1199个滴答后，定时器超时。从图25-8中可知，在四个定时计数器递减并做超时判定之后，t_idle才会增加，因此t_idle等于1198(我们假定连接在10分钟内一直空闲)。因为条件表达式“1198小于或等于1200”为真，FIN_WAIT_2定时器设为150(tcp_keepintvl)。定时器75秒后再次超时，假定连接一直空闲，t_idle应为1348，条件表达式为假，tcp_close被调用。

第一次10分钟定时后加入另一个75秒定时是因为除非持续空闲时间超过10分钟，否则处于FIN_WAIT_2状态的连接不会被关闭。如果第一个10分钟定时器还未超时，测试t_idle值是没有意义的，但只要过了这段时间，每隔75秒就会进行一次测试。由于有可能收到重复的报文段，即一个重复的ACK使得连接从FIN_WAIT_1状态变迁到FIN_WAIT_2状态，因此每收到一个报文段，10分钟等待将重新开始(因为t_idle重设为0)。

处于FIN_WAIT_2状态的连接在10分钟空闲后将被关闭，这一点并不符合协议规范，但在实际中是可行的。处于FIN_WAIT_2状态，应用进程调用close，连接上的所有数据都已发送并被确认，FIN已被对端确认，TCP等待对端应用进程调用close。如果对端进程永远不关闭它的连接，本地TCP将一直滞留在FIN_WAIT_2状态。应定义计数器保存由于这种原因而终止的连接数，从而了解这种状况出现的频率。

25.6.2 持续定时器

图25-13给出了处理持续定时器超时的case语句。

tcp_timer.c

```
210         /*
211          * Persistence timer into zero window.
212          * Force a byte to be output, if possible.
213          */
214     case TCPT_PERSIST:
215         tcpstat.tcps_persisttimeo++;
216         tcp_setpersist(tp);
217         tp->t_force = 1;
218         (void) tcp_output(tp);
219         tp->t_force = 0;
220         break;
```

tcp_timer.c

图25-13 tcp_timers 函数：持续定时器超时

强制发送窗口探测报文段

210-220 持续定时器超时后，由于对端已通告接收窗口为0，TCP无法向对端发送数据。此时，tcp_setpersist计算持续定时器的下一个设定值，并存储在TCPT_PERSIST计数器中。t_force标志置位，强制tcp_output发送1字节数据。

图25-14给出了局域网环境下，持续定时器的典型值，假定连接的重传时限为1.5秒(见卷1的图22-1)。

图25-14 持续定时器取值的时间表：探测对端接收窗口

一旦持续定时器取值达到60秒，TCP将每隔60秒发送一次窗口探测报文段。由于持续定时器取值的下限为5秒，上限为60秒，因此定时器头两次均设定为5秒，而不是1.5秒和3秒。从图中可知，定时器采用了指数退避策略，新的取值等于原有值乘以2，25.9节中将介绍这一算法的实现。

25.6.3 连接建立定时器和保活定时器

TCP的TCPT_KEEP计数器实现了两个定时器：

1) 当应用进程调用connect，连接转移到SYN_SENT状态(主动打开)，或者当连接从LISTEN状态变迁到SYN_RCVD状态(被动打开)时，SYN发送之后，将连接建立定时器设定为75秒(TCPTV_KEEP_INIT)。如果75秒内连接未能进入ESTABLISHED状态，则该连接被丢弃。

2) 收到一个报文段后，tcp_input将复位连接的保活定时器，重设为2小时(tcp_keepidle)，并清零连接的t_idle计数器。上述操作适用于系统中所有的TCP连接，无论是否置位了插口的保活选项。如果保活定时器超时(收到最后一个报文段2小时后)，并且置位了插口的保活选项，则TCP将向对端发送连接探测报文段。如果定时器超时，且未置位插口选项，则TCP将只复位定时器，重设为2小时。

图25-15给出了处理TCP的TCPT_KEEP计数器的case语句。

tcp_timer.c

```
221         /*
222          * Keep-alive timer went off; send something
223          * or drop connection if idle for too long.
224          */
225     case TCPT_KEEP:
226         tcpstat.tcps_keeptimeo++;
227         if (tp->t_state < TCPS_ESTABLISHED)
228             goto dropit;        /* connection establishment timer */
229         if (tp->t_inpcb->inp_socket->so_options & SO_KEEPALIVE &&
230             tp->t_state <= TCPS_CLOSE_WAIT) {
231             if (tp->t_idle >= tcp_keepidle + tcp_maxidle)
232                 goto dropit;
233             /*
234              * Send a packet designed to force a response
235              * if the peer is up and reachable:
236              * either an ACK if the connection is still alive,
237              * or an RST if the peer has closed the connection
238              * due to timeout or reboot.
239              * Using sequence number tp->snd_una-1
240              * causes the transmitted zero-length segment
241              * to lie outside the receive window;
242              * by the protocol spec, this requires the
243              * correspondent TCP to respond.
244              */
245             tcpstat.tcps_keepprobe++;
246             tcp_respond(tp, tp->t_template, (struct mbuf *) NULL,
247                         tp->rcv_nxt, tp->snd_una - 1, 0);
248             tp->t_timer[TCPT_KEEP] = tcp_keepintvl;
249         } else
250             tp->t_timer[TCPT_KEEP] = tcp_keepidle;
251         break;
252       dropit:
253         tcpstat.tcps_keepdrops++;
254         tp = tcp_drop(tp, ETIMEDOUT);
255         break;
```

tcp_timer.c

图25-15 tcp_timer 函数：保活时钟超时处理

1. 连接建立定时器75秒后超时

221-228 如果状态小于ESTABLISHED(图24-16)，TCPT_KEEP计数器代表连接建立定时器。定时器超时后，控制转到dropit，调用tcp_drop终止连接，给出差错代码ETIMEDOUT。我们将看到，ETIMEDOUT是默认差错码——例如，连接收到了某个差错报告，比如ICMP的主机不可达，返回应用进程的差错码将变为EHOSTUNREACH，而非默认差错码。

我们将在图30-4中看到，TCP发送SYN的同时初始化了两个定时器：正在讨论的连接建立定时器，设定为75秒，和重传定时器，保证对端无响应时可重传SYN。图25-16给出了这两个

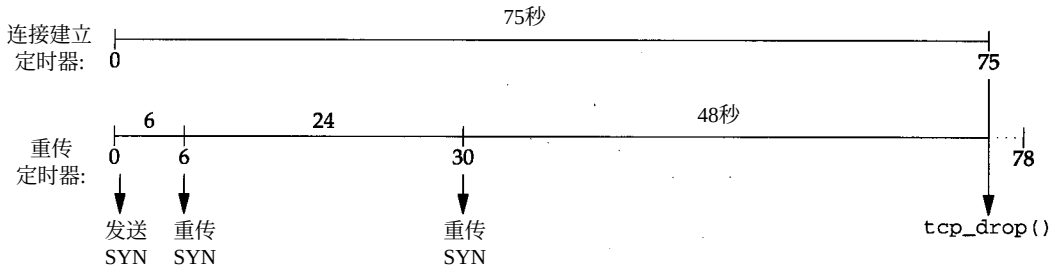


图25-16 SYN发送后：连接建立定时器和重传定时器

定时器。

对于一个新连接，重传定时器初始化为6 秒(图25-19)，后续值分别为24秒和48秒，25.7节中将详细讨论定时器取值的计算方法。重传定时器使得 SYN报文段在0秒、6秒和30秒处连续三次被重传。在75秒处，也就是重传定时器再次超时之前 3秒钟，连接建立定时器超时，调用tcp_drop终止连接。

2. 保活定时器在2小时空闲后超时

229-230 所有连接上的保活定时器在连续 2小时空闲后超时，无论连接是否选取了插口的SO_KEEPALIVE选项。如果插口选项置位，并且连接处于 ESTABLISHED状态或CLOSE_WAIT状态(图24-15)，TCP将发送连接探测报文段。但如果应用进程调用了 close(状态大于CLOSE_WAIT)，即使连接已空闲了2小时，TCP也不会发送连接探测报文段。

3. 无响应时丢弃连接

231-232 如果连接总的空闲时间大于或等于2小时(tcp_keepidle)加10分钟(tcp_maxidle)，连接将被丢弃。也就是说，对端无响应时， TCP最多发送9个连接探测报文段，间隔75秒(tcp_keepintvl)。TCP在确认连接已死亡之前必须发送多个连接探测报文段的一个原因是，对端的响应很可能是不带数据的纯 ACK报文段，TCP无法保证此类报文段的可靠传输，因此，连接探测报文段的响应有可能丢失。

4. 进行保活测试

233-248 如果TCP进行保活测试的次数还在许可范围之内， tcp_respond将发送连接探测报文段。报文段的确认字段(tcp_respond的第四个参数)填入rcv_nxt，期待接收的下一序号；序号字段填入snd_una -1，即对端已确认过的序号(图24-17)。由于这一特定序号落在接收窗口之外，对端必然会发送 ACK，给定它所期待的下一序号。

图25-17小结了保活定时器的用法

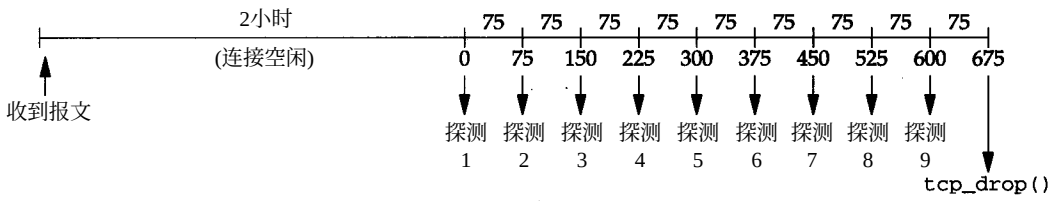


图25-17 保活定时器小结：判定对端是否可达

从0秒起，每隔75秒连续9次发送连接探测报文段，直至 600秒。675秒时(定时器2小时超时后的11.25分钟)连接被丢弃。请注意，尽管常量TCPTV_KEEPCNT(图25-4)的值设为8，却发

送了9次报文段，这是因为代码首先完成定时器递减、与0比较并做可能的处理后才递增变量t_idle(图25-8)。当tcp_input接收了一个报文段，就会复位保活定时器为14400(tcp_keepidle)，并清零t_idle。下一次调用tcp_slowtimo时，定时器减为14339而t_idle增为1。约2小时后，定时器从1减为0时将调用tcp_timers，而此时t_idle的值将为14339。图25-18列出了每次调用tcp_timers时t_idle的取值。

探测次数	图25-17中的时间	t_idle
1	0	14399
2	75	14549
3	150	14699
4	225	14849
5	300	14999
6	375	15149
7	450	15299
8	525	15449
9	600	15599
	675	15749

图25-18 调用tcp_timers 处理保活定时器时t_idle 的取值

图25-15中的代码一直等待t_idle大于或等于15600(tcp_keepidle+tcp_maxidle)，这一事件只可能发生在图25-17中的675秒处，即连续发送了9次连接探测报文段之后。

5. 复位保活定时器

249-250 如果插口选项未置位，或者连接状态大于CLOSE_WAIT，连接的保活定时器将复位，重设为2小时(tcp_keepidle)。

遗憾的是，计数器tcp_keepdrops(253行)不加区分地统计TCPT_KEEP定时计数器的两种不同用法所造成的连接丢弃：连接建立计数器和保活计数器。

25.7 重传定时器的计算

到目前为止，讨论过的定时器的取值都是固定的：延迟 ACK 200ms，连接建立定时器75秒，保活定时器2小时等等。最后两个定时器——重传定时器和持续定时器——的取值依于连接上测算得到的RTT。在讨论实现定时器时限计算和设定的代码之前，首先应理解连接 RTT的测算方法。

TCP的一个基本操作是在发送了需对端确认的报文段后，设置重传定时器。如果在定时器时限范围内未收到 ACK，该报文段被重发。TCP要求对端确认所有数据报文段，不携带数据的报文段则无需确认(例如纯ACK报文段)。如果估算的重传时间过小，响应到达前即超时，造成不必要的重传；如果过大，在报文段丢失之后，发送重传报文段之前将等待一段额外的时间，降低了系统的效率。更为复杂的是，主机间的往返时间动态改变，且变化范围显著。

Net/3中TCP计算重传时限(RTO)时不仅要测量数据报文段的往返时间 (*nticks*)，还要记录已平滑的RTT估计器(*srtt*)和已平滑的RTT平均偏差估计器(*rttvar*)。平均偏差是标准方差的良好近似，计算较为容易，无需标准方差的求平方根运算。 [Jacobson 1988b]讨论了RTT测算的其他细节，给出下面的公式：

$$delta = nticks - srtt$$

$$srtt \leftarrow srtt + g \times delta$$

$$rttvar \leftarrow rttvar + h(|delta| - rttvar)$$

$$RTO = srtt + 4 \times rttvar$$

*delta*是最新测量的往返时间(*nticks*)与当前已平滑的RTT估计器(*srtt*)间的差值。g是用到RTT估计器的增益，设为1/8。h是用到平均偏差估计器的增益，设为1/4。这两个增益和*RTO*计算中的乘数4有意取为2的乘方，从而无需乘、除法，只需简单的移位操作就能够完成运算。

[Jacobson 1988b]规定RTO算式应使用2×*rttvar*，但经过进一步的研究，[Jacobson 1990d]更正为 4×*rttvar*，即Net/1实现中采用的算式。

下面首先介绍TCP重传定时器计算中用到的各种变量和算式，它们在 TCP代码中出现的频率很高。图25-19列出了控制块中与重传定时器有关的变量。

tcpcb的成员	单　位	tcp_newtcpcb初始值	秒　数	描　述
t_srtt	滴答×8	0		已平滑的RTT估计器：*srtt*×8
t_rttvar	滴答×4	24	3	已平滑的RTT平均偏差估计器：*rttvar*×4
t_rxtcur	滴答	12	6	当前重传时限：*RTO*
t_rttmin	滴答	2	1	重传时限最小值
t_rxtshift	不用	0		tcp_backoff[ ]数组索引(指数退避)

图25-19　用于重传定时器计算的控制块变量

tcp_backoff数组将在25.9节末尾定义。tcp_newtcpcb函数设定这些变量的初始值，实现代码将在下一节详细讨论。对变量t_rxtshift中的shift及其上限TCP_MAXRXTSHIFT的命名并不十分准确。它指的并不是比特移位，而是如图 25-19中所声明的，指数组索引。

TCP时限计算中不易理解的地方是已平滑的 RTT估计器和已平滑的RTT平均偏差估计器(t_rtt和t_rttvar)在C代码中都定义为整型，而不是浮点型。这样可以避免内核中的浮点运算，代价是增加了代码的复杂性。

为了区分缩放前和缩放后(scaled)的变量，斜体变量*srtt*和*rttvar*表示前面公式中未缩放的变量，t_srtt和t_rttvar表示TCP控制块中缩放后的变量。

图25-20列出了将遇到的四个常量，它们分别定义了 t_srtt的缩放因子和t_rttvar的缩放因子，分别为8和4。

常　量	值	描　述
TCP_RTT_SCALE	8	相乘：t_srtt=*srtt*×8
TCP_RTT_SHIFT	3	移位：t_srtt=*srtt*<<3
TCP_RTTVAR_SCALE	4	相乘：t_rttvar=*rttvar*×4
TCP_RTTVAR_SHIFT	2	移位：t_rttvar=*rttvar*<<2

图25-20　RTT均值与偏差的乘法与移位

25.8　tcp_newtcpcb算法

图25-21定义了tcp_newtcpcb，分配一个新的TCP控制块并完成初始化。创建新的插口时，TCP的PRU_ATTACH请求将调用它(图30-2)。调用者已事先为该连接分配了一个 Internet PCB，并在入口参数inp中包含指向该结构的指针。我们在这里给出函数代码，是因为它初始化了TCP的定时器变量。

167-175　内核函数malloc分配控制块所需内存，bzero清零新分配的内存块。

176　变量seg_next和seg_prev指向未按正常次序到达当前连接的报文段的重组队列。我们将在27.9节中详细讨论这一重组队列。

tcp_subr.c

```
167 struct tcpcb *
168 tcp_newtcpcb(inp)
169 struct inpcb *inp;
170 {
171     struct tcpcb *tp;

172     tp = malloc(sizeof(*tp), M_PCB, M_NOWAIT);
173     if (tp == NULL)
174         return ((struct tcpcb *) 0);
175     bzero((char *) tp, sizeof(struct tcpcb));
176     tp->seg_next = tp->seg_prev = (struct tcpiphdr *) tp;
177     tp->t_maxseg = tcp_mssdflt;
178     tp->t_flags = tcp_do_rfc1323 ? (TF_REQ_SCALE | TF_REQ_TSTMP) : 0;
179     tp->t_inpcb = inp;
180     /*
181      * Init srtt to TCPTV_SRTTBASE (0), so we can tell that we have no
182      * rtt estimate.  Set rttvar so that srtt + 2 * rttvar gives
183      * reasonable initial retransmit time.
184      */
185     tp->t_srtt = TCPTV_SRTTBASE;
186     tp->t_rttvar = tcp_rttdflt * PR_SLOWHZ << 2;
187     tp->t_rttmin = TCPTV_MIN;
188     TCPT_RANGESET(tp->t_rxtcur,
189                   ((TCPTV_SRTTBASE >> 2) + (TCPTV_SRTTDFLT << 2)) >> 1,
190                   TCPTV_MIN, TCPTV_REXMTMAX);

191     tp->snd_cwnd = TCP_MAXWIN << TCP_MAX_WINSHIFT;
192     tp->snd_ssthresh = TCP_MAXWIN << TCP_MAX_WINSHIFT;

193     inp->inp_ip.ip_ttl = ip_defttl;
194     inp->inp_ppcb = (caddr_t) tp;
195     return (tp);
196 }
```

tcp_subr.c

图25-21 tcp_newtcpcb 函数：创建并初始化一个新的TCP控制块

177-179 发送报文段的最大长度，t_maxseq，默认为512(tcp_mssdflt)。收到对端MSS选项后，它将被tcp_mss函数更改(新连接建立后，TCP也会向对端发送MSS选项)。如果配置要求系统实现 RFC 1313规定的可变窗口和时间戳功能 (图24-3中的全局变量tcp_do_rfc1313，默认值为1)，TF_REQ_SCALE和TF_REQ_TSTMP两个标志将被置位。TCP控制块中的t_inpcb指针将指向由调用者传来的Internet PCB。

180-185 初始化图 25-19中列出的四个变量 t_srtt、t_rttvar、t_rttmin和t_rxtcur。首先，已平滑的RTT估计器被设为0(TCPTV_SRTTBASE)，这个取值非常特殊，指明连接上还不存在RTT估计器。首次进行RTT测量时，tcp_xmit_timer函数将判定已平滑的RTT估计器是否等于0，以采取相应动作。

186-187 已平滑的RTT平均偏差估计器t_rttvar定义为24：3(tcp_rttdflt，图24-3)乘以2(PR_SLOWHZ)后左移2 bit(即乘以4)。由于t_rttvar是变量*rttvar*的4倍，也就等于6个滴答，即3秒钟。*RTO*的最小值，t_rttmin，为2个滴答。

188-190 变量t_rxtcu保存了当前 *RTO*值，以滴答为单位，最小值为 2个滴答(TCPTV_MIN)，最大值为128个滴答(TCPTV_REXMTMAX)。TCPT_RANGESET的第二个参数，表达式计算后等于12个滴答，即6秒钟，是连接的第一个*RTO*值。

理解上述C表达式和RTT缩放值的概念并不是一件容易的事，下面的讨论可能会对您有所

帮助。首先从原始的计算公式开始，并将缩放后的变量替代其中缩放前的变量。下面的算式用于计算第一个*RTO*，以乘数2替代了乘数4。

$$RTO = srtt + 2 \times rttvar$$

使用乘数2而非4是最初4.3BSD Tahoe实现的一个遗留问题[Paxson 1994]。

把下面两个缩放后的变量代入上式：

$$t\_srtt = 8 \times srtt$$

$$t\_rttvar = 4 \times rttvar$$

得到：

$$RTO = \frac{t\_srtt}{8} + 2 \times \frac{t\_rttvar}{4} = \frac{\frac{t\_srtt}{4} + t\_rttvar}{2}$$

也就是图25-21代码中TCPT_RANGESET第二个参数的表达式，只不过用常量——值为6个滴答的TCPTV_SRTTDFLT乘以4后(缩放运算)代替了变量t_rttvar。

191-192 拥塞窗口(snd_cwnd)和慢起动门限(snd_ssthresh)初始化为1 073 725 440 (约为1 G字节)，如是配置了动态窗口选项，这已是TCP窗口大小的上限(卷1的21.6节详细讨论了慢起动和避免拥塞策略)，即TCP首部窗口字段的最大值(65535，TCP_MAXWIN)乘以2^{14}，14是窗口缩放因子的最大值(TCP_MAX_WINSHIFT)。后面将看到，连接上发送或接收了一个SYN时，tcp_mss复位snd_cwnd为1。

193-194 Internet PCB中的IP TTL的默认值初始化为64(ip_defttl)，而PCB则指向新的TCP控制块。

代码中没有明确初始化的其他变量，如移位变量t_rxtshift，均为0，这是因为控制块内存分配后已由bzero清零。

25.9 tcp_setpersist函数

接下来要讨论的函数是tcp_setpersist，它用到了TCP的重传超时算法。从图25-13中可知，持续定时器超时后，将调用此函数。当TCP有数据要发送，而连接对端通告接收窗口为0时，持续定时器启动。图25-22给出了函数实现代码，计算并存储定时器的下个取值。

tcp_output.c

```
493 void
494 tcp_setpersist(tp)
495 struct tcpcb *tp;
496 {
497     t = ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1;

498     if (tp->t_timer[TCPT_REXMT])
499         panic("tcp_output REXMT");
500     /*
501      * Start/restart persistance timer.
502      */
503     TCPT_RANGESET(tp->t_timer[TCPT_PERSIST],
504                   t * tcp_backoff[tp->t_rxtshift],
505                   TCPTV_PERSMIN, TCPTV_PERSMAX);
506     if (tp->t_rxtshift < TCP_MAXRXTSHIFT)
507         tp->t_rxtshift++;
508 }
```

tcp_output.c

图25-22 tcp_setpersist 函数：计算并存储持续定时器的下一次取值

1. 确认重传定时器未设定

493-499 持续定时器设定之前，首先检查确认重传定时器未启动，这是因为两个定时器彼此互斥：如果数据已被发送，说明对端通告的接收窗口必然非零，但持续时钟仅当对端通告零接收窗口时才会设定。

2. 计算RTO

500-505 函数起始处，计算RTO值并存储到变量t中。使用的计算公式为

$$RTO = srtt + 2 \times rttvar$$

与上小节结束时讨论过的公式相同。通过变量替换可得到

$$RTO = \frac{\frac{t\_srtt}{4} + t\_rttvar}{2}$$

即变量t的计算式。

3. 指数退避算法

506-507 *RTO*计算中还用到了指数退避算法，将上式计算得到的*RTO*与tcp_backoff数组中的某个值相乘：

```
int tcp_backoff[ TCP_MAXRXTSHIFT + 1] =
          {1, 2, 4, 8, 16, 32, 64, 64, 64, 64, 64, 64, 64 };
```

tcp_output第一次为连接设置持续定时器的代码是：

```
        tp->t_rxtshift=0;
        tcp_setpersist(tp);
```

因此，第一次调用tcp_setpersist时，t_rxtshift= 0。由于tcp_backoff[0]=1，持续时限等于t。TCPT_RANGESET宏确保*RTO*值位于5秒~60秒之间。t_rxtshift每次增加1，直到最大值12(TCP_MAXRXTSHIFT)，tcp_backoff[12]是数组的最后一个元素。

25.10 tcp_xmit_timer函数

下一个讨论的函数，tcp_xmit_timer，在得到了一个RTT测量值，从而更新已平滑的RTT估计器(*srtt*)和平均偏差(*rttvar*)时被调用。

参数rtt传递了得到的RTT测量值。它的值为*nticks*+1 (与25.7节中的符号一致)，可以通过下面两种方法之一得到。

如果收到的报文段中存在时间戳选项， RTT测量值应等于当前时间(tcp_now)减去时间戳值。我们将在26.6节中讨论时间戳选项，现在只需了解tcp_now每500ms递增一次(图25-8)。发送报文段时，tcp_now做为时间戳被发送，连接对端在相应的ACK中回显该时间戳。

如果未使用时间戳，可以对数据报文计时。从图 25-8可知，连接上的计数器t_rtt每500 ms递增一次。在25.5节也曾提到，该计数器初始化为1，因此收到ACK时，该计数器中的值即为RTT测量值加1(以滴答为单位)。

tcp_input中调用tcp_xmit_timer的典型代码如下：

```
      if (ts_present)
           tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
      else if (tp->rtt  &&  SEQ_GT(ti->ti_ack, tp->t_rtseq))
           tcp_xmit_timer(tp, tp->t_rtt);
```

如果报文段中存在时间戳(ts_present)，RTT测量值等于当前时间(tcp_now)减去回显

的时间戳(ts_ecr)再加1，RTT估计器将被更新(后面将介绍加1的原因)。

如果不存在时间戳，但收到的ACK报文确认了一个正在计时的数据报文，这种情况下RTT估计器也将被更新。每个TCP控制块(t_rtt)中只存在一个RTT计数器，因此，在一条连接上只可能对一个特定数据报文计时。这个报文发送时的起始序号存储在t_rtseq中，与收到的ACK比较，可以确定该报文对应ACK返回的时间。如果收到的确认序号(ti_ack)大于正在计时的数据报文起始序号(t_rtseq)，t_rtt即为RTT新的样本，从而更新RTT估计器。

在支持RFC 1323的时间戳功能之前，t_rtt是TCP测量RTT的唯一方法。但这个变量还用作确认报文段是否被计时的标志（图25-8)：如果t_rtt大于0，则tcp_slowtimo每隔500ms完成t_rtt的加1操作；因此，t_rtt非零时，它等于所用的滴答数再加1。我们将看到，tcp_xmit_timer函数中对得到的第二个参数减1，以纠正上述偏差。因此，使用时间戳时，向tcp_xmit_timer传送的第二个参数必须加1，以保持一致。

序号的大于判定是因为ACK是累积的：如果TCP发送并计时的报文序号为1~1024(t_rtseq等于1)，然后立即发送(但未计时)下一个报文序号为1025~2048，接着收到一个ACK报文，其ti_ack等于2049，它确认了序号1~2048，即同时确认了第一个计时报文和第二个未计时报文。注意，如果使用了RFC 1323定义的时间戳，则不存在序号比较问题。如果对端发送了时间戳选项，意味着它填入了回应时间(ts_ecr)，从而可直接计算RTT。

图25-23给出了函数更新RTT估算值的部分代码。

tcp_input.c

```
1310 void
1311 tcp_xmit_timer(tp, rtt)
1312 struct tcpcb *tp;
1313 short   rtt;
1314 {
1315     short   delta;

1316     tcpstat.tcps_rttupdated++;
1317     if (tp->t_srtt != 0) {
1318         /*
1319          * srtt is stored as fixed point with 3 bits after the
1320          * binary point (i.e., scaled by 8).  The following magic
1321          * is equivalent to the smoothing algorithm in rfc793 with
1322          * an alpha of .875 (srtt = rtt/8 + srtt*7/8 in fixed
1323          * point).  Adjust rtt to origin 0.
1324          */
1325         delta = rtt - 1 - (tp->t_srtt >> TCP_RTT_SHIFT);
1326         if ((tp->t_srtt += delta) <= 0)
1327             tp->t_srtt = 1;
1328         /*
1329          * We accumulate a smoothed rtt variance (actually, a
1330          * smoothed mean difference), then set the retransmit
1331          * timer to smoothed rtt + 4 times the smoothed variance.
1332          * rttvar is stored as fixed point with 2 bits after the
1333          * binary point (scaled by 4).  The following is
1334          * equivalent to rfc793 smoothing with an alpha of .75
1335          * (rttvar = rttvar*3/4 + |delta| / 4).  This replaces
1336          * rfc793's wired-in beta.
1337          */
```

图25-23 tcp_xmit_timer 函数：利用新的RTT测量值计算已平滑的RTT估计器

```
1338            if (delta < 0)
1339                delta = -delta;
1340            delta -= (tp->t_rttvar >> TCP_RTTVAR_SHIFT);
1341            if ((tp->t_rttvar += delta) <= 0)
1342                tp->t_rttvar = 1;
1343        } else {
1344            /*
1345             * No rtt measurement yet - use the unsmoothed rtt.
1346             * Set the variance to half the rtt (so our first
1347             * retransmit happens at 3*rtt).
1348             */
1349            tp->t_srtt = rtt << TCP_RTT_SHIFT;
1350            tp->t_rttvar = rtt << (TCP_RTTVAR_SHIFT - 1);
1351        }
```

tcp_input.c

图25-23 (续)

1. 更新已平滑的RTT估计器

1310-1325 前面已介绍过，tcp_newtcpcb初始化已平滑的RTT估计器(t_srtt)为0，指明连接上不存在RTT估计器。delta是RTT测量值与当前已平滑的RTT估计器间的差值，以未缩放的滴答为单位。t_srtt除以8，单位从缩放后的滴答转换为未缩放的滴答。

1326-1327 已平滑的RTT估计器用以下公式进行更新：

$$srtt \leftarrow srtt + g \times delta$$

由于增益g=1/8，公式变为

$$8 \times srtt \leftarrow 8 \times srtt + delta$$

也就是

$$t\_srtt \leftarrow t\_srtt + delta$$

1328-1342 已平滑的RTT平均偏差估计器的计算公式如下：

$$rttvar \leftarrow rttvar + h(|delta| - rttvar)$$

将h=1/4和缩放后的t_rttvar=4×$rttvar$代入，得到：

$$\frac{t\_rttvar}{4} \leftarrow \frac{t\_rttvar}{4} + \frac{|delta| - \frac{t\_rttvar}{4}}{4}$$

也就是：

$$t\_rttvar \leftarrow t\_rttvar + |delta| - \frac{t\_rttvar}{4}$$

最后一个表达式即为C代码中的表达式。

2. 第一次测量RTT时初始化平滑的估计器值

1343-1350 如果是首次测量某连接的RTT值，已平滑的RTT估计器初始化为测量得到的样本值。下面的计算用到了参数rtt，前面已介绍过rtt等于测量到的RTT值加1($nticks$+1)，而前面公式中用到的delta是从rtt中减1得到的。

$$srtt = nticks + 1$$

或

$$\frac{t\_srtt}{8} = nticks + 1$$

也就是

$$t\_srtt=(nticks+1)\times 8$$

平均偏差等于测量到的RTT值的一半：

$$rttvar = \frac{srtt}{2}$$

也就是

$$\frac{t\_rttvar}{4} = \frac{nticks+1}{2}$$

或者

$$t\_rttvar=(nticks+1)\times 2$$

代码中的注释指出，已平滑的平均偏差的这种初始取值使得 RTO的初始值等于$3\times srtt$。因为

$$RTO=srtt+4\times rttvar$$

替换掉rttvar，得到：

$$RTO = srtt + 4 \times \frac{srtt}{2}$$

也就是：

$$RTO=3\times srtt$$

图25-24给出了tcp_xmit_timer函数最后一部分的代码。

tcp_input.c

```
1352     tp->t_rtt = 0;
1353     tp->t_rxtshift = 0;

1354     /*
1355      * the retransmit should happen at rtt + 4 * rttvar.
1356      * Because of the way we do the smoothing, srtt and rttvar
1357      * will each average +1/2 tick of bias.  When we compute
1358      * the retransmit timer, we want 1/2 tick of rounding and
1359      * 1 extra tick because of +-1/2 tick uncertainty in the
1360      * firing of the timer.  The bias will give us exactly the
1361      * 1.5 tick we need.  But, because the bias is
1362      * statistical, we have to test that we don't drop below
1363      * the minimum feasible timer (which is 2 ticks).
1364      */
1365     TCPT_RANGESET(tp->t_rxtcur, TCP_REXMTVAL(tp),
1366                   tp->t_rttmin, TCPTV_REXMTMAX);

1367     /*
1368      * We received an ack for a packet that wasn't retransmitted;
1369      * it is probably safe to discard any error indications we've
1370      * received recently.  This isn't quite right, but close enough
1371      * for now (a route might have failed after we sent a segment,
1372      * and the return path might not be symmetrical).
1373      */
1374     tp->t_softerror = 0;
1375 }
```

tcp_input.c

图25-24 tcp_xmit_timer 函数：最后一部分

1352-1353 RTT计数器(t_rtt)和重传移位计数器(t_rxtshift)同时复位为0，为下一个报文的发送和计时做准备。

1354-1366 连接的下一个*RTO*(t_rxtcur)计算用到宏

```
#define TCP_REXMTVAL(tp) \
        (((tp)->t_srtt >> TCP_RTT_SHIFT) + (tp)->t_ttvar)
```

其实，这就是我们很熟悉的公式

$$RTO = srtt + 4 \times rttvar$$

用tcp_xmit_timer更新过的缩放后的变量替代上式中的 *srtt*和*rttvar*，得到宏的表达式：

$$RTO = \frac{t\_srtt}{8} + 4 \times \frac{t\_rttvar}{4} = \frac{t\_srtt}{8} + t\_rttvar$$

此外，*RTO*取值应在规定范围之内，最小值为连接上设定的最小 RTO(t_rttmin，t_newtcpcb初始化为2个滴答)，最大值为128个滴答(TCPTV_REXMTMAX)。

3. 清除软错误变量

1367-1374 由于只有当收到了已发送的数据报文的确认时，才会调用tcp_xmit_timer，如果连接上发生了软错误(t_softerror)，该错误将被丢弃。下一节中将详细讨论软错误。

25.11 重传超时：tcp_timers函数

我们现在回到tcp_timers函数，讨论25.6节中未涉及的最后一个case语句：处理重传定时器。如果在*RTO*内没有收到对端对一个已发送数据报的确认，则执行此段代码。

图25-25小结了重传定时器的操作。假定 tcp_output计算的报文首次重传时限为 1.5秒，这是LAN的典型值(参见卷1的图21-1)。

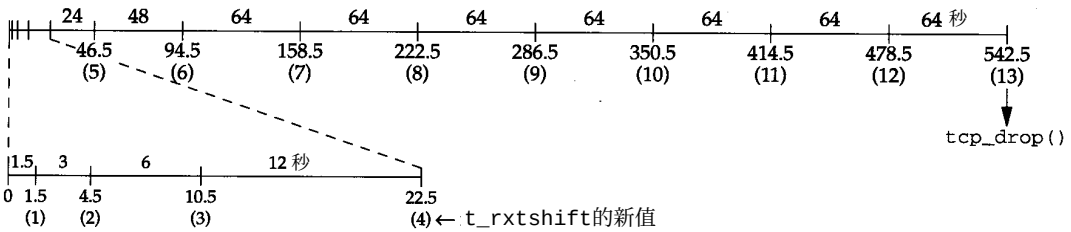


图25-25 发送数据时重传定时器小结

x轴为时间轴，以秒为单位，标注依次为： 0、1.5、4.5等等。这些数字的下方，给出了代码中用到的t_rxtshift的值。连续12次重传后，总共为542.5秒(约9分钟)，TCP将放弃并丢弃连接。

RFC 793建议在建立新连接时，无论主动打开或被动打开，应定义一个参数规定TCP发送数据的总时限，也就是TCP在放弃发送并丢弃连接之前试图传输给定数据报文的总时间。推荐的默认值为5分钟。

RFC 1122要求应用程序必须为连接指定一个参数，限定 TCP总的重传次数或者TCP试图发送数据的总时间。这个参数如果设为“ 无限 ”，那么TCP永不会放弃，还可能不允许终端用户终止连接。

在代码中可看到， Net/3不支持应用程序的上述控制权： TCP放弃传输之前的重传次数是固定的(12)，所用的总时间取决于RTT。

图25-26给出了重传超时case语句的前半部分。

tcp_timer.c

```
140         /*
141          * Retransmission timer went off.  Message has not
142          * been acked within retransmit interval.  Back off
143          * to a longer retransmit interval and retransmit one segment.
144          */
145     case TCPT_REXMT:
146         if (++tp->t_rxtshift > TCP_MAXRXTSHIFT) {
147             tp->t_rxtshift = TCP_MAXRXTSHIFT;
148             tcpstat.tcps_timeoutdrop++;
149             tp = tcp_drop(tp, tp->t_softerror ?
150                           tp->t_softerror : ETIMEDOUT);
151             break;
152         }
153         tcpstat.tcps_rexmttimeo++;
154         rexmt = TCP_REXMTVAL(tp) * tcp_backoff[tp->t_rxtshift];
155         TCPT_RANGESET(tp->t_rxtcur, rexmt,
156                       tp->t_rttmin, TCPTV_REXMTMAX);
157         tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
158         /*
159          * If losing, let the lower level know and try for
160          * a better route.  Also, if we backed off this far,
161          * our srtt estimate is probably bogus.  Clobber it
162          * so we'll take the next rtt measurement as our srtt;
163          * move the current srtt into rttvar to keep the current
164          * retransmit times until then.
165          */
166         if (tp->t_rxtshift > TCP_MAXRXTSHIFT / 4) {
167             in_losing(tp->t_inpcb);
168             tp->t_rttvar += (tp->t_srtt >> TCP_RTT_SHIFT);
169             tp->t_srtt = 0;
170         }
171         tp->snd_nxt = tp->snd_una;
172         /*
173          * If timing a segment in this window, stop the timer.
174          */
175         tp->t_rtt = 0;
```

tcp_timer.c

图25-26 tcp_timers 函数：重传定时器超时，前半部分

1. 递增移位计数器

146 重传移位计数器(t_rxtshift)在每次重传时递增，如果大于12(TCP_MAXRXTSHIFT)，连接将被丢弃。图25-25给出了t_rxtshift每次重传时的取值。请注意两种丢弃连接的区别，由于收不到对端对已发送数据报文的确认而造成的丢弃连接，和由于保活定时器的作用，在长时间空闲且收不到对端响应时丢弃连接。两种情况下，TCP都会向应用进程报告ETIMEDOUT差错，除非连接收到了一个软错误。

2. 丢弃连接

147-152 软错误指不会导致TCP终止已建立的连接或正试图建立的连接的错误，但系统会记录出现的软错误，以备TCP将来放弃连接时参考。例如，如果TCP重传SYN报文段，试图建立新的连接，但未收到响应，TCP将向应用进程报告ETIMEDOUT差错。但如果在重传期间，收到一个ICMP“主机不可达”差错代码，tcp_notify会在t_softerror中存储这一软错误。如果TCP最终决定放弃重传，返回给应用进程的差错代码将为EHOSTUNREACH，而不是

ETIMEDOUT，从而向应用进程提供了更多的信息。如果TCP发送SYN后，对端的响应为RST，这是个硬错误，连接立即被终止，返回差错代码ECONNRFUSED(图28-18)。

3. 计算新的*RTO*

153-157 利用TCP_REXMTVAL宏实现指数退避，计算新的*RTO*值。代码中，给定报文第一次重传时t_rxtshift等于1，因此，*RTO*值为TCP_REXMTVAL计算值的两倍。新的*RTO*值存储在t_rxtcur中，供连接的重传定时器——t_timer[TCPT_REXMT]——使用，tcp_input在启动重传定时器时会用到它(图28-12和图29-6)。

4. 向IP询问更换路由

158-167 如果报文段已重传4次以上，in_losing将释放缓存中的路由(如果存在)，tcp_output再次重传该报文时(图25-27中case语句的结尾处)，将选择一条新的，也许好一些的路由。从图25-25可看到，每次重传定时器超时时，如果重传时限已超过22.5秒，将调用in_losing。

5. 清除RTT估计器

168-170 代码中，已平滑的RTT估计器(t_srtt)被置为0(t_newtcpcb中曾将其初始化为0)，强迫tcp_xmit_timer将下一个RTT测量值做为已平滑的RTT估计器，这是因为报文段重传次数已超过4次，意味着TCP的已平滑的RTT估计器可能已失效。若重传定时器再次超时，进入case语句后，将利用TCP_REXMTVAL计算新的*RTO*值。由于t_srtt被置为0，新的计算值应与本次重传中的计算值相同，再利用指数退避算法加以修正(图25-28中，在42.464秒处的重传很好地说明了上面讨论的概念)。

再次计算*RTO*时，利用公式

$$RTO = \frac{\texttt{t\_srtt}}{8} + \texttt{t\_rttvar}$$

由于t_srtt等于0，*RTO*取值不变。如果报文的重传定时器再次超时(图25-28中从84.064秒到217.84秒)，case语句再次被执行，t_srtt等于0，t_rttvar不变。

6. 强迫重传最早的未确认数据

171 下一个发送序号(snd_nxt)被置为最早的未确认的序号(snd_una)。回想图24-17中，snd_nxt大于snd_una。把snd_nxt回移，将重传最早的未确认过的报文。

7. Karn算法

172-175 RTT计数器，t_rtt，被置为0。Karn算法认为由于该报文即将重传，对该报文的计时也就失去了意义。即使收到了ACK，也无法区分它是对第一次报文，还是对第二次报文的确认。[Karn and Partridge 1987]和卷1的21.3节中都介绍了这一算法。因此，TCP只对未重传报文计时，利用t_rtt计数器得到样本值，并据此修正RTT估计器。在后面的图29-6中将看到，如何使用RFC 1323的时间戳功能取代Karn算法。

25.11.1 慢起动和避免拥塞

图25-27给出了case语句的后半部分，实现慢起动和避免拥塞，并重传最早的未确认过的报文。

由于重传定时器超时，网络中很可能发生了拥塞。这种情况下，需要用到TCP的拥塞避免算法。如果最终收到了对端发送的确认，TCP采用慢起动算法以较慢的速率继续进行数据

传输。卷1的20.6节和21.6节详细讨论了这两种算法。

176-205 win被置为现有窗口大小(接收方通告的窗口大小snd_wnd和发送方拥塞窗口大小snd_cwnd，两者之中的较小值)的一半，以报文为单位，而非字节(因此除以t_maxseg)，最小值为2。它的值等于网络拥塞时现有窗口大小的一半，也就是慢起动门限，t_ssthresh(以字节为单位，因此乘以t_maxseq)。拥塞窗口的大小，snd_cwnd，被置为只容纳1个报文，强迫执行慢起动。上述做法假定造成网络拥塞的原因之一是本地数据发送太快，因此在拥塞发生时，必须降低发送窗口大小。

这段代码放在一对括号中，是因为它是在4.3BSD和Net/1实现之间添加的，并要求有自己的局部变量(win)。

206 连续重复ACK计数器，t_dupacks(用于29.4节中将介绍的快速重传算法)被置为0。我们将在第29章中介绍它在TCP快速重传和快速恢复算法中的用途。

208 tcp_output重新发送包含最早的未确认序号的报文，即由于重传定时器超时引发了报文重传。

tcp_timer.c

```
176         /*
177          * Close the congestion window down to one segment
178          * (we'll open it by one segment for each ack we get).
179          * Since we probably have a window's worth of unacked
180          * data accumulated, this "slow start" keeps us from
181          * dumping all that data as back-to-back packets (which
182          * might overwhelm an intermediate gateway).
183          *
184          * There are two phases to the opening: Initially we
185          * open by one mss on each ack.  This makes the window
186          * size increase exponentially with time.  If the
187          * window is larger than the path can handle, this
188          * exponential growth results in dropped packet(s)
189          * almost immediately.  To get more time between
190          * drops but still "push" the network to take advantage
191          * of improving conditions, we switch from exponential
192          * to linear window opening at some threshhold size.
193          * For a threshhold, we use half the current window
194          * size, truncated to a multiple of the mss.
195          *
196          * (the minimum cwnd that will give us exponential
197          * growth is 2 mss.  We don't allow the threshhold
198          * to go below this.)
199          */
200         {
201             u_int   win = min(tp->snd_wnd, tp->snd_cwnd) / 2 / tp->t_maxseg;
202             if (win < 2)
203                 win = 2;
204             tp->snd_cwnd = tp->t_maxseg;
205             tp->snd_ssthresh = win * tp->t_maxseg;
206             tp->t_dupacks = 0;
207         }
208         (void) tcp_output(tp);
209         break;
```

tcp_timer.c

图25-27 tcp_timer 函数：重传定时器超时，后半部分

25.11.2 精确性

TCP维护的这些估计器的精确性如何呢？首先应指出，因为RTT以500 ms为测量单位，是非常不精确的。已平滑的RTT估计器和平均偏差的精确性要高一些(缩放因子为8和4)，但也不够，LAN的RTT是毫秒级，横跨大陆的RTT约为60ms左右。这些估计器仅仅给出了RTT的上限，从而在设定重传定时器时，可以不考虑由于重传时限过小而造成不必要的重传。

[Brakmo, O'Malley, and Peterson 1994]描述的TCP实现，能够提供高精度的RTT样本。他们的做法是，发送报文段时记录系统时钟读数(精度比以500 ms为测量单位要高得多)，收到ACK时再次读取系统时钟，从而得到高精度的RTT。

Net/3支持的时间戳功能(26.6节)本来可以提供较高精度的RTT，但Net/3将时间戳的精度也定为500 ms。

25.12 一个RTT的例子

下面讨论一个具体的例子，说明上述计算是如何进行的。我们从主机`bsdi`向`vangogh.cs.berkeley.edu`发送12288字节的数据。在发送过程中，故意断开工作中的PPP链路，之后再恢复，看看TCP如何处理报文的超时与重传。为发送数据，我们运行自己的`sock`程序(参见卷1的附录C)，加-D选项，置位插口的`SO_DEBUG`选项(27.10节)。传输结束后，运行`trpt` (8)程序检查留在内核的环形缓存中的调试记录，之后打印TCP控制块中我们感兴趣的时钟变量。

图25-28列出了各变量在不同时刻的值。我们用M：N表示序号M~N-1已被发送。本例中的每个报文段都携带了512字节的数据。符号“ACK M”表示ACK报文的确认字段为M。标注“实际差值(ms)”栏列出了RTT定时器打开时刻和关闭时刻间的时间差值。标注“`rtt` (参数)”栏列出了调用`tcp_xmit_timer`时第二个参数的值：RTT定时器打开时刻和关闭时刻间的滴答数再加1。

`tcp_newtcpcb`函数完成`t_srtt`、`t_rttvar`和`t_rxtcur`的初始化，时刻0.0对应的即为变量初始值。

第一个计时报文是最初的SYN报文，365 ms后收到了对端的ACK，调用`tcp_xmit_timer`，`rtt`参数值为2。由于这是第一个RTT测量值(`t_srtt=0`)，执行图25-23中的`else`语句，计算RTT估计器初始值。

携带1~512字节的数据报文是第二个计时报文，1.259秒时收到对应的ACK，RTT估计器被更新。

从接下来的三个报文可看出，连续报文是如何被确认的。1.260秒时发送携带513~1024字节的报文，并启动定时器。之后又发送了携带1025~1526字节的报文，在2.206秒时收到了对端的ACK，同时确认了已发送的两个报文。RTT估计器被更新，因为ACK确认了正计时报文的起始序号(513)。

2.206秒时发送携带1537~2048字节的报文，并启动定时器。3.132秒时收到对应的ACK，RTT估计器被更新。

对3.132秒时发送的报文段计时，重传定时器设为5个滴答(`t_rxtcur`的当前值)。这时，路由器`sun`和`netb`间的PPP链路中断，几分钟后恢复正常。重传定时器在6.064秒超时，执行图25-26中的代码更新RTT变量。`t_rxtshift`从0增至1，`t_rxtcur`置为10个滴答(指数退

避)，重传最早的未确认过的序号 (snd_una=3073)。5秒钟后，定时器再次超时，t_rxtshift递增为2，重传定时器设为20个滴答。

发送时间	发送	接收	RTT定时器	实际时间差(ms)	rtt参数	t_srtt(8个滴答)	t_rttvar(4个滴答)	t_rxtcur(滴答)	t_rxtshift
0.0	SYN		on			0	24	12	
0.365		SYN,ACK	off	365	2	16	4	6	
0.365	ACK								
0.415	1:513		on						
1.259		ack 513	off	844	2	15	4	5	
1.260	513:1025		on						
1.261	1025:1537								
2.206		ack 1537	off	946	3	16	4	6	
2.206	1537:2049		on						
2.207	2049:2561								
2.209	2561:3073								
3.132		ack 2049	off	926	3	16	3	5	
3.132	3073:3585		on						
3.133	3585:4097								
3.736		ack 2561							
3.736	4097:4609								
3.737	4609:5121								
3.739		ack 3073							
3.739	5121:5633								
3.740	5633:6145								
6.064	3073:3585		off			16	3	10	1
11.264	3073:3585		off			16	3	20	2
21.664	3073:3585		off			16	3	40	3
42.464	3073:3585		off			0	5	80	4
84.064	3073:3585		off			0	5	128	5
150.624	3073:3585		off			0	5	128	6
217.184	3073:3585		off			0	5	128	7
217.944		ack 6145							
217.944	6145:6657		on						
217.945	6657:7169								
218.834		ack 6657	off	890	3	24	6	9	
218.834	7169:7681		on						
218.836	7681:8193								
219.209		ack 7169							
219.209	8193:8705								
219.760		ack 7681	off	926	2	22	7	9	
219.760	8705:9217		on						
220.103		ack 8705							
220.103	9217:9729								
220.105	9729:10241								
220.106	10241:10753								
220.821		ack 9217	off	1061	3	22	6	8	
220.821	10753:11265		on						
221.310		ack 9729							
221.310	11265:11777								
221.312		ack 10241							
221.312	11777:12289								
221.674		ack 10753							
221.955		ack 11265	off	1134	3	22	5	7	

图25-28 实例中的RTT变量值和估计器

42.464秒时，重传定时器再次超时，t_srtt清零，t_rttvar置为5。我们在图25-26的讨论中提到过，此时t_rxtcur运算得到的结果相同(因此，下一次运算的结果应为160)。但

由于t_srtt重置为0，下一次更新RTT估计器时(218.834秒)，与建立一条新的连接相类似，得到的RTT测量值将成为新的已平滑的RTT估计器。

之后继续进行数据传输，并且又多次更新了RTT估计器。

25.13 小结

内核每隔200 ms和500 ms，分别调用tcp_fasttimo函数和tcp_slowtimo函数。这两个函数负责维护TCP为连接建立的各种定时器。

TCP为每条连接维护下列7个定时器：

- 连接建立定时器；
- 重传定时器；
- 延迟ACK定时器；
- 持续定时器；
- FIN_WAIT_2定时器；
- 2MSL定时器；

延迟ACK定时器与其他6个定时器不同，设置它时意味着下一次TCP200 ms定时器超时时，延迟的ACK报文必须被发送。其他6个定时器都是计数器，每次TCP 500 ms定时器超时时，计数器减1。任何一个计数器减为0时，触发TCP完成相应动作：丢弃连接、重传报文、发送连接探测报文等等，这些内容本章中都有详细讨论。由于某些定时器是彼此互斥的，代码用4个计数器实现了这6个定时器，复杂性有所增加。

本章还介绍了重传定时器取值的标准计算方法。TCP为每条连接维护两个RTT估计器：已平滑的RTT估计器(*srtt*)和已平滑的RTT平均偏差估计器(*rttvar*)。尽管算法简单清楚，但由于使用了缩放因子(在不使用内核浮点运算的情况下保证足够的精度)，使得代码较为复杂。

习题

25.1 TCP快速超时处理函数的效率如何？(提示：参考图24-5中列出的延迟ACK的次数)有没有另外的实现方式？

25.2 为什么在tcp_slowtimo函数，而不是在tcp_init函数中初始化tcp_maxidle？

25.3 tcp_slowtimo递增t_idle，前面已介绍过t_idle用于计数从连接上收到最后一个报文起到当前为止的滴答数。TCP是否需要计数从连接上发送最后一个报文段起计时的空闲时间？

25.4 重写图25-10中的代码，分离TCPT_2MSL计数器两种不同用法的处理逻辑。

25.5 图25-12中，连接进入FIN_WIN_2状态75秒后收到一个重复的ACK。会发生什么？

25.6 应用程序设置SO_KEEPALIVE选项时连接已空闲了1小时。第一次连接探测报文在何时发送，1小时后还是2小时后？

25.7 为什么tcp_rttdflt是一个全局变量，而非常量？

25.8 重写与习题25.6有关的代码，实现另一种结果。

第26章 TCP 输 出

26.1 引言

函数tcp_output负责发送报文段，代码中有很多地方都调用了它。

tcp_usrreq在多种请求处理中调用了这一函数：处理 PRU_CONNECT，发送初始SYN；处理PRU_SHUTDOWN， 发送FIN；处理PRU_RCVD，应用进程从插口接收缓存中读取若干数据后可能需要发送新的窗口大小通告；处理 PRU_SEND，发送数据；处理PRU_SENDOOB，发送带外数据。

- tcp_fasttimo调用它发送延迟的ACK；
- tcp_timers在重传定时器超时时，调用它重传报文段；
- tcp_timers在持续定时器超时时，调用它发送窗口探测报文段；
- tcp_drop调用它发送RST；
- tcp_disconnect调用它发送FIN；
- tcp_input在需要输出或需要立即发送ACK时调用它；
- tcp_input在收到一个纯ACK报文段且本地有数据发送时调用它(纯ACK报文段指不携带数据，只确认已接收数据的报文段)；
- tcp_input在连续收到3个重复的ACK时，调用它发送一个单一报文段 (快速重传算法)；

tcp_input首先确定是否有报文段等待发送。除了存在需要发往连接对端的数据外，TCP输出还受到其他许多因素的控制。例如，对端可能通告接收窗口为零，阻止 TCP发送任何数据；Nagle算法阻止TCP发送大量小报文段；慢启动和避免拥塞算法限制 TCP发送的数据量。相反，有些函数置位一些特殊标志，强迫 tcp_output发送报文段，如TF_ACKNOW标志置位意味着必须立即发送一个 ACK。如果tcp_output确定不发送某个报文段，数据 (如果存在)将保留在插口的发送缓存中，等待下一次调用该函数。

26.2 tcp_output概述

tcp_output函数很大，我们将分14个部分予以讨论。图26-1给出了函数的框架结构。

1. 是否等待对端的ACK？

61 如果发送的最大序号(snd_max)等于最早的未确认过的序号 (snd_una)，即不等待对端发送ACK，idle为真。图24-17中，idle应为假，因为序号4~6已发送但还未被确认， TCP在等待对端发送对上述序号的确认。

2. 返回慢启动

62-68 如果TCP不等待对端发送ACK，而且在一个往返时间内也没有收到对端发送的其他报文段，设置拥塞窗口为仅能容纳一个报文段(t_maxseg字节)，从而在发送下一个报文段时，

强迫执行慢启动算法。如果数据传输中出现了显著的停顿（“显著”指停顿时间超过RTT），说明与先前测量RTT时相比，网络条件已发生了变化。Net/3假定出现了最坏情况，因而返回慢起动状态。

tcp_output.c

```
43 int
44 tcp_output(tp)
45 struct tcpcb *tp;
46 {
47     struct socket *so = tp->t_inpcb->inp_socket;
48     long    len, win;
49     int     off, flags, error;
50     struct mbuf *m;
51     struct tcpiphdr *ti;
52     u_char  opt[MAX_TCPOPTLEN];
53     unsigned optlen, hdrlen;
54     int     idle, sendalot;

55     /*
56      * Determine length of data that should be transmitted
57      * and flags that will be used.
58      * If there are some data or critical controls (SYN, RST)
59      * to send, then transmit; otherwise, investigate further.
60      */
61     idle = (tp->snd_max == tp->snd_una);
62     if (idle && tp->t_idle >= tp->t_rxtcur)
63         /*
64          * We have been idle for "a while" and no acks are
65          * expected to clock out any data we send --
66          * slow start to get ack "clock" running again.
67          */
68         tp->snd_cwnd = tp->t_maxseg;

69 again:
70     sendalot = 0;   /* set nonzero if more than one segment to output */

                /* look for a reason to send a segment;  */
                /* goto send if a segment should be sent */

218     /*
219      * No reason to send a segment, just return.
220      */
221     return (0);

222 send:

                /* form output segment, call ip_output() */

489     if (sendalot)
490         goto again;
491     return (0);
492 }
```

tcp_output.c

图26-1 tcp_output 函数：框架结构

3. 发送多个报文段

69-70 控制跳转至send后，调用ip_output发送一个报文段。但如果ip_output确定有

多个报文段需要发送，sendalot置为1，函数将试图发送另一个报文段。因此，ip_output的一次调用能够发送多个报文段。

26.3 决定是否应发送一个报文段

某些情况下，在报文段准备好之前已调用了tcp_output。例如，当插口层从插口的接收缓存中移走数据，传递给用户进程时，会生成PRU_RCVD请求。尽管不一定，但完全有可能因为应用进程取走了大量数据，而使得TCP有必要发送新的窗口通告。tcp_output的前半部分确定是否存在需要发往对端的报文段。如果没有，则函数返回，不执行发送操作。

图26-2给出了判定“是否有报文段发送”测试代码的第一部分。

tcp_output.c

```
 71      off = tp->snd_nxt - tp->snd_una;
 72      win = min(tp->snd_wnd, tp->snd_cwnd);

 73      flags = tcp_outflags[tp->t_state];
 74      /*
 75       * If in persist timeout with window of 0, send 1 byte.
 76       * Otherwise, if window is small but nonzero
 77       * and timer expired, we will send what we can
 78       * and go to transmit state.
 79       */
 80      if (tp->t_force) {
 81          if (win == 0) {
 82              /*
 83               * If we still have some data to send, then
 84               * clear the FIN bit.  Usually this would
 85               * happen below when it realizes that we
 86               * aren't sending all the data.  However,
 87               * if we have exactly 1 byte of unsent data,
 88               * then it won't clear the FIN bit below,
 89               * and if we are in persist state, we wind
 90               * up sending the packet without recording
 91               * that we sent the FIN bit.
 92               *
 93               * We can't just blindly clear the FIN bit,
 94               * because if we don't have any more data
 95               * to send then the probe will be the FIN
 96               * itself.
 97               */
 98              if (off < so->so_snd.sb_cc)
 99                  flags &= ~TH_FIN;
100              win = 1;
101          } else {
102              tp->t_timer[TCPT_PERSIST] = 0;
103              tp->t_rxtshift = 0;
104          }
105      }
```

tcp_output.c

图26-2 tcp_output 函数：强迫数据发送

71-72 off指从发送缓存起始处算起指向第一个待发送字节的偏移量，以字节为单位。它指向的第一个字节为snd_una(已发送但还未被确认的字节)。

win是对端通告的接收窗口大小(snd_wnd)与拥塞窗口大小(snd_cwnd)间的最小值。

73 图24-16给出了tcp_outflags数组，数组值取决于连接的当前状态。flags包括下列标志比特的组合：TH_ACK、TH_FIN、TH_RST和TH_SYN，分别表示需向对端发送的报文段类型。其他两个标志比特，TH_PUSH和TH_URG，如果需要，在报文段发送之前加入，与前4个标志比特是逻辑或的关系。

74-105 t_force标志非零表示持续定时器超时，或者有带外数据需要发送。这两种条件下，调用tcp_output的代码均为：

```
tp->t_force = 1;
error = tcp_output(tp);
tp->t_force = 0;
```

从而强迫TCP发送数据，尽管在正常情况下不会执行任何发送操作。

如果win等于0，连接处于持续状态(因为t_force非零)。如果此时插口的发送缓存中还存在数据，则FIN标志被清除。win必须置为1，以强迫发送一个字节的数据。

如果win非零，即有带外数据需要发送，则持续定时器复位，指数退避算法的索引，t_rxtshift被置为0。

图26-3给出了tcp_output的下一模块，计算发送的数据量。

tcp_output.c

```
106     len = min(so->so_snd.sb_cc, win) - off;
107     if (len < 0) {
108         /*
109          * If FIN has been sent but not acked,
110          * but we haven't been called to retransmit,
111          * len will be -1.  Otherwise, window shrank
112          * after we sent into it.  If window shrank to 0,
113          * cancel pending retransmit and pull snd_nxt
114          * back to (closed) window.  We will enter persist
115          * state below.  If the window didn't close completely,
116          * just wait for an ACK.
117          */
118         len = 0;
119         if (win == 0) {
120             tp->t_timer[TCPT_REXMT] = 0;
121             tp->snd_nxt = tp->snd_una;
122         }
123     }
124     if (len > tp->t_maxseg) {
125         len = tp->t_maxseg;
126         sendalot = 1;
127     }
128     if (SEQ_LT(tp->snd_nxt + len, tp->snd_una + so->so_snd.sb_cc))
129         flags &= ~TH_FIN;

130     win = sbspace(&so->so_rcv);
```

tcp_output.c

图26-3 tcp_output 函数：计算发送的数据量

1. 计算发送的数据量

106 len等于发送缓存中比特数和win(对端通告的接收窗口与拥塞窗口间的最小值，强迫TCP发送数据时也可能等于1字节)，两者间的最小值减去off。减去off是因为发送缓存中的许多字节已发送过，正等待对端的确认。

2. 窗口缩小检查

107-117 造成len小于零的一种可能情况是接收方缩小了窗口，即接收方把窗口的右界移向左侧，下面的例子说明了这种情况。开始时，接收方通告接收窗口大小为 6字节，TCP发送报文段，携带字节 4、5和6。紧接着，TCP又发送一个报文段，携带字节 7、8和9。图26-4显示了两个报文段发送后本地的状态。

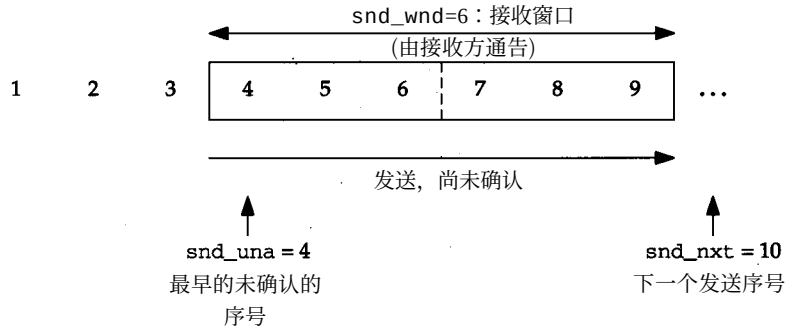


图26-4 发送4~9字节后的本地发送缓存

之后，收到一个ACK，确认序号字段为7(确认所有序号小于7的数据，包括字节6)，但窗口字段为1。接收方缩小了接收窗口，此时本地的状态如图 26-5所示。

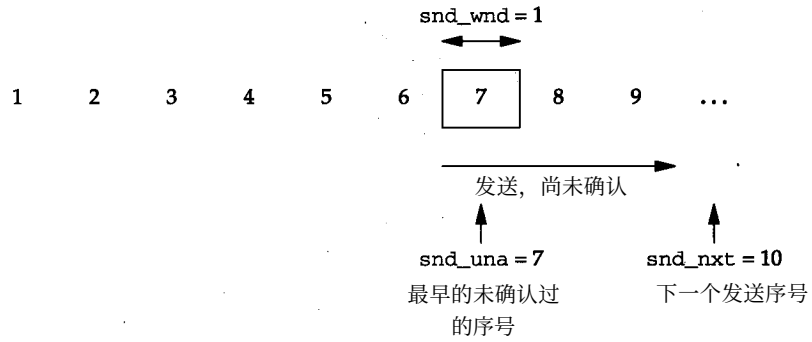


图26-5 收到4~7字节的确认后，本地发送缓存

窗口缩小后，执行图26-2和图26-3中的计算，得到：

```
off = snd_nxt - snd_una = 10 - 7 = 3
win = 1
len = min(so_snd.sb_cc, win) -off = min(3, 1) - 3 = -2
```

假定发送缓存仅包含字节7、8和9。

RFC 793和RFC 1122都非常不赞成缩小窗口。尽管如此，具体实现必须考虑这一问题并加以处理。这种做法遵循了在 RFC 791中首次提出的稳健性原则：“ 对接收报文段的假设尽量少一些，对发送报文段的限制尽量多一些 ”。

造成len小于0的另一种可能情况是，已发送过 FIN，但还未收到确认 (见习题26.2)。图26-6给出了这种情况。

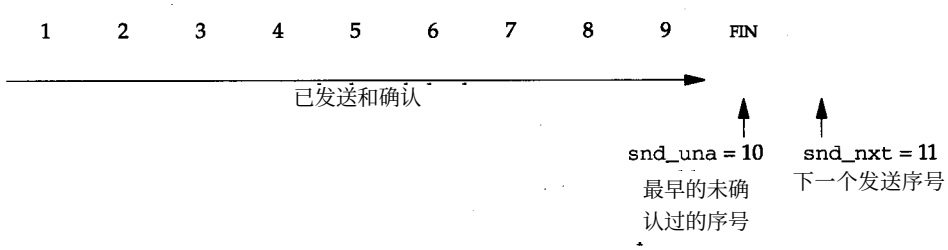


图26-6 字节1~9已发送并收到对端确认，之后关闭连接

图26-6是图26-4的继续，假定字节7~9已被确认，snd_una的当前值为10。应用进程随后关闭连接，向对端发送FIN。在本章后续部分将看到，TCP发送FIN时，snd_nxt将增加1(因为FIN也需要序号)，在本例中，snd_nxt将等于11，而FIN的序号为10。执行图26-2和图26-3中的计算，得到：

```
off = snd_nxt - snd_una = 11 - 10 = 1
win = 6
len = min( so_snd.sb_cc, win ) - off = min(0, 6) - 1 = -1
```

我们假定接收方通告接收窗口大小为6。这个假定无关紧要，因为发送缓存中待发送的字节数(0)小于它。

3. 进入持续状态

118-122 len被置为0。如果对端通告的接收窗口大小为0，则重传定时器将被置为0，任何等待的重传将被取消。令snd_nxt等于snd_una，指针返回发送窗口的最左端，连接将进入持续状态。如果接收方最终打开了接收窗口，则TCP将从发送窗口的最左端开始重传。

4. 一次发送一个报文段

124-127 如果需要发送的数据超过了一个报文段的容量，len置为最大报文段长度，sendalot置为1。如图26-1所示，这将使tcp_output在报文段发送完毕后进入另一次循环。

5. 如果发送缓存不空，关闭FIN标志

128-129 如果本次输出操作未能清空发送缓存，FIN标志必须被清除(防止该标志在flags中被置位)。图26-7举例说明了这一情况。

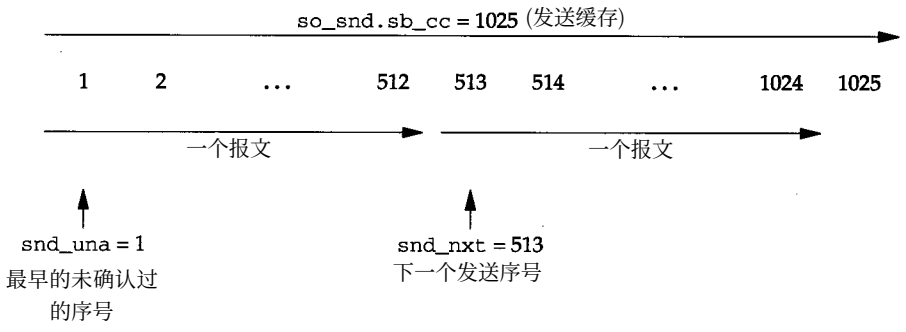


图26-7 实例：FIN置位时，发送缓存不空

这个例子中，第一个512字节的报文段已发送(还未被确认)，TCP正准备发送第二个报文段(512~1024字节)。此时，发送缓存中仍有1字节的数据(1025字节)，应用进程关闭了连接。

len=512(一个报文段)，图26-3中的C表达式变为：

```
SEQ_LT (1025, 1026)
```

如果表达式为真，则FIN标志被清除；否则，TCP无法向对端发送序号为1025的字节。

6. 计算接收窗口大小

130 win设定为本地接收缓存中可用空间的大小，即TCP向对端通告的接收窗口的大小。请注意，这是第二次用到这个变量。在函数前一部分中，它等于允许 TCP发送的最大数据量，但从现在起，它等于本地向对端通告的接收窗口的大小。

糊涂窗口综合症(简写为SWS，详见卷1第22.3节)指连接上交换的都是短报文段，而不是最大长度报文段。这种现象的出现是由于接收方通告的接收窗口过小，或者发送方传输了许多小报文段，因此，避免糊涂窗口综合症，需要发送方和接收方的共同努力。图 26-8给出了发送方避免糊涂窗口综合症的做法。

tcp_output.c

```
131     /*
132      * Sender silly window avoidance.  If connection is idle
133      * and can send all data, a maximum segment,
134      * at least a maximum default-sized segment do it,
135      * or are forced, do it; otherwise don't bother.
136      * If peer's buffer is tiny, then send
137      * when window is at least half open.
138      * If retransmitting (possibly after persist timer forced us
139      * to send into a small window), then must resend.
140      */
141     if (len) {
142         if (len == tp->t_maxseg)
143             goto send;
144         if ((idle || tp->t_flags & TF_NODELAY) &&
145             len + off >= so->so_snd.sb_cc)
146             goto send;
147         if (tp->t_force)
148             goto send;
149         if (len >= tp->max_sndwnd / 2)
150             goto send;
151         if (SEQ_LT(tp->snd_nxt, tp->snd_max))
152             goto send;
153     }
```

tcp_output.c

图26-8 tcp_output 函数：发送方避免糊涂窗口综合症

7. 发送方避免糊涂窗口综合症的方法

142-143 如果待发送报文段是最大长度报文段，则发送它。

144-146 如果无需等待对端的ACK(idle为真)，或者Nagle算法被取消(TF_NODELAY为真)，并且TCP正在清空发送缓存，则发送数据。Nagle算法(详见卷1第19.4节)的思想是：如果某个连接需要等待对端的确认，则不允许 TCP发送长度小于最大长度的报文段。通过设定插口选项TCP_NODELAY，可以取消这个算法。对于正常的交互式连接 (如Telnet或Rlogin)，即使连接上存在未确认过的数据，代码中的 if语句也为假，因为默认条件下TCP会采用Nagle算法。

147-148 如果由于持续定时器超时，或者有带外数据，强迫 TCP执行发送操作，则数据将被发送。

149-150 如果接收方的接收窗口已至少打开了一半，则发送数据。这个限制条件是为了处

理对端一直发送小窗口通告，甚至小于报文段长度的情况。变量 max_sndwnd由tcp_input维护，等于连接对端发送的所有窗口通告中的最大值。实际上， TCP试图猜测对端接收缓存的大小，并假定对端永远不会减小其接收缓存。

151-152 如果重传定时器超时，则必须发送一个报文段。snd_max是已发送过的最高序号，从图25-26可知，重传定时器超时后， snd_nxt将被设为snd_una，即snd_nxt会指向窗口的左侧，从而小于snd_max。

图26-9给出了tcp_output的下一部分，确定 TCP是否必须向对端发送新的窗口通告，称之为“ 窗口更新 ”。

tcp_output.c

```
154     /*
155      * Compare available window to amount of window
156      * known to peer (as advertised window less
157      * next expected input).  If the difference is at least two
158      * max size segments, or at least 50% of the maximum possible
159      * window, then want to send a window update to peer.
160      */
161     if (win > 0) {
162         /*
163          * "adv" is the amount we can increase the window,
164          * taking into account that we are limited by
165          * TCP_MAXWIN << tp->rcv_scale.
166          */
167         long    adv = min(win, (long) TCP_MAXWIN << tp->rcv_scale) -
168         (tp->rcv_adv - tp->rcv_nxt);

169         if (adv >= (long) (2 * tp->t_maxseg))
170             goto send;
171         if (2 * adv >= (long) so->so_rcv.sb_hiwat)
172             goto send;
173     }
```

tcp_output.c

图26-9 tcp_output 函数：判定是否需要发送窗口更新报文

154-168 表达式

```
min (win, (long) TCP_MAXWIN << tp->rcv_scale)
```

等于插口接收缓存可用空间大小 (win)和连接上所允许的最大窗口大小之间的最小值，即 TCP当前能够向对端发送的接收窗口的最大值。表达式

```
(tp->rcv_adv - tp->rcv_nxt)
```

等于TCP最后一次通告的接收窗口中剩余空间的大小，以字节为单位。两者相减得到 adv，窗口已打开的字节数。 tcp_input顺序接收数据时，递增rcv_nxt。tcp_output在通告窗口边界向右移动时，递增rcv_adv(代码见图26-32)。

回想图 24-18，假定收到了字节 4、5和6，并提交给应用进程。图 26-10给出了此时tcp_output中接收缓存的状态。

adv等于3，因为接收空间中还有3个字节(字节10、11和12)等待对端填充。

169-170 如果剩余的接收空间能够容纳两个或两个以上的报文段，则发送窗口更新报文。在收到最大长度报文段后， TCP将确认收到的所有其他报文段：“ 确认所有其他报文段 (ACK-every-other-segment) ”的属性(马上就会看到具体的实例)。

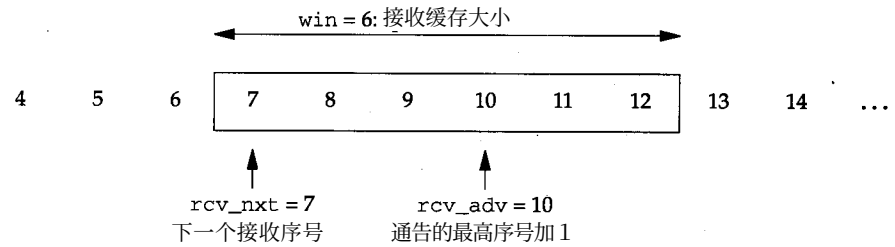


图26-10 收到字节4、5和6后，图24-18中连接的状态变化

171-172 如果可用空间大于插口接收缓存的一半，则发送窗口更新报文。

图26-11给出了tcp_output下一部分的代码，判定输出标志是否置位，要求 TCP发送相应报文段。

tcp_output.c

```
174     /*
175      * Send if we owe peer an ACK.
176      */
177     if (tp->t_flags & TF_ACKNOW)
178         goto send;
179     if (flags & (TH_SYN | TH_RST))
180         goto send;
181     if (SEQ_GT(tp->snd_up, tp->snd_una))
182         goto send;
183     /*
184      * If our state indicates that FIN should be sent
185      * and we have not yet done so, or we're retransmitting the FIN,
186      * then we need to send.
187      */
188     if (flags & TH_FIN &&
189         ((tp->t_flags & TF_SENTFIN) == 0 || tp->snd_nxt == tp->snd_una))
190         goto send;
```

tcp_output.c

图26-11 tcp_output 函数：是否需要发送特定报文段

174-178 如果TF_ACKNOW置位，要求立即发送 ACK，则发送相应报文段。有多种情况可导致TF_ACKNOW置位：200 ms延迟ACK定时器超时，报文段未按顺序到达 (用于快速重传算法)，三次握手时收到了SYN，收到了窗口探测报文，收到了 FIN。

179-180 如果输出标志flags要求发送SYN或RST，则发送相应报文段。

181-182 如果紧急指针，snd_up，超出了发送缓存的起始边界，则发送相应报文段。紧急指针由PRU_SENDOOB请求处理代码(图30-9)负责维护。

183-190 如果输出标志flags要求发送FIN，并且满足下列条件：FIN未发送过或者FIN等待重传，则发送相应报文段。FIN发送后，函数将置位TF_SENTFIN标志。

到目前为止，tcp_output还没有真正发送报文段，图 26-12给出了函数返回前的最后一段代码。

191-217 如果发送缓存中存在需要发送的数据 (so_snd.sb_cc非零)，并且重传定时器和持续定时器都未设定，则启动持续定时器。这是为了处理对端通告的接收窗口过小，无法接收最大长度报文段，而且也没有特殊原因需要发送立即发送报文段的情况。

218-221 由于不需要发送报文段，tcp_output返回。

tcp_output.c

```
191     /*
192      * TCP window updates are not reliable, rather a polling protocol
193      * using 'persist' packets is used to ensure receipt of window
194      * updates.  The three 'states' for the output side are:
195      *  idle                 not doing retransmits or persists
196      *  persisting           to move a small or zero window
197      *  (re)transmitting     and thereby not persisting
198      *
199      * tp->t_timer[TCPT_PERSIST]
200      *      is set when we are in persist state.
201      * tp->t_force
202      *      is set when we are called to send a persist packet.
203      * tp->t_timer[TCPT_REXMT]
204      *      is set when we are retransmitting
205      * The output side is idle when both timers are zero.
206      *
207      * If send window is too small, there is data to transmit, and no
208      * retransmit or persist is pending, then go to persist state.
209      * If nothing happens soon, send when timer expires:
210      * if window is nonzero, transmit what we can,
211      * otherwise force out a byte.
212      */
213     if (so->so_snd.sb_cc && tp->t_timer[TCPT_REXMT] == 0 &&
214         tp->t_timer[TCPT_PERSIST] == 0) {
215         tp->t_rxtshift = 0;
216         tcp_setpersist(tp);
217     }
218     /*
219      * No reason to send a segment, just return.
220      */
221     return (0);
```

tcp_output.c

图26-12 tcp_output 函数：进入持续状态

举例

应用进程向某个空闲的连接写入100字节，接着又写入50字节。假定报文段大小为512字节。在第一次写入操作时，由于连接空闲，且 TCP正在清空发送缓存，图 26-8中的代码(144~146行)被执行，发送一个报文段，携带100字节的数据。

在第二次写入50字节时，图26-8中的代码被执行，但未发送报文段：待发送数据不能构成一个最大长度报文段，连接未空闲 (假定TCP正在等待第一个报文段的 ACK)，默认时采用Nagle算法，`t_force`未置位，并且假定正常情况下接收窗口大小为 4096，50不满足大于等于2048的条件。这50字节的数据将暂留在发送缓存中，也许会一直等到第一个报文段的 ACK到达。由于对端可能延迟发送ACK，最后50字节数据发送前的延迟有可能会更长。

这个例子说明采用Nagle算法时，如果待发送数据无法构成最大长度报文段，如何计算它的延时。参见习题26.12。

举例

本例说明TCP的“确认所有其他报文段”属性。假定连接的报文段大小为 1024字节，接收缓存大小为4096字节。本地不发送数据，只接收数据。

发向对端的对SYN的ACK报文中，通告接收窗口大小为4096，图26-13给出了两个变量rcv_nxt和rcv_adv的初始值。接收缓存为空。

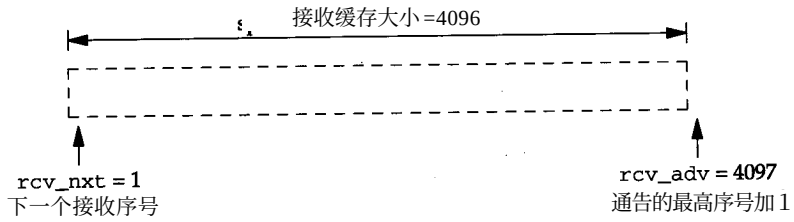


图26-13 接收方通告接收窗口大小为4096

对端发送1~1024字节的报文段，tcp_input处理报文段后，设置连接的延迟ACK标志，把1024字节的数据放入插口的接收缓存中(图28-13)。更新rcv_nxt，如图26-14所示。

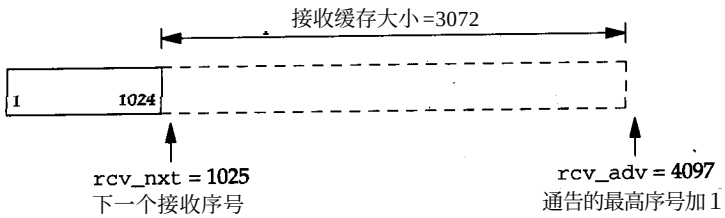


图26-14 图26-13所示的连接在收到1~1024字节后的状态变迁

应用进程从插口的接收缓存中读取 1024字节的数据。从图 30-6中可看到，生成的PRU_RCVD请求在处理过程中会调用 tcp_output，因为应用进程从接收缓存读取数据后，可能需要发送窗口更新报文。当 tcp_output被调用时，rcv_nxt和rcv_adv的值与图26-14相同，唯一的区别是接收缓存的可用空间增加至 4096，因为应用进程从中读取了第一个1024字节的数据。把上述具体数值代入图 26-9中的算式，得到：

```
adv = min(4096, 65535) - (4097 - 1025)
    = 1024
```

TCP_MAXWIN等于65535，我们假定接收窗口大小偏移量为0。由于窗口的增加值小于两个最大报文段长度(2048)，无需发送窗口更新报文。但由于延迟ACK标志置位，如果200ms定时器超时，将发送ACK。

当TCP收到下一个1025~2048字节的报文段时，tcp_input处理后，设定连接的延迟ACK标志(这个标志已置位)，把1024字节的数据放入插口的接收缓存中，更新rcv_nxt，如图26-15所示。

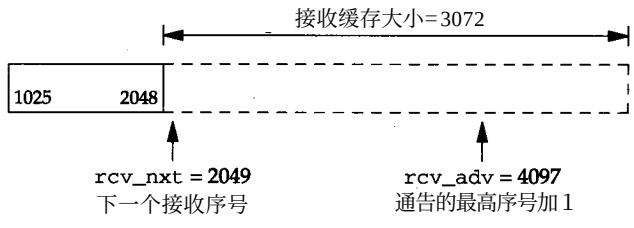


图26-15 图26-14所示的连接收到1025~2048字节后的状态变迁

应用进程读取1024~2048字节的数据，调用tcp_output。rcv_nxt和rcv_adv的值与图26-15相同，尽管应用进程读取1024字节的数据后，接收缓存的可用空间增加至4096。把上述具体数值代入图26-9的算式中，得到：

```
adv = min(4096, 65535) - (4097 - 2049)
    = 2048
```

它等于两个报文段的长度，因此发送窗口更新报文，确认序号字段为 2049，通告窗口字段为4096，表示接收方希望接收序号2049~6145的数据。我们在后面将看到，函数发送完窗口更新报文后，将更新rcv_adv的值为6145。

本例说明了如果数据接收时间少于200ms延迟定时器时限，在有两个或两个以上报文段到达，而且应用进程连续读取数据引起了接收窗口的不断变化时，将发送 ACK，确认所有接收到的报文段。如果有数据到达，但应用进程没有从插口的接收缓存中读取数据，则“确认所有其他报文段”的属性不会出现。相反，发送方只能看到多个延迟 ACK，每个ACK的窗口字段均较前一个要小，直到接收缓存被填满，接收窗口缩小为0。

26.4 TCP选项

TCP首部可以有任选项。由于tcp_output的下一部分代码将试图确定哪些选项需要发送，并据此组织将发送的报文段，下面我们将暂时离开函数代码，转而讨论这些选项。

图26-16列出了Net/3支持的选项格式。

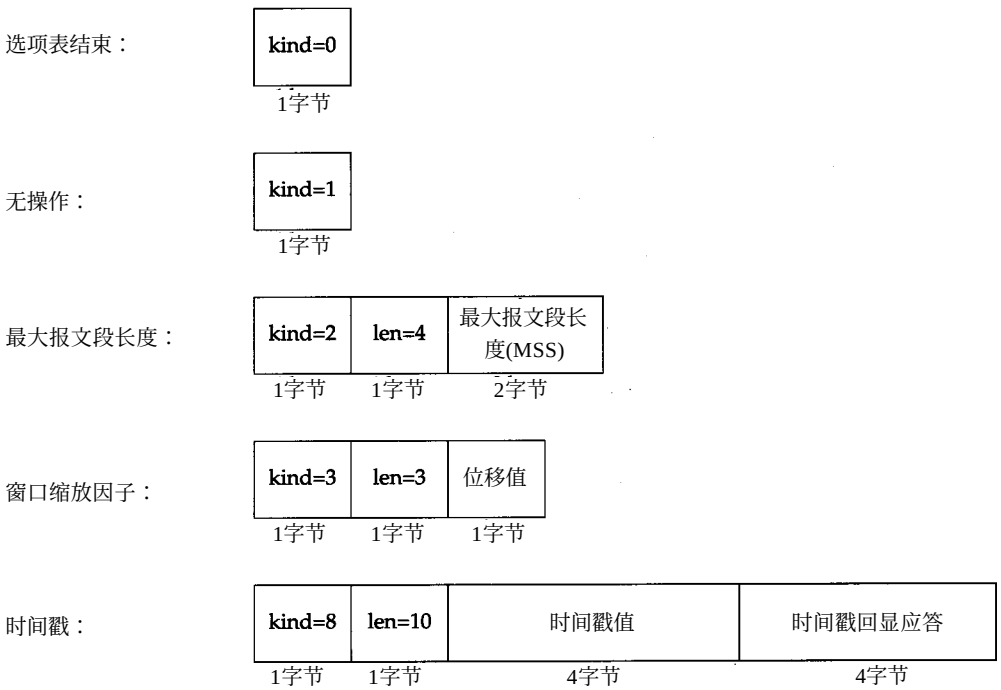


图26-16 Net/3支持的TCP选项

所有选项以1字节的*kind*字段开头，确定选项类型。头两个选项 (*kind*=0或*kind*=1)只有1个字节。其余3个选项都是多字节的，带有*len*字段，位于*kind*字段之后，存储选项的长度。长度

中包括*kind*字段和*len*字段。

多字节整数——MSS和两个时间戳值——遵照网络字节序存储。

最后两个选项，窗口大小和时间戳，是新增的，因此许多系统都不支持。为了与以前的系统兼容，应遵循下列原则：

1) TCP主动打开时(发送不带ACK的SYN)，可以在初始SYN中同时发送这两个选项，或发送其中的任何一个。如果全局变量`tcp_do_rfc1323`非零(默认值等于1)，则Net/3同时支持这两个选项。此项功能由`tcp_newtcpcb`函数实现。

2) 只有对端返回的SYN中包含同样的选项时，才可以使用这些选项。图 28-20和图29-2中的代码实现此类处理。

3) TCP被动打开时，如果收到的 SYN中包含了这两个选项，而且也希望使用这些选项，则发向对端的响应(带有ACK的SYN)中必须包含它们，如图26-23所示。

由于系统必须忽略它不了解的选项，因此新增的选项只有当连接双方都了解这一选项，且同时希望支持它时才会被使用。

27.5节将讨论如何处理MSS选项。下面两节将总结Net/3处理两个新选项的做法：窗口大小和时间戳。

> 还有其他可能的选项。*kinds*等于4、5、6和7，称为选择性ACK和回显选项，在RFC 1072[Jacobson and Braden 1998]中定义。图26-16中并未给出这些选项，因为回显选项已被时间戳选项所代替，选择性 ACK选项目前还未形成正式标准，未在 RFC 1323中出现。此外，处理TCP交易的T/TCP建议(RFC 1644[Braden 1994]和卷1的24.7节)规定了其他3个选项，*kinds*分别为11、12和13。

26.5 窗口大小选项

窗口大小选项，在RFC 1323中定义，避免了TCP首部窗口大小字段只有16 bit的限制(图24-10)。如果网络带宽较高或延时较长 (如，RTT较长)，则需要较大的窗口，称为长肥管道 (long fat pipe)。卷1的第24.3节举例说明了现代网络需要较大的窗口，以获取最大的TCP吞吐量。

图26-16中的偏移量最小值为 0 (无缩放)，最大值为14，即窗口最大可设定为 1 073 725 440 (65535×2^{14})字节。Net/3内部实现时，利用32 bit，而非16 bit整数表示窗口大小。

窗口大小选项只能出现在 SYN中，因此，连接建立后，每个传输方向上的缩放因子是固定不变的。

TCP控制块中的两个变量`snd_scale`和`rcv_scale`，分别规定了发送窗口和接收窗口的偏移量。它们的默认值均为 0，无缩放。每次收到对端发送的窗口通告时， 16 bit的窗口大小值被左移`snd_scale`比特，得到真正的32 bit的对端接收窗口大小(图28-6)。每次准备向对端发送窗口通告时，内部的 32 bit窗口大小值被右移`rcv_scale`比特，得到可填入TCP首部窗口字段的16 bit值。

TCP发送SYN时，无论是主动打开或被动打开，都是根据本地插口接收缓存大小选取`rcv_scale`值，填充窗口大小选项的偏移量字段。

26.6 时间戳选项

RFC 1323中还定义了时间戳选项。发送方在每个报文段中放入时间戳，接收方在 ACK中

将时间戳发回。对于每个收到的ACK，发送方根据返回的时间戳计算相应的RTT样本值。

图26-17总结了时间戳选项所用到的变量。

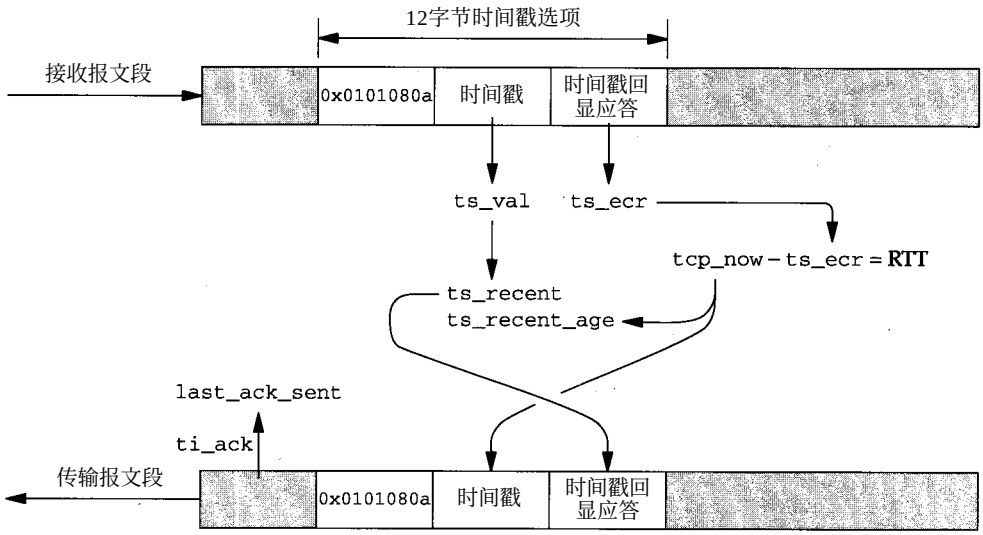


图26-17 时间戳选项中用到的变量小结

全局变量tcp_now是一个时间戳时钟。内核初启时它初始化为0，之后每500 ms增加1(图25-8)。为实现时间戳选项，TCP控制块中定义了下面3个变量：

- ts_recent等于对端发送的最新的有效时间戳(后面很快会介绍什么是“有效的”时间戳)。
- ts_recent_age是最近一次tcp_recent被更新时的tcp_now值。
- last_ack_sent是最近一次发送报文段时确认字段(ti_ack)的值(图26-32)。除非ACK被延迟，正常情况下，它等于rcv_nxt，下一个等待接收的序号。

tcp_input函数中的两个局部变量ts_val和ts_ecr，保存时间戳选项的两个值：

- ts_val是对端发送的数据中携带的时间戳。
- ts_ecr是由收到的报文段确认的本地发送报文段中携带的时间戳。

发送报文段中，时间戳选项的前4个字节为0x0101080a，这是RFC 1323附录A中建议的填充值。第一和第二字节都等于1，为NOP；第三字节为*kind*字段，等于8；第四字节为*len*字段，等于10。在选项之前添加两个NOP后，紧接着的两个32 bit时间戳和后续数据都可按照32 bit边界对齐。此外，图26-17中还给出了接收到的时间戳选项，同样采用了推荐的12字节格式(Net/3通常生成的格式)。不过，处理接收选项的应用进程代码(图28-10)，并不要求必须使用此格式。图26-16中定义的10字节格式中，没有两个前导的NOP，对端接收处理代码一样工作正常(参见习题28.4)。

从发送报文段至收到其ACK间的RTT等于tcp_now减ts_ecr，单位为500ms滴答，因为这是Net/3时间戳的单位。

时间戳选项还可以支持TCP执行PAWS：防止序号回绕(protection against wrapped sequence number)。28.7节将详细讨论这一算法。PAWS中会用到ts_recent_age变量。

tcp_output向输出报文段中填充时间戳选项时，复制tcp_now到时间戳字段，复制

ts_recent到时间戳回显字段(图26-24)。如果连接采用了时间戳选项，则必须为所有输出报文段执行这一操作，除非RST标志置位。

26.6.1 哪个时间戳需要回显，RFC 1323算法

TCP通过时间戳的有效性测试决定是否更新ts_recent，因为这个变量会被填充到时间戳回显字段中，也就决定了对端发送的哪个时间戳需要回显。RFC 1323规定了下面的测试条件：

```
ti_seq <= last_ack_sent < ti_seq + ti_len
```

图26-18中的C代码实现它。

```
if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
    SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len)) {
      tp->ts_recent_age = tcp_now;
      tp->ts_recent = ts_val;
}
```

图26-18 判定接收时间戳是否有效的典型代码

如果收到的报文段中携带时间戳选项，则变量ts_present为真。我们在tcp_input中两次遇到这段代码：图28-11首部预测代码中的测试；和图28-35正常输入处理中的测试。

为了理解测试条件的具体含义，图26-19给出了5种不同的实例，分别对应于连接上收到的5种不同的报文段。每个例子中，ti_len都等于3。

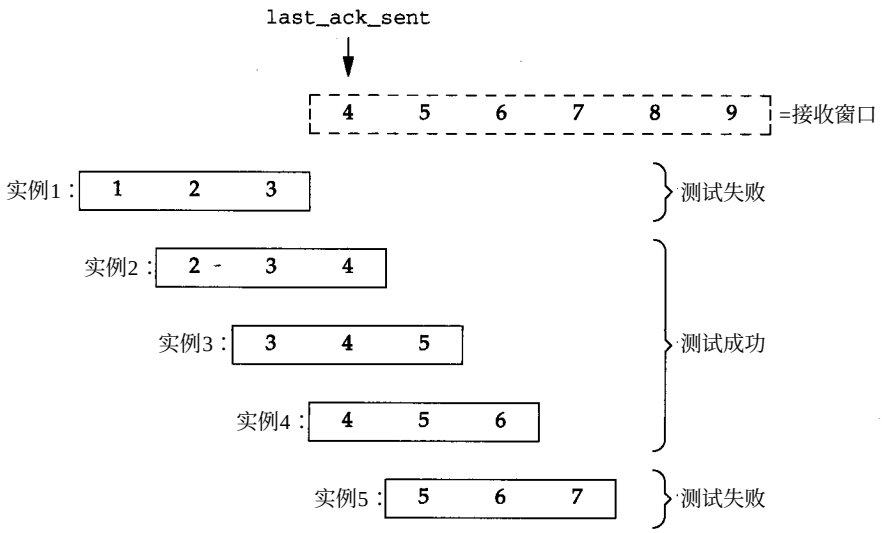


图26-19 举例：收到5个不同报文段时的接收窗口

接收窗口左边界的序号从4开始。实例1中，报文段中携带的全部是重复数据。图28-11中的SEQ_LEQ测试为真，但SEQ_LT测试失败。对于实例2、3和4，由于收到其中任何一个报文段，接收窗口左边界都会增加，SEQ_LEQ和SEQ_LT测试都为真，尽管实例2中包含2个重复

数据，实例 3中也包含 1个重复数据。实例 5，由于它无法增加接收窗口左边界，所以SEQ_LEQ测试失败。这是一个未来报文段，而非等待的下一个报文段，意味着它前面的报文段丢失或报文段序列错误。

不幸的是，这个用于判定是否更新ts_recent的测试条件存在问题[Braden 1993]，考虑下面的例子。

1) 假定图 26-19中的连接开始时收到了一个报文段，携带字节1、2和3。因为last_ack_sent等于1，报文段的时间戳被保存到ts_recent中。发送ACK，确认序号为4，last_ack_sent设为4(rcv_nxt的值)，得到如图26-19所示的接收窗口。

2) ACK丢失。

3) 对端超时后重传前一个报文段，携带字节 1、2和3，即为图26-19中实例1的报文段。由于图26-18中的SEQ_LT测试失败，ts_recent不会更新为重传报文段中的值。

4) TCP发送一个重复的ACK，确认序号为4，但时间戳回显字段填入的ts_recent，即从步骤1的原始报文段中获取的时间戳值。接收方利用这个值计算 RTT时，将(不正确地)计入原始传输、丢失的ACK、定时器超时、重传和重复 ACK，得到它们的总时延。

为了使对端能够正确地计算RTT，重发ACK中应该携带重传报文中的时间戳值。

图26-18中的测试在收到的报文长度为 0时，由于无法移动接收窗口左边界，同样不能更新rs_recent。此外，这个错误的测试条件还会造成生存时间过长的 (大于24天，参见28.7节中讨论的PAWS限制)、单方向的(数据流只在一个方向上存在，从而数据发送方总是输出相同的ACK)连接。

26.6.2 哪个时间戳需要回显，正确的算法

Net/3源代码中使用了图26-18所示的算法。[Braden 1993]定义了正确的算法，如图 26-20所示。

```
if (ts_present && TSTMP_GEQ(ts_val, tp->ts_recent) &&
    SEQ_LEQ(ti->ti_seq, tp->last_ack_sent)) {
```

图26-20 判定接收时间戳是否有效的正确代码

它不关心接收窗口左侧是否移动，只确认新的时间戳 (ts_val)大于等于前一个时间戳(ts_recent)，并且接收到的报文段的起始序号不大于窗口的左边界。图 26-19中实例5的报文仍旧无法通过新的测试，因为这是一个乱序报文。

宏TSTMP_GEQ与图24-21中的SEQ_GEQ相同。它用于处理时间戳，因为时间戳是 32 bit的无符号整数，与序号一样存在回绕的问题。

26.6.3 时间戳与延迟ACK

正确理解延迟 ACK是如何影响时间戳和 RTT计算是很重要的。回想图 26-17，TCP把ts_recent填入到发送报文段的时间戳回显字段中，对端据此计算新的 RTT样本值。如果ACK被延迟，对端计算时应把延迟时间也考虑在内，否则会造成频繁重传。下面的例子中，我们使用图26-20中的代码，不过图26-18的代码也能正确处理延迟ACK。

考虑图26-21所示的接收窗口收到携带字节4和5的报文段时的变化。

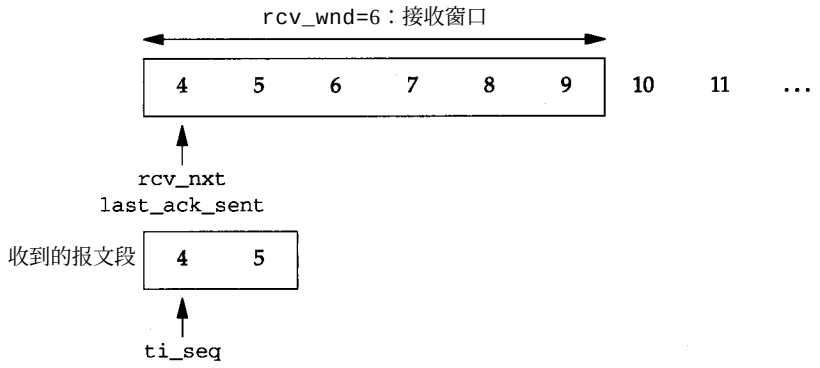


图26-21 当字节4和5到达时的接收序号空间

由于ti_seq小于等于last_ack_sent，ts_recent被更新。rcv_nxt增加2。

假定对这两个字节的ACK被延迟，而且在延迟ACK发送之前，收到了下一个按序到达的报文段，如图26-22所示。

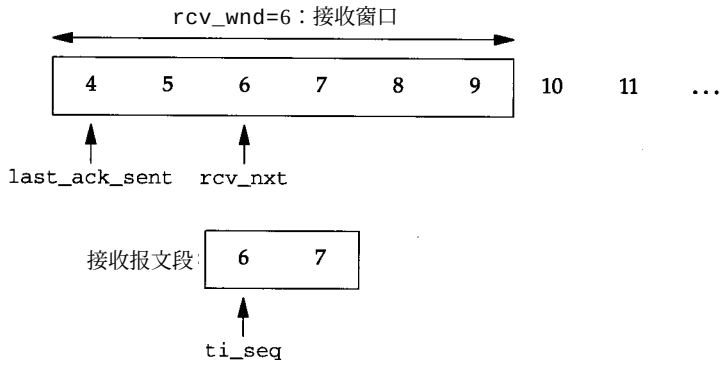


图26-22 当字节6和7到达时的接收序号空间

这一次ti_seq大于last_ack_sent，因此，不会更新ts_recent。这样做是有目的的。假定TCP现在发送确认序号4~7的ACK，对端据此了解存在延迟ACK，因为时间戳回显字段填入的是携带序号4和5的报文段的时间戳值(图26-24)。图26-22还说明了除非使用了延迟ACK，否则，rcv_nxt应该等于last_ack_sent。

26.7 发送一个报文段

tcp_output接下来的代码负责发送报文段——填充TCP报文首部的所有字段，并传递给IP层准备发送。

图26-23给出了这段代码的第一部分，发送SYN报文段，携带MSS选项和窗口大小选项。

223-234 TCP选项字段构建时用到数组opt，整数optlen记录累积的字节数(因为一次可发送多个选项)。如果SYN标志置位，snd_nxt复位为初始发送序号(iss)。如果主动打开，则创建TCP控制块时在PRU_CONNECT请求处理中对iss赋值；如果被动打开，则tcp_input创建TCP控制块的同时对iss赋值。两种情况下，iss都等于全局变量tcp_iss。

235 查看标志TF_NOOPT。但事实上，这个标志永远都不会置位，因为没有代码实现置位操

作。因此，SYN报文段中必然存在MSS选项。

Net/1版的tcp_newtcpcb中，初始化t_flags为0的代码旁有一条注释“发送选项！”。TF_NOOPT标志很可能是从早期的Net/1版本中遗留下来的问题。早期版本发送MSS选项时与其他主机系统不兼容，只好默认设置不发送这一选项。

tcp_output.c

```
223     /*
224      * Before ESTABLISHED, force sending of initial options
225      * unless TCP set not to do any options.
226      * NOTE: we assume that the IP/TCP header plus TCP options
227      * always fit in a single mbuf, leaving room for a maximum
228      * link header, i.e.
229      *  max_linkhdr + sizeof (struct tcpiphdr) + optlen <= MHLEN
230      */
231     optlen = 0;
232     hdrlen = sizeof(struct tcpiphdr);
233     if (flags & TH_SYN) {
234         tp->snd_nxt = tp->iss;
235         if ((tp->t_flags & TF_NOOPT) == 0) {
236             u_short mss;

237             opt[0] = TCPOPT_MAXSEG;
238             opt[1] = 4;
239             mss = htons((u_short) tcp_mss(tp, 0));
240             bcopy((caddr_t) & mss, (caddr_t) (opt + 2), sizeof(mss));
241             optlen = 4;

242             if ((tp->t_flags & TF_REQ_SCALE) &&
243                 ((flags & TH_ACK) == 0 ||
244                  (tp->t_flags & TF_RCVD_SCALE))) {
245                 *((u_long *) (opt + optlen)) = htonl(TCPOPT_NOP << 24 |
246                                                      TCPOPT_WINDOW << 16 |
247                                                      TCPOLEN_WINDOW << 8 |
248                                                      tp->request_r_scale);
249                 optlen += 4;
250             }
251         }
252     }
```

tcp_output.c

图26-23 tcp_output 函数：发送第一个SYN时加入选项

1. 构造MSS选项

236-241 opt[0]等于2(TCPOPT_MAXSEG)，opt[1]等于4，即MSS选项长度，以字节为单位。函数tcp_mss计算准备向对端发送的MSS值，27.5节将讨论这个函数。bcopy把16 bit的MSS存储到opt[2]和opt[3]中(习题26.5)。注意，Net/3总是在建立连接的SYN中发送MSS。

2. 是否发送窗口大小选项

242-244 即使TCP请求窗口大小功能，也只有在主动打开(TH_ACK未置位)时，或者被动打开但对端SYN中已包含了窗口大小选项时，才会发送这一选项。回想图 25-21中TCP控制块创建时，如果全局变量 tcp_do_rfc1323非零(默认值)，那么t_flags就等于TF_REQ_SCALE|TF_REQ_TSTMP。

3. 构造窗口大小选项

245-249 由于窗口大小选项占用3个字节(图26-16)，在它前面加入1字节的NOP，强迫其长

度为4字节，从而后续数据都可以按照4字节边界对齐。如果主动打开，则在PRU_CONNECT请求处理代码中计算request_r_scale。如果被动打开，则tcp_input在收到SYN时计算窗口大小因子。

RFC 1323规定如果TCP支持缩放窗口，即使自己的偏移量为0，也应该发送窗口大小选项。因为这个选项有两个目的：通知对端自己支持此选项；通告本地的偏移量。即使TCP计算得到的本地偏移量为0，对端可能希望使用不同的值。

图26-24给出了tcp_output的下一部分，完成在外出报文段中构造选项。

tcp_output.c

```
253     /*
254      * Send a timestamp and echo-reply if this is a SYN and our side
255      * wants to use timestamps (TF_REQ_TSTMP is set) or both our side
256      * and our peer have sent timestamps in our SYN's.
257      */
258     if ((tp->t_flags & (TF_REQ_TSTMP | TF_NOOPT)) == TF_REQ_TSTMP &&
259         (flags & TH_RST) == 0 &&
260         ((flags & (TH_SYN | TH_ACK)) == TH_SYN ||
261          (tp->t_flags & TF_RCVD_TSTMP))) {
262         u_long *lp = (u_long *) (opt + optlen);

263         /* Form timestamp option as shown in appendix A of RFC 1323. */
264         *lp++ = htonl(TCPOPT_TSTAMP_HDR);
265         *lp++ = htonl(tcp_now);
266         *lp = htonl(tp->ts_recent);
267         optlen += TCPOLEN_TSTAMP_APPA;
268     }
269     hdrlen += optlen;

270     /*
271      * Adjust data length if insertion of options will
272      * bump the packet length beyond the t_maxseg length.
273      */
274     if (len > tp->t_maxseg - optlen) {
275         len = tp->t_maxseg - optlen;
276         sendalot = 1;
277     }
```

tcp_output.c

图26-24 tcp_output 函数：完成发送选项构造

4. 是否需要发送时间戳

253-261 如果下列3个条件均为真，则发送时间戳选项：(1)TCP当前配置要求支持时间戳选项；(2)正在构造的报文段不包含RST标志；(3)主动打开(flags中SYN标志置位，ACK标志未置位)，或者TCP收到了对端发送的时间戳(TF_RCVS_TSTMP)。与MSS和窗口大小选项不同，只要连接双方都同意支持它，时间戳可加入到任意报文段中。

5. 构造时间戳选项

263-267 时间戳选项(26.6节)占用12字节(TCPOLEN_TSTAMP_APPA)。头4个字节为0x0101080a(常量TCPOPT_TSTAMP_HDR)，如图26-17所示。时间戳值等于tcp_now(系统初启到现在的500ms滴答数)。时间戳回显字段值等于由tcp_input设定的ts_recent。

6. 选项加入后是否会造成报文段长度越界

270-277 加入选项后，TCP首部长度会增加optlen字节。如果发送数据的长度(len)大于

MSS减去选项长度(optlen)，则必须相应地减少数据量，并置位sendalot标志，强迫函数发送完当前报文段后进入另一个循环(图26-1)。

MSS和窗口大小选项只出现在SYN报文段中。由于Net/3不在SYN中添加用户数据，因此数据长度的调整对这两个选项不起作用。但如果存在时间戳选项，它可以出现在所有报文段中，从而降低了一次可发送的数据量。最大长度报文段可携带的数据从通告的MSS降至MSS减去12字节。

图26-25给出了tcp_output下一部分代码，更新部分统计值，并为IP和TCP首部分配mbuf。它在输出报文段携带有用户数据(len大于0)时执行。

tcp_output.c

```
278     /*
279      * Grab a header mbuf, attaching a copy of data to
280      * be transmitted, and initialize the header from
281      * the template for sends on this connection.
282      */
283     if (len) {
284         if (tp->t_force && len == 1)
285             tcpstat.tcps_sndprobe++;
286         else if (SEQ_LT(tp->snd_nxt, tp->snd_max)) {
287             tcpstat.tcps_sndrexmitpack++;
288             tcpstat.tcps_sndrexmitbyte += len;
289         } else {
290             tcpstat.tcps_sndpack++;
291             tcpstat.tcps_sndbyte += len;
292         }
293         MGETHDR(m, M_DONTWAIT, MT_HEADER);
294         if (m == NULL) {
295             error = ENOBUFS;
296             goto out;
297         }
298         m->m_data += max_linkhdr;
299         m->m_len = hdrlen;
300         if (len <= MHLEN - hdrlen - max_linkhdr) {
301             m_copydata(so->so_snd.sb_mb, off, (int) len,
302                        mtod(m, caddr_t) + hdrlen);
303             m->m_len += len;
304         } else {
305             m->m_next = m_copy(so->so_snd.sb_mb, off, (int) len);
306             if (m->m_next == 0)
307                 len = 0;
308         }
309         /*
310          * If we're sending everything we've got, set PUSH.
311          * (This will keep happy those implementations that
312          * give data to the user only when a buffer fills or
313          * a PUSH comes in.)
314          */
315         if (off + len == so->so_snd.sb_cc)
316             flags |= TH_PUSH;
```

tcp_output.c

图26-25 tcp_output 函数：更新统计值，为IP和TCP首部分配mbuf

7. 更新统计值

284-292 如果t_force非零，且用户数据只有1字节，可知是一个窗口探测报文。如果snd_nxt小于snd_max，则是一个重传报文。其他的都是正常的数据传输报文。

8. 为IP和TCP首部分配mbuf

293-297 MGETHDR为带有数据分组首部的mbuf分配内存，mbuf中保存IP和TCP的首部及可能的数据(若空间允许)。尽管tcp_output调用通常作为系统调用的一部分(如，write)，它也可在软件中断级由tcp_input调用，或作为定时器处理的一部分。因此，定义了M_DONTWAIT。如果返回错误，控制跳转至“out”处。它位于函数的末尾，如图26-32所示。

9. 向mbuf中复制数据

298-308 如果数据少于44字节(100-40-16，假定没有TCP选项)，数据由m_copydata直接从插口的发送缓存中复制到新的数据组首部mbuf中。若数据量较大，m_copy创建新的mbuf链表，复制插口发送缓存中的数据，最后与前面创建的数据组首部mbuf链接。回想2.9节中介绍过的m_copy函数，如果数据本身已是一个簇，m_copy将不复制，只引用这个簇。

10. 置位PSH标志

309-316 如果TCP发送了从发送缓存得到的所有数据，则PSH标志被置位。如同注释中提到的，这是因为有些接收系统只有在收到PSH标志或者接收缓存已满时，才会向应用程序递交收到的数据。我们在tcp_input中将看到，Net/3绝不会为了等待PSH标志，而把数据滞留在接收缓存中。

图26-26给出了tcp_output下一部分的代码，从在len等于0时执行的else语句开始，处理不携带用户数据的TCP报文段。

tcp_output.c

```
317     } else {                        /* len == 0 */
318         if (tp->t_flags & TF_ACKNOW)
319             tcpstat.tcps_sndacks++;
320         else if (flags & (TH_SYN | TH_FIN | TH_RST))
321             tcpstat.tcps_sndctrl++;
322         else if (SEQ_GT(tp->snd_up, tp->snd_una))
323             tcpstat.tcps_sndurg++;
324         else
325             tcpstat.tcps_sndwinup++;

326         MGETHDR(m, M_DONTWAIT, MT_HEADER);
327         if (m == NULL) {
328             error = ENOBUFS;
329             goto out;
330         }
331         m->m_data += max_linkhdr;
332         m->m_len = hdrlen;
333     }
334     m->m_pkthdr.rcvif = (struct ifnet *) 0;
335     ti = mtod(m, struct tcpiphdr *);
336     if (tp->t_template == 0)
337         panic("tcp_output");
338     bcopy((caddr_t) tp->t_template, (caddr_t) ti, sizeof(struct tcpiphdr));
```

tcp_output.c

图26-26 tcp_output 函数：更新统计值，为IP和TCP首部分配mbuf

11. 更新统计值

318-325 需要更新的统计值有：TF_ACKNOW和长度为0说明是一个纯ACK报文段。如果SYN、FIN或RST中任何一个置位，即为控制报文段。如果紧急指针超过snd_una，是为了

通知对端紧急指针的位置。如果上述条件均为假，则是窗口更新报文段。

12. 得到存储IP和TCP首部的mbuf

326-335 为带有数据包组首部的mbuf分配内存，以保存IP和TCP的首部。

13. 向mbuf中复制IP和TCP首部模板

336-338 bcopy把IP和TCP首部模板从t_template复制到mbuf中。这个模板由tcp_template创建。

图26-27给出了tcp_output下一部分的代码，填充TCP首部剩余的字段。

tcp_output.c

```
339     /*
340      * Fill in fields, remembering maximum advertised
341      * window for use in delaying messages about window sizes.
342      * If resending a FIN, be sure not to use a new sequence number.
343      */
344     if (flags & TH_FIN && tp->t_flags & TF_SENTFIN &&
345         tp->snd_nxt == tp->snd_max)
346         tp->snd_nxt--;
347     /*
348      * If we are doing retransmissions, then snd_nxt will
349      * not reflect the first unsent octet.  For ACK only
350      * packets, we do not want the sequence number of the
351      * retransmitted packet, we want the sequence number
352      * of the next unsent octet.  So, if there is no data
353      * (and no SYN or FIN), use snd_max instead of snd_nxt
354      * when filling in ti_seq.  But if we are in persist
355      * state, snd_max might reflect one byte beyond the
356      * right edge of the window, so use snd_nxt in that
357      * case, since we know we aren't doing a retransmission.
358      * (retransmit and persist are mutually exclusive...)
359      */
360     if (len || (flags & (TH_SYN | TH_FIN)) || tp->t_timer[TCPT_PERSIST])
361         ti->ti_seq = htonl(tp->snd_nxt);
362     else
363         ti->ti_seq = htonl(tp->snd_max);
364     ti->ti_ack = htonl(tp->rcv_nxt);
365     if (optlen) {
366         bcopy((caddr_t) opt, (caddr_t) (ti + 1), optlen);
367         ti->ti_off = (sizeof(struct tcphdr) + optlen) >> 2;
368     }
369     ti->ti_flags = flags;
```

tcp_output.c

图26-27 tcp_output 函数：置位ti_seq 、ti_ack 和ti_flags

14. 如果FIN将重传，递减snd_nxt

339-346 如果TCP已经发送过FIN，则发送序列空间如图26-28所示。因此，如果TH_FIN置位，则TF_SENTFIN也置位，并且snd_nxt等于snd_max，可知FIN等待重传。不久将看到(图26-31)，发送FIN时，snd_nxt会递增1(由于FIN也要占用一个序号)，因此，这里的代码递减snd_nxt。

15. 设置报文段的序号字段

347-363 报文段的序号字段通常等于snd_nxt，但在满足下列条件时，应等于snd_max：如果(1) 不传输数据(len等于0)；(2) SYN标志和FIN标志都未置位；(3) 持续定时器未置位。

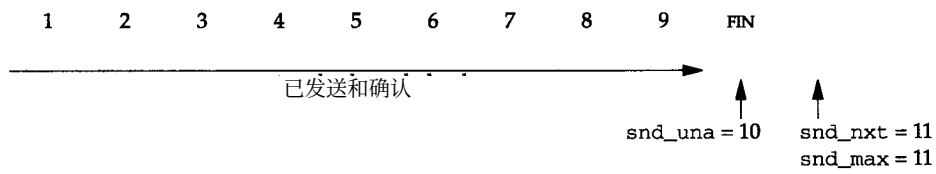


图26-28 FIN发送后的发送序列空间

16. 设置报文段的确认字段

364 报文段的确认字段通常等于rcv_nxt，期待接收的下一个序号。

17. 如果存在首部选项，设置首部长度字段

365-368 如果存在TCP选项(optlen大于0)，代码把选项内容复制到TCP首部，TCP首部4 bit的首部长度字段(图24-10的th_off)等于TCP首部的固定长度(20字节)加上选项总长度后除以4。这个字段是以32 bit为单位的首部长度值，包括TCP选项。

369 TCP首部的标志字段根据变量flags设定。

图26-29给出了下一部分的代码，填充TCP首部其他字段，并计算TCP检验和。

tcp_output.c

```
370     /*
371      * Calculate receive window.  Don't shrink window,
372      * but avoid silly window syndrome.
373      */
374     if (win < (long) (so->so_rcv.sb_hiwat / 4) && win < (long) tp->t_maxseg)
375         win = 0;
376     if (win > (long) TCP_MAXWIN << tp->rcv_scale)
377         win = (long) TCP_MAXWIN << tp->rcv_scale;
378     if (win < (long) (tp->rcv_adv - tp->rcv_nxt))
379         win = (long) (tp->rcv_adv - tp->rcv_nxt);
380     ti->ti_win = htons((u_short) (win >> tp->rcv_scale));
381     if (SEQ_GT(tp->snd_up, tp->snd_nxt)) {
382         ti->ti_urp = htons((u_short) (tp->snd_up - tp->snd_nxt));
383         ti->ti_flags |= TH_URG;
384     } else
385         /*
386          * If no urgent pointer to send, then we pull
387          * the urgent pointer to the left edge of the send window
388          * so that it doesn't drift into the send window on sequence
389          * number wraparound.
390          */
391         tp->snd_up = tp->snd_una;   /* drag it along */
392     /*
393      * Put TCP length in extended header, and then
394      * checksum extended header and data.
395      */
396     if (len + optlen)
397         ti->ti_len = htons((u_short) (sizeof(struct tcphdr) +
398                                       optlen + len));
399     ti->ti_sum = in_cksum(m, (int) (hdrlen + len));
```

tcp_output.c

图26-29 tcp_output 函数：填充其他TCP首部字段并计算检验和

18. 通告的窗口大小应大于最大报文段长度

370-375 计算向对端通告的窗口大小(ti_win)时，应考虑如何避免糊涂窗口综合症。回想图26-3结尾处，win等于插口的接收缓存大小。如果win小于接收缓存大小的1/4(so_rcv.sb_hiwat)，并且小于一个最大报文段长度，则通告的窗口大小设为0，从而在后续测试中防止窗口缩小。也就是说，如果可用空间已达到接收缓存大小的1/4，或者等于最大报文段长度，将向对端发送窗口更新通告。

19. 遵守连接的通告窗口大小的上限

376-377 如果win大于连接规定的最大值，应将其减少为最大值。

20. 不要缩小窗口

378-379 回想图26-10中，rcv_adv减去rcv_nxt等于最近一次向发送方通告的窗口大小中的剩余空间。如果win小于它，应将其设定为该值，因为不允许缩小窗口。有时尽管剩余的可用空间小于最大报文段长度(因此，win在代码起始处被置为0)，但还可以容纳一些数据，就会出现这种情况。卷1中的图22-3举例说明了这一现象。

21. 设置紧急数据偏移量

381-383 如果紧急指针(snd_up)大于snd_nxt，则TCP处于紧急方式。TCP首部的紧急数据偏移量字段设定为以报文段起始序号为基准的紧急指针的16 bit偏移量，并且置位URG标志。无论所指向的紧急数据是否包含在当前处理的报文段中，TCP都会发送紧急数据偏移量和URG标志。

图26-30举例说明了如何计算紧急数据偏移量，假定应用进程执行了

```
send(fd, buf, 3, MSG_OOB);
```

并且调用send时发送缓存为空。这种做法表明基于Berkeley的系统认为紧急指针应指向带外数据后的第一个字节。回想图24-10中，我们区分了数据流中32 bit的紧急指针(snd_up)，和TCP首部中的16 bit紧急数据偏移量(ti_urp)。

这里有个小错误。无论是否采用窗口大小选项，如果发送缓存大于65535，并且几乎为空，则应用进程发送带外数据时，从snd_nxt算起的紧急指针的偏移量有可能超过65535。但偏移量是一个16 bit的无符号整数，如果计算结果超过65535，高位16 bit被丢弃，发送到对端的数据必然是错误的。解决办法参见习题26.6。

384-391 如果TCP不处于紧急方式，则紧急指针移向窗口的最左端(snd_una)。

392-399 TCP长度存储在伪首部中以计算TCP检验和。到目前为止，TCP首部的所有字段已填充完毕，而且从t_template复制IP和TCP首部模板时(图26-26)，对伪首部中用到的IP首部部分字段预先做了初始化(见图23-19中UDP检验和的计算)。

图26-31给出了tcp_output下一部分的代码，SYN或FIN标志置位时更新序号，并启动重传定时器。

22. 保存起始序号

400-405 如果TCP不处于持续状态，则起始序号保存在startseq中。图26-31中的代码在对报文段计时时用到这一变量。

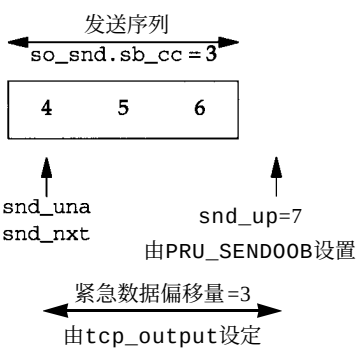


图26-30 紧急指针与紧急数据偏移量计算举例

tcp_output.c

```
400     /*
401      * In transmit state, time the transmission and arrange for
402      * the retransmit.  In persist state, just set snd_max.
403      */
404     if (tp->t_force == 0 || tp->t_timer[TCPT_PERSIST] == 0) {
405         tcp_seq startseq = tp->snd_nxt;
406         /*
407          * Advance snd_nxt over sequence space of this segment.
408          */
409         if (flags & (TH_SYN | TH_FIN)) {
410             if (flags & TH_SYN)
411                 tp->snd_nxt++;
412             if (flags & TH_FIN) {
413                 tp->snd_nxt++;
414                 tp->t_flags |= TF_SENTFIN;
415             }
416         }
417         tp->snd_nxt += len;
418         if (SEQ_GT(tp->snd_nxt, tp->snd_max)) {
419             tp->snd_max = tp->snd_nxt;
420             /*
421              * Time this transmission if not a retransmission and
422              * not currently timing anything.
423              */
424             if (tp->t_rtt == 0) {
425                 tp->t_rtt = 1;
426                 tp->t_rtseq = startseq;
427                 tcpstat.tcps_segstimed++;
428             }
429         }
430         /*
431          * Set retransmit timer if not currently set,
432          * and not doing an ack or a keepalive probe.
433          * Initial value for retransmit timer is smoothed
434          * round-trip time + 2 * round-trip time variance.
435          * Initialize  counter which is used for backoff
436          * of retransmit time.
437          */
438         if (tp->t_timer[TCPT_REXMT] == 0 &&
439             tp->snd_nxt != tp->snd_una) {
440             tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
441             if (tp->t_timer[TCPT_PERSIST]) {
442                 tp->t_timer[TCPT_PERSIST] = 0;
443                 tp->t_rxtshift = 0;
444             }
445         }
446     } else if (SEQ_GT(tp->snd_nxt + len, tp->snd_max))
447         tp->snd_max = tp->snd_nxt + len;
```

tcp_output.c

图26-31 tcp_output 函数：更新序号并启动重传定时器

23. 增加snd_nxt

406-417 由于SYN和FIN都占用一个序号，其中任一标志置位，snd_nxt都必须增加。FIN发送过后，TF_SENTFIN将置位。之后，snd_nxt增加发送的数据字节数(len)，可以为0。

24. 更新snd_max

418-419 如果snd_nxt的最新值大于snd_max，则不是重传报文。snd_max值被更新。

420-428 如果连接目前还没有RTT值(t_rtt=0)，则定时器启动(t_rtt=1)，计时报文段的起始序号保存在t_rtseq中。tcp_input利用它确定计时报文段ACK的到达时间，从而更新RTT。根据25.10节中的讨论，代码应为：

```
if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->t_rtseq))
   tcp_xmit_timer(tp, tp->t_rtt);
```

25. 设定重传定时器

430-440 如果重传定时器还未启动，并且报文段中有数据，则重传定时器时限设定为t_rxtcur。前面已经介绍过，通过测量RTT样本值，tcp_xmit_timer将更新t_rxtcur。但如果snd_nxt等于snd_una(此时snd_nxt中已加入了len)，则是一个纯ACK报文段，而只有在发送数据报文段时才需要启动重传定时器。

441-444 如果持续定时器已启动，则关闭它。对于给定连接，可以在任何时候启动重传定时器或者持续定时器，但两者不允许同时存在。

26. 持续状态

446-447 由于t_force非零，而且持续定时器已设定，可知连接处于持续状态(与图26-31起始处的if语句配对的else语句)。需要时，更新snd_max。处于持续状态时，len应等于1。

tcp_output的最后一部分，在图 26-32中给出，输出报文段准备完毕，调用ip_output发送数据报。

tcp_output.c

```
448     /*
449      * Trace.
450      */
451     if (so->so_options & SO_DEBUG)
452         tcp_trace(TA_OUTPUT, tp->t_state, tp, ti, 0);

453     /*
454      * Fill in IP length and desired time to live and
455      * send to IP level.  There should be a better way
456      * to handle ttl and tos; we could keep them in
457      * the template, but need a way to checksum without them.
458      */
459     m->m_pkthdr.len = hdrlen + len;
460     ((struct ip *) ti)->ip_len = m->m_pkthdr.len;
461     ((struct ip *) ti)->ip_ttl = tp->t_inpcb->inp_ip.ip_ttl;     /* XXX */
462     ((struct ip *) ti)->ip_tos = tp->t_inpcb->inp_ip.ip_tos;     /* XXX */
463     error = ip_output(m, tp->t_inpcb->inp_options, &tp->t_inpcb->inp_route,
464                       so->so_options & SO_DONTROUTE, 0);
465     if (error) {
466       out:
467         if (error == ENOBUFS) {
468             tcp_quench(tp->t_inpcb, 0);
469             return (0);
470         }
471         if ((error == EHOSTUNREACH || error == ENETDOWN)
472             && TCPS_HAVERCVDSYN(tp->t_state)) {
473             tp->t_softerror = error;
474             return (0);
475         }
476         return (error);
477     }
```

图26-32 tcp_output 函数：调用 ip_output 发送报文段

```
478     tcpstat.tcps_sndtotal++;

479     /*
480      * Data sent (as far as we can tell).
481      * If this advertises a larger window than any other segment,
482      * then remember the size of the advertised window.
483      * Any pending ACK has now been sent.
484      */
485     if (win > 0 && SEQ_GT(tp->rcv_nxt + win, tp->rcv_adv))
486         tp->rcv_adv = tp->rcv_nxt + win;
487     tp->last_ack_sent = tp->rcv_nxt;
488     tp->t_flags &= ~(TF_ACKNOW | TF_DELACK);

489     if (sendalot)
490         goto again;
491     return (0);
492 }
```

tcp_output.c

图26-32 (续)

27. 为插口调试添加路由记录

448-452 如果选用了SO_DEBUG选项，tcp_trace会在TCP的循环路由缓存中添加一条记录，27.10节将详细讨论这个函数。

28. 设置IP长度、TTL和TOS

453-462 IP首部的3个字段必须由传输层设置：IP长度、TTL和TOS，图23-19底部用星号强调了这3个特殊字段。

注意，注释的内容为“ XXX ”，这是因为尽管对于给定连接， TTL和TOS通常是常量，可以保存在首部模板中，无需每次发送报文段时都明确赋值。只有当 TCP检验和计算完毕后，这两个字段才能填入IP首部，因此只能这样实现。

29. 向IP传递数据报

463-464 ip_output发送携带TCP报文段的数据报。TCP的插口选项和SO_DONTROUTE逻辑与，从而能向IP层传送的插口选项只有一个：SO_DONTROUTE。尽管ip_output还测试另一个选项SO_BROADCAST，但即使设定了它，与SO_DONTROUTE的逻辑与也会将其关闭。也就是说，应用进程不允许向一个广播地址发送connect，即使它设定了SO_BROADCAST选项。

467-470 如果接口队列已满，或者 IP请求分配 mbuf失败，则返回差错码 ENOBUFS。tcp_quench把拥塞窗口设定为只能容纳一个最大报文段长度，强迫连接执行慢起动。注意，出现上述情况时， TCP仍旧返回 0(OK)，而非错误，即使数据报实际已丢弃。这与udp_output(图23-20)不同，后者返回一个错误。 TCP将通过超时重传该数据报 (数据报文段)，希望那时在接口输出队列中会有可用空间或者能申请到更多的 mbuf。如果TCP报文段不包含数据，对端由于未收到ACK而引发超时时，将重传由丢失的ACK所确认的数据。

471-475 如果连接已收到一个SYN，但找不到至目的地的路由，则记录连接上出现了一个软错误。

当tcp_output被tcp_usrreq调用，做为应用进程系统调用的一部分时 (参见第30章，PRU_CONNECT、PRU_SEND、PRU_SENDOOB和PRU_SHUTDOWN请求)，应用进程将接收tcp_output的返回值。其他调用 tcp_output的函数，如tcp_input、快超时函数和慢超时函数，忽略其返回值(因为这些函数不向应用进程返回差错码)。

30. 更新rcv_adv和last_ack_sent

479-486 如果报文段中通告的最高序号(rcv_nxt加上win)大于rcv_adv，则保存新的值。回想图26-9中利用rcv_adv确定最后一个报文段发送后新增的可用空间，以及图 26-29中利用它确定TCP没有缩小窗口。

487 报文段确认字段的值保存在last_ack_sent中，tcp_input利用它处理时间戳选项(图26-6)。

488 由于所有延迟的ACK都已被发送，TF_ACKNOW和TF_DELACK标志被清除。

31. 是否还有数据需要发送

489-490 如果sendalot标志置位，控制跳回到again处(图26-1)。如果发送缓存中的数据超过一个最大长度报文段的容量 (图26-3)，或者由于加入TCP选项，降低了最大长度报文段的数据容量，无法在一个报文段中将缓存中的数据发送完毕时，控制将折回。

26.8 tcp_template函数

创建插口时，将调用tcp_newtcpcb(见前一章)为TCP控制块分配内存，并完成部分初始化。当在插口上发送或接收第一个报文段时 (主动打开，PRU_CONNECT请求，或者在监听的插口上收到了一个SYN)，tcp_template为连接的IP和TCP的首部创建一个模坂，从而减少了报文段发送时tcp_output的工作量。

图26-33给出了tcp_template函数。

tcp_subr.c

```
59 struct tcpiphdr *
60 tcp_template(tp)
61 struct tcpcb *tp;
62 {
63     struct inpcb *inp = tp->t_inpcb;
64     struct mbuf *m;
65     struct tcpiphdr *n;

66     if ((n = tp->t_template) == 0) {
67         m = m_get(M_DONTWAIT, MT_HEADER);
68         if (m == NULL)
69             return (0);
70         m->m_len = sizeof(struct tcpiphdr);
71         n = mtod(m, struct tcpiphdr *);
72     }
73     n->ti_next = n->ti_prev = 0;
74     n->ti_x1 = 0;
75     n->ti_pr = IPPROTO_TCP;
76     n->ti_len = htons(sizeof(struct tcpiphdr) - sizeof(struct ip));
77     n->ti_src = inp->inp_laddr;
78     n->ti_dst = inp->inp_faddr;
79     n->ti_sport = inp->inp_lport;
80     n->ti_dport = inp->inp_fport;
81     n->ti_seq = 0;
82     n->ti_ack = 0;
83     n->ti_x2 = 0;
84     n->ti_off = 5;              /* 5 32-bit words = 20 bytes */
85     n->ti_flags = 0;
86     n->ti_win = 0;
87     n->ti_sum = 0;
88     n->ti_urp = 0;
89     return (n);
90 }
```

tcp_subr.c

图26-33 tcp_template 函数：创建IP和TCP首部的模板

1. 分配mbuf

59-72 IP和TCP的首部模板在一个mbuf中组建，指向这个mbuf的指针存储在TCP控制块的t_template成员变量中。由于这个函数可在软件中断级被tcp_input调用，M_DONTWAIT标志置位。

2. 初始化首部字段

73-88 除下列字段外，IP和TCP首部的其他字段均置为0：ti_pr等于TCP的IP协议值(6)；ti_len等于20，TCP首部的默认值；ti_off等于5，TCP首部长度，以32 bit为单位；此外，还要从Internet PCB中把源IP地址、目的IP地址和TCP端口号复制到TCP首部模板中。

3. 用于TCP检验和计算的伪首部

73-88 由于预先对IP和TCP首部中许多字段做了初始化，简化了TCP检验和的计算，方法与23.6节中讨论过的UDP首部检验和的计算方式相同。参考图23-19中的udpiphdr结构，请读者自己思考为什么tcp_template将ti_next和ti_prev等字段初始化为0。

26.9 tcp_respond函数

函数tcp_respond尽管也调用ip_output发送IP数据报，但用途不同。主要在下面两种情况下调用它：

1) tcp_input调用它生成RST报文段，携带或不携带ACK；

2) tcp_timers调用它发送保活探测报文。

在这两种特殊情况下，TCP调用tcp_respond，取代tcp_output中复杂的逻辑。但请注意，下一章中讨论的tcp_drop函数调用tcp_output来生成RST报文段。并非所有的RST报文段都由tcp_respond生成。

图26-34给出了tcp_respond的前半部分。

tcp_subr.c

```
104 void
105 tcp_respond(tp, ti, m, ack, seq, flags)
106 struct tcpcb *tp;
107 struct tcpiphdr *ti;
108 struct mbuf *m;
109 tcp_seq ack, seq;
110 int     flags;
111 {
112     int     tlen;
113     int     win = 0;
114     struct route *ro = 0;

115     if (tp) {
116         win = sbspace(&tp->t_inpcb->inp_socket->so_rcv);
117         ro = &tp->t_inpcb->inp_route;
118     }
119     if (m == 0) {                   /* generate keepalive probe */
120         m = m_gethdr(M_DONTWAIT, MT_HEADER);
121         if (m == NULL)
122             return;
123         tlen = 0;                   /* no data is sent */
124         m->m_data += max_linkhdr;
125         *mtod(m, struct tcpiphdr *) = *ti;
```

图26-34 tcp_respond 函数：前半部分

```
126         ti = mtod(m, struct tcpiphdr *);
127         flags = TH_ACK;
128     } else {                    /* generate RST segment */
129         m_freem(m->m_next);
130         m->m_next = 0;
131         m->m_data = (caddr_t) ti;
132         m->m_len = sizeof(struct tcpiphdr);
133         tlen = 0;
134 #define xchg(a,b,type) { type t; t=a; a=b; b=t; }
135         xchg(ti->ti_dst.s_addr, ti->ti_src.s_addr, u_long);
136         xchg(ti->ti_dport, ti->ti_sport, u_short);
137 #undef xchg
138     }
```

tcp_subr.c

图26-34 (续)

104-110 图26-35列出了3种不同情况下调用tcp_respond时其参数的变化。

	参 数					
	tp	ti	m	ack	seq	flags
生成不带ACK的RST	tp	ti	m	0	ti_ack	TH_RST
生成带ACK的RST	tp	ti	m	ti_seq + ti_len	0	TH_RST|TH_ACK
生成保活探测	tp	t_template	NULL	rcv_nxt	snd_una	0

图26-35 tcp_respond 的参数

tp是指向TCP控制块的指针(可能为空)；ti是指向IP和TCP首部模板的指针；m是指向mbuf的指针，其中的报文段引发RST。最后3个参数是确认字段、序号字段和待生成报文段的标志字段。

113-118 如果tcp_input收到一个不属于任何连接的报文段，则有可能生成RST。例如，收到的报文段中没有指明任何现存连接(如，SYN指明的端口上没有正在监听的服务器)。这种情况下，tp为空，使用win和ro的初始值。如果tp不空，则通告窗口大小将等于接收缓存中的可用空间，指向缓存路由的指针保存在ro中，在后面调用tcp_input时会用到。

1. 保活定时器超时后发送保活探测

119-127 参数m是指向接收报文段的mbuf链表的指针。但保活探测报文只有当保活定时器超时时才会被发送，收到的TCP报文段不可能引发此项操作，因此m为空，由m_gethdr分配保存IP和TCP首部的mbuf。TCP数据长度tlen，设为0，因为保活探测报文不包含任何用户数据。

有些基于4.2BSD的较老的系统不响应保活探测报文，除非它携带数据。通过配置，在编译内核时定义TCP_COMPAT_42，Net/3能够在保活探测报文中携带一个字节的无效数据，以引出这些系统的响应。这种情况下，tlen设为1，而非0。无效字节不会造成不良后果，因为它不是对方正等待(而是一个对方已接收并确认过)的字节，对端将丢弃它。

利用赋值语句把ti指向的TCP首部模板结构复制到mbuf的数据部分，之后指针ti将被重

新设定，指向mbuf中的首部模板。

2. 发送RST报文段

128-138 接收到的报文段有可能会引发tcp_input发送RST。发送RST时，保存输入报文段的mbuf可以重用。因为tcp_respond生成的报文段中只包含IP首部和TCP首部，因此，除第一个mbuf之外(数据分组首部)，m_free将释放链表中其余的所有mbuf。另外，IP首部和TCP首部中的源IP地址和目的IP地址及端口号应互换。

图26-36给出了tcp_respond的后半部分。

tcp_subr.c

```
139     ti->ti_len = htons((u_short) (sizeof(struct tcphdr) + tlen));
140     tlen += sizeof(struct tcpiphdr);
141     m->m_len = tlen;
142     m->m_pkthdr.len = tlen;
143     m->m_pkthdr.rcvif = (struct ifnet *) 0;
144     ti->ti_next = ti->ti_prev = 0;
145     ti->ti_x1 = 0;
146     ti->ti_seq = htonl(seq);
147     ti->ti_ack = htonl(ack);
148     ti->ti_x2 = 0;
149     ti->ti_off = sizeof(struct tcphdr) >> 2;
150     ti->ti_flags = flags;
151     if (tp)
152         ti->ti_win = htons((u_short) (win >> tp->rcv_scale));
153     else
154         ti->ti_win = htons((u_short) win);
155     ti->ti_urp = 0;
156     ti->ti_sum = 0;
157     ti->ti_sum = in_cksum(m, tlen);
158     ((struct ip *) ti)->ip_len = tlen;
159     ((struct ip *) ti)->ip_ttl = ip_defttl;
160     (void) ip_output(m, NULL, ro, 0, NULL);
161 }
```

tcp_subr.c

图26-36 tcp_respond 函数：后半部分

139-157 为计算TCP检验和，IP和TCP首部字段必须被初始化。这些语句与tcp_template初始化t_template字段的方式类似。序号和确认字段由调用者提供，最后调用ip_output发送数据分组。

26.10 小结

本章讨论了生成大多数TCP报文段的通用函数(tcp_output)及生成RST报文段和保活探测的特殊函数(tcp_respond)。

TCP是否发送报文段取决于许多因素：报文段中的标志、对端通告的窗口大小、待发送的数据量以及连接上是否存在未确认的数据等等。因此，tcp_output中的逻辑决定了是否发送报文段(函数的前半部分)，如果需要发送，如何填充TCP首部的字段(函数后半部分)。报文段发送之后，还需要更新TCP控制块中的相应变量。

tcp_output一次只生成一个报文段，但它在结尾处会测试是否还有剩余数据等待发送，如果有，控制将折回，并试图发送下一个报文段。这样的循环会一直持续到数据全部发送完毕，或者有其他停止传输的条件出现(接收方的窗口通告)。

TCP报文段中可以携带选项。Net/3支持的选项规定了最大报文段长度、窗口大小缩放因子和一对时间戳。头两个选项只能出现在SYN报文段中，而时间戳选项(如果连接双方都支持)能够出现在所有报文段中。因为窗口大小和时间戳是新增的选项，如果主动打开的一端希望使用这些选项，则必须在自己发送的SYN中添加它们，并且只有在对端发回的SYN也包含了同样的选项时才能使用。

习题

26.1 图26-1中，如果发送数据过程中出现停顿，TCP将返回慢启动状态，而空闲时间被设定为从最后一次收到报文段到现在的时间。为什么TCP不将空闲时间设定为从最后一次发送报文到现在的时间？

26.2 图26-6中，我们说如果FIN已发送，但还未被确认且没有重传，此时len小于0。如果FIN已重传，情况会怎样？

26.3 Net/3总在主动打开时发送窗口大小和时间戳选项。为什么需要全局变量tcp_do_rfc1323？

26.4 图25-28中的例子未使用时间戳，RTT估算值被更新了8次。如果使用了时间戳，RTT估算值会被更新几次？

26.5 图26-23中，调用bcopy把收到的MSS存储在变量mss中。为什么不对指向opt[2]的指针做强制转换，变为不带符号的短整型指针，并利用赋值语句完成这一操作？

26.6 在图26-29后面，我们讨论了代码的一个错误，可能会导致发送一个错误的紧急数据偏移量。提出你的解决方案。(提示：一个TCP报文中能够发送的最大数据量是多少？)

26.7 图26-32中，我们提到不会向应用进程返回差错代码ENOBUFS，因为(1)如果丢弃的是数据报文，重传定时器超时后数据将被重传；(2)如果丢弃的是纯ACK报文，对端收不到ACK时会重传对应的数据报文。如果丢弃的是RST报文，情况会怎样？

26.8 解释卷1图20-3中PSH标志的设定。

26.9 为什么图26-36使用ip_defttl作为TTL的值，而图26-32却使用PCB？

26.10 如果应用进程规定的IP选项是用于TCP连接的，图26-25中分配的mbuf会出现什么情况？实现一个更好的方案。

26.11 tcp_output函数很长(包括注释约500行)，看上去效率不高，其中许多代码用于处理特殊情况。假定函数只用于处理准备好的最大长度报文，且没有特殊情况：无IP选项，无特殊标志如SYN、FIN或URG。实际执行的约有多少行C代码？报文递交给ip_output之前会调用多少函数？

26.12 26.3节结尾的例子中，应用程序向连接写入100字节，接着又写入50字节。如果应用程序为两个缓存各调用一次writev，而不是调用write两次，有何不同？如果两个缓存大小分别为200和300，而不是100和50，调用writev时又有何不同？

26.13 在时间戳选项中发送的时间戳来自于全局变量tcp_now，它每500ms递增一次。修改TCP代码，使用更精确的时间戳值。

第27章 TCP的函数

27.1 引言

本章介绍多个TCP函数，它们为下两章进一步讨论TCP的输入打下了基础：

- `tcp_drain`是协议的资源耗尽处理函数，当内核的 mbuf用完时被调用。实际上，不做任何处理。
- `tcp_drop`发送RST来丢弃连接。
- `tcp_close`执行正常的TCP连接关闭操作：发送 FIN，并等待协议要求的 4次报文交换以终止连接。卷1的18.2节讨论了连接关闭时双方需要交换的 4个报文。
- `tcp_mss`处理收到的MSS选项，并在TCP发送自己的MSS选项时计算应填入的MSS值。
- `tcp_ctlinput`在收到对应于某个 TCP报文段的ICMP差错时被调用，它接着调用`tcp_notify`处理ICMP差错。`tcp_quench`专门负责处理ICMP的源站抑制差错。
- `TCP_REASS`宏和`tcp_reass`函数管理连接重组队列中的报文段。重组队列处理收到的乱序报文段，某些报文段还可能互相重复。
- `tcp_trace`向内核的TCP调试循环缓存中添加记录 (插口选项`SO_DEBUG`)。运行`trpt` (8)程序可以打印缓存内容。

27.2 tcp_drain函数

`tcp_drain`是所有TCP函数中最简单的。它是协议的`pr_drain`函数，在内核的mbuf用完时，由`m_reclaim`调用。图10-32中，`ip_drain`丢弃其重组队列中的所有数据报分片，而UDP则不定义自己的资源耗尽处理函数。尽管 TCP也占用mbuf —— 位于接收窗口内的乱序报文段 —— 但Net/3实现的 TCP并不丢弃这些 mbuf，即使内核的 mbuf已用完。相反，`tcp_drain`不做任何处理，假定收到的(但次序差错)的TCP报文段比IP分片重要。

27.3 tcp_drop函数

`tcp_drop`在整个系统中多次被调用，发送RST报文段以丢弃连接，并向应用进程返回差错。它与关闭连接(`tcp_disconnect`函数)不同，后者向对端发送 FIN，并遵守TCP状态变迁图所规定的连接终止步骤。

图27-1列出了调用`tcp_drop`的7种情况和相应的`errno`参数。

图27-2给出了`tcp_drop`函数。

202-213 如果TCP收到了一个SYN，连接被同步，则必须向对端发送RST。`tcp_drop`把状态设为CLOSED，并调用`tcp_output`。从图24-16可知，CLOSED状态的`tcp_outflags`数组中包含RST标志。

214-216 如果`errno`等于`ETIMEDOUT`，且连接上曾收到过软差错 (如`EHOSTUNREACH`)，

软差错代码将取代内容不确定的ETIMEDOUT，做为返回的插口差错。

217 tcp_close结束插口关闭操作。

函　数	errno	描　述
tcp_input	ENOBUFS	监听服务器收到SYN，但内核无法为t_template分配所需的mbuf
tcp_input	ECONNREFUSED	收到的RST是对本地发送的SYN的响应
tcp_input	ECONNRESET	在现存连接上收到了RST
tcp_timers	ETIMEDOUT	重传定时器连续超时13次，仍未收到对端的ACK(图25-25)
tcp_timers	ETIMEDOUT	连接建立定时器超时(图25-16)，或者保活定时器超时，且连续9次发送窗口探测报文段，对方均无响应
tcp_usrreq	ECONNABORTED	PRU_ABORT请求
tcp_usrreq	0	关闭插口，设定SO_LINGER选项，且拖延时间为0

图27-1　调用tcp_drop　函数和errno 参数

tcp_subr.c

```
202 struct tcpcb *
203 tcp_drop(tp, errno)
204 struct tcpcb *tp;
205 int     errno;
206 {
207     struct socket *so = tp->t_inpcb->inp_socket;

208     if (TCPS_HAVERCVDSYN(tp->t_state)) {
209         tp->t_state = TCPS_CLOSED;
210         (void) tcp_output(tp);
211         tcpstat.tcps_drops++;
212     } else
213         tcpstat.tcps_conndrops++;
214     if (errno == ETIMEDOUT && tp->t_softerror)
215         errno = tp->t_softerror;
216     so->so_error = errno;
217     return (tcp_close(tp));
218 }
```

tcp_subr.c

图27-2　tcp_drop　函数

27.4　tcp_close函数

通常情况下，如果应用进程被动关闭，且在LAST_ACK状态时收到了ACK，tcp_input将调用tcp_close关闭连接；或者当2MSL定时器超时，插口从TIME_WAIT状态变迁到CLOSED状态时，tcp_timers也会调用tcp_close。它也可以在其他状态被调用，一种可能是发生了差错，如上一小节讨论过的情况。tcp_close释放连接占用的内存(IP和TCP首部模板、TCP控制块、Internet PCB和保存在连接重组队列中的所有乱序报文段)，并更新路由特性。

我们分3部分讲解这个函数，前两部分讨论路由特性，最后一部分介绍资源释放。

27.4.1　路由特性

rt_metrics结构(图18-26)中保存了9个变量，有6个用于TCP。其中8个变量可通过

route (8)命令读写(第9个，rmx_pksent未使用)：图27-3列出了这些变量。此外，运行route命令时，加入-lock选项，可以设置rmx_locks成员变量(图20-13)中对应的RTV_*xxx*比特，告诉内核不要更新对应的路由参数。

关闭TCP 插口时，如果下列条件满足：连接上传输的数据量足够生成有效的统计值，并且变量未被锁定，tcp_close将更新3个路由参数 —— 已平滑的RTT估计器、已平滑的RTT平均偏差估计器和慢起动门限。

rt_metrics成员	tcp_close是否保存该成员	tcp_mss是否使用该成员	route(8)附加参数
rmx_expire			-expire
rmx_hopcount			-hopcount
rmx_mtu		•	-mtu
rmx_recvpipe		•	-recvpipe
rmx_rtt	•	•	-rtt
rmx_rttvar	•	•	-rttvar
rmx_sendpipe		•	-sendpipe
rmx_ssthresh	•	•	-ssthresh

图27-3 TCP用到的rt_metrics 结构中的变量

图27-4给出了tcp_close的第一部分。

1. 判断是否发送了足够的数据量

234-248 默认的发送缓存大小为8192字节(sb_hiwat)，因此首先比较初始发送序号和连接上已发送的最大序号，测试是否已传输了 131 072字节(16个完整的缓存)的数据。此外，插口还必须有一条非默认路由的缓存路由(参见习题19.2)。

请注意，如果传输的数据量在 $N \times 2^{32}$(N>1)和$N \times 2^{32}$+131072(N>1)之间，则因为序号可能回绕，比较时也许会出现问题，尽管可能性不大。但目前很少有连接会传输4 G的数据。

尽管Internet上存在大量的默认路由，缓存路由对于维护有效的路由表还是很有用的。如果主机长期与另外某个主机(或网络)交换数据，即使默认路由可用，也应运行route命令向路由表中添加源站选路和目的选路的路由，从而在整条连接上维护有效的路由信息(参见习题19.2)。这些信息在系统重启时丢失。

250 管理员可以锁定图27-3中的变量，防止内核修改它们。因此，代码在更新这些变量之前，必须先检查其锁定状态。

2. 更新RTT

251-264 t_srtt的单位为8个滴答(图25-19)，而rmx_rtt的单位为微秒。因此，首先必须实现单位换算，t_srtt乘以1 000 000(RTM_RTTUNIT)，除以2(滴答/秒)再乘以8，得到RTT的最新值。如果rmx_rtt值已存在，它被更新为最新值与原有值和的一半，即两者的平均值。如果不存在，最新值将直接赋给rmx_rtt变量。

3. 更新平均偏差

265-273 更新平均偏差的算法与更新RTT的类似，也需要把单位为4个滴答的t_rttvar换算为以微秒为单位。

tcp_subr.c

```
225 struct tcpcb *
226 tcp_close(tp)
227 struct tcpcb *tp;
228 {
229     struct tcpiphdr *t;
230     struct inpcb *inp = tp->t_inpcb;
231     struct socket *so = inp->inp_socket;
232     struct mbuf *m;
233     struct rtentry *rt;
234     /*
235      * If we sent enough data to get some meaningful characteristics,
236      * save them in the routing entry.  'Enough' is arbitrarily
237      * defined as the sendpipesize (default 8K) * 16.  This would
238      * give us 16 rtt samples assuming we only get one sample per
239      * window (the usual case on a long haul net).  16 samples is
240      * enough for the srtt filter to converge to within 5% of the correct
241      * value; fewer samples and we could save a very bogus rtt.
242      *
243      * Don't update the default route's characteristics and don't
244      * update anything that the user "locked".
245      */
246     if (SEQ_LT(tp->iss + so->so_snd.sb_hiwat * 16, tp->snd_max) &&
247         (rt = inp->inp_route.ro_rt) &&
248      ((struct sockaddr_in *) rt_key(rt))->sin_addr.s_addr != INADDR_ANY) {
249         u_long  i;
250         if ((rt->rt_rmx.rmx_locks & RTV_RTT) == 0) {
251             i = tp->t_srtt *
252                 (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));
253             if (rt->rt_rmx.rmx_rtt && i)
254                 /*
255                  * filter this update to half the old & half
256                  * the new values, converting scale.
257                  * See route.h and tcp_var.h for a
258                  * description of the scaling constants.
259                  */
260                 rt->rt_rmx.rmx_rtt =
261                     (rt->rt_rmx.rmx_rtt + i) / 2;
262             else
263                 rt->rt_rmx.rmx_rtt = i;
264         }
265         if ((rt->rt_rmx.rmx_locks & RTV_RTTVAR) == 0) {
266             i = tp->t_rttvar *
267                 (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
268             if (rt->rt_rmx.rmx_rttvar && i)
269                 rt->rt_rmx.rmx_rttvar =
270                     (rt->rt_rmx.rmx_rttvar + i) / 2;
271             else
272                 rt->rt_rmx.rmx_rttvar = i;
273         }
```

tcp_subr.c

图27-4 tcp_close 函数：更新RTT和平均偏差

图27-5给出了tcp_close的下一部分代码，更新路由的慢起动门限。

274-283 满足下列条件时，慢起动门限被更新：(1)它被更新过(rmx_ssthresh非零)；(2)管理员规定了rmx_sendpipe，而snd_ssthresh的最新值小于rmx_sendpipe的一半。如同代码注释中指出的，TCP不会更新rmx_ssthresh值，除非因为数据分组丢失而不得不

这样做。从这个角度出发，除非十分必要，TCP不会修改门限值。

tcp_subr.c

```
274         /*
275          * update the pipelimit (ssthresh) if it has been updated
276          * already or if a pipesize was specified & the threshhold
277          * got below half the pipesize.  I.e., wait for bad news
278          * before we start updating, then update on both good
279          * and bad news.
280          */
281         if ((rt->rt_rmx.rmx_locks & RTV_SSTHRESH) == 0 &&
282             (i = tp->snd_ssthresh) && rt->rt_rmx.rmx_ssthresh ||
283             i < (rt->rt_rmx.rmx_sendpipe / 2)) {
284             /*
285              * convert the limit from user data bytes to
286              * packets then to packet data bytes.
287              */
288             i = (i + tp->t_maxseg / 2) / tp->t_maxseg;
289             if (i < 2)
290                 i = 2;
291             i *= (u_long) (tp->t_maxseg + sizeof(struct tcpiphdr));
292             if (rt->rt_rmx.rmx_ssthresh)
293                 rt->rt_rmx.rmx_ssthresh =
294                     (rt->rt_rmx.rmx_ssthresh + i) / 2;
295             else
296                 rt->rt_rmx.rmx_ssthresh = i;
297         }
298     }
```

tcp_subr.c

图27-5 tcp_close 函数：更新慢起动门限

284-290 变量snd_ssthresh以字节为单位，除以MSS(t_maxseg)得到报文段数，加上1/2t_maxseg是为了保证总报文段容量必定大于 snd_ssthresh字节。报文段数的下限为2个报文段。

291-297 MSS加上IP和TCP首部大小(40)，再乘以报文段数，利用得到的结果来更新rmx_ssthresh，采用的算法与图27-4中的相同(新值的1/2加上原有值的1/2)。

27.4.2 资源释放

图27-6给出了tcp_close的最后一部分，释放插口占用的内存资源。

tcp_subr.c

```
299     /* free the reassembly queue, if any */
300     t = tp->seg_next;
301     while (t != (struct tcpiphdr *) tp) {
302         t = (struct tcpiphdr *) t->ti_next;
303         m = REASS_MBUF((struct tcpiphdr *) t->ti_prev);
304         remque(t->ti_prev);
305         m_freem(m);
306     }
307     if (tp->t_template)
308         (void) m_free(dtom(tp->t_template));
309     free(tp, M_PCB);
310     inp->inp_ppcb = 0;
```

图27-6 tcp_close 函数：释放连接资源

```
311     soisdisconnected(so);
312     /* clobber input pcb cache if we're closing the cached connection */
313     if (inp == tcp_last_inpcb)
314         tcp_last_inpcb = &tcb;
315     in_pcbdetach(inp);
316     tcpstat.tcps_closed++;
317     return ((struct tcpcb *) 0);
318 }
```

tcp_subr.c

图27-6 (续)

1. 释放重组队列占用的mbuf

299-306 如果连接重组队列中还有报文段，则丢弃它们。重组队列用于存放收到位于接收窗口内、但次序差错的报文段。在等待接收的正常序列报文段到达之前，它们会一直保存在重组队列中；之后，报文段被重组并递交给应用程序。27.9节会详细讨论这一过程。

2. 释放首部模板和TCP控制块

307-309 调用m_free释放IP和TCP首部模板，调用free释放TCP控制块，调用sodisconnected发送PRU_DISCONNECT请求，标记插口已断开连接。

3. 释放PCB

310-318 如果插口的Internet PCB保存在TCP的高速缓存中，则把TCP的PCB链表表头赋给tcp_last_inpcb，以清空缓存。接着调用in_pcbdetach释放PCB占用的内存。

27.5 tcp_mss函数

tcp_mss被两个函数调用：

1) tcp_output，准备发送SYN时调用，以添加MSS选项；

2) tcp_input，收到的SYN报文段中包含MSS选项时调用；

tcp_mss函数检查到达目的地的缓存路由，计算用于该连接的MSS。

图27-7给出了tcp_mss第一部分的代码，如果PCB中没有到达目的地的路由，则设法得到所需的路由。

tcp_input.c

```
1391 int
1392 tcp_mss(tp, offer)
1393 struct tcpcb *tp;
1394 u_int   offer;
1395 {
1396     struct route *ro;
1397     struct rtentry *rt;
1398     struct ifnet *ifp;
1399     int     rtt, mss;
1400     u_long  bufsize;
1401     struct inpcb *inp;
1402     struct socket *so;
1403     extern int tcp_mssdflt;

1404     inp = tp->t_inpcb;
1405     ro = &inp->inp_route;

1406     if ((rt = ro->ro_rt) == (struct rtentry *) 0) {
```

图27-7 tcp_mss 函数：如果PCB中没有路由，则设法得到所需路由

```
1407            /* No route yet, so try to acquire one */
1408            if (inp->inp_faddr.s_addr != INADDR_ANY) {
1409                ro->ro_dst.sa_family = AF_INET;
1410                ro->ro_dst.sa_len = sizeof(ro->ro_dst);
1411                ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
1412                    inp->inp_faddr;
1413                rtalloc(ro);
1414            }
1415            if ((rt = ro->ro_rt) == (struct rtentry *) 0)
1416                return (tcp_mssdflt);
1417        }
1418        ifp = rt->rt_ifp;
1419        so = inp->inp_socket;
```

tcp_input.c

图27-7 (续)

1. 如果需要，就获取路由

1391-1417 如果插口没有高速缓存路由，则调用rtalloc得到一条。与外出路由相关的接口指针存储在ifp中。外出接口是非常重要的，因为其MTU会影响TCP通告的MSS。如果无法得到所需路由，函数就立即返回默认值512(tcp_mssdflt)。

图27-8给出了tcp_mss的下一部分代码，判断得到的路由是否有相应的参数表。如果有，则变量t_rttmin、t_srtt和t_rttvar将初始化为参数表中的对应值。

tcp_input.c

```
1420        /*
1421         * While we're here, check if there's an initial rtt
1422         * or rttvar.  Convert from the route-table units
1423         * to scaled multiples of the slow timeout timer.
1424         */
1425        if (tp->t_srtt == 0 && (rtt = rt->rt_rmx.rmx_rtt)) {
1426            /*
1427             * XXX the lock bit for RTT indicates that the value
1428             * is also a minimum value; this is subject to time.
1429             */
1430            if (rt->rt_rmx.rmx_locks & RTV_RTT)
1431                tp->t_rttmin = rtt / (RTM_RTTUNIT / PR_SLOWHZ);
1432            tp->t_srtt = rtt / (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));

1433            if (rt->rt_rmx.rmx_rttvar)
1434                tp->t_rttvar = rt->rt_rmx.rmx_rttvar /
1435                    (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
1436            else
1437                /* default variation is +- 1 rtt */
1438                tp->t_rttvar =
1439                    tp->t_srtt * TCP_RTTVAR_SCALE / TCP_RTT_SCALE;

1440            TCPT_RANGESET(tp->t_rxtcur,
1441                          ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1,
1442                          tp->t_rttmin, TCPTV_REXMTMAX);
1443        }
```

tcp_input.c

图27-8 tcp_mss 函数：判断路由是否有相应的RTT参数表

2. 初始化已平滑的RTT估计器

1420-1432 如果连接上不存在RTT样本值(t_srtt=0)，并且rmx_rtt非零，则将后者赋

给已平滑的RTT估计器t_srtt。如果路由参数表锁定标志的RTV_RTT比特置位，表明连接的最小RTT(t_rttmin)也应初始化为rmx_rtt。前面介绍过，tcp_newtcpcb把t_rttmin初始化为2个滴答。

rmx_rtt(以微秒为单位)转换为t_srtt(以8个滴答为单位)，这是图27-4的反变换。注意，t_rttmin等于t_srtt的1/8，因为前者没有除以缩放因子TCP_RTT_SCALE。

3. 初始化已平滑的RTT平均偏差估计器

1433-1439 如果存储的rmx_rttvar(以微秒为单位)值非零，将其转换为t_rttvar(以4个滴答为单位)。但如果为零，则t_rttvar等于t_rtt，即偏差等于均值。已平滑的RTT平均偏差估计器默认设置为±1 RTT。由于t_rttvar的单位为4个滴答，而t_rtt的单位为8个滴答，t_srtt值也必须做相应转换。

4. 计算初始RTO

1440-1442 计算当前的*RTO*，并存储在t_rxtcur中，采用下列算式更新：

$$RTO = srtt + 2 \times rttvar$$

计算第一个*RTO*时，乘数取2，而非4，上式与图25-21中用到的算式相同。将缩放关系代入，得到：

$$RTO = \frac{t\_srtt}{8} + 2 \times \frac{t\_rttvar}{4} = \frac{\frac{t\_srtt}{4} + t\_rttvar}{2}$$

即为TCPT_RANGESET的第二个参数。

图27-9给出了tcp_mss的下一部分，计算MSS。

tcp_input.c

```
1444     /*
1445      * if there's an mtu associated with the route, use it
1446      */
1447     if (rt->rt_rmx.rmx_mtu)
1448         mss = rt->rt_rmx.rmx_mtu - sizeof(struct tcpiphdr);
1449     else {
1450         mss = ifp->if_mtu - sizeof(struct tcpiphdr);
1451 #if (MCLBYTES & (MCLBYTES - 1)) == 0
1452         if (mss > MCLBYTES)
1453             mss &= ~(MCLBYTES - 1);
1454 #else
1455         if (mss > MCLBYTES)
1456             mss = mss / MCLBYTES * MCLBYTES;
1457 #endif
1458         if (!in_localaddr(inp->inp_faddr))
1459             mss = min(mss, tcp_mssdflt);
1460     }
```

tcp_input.c

图27-9 tcp_mss 函数：计算mss

5. 从路由表中的MTU得到MSS

1444-1450 如果路由表中的MTU有值，则将其赋给mss。如果没有，则mss初始值等于外出接口的MTU值减去40(IP和TCP首部默认值)。对于以太网，MSS初始值应为1460。

6. 减小MSS，令其等于MCLBYTES的倍数

1451-1457 如果mss大于MCLBYTES，则减小mss的值，令其等于最接近的

MCLBYTES(mbuf簇大小)的整数倍。如果MCLBYTES值(通常等于1024或2048)与MCLBYTES值减1逻辑与后等于0，说明MCLBYTES等于2的倍数。例如，1024(0x400)逻辑与1023(0x3ff)等于0。

代码通过清零mss的若干低位比特，将mss减小到最接近的MCLBYTES的倍数：如果mbuf簇大小为1024，mss与1023的二进制补码(0xfffffc00)逻辑与，低位的10 bit被清零。对于以太网，mss将从1460减至1024。如果mbuf簇大小为2048，与2047的二进制补码(0xffff8000)逻辑与，低位的11 bit被清零。对于令牌环，MTU大小为4464，上述运算将mss从4424减为4096。如果MCLBYTES不是2的倍数，代码用mss整数除以MCLBYTES后，再乘上MCLBYTES，从而将mss减小到最接近的MCLBYTES的倍数。

7. 判断目的地是本地地址还是远端地址

1458-1459 如果目的IP不是本地地址(in_localaddr返回零)，且mss大于512(tcp_mssdflt)，则将mss设为512。

> IP地址是否为本地地址取决于全局变量subnetsarelocal，内核编译时把符号变量SUBNETSARELOCAL的值赋给它。默认值为1，意味着如果给定IP地址与主机任一接口的IP地址具有相同的网络ID，则被认为是一个本地地址。如果为0，则给定IP地址必须与主机任一接口的IP地址具有相同的网络号和子网号，才会被认为是一个本地地址。
>
> 对于非本地地址，将MSS最小化是为了避免IP数据报经广域网时被分片。绝大多数WAN链路的MTU只有1006，这是从ARPANET遗留下来的一个问题。在卷1的11.7节中讨论过，现代的多数WAN支持1500，甚至更大的MTU。感兴趣的读者还可阅读卷1的24.2节中讨论的路由MTU发现特性(RFC 1191，[Mogul and Deering 1990])。Net/3不支持路由MTU发现。

图27-10给出了tcp_mss最后一部分的代码。

8. 对端的MSS用作上限

1461-1472 如果tcp_mss被tcp_input调用，参数offer非零，等于对端通告的mss值。如果mss大于对端通告的值，则将offer赋给它。例如，如果函数计算得到的mss等于1024，但对端通告的值只有512，则mss必须被设定为512。相反，如果mss等于536(即输出MTU等于576)，而对端通告的值为1460，TCP仍旧使用536。只要不超过对端通告的值，mss可以取小于它的任何一个值。如果tcp_mss被tcp_output调用，offer等于0，用于发送MSS选项。注意，尽管mss的上限可变，其下限固定为32。

1473-1483 如果mss小于tcp_newtcpcb中设定的默认值t_maxseg(512)，或者如果TCP正在处理收到的MSS选项(offer非零)，则需执行下列步骤。首先，如果路由的rmx_sendpipe有值，则采用它做为发送缓存的高端(high-water)标志(图16-4)。如果缓存小于mss，则使用较小的值。除非是应用程序有意把发送缓存定得很小，或者管理员将rmx_sendpipe定得很小，这种情况一般不会发生，因为发送缓存的上限默认值为8192，大于绝大多数的mss。

9. 增加缓存大小，令其等于最近的MSS整数倍

1484-1489 增加缓存大小，令其等于最近的mss整数倍，上限为sb_max(Net/3中定义为262 144，即256×1024)。插口发送缓存的上限设定为sbreserve。例如，上限默认值等于

8192，但对于以太网上的本地TCP传输，其mbuf簇大小为2048(假定mss等于1460)，代码把上限值增加到8760(等于6×1460)。但对于非本地的连接，mss等于512，上限值保持8192不变。

tcp_input.c

```
1461     /*
1462      * The current mss, t_maxseg, was initialized to the default value
1463      * of 512 (tcp_mssdflt) by tcp_newtcpcb().
1464      * If we compute a smaller value, reduce the current mss.
1465      * If we compute a larger value, return it for use in sending
1466      * a max seg size option, but don't store it for use
1467      * unless we received an offer at least that large from peer.
1468      * However, do not accept offers under 32 bytes.
1469      */
1470     if (offer)
1471         mss = min(mss, offer);
1472     mss = max(mss, 32);          /* sanity */
1473     if (mss < tp->t_maxseg || offer != 0) {
1474         /*
1475          * If there's a pipesize, change the socket buffer
1476          * to that size.  Make the socket buffers an integral
1477          * number of mss units; if the mss is larger than
1478          * the socket buffer, decrease the mss.
1479          */
1480         if ((bufsize = rt->rt_rmx.rmx_sendpipe) == 0)
1481             bufsize = so->so_snd.sb_hiwat;
1482         if (bufsize < mss)
1483             mss = bufsize;
1484         else {
1485             bufsize = roundup(bufsize, mss);
1486             if (bufsize > sb_max)
1487                 bufsize = sb_max;
1488             (void) sbreserve(&so->so_snd, bufsize);
1489         }
1490         tp->t_maxseg = mss;
1491         if ((bufsize = rt->rt_rmx.rmx_recvpipe) == 0)
1492             bufsize = so->so_rcv.sb_hiwat;
1493         if (bufsize > mss) {
1494             bufsize = roundup(bufsize, mss);
1495             if (bufsize > sb_max)
1496                 bufsize = sb_max;
1497             (void) sbreserve(&so->so_rcv, bufsize);
1498         }
1499     }
1500     tp->snd_cwnd = mss;
1501     if (rt->rt_rmx.rmx_ssthresh) {
1502         /*
1503          * There's some sort of gateway or interface
1504          * buffer limit on the path.  Use this to set
1505          * the slow start threshhold, but set the
1506          * threshold to no less than 2*mss.
1507          */
1508         tp->snd_ssthresh = max(2 * mss, rt->rt_rmx.rmx_ssthresh);
1509     }
1510     return (mss);
1511 }
```

tcp_input.c

图27-10 tcp_mss 函数：结束处理

1490 由于t_maxseg已小于默认值(512)，或者由于收到了对端发送的MSS选项，所以应更

新它。

1491-1499 对接收缓存的处理与发送缓存相同。

10. 初始化拥塞窗口和慢起动门限

1500-1509 拥塞窗口的值，snd_cwnd，等于一个最大报文段长度。如果路由表中的rmx_ssthresh非零，慢起动门限(snd_ssthresh)初始化为该值，但应保证其下限为两个最大报文段长度。

1510 函数最后返回mss。tcp_input忽略这一返回值(图28-10，因为它已收到对端的MSS选项)，但图26-23中，tcp_output将它用作MSS通告。

举例

下面通过一个连接建立的实例说明tcp_mss的操作过程。连接建立过程中，它会被调用两次：发送SYN时和收到对端带有MSS选项的SYN时。

1) 创建插口，tcp_newtcpcb初始化t_maxseg为512。

2) 应用进程调用connect。为了在SYN报文段中加入MSS选项，tcp_output调用tcp_mss，参数offer等于零。假定目的IP为本地以太网地址，mbuf簇大小为2048，执行图27-9中的代码后，mss等于1460。由于offer等于零，图27-10中的代码不修改mss值，函数返回1460。因为1460大于默认值(512)而且未收到对端的MSS选项，缓存大小不变。tcp_output发送MSS选项，通告MSS大小为1460。

3) 对端发送响应SYN，通告mss大小为1024。tcp_input调用tcp_mss，参数offer等于1024。图27-9的代码逻辑仍旧设定mss为1460，但在图27-10起始处的min语句将mss减小为1024。因为offer非零，缓存大小增加至最近的1024的整数倍(等于8192)。t_maxseg更新为1024。

> 初看上去，tcp_mss的逻辑存在问题：TCP向对端通告mss大小为1460，之后从对端收到的mss只有1024。尽管TCP只能发送1024字节的报文段，对端却能够发送1460字节的报文段。读者可能会认为发送缓存应等于1024的倍数，而接收缓存则应等于1460的倍数。但图27-10中的代码却将两个缓存大小都设为对端通告的mss的倍数。这是因为尽管TCP通告mss为1460，但对端通告的mss仅为1024，对端有可能不会发送1460字节的报文段，而将发送报文段限制为1024字节。

27.6 tcp_ctlinput函数

回想图22-32中，tcp_ctlinput处理5种类型的ICMP差错：目的地不可达、数据报参数错、源站抑制、数据报超时和重定向。所有重定向差错会上交给相应的TCP或UDP进行处理。

对于其他4种差错，仅当它们是被TCP报文段引发的，才会调用tcp_ctlinput进行处理。

图27-11给出了tcp_ctlinput函数，它与图23-30的udp_ctlinput函数类似。

365-366 在逻辑上，tcp_ctlinput与udp_ctlinput的唯一区别是如何处理ICMP源站抑制差错。因为inetctlerrmap等于0，UDP忽略源站抑制差错。TCP检查源站抑制差错，并把notify函数的默认值tcp_notify改为tcp_quench。

tcp_subr.c

```
355 void
356 tcp_ctlinput(cmd, sa, ip)
357 int     cmd;
358 struct sockaddr *sa;
359 struct ip *ip;
360 {
361     struct tcphdr *th;
362     extern struct in_addr zeroin_addr;
363     extern u_char inetctlerrmap[];
364     void    (*notify) (struct inpcb *, int) = tcp_notify;

365     if (cmd == PRC_QUENCH)
366         notify = tcp_quench;
367     else if (!PRC_IS_REDIRECT(cmd) &&
368              ((unsigned) cmd > PRC_NCMDS || inetctlerrmap[cmd] == 0))
369         return;
370     if (ip) {
371         th = (struct tcphdr *) ((caddr_t) ip + (ip->ip_hl << 2));
372         in_pcbnotify(&tcb, sa, th->th_dport, ip->ip_src, th->th_sport,
373                      cmd, notify);
374     } else
375         in_pcbnotify(&tcb, sa, 0, zeroin_addr, 0, cmd, notify);
376 }
```

tcp_subr.c

图27-11 tcp_ctlinput 函数

27.7 tcp_notify函数

tcp_notify被tcp_ctlinput调用，处理目的地不可达、数据报参数错、数据报超时和重定向差错。与UDP的差错处理函数相比，它要复杂得多，因为TCP必须灵活地处理连接上收到的各种软差错。图27-12给出了tcp_motify函数。

tcp_subr.c

```
328 void
329 tcp_notify(inp, error)
330 struct inpcb *inp;
331 int     error;
332 {
333     struct tcpcb *tp = (struct tcpcb *) inp->inp_ppcb;
334     struct socket *so = inp->inp_socket;

335     /*
336      * Ignore some errors if we are hooked up.
337      * If connection hasn't completed, has retransmitted several times,
338      * and receives a second error, give up now.  This is better
339      * than waiting a long time to establish a connection that
340      * can never complete.
341      */
342     if (tp->t_state == TCPS_ESTABLISHED &&
343         (error == EHOSTUNREACH || error == ENETUNREACH ||
344          error == EHOSTDOWN)) {
345         return;
346     } else if (tp->t_state < TCPS_ESTABLISHED && tp->t_rxtshift > 3 &&
347                tp->t_softerror)
348         so->so_error = error;
```

图27-12 tcp_notify 函数

```
349     else
350         tp->t_softerror = error;
351     wakeup((caddr_t) & so->so_timeo);
352     sorwakeup(so);
353     sowwakeup(so);
354 }
```
tcp_subr.c

图27-12 (续)

328-345 如果连接状态为ESTABLISHED，则忽略EHOSTUNREACH、ENETUNREACH和EHOSTDOWN差错代码。

处理这3个差错是4.4BSD中新增的功能。Net/2及早期版本在连接的软差错变量(t_softerror)中记录这些差错，如果连接最终失败，则向应用进程返回相应的差错码。回想一下，tcp_xmit_timer在收到一个ACK，确认未发送过的报文段时，复位t_softerror为零。

346-353 如果连接还未建立，而且TCP已经至少4次重传了当前报文段，t_softerror中已存在差错记录，则最新的差错将被保存在插口的so_error变量中，从而应用进程可以调用select对插口进行读写。如果上述条件不满足，当前差错将仍旧保存在t_softerror中。我们在tcp_drop函数中讨论过，如果连接最终由于超时而被丢弃，tcp_drop会把t_softerror赋给插口差错变量errno。任何在插口上等待接收或发送数据的应用进程会被唤醒，并得到相应的差错代码。

27.8 tcp_quench函数

tcp_quench的函数代码在图27-13中给出。TCP在两种情况下调用它：当连接上收到源站抑制差错时，由tcp_input调用。当ip_output返回ENOBUFS差错代码时，由tp_output调用。

tcp_subr.c
```
381 void
382 tcp_quench(inp, errno)
383 struct inpcb *inp;
384 int     errno;
385 {
386     struct tcpcb *tp = intotcpcb(inp);
387     if (tp)
388         tp->snd_cwnd = tp->t_maxseg;
389 }
```
tcp_subr.c

图27-13 tcp_quench 函数

拥塞窗口设定为最大报文段长度，强迫TCP执行慢起动。慢起动门限不变(与tcp_timers处理重传超时的思想相同)，因此，窗口大小将成指数地增加，直至达到snd_ssthresh门限或发生拥塞。

27.9 TCP_REASS宏和tcp_reass函数

TCP报文段有可能乱序到达，因此，在数据上交给应用进程之前，TCP必须设法恢复正确

的报文段次序。例如，如果接收方的接收窗口大小为4096，等待接收的下一个序号为 0。收到的第一个报文段携带0 ~ 1023字节的数据(次序正确)，第二个报文段携带了 2048 ~ 3071字节的数据，很明显，第二个报文段到达的次序差错。如果乱序报文段位于接收窗口内， TCP并不丢弃它，而是将其保存在连接的重组队列中，继续等待中间缺失的报文段 (携带1024 ~ 2047字节的报文段)。这一节我们将讨论处理TCP重组队列的代码，为后两章讨论 tcp_input打下基础。

如果假定某个mbuf中包含IP首部、TCP首部和4字节的用户数据 (回想图2-14的左半部分)，如图27-14所示。此外还假定数据的序号依次为7、8、9和10。

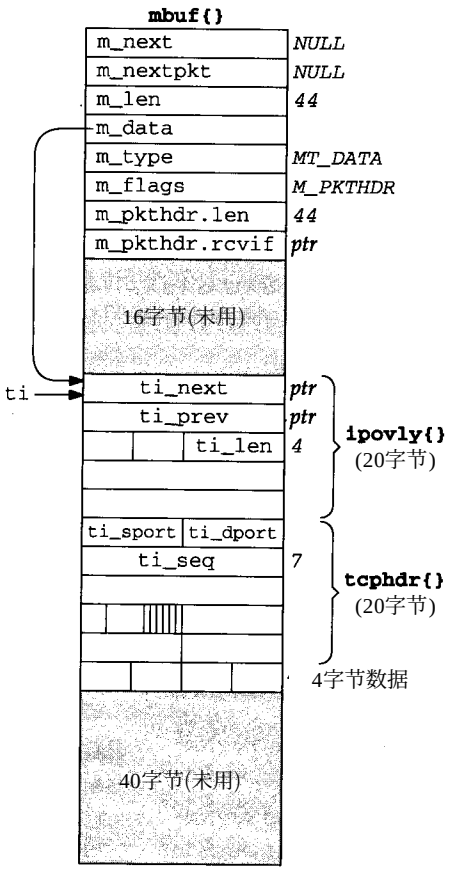


图27-14 举例：带有4字节数据的IP和TCP首部

图24-12中定义的tcpiphdr结构里包含了ipovly和tcphdr两个结构，tcphdr结构在图24-12中给出。图27-14只列出了与重组有关的一些变量： ti_next、ti_prev、ti_len、ti_dport和ti_seq。头两个指针指向由给定连接所有乱序报文段组成的双向链表。链表头保存在连接的 TCP控制块中：结构的头两个成员变量为seg_next和seg_prev。ti_next和ti_prev指针与IP首部的头8个字节重复，只要数据报到达了 TCP，就不再需要这些内容。ti_len等于TCP数据的长度， TCP计算检验和之前首先计算并存储这个字段。

27.9.1 TCP_REASS宏

tcp_input收到数据后，就调用图27-15中的宏TCP_REASS，把数据放入连接的重组队列。TCP_REASS只在一种情况下被调用：参见图29-22。

54-63 tp是指向连接TCP控制块的指针， ti是指向接收报文段的tcpiphdr结构的指针。如果下列3个条件均为真：

1) 报文段到达次序正确 (序号ti_seq等于连接上等待接收的下一序号， rcv_nxt)；并且

2) 连接的重组队列为空(seg_next指向自己，而不是某个mbuf)；并且

3) 连接处于ESTABLISHED状态。

则执行下列步骤：设定延迟 ACK标志；更新rcv_nxt，增加报文段携带的数据长度；如果报文段TCP首部中FIN标志置位，则flags参数中增加TH_FIN标志；更新两个统计值；数据放入插口的接收缓存；唤醒所有在插口上等待接收的应用进程。

tcp_input.c

```
53 #define TCP_REASS(tp, ti, m, so, flags) { \
54     if ((ti)->ti_seq == (tp)->rcv_nxt && \
55         (tp)->seg_next == (struct tcpiphdr *)(tp) && \
56         (tp)->t_state == TCPS_ESTABLISHED) { \
57         tp->t_flags |= TF_DELACK; \
58         (tp)->rcv_nxt += (ti)->ti_len; \
59         flags = (ti)->ti_flags & TH_FIN; \
60         tcpstat.tcps_rcvpack++; \
61         tcpstat.tcps_rcvbyte += (ti)->ti_len; \
62         sbappend(&(so)->so_rcv, (m)); \
63         sorwakeup(so); \
64     } else { \
65         (flags) = tcp_reass((tp), (ti), (m)); \
66         tp->t_flags |= TF_ACKNOW; \
67     } \
68 }
```

tcp_input.c

图27-15 TCP_REASS 宏：向连接的重组队列中添加数据

必须满足前述3个条件的原因是：首先，如果数据次序差错，则必须将其放入重组队列，直至收到了中间缺失的报文段，才能把数据提交给应用进程。第二，即使当前数据到达次序正确，但如果重组队列中已存在乱序数据，则新的数据有可能就是所需的缺失数据，从而能够向应用进程同时提交多个报文段中的数据；第三，尽管允许请求建立连接的 SYN报文段中携带数据，但这些数据在连接进入ESTABLISHED状态之前，必须保存在重组队列中，不允许直接提交给应用进程。

64-67 如果这3个条件不是同时满足，则TCP_REASS宏调用TCP_REASS函数，向重组队列中添加数据。由于收到的报文段如果不是乱序报文段，就有可能是所需的缺失报文段，因此，置位TF_ACKNOW，要求立即发送ACK。TCP的一个重要特性是收到乱序报文段时，必须立即发送ACK，这有助于快速重传算法(29.4节)的实现。

在讨论TCP_REASS函数代码之前，需要先了解图 27-14中TCP首部的两个端口号，ti_sport和ti_dport，所起的作用。其实，只要找到了TCP控制块并调用了TCP_REASS，就不再需要它们了。因此， TCP报文段放入重组队列时，可以把对应 mbuf的地址存储在这两个端口号变量中。对于图 27-14中的报文段，无需这样做，因为IP和TCP的首部都存储在mbuf的数据部分，可直接使用dtom宏。但我们在2.6节讨论m_pullup时曾指出，如果IP和TCP的首部保存在簇中(如图2-16所示，对于最大长度报文这是很正常的)，dtom宏将无法使用。我们在该节中曾提到， TCP把从TCP首部指向mbuf的后向指针(back pointer)存储在TCP的两个端口号字段中。

图27-16举例说明了这一技术的用法，利用它处理连接上的两个乱序报文段，每个报文段都存储在一个mbuf簇中。乱序报文段双向链表的表头是连接的 TCP控制块中的seg_next成员变量。为简化起见，图中未标出 seg_prev指针和指向链表最后一个报文段的 ti_next指针。

接收窗口等待接收的下一个序号为 1(rcv_nxt)，但我们假定这个报文段丢失了。接着又收到了两个报文段，携带 1461~4380字节的数据，这是两个乱序报文段。 TCP调用m_devget把它们放入mbuf簇中，如图2-16所示。

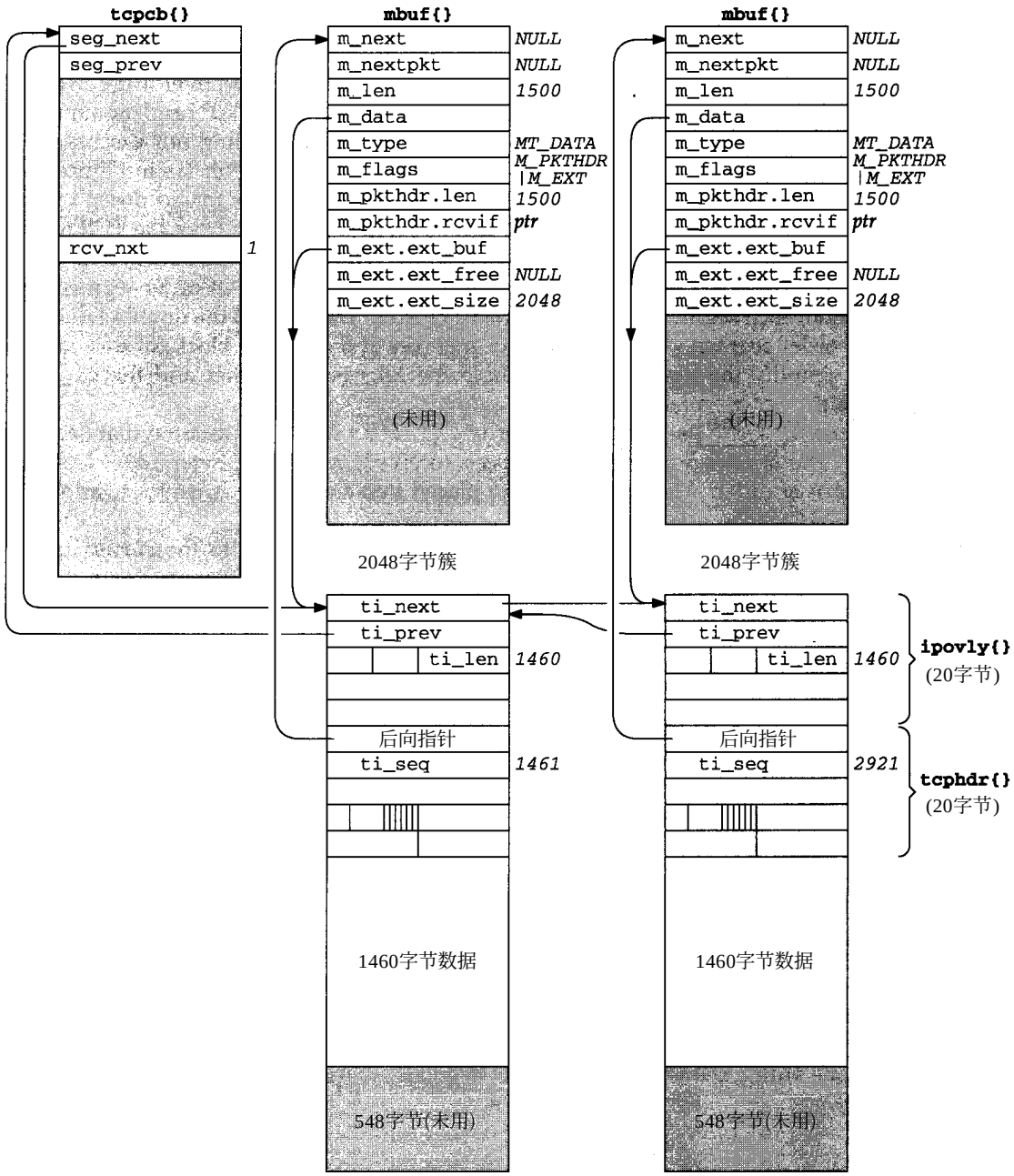


图27-16 两个乱序TCP报文段存储在mbuf簇中

TCP首部的头32 bit存储指向对应mbuf的指针，下面介绍的TCP_REASS函数将用到这个后向指针。

27.9.2 TCP_REASS函数

图27-17给出了TCP_REASS函数的第一部分。参数包括：tp，指向TCP控制块的指针；ti，指向接收报文段IP和TCP首部的指针；m，指向存储接收报文段的mbuf链表的指针。前

面曾提到过，ti既可以指向由m所指向的mbuf的数据区，也可以指向一个簇。

tcp_input.c

```
69 int
70 tcp_reass(tp, ti, m)
71 struct tcpcb *tp;
72 struct tcpiphdr *ti;
73 struct mbuf *m;
74 {
75     struct tcpiphdr *q;
76     struct socket *so = tp->t_inpcb->inp_socket;
77     int     flags;

78     /*
79      * Call with ti==0 after become established to
80      * force pre-ESTABLISHED data up to user socket.
81      */
82     if (ti == 0)
83         goto present;

84     /*
85      * Find a segment that begins after this one does.
86      */
87     for (q = tp->seg_next; q != (struct tcpiphdr *) tp;
88          q = (struct tcpiphdr *) q->ti_next)
89         if (SEQ_GT(q->ti_seq, ti->ti_seq))
90             break;
```

tcp_input.c

图27-17 TCP_REASS 函数：第一部分

69-83 后面将看到，TCP收到一个对SYN的确认时，tcp_input将调用TCP_REASS，并传递一个空的ti指针(图28-20和图29-2)。这意味着连接已建立，可以把SYN报文段中携带的数据(TCP_REASS已将其放入重组队列)提交给应用程序。连接未建立之前，不允许这样做。标志“present”位于图27-23中。

84-90 遍历从seg_next开始的乱序报文段双向链表，寻找序号大于接收报文段序号(ti_seq)的第一个报文段。注意，for循环体中只包含一个if语句。

图27-18的例子中，新报文段到达时重组队列中已有两个报文段。图中标出了指针 q，指向链表的下一个报文段，带有字节 10~15。此外，图中还标出了两个指针 ti_next和ti_prev，起始序号(ti_seq)、长度(ti_len)和数据字节的序号。由于这些报文段较小，每个报文段很可能存储在单一的mbuf中，如图27-14所示。

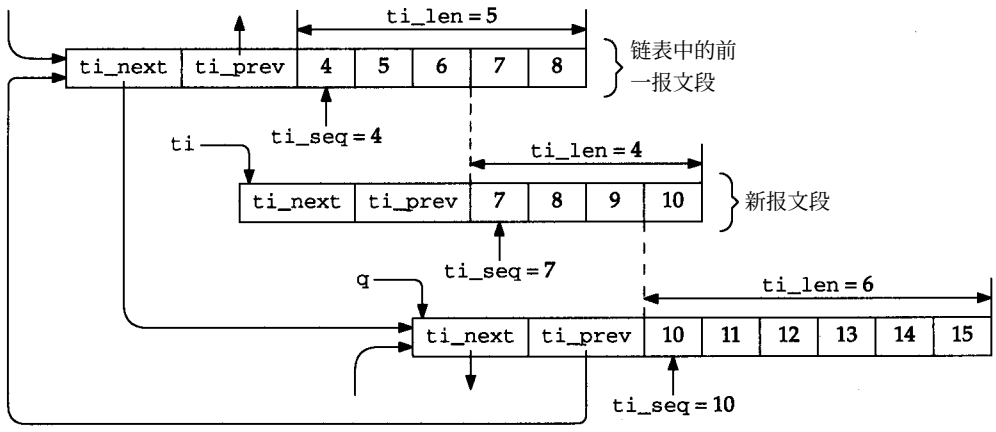


图27-18 存储重复报文段的重组队列举例

图27-19给出了TCP_REASS下一部分的代码

tcp_input.c

```
91      /*
92       * If there is a preceding segment, it may provide some of
93       * our data already.  If so, drop the data from the incoming
94       * segment.  If it provides all of our data, drop us.
95       */
96      if ((struct tcpiphdr *) q->ti_prev != (struct tcpiphdr *) tp) {
97          int     i;
98          q = (struct tcpiphdr *) q->ti_prev;
99          /* conversion to int (in i) handles seq wraparound */
100         i = q->ti_seq + q->ti_len - ti->ti_seq;
101         if (i > 0) {
102             if (i >= ti->ti_len) {
103                 tcpstat.tcps_rcvduppack++;
104                 tcpstat.tcps_rcvdupbyte += ti->ti_len;
105                 m_freem(m);
106                 return (0);
107             }
108             m_adj(m, i);
109             ti->ti_len -= i;
110             ti->ti_seq += i;
111         }
112         q = (struct tcpiphdr *) (q->ti_next);
113     }
114     tcpstat.tcps_rcvoopack++;
115     tcpstat.tcps_rcvoobyte += ti->ti_len;
116     REASS_MBUF(ti) = m;          /* XXX */
```

tcp_input.c

图27-19 TCP_REASS 函数：第二部分

91-107 如果双向链表中q指向的报文段前还存在报文段，则该报文段有可能与新报文段重复，因此，挪动指针q，令其指向q的前一个报文段(图27-18中携带字节4~8的报文段)，计算重复的字节数，并存储在变量i中：

```
i = q->ti_seq + q->ti_len - ti->ti_seq;
  = 4 + 5 -7
  = 2
```

如果i大于0，则链表中原有报文段与新报文段携带的数据间存在重复，如例子中给出的报文段。如果重复的字节数(i)大于或等于新报文段的大小，即新报文段中所有的数据都已包含在原有报文段中，新报文段是重复报文段，应予以丢弃。

108-112 如果只有部分数据重复(如图27-18所示)，m_adj丢弃新报文段起始i字节的数据，并相应更新新报文段的序号和长度。挪动q指针，指向链表中的下一个报文段。图27-20给出了图27-18中各报文段和变量此时的状态。

116 mbuf的地址m存储在TCP首部的源端口号和目的端口号中，也就是我们在本节前面曾提到的后向指针，防止TCP首部被存放在mbuf簇中，而无法使用dtom宏。宏REASS_MBUF定义为：

```
#define REASS_MBUF(ti) (*(struct mbuf **)&((ti)->ti_t))
```

ti_t是一个tcphdr结构(图24-12)，最初的两个成员变量是两个16bit的端口号。请注意图27-19中的注释“XXX”，其中隐含了这样一个假定，指针能够存放在两个端口号占用的32bit空间中。

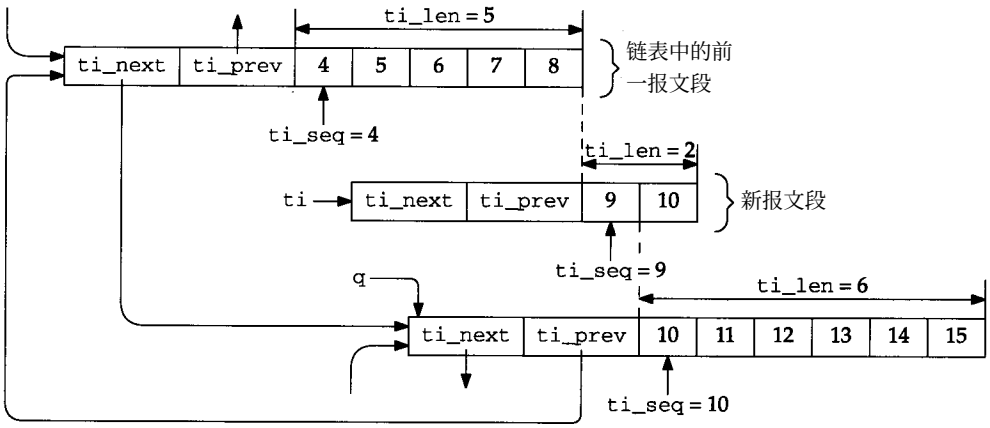


图27-20 删除新报文段中的字节7和8后，更新图27-18

图27-21给出了tcp_reass的第三部分，删除重组队列下一报文段中可能的重复字节。

117-135 如果还有后续报文段，则计算新报文段与下一报文段间重复的字节数，并存储在变量i中。还是以图27-18中的报文段为例，得到：

```
i = 9 + 2 - 10
  = 1
```

因为序号10的字节同时存在于两个报文段中。

根据i值的大小，有可能出现3种情况：

1) 如果i小于等于0，无重复。

2) 如果i小于下一报文段的字节数(q->ti_len)，则有部分重复，调用m_adj，从该报文段中丢弃起始的i字节。

3) 如果i大于等于下一报文段的字节数，则出现完全重复，从链表中删除该报文段。

136-139 代码最后调用insque，把新报文段插入连接的重组双向链表中。图 27-22给出了图27-18中各报文段和变量此时的状态。

tcp_input.c

```
117     /*
118      * While we overlap succeeding segments trim them or,
119      * if they are completely covered, dequeue them.
120      */
121     while (q != (struct tcpiphdr *) tp) {
122         int     i = (ti->ti_seq + ti->ti_len) - q->ti_seq;
123         if (i <= 0)
124             break;
125         if (i < q->ti_len) {
126             q->ti_seq += i;
127             q->ti_len -= i;
128             m_adj(REASS_MBUF(q), i);
129             break;
130         }
131         q = (struct tcpiphdr *) q->ti_next;
132         m = REASS_MBUF((struct tcpiphdr *) q->ti_prev);
133         remque(q->ti_prev);
```

图27-21 TCP_REASS 函数：第三部分

```
134             m_freem(m);
135         }

136         /*
137          * Stick new segment in its place.
138          */
139         insque(ti, q->ti_prev);
```

tcp_input.c

图27-21 (续)

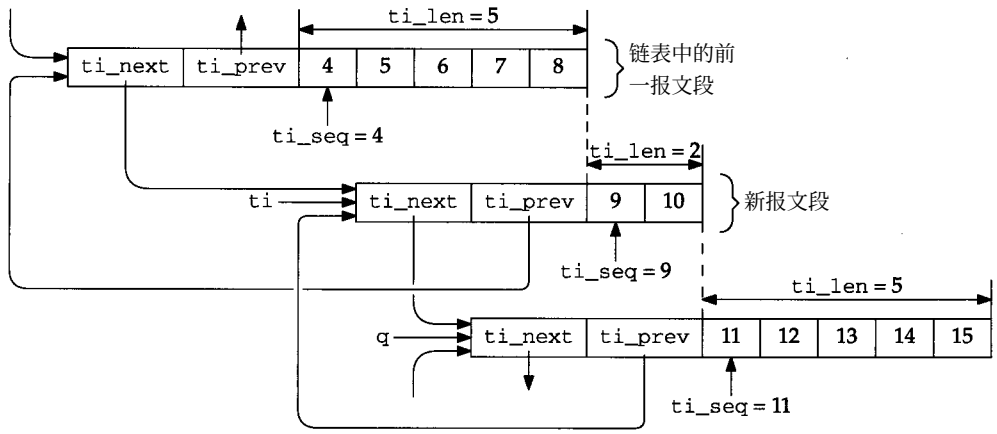


图27-22 丢弃所有重复字节后，更新图27-20

图27-23给出了tcp_reass最后一部分的代码，如果可能，向应用进程递交数据。

tcp_input.c

```
140   present:
141     /*
142      * Present data to user, advancing rcv_nxt through
143      * completed sequence space.
144      */
145     if (TCPS_HAVERCVDSYN(tp->t_state) == 0)
146         return (0);
147     ti = tp->seg_next;
148     if (ti == (struct tcpiphdr *) tp || ti->ti_seq != tp->rcv_nxt)
149         return (0);
150     if (tp->t_state == TCPS_SYN_RECEIVED && ti->ti_len)
151         return (0);
152     do {
153         tp->rcv_nxt += ti->ti_len;
154         flags = ti->ti_flags & TH_FIN;
155         remque(ti);
156         m = REASS_MBUF(ti);
157         ti = (struct tcpiphdr *) ti->ti_next;
158         if (so->so_state & SS_CANTRCVMORE)
159             m_freem(m);
160         else
161             sbappend(&so->so_rcv, m);
162     } while (ti != (struct tcpiphdr *) tp && ti->ti_seq == tp->rcv_nxt);
163     sorwakeup(so);
164     return (flags);
165 }
```

tcp_input.c

图27-23 tcp_reass 函数：第四部分

145-146 如果连接还没有收到SYN(连接处于LISTEN状态或SYN_SENT状态)，不允许向应用进程提交数据，函数返回。当函数被宏 TCP_REASS调用时，返回值 0被赋给宏的参数flags。这种做法带来的副作用是可能会清除 FIN标志。当宏TCP_REASS被图29-22的代码调用时，如果接收报文段包含了 SYN、FIN和数据(尽管不常见，但却是有效的报文段)，会出现这种情况。

147-149 ti设定为链表的第一个报文段。如果链表为空，或者第一个报文段的起始序号(ti->ti_seq)不等于连接等待接收的下一序号 (rcv_nxt)，则函数返回0。如果第二个条件为真，说明在等待接收的下一序号与已收到的数据之间仍然存在缺失报文段。例如，图 27-22中，如果携带4 ~ 8字节的报文段是链表的起始报文段，但 rcv_nxt等于2，字节2和3仍旧缺失，因此，不能把4 ~ 15字节提交给应用进程。返回值0将清除FIN标志(如果该标志设定)，这是因为还有未收到的数据，所以暂时不能处理 FIN。

150-151 如果连接处于SYN_RCVD状态，且报文段长度非零，则函数返回 0。如果两个条件均为真，说明插口在监听过程中收到了携带数据的 SYN报文段。数据将保存在连接队列中，等待三次握手过程结束。

152-164 循环从链表的第一个报文段开始(从前面的测试条件可知，它携带数据的次序已经正确)，把数据放入插口的接收缓存，并更新 rcv_nxt。当链表为空，或者链表下一报文段的序号又出现差错，即当前处理报文段与下一报文段间存在缺失报文段时，循环结束。此时，flags变量(函数的返回值)等于0或者为TH_FIN，取决于放入插口接收缓存的最后一个报文段中是否带有FIN标志。

在所有mbuf都放入插口的接收缓存后， sorwakeup唤醒所有在插口上等待接收数据的应用进程。

27.10 tcp_trace函数

图26-32中，在向IP递交报文段之前， tcp_output调用了tcp_trace函数：

```
if (so->so_options & SO_DEBUG)
    tcp_trace(TA_OUTPUT, tp->t_state, tp, ti, 0);
```

在内核的环形缓存中添加一条记录，这些记录可通过 trpt (8)程序读取。此外，如果内核编译时定义了符号TCPDEBUG，并且变量tcpconsdebug非零，则信息将输出到系统控制台。

任何进程都可以设定TCP的插口选项SO_DEBUG，要求TCP把信息存储到内核的环形缓存中。但只有特权进程或系统管理员才能运行 trpt，因为它必须读取系统内存才能获取这些信息。

尽管可以为任何类型的插口设定 SO_DUBUG选项(如UDP或原始IP)，但只有TCP才会处理它。

这些信息被保存在tcp_debug结构中，如图27-24所示。

35-43 tcp_debug很大(196字节)，因为它包含了其他两个结构：保存 IP和TCP首部的tcpiphdr和完整的TCP控制块tcpcb。由于保存了 TCP控制块，其中的任何变量都可通过trpt打印出来。也就是说，如果 trpt标准输出中没有包含读者感兴趣的信息，可修改源代码以打印控制块中任何想要的信息 (Net/3版支持这种修改)。图25-28中的RTT变量就是通过这种方式得到的。

tcp_debug.h

```
35 struct tcp_debug {
36     n_time  td_time;            /* iptime(): ms since midnight, UTC */
37     short   td_act;             /* TA_xxx value (Figure 27.25) */
38     short   td_ostate;          /* old state */
39     caddr_t td_tcb;             /* addr of TCP connection block */
40     struct tcpiphdr td_ti;      /* IP and TCP headers */
41     short   td_req;             /* PRU_xxx value for TA_USER */
42     struct tcpcb td_cb;         /* TCP connection block */
43 };

53 #define TCP_NDEBUG 100
54 struct tcp_debug tcp_debug[TCP_NDEBUG];
55 int     tcp_debx;
```

tcp_debug.h

图27-24 tcp_debug 结构

53-55 图27-24还定义了数组 tcp_debug，也就是前面提到的环形缓存。数组指针(tcp_debx)初始化为零，该数组约占20 000字节。

内核只调用了tcp_trace 4次，每次调用都会在结构的 td_act变量中存入一个不同的值，如图27-25所示。

td_act	描 述	参 考
TA_DROP	当输入报文段被丢弃时，被tcp_input调用	图29-27
TA_INPUT	输入处理完毕后，调用tcp_output之前	图29-26
TA_OUTPUT	调用ip_output发送报文段之前	图26-32
TA_USER	RPU_*xxx*请求处理完毕后，被tcp_usrreq调用	图30-1

图27-25 td_act 值及相应的tcp_trace 调用

图27-26给出了tcp_trace函数的主要部分，我们忽略了直接输出到控制台的那部分代码。

48-133 在函数被调用时，ostate中保存了连接的前一个状态，与连接的当前状态(保存在控制块中)相比较，可了解连接的状态变迁状况。图 27-25中，TA_OUTPUT不改变连接状态，但其他3个调用则会导致状态的转移。

tcp_debug.c

```
48 void
49 tcp_trace(act, ostate, tp, ti, req)
50 short   act, ostate;
51 struct tcpcb *tp;
52 struct tcpiphdr *ti;
53 int     req;
54 {
55     tcp_seq seq, ack;
56     int     len, flags;
57     struct tcp_debug *td = &tcp_debug[tcp_debx++];

58     if (tcp_debx == TCP_NDEBUG)
59         tcp_debx = 0;           /* circle back to start */

60     td->td_time = iptime();
61     td->td_act = act;
```

图27-26 tcp_trace 函数：在内核的环形缓存中保存信息

```
62      td->td_ostate = ostate;
63      td->td_tcb = (caddr_t) tp;
64      if (tp)
65          td->td_cb = *tp;          /* structure assignment */
66      else
67          bzero((caddr_t) & td->td_cb, sizeof(*tp));
68      if (ti)
69          td->td_ti = *ti;          /* structure assignment */
70      else
71          bzero((caddr_t) & td->td_ti, sizeof(*ti));
72      td->td_req = req;
73  #ifdef TCPDEBUG
74      if (tcpconsdebug == 0)
75          return;
                    /* output information on console */
132 #endif
133 }
```

tcp_debug.c

图27-26 (续)

输出举例

图27-27列出了tcpdump输出的前4行，反映25.12节例子中的三次握手过程和发送的第一个数据报文段(卷1附录A提供了tcpdump输出格式的细节)。

```
1  0.0                  bsdi.1025 > vangogh.discard: S 20288001:20288001(0)
                                                     win 4096 <mss 512>
2  0.362719 (0.3627)    vangogh.discard > bsdi.1025: S 3202722817:3202722817(0)
                                                     ack 20288002 win 8192
                                                     <mss 512>
3  0.364316 (0.0016)    bsdi.1025 > vangogh.discard: . ack 1 win 4096
4  0.415859 (0.0515)    bsdi.1025 > vangogh.discard: . 1:513(512) ack 1 win 4096
```

图27-27 反映图25-28实例的tcpdump 输出

图27-28列出了与之对应的trpt的输出。

图27-28的输出与正常的trpt输出相比略有一些不同：32 bit的数字序号显示为无符号整数(trpt将其差错地打印为有符号整数)；有些trpt按16进制输出的值被改为10进制；为了编制图25-28，作者人为地把从t_rtt到t_rxtcur的值加入到trpt中。

```
953738 SYN_SENT: output 20288001:20288005(4) @0 (win=4096)
       <SYN> -> SYN_SENT
       rcv_nxt 0, rcv_wnd 0
       snd_una 20288001, snd_nxt 20288002, snd_max 20288002
       snd_wl1 0, snd_wl2 0, snd_wnd 0
       REXMT=12 (t_rxtshift=0), KEEP=150
```

图27-28 反映图25-28实例的trpt 输出

```
        t_rtt=1, t_srtt=0, t_rttvar=24, t_rxtcur=12

953739 CLOSED: user CONNECT -> SYN_SENT
        rcv_nxt 0, rcv_wnd 0
        snd_una 20288001, snd_nxt 20288002, snd_max 20288002
        snd_wl1 0, snd_wl2 0, snd_wnd 0
        REXMT=12 (t_rxtshift=0), KEEP=150
        t_rtt=1, t_srtt=0, t_rttvar=24, t_rxtcur=12

954103 SYN_SENT: input 3202722817:3202722817(0) @20288002 (win=8192)
        <SYN,ACK> -> ESTABLISHED
        rcv_nxt 3202722818, rcv_wnd 4096
        snd_una 20288002, snd_nxt 20288002, snd_max 20288002
        snd_wl1 3202722818, snd_wl2 20288002, snd_wnd 8192
        KEEP=14400
        t_rtt=0, t_srtt=16, t_rttvar=4, t_rxtcur=6

954103 ESTABLISHED: output 20288002:20288002(0) @3202722818 (win=4096)
        <ACK> -> ESTABLISHED
        rcv_nxt 3202722818, rcv_wnd 4096
        snd_una 20288002, snd_nxt 20288002, snd_max 20288002
        snd_wl1 3202722818, snd_wl2 20288002, snd_wnd 8192
        KEEP=14400
        t_rtt=0, t_srtt=16, t_rttvar=4, t_rxtcur=6

954153 ESTABLISHED: output 20288002:20288514(512) @3202722818 (win=4096)
        <ACK> -> ESTABLISHED
        rcv_nxt 3202722818, rcv_wnd 4096
        snd_una 20288002, snd_nxt 20288514, snd_max 20288514
        snd_wl1 3202722818, snd_wl2 20288002, snd_wnd 8192
        REXMT=6 (t_rxtshift=0), KEEP=14400
        t_rtt=1, t_srtt=16, t_rttvar=4, t_rxtcur=6
```

图27-28 (续)

在时刻953 738，发送SYN。注意，代码中的时间变量有 8位数字，以毫秒为单位，这里只输出了低6位。输出的结束序号(20 288 005)是差错的。SYN中确实携带了4字节的内容，但并非数据，而是MSS选项。重传定时器设定为6秒(REXMT)，保活定时器为75秒(KEEP)，这些定时器值均以500 ms滴答为单位。t_rtt等于1，意味对该报文段计时，测量RTT样本值。

发送SYN是为了响应应用进程的connect调用。一毫秒后，这次系统调用的信息被写入内核的环形缓存。尽管是因为应用进程调用了 connect，才导致发送SYN报文段，但TCP在处理完PRU_CONNECT请求后，才调用tcp_trace，环形缓存中实际写入了两条记录。此外，应用进程调用connect时，连接状态为CLOSED，发送完SYN后，状态变迁至SYN_SENT，这也是两条记录仅有的不同之处。

第三条记录，时刻954 103，与第一条记录相隔365 ms (tcpdump显示时间差为362.7ms)，即为图25-28中“实际时间差(ms)”一栏的填充值。收到带有 SYN和ACK的报文段后，连接状态从SYN_SENT转移到ESTABLISHED。因为计时报文段已得到确认，更新 RTT估计器值。

第四条记录反映了三次握手过程中的第三个报文段：确认对端的 SYN。因为是纯ACK报文段，不用对它计时(rtt等于0)，它在时刻954 103被发送。connect系统调用返回，应用进程接着调用write发送数据，产生TCP输出。

第五条记录反映了这个数据报文段，在时刻 954 153，三次握手结束后50 ms，被发送。它携带50字节 的数据，起始序号为20 288 002。重传定时器设为3秒，需要计时。

应用进程继续调用write发送数据。尽管不再显示更多记录，但很明显，接下来的 3条记

录也都是在TCP处理完PRU_SEND请求后写入环形缓存的。第一次PRU_SEND请求，生成我们已看到的第一个512字节的输出报文段，其他3次请求不会引发TCP输出报文段，此时连接正处于慢起动状态。只生成4条记录是因为，图25-28的例子中的TCP发送缓存大小只有4096，mbuf簇大小为1024。一旦发送缓存被占满，应用进程就进入休眠状态。

27.11 小结

本章介绍了各种TCP函数，为后续章节打下基础。

TCP连接正常关闭时，向对端发送FIN，并等待4次报文交换过程结束。它被丢弃时，只需发送RST。

路由表中的每条记录都包含8个变量，其中有3个在连接关闭时更新，有6个用于新连接的建立，从而内核能够跟踪与同一目标之间建立的正常连接的某些特性，如RTT估计器值和慢起动门限。系统管理员可以设置或锁定部分变量，如MTU、接收管道大小和发送管道大小，这些特性会影响到达该目标的连接的性能。

TCP对收到的ICMP差错有一定的容错性——不会导致TCP终止已建立的连接。Net/3处理ICMP差错的方式与早期的Berkeley版本不同。

TCP报文段可能乱序到达，并包含重复数据，TCP必须处理这些异常现象。TCP为每条连接维护一个重组队列，保存乱序报文段，处理之后再提交给应用进程。

最后介绍了选定插口选项SO_DEBUG时，内核中保存的信息。除某些程序如tcpdump之外，这些内容也是很有用的调试工具。

习题

27.1 为什么图27-1中最后一行的errno等于0？

27.2 rmx_rtt中存储的最大值是多少？

27.3 为了保存某个给定主机的路由信息(图27-3)，我们用手工在本地的路由表中添加一条到达该主机的路由。之后，运行FTP客户程序，向这台主机发送足够多的数据，如图27-4所要求的。但终止FTP客户程序后，检查路由表，到达该主机的所有变量依旧为0。出了什么问题？

第28章　TCP的输入

28.1　引言

TCP输入处理是系统中最长的一部分代码，函数 tcp_input约有1100行代码。输入报文段的处理并不复杂，但非常繁琐。许多实现，包括 Net/3，都完全遵循RFC 793中定义的输入事件处理步骤，它详细定义了如何根据连接的当前状态，处理不同的输入报文段。

当收到的数据报的协议字段指明这是一个 TCP报文段时， ipintr(通过协议转换表中的pr_input函数)会调用tcp_input进行处理。tcp_input在软件中断一级执行。

函数非常长，我们将分两章讨论。图 28-1列出了tcp_input中的处理框架。本章将结束对RST报文段处理的讲解，从下一章开始介绍 ACK报文段的处理。

头几个步骤是非常典型的：对输入报文段做有效性验证 (检验和、长度等)，以及寻找连接的PCB。尽管后面还有大量代码，但通过“ 首部预测 (header prediction) ”(28.4节)，算法却有可能完全跳过后续的逻辑。首部预测算法是基于这样的假定：一般情况下，报文段既不会丢失，次序也不会错误，因此，对于给定连接， TCP总能猜到下一个接收报文段的内容。如果算法起作用，函数直接返回，这是tcp_input中最快的一条执行路径。

如果算法不起作用，函数在“ dodata ”处结束，测试几个标志，并且若需要对接收报文段作出响应，则调用tcp_output。

```
void
tcp_input()
{
    checksum TCP header and data;
findpcb:
    locate PCB for segment;
    if (not found)
        goto dropwithreset;
    reset idle time to 0 and keepalive timer to 2 hours;
    process options if not LISTEN state;
    if (packet matched by header prediction) {
        completely process received segment;
        return;
    }
    switch (tp->t_state) {
    case TCPS_LISTEN:
        if SYN flag set, accept new connection request;
        goto trimthenstep6;
    case TCPS_SYN_SENT:
        if ACK of our SYN, connection completed;
trimthenstep6:
        trim any data not within window;
```

图28-1　TCP输入处理步骤小结

```
            goto step6;
        }

        process RFC 1323 timestamp;
        check if some data bytes are within the receive window;
        trim data segment to fit within window;

        if (RST flag set) {
            process depending on state;
            goto drop;
        }                               /* Chapter 28 finishes here */

        if (ACK flag set) {             /* Chapter 29 starts here */
            if (SYN_RCVD state)
                simultaneous open complete;
            if (duplicate ACK)
                fast recovery algorithm;
            update RTT estimators if segment timed;
            open congestion window;
            remove ACKed data from send buffer;
            change state if in FIN_WAIT_1, CLOSING, or LAST_ACK state;
        }
    step6:
        update window information;
        process URG flag;
    dodata:
        process data in segment, add to reassembly queue;

        if (FIN flag is set)
            process depending on state;

        if (SO_DEBUG socket option)
            tcp_trace(TA_INPUT);

        if (need output || ACK now)
            tcp_output();
        return;

    dropafterack:
        tcp_output() to generate ACK;
        return;

    dropwithreset:
        tcp_respond() to generate RST;
        return;

    drop:
        if (SO_DEBUG socket option)
            tcp_trace(TA_DROP);
        return;
    }
```

图28-1 (续)

函数结尾处有3个标注，处理出现差错时控制会跳转到这些地方： dropafterack、dropwithreset和drop。标注中出现的“ drop ”指丢弃当前处理的报文段，而非丢弃连接。不过，当控制跳转到dropwithreset时，将发送RST，从而丢弃连接。

函数仅有的另一个分支是首部预测算法后的 switch语句，如果连接处于 LISTEN或SYN_SENT状态时收到了一个有效的SYN报文段，它负责分别进行处理。 trimthenstep6

处的代码结束后，跳转到step 6继续执行正常的流程。

28.2 预处理

图28-2的代码包含一些声明，并对收到的TCP报文段进行预处理。

1. 从第一个mbuf中获取IP和TCP首部

170-204 参数iphlen等于IP首部长度，包括可能的IP选项。如果长度大于20字节，可知存在IP选项，调用ip_stripoptions丢弃这些选项。TCP忽略除源选路之外的所有IP选项，源选路选项由IP特别保存(9.6节)，TCP能够读取其内容(图28-7)。如果簇中第一个mbuf的容量小于IP/TCP首部大小(40字节)，则调用m_pullup，试着把最初的40字节移入第一个mbuf中。

tcp_input.c

```
170 void
171 tcp_input(m, iphlen)
172 struct mbuf *m;
173 int     iphlen;
174 {
175     struct tcpiphdr *ti;
176     struct inpcb *inp;
177     caddr_t optp = NULL;
178     int     optlen;
179     int     len, tlen, off;
180     struct tcpcb *tp = 0;
181     int     tiflags;
182     struct socket *so;
183     int     todrop, acked, ourfinisacked, needoutput = 0;
184     short   ostate;
185     struct in_addr laddr;
186     int     dropsocket = 0;
187     int     iss = 0;
188     u_long  tiwin, ts_val, ts_ecr;
189     int     ts_present = 0;

190     tcpstat.tcps_rcvtotal++;
191     /*
192      * Get IP and TCP header together in first mbuf.
193      * Note: IP leaves IP header in first mbuf.
194      */
195     ti = mtod(m, struct tcpiphdr *);
196     if (iphlen > sizeof(struct ip))
197                 ip_stripoptions(m, (struct mbuf *) 0);
198     if (m->m_len < sizeof(struct tcpiphdr)) {
199         if ((m = m_pullup(m, sizeof(struct tcpiphdr))) == 0) {
200             tcpstat.tcps_rcvshort++;
201             return;
202         }
203         ti = mtod(m, struct tcpiphdr *);
204     }
```

tcp_input.c

图28-2 tcp_input 函数：变量声明及预处理

图28-3给出了函数下一部分的代码，验证TCP检验和及偏移字段。

2. 验证TCP检验和

205-217 tlen指TCP报文段的长度，即IP首部后的字节数。前面介绍过，IP已经从

ip_len中减去了IP的首部长度，因此，变量len就等于整个IP数据报的长度，即包括伪首部在内的需要计算检验和的数据长度。根据TCP检验和计算的要求，填充伪首部中的各个字段，如图23-19所示。

tcp_input.c

```
205     /*
206      * Checksum extended TCP header and data.
207      */
208     tlen = ((struct ip *) ti)->ip_len;
209     len = sizeof(struct ip) + tlen;
210     ti->ti_next = ti->ti_prev = 0;
211     ti->ti_x1 = 0;
212     ti->ti_len = (u_short) tlen;
213     HTONS(ti->ti_len);
214     if (ti->ti_sum = in_cksum(m, len)) {
215         tcpstat.tcps_rcvbadsum++;
216         goto drop;
217     }
218     /*
219      * Check that TCP offset makes sense,
220      * pull out TCP options and adjust length.       XXX
221      */
222     off = ti->ti_off << 2;
223     if (off < sizeof(struct tcphdr) || off > tlen) {
224         tcpstat.tcps_rcvbadoff++;
225         goto drop;
226     }
227     tlen -= off;
228     ti->ti_len = tlen;
```

tcp_input.c

图28-3 tcp_input 函数：验证TCP检验和及偏移字段

3. 验证TCP偏移字段

218-228 TCP的偏移字段，ti_off，是以32 bit为单位的TCP首部长度值，包括所有的TCP选项。把它乘以4(得到TCP报文段中第一个数据字节所在位置的偏移量)，并验证其有效性。偏移量必须大于等于标准TCP首部的大小(20字节)，并且小于等于TCP报文段的长度。

从TCP长度变量tlen中减去首部长度，得到报文段中携带的数据字节数 (可能为0)，并把这个值赋给tlen，以及TCP首部的变量ti_len。函数中会多次用到这个值。

图28-4给出了函数下一部分的代码：处理特定的TCP选项。

tcp_input.c

```
229     if (off > sizeof(struct tcphdr)) {
230         if (m->m_len < sizeof(struct ip) + off) {
231             if ((m = m_pullup(m, sizeof(struct ip) + off)) == 0) {
232                 tcpstat.tcps_rcvshort++;
233                 return;
234             }
235             ti = mtod(m, struct tcpiphdr *);
236         }
237         optlen = off - sizeof(struct tcphdr);
238         optp = mtod(m, caddr_t) + sizeof(struct tcpiphdr);
239         /*
240          * Do quick retrieval of timestamp options ("options
```

图28-4 tcp_input 函数：处理特定的TCP选项

```
241          * prediction?").  If timestamp is the only option and it's
242          * formatted as recommended in RFC 1323 Appendix A, we
243          * quickly get the values now and not bother calling
244          * tcp_dooptions(), etc.
245          */
246         if ((optlen == TCPOLEN_TSTAMP_APPA ||
247              (optlen > TCPOLEN_TSTAMP_APPA &&
248               optp[TCPOLEN_TSTAMP_APPA] == TCPOPT_EOL)) &&
249             *(u_long *) optp == htonl(TCPOPT_TSTAMP_HDR) &&
250             (ti->ti_flags & TH_SYN) == 0) {
251             ts_present = 1;
252             ts_val = ntohl(*(u_long *) (optp + 4));
253             ts_ecr = ntohl(*(u_long *) (optp + 8));
254             optp = NULL;        /* we've parsed the options */
255         }
256     }
```

tcp_input.c

图28-4 (续)

4. 把IP和TCP首部及选项放入第一个mbuf

230-236 如果首部长度大于20，说明存在TCP选项。必要时调用m_pullup，把标准IP首部、标准TCP首部的所有TCP选项放入簇中的第一个mbuf中。因为3部分数据最大只能为80字节(20+20+40)，因此，必定能够放入第一个存储数据分组首部的mbuf中。

此处能够造成m_pullup失败的惟一原因是IP数据分组的字节数小于20加上TCP首部长度，而且已通过TCP检验和的验证，我们认为m_pullup不可能失败。但有一点，图28-2中调用的m_pullup，将共享计数器tcps_rcvshort，因此，查看tcps_rcvshort并不能说明哪一个调用失败。不管怎样，从图24-5可知，即使收到九百万个TCP报文段之后，这个计数器仍旧为0。

5. 快速处理时间戳选项

237-255 optlen等于首部中TCP选项的长度，optp是指向第一个选项字节的指针。如果下列3个条件均为真，说明只存在时间戳选项，而且格式正确：

1) TCP选项长度等于12(TCPOLEN_TSTAMP_APPA)；或TCP选项长度大于12，但optp[12]等于选项结束字节。

2) 选项的头4个字节等于0x0101080a(TCPOPT_TSTAMP_HDR，在26.6节曾讨论过)。

3) SYN标志未置位(说明连接已建立，如果报文段中出现时间戳选项，意味着连接双方都同意使用这一选项)。

如果上述条件全部满足，则ts_present置为1；从接收报文段首部获取两个时间戳值，分别赋给ts_val和ts_ecr；optp置为空，因为所有选项已处理完毕。这种辨认时间戳的方法可以避免调用通用选项处理函数tcp_dooptions，从而使后者能够专门处理只出现在SYN报文段中的各种选项(MSS和窗口大小选项)。如果连接双方同意使用时间戳，那么在建立的连接上交换的几乎所有报文段中都可能带有时间戳选项，因此，必须加快其处理速度。

图28-5给出了函数下一部分的代码，寻找报文段的Internet PCB。

6. 保存输入标志，把字段转换为主机字节序

257-264 接收报文段中的标志(SYN、FIN等)被保存在本地变量tiflags中，因为函数在处理过程中会多次引用这些标志。TCP首部的两个16 bit字段和两个32 bit序号被转换回主机字

节序，而两个16 bit端口号则不做处理，依旧为网络字节序，因为 Internet PCB中的端口号是依照网络字节序存储的。

tcp_input.c

```
257     tiflags = ti->ti_flags;

258     /*
259      * Convert TCP protocol specific fields to host format.
260      */
261     NTOHL(ti->ti_seq);
262     NTOHL(ti->ti_ack);
263     NTOHS(ti->ti_win);
264     NTOHS(ti->ti_urp);

265     /*
266      * Locate pcb for segment.
267      */
268   findpcb:
269     inp = tcp_last_inpcb;
270     if (inp->inp_lport != ti->ti_dport ||
271         inp->inp_fport != ti->ti_sport ||
272         inp->inp_faddr.s_addr != ti->ti_src.s_addr ||
273         inp->inp_laddr.s_addr != ti->ti_dst.s_addr) {
274         inp = in_pcblookup(&tcb, ti->ti_src, ti->ti_sport,
275                            ti->ti_dst, ti->ti_dport, INPLOOKUP_WILDCARD);
276         if (inp)
277             tcp_last_inpcb = inp;
278         ++tcpstat.tcps_pcbcachemiss;
279     }
```

tcp_input.c

图28-5 tcp_input 函数：寻找报文段的Internet PCB

7. 寻找Internet PCB

265-279 TCP的缓存(tcp_last_inpcb)中保存了收到的最后一个报文段的PCB地址，采用的技术与UDP相同。TCP使用一对插口来识别连接，寻找PCB时插口对中4个元素的比较次序与udp_input相同。如果与TCP缓存中的记录不匹配，则调用 in_pcblookup，把新的PCB放入缓存。

TCP中不会出现我们在 UDP中曾遇到过的问题：由于高速缓存中存在通配项 (wildcard entry)，导致匹配成功率很低。因为只有处于监听状态的服务器，才可能在其插口中保存通配项。连接一旦建立，插口对的4个元素将全部填入确定值。从图 24-5可知，高速缓存命中率能够达到80%。

图28-6给出了函数下一部分的代码。

tcp_input.c

```
280     /*
281      * If the state is CLOSED (i.e., TCB does not exist) then
282      * all data in the incoming segment is discarded.
283      * If the TCB exists but is in CLOSED state, it is embryonic,
284      * but should either do a listen or a connect soon.
285      */
286     if (inp == 0)
287         goto dropwithreset;
288     tp = intotcpcb(inp);
```

图28-6 tcp_input 函数：判断是否应丢弃报文段

```
289     if (tp == 0)
290         goto dropwithreset;
291     if (tp->t_state == TCPS_CLOSED)
292         goto drop;
293     /* Unscale the window into a 32-bit value. */
294     if ((tiflags & TH_SYN) == 0)
295         tiwin = ti->ti_win << tp->snd_scale;
296     else
297         tiwin = ti->ti_win;
```

tcp_input.c

图28-6 (续)

8. 丢弃报文段并生成RST

280-287 如果没有找到PCB，则丢弃输入报文段，并发送RST作为响应。例如，TCP收到了一个SYN，但报文段指定的服务器并不存在，则直接向对端发送 RST。回想一下，出现这种情况时UDP的处理方式，它将发送一个ICMP端口不可达差错。

288-290 如果PCB存在，但对应的TCP控制块不存在，可能插口已关闭(tcp_close释放TCP之后，才释放PCB)，则丢弃输入报文段，并发送RST作为响应。

9. 丢弃报文段且不发送响应

291-292 如果TCP控制块存在，但连接状态为CLOSED，说明插口已创建，且得到了本地地址和本地端口号，但还未调用connect或者listen。报文段被丢弃，且不发送任何响应。举例来说，如果客户向服务器发送的连接请求报文段到达时，服务器已调用了 bind，但还未调用listen。这种情况下，客户连接请求将超时，导致重传 SYN。

10. 不改变通告窗口大小

293-297 如果需要支持窗口大小选项，连接双方都必须在连接建立时通过窗口大小选项规定窗口缩放因子。如果报文段中包含 SYN，说明此时窗口缩放因子还未定义，因此，直接把TCP首部的窗口字段值复制给 tiwin；否则，首部中的16 bit数值应根据窗口缩放因子左移，得到32 bit的数值。

图28-7给出了函数的下一部分代码，如果选取了插口的 SO_DEBUG选项，或者插口正处于监听状态，则完成一些相应的预处理工作。

tcp_input.c

```
298     so = inp->inp_socket;
299     if (so->so_options & (SO_DEBUG | SO_ACCEPTCONN)) {
300         if (so->so_options & SO_DEBUG) {
301             ostate = tp->t_state;
302             tcp_saveti = *ti;
303         }
304         if (so->so_options & SO_ACCEPTCONN) {
305             so = sonewconn(so, 0);
306             if (so == 0)
307                 goto drop;
308             /*
309              * This is ugly, but ....
310              *
311              * Mark socket as temporary until we're
312              * committed to keeping it.  The code at
313              * 'drop' and 'dropwithreset' check the
```

图28-7 tcp_input 函数：处理调试选项和监听状态的插口

```
314                 * flag dropsocket to see if the temporary
315                 * socket created here should be discarded.
316                 * We mark the socket as discardable until
317                 * we're committed to it below in TCPS_LISTEN.
318                 */
319                dropsocket++;
320                inp = (struct inpcb *) so->so_pcb;
321                inp->inp_laddr = ti->ti_dst;
322                inp->inp_lport = ti->ti_dport;
323 #if BSD>=43
324                inp->inp_options = ip_srcroute();
325 #endif
326                tp = intotcpcb(inp);
327                tp->t_state = TCPS_LISTEN;

328                /* Compute proper scaling value from buffer space */
329                while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
330                       TCP_MAXWIN << tp->request_r_scale < so->so_rcv.sb_hiwat)
331                    tp->request_r_scale++;
332            }
333        }
```

tcp_input.c

图28-7 (续)

11. 如果选定了插口调试选项，则保存连接状态及IP和TCP首部

300-303 如果SO_DEBUG选项置位，则保存当前连接状态 (ostate)及IP和TCP首部(tcp_saveti)。函数结束时，这些信息将作为参数传给tcp_trace(图29-26)。

12. 如果监听插口收到了报文段，则创建新的插口

304-319 如果有报文段到达处于监听状态的插口(listen置位SO_ACCEPTCONN)，则调用sonewconn创建新的插口。发出PRU_ATTACH协议请求(图30-2)，分配Internet PCB和TCP控制块。在TCP最终接受连接请求之前，还需做更多的处理(如一个最基本的问题，报文段中是否包含SYN)，如果发现差错，置位dropsocket标志，控制跳转至标注“drop”和“dropwithreset”，丢弃新的插口。

320-326 inp和tp将指向新建的插口。本地地址和本地端口号直接从接收报文段TCP首部的目的地址和目的端口号字段中复制。如果输入数据报中有源选路的路由，TCP调用ip_srcroute，得到指向保存数据报源选路选项的mbuf的指针，并赋给inp_options。TCP向连接发送数据时，tcp_output会把源选路选项传给ip_output，使用与之相同的逆向路由。

327 新插口的状态设为LISTEN。如果接收报文段中包含SYN，控制将转到图28-16中的代码，完成连接建立请求的处理。

13. 计算窗口缩放因子

328-331 窗口缩放因子取决于接收缓存的大小。如果接收缓存大于通告窗口的最大值(65535，TCP_MAXWIN)，则左移65535，直到结果大于接收缓存大小，或者窗口缩放因子已等于最大值(14，TCP_MAX_WINSHIFT)。注意，窗口缩放因子的选取基于监听插口的接收缓存，也就是说，应用进程调用listen进入监听状态之前，应首先设定SO_RCVBUF插口选项，或者继承tcp_recvspace中的默认值。

窗口缩放因子最大值等于14，而65535×2^{14}等于1 073 725 440，已远远大于接收

缓存的最大值(Net/3中为2626 144)，因此，在窗口缩放因子远小于14时，循环即终止。参见习题28.1和28.2。

图28-8给出了TCP输入处理下一部分的代码。

tcp_input.c

```
334     /*
335      * Segment received on connection.
336      * Reset idle time and keepalive timer.
337      */
338     tp->t_idle = 0;
339     tp->t_timer[TCPT_KEEP] = tcp_keepidle;

340     /*
341      * Process options if not in LISTEN state,
342      * else do it below (after getting remote address).
343      */
344     if (optp && tp->t_state != TCPS_LISTEN)
345         tcp_dooptions(tp, optp, optlen, ti,
346                       &ts_present, &ts_val, &ts_ecr);
```

tcp_input.c

图28-8 tcp_inpu t函数：复位空闲时间和保活定时器，处理应用进程选项

14. 复位空闲时间和保活定时器

334-339 由于连接上收到了报文段，t_idle重设为0。保活定时器复位为2小时。

15. 如果不处于监听状态，处理TCP选项

340-346 如果TCP首部中有选项，并且连接状态不等于LISTEN，调用tcp_dooptions进行处理。前面介绍过，如果连接已建立，接收报文段中只存在时间戳选项，并且时间戳选项格式符合RFC 1323附录A的建议，这种情况在图28-4中已处理过，而且optp被置为空。如果插口处于监听状态，TCP把对端地址保存在PCB中之后，才会调用tcp_dooptions，这是因为处理MSS选项时需要了解到达对端的路由，具体代码如图28-17所示。

28.3 tcp_dooptions函数

函数处理Net/3支持的5个TCP选项(图26-4)：EOL、NOP、MSS、窗口大小和时间戳。图28-9给出了函数的第一部分。

tcp_input.c

```
1213 void
1214 tcp_dooptions(tp, cp, cnt, ti, ts_present, ts_val, ts_ecr)
1215 struct tcpcb *tp;
1216 u_char *cp;
1217 int     cnt;
1218 struct tcpiphdr *ti;
1219 int    *ts_present;
1220 u_long *ts_val, *ts_ecr;
1221 {
1222     u_short mss;
1223     int     opt, optlen;

1224     for (; cnt > 0; cnt -= optlen, cp += optlen) {
1225         opt = cp[0];
1226         if (opt == TCPOPT_EOL)
```

图28-9 tcp_dooptions 函数：处理EOL和NOP选项

```
1227            break;
1228        if (opt == TCPOPT_NOP)
1229            optlen = 1;
1230        else {
1231            optlen = cp[1];
1232            if (optlen <= 0)
1233                break;
1234        }
1235        switch (opt) {

1236        default:
1237            continue;
```

tcp_input.c

图28-9 (续)

1. 获取选项类型的长度

1213-1229 代码遍历TCP首部选项，遇到EOL(选项终止)时终止循环，函数返回；遇到NOP时，将其长度置为1，因为它后面不带长度字段（图26-16），控制转到switch语句的default子句，对其不做处理。

1230-1234 所有其他选项的长度保存在optlen中。

所有新增的Net/3不支持的TCP选项都被忽略。这是因为：

1) 将来定义的所有新选项都将带有长度字段(NOP和EOL是仅有的两个不带长度字段的选项)，而for语句的每次循环都跳过optlen字节。

2) switch语句的default子句忽略所有未知选项。

图28-10给出了tcp_dooptions最后一部分的代码，处理MSS、窗口大小和时间戳选项。

2. MSS选项

1238-1246 如果长度不等于4(TCPOLEN_MAXSEG)，或者报文段不带SYN标志，则忽略该选项。否则，复制两个MSS字节到本地变量，转换为主机字节序，调用tcp_mss完成处理。tcp_mss负责更新TCP控制块中的变量t_maxseg，即发向对端的报文段中允许携带的最大字节数。

3. 窗口大小选项

1247-1254 如果长度不等于4(TCPOLEN_WINDOW)，或者报文段不带SYN标志，则忽略该选项。Net/3置位TF_RCVD_SCALE，说明收到了一个窗口大小选项请求，并在requested_s_scale中保存缩放因子。由于cp[2]只有一个字节，因此，不存在边界问题。当连接转移到ESTABLISHED状态时，如果连接双方都同意支持窗口大小选项，则使用这一功能。

4. 时间戳选项

1255-1273 如果长度不等于10(TCPOLEN_TIMESTAMP)，则忽略该选项。否则，ts_present指向的标志被置位1，两个时间戳值分别保存在ts_val和ts_ecr所指向的变量中。如果收到的报文段带有SYN标志，Net/3置位TF_RCVD_TSTMP，说明收到了一个时间戳请求。ts_recent等于收到的时间戳值，ts_recent_age等于tcp_now，从系统初启到目前的时间，以500ms滴答为单位。

[Partridge 1993]介绍了一种基于Van Jacobson的研究成果，速度更快的TCP首部预测算法实现。

图28-11给出了首部预测的第一部分代码。

tcp_input.c

```
347     /*
348      * Header prediction: check for the two common cases
349      * of a uni-directional data xfer.  If the packet has
350      * no control flags, is in-sequence, the window didn't
351      * change and we're not retransmitting, it's a
352      * candidate.  If the length is zero and the ack moved
353      * forward, we're the sender side of the xfer.  Just
354      * free the data acked & wake any higher-level process
355      * that was blocked waiting for space.  If the length
356      * is non-zero and the ack didn't move, we're the
357      * receiver side.  If we're getting packets in order
358      * (the reassembly queue is empty), add the data to
359      * the socket buffer and note that we need a delayed ack.
360      */
361     if (tp->t_state == TCPS_ESTABLISHED &&
362      (tiflags & (TH_SYN | TH_FIN | TH_RST | TH_URG | TH_ACK)) == TH_ACK &&
363         (!ts_present || TSTMP_GEQ(ts_val, tp->ts_recent)) &&
364         ti->ti_seq == tp->rcv_nxt &&
365         tiwin && tiwin == tp->snd_wnd &&
366         tp->snd_nxt == tp->snd_max) {

367         /*
368          * If last ACK falls within this segment's sequence numbers,
369          *  record the timestamp.
370          */
371         if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
372             SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len)) {
373             tp->ts_recent_age = tcp_now;
374             tp->ts_recent = ts_val;
375         }
```

tcp_input.c

图28-11 tcp_input 函数：首部预测，第一部分

1. 判定收到的报文段是否是等待接收的报文段

347-366 下列6个条件必须全真，才能说明收到的报文段是连接正等待接收的数据报文段或ACK报文段：

1) 连接状态等于ESTABLISHED。

2) 下列4个控制标志必须不设定：SYN、FIN、RST、或URG。但ACK标志必须置位。换言之，TCP的6个控制标志中，ACK标志必须置位，前面列出的4个标志必须清除，PSH标志置位与否无关紧要(连接处于ESTABLISHED状态时，除非RST标志置位，一般情况下，ACK都会置位)。

3) 如果报文段带有时间戳选项，则最新时间戳值(ts_val)必须大于或等于连接上以前收到的时间戳值(ts_recent)。本质上说，这就是PAWS测试，28.7节将详细介绍PAWS。如果ts_val小于ts_recent，则新报文段是乱序报文段，因为它的发送时间早于连接上收到的上一个报文段。由于对端通常把时钟值填充到时间戳字段(Net/3的全局变量tcp_now)，收到的时间戳正常情况下应该是一个单调递增的序列。

并非每个顺序到达报文段中的时间戳都会增加。事实上，Net/3系统每500 ms增加一次时

钟值(tcp_now)，在这段时间间隔中，完全可能发送多个报文段。假定利用时间戳和序号构成一个64 bit的数值，序号放在低32 bit，时间戳放在高32 bit，对于每个顺序报文段，这个64 bit的值都至少会增加1(应考虑取模算法)。

4) 报文段的起始序号(ti_seq)必须等于连接上等待接收的下一个序号(rcv_nxt)。如果这个条件为假，那么收到的报文段是重传报文段或是乱序报文段。

5) 报文段通告的窗口大小必须非零(tiwin)，必须等于当前发送窗口(snd_wnd)。也就是说，无需更新当前发送窗口。

6) 下一个发送序号(snd_nxt)必须等于已发送的最大序号(snd_max)，也就是说，上一个发送报文段不是重传报文段。

2. 根据接收的时间戳更新ts_recent

367-375 如果存在时间戳选项，并且时间戳值满足图26-18中的测试条件，则把收到的时间戳值(ts_val)赋给ts_recent，并在ts_recent_age中保存当前时钟(tcp_now)。

前面讨论过图26-18中的时间戳有效性测试条件所存在的问题，并在图26-20中给出了正确的测试条件。但在首部预测算法的实现中，图26-20中的TSTMP_GEQ测试是多余的，因为图28-11起始处的if语句已完成了这一测试

图28-12给出了首部预测的下一部分代码，用于单向数据的发送方：处理输出数据的ACK。

tcp_input.c

```
376         if (ti->ti_len == 0) {
377             if (SEQ_GT(ti->ti_ack, tp->snd_una) &&
378                 SEQ_LEQ(ti->ti_ack, tp->snd_max) &&
379                 tp->snd_cwnd >= tp->snd_wnd) {
380                 /*
381                  * this is a pure ack for outstanding data.
382                  */
383                 ++tcpstat.tcps_predack;
384                 if (ts_present)
385                     tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
386                 else if (tp->t_rtt &&
387                         SEQ_GT(ti->ti_ack, tp->t_rtseq))
388                     tcp_xmit_timer(tp, tp->t_rtt);
389                 acked = ti->ti_ack - tp->snd_una;
390                 tcpstat.tcps_rcvackpack++;
391                 tcpstat.tcps_rcvackbyte += acked;
392                 sbdrop(&so->so_snd, acked);
393                 tp->snd_una = ti->ti_ack;
394                 m_freem(m);
395                 /*
396                  * If all outstanding data is acked, stop
397                  * retransmit timer, otherwise restart timer
398                  * using current (possibly backed-off) value.
399                  * If process is waiting for space,
400                  * wakeup/selwakeup/signal.  If data
401                  * is ready to send, let tcp_output
402                  * decide between more output or persist.
403                  */
404                 if (tp->snd_una == tp->snd_max)
405                     tp->t_timer[TCPT_REXMT] = 0;
```

图28-12 tcp_input 函数：首部预测，发送方处理

```
406                 else if (tp->t_timer[TCPT_PERSIST] == 0)
407                     tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;

408                 if (so->so_snd.sb_flags & SB_NOTIFY)
409                     sowwakeup(so);
410                 if (so->so_snd.sb_cc)
411                     (void) tcp_output(tp);
412                 return;
413             }
```

tcp_input.c

图28-12 (续)

3. 纯ACK测试

376-379 如果下列4个条件全真，则收到的是一个纯ACK报文段。

1) 报文段不携带数据(ti_len等于0)。

2) 报文段的确认字段(ti_ack)大于最大的未确认序号(snd_una)。由于测试条件是“大于”，而非“大于等于”，也就是要求收到的ACK必须确认未曾确认过的数据。

3) 报文段的确认字段(ti_ack)小于等于已发送的最大序号(snd_max)。

4) 拥塞窗口大于等于当前发送窗口(snd_wnd)，要求窗口完全打开，连接不处于慢起动或拥塞避免状态。

4. 更新RTT值

384-388 如果报文段携带时间戳选项，或者报文段中的确认序号大于某个计时报文段的起始序号，则调用tcp_xmit_timer更新RTT值。

5. 从发送缓存中删除被确认的字节

389-394 acked等于接收报文段确认字段所确认的字节数，调用sbdrop从发送缓存中删除这些字节。更新最大的未确认过的序号(snd_una)为报文段的确认字段值，释放保存接收报文段的mbuf链表(由于数据长度等于0，实际只有一个保存首部的mbuf)。

6. 终止重传定时器

395-407 如果接收报文段确认了所有已发送数据(snd_una等于snd_max)，则关闭重传定时器。若条件不满足，且持续定时器未设定，则重启重传定时器，时限设为t_rxtcur。

前面介绍过，tcp_output发送报文段时，只有重传定时器未启动，才会设定它。如果连续发送两个报文段，发送第一个报文段时定时器启动，发送第二个报文段时定时器不变。但如果只收到第一个报文段的确认，则重传定时器必须重启，防止第二个报文段丢失。

7. 唤醒等待进程

408-409 如果发送缓存修改后，有必要唤醒等待的应用进程，则调用sowwakeup。从图16-5可知，如果有应用进程正在等待缓存空间，或者设定了与缓存有关的select选项，或者正等待插口上的SIGIO，则SB_NOTIFY为真。

8. 生成更多的输出

410-411 如果发送缓存中有数据，则调用tcp_output，因为发送窗口已经向右移动。snd_una已增加，但snd_wnd未变化，因此，图24-17中的整个窗口将向右移动。

图28-13给出了首部预测的下一部分代码，接收方收到顺序到达的数据时进行的各种处理。

9. 测试收到报文段是否是连接等待接收的下一个报文段

414-416 如果下列4个条件均为真，则收到的报文段是连接上等待接收的下一报文段，并且插口缓存中的剩余空间能够容纳到达的数据。

1) 报文段的数据量(ti_len)大于0，即图28-12起始处if语句的else子句。

2) 确认字段(ti_ack)等于最大的未确认序号，即报文段未确认任何数据。

3) 连接乱序报文段的重组队列为空(seq_next等于tp)。

4) 接收缓存能够容纳报文段数据。

10. 完成接收数据的处理

423-435 等待接收序号(rcv_nxt)递增收到的数据字节数。从mbuf链中丢弃IP首部、TCP首部和所有TCP选项，将剩余的mbuf链附加到插口的接收缓存，调用sorwakeup唤醒接收进程。注意，代码没有调用TCP_REASS宏，因为宏代码中的条件判定已经包含在首部预测的测试条件中。设定延迟ACK标志，输入处理结束。

tcp_input.c

```
414         } else if (ti->ti_ack == tp->snd_una &&
415                    tp->seg_next == (struct tcpiphdr *) tp &&
416                    ti->ti_len <= sbspace(&so->so_rcv)) {
417             /*
418              * this is a pure, in-sequence data packet
419              * with nothing on the reassembly queue and
420              * we have enough buffer space to take it.
421              */
422             ++tcpstat.tcps_preddat;
423             tp->rcv_nxt += ti->ti_len;
424             tcpstat.tcps_rcvpack++;
425             tcpstat.tcps_rcvbyte += ti->ti_len;
426             /*
427              * Drop TCP, IP headers and TCP options then add data
428              * to socket buffer.
429              */
430             m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
431             m->m_len -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
432             sbappend(&so->so_rcv, m);
433             sorwakeup(so);
434             tp->t_flags |= TF_DELACK;
435             return;
436         }
437     }
```

tcp_input.c

图28-13 tcp_input 函数：首部预测的接收方处理

统计量

首部预测能在多大程度上改善系统性能？让我们做个简单的实验，跨越 LAN(bdsi和svr4间的双向通信)的数据传输，和跨越 WAN(vangogh.cs.berkeley.edu和ftp.uu.net之间的双向通信)的数据传输。运行netstat，得到类似于图24-5的输出，列出了两种情况下的首部预测寄存器的值。

跨越LAN传输时，无数据分组丢失，只有一些重复的 ACK。利用首部预测处理的报文段可占到97%~100%。跨越WAN时，比例有所降低，约为83%~99%之间。

请注意，首部预测的应用限定于单独的连接，无论主机是否收到了额外的 TCP流量，PCB

缓存必须在主机范围内共享。即使 TCP流量的丢失造成了 PCB缓存缺失，但如果给定连接上的数据分组未丢失，这条连接上的首部预测仍能工作。

28.5 TCP输入：缓慢的执行路径

下面讨论首部预测失败时的处理代码，`tcp_input`中较慢的一条执行路径。图 28-14给出了下一部分代码，为输入报文段的处理完成一些准备工作。

tcp_input.c

```
438     /*
439      * Drop TCP, IP headers and TCP options.
440      */
441     m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
442     m->m_len -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);

443     /*
444      * Calculate amount of space in receive window,
445      * and then do TCP input processing.
446      * Receive window is amount of space in rcv queue,
447      * but not less than advertised window.
448      */
449     {
450         int     win;

451         win = sbspace(&so->so_rcv);
452         if (win < 0)
453             win = 0;
454         tp->rcv_wnd = max(win, (int) (tp->rcv_adv - tp->rcv_nxt));
455     }
```

tcp_input.c

图28-14 tcp_input 函数：丢弃IP和TCP首部

1. 丢弃IP和TCP首部，包括TCP选项

438-442 更新数据指针和mbuf链表中的第一个mbuf的长度，以跳过IP首部、TCP首部和所有TCP选项。因为`off`等于TCP首部长度，包括TCP选项，因此，表达式中减去了标准 TCP首部的大小(20字节)。

2. 计算接收窗口

443-455 `win`等于插口接收缓存中可用的字节数，`rcv_adv`减去`rcv_nxt`等于当前通告的窗口大小，接收窗口等于上述两个值中较大的一个，这是为了保证接收窗口不小于当前通告窗口的大小。此外，如果最后一次窗口更新后，应用进程从插口接收缓存中取走了数据，`win`可能大于通告窗口，因此，TCP最多能够接收`win`字节的数据(即使对端不会发送超过通告窗口大小的数据)。

因为函数后面的代码必须确定通告窗口中能放入多少数据 (如果有)，所以现在必须计算通告窗口的大小。落在通告窗口之外的接收数据被丢弃：落在窗口左侧的数据是已接收并确认过的数据，落在窗口右侧的数据是暂不允许对端发送的数据。

28.6 完成被动打开或主动打开

如果连接状态等于LISTEN或者SYN_SENT，则执行本节给出的代码。连接处于这两个状态时，等待接收的报文段为SYN，任何其他报文段将被丢弃。

28.6.1 完成被动打开

连接状态等于LISTEN时，执行图28-15中的代码，其中变量tp和inp指向图28-7所创建的新的插口，而非服务器的监听插口。

tcp_input.c

```
456     switch (tp->t_state) {

457         /*
458          * If the state is LISTEN then ignore segment if it contains an RST.
459          * If the segment contains an ACK then it is bad and send an RST.
460          * If it does not contain a SYN then it is not interesting; drop it.
461          * Don't bother responding if the destination was a broadcast.
462          * Otherwise initialize tp->rcv_nxt, and tp->irs, select an initial
463          * tp->iss, and send a segment:
464          *     <SEQ=ISS><ACK=RCV_NXT><CTL=SYN,ACK>
465          * Also initialize tp->snd_nxt to tp->iss+1 and tp->snd_una to tp->iss
466          * Fill in remote peer address fields if not previously specified.
467          * Enter SYN_RECEIVED state, and process any other fields of this
468          * segment in this state.
469          */
470     case TCPS_LISTEN:{
471             struct mbuf *am;
472             struct sockaddr_in *sin;

473             if (tiflags & TH_RST)
474                 goto drop;
475             if (tiflags & TH_ACK)
476                 goto dropwithreset;
477             if ((tiflags & TH_SYN) == 0)
478                 goto drop;
```

tcp_input.c

图28-15 tcp_input 函数：检测监听插口上是否收到了SYN

1. 丢弃RST、ACK或非SYN

473-478 如果接收报文段中带有RST标志，则丢弃它。如果带有ACK，则丢弃它并发送RST作为响应(建立连接的最初的SYN报文段是少数几个不允许携带ACK的报文段之一)。如果未带有SYN，则丢弃它。case子句的后续代码处理连接处于LISTEN状态时收到了SYN的状况。新的连接状态等于SYN_RCVD。

图28-16给出了case语句接下来的代码。

tcp_input.c

```
479             /*
480              * RFC1122 4.2.3.10, p. 104: discard bcast/mcast SYN
481              * in_broadcast() should never return true on a received
482              * packet with M_BCAST not set.
483              */
484             if (m->m_flags & (M_BCAST | M_MCAST) ||
485                 IN_MULTICAST(ti->ti_dst.s_addr))
486                 goto drop;

487             am = m_get(M_DONTWAIT, MT_SONAME);  /* XXX */
488             if (am == NULL)
489                 goto drop;
490             am->m_len = sizeof(struct sockaddr_in);
```

图28-16 tcp_input 函数：处理监听插口上收到的SYN报文段

```
491             sin = mtod(am, struct sockaddr_in *);
492             sin->sin_family = AF_INET;
493             sin->sin_len = sizeof(*sin);
494             sin->sin_addr = ti->ti_src;
495             sin->sin_port = ti->ti_sport;
496             bzero((caddr_t) sin->sin_zero, sizeof(sin->sin_zero));
497             laddr = inp->inp_laddr;
498             if (inp->inp_laddr.s_addr == INADDR_ANY)
499                 inp->inp_laddr = ti->ti_dst;
500             if (in_pcbconnect(inp, am)) {
501                 inp->inp_laddr = laddr;
502                 (void) m_free(am);
503                 goto drop;
504             }
505             (void) m_free(am);
```

tcp_input.c

图28-16 (续)

2. 如果是广播报文段或多播报文段，则丢弃它

479-486 如果数据报被发送到广播地址或多播地址，则丢弃它， TCP只支持点到点的应用。前面介绍过，根据数据帧携带的目的硬件地址， ether_input置位M_BCAST和M_MCAST标志， IN_MULTICAST宏可判定IP地址是否为D类地址。

注释引用了in_broadcast，因为Net/1代码(它不支持多播)在此处调用了这个函数，以检测目的IP地址是否为广播地址。 Net/2中改为根据目的硬件地址，通过ether_input设定M_BCAST和M_MCAST标志。

Net/3只测试目的硬件地址是否为广播地址，而且不调用 in_broadcast测试目的IP地址是否为广播地址。它假定除非目的硬件地址是广播地址，否则，目的 IP地址绝不可能是广播地址，从而避免调用 in_broadcast。另外，如果Net/3真的收到了一个数据帧，其目的硬件地址为单播地址，而目的 IP地址为广播地址，将执行图 28-16中的代码处理此种报文段。

目的地址参数IN_MULTICAST需要被转换为主机字节序。

3. 为客户端的IP地址和端口号分配mbuf

487-496 分配一个mbuf，保存sockaddr_in结构，其中带有客户端的 IP地址和端口号。IP地址从IP首部的源地址字段中复制，端口号从 TCP首部的源端口号字段中复制。这个结构用于把服务器的PCB连到客户，之后mbuf被释放。

注释中的“ XXX ”，是因为获取mbuf的消耗等同于之后调用in_pcbconnect的消耗。不过此处的代码位于 tcp_input中较慢的一条执行路径。从图 24-5可知，不足%2的接收报文段的处理中会用到这段处理代码。

4. 设定PCB中的本地地址

497-499 laddr是绑定在插口上的本地地址。如果服务器没有为插口绑定一个确定地址 (正常情况下)，IP首部的目的地址将成为 PCB中的本地地址。注意，不管数据报是在哪个端口收到的，都将保存IP首部中的目的地址。

注意，laddr不会是通配地址，因为图 28-7中的代码已将收到报文段中的目的 IP

地址赋给了它。

5. 填充PCB中的对端地址

500-505 调用in_pcbconnect，把服务器的PCB与客户相连，填充PCB中的对端地址和对端端口号。之后，释放mbuf。

图28-17给出了函数下一部分的代码，结束case语句的处理。

tcp_input.c

```
506             tp->t_template = tcp_template(tp);
507             if (tp->t_template == 0) {
508                 tp = tcp_drop(tp, ENOBUFS);
509                 dropsocket = 0; /* socket is already gone */
510                 goto drop;
511             }
512             if (optp)
513                 tcp_dooptions(tp, optp, optlen, ti,
514                               &ts_present, &ts_val, &ts_ecr);
515             if (iss)
516                 tp->iss = iss;
517             else
518                 tp->iss = tcp_iss;
519             tcp_iss += TCP_ISSINCR / 2;
520             tp->irs = ti->ti_seq;
521             tcp_sendseqinit(tp);
522             tcp_rcvseqinit(tp);
523             tp->t_flags |= TF_ACKNOW;
524             tp->t_state = TCPS_SYN_RECEIVED;
525             tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;
526             dropsocket = 0;     /* committed to socket */
527             tcpstat.tcps_accepts++;
528             goto trimthenstep6;
529         }
```

tcp_input.c

图28-17 tcp_input 函数：完成LISTEN状态下收到SYN报文段的处理

6. 分配并初始化IP和TCP首部模板

506-511 调用tcp_template创建IP和TCP首部的模板。图28-7中调用sonewconn时，为新连接分配了PCB和TCP控制块，但未分配首部模板。

7. 处理所有的TCP选项

512-514 如果存在TCP选项，则调用tcp_dooptions进行处理。图28-8中曾调用过一次tcp_dooptions，但只处理非LISTEN状态时的TCP选项。现在，插口处于监听状态， PCB中的对端地址已填入(tcp_mss函数中会用到对端地址)，调用tcp_dooptions：获取到达对端的路由；查看对端主机是本地结点还是远端结点 (选择MSS时，需考虑到对端的网络ID和子网ID)。

8. 初始化ISS

515-519 通常情况下，初始发送序号复制自全局变量 tcp_iss，之后增加 64 000 (TCP_ISSINCR除以2)。如果局部变量iss非零，则使用iss取代tcp_iss，初始化连接的发送序号。

出现以下事件序列时，会用到iss：

• 服务器的IP地址为128.1.2.3，端口号为27。

- IP地址等于192.3.4.5的客户与前述服务器建立了连接，客户端口号等于 3000。服务器的插口对为{128.1.2.3, 27, 192.3.4.5, 3000}。
- 服务器主动关闭了连接，上述插口对的状态转移到 TIME_WAIT。连接处于这种状态时，最后收到的序号保存在TCP控制块中。假设序号等于100 000。
- 连接离开TIME_WAIT状态之前，收到来自于同一客户主机、同一端口号 (192.3.4.5, 3000)的新的SYN，TCP寻找处于TIME_WAIT状态的连接所对应的PCB，而不是监听服务器的PCB。假定新SYN报文段的序号等于200 000。
- 因为连接状态不等于LISTEN，所以将不执行刚讨论过的图28-17中的代码，而是执行图28-28中的代码。我们将看到，其中包含了下列处理逻辑：如果新 SYN报文段的序号(200 000)大于客户最后发来的序号(100 000)，那么：(1)局部变量iss等于100 000加上128 000；(2)处于TIME_WAIT状态的连接被完全关闭(PCB和TCP控制块被删除)；(3)控制跳转到findpcb(图28-5)。
- 寻找服务器监听插口的PCB(假定监听服务器还在运行)，执行本节中介绍的代码。图28-17中的代码将使用局部变量iss(现在等于228 000)初始化新连接的tcp_iss。

RFC 1122中定义的这种处理逻辑，允许同一个客户和服务器重用同样的插口连接对，只要服务器主动关闭原有连接。它也解释了为什么只要有进程调用 connect，全局变量tcp_iss就递增64 000(图30-4)：为了确保在某个客户不断地重建与同一个服务器的连接的情况下，即使前一次连接上没有传输数据，甚至 500 ms定时器都未超时(定时器超时处理代码会增加tcp_iss)，新建连接仍可以使用较大的ISS。

9. 初始化控制块中的序号变量

520-522 图28-17中，初始接收序号复制自SYN报文段中的序号字段(irs)。下面两个宏初始化了TCP控制块中的相关变量。

```
#define tcp_rcvseqinit(tp) \
     (tp)->rcv_adv = (tp)->rcv_nxt = (tp)->irs + 1

#define tcp_sendseqinit(tp) \
     (tp)->snd_una = (tp)->snd_nxt = (tp)->snd_max = (tp)->snd_up = \
          (tp)->iss
```

因为SYN占据一个序号，所以第一个宏表达式需加1。

10. 确认SYN并更新状态

523-525 因为对于SYN的确认必须立即发送，所以置位TF_ACKNOW标志。连接状态转移到SYN_RCVD，连接建立定时器设为75秒(TCPTV_KEEP_INIT)。因为TF_ACKNOW置位，函数结束时将调用tcp_output。从图24-16可知，此种tcp_outflags会导致发送携带SYN和ACK的报文段。

526-528 现在，TCP结束了从图28-7开始的新插口的创建，drop插口标志被清除。控制跳转到trimthenstep6处，完成SYN报文段的处理。前面介绍过，SYN报文段能够携带数据，尽管只有等连接进入ESTABLISHED状态后，数据才会被提交给应用程序。

28.6.2 完成主动打开

图28-18给出了连接进入SYN_SENT状态后，处理代码的第一部分。TCP等待接收SYN。

tcp_input.c

```
530         /*
531          * If the state is SYN_SENT:
532          *  if seg contains an ACK, but not for our SYN, drop the input.
533          *  if seg contains an RST, then drop the connection.
534          *  if seg does not contain SYN, then drop it.
535          * Otherwise this is an acceptable SYN segment
536          *  initialize tp->rcv_nxt and tp->irs
537          *  if seg contains ack then advance tp->snd_una
538          *  if SYN has been acked change to ESTABLISHED else SYN_RCVD state
539          *  arrange for segment to be acked (eventually)
540          *  continue processing rest of data/controls, beginning with URG
541          */
542     case TCPS_SYN_SENT:
543         if ((tiflags & TH_ACK) &&
544             (SEQ_LEQ(ti->ti_ack, tp->iss) ||
545              SEQ_GT(ti->ti_ack, tp->snd_max)))
546             goto dropwithreset;
547         if (tiflags & TH_RST) {
548             if (tiflags & TH_ACK)
549                 tp = tcp_drop(tp, ECONNREFUSED);
550             goto drop;
551         }
552         if ((tiflags & TH_SYN) == 0)
553             goto drop;
```

tcp_input.c

图28-18 tcp_input 函数：判定收到的SYN是否是所需的响应

1. 验证收到的ACK

530-546 当应用进程主动打开，TCP发送SYN时，从图30-4可知，连接的iss将等于全局变量tcp_iss，宏tcp_sendseqinit(前一节结尾给出了定义)被执行。假设ISS等于365，图28-19给出了tcp_output发送SYN后的发送序号变量。

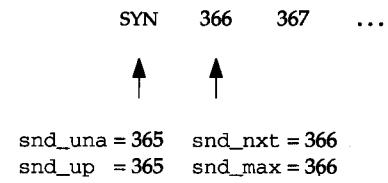


图28-19 ISS等于365的SYN发送后的发送序号变量

tcp_sendseqinit初始化图28-19中的4个变量为365，接着图26-31中的代码在发送SYN之后，把其中两个增至366。因此，如果图28-18中的接收报文段包含ACK，并且确认字段小于等于iss(365)，或者大于snd_max(366)，ACK无效，丢弃报文段并发送RST作为响应。注意，连接处于SYN_SENT状态时，收到的报文段中无需携带ACK。它可以只包括SYN，这种情况称为同时打开(simultaneous open)(图24-15)。

2. 处理并丢弃RST报文段

547-551 如果接收报文段中带有RST，则丢弃它。但首先应查看ACK标志，因为如果报文段同时携带了有效的ACK(已验证过)和RST，则说明对端拒绝本次连接请求，通常是因为服务器进程未运行。这种情况下，tcp_drop设定插口的so_error变量，并向调用connect的应用进程返回差错。

3. 判定SYN标志是否置位

552-553 如果收到报文段中的SYN标志未置位，则丢弃它。

这个case语句的其余代码用于处理本地发送连接请求后，收到对端响应的SYN报文段(及可选的ACK)的情况。图28-20给出了tcp_input下一部分的代码，继续处理SYN。

tcp_input.c

```
554         if (tiflags & TH_ACK) {
555             tp->snd_una = ti->ti_ack;
556             if (SEQ_LT(tp->snd_nxt, tp->snd_una))
557                 tp->snd_nxt = tp->snd_una;
558         }
559         tp->t_timer[TCPT_REXMT] = 0;
560         tp->irs = ti->ti_seq;
561         tcp_rcvseqinit(tp);
562         tp->t_flags |= TF_ACKNOW;
563         if (tiflags & TH_ACK && SEQ_GT(tp->snd_una, tp->iss)) {
564             tcpstat.tcps_connects++;
565             soisconnected(so);
566             tp->t_state = TCPS_ESTABLISHED;
567             /* Do window scaling on this connection? */
568             if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
569                 (TF_RCVD_SCALE | TF_REQ_SCALE)) {
570                 tp->snd_scale = tp->requested_s_scale;
571                 tp->rcv_scale = tp->request_r_scale;
572             }
573             (void) tcp_reass(tp, (struct tcpiphdr *) 0,
574                              (struct mbuf *) 0);
575             /*
576              * if we didn't have to retransmit the SYN,
577              * use its rtt as our initial srtt & rtt var.
578              */
579             if (tp->t_rtt)
580                 tcp_xmit_timer(tp, tp->t_rtt);
581         } else
582             tp->t_state = TCPS_SYN_RECEIVED;
```

tcp_input.c

图28-20 tcp_input 函数：发送连接请求后，收到对端响应的SYN

4. 处理ACK

554-558 如果报文段中有ACK，令snd_una等于报文段的确认字段。以图28-19为例，snd_una应更新为366，因为确认字段惟一有效的值就是366。如果snd_nxt小于snd_una(在图28-19的例子中不可能发生)，令snd_nxt等于snd_una。

5. 关闭连接建立定时器

559 连接建立定时器被关闭。

此处代码有错误。连接建立定时器只有在ACK标志置位时才能被关闭，因为收到一个不带ACK的SYN报文段，只是说明连接双方同时打开，而不意味着对端已收到了SYN。

6. 初始化接收序号

560-562 初始接收序号从接收报文段的序号字段中复制。tcp_rcvseqinit宏(上一节结束时给出了定义)初始化rcv_adv和rcv_nxt为接收序号加1。置位TF_ACKNOW标志，从而在函数结尾处调用tcp_output，发送报文段携带的确认字段应等于rcv_nxt(图26-27)，确认刚收到的SYN。

563-564 如果接收报文段带有ACK，并且snd_una大于连接的ISS，主动打开处理完毕，连接进入ESTABLISHED状态。

第二个测试条件其实是多余的。图28-20起始处，如果ACK标志置位，snd_una将等于接收报文段的确认字段值。另外，图28-18中紧跟着case语句的if语句验证了收到的确认字段大于ISS。所以，此处只要ACK置位，就可以确保snd_una大于ISS。

7. 连接建立

565-566 soisconnected设定插口进入连接状态，TCP连接的状态转移到ESTABLISHED。

8. 查看窗口大小选项

567-572 如果TCP在本地SYN中加入窗口大小选项，并且收到的SYN中也包含了这一选项，使用窗口缩放功能，设定snd_scale和rcv_scale。因为tcp_newtcpcb初始化TCP控制块为0，所以，如果不使用窗口大小选项，这两个变量的默认值为0。

9. 向应用进程提交队列中的数据

573-574 由于数据可能在连接未建立之前到达，调用tcp_reass把数据放入接收缓存，第二个参数为空。

测试条件其实不必要的。因为TCP刚收到带有ACK的SYN报文段，状态从SYN_SENT转移到ESTABLISHED。即使有数据出现在SYN中，也会被暂时搁置，直到函数快结束，控制转到dodata标注时才会被处理。如果TCP收到不带ACK的SYN(同时打开)，即使报文段携带数据，也会被暂时搁置，等到收到了ACK，连接从SYN_RCVD转移到ESTABLISHED之后，才会被处理。

尽管SYN中可以携带数据，并且Net/3能够正确处理这样的报文段，但Net/3自己不会产生这样的报文段。

10. 更新RTT估计器值

575-580 如果确认的SYN正被计时，tcp_xmit_timer将根据得到的对SYN报文段的测量值初始化RTT估计器值。

TCP在此处忽略收到的时间戳选项，只查看t_rtt计数器。TCP主动打开时，在第一个SYN中加入时间戳选项(图26-24)，如果对端也同意采用时间戳，就会在它响应的SYN中回应收到的时间戳(参见图28-10，Net/3在SYN中回应收到的时间戳)。因此，TCP在此处可以使用收到的时间戳，而不用t_rtt，但因为两者的精度相同(500ms)，在这一点上时间戳并无优势可言。使用时间戳，而非t_rtt计数器的真正好处在于高速网络中同时发送大量数据时，能提供更多的RTT测量值和更好的估计器值(希望如此)。

11. 同时打开

581-582 如果TCP在SYN_SENT状态收到不带ACK的SYN，则称为同时打开，连接转移到SYN_RCVD状态。

图28-21给出了函数的下一部分代码，处理SYN中可能携带的数据。图28-17结尾处，代码跳转至trimthenstep6标注处，这里也有类似的情况。

tcp_input.c

```
583       trimthenstep6:
584         /*
585          * Advance ti->ti_seq to correspond to first data byte.
586          * If data, trim to stay within window,
587          * dropping FIN if necessary.
588          */
```

图28-21 tcp_input 函数：接收SYN的通用处理

```
589             ti->ti_seq++;
590             if (ti->ti_len > tp->rcv_wnd) {
591                 todrop = ti->ti_len - tp->rcv_wnd;
592                 m_adj(m, -todrop);
593                 ti->ti_len = tp->rcv_wnd;
594                 tiflags &= ~TH_FIN;
595                 tcpstat.tcps_rcvpackafterwin++;
596                 tcpstat.tcps_rcvbyteafterwin += todrop;
597             }
598             tp->snd_wl1 = ti->ti_seq - 1;
599             tp->rcv_up = ti->ti_seq;
600             goto step6;
601         }
```

tcp_input.c

图28-21 (续)

584-589 报文段序号加1，以计入SYN。如果SYN带有数据，ti_seq现在应等于数据第一个字节的序号。

12. 丢弃落在接收窗口外的数据

590-597 ti_len等于报文段中的数据字节数。如果它大于接收窗口，超出部分的数据(ti_len减去rcv_wnd)将被m_adj丢弃。函数参数为负值，所以，将从mbuf链尾部起逆向删除数据(图2-20)。更新ti_len，等于数据删除后mbuf中剩余的数据量。清除FIN标志，这是因为FIN可能跟在最后一个数据字节之后，落在接收窗口外而被丢弃。

如果SYN是对本地连接请求的响应，且携带的数据过多，则说明对端收到的SYN报文段中带有窗口通告，但对端忽略了通告的窗口大小，并禁止不规范的行为。但如果主动打开的SYN报文段中带有大量数据，则说明对端还未收到窗口通告，所以不得不猜测SYN中能够携带多少数据。

13. 强制更新窗口变量

598-599 snd_wl1等于接收序号减1。从图29-15中可看到，这将强制更新3个窗口变量：snd_wnd、snd_wl1和snd_wl2。接收紧急指针(rcv_up)等于接收序号。控制跳转到标注step6处，与RFC 793定义的步骤相对应，我们将在图29-15中详细讨论。

28.7 PAWS：防止序号回绕

图28-22给出了tcp_input下一部分的代码，处理可能出现的序号回绕：RFC 1323中定义的PAWS算法。请回想一下我们在26.6节关于时间戳的讨论。

tcp_input.c

```
602     /*
603      * States other than LISTEN or SYN_SENT.
604      * First check timestamp, if present.
605      * Then check that at least some bytes of segment are within
606      * receive window.  If segment begins before rcv_nxt,
607      * drop leading data (and SYN); if nothing left, just ack.
608      *
609      * RFC 1323 PAWS: If we have a timestamp reply on this segment
610      * and it's less than ts_recent, drop it.
611      */
612     if (ts_present && (tiflags & TH_RST) == 0 && tp->ts_recent &&
```

图28-22 tcp_input 函数：处理时间戳选项

```
613         TSTMP_LT(ts_val, tp->ts_recent)) {

614         /* Check to see if ts_recent is over 24 days old.  */
615         if ((int) (tcp_now - tp->ts_recent_age) > TCP_PAWS_IDLE) {
616             /*
617              * Invalidate ts_recent.  If this segment updates
618              * ts_recent, the age will be reset later and ts_recent
619              * will get a valid value.  If it does not, setting
620              * ts_recent to zero will at least satisfy the
621              * requirement that zero be placed in the timestamp
622              * echo reply when ts_recent isn't valid.  The
623              * age isn't reset until we get a valid ts_recent
624              * because we don't want out-of-order segments to be
625              * dropped when ts_recent is old.
626              */
627             tp->ts_recent = 0;
628         } else {
629             tcpstat.tcps_rcvduppack++;
630             tcpstat.tcps_rcvdupbyte += ti->ti_len;
631             tcpstat.tcps_pawsdrop++;
632             goto dropafterack;
633         }
634     }
```

tcp_input.c

图28-22 (续)

1. 基本PAWS测试

602-613 如果存在时间戳，则调用tcp_dooptions设定ts_present。如果下列3个条件全真，则丢弃报文段：

1) RST标志未置位(参见习题28.8)。

2) TCP曾收到过对端发送的有效的时间戳(ts_recent非零)；并且

3) 当前报文段中的时间戳(ts_val)小于原先收到的时间戳。

PAWS算法基于这样的假定：对于高速连接， 32 bit时间戳值回绕的速度远小于32 bit序号回绕的速度。习题28.6说明，即使是最高的时钟计数器更新频率 (每毫秒加1)，时间戳的符号位也要24天才会回绕一次。而在千兆级网络中，序号可能 17秒就回绕一次(卷1的24.3节)。因此，如果报文段时间戳小于从同一个连接收到的最近一次的时间戳，说明是个重复报文段，应被丢弃(还需进行后续的时间戳过期测试)。尽管因为序号已过时， tcp_input也可将其丢弃，但PAWS算法能够有效地处理序号回绕速率很高的高速网。

注意， PAWS算法是对称的：它不仅丢弃重复的数据报文段，也丢弃重复的 ACK。PAWS处理所有收到的报文段，前面介绍过，首部预测代码也采用了 PAWS测试(图28-11)。

2. 检查过期时间戳

614-627 尽管可能性不大， PAWS测试还是会失败，因为连接有可能长时间空闲。收到的报文段并非重复报文段，但连接空闲时间过长，造成时间戳值回绕，从而小于从同一个连接收到的最近一次的时间戳。

无论何时， ts_recent保存接收报文段中的时间戳值， ts_recent_age记录当前时间(tcp_now)。如果ts_recent的最后一次更新发生在24天之前，则将其清零，不是一个有效的时间戳值。常量TCP_PAWS_IDLE定义为(24×24×60×60×2)，最后的乘数2指每秒钟2个滴答。这种情况下，不丢弃接收报文段，因为问题是时间戳过期，而非重复报文段。参见习

题28.6和28.7。

图28-23举例说明了时间戳过期问题。连接左侧的系统是一个非 Net/3的TCP实现，以RFC 1323中规定的最高速度，每毫秒更新一次时钟。连接右侧是 Net/3实现。

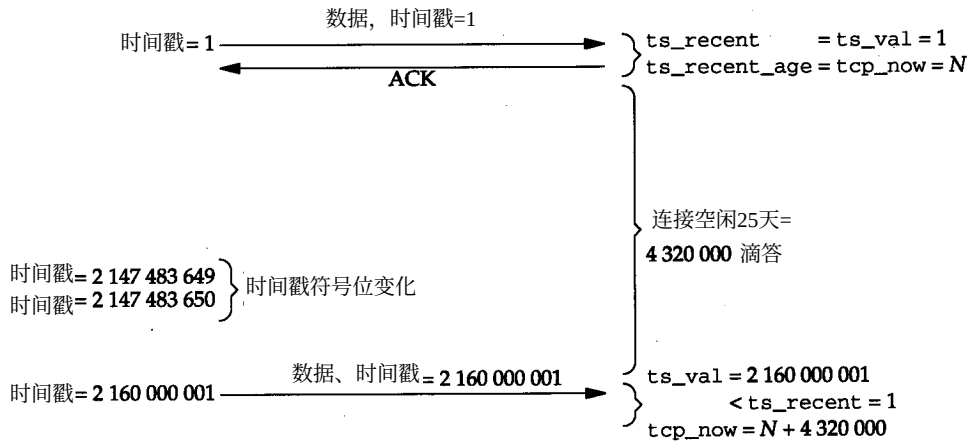


图28-23 过期时间戳举例

第一个数据报文段中携带的时间戳值等于1，所以ts_recent等于1，ts_recent_age等于当前时间(tcp_now)，如图28-11和图28-35所示。连接空闲25天，在此期间tcp_now增加了4 320 000 (25×24×60×60×1000)，对端的时间戳值增加了2 160 000 000 (25×24×60×60×1000)，时间戳值的符号位改变，即2 147 483 649大于1，而2 147 483 650小于1(回想图24-26)。因此，当收到的数据报文段中的时间戳等于 2 160 000 001时，调用TSTMP_LT宏进行比较，收到的时间戳小于ts_recent(1)，PAWS测试失败。但因为tcp_now减去ts_recent_age大于24天，说明造成失败的原因是连接空闲时间过长，报文段被接受。

3. 丢弃重复报文段

628-633 如果PAWS算法测试说明收到的是一个重复报文段，确认之后丢弃该报文段 (所有重复报文段都必须被确认)，不更新本地时间戳变量。

图24-5中，tcp_pawsdrop (22)远小于tcps_rcvduppack (46 953)。这可能是因为目前只有很少的系统支持时间戳，导致绝大多数重复报文段直到 TCP输出处理中才被发现和丢弃，而非PAWS。

28.8 裁剪报文段使数据在窗口内

本节讨论如何调整收到的报文段，确保它只携带能够放入接收窗口内的数据：

- 丢弃接收报文段起始处的重复数据；并且
- 从报文段尾部起，丢弃超出接收窗口的数据。

从而只剩下可放入接收窗口的新数据。图 28-24给出的代码，用于判定报文段起始处是否存在重复数据。

1. 查看报文段前部是否存在重复数据

635-636 如果接收报文段的起始序号 (ti_seq)小于等待接收的下一序号 (rcv_nxt)，则

todrop大于0，报文段前部有重复数据。这些数据已被确认并提交给应用进程 (图24-18)。

tcp_input.c

```
635     todrop = tp->rcv_nxt - ti->ti_seq;
636     if (todrop > 0) {
637         if (tiflags & TH_SYN) {
638             tiflags &= ~TH_SYN;
639             ti->ti_seq++;
640             if (ti->ti_urp > 1)
641                 ti->ti_urp--;
642             else
643                 tiflags &= ~TH_URG;
644             todrop--;
645         }
```

tcp_input.c

图28-24 tcp_input 函数：查看报文段起始处的重复数据

2. 丢弃重复SYN

637-645 如果SYN标志置位，它必然指向报文段的第一个数据序号，现已知是重复数据。清除SYN，报文段的起始序号加1，以越过重复的SYN。此外，如果接收报文段中的紧急指针大于1 (ti_urp)，则必须将其减1，因为紧急数据偏移量以报文段起始序号为基准。如果紧急指针等于0或者1，则不做处理，为防止出现等于1的情况，清除URG标志。最后，todrop减1(因为SYN占用一个序号)。

图28-25继续处理报文段前部的重复数据。

tcp_input.c

```
646         if (todrop >= ti->ti_len) {
647             tcpstat.tcps_rcvduppack++;
648             tcpstat.tcps_rcvdupbyte += ti->ti_len;
649             /*
650              * If segment is just one to the left of the window,
651              * check two special cases:
652              * 1. Don't toss RST in response to 4.2-style keepalive.
653              * 2. If the only thing to drop is a FIN, we can drop
654              *    it, but check the ACK or we will get into FIN
655              *    wars if our FINs crossed (both CLOSING).
656              * In either case, send ACK to resynchronize,
657              * but keep on processing for RST or ACK.
658              */
659             if ((tiflags & TH_FIN && todrop == ti->ti_len + 1)
660                 ) {
661                 todrop = ti->ti_len;
662                 tiflags &= ~TH_FIN;
663                 tp->t_flags |= TF_ACKNOW;
664             } else {
665                 /*
666                  * Handle the case when a bound socket connects
667                  * to itself. Allow packets with a SYN and
668                  * an ACK to continue with the processing.
669                  */
670                 if (todrop != 0 || (tiflags & TH_ACK) == 0)
671                     goto dropafterack;
672             }
673         } else {
674             tcpstat.tcps_rcvpartduppack++;
675             tcpstat.tcps_rcvpartdupbyte += todrop;
```

图28-25 tcp_input 函数：处理完全重复的报文段

```
676         }
677         m_adj(m, todrop);
678         ti->ti_seq += todrop;
679         ti->ti_len -= todrop;
680         if (ti->ti_urp > todrop)
681             ti->ti_urp -= todrop;
682         else {
683             tiflags &= ~TH_URG;
684             ti->ti_urp = 0;
685         }
686     }
```

tcp_input.c

图28-25 (续)

3. 判定报文段数据是否完全重复

646-648 如果报文段前部重复的数据字节数大于等于报文段大小，则是一个完全重复的报文段。

4. 判定重复FIN

649-663 接下来测试FIN是否重复，图28-26举例说明了这一情况。

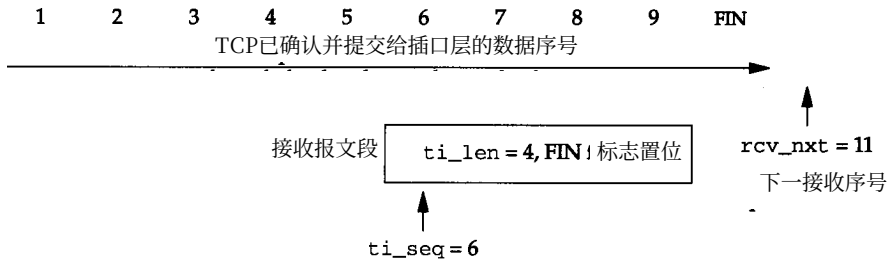


图28-26 举例：带有FIN标志的重复报文段

图28-26的例子中，todrop等于5，大于等于ti_len(4)。因为FIN置位，并且todrop等于ti_len加1，所以清除FIN标志，todrop重设为4，置位TF_ACKNOW，函数结束时立即发送ACK。这个例子也适用于其他报文段，如果ti_seq加上ti_len等于10。

代码的注释提到了4.2BSD实现中的保活定时器，Net/3省略了相关处理(if语句中的另一项测试)。

5. 生成重复ACK

664-672 如果todrop非零(报文段携带的全部是重复数据)，或者ACK标志未置位，则丢弃报文段，调用dropafterack生成ACK。出现这种情况，一般是因为对端未收到ACK，导致报文段重发。TCP生成新的ACK。

6. 处理同时打开或半连接

664-672 代码还处理同时打开，以及插口与自己建立连接的情况，将在下一节中详细讨论。如果todrop等于0(完全重复报文段中不包含数据)，且ACK标志置位，则继续下一步的处理。

if语句是4.4BSD版中新加的。早期的基于Berkeley的系统只是简单地跳转到dropafterack，即不处理同时打开，也不处理与自己建立连接的情况。

即使做了改进，这段代码仍有错误，我们在本节结束时将谈到这一点。

7. 收到部分重复报文段时，更新统计值

673-676 当todrop小于报文段长度时，执行else语句：报文段携带数据中只有部分重复。

8. 删除重复数据，更新紧急指针

677-685 调用m_adj，从mbuf链的首部开始删除重复数据，并相应地调整起始序号和长度。如果紧急指针指向的数据仍在 mbuf中，也需做相应的调整。否则，紧急指针清零，并清除URG标志。

图28-27给出了函数下一部分的代码，处理应用进程终止后到达的数据。

tcp_input.c

```
687     /*
688      * If new data is received on a connection after the
689      * user processes are gone, then RST the other end.
690      */
691     if ((so->so_state & SS_NOFDREF) &&
692         tp->t_state > TCPS_CLOSE_WAIT && ti->ti_len) {
693         tp = tcp_close(tp);
694         tcpstat.tcps_rcvafterclose++;
695         goto dropwithreset;
696     }
```

tcp_input.c

图28-27 tcp_input 函数：处理应用进程终止后到达的数据

687-696 如果找不到插口的描述符，说明应用进程已关闭了连接 (连接状态等于图24-16中大于CLOSE_WAIT的5个状态中的任何一个)，若接收报文段中有数据，则连接被关闭。报文段被丢弃，输出RST做为响应。

因为TCP支持半关闭功能，如果应用进程意外终止 (也许被某个信号量终止)，做为进程终止的一部分，内核将关闭所有打开的描述符， TCP将发送FIN。连接转移到FIN_WAIT_1状态。因为FIN的接收者无法知道对端执行的是完全关闭，还是半关闭。如果它假定是半关闭，并继续发送数据，那么将收到图28-27中发送的FIN。

图28-28给出了函数下一部分的代码，从接收报文段中删除落在通告窗口右侧的数据。

tcp_input.c

```
697     /*
698      * If segment ends after window, drop trailing data
699      * (and PUSH and FIN); if nothing left, just ACK.
700      */
701     todrop = (ti->ti_seq + ti->ti_len) - (tp->rcv_nxt + tp->rcv_wnd);
702     if (todrop > 0) {
703         tcpstat.tcps_rcvpackafterwin++;
704         if (todrop >= ti->ti_len) {
705             tcpstat.tcps_rcvbyteafterwin += ti->ti_len;
706             /*
707              * If a new connection request is received
708              * while in TIME_WAIT, drop the old connection
709              * and start over if the sequence numbers
710              * are above the previous ones.
711              */
712             if (tiflags & TH_SYN &&
713                 tp->t_state == TCPS_TIME_WAIT &&
```

图28-28 tcp_input 函数：删除落在窗口右侧的数据

```
714                 SEQ_GT(ti->ti_seq, tp->rcv_nxt)) {
715                 iss = tp->rcv_nxt + TCP_ISSINCR;
716                 tp = tcp_close(tp);
717                 goto findpcb;
718             }
719             /*
720              * If window is closed can only take segments at
721              * window edge, and have to drop data and PUSH from
722              * incoming segments.  Continue processing, but
723              * remember to ack.  Otherwise, drop segment
724              * and ack.
725              */
726             if (tp->rcv_wnd == 0 && ti->ti_seq == tp->rcv_nxt) {
727                 tp->t_flags |= TF_ACKNOW;
728                 tcpstat.tcps_rcvwinprobe++;
729             } else
730                 goto dropafterack;
731         } else
732             tcpstat.tcps_rcvbyteafterwin += todrop;
733         m_adj(m, -todrop);
734         ti->ti_len -= todrop;
735         tiflags &= ~(TH_PUSH | TH_FIN);
736     }
```

tcp_input.c

图28-28 (续)

9. 计算落在通告窗口右侧的字节数

697-703 todrop等于接收报文段中落在通告窗口右侧的字节数。例如，在图 28-29中，todrop等于(6+5)减去(4+6)，即等于1。

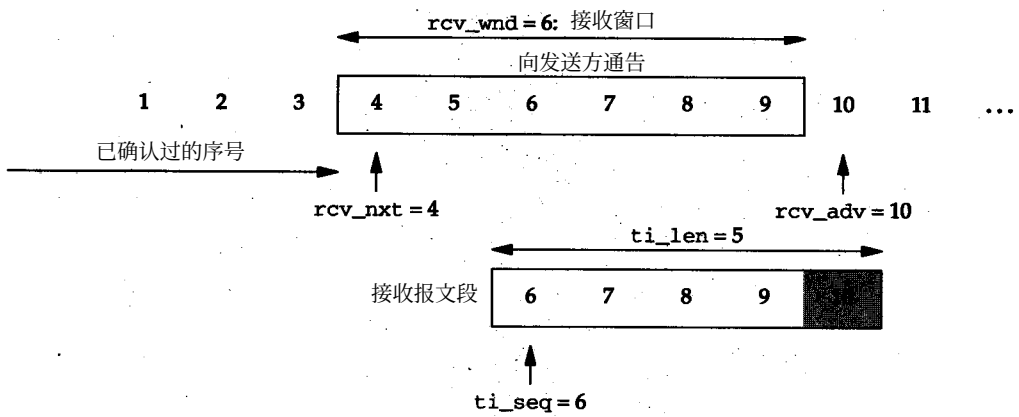


图28-29 举例：接收报文段部分数据落在窗口右侧

10. 如果连接处于TIME_WAIT状态，查看有无新的连接请求

704-718 如果todrop大于等于报文段长度，则丢弃整个报文段。如果下列3个条件全真：

1) SYN标志置位；并且

2) 连接处于TIME_WAIT状态；并且

3) 新的起始序号大于连接上最后收到的序号；

说明对端要求在已被关闭且正处于 TIME_WAIT状态的连接上重建连接。 RFC 1122允许这种情况，但要求新连接的ISS必须大于最后收到的序号(rcv_nxt)。TCP在rcv_nxt的基础上增加

128 000(TCP_ISSINCR)，得到执行图28-17中的代码时所使用的ISS。调用tcp_close释放处于TIME_WAIT状态的原有连接的PCB和TCP控制块。控制跳转到findpcb(图28-5)，寻找监听服务器的PCB(假定服务器仍在运行)。然后执行图28-7中的代码，为新连接创建新的插口，最后执行图28-16和图28-17中的代码，完成新连接请求的处理。

11. 判定是否为窗口探测报文段

719-728 如果接收窗口已关闭(rcv_wnd等于0)，且接收报文段中的数据从窗口最左端开始(rcv_nxt)，说明是对端发送的窗口探测报文段。TCP立即发送响应ACK，其中包含等待接收的序号。

12. 丢弃完全落在窗口之外的其他报文段

729-730 如果报文段整个落在窗口之外，且并非窗口探测报文段，则丢弃该报文段，并发送携带等待接收序号的ACK，作为响应。

13. 处理携带部分有效数据的报文段

731-735 通过m_adj，从mbuf链中删除落在窗口右侧的数据，并更新ti_len。如果接收报文段是对端发送的窗口探测报文段，m_adj将丢弃mbuf链中的所有数据，并将ti_len设为0，最后清除FIN和PSH标志。

何时丢弃ACK

图28-25中的代码有错误，在几种情况下，本应继续进行报文段处理，控制却跳转到dropafterack[Carlson 1993; Lanciani 1993]。系统实际运行时，如果连接双方重组队列中都存在缺失报文段，并都进入持续状态，将造成死锁，因为双方都将丢弃正常的ACK。

纠正的方法是简化图28-25起始处的代码。控制不再跳转到dropafterack，如果收到了完全重复报文段，则关闭FIN标志，并在函数结束时强迫立即发送ACK。删除图28-25中的646~676行的代码，而代之以图28-30中的代码。此外，新代码还更正了原代码中的另一个错误(习题28.9)。

```
if (todrop > ti->ti_len ||
    todrop == ti->ti_len && (tiflags & TH_FIN) == 0) {

    /*
     * Any valid FIN must be to the left of the window.
     * At this point the FIN must be a duplicate or
     * out of sequence; drop it.
     */
     tiflags &= ~TH_FIN;

    /*
     * Send an ACK to resynchronize and drop any data.
     * But keep on processing for RST or ACK.
     */
    tp->t_flags |= TF_ACKNOW;
    todrop = ti->ti_len;
    tcpstat.tcps_rcvdupbyte += todrop;
    tcpstat.tcps_rcvduppack++;

} else {
    tcpstat.tcps_rcvpartduppack++;
    tcpstat.tcps_rcvpartdupbyte += todrop;
}
```

图28-30 图28-28中646~676行代码的修正

28.9 自连接和同时打开

读者应首先理解插口与自己建立连接的步骤。接着会看到在 4.4BSD中，如何巧妙地通过一行代码修正图28-25中的错误，从而不仅能够处理自连接，还能处理 4.4BSD以前的版本中都无法正确处理的同时打开。

应用进程创建一个插口，并通过下列系统调用建立自连接： `socket`，`bind`绑定到一个本地端口(假定为3000)，之后`connect`试图与同一个本地地址和同一个端口号建立连接。如果`connect`成功，则插口已建立了与自己的连接：向这个插口写入的所有数据，都可以在同一插口上读出。这有点类似于全双工的管道，但只有一个，而非两个标识符。尽管很少有应用进程会这样做，但实际上它是一种特殊的同时打开，两者的状态变迁图相同。如果系统不允许插口建立自连接，那么它也很可能无法正确处理同时打开，而后者是 RFC 1122所要求的。有些人对于自连接能成功感到非常惊诧，因为只用了一个 Internet PCB和一个TCP控制块。不过，TCP是全双工的、对称的协议，它为每个方向上的数据流保留一份专有数据。

图28-31给出了应用进程调用 `connect`时的发送序号空间， SYN已发送，连接状态为SYN_SENT。

插口收到SYN后，执行图28-18和图28-20中的代码，但因为SYN中未包含ACK，连接状态转移到SYN_RCVD。从状态变迁图(图24-15)可知，与同时打开类似。图 28-32给出了接收序号空间。图28-20置位`TF_ACKNOW`，`tcp_output`生成的报文段将包含SYN和ACK(图24-16中的`tcp_outflags`)。SYN序号等于153，而确认序号等于154。

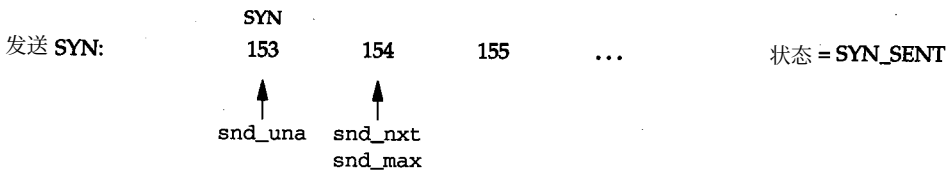


图28-31 自连接：SYN发送后的发送序号空间

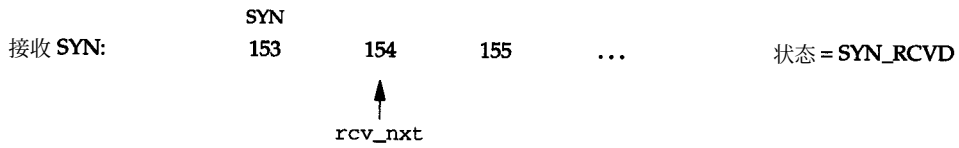


图28-32 自连接：收到的SYN处理完毕后的接收序号空间

与图28-20处理的正常情况相比，发送序号空间没有变化，只是连接状态等于 SYN_SENT。图28-33给出了收到同时带有SYN和ACK的报文段时，接收序号空间的状态。

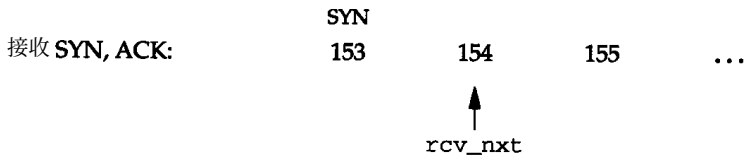


图28-33 收到带有SYN和ACK报文段时，接收序号空间的状态

因为连接状态等于SYN_RCVD，将执行图29-2中的代码处理收到的报文段，而不用我们在本章前面讨论过的处理主动打开或被动打开的代码。但在此之前，首先遇到的是图 28-24中的代码，而且从测试结果看似乎是一个重复 SYN：

```
todrop = rcv_nxt - rcv_seq
       = 154 - 153
       = 1
```

因为SYN标志置位，清除该标志，ti_seq等于154，todrop等于0。但因为todrop等于报文段长度(0)，图28-25开始处的测试条件为真，从而判定是一个重复报文段，执行注释为“处理绑定插口自连接的情况”的代码。早期的 TCP实现直接跳到 dropafterack，略过了SYN_RCVD状态的处理逻辑，不可能建立连接。相反，即使 todrop等于0，且ACK标志置位(本例中两个条件都成立)，Net/3仍旧继续处理收到的报文段，从而进入函数后面对SYN_RCVD状态的处理，连接转移到ESTABLISHED状态。

图28-34给出了自连接处理中函数调用的情况，是非常有意思的。

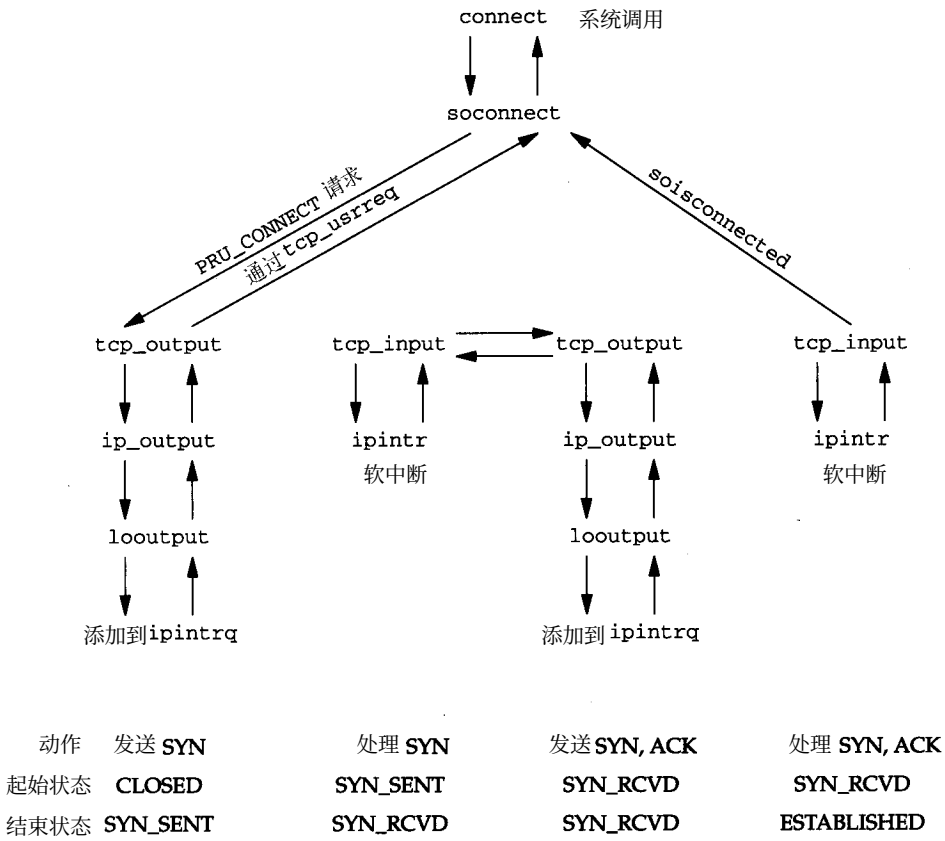


图28-34 自连接处理中的函数调用序列

操作顺序从左至右，首先应用进程调用 connect，发出PRU_CONNECT请求，经协议栈发送SYN。因为报文段发向主机自己的 IP地址，直接通过环回接口加入到 ipintrq，并生成一个软中断。

系统在软中断处理中调用 ipintr，ipintr调用tcp_input，tcp_input再调用

tcp_output，经协议栈发送带有ACK的SYN。这个报文段也经由环回接口加入到ipintrq，并生成一个软中断。系统调用 ipintr处理软中断， ipintr调用tcp_input，连接进入ESTABLISHED状态。

28.10 记录时间戳

图28-35给出了tcp_input下一部分的代码，处理收到的时间戳选项。

tcp_input.c

```
737     /*
738      * If last ACK falls within this segment's sequence numbers,
739      * record its timestamp.
740      */
741     if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
742         SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len +
743                ((tiflags & (TH_SYN | TH_FIN)) != 0))) {
744         tp->ts_recent_age = tcp_now;
745         tp->ts_recent = ts_val;
746     }
```

tcp_input.c

图28-35 tcp_input 函数：记录时间戳

737-746 如果收到的报文段中带有时间戳，时间戳值保存在 ts_recent中。我们在26.6节曾讨论过Net/3的处理代码有错误。如果FIN和SYN标志均未置位，表达式

```
((tiflags & (TH_SYN|TH_FIN)) != 0)
```

等于0；如果有一个置位，则等于1。

28.11 RST处理

图28-36给出了处理RST标志的switch语句，取决于当前的连接状态。

1. SYN_RCVD状态

759-761 插口差错代码设定为ECONNREFUSED，控制向前跳转若干行，关闭插口。在两种状况下，连接进入此状态。一般地讲，连接收到 SYN后，从LISTEN转移到SYN_RCVD状态。TCP发送带有ACK的SYN做为响应，但接着却收到了对端的 RST。此时， so引用的插口是在图28-7中调用sonewconn新创建的。因为dropsocket为真，在标注drop处，插口被丢弃，监听插口不受影响。这也是图 24-15中状态从SYN_RCVD转回LISTEN的原因。

另一种情况是，应用进程调用 connect后，出现同时打开，状态也转移到 SYN_RCVD。收到RST后，向应用进程返回插口差错。

tcp_input.c

```
747     /*
748      * If the RST bit is set examine the state:
749      *    SYN_RECEIVED state:
750      *  If passive open, return to LISTEN state.
751      *  If active open, inform user that connection was refused.
752      *    ESTABLISHED, FIN_WAIT_1, FIN_WAIT2, CLOSE_WAIT states:
753      *  Inform user that connection was reset, and close tcb.
754      *    CLOSING, LAST_ACK, TIME_WAIT states
755      *  Close the tcb.
756      */
```

图28-36 tcp_input 函数：处理RST标志

```
757     if (tiflags & TH_RST)
758         switch (tp->t_state) {

759         case TCPS_SYN_RECEIVED:
760             so->so_error = ECONNREFUSED;
761             goto close;

762         case TCPS_ESTABLISHED:
763         case TCPS_FIN_WAIT_1:
764         case TCPS_FIN_WAIT_2:
765         case TCPS_CLOSE_WAIT:
766             so->so_error = ECONNRESET;
767           close:
768             tp->t_state = TCPS_CLOSED;
769             tcpstat.tcps_drops++;
770             tp = tcp_close(tp);
771             goto drop;

772         case TCPS_CLOSING:
773         case TCPS_LAST_ACK:
774         case TCPS_TIME_WAIT:
775             tp = tcp_close(tp);
776             goto drop;
777         }
```

tcp_input.c

图28-36 (续)

2. 其他状态

762-777 如果在ESTABLISHED、FIN_WAIT_1、FIN_WAIT_2或CLOSE_WAIT状态收到RST，则返回差错代码ECONNRESET。如果状态为CLOSING、LAST_ACK或TIME_WAIT，由于应用进程已关闭插口，无需返回差错代码。

如果允许RST终止处于TIME_WAIT状态的连接，那么TIME_WAIT状态也就没有存在的必要。RFC 1337 [Braden 1992]讨论了这一点，及其他取消TIME_WAIT状态的可能状况，建议不允许RST永久终止处于TIME_WAIT状态的连接。参见习题28.10中的例子。

图28-37给出了函数下一部分的代码，验证SYN是否出错，ACK是否存在。

tcp_input.c

```
778     /*
779      * If a SYN is in the window, then this is an
780      * error and we send an RST and drop the connection.
781      */
782     if (tiflags & TH_SYN) {
783         tp = tcp_drop(tp, ECONNRESET);
784         goto dropwithreset;
785     }
786     /*
787      * If the ACK bit is off we drop the segment and return.
788      */
789     if ((tiflags & TH_ACK) == 0)
790         goto drop;
```

tcp_input.c

图28-37 tcp_input 函数：处理带有多余SYN或者缺少ACK的报文段

778-785 如果SYN标志依旧置位，说明出现了差错，连接被丢弃，返回差错代码ECONNRESET。

786-790 如果ACK标志未置位，则报文段被丢弃。我们将在下一章讨论函数剩余部分的代码，其中假定ACK标志均置位。

28.12 小结

本章详细介绍了TCP输入处理的前半部分，下一章将继续讨论函数剩余的部分。

本章介绍了如何验证报文段检验和，处理各种 TCP选项，处理发起和结束连接建立的SYN报文段，从报文段头尾两个方向删除无效数据，及处理 RST标志。

首部预测算法处理正常情况的数据流是非常有效的，执行速度最快。尽管我们讨论的多数处理逻辑用于覆盖所有可能发生的情况，但多数报文段都是正常的，只需很少的处理步骤。

习题

28.1 假定Net/3中插口缓存最大等于262 444，基于图28-7的算法，得到的窗口缩放因子是多少？

28.2 假定Net/3中插口缓存最大等于262 444，如果往返时间等于60ms，可能的最大吞吐量是多少？(提示：见卷1的图24-5及带宽的解)

28.3 为什么图28-10中，调用bcopy获取时间戳值？

28.4 我们在26.6节中提到，TCP要求的时间戳选项格式与RFC 1323附录A中定义的不同。尽管TCP能够正确处理时间戳，但由于采用与标准不同的格式，会付出什么代价？

28.5 处理PRU_ATTACH请求时会分配 PCB和TCP控制块，为什么不接着调用tcp_template分配首部模板？而是直至收到了SYN，在图28-17中才进行这一操作。

28.6 阅读RFC 1323，理解为什么图28-22中选取24天做为空闲时间的界限？

28.7 在图28-22中，如果连接空闲时间超过 24天，tcp_now-ts_recent_age与TCP_PAWS_IDLE的比较，会出现符号位回绕的问题。 Net/3中采取500ms做为时间戳单位，会在什么时间出现问题？

28.8 阅读RFC 1323，回答为什么图28-22中PAWS测试不包括RST报文段？

28.9 客户发送了SYN，服务器响应SYN/ACK。客户转移到ESTABLISHED状态，并发送响应ACK。但这个ACK丢失，服务器重发 SYN/ACK。描述一下客户收到重发的SYN/ACK时的处理步骤。

28.10 客户和服务器已建立了连接，服务器主动关闭。连接正常终止，服务器上的插口对转移到TIME_WAIT状态。在服务器的2MSL定时器超时前，同一客户 (客户端的同一个插口对)向服务器发送SYN，但起始序号小于连接上最后收到的序号。会发生什么？

第29章　TCP的输入(续)

29.1　引言

本章从前一章结束的地方开始，继续介绍 TCP输入处理。回想一下图 28-37中最后的测试条件，如果ACK未置位，输入报文段被丢弃。

本章处理ACK标志，更新窗口信息，处理 URG标志及报文段中携带的所有数据，最后处理FIN标志，如果需要，则调用`tcp_output`。

29.2　ACK处理概述

在本章中，我们首先讨论 ACK的处理，图29-1给出了ACK处理的框架。SYN_RCVD状态需要特殊处理，紧跟着是其他状态的通用处理代码 (前一章已讨论过在 LISTEN和SYN_SENT状态下收到 ACK时的处理逻辑)。接着是对 TCPS_FIN_WAIT_1、TCPS_CLOSING和TCPS_LAST_ACK状态的一些特殊处理，因为在这些状态下收到 ACK会导致状态的转移。此外，在TIME_WAIT状态下收到ACK还会导致2MSL定时器的重启。

```
switch (tp->t_state) {

case TCPS_SYN_RECEIVED:
    complete processing of passive open and process
        simultaneous open or self-connect;
    /* fall into ... */

case TCPS_ESTABLISHED:
case TCPS_FIN_WAIT_1:
case TCPS_FIN_WAIT_2:
case TCPS_CLOSE_WAIT:
case TCPS_CLOSING:
case TCPS_LAST_ACK:
case TCPS_TIME_WAIT:
    process duplicate ACK;
    update RTT estimators;
    if all outstanding data ACKed, turn off retransmission timer;
    remove ACKed data from socket send buffer;

    switch (tp->t_state) {

    case TCPS_FIN_WAIT_1:
        if (FIN is ACKed) {
            move to FIN_WAIT_2 state;
            start FIN_WAIT_2 timer;
        }
        break;

    case TCPS_CLOSING:
        if (FIN is ACKed) {
```

图29-1　ACK处理框架

```
                move to TIME_WAIT state;
                start TIME_WAIT timer;
            }
            break;

        case TCPS_LAST_ACK:
            if (FIN is ACKed)
                move to CLOSED state;
            break;

        case TCPS_TIME_WAIT:
            restart TIME_WAIT timer;
            goto dropafterack;
        }
    }
```

图29-1 (续)

29.3 完成被动打开和同时打开

图29-2给出了如何处理SYN_RCVD状态下收到的ACK报文段。如前一章中提到过的，这也将完成被动打开(一般情况)，或者是同时打开及自连接(特殊情况)的连接建立过程。

1. 验证收到的ACK

801-806 如果收到的ACK确认了已发送的SYN，它必须大于snd_una (tcp_sendseqinit将snd_una设定为连接的ISS，SYN报文段的序号)，且小于等于snd_max。如果条件满足，则插口进入连接状态ESTABLISHED。

tcp_input.c

```
791     /*
792      * Ack processing.
793      */
794     switch (tp->t_state) {

795         /*
796          * In SYN_RECEIVED state if the ack ACKs our SYN then enter
797          * ESTABLISHED state and continue processing, otherwise
798          * send an RST.
799          */
800     case TCPS_SYN_RECEIVED:
801         if (SEQ_GT(tp->snd_una, ti->ti_ack) ||
802             SEQ_GT(ti->ti_ack, tp->snd_max))
803             goto dropwithreset;
804         tcpstat.tcps_connects++;
805         soisconnected(so);
806         tp->t_state = TCPS_ESTABLISHED;
807         /* Do window scaling? */
808         if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
809             (TF_RCVD_SCALE | TF_REQ_SCALE)) {
810             tp->snd_scale = tp->requested_s_scale;
811             tp->rcv_scale = tp->request_r_scale;
812         }
813         (void) tcp_reass(tp, (struct tcpiphdr *) 0, (struct mbuf *) 0);
814         tp->snd_wl1 = ti->ti_seq - 1;
815         /* fall into ... */
```

tcp_input.c

图29-2 tcp_input 函数：在SYN_RCVD 状态收到ACK

在收到三次握手的最后一个报文段后，调用 soisconnected唤醒被动打开的应用进程(一般为服务器)。如果服务器在调用 accept上阻塞，则该调用现在返回。如果服务器调用select等待连接可读，则连接现在已经可读。

2. 查看窗口大小选项

807-812 如果TCP曾发送窗口大小选项，并且收到了对方的窗口大小选项，则在 TCP控制块中保存发送缩放因子和接收缩放因子。另外， TCP控制块中的snd_scale和rcv_scale的默认值为0 (无缩放)。

3. 向应用进程提交队列中的数据

813 现在可以向应用进程提交连接重组队列中的数据，调用 tcp_reass，第二个参数为空。重组队列中的数据可能是SYN报文段中携带的，它同时将连接状态变迁为 SYN_RCVD。

814 snd_wl1等于收到的序号减1，从图29-15可知，这样将导致更新3个窗口变量。

29.4 快速重传和快速恢复的算法

图29-3给出了ACK处理的下一部分代码，处理重复 ACK，并决定是否起用 TCP的快速重传和快速恢复算法[Jacobson 1990c]。两个算法各自独立，但一般都在一起实现 [Floyd 1994]。

- 快速重传算法用于连续出现几次 (一般为3次)重复ACK时，TCP认为某个报文段已丢失并且从中推断出丢失报文段的起始序号，丢失报文段被重传。 RFC 1122中的4.2.2.21节提到了这一算法，建议 TCP收到乱序报文段后，立即发送 ACK。我们看到，在图 27-15中，Net/3正是这样做的。这个算法最早出现在 4.3BSD Tahoe版及后续的Net/1实现中，丢失报文段被重传之后，连接执行慢起动。
- 快速恢复算法认为采用快速重传算法之后 (即丢失报文段已重传)，应执行拥塞避免算法，而非慢起动。这样，如果拥塞不严重，还能保证较大的吞吐量，尤其窗口较大时。这个算法最早出现在4.3BSD Reno版和后续的Net/2实现中。

Net/3同时实现了快速重传和快速恢复算法，下面将做简单介绍。

在图24-17节中，我们提到有效的ACK必须满足下面的不等式：

```
snd_una < 确认字段 <= snd_max
```

第一步只与snd_una做比较，之后在图 29-5中再进行不等式第二部分的比较。分开比较的原因是为了能对收到的ACK完成下列5项测试：

1) 如果确认字段小于等于snd_una；并且

2) 接收报文段长度为0；并且

3) 窗口通告大小未变；并且

4) 连接上部分发送数据未被确认 (重传定时器非零)；并且

5) 接收报文段的确认字段是TCP收到的最大的确认序号 (确认字段等于snd_una)。

之后可确认报文段是完全重复的 ACK(测试项1、2和3在图29-3中，测试4和5在图29-4的起始处)。

TCP统计连续收到的重复 ACK的个数，保存在变量 t_dupacks中，次数超过门限(tcprexmtthresh，3)时，丢失报文段被重传。这也就是卷 1第21.7节中介绍的快速重传算法。它与图27-15中的代码互相配合：当TCP收到乱序报文段时，立即生成一个重复的 ACK，

告诉对端报文段有可能丢失和等待接收的下一个序号值。快速重传算法是为了让 TCP立即重传看上去已经丢失的报文段，而不是被动地等待重传定时器超时。卷 1第21.7节举例详细说明了这个算法是如何工作的。

tcp_input.c

```
816         /*
817          * In ESTABLISHED state: drop duplicate ACKs; ACK out-of-range
818          * ACKs.  If the ack is in the range
819          *  tp->snd_una < ti->ti_ack <= tp->snd_max
820          * then advance tp->snd_una to ti->ti_ack and drop
821          * data from the retransmission queue.  If this ACK reflects
822          * more up-to-date window information we update our window information.
823          */
824     case TCPS_ESTABLISHED:
825     case TCPS_FIN_WAIT_1:
826     case TCPS_FIN_WAIT_2:
827     case TCPS_CLOSE_WAIT:
828     case TCPS_CLOSING:
829     case TCPS_LAST_ACK:
830     case TCPS_TIME_WAIT:

831         if (SEQ_LEQ(ti->ti_ack, tp->snd_una)) {
832             if (ti->ti_len == 0 && tiwin == tp->snd_wnd) {
833                 tcpstat.tcps_rcvdupack++;
834                 /*
835                  * If we have outstanding data (other than
836                  * a window probe), this is a completely
837                  * duplicate ack (ie, window info didn't
838                  * change), the ack is the biggest we've
839                  * seen and we've seen exactly our rexmt
840                  * threshold of them, assume a packet
841                  * has been dropped and retransmit it.
842                  * Kludge snd_nxt & the congestion
843                  * window so we send only this one
844                  * packet.
845                  *
846                  * We know we're losing at the current
847                  * window size so do congestion avoidance
848                  * (set ssthresh to half the current window
849                  * and pull our congestion window back to
850                  * the new ssthresh).
851                  *
852                  * Dup acks mean that packets have left the
853                  * network (they're now cached at the receiver)
854                  * so bump cwnd by the amount in the receiver
855                  * to keep a constant cwnd packets in the
856                  * network.
857                  */
```

tcp_input.c

图29-3 tcp_input 函数：判定完全重复的ACK报文段

另一方面，重复ACK的接收方也能确认某个数据分组已“离开了网络”，因为对端已收到了一个乱序报文段，从而开始发送重复的 ACK。快速恢复算法要求连续收到几个重复 ACK后，TCP应该执行拥塞避免算法(如降低速度)，而不一定必需等待连接两端间的管道清空 (慢起动)。“离开了网络”指数据分组已被对端接收，并加入到连接的重组队列中，不再滞留在传输途中。

如果前述5项测试条件只有前3项为真，说明ACK是重复报文段，统计值tcps_rcvdupack加1，而连续重复ACK计数器(t_dupacks)复位为0。如果仅有第一项测试条件为真，则计数器t_dupacks复位为0。

图29-4给出了快速重传算法其余的代码，当所有5个测试条件全部满足时，根据已连续收到的重复ACK数目的不同，运用快速重传算法处理收到的报文段。

1) t_dupacks等于3(tcprexmtthresh)，则执行拥塞避免算法，并重传丢失报文段。

2) t_dupacks大于3，则增大拥塞窗口，执行正常的TCP输出。

3) t_dupacks小于3，不做处理。

tcp_input.c

```
858                 if (tp->t_timer[TCPT_REXMT] == 0 ||
859                     ti->ti_ack != tp->snd_una)
860                     tp->t_dupacks = 0;
861                 else if (++tp->t_dupacks == tcprexmtthresh) {
862                     tcp_seq onxt = tp->snd_nxt;
863                     u_int win =
864                         min(tp->snd_wnd, tp->snd_cwnd) / 2 /
865                             tp->t_maxseg;
866                     if (win < 2)
867                         win = 2;
868                     tp->snd_ssthresh = win * tp->t_maxseg;
869                     tp->t_timer[TCPT_REXMT] = 0;
870                     tp->t_rtt = 0;
871                     tp->snd_nxt = ti->ti_ack;
872                     tp->snd_cwnd = tp->t_maxseg;
873                     (void) tcp_output(tp);
874                     tp->snd_cwnd = tp->snd_ssthresh +
875                         tp->t_maxseg * tp->t_dupacks;
876                     if (SEQ_GT(onxt, tp->snd_nxt))
877                         tp->snd_nxt = onxt;
878                     goto drop;
879                 } else if (tp->t_dupacks > tcprexmtthresh) {
880                     tp->snd_cwnd += tp->t_maxseg;
881                     (void) tcp_output(tp);
882                     goto drop;
883                 }
884             } else
885                 tp->t_dupacks = 0;
886             break;              /* beyond ACK processing (to step 6) */
887         }
```

tcp_input.c

图29-4 tcp_input 函数：处理重复的ACK

1. 连续收到的重复ACK次数已达到门限值3

861-868 t_dupacks等于3(tcprexmtthresh)时，在变量onxt中保存snd_nxt值，令慢起动门限(ssthresh)等于当前拥塞窗口大小的一半，最小值为两个最大报文段长度。这与图25-27中重传定时器超时处理中的慢起动门限设定操作类似，但我们将看到，超时处理中把拥塞窗口设定为一个最大报文段长度，快速重传算法并不这样做。

2. 关闭重传定时器

869-870 关闭重传定时器。为防止TCP正对某个报文段计时，t_rtt清零。

3. 重传缺失报文段

871-873 从连续收到的重复ACK报文段中可判断出丢失报文段的起始序号(重复ACK的确认字段)，将其赋给snd_nxt，并将拥塞窗口设定为一个最大报文段长度，从而tcp_output将只发送丢失报文段(参见卷1的图21-7中的63号报文段)。

4. 设定拥塞窗口

874-875 拥塞窗口等于慢起动门限加上对端高速缓存的报文段数。“高速缓存”指对端已收到的乱序报文段数，且为这些报文段发送了重复的ACK。除非对端收到了丢失的报文段(刚刚发送)，这些缓存报文段中的数据不会被提交给应用进程。卷1的图21-10和图21-11给出了快速重传算法起作用时,拥塞窗口和慢起动门限的变化情况。

5. 设定snd_nxt

876-878 比较下一发送序号(snd_nxt)的先前值(onxt)和当前值，将两者中最大的一个重新赋还给snd_nxt，因为重传报文段时，tcp_output会改变snd_nxt。一般情况下，snd_nxt将等于原来保存的值，意味着只有丢失报文段被重传，下一次调用tcp_output时，将继续发送序列中的下一报文段。

6. 连续收到的重复ACK数超过门限3

879-883 因为t_dupacks等于3时，已重传了丢失的报文段，再次收到重复ACK说明又有另一个报文段离开了网络。拥塞窗口大小加1，调用tcp_output发送序列中的下一报文段，并丢弃重复的ACK(参见卷1的图21-7的67号、69号和71号报文段)。

884-885 如果收到的报文段中带有重复的ACK，且长度非零或者通告窗口大小发生变化，则执行这些语句。此时，前面提到的5个测试条件中只有第一个为真，连续收到的重复ACK数被清零。

7. 略过ACK处理的其余部分

886 break语句在下列3种情况下被执行：(1)前述5个测试条件中只有第一个条件为真；(2)只有前3个条件为真；(3)重复ACK次数小于门限值3。任何一种情况下，尽管收到的是重复ACK，将执行break语句，控制跳到图29-2中switch语句的结尾处，在标注step6处继续执行。

为了理解前面的窗口操作步骤，请看下面的例子。假定对端接收窗口只能容纳8个报文段，而本地报文段1~8已发送。报文段1丢失，其余报文段均正常到达且被确认。收到对报文段2、3和4的确认后，重传丢失的报文段(1)。尽管在收到后续的对报文段5~8的确认后，TCP希望能够发送报文段9，以保证高的吞吐率。但窗口大小等于8，禁止发送报文段9及其后续报文段。因此，每当再次收到一个重复的ACK，就暂时把拥塞窗口加1，因为收到重复的ACK告诉TCP又有一个报文段已在对端离开了网络。最终收到对报文段1的确认后，下面将介绍的代码会减少拥塞窗口大小，令其等于慢起动门限。卷1的图21-10举例说明了这一过程，重复ACK到达时，增加拥塞窗口大小，之后收到新的ACK时，再相应地减少拥塞窗口。

29.5 ACK处理

图29-5中的代码继续处理ACK。

tcp_input.c

```
888         /*
889          * If the congestion window was inflated to account
890          * for the other side's cached packets, retract it.
891          */
892         if (tp->t_dupacks > tcprexmtthresh &&
893             tp->snd_cwnd > tp->snd_ssthresh)
894             tp->snd_cwnd = tp->snd_ssthresh;
895         tp->t_dupacks = 0;

896         if (SEQ_GT(ti->ti_ack, tp->snd_max)) {
897             tcpstat.tcps_rcvacktoomuch++;
898             goto dropafterack;
899         }
900         acked = ti->ti_ack - tp->snd_una;
901         tcpstat.tcps_rcvackpack++;
902         tcpstat.tcps_rcvackbyte += acked;
```

tcp_input.c

图29-5 tcp_input 函数：继续ACK处理

1. 调整拥塞窗口

888-895 如果连续收到的重复ACK数超过了门限值3，说明这是在收到了4个或4个以上的重复ACK后，收到的第一个非重复的ACK。快速重传算法结束。因为从收到的第4个重复ACK开始，每收到一个重复ACK就会导致拥塞窗口加1，如果它已超过了慢起动门限，令其等于慢起动门限。连续收到的重复ACK计数器清零。

2. 检查ACK的有效性

896-899 前面介绍过，有效的ACK必须满足下列不等式：

```
snd_una < 确认字段 <= snd_max
```

如果确认字段大于snd_max，可对端正在确认了TCP尚未发送的数据。可能的原因是，对于高速连接，某个失踪的ACK再次出现时，序号已回绕，从图24-5可知，这是极为罕见的(因为实际的网络不可能那么快)。

3. 计算确认的字节数

900-902 经过前面的测试，已知这是一个有效的ACK。acked等于确认的字节数。

图29-6给出了ACK处理的下一部分代码，完成RTT测算和重传定时器的操作。

4. 更新RTT测算值

903-915 如果(1)时间戳选项存在；或者(2)TCP对某个报文段计时，且收到的确认字段大于该报文段的起始序号，则调用tcp_xmit_timer更新RTT测算值。注意，使用时间戳时，tcp_xmit_timer的第二个参数等于当前时间(tcp_now)减去收到的时间戳回显(ts_ecr)加1(因为函数处理中减了1)。

由于延迟ACK的存在，在前面的测试不等式中应采用大于号。例如，假定TCP发送了一个报文段，携带字节1~1024，并对其计时，接着又发送了一个报文段，携带字节1025~2048。如果收到的确认字段等于2049，因为2049大于1(计时报文段的起始序号)，TCP将更新RTT测算值。

5. 是否确认了所有已发送数据

916-924 如果收到报文段的确认字段(ti_ack)等于TCP的最大发送序号(snd_max)，说明所有已发送数据都已被确认。关闭重传定时器，并置位needoutput标志，从而在函数结束

时强迫调用tcp_output。这是因为在此之前，有可能因为发送窗口已满， TCP拒绝了等待发送的数据，而现在收到了新的 ACK，确认了全部已发送数据，发送窗口能够向右移动 (图29-8中的snd_una被更新)，允许发送更多的数据。

tcp_input.c

```
903         /*
904          * If we have a timestamp reply, update smoothed
905          * round-trip time.  If no timestamp is present but
906          * transmit timer is running and timed sequence
907          * number was acked, update smoothed round-trip time.
908          * Since we now have an rtt measurement, cancel the
909          * timer backoff (cf., Phil Karn's retransmit alg.).
910          * Recompute the initial retransmit timer.
911          */
912         if (ts_present)
913             tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
914         else if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->t_rtseq))
915             tcp_xmit_timer(tp, tp->t_rtt);

916         /*
917          * If all outstanding data is acked, stop retransmit
918          * timer and remember to restart (more output or persist).
919          * If there is more data to be acked, restart retransmit
920          * timer, using current (possibly backed-off) value.
921          */
922         if (ti->ti_ack == tp->snd_max) {
923             tp->t_timer[TCPT_REXMT] = 0;
924             needoutput = 1;
925         } else if (tp->t_timer[TCPT_PERSIST] == 0)
926             tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
```

tcp_input.c

图29-6 tcp_input 函数：RTT测算值和重传定时器

6. 存在未确认的数据

925-926 由于发送缓存中还存在未被确认的数据，如果持续定时器未设定，则启动重传定时器，时限等于t_rxtcur的当前值。

Karn算法和时间戳

注意，时间戳的运用取消了 Karn算法的部分规定 (卷1的21.3节)：如果重传定时器超时，则报文段被重传，收到对重传报文段的确认时，不应据此更新 RTT测算值(重传确认的二义性问题)。在图25-26中，我们看到当发生重传时，遵从 Karn算法， t_rtt被设为0。如果时间戳不存在，且收到的是对重传报文段的确认，则图 29-6中的代码不会更新RTT测算值，因为此时t_rtt等于0。但如果时间戳存在，则不查看 t_rtt值，允许利用收到的时间戳回显字段更新RTT测算值。根据RFC 1323，时间戳的运用不存在二义性，因为 ts_ecr的值复制自被确认的报文段。Karn算法中关于重传报文段时应采用指数退避的策略依旧有效。

图29-7给出了ACK处理的下一部分代码，更新拥塞窗口。

tcp_input.c

```
927         /*
928          * When new data is acked, open the congestion window.
929          * If the window gives us less than ssthresh packets
```

图29-7 tcp_input 函数：响应收到的ACK，打开拥塞窗口

```
930             * in flight, open exponentially (maxseg per packet).
931             * Otherwise open linearly: maxseg per window
932             * (maxseg^2 / cwnd per packet), plus a constant
933             * fraction of a packet (maxseg/8) to help larger windows
934             * open quickly enough.
935             */
936            {
937                u_int    cw = tp->snd_cwnd;
938                u_int    incr = tp->t_maxseg;

939                if (cw > tp->snd_ssthresh)
940                    incr = incr * incr / cw + incr / 8;
941                tp->snd_cwnd = min(cw + incr, TCP_MAXWIN << tp->snd_scale);
942            }
```

tcp_input.c

图29-7 (续)

1. 更新拥塞窗口

927-942 慢起动和拥塞避免的一条原则是收到ACK后将增大拥塞窗口。默认情况下，每收到一个ACK(慢起动)，拥塞窗口将加1。但如果当前拥塞窗口大于慢起动门限，增加值等于1除以拥塞窗口大小，并加上一个常量。表达式

```
incr * incr / cw
```

等于

```
t_maxseg * t_maxseg / snd_cwnd
```

即1除以拥塞窗口，因为snd_cwnd的单位为字节，而非报文段。表达式的常量部分等于最大报文段长度的1/8。此外，拥塞窗口的上限等于连接发送窗口的最大值。算法的举例参见卷1的21.8节。

添加一个常量(最大报文段长度的1/8)是错误的[Floyd 1994]。但它一直存在于BSD源码中，从4.3BSD到4.4BSD和Net/3，应将其删除。

图29-8给出了tcp_input下一部分的代码，从发送缓存中删除已确认的数据。

tcp_input.c

```
943             if (acked > so->so_snd.sb_cc) {
944                 tp->snd_wnd -= so->so_snd.sb_cc;
945                 sbdrop(&so->so_snd, (int) so->so_snd.sb_cc);
946                 ourfinisacked = 1;
947             } else {
948                 sbdrop(&so->so_snd, acked);
949                 tp->snd_wnd -= acked;
950                 ourfinisacked = 0;
951             }
952             if (so->so_snd.sb_flags & SB_NOTIFY)
953                 sowwakeup(so);
954             tp->snd_una = ti->ti_ack;
955             if (SEQ_LT(tp->snd_nxt, tp->snd_una))
956                 tp->snd_nxt = tp->snd_una;
```

tcp_input.c

图29-8 tcp_input 函数：从发送缓存中删除已确认的数据

2. 从发送缓存中删除已确认的字节

943-946 如果确认字节数超过发送缓存中的字节数，则从snd_wnd中减去发送缓存中的字

节数，并且可知本地发送的FIN已被确认。调用sbdrop从发送缓存中删除所有字节。能够以这种方式检查对FIN报文段的确认，是因为FIN在序号空间中只占一个字节。

947-951 如果确认字节数小于或等于发送缓存中的字节数，ourfinisacked等于0，并从发送缓存中丢弃acked字节的数据。

3. 唤醒等待发送缓存的进程

951-956 调用sowwakeup唤醒所有等待发送缓存的应用进程，更新snd_una保存最老的未被确认的序号。如果snd_una的新值超过了snd_nxt，则更新后者，因为这说明中间的数据也被确认。

图29-9举例说明了为什么snd_nxt保存的序号有可能小于snd_una。假定传输了两个报文段，第一个携带字节1~512，而第二个携带字节513~1024。

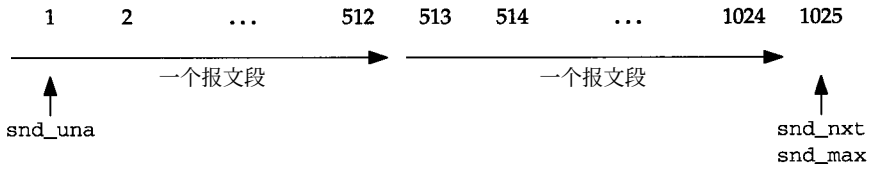


图29-9 连接上发送了两个报文段

确认返回前，重传定时器超时。图 25-26中的代码将snd_nxt设定为snd_una，进入慢起动状态，调用tcp_output重传携带1~512字节的报文段。tcp_output将snd_nxt增加为513，如图29-10所示。

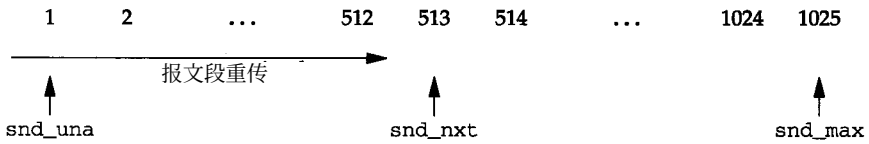


图29-10 重传定时器超时后的连接(接图29-9)

此时，确认字段等于1025的ACK到达(或者是最初发送的两个报文段或者是ACK在网络中被延迟)。这个ACK是有效的，因为它小于等于snd_max，但它也将小于更新后的snd_una值。

一般性的ACK处理现在已结束，图29-11中的switch语句接着处理了4种特殊情况。

tcp_input.c

```
957         switch (tp->t_state) {

958             /*
959              * In FIN_WAIT_1 state in addition to the processing
960              * for the ESTABLISHED state if our FIN is now acknowledged
961              * then enter FIN_WAIT_2.
962              */
963         case TCPS_FIN_WAIT_1:
964             if (ourfinisacked) {
965                 /*
966                  * If we can't receive any more
967                  * data, then closing user can proceed.
968                  * Starting the timer is contrary to the
```

图29-11 tcp_input 函数：在FIN_WAIT_1状态时收到了ACK

```
969                     * specification, but if we don't get a FIN
970                     * we'll hang forever.
971                     */
972                    if (so->so_state & SS_CANTRCVMORE) {
973                        soisdisconnected(so);
974                        tp->t_timer[TCPT_2MSL] = tcp_maxidle;
975                    }
976                    tp->t_state = TCPS_FIN_WAIT_2;
977                }
978                break;
```

tcp_input.c

图29-11 (续)

4. 在FIN_WAIT_1状态时收到了ACK

958-971 此时，应用进程已关闭了连接，TCP已发送了FIN，但还有可能收到对在FIN之前发送的报文段的确认。因此，只有在收到FIN的确认后，连接才会转移到FIN_WAIT_2状态。图29-8中，ourfinisacked标志已置位，这取决于确认的字节数是否超过发送缓存中的数据量。

5. 设定FIN_WAIT_2定时器

972-975 我们在25.6节中介绍了Net/3如何设定FIN_WAIT_2定时器，以防止在FIN_WAIT_2状态无限等待。只有当应用进程完全关闭了连接(如close系统调用，或者在应用进程被某个信号量终止时与close类似的内核调用)，而不是半关闭时(如已发送了FIN，但应用进程仍在连接上接收数据)，定时器才会启动。

图29-12给出了在CLOSING状态收到ACK时的处理代码。

tcp_input.c

```
979                /*
980                 * In CLOSING state in addition to the processing for
981                 * the ESTABLISHED state if the ACK acknowledges our FIN
982                 * then enter the TIME-WAIT state, otherwise ignore
983                 * the segment.
984                 */
985            case TCPS_CLOSING:
986                if (ourfinisacked) {
987                    tp->t_state = TCPS_TIME_WAIT;
988                    tcp_canceltimers(tp);
989                    tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
990                    soisdisconnected(so);
991                }
992                break;
```

tcp_input.c

图29-12 tcp_input 函数：在CLOSING状态收到ACK

6. 在CLOSING状态收到ACK

979-992 如果收到的ACK是对FIN的确认(而非之前发送的数据报文段)，则连接转移到TIME_WAIT状态。所有等待的定时器都被清除(如等待的重传定时器)，TIME_WAIT定时器被启动，时限等于两倍的MSL。

图29-13给出了在LAST_ACK状态收到ACK的处理代码。

tcp_input.c

```
 993             /*
 994              * In LAST_ACK, we may still be waiting for data to drain
 995              * and/or to be acked, as well as for the ack of our FIN.
 996              * If our FIN is now acknowledged, delete the TCB,
 997              * enter the closed state, and return.
 998              */
 999         case TCPS_LAST_ACK:
1000             if (ourfinisacked) {
1001                 tp = tcp_close(tp);
1002                 goto drop;
1003             }
1004             break;
```

tcp_input.c

图29-13 tcp_input 函数：在LAST_ACK状态收到ACK

7. 在LAST_ACK状态收到ACK

993-1004 如果FIN已确认，连接将转移到CLOSED状态。tcp_close将负责这一状态变迁，并同时释放Internet PCB和TCP控制块。

图29-14给出了在TIME_WAIT状态收到ACK的处理代码。

tcp_input.c

```
1005             /*
1006              * In TIME_WAIT state the only thing that should arrive
1007              * is a retransmission of the remote FIN.  Acknowledge
1008              * it and restart the finack timer.
1009              */
1010         case TCPS_TIME_WAIT:
1011             tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1012             goto dropafterack;
1013         }
1014     }
```

tcp_input.c

图29-14 tcp_input 函数：在TIME_WAIT状态收到ACK

8. 在TIME_WAIT状态收到ACK

1005-1014 此时，连接两端都已发送过FIN，且两个FIN都已被确认。但如果TCP对远端FIN的确认丢失，对端将重传FIN(带有ACK)。TCP丢弃报文段并重传ACK。此外，TIME_WAIT定时器必须被重传，时限等于两倍的MSL。

29.6 更新窗口信息

TCP控制块中还有两个窗口变量我们未曾提及：snd_wl1和snd_wl2。

- nd_wl1记录最后接收报文段的序号，用于更新发送窗口(snd_wnd)。
- snd_wl2记录最后接收报文段的确认序号，用于更新发送窗口。

到目前为止，只在连接建立时(主动打开、被动打开或同时打开)遇到过这两个变量，snd_wl1被设定为ti_seq减1。当时说是为了保证窗口更新，下面的代码将证明这一点。

如果下列3个条件中的任一个被满足，则应根据接收报文段中的通告窗口值(tiwin)更新发送窗口(snd_wnd)：

1) 报文段携带了新数据。因为snd_wl1保存了用于更新窗口的最后接收报文段的起始序

号，如果snd_wl1<ti_seq，说明此条件为真。

2) 报文段未携带新数据(snd_wl1等于ti_seq)，但报文段确认了新数据。因为snd_wl2保存了用于更新窗口的最后接收报文段的确认序号，如果 snd_wl2<ti_ack，说明此条件为真。

3) 报文段未携带新数据，也未确认新数据，但通告窗口大于当前发送窗口。

这些测试条件的目的是为了防止旧的报文段影响发送窗口，因为发送窗口并非绝对的序号序列，而是从snd_una算起的偏移量。

图29-15给出了更新发送窗口的代码。

tcp_input.c

```
1015    step6:
1016      /*
1017       * Update window information.
1018       * Don't look at window if no ACK: TAC's send garbage on first SYN.
1019       */
1020      if ((tiflags & TH_ACK) &&
1021          (SEQ_LT(tp->snd_wl1, ti->ti_seq) || tp->snd_wl1 == ti->ti_seq &&
1022           (SEQ_LT(tp->snd_wl2, ti->ti_ack) ||
1023            tp->snd_wl2 == ti->ti_ack && tiwin > tp->snd_wnd))) {
1024          /* keep track of pure window updates */
1025          if (ti->ti_len == 0 &&
1026              tp->snd_wl2 == ti->ti_ack && tiwin > tp->snd_wnd)
1027              tcpstat.tcps_rcvwinupd++;
1028          tp->snd_wnd = tiwin;
1029          tp->snd_wl1 = ti->ti_seq;
1030          tp->snd_wl2 = ti->ti_ack;
1031          if (tp->snd_wnd > tp->max_sndwnd)
1032              tp->max_sndwnd = tp->snd_wnd;
1033          needoutput = 1;
1034      }
```

tcp_input.c

图29-15 tcp_input 函数：更新窗口信息

1. 是否需要更新发送窗口

1015-1023 if语句检查报文段的ACK标志是否置位，且前述3个条件中是否有一个被满足。前面介绍过，在LISTEN状态或SYN_SENT状态收到SYN后，控制将跳转到 step6，而在LISTEN状态收到的SYN不带ACK。

注释中的TAC指“终端接入控制器(terminal access controller)”，是ARPANET上的Telnet客户。

1024-1027 如果收到一个纯窗口更新报文段(长度为0，ACK未确认新数据，但通告窗口增加)，统计值tcps_rcvwinupd递增。

2. 更新变量

1028-1033 更新发送窗口，保存新的snd_wl1和snd_wl2值。此外，如果新的通告窗口是TCP从对端收到的所有窗口通告中的最大值，则新值被保存在 max_sndwnd中。这是为了猜测对端接收缓存的大小，在图 26-8中用到了此变量。更新 snd_wnd后，发送窗口可用空间增加，从而能够发送新的报文段，因此， needoutput标志置位。

29.7 紧急方式处理

TCP输入处理的下一部分是URG标志置位时的报文段。如图29-16所示。

tcp_input.c

```
1035     /*
1036      * Process segments with URG.
1037      */
1038     if ((tiflags & TH_URG) && ti->ti_urp &&
1039         TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1040         /*
1041          * This is a kludge, but if we receive and accept
1042          * random urgent pointers, we'll crash in
1043          * soreceive.  It's hard to imagine someone
1044          * actually wanting to send this much urgent data.
1045          */
1046         if (ti->ti_urp + so->so_rcv.sb_cc > sb_max) {
1047             ti->ti_urp = 0;         /* XXX */
1048             tiflags &= ~TH_URG;     /* XXX */
1049             goto dodata;            /* XXX */
1050         }
```

tcp_input.c

图29-16 tcp_input 函数：紧急方式的处理

1. 是否需要处理URG标志

1035-1039 只有满足下列条件的报文段才会被处理： URG标志置位，紧急数据偏移量(ti_urp)非零，连接还未收到 FIN。只有当连接的状态等于 TIME_WAIT时，宏TCPS_HAVERCVDFIN才会为真，因此，连接处于任何其他状态时， URG都会被处理。在后面的注释中提到，连接处于CLOSE_WAIT、CLOSING、LAST_ACK和TIME_WAIT等几个状态时，URG标志会被忽略，这种说法是错误的。

2. 忽略超出的紧急指针

1040-1050 如果紧急数据偏移量加上接收缓存中已有的数据超过了插口缓存可容纳的数据量，则忽略紧急标志。紧急数据偏移量被清零， URG标志被清除，剩余的紧急方式处理逻辑被忽略。

图29-17给出了tcp_input下一部分的代码，处理紧急指针。

tcp_input.c

```
1051         /*
1052          * If this segment advances the known urgent pointer,
1053          * then mark the data stream.  This should not happen
1054          * in CLOSE_WAIT, CLOSING, LAST_ACK or TIME_WAIT states since
1055          * a FIN has been received from the remote side.
1056          * In these states we ignore the URG.
1057          *
1058          * According to RFC961 (Assigned Protocols),
1059          * the urgent pointer points to the last octet
1060          * of urgent data.  We continue, however,
1061          * to consider it to indicate the first octet
1062          * of data past the urgent section as the original
1063          * spec states (in one of two places).
1064          */
1065         if (SEQ_GT(ti->ti_seq + ti->ti_urp, tp->rcv_up)) {
```

图29-17 tcp_input 函数：处理收到的紧急指针

```
1066                tp->rcv_up = ti->ti_seq + ti->ti_urp;
1067                so->so_oobmark = so->so_rcv.sb_cc +
1068                    (tp->rcv_up - tp->rcv_nxt) - 1;
1069                if (so->so_oobmark == 0)
1070                    so->so_state |= SS_RCVATMARK;
1071                sohasoutofband(so);
1072                tp->t_oobflags &= ~(TCPOOB_HAVEDATA | TCPOOB_HADDATA);
1073            }
1074            /*
1075             * Remove out-of-band data so doesn't get presented to user.
1076             * This can happen independent of advancing the URG pointer,
1077             * but if two URG's are pending at once, some out-of-band
1078             * data may creep in... ick.
1079             */
1080            if (ti->ti_urp <= ti->ti_len
1081 #ifdef SO_OOBINLINE
1082                && (so->so_options & SO_OOBINLINE) == 0
1083 #endif
1084                )
1085                tcp_pulloutofband(so, ti, m);
1086        } else {
1087            /*
1088             * If no out-of-band data is expected, pull receive
1089             * urgent pointer along with the receive window.
1090             */
1091            if (SEQ_GT(tp->rcv_nxt, tp->rcv_up))
1092                tp->rcv_up = tp->rcv_nxt;
1093        }
```

tcp_input.c

图29-17 (续)

1051-1065 如果接收报文段的起始序号加上紧急数据偏移量超过了当前接收紧急指针，说明已收到了一个新的紧急指针。例如，图 26-30中的携带3字节的报文段到达接收方，如图 29-18所示。

一般情况下，收到的紧急指针(rcv_up)等于rcv_nxt。这个例子中，因为if语句为真(4加3大于4)，rcv_up的新值等于7。

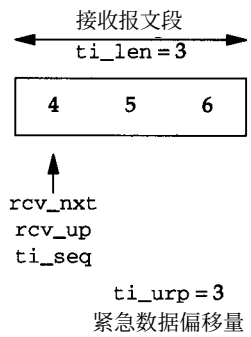


图29-18 图26-30中发送的报文段到达接收方

3. 计算收到的紧急指针

1066-1070 计算插口接收缓存中带外数据的分界点，应计入接收缓存中已有的数据(so_rcv.sb_cc)。在上面的例子中，假定接收缓存为空，so_oobmark等于2：序号为6的字节被认为是带外数据。如果这个带外数据标记等于0，说明插口正处在带外数据分界点上。如果发送带外数据的send系统调用给定长度为1，并且这个报文段到达对端时接收缓存为空，就会发生这一现象，同时也再次重申了 Berkeley系统认为紧急指针应指向带外数据后的第一字节。

4. 向应用进程通告TCP的紧急方式

1071-1072 调用sohasoutofband告知应用进程有带外数据到达了插口，清除两个标志

TCPOOB_HAVEDATA和TCPOOB_HADDATA，它们用于图30-8中的PRU_RCVOOB请求处理。

5. 从正常的数据流中提取带外数据

1074-1085 如果紧急数据偏移量小于等于接收报文段中的字节数，说明带外数据包含在报文段中。TCP的紧急方式允许紧急数据偏移量指向尚未收到的数据。如果定义了SO_OOBINLINE常量(正常情况下，Net/3定义了此常量)，而且未选用对应的插口选项，则接收进程将从正常的数据流中提取带外数据，并保存在t_iobc变量中。完成这一功能的函数，是我们将在下一节介绍的tcp_pulloutofband。

注意，无论紧急指针指向的字节是否可读，TCP都将通知接收进程发送方已进入紧急方式。这是TCP紧急方式的一个特性。

6. 如果不处于紧急方式，调整接收紧急指针

1086-1093 在接收方未处理紧急指针时，如果rcv_nxt大于接收紧急指针，则rcv_up向右移动，并等于rcv_nxt。这使接收紧急指针一直指向接收窗口的左侧，确保在收到URG标志时，图29-17起始处的宏SEQ_GT能够得出正确的结果。

如果要实现习题26.6中提出的方案，也必须相应修改图29-16和图29-17中的代码。

29.8 tcp_pulloutofband函数

图29-17中的代码调用了这个函数，如果：

1) 接收报文段中带有紧急方式标志；并且

2) 带外数据包含在接收报文段中(如，紧急指针指向接收报文段)；并且

3) 未选用SO_OOBINLINE选项。

函数从正常的数据流(保存接收报文段的mbuf链)中提取带外字节，并保存在连接TCP控制块中的t_iobc变量中。应用进程通过recv系统调用，置位MSG_OOB标志，读取这个变量：图30-8中的PRU_RCVOOB请求。图29-19给出了函数代码。

tcp_input.c

```
1282 void
1283 tcp_pulloutofband(so, ti, m)
1284 struct socket *so;
1285 struct tcpiphdr *ti;
1286 struct mbuf *m;
1287 {
1288     int     cnt = ti->ti_urp - 1;

1289     while (cnt >= 0) {
1290         if (m->m_len > cnt) {
1291             char    *cp = mtod(m, caddr_t) + cnt;
1292             struct tcpcb *tp = sototcpcb(so);

1293             tp->t_iobc = *cp;
1294             tp->t_oobflags |= TCPOOB_HAVEDATA;
1295             bcopy(cp + 1, cp, (unsigned) (m->m_len - cnt - 1));
1296             m->m_len--;
1297             return;
1298         }
1299         cnt -= m->m_len;
```

图29-19 tcp_pulloutofband 函数：将带外数据保存在t_iobc 变量中

```
1300            m = m->m_next;
1301            if (m == 0)
1302                break;
1303        }
1304        panic("tcp_pulloutofband");
1305    }
```

tcp_input.c

图29-19 (续)

1282-1289 考虑图29-20中的例子。紧急数据偏移量等于3，因此紧急指针等于7，带外字节的序号等于6。接收报文段携带了5字节的数据，全部保存在一个mbuf中。

变量cnt等于2，因为m_len(等于5)大于2，执行if语句为真部分的代码。

1290-1298 cp指向序号为6的字节，它被放入保存带外字节的变量t_iobc中。置位TCPOOB_HAVEDATA标志，调用bcopy将接下来的两个字节(序号7和8)左移1字节，如图29-21所示。

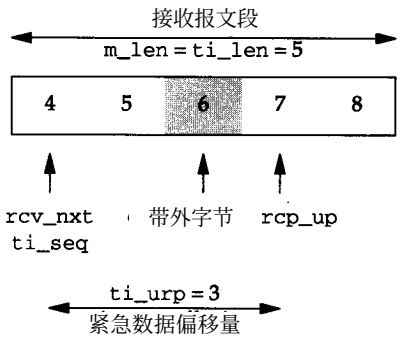


图29-20 携带带外字节的报文段

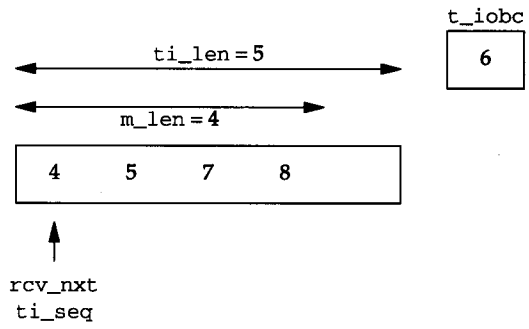


图29-21 移走带外数据后的结果(接图29-20)

注意，数字7和8指数据字节的序号，而不是其内容。 mbuf的长度从5减为4，但ti_len仍等于5不变，这是为了按序把报文段放入插口的接收缓存。 TCP_REASS宏和tcp_reass函数(在下一节调用)都会给rcv_nxt增加ti_len，本例中ti_len必须等于5，因为下一个等待接收的序号等于 9。还请注意，函数没有对第一个 mbuf中的数据分组首部长度(m_pkthdr.len)减1，这是因为负责把数据添加到插口接收缓存的 sbappend不使用此长度值。

跳至链中的下一个mbuf

1299-1302 如果带外数据未保存在此mbuf中，则从cnt中减去mbuf中的字节数，处理链中的下一个mbuf。因为只有当紧急数据移量指向接收报文段时，才会调用此函数，所以，如果链已结束，不存在下一个mbuf，则执行break语句，跳转到标注panic处。

29.9 处理已接收的数据

tcp_input接着提取收到的数据(如果存在)，将其添加到插口接收缓存，或者放入插口的乱序重组队列中。图29-22给出了完成此项功能的代码。

tcp_input.c

```
1094    dodata:                         /* XXX */
1095      /*
1096       * Process the segment text, merging it into the TCP sequencing queue,
1097       * and arranging for acknowledgment of receipt if necessary.
1098       * This process logically involves adjusting tp->rcv_wnd as data
1099       * is presented to the user (this happens in tcp_usrreq.c,
1100       * case PRU_RCVD).  If a FIN has already been received on this
1101       * connection then we just ignore the text.
1102       */
1103      if ((ti->ti_len || (tiflags & TH_FIN)) &&
1104          TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1105          TCP_REASS(tp, ti, m, so, tiflags);
1106          /*
1107           * Note the amount of data that peer has sent into
1108           * our window, in order to estimate the sender's
1109           * buffer size.
1110           */
1111          len = so->so_rcv.sb_hiwat - (tp->rcv_adv - tp->rcv_nxt);
1112      } else {
1113          m_freem(m);
1114          tiflags &= ~TH_FIN;
1115      }
```

tcp_input.c

图29-22 tcp_input 函数：把收到的数据放入插口接收队列

1094-1105 报文段数据将被处理，如果：

1) 接收数据的长度大于0，或者FIN标志置位；并且

2) 连接还未收到FIN；

则调用宏TCP_REASS处理数据。如果数据次序正确(如，连接等待接收的下一序号)，置位延迟ACK标志，增加rcv_nxt，并把数据添加到插口的接收缓存中。如果数据次序错误，宏会调用tcp_reass函数，把数据加入到连接的重组队列中(新到数据有可能填充队列中的缺口，从而将已排队的数据添加到插口的接收缓存中)。

前面介绍过，宏的最后一个参数(tiflags)是可修改的。特别地，如果数据次序错误，tcp_reass令tiflags等于0，清除FIN标志(如果它已置位)。这也就是为什么即使报文段中没有数据，只要FIN置位，if语句也为真。

考虑下面的例子。连接建立后，发送方立即发送报文段：一个携带字节 1~1024，另一个携带字节1025~2048，还有一个未带数据的FIN。第一个报文段丢失，因此，第二个报文段到达时(字节1025~2048)，接收方将其放入乱序重组队列，并立即发送 ACK。当第三个带有FIN标志的报文段到达时，图29-22中的代码被执行。即使数据长度等于0，因为FIN置位，导致调用TCP_REASS，它接着调用tcp_reass。因为ti_seq(2049，FIN的序号)不等于rcv_nxt(1)，tcp_reass返回0(图27-23)。在TCP_REASS宏中，tiflags被设为0，从而清除了FIN标志，阻止后续代码(图29-10)继续处理FIN。

猜测对端发送缓存大小

1106-1111 计算len，实际上是在猜测对端发送缓存的大小。考虑下面的例子。插口接收缓存大小等于8192(Net/3的默认值)，因此，TCP在SYN中通告窗口大小为8192。之后收到第一个报文段，携带字节1~1024。图29-23给出了在TCP_REASS增加rcv_nxt以反应收到的数据后接收空间的状态。

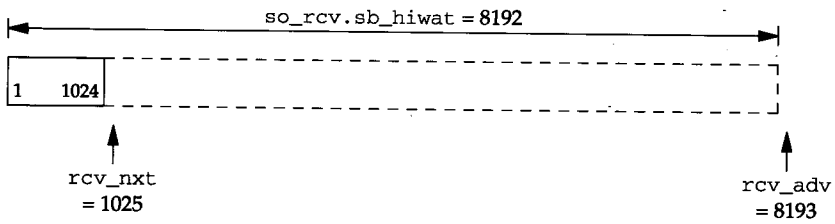


图29-23 大小为8192的接收窗口收到字节1~1024后的状态

此时，经计算，len等于1024。对端向接收窗口发送更多数据后，len值将增加，但绝不会超过对端发送缓存的大小。前面介绍过，图29-15中对变量max_sndwnd的计算，是在猜测对端接收缓存的大小。

事实上，变量len从未被使用。它是从Net/1遗留下来的，len计算后被存储到TCP控制块的max_rcvd变量中：

```
if (len > tp->max_rcvd)
   tp->max_rcvd = len;
```

但即使在Net/1中，变量max_rcvd也未被使用。

1112-1115 如果len等于0，且FIN标志未置位，或者连接上已收到了FIN，则丢弃保存接收报文段的mbuf链，并清除FIN。

29.10 FIN处理

tcp_input的下一步，在图29-24中给出，处理FIN标志。

tcp_input.c

```
1116     /*
1117      * If FIN is received ACK the FIN and let the user know
1118      * that the connection is closing.
1119      */
1120     if (tiflags & TH_FIN) {
1121         if (TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1122             socantrcvmore(so);
1123             tp->t_flags |= TF_ACKNOW;
1124             tp->rcv_nxt++;
1125         }
1126         switch (tp->t_state) {

1127             /*
1128              * In SYN_RECEIVED and ESTABLISHED states
1129              * enter the CLOSE_WAIT state.
1130              */
1131         case TCPS_SYN_RECEIVED:
1132         case TCPS_ESTABLISHED:
1133             tp->t_state = TCPS_CLOSE_WAIT;
1134             break;
```

tcp_input.c

图29-24 tcp_input 函数：FIN处理，前半部分

1. 处理收到的第一个FIN

1116-1125 如果接收报文段FIN置位，并且是连接上收到的第一个FIN，则调用socantrcvmore，把插口设为只读，置位TF_ACKNOW，从而立即发送ACK(无延迟)。

rcv_nxt加1，越过FIN占用的序号。

1126 FIN处理的其余部分是一个大的switch语句，根据连接的状态进行转换。注意，连接处于CLOSED、LISTEN和SYN_SENT状态时，不处理FIN，因为处于这3个状态时，还未收到对端发送的SYN，无法同步接收序号，也就无法验证FIN序号的有效性。此外，连接处于CLOSING、CLOSE_WAIT和LAST_ACK状态时，也不处理FIN，因为在这3个状态下收到的FIN必然是一个重复报文段。

2. SYN_RCVD和ESTABLISHED状态

1127-1134 如果连接处于SYN_RCVD或ESTABLISHED状态，收到FIN后，新的状态为CLOSE_WAIT。

尽管在SYN_RCVD状态下收到FIN是合法的，但却极为罕见。图24-15的状态图未列出这一状态变迁。它意味着处于LISTEN状态的插口收到一个同时带有SYN和FIN的报文段。或者，正在监听的插口收到了SYN，连接转移到SYN_RCVD状态，但在收到ACK之前，先收到了FIN(从分析可知，FIN未携带有效的ACK，否则，图29-2中的代码会使连接转移到ESTABLISHED状态)。

图29-25给出了FIN处理的下一部分。

tcp_input.c

```
1135             /*
1136              * If still in FIN_WAIT_1 state FIN has not been acked so
1137              * enter the CLOSING state.
1138              */
1139         case TCPS_FIN_WAIT_1:
1140             tp->t_state = TCPS_CLOSING;
1141             break;

1142             /*
1143              * In FIN_WAIT_2 state enter the TIME_WAIT state,
1144              * starting the time-wait timer, turning off the other
1145              * standard timers.
1146              */
1147         case TCPS_FIN_WAIT_2:
1148             tp->t_state = TCPS_TIME_WAIT;
1149             tcp_canceltimers(tp);
1150             tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1151             soisdisconnected(so);
1152             break;

1153             /*
1154              * In TIME_WAIT state restart the 2 MSL time_wait timer.
1155              */
1156         case TCPS_TIME_WAIT:
1157             tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1158             break;
1159         }
1160     }
```

tcp_input.c

图29-25 tcp_input 函数：FIN处理，后半部分

3. FIN_WAIT_1状态

1135-1141 因为报文段的ACK处理已结束，如果处理FIN时，连接处于FIN_WAIT_1状态，意味着连接两端同时关闭连接——两端发送的两个FIN在网络中交错。连接进入CLOSING状态。

4. FIN_WAIT_2状态

1142-1148 收到FIN将使连接进入TIME_WAIT状态。当在FIN_WAIT_1状态收到携带ACK和FIN的报文段时(典型情况)，尽管图24-15显示连接直接从FIN_WAIT_1转移到TIME_WAIT状态，但在图29-11中处理ACK时，连接实际已进入FIN_WAIT_2状态。此处的FIN处理再将连接转到TIME_WAIT状态。因为ACK在FIN之前处理，所以连接总会经过FIN_WAIT_2状态，尽管是暂时性的。

5. 启动TIME_WAIT定时器

1149-1152 关闭所有等待的TCP定时器，并启动TIME_WAIT定时器，时限等于MSL(如果接收报文段中包含ACK和FIN，图29-11中的代码会启动FIN_WAIT_2定时器)。插口断开连接。

6. TIME_WAIT状态

1153-1159 如果在TIME_WAIT状态时收到FIN，说明这是一个重复报文段。与图29-14中的处理类似，启动TIME_WAIT定时器，时限等于两倍的MSL。

29.11 最后的处理

图29-26给出了tcp_input函数中首部预测失败时，较慢的执行路径中最后一部分的代码，以及标注dropafterack。

tcp_input.c

```
1161     if (so->so_options & SO_DEBUG)
1162         tcp_trace(TA_INPUT, ostate, tp, &tcp_saveti, 0);

1163     /*
1164      * Return any desired output.
1165      */
1166     if (needoutput || (tp->t_flags & TF_ACKNOW))
1167         (void) tcp_output(tp);
1168     return;

1169   dropafterack:
1170     /*
1171      * Generate an ACK dropping incoming segment if it occupies
1172      * sequence space, where the ACK reflects our state.
1173      */
1174     if (tiflags & TH_RST)
1175         goto drop;
1176     m_freem(m);
1177     tp->t_flags |= TF_ACKNOW;
1178     (void) tcp_output(tp);
1179     return;
```

tcp_input.c

图29-26 tcp_input 函数：最后的处理

1. SO_DEBUG插口选项

1161-1162 如果选用了SO_DEBUG插口选项，则调用tcp_trace向内核的环形缓存中添加记录。回想一下，图28-7中的代码同时保存了原有连接状态， IP和TCP的首部，因为函数有可能改变这些值。

2. 调用tcp_output

1163-1168 如果needoutput标志置位(图29-6和图29-15)，或者需要立即发送ACK，则调用tcp_output。

3. dropafterack

1169-1179 只有当RST标志未置位时，才会生成ACK(带有RST的报文段不会被确认)，释放保存接收报文段的mbuf链，调用tcp_output立即发送ACK。

图29-27结束tcp_input函数。

tcp_input.c

```
1180   dropwithreset:
1181     /*
1182      * Generate an RST, dropping incoming segment.
1183      * Make ACK acceptable to originator of segment.
1184      * Don't bother to respond if destination was broadcast/multicast.
1185      */
1186     if ((tiflags & TH_RST) || m->m_flags & (M_BCAST | M_MCAST) ||
1187         IN_MULTICAST(ti->ti_dst.s_addr))
1188         goto drop;
1189     if (tiflags & TH_ACK)
1190         tcp_respond(tp, ti, m, (tcp_seq) 0, ti->ti_ack, TH_RST);
1191     else {
1192         if (tiflags & TH_SYN)
1193             ti->ti_len++;
1194         tcp_respond(tp, ti, m, ti->ti_seq + ti->ti_len, (tcp_seq) 0,
1195                     TH_RST | TH_ACK);
1196     }
1197     /* destroy temporarily created socket */
1198     if (dropsocket)
1199         (void) soabort(so);
1200     return;
1201   drop:
1202     /*
1203      * Drop space held by incoming segment and return.
1204      */
1205     if (tp && (tp->t_inpcb->inp_socket->so_options & SO_DEBUG))
1206         tcp_trace(TA_DROP, ostate, tp, &tcp_saveti, 0);
1207     m_freem(m);
1208     /* destroy temporarily created socket */
1209     if (dropsocket)
1210         (void) soabort(so);
1211     return;
1212 }
```

tcp_input.c

图29-27 tcp_input 函数：最后的处理

4. dropwithreset

1180-1188 除了接收报文段也有RST,或者接收报文段是多播和广播报文段的情况之外,应发送RST。绝不允许因为响应RST而发送新的RST,这将引起RST风暴(两个端点间连续不断地交换RST)。

此处的代码存在与图28-16同样的错误：它不检查接收报文段的目的地址是否为广播地址。

类似地，IN_MULTICAST的目的地址参数应转换为主机字节序。

5. RST报文段的序号和确认序号

1189-1196 RST报文段的序号字段值、确认字段值和ACK标志取决于接收报文段中是否带有ACK。

图29-28总结了生成RST报文段中的这些字段。

接收到的报文段	生成的RST报文段		
	序 号 值	确认序号	输出标志
带有ACK	接收到的确认字段	0	TH_RST
不带ACK	0	接收到的序号字段	TH_RST|TH_ACK

图29-28 生成RST报文段各字段的值

正常情况下，除了起始的SYN(图24-16)，所有报文段都带有ACK。tcp_respond的第四个参数是确认序号，第五个参数是序号。

6. 拒绝连接

1192-1193 如果SYN置位，则ti_len必须加1，从而生成RST的确认字段比收到的SYN报文段的起始序号大1。如果到达的SYN请求与不存在的服务器建立连接，会执行这一段代码。此时，由于图28-6中的代码找不到请求的Internet PCB，控制跳转到dropwithreset。但为了使发送的RST能被对端接受，报文段必须确认SYN(图28-18)。卷1的18.14节举例说明了这种类型的RST。

最后请注意，tcp_respond利用保存接收报文段的第一个mbuf构造RST，并且释放链上的其他mbuf。当第一个mbuf最终到达设备驱动程序后，它也会被丢弃。

7. 释放临时创建的插口

1197-1199 如果在图28-7中为监听的服务器创建了临时的插口，但图28-16中的代码发现接收报文段有错误，它会置位drop socket。如果出现了这种情况，插口在此处被释放。

8. 丢弃(不带ACK或RST)

1201-1206 如果接收报文段被丢弃，且不生成ACK或RST，则调用tcp_trace。如果SO_DEBUG置位且生成了ACK，则tcp_output将向内核的环形缓存中添加一条跟踪记录。如果SO_DEBUG置位且生成了RST，系统不会为RST添加新的跟踪记录。

1207-1211 释放保存接收报文段的mbuf链。如果dropsocket非零，则释放临时创建的插口。

29.12 实现求精

为了加速TCP处理而进行的优化与UDP类似(23.12节)。应利用复制数据计算检验和，并避免在处理中多次遍历数据。[Dalton et al. 1993]讨论了这些修订。

连接数增加时，对TCP PCB的线性搜索也是一个处理瓶颈。[McKenney and Dove 1992]讨论了这个问题，利用哈希表替代了线性搜索。

[Partridge 1993]介绍了Van Jacobson开发的一个用于研究目的的协议实现，极大地减少了TCP的输入处理。接收数据分组首先由IP进行处理(RISC系统中约有25条指令)，之后由分用器(demultiplexer)寻找PCB(约10条指令)，最后由TCP处理(约30条指令)。这30条指令完成了首部预测，并计算伪首部检验和。如果数据报文段通过了首部预测，且应用进程正等待接收数据，则复制数据到应用进程缓存，计算TCP检验和并完成验证(一次遍历中完成数据复制和检验和计算)。如果TCP首部预测失败，则执行TCP输入处理中较慢的路径。

29.13 首部压缩

下面介绍TCP首部压缩。尽管首部压缩不是TCP输入处理的一部分，但需要彻底了解TCP

的工作机制后，才能很好地理解首部压缩。RFC 1144[Jacobson 1994a]中详细定义了首部压缩，因为Van Jacobson首先提出了这一算法，通常也称为 VJ 首部压缩。本节的目的不是详细讨论首部压缩的源代码(RFC 1144给出了实现代码，其中有很好的注释，程序量与 `tcp_output`差不多)，而是概括性地介绍一下算法的思想。请注意区分首部预测 (28.4节)和首部压缩。

29.13.1 引言

多数的SLIP和PPP实现支持首部压缩。尽管首部压缩，在理论上，适用于任何数据链路，但主要还是面向慢速串行链路。首部压缩只处理TCP报文段 —— 与其他的IP协议无关(如ICMP、IGMP、UDP等等)。它能够把IP/TCP组合首部从正常的40字节压缩到只有3字节，从而降低了交互性应用，如远程登录或 Telnet中TCP报文段的大小，从典型的 41字节减少到只剩 4字节 —— 大大提高了慢速串行链路的效率。

串行链路的两端，每端都维护着两个连接状态表，一个用于数据报的发送，另一个用于数据报的接收。每张表最多保存 256条记录，但典型的只有 16条，即同一时间内最多允许 16条不同的TCP连接执行首部压缩算法。每条记录中保存一个 8 bit的连接ID(限制记录数最多只能为256)、某些标志和最近接收/发送的数据报的未被压缩的首部。 96 bit的插口对可惟一确定一条连接 —— 源端IP地址和TCP端口、目的IP地址和TCP端口 —— 这些信息都保存在未压缩的首部中。图29-29举例说明了这些表的结构。

因为TCP连接是全双工的，在两个方向的数据流上都可执行首部压缩算法。连接两端必须同时实现压缩和解压缩。同一条连接在两端的表中都会出现，如图 29-29所示。在这个例子上部的两张表中，连接 ID等于1的表项的源端IP地址都等于128.1.2.3，源端TCP端口号都等于1500，目的IP地址等于192.3.4.5，目的TCP端口号都等于 25。在底部的两张表中，连接 ID等于2的记录保存了同一条连接反方向数据流的信息。

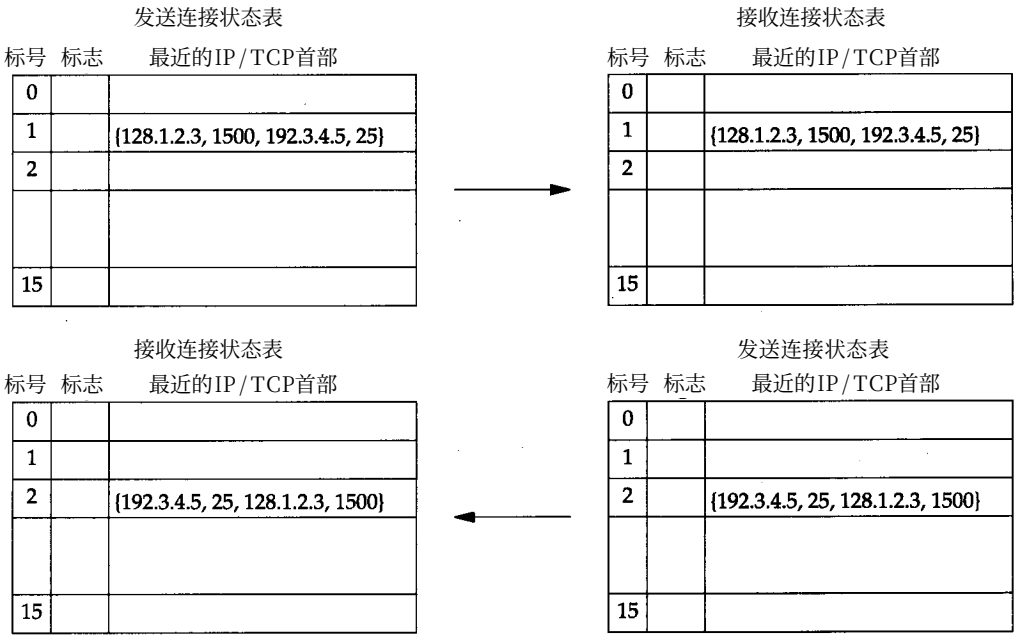


图29-29 链路(如SLIP链路)两端的一组连接状态表

我们在图29-29中利用数组表示这些表，但在源代码中，表项定义为一个结构，连接状态表定义为这些结构组成的环形链表，最近一次用过的结构位于表头。

因为连接两端都保存了最近用过的未压缩的数据报首部，所以只需在链路上传送当前数据报与前一数据报不同的字段（及一个特殊的前导字节，指明后续的是哪一个字段）。因为某些首部字段在相邻的数据报之间不会变化，而其他的首部字段变化也很小，这种差分处理是压缩算法的核心。首部压缩只适用于IP和TCP首部——TCP报文段的数据部分不变。

图29-30给出了发送方利用首部压缩算法，在串行链路上发送IP数据报时采取的步骤。

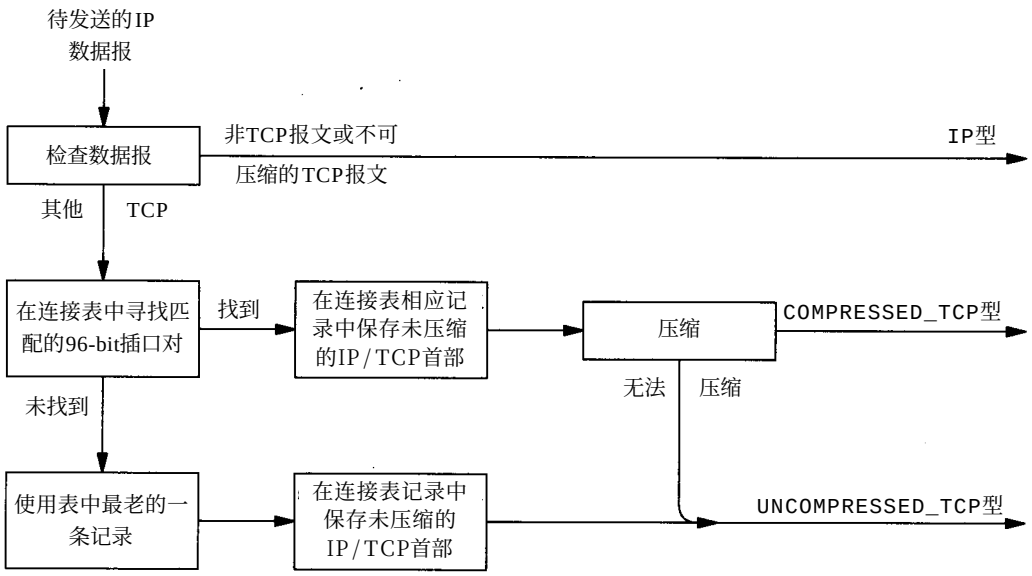


图29-30 发送方采用首部压缩时的步骤

接收方必须能够识别下面3种类型的数据报：

1) IP型数据报，前导字节的高位4比特等于4。这也是IP首部中正常的IP版本号(图8-8)，说明链路上发送的是正常的、未压缩的数据报。

2) COMPRESSED_TCP型数据报，前导字节的最高位置为1，类似于IP版本号介于8和15之间(剩余的7bit由压缩算法使用)，说明链路上发送的是压缩过的首部和未压缩的数据，接下来我们还会谈到这种类型的数据报。

3) UNCOMPRESSED_TCP型数据报，前导字节的高位4比特等于7，说明链路上发送的是正常的、未压缩的数据报，但IP的协议字段(等于6，对TCP)被替换为连接ID，接收方可据此从连接状态表中找到正确的记录。

接收方查看数据报的第一个字节，即前导字节，确定其类型，实现代码参见图5-13。图5-16中，发送方调用sl_compress_tcp确认TCP报文段是可压缩的，函数返回值与数据报首字节逻辑或后，结果依然保存在首字节中。

图29-31列出了链路上传送的前导字节，其中4位“-”表示正常的IP首部长度字段。7位“C、I、P、S、A、W和E”指明后续的是哪些可选字段，后面会简单地介绍这些字母的含义。

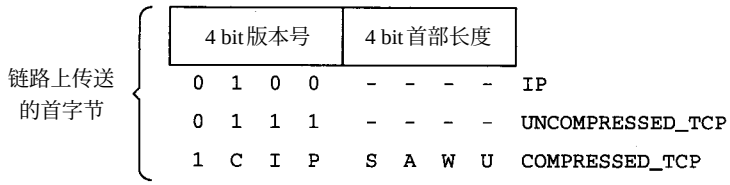


图29-31 链路上传送的前导字节

图29-32给出了使用压缩算法之后，不同类型的完整的IP数据报。

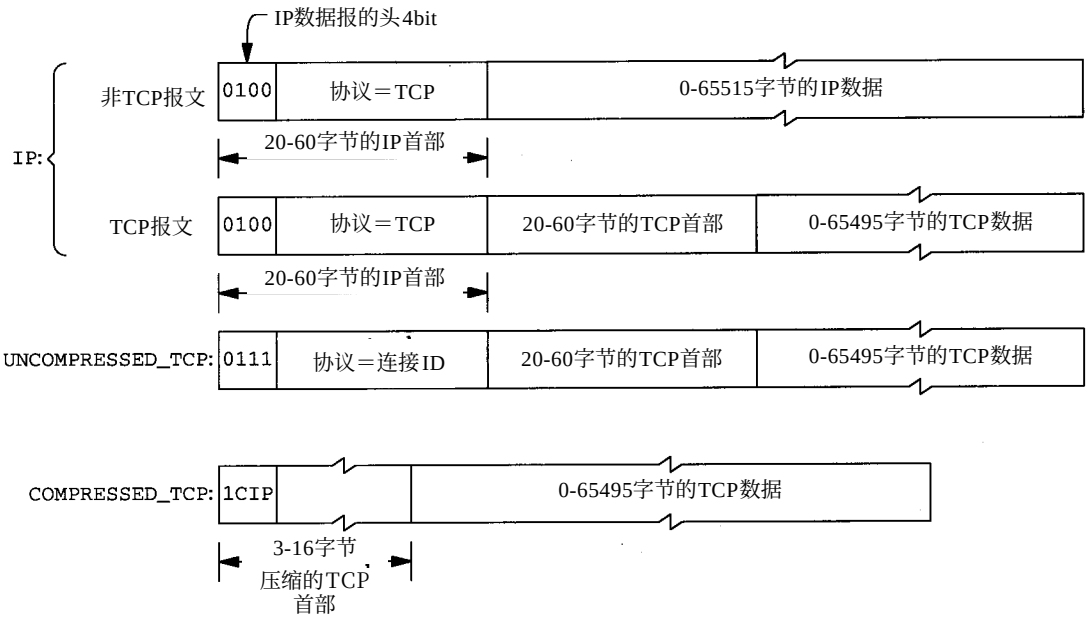


图29-32 采用首部压缩后的不同类型的IP数据报

图中给出了两个IP型数据报：一个携带了非TCP报文段(如UDP、ICMP或IGMP协议报文段)，另一个携带了TCP报文段。这是为了说明做为IP型数据报发送的TCP报文段与做为UNCOMPRESSED_TCP型数据报发送的TCP报文段间的差异：前导字节的高位4比特互不相同，类似于IP首部的协议字段。

如果IP数据报的协议字段不等于TCP，或者协议是TCP，但下列条件之一为真，都不会采用首部压缩算法。

- 数据报是一个IP分片：分片偏移量非零或者分片标志置位；
- SYN、FIN或RST中的任何一个置位；
- ACK标志未置位。

上述3个条件中只要有一个为真，都将作为IP型数据报发送。

此外，即使数据报携带了可压缩的TCP报文段，压缩算法也可能失败，生成UNCOMPRESSED_TCP型的数据报。可能因为当前数据报与连接上发送的上一个数据报比较时，有些特殊字段发生了变化，而正常情况下，对于给定的连接，它们应该不变，从而导致压缩算法无法反映存在的变化。例如，TOS字段，分片标志位。此外，如果某些字段数值的

差异超过65535，压缩算法也会失败。

29.13.2 首部字段的压缩

下面介绍如何压缩图29-33中给出的IP和TCP的首部字段，阴影字段指对于给定连接，正常情况下不会发生变化的字段。

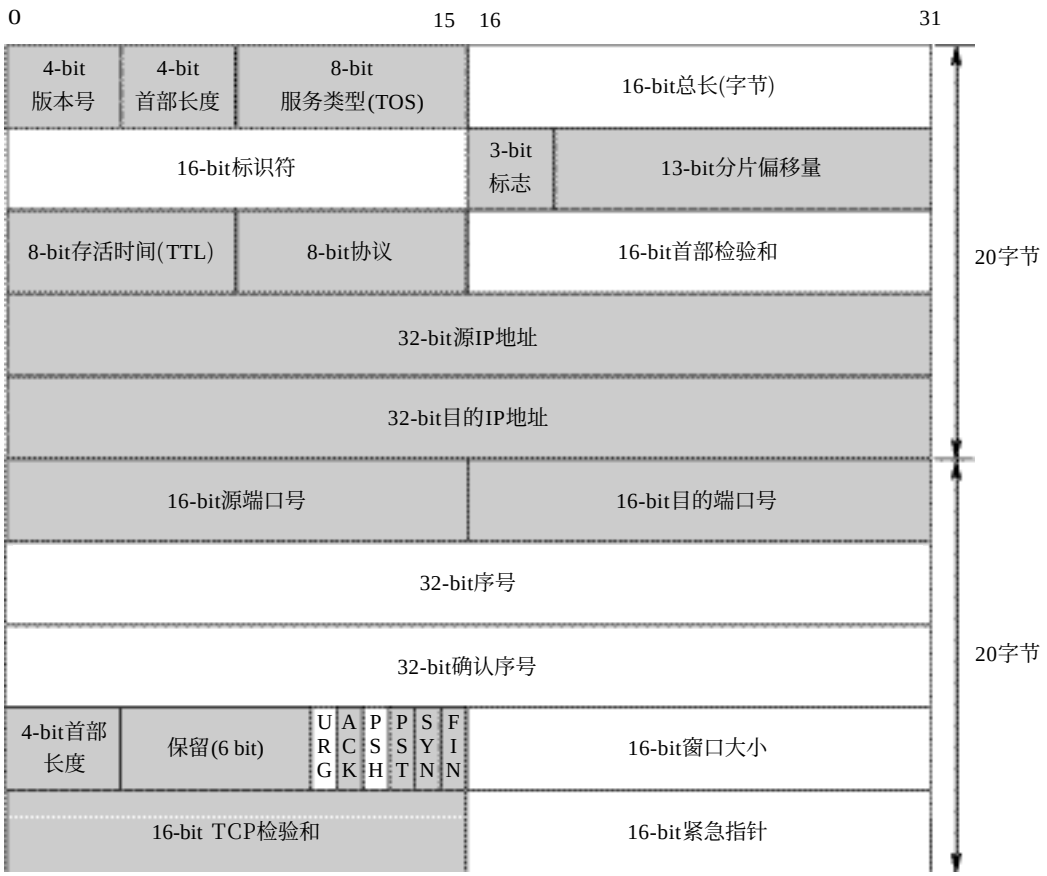


图29-33 组合的IP和TCP首部：阴影字段通常不变化

如果连接上发送的前一个报文段与当前报文段之间，有阴影字段发生变化，则压缩算法失败，报文段被直接发送。图中未列出 IP和TCP选项，但如果它们存在，且这些选项字段发生了变化，则报文段也不压缩，而被直接发送 (习题29.7)。

如果阴影字段均未变化，即使算法只传输非阴影字段，也会节省 50%的传输容量。 VJ首部压缩甚至做得更好，图 29-34给出了压缩后的IP/TCP首部格式。

最小的压缩后的IP/TCP首部只有3个字节：第一个字节(标志比特)，加上16 bit的TCP检验和。为了防止可能的链路错误，一般不改动 TCP检验和(SLIP不提供链路层的检验和，尽管PPP提供一个)。

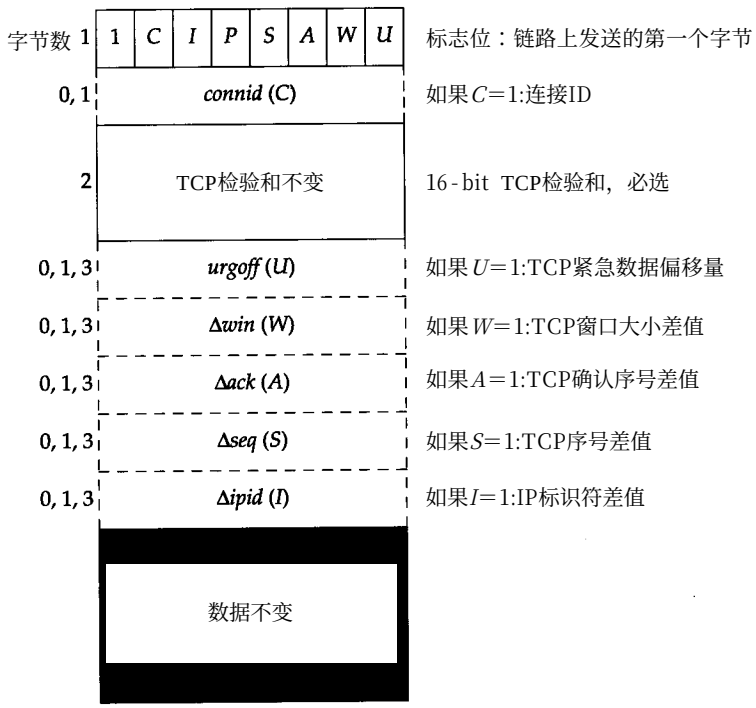


图29-34 压缩后的IP/TCP首部格式

其他的6个字段：*connid*、*urgoff*、Δ*win*、Δ*ack*、Δ*seq*和Δ*ipid*，都是可选的。图29-34的最左侧列出了各字段压缩后所需的字节数。读者可能认为压缩后的首部最大应占用 19字节，但实际上压缩后的首部中4 bit的*SAWU*绝不可能同时置位，因此，压缩首部最大为16字节，后面我们还会详细讨论这个问题。

第一个字节的最高位比特必须设为 1，说明这是`COMPRESSED_TCP`型的数据报。其余7 bit中的6个规定了后续首部中存在哪些可选字段，图29-35小结了这7 bit的用法。

标志比特	描　述	结构变量	标志等于0说明	标志等于1说明
C	连接ID		连接ID不变	*connid*=连接ID
I	IP标识符	`ip_id`	`ip_id`已加1	Δ*ipid*=IP标识符差值
P	TCP推标志		`PSH`标志清除	`PSH`标志置位
S	TCP序号	`th_seq`	`th_seq`不变	Δ*seq*=TCP序号差值
A	TCP确认序号	`th_ack`	`th_ack`不变	Δ*ack*=TCP确认序号差值
W	TCP窗口	`th_win`	`th_win`不变	Δ*win*=TCP窗口字段差值
U	TCP紧急数据偏移量	`th_urg`	URG标志未置位	*urgoff*=紧急数据偏移量

图29-35 压缩首部中的7个标志比特

C 如果*C*比特等于0，则当前报文段与前一报文段 (无论是压缩的或非压缩的)具有相同的连接ID。如果等于1，则*connid*将等于连接ID，其值位于0~255之间。

I 如果*I*比特等于0，当前报文段的IP标识符较前一报文段加 1(典型情况)。如果等于1，Δ*ipid*等于`ip_id`的当前值减去它的前一个值。

P 这个比特复制自TCP报文段中的PSH标志位。因为PSH标志不同于其他的正常方式，必须在每个报文段中明确地定义这一标志。

S 如果*S*比特等于0，TCP序号不变。如果等于1，Δ*seq*等于th_seq的当前值减去它的前一个值。

A 如果*A*比特等于0，TCP确认序号不变(典型情况)。如果等于1，Δ*ack*等于th_ack的当前值减去它的前一个值。

W 如果*W*比特等于0，TCP窗口大小不变。如果等于1，Δ*win*等于th_win的当前值减去它的前一个值。

U 如果*U*比特等于0，报文段的URG标志未置位，紧急数据偏移量不变(典型情况)。如果等于1，说明URG标志置位，*urgoff*等于th_urg的当前值。如果URG标志未置位时，紧急数据偏移量发生改变，报文段将被直接发送(这种现象通常发生在紧急数据传送完毕后的第一个报文段)。

通过字段的当前值减去它的前一个值，得到需传输的差值。正常情况下，得到的是一个小正数(Δ*win*是个例外)。

请注意，图29-34中有5个字段的长度可变，可占用0、1或3字节。

0字节：对应标志未置位，此字段不存在；

1字节：发送值在1~255之间，只需占用1字节；

3字节：如果发送值等于0或者在256~65535之间，则需要用3个字节才能表示：第一个字节全0，后两个字节保存实际值。这种方法一般用于3个16 bit的值：*urgoff*、Δ*win*和Δ*ipid*。但如果两个32 bit字段Δ*ack*和Δ*seq*的差值小于0或者大于65535，报文段将被直接发送。

如果把图29-33中不带阴影的字段与图29-34中可能的传输字段进行比较，会发现有些字段永远不会被传输。

- IP总长度字段不会被传输，因为绝大多数链路层向接收方提供接收数据分组的长度。
- 因为IP首部中被传输的惟一字段是16 bit的IP标识符，IP检验和被忽略。因为它只在一段链路上保护IP首部，每次转发都会被重新计算。

29.13.3 特殊情况

算法检查输入报文段，如果出现两种特定情况，则用前导字节的低位4比特——*SAWU*——的两种特殊组合，分别加以表示。因为紧急数据很少出现，如果报文段中URG标志置位，并且与前一报文段相比，序号与窗口字段都发生了变化(意味着低位4比特应为1011或1111)，此种报文段会跳过压缩算法，被直接发送。因此，如果低位4比特等于1011(称为**SA*)或1111(称为**S*)，就说明出现了下面两种特定情况：

*SA 序号与确认序号都增加，差值等于前一报文段的数据量，窗口大小与紧急数据偏移量不变，URG标志未置位。采用这种表示法可以避免传送Δ*seq*和Δ*ack*。
如果对端回送终端数据，那么两个传输方向上的数据报文段中都会经常出现这一现象。卷1的图19-3和图19-4，举例说明了远程登录应用中出现的这种类型的数据。

*S 序号增加，差值等于前一报文段的数据量，确认序号、窗口大小与紧急数据偏移量均不变，URG标志未置位。采用这种表示法可以避免传送Δ*seq*。
这种类型的数据通常出现在单向数据传输(如FTP)的发送方。卷1的图20-1、图20-2

和图20-3举例说明了这种类型的数据传输。此外，如果对端不回送终端数据，那么在数据发送方的数据报文段中也会出现这种现象。

29.13.4 实例

下面的两个例子，在图1-17中的bsdi和slip两个系统间，利用SLIP链路传输数据。这条SLIP链路在两个传输方向上都采用了首部压缩算法。在主机bsdi上运行tcpdump程序(卷1的附录A)，保存所有数据帧的备份。这个程序还支持一个选项，能够输出压缩后的首部，列出图29-34中的所有字段。

在主机间已建立了两条连接：一条远程登录连接，另一条是从bsdi到slip的文件传输(FTP)。图29-36列出了两条连接上不同类型数据帧出现的次数。

帧 类 型	远程登录		FTP	
	输入	输出	输入	输出
IP	1	1	5	5
UNCOMPRESSED_TCP	3	2	2	3
COMPRESSED_TCP				
特殊情况*SA	75	75	0	0
特殊情况*S	25	1	1	325
一般情况	9	93	337	13
总数	113	172	345	346

图29-36 远程登录和FTP连接上，不同类型数据帧出现的次数

远程登录连接中，在两个传输方向上，*SA都出现了75次，从而证明了在对端回显终端流量时，这一特定情况在两个传输方向上都会经常出现。FTP连接中，在数据的发送方，*S出现了325次，也证明了对于单向数据传输，这一特定情况会经常出现在数据的发送方。

FTP连接中，IP型的数据帧出现了10次，对应于4个带有SYN的报文段，和6个带有FIN的报文段。FTP使用了两条连接：一条用于传输交互式命令，另一条用于文件传输。

UNCOMPRESSED_TCP型数据帧一般对应于连接建立后的第一个报文段，即同步连接ID的报文段。这两个例子中还有少量的其他类型的报文段，主要用于服务类型设定(Net/3中的远程登录及FTP客户及服务器都是在连接建立后才设定TOS字段)。

字节数	远程登录		FTP	
	输入	输出	输入	输出
3	102	44	2	250
4		94		78
5	7	12	5	2
6		6	325	5
7		13	2	1
8				1
9			4	1
总数	109	169	338	338

图29-37 压缩首部大小的分布

图29-37给出了压缩首部大小的分布情况，后 4栏中压缩首部的平均大小为分别等于 3.1、4.1、6.0和3.3字节，与原来的40字节相比，大大提高了系统的传输效率，尤其对于交互式连接，效果更加明显。

在FTP输入一栏中，压缩首部大小为6字节的报文段有325个，其中绝大多数只携带了值等于256的Δ*ack*字段，因为256大于255，所以必须用3个字节表示。 SLIPMTU等于296，因此，TCP采用了256的MSS。在FTP输出一栏中，压缩首部大小为 3字节的报文段有250个，其中绝大多数都代表*S类的特定情况(只有序号发生变化)，差值等于256。但因为*S的序号差值默认为前一报文段的数据量，所以只需传输前导字节和 TCP检验和。在FTP输出一栏中，78个压缩首部大小为4字节的报文段也属于同一情况，只不过 IP标识符也发生了变化(习题29.8)。

29.13.5 配置

对给定的SLIP或PPP链路，首部压缩必须被选定后才能起作用。配置 SLIP链路接口时，一般可设定两个标志：首部压缩标志和自动首部压缩标志。配置命令是 `ifconfig`，分别带选项`link0`和`link2`。正常情况下，由客户端 (拨号主机)决定是否采用首部压缩算法，服务器(客户通过拨号接入的主机或终端服务器)只选择是否置位自动首部压缩标志。如果客户选用了首部压缩算法，它的 TCP首先发送一个`UNCOMPRESSED_TCP`型的数据报，规定连接 ID。如果服务器收到这个数据报，它也开始采用首部压缩算法 (服务器处于自动方式)；如果未收到这个数据报，服务器绝不会在这条链路上采用首部压缩。

PPP允许在链路建立时，连接双方共同协商传输选项，其中的一个选项即是否支持首部压缩算法。

29.14 小结

本章结束了我们对 TCP输入处理的详细介绍。首先介绍了如果连接在 SYN_RCVD状态时收到了ACK，该如何处理，即如何完成被动打开、同时打开或自连接。

快速重传算法指 TCP在连续收到的重复 ACK数超过规定的门限值后，能够检测到丢失的报文段并进行重发，即使重传定时器还未超时。 Net/3结合了快速重传算法与快速恢复算法，执行拥塞避免算法而非慢起动，尽量保证发送方到接收方的数据流不中断。

ACK处理负责从插口的发送缓存中丢弃已确认的数据，并且在收到的 ACK会改变连接当前状态时，对一些TCP状态做特殊处理。

处理接收报文段的 URG标志，如果置位，则通过 TCP紧急方式的处理，提取带外数据。这一操作是非常复杂的，因为应用进程可以利用正常的数据流缓存，或者特殊的带外数据缓存接收带外数据，而且TCP收到URG时，紧急指针所指向的数据可能还未到达。

TCP输入处理结束时，会调用 `TCP_REASS`，提取报文段中的数据放入插口的接收缓存或重组队列，处理 FIN标志，并且在接收报文段需要响应时，调用 `tcp_output`输入响应报文段。

TCP首部压缩是用于SLIP和PPP链路的一种技术，能够把 IP和TCP首部长度从40字节减少到约为3~6字节(典型情况)。这是因为对于给定连接，相邻两个报文段之间，首部的多数字段不会改变，即使有些字段的值发生了变化，其差值也很小，从而可以通过前导字节中的标志比特，指明哪些字段发生了变化，在后续部分只传输这些字段的当前值与前一报文段间的差值。

习题

29.1 客户与服务器建立连接，不考虑报文段丢失，哪一个应用进程，客户或服务器，首先完成连接建立过程？

29.2 Net/3系统中，监听服务器收到了一个 SYN，它同时携带了50字节的数据。会发生什么？

29.3 继续前一个习题，假定客户没有重传 50字节的数据，而是在对服务器 SYN/ACK报文段的确认中置位FIN标志，会发生什么？

29.4 Net/3客户向服务器发送SYN，服务器响应SYN/ACK，其中还携带了50字节的数据和FIN标志。列出客户端TCP的处理步骤。

29.5 卷1的图18-19和RFC 793的图14，都给出了出现同时关闭时，连接双方交换的4个报文段。但如果连接两端都是 Net/3系统，出现同时关闭时，或者一个 Net/3系统的自连接关闭时，彼此将交换6个报文段，而不是4个，多余出两个报文段是因为连接两端各自收到对端的FIN后，将向对端重发FIN。问题出在什么地方，如何解决？

29.6 RFC 793第72页建议，如果发送缓存中的数据已被对端确认，“应给用户一个确认，指明缓存中已发送且被确认的数据 (例如，发送缓存返回时应带有‘ OK ’响应)”。Net/3是否提供了这种机制？

29.7 RFC 1323中定义的选项对TCP首部压缩有何影响？

29.8 Net/3对IP标识符字段的赋值方式，对TCP首部压缩有何影响？

第30章 TCP的用户需求

30.1 引言

本章介绍TCP的用户请求处理函数tcp_usrreg，它被协议的pr_usrreq函数调用，处理各种与TCP插口有关的系统调用。此外，还将介绍 tcp_ctloutput，应用进程调用setsockopt设定TCP 插口选项时，会用到它。

30.2 tcp_usrreq函数

TCP的用户请求函数用于处理多种操作。图 30-1给出了tcp_usrreq函数的基本框架，其中switch的语句体部分将在后续部分逐一展开。图 15-17中列出了函数的参数，其具体含义取决于所处理的用户请求。

tcp_usrreq.c

```
45 int
46 tcp_usrreq(so, req, m, nam, control)
47 struct socket *so;
48 int     req;
49 struct mbuf *m, *nam, *control;
50 {
51     struct inpcb *inp;
52     struct tcpcb *tp;
53     int     s;
54     int     error = 0;
55     int     ostate;
56     if (req == PRU_CONTROL)
57         return (in_control(so, (int) m, (caddr_t) nam,
58                            (struct ifnet *) control));
59     if (control && control->m_len) {
60         m_freem(control);
61         if (m)
62             m_freem(m);
63         return (EINVAL);
64     }
65     s = splnet();
66     inp = sotoinpcb(so);
67     /*
68      * When a TCP is attached to a socket, then there will be
69      * a (struct inpcb) pointed at by the socket, and this
70      * structure will point at a subsidary (struct tcpcb).
71      */
72     if (inp == 0 && req != PRU_ATTACH) {
73         splx(s);
74         return (EINVAL);        /* XXX */
75     }
76     if (inp) {
77         tp = intotcpcb(inp);
```

图30-1 tcp_usrreq 函数体

```
78          /* WHAT IF TP IS 0? */
79          ostate = tp->t_state;
80      } else
81          ostate = 0;
82      switch (req) {
                                    /* switch cases */
276     default:
277         panic("tcp_usrreq");
278     }
279     if (tp && (so->so_options & SO_DEBUG))
280         tcp_trace(TA_USER, ostate, tp, (struct tcpiphdr *) 0, req);
281     splx(s);
282     return (error);
283 }
```

tcp_usrreq.c

图30-1 (续)

1. in_control处理ioctl请求

45-58 PRU_CONTROL请求来自于ioctl系统调用，函数in_control负责处理这一请求。

2. 控制信息无效

59-64 如果试图调用sendmsg，为TCP 插口配置控制信息，代码将释放 mbuf，并返回EINVAL差错代码，声明这一操作无效。

65-66 函数接着执行splnet。这种做法极为保守，因为并非在所有情况下都需要锁定，只是为了防止在case语句中单个地调用splnet。我们在图23-15中曾提到，调用splnet设定处理器的优先级，唯一的作用是阻止软中断执行 IP输入处理(它会接着调用tcp_input)，但却无法阻止接口层接收输入数据分组并放入到IP的输入队列中。

通过指向插口结构的指针，可得到指向 Internet PCB的指针。只有在应用进程调用socket系统调用，发出PRU_ATTACH请求时，该指针才允许为空。

67-81 如果inp非空，当前连接状态将保存在 ostate中，以备函数结束时可能会调用tcp_trace。

下面我们开始讨论单独的case语句。应用进程调用socket系统调用，或者监听服务器收到连接请求(图28-7)，调用sonewconn函数时，都会发出PRU_ATTACH请求，图30-2给出了这一请求的处理代码。

tcp_usrreq.c

```
83          /*
84           * TCP attaches to socket via PRU_ATTACH, reserving space,
85           * and an internet control block.
86           */
87      case PRU_ATTACH:
88          if (inp) {
89              error = EISCONN;
90              break;
91          }
92          error = tcp_attach(so);
93          if (error)
```

图30-2 tcp_usrreq 函数：PRU_ATTACH 和PRU_DETACH 请求

```
94              break;
95          if ((so->so_options & SO_LINGER) && so->so_linger == 0)
96              so->so_linger = TCP_LINGERTIME;
97          tp = sototcpcb(so);
98          break;

99          /*
100         * PRU_DETACH detaches the TCP protocol from the socket.
101         * If the protocol state is non-embryonic, then can't
102         * do this directly: have to initiate a PRU_DISCONNECT,
103         * which may finish later; embryonic TCB's can just
104         * be discarded here.
105         */
106     case PRU_DETACH:
107         if (tp->t_state > TCPS_LISTEN)
108             tp = tcp_disconnect(tp);
109         else
110             tp = tcp_close(tp);
111         break;
```

tcp_usrreq.c

图30-2 (续)

3. PRU_ATTACH请求

83-94 如果插口结构已经指向某个PCB，则返回EISCONN差错代码。调用tcp_attach完成处理：分配并初始化Internet PCB和TCP控制块。

95-96 如果选用了SO_LINGER插口选项，且拖延时间为0，则将其设为120(TCP_LINGERTIME)。

为什么在PRU_ATTACH请求发出之前，就可以设定插口选项？尽管不可能在调用socket之前就设定插口选项，但sonewconn也会发送PRU_ATTACH请求。它在把监听插口的so_options复制到新建插口之后，才会发送PRU_ATTACH请求。此处的代码防止新建连接从监听插口中继承拖延时间为0的SO_LINGER选项。

请注意，此处的代码有错误。常量TCP_LINGERTIME在tcp_timer.h中初始化为120，该行的注释为“最多等待2分钟”。但SO_LINGER值也是内核tsleep函数(由soclose调用)的最后一个参数，从而成为内核的timeout函数的最后一个参数，单位为滴答，而非秒。如果系统的滴答频率(hz)等于100，则拖延时间将变为1.2秒，而非2分钟。

97 现在，tp已指向插口的TCP控制块。这样，如果选定了SO_DEBUG插口选项，函数结束时就可以输出所需信息。

4. PRU_DETACH请求

99-111 close系统调用在PRU_DISCONNECT请求失败后，将发送PRU_DETACH请求。如果连接尚未建立(连接状态小于ESTABLISHED)，则无需向对端发送任何信息。但如果连接已建立，则调用tcp_disconnect初始化TCP的连接关闭过程(发送所有缓存中的数据，之后发送FIN)。

代码if语句的测试条件要求状态大于LISTEN，这是不正确的。因为如果连接状态等于SYN_SENT或者SYN_RCVD，两者都大于LISTEN，此时tcp_disconnect会直接调用tcp_close。实际上，这个case语句可以简化为直接调用tcp_disconnect。

图30-3给出了bind和listen系统调用的处理代码。

tcp_usrreq.c

```
112         /*
113          * Give the socket an address.
114          */
115     case PRU_BIND:
116         error = in_pcbbind(inp, nam);
117         if (error)
118             break;
119         break;

120         /*
121          * Prepare to accept connections.
122          */
123     case PRU_LISTEN:
124         if (inp->inp_lport == 0)
125             error = in_pcbbind(inp, (struct mbuf *) 0);
126         if (error == 0)
127             tp->t_state = TCPS_LISTEN;
128         break;
```

tcp_usrreq.c

图30-3 tcp_usrreq 函数：PRU_BIND 和PRU_LISTEN 请求

112-119 PRU_BIND请求的处理只是简单地调用in_pcbbind。

120-128 对于PRU_LISTEN请求，如果插口还未绑定在某个本地端口上，则调用in_pcbbind自动为其分配一个。这种情况十分少见，因为多数服务器会明确地绑定一个知名端口，尽管RPC(远端过程调用)服务器一般是绑定在一个临时端口上，并通过 Port Mapper向系统注册该端口(卷1的29.4节介绍了Port Mapper)。连接状态变迁到LISTEN，完成了listen调用的主要目的：设定插口的状态，以便接受到达的连接请求 (被动打开)。

图30-4给出了connect系统调用的处理代码：客户发起的主动打开。

tcp_usrreq.c

```
129         /*
130          * Initiate connection to peer.
131          * Create a template for use in transmissions on this connection.
132          * Enter SYN_SENT state, and mark socket as connecting.
133          * Start keepalive timer, and seed output sequence space.
134          * Send initial segment on connection.
135          */
136     case PRU_CONNECT:
137         if (inp->inp_lport == 0) {
138             error = in_pcbbind(inp, (struct mbuf *) 0);
139             if (error)
140                 break;
141         }
142         error = in_pcbconnect(inp, nam);
143         if (error)
144             break;

145         tp->t_template = tcp_template(tp);
146         if (tp->t_template == 0) {
147             in_pcbdisconnect(inp);
148             error = ENOBUFS;
```

图30-4 tcp_usrreq 函数：PRU_CONNECT 请求

```
149             break;
150         }
151         /* Compute window scaling to request.  */
152         while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
153                (TCP_MAXWIN << tp->request_r_scale) < so->so_rcv.sb_hiwat)
154             tp->request_r_scale++;
155         soisconnecting(so);
156         tcpstat.tcps_connattempt++;
157         tp->t_state = TCPS_SYN_SENT;
158         tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;

159         tp->iss = tcp_iss;
160         tcp_iss += TCP_ISSINCR / 2;
161         tcp_sendseqinit(tp);

162         error = tcp_output(tp);
163         break;
```

tcp_usrreq.c

图30-4 (续)

5. 分配临时端口

129-141 如果插口还未绑定在某个本地端口上，调用ip_pcbbind自动为其分配一个。对于客户端，这是很常见的，因为客户一般不关心本地端口值。

6. 连接PCB

142-144 调用in_pcbconnect，获取到达目的地的路由，确定外出接口，验证插口对不重复。

7. 初始化IP和TCP首部

145-150 调用tcp_template分配mbuf，保存IP和TCP的首部，并初始化两个首部，填入尽可能多的信息。会造成函数失败的唯一原因是内核耗尽了 mbuf。

8. 计算窗口缩放因子

151-154 计算用于接收缓存的窗口缩放因子：左移65535(TCP_MAXWIN)，直到它大于或等于接收缓存的大小(so_rcv.sb_hiwat)。得到的位移次数(0~14之间)，就是需要在SYN中发送的缩放因子值(图28-7处理被动打开时，有相同的代码)。应用进程必须在调用 connect之前，设定SO_RCVBUF插口选项，TCP才会在SYN中添加窗口大小选项，否则，将使用接收缓存大小的默认值(图24-3中的tcp_recvspace)。

9. 设定插口和连接的状态

155-158 调用soisconnecting，置位插口状态变量中恰当的比特，设定TCP连接状态为SYN_SENT，从而在后续的tcp_output调用中发送SYN(参见图24-16的tcp_outlags值)。连接建立定时器启动，时限初始化为75秒。tcp_output还会启动SYN的重传定时器，如图25-16所示。

10. 初始化序号

159-161 令初始序号等于全局变量tcp_iss，之后令tcp_iss增加64 000(TCP_ISSINCR除以2)。在监听服务器收到SYN并初始化ISS时(图28-17)，对tcp_iss的相同的操作。接着调用tcp_sendseqinit初始化发送序号。

11. 发送初始SYN

162 调用tcp_output发送初始SYN，以建立连接。如果tcp_output返回错误(例如，mbuf耗尽或没有到达目的地的路由)，该差错代码将成为tcp_usrreq的返回值，报告给应用进程。

图30-5给出了PRU_CONNECT2、PRU_DISCONNECT和PRU_ACCEPT请求的处理代码。

164-169 PRU_CONNECT2请求，来自于socketpair系统调用，对TCP协议无效。

170-183 close系统调用会发送PRU_DISCONNECT请求。如果连接已建立，应调用tcp_disconnect，发送FIN，执行正常的TCP关闭操作。

tcp_usrreq.c

```
164         /*
165          * Create a TCP connection between two sockets.
166          */
167     case PRU_CONNECT2:
168         error = EOPNOTSUPP;
169         break;

170         /*
171          * Initiate disconnect from peer.
172          * If connection never passed embryonic stage, just drop;
173          * else if don't need to let data drain, then can just drop anyway,
174          * else have to begin TCP shutdown process: mark socket disconnecting,
175          * drain unread data, state switch to reflect user close, and
176          * send segment (e.g. FIN) to peer.  Socket will be really disconnected
177          * when peer sends FIN and acks ours.
178          *
179          * SHOULD IMPLEMENT LATER PRU_CONNECT VIA REALLOC TCPCB.
180          */
181     case PRU_DISCONNECT:
182         tp = tcp_disconnect(tp);
183         break;

184         /*
185          * Accept a connection.  Essentially all the work is
186          * done at higher levels; just return the address
187          * of the peer, storing through addr.
188          */
189     case PRU_ACCEPT:
190         in_setpeeraddr(inp, nam);
191         break;
```

tcp_usrreq.c

图30-5 tcp_usrreq函数：PRU_CONNECT2、PRU_DISCONNECT和PRU_ACCEPT请求

请注意以“应该实现”起头的注释，这是因为无法接着使用出现错误的插口。例如，客户调用connect，并得到一个错误，它就无法在同一个插口上再次调用connect，而必须首先关闭插口，调用socket创建新的插口，在新的插口上才能再次调用connect。

184-191 与accept系统调用有关的工作全部由插口层和协议层完成。PRU_ACCEPT请求只简单地向应用进程返回对端的IP地址和端口号。

图30-6给出了PRU_SHUTDOWN、PRU_RCVD和PRU_SEND请求的处理代码。

12. PRU_SHUTDOWN请求

192-200 应用进程调用shutdown，禁止更多的输出时，soshutdown会发送PRU_SHUTDOWN请求。调用socantsendmore置位插口的标志，禁止继续发送报文段。接着调用tcp_usrclosed，根据图24-15的状态变迁图，设定正确的连接状态。tcp_output发送FIN之前，如果发送缓存中仍有数据，会首先发送等待数据。

tcp_usrreq.c

```
192         /*
193          * Mark the connection as being incapable of further output.
194          */
195     case PRU_SHUTDOWN:
196         socantsendmore(so);
197         tp = tcp_usrclosed(tp);
198         if (tp)
199             error = tcp_output(tp);
200         break;

201         /*
202          * After a receive, possibly send window update to peer.
203          */
204     case PRU_RCVD:
205         (void) tcp_output(tp);
206         break;

207         /*
208          * Do a send by putting data in output queue and updating urgent
209          * marker if URG set.  Possibly send more data.
210          */
211     case PRU_SEND:
212         sbappend(&so->so_snd, m);
213         error = tcp_output(tp);
214         break;
```

tcp_usrreq.c

图30-6 tcp_usrreq 函数：PRU_SHUTDOWN 、PRU_RCVD 和PRU_SEND 请求

13. PRU_RCVD请求

201-206 应用进程从插口的接收缓存中读取数据后，soreceive会发送这个请求。此时接收缓存已扩大，也许会有足够的空间，让TCP发送更大的窗口通告。tcp_output会决定是否需要发送窗口更新报文段。

14. PRU_SEND请求

207-214 图23-14中给出的5个写函数，都以这一请求结束。调用sbappend，向插口的发送缓存中添加数据(它将一直保存在缓存中，直到被确认)，并调用tcp_output发送新报文段(如果条件允许)。

图30-7给出了PRU_ABORT和PRU_SENSE请求的处理代码。

tcp_usrreq.c

```
215         /*
216          * Abort the TCP.
217          */
218     case PRU_ABORT:
219         tp = tcp_drop(tp, ECONNABORTED);
220         break;

221     case PRU_SENSE:
222         ((struct stat *) m)->st_blksize = so->so_snd.sb_hiwat;
223         (void) splx(s);
224         return (0);
```

tcp_usrreq.c

图30-7 tcp_usrreq 函数：PRU_ABORT 和PRU_SENSE 请求

15. PRU_ABORT请求

215-220 如果插口是监听插口(如服务器)，并且存在等待建立的连接，例如已发送初始

SYN或已完成三次握手过程，但还未被服务器 accept的连接，调用 soclose会导致发送PRU_ABORT请求。如果连接已同步， tcp_drop将发送RST。

16. PRU_SENSE请求

221-224 fstat系统调用会生成 PRU_SENSE请求。TCP返回发送缓存的大小，保存在stat结构的成员变量st_blksize中。

图30-8给出了PRU_RCVOOB的处理代码。当应用进程置位 MSG_OOB标志，试图读取带外数据时， soreceive会发送这一请求。

tcp_usrreq.c

```
225     case PRU_RCVOOB:
226         if ((so->so_oobmark == 0 &&
227              (so->so_state & SS_RCVATMARK) == 0) ||
228             so->so_options & SO_OOBINLINE ||
229             tp->t_oobflags & TCPOOB_HADDATA) {
230             error = EINVAL;
231             break;
232         }
233         if ((tp->t_oobflags & TCPOOB_HAVEDATA) == 0) {
234             error = EWOULDBLOCK;
235             break;
236         }
237         m->m_len = 1;
238         *mtod(m, caddr_t) = tp->t_iobc;
239         if (((int) nam & MSG_PEEK) == 0)
240             tp->t_oobflags ^= (TCPOOB_HAVEDATA | TCPOOB_HADDATA);
241         break;
```

tcp_usrreq.c

图30-8 tcp_usrreq 函数：PRU_RCVOOB 请求

17. 能否读取带外数据

225-232 如果下列3个条件有一个为真，应用进程读取带外数据的努力就会失败。

1) 如果插口的带外数据分界点 (so_oobmark)等于0，并且插口的SS_RCVATMARK标志未设定；或者

2) 如果SO_OOBINLINE插口选项设定；或者

3) 如果连接的TCPOOB_HADDATA标志设定(例如，连接的带外数据已被读取)。

如果上述3个条件中任何一个为真，则返回差错代码 EINVAL。

18. 是否有带外数据到达

233-236 如果上述3个条件全假，但TCPOOB_HAVEDATA标志置位，说明尽管TCP已收到了对端发送的紧急方式通告，但尚未收到序号等于紧急指针减 1的字节(图29-17)，此时返回差错代码EWOULDBLOCK，有可能因为发送方发送紧急数据通告时，紧急数据偏移量指向了尚未发送的字节。卷1的图26-7举例说明了这种情况，发送方的数据传输被对端的零窗口通告停止时，常出现这种现象。

19. 返回带外数据字节

237-238 tcp_pulloutofband向应用进程返回存储在t_iobc中的一个字节的带外数据。

20. 更新标志

239-241 如果应用进程已读取了带外数据(而不是仅大致了解带外数据的情况， MSG_PEEK标志置位)，TCP清除HAVE标志，并置位HAD标志。case语句执行到此处时，通过前面的代码可

以肯定，HAVE标志已置位，而HAD标志被清除。置位HAD标志的目的是防止应用进程试图再次读取带外数据。一旦HAD标志置位，在收到新的紧急指针之前，它不会被清除(图29-17)。

代码使用了让人费解的异或运算，而不是简单的

```
tp->t_oobflags = TCPOOB_HADDATA;
```

是为了能够在t_oobflags中定义更多的比特。但Net/3中，实际只用到了上面提及的两个标志比特。

图30-9中的PRU_SENDOOB请求，是在应用进程写入数据并置位MSG_OOB时，由sosend发送的。

tcp_usrreq.c

```
242     case PRU_SENDOOB:
243         if (sbspace(&so->so_snd) < -512) {
244             m_freem(m);
245             error = ENOBUFS;
246             break;
247         }
248         /*
249          * According to RFC961 (Assigned Protocols),
250          * the urgent pointer points to the last octet
251          * of urgent data.  We continue, however,
252          * to consider it to indicate the first octet
253          * of data past the urgent section.
254          * Otherwise, snd_up should be one lower.
255          */
256         sbappend(&so->so_snd, m);
257         tp->snd_up = tp->snd_una + so->so_snd.sb_cc;
258         tp->t_force = 1;
259         error = tcp_output(tp);
260         tp->t_force = 0;
261         break;
```

tcp_usrreq.c

图30-9 tcp_usrreq 函数：PRU_SENDOOB 请求

21. 确认发送缓存中有足够空间并添加新数据

242-247 发送带外数据时，允许应用进程写入数据后，待发送数据量超过发送缓存大小，超出量最多为512字节。插口层的限制要宽松一些，写入带外数据后，最多可超出发送缓存1024字节(图16-24)。调用sbappend向发送缓存末端添加数据。

22. 计算紧急指针

248-257 紧急指针(snd_up)指向写入的最后一个字节之后的字节。图 26-30举例说明了这一点，假定发送缓存为空，应用进程写入 3字节的数据，且置位了MSG_OOB标志。这是考虑到若应用进程置位MSG_OOB标志，且写入的数据量超过 1字节，如果接收方为伯克利系统，则只有最后一个字节会被认为是带外数据。

23. 强制TCP输出

258-261 令t_force等于1，并调用tcp_output。即使收到了对端的零窗口通告， TCP也会发送报文段， URG标志置位，紧急指针偏移量非零。卷 1的图26-7说明了如何向一个关闭的接收窗口发送紧急报文段。

图30-10给出了最后3个请求的处理。

tcp_usrreq.c

```
262     case PRU_SOCKADDR:
263         in_setsockaddr(inp, nam);
264         break;

265     case PRU_PEERADDR:
266         in_setpeeraddr(inp, nam);
267         break;

268         /*
269          * TCP slow timer went off; going through this
270          * routine for tracing's sake.
271          */
272     case PRU_SLOWTIMO:
273         tp = tcp_timers(tp, (int) nam);
274         req |= (int) nam << 8;  /* for debug's sake */
275         break;
```

tcp_usrreq.c

图30-10 tcp_usrreq 函数：PRU_SOCKADDR 、PRU_PEERADDR 和PRU_SLOWTIMO 请求

262-267 getsockname和getpeername系统调用分别发送 PRU_SOCKADDR和PRU_PEERADDR请求。调用in_setsockaddr和in_setpeeraddr函数，从PCB中获取需要信息，存储在addr参数中。

268-275 执行tcp_slowtimo函数会发送PRU_SLOWTIMO函数。如同注释所指出的，tcp_slowtimo不直接调用 tcp_timers的唯一原因是为了能够在函数结尾处调用tcp_trace，跟踪记录定时器超时事件(图30-1)。为了在记录中指明是4个TCP定时器中的哪一个超时，tcp_slowtimo通过nam参数传递了t_timer数组(图25-1)的指针，并左移8位后与请求值(req)逻辑或。trpt程序了解这种做法，并据此完成相应的处理。

30.3 tcp_attach函数

tcp_attach函数，在处理PRU_ATTACH请求(例如，插口系统调用，或者监听插口上收到了新的连接请求)时，由tcp_usrreq调用。图30-11给出了它的代码。

1. 为发送缓存和接收缓存分配资源

361-372 如果还未给插口的发送和接收缓存分配空间，sbreserve将两者都设为8192，即全局变量tcp_sendspace和tcp_recvspace的默认值(图24-3)。

这些默认值是否够用，取决于连接两个传输方向上的 MSS，后者又取决于MTU。例如，[Comer and lin 1994]论证了，如果发送缓存小于 3倍的MSS，则会出现异常，严重降低系统性能。某些实现定义的默认值很大，如 61 444字节，已考虑到这些默认值对性能的影响，尤其对较大的MTU(如FDDI和ATM)更是如此。

2. 分配Internet PCB和TCP控制块

373-377 in_pcballoc分配Internet PCB，而tcp_newtcpcb分配TCP控制块，并将其与对应的PCB相连。

378-384 如果tcp_newtcpcb调用malloc时失败，则执行注释为“ XXX ”的代码。前面已介绍过，PRU_ATTACH请求是插口系统调用或监听插口收到新的连接请求 (sonewconn)的结果。对于后一种情况，插口标志 SS_NOFDREF置位。如果此标志置位， in_pcballoc调用sofree时会释放插口结构。但我们在 tcp_input中看到，除非该函数已完成接收报文段

的处理(图29-27中的dropsocket标志)，否则，不应释放插口结构。因此，调用in_pcbdetach时，应将SS_NOFDREF标志的当前值保存在变量 nofd中，并在tcp_attach返回前重设该标志。

385-386 TCP连接状态初始化为CLOSED。

tcp_usrreq.c

```
361 int
362 tcp_attach(so)
363 struct socket *so;
364 {
365     struct tcpcb *tp;
366     struct inpcb *inp;
367     int      error;

368     if (so->so_snd.sb_hiwat == 0 || so->so_rcv.sb_hiwat == 0) {
369         error = soreserve(so, tcp_sendspace, tcp_recvspace);
370         if (error)
371             return (error);
372     }
373     error = in_pcballoc(so, &tcb);
374     if (error)
375         return (error);
376     inp = sotoinpcb(so);
377     tp = tcp_newtcpcb(inp);
378     if (tp == 0) {
379         int      nofd = so->so_state & SS_NOFDREF;    /* XXX */

380         so->so_state &= ~SS_NOFDREF;     /* don't free the socket yet */
381         in_pcbdetach(inp);
382         so->so_state |= nofd;
383         return (ENOBUFS);
384     }
385     tp->t_state = TCPS_CLOSED;
386     return (0);
387 }
```

tcp_usrreq.c

图30-11 tcp_attach 函数：创建新的TCP插口

30.4 tcp_disconnect函数

图30-12给出的tcp_disconnect函数，准备断开TCP连接。

1. 连接未同步

396-402 如果连接还未进入ESTABLISHED状态(如LISTEN、SYN_SENT或SYN_RCVD)，tcp_close只释放PCB和TCP控制块。无需向对端发送任何报文段，因为连接尚未同步。

2. 硬性断开

403-404 如果连接已同步，且SO_LINGER插口选项置位，拖延时间(SO_LINGER)设为零，则调用tcp_drop丢弃连接。连接不经过 TIME_WAIT，直接更新为 CLOSED，向对端发送RST，释放PCB和TCP控制块。调用close会发送PRU_DISCONNECT请求，丢弃仍在发送或接收缓存中的任何数据。

如果SO_LINGER插口选项置位，且拖延时间非零，则调用 soclose进行处理。

3. 平滑断开

405-406 如果连接已同步，且SO_LINGER选项未设定，或者选项设定且拖延时间不为零，

则执行TCP正常的连接终止步骤。soisdisconnecting设定插口状态。

tcp_usrreq.c

```
396 struct tcpcb *
397 tcp_disconnect(tp)
398 struct tcpcb *tp;
399 {
400     struct socket *so = tp->t_inpcb->inp_socket;

401     if (tp->t_state < TCPS_ESTABLISHED)
402         tp = tcp_close(tp);
403     else if ((so->so_options & SO_LINGER) && so->so_linger == 0)
404         tp = tcp_drop(tp, 0);
405     else {
406         soisdisconnecting(so);
407         sbflush(&so->so_rcv);
408         tp = tcp_usrclosed(tp);
409         if (tp)
410             (void) tcp_output(tp);
411     }
412     return (tp);
413 }
```

tcp_usrreq.c

图30-12 tcp_disconnect 函数：准备断开TCP连接

4. 丢弃滞留的接收数据

407 调用sbflush，丢弃所有滞留在接收缓存中的数据，因为应用进程已关闭了插口。发送缓存中的数据仍保留，tcp_output将试图发送剩余的数据。我们说“试图”，因为不能保证数据还能成功地被发送。在收到并确认这些数据之前，对端可能已崩溃，即使对端的TCP模块能够接收并确认这些数据，在应用程序读取数据之前，系统也可能崩溃。因为本地进程已关闭了插口，即使TCP放弃发送仍滞留在发送缓存中的数据(因为重传定时器最终超时)，也无法向应用进程通告错误。

5. 改变连接状态

408-410 tcp_usrclosed基于连接的当前状态，促使其进入下一状态。通常情况下，连接将转移到FIN_WAIT_1状态，因为连接关闭时一般都处于ESTABLIDHED状态。后面会看到，tcp_usrclosed通常返回当前控制块的指针(tp)。因为状态必须先同步才会执行此处的代码，所以总需要调用tcp_output发送报文段。如果连接从ESTABLISHED转移到FIN_WAIT_2，将发送FIN。

30.5 tcp_usrclosed函数

图30-13给出的这个函数，在PRU_SHUTDOWN处理中，由tcp_disconnect调用。

1. 未收到SYN时的简单关闭

429-434 如果连接上还未收到SYN，则无需发送FIN。新的状态等于CLOSED，tcp_close将释放Internet PCB和TCP控制块。

2. 转移到FIN_WAIT_1状态

435-438 如果连接当前状态等于SYN_RCVD和ESTABLISHED，新的状态将等于FIN_WAIT_1，再次调用tcp_output时，将发送FIN(图24-16中的tcp_outflags值)。

3. 转移到LAST_ACK状态

439-441 如果连接当前状态等于CLOSE_WAIT，新状态等于LAST_ACK，则再次调用tcp_output时，将发送FIN。

443-444 如果连接当前状态等于FIN_WAIT_2或TIME_WAIT，soisdisconnected将正确地标注插口的状态。

tcp_usrreq.c

```
424 struct tcpcb *
425 tcp_usrclosed(tp)
426 struct tcpcb *tp;
427 {
428     switch (tp->t_state) {
429     case TCPS_CLOSED:
430     case TCPS_LISTEN:
431     case TCPS_SYN_SENT:
432         tp->t_state = TCPS_CLOSED;
433         tp = tcp_close(tp);
434         break;
435     case TCPS_SYN_RECEIVED:
436     case TCPS_ESTABLISHED:
437         tp->t_state = TCPS_FIN_WAIT_1;
438         break;
439     case TCPS_CLOSE_WAIT:
440         tp->t_state = TCPS_LAST_ACK;
441         break;
442     }
443     if (tp && tp->t_state >= TCPS_FIN_WAIT_2)
444         soisdisconnected(tp->t_inpcb->inp_socket);
445     return (tp);
446 }
```

tcp_usrreq.c

图30-13 tcp_usrclosed 函数：基于连接关闭的处理进程，将连接转移到下一状态

30.6 tcp_ctloutput函数

tcp_ctloutput函数被getsockopt和setsockopt函数调用，如果它们的描述符参数指明了一个TCP插口，且level不是SOL_SOCKET。图30-14列出了TCP支持的两个插口选项。

选 项 名	变 量	存 取	描 述
TCP_NODELAY	t_flags	读、写	Nagel算法(图26-8)
TCP_MAXSEG	t_maxseg	读、写	TCP将发送的最大报文段长度

图30-14 TCP支持的插口选项

图30-15给出了函数的第一部分。

tcp_usrreq.c

```
284 int
285 tcp_ctloutput(op, so, level, optname, mp)
286 int     op;
287 struct socket *so;
288 int     level, optname;
```

图30-15 tcp_ctloutput 函数：第一部分

```
289 struct mbuf **mp;
290 {
291     int     error = 0, s;
292     struct inpcb *inp;
293     struct tcpcb *tp;
294     struct mbuf *m;
295     int     i;

296     s = splnet();
297     inp = sotoinpcb(so);
298     if (inp == NULL) {
299         splx(s);
300         if (op == PRCO_SETOPT && *mp)
301             (void) m_free(*mp);
302         return (ECONNRESET);
303     }
304     if (level != IPPROTO_TCP) {
305         error = ip_ctloutput(op, so, level, optname, mp);
306         splx(s);
307         return (error);
308     }
309     tp = intotcpcb(inp);
```

tcp_usrreq.c

图30-15 (续)

296-303 函数执行时，处理器优先级设为splnet，inp指向插口的Internet PCB。如果inp为空，且操作类型是设定插口选项，则释放mbuf并返回错误。

304-308 如果*level*(getsockopt和setsockopt系统调用的第二个参数)不等于IPPROTO_TCP，说明操作的是其他协议(如IP)。例如，可以创建一个TCP插口，并设定其IP源选路插口选项。此时应由IP处理这个插口选项，而不是TCP。ip_ctloutput处理命令。

309 如果是对TCP选项进行操作，tp将指向TCP控制块。

函数的剩余部分是一个switch语句，有两个分支：一个处理PRCO_SETOPT(图30-16中给出)，另一个处理PRCO_GETOPT(图30-17中给出)。

tcp_usrreq.c

```
310     switch (op) {

311     case PRCO_SETOPT:
312         m = *mp;
313         switch (optname) {

314         case TCP_NODELAY:
315             if (m == NULL || m->m_len < sizeof(int))
316                     error = EINVAL;
317                 else if (*mtod(m, int *))
318                         tp->t_flags |= TF_NODELAY;
319                 else
320                     tp->t_flags &= ~TF_NODELAY;
321             break;

322         case TCP_MAXSEG:
323             if (m && (i = *mtod(m, int *)) > 0 && i <= tp->t_maxseg)
324                     tp->t_maxseg = i;
325             else
326                 error = EINVAL;
327             break;
```

图30-16 tcp_ctloutput 函数：设定插口选项

```
328         default:
329             error = ENOPROTOOPT;
330             break;
331         }
332         if (m)
333             (void) m_free(m);
334         break;
```

tcp_usrreq.c

图30-16 (续)

315-316 m是一个mbuf，保存了setsockopt的第四个参数。对于两个 TCP插口选项，mbuf中都必须是整数。如果任何一个 mbuf指针为空，或者 mbuf中的数据长度小于整数大小，则返回错误。

1. TCP_NODELAY选项

317-321 如果整数值非零，则置位TF_NODELAY标志，从而取消图26-8中的Negal算法。如果整数值等于0，则使用Negal算法(默认值)，并清除TF_NODELAY标志。

2. TCP_MAXSEG选项

322-327 应用进程只能减少MSS。TCP插口创建时，tcp_newtcpcb初始化t_maxseg为默认值512。当收到对端SYN中包含的MSS选项时，tcp_input调用tcp_mss，t_maxseg最高可等于外出接口的 MTU(减去40字节，IP和TCP首部的默认值)，以太网等于1460。因此，调用插口之后，连接建立之前，应用进程只能以默认值 512为起点，减少MSS。连接建立后，应用进程可以从tcp_mss选取的任何值起，减少MSS。

4.4BSD是伯克利版本中第一次支持 MSS做为插口选项，以前的版本只允许利用getsockopt读取MSS值。

3. 释放mbuf

332-333 释放mbuf链。

图30-17给出了PRCO_GETOPT命令的处理。

```
335     case PRCO_GETOPT:
336         *mp = m = m_get(M_WAIT, MT_SOOPTS);
337         m->m_len = sizeof(int);
338         switch (optname) {
339         case TCP_NODELAY:
340             *mtod(m, int *) = tp->t_flags & TF_NODELAY;
341             break;
342         case TCP_MAXSEG:
343             *mtod(m, int *) = tp->t_maxseg;
344             break;
345         default:
346             error = ENOPROTOOPT;
347             break;
348         }
349         break;
350     }
351     splx(s);
352     return (error);
353 }
```

tcp_usrreq.c

图30-17 tcp_ctloutput 函数：读取插口选项

335-337 两个TCP插口选项都向应用进程返回一个整数值，因此，调用 m_get得到一个mbuf，其长度等于整数长度。

339-341 TCP_NODELAY返回TF_NODELAY标志的当前状态：如果标志未置位 (使用Nagel算法)，则等于0；如果标志置位，则等于TF_NODELAY。

342-344 TCP_MAXSEG选项返回t_maxseg的当前值。前面讨论PRCO_SETOPT命令时曾提到，返回值取决于插口是否已进入连接状态。

30.7 小结

tcp_usrreq函数处理逻辑很简单，因为绝大多数处理都由其他函数完成。 PRU_*xxx*请求是独立于协议的系统调用与TCP协议处理间的桥梁。

tcp_ctlsockopt函数也很简单，因为TCP只支持两个插口选项：使用或取消 Nagel算法，设置或读取最大报文段长度。

习题

30.1 现在，我们已经结束了对 TCP的讨论，如果某个客户执行了正常的 socket、connect、 write (向服务器请求)和read(读取服务器响应)，分别列出客户端和服务器端的处理步骤及TCP状态变迁。

30.2 如果应用进程设定SO_LINGER插口选项，且拖延时间等于0，之后调用close，我们给出了如何调用 tcp_disconnect，从而发送RST。如果应用进程设定了这个插口选项，且拖延时间等于 0，之后进程被某个信号杀死 (kill)，而非调用close，会发生什么？还会发送RST报文段吗？

30.3 图25-4中描述TCP_LINGERTIME时，称之为“ SO_LINGER插口选项的最大秒数 ”。根据图30-2中的代码，这个说法正确吗？

30.4 某个Net/3客户调用socket和connect，主动与服务器建立连接，使用了客户的默认路由。客户主机向服务器发送了 1 129个报文段。假定到达目的地的路由未变，为了这条连接，客户主机需要搜索多少次路由表？解释你的结论。

30.5 找到卷1的附录C中提到的sock程序。把该程序做为服务器运行，读取数据前有停顿(-p)，且有较大的接收缓存。之后在另一个系统中运行同一个程序，但做为客户。通过tcpdump查看数据。确认TCP“ 确认所有其他报文段 ”的属性未出现，服务器送出的ACK全部是延迟ACK。

30.6 修改SO_KEEPALIVE插口选项，从而能够配置每个连接的参数。

30.7 阅读RFC 1122，了解为什么它建议TCP应该允许RST报文段携带数据。修改Net/3代码以实现此功能。

第31章　BPF：BSD 分组过滤程序

31.1　引言

BSD分组过滤程序(BPF)是一种软件设备，用于过滤网络接口的数据流，即给网络接口加上“开关”。应用进程打开/dev/bpf0、/dev/bpf1等等后，可以读取BPF设备。每个应用进程一次只能打开一个BPF设备。

因为每个BPF设备需要8192字节的缓存，系统管理员一般限制 BPF设备的数目。如果open返回EBUSY，说明该设备已被使用，应用进程应该试着打开下一 BPF设备，直到open成功为止。

通过若干ioctl命令，可以配置 BPF设备，把它与某个网络接口相关连，并安装过滤程序，从而能够选择性地接收输入的分组。 BPF设备打开后，应用进程通过读写设备来接收分组，或将分组放入网络接口队列中。

我们将一直使用“分组”，尽管“帧”可能更准确一些，因为 BPF工作在数据链路层，在发送和接收的数据帧中包含了链路层的首部。

BPF设备工作的前提是网络接口必须能够支持 BPF。第3章中提到以太网、 SLIP和环回接口的驱动程序都调用了 bpfattach，用于配置读取 BPF设备的接口。本节中，我们将介绍BPF设备驱动程序是如何组织的，以及数据分组在驱动程序和网络接口之间是如何传递的。

BPF一般情况下用作诊断工具，查看某个本地网络上的流量，卷 1附录A 介绍的tcpdump程序是此类工具中最好的一个。通常情况下，用户感兴趣的是一组指定主机间交互的分组，或者某个特定协议，甚至某个特定 TCP连接上的数据流。BPF设备经过适当配置，能够根据过滤程序的定义丢弃或接受输入的分组。过滤程序的定义类似于伪机器指令， BPF的细节超出了本书的讨论范围，感兴趣的读者请参阅 bpf(4)和[McCanne and Jacobson 1993]。

31.2　代码介绍

下面将要介绍的有关 BPF设备驱动程序的代码，包括两个头文件和一个 C文件，在图31-1中给出。

文　件	描　述
net/bpf.h	BPF常量
net/bpfdesc.h	BPF结构
net/bpf.c	BPF设备支持

图31-1　本章讨论的文件

31.2.1　全局变量

本章用到的全局变量在图31-2中给出。

变 量	数 据 类 型	描 述
bpf_iflist	struct bpf_if *	支持BPF的接口组成的链表
bpf_dtab	struct bpf_d []	BPF描述符数组
bpf_bufsize	int	BPF缓存大小默认值

图31-2 本章用到的全局变量

31.2.2 统计量

图31-3列出了bpf_d结构中为每个活动的BPF设备维护的两个统计量。

bpf_d成员变量	描 述
bd_rcount	从网络接口接收的分组的数目
bd_dcount	由于缓存空间不足而丢弃的分组的数目

图31-3 本章讨论的统计值

本章的其余内容分为4个部分：

- BPF接口结构；
- BPF设备描述符；
- BPF输入处理；和
- BPF输出处理。

31.3 bpf_if结构

BPF维护一个链表，包括所有支持BPF的网络接口。每个接口都由一个bpf_if结构描述，全局指针bpf_iflist指向表中的第一个结构。图31-4给出了BPF接口结构。

bpfdesc.h

```
67 struct bpf_if {
68     struct bpf_if *bif_next;    /* list of all interfaces */
69     struct bpf_d *bif_dlist;    /* descriptor list */
70     struct bpf_if **bif_driverp;    /* pointer into softc */
71     u_int   bif_dlt;            /* link layer type */
72     u_int   bif_hdrlen;         /* length of header (with padding) */
73     struct ifnet *bif_ifp;      /* correspoding interface */
74 };
```

bpfdesc.h

图31-4 bpf_if 结构

67-79 bif_next指向链表中的下一个BPF接口结构。bif_dlist指向另一个链表，包括所有已打开并配置过的BPF设备。

70 如果某个网络接口已配置了BPF设备，即被加上了开关，则bif_driverp将指向ifnet结构中的bpf_if指针。如果网络接口还未加上开关，*bif_driverp为空。为某个网络接口配置BPF设备时，*bif_driverp将指向bif_if结构，从而告诉接口可以开始向BPF传递分组。

71 接口类型保存在bif_dlt中。图31-5中列出了前面提到的几个接口所分别对应的常量值。

bif_dlt	描　　述
DLT_EN10MB	10 Mb以太网接口
DLT_SLIP	SLIP接口
DLT_NULL	环回接口

图31-5 bif_dlt 值

72-74 BPF接受的所有分组都有一个附加的BPF首部。bif_hdrlen等于首部大小。最后，bif_ifp指向对应接口的ifnet结构。

图31-6给出了每个输入分组中附加的bpf_hdr结构。

bpf.h

```
122 struct bpf_hdr {
123     struct timeval bh_tstamp;    /* time stamp */
124     u_long  bh_caplen;           /* length of captured portion */
125     u_long  bh_datalen;          /* original length of packet */
126     u_short bh_hdrlen;           /* length of bpf header (this struct plus
127                                     alignment padding) */
128 };
```

bpf.h

图31-6 bpf_hdr 结构

122-128 bh_tstamp记录了分组被捕捉的时间。bh_caplen等于BPF保存的字节数，bh_datalen等于原始分组中的字节数。bh_headlen等于bpf_hdr的大小加上所需填充字节的长度。它用于解释从BPF设备中读取的分组，应该等同于接收接口的bif_hdrlen。

图31-7给出了bpf_if结构是如何与前述3个接口(le_softc [0]、sl_softc[0]和loif)的ifnet结构建立连接的。

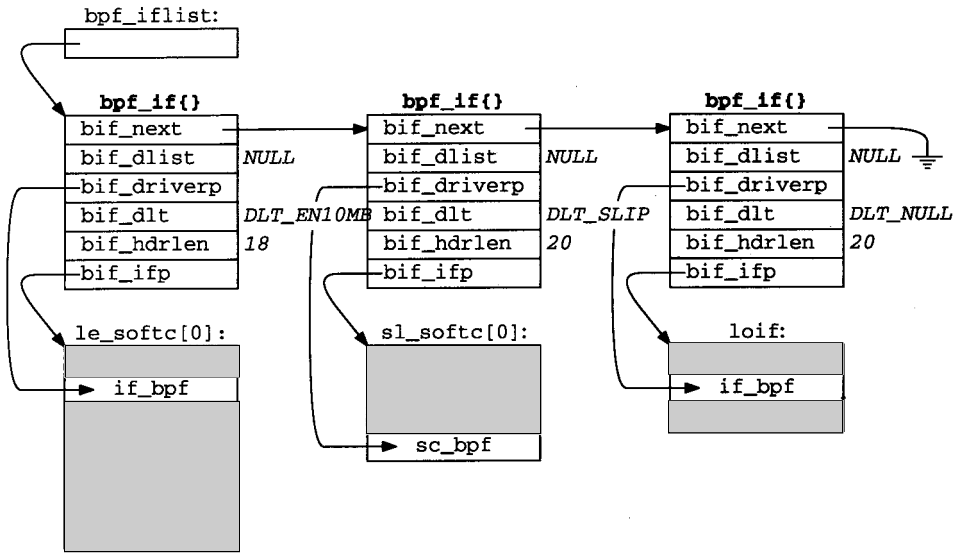


图31-7 bpf_if 和ifnet 结构

注意，bif_driverp指向网络接口的if_bpf和sc_bpf指针，而不是接口结构。

SLIP设备使用sc_bpf，而不是if_bpf。这可能是因为SLIP BPF代码完成时，

if_bpf成员变量还未加入到ifnet结构中。Net/2中的ifnet结构不包括if_bpf成员。

按照各接口驱动程序调用bpfattach时给出的信息，对3个接口初始化链路类型和首部长度成员变量。

第3章介绍了bpfattach被以太网、SLIP和环回接口的驱动程序调用。每个设备驱动程序初始化调用bpfattach时，将构建BPF接口结构链表。图31-8给出了该函数。

bpf.c

```
1053 void
1054 bpfattach(driverp, ifp, dlt, hdrlen)
1055 caddr_t *driverp;
1056 struct ifnet *ifp;
1057 u_int   dlt, hdrlen;
1058 {
1059     struct bpf_if *bp;
1060     int     i;
1061     bp = (struct bpf_if *) malloc(sizeof(*bp), M_DEVBUF, M_DONTWAIT);
1062     if (bp == 0)
1063         panic("bpfattach");

1064     bp->bif_dlist = 0;
1065     bp->bif_driverp = (struct bpf_if **) driverp;
1066     bp->bif_ifp = ifp;
1067     bp->bif_dlt = dlt;

1068     bp->bif_next = bpf_iflist;
1069     bpf_iflist = bp;

1070     *bp->bif_driverp = 0;

1071     /*
1072      * Compute the length of the bpf header.  This is not necessarily
1073      * equal to SIZEOF_BPF_HDR because we want to insert spacing such
1074      * that the network layer header begins on a longword boundary (for
1075      * performance reasons and to alleviate alignment restrictions).
1076      */
1077     bp->bif_hdrlen = BPF_WORDALIGN(hdrlen + SIZEOF_BPF_HDR) - hdrlen;

1078     /*
1079      * Mark all the descriptors free if this hasn't been done.
1080      */
1081     if (!D_ISFREE(&bpf_dtab[0]))
1082         for (i = 0; i < NBPFILTER; ++i)
1083             D_MARKFREE(&bpf_dtab[i]);

1084     printf("bpf: %s%d attached\n", ifp->if_name, ifp->if_unit);
1085 }
```

bpf.c

图31-8 bpfattach 函数

1053-1063 每个支持BPF的设备驱动程序都将调用 bpfattach。第一个参数是保存在bif_driverp的指针(图31-4给出)，第二个参数指向接口的ifnet结构，第三个参数确认数据链路层类型，第四个参数传递分组中的数据链路首部大小，为接口分配一个新的 bpf_if结构。

1. 初始化bpf_if结构

1064-1070 bpf_if结构根据函数的参数进行初始化，并插入到 BPF接口链表，bpf_iflist，的表头。

2. 计算BPF首部大小

1071-1077 设定bif_hdrlen大小，强迫网络层首部 (如IP首部)从一个长字的边界开始。这样可以提高性能，避免为 BPF加入不必要的对齐限制。图 31-9列出了在前述3个接口上，各自捕捉到的BPF分组的总体结构。

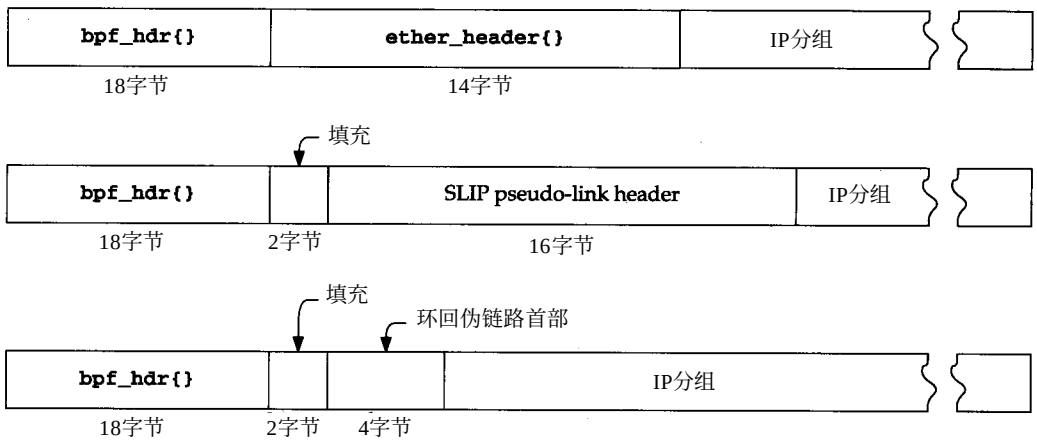


图31-9 BPF分组结构

ether_header结构在图4-10中给出，SLIP伪链路首部在图5-14中给出，而环回接口伪链路首部在图5-28中给出。

请注意，SLIP和环回接口分组需要填充2字节，以强迫IP首部按4字节对齐。

3. 初始化bpf_dtab表

1078-1083 代码初始化图 31-10中给出的 BPF描述符表。注意，仅在第一次调用bpfattach时进行初始化，后续调用将跳过初始化过程。

4. 打印控制台信息

1084-1085 系统向控制台输出一条短信息，宣告接口已配置完毕，可以支持 BPF。

31.4 bpf_d结构

为了能够选择性地接收输入报文，应用进程首先打开一个 BPF设备，调用若干ictl命令规定BPF过滤程序的条件，指明接口、读缓存大小和超时时限。每个 BPF设备都有一个相关的bpf_d结构，如图31-10所示。

45-46 如果同一网络接口上配置了多个BPF设备，与之相应的bpf_d结构将组成一个链表。bd_next指向链表中的下一个结构。

分组缓存

47-52 每个bpf_d结构都有两个分组缓存。输入分组通常保存在 bd_sbuf所对应的缓存(存储缓存)中。另一个缓存要么对应于 bd_fbuf(空闲缓存)，意味着缓存为空；或者对应于bd_hbuf(暂留缓存)，意味着缓存中有分组等待应用进程读取。 bd_slen和bd_hlen分别记

录了保存在存储缓存和暂留缓存中的字节数。

bpfdesc.h

```
45 struct bpf_d {
46     struct bpf_d *bd_next;      /* Linked list of descriptors */
47     caddr_t bd_sbuf;            /* store slot */
48     caddr_t bd_hbuf;            /* hold slot */
49     caddr_t bd_fbuf;            /* free slot */
50     int     bd_slen;            /* current length of store buffer */
51     int     bd_hlen;            /* current length of hold buffer */

52     int     bd_bufsize;         /* absolute length of buffers */

53     struct bpf_if *bd_bif;      /* interface descriptor */
54     u_long  bd_rtout;           /* Read timeout in 'ticks' */
55     struct bpf_insn *bd_filter; /* filter code */
56     u_long  bd_rcount;          /* number of packets received */
57     u_long  bd_dcount;          /* number of packets dropped */

58     u_char  bd_promisc;         /* true if listening promiscuously */
59     u_char  bd_state;           /* idle, waiting, or timed out */
60     u_char  bd_immediate;       /* true to return on packet arrival */
61     u_char  bd_pad;             /* explicit alignment */
62     struct selinfo bd_sel;      /* bsd select info */
63 };
```

bpfdesc.h

图31-10 bpf_d 结构

如果存储缓存已满，它将被连接到bd_hbuf，而空闲缓存将被连接到bd_sbuf。当暂留缓存清空时，它会被连接到 bd_fbuf。宏ROTATE_BUFFERS负责把存储缓存连接到bd_hbuf，空闲缓存连接到bd_sbuf，并清空bd_fbuf。存储缓存满或者应用进程不想再等待更多的分组时调用该宏。

bd_bufsize记录与设备相连的两个缓存的大小，其默认值等于4096(BPF_BUFSIZE)字节。修改内核代码可以改变默认值大小，或者通过 BIOCSBLEN ioctl命令改变某个特定BPF设备的bd_bufsize。BIOCGBLEN命令返回bd_bufsize的当前值，其最大值不超过32768 (BPF_MAXBUFSIZE)字节，最小值为32 (BPF_MINBUFSIZE)字节。

53-57 bd_bif指向BPF设备所对应的 bpf_if结构。 BIOCSETIF命令可指明设备。bd_rtout是等待分组时，延迟的滴答数。 bd_filter指向BPF设备的过滤程序代码。两个统计值，应用进程可通过 BIOCGSTATS命令读取，分别保存在 bd_rcount和bd_dcount中。

58-63 bd_promisc通过BIOCPROMISC命令置位，从而使接口工作在混杂(promiscuous)状态。bd_state未使用。bd_immediate通过BIOCIMMEDIATE命令置位，促使驱动程序收到分组后即返回，不再等待暂留缓存填满。 bd_pad填弃bpf_d结构，从而与长字边界对齐。bd_sel保存的selinfo结构，可用于select系统调用。我们不准备介绍如何对BPF设备使用select系统调用，16.13节已介绍了select的一般用法。

31.4.1 bpfopen函数

应用进程调用open，试图打开一个BPF设备时，该调用将被转到bpfopen(图31-11)。

256-263 系统编译时，BPF设备的数目受到NBPFILTER的限制。如果设备的最小设备号大

于NBPFILTER，则返回ENXIO，这是因为系统管理员创建的/dev/bpfx项数大于NBPFILTER的值。

bpf.c

```
256 int
257 bpfopen(dev, flag)
258 dev_t   dev;
259 int     flag;
260 {
261     struct bpf_d *d;

262     if (minor(dev) >= NBPFILTER)
263         return (ENXIO);
264     /*
265      * Each minor can be opened by only one process.  If the requested
266      * minor is in use, return EBUSY.
267      */
268     d = &bpf_dtab[minor(dev)];
269     if (!D_ISFREE(d))
270         return (EBUSY);

271     /* Mark "free" and do most initialization. */
272     bzero((char *) d, sizeof(*d));
273     d->bd_bufsize = bpf_bufsize;

274     return (0);
275 }
```

bpf.c

图31-11 bpfopen 函数

分配bpf_d结构

264-275 同一时间内，一个应用进程只能访问一个BPF设备。如果bpf_d结构已被激活，则返回EBUSY。应用程序，如tcpdump，收到此返回值时，会自动寻找下一个设备。如果该设备已存在，最小设备号所指定的bpf_dtab表中的项被清除，分组缓存大小复位为默认值。

31.4.2 bpfioctl函数

设备打开后，可通过ioctl命令进行配置。图31-12总结了与BPF设备有关的ioctl命令。图31-13给出了bpfioctl函数，只列出BIOCSETF和BIOCSETIF的处理代码，其他未涉及到的ioctl命令则被忽略。

命　令	第三个参数	函　数	描　述
FIONREAD	u_int	bpfioctl	返回暂留缓存和存储缓存中的字节数
BIOCGBLEN	u_int	bpfioctl	返回分组缓存大小
BIOCSBLEN	u_int	bpfioctl	设定分组缓存大小
BIOCSETF	struct bpf_program	bpf_setf	安装BPF程序
BIOCFLUSH		reset_d	丢弃挂起分组
BIOCPROMISC		ifpromisc	设定混杂方式
BIOCGDLT	u_int	bpfioctl	返回bif_dlt
BIOCGETIF	struct ifreq	bpf_ifname	返回所属接口的名称
BIOCSETIF	struct ifreq	bpf_setif	为网络接口添加设备
BIOCSRTIMEOUT	struct timeval	bpfioctl	设定“读”操作的超时时限
BIOCGRTIMEOUT	struct timeval	bpfioctl	返回“读”操作的超时时限
BIOCGSTATS	struct bpf_stat	bpfioctl	返回BPF统计值
BIOCIMMEDIATE	u_int	bpfioctl	设定立即方式
BIOCVERSION	struct bpf_version	bpfioctl	返回BPF版本信息

图31-12 BPF ioctl 命令

bpf.c

```
501 int
502 bpfioctl(dev, cmd, addr, flag)
503 dev_t   dev;
504 int     cmd;
505 caddr_t addr;
506 int     flag;
507 {
508     struct bpf_d *d = &bpf_dtab[minor(dev)];
509     int     s, error = 0;
510     switch (cmd) {
511         /*
512          * Set link layer read filter.
513          */
514     case BIOCSETF:
515         error = bpf_setf(d, (struct bpf_program *) addr);
516         break;
517         /*
518          * Set interface.
519          */
520     case BIOCSETIF:
521         error = bpf_setif(d, (struct ifreq *) addr);
522         break;

            /* other ioctl commands from Figure 31.12 */

668     default:
669         error = EINVAL;
670         break;
671     }
672     return (error);
673 }
```

bpf.c

图31-13 bpfioctl 函数

501-509 与bpfopen类似，通过最小设备号从bpf_dtab表中选取相应的bpf_d结构。整个命令处理是一个大的switch/case语句。我们给出了两个命令，BIOCSETF和BIOCSETIF，以及default子句。

510-522 bpf_setf函数安装由addr指向的过滤程序，bpf_setif建立起指定名称接口与bpf_d结构间的对应关系。本书中没有给出bpf_setf的实现代码。

668-673 如果命令未知，则返回EINVAL。

图31-14的例子中，bpf_setif已把bpf_d结构连接到LANCE接口上。

图中，bif_dlist指向bpf_dtab[0]，以太网接口描述符链表中的第一个也是仅有的一个描述符。在bpf_dtab[0]中，bd_sbuf和bd_hbuf成员分别指向存储缓存和暂留缓存。两个缓存大小都等于4096(bd_bufsize)字节。bd_bif回指接口的bpf_if结构。

ifnet结构(le_softc[0])中的if_bpf也指回bpf_if结构。如图4-19和图4-11所示，如果if_bpf非空，则驱动程序开始调用bpf_tap，向BPF设备传递分组。

图31-15接着图31-10，给出了打开第二个BPF设备，并连接到同一个以太网网络接口后的各结构变量的状态。

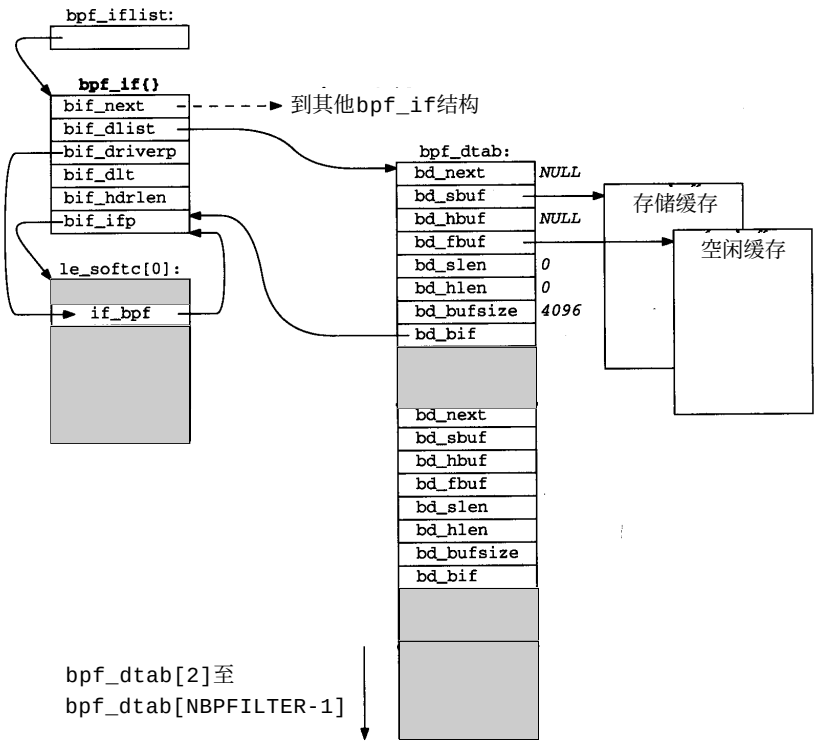


图31-14 连接到以太网接口的BPF设备

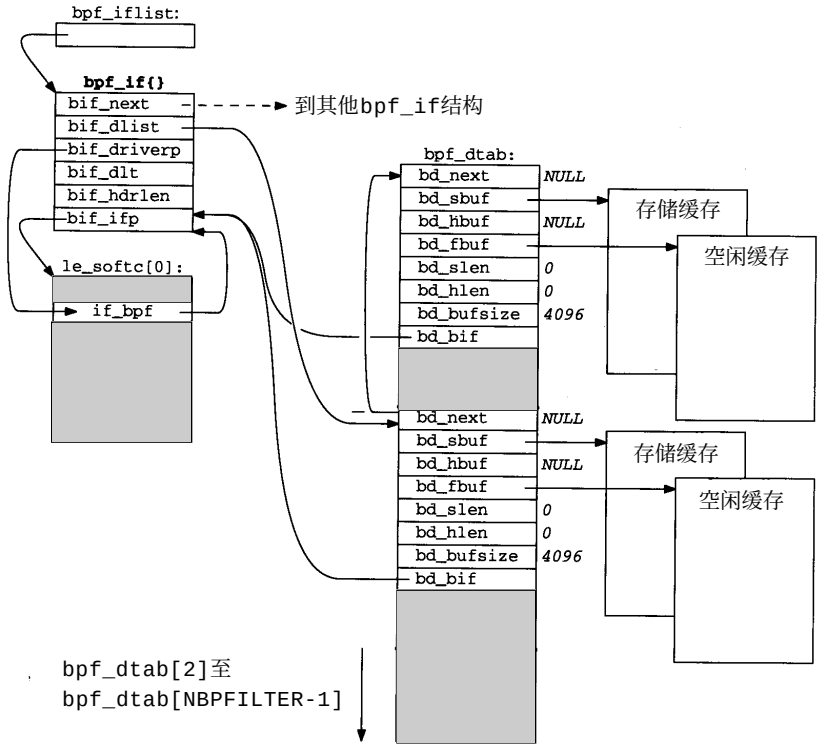


图31-15 连接到以太网接口的两个BPF设备

第二个BPF设备打开时，在bpf_dtab表中分配一个新的bpf_d结构，本例中为bpf_dtab[1]。因为第二个BPF设备也连接到同一个以太网接口，bif_dlist指向bpf_dtab[1]，并且bpf_dtab[1].bd_next指向bpf_dtab[0]，即以太网上对应的第一个BPF描述符。系统为新的描述符结构分别分配存储缓存和暂留缓存。

31.4.3 bpf_setif函数

bpf_setif函数，负责建立BPF描述符与网络接口间的连接，如图31-16所示。

bpf.c

```
721 static int
722 bpf_setif(d, ifr)
723 struct bpf_d *d;
724 struct ifreq *ifr;
725 {
726     struct bpf_if *bp;
727     char   *cp;
728     int     unit, s, error;

729     /*
730      * Separate string into name part and unit number.  Put a null
731      * byte at the end of the name part, and compute the number.
732      * If the a unit number is unspecified, the default is 0,
733      * as initialized above.  XXX This should be common code.
734      */
735     unit = 0;
736     cp = ifr->ifr_name;
737     cp[sizeof(ifr->ifr_name) - 1] = '\0';
738     while (*cp++) {
739         if (*cp >= '0' && *cp <= '9') {
740             unit = *cp - '0';
741             *cp++ = '\0';
742             while (*cp)
743                 unit = 10 * unit + *cp++ - '0';
744             break;
745         }
746     }
747     /*
748      * Look through attached interfaces for the named one.
749      */
750     for (bp = bpf_iflist; bp != 0; bp = bp->bif_next) {
751         struct ifnet *ifp = bp->bif_ifp;

752         if (ifp == 0 || unit != ifp->if_unit
753             || strcmp(ifp->if_name, ifr->ifr_name) != 0)
754             continue;
755         /*
756          * We found the requested interface.
757          * If it's not up, return an error.
758          * Allocate the packet buffers if we need to.
759          * If we're already attached to requested interface,
760          * just flush the buffer.
761          */
762         if ((ifp->if_flags & IFF_UP) == 0)
763             return (ENETDOWN);
```

图31-16 bpf_setif 函数

```
764         if (d->bd_sbuf == 0) {
765             error = bpf_allocbufs(d);
766             if (error != 0)
767                 return (error);
768         }
769         s = splimp();
770         if (bp != d->bd_bif) {
771             if (d->bd_bif)
772                 /*
773                  * Detach if attached to something else.
774                  */
775                 bpf_detachd(d);

776             bpf_attachd(d, bp);
777         }
778         reset_d(d);
779         splx(s);
780         return (0);
781     }
782     /* Not found. */
783     return (ENXIO);
784 }
```

bpf.c

图31-16 (续)

721-746 bpf_setif的第一部分完成ifreq结构(图4-23)中接口名的正文与数字部分的分离，数字部分保存在unit中。例如，如果ifr_name的头4字节为“sl1\0”，代码执行完毕后，将等于“sl\0\0”，且unit等于1。

1. 寻找匹配的ifnet结构

747-754 for循环用于在支持BPF的接口(bpf_iflist中)中查找符合ifreq定义的接口。

755-768 如果未找到匹配的接口，则返回ENETDOWN。如果接口存在，bpf_allocate为bpf_d分配空闲缓存和存储缓存，如果它们还未被分配的话。

2. 连接bpf_d结构

769-777 如果BPF设备还未与网络接口建立连接关系，或者连接的网络接口不是ifreq中指定的接口，则调用bpf_detachd丢弃原先的接口(如果存在)，并调用bpf_attachd将其连接到新的接口上。

778-784 reset_d复位分组缓存，丢弃所有在应用进程中等待的分组。函数返回0，说明处理成功；或者ENXIO，说明未找到指定接口。

31.4.4 bpf_attachd函数

图31-17给出的bpf_attachd函数，建立起BPF描述符与BPF设备和网络接口间的对应关系。

bpf.c

```
189 static void
190 bpf_attachd(d, bp)
191 struct bpf_d *d;
192 struct bpf_if *bp;
193 {
194     /*
195      * Point d at bp, and add d to the interface's list of listeners.
```

图31-17 bpf_attachd 函数

```
196         * Finally, point the driver's bpf cookie at the interface so
197         * it will divert packets to bpf.
198         */
199        d->bd_bif = bp;
200        d->bd_next = bp->bif_dlist;
201        bp->bif_dlist = d;

202        *bp->bif_driverp = bp;
203    }
```

bpf.c

图31-17 (续)

189-203 首先，令bd_bif指向网络接口的BPF接口结构。接着，bpf_d结构被插入到与设备对应的bpf_d结构链表的头部。最后，改变网络接口中的BPF指针，指向当前BPF结构，从而促使接口向BPF设备传递分组。

31.5 BPF的输入

一旦BPF设备打开并配置完毕，应用进程就通过read系统调用从接口中接收分组。BPF过滤程序复制输入分组，因此，不会干扰正常的网络处理。输入分组保存在与BPF设备相连的存储缓存和暂留缓存中。

31.5.1 bpf_tap函数

下面列出了图4-11中LANCE设备驱动程序调用bpf_tap的代码，并利用这一调用介绍bpf_tap函数。图4-11中的调用如下：

```
bpf_tap(le->sc_if.if_bpf, buf, len + sizeof(struct ether_header));
```

图31-18给出了bpf_tap函数。

bpf.c

```
869 void
870 bpf_tap(arg, pkt, pktlen)
871 caddr_t arg;
872 u_char *pkt;
873 u_int   pktlen;
874 {
875     struct bpf_if *bp;
876     struct bpf_d *d;
877     u_int   slen;
878     /*
879      * Note that the ipl does not have to be raised at this point.
880      * The only problem that could arise here is that if two different
881      * interfaces shared any data.  This is not the case.
882      */
883     bp = (struct bpf_if *) arg;
884     for (d = bp->bif_dlist; d != 0; d = d->bd_next) {
885         ++d->bd_rcount;
886         slen = bpf_filter(d->bd_filter, pkt, pktlen, pktlen);
887         if (slen != 0)
888             catchpacket(d, pkt, pktlen, slen, bcopy);
889     }
890 }
```

bpf.c

图31-18 bpf_tap 函数

869-882 第一个参数是指向bpf_if结构的指针，由bpfattach设定。第二个参数是指向进入分组的指针，包括以太网首部。第三个参数等于缓存中包含的字节数，本例中，等于以太网首部(14字节)大小加上以太网帧的数据部分。

向一个或多个BPF设备传递分组

883-890 for循环遍历连接到网络接口的BPF设备链表。对每个设备，分组被递交给bpf_filter。如果过滤程序接受了分组，它返回捕捉到的字节数，并调用catchpacket复制分组。如果过滤程序拒绝了分组，slen等于0，循环继续。循环终止时，bpf_tap返回。这一机制确保了同一网络接口上对应了多个BPF设备时，每个设备都能拥有一个独立的过滤程序。

环回驱动程序调用bpf_mtap，向BPF传递分组。这个函数与bpf_tap类似，然而是在mbuf链，而不是在一个内存的连续区域中复制分组。本书中不介绍这个函数。

31.5.2 catchpacket函数

图31-18中，过滤程序接受了分组后，将调用catchpacket，图31-19给出了这个函数。

bpf.c

```
946 static void
947 catchpacket(d, pkt, pktlen, snaplen, cpfn)
948 struct bpf_d *d;
949 u_char *pkt;
950 u_int   pktlen, snaplen;
951 void    (*cpfn) (const void *, void *, u_int);
952 {
953     struct bpf_hdr *hp;
954     int     totlen, curlen;
955     int     hdrlen = d->bd_bif->bif_hdrlen;
956     /*
957      * Figure out how many bytes to move.  If the packet is
958      * greater or equal to the snapshot length, transfer that
959      * much.  Otherwise, transfer the whole packet (unless
960      * we hit the buffer size limit).
961      */
962     totlen = hdrlen + min(snaplen, pktlen);
963     if (totlen > d->bd_bufsize)
964         totlen = d->bd_bufsize;
965     /*
966      * Round up the end of the previous packet to the next longword.
967      */
968     curlen = BPF_WORDALIGN(d->bd_slen);
969     if (curlen + totlen > d->bd_bufsize) {
970         /*
971          * This packet will overflow the storage buffer.
972          * Rotate the buffers if we can, then wakeup any
973          * pending reads.
974          */
975         if (d->bd_fbuf == 0) {
976             /*
977              * We haven't completed the previous read yet,
978              * so drop the packet.
979              */
980             ++d->bd_dcount;
```

图31-19 catchpacket 函数

```
981             return;
982         }
983         ROTATE_BUFFERS(d);
984         bpf_wakeup(d);
985         curlen = 0;
986     } else if (d->bd_immediate)
987         /*
988          * Immediate mode is set.  A packet arrived so any
989          * reads should be woken up.
990          */
991         bpf_wakeup(d);
992     /*
993      * Append the bpf header.
994      */
995     hp = (struct bpf_hdr *) (d->bd_sbuf + curlen);
996     microtime(&hp->bh_tstamp);
997     hp->bh_datalen = pktlen;
998     hp->bh_hdrlen = hdrlen;
999     /*
1000     * Copy the packet data into the store buffer and update its length.
1001     */
1002    (*cpfn) (pkt, (u_char *) hp + hdrlen, (hp->bh_caplen = totlen - hdrlen));
1003    d->bd_slen = curlen + totlen;
1004 }
```

bpf.c

图31-19 (续)

946-955 catchpacket的参数包括：d，指向BPF设备结构的指针；pkt，指向进入分组的通用指针；pktlen，分组被接收时的长度；snaplen，从分组中保存下来的字节数；cpfn，函数指针，把分组从pkt中复制到一块连续内存中。如果分组已经保存连续内存中，则cptn等于bcopy。如果分组被保存在mbuf中(pkt指向mbuf链表中的第一个mbuf，如环回驱动程序)，则cptn等于bpf_mcopy。

956-964 除了链路层首部和分组，catchpacket为每个分组添加bpf_hdr。从分组中保存的字节数等于snaplen和pktlen中较小的一个。处理过的分组和bpf_hdr必须能放入分组缓存中(bd_bufsize字节)。

1. 分组能否放入缓存

965-985 curlen等于存储缓存中已有的字节数加上所需的填充字节，以保证下一分组能从长字边界处开始存放。如果进入分组无法放入剩余的缓存空间，说明存储缓存已满。如果空闲缓存不可用(如应用进程正从暂留缓存中读取数据)，则进入分组被丢弃。如是空闲缓存可用，则调用ROTATE_BUFFERS宏轮转缓存，并通过bpf_wakeup唤醒所有等待输入数据的应用进程。

2. 立即方式处理

986-991 如果设备处于立即方式，则唤醒所有等待进程以处理进入分组——内核中没有分组的缓存。

3. 添加BPF 首部

992-1004 当前时间(microtime)、分组长度和首部长度均保存在bpf_hdr中。调用cptf所指的函数，把分组复制到存储缓存，并更新存储缓存的长度。因为在把分组从设备缓存传送到某个mbuf链表之前，bpf_tab已由leread直接调用，接收时间戳近似等于实际的接收

时间。

31.5.3 bpfread函数

内核把针对BPF设备的read转交给bpfread处理。通过BIOCSRTIMEOUT命令，BPF支持限时读取。这个“特性”也可通过select系统调用来实现，但至少tcpdump还是采用了BIOCSRTIMEOUT，而非select。应用进程提供一个读缓存，能够与设备的暂留缓存大小相匹配。BICOGBLEN命令返回缓存大小。一般情况下，读操作在存储缓存已满时返回。内核轮转缓存，把存储缓存转给暂留缓存，后者在read系统调用时被复制到应用进程提供的读缓存，同时BPF设备继续向存储缓存中存放进入分组。图31-20给出了bpfread。

bpf.c

```
344 int
345 bpfread(dev, uio)
346 dev_t   dev;
347 struct uio *uio;
348 {
349     struct bpf_d *d = &bpf_dtab[minor(dev)];
350     int     error;
351     int     s;

352     /*
353      * Restrict application to use a buffer the same size as
354      * as kernel buffers.
355      */
356     if (uio->uio_resid != d->bd_bufsize)
357         return (EINVAL);

358     s = splimp();
359     /*
360      * If the hold buffer is empty, then do a timed sleep, which
361      * ends when the timeout expires or when enough packets
362      * have arrived to fill the store buffer.
363      */
364     while (d->bd_hbuf == 0) {
365         if (d->bd_immediate && d->bd_slen != 0) {
366             /*
367              * A packet(s) either arrived since the previous
368              * read or arrived while we were asleep.
369              * Rotate the buffers and return what's here.
370              */
371             ROTATE_BUFFERS(d);
372             break;
373         }
374         error = tsleep((caddr_t) d, PRINET | PCATCH, "bpf", d->bd_rtout);
375         if (error == EINTR || error == ERESTART) {
376             splx(s);
377             return (error);
378         }
379         if (error == EWOULDBLOCK) {
380             /*
381              * On a timeout, return what's in the buffer,
382              * which may be nothing.  If there is something
383              * in the store buffer, we can rotate the buffers.
384              */
385             if (d->bd_hbuf)
```

图31-20 bpfread 函数

```
386                 /*
387                  * We filled up the buffer in between
388                  * getting the timeout and arriving
389                  * here, so we don't need to rotate.
390                  */
391                 break;
392             if (d->bd_slen == 0) {
393                 splx(s);
394                 return (0);
395             }
396             ROTATE_BUFFERS(d);
397             break;
398         }
399     }
400     /*
401      * At this point, we know we have something in the hold slot.
402      */
403     splx(s);

404     /*
405      * Move data from hold buffer into user space.
406      * We know the entire buffer is transferred since
407      * we checked above that the read buffer is bpf_bufsize bytes.
408      */
409     error = uiomove(d->bd_hbuf, d->bd_hlen, UIO_READ, uio);

410     s = splimp();
411     d->bd_fbuf = d->bd_hbuf;
412     d->bd_hbuf = 0;
413     d->bd_hlen = 0;
414     splx(s);

415     return (error);
416 }
```

bpf.c

图31-20 (续)

344-357 通过最小设备号在bpf_dtab中寻找相应的BPF设备。如果读缓存不能匹配BPF设备缓存的大小，则返回EINVAL。

1. 等待数据

358-364 因为多个应用进程能够从同一个BPF设备中读取数据，如果有某个进程已先读取了数据，while循环将强迫读操作继续。如果暂留缓存中存在数据，循环被跳过。这与两个应用进程通过两个不同的BPF设备过滤同一个网络接口的情况(见习题31.2)是不同的。

2. 立即方式

365-373 如果设备处于立即方式，且存储缓存中有数据，则轮回缓存，while循环被终止。

3. 无可用的分组

374-384 如果设备不处于立即方式，或者存储缓存中没有数据，则应用进程进入休眠状态，直到某个信号到达，读定时器超时，或者有数据到达暂留缓存。如果有信号到达，则返回EINTR或ERESTART。

记住，应用进程不会见到ERESTART，因为syscall函数将处理这一错误，且不会向应用进程返回这一错误。

4. 查看暂留缓存

385-391 如果定时器超时，且暂留缓存中存在数据，则循环终止。

5. 查看存储缓存

392-399 如果定时器超时，且存储缓存中没有数据，则read返回0。应用进程执行限时读取时，必须考虑到这种情况。如果定时器超时，且存储缓存中存在数据，则把存储缓存转给暂留缓存，循环终止。

如果tsleep返回正常且存在数据，同时while循环测试失败，则循环终止。

6. 分组可用

400-416 循环终止时，暂留缓存中已有数据。uiomove从暂留缓存中移出bd_hlen个字节，交给应用进程。把暂留缓存转给空闲缓存，清除缓存计数器，函数返回。 uiomove调用前的注释指出，uiomove通常能向应用进程复制bd_hlen字节的数据，因为前面已检查过读缓存大小，确保它大于BPF设备缓存的最大值，即bd_bufsize。

31.6 BPF的输出

最后，我们讨论如何向带有BPF设备的网络接口输出队列中添加分组。首先，应用进程必须构造完整的数据链路帧。对以太网而言，包括源和目的主机的硬件地址和数据帧类型(图4-8)。内核在把它放入接口的输出队列前不会修改链路帧。

bpfwrite函数

内核把应用进程的write系统调用转给图31-21给出的bpfwrite处理，数据帧被传给BPF设备。

bpf.c

```
437 int
438 bpfwrite(dev, uio)
439 dev_t   dev;
440 struct uio *uio;
441 {
442     struct bpf_d *d = &bpf_dtab[minor(dev)];
443     struct ifnet *ifp;
444     struct mbuf *m;
445     int     error, s;
446     static struct sockaddr dst;
447     int     datlen;

448     if (d->bd_bif == 0)
449         return (ENXIO);

450     ifp = d->bd_bif->bif_ifp;

451     if (uio->uio_resid == 0)
452         return (0);

453     error = bpf_movein(uio, (int) d->bd_bif->bif_dlt, &m, &dst, &datlen);
454     if (error)
455         return (error);

456     if (datlen > ifp->if_mtu)
457         return (EMSGSIZE);

458     s = splnet();
459     error = (*ifp->if_output) (ifp, m, &dst, (struct rtentry *) 0);
460     splx(s);
```

图31-21 bpfwrite 函数

```
461     /*
462      * The driver frees the mbuf.
463      */
464     return (error);
465 }
```
bpf.c

图31-21 (续)

1. 检查设备号

437-449 通过最小的设备号选择BPF设备，它必须已连接到某个网络接口。如果还没有，则返回ENXIO。

2. 向mbuf链中复制数据

450-457 如果write给出的写入数据长度等于0，则立即返回0。bpf_movein从应用进程复制数据到一个mbuf链表，并基于由bif_dlt传递的接口类型计算去除了链路层首部后的分组长度，并在datlen中返回该值。它还在dst中返回一个已初始化过的sockaddr结构。对以太网而言，这个地址结构的类型应该等于AF_UNSPEC，说明mbuf链中保存了外出数据帧的数据链路层首部。如果分组大于接口的MTU，则返回EMSGSIZE。

3. 分组排队

458-465 调用ifnet结构中指定的if_output函数，得到的mbuf链被提交给网络接口。对于以太网，if_output等于ether_output。

31.7 小结

本章中，我们讨论了如何配置BPF设备，如何向BPF设备递交进入数据帧，及如何在一个BPF设备上传送外出数据帧。

一个网络接口可以有多个BPF设备，每个BPF设备都有自己的过滤程序。存储缓存和暂留缓存最大限度地减少了应用进程为了处理进入数据帧而调用read的次数。

本章中只介绍了BPF的一些主要特性。有关BPF设备过滤程序代码的详细情况和其他一些特性，感兴趣的读者请参阅源代码和Net/3手册。

习题

31.1 为什么在分组存入BPF缓存之前，就能在catchpacket中调用bpf_wakeup？

31.2 图31-20中，我们提到可能会有两个进程在同一 BPF设备上等待数据。图31-11中，我们指出同一时间只能有一个应用进程可以打开一个特定的 BPF设备。为什么这两种说法都正确呢？

31.3 如果BIOCSETIF命令中指定的设备不支持BPF，会发生什么现象？

第32章 原 始 IP

32.1 引言

应用进程在Internet域中创建一个SOCK_RAW类型的插口，就可以利用原始IP层。一般有下列3种用法：

1) 应用进程可利用原始插口发送和接收ICMP和IGMP报文。
   Ping程序利用这种类型的插口，发送ICMP回显请求和接收ICMP回显应答。
   有些选路守护程序，利用这一特性跟踪通常由内核处理的ICMP重定向报文段。我们在19.7节中提到，Net/3处理重定向报文段时，会在需重定向的插口上生成RTM_REDIRECT消息，从而无需利用原始插口的这一功能。
   这个特性还用于实现基于ICMP的协议，如路由通告和路由请求(卷1的9.6节)，它们需用到ICMP，不过最好由应用进程，而不是内核完成相应处理。
   多播路由守护程序利用原始IGMP插口，发送和接收IGMP报文。
2) 应用进程可利用原始插口构造自己的IP首部。路由跟踪程序利用这一特性生成自己的UDP数据报，包括IP和UDP首部。
3) 应用进程可利用原始插口读写内核不支持的IP协议的IP数据报。
   gated程序利用这一特性支持基于IP的路由协议：EGP、HELLO和OSPF。
   这种类型的原始插口还可用于设计基于IP的新的运输层协议，而无需增加对内核的支持。调试应用进程代码比调试内核代码容易得多。

本章介绍原始IP插口的实现。

32.2 代码介绍

图32-1给出的C文件中包含了5个原始IP处理函数。

文　件	描　述
netinet/raw_ip.c	原始IP处理函数

图32-1 本章讨论的文件

图32-2给出了5个原始IP函数与其他内核函数间的关系。

带阴影的椭圆表示我们在本章中将要讨论的5个函数。请注意，原始IP函数名中的前缀“rip”表示“原始IP (Raw IP)”，而不是“选路信息协议(Routing Information Protocol)”，后者的缩写也是RIP。

32.2.1 全局变量

本章中用到4个全局变量，如图32-3所示。

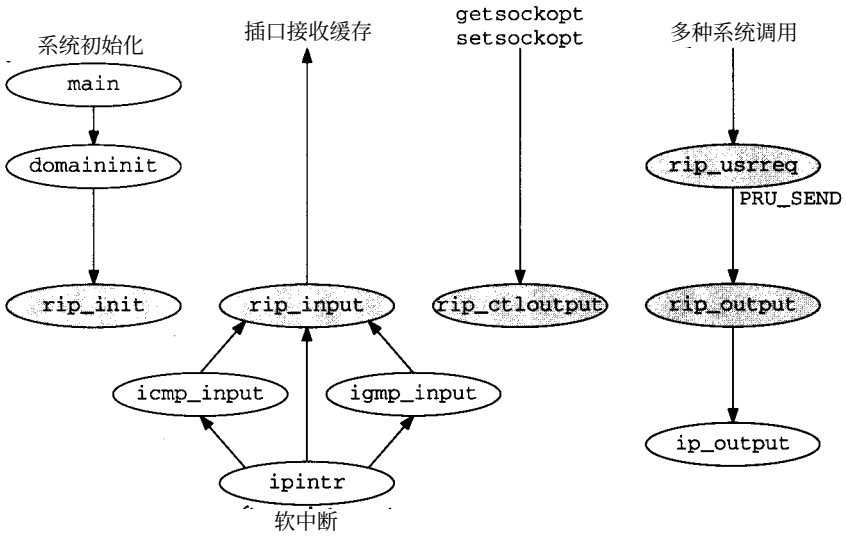


图32-2 原始IP函数与其他内核函数间的关系

变 量	数据类型	描 述
rawinpcb	struct inpcb	原始IP的Internet PCB链表表头
ripsrc	struct sockaddr_in	在输入中包含发送方的IP地址
rip_recvspace	u_long	插口接收缓存大小默认值，8192字节
rip_sendspace	u_long	插口发送缓存大小默认值，8192字节

图32-3 本章介绍的全局变量

32.2.2 统计量

原始IP在ipstat结构(图8-4)中维护两个计数器，如图32-4所示。

ipstat成员变量	描 述	SNMP变量使用
ips_noproto	协议类型未知或不支持的数据报数目	•
ips_rawout	生成的原始IP数据报总数	

图32-4 ipstat 结构中维护的原始IP统计量

图8-6给出了如何在SNMP中使用ips_noproto计数器。图8-5给出了这两个计数器输出值的例子。

32.3 原始IP的protosw结构

与所有其他协议不同，inetsw数组有多条记录都可以读写原始IP。inetsw结构中有4个记录的插口类型都等于SOCK_RAW，但协议类型则各不相同：

- IPPROTO_ICMP(协议值1)；
- IPPROTO_IGMP(协议值2)；
- IPPROTO_RAW(协议值255)；和

- 原始IP通配记录(协议值0)。

其中ICMP和IGMP，前面已介绍过(图11-12和图13-9)。四项记录间的区别总结如下：

- 如果应用进程创建了一个原始插口 (SOCK_RAW)，协议值非零(socket的第三个参数)，并且如果协议值等于IPPROTO_ICMP、IPPROTO_IGMP或IPPROTO_RAW，则会使用对应的protosw记录。
- 如果应用进程创建了一个原始插口 (SOCK_RAW)，协议值非零，但内核不支持该协议，pffindproto会返回协议值为0的通配记录，从而允许应用进程处理内核不支持的 IP协议，而无需修改内核代码。

我们在7.8节中提到，ip_protox数组中的所有未知记录都指向IPPROTO_RAW，它的协议转换类型如图32-5所示。

成 员	inetsw[3]	描 述
pr_type	*SOCK_RAW*	原始插口
pr_domain	*& inetdomain*	属于Internet域的原始IP
pr_protocol	*IPPROTO_RAW*(255)	出现在IP首部的ip_p字段
pr_flags	*PR_ATOMIC|PR_ADDR*	插口层标志，不用于协议处理
pr_input	*rip_input*	从IP层接收报文段
pr_output	*0*	原始IP不使用
pr_ctlinput	*0*	原始IP不使用
pr_ctloutput	*rip_ctlinput*	响应应用进程的管理请求
pr_usrreq	*rip_usrreq*	响应应用进程的通信请求
pr_init	*0*	原始IP不使用
pr_fasttimo	*0*	原始IP不使用
pr_slowtimo	*0*	原始IP不使用
pr_drain	*0*	原始IP不使用
pr_sysctl	*0*	原始IP不使用

图32-5 原始IP的protosw 结构

本章中我们将介绍3个以rip_开头的函数，此外还大致提一下rip_output函数，它没有出现在协议转换记录中，但输出原始IP报文段时，rip_usrreq将会调用它。

第五个原始IP函数，rip_init，只出现在通配处理记录中。初始化函数只能调用一次，所以它既可以出现在IPPROTP_RAW记录中，也可以放在通配记录中。

不过，图32-5中并没有说明其他协议 (ICMP和IGMP)，在它们自己的protosw结构中也用到了一些原始IP函数。图32-6对4个SOCK_RAW协议各自protosw结构的相关成员变量做了一个比较。为了强调指出彼此间的区别，不同之处都用黑体字标出。

protosw 记录	SOCK_RAW 协议类型			
	IPPROTO_ICMP (1)	IPPROTO_IGMP (2)	IPPROTO_RAW (255)	通配(0)
pr_input	***icmp_input***	***igmp_input***	***rip_input***	***rip_input***
pr_output	*rip_output*	*rip_output*	*rip_output*	*rip_output*
pr_ctloutput	*rip_ctloutput*	*rip_ctloutput*	*rip_ctloutput*	*rip_ctloutput*
pr_usrreq	*rip_usrreq*	*rip_usrreq*	*rip_usrreq*	*rip_usrreq*
pr_init	*0*	***igmp_init***	*0*	***rip_init***
pr_sysctl	***icmp_sysctl***	*0*	*0*	*0*
pr_fasttimo	*0*	***igmp_fasttimo***	*0*	*0*

图32-6 原始插口的协议散转值的比较

不同BSD版本中，原始IP的实现各有不同。ip_protox表中，协议号等于IPPROTO_RAW记录通常都用做通配记录以支持未知的IP协议，而协议号等于0的记录通常做为默认记录，从而允许应用进程读写内核不支持的IP协议数据报。

应用进程使用IPPROTO_RAW记录，最早见于Van Jacobson开发的Traceout，这是第一个需要自己写IP首部(改变TTL字段)的应用进程。为了支持Traceout，修订了4.3BSD和Net/1，包括修改rip_output，在收到协议号等于IPPROTO_RAW的数据报时，假定应用进程提交了一个完整的IP数据报，包括IP首部。在Net/2中，引入了IP_HDRINCL插口选项，简化了IPPROTO_RAW的用法，允许应用进程利用通配记录发送自己的IP首部。

32.4 rip_init函数

系统初始化时，domaininit函数调用原始IP初始化函数rip_init (图32-7)。

raw_ip.c

```
47 void
48 rip_init()
49 {
50     rawinpcb.inp_next = rawinpcb.inp_prev = &rawinpcb;
51 }
```

raw_ip.c

图32-7 rip_init 函数

这个函数执行的唯一操作是令PCB首部(rawinpcb)中的前向和后向指针都指向自己，实现一个空的双向链表。

只要某个socket系统调用创建了SOCK_RAW类型的插口，下面将介绍的原始IP PRU_ATTACH函数就创建一个Internet PCB，并插入到rawinpcb链表中。

32.5 rip_input函数

因为ip_protox数组中保存的所有关于未知协议记录都指向IPPROTO_RAW(图7-8)，且后者的pr_input函数指向rip_input (图32-6)，所以只要某个接收IP数据报的协议号内核无法识别，就会调用此函数。但从图32-2可看出，ICMP和IGMP都可能调用rip_input，只要满足下列条件：

- icmp_input调用rip_input处理所有未知的ICMP报文类型和所有非响应的ICMP报文。
- igmp_input调用rip_input处理所有IGMP分组。

上述两种情况下，调用rip_input的一个原因是允许创建了原始插口的应用进程处理新增的ICMP和IGMP报文，内核可能不支持它们。

图32-8给出了rip_input函数。

raw_ip.c

```
59 void
60 rip_input(m)
61 struct mbuf *m;
62 {
63     struct ip *ip = mtod(m, struct ip *);
64     struct inpcb *inp;
```

图32-8 rip_input 函数

```
65      struct socket *last = 0;

66      ripsrc.sin_addr = ip->ip_src;
67      for (inp = rawinpcb.inp_next; inp != &rawinpcb; inp = inp->inp_next) {
68          if (inp->inp_ip.ip_p && inp->inp_ip.ip_p != ip->ip_p)
69              continue;
70          if (inp->inp_laddr.s_addr &&
71              inp->inp_laddr.s_addr == ip->ip_dst.s_addr)
72              continue;
73          if (inp->inp_faddr.s_addr &&
74              inp->inp_faddr.s_addr == ip->ip_src.s_addr)
75              continue;
76          if (last) {
77              struct mbuf *n;
78              if (n = m_copy(m, 0, (int) M_COPYALL)) {
79                  if (sbappendaddr(&last->so_rcv, &ripsrc,
80                                   n, (struct mbuf *) 0) == 0)
81                      /* should notify about lost packet */
82                      m_freem(n);
83                  else
84                      sorwakeup(last);
85              }
86          }
87          last = inp->inp_socket;
88      }
89      if (last) {
90          if (sbappendaddr(&last->so_rcv, &ripsrc,
91                           m, (struct mbuf *) 0) == 0)
92              m_freem(m);
93          else
94              sorwakeup(last);
95      } else {
96          m_freem(m);
97          ipstat.ips_noproto++;
98          ipstat.ips_delivered--;
99      }
100 }
```

raw_ip.c

图32-8 (续)

59-66 IP数据报中的源地址被保存在全局变量 `ripsrc`中，只要找到了匹配的 PCB，`ripsrc`将做为参数传给`sbappendaddr`。与UDP不同，原始IP没有端口号的概念，因此`sockaddr_in`结构中的`sin_port`总等于0。

2. 在所有原始IP PCB中寻找一个或多个匹配的记录

67-88 原始IP处理PCB表的方式与UDP和TCP不同。前面介绍过，这两个协议维护一个指针，总是指向最近收到的报文段 (单报文段缓存)，并调用通用函数`in_pcblookup`寻找一个最佳匹配(如果收到的数据报不同于缓存中的记录)。由于原始IP数据报可能发送到多个插口上，所以无法使用`in_pcblookup`，因此，必须遍历原始 PCB链表中的所有 PCB。这一点类似于UDP处理广播报文段和多播报文段的方式 (图23-26)。

3. 协议比较

68-69 如果PCB中的协议字段非零，并且与IP首部的协议字段不匹配，则PCB被忽略。也说明协议值等于0(`socket`的第三个参数)的原始插口能够匹配所有收到的原始 IP报文段。

4. 比较本地和远端IP地址

70-75 如果PCB中的本地地址非零，并且与IP首部的目的IP地址不匹配，则PCB被忽略。如果PCB中的远端地址非零，并且与IP首部的源IP地址不匹配，PCB被忽略。

上述3种测试说明应用进程能够创建一个协议号等于0的原始插口，即不绑定到本地地址，也不与远端地址建立连接，可以接收经rip_input处理的所有数据报。

代码71行和74行都有同样的错误：相等测试，实际应为不相等测试。

5. 递交接收数据报的复制报文段以备处理

76-94 sbappendaddr向应用进程提交一个接收数据报的复制报文段。变量last的使用与图23-26中的用法类似：因为sbappendaddr把报文段放入到适当队列中后将释放所有mbuf，如果有多个进程接收数据报的复制报文段，rip_input必须调用m_copy保存一份复制报文段。但如果只有一个应用进程接收数据报，则无需复制。

6. 无法上交的数据报

95-99 如果无法为数据报找到相匹配的插口，则释放mbuf，递增ips_noproto，递减ips_delivered。IP在调用rip_input之前已经递增过后一个计数器(图8-15)。由于数据报实际上没有上交给运输层，因此，必须递减ips_delivered，确保两个SNMP计数器，ipInDiscards和ipInDelivers(图8-16)，的正确性。

本节开始时，我们提到，icmp_input会为未知报文类型或非响应报文调用rip_input，意味着如果收到ICMP主机不可达报文，且rip_input找不到可匹配的原始插口PCB，ips_noproto会递增。这也说明为什么图8-5中的计数器值较大。在前面对该计数器的描述中提到“未知或不支持的协议”，这种说法是不正确的。

如果收到的IP数据报带有的协议字段，既无法为内核辩识，也无法由某个应用进程通过原始插口处理，Net/3不会生成差错代码等于2(协议不可达)的ICMP目的不可达报文。RFC 1122建议出现此种情况时应该生成ICMP差错报文(参见习题32.4)。

32.6 rip_output函数

图32-6中，ICMP、IGMP和原始IP都调用rip_output实现原始IP输出。应用进程调用5个写函数之一：send、sendto、sendmsg、write和writev，系统将输出报文段。如果插口已建立连接，就可以任意调用上述5个函数，尽管sendto和sendmsg中不能规定目的地址。如果插口没有建立连接，则只能调用sendto和sendmsg，且必须规定目的地址。

图32-9给出了rip_output函数。

1. 内核填充IP首部

119-128 如果IP_HDRINCR插口选项未定义，M_PREPEND为IP首部分配空间，并填充IP首部各字段。此处未填充的字段留待ip_output初始化(图8-22)。协议字段等于PCB中保存的值，并且是图32-10中socket系统调用的第三个参数。

TOS等于0，TTL等于255。内核为原始IP插口填充各首部字段时，通常都使用这些固定值。这与UDP和TCP不同，应用进程能够通过插口选项设定IP_TTL和IP_TOS值。

129 应用程序通过IP_OPTIONS插口选项设定的所有IP选项，都通过opts变量传给ip_output函数。

2. 调用者填充IP首部：IP_HDRINCR插口选项

130-133 如果选用了IP_HDRINCR插口选项，调用者在数据报前提供完整的IP首部。如果应用进程提供的ID字段等于0，对此类IP首部需做的唯一修改是ID字段。IP数据服的ID字段可以等于0。此处，rip_output对ID字段的赋值可以简化应用进程的处理，直接设 ID字段等于0，rip_output向内核请求内核变量ip_id的当前值，做为IP报文段的ID值。

134-136 令opts为空，忽略应用进程通过IP_OPTIONS可能设定的任何IP选项。如果调用者构建了自己的IP首部，其中肯定已包括了调用者希望加入的 IP选项。flags变量中必须有IP_RAWOUTPUT标志，告诉ip_output不要修改IP首部。

raw_ip.c

```
105 int
106 rip_output(m, so, dst)
107 struct mbuf *m;
108 struct socket *so;
109 u_long  dst;
110 {
111     struct ip *ip;
112     struct inpcb *inp = sotoinpcb(so);
113     struct mbuf *opts;
114     int     flags = (so->so_options & SO_DONTROUTE) | IP_ALLOWBROADCAST;
115     /*
116      * If the user handed us a complete IP packet, use it.
117      * Otherwise, allocate an mbuf for a header and fill it in.
118      */
119     if ((inp->inp_flags & INP_HDRINCL) == 0) {
120         M_PREPEND(m, sizeof(struct ip), M_WAIT);
121         ip = mtod(m, struct ip *);
122         ip->ip_tos = 0;
123         ip->ip_off = 0;
124         ip->ip_p = inp->inp_ip.ip_p;
125         ip->ip_len = m->m_pkthdr.len;
126         ip->ip_src = inp->inp_laddr;
127         ip->ip_dst.s_addr = dst;
128         ip->ip_ttl = MAXTTL;
129         opts = inp->inp_options;
130     } else {
131         ip = mtod(m, struct ip *);
132         if (ip->ip_id == 0)
133             ip->ip_id = htons(ip_id++);
134         opts = NULL;
135         /* XXX prevent ip_output from overwriting header fields */
136         flags |= IP_RAWOUTPUT;
137         ipstat.ips_rawout++;
138     }
139     return (ip_output(m, opts, &inp->inp_route, flags, inp->inp_moptions));
140 }
```

raw_ip.c

图32-9 rip_output 函数

137 计数器ips_rawout递增。执行Traceroute时，Traceroute每发送一个变量就会导致此变量加1。

rip_output的操作在不同版本中也有所变化。在Net/3中使用IP_HDRINCL插口选项时，rip_output对IP首部所做的唯一修改就是填充ID字段，如果应用进程将其定为0。因为IP_RAWOUTPUT标志置位，Net/3中的ip_output函数不改动IP首

部。但在Net/2中，即使IP_HDRINCL插口选项设定时，它也会修改IP首部中特定字段：IP版本号等于4，分片偏移量等于0，分片标志被清除。

32.7 rip_usrreq函数

协议的用户请求处理函数能够完成多种操作。与UDP和TCP的用户请求处理函数类似，rip_usrreq是一个很大的switch语句，每个PRU_*xxx*请求，都有一个对应的case子句。

图32-10给出的PRU_ATTACH请求，来自socket系统调用。

raw_ip.c

```
194 int
195 rip_usrreq(so, req, m, nam, control)
196 struct socket *so;
197 int     req;
198 struct mbuf *m, *nam, *control;
199 {
200     int     error = 0;
201     struct inpcb *inp = sotoinpcb(so);
202     extern struct socket *ip_mrouter;
203     switch (req) {

204     case PRU_ATTACH:
205         if (inp)
206             panic("rip_attach");
207         if ((so->so_state & SS_PRIV) == 0) {
208             error = EACCES;
209             break;
210         }
211         if ((error = soreserve(so, rip_sendspace, rip_recvspace)) ||
212             (error = in_pcballoc(so, &rawinpcb)))
213             break;
214         inp = (struct inpcb *) so->so_pcb;
215         inp->inp_ip.ip_p = (int) nam;
216         break;
```

raw_ip.c

图32-10 rip_usrreq 函数：PRU_ATTACH 请求

194-206 每次socket函数被调用时，都会创建新的socket结构，此时还没有指向某个Internet PCB。

1. 确认超级用户

207-210 只有超级用户才能创建原始插口，这是为了防止普通用户向网络发送自己的IP数据报。

2. 创建Internet PCB，保留缓存空间

211-215 为输入和输出队列保留所需空间，调用in_pcballoc分配新的Internet PCB，添加到原始IP PCB链表中(rawinpcb)，并与socket结构建立对应关系。rip_usrreq的nam参数就是socket系统调用的第三个参数：协议。它被保存在PCB中，因为rip_input需用它上交收到的数据报，rip_output也要把它填入到外出数据报的协议字段中(如果IP_HDRINCL未设定)。

原始IP插口与远端IP地址建立的连接，与UDP插口和远端IP地址建立的连接相类似。它固定了原始插口只能接收来自于特定地址的数据报，如我们在rip_input中所看到的。原始IP

与UDP一样，是一个无连接协议，下面两种情况下会发送PRU_DISCONNECT请求：

1) 关闭建立连接的原始插口时，在PRU_DETACH之前会先发送PRU_DISCONNECT请求。

2) 如果对一个已建立连接的原始插口调用connect，soconnect在发送PRU_CONNECT请求前会先发送PRU_DISCONNECT请求。

图32-11给出了PRU_DISCONNECT、PRU_ABORT和PRU_DETACH请求。

raw_ip.c

```
217     case PRU_DISCONNECT:
218         if ((so->so_state & SS_ISCONNECTED) == 0) {
219             error = ENOTCONN;
220             break;
221         }
222         /* FALLTHROUGH */

223     case PRU_ABORT:
224         soisdisconnected(so);
225         /* FALLTHROUGH */

226     case PRU_DETACH:
227         if (inp == 0)
228             panic("rip_detach");
229         if (so == ip_mrouter)
230             ip_mrouter_done();
231         in_pcbdetach(inp);
232         break;
```

raw_ip.c

图32-11 rip_usrreq 函数：PRU_DISCONNECT 、PRU_ABORT 和PRU_DETACH 请求

217-222 如果处理PRU_DISCONNECT请求的插口没有进入连接状态，则返回错误。

223-225 尽管禁止在一个原始插口上发送PRU_ABORT请求，这个case语句实际上是PRU_DISCONNECT请求处理的延续。插口转入断开状态。

226-230 close系统调用发送PRU_DETACH请求，这个case语句还将结束PRU_DISCONNECT请求的处理。如果socket结构用于多播选路(ip_mrouter)，则调用ip_mrouter_done取消多播选路。一般情况下，mrouted (8)守护程序会通过DVMPR_DONE插口选项取消多播选路，因此，这个条件用于防止mrouted (8) 在没有正确设定插口选项之前就异常终止了。

231 调用in_pcbdetach释放Internet PCB，并从原始IP PCB表(rawinpcb)中删除。

通过PRU_BIND请求，可以把原始IP插口绑定到某个本地IP地址上，如图32-12所示。我们在rip_input中指出，插口将只能接收发向该地址的数据报。

233-250 应用进程向sockaddr_in结构填充本地IP地址。下列3个条件必须全真，否则将返回差错代码EADDRNOTAVAIL：

1) 至少配置了一个IP接口；

2) 地址族应等于AF_INET (或者AF_IMPLINK，历史上人为造成的不一致)；和

3) 如果绑定的IP地址不等于0.0.0.0，它必须对应于某个本地接口。调用者的sockaddr_in中的端口号必须等于0，否则，ifa_ifwithaddr将返回错误。

本地IP地址保存在PCB中。

raw_ip.c

```
233     case PRU_BIND:
234         {
235             struct sockaddr_in *addr = mtod(nam, struct sockaddr_in *);
236             if (nam->m_len != sizeof(*addr)) {
237                 error = EINVAL;
238                 break;
239             }
240             if ((ifnet == 0) ||
241                 ((addr->sin_family != AF_INET) &&
242                  (addr->sin_family != AF_IMPLINK)) ||
243                 (addr->sin_addr.s_addr &&
244                  ifa_ifwithaddr((struct sockaddr *) addr) == 0)) {
245                 error = EADDRNOTAVAIL;
246                 break;
247             }
248             inp->inp_laddr = addr->sin_addr;
249             break;
250         }
```

raw_ip.c

图32-12 rip_usrreq 函数：PRU_BIND 请求

应用进程还可以在原始IP插口与某个特定远端IP地址间建立连接。我们在rip_input中指出，这样可以限制应用进程只能接收源 IP地址等于连接对端IP地址的数据报。应用进程可以同时调用bind和connect，或者两者都不调用，取决于它希望rip_input对接收数据报采用的过滤方式。图32-13给出了PRU_CONNECT请求的处理逻辑。

251-270 如果调用者的sockaddr_in初始化正确，且至少配置了一个IP接口，则指定的远端地址将存储在PCB中。注意，这一处理和UDP插口建立与远端IP地址的连接有所不同。对于UDP，in_pcbconnect申请到达远端地址的一条路由，并把外出接口视为本地地址（图22-9）。对于原始IP，只有远端IP地址存储到PCB中，除非应用进程还调用了bind，rip_input将只比较远端地址。

raw_ip.c

```
251     case PRU_CONNECT:
252         {
253             struct sockaddr_in *addr = mtod(nam, struct sockaddr_in *);
254             if (nam->m_len != sizeof(*addr)) {
255                 error = EINVAL;
256                 break;
257             }
258             if (ifnet == 0) {
259                 error = EADDRNOTAVAIL;
260                 break;
261             }
262             if ((addr->sin_family != AF_INET) &&
263                 (addr->sin_family != AF_IMPLINK)) {
264                 error = EAFNOSUPPORT;
265                 break;
266             }
267             inp->inp_faddr = addr->sin_addr;
268             soisconnected(so);
269             break;
270         }
```

raw_ip.c

图32-13 rip_usrreq 函数：PRU_CONNECT 请求

应用进程结束发送数据后，调用shutdown，生成PRU_SHUTDOWN请求，尽管应用进程很少为原始IP插口调用shutdown。图32-14给出了PRU_CONNECT2和PRU_SHUTDOWN请求的处理逻辑。

raw_ip.c

```
271     case PRU_CONNECT2:
272         error = EOPNOTSUPP;
273         break;

274         /*
275          * Mark the connection as being incapable of further input.
276          */
277     case PRU_SHUTDOWN:
278         socantsendmore(so);
279         break;
```

raw_ip.c

图32-14 PRU_CONNECT2 和PRU_SHUTDOWN 请求

271-273 原始IP插口不支持PRU_CONNECT2请求。

274-279 socantsendmore置位插口标志，禁止所有输出。

图23-14中，我们给出了5个写函数如何调用协议的pr_usrreq函数，发送PRU_SEND请求。图32-15给出了这个请求的处理逻辑。

raw_ip.c

```
280         /*
281          * Ship a packet out.  The appropriate raw output
282          * routine handles any massaging necessary.
283          */
284     case PRU_SEND:
285         {
286             u_long  dst;

287             if (so->so_state & SS_ISCONNECTED) {
288                 if (nam) {
289                     error = EISCONN;
290                     break;
291                 }
292                 dst = inp->inp_faddr.s_addr;
293             } else {
294                 if (nam == NULL) {
295                     error = ENOTCONN;
296                     break;
297                 }
298                 dst = mtod(nam, struct sockaddr_in *)->sin_addr.s_addr;
299             }
300             error = rip_output(m, so, dst);
301             m = NULL;
302             break;
303         }
```

raw_ip.c

图32-15 rip_usrreq 函数：PRU_SEND 请求

280-303 如果插口处于连接状态，则调用者不能指定目的地址(nam参数)。如果插口未建立连接，则需要指明目的地址。不管哪种情况，只要条件满足，dst将等于目的IP地址。rip_output发送数据报。令mbuf指针m为空，防止函数结束时释放mbuf链。因为接口输出

例程发送数据报之后会释放 mbuf链(记住，rip_output向ip_output提交mbuf链，ip_output把它加入到接口的输出队列中)。

图32-16给出了rip_usrreq的最后一部分代码。由fstat系统调用生成的PRU_SENSE请求，没有返回值。PRU_SOCKADDR和PRU_PEERADDR请求分别由getsockname和getpeername系统调用生成。原始IP插口不支持其余请求。

319-324 函数in_setsockaddr和in_setpeeraddr能够从PCB中读取信息，在nam参数中返回结果。

raw_ip.c

```
304     case PRU_SENSE:
305         /*
306          * fstat: don't bother with a blocksize.
307          */
308         return (0);

309         /*
310          * Not supported.
311          */
312     case PRU_RCVOOB:
313     case PRU_RCVD:
314     case PRU_LISTEN:
315     case PRU_ACCEPT:
316     case PRU_SENDOOB:
317         error = EOPNOTSUPP;
318         break;

319     case PRU_SOCKADDR:
320         in_setsockaddr(inp, nam);
321         break;

322     case PRU_PEERADDR:
323         in_setpeeraddr(inp, nam);
324         break;

325     default:
326         panic("rip_usrreq");
327     }
328     if (m != NULL)
329         m_freem(m);
330     return (error);
331 }
```

raw_ip.c

图32-16 rip_usrreq 函数：剩余的请求

32.8 rip_ctloutput函数

setsockopt和getsockopt函数会调用rip_ctloutput，它处理一个IP插口选项和8个用于多播选路的插口选项。

图32-17给出了rip_ctloutput函数的第一部分。

raw_ip.c

```
144 int
145 rip_ctloutput(op, so, level, optname, m)
146 int     op;
147 struct socket *so;
```

图32-17 rip_usrreq 函数：处理IP_HDRINCL 插口选项

```
148 int     level, optname;
149 struct mbuf **m;
150 {
151     struct inpcb *inp = sotoinpcb(so);
152     int     error;

153     if (level != IPPROTO_IP)
154         return (EINVAL);

155     switch (optname) {

156     case IP_HDRINCL:
157         if (op == PRCO_SETOPT || op == PRCO_GETOPT) {
158             if (m == 0 || *m == 0 || (*m)->m_len < sizeof(int))
159                         return (EINVAL);
160             if (op == PRCO_SETOPT) {
161                 if (*mtod(*m, int *))
162                             inp->inp_flags |= INP_HDRINCL;
163                 else
164                     inp->inp_flags &= ~INP_HDRINCL;
165                 (void) m_free(*m);
166             } else {
167                 (*m)->m_len = sizeof(int);
168                 *mtod(*m, int *) = inp->inp_flags & INP_HDRINCL;
169             }
170             return (0);
171         }
172         break;
```

raw_ip.c

图32-17 (续)

144-172 保存新选项值或者选项当前值的mbuf至少要能容纳一个整数。对于setsockopt系统调用，如果mbuf中的整数值非零，则设定该标志，否则清除它。对于getsockopt系统调用，mbuf中的返回值要么等于0，要么是非零的选项值。函数返回，以避免switch语句结束时处理其他IP选项。

图32-18给出了rip_ctloutput函数的最后一部分，处理8个多播选路插口选项。

raw_ip.c

```
173     case DVMRP_INIT:
174     case DVMRP_DONE:
175     case DVMRP_ADD_VIF:
176     case DVMRP_DEL_VIF:
177     case DVMRP_ADD_LGRP:
178     case DVMRP_DEL_LGRP:
179     case DVMRP_ADD_MRT:
180     case DVMRP_DEL_MRT:
            /* shown in Figure 14.9 */
188     }
189     return (ip_ctloutput(op, so, level, optname, m));
190 }
```

raw_ip.c

图32-18 rip_usrreq 函数：处理多播选路插口选项

173-188 这8个插口选项只对 setsockopt系统调用有效，它们由图 14-9讨论的

ip_mrouter_cmd函数处理。

189 所有其他IP插口选项，如设定IP选项的IP_OPTIONS，则由ip_ctloutput处理。

32.9 小结

原始插口为IP主机提供3种功能。

1) 用于发送和接收ICMP和IGMP报文。

2) 支持应用进程构建自己的IP首部。

3) 允许应用进程支持基于IP的其他协议。

原始IP较为简单——只填充IP首部的有限几个字段——但它允许应用进程提供自己的IP首部。例如，调试程序就能发送任何类型的IP数据报。

原始IP输入提供了3种处理方式，能够选择性地接收进入的IP数据报。应用进程基于下列因素选择接收数据报：(1) 协议字段；(2) 源IP地址(由connect指明)；(3) 目的IP地址(由bind指明)。应用进程可以任意组合上述3种过滤条件。

习题

32.1 假定IP_HDRINCL插口选项未设定。如果socket的第三个参数等于0，rip_output填入IP首部协议字段(ip_p)的值是多少？如果socket的第三个参数等于IPPROTO_RAW (255)，rip_output填入该段(ip_p)的值又是多少？

32.2 应用进程创建了一个原始插口，协议值等于IPPROTO_RAW (255)。应用进程在这个插口上将收到什么类型的IP数据报？

32.3 应用进程创建了一个原始插口，协议值等于0。应用进程在这个插口上将收到什么类型的IP数据报？

32.4 修改rip_input，在适当情况下发送代码等于2 (协议不可达)的ICMP目的不可达报文。请注意，不要为rip_input正处理的ICMP或IGMP数据报生成一个差错。

32.5 如果应用进程希望生成自己的IP数据报，自己填充IP首部字段，可使用IP_HDRINCL选项置位的原始IP插口，或者采用BPF(第31章)，两种方法的区别是什么？

32.6 什么时候应用进程应该读取原始IP插口？什么时候读取BPF？

附录A　部分习题的解答

第1章

1.2　SLIP驱动程序执行spltty(图1-13)，其优先级必须低于或等于 splimp，且高于splnet。因此，SLIP驱动程序由于中断而被阻塞。

第2章

2.1　M_EXT标志是mbuf自身的一个属性，而不是mbuf中保存的数据报的属性。

2.2　调用者请求大于100字节(MHLEN)的连续空间。

2.3　不可行，因为多个mbuf都可指向簇(2.9节)。此外，簇中也没有用于后向指针的空间(习题2.4)。

2.4　在<sys/mbuf.h>定义的宏MCLALLOC和MCLFREE中，我们看到引用计数器是一个名为mclrefcnt的数组。它在内核初始化时被分配，代码文件为 machdep.c。

第3章

3.3　采用很大的交互式队列，并不符合建立队列的目的，新的交互式流量跟在原有流量之后，会造成附加时延。

3.4　因为sl_softc结构都是全局变量，内核初始化时都被置为 0。

3.5

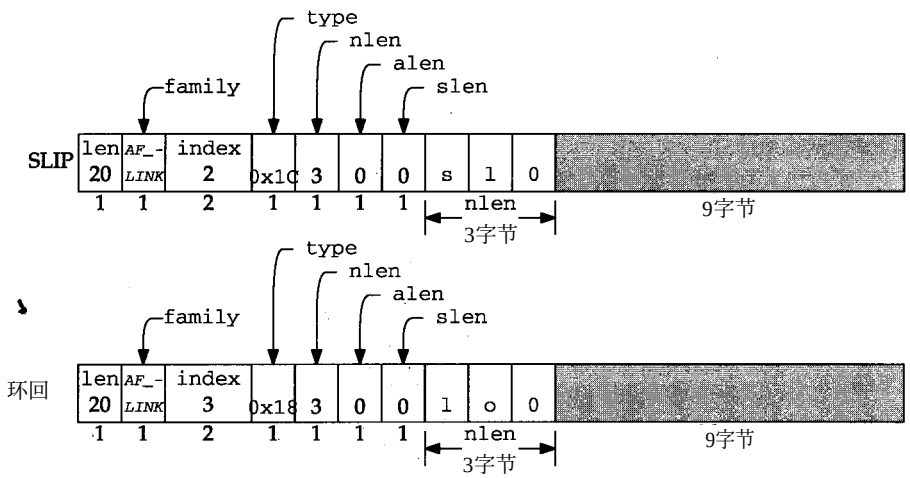


图　A-1

第4章

4.1　leread必须查看数据报，确认把数据报提交给 BPF之后，是否需要将其丢弃。因为

BPF开关会造成接口处于一种混杂模式，数据报的目的地有可能是以太网中的其他主机，BPF处理完毕后，必须将其丢弃。

如果接口没有加开关，则必须在ether_input中完成这一测试。

4.2 如果测试反过来，广播标志永远不会置位。

如果第二个if前没有else，所有广播分组都会带上多播标志。

第5章

5.1 环回接口不需要输入函数，因为它接收的所有分组都直接来自于 looutput，后者实际完成了输入功能。

5.2 堆栈分配快于动态存储器分配。对 BPF处理，性能是首要考虑的因素，因为对每个进入数据报都会执行该代码。

5.5 缓存溢出的第一个字节被丢弃， SC_ERROR置位， slinput重设簇指针，从缓存起始处开始收集字符。因为SC_ERROR置位， slinput收到SLIP END字符后，丢弃当前接收的数据帧。

5.6 如果检验和无效或者IP首部长度与实际数据报长度不匹配，数据报被丢弃。

5.7 因为ifp指向le_softc结构的第一个成员，

```
sc = (struct le_softc*) ifp;
```

sc初始化正确。

5.8 这是非常困难的。某些路由器在开始丢弃数据报时，可能会发送 ICMP源站抑制报文段，但Net/3实现中的UDP插口丢弃这些报文段(图23-30)。应用程序可以使用与TCP所采用的相同技术：根据确认的数据报估算往返时间，确认可用的带宽和时延。

第6章

6.1 IP子网出现之前(RFC 950 [Mogul和Postel 1985])，IP地址的网络和主机部分都以字节为界。in_addr结构的定义如下：

```
struct in_addr {
        union {
                struct { u_char s_b1, s_b2, s_b3, s_b4; } S_un_b;
                struct { u_short s_w1, s_w2; } S_un_w;
                u_long S_addr;
        } S_un;
#define s_addr  S_un.S_addr         /* should be used for all code */
#define s_host  S_un.S_un_b.s_b2    /* OBSOLETE: host on imp */
#define s_net   S_un.S_un_b.s_b1    /* OBSOLETE: network */
#define s_imp   S_un.S_un_w.s_w2    /* OBSOLETE: imp */
#define s_impno S_un.S_un_b.s_b4    /* OBSOLETE: imp # */
#define s_lh    S_un.S_un_b.s_b3    /* OBSOLETE: logical host */
};
```

图 A-2

Internet地址读写单位既可以是 8 bit字节，也可以是 16 bit单字，或32 bit双字。宏s_host、s_net、s_imp等等，它们的名字明确地反应出早期TCP/IP网络的结构。子网和超网概念的引入，淘汰了这种以字节和单字区分的做法。

6.2 返回指向结构sl_softc[0]的指针。

6.3 接口输出函数，如ether_output，只有一个指向接口ifnet结构的指针，而没有指向ifaddr的指针。在arpcom结构(最后一次为接口设定的IP地址)中使用IP地址可以避免从ifaddr地址链表中寻找所需地址。

6.4 只有超级用户进程才能创建原始IP插口。通过UDP插口，任何用户进程能够查看接口配置，但内核仍拥有超级用户特权，能够修改接口地址。

6.5 有3个函数循环处理网络掩码，一次处理一个字节。它们是ifa_ifwithnet、ifaof_ifpforaddr和rt_maskedcopy。较短的网络掩码能够提高这些函数的性能。

6.6 与远端系统建立Telnet连接。Net/2系统不应该转交这些数据报，而其他系统不会接受到达环回接口之外的非环回接口的环回数据报。

第7章

7.1 下列调用返回指向inetsw[6]的指针：

```
pffindproto(PF_INET, 0, SOCK_RAW);
```

第8章

8.1 可能不会。系统不可能响应任意的广播报文，因为没有可供响应的源地址。

8.4 因为数据报已经损坏，无法知道首部中的地址是否正确。

8.5 如果应用程序选取的源地址与指定的外出接口的地址不同，则无法发送到下一跳路由器。如果下一跳路由器发现数据报源地址与其到达的子网地址不符，则不会执行下一步的转发操作。这是尽量减少终端系统复杂性带来的后果，RFC 1122指出了这一问题。

8.6 新主机认为广播报文来自于某个没有划分子网的网络中的主机，并试图将数据报发回给源主机。网络接口开始广播ARP请求，向网络请求该广播地址，当然，这一请求永远不会收到响应。

8.7 减少TTL的操作出现在小于等于1的测试之后，是为了避免收到的TTL等于0，减1后将等于255，从而引起操作差错。

8.8 如果两个路由器彼此认为对方是某个数据报的下一跳路由器，则形成环路。除非该环路被打破，原始数据报在两个路由器间来回传递，并且每个路由器都向源主机发送ICMP重定向报文段，如果该主机与路由器处于同一个网络中。路由更新时，不同路由器中的路由表暂时存在的不一致现象，会造成这种环路。

原始报文段的TTL最终减为0，数据报被丢弃。这是TTL存在的一个主要原因。

8.9 不会检查4个以太网广播地址，因为它们不属于接收接口。但应检查有限的广播地址，说明带有SLIP链路的系统采用有限的广播地址，即使不知道对端地址，也能与对端通信。

8.10 只对数据报的第一个分片(分片偏移量等于0)生成ICMP差错报文。无论是主机字节序，还是网络字节序，0的表示都相同，因此无需转换。

第9章

9.1 RFC 1122建议如果数据报中的选项彼此冲突，处理方式由各实现代码自己决定。Net/3能正确处理第一个源路由选项，但因为它会更新数据报首部的ip_dst，第二条源路由处理将出现差错。

9.2 网络中的主机也可以用做到达网络其他主机的中继。如果目的主机不可直接到达，

源主机可在数据报中加入路由，首先到达中继主机，接着到达最终的目的主机。路由器不会丢弃数据报，因为目的地址指向中继主机，后者将处理路由并把数据报转发给最终目的主机。目的主机把路由反转，同样利用中继主机转发响应。

9.3　采用与前一个习题同样的原则。我们选取一个能够同时与源主机和目的主机通信的中继路由器，并构造源路由，穿过中继路由器到达目的地址。中继路由器必须与目的地址处于同一个网络，通信中无需默认路由。

9.4　如果源路由是仅有的IP选项，NOP选项使得所有IP地址以4字节边界对齐，从而能够优化存储器中的地址读取操作。这种对齐技术也适用于多个 IP选项，如果每个IP选项都通过NOP填充，保证按4字节边界对齐。

9.5　不应混淆非标准时间值和标准时间值，最大的标准时间值等于86 399 399(24×60×60×1000 - 1)，需要28 bit才能表示。由于时间值有32 bit，从而避免了高位比特的混淆问题。

9.6　源路由选项代码在处理过程中可能会改变`ip_dst`。保存目的地址，从保证时间戳处理使用原始目的地址。

第10章

10.2　重装后，只有第一个分片的选项上交给运输层协议。

10.3　因为数据长度(204 + 20)大于208(图2-16)。

图10-11中的`m_pullup`把头40字节复制到一个单独的mbuf中，如图2-18所示。

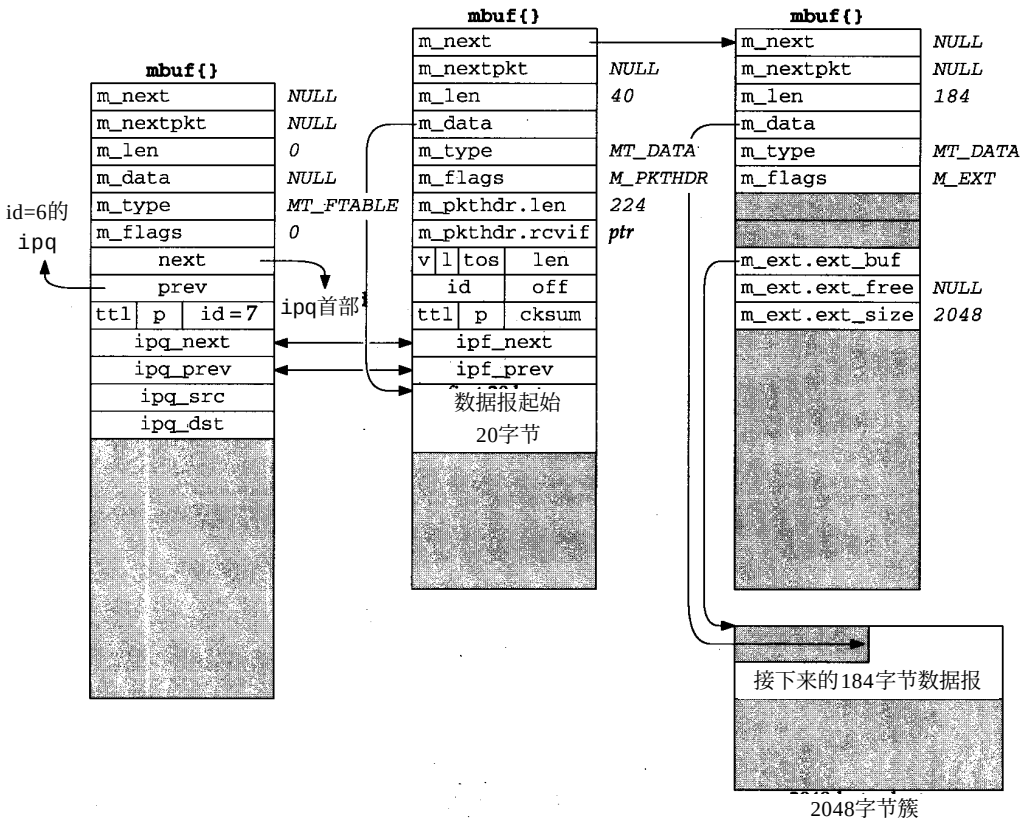


图　A-3

10.5 平均每个数据报收到的分片数等于

$$\frac{72786-349}{16\ 557} = 4.4$$

平均每个输出数据报新建的平均分片数等于

$$\frac{796\ 084}{2\ 6\ 0484} = 3.1$$

10.6 图10-11中，数据报最初被做为分片处理。当ip_off左移时，保留的比特位被丢弃。得到的数据报被视为分片或一个完整的数据报，取决于 MF和分片偏移量的值。

第11章

11.1 输出响应使用收到请求的接口的源地址。主机可能无法辩识 0.0.0.0是一个有效的广播地址，因此，有可能忽略请求。推荐的广播地址等于 255.255.255.255。

11.2 假定主机发送了一个链路层的广播数据报，其源 IP地址是另一台主机的地址，且数据报有差错，如内容差错的选项。所有主机都能接收并检测出差错，因为这是一个链路层的广播报文，而且选项的处理先于最终目的地的检测。许多发现差错的主机会向数据报的源IP地址发送ICMP报文，即使原数据报属于链路层广播。另一台主机将收到大量假的ICMP差错。这就是为什么不允许为链路层广播而发送 ICMP差错报文。

11.3 第一个例子中，这种重定向报文不会诱骗主机向另一个子网中的某个主机发送报文段。这台主机可能被误认为是路由器，但它确实记录收到的流量。 RFC 1009规定路由器只能向位于同一个子网的其他路由器发送重定向报文。即使主机忽略了这些要求把数据报转发到另一个子网的报文段，但如果报文段发送者与主机处于同一个子网中，它们就会被接受。第二个例子，为了防止出现上述现象，要求主机只接受它(错误地)选定的原始路由器的重定向报文，即假定这个错误的路由器是管理员指定的默认路由器。

11.4 通过向rip_input传递报文段，进程级的守护程序能够正确响应，一些依赖于这种行为的老系统能够继续得到支持。

11.5 ICMP差错只针对IP数据报的第一个分片。因为第一个分片的偏移量值必等于 0，字段的字节表示顺序是无关紧要的。

11.6 如果收到ICMP请求的接口还未配置IP地址，则ia将为空，且不生成响应。

11.7 Net/3处理与时间戳响应一起到达的数据。

11.10 高位比特被保留，并必须设为0。如果它必须被发送，则icmp_error将丢弃数据报。

11.11 返回值被丢弃，因为icmp_send不返回差错。更重要的是， ICMP报文处理过程中生成的差错将被丢弃，以避免进入死循环，不断生成差错报文。

第12章

12.1 以太网中， IP广播地址255.255.255.255转换为以太网的广播地址 ff:ff:ff:ff:ff:ff，网络中的所有以太网接口都会接收这样的数据帧。没有运行 IP软件的系统必须主动接

收并丢弃这种广播报文。

数据报需发送给多播组 224.0.0.1中的所有主机，转换后的以太网多播地址为01:00:5e:00:00:01，只有明确要求其接口接收IP多播报文的系统才会收到它。没有运行IP或者在链路层不兼容的系统不会收到这些报文段，因为以太网接口的硬件已直接丢弃了这些报文段。

12.2 一种替代方案是通过文本名规定接口，如同ifreq结构和ioctl命令存取接口信息采取的方式一样。ip_setmoptions和ip_getmoptions可以调用ifunit，取代INADDR_TO_IFP，寻找指向接口ifnet结构的指针。

12.3 多播组高位4 bit通常为1110，因此，只有5个有意义的比特被匹配函数丢弃。

12.4 完整的ip_moptions结构必须能放入单个mbuf中，从而限制结构最大只能等于108字节(记住20字节的mbuf首部)。IP_MAX_MEMBERSHIPS可以大一些，但必须小于等于25(4+1+1+2+(4×25)=108)。

12.5 数据报重复，在IP输入队列中有两份复制的数据报。多播应用程序必须能识别并丢弃重复的数据报。

12.6

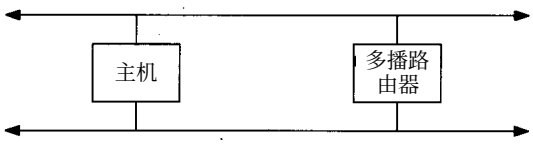


图 A-4

12.8 应用进程可以创建第二个插口，并通过第二个插口请求IP_MAX_MEMBERSHIPS。

12.9 为mbuf首部的m_flags成员变量定义一个新的mbuf标志M_LOCAL。ip_output处理环回数据报时置位该标志，从而取代检验和。如果该标置位，ipintr就跳过检验和验证。SunOS 5.X提供完成此功能的选项(ip_local_cksum，卷1的531页)。

12.10 存在$2^{23}-1$(8 388 607)个独立的以太网IP多播地址。记住保留的IP组224.0.0.0。

12.11 这个假设正确，因为in_addmulti拒绝所有新增请求，如果接口没有调用ioctl函数，说明如果if_ioctl为空，则永不会调用in_delmulti。

12.12 mbuf永远不会被释放，说明ip_getmoptions包含了一个存储器泄露。ip_getmoptions由ip_ctloutput调用，调用语句如下：

```
ip_getmoptions(IP_ADD_MEMBERSHIP, 0, mp)
```

会引发ip_getmoptions中的一个差错。

第13章

13.1 要求环回接口响应ICMP请求是没有必要的，因为本地主机是环回网络中唯一的系统，它已经知道自己的成员状态。

13.2 max_linkhdr + sizeof(struct ip) + IGMP_MINLEN=16+20+8=44<100

13.3 报告成员状态时出现随机延迟的主要原因是为了最大限度地减少出现在多播网络中的报告数(理想情况下应等于1)。一个点到点网络只包括两个接口，因此，无需延迟

以减少响应的数量。一个接口(假定是一个多播路由器)发出请求，另一个接口响应。另一个原因是避免过多的成员状态报告淹没接口的输出队列。大量 IGMP成员状态报文可能会超出输出队列关于数据报和字节的限制。例如，在 SLIP驱动程序中，如果输出队列已满或设备过忙，就会丢弃队列中所有等待的数据报 (图5-16)。

第14章

14.1 5个，分别对应网络A~E。

14.2 `grplst_member`只被`ip_mforward`调用，但在协议处理过程中，`ip_mforward`又将被`ipintr`或者`ip_output`调用，`ip_output`可以由插口层间接调用。缓存是一个共享数据区，在更新时必须加以保护。 `add_lgrp`和`del_lgrp`在更新成员列表时，通过`splx`保护此共享数据结构。

14.3 `SIOCDELMULTI`命令只影响以太网接口的多播列表，不改变 IP多播组列表，因此，接口仍然保留为组中成员。只要依旧是接口 IP组列表中的一员，接口将继续接收属于该组的多播数据报。

14.4 只有虚接口才能成为多播树的父接口。如果分组在隧道上接收，那么对应的物理接口不可能成为父接口，`ip_mforward`丢弃分组。

第15章

15.1 插口可以在分支上共享，或通过UNIX域插口传给应用进程([Stevens])。

15.2 `accept`返回后，结构的`sa_len`成员大于缓存大小。对固定长度的 Internet地址而言，这不是问题，但它有可能用于可变长度的地址，例如 OSI协议支持的地址格式。

15.4 只有`so_qlen`不等于0时，才会调用`soqremque`。如果`soqremque`返回一个空指针，说明插口队列代码必然出现了内核无法处理的问题。

15.5 复制的目的在于结构锁定时仍可调用 `bzero`清零，并可在 `splx`后接着调用`dom_dispose`和`sbrealse`，从而最大程度地减少了 CPU停留在`splimp`的时间，即网络中断被阻塞的时间。

15.6 宏`sbspace`返回0，从而`sbappendaddr`和`sbappendcontrol`函数(由UDP调用)将拒绝向队列添加新报文段。 TCP调用`sbappend`，后者假定调用者已事先检查过可用空间。即使`sbspace`返回0，TCP也会调用`sbappend`，但放入接收队列中的数据还不能提交给应用进程，因为`SS_CANTRCVMORE`标志阻止`read`系统调用返回任何数据。

第16章

16.1 如果给uio结构中的`uio_resid`赋值，它将成为一个大负数。 `sosend`拒绝带有`EINVAL`的报文段。

Net/2不检查负值，`sosend`起始处的注释说明了这个问题(图16-23)。

16.2 不。向簇中填充的字节数少于`MCLBYTES`只可能出现在报文段尾部，此时剩余的字节数小于`MCLBYTES`。此时，`resid`等于0，循环在394行`break`语句处终止，还未

到达测试条件spce>0。

16.5　应用进程阻塞，直到缓存解锁。本例中，只有在另一个进程检查缓存或向协议层传送数据时，缓存才会被锁定；而在应用进程等待缓存可用空间时不会加锁，后者有可能等待无限长的时间。

16.6　如果发送缓存包括许多mbuf，每个都包括若干字节的数据，那么当mbuf分配大块存储器时，sb_cc很可能大大低于sb_hiwat规定的限制。如果内核不限制每个缓存可拥有的mbuf的数量，应用进程就能轻易地造成存储器枯竭。

16.7　recvit分别由recvfrom和recvmsg调用。只有recvmsg处理控制信息。它把完整的msghdr结构，包括控制信息长度，复制给应用进程。至于地址信息，recvmsg把namelenp参数设为空，因为它可从msg_namelen中得到所需长度。当recvfrom调用recvit时，namelenp非空，因为函数需要从*namelenp中得到所需长度。

16.8　MSG_EOR由soreceive清除，因此，它不可能在M_EOR mbuf被处理前，被soreceive返回。

16.9　select检查描述符时，实际上存在一种竞争。如果某个选定事件发生在selscan查看描述符之后，但在select调用tsleep之前，该事件不会被发现，应用进程将保持睡眠状态，直到下一个选定事件发生。

第17章

17.1　简化在内核和应用进程间复制数据的代码。copyin和copyout可用于单个的mbuf，但需要uiomove处理多个mbuf。

17.2　代码工作正确，因为linger结构的第一个成员是所要求的整数标志。

第18章

18.1　做一个8行的表格，每行对应一种查找键、路由表键和路由表掩码中比特的组合方式：

行	1 查找键	2 路由表键	3 路由表掩码	1 & 3	2 = =4?	1 ^ 2	6 & 3
1	0	0	0	0	是	0	0=是
2	0	0	1	0	是	0	0=是
3	0	1	0	0	否	1	0=是
4	0	1	1	0	否	1	1=否
5	1	0	0	0	是	1	0=是
6	1	0	1	1	否	1	1=否
7	1	1	0	0	否	0	0=是
8	1	1	1	1	是	0	0=是

图　A-5

标注为“2 == 4？”和标注为“6 & 3”的两栏，值应相等。第一眼看上去，似乎并不完全相同，但我们可以略过第3行和第7行，因为这两行中路由表比特等于1，而

在路由表掩码中的对应比特也等于 1。构建路由表时，键值与掩码逻辑与，保证掩码中的等于0每一比特位，键值中的对应比特位也等于 0。

可以从另一个角度理解图 18-40中的异或和逻辑与操作，异或结果等于 1的条件是查找键比特不同于路由表键值中对应的比特位。之后的逻辑与操作忽略所有与掩码中等于0的比特相对应的比特。如果结果依然非零，则查找键与路由表键值不匹配。

18.2 rtentry结构的大小等于120字节，其中包括两个radix_node结构。每条记录还要求两个sockaddr_in结构(图18-28)，有152字节。总数约为3兆字节。

18.3 因为rn_b是一个短整数，假定短整数占 16 bit，因此，每个键值最多有 32767 bit(4095字节)。

第19章

19.1 图19-15中，如果重定向报文创建了新的路由，将置位 RTF_DYNAMIC标志；如果重定向报文修改了现有路由的网关字段，则置位 RTF_MODIFIED标志。如果重定向报文新建了一条路由，之后另一个重定向报文又修改了它，则两个标志都会置位。

19.2 在每个可通过默认路由到达的主机上创建一条主机路由。 TCP能够对每个主机维护并更新路由矩阵(图27-3)。

19.3 每个rt_msghdr结构需要76字节。主机路由中包括还两个 sockaddr_in结构(目的地和网关)，因此，报文段大小为 108字节。每条ARP记录的报文段为 112字节：一个sockaddr_in和一个sockaddr_dl。总长度等于(15×112+20×108)即3840字节。一条网络路由(非主机路由)还需要另外的8个字节存放网络掩码(数据大小等于116字节，而非108字节)，因此，如果20条路由全部为网络路由，总长度等于 4000字节。

第20章

20.1 返回值放入报文段的rtm_errno成员变量中(图20-14)，同时也做为write的返回值(图20-22)。后者更可靠，因为前者可能会因为 mbuf短缺，而丢弃响应报文段 (图20-17)。

20.2 对SOCK_RAW型的插口，pffindproto函数(图7-20)将返回协议值等于 0(通配)的记录，如果没有找到可匹配的记录。

第21章

21.1 它基于假定ifnet结构位于arpcom的开头，事实也是如此(图3-20)。

21.2 发送ICMP的回显请求不需要ARP，因为目的地址是广播地址。但 ICMP的回显响应一般都是点对点的，因此，发送者必须通过 ARP确定目的以太网地址。本地主机收到ARP请求时，in_arpinput应答并为另一主机创建一条记录。

21.3 如果创建了一条新的 ARP记录，图 19-8中的rtrequest从源记录中复制rt_gateway值，本例中为sockaddr_dl结构。图21-1中，我们看到该记录的sdl_alen值等于0。

21.4 Net/3中，如果arpresolve的调用者提供了指向路由表表项的指针，则不会再调用arplookup，通过rt_gateway指针可得到所需的以太网地址 (假定它还未超

时)。这样可以避免通常意义上的任何类型的查询。第 22章中，我们将看到 TCP和UDP在自己的协议控制块中保存指向路由表的指针， TCP不再需要搜索路由表 (连接的目的IP地址不会变化)，在目的地址不变时UDP也不需这样做。

21.5 如果ARP记录不完整，则它在记录创建后 0~5分钟超时。arpresolve发送ARP请求时，令rt_expire等于当前时间。下一次执行 arpresolve时，如果记录还没有解析，则删除它。

21.6 ether_output返回EHOSTUNREACH，而非EHOSTDOWN，从而ip_forward将发送ICMP主机不可达差错报文。

21.7 图21-28中，为140.252.13.35创建记录时，值等于当前时间。它不会改变。140.252.13.33和140.252.13.34记录的值复制自 140.252.13.32，因为rtrequest根据140.252.13.32复制前两条记录。之后， arpresolve发送ARP请求时，把这两条记录的值更新为当前时间，最后由 in_arpinput将其更新为收到ARP响应的时间加上20分钟。

21.8 修改图21-19开始处的arplookup，第二个参数永远等于1(创建标志)。

21.9 在下一秒的后半秒发送第一个数据报。因此，第一个和第二个数据报都会导致发送ARP请求，间隔约为500 ms，因为内核的time.tv_sec变量在这两个数据报发送时的值不同。

21.10 每个待发送的数据报都是一个 mbuf链，m_nextpkt指针指向每个链的第一个mbuf，用于构成等待传输的mbuf链表。

第22章

22.1 无限循环等待某个端口变为可用，假定允许应用进程打开足够多的描述符，绑定所有临时端口。

22.2 极少有服务器支持此选项。 [Cheswick和Bellovin 1994]提到为什么它可用于实现防火墙系统。

22.4 udb结构初始化为0，因此， udb.inp_lport从0开始。第一次调用 ip_pcbbin时，它增加为1，因为小于1024，所以被设定为1024。

22.5 一般情况下，调用者把地址族(sa_family)设为AF_INET，但我们在图22-20的注释中看到，最好不进行关于地址族的测试。调用者设定长度变量 (sa_len)，但我们在图 15-20中看到，函数 sockargs将其做为 bind的第3个参数，对于sockaddr_in结构，应等于16，通常都使用C的sizeof操作符。

本地IP地址(sin_addr)可以指明为通配地址或某个本地 IP地址。本地端口号(sin_port)，可以等于0(告诉内核选择一个临时端口)或非0，如果应用进程希望指明端口号。通常情况下， TCP或UDP服务器指明一个通配 IP地址，端口号等于0。

22.6 应用进程可以bind一个本地广播地址，因为ifa_ifwithaddr(图22-22)的调用成功。它被用做在该插口上发送的 IP数据报的源地址。 C.2节中指出， RFC 1122不允许这种做法。但试图绑定 255.255.255.255时会失败，因为ifa_ifwithaddr不接受该地址。

第23章

23.1 sosend把用户数据放入单个的mbuf中，如果其长度小于等于100字节；放入两个mbuf中，如果长度小于等于207字节；否则，放入多个mbuf中，每个都带有一个簇。此外，如果长度小于100字节，sosend调用MH_ALIGN，希望能在mbuf起始处为协议首部保留空间。因为udp_output调用M_PREPEND，下述5种情况都是可能的：(1)如果用户数据长度小于等于72字节，一个mbuf就可以存放IP首部、UDP首部和数据；(2)如果长度位于73字节和100字节之间，sosend为用户数据分配一个mbuf，M_PERPEND为IP和TCP首部再分配一个mbuf；(3)如果长度位于101字节和207字节之间，sosend为用户数据分配两个mbuf，M_PREPEND为IP和TCP首部再分配一个mbuf；(4)如果长度位于208字节和MCLBYTES之间，sosend为用户数据分配一个带簇的mbuf，M_PERPEND为IP和TCP首部再分配一个mbuf；(5)如果长度超出，则sosend分配足够多的mbuf和簇，以存放数据(最大数据长度65507字节，需分配64个带1024字节簇的mbuf)，M_PERPEND为IP和TCP首部再分配一个mbuf。

23.2 IP选项提交给ip_output，后者调用ip_insertoptions在输出IP数据报中插入IP选项。它接着分配一个新的mbuf，存放带有IP选项的IP首部，如果第一个mbuf指向一个簇(UDP输出不可能出现这种情况)，或者第一个mbuf中没有足够的剩余空间存放新增选项。上个习题中给出的第一种情况中，选项大小将决定ip_insertoptions是否分配另一个mbuf：如果用户数据长度小于100-28-optlen(IP选项占用的字节数)，说明mbuf足够存放IP首部、IP选项、UDP首部和数据。第2、3、4和5种情况中，第一个mbuf都由M_PREPEND分配，只存放IP和UDP首部。M_PREPEND调用M_PREPEND，接着调用MH_ALIGN，把28字节的首部移到mbuf尾部，因此，第一个mbuf中必定有空间存放最大为40字节的IP选项。

23.3 不。函数in_pcbconnect只有在应用程序调用connect，或者在一个未连接的UDP插口上发送第一个数据报时，才会被调用。因为本地地址是通配地址，本地端口号等于0，所以in_pcbconnect给本地端口号赋一个临时端口(通过调用in_pcbbind)，并根据到达目的地的路由设定本地地址。

23.4 处理器优选级仍为splnet不变，没有还原为初始值，这是代码的一个差错。

23.5 不。in_pcbconnect不允许与等于0的端口建立连接。即使应用进程没有直接调用connect，也会间接地执行connect，因此，in_pcbconnect总会被调用。

23.6 应用程序必须调用ioctl，命令为SIOCGIFCONF，返回所有已配置的IP接口信息。之后，在ioctl返回的所有IP地址和广播地址中寻找接收数据报中的目的地址(也可不用ioctl，19.14节中介绍的sysctl系统调用也能够返回所有配置接口的信息)。

23.7 recvit释放带有控制信息的mbuf。

23.8 为了断开一个已建立连接的UDP插口，调用connect，传递一个无效的地址参数，如0.0.0.0，端口号等于0。因为插口已经建立了连接，soconnect调用sodisconnect，后者调用udp_usrreq，发送PRU_DISCONNECT请求，令远端地址等于0.0.0.0，远端端口号等于0。这样，接下来调用sendto时可以指明目的地

址。由于指明地址无效，sodisconnect发送的PRU_CONNECT请求失败。实际上，我们不希望connect成功，只是要执行PRU_DISCONNECT请求，而且通过connect来执行这一请求的做法是唯一可行的方案，因为插口API没有提供disconnect函数。

手册中关于connect(2)的描述通常包括下述说明："可通过把数据报插口连接到一个无效地址，如空地址，来断开其当前连接。"但没有明确指出调用connect时，如果传送的地址无效，会返回一个差错。"空地址"的含义也易造成混淆，它指IP地址0.0.0.0，而非bind的第二个参数的空指针。

23.9 因为in_pcbbind能够建立UDP插口与远端IP地址间的临时连接，情况与应用进程调用connect类似：如果某接口的目的IP地址与该接口的广播地址对应，则从该接口发送数据报。

23.10 服务器必须设定IP_RECVDSTADDR插口选项，并调用recvmsg从客户请求中获取目的IP地址。为了成为响应报文段中的源地址，必须将其绑定在插口上。由于一个插口只能bind一次，服务器每次响应时都必须创建新的插口。

23.11 注意，ip_output(图8-22)中，IP不修改调用者传递的DF比特。需要定义新的插口选项，促使udp_output在把数据报传递给IP之前，设定DF比特。

23.12 不。它只被udp_input使用，且应为该函数的局部变量。

第24章

24.1 状态为ESTABLISHED的连接总数为126 820。除以发送和接收的总字节数，得到每个方向上的平均字节数，约为30 000字节。

24.2 tcp_output中，保存IP和TCP首部的mbuf还有空间容纳链路层首部(max_linkhdr)。试图通过bcopy把IP和TCP首部原型复制到mbuf中是行不通的，因为有可能会把40字节的首部分散在两个mbuf中。尽管40字节的首部必须放入单个mbuf中，但链路层首部不存在这样的限制。不过这样做会降低性能，因为后续处理不得不为链路层首部再次分配mbuf。

24.3 在作者的bsdi系统中，计数器等于16，其中15个是标准系统守护程序(Telnet、Rlogin、FTP，等等)。而vangogh.cs.berkeley.edu系统，一个约有20个用户的中等规模的多用户系统，计数器约为60。对于大型的带有150个用户的多用户系统(world.std.com)，则有417个TCP端点和809个UDP端点。

第25章

25.1 图24-5中，2 592 000秒(30天)中出现了531 285次延迟ACK，平均每5秒钟有一次延迟ACK，或者说每25次调用tcp_fasttimo，才会有一次延迟ACK。这说明在代码检查所有TCP控制块，判定延迟ACK标志是否置位时，96%(25次中有24次)的时间都未置位。对于习题24.3中给出的大型的多用户系统，意味着需查看超过400个的TCP控制块，每秒钟查询5次。

另一种解决方案是，在需要延迟ACK时，设定全局标志。只有当全局标志置位时，才检查控制块列表。或者为需要延迟ACK的控制块单独建立并维护一个列表。例如，

图13-14中的变量igmp_timers_are_running。

25.2 这样使得变量tcp_keepintvl绑定在运行中的内核上，下次调用tcp_slowtimo时，内核可以改变tcp_maxidle的值。

25.3 t_idle中保存的实际上是从最后一次接收或发送报文段后算起的时间。因为TCP的输出必须被对端确认，与收到数据报文段相同，收到ACK也将清零t_idle。

25.4 图A-6给出代码的一种可能的重写方式。

```
case TCPT_2MSL:
    if (tp->t_state == TCPS_TIME_WAIT)
        tp = tcp_close(tp);
    else {
        if (tp->t_idle <= tcp_maxidle)
            tp->t_timer[TCPT_2MSL] = tcp_keepintvl;
        else
            tp = tcp_close(tp);
    }
    break;
```

图 A-6

25.5 如果收到了重复的ACK，t_idle等于150，但被复位为0。FIN_WAIT_2时钟超时后，t_idle将等于1048 (1198 - 150)，因此，定时器被设定为150个滴答。定时器再次超时，t_idle应等于1198+150，导致连接被关闭。重复ACK令时间延长，直到连接被关闭。

25.6 第一次连接探测报文段将在1小时后发送。应用进程设定该选项时，实际只置位了socket结构中的SO_KEEPALIVE选项。由于设定了该选项，定时器将于1小时后超时，图25-15中的代码将发送第一次连接探测报文段。

25.7 tcp_rttdflt用于为每条TCP连接初始化RTT估计器的值。如果需要，主机可通过更改全局变量，改变默认设置。如果通过#define将其定义为常量，则只有通过重新编译内核文件才能改变默认值。

第26章

26.1 事实上，TCP并没有刻意计算从连接上最后一次发送报文段算起的时间，因为连接上的计时器t_idle一直在起作用。

26.2 图25-26中，snd_nxt被设定为snd_una，len等于0。

26.3 如果运行Net/3系统，但对端主机却无法处理某个新选项(例如，对端拒绝建立连接，即使被要求忽略无法辨识的选项)。遇到这种情况时，通过在内核中改变这个全局变量的值，可以禁止某一个或两个选项。

26.4 时间戳选项能够在每次收到对新数据的ACK时，更新RTT估计器值。因此，使用时间戳选项后，RTT估计器值将被更新16次，是不使用该选项时更新次数的两倍。注意，在时刻217.944时，收到了对6145的ACK，RTT估计器值被更新，但这个新的计算值并不准确——或者是在时刻3.740发送的携带5633~6144字节的数据段，或者是收到的对6145的ACK，在网络中延迟了200秒。

26.5 这种存储器引用时，无法确保2字节的MSS值能够正确地对齐。

26.6 (该解决方案来自于 Dave Borman)一个数据段能够携带的 TCP数据量的最大值为65495字节，即65535减去IP和TCP首部的最小值(40)。因此，紧急数据偏移量可取值范围中有39个值是无意义的：65496~65535，包括65535。无论何时，只要发送方得到一个超过65495的紧急数据偏移量，则将其替换为 65535，并置位URG标志。从而迫使接收方进入紧急模式，并告知接收方紧急数据偏移量所指向的数据尚未被发送。发送方将持续发送紧急数据偏移量等于 65535、且URG标志置位的数据报文段，直到紧急数据偏移量小于等于65495，说明真正的紧急数据偏移量的开始。

26.7 我们提到，数据段的传输是可靠的 (重传机制)，而ACK则有可能丢失。RST报文段的传输同样也是不可靠的。如果连接上收到了一个假报文段 (例如，不属于本连接的报文段，或者一个不属于任何连接的报文段)，则传送RST报文段。如果RST报文段被ip_output丢弃，当对端重传导致发送 RST报文段的数据报文段时，将再次生成RST报文段。

26.8 应用程序执行了8次写入1024字节的操作。头4次调用sosend时，tcp_output被调用，报文段被发送。因为这4个报文段都包含了发送缓存中最后一个字节的数据，每个报文段的PSH标志都置位(图26-25)。第二个缓存装满后，应用进程进行下一次写操作，调用sosend时被挂起。收到对端通告窗口大小等于 0的ACK后，丢弃发送缓存中已被确认的4096字节的数据，应用进程被唤醒，又连续执行了 4次写操作，发送缓存再次被填满。但只有当接收方通告窗口大小不等于 0时，才能继续发送数据。条件满足时，接下来的 4个报文段被发送，但只有最后一个报文段的 PSH标志置位，因为前3个报文段并未清空发送缓存。

26.9 如果正在发送的报文段不属于任何连接，传给 tcp_respond的tp参数可以是空指针。代码只有在指针为空时，才会查看 tp，并代之以默认值。

26.10 tcp_output通常调用MGETHDR，分配一个仅能容纳IP和TCP首部的mbuf，参见图26-25和图26-26。在新的mbuf的前部，代码只预留了链路层首部(max_linkhdr)大小的空间。如果使用了 IP选项，而且选项的大小超过了max_linkhdr，ip_insertoptions会自动分配另一个mbuf。但如果IP选项的大小小于等于max_linkhdr，则ip_insertoptions也会占用mbuf首部的空间，从而导致ether_output仅为链路层首部分配另一个mbuf(假定以太网输出)。
为了避免多余的mbuf，图26-25和图26-26中的代码，可以在报文段中携带IP选项时调用MH_ALIGN。

26.11 约有80行代码，假定采用了RFC 1323中的时间戳选项，且报文段被计时。
宏MGETHDR调用了宏MALLOC，后者可能调用函数malloc。函数m_copy也会被调用，但一个完整大小的报文段可能需要一个簇，因此，不复制 mbuf，而是保存一个对簇的引用。m_copy中调用MGET，可能会导致对malloc的调用。函数bcopy复制模板，而in_cksum计算TCP的检验和。

26.12 调用writev没有区别，因为处理逻辑由 sosend实现。因为数据大小等于 150字节，小于MINCLSIZE (208)，所以为头100个字节分配了一个 mbuf。并且因为协议支持数据的分段，PRU_SEND请求被发送。接着为剩余的 50字节再分配一个mbuf，并发送相应的PRU_SEND请求(对于PR_ATOMIC协议，如UDP，writev只

生成一条“记录”，即只发送一个PRU_SEND请求。)
如果两个缓存的长度分别等于200和300，总长度超过了MINCLSIZE，则分配一个mbuf簇，且只发送一次PRU_SEND请求。TCP只生成一个500字节的报文段。

第27章

27.1 表中前6行记录的差错，都是由于接收报文段或者定时器超时引起的异步差错。通过在so_error中保存非零的差错代码，应用进程能够在下一次读/写操作中收到差错信息。但如果调用来自tcp_disconnect，说明应用进程调用了close，或者应用进程终止时系统自动关闭其所拥有的描述符。无论是哪一种情况，描述符被关闭，应用进程不可能再通过读/写操作来获取差错代码。此外，因为应用进程必须明确设定插口选项，强迫RST置位，此时返回一个差错代码并不能向应用进程提供有用的信息。

27.2 假定它是32 bit的u_long，最大值小于4298秒(1.2小时)。

27.3 路由表中的统计数据由tcp_close更新，但只有当连接进入CLOSED状态时，它才会被调用。因为FTP客户终止向对端发送数据(执行主动关闭)，本地连接端点进入TIME_WAIT状态。必须经过2MSL后，路由表统计值才会被更新。

第28章

28.1 0、1、2和3。

28.2 34.9Mb/s。对于更高的速率，连接两端需要更大的缓存。

28.3 通常，tcp_dooption不知道两个时间戳值是否按32 bit边界对齐。图28-4中的代码，在指定情况下，能够确认时间戳值按32 bit边界对齐，从而避免调用bcopy。

28.4 图28-4中实现“选项预测”代码，只能处理系统推荐的格式。如果连接对端未采用系统推荐的格式，会导致为每个接收到的报文段调用tcp_dooptions，降低了处理速度。

28.5 如果在每次创建插口时，而非每次连接建立时，调用tcp_template，则系统中的每个监听服务器都会拥有一个tcp_template，而该结构可能永远不会被使用。

28.6 时间戳时钟频率应该在1 b/ms和1 b/s之间(Net/3采用了2 b/s)。如果采用最高的时钟频率1 b/ms，32 bit的时间戳将在2^{31}/(24×60×60×1000)天，即24.8天后发生符号位回绕。

28.7 如果频率为每500 ms 1 bit，32 bit的时间戳将在2^{31}/(24×60×60×2)天，即12 427天，约34年后才会出现符号位回绕。

28.8 对RST报文段的处理应优先于时间戳，而且，RST报文段中最好不携带时间戳选项(图26-24中的tcp_input代码确保了这一点)。

28.9 因为客户端状态为ESTABLISHED，处理将在图28-24的代码处结束。todrop等于1，因为rcv_nxt在收到第一个SYN时已递增过。SYN标志被清除(因为这是一个重复报文段)，ti_seq递增，todrop减为0。因为todrop和ti_len都等于0，执行图28-25起始处的if语句，并跳过下一个if语句，直接调用m_adj。但下一章中介绍tcp_input后续代码时，将谈到在某些情况下不会调用tcp_output，本题即是一例。因此，客户端会不响应重复的SYN/ACK。服务器端超时后，再次发送SYN/ACK

(图28-17中介绍了某个被动打开的插口收到 SYN时，定时器的设置)，这个重发的SYN/ACK报文段同样被忽略。我们现在讨论的其实是图28-25代码中的另一个差错，图28-30中给出的代码同样也纠正了这一差错。

28.10 客户发出的SYN到达服务器，并被交给处于 TIME_WAIT状态的插口。图28-24中的代码关闭SYN标志，图28-25中的代码跳转至dropafterack，丢弃该报文段，但生成一个 ACK，确认字段等于 rcv_nxt(图26-27)。它被称作“再同步(resynchronization) ACK”报文段，因为其目的是告诉对端本地希望接收的下一序号。客户端收到此ACK后(客户处于SYN_SENT状态)，发现它的确认字段所携带的序号不等于自己期待得到的序号后 (图28-18)，向服务器发送RST报文段。RST报文段的ACK标志被清除，且序号等于再同步 ACK报文段中确认字段携带的序号(图29-28)。服务器收到此RST报文段后，其TIME_WAIT状态提前终止，相应插口被关闭(图28-36)。客户端6秒钟后超时，重传SYN报文段。假定监听服务器进程在服务器主机上运转正常，新的连接将建立。由于TIME_WAIT状态的这种防护作用，新连接建立时，下一个 SYN报文段携带的序号既可以高于前一连接上最后收到的序号(图28-28中的测试)，也可以低于该序号。

第29章

29.1 假定RTT等于2秒钟。服务器被动打开，客户端在时刻 0主动打开。服务器在时刻 1收到客户发出的SYN，并作出响应，发送自己的SYN和对客户SYN的ACK。客户端在时刻2收到报务器的响应报文段，图 28-20中的代码调用soisconnected(唤醒客户进程)，完成主动打开过程，并向服务器发送 ACK响应。服务器在时刻3收到客户的ACK，图29-2中的代码完成服务器端的被动打开过程，控制返回给服务器进程。一般情况下，客户进程比服务器进程提早 1/2 RTT时间得到控制。

29.2 假定SYN的序号等于1000，50字节数据的序号等于1001~1050。tcp_input处理此SYN报文段时，首先执行图28-15中的起始case语句，令rcv_nxt等于1001，接着跳到step6。图29-22中的代码调用cp_reass，把数据放入插口的重组队列中。但数据还不能放入插口的接收缓存 (图27-23)，因此，rcv_nxt还是等于 1001。在调用tcp_output生成ACK响应时，rcv_nxt (1001)被放入ACK报文段的确认字段。也说是说，SYN被确认，但与之同时到达的50字节的数据没有被确认。因此，客户端不得不重发50字节的数据，所以，在完成主动打开的SYN报文段中携带数据是没有意义的。

29.3 客户端的ACK/FIN报文段到达时，服务器处于 SYN_RCVD状态，因此，图29-2中的tcp_input代码将结束对 ACK的处理。连接转移到 ESTABLISHED状态，tcp_reass把已在重组队列中的数据放入接收缓存，rcv_nxt递增为1051。tcp_input继续执行，图29-24中的代码负责处理FIN标志，此时TF_ACKNOW标志置位，rcv_nxt等于1052。socantrcvmove设定插口的状态，使服务器在读取50字节的数据之后，得到“文件结束”指示。服务器的插口也转移到CLOSE_WAIT状态。调用tcp_output，确认客户端的 FIN(因为rcv_nxt等于1052)。假定服务器进程在收到“文件结束”指示后，关闭其插口，服务器也将向

客户发送FIN，并等待回应。

在这个例子中，这了从客户端向服务器传送50字节的数据，双方需3个来回，共发送6个报文段。为了减少所需的报文段数，应采用“用于交易的TCP扩展[Braden 1994]”。

29.4 收到服务器响应时，客户插口处于 SYN_SENT状态。图28-20中的代码处理该报文段，连接转移到ESTABLISHED状态，控制跳转到step6，由图29-22中的代码继续处理数据。TCP_REASS把数据添加到插口的接收缓存，并递增 rcv_nxt。之后，图29-24中的代码开始处理FIN，再次递增rcv_nxt，连接转移到CLOSE_WAIT状态。在调用tcp_output时，rcv_nxt同时确认了SYN、50字节的数据和FIN。随后的客户进程首先读取50字节的数据，接着是“文件结束”指示，并可能关闭其插口。客户端连接进入LAST_ACK状态，向服务器发送FIN报文段，并等待其响应报文段。

29.5 问题出在图24-16中的tcp_outflags[TCPS_CLOSING]。它设定了TH_FIN标志，而状态变迁图(图24-15)并未规定FIN应被重传。解决问题的方法是，从该状态的tcp_outflags中除去TH_FIN标志。这个问题没有什么危害——只不过多交换两个报文段——而且同时关闭或者在关闭后紧接着自连接的情况是非常罕见的。

29.6 没有。系统调用write返回OK，只说明数据已复制到插口的缓存中。在数据得到对端确认时，Net/3不再通知应用进程。如果需要得到此类信息，应设计并实现应用级的确认机制。

29.7 RFC 1323的时间戳选项，造成“首部压缩”失效。因为只要时间戳变化，即 TCP选项发生了改变，报文段发送时就不会被压缩。窗口大小选项无效，因为 TCP首部中值的长度仍为16 bit。

29.8 IP 中ID字段的取值来自一个全局变量，只要发送一个 IP数据报，该变量递增一次。这种方式导致在同一 TCP连接上两个连续TCP报文段间的ID差值大于1的可能性大大增加。一旦ID差值大于1，图29-34中的Δ*ipid*字段将被发送，增大了压缩首部的大小。一个更好的解决方案是，TCP自己维护一个计数器，用于ID的赋值。

第30章

30.2 是的，仍会发送RST报文段。应用进程终止的处理中包括关闭它打开的所有描述符。同一个函数(soclose)最终会被调用，无论是应用进程明确地关闭了插口描述符，还是隐含地进行了关闭(首先被终止)。

30.3 不。这个常量只有在监听插口设定 SO_LINGER选项，且延迟时间等于0时，才会被用到。正常情况下，插口选项的这种设定方式会导致在连接关闭时发送 RST报文段(图30-12)，但图30-2中对于接收连接请求的监听插口，把该值从 0改为120(滴答)。

30.4 如果这是第一次使用默认路由，则为两次；否则为一次。当创建插口时，in_pcballoc将Internet PCB置为0，从而将PCB结构中的route结构设为0。发送第一个报文段(SYN)时，tcp_output调用ip_output。因为ro_rt指针为空，因此向ro_dst填充IP数据报的目的地址，并调用 rtalloc。在该连接的PCB中，route结构的ro_rt变量中保存默认路由。当ip_output调用ether_output时，后者检查路由表中的rt_gwroute变量是否为空。如果是，则调用 rtalloc1。假

定路由没有改变，该连接每次调用tcp_output时，都会使用保存的ro_rt指针，以避免多余的路由表查询。

第31章

31.1 因为在bpf_wakeup调用唤醒任何沉睡进程之前，catchpacket肯定会结束。

31.2 打开BPF设备的应用进程可能调用fork，导致多个应用进程都有权访问同一个BPF设备。

31.3 只有支持BPF的设备才会出现在BPF接口表(bpf_iflist)中，因此，如果无法找到指定接口，bpf_setif将返回ENXIO。

第32章

32.1 在第一个例子中等于0，第二个例子中等于255。这些值都是RFC 1700 [Reynolds和Postel 1994]中的保留值，不应出现在数据报中。也就是说，如果某个插口创建时的协议号设定为IPPROTO_RAW，则必须设定其IP_HDRINCL插口选项，且写入到该插口的数据报必须拥有一个有效的协议值。

32.2 因为IP协议值255是保留值，不会出现在网络中传送的数据报中。但这又是一个非零的协议值，rip_input的3项测试中的第一项测试将忽略所有协议值不等于255的数据报。因此，应用进程无法在该插口上收到任何数据报。

32.3 即使该协议值是一个保留值，不会出现在网络中传送的数据报中，但rip_input的3项测试中的第一项测试保证此类型的插口能够接收任何协议类型的数据报。如果应用进程调用了connect或者bind，或者两者都调用，对于此种原始插口而言，对输入的唯一限制是IP报的源地址和目的地址。

32.4 因为ip_protox数组(图7-22)保存了有关内核所能支持的协议类型的信息，只有在该协议既没有相关的原始监听插口，而且指针inetsw[ip_protox[ip->ip_p]].pr_input等于rip_input时，才会生成ICMP差错报告。

32.5 两种情况下，应用进程都必须自己构造IP首部，以及其后的内容(UDP报文段，TCP报文段或任何其他的报文段)。对于原始IP插口，输出时同样调用sendto，通过Internet插口地址结构指明目的IP地址。调用ip_output，并依据给定的目的IP地址执行正常的IP选路。

BPF要求应用进程提供完整的数据链路层首部，例如以太网首部。输出时，需调用write，因为无法指明目的地址。数据分组被直接交给接口输出函数，跳过ip_output函数(图31-20)。应用进程通过BIOCSETIFioctl(图31-16)选择外出接口。因为未执行IP选路，数据帧只能发给是直接相连的网络上的另一个主机(除非应用进程重复IP选路函数，并将数据帧发给直接相联网络上的某个路由器，由路由器根据目的IP地址完成转发)。

32.6 原始IP插口只能接收具有内核不处理的协议类型的数据报，例如，应用进程无法在原始插口上接收TCP报文段或UDP报文段。

BPF能够接收到达指定接口的所有数据帧，无论它们是否是IP数据报。BIOCPROMISCioctl使接口处于一种混杂状态，甚至能够接收不是发给本主机的数据报。

附录B　源代码的获取

URL：统一资源定位符

本附录列出源代码所在的网址和下载方式。例如，常见的“匿名 FTP”地址表示如下：

```
ftp://ftp.cdrom.com/pub/bsd-sources/4.4BSD-Lite.tar.gz
```

即主机为`ftp.cdrom.com`。通过匿名FTP客户登录后，从目录`pub/bsd-sources`下载文件`4.4BSD-Lite.tar.gz`。后缀`.tar`说明文件以标准的`tar`(1)格式存储，`.gz`说明文件由`GNU gzip`(1)程序压缩。

4.4BSD-Lite

有多种方式可得到 4.4BSD-Lite的正式版代码。完整的 4.4BSD-Lite正式版代码可通过Walnut Creek CD-ROM公司得到，网址为

```
ftp://ftp.cdrom.com/pub/bsd-sources/4.4BSD-Lite.tar.gz
```

或者直接得到其光碟版。联系电话为 18007869907或+1510 674 0783。

O’Reilly & Associates出版的CD-ROM，包括全套的4.4BSD手册和4.4BSD-Lite正式版代码。联系电话为1 800 889 8989或者+1 707 829 0515。

运行4.4BSD-Lite 网络软件的操作系统

4.4BSD-Lite正式版不是一个完整的操作系统。为了测试本书中介绍的网络软件，需要内置4.4BSD-Lite正式版的操作系统，或者支持 4.4BSD-Lite的操作系统。

作者使用的操作系统是 Berkeley Software Design Inc.生产的商用系统，联系电话为 1 800 ITS BSD8，+1 719 260 8114，或者`info@bsdi.com`。

还有些免费的操作系统，已内置了 4.4BSD-Lite，如NetBSD、386BSD和FreeBSd。详情请见Walnut Creek CD-ROM(`ftp.cdrom.com`)或者`comp.os.386bsd` Usenet新闻组。

RFC

所有RFC都是免费的，通过电子邮件或匿名 FTP服务器可从英特网上得到所需文档。向下述地址发送电子邮件：

```
To: rfc-info@ISI.EDU.
Subject: getting rfcs
help: ways_to_get_rfcs
```

回复邮件中会列出通过电子邮件或匿名 FTP服务器获取RFC不同方法的详细说明。

记住，首先应先下载最新的 RFC 索引，从中查找所需的 RFC，确认所需的RFC没有被新的RFC更新或取代。

GNU软件

利用GNU Indent程序对本书出现的所有源代码进行格式调整，并利用 GNU Gzip程序对文件做了压缩。这些程序可在下列站点找到：

```
ftp://prep.ai.mit.edu/pub/gnu/indent-1.9.1.tar.gz
ftp://prep.ai.mit.edu/pub/gnu/gzip-1.2.2.tar
```

文件名中的数字随版本的不同而不同。此外还有用于其他操作系统的 Gzip程序，如MS-DOS。

英特网上还有许多其他站点也提供 GNU资源，`prep.ai.mit.edu`主机的问候词中列出了这些站点的名称。

PPP软件

有些PPP实现是免费的。`comp.protocols.ppp` FAQ的第5部分提供了很多有价值的信息。

```
http://cs.uni-bonn.de/ppp/part5.html
```

mrouted软件

`mrouted`软件的最新版本和其他多播应用程序可从 Xerox Palo Alto研究中心的站点得到：

```
ftp://parcftp.xerox.com/pub/net-research/
```

ISODE软件

ISODE软件包中的SNMP代理实现与 Net/3兼容。详细信息参见ISODE论坛的网站：

```
http://www.isode.com/
```

第一部分　TCP事务协议

第1章　T/TCP　概　述

1.1　概述

本章首先介绍客户-服务器事务概念。我们从使用 UDP的客户-服务器应用开始，这是最简单的情形。接着我们编写使用 TCP的客户和服务器程序，并由此考察两台主机间交互的TCP/IP分组。然后我们使用T/TCP，证明利用T/TCP可以减少分组数，并给出为利用 T/TCP需要对两端的源代码所做的最少改动。

接下来介绍了运行书中示例程序的测试网络，并对分别使用 UDP、TCP和T/TCP的客户-服务器应用程序进行了简单的时间耗费比较。我们考察了一些使用 TCP的典型Internet应用程序，看看如果两端都支持 T/TCP，将需要做哪些修改。紧接着，简要介绍了 Internet协议族中事务协议的发展历史，概略叙述了现有的 T/TCP实现。

本书全文以及有关T/TCP的文献中，事务一词的含义都是指客户向服务器发出一个请求，然后服务器对该请求作出应答。 Internet中最常见的一个例子是，客户向域名服务器 (DNS)发出请求，查询域名对应的 IP地址，然后域名服务器给出响应。本书中的事务这个术语并没有数据库中的事务那样的含义：加锁、两步提交、回退，等等。

1.2　UDP上的客户-服务器

我们先来看一个简单的 UDP客户-服务器应用程序的例子，其客户程序源代码如图 1-1所示。在这个例子中，客户向服务器发出一个请求，服务器处理该请求，然后发回一个应答。

udpcli.c

```
 1 #include    "cliserv.h"

 2 int
 3 main(int argc, char *argv[])
 4 {                                   /* simple UDP client */
 5     struct sockaddr_in serv;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n;

 8     if (argc != 2)
 9         err_quit("usage: udpcli <IP address of server>");

10     if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
11         err_sys("socket error");

12     memset(&serv, 0, sizeof(serv));
13     serv.sin_family = AF_INET;
14     serv.sin_addr.s_addr = inet_addr(argv[1]);
15     serv.sin_port = htons(UDP_SERV_PORT);
```

图1-1　UDP上的简单客户程序

```
16      /* form request[] ... */
17      if (sendto(sockfd, request, REQUEST, 0,
18                 (SA) &serv, sizeof(serv)) != REQUEST)
19          err_sys("sendto error");
20      if ((n = recvfrom(sockfd, reply, REPLY, 0,
21                        (SA) NULL, (int *) NULL)) < 0)
22          err_sys("recvfrom error");
23      /* process "n" bytes of reply[] ... */
24      exit(0);
25  }
```

udpcli.c

图1-1 (续)

本书中所有源代码的格式都是这样。每一非空行前面都标有行号。正文中叙述某段源代码时，这段源代码的起始和结束行号标记于正文段落的左边，如下面的正文所示。有时这些段落前面会有一小段说明，对所描述的源代码进行概要说明。源代码段开头和结尾处的水平线标明源代码段所在的文件名。这些文件名通常都是指我们在1.9节中将介绍的4.4版BSD-Lite中发布的文件。

我们来讨论这个程序的一些有关特性，但不详细描述插口函数，因为我们假设读者对这些函数有一些基本的认识。关于插口函数的细节在参考书 [Stevens 1990]的第6章中可以找到。图1-2给出了头文件`cliserv.h`。

1. 创建UDP插口

10-11 `socket`函数用于创建一个UDP插口，并将一个非负的插口描述符返回给调用进程。出错处理函数`err_sys`参见参考书[Stevens 1992]的附录B.2。这个函数可以接受任意数目的参数，但要用`vsprintf`函数对它们格式化，然后这个函数会打印出系统调用所返回的`errno`值所对应的Unix出错信息，然后终止进程。

2. 填写服务器地址

12-15 首先用`memset`函数将Internet插口地址结构清零，然后填入服务器的IP地址和端口号。为简明起见，我们要求用户在程序运行中通过命令行输入一个点分十进制数形式的IP地址(`argv[1]`)。服务器端口号(`UDP_SERV_PORT`)在头文件`cliserv.h`中用`#define`定义，在本章的所有程序首部中都包含了该头文件。这样做是为了使程序简洁，并避免使调用`gethostbyname`和`getservbyname`函数的源代码复杂化。

3. 构造并向服务器发送请求

16-19 客户程序构造一个请求(只用一行注释来表示)，并用`sendto`函数将其发出，这样就有一个UDP数据报发往服务器。同样是为了简明起见，我们假设请求(`REQUEST`)和应答(`REPLY`)的报文长度为固定值。实用的程序应当按照请求和应答的最大长度来分配缓存空间，但实际的请求和应答报文长度是变化的，而且一般都比较小。

4. 读取和处理服务器的应答

20-23 调用`recvfrom`函数将使进程阻塞(即置为睡眠状态)，直至收到一个数据报。接着客户进程处理应答(用一行注释来表示)，然后进程终止。

由于`recvfrom`函数中没有超时机制，请求报文或应答报文中任何一个丢失都将造成该进程永久挂起。事实上，UDP客户-服务器应用的一个基本问题就是对现实世界中的此类错误缺少健壮性。在本节的末尾将对这个问题做更详细的讨论。

在头文件cliserv.h中，我们将SA定义为struct sockaddr *；即指向一般的插口地址结构的指针。每当有一个插口函数需要一个指向插口地址结构的指针时，该指针必须被置为指向一个一般性插口地址结构的指针。这是由于插口函数先于ANSI C标准出现，在80年代早期开发插口函数的时候，void*(空类型)指针类型尚不可用。问题是，“struct sockaddr *”总共有17个字符，这经常使这一行源代码超出屏幕(或书本页面)的右边界，因此我们将其缩写成为 SA。这个缩写是从BSD内核源代码中借用过来的。

图1-2给出了在本章所有程序中都包含的头文件 cliserv.h。

cliserv.h

```
 1 /* Common includes and defines for UDP, TCP, and T/TCP
 2  * clients and servers */

 3 #include    <sys/types.h>
 4 #include    <sys/socket.h>
 5 #include    <netinet/in.h>
 6 #include    <arpa/inet.h>
 7 #include    <stdio.h>
 8 #include    <stdlib.h>
 9 #include    <string.h>
10 #include    <unistd.h>
11 #define REQUEST 400             /* max size of request, in bytes */
12 #define REPLY   400             /* max size of reply, in bytes */

13 #define UDP_SERV_PORT   7777    /* UDP server's well-known port */
14 #define TCP_SERV_PORT   8888    /* TCP server's well-known port */
15 #define TTCP_SERV_PORT  9999    /* T/TCP server's well-known port */

16     /* Following shortens all the type casts of pointer arguments */
17 #define SA  struct sockaddr *

18 void    err_quit(const char *,...);
19 void    err_sys(const char *,...);
20 int     read_stream(int, char *, int);
```

cliserv.

图1-2 本章各程序中均包含的头文件 cliserv.h

图1-3给出了相应的UDP服务器程序。

udpserv.

```
 1 #include    "cliserv.h"

 2 int
 3 main()
 4 {                               /* simple UDP server */
 5     struct sockaddr_in serv, cli;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n, clilen;

 8     if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
 9         err_sys("socket error");

10     memset(&serv, 0, sizeof(serv));
11     serv.sin_family = AF_INET;
12     serv.sin_addr.s_addr = htonl(INADDR_ANY);
13     serv.sin_port = htons(UDP_SERV_PORT);
```

图1-3 与图1-1的UDP客户程序对应的UDP服务器程序

```
14      if (bind(sockfd, (SA) &serv, sizeof(serv)) < 0)
15          err_sys("bind error");

16      for (;;) {
17          clilen = sizeof(cli);
18          if ((n = recvfrom(sockfd, request, REQUEST, 0,
19                            (SA) &cli, &clilen)) < 0)
20              err_sys("recvfrom error");

21          /* process "n" bytes of request[] and create reply[] ... */

22          if (sendto(sockfd, reply, REPLY, 0,
23                     (SA) &cli, sizeof(cli)) != REPLY)
24              err_sys("sendto error");
25      }
26  }
```

udpserv.c

图1-3 (续)

5. 创建UDP插口和绑定本机地址

8-15 调用socket函数创建一个UDP插口，并在其Internet插口地址结构中填入服务器的本机地址。这里本机地址设置为通配符 (INADDR_ANY)，这意味着服务器可以从任何一个本机接口接收数据报(假设服务器是多宿主的，即可以有多个网络接口)。端口号设为服务器的知名端口(UDP_SERV_PORT)，该常量也在前面讲过的头文件 cliserv.h中定义。本机IP地址和知名端口用bind函数绑定到插口上。

6. 处理客户请求

16-25 接下来，服务器程序就进入一个无限循环：等待客户程序的请求到达 (recvfrom)，处理该请求(我们只用一行注释来表示处理动作)，然后发出应答(sendto)。

这只是最简单的UDP客户 - 服务器应用。实际中常见的例子是域名服务系统 (DNS)。DNS客户(称作解析器)通常是一般客户应用程序 (例如，Telnet客户、FTP客户或WWW浏览器)的一个部分。解析器向 DNS服务器发出一个UDP数据报，查询某一域名对应的 IP地址。服务器发回的应答通常也是一个UDP数据报。

如果观察客户向服务器发送请求时双方交换的分组，我们就会得到图 1-4这样的时间系列，页面上时间自上而下递增。服务器程序先启动，其行为过程给在图 1-4的右半部，客户程序稍后启动。

我们分别来看客户和服务器程序中调用的函数及其相应内核执行的动作。在对 socket函数的两次调用中，上下紧挨着的两个箭头表示内核执行请求的动作并立即返回。在调用sendto函数时，尽管内核也立即返回，但实际上已经发出了一个 UDP数据报。为简明起见，我们假设客户程序的请求和服务器程序的应答所生成的 IP数据报的长度都小于网络的最大传输单元(MTU)，IP数据报不必分段。

在这个图中，有两次调用 recvfrom函数使进程睡眠，直到有数据报到达才被唤醒。我们把内核中相应的例程记为sleep和wakeup。

最后，我们还在图中标出了事务所耗费的时间。图 1-4的左侧标示的是客户端测得的事务时间：从客户发出请求到收到服务器的应答所经历的时间。组成这段事务时间的数值标在图的右侧：RTT + SPT，其中RTT是网络往返时间，SPT是服务器处理客户请求的时间。UDP客户 - 服务器事务的最短时间就是 RTT + SPT。

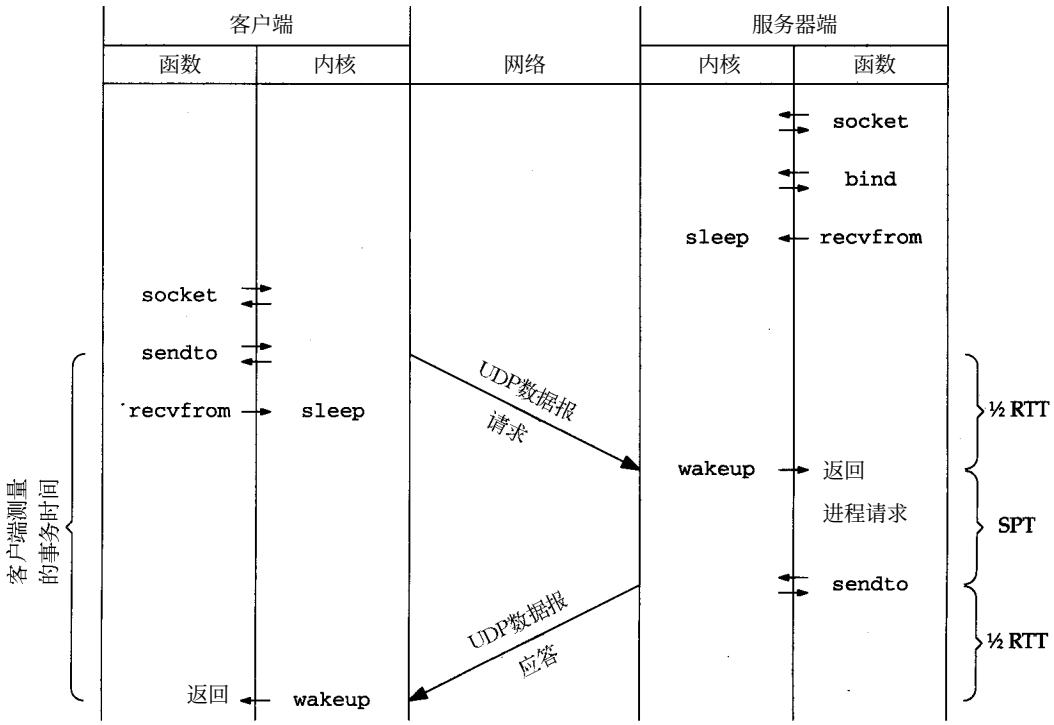


图1-4 UDP客户-服务器事务的时序图

尽管没有明确说明，但我们已经假设从客户到服务器的路径需要 ½ RTT时间，返回的路径又需½ RTT时间。但实际情况并非总是如此。据对大约 600条Internet路径的研究[Paxson 1995b]发现：30%的路径呈现明显的不对称性，说明两个方向上的路由经过了不同的站点。

我们的UDP客户-服务器看起来非常简捷 (每个程序只有大约30行有关网络的源代码)，但在实际环境中应用还不够健壮。由于 UDP是不保证可靠的协议，数据报可能会丢失、失序或重复，因此实用的应用程序必须处理这些问题。这通常是在客户程序调用 `recvfrom`时设置一个超时定时器，用以检测数据报的丢失，并重传请求。如果要使用超时定时器，客户程序就要测量RTT并动态更新，这是因为互连网上的RTT会在很大范围内变化，并且变化很快。但如果是服务器的应答丢失，而不是请求，那么服务器就要再次处理同一个请求，这可能会给某些服务带来问题。解决这个问题的办法之一是让服务器将每个客户最近一次请求的响应暂存起来，必要时重传这个应答即可，而不需要再次处理这个请求。最后，典型的情况是，客户向服务器发送的每个请求中都有一个不同的标识，服务器把这个标识在响应中传回来，使客户能把请求和响应匹配起来。在参考书 [Stevens 1990]的 8.4节中给出了UDP上的客户-服务器处理这些问题的源代码细节，但这将在程序中增加大约 500行源代码。

一方面，许多UDP应用程序都通过执行所有这些额外步骤 (超时机制、RTT值测量、请求标识，等等)来增加可靠性；另一方面，随着新的 UDP应用程序不断出现，这些步骤也在不断地推陈出新。参考书 [Patridge 1990b]中指出，“为了开发‘可靠的UDP应用程序’，你要有状态信息(序列号、重传计数器和往返时间估计器)，原则上你要用到当前TCP连接块中的全部信

息。因此，构筑一个‘可靠的UDP’，本质上和开发TCP一样难”。

有些应用程序并不实现上面所述的所有步骤：例如在接收时使用超时机制，但并不测量RTT值，当然更不会动态地更新RTT值。这样，当应用程序从一个环境(比如局域网)移植到另一个环境(比如广域网)中应用时，就可能会引发一些问题。比较好的解决办法是用TCP而不是用UDP，这样就可以利用TCP提供的所有可靠传输特性。但是这种办法会使客户端测得的事务时间由RTT + SPT增加到2×RTT + SPT(见下一节)，而且还会大大增加两个系统之间交换的分组数目。对付这些新的问题也有一个办法，即用T/TCP取代TCP，我们将在1.4节中对此进行讨论。

1.3 TCP上的客户-服务器

下一个例子是TCP上的客户-服务器事务应用。图1-5给出了客户程序。

tcpcli.c

```
 1 #include    "cliserv.h"
 2 int
 3 main(int argc, char *argv[])
 4 {                                   /* simple TCP client */
 5     struct sockaddr_in serv;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n;
 8     if (argc != 2)
 9         err_quit("usage: tcpcli <IP address of server>");
10     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");
12     memset(&serv, 0, sizeof(serv));
13     serv.sin_family = AF_INET;
14     serv.sin_addr.s_addr = inet_addr(argv[1]);
15     serv.sin_port = htons(TCP_SERV_PORT);
16     if (connect(sockfd, (SA) &serv, sizeof(serv)) < 0)
17         err_sys("connect error");
18     /* form request[] ... */
19     if (write(sockfd, request, REQUEST) != REQUEST)
20         err_sys("write error");
21     if (shutdown(sockfd, 1) < 0)
22         err_sys("shutdown error");
23     if ((n = read_stream(sockfd, reply, REPLY)) < 0)
24         err_sys("read error");
25     /* process "n" bytes of reply[] ... */
26     exit(0);
27 }
```

tcpcli.c

图1-5 TCP事务的客户

1. 创建TCP插口和连接到服务器

10-17 调用socket函数创建一个TCP插口，然后在Internet插口地址结构中填入服务器的IP地址和端口号。对connect函数的调用启动TCP的三次握手过程，在客户和服务器之间建立起连接。卷1的第18章给出了TCP连接建立和释放过程中交换分组的详细情况。

2. 发送请求和半关闭连接

19-22 客户的请求是用write函数发给服务器的。之后客户调用shutdown函数(函数的第2

个参数为1)关闭连接的一半，即数据流从客户向服务器的方向。这就告知服务器客户的数据已经发完了：从客户端向服务器传递了一个文件结束的通知。这时有一个设置了 FIN标志的TCP报文段发给服务器。客户此时仍然能够从连接中读取数据——只关闭了一个方向的数据流。这就叫做TCP的半关闭(half-close)。卷1的第18.5节给出了有关细节。

3. 读取应答

23-24 读取应答是由函数read_stream完成的，如图1-6所示。由于TCP是一个面向字节流的协议，没有任何形式的记录定界符，因而从服务器端 TCP传回的应答可能会包含在多个TCP报文段中。这也就可能会需要多次调用 read函数才能传递给客户进程。而且我们知道，当服务器发送完应答后就会关闭连接，使得 TCP向客户端发送一个带FIN的报文段，在read函数中返回一个文件结束标志 (返回值为0)。为了处理这些细节问题，在 read_stream函数中不断调用read函数直到接收缓存满或者read函数返回一个文件结束标志。 read_stream函数的返回值就是读取到的字节数。

tcpcli.c

```
 1 #include    "cliserv.h"
 2 int
 3 main(int argc, char *argv[])
 4 {                                   /* simple TCP client */
 5     struct sockaddr_in serv;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n;
 8     if (argc != 2)
 9         err_quit("usage: tcpcli <IP address of server>");
10     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");
12     memset(&serv, 0, sizeof(serv));
13     serv.sin_family = AF_INET;
14     serv.sin_addr.s_addr = inet_addr(argv[1]);
15     serv.sin_port = htons(TCP_SERV_PORT);
```

图1-6 read_stream 函数

还有一些别的方法可以在类似 TCP这样的流协议中用来给记录定界。许多Internet应用程序(FTP、SMTP、HTTP和NNTP)使用回车和换行符来标记记录的结束。其他一些应用程序(DNS，RPC)则在每个记录的前面加上一个定长的记录长度字段。在我们的例子中，利用了TCP的文件结束标志(FIN)，因为在每次事务中客户只向服务器发送一个请求，而服务器也只发回一个应答。 FTP也在其数据连接中采用了这项技术，用以告知对方文件已经结束。

图1-7给出的是TCP的服务器程序。

tcpserv.c

```
 1 #include    "cliserv.h"
 2 int
 3 main()
 4 {                                   /* simple TCP server */
 5     struct sockaddr_in serv, cli;
 6     char    request[REQUEST], reply[REPLY];
```

图1-7 TCP事务的服务器程序

```
 7      int     listenfd, sockfd, n, clilen;

 8      if ((listenfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
 9          err_sys("socket error");

10      memset(&serv, 0, sizeof(serv));
11      serv.sin_family = AF_INET;
12      serv.sin_addr.s_addr = htonl(INADDR_ANY);
13      serv.sin_port = htons(TCP_SERV_PORT);

14      if (bind(listenfd, (SA) &serv, sizeof(serv)) < 0)
15          err_sys("bind error");

16      if (listen(listenfd, SOMAXCONN) < 0)
17          err_sys("listen error");

18      for (;;) {
19          clilen = sizeof(cli);
20          if ((sockfd = accept(listenfd, (SA) &cli, &clilen)) < 0)
21              err_sys("accept error");

22          if ((n = read_stream(sockfd, request, REQUEST)) < 0)
23              err_sys("read error");

24          /* process "n" bytes of request[] and create reply[] ... */

25          if (write(sockfd, reply, REPLY) != REPLY)
26              err_sys("write error");

27          close(sockfd);
28      }
29  }
```

tcpserv.c

图1-7 (续)

4. 创建监听用TCP插口

8-17 用于创建一个TCP插口，并将服务器的知名端口绑定到该插口上。与UDP服务器一样，TCP服务器也将通配符作为其IP地址。调用listen函数将新创建的插口作为监听插口，用于等待客户端发起的连接。listen函数的第二个参数规定了允许的最大挂起连接数，内核要为该插口将这些连接进行排队处理。

SOMAXCONN在头文件<sys/socket.h>中定义。其数值过去一直都取5，但现在有一些比较新的系统将其定为10。对于一些很繁忙的服务器(例如：Web服务器)，已经发现需要取更大的值，比如256或1024。在14.5节中我们还将对此问题进行更多的讨论。

5. 接受连接和处理请求

18-28 服务器进程调用accept函数后就进入阻塞状态，直到有客户进程调用connect函数而建立起一个连接。函数accept返回一个新的插口描述符sockfd，代表与客户和服务器之间所建立的连接。服务器调用函数read_stream读取客户的请求(图1-6)，再调用write函数向客户发送应答。

这是一个反复循环的服务器：把当前的客户请求处理完毕后才又调用accept去接受另一个客户的连接。并发服务器可以并行地处理多个客户请求(即：同时处理)。在Unix的主机上实现并发服务器的常用技术是：在accept函数返回后，调用Unix的fork函数创建一个子进程，由子进程处理客户的请求，父进程则紧接着又调用accept去接受别的客户连接。实现并发服务器的另一项技术是为每个新建立的连接

创建一个线程(叫做轻量进程)。为了避免那些与网络无关的进程控制函数把我们的例子搞复杂，我们只给出了反复循环的服务器。参考书 [Stevens 1992]的第4章讨论比较了循环服务器和并发服务器。

还有第三个选择是采用预分支服务器。即服务器启动时连续调用 fork函数数次，并让每个子进程都在同一个监听插口描述符上调用 accept函数。这种办法节省了为每个客户的连接请求临时创建子进程的时间开销，这对于繁忙的服务器来说，是很大的节省。有些HTTP服务器就采用了这项技术。

图1-8给出了TCP上客户 - 服务器事务的时间系列。我们首先注意到，与图 1-4中UDP上的

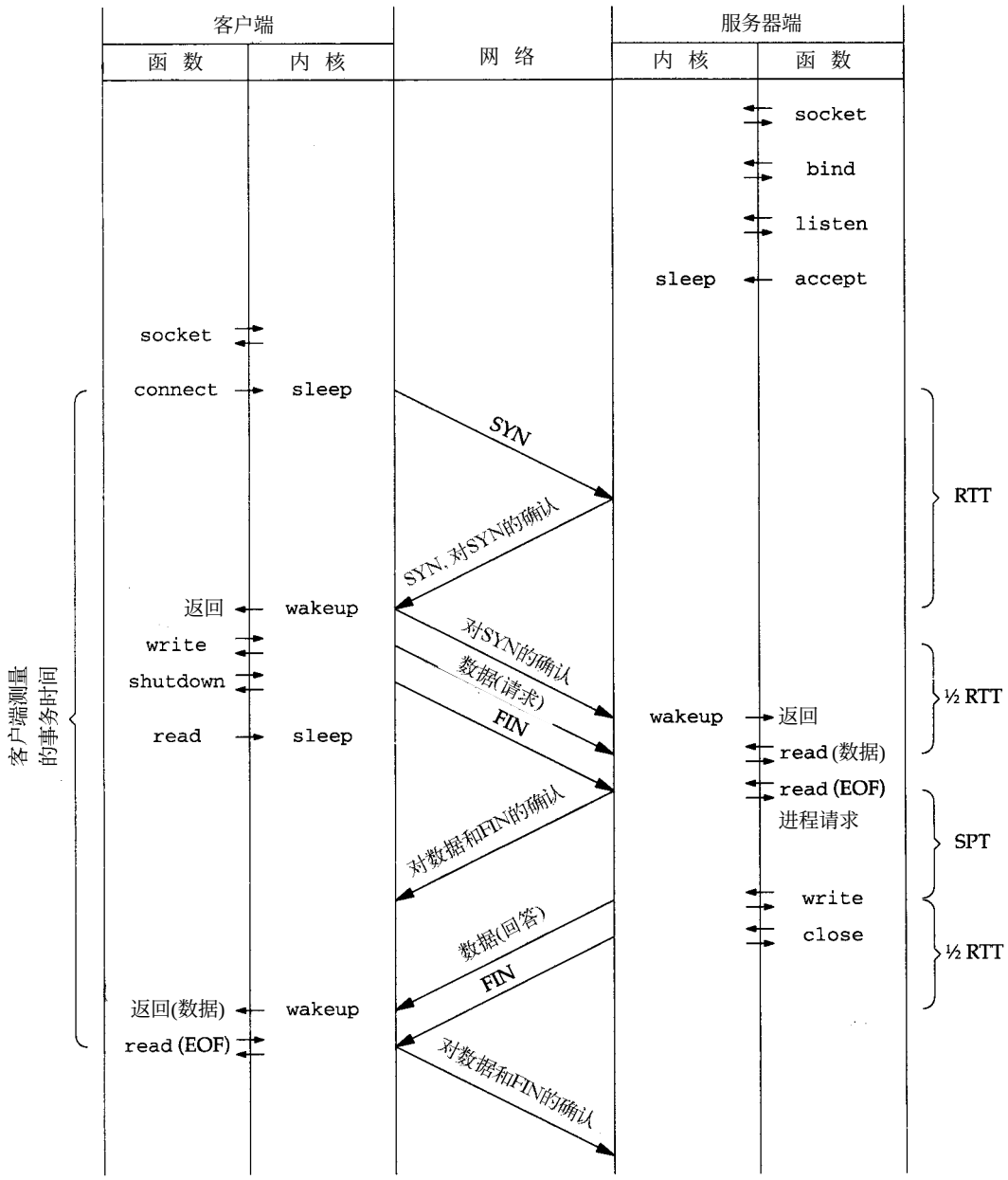


图1-8 TCP上客户 - 服务器事务的时序

事务相比，网络上交换的分组数增加了： TCP上事务的分组数是9，而UDP上的则是2。采用TCP后，客户端测量的事务时间是不少于2 × RTT + SPT。通常，中间三个从客户到服务器的报文段(对服务器SYN的ACK、请求以及客户的FIN)是紧密相连的；后面两个从服务器到客户的报文段(服务器的应答和FIN)也是紧密相连的。这使实际事务时间比从图1-8中看到的更接近2×RTT + SPT。

本例中多出来的一个RTT源于TCP连接建立的时间开销：图1-8中前两个报文段所花的时间。如果TCP可以把建连和发送客户数据以及客户 FIN(图中客户端发出的前四个报文段)合起来，再把服务器的应答和FIN合起来，事务时间就又可以回到RTT + SPT了，这与UDP的一样。事实上，这就是T/TCP中采用的基本技巧。

6. TCP的TIME_WAIT状态

TCP要求，首先发出FIN的一端(我们的例子中是客户)，在通信双方都完全关闭连接之后，仍然要保持在TIME_WAIT状态直至两倍的报文段最大生存时间 (MSL)。MSL的建议值是120秒，也即处于TIME_WATE状态要达到4分钟。当连接处于TIME_WAIT状态时，同一连接(即客户IP地址和端口号，以及服务器IP地址和端口号这4个值相同)不能重复打开(我们在第4章中还要更多地讨论TIME_WAIT状态)。

> 许多基于伯克利代码的TCP实现，在TIME_WAIT状态的保持时间仅仅为60秒，而不是RFC 1122 [Braden 1989]中指定的240秒。在本书的所有计算中，我们还是假定正确的等待周期为240秒。

在我们的例子中，客户端首先发出 FIN，这称为主动关闭，因而 TIME_WAIT状态出现在客户端。在这个状态延续期内， TCP要为这个已经关闭的连接保留一定的状态信息，以便能正确处理那些在网络中延迟一段时间、在连接关闭之后到达的报文段。同样，如果最后一个ACK丢失了，服务器将重传FIN，使客户端重传最后的ACK。

其他一些应用程序，特别是WWW中的HTTP，要求客户程序发送一个专门的命令来指示已经将请求发送完毕(而不是像我们的客户程序那样采用半关闭连接的办法)；接着服务器就发回应答，紧接着就是服务器的 FIN。然后客户程序再发出FIN。这样做与前面所述的不同之处在于，现在的TIME_WAIT状态出现在服务器端而不是客户端。对许多客户访问的繁忙服务器来说，需要保留的状态信息会占用服务器的大量内存。因此，当设计一个事务性客户服务器应用程序时，让连接的哪一端关闭后进入 TIME_WAIT状态值得仔细斟酌。我们还将看到，T/TCP可以让TIME_WAIT状态的延续时间从240秒减少到大约12秒。

7. 减少TCP中的报文段数

像图1-9所示的那样，把数据和控制报文段合并起来可以减少图1-8中所示的TCP报文段数。请注意，这里的第一个报文段中包含有 SYN、数据和FIN，而不像图1-8中那样仅仅是SYN。类似地，服务器的应答和服务器的 FIN也可以合并。虽然这样的分组序列也符合 TCP的规定，但是作者无法在应用程序中利用现有的插口 API使TCP产生这样的报文段序列(因此才在图1-9中客户端产生第一个报文段时和服务器端产生最后一个报文段时标上了问号)；而且据作者所知，也没有哪一个应用程序确实生成了这样的报文段序列。

值得一提的是，尽管我们把报文段的数目由9减少到了5，但客户端观测的事务依然是2×RTT + SPT。这是因为TCP中规定，服务器端的TCP在三次握手结束之前不能向服务器进程提交数据(卷2的第27.9节说明了TCP是如何在连接建立之前将到达的数据进行排队缓存的)。加

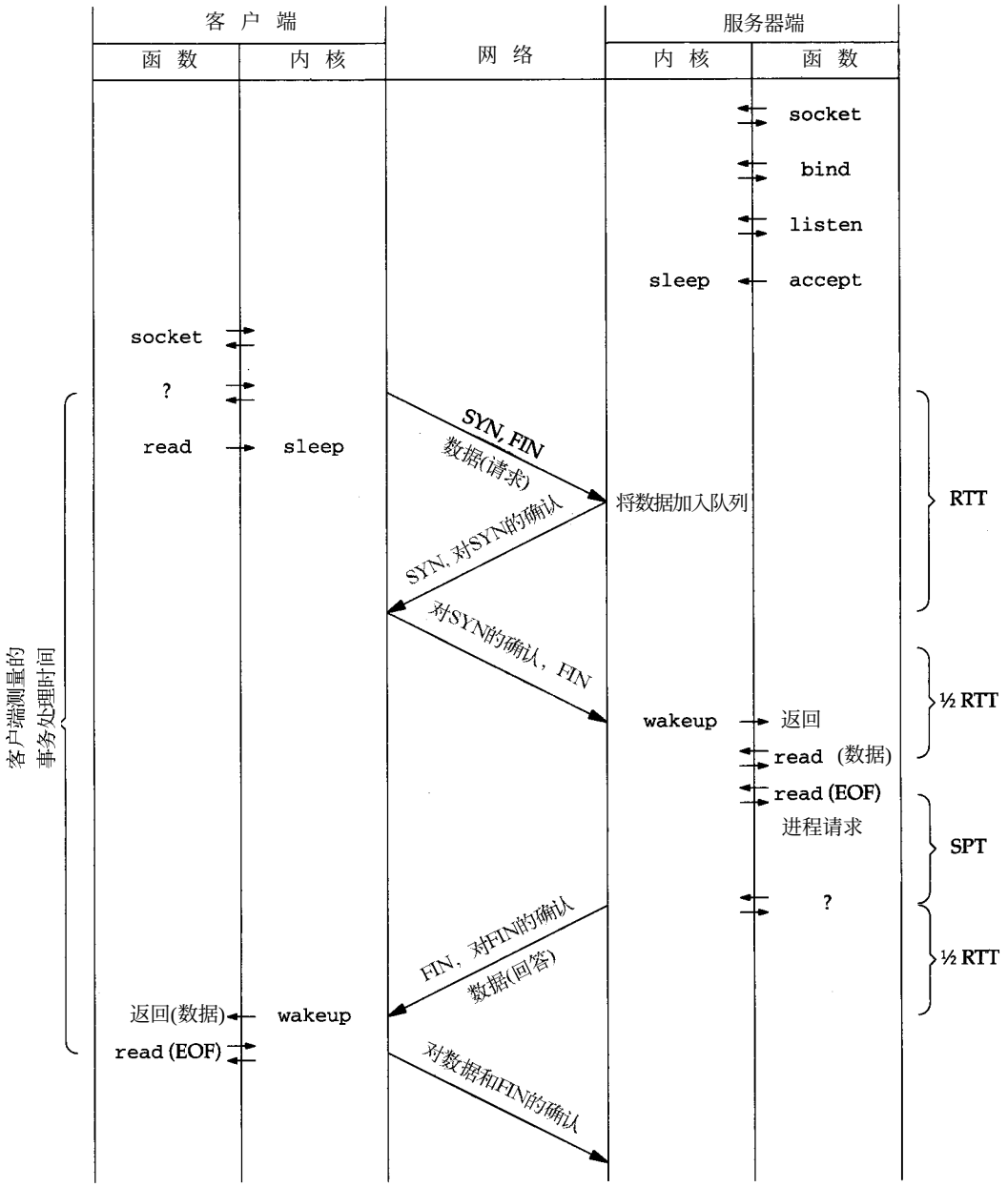


图1-9 最少TCP事务的时序

上这种限制的原因是服务器必须确信来自客户的 SYN是“ 新的 ”，即不是以前某次连接的SYN在网络中延迟一段时间后到达服务器端的。确认过程是这样的：服务器对客户发送的 SYN发送确认，再发出自己的 SYN，然后等待客户对该 SYN的确认。当三次握手完成之后，通信双方就都知道对方的 SYN是新的。由于在三次握手结束之前服务器无法开始处理客户的请求，故分组数的减少并没有缩短客户端测得的事务时间。

下面这段话引自 RFC 1185 [Jacobson, Braden, and Zhang 1990]的附录：“ 注意：使连接能够尽快重复利用是早期 TCP开发的重要目标。之所以有这样的要求是因为当

时人们希望TCP既是应用层事务协议的基础，同时也是面向连接协议的基础。当时讨论中甚至把既包含有SYN和FIN比特，同时又包含数据的报文段叫做‘圣诞树’报文段和‘Kamikaze(敢死队)’报文段。但这种热情很快被泼了冷水，因为人们发现，三次SYN握手和FIN握手意味着一次数据交换至少需要5个分组。而且，TIME_WAIT状态的延续说明同一个连接不可能马上再次打开。于是，再没有人在这个领域做进一步的研究，尽管现在的某些应用程序(比如，简单邮件传送协议，SMTP)经常会产生很短的会话。人们一般都可以采用为每个连接选用不同的端口对的办法来避开重用问题”。

RFC 1379 [Braden 1992b]中写到：“这些‘Kamikaze(敢死队)’报文段不是作为一种支持的服务来提供，而主要用来搞垮其他实验性的TCP！”

作为一个实验，作者编写了一个测试程序，这个程序把SYN与数据和FIN在一个报文段中发出去，即图1-9中的第一个报文段。该报文段发给8个不同版本Unix的标准echo服务器(卷1的第1.12节)，再用Tcpdump观察所交换的数据。其中的7个(4.4BSD、AIX 3.2.2、BSD/OS 2.0、HP-UX 9.01、IRIX System V.3、SunOS 4.1.3和System V Release 4.0)都能正确处理该报文段，另外一个(Solaris 2.4)则把随SYN一起传送的数据扔掉，迫使客户程序重传数据。

那7个系统中的报文段序列与图1-9所描绘的不尽相同。当三次握手结束后，服务器立刻就对客户的数据和FIN发出确认。另外，由于echo服务器无法把数据和FIN捆绑在一起(图1-9中的第四个报文段)发送，结果是发了两个报文段而不只是一个：应答和紧接其后的FIN。因此，报文段的总数是7而不是图1-9中所示的5。我们在3.7节中会进一步讨论与非T/TCP实现的兼容性问题，并给出一些Tcpdump的输出结果。

许多从伯克利演变而来的系统中，服务器无法处理接收到的报文段中只有SYN、FIN，而没有数据、ACK的情况。这个bug使得新创建的插口保持在CLOSE_WAIT状态直到主机重新启动。但这却是一个合法的T/TCP报文段：客户建立起了一个连接，没有发送任何数据，然后就关闭连接。

1.4 T/TCP上的客户-服务器

我们的T/TCP客户-服务器的源代码和上一节的TCP客户-服务器的源代码略有不同，以便能够利用T/TCP的优势。图1-10给出了T/TCP上的客户程序。

ttcpcli.c

```
 1 #include    "cliserv.h"

 2 int
 3 main(int argc, char *argv[])
 4 {                                /* T/TCP client */
 5     struct sockaddr_in serv;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n;

 8     if (argc != 2)
 9         err_quit("usage: ttcpcli <IP address of server>");

10     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");
```

图1-10 T/TCP上的事务客户程序

```
12      memset(&serv, 0, sizeof(serv));
13      serv.sin_family = AF_INET;
14      serv.sin_addr.s_addr = inet_addr(argv[1]);
15      serv.sin_port = htons(TCP_SERV_PORT);

16      /* form request[] ... */

17      if (sendto(sockfd, request, REQUEST, MSG_EOF,
18                 (SA) &serv, sizeof(serv)) != REQUEST)
19          err_sys("sendto error");

20      if ((n = read_stream(sockfd, reply, REPLY)) < 0)
21          err_sys("read error");

22      /* process "n" bytes of reply[] ... */

23      exit(0);
24  }
```

ttcpcli.c

图1-10 (续)

1. 创建TCP插口

10-15 对socket函数的调用与TCP上的客户程序一样，在Internet插口地址结构中同样也填入服务器的IP地址和端口号。

2. 向服务器发送请求

17-19 T/TCP上的客户程序不调用connect函数。而是直接调用标准的sendto函数，该函数向服务器发送请求，同时与服务器建立起连接。此外，我们还用 sendto函数的第4个参数指定了一个新的标志MSG_EOF，用以告诉系统内核数据已经发送完毕。这样做就相当于图 1-5中调用shutdown函数，向服务器发送一个FIN。MSG_EOF标志是T/TCP实现中新加入的，不要把它与MSG_EOR标志混淆，后者是基于记录的协议(比如OSI的运输层协议)中用来标志记录结束的。我们将在图1-12中看到，调用sendto函数的结果是客户端的SYN、客户的请求以及FIN都包含在一个报文段中发送出去。换言之，调用一个 sendto函数就实现了connect、write和shutdown三个函数的功能。

3. 读服务器的应答

20-21 读服务器的应答还是用 read_stream函数，与前文讨论过的TCP上的客户程序一样。

图1-11所示的是T/TCP上的服务器程序。

ttcpserv.c

```
1 #include    "cliserv.h"

2 int
3 main()
4 {                                   /* T/TCP server */
5     struct sockaddr_in serv, cli;
6     char    request[REQUEST], reply[REPLY];
7     int     listenfd, sockfd, n, clilen;

8     if ((listenfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
9         err_sys("socket error");
```

图1-11 T/TCP上的事务服务器程序

```
10      memset(&serv, 0, sizeof(serv));
11      serv.sin_family = AF_INET;
12      serv.sin_addr.s_addr = htonl(INADDR_ANY);
13      serv.sin_port = htons(TCP_SERV_PORT);

14      if (bind(listenfd, (SA) &serv, sizeof(serv)) < 0)
15          err_sys("bind error");

16      if (listen(listenfd, SOMAXCONN) < 0)
17          err_sys("listen error");

18      for (;;) {
19          clilen = sizeof(cli);
20          if ((sockfd = accept(listenfd, (SA) &cli, &clilen)) < 0)
21              err_sys("accept error");

22          if ((n = read_stream(sockfd, request, REQUEST)) < 0)
23              err_sys("read error");

24          /* process "n" bytes of request[] and create reply[] ... */

25          if (send(sockfd, reply, REPLY, MSG_EOF) != REPLY)
26              err_sys("send error");

27          close(sockfd);
28      }
29  }
```

ttcpserv.c

图1-11 (续)

这个程序与图1-7中TCP上的服务器程序几乎完全一样：对 socket函数、bind函数、listen函数、accept函数和read_stream函数的调用都一模一样。唯一的不同在于T/TCP上的服务器发送应答时调用的是 send函数，而不是write函数。这样就可以设置MSG_EOF标志，从而可以将服务器的应答和服务器的FIN合并在一起发送。

图1-12所示的是T/TCP上客户-服务器事务的时序图。

T/TCP上的客户测量到的事务时间和UDP上的几乎一样(图1-4)：RTT + SPT。我们估计T/TCP上的时间会比UDP上的时间稍长一点，这是因为TCP协议需要处理的事情比UDP要多一些，而且通信双方都要执行两次 read操作分别读数据和文件结束标志(而UDP环境下双方都只要调用一次recvfrom函数即可)。但是双方主机上这一段额外的处理时间比一次网络往返时间RTT要小得多(我们在1.6节中给出了一些测试数据，用来比较UDP、TCP和T/TCP上的客户-服务器事务的差别)。由此我们可以得出结论：T/TCP上的事务时间要比TCP上的事务小大约一次网络往返时间 RTT。T/TCP中省下来的这个RTT来自于TAO，即TCP加速打开(TCP Accelerated Open)。这种方式跳过了三次握手的过程。下面两章中我们将说明其实现方法；在4.5节中我们还将证明这样做的正确性。

UDP上的事务需要两个分组来传送，T/TCP上的事务需要3个分组，而TCP上的事务则需要9个分组(这些数字的前提是没有分组丢失)。因此，T/TCP不仅缩短了客户端的事务处理时间，而且也减少了网络上传送的分组数。我们希望减少网络上的分组数，因为路由器往往受限于它们可以转发的分组数，而不是每个分组的长度。

概括地讲，T/TCP以一个额外的分组和可以忽略的延续时间为代价，同时具有了可靠性和适应性这两个对网络应用至关重要的特性。

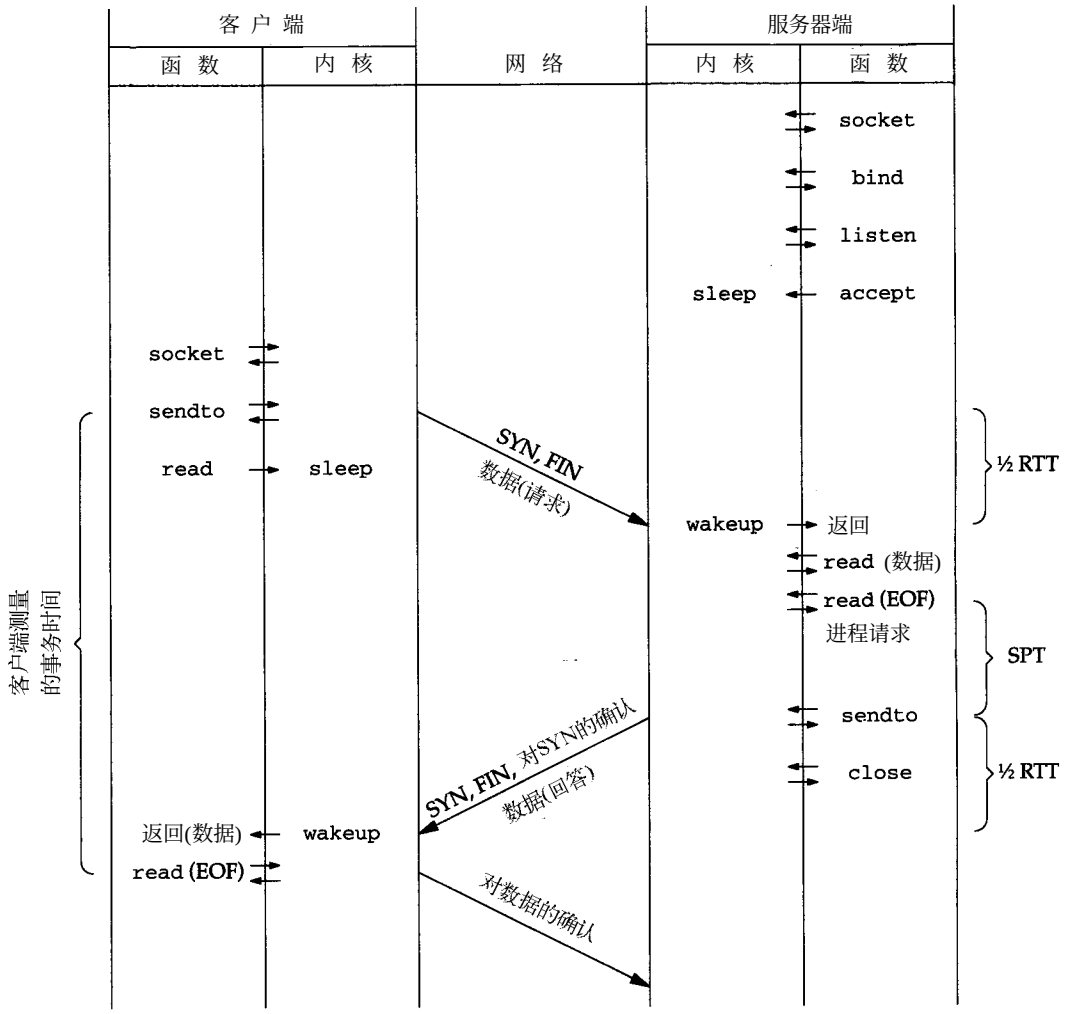


图1-12 T/TCP上客户-服务器事务的时序

1.5 测试网络

图1-13画出了用于验证本书所有例子的测试网络。

书中大多数的示例程序都运行在laptop和bsdi这两个系统上，它们都支持T/TCP协议。图1-13中所有的IP地址都属于B类子网140.252.0.0。所有主机的名字都属于tuc.noao.edu域。noao表示“国家光学空间观测站”，tuc表示Tucson。图中每个方框上部的记号表示在该系统运行的操作系统。

1.6 时间测量程序

我们可以分别测量三种客户-服务器事务的时间，并比较其测量结果。我们要对客户程序作如下改动：

• 在图1-1所示的UDP上客户程序中，我们在即将调用sendto函数前和recvfrom函数刚

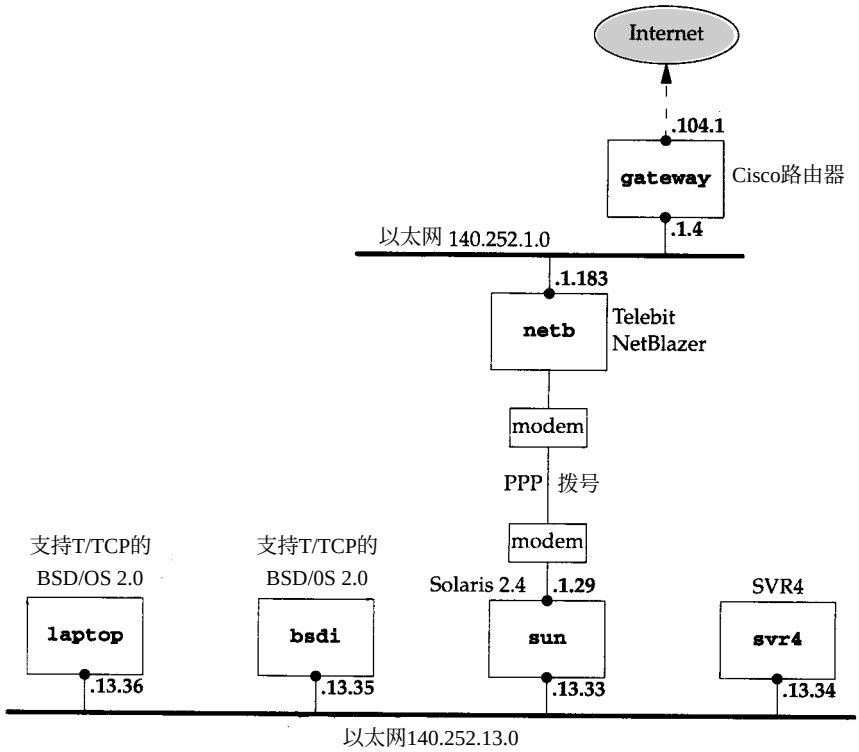


图1-13用于验证本书所有例子的测试网络，所有 IP地址都以140.252打头

刚返回后分别读取当前系统时间。这两个时间的差值即为客户端测得的事务时间。

- 在图1-5所示的TCP上客户程序中，我们在即将调用 `connect`函数前和`read_stream`函数刚刚返回后分别读取当前系统时间。
- 在图1-10所示的T/TCP上客户程序中，我们取当前的系统时间为即将调用 `sendto`函数前和`read_stream`函数刚刚返回后。

图1-14给出了以14种不同长度的请求和应答分别测得的结果。客户和服务器分别为图 1-13中的`bsdi`和`laptop`。附录A中给出了这些测量的细节，并分析了影响结果的因素。

T/TCP上的事务时间总是比同样条件下的 UDP上的事务时间要长几个毫秒 (由于这个时间差是软件造成的，因此这个时间差会随着计算机速度的提高而缩短)。T/TCP协议栈比UDP协议栈所做的操作要多 (图A-8)，而且T/TCP上的客户和服务器要分别调用两次 `read`函数，而UDP上的客户和服务器则只需分别调用一次 `recvfrom`函数。

TCP上的事务时间总是比相同条件下 T/TCP上的事务要长大约 20 ms。其中部分原因是由于TCP建立连接时的三次握手。两个 SYN报文段的长度是 44字节(20字节的IP首部、20字节的标准TCP首部和4字节的TCP MSS选项)。这相当于用户数据为 16字节的Ping；从图A-3可知，其网络往返时间 RTT大约为 10 ms。另外10 ms的差值可能是因为 TCP协议需要处理额外 6个TCP报文段造成的。

因此我们可以得出结论：T/TCP上的事务时间接近、但比 UDP上的事务时间略大，比TCP上的事务时间短至少相当于一个 44字节报文段的网络往返时间。

就客户段测量的事务时间而言，用 T/TCP取代TCP带来的好处依赖于 RTT和SPT之间的关

系。比如，在一个局域网上的RTT为3 ms(如图A-2)，服务器的平均处理时间为500 ms，那么TCP上的事务时间大约为506 ms(2×RTT+SPT)，而T/TCP的事务时间则大约为503 ms。但如果是一个网络往返时间RTT为200ms的广域网(见第14.4节)，服务器处理时间SPT的平均值为100 ms，那么TCP上和T/TCP上的事务时间就分别为大约500 ms和300 ms。我们已经看到，使用T/TCP所需传送的网络分组数少(从图1-8和图1-12的比较中看分别是3个和9个)，因此，不管客户端所测得的事务时间减少了多少，使用T/TCP总是能减少网络分组数。减少了网络分组数就可以减少分组丢失的概率，而在Internet中，分组丢失对整个网络的稳定性有很大影响。

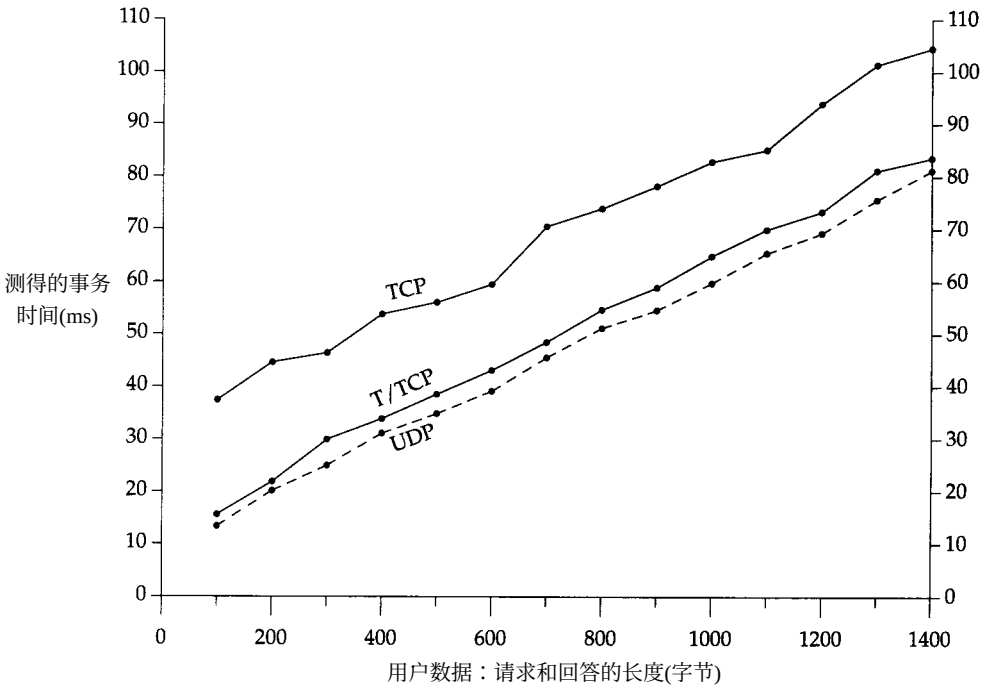


图1-14 UDP、T/TCP和TCP上客户-服务器事务的时间系列

在A.3节里，我们介绍了传播时迟和带宽的差异。这两者对RTT都有影响；但是当网络变快以后，传播时迟的影响也就变大了。此外，传播时延是我们几乎无法控制的，因为它的大小取决于客户和服务器之间的信号传播距离和光在介质中的传播速度。于是，在网络速率越来越快的条件下，省下一个RTT的时间就显得尤为可贵，使用T/TCP的相对好处也就越发明显。

现在可以公开获得并支持T/TCP的用于测量网络性能的工具：

```
http://www.cup.hp.com/netperf/netperfpage.html
```

1.7 应用

T/TCP给所有TCP上的应用程序带来的第一个好处就是可以缩短TIME_WAIT状态的持续时间。这样，一般情况下协议必须处理的控制块也跟着少了。4.4节详细介绍了T/TCP协议的这个特性。现在我们可以这样说：对于连接时间很短(典型值为小于2分钟)的所有TCP应用程

序，如果通信双方的主机都支持T/TCP的话，它们都将因使用该协议而获益。

使用T/TCP的最大好处或许在于避免了三次握手过程，对于那些交换的数据量比较小的应用程序，T/TCP减少的时延将给它们带来好处。我们将给出几个例子来说明这一点（附录B谈到了利用T/TCP来避免三次握手过程要对应用程序做怎样的修改）。

1. WWW：超文本传输协议

WWW及其所依赖的HTTP协议(将在第13章介绍该协议)将可能大大地受益于T/TCP协议。参考书[Mogul 1995b]中指出：“然而，构成Web应用传输时延的主要因素是网络通信……即便我们无法提高光的传播速度，但我们至少应该想办法减少一次交互过程中的往返传输次数。当前Web网中使用的超文本传输协议(HTTP)实际上造成了大量不必要的往返传输”。

比如，[Mogul 1995b]中对随机抽取的200 000个HTTP请求的统计发现，应答长度的中值为1770字节(通常使用中值而不使用均值，这是因为很少出现的大文件会使均值变大)。Mogul还引用了另一个例子。该例随机抽样了大约150万个请求，其应答的长度中值为958字节。客户的请求一般很短：在100~300字节之间。

典型的HTTP客户-服务器事务和图1-8所示的很相似。客户端主动打开，向服务器发出很短的请求，服务器收到请求后发出应答，然后服务器关闭连接。这种情况非常适于使用T/TCP协议，把客户端的SYN和客户的请求合并在一起传送省去三次握手中的往返时间。这也还减少了网络上的分组数，而这对于已经非常巨大的Web通信量来说也是很有意义的。

2. FTP数据连接

FTP数据连接也会从使用T/TCP协议中获益。从一项对Internet通信量的统计调查中，[Paxson 1994b]发现平均每个FTP数据连接所传输的数据量约为3 000字节。卷1的第323页给出了FTP数据连接的一个例子。虽然例子中的数据流是单向的，但其传输过程还是与图1-12所示的十分相似。采用T/TCP后，图中的8个报文段减少到了3个。

3. 域名服务系统(DNS)

DNS客户的查询请求是用UDP传送到DNS服务器的。服务器仍然用UDP发送给客户的应答。但如果应答超过512字节，那么只有前512字节会在应答中返回给客户，同时在应答中有“truncated(截断)”标志，表示还有信息要传给客户。于是客户用TCP向服务器重新发送查询请求，而后服务器用TCP向客户传送完整的应答。

采用这项技术的原因是不能保证特定的主机能够重组长度超过576字节的IP数据报(实际上，许多UDP应用程序都把用户数据的长度限定在512字节以内，以保证不超过576字节的限制)。由于TCP是一个字节流协议，应答数据量再大也不会有问题。发送方TCP会根据连接建立时对等端声明的报文段最大长度(MSS)限制，把应用程序的应答数据分割成适当长度的报文段发给对方。接收方TCP会把这些报文段拼接起来，并以应用程序读取时指定的数据长度交给接收的应用程序。

DNS的客户和服务器可以利用T/TCP，既达到UDP的请求-应答速度，又具有TCP的所有好处。

4. 远程过程调用(RPC)

在所有论述将传输协议用于事务的论文中，无不将RPC作为一个候选的应用协议。RPC中客户要向载有待执行程序的服务器发送请求，请求中带有客户给定的参数；服务器的应答中包括过程执行后所返回的结果。参考书[Stevens 1994]的第29.2节中讨论了Sun RPC。

RPC的数据包往往会非常大，必须给 RPC协议增加可靠性，使其能在像 UDP这样不保证可靠性的协议上运行，同时还要避免 TCP的三次握手。使用T/TCP协议就能实现这一目标，既有TCP的可靠性，又没有三次握手的开销。

所有建立在RPC基础上的应用程序，比如网络文件系统 (NFS)等都可以采用T/TCP协议。

1.8 历史

RFC 938 [Miller 1985]是较早讲述事务的 RFC文档之一。该文档中规定了 IRTP，即：Internet 可靠的事务协议，能保证数据分组的可靠、按顺序提交。该文档中把事务定义为一个短小的、自包含的报文；而 IRTF定义了任意两台主机 (即IP地址)之间持续存在的优选连接，当其中任何一台主机重新启动后，该连接都重新同步。 IRTF协议位于IP协议之上，并定义了专门的8字节首部。

RFC 955 [Braden 1985]本质上并未规定任何协议，而只是给出了事务协议的一些设计准则。它认为UDP和TCP这两个主流的运输层协议所提供的业务相差太大，而事务协议正好填补TCP和UDP之间的空档。该RFC文档把事务定义为一次简单的报文交换：一个请求发给服务器，然后一个应答发回到客户。它还认为各种事务都有如下特征：不对称的模式 (一端是服务器，另一端是客户)、单工数据传递(任一时刻都只有一个方向有数据传输)、持续时间短(可能延续几十秒，但绝不可能几小时)、时延小、数据分组少以及面向报文 (不是字节流)。

该RFC中列举了域名服务系统 DNS的例子。它认为，在考虑是用 UDP还是用TCP作为域名服务系统的运输层协议时，设计者往往陷入两难的境地。一个理想的解决方案应该既能提供可靠的数据传输，又不需要专门地建立和释放连接，不需要报文的分段和重组 (从而应用程序不再需要知道像576这类的神秘数字)，同时还能使两端的空闲状态所处时间最短。 TCP什么都好，只可惜它需要建立和释放连接。

另一个相关的协议是RDP，即可靠数据协议。该协议在RFC 908 [Velten，inden，and Sax 1984]中定义，后来又更新为RFC 1151 [Patridge and Hinden 1990]。与RDP实现有关的经验在参考文献[Patridge 1987] 中可以找到。参考文献[Patridge 1990a]中对RDP有如下评价：“ 当人们寻求一个可靠的数据报协议时，他们通常是想要一个事务协议，一个能够让他们与多个远端系统可靠地交换数据单元的协议，一个类似于可靠 UDP的协议。RDP应该看作是一个面向记录的TCP协议，它利用连接可靠地传输有格式数据块流。 RDP并不是一个事务协议。” (RDP不是一个事务协议的理由是因为它和 TCP一样采用了三次握手技术)。

RDP使用通常的插口应用编程接口 (API)。与TCP类似， RDP提供流插口接口 (SOCK_STREAM)。另外， RDP还提供SOCK_RDM插口类型(可靠的报文提交)和SOCK_SEQPACKET插口类型(有序的分组)。

VMTP，即通用报文事务协议，是在 RFC 1045 [Cheriton 1998]中规定的，是一个专门用于事务的协议，就像远程过程调用一样。像 IRTP和RDP那样， VMTP也是IP之上的运输层协议，但VMTP还支持多播通信，这个特性是 T/TCP以及本节提到过的其他协议所不具备的 (参考文献[Floyd et al. 1995]中有不同意见，他们认为提供可靠的多播通信是应用层的任务，而不是运输层的任务)。

VMTP还为应用程序提供不一样的应用编程接口 API，其插口类型为SOCK_TRANSACT。

具体定义详见RFC 1045。

虽然T/TCP的许多概念早在RFC 955中就已经出现，但直到RFC 1379 [Braden 1992b]发布才正式有了T/TCP的第一个规范。该RFC文档定义了T/TCP的概念，接下来的RFC 1644 [Braden 1994]给出了更多的细节，并讨论了一些实现问题。

图1-15比较了实现各种运输层协议分别都需要多少行C源代码。

协　议	源代码行数
UDP(卷2)	800
RDP	2 700
TCP(卷2)	4 500
T/TCP模式的TCP	5 700
VMTP	21 000

图1-15　实现各种运输层协议所需要的源代码行数

为支持T/TCP所需增加的源代码行数(大约1200行)是UDP协议源代码行数的1.5倍。为使4.4BSD支持多播通信，需要增加大约2000行源代码(设备驱动程序的改变和支持多播路由所需要的代码行数尚未计算在内)。

VMTP可以从`ftp://gregorio.stanford.edu/vmtp-ip`得到。RDP通常还得不到。

1.9 实现

第一个T/TCP实现是由Bob Braden和Liming Wei在南加州大学的信息科学学院(USC ISI)完成的。该项工作得到了国家科学基金NSF的部分资助，批准号为NCR-8 922 231。该实现是为SunOS 4.1.3(从伯克利演变而来的内核)做的，1994年9月就可以用匿名的FTP得到了。SunOS 4.1.3的源代码补丁可以从`ftp://ftp.isi.edu/pub/braden/TTCP.tar.Z`得到，但你必须有SunOS内核的源代码才能应用这些补丁。

Twente大学(荷兰)的Andras Olah修改了USC ISI的实现，并于1995年3月将其在FreeBSD 2.0版中发布。FreeBSD 2.0中的网络代码是基于4.4BSD-Lite版的(卷2中有介绍)。图1-16给出了各种BSD版本的演变历程。与路由表(我们将在第6章中讨论)有关的所有工作都是由麻省理工学院(Massachusetts Institute of Technology)的Garrett Wollman完成的。FreeBSD实现的有关信息可以从`http://www.freebsd.org`得到。

本书作者把FreeBSD实现移植到了BSD/OS 2.0内核(该内核也基于4.4BSD-Lite中的网络代码)中，也就是运行在主机`bsdi`和`laptop`(图1-13中)中的代码，本书从头至尾都用它们。为了支持T/TCP而对BSD/OS所做的修改,可以从作者的个人主页里找到：`http://www.noao.edu/~rstevens`。

图1-16给出了各个BSD版本的演变历程，其中还标出了重要的TCP/IP特性。图中左边显示的是可以公开得到源代码的版本，其中有所有网络代码：协议本身、网络接口的内核例程以及许多应用程序和实用工具(比如Telnet和FTP)。

本书中所描述的T/TCP实现的基础软件的正式名称是4.4BSD-Lite，但我们一般简称其为Net/3。还要说明的是，可以公开得到的Net/3版本中不包括本书所述为支持T/TCP而做的修改。

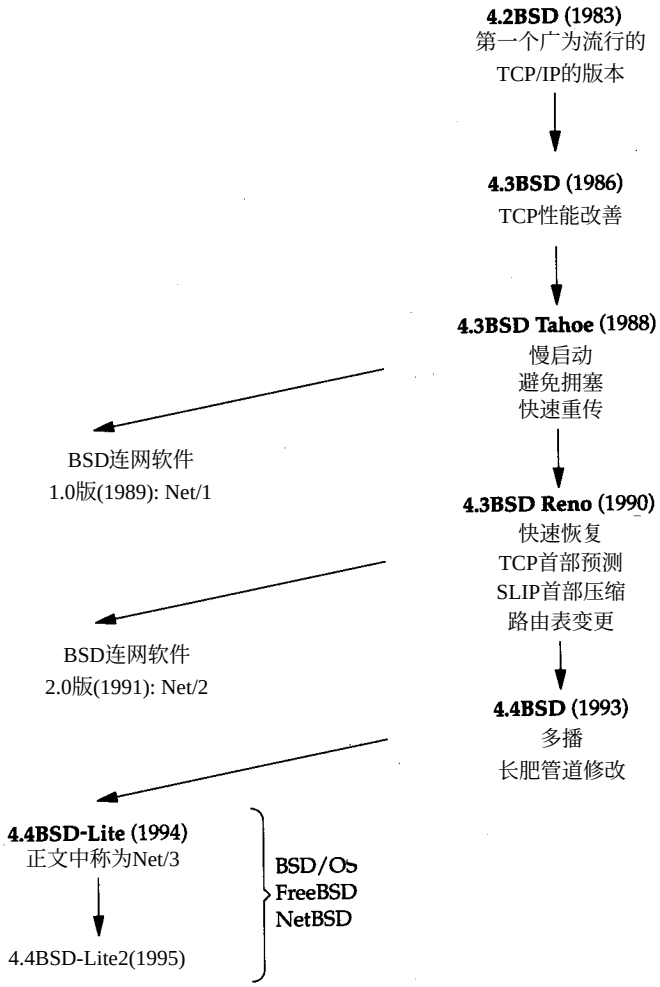


图1-16 带有重要TCP/IP特性的各种BSD发行版

当提到Net/3这个术语时，实际所指的就是这个不包含 T/TCP的、可公开得到的版本。

4.4BSD-Lite2是1995年对4.4BSD-Lite的升级。从网络部分来看，从 Lite到Lite2仅仅是解决了一些bug，以及少量的改进(比如我们将在14.9节中介绍的坚持探测的超时)。我们给出了3个基于Lite代码的系统：BSD/OS、FreeBSD和NetBSD。本书所述全部都是基于 Lite代码的，但所有以上的3个版本都应该在下一个主要版本中升级到 Lite2。可以从下面的 Walnut Creek CDROM站点得到含有Lite2版本的光盘：`http://www.cdrom.com`。

本书全书都将用“从伯克利演变而来的实现”这个术语指称厂商的实现，比如 SunOS、SVR4(System V Release 4)和AIX，因为所有这些实现的TCP/IP代码最初都来自于伯克利源代码，它们之间有许多共同点，甚至连程序中的差错都相同！

1.10 小结

本章的目的是要让读者相信 T/TCP的确为许多实际中的网络应用问题提供了一个解决方案。我们从比较一个分别用 UDP、TCP和T/TCP编写的、简单的客户-服务器程序开始。用

UDP协议需要交换两个分组，用TCP需要9个，而用T/TCP需要3个。我们还发现，用T/TCP和用UDP时在客户端测得的事务时间相差无几。图1-14所示的时间测量结果证明了我们的结论。除了达到UDP的性能之外，T/TCP还具有可靠性和适应性，这两点都是对UDP的重大改进。

T/TCP因为避免了常规TCP中的三次握手而获得上述各种优点。为了利用这些优点，客户和服务器程序在应用T/TCP时必须对源代码做一些简单的改动，主要是在客户端用`sendto`函数代替`connect`、`write`和`shutdown`三个函数。

在后面的3章中，我们将研究协议是如何工作的，同时还会研究更多的T/TCP应用例子。

第2章 T/TCP协议

2.1 概述

我们分两章(第2章和第4章)讨论T/TCP协议。这样，在深入研究T/TCP协议之前(第3章)，我们可以先看一些应用T/TCP的例子。本章主要对协议应用技巧和实现中用到的变量做一个介绍。下一章我们学习一些T/TCP应用的示例程序。第4章结束我们对T/TCP协议的学习。

在第1章中我们已经看到了，当把TCP协议应用于客户-服务器事务时会存在两个问题：

1) 如图1-8所示，三次握手使客户端测得的事务时间额外多出一个RTT。

2) 由于客户进程主动关闭连接(即由客户进程首先发出FIN)，因而在客户收到服务器的FIN后还要在TIME_WAIT状态滞留大约240秒。

   TIME_WAIT状态和16比特TCP端口号这两者结合起来限制了两台主机之间的最大事务速率。例如，如果同一台客户主机要不断地和同一台服务器主机进行事务通信，那么它要么每完成一次事务后等待240秒才开始下一个事务，要么为紧接着的事务选择另外一个端口号。但每240秒的时间内至多只能有64 512个端口(65 535减去1 023个知名端口)可用，从而每秒最多也就只能处理268个事务。在RTT值大约为1~3ms的局域网上，实际上可能会超过这个速率。

   而且，即使应用程序的事务速率低于每秒240次，比如每240秒只有50 000次。当客户端处于TIME_WAIT状态时，协议还是需要控制块来保存连接的状态。卷2中给出的BSD实现中，每个连接都需要一个IP控制块(84字节)，一个TCP控制块(140字节)和一个TCP/IP首部模板(40字节)。这样总共就需要13 200 000字节的内核存储空间。这个开销即便在内存不断扩大的今天依然显得大了些。

现在，T/TCP协议解决了这两个问题，采用的方法是绕过三次握手，并把TIME_WAIT状态的保持时间由240秒缩短到大约12秒。我们将在第4章中详细研究这两个特点。

T/TCP协议的核心称为TAO，即TCP加速打开，跳过了TCP的三次握手。T/TCP给主机建立的每个连接分配一个唯一的标识符，称为连接计数(CC)。每台T/TCP主机都要将不同主机对之间的最新连接计数CC保持一段时间。当服务器收到来自T/TCP客户的SYN时，如果其中携带的CC大于该主机对最新连接的CC，就保证这是一个新的SYN，于是就接受该连接请求，而不需要三次握手。这个过程称为TAO测试。如果测试失败，TCP还是用三次握手的老方法来确认当前这个SYN是否为新的。

2.2 T/TCP中的新TCP选项

T/TCP协议中有三个新的TCP选项。图2-1给出了目前TCP协议使用的所有选项。其中前3个出自最初的TCP协议规范，即RFC 793 [Postel 1981b]。而窗口宽度和时间戳则是在RFC 1323 [Jacobson, Braden, and Borman 1992]中定义的。最后三个选项(CC、CCnew和CCecho)则是T/TCP协议新引入的，在RFC 1644 [Braden 1994]中定义。最后这几个选项的使用规则如下：

1) CC选项在客户执行主动打开操作时发出的第一个 SYN报文段中使用。它也可以在其他一些报文段中使用，但前提是对方发过来的 SYN报文段中带有CC或CCnew选项。
2) CCnew选项只能在第一个 SYN报文段中使用。当需要执行正常的三次握手操作时，客户端的TCP协议就使用CCnew选项而不用CC选项。
3) CCecho选项仅在三次握手过程中的第二个报文段中使用：通常由服务器发出该报文段，并携带有SYN和ACK。该报文段将CC或CCnew的值返回给客户，告知客户本服务器支持T/TCP协议。

本章以及下一章的例子中我们还会进一步讨论这些选项。

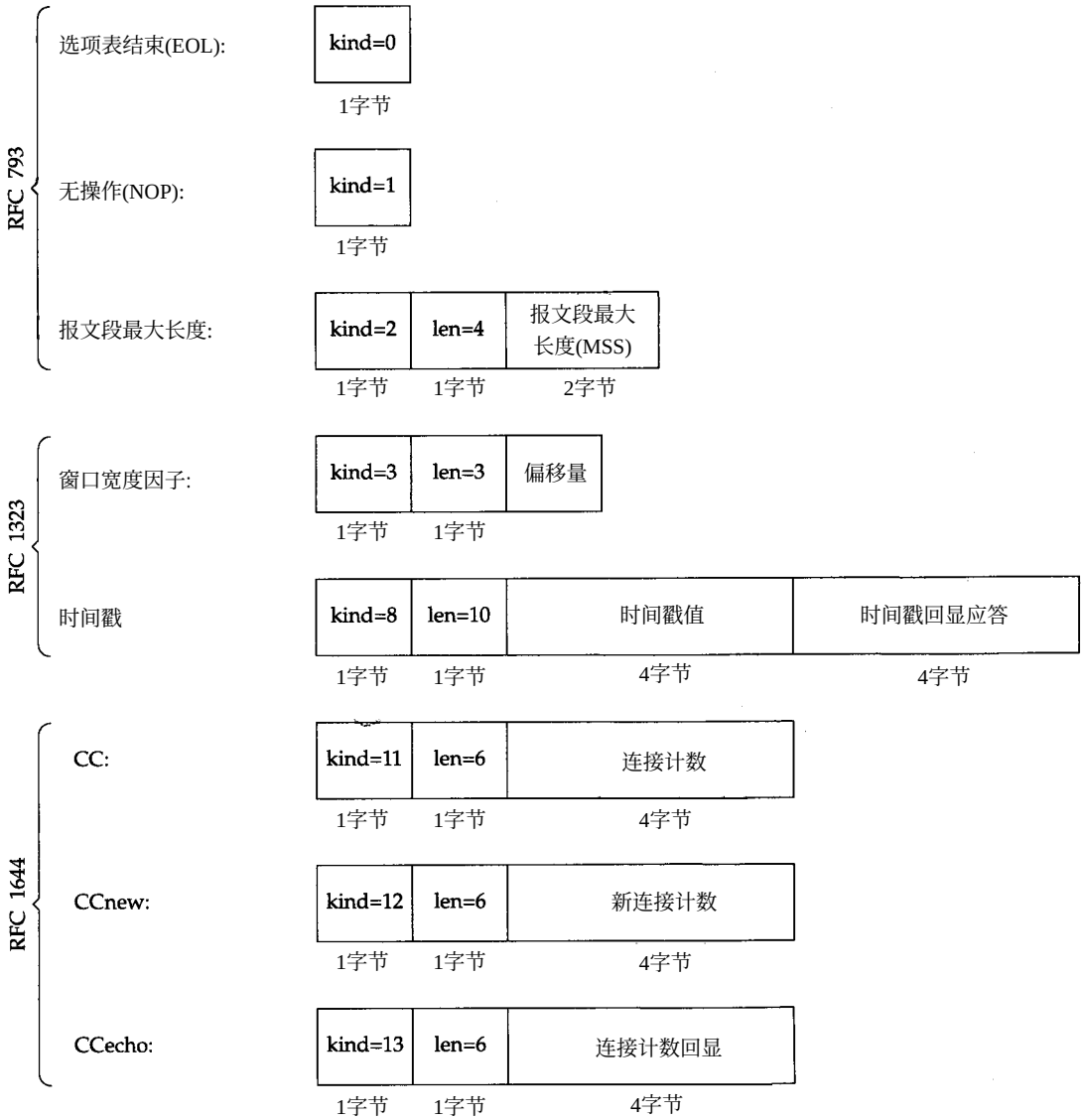


图2-1 TCP选项

不难发现，T/TCP的3个新选项均为6字节长。为了使这些选项继续按4字节定界(这在某些系统体系结构中有助于提高性能)，我们通常在这些选项的前面加上两个单字节的无操作(NOP)。

如果客户既支持RFC 1323，也支持T/TCP协议，这时客户发给服务器的第一个SYN报文段中的TCP选项，如图2-2所示。我们特意给出了每个选项的类型值和长度值；NOP用阴影表示，其类型值为1。第二个选项是窗口宽度，这里用“ WS ”标记。方格上方的数字是每个选项相对于选项字段起始的字节偏移量。TCP协议选项的最大长度为40字节，本例中的TCP选项共需28字节。从图中可以看出，采用NOP填充以后，所有4个4字节的值都符合4字节定界规则。

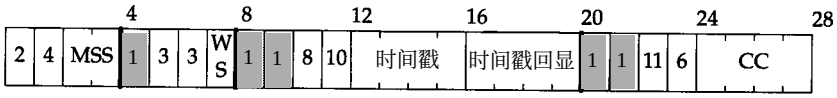


图2-2 同时支持RFC 1323和T/TCP的客户发给服务器的第一个SYN报文段的TCP选项

如果服务器既不支持RFC 1323，也不支持T/TCP协议，它发给客户带有SYN和ACK的应答中就只有报文段最大长度(MSS)选项。但如果服务器既支持RFC 1323，也支持T/TCP协议，那么它给客户的应答中将包含图2-3所示的TCP选项，总长为36字节。

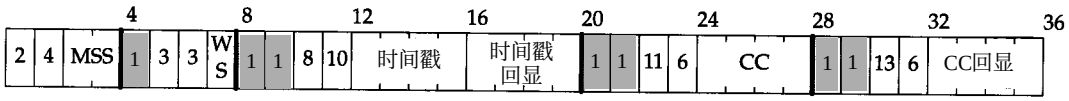


图2-3 服务器对图2-2所示请求的应答中的TCP选项

由于CCecho选项总是和CC选项一起发送，因此T/TCP协议的设计本可以把这两个选项合二为一，从而为宝贵的TCP协议选项空间节省4个字节。或者也可以这样，这种最坏的选项排列只在服务器给出SYN/ACK时出现，而它们的出现无论如何总要使TCP处理速度变慢的，因此索性连NOP字节也省去，实际上可以节省7个字节。

因为报文段的最大长度和窗口宽度选项只在SYN报文段中出现，而CCecho选项只在SYN/ACK报文段中出现，因此，如果连接两端都支持RFC 1323和T/TCP协议，则自此以后的报文段中也都只包含时间戳和CC选项，如图2-4所示。

图2-4 两端都支持RFC 1323和T/TCP时非SYN报文段所包含的TCP选项

可以看出，一旦连接建立，时间戳和连接计数CC选项给所有的TCP报文段都增加了20字节。当讲到T/TCP协议时，我们常常用一般术语CC选项作为本节所引入的3个TCP选项的统称。

时间戳和CC选项带来了多大的额外开销呢？假设两台主机位于两个不同的网络上，报文段最大长度MSS设为典型值512字节。要传递一兆字节的文件，如果没有这些选项，则需要1 954个报文段；如果使用时间戳和CC选项，则需要2 033个报文段，较前者增加了4% 。如果报文段最大长度MSS为1 460字节，那么报文段数只增加了1.5%。

2.3 T/TCP 实现所需变量

T/TCP协议要求内核保存一些新增的信息，本节将对这些信息加以描述，后面几节将讨论

如何使用这些新信息。

(1) tcp_ccgen：这是一个32位的全局整型变量，记录待用的CC值。每当主机建立了一个连接，该变量的值就加1，无论是主动还是被动，也不论是否使用T/TCP协议。该变量永不为0。当变量渐渐增长时，如果又回到了0，那么就将其值置为1。

(2) 每主机高速缓存(per-host cache)，其中包含了三个新变量：tao_cc、tao_ccsent和tao_mssopt。该高速缓存也称为TAO高速缓存。我们将看到，T/TCP协议为每一个与之通信的主机创建一个路由表项，并把这些信息存储在路由表项中(把每主机高速缓存安排在路由表中是很方便的。当然也可以另开一张完全分离的表作为这个每主机高速缓存。T/TCP协议不需要对IP路由功能做任何改动)。在每主机高速缓存中创建一个新表项时，tao_cc和tao_ccsent必须初始化为0，表示它们尚未定义。

tao_cc记录的是最后一次从对应主机接收到且不含ACK的合法的SYN报文段(即主动打开连接)中的CC值。当T/TCP主机收到一个带有CC选项的SYN报文段时，如果CC选项的值大于tao_cc，那么主机就知道这是一个新的SYN报文段，而不是一个重复的老SYN，这样就可以跳过三次握手(TAO测试)。

tao_ccsent记录的是发给相应主机的最后一个不含ACK的SYN报文段(即主动打开连接)中的CC值。如果该值未定义(为0)，那么只有当对方发回一个CCecho选项，表示其可以使用T/TCP协议时，才将tao_ccsent设置为非0。

tao_mssopt是最后一次从相应主机接收到的报文段最大长度选项值。

(3) 现有的TCP控制块中需要增加三个新的变量：cc_send、cc_recv和t_duration。第1个变量记录的是该连接上发送的每一个报文段中的CC值，第2个变量记录的是希望对方发来的报文段中所携带的CC值，最后一个变量则用来记录连接已经建立了多长时间(以系统的时钟滴答计算)。当连接主动关闭时，如果该时间计数器显示的连接持

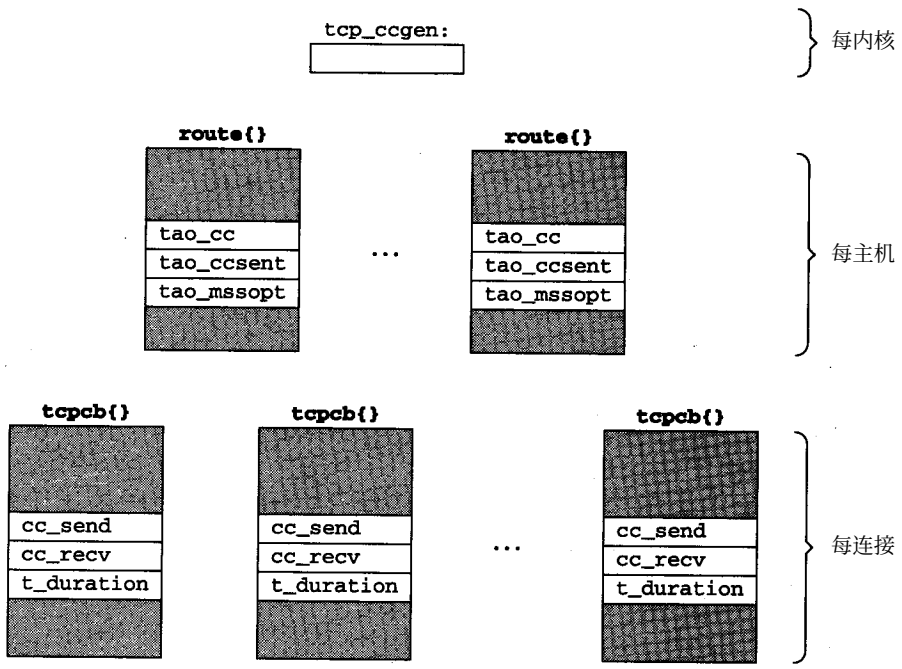


图2-5 T/TCP实现中的变量

续时间小于报文段最大生存时间 MSL，则TIME_WAIT状态将被截断。我们在 4.4节中将更详细地讨论这个问题。

我们在图2-5中给出这些新变量。在后续章节讲 T/TCP协议实现时就用这些变量。

在这个图中，我们用{}表示结构。图中的TCP控制块是一个tcpcb结构。所有TCP协议的实现都必须为其中的连接保存并维护一个控制块，控制块的形式可以这样那样，但必须包含特定连接的所有变量。

2.4 状态变迁图

TCP协议的工作过程可以用图 2-6所示的状态变迁图来描述。大多数状态变迁图都把状态变迁时发送的报文段标在变迁线的边上。例如，从 CLOSED状态到SYN_SENT状态的变迁就标明发送了一个SYN报文段。在图2-6中则没有采用这种标记方法，而是在每个状态框中标出处于该状态时要发送的报文段类型。例如，当处于 SYN_RECV状态时，要发出一个带有 SYN的报文段，其中还包括对所收到 SYN的确认(ACK)。而当处于CLOSE_WAIT状态时，要发出对所收到FIN的确认(ACK)。

我们之所以要这样做是因为，在 T/TCP协议中我们经常需要处理可能造成多次状态变迁的报文段。于是在处理一个报文段时，重要的是处理完报文段后连接所处的最终状态，因为它决定了应答的内容。而如果不使用 T/TCP协议，每收到一个报文段通常至多只引起一次状态变迁，只有在收到SYN/ACK报文段时才是例外，很快我们就要讨论这个问题。

与RFC 793 [Postel 1981b]中的TCP协议状态变迁图相比，图2-6还有另外一些不同之处。

- RFC 793的状态变迁图中，当应用程序发送数据时，会有从 LISTEN状态到SYN_SENT状态的变迁。但实际上典型的 API很少提供这种功能。
- RFC 1122 [Braden 1989]中描绘了一个直接从FIN_WAIT_1状态到TIME_WAIT状态的变迁，这发生在收到了一个带有 FIN和对所发FIN的确认(ACK)的报文段时。但是当收到这样一个报文段时，通常都是先处理 ACK使状态变迁到 FIN_WAIT_2，接着再处理 FIN，并变迁到TIME_WAIT状态。因此，图2-6也能正确处理这样的报文段。这就是收到一个报文段导致两次状态变迁的例子。
- 除了SYN_SENT之外的所有状态都发送 ACK(处于LISTEN这个末梢状态时，则什么也不发送)。这是因为发送 ACK是不受条件限制的：标准 TCP报文段的首部总是留有 ACK的位置。因此， TCP总是确认已接收到的报文段最高序列号 (加1)，只有在处理主动打开(SYN_SENT)的SYN报文段和一些重建(RST)报文段时才是例外。

TCP输入的处理顺序

TCP协议收到报文段时，对其中所携带的各种控制信息 (SYN、FIN、ACK、URG和RST标志，还可能有数据和选项)的处理顺序不是随意的，也不是各种实现可以自行决定的。 RFC 793中对处理顺序有明确的规定。图 11-1给这些步骤做了个小结，该小结同时也用黑体标明了T/TCP中所做的改动。

例如，当T/TCP客户收到一个携带有SYN、数据、 FIN和ACK的报文段时，协议首先处理的是SYN(因为此时的插口还处于 SYN_SENT状态)，接着是处理 ACK标志，再接着是数据，最后才是FIN。三个标志中的任何一个都有可能引起相应插口的连接状态改变。

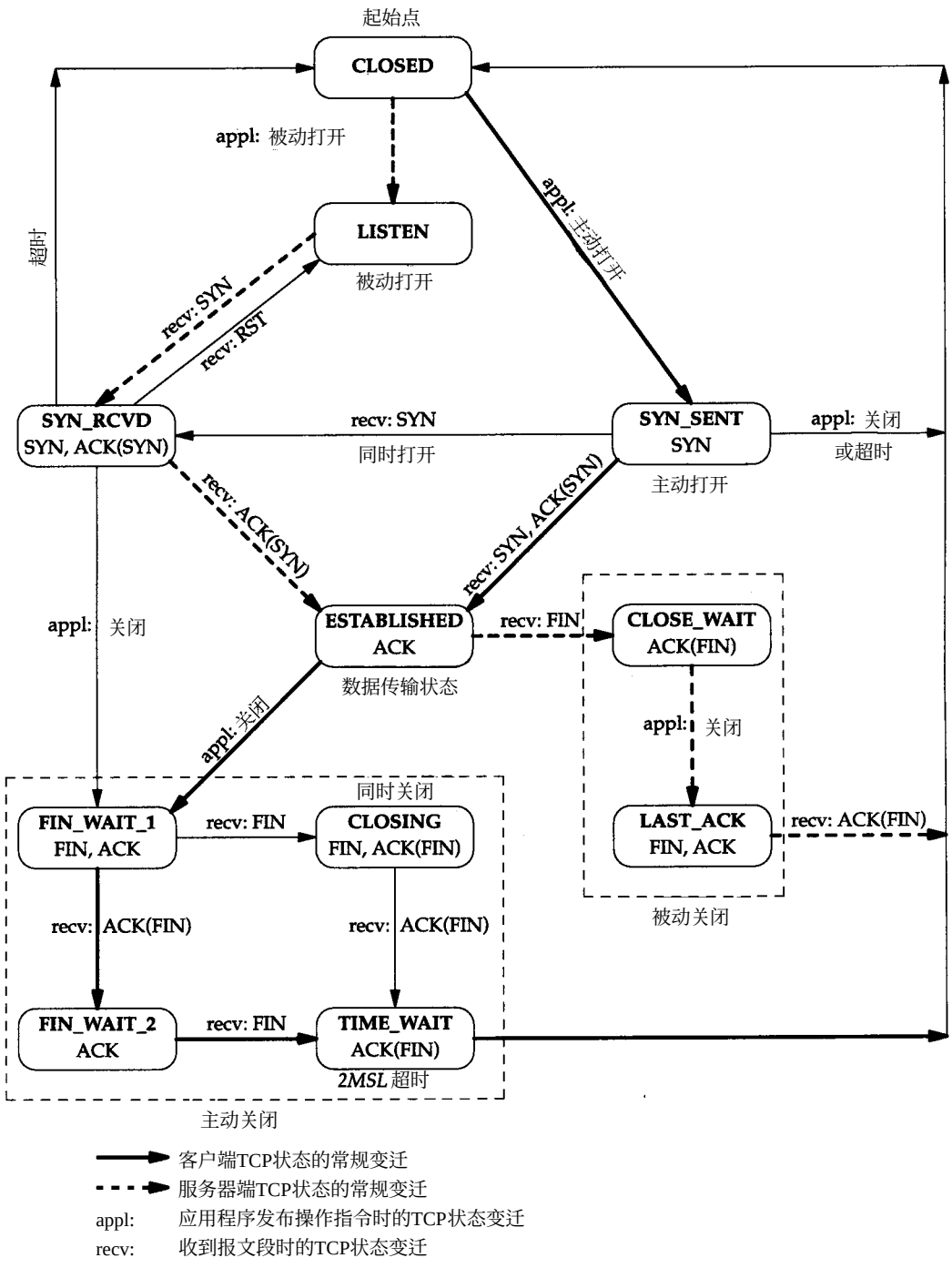


图2-6 TCP的状态变迁图

2.5 T/TCP的扩展状态

T/TCP中定义了7个扩展状态，这些扩展状态都称为加星状态。它们分别是：SYN_SENT*、SYN_RCVD*、ESTABLISHED*、CLOSE_WAIT*、LAST_ACK*、FIN_WAIT_1*和

CLOSING*。例如，在图1-12中，客户发出的第一个报文段中包含有 SYN标志、数据和FIN。当该报文段是在主动打开中发送出去时，客户随即进入 SYN_SENT*状态，而不是进入通常的SYN_SENT状态，这是因为随报文段还必须发出一个 FIN。当收到服务器的应答时，该应答中包含有服务器的SYN、数据和FIN，以及对客户的SYN、数据和FIN的确认(ACK)。这时客户端插口的连接状态要经历一系列的状态变迁：

- 对客户SYN的ACK将连接的状态变迁到FIN_WAIT_1。传统的ESTABLISHED状态就这样完全跳过去了，因为这时客户已经发出了 FIN。
- 对客户FIN的ACK将连接状态变迁到了FIN_WAIT_2。
- 收到服务器的FIN，连接状态变迁到TIME_WAIT。

RFC 1379详细描述了包括所有这些加星状态后的状态变迁图演变过程。当然，得到的结果远比图2-6复杂，其中有很多重叠的线。幸运的是，无星状态和对应的加星状态之间只是一些简单的关系。

- SYN_SENT*状态和SYN_RCVD*状态与对应的无星状态几乎完全相同，唯一的不同之处是在加星状态下要发出一个 FIN。这就是说，当一端主动打开连接、并且应用程序在连接建立之前就指定了 MSG_EOF(发送FIN)时就进入相应的加星状态。在这种情况下，客户端一般是进入SYN_SENT*状态，SYN_RCVD*状态只有当双方碰巧同时执行打开连接操作的偶然情况下才会出现，关于这一点我们在卷 1的18.8节中已有详细讨论。
- ESTABLISHED*、CLOSE_WAIT*、LAST_ACK*、FIN_WAIT_1*和CLOSING*这五个状态与对应的不加星状态除了要发送SYN外也完全相同。当连接处于这五个状态之一时，叫做已经半同步了。当接收端处于被动状态且收到一个带有 TAO测试、可选数据和可选FIN的SYN报文段时，连接即进入这些加星状态 (4.5节详细描述了TAO测试)。之所以用半同步这个词是因为，一旦收到 SYN接收端就认为连接已经建立了 (因为已经通过了TAO测试)，尽管此时刚刚完成了常规三次握手过程的一半。

图2-7给出了加星状态和对应的常规状态。对于每个可能的状态，表中还列出了所发出的报文段类型。

我们将会看到，从实现的角度来看，这些加星的状态是很容易处理的。除了要保持当前已有的无星状态外，在每个连接的TCP控制块中还有两个额外的标志：

- TF_SENDFIN表示需要发送FIN(对应于SYN_SENT*状态和SYN_RCVD*状态)；
- TF_SENDSYN表示需要发送SYN(对应于图2-7中的5个半同步加星状态)。

常规状态	说　明	发　送	加星状态	发　送
CLOSED	关闭	RST, ACK		
LISTEN	监听连接请求(被动打开)			
SYN_SENT	已发出SYN(主动打开)	SYN	SYN_SENT*	SYN, FIN
SYN_RCVD	已经发出和收到SYN；等待ACK	SYN, ACK	SYN_RCVD*	SYN, FIN, ACK
ESTABLISHED	连接已经建立(数据传输)	ACK	ESTABLISHED*	SYN, ACK
CLOSE_WAIT	收到FIN，等待应用程序关闭	ACK	CLOSE_WAIT*	SYN, ACK
FIN_WAIT_1	已经关闭，发出FIN；等待ACK和FIN	FIN, ACK	FIN_WAIT_1*	SYN, FIN, ACK
CLOSING	两端同时关闭；等待ACK	FIN, ACK	CLOSING*	SYN, FIN, ACK
LAST_ACK	收到FIN已经关闭；等待ACK	FIN, ACK	LAST_ACK*	SYN, FIN, ACK
FIN_WAIT_2	已经关闭；等待FIN	ACK		
TIME_WAIT	主动关闭后长达2MSL的等待状态	ACK		

图2-7　TCP根据不同的当前状态(常规或加星)所发送的内容

在图2-7中，加星状态下把SYN和FIN这两个新标志置于开状态时用黑体标出。

2.6 小结

T/TCP的核心是TAO，即TCP加速打开。这项技术使得 T/TCP服务器收到T/TCP客户的SYN报文段后能够知道这个 SYN是新的，从而可以跳过三次握手。确保服务器所收 SYN是新SYN的技术(TAO测试)是为主机已经建立的每个连接分配一个唯一的标识符：连接计数 CC。每个T/TCP主机都要把与每一个对等主机之间最新连接的 CC值保留一段时间。如果所收 SYN报文段的CC值大于从对等主机接收的最新CC值，那么TAO测试成功。

T/TCP定义了3个新的选项： CC、CCnew和CCecho。所有选项都包含一个长度域 (这和RFC 1323中规定的其他选项一样)，使不认识这些选项的TCP实现能跳过它们。如果某个连接使用了T/TCP协议，那么每个报文段都将包含连接计数选项 (不过有时在客户的SYN报文段中用CCnew代替CC)。

T/TCP加入了一个全局内核变量，还在每主机高速缓存中加入了 3个变量，并为正在使用的每个连接控制块增加了3个变量。本书中讨论的T/TCP实现利用业已存在的路由表作为每主机高速缓存。

TCP的状态变迁图有10个状态， T/TCP协议在此基础上还增加了7个额外的状态。但实际上协议实现是简单的：由于新的状态只是已有状态的扩充，因而只需要为每个连接引入两个新的标志，分别指示是否需要发送一个 SYN报文段以及是否需要发送一个FIN报文段，即可定义7种新的状态。

第3章 T/TCP使用举例

3.1 概述

本章中我们将通过几个T/TCP应用程序例子来学习如何使用这3个新引入的TCP选项。这几个例子说明，针对以下几种情形，T/TCP是如何处理的：

- 客户重新启动；
- 常规的T/TCP事务；
- 服务器收到一个过时的重复SYN报文段；
- 服务器重新启动；
- 请求或应答的长度超过报文段最大长度MSS；
- 与不支持T/TCP协议的主机的向下兼容；

下一章我们还将研究另外2个例子：SYN报文段到达服务器没有过时也不重复，但其到达的顺序错乱；客户对重复的服务器SYN/ACK响应的处理。

这些例子中的T/TCP客户是`bsdi`(图1-13)，而服务器则是`laptop`。这些主机上运行的T/TCP客户程序如图1-10所示；T/TCP服务器程序如图1-11所示。客户程序发出长度为300字节的请求，服务器则给出长度为400字节的应答。

在这些例子中，客户程序中支持RFC 1323的部分已经关闭。这样，在客户发起的SYN报文段中就不会含有窗口宽度和时间戳选项(由于只要客户不发送这两个选项，服务器的响应中也不会包含这两个选项，从而服务器是否支持RFC 1323就是无关紧要的)。这样做是为了避免让那些与我们讨论的主题无关的因素把例子弄得太复杂。但在正常情况下，由于时间戳选项可以防止把重复的报文段误认为是当前连接的报文段，因而我们可以在T/TCP应用中支持RFC 1323。也就是说，在宽带连接和大数据量传送的情况下，即便是T/TCP协议也一样需要防止序号重叠(PAWS，见卷1的24.6节)。

3.2 客户重新启动

客户一旦启动，客户-服务器事务过程也就开始了。客户程序调用`sendto`函数，即在路由表中为对端服务器增加一个表项，其中`tao_ccsent`的值初始化为0(表示未定义)。于是TCP协议就会发出CCnew选项而不是发出CC选项。服务器上的TCP协议收到CCnew选项后就执行常规的三次握手操作，其过程可见图3-1所示的Tcpdump的输出(不熟悉Tcpdump操作及其输出的读者可参见卷1的附录A。在跟踪观察这些分组的时候，不要忘了SYN和FIN在序号空间中各占用一个字节)。

从第1行的CCnew选项可以看出，客户端`tcp_ccgen`的值为1。在第2行，服务器对客户的CCnew给出了回应，服务器的`tcp_ccgen`值为18。服务器给客户的SYN发出确认，但不确认客户的数据。由于收到了客户的CCnew选项，即使服务器在其单机高速缓存中有该客户的表项，它也必须完成正常的三次握手过程。只有当三次握手完成以后，服务器的TCP协议才

```
1  0.0                bsdi.1024 > laptop.8888: SFP 36858825:36859125(300)
                                                win 8568 <mss 1460,nop,nop,ccnew 1>
2  0.020542 (0.0205)  laptop.8888 > bsdi.1024: S 76355292:76355292(0)
                                                ack 36858826 win 8712
                                                <mss 1460,nop,nop,cc 18,
                                                 nop,nop,ccecho 1>
3  0.021479 (0.0009)  bsdi.1024 > laptop.8888: F 301:301(0)
                                                ack 1 win 8712 <nop,nop,cc 1>
4  0.029471 (0.0080)  laptop.8888 > bsdi.1024: .
                                                ack 302 win 8412 <nop,nop,cc 18>
5  0.042086 (0.0126)  laptop.8888 > bsdi.1024: FP 1:401(400)
                                                ack 302 win 8712 <nop,nop,cc 18>
6  0.042969 (0.0009)  bsdi.1024 > laptop.8888: .
                                                ack 402 win 8312 <nop,nop,cc 1>
```

图3-1 T/TCP客户重启后向服务器发送一个事务

能把收到的300字节数据提交给当前的服务进程。

第3行显示的是三次握手过程的最后一个报文段：客户对服务器发出 SYN的确认。在这个报文段中客户将FIN重传，但不包括300字节数据。服务器收到该报文段后，立刻确认了收到的数据和FIN(第4行)。与一般的报文段不同的是，这个确认是即时发出的，没有被耽搁。这么做是为了防止客户第1行发出数据后超时而重传。

第5行显示的是服务器给出的应答以及服务器的 FIN，第6行中客户对服务器的FIN和应答都做了确认。注意，第3、4、5和6行中都有CC选项，而CCnew和CCecho选项则分别只出现在第1和第2个报文段中。

从现在开始，我们不再专门地在T/TCP报文段中标示NOP了，因为NOP不是必需的，而且会把图搞复杂。插入 NOP，使选项长度保持为4字节整数倍的做法是出于对提高主机性能的考虑。

机敏的读者可能会注意到，客户端刚刚重新启动时，客户 TCP协议所用的初始序号(ISN)与卷1中习题18.1所讨论的一般模式不一样。而且，服务器的初始序号是个偶数，这在通常从伯克利演变而来的实现中是从来没有的。其原因在于这里的连接所使用的初始序号是随机选取的；而且每隔 500ms对内核的初始序号所加的增量也是随机的。这种改动有助于防护序号攻击，具体内容可见参考文献 [Bellovn 1989]。这种改动是1994年12月[Shimomura 1995]很有名的一次因特网侵入事件发生后，首先在BSD/OS 2.0，然后在4.4BSD-Lite2中加入的。

时间系列图

图3-2给出的是图3-1所描述的报文段交换过程的时序图。

图中，包含数据的第 1和第5这两个报文段用粗黑线标记。图的两侧还分别标注了客户和服务器收到报文段后各自发生的状态变迁。开始的时候，客户进程调用 `sendto`函数，并指定`MSG_EOF`标志，后进入SYN_SENT*状态。服务器收到并处理了第 3个报文段后发生了两次状态变迁。先是处理客户对服务器发出 SYN的确认后，连接的状态由 SYN_RCVD变迁到ESTABLISHED状态；紧接着处理客户发来的FIN又变迁到CLOSE_WAIT状态。当服务器向客户发出设置了`MSG_EOF`标志的应答后，即进入 LAST_ACK状态。注意，客户在第 3个报文段

中重传了FIN标志(回忆一下图2-7)。

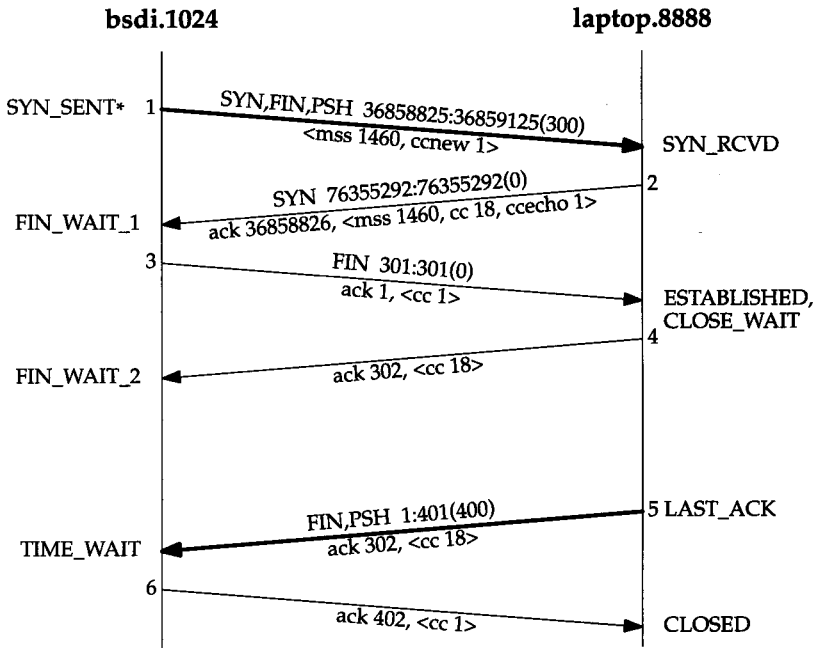


图3-2 图3-1中报文段交换过程的时间系列图

3.3 常规的T/TCP事务

下面我们还是在上面那对客户和服务器之间发起另一次事务。这一次客户在自己的单机高速缓存中取到该服务器的tao_ccsent值非0，于是就发出一个CC选项，其中下一个tcp_ccgen的值为2 (2表示这是客户端重新启动后TCP协议建立的第2个连接)。报文段交换的过程如图3-3所示。

```
1  0.0                  bsdi.1025 > laptop.8888: SFP 40203490:40203790(300)
                                                 win 8712 <mss 1460,cc 2>
2  0.026469 (0.0265)    laptop.8888 > bsdi.1025: SFP 79578838:79579238(400)
                                                 ack 40203792 win 8712
                                                 <mss 1460,cc 19,ccecho 2>
3  0.027573 (0.0011)    bsdi.1025 > laptop.8888: .
                                                 ack 402 win 8312 <cc 2>
```

图3-3 常规的T/TCP客户 - 服务器事务

这是一个常规的、仅包含3个报文段的最小规模T/TCP报文交换过程。图3-4显示了该次报文交换的时序图以及状态变迁过程。

客户发出包含有SYN标志、数据和FIN标志的报文段后进入SYN_SENT*状态。服务器收到该报文段，且TAO测试成功时，进入半同步的ESTABLISHED*状态。其中的数据经处理后交给服务器进程。接着处理完报文段的FIN标志后服务器进入CLOSE_WAIT*状态。由于还未发出SYN报文段，因而服务器一直都处于加星状态。当服务器发出应答并在其中设置MSG_EOF标志后，服务器端随即转入LAST_ACK*状态。如图2-7所示，这个状态的服务器发出的报文段中包含了SYN、FIN和ACK标志。

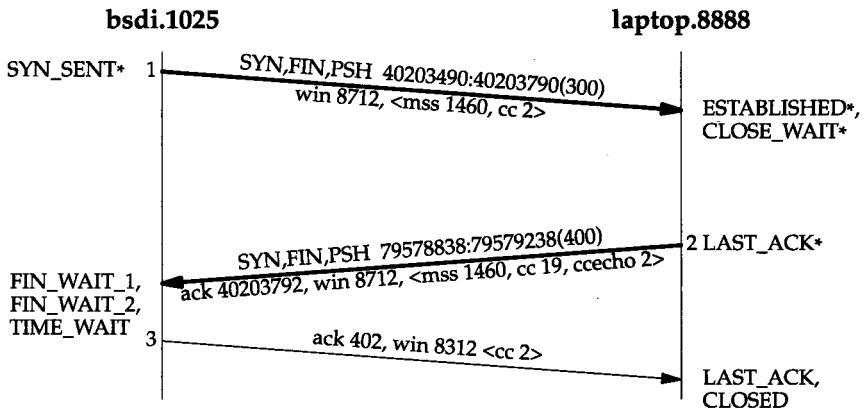


图3-4 图3-3中报文段交换的时间系列图

客户收到第2个报文段后，其中对SYN的确认使客户端的连接状态转入FIN_WAIT_1状态。接着客户处理报文段中对所发FIN的确认，并进入FIN_WAIT_2状态。服务器的应答则送到客户进程。然后，客户处理该报文段中服务器所发的 FIN后进入TIME_WAIT状态。在这个最终的状态，客户发出对服务器所发FIN的确认。

服务器收到第3个报文段后，其中对服务器所发SYN的确认使服务器进入LAST_ACK状态，对服务器所发FIN的确认则使服务器进入CLOSED状态。

这个例子清晰地显示了在T/TCP事务过程中，收到一个报文段是怎样引起多次状态变迁的。它同时也显示了尚不处于ESTABLISHED状态的进程是如何接收数据的：客户进程半关闭 (发出第1个报文段)了与服务器的连接，处于FIN_WAIT_1状态下，但仍然能接收数据 (第2个报文段)。

3.4 服务器收到过时的重复SYN

如果服务器收到了一个看似过时的CC值该怎么办呢？我们让客户发出一个连接计数值 CC为1的SYN报文段，这个值小于服务器刚刚从该客户收到的 CC值(2，见图3-3)。事实上，这种情况也是可能发生的，比如： CC值等于1的这个报文段是客户和服务器之间此前某个连接的，它在网络上耽搁了一段时间，但还没有超过其报文段最大生存时间 (发出后MSL秒)，最终到达了服务器。

一个连接是由一对插口定义的，即包含客户端 IP地址和端口号及服务器端 IP地址和端口号的四元组。连接的新实例称为该连接的替身。

从图3-5我们可以看出，服务器收到一个CC值为1的SYN报文段后强迫执行三次握手操作，因为它无法判断该报文段是过时重复的还是新的。

由于激活了三次握手(这一点我们可以从服务器仅仅确认了客户的SYN而没有确认客户的数据来判断)，服务器的TCP协议在握手过程完全结束以后才会把300字节的数据提交给服务器进程。

本例中，第1个报文段就是一个过时的重复报文段 (客户的TCP此时并不在等待对这个报文段中SYN的响应)，于是当第2个报文段中服务器发出的 SYN/ACK到达时，客户端TCP协议的响应是要求重新建立连接 RST(第3个报文段)。这样做也是理所应当的。服务器的 TCP协议收到这个RST后就扔掉那300字节的数据，而且accept函数也不返回到服务器进程。

```
1  0.0                  bsdi.1027 > laptop.8888: SFP 80000000:80000300(300)
                                                 win 4096 <mss 1460,cc 1>
2  0.018391 (0.0184)    laptop.8888 > bsdi.1027: S 132492350:132492350(0)
                                                 ack 80000001 win 8712
                                                 <mss 1460,cc 21,ccecho 1>
3  0.019266 (0.0009)    bsdi.1027 > laptop.8888: R 80000001:80000001(0) win 0
```

图3-5 T/TCP服务器收到过时的重复SYN报文段

第1个报文段是由一个特殊的测试程序生成的。我们无法让客户的 T/TCP协议自己生成这样的报文段，而只能让它以过时的重复报文段出现。作者曾试着把内核的tcp_ccgen变量值改为 1，但是，正如我们将在图 12-3中看到的，当内核的tcp_ccgen小于它最近一次发给对端的CC值时，TCP协议自动地发出一个CCnew选项而不是发出一个CC选项。

图3-6所示的就是这对客户-服务器之间的下一次、也是常规的一次 T/TCP事务。正如我们所预期的，这是一个包含3个报文段的交换过程。

```
1  0.0                  bsdi.1026 > laptop.8888: SFP 101619844:101620144(300)
                                                 win 8712 <mss 1460,cc 3>
2  0.028214 (0.0282)    laptop.8888 > bsdi.1026: SFP 140211128:140211528(400)
                                                 ack 101620146 win 8712
                                                 <mss 1460,cc 22,ccecho 3>
3  0.029330 (0.0011)    bsdi.1026 > laptop.8888: .
                                                 ack 402 win 8312 <cc 3>
```

图3-6 常规的T/TCP客户-服务器事务

服务器希望这个客户发来的CC值大于2，因此收到CC值为3的SYN后TAO测试成功。

3.5 服务器重启动

现在我们将服务器重新启动，并让客户在服务器刚启动，即服务器监听进程刚开始运行的时候就立即发送一个事务请求。图 3-7为报文段交换的情况。

```
1  0.0                  bsdi.1027 > laptop.8888: SFP 146513089:146513389(300)
                                                 win 8712 <mss 1460,cc 4>
2  0.025420 (0.0254)    arp who-has bsdi tell laptop
3  0.025872 (0.0005)    arp reply bsdi is-at 0:20:af:9c:ee:95
4  0.033731 (0.0079)    laptop.8888 > bsdi.1027: S 27338882:27338882(0)
                                                 ack 146513090 win 8712
                                                 <mss 1460,cc 1,ccecho 4>
5  0.034697 (0.0010)    bsdi.1027 > laptop.8888: F 301:301(0)
                                                 ack 1 win 8712 <cc 4>
6  0.044284 (0.0096)    laptop.8888 > bsdi.1027: .
                                                 ack 302 win 8412 <cc 1>
7  0.066749 (0.0225)    laptop.8888 > bsdi.1027: FP 1:401(400)
                                                 ack 302 win 8712 <cc 1>
8  0.067613 (0.0009)    bsdi.1027 > laptop.8888: .
                                                 ack 402 win 8312 <cc 4>
```

图3-7 服务器刚刚重启动后，T/TCP的交换分组情况

由于客户并不知道服务器已经重新启动了，因而它发出的仍是一个常规的 T/TCP请求，其

中CC值为4(见第1行)。服务器重新启动使其ARP缓存中的客户硬件地址丢失，于是服务器发出一个ARP请求，客户给出应答。服务器强迫执行三次握手操作 (见第4行)，因为它不记得上次从该客户收到的连接计数值CC。

与我们在图3-1中看到的类似，客户发出一个带有 FIN标志的确认报文段完成三次握手过程，300字节的数据则不重传。只有当客户端的重传定时器超时时客户才会重传数据，我们将在图3-11中看到这种情况。收到这第 3个报文段后，服务器立即对数据和 FIN发出确认。服务器发出应答(见第7行)，第8行则是客户给出的确认。

看过图3-7那样的报文交换过程后，我们来看看这对客户和服务器之间接下来继续通信时的一个最小T/TCP事务，如图3-8所示。

```
1  0.0                    bsdi.1028 > laptop.8888: SFP 152213061:152213361(300)
                                                   win 8712 <mss 1460,cc 5>
2  0.034851 (0.0349)      laptop.8888 > bsdi.1028: SFP 32869470:32869870(400)
                                                   ack 152213363 win 8712
                                                   <mss 1460,cc 2,ccecho 5>
3  0.035955 (0.0011)      bsdi.1028 > laptop.8888: .
                                                   ack 402 win 8312 <cc 5>
```

图3-8 常规的T/TCP客户-服务器事务

3.6 请求或应答超出报文段最大长度MSS

到目前为止，在我们所举的所有例子中，无论是客户的请求报文段还是服务器的应答报文段都没有超过报文段最大长度(MSS)。如果客户要发送超出报文段最大长度的数据，而且也确信对等端支持T/TCP协议，那么它就会发出多个报文段。由于对等端的报文段最大长度存储在TAO高速缓存中(图2-5的tao_mssopt)，因而客户的TCP协议能够知道服务器的报文段最大长度，但无法知道服务器的接收窗口宽度 (卷1的18.4节和20.4节分别讨论了报文段最大长度和窗口宽度)。对一个特定的主机来说，报文段最大长度一般是个固定值，而此接收窗口的宽度却会随应用程序改变其插口接收缓存的大小而相应地变化。而且，即使对等端告知了一个较大的接收窗口 (比如说，32 768字节)，但如果报文段最大长度为512字节，那么很可能会有一些中间的路由器无法处理客户一下子发给服务器的前 64个报文段(也即，TCP协议的慢启动是不能跳过的)。T/TCP协议加了两条限制来解决这些问题：

1) T/TCP协议将刚开始时的发送窗口宽度设定为 4 096字节。在Net/3中，这就是变量snd_wnd的值。该变量控制着TCP输出流可以发出多少数据。当对等端带有窗口通告的第1个报文段到达后，窗口宽度的初始值4 096将被改变为所需值。
2) 只有当对等端不在本地时， T/TCP协议才使用慢启动方式开始通信。 TCP协议将snd_cwnd变量设置为1个报文段时就是慢启动。图10-14给出了本地/非本地测试程序，以内核的in_localaddr函数为基础。如果(a)与本机拥有相同的网络号和子网号，或者(b)虽然网络号相同子网号不同，但内核的 subnetsarelocal变量值非0，这样的对等主机就是本地主机。

   Net/3总是用慢启动方式开始每一条连接 (卷2第721页)，但这样就使客户在启动事务时无法连续发出多个报文段。折衷的结果是，允许向本地的对等主机发送多个报文段，但最多4 096 字节。

每次调用TCP协议的输出模块，它总是选择snd_wnd和snd_cwnd中较小的一个作为其可发送数据量的上限值。前者的初始值为TCP滑动窗口通告中的最大值，我们假设为65 535字节(如果使用窗口大小选项，那么这个最大值可以为 65 535×2^{14}，大约为1吉字节)。如果对等主机在本地，那么snd_wnd和snd_cwnd的初始值分别为4 096和65 535。TCP协议在连接刚开始时还未收到对方的窗口通告前，可以发出至多 4 096字节的数据。如果对方通告的窗口宽度为32 768字节，那么 TCP协议可以持续发送数据直到对等主机的接收窗口满为止 (因为32 768和65 535的最小值是32 768)。这样，TCP协议既可以避开慢启动过程，发送数据量又可以受限于对方通告的窗口宽度。

如果对等主机不在本地，那么snd_wnd的初始值仍为4 096，但snd_cwnd的初始值则为1个报文段(假设保存的对等主机报文段最大长度 MSS为512)。TCP协议在连接一开始的时候只能发出一个报文段，当收到对等主机的窗口通告后，每收到一个确认， snd_wnd的值就加1。这时慢启动机制在起作用，可以发出的数据量受限于拥塞窗口，直至拥塞窗口宽度超过了对等主机通告的接收窗口。

作为一个例子，我们对第 1章中的T/TCP客户和服务器程序加以修改，使请求和应答中的数据量分别为3 300和3 400字节。图3-9给出了分组交换过程。

这个例子要显示T/TCP交换的多个报文段的序列号，恰好暴露了 Tcpdump的一个打印bug 。第6、第8和第10个报文段的确认号应当打印3 302而不是1。

```
1   0.0                 bsdi.1057 > laptop.8888: S 3846892142:3846893590(1448)
                                                 win 8712 <mss 1460,cc 7>
2   0.001556 (0.0016)   bsdi.1057 > laptop.8888: .  3846893591:3846895043(1452)
                                                 win 8712 <cc 7>
3   0.002672 (0.0011)   bsdi.1057 > laptop.8888: FP 3846895043:3846895443(400)
                                                 win 8712 <cc 7>

4   0.138283 (0.1356)   laptop.8888 > bsdi.1057: S 3786170031:3786170031(0)
                                                 ack 3846895444 win 8712
                                                 <mss 1460,cc 6,ccecho 7>
5   0.139273 (0.0010)   bsdi.1057 > laptop.8888: .
                                                 ack 1 win 8712 <cc 7>

6   0.179615 (0.0403)   laptop.8888 > bsdi.1057: . 1:1453(1452)
                                                 ack 1 win 8712 <cc 6>
7   0.180558 (0.0009)   bsdi.1057 > laptop.8888: .
                                                 ack 1453 win 7260 <cc 7>

8   0.209621 (0.0291)   laptop.8888 > bsdi.1057: . 1453:2905(1452)
                                                 ack 1 win 8712 <cc 6>
9   0.210565 (0.0009)   bsdi.1057 > laptop.8888: .
                                                 ack 2905 win 7260 <cc 7>

10  0.223822 (0.0133)   laptop.8888 > bsdi.1057: FP 2905:3401(496)
                                                 ack 1 win 8712 <cc 6>
11  0.224719 (0.0009)   bsdi.1057 > laptop.8888: .
                                                 ack 3402 win 8216 <cc 7>
```

图3-9 3 300字节的客户请求和3 400字节的服务器应答

由于客户知道服务器支持T/TCP协议，客户可以立即发出4 096字节。在前2.6 ms的时间里，客户发出了第1、第2和第3个报文段。第1个报文段携带了SYN标志、1 448字节数据和12字节TCP选项(报文段最大长度MSS和连接计数CC)。第2个报文段没有带标志，只有1 452字节数据

和8字节TCP选项。第3个报文段携带FIN和PSH标志、8字节TCP选项以及剩余的400字节数据。第2个报文段是唯一一个没有设置任何 TCP标志(共有6个标志)，甚至不带ACK标志的报文段。通常情况下，ACK标志总是携带的，除非是客户端主动打开，此时的报文段带有 SYN标志(在收到服务器的报文段之前，客户是绝不能发出任何确认的)。

第4个报文段是服务器的SYN报文段，它同时也对客户所发来的所有内容做出了确认，包括SYN标志、数据和FIN标志。在第5个报文段中，客户立即确认了服务器的 SYN报文段。

第6个报文段晚了40 ms才到达客户端，它携带了服务器应答的第 1段数据。客户立即对此给出了确认。第8~11报文段继续同样的过程。服务器的最后一个报文段 (第10行)带有FIN标志，客户发出的最后一个ACK报文段对这最后的数据以及 FIN标志做了确认。

一个问题是，为什么客户对 3个服务器应答报文中的前两个立即给出了确认，是因为它们在很短的时间(44ms)内就到达了吗？答案在 TCP_REASS宏(卷2第726页)中，客户每收到一个带有数据的报文段就要调用该宏。由于连接的客户端处理完第 4个报文段后就进入了FIN_WAIT_2状态，于是在TCP_REASS宏中对连接是否处于ESTABLISHED状态的测试失败，从而使客户端立即发出 ACK而不是延迟一会儿再发。这一“特性”并非 T/TCP协议所独有，在Net/3的程序中，如果任何一端半关闭了TCP连接而进入FIN_WAIT_1或FIN_WAIT_2状态后，都会出现这种情形。从此以后，来自对等主机的每一个数据报文段都立即给予确认。TCP_REASS宏中对是否已进入ESTABLISHED状态的测试使协议无法在三次握手完成之前把数据提交给应用程序。实际上，当连接状态大于 ESTABLISHED时，没有必要立刻确认按序收到的每个报文段(即：应当修改这种测试)。

TCP_NOPUSH插口选项

运行该示例程序之前需要对客户程序再做一些修改。下面这段程序打开了 TCP_NOPUSH插口选项(T/TCP协议新引入的选项)：

```
int  n;

n = 1;
if (setsockopt(sockfd, IPPROTO_TCP, TCP_NOPUSH, (char *) &n, sizeof(n)) < 0)
    err_sys("TCP_NOPUSH error");
```

这段程序在图1-10中调用socket函数之后执行。设置该选项的目的是告诉 TCP协议不要仅仅为了清空发送缓存而发送报文段。

如果要了解设置该插口选项的原因，我们必须跟踪用户进程调用 sendto 函数发送3 300字节数据并设置MSG_EOF标志的请求后内核所执行的动作。

1) 内核最终要调用sosend函数(卷2的第16.7节)来处理输出请求。它把前 2 048字节数据放入一个mbuf簇中，并向TCP协议发出一个PRU_SEND请求。
2) 于是内核调用tcp_output函数(图12-4)。由于可以发送一个满长度 (full-sized)的报文段，因此发出mbuf簇中的前1 448字节数据，并设置 SYN标志(该报文段中包含12字节的TCP选项)。
3) 由于mbuf簇中还剩下600字节数据，于是再次循环调用 tcp_output函数。我们也许会认为Nagle算法将不会使另一个报文段发出去，但是注意卷 2第681页可以看到，第1次执行tcp_output函数后，idle变量的值为1。当程序发出长为1 448字节的第1个报文段后进入again分支时，idle变量没有重新计算。因此，程序在图 9-3所示程序

段(“发送方的糊涂窗口避免(sender silly window avoidance)”)中结束。如果idle变量为真，待发送的数据将把插口发送缓存清空，因此，决定是否发送报文段的是TF_NOPUSH标志的当前值。

在T/TCP协议引入这个标志以前，如果某个报文段要清空插口的发送缓存，并且Nagle算法允许，这段程序就总是会发送一个不满长的报文段。但是如果应用程序设置了TF_NOPUSH标志(利用新的TF_NOPUSH插口选项)，这时TCP协议就不会仅仅为清空发送缓存而强迫发出数据。TCP协议将允许现有的数据与后面写操作补充来的数据结合起来，以期发出较大的报文段。

4) 如果应用程序设置了TCP_NOPUSH标志，那就不会发送报文段，tcp_output函数返回，程序执行的控制权又回到sosend函数。

如果应用程序没有设置TCP_NOPUSH标志，那么协议就发出那个600字节的报文段，并在其中设置PSH标志。

5) sosend函数把剩余的1 252字节数据放入一个mbuf簇，并发出一个PRU_SEND_EOF请求(图5-2)，该请求再次结束tcp_output函数的调用。然而在这次调用之前，已经调用过tcp_usrclosed函数(图12-4)，使连接的状态由SYN_SENT变迁至SYN_SENT*(图12-5)。设置了TF_NOPUSH标志后，当前插口发送缓存中共有1 852字节的数据，于是协议又发出一个满长度的报文段，该报文段包含1 452字节数据和8字节TCP选项(如图3-9所示)。发出该报文段的原因就是因为它是满长度的(亦即：Nagle算法不起作用)。尽管SYN_SENT*状态的标志中包含有FIN标志(图2-7)，但由于发送缓存中还有额外的数据，因此FIN标志被关掉了(卷2第683页)。

6) 程序又执行了一次循环从而再次调用tcp_output函数发送缓存中剩余的400字节数据。然而这一回FIN标志是打开的，因为发送缓存已经空了。尽管图9-3中的Nagle算法不允许发出数据，但由于设置了FIN标志，只有400字节的报文段还是发出去了(卷2第688页)。

本例中，给插口设置了TCP_NOPUSH属性之后，在报文段最大长度MSS为1 460字节的以太网上发出一个3 300字节的请求就引发出3个报文段，长度分别为1 448、1 452和400字节。如果不设置该选项，那么仍然会有3个报文段，但其长度分别为1 448、600和1 252字节。但如果请求的长度为3 600字节，则设置了TCP_NOPUSH选项时产生3个报文段(长度分别为1 448、1 452和700字节)，而不设置该选项就会产生4个报文段(长度分别为1 448、600、1 452和100字节)。

总之，当客户程序仅调用一次sendto函数发出请求时，通常应该设置TCP_NOPUSH插口选项。这样，当请求长度超过报文段最大长度MSS时，协议就会尽可能发出满长度的报文段。这样可以减少报文段的数量，减少的程度取决于每次发送的数据量。

3.7 向后兼容性

我们还需要研究一下如果客户用T/TCP协议给一台不支持T/TCP协议的主机发送数据会发生什么样的情况。

图3-10显示的就是主机bsdi上的T/TCP客户程序向主机svr4(一个运行System V Release 4的主机，不支持T/TCP)上的TCP服务器发起事务时，它们二者之间分组交换的情况。

```
1  0.0                  bsdi.1031 > svr4.8888: SFP 2672114321:2672114621(300)
                                                win 8568 <mss 1460,ccnew 10>
2  0.006265 (0.0063)    svr4.8888 > bsdi.1031: S 879930881:879930881(0)
                                                ack 2672114322 win 4096 <mss 1024>
3  0.007108 (0.0008)    bsdi.1031 > svr4.8888: F 301:301(0)
                                                ack 1 win 9216
4  0.012279 (0.0052)    svr4.8888 > bsdi.1031: .
                                                ack 302 win 3796
5  0.071683 (0.0594)    svr4.8888 > bsdi.1031: P 1:401(400)
                                                ack 302 win 4096
6  0.072451 (0.0008)    bsdi.1031 > svr4.8888: .
                                                ack 401 win 8816
7  0.078373 (0.0059)    svr4.8888 > bsdi.1031: F 401:401(0)
                                                ack 302 win 4096
8  0.079642 (0.0013)    bsdi.1031 > svr4.8888: .
                                                ack 402 win 9216
```

图3-10 T/TCP客户程序向TCP服务器发起事务

客户端的TCP程序发出的第1个报文段中包含有SYN、FIN和PSH标志，还包含了300字节数据。由于客户端的TCP协议在其TAO高速缓存中还没有该服务器主机svr4的连接计数CC值，因而它发出的报文段中带上了CCnew选项。图中第2行就是服务器对该报文段的响应，这是标准三次握手过程中的第2个报文段；而客户端在第3行中对该响应做出了确认。注意：第3行中没有重传数据。

服务器端收到第3行的报文段后，立即确认了客户一开始发过来的300字节数据和FIN标志(如卷2第791页所示，对FIN的确认从不推迟)。服务器端TCP将上述数据保存在队列中，直至三次握手过程结束才将其交给服务进程。

第5行显示的是服务器给出的响应(400字节数据)，客户端在第6行中立刻对此做出了确认。第7行显示的是服务器发出的FIN报文段，客户端同样也迅速地做出了确认。注意，服务器进程无法把第5行的数据和第7行的FIN结合在一起发送。

如果我们在同样还是这一对客户和服务器之间再发起一次事务，则报文段交换的顺序与上一次完全相同。由于在图3-10中服务器端并没有发回一个CCecho选项，因此客户端仍然无法向svr4主机发出带有CC选项的报文段，从而客户端发出的第1个报文段(即初始化报文段)仍然带有CCnew选项，其值为11。支持T/TCP的客户端总是发出CCnew选项的原因是，对不支持T/TCP的服务器，它从来不会更新在其单机高速缓存中的相关表项，因而tao_ccsent值总是0(未定义)。

在下面的例子(图3-11)中，服务器主机运行Solaris 2.4，这也是一个基于SVR4(与图3-10中的服务器一样)的系统，但二者的TCP/IP协议栈实现却完全不同。

第1~3行与图3-10中的相同：带有SYN、FIN、PSH标志和300字节数据的报文段，接着是服务器的SYN/ACK报文段，然后是客户的ACK报文段。这是一次正常的三次握手过程。同样，由于不知道该服务器的连接计数CC值，客户端TCP协议发出的是一个带有CCnew选项的报文段。

Solaris主机发出的每个报文段中携带的“不分段”标志(DF)，用于路径最大传输单元发现(RFC 1191 [Mogul and Deering 1990])。

```
1  0.0                    bsdi.1033 > sun.8888: SFP 2693814107:2693814407(300)
                                                win 8712 <mss 1460,ccnew 12>
2  0.002808 (0.0028)      sun.8888 > bsdi.1033: S 3179040768:3179040768(0)
                                                ack 2693814108 win 8760
                                                <mss 1460> (DF)
3  0.003679 (0.0009)      bsdi.1033 > sun.8888: F 301:301(0)
                                                ack 1 win 8760
4  1.287379 (1.2837)      bsdi.1033 > sun.8888: FP 1:301(300)
                                                ack 1 win 8760
5  1.289048 (0.0017)      sun.8888 > bsdi.1033: .
                                                ack 302 win 8760 (DF)
6  1.291323 (0.0023)      sun.8888 > bsdi.1033: P 1:401(400)
                                                ack 302 win 8760 (DF)
7  1.292101 (0.0008)      bsdi.1033 > sun.8888: .
                                                ack 401 win 8360
8  1.292367 (0.0003)      sun.8888 > bsdi.1033: F 401:401(0)
                                                ack 302 win 8760 (DF)
9  1.293151 (0.0008)      bsdi.1033 > sun.8888: .
                                                ack 402 win 8360
```

图3-11 T/TCP客户向Solaris 2.4上的TCP服务器发送事务请求

不幸的是，我们遇到了在 Solaris的TCP/IP实现中的一个 bug。因为这个 bug，服务器端TCP把第1行中的数据部分扔掉了 (第2个报文段中没有对该数据做确认)，造成客户端的TCP超时，并在第4行重传了数据，同时也重传了 FIN。接着，服务器端确认了客户端发来的数据和FIN(第5行)，然后服务器端在第 6行发出应答。客户端在第 7行对应答给出确认，紧接着是服务器发出FIN报文段(第8行)，最后是客户端的确认(第9行)。

RFC 793 [Postel 1981b]的第30页中指出：“ 尽管这些例子并不证明采用附带数据的报文段也能实现连接同步，但这样处理也完全是合法的，接收端的 TCP只有在搞清楚了数据是正确的以后才能将数据交付给用户 (即，接收端对数据进行缓存，等到连接状态进入 ESTABLISHED以后才能交付给用户) ”。该 RFC的第 66页还说，在LISTEN状态处理接收到的 SYN时，“ 任何其他控制信息和正文数据都要先放入队列待以后处理 ”。

有一个评论者声称，把上述现象叫做“ bug ”是不对的，因为 RFC中并没有强制要求服务器在处理 SYN的同时接受其中附带的数据。声明中还说， Solaris的实现是正确的，因为还没有向客户端通告接收窗口，这时服务器完全可以丢弃已到达的数据，因为这些数据都落在窗口之外。不管你如何评价这个特点 (作者仍然称它们为bug，SUN公司也已经为这个问题分配了一个 Bug ID，即1 222 490，因此也将会在今后的版本中进行修正)，处理这样的情况还要符合健壮性原则，该原则在 RFC 791 [Postel 1981a]中首次提出：“ 你有自由去决定接受什么，但你发送什么却必须遵守规定。”

3.8 小结

我们可以对本章中的例子做下面这样的总结：

1) 如果客户端丢失了服务器的状态信息 (例如，客户端重新启动)，那么客户端在主动打开

时将发出CCnew选项，从而强迫执行三次握手过程。

2) 如果服务器丢失了客户端的状态信息，或者服务器收到的 SYN报文段中的CC值小于期望的值，那么服务器返回给客户的响应将只是一个 SYN/ACK报文段，从而强迫执行三次握手过程。在这种情况下，直到三次握手过程完全结束以后，服务器的 TCP才会把客户在SYN报文段中附带的数据交给上层的服务器进程。

3) 如果服务器想在连接中使用 T/TCP协议，那么它总是用 CCecho选项对客户的 CC或CCnew选项作出应答。

4) 如果客户端和服务器端彼此都掌握有对方的状态信息，那么整个事务过程所收发的报文段个数将达到最少：3个(假设请求和响应的长度都小于或等于报文段最大长度 MSS)。此时收发的分组数最少，时延也最小：为 RTT + SPT。

以上这些例子同时也说明了 T/TCP协议中多个状态的变迁是如何发生的，以及如何使用那些新扩充的(加星的)状态。

如果客户端向一个不支持 T/TCP协议的主机发送带有 SYN、数据和FIN的报文段，那么采用伯克利网络代码的系统 (包括SVR4，但不包括Solaris)能够正确地将数据存储在队列中，直至三次握手过程完成。然而，其他的一些网络代码也有可能错误地把 SYN报文段中的数据扔掉，造成客户端超时，并重传数据。

第4章 T/TCP协议(续)

4.1 概述

本章继续讨论T/TCP协议。我们首先讨论T/TCP客户程序如何根据连接持续时间是否会大于报文段最大生存时间MSL来分配端口号，以及这个分配结果对TCP的TIME_WAIT状态有什么影响。接下来我们研究TCP协议为什么要定义TIME_WAIT状态，因为人们对TCP协议的这一特点普遍缺乏理解。T/TCP协议的重要优点之一就是在连接持续时间小于报文段最大生存时间MSL时，使协议的TIME_WAIT状态由240秒缩短至大约12秒。我们将讨论T/TCP协议是如何实现这一点的，以及这样做的正确性。

本章最后我们将讨论T/TCP协议的TAO，即TCP加速打开。它使T/TCP的客户-服务器事务能够跳过三次握手过程，从而节省了一次往返时间，这也正是T/TCP协议给我们带来的最大好处。

4.2 客户的端口号和TIME_WAIT状态

我们编写TCP客户程序的时候通常不关心如何选择端口号。大部分TCP客户程序(如Telnet、FTP以及WWW等)都是使用临时端口，让主机的TCP模块选择一个当前未使用的端口。从伯克利演变来的系统往往选择1 024~5 000之间的临时端口(见图14-14)，而Solaris则在32 768~65 535之间选择。然而，T/TCP协议根据事务速率和持续时间，对端口号的选择有额外的要求。

常规的TCP主机和常规的TCP客户程序

图4-1描述的是一个TCP客户程序(例如图1-5所示的程序)与同一个服务器之间执行的三次事务，每次事务的持续时间为1秒，前后事务之间的间隔也为1秒。三次连接分别开始于第0秒、第2秒和第4秒，而分别终止于第1秒、第3秒和第5秒。*x*轴表示时间，单位为秒；三次连接分别用粗线段表示。

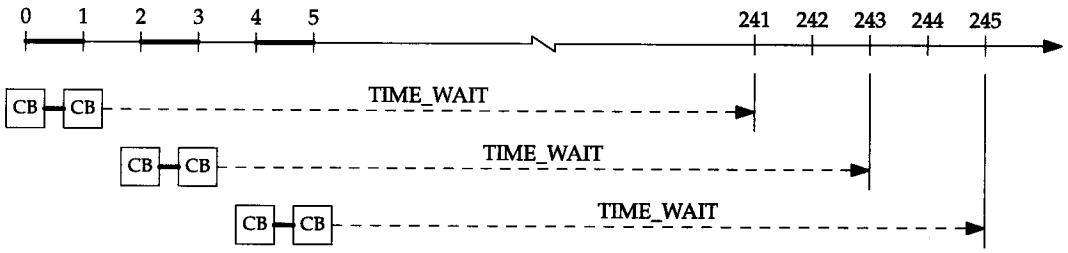


图4-1 TCP客户，不同的事务选用不同的本地端口

每次事务各建立一条TCP连接。我们假定客户程序在创建插口时并不显式地将其绑定到某个端口，而是让系统的TCP模块来选择临时端口。我们还假定客户端TCP模块的报文段最大

生存时间MSL为120秒。第1条连接要保持在TIME_WAIT状态，直至第241秒；第2条和第3条连接则分别从第3秒和第5秒开始保持TIME_WAIT状态，直至第243秒和第245秒。

在图中，CB表示“控制块”，实际上表示连接使用期间和处于TIME_WAIT状态期间，TCP协议维持的几个控制块的组合，包括：Internet进程控制块PCB、TCP控制块和首部模板。在第2章一开始时我们就说过，在Net/3实现中，这3个控制块的大小总和为264字节。除了内存要求以外，TCP协议还需要占用CPU时间来周期性地处理这些控制块(例如在卷2的25.4节和25.5节中，协议每200ms和500ms就要对所有TCP控制块处理一遍)。

Net/3中为每个连接保存一份TCP和IP首部作为“首部模板”(卷2的第26.8节)。该模板中包含了给定连接中用到的所有字段，这些字段在该连接中不会有变化。这样就节省了每次发送报文段的处理时间，因为程序代码只要把首部模板中的内容复制到正在构造的输出分组中即可，而不需要分别填写每个字段。

常规的TCP是无法跳过三次握手过程的。客户程序不能在相继的3条连接中使用同一个本地端口，即使为插口设置了SO_REUSEADDR属性也是如此(卷2第592页给出了一个示例程序)。

T/TCP 主机，每次事务用不同的客户端口

图4-2给出的是与图4-1一样的三次事务序列，但这里我们假定两端的主机都支持T/TCP协议。我们的客户程序与图4-1中的也是同一个。这有很重要的区别：客户和服务器应用程序不需要知道是TCP还是T/TCP，我们只要求两端的主机都支持T/TCP协议(即支持CC选项)。

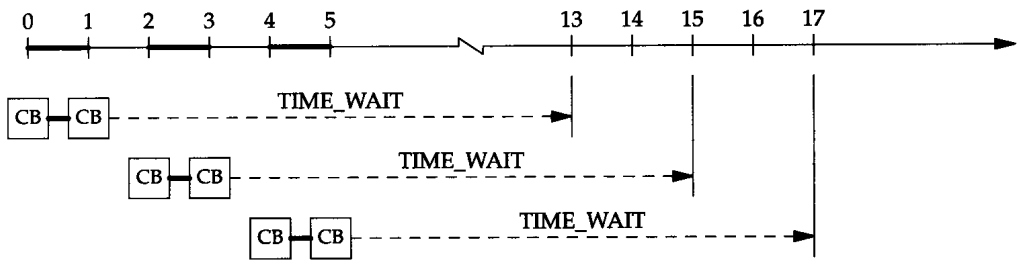


图4-2 当客户和服务器端都支持T/TCP协议时的TCP客户程序

图4-2与图4-1的不同之处在于，连接处于TIME_WAIT状态的时间被截断了，因为两端的主机都支持CC选项。我们这里假定重传超时是1.5秒(在局域网上运行的Net/3中，这是典型值，见[Brakmo and Peterson 1994])，T/TCP的TIME_WAIT是8倍，这样就将TIME_WAIT的保持时间从240秒缩短到了12秒。

当两端的主机都支持CC选项并且连接持续时间小于报文段最大生存时间MSL(120秒)时，T/TCP允许TIME_WAIT状态的保持被截断。这是因为CC选项提供了另外一种保护机制，可以防止过时的重复报文段被投递给另一个新的连接，这一点将在4.4节中讨论。

T/TCP主机，各次事务用同一个客户端口

图4-3给出了与图4-2相同的三次事务的序列，但不同的是，我们在这里假定每次事务中客户端都重复使用同一个端口。为了做到这一点，客户程序必须为插口设置SO_REUSEADDR选

项，并调用bind函数将该插口绑定到某一个特定的本地端口，然后再调用 connect函数(对常规的TCP客户程序)或sendto函数(对T/TCP客户程序)。与图4-2中一样，这里也假定两端的主机都支持T/TCP协议。

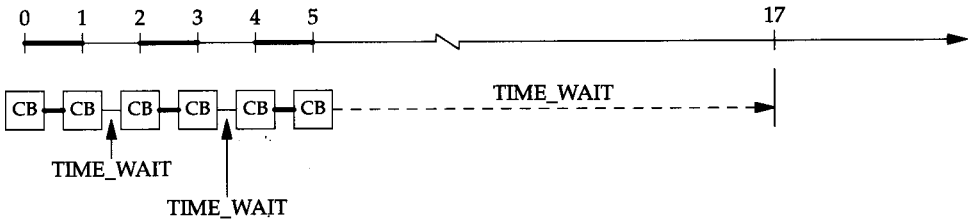


图4-3 TCP客户程序重用同一个端口；客户和服务器主机同时支持 T/TCP协议

在第2秒和第4秒创建连接时， TCP发现了具有相同插口对的控制块，并且正处于TIME_WAIT状态。但是由于前一条连接替身使用了 CC选项，尽管连接的持续时间小于报文段最大生存时间MSL，TIME_WAIT状态的持续时间还是被截断了，并且，当前的连接控制块将被删除，系统将为新的连接分配一个控制块 (新分配的连接控制块可能就是刚刚被删除的旧连接控制块，但那是实现的细节问题。重要的是当前连接控制块的总数没有增加)。当第3条连接在第5秒被关闭后，TIME_WAIT状态的持续时间也只有12秒，与图4-2所示的一样。

总之，本节说明了事务过程中的客户程序有两种可能的优化方式：

1) 不需要改动任何程序源代码，只要客户和服务器端都支持 T/TCP协议，就可将TIME_WAIT的持续时间缩短到连接中重传超时的8倍，而不是原来的240秒。
2) 只修改客户程序，使其重用同一个端口号，这时不但 TIME_WAIT状态的持续时间可以像前一种情况那样截断到连接中重传超时的 8倍，而且，如果同一连接的另一个替身被创建，TIME_WAIT状态就会更快地终止。

4.3 设置TIME_WAIT状态的目的

TIME_WAIT状态是TCP协议中最容易被误解的特性之一。这很可能是因为最初的规约RFC 793中只对该状态做了扼要的解释，尽管后来的 RFC，如RFC 1185，对TIME_WAIT状态做了详细说明。设置TIME_WAIT状态的原因主要有两个：

1) 它实现了全双工的连接关闭。
2) 它使过时的重复报文段作废。

下面我们对这两个原因做进一步的讨论。

TCP全双工关闭

图4-4给出了一般情况下连接关闭时的报文段交换过程。图中还给出了连接状态的变迁和在服务器端测得的RTT值。

图中左侧为客户端，右侧为服务器端。要注意，其中的任何一端都可以主动关闭连接，但一般都是客户端执行主动关闭。

下面我们来看看最后一个报文段 (最后一个ACK)丢失时会发生什么现象。这个现象就给在图4-5中。

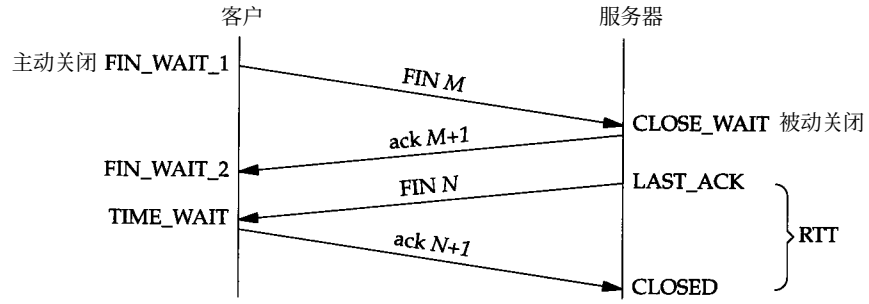


图4-4 通常情况下连接关闭时的报文段交换

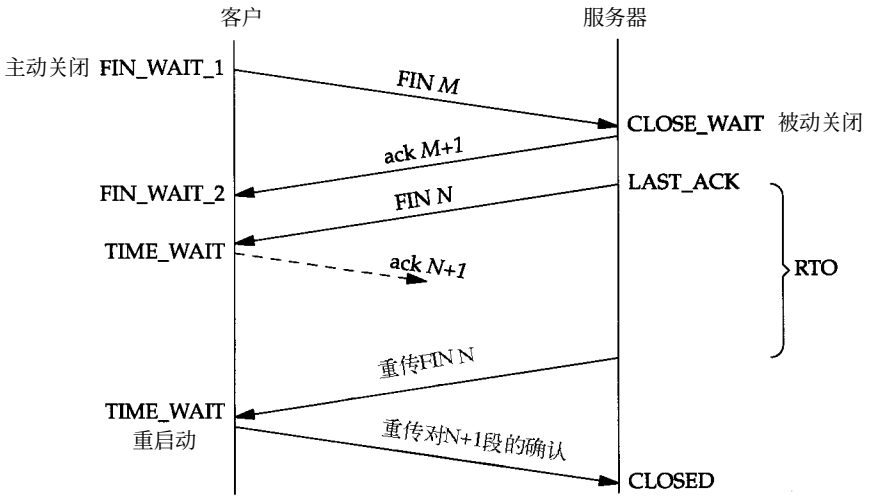


图4-5 最后一个报文段丢失时的TCP连接关闭

由于没有收到客户的最后一个确认，服务器会超时，并重传最后一个 FIN报文段。我们特意把服务器的重传超时(RTO)给得比图4-4中的RTT大，这是因为RTO的取值是估计的RTT值加上若干倍的RTT方差(卷2的第25章详细论述了如何测量 RTT值以及如何计算RTO)。处理最后一个FIN报文段丢失的方法也是一样：服务器在超时后继续重传 FIN。

这个例子说明了为什么TIME_WAIT状态要出现在执行主动关闭的一端：该端发出最后一个ACK报文段，而如果这个 ACK丢失或是最后一个FIN丢失了，那么另一端将超时并重传最后的FIN报文段。因此，在主动关闭的一端保留连接的状态信息，这样它才能在需要的时候重传最后的确认报文段；否则，它收到最后的 FIN报文段后就无法重传最后一个 ACK，而只能发出RST报文段，从而造成虚假的错误信息。

图4-5还说明了另一个问题，即如果重传的 FIN报文段在客户端主机仍处于 TIME_WAIT状态的时候到达，那么不仅仅最后一个 ACK会重传，而且TIME_WAIT状态也重新开始。这时，TIME_WAIT状态的持续时间定时器重置为2倍的报文段最大生存时间，即2MSL。

问题是，执行了主动关闭的一端，为了处理图 4-5所示的情况，需要在 TIME_WAIT状态保持多长的时间？这取决于对端的 RTO值；而RTO又取决于该连接的RTT值。RFC 1185中指出RTT的值超过1分钟不太可能。但实际上 RTO却很有可能长达1分钟：在广域网发生拥塞期间时就会有这种情形。这是因为拥塞会导致多次重传的报文段仍然丢失，从而使 TCP的指数退避算法生效，RTO的值越来越大。

过时的重复报文段失效

设置TIME_WAIT状态的第二个原因是为让过时的重复报文段失效。 TCP协议的运行基于一个基本的假设，即：互连网上的每一个 IP数据报都有一个有限的生存期限，这个期限值是由IP首部的TTL(生存时间)字段决定的。每一台路由器在转发 IP数据报时都要将其 TTL值减1；但如果该IP数据报在路由器中等待的时间超过 1秒，那就要把TTL的值减去等待的时间。实际上，很少有IP数据报在路由器中的等待时间超过 1秒的，因而每个路由器通常都是把 TTL的值减1(RFC 1812 [Baker 1995])的5.3.1节)。由于TTL字段的长度是8比特，因此每个IP数据报所能经历的转发次数至多为255。

RFC 793把该限制定义为报文段最大生存时间 MSL，并规定其值为2分钟。该RFC同时指出，将报文段最大生存时间MSL定义为2分钟是一个工程上的选择，其值可以根据经验进行修改。最后， RFC 793规定TIME_WAIT状态的持续时间为MSL的2倍。

图4-6给出的是一个连接关闭后在 TIME_WAIT状态保持了 2倍报文段最大生存时间(2MSL)，然后发起建立新的连接替身。

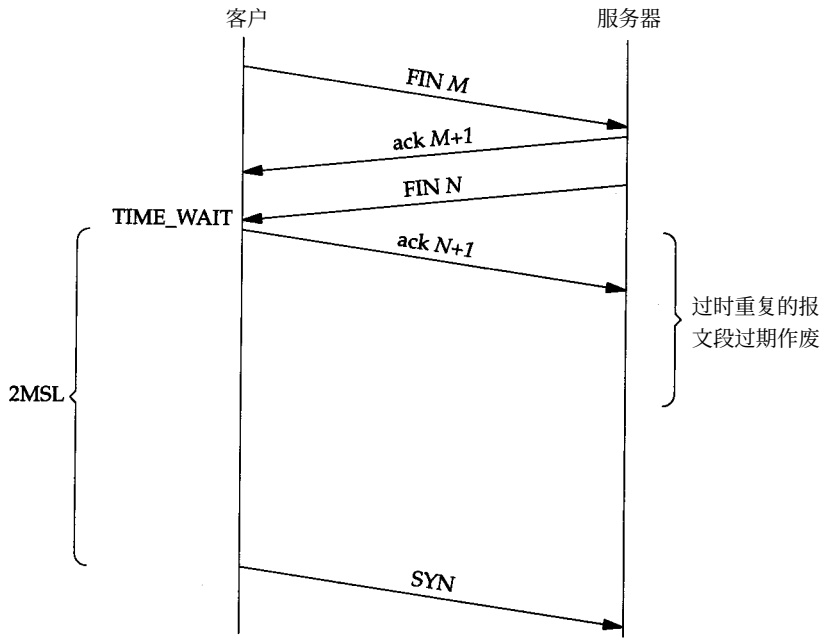


图4-6 前一个连接替身关闭2倍报文段最大生存时间后发起该连接的一个新的替身

由于该连接的新的替身必须在前一个连接替身关闭 2MSL之后才能再次发起，而且由于前一个连接替身的过时重复报文段在 TIME_WAIT状态的第1个报文段最大生存时间里就已经消失，因此我们可以保证前一次连接的过时重复报文段不会在新的连接中出现，也就不可能被误认为是第二次连接的报文段。

TIME_WAIT状态的自结束

RFC 793中规定，处于TIME_WAIT状态的连接在收到RST后变迁到CLOSED状态，这称为TIME_WAIT状态的自结束。 RFC 1337 [Braden 1992a]中则建议不要用 RST过早地结束

TIME_WAIT状态。

4.4 TIME_WAIT状态的截断

我们从图4-2和图4-3中已经看到，T/TCP协议可以截断TIME_WAIT状态的保持时间。采用T/TCP协议后，保持时间由报文段最大生存时间MSL的2倍缩短为RTO(重传超时)的8倍。在4.3节，我们也看到了设置TIME_WAIT状态的原因有两个。那么，截断了该状态的保持时间后，对应于每一个原因都分别产生了什么样的后果？

TCP全双工关闭

设置TIME_WAIT状态的第一个原因是最后一个FIN的重传保持所需的信息。如图4-5所示，花在TIME_WAIT状态的时间实际上应该根据RTO来定，而不是根据MSL。T/TCP中选用乘数8是为了保证对方有足够的时间超时，并重传最后一个报文段。这样就产生了如图 4-2所示的情形：双方都在等待截断的TIME_WAIT保持期(图中为12秒)过去。

但是让我们来看看图4-3中发生的情况。那里因新的客户程序又使用了同一个插口对，从而TIME_WAIT状态在8倍RTO的保持时间到期之前就被截断了。图4-7给出了一个例子。

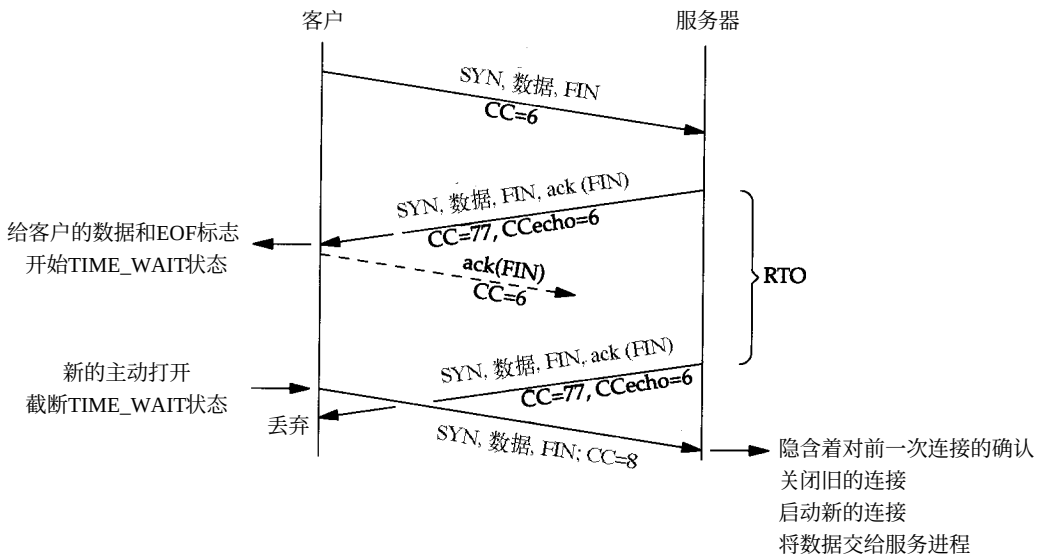


图4-7 最后一个ACK丢失时TIME_WAIT状态被截断的情形

图中，最后一个ACK丢失了，但是客户在收到服务器重传的最后一个报文段之前又再次发起了同一个连接的另一个替身。当服务器收到新的SYN报文段时，由于TAO测试成功(因为8大于6)，这就隐含着确认了服务器的待确认的报文段(图中第2个报文段)。老的连接于是关闭，新的连接开始建立。由于TAO测试成功，新SYN报文段中的数据被交付给服务器进程。

有意思的是，由于T/TCP协议把TIME_WAIT状态的保持时间定义为执行主动关闭一端所测得的RTO的函数，因而就有了一个隐含的假定：两端测得的RTO值相近，并且在一定的范围之内[Olah 1995]。如果执行主动关闭的一端在另一端重传最后的FIN之前就结束了TIME_WAIT状态，那么对重传FIN报文段的响应将是RST而不是重

传端所等待的ACK。

当RTT的值较小，最小的3个T/TCP交换报文段中的第3个报文段丢失，以及客户端和服务器端具有不同的软件时钟速率和不同的 RTO最小值时，就会发生上述情况(第14.7节给出了客户常用的一些 RTO值)。无论如何，当服务器无法测量 RTT的时候(由于第3个报文段丢失)，客户可以测出较小的 RTT值。例如，假设客户测得的 RTT值为10ms，RTO的最小值为100ms，这时客户在收到服务器响应 800ms以后就截断TIME_WAIT状态。但如果服务器是从伯克利演变而来的版本，那么其缺省的 RTO为6秒(如图14-13所示)。当服务器在大约6秒后重传其SYN/ACK/data/FIN时，客户端将发出一个RST作为响应，给服务器应用程序造成一个虚假的错误。

过时的重复报文段失效

TIME_WAIT状态的截断是可行的，因为CC选项能够防止过时重复报文段错误地传递给后续连接。但截断的前提是连接的持续时间小于报文段最大生存时间MSL。考虑图4-8所示的情况。我们让CC生成器(tcp_ccgen)以最大可能速率递增：每两个报文段最大生存时间 (2MSL)就增长$2^{32}-1$。这使得事务速率达到最大，为4 294 967 295除240，大约为每秒18 000 000次事务。

假设tcp_ccgen的值在时刻0时为1，并以上述最大速率递增，那么在 2MSL、4MSL等时刻，tcp_ccgen的值又重新回到1。而且由于tcp_ccgen的值永远不取0，在2MSL的时间里只有$2^{32}-1$个值而不是2^{32}个；因此我们在图中画出 MSL时的2 147 483 648这个值实际上是在MSL时刻之前很短的时间里出现。

我们假设连接始于时刻 0，CC值为1，连接持续时间为 100秒。TIME_WAIT状态从第100秒开始，一直保持到第 112秒，或者在主机发起下一次连接时提前结束 (这里假定RTO为1.5秒，因此TIME_WAIT状态的保持时间为 12秒)。由于连接的持续时间 (100秒)小于报文段最大生存时间MSL(120秒)，因而可以保证该次连接的所有过时重复报文段在第 220秒以后一定会消失。我们还假定tcp_ccgen计数器是以最大可能速率递增的。也就是说，主机在第 0~240秒的时间里建立超过40亿条TCP连接。

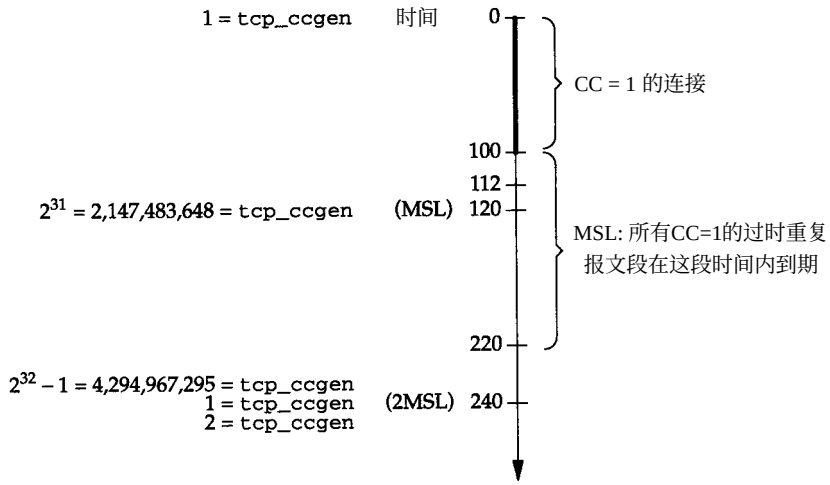


图4-8 持续时间小于MSL的连接：TIME_WAIT状态截断是可行的

因此，只要连接的持续时间小于报文段最大生存时间 MSL，将TIME_WAIT状态的保持时间截断就是安全的，因为CC选项的值直到所有的过时重复报文段都消失以后才会重复。

要明白为什么只有当连接的持续时间小于报文段最大生存时间 MSL时才能截断TIME_WAIT状态的保持时间，那得考察图4-9所示的情形。

我们仍然假设tcp_ccgen计数器以可能的最快速率递增。某个连接开始于时刻 0，CC值为2，持续时间为140秒。由于持续时间大于报文段最大生存时间 MSL，故TIME_WAIT状态不能被截断，从而该连接的插口对只有等到第 380秒以后才能被重用 (从技术上讲，由于我们给定tcp_ccgen在0时刻的值为1，故CC值为2的连接会在0时刻稍后建立，并在第140秒稍后终止。但这不影响我们的讨论)。

在第240~260秒之间，CC值2可以重用。如果 TIME_WAIT状态被截断(比如发生在第 140秒~第152秒之间的某个时刻)，并且如果同一条连接的另一次实现在第 240~第260秒之间重新建立且CC值为2，那么由于所有过时的重复报文段只有在第 260秒以后才会全部消失，这样那些属于前一次连接的过时重复报文段就有可能被误认为是第 2次连接的新报文段。在第240~260秒这段时间，其他的连接 (即不同的插口对)将CC的值取为2并没有什么问题；但是同一个插口对就不能重复使用这个 CC值，因为此时网络中可能还有属于该连接的过时重复报文段。

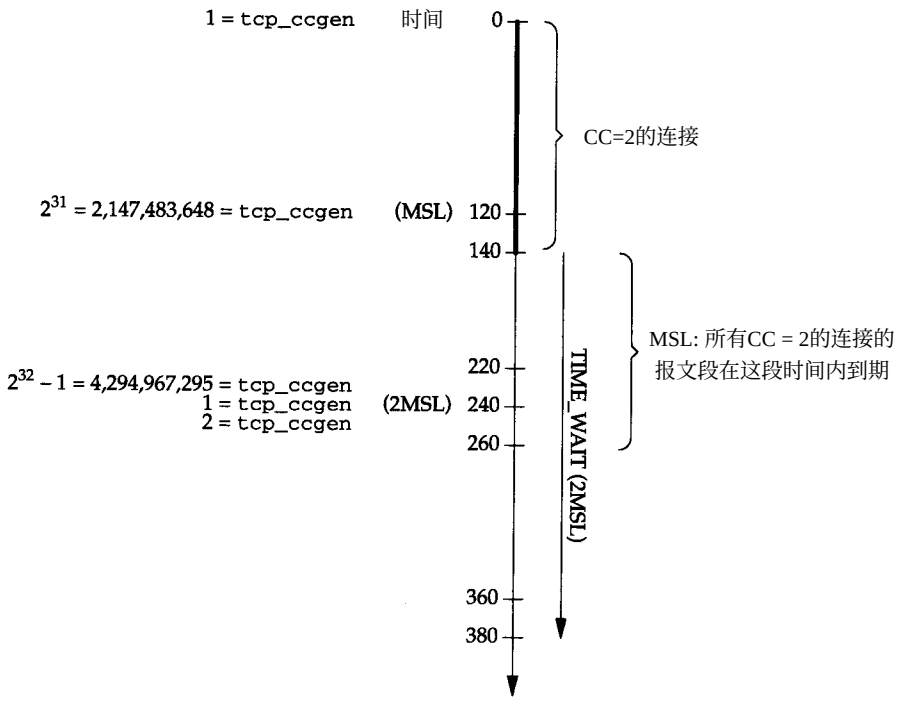


图4-9 持续时间大于MSL的连接：TIME_WAIT状态无法被截断

从应用程序的角度而言，所谓 TIME_WAIT状态截断就意味着客户程序在与同一个服务器进行一系列事务时，必须选择是让一系列连接使用同一个本地端口，还是让每次事务使用各自不同的本地端口。当连接的持续时间小于报文段最大生存时间 MSL时(在事务中这是典型的情况)，重用本地端口可以节约TCP资源(即减少了对控制块内存的需求，见图 4-2和图4-3)。但

是如果客户程序在前一次连接的持续时间大于报文段最大生存时间 MSL的情况下试图重用本地端口，建立连接时将返回EADDRINUSE错误(图12-2)。

如图4-2所示，无论应用程序采用哪种端口使用策略，如果两端的主机都支持 T/TCP协议，而且连接的持续时间小于报文段最大生存时间 MSL，那么TIME_WAIT状态的保持时间总是可以从2倍MSL截断到8倍RTO。这样就节约了资源(即内存和CPU时间)。这对支持T/TCP协议的2台主机之间的任何 TCP连接(如FTP、SMTP、HTTP以及其他等)，只要连接持续时间小于MSL就都适用。

4.5 利用TAO跳过三次握手

T/TCP协议的主要好处就是能够跳过三次握手。为了理解何以能跳过三次握手，我们需要先了解三次握手的目的。 RFC 793中对此只是做了一个简单的说明：“ 引入三次握手的主要原因是为了避免过时的重复连接在再次建连时造成的混乱。为此，专门定义了一个特殊的控制报文reset来解决这个问题 ”。

在三次握手中，每一端都是先发出 SYN报文段，其中含有各自的起始序列号；然后每一端都要确认对方的SYN报文段。这样就可以排除当重复的过时报文段到达某一端时可能带来的混淆。此外，常规的TCP不会在进入ESTABLISHED状态之前就把在SYN报文段中一起传送过来的数据交付给上层的用户进程。

T/TCP协议必须提供一种方法，使收到SYN报文段的一方能够不经过三次握手就保证这个SYN不是过时的重复报文段，从而使得随该 SYN报文段一起传送过来的数据能立刻交付给上层的用户进程。这里的保护手段是客户发出的 SYN报文段中附带的CC选项和服务器缓存的最近一次从该客户收到的合法CC值。

考虑图4-10时序图所示的情况。与图4-8中一样，我们假设tcp_ccgen计数器以最大可能速率递增：即每2MSL递增$2^{32}-1$。

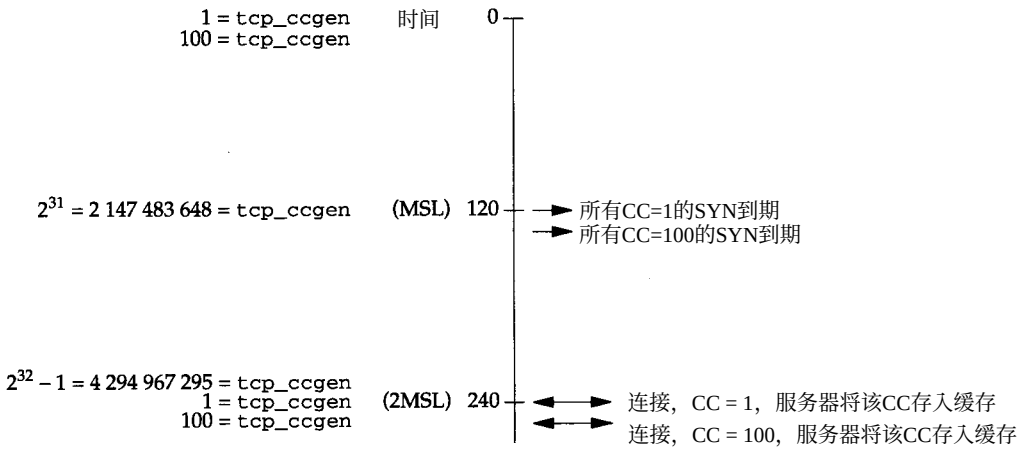


图4-10 SYN中较高的CC值保证该SYN不是过时复制品

在0时刻，tcp_ccgen的值为1；很小的一段时间后其值就变为 100。由于IP数据报的生存期有限，我们可以确信在第 120秒的时候(MSL秒以后)，CC值为1的所有SYN报文段都已经过期而在网络中消失；此后再过一小段时间，所有 CC值为100的SYN报文段也都消失了。

于是在第240秒的时候，又建立了一个CC值为1的新连接。假设服务器对该报文段的 TAO测试成功，那么它就把该客户的最新 CC值缓存下来。此后不久，该服务器主机又收到同一客户的另一个连接请求。由于这时 SYN报文段中的CC的值(为100)比服务器当前保存的该客户最近一个合法CC值(1)要大，而且以前CC值为100的连接中的所有 SYN都已经超过了MSL时间，因而服务器可以断定这是一个新的 SYN。

事实上，这就是RFC 1644中所述的TAO测试：“ 如果某个特定客户主机的第一个 SYN报文段(即只含SYN位而不含ACK位的报文段)中所携带的CC值大于缓存中的该客户 CC值， CC值的单调递增特性可以确保这是一个新的 SYN报文段，可以立即接收下来 ”。正是CC值的单调递增特性以及下面的两个假设确保了 SYN报文段是新的，使得T/TCP协议能够跳过三次握手：

1) 所有的报文段都只有有限的MSL秒的生存期；

2) `tcp_ccgen`计数器在2MSL的时间内的递增量不超过 $2^{32}-1$。

失序的SYN报文段

图4-11给出了两台T/TCP主机和一个失序到达的SYN报文段。这个SYN并不是一个过时重复报文段，只不过是没有按照正确的顺序到达服务器。

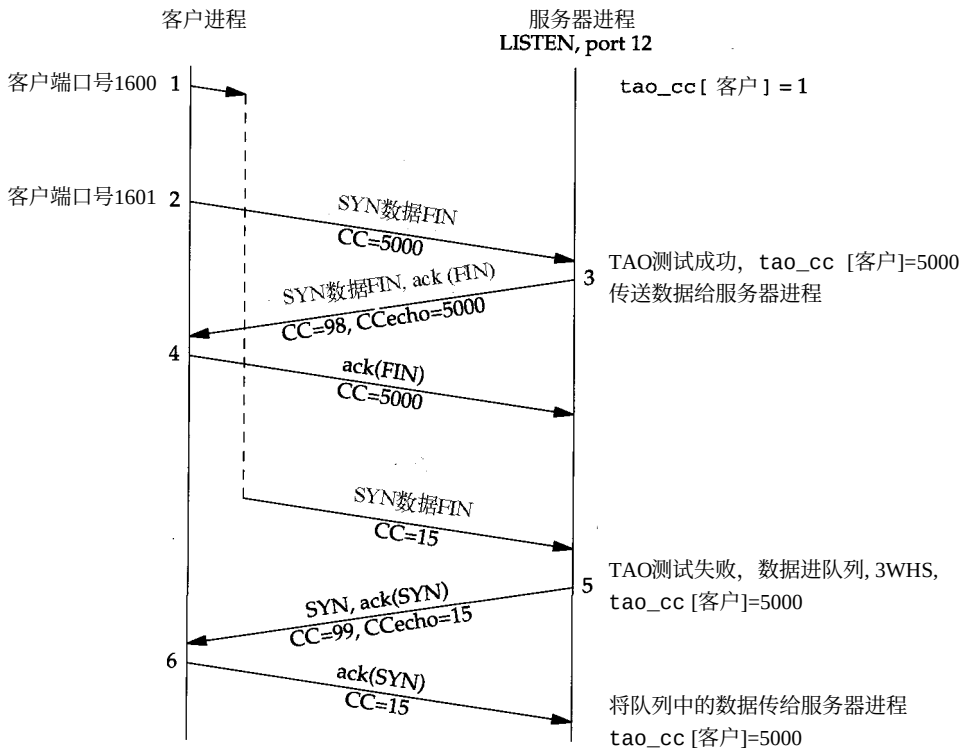


图4-11 两台T/TCP主机和失序到达的SYN报文段

服务器缓存的该客户的CC值为1。报文段1是从客户的端口1600发出的，携带的CC值为15，但它在网络上延迟了一段时间。报文段 2是从客户的端口 1601发出的，携带的 CC值为5000。当报文段2到达服务器时， TAO测试成功(5000大于1)，于是对该客户缓存的最新 CC值改为5000，并把数据交付给进程。报文段3和报文段4完成该次事务。

当报文段1终于到达服务器时，TAO测试失败(15小于5000)，于是服务器给出的响应也是一个SYN报文段，其中带有对所收到SYN的ACK，强迫执行三次握手过程。只有当三次握手完成后才会把数据交付给服务器进程。报文段6结束三次握手过程，队列中的数据于是被交付给服务器进程(图中没有给出该事务的后续过程)。但是，尽管三次握手成功结束，CC值为15的报文段也并不是一个过时的重复报文段(它只是到达的顺序不对)，服务器所记录的该客户的CC值并不更新。如果更新CC会使其值由5000回到15，可能会使服务器错误地收下来自该客户的、CC值介于15~5000之间的过时重复报文段。

翻转了符号位的CC值

在图4-11中我们看到，当TAO测试失败后，服务器强迫执行三次握手；即使握手过程成功结束，服务器所记录的该客户CC值也不更新。从协议的角度出发，这样做是正确的，但却降低了效率。

服务器端TAO测试失败是很可能发生的，因为CC值是客户端生成的。对客户端来说，这是所有连接的全局变量；对服务器来说，则是“翻转了它们的符号位(wrap their sign bit)”。(类似于TCP的序列号，CC值也是无符号的32位数。对CC值进行比较采用模运算，具体算法可见卷2第649~650页。当我们说CC值a的符号位相对于值b“翻转”了时，其具体含义是a的值增加后不再比b大了，反倒是比b小了)。看图4-12。

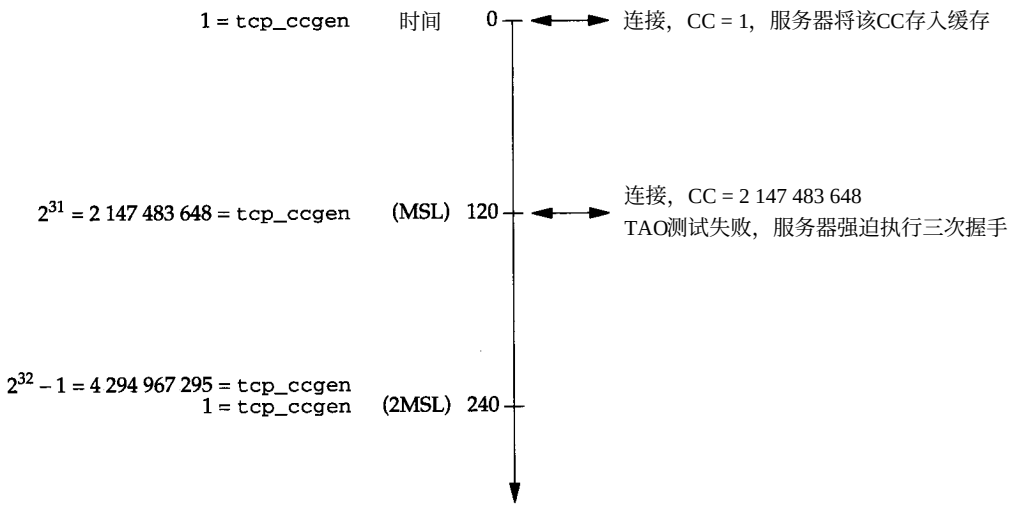


图4-12 CC的值翻转其符号位造成TAO测试失败

客户在0时刻与服务器建立连接，CC值为1。服务器TAO测试成功，并把该CC值记录为该客户当前的CC值。接着该客户与其他服务器建立了2 147 483 646个连接。在第120秒时，它与0时刻建立起连接的服务器又建立一个连接，但此时的CC值为2 147 483 648。服务器收到SYN后，TAO测试失败(按模运算，2 147 483 648比1小，如卷2的图24-26所示)，然后三次握手过程验证了该SYN报文段，但是服务器当前记录的该客户的CC值仍然是1。

这就意味着，从此开始到第240秒的这一段时间里，该客户向该服务器发送任何一个SYN都要经过三次握手过程。这里假设tcp_ccgen计数器持续以最快的速率递增。但更可能的情况是该计数器的递增速率会低得多，意味着计数器由2 147 483 648递增到4 294 967 295就不

只需要120秒了，而可能是几个小时甚至几天的时间。可是在该计数器的符号位再次翻转以前，该客户和该服务器之间的所有T/TCP连接都要执行三次握手。

这个问题的解决需要连接双方的共同努力。首先，不仅服务器要为每个客户记录其最新的CC值，而且客户也要记录发给每个服务器的最新CC值。这两个变量即为图2-5中的tao_cc和tao_ccsent。

其次，当客户发现它将要使用的tcp_ccgen值小于它最近发送给服务器的CC值时，客户就发出CCnew选项而不是CC选项。该选项强迫两台主机再次同步它们各自记录的CC值。

服务器收到一个带有CCnew选项的SYN报文段后，它把该客户的tao_cc标记为0(未定义)。三次握手成功结束后，如果TAO缓存中该客户的表项还是未定义，就将其更新为这次收到的CC值。

由此可见，当客户端没有记录该服务器的CC值(比如客户端重启，或是缓存中对应于该服务器的表项被冲掉)时，或者客户端检测到该服务器的CC值已翻转时，该客户将在其初始SYN报文段中发送CCnew选项而不是CC选项。

重复的SYN/ACK报文段

到目前为止，我们的注意力一直集中在服务器如何确定所收到的SYN报文段是新的还是过时和重复的。这使得服务器可以绕开三次握手的过程。但是客户端又如何确定所收到的服务器响应(SYN/ACK报文段)不是过时和重复的呢？

在常规的TCP协议中，客户端发送的SYN报文段中不带数据，因此服务器要确认的内容只有一个字节：SYN。此外，从伯克利演变来的实现又在建立新连接时将初始发送序号(ISS)递增64 000(TCP_ISSINCR除以2) (见卷2第808页)，这样客户端发送的后续SYN中的初始发送序号就总是比前一次连接的要大。因而过时的重复SYN/ACK报文段就不太可能含有客户端可接受的确认字段。

然而在T/TCP协议中，客户端的SYN报文段中通常都带有数据，这就扩大了客户端从服务器收到可接受确认的范围。RFC 1379 [Braden 1992b]的图7给出的一个例子中，客户端就错误地接受了一个过时的重复SYN/ACK报文段。然而，该例子出现这个问题的原因在于前后两个相继连接的初始发送序号只差100，小于客户端在SYN报文段中附带的数据字节数(300字节)。我们前面提到过，从伯克利演变来的实现中，每建立一个新的连接时总会把ISS增加至少64 000。而64 000已经大于T/TCP协议的默认发送窗口宽度(通常为4 096字节)，这就使客户端不可能接受一个过时的重复SYN/ACK报文段。

> 4.4BSD-Lite2系统中采用了我们在3.2节中讨论过的随机产生ISS值方法。ISS的实际增量平均分布于31 232和96 767之间，均值为63 999。31 232仍然比默认的发送窗口大，下面我们将要讨论的CCecho选项使得这个问题存有争议。

然而，T/TCP协议用CCecho选项对过时的重复SYN/ACK报文段问题提供完整的保护。客户端知道自己发出的SYN报文段中的CC值，而服务器必须在CCecho选项中把该值原样发回给客户端。如果服务器的响应中不含所期望的CCecho值，那么客户端就把该响应丢弃(图11-8)。CC的值具有单调递增特性，而且在至多2 MSL秒的时间内就循环一次，这就可以保证客户端不会接受过时的重复SYN/ACK报文段。

注意，客户不能对服务器传来的SYN/ACK报文段执行TAO测试：太晚了。服务器已经接

受了客户端的SYN，数据已经被交付给了服务器进程，而客户收到的 SYN/ACK报文段中也含有服务器给出的响应。此时客户端再要强迫执行三次握手就太迟了。

重传的SYN报文段

RFC 1644和我们在本节中的讨论都忽略了客户端的 SYN或服务器的SYN的重传可能性。例如，在图4-10中，我们假设tcp_ccgen在0时刻的值为1；接着假设所有CC值为1的SYN报文段在第120秒时都已经过时消失。而实际上也可能是这样的： CC值为1的SYN报文段可能在第0~75秒之间的某个时刻重传了，于是所有CC值为1的SYN报文段在第195秒时才会过时消失，而不是在第120秒时就过时了(如卷2第664页所述，从伯克利演变来的实现中，客户端 SYN报文段和服务器SYN报文段的重传超时上限为75秒)。

这并不影响TAO测试的正确性，但却降低了tcp_ccgen计数器的最大递增速率。前面我们讲过，tcp_ccgen计数器的最大递增速率是在2 MSL秒的时间里递增$2^{32}-1$，从而使最大事务速率达到大约每秒18 000 000次。当考虑到SYN报文段的重传之后，上述最大速率就变为在2 MSL+2 MRX秒的时间里递增$2^{32}-1$，其中MRX是系统规定的SYN报文段重传时限(在Net/3中是75秒)。最大事务速率也由此降低到大约每秒 11 000 000次。

4.6 小结

TCP协议的TIME_WAIT状态有两个功能：

1) 实现了连接的全双工关闭。

2) 使得过时的重复报文段超时作废。

如果T/TCP连接的持续时间小于120秒(1倍的报文段最大生存时间MSL)，那么TIME_WAIT状态的保持时间只要8倍的重传超时，而不是240秒。而且，在前一次连接还处于 TIME_WAIT状态时，客户端就可以建立一个连接的新的替身，从而进一步截短了等待时间。我们说明了为什么这么做是可行的，仅受限于T/TCP协议能支持的最大事务速率(大约每秒18 000 000次)。如果T/TCP客户知道要与同一台服务器执行大量事务，那么它每次都可以使用同一个本地端口号，从而可减少TIME_WAIT状态的控制块的数量。

TCP的TAO(TCP加速打开)测试使得T/TCP的客户 - 服务器程序可以跳过三次握手过程。这是在服务器收到客户端的 CC值大于服务器记录的该客户 CC值时实现的。正是CC值的单调递增性以及以下两点假定才确保了服务器能够断定客户端的 SYN是新的，使T/TCP能够跳过三次握手过程：

1) 所有报文段都有一个有限的生存期限： MSL秒；

2) tcp_ccgen计数器的最大递增速率为：在2MSL秒内增加$2^{32}-1$。

第5章　T/TCP协议的实现：插口层

5.1　概述

从本章开始我们讨论Net/3版中T/TCP协议的实现。我们沿用卷2的编排顺序和表达风格：

- 第5章：插口层
- 第6章：路由表
- 第7章：协议控制块(PCB)
- 第8章：TCP概述
- 第9章：TCP输出
- 第10章：TCP函数
- 第11章：TCP输入
- 第12章：TCP用户请求

在这些章节的介绍中，我们都假设用户已经有了本系列书的卷2或者其中源代码的副本。这样我们就只要介绍实现T/TCP协议所需的1 200行新代码，而不需要重述卷2已介绍过的15 000行代码。

T/TCP协议对插口层所作的改动是很小的：sosend函数需要处理MSG_EOF标志，允许协议调用sendto函数和sendmsg函数隐式地打开和关闭连接。

5.2　常量

T/TCP协议中需要使用三个常量：

1) <sys/socket.h>中定义的MSG_EOF。如果在调用send、sendto或sendmsg函数时设置了该标志，那么利用该连接发送数据就结束了，实际上就是结合了 write和shutdown两个函数的功能。在卷2的图16-12中应该加上该标志。
2) <sys/protosw.h>中定义了一个新的协议请求PRU_SEND_EOF。在卷2的图15-17中应该加上该请求。这个请求是由sosend函数发出的，已在后面的图5-2中给出。
3) <sys/protosw.h>中定义了一个新的协议标志PR_IMPLOPCL(意指“隐式打开和关闭”)。这个标志有两重含义；(a)协议允许在调用sendto或sendmsg函数时给定对等端的地址，而不必在此之前调用connect函数(隐式打开)；(b)协议能够识别MSG_EOF标志(隐式关闭)。注意，只有在面向连接的协议(如TCP)中才需要(a)，因为在无连接的协议中总是可以直接调用 sendto和sendmsg函数而不需要事先调用 connect函数。应该在卷2的图7-9中加上该标志。

   协议代码中switch程序块的TCP入口inetsw[2](卷2中图7-13的第51~55行)应该在其pr_flags的标志值中加上PR_IMPLOPCL。

5.3　sosend函数

sosend函数有两处改动。图5-1所示的代码用来替代卷2第397页中第314~321行的程序代码。

uipc_socket.c

```
320         if ((so->so_state & SS_ISCONNECTED) == 0) {
321             /*
322              * sendto and sendmsg are allowed on a connection-
323              * based socket only if it supports implied connect
324              * (e.g., T/TCP).
325              * Return ENOTCONN if not connected and no address is
326              * supplied.
327              */
328             if ((so->so_proto->pr_flags & PR_CONNREQUIRED) &&
329                 (so->so_proto->pr_flags & PR_IMPLOPCL) == 0) {
330                 if ((so->so_state & SS_ISCONFIRMING) == 0 &&
331                     !(resid == 0 && clen != 0))
332                     snderr(ENOTCONN);
333             } else if (addr == 0)
334                 snderr(so->so_proto->pr_flags & PR_CONNREQUIRED ?
335                        ENOTCONN : EDESTADDRREQ);
336         }
```

uipc_socket.c

图5-1 sosend 函数：差错检查

注意，对代码的替换在这里实际上是从第 320行开始的，而不是第 314行。这是因为在这个文件的前面部分还有一些与 T/TCP无关的修改。由于我们要用这里的 17行代码替换卷2中的8行代码以支持T/TCP协议，因而该文件后面部分代码段中的行号也与卷2中对应的代码段的行号不一样。当本书提到卷 2中的代码段时，我们所说的行号通常都是指在卷 2中的行号。由于卷 3中的代码是卷 2中的相应代码经过增删后得到的，因而相同功能代码段的行号会比较接近，但不一定相同。

320-336 修改后代码段的作用是：当设置了协议的 PR_IMPLOPCL标志、并且调用进程给出了目的地址时，允许在面向连接的插口上调用 sendto函数和sendmsg函数。如果调用进程没有给出目的地址，那么对应于 TCP插口将返回 ENOTCONN，对应于 UDP插口则返回 EDESTADDRREQ。

330-331 这个if语句使得当连接处于SS_ISCONFIRMING状态时，允许只写控制信息而不写任何协议数据。OSI TP4协议采用了这种做法，TCP/IP协议则没有采用。

图5-2所示的是对sendto函数的修改，图中代码用来替代卷 2第400页的第399~403行。

uipc_socket.c

```
415             s = splnet();       /* XXX */
416             /*
417              * If the user specifies MSG_EOF, and the protocol
418              * understands this flag (e.g., T/TCP), and there's
419              * nothing left to send, then PRU_SEND_EOF instead
420              * of PRU_SEND.  MSG_OOB takes priority, however.
421              */
422             req = (flags & MSG_OOB) ? PRU_SENDOOB :
423                 ((flags & MSG_EOF) &&
424                  (so->so_proto->pr_flags & PR_IMPLOPCL) &&
425                  (resid <= 0)) ? PRU_SEND_EOF : PRU_SEND;
426             error = (*so->so_proto->pr_usrreq) (so, req, top, addr, control);
427             splx(s);
```

uipc_socket.c

图5-2 sosend 函数：协议发送

我们第一次看到内容为XXX的评注。这是为了提醒读者，所注释的代码作用不明确，副作用也不明显，抑或是一个难题的快捷解决方法。本例中， splnet函数用于提高处理优先级，以优先执行这段代码。处理优先级用图 5-2底部所示的splx恢复。卷2的1.12节叙述了Net/3中各种中断的级别。

416-427 如果指定了MSG_OOB标志，那就发出PRU_SENDOOB请求。否则，如果指定了MSG_EOF标志，协议又支持PR_IMPLOPCL标志，而且再没有数据要交给协议了(resid小于或等于0)，那就发出PRU_SEND_EOF请求而不是通常的PRU_SEND请求。

回忆3.6节中的例子。应用程序调用 sendto函数发送了3300字节数据，并指定了MSG_EOF标志。在sosend函数执行的第一次循环中，图5-2所示的代码发出了一个PRU_SEND请求，以发送前2 048字节数据(一个mbuf簇)。在第二次循环中，发出PRU_SEND_EOF请求，以发送剩下的1 252字节数据(在另一个mbuf簇中)。

5.4 小结

T/TCP给TCP增加了隐式打开和关闭的功能。所谓隐式打开是指应用程序不是通过调用connect函数建立连接，而是调用sendto函数或sendmsg函数并指定目的地址来建立连接。而隐式关闭则是指允许应用程序在调用 send、sendto或sendmsg函数时指定MSG_EOF标志，从而把输出和关闭合并起来发布。图 1-10中对sendto函数的调用就把打开、写数据和关闭合并在一个系统调用中实现。本章所示的程序代码修改给 Net/3的插口层加上了隐式打开和关闭功能。

第6章 T/TCP的实现：路由表

6.1 概述

T/TCP需要在其每主机高速缓存中为每一个与之进行过通信的主机创建一个记录项。每个记录项包括图2-5所示的tao_cc、tao_ccsent和tao_mssopt三个变量。已有的IP路由表是每主机高速缓存的最合适位置。在 Net/3中，利用卷2第19章介绍的“克隆”标志，很容易为每一个主机创建一个每主机路由表记录项。

在卷2中，我们已经知道网际协议(没有T/TCP)利用了Net/3提供的一般路由表功能。卷2的图18-17说明了调用rn_addroute函数就可增加路由记录，调用rn_delete可以删除路由记录，调用rn_match可以查找路由记录，以及调用rn_walktree可以遍历整棵树(Net/3中用二叉树来存储其路由表，叫做基树(radix tree)。在TCP/IP中，除了这些一般功能外，不再需要有其他功能支持。然而在T/TCP中就不一样了。

既然一个主机可以在一个很短的时间内与成百上千的主机通信(例如几个小时，或者对于一个非常繁忙的WWW服务器来说可能不需要一个小时，详见14.10节的示例)，因此就需要有一些方法使每主机路由表中的路由记录超时作废。本章我们主要研究T/TCP协议在IP路由表中动态创建和删除每主机路由表记录项的功能。

> 卷2中的习题19.2给出了自动地为每一个与之通信的对等主机创建每主机路由表记录项的一个琐细方法。我们在本章中所叙述的方法在概念上与其非常相似，但对大多数TCP/IP路由都能自动进行。习题中创建的每主机路由是不会超时的；创建以后它们就一直存在，直到主机再次启动或者管理人员手工删除。这就需要有一个更好的方法来自动地管理所有的每主机路由。
>
> 并非每一个人都认为已有的路由表是开设T/TCP每主机高速缓存的好地方。另一个方法是将T/TCP每主机高速缓存在内核中作为其自身基树来存储。这项技术(一棵分立的基树)容易实现，利用了内核中已有的一般基树功能，在Net/3的网络文件系统NFS中就采用了这个方法。

6.2 代码介绍

C语言文件netinet/in_rmx.c中定义了T/TCP为TCP/IP的路由功能所增加的函数。这个文件中只包含了我们在本章中所介绍的专门用于Internet的函数。我们将不会介绍卷2第18、19和20章中所叙述的所有路由函数。

图6-1中给出了专门用于Internet的新增路由函数(在本章中介绍的函数用带阴影椭圆表示，函数名字用in_开头)和一般路由函数(这些函数的名字通常用rn_或rt开头)之间的关系。

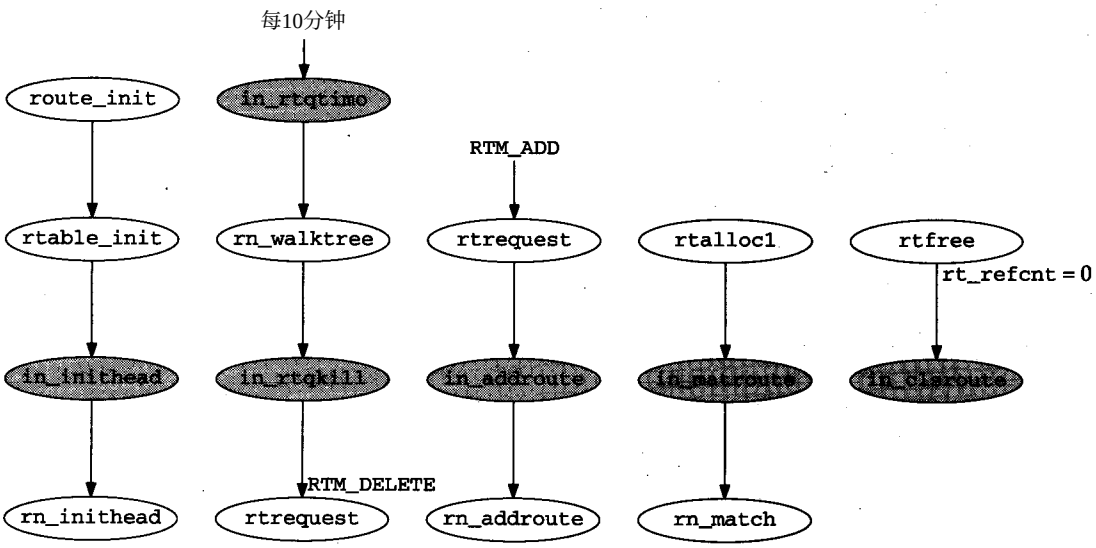


图6-1 专用于Internet的路由函数之间的关系

全局变量

图6-2中给出了专用于Internet的新增全局变量。

FreeBSD版允许系统管理员用sysctl程序修改图6-2中最后三个变量的值，程序要加前缀net.inet.ip。我们没有给出完成这个功能的程序代码，因为它只是对卷2图8-35中的ip_sysctl函数作了一些小小的补充。

变 量	数据类型	说 明
rtq_timeout	int	in_rtqtimo运行的频率(默认值= 每一次10分钟)
rtq_toomany	int	在动态删除开始前有多少路由
rtq_reallyold	int	路由已经确实很陈旧时，存在了多长时间
rtq_minreallyold	int	rtq_reallyold最小值

图6-2 专用于Internet的全局路由变量

6.3 radix_node_head结构

在radix_node_head结构中新增加了一个指针 (卷2的图18-16)：rnh_close。当它指向in_clsroute时，除了指向某个IP路由表外，它的值总是空的，见后面的图 6-7中。

这个函数指针用在rtfree函数中。卷2的图19-5中的第108行和第109行之间要加上下面这些程序行，以说明并初始化自动变量rnh：

```
struct radix_node_head *rnh = rt_tables[rt_key(rt)->sa_family];
```

以下3行程序则加在第112~113行之间：

```
if(rnh->rnh_close && rt-> rt- refcnt == 0) {
        rnh->rnh_close((struct radix_node *)rt, rnh);
}
```

如果这个函数指针非空，并且引用计数达到0，就要调用关闭函数。

6.4 rtentry结构

T/TCP需要在rtentry结构中增加两个路由标志(卷2第464页)。但是现有的rt_flags项是一个16位短整数，并且15位已经占用了(卷2第464页)。为此要在rtentry结构中新增一个标志项rt_prflags。

另一个解决的方法是将短整数rt_flags改为长整数，这种做法在将来的版本中可能会有。

T/TCP使用了rt_prflags的两个标志位。

- RTPRF_WASCLONED是由rtrequest设置的(卷2第488页第335~336行)，从设置了RTF_CLONING标志的记录项创建一个新的记录项时就要设置该标志。
- RTPRF_OURS是由in_clsroute设置的(图6-7)，当IP路由的最后一个克隆参考项关闭时就要设置该标志。这时，要设置一个定时器，以便在将来的某个时间将这个路由表项删除。

6.5 rt_metrics结构

T/TCP修改路由表的目的是在每个路由表记录项中存储附加的每主机信息，实际上也就是三个变量：tao_cc、tao_ccsent和tao_mssopt。为了容纳这些附加的信息，在rt_metrics结构(卷2第464页)中就需要有一个新的字段：

```
u_long rmx_filler[4];   /* protocal family specific metrics */
```

这就有了一个16字节的协议专用向量，T/TCP可以利用，如图6-3所示：

tcp_var.h

```
153 struct rmxp_tao {
154     tcp_cc  tao_cc;             /* latest CC in valid SYN from peer */
155     tcp_cc  tao_ccsent;         /* latest CC sent to peer */
156     u_short tao_mssopt;         /* latest MSS received from peer */
157 };

158 #define rmx_taop(r) ((struct rmxp_tao *)(r).rmx_filler)
```

tcp_var.h

图6-3 T/TCP用作TAO高速缓存的rmxp_tao 结构

153-157 tcp_cc数据类型用于连接计数，是用typedef定义的一个无符号长整数(类似于TCP的序号)。tcp_cc变量的值为0，表示它还未定义。

158 当给定一个指向rtentry结构的指针，宏rmx_taop就返回一个指向相应rmxp_tao结构的指针值。

6.6 in_inithead函数

卷2第504页详细介绍了Net/3中路由表初始化工作的所有步骤。T/TCP所做的第一项修改是将inetdomain结构中的dom_rtattach字段指向in_inithead，而不是指向rn_inithead(卷2第151页)。图6-4中给出了in_inithead函数。

1. 执行路由表的初始化

222-225 rn_inithead用于分配并初始化一个radix_node_head结构。在Net/3中也就

这些功能。该函数的其他功能是 T/TCP新增加的，并且仅仅在“实”路由表初始化时执行。在NFS装载点初始化另一个路由表时也会调用这个函数。

2. 改变函数指针

226-229 rn_inithead还要将radix_node_head结构中取默认值的两个函数指针修改为rnh_addaddr和rnh_matchaddr。它们也是在卷2图18-17中给出的四个指针中的两个。这就使得在调用一般基结点函数前可以执行 Internet中专有的一些动作。 rnh_close函数指针是T/TCP中新加的。

in_rmx.c

```
218 int
219 in_inithead(void **head, int off)
220 {
221     struct radix_node_head *rnh;

222     if (!rn_inithead(head, off))
223         return (0);

224     if (head != (void **) &rt_tables[AF_INET])
225         return (1);             /* only do this for the real routing table */

226     rnh = *head;
227     rnh->rnh_addaddr = in_addroute;
228     rnh->rnh_matchaddr = in_matroute;
229     rnh->rnh_close = in_clsroute;
230     in_rtqtimo(rnh);            /* kick off timeout first time */
231     return (1);
232 }
```

in_rmx.c

图6-4 in_inithead 函数

3. 初始化超时函数

230 in_rtqtimo是超时函数，是第一次调用。这个函数的每一次调用，它总会安排在将来再次被调用。

6.7 in_addroute函数

用rtrequest可以创建一个新的路由表记录项，它们或者是 RTM_ADD命令的结果，也可能是RTM_RESOLVE命令的结果。这两个命令都会从已经存在并且设置了克隆标志的记录项中创建一个新的记录项(卷2第488~489页)。创建以后就要调用rnh_addaddr函数，我们在Internet协议中看到的是in_addroute函数。图6-5给出了这个新函数。

in_rmx.c

```
47 static struct radix_node *
48 in_addroute(void *v_arg, void *n_arg, struct radix_node_head *head,
49             struct radix_node *treenodes)
50 {
51     struct rtentry *rt = (struct rtentry *) treenodes;

52     /*
53      * For IP, all unicast non-host routes are automatically cloning.
54      */
55     if (!(rt->rt_flags & (RTF_HOST | RTF_CLONING))) {
56         struct sockaddr_in *sin = (struct sockaddr_in *) rt_key(rt);
57         if (!IN_MULTICAST(ntohl(sin->sin_addr.s_addr))) {
```

图6-5 in_addroute 函数

```
58              rt->rt_flags |= RTF_CLONING;
59          }
60      }
61      return (rn_addroute(v_arg, n_arg, head, treenodes));
62  }
```
in_rmx.c

图6-5 (续)

52-61 如果所增加的路由不是一个主机路由，也没有设置克隆标志，这时就要检查路由表的主键(IP地址)。如果IP地址不是一个多播地址，该新创建的路由表记录项就要设置克隆标志。rn_addroute为路由表增加记录项。

这个函数的功能是为所有非多播网络路由设置克隆标志，包括默认的路由。这个克隆标志的作用是为任何一个在路由表中能够查到一条非多播网络路由或默认路由的目的地址创建一个新的主机路由。这个新克隆的主机路由是在它第一次查找时创建的。

6.8 in_matroute函数

rtallocl(卷2第483页)在查找一个路由时调用了rnh_matchaddr指针所指向的函数(即图6-6中所示的in_matroute函数)。

```
68 static struct radix_node *
69 in_matroute(void *v_arg, struct radix_node_head *head)
70 {
71      struct radix_node *rn = rn_match(v_arg, head);
72      struct rtentry *rt = (struct rtentry *) rn;

73      if (rt && rt->rt_refcnt == 0) {      /* this is first reference */
74          if (rt->rt_prflags & RTPRF_OURS) {
75              rt->rt_prflags &= ~RTPRF_OURS;
76              rt->rt_rmx.rmx_expire = 0;
77          }
78      }
79      return (rn);
80 }
```
in_rmx.c

图6-6 in_matroute 函数

调用rn_match来查找路由

71-78 rn_match在路由表中查找路由。如果找到了一个路由并且其参考计数值为0，这就是该路由表记录项的第一个参考路由。如果记录项已经超时，也就是说，如果设置了RTPRF_OURS标志，这时就要把这个标志将关闭，并且将rmx_expire定时器设置为0。当路由已经关闭，但在删除前又重新使用该路由时，就往往会发生这种情况。

6.9 in_clsroute函数

我们曾经提到过，T/TCP在radix_node_head结构中增加了一个新的函数指针rnh_close。当参考计数值为零时，这个函数就要在rtfree中调用，这又将调用in_clsroute函数，如图6-7所示。

1. 检查标志

93-99 要做以下的测试：路由必须是正常的，RTF_HOST标志必须是打开的(即这不是一个

in_rmx.c

```
 89 static void
 90 in_clsroute(struct radix_node *rn, struct radix_node_head *head)
 91 {
 92     struct rtentry *rt = (struct rtentry *) rn;

 93     if (!(rt->rt_flags & RTF_UP))
 94         return;
 95     if ((rt->rt_flags & (RTF_LLINFO | RTF_HOST)) != RTF_HOST)
 96         return;
 97     if ((rt->rt_prflags & (RTPRF_WASCLONED | RTPRF_OURS))
 98         != RTPRF_WASCLONED)
 99         return;

100     /*
101      * If rtq_reallyold is 0, just delete the route without
102      * waiting for a timeout cycle to kill it.
103      */
104     if (rtq_reallyold != 0) {
105         rt->rt_prflags |= RTPRF_OURS;
106         rt->rt_rmx.rmx_expire = time.tv_sec + rtq_reallyold;
107     } else {
108         rtrequest(RTM_DELETE,
109                   (struct sockaddr *) rt_key(rt),
110                   rt->rt_gateway, rt_mask(rt),
111                   rt->rt_flags, 0);
112     }
113 }
```

in_rmx.c

图6-7 in_clsroute 函数

网络路由)，RTF_LLINFO标志必须是关闭的(对ARP记录项，该标志要打开)，RTPRF_WASCLONED必须是打开的(记录项是克隆的)，RTPRF_OURS必须是关闭的(该记录项还未超时)。如果这些测试中有任何一项失败，函数都将结束并返回。

2. 设置路由表记录项的终止时间

100-112 在通常的情况下，如果rtq_reallyold非零，就要打开RTPRF_OURS标志，并且要将rmx_expire时间值设置为当前时钟的秒值(time.tv_sec)加上rtq_reallyold值(一般为3 600秒，即1小时)。如果系统管理员用sysctl程序将rtq_reallyold的值设置为0，该路由就会立即被rtrequest删除。

6.10 in_rtqtimo函数

图6-4中，in_inithead首次调用in_rtqtimo函数。每一次调用执行in_rtqtimo时，它都会自动安排在rtq_timeout(默认值为600秒或者10分钟)后再次得到调用。

in_rtqtimo的目的是(用一般的rn_walktree函数)找遍整个IP路由表，对每一个记录项调用in_rtqkill。in_rtqkill要决定是否删除相应记录项。需要从 in_rtqtimo传递有关信息给 in_rtqkill(回顾图6-1)，或者反过来。传递是通过给 rn_walktree的第三个变量实现的。这个变量是 rn_walktree传递给in_rtqkill的一个指针。由于该变量是一个指针，所以信息可以在 in_rtqtimo和in_rtqkill之间的任何一个方向上传递。

in_rtqtimo传递给rn_walktree的指针指向rtqk_arg结构，这个结构如图6-8所示。

in_rmx.c

```
114 struct rtqk_arg {
115     struct radix_node_head *rnh;     /* head of routing table */
116     int     found;                  /* #entries found that we're timing out */
117     int     killed;                 /* #entries deleted by in_rtqkill */
118     int     updating;               /* set when deleting excess entries */
119     int     draining;               /* normally 0 */
120     time_t  nextstop;               /* time when to do it all again */
121 };
```

in_rmx.c

图6-8 rtqk_arg 结构：in_rtqtimo 与in_rtqkill 之间传递的信息

研究in_rtqtimo函数时，我们可以看到这些字段是怎样用的。如图 6-9所示。

in_rmx.c

```
159 static void
160 in_rtqtimo(void *rock)
161 {
162     struct radix_node_head *rnh = rock;
163     struct rtqk_arg arg;
164     struct timeval atv;
165     static time_t last_adjusted_timeout = 0;
166     int     s;

167     arg.rnh = rnh;
168     arg.found = arg.killed = arg.updating = arg.draining = 0;
169     arg.nextstop = time.tv_sec + rtq_timeout;
170     s = splnet();
171     rnh->rnh_walktree(rnh, in_rtqkill, &arg);
172     splx(s);

173     /*
174      * Attempt to be somewhat dynamic about this:
175      * If there are 'too many' routes sitting around taking up space,
176      * then crank down the timeout, and see if we can't make some more
177      * go away.  However, we make sure that we will never adjust more
178      * than once in rtq_timeout seconds, to keep from cranking down too
179      * hard.
180      */
181     if ((arg.found - arg.killed > rtq_toomany) &&
182         (time.tv_sec - last_adjusted_timeout >= rtq_timeout) &&
183         rtq_reallyold > rtq_minreallyold) {
184         rtq_reallyold = 2 * rtq_reallyold / 3;
185         if (rtq_reallyold < rtq_minreallyold)
186             rtq_reallyold = rtq_minreallyold;

187         last_adjusted_timeout = time.tv_sec;
188         log(LOG_DEBUG, "in_rtqtimo: adjusted rtq_reallyold to %d\n",
189             rtq_reallyold);
190         arg.found = arg.killed = 0;
191         arg.updating = 1;
192         s = splnet();
193         rnh->rnh_walktree(rnh, in_rtqkill, &arg);
194         splx(s);
195     }
196     atv.tv_usec = 0;
197     atv.tv_sec = arg.nextstop;
198     timeout(in_rtqtimo, rock, hzto(&atv));
199 }
```

in_rmx.c

图6-9 in_rtqtimo 函数

1. 设置rtqk_arg结构并调用rn_walktree

167-172 rtqk_arg结构的初始化包括：在IP路由表的首部设置rnh，计数器found和killed清零，draining和update标志清零，将nextstop设置为当前时间(秒级)加上rtq_timeout(600秒，即10分钟)。rn_walktree要找遍整个IP路由表，对每一个记录项调用in_rtqkill(图6-11)。

2. 检查路由表记录项是否过多

173-189 如果以下三个条件满足，就说明路由表中的记录项过多：

1) 已经超时但仍未删除的路由表记录项数(found减去killed)超过了rtq_toomany(默认值为128)。

2) 上一次执行本项操作至今所经过的秒数超过了rtq_timeout(600秒，即10分钟)

3) rtq_really超过了rtq_minreallyold(默认值为10)。

如果以上条件全部成立，则将rtq_reallyold设置为其当前值的2/3(用整数除法)。由于该值的初始值为3 600秒(60分钟)，因此它的取值就会分别是3600、2400、1600、1066和710，等等。但是该值不允许低于rtq_minreallyold(默认值为10秒)。当前时间值记录在静态变量last_adjusted_timeout中，并且要有一个调试消息发送给syslogd守护程序([Stevens 1992]的13.4.2节中给出了如何用log函数发送消息给syslogd守护程序)。这段代码以及减少rtq_reallyold值的目的是缩短路由表的处理周期，在路由表中记录项过多时删除过时的路由。

190-195 rtqk_arg结构中的计数器found和killed又初始化为0，updating标志此时设置为1，再次调用rn_walktree。

196-198 in_reqkill函数将rtqk_arg结构中的nextstop字段设置为下次调用in_rtqtimo的时间。内核的timeout函数会安排这个事件在需要的时候发生。

> 每10分钟就游历整个路由表一遍需要多大的开销？很明显这依赖于路由表中记录项的数目。在14.10节中我们模拟了一个繁忙的Web服务器中的T/TCP路由表大小，发现即使24小时内服务器要与5 000个不同的客户连接，并且主机路由的超时间隔为1小时，路由表中也从不会超过550个记录项。目前，一些Internet主干路由器中有成千上万条的路由表记录项，但是它们不是主机，而是路由器。我们并不会希望主干路由器支持T/TCP，因此也不必有规律地游历这样一个非常大的路由表以删除过时的路由。

6.11 in_rtqkill函数

rn_walktree要调用in_rtqkill函数，其中rn_walktree又是被in_rtqtimo调用的。我们在图6-11中所示的程序in_rtqkill就用于在必要时删除IP路由表记录项。

1. 只处理已经超时的记录项

134-135 这个函数只对设置了RTPRF_OURS标志的记录项进行处理，也就是说，只处理已经被in_clsroute关闭了的记录项(即它们的参考计数值已经达到零)和已经过了一个超时间隔(通常为1小时)而过期的记录项。这个函数不影响正在使用的路由 (因为这些路由的RTPRF_OURS标志不会打开的)。

136-146 如果设置了draining标志(在当前的实现中是永远不会设置的)，或者超时间隔已到(rmx_expire时间小于当前时间)，相应的路由就被rtrequest删除。rtqk_arg结构中

的found字段累计已经设置了RTPRF_OURS标志位的路由表记录项数，killed字段则用于累计被删除的记录项数。

147-151 else语句在当前记录项还没有超时时执行。如果设置了updating标志(图6-9中我们已经看到，当过期的路由太多时就会设置该标志，或者下一次对整个路由表进行处理时也会设置该标志)，并且还远未到过期时间(一定是在将来某一时刻，以便相减时产生一个正值)，这时就将过期时间重新设置为当前时间加上rtq_reallyold。考虑图6-10所示的例子就容易理解了。

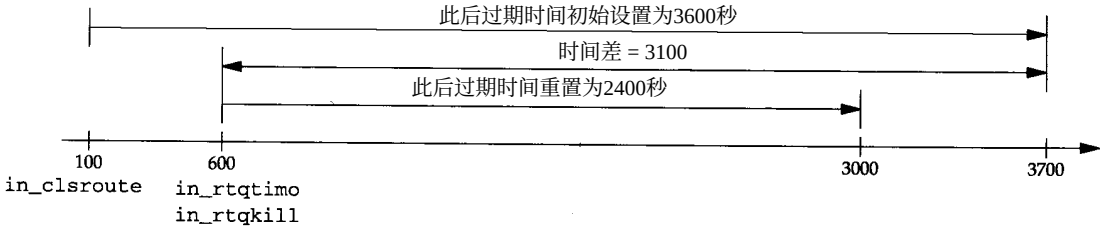


图6-10 in_rtqkill 重新设置过期时间

in_rmx.c

```
127 static int
128 in_rtqkill(struct radix_node *rn, void *rock)
129 {
130     struct rtqk_arg *ap = rock;
131     struct radix_node_head *rnh = ap->rnh;
132     struct rtentry *rt = (struct rtentry *) rn;
133     int     err;

134     if (rt->rt_prflags & RTPRF_OURS) {
135         ap->found++;

136         if (ap->draining || rt->rt_rmx.rmx_expire <= time.tv_sec) {
137             if (rt->rt_refcnt > 0)
138                 panic("rtqkill route really not free");

139             err = rtrequest(RTM_DELETE,
140                             (struct sockaddr *) rt_key(rt),
141                             rt->rt_gateway, rt_mask(rt),
142                             rt->rt_flags, 0);
143             if (err)
144                 log(LOG_WARNING, "in_rtqkill: error %d\n", err);
145             else
146                 ap->killed++;
147         } else {
148             if (ap->updating &&
149                 (rt->rt_rmx.rmx_expire - time.tv_sec > rtq_reallyold)) {
150                 rt->rt_rmx.rmx_expire = time.tv_sec + rtq_reallyold;
151             }
152             ap->nextstop = lmin(ap->nextstop, rt->rt_rmx.rmx_expire);
153         }
154     }
155     return (0);
156 }
```

in_rmx.c

图6-11 in_rtqkill 函数

图中x轴为时间，单位是秒。一个路由在时刻100时被in_clsroute关闭(当它的参考计

数值达到零时)，同时rtq_reallyold有了初始值3600(1小时)。这样，这个路由的过期时间就为3 700。但在时刻600，执行了in_rtqtimo，并且路由未删除 (因为它的过期时间是在3 100秒，还未到)，但由于路由记录项太多，使得 in_rtqtimo将rtq_reallyold的值重置为2 400、将updating设置为1，并且rn_walktree再次处理整个 IP路由表。此时in_rtqkill发现updating已经是1并且路由将在3100秒时过期。因为3100大于2400，过期时间就要重置为过2400秒以后，也就是在3000秒的时刻。路由表变大时，过期时间也就变短。

2. 计算下一个过期时间

152-153 每当发现一个记录项已经过期但其过期时间还未到时，就要执行这段代码。nextstop要设置为其当前值与路由表记录项过期时间中的最小值。前面讲过， nextstop的初始值是由in_rtqtimo设置的，设置值为当前时间加上 rtq_timeout的值(即10分钟以后)。

想想图6-12所示的例子。*x*轴代表时间，单位为秒，黑点的时刻为 0、600、……，等等，是调用in_rtqtimo函数的时刻。

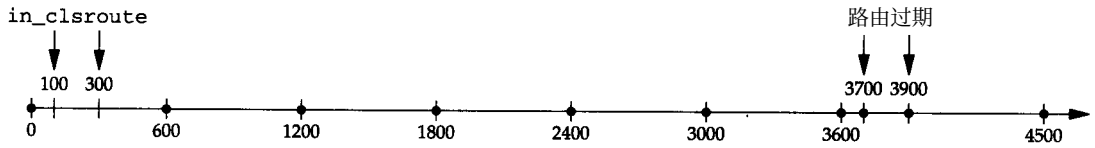


图6-12 根据路由过期时间执行 in_rtqtimo

in_addroute创建一个IP路由，然后在时刻100时被in_clsroute关闭。它的过期时间设置为3700(1小时以后)。在时刻300创建了第二个路由，然后被关闭，其过期时间设为 3900。in_rtqtimo函数每十分钟就执行一次，分别是在时刻 0、600、1200、1800、2400、3000和3600等。从时刻0至时刻3000，nextstop的值设置为当前时刻加上 600，这样在时刻3000为这两个路由分别调用in_rtqkill时，nextstop就改为了3600，因为3600小于3700和3900。但是在时刻3600为这两个路由分别又调用 in_rtqkill时，nextstop就设置为3700，因为3700小于3900，也小于4200。这就意味着在时刻3700将再次调用in_rtqtimo，而不是在时刻4200。此外，当in_rtqkill在时刻3700被调用时，因为另一个路由需要在时刻 3900过期，这就要将 nextstop设置为 3900。假设没有别的 IP路由要过期，在时刻 3900执行了in_rtqtimo以后，它将在4500、5100、……，等等时刻再次执行。

过期时间的交互影响

在路由表记录项的过期时间和 rt_metrics结构中的rmx_expire字段之间会有一些微小的交互影响。首先，地址解析协议 ARP也同样用该字段实现 ARP记录项的超时 (卷2的第21章)。这意味着路由表中有关本地子网中某一主机的路由表记录项 (以及与其相关的TAO信息)在该主机的ARP记录项被删除时也会同时被删除，通常是每 20分钟执行一次删除。这个间隔比in_rtqkill(1小时)所用的默认过期时间要短得多。回顾前面应该记得， in_clsroute明确地忽略已设置了 RTM_LLINFO标志的ARP记录项(图6-7)，让ARP对它们执行超时处理，而不用in_rtqkill。

其次，执行route程序去读取并打印一个克隆的 T/TCP路由表记录项的度量数据和过期

时间的副作用是会重置其过期时间。这种情况是这样的。假设有一个使用过的路由关闭了 (其参考计数值变为零)。关闭时，其过期时间设置为 1小时以后。59分钟过去了，但就在它即将过期前的1分钟，调用了route程序来打印这个路由的度量数据。以下是执行 route程序时需要调用的内核函数：route_output调用rtallocl，rtallocl又调用in_matroute(Internet专有的函数rnh_matchaddr)，in_matroute函数加大了参考计数值，即从 0到1。当这些操作全部完成后，假设参考值又从 1回到0，rtfree就调用in_clsroute，而in_clsroute将过期时间重置为1小时以后。

6.12 小结

在T/TCP中，我们为rt_metrics结构增加了16字节。其中的10个字节被T/TCP用作TAO缓存：

- tao_cc，从对等端收到的最后一个有效SYN中的CC值；
- tao_ccsent，发给对等端的最后一个CC值；
- tao_mssopt，从对等端收到的最后一个MSS值。

在radix_node_head结构中新增加了一个函数指针：rnh_close字段，当路由的参考计数值达到0时，就要调用该指针所指的函数(如果有定义)。

专门为Internet协议增加了四个新的函数：

1) in_inithead用于初始化Internet的radix_node_head结构，设置我们现在讲述的这四个函数指针。
2) in_addroute在IP路由表中增加新路由时调用。它为每一个非主机路由和非多播地址路由的IP路由打开克隆标志。
3) in_matroute在每次查找到IP路由时调用。如果路由被in_clsroute函数设置为超时，就要把它的过期时间重置为0，因为这个路由又有用了。
4) in_clsroute在IP路由的最后一个参考也被关闭时调用。它将路由的过期时间设置为1小时以后。我们也看到了，如果路由表过大时，过期时间要缩短。

第7章 T/TCP实现：协议控制块

7.1 概述

对于T/TCP而言，协议控制块 PCB函数(卷2的第22章)需要作一些小的修改。函数in_pcbconnect(卷2第22.2节)现在要分为两部分：一个名为 in_pcbladdr的内部例程，用于分配本地接口地址；另一个为 in_pcbconnect函数，完成原来的功能 (它要调用in_pcbladdr)。

我们把这两部分功能分开的原因是因为，当同一连接 (即相同的插口对)的前一次操作还处在TIME_WAIT状态时，T/TCP就可以发布下一个connect了。如果先前一次连接的持续时间少于MSL，并且两端都使用了CC选项，那么处于TIME_WAIT状态的连接就关闭，允许建立新的连接。如果我们没有做上述修改，并且 T/TCP使用了未修改的in_pcbconnect，当遇到现有PCB处于TIME_WAIT状态时，应用程序就会收到“地址已被使用”这样的出错消息。

不仅在发布TCP的connect时要调用in_pcbconnect，并且在新的TCP连接请求到达时、发布UDP connect以及发布UDP sendto时都要调用该函数。图7-1总结了Net/3中修改之前的调用关系。

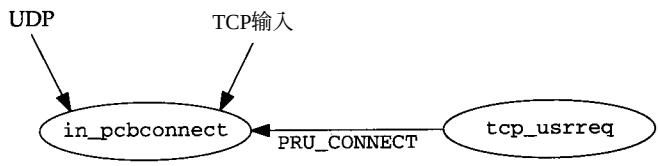


图7-1 Net/3中调用in_pcbconnect 的小结

在TCP输入和UDP中，对in_pcbconnect的调用是一样的，但是处理 TCP connect (PRU_CONNECT请求)时就要调用一个新的函数tcp_connect(图12-2和图12-3)，该函数又调用新的函数in_pcbladdr。另外，当T/TCP客户采用sendto或sendmsg隐式打开连接时，所产生的PRU_SEND或PRU_SEND_EOF请求也将调用tcp_connect。我们在图7-2中给出了这种新的调用方案。

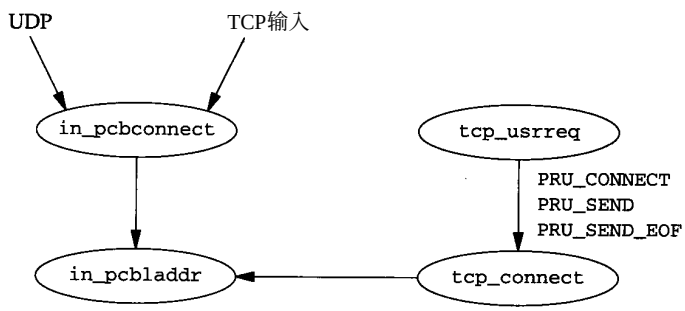


图7-2 in_pcbconnect 和in_pcbladdr 的新安排

7.2 in_pcbladdr函数

in_pcbladdr函数的第一部分如图7-3所示。这一部分仅仅给出了变量定义和头两行代码，它与卷2第590页的第138~139行完全相同。

in_pcb.c

```
136 int
137 in_pcbladdr(inp, nam, plocal_sin)
138 struct inpcb *inp;
139 struct mbuf *nam;
140 struct sockaddr_in **plocal_sin;
141 {
142     struct in_ifaddr *ia;
143     struct sockaddr_in *ifaddr;
144     struct sockaddr_in *sin = mtod(nam, struct sockaddr_in *);

145     if (nam->m_len != sizeof(*sin))
146         return (EINVAL);
```

in_pcb.c

图7-3 in_pcbladdr 函数：第一部分

136-140 头两个变量与in_pcbconnect中是一样的，第三个变量是一个指针的指针，用于返回本地地址。

这个函数的其余部分与卷2中图22-25和图22-26完全相同，与该卷图22-27的大部分也相同。卷2的图22-27中最后两行，即第593页，则用图7-4中的代码代替。

in_pcb.c

```
232         /*
233          * Don't call in_pcblookup here; return interface in
234          * plocal_sin and exit to caller, who will do the lookup.
235          */
236         *plocal_sin = &ia->ia_addr;

237     }
238     return (0);
239 }
```

in_pcb.c

图7-4 in_pcbladdr 函数：最后一部分

232-236 如果调用进程给定了通配符作为本地地址，指向 sockaddr_in结构的一个指针就会通过第三个变量返回。

基本上，in_pcbladdr所做的全部操作是进行差错检查，目标地址为 0.0.0.0或255.255.255.255这些特殊情况的处理，接着进行本地 IP地址的分配(如果调用进程还没有分配IP地址)。connect所需要的其他处理操作都在in_pcbconnect中实现。

7.3 in_pcbconnect函数

图7-5中给出了in_pcbconnect函数。这个函数调用了上一节所介绍的 in_pcbladdr，然后接下来就是卷2中图22-28中的代码。

1. 分配本地地址

255-259 如果调用进程还未将一个 IP地址绑定到其插口，则调用 in_pcbladdr函数计算出本地IP地址，然后通过ifaddr指针返回。

2. 验证插口对的唯一性

260-266 in_pcblookup验证插口对是唯一的。在TCP客户端调用connect(当客户端尚

未将一个本地端口或本地地址绑定到一个插口时)的一般情况下，本地端口号为 0，in_pcblookup就总是返回0，因为端口0是不会与任何一个现有的PCB匹配上的。

3. 如果还没有绑定，则绑定本地地址和本地端口

267-271 如果还没有本地地址和本地端口绑定到插口上，in_pcbbind要对这两者都进行分配。如果只是还没有本地地址绑定到插口，本地端口号已经为非零，则 in_pcbladdr返回的本地地址记录在PCB中。在本地端口号还是0时是不可能将一个本地地址绑定上去的，因为调用in_pcbbind函数绑定本地地址的同时会给插口分配一个临时使用的端口号。

272-273 外部地址和外部端口(in_pcbconnect的变量)记录在PCB中。

in_pcb.c

```
247 int
248 in_pcbconnect(inp, nam)
249 struct inpcb *inp;
250 struct mbuf *nam;
251 {
252     struct sockaddr_in *ifaddr;
253     struct sockaddr_in *sin = mtod(nam, struct sockaddr_in *);
254     int     error;

255     /*
256      * Call inner function to assign local interface address.
257      */
258     if (error = in_pcbladdr(inp, nam, &ifaddr))
259         return (error);

260     if (in_pcblookup(inp->inp_head,
261                      sin->sin_addr,
262                      sin->sin_port,
263                  inp->inp_laddr.s_addr ? inp->inp_laddr : ifaddr->sin_addr,
264                      inp->inp_lport,
265                      0))
266         return (EADDRINUSE);
267     if (inp->inp_laddr.s_addr == INADDR_ANY) {
268         if (inp->inp_lport == 0)
269             (void) in_pcbbind(inp, (struct mbuf *) 0);
270         inp->inp_laddr = ifaddr->sin_addr;
271     }
272     inp->inp_faddr = sin->sin_addr;
273     inp->inp_fport = sin->sin_port;
274     return (0);
275 }
```

in_pcb.c

图7-5 in_pcbconnect 函数

7.4 小结

T/TCP所作的修改是从in_pcbconnect函数中移去计算本地地址的所有代码，创建一个名为in_pcbladdr的新函数来完成这项任务。in_pcbconnect调用该函数，然后完成正常的连接处理过程。这将使处理 T/TCP客户连接请求 (或者用 connect显式建连，或者用sendto隐式地建连)时可以调用in_pcbladdr来计算本地地址。T/TCP客户的端处理则是复制了图7-5中的处理步骤，但即使前一次连接尚处于 TIME_WAIT状态，T/TCP也还是允许处理同一连接的后续请求。常规的 TCP是不允许这种情况发生的；这时图 7-5的in_pcbconnect将返回EADDRINUSE。

第8章　T/TCP的实现：TCP概要

8.1　概述

本章内容覆盖了T/TCP对TCP数据结构和函数所做的全局性修改。增加了两个全局变量：tcp_ccgen，即全局CC计数器，以及tcp_do_rfcl644，这是一个标志变量，说明是否选用CC选项。TCP的协议交换记录项也作了修改，以支持隐式的打开和关闭。另外还在TCP控制块中增加了4个变量。

对tcp_slowtimo函数也作了简单修改，以便能够测量每个连接的持续时间。给定一个连接的持续时间，如果持续时间短于MSL，则如4.4节所述，T/TCP将截断TIME_WAIT状态的保持时间。

8.2　代码介绍

T/TCP没有增加新的源文件，但是需要一些新的变量。

全局变量

图8-1中给出了T/TCP新增加的全局变量，在各个TCP函数中都会用到。

变　量	数据类型	说　明
tcp_ccgen	tcp_cc	要发送的下一个CC值
tcp_do_rfc1644	int	如果为真(默认)，发送CC或CCnew选项

图8-1　T/TCP新增的全局变量

在第3章我们给出了一些有关tcp_ccgen变量的例子。在6.5节中也提到了tcp_cc的数据类型是用typedef定义的，是无符号长整数。tcp_cc变量值为0，表示它尚未定义。tcp_ccgen变量总是这样存取的：

```
tp->cc_send = CC_INC(tcp_ccgen);
```

其中cc_send是TCP控制块的新字段(见后面的图8-3)。宏CC_INC是在<netinet/tcp_seq.h>中定义的：

```
#define CC_INC(c)    (++(c) == 0 ? ++(c) : (c))
```

由于这个值是在使用之前增加的，因此，tcp_ccgen要初始化为0，且它的第一个有用值为1。

为了按照模运算比较CC的值，定义了四个宏：CC_LT、CC_LEQ、CC_GT和CC_GEQ。这四个宏与卷2第649页定义的四个SEQ_*xx*宏完全一样。

变量tcp_do_rfcl644与卷2中介绍的变量tcp_do_rfcl323相似。如果tcp_do_rfcl644为0，TCP不会向对方发送CC或CCnew选项。

统计量

T/TCP新增了5个计数器，如图8-2所示。它们加在tcpstat结构中，卷2第638页对这个结构有介绍。

tcpstat字段	说　明
tcps_taook	TAO正确时接收到SYN
tcps_taofail	接收到带有CC选项的SYN，但TAO测试失败
tcps_badccecho	CCecho选项错误的SYN/ACK报文段
tcps_impliedack	隐含着对前一次连接的ACK的新SYN
tcps_ccdrop	因为无效的CC选项而丢弃的报文段

图8-2　在tcpstat 结构中新增的T/TCP统计量

程序netstat必须经修改才能打印这些新字段的值。

8.3 TCP的protosw结构

我们在第5章提到过，TCP的protosw记录项inetsw[2](卷2第641页)的pr_flags字段在T/TCP中作了修改。新的插口层标志 PR_IMPLOPCL必须包括在内，已有的标志PR_CONNREQUIRED和PR_WANTRCVD也必须包含在内。在sosend中，如果调用进程给出了一个目标地址，这个新的标志允许对一个未建连接的插口调用 sendto，并且如果指定了MSG_EOF标志，它所起的作用是发出一个PRU_SEND_EOF请求而不是PRU_SEND请求。

对protosw记录项所作的修改中有一个不是T/TCP所需的，即定义了tcp_sysctl函数作为pr_sysctl字段。这就允许系统管理员用前缀为net.inet.tcp的sysctl程序来修改能够控制TCP操作的一些变量值(卷2介绍的Net/3代码仅仅支持sysctl程序通过ip_sysctl、icmp_sysctl和udp_sysctl函数对IP、ICMP和UDP的一些变量进行控制)。在图12-6中给出了tcp_sysctl函数。

8.4 TCP控制块

TCP控制块中新增了四个变量，卷2第643~644页说明了TCP控制块的tcpcb结构。我们在图8-3中仅给出了新的字段，并非整个结构。

变　量	数据类型	说　明
t_duration	u_long	以500ms为单位的连接持续时间
t_maxopd	u_short	MSS加上通常选项的长度
cc_send	tcp_cc	发送给对等端的CC值
cc_recv	tcp_cc	从对等端中接收到的CC值

图8-3　T/TCP在tcpcb 结构中新增的字段

t_duration用于确定T/TCP是否可以截断TIME_WAIT状态的保持时间，见4.4节的讨论。当控制块创建时它的值为0，由tcp_slowtimo(8.6节)每过500 ms加1。

t_maxopd是为了代码的方便而设的。它的取值是已有t_maxseg字段的值加上TCP选项通常所占用的字节数。t_maxseg是每个报文段中的数据字节数。例如，在MTU 为1500字节的一个以太网上，如果时间戳和T/TCP都用上了，t_maxopd将为1460，t_maxseg则为1440。

它们之间的差值 20字节是由 12字节的时间戳选项加上 8字节的 CC选项(图2-4)造成的。t_maxopd和t_maxseg都是在tcp_mssrcvd函数中计算并记录的。

最后两个变量来自RFC 1644，在第2章给出了有关这三个变量的例子。如果一个连接的两端主机都用了CC选项，cc_recv的值将为非0。

在TCP控制块的t_flags字段中新定义了6个标志，如图8-4所示，是对卷2的图24-14中的9个标志的补充。

t_flags	说 明
TF_SENDSYN	发送SYN(隐藏的半同步连接状态标志)
TF_SENDFIN	发送FIN(隐藏的状态标志)
TF_SENDCCNEW	主动打开时发送CCnew选项而不是CC选项
TF_NOPUSH	不发送报文段，只清空发送缓存
TF_RCVD_CC	当对端在SYN中发送了CC选项时设置该标志
TF_REQ_CC	已经/将在SYN中申请CC选项

图8-4 T/TCP新增的t_flags 及其取值

不要把T/TCP中的两个标志TF_SENDFIN与TF_SENTFIN混淆，前者表示TCP需要发送FIN，而后者表示已经发出FIN。

> TF_SENDSYN和TF_SENDFIN这两个名字源于Bob Braden的“T/TCP的实现”。FreeBSD实现中将这两个名字改为TF_NEEDSYN和TF_NEEDFIN。我们选用了前面的名字，因为已经用新的标志来表示是否需要发送控制标志，如果选用后面的名字就会误解为需要接收SYN或FIN。然而请注意，因为选用了这样的名字，T/TCP的TF_SENDFIN标志和已有的TF_SENTFIN标志(表明TCP已经发出了FIN)仅有一个字符之差。

我们将在下一章的图9-3和图9-7中分别介绍TF_NOPUSH和TF_SENDCCNEW标志。

8.5 tcp_init函数

所有的T/TCP变量都不需要显式的初始化，因此卷 2中介绍的tcp_init函数没有变化。全局变量tcp_ccgen是没有初始化的外部变量，按照C语言的规则，它的默认值为0。这样做不会出错，因为在8.2节中定义的宏CC_INC是先对该变量加1，然后再用，因此在重启动后，tcp_ccgen的第一个有用值是1。

T/TCP也要求在重启动时将TAO缓存全部清空，由于在重启动时要初始化IP路由表，所以TAO缓存不需要专门处理。在路由表中每增加一个新的rtentry结构，rtrequest要将该结构初始化为0(卷2第489页)。这就意味着rmxp_tao结构中3个TAO变量的默认值都为0(图6-3)。为新主机创建新的TAO记录项时，T/TCP需要将tao_cc的值初始化为0。

8.6 tcp_slowtimo函数

两个TCP定时函数中有一个增加了一行：每次处理 500 ms定时器时，要对每个TCP控制块的t_duration字段执行加1操作，卷2第666页给出了tcp_slowtimo函数。下面这一行

```
tp->t_duration++;
```

加在这个图的第94~95行之间。这个变量的用途是测量每个连接的长度，以500ms为单位。如果连接持续时间短于MSL，TIME_WAIT状态的保持时间就可以截断，在4.4节中已经讨论过。

与这项优化有关的工作是在<netinet/timer.h>头文件中定义了下面这个常量：

```
#define TCPTV_TWTRUNC   8    /* RTO factor to truncate TIME_WAIT */
```

我们在图11-17和图11-19中将可以看到，如果T/TCP连接是主动关闭，并且t_duration的值小于TCPTV_MSL(60个500 ms，即30秒)，那么TIME_WAIT状态的保持时间就是当前重传超时(RTO)乘以TCPTV_TWTRUNC。在局域网环境中，RTO通常为3个500ms，即1.5秒，这将使TIME_WAIT状态的保持时间缩短到12秒。

8.7 小结

T/TCP新增了两个全局变量(tcp_ccgen和tcp_do_rfcl 644)、4个TCP控制块字段和5个TCP统计结构计数器。

tcp_slowtimo函数也作了修改，以500ms为时间单位计量每个TCP连接的持续时间。这个持续时间决定了T/TCP能否在主动关闭时截断TIME_WAIT状态的保持时间。

第9章　T/TCP的实现：TCP输出

9.1　概述

本章介绍为了支持T/TCP而对tcp_output函数所做的修改。在TCP中有许多程序段都要调用该函数来决定是否应该发出一个报文段，并且如果必要就发出一个。在　T/TCP中作了以下修改：

- 两个隐藏的状态标志可以打开TH_SYN和TH_FIN标志。
- T/TCP可以在SYN_SENT状态下发出多个报文段，但其前提是确知对等端也支持 T/TCP。
- 发送程序糊涂窗口避免机制必须考虑到新的 TF_NOPUSH标志，这个标志我们在3.6节中讨论过。
- 可以发出新的T/TCP选项(CC、CCnew和CCecho)。

9.2　tcp_output函数

9.2.1　新的自动变量

在tcp_output中说明了两个新的自动变量：

```
struct rmxp_tao *taop;
struct rmxp_tao  tao_noncached;
```

其中第一个变量是一个指针，指向相应对等端的 TAO缓存记录项。如果TAO缓存记录项不存在(这种情况不应该发生)，则taop指向tao_noncached，并且将这个结构初始化为0(这样它的tao_cc值就是未定义的)。

9.2.2　增加隐藏的状态标志

在tcp_output的开头，要从tcp_outflags向量中读取说明当前连接状态的TCP标志。图2-7给出了每个状态的相关标志。图 9-1中的代码用于在相应的隐藏状态标志处于开状态时，对TH_FIN标志和TH_SYN标志执行逻辑或。

tcp_output.c

```
71   again:
72     sendalot = 0;
73     off = tp->snd_nxt - tp->snd_una;
74     win = min(tp->snd_wnd, tp->snd_cwnd);

75     flags = tcp_outflags[tp->t_state];
76     /*
77      * Modify standard flags, adding SYN or FIN if requested by the
78      * hidden state flags.
79      */
```

图9-1　tcp_output　：增加隐藏状态标志

```
80      if (tp->t_flags & TF_SENDFIN)
81          flags |= TH_FIN;
82      if (tp->t_flags & TF_SENDSYN)
83          flags |= TH_SYN;
```
tcp_output.c

图9-1 (续)

这些代码位于卷2第681~682页。

9.2.3 在SYN_SENT状态不要重传SYN

图9-2中的程序读取对等端的TAO缓存内容，并且查看是否已经发出了SYN。这段代码位于卷2中图26-3的开头。

1. 读取TAO缓存记录项

117-119 读取对等端的TAO缓存内容，如果不存在，则改用自动变量tao_noncached，其初始值置为0。

> 如果使用了全0的记录项，它的值永远不变。这样，tao_noncached结构就可以静态分配并初始化为0，而不必用bzero将其设置为0。

2. 检查客户请求是否超过MSS

121-133 如果状态标志表明需要发送SYN，并且如果已经发出SYN，那么TH_SYN标志就要关闭。当一个应用程序用T/TCP向对等端发送多倍MSS数量的数据时可能发生这种情况 (见3.6节)。如果对等端支持T/TCP协议，这时可以分多个报文段发送，但只有第一个报文段应该设置SYN标志。如果我们不能确定对等端是否支持T/TCP(tao_ccsent值为0)，这时我们必须在三次握手过程完成以后才可以发送多个报文段。

tcp_output.c
```
116     len = min(so->so_snd.sb_cc, win) - off;

117     if ((taop = tcp_gettaocache(tp->t_inpcb)) == NULL) {
118         taop = &tao_noncached;
119         bzero(taop, sizeof(*taop));
120     }
121     /*
122      * Turn off SYN bit if it has already been sent.
123      * Also, if the segment contains data, and if in the SYN-SENT state,
124      * and if we don't know that foreign host supports TAO, suppress
125      * sending segment.
126      */
127     if ((flags & TH_SYN) && SEQ_GT(tp->snd_nxt, tp->snd_una)) {
128         flags &= ~TH_SYN;
129         off--, len++;
130         if (len > 0 && tp->t_state == TCPS_SYN_SENT &&
131             taop->tao_ccsent == 0)
132             return (0);
133     }
134     if (len < 0) {
```
tcp_output.c

图9-2 tcp_output ：在SYN_SENT状态不重传SYN

9.2.4 发送器的糊涂窗口避免机制

发送器的糊涂窗口避免机制有两处作了修改 (卷2第715页)，如图9-3所示

tcp_output.c

```
168     if (len) {
169         if (len == tp->t_maxseg)
170             goto send;
171         if ((idle || tp->t_flags & TF_NODELAY) &&
172             (tp->t_flags & TF_NOPUSH) == 0 &&
173             len + off >= so->so_snd.sb_cc)
174             goto send;
175         if (tp->t_force)
176             goto send;
177         if (len >= tp->max_sndwnd / 2 && tp->max_sndwnd > 0)
178             goto send;
179         if (SEQ_LT(tp->snd_nxt, tp->snd_max))
180             goto send;
181     }
```

tcp_output.c

图9-3 tcp_output ：糊涂窗口避免机制中，确定是否发送报文段

1. 发送最大报文段

169-170 如果允许，就发出最大报文段。

2. 允许应用程序关闭隐式推送

171-174 BSD实现中是这样处理的：如果不是正在等待对等端的ACK(idle值为真)，或者如果Nagle算法禁用(TF_NODELAY值为真)，并且TCP正在清空发送缓存，那么它总是发出一个报文段。有时称这种方式为隐式推送，因为除非受Nagle算法所限，否则应用程序每写一次都会导致一个报文段发送出去。T/TCP提供了一个新的插口选项，可以使BSD的隐式推送失效，这个选项就是TCP_NOPUSH，最后变成了TF_NOPUSH标志。我们在3.6节研究过有关这个标志的一个例子。在这段代码中，我们看到了只有以下三个条件同时为真，报文段才能发出：

1) 并不在等待ACK(idle值为真)，或者Nagle算法已经禁用(TF_NODELAY值为真)；

2) TCP_NOPUSH插口选项没有使用(默认值)；

3) TCP正在清空发送缓存(即所有未发的数据可以在一个报文段中发出)。

3. 检查接收窗口是否打开了至少一半

177-178 在常规的TCP中，整个这部分代码段不会因为收到第一个SYN而执行，因为这时len应该是0。但是在T/TCP中，很有可能在接收到另一端发来的SYN之前就发送数据。这就意味着需要根据max_sndwnd是否大于0来检测接收窗口是否已经打开了一半。这个变量是对等端通告的最大窗口，但是在从对等端收到通告前，它一直是0(即一直到收到对等端的SYN)。

4. 重传定时器到时发送

179-180 重传定时器到时后，snd_nxt小于snd_max。

9.2.5 有RST或SYN标志时强制发送报文段

卷2第688页的179~180行代码在SYN标志或RST标志打开时总是要发送一个报文段。这两行要用图9-4中的代码替代。

tcp_output.c

```
207     if ((flags & TH_RST) ||
208         ((flags & TH_SYN) && (tp->t_flags & TF_SENDSYN) == 0))
209         goto send;
```

tcp_output.c

图9-4 tcp_output ：检查RST和SYN标志，确定是否发送报文段

207-209 如果RST标志打开了，就总要发出一个报文段。如果 SYN标志打开了，则只有在相应的隐藏状态标志关闭时才会发出报文段。加上这项限制的理由可以看图 2-7。在最后5个服务器加星状态(半同步状态)下，TF_SENDSYN标志是打开的，这就会使图 9-1中的SYN标志被打开。在tcp_output中执行这项测试的目的是只在 SYN_SENT、SYN_RCVD、SYN_SENT*和SYN_RCVD*状态下才发出报文段。

9.2.6 发送MSS选项

这一小段代码(卷2第697页)有一个小小的变化。Net/3中的函数tcp_mss(有两个参量)改为tcp_msssend(仅仅以tp为参量)。这是因为我们需要把计算 MSS并发送与处理收到的MSS选项区分开来。Net/3中的tcp_mss函数同时完成这两项处理；在 T/TCP中，我们则用两个不同的函数来完成，它们是tcp_msssend和tcp_mssrcvd，我们将在下一章讨论这两个函数。

9.2.7 是否发送时间戳选项

卷2第698页，如果以下三个条件都成立，就发出时间戳选项： (1)TCP配置中要求使用时间戳选项；(2)正在构造的报文段不包括 RST标志；以及(3)要么这是一次主动打开或者 TCP已经从另一端接收到了一个时间戳(TF_RCVD_TSTMP)。对主动打开的测试只要查看 SYN标志是否打开以及ACK标志是否关闭即可。完成这三项测试的 T/TCP代码如图9-5所示。

tcp_output.c

```
283     /*
284      * Send a timestamp and echo-reply if this is a SYN and our side
285      * wants to use timestamps (TF_REQ_TSTMP is set) or both our side
286      * and our peer have sent timestamps in our SYN's.
287      */
288     if ((tp->t_flags & (TF_REQ_TSTMP | TF_NOOPT)) == TF_REQ_TSTMP &&
289         (flags & TH_RST) == 0 &&
290         ((flags & TH_ACK) == 0 ||
291          (tp->t_flags & TF_RCVD_TSTMP))) {
```

tcp_output.c

图9-5 tcp_output ：是否发送时间戳选项？

283-291 因为我们希望从客户端到服务器方向上发送的所有第一个报文段都携带时间戳选项(在多报文段请求的情况下，如图 3-9所示)，而不仅仅只是含有 SYN的第一个报文段，所以在T/TCP中第三项测试的前一半有所改变。对所有初始报文段的新测试项都是在没有 ACK标志的情况下进行的。

9.2.8 发送T/TCP的CC选项

是否发送三个新 CC选项之一的测试是看 TF_REQ_CC标志是否打开 (如果全局变量tcp_do_rfc1644非零，该标志由tcp_newtcpcb激活)、TF_NOOPT标志是否关闭以及RST标志是否关闭。发送哪个 CC选项则取决于输出报文段中 SYN标志和ACK标志的状态。这样就有四种可能的组合，前两种如图 9-6所示(这段代码在卷2第698页的第268~269行)。

TF_NOOPT标志是由新增的 TCP_NOOPT插口选项控制的。该插口选项出现在 Thomas Skibo写的 RFC 1323 的代码中(见12.7节)。在卷2中曾指出，这个标志(不是指插口选项)自从4.2BSD以后就已经在伯克利源代码中存在了，但通常无法将其打开。

如果设置了这个选项，TCP就不用随SYN发送任何选项。新增这个选项用来处理TCP实现中的不一致性，因为这些实现不能忽略未知的TCP选项(自从RFC 1323修改以后，增加了两个新的TCP选项)。

T/TCP所作的修改并没有改变确定是否发送MSS选项的那段代码(卷2第697页)。如果TF_NOOPT标志没有设置，这段代码就不发送MSS选项。但是Bob Braden在他的RFC 1323代码中指出，没有真正的理由需要阻止发送MSS选项。MSS选项是RFC 793规范中的一部分内容。

tcp_output.c

```
299     /*
300      * Send CC-family options if our side wants to use them (TF_REQ_CC),
301      * options are allowed (!TF_NOOPT) and it's not a RST.
302      */
303     if ((tp->t_flags & (TF_REQ_CC | TF_NOOPT)) == TF_REQ_CC &&
304         (flags & TH_RST) == 0) {
305         switch (flags & (TH_SYN | TH_ACK)) {
306             /*
307              * This is a normal ACK (no SYN);
308              * send CC if we received CC from our peer.
309              */
310         case TH_ACK:
311             if (!(tp->t_flags & TF_RCVD_CC))
312                 break;
313             /* FALLTHROUGH */
314             /*
315              * We can only get here in T/TCP's SYN_SENT* state, when
316              * we're sending a non-SYN segment without waiting for
317              * the ACK of our SYN.  A check earlier in this function
318              * assures that we only do this if our peer understands T/TCP.
319              */
320         case 0:
321             opt[optlen++] = TCPOPT_NOP;
322             opt[optlen++] = TCPOPT_NOP;
323             opt[optlen++] = TCPOPT_CC;
324             opt[optlen++] = TCPOLEN_CC;
325             *(u_int32_t *) & opt[optlen] = htonl(tp->cc_send);
326             optlen += 4;
327             break;
```

tcp_output.c

图9-6 tcp_output ：发送一个CC选项，第一部分

1. SYN关闭，ACK打开

310-313 如果SYN标志关闭，但ACK标志打开，这就是常规的ACK(即连接已经建立)。只有从对等端收到一个CC选项以后，才会发送CC选项。

2. SYN关闭，ACK关闭

314-320 只有在SYN_SENT*状态下，即在连接建立以前就发送了一个非SYN报文段时，这两个标志才会同时关闭。也就是说，在客户一次发送了多倍MSS数量的数据时才会这样。图9-2中的代码能够确保只有在对等端也支持T/TCP时才会进入这种状态。这种情况下就要发送CC选项。

3. 构造CC选项

321-327 在构造CC选项时，要先加上两个空字符。该连接cc_send的值就作为CC选项的

内容发送出去。

SYN标志和ACK标志状态组合的剩余两种情况如图9-7所示。

tcp_output.c

```
328             /*
329              * This is our initial SYN (i.e., client active open).
330              * Check whether to send CC or CCnew.
331              */
332         case TH_SYN:
333             opt[optlen++] = TCPOPT_NOP;
334             opt[optlen++] = TCPOPT_NOP;
335             opt[optlen++] =
336                 (tp->t_flags & TF_SENDCCNEW) ? TCPOPT_CCNEW : TCPOPT_CC;
337             opt[optlen++] = TCPOLEN_CC;
338             *(u_int32_t *) & opt[optlen] = htonl(tp->cc_send);
339             optlen += 4;
340             break;
                    .
341             /*
342              * This is a SYN, ACK (server response to client active open).
343              * Send CC and CCecho if we received CC or CCnew from peer.
344              */
345         case (TH_SYN | TH_ACK):
346             if (tp->t_flags & TF_RCVD_CC) {
347                 opt[optlen++] = TCPOPT_NOP;
348                 opt[optlen++] = TCPOPT_NOP;
349                 opt[optlen++] = TCPOPT_CC;
350                 opt[optlen++] = TCPOLEN_CC;
351                 *(u_int32_t *) & opt[optlen] = htonl(tp->cc_send);
352                 optlen += 4;

353                 opt[optlen++] = TCPOPT_NOP;
354                 opt[optlen++] = TCPOPT_NOP;
355                 opt[optlen++] = TCPOPT_CCECHO;
356                 opt[optlen++] = TCPOLEN_CC;
357                 *(u_int32_t *) & opt[optlen] = htonl(tp->cc_recv);
358                 optlen += 4;
359             }
360             break;
361         }
362     }
363     hdrlen += optlen;
```

tcp_output.c

图9-7 tcp_output ：发送CC选项之一，第二部分

4. SYN打开，ACK关闭(客户主动打开)

328-340 当客户执行主动打开时，SYN打开且ACK关闭。如果应该发送CCnew选项而不是CC选项，图12-3中的代码完成TF_SENDCCNEW标志的设置，同时也设置cc_send值。

5. SYN打开，ACK打开(服务器响应客户端的SYN)

341-360 当SYN标志和ACK标志同时处于打开状态时，这就是服务器对对等端的主动打开作出了响应。如果对等端发送了CC或CCnew选项之一(设置了TF_RCVD_CC)，这时我们要向对等端发送CC选项(cc_send)和对对等端CC值(cc_recv)的CCecho。

6. 根据TCP选项调整TCP首部长度

363 所有的TCP选项都加长了TCP首部的长度。

9.2.9 根据TCP选项调整数据长度

t_maxopd是tcpcb结构的新字段且是最大数据长度，并且也是常规 TCP报文段的选项。因为窗口宽度选项和CCecho选项只在SYN报文段中出现，因此在SYN报文段中的选项(见图2-2和图2-3)很有可能会比非SYN报文段的选项(见图2-4)需要更多的字节空间。图9-8中的代码根据TCP选项的大小调整发送报文段中的数据量。这段代码用于替代卷 2第698页的第270~277行。

tcp_output.c

```
364     /*
365      * Adjust data length if insertion of options will
366      * bump the packet length beyond the t_maxopd length.
367      * Clear the FIN bit because we cut off the tail of
368      * the segment.
369      */
370     if (len + optlen > tp->t_maxopd) {
371         /*
372          * If there is still more to send, don't close the connection.
373          */
374         flags &= ~TH_FIN;
375         len = tp->t_maxopd - optlen;
376         sendalot = 1;
377     }
```

tcp_output.c

图9-8 tcp_output ：根据TCP选项的大小调整发送数据量

364-377 如果数据长度(len)加上选项长度超过了t_maxopd，发送的数据量就要缩减，FIN标志关闭(如果它原来是开状态)，且sendalot打开(在当前报文段发出后强迫再次执行tcp_output循环)。

这些代码并不是T/TCP专有的。它应该对任何一个既携带数据又有 TCP选项(例如：RFC 1323时间戳选项)的报文段执行。

9.3 小结

T/TCP在原本500行的tcp_output函数上增加了大约100行代码。增加的大部分代码都与发送新增的T/TCP选项CC、CCnew和CCecho等有关。

另外，如果对等端支持T/TCP，T/TCP的tcp_output函数可以在SYN_SENT状态下发送多个报文段。

第10章 T/TCP实现：TCP函数

10.1 概述

本章包括了T/TCP作过修改的各个TCP函数。也就是说，tcp_output(前一章)、tcp_input，和tcp_usrreq(后两章)以外的所有函数。本章定义了两个新的函数，tcp_rtlookup和tcp_gettaocache，用于在TAO缓存中查找记录项。

tcp_close函数修改以后，当使用T/TCP的连接关闭时，可以在路由表中记录往返时间估计值(平滑的平均值和平均偏差估计)。常规协议只在连接上传送了至少16个满数据报文段后才记录。然而，T/TCP通常只发送少量数据，但与同一对等端之间的这些不同连接的估计值应该保留下来。

T/TCP中对MSS选项的处理也有所改变。有一部分改变是为了在Net/3中清理过载的tcp_mss函数，这样就把它分成了一个计算MSS以便发送的函数(tcp_mssend)和另一个处理接收到的MSS选项的函数(tcp_mssrcvd)。T/TCP同时也将从对等端收到的最新MSS值保存到TAO缓存记录项中。在接收到服务器的SYN和最新的MSS之前，如果要随SYN发送数据，T/TCP就用这个记录来初始化发送MSS。

Net/3中的tcp_dooptions函数修改以后能够识别三个新的T/TCP选项：CC、CCnew和CCecho。

10.2 tcp_newtcpcb函数

用PRU_ATTACH请求创建新的插口时要调用该函数。图10-1中的五行代码用来代替卷2第667页的第177~178行。

tcp_subr.c

```
180     tp->t_maxseg = tp->t_maxopd = tcp_mssdflt;

181     if (tcp_do_rfc1323)
182         tp->t_flags = (TF_REQ_SCALE | TF_REQ_TSTMP);
183     if (tcp_do_rfc1644)
184         tp->t_flags |= TF_REQ_CC;
```

tcp_subr.c

图10-1 tcp_newtcpcb 函数：T/TCP所做的修改

180 在前面图8-3有关的介绍中提到过，t_maxopd是每个报文段中可以发送的TCP选项加上数据的最大字节数。它和t_maxseg的默认值均为512(tcp_mssdflt)。由于这两个值相等，表明报文段中不能再有TCP选项。在后面的图10-13和图10-14中，如果时间戳选项或者CC选项(或者两者同时)需要在报文段中发送，就要减小t_maxseg的值。

183-184 如果全局变量tcp_do_rfc1644非零(它的默认值为1)，且设置了TF_REQ_CC标志，这将使tcp_output伴随SYN发出CC或CCnew选项(图9-6)。

10.3 tcp_rtlookup函数

tcp_mss(卷2第717~718页)执行的第一项操作是读取为该连接所缓存的路由 (存储在

route.h

```
46 struct route {
47     struct rtentry *ro_rt;       /* pointer to struct with information */
48     struct sockaddr ro_dst;      /* destination of this route */
49 };
```

route.h

图10-2 route 结构

tcp_subr.c

```
432 struct rtentry *
433 tcp_rtlookup(inp)
434 struct inpcb *inp;
435 {
436     struct route *ro;
437     struct rtentry *rt;

438     ro = &inp->inp_route;
439     rt = ro->ro_rt;
440     if (rt == NULL) {
441         /* No route yet, so try to acquire one */
442         if (inp->inp_faddr.s_addr != INADDR_ANY) {
443             ro->ro_dst.sa_family = AF_INET;
444             ro->ro_dst.sa_len = sizeof(ro->ro_dst);
445             ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
446                 inp->inp_faddr;
447             rtalloc(ro);
448             rt = ro->ro_rt;
449         }
450     }
451     return (rt);
452 }
```

tcp_subr.c

图10-3 tcp_rtlookup 函数

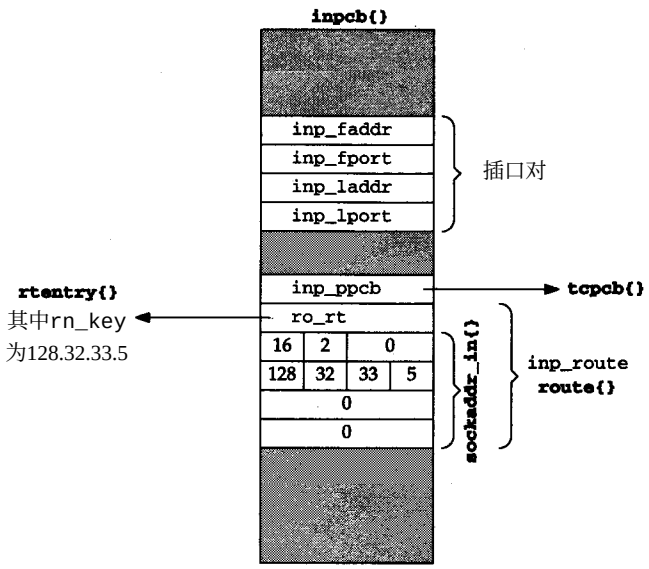


图10-4 在Internet PCB中缓存的路由全貌

Internet PCB的inp_route字段中)，如果该路由还没有缓存过，则调用 rtalloc查找路由。现在这项操作安排在另一个独立的函数 tcp_rtlookup中实现，我们将在图 10-3中介绍。这样做是因为连接的路由表记录项中包括有 TAO信息，T/TCP需要更经常地执行这一项操作。

438-452 如果这个连接的路由还没有在缓存中记录， rtalloc就计算出路由。但仅仅当PCB中的外部地址非 0时才能计算路由。在调用 rtalloc之前，要先填写 route结构中的sockaddr_in结构。

图10-2给出了route结构，其中的一个结构在每个 Internet PCB中都有。

图10-4给出了这个结构的全貌，图中假定外部地址为 128.32.33.5。

10.4 tcp_gettaocache函数

一个给定主机的 TAO信息保存在该主机的路由表记录项中，确切地说，是在 rt_metrics结构的rmx_filler字段中(见6.5节)。图10-5所示的函数tcp_gettaocache返回指向该主机TAO缓存的指针。

tcp_subr.c

```
458 struct rmxp_tao *
459 tcp_gettaocache(inp)
460 struct inpcb *inp;
461 {
462     struct rtentry *rt = tcp_rtlookup(inp);

463     /* Make sure this is a host route and is up. */
464     if (rt == NULL ||
465         (rt->rt_flags & (RTF_UP | RTF_HOST)) != (RTF_UP | RTF_HOST))
466         return (NULL);

467     return (rmx_taop(rt->rt_rmx));
468 }
```

tcp_subr.c

图10-5 tcp_gettaocache 函数

460-468 tcp_rtlookup返回的指针指向外部主机的 rtentry结构。如果查找成功，并且RTF_UP和RTF_HOST标志均打开了，则宏 rmx_taop(见图6-3)返回的指针指向rmxp_tao结构。

10.5 重传超时间隔的计算

Net/3的TCP要测量数据报文段往返时间、跟踪平滑的 RTT估计器(*srtt*)和平滑的平均偏差估计器(*rttvar*)，并据此计算重传超时间隔(RTO)。平均偏差是标准差的良好逼近，比较容易计算，因为与标准差不一样，平均偏差不需要做平方根运算。文献 [Jacobson 1988]给出了RTT测量的其他细节，并导出以下的计算公式：

$$delta = data - srtt$$

$$srtt \leftarrow srtt + g \times delta$$

$$rttvar \leftarrow rttvar + h(|delta| - rttvar)$$

$$RTO = srtt + 4 \times rttvar$$

其中，*delta*是刚刚得到的往返时间测量值 (*data*)与当前的平滑的RTT估计器(*srtt*)之差；*g*是应用于RTT估计器的增益，等于 1/8；*h*是应用于平均偏差估计器的增益，等于 1/4。在RTO计算

中的两个增益和乘数4特意取为2的乘幂，因此可以通过移位操作来代替乘除运算。卷 2的第25章给出了如何用定点整数来保存这些值的有关细节。

在常规的TCP连接中，在计算*srtt*和*rttvar*这两个估计器时，通常要对多个 RTT取样，对于图1-9中的给定最小TCP连接来说，至少要有两个样本。而且，在一定条件下， Net/3将对相同主机之间的多个连接运用这两个估计器。这是 `tcp_close`函数实现的，在一个连接关闭时，如果有关对等端的路由表记录项不是默认路由，并且至少得到了 16个RTT样值。估计的结果存储在路由表记录项中`rt_metrics`结构的`rmx_rtt`和`rmx_rttvar`字段中。新连接建立时，`tcp_mssrcvd`(见10.8节)从路由表记录项中取出这两个值作为 *srtt*和*rttvar*这两个估计器的初始值。

T/TCP中出现的问题是，一个最小连接只有一个 RTT测量值，而且少于16个样值是很正常的，因此在两个对等端之间相继建立拆除的 T/TCP连接对上述测量和估计一点贡献也没有。这就意味着在T/TCP中，第一个报文段发出去时根本就不知道 RTO的取值应该是多少。卷 2的25.8节讨论了`tcp_newtcpcb`执行初始化时是怎样确定第一个RTO应该是6秒的。

让`tcp_close`在即使只收集到少于 16个样值也存储对T/TCP连接的平滑估计结果并不难(在10.6节中我们会看到为此所做的修改)，但问题是：如何将新估计值与以前的估计值进行归并？不幸的是，这仍然还是一个正在研究的问题 [Paxson 1995a]。

为了理解各种不同的可能性，请考虑图 10-6中的情况。从作者的一台主机上通过 Internet向另一台主机上的回显服务器发送 100个400字节长的UDP数据报(在一个工作日的下午，通常是Internet上最为拥挤的时候)。93个数据报有回显返回(还有7个不知在Internet的哪些地方丢失了)，在图10-6中给出了前91个数据报。样值是在30分钟的时间内采集到的，前后数据报之间的时间间隔是在0~30秒之间均匀分布的随机数。实际的RTT是在客户主机上运行Tcpdump得到的。黑点就是测量得到的 RTT。另外的三条实线(从上至下依次是RTO、*srtt*和*rttvar*)是运用本

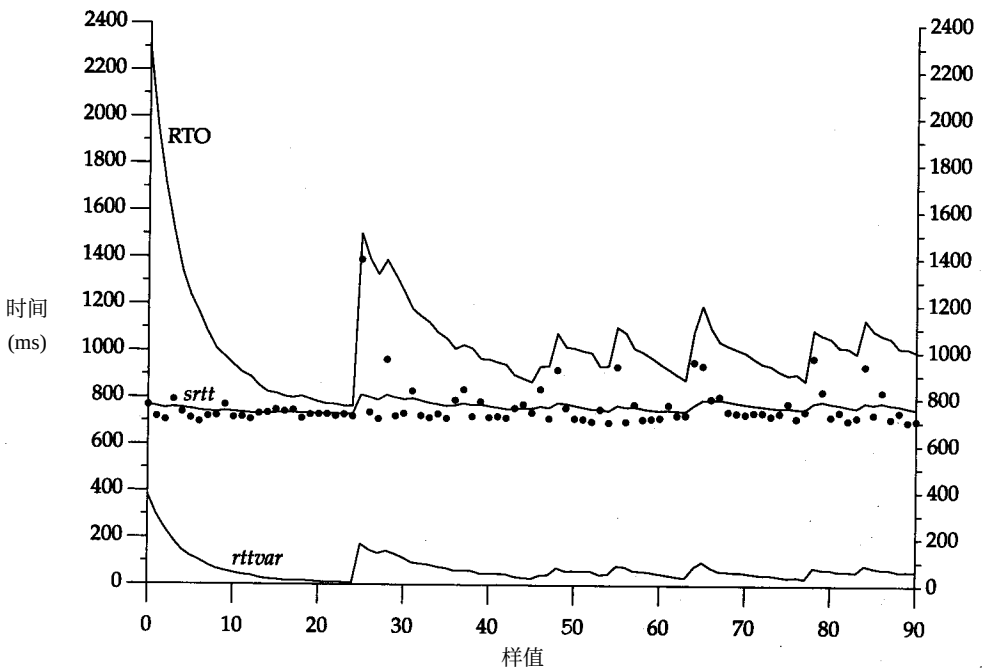


图10-6 RTT测量和对应的RTO、*srtt*和*rttvar*

节开头的公式从测得的RTT计算出来的。计算是用浮点算术完成的，而不是 Net/3中实际所用的定点整数方法。图上所示的RTO就是从相应的数据点计算出来的值。也就是说，第一个数据点(大约2200 ms)的RTO是从第一个数据点计算得来的，将用作下一个报文段发送时的 RTO。

尽管所测得的RTT值平均都在800 ms以下(作者的客户系统是通过拨号线上的 PPP连接到Internet上的，穿越整个国家才能到达服务器)，第26个样值的RTT几乎达到1400 ms，此后有少量的一些点在1000 ms左右。[Jacobson 1994]指出，“ 只要是有竞争的连接共享一条路由，瞬间RTT波动达到2倍最小值是完全正常的(它们仅仅表示另外一个连接的开始或丢失后重新开始)，因此，RTO小于2×RTT从来就不会是合理的 ”。

当估计器有新值存储到路由表记录项中时，必须做出判断，对应于已经过去的历史，有多少信息是新的。这样，计算公式就为：

$$savesrtt=g\times savesrtt+(1-g)\times srtt$$

$$saverttvar=g\times saverttvar+(1-g)\times rttvar$$

这是一个低通滤波器，其中 g是取值在0~1之间的过滤增益常量，*savesrtt*和*saverttvar*是存储在路由表记录项中的数值。当 Net/3用这些公式更新路由表记录时 (当一个连接关闭，并假定得到了16个样值)，它采用的增益是0.5：存储在路由表中的值有一半是路由表中的旧值，另有一半是当前估计的值。 Bob Braden的 T/TCP代码中取增益为0.75。

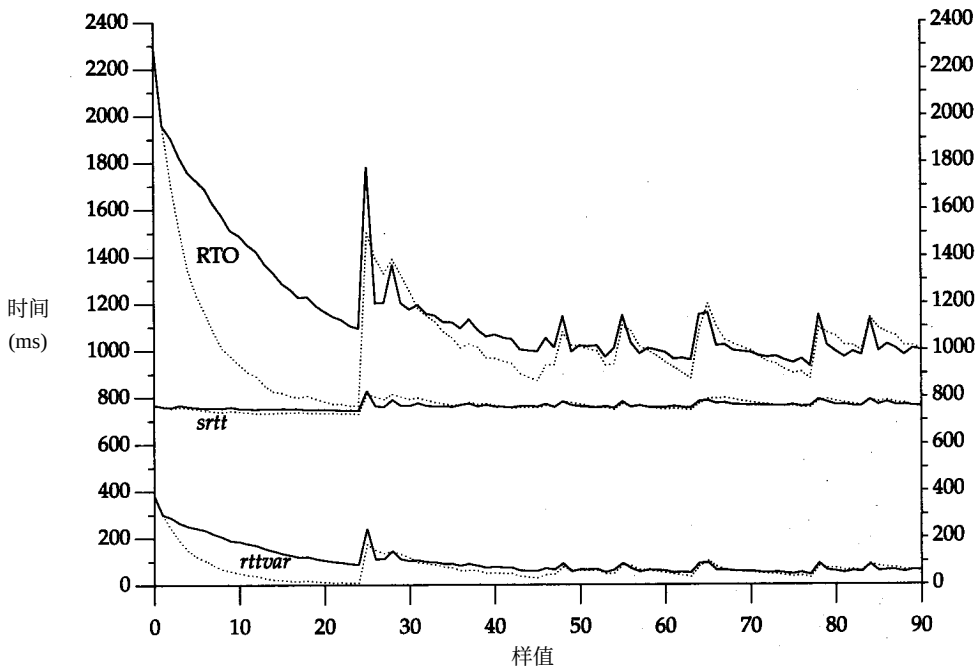


图10-7 TCP平滑与T/TCP平滑的比较

图10-7给出了从图10-6中的数据用常规TCP计算方法算出的结果与用滤波器增益 0.75平滑的计算结果之间的比较。图中的三条虚线就是图 10-6中的三个变量(RTO在最上方，*srtt*在中间，*rttvar*在底部)。三条实线则是假定每一个数据点都是一个独立 T/TCP连接(每一个连接有一个RTT测量值)所对应的变量，并且采用滤波增益 0.75进行了平滑。要知道有这样的差别：虚线对应的是一个TCP连接在30分钟内的91个RTT样本；而实线对应的则是在同样的 30分钟内91

个独立的T/TCP连接，每个连接有一次RTT测量。实线同时还是91个连接的所有相继两个估计值归并后记录到两个路由度量值中的。

代表*srtt*的实线和虚线差别不大，但是代表*rttvar*的实线和虚线之间就有明显的差别。*rttvar*的实线(T/TCP情况)取值通常大于虚线(单个TCP连接)，使T/TCP的重传超时间隔可以取更大的值。

还有其他因素也在影响T/TCP中的RTT测量。从客户端来看，所测得的RTT通常包括服务器的处理时间或者服务器的延迟ACK定时值，因为服务器的应答通常会延迟到这些事件发生后才给出。在Net/3中，延迟ACK的定时器值是每200ms到时一次，而RTT测量的时间单位为500 ms，因此应答时延不会是一个大的因素。而且T/TCP报文段的处理常常会在TCP输入处理中遭遇慢通道(例如，报文段常常不被用于首部预测)，会加大测得的RTT值(然而快通道与慢通道的差别相对于200 ms的延迟ACK定时器值来说很可能是可以忽略的)。最后，如果存储在路由表中的值“过时”了(就是说，其最后一次更新是在一个小时以前)，在当前事务完成以后，或许应该用当前的测量值直接替换路由表中的值，而不是用新的测量值与已有的测量值归并。

如RFC 1644中所指出的，需要对TCP中的动态特性作更多的研究，特别是T/TCP，以及RTT估计。

10.6 tcp_close函数

tcp_close的唯一改变是要为T/TCP事务记录RTT估计值，即使还没有凑足16个样值。我们在前一节中已经叙述了这样做的原因。图10-8给出了代码。

tcp_subr.c

```
252     if (SEQ_LT(tp->iss + so->so_snd.sb_hiwat * 16, tp->snd_max) &&
253         (rt = inp->inp_route.ro_rt) &&
254       ((struct sockaddr_in *) rt_key(rt))->sin_addr.s_addr != INADDR_ANY) {
                        /* pp. 895-896 of Volume 2 */
304     } else if (tp->cc_recv != 0 &&
305                (rt = inp->inp_route.ro_rt) &&
306       ((struct sockaddr_in *) rt_key(rt))->sin_addr.s_addr != INADDR_ANY) {
307         /*
308          * For transactions we need to keep track of srtt and rttvar
309          * even if we don't have 'enough' data for above.
310          */
311         u_long  i;
312         if ((rt->rt_rmx.rmx_locks & RTV_RTT) == 0) {
313             i = tp->t_srtt *
314                 (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));
315             if (rt->rt_rmx.rmx_rtt && i)
316                 /*
317                  * Filter this update to 3/4 the old plus
318                  * 1/4 the new values, converting scale.
319                  */
320                 rt->rt_rmx.rmx_rtt =
321                     (3 * rt->rt_rmx.rmx_rtt + i) / 4;
322             else
323                 rt->rt_rmx.rmx_rtt = i;
324         }
```

图10-8 tcp_close 函数：为T/TCP事务保存RTT估计值

```
325         if ((rt->rt_rmx.rmx_locks & RTV_RTTVAR) == 0) {
326             i = tp->t_rttvar *
327                 (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
328             if (rt->rt_rmx.rmx_rttvar && i)
329                 rt->rt_rmx.rmx_rttvar =
330                     (3 * rt->rt_rmx.rmx_rttvar + i) / 4;
331             else
332                 rt->rt_rmx.rmx_rttvar = i;
333         }
334     }
```

tcp_subr.c

图10-8 (续)

1. 只对T/TCP事务进行更新

304-311 只有在连接中使用了T/TCP(cc_recv非0)、有一路由表记录项存在及不是默认路由时才更新路由表记录项中的度量值。而且，只有当两个 RTT估计值没有加锁(RTV_RTT和RTV_RTTVAR位)时才更新。

2. 更新RTT

312-324 t_srtt是以500 ms×8为时间单位保存的，rmx_rtt则以μs为单位保存。这样，t_srtt就要乘1 000 000(RTM_RTTUNIT)除2(时间单位/秒)再乘8。如果rmx_rtt已经有值，新记录值就是旧值的四分之三加上新值的四分之一。这就是取滤波增益为 0.75，我们在前一节已讨论过。否则，直接将新值保存到rmx_rtt中。

3. 更新平均偏差

325-334 对平均偏差估计值应用同样的算法。它也以ms为单位保存，需要将t_rttvar中的单位时间×4。

10.7 tcp_msssend函数

在Net/3中，有一个函数tcp_mss(卷2的27.5节)，在处理MSS选项时tcp_input要调用它，在需要发送 MSS选项时 tcp_output也要调用它。在 T/TCP中，这个函数改名为tcp_mssrcvd，在执行隐式连接建立时，收到 SYN后，tcp_input要调用它(在后面的图10-18中，确定是否需要在SYN中包含MSS选项)，以及PRU_SEND和PRU_SEND_EOF请求要调用它(见图12-4)。有一个新的函数tcp_msssend，如图10-9所示，只有当发出了MSS选项时，才会被tcp_output调用。

tcp_input.c

```
1911 int
1912 tcp_msssend(tp)
1913 struct tcpcb *tp;
1914 {
1915     struct rtentry *rt;
1916     extern int tcp_mssdflt;

1917     rt = tcp_rtlookup(tp->t_inpcb);
1918     if (rt == NULL)
1919         return (tcp_mssdflt);

1920     /*
1921      * If there's an mtu associated with the route, use it,
```

图10-9 tcp_msssend 函数：返回MSS值，并在MSS选项中发出

```
1922        * else use the outgoing interface mtu.
1923        */
1924       if (rt->rt_rmx.rmx_mtu)
1925           return (rt->rt_rmx.rmx_mtu - sizeof(struct tcpiphdr));

1926       return (rt->rt_ifp->if_mtu - sizeof(struct tcpiphdr));
1927   }
```

tcp_input.c

图10-9 (续)

1. 读取路由表记录项

1917-1919 为每一个对等主机搜索路由表，如果没有找到记录项，则返回默认值512(tcp_mssdflt)。除非对等主机不可达，否则总是可以查找到一个路由表记录项的。

2. 返回MSS

1920-1926 如果路由表有一个关联的MTU(rt_metrics结构中的rmx_mtu字段，系统管理员可以用route程序设置)，就返回该值。否则，返回值就取输出接口的MTU减去40(例如，以太网上是1460)。因为路由已经由tcp_rtlookup确定，输出接口也是已知的。

在路由表中存储MTU度量的另一个来源是利用路由MTU发现过程(卷1的24.2节)，尽管Net/3中还不支持这种方法。

这个函数不同于通常的BSD做法。如果对等端是非本地主机(由in_localaddr函数决定)，而且rmx_mtu度量值为0，则Net/3代码(卷2第719页)中总是将MSS取为512(tcp_mssdflt)。

MSS选项的目的是告诉另一端，该选项发送者准备接收多大报文段。 RFC 793中指出，MSS选项“用于交流发送这个报文段的TCP的最大可接收报文段”。在一些实现中，这可能受主机能够重装的最大IP数据报限制。然而在当前的大多数系统中，合理的限制决定于输出接口的MTU，因为如果需要分段并且发生报文段丢失，则TCP的性能会下降。

下面的注释摘抄于Bob Braden的T/TCP源码修改：“非常不幸，使用TCP选项要求对BSD作可观的修改，因为它对MSS的处理是错误的。BSD总是要发出MSS选项，并且对非本地网络的主机，这个选项的值是536。这是对MSS选项用途的误解，这个选项是要告诉发送者，接收者准备处理什么。这时发送主机要决定用多大的MSS，既要考虑它接收的MSS选项，还要考虑到路由情况。当我们有了MTU发现以后，这个路由很可能有一个大于536的MTU；这样，BSD就会降低吞吐率。因此，这个程序只确定了应该发送什么样的 MSS选项：本地接口的MTU减去40。”(这段注释中讲到的值536应为512)。

我们在下一节(图10-12)中会看到，如果对等端是非本地主机， MSS选项的接收者才把MSS减到512。

10.8 tcp_mssrcvd函数

在执行隐式连接建立时，收到 SYN以后的tcp_input要调用tcp_mssrcvd，PRU_SEND和PRU_SEND_EOF也都要调用它。该函数与卷2中的tcp_mss函数相似，但是它们之间还是有足够的差别，能够完成我们所需的全部功能。这个函数的主要目标是设置两个变量，一个是t_maxseg(我们在每个报文段中发送的最大数据量)，另一个是t_maxopd(在每个报文段中发送的数据加选项的最大长度)。图10-10给出了这个函数的第一部分。

tcp_input.c

```
1755 void
1756 tcp_mssrcvd(tp, offer)
1757 struct tcpcb *tp;
1758 int     offer;
1759 {
1760     struct rtentry *rt;
1761     struct ifnet *ifp;
1762     int     rtt, mss;
1763     u_long  bufsize;
1764     struct inpcb *inp;
1765     struct socket *so;
1766     struct rmxp_tao *taop;
1767     int     origoffer = offer;
1768     extern int tcp_mssdflt;
1769     extern int tcp_do_rfc1323;
1770     extern int tcp_do_rfc1644;

1771     inp = tp->t_inpcb;
1772     if ((rt = tcp_rtlookup(inp)) == NULL) {
1773         tp->t_maxopd = tp->t_maxseg = tcp_mssdflt;
1774         return;
1775     }
1776     ifp = rt->rt_ifp;
1777     so = inp->inp_socket;

1778     taop = rmx_taop(rt->rt_rmx);
1779     /*
1780      * Offer == -1 means we haven't received a SYN yet;
1781      * use cached value in that case.
1782      */
1783     if (offer == -1)
1784         offer = taop->tao_mssopt;
1785     /*
1786      * Offer == 0 means that there was no MSS on the SYN segment,
1787      * or no value in the TAO Cache.  We use tcp_mssdflt.
1788      */
1789     if (offer == 0)
1790         offer = tcp_mssdflt;
1791     else
1792         /*
1793          * Sanity check: make sure that maxopd will be large
1794          * enough to allow some data on segments even if all
1795          * the option space is used (40 bytes).  Otherwise
1796          * funny things may happen in tcp_output.
1797          */
1798         offer = max(offer, 64);
1799     taop->tao_mssopt = offer;
```

tcp_input.c

图10-10 tcp_mssrcvd 函数：第一部分

1. 取对等端的路由及其TAO缓存

1771-1777 tcp_rtlookup查找到达对等端的路由。如果由于某种原因，查找路由失败了，t_maxseg和t_maxopd就同时设置为512(tcp_mssdflt)。

1778-1799 taop指向该对等端的TAO缓存，位于路由表的记录项中。如果因为用户进程调用了sendto(一次隐式连接建立，是PRU_SEND和PRU_SEND_EOF请求的一部分)而调用tcp_mssrcvd，则offer设置为TAO缓存中保存的值。如果TAO中的该值为0，offer就设置为512。TAO缓存中的值被更新。

图10-11给出了该函数的第二部分，与卷2第718页完全相同。

tcp_input.c

```
1800      /*
1801       * While we're here, check if there's an initial rtt
1802       * or rttvar.  Convert from the route-table units
1803       * to scaled multiples of the slow timeout timer.
1804       */
1805      if (tp->t_srtt == 0 && (rtt = rt->rt_rmx.rmx_rtt)) {
1806          /*
1807           * XXX the lock bit for RTT indicates that the value
1808           * is also a minimum value; this is subject to time.
1809           */
1810          if (rt->rt_rmx.rmx_locks & RTV_RTT)
1811              tp->t_rttmin = rtt / (RTM_RTTUNIT / PR_SLOWHZ);
1812          tp->t_srtt = rtt / (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));
1813          if (rt->rt_rmx.rmx_rttvar)
1814              tp->t_rttvar = rt->rt_rmx.rmx_rttvar /
1815                  (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
1816          else
1817              /* default variation is +- 1 rtt */
1818              tp->t_rttvar =
1819                  tp->t_srtt * TCP_RTTVAR_SCALE / TCP_RTT_SCALE;
1820          TCPT_RANGESET(tp->t_rxtcur,
1821                        ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1,
1822                        tp->t_rttmin, TCPTV_REXMTMAX);
1823      }
```

tcp_input.c

图10-11 tcp_mssrcvd 函数：用路由表度量值初始化RTT变量

1800-1823 如果还没有该连接的RTT测量值(t_srtt为0)，并且rmx_rtt度量值为非0，这时变量t_srtt、t_rttvar和t_rxtcur就用路由表记录项中保存的度量值初始化。

如果路由表度量值加锁标志中的RTV_RTT位已经设置，则它表明还要用rmx_rtt来初始化这次连接的最小RTT(t_rttmin)。默认情况下，t_rttmin初始化为两个时钟步进，这为系统管理员替换该默认值提供了一个方法。

图10-12给出了tcp_mssrcvd的第三部分，用于设置自动变量mss的值。

tcp_input.c

```
1824      /*
1825       * If there's an mtu associated with the route, use it.
1826       */
1827      if (rt->rt_rmx.rmx_mtu)
1828          mss = rt->rt_rmx.rmx_mtu - sizeof(struct tcpiphdr);
1829      else {
1830          mss = ifp->if_mtu - sizeof(struct tcpiphdr);
1831          if (!in_localaddr(inp->inp_faddr))
1832              mss = min(mss, tcp_mssdflt);
1833      }
1834      mss = min(mss, offer);

1835      /*
1836       * t_maxopd contains the maximum length of data AND options
1837       * in a segment; t_maxseg is the amount of data in a normal
1838       * segment.  We need to store this value (t_maxopd) apart
1839       * from t_maxseg, because now every segment can contain options
1840       * therefore we normally have somewhat less data in segments.
1841       */
1842      tp->t_maxopd = mss;
```

tcp_input.c

图10-12 tcp_mssrcvd 函数：计算mss变量的值

1824-1834 如果该路由关联于一个MTU(rmx_mtu度量值)，那就用这个值。否则，mss就取输出接口的MTU减去40。另外，如果对等端是在另一个网络，或者也可能在另一个子网 (由in_localaddr函数决定)中，这时mss的最大值取为512(tcp_mssdflt)。如果路由表记录项中已经保存有MTU，那就不再进行本地-非本地测试。

2. 设置t_maxopd

1835-1842 t_maxopd设置为mss，包括了数据和选项的最大报文段长度。

图10-13给出的是第四部分代码，将mss减去在每一个报文段中都有的选项长度。

tcp_input.c

```
1843     /*
1844      * Adjust mss to leave space for the usual options.  We're
1845      * called from the end of tcp_dooptions so we can use the
1846      * REQ/RCVD flags to see if options will be used.
1847      */
1848     /*
1849      * In case of T/TCP, origoffer == -1 indicates that no segments
1850      * were received yet (i.e., client has called sendto).  In this
1851      * case we just guess, otherwise we do the same as before T/TCP.
1852      */
1853     if ((tp->t_flags & (TF_REQ_TSTMP | TF_NOOPT)) == TF_REQ_TSTMP &&
1854         (origoffer == -1 ||
1855          (tp->t_flags & TF_RCVD_TSTMP) == TF_RCVD_TSTMP))
1856         mss -= TCPOLEN_TSTAMP_APPA;

1857     if ((tp->t_flags & (TF_REQ_CC | TF_NOOPT)) == TF_REQ_CC &&
1858         (origoffer == -1 ||
1859          (tp->t_flags & TF_RCVD_CC) == TF_RCVD_CC))
1860         mss -= TCPOLEN_CC_APPA;

1861 #if (MCLBYTES & (MCLBYTES - 1)) == 0
1862     if (mss > MCLBYTES)
1863         mss &= ~(MCLBYTES - 1);
1864 #else
1865     if (mss > MCLBYTES)
1866         mss = mss / MCLBYTES * MCLBYTES;
1867 #endif
```

tcp_input.c

图10-13 tcp_mssrcvd 函数：根据选项减小mss

3. 如果使用时间戳选项就减小mss

1843-1856 如果下面中的任何一个条件为真，则 mss就减去时间戳选项的长度(TCPOLEN_TSTAMP_APPA，即12字节)：

1) 本地端将使用时间戳选项(TF_REQ_TSTAMP)，并且还没有收到另一端发来的mss选项(origoffer等于-1)；或

2) 已经收到另一个端发来的时间戳选项。

在代码的注释中指出，由于tcp_mssrcvd是在tcp_dooptions结束时所有的选项处理完以后调用的(见图10-18)，因此第二项测试是成功的。

4. 如果使用CC选项，就减少mss

1857-1860 通过相似的逻辑，mss的值减去8字节(TCPOLEN_CC_APPA)。

这两个长度名称中出现术语APPA是因为，RFC 1 323的附录A中建议在时间戳选项前面置两个空字符 NOP，以便两个4字节时间戳值的长度都能取 4字节的整数倍。

同时RFC 1644也有一个附录A，它对选项排列没有说什么。无论怎样，在三个 CC选项的任一个前面都置两个NOP是有一定道理的，如图9-6所示。

5. 舍入MSS为MCLBYTES的倍数

1861-1867 mss要舍入取整为MCLBYTES的整数倍，即每个mbuf簇的字节数(通常为1024或2 048)。

这段代码有一个糟糕的优化企图，即如果 MCLBYTES是2的整数幂，则可以用逻辑操作来代替乘法或除法运算。自从 Net/1开始，它就已经是一条弯路，应该清除掉。

图10-14给出了tcp_mssrcvd代码的最后一部分，用于设置发送缓存和接收缓存的大小。

tcp_input.c

```
1868     /*
1869      * If there's a pipesize, change the socket buffer
1870      * to that size.  Make the socket buffers an integral
1871      * number of mss units; if the mss is larger than
1872      * the socket buffer, decrease the mss.
1873      */
1874     if ((bufsize = rt->rt_rmx.rmx_sendpipe) == 0)
1875         bufsize = so->so_snd.sb_hiwat;
1876     if (bufsize < mss)
1877         mss = bufsize;
1878     else {
1879         bufsize = roundup(bufsize, mss);
1880         if (bufsize > sb_max)
1881             bufsize = sb_max;
1882         (void) sbreserve(&so->so_snd, bufsize);
1883     }
1884     tp->t_maxseg = mss;

1885     if ((bufsize = rt->rt_rmx.rmx_recvpipe) == 0)
1886         bufsize = so->so_rcv.sb_hiwat;
1887     if (bufsize > mss) {
1888         bufsize = roundup(bufsize, mss);
1889         if (bufsize > sb_max)
1890             bufsize = sb_max;
1891         (void) sbreserve(&so->so_rcv, bufsize);
1892     }
1893     /*
1894      * Don't force slow-start on local network.
1895      */
1896     if (!in_localaddr(inp->inp_faddr))
1897         tp->snd_cwnd = mss;

1898     if (rt->rt_rmx.rmx_ssthresh) {
1899         /*
1900          * There's some sort of gateway or interface
1901          * buffer limit on the path.  Use this to set
1902          * the slow start threshhold, but set the
1903          * threshold to no less than 2*mss.
1904          */
1905         tp->snd_ssthresh = max(2 * mss, rt->rt_rmx.rmx_ssthresh);
1906     }
1907 }
```

tcp_input.c

图10-14 tcp_mssrcvd 函数：设置发送和接收缓存的大小

6. 改变插口发送缓存的大小

1868-1883 系统管理员可以用route程序设置rmx_sendpipe和rmx_recvpipe这两个度量值。bufsize的值就设置为rmx_sendpipe的值(如果已有定义)，或者当前插口发送缓存的高位值。如果bufsize的值小于mss，mss值就减小为取bufsize的值(这是一种强迫MSS取比给定目标的默认值还小的取值方法)。否则，bufsize的值放大，取mss的整数倍(插口缓存的大小总是取报文段长度的整数倍)。上限为sb_max，在Net/3就是262 144。插口缓存的高位值由sbreserve设置。

7. 设置t_maxseg

1884 t_maxseg设置为TCP将发给对等端的最大数据量(不包括常规选项)。

8. 改变插口接收缓存的大小

1885-1892 插口接收缓存的高位值可以用类似的逻辑来设置。例如，对于以太网上的本地连接来说，假定时间戳选项和CC选项同时都在用，则t_maxopd将是1460，t_maxseg为1440(见图2-4)。插口发送缓存和接收缓存都将从它们的默认值 8192(卷2的图16-4)舍入到8640(1440×6)。

9. 非本地对等端才有的慢启动

1893-1897 如果对等端不是在本地的网络中(in_localaddr为假)，则把拥塞窗口(snd_cwnd)设置为1个报文段就开始了慢启动过程。

> 仅仅当对等端在非本地网中才强迫使用慢启动是T/TCP修改后的结果。这就使T/TCP的客户端或服务器端可以向本地对等端发送多个报文段，又不需要慢启动所要求的额外RTT等待时间(见3.6节)。在Net/3中，总是执行慢启动过程(卷2第721页)。

10. 设置慢启动门限

1898-1906 如果慢启动门限度量值(rmx_ssthresh)非0，snd_ssthresh就设置取该值。

我们在图3-1和图3-3中可以看到MSS和TAO缓存与接收缓存大小之间的交叉影响。在图3-1中，客户端执行了一次隐式连接建立，PRU_SEND_EOF请求调用tcp_mssrcvd，其中offer为-1，该函数查找到对应服务器的tao_mssopt值为0(因为客户端刚刚重启)。取MSS为默认值512，因为只使用了CC选项(在第2章的例子中，时间戳无效)，减去8字节后变为504。注意，8192舍入为504的整数倍后为8568，这是客户端SYN所通告的窗口。然而，当服务器调用tcp_mssrcvd时，它已经接收到客户端的SYN，其中给定MSS为1460。这个值减去8字节(选项长度)后为1452，8192舍入到1452的整数倍后为8172。这是服务器的SYN中通告的窗口。当客户端处理完服务器的SYN后(图中第三段)，客户端再次调用tcp_mssrcvd，这一次offer为1460。这就将客户端的t_maxopd增大至1460，客户端的t_maxseg则增大至1452，客户端的接收缓存因舍入而增至8172。这就是客户端在对服务器的SYN作出ACK时通告的窗口。

在图3-3中，当客户端执行了隐式连接建立时，tao_mssopt值为1460——最近一次从对等端收到的值。客户端通告的窗口为8712，1452的整数倍且大于8192。

10.9 tcp_dooptions函数

在Net/1和Net/2版中，tcp_dooptions只能识别NOP、EOL和MSS选项，并且函数有3个参数。在Net/3中增加了对窗口宽度和时间戳选项的支持后，参数的数量也增加到7个(卷2第

745~746页)，其中有3个就是为了时间戳选项而加的。现在又需要支持 CC、CCnew和CCecho选项，参数的数量不是增加反而减少到了 5个，因为采用了另一种技术来返回选项是否存在以及它们各自的取值信息。

图10-15给出了 tcpopt结构。其中的一个结构是在 tcp_input(唯一可以调用tcp_dooptions的函数)中分配的，并且将指向该结构的指针传给 tcp_dooptions，该函数填写结构的内容。在处理接收到的报文段时， tcp_input要用到存储在该结构中的值。

tcp_var.h

```
138 struct tcpopt {
139     u_long  to_flag;            /* TOF_xxx flags */
140     u_long  to_tsval;           /* timestamp value */
141     u_long  to_tsecr;           /* timestamp echo reply */
142     tcp_cc  to_cc;              /* CC or CCnew value */
143     tcp_cc  to_ccecho;          /* CCecho value */
144 };
```

tcp_var.h

图10-15 tcpopt 结构，由tcp_dooptions 填写数据

图10-16给出了to_flag字段可以组合出的4个值。

to_flag	说 明
TOF_CC	CC选项存在
TOF_CCNEW	CCnew选项存在
TOF_CCECHO	CCecho选项存在
TOF_TS	时间戳选项存在

图10-16 to_flag 的取值

图10-17给出了这个函数的参数说明。前 4个参数与Net/3中的相同，但第5个参数替换了Net/3版本中的最后3个参数。

tcp_input.c

```
1520 void
1521 tcp_dooptions(tp, cp, cnt, ti, to)
1522 struct tcpcb *tp;
1523 u_char *cp;
1524 int     cnt;
1525 struct tcpiphdr *ti;
1526 struct tcpopt *to;
1527 {
```

tcp_input.c

图10-17 tcp_dooptions 函数：参数

因为处理EOL、NOP、MSS、窗口宽度和时间戳选项的代码与卷 2第745~747页的代码几乎相同，所以这里不再重复介绍 (差别主要在于对新参数的处理，我们刚刚讨论过)。图10-18给出了这个函数的最后一部分代码，它们用于 T/TCP处理3个新的选项。

1. 检查长度和是否处理选项

1580-1584 选项长度要验证(所有3个CC选项的长度必须都是6)。处理接收到的CC选项时，我们也必须发送相应选项 (如果内核的 tcp_do_rfc1644标志已经设置，则 tcp_newtcpcb要设置TF_REQ_CC标志)，并且TF_NOOPT标志不能设置 (最后这个标志不允许TCP 在其SYN中发送任何选项)。

2. 设置相应标志并复制4字节值

1585-1588 设置相应的to_flag值。四个字节的选项值存储在tcpopt结构的to_cc字段中，并且要先转换成主机的字节顺序。

1589-1595 如果这是一个SYN报文段，要为该连接设置TF_RCVD_CC标志，因为收到了CC选项。

3. CCnew和CCecho选项

1596-1623 CCnew和CCecho选项的处理步骤与CC选项的相似。但因为CCnew和CCecho选项仅在SYN报文段中有效，所以要附加一项检测，检查报文段中是否包含 SYN标志。

尽管TOF_CCNEW标志都有正确设置，但从来不去检查它。这是因为在图 11-6中，如果CC选项不存在，则缓存的CC值是无效的(即需设置为0)。如果存在CCnew选项，则cc_recv仍然有正确设置(注意，在图10-18中，CC和CCnew选项都在to_cc中存储其值)，并且当三次握手完成时(图11-14)，所缓存的值tao_cc是从cc_recv中复制过来的。

4. 处理收到的MSS

1625-1626 局部变量mss记录的或者是MSS选项的值(如果选项存在)，或者是表示选项不存在的0值。在这两种情况下，tcp_mssrcvd都要设置变量t_maxseg和t_maxopd的值。在tcp_dooptions快结束时要调用该函数，因为如图 10-13所示，tcp_mssrcvd使用了TF_RCVD_TSTMP和TF_RCVD_CC标志。

10.10 tcp_reass函数

当服务器收到包含数据的 SYN时，假定TAO测试失败或报文段中不包含 CC选项，那么tcp_input就将数据存入缓存队列，等待三次握手过程的完成。在图 11-6中，协议的状态设置为SYN_RCVD，程序有一个分支trimthenstep6，在标号为dodata的程序行(卷2第790页)，宏TCP_REASS发现协议状态不是ESTABLISHED，因此调用tcp_reass将报文段存入该连接的失序报文队列(其中的数据并非真的失序；只是因为它是在三次握手过程完成之前到达的。然而，卷2的图27-19底部的两个统计计数器tcps_rcvoopack和tcps_rcvoobyte的累进是不正确的)。

tcp_input.c

```
1580          case TCPOPT_CC:
1581              if (optlen != TCPOLEN_CC)
1582                  continue;
1583              if ((tp->t_flags & (TF_REQ_CC | TF_NOOPT)) != TF_REQ_CC)
1584                  continue;       /* we're not sending CC opts */
1585              to->to_flag |= TOF_CC;
1586              bcopy((char *) cp + 2, (char *) &to->to_cc,
1587                    sizeof(to->to_cc));
1588              NTOHL(to->to_cc);
1589              /*
1590               * A CC or CCnew option received in a SYN makes
1591               * it OK to send CC in subsequent segments.
1592               */
1593              if (ti->ti_flags & TH_SYN)
1594                  tp->t_flags |= TF_RCVD_CC;
```

图10-18 tcp_dooptions 函数：新T/TCP选项的处理

```
1595                break;

1596            case TCPOPT_CCNEW:
1597                if (optlen != TCPOLEN_CC)
1598                    continue;
1599                if ((tp->t_flags & (TF_REQ_CC | TF_NOOPT)) != TF_REQ_CC)
1600                    continue;       /* we're not sending CC opts */
1601                if (!(ti->ti_flags & TH_SYN))
1602                    continue;
1603                to->to_flag |= TOF_CCNEW;
1604                bcopy((char *) cp + 2, (char *) &to->to_cc,
1605                      sizeof(to->to_cc));
1606                NTOHL(to->to_cc);
1607                /*
1608                 * A CC or CCnew option received in a SYN makes
1609                 * it OK to send CC in subsequent segments.
1610                 */
1611                tp->t_flags |= TF_RCVD_CC;
1612                break;

1613            case TCPOPT_CCECHO:
1614                if (optlen != TCPOLEN_CC)
1615                    continue;
1616                if (!(ti->ti_flags & TH_SYN))
1617                    continue;
1618                to->to_flag |= TOF_CCECHO;
1619                bcopy((char *) cp + 2, (char *) &to->to_ccecho,
1620                      sizeof(to->to_ccecho));
1621                NTOHL(to->to_ccecho);
1622                break;
1623            }
1624        }
1625        if (ti->ti_flags & TH_SYN)
1626            tcp_mssrcvd(tp, mss);    /* sets t_maxseg */
1627    }
```

tcp_input.c

图10-18 (续)

当对服务器所发的SYN的ACK(通常是三次握手中的第三个报文段)姗姗来迟时，执行卷2第774页的case TCPS_SYN_RECEIVED语句，使连接进入到ESTABLISHED状态，然后调用rcp_reass函数将队列中的数据交付给进程，该函数第二个参数为0。但在图11-14中，我们会看到，如果新的报文段中有数据，或者如果设置了FIN标志，就跳过对tcp_reass函数的调用，因为这两种情况的任何一种都会引起对标号为dodata的TCP_REASS函数的调用。问题是，如果新的报文段完全与以前的报文段重复，则对TCP_REASS函数的调用不会强行将队列中的数据交付给进程。

修改tcp_reass函数只需做很小的改变：将卷2第729页的第106行的return改为执行标号为present的分支。

10.11 小结

给定主机的TAO信息保存在路由表的记录项中。函数tcp_gettaocache读取为某主机缓存的TAO数据，但如果在PCB的路由缓存中尚不存在相应的路由，则调用tcp_rtlookup来查找主机。

T/TCP修改tcp_close函数，在路由表中为T/TCP连接保存两个估计值*srtt*和*rttvar*，即使连接中只传送了不到16个满长度的报文段。这样就使与该主机的下一次T/TCP连接可以在开始时使用这两个估计值(假设路由表记录项在下一次连接时还没有超时)。

Net/3的函数tcp_mss在T/TCP中分成了两个函数：tcp_mssrcvd和tcp_msssend。前者在收到MSS选项后调用，后者在发出MSS选项时调用。后者与通常的BSD做法的不同之处在于，它一般声明其MSS为输出接口的MTU减去TCP和IP首部的长度。BSD系统会向非本地对等主机声明取值为512的MSS。

Net/3中的tcp_dooptions函数在T/TCP中也有改变。函数的若干个参数取消了，用一个结构来代替。这就使函数可以处理新的选项(例如T/TCP新增加的3个选项)，而不需增加参数。

第11章 T/TCP实现：TCP输入

11.1 概述

T/TCP对TCP所做的大多数修改是在tcp_input函数中。在整个函数的前前后后都出现的修改是tcp_dooptions(10.9节)中新增的变量和返回值。我们不打算给出受这一变化而影响的每一块代码。

图11-1是卷2中图28-1的重写，T/TCP所做的修改用黑体字表示。

我们在说明tcp_input函数所做的修改时，按照各部分在整个函数中出现的顺序来分别介绍。

```
void
tcp_input()
{
    checksum TCP header and data;
    skip over IP/TCP headers in mbuf;
findpcb:
    locate PCB for segment;
    if (not found)
        goto dropwithreset;
    reset idle time to 0 and keepalive timer to 2 hours;
    process options if not LISTEN state;
    if (packet matched by header prediction) {
        completely process received segment;
        return;
    }

    switch (tp->t_state) {
    case TCPS_LISTEN:
        if SYN flag set, accept new connection request;
        perform TAO test;
        goto trimthenstep6;

    case TCPS_SYN_SENT:
        check CCecho option;
        if ACK of our SYN, connection completed;
trimthenstep6:
        trim any data not within window;
        if (ACK flag set)
            goto processack;
        goto step6;

    case TCPS_LAST_ACK:
    case TCPS_CLOSING:
    case TCPS_TIME_WAIT:
        check for new SYN as implied ACK of previous incarnation;
    }
```

图11-1 TCP输入处理步骤小结：T/TCP所做的修改用黑体表示

```
    process RFC 1323 timestamp;
    check CC option;
    check if some data bytes are within the receive window;
    trim data segment to fit within window;
    if (RST flag set) {
        process depending on state;
        goto drop;
    }
     if (ACK flag off)
        if (SYN_RCVD || half-synchronized)
            goto step6;
        else
            goto drop;
    if (ACK flag set) {
        if (SYN_RCVD state)
            passive open or simultaneous open complete;
        if (duplicate ACK)
            fast recovery algorithm;
processack:
        update RTT estimators if segment timed;
        if (no data was ACKed)
            goto step6;
        open congestion window;
        remove ACKed data from send buffer;
        change state if in FIN_WAIT_1, CLOSING, or LAST_ACK state;
    }
step6:
    update window information;
    process URG flag;
dodata:
    process data in segment, add to reassembly queue;
    if (FIN flag is set)
        process depending on state;
    if (SO_DEBUG socket option)
        tcp_trace(TA_INPUT);
    if (need output || ACK now)
        tcp_output();
    return;
dropafterack:
    tcp_output() to generate ACK;
    return;
dropwithreset:
    tcp_respond() to generate RST;
    return;
drop:
    if (SO_DEBUG socket option)
        tcp_trace(TA_DROP);
    return;
}
```

图11-1 (续)

11.2 预处理

定义了三个新的自动变量，其中之一是tcpopt结构，在tcp_dooptions中使用。下面的几行语句用于替换卷2第739页的第190行。

```
struct tcpopt to;               /* options in this segment */
struct rmxp_tao *taop;          /* pointer to our TAO cache entry */
struct rmxp_tao tao_noncached;  /* in case there's no cached entry */

bzero((char *)&to, sizeof(to));
tcpstat.tcps_rcvtotal++;
```

将tcpopt结构初始化为0是非常重要的：这样就会将to_cc字段(接收到的CC值)设置为0，表明它未定义。

在Net/3中，唯一回到标号findpcb的分支是在一个连接处于TIME_WAIT状态时又收到一个新的SYN报文段(卷2第765~766页)。因为下面的这两行代码有问题，因而该分支存在一个缺陷

```
m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
m->m_len  -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
```

这两行代码在findpcb后出现了两次，在goto后又执行了一次(这两行代码在卷2第751页出现了一次，在第752页又出现一次；这两处中只能有一处执行，决定于该报文段是否与首部所指示的相一致)。这在T/TCP之前并不会带来问题，因为SYN不携带数据，上述这个缺陷只在当一个连接处于TIME_WAIT状态又收到一个携带数据的新SYN时才会表现出来。然而在T/TCP中，还会有第2个回到findpcb的分支(在后面的图11-11中会说明，这个分支处理图4-7所示的隐式ACK)，并且要处理的SYN很可能携带数据。这样，在findpcb之前的上述这两行代码就必须删去，如图11-2所示。

tcp_input.c

```
274     /*
275      * Skip over TCP, IP headers, and TCP options in mbuf.
276      * optp & ti still point into TCP header, but that's OK.
277      */
278     m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
279     m->m_len -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);

280     /*
281      * Locate pcb for segment.
282      */
283   findpcb:
```

tcp_input.c

图11-2 tcp_input：在findpcb前修改mbuf指针和长度

这样就从卷2的第751页和第752页中删除上述的两行。

下一个修改位于卷2第744页的第327行，这段代码在一个新报文段到达监听插口时创建一个新插口。在t_state设置为TCPS_LISTEN以后，TF_NOPUSH和TF_NOOPT这两个标志必须从监听插口复制到新的插口：

```
tp->t_flags |= tp0->t_flags & (TF_NOPUSH|TF_NOOPT);
```

其中tp0是指向监听插口tcpcb的自动变量。

卷2第745页的第344~345行中对tcp_dooptions的调用要改为新的调用序列(10.9节)：

```
if (optp && tp->t_state != TCPS_LISTEN)
    tcp_dooptions(tp, optp, optlen, ti, &to);
```

11.3 首部预测

是否应用首部预测(卷2第748页)的第一项测试是检查隐藏状态标志是否关闭。如果这些标志中有任何一个处于打开状态，则需要用 tcp_input中的慢通道处理将其关闭。图 11-13给出了新的测试过程。

tcp_input.c

```
398     if (tp->t_state == TCPS_ESTABLISHED &&
399       (tiflags & (TH_SYN | TH_FIN | TH_RST | TH_URG | TH_ACK)) == TH_ACK &&
400         ((tp->t_flags & (TF_SENDSYN | TF_SENDFIN)) == 0) &&
401         ((to.to_flag & TOF_TS) == 0 ||
402          TSTMP_GEQ(to.to_tsval, tp->ts_recent)) &&
403     /*
404      * Using the CC option is compulsory if once started:
405      *   the segment is OK if no T/TCP was negotiated or
406      *   if the segment has a CC option equal to CCrecv
407      */
408     ((tp->t_flags & (TF_REQ_CC | TF_RCVD_CC)) != (TF_REQ_CC | TF_RCVD_CC) ||
409      (to.to_flag & TOF_CC) != 0 && to.to_cc == tp->cc_recv) &&
410         ti->ti_seq == tp->rcv_nxt &&
411         tiwin && tiwin == tp->snd_wnd &&
412         tp->snd_nxt == tp->snd_max) {
413         /*
414          * If last ACK falls within this segment's sequence numbers,
415          * record the timestamp.
416          * NOTE that the test is modified according to the latest
417          * proposal of the tcplw@cray.com list (Braden 1993/04/26).
418          */
419         if ((to.to_flag & TOF_TS) != 0 &&
420             SEQ_LEQ(ti->ti_seq, tp->last_ack_sent)) {
421             tp->ts_recent_age = tcp_now;
422             tp->ts_recent = to.to_tsval;
423         }
```

tcp_input.c

图11-3 tcp_inpu t：是否可以应用首部预测

1. 验证隐藏状态标志关闭

400 这里的第一项修改就是验证TF_SENDSYN和TF_SENDFIN标志是否同时处于关闭状态。

2. 检查时间戳选项(如果存在)

401-402 第2项修改与修改后的tcp_dooptions函数有关的是：不再测试ts_present，而是测试to_flag中的TOF_TS标志位，并且如果时间戳存在，它的值是在 to_tsval而不是ts_val中。

3. 如果使用T/TCP，就验证CC

403-409 最后，如果没有完成T/TCP协商(我们要求有CC选项，但另一个端没有发送，或者我们根本就没有要求)，则if测试继续进行。如果使用了 T/TCP，则接收到的报文段必须包含一个CC选项，且CC值必须等于cc_recv值，这样才继续进行if测试。

我们希望在简短的T/TCP事务中不要频繁使用首部预测。这是因为在一次最小的 T/TCP报文段交换中，其最初的两个报文段携带有控制标志 (SYN和FIN)，这会使图11-3中在第二项测

试失败。这些T/TCP报文段用tcp_input的慢通道进行处理。但是，在支持T/TCP的两个主机之间的长连接(例如成批的数据传送)可以使用CC选项，并从首部预测中获益。

4. 用接收到的时间戳更新ts_recent

413-423 ts_recent是否因该更新的测试与卷2第748页的第371~372行有所不同。图11-3中采用新测试代码的原因在卷2第694~695页有详细叙述。

11.4 被动打开的启动

我们现在替换掉卷2第755页的全部代码：处于LISTEN状态的插口处理所收到的SYN的代码的最后一部分。这是当服务器从一个客户端接收到一个 SYN时被动打开的启动(我们不想重复卷2第753~754页中在该状态下进行初始化的代码)。图11-4给出了这段代码的第一部分。

tcp_input.c

```
545             tp->t_template = tcp_template(tp);
546             if (tp->t_template == 0) {
547                 tp = tcp_drop(tp, ENOBUFS);
548                 dropsocket = 0; /* socket is already gone */
549                 goto drop;
550             }
551             if ((taop = tcp_gettaocache(inp)) == NULL) {
552                 taop = &tao_noncached;
553                 bzero(taop, sizeof(*taop));
554             }
555             if (optp)
556                 tcp_dooptions(tp, optp, optlen, ti, &to);
557             if (iss)
558                 tp->iss = iss;
559             else
560                 tp->iss = tcp_iss;
561             tcp_iss += TCP_ISSINCR / 4;
562             tp->irs = ti->ti_seq;
563             tcp_sendseqinit(tp);
564             tcp_rcvseqinit(tp);
565             /*
566              * Initialization of the tcpcb for transaction:
567              *   set SND.WND = SEG.WND,
568              *   initialize CCsend and CCrecv.
569              */
570             tp->snd_wnd = tiwin;     /* initial send-window */
571             tp->cc_send = CC_INC(tcp_ccgen);
572             tp->cc_recv = to.to_cc;
```

tcp_input.c

图11-4 tcp_input ：取TAO记录项，初始化事务的控制块

1. 取客户端的TAO记录项

551-554 tcp_gettaocache查找该客户端的TAO记录项。如果没有找到，在全部设置为0后使用自动变量。

2. 处理选项和初始化序号

555-564 tcp_dooptions处理所有的选项(由于连接处于LISTEN状态，这个函数在此之前是不会调用的)。初始化发送序号(iss)和初始接收序号(irs)。控制块中的所有序号变量都由tcp_sendseqinit和tcp_rcvseqinit进行初始化。

3. 更新发送窗

565-570 tiwin是在接收到的SYN中由客户端通告的窗口(卷2第742~743页)。它是新插口的初始化发送窗口。通常，发送窗口要一直等到收到了一个带有 ACK的报文段才会更新(卷2第785页)。但T/TCP要利用所收到的 SYN报文段中的发送窗口值，即使这个报文段不包含ACK。这个窗口影响到服务器端给出应答时可以立即发送给客户端的数据有多少 (T/TCP交换中最小三报文段的第2个报文段)。

4. 设置cc_send和cc_recv

571-572 cc_send设置为tcp_ccgen的值，并且如果CC选项存在，则cc_recv设置为CC值。如果CC选项不存在，因为在函数的一开始已经将 to初始化为0，所以cc_recv也是0(未定义)。

5. 执行TAO测试

573-587 仅仅在报文段中包含有CC选项时才进行TAO测试。如果接收到的CC值非0且大于该客户端的缓存值(tao_cc)，则TAO测试成功。

6. TAO测试成功；更新客户端的TAO缓存

588-594 对这个客户端的缓存值进行更新，并且将连接状态设置为 ESTABLISHED*(隐藏状态变量在稍后的几行中设置，使之成为半同步加星状态)。

7. 决定是否延迟发送ACK

595-606 如果报文段中包含FIN，或者如果报文段中包含数据，那么客户端应用程序必须按使用T/TCP来编程(即调用sendto，并指定MSG_EOF，在此之前不能调用connect、write和shutdown)。在这种情况下， ACK要延迟发送，以便让服务器的应答来捎带服务器给出的SYN/ACK。

tcp_input.c

```
573             /*
574              * Perform TAO test on incoming CC (SEG.CC) option, if any.
575              * - compare SEG.CC against cached CC from the same host,
576              *  if any.
577              * - if SEG.CC > cached value, SYN must be new and is accepted
578              *  immediately: save new CC in the cache, mark the socket
579              *  connected, enter ESTABLISHED state, turn on flag to
580              *  send a SYN in the next segment.
581              *  A virtual advertised window is set in rcv_adv to
582              *  initialize SWS prevention.  Then enter normal segment
583              *  processing: drop SYN, process data and FIN.
584              * - otherwise do a normal 3-way handshake.
585              */
586             if ((to.to_flag & TOF_CC) != 0) {
587                 if (taop->tao_cc != 0 && CC_GT(to.to_cc, taop->tao_cc)) {
588                     /*
589                      * There was a CC option on the received SYN
590                      * and the TAO test succeeded.
591                      */
592                     tcpstat.tcps_taook++;
593                     taop->tao_cc = to.to_cc;
594                     tp->t_state = TCPS_ESTABLISHED;

595                     /*
596                      * If there is a FIN, or if there is data and the
597                      * connection is local, then delay SYN,ACK(SYN) in
```

图11-5 tcp_input ：对收到的报文段执行TAO测试

```
598                     * the hope of piggybacking it on a response
599                     * segment.  Otherwise must send ACK now in case
600                     * the other side is slow starting.
601                     */
602                    if ((tiflags & TH_FIN) ||
603                        (ti->ti_len != 0 && in_localaddr(inp->inp_faddr)))
604                        tp->t_flags |= (TF_DELACK | TF_SENDSYN);
605                    else
606                        tp->t_flags |= (TF_ACKNOW | TF_SENDSYN);
607                    tp->rcv_adv += tp->rcv_wnd;
608                    tcpstat.tcps_connects++;
609                    soisconnected(so);
610                    tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;
611                    dropsocket = 0;     /* committed to socket */
612                    tcpstat.tcps_accepts++;
613                    goto trimthenstep6;
614                } else if (taop->tao_cc != 0)
615                    tcpstat.tcps_taofail++;
```

tcp_input.c

图11-5 (续)

如果FIN标志未设置，但是报文段中包含数据，那么由于报文段中同时也包含了SYN标志，这很有可能是客户端发来的多报文段数据中的第一段。在这种情况下，如果客户端不在本地子网中(in_localaddr函数返回的是0)，这时因为客户端可能处于慢启动状态，确认不再延迟。

8. 设置rcv_adv

607 rcv_adv定义为所接收通告的最高序列号加1(卷2的图24-28)，但是在图11-4中的宏tcp_rcvseqinit将其初始化为接收序列号加1。在这个处理点上，rcv_wnd将是插口接收缓存的大小(卷2第752页)。这样，将rcv_wnd加到rcv_adv以后，后者刚好超出当前的接收窗口。rcv_adv必须在这里进行初始化，因为它的值要用在tcp_output的糊涂窗口避免机制中(卷2第700页)。rcv_adv在tcp_output快结束时设置，通常是在发送第一个报文段时(在这里应该是服务器对客户端SYN的响应SYN/ACK)。但是在T/TCP中，rcv_adv需要在tcp_output中进行第一次设置，因为我们可能在所发送的第一个报文段中发送数据。

9. 完成连接

608-609 递增tcps_connects和调用soisconnected通常是在接收到三次握手中的第三个报文段时进行的(卷2第774页)。既然连接已经完成，在T/TCP中这时就执行这两个步骤。

610-613 连接建立定时器设置为75秒，dropsocked标志设置为0，增加了标签为trimthenstep6的分支。

图11-6 给出了在LISTEN状态的插口收到一个SYN时的其余处理代码。

```
616                } else {
617                    /*
618                     * No CC option, but maybe CCnew:
619                     * invalidate cached value.
620                     */
621                    taop->tao_cc = 0;
622                }
```

tcp_input.c

图11-6 tcp_input：LISTEN处理：没有CC选项或TAO测试失败

```
623             /*
624              * TAO test failed or there was no CC option,
625              * do a standard 3-way handshake.
626              */
627             tp->t_flags |= TF_ACKNOW;
628             tp->t_state = TCPS_SYN_RECEIVED;
629             tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;
630             dropsocket = 0;      /* committed to socket */
631             tcpstat.tcps_accepts++;
632             goto trimthenstep6;
633         }
```

tcp_input.c

图11-6 (续)

10. 无CC选项；缓存设置为CC未定义

612-622 当CC选项不存在时，执行else语句(图11-5的第一个if语句在)。缓存中的CC值设置为0(即未定义)。如果该段程序中发现报文段中含有 CCnew选项，则当三次握手过程完成以后要更新缓存的CC值(图11-14)。

11. 要求执行三次握手

623-633 执行到这一点，要么报文段中没有 CC选项，要么有CC选项但TAO测试失败。在任何一种情况下，都要求执行三次握手过程。剩余的代码行与卷 2中图28-17的最后部分完全一样：设置TF_ACKNOW标志，状态设置为SYN_RCVD状态，这样就会立即发送SYN/ACK。

11.5 主动打开的启动

下一个case是SYN_SENT状态。TCP先前发送过一个SYN(一次主动打开)，现在是处理服务器的应答。图11-7给出了处理程序的第一部分。相应的 Net/3代码从卷2第757页开始。

tcp_input.c

```
634         /*
635          * If the state is SYN_SENT:
636          *  if seg contains an ACK, but not for our SYN, drop the input.
637          *  if seg contains a RST, then drop the connection.
638          *  if seg does not contain SYN, then drop it.
639          * Otherwise this is an acceptable SYN segment
640          *  initialize tp->rcv_nxt and tp->irs
641          *  if seg contains ack then advance tp->snd_una
642          *  if SYN has been acked change to ESTABLISHED else SYN_RCVD state
643          *  arrange for segment to be acked (eventually)
644          *  continue processing rest of data/controls, beginning with URG
645          */
646     case TCPS_SYN_SENT:
647         if ((taop = tcp_gettaocache(inp)) == NULL) {
648             taop = &tao_noncached;
649             bzero(taop, sizeof(*taop));
650         }
651         if ((tiflags & TH_ACK) &&
652             (SEQ_LEQ(ti->ti_ack, tp->iss) ||
653              SEQ_GT(ti->ti_ack, tp->snd_max))) {
654             /*
655              * If we have a cached CCsent for the remote host,
656              * hence we haven't just crashed and restarted,
657              * do not send a RST.  This may be a retransmission
```

图11-7 tcp_input ：在SYN_SENT状态的初始处理

```
658             * from the other side after our earlier ACK was lost.
659             * Our new SYN, when it arrives, will serve as the
660             * needed ACK.
661             */
662            if (taop->tao_ccsent != 0)
663                goto drop;
664            else
665                goto dropwithreset;
666        }
667        if (tiflags & TH_RST) {
668            if (tiflags & TH_ACK)
669                tp = tcp_drop(tp, ECONNREFUSED);
670            goto drop;
671        }
672        if ((tiflags & TH_SYN) == 0)
673            goto drop;
674        tp->snd_wnd = ti->ti_win;    /* initial send window */
675        tp->cc_recv = to.to_cc; /* foreign CC */

676        tp->irs = ti->ti_seq;
677        tcp_rcvseqinit(tp);
```

tcp_input.c

图11-7 (续)

1. 取TAO缓存记录项

647-650 取该服务器的TAO缓存记录项。因为我们最近刚刚发送过SYN，应该有一个记录项。

2. 处理不正确的ACK

651-666 如果报文段中包含有ACK，但是其确认字段不正确(见卷2中图28-19对进行比较的几个字段的叙述)，我们的应答就依赖于是否已经为该主机缓存了 `tao_ccsent`。如果`cc_ccsent`非0，则丢弃该报文段，而不发送RST。这个处理步骤的代码段在图4-7中的标号为“discard”处。但如果`tao_ccsent`为0，我们丢弃该报文段，并发送RST，这是在该状态下对不正确ACK的正常TCP响应。

3. 检查RST

667-671 如果接收到的报文段中设置了RST标志，则丢弃报文段。另外，如果设置了ACK标志，则说明服务器的TCP主动拒绝连接，并将`ECONNREFUSED`错误返回给调用进程。

4. 必须设置SYN

672-673 如果SYN标志没有设置，则丢弃报文段。

674-677 初始发送窗口设置为报文段中通告的窗口宽度，并将 `cc_recv`设置为接收到的CC值(如果CC选项不存在，就为0)。`irs`是初始接收序号，宏`tcp_rcvseqinit`对控制块中的接收变量进行初始化。

代码中现在开始出现分支，决定于是否报文段中包含一个对所发SYN的确认ACK(通常情况下)，或者是否ACK标志没有打开(双方同时进行打开的情况较少发生)。图11-18给出了通常的情况。

```
678        if (tiflags & TH_ACK) {
679            /*
680             * Our SYN was acked.  If segment contains CCecho
681             * option, check it to make sure this segment really
```

tcp_input.c

图11-8 tcp_input：在SYN_SENT状态处理SYN/ACK响应

```
682             * matches our SYN.  If not, just drop it as old
683             * duplicate, but send an RST if we're still playing
684             * by the old rules.
685             */
686            if ((to.to_flag & TOF_CCECHO) &&
687                tp->cc_send != to.to_ccecho) {
688                if (taop->tao_ccsent != 0) {
689                    tcpstat.tcps_badccecho++;
690                    goto drop;
691                } else
692                    goto dropwithreset;
693            }
694            tcpstat.tcps_connects++;
695            soisconnected(so);

696            /* Do window scaling on this connection? */
697            if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
698                (TF_RCVD_SCALE | TF_REQ_SCALE)) {
699                tp->snd_scale = tp->requested_s_scale;
700                tp->rcv_scale = tp->request_r_scale;
701            }
702            /* Segment is acceptable, update cache if undefined. */
703            if (taop->tao_ccsent == 0)
704                taop->tao_ccsent = to.to_ccecho;

705            tp->rcv_adv += tp->rcv_wnd;
706            tp->snd_una++;      /* SYN is acked */
707            /*
708             * If there's data, delay ACK; if there's also a FIN
709             * ACKNOW will be turned on later.
710             */
711            if (ti->ti_len != 0)
712                tp->t_flags |= TF_DELACK;
713            else
714                tp->t_flags |= TF_ACKNOW;
715            /*
716             * Received <SYN,ACK> in SYN_SENT[*] state.
717             * Transitions:
718             *  SYN_SENT  --> ESTABLISHED
719             *  SYN_SENT* --> FIN_WAIT_1
720             */
721            if (tp->t_flags & TF_SENDFIN) {
722                tp->t_state = TCPS_FIN_WAIT_1;
723                tp->t_flags &= ~TF_SENDFIN;
724                tiflags &= ~TH_SYN;
725            } else
726                tp->t_state = TCPS_ESTABLISHED;
```

tcp_input.c

图11-8 (续)

5. ACK标志是打开的

678 如果ACK标志处于开的状态，我们从图11-7中的`ti_ack`测试可以知道，ACK确认了我们的SYN。

6. 检查CCecho值是否存在

679-693 如果报文段中包含了CCecho选项，但CCecho的值与我们发出的不相等，则丢弃该报文段(除非另一端发生故障，否则，因为已经收到了对所发SYN的ACK，所以这是“决不应该发生的”)。如果我们并没有发送过CC选项(`tao_ccsent`为0)，那么就要发送RST。

7. 标记插口为已连接和处理窗口宽度选项

694-701 插口标记为已经建立连接，并对窗口宽度选项进行处理 (如果存在)。

Bob Braden的 T/TCP实现有错误，不应在测试CCecho值之前就执行这两行代码。

8. 如果未定义，则更新TAO缓存

702-704 报文段可以接受，这样，如果这个服务器的 TAO缓存还没有定义(例如客户重新启动了，或发送了一个 CCnew选项)，我们就用接收到的CCecho值(如果CCecho选项不存在，它的值为0)对其进行更新。

9. 设置rcv_adv

705-706 更新rcv_adv，如图11-4所示。在发出的SYN得到确认后，snd_una(尚未确认的最小序号数据)加1。

10. 确定是否延迟发送ACK

707-714 如果服务器在其 SYN中发送数据，那我们就延迟发送 ACK；否则，立即发出ACK(因为这很可能是三次握手中的第二个报文段)。延迟发送ACK是因为，如果服务器发来的SYN中包含了数据，那么服务器很可能正在使用 T/TCP，这样就很可能还会接收到另外的报文段，其中包含剩余的应答数据，这时就没有必要立即发送 ACK。但如果这个报文段中同时还包含有服务器的FIN(最小的三报文段T/TCP交换中的第二个报文段)，图11-18中的代码会打开TF_ACKNOW标志，以便立即发送ACK。

715-726 我们知道t_state等于TCPS_SYN_SENT，但如果隐藏状态标志TF_SENDFIN也是打开的，我们的状态实际上是 SYN_SENT*。在这种情况下，我们的状态变迁到FIN_WAIT_1状态(如果看看RFC 1644中的状态变迁图就可以看出，这实际上是两个状态变迁的组合。在SYN_SENT*状态下接收到SYN就变迁到FIN_WAIT_1*状态，而对SYN的ACK则变迁到FIN_WAIT_1状态)。

对应于图11-8开头的if的else代码如图11-9所示。它对应的是两端同时打开：我们发出了一个SYN，然后收到了一个没有ACK的SYN。这个图取代了卷2第758页的第581~582行。

tcp_input.c

```
727         } else {
728             /*
729              * Simultaneous open.
730              * Received initial SYN in SYN-SENT[*] state.
731              * If segment contains CC option and there is a
732              * cached CC, apply TAO test; if it succeeds,
733              * connection is half-synchronized.
734              * Otherwise, do 3-way handshake:
735              *        SYN-SENT -> SYN-RECEIVED
736              *        SYN-SENT* -> SYN-RECEIVED*
737              * If there was no CC option, clear cached CC value.
738              */
739             tp->t_flags |= TF_ACKNOW;
740             tp->t_timer[TCPT_REXMT] = 0;
741             if (to.to_flag & TOF_CC) {
742                 if (taop->tao_cc != 0 && CC_GT(to.to_cc, taop->tao_cc)) {
743                     /*
744                      * update cache and make transition:
745                      *        SYN-SENT -> ESTABLISHED*
746                      *        SYN-SENT* -> FIN-WAIT-1*
```

图11-9 tcp_input ：同时打开

```
747                      */
748                     tcpstat.tcps_taook++;
749                     taop->tao_cc = to.to_cc;
750                     if (tp->t_flags & TF_SENDFIN) {
751                         tp->t_state = TCPS_FIN_WAIT_1;
752                         tp->t_flags &= ~TF_SENDFIN;
753                     } else
754                         tp->t_state = TCPS_ESTABLISHED;
755                     tp->t_flags |= TF_SENDSYN;
756                 } else {
757                     tp->t_state = TCPS_SYN_RECEIVED;
758                     if (taop->tao_cc != 0)
759                         tcpstat.tcps_taofail++;
760                 }
761             } else {
762                 /* CCnew or no option => invalidate cache */
763                 taop->tao_cc = 0;
764                 tp->t_state = TCPS_SYN_RECEIVED;
765             }
766         }
```

tcp_input.c

图11-9 (续)

11. 立即ACK和关闭重传定时器

739-740 立即发出ACK，并且关闭重传定时器。尽管定时器被关闭，但由于 TF_ACKNOW 标志是设置了的，在 tcp_input快结束时调用 tcp_output。在发送ACK时，因为至少有一个数据字节(SYN)已经发出且未得到确认，重新启动重传定时器。

12. 执行TAO测试

741-755 如果报文段中包含有CC选项，那么就要执行TAO测试：缓存的值(tao_cc)必须非0，接收到的CC值必须大于缓存中的值。如果通过了 TAO测试，缓存的值就要用接收到的 CC值进行更新，这时要么从 SYN_SENT状态变迁到ESTABLISHED*状态，要么从SYN-SNET*状态变迁到FIN_WAIT_1*状态。

13. TAO测试失败或没有CC选项

756-765 如果TAO测试失败，新的状态就是 SYN_RCVD。如果没有CC选项，则TAO缓存的内容置0(未定义)，新的状态也是SYN_RCVD。

图11-10给出了标号为 trimthenstep6的代码段，是在处理 LISTEN状态结束时的一个分支(见图11-5)。这个图中的大部分代码是从卷 2第759页复制过来的。

tcp_input.c

```
767         trimthenstep6:
768           /*
769            * Advance ti->ti_seq to correspond to first data byte.
770            * If data, trim to stay within window,
771            * dropping FIN if necessary.
772            */
773           ti->ti_seq++;
774           if (ti->ti_len > tp->rcv_wnd) {
775               todrop = ti->ti_len - tp->rcv_wnd;
776               m_adj(m, -todrop);
777               ti->ti_len = tp->rcv_wnd;
778               tiflags &= ~TH_FIN;
```

图11-10 tcp_input :处理完主动或被动打开后执行的 trimthenstep6 代码段

```
779             tcpstat.tcps_rcvpackafterwin++;
780             tcpstat.tcps_rcvbyteafterwin += todrop;
781         }
782         tp->snd_wl1 = ti->ti_seq - 1;
783         tp->rcv_up = ti->ti_seq;
784         /*
785          *  Client side of transaction: already sent SYN and data.
786          *  If the remote host used T/TCP to validate the SYN,
787          *  our data will be ACK'd; if so, enter normal data segment
788          *  processing in the middle of step 5, ack processing.
789          *  Otherwise, goto step 6.
790          */
791         if (tiflags & TH_ACK)
792             goto processack;
793         goto step6;
```

tcp_input.c

图11-10 (续)

14. 是客户端则不要跳过ACK处理

784-793 如果ACK标志打开，我们就是事务过程中的客户端。也就是说，我们发送的SYN得到了ACK，我们是从SYN_SENT状态变迁到当前状态的，而不是从LISTEN状态变迁来的。在这种情况下，我们不能执行step6分支，因为那样就会跳过对ACK的处理过程(见图11-1)，而如果我们在SYN报文段中发送了数据，就需要对数据的ACK进行处理(常规的TCP会在这里跳过对ACK的处理过程，因为它从来不会随SYN一起发送数据)。

处理过程中的下一个步骤是T/TCP中新加的。通常，卷2第753页开始的switch语句中只有LISTEN和SYN_SENT状态这两个case处理代码(这两种情况我们都刚刚介绍过)。T/TCP增加了LAST_ACK、CLOSING和TIME_WAIT状态这三段case处理代码，如图11-11所示。

tcp_input.c

```
794         /*
795          * If the state is LAST_ACK or CLOSING or TIME_WAIT:
796          *  if segment contains a SYN and CC [not CCnew] option
797          *  and peer understands T/TCP (cc_recv != 0):
798          *              if state == TIME_WAIT and connection duration > MSL,
799          *                  drop packet and send RST;
800          *
801          *      if SEG.CC > CCrecv then is new SYN, and can implicitly
802          *          ack the FIN (and data) in retransmission queue.
803          *                  Complete close and delete TCPCB.  Then reprocess
804          *                  segment, hoping to find new TCPCB in LISTEN state;
805          *
806          *      else must be old SYN; drop it.
807          *      else do normal processing.
808          */
809     case TCPS_LAST_ACK:
810     case TCPS_CLOSING:
811     case TCPS_TIME_WAIT:
812         if ((tiflags & TH_SYN) &&
813             (to.to_flag & TOF_CC) && tp->cc_recv != 0) {
814             if (tp->t_state == TCPS_TIME_WAIT &&
815                 tp->t_duration > TCPTV_MSL)
816                 goto dropwithreset;
817             if (CC_GT(to.to_cc, tp->cc_recv)) {
818                 tp = tcp_close(tp);
```

图11-11 tcp_input：LAST_ACK、CLOSING和TIME_WAIT状态的初始处理

```
819                 tcpstat.tcps_impliedack++;
820                 goto findpcb;
821             } else
822                 goto drop;
823         }
824         break;                  /* continue normal processing */
825     }
```
tcp_input.c

图11-11 (续)

812-813 只有在接收到的报文段中包含了SYN和CC选项，并且我们已经有该主机的缓存CC值(cc_recv非0)，才执行接下来的特殊测试。同时知道要进入三种状态之一，TCP已发出了一个FIN，并接收到一个FIN(图2-6)。在LAST_ACK和CLOSING状态下，TCP等待对其所发FIN的ACK。所以要执行的测试是在TIME_WAIT状态下收到新的SYN时是否可以安全地截断TIME_WAIT状态，或者在LAST_ACK或CLOSING状态下收到一个新的SYN是否隐含着我们所发送FIN的ACK。

15. 如果持续时间大于MSL就不允许截断TIME_WAIT状态

814-816 通常，处于TIME_WAIT状态下的连接是允许接收新SYN的(卷2第765~766页)。这是从伯克利演变来的系统所允许的隐式截断TIME_WAIT状态，至少从NET/1以后就是这样了(卷1的习题18.5的解答就说明了这一特性)。如果连接处于TIME_WAIT状态的持续时间大于MSL，上述做法在T/TCP中是不允许的，这时要发送RST。我们在4.4节中讲到过这个限制。

16. 新SYN是现存连接的隐含ACK

817-820 如果接收到的CC值大于缓存的CC值，则TAO测试成功(即这是一个新的SYN)。这时关闭当前连接，回头执行findpcb分支，希望找到一个处于LISTEN状态的插口来处理新的SYN。图4-7给出了服务器插口的一个例子，在处理隐含的ACK时，插口处于LAST_ACK状态。

11.6 PAWS：防止序号重复

卷2第740页的PAWS测试没有变化——就是处理时间戳的代码。图11-12所示的测试在这些时间戳测试之后执行，验证接收到的CC。

tcp_input.c
```
860     /*
861      * T/TCP mechanism:
862      *   If T/TCP was negotiated, and the segment doesn't have CC
863      *   or if its CC is wrong, then drop the segment.
864      *   RST segments do not have to comply with this.
865      */
866     if ((tp->t_flags & (TF_REQ_CC | TF_RCVD_CC)) == (TF_REQ_CC | TF_RCVD_CC) &&
867         ((to.to_flag & TOF_CC) == 0 || tp->cc_recv != to.to_cc) &&
868         (tiflags & TH_RST) == 0) {
869         tcpstat.tcps_ccdrop++;
870         goto dropafterack;
871     }
```
tcp_input.c

图11-12 tcp_input：验证接收到的CC

860-871 如果使用T/TCP(TF_REQ_CC和TF_RCVD_CC选项同时打开)，这时接收到的报文段必须包含CC选项，且CC值必须等于该连接所用的值(cc_recv)；否则，报文段就是过时

重复的，要丢弃(但要给出确认，因为所有重复的报文段都需要确认)。如果报文段中包含了RST，就不丢弃，允许处理该报文段的函数稍后可以对 RST进行处理。

11.7 ACK处理

在卷2第771页上，RST处理后，如果ACK标志没有打开，报文段就被丢弃。这是常规的TCP处理过程。T/TCP改变这一点，如图11-13所示。

tcp_input.c

```
1024    /*
1025     * If the ACK bit is off: if in SYN-RECEIVED state or SENDSYN
1026     * flag is on (half-synchronized state), then queue data for
1027     * later processing; else drop segment and return.
1028     */
1029    if ((tiflags & TH_ACK) == 0) {
1030        if (tp->t_state == TCPS_SYN_RECEIVED ||
1031            (tp->t_flags & TF_SENDSYN))
1032            goto step6;
1033        else
1034            goto drop;
1035    }
```

tcp_input.c

图11-13 tcp_input ：处理没有ACK标志的报文段

1024-1035 如果ACK标志关闭，并且状态是SYN_RCVD，或者TF_SESNDSYN标志打开(即半同步)，则执行step6分支，而不是丢弃该报文段。这样做处理的是在连接建立前、但第一个SYN之后、不带ACK的数据报文段到达的情况(例如图3-9的第2报文段和第3报文段)。

11.8 完成被动打开和同时打开

如卷2的第29章一样，继续对 ACK进行处理。第774页的大部分代码还是一样的 (删除第806行)，但用图11-14的代码替代其中的813~815行。这时我们处于SYN_RCVD状态，处理的是完成三次握手的最后一个ACK。这是在服务器上对连接的常规处理过程。

1. 如果缓存的CC值未定义，就更新

1057-1064 读取这个对等端的TAO记录项，如果所缓存的CC值为0(未定义)，则用接收到的CC值更新。注意，只有在缓存的值未定义时才执行更新操作。回顾前面，图 11-6的代码在CC选项不存在时明确地将tao_cc设置为0(这样，当三次握手完成时就会进行更新)。但是如果TAO测试失败，也就不会修改 tao_cc的值。后面这个动作实际上就是收到了一个失序的SYN，不应引起缓存tao_cc的改变，如我们在图4-11中所述。

2. 变迁到新状态

1065-1074 从SYN_RCVD状态变迁到ESTABLISHED状态，是服务器完成三次握手过程的常规TCP状态变迁。因为进程已经用 MSG_EOF标志关闭了用于发送的半个连接，连接状态从SYN_RCVD*变迁到FIN_WAIT_1状态。

tcp_input.c

```
1057        /*
1058         * Upon successful completion of 3-way handshake,
1059         * update cache.CC if it was undefined, pass any queued
1060         * data to the user, and advance state appropriately.
```

图11-14 tcp_input ：被动打开或同时打开的完成

```
1061            */
1062           if ((taop = tcp_gettaocache(inp)) != NULL &&
1063               taop->tao_cc == 0)
1064               taop->tao_cc = tp->cc_recv;

1065           /*
1066            * Make transitions:
1067            *      SYN-RECEIVED  -> ESTABLISHED
1068            *      SYN-RECEIVED* -> FIN-WAIT-1
1069            */
1070           if (tp->t_flags & TF_SENDFIN) {
1071               tp->t_state = TCPS_FIN_WAIT_1;
1072               tp->t_flags &= ~TF_SENDFIN;
1073           } else
1074               tp->t_state = TCPS_ESTABLISHED;

1075           /*
1076            * If segment contains data or FIN, will call tcp_reass()
1077            * later; if not, do so now to pass queued data to user.
1078            */
1079           if (ti->ti_len == 0 && (tiflags & TH_FIN) == 0)
1080               (void) tcp_reass(tp, (struct tcpiphdr *) 0,
1081                                (struct mbuf *) 0);
1082           tp->snd_wl1 = ti->ti_seq - 1;
1083           /* fall into ... */
```

tcp_input.c

图11-14 (续)

3. 检查数据或FIN

1075-1081 如果报文段中包含有数据或FIN标志，那么在标号为dodata的代码行就要调用宏TCP_REASS(回顾图11-1)将数据交付给用户进程。卷2第790页给出了在标号dodata处对这个宏的调用，这段代码在T/TCP中没有改变。否则就要调用tcp_reass，其第二个参数为0，将队列中的所有数据交付给用户进程。

11.9 ACK处理(续)

快速重传和快速恢复算法(卷2的29.4节)保持不变。图11-15中的代码插入在卷2第779页的899~900行之间。

tcp_input.c

```
1168           /*
1169            *  If we reach this point, ACK is not a duplicate,
1170            *     i.e., it ACKs something we sent.
1171            */
1172           if (tp->t_flags & TF_SENDSYN) {
1173               /*
1174                *   T/TCP: Connection was half-synchronized, and our
1175                *   SYN has been ACK'd (so connection is now fully
1176                *   synchronized).  Go to non-starred state and
1177                *   increment snd_una for ACK of SYN.
1178                */
1179               tp->t_flags &= ~TF_SENDSYN;
1180               tp->snd_una++;
1181           }
1182       processack:
```

tcp_input.c

图11-15 tcp_input：如果TF_SENDSYN打开了，就将其关闭

1. 关闭隐藏状态标志TF_SENDSYN

1168-1181 如果隐藏状态标志TF_SENDSYN处于打开状态，则它将被关闭。这是因为接收到的ACK确认了已经发送出去的一些东西，连接已不再是半同步。因为 SYN已经得到确认，并且SYN占用了1字节序号空间，snd_una加1。

图11-16插在卷2第780~781页的926~927行之间。

tcp_input.c

```
1210          /*
1211           * If no data (only SYN) was ACK'd,
1212           *    skip rest of ACK processing.
1213           */
1214          if (acked == 0)
1215              goto step6;
```

tcp_input.c

图11-16 tcp_input ：如果没有对数据的ACK就跳过剩ACK处理过程

2. 如果没有对数据的ACK，就跳过剩余ACK处理过程

1210-1215 如果没有对数据的确认(仅对我们的SYN给出了确认)，ACK处理过程的剩余部分就跳过去。跳过去的处理代码包括打开拥塞窗口和将已得到确认的数据从发送缓存中移去。

这项测试和程序分支在T/TCP中不存在。这纠正了在14.12节的最后讨论的一个程序缺陷，在那里连接的服务器端通过发送两个背靠背段来执行慢启动，而不是一个报文段。

第2个变化如图11-17所示，用于替代卷2的图29-12中的代码。这时我们处于CLOSING状态并对ACK进行处理，处理结果是将连接的状态变迁到 TIME_WAIT。T/TCP允许截断TIME_WAIT状态(见4.4节)。

tcp_input.c

```
1266              /*
1267               * In CLOSING STATE in addition to the processing for
1268               * the ESTABLISHED state if the ACK acknowledges our FIN
1269               * then enter the TIME-WAIT state, otherwise ignore
1270               * the segment.
1271               */
1272          case TCPS_CLOSING:
1273              if (ourfinisacked) {
1274                  tp->t_state = TCPS_TIME_WAIT;
1275                  tcp_canceltimers(tp);
1276                  /* Shorten TIME_WAIT [RFC 1644, p.28] */
1277                  if (tp->cc_recv != 0 && tp->t_duration < TCPTV_MSL)
1278                      tp->t_timer[TCPT_2MSL] = tp->t_rxtcur * TCPTV_TWTRUNC;
1279                  else
1280                      tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1281                  soisdisconnected(so);
1282              }
1283              break;
```

tcp_input.c

图11-17 tcp_input :在CLOSING状态收到ACK：设置TIME_WAIT定时器

1276-1280 如果我们从对等端接收到一个CC值，并且连接的持续时间少于MSL，这时TIME_WAIT定时器就设置为当前重传超时的TCPTV_TWTRUNC(8)倍。否则，TIME_WAIT定时器设置为通常的2倍MSL。

11.10 FIN处理

TCP输入处理的接下来的三部分(更新窗口信息、紧急模式处理和接收数据处理)在T/TCP中都没有改变(回忆图11-1)。再回顾卷2的29.9节，如果设置了FIN标志，但因为序号空间的空洞，它不会得到确认，那一节中的代码就用于清除FIN标志。因此，在这里我们知道FIN是要等待确认的。

FIN处理过程的另一个变化如图11-18所示。这个修改用于替代卷2第791页的第1123行。

tcp_input.c

```
1407     /*
1408      * If FIN is received ACK the FIN and let the user know
1409      * that the connection is closing.
1410      */
1411     if (tiflags & TH_FIN) {
1412         if (TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1413             socantrcvmore(so);
1414             /*
1415              *  If connection is half-synchronized
1416              *  (i.e., TF_SENDSYN flag on) then delay the ACK
1417              *  so it may be piggybacked when SYN is sent.
1418              *  Else, since we received a FIN, no more
1419              *  input can be received, so we send the ACK now.
1420              */
1421             if (tp->t_flags & TF_SENDSYN)
1422                 tp->t_flags |= TF_DELACK;
1423             else
1424                 tp->t_flags |= TF_ACKNOW;
1425             tp->rcv_nxt++;
1426         }
```

tcp_input.c

图11-18 tcp_input：确定是否迟延发送FIN的ACK

1. 决定是否延迟发送ACK

1414-1424 如果连接是半同步的(隐藏状态标志TF_SENDSYN打开)，ACK就要延迟发送，试图在数据报文段中捎带ACK。这是一种典型的情况，处于LISTEN状态的T/TCP服务器收到SYN，这样图11-5中的代码就会设置TF_SENDSYN标志。注意，图中代码已将延迟发送ACK的标志打开，但这里是TCP要根据已经设置了的FIN标志确定怎样去做。如果TF_SENDSYN标志没有打开，则ACK不能延迟。

常规的变迁是从FIN_WAIT_2状态到TIME_WAIT状态，在T/TCP中这也需要修改，以便使TIME_WAIT状态可能被截断(见4.4节)。图11-19给出了这些修改，用于取代卷2第792页的1142~1152行。

tcp_input.c

```
1443             /*
1444              * In FIN_WAIT_2 state enter the TIME_WAIT state,
1445              * starting the time-wait timer, turning off the other
1446              * standard timers.
1447              */
1448         case TCPS_FIN_WAIT_2:
1449             tp->t_state = TCPS_TIME_WAIT;
1450             tcp_canceltimers(tp);
1451             /* Shorten TIME_WAIT [RFC 1644, p.28] */
```

图11-19 tcp_input：变迁到TIME_WAIT状态以便可能截断超时间隔

```
1452            if (tp->cc_recv != 0 && tp->t_duration < TCPTV_MSL) {
1453                tp->t_timer[TCPT_2MSL] = tp->t_rxtcur * TCPTV_TWTRUNC;
1454                /* For transaction client, force ACK now. */
1455                tp->t_flags |= TF_ACKNOW;
1456            } else
1457                tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1458            soisdisconnected(so);
1459            break;
```

tcp_input.c

图11-19 (续)

2. 设置TIME_WAIT超时间隔

1451-1453 如图11-17所示，只有当我们从对等端收到一个 CC选项，并且连接时间短于MSL时，TIME_WAIT状态才能截断。

3. 强迫立即发送FIN的ACK

1454-1455 这个变迁通常是在T/TCP的客户端，当收到服务器的响应以及服务器的 SYN和FIN时发生的。服务器的FIN应该立即给出ACK，因为两端都已经发送了 FIN，已经没有理由再延迟发送ACK了。

在两个地方要重启动 TIME_WAIT定时器：处于TIME_WAIT状态时接收到ACK和处于TIME_WAIT状态时接收到FIN(卷2第784页和第792页)。T/TCP没有修改这些代码。这表明，即使状态 TIME_WAIT被截断，如果在这时收到重复的 ACK或FIN，定时器就要在2MSL时重启动，而不是在截断后的值。重启动定时器所需要的信息在截断后的值时也能得到(即控制块)，但是由于对等端必须重传，更保守的做法是不截断TIME_WAIT状态。

11.11 小结

T/TCP所做的修改大部分都是在 `tcp_input`中，并且其中的大部分修改都与打开新连接有关。

在LISTEN状态收到SYN时要执行TAO测试。如果报文段通过了这个测试，报文段就不是过时的重复报文段，三次握手也就不需要了。在 SYN_SENT状态收到SYN时，CCecho选项(如果存在)就用于验证该 SYN不是过时的重复报文段。当处于 LAST_ACK、CLOSING和TIME_WAIT状态收到SYN时，很有可能SYN是一个隐含的ACK，可以完成现存连接的关闭。

当主动关闭一个连接时，如果连接的持续时间短于 MSL，则TIME_WAIT状态被截断。

第12章 T/TCP实现：TCP用户请求

12.1 概述

tcp_usreq函数处理来自插口层的所有 PRU_*xxx*请求。在本章中我们仅仅介绍PRU_CONNECT、PRU_SEND和PRU_SEND_EOF请求，因为T/TCP中只对这三个请求做了修改。我们也会介绍 tcp_usrclosed函数，当进程发送完数据时要调用这个函数。还有tcp_sysctl函数也会介绍，它用来处理新的TCP中的sysctl变量。

我们不打算介绍tcp_ctloutput函数(见卷2的30.6节)所需的修改，这个函数用于设置和读取两个新的插口选项：TCP_NOPUSH和TCP_NOOPT。所需的修改是非常细微具体的，只要阅读源代码就很容易理解。

12.2 PRU_CONNECT请求

在Net/3中，大约需要 25行代码 (卷2第808~809页)来处理 tcp_usrreq发出的PRU_CONECT请求。在T/TCP，大部分这些代码都移到了tcp_connect函数中(下一节介绍)，只留下了图12-1所给出的代码。

tcp_usrreq.c

```
137     case PRU_CONNECT:
138         if ((error = tcp_connect(tp, nam)) != 0)
139             break;
140         error = tcp_output(tp);
141         break;
```

tcp_usrreq.c

图12-1 PRU_CONNECT 请求

137-141 tcp_connect执行连接建立所需的步骤，tcp_output发出SYN报文段(主动打开)。

当某个进程调用connect时，即使本地主机和待连接的对等端主机都支持 T/TCP，仍然要经历正常的三次握手过程。这是因为不可能用 connect函数传递数据，这样tcp_output就仅仅发送 SYN。为了跳过三次握手过程，应用程序必须避免使用 connect，而是使用sendto或sendmsg，并给定数据和对等端服务器的地址。

12.3 tcp_connect函数

新的tcp_connect函数执行主动打开所需的处理步骤。当进程调用 connect (PRU_CONNECT请求)或者当进程调用sendto或sendmsg时，要改为调用该函数，指定待连接的对等端地址(PRU_SEND和PRU_SEND_EOF请求)。tcp_connect的第一部分在图12-2中给出。

1. 绑定本地端口

308-312 nam指向一个Internet插口地址结构，其中包含待连接的服务器的 IP地址和端口号。

如果还没有给插口指定一个本地端口(通常的情况)，调用in_pcbbind就会分配一个端口(卷2第558页)。

2. 指定本地地址，检查插口对的唯一性

313-323 如果还没有给插口绑定一个本地IP地址(通常的情况下)，调用in_pcbladdr就可分配本地IP地址。in_pcblookup查找匹配的PCB，如果找到，就返回一个非空指针。仅仅在进程绑定了一个专门指定的本地端口时才可能找到一个匹配的PCB，因为如果in_pcbbind选择本地端口，就会选择一个目前不在使用的本地端口。但是在T/TCP中，更有可能的是一个客户端进程为一系列事务绑定同一个本地端口(见4.2节)。

3. 存在已有连接；检查TIME_WAIT状态是否可以截断

324-332 如果找到一个匹配的PCB，进行下面的三项测试：

1) PCB是否处于TIME_WAIT状态；

2) 连接持续时间是否短于MSL；

3) 连接是否使用T/TCP(也就是说，是否从对等端收到了一个CC选项或CCnew选项)。

如果上述这三个条件同时为真，则调用tcp_close关闭现有的PCB。这就是我们在4.4节中讨论过的，当一个新的连接再次使用同一插口对并执行一次主动打开时，TIME_WAIT状态的截断。

4. 在互连网PCB中完成插口对

333-336 如果本地地址还是通配符，则in_pcbladdr计算出的值存储在PCB中。外部地址和外部端口也存储在PCB中。

图12-2中的步骤与图7-5中的最后一部分相似。tcp_connect的最后一部分在图12-3中给出。这段代码与卷2第808~809页PRU_CONNECT请求的最后一部分相似。

tcp_usrreq.c

```
295 int
296 tcp_connect(tp, nam)
297 struct tcpcb *tp;
298 struct mbuf *nam;
299 {
300     struct inpcb *inp = tp->t_inpcb, *oinp;
301     struct socket *so = inp->inp_socket;
302     struct tcpcb *otp;
303     struct sockaddr_in *sin = mtod(nam, struct sockaddr_in *);
304     struct sockaddr_in *ifaddr;
305     int     error;
306     struct rmxp_tao *taop;
307     struct rmxp_tao tao_noncached;

308     if (inp->inp_lport == 0) {
309         error = in_pcbbind(inp, NULL);
310         if (error)
311             return (error);
312     }
313     /*
314      * Cannot simply call in_pcbconnect, because there might be an
315      * earlier incarnation of this same connection still in
316      * TIME_WAIT state, creating an ADDRINUSE error.
317      */
318     error = in_pcbladdr(inp, nam, &ifaddr);
319     oinp = in_pcblookup(inp->inp_head,
```

图12-2 tcp_connect 函数：第一部分

```
320                         sin->sin_addr, sin->sin_port,
321                         inp->inp_laddr.s_addr != INADDR_ANY ?
322                             inp->inp_laddr : ifaddr->sin_addr,
323                         inp->inp_lport, 0);
324     if (oinp) {
325         if (oinp != inp && (otp = intotcpcb(oinp)) != NULL &&
326             otp->t_state == TCPS_TIME_WAIT &&
327             otp->t_duration < TCPTV_MSL &&
328             (otp->t_flags & TF_RCVD_CC))
329             otp = tcp_close(otp);
330         else
331             return (EADDRINUSE);
332     }
333     if (inp->inp_laddr.s_addr == INADDR_ANY)
334         inp->inp_laddr = ifaddr->sin_addr;
335     inp->inp_faddr = sin->sin_addr;
336     inp->inp_fport = sin->sin_port;
```

tcp_usrreq.c

图12-2 (续)

tcp_usrreq.c

```
337     tp->t_template = tcp_template(tp);
338     if (tp->t_template == 0) {
339         in_pcbdisconnect(inp);
340         return (ENOBUFS);
341     }
342     /* Compute window scaling to request.  */
343     while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
344            (TCP_MAXWIN << tp->request_r_scale) < so->so_rcv.sb_hiwat)
345         tp->request_r_scale++;
346     soisconnecting(so);
347     tcpstat.tcps_connattempt++;
348     tp->t_state = TCPS_SYN_SENT;
349     tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;
350     tp->iss = tcp_iss;
351     tcp_iss += TCP_ISSINCR / 4;
352     tcp_sendseqinit(tp);
353     /*
354      * Generate a CC value for this connection and
355      * check whether CC or CCnew should be used.
356      */
357     if ((taop = tcp_gettaocache(tp->t_inpcb)) == NULL) {
358         taop = &tao_noncached;
359         bzero(taop, sizeof(*taop));
360     }
361     tp->cc_send = CC_INC(tcp_ccgen);
362     if (taop->tao_ccsent != 0 &&
363         CC_GEQ(tp->cc_send, taop->tao_ccsent)) {
364         taop->tao_ccsent = tp->cc_send;
365     } else {
366         taop->tao_ccsent = 0;
367         tp->t_flags |= TF_SENDCCNEW;
368     }
369     return (0);
370 }
```

tcp_usrreq.c

图12-3 tcp_connect 函数：第二部分

5. 初始化IP和TCP首部

337-341 tcp_template分配一个mbuf，用于缓存IP和TCP首部，并用尽可能多的信息来初始化这两个首部。

6. 计算窗口宽度因子

342-345 计算接收缓存的窗口宽度值。

7. 设置插口和连接的状态

346-349 soisconnecting在插口状态变量中设置特定的一些标志位，并设置TCP连接的状态为SYN_SENT(如果进程给出MSG_EOF标志，并调用sendto或者sendmsg，而不是调用connect，我们很快就会看到tcp_usrclosed设置TF_SENDSYN隐藏状态标志，连接状态变迁到SYN_SENT*)。连接建立定时器初始化为75秒。

8. 初始化序号

350-352 从全局变量tcp_iss中复制初始发送序号，然后该全局变量值要增加，即加上除以4后的TCP_ISSINCR。发送序号由tcp_sendseqinit初始化。

我们在3.2节中讨论过的ISS随机化在宏TCP_ISSINCR中实现。

9. 生成CC值

353-361 读取对等端的TAO缓存记录项。全局变量tcp_ccgen值加上CC_INC(见8.2节)后存储在T/TCP的变量tcp_ccgen中。如同我们以前所述，不论是否使用了CC选项，主机每建立一个连接，tcp_ccgen就要加1。

10. 确定是否使用CC或CCnew选项

362-368 如果对应这个主机的TAO缓存(tao_ccsent)非0(说明与该主机之间已经不是第一次连接)，并且cc_send的值大于或等于tao_ccsent(CC值还没有回到0，继续循环)，这时发出一个CC选项并用新的CC值更新TAO缓存。否则发送一个新的CCnew选项，并将tao_ccsent设置为0(即未定义)。

回想图4-12中，那里的情况可以作为上述if条件中的第二部分不成立的一个实例：最后一次发送这个主机的CC值是1(tao_ccsent)，但tcp_ccgen(对这个连接来说，变为cc_send)的当前值是2 147 483 648。这样，T/TCP就必须发送CCnew选项而不是CC选项，因为如果我们发出的CC选项值为2 147 483 648，而对方主机还在其缓存中记着我们上次发送的CC值(即1)，那个主机会强制执行三次握手操作，因为CC值已经回到0并继续循环。对方主机无法区分CC值为2 147 483 648的SYN是否是一个过时的重复报文段。而且，如果我们发送了CC选项，即使三次握手过程顺利完成，对方主机也不会更新对应于本主机的缓存记录项(请再看看图4-12)。如果发送的是CCnew选项，客户端强制执行三次握手操作，并且会使服务器在三次握手操作完成后更新对应于本主机的缓存值。

Bob Braden的T/TCP实现是在tcp_output中测试是发送CC选项还是CCnew选项，而不是在这个函数中。这就导致了一个微小的缺陷，见下面的解释[Olah 1995]。考虑图4-11，但假定报文段1被中途的某个路由器丢弃。报文段2~4如图所示，从客户端口1 601发起的连接成功地建立。客户端发出的下一个报文段是重传的报文段1，但其中包含一个取值为15的CCnew选项。假设该报文段成功地收到，服务器强制执行三次握手，完成以后，服务器将对应于该客户端的CC缓存值更新为15。如果此后

网络交付了一个过时的重复报文段2，其中的CC值为5000，服务器收到后就会收下。解决的方法是在客户端执行主动打开时判断是发送CC选项还是CCnew选项，而不是在tcp_output函数中发送报文段时判断。

12.4 PRU_SEND和PRU_SEND_EOF请求

在卷2第811页中，对PRU_SEND请求的处理仅仅是先调用sbappend，然后再调用tcp_output。在T/TCP中，对这个请求的处理还是一样，只是代码中加上了对PRU_SEND_EOF请求的处理，如图12-4所示。我们可以看到，对TCP，PRU_SEND_EOF请求是在指定了MSG_EOF标志(见图5-2)并且当最后一个mbuf发送给协议时由sosend产生的。

tcp_usrreq.c

```
189     case PRU_SEND_EOF:
190     case PRU_SEND:
191         sbappend(&so->so_snd, m);
192         if (nam && tp->t_state < TCPS_SYN_SENT) {
193             /*
194              * Do implied connect if not yet connected,
195              * initialize window to default value, and
196              * initialize maxseg/maxopd using peer's cached
197              * MSS.
198              */
199             error = tcp_connect(tp, nam);
200             if (error)
201                 break;
202             tp->snd_wnd = TTCP_CLIENT_SND_WND;
203             tcp_mssrcvd(tp, -1);
204         }
205         if (req == PRU_SEND_EOF) {
206             /*
207              * Close the send side of the connection after
208              * the data is sent.
209              */
210             socantsendmore(so);
211             tp = tcp_usrclosed(tp);
212         }
213         if (tp != NULL)
214             error = tcp_output(tp);
215         break;
```

tcp_usrreq.c

图12-4 PRU_SEND和PRU_SEND_EOF请求

1. 隐式连接建立

192-202 如果nam参数非空，进程就调用sendto或sendmsg，并指定一个对等端地址。如果连接状态是CLOSED或LISTEN，那么tcp_connect就执行隐式连接建立。初始发送窗口设置为4 096(TTCP_CLIENT_SND_WND)，因为在T/TCP中，客户端可以在收到服务器的窗口通告以前就发送数据(见3.6节)。

2. 为连接设置初始MSS

203-204 调用tcp_mssrcvd函数时第二个参数为-1，表示我们还没有收到SYN，所以用这个主机的缓存值(tao_mssopt)作为初始MSS。当tcp_mssrcvd函数返回时，根据缓存的tao_mssopt值或系统管理员在路由表记录项中设置的值(rt_metrics结构中的rmx_mtu

成员)设置变量t_maxseg和t_maxopd的值。如果并且当收到服务器发出的带有 MSS选项的SYN时，tcp_mssrcvd将再次被tcp_dooptions调用。因为在收到对等端的 MSS选项之前就发出了数据，现在 T/TCP需要在收到SYN之前就在TCP控制块中设置MSS变量的值。

3. 处理MSG_EOF标志

205-212 如果进程指定了 MSG_EOF标志，这时 socantsendmore就要设置插口的SS_CANTSENDMORE标志。然后 tcp_usrclosed就把连接状态从 SYN_SENT(由tcp_connect设置)变迁到SYN_SENT*状态。

4. 发送第一个报文段

213-214 tcp_output检查是否应该发送报文段。在T/TCP客户端刚刚指定MSG_EOF标志调用了sendto(见图1-10)时，这个调用就发出一个报文段，其中包含 SYN、数据和FIN。

12.5 tcp_usrclosed函数

Net/3中，在处理PRU_SHUTDOWN请求时，该函数由tcp_disconnect调用。我们在图12-4中可以看到，在T/TCP中，这个函数也被PRU_SEND_EOF请求调用。图12-5给出了这个函数，替代卷2第817页中的代码。

tcp_usrreq.c

```
533 struct tcpcb *
534 tcp_usrclosed(tp)
535 struct tcpcb *tp;
536 {

537     switch (tp->t_state) {

538     case TCPS_CLOSED:
539     case TCPS_LISTEN:
540         tp->t_state = TCPS_CLOSED;
541         tp = tcp_close(tp);
542         break;

543     case TCPS_SYN_SENT:
544     case TCPS_SYN_RECEIVED:
545         tp->t_flags |= TF_SENDFIN;
546         break;

547     case TCPS_ESTABLISHED:
548         tp->t_state = TCPS_FIN_WAIT_1;
549         break;

550     case TCPS_CLOSE_WAIT:
551         tp->t_state = TCPS_LAST_ACK;
552         break;
553     }
554     if (tp && tp->t_state >= TCPS_FIN_WAIT_2)
555         soisdisconnected(tp->t_inpcb->inp_socket);
556     return (tp);
557 }
```

tcp_usrreq.c

图12-5 tcp_usrclosed 函数

541-546 在T/TCP中，通过设置TF_SENDFIN状态标志，用户在SYN_SENT或SYN_RVD状态下发起关闭过程，将状态变迁到相应的加星状态。其余的状态变迁在 T/TCP中没有改变。

12.6 tcp_sysctl函数

在为T/TCP而做修改时，用sysctl程序修改TCP变量的能力也同时加上了。T/TCP对此功能并没有严格要求，但这个功能提供了改变特定TCP变量值的一个简便方法，而不必再使用调试程序对内核进行修补。TCP变量都以前缀net.inet.tcp来标识访问。在TCP protosw结构的pr_sysctl字段中(卷2第641页)记录着指向该函数的一个指针。图12-6给出了这个函数。

570-572 目前只支持三个变量，但是很容易加上更多的变量。

tcp_usrreq.c

```
561 int
562 tcp_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
563 int    *name;
564 u_int   namelen;
565 void   *oldp;
566 size_t *oldlenp;
567 void   *newp;
568 size_t  newlen;
569 {
570     extern int tcp_do_rfc1323;
571     extern int tcp_do_rfc1644;
572     extern int tcp_mssdflt;

573     /* All sysctl names at this level are terminal. */
574     if (namelen != 1)
575         return (ENOTDIR);

576     switch (name[0]) {
577     case TCPCTL_DO_RFC1323:
578         return (sysctl_int(oldp, oldlenp, newp, newlen, &tcp_do_rfc1323));
579     case TCPCTL_DO_RFC1644:
580         return (sysctl_int(oldp, oldlenp, newp, newlen, &tcp_do_rfc1644));
581     case TCPCTL_MSSDFLT:
582         return (sysctl_int(oldp, oldlenp, newp, newlen, &tcp_mssdflt));
583     default:
584         return (ENOPROTOOPT);
585     }
586     /* NOTREACHED */
587 }
```

tcp_usrreq.c

图12-6 tcp_sysctl 函数

12.7 T/TCP的前景

有一件有趣的事，看看在RFC 1323中定义的TCP修改方案的普及，实际上是关于窗口宽度和时间戳选项的变化。这些变化受日益增长的网络速度(T3电话线路和FDDI)以及潜在的长时延路由(卫星线路)等的驱动。Thomas Skibo为SGI工作站所完成的修改是最早的实现之一。然后他又在伯克利Net/2版中做了这些修改，使这些修改在1992年5月可以公开得到(图1-16中详细给出了各个BSD版本之间的区别及其发行时间)。大约一年以后(1993年4月)，Bob Braden和Liming Wei公布了SunOS 4.1.1中类似于RFC 1323的源码修改。1993年8月，伯克利把Skibo的修改加到了4.4BSD版中，这使公众在1994年4月可以得到4.4BSD-Lite版。到1995年，有一些销售商已经加上了对RFC 1323的支持，另有一些销售商则宣称准备加上对RFC 1 323的支

持。但RFC 1 323并不是很通用的，特别是PC机上的实现(事实上，在14.6节中我们会看到，只有不到2%的客户遇到过发送窗口宽度和时间戳选项的特殊WWW服务器)。

T/TCP很可能会走类似的路。1994年9月的第一次实现(见1.9节)只是对SunOS 4.1.3的源码做了修改，大多数用户对此都不是很感兴趣，除非他们在使用 SunOS的源码。然而这只不过是T/TCP设计者的一个参考实现。普遍存在的 80×86硬件平台上的FreeBSD实现(引入了SunOS源码中的修改部分)在1995年的早期就可以公开得到了，它应该会将T/TCP传播到很多的用户。

本书这部分章节的目的是用T/TCP实例来说明为什么T/TCP 是对TCP的很有价值的改进，给出文档的细节并解释源码的变化。如同 RFC 1323中的修改一样， T/TCP实现与非T/TCP实现可以互通，仅仅当两端同时都支持CC选项时才使用它。

12.8 小结

`tcp_connect`函数是新的，已经有了T/TCP所需的修改，显式`connect`要调用它，隐式连接建立(指定目标地址的`sendto`或者`sendmsg`)也调用它。如果连接使用的是T/TCP，并且持续时间短于MSL，则该函数允许还处于TIME_WAIT状态的连接再次建立新连接。

`PRU_SEND_EOF`请求是新的，它在最后一次调用协议输出并且应用程序指定了`MSG_EOF`标志时由插口层产生。该请求允许采用隐式连接建立，并且在指定了`MSG_EOF`标志时还调用`tcp_usrclosed`。

对`tcp_usrclosed`函数所做的唯一修改是允许一个进程可以关闭尚处于 SYN_SENT或SYN_RCVD状态的连接。这时要设置隐藏标志`TF_SENDFIN`。

第二部分　TCP的其他应用

第13章　HTTP：超文本传送协议

13.1　概述

超文本传送协议(Hypertext Transfer Protocol，HTTP)是万维网(World Wide Web，WWW，也简称为Web)的基础。本章我们介绍HTTP协议，在下一章讨论一个实际的Web服务器的运作，它综合运用了卷1和卷2中的许多有关实际应用的内容。但本章并不介绍Web或如何使用Web浏览器。

NFSnet骨干网提供的统计数据(见图13-1)表明，自1994年1月以来，使用HTTP协议的增长速度令人吃惊。

月份	HTTP	NNTP	FTP 数据	Telnet	SMTP	DNS	分组数 $\times 10^9$
1994.01	1.5 %	8.8 %	21.4 %	15.4 %	7.4 %	5.8 %	55
1994.04	2.8	9.0	20.0	13.2	8.4	5.0	71
1994.07	4.5	10.6	19.8	13.9	7.5	5.3	74
1994.10	7.0	9.8	19.7	12.6	8.1	5.3	100
1995.01	13.1	10.0	18.8	10.4	7.4	5.4	87
1995.04	21.4	8.1	14.0	7.5	6.4	5.4	59

图13-1　NFSnet骨干网上各种协议的分组数量百分比

以上这些百分数是基于分组数量统计得来的，而不是基于字节数的 (这些统计数据均可从`ftp://ftp.merit.edu/statistics`获得)。随着HTTP协议所占百分比的上升，FTP和Telnet的比例在下降。同时我们注意到，分组的总数量在整个 1994年都在上升，而1995年初开始下降。这是因为1994年12月有其他的骨干网开始取代NFSnet骨干网。不过，分组数的百分比仍然有效，它表明使用HTTP协议的通信量在增加。

Web的简单结构如图13-2所示。

如上图示，Web客户(通常称为浏览器)与Web服务器使用一个或多个TCP连接进行通信。知名的Web服务器端口是TCP的80号端口。Web浏览时客户端与服务器在TCP连接上进行通信，所采用的协议就是本章描述的HTTP，即超文本传送协议。我们也可看出，一个Web服务器可以通过超文本链接“指向”另一Web服务器。Web服务器上的这些链接并不是只可以指向Web服务器，还可以是其他类型的服务器，例如：一台FTP或是Telnet服务器。

尽管HTTP协议从1990就开始使用，但第一个可用的文档出现在 1993年([Berners-Lee 1993] 大致描述了HTTP协议的1.0版本)，但是该Internet草案早就过期了。虽然有新的可用的文档([Berners-Lee, Fielding和Nielsen 1995])出现，但是仍旧只是一个Internet草案。

[Berners-Lee, Connolly 1995]中描述了一种从Web服务器返回给客户进程的文档，称为HTML(超文本标记语言)文档。Web服务器还返回其他类型的文档 (图象，PostScript文件，无格式文本文件，等等)，我们将在本章的后面举例说明这些文档。

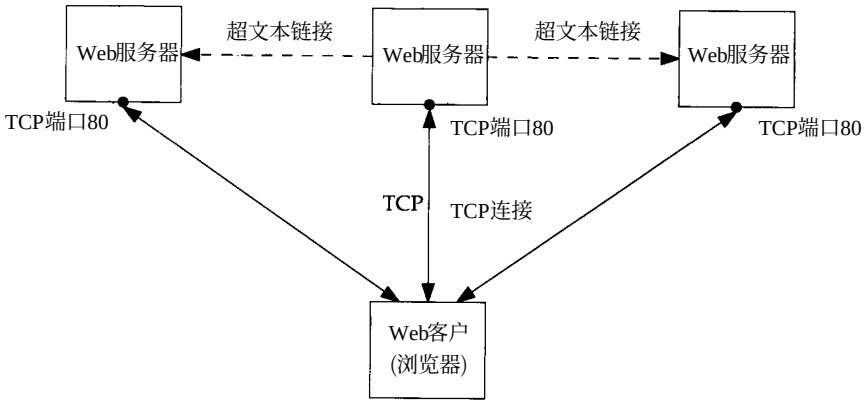


图13-2 Web客户 - 服务器结构

下一节我们将简要介绍HTTP协议和HTML文档，随后对协议进行详细描述。然后我们讨论一个流行的浏览器 (Netscape)是怎样使用该协议的， HTTP协议使用TCP的一些统计数据和HTTP协议的一些性能问题。[Stein 1995]讨论了运作一个Web站点的许多有关细节。

13.2 HTTP和HTML概述

HTTP是一个简单的协议。客户进程建立一条同服务器进程的 TCP连接，然后发出请求并读取服务器进程的响应。服务器进程关闭连接表示本次响应结束。服务器进程返回的文件通常含有指向其他服务器上文件的指针 (超文本链接)。用户显然可以很轻松地沿着这些链接从一个服务器到下一个服务器。

客户进程(浏览器)提供简单、漂亮的图形界面。 HTTP服务器进程只是简单返回客户进程所请求的文档，因此 HTTP服务器软件比HTTP客户软件要小得多。例如，NCSA版本1.3的Unix服务器由大约6 500行C代码写成，而X Window环境下的Unix Mosaic 2.5浏览器有约80 000行C代码。

我们可以用一个简单的方法来了解许多Internet协议是怎么工作的：那就是运行一个 Telnet的客户程序与相应的服务器程序通信。这种方法对 HTTP协议也是可行的，这是因为客户进程发送给服务器进程的语句包含有 ASCII命令(以回车和紧跟的换行符表示结束，称为 CR/LF)，服务器进程返回的内容也是以 ACSII字符行开始。HTTP协议使用的是8 bit的ISO Latin 1字符集，该字符集由ASCII字符及一些西欧语言中的字符组成 (以下网站可以找到各种字符集的信息：`http://unicode.drg`)。

下面是我们获取Addison-Wesley主页的例子。

```
sun % telnet www.aw.com 80        连接到服务器的80号端口
Trying 192.207.117.2...           由Telnet客户输出
Connected to aw.com.              由Telnet客户输出
Escape character is '^]'.         由Telnet客户输出
GET /                             我们只输入了这一行
<HTML>                            Web服务器输出的第一行
<HEAD>
<TITLE>AW's HomePage</TITLE>
</HEAD>
<BODY>
```

```
<CENTER><IMG SRC  = "awplogob.gif" ALT=" "><BR></CENTER>
<P><CENTER><H1>Addison-Wesley Longman</H1></CENTER>
Welcome to our Web server.
 ...                                    这里我们省略了33行输出
<DD><IMG ALIGN=bottom SRC="ball_whi.gif" ALT=" ">
Information Resource
<A HREF = "http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/MetaIndex.html">
Meta-Index</A>
 ...                                    这里我们省略了4行输出
</BODY>
</HTML>
Connection closed by foreign host.  由Telnet客户输出
```

我们只输入了GET /，服务器却返回了51行，共3611字节。这样，从该Web服务器的根目录下取得了它的主页。Telnet的客户进程输出的最后一行信息表示服务器进程在输出最后一行后关闭了TCP连接。

一个完整的HTML文档以<HTML>开始，以</HTML>结束。大部分的HTML命令都像这样成对出现。HTML文档含有以<HEAD>开始、以</HEAD>结束的首部和以<BODY>开始、以</BODY>结束的主体部分。标题通常由客户程序显示在窗口的顶部。 [Raggett, Lam, and Alexander 1996]中详细讨论了HTML。

下面这一行指定了一张图片(本例中为公司的标识)。

```
<CENTER><IMG SRC = "awplogob.gif" ALT=" "><BR></CENTER>
```

<CENTER>标志告诉客户程序将该图片放在屏幕中央，<IMG>标志含有该图片的相关信息。客户程序要取得该图片的文件名由SRC指示，ALT给出当使用纯文本客户程序时要显示的字符串(本例中是一个空字符串)。
实现强制换行。Web服务器程序返回这个主页时并不返回图片文件本身，它只返回图片文件的文件名，客户程序必须打开另一条 TCP连接来取得该文件(在本章的后面我们将看到为每一个指定的图像申请不同的连接将增加 Web的负载)。

下面这一行表示开始新的一段(<P>)。

```
<P><CENTER><H1>Addison-Wesley Longman</H1></CENTER>
```

这一段位于窗口的中央，它是第一级标题(<H1>)。客户程序可以选择怎样显示第一级标题(相对应的有2~7级标题)，通常采用比正常更大更粗的字体显示。

从上面可以看出，标记语言 (如HTML)与其他格式化语言 (如Troff, TeX, PostScript)之间的区别。HTML起源于SMGL，即标准通用标记语言 (Standard Generalized Markup Language) (http://www.sgmlopen.org包含更多的有关SGML的信息)。HTML指定了文档的数据和结构(本例中为一个1级标题)，但是没有指定浏览器怎样对文档进行格式化。

接着我们先忽略主页中跟在“ Welcome ”后面的很多问候语，看下面几行：

```
<DD><IMG ALIGN=bottom SRC="ball_whi.gif" ALT=" ">
Information Resource
<A HREF = "http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/MetaIndex.html">
Meta-Index</A>
```

其中<DD>指明一张定义表的入口。该入口以一张图片 (一个白球)开始，后面跟着文字"Information Resource Meta-Index"，最后一部分指明一个超文本链接 (<A>标志)和一个以"http://www.ncsa.uiuc.edu"开头的超文本引用(HREF属性)。像这样的超文本链接通

常在客户程序中被加上下划线或以不同的颜色来显示。当遇到上面所示的图象(公司的标志)时，服务器不会返回超链接引用的该图象或 HTML文档。客户程序通常会立即下载图象并显示在主页上，但对于超文本链接，除非用户点击(也就是说，把鼠标移到该链接上并单击)，否则客户程序通常不加处理。当用户点击了该链接，客户程序将打开一个到 www.ncsa.uiuc.edu 站点的HTTP连接，并执行GET，得到指明的文档。

类似http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/MetaIndex.html这样的表示被称为URL：统一资源定位符(Uniform Resource Locator)。URL的详细说明和意义在RFC 1738 [Berners-Lee, Masinter and McCahill 1994], 和RFC 1808 [Fielding 1995]中给出。URL是另一个重要的机制：统一资源标识符 URI(Uniform Resource Identifier)的一部分，URI还包括通用资源名称URN(Universal Resource Name)。RFC 1630 [Berners-Lee 1994]中描述了URI。URN试图比URL做得更好，但还没有制定出来。

大多数浏览器都提供查看 Web页面HTML源文件的功能，例如， Netscape和Mosaic都提供“ View Source ”的特性。

13.3 HTTP

上节的例子中，客户程序发出的GET /命令是HTTP版本0.9的命令，大多数服务器均支持这个版本(为了提供向后兼容性)。但目前HTTP的版本是1.0。因为1.0版本HTTP协议的客户程序在请求命令行中指出版本号，例如：

```
GET  /  HTTP/1.0
```

因此服务器能得知客户程序所采用的 HTTP协议的版本。本节我们将更详细地了解HTTP/1.0。

13.3.1 报文类型：请求与响应

HTTP/1.0报文有两种类型：请求和响应。 HTTP/1.0请求的格式是：

request-line
headers (0或有多个)
<blank line>
body (只对POST请求有效)

*request-line*的格式是：

*request request-URI HTTP*版本号

支持以下三种请求：

1) GET请求，返回*request-URI*所指出的任意信息。
2) HEAD请求，类似于GET请求，但服务器程序只返回指定文档的首部信息，而不包含实际的文档内容。该请求通常被用来测试超文本链接的正确性、可访问性和最近的修改。
3) POST请求用来发送电子邮件、新闻或发送能由交互用户填写的表格。这是唯一需要在请求中发送body的请求。使用POST请求时需要在报文首部Content-Length字段中指出body的长度。

对一个繁忙的 Web服务器进行采样，统计结果表明： 500 000个客户程序的请求中有

99.68%是GET请求，0.25%是HEAD请求，0.07%是POST请求。当然，如果是在一个接受比萨饼定购的站点上，POST请求的百分比将会更高。

HTTP/1.0响应的格式是：

status-line
headers (0个或有多个)
<blank line>
body

*status-line*的格式是：

*HTTP*版本号 *response-code response-phrase*

下面我们就要讨论这几个字段。

13.3.2 首部字段

HTTP/1.0的请求和响应报文的首部均可包含可变数量的字段。用一个空行将所有首部字段与报文主体分隔开来。一个首部字段由字段名 (如图13-3所示)和随后的冒号、一个空格和字段值组成，字段名不区分大小写。

报文头可分为三类：一类应用于请求，一类应用于响应，还有一类描述主体。有一些报文头(例如：Date)既可用于请求又可用于响应。描述主体的报文头可以出现在 POST请求和所有响应报文中。图13-3列出了17种不同的报文头，在[Berners-Lee, Fielding, and Nielsen 1995]中均有详细的描述。未知的报文头字段将被接收者忽略。我们讨论完响应代码后将回过头来看几个通用的报文头例子。

首部名称	请求？	响应？	主体？
Allow			•
Authorization	•		
Content-Encoding			•
Content-Length			•
Content-Type			•
Date	•	•	
Expires			•
From	•		
If-Modified-Since	•		
Last-Modified			•
Location		•	
MIME-Version	•	•	
Pragma	•	•	
Referer	•		
Server		•	
User-Agent	•		
WWW-Authenticate		•	

图13-3 HTTP报文首部的名称

13.3.3 响应代码

服务器程序响应的第一行叫状态行。状态行以 HTTP版本号开始，后面跟着 3位数字表示响应代码，最后是易读的响应短语。图 13-4列出了3位数字的响应代码的含义。根据第一位可以把响应分成5类。

使用这种3位的响应代码并不是任意的选择。我们将看到 NTTP(见图15-2)及其他的Internet应用如FTP、SMTP也使用这些类型的响应代码。

响　应	说　明
1yz	信息型，当前不用
	成功
200	OK，请求成功
201	OK，新的资源建立(post命令)
202	请求被接受，但处理未完成
204	OK，但没有内容返回
	重定向；需要用户代理执行更多的动作
301	所请求的资源已被指派为新的固定URL
302	所请求的资源临时位于另外的URL
304	文档没有修改(条件GET)
	客户差错
400	错误的请求
401	未被授权；该请求要求用户认证
403	不明原因的禁止
404	没有找到
	服务器差错
500	内部服务器差错
501	没有实现
502	错误的网关；网关或上游服务器来的无效响应
503	服务暂时失效

图13-4　HTTP 3位响应码

13.3.4　各种报文头举例

如果我们使用HTTP/1.0来获取上节列出的主页中所引用的标识图，则需要执行以下一些操作：

```
sun % telnet www.aw.com 80
Trying 192.207.117.2...
Connected to aw.com.
Escape character is '^]'.
GET /awplogob.gif HTTP/1.0                  我们输入了这一行
From: rstevens@noao.edu                     以及这一行
                                            然后输入一个空行表示请求结束
HTTP/1.0 200 OK                             服务器响应的第一行
Date: Saturday, 19-Aug-95 20:23:52 GMT
Server: NCSA/1.3
MIME-version: 1.0
Content-type: image/gif
Last-modified: Monday, 13-Mar-95 01:47:51 GMT
Content-length: 2859
                                            空行表示服务器响应头部的结束
                                        ←   这里收到了2859字节的二进制GIF图象
Connection closed by foreign host.          由Telnet客户输出
```

- 在GET请求中指出版本1.0。
- 发送一个可以被服务器记录的简单的报文头：From。

- 服务器返回的状态行给出了版本号、响应代码200和响应短语“OK”。
- Date报文头给出服务器上的时间和日期，通常是格林尼治时间。上例中服务器返回一个老式时间串。推荐的格式应是：缩写的天，日期中不含连字符， 4位数的年，如：

```
Date: Sat, 19 Aug 1995 20:23:52 GMT
```

- 服务器程序的类型和版本号分别是： NCSA Server版本1.3。
- MIME版本是1.0。在卷1的28.4节和 [Rose 1993]中有较多关于MIME的内容。
- 报文体的数据类型由Content-Type和Content-Encoding字段指出。Content-Type指出的是类型，类型后跟一‘ / ’，然后是子类型。本例中类型是 image，子类型是gif。HTTP使用的Internet媒体类型在最近的Assigned Number RFC文档中定义(本书写作时最新的文档是：[Reynolds and Postel 1994])。

其他的典型值是：

```
Content-Type: text/html
Content-Type: text/plain
Content-Type: application/postscript
```

如果报文主体是经过编码的，则 Content-Encoding报文头也会出现。例如：如果返回的报文中含有经过Unix的compress程序压缩的PostScript文件(通常带有.ps.Z后缀)，下面的两种报文头会同时出现：

```
Content-Type: application/postscript
Content-Encoding: x-compress
```

- Last-Modified指出了最后一次修改资源的时间。
- 图象文件的长度(2 859字节)在Content-Length报文头中指出。

在最后一个响应报文首部的后面，服务器程序紧跟着图象后发送了一个空行 (一个回车/换行对)。因为HTTP协议交换8 bit字节的数据，所以可以通过TCP连接发送二进制数据。这点不同于其他的Internet应用，特别是SMTP协议(卷1的第28章)，它通过TCP连接传输7 bit的ASCII字符，显式地将每字节的高位设置为0，阻止了二进制数据的交换。

User-Agent是公用的客户程序报文头，它用来标识客户程序的类型。下面是一些公用报文头的例子：

```
User-Agent: Mozilla/1.1N (Windows; I; 16bit)
User-Agent: NCSA Mosaic/2.6b1 (X11;SunOS 5.4 sun4m)  libwww/2.12 modified
```

13.3.5 例子：客户程序缓存

许多客户程序根据获取文件中的日期和时间在硬盘上缓存 HTTP文档。如果客户程序要获取的文档已存储在客户程序的缓存中，则客户程序将发送 If-Modified-Since 报文首部。这样，如果服务器程序发现该文档没有发生任何变化，就无需再发送一次该文档了。这称为条件GET请求。

```
sun % telnet www.aw.com 80
Trying 192.207.117.2...
Connected to aw.com.
Escape character is '^]'.
GET /awplogob.gif HTTP/1.0
If-Modified-Since: Saturday, 08-Aug-95 20:20:14 GMT
                                                    用空行结束客户请求
HTTP/1.0 304 Not modified
```

```
Date: Saturday, 19-Aug-95 20:25:26 GMT
Server: NCSA/1.3
MIME-version: 1.0
```

空行表示服务器响应头部的结束

```
Connection closed by foreign host.
```

上例中响应报文的响应代码为304，它表示文档没有变化。从TCP协议来看，这样做避免了将文档的主体(上例中是一个2 859字节的GIF图象)从服务器程序传送给客户程序。但是余下的TCP连接的开销(三次握手、终止连接的四个分组)还是必须的。

13.3.6 例子：服务器重定向

下面是一个服务器重定向的例子。我们试着去获取作者的主页，但是故意省略最后的“ / ”(这是用来指定目录的URL所必需的一部分)。

```
sun % telnet www.noao.edu 80
Trying 140.252.1.11...
Connected to gemini.tuc.noao.edu.
Escape character is '^]'.
GET /~rstevens HTTP/1.0
```

用空行结束客户请求

```
HTTP/1.0 302 Found
Date: Wed, 18 Oct 1995 16:37:23 GMT
Server: NCSA/1.4
Location: http://www.noao.edu/~rstevens/
Content-type: text/html
```

空行表示服务器响应头部的结束

```
<HEAD><TITLE>Document moved</TITLE></HEAD>
<BODY><H1>Document moved</H1>
This document has moved <A HREF="http://www.noao.edu/~rstevens/">here</A>.<P>
</BODY>
Connection closed by foreign host.
```

例子中响应报文的响应代码为302，表示所请求的URL已经被移动。Location报文首部指出了以“ / ”结尾的新位置。许多浏览器能自动去连接这个新的URL。但如果浏览器不愿意自动去访问这个新的URL，服务器程序也将返回一个可供浏览器显示的HTML文件。

13.4 一个例子

下面有一个使用流行的Web客户程序(Netscape 1.1N)的详细例子，通过它我们来逐一查看HTTP和TCP的使用。我们从Addison-Wesley的主页(http://www.aw.com)开始，然后到它指向的三个链接(都在www.aw.com上)，最后到卷1中描述过的页面上结束。共使用17条TCP连接，客户主机发送3 132字节，服务器主机返回47 483字节。在17条连接中，4条是为了传输HTML文档(共28 159字节)，还有13条是传输GIF图象(共19 324字节)。在进行这个会话前，先清除硬盘上的Netscape使用的缓存，迫使客户程序从服务器重新取得所有文件。我们还在客户主机上运行Tcpdump软件，记录客户程序发送和接收的所有报文段。

如我们所预期的，第一条TCP连接是访问主页(GET /)，主页的HTML文档共涉及了7个GIF图象。客户程序收到这个主页后，马上并行地打开4条TCP连接去获取前4个GIF图象。这是Netscape程序为了减少打开主页总时间的一种方法(大多数Web客户程序并不像这样，而是只能一次下载一个图象)。并行连接数量可由用户来配置，默认是4个。当这些连接中有一条结束时，客户程序会立即打开一条新的连接来获取下一个图象，直到客户程序取得全部7个图

象。图13-5表示这8条TCP连接的时间线，*y*轴是时间，单位为秒。

这8条TCP连接都由客户程序发起，依次使用1 114~1 121的8个端口号。而8条连接均由服务器程序关闭。我们把客户程序发送最初的SYN(客户的connect)看作连接的开始，客户程序收到服务器程序的FIN后发送FIN(客户的close)认为是连接的结束。取得这个主页以及它所涉及的所有7个图象共需要约12秒的时间。

下一章的图14-22中给出了由客户程序发起的第一条连接(端口 1 114)的Tcpdump分组跟踪情况。

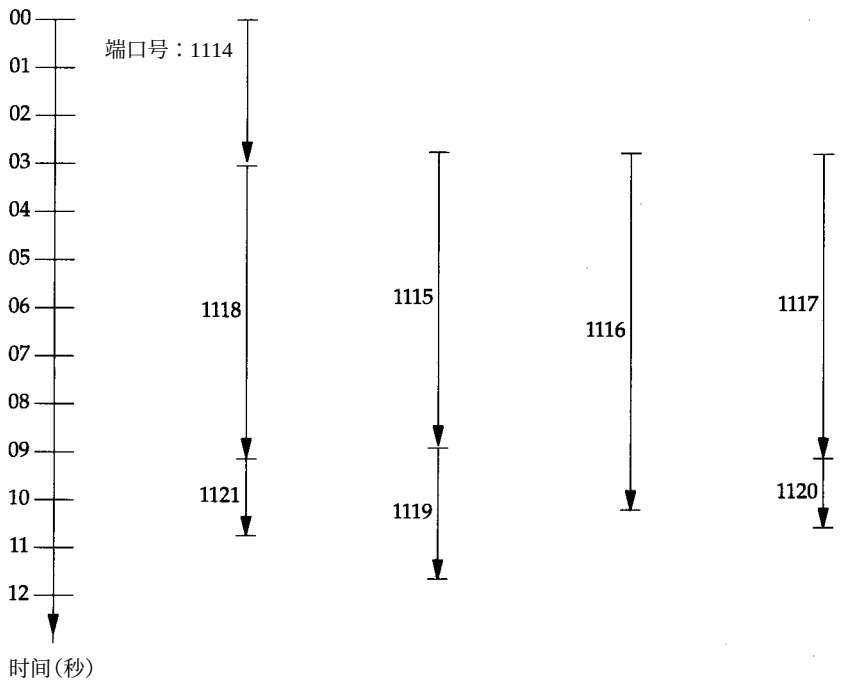


图13-5 一个主页和7个GIF图象的8条TCP连接的时间线

注意，端口号为1 115，1 116，1 117的三条连接是在第一条连接(端口号为1 114)结束之前建立的，这是因为Netscape的客户程序在读到第一条连接上的文件结束标志以后，并在关闭第一条连接之前发起三条无阻塞的连接。实际上，在图 14-22中我们可以注意到，客户程序在收到FIN标志后约半秒钟才发出FIN分组。

同时使用多条TCP连接是否真的能减少交互式用户所需的处理时间呢？为了测试这一点，我们在主机 sun上运行Netscape客户程序(图1-13)，还是来获取Addison-Wesley的主页。但这台主机是采用如今常用的方式连接Internet，即通过拔号调制解调器以28.8 Kb/s的速度连接Internet。我们修改客户程序的首选文件，对客户程序最大的连接数从1至7都进行了测试。测试时关闭了客户程序的硬盘缓存功能。在每一种最大连接数下客户程序均运行三次，取结果的平均值。图13-6是测试结果。

同时存在的连接数	总时间(秒)
1	14.5
2	11.4
3	10.5
4	10.2
5	10.2
6	10.2
7	10.2

图13-6 Web客户程序并行连接数与总时间的比较

上图可以看出，从1到4，随着连接数增加，总时间在减少。

但是如果用Tcpdump来跟踪这种交换，我们会发现，虽然用户可能把连接数设成超过 4，但是程序的极限是4。不管怎么说，超过4条连接后增加连接数对总时间即便有影响也是很小，不如从1~2，2~3，3~4那么明显。

图13-5所示的总时间比图13-6所示的最短时间(10.2秒)要多约2秒，这是因为客户主机的显示硬件速度有差异。图13-6所示的测试是客户程序运行在一台工作站上，而图13-5所示的测试客户程序运行在一台显示速度和运行速度均较慢的个人计算机上。

[Padmanabhan 1995]指出了多连接方法的两个问题。首先，这样做对其他协议不公平。例如，FTP协议获取多个文件时每次只能使用一条连接 (不包括控制连接)。其次，当在一条连接上遇到拥塞并执行拥塞避免 (在卷1的第21.6节中有描述)时，拥塞避免信息不会传递到其他连接上去。

对客户程序来说，同时对同一主机使用多条连接实际上使用的可能是同一条路径。如果处于瓶颈的路由器因为拥塞而丢弃某条连接的分组，那么其他连接的分组通过该路由器时也同样可能会被丢弃。

客户程序同时使用多个连接带来的另一个问题是容易造成服务器程序未完成的连接队列溢出，这样会使得客户主机重传它的 SYN分组而造成较大的时延。下一章我们讨论 Web服务器时，将在14.5节中详细讨论服务器程序的未完成连接队列。

13.5 HTTP的统计资料

在下一章中我们将仔细讨论TCP/IP协议族的一些特性和怎样在一个繁忙的 HTTP服务器上使用(和误用)它们。本节我们感兴趣的是一个典型的 HTTP连接到底是怎么回事。我们将使用下一章一开始要讲到的24小时的Tcpdump数据集。

图13-7列出了对近130 000个独立的HTTP连接进行统计的结果。如果客户程序非正常关闭连接，例如电话掉线等，我们就无法通过 Tcpdump的输出计算图中字节计数的中间值 (Median)或均值(Mean)或两者的值。存在一些因服务器超时而结束的连接会使图中连接持续时间的均值比正常值偏高。

	中 值	均 值
客户发送的字节数/连接	224	266
服务器发送的字节/连接	3 093	7 900
连接持续时间(秒)	3.4	22.3

图13-7 独立的HTTP连接的统计

大多数关于HTTP连接的统计均采用中间值和均值。中间值能较好地体现“正常”连接的情况，而均值则会因为少数长文件而较高。 [Mogul 1995b]中统计了200 000个HTTP连接，发现服务器返回数据量的中间值和均值分别为 1770字节和12 925字节。此文还对另一个服务器上约150万个检索进行统计，结论是返回数据量的中间值和均值分别为 958字节和2 394字节。[Braun and Claffy 1994] 列出的对NCSA服务器进行统计的结果是中间值为3 000字节，均值为17 000字节。明显可以看出服务器返回数据量的大小取决于该服务器上所提供的文件，不同的服务器之间有很大的差别。

在本节中迄今为止我们讨论的都是单个使用 TCP的HTTP连接。大多数运行 Web浏览器的用户会在一个HTTP会话期间访问给定服务器的多个文件。因为服务器可利用的信息就是客户主机的IP地址，所以要测量 HTTP会话的特性比较困难。多个用户能在同一时间利用同一客户主机访问同一个服务器。此外，还有一些组织把所有的 HTTP客户请求集中起来通过少数几个服务器(有时结合防火墙网关使用)去访问外部网的Web服务器，这样在Web服务器端看起来很多用户都在使用少数几个 IP地址(这些少数的服务器通常称为代理服务器，在 [Stein 1995]的第4章中对它进行了讨论)。不管怎么说，[Kwan, McGrath, and Reed 1995] 还是试图在NCSA服务器上对会话的特性进行测定，并定义一个会话最长持续时间为 30分钟。在这30分钟的会话中平均每个客户执行6个HTTP请求，服务器共返回95 000字节数据。

本节中提到的统计都是在服务器端进行测量的，因此结论都受服务器所提供的 HTTP文档类型的影响，例如，一个提供庞大气象图的 Web服务器平均每个HTTP会话所返回的字节数要比一个主要提供文本信息的服务器大得多。通常在 Web上跟踪大量客户程序对不同服务器的HTTP请求能获得更好的统计结果。 [Cunha, Bestavros, and Crovella 1995]中提供了一组测量数据。他们对4700个HTTP会话进行了测试，其中包括 591个不同用户对575772个文件的访问。测量的结果表明这些文件的平均长度为 11 500字节, 同时他们也提供了不同类型文件 (如HTTP、图象、声音、视频、文本等)的平均长度。通过其他测试，他们发现文件长度的分布状态曲线有一个大尾巴，即有大量的大型文件，这些文件影响了文件的平均字节数。他们发现大量被访问的是小文件。

13.6 性能问题

随着HTTP协议使用的增长 (图13-1)，它对Internet产生了广泛而重要的影响。 [Kwan, McGrath, and Reed 1995]中给出了NCSA服务器上HTTP协议一般应用的特性。 1994年，上文的作者对服务器的日志文件进行了为期五个月的检查后得出了一些结论。例如，他们注意到58%的客户请求是由个人计算机发起的，这类请求的每月增长率在 11%~14%之间。他们在文中还提供了一周中各天的请求数量、平均连接时间等统计数据。 [Braun and Claffy 1994]中提供了对NCSA服务器的其他分析。在这篇论文中，作者还讨论了 HTTP服务器可以通过缓存经常被访问的文档来提高性能。

影响交互式用户响应时间的最大因素是 HTTP协议中使用的TCP连接。前面我们看到每个要传输的文档使用一个TCP连接。[Spero 1994a]中以“ HTTP / 1.0 与TCP交互不协调 ”为标题对这个问题进行了描述。客户与服务器之间的 RTT和服务器的负载是影响响应时间的其他因素。

[Spero 1994a]也提出连接建立较慢 (卷1的20.6节中有描述)增加了时延。连接建立时间主要取决于客户请求报文和服务器的 MSS通告报文(通过Internet的客户连接，典型长度为 512或536字节)的长度。设想如果客户的请求报文小于或等于 512字节，一个MSS报文的长度为512字节，那么连接建立时间就不会长了 (但是要注意，很多基于伯克利的实现中的对 mbuf(在14.11节中有描述)的访问会引起连接建立慢的问题)。当客户的请求报文超过服务器的 MSS时，较慢的建立还要加上额外的RTT。客户请求报文的长度取决于浏览器软件。 [Spero 1994a]中提到当Xmoasic浏览器请求三个TCP报文段时发起了一个1 130字节的请求报文(这个请求报文共有42行，其中41行是Accept报文首部)。在13.4节的例子中， Netscape 1.1N浏览器共发起17

个请求，报文长度的范围是150~197字节，因此没有发生长时延的情况。图13-7列出了客户程序请求报文长度的中间值和平均值，从中可以看出大多数客户发起向服务器的请求不会引起长时延，但服务器的应答报文则会引起长时延。

我们刚刚提到，Mosaic客户程序会发出许多Accept报文首部，但这些报文首部并没有在图13-3中列出来(因为它们没有在[Berners-Lee, Fielding, and Nielsen 1995]中出现)。因为少数服务器不对这些报文首部作任何处理，所以在这个Internet草案中没有提到它们。这些报文首部的作用是告诉服务器，客户程序能接受哪些数据格式，如GIF图象、PostScript文件等。但也有少数服务器提供一个文档的不同格式的副本，而且目前还没有提供客户程序与服务器协商文档内容的方法。

另外重要的一点是：HTTP连接通常由服务器关闭，服务器经过TIME_WAIT时延后关闭连接，导致在繁忙的服务器上许多控制块停留在该状态。

[Padmanabhan 1995]和[Mogul 1995b]中建议客户与服务器保持一个打开的TCP连接，而不是服务器在发出响应后关闭连接。当服务器知道生成的响应报文的长度时才可以这样做，回想前面13.3.4节中我们提到的例子，Content-Length报文首部中指出GIF图象的大小。否则，服务器必须通过关闭连接来为客户程序指出响应的结尾。对协议作这样的修改必须同时修改客户端和服务器端。客户端规定Pragma: hold-connection报文首部，提供向后兼容的能力。如果服务器不能识别这种Pragma，就会忽略它，然后在发送完响应后关闭连接。这种Pragma允许新客户程序在尽可能情况下保持连接，同时访问新的服务器，还允许现有所有客户和服务器交互操作。

HTPP协议的下一版本(版本1.1)中可能会支持持续的连接，虽然具体怎么做可能会有变化。

在这里我们实际上提到了当前定义的三种服务器结束响应的方法。最好的办法是使用Content-Length报文首部，其次是服务器发送一个带有boundary= 属性的Content-Type报文首部([Rose 1993]的6.1.1节中给出了怎样使用这种属性的例子，但是并非所有的客户程序都支持这种特性)。最差的选择(但最广泛运用的)便是服务器关闭连接。

Padmanabhan和Mogul也提出两种新的客户请求报文，用来允许服务器流水线式的响应。这两种请求是GETALL(服务器将在单个响应内返回一个 HTML文档和所有内嵌的图象)和GETLIST(类似客户程序执行一系列的GET请求)。当客户程序确认在它的缓存中没有所要请求的任何文件时，可以使用GETALL报文。当客户程序发起对一个HTML文档的GET请求后，用GETLIST命令可以取得该HTML文档所引用的、不在缓存中的所有文件。

HTTP协议的一个重要问题是：面向字节的 TCP数据流与面向报文的HTTP服务不匹配。一种理想的解决方法是：在TCP协议之上制定一个在HTTP客户和服务器之间、单个TCP连接之上、提供面向报文接口的会话层协议。 [Spero 1994b]中描述了这样一种称为HTTP-NG的解决方法。HTTP-NG在单个TCP连接上提供多个会话。其中一个会话携带控制信息 (客户请求和服务器响应报文)，其他的会话从服务器返回所请求的文件。通过 TCP连接交换的数据包括一个8字节的会话首部(包含一些标志位、一个会话ID和所跟数据的长度)，会话首部后跟着这个会话的数据。

13.7 小结

HTTP是一个简单的协议。客户程序与服务器建立一个 TCP连接，发送请求并读回服务器的响应。服务器通过关闭连接来指示它的响应结束。服务器所返回的文件通常含有指针 (超文本链接)指向一些位于其他服务器的文件。用户可以轻松地跟随这些链接从一个服务器到另一个服务器。

客户请求是简单的 ASCII文本，服务器的响应也是以 ASCII文本开始(首部)，后面跟着数据(可以是ASCII或二进制数据)。客户程序软件 (浏览器)分析服务器的响应，并把它格式化输出，同时以高亮显示指向其他文档的链接。

通过HTTP连接传输的数据量较小。客户请求报文长度，为几百字节，服务器响应报文的典型值也在几百字节至 10 000字节间。因为一些大文档 (如图象或大的 PostScript文件)会将服务器响应报文长度的平均值拉大，所以 HTTP统计通常报告中间值。许多研究表明，服务器响应报文长度的中间值小于3000字节。

HTTP带来的最大的性能问题是每个文件使用一条 TCP连接。我们看一下 13.4节中提到的例子，为了打开一个主页，客户程序建立了 8条TCP连接。当客户请求报文的长度超过服务器通告的MSS时，缓慢的建立使每一个 TCP连接增加了额外的时延。另一个问题是：服务器进程正常关闭连接将引起在服务器主机上产生 TIME_WAIT时延，在一个繁忙的服务器上可以看到很多这种待终止的连接。

我们比较一下几乎与 HTTP协议同时开发的 Gopher协议。Gopher协议的文档号是 RFC 1436[Anklesaria et al. 1993]。从网络的观点来看， HTTP与Gopher非常相似。客户程序打开一条与服务器的连接 (Gopher使用70号端口)，并发起请求。服务器返回带有应答的响应，并关闭连接。它们的主要区别在于服务器送回给客户的报文的内容。尽管 Gopher协议允许服务器返回非文本信息，如GIF文件，但大多数Gopher客户程序是为ASCII终端设计的。因此Gopher服务器返回的文档，大多数是 ASCII文本文件。因为HTTP协议有明显的优势，所以写作本书时Internet上的许多站点已关闭了它们的 Gopher服务程序。当URL为`gopher://hostname`时，也有很多Web浏览器能识别Gopher协议，并与这些Gopher服务器通信。

HTTP协议的下一个版本 (HTTP / 1.1)将在1995年12月作为一个Internet草案公布。届时，包括认证(MD5签名)、持续的TCP连接、连接协商等方面均将有所增强。

第14章　在HTTP服务器上找到的分组

14.1　概述

本章我们将通过分析一个繁忙的 HTTP服务器上所处理的分组，从另外的角度来分析HTTP协议，同时还将对Internet协议族中的一些特性进行一般性的分析。这样我们就能把卷 1和卷2中描述的TCP/IP协议的一些特性与现实世界中的联系起来。从本章也可看到， TCP协议的行为和实现的变化很多，有时甚至明显不合理。本章有很多主题，我们把它们近似地按照TCP连接动作的顺序来安排：连接建立、数据传输和连接终止。

我们是从一个商业的`Internet`服务提供商的系统上收集数据。这个系统为 22个组织提供HTTP服务，同时运行NCSA `httpd`服务器的22个副本(我们将在下一节中讨论运行多个服务器程序)。该系统的CPU是Intel奔腾处理器，运行的操作系统是BSD / OS V1.1。

我们收集了三种数据：

1) 在连续的5天当中每小时运行一次`netstat`程序，运行该程序时带 -s选项，用来收集Internet协议维护的所有计数器。这些计数器在卷 2中我们都有介绍，如第 164页(IP)、第639页(TCP)等。
2) 在这5天当中Tcpdump程序(见卷1附录A)24小时运行，记录所有发出的和从 80端口来的带有SYN、FIN或RST标志的TCP分组。这样，我们可以详细考查TCP连接的统计结果。在这期间共收集到 686 755个符合上述条件的分组，它们分属于 147 103次TCP连接尝试。
3) 在5天的测量中，做了一次为期2.5小时的统计，记录所有发出的和从80端口来的TCP分组。因为我们可以对除了带有 SYN、FIN或RST标记以外的更多的分组进行检查，所以我们可以对少数特殊情况进行更详细的分析。在这次统计中共记录了 1 039 235个分组，平均每秒115个。

收集24小时内的SYS / FIN / RST分组的命令是：

```
$ tcpdump -p -w data.out 'tcp and port 80 and tcp[13:1] & 0x7 != 0'
```

-p标志没有把网络接口置于混合模式 (promiscuous)，所以只有运行Tcpdump程序的主机发出或接收的分组才可能被捕捉，这也正是我们所需要的。这样减少了从本地网络中收集的数据量，同时也使Tcpdump程序减少了分组的丢失。

> 这个标志没有保证非混合模式。也有人可以将网络接口设为混合模式。
>
> 在这个主机上多次长时间运行 Tcpdump，报告的分组丢失情况为：每 16 000个丢失1个至每22 000个丢失1个之间。

`-w`标志将收集结果以二进制格式存入文件，而不是以文本方式在终端上输出。这个输出文件的二进制数据随后可以用`-r`标志转换成我们所期望的文本文件。

只有发出的或从80端口来的TCP分组才被收集。此外还要求：从TCP分组首部开始算，取第13字节与数字7进行逻辑与运算，结果必须为0。这是用来测试SYN、FIN或RST标志是否被

置位(见卷1第171页)。通过收集满足上述条件的分组，然后分析 SYN和FIN上的TCP序号，我们能得到连接的每个方向上，传输数据的字节数。 Vern Paxson的`tcpdump-reduce`软件就是采用了这种简化方式(`http://town.hall.org/Archive/pub/ITA`)。

我们给出的第一张图 (图14-1)是5天中尝试连接的总数，包括主动和被动建立的连接。图中表示的是两个TCP计数器：`tcps_connattempt`和`tcps_accepts`的时间曲线，摘自卷2第639页。当为了打开连接而发送一个 SYN分组时，第一个计数器加一；当在侦听端口收到一个SYN分组时，第二个计数器加一。这些计数器对主机上的所有 TCP连接进行计数，而不只是HTTP连接。我们期望系统收到的连接请求比它发出的连接请求要多，因为系统主要提供Web服务(当然系统也用作其他用途，但主要的 TCP / IP流量是由HTTP分组组成)。

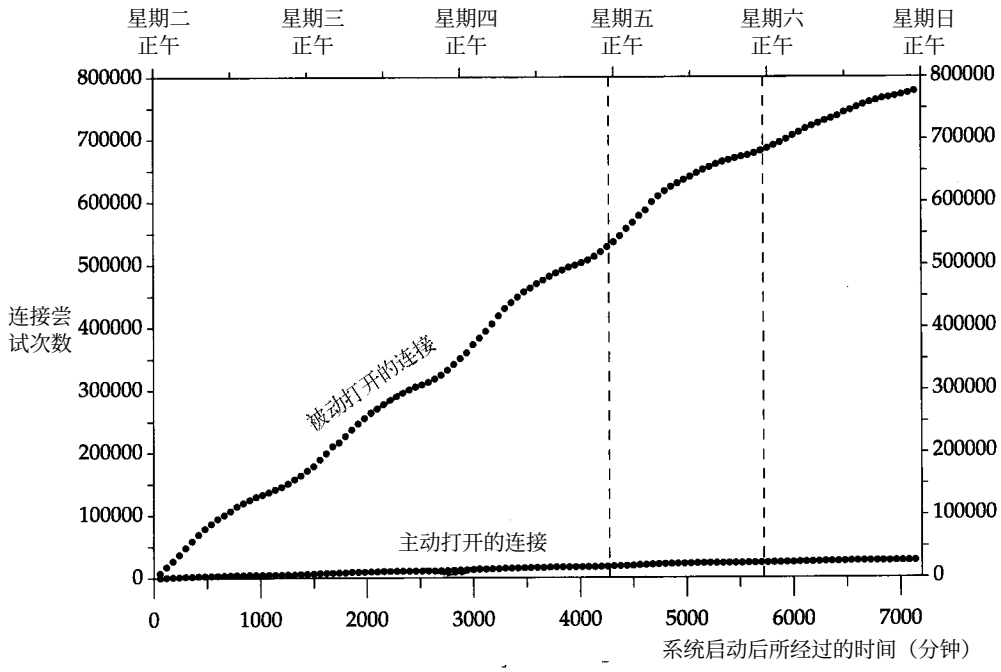


图14-1 主动与被动连接尝试次数累计

图中星期五正午附近和星期六正午附近的虚线描绘了一个 24小时周期，在这24小时中对SYN/FIN/RST分组进行了跟踪、收集。注意被动连接尝试的次数曲线，它的斜率像我们所预期的那样在正午后一直到午夜前都比较大。我们也可看出，从星期五的午夜开始到周末这段时间，曲线的斜率一直在减小。我们绘出每小时被动连接尝试次数的曲线，如图 14-2所示，从中很容易看出每天的周期性规律。

“繁忙”服务器的定义是什么？我们进行分析的系统每天收到超过150 000个TCP连接请求，这相当于平均每秒1.74个连接请求。[Braun and Claffy 1994]提供了NCSA服务器的详细情况：在1994年9月，平均每天有360 000个TCP连接请求(这个数据每6~8个星期翻一番)。[Mogul 1995b]中描述了两个被作者称为“相对繁忙”的服务器，其中一个是每天处理一百万个连接请求，而另一个则是在近3个月时间内平均每天收到40 000个连接请求。1995年6月21日的《华尔街》杂志列出了最繁忙的10个Web服务器，统计了从1995年5月1日至7日之间对它们的点击次数，最高的达每周430万次(`www.netscape.com`)，最低的每天也有30万次。说了这么多，我们还是得提醒读者注

意他们声称的Web服务器的性能和统计数据。如本章中我们所看到的，以下这些提法有很大的区别：每天点击次数、每天连接数、每天客户数和每天会话数。另一个要搞清的事实是一个组织的Web服务器程序运行在几台主机上，我们将在下一节讨论这种情况。

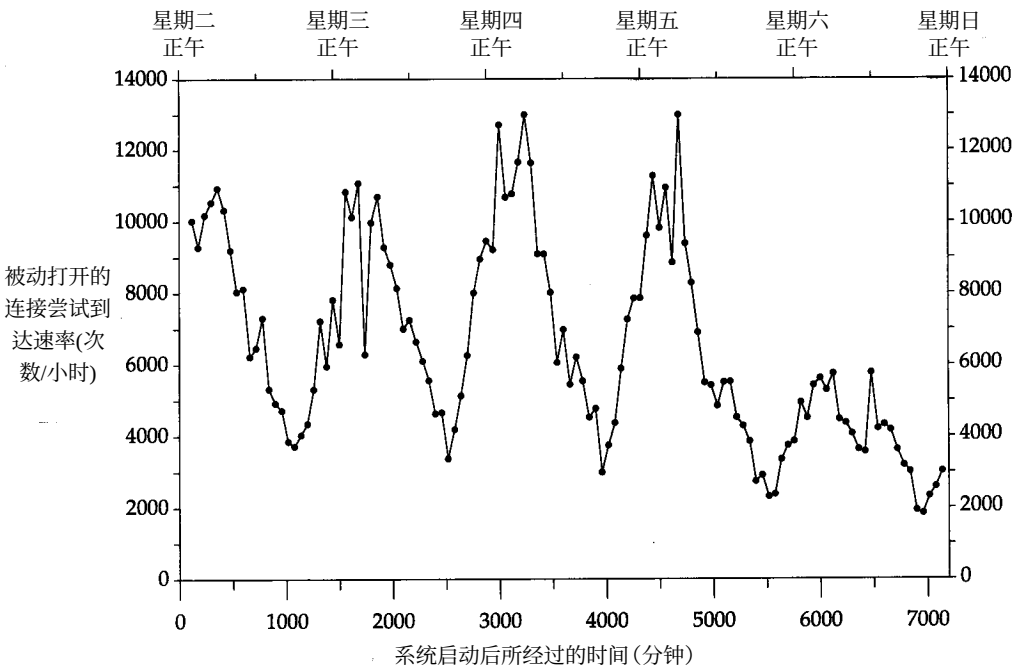


图14-2 每小时被动连接尝试次数

14.2 多个HTTP服务器

最简单的HTTP服务器安排是一台主机上运行一个 HTTP服务器程序。有很多 Web站点是这样做的，但也有两种较为普遍的变形：

1) 一台主机，多个服务器程序。本章中所分析的数据就来源于一台按这种方式运行的主机。单个主机为多个组织提供HTTP服务。每一个组织的WWW域名(www.organization.com)映射一个不同的IP地址(都在同一子网上)，单个以太网接口分别对每一个不同的 IP地址赋予别名(第6.6节中描述了Net / 3怎样允许单个网络接口上的多个 IP地址。在主IP地址之后指派给网络接口的IP地址均称为别名)。这22个httpd服务器实例中的每一个都只使用一个IP地址。当服务器程序启动时，它把本地的IP地址绑定到它的监听TCP插口上，因此它只收到那些目的地址是它的IP地址的连接。
2) 多台主机，每台均提供服务器程序的一个副本。这种技术用于繁忙的组织在多个主机上分布输入负载(即负载平衡)。对应组织的WWW域名：www.organization.com指派了多个IP地址，每一个提供HTTP服务的主机有不同的IP地址(卷1的第14章，DNS中的多条A记录)。这种组织的DNS服务器响应DNS客户请求时，必须能以不同的顺序返回多个不同的IP地址。DNS中把这个称为循环使用 (round-robin)，例如，在通常的DNS服务器程序当前版本中均支持这种功能。

   例如，NCSA提供9个HTTP服务器。我们第一次查询它们的域名服务器时，返回如下：

```
$ host -t a www.ncsa.uiuc.edu newton.ncsa.uiuc.edu
Server: newton.ncsa.uiuc.edu
Address: 141.142.6.6 141.142.2.2
www.ncsa.uiuc.edu       A       141.142.3.129
www.ncsa.uiuc.edu       A       141.142.3.131
www.ncsa.uiuc.edu       A       141.142.3.132
www.ncsa.uiuc.edu       A       141.142.3.134
www.ncsa.uiuc.edu       A       141.142.3.76
www.ncsa.uiuc.edu       A       141.142.3.70
www.ncsa.uiuc.edu       A       141.142.3.74
www.ncsa.uiuc.edu       A       141.142.3.30
www.ncsa.uiuc.edu       A       141.142.3.130
```

(host程序在卷1第14章有描述并用到了它。)上例命令中的最后一个参数是我们要查询的NCSA的DNS服务器的名字，使用该参数的原因是：在缺省情况下，host程序将使用本地DNS服务器，而本地域名服务器的缓存中可能有这9个记录，而且可能每次返回同一个IP地址。

第二次我们再运行上例中的程序时，得到了不同次序。

```
$ host -t a www.ncsa.uiuc.edu newton.ncsa.uiuc.edu
Server: newton.ncsa.uiuc.edu
Address: 141.142.6.6 141.142.2.2

www.ncsa.uiuc.edu       A       141.142.3.132
www.ncsa.uiuc.edu       A       141.142.3.134
www.ncsa.uiuc.edu       A       141.142.3.76
www.ncsa.uiuc.edu       A       141.142.3.70
www.ncsa.uiuc.edu       A       141.142.3.74
www.ncsa.uiuc.edu       A       141.142.3.30
www.ncsa.uiuc.edu       A       141.142.3.130
www.ncsa.uiuc.edu       A       141.142.3.129
www.ncsa.uiuc.edu       A       141.142.3.131
```

14.3 客户端SYN的到达间隔时间

下面我们来做一件有趣的事情：通过观察客户端SYN的到达，我们来看平均请求速率和最大请求速率之间的区别。服务器应有能力应付峰值负载，而不是平均负载。

通过对SYN / FIN / RST进行24小时跟踪，我们可以分析客户端SYN的到达时间间隔。在这个24小时的跟踪期间共有160 948个SYN到达(在本章的开头我们曾提到，在这期间有147 103次连接尝试。这中间的不同是因为SYN的重传。注意到，大约有10%的SYN须重传)。最小的到达间隔时间是0.1 ms，最大值是44.5秒，平均值是538 ms，中间值是222 ms。91%的到达间隔时间小于1.5秒。图14-3给出了到达间隔时间的柱状图。

这张图虽然有趣，但它不能提供峰值到达速率。为了测定峰值速率，我们把一天的24小时划分为1秒的时间间隔，并计算每秒的SYN到达个数(实际测量了86 622秒，比24小时长几分钟)。图14-4列出了前20个时间间隔内计数器的值。图中第二列给出了所有到达的SYN数，第三列的计数器表示的是忽略重传后的SYN到达数。在本节的最后，我们将用到第三列的数据。

例如，考虑所有到达的SYN，一天有27 868秒(一天中的32%)内没有SYN到达，22 471秒(一天中的26%)内只有一次SYN到达，等等。一秒中最大的SYN到达数为73次，一天中共有两次这种情况。我们观察所有SYN到达次数超过50次的“秒”，将发现它们都在一个3分钟的时间段内，这就是我们要找的峰值。

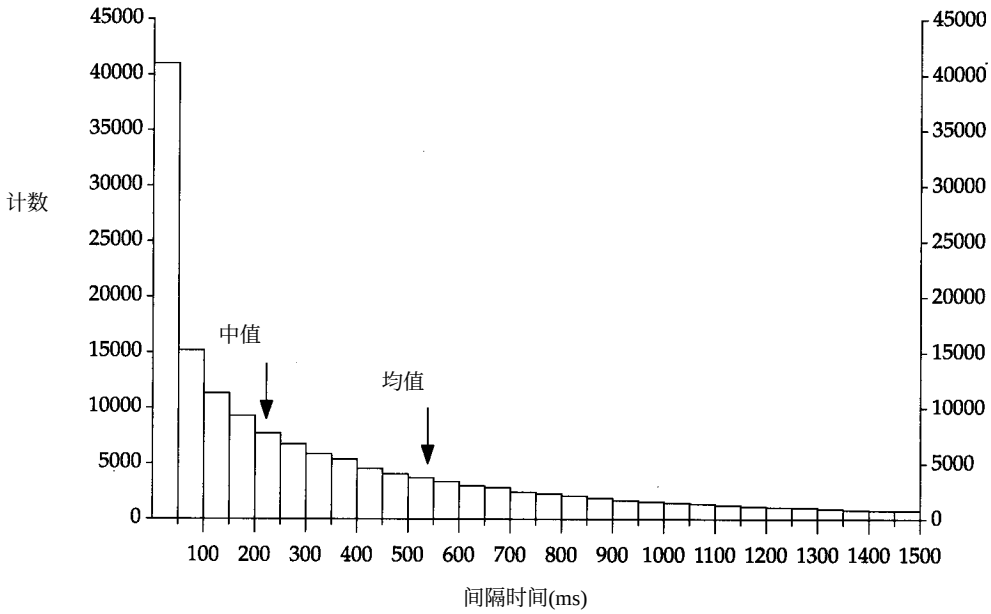


图14-3 客户端SYN的到达间隔时间分布

1秒钟内到达的SYN个数	所有SYN的累计数	新SYN的累计数
0	27 868	30 565
1	22 471	22 695
2	13 036	12 374
3	7 906	7 316
4	5 499	5 125
5	3 752	3 441
6	2 525	2 197
7	1 456	1 240
8	823	693
9	536	437
10	323	266
11	163	130
12	90	66
13	50	32
14	22	18
15	14	10
16	12	9
17	4	3
18	5	2
19	2	1
20	3	0
	86 560	86 620

图14-4 给定秒数内到达的SYN数

图14-6是含有峰值的那个小时的情况。在这个图中，我们把每 30秒到达的SYN数取平均值，*y*轴表示的是每秒到达的 SYN数，平均到达速率约为每秒 3.5个，因此，这个小时处理的到达的SYN几乎为平均值的两倍。

图14-7给出了包含峰值的那个3分钟的更详细的情况。

在这3分钟中的变化有违人们的直觉，也表明某些客户有反常行为。如果我们检查这 3分钟Tcpdump程序的输出会发现，问题果然来自一个特别的客户。在包含图 14-7最左边尖峰的30秒中，那个客户在两个不同的端口发送 1 024个SYN，平均每秒30个。有少数几秒还在 60~65次之间，再加上其他客户发送的，在图中的峰值就接近 70个。图14-7中中间的尖峰也是由这个客户引起的。

图14-5列出了与这个客户相关的部分 Tcmdump输出。

```
1   0.0                client.1537 > server.80: S 1317079:1317079(0)
                                                  win 2048 <mss 1460>
2   0.001650 (0.0016)  server.80 > client.1537: S 2104019969:2104019969(0)
                                                  ack 1317080 win 4096 <mss 512>
3   0.020060 (0.0184)  client.1537 > server.80: S 1317092:1317092(0)
                                                  win 2048 <mss 1460>
4   0.020332 (0.0003)  server.80 > client.1537: R 2104019970:2104019970(0)
                                                  ack 1317080 win 4096
5   0.020702 (0.0004)  server.80 > client.1537: R 0:0(0)
                                                  ack 1317093 win 0
6   1.938627 (1.9179)  client.1537 > server.80: R 1317080:1317080(0) win 2048

7   1.958848 (0.0202)  client.1537 > server.80: S 1319042:1319042(0)
                                                  win 2048 <mss 1460>
8   1.959802 (0.0010)  server.80 > client.1537: S 2105107969:2105107969(0)
                                                  ack 1319043 win 4096 <mss 512>
9   2.026194 (0.0664)  client.1537 > server.80: S 1319083:1319083(0)
                                                  win 2048 <mss 1460>
10  2.027382 (0.0012)  server.80 > client.1537: R 2105107970:2105107970(0)
                                                  ack 1319043 win 4096
11  2.027998 (0.0006)  server.80 > client.1537: R 0:0(0)
                                                  ack 1319084 win 0
```

图14-5 违规的客户以高速率发送无效的SYN

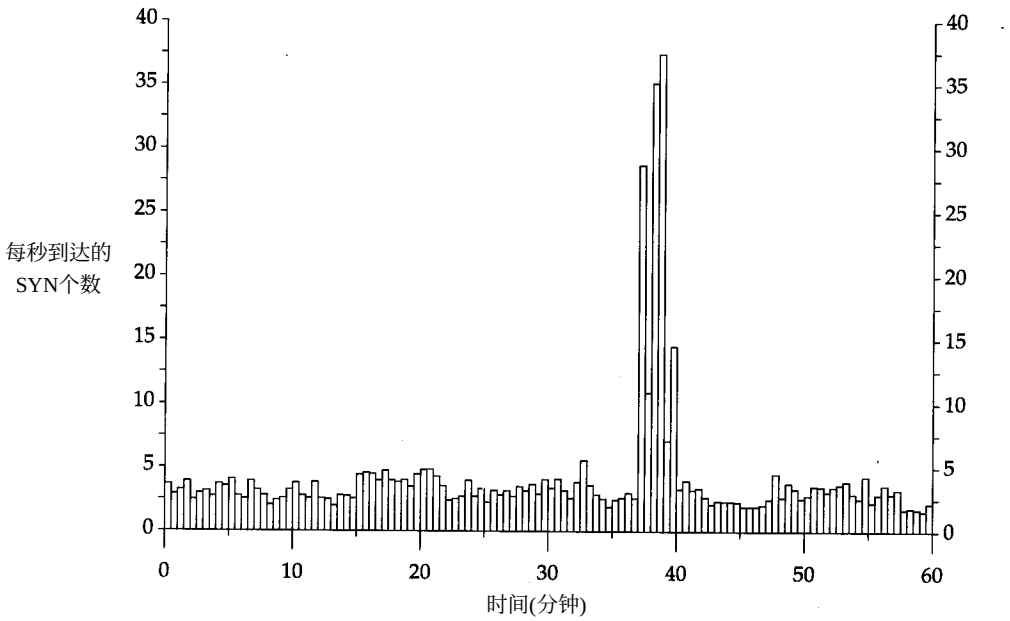


图14-6 60分钟时间内每秒到达的SYN数

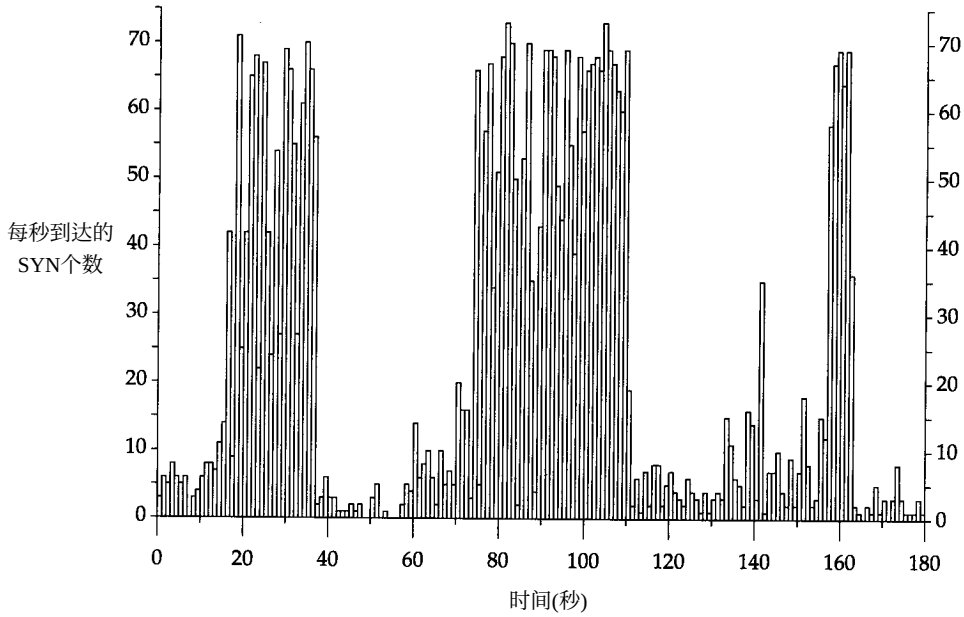


图14-7 3分钟的峰值时间内每秒到达的SYN数

第一行是表示客户的SYN，第二行是服务器的SYN / ACK。第三行是从同一个客户的同一个端口来的另一个SYN，但它的起始序列号是13，比第一行的高。第四行是服务器发送一个RST，第五行发送另一个RST，第六行是客户发送的RST。从第7行开始又重复这个情况。

为什么服务器要在一行内给客户发送两个RST(第五行和第六行)？可能是由于设有打印出来的某些数据段引起，因为遗憾的是Tcpdump跟踪程序仅包含有SYN、FIN或RST标志的报文段。然而，这个客户显然违规了，在同一个端口如此高速率地发送SYN，并且从一个SYN到下一个的序列号增加很小。

忽略重传的SYN后的计算结果

我们需要忽略重传的SYN，重新分析客户SYN的到达间隔时间。因为从上面我们可以看出，一个违反常规的客户就可以将数据拉出显著的峰值来。正如我们在本节的前面所提到的，忽略重传可以减少约10%的SYN。同样，通过考察有效的SYN，我们可以来分析连接到达服务器的速率。所有到达的SYN均影响TCP/IP协议的处理(因为每一个SYN要经过设备驱动程序、IP输入，然后才是TCP输入)，连接的到达速率影响HTTP服务器(服务器程序为每一个连接处理新的客户请求)。

在忽略重传SYN后，图14-3中的平均值由538 ms增加至600 ms，中间值由222 ms增加至251 ms。在图14-4中我们已给出每秒到达的SYN的分布图。峰值也像图14-6中表示的那样，不过要小得多。一天中到达的SYN数最大的3秒内分别为有19、21、33个SYN到达。这就给我们一个范围，从每秒4个(由到达时间间隔中值251 ms得来)到33个SYN，约为8倍的关系。这就意味着，当我们设计一个Web服务器时，应使它能适应的峰值在这种平均值之上。在14.5节中我们将看到这种入连接请求队列中的峰值到达速率的作用。

14.4 RTT的测量

下一个我们感兴趣的内容是各种客户与服务器之间的往返时间。不幸的是，我们不能通过在服务器上跟踪SYN/FIN/RST来测量它。图14-8描述了TCP三次握手和用四个报文段来终止一个连接的情况(第一个FIN由服务器发出)。加粗的线表示在跟踪SYN / FIN / RST时可以被跟踪到。

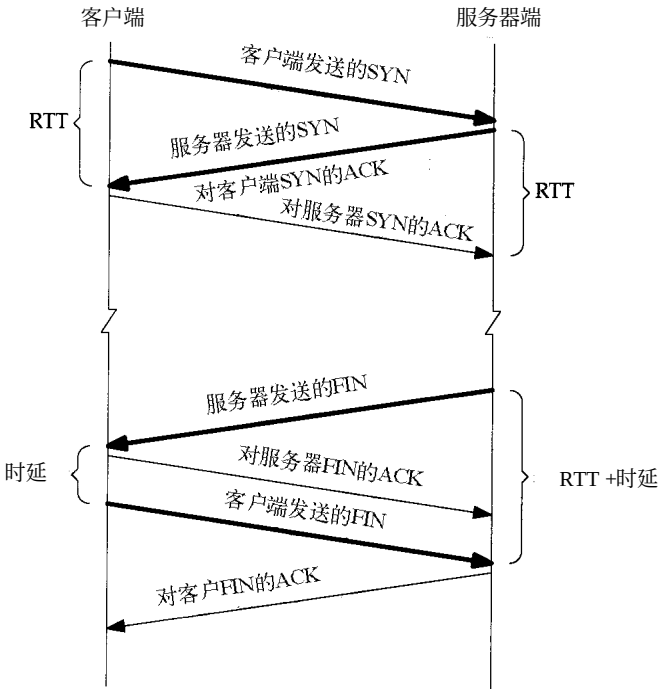


图14-8 TCP的三次握手和连接终止

在客户端可以测量RTT，即发送SYN与接收服务器发来的SYN之间的时间间隔，但我们的测量均在服务器端。我们可以通过测量服务器发送 FIN与接收客户发来的FIN之间的时间间隔来测量RTT，但是这种测量包含一个不确定的时延：客户应用程序收到文件结束标志与关闭连接之间的时间。

我们需要跟踪服务器上的所有分组来测量 RTT, 因此我们使用前面提到的2.5小时的跟踪，并测量服务器发送SYN / ACK与收到客户的ACK之间的时间间隔。客户发送的、用来确认服务器SYN的ACK报文通常不会被延迟 (卷2第758页)，因此这种测量不会包含一个时延的ACK。这些报文通常都是尽可能的小 (服务器的SYN为44字节，通常包括一个服务器上使用的MSS选项，客户的ACK为40字节)，因此在较慢的SLIP或PPP链路上也不会产生明显的时延。

在2.5小时内，进行了19 195次RTT的测量，涉及810个不同的IP地址。最小的RTT等于0(从同一主机的客户程序)，最大的RTT是12.3秒，平均值是445 ms，中间值是187 ms。图14-9给出了3秒以内的RTT的分布。98.5% 的RTT在 3秒以内。这些测量表明，由大西洋岸至太平洋岸的RTT最好的情况在60 ms左右，典型情况下的RTT值至少是这个值的三倍。

为什么中间值(178 ms)比由大西洋岸至太平洋岸的RTT(60 ms)小这么多？一种可能是目前情况下，大量的用户仍使用拨号线访问 Internet，即使是最快的调制解调器(28 800 bps)，也给每个RTT增加100~200 ms的时延。另外一个原因是，有些客户实现在处理三次握手的第三个报文段(客户发送的、用来确认服务器SYN的ACK报文)时产生了时延。

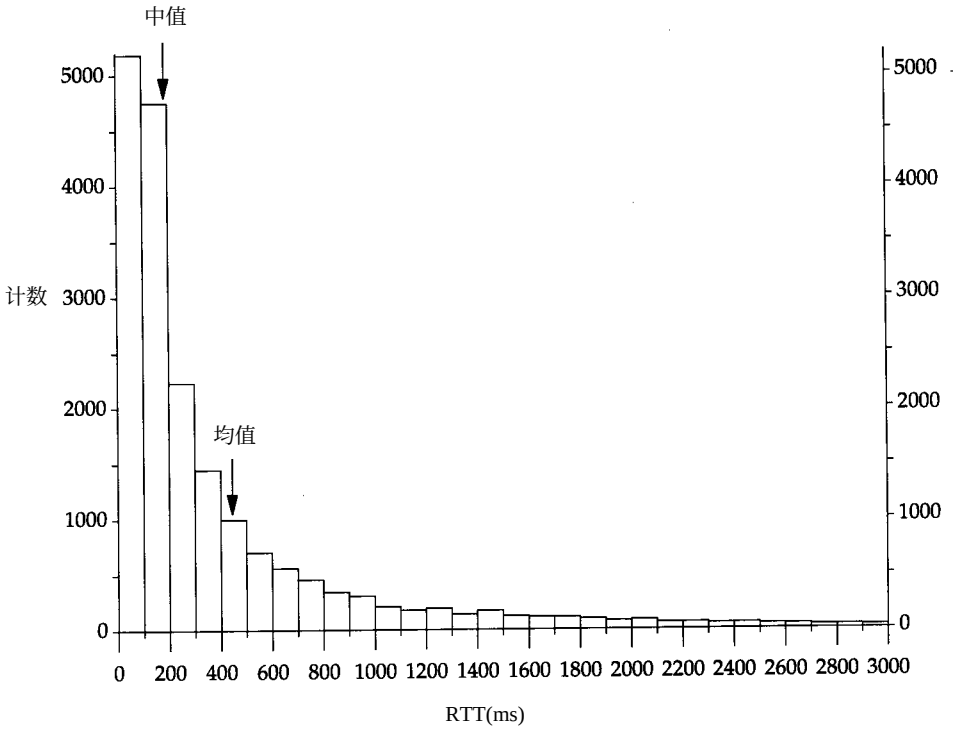


图14-9 客户往返时间的分布

14.5 用listen设置入连接队列的容量

为了准备一个接收入连接请求的插口，服务器通常执行下面的调用：

```
listen(sockfd, 5);
```

第二个参数称为*backlog*，指示listen调用的入连接队列的容量。BSD内核因为历史的原因，通过在<sys/socket.h>头文件中定义SOMAXCONN常量，将入连接队列的容量的上限设为5。如果应用程序指定了一个大于5的值，内核将不作任何提示地把它置为SOMAXCONN。新的内核将SOMAXCONN的值增加至10或更高，增加的原因我们马上要介绍。

在插口数据结构中，so_qlimit值就等于backlog参数值(卷2第365页)。当一个TCP入连接请求到达时(客户端的SYN)，TCP程序执行sonewconn调用，紧跟着进行如下测试(卷2第370页的第130~131行)：

```
if (head->so_qlen + head->so_q0len > 3 * head->so_qlimit / 2)
    return ((struct socket *)0);
```

正如卷2中所描述的，把应用程序指定的backlog乘以一个毫无根据的因子：3/2，

确实能在内核指定backlog为5时将等待的连接数增加至8。这个毫无根据的因子只在基于伯克利的实现中有作用(卷1第195页)。

这个队列长度的上限限制以下两项的和：

1) 未完成连接队列(so_q0len，一个SYN已经到达、但三次握手还没有完成的连接)中的项数。

2) 已完成连接队列(so_qlen，三次握手已完成、内核正等待进程执行accept调用)中的项数。

卷2第369页详细描述了当一个TCP连接请求到达时，服务器端处理的步骤。

当已完成连接队列被填满（例如，服务器进程或服务器主机非常繁忙时，进程执行accept调用不够快，不能及时清空队列），或未完成连接队列被填满时，将达到backlog的上限。当服务器主机与客户主机的往返时间较长，而相比较而言，新的连接请求到达较快，那么服务器就要面对上述的后一个问题，因为一个新的SYN占用队列中的一个记录项的时间是一次往返时间。图14-10描述了未完成连接队列的这部分时间。

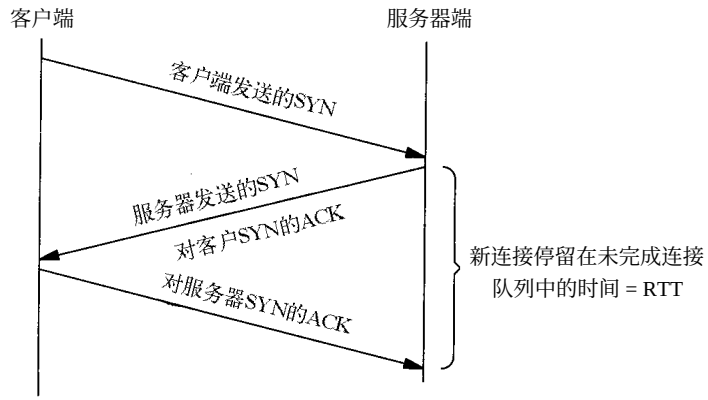


图14-10 用分组表示的未完成连接队列中一个记录项的占用时间。

为了检验未完成连接队列是否已满（不是已完成连接队列），我们使用一个被修改过的netstat程序，在最繁忙的HTTP监听服务器上连续打印so_q0len和so_qlen这两个变量的值。这个程序共运行了2个小时，进行了379 076次采样，或者说约每19 ms进行一次采样。图14-11给出了结果。

队列长度	未完成连接队列计数	已完成连接队列计数
0	167 123	379 075
1	116 175	1
2	42 185	
3	18 842	
4	12 871	
5	14 581	
6	6 346	
7	708	
8	245	
	379 076	379 076

图14-11 繁忙的HTTP服务器的连接队列长度分布

前面曾经提到，将backlog设为5时，实际上可以有8条连接在排队。已完成连接队列绝大部分时间是空的，因为当有连接进入这个队列时，只要服务器程序的accept调用一返回，这条连接便会马上从该队列中被取走。

当队列已满时，TCP丢弃入连接请求(卷2第743页)，并且假定客户程序会发生超时，重传它的SYN，希望在几秒钟以后在队列中找到空闲位置。但是Net/3的内核并不提供有关丢失的SYN的统计数据，因此系统管理员无法知道这种情况发生的频度。我们把系统中这一段代码作了如下修改：

```
if (so->so_options & SO_ACCEPTCONN) {
    so = sonewconn(so, 0);
    if (so == 0) {
        tcpstat.tcp_listendrop++;    /* new counter */
        goto drop;
    }
```

所作的修改就是增加了一个计数器。

图14-12中列出了为期5天、一小时采集一次得到的该计数器的值。这个计数器是对主机上的所有服务器程序进行统计的，但是我们假定所监视的主机主要是作为一台 Web服务器，实际上绝大多数的溢出也是发生在httpd的侦听插口上。从平均上来说，这台主机每分钟的呼入连接请求溢出刚好超过三个(22 918次溢出除以7 139分钟)，但是这里也有几个值得注意的连接丢失数量的跳跃点。大约在第4500分钟 (星期五下午4：00) 左右，一个小时内丢弃了1964个入连接请求，约为每分钟32个(每两秒钟一个)。其他两次值得注意的跳跃发生在星期二下午的早些时候。

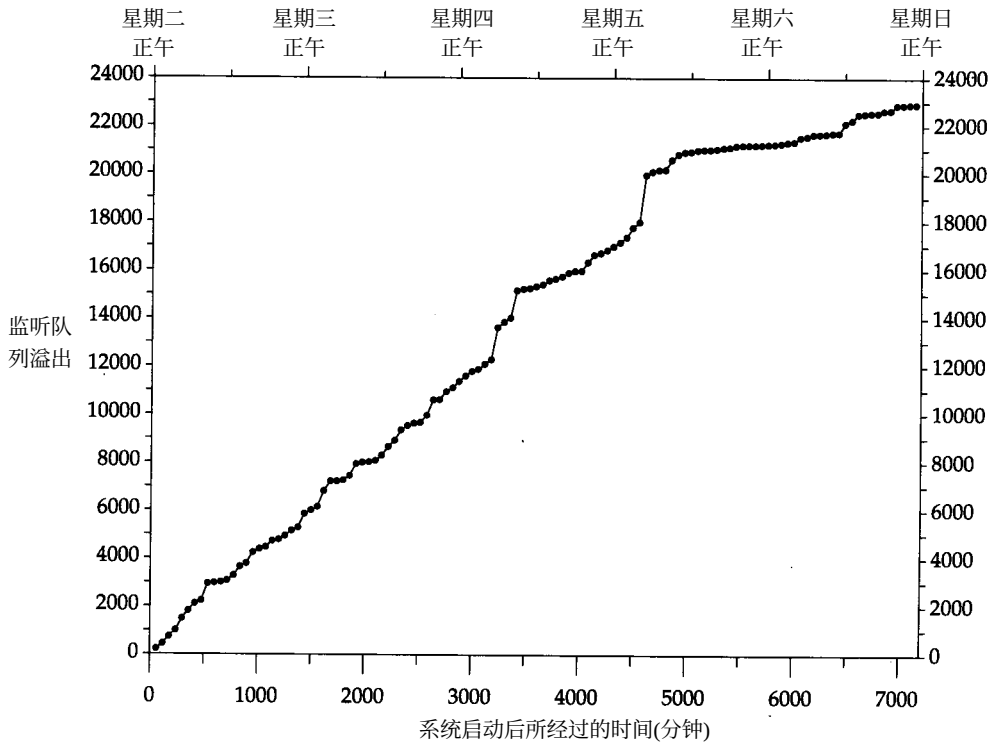


图14-12 服务器监听队列的溢出

必须增加支持繁忙服务器的内核的 backlog参数的上限，同时必须修改繁忙服务器应用程序(例如httpd)，使之设置一个更大的backlog。例如，httpd的1.3版就存在这个问题，因为它用下面的语句将backlog强制设置为5：

```
listen(s1, 5);
```

1.4版将backlog增加到了35，但这对于某些繁忙的服务器来说还是不够。

不同的厂商采用不同的方法来增加内核的 backlog的上限。例如，BSD / OS V2.0内核将somaxconn全局变量指定为16，但系统管理员可以将它调整至更大的值。Solaris 2.4允许系统管理员使用ndd程序改变TCP参数：tcp_conn_req_max，这个参数的默认值为 5，最

大可以到32。Solaris 2.5将默认值增加到32，而最大可以到1 024。不幸的是，应用程序使用listen调用时，没有一个简单的办法来确定当前操作系统内核所允许的队列最大值，所以最好的办法是应用程序的代码中给这个参数赋一个很大的值（因为使用listen调用时不会因为这个值太大而返回错误），或者让用户可以在命令行中指定这个参数。在[Mogul 1995c]提出一种思想，认为在listen调用中应忽略这个参数，而由系统内核直接把它设为最大值。

有些应用程序特意将backlog参数设为一个较低的值来限制服务器的负载，因此，在这种情况下我们要避免增加这些应用程序中的这个参数值。

SYN_RCVD错误

当我们检查netstat的输出时发现，插口在SYN_RCVD状态下保持了几分钟。Net / 3用它的连接建立定时器限制这个状态保持的时间为75秒(卷2第664页和755页)，为什么还会出现这种现象？图14-13列出了Tcpdump的输出。

```
 1     0.0                 client.4821 > server.80: S 32320000:32320000(0)
                                                    win 61440 <mss 512>
 2     0.001045 ( 0.0010) server.80 > client.4821: S 365777409:365777409(0)
                                                    ack 32320001 win 4096 <mss 512>

 3     5.791575 ( 5.7905) server.80 > client.4821: S 365777409:365777409(0)
                                                    ack 32320001 win 4096 <mss 512>
 4     5.827420 ( 0.0358) client.4821 > server.80: S 32320000:32320000(0)
                                                    win 61440 <mss 512>
 5     5.827730 ( 0.0003) server.80 > client.4821: S 365777409:365777409(0)
                                                    ack 32320001 win 4096 <mss 512>

 6    29.801493 (23.9738) server.80 > client.4821: S 365777409:365777409(0)
                                                    ack 32320001 win 4096 <mss 512>
 7    29.828256 ( 0.0268) client.4821 > server.80: S 32320000:32320000(0)
                                                    win 61440 <mss 512>
 8    29.828600 ( 0.0003) server.80 > client.4821: S 365777409:365777409(0)
                                                    ack 32320001 win 4096 <mss 512>

 9    77.811791 (47.9832) server.80 > client.4821: S 365777409:365777409(0)
                                                    ack 32320001 win 4096 <mss 512>
10   141.821740 (64.0099) server.80 > client.4821: S 365777409:365777409(0)
                                                    ack 32320001 win 4096 <mss 512>

                          服务器每64秒重传ACK / SYN

18   654.197350 (64.1911) server.80 > client.4821: S 365777409:365777409(0)
                                                    ack 32320001 win 4096 <mss 512>
```

图14-13 服务器插口在SYN_RCVD状态被阻塞近11分钟

客户发送的SYN在第一个报文段中到达，服务器的SYN / ACK在第二个报文段发出。同时服务器设置连接建立定时器为75秒，重传定时器为6秒。上图的第3行中，重传定时器溢出，服务器重传SYN / ACK。这正是我们所期望的。

第4行中可以看到客户端的响应，但这个响应是重传第1行中的最初的那个SYN，而不是我们所期望的对服务器SYN的响应ACK。客户端好像是被中断了。服务器给出了正确的响应：重传SYN / ACK。收到第4个报文段后，服务器端的TCP程序将这条连接的保活定时器(keepalive timer)的超时间隔设为2小时(卷2第745页)。但是，保活定时器与连接建立定时器使用连接控制块中的同一个计数器(卷2的图25-2)，因此，程序清除该计数器的当前值69秒，而

把它设成2小时。通常客户端用一个响应服务器 SYN的ACK来完成三次握手，建立 TCP连接。当这个ACK报文被处理后，保活定时器被设为 2小时，重传定时器则被关闭。

第6、7、8 行的情况类似。服务器的重传定时器在 24秒后超时，重传它的SYN / ACK，但是客户端的响应(它又一次重传了最初的SYN)不正确，因此服务器再次正确地重传 SYN / ACK。在第9行可以看到，服务器的重传定时器在 48秒后再次超时，同样重传它的SYN / ACK。这样，重传定时器到了它的最大值： 64秒，在连接被丢弃之前共发生了 12次重传(12是卷2第674页中的常量TCP_MAXRXTSHIFT的值)。

因为保活定时器、连接建立定时器共用 TCPT_KEEP计数器，所以修补这个故障的方法是：连接还没有完全建立好时 (卷2第745页)，不将保活定时器的超时间隔设为 2小时。当然，作了上述修改后，就要求当连接转移到已建立的状态后把保活定时器设置成初始值 2小时。

14.6 客户端的SYN选项

我们在为期24小时的跟踪中收集了所有的 SYN报文段，从中我们可以看到伴随 SYN的一些不同的参数和选项。

客户端口号

基于伯克利的系统分配的客户临时使用的端口号的范围是 1024~5000(卷2第588页)。正如我们所期望的那样，超过 160 000个客户中有93.5%使用的端口在这个范围内。有 14个客户连接请求使用的端口号小于1024(端口号小于1024的在Net / 3中通常作为保留端口)，其余的6.5%都在5001~65535之间。有些系统，特别是Solaris 2.x，分配的客户端口号都大于32768。

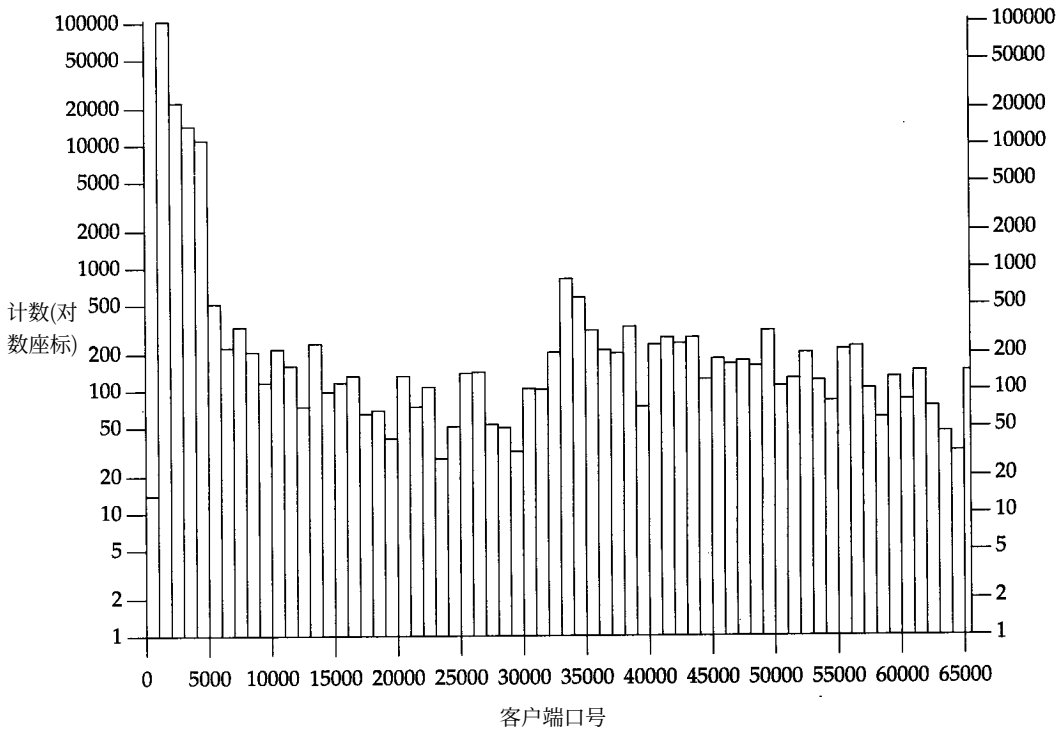


图14-14 客户端口号的范围

图14-14是一个客户使用端口号的分布图，每1000个端口(如1001~2000、2001~3000)作为一个统计范围。请注意 y轴是对数座标。同时我们也看到，绝大部分客户使用的端口在1024~5000之间，而且2/3的端口在1024~2000之间。

最大报文长度

可以基于选用网络的MTU(见前面我们对图10-9的讨论)或直接使用固定值(非本地的同层之间使用512或536，较老的BSD系统使1024，等等)来设置MSS。RFC 1191[Mogul and Deering 1990]列出了典型的16种不同的MTU。因此，在实验中我们希望能找到Web客户所发出的不同的MSS的值有十几种或更多。事实上我们找到了117种不同的值，范围在128~17 520之间。

图14-15列出了最常见的13种客户通告的MSS的值。这5071个连到Web服务器的客户占总客户数5386的94%。第一栏中标记“none”的意思是客户的SYN中没有通告MSS。

MSS	计 数	注 释
无	703	RFC 1122指出如不使用选项，则设为536
212	53	
216	47	256-40
256	516	MTU为296的PPP或SLIP链路
408	24	
472	21	512-40
512	465	非本地主机的常用默认值
536	1097	非本地主机的常用默认值
966	123	ARPANET MTU (1006) - 40
1024	31	老版本BSD中本地主机的默认值
1396	117	
1440	248	以太网MTU (1500) - 40
1460	1626	
	5071	

图14-15 客户所通告的MSS值的分布

初始窗口宽度通告

客户的SYN中也包含了客户端的初始窗口宽度的通告。这里共有117种不同的值，跨越了整个允许值的范围：0~65 535。图14-16列出了最常见的14种值的使用统计数。这4990个值占5386个不同客户的93%。有些值有特殊的意义，但有些值让人感到迷惑，例如：22 099。

窗 口	计 数	注 释
0	317	
512	94	
848	66	
1024	67	
2048	254	
2920	296	2×1460
4096	2062	接收缓存大小的默认值
8192	683	小于常用的默认值
8760	179	6×1460 (以太网上常用)
16384	175	
22099	486	$7 \times 7 \times 11 \times 41$?
22792	128	$7 \times 8 \times 11 \times 37$?
32768	94	
61440	89	60×1024
	4 990	

图14-16 客户所通告的初始窗口尺寸的分布

好像有些PC平台上的Web浏览器允许用户指定MSS和初始窗口尺寸。这就是我们看到一些奇怪值的一个原因，用户设置这些值时可能并没有理解它们的作用。

不管怎么说，我们共找到117种不同的MSS值和117种不同的初始窗口尺寸，并检查了267种不同的MSS和初始窗口尺寸的组合，并没有发现它们之间有明显的相关性。

窗口比例和时戳选项

RFC 1323指定了窗口比例和时戳选项(图2-1)。在5 386个不同的客户中共有78个只发送了窗口比例选项，23个既发送了窗口比例选项，又发送了时戳选项，没有一个只发送时戳选项。在所有的窗口比例选项中都通告了偏移因子 0(意味着比例因子是1，或就是通告的TCP窗口的宽度)。

利用SYN发送数据

五个客户在发送的SYN中捎带数据，但这些SYN并不包含新的T / TCP的选项。检查这些分组，发现这些连接都是同一个模式。客户发送一个普通的 SYN，不含任何数据。三次握手的第二个报文段是服务器的响应，但是响应好像丢失了，因此客户重传了它的 SYN。但是每一个客户重传的SYN中都包含有数据(在200~300字节之间，一个常见的HTTP客户请求)。

路径MTU发现

在RFC 1191[Mogul and Deering 1990]和卷1的第24.2节均描述了路径MTU发现。通过检查客户所发送的SYN报文段中的DF 位(不分段)，可以判断客户是否支持这个选项。在我们的例子中，共有679个客户(占12.6%)支持路径MTU发现。

客户初始序列号

有大量的客户(超过10%)使用0作为初始序列号，用0作为初始序列号明显违反了TCP规范。这些客户的TCP / IP实现中对所有的主动连接都使用 0作为初始序列号，在跟踪中我们发现，同一个客户在几秒钟内在不同端口发出的多个连接请求均使用 0作为初始序列号。图 14-19 列出一个这样的客户。

14.7 客户端的SYN重传

伯克利派生系统是在初始 SYN发出6秒后重传SYN(如果需要)，如果在24秒内仍收不到响应，就再重传(卷2第664页)。因为在24小时的跟踪中我们记录下了所有的 SYN报文(包括那些没有被网络和Tcpdump丢弃的)，所以我们能从中看出客户重传 SYN有多频繁和每一次重传之间的时间。

在这24小时的跟踪中共有160 948个SYN到达(见第14.3节)，其中17 680个(占11%)是重复的(真正的重传数量要小一些，因为如果指定 IP地址和端口号的连续两个 SYN报文之间的时间非常长，那么第二个 SYN就不是重传，而是后来发起的另一条连接。我们没有试图去减掉这部分重传，因为它只占11%中的一小部分)。

SYN只重传一次(最通常的情况)，重传时间典型值是在发出初始 SYN以后3、4或5秒。如果要重传多次，许多客户使用BSD的算法：第一次重传是在6秒以后，接着的下一次是24秒后。我们用{6, 24}来表示这种序列。其他观察到的序列是：

- {3, 6, 12, 14}；
- {5, 10, 20, 40, 60, 60}；
- {4, 4, 4, 4}(违反了RFC 1122中指数增长的要求)；

- {0.7, 1.3}(20跳以外的主机过分频繁的重传；实际上，在这个主机上有 20个连接重传SYN，所有的重传间隔都小于500 ms ！)；
- {3, 6.5, 13, 26, 3, 6.5, 13, 26, 3, 6.5, 13, 26}(这个主机每4次重传后重新按指数退避方法重传)；
- {2.75, 5.5, 11, 22, 44}；
- {21, 17, 106}；
- {5, 0.1, 0.2, 0.4, 0.8, 1.4, 3.2, 6.4}(第一次超时后太主动地重传)；
- {0.4, 0.9, 2, 4}(另一个19跳以外的主机过分频繁的重传)；
- {3, 18, 168, 120, 120, 240}。

就像我们所看到的，上面有些奇怪的序列。有些 SYN被重传很多次，可能是因为发送它的客户有路由问题：它能发送数据到服务器，但收不到服务器的任何响应。同样，也有可能是前一个连接请求的新的实例(卷2第765~766页描述了BSD服务器是如何处理这种情况的：当新的SYN的序列号比处在TIME_WAIT状态的连接的最后一个SYN的序列号还大时，服务器将接受这个新的连接请求)，但是这个时间(例如，明显的是3秒或6秒的倍数)又让人看起来不太像。

14.8 域名

在24小时期间共有5386个不同IP地址的客户连接到Web服务器。因为Tcpdump(带-w标志)只记录带IP地址的分组首部，因此我们必须再来找相应的域名。

我们第一轮用DNS查找名字，试图把这些IP地址映射到它们的域名，只找到了 4052个(占75%)。然后我们在DNS上运行了一天，查找剩下的 1334个IP地址，又找到了62个域名。这意味着有23.6%的客户的IP地址到域名的逆映射不正确(卷1的第14.5节讨论了这些指针查询)。虽然这些客户中的大部分都是通过拨号上网，而且大部分时间是离线的，但他们也应该有他们的名字服务器来提供名字服务，而且名字服务器应是任何时候都接入 Internet的。

在DNS查找名字失败后，我们马上对剩下的 1272个客户运行Ping程序，验证这些没有地址-名字映射的客户是不是会临时不可达。结果是 Ping测试成功了520台主机(占41%)。

分析这些没有映射到一个域名的 IP地址的顶级域名的分布，发现它们来自 57个不同的顶级域名。其中 50个是除美国以外其他国家的两个字母的域名，这也说明用“世界范围内(world wide)”这个词来形容Web是恰当的。

14.9 超时的持续探测

Net / 3从没有放弃过发送持续探测(persist probe)。那就是，当Net / 3收到对等端发送的宽度为0的窗口通告后，它不管是否曾经收到过对方的任何报文，都不断地发送持续探测。当对等端完全消失时(例如，在SLIP或PPP连接时挂断电话)，这样做就会产生问题。回忆一下卷 2第723页提到的，当客户端消失时，有些中间路由器会发送一个主机不可达错误的 ICMP报文，一旦连接建立，TCP将忽略这些错误。

如果连接没有被丢弃， TCP会每60秒往已经消失了的主机发送一个持续探测报文 (浪费Internet资源)，同时每一条连接还继续占用主机上的内存和 TCP访问控制块。

图14-17列出的4.4BSD-Lite2中的代码修补了这个问题，用它来替代卷 2第662页的代码。

tcp_timer.c

```
case TCPT_PERSIST:
    tcpstat.tcps_persisttimeo++;
    /*
     * Hack: if the peer is dead/unreachable, we do not
     * time out if the window is closed.  After a full
     * backoff, drop the connection if the idle time
     * (no responses to probes) reaches the maximum
     * backoff that we would use if retransmitting.
     */
    if (tp->t_rxtshift == TCP_MAXRXTSHIFT &&
        (tp->t_idle >= tcp_maxpersistidle ||
         tp->t_idle >= TCP_REXMTVAL(tp) * tcp_totbackoff)) {
            tcpstat.tcps_persistdrop++;
            tp = tcp_drop(tp, ETIMEDOUT);
            break;
    }
    tcp_setpersist(tp);
    tp->t_force = 1;
    (void) tcp_output(tp);
    tp->t_force = 0;
    break;
```

tcp_timer.c

图14-17 正确处理持续超时的代码

图中的if语句是新代码。变量tcp_maxpersistidle是一个新定义的变量，它的初值是TCPTV_KEEP_IDLE(14 400个500 ms的时钟嘀哒(clock tick)，或2小时)。变量tcp_totbackoff也是一个新变量，它的值是511，是tcp_backoff数组(卷2第669页)中所有元素之和。最后，tcps_persistdrop是tcpstat结构(卷2第638页)中的一个新的计数器，它统计被丢弃的连接。

TCP_MAXRXTSHIFT指定了TCP在等待ACK时的最大重传次数，它的值是12。如果在2小时或对等端的当前RTO的511倍(取两个中较小的)内没有收到对方任何报文，在12次重传后将丢弃连接。例如，RTO是2.5秒(5个时钟嘀哒，一个合理的值)，在22分钟(即2640个时钟嘀哒)后，OR测试条件中的后一个将引起丢弃连接，因为2640大于2555(即5×511)。

代码中的“Hack”注释不是必需的。RFC 1122中规定：即使提供的窗口宽度为0，但只要接收TCP继续给探测报文发送响应，TCP就必须无限期地保持一个连接在打开状态。如果长时间内探测没有响应，最好还是丢弃连接。

在系统中加入的代码可以看出这种情况的发生有多频繁。图14-18给出了为期5天的这个新计数器的值。这个系统平均每天丢弃90个连接，每小时约4个。

让我们详细看一下其中一条连接。图14-19给出了Tcpdump分组跟踪的详细情况。

第1~3行中除了初始序列号错误(0)、MSS值有些奇怪以外，是比较常见的TCP三次握手过程。在第4行，客户发送了一个182字节的请求报文。第5行中服务器对请求进行了响应，在响应报文中包含了应答数据的前512字节，第6行是包含后512字节数据的应答。

在第7行客户发送了一个FIN，第8行中服务器对FIN进行了响应：ACK，紧接着在第9行服务器发送了1024字节的应答。客户在第10行确认了服务器的前512字节的应答，并重传它的FIN。第11行、第12行是服务器的后1024字节的应答。第13~15行中延续了这种情况。

注意，当服务器发送数据时，客户在第7、10、13和16行通告了窗口的减小，直到第17行

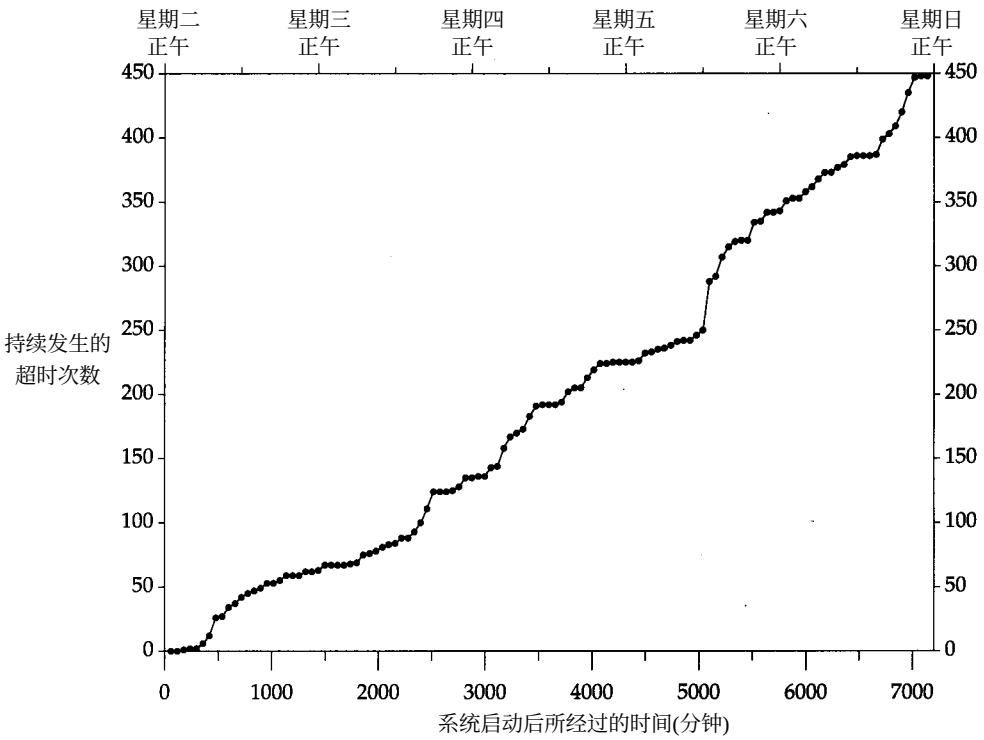


图14-18　持续探测超时后丢弃连接的数量

```
 1     0.0              client.1464 > serv.80: S 0:0(0) win 4096 <mss 1396>
 2     0.001212 (0.0012) serv.80 > client.1464: S 3239320113:3239320113(0)
                                                ack 1 win 4096 <mss 512>
 3     0.354841 (0.3536) client.1464 > serv.80: P ack 1 win 4096

 4     0.481275 (0.1264) client.1464 > serv.80: P 1:183(182) ack 1 win 4096
 5     0.546304 (0.0650) serv.80 > client.1464: . 1:513(512) ack 183 win 4096
 6     0.546761 (0.0005) serv.80 > client.1464: P 513:1025(512) ack 183 win 4096

 7     1.393139 (0.8464) client.1464 > serv.80: FP 183:183(0) ack 513 win 3584
 8     1.394103 (0.0010) serv.80 > client.1464: . 1025:1537(512) ack 184 win 4096
 9     1.394587 (0.0005) serv.80 > client.1464: . 1537:2049(512) ack 184 win 4096

10     1.582501 (0.1879) client.1464 > serv.80: FP 183:183(0) ack 1025 win 3072
11     1.583139 (0.0006) serv.80 > client.1464: . 2049:2561(512) ack 184 win 4096
12     1.583600 (0.0005) serv.80 > client.1464: . 2561:3073(512) ack 184 win 4096
13     2.851548 (1.2679) client.1464 > serv.80: P ack 2049 win 2048
14     2.852214 (0.0007) serv.80 > client.1464: . 3073:3585(512) ack 184 win 4096
15     2.852672 (0.0005) serv.80 > client.1464: . 3585:4097(512) ack 184 win 4096

16     3.812675 (0.9600) client.1464 > serv.80: P ack 3073 win 1024
17     5.257997 (1.4453) client.1464 > serv.80: P ack 4097 win 0

18    10.024936 (4.7669) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
19    16.035379 (6.0104) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
20    28.055130 (12.0198) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
21    52.086026 (24.0309) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
22   100.135380 (48.0494) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
23   160.195529 (60.0601) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
24   220.255059 (60.0595) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
```

图14-19　Tcpdump对持续超时的跟踪

```
                    持续探测连续进行
140 7187.603975 (60.0501) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
141 7247.643905 (60.0399) serv.80 > client.1464: R 4098:4098(0) ack 184 win 4096
```

图14-19 (续)

窗口变为0。到17行为止，客户已接收了从服务器发来的4096字节的数据，4096字节的接收缓存已满了，所以客户通告窗口为0。客户端应用程序没有从接收缓存区中读任何数据。

第18行中服务器发出了它的第一个持续探测报文，它是收到客户窗口为 0的通告约5秒钟后发出的。持续探测报文的间隔时间按照卷 2图25-14的典型情况进行。在第 17行和18行之间的时间内，客户离开了 Internet。在接下来的2小时内，服务器共发送了 124个持续探测报文，最后服务器丢弃了连接，并在第 141行发送了一个RST报文(RST是由tcp_drop调用发送的，见卷2第713页)。

为什么这个例子中服务器发送持续探测报文时间长达 2小时，为什么没有按我们在本节前面讨论过的4.4BSD-Lite2源代码中的OR测试的后半个条件来执行？我们所监视的系统使用的是BSD / OS V2.0，其中持续超时测试代码只测试 t_idle是否大于或等于tcp_maxpersistidle。OR条件测试的后半部分是在4.4BSD-Lite2中加入的新代码。在上面的例子中，我们也可以看出加入这段代码的原因：当通信的另一端显然已离开了Internet时，就不再需要进行2小时的持续探测了。

我们在上面提到系统平均每天有 90个这种持续超时的连接，这就意味着如果系统内核不终止这些连接，4天以后系统中将有360个这样的“保留”连接，这将引起每秒发送 6个无用的TCP报文。另外，因为HTTP服务器还将试图给这些客户发送数据，所以还会产生一些 mbuf在连接发送等待队列中等待发送。[Mogul 1995a]中提到：“当客户过早地终止TCP连接时，会引发服务器程序中隐藏的故障，从而真正地影响性能”。

图14-19的第7行中，服务器收到客户发来的一个 FIN。这使服务器把连接置为CLOSE_WAIT状态。但在跟踪过程中，有时服务器调用close调用，而转至LAST_ACK状态，我们并不能从Tcpdump的输出中区别出来。的确，绝大多数这种连接均在 LAST_ACK状态持续发送探测报文。

在1995年早期，最初开始对插口阻塞在 LAST_ACK状态的问题进行讨论时，有人建议设置SO_KEEPALIVE选项来检测客户退出的时间，然后终止连接 (卷1的第23章讨论了这个选项是怎么工作的，卷 2的第25.6节提供了使用它的细节)。不幸的是，这样做还是解决不了问题。注意卷 2第663页，KEEPALIVE选项在FIN_WAIT_1、FIN_WAIT_2、CLOSING和LAST_ACK状态并不终止连接。据报导，有些厂商对此作了改变。

14.10 T/TCP路由表大小的模拟

实现T/TCP的主机为每一个与它通信的主机保留一条路由表的表项 (第6章)。如今的大部分主机维护的路由表只有一条缺省路由和少数显式指定的路由，所以实现 T/TCP可能要建立一个比通常使用的路由表大得多的路由表。我们将使用 HTTP服务器发出的数据来模拟T/TCP的路由表，看它的空间大小是怎么变化的。

我们只进行简单的模拟。我们通过对这个主机进行 24小时的分组跟踪来建立一个路由表，其中包含每一个与HTTP服务器通信的主机(共有5386个不同的IP地址)的路由。路由表保留的每一条路由信息都设有最后一次更新后的失效时间。我们把失效时间分别设为 30分钟、60分钟和2小时来进行仿真。每 10分钟扫描一次路由表，把所有超过失效时间的路由信息删除 (模仿6.10节中的`in_rtqtimo`的动作)，用一个计数器来记录表中剩下的条目。这些计数器都列在图14-20中。

在卷2的习题18.2中我们注意到，每一条Net / 3的路由表表项要占用152字节。在T/TCP中，这个数字变成了168字节，增加的16字节是`rt_metrics`结构，用作TAO缓存，不过在BSD的内存分配策略中，实际分配的是 256字节。如果取最大的失效时间： 2小时，路由表的表项数将达到1 000个，即需要256 000字节。将失效时间减半，可以使内存的占用量减小一半。

当有5 386个不同的IP地址访问这个服务器时，如果失效时间设为 30分钟，路由表最大可到约300条表项。这样的空间对路由表而言并不是很不切实际的。

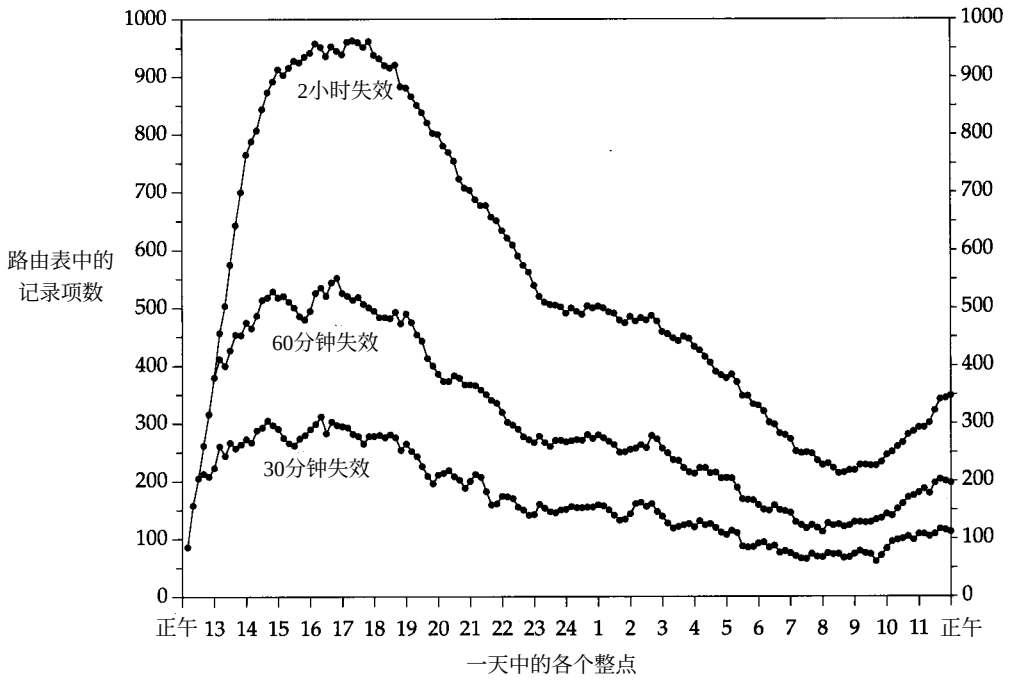


图14-20 T/TCP路由表模拟：每次不同的表项数

路由表的重用

图14-20告诉我们当使用不同的失效时间时路由表会变得多大。但是另一个我们关心的问题是：路由表中保留的这些路由信息中有多少被重用。没有必要保留那些很少用第二次的路由信息。

为了考察这一点，我们检查从 24小时跟踪中得来的686 755个分组，并从中找寻在客户发出最后一个分组至少 10分钟以后发出的SYN。图14-21给出了主机数与静默时间 (分钟)的相对关系图。例如，在 5386个不同的客户所在的主机中 ,有683台主机在10分钟或超过10分钟的静

默时间后发送了另一个SYN。在11分钟或超过11分钟的静默时间后发送了另一个SYN的主机减少至669台，静默时间超过120分钟发送了另一个SYN的主机为367台。

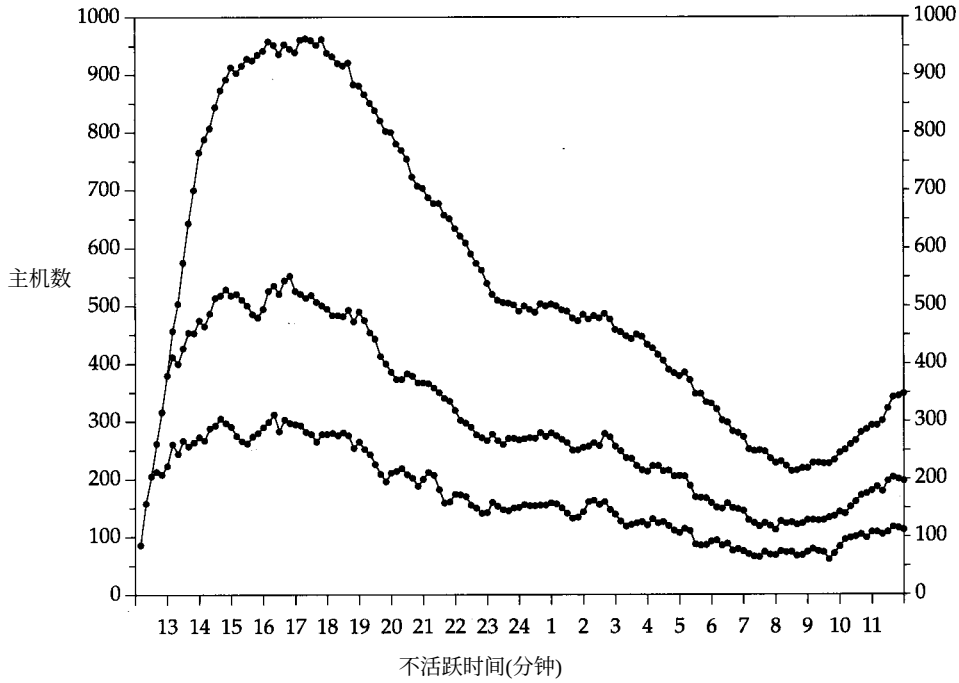


图14-21 在一段时间的静默后发送一个SYN的主机数

如果我们留意一下静默一段时间后又重现的主机，它们的IP地址所对应的主机名都是一些wwwproxy1、webgate1、proxy、gateway和类似于这样的名字，也就是说，多数是一些组织的代理服务器。

14.11 mbuf的交互

在用Tcpdump对HTTP数据交换进行监视时，我们发现了一个有趣的现象。尽管MSS的值大于208(通常都是这样)，可是当应用程序写数据的字节数在101~208之间时，4.4BSD系统还是把它分成了两个mbuf(一个用来存放前100字节，另一个存放剩余的1~108字节)，这样成了两个TCP报文段。这个反常现象的原因是：sosend函数(卷2第399页和第400页)。因为TCP不是一个原子协议，所以每填充一个mbuf，协议的输出函数就被调用一次。

使事情变得更糟的是：因为现在客户的请求被分成多个报文，慢启动现象就产生了。客户只有在收到服务器对第一个报文的确认后才发送第二个报文，这样就增加了一个RTT时延。

大量的HTTP请求的长度在101~208字节间。的确，在13.4节中我们讨论的17个请求的长度均在152~197字节间。这是因为客户的请求基本上都是一个固定的格式，从一个请求转换到另一个请求只是改变URL。

要修补这个问题很简单(如果你有系统内核的源代码)。常量MINCLSIZE的值应从208改为101。这就使得要写101~208字节时，不再使用两个mbuf，而是把超过100字节的数据放入一个或多个mbuf串中。作了这个改变后，还可以摆脱在图A-6和A-7中200字节数据附近的尖峰现象。

图14-22(后面给出)中Tcpdump跟踪的客户就已经作了这个修补。如果没有进行这个修补，客户的第一个报文将只含有100字节，客户将为了等待这个报文的确认而花去一个RTT(慢起动)，然后客户才发送剩余的52字节。只有在收到剩余的字节后，服务器才会发出第一个应答报文。

这里有一些其他的修补方法。第一种方法是：一个mbuf的大小可以由128字节增加至256字节。有些基于伯克利源码的系统已经作了这种修改(例如，AIX)。第二种：对sosend作修改，当使用多个mbuf时，避免多次调用TCP输出。

14.12 TCP的PCB高速缓存和首部预测

当Net/3收到一个报文时，它把指针保存在相应的Internet PCB(指向inpcb结构的tcp_last_inpcb指针，见卷2图28-5)中，并希望下个到达的报文还是属于同一条连接。这样做避免了查找TCP的PCB链表，而这样的查找的代价是昂贵的。每一次缓存比较失败，计数器tcps_pcbcachemiss就增加。在卷2图24-5的抽样统计中缓存的命中率接近80%，但被统计的系统不是一个HTTP服务器而是一个普通的分时系统。

当给定连接所接收的下一个报文不是下一个希望的ACK(在数据发送方)，就是下一个希望的数据报文(在数据接收方)时，TCP的输入也执行一些首部预测(卷2第28.4节)。

在本章讨论的HTTP服务器上，我们观察到了下面的一些百分数：

- 20%的PCB缓存命中率(18~20%);
- 对下一个ACK报文的15%的首部预测率(14~15%);
- 对下一个数据报文的30%的首部预测率(20~35%)。

所有这些比率都是比较低的。两天中每个小时对这些百分数进行测量，发现它们的变化都很小：上面括号中列出了高低值的范围。

作为一个在同一时刻有大量不同客户使用TCP的HTTP服务器，PCB缓存命中率比较低并不让人感到奇怪。这种低的比率与HTTP是一个传输协议相适应，[McKenney and Dove 1992]中表明了Net / 3的PCB缓存机制对事务协议不太有效。

通常一个HTTP服务器发送的数据报文比它接收的要多。图14-22是图13-5中客户的第一个HTTP请求的时间线(客户端口号为1114)。客户的请求是第4段报文，服务器的应答是第5、6、8、9、11、13和14段报文。这里，服务器只有一个可能的数据报文预测，那就是第4段报文。服务器的下一个可能的ACK报文预测是第7、10、12、15和16段报文(当第3段报文到达时，连接还没有完全建立好，第17段报文中FIN标志使程序不再对首部进行预测)。这些ACK报文究竟会不会限制依赖于窗口通告的首部预测，取决于客户端发送ACK时它读取了多少服务器返回的数据。例如在第7段报文，TCP确认了收到1024字节数据，但是HTTP客户应用程序只从插口缓存中读取了260字节数据(1024 - 8192+7428)。

当TCP的200 ms定时器超时时，会发送一个延迟的ACK，它带有一个可笑的窗口通告。同样都是延迟的ACK，第7段与第12段报文时间上的差距是799 ms：4个TCP的200 ms时钟中断。这就暗示它们都是延迟的ACK，发送它们是因为时钟中断，而不是因为进程执行了新的、从插口缓存读数据的调用。第10段报文看上去好像也是延迟的ACK，因为它与第7段报文之间的时间为603 ms。

带有小的窗口通告的ACK报文的发送也会使首部预测失效，因为只有当窗口通告的值等于当前发送窗口的值时，才会执行首部预测。

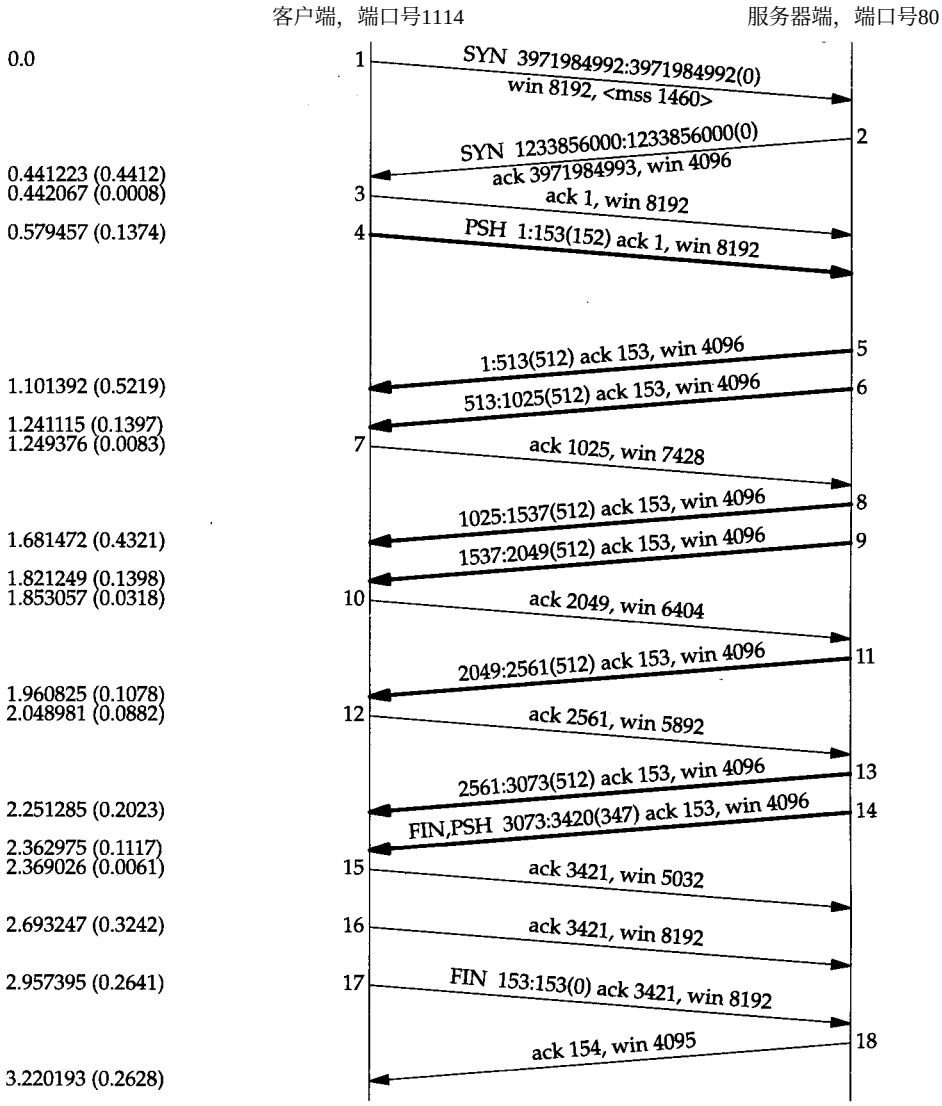


图14-22 HTTP客户-服务器事务

总的来说，我们对HTTP服务器上首部预测成功率低并不感到惊讶。在TCP连接上交换大量的数据时首部预测工作得最好。因为系统内核首部预测的统计是计算所有的TCP连接，我们只能猜测这台主机上对下一个数据报文的首部预测的高百分比(与对下一个ACK的预测相比)是来自非常长时间的NNTP连接(图15-3)，这种NNTP连接平均每条TCP连接接收约1300万字节。

慢启动错误

注意到图14-22中当服务器发送它的应答时没有像预期的那样发生慢启动。我们预期的是服务器先发送512字节的报文，等待客户的ACK，然后发送下一个512字节的报文。而服务器

不是这样做的，它没有等待客户的 ACK而是立即发送了两个 512字节的报文(第5段和第6段报文)。事实上这种现象在绝大多数伯克利的派生系统是很少见和异常的，因为许多应用程序都是由客户发送大多数数据给服务器。甚至对于 FTP也是这样，例如，从一个 FTP服务器上获取一个文件时，FTP服务器打开一个数据传输连接，实际上成为了数据传输的客户端 (卷1图27-7给出了一个这样的例子)。

这个错误出在`tcp_input`函数上。新的连接启动时拥塞窗口为一个报文。当客户完成连接建立后(卷2图28-21)，代码执行转移到`step6`，跳过了ACK的处理。当客户发送第一个数据段时，它的拥塞窗口是一个报文，这是不正确的。但是，当服务器完成连接建立后 (卷2图29-2)，紧接着执行处理ACK的代码，收到ACK后拥塞窗口增加1个报文(卷2图29-7)。这就是为什么服务器一开始就连续发送两个报文。解决这个问题的办法是把图 11-16中的代码加进去，不管这样是不是支持T/TCP。

当服务器在第7段报文中收到ACK时，它的拥塞窗口增加至 3个报文段，但接着服务器却只发送2个报文段(第8和第9段报文)。我们不能从图14-22中找出原因来，因为我们只在连接的一端记录报文(在客户端运行Tcpdump)，第10段和第11段报文可能在网络中客户端与服务器端中间的什么地方。如果真是这样，那么服务器就的确像我们所预期的那样：拥塞窗口的宽度为3个报文段。

这些报文交互的线索是从对分组跟踪得来的 RTT值。在客户端测量出来的第 1段与第2段报文之间的RTT是441 ms，第4段与第5段之间是521 ms，第7段与第8段之间是432 ms。这些都是可能的值，在客户端使用 Ping程序(指定分组长度为300字节)也表明到这个服务器之间的RTT大约是461 ms。但第10段与第11段报文之间的RTT非常小，只有107 ms。

14.13 小结

通过运行一个繁忙的Web服务器来重点考察TCP / IP的实现。我们可以看到，服务器会收到Internet上各种各样的客户发来的一些奇怪的分组。

在本章中，我们对一个繁忙的 Web服务器的分组进行跟踪，并对跟踪结果进行分析，着眼于各种实现中的特性。我们得到了如下结论：

- 客户端SYN的峰值到达速率约为平均到达速率的8倍(忽略不正常的客户)。
- 客户到服务器之间的RTT平均值是445 ms，中间值是187 ms。
- 采用典型的backlog极限值5或10时，未完成连接队列很容易溢出。这个问题不是因为服务器进程太忙，而是因为客户的 SYN至少要在队列中停留一个 RTT时间。一个繁忙的Web服务器的这个队列需要比这大得多的容量。同时内核也提供一个计数器对这个队列的溢出次数进行统计，这样系统管理员就可以知道这种溢出发生的频度。
- 对阻塞在LAST_ACK状态的连接不断进行持续探测，因为这种情况经常出现，所以系统必须提供一种办法能让这种连接超时。
- 许多伯克利派生系统在客户请求报文的长度为 101~208字节时(通常许多客户均为这样)使用mubf的效率比较低。
- 许多伯克利派生系统的实现提供TCP PCB高速缓存，同时绝大多数的系统也提供首部预测，但是它们对一个繁忙的Web服务器的帮助却很小。

[Mogul 1995d]中提供了对另一个繁忙的Web服务器进行了相似的分析。

第15章　NNTP：网络新闻传送协议

15.1　概述

NNTP，即网络新闻传送协议，在协作的主机之间发布新闻文章。NNTP是一个使用TCP的应用协议，RFC 977[Kantor and Lapsley 1986]对它进行了详细描述。[Barber 1995]对它的一般实现进行了扩展。RFC 1 036 [Horton and Adams 1987]对新闻文章中的各种首部字段进行了说明。

网络新闻起源于ARPANET上的邮件列表，随后发展成为Usenet新闻系统。邮件列表今天还很流行，但如果纯粹从容量来看，网络新闻在过去的十年有很大的增长。从图 13-1中可以看出，NNTP有跟电子邮件一样多的分组数。[Paxson 1994a]中提到，从1984年以来网络新闻的流量保持了每年约75%的增长。

Usenet不是一个物理的网络，而是建立在多个不同类型物理网络上的一个逻辑网。多年以前，在Usenet上流行的交换网络新闻的手段是通过电话线拨号 (为了省钱通常在几个小时以后)，而在今天，Internet是绝大多数新闻发布的主要渠道。[Salus 1995]中的第15章详细讲述了Usenet的历史。

图15-1是一个典型的新闻系统的概况。一台作为组织的新闻服务器的主机在磁盘上保留

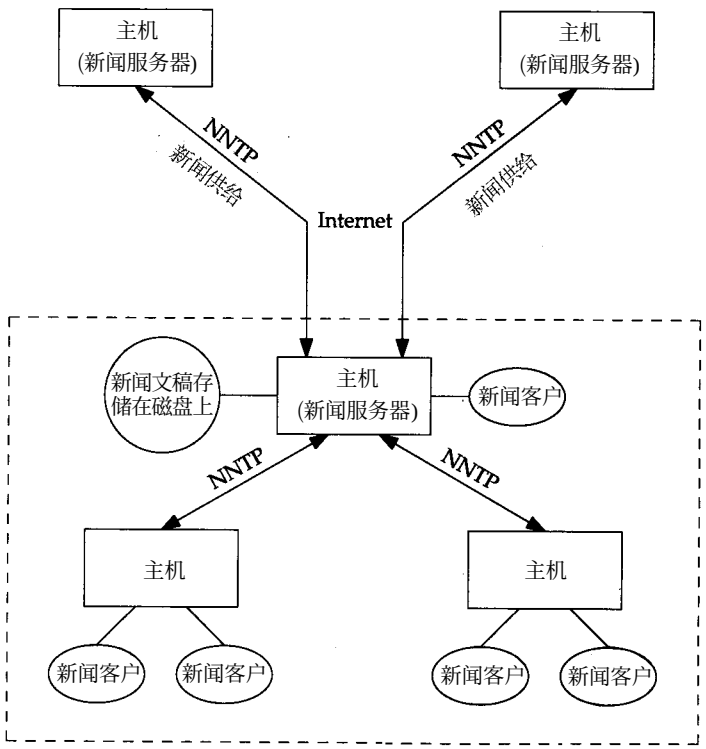


图15-1　典型的新闻系统

了所有新闻文章。这个新闻服务器通过 Internet与其他的新闻服务器通信，互相供给新闻。新闻服务器之间的通信使用 NNTP协议。新闻服务器有各种不同的实现， INN(InterNetNews)正成为Unix平台上最流行的新闻服务器程序。

组织中的其他主机通过访问新闻服务器来阅读新闻文章和选择新闻组粘贴新闻。我们把这些客户程序称为“ 新闻客户 ”。这些客户程序与新闻服务器之间的通信也采用 NNTP协议。另外，如果新闻客户与新闻服务器在同一主机上，客户也用 NNTP阅读和粘贴新闻。

不同的客户操作系统平台上有十几种新闻阅读器 (客户程序)。最原始的Unix新闻客户程序是Readnews，接着是Rn和它的变种：Rrn是一个支持远程操作(remote)的版本，它允许客户和服务器在不同的主机上；Trn的意思是“ 线索(threaded)Rn ”，它可以使用多条线索在一个新闻组中讨论；Xrn是Rn的X11窗口系统的版本。 GNUS是一个内含Emacs编辑器的流行的新闻阅读器。也有一些通用的 Web浏览器，例如 Netscape，在浏览器中内置访问新闻服务器的接口，这样就不需要单独的新闻客户程序。就像不同的电子邮件程序提供许多不同的用户接口一样，每一种不同的新闻客户程序也提供不同的用户接口。

不管使用哪种客户程序，对新闻服务器来说，这些不同的新闻客户程序的相同点是：都使用NNTP协议，这正是我们在本章要讨论的。

15.2 NNTP协议

NNTP使用TCP协议，知名的 NNTP服务的端口号是 119。NNTP也像其他的 Internet应用(HTTP，FTP，SMTP，等等)一样，客户发送ASCII命令给服务器，服务器返回数值的响应码，后面跟着可选的ASCII数据(取决于客户的命令)。命令和响应都以回车加换行结束。

考察这种协议最简单的办法就是用 Telnet程序来连接一台主机上的 NNTP端口，当然，这台主机运行了NNTP服务器程序。但是，通常我们必须从一台能被服务器主机识别的主机上运行客户程序，典型的情况就是从同一组织网络中的一台主机。例如，我们通过 Internet从其他网络的主机上来登录本地的新闻服务器，会收到如下的错误信息：

```
vangogh.cs.berkley.edu % telnet noao.edu nntp
Trying 140.252.1.54...                          由Telnet客户程序输出
Connected to noao.edu.                          由Telnet客户程序输出
Escape character is '^]'.                       由Telnet客户程序输出
502 You have no permission to talk.  Goodbye.
Connection closed by foreign host.              由Telnet客户程序输出
```

输出的第4行是由NNTP服务器输出的，响应码是 502。当TCP连接被建立后， NNTP服务器收到客户的IP地址，将它与配置中允许的IP地址进行比较。

在下面的例子中，我们从一台“ 本地 ”主机连接到新闻服务器。

```
sun.tuc.noao.edu % telnet noao.edu nntp
Trying 140.252.1.54...
Connected to noao.edu.
Escape character is '^]'.
200 noao InterNetNews NNRP server INN 1.4 22-Dec-93 ready (posting ok).
```

这次从服务器来的响应码为200(命令OK)，响应行中余下的是服务器的有关信息。返回信息的最后是“ posting ok ”或“ no posting ”，这取决于是否允许客户粘贴新闻 (这个由系统管理员根据客户的IP地址来控制)。

我们注意到服务器的响应信息中提到，这个服务器是 NNRP(Network News Reading Protocol)服务器，而不是INND(InterNetNews daemon)服务器。先是INND服务器接收客户的请求，查找客户的IP地址。如果客户的IP地址是被允许的，并且客户不是一个已知的、提供新闻的主机，那么NNRP服务器被激活，替代INND服务器，假定客户是想要读新闻而不是想要给服务器提供新闻。这就可以将新闻供给服务器 (约10 000行C代码)与新闻阅读服务器(约5 000行C代码)分别来实现。

图15-2列出了数字响应码中第1位和第2位的含义。这与FTP中的用法也很相似(卷1的319页)。

应 答	说 明
1yz	报告情况的消息
2yz	命令执行成功
3yz	迄今为止命令执行成功；发送余下的命令
4yz	命令正确，但因为某些原因不能执行
5yz	命令未实现，或命令不正确，或遇到了严重的程序差错
x0z	连接、设置和杂项消息
x1z	新闻组选择
x2z	文章选择
x3z	分发功能
x4z	发送
x8z	非标准扩展
x9z	调试用输出

图15-2 三位响应码中第1位和第2位的含义

我们发给新闻服务器的第一个命令是 help，help命令会将这个新闻服务器支持的所有命令列出来。

```
help
100 Legal commands                          100是响应吗
  authinfo user Name|pass Password
  article [MessageID|Number]
  body [MessageID|Number]
  date
  group newsgroup
  head [MessageID|Number]
  help
  ihave
  last
  list [active|newsgroups|distributions|schema]
  listgroup newsgroup
  mode reader
  newgroups yymmdd hhmmss ["GMT"] [<distributions>]
  newnews newsgroups yymmdd hhmmss ["GMT"] [<distributions>]
  next
  post
  slave
  stat [MessageID|Number]
  xgtitle [group_pattern]
  xhdr header [range|MessageID]
  xover [range]
  xpat header range|MessageID pat [morepat...]
  xpath xpath MessageID
Report problems to <usenet@noao.edu>
.                                           这一行只有一个句点，表示服务器响应的结束
```

因为客户无法确认服务器返回的信息到底有多少行，所以协议要求服务器以一个只包含句号的行来结束返回。如果某一行恰好就是要以句号开头，那么服务器会在前面再加上一个句号再发送，客户收到这行后先去掉这个句号。

下面我们来看一下list命令。如果不带任何参数执行list命令，它列出所有本服务器上每一个新闻组的名字，后面跟着这个组中最后一篇新闻和组中第一篇新闻的编号，最后是“y”或“m”，表示是否允许这个组粘贴新闻或只是一个普通的组。

```
list
215 Newsgroups in form "group high low flags".          215是响应码
alt.activism 0000113976 13444 y
alt.aquaria 0000050114 44782 y
                                                  还有许多行没有显示出来

comp.protocols.tcp-ip 0000043831 41289 y
comp.security.announce 0000000141 00117 m
                                                  还有许多行没有显示出来

rec.skiing.alpine 0000025451 03612 y
rec.skiing.nordic 0000007641 01507 y
                                                  这一行只有一个句点，表示服务器响应的结束
```

当然，215是响应码，而不是新闻组的编号。这个例子中的服务器向客户返回了 4 238个新闻组的情况，共175 833字节的TCP数据。返回的新闻组信息没有按字母排序。

在新闻客户上通过较慢的拨号线从一个新闻服务器获取这样一个列表，通常会感到很慢。例如，如果数据传输速率是 28 800 bit/s，那么这个过程将花费约 1分钟(实际测量时，使用这样一个调制解调器，并在数据发送时进行压缩，大约需 50秒)。在以太网上，这个过程所需时间不到1秒。

group命令用来指定某一新闻组作为客户的“当前”新闻组。下面的命令就是把comp.protocols.tcp-ip设为当前新闻组。

```
group comp.protocols.tcp-ip
211 181 41 289 43 831 comp.protocols.tcp-ip
```

服务器以响应码211(命令执行成功)开头，后面跟着这个组中新闻总数的估计值 (181)，然后是本组中第一篇新闻文章的编号 (41 289)、最后一篇新闻文章的编号 (43 831)和新闻组的名字。在新闻文章的起始和结束编号之间的差值 (43831 - 41289=2542)通常大于新闻文章数(181)。一方面是因为有些文章是典型的 FAQ(Frequently Asked Questions)，它们的失效时间(通常为1个月)比起其他大多数新闻 (很少的几天，取决于服务器硬盘的容量)要长得多。另一个原因是这些文章能被显式地删除。

下面我们用head命令来看一篇特殊文章(编号为43 814)的首部内容。

```
head 43814
221 43814 <3vtrje$ote@noao.edu> head
Path: noao!rstevens
From. rstevens@noao.edu (W. Richard Stevens)
Newsgroups: comp.protocols.tcp-ip
Subject: Re: IP Mapper:  Using RAW sockets?
Date: 4 Aug 1995 19:14:54 GMT
Organization: National Optical Astronomy Observatories, Tucson, AZ, USA
Lines: 29
Message-ID: <3vtrje$ote@noao.edu>
```

```
References: <3vtdhb$jnf@oclc.org>
NNTP-Posting-Host: gemini.tuc.noao.edu
```

应答的第一行带有响应码221(命令执行成功)，后面是10行的首部，最后是只含有句号的一行。

大多数首部字段无需解释，但消息的ID看上去有些让人迷惑。INN试图按下面的格式生成唯一的消息ID格式：当前时间，一个$符号，进程ID，一个@符号，本地主机的完整域名。时间和进程号的数值都以32进制的数字串输出：数字由每5位二进制数为一组，每一组用字母：0...9a...v来表示。

接着我们用body命令返回同一篇文章的主体。

```
body 43814
222 43814 <3vtrje$ote@noao.edu> body
> My group is looking at implementing an IP address mapper on a UNIX
```

文章中还有28行没有列出来

新闻的首部和主体可以用一个命令(article)获取，但绝大多数新闻客户是先取得文章的首部，允许客户根据新闻的主题进行选择，然后只取回用户所选取的文章的主体。

我们用quit命令来终止到服务器的连接。

```
quit
205
Connection closed by foreign host.
```

服务器的响应是数字代码：201。我们的客户程序Telnet显示服务器关闭了连接。

整个客户与服务器的交互过程只使用单个的、由客户发起的TCP连接。但是连接上大部分数据都是由服务器发向客户的。连接的持续时间以及数据的交换量均取决于用户阅读新闻时间的长短。

15.3 一个简单的新闻客户

下面我们通过使用一个简单的新闻客户程序，进行简要的新闻会话来看一下NNTP命令与响应之间的交互。我们使用最老的新闻阅读器中的一种：Rn，它简单而且容易使用，同时选用它还因为它带有调试选项(-D16命令行选项，假定客户程序编译时打开了调试选项)。这让我们可以看到客户发出的NNTP命令以及相应的服务器的响应。我们用黑体字来表示客户端的命令。

1) 第一个命令是list，在上一节中我们看到从服务器返回了约175 000字节，每行表示一个新闻组。同时Rn也把用户想要阅读的新闻组以及在这组中最后读过的新闻的编号的列表保存在文件.newsrc(在用户的主目录中)中。例如，某一行：

```
comp.protocols.tcp-ip: 1-43815
```

通过把文件中保存的、最后读过的新闻的编号与现有最新的新闻的编号进行比较，客户就知道本组中是否还有没读过的新闻。

2) 然后客户检查是否有新的新闻组建立。

```
NEWGROUPS 950803 192708 GMT
231 New newsgroups follow.          231是响应码
.
```

Rn在用户的主目录的文件.rnlast中保存了最近一次通报新的新闻组的时间。这个时间成为newsgroups命令的参数(NNTP命令和命令的参数都与大小写无关)。在这个例子中保存的时间是：格林尼治时间 1995年8月3日，19:27:08。服务器返回为空(在返回码231与只包含句点的行中没有其他的内容)，指示没有新的新闻组建立。如果有新的新闻组建立，客户程序会询问用户是否要加入这个组。

3) 接着Rn将显示前5个新闻组中未读新闻的编号，并询问是否要阅读第一个新闻组：comp.protocols.tcp-ip。我们以一个等于号响应，让Rn返回一个对该组所有文章的一行摘要，然后我们可以选择想要阅读的文章(可以用.rninit文件对Rn进行配置，让Rn按照我们期望的方式给出每一篇新闻文章的摘要。书的作者配置的摘要包括文章编号、主题、文章的行数和文章的作者)。group命令是由Rn发出的，设置当前的新闻组。

```
GROUP comp.protocols.tcp-ip
211 182 41289 43832 comp.protocols.tcp-ip
```

第一篇未读文章的首部和主体可以用以下命令获得：

```
ARTICLE 43815
220 43815 <3vtq8o$5p1@newsflash.concordia.ca> article
.                                    文章未列出
```

第一篇未读文章的一行摘要显示在终端上。

4) 对本组中剩下的17个未读的新闻执行xhrd命令，然后再用head命令。如下面的例子：

```
XHDR subject 43816
221 subject fields follow
43816 Re: RIP-2 and messy sub-nets
.
HEAD 43816
221 43816 <3vtqe3$cgb@xap.xyplex.com> head
.                                    首部的14行未列出
```

xhdr命令能接受的参数不但可以是单个文章编号，还可以是号码范围，这就是为什么服务器返回了许多行，然后以只含有句点的行结束的原因。每篇文章的一行摘要显示在终端上。

5) 我们敲空格键选择第一篇未读的文章，客户程序发出 head命令，接着是article命令。文章便显示在终端上。对所有文章相继使用这两个命令。

6) 当我们读完这个组的新闻后，便移到另一个组，这时客户程序又发出另一个 group命令。我们向服务器请求每一篇未读新闻的一行摘要，在新的组中再一次执行上面讨论过的命令。

我们注意到的第一件事情是Rn发出了太多的命令。例如，为了对所有的未读文章取得一行摘要，先要发出xhdr取得摘要，然后是用head命令取得文章的首部。这两个命令中的第一个是不必要的。增加这些额外命令的原因之一是最开始这些命令是为工作在主机(也是新闻服务器)上的客户程序设计的，因此这些附加命令可能要快一些，因为没有网络的传输时间。使用NNTP访问远程新闻服务器的功能是后来加上去的。

15.4 一个复杂的新闻客户

下面我们来看一个更复杂的新闻客户程序： Netscape 1.1N版的Web浏览器，它内置了新

闻阅读器。这个客户程序没有调试选项，所以我们只有跟踪它与服务器之间交换的 TCP分组来看它是怎样工作的。

1) 当我们启动客户程序，并选择新闻阅读特性时，它读 .newsrc文件，并且只向服务器请求在这个文件中我们所预订的新闻组的相关内容。对每一个预订的新闻组都发出group命令来确定起始和结束的文章编号，并与 .newsrc文件中所存储的最后阅读的文章编号进行比较。这个例子中，作者在 4 000多个新闻组中只预订了77个，因此共有77个group命令发向服务器。这在拨号线的 PPP链路上仅需23秒，相比较而言， Rn所使用的list命令要50秒。

如果新闻组的数量由 4 000减少至77，客户所花的时间应小于 23秒。实际上，用 sock (卷1附录C)发送77个同样的group命令只需约3秒。看起来浏览器在这 77个命令上叠加了其他的启动处理。

2) 我们选择一个有未读文章的新闻组： comp.protocols.tcp-ip，接着执行下面的命令：

```
group comp.protocols.tcp-ip
211 181 41289 43831 comp.protocols.tcp-ip
xover 43815-43831
224 data follows
43815\tping works but netscape is flaky\troot@PROBLEM_WITH_INEWS
_DOMAIN_FILE (root)\t4 Aug 1995 18:52:08 GMT\t<3vtq8o$5p1@newsfl
ash.concordia.ca>\t\t1202\t13
43816\tRe: help me to select a terminal server\tgvcnet@hntp2.hin
et.net (gvcnet)\t5 Aug 1995 09:35:08 GMT\t<3vve0c$gq5@serv.hinet
.net>\t<claude.80753760 @bauv111>\t1503\t23
.                                        指定范围内剩余文章的一行摘要
```

第一个命令设置当前新闻组，第二个命令向服务器请求指定文章的概况。在这个组中，43 815是第一篇、43 831是最后一篇未读的文章。每篇文章的一行摘要包括：文章编号、主题、作者、日期和时间、消息 ID、文章的引用、字节数和行数 (注意每个一行摘要都很长，所以上面我们把每一行都几次换行。同时我们还把分隔字段的 tab符换成了\t，这样便于看清楚)。

Netscape客户程序按主题组织返回的概况，并显示未读主题的列表以及文章的作者和行数。将一篇文章及其应答组合在一起，称为编线索，因为一个议题的线索都是组合在一起的。

3) 对每一篇我们选择阅读的文章，执行一次article命令，文章便显示出来。

从上面的Netscape新闻客户程序的概况中可以看出，它采用两种优化措施来减少用户的等待时间。第一个措施是只向服务器请求用户所需要阅读的新闻组，而不是使用 list命令。第二，它使用xover命令提供每一篇文章的摘要，而不是对组中的每一篇文章使用一次 head和xhrd命令。

15.5 NNTP的统计资料

为了理解典型的 NNTP的用法，我们在第 14章曾提到的主机上运行 Tcpdump，来收集NNTP所使用的SYN、FIN和RST报文。这个主机从一台 NNTP新闻供给主机上获得新闻 (可能有其他备用的新闻供给主机，但是观察到的报文都是来自同一台主机)，然后分发给10个其他

站点。在这10个站点中只有两个使用NNTP，其他都是使用UUCP，所以我们的Tcpdump只记录到两个NNTP供给主机。两个流出的NNTP主机收到的新闻只是这台主机收到的新闻的一小部分。最后，因为这台主机属于一个Internet服务提供商，所以各式各样的客户把主机当成新闻服务器来阅读新闻。所有的客户阅读新闻均使用NNTP协议，包括同在主机上的新闻阅读进程和其他主机上的新闻阅读进程(典型的是通过PPP或SLIP连接)。Tcpdump连续运行了113小时(4.7天)，共收集了1 250个连接上的信息。图15-3汇总了这些信息。

	1个输入新闻供给	两个输出新闻供给	新闻阅读客户	总　计
连接数	67	32	1 151	1 250
流入字节总数	875 345 619	4 499	593 731	875 943 849
流出字节总数	4 071 785	1 194 086	56 488 715	61 754 586
总持续时间(分钟)	6 686	407	21 758	28 851
每连接流入字节数	13 064 860	141	516	
每连接流出字节数	60 773	37 315	49 078	
连接平均持续时间(分钟)	100	13	19	

图15-3　单个主机上4.7天的NNTP统计资料

我们首先注意输入新闻供给主机，它每天收到约1.86亿字节的新闻，平均每小时约为8百万字节。同时我们也可以看出，到主新闻供给主机的NNTP连接的持续时间很长：100分钟，交换了1300万字节的数据。这台主机与它的输入新闻供给主机之间的TCP连接经过一段时间的静默后由新闻服务器关闭。下次需要时再重新建立连接。

典型的新闻阅读程序使用NNTP连接约19分钟，读取约50 000字节的新闻。绝大多数NNTP流量是单向的：从主新闻供给主机流向新闻服务器，从新闻服务器流向新闻阅读客户。

站点-站点的NNTP流量之间有巨大的差异。上面的统计数据就是一个例子：这些统计数据中没有典型值。

15.6　小结

NNTP是又一个使用TCP协议的简单协议。客户发出ASCII命令(服务器支持超过20种不同的命令)，服务器的响应先是响应码，然后跟着一行或多行的应答，最后以只包含句号的行结束(如果响应是可变长度)。类似其他的Internet协议，NNTP协议本身已多年没有变化，但是由客户程序提供给交互式用户的接口却变化很快。

不同新闻阅读程序之间的很多区别都取决于应用程序怎样使用协议。我们看到Rn客户程序和Netscape程序之间的不同有：确定哪些文章未读的方法不同，取得未读文章的方法也不同。

NNTP协议使用单个TCP连接维持整个的客户-服务器数据交换。这一点与HTTP协议不同，HTTP协议从服务器每获取一个文件都要建立一条TCP连接。这种差异一方面是因为NNTP客户只与一个服务器通信，而HTTP客户能同时与多个不同服务器通信。同时我们也看到绝大多数TCP连接上的NNTP协议的数据流是单向的。

第三部分　Unix域协议

第16章　Unix域协议：概述

16.1　概述

Unix域协议是进程间通信 (IPC)的一种形式，可以通过与网络通信中使用的相同插口 API来访问它们。图 16-1的左边表示使用插口写成的客户程序和服务器程序，它们在同一台主机上利用Internet协议进行通信。图 16-1的右边表示用插口写的利用 Unix域协议进行通信的客户程序和服务器程序。

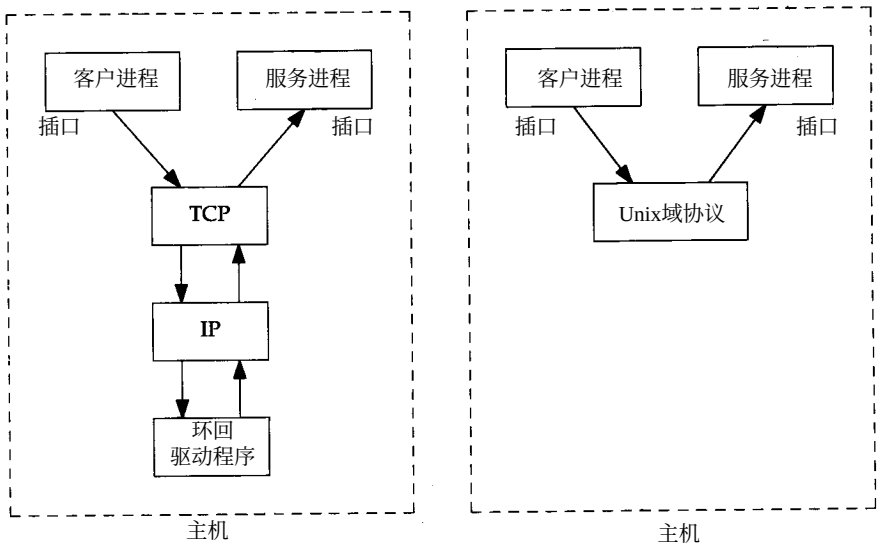


图16-1　使用Internet协议和Unix域协议的客户程序与服务器程序

当客户进程通过TCP往服务器进程发送数据时，数据首先由 TCP输出处理，然后再经过 IP输出处理，最后发往环回驱动器 (见卷2的5.4节)，在环回驱动器中，数据首先被放到 IP输入队列，然后经过 IP输入和TCP输入处理，最后传送到服务器。这样工作得很好，并且对于在相同主机上的对等端 (客户进程和服务器进程)来说是透明的。然而，在 TCP/IP协议栈里需要大量的处理过程，当数据没有离开主机时，这些处理过程实际上是不需要的。

Unix域协议由于知道数据不会离开主机，所以只需要较少的处理过程 (这样数据传送就快多了)。不需要进行检验和的计算和验证，数据也不会失序，由于内核能控制客户进程和服务器进程的执行过程，流量控制也被大大简化了，等等。虽然 IPC的其他形式也有这些优点 (消息队列、共享内存、命名管道，等等)，但是，Unix域协议的优点在于它们使用的接口与网络应用程序使用的插口接口完全一样：客户程序调用 `connect`，服务器程序调用 `listen`和

accept，两者都调用read和write，等等。而其他形式的IPC使用完全不同的API，其中有一些不能与插口以及其形方式的 I/O较好地交互 (例如，在系统 V消息队列中我们不能使用select函数)。

一些TCP/IP实现努力通过优化去提高性能，例如当目的地址是环回接口时可以忽略TCP检验和的计算和验证。

Unix域协议既提供一个流插口 (SOCK_STREAM，与TCP字节流相似)，又提供一个数据报插口(SOCK_DGRAM，与UDP数据报相似)。Unix域插口的地址族是AF_UNIX。在Unix域协议中用于标识插口的名字是文件系统的路径名 (Internet协议使用IP地址和端口号的组合来标识TCP和UDP插口)。

网络编程API标准IEEE POSIX 1003.1g 也支持Unix域协议，它使用的名称是“local IPC”。其地址族是AF_LOCAL，协议族是PF_LOCAL。因而使用术语“Unix”来描述这些协议也许已成为历史。

Unix域协议还能提供在不同机器之间进程间通信时所没有的功能。这一功能就是描述符传递，即通过Unix域协议在互不相关的进程间传送描述符的能力，这个例子我们将在第 18章讨论。

16.2 用途

许多应用程序使用Unix域协议：

1) 管道。在一个源于伯克利的内核里，使用 Unix域流插口来实现管道。在17.13节里我们将讨论pipe系统调用的实现。
2) X Window系统。当与X11服务器相连时， X11客户进程通常基于DISPLAY环境变量的值或基于-display命令行参数值来决定使用什么协议。这个值的形式是 *hostname:display.screen*，这里hostname是可选的，默认值是当前主机，使用的协议是最有效的通信方式，其中典型的是 Unix域流协议。值unix强制使用Unix域流协议。服务器绑定到Unix插口上的名字类似于/tmp/.X11-unix/X0。
   由于X服务器进程通常处理在同一台主机或者网络上的客户进程的请求，这就意味着服务器进程需要等待一个连接请求到达TCP插口或者Unix流插口。
3) BSD打印假脱机系统(lpr客户进程和lpd服务器进程，在[Stevens 1990]的第13章详细描述)使用一个名为/dev/lp的Unix域流插口在相同的主机上进行通信。像 X服务器一样， lpd服务器使用Unix插口处理在相同主机上客户进程的连接，或者使用 TCP插口处理网络上客户进程的连接。
4) BSD系统记录器(syslog库函数，可以被任何应用程序调用)和syslogd服务器程序(使用一个名为/dev/log的Unix域数据报插口在相同的主机上进行通信)。客户进程写一个消息到插口上，服务器进程读出来并进行处理。服务器进程也处理来自其他主机上使用UDP插口的客户进程的消息，关于这种机制的详细介绍见 [Stevens 1992]的13.4.2节。
5) InterNetNews守护程序(innd)创建一个Unix 数据报插口来读取控制报文，一个 Unix流插口来读取本地新闻阅读器上的文章。这两个插口分别是 control和nntpin，通常

是在/var/news/run目录里。

以上内容并不全面，还有其他的应用程序使用 Unix插口。

16.3 性能

比较Unix域插口与TCP插口的性能是非常有趣的。除了 TCP和UDP插口，修改公共域ttcp程序的一个版本，使之使用一个 Unix域流插口。我们在同一台主机上运行的两个程序副本之间传送16 777 216个字节的数据，结果如图16-2所示。

内 核	最快的TCP (字节/秒)	Unix域 (字节/秒)	增长 百分比
DEC OSF/1 V3.0	14 980 000	32 109 000	114 %
SunOS 4.1.3	4 877 000	11 570 000	137
BSD/OS V1.1	3 459 000	7 626 000	120
Solaries 2.4	2 829 000	3 570 000	26
AIX 3.2.2	1 592 000	3 948 000	148

图16-2 Unix域插口与TCP插口吞吐量的比较

我们感兴趣的是从一个TCP插口到一个Unix域插口速度的增长率，而不是绝对速度 (这些测试运行在五个不同系统上，覆盖了不同的处理器速度，在不同的行上进行速度比较毫无意义)。所有的内核都是源于伯克利，而不是 Solaris 2.4。我们可以看到，在源于伯克利内核上的Unix插口比TCP插口要快两倍多，在Solaris下增长率要慢得多。

SVR4以及源于它的Solaris，采用了完全不同的方法来实现 Unix域插口。[Rago 1993]的7.5节描述了基于流的SVR4中实现Unix域插口的方法。

在这些测试里，术语“ Fastest TCP(最块的TCP) ”意味着这些测试是在下列情况下进行的：将发送缓存和接收缓存都设置为 32 768(这个值要比一些系统中的默认值大)，直接指定环回地址而不是主机自己的 IP地址。在早期的 BSD实现中，如果指定了主机的 IP地址，那么在 ARP码执行之前分组不会发送到环回接口 (见卷1图2-4)，这稍微降低了性能 (这就是为什么定时测试运行时要指定环回地址)。这些主机都有一个本地子网的网络入口，其接口就是网络的设备驱动程序，卷1第87页中间网络入口140.252.13.32就是一个例子(SunOS 4.1.3)。较新的BSD内核有一条到主机本身 IP地址的路由，其接口就是环回驱动程序，卷 2图18-2中入口140.252.13.35就是一个例子(BSD/OS V2.0)。

我们将在讨论Unix域协议的实现后，在18.11节再返回到性能问题。

16.4 编码举例

为了说明如何缩小一个 TCP客户 - 服务器与一个 Unix域客户 - 服务器之间的差别，我们修改了图1-5和图1-7中的客户 - 服务器，使它们利用 Unix域协议通信。图16-3表示Unix域客户程序，与图1-5的不同之处用黑体字表示。

2-6 我们包含了<sys/un.h>头文件，客户和服务器的插口地址结构现在是 sockaddr_un类型。

11-15 socket调用的协议族是PF_UNIX，调用strncpy将与服务器相联系的路径名(从命令行参数得到)写入插口的地址结构。

unixcli.c

```
 1 #include    "cliserv.h"
 2 #include    <sys/un.h>

 3 int
 4 main(int argc, char *argv[])
 5 {                                   /* simple Unix domain client */
 6     struct sockaddr_un serv;
 7     char    request[REQUEST], reply[REPLY];
 8     int     sockfd, n;

 9     if (argc != 2)
10         err_quit("usage: unixcli <pathname of server>");

11     if ((sockfd = socket(PF_UNIX, SOCK_STREAM, 0)) < 0)
12         err_sys("socket error");

13     memset(&serv, 0, sizeof(serv));
14     serv.sun_family = AF_UNIX;
15     strncpy(serv.sun_path, argv[1], sizeof(serv.sun_path));

16     if (connect(sockfd, (SA) &serv, sizeof(serv)) < 0)
17         err_sys("connect error");

18     /* form request[] ... */

19     if (write(sockfd, request, REQUEST) != REQUEST)
20         err_sys("write error");
21     if (shutdown(sockfd, 1) < 0)
22         err_sys("shutdown error");

23     if ((n = read_stream(sockfd, reply, REPLY)) < 0)
24         err_sys("read error");

25     /* process "n" bytes of reply[] ... */

26     exit(0);
27 }
```

unixcli.c

图16-3 Unix域事务客户程序

当我们在下一章讨论具体实现时，就会看到导致这些差别的原因。

图16-4为Unix域服务器程序，与图1-7的不同之处用黑体字表示。

unixserv.c

```
 1 #include    "cliserv.h"
 2 #include    <sys/un.h>

 3 #define SERV_PATH   "/tmp/tcpipiv3.serv"

 4 int
 5 main()
 6 {                                   /* simple Unix domain server */
 7     struct sockaddr_un serv, cli;
 8     char    request[REQUEST], reply[REPLY];
 9     int     listenfd, sockfd, n, clilen;

10     if ((listenfd = socket(PF_UNIX, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");

12     memset(&serv, 0, sizeof(serv));
13     serv.sun_family = AF_UNIX;
14     strncpy(serv.sun_path, SERV_PATH, sizeof(serv.sun_path));
```

图16-4 Unix域事务服务器程序

```
15      if (bind(listenfd, (SA) &serv, sizeof(serv)) < 0)
16          err_sys("bind error");

17      if (listen(listenfd, SOMAXCONN) < 0)
18          err_sys("listen error");

19      for (;;) {
20          clilen = sizeof(cli);
21          if ((sockfd = accept(listenfd, (SA) &cli, &clilen)) < 0)
22              err_sys("accept error");

23          if ((n = read_stream(sockfd, request, REQUEST)) < 0)
24              err_sys("read error");

25          /* process "n" bytes of request[] and create reply[] ... */

26          if (write(sockfd, reply, REPLY) != REPLY)
27              err_sys("write error");

28          close(sockfd);
29      }
30  }
```

unixserv.c

图16-4 (续)

2-7 我们包含了<sys/un.h>头文件，并且定义了与服务器相联系的路径名 (通常路经名应在客户程序和服务器程序都包含的头文件中定义，为了简单，我们在这里定义)。现在的插口地址结构是sockaddr_un类型。

13-14 调用strncpy将路径名填入到服务器的插口地址结构。

16.5 小结

Unix域协议提供了进程间通信的一种形式，它使用同网络通信相同的编程接口 (插口)。Unix域协议既提供类似于 TCP的流插口，又提供类似于 UDP的数据报插口。从Unix域协议能获得的优点是速度：在一个源于伯克利的内核上， Unix域协议要比TCP/IP大约快两倍。

Unix域协议的最大用户是管道和X Window系统。如果X客户进程发现X服务器进程与X客户进程在同一台主机上，它就会使用 Unix域流连接来代替TCP连接， TCP客户-服务器程序和Unix域客户-服务器程序代码变化是很小的。

下面的两章描述Net/3内核中Unix域插口的实现。

第17章 Unix域协议：实现

17.1 概述

在uipc_usrreq.c文件中实现Unix域协议的源代码包含16个函数，总共大约有1000行C语言源程序，这与在卷2中实现UDP的800行源程序长度差不多，比实现TCP的4500行源程序要短得多。

我们分两章来描述Unix域协议的实现，下一章讨论I/O和描述符传递，其他的内容都在本章讨论。

17.2 代码介绍

在一个C文件中有16个Unix域函数，在其他C文件和两个头文件中还有其他有关的定义，如图17-1所示。

文　件	说　明
sys/un.h	sockaddr_un结构的定义
sys/unpcb.h	unpcb结构的定义
kern/uipc_proto.c	Unix域protosw{}和domain{}的定义
kern/uipc_usrreq.c	Unix域函数
kern/uipc_syscalls.c	pipe和socketpair系统调用

图17-1 在本章中讨论的文件

在本章我们也介绍pipe和socketpair系统调用，它们都使用本章描述的Unix域函数。

全局变量

图17-2列出了在本章和下一章中讨论的11个全局变量。

变　量	数 据 类 型	说　明
unixdomain	struct domain	域定义(图17-4)
unixsw	struct protosw	协议定义(图17-5)
sun_noname	struct sockaddr	包含空路径名的插口地址结构
unp_defer	int	延迟入口的无用单元收集计数器
unp_gcing	int	如果当前执行无用单元收集函数，就设置
unp_ino	ino_t	下一个分配的伪i_node号的值
unp_rights	int	当前传送中的文件描述符数
unpdg_recvspace	u_long	数据报插口接收缓存的默认范围，4096字节
unpdg_sendspace	u_long	数据报插口发送缓存的默认范围，2048字节
unpst_recvspace	u_long	流插口接收缓存的默认范围，4096字节
unpst_sendspace	u_long	流插口接收缓存的默认范围，4096字节

图17-2 在本章中介绍的全局变量

17.3 Unix domain和protosw结构

图17-3表示了Net/3系统中常见的三个domain结构，同时还有相应的protosw数组。

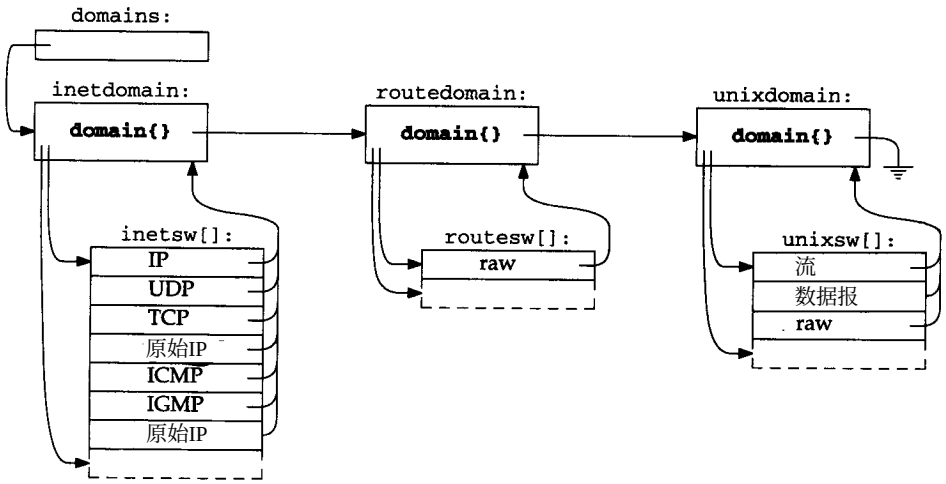


图17-3 domain 表和protosw 数组

卷2描述了Internet和路由选择域，图17-4描述了Unix域协议使用的domain结构(卷2图7-5)中的字段。

由于历史的原因，两个raw IP记录项在卷2图7-12中描述。

单 元	值	说 明
dom_family	*PF_UNIX*	域协议族
dom_name	*unix*	名字
dom_init	0	在Unix域中没有使用
dom_externalize	*unp_externalize*	外部化访问权(图18-12)
dom_dispose	*unp_dispose*	释放内部化权利(图18-14)
dom_protosw	*unixsw*	协议转换数组(图17-5)
dom_protoswNPROTOSW		协议转换数组的尾部指针
dom_next		由domaininit填充，卷2
dom_rtattach	0	在Unix域中没有使用
dom_rtoffset	0	在Unix域中没有使用
dom_maxrtkey	0	在Unix域中没有使用

图17-4 unixdomain 结构

仅有Unix domain定义了dom_externalize和dom_dispose两个函数，我们在第18章中讨论描述符传递时再描述这两个函数，由于 Unix 域没有路由选择表，所以domain结构的最后三个元素没有定义。

图17-5描述了unixsw结构的初始化(卷2图7-13描述了Internet协议的对应结构)。

定义三个协议：

- 与TCP相似的流协议；
- 与UDP相似的数据报协议；
- 与原始IP相似的raw协议。

uipc_proto.c

```
41 struct protosw unixsw[] =
42 {
43     {SOCK_STREAM, &unixdomain, 0, PR_CONNREQUIRED | PR_WANTRCVD | PR_RIGHTS,
44      0, 0, 0, 0,
45      uipc_usrreq,
46      0, 0, 0, 0,
47     },
48     {SOCK_DGRAM, &unixdomain, 0, PR_ATOMIC | PR_ADDR | PR_RIGHTS,
49      0, 0, 0, 0,
50      uipc_usrreq,
51      0, 0, 0, 0,
52     },
53     {0, 0, 0, 0,
54      raw_input, 0, raw_ctlinput, 0,
55      raw_usrreq,
56      raw_init, 0, 0, 0,
57     },
58 };
```

uipc_proto.c

图17-5 unixsw 数组的初始化

由于Unix 域支持访问权(就是我们在下一章要讲的描述符传递), Unix域流协议和数据报协议都设置PR_RIGHTS标志。流协议的另外两个标志PR_CONNREQUIRED和PR_WANTRCVD与TCP的标志一样；数据报协议的两个标志PR_ATOMIC和PR_ADDR与UDP的标志一样。需要注意的是流协议与数据报协议定义的唯一一个函数指针是 uipc_usrreq，用它处理所有的用户请求。

在raw协议的protosw结构中的四个函数指针都是以raw_开头，与PR_ROUTE域中的一样，这些内容在卷2的第20章介绍。

作者从来没有听到过一个应用程序使用 Unix域的raw协议。

17.4 Unix域插口地址结构

图17-6描述了一个Unix 域插口地址结构的定义，一个sockaddr_un结构长度为106个字节。

un.h

```
38 struct sockaddr_un {
39     u_char  sun_len;            /* sockaddr length including null */
40     u_char  sun_family;         /* AF_UNIX */
41     char    sun_path[104];      /* path name (gag) */
42 };
```

un.h

图17-6 Unix 域插口地址结构

开始的两个域与其他的插口地址结构一样：地址族 (AF_UNIX)后紧跟着一个长度字节。

自从4.2BSD以来，注解“ gag ”就存在了，也许原作者并不喜欢使用路经名来标识Unix域插口，或者是因为一个完整的路径名太长以至在 mbuf中写不下(路经名的长度能达到1024字节)。

我们将要看到 Unix 域插口使用文件系统中的路经名来标识插口，并且路径名存储在sun_path中。sun_path的大小为104字节，一个mbuf的大小为128个字节，刚好存放插口地址结构和一个表示终止的空字节。如图 17-7所示。

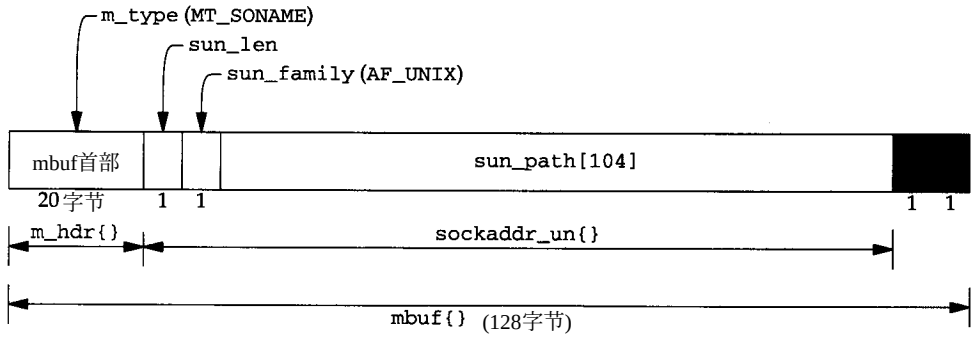


图17-7 存储在一个mbuf中的Unix域插口地址结构

我们将mbuf中的m_type字段设置成MT_SONAME，因为当mbuf含有一个插口地址结构时m_type就是这个普通值。虽然从图上看，最后两个字节没有使用，并且与这些插口相联系的最长路径名是104字节，但是我们将看到unp_bind和unp_connect两个函数允许一个路径名后面跟一个空字节时可以长达105字节。

Unix域插口在一些地方需要一个命名空间，由于文件系统的命名空间已经存在，所以就选定了路径名。与其他例子一样， Internet协议使用IP地址和端口号作为命名空间，系统V IPC([Stevens1992]的第14章)使用32比特密钥。由于Unix域客户进程用路径名来与服务器进程同步，从而通常使用绝对路径名 (以/开头)。如果使用相对路径名，客户程序和服务器程序必须在相同的目录中，或者服务器程序的绑定路经名不会被客户程序的connect和sendto发现。

17.5 Unix域协议控制块

Unix域插口有一个相关联的协议控制块 (PCB)，一个unpcb结构，我们在图17-8中描述了这个36字节的结构。

unpcb.h

```
60 struct unpcb {
61     struct socket *unp_socket;   /* pointer back to socket structure */
62     struct vnode *unp_vnode;     /* nonnull if associated with file */
63     ino_t   unp_ino;             /* fake inode number */
64     struct unpcb *unp_conn;      /* control block of connected socket */
65     struct unpcb *unp_refs;      /* referencing socket linked list */
66     struct unpcb *unp_nextref;   /* link in unp_refs list */
67     struct mbuf *unp_addr;       /* bound address of socket */
68     int     unp_cc;              /* copy of rcv.sb_cc */
69     int     unp_mbcnt;           /* copy of rcv.sb_mbcnt */
70 };

71 #define sotounpcb(so)    ((struct unpcb *)((so)->so_pcb))
```

unpcb.h

图17-8 Unix域协议控制块

不像路由域中使用的Internet PCB和控制块，这两者都是通过内核MALLOC函数来分配的(分别见卷2图20-18和图22-6)，而unpcb结构却存储在mbuf中，这可能是一个历史的人为因素。

另一个不同点是除了Unix域协议控制块以外，所有的控制块都保留在一个双向循环链表上，当数据到达时能通过查找这个链表将数据传递给相应的插口。对于所有的Unix域协议控制块而言，没有必要维护这样的链表，因为同样的操作，也就是当客户进程调用connect时，查找服务器的控制块是通过内核中已有的路径名查找函数来实现的。一旦找到服务器的unpcb，就可以将它的地址存储在客户进程的unpcb中，因为Unix域插口的客户进程与服务器进程在相同的主机上。

图17-9描述了处理Unix域插口的不同数据结构的关系，在这个图中我们描述了两个Unix

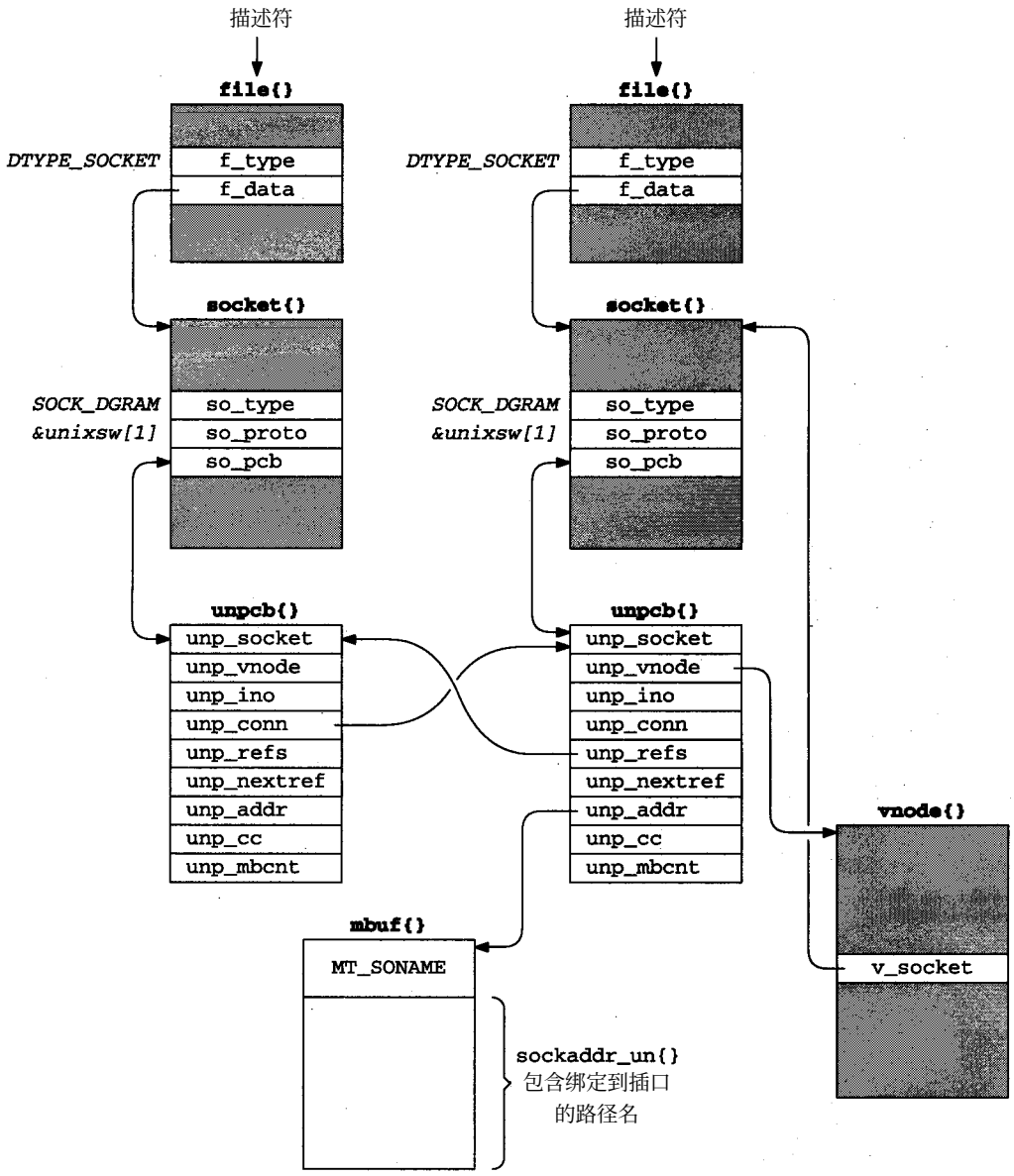


图17-9 互相连接的两个Unix域数据报插口

域数据报插口，我们假定右边的(服务器进程)插口已经绑定了一个路径名到它的插口，左边的(客户进程)插口已经连接到服务器的路径名上。

客户进程PCB的unp_conn单元指向服务器进程的PCB，服务器进程的unp_refs指向连接到这个PCB上的第一个客户进程（不像流插口，多个数据报客户进程可以连接到同一个服务器进程上，在17.11节我们要详细讨论Unix域数据报插口的连接)。

服务器的unp_vnode单元指向vnode，vnode与绑定到服务器插口的路径名相联系，它的v_socket单元指向服务器的socket，这就是定位一个已经绑定了路径名的unpcb所需的链接。例如，当服务器绑定了一个路径名到它的Unix域插口时，就会创建一个vnode结构，并且将unpcb的指针存储在v_node的v_socket中。当客户进程连接到服务器上时，内核中的路径名查找代码定位v-node，然后从v_socket指针获得服务器进程的unpcb指针。

被绑定到服务器插口的名字包含在socksddr_un结构中，sockaddr_un结构本身包含在unp_sddr指向的mbuf结构中。Unix 的v-node从来没有包含指向v-node的路径名，因为在一个Unix文件系统中多个名字(即目录记录项)能同时指向一个给定的文件(即v-node)。

图17-9表示两个连接的数据报插口，在图 17-26中我们将看到，处理流插口时与这里有些不同。

17.6 uipc_usrreq函数

在图17-5中我们看到，对于流和数据报协议，unixsw结构中引用的唯一函数是uipc_usrreq，图17-10给出了这个函数的要点。

uipc_usrreq.c

```
47 int
48 uipc_usrreq(so, req, m, nam, control)
49 struct socket *so;
50 int     req;
51 struct mbuf *m, *nam, *control;
52 {
53     struct unpcb *unp = sotounpcb(so);
54     struct socket *so2;
55     int     error = 0;
56     struct proc *p = curproc;    /* XXX */

57     if (req == PRU_CONTROL)
58         return (EOPNOTSUPP);
59     if (req != PRU_SEND && control && control->m_len) {
60         error = EOPNOTSUPP;
61         goto release;
62     }
63     if (unp == 0 && req != PRU_ATTACH) {
64         error = EINVAL;
65         goto release;
66     }
67     switch (req) {

            /* switch cases (discussed in following sections) */

246     default:
```

图17-10 uipc_usrreq 函数体

```
247             panic("piusrreq");
248         }
249     release:
250         if (control)
251             m_freem(control);
252         if (m)
253             m_freem(m);
254         return (error);
255 }
```
uipc_usrreq.c

图17-10 (续)

1. 无效的PRU_CONTROL请求

57-58 PRU_CONTROL请求来自ioctl系统调用，不被Unix域支持。

2. 仅为PRU_SEND支持的控制信息

59-62 如果进程传送控制信息(使用sendmsg系统调用)，请求必须是PRU_SEND；否则，返回一个错误。描述符在使用该请求的控制信息的进程间传递，这部分我们在第18章中讨论。

3. 插口必须有一个控制块

63-66 如果socket结构没有指向一个Unix域控制块，请求必须是PRU_ATTACH；否则，返回一个错误。

67-248 在下面几节中我们讨论这个函数的每一个case语句，以及调用的不同unp_*xxx*函数。

249-255 释放任何控制信息和数据mbuf，然后函数返回。

17.7 PRU_ATTACH请求和unp_attach函数

当一个连接请求到达一个处于监听状态的流插口时，socket系统调用和sonewconn函数(卷2图15-29)产生PRU_ATTACH请求，如图17-11所示。

uipc_usrreq.c
```
68      case PRU_ATTACH:
69          if (unp) {
70              error = EISCONN;
71              break;
72          }
73          error = unp_attach(so);
74          break;
```
uipc_usrreq.c

图17-11 PRU_ATTACH 请求

unp_attach函数完成这个请求的所有处理工作，如图17-12所示。socket结构已经被插口层分配和初始化，现在轮到协议层分配和初始化自身的协议控制块，在本例中这个协议控制块为unpcb结构。

uipc_usrreq.c
```
270 int
271 unp_attach(so)
272 struct socket *so;
273 {
274     struct mbuf *m;
275     struct unpcb *unp;
276     int     error;
```

图17-12 unp_attack 函数

```
277     if (so->so_snd.sb_hiwat == 0 || so->so_rcv.sb_hiwat == 0) {
278         switch (so->so_type) {

279         case SOCK_STREAM:
280             error = soreserve(so, unpst_sendspace, unpst_recvspace);
281             break;

282         case SOCK_DGRAM:
283             error = soreserve(so, unpdg_sendspace, unpdg_recvspace);
284             break;

285         default:
286             panic("unp_attach");
287         }
288         if (error)
289             return (error);
290     }
291     m = m_getclr(M_DONTWAIT, MT_PCB);
292     if (m == NULL)
293         return (ENOBUFS);
294     unp = mtod(m, struct unpcb *);
295     so->so_pcb = (caddr_t) unp;
296     unp->unp_socket = so;
297     return (0);
298 }
```

uipc_usrreq.c

图17-12 (续)

1. 设置插口高水位标记

277-290 如果插口发送和接收的高水位标记为0，则soreserve将它们设置成图17-2所示的默认值，高水位标记限制了存放在插口发送和接收缓存中的数据量。当通过socket系统调用来调用unp_attach时，这两个标记都为0，但是当通过sonewconn调用unp_attach时，它们等于监听插口中的值。

2. 分配并初始化PCB

291-296 m_getclr获得一个mbuf用于unpcb结构，将mbuf清零并将类型设置成MT_PCB。注意所有的PCB单元都被初始化为0。通过so_pcb和unp_socket指针将socket和unpcb结构连接起来。

17.8 PRU_DETACH请求和unp_detach函数

当一个插口关闭时发出PRU_DETACH请求(卷2图15-39)，这个请求跟随在PRU_DISCONNECT请求(仅针对有连接的插口)的后面，如图17-13所示。

uipc_usrreq.c

```
75      case PRU_DETACH:
76          unp_detach(unp);
77          break;
```

uipc_usrreq.c

图17-13 PRU_DETACH 请求

75-77 图17-14中的unp_detach函数完成PRU_DETACH请求的所有处理工作。

1. 释放v-node

303-307 如果插口与一个v-node相联系，那么将指向PCB结构的指针置为0，并且调用vrele释放v_node。

uipc_usrreq.c

```
299 void
300 unp_detach(unp)
301 struct unpcb *unp;
302 {
303     if (unp->unp_vnode) {
304         unp->unp_vnode->v_socket = 0;
305         vrele(unp->unp_vnode);
306         unp->unp_vnode = 0;
307     }
308     if (unp->unp_conn)
309         unp_disconnect(unp);
310     while (unp->unp_refs)
311         unp_drop(unp->unp_refs, ECONNRESET);
312     soisdisconnected(unp->unp_socket);
313     unp->unp_socket->so_pcb = 0;
314     m_freem(unp->unp_addr);
315     (void) m_free(dtom(unp));
316     if (unp_rights) {
317         /*
318          * Normally the receive buffer is flushed later,
319          * in sofree, but if our receive buffer holds references
320          * to descriptors that are now garbage, we will dispose
321          * of those descriptor references after the garbage collector
322          * gets them (resulting in a "panic: closef: count < 0").
323          */
324         sorflush(unp->unp_socket);
325         unp_gc();
326     }
327 }
```

uipc_usrreq.c

图17-14 unp_detach 函数

2. 如果插口连接了其他插口，则断开连接

308-309 如果关闭的插口连接到另一个插口上，那么unp_disconnect就要断开这两个插口的连接，这种情况在流和数据报插口中都会发生。

3. 断开连接到关闭插口的插口

310-311 如果其他的数据报插口连接到这个插口，则调用unp_drop断开这些连接，那些插口就会接收到ECONNRESET错误。while循环检查连接到这个unpcb的所有unpcb结构链表。函数unp_drop调用unp_disconnect，它改变PCB的unp_refs单元去指向链表的下一个单元。当整个链表已经被处理后，PCB的unp_refs指针将为0。

312-313 被关闭的插口由soisdisconnect断开连接，指向PCB的socket结构中的指针置为0。

4. 释放地址和PCB mbuf

314-315 如果插口已经绑定到一个地址，m_freem就释放存储这个地址的mbuf。注意程序不检查unp_addr是否为空，因为m_freem会检查。unpcb由m_free来释放。

这个对m_free的调用应当移到函数的末尾，因为指针unp可能会在下一段程序里使用。

5. 检查被传送的描述符

316-326 如果内核里任何进程传来了描述符，则unp_rights为非0，这会导致调用

sorflush和unp_gc(无用单元收集函数)。我们将在第18章中讨论描述符的传送。

17.9 PRU_BIND请求和unp_bind函数

可以通过bind将Unix 域中的流和数据报插口绑定到文件系统中的路径名上， bind系统调用产生PRU_BIND请求，如图17-15所示。

uipc_usrreq.c

```
78      case PRU_BIND:
79          error = unp_bind(unp, nam, p);
80          break;
```

uipc_usrreq.c

图17-15 PRU_BIND 请求

78-80 所有的工作都由unp_bind函数来完成，如图17-16所示。

1. 初始化nameidata结构

338-339 unp_bind分配一个nameidata结构，这个结构封装所有传给 namei函数的参数，并使用NDINIT宏来初始化这个结构。 CREATE参数指定要创建的路径名， FOLLOW允许紧跟的符号连接， LOCKPARENT指明在返回时必须要锁定父亲的v-node (防止我们在完成工作之前其他进程修改v-node)。UIO_SYSSPACE指明路径名在内核中(由于bind系统调用将路径名从用户空间复制到一个mbuf中)。soun->sun_path是路径名的起始地址(它被作为nam参数传送给unp_bind)。最后， p是指向发布bind系统调用的进程的proc结构的指针，这个结构包含所有有关一个进程的信息，内核需要一直将该进程存放在内存中。 NDINIT宏仅仅初始化这个结构，对namei的调用在这个函数后面。

uipc_usrreq.c

```
328 int
329 unp_bind(unp, nam, p)
330 struct unpcb *unp;
331 struct mbuf *nam;
332 struct proc *p;
333 {
334     struct sockaddr_un *soun = mtod(nam, struct sockaddr_un *);
335     struct vnode *vp;
336     struct vattr vattr;
337     int     error;
338     struct nameidata nd;

339     NDINIT(&nd, CREATE, FOLLOW | LOCKPARENT, UIO_SYSSPACE, soun->sun_path, p);
340     if (unp->unp_vnode != NULL)
341         return (EINVAL);
342     if (nam->m_len == MLEN) {
343         if (*(mtod(nam, caddr_t) + nam->m_len - 1) != 0)
344             return (EINVAL);
345     } else
346         *(mtod(nam, caddr_t) + nam->m_len) = 0;
347 /* SHOULD BE ABLE TO ADOPT EXISTING AND wakeup() ALA FIFO's */
348     if (error = namei(&nd))
349         return (error);
350     vp = nd.ni_vp;
351     if (vp != NULL) {
352         VOP_ABORTOP(nd.ni_dvp, &nd.ni_cnd);
353         if (nd.ni_dvp == vp)
```

图17-16 unp_bind 函数

```
354             vrele(nd.ni_dvp);
355         else
356             vput(nd.ni_dvp);
357         vrele(vp);
358         return (EADDRINUSE);
359     }
360     VATTR_NULL(&vattr);
361     vattr.va_type = VSOCK;
362     vattr.va_mode = ACCESSPERMS;
363     if (error = VOP_CREATE(nd.ni_dvp, &nd.ni_vp, &nd.ni_cnd, &vattr))
364         return (error);
365     vp = nd.ni_vp;
366     vp->v_socket = unp->unp_socket;
367     unp->unp_vnode = vp;
368     unp->unp_addr = m_copy(nam, 0, (int) M_COPYALL);
369     VOP_UNLOCK(vp, 0, p);
370     return (0);
371 }
```

uipc_usrreq.c

图17-16 (续)

历史上，在文件系统中查询路径名的函数名一直是 namei，它代表“ name-to-inode ”。这个函数要搜索整个文件系统去查找指定的名字，如果成功，就初始化内核中的inode结构，这个结构包含从磁盘上得到的文件的 i_node信息的副本。尽管 v-node已经取代了i-node，但是术语namei仍然保留了下来。

这是我们第一次涉及到BSD内核中文件系统代码。BSD内核支持许多不同的文件系统类型：标准的磁盘文件系统 (有时也叫作“ 快速文件系统 ”)，网络文件系统(NFS)，CD-ROM文件系统，MS-DOS文件系统，基于存储器的文件系统 (对于目录，例如/tmp)，等等。[Kleiman 1986]描述了一个早期的v-node实现。以VOP_作为名字开始的函数一般是v-node操作函数。这样的函数大约有 40个，当被调用时，每个函数调用一个文件系统定义的函数去执行这个操作。以一个小写字母 v开头的函数是内核函数，这些函数可能调用一个或更多的 VOP_函数。例如，vput调用VOP_UNLOCK，然后再调用vrele。vrele函数释放一个v-node:v-node的引用计数器递减，如果达到0，就调用VOP_INACTIVE。

2. 检查插口是否被绑定

340-341 如果插口PCB的unp_vnode非空，插口就已经被绑定，这是一个错误。

3. 以空字符(null)结束的路径名

342-346 如果包含sockaddr_un结构的mbuf长度是108(MLEN)，长度值是从bind系统调用的第三个参数复制的，则 mbuf的最后一个字节必须是一个空字节。这就保证路径名以空字符结尾，当在文件系统中查找路径名时这是必需的 (卷2图15-20中的sockargs函数保证由进程传送的插口地址结构长度不超过 108字节)。如果mbuf的长度小于108个字节，则在路径名的结尾存放一个空字节，以免进程没有以空字符来结束路径名。

4. 在文件系统中查找路径名

347-349 namei在文件系统中查找路径名，并且尽可能在相应的目录中为指定的路径名创建一个记录项。例如，如果绑定到插口的路径名是 /tmp/.Xll-unix/X0，那么文件名X0必

须被加到目录 /tmp/.Xll-unix中，包含X0的记录项的目录叫作父目录。如果目录/tmp/.Xll-unix不存在，或者如果存在，但是已经包含一个 X0的文件，那么就要返回一个错误。另一个可能的错误是调用进程没有权限在父目录中创建一个新的文件。从 namei想得到的结果是从函数返回一个 0值，nd.ni_vp返回的是一个空指针 (文件不存在)。如果这两个条件都正确，那么nd.ni_dvp就包含要创建新文件名的加锁父目录。

347行的注释指的是如果路径名已经存在将导致 bind返回错误。所以大部分绑定Unix域插口的应用程序在调用bind之前先调用unlink删除已存在的路径名。

5. 路径名已经存在

350-359 如果nd.ni_vp非空，那么路径名就已经存在。 v-node引用被释放，并且返回EADDRINUSE给进程。

6. 创建v-node

360-365 VATTR_NULL宏初始化vattr结构，类型被设置为VSOCK(一个插口)，访问模式设置为八进制777(ACCESSPERMS)。这九个权限比特允许文件所有者、组里的成员和其他用户(也就是每一个用户)执行读、写和执行操作。在指定的目录中，文件由文件系统的创建函数间接通过VOP_CREATE函数创建。传递给创建函数的参数是 nd.ni_dvp(父目录v-node的指针)，nd.ni_cnd(来自namei需要传送给VOP函数的附加信息)，以及vattr结构。第二个参数nd.ni_vp接收返回信息，nd.ni_vp指向新创建的v-node(如果创建成功)。

7. 链接结构

365-367 vnode和socket通过v_socket和unp_vnode指针互相指向对方。

8. 保存路径名

368-371 调用m_copy将刚刚绑定到插口的路径名复制到一个 mbuf中，PCB的unp_addr指向这个新的mbuf。将v-node解锁。

17.10 PRU_CONNECT请求和unp_connect函数

图17-17描述了PRU_LISTEN和PRU_CONNECT请求。

uipc_usrreq.c

```
81      case PRU_LISTEN:
82          if (unp->unp_vnode == 0)
83              error = EINVAL;
84          break;

85      case PRU_CONNECT:
86          error = unp_connect(so, nam, p);
87          break;
```

uipc_usrreq.c

图17-17 PRU_LISTEN 和PRU_CONNECT 请求

1. 验证监听插口是否已经被绑定

81-84 只能在一个已经绑定了一个路径名的插口上执行 listen系统调用。TCP没有这个需求，在卷2图30-3我们看到对一个没有绑定的 TCP插口调用listen时，TCP就会选择一个临时的端口，并把它分配给插口。

85-87 PRU_CONNECT请求的所有处理工作都由unp_connect函数来执行，函数的第一部分如图17-18所示。对于流插口，该函数被 PRU_CONNECT请求调用；当临时连接一个无连接

的数据报插口时，该函数被PRU_SEND请求调用。

uipc_usrreq.c

```
372 int
373 unp_connect(so, nam, p)
374 struct socket *so;
375 struct mbuf *nam;
376 struct proc *p;
377 {
378     struct sockaddr_un *soun = mtod(nam, struct sockaddr_un *);
379     struct vnode *vp;
380     struct socket *so2, *so3;
381     struct unpcb *unp2, *unp3;
382     int     error;
383     struct nameidata nd;
384     NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF, UIO_SYSSPACE, soun->sun_path, p);
385     if (nam->m_data + nam->m_len == &nam->m_dat[MLEN]) {    /* XXX */
386         if (*(mtod(nam, caddr_t) + nam->m_len - 1) != 0)
387             return (EMSGSIZE);
388     } else
389         *(mtod(nam, caddr_t) + nam->m_len) = 0;
390     if (error = namei(&nd))
391         return (error);
392     vp = nd.ni_vp;
393     if (vp->v_type != VSOCK) {
394         error = ENOTSOCK;
395         goto bad;
396     }
397     if (error = VOP_ACCESS(vp, VWRITE, p->p_ucred, p))
398         goto bad;
399     so2 = vp->v_socket;
400     if (so2 == 0) {
401         error = ECONNREFUSED;
402         goto bad;
403     }
404     if (so->so_type != so2->so_type) {
405         error = EPROTOTYPE;
406         goto bad;
407     }
```

uipc_usrreq.c

图17-18 unp_connect 函数：第一部分

2. 初始化用作路径名查找的nameidata结构

383-384 nameidata结构由NDINIT宏进行初始化。LOOKUP参数指明应当查找的路径名，FOLLOW允许紧跟的符号连接，LOCKLEAF参数指明返回时必须锁定v-node(防止在执行结束前其他进程修改这个v-node)，UIO_SYSSPACE参数指明路径名在内核中，soun->sun_path是路径名的起始地址(它被作为nam参数传递给unp_connect)。p指向发布connect或sendto系统调用的进程的proc结构。

3. 以空字节结束路径名

385-389 如果插口地址结构的长度是108字节，最后一个字节必须为空，否则在路径名的结尾要存储一个空字节。

这段代码与图17-16中的代码相似，但实际上是不同的。不仅第一个if语句不同，而且当最后一个字节非空时返回的错误也不同：这里是EMSGSIZE，而图17-16中是

EINVAL。另外，这个测试对检查数据是否包含在一个簇中有负面影响，虽然这可能是偶然的，因为sockargs函数从来不会把插口地址结构放进一个簇中。

4. 查找路径名并检验其正确性

390-398 namei在文件系统中查找路径名，如果返回值是OK，那么在nd.ni_vp中就返回vnode结构的指针。v-node的类型必须是VSOCK，并且当前进程对插口一定要有写权限。

5. 验证插口是否已绑定到路径名

399-403 一个插口当前必须被绑定到路径名上，这就是说，在v-node中的v_socket指针必须非空。如果情况不是这样，连接就要被拒绝。如果服务器当前没有运行，但是在上一次运行时路径名留在文件系统中，这种情况就有可能发生。

6. 验证插口类型

404-407 连接的客户进程插口(so)的类型必须与被连接的服务器进程插口(so2)的类型相同。也就是说，一个流插口不能连接到一个数据报插口或者相反。

图17-19描述了unp_connect函数的剩余部分，它首先处理连接流插口，然后调用unp_connect2去链接两个unpcb结构。

uipc_usrreq.c

```
408     if (so->so_proto->pr_flags & PR_CONNREQUIRED) {
409         if ((so2->so_options & SO_ACCEPTCONN) == 0 ||
410             (so3 = sonewconn(so2, 0)) == 0) {
411             error = ECONNREFUSED;
412             goto bad;
413         }
414         unp2 = sotounpcb(so2);
415         unp3 = sotounpcb(so3);
416         if (unp2->unp_addr)
417             unp3->unp_addr =
418                 m_copy(unp2->unp_addr, 0, (int) M_COPYALL);
419         so2 = so3;
420     }
421     error = unp_connect2(so, so2);
422   bad:
423     vput(vp);
424     return (error);
425 }
```

uipc_usrreq.c

图17-19 unp_connect 函数：第二部分

7. 连接流插口

408-415 流插口需要特殊处理，因为必须根据监听插口创建一个新的插口。首先，服务器插口必须是监听插口：SO_ACCEPTCONN标志必须被设置(由卷2图15-24的solisten函数来完成)。然后调用sonewconn创建一个新的插口，sonewconn还把这个新的插口放到监听插口的未完成的连接队列中。

8. 复制绑定到监听插口的名字

416-418 如果监听插口包含一个指向mbuf的指针，mbuf包含一个sockaddr_un，并且sockaddr_un带有绑定到插口的路径名(这应当总是对的)，那么调用m_copy将该mbuf复制给新创建的插口。

图17-20给出了在so2=so3赋值之前的不同结构的状态，步骤如下：

- 服务器进程调用socket创建最右边的file、socket和unpcb结构，然后调用bind创建对vnode和包含路径名的mbuf的引用。随后调用listen，允许客户进程发起连接。
- 客户进程调用socket创建最左边的file、socket和unpcb结构，然后调用connect，connect调用unp_connect。
- 我们称中间的socket结构为“已连接的服务器插口”，它由sonewconn创建，

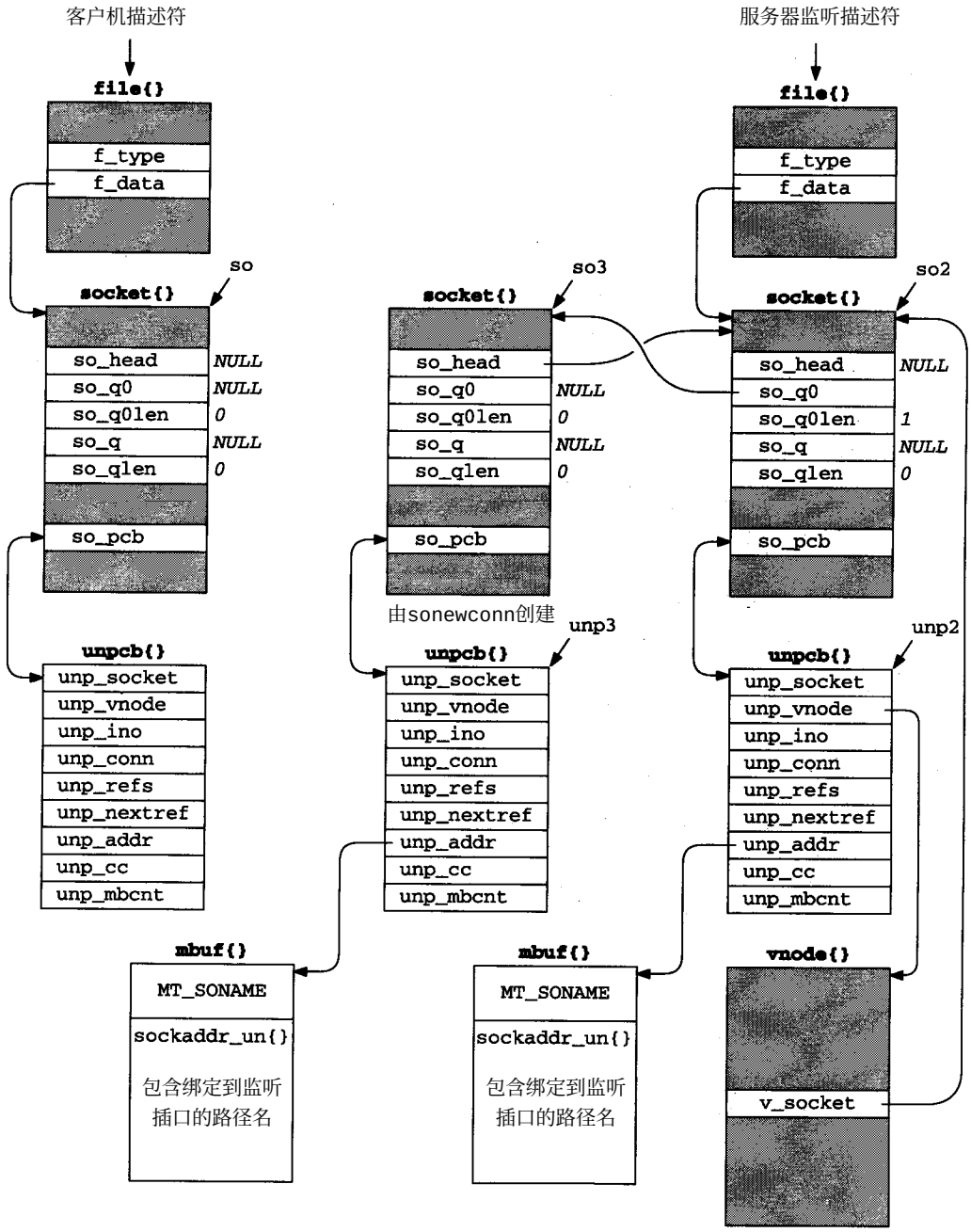


图17-20 流插口的connect 调用中的各种结构

sonewconn创建完该结构后发出PRU_ATTACH请求，创建相应的unpcb结构。

- sonewconn也调用soqinseque将刚产生的socket放入监听插口的未完成的连接队列中(我们假定队列开始是空的)。我们还看到监听插口的已完成连接队列(so_q和so_qlen)为空，新建socket的so_head指针反过来指向监听插口。
- unp_connect调用m_copy创建包含绑定到监听插口的路径名的mbuf的副本，中间的unpcb指向这个mbuf。我们将看到getpeername系统调用需要这个副本。
- 最后要注意的是，还没有一个file结构指向新建的socket(事实上是通过sonewconn设置SS_NOFDREF标志来说明这一点的)。当监听服务器进程调用accept时，就会给该socket分配一个file结构和对应的文件描述符。

vnode指针没有从监听插口复制到连接的服务器插口。vnode结构的唯一作用就是允许客户进程通过v_socket指针调用connect定位相应的服务器的socket结构。

9. 连接两个流或数据报插口

421 unp_connect中的最后一步是调用unp_connect2(下一节描述)，这对于流和数据报插口是一样的。就图17-20而言，该函数连接最左边的两个unpcb结构的unp_conn字段，并且将新创建的插口从监听服务器的socket的未完成连接队列移到已完成连接队列中，我们将在后面的章节中描述最终的数据结构(图17-26)。

17.11 PRU_CONNECT2请求和unp_connect2函数

图17-21中的PRU_CONNECT2请求仅仅作为socketpair系统调用产生的一个结果，而且这个请求只在Unix域中得到支持。

uipc_usrreq.c

```
88      case PRU_CONNECT2:
89          error = unp_connect2(so, (struct socket *) nam);
90          break;
```

uipc_usrreq.c

图17-21 PRU_CONNECT2 请求

88-90 这个请求的所有处理工作都由unp_connect2函数来完成，正如我们在图17-22中看到的一样，unp_connect2函数又是从内核中的其他两个地方调用的。

我们将在17.12节介绍socketpair系统调用和soconnect2函数，在17.13节介绍pipe系统调用。图17-23描述了unp_connect2函数。

1. 检验插口类型

426-434 两个参数都是指向socket结构的指针：so连接到so2。首先检查两个插口的类型是否相同：是流插口或者是数据报插口。

2. 把第一个插口连接到第二个插口

435-436 通过字段unp_conn将第一个unpcb连接到第二个unpcb，然而，下面的步骤在流和数据报之间是不同的。

3. 连接数据报插口

438-442 PCB的unp_nextref和unp_refs字段连接数据报插口。例如，考虑一个绑定了路径名/tmp/foo的数据报服务器插口，然后一个数据报客户进程连接到这个路径名。图17-24给出了在unp_connect2返回后得到的unpcb结构(为了简便起见，我们没有描述相应

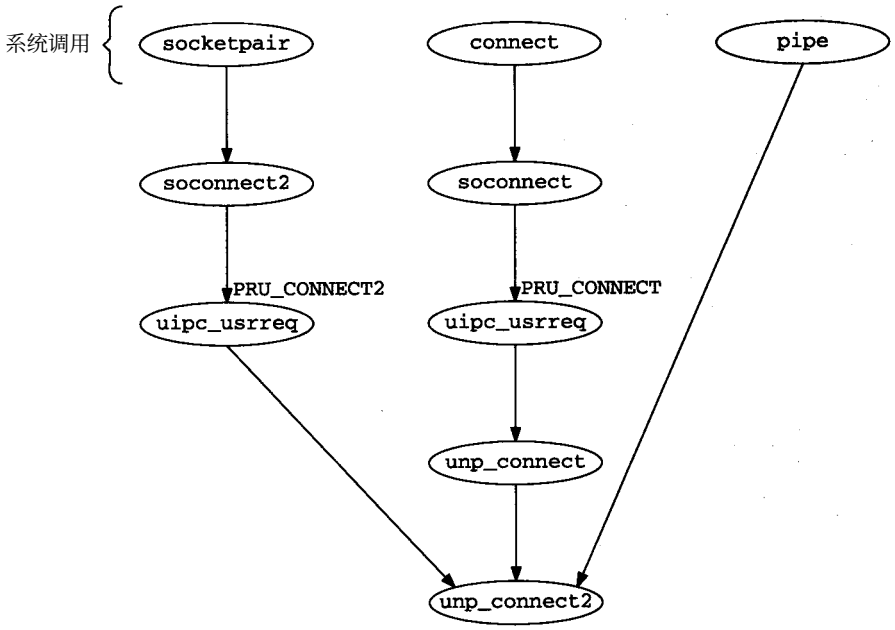


图17-22 unp_connect2 函数的调用者

uipc_usrreq.c

```
426 int
427 unp_connect2(so, so2)
428 struct socket *so;
429 struct socket *so2;
430 {
431     struct unpcb *unp = sotounpcb(so);
432     struct unpcb *unp2;
433     if (so2->so_type != so->so_type)
434         return (EPROTOTYPE);
435     unp2 = sotounpcb(so2);
436     unp->unp_conn = unp2;
437     switch (so->so_type) {
438     case SOCK_DGRAM:
439         unp->unp_nextref = unp2->unp_refs;
440         unp2->unp_refs = unp;
441         soisconnected(so);
442         break;
443     case SOCK_STREAM:
444         unp2->unp_conn = unp;
445         soisconnected(so);
446         soisconnected(so2);
447         break;
448     default:
449         panic("unp_connect2");
450     }
451     return (0);
452 }
```

uipc_usrreq.c

图17-23 unp_connect2 函数

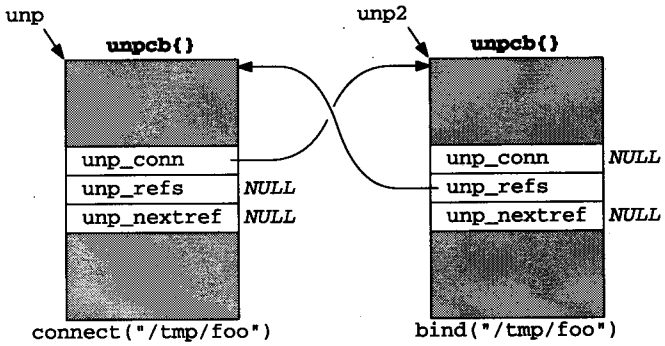


图17-24 连接的数据报插口

的file或socket结构，或者与最右边插口相连接的 vnode)。我们描述了在unp_connect2中用到的两个指针unp和unp2。

对于一个已经有连接的数据报插口，unp_refs指向连接到该插口的所有插口的链表的第一个PCB。通过unp_nextref指针遍历这个链表。

图17-25表示了第三个数据报插口 (左边的那个)连接到同一服务器后的三个 PCB的状态，绑定路径名都是/tmp/foo。

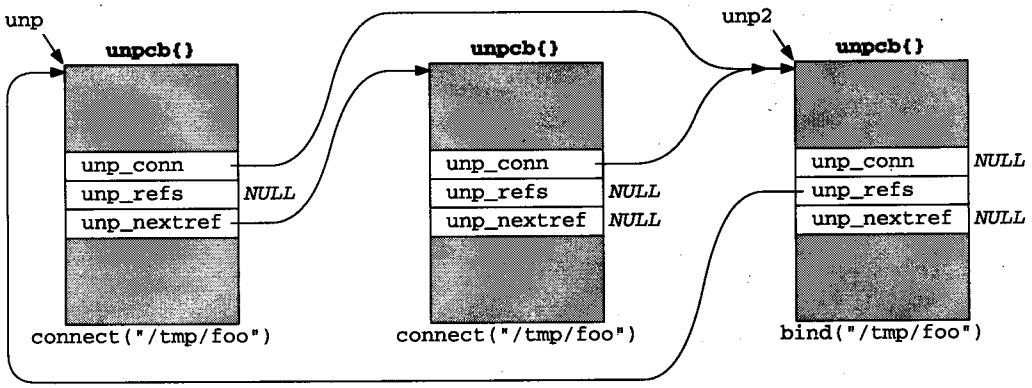


图17-25 另一个插口(左边)连接到右边的插口

两个PCB字段unp_refs和unp_nextref必须分开，因为图17-25中右边的插口自己能连接到其他的数据报插口。

4. 连接流插口

443-447 流插口的连接与数据报插口的连接是不同的，这是因为只能有一个客户进程连接到一个流插口上(服务器进程)，客户进程和服务器进程的PCB的unp_conn指针分别指向对方的PCB，如图17-26所示(这个图是图17-20的延续)。

这个图中的另一个变化是对于带有so2参数的soisconnected的调用，这个调用将插口从监听插口的未完成连接队列(图17-20中的so_q0)移到已完成连接队列(so_q)中。accept要从这个队列中获取新创建的插口(卷2图15-34)。需要注意的是，soinconnected(卷2图15-30)设置so_state中的SS_ISCONNECTED标志，仅当插口的so_head指针非空时才将socket从未完成连接队列移到已完成连接队列(如果插口的so_head指针为空时，插口不在任何一个队列中)。所以，在图17-23中，对带有so参数的soisconnected的第一次调用仅仅改变so_state。

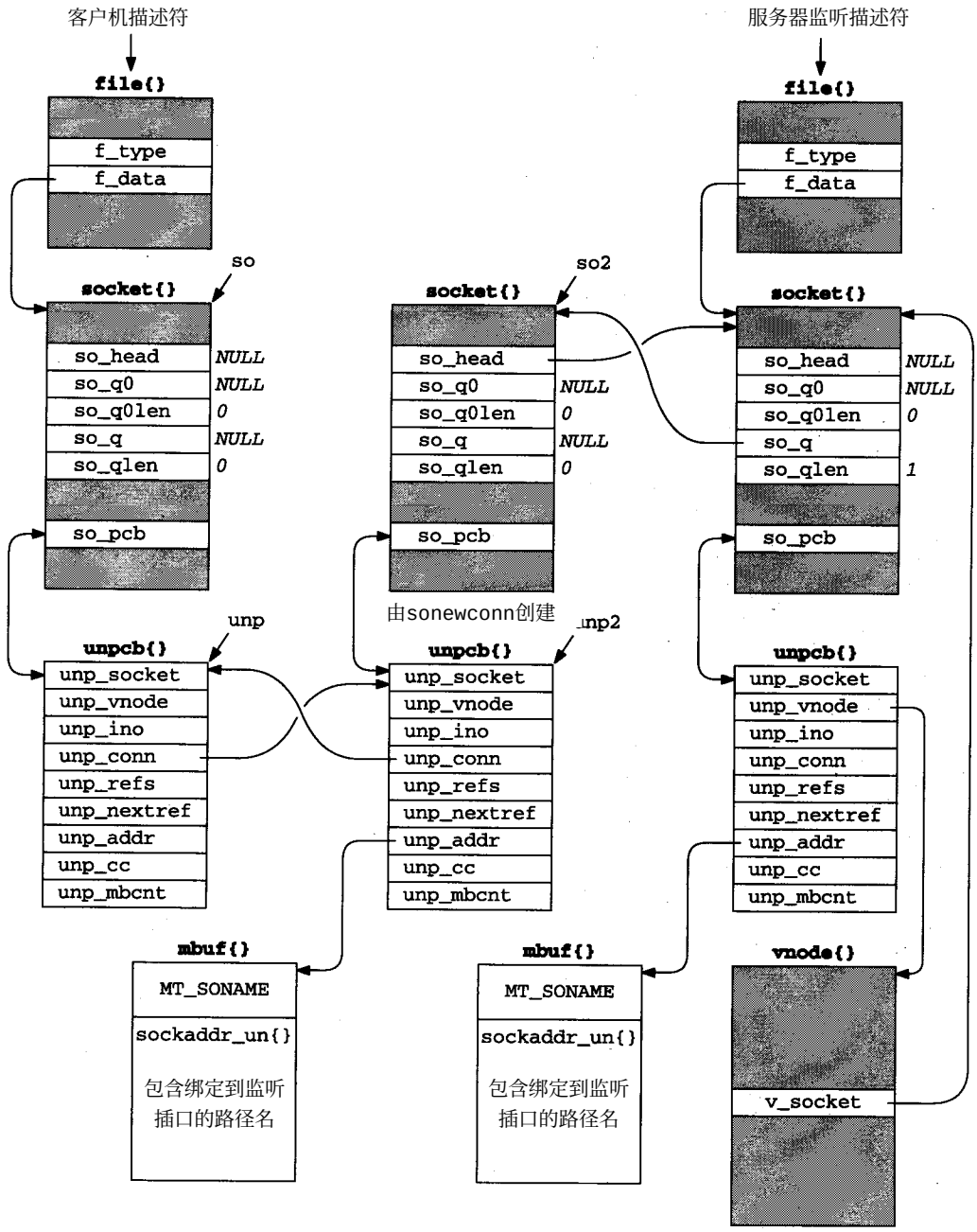


图17-26 已建连的流插口

17.12 socketpair系统调用

socketpair系统调用仅在Unix域中得到支持。它创建两个插口并连接它们，同时返回两个描述符，互相连接在一起。例如，一个用户进程发出调用：

```
int   fd[2];
socketpair(PF_UNIX, SOCK_STREAM, 0, fd);
```

创建一对连接在一起的全双工Unix域流插口。在fd[0]中返回第一个描述符，在fd[1]中返回第二个描述符。如果第二个参数是SOCK_DGRAM，则创建一对互相连接的Unix域数据报插口。如果调用成功，socketpair返回0；否则，返回-1。

图17-27描述了socketpair系统调用的实现。

uipc_syscalls.c

```
229 struct socketpair_args {
230     int     domain;
231     int     type;
232     int     protocol;
233     int    *rsv;
234 };

235 socketpair(p, uap, retval)
236 struct proc *p;
237 struct socketpair_args *uap;
238 int     retval[];
239 {
240     struct filedesc *fdp = p->p_fd;
241     struct file *fp1, *fp2;
242     struct socket *so1, *so2;
243     int     fd, error, sv[2];

244     if (error = socreate(uap->domain, &so1, uap->type, uap->protocol))
245         return (error);
246     if (error = socreate(uap->domain, &so2, uap->type, uap->protocol))
247         goto free1;

248     if (error = falloc(p, &fp1, &fd))
249         goto free2;
250     sv[0] = fd;
251     fp1->f_flag = FREAD | FWRITE;
252     fp1->f_type = DTYPE_SOCKET;
253     fp1->f_ops = &socketops;
254     fp1->f_data = (caddr_t) so1;

255     if (error = falloc(p, &fp2, &fd))
256         goto free3;
257     fp2->f_flag = FREAD | FWRITE;
258     fp2->f_type = DTYPE_SOCKET;
259     fp2->f_ops = &socketops;
260     fp2->f_data = (caddr_t) so2;
261     sv[1] = fd;

262     if (error = soconnect2(so1, so2))
263         goto free4;
264     if (uap->type == SOCK_DGRAM) {
265         /*
266          * Datagram socket connection is asymmetric.
267          */
268         if (error = soconnect2(so2, so1))
269             goto free4;
270     }
271     error = copyout((caddr_t) sv, (caddr_t) uap->rsv, 2 * sizeof(int));
272     retval[0] = sv[0];          /* XXX ??? */
273     retval[1] = sv[1];          /* XXX ??? */
274     return (error);

275 free4:
```

图17-27 socketpair 系统调用

```
276     ffree(fp2);
277     fdp->fd_ofiles[sv[1]] = 0;
278   free3:
279     ffree(fp1);
280     fdp->fd_ofiles[sv[0]] = 0;
281   free2:
282     (void) soclose(so2);
283   free1:
284     (void) soclose(so1);
285     return (error);
286 }
```

uipc_syscalls.c

图17-27 (续)

1. 参数

229-239 四个整型参数，从domian到rsv，在本节开始部分用户调用socketpair的例子中进行了描述。函数socketpair定义中描述的三个参数(p、uap和retval)是传送到内核中的系统调用的参数。

2. 创建两个插口和两个描述符

244-261 调用socreate两次，创建两个插口。两个描述符中的第一个由falloc分配。在fd中返回描述符的值，而指向相应file结构的指针在fp1中返回。设置FREAD和FWRITE标

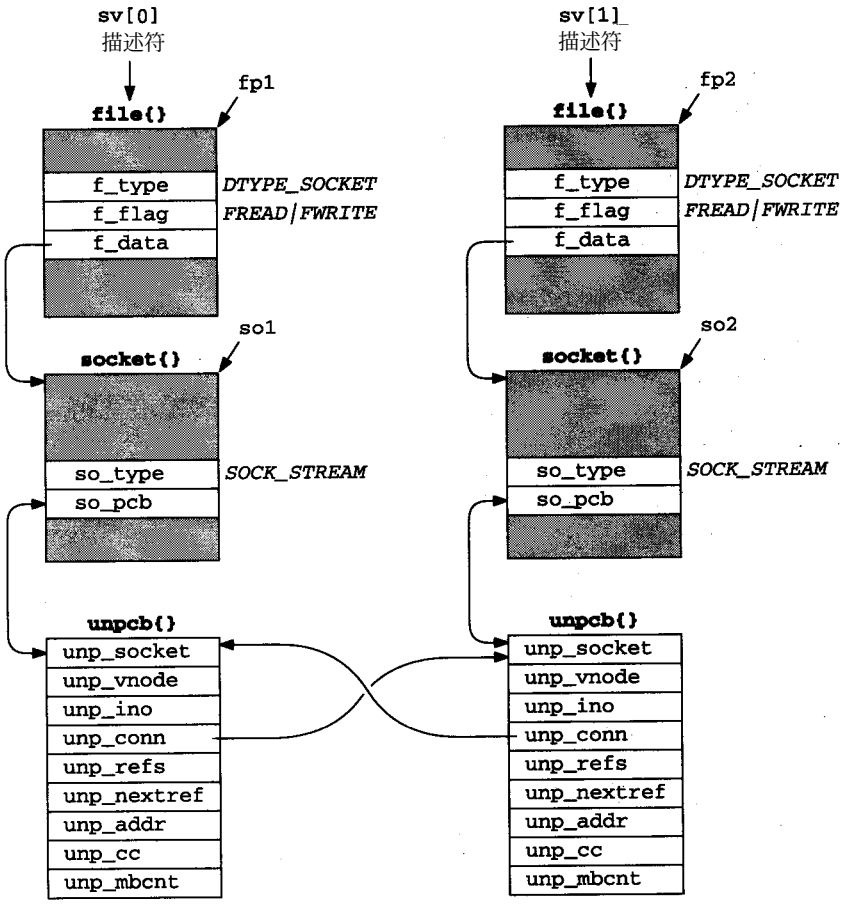


图17-28 由sockerpair 创建的两个流插口

志 (由于插口是全双工的)，文件类型设置为DTYPE_SOCKET，设置f_ops指向五个插口函数指针的数组(卷2图15-13)，设置f_data指向socket结构。第二个描述符由falloc分配，并且初始化相应的file结构。

3. 连接两个插口

262-270 soconnect2发出PRU_CONNECT2请求，这个请求仅在Unix域中得到支持。如果系统调用正在创建流插口，立即从soconnect2中返回。此时的结构如图17-28所示。

如果创建两个数据报插口，就需要调用soconnect2两次，每一次调用连接一个方向。两次调用以后，我们就有了图17-29中的结构。

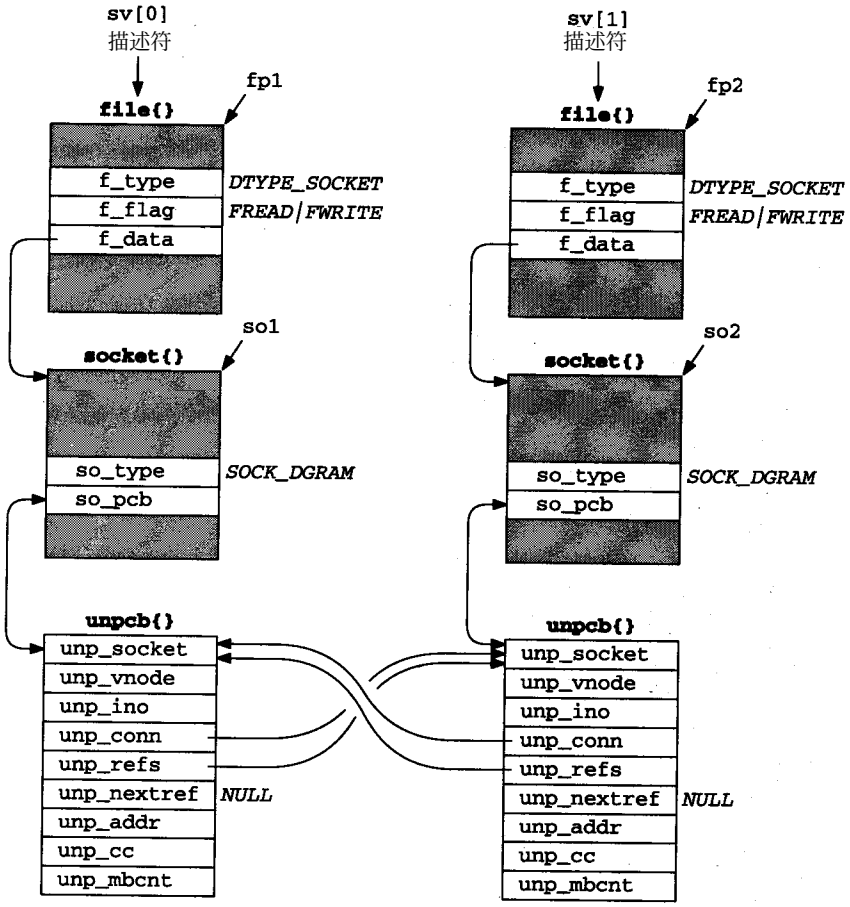


图17-29 由sockerpair 创建的两个数据报插口

4. 将两个描述符复制给进程

271-274 copyout将两个描述符复制给进程。

带有注释ＸＸＸ ？？的两个表达式第一次出现在 4.3BSD Reno版本里。因为copyout把两个描述符复制给进程，所以不需要这两个表达式。我们将看到 pipe系统调用通过设置retval[0]和retval[1]返回两个描述符，其中retval是系统调用的第三个参数。内核中处理系统调用的汇编子程序总是将两个整数 retval[0]和retval[1]放在机器的寄存器里作为任何系统调用的返回值。但是在用户进程中，

激活系统调用的汇编子程序必须查看这些寄存器并且返回进程希望得到的值。 C函数库中的pipe函数实际上是这样做的，但是socketpair函数并不这么做。

5. soconnect2函数

图17-30中的函数发出PRU_CONNECT2请求，该函数仅在socketpair系统调用中被调用。

uipc_socket.c

```
225 soconnect2(so1, so2)
226 struct socket *so1;
227 struct socket *so2;
228 {
229     int     s = splnet();
230     int     error;

231     error = (*so1->so_proto->pr_usrreq) (so1, PRU_CONNECT2,
232                 (struct mbuf *) 0, (struct mbuf *) so2, (struct mbuf *) 0);
233     splx(s);
234     return (error);
235 }
```

uipc_socket.c

图17-30 soconnect2 函数

17.13 pipe系统调用

图17-31中的pipe系统调用与socketpair系统调用几乎相同。

uipc_syscalls.c

```
645 pipe(p, uap, retval)
646 struct proc *p;
647 struct pipe_args *uap;
648 int     retval[];
649 {
650     struct filedesc *fdp = p->p_fd;
651     struct file *rf, *wf;
652     struct socket *rso, *wso;
653     int     fd, error;

654     if (error = socreate(AF_UNIX, &rso, SOCK_STREAM, 0))
655         return (error);
656     if (error = socreate(AF_UNIX, &wso, SOCK_STREAM, 0))
657         goto free1;
658     if (error = falloc(p, &rf, &fd))
659         goto free2;
660     retval[0] = fd;
661     rf->f_flag = FREAD;
662     rf->f_type = DTYPE_SOCKET;
663     rf->f_ops = &socketops;
664     rf->f_data = (caddr_t) rso;
665     if (error = falloc(p, &wf, &fd))
666         goto free3;
667     wf->f_flag = FWRITE;
668     wf->f_type = DTYPE_SOCKET;
669     wf->f_ops = &socketops;
670     wf->f_data = (caddr_t) wso;
671     retval[1] = fd;
672     if (error = unp_connect2(wso, rso))
673         goto free4;
```

图17-31 pipe 系统调用

```
674     return (0);
675   free4:
676     ffree(wf);
677     fdp->fd_ofiles[retval[1]] = 0;
678   free3:
679     ffree(rf);
680     fdp->fd_ofiles[retval[0]] = 0;
681   free2:
682     (void) soclose(wso);
683   free1:
684     (void) soclose(rso);
685     return (error);
686 }
```

uipc_syscalls.c

图17-31 (续)

654-686 调用socreate创建两个Unix域流插口，pipe系统调用与socketpair系统调用的唯一差别就是pipe把两个描述符中的第一个设置成只读 (read-only)，把第二个设置成只写 (write_only)；两个描述符由 retval参数返回，而不是通过 copyout；pipe直接调用unp_connect2，而不是通过soconnect2函数。

Unix的一些版本，特别是SVR4，创建的管道两端均可进行读写。

17.14 PRU_ACCEPT请求

对于一个流插口，接受一个新的连接所需的大部分处理工作由其他内核函数完成：sonewconn创建新的socket结构，并发出PRU_ATTACH请求，accept系统调用将插口从已完成连接队列中删除并调用 soaccept。soaccept(卷2)仅发出PRU_ACCEPT请求，用于Unix域的PRU_ACCEPT请求。如图17-33所示。

返回客户进程的路径名

94-108 如果客户进程调用bind，并且同客户进程的连接仍然存在，那么这个请求把含有客户进程路径名的 sockaddr_un复制到由 nam参数指向的 mbuf。否则，返回空路径名 (sun_nonname)。

```
91      case PRU_DISCONNECT:
92          unp_disconnect(unp);
93          break;
```

uipc_usrreq.c

图17-32 PRU_DISCONNECT 请求

```
94      case PRU_ACCEPT:
95          /*
96           * Pass back name of connected socket,
97           * if it was bound and we are still connected
98           * (our peer may have closed already!).
99           */
100         if (unp->unp_conn && unp->unp_conn->unp_addr) {
101             nam->m_len = unp->unp_conn->unp_addr->m_len;
102             bcopy(mtod(unp->unp_conn->unp_addr, caddr_t),
```

uipc_usrreq.c

图17-33 PRU_ACCEPT 请求

```
103                     mtod(nam, caddr_t), (unsigned) nam->m_len);
104             } else {
105                 nam->m_len = sizeof(sun_noname);
106                 *(mtod(nam, struct sockaddr *)) = sun_noname;
107             }
108             break;
```
uipc_usrreq.c

图17-33 (续)

17.15 PRU_DISCONNECT请求和unp_disconnect函数

如果插口已建连，close系统调用就发出PRU_DISCONNECT请求，如图17-32所示。

91-93 p_disconnect函数完成所有的断连工作，如图17-34所示。

uipc_usrreq.c

```
453 void
454 unp_disconnect(unp)
455 struct unpcb *unp;
456 {
457     struct unpcb *unp2 = unp->unp_conn;
458     if (unp2 == 0)
459         return;
460     unp->unp_conn = 0;
461     switch (unp->unp_socket->so_type) {
462     case SOCK_DGRAM:
463         if (unp2->unp_refs == unp)
464             unp2->unp_refs = unp->unp_nextref;
465         else {
466             unp2 = unp2->unp_refs;
467             for (;;) {
468                 if (unp2 == 0)
469                     panic("unp_disconnect");
470                 if (unp2->unp_nextref == unp)
471                     break;
472                 unp2 = unp2->unp_nextref;
473             }
474             unp2->unp_nextref = unp->unp_nextref;
475         }
476         unp->unp_nextref = 0;
477         unp->unp_socket->so_state &= ~SS_ISCONNECTED;
478         break;
479     case SOCK_STREAM:
480         soisdisconnected(unp->unp_socket);
481         unp2->unp_conn = 0;
482         soisdisconnected(unp2->unp_socket);
483         break;
484     }
485 }
```
uipc_usrreq.c

图17-34 unp_disconnect 函数

1. 检查插口是否有连接

458-460 如果插口没有连接到其他插口，则函数立即返回；否则就将 unp_conn置为0。表明这个插口没有连接到其他插口。

2. 将关闭的数据报PCB从链表中删除

462-478 这部分代码把关闭插口的PCB从已连接数据报PCB的链表中删除。例如，如果我们从图17-25开始，然后关闭最左边的插口，就得到图 17-35中的数据结构。由于 `unp2->unp_refs`等于unp(被关闭的PCB是链表的头)，所以被关闭的PCB的`unp_nextref`指针成为新的链表头。

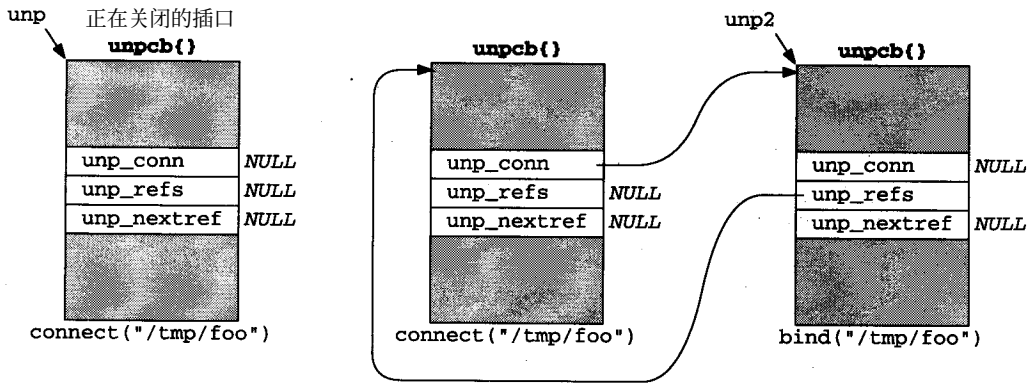


图17-35 最左边插口关闭后图17-25中的链表所发生的变化

如果我们再从图17-25开始，关闭中间的插口，就得到图 17-36中的数据结构。这一次被关闭插口的PCB就不是链表的头。unp2从链表的头开始查看被关闭的 PCB之前的PCB。删除关闭的PCB之后，unp2就指向图17-36中最左边的PCB。然后将关闭PCB的`unp_nextref`指针赋给链表(unp)上前一个PCB的`unp_nextref`。

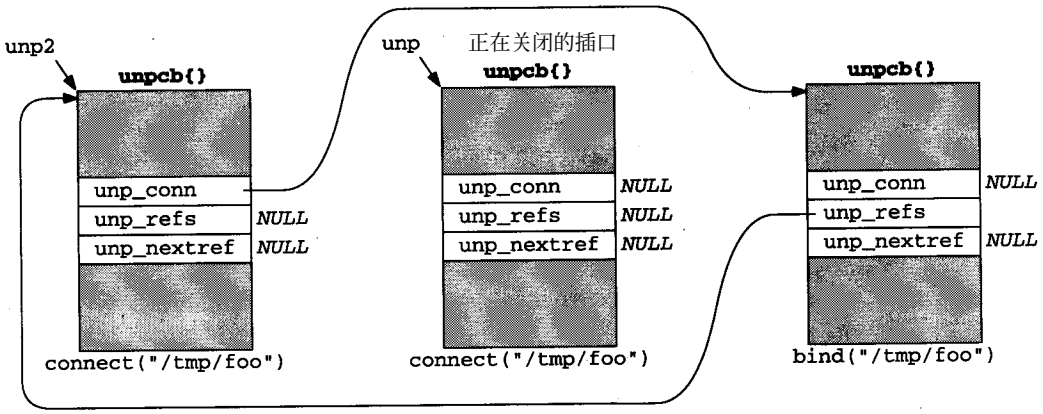


图17-36 中间插口关闭后图17-25中的链表所发生的变化

3. 完成流插口的断连

479-483 由于一个Unix域流插口只能同一个流插口建连，因而不涉及到链表，断开连接就比较简单。将连接对方的 `unp_conn`指针置为 0，并且对客户进程和服务器进程均调用`soisdisconnected`。

17.16 PRU_SHUTDOWN请求和unp_shutdown函数

当进程调用shutdown禁止任何进一步输出时，发出PRU_SHUTDOWN请求，如图17-37所示。

uipc_usrreq.c

```
109     case PRU_SHUTDOWN:
110         socantsendmore(so);
111         unp_shutdown(unp);
112         break;
```

uipc_usrreq.c

图17-37 PRU_SHUTDOWN 请求

109-112 socantsendmore设置插口标志禁止任何进一步的输出，然后调用图 17-38中的unp_shutdown函数。

uipc_usrreq.c

```
494 void
495 unp_shutdown(unp)
496 struct unpcb *unp;
497 {
498     struct socket *so;

499     if (unp->unp_socket->so_type == SOCK_STREAM && unp->unp_conn &&
500         (so = unp->unp_conn->unp_socket))
501         socantrcvmore(so);
502 }
```

uipc_usrreq.c

图17-38 unp_shutdown 函数

如果是流插口通知对等插口

499-502 对于数据报插口不需要再做什么。但是，如果这个插口是流插口，并且还与另一个插口相连，且对等端插口还有一个socket结构，则对对等端插口调用socantrcvmore。

17.17 PRU_ABORT请求和unp_drop函数

如果插口是一个监听插口，并且未完成的连接依然在队列中，那么 soclose就发出PRU_ABORT请求，如图17-39所示。soclose对在未完成连接队列和已完成连接队列中的每一个插口都发出这个请求(卷2图15-39)。

uipc_usrreq.c

```
209     case PRU_ABORT:
210         unp_drop(unp, ECONNABORTED);
211         break;
```

uipc_usrreq.c

图17-39 PRU_ABORT 请求

209-211 图17-40中的unp_drop函数产生一个ECONNABORTED错误，我们在图17-14中看到，unp_detach也调用带有参数ECONNRESET的unp_drop函数。

uipc_usrreq.c

```
503 void
504 unp_drop(unp, errno)
505 struct unpcb *unp;
506 int     errno;
507 {
508     struct socket *so = unp->unp_socket;

509     so->so_error = errno;
```

图17-40 unp_drop 函数

```
510     unp_disconnect(unp);
511     if (so->so_head) {
512         so->so_pcb = (caddr_t) 0;
513         m_freem(unp->unp_addr);
514         (void) m_free(dtom(unp));
515         sofree(so);
516     }
517 }
```

uipc_usrreq.c

图17-40 (续)

1. 保存错误，断开插口连接

509-510 设置插口的so_error值，并且如果插口上有连接，就调用unp_disconnect。

2. 如果插口在监听服务器的队列上，就删除数据结构

511-516 如果插口的so_head指针非空，那么插口当前不是在监听插口的未完成连接队列上，就是在监听插口的已完成连接队列上。从 socket到unpcb的指针都置为0，调用m_freem释放包含绑定到监听插口的路径名的mbuf(回想图17-20)，下一次调用m_free释放unpcb结构。sofree释放socket结构。由于插口在监听服务器的任何一个队列中，所以还没有与它相对应的 file结构，因为该结构是在插口从已完成连接队列中被删除时调用accept分配的。

17.18 其他各种请求

图17-41描述了其余六个尚未讨论的请求。

uipc_usrreq.c

```
212     case PRU_SENSE:
213         ((struct stat *) m)->st_blksize = so->so_snd.sb_hiwat;
214         if (so->so_type == SOCK_STREAM && unp->unp_conn != 0) {
215             so2 = unp->unp_conn->unp_socket;
216             ((struct stat *) m)->st_blksize += so2->so_rcv.sb_cc;
217         }
218         ((struct stat *) m)->st_dev = NODEV;
219         if (unp->unp_ino == 0)
220             unp->unp_ino = unp_ino++;
221         ((struct stat *) m)->st_ino = unp->unp_ino;
222         return (0);

223     case PRU_RCVOOB:
224         return (EOPNOTSUPP);

225     case PRU_SENDOOB:
226         error = EOPNOTSUPP;
227         break;

228     case PRU_SOCKADDR:
229         if (unp->unp_addr) {
230             nam->m_len = unp->unp_addr->m_len;
231             bcopy(mtod(unp->unp_addr, caddr_t),
232                   mtod(nam, caddr_t), (unsigned) nam->m_len);
233         } else
234             nam->m_len = 0;
235         break;
```

图17-41 其他的PRU_ *xxx*请求

```
236     case PRU_PEERADDR:
237         if (unp->unp_conn && unp->unp_conn->unp_addr) {
238             nam->m_len = unp->unp_conn->unp_addr->m_len;
239             bcopy(mtod(unp->unp_conn->unp_addr, caddr_t),
240                   mtod(nam, caddr_t), (unsigned) nam->m_len);
241         } else
242             nam->m_len = 0;
243         break;

244     case PRU_SLOWTIMO:
245         break;
```

uipc_usrreq.c

图17-41 (续)

1. PRU_SENSE请求

212-217 这个请求是由fstat系统调用发出的。将插口发送缓存高水位标记的当前值赋给stat结构的st_blksize作为返回值。另外，如果这个插口是一个有连接的流插口，那么将对等端插口接收缓存中的字节数加到这个值上。当我们讨论 18.2节中的PRU_SEND请求时会看到，这两个值之和就是两个相连的流插口间的实际“管道”容量。

218 将st_dev置为NODEV(所有比特为全1的常数值，代表一个不存在的设备)。

219-221 I-node号标识文件系统中的文件。该值(stat结构的st_ino字段)是作为一个Unix域插口的i-node号返回的，它是从全局变量 unp_ino得到的一个唯一值。如果还没有为unpcb分配一个这类伪i-node号，就将unp_ino的当前值赋给该unpcb作为其i-node号，然后将unp_ino加1。之所以称这些i-node号为伪i-node号，是因为它们并不是文件系统中的实际文件。它们仅在需要时由一个全局计数器产生。如果要求将 Unix域插口绑定到文件系统中的一个路径名(不是这种情况)，PRU_SENSE请求就能使用st_dev和st_ino值来代替绑定路径名。

全局变量unp_ino的递增应当在赋值之前而不是在赋值之后完成。在内核重启后，对Unix域插口第一次调用fstat时，存储在插口unpcb中的值将为0。但是，如果对相同的插口再次调用 fstat，由于unpcb中的当前值是 0，所以将全局变量unp_ino的非0值保存在其PCB中。

2. PRU_RCVOOB和PRU_SENDOOB请求

223-227 Unix域不支持带外数据。

3. PRU_SOCKADDR请求

228-235 这个请求返回绑定到插口的协议地址 (在Unix域插口中为路径名)。如果路径名绑定到插口，unp_addr就指向包含存储路径名的 sockaddr_un的mbuf。uipc_usrreq的nam参数指向由调用者分配的、用于接收结果的 mbuf。调用m_copy产生插口地址结构的副本。如果路径名没有绑定到插口，那么将 mbuf的长度域设置为0。

4. PRU_PEERADDR请求

236-243 处理这个请求与前一个请求相似，但是期望的路径名是绑定到与发起连接的插口相连的插口的名字。如果发起连接的插口已连接到一个对等端插口，那么 unp_conn非空。

没有绑定路径名的插口对这两个请求的处理与 PRU_ACCEPT请求的处理不同(图17-33)。当没有名字存在时，getsockname和getpeername系统调用通过第三个

参数返回0。而accept函数通过第三个参数返回16，通过第二个参数返回包含在sockaddr_un中由空字节组成的路径名(sun_noname是一个通用的sockaddr结构，它的长度是16个字节)。

5. PRU_SLOWTIMO请求

244-245 由于Unix域协议不使用定时器，所以从来不会发出这个请求。

17.19 小结

我们在本章看到的Unix域协议实现简单直观。它提供了流和数据报插口，其中流协议类似于TCP，数据报协议类似于UDP。

路径名能绑定到Unix域插口。服务器进程绑定其知名的路径名，客户进程连接到这个路径名。数据报插口也可以建连，与UDP一样，多个客户进程可以连接到同一个服务器进程上。Socketpair函数也可以创建尚未命名的Unix域插口。Unix pipe系统调用能创建两个互相连接的Unix域流插口，源于伯克利系统的管道实际上就是Unix域流插口。

与Unix域插口有关的协议控制块是unpcb结构。与其他域不同的是这些PCB并不保存在一个链表中。然而，当一个Unix域插口需要与另一个Unix域插口同步时(connect或sendto)，通过内核中的路径名查找函数namei来定位目的unpcb，函数namei得到一个vnode结构，通过这个结构得到目的unpcb。

第18章 Unix域协议：I/O和描述符的传递

18.1 概述

本章继续描述上一章的Unix域协议实现。本章的第一节讲述I/O、PRU_SEND和PRU_RCVD请求，其余部分介绍描述符传递。

18.2 PRU_SEND和PRU_RCVD请求

无论什么时候，当一个进程给Unix域插口发送数据或者控制信息时都要发出PRU_SEND请求。请求的第一部分首先处理控制信息，然后处理数据报插口，如图18-1所示。

uipc_usrreq.c

```
140     case PRU_SEND:
141         if (control && (error = unp_internalize(control, p)))
142             break;
143         switch (so->so_type) {

144         case SOCK_DGRAM:{
145                 struct sockaddr *from;

146                 if (nam) {
147                     if (unp->unp_conn) {
148                         error = EISCONN;
149                         break;
150                     }
151                     error = unp_connect(so, nam, p);
152                     if (error)
153                         break;
154                 } else {
155                     if (unp->unp_conn == 0) {
156                         error = ENOTCONN;
157                         break;
158                     }
159                 }
160                 so2 = unp->unp_conn->unp_socket;
161                 if (unp->unp_addr)
162                     from = mtod(unp->unp_addr, struct sockaddr *);
163                 else
164                     from = &sun_noname;
165                 if (sbappendaddr(&so2->so_rcv, from, m, control)) {
166                     sorwakeup(so2);
167                     m = 0;
168                     control = 0;
169                 } else
170                     error = ENOBUFS;
171                 if (nam)
172                     unp_disconnect(unp);
173                 break;
174             }
```

uipc_usrreq.c

图18-1 数据报插口的PRU_sSEND 请求

1. 初始化所有控制信息

141-142 如果进程使用sendmsg发送控制信息，函数unp_internalize将嵌入的描述符转换成file指针，我们将在18.4节中描述这个函数。

2. 暂时连接一个无连接的数据报插口

146-153 如果进程传送一个带有目的地址的插口地址结构(也就是说，nam参数非空)，那么插口必须是无连接的，否则返回EISCONN错误。通过unp_connect连接无连接的插口。暂时连接一个无连接的数据报插口的代码与卷2图23-15的UDP代码相似。

154-159 如果进程没有传递一个目的地址，那么对于一个无连接的插口就返回ENOTCONN错误。

3. 传递发送者的地址

160-164 so2指向目的插口的socket结构。如果发送插口(unp)已经绑定了一个路径名，from就指向包含路径名的sockaddr_un结构；否则，from指向sun_noname，sun_noname是一个以空字节作为路径名首字符的sockaddr_un结构。

如果一个Unix域数据报的发送者没有绑定一个路径名到它的插口，数据报的接收者由于没有目的地址(例如，路径名)而不能用sendto发送应答。这就与UDP不同，当数据报第一次到达一个未绑定的数据报插口时，协议就会自动为其分配一个临时的端口号。UDP能为应用程序自动选择端口号的一个原因是这些端口号仅由UDP使用。然而，文件系统中的路径名并不是仅为Unix域插口保留。因而为一个没有绑定的Unix域插口自动选择路径名可能会在后面产生冲突。

是否需要一个应答取决于应用程序。例如，syslog函数没有绑定一个路径名到它的Unix域数据报插口，它仅发送报文到本地syslogd守护进程而不想得到一个应答。

4. 把控制、地址和数据mbuf添加到插口接收队列

165-170 sbappendaddr将控制信息(如果需要)、发送者地址和数据添加到接收插口的接收队列。如果函数调用成功，sorwakeup就要唤醒所有等待这些数据的接收者，为了防止mbuf指针m和control在函数结束时被释放，将它们全置为0(图17-10)。如果出现错误(可能因为在接收队列上没有足够空间来存放数据、地址和控制信息)，就返回ENOBUFS。

处理这种错误与UDP不同。如果在接收队列上没有足够的空间，使用Unix域数据报插口的sender就会收到从它的输出操作返回的错误。同UDP一样，如果在接口输出队列上有足够的空间，那么发送者的输出操作就会成功。如果接收UDP发现在接收插口的接收队列上没有空间，它通常发送一个ICMP端口不可达的错误给发送者，但是如果发送者没有连接到接收者，它也就不可能收到这个错误(如同卷2第600~601页描述的一样)。为什么当接收者的缓存满时Unix域发送者不阻塞，而是收到ENOBUFS错误？传统上，数据报不保证可靠的数据传输。[Rago1993]认为，在SVR4下编译内核时，是否给Unix域数据报插口提供流量控制是由厂家来决定的。

5. 暂时断开与相连插口的连接

171-172 unp_disconnect断开暂时连接的插口。

图18-2给出了对于流插口的PRU_SEND请求的处理。

uipc_usrreq.c

```
175         case SOCK_STREAM:
176 #define rcv (&so2->so_rcv)
177 #define snd (&so->so_snd)
178             if (so->so_state & SS_CANTSENDMORE) {
179                 error = EPIPE;
180                 break;
181             }
182             if (unp->unp_conn == 0)
183                 panic("uipc 3");
184             so2 = unp->unp_conn->unp_socket;
185             /*
186              * Send to paired receive port, and then reduce
187              * send buffer hiwater marks to maintain backpressure.
188              * Wake up readers.
189              */
190             if (control) {
191                 if (sbappendcontrol(rcv, m, control))
192                     control = 0;
193             } else
194                 sbappend(rcv, m);
195             snd->sb_mbmax -=
196                 rcv->sb_mbcnt - unp->unp_conn->unp_mbcnt;
197             unp->unp_conn->unp_mbcnt = rcv->sb_mbcnt;
198             snd->sb_hiwat -= rcv->sb_cc - unp->unp_conn->unp_cc;
199             unp->unp_conn->unp_cc = rcv->sb_cc;
200             sorwakeup(so2);
201             m = 0;
202 #undef snd
203 #undef rcv
204             break;

205         default:
206             panic("uipc 4");
207         }
208         break;
```

uipc_usrreq.c

图18-2 流插口的PRU_SEND 请求

6. 验证插口状态

175-183 如果插口的发送方已经关闭，就返回EPIPE。因为sosend验证需要一个连接的插口是否已建立连接，所以这个插口必须已建连，否则调用panic(卷2图16-24)。

第一次测试好像是一个早期版本中遗留下来的，sosend已经做了这个测试(卷2图16-24)。

7. 把mbuf添加到接收缓存

184-194 so2指向接收插口的socket结构。如果进程使用sendmsg传送了控制信息，那么控制mbuf和任何数据mbuf都要通过sbappendcontrol添加到接收插口的接收缓存。否则，sbappend将数据mbuf添加到接收缓存。如果sbappendcontrol失败，为了防止在函数结尾调用m_freem，将control指针设置为0(图17-10)，因为sbappendcontrol已经释放了mbuf。

8. 更新发送者和接收者的计数器(端到端的流量控制)

195-199 对于发送者要更新两个变量：sb_mbmax(缓存中所有mbuf允许的最大字节数)和

sb_hiwat(缓存中允许存放实际数据的最大字节数)，在卷2的图16-24中我们注意到，对mbuf所做的限制防止了大量小报文消耗太多的mbuf。

对于Unix域流插口，这两个限制指的是接收缓存和发送缓存中的两个计数器的和。例如，一个Unix域流插口的发送缓存和接收缓存的sb_hiwat初始值都是4096(图17-2)。如果发送者把1 024字节写到插口上，不仅接收者的sb_cc(插口缓存中的当前字节数)从0增长到1024(正如我们所希望的)，而且发送者的sb_hiwat从4096减到3072(这是我们所不希望的)。对于其他协议如TCP，如果没有显式设置插口的选项，缓存的sb_hiwat值决不会变化。sb_mbmax也是一样：当接收者的sb_mbcnt值增加时，发送者的sb_mbmax值下降。

因为发送给Unix域流插口的数据从来不会放在发送插口的发送缓存中，所以要改变发送者的缓存限制和接收者的当前计数。数据被立即加到接收插口的接收缓存中，没有必要浪费时间把数据放到发送插口的发送队列上，然后立即或晚些时候把它发送到接收队列上。如果接收缓存中没有空闲空间，发送者就要被阻塞。但是，如果sosend阻塞发送者，发送缓存中的空间大小必须反映相应接收缓存中的空间大小。代替修改发送缓存数，当发送缓存中没有数据时，很容易修改发送缓存限制来反映相应接收缓存中的空间大小。

198-199 如果我们只是检验发送者的sb_hiwat和接收者的unp_cc的操作(sb_mbmax和unp_mbcnt的操作也基本相同)，在这一点上由于数据刚被添加到接收缓存，所以 rcv->sb_cc就等于接收缓存中的字节数。unp->unp_conn->unp_cc是rcv->sb_cc的前一个值，所以它们之间的差值就是刚刚添加到接收缓存的字节数 (也就是写的字节数)。同时，将snd->sb_hiwat的值减去相同的字节数(刚写的字节数)。接收缓存中的当前字节数保存在unp->unp_conn->unp_cc中，所以下一次通过这段代码我们能计算出写了多少数据。

例如，当创建插口时，发送者的 sb_hiwat是4096，接收者的sb_cc和unp_cc都为0。如果写了1 024字节，那么发送者的sb_hiwat变为3072，接收者的sb_cc和unp_cc都是1 024。在图18-3中我们还将看到，当接收进程读这1 024个字节时，发送者的sb_hiwat增加到4096，而接收者的sb_cc和unp_cc都降为0。

9. 唤醒等待数据的所有进程

200-201 sorwakeup唤醒等待数据的任何进程，由于mbuf现在在接收队列上，所以为了防止在函数结尾调用m_freem，将m设置为0。

图18-3中I/O代码的最后部分是 PRU_RCVD请求，当从一个插口读数据并且协议设置PR_WANTRCVD标志时，soreceive发出这个请求(卷2图16-51)，图17-5中对Unix域流协议设置这个标志。这个请求的目的是当插口层把数据从一个插口的接收缓存中移走时让协议层获得控制。例如，由于插口接收缓存中现在有更多的自由空间， TCP使用这个请求来判断是否应该将新的窗口宽度发送到对端，Unix域流协议使用这个请求去更新发送者和接收者的缓存计数器。

10. 检查对等实体是否终止

121-122 如果写数据的对等实体已经结束，不需做任何工作。注意，接收者的数据并不丢弃；然而，由于发送进程关闭了它的插口，所以发送者的缓存计数器就不能更新。由于发送者不再往插口写任何数据，所以没有必要更新缓存计数器。

11. 更新缓存计数器

123-131 so2指向发送者socket结构。根据读到的数据来更新发送者的 sb_mbmax和sb_hiwat。例如，unp->unp_cc减去rcv->sb_cc就是所读到的数据字节数。

uipc_usrreq.c

```
113     case PRU_RCVD:
114         switch (so->so_type) {

115         case SOCK_DGRAM:
116             panic("uipc 1");
117             /* NOTREACHED */

118         case SOCK_STREAM:
119 #define rcv (&so->so_rcv)
120 #define snd (&so2->so_snd)
121             if (unp->unp_conn == 0)
122                 break;
123             so2 = unp->unp_conn->unp_socket;
124             /*
125              * Adjust backpressure on sender
126              * and wake up any waiting to write.
127              */
128             snd->sb_mbmax += unp->unp_mbcnt - rcv->sb_mbcnt;
129             unp->unp_mbcnt = rcv->sb_mbcnt;
130             snd->sb_hiwat += unp->unp_cc - rcv->sb_cc;
131             unp->unp_cc = rcv->sb_cc;
132             sowwakeup(so2);
133 #undef snd
134 #undef rcv
135             break;

136         default:
137             panic("uipc 2");
138         }
139         break;
```

uipc_usrreq.c

图18-3 PRU_RCVD 请求

12. 唤醒任何发送数据进程

132 当从接收队列读数据时，增加发送者的sb_hiwat。由于可能有空间，所以任何等待往插口写数据的进程都被唤醒。

18.3 描述符的传递

描述符的传递对于进程间通信来说是一项重大的技术。[Stevens 1992]的第15章在4.4BSD和SVR4下有使用这种技术的例子。虽然在这两种实现中的系统调用不同，但是那些例子提供了对应用程序屏蔽实现差异的库函数。

历史上描述符传递一直被称为访问权（access right）。描述符代表一种对底层对象执行I/O的权力(如果我们没有这个权力，内核就不会为我们打开描述符)。但是这个能力仅在打开描述符的进程环境中才有意义。例如，将描述符号，假定等于 4，从一个进程传到另一个进程，但并不传递这些权力，因为在接收进程中描述符 4也许并没有打开，并且即使已经打开了，它代表的文件也可能与发送进程中所代表的文件不相同。描述符只是一个在给定进程中才有意义的标识符。一个描述符以及与其相联系的权力从一个进程传送到另一个进程需要从内核得到额外的支持。唯一能从一个进程传到另一个进程的访问权就是描述符。

图18-4显示了涉及到将描述符从一个进程传到另一个进程的数据结构。传送过程如下：

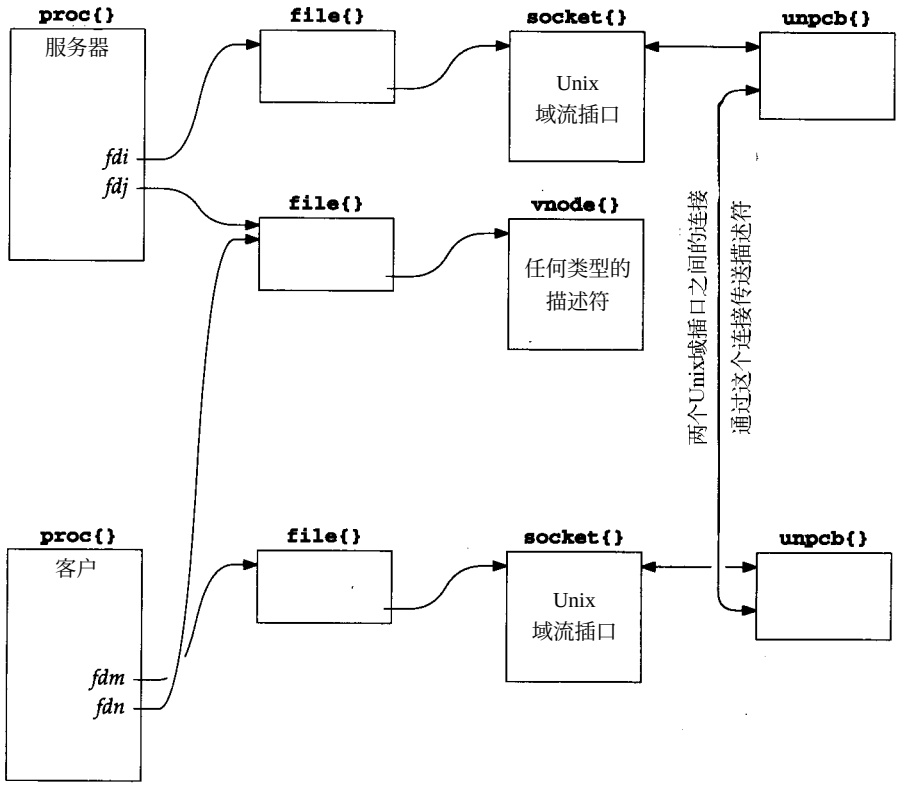


图18-4 在描述符传递中涉及到的数据结构

1) 我们假定最上面进程是一个从 Unix域流插口上接受连接的服务器进程。客户进程是最下面的进程，它创建一个Unix域流插口并与服务器进程的插口建连。客户进程用 *fdm*引用它的插口，而服务器进程用 *fdi*来引用它的插口。在这个例子中我们用的是流插口，但是我们还将看到描述符传递也能在 Unix域数据报插口间进行。我们也假定 17.10节中`accept`返回的*fdi*作为服务器进程的连接插口，为了简单起见，我们不显示服务器进程监听插口的结构。
2) 服务器进程还打开另一个文件，并用 *fdj*来访问它。通过描述符访问的文件可能是任何类型的文件：文件、设备、插口，等等，我们用 `vnode`来表示这类文件。文件的访问计数，也就是它的`file`结构的`f_count`字段，在文件第一次打开时等于 1。
3) 服务器在*fdi*上调用`sendmsg`发送包含一个类型值`SCM_RIGHTS`和 *fdj* 值的控制信息。从而将描述符传送给接收者，即客户进程中的 *fdm*。将与 *fdj* 相联系的`file`结构中的引用数增加到2。
4) 客户进程在*fdm*上调用带有控制信息缓存的 `recvmsg`，返回的控制信息有一个类型值`SCM_RIGHTS`和 *fdn*值，在客户进程中*fdn*是最低的、尚未使用的描述符。
5) 在服务器进程中，当`sendmsg`返回后，服务器通常会关闭刚才传送的描述符(*fdj*)。这会导致引用计数减到1。我们说在`sendmsg`和`recvmsg`之间描述符在“传送中”(*in flight*)。

三个计数器由内核负责维护，描述符传递中要用到它们。

1) f_count是file结构的一个字段，用来记录该结构的引用次数。当多个描述符共享相同的file结构时，这个字段等于描述符数。例如当一个进程打开一个文件时，该文件的f_count为1。如果进程接着调用fork，由于file结构在父进程和子进程间共享，所以f_count的值变为2，并且父进程和子进程都有一个描述符指向相同的文件结构。当一个描述符被关闭时，f_count值以1递减，如果值减到0，相应的文件或插口被关闭，并且file结构能重新使用。
2) f_msgcount也是file结构的一个字段，但是它仅在传送描述符时等于非0。当描述符由sendmsg传送时，f_msgcount以1递增。当recvmsg接收到描述符时，f_msgcount以1递减。f_msgcount值是这个file结构的引用数，file结构由插口接收队列中的描述符保持着(即目前是在传送中)。
3) unp_rights是一个全局变量，用来记录当前正被传送的描述符个数，也就是当前插口接收队列中的描述符总数。

对于一个已打开，但还没有被传送的描述符，f_count的值大于0，f_msgcount的值等于0。图18-5显示了当一个描述符传送时三个变量的值，我们假定当前内核没有传送其他的描述符。

	f_count	f_msgcount	unp_rights
发送方执行open后	1	0	0
发送方执行sendmsg后	2	1	1
在接收方的队列上	2	1	1
接收方执行recvmsg后	2	0	0
发送方执行close后	1	0	0

图18-5 描述符传送过程中内核变量的值

在这个图中我们假定，接收者的recvmsg返回后发送者关闭描述符。但是在接收者调用recvmsg之前，允许发送者在描述符传递过程中关闭它，图18-6表示了这种情况发生时三个变量的值。

	f_count	f_msgcount	unp_rights
发送方执行open后	1	0	0
发送方执行sendmsg后	2	1	1
在接收方的队列上	2	1	1
发送方执行close后	1	1	1
在接收方的队列上	1	1	1
接收方执行recvmsg后	1	0	0

图18-6 描述符传送过程中内核变量的值

无论发送者在接收者调用recvmsg之前或之后关闭描述符，最终结果都是一样的。我们从上面两个图中也能看到，sendmsg增加所有的三个计数器，而recvmsg只减少表中的最后两个计数器。

用来传送描述符的内核代码从概念上看是比较简单的。将传送的描述符转换成相应的file指针并传送到Unix域插口的另一端。在接收进程中，接收者把file指针转换为最低的、没有使用的描述符。然而在处理可能的错误时就有问题了，例如，当一个描述符在它的接收

队列上时，接收进程就能关闭它的Unix域插口。

将一个描述符转换成相应的file指针叫做内部化(internalizing)，在接收进程中，随后的file指针转换成最低的。没有使用的描述符叫做外部化(externalizing)。如果进程传送控制信息，图18-1中的PRU_SEND请求将调用unp_internalize函数。如果进程正在读MT_CONTROL类型的一个mbuf，soreceive就调用unp_externalize函数(卷2图16-44)。

图18-7显示了被进程传送到sendmsg的控制信息的定义，这里控制信息用于传送描述符。当接收到一个描述符时，recvmsg填充相同类型的一个结构。

socket.h

```
251 struct cmsghdr {
252     u_int   cmsg_len;           /* data byte count, including hdr */
253     int     cmsg_level;         /* originating protocol */
254     int     cmsg_type;          /* protocol-specific type */
255 /* followed by  u_char  cmsg_data[]; */
256 };
```

socket.h

图18-7 cmsghdr 结构

例如，如果进程发送两个描述符，它们的值分别是3和7，图18-8给出了控制信息的格式。我们还给出了msghdr结构中描述控制信息的两个字段。

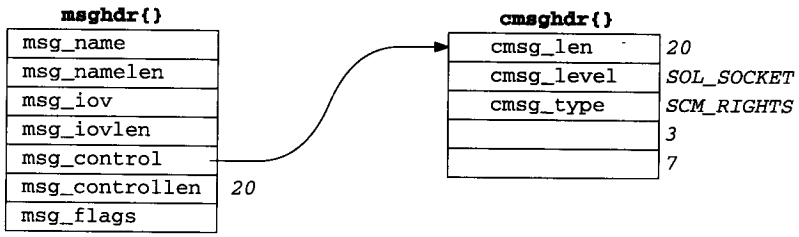


图18-8 传送两个描述符的控制信息的例子

通常一个进程使用一个sendmsg能发送任意个描述符，但是传送描述符的应用程序典型情况下只传送一个描述符。有一个内部约束限制着控制信息总的大小必须适合一个 mbuf(由sockargs函数强加的，这个sockargs函数又是被sendit函数调用的，分别见卷2图15-20和图16-21)，这样就限制了任何进程最多只能传送24个描述符。

在4.3BSD Reno之前，msghdr结构的msg_control和msg_controllen字段分别为msg_accrights和msg_accrightslen。

明显冗余的cmsg_len字段总是等于msg_controllen，这其中的原因是允许多条控制信息出现在同一个控制缓存中，但是我们将看到源代码不支持这种情况，而是要求每个控制缓存仅有一个控制报文。

对于一个UDP数据报，Internet域中支持的唯一控制信息是返回目的IP地址(卷2图23-25)。对于各种特定OSI用途的OSI协议支持四种不同类型的控制信息。

图18-9总结了在发送和接收描述符过程中调用的函数，带阴影的函数在本卷中讲述，其余的函数见卷2。

图18-10总结了unp_internalize和unp_externalize对用户控制缓存和内核mbuf中的描述符和file指针的各种操作。

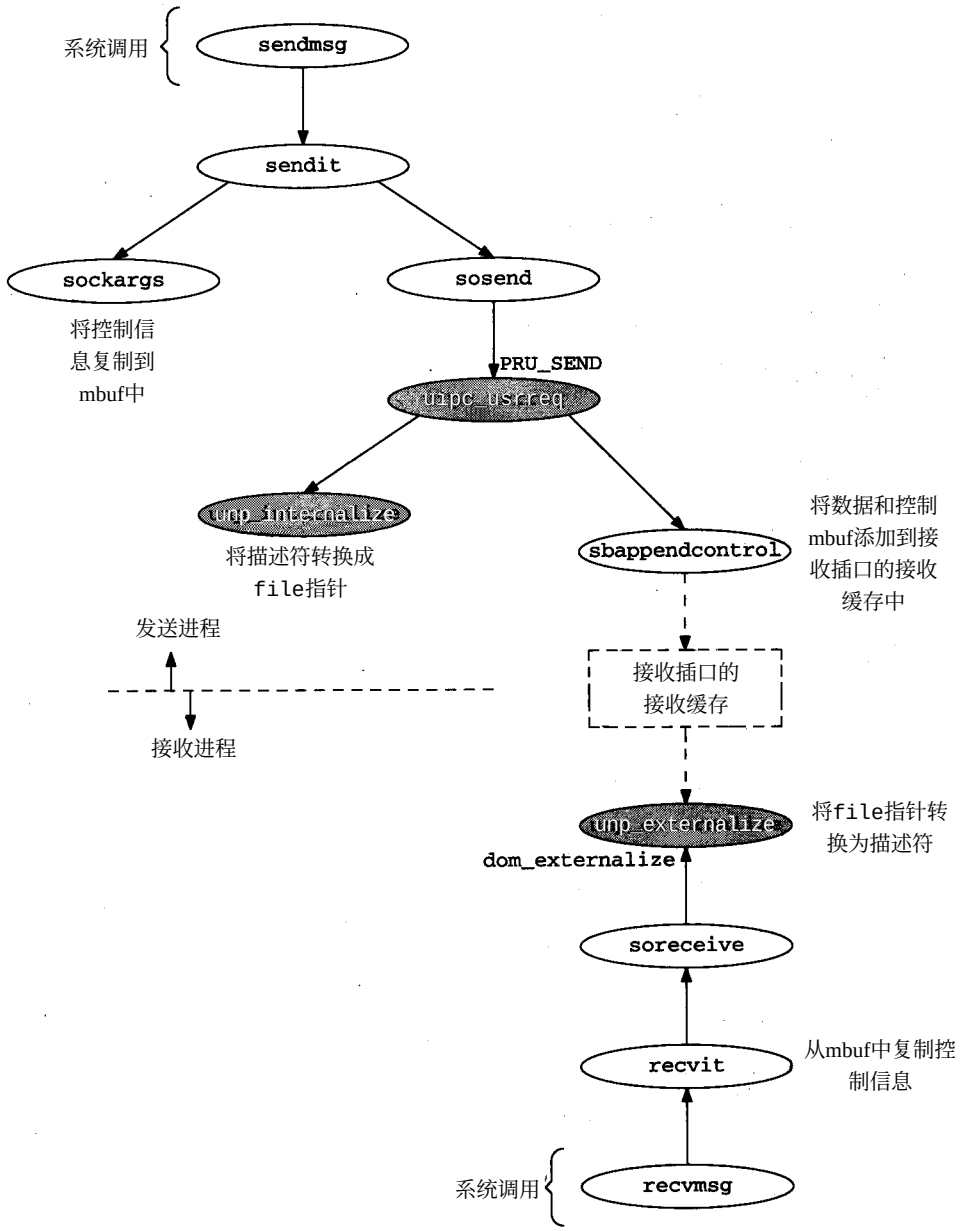


图18-9 在传送描述符过程中涉及到的函数

18.4 unp_internalize函数

图18-11描述了unp_internalize函数。正如我们在图 18-1中看到的一样，当发出PRU_SEND请求并且进程正传送描述符时，uipc_usrreq调用这个函数。

1. 验证cmsghdr字段

564-566 用户的cmsghdr结构必须指定类型SCM_RIGHTS和级别SOL_SOCKET，并且它的长度字段必须等于mbuf中的数据量(这是msghdr结构中msg_controllen字段的一个副本，msghdr结构由进程传送到sendmag)。

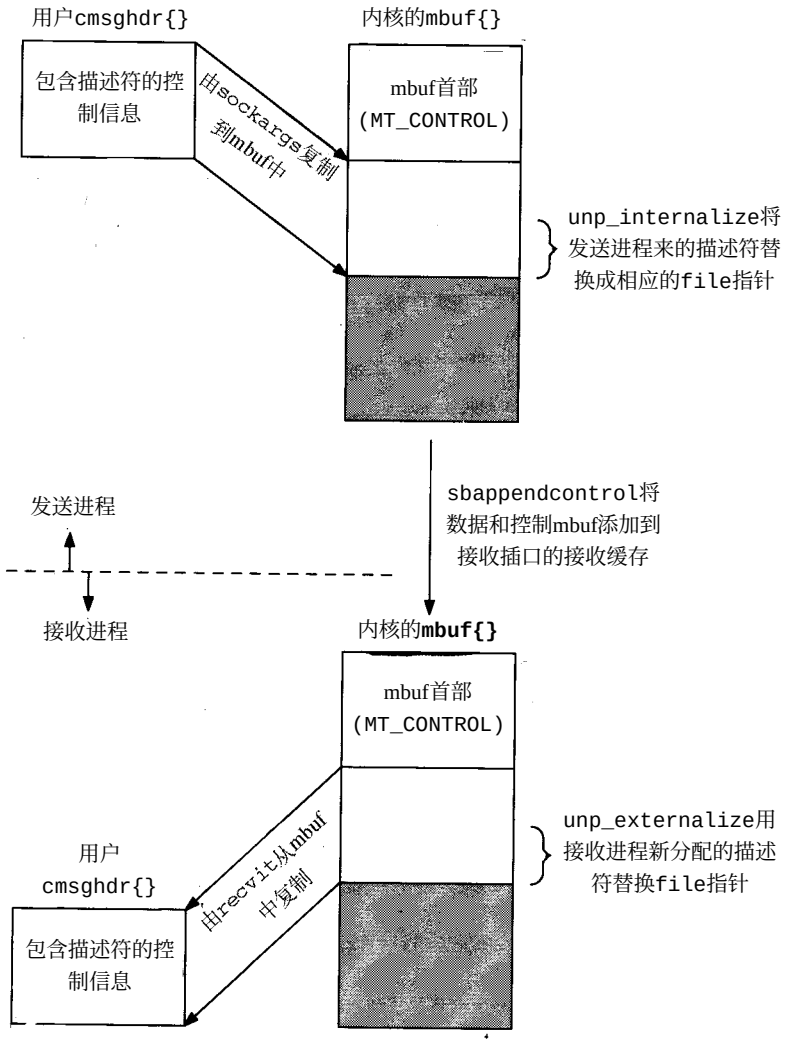


图18-10 由unp_internalize 和unp_externalize 执行的操作

2. 验证传送描述符的有效性

567-574 oldfds设置为被传送的描述符数，rp指向第一个描述符。对于每一个被传送的描述符，for循环验证这个描述符不会比当前被进程使用的最大描述符还大，以及指针非空(即描述符已打开)。

3. 用file指针替换描述符

575-578 将rp重新设置为指向第一个描述符，for循环用引用的file指针fp替换每一个描述符。

4. 增加三个计数器

579-581 file结构的f_count和f_msgcount元素递增，前者在每一次描述符关闭时递减，而后者由unp_externalize递减。另外，对于每一个由unp_internalize传送的描述符来说，全局变量unp_rights递增。我们将看到，对于每一个由unp_externalize接受的描述符，unp_rights将递减。任何时候它的值都是当前内核中正在传送的描述符数。

uipc_usrreq.c

```
553 int
554 unp_internalize(control, p)
555 struct mbuf *control;
556 struct proc *p;
557 {
558     struct filedesc *fdp = p->p_fd;
559     struct cmsghdr *cm = mtod(control, struct cmsghdr *);
560     struct file **rp;
561     struct file *fp;
562     int     i, fd;
563     int     oldfds;
564     if (cm->cmsg_type != SCM_RIGHTS || cm->cmsg_level != SOL_SOCKET ||
565         cm->cmsg_len != control->m_len)
566         return (EINVAL);
567     oldfds = (cm->cmsg_len - sizeof(*cm)) / sizeof(int);
568     rp = (struct file **) (cm + 1);
569     for (i = 0; i < oldfds; i++) {
570         fd = *(int *) rp++;
571         if ((unsigned) fd >= fdp->fd_nfiles ||
572             fdp->fd_ofiles[fd] == NULL)
573             return (EBADF);
574     }
575     rp = (struct file **) (cm + 1);
576     for (i = 0; i < oldfds; i++) {
577         fp = fdp->fd_ofiles[*(int *) rp];
578         *rp++ = fp;
579         fp->f_count++;
580         fp->f_msgcount++;
581         unp_rights++;
582     }
583     return (0);
584 }
```

uipc_usrreq.c

图18-11 unp_internalize 函数

我们在图17-14中看到，当任何Unix域插口关闭，并且计数器非0时，调用无用单元收集函数unp_gc，以免关闭的插口在它的接收队列上包含任何正在传送的描述符。

18.5 unp_externalize函数

图18-12表示了unp_externalize函数，当一个类型为MT_CONTROL的mbuf在插口的接收队列上，并且进程正准备接收控制信息时，soreceive像调用dom_externalize函数一样调用unp_externalize(卷2图16-44)。

1. 验证接收进程是否有足够的可用描述符

532-541 newfds是外部化的mbuf中file指针的数目。fdavail是检验进程是否有足够可用描述符的一个内核函数。如果没有足够的可用描述符，那么对于每一个描述符调用unp_discard(在下一节描述)，并且返回EMSGSIZE给进程。

2. 把file指针转换成描述符

542-546 对于进程中每一个传送的file指针，最小的没有使用的描述符由fdalloc来分配。fdalloc的第二个参数0告诉它不需要分配一个file结构，因为此时需要的只是一个描述符。fdalloc通过f返回描述符。进程中的描述符指向file指针。

3. 递减两个计数器

547-548 对于每一个传送的描述符，两个计数器f_msgcount和unp_rights都要递减。

4. 用描述符替换file指针

549 新分配的描述符替换mbuf中的file指针，这是作为控制信息返回到进程中的值。

如果由进程传送到recvmsg的控制缓存不够，接收传送的描述符怎么办？unp_externalize仍然分配进程中需要的描述符数，描述符全部指向正确的file结构，但是recvit(卷2的图16-44)仅仅返回与进程分配的缓存相适应的控制信息。如果导致控制信息的不完整截断，那么就要置上msg_flags字段中的MSG_CTRUNC标志，进程通过测试这个标志来判断recvmsg返回的控制信息的完整性。

uipc_usrreq.c

```
523 int
524 unp_externalize(rights)
525 struct mbuf *rights;
526 {
527     struct proc *p = curproc;    /* XXX */
528     int     i;
529     struct cmsghdr *cm = mtod(rights, struct cmsghdr *);
530     struct file **rp = (struct file **) (cm + 1);
531     struct file *fp;
532     int     newfds = (cm->cmsg_len - sizeof(*cm)) / sizeof(int);
533     int     f;

534     if (!fdavail(p, newfds)) {
535         for (i = 0; i < newfds; i++) {
536             fp = *rp;
537             unp_discard(fp);
538             *rp++ = 0;
539         }
540         return (EMSGSIZE);
541     }
542     for (i = 0; i < newfds; i++) {
543         if (fdalloc(p, 0, &f))
544             panic("unp_externalize");
545         fp = *rp;
546         p->p_fd->fd_ofiles[f] = fp;
547         fp->f_msgcount--;
548         unp_rights--;
549         *(int *) rp++ = f;
550     }
551     return (0);
552 }
```

uipc_usrreq.c

图18-12 unp_externalize 函数

18.6 unp_discard函数

当判断出接收进程没有足够的可用描述符时，在图 18-12中对于每一个传送的描述符调用图18-13中的unp_discard。

1. 递减两个计数器

730-731 f_msgcount和unp_rights两个计数器全都递减。

uipc_usrreq.c

```
726 void
727 unp_discard(fp)
728 struct file *fp;
729 {
730     fp->f_msgcount--;
731     unp_rights--;
732     (void) closef(fp, (struct proc *) NULL);
733 }
```

uipc_usrreq.c

图18-13 unp_discard 函数

2. 调用closef

732 closef关闭file，如果f_count现在是0，closef就减小f_count，并且调用描述符的fo_close函数(卷2的图15-38)。

18.7 unp_dispose函数

回想图17-14中，如果全局变量unp_rights非0(即有描述符在传送中)，那么当关闭一个Unix域插口时，unp_detach函数就要调用sorflush。如果有定义，并且协议设置了PR_RIGHTS标志，sorflush(卷2图15-37)执行的最后操作之一就是调用域的dom_dispose函数。因为将要刷新(释放)的mbuf也许包含正在传送中的描述符，所以需要执行这个调用。由于file结构中的两个计数器f_count和f_msgcount以及全局变量unp_rights都要由unp_internalize来递增，对于已传送但没有被接收的描述符，这些计数器全都必须要调整。

Unix域的dom_dispose函数就是unp_dispose(图17-4)，如图18-14所示。

uipc_usrreq.c

```
682 void
683 unp_dispose(m)
684 struct mbuf *m;
685 {
686     if (m)
687         unp_scan(m, unp_discard);
688 }
```

uipc_usrreq.c

图18-14 unp_dispose 函数

调用unp_scan

686-687 unp_scan完成的所有工作我们在下一节描述。该调用的第二个参数是指向函数unp_discard的一个指针，正如我们在上一节看到的一样，unp_discard删除在插口接收队列上unp_scan发现的控制缓存中的任何描述符。

18.8 unp_scan函数

从unp_dispose调用unp_scan函数，其第二个参数为unp_discard，并且这个函数在后面的unp_gc中也会被调用，其第二个参数为unp_mark。我们在图18-15中给出了unp_scan。

uipc_usrreq.c

```
689 void
690 unp_scan(m0, op)
691 struct mbuf *m0;
692 void    (*op) (struct file *);
693 {
694     struct mbuf *m;
695     struct file **rp;
696     struct cmsghdr *cm;
697     int     i;
698     int     qfds;

699     while (m0) {
700         for (m = m0; m; m = m->m_next)
701             if (m->m_type == MT_CONTROL &&
702                 m->m_len >= sizeof(*cm)) {
703                 cm = mtod(m, struct cmsghdr *);
704                 if (cm->cmsg_level != SOL_SOCKET ||
705                     cm->cmsg_type != SCM_RIGHTS)
706                     continue;
707                 qfds = (cm->cmsg_len - sizeof *cm)
708                     / sizeof(struct file *);
709                 rp = (struct file **) (cm + 1);
710                 for (i = 0; i < qfds; i++)
711                     (*op) (*rp++);
712                 break;          /* XXX, but saves time */
713             }
714         m0 = m0->m_nextpkt;
715     }
716 }
```

uipc_usrreq.c

图18-15 unp_scan 函数

1. 查找控制mbuf

699-706 这个函数检查插口接收队列上(m0参数)所有的分组，并且扫描每一个分组的mbuf链去查找一个类型为 MT_CONTROL的mbuf。当发现一个控制报文时，如果层次是SOL_SOCKET，类型是SCM_RIGHTS，那么mbuf包含没有被接收的传送中的描述符。

2. 释放保持的file引用

707-716 qfds是控制信息中 file表指针的数量，对每一个 file指针调用 op函数(unp_discard或unp_mark)。op函数的参数是控制信息中的 file指针。当处理完该控制mbuf时，执行break，跳出循环，处理接收缓存中的下一个分组。

712行的注释XXX表示：因为break假定每个mbuf链仅有一个控制mbuf，这实际上是对的。

18.9 unp_gc函数

我们已经看到用来处理传送中的描述符的无用单元收集函数的一种形式：在unp_detach中，无论什么时候关闭一个Unix域插口，并且描述符在传送中，sorflush就释放任何传送中的、包含在关闭插口接收队列上的描述符。然而，在 Unix域插口间传送的描述符也有可能“丢失”，在三种情况下这种事情可能发生。

1) 当描述符被传送时，一个类型为 MT_CONTROL的mbuf由sbappendcontrol(图18-2)放在插口接收队列上。但是，如果接收进程调用 recvmsg却没有说明想接收控制信息，

或者调用一个不能接收控制信息的其他输入函数，soreceive就调用MFREE，从插口接收缓存中删除类型为MT_CONTROL的mbuf，并释放它(卷2图15-44)。但是，当由这个mbuf引用的file结构被发送者关闭时，它的f_count和f_msgcount将全为1(回想图18-6)，全局变量unp_rights仍然表明这个描述符在传送中。这是一个没有被其他任何描述符引用的file结构，并且将来也不会被一个描述符引用，但是仍在内核的活动file结构链表上。

[Leffler et al.1989]的第305页讲到，问题是在报文被传送到插口层等待传送之后，内核不允许协议再访问该报文；他们还后见之明地讲到，当一个类型为MT_CONTROL的mbuf被释放时，这个问题应当由触发的每个域的处理函数来处理。

2) 当描述符被传送，但是接收插口没有空间存放这个报文时，不需要任何说明就丢弃这个传送中的描述符。这种情况在一个 Unix域流插口中应当不会发生，因为在 18.2节中我们看到，发送者的高水位标记反映了接收者缓存中的空间大小，使得在接收缓存有了空间之前发送者的高水位标记一直阻塞发送者。但是在一个 Unix域数据报插口中可能会失败，如果接收缓存没有足够的空间，sbappendaddr(在图18-1中调用)返回0，error设置为ENOBUFS，在标号release处的代码会删除包含控制报文的 mbuf，这就如同在前一个例子中一样导致相同的情况：一个没有被任何描述符引用的file结构，并且将来也不会被一个描述符引用。
3) 当一个Unix域插口*fdi*在另一个Unix域插口*fdj*上传送时，*fdj*也在*fdi*上传送。如果两个Unix域插口在没有接收到传送的描述符时关闭，这些描述符就有可能丢失。我们将看到4.4BSD直接处理了这个问题(图18-18)。

开始两种情况的关键事实是，“丢失的”file结构的f_count等于它的f_msgcount(即对这个描述符的引用是在控制报文中)，并且file结构当前没有被内核中所有 Unix域插口的接收队列中任何控制报文引用。如果 file结构的f_count超过了它的f_msgount，那么差别就是在引用结构的进程中描述符数，所以结构没有丢失 (一个file的f_count值必须不能小于它的f_msgcount值，否则某些事情就要受到破坏)。如果f_count等于f_msgcount，但是file结构被Unix域插口上的控制报文引用，由于一些进程仍然能从该插口接收描述符，因而不会出现问题。

无用单元收集函数unp_gc找到这些丢失的file结构，并回收它们。调用closef来回收file结构，如图18-13所示，因为closef返回一个无用的file结构给内核的空闲缓存池。注意这个函数仅在有传送中描述符时才调用，这就是说，仅当 unp_rights非0(图17-14)和一些Unix域插口关闭时才调用这个函数。因而由于这个函数似乎涉及过多的开销，它应当很少调用。

unp_gc使用标记-回收(mark-and-sweep)算法去执行无用单元收集，这个函数的前一部分，即标记阶段，检查内核中的每一个 file结构，并把那些正在使用的置上标志： file结构要么被进程中的描述符引用，要么被 Unix域插口的接收队列上的控制报文引用 (这就是说，file结构对应一个当前在传送中的描述符)。函数的后一部分，即回收阶段，回收所有尚未置上标志的file结构，因为这些file结构不在使用中。

图18-16给出了unp_gc的前半部分。

1. 防止函数被递归调用

594-596 全局变量unp_gcing防止函数被递归调用，因为unp_gc能调用sorflush，而

sorflush能调用unp_dispose，unp_dispose能调用unp_discard，unp_discard能调用closef，closef能调用unp_detach，unp_detach又能再次调用unp_gc。

2. 清除FMARK和FDEFER标志

598-599 第一个循环检验内核里面的所有file结构，并且清除FMARK和FDEFER标志。

3. 循环到unp_defer等于零

600-622 只要unp_defer标志非0，就执行do while循环。我们将看到，一旦发现一个以前处理过的file结构，就置上这个标志，我们认为这个file结构不在使用中，但是实际上是在使用的。一旦这种情况发生，我们需要再次回过头来检查所有的file结构，因为有一个可能，就是我们刚才标志为忙的结构本身就是一个Unix域插口，并且这个Unix域插口在它的接收队列上包括file引用。

4. 循环检查所有的file结构

601-603 这个循环检查内核中的所有file结构，如果一个结构不在使用中(f_count等于0)，我们就跳过去。

uipc_usrreq.c

```
587 void
588 unp_gc()
589 {
590     struct file *fp, *nextfp;
591     struct socket *so;
592     struct file **extra_ref, **fpp;
593     int     nunref, i;

594     if (unp_gcing)
595         return;
596     unp_gcing = 1;
597     unp_defer = 0;
598     for (fp = filehead.lh_first; fp != 0; fp = fp->f_list.le_next)
599         fp->f_flag &= ~(FMARK | FDEFER);
600     do {
601         for (fp = filehead.lh_first; fp != 0; fp = fp->f_list.le_next) {
602             if (fp->f_count == 0)
603                 continue;
604             if (fp->f_flag & FDEFER) {
605                 fp->f_flag &= ~FDEFER;
606                 unp_defer--;
607             } else {
608                 if (fp->f_flag & FMARK)
609                     continue;
610                 if (fp->f_count == fp->f_msgcount)
611                     continue;
612                 fp->f_flag |= FMARK;
613             }

614             if (fp->f_type != DTYPE_SOCKET ||
615                 (so = (struct socket *) fp->f_data) == 0)
616                 continue;
617             if (so->so_proto->pr_domain != &unixdomain ||
618                 (so->so_proto->pr_flags & PR_RIGHTS) == 0)
619                 continue;
620             unp_scan(so->so_rcv.sb_mb, unp_mark);
621         }
622     } while (unp_defer);
```

uipc_usrreq.c

图18-16 unp_gc 函数：第一部分，标记阶段

5. 处理延迟的结构

604-606 如果已经置上FDEFER标志，那么就要关闭这个标志，并且unp_defer计数器也要减小。当unp_mark置上FDEFER标志时，FMARK标志也被置上，这样我们就知道这个记录项在使用中，并且我们还将检查在if语句的末尾是否是一个Unix域插口。

6. 跳过已经处理过的结构

607-609 如果设置了FMARK标志，那么记录项正在使用中，并且已经被处理过了。

7. 不标记丢失的结构

610-611 如果f_count等于f_msgcount，则这个记录项会可能丢失。它没有被标记，并被跳过去了。由于它似乎不在使用中，所以我们不能检查它是否是一个在接收队列上有传送中描述符的Unix域插口。

8. 标记使用中的结构

612 在这一点上我们知道这个记录项在使用中，所以要置上 FMARK标志。

9. 检验结构是否与一个Unix域插口相连

614-619 既然这个记录项在使用中，我们就检验看它是否是一个有 socket结构的插口。下一次检验确定这个插口是否是带有 PR_RIGHTS标志集的Unix域插口。设置这个标志是为了Unix域流和数据报协议。如果任何一个测试结果是错的，就要跳过这个记录项。

10. 扫描Unix域插口接收队列上传送中的描述符

620 在这一点上file结构对应一个Unix域插口。unp_scan遍历插口接收队列，寻找包含传送中描述符的类型为MT_CONTROL的mbuf。如果发现，就调用unp_mark。

> 在此处，源代码也应当能处理 Unix域插口的已完成连接队列 (so_q)[McKusick et al.1996]，一个客户进程把描述符传送给一个新创建的还在等着接收的服务器插口是完全可能的。

图18-17给出了一个标记阶段的例子，并且在标记阶段可能需要多次扫描 file结构链表。这个图描述了在标记阶段第一次扫描完成时的结构状态，此时 unp_defer为1，需要再一次扫描所有的file结构。当从左至右处理四个file结构时，开始下列处理过程。

1) file结构在引用它的进程中有两个描述符 (f_count等于2)，但是没有引用传送中的描述符(f_msgcount等于0)。图18-16中的代码设置f_flag字段中的FMARK比特位。这个结构指向一个 vnode(我们忽略了f_type值的DTYPE_前缀，另外我们仅仅给出了f_flag字段中的FMARK和FDEFER标志，实际上其他标志值也有可能在这个字段上出现)。
2) 因为f_count等于f_msgcount，所以这个结构好像没有被引用。当被标记阶段处理后，f_flag字段不变。
3) 因为这个结构被进程中的一个描述符引用，所以要置上 FMARK标志。还有，由于这个结构对应于一个Unix域插口，unp_scan还要处理插口接收队列上的任何控制报文。
   控制报文中的第一个描述符指向第二个 file结构，并且由于在第二步中没有设置FMARK标志，unp_mark就要置上FMARK和FDEFER这两个标志。因为这个结构已经处理过，并且发现没有被引用，所以unp_defer也要增加到1。
   控制报文中的第二个描述符指向第四个 file结构，并且由于没有置上 FMARK标志(它甚至还没有被处理过)，因此就要置上FMARK和FDEFER标志，unp_defer增加到2。

4) 设置这个结构的FDEFER标志，所以图18-16中的代码关闭这个标志，并将unp_defer减小到1。即使这个结构也被进程中的描述符引用，但是因为已经知道这个结构被传送中的描述符引用，从而也就不需要检查它的f_count和f_msgcount值。

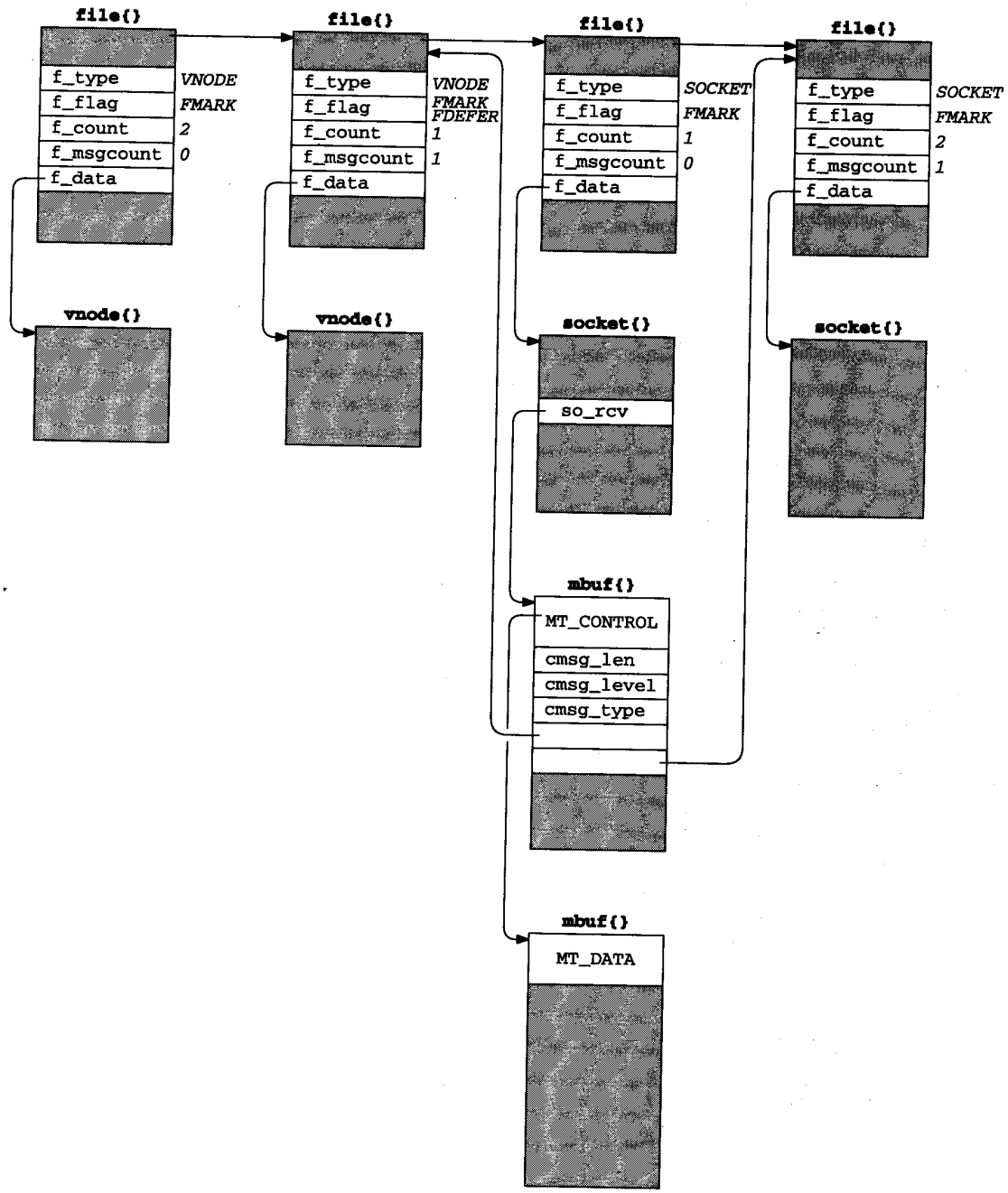


图18-17 标记阶段第一次扫描后的数据结构

此时，所有四个file结构都被处理过了，但是unp_defer等于1，所以就需要再一次扫描所有的结构。因为确信第一次循环没有引用过的第二个结构也许是一个 Unix域插口，并且

在这个Unix域插口的接收队列上有控制报文，所以要产生再一次循环（这不在我们的例子中）。那个结构需要被再次处理，并且当情况是这样时，这次循环可能会在认为没有被引用的链表中靠前的一些结构中置上FMARK和FDEFER标志。

在标记阶段的结尾涉及到多次扫描内核的file结构链表，其中没有标志的结构不在使用中。函数的第二段，即回收(sweep)部分，如图18-18所示。

uipc_usrreq.c

```
623     /*
624      * We grab an extra reference to each of the file table entries
625      * that are not otherwise accessible and then free the rights
626      * that are stored in messages on them.
627      *
628      * The bug in the orginal code is a little tricky, so I'll describe
629      * what's wrong with it here.
630      *
631      * It is incorrect to simply unp_discard each entry for f_msgcount
632      * times -- consider the case of sockets A and B that contain
633      * references to each other.  On a last close of some other socket,
634      * we trigger a gc since the number of outstanding rights (unp_rights)
635      * is non-zero.  If during the sweep phase the gc code unp_discards,
636      * we end up doing a (full) closef on the descriptor.  A closef on A
637      * results in the following chain.  Closef calls soo_close, which
638      * calls soclose.   Soclose calls first (through the switch
639      * uipc_usrreq) unp_detach, which re-invokes unp_gc.  Unp_gc simply
640      * returns because the previous instance had set unp_gcing, and
641      * we return all the way back to soclose, which marks the socket
642      * with SS_NOFDREF, and then calls sofree.  Sofree calls sorflush
643      * to free up the rights that are queued in messages on the socket A,
644      * i.e., the reference on B.  The sorflush calls via the dom_dispose
645      * switch unp_dispose, which unp_scans with unp_discard.  This second
646      * instance of unp_discard just calls closef on B.
647      *
648      * Well, a similar chain occurs on B, resulting in a sorflush on B,
649      * which results in another closef on A.  Unfortunately, A is already
650      * being closed, and the descriptor has already been marked with
651      * SS_NOFDREF, and soclose panics at this point.
652      *
653      * Here, we first take an extra reference to each inaccessible
654      * descriptor.  Then, we call sorflush ourself, since we know
655      * it is a Unix domain socket anyhow.  After we destroy all the
656      * rights carried in messages, we do a last closef to get rid
657      * of our extra reference.  This is the last close, and the
658      * unp_detach etc will shut down the socket.
659      *
660      * 91/09/19, bsy@cs.cmu.edu
661      */
662     extra_ref = malloc(nfiles * sizeof(struct file *), M_FILE, M_WAITOK);
663     for (nunref = 0, fp = filehead.lh_first, fpp = extra_ref; fp != 0;
664          fp = nextfp) {
665         nextfp = fp->f_list.le_next;
666         if (fp->f_count == 0)
667             continue;
668         if (fp->f_count == fp->f_msgcount && !(fp->f_flag & FMARK)) {
669             *fpp++ = fp;
670             nunref++;
671             fp->f_count++;
```

图18-18 unp_gc 函数：第二部分，回收阶段

```
672             }
673         }
674         for (i = nunref, fpp = extra_ref; --i >= 0; ++fpp)
675             if ((*fpp)->f_type == DTYPE_SOCKET)
676                 sorflush((struct socket *) (*fpp)->f_data);
677         for (i = nunref, fpp = extra_ref; --i >= 0; ++fpp)
678             closef(*fpp, (struct proc *) NULL);
679         free((caddr_t) extra_ref, M_FILE);
680         unp_gcing = 0;
681 }
```

uipc_usrreq.c

图18-18 (续)

11. 更正错误的注释

623-661 注释涉及到4.3BSD Reno和Net/2版本里的一个错误，这个错误在4.4BSD里由Bennet S. Yee更正，注释里提到的旧代码如图18-19所示。

12. 分配临时区域

662 malloc为指向内核中所有file结构的指针数组分配空间。nfiles是当前使用中的file结构数量。M_FILE标识使用内存的目的(vmstat -m命令输出关于内核存储器使用的信息)。如果当前得不到可用内存，那么M_WAITOK导致进程转入睡眠状态。

13. 遍历所有的file结构

663-665 为了发现所有没有引用的(丢失的)结构，这个循环再次检查内核中的所有file结构。

14. 跳过没有使用过的结构

666-667 如果file结构的f_count是0，就跳过这个结构。

15. 检查未引用的结构

668 如果在标记阶段，f_count等于f_msgcount(唯一的引用来自于传送中的描述符)，并且没有设置FMARK标志(传送中的描述符没有出现在任何Unix域插口接收队列上)，那么这个记录项是没有被引用的。

16. 保存指向file结构的指针

669-671 fp的一个副本，即指向file结构的指针，保存在分配的数组中，递增计数器nunref，递减file结构的f_count。

17. 对没有引用的插口调用sorflush

674-676 对每一个没有被引用的插口文件调用sorflush函数。函数sorflush(卷2的图15-37)调用域的dom_dispose和unp_dispose函数，unp_dispose调用unp_scan删除当前插口接收队列上任何传送中的描述符。 unp_discard递减f_msgcount和unp_rights，并且对在插口接收队列上控制报文中的所有file结构调用closef。由于我们对这个file结构(早些时候完成f_count的递增)有一个额外的引用，而且由于那个循环忽略了f_count为0的结构，从而我们确信 f_count等于2或者比2还要大。所以作为sorflush的结果去调用closef将把file结构的f_count减小到一个非0值，从而避免完全关闭该结构。这就是为什么对结构的额外引用进行得比较早。

18. 执行最后的关闭

677-678 对所有没有引用的file结构调用closef。这是最后一次关闭，也就是说，

f_count应当从1减到0，从而导致插口关闭，并返回file结构给内核的空闲缓存池。

19. 返回临时数组

679-680 返回早些时候由malloc分配的数组，并清除unp_gcing标志。

图18-19表示了unp_gc函数的回收阶段，同Net/2版本一样，这部分代码被图18-18中的代码替换。

```
    for (fp = filehead; fp; fp = fp->f_filef) {
        if (fp->f_count == 0)
            continue;
        if (fp->f_count == fp->f_msgcount && (fp->f_flag & FMARK) == 0)
            while (fp->f_msgcount)
                unp_discard(fp);
    }
    unp_gcing = 0;
}
```

图18-19 Net/2版本中unp_gc 函数回收阶段的错误代码

这就是在图18-18开始部分的注释中谈到的代码。

不幸的是，虽然在本节讨论的Net/3代码对图18-19中的代码进行了改进，并且在图18-18的开始部分描述了错误的更正，但是代码仍然是不正确的，在本节开始部分提到的前两种情况下，file结构仍是有可能丢失的。

18.10 unp_mark函数

当unp_gc调用unp_scan时，unp_mark函数被unp_scan调用去标记一个file结构。当在插口接收队列上发现传送中的描述符时完成标记过程，图18-20给出了这个函数。

uipc_usrreq.c

```
717 void
718 unp_mark(fp)
719 struct file *fp;
720 {
721     if (fp->f_flag & FMARK)
722         return;
723     unp_defer++;
724     fp->f_flag |= (FMARK | FDEFER);
725 }
```

uipc_usrreq.c

图18-20 unp_mark 函数

717-720 参数fp是指向file结构的指针，这个file结构是在Unix域插口接收队列上的控制报文里发现的。

1. 如果记录项已经被标记，就返回

721-722 如果file结构已经被标记，则不需做任何工作，因为已经知道file结构在使用过程中。

2. 设置FMARK和FDEFER标志

723-724 递减unp_defer计数器，并且设置FMARK和FDEFER标志。如果在内核列表里这

个file结构比Unix域插口file结构出现得早(也就是说，这个file结构已经由unp_gc处理过了，并且似乎不在使用过程中，所以没有被标记)，那么在unp_gc函数的标记阶段unp_defer的增加会导致另一次对所有file结构的遍历。

18.11 性能(再讨论)

我们已经讨论了Unix域协议的实现，现在返回到它们的性能上来看看，为什么要比 TCP快两倍(图16-2)。

所有的插口I/O都调用sosend和soreceive，与协议无关。这有利有弊，有利是因为这两个函数满足许多不同协议的需要，从字节流 (TCP)到数据报协议(UDP)，以及基于记录的协议(OSI TP4)。不利的原因是其一般性降低了性能，并使代码复杂化。对于不同的协议形式，这两个函数的优化版本会提高性能。

比较输出性能，对于TCP，通过sosend的路径几乎与Unix域流协议的路径相同。假定大的应用程序进行写操作(图16-2中用32 768字节写)，sosend函数把用户数据打包放到mbuf簇中，然后通过PRU_SEND请求将每一个2 048字节簇传送给协议。所以，无论是 TCP还是Unix域都要处理相同数量的PRU_SEND请求。对于速度上的差异， Unix域PRU_SEND(图18-2)的输出应当比TCP输出(它调用IP输出把每一段添加到环回驱动器输出队列)简单。

由于PRU_SEND请求把数据放到接收插口的接收缓存，所以在接收方唯一与 Unix域插口有关的函数是soreceive。尽管如此，对于TCP，环回驱动器把每一段数据放到 IP输入队列上，后面紧跟着IP处理，再后面跟着TCP输入处理把每一段分解到正确的插口，然后将数据放到插口的接收缓存。

18.12 小结

当把数据写到一个Unix域插口时，立即将数据添加到接收插口的接收缓存中，没有必要将发送插口发送缓存里的数据进行缓存。基于这个原因，为了使流插口能正确地工作，PRU_SEND和PRU_RCVD请求操纵发送缓存的高水位标记，从而使得这个标记总是反映对等端接收缓存中的空间数量。

Unix域插口提供了将描述符从一个进程传送到另一个进程的机制。对于进程间通信来说，这是一项强大的技术。当一个描述符从一个进程传送到另一个进程时，首先这个描述符要内部化(转换成对应的file指针)，再将这个指针传送到接收插口。当接收进程读到控制信息时，file指针要外部化(转换成接收进程中最小的、没有编号的描述符)。然后将描述符再返回到这个进程。

容易处理的一个错误情况是，当Unix域插口的接收缓存中有传送中描述符的控制信息时，插口关闭。不幸的是还有其他两种不容易处理的错误情况：一种是接收进程没有请求接收在其接收缓存中的控制信息；另一种是接收缓存中没有足够的空间来保存控制信息。在这两种错误情况下就会丢失file结构，这就是说，它们既不在内核的空闲缓存池中，也不在使用中。从而需要无用单元收集函数回收这些丢失的结构。

无用单元收集函数执行一个标记阶段，在这个标记阶段中扫描所有内核的 file结构，同时把在使用中的描述符置上标志，在后面紧跟着回收阶段，回收所有没有被标记的结构。虽然需要这个函数，但是很少使用它。

附录A　测量网络时间

本书正文中用到了分组经过网络传输时传输时间的测量。本附录给出其细节和我们能够测量的各种时间的测量例子。我们要介绍用 Ping程序实现的RTT测量，向上和向下经过协议栈的时间测量，以及等待时间与带宽的差异。

网络程序员或系统管理员通常有两种办法可以用来测量应用事务所需的时间：

1) 采用应用程序定时器。例如，在图 1-1的UDP客户程序中，我们在调用 `sendto`之前取到了系统时钟，在`recvfrom`函数返回后又取到了系统时钟，其差额就是应用程序发送请求至收到应答的时间。

   如果内核提供了高精度的时钟(ms数量级)，则我们所测得的值(几毫秒或以上)就很精确。卷1的附录A给出了这一类测量方法的更多细节。

2) 采用软件工具来监视指定分组，并计算相应的时间差，如嵌入到数据链路层的Tcpdump。在卷1的附录A中有这些工具的更多细节。

   在这本书中，我们假定数据链路的嵌入点在 Tcpdump中用BSD分组过滤器(BPF)提供。卷2的第31章给出了BPF实现的许多细节。卷2图4-11和图4-19说明了在典型以太网驱动程序中哪里要有BPF调用，图15-27则说明了在环路测试驱动程序中的 BPF调用。

我们注意到本书中的例子用到的系统 (图1-13)，包括80386上的BSD/OS 2.0和Sparcstation ELC上的Solaris 2.4，都给应用程序计时和Tcpdump时间戳提供高精度的定时器。

最可靠的方法是在网络电缆上连接一个网络分析仪，但往往没有这样的仪器。

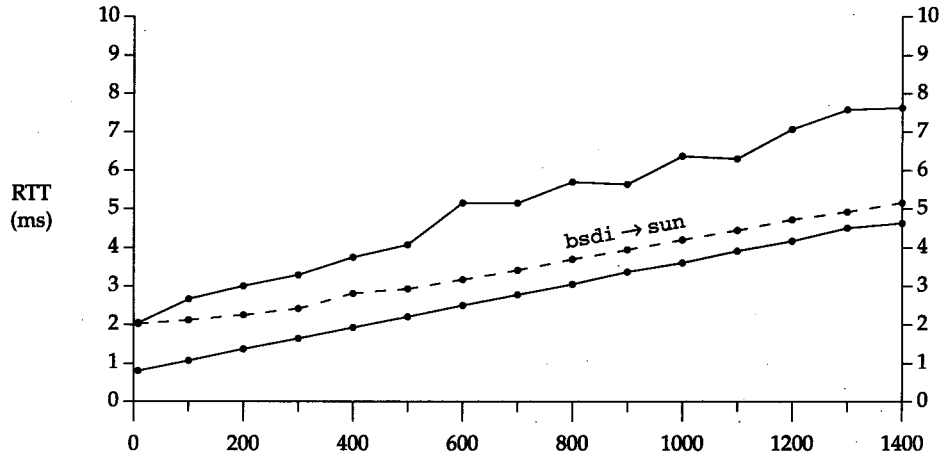
A.1　利用Ping的RTT测量

在卷1的第7章详细介绍了无所不在的 Ping程序，利用应用定时器来计算 ICMP分组的RTT(往返时间)。程序发送一个ICMP回显请求分组给服务器，服务器紧接着向客户回复一个回显应答分组。客户可以在回显请求分组中将发送时的时钟值作为用户可选数据记录在该分组中，然后服务器会在应答中返回这个时钟值。客户收到回显应答时，它就取当前时钟值计算出RTT，然后打印出来。图A-1给出了Ping分组的格式。

图A-1　Ping分组：ICMP回显请求或ICMP回显应答

Ping程序允许我们指定分组中可选用户数据的长度，使我们能够测量分组长度对 RTT的影响。如果是用Ping来测量RTT，可选数据的长度必须至少8字节(客户发出和服务器应答的时间戳要占用8个字节)。如果指定的用户数据长度少于8字节，Ping也能工作，但不能计算并打印RTT。

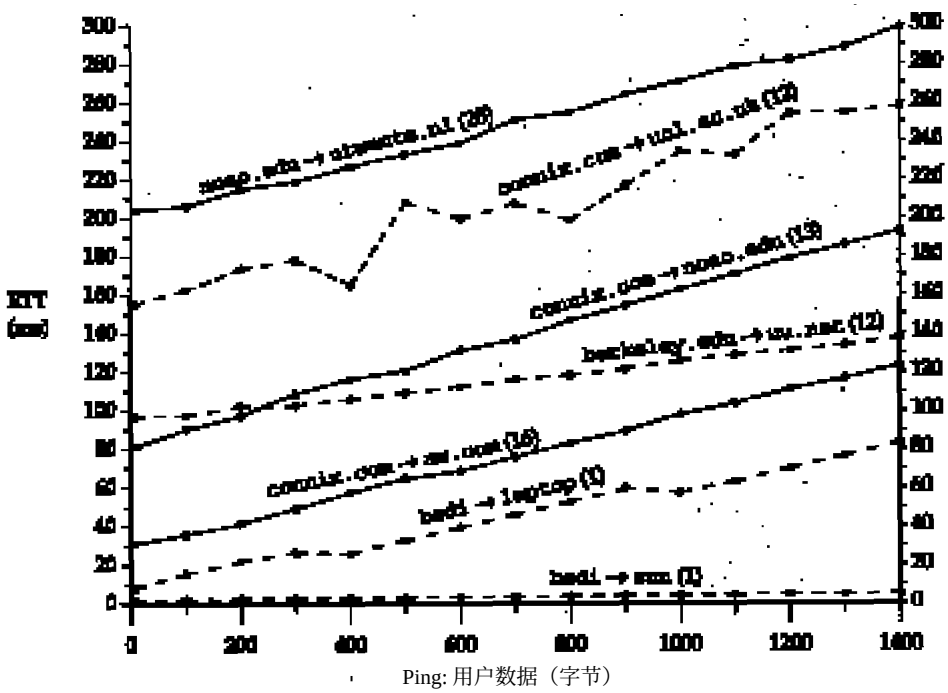
图A-2画出了在三个不同局域网上的主机间用 Ping测得的RTT典型值。图中间的一条线是图1-13中主机`bsdi`和`sun`间的RTT。



图A-2 三个以太网上的主机间Ping RTT值

用15个不同的分组长度进行了测量：8字节用户数据以及从100至1400字节的用户数据(以100字节递增)。加上20字节的IP首部和8字节的ICMP首部，IP数据报的长度就在36~1428字节之间。对每一个分组长度都进行了10次测量，图中只画出了10个值中最小的那个。与我们所期望的一致，分组长度增加后RTT也增大。三条线之间的差别是因处理器速度、接口卡和操作系统的不同而造成的。

图A-3给出了经Internet、WAN互连的各种主机之间典型的RTT值。注意*y*轴的刻度与图A-2中的有差别。



图A-3 经Internet(一个WAN)互连的主机间Ping RTT值

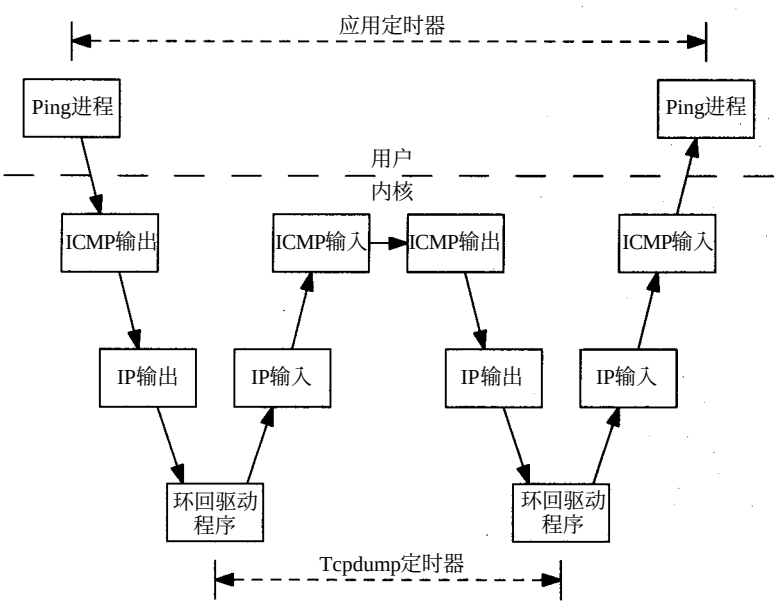
与在LAN上测量那样，对WAN进行了同样的测量：对15个不同长度的分组各进行了10次测量，每个分组长度只画出了10个值中最小的那个值。图中的括号中是每对主机之间的转发段数。

图中最上边的曲线（最长的RTT）表示Internet上分别位于Arizona(noao.edu)和Netherlands(utwente.n1)的一对主机之间需要25段转发。自上而下的第2条曲线也是跨越大西洋的，是Connecticut(connix.com)和London(ucl.ac.uk)之间的一对主机。接下来两条曲线在美国内部，分别是Connecticut和Arizona之间的一对主机(connix.com与noao.edu)，以及California和Washington D.C.之间的一对主机(berkeley.edu和uu.net)。再接下来的曲线是地理上很近的一对主机(Connecticut的connix.com和Boston的aw.com)，但从经过Internet传送的转发段数(16)来衡量，却是离得很远的。

图中底部的两条线(RTT值最小的)是作者所在局域网(图1-13)上的主机之间的。其中最底下的那条线是从图A-2复制来的，以便对典型的LAN上的RTT与典型的WAN上的RTT进行比较。在最底下的第2条线，即bsdi和laptop之间的RTT线，后者的以太网卡是插在计算机的并行口上的。尽管该系统也是接在以太网上的，但由于并行口的传输速率较慢，看上去就像是接在WAN上一样。

A.2 协议栈测量

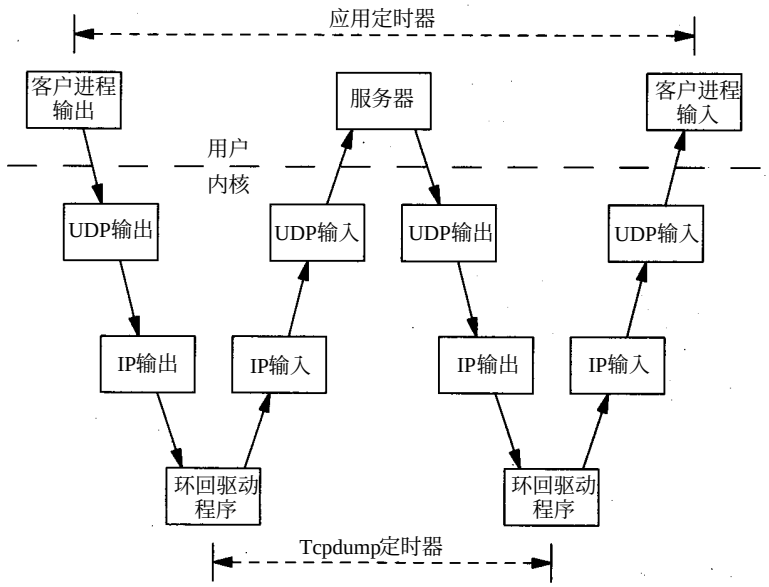
我们也可以用Ping，并加上Tcpdump来测量在协议栈上花费的时间。例如，图A-4中就给出了在一台主机上运行Ping和Tcpdump，对环回测试地址(一般是127.0.0.1)，Ping的执行步骤。



图A-4 在一台主机上运行Ping和Tcpdump

假设应用程序在就要向操作系统发出回显请求分组时启动定时器，然后在操作系统返回回显应答时停掉定时器，应用程序测得的时间差和Tcpdump测得的时间差就分别是ICMP输出、IP输出、IP输入和ICMP输入之间所需的时间。

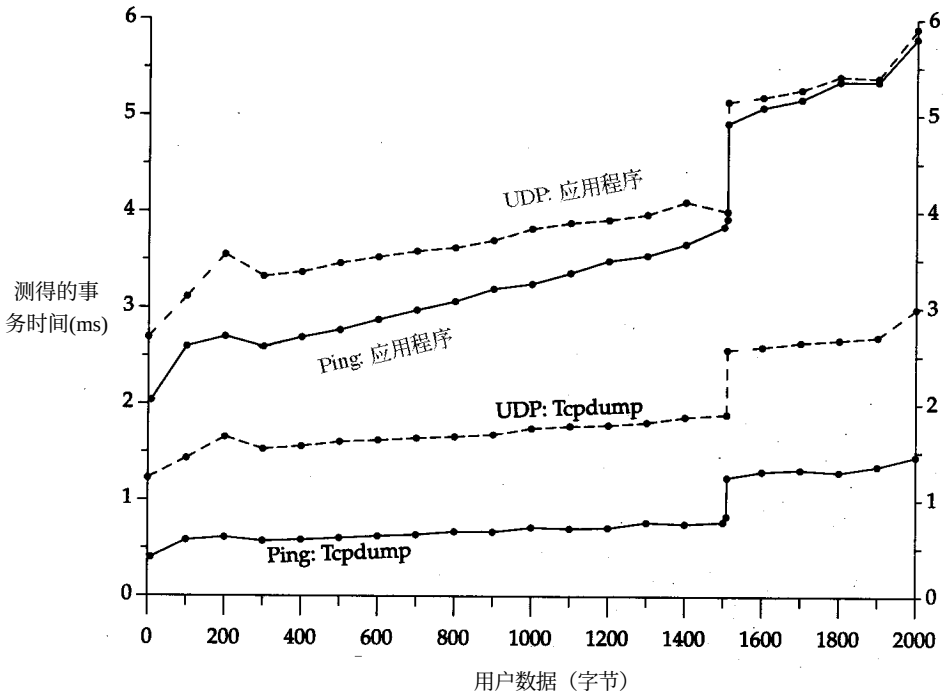
我们也可以测量任何客户—服务器应用程序之间的类似值。图A-5给出了1.2节UDP客户—服务器应用的处理步骤，其中假设客户和服务器在同一台主机上。



图A-5 UDP客户-服务器事务的处理步骤

这个UDP的客户-服务器例子与图A-4的Ping例子之间的一个不同之处是，这里的UDP服务器是一个用户进程，而Ping服务器则是ICMP内核的一部分(卷2图11-21)。因此，UDP服务器中在内核和用户进程之间要有两份客户数据：服务器输入和服务器输出。内核与用户进程之间复制数据通常都是比较费时的操作。

图A-6给出了在主机bsdi上进行的各项测试结果，可以比较Ping客户-服务器和UDP客



图A-6 单个主机上(环回接口)的Ping和Tcpdump测量结果

户-服务器这两种方式。图中y轴标的是“测得的事务时间”，因为RTT通常都是指网络的往返时间或Ping的时间输出(在图A-8中可以看到，它与网络的RTT非常接近)。在这里的UDP、TCP和T/TCP客户-服务器方式中，可以测量应用程序的事务时间。在TCP和T/TCP的例子中，这可能要包括多个分组和多次网络RTT。

在这个图的Ping测量中采用了23种不同的分组长度：用户数据从100字节到2 000字节变化，增量为100字节，再加上8、1508和1509字节。其中8字节是用Ping来测量RTT的最短用户数据长度，1508是不会在IP层分段的最大数据长度，因为BSD/OS采用了1536的MTU作为环测接口(1508+20+8)。1509字节则是会在IP层进行分段的最小数据长度。

在UDP测量中也采用了23种类似长度的分组：用户数据长度从100字节到2 000字节变化(增量100)，再加上0、1508和1509。0字节的UDP数据报也是允许的。由于UDP的首部与ICMP回显测试分组的首部一样长(8字节)，1508又是避免在环测接口上分段的最大分组，1 509是需要分段的最小分组。

我们首先注意到的是在用户数据为1509字节时的时间跳变，这时需要分段。这也是想像之中的。当出现分段时，在图A-4和图A-5中左边对“IP输出”的一次调用会产生两次对“环测驱动程序”的调用，每段一次。从1508到1509，即使用户数据只增加了一个字节，应用程序就感觉到事务时间增加了近25%，因为多出一个每分组处理时间。

所有4条线中，在200字节点的时间增加是由于BSD的mbuf实现中的非自然处理造成的(卷2的第2章)。对于最小分组(UDP测量中的0字节用户数据和Ping测量中的8字节用户数据)，数据和分组的首部可以写入一个mbuf中，在100字节点需要第二个mbuf，在200字节点则需要第三个mbuf。最后，在300字节点，内核开始采用2048字节的mbuf簇来代替较小的mbuf。看起来，用一个mbuf簇比用多个mbuf会快一些(例如，在100字节点)，可以减少处理时间。这是典型的时间—空间折衷的例子。从采用较小的mbuf到采用较大的mbuf簇的切换条件是数据量是否超过208字节，这是在许多年前当内存还很紧张时设计的。

图1-14中的定时测量是用修改后的BSD/OS内核实现的，其中的常数MINCLSIZE(卷2图2-7和图16-25)从208改为101。这样就使得一旦用户数据超过100字节就分配mbuf簇。只要注意就可以看到，图1-14中没有在200字节点出现尖角。

我们在14-11节也讨论过这个问题，在那里我们看到，许多Web客户请求都在100~200字节之间。

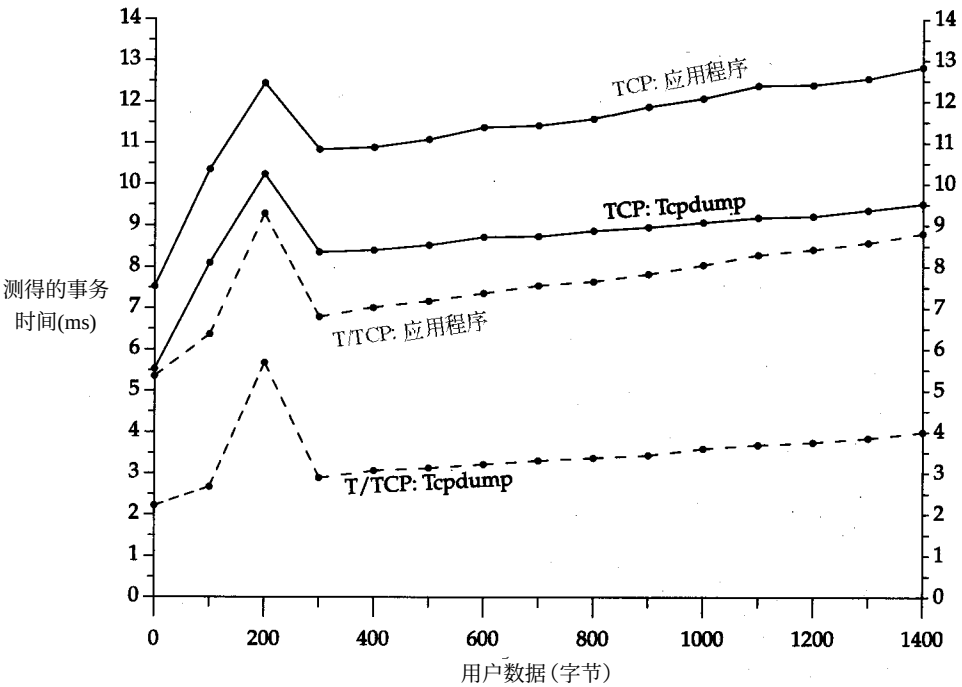
图A-6中，开始分段之前，两条UDP曲线之间的相差在1.5~2 ms之间。因为这个差额已经把UDP输出、IP输出、IP输入和UDP输入(图A-5)考虑在内，如果我们假设协议的输出逼近于协议输入，那么就相当于分组发送时向下穿过协议栈要花不到1ms的时间，接收时分组向上穿过协议栈又要花不到1 ms的时间。这些时间包括了发送时要将多份数据从进程传递给内核，以及数据返回时从内核传递到进程。

由于图A-5中Tcpdump测量要经历同样的4个步骤(IP输入、UDP输入、UDP输出和IP输出)，我们可以预计到UDP Tcpdump的两条曲线相差也在1.5~2 ms之间(只考虑发生分段前的值)。与第一个数据点不同，图A-6中其余的数据也在1.5~2 ms之间。

如果我们考虑发生了分段以后的值，图A-6中两条UDP曲线之间相差2.5~3 ms。跟预期的一样，UDP Tcpdump的值也在2.5~3 ms之间。

最后可以看到，图A-6中，Ping的Tcpdump曲线几乎是平坦的，但Ping的应用程序测量则有一个正的斜率。这很可能是因为应用程序测量了两份用户进程和内核之间的数据，但Tcpdump则一份也不需要测量(因为Ping服务器是内核的ICMP实现的一部分)。另外，Ping的Tcpdump曲线非常轻微的正斜率很可能是由于内核Ping服务器的两次操作造成的，这些操作对每一个字节都要执行：接收ICMP的检验和验证和输出ICMP的检验和计算。

我们也可以修改1.3节和1.4节的TCP和T/TCP客户-服务器应用，以测量每一次事务的时间(见1.6节的叙述)，并对不同分组长度进行测量。测量结果见图A-7(在本附录余下的事务测量中，我们测到用户数据长度为1400字节就结束了，因为TCP不分段)。



图A-7 单个主机上(环回接口)的TCP和T/TCP客户-服务器事务时间测量结果

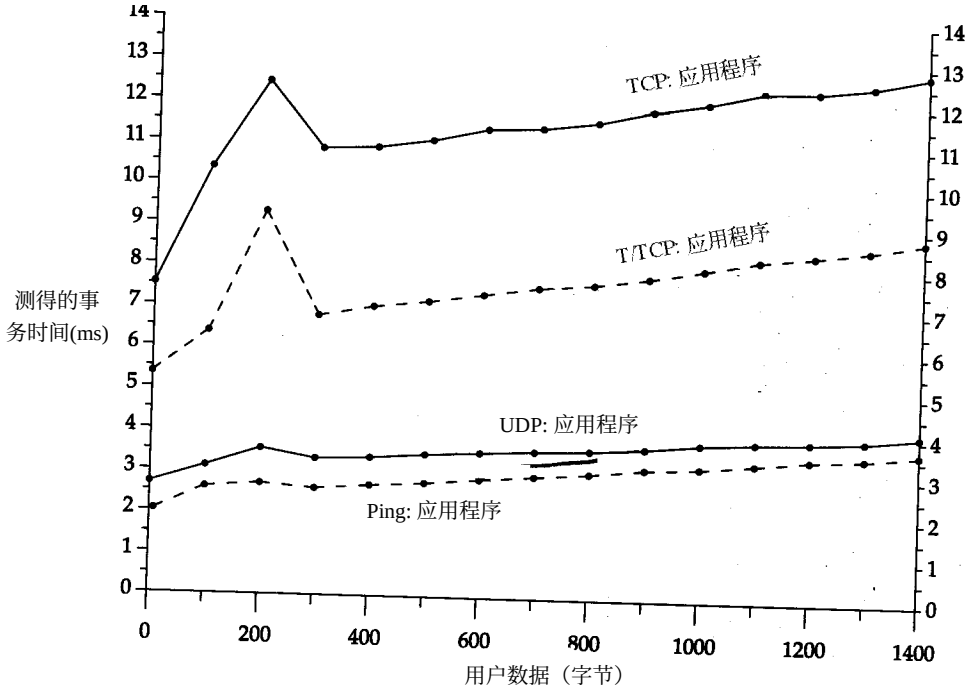
Tcpdump曲线测量的是从客户发出第一个报文段(对TCP客户是一个SYN，对T/TCP客户则是SYN、数据和FIN的组合)至收到服务器发回的最后一个报文段(对TCP客户是一个FIN，对T/TCP客户则是数据和FIN的组合)为止的时间间隔。相应地，TCP应用曲线和TCP Tcpdump曲线之间的差别也就是协议栈用于处理connect和FIN所需的时间(图1-8给出了分组交换)。T/TCP客户-服务器的应用曲线和Tcpdump曲线的差别就是协议栈用于处理客户的sendto所需的时间，其中包括客户数据和最后一个FIN(图1-12给出了分组交换)。我们可以看到，两条T/TCP曲线之间的差距(大约4 ms)大于两条TCP曲线之间的差距(大约2.5 ms)，这是合理的，因为T/TCP的协议栈处理量(在第一段报文中要发送SYN、数据和FIN)比TCP的(第一段报文中只发送SYN)要大。

4条曲线均在200字节处开始上升，再次说明内核应该尽快采用mbuf簇。注意，TCP和T/TCP在200字节处的增加比在图A-6中Ping和UDP的要大得多。对于数据报协议(ICMP和UDP)来说，尽管分配了3个mbuf来缓存首部和用户数据，但内核中插口层对协议输出例程的

调用只有一次(卷2的16.7节称，该调用是sosend函数)。而对流协议(TCP)来说，对TCP输出例程的调用有两次：一次是前 100字节用户数据，另一次是第 2个100字节用户数据。确实，Tcpdump证实了要传送两个 100字节报文段的事实。对协议输出例程多了一次调用就增加了开销。

TCP和T/TCP应用曲线之间的差别大约是 4 ms，对所有分组长度几乎都一样，因为 T/TCP处理的报文段少。图 1-8和图1-12给出了9个TCP报文段和3个TCP报文段。报文段数的减少明显减轻了两端主机的处理开销。

图A-8总结了图A-6和图A-7中的Ping、UDP以及T/TCP和TCP客户 - 服务器的应用时间测量，没有考虑Tcpdump的时间测量。



图A-8　单个主机上(环回接口)的Ping、UDP及TCP和T/TCP客户 - 服务器事务时间测量

结果是预料之中的。Ping所需的时间最少，没有比它更快的了，因为 Ping服务器是在内核中的。UDP事务所需时间略大于 Ping的时间，因为数据要在内核与服务器之间复制两次以上，但并不大，是 UDP所需的最小处理时间。 T/TCP事务所需的时间大约是 UDP的两倍，因为尽管分组数量与UDP相同，但需要更多的协议处理时间 (我们的应用程序定时器并不包括图 1-12中最后的ACK)。TCP的事务时间大约比 T/TCP多50%，因为协议需要处理的分组数较多。图A-8中UDP、T/TCP和TCP之间的相对时间差与图 1-14中的不一样，因为第 1章中的测量是在实际网络上进行的，而本附录中的测量是在环测接口上进行的。

A.3　滞后和带宽

在网络通信中，有两个因素在决定交换信息所需的时间：滞后和带宽 [Bellovin 1992]。这里忽略了服务器处理时间和网络负荷，以及其他明显影响客户事务时间的因素。

滞后(也称为传播时延)是将一个比特从客户传递到服务器再传回来所需的固定时间，受光速的限制，从而决定于两个主机之间电或光信号的传播距离。横跨美国东西两岸之间的事务RTT不会低于60 ms，除非有人可以提高光速。对滞后可做的唯一控制是将两台主机移近，或避免使用高滞后的路径(如卫星链路)。

理论上，光波穿越美国的时间应该是大约16 ms，最小的RTT是32 ms。60 ms是实际的RTT。作为试验，作者曾在分别位于美国东西海岸的主机上运行过Traceroute，只观察横跨美国的直达链路两端的路由器之间的RTT。加州与华盛顿之间的RTT是58 ms，加州与波士顿之间的RTT是80 ms。

另一方面，带宽度量每个比特进入网络的速度，发送方以这个速度顺序将数据送入网络。增加带宽只要购买更快的网络即可。例如，如果T1线路还嫌不够快(大约1 544 000 bit/s)，你可以租用T3线路(大约45 000 000 bit/s)。

可以用公园的软管作恰当的比喻(感谢Ian Lance Taylor)：滞后是水从水龙头流到喷口所需的时间，而带宽就相当于每秒从喷口流出的水量。

一个问题是，网络越来越快(即，带宽增加)，但滞后保持不变。例如，要用横跨美国的T1线路发送100万字节的数据(假设单程滞后是30 ms)，需要5.21秒：5.18秒是带宽决定的，另外0.03秒是滞后造成的。这时带宽是主要影响。但是，如果采用T3线路，则总时间是208 ms：178 ms是带宽决定的，另外30 ms是滞后造成的。这时滞后是带宽时延的1/6。而以150 000 000 b/s发送则需要82 ms：52 ms是带宽决定的，30 ms是滞后造成的。在最后这个例子中，滞后越来越接近带宽时延，而在更快的网络中，滞后就成为主要的时延因素，而不再是带宽。

在图A-3中，往返滞后基本上就是每条曲线与*y*轴的相交点。最上面两条曲线(大约在202ms和155 ms处相交)是美国和欧洲之间的，接下来的两条曲线(在98 ms和80 ms处相交)是横跨整个美国的，再下面这条(大约在30 ms处相交)是美国东海岸的两台主机之间的。

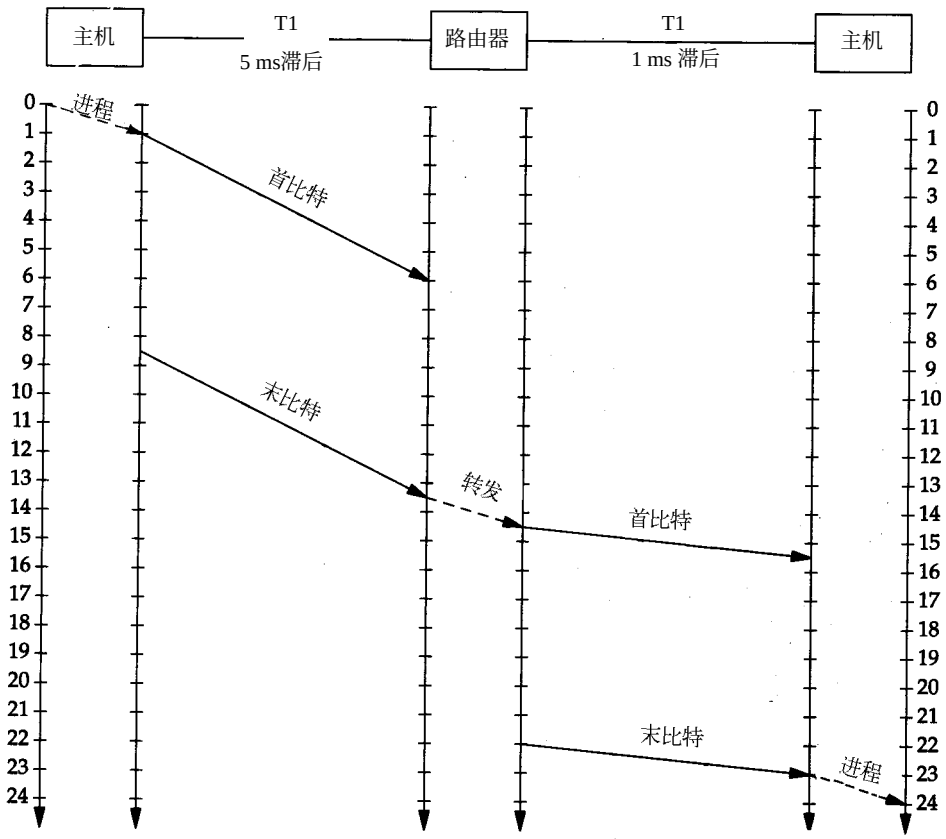
随着带宽增加，滞后变得越来越重要，这使得T/TCP更显优越。T/TCP至少使滞后减少了一个RTT。

顺序发送时延和路由器

如果我们将T1线路租给Internet服务商，用于向另一台以T1线路连接到Internet的主机发送数据，并且已知所有的中间线路都是T1或更高速率的线路，我们会对这样带来的结果感到惊奇。

例如，在图A-3中，如果我们来研究起始点为80 ms、终止点为193 ms的曲线，它是位于Connecticut的`connix.com`主机和位于Arizona的`noao.edu`主机之间的，它与*y*轴相交在80 ms处，正好反映了东西海岸之间的RTT(运行Traceroute程序，这在卷1的第8章有详细介绍，其结果说明分组的确切路由是从Arizona出发，回到California，然后到Texas、Washington DC，最后到Connecticut)。但如果我们计算在T1线路上发送1400字节所需的时间，大约只需要7.5 ms，因此可以估计1400字节分组的RTT应该是95 ms左右，远远低于实际测得的值193 ms。

出现这种情况是因为发送时延与中间路由器的数量成线性关系，因为每个路由器都必须在转发之前接收到整个数据报。考虑图A-9中的例子，要从左边的主机通过中间路由器传送一个1428字节长的分组到右边的主机。假设两条链路都是T1线路，发出1428字节大约需要7.5 ms。图中的时钟是从上向下增长的。



图A-9 数据的顺序发送

第1个箭头，从时刻0到1是主机处理输出数据报，根据本附录前面的测量，假设它需要1 ms。然后这些数据被发送到网络上，从开始发出第 1个比特到最后一个比特发完，需要 7.5 ms。另外在线路两端之间还有 5 ms的滞后，因此第1个比特到达路由器是时刻 6，最后一个比特则是在时刻13.5到达。

只有在时刻13.5最后一个比特到达以后，路由器才能转发该分组，我们假设转发又需要1ms时间。这样，路由器在时刻 14.5发出第1个比特，并且1 ms(第2段链路的滞后)以后到达目的主机。最后一个比特到达目的主机是在时刻 25。最后我们假设目的主机的处理又需要 1 ms。

确切的数据速率是在24 ms内传送了1428字节，或476 000 b/s，比T1的1/3还小。如果我们忽略主机和路由器处理分组的时间共3 ms，数据速率是544 000 b/s。

如前所述，顺序发送时延与分组所经过的路由器数量成线性关系。这项时延决定于线路速率(带宽)、分组长度和中间转发次数(路由器数)。例如，552字节分组(包含512字节数据的典型TCP报文段)在56 kb/s线路上是80 ms，在T1线路上是2.86 ms，而在T3线路上则只要0.10 ms。这样，10段T1线路就要给总时间加上 28.6 ms(几乎等于东西海岸之间的单程滞后)，而10段T3线路只增加1 ms(与滞后相比几乎可以忽略)。

最后，顺序发送时延是一种滞后效应，而不是带宽效应。例如，在图 A-9中，左边的发送主机可以在时刻8.5开始发送下一个分组的第 1比特，而不必等到时刻 24以后才开始发送下一

个分组。如果左边的主机连续发送 10个1428字节分组，假设分组之间没有间隙，则最后一个分组的最后一个比特的到达时刻是91.5(24+9×7.5)。这样的数据速率是1 248 525 b/s，非常接近T1的速率。对TCP来说，只是需要一个比较大的窗口来抵消顺序发送时延。

回到我们前面的例子，从connix.com到noao.edu，如果我们用Traceroute确定了确切的路径，知道了每条链路的速率，就可以把两台主机之间 12个路由器上的顺序发送时延考虑进去。这样，再假设滞后时间 80 ms，每个中间线路段有0.5 ms的处理时延，我们估算的总时延就是187 ms。这已经很接近实测值193 ms，比前面的估算值95 ms要接近得多。

附录B　编写T/TCP应用程序

在第一部分，我们介绍了T/TCP的两大好处：

1) 避免了TCP的三次握手。

2) 减少了连接持续时间短于MSL时处于TIME_WAIT状态的时间。

如果一个TCP连接两端的主机支持T/TCP，那么第2条好处是所有的TCP应用程序都能感受到的，不需要修改程序。

然而，为了避免三次握手，应用程序中对 `connect`和`write`函数的调用要改写为调用`sendto`和`sendmsg`。为了把FIN标志与数据组合在一起，应用程序必须在最后一次调用`send`、`sendto`或`sendmsg`函数时指定`MSG_EOF`标志，同时不再调用`shutdown`。我们在第1章介绍TCP和T/TCP的客户和服务器时说明了这些差别。

为了使可移植性最好，我们要求在编写应用程序时充分利用T/TCP，其条件是：

1) 将要执行编译的主机支持T/TCP，并且

2) 应用程序要编译成支持T/TCP。

如果运行程序的主机支持T/TCP，那么第2个条件也是在运行时要确定的，因为有时会在操作系统的某个版本上编译程序，而在另一个版本上运行。

如果在`<sys/socket.h>`头文件中定义了`MSG_EOF`标志，那就是说要执行程序编译的主机支持T/TCP。这会在C预处理器的`#ifdef`语句中用到。

```
#ifdef  MSG_EOF
        /* 主机支持  T/TCP      */
#else
        /* 主机不支持 T/TCP     */
#endif
```

第2个条件要求应用程序采用隐式打开（用`sendto`或`sendmsg`指定目标地址，不调用`connect`），但要考虑在主机不支持T/TCP时处理连接失败。在不支持T/TCP的主机上，当采用面向连接的插口但没有连接上时，所有的输出函数都会返回`ENOTCONN`(卷2的图16-34)。这一点对伯克利版系统和SVR4插口库都适用。举个例子，如果应用程序在调用`sendto`时接收到错误指示，那它就改为调用`connect`。

TCP或T/TCP的客户和服务器

我们可以在下面的程序中实现这些思想，这些程序只是对第1章的T/TCP与TCP的客户和服务器程序作了简单修改。与第1章中的C语言程序一样，这里也不对程序作详细介绍，同样假设读者已经对插口编程有一定的了解。第一个程序如图B-1所示，是客户的`main`函数。

8-13　用服务器的IP地址和端口号填入Internet插口地址结构，这两个参数来自命令行。

15-17　用函数`send_request`向服务器发送请求。如果一切正常，则这个函数返回插口描述符；否则返回一个负数，表示错误。第3个变量(1)告诉函数要在发送完请求以后再发送一个

结束标志。

18-19 函数read_stream与图1-6中的同名函数一样。

client.c

```
 1 #include    "cliserv.h"

 2 int
 3 main(int argc, char *argv[])
 4 {                                   /* T/TCP or TCP client */
 5     struct sockaddr_in serv;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n;

 8     if (argc != 3)
 9         err_quit("usage: client <IP address of server> <port#>");

10     memset(&serv, 0, sizeof(serv));
11     serv.sin_family = AF_INET;
12     serv.sin_addr.s_addr = inet_addr(argv[1]);
13     serv.sin_port = htons(atoi(argv[2]));

14     /* form request[] ... */

15     if ((sockfd = send_request(request, REQUEST, 1,
16                                (SA) &serv, sizeof(serv))) < 0)
17         err_sys("send_request error %d", sockfd);

18     if ((n = read_stream(sockfd, reply, REPLY)) < 0)
19         err_sys("read error");

20     /* process "n" bytes of reply[] ... */

21     exit(0);
22 }
```

client.c

图B-1 T/TCP或TCP客户的main 函数

函数send_request如图B-2所示。

sendrequest.c

```
 1 #include    "cliserv.h"
 2 #include    <errno.h>
 3 #include    <netinet/tcp.h>

 4 /* Send a transaction request to a server, using T/TCP if possible,
 5  * else TCP.  Returns < 0 on error, else nonnegative socket descriptor. */

 6 int
 7 send_request(const void *request, size_t nbytes, int sendeof,
 8               const SA servptr, int servsize)
 9 {
10     int     sockfd, n;

11     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
12         return (-1);

13 #ifdef  MSG_EOF                   /* T/TCP is supported on compiling host */

14     n = 1;
15     if (setsockopt(sockfd, IPPROTO_TCP, TCP_NOPUSH,
16                    (char *) &n, sizeof(n)) < 0) {
17         if (errno == ENOPROTOOPT)
```

图B-2 send_request 函数：用T/TCP或TCP发送请求

```
18              goto doconnect;
19          return (-2);
20      }
21      if (sendto(sockfd, request, nbytes, sendeof ? MSG_EOF : 0,
22                 servptr, servsize) != nbytes) {
23          if (errno == ENOTCONN)
24              goto doconnect;
25          return (-3);
26      }
27      return (sockfd);            /* success */

28    doconnect:                    /* run-time host does not support T/TCP */
29  #endif

30      /*
31       * Must include following code even if compiling host supports
32       * T/TCP, in case run-time host does not support T/TCP.
33       */

34      if (connect(sockfd, servptr, servsize) < 0)
35          return (-4);
36      if (write(sockfd, request, nbytes) != nbytes)
37          return (-5);
38      if (sendeof && shutdown(sockfd, 1) < 0)
39          return (-6);

40      return (sockfd);            /* success */
41  }
```

sendrequest.c

图B-2 (续)

1. 试试T/TCP的sendto

13-29 如果执行编译的主机支持T/TCP，这段程序就会执行。我们在3.6节讨论过插口选项TCP_NOPUSH。如果运行该程序的主机不支持T/TCP，则对setsockopt函数的调用将返回ENOPROTOOPT，程序将转移到前面的分支，执行常规的TCP调用connect。如果函数要求的第3个变量为非0值，则发出请求后还会发出结束标志。

2. 发出正常的TCP调用

30-40 这些是常规的TCP程序：connect、write和可选的shutdown。

服务器的main函数如图B-3所示，几乎没有改变。

server.c

```
1  #include    "cliserv.h"

2  int
3  main(int argc, char *argv[])
4  {                               /* T/TCP or TCP server */
5      struct sockaddr_in serv, cli;
6      char    request[REQUEST], reply[REPLY];
7      int     listenfd, sockfd, n, clilen;

8      if (argc != 2)
9          err_quit("usage: server <port#>");

10     if ((listenfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");

12     memset(&serv, 0, sizeof(serv));
```

图B-3 服务器的main 函数

```
13      serv.sin_family = AF_INET;
14      serv.sin_addr.s_addr = htonl(INADDR_ANY);
15      serv.sin_port = htons(atoi(argv[1]));

16      if (bind(listenfd, (SA) &serv, sizeof(serv)) < 0)
17          err_sys("bind error");

18      if (listen(listenfd, SOMAXCONN) < 0)
19          err_sys("listen error");

20      for (;;) {
21          clilen = sizeof(cli);
22          if ((sockfd = accept(listenfd, (SA) &cli, &clilen)) < 0)
23              err_sys("accept error");

24          if ((n = read_stream(sockfd, request, REQUEST)) < 0)
25              err_sys("read error");

26          /* process "n" bytes of request[] and create reply[] ... */

27  #ifndef MSG_EOF
28  #define MSG_EOF 0               /* send() with flags=0 identical to write() */
29  #endif

30          if (send(sockfd, reply, REPLY, MSG_EOF) != REPLY)
31              err_sys("send error");

32          close(sockfd);
33      }
34  }
```

server.c

图B-3 (续)

27-31 唯一的修改是这里总是调用send(图1-7中调用了write)，但如果主机不支持T/TCP，就让第4个变量的值为0。即使编译时主机是支持T/TCP的，到运行时也可能主机并不支持T/TCP(因此运行时的内核不一定能够理解编译时的MSG_EOF值)，因此，在伯克利版内核中的sosend并不会对它所不理解的标志作出反映。

参考文献

所有的RFC都可以通过电子邮件、匿名 FTP或WWW免费得到，从 `http://www.internic.net`开始即可。`ftp://ds.internic.net/rfc`就是一个RFC目录。

标记有“ Internet Draft ”的项目是Internet工程任务组 (IETF)正在进行的工作，通过Internet也可以免费得到，与RFC类似。这些草案在发布6个月以后就算过期，因此部分草案的版本已经在本书出版以后有了变化，有的草案也已经作为 RFC发布。

在本参考书目中，只要作者指定了参考论文或报告的电子文档位置，一定会包括其URL(统一资源定位符)。也给出了每份Internet草案的URL文件名部分，因为文件名中包含了其版本号。Internet草案的主要存储站点是在目录`ftp://ds.internic.net/internet-drafts`。本参考书目中没有给出RFC的URL。

Anklesaria, F., McCahill, M., Lindner, P., Johnson, D., Torrey, D., and Alberti, B. 1993. "The Internet Gopher Protocol," RFC 1436, 16 pages(Mar.).

Baker, F., ed. 1995. "Requirements for IP Version 4 Routers," RFC 1812 175pages(June).

有关路由器的的文档是 RFC 1122 [Braden 1989]。这个RFC文档废弃了RFC 1009和RFC 1716。

Barber, S. 1995. "Common NNTP Extensions," Internet Draft(June).

`draft-barber-nntp-imp-01.txt`

Bellovin, S.M. 1989 . "Security Problems in the TCP/IP Protocol Suite," *Computer Communication Review*, vol.19 , no.2, pp.32-48(Apr.).

`ftp://ftp.research.att.com/dist/internet_security/ipext.ps.z`

Bellovin, S. M. 1992 . *A Best-Case Network Performance Model.* Private Communication.

Berners-Lee, T. 1993. "Hypertext Transfer Protocol," Internet Draft, 31 pages(Nov.).

这是一个Internet草案文档，现在已经过期。不过，它是 HTTP第1版协议的最初规格说明。

`draft-ietf-iiir-http-00.txt`

Berners-Lee, T. 1994. "Universal Resource Identifiers in WWW : A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as Used in the World-Wide Web," RFC 1630, 28 pages(June).

`http://www.w3.org/hypertext/www/Addressing/URL/URI_Overview.html`

Berners-Lee, T., and Connolly, D. 1995. "Hypertext Markup Language—2.0," Internet Draft(Aug.).

`draft-ietf-html-spec-05.txt`

Berners-Lee, T., Fielding, R. T., and Nielsen, H. F. 1995. "Hypertext Transfer Protocol—

HTTP/1.0, " Internet Draft, 45 pages(Aug.).

draft-ietf-http-v10-spec-02.ps

Berners-Lee, T.,Masinter,L., and McCahill,M.,eds. 1994. "Uniform Resource Locators(URL)," RFC 1738,25 pages(Dec.).

Braden, R.T., 1985. " Towards a Transport Service for Transaction Processing Applications," RFC 955 ,10 pages(Sept.).

Braden, R. T., ed. 1989. "Requirements for Internet Hosts-Communication Layers," RFC 1122, 116 pages(Oct.).

这是有关对主机要求的RFC的前一半，这一半覆盖了链路层、 IP、TCP和UDP。

Braden, R.T. 1992a. "TIME-WALL Assassination Hazards in TCP," RFC 1337,11 pages(May.).

Braden, R.T. 1992b. "Extending TCP for Transactions-Concepts," RFC 1379, 38 pages(Nov.).

Braden, R.T. 1993. "TCP Extensions for High Performance: An Update," Internet Draft, 10 pages (June).

这是对RFC 1323[Jacobson, Braden, and Borman 1992]的更新。

http:// www.noao.edu/~rstevens/tcplw-extensions.txt

Braden, R.T. 1994. "T/TCP-TCP Extensions for Transactions, Functional Specification," RFC 1644, 38 pages(July).

Brakmo, L.S., and Peterson, L. L., 1994. Performance Problems in BSD4.4 TCP.

ftp://cs.arizona.edu/xkernel/papers/tcp_problems.ps

Braun, H-W., and Claffy, K.C. 1994. " Web Traffic Characterization:An Assessment of the Impact of Caching Documents from NCSA's Web Server," *Proceedings of the Second World Wide Web Conference' 94 : Mosaic and the Web*, pp.1007-1027(Oct.), Chicago, Ill.

http://www.ncsa.uiuc. edu/SDG/IT94/Proceedings/DDay/claffy/main. html

Cheriton, D.P. 1988."VMTP:Versatile Message transaction protocol," RFC 1045, 123 pages(Feb.).

Cunha, C.R., Bestavros, A., and Crovella, M.E. 1995. "Characteristics of WWW Client-based Traces, "BU-CS-95-010, Computer Science Department, Boston University(July).

ftp://cs-ftp.bu.edu/techreports/95-010-www-client-traces.ps.Z

Fielding, R.T. 1995. "Relative Uniform Resource Locators," RFC 1808, 16 pages(June).

Floyd, S., Jacobson, V., McCanne, S., Liu, C., -G., and Zhang, L. 1995. "A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing," *Computer Communication Review*, vol. 25, no.4 , pp.342-356(Oct.).

ftp://ftp.ee.lbl.gov/papers/srml.tech.ps.Z

Horton, M., and Adams, R. 1987. "Standard for Interchange of USENET Messages, " RFC 1036, 19 pages(Dec.).

Jacobson, V.1988. "Congestion Avoidance and Control, " *Computer Communication Review*, vol.18, no.4, pp.314-329(Aug.).

这是一篇介绍TCP中的慢启动和拥塞避免算法的经典论文。

ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z

Jacobson, V.1994. "Problems with Arizona's Vegas, " March 14, 1994, end2end-tf mailing list(Mar.).

http://www.noao.edu/~rstevens/vanj.94mar14.txt

Jacobson, V., Braden, R.T., and Borman, D.A. 1992. " TCP Extensions for High Performance, " RFC 1323, 37 pages(May.).

介绍了窗口宽度选项、时间戳选项和 PAWS算法，并且给出了为什么要做所需修改的原因，[braden 1993]更新了该RFC文档。

Jacobson, V., Braden, R. T., and Zhang, L. 1990. " TCP Extensions for High-Speed Paths," RFC 1185, 21 pages(Oct.).

尽管这个RFC文档已经被RFC 1323废弃，但其中的附录介绍了在 TCP中防止过时重复报文段的错误接收问题，值得一读。

Kantor, B., and Lapsley, P.1986. "Network News Transfer Protocol," RFC 977, 27 pages(Feb.).

Kleiman, S.R. 1986. "Vnodes:An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of the 1986 summer USENIX conference*, pp.238-247, Atlanta, Ga.

Kwan, T.T., McGrath, R.E., and Reed, D.A., 1995. *User Access Patterns to NCSA's World Wide Web Server*.

http://www-pablo.cs.uiuc.edu/papers/WWW.ps.Z

Leffler, S.J., McKusick, M.K., Karels, M. J., and Quarterman, J.S. 1989. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, Reading, Mass.

这本书讲述了4.3BSD Tahoe版。它将被[McKusick et al. 1996]所取代。

McKenney, P. E., and Dove, K. F. 1992. "Efficient Demultiplexing of Incoming TCP Packets," *Computer Communication Review*, vol.22, no. 4, pp. 269-279 (Oct.).

Mckusick, M.K., Bostic, K., Karels, M. J ., and Quarterman, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating Syserm*. Addision-Wesley, Reading, Mass.

Miller, T. 1985. "Internet Reliable Transaction Protocol Functional and Interface Specification," RFC 938, 16 pages (Feb.).

Mogul, J. C. 1995a. "Operating Systems Support for Busy Internet Servers," TN-49, Digital Western Research Laboratory(May.).

http://www.research.digital.com/wrl/techreports/abstracts/TN-49.html

Mogul, J. C. 1995b. "The Case for Persistent-Connection HTTP," *Computer Communication Review*, vol. 25, no.4, pp.299-313(Oct.).

http://www.research.digital.com/wrl/techreports/abstracts/95.4.html

Mogul, J. C. 1995c. Private Communication.

Mogul, J. C. 1995d. "Network Behavior of a Busy Web Server and its Clients," WRL Research Report 95/5, Digital Western Research Laboratory(Oct.).

http://www.research.digital.com/wrl/techreports/abstracts/95.5.html

Mogul, J. C., and Deering, S. E. 1990. "Path MTU Discovery," RFC 1191, 19 pages (Apr.).

Olah, A. 1995. Private Communication.

Padmanabhan, V. N. 1995. "Improving World Wide Web Latency," UCB/CSD-95-875, Computer Science Division, University of California, Berkeley(May.).

http://www.cs.berkeley.edu/~padmanab/papers/masters-tr.ps

Partridge, C. 1987. "Implementing the Reliable Data Protocol(RDP)," *Proceedings of the 1987 Summer USENIX Conference*, pp.367-379, Phoenix, Ariz.

Partridge, C. 1990a. "Re:Reliable Datagram Protocol," Message-ID <60240@bbn.BBN.COM>, Usenet, comp.protocols.tcp-ip Newsgroup(Oct.).

Partridge, C. 1990b. "Re: Reliable Datagram ??? Protocols," Message-ID <60340@bbn.BBN.COM>, Usenet, comp.protocols.tcp-ip Newsgroup(Oct.).

Partridge, C., and Hinden, R. 1990. "Version 2 of the Reliable Data Protocol(RDP)," RFC 1151, 4 pages(Apr.).

Paxson, V. 1994a. "Growth Trends in Wide-Area TCP Connections," *IEEE Network*, vol.8,no. 4, pp.8-17(July/Aug.).

ftp://ftp.ee.lbl.gov/papers/WAN-TCP-growth-trends.ps.z

Paxson, V. 1994b. "Empirically-Derived Analytic Models of Wide-Area TCP Connections," *IEEE/ACM Transactions on Networking*, vol.2, no.4, pp.316-336(Aug.).

ftp://ftp.ee.lbl.gov/papers/WAN-TCP-models.ps.z

Paxson, V. 1995a. Private Communication

Paxson, V. 1995b. "Re:Traceroute and TTL," Message-ID <48407@dog.ee.lbl.gov>, Usenet, comp.protocols.tcp-ip Newsgroup(Sept.).

http://www.noao.edu/~rstevens/paxson.95sep29.txt

Postel, J. B., ed. 1981a. "Internet Protocol, "RFC 791, 45pages(Sept.).

Postel, J. B., ed. 1981b. "Transmission Control Protocol,"RFC 793,85 pages(Sept.).

Raggett, D., Lam, J., and Alexander, I.1996. *The Definitive Guide to HTML 3.0: Electronic Publishing on the World Wide Web*. Addison-Wesley, Reading, Mass.

Rago, S.A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, Mass.

Reynolds, J. K., and Postel, J. B. 1994. "Assigned Numbers," RFC 1700, 230 pages(Oct.).

这个RFC是定期更新的，请查看最新的RFC编号。

Rose, M. T. 1993. *The Internet Message: Closing the Book with Electronic Mail*. Prentice-Hall, Upper Saddle River, N. J.

Salus, P. H. 1995. *Casting the Net: From ARPANET to Internet and Beyond*. Addison-Wesley, Reading, Mass.

Shimomura, Tsutomu.1995. "Technical details of the attack described by Markoff in NYT, " Message-ID<3g5gkl$5jl@ariel.sdsc.edu>, Usenet, comp.protocols.tcp-ip Newsgroup(Jan.).

对1994年12月的Internet突破给出了详细的技术分析，并给出了相应的 CERT 咨询报告。

`http://www.noao.edu/~rstevens/shimomura.95jan25.txt`

Spero, S. E., 1994a. *Analysis of HTTP Performance Problems*.

`http://sunsite.unc.edu/mdma-release/http-prob.html`

Spero, S.E., 1994b. *Progress on HTTP-NG*.

`http://www.w3.org/hypertext/www/Protocols/HTTP-NG/http-ng-status.html`

Stein, L. D. 1995. *How to Set Up and Maintain a World Wide Web Site: The Guide for Information Providers*. Addison-Wesley, Reading, Mass.

Stevents, W.R. 1990. *UNIX Network Programming*. Prentice-HALL, Upper Saddle River, N.J.

Stevents, W.R. 1992. *Advanced Programming in the UNIX Environment*. Addision-Wesley, Reading, Mass.

Stevents, W.R. 1994. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, Mass.

该系列书的卷1对Internet协议有比较完整的介绍。

Velten, D., Hinden, R., and Sax, J. 1984. "Reliable Data Protocol," RFC 908, 57 pages (July).

Wright, G. R., and Stevens, W.R. 1995. *TCP/IP Illustrated,Volume 2: The Implementation*. Addison-Wesley, Reading, Mass.

该系列书的卷2研究讨论了4.4BSD-Lite操作系统中的Internet协议实现。

缩 略 语

ACK	TCP首部中的确认(ACKnowledgment)标志
ANSI	American National Standards Institute，美国国家标准协会
API	Application Programming Interface，应用编程接口
ARP	Address Resolution Protocol，地址解析协议
ARPANET	Advanced Research Projects Agency NETwork，远景研究规划局(美国国防部)网
ASCII	American Standard Code for Information Interchange，美国信息交换标准代码
BPF	BSD Packet Filter，BSD分组过滤程序
BSD	Berkeley Software Distribution，伯克利软件发布
CC	Connection Count，连接计数
CERT	Computer Emergency Response Team，计算机应急响应工作队
CR	Carriage Return，回车
DF	TCP首部中的不分段(Don't Fragment)标志
DNS	Domain Name System，域名系统
EOL	End of Option List，选项表结束
FAQ	Frequently Asked Question，经常提出的问题
FIN	TCP首部中的终止(FINish)标志
FTP	File Transfer Protocol，文件传送协议
GIF	Graphics Interchange Format，图形交换格式
HTML	HyperText Markup Language，超文本置标语言
HTTP	HyperText Transfer Protocol，超文本传送协议
ICMP	Internet Control Message Protocol，因特网控制报文协议
IEEE	Institute of Electrical and Electronics Engineers，电气和电子工程师学会(美国)
INN	InterNet News，因特网新闻
INND	InterNet News Daemon，因特网新闻守护程序
IP	Internet Protocol，网际协议
IPC	InterProcess Communication，进程间通信
IRTP	Internet Reliable Transaction Protocol，因特网可靠事务协议
ISN	Initial Sequence Number，初始序号
ISO	International Organization for Standardization，国际标准化组织
ISS	Initial Send Sequence number，初始发送序号
LAN	Local Area Network，局域网
LF	Line Feed，换行
MIME	Multipurpose Internet Mail Extensions，通用因特网邮件扩充

MSL	Maximum Segment Lifetime，报文段最大生存时间
MSS	Maximum Segment Size，报文段最大长度
MTU	Maximum Transmission Unit，最大传输单元
NCSA	National Center for Supercomputing Applications，国家超级计算中心(美国)
NFS	Network File System，网络文件系统
NNRP	Network News Reading Protocol，网络新闻读取协议
NNTP	Network News Transfer Protocol，网络新闻传送协议
NOAO	National Optical Astronomy Observation，国家光学天文观测(美国)
NOP	No Operation，无操作
OSF	Open Software Foundation，开放软件基金
OSI	Open Systems Interconnection，开放系统互连
PAWS	Protection Against Wrapped Sequence number，防止序号重叠
PCB	Protocol Control Block，协议控制快
POSIX	Portable Operation System Interface，可移植操作系统接口
PPP	Point-to-Point Protocol，点对点协议
PSH	TCP首部中的急迫(PuSH)标志
RDP	Reliable Datagram Protocol，可靠数据报
RFC	Request For Comment，是Internet的文档，意思是“请提意见”。
RPC	Remote Procedure Call，远程过程调用
RST	TCP首部中的重建(ReSeT)标志
RTO	Retransmission Time Out，重传超时
RTT	Round-Trip Time，往返时间
SLIP	Serial Line Internet Protocol，串行线路因特网协议
SMTP	Simple Mail Transfer Protocol，简单邮件传送协议
SPT	Server Processing Time，服务器处理时间
SVR4	System V Release 4，系统V版本4
SYN	TCP首部中的序号同步(SYNchronous sequence number)标志
TAO	TCP Accelerated Open，TCP加速打开
TCP	Transmission Control Protocol，传输控制协议
TTL	Time-To-Live，寿命，或生存时间
Telnet	远程登录协议
UDP	User Datagram Protocol，用户数据报协议
URG	TCP首部中的紧急(URGent)指针
URI	Universal Resource Identifier，通用资源标识符
URL	Uniform Resource Locator，统一资源定位符
URN	Uniform Resource Name，统一资源名字
VMTP	Versatile Message Transaction Protocol，通用报文事务协议
WAN	Wide Area Network，广域网
WWW	World Wide Web，万维网