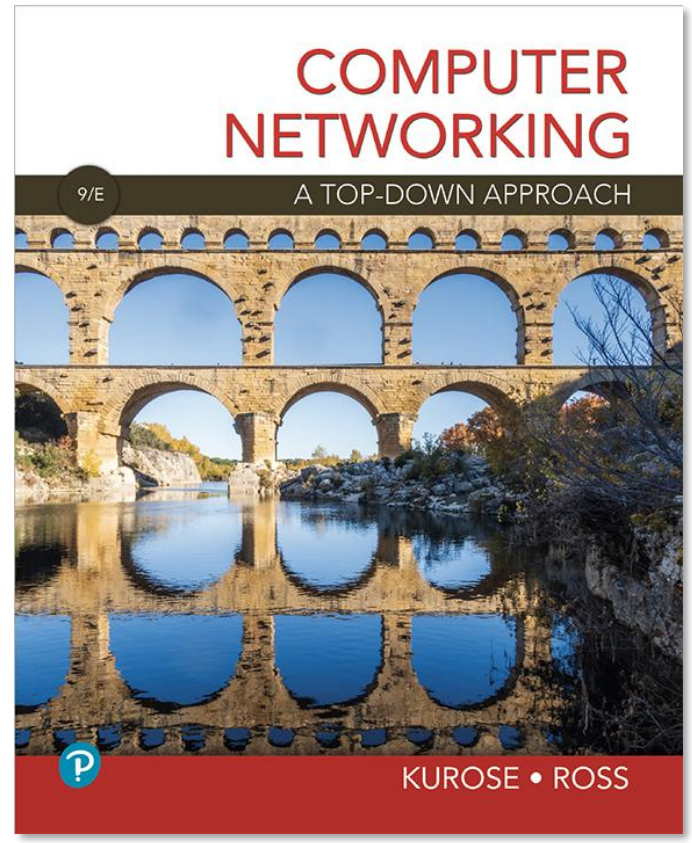
Chapter 3
Transport Layer



Computer Networking: A Top-Down Approach
9th edition
Jim Kurose, Keith Ross
Pearson, 2025

A note on the use of these PowerPoint slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2025
J.F Kurose and K.W. Ross, All Rights Reserved

SPEAKER NOTES

Version 9.0: updated to 9th Edition, June 2025

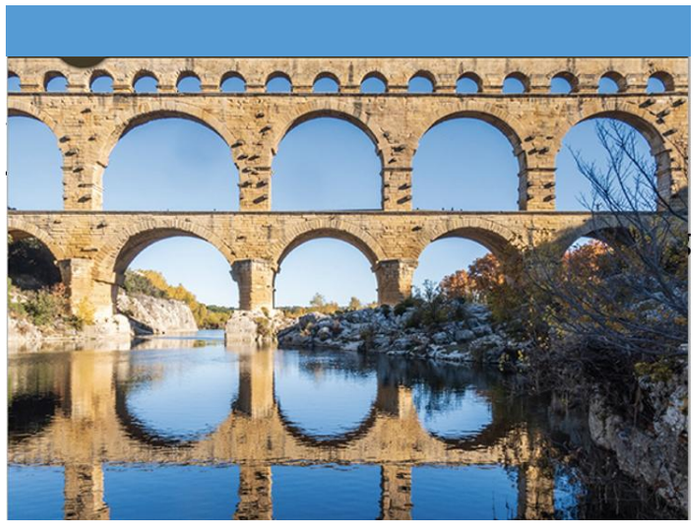
Transport layer: overview

Our goal:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Transport layer: roadmap

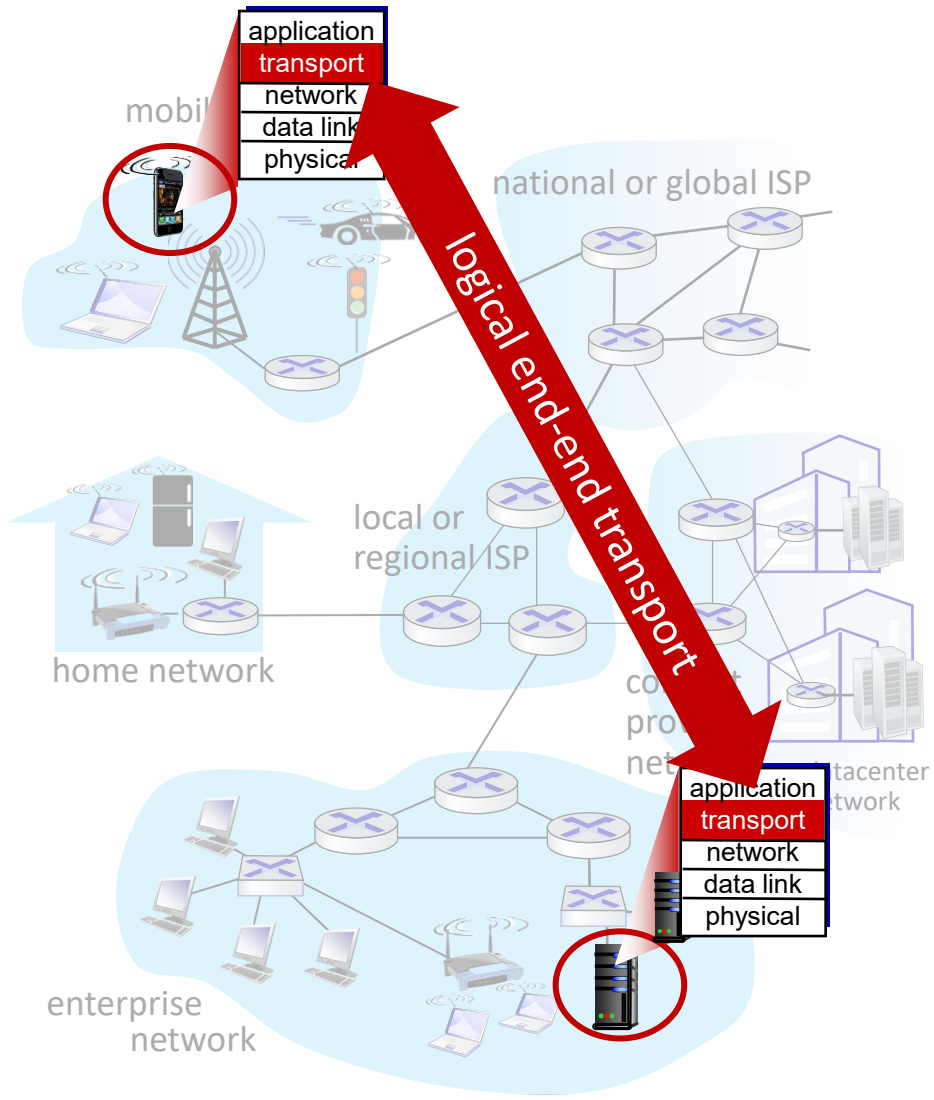
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



Troy University Resources

Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



SPEAKER NOTES

By logical communication , we mean that from an application's perspective, it is as if the hosts running the processes were directly connected; in reality, the hosts may be on opposite sides of the planet, connected via numerous routers and a wide range of link types. Application processes use the logical communication provided by the transport layer to send messages to each other, free from the worry of the details of the physical infrastructure used to carry these messages.

Let's look at each of these three (logical communications, actions,

Transport vs. network layer services and protocols



household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes

Transport vs. network layer services and protocols

- transport layer*:* communication between *processes*
  - relies on, enhances, network layer services
- network layer: communication between *hosts*

household analogy:

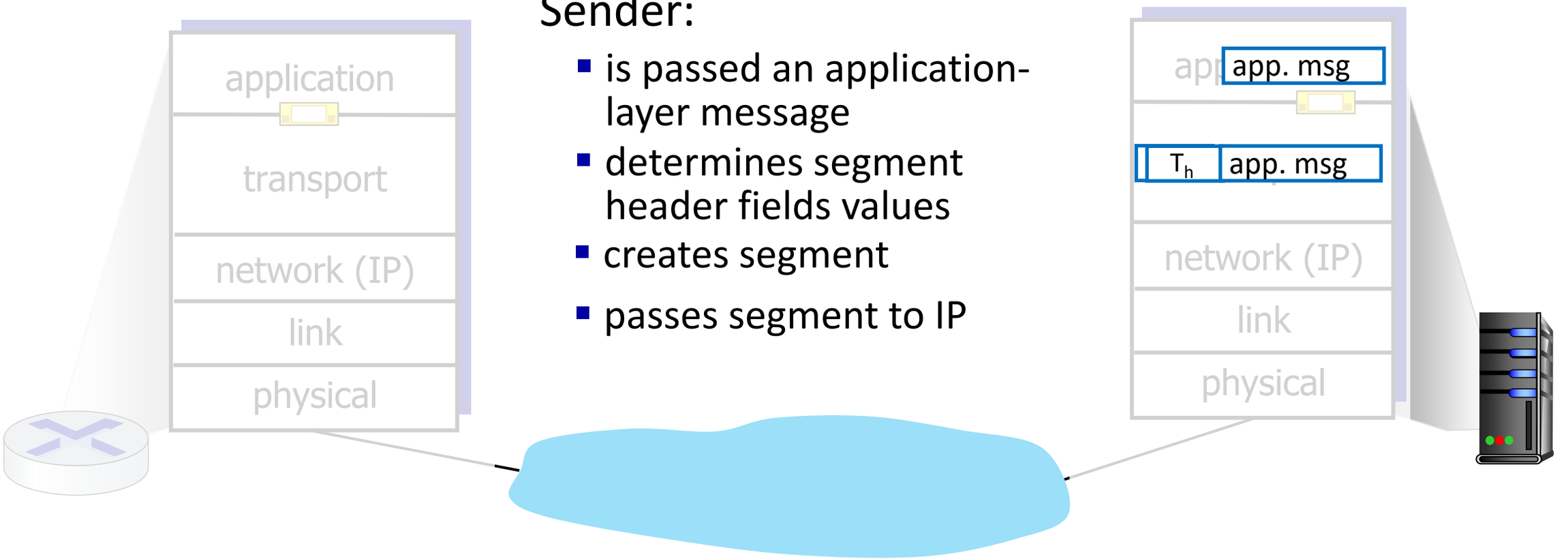
12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Transport Layer Actions

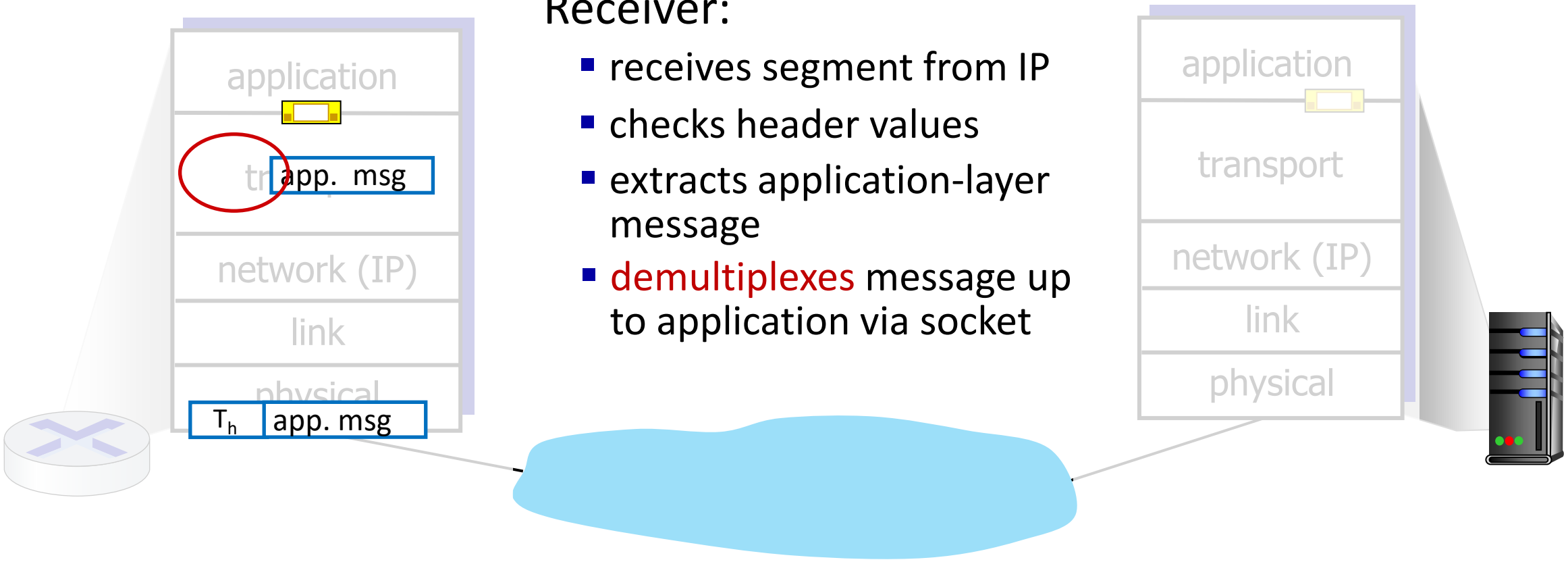
Sender:

- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP



Troy University Resources

Transport Layer Actions

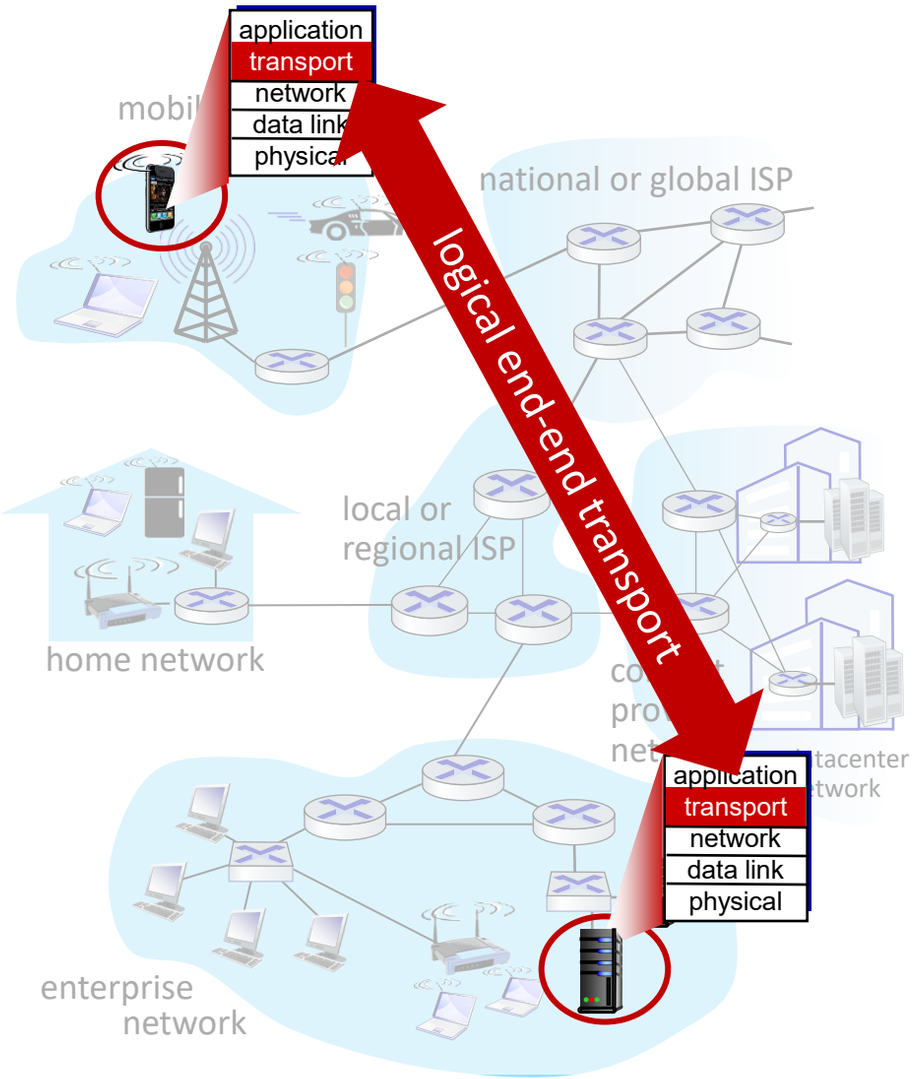


Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket

Two principal Internet transport protocols

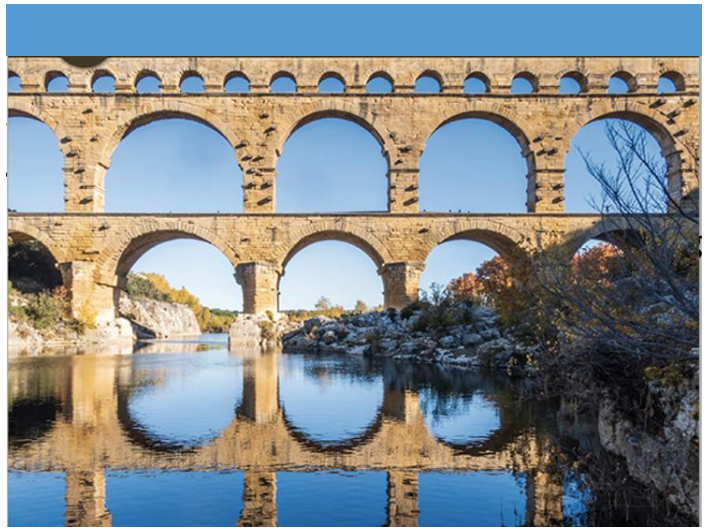
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services *not* available:
  - delay guarantees
  - bandwidth guarantees



Troy University Resources

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



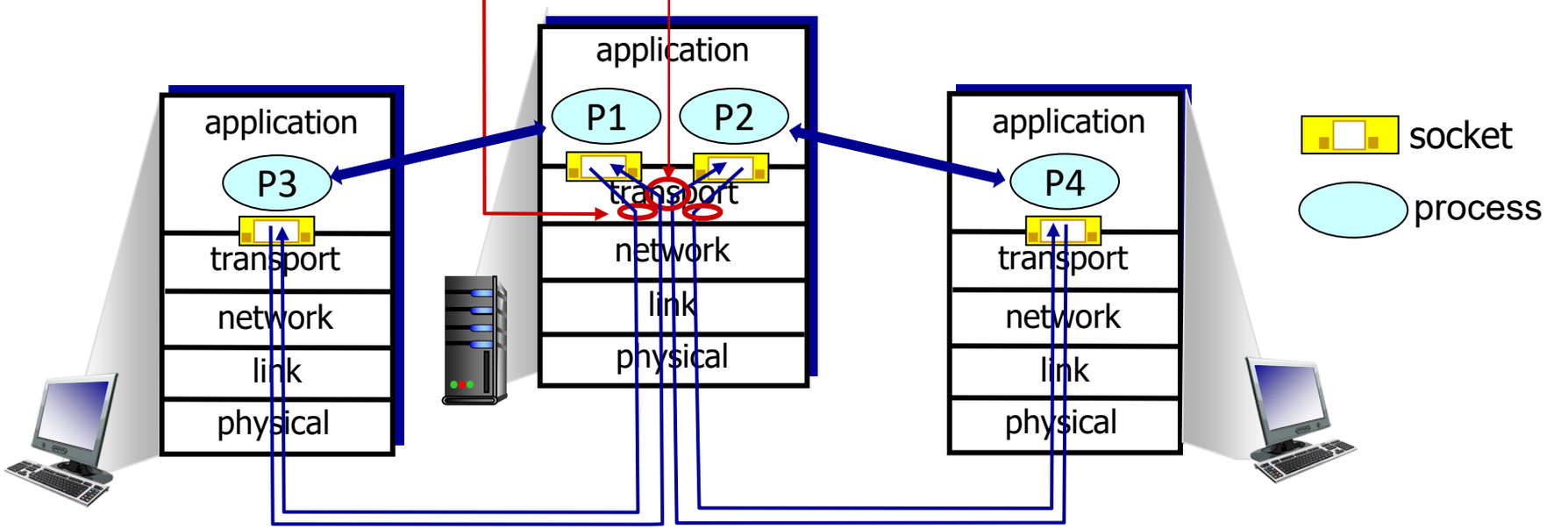
Multiplexing/demultiplexing

multiplexing as sender:

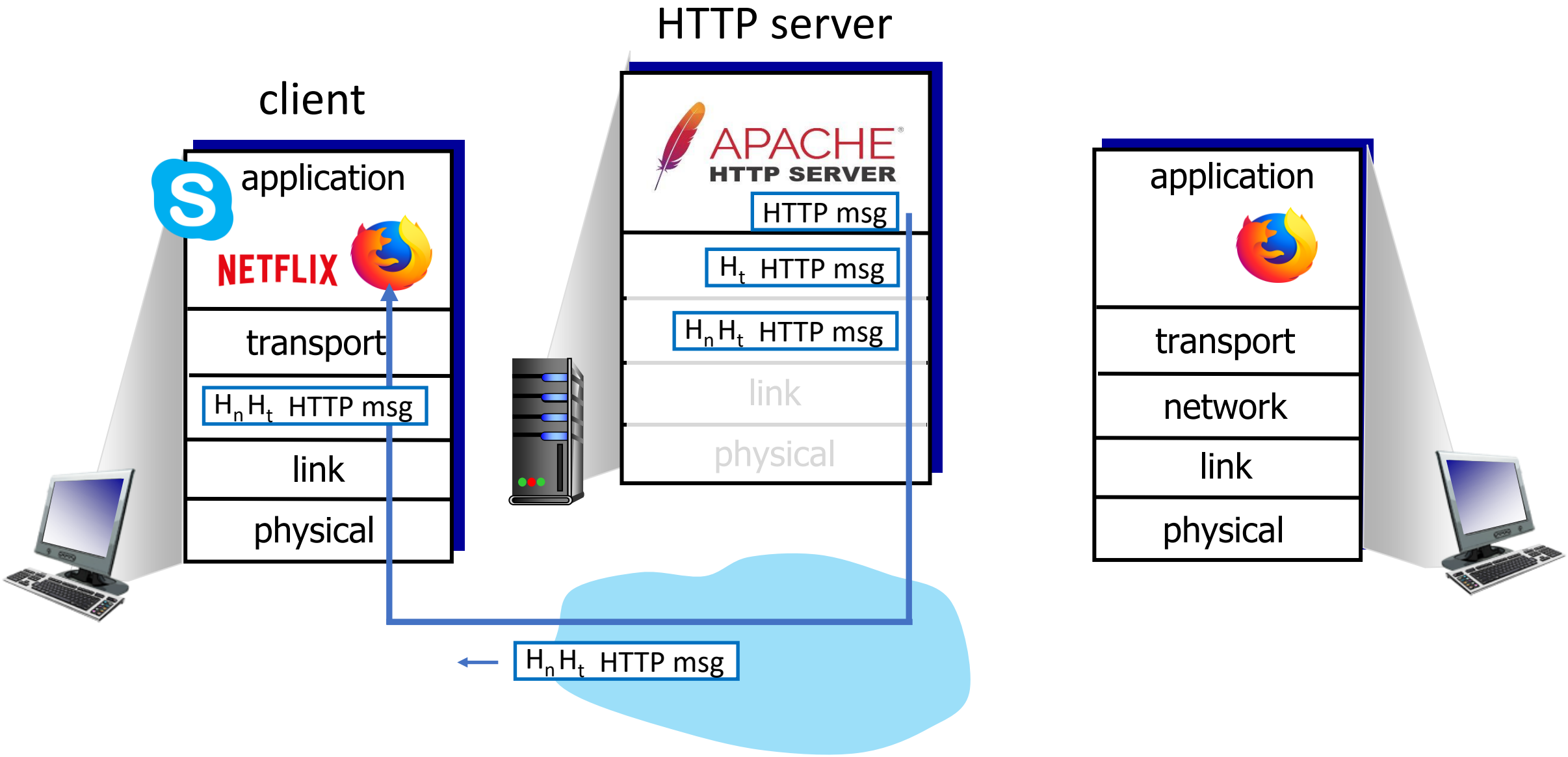
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing as receiver:

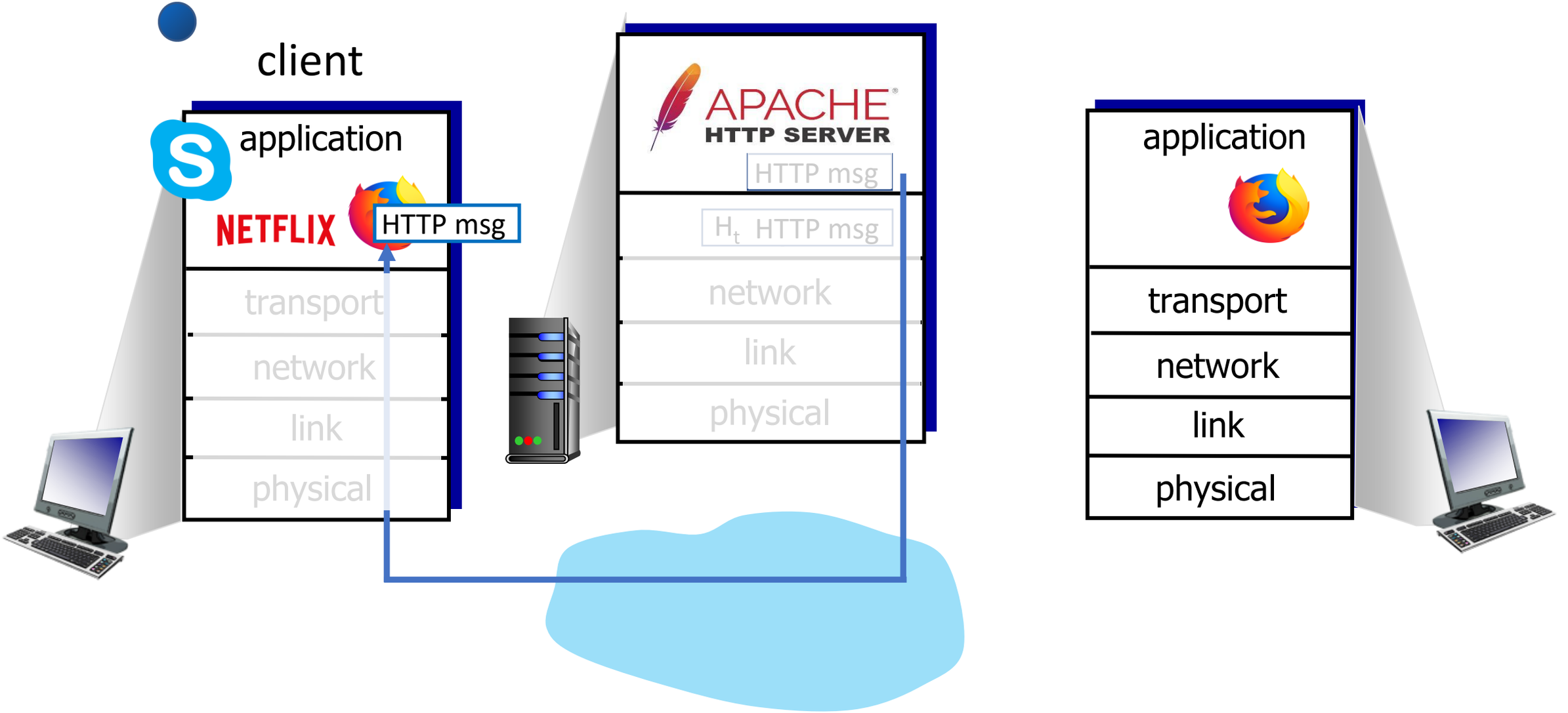
use header info to deliver received segments to correct socket

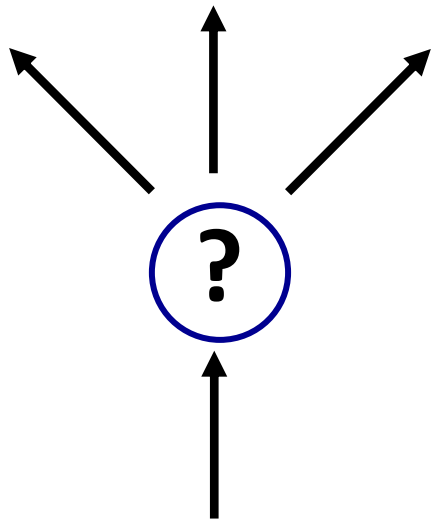


Troy University Resources

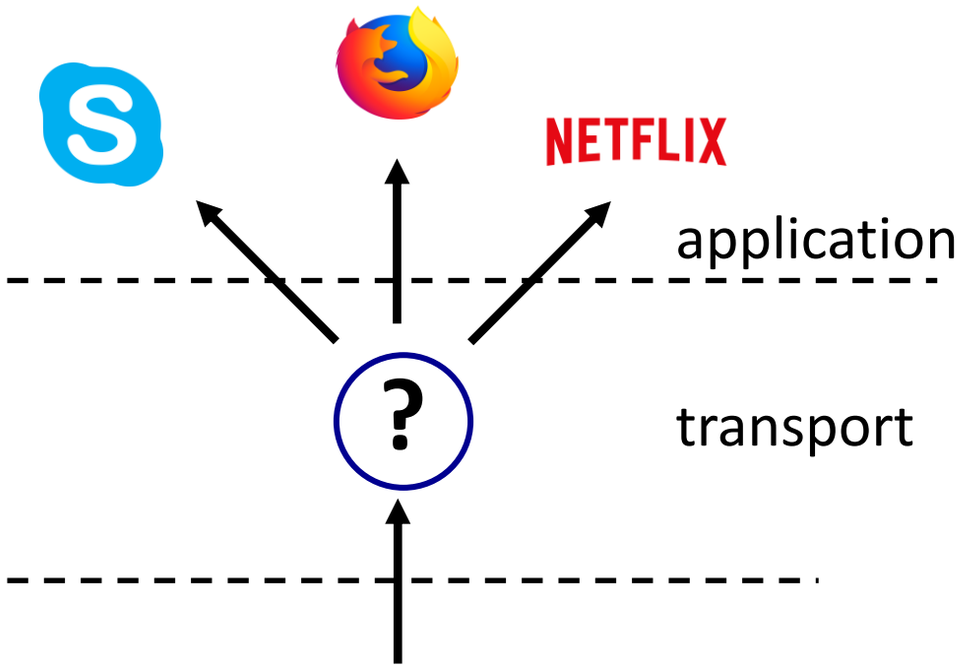

client
application
NETFLIX
transport
$H_n H_t$ HTTP msg
link
physical
HTTP server
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
$H_n H_t$ HTTP msg
link
physical
application
transport
network
link
physical
$H_n H_t$ HTTP msg

Q: how did transport layer know to deliver message to Firefox browser process rather then Netflix process or Skype process?





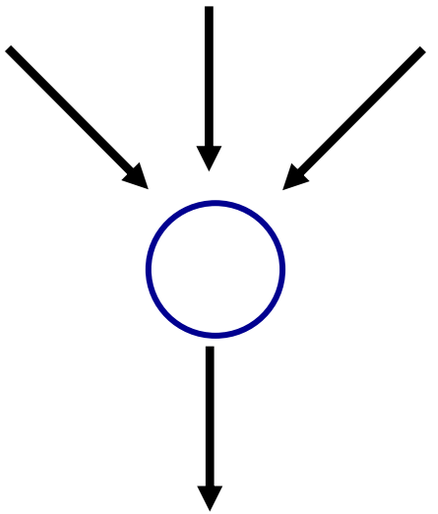
de-multiplexing



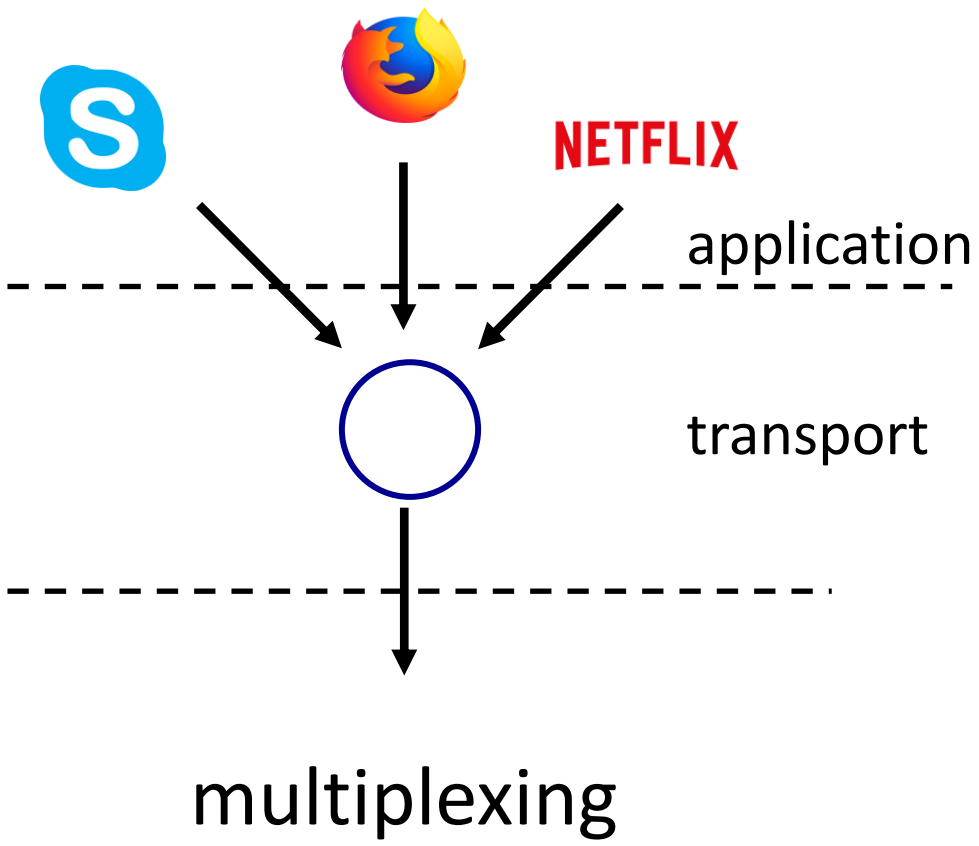
de-multiplexing



Demultiplexing



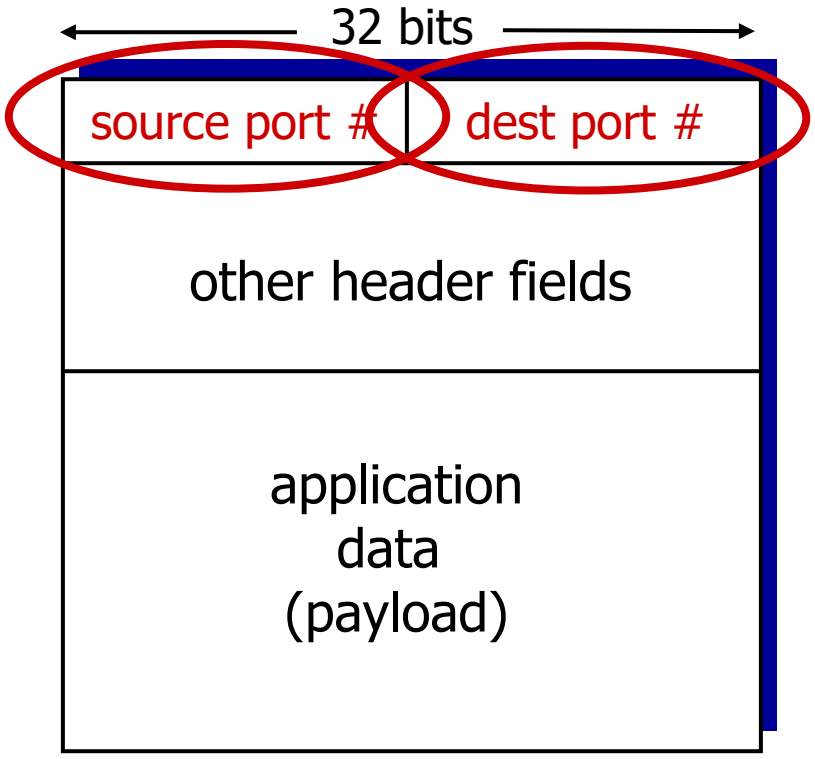
multiplexing


NETFLIX
application
transport
multiplexing


Multiplexing

How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

Recall:

- when creating socket, must specify *host-local* port #:

```
DatagramSocket mySocket1
 = new DatagramSocket(12534);
```

- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #

IP/UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

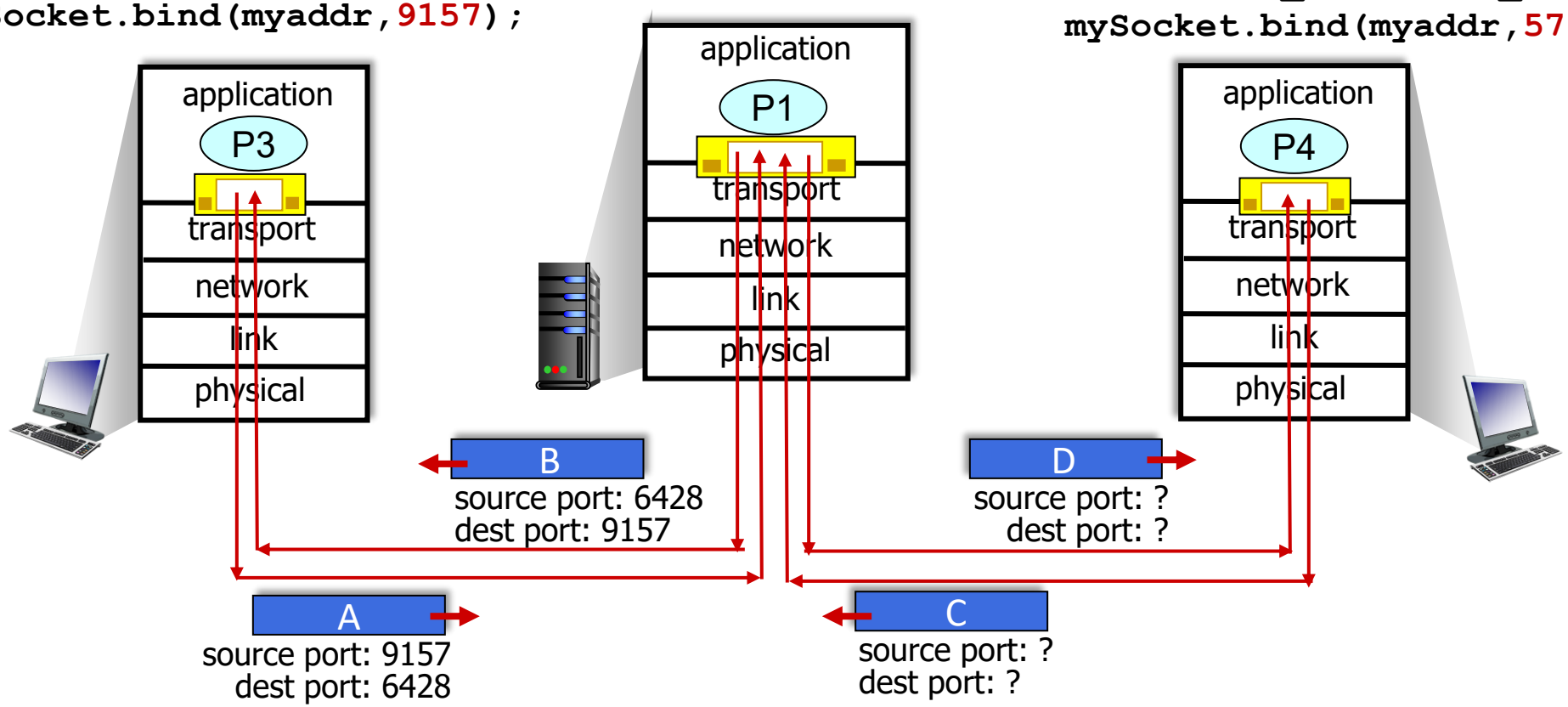
Troy University Resources

Connectionless demultiplexing: an example

```
mySocket =
 socket(AF_INET,SOCK_DGRAM)
mySocket.bind(myaddr,6428);
```

```
mySocket =
 socket(AF_INET,SOCK_DGRAM)
mySocket.bind(myaddr,9157);
```

```
mySocket =
 socket(AF_INET,SOCK_DGRAM)
mySocket.bind(myaddr,5775);
```



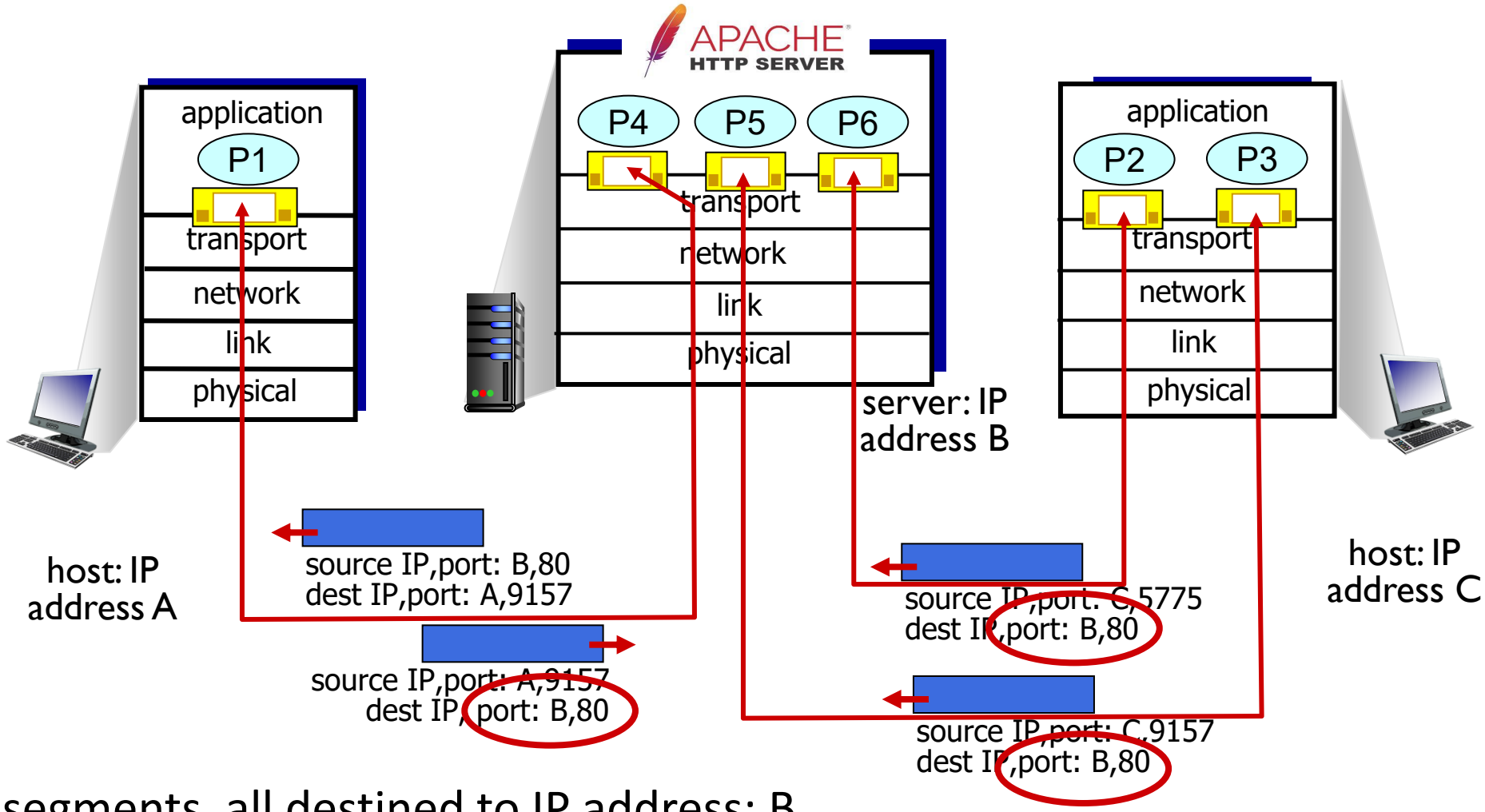
Connection-oriented demultiplexing

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

Troy University Resources

Connection-oriented demultiplexing: example



Three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers

Troy University Resources

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



UDP: User Datagram Protocol

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

Troy University Resources

UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768

J. Postel
ISI
28 August 1980

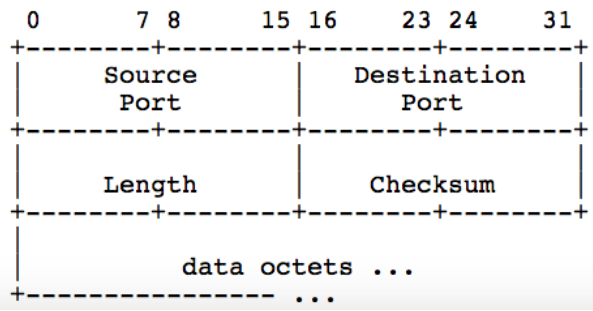
User Datagram Protocol

Introduction

This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

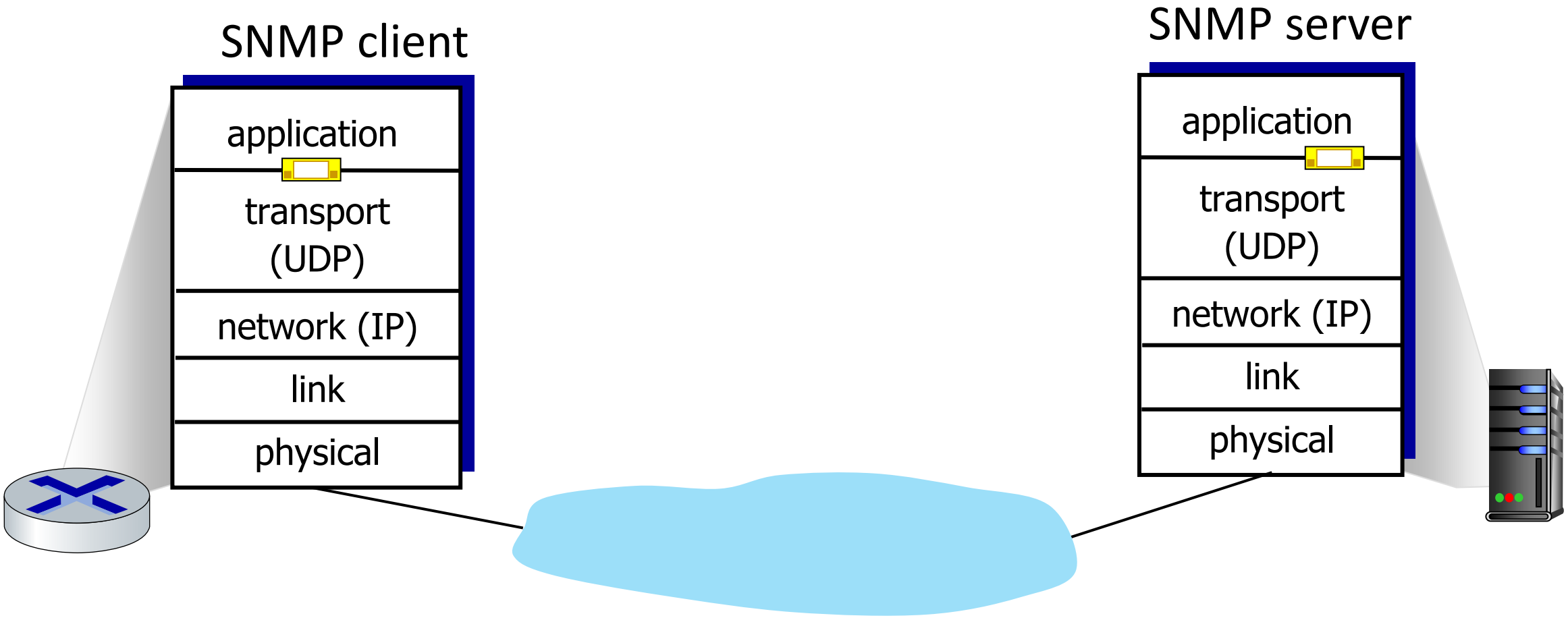
Format



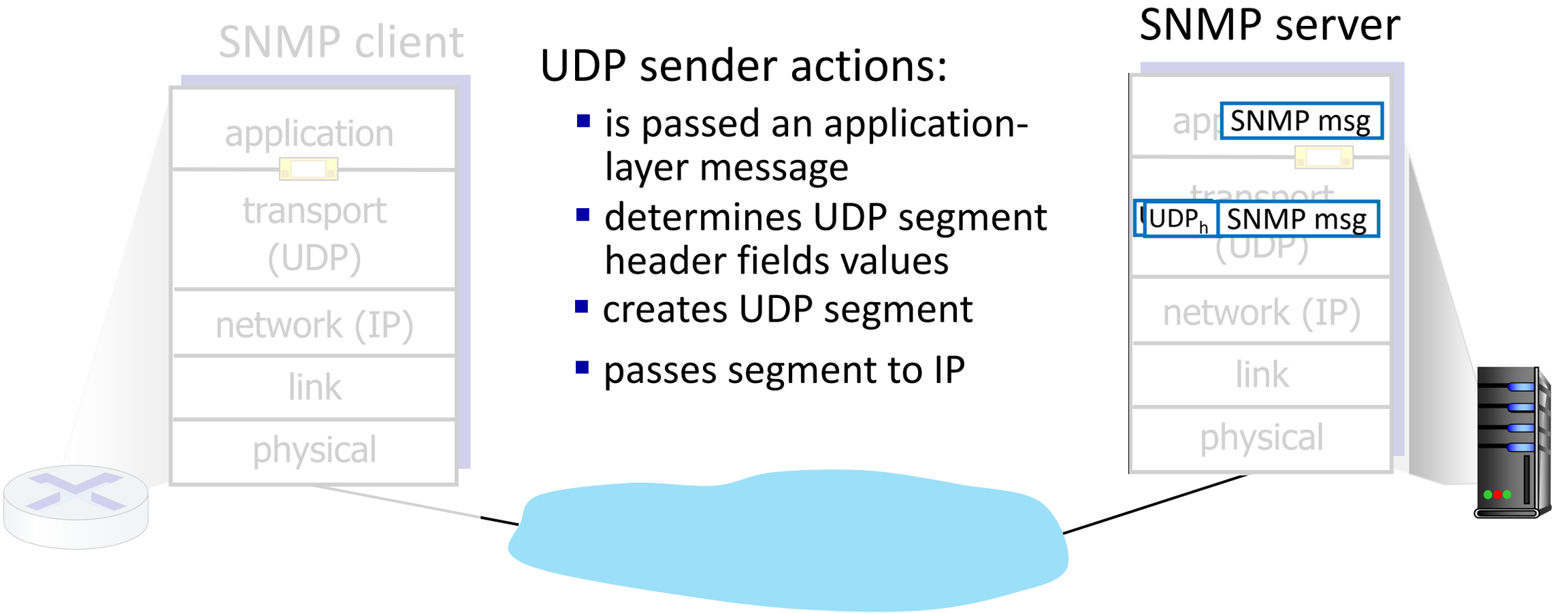
SPEAKER NOTES

read this RFC – it's only 2.5 pages!

UDP: Transport Layer Actions



UDP: Transport Layer Actions

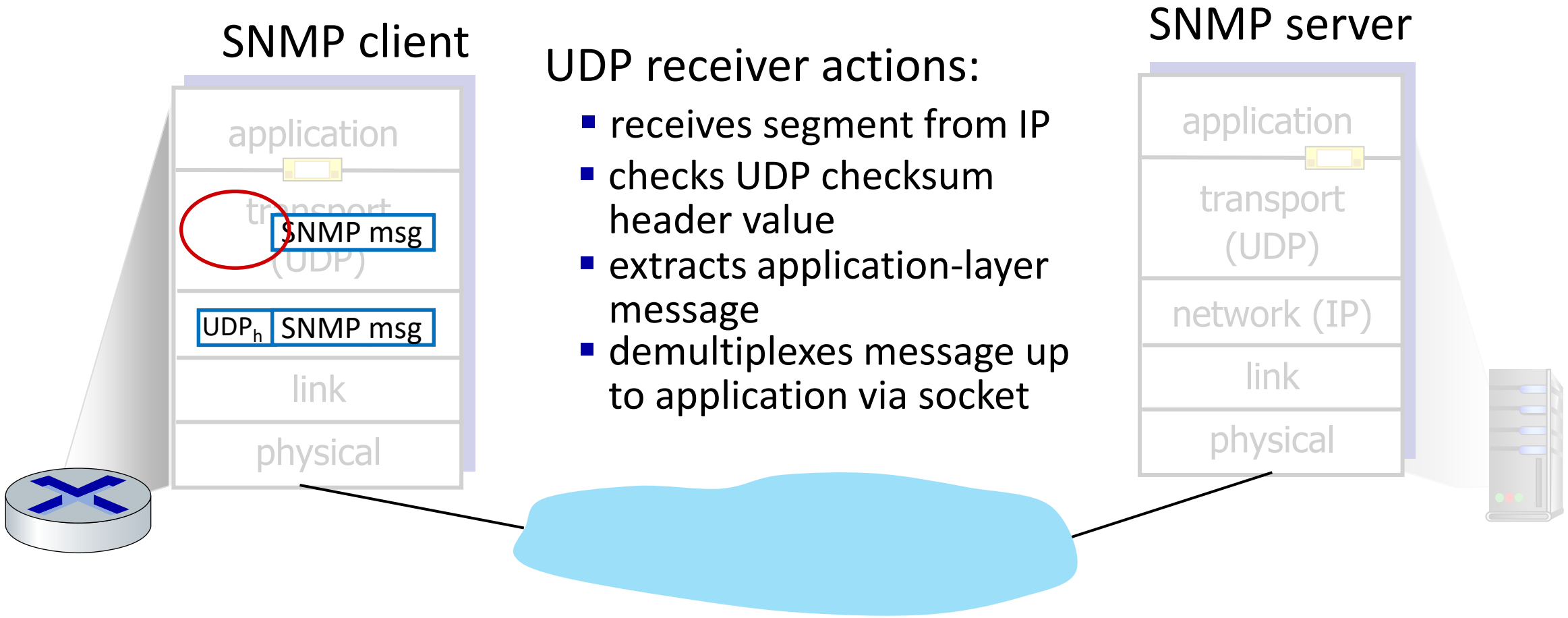


UDP sender actions:

- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

Troy University Resources

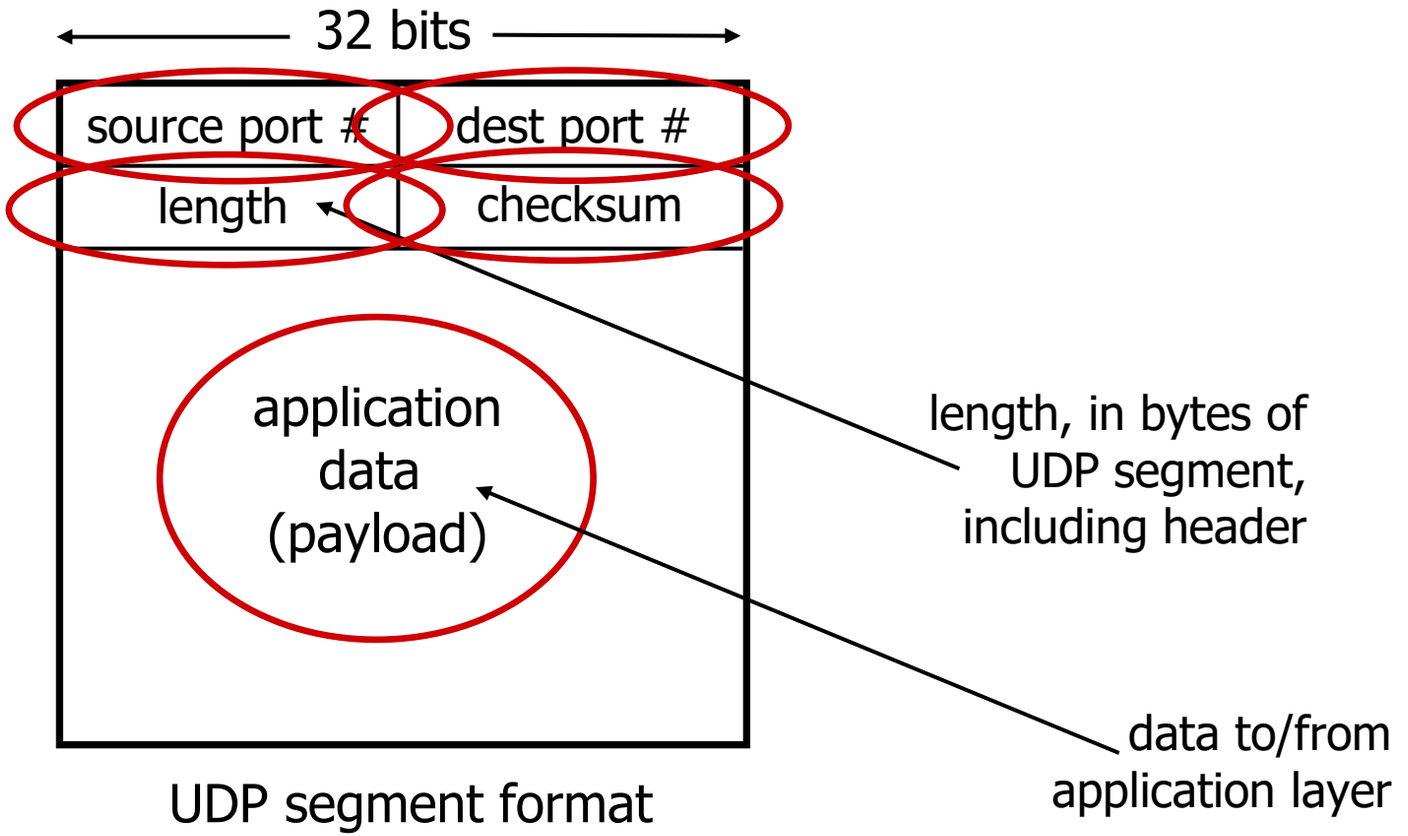
UDP: Transport Layer Actions



UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

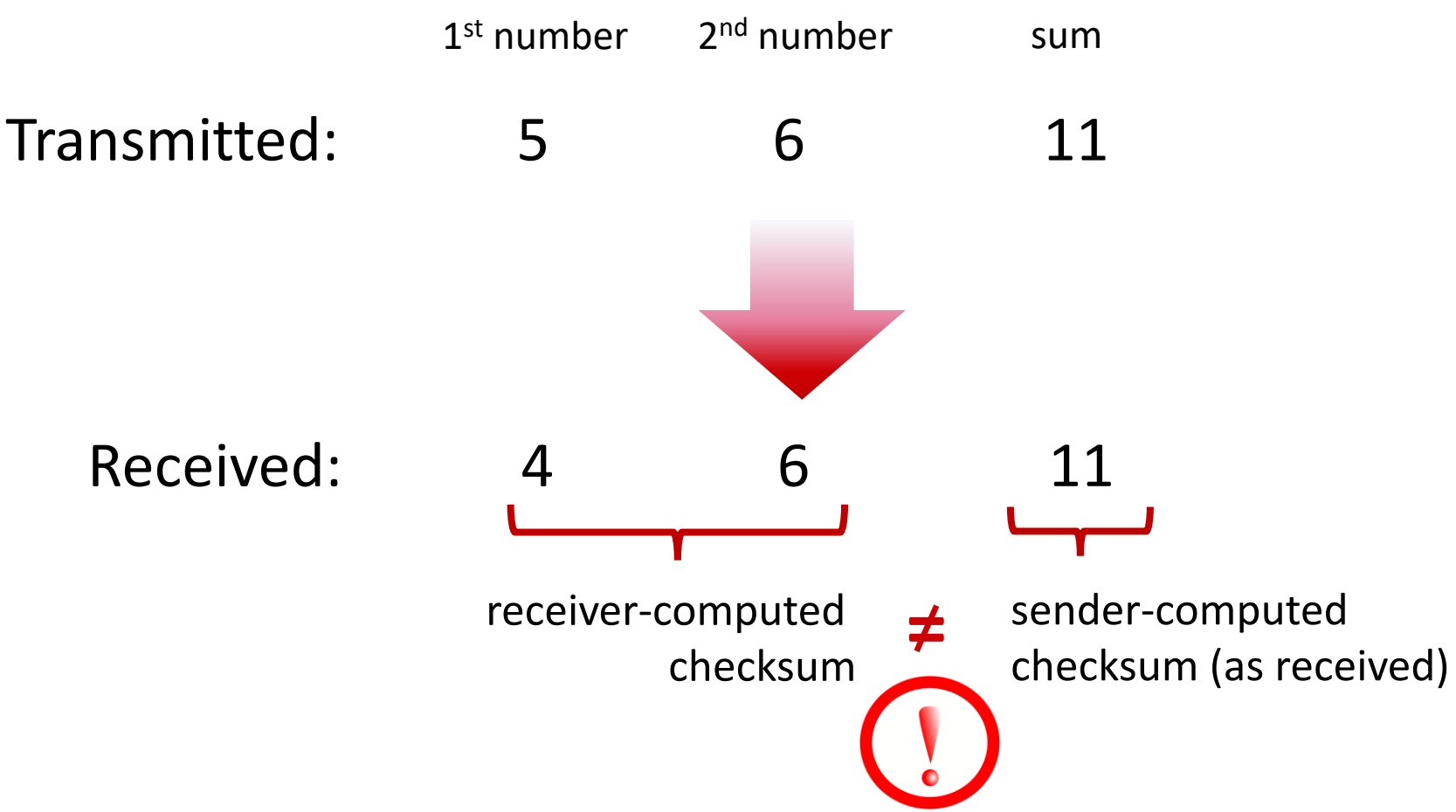
UDP segment header



Troy University Resources

UDP checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment



Internet checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment

sender:

- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

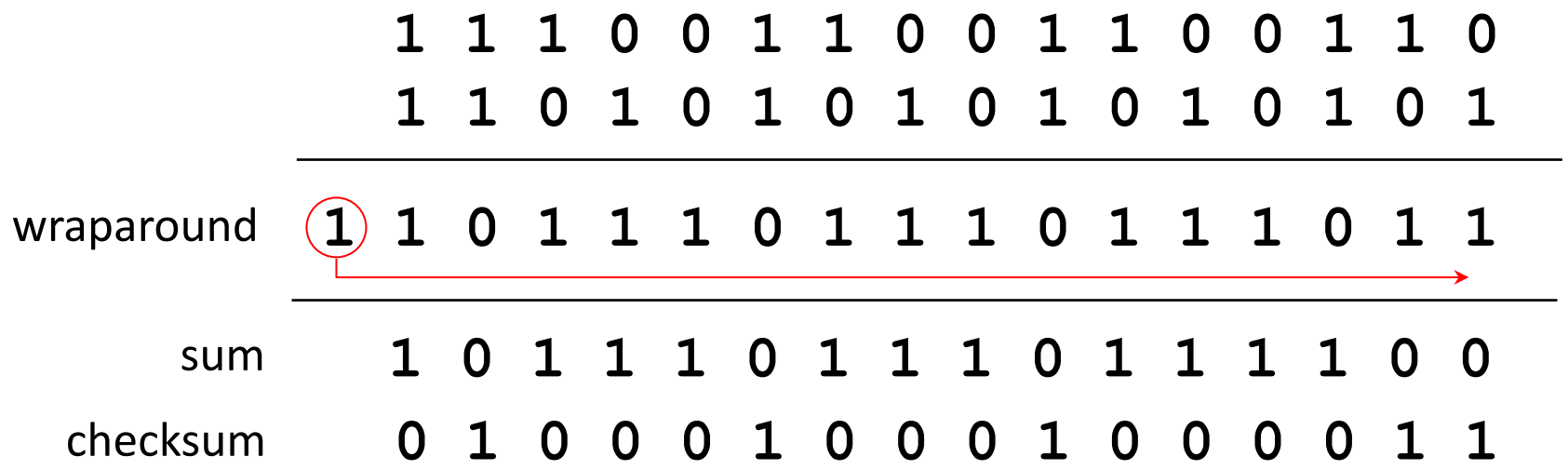
receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - not equal - error detected
  - equal - no error detected. *But maybe errors nonetheless?* More later ….

Troy University Resources

Internet checksum: an example

example: add two 16-bit integers

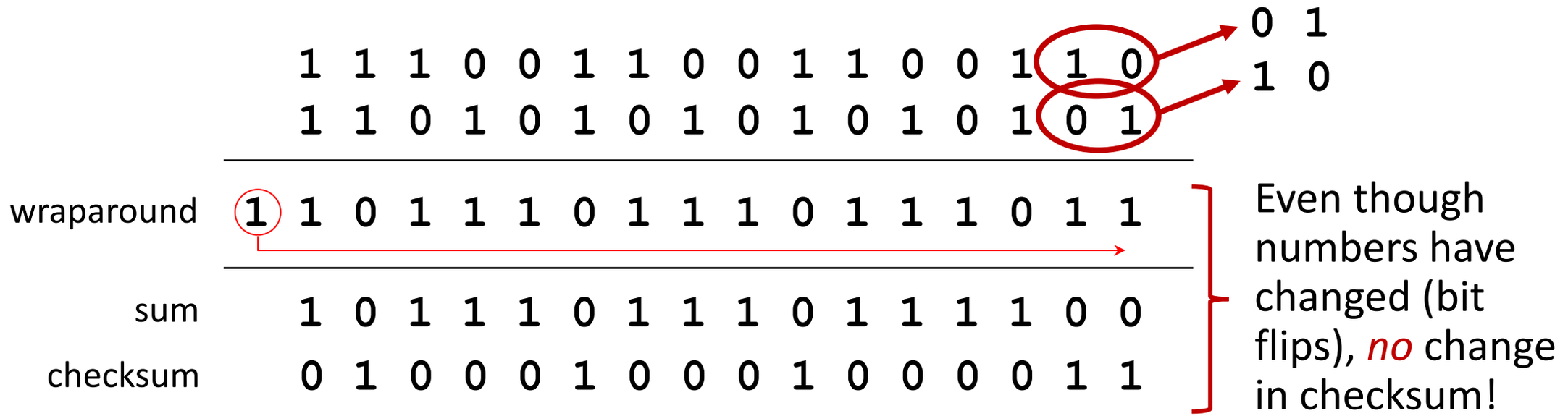


Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Internet checksum: weak protection!

example: add two 16-bit integers



Troy University Resources

Summary: UDP

- “no frills” protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- **Principles of reliable data transfer**
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



Troy University Resources

Principles of reliable data transfer



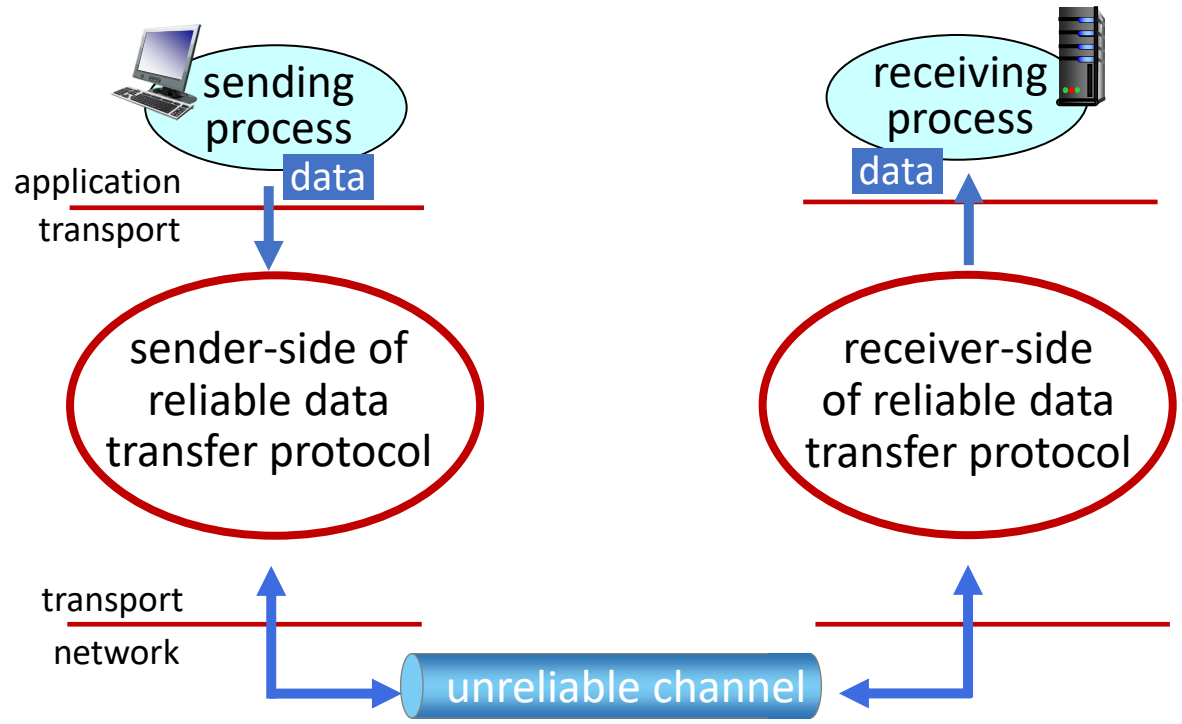
reliable service *abstraction*

SPEAKER NOTES

note arrows through reliable data transfer channel is just one way – reliably send from sender to receiver

Principles of reliable data transfer

reliable service *abstraction*



reliable service *implementation*

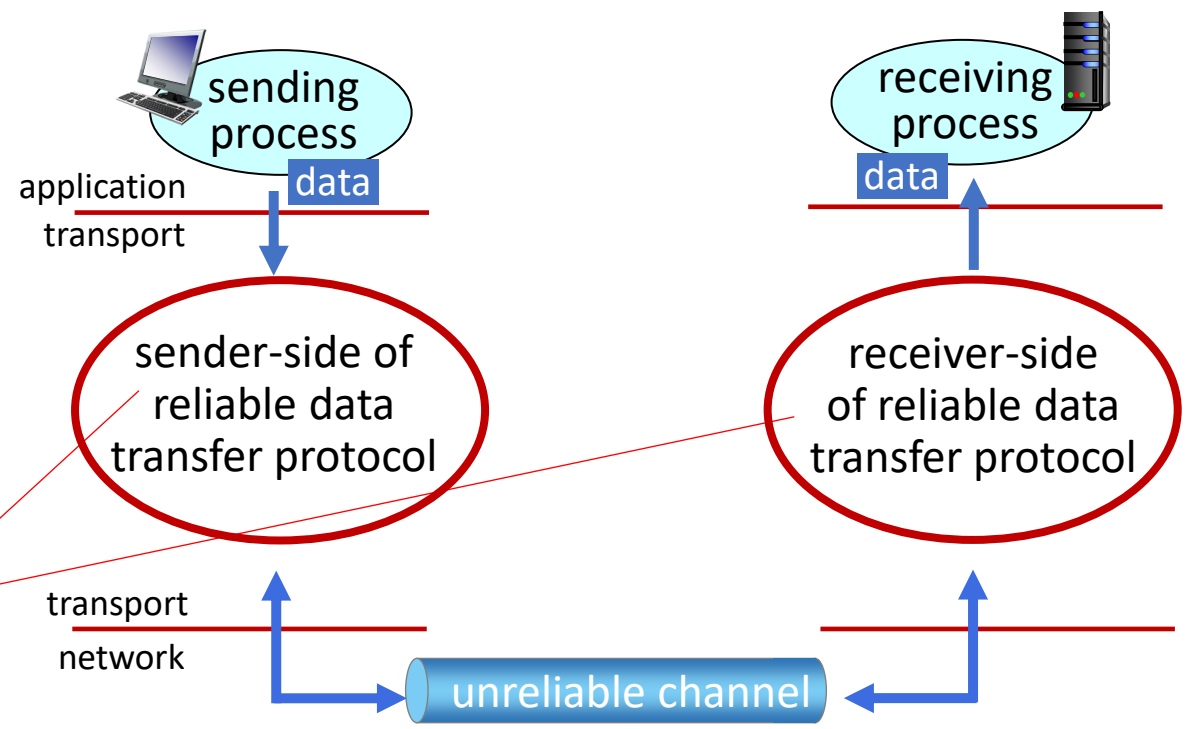
SPEAKER NOTES

Spend some time talking about how its to the sender and receiver side protocol that IMPLEMENTS reliable data transfer

Communication over unreliable channel is TWO-way: sender and receiver will exchange messages back and forth to IMPLEMENT one-way reliable data transfer

Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



reliable service *implementation*

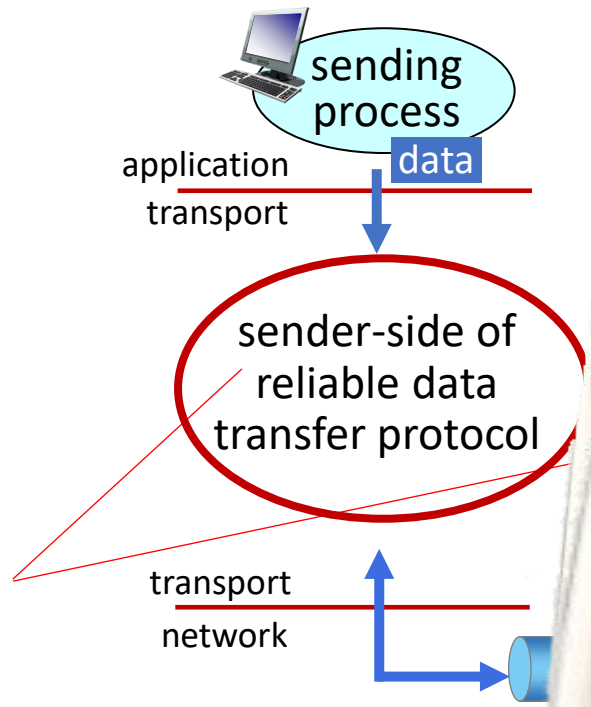
SPEAKER NOTES

So we have a sender side and a receiver side. How much work they'll have to do depends on the IMPAIRMENTS introduced by channel – if the channel is perfect – no problem!

Principles of reliable data transfer

Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- unless communicated via a message



reliable service *implementation*

SPEAKER NOTES

Here's a point of view to keep in mind – it's easy for US to look at sender and receiver together and see what is happening. OH – that message sent was lost.

But think about it say from senders POV How does the sender know if its transmitted message over the unreliable channel got though?? ONLY if receiver somehow signals to the sender that it was received.

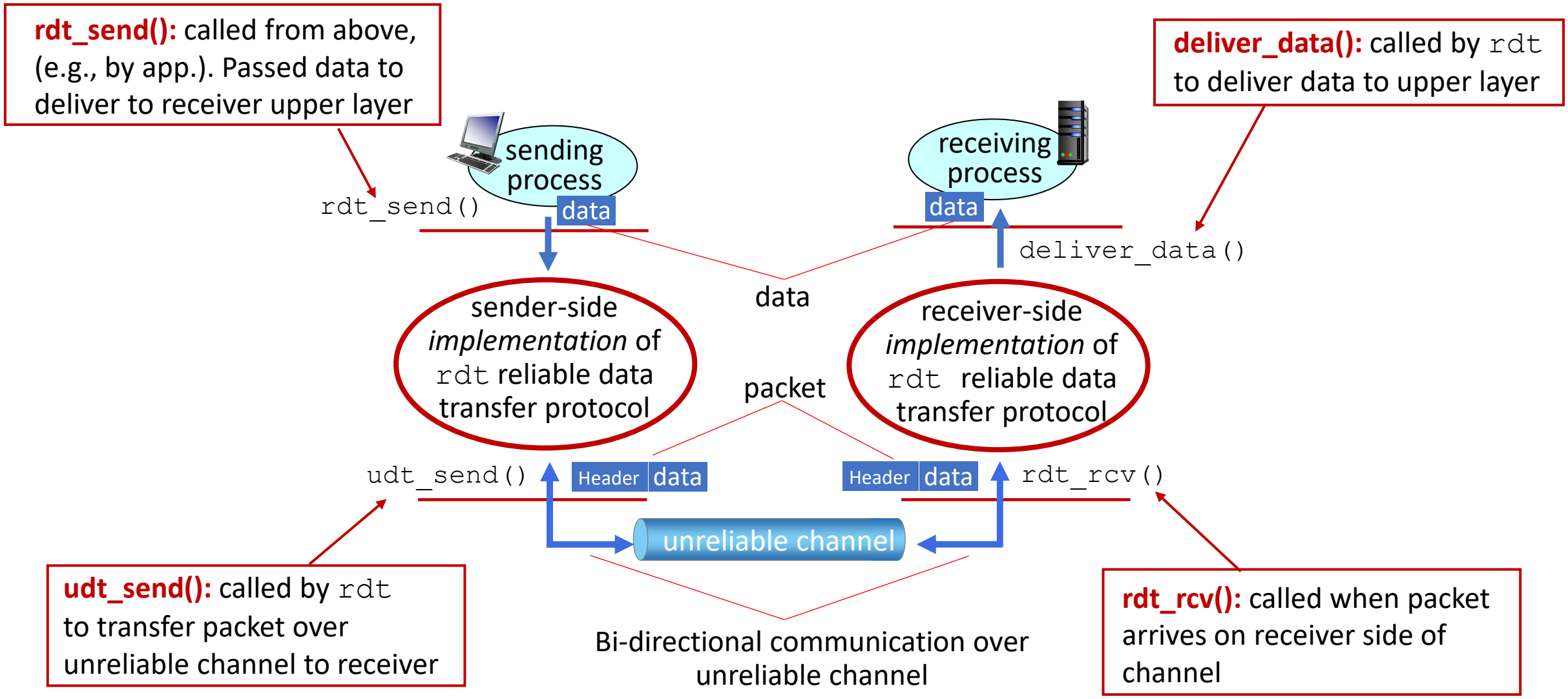
The key point here is that one side does NOT know what is going on at the other side – it's as if there's a curtain between them. Everything they know about the other can ONLY be learned by sending/receiving messages.

Sender process wants to make sure a segment got through. But it can just somehow magically look through curtain to see if receiver got it. It will be up to the receiver to let the sender KNOW that it (the receiver) has correctly received the segment.

How will the sender and receiver do that – that's the PROTOCOL.

Before starting to develop a protocol, let's look more closely at the interface (the API if you will)

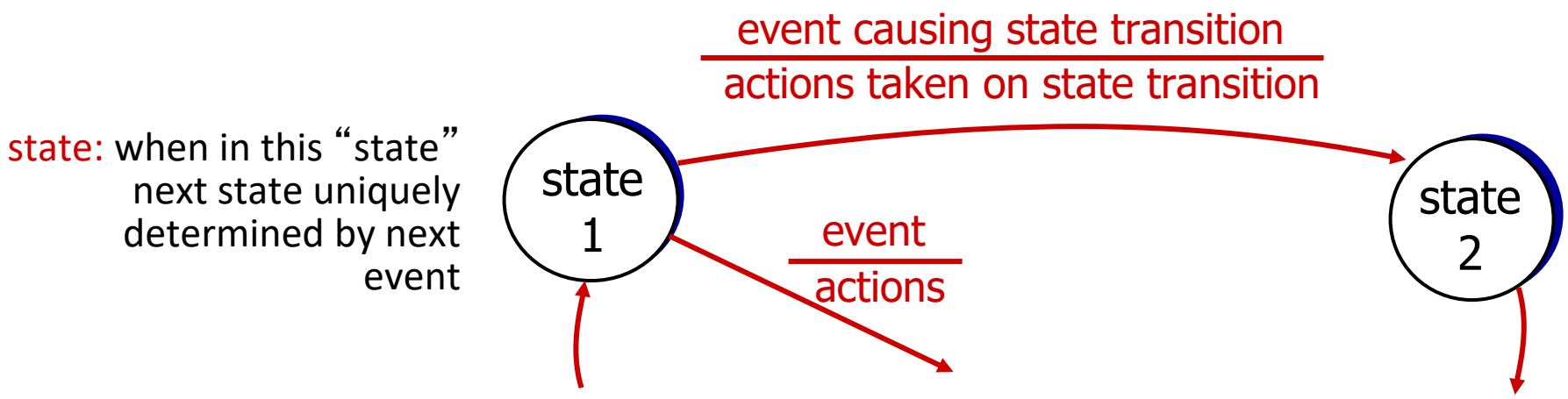
Reliable data transfer protocol (rdt): interfaces



Reliable data transfer: getting started

We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (`rdt`)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



SPEAKER NOTES

SO let's get started in developing our reliable data transfer protocol, which we'll call rdt (need a good acronym for protocol – like HTTP, TCP, UDP, IP)

Bullet points 1 and 2

NOW if we are going to develop a protocol, so we'll need some way to SPECIFY a protocol. How do we do that?

We could write text, but as all know, that's prone to misinterpretation, and might be incomplete. You might write a specification, and then think "oh yeah – I forgot about that case"

What we need is more formal way to specify a protocol. In fact, with a formal specification there may be ways to PROVE PROPERTIES about a specification. But that's an advanced topic we won't get into here. We'll start here by adopting a fairly simple protocol specification technique known as finite state machines (FSM)

And as the name might suggest, a central notion of finite state machines is the notion of STATE
<talk about state>

rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- *separate* FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel





SPEAKER NOTES

We'll start with the simplest case possible - an unreliable channel that is, in fact perfect – no segments are lost, corrupted, duplicated or reordered. The sender just sends and it pops out the other side(perhaps after some delay) perfectly.

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- *the* question: how to recover from errors?

How do humans recover from "errors" during conversation?

Troy University Resources

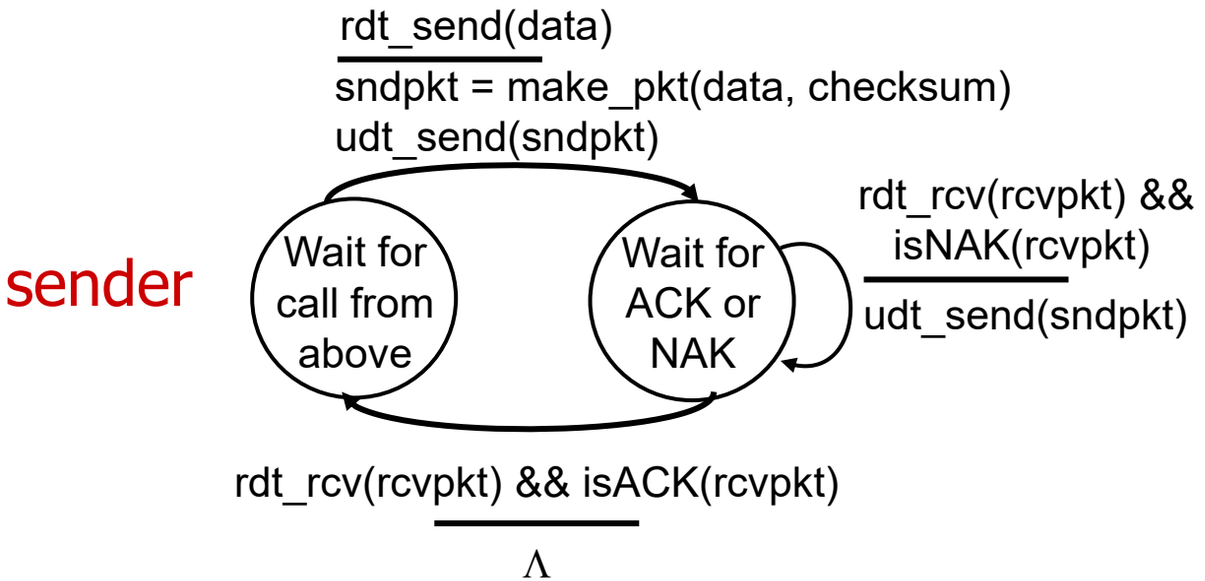
rdt2.0: channel with bit errors

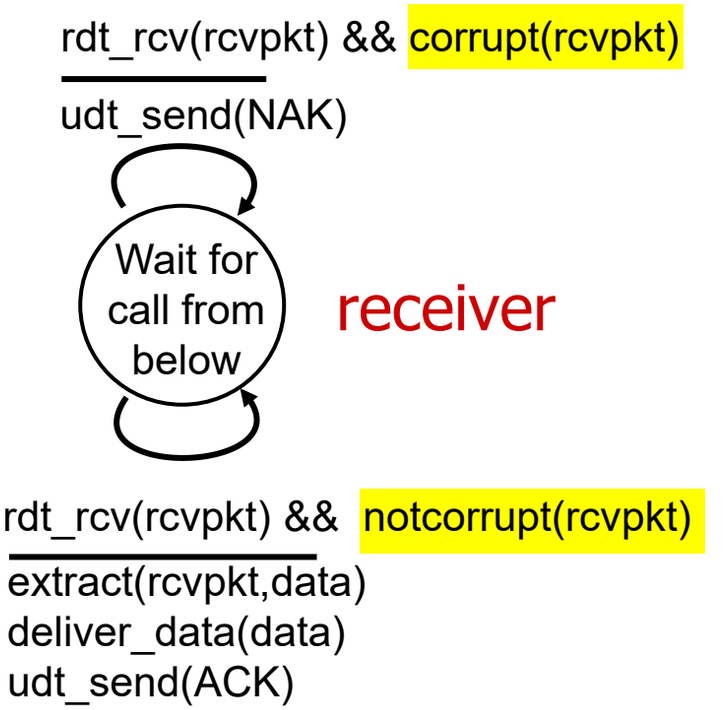
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender *retransmits* pkt on receipt of NAK

stop and wait

sender sends one packet, then waits for receiver response

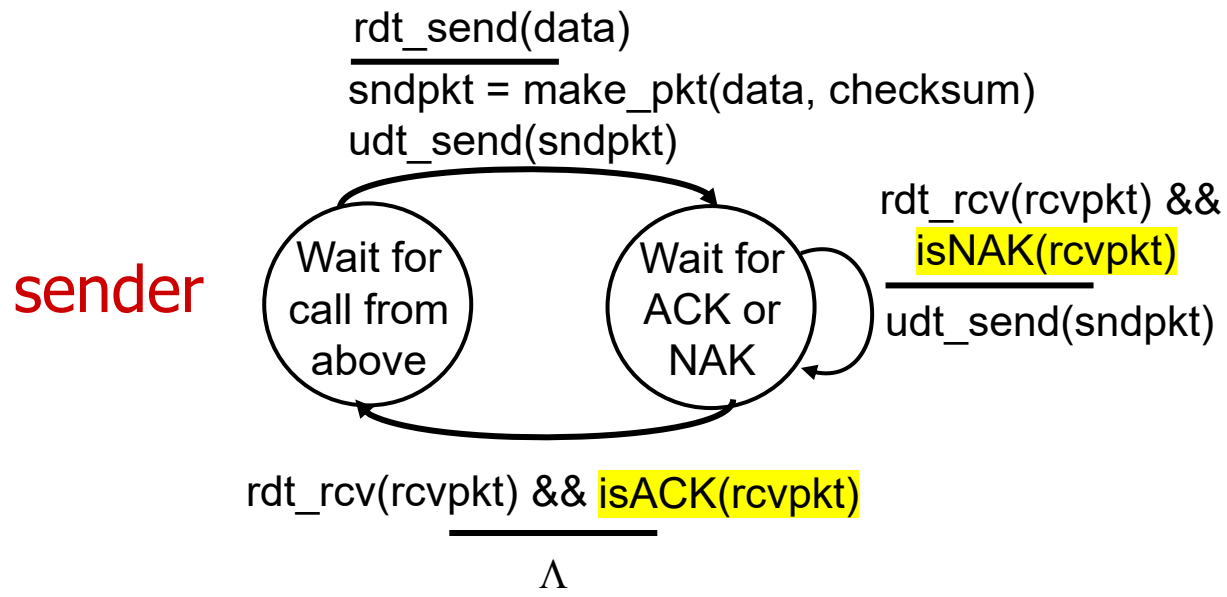
rdt2.0: FSM specifications





Troy University Resources

rdt2.0: FSM specification

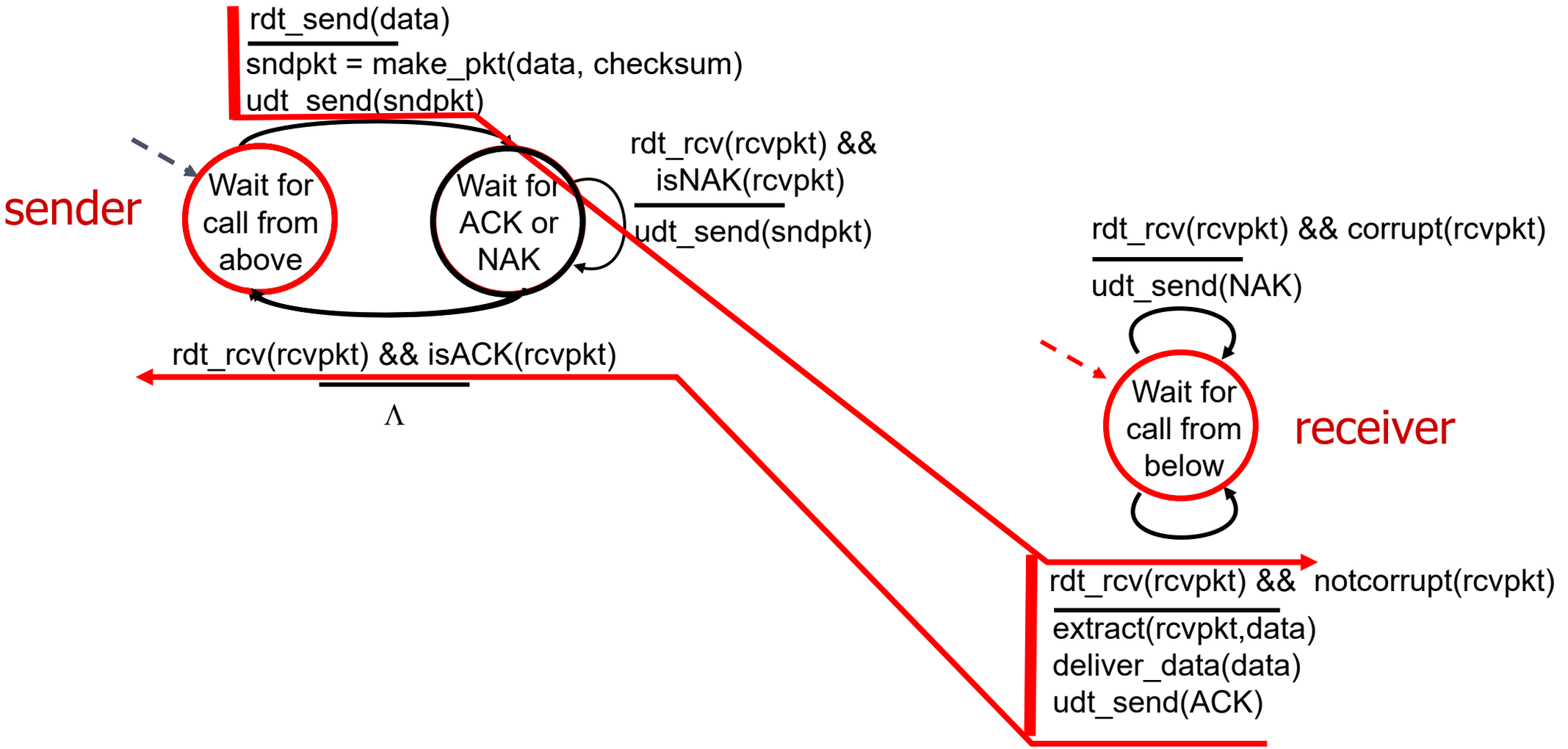


Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

- that's why we need a protocol!

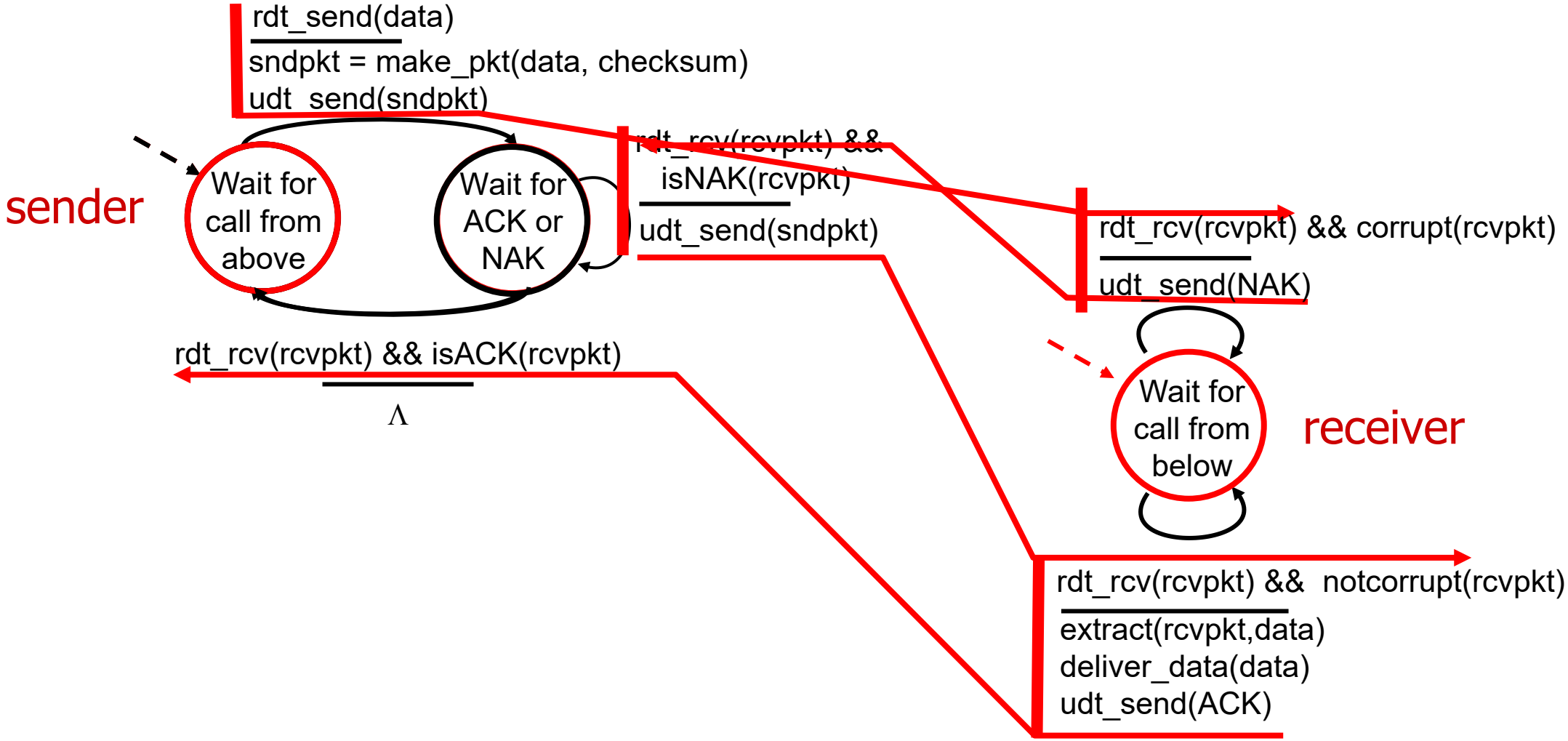


rdt2.0: operation with no errors



Troy University Resources

rdt2.0: corrupted packet scenario



rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

handling duplicates:

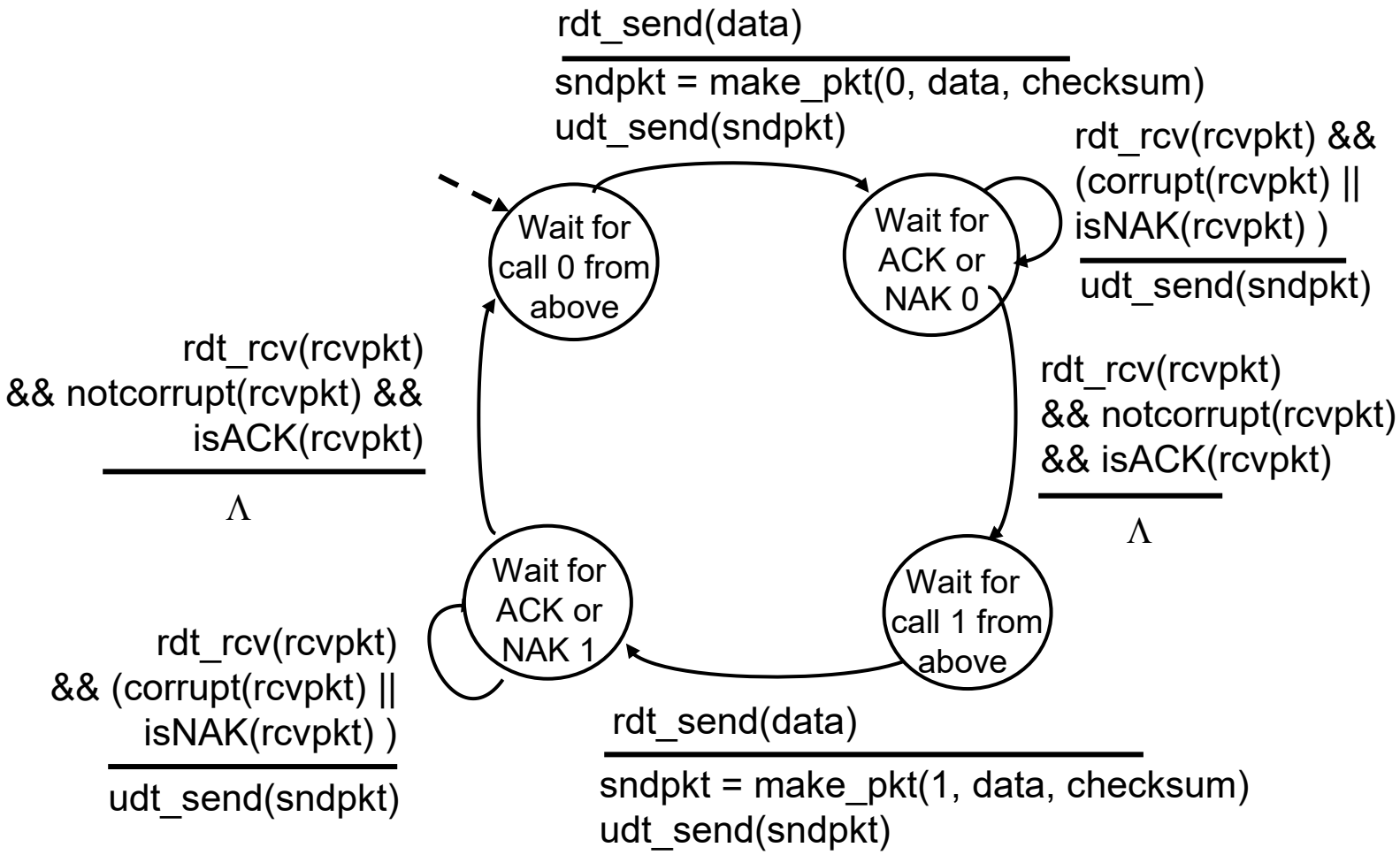
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

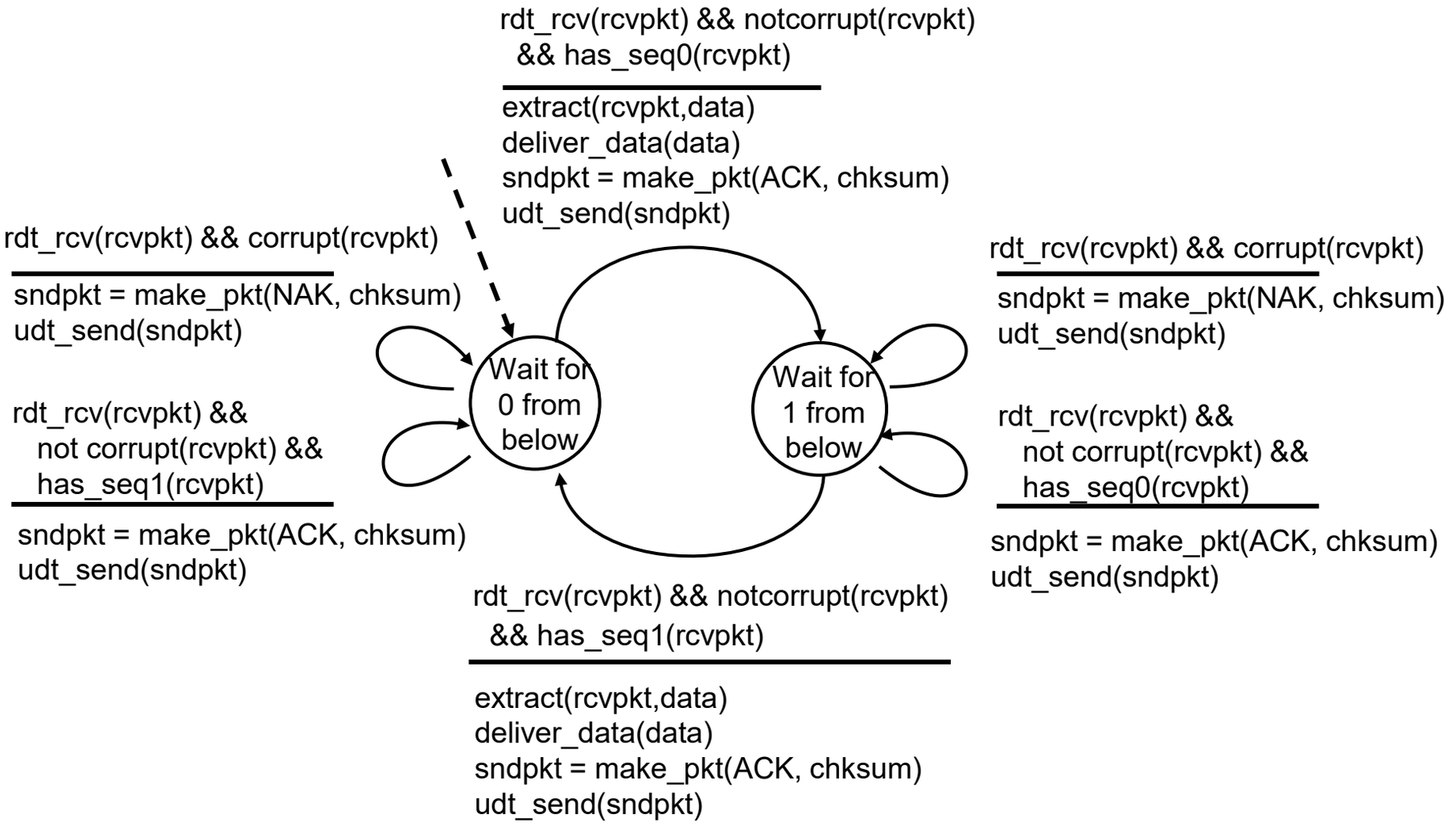
sender sends one packet, then waits for receiver response

Troy University Resources

rdt2.1: sender, handling garbled ACK/NAKs



rdt2.1: receiver, handling garbled ACK/NAKs



Troy University Resources

rdt2.1: discussion

sender:

- seq # added to pkt
- two seq. #s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

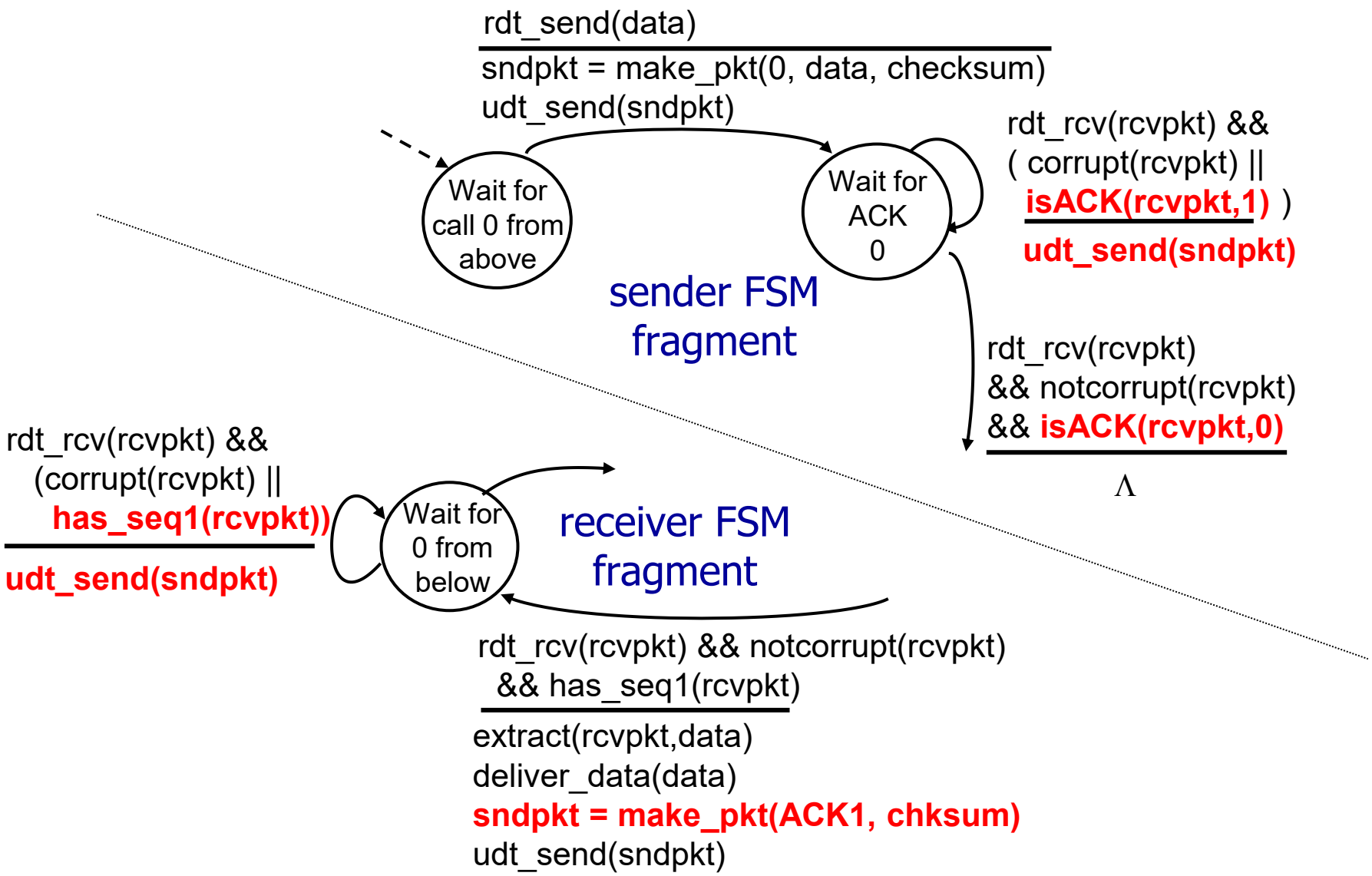
rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

As we will see, TCP uses this approach to be NAK-free

Troy University Resources

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

New channel assumption: underlying channel can also *lose* packets (data, ACKs)

- checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

Q: How do *humans* handle lost sender-to-receiver words in conversation?

Troy University Resources

rdt3.0: channels with errors *and* loss

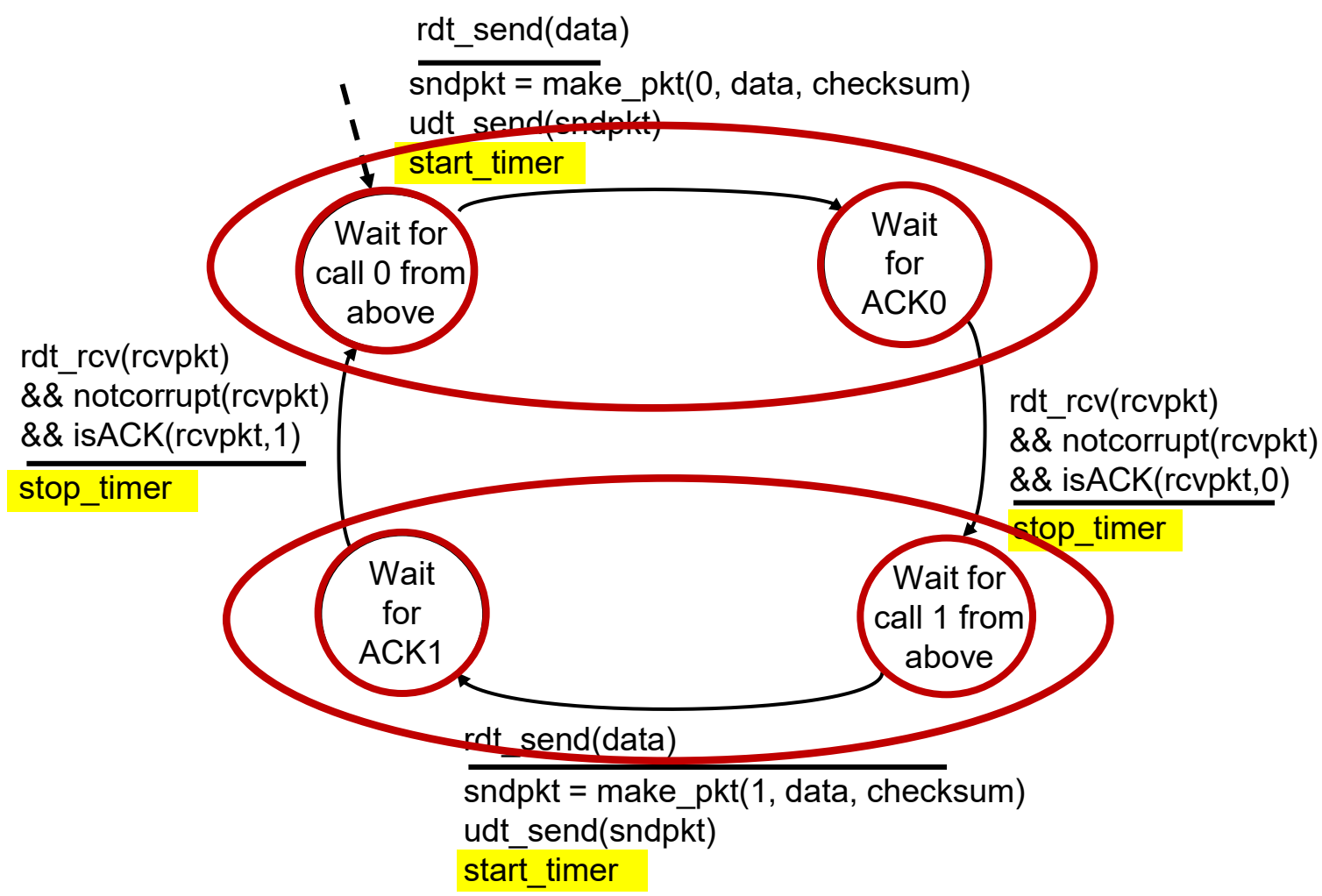
Approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time



timeout

rdt3.0 sender

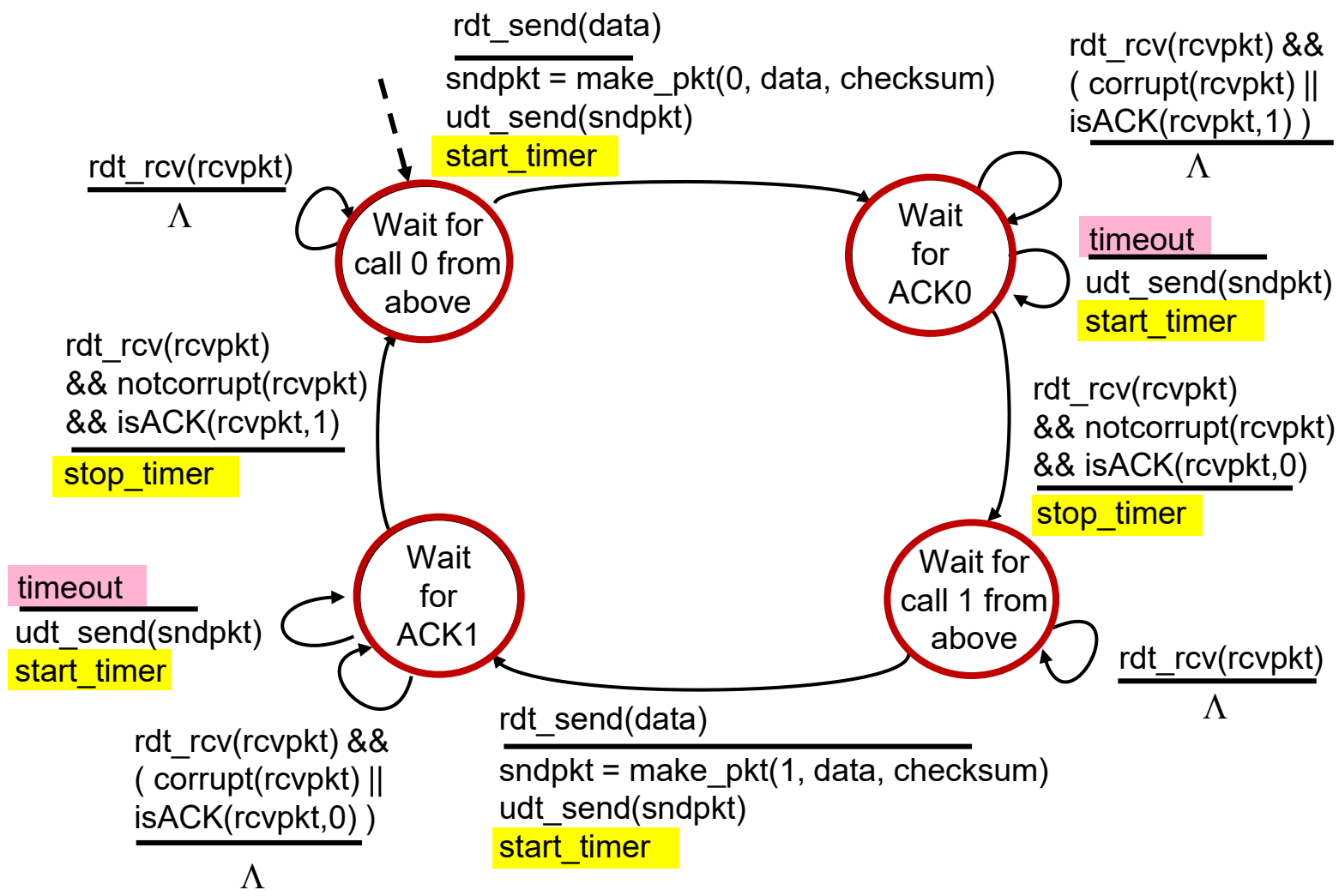


SPEAKER NOTES

We see the familiar four state sender from our rdt 2.1 and 2.2 protocols. Top two states when sending packet with zero seq # and bottom two

Troy University Resources

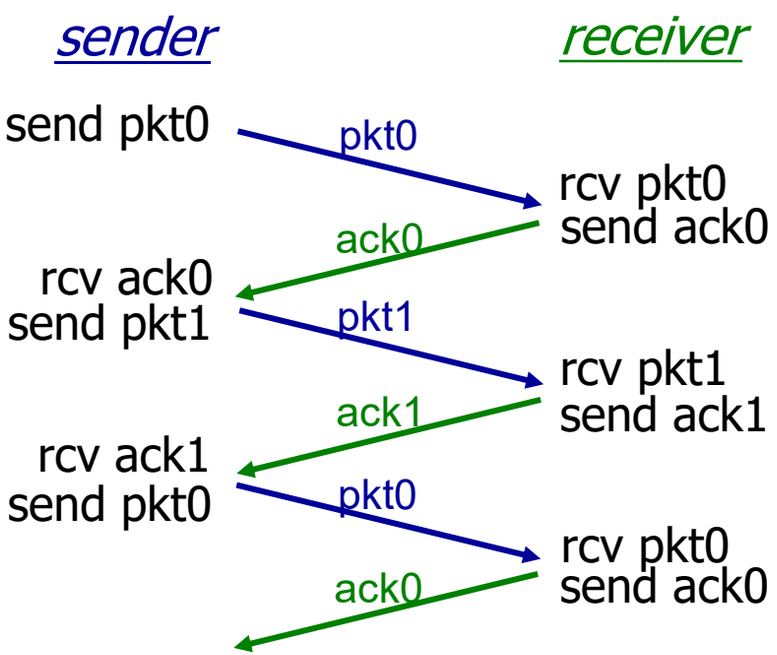
rdt3.0 sender



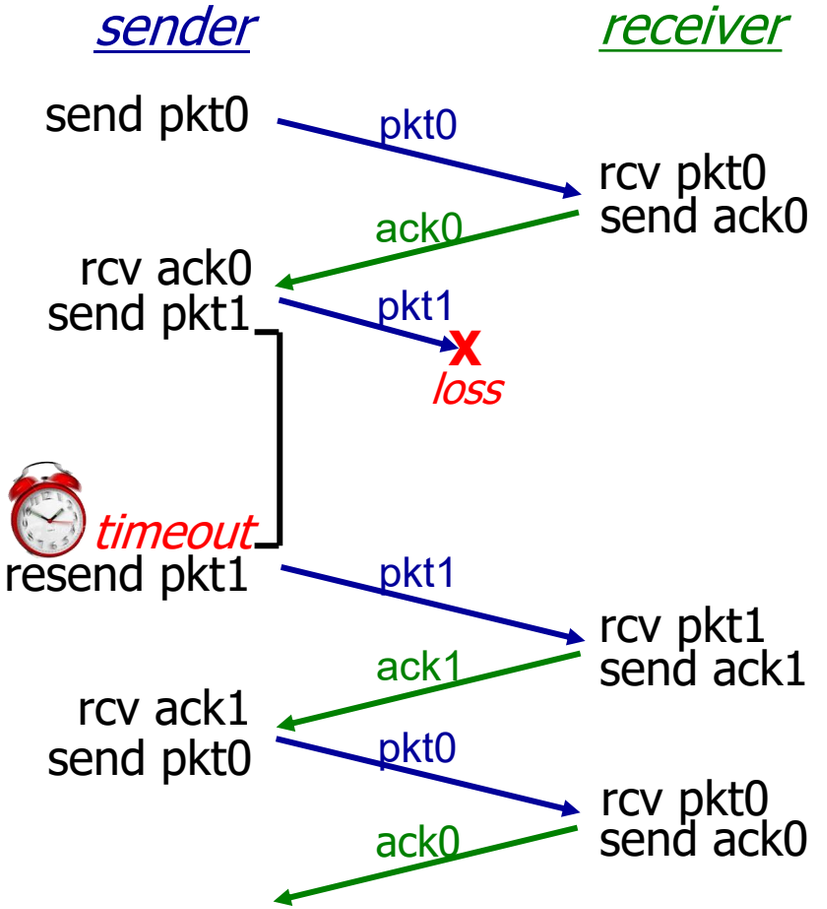
SPEAKER NOTES

We see the familiar four state sender from our rdt 2.1 and 2.2 protocols. Top two states when sending packet with zero seq # and bottom two

rdt3.0 in action



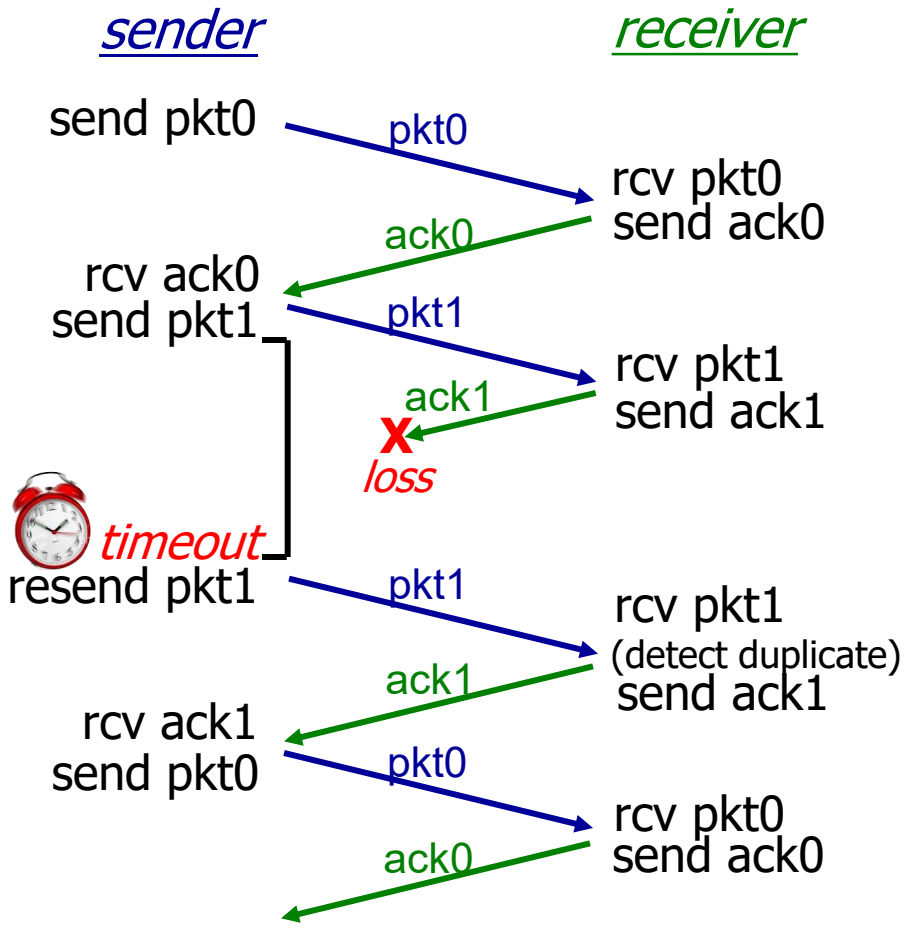
(a) no loss



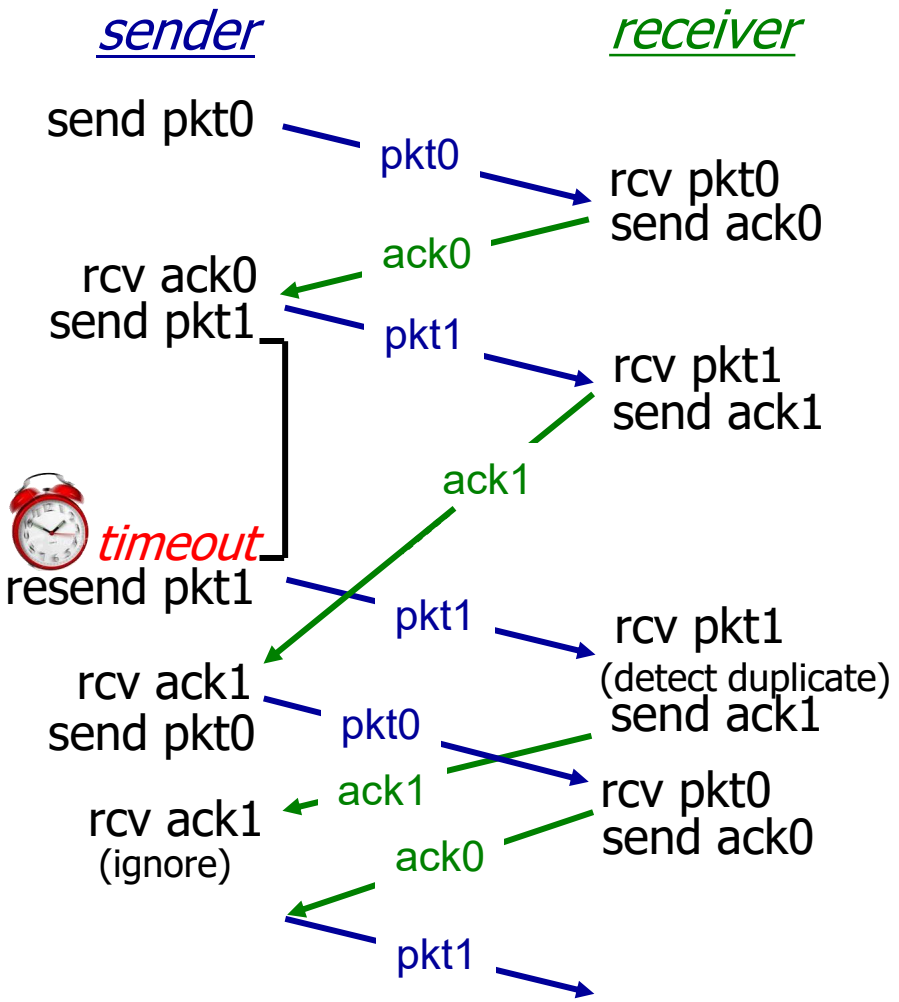
(b) packet loss

Troy University Resources

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt3.0 (stop-and-wait)

- U_{sender}: *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

SPEAKER NOTES

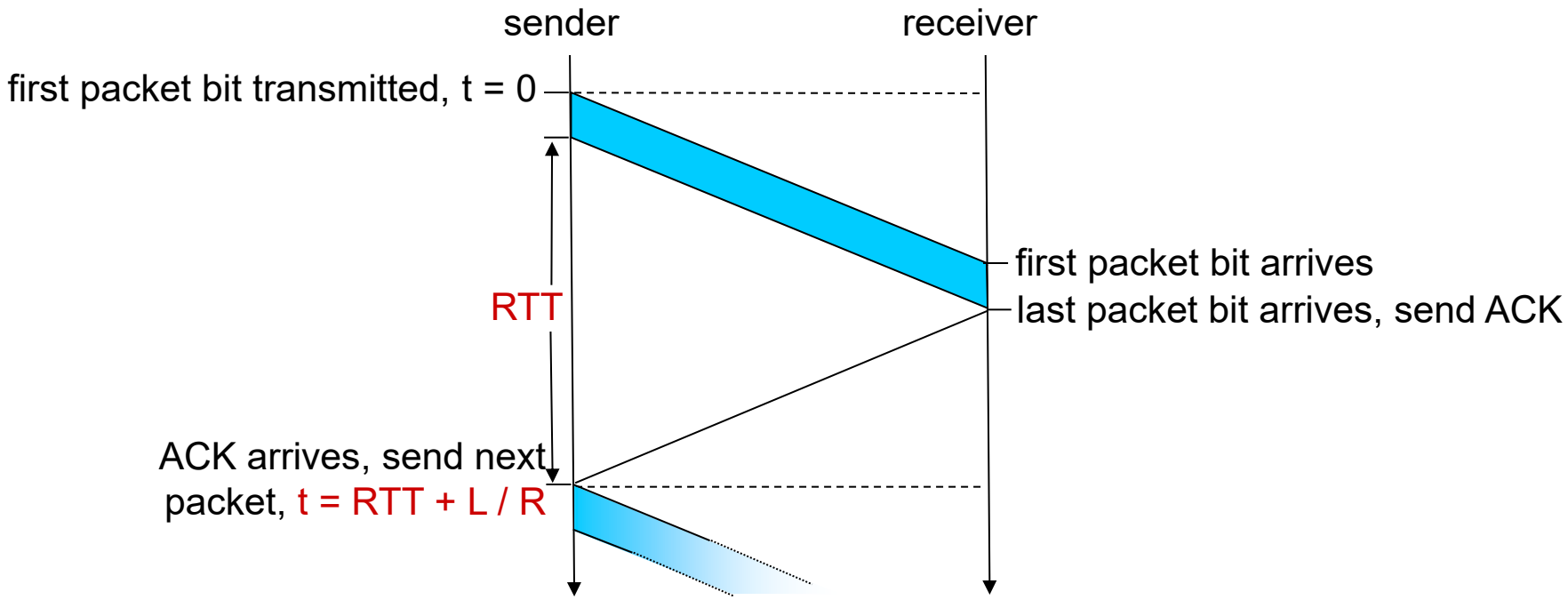
if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link

network protocol limits use of physical resources

Let’s develop a formula for utilization

Troy University Resources

rdt3.0: stop-and-wait operation

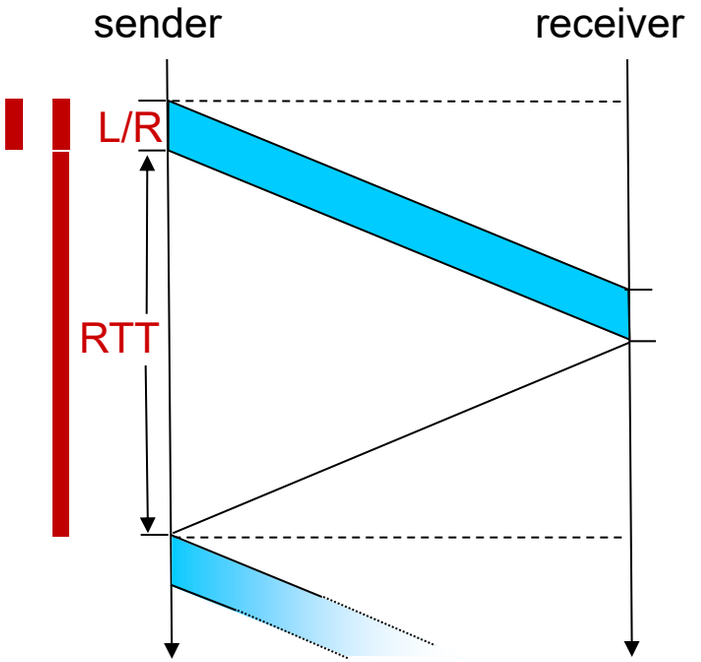


rdt3.0: stop-and-wait operation

$$U_{sender} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.00027$$



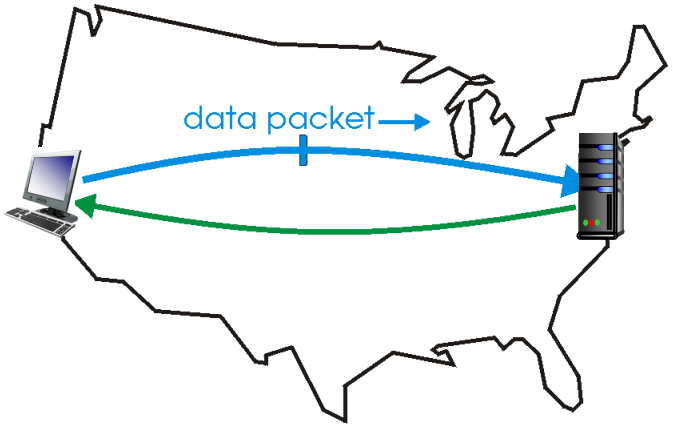
- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

Troy University Resources

rdt3.0: pipelined protocols operation

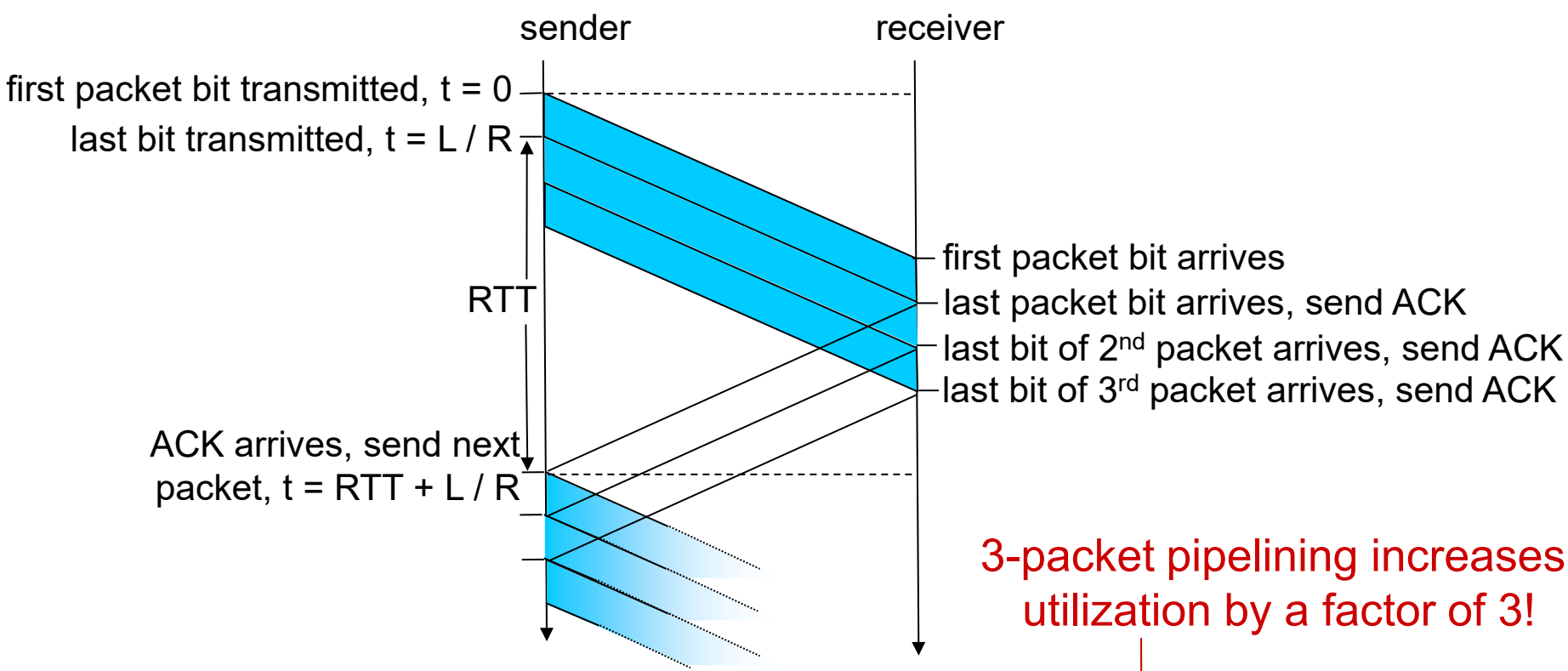
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation

Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

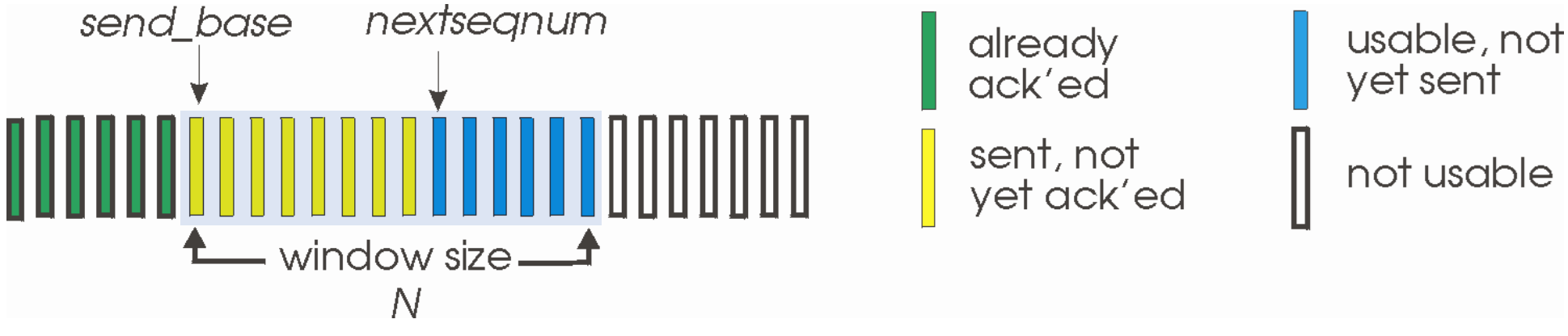
SPEAKER NOTES

two generic forms of pipelined protocols: go-Back-N, selective repeat

Troy University Resources

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header



- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

SPEAKER NOTES

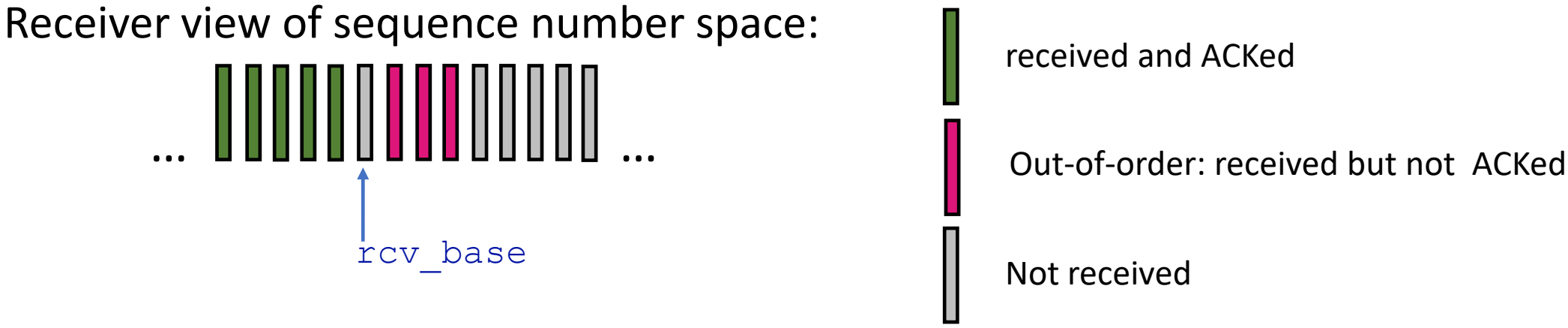
window size of 14, 8 have been sent but are not yet acknowledged, 6 sequence numbers are available for us. In window, but no calls from above to use them.

Note – we'll skip the Go-Back-N FSM specification you can check that out in PowerPoint slides or book)

TCP uses cumulative ACK

Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

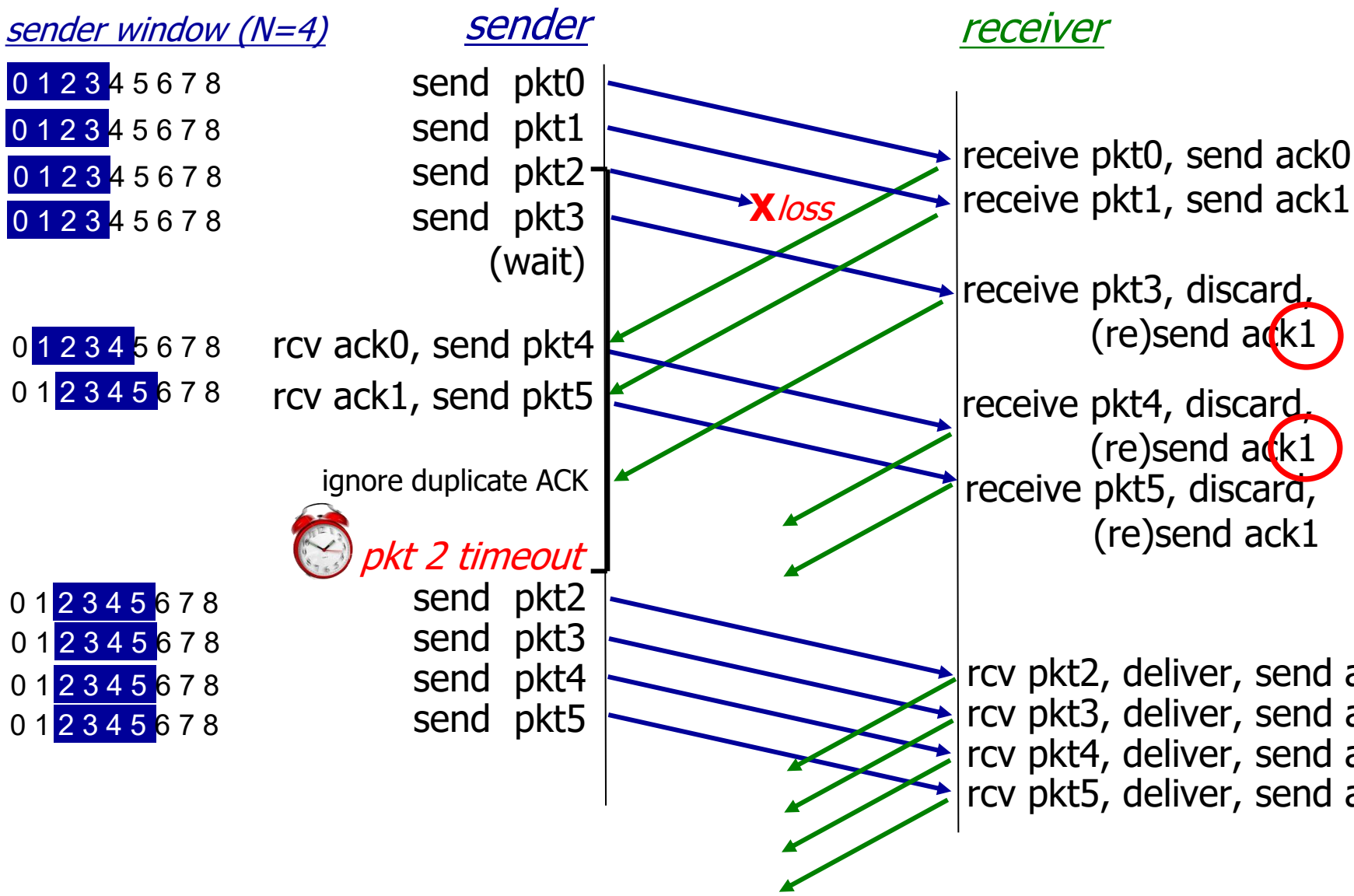


SPEAKER NOTES

Note – we'll skip the Go-Back-N FSM specification (actually it's

Troy University Resources

Go-Back-N in action



SPEAKER NOTES

Let's skip FSM specification for GBN – check out the book or ppt – and let's watch GBN sender and receivers in action.
Let assume a window size of 4. at t=0, sender sends packets 0, 1, 2 3, 4, and packet 2 will be lost

At the receiver:
Packet 0 received ACK0 generated
Packet 1 received ACK1 generated
Packet 2 is lost, and so when packet 3 is received, ACK 1 is sent – that's the cumulative ACK, re-Acknowledging the receipt of packet 1. and in this implementation packet 3 is discarded

Selective repeat: the approach

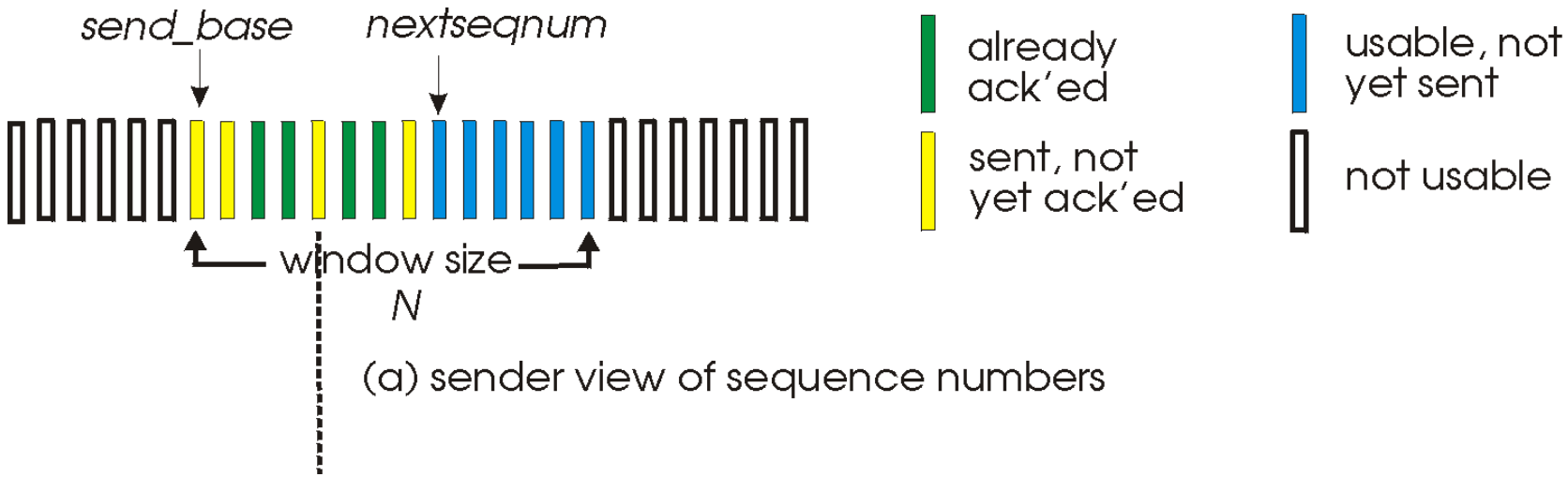
- *pipelining*: *multiple* packets in flight
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer
- sender:
  - maintains (conceptually) a timer for each unACKed pkt
    - timeout: retransmits single unACKed packet associated with timeout
  - maintains (conceptually) "window" over *N* consecutive seq #s
    - limits pipelined, "in flight" packets to be within this window

SPEAKER NOTES

An important mechanism of GBN was the use of the cumulative acknowledgements, and as we mentioned, cumulative ACKs are used in TCP

An alternate ACK mechanism would be for the receiver to individually acknowledge specific packets as they are received. This mechanism is at the heart of the Selective repeat protocol.

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase+N-1]:

- mark packet *n* as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

receiver

packet *n* in [rcvbase, rcvbase+N-1]

- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

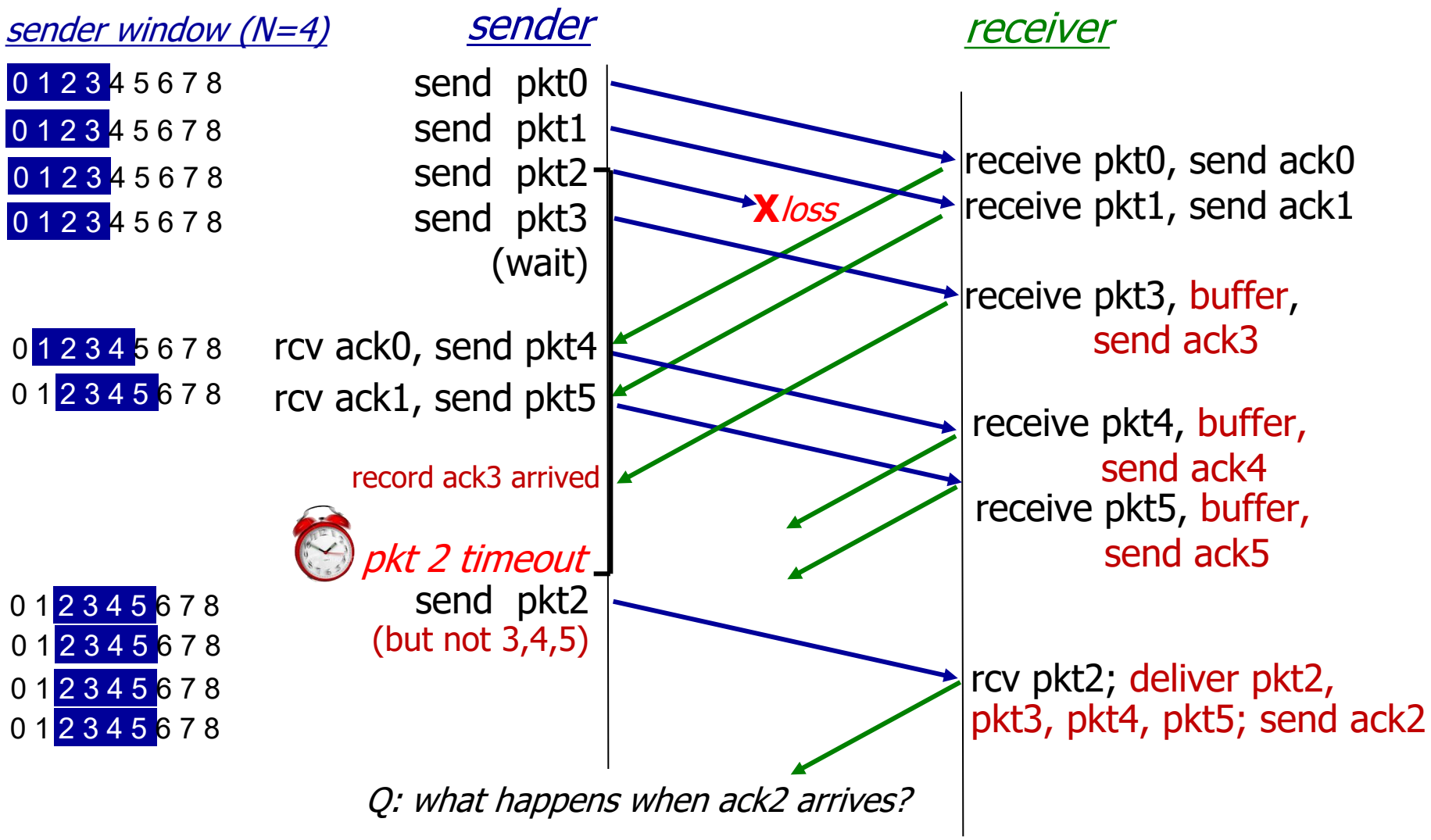
- ignore

SPEAKER NOTES

If the packet is in order, its data will be delivered, as will any buffered data that can now be delivered in order

Troy University Resources

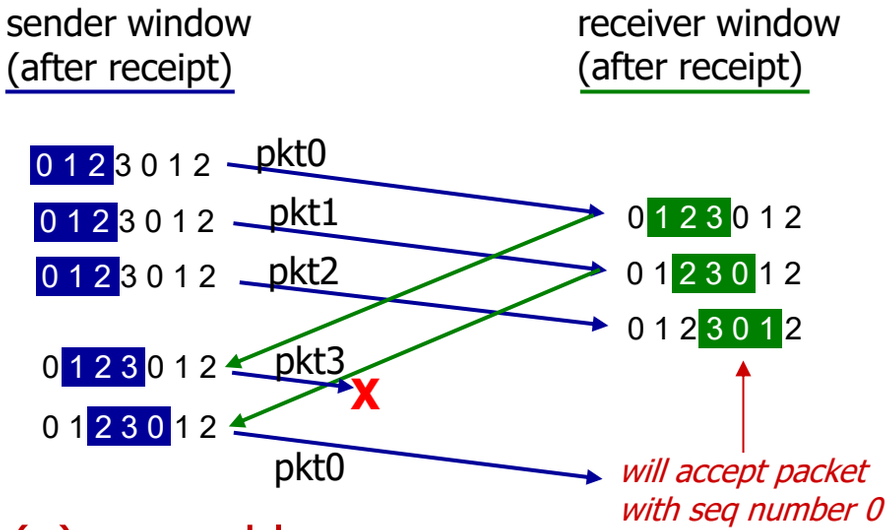
Selective Repeat in action



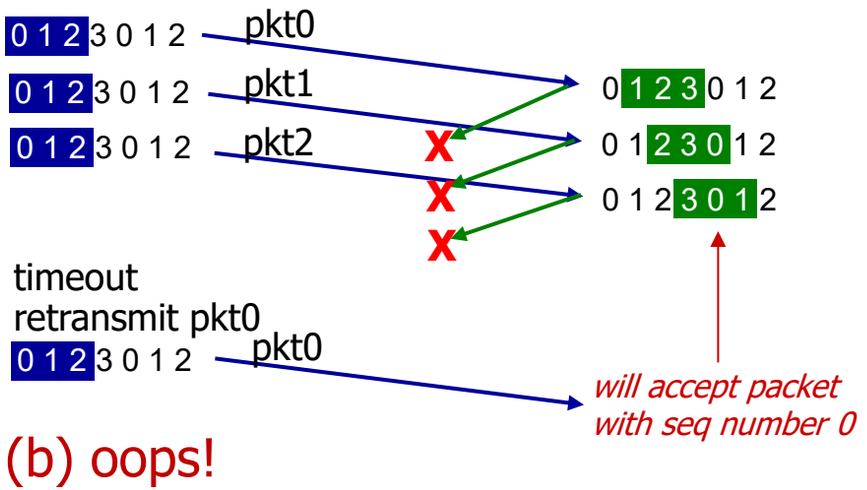
Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



(a) no problem



(b) oops!

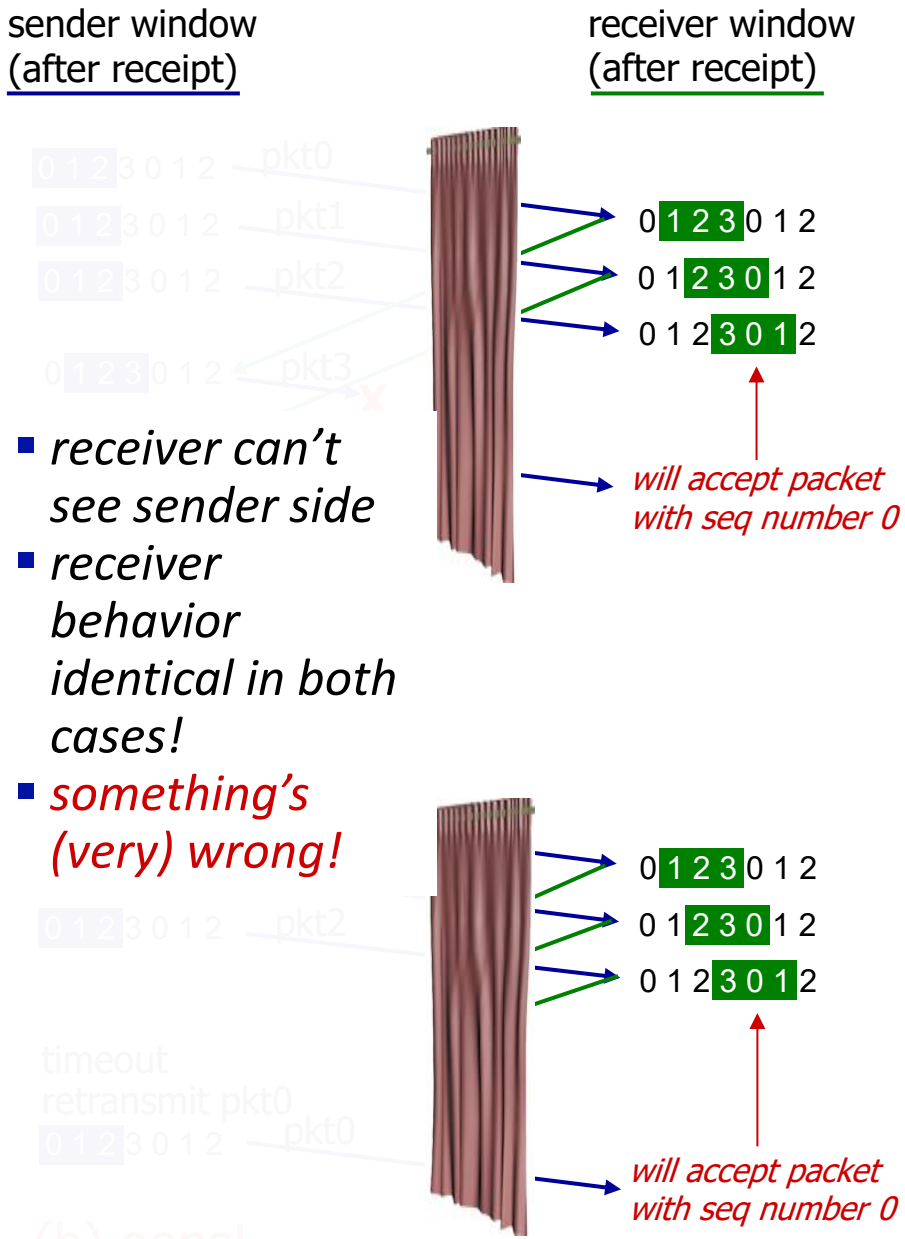
Troy University Resources

Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?



Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control



Troy University Resources

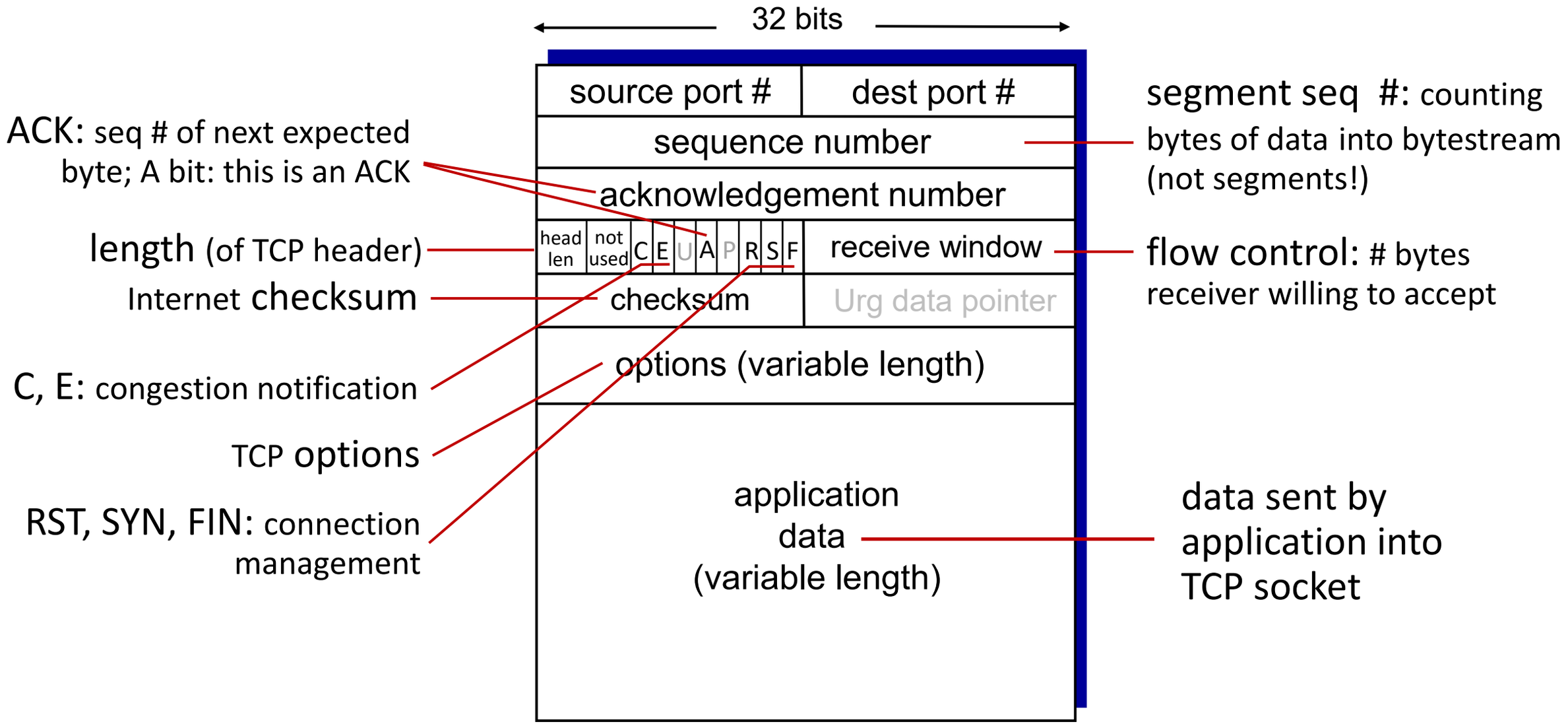
TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

SPEAKER NOTES

Typical MSS is 1460 bytes

TCP segment structure



Troy University Resources

TCP sequence numbers, ACKs

Sequence numbers:

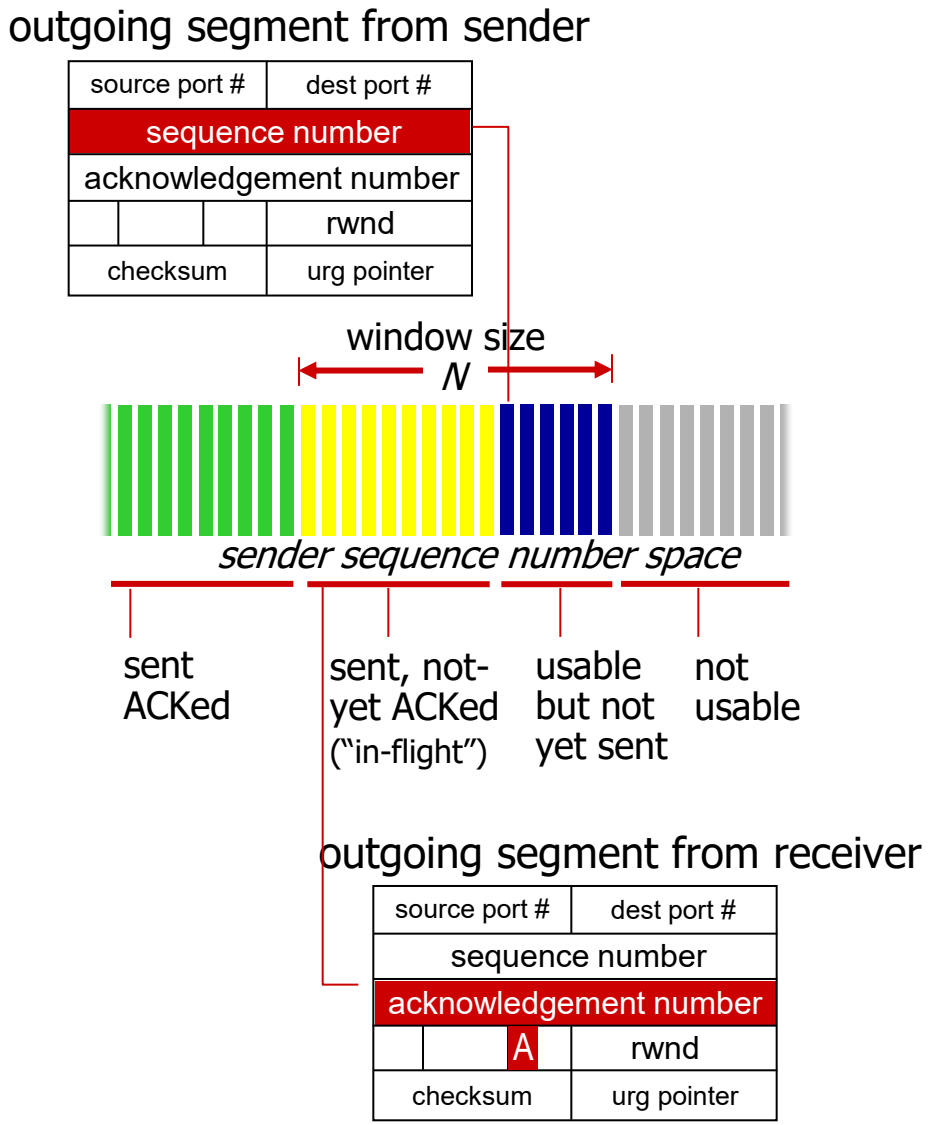
- byte stream "number" of first byte in segment's data

Acknowledgements:

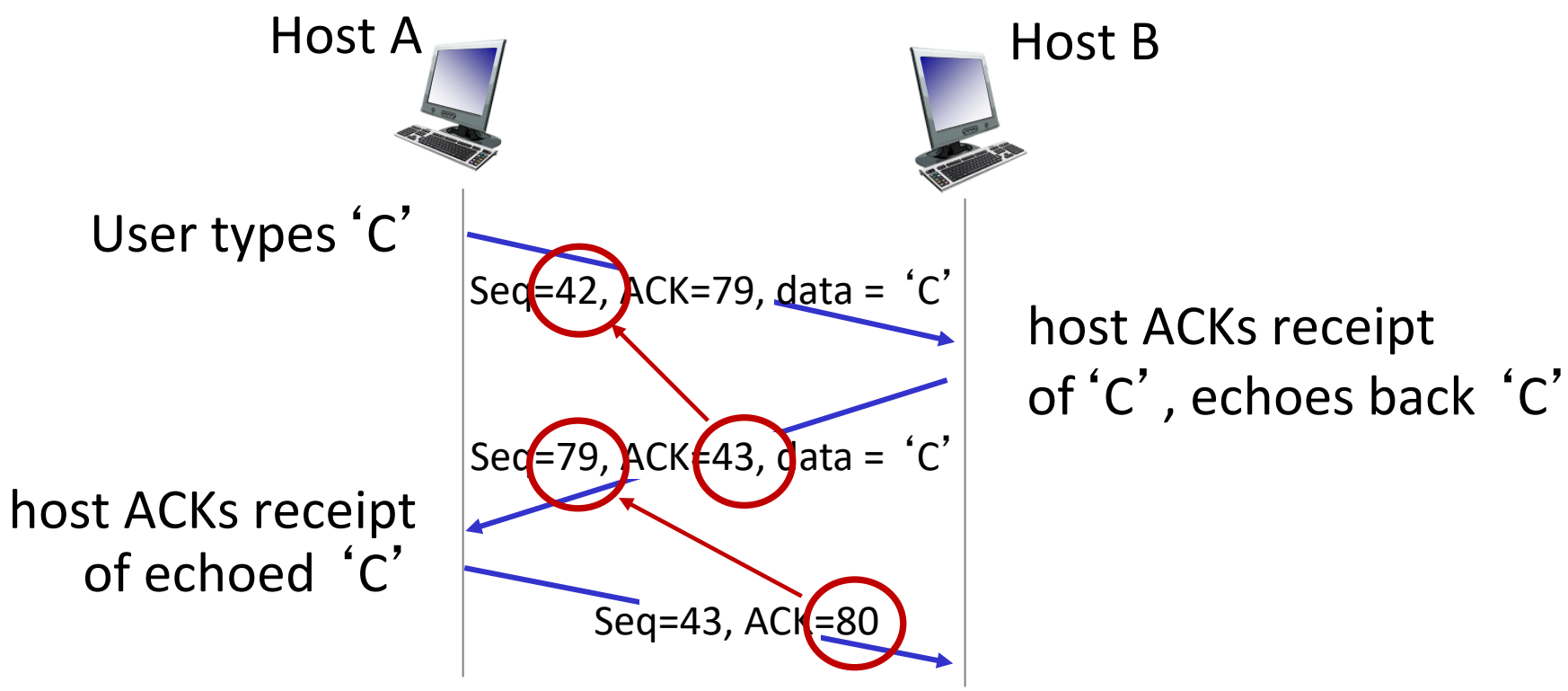
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- *A:* TCP spec doesn't say, - up to implementor



TCP sequence numbers, ACKs



simple telnet scenario

SPEAKER NOTES

The key thing to note here is that the ACK number (43) on the B-to-A segment is one more than the sequence number (42) on the A-to-B segment that triggered that ACK

Similarly, the ACK number (80) on the last A-to-B segment is one more than the sequence number (79) on the B-to-A segment that triggered that ACK

Troy University Resources

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

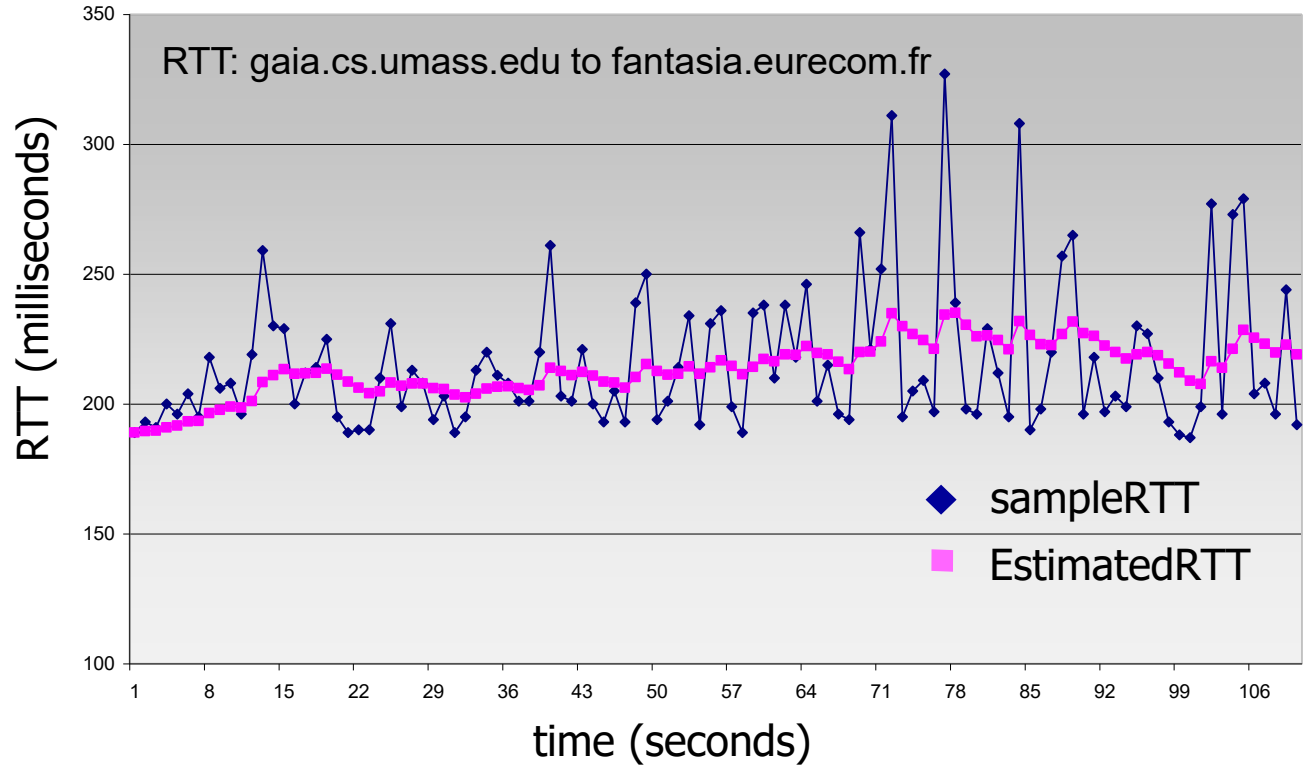
Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current `SampleRTT`

TCP round trip time, timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



SPEAKER NOTES

This is how TCP re-computes the estimated RTT each time a new SampleRTT is taken.
The process is knows as an exponentially weighted moving average, shown by the equation here.
<say it>
Where alpha reflects the influence of the most recent measurements on the estimated RTT; a typical value of alpha used in implementations is .125

The graph at the bottom show measured RTTs between a host in the Massachusetts and a host in France, as well as the estimated, “smoothed” RTT

TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT**: want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```



estimated RTT

"safety margin"

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

(typically, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

SPEAKER NOTES

Given this value of the estimated RTT, TCP computes the timeout interval to be the estimated RTT plus a "safety margin"

And the intuition is that if we are seeing a large variation in SAMPLERTT – the RTT estimates are fluctuating a lot - then we'll want a larger safety margin

So TCP computes the Timeout interval to be the Estimated RTT plus 4 times a measure of deviation in the RTT.

The deviation in the RTT is computed as the eWMA of the difference between the most recently measured SampleRTT from the Estimated RTT

TCP Sender (simplified)

event: data received from application

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unACKed segment
  - expiration interval: **`TimeOutInterval`**

event: timeout

- retransmit segment that caused timeout
- restart timer

event: ACK received

- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments

SPEAKER NOTES

Given these details of TCP sequence numbers, acks, and timers, we can now describe the big picture view of how the TCP sender and receiver operate

You can check out FSMs in book; let's just give an English text description here and let's start with the sender.

Troy University Resources

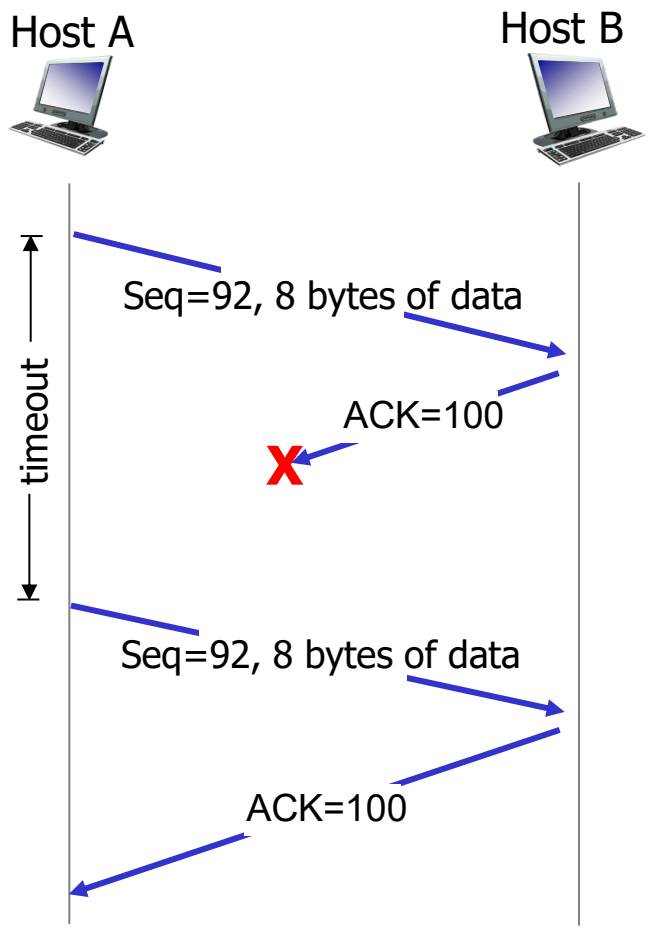
TCP Receiver: ACK generation [RFC 5681]

Event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK (acking last correctly received in order byte),* indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

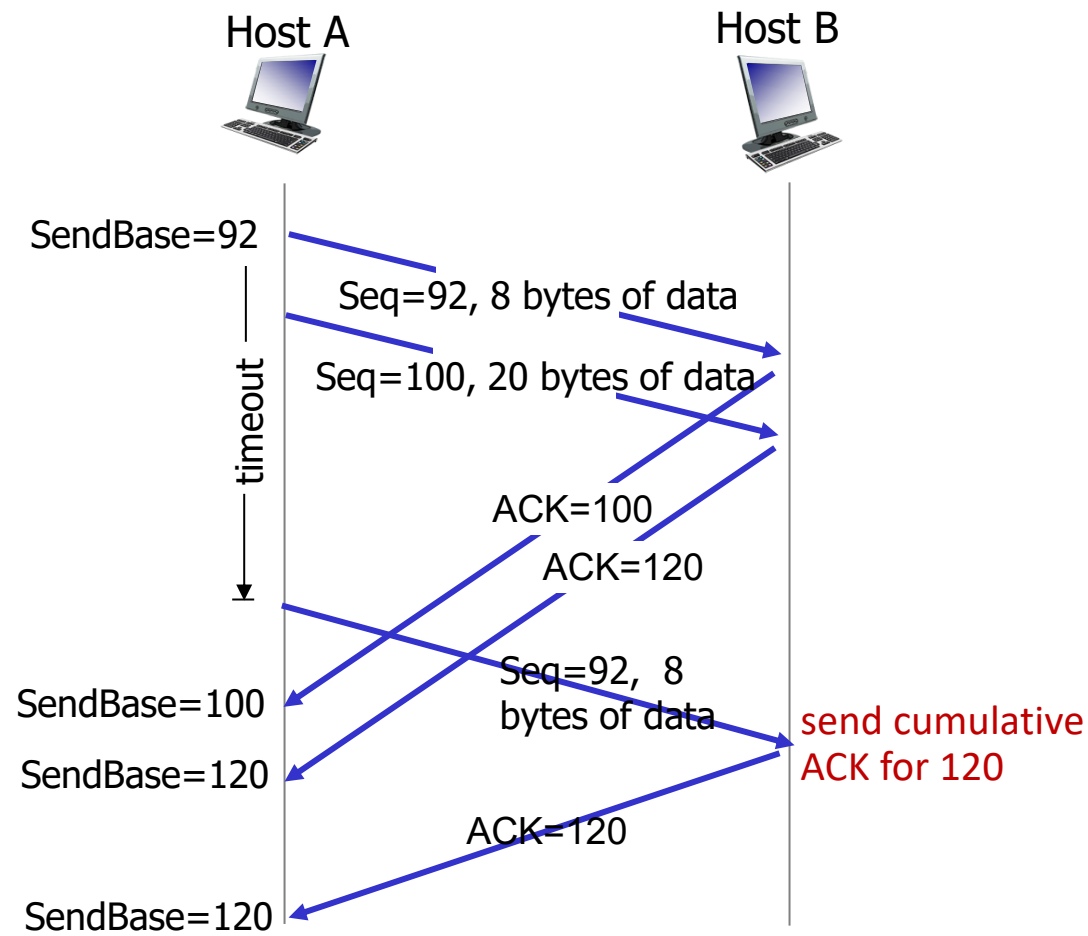
SPEAKER NOTES

Rather than immediately Acknowledging this segment, many TCP implementations will wait for half a second for another in-order segment to arrive, and then generate a single cumulative ACK for both segments – thus decreasing the amount of ACK traffic. The arrival of this second in-order segment and the cumulative ACK generation that covers both segments is the second row in this table.

TCP: retransmission scenarios



lost ACK scenario



premature timeout

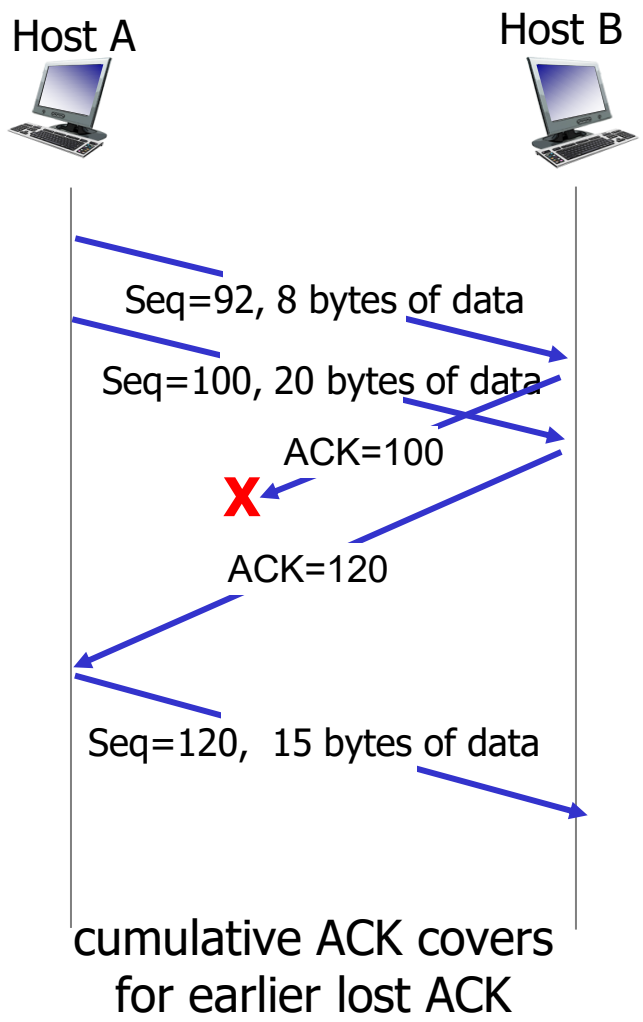
SPEAKER NOTES

To cement our understanding of TCP reliability, let's look a a few retransmission scenarios

In the first case a TCP segments is transmitted and the ACK is lost, and the TCP timeout mechanism results in another copy of being transmitted and then re-ACKed a the sender

In the second example two segments are sent and acknowledged, but there is a premature timeout e for the first segment, which is retransmitted. Note that when this retransmitted segment is received, the receiver has already received the first two segments, and so resends a cumulative ACK for both segments received so far, rather than an ACK for just this fist segment.

TCP: retransmission scenarios



SPEAKER NOTES

And in this last example, two segments are again transmitted, the first ACK is lost but the second ACK, a cumulative ACK arrives at the sender, which then can transmit a third segment, knowing that the first two have arrived, even though the ACK for the first segment was lost

TCP fast retransmit

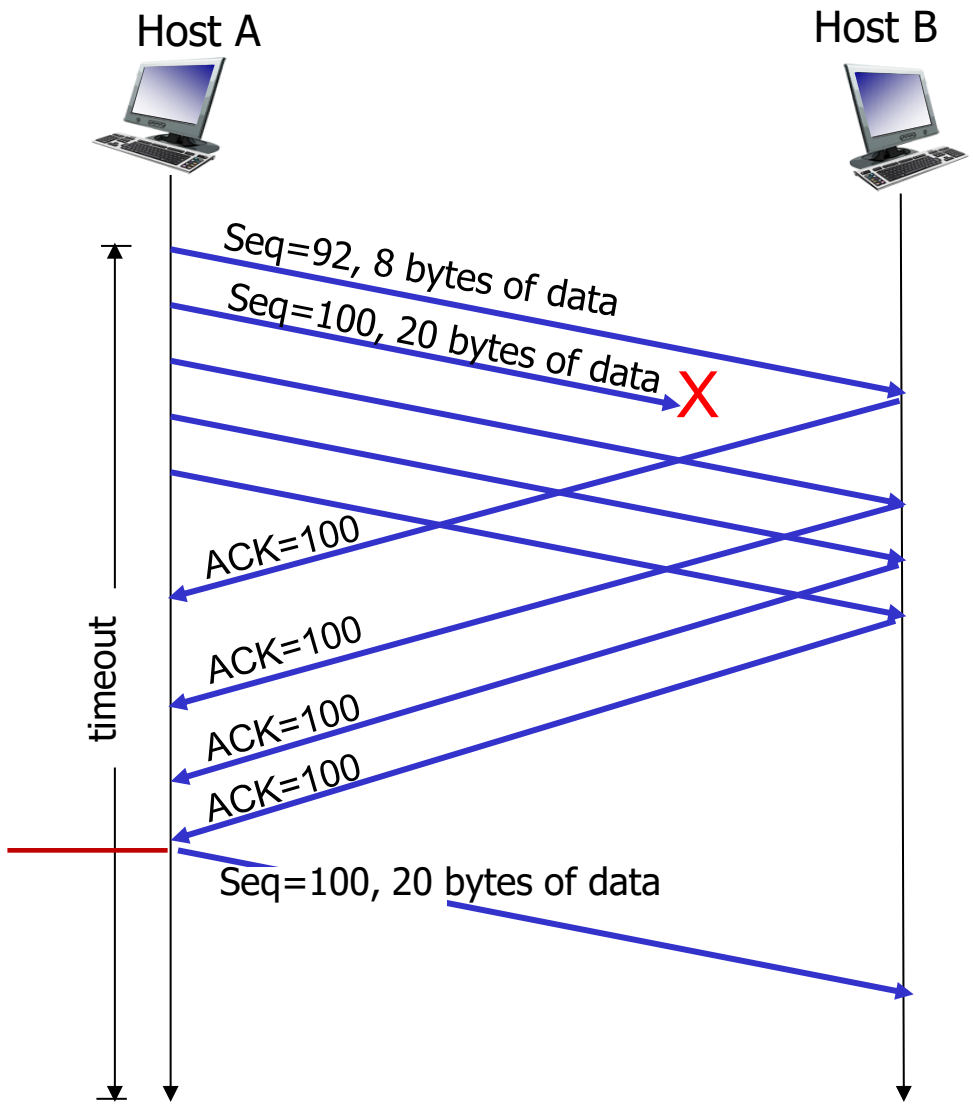
TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout



Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



SPEAKER NOTES

Let's wrap up our study of TCP reliability by discussing an optimization to the original TCP known as TCP fast retransmit,

Take a look at this example on the right where 5 segments are transmitted and the second segment is lost. In this case the TCP receiver sends an ACK 100 acknowledging the first received segment.
When the third segment arrives at the receiver, the TCP receiver sends another ACK 100 since the second segment has not arrived. And similarly for the 4th and 5th segments to arrive.

Now what does the sender see? The sender receives the first ACK 100 it has been hoping for, but then three additional duplicate ACK100s arrive. The sender knows that somethings' wrong – it knows the first segment arrived at the receiver but three later arriving segments at the receiver – the ones that generated the three duplicate ACKs – we received correctly but were not in order. That is, that there was a missing segment at the receiver when each of the three duplicate ACK were generated.

With fast retransmit, the arrival of three duplicate ACK causes the sender to retransmit its oldest unACKed segment, without waiting for a timeout event. This allows TCP to recover more quickly from what is very likely a loss event

specifically that the second segment has been lost, since three higher -numbered segments were received

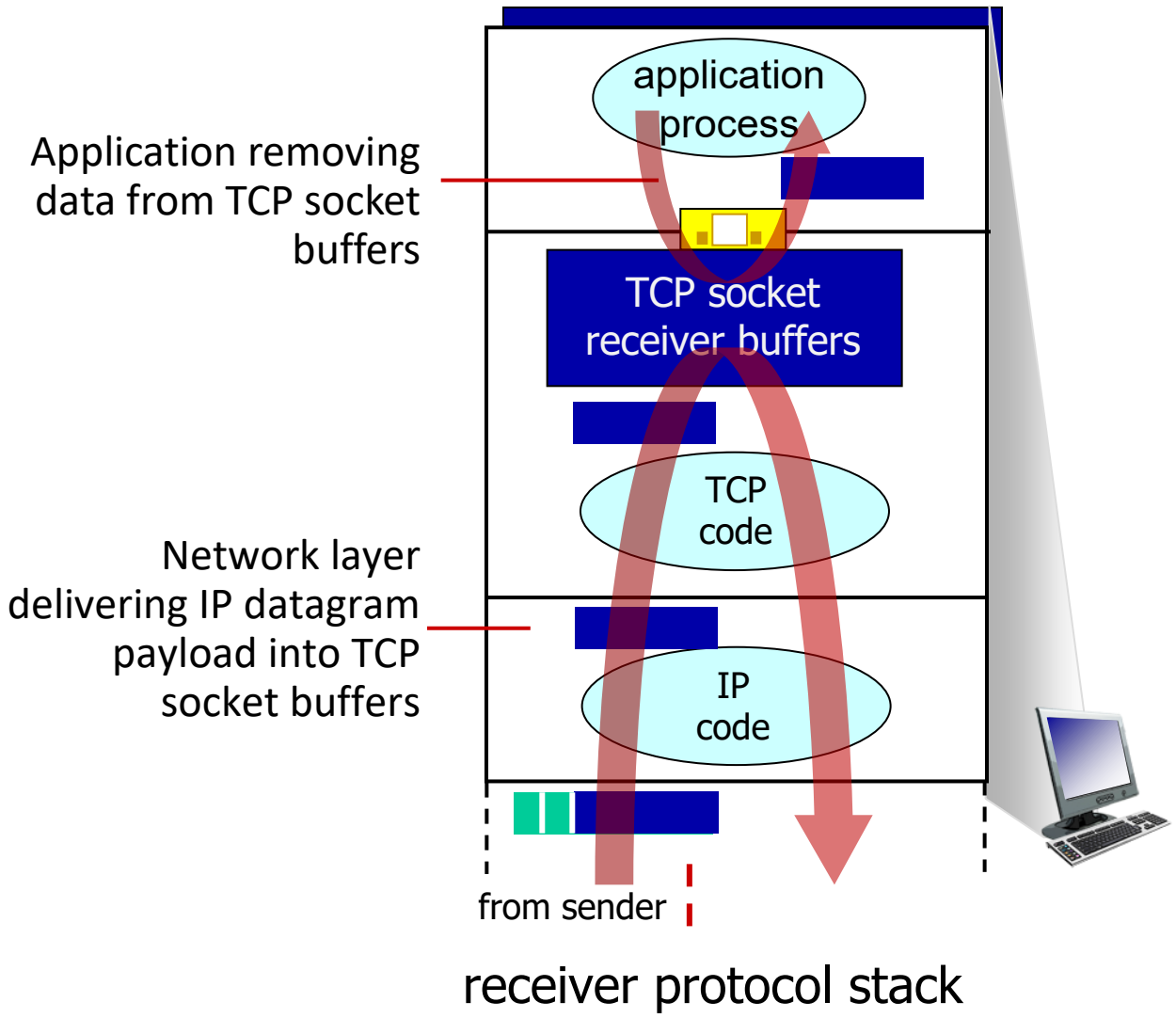
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
  - segment structure
  - reliable data transfer
  - **flow control**
  - **connection management**
- Principles of congestion control
- TCP congestion control



TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



SPEAKER NOTES

(Presuming an intro)

Before diving into the details of TCP flow control, let's first get the general context and motivate the need for flow control.

This diagram show a typical transport-layer implementation

A segment is brought up the protocol stack to the transport layer, and the segment's payload is removed from the segment and written INTO socket buffers.

How does data get taken OUT of socket buffers? By applications performing socket reads, as we learned in Chapter 2.

And so the question is "What happens if network layer delivers data faster than an application-layer process removes data from socket buffers?"

Let's watch a video of what happens when things arrive way too fast to fast to be processed.

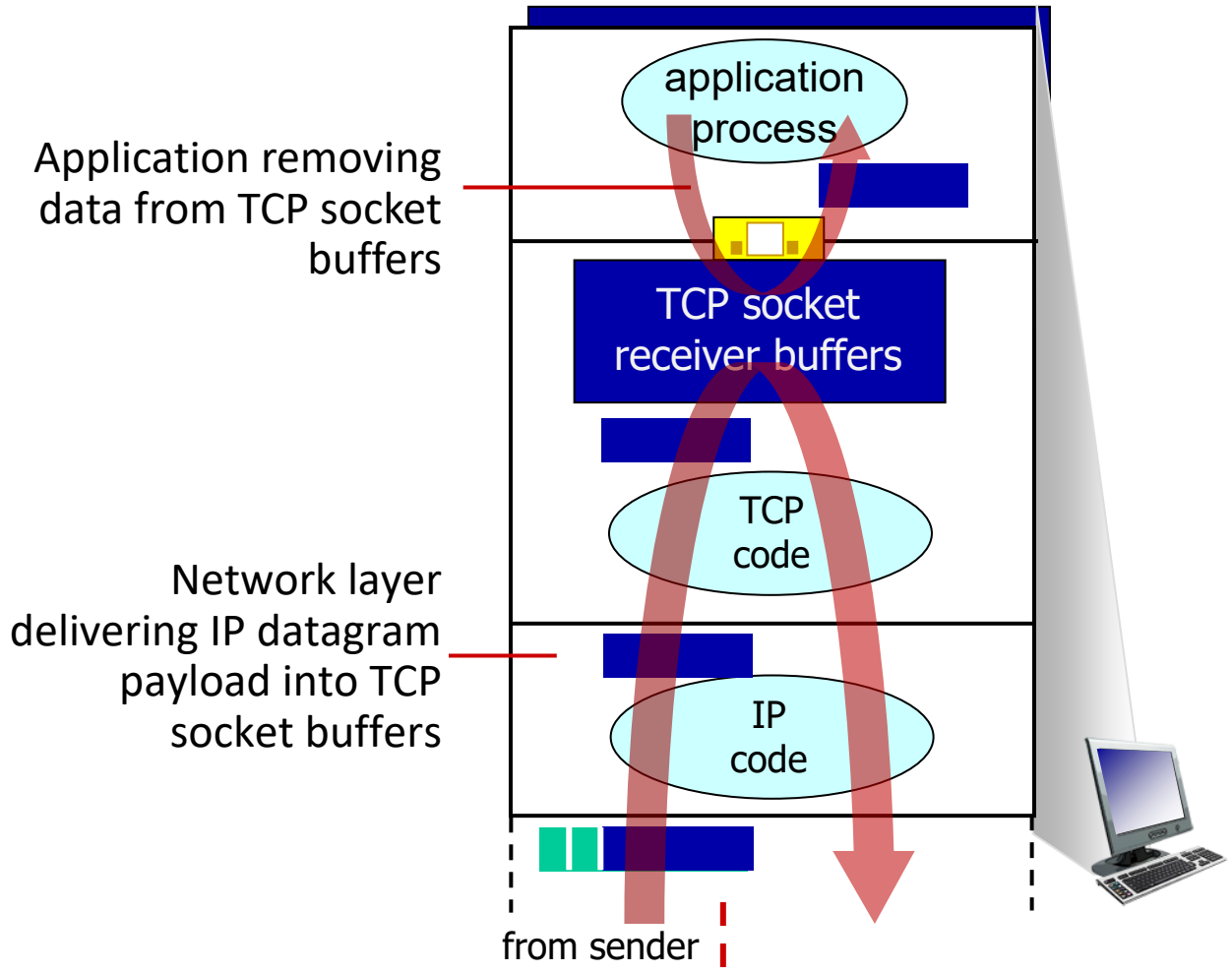
<video>. (I love that video). Another human analogy showing the need for flow control is the saying – to use some English slang - "no one can drink from a firehose"

Flow control is a mechanism to the calamity of a receiver being over-run by a sender that is sending too fast – it allows the RECEIVER to explicitly control the SENDER so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?





SPEAKER NOTES

(Presuming an intro)

Before diving into the details of TCP flow control, let's first get the general context and motivate the need for flow control.

This diagram show a typical transport-layer implementation

A segment is brought up the protocol stack to the transport layer, and the segment's payload is removed from the segment and written INTO socket buffers.

How does data get taken OUT of socket buffers? By applications performing socket reads, as we learned in Chapter 2.

And so the question is "What happens if network layer delivers data faster than an application-layer process removes data from socket buffers?"

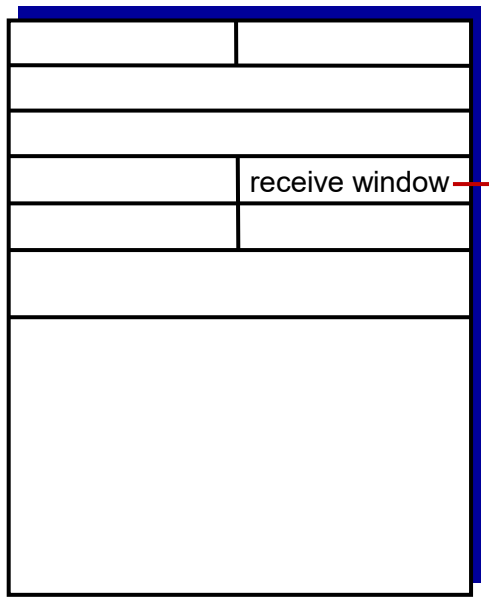
Let's watch a video of what happens when things arrive way too fast to fast to be processed.

<video>. (I love that video). Another human analogy showing the need for flow control is the saying – to use some English slang - "no one can drink from a firehose"

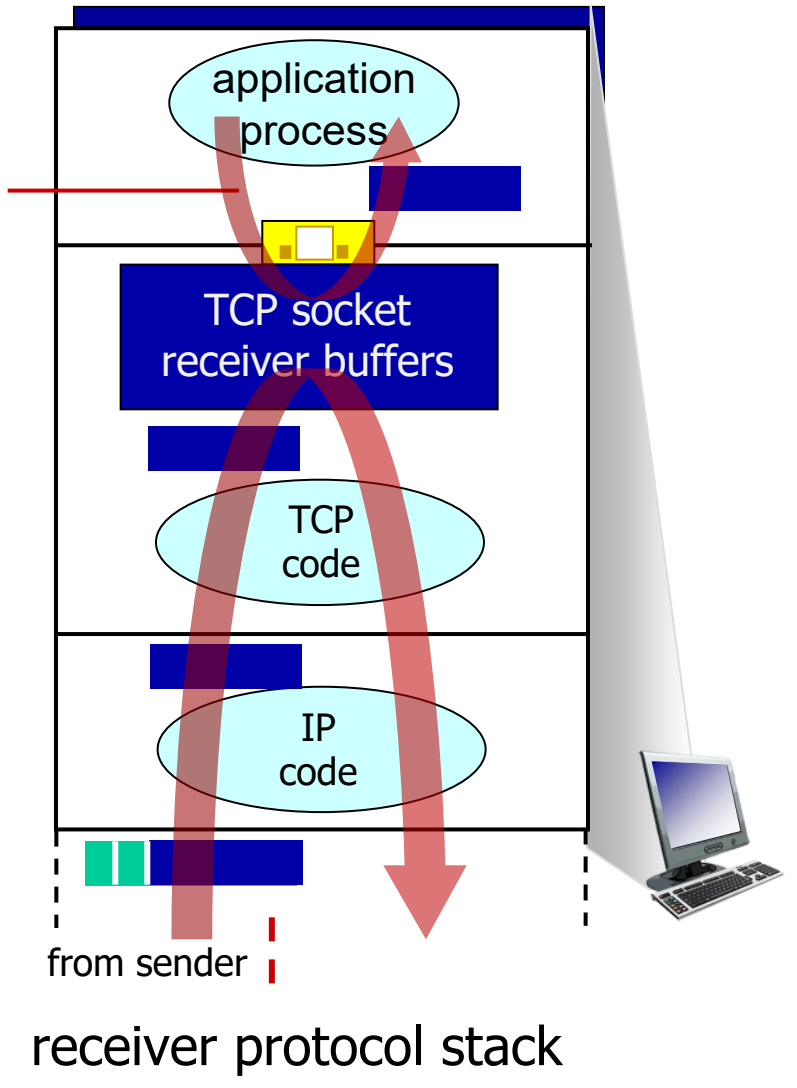
Flow control is a mechanism to the calamity of a receiver being over-run by a sender that is sending too fast – it allows the RECEIVER to explicitly control the SENDER so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



flow control: # bytes receiver willing to accept



SPEAKER NOTES

(Presuming an intro)

Before diving into the details of TCP flow control, let's first get the general context and motivate the need for flow control.

This diagram show a typical transport-layer implementation

A segment is brought up the protocol stack to the transport layer, and the segment's payload is removed from the segment and written INTO socket buffers.

How does data get taken OUT of socket buffers? By applications performing socket reads, as we learned in Chapter 2.

And so the question is "What happens if network layer delivers data faster than an application-layer process removes data from socket buffers?"

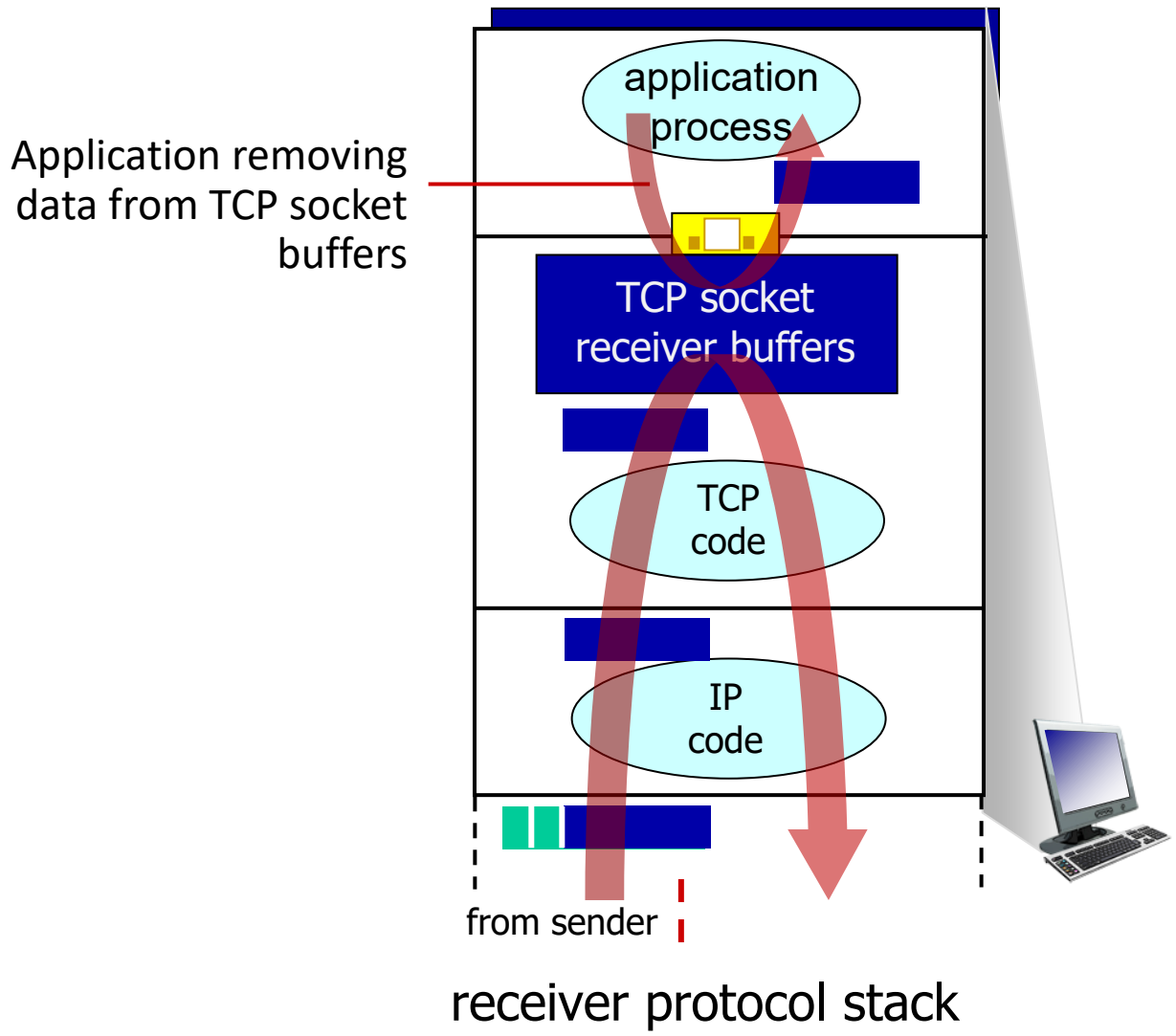
Let's watch a video of what happens when things arrive way too fast to fast to be processed.

<video>. (I love that video). Another human analogy showing the need for flow control is the saying – to use some English slang - "no one can drink from a firehose"

Flow control is a mechanism to the calamity of a receiver being over-run by a sender that is sending too fast – it allows the RECEIVER to explicitly control the SENDER so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

SPEAKER NOTES

(Presuming an intro)

Before diving into the details of TCP flow control, let's first get the general context and motivate the need for flow control.

This diagram show a typical transport-layer implementation

A segment is brought up the protocol stack to the transport layer, and the segment's payload is removed from the segment and written INTO socket buffers.

How does data get taken OUT of socket buffers? By applications performing socket reads, as we learned in Chapter 2.

And so the question is "What happens if network layer delivers data faster than an application-layer process removes data from socket buffers?"

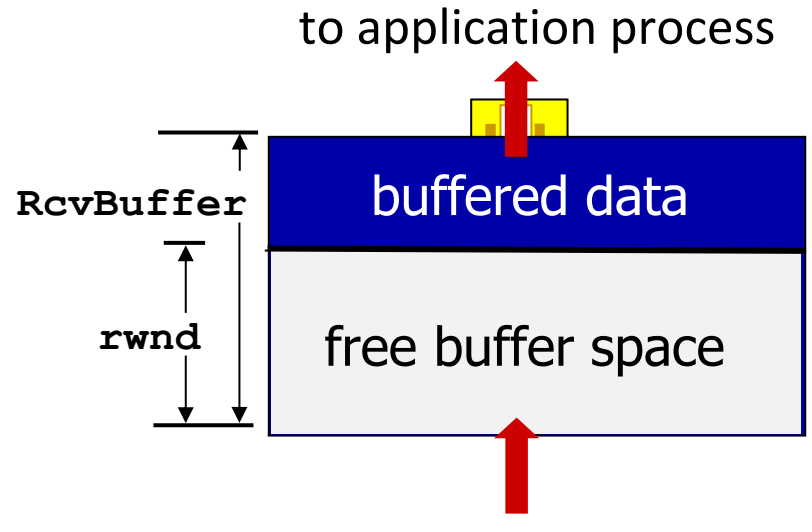
Let's watch a video of what happens when things arrive way too fast to fast to be processed.

<video>. (I love that video). Another human analogy showing the need for flow control is the saying – to use some English slang - "no one can drink from a firehose"

Flow control is a mechanism to the calamity of a receiver being over-run by a sender that is sending too fast – it allows the RECEIVER to explicitly control the SENDER so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP receiver-side buffering

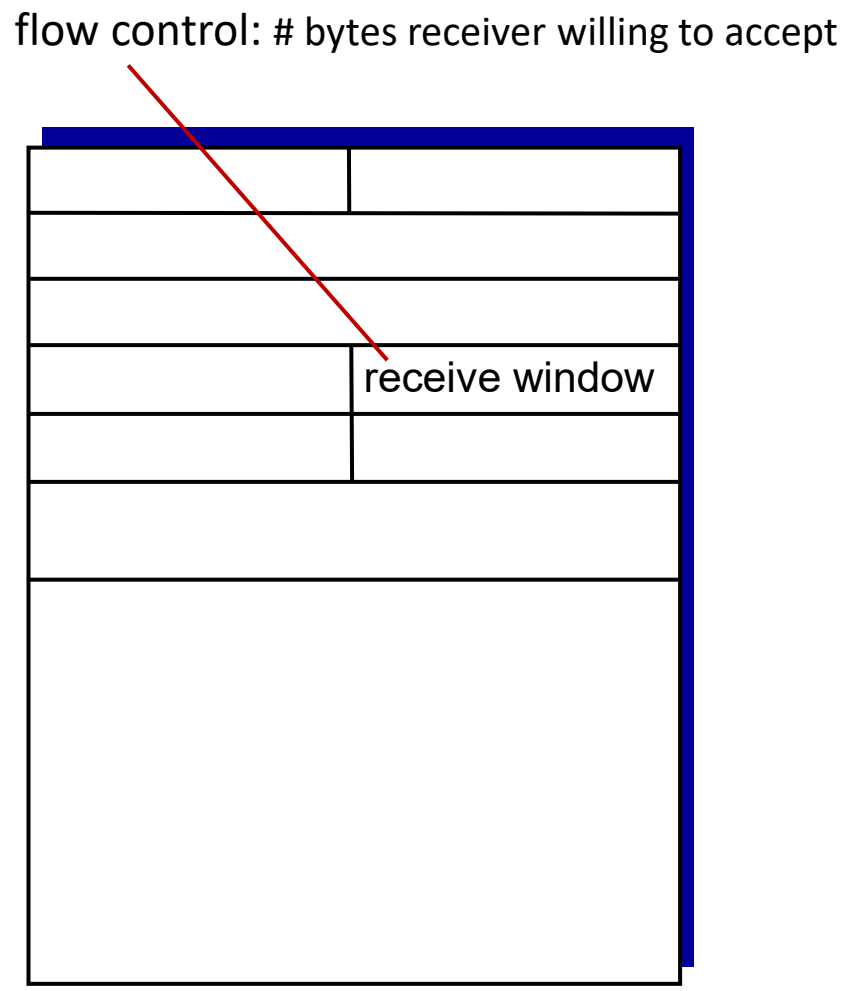
SPEAKER NOTES

Here's how TCP implement flow control. The basic idea is simple – the receiver informs the sender how much free buffer space there is, and the sender is limited to send no more than this amount of data. That the value o RWND in the diagram to the right.

This information is carried from the receiver to the sender in the "receiver advertised window" (do a PIP of header) in the TCP header, and the value will change as the amount of free buffer space fluctuates over time.

TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP segment format

SPEAKER NOTES

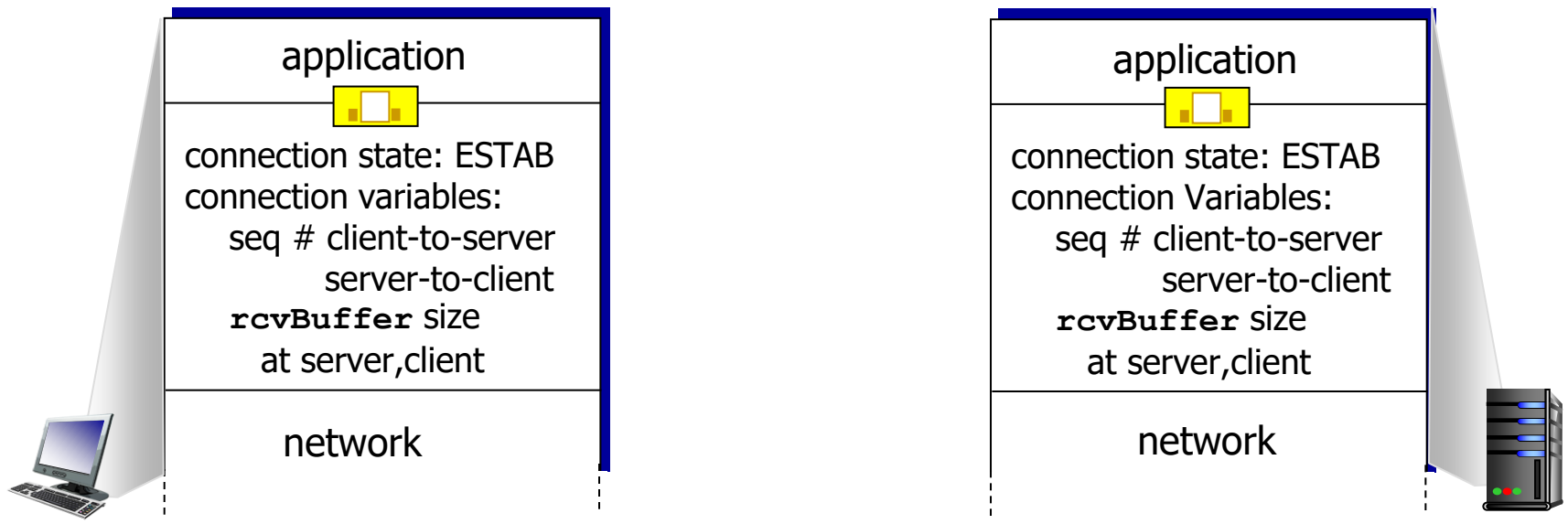
Here's how TCP implement flow control. The basic idea is simple – the receiver informs the sender how much free buffer space there is, and the sender is limited to send no more than this amount of data. That the value o RWND in the diagram to the right.

This information is carried from the receiver to the sender in the "receiver advertised window" (do a PIP of header) in the TCP header, and the value will change as the amount of free buffer space fluctuates over time.

TCP connection management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)




```
Socket clientSocket =
  newSocket("hostname","port number");
```

```
Socket connectionSocket =
  welcomeSocket.accept();
```


SPEAKER NOTES

The other TCP topic we’ll want to consider here is that of “connection management”

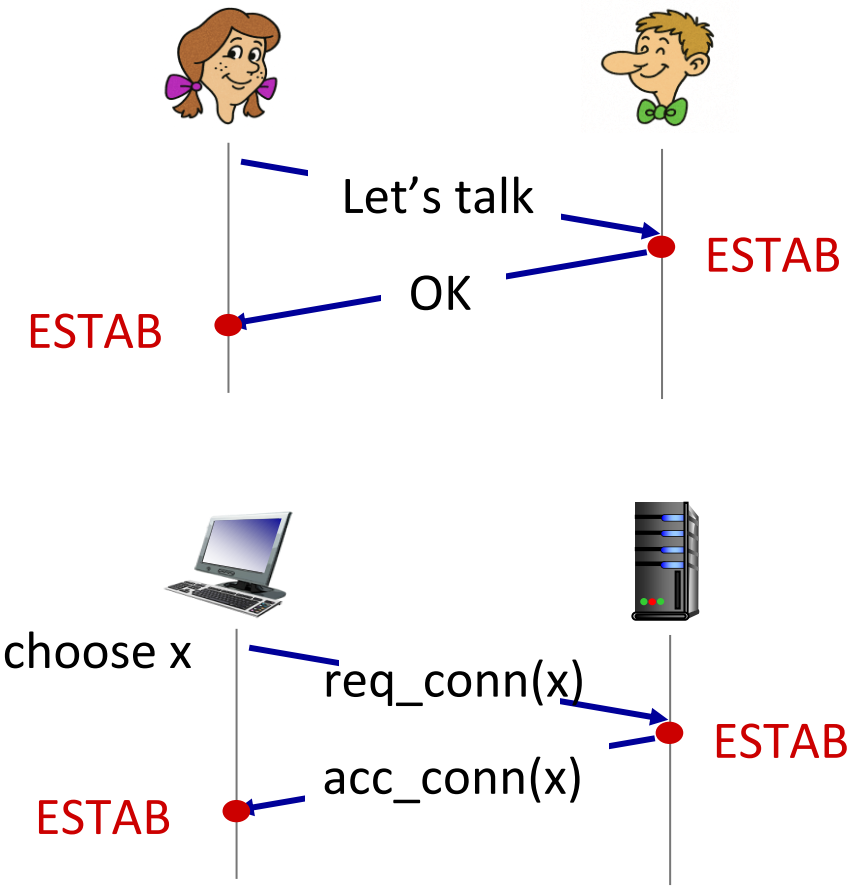
The TCP sender and receiver have a number of pieces of shared state that they must establish before actually communicating
FIRST they must both agree that they WANT to communicate with each other
Secondly there are connection parameters – the initial sequence number and the initial receiver-advertised bufferspace that they’ll want to agree on

This is done via a so-called handshake protocol – the client reaching our to the server, and the server answering back.

And before diving into the TCP handshake protocol, let’s first consider the problem of handshaking, of establishing shared state.

Agreeing to establish a connection

2-way handshake:



Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

SPEAKER NOTES

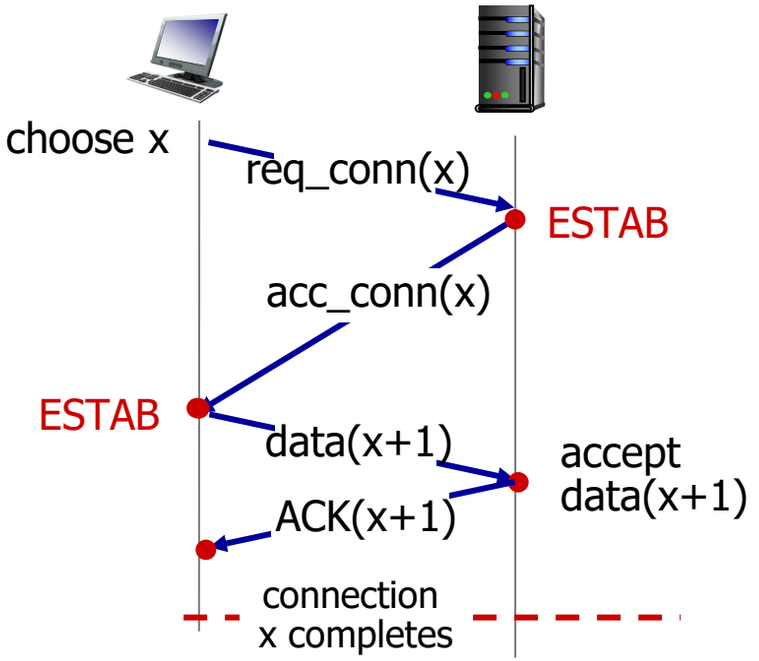
Here's an example of a two way handshake. Alice reaches out to Bob and say's "let's talk" and Bob says OK, and they start their conversation

For a network protocol, the equivalent protocol would be a client sending a "request connection" message saying "let's talk, the initial sequence number is x"
And the server would respond with a message "I accept your connect x"

And the question we want to ask ourselves is <talk through>

Will this work? Let's look at a few scenarios…

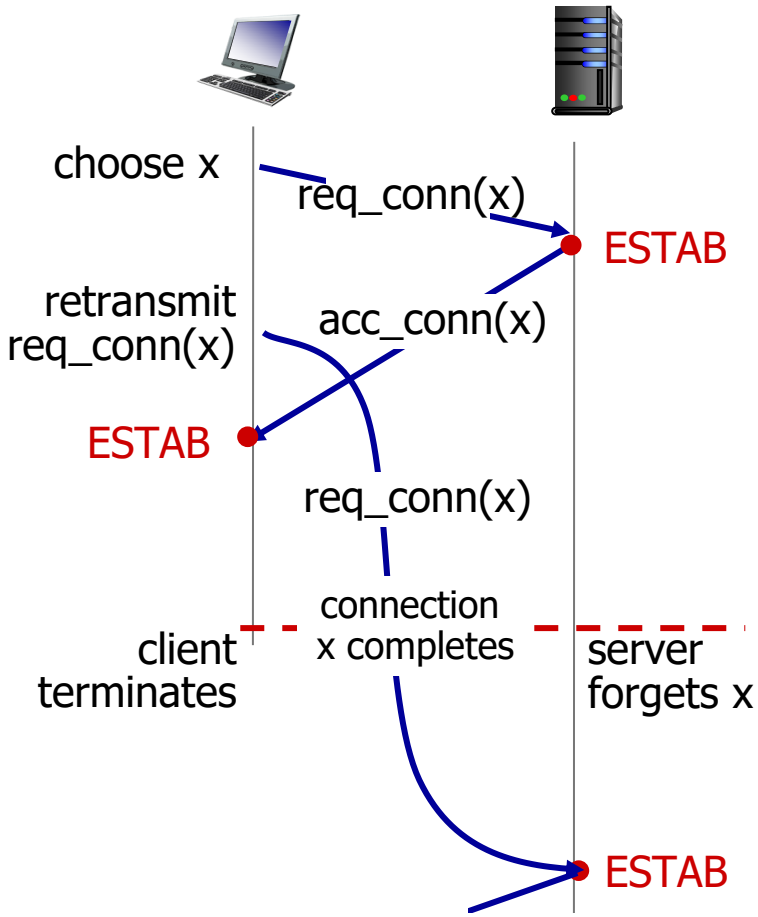
2-way handshake scenarios



No problem!

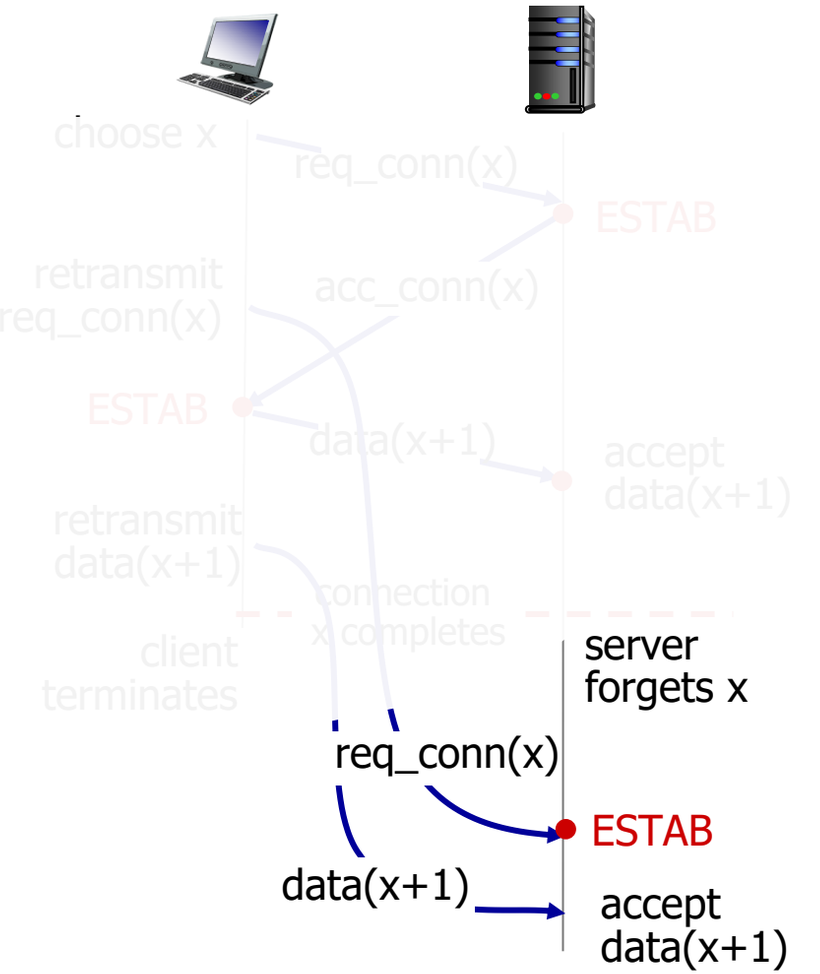


2-way handshake scenarios



Problem: half open connection! (no client)

2-way handshake scenarios



Problem: dup data accepted!

TCP 3-way handshake

Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName,serverPort))
```



choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x



choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

ESTAB

Server state

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

LISTEN

SYN RCVD

ESTAB

SPEAKER NOTES

TCP's three way handshake, that operates as follows

Let's say the client and server both create a TCP socket as we learned about in Chapter 2 and enter the LISTEN state

The client then connects to the server sending a SYN message with a sequence number x (SYN Message is an TCP Segment with SYN but set in the header – you might want to go back and review the TCP segment format!)

The server is waiting for a connection, and receives the SYN message enters the SYN received state (NOT the established state and sends a SYN ACK message back.

Finally the client sends an ACK message to the server, and when the server receiver this enters the Established state. This is when the application process would see the return from the wait on the socket accept() call

A human 3-way handshake protocol



SPEAKER NOTES

As usual, there's a human protocol analogy to the three way handshake, and I still remember thinking about this clinging for my life while climbing up a rockface

When you want start climbing you first say ON BELOW (meaning ARE YOU READY WITH MY SAFETY ROPE)
THE BELYER (server) responds BELAY ON (that lets you know the belayer is ready for you)
And then you say CLIMING

It's amazing what can pass through your head when your clinging for your life o a

Closing a TCP connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

SPEAKER NOTES

All good things must come to an end, and that's true for a TCP connection as well.

And of course there's a protocol for one side to gracefully close of a TCP connection using a FIN message, to which the other side sends a FINACK message and waits around a bit to respond to any retransmitted FIN messages before timing out.

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- **Principles of congestion control**
- TCP congestion control
- Evolution of transport-layer functionality



Troy University Resources

Principles of congestion control

Congestion:

- informally: “too many sources sending too much data too fast for *network* to handle”
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!





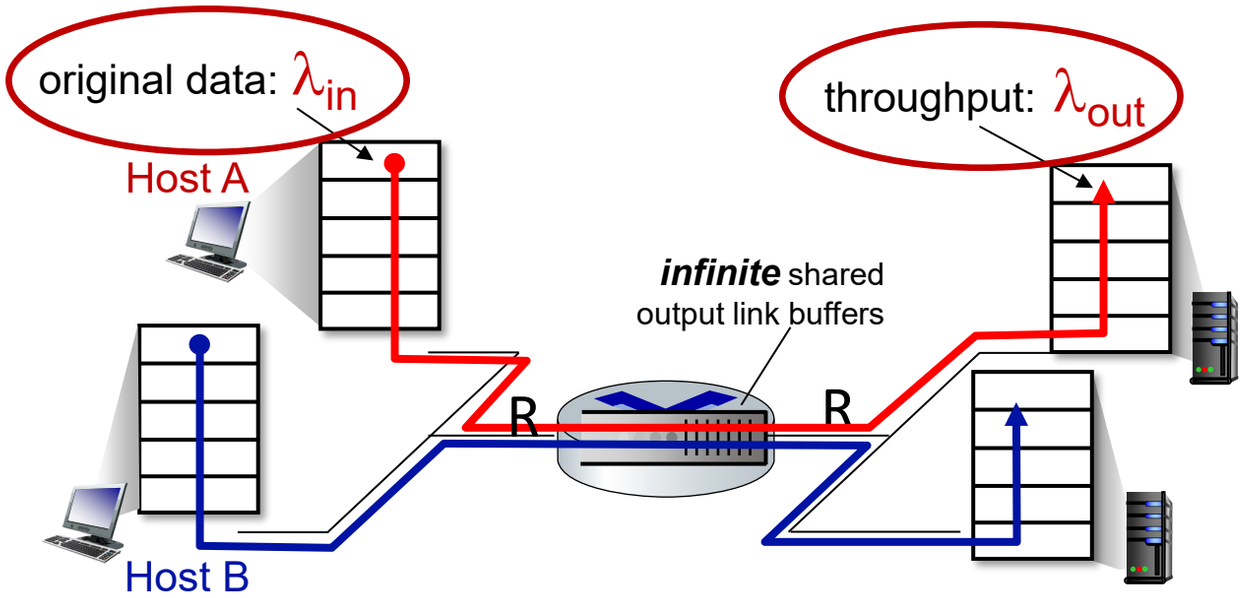
congestion control:
too many senders,
sending too fast

flow control: one sender
too fast for one receiver

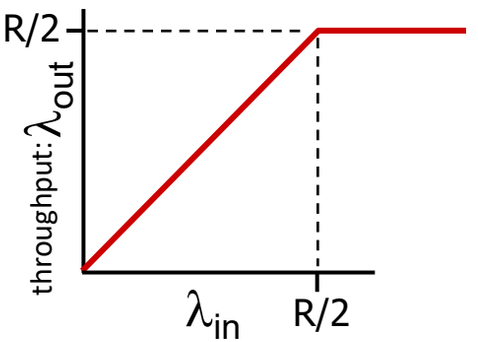
Causes/costs of congestion: scenario 1

Simplest scenario:

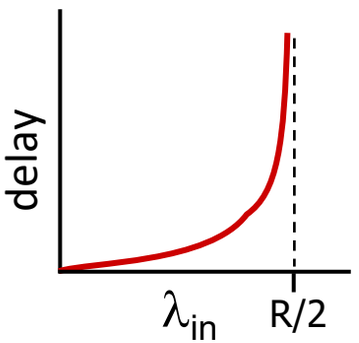
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed



Q: What happens as arrival rate λ_{in} approaches R/2?



maximum per-connection throughput: R/2

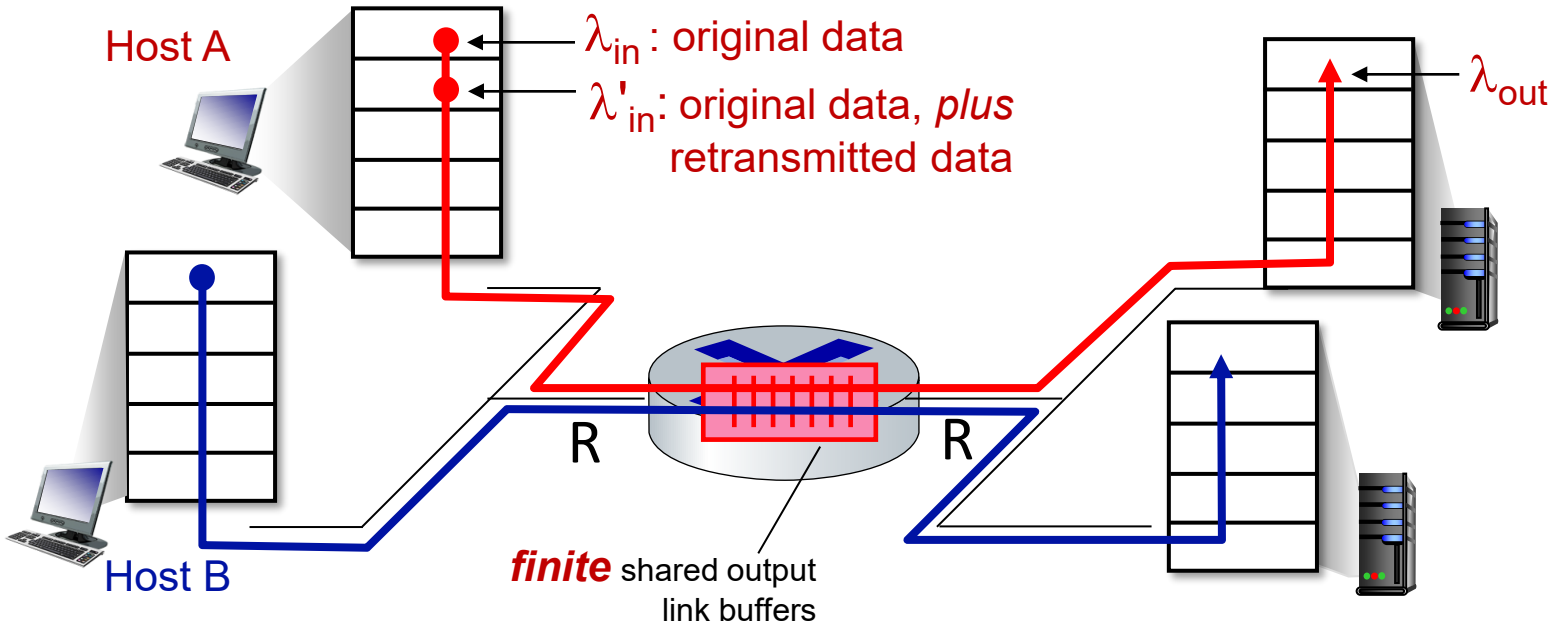


large delays as arrival rate λ_{in} approaches capacity

Troy University Resources

Causes/costs of congestion: scenario 2

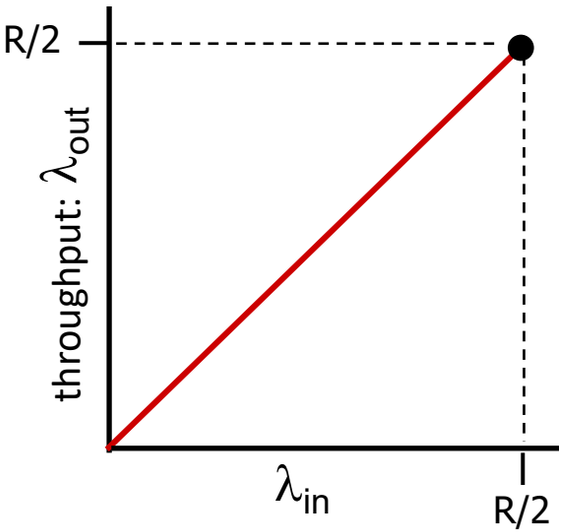
- one router, *finite* buffers
- sender retransmits lost, timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

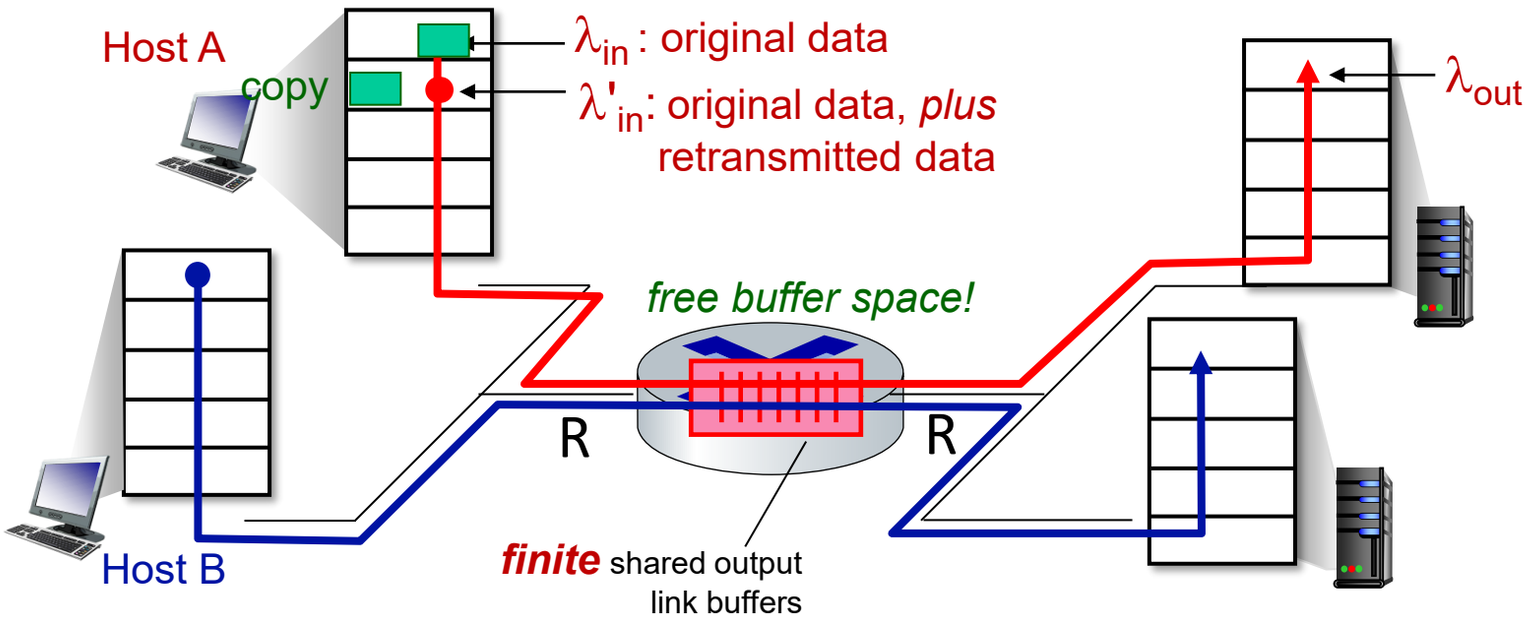


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available



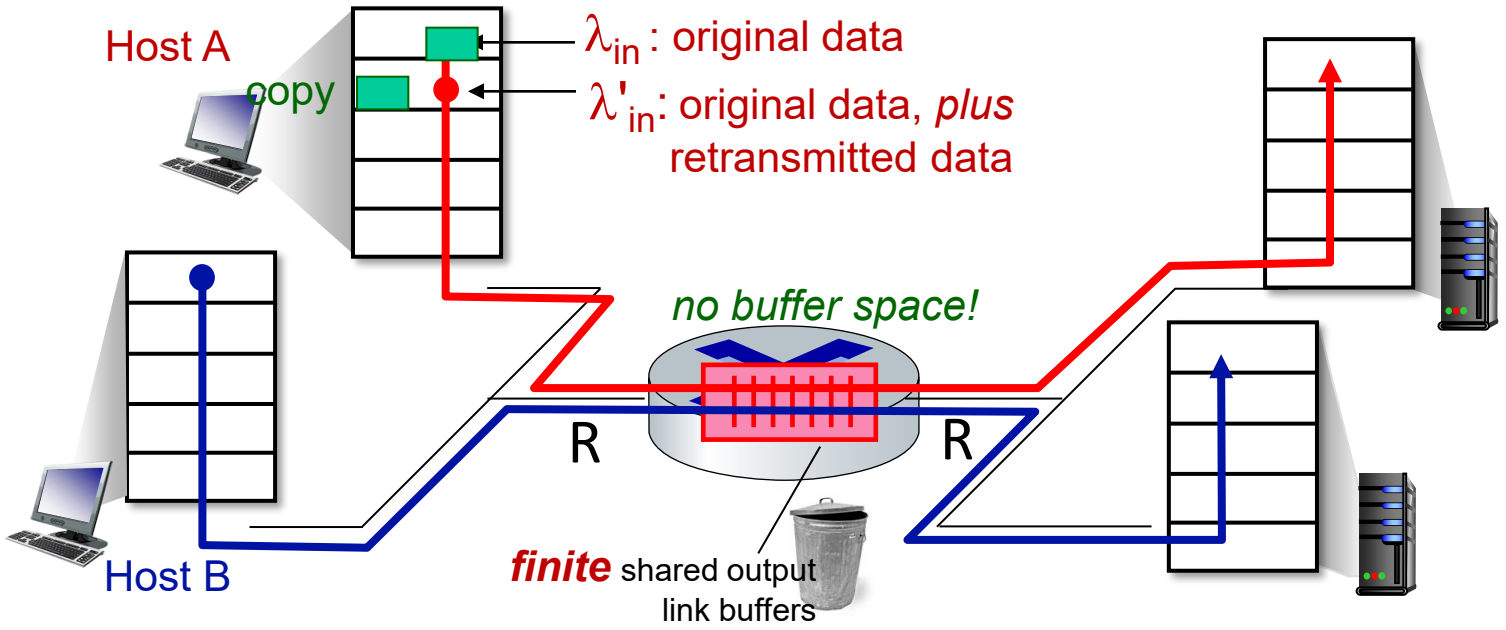


Troy University Resources

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

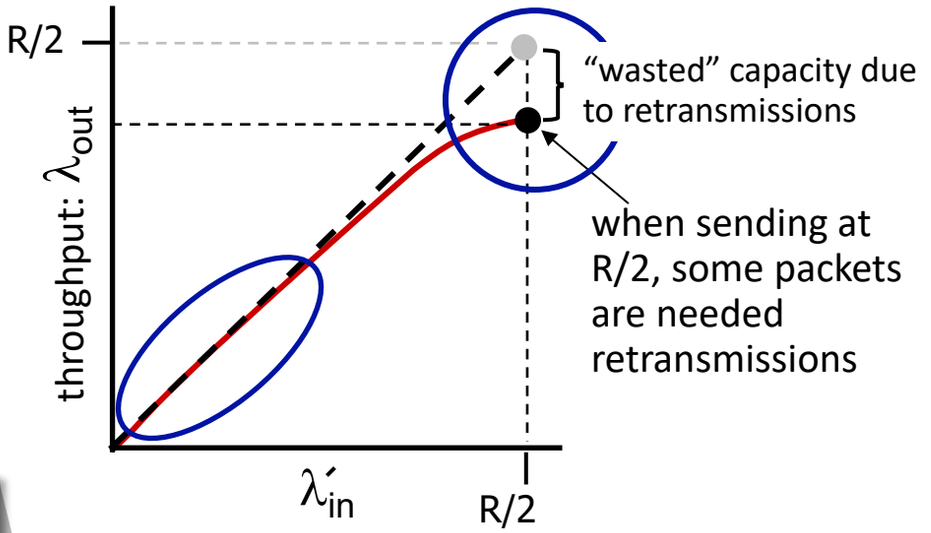
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost

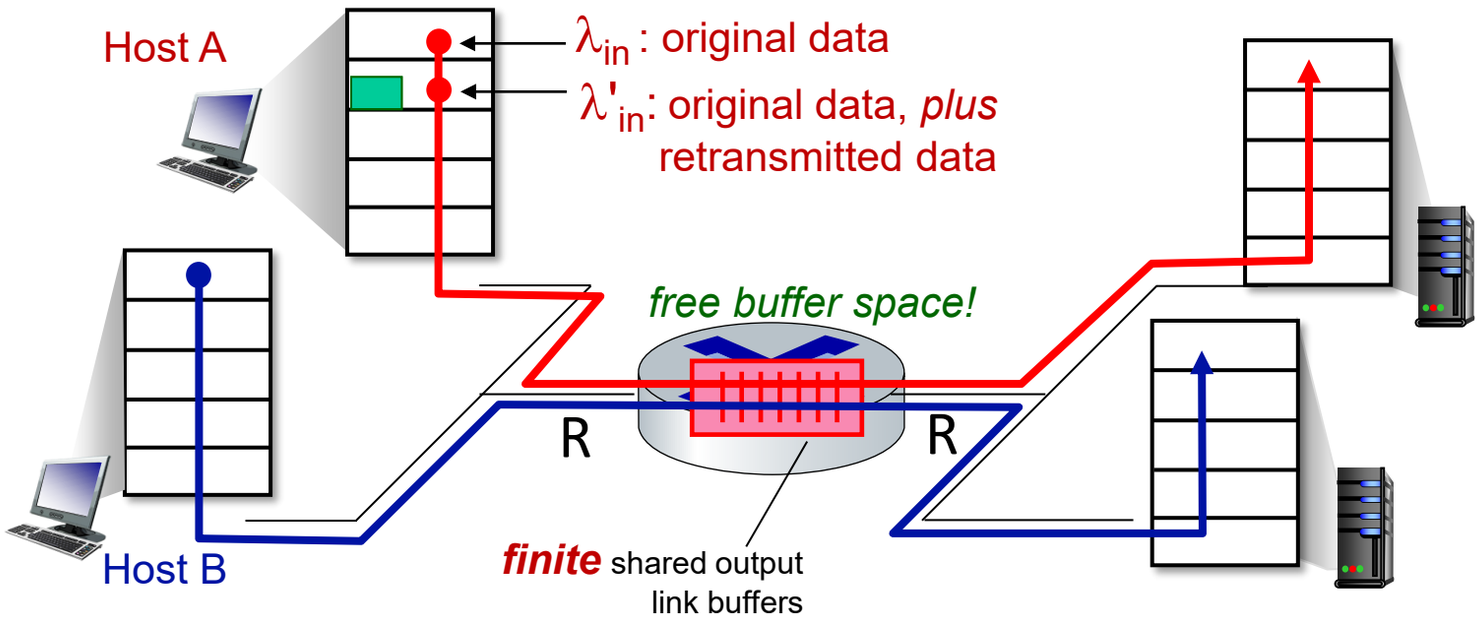


Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



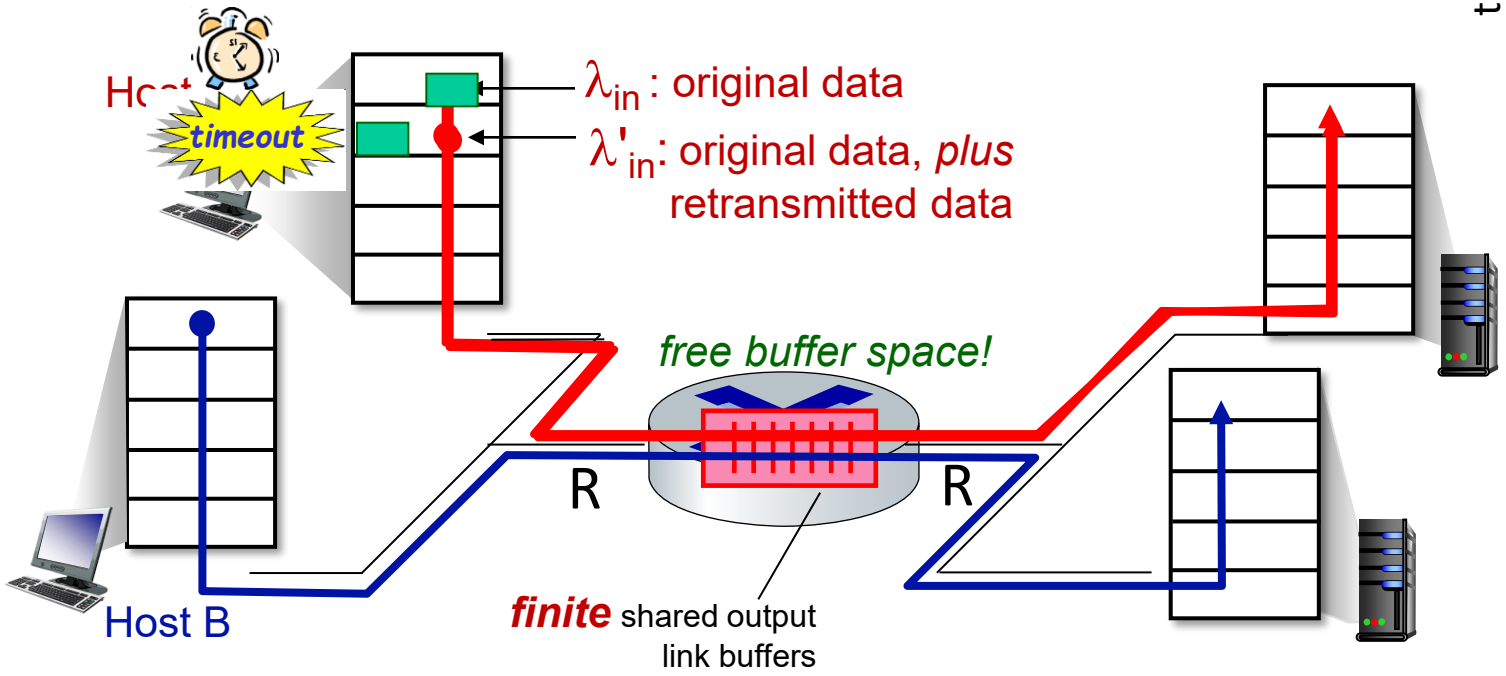


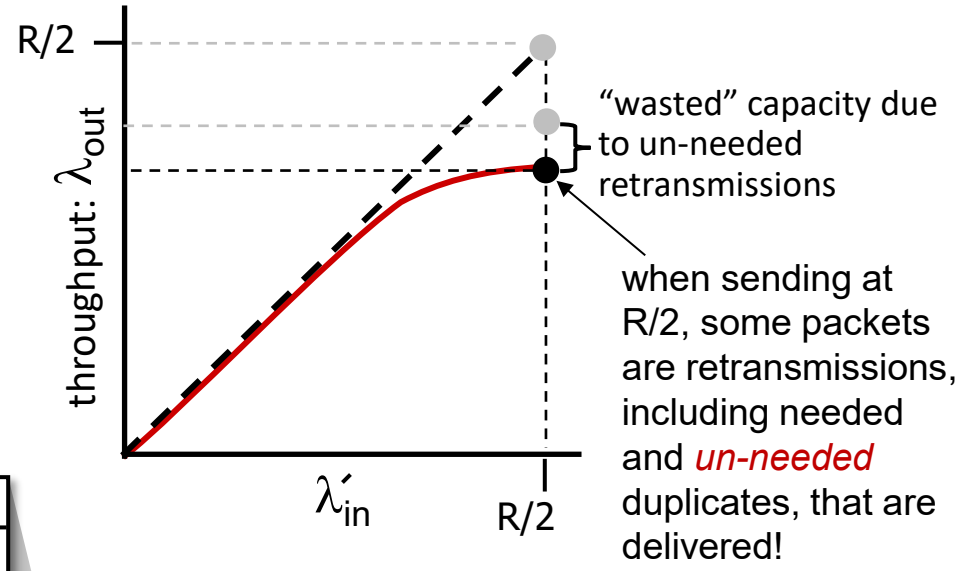
Troy University Resources

Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered

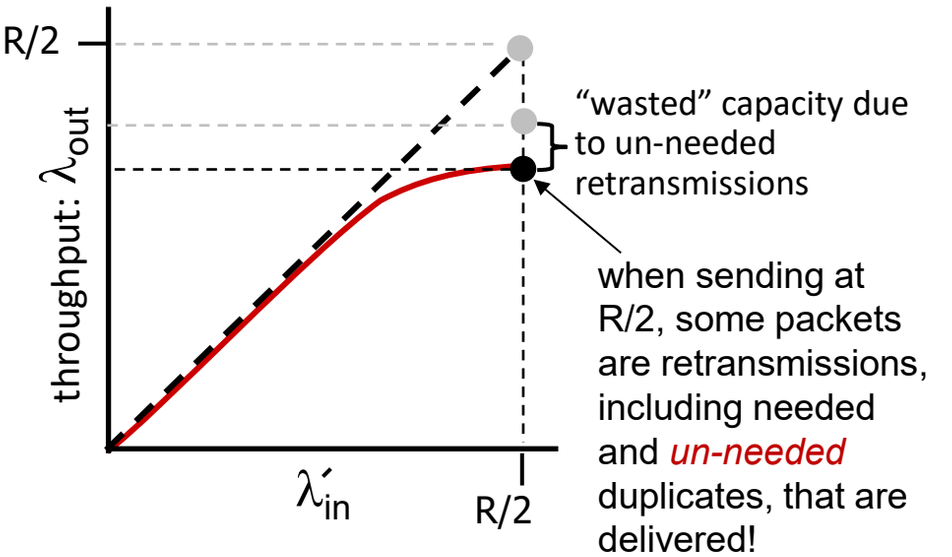




Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



"costs" of congestion:

- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

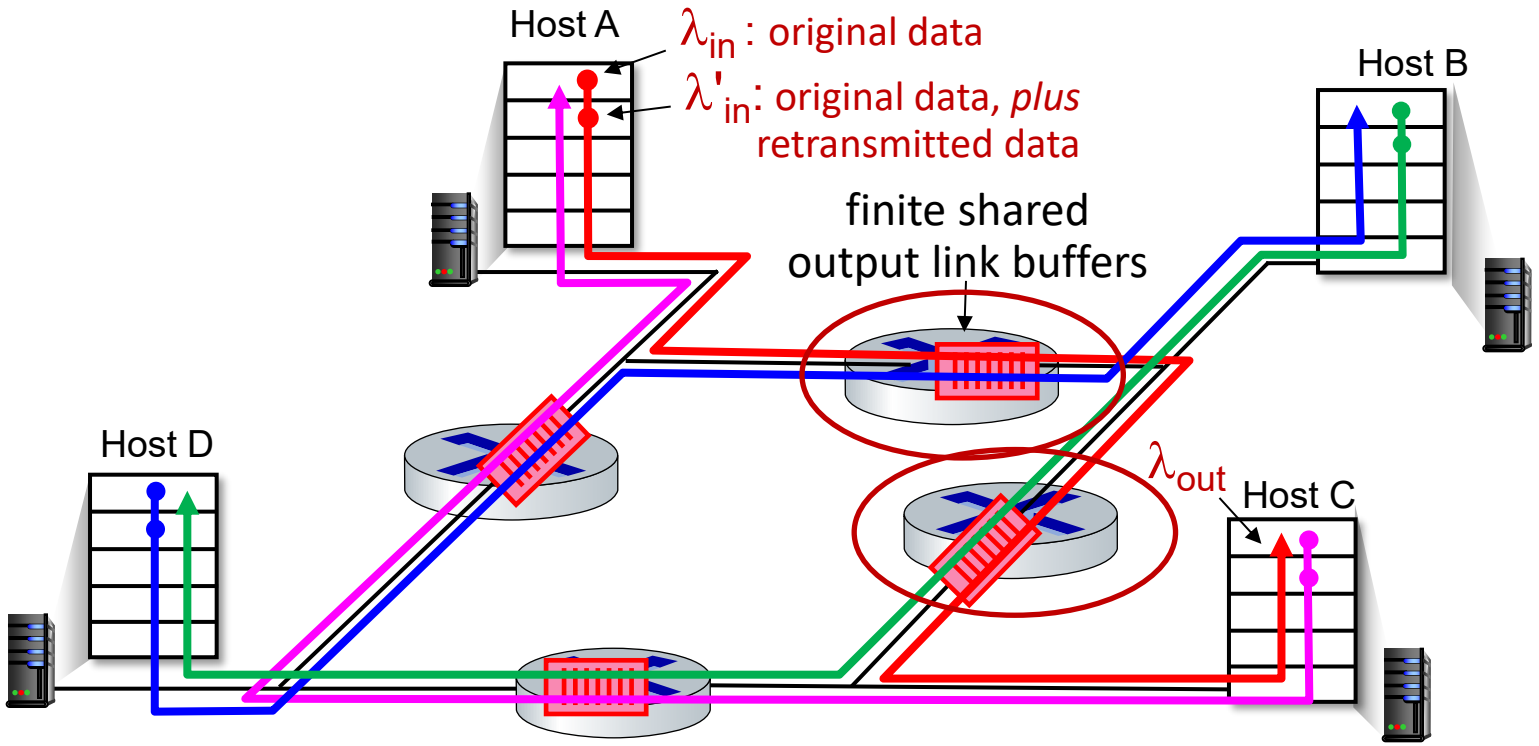
Troy University Resources

Causes/costs of congestion: scenario 3

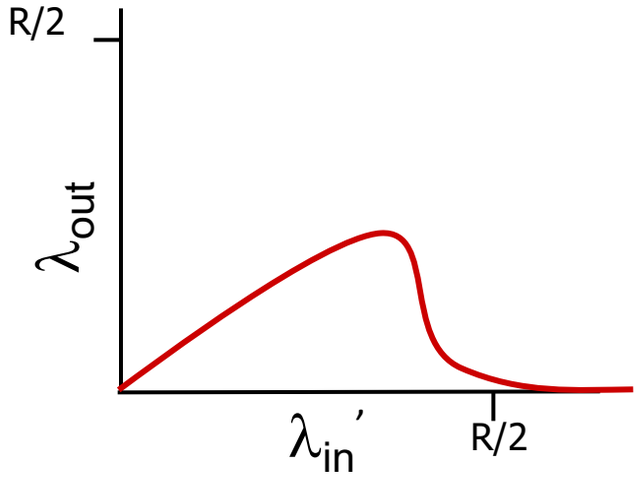
- *four* senders
- *multi-hop* paths
- timeout/retransmit

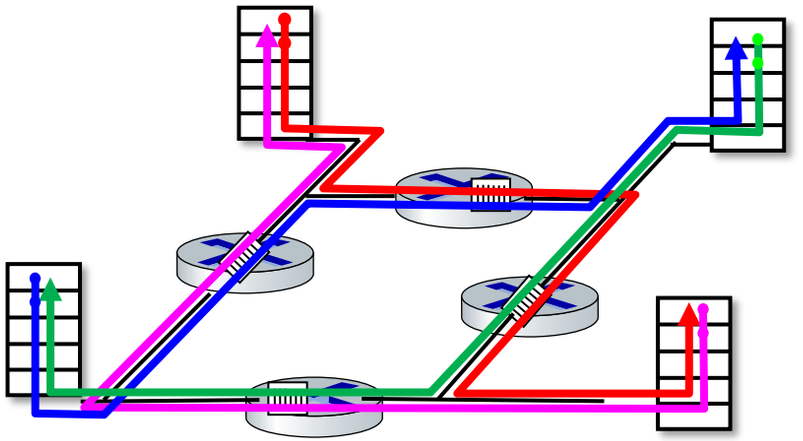
Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



Causes/costs of congestion: scenario 3





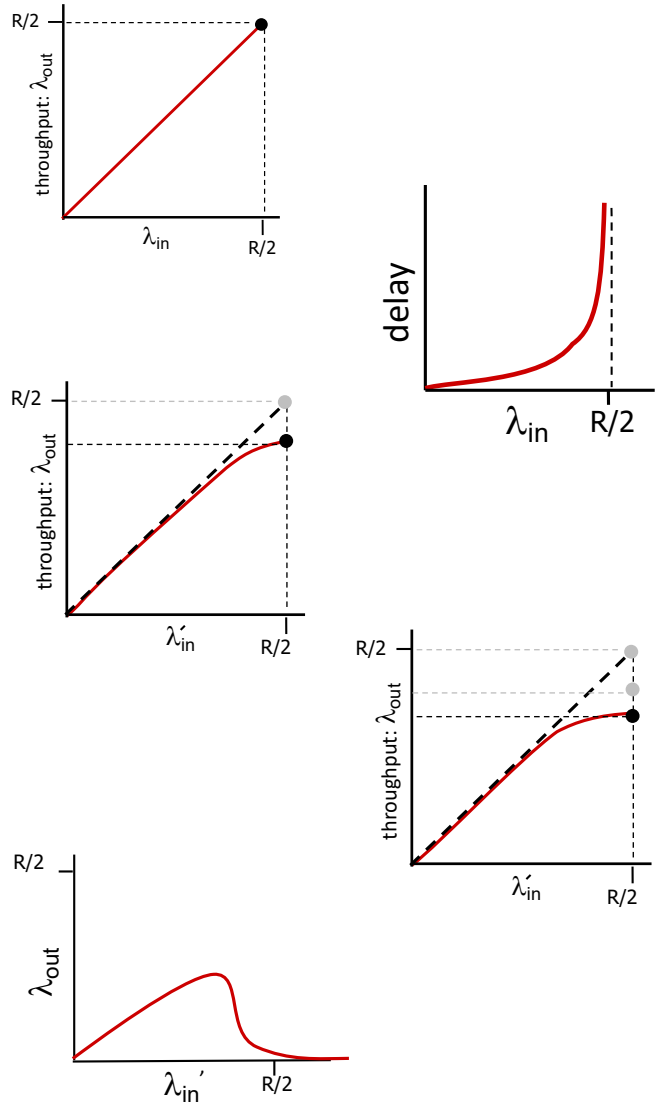
another "cost" of congestion:

- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

Troy University Resources

Causes/costs of congestion: insights

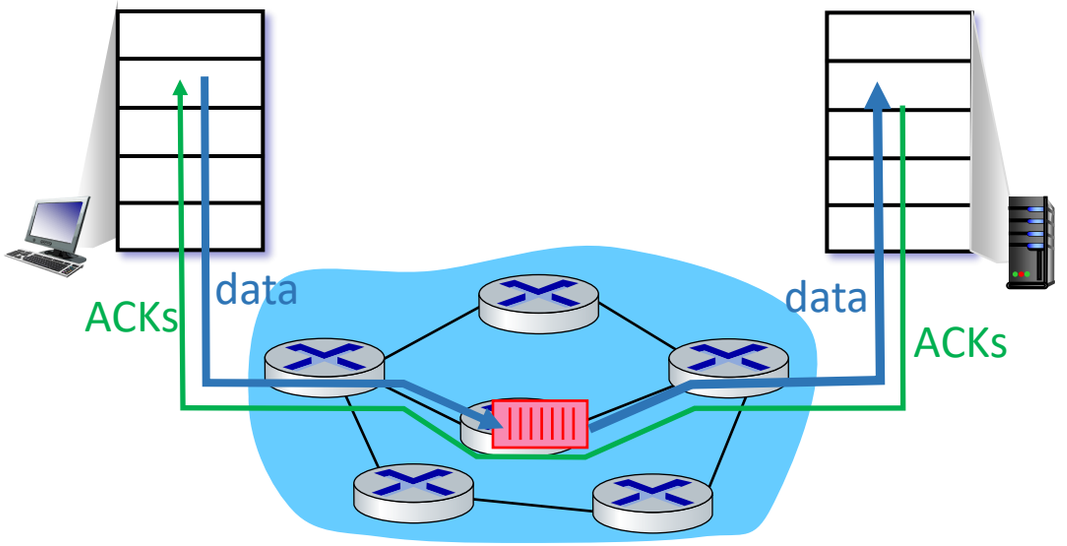
- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



Approaches towards congestion control

End-end congestion control:

- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP

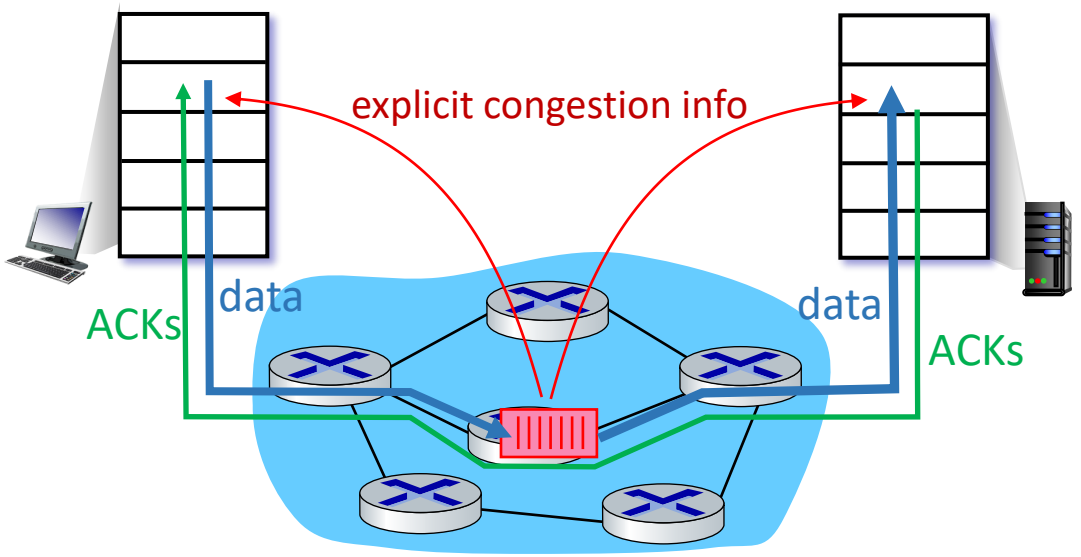


Troy University Resources

Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



Troy University Resources

TCP congestion control: AIMD

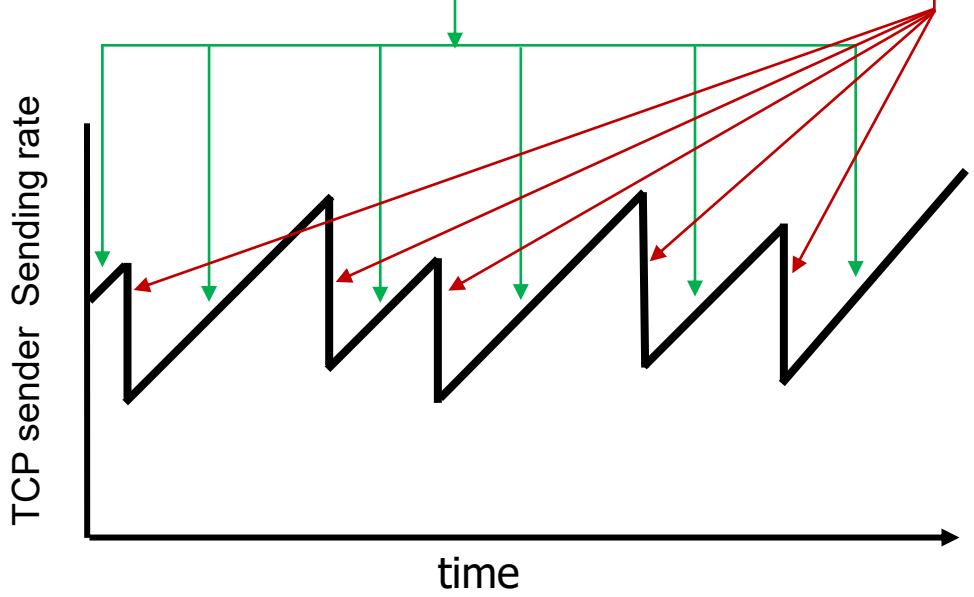
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

Multiplicative decrease detail: sending rate is

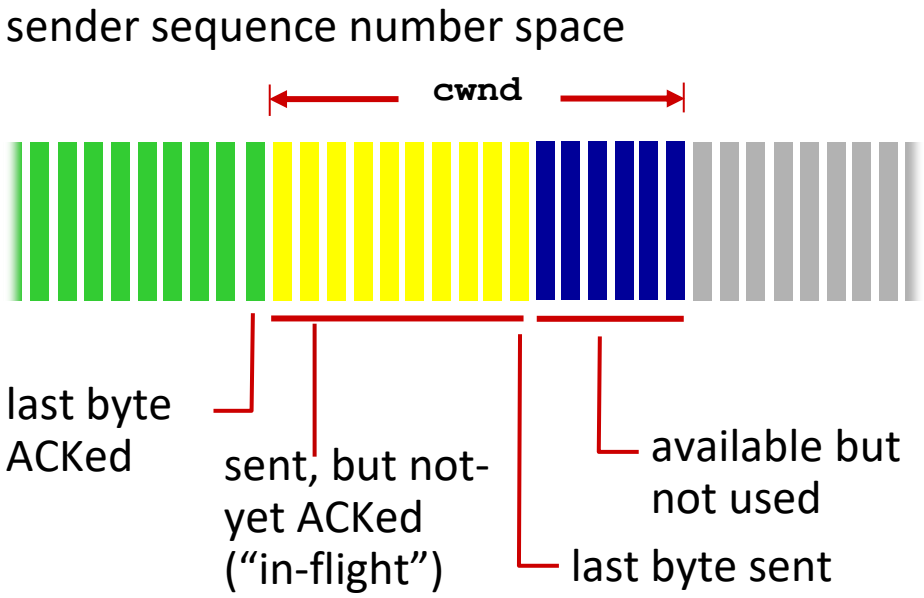
- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide [Kelly]!
  - have desirable stability properties

Troy University Resources

TCP congestion control: details



TCP sending behavior:

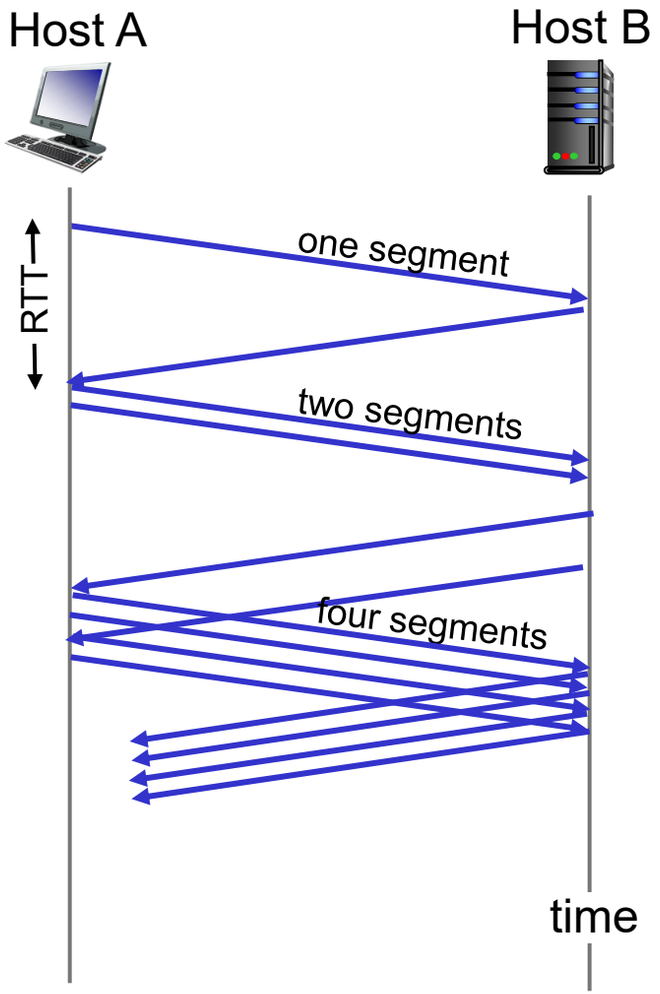
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- TCP sender limits transmission: `LastByteSent- LastByteAcked ≤ cwnd`
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



Troy University Resources

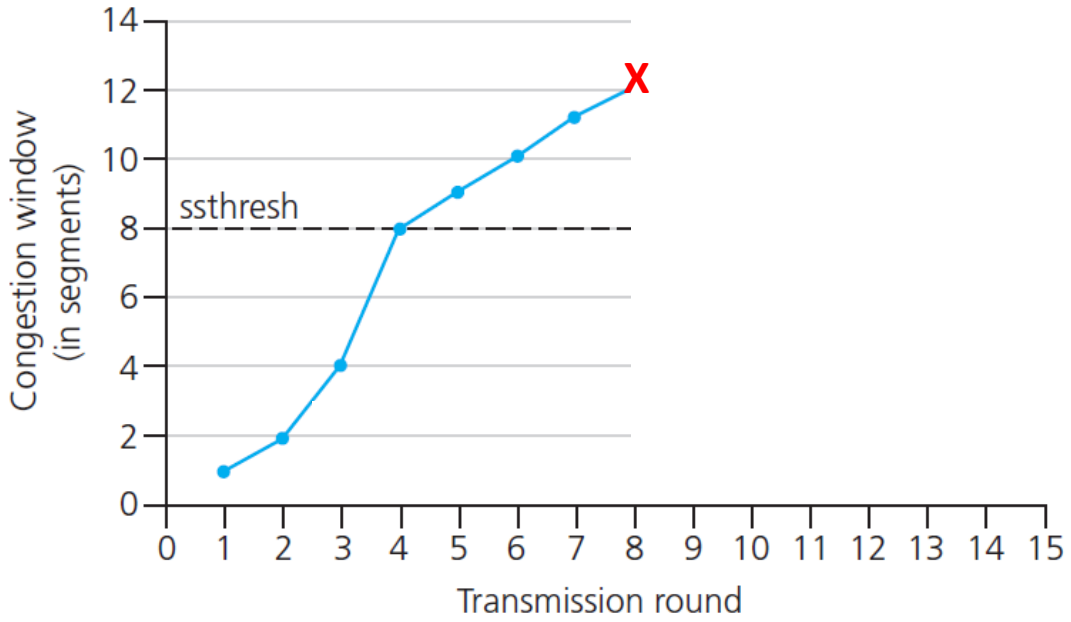
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

A: when **`cwnd`** gets to 1/2 of its value before timeout.

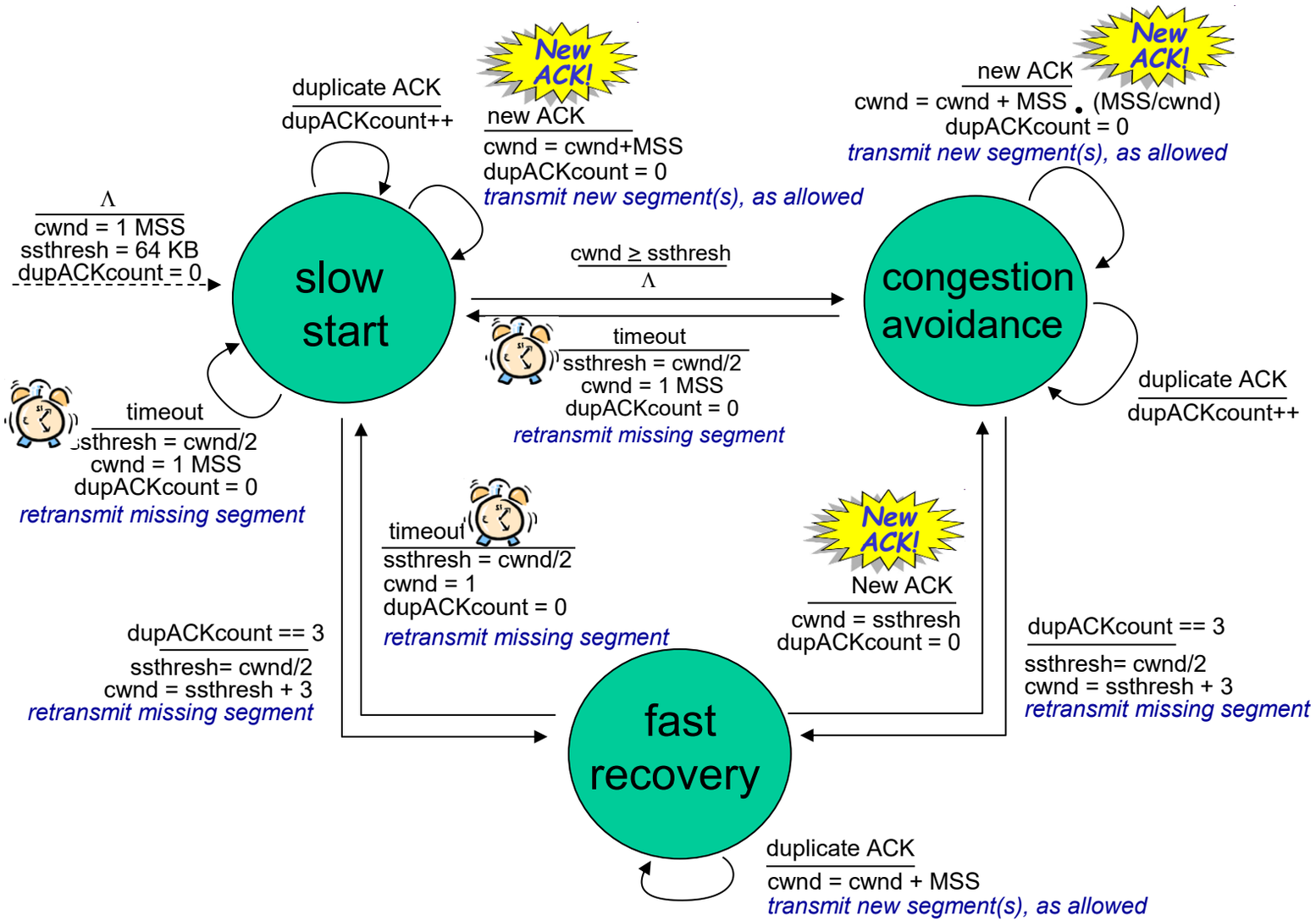
Implementation:

- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event



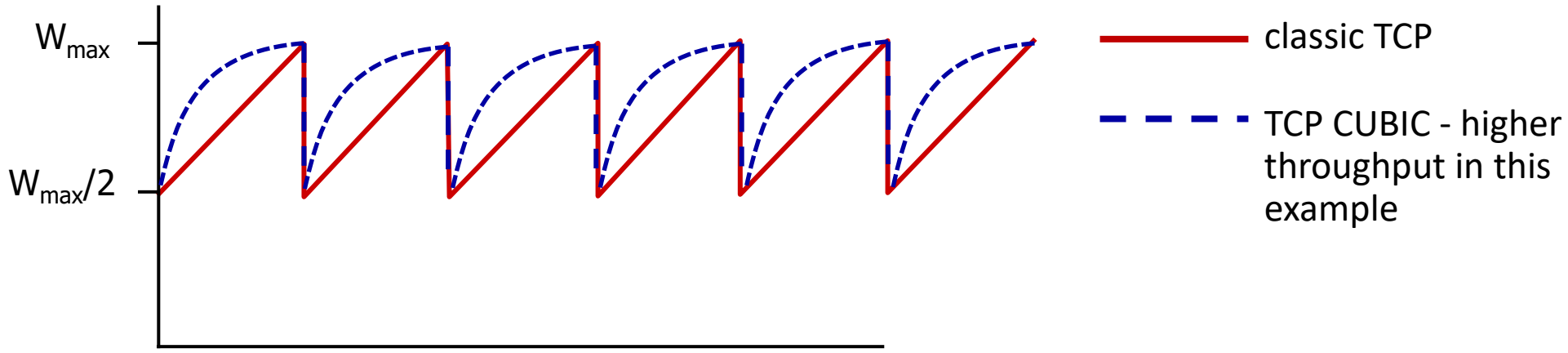
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Summary: TCP congestion control



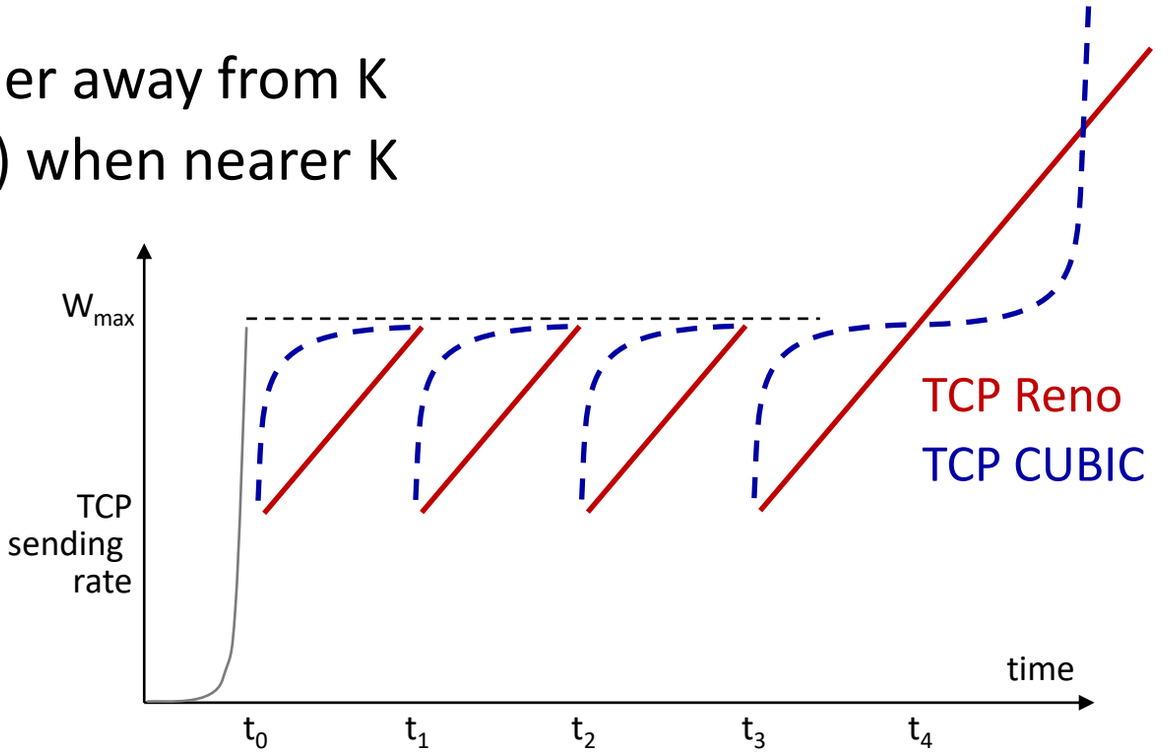
TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



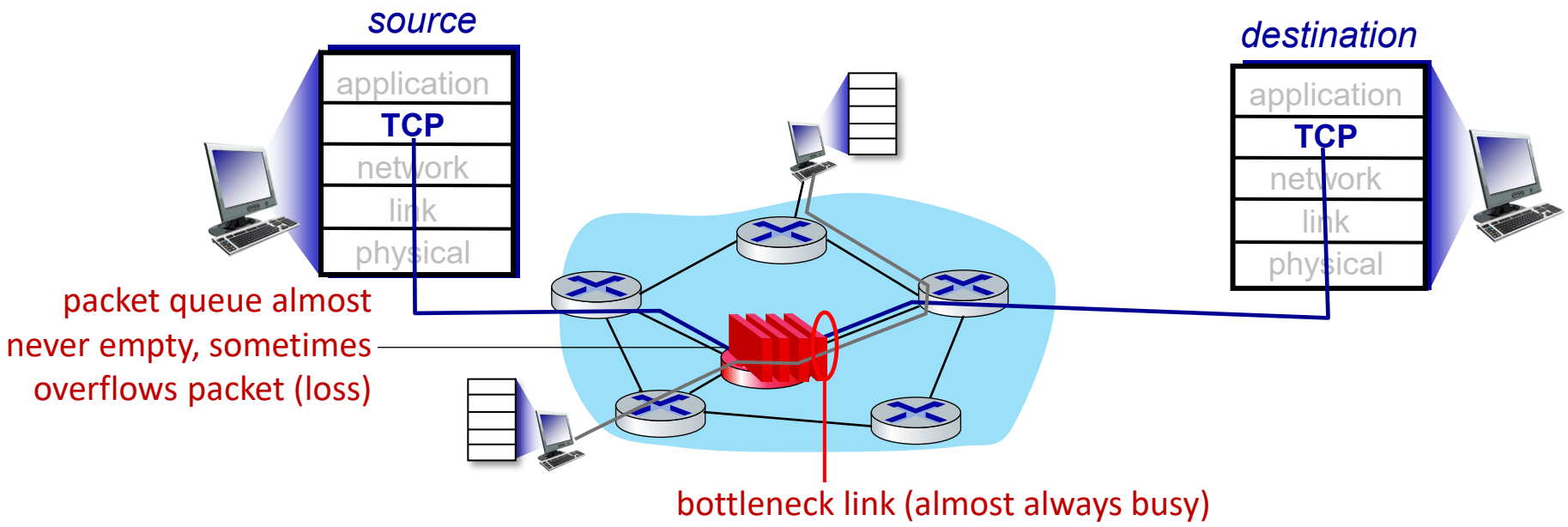
TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tunable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers (until ~2024)



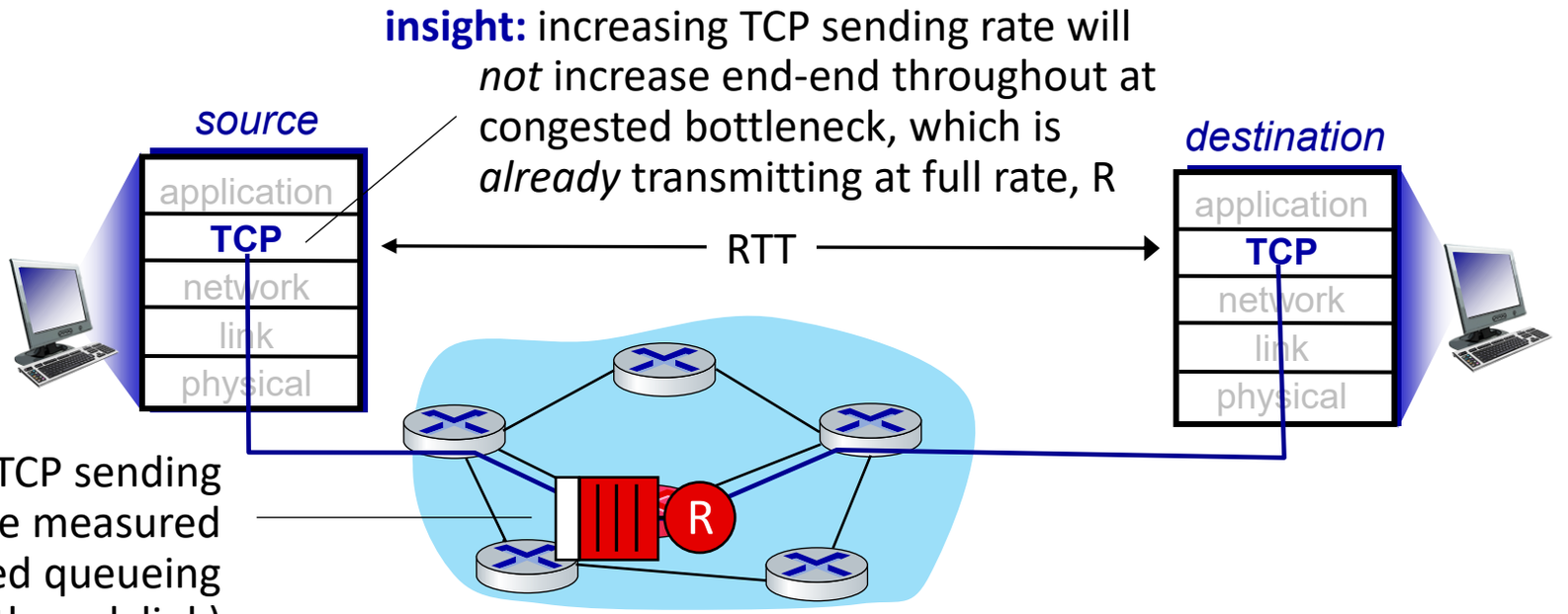
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link

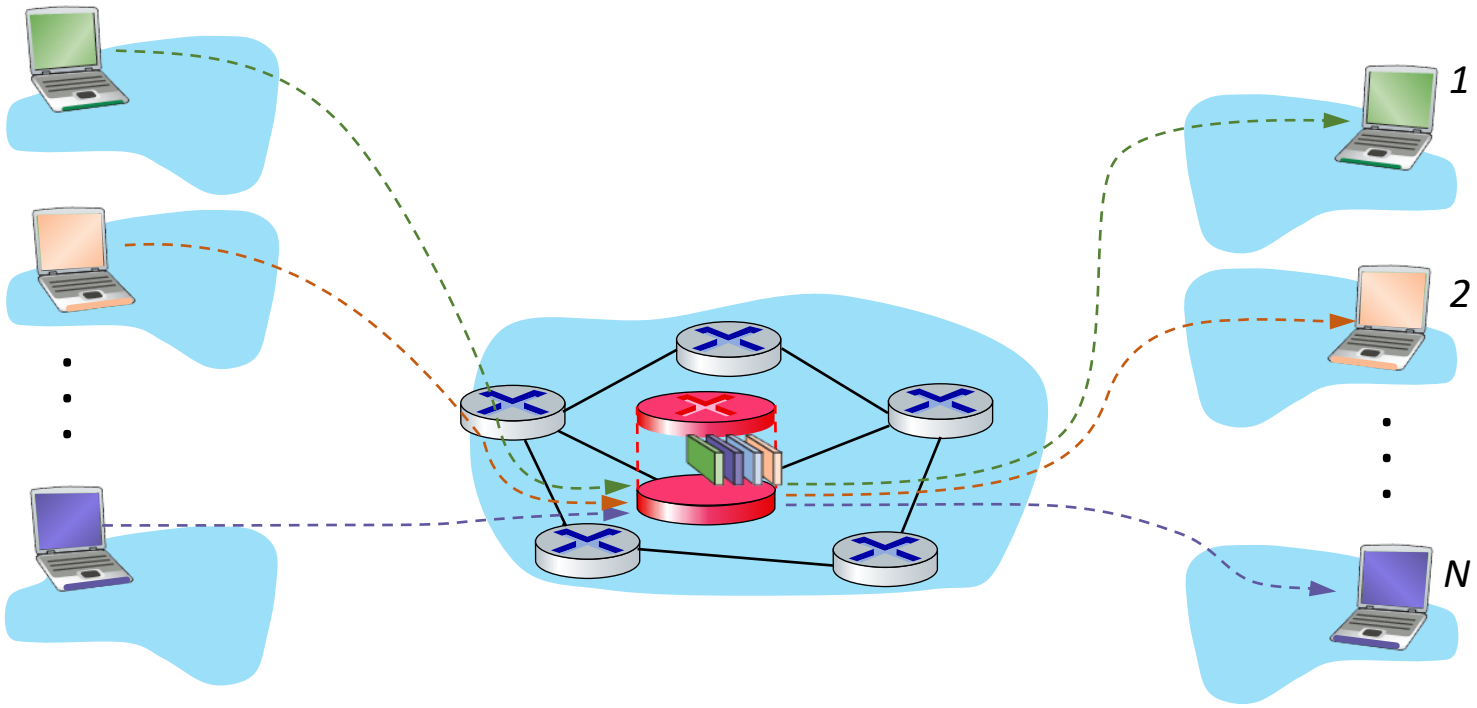


Goal: *"keep the end-end pipe just full, but not fuller"*

Troy University Resources

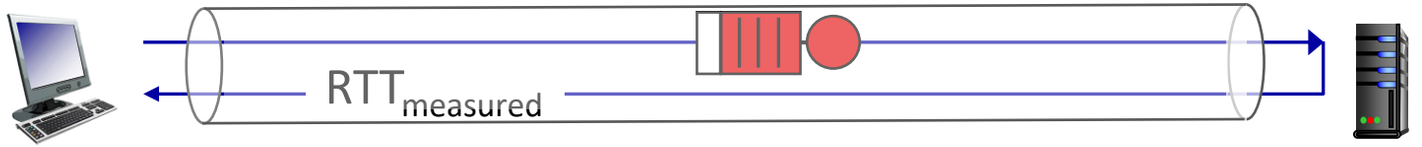
TCP and the congested "bottleneck link"

- multiple flows collectively congest bottleneck link:
  - each their executing own TCP congestion control
  - each of *N* flows should ideally receive throughput of *R/N*



Keeping the pipe just full enough

Keeping pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



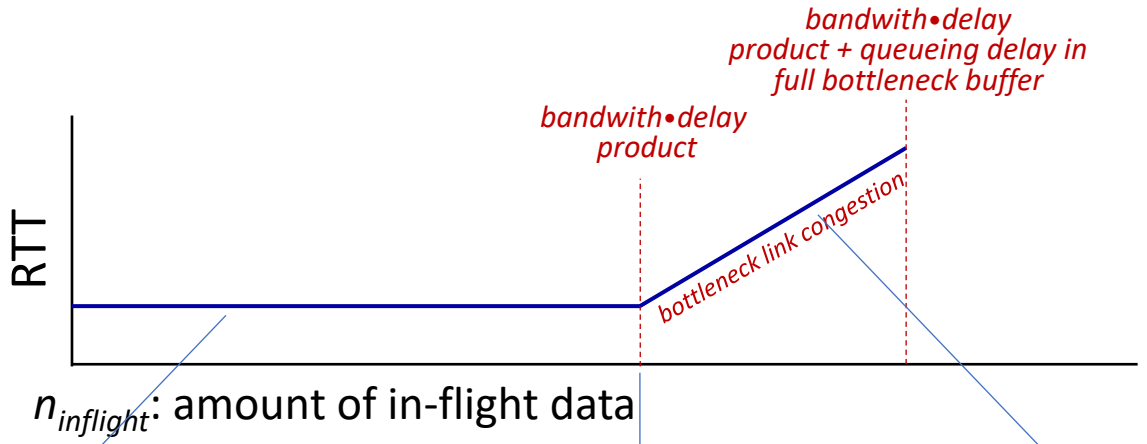
$$\text{measured throughput} = \frac{\text{\# bytes sent (inflight) in RTT interval}}{\text{RTT}}$$

Intuition:

$$\underbrace{\text{ideal amount of inflight data for a connection}}_{n_{inflight}} = \underbrace{\textit{connection's share of bottleneck_bandwidth} \bullet \textit{min_RTT}}_{\text{connection's ``}\textit{bandwidth-delay product}\text{''}}$$

Troy University Resources

Keeping the pipe just full, but no fuller

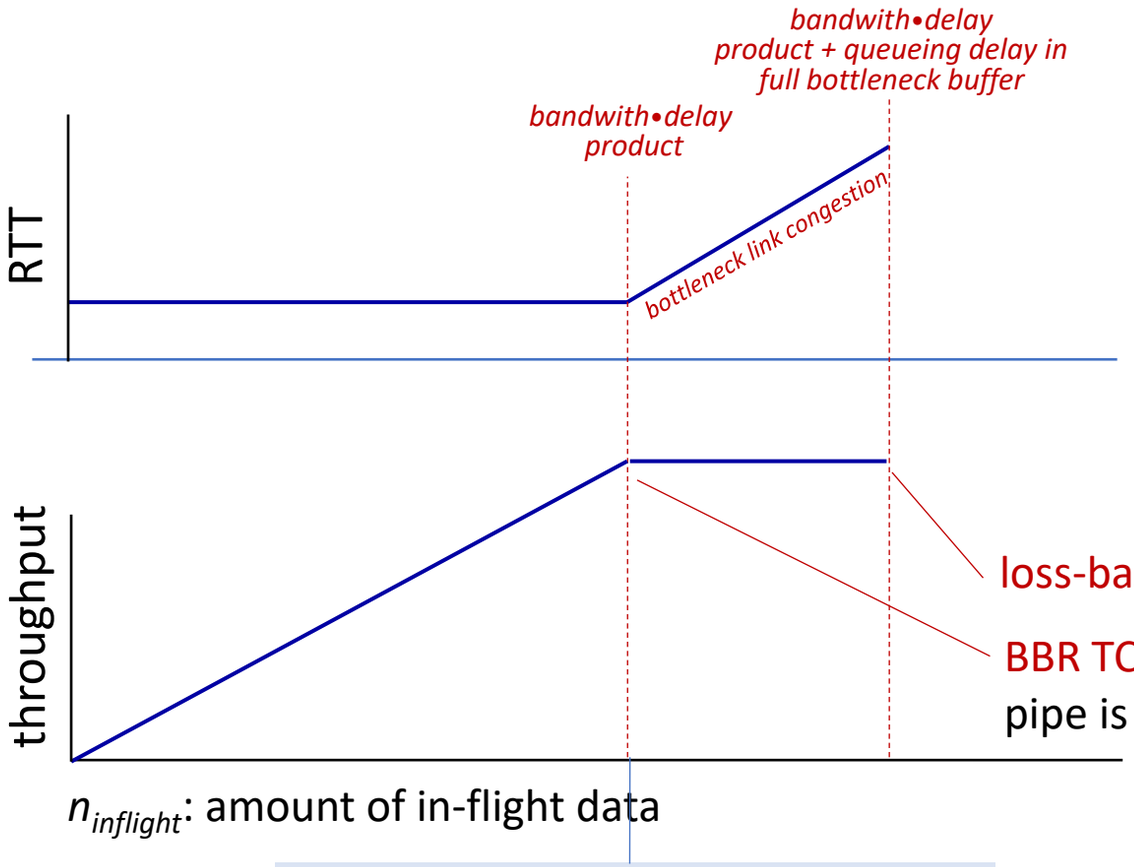


increasing $n_{inflight}$ in this region will not increase RTT, since arrival rate of packet data to queue is less that transmission rate

Here, arrival rate of packet data to bottleneck link is equals link transmission rate

Increasing $n_{inflight}$ in this region increases RTT since arrival rate of packet data to bottleneck link exceeds link transmission rate

Keeping the pipe just full, but no fuller



loss-based TCP seeks to operate here

BBR TCP seeks to operate here - the pipe is kept just full, but no fuller

Here, arrival rate of packet data to bottleneck link is equals link transmission rate

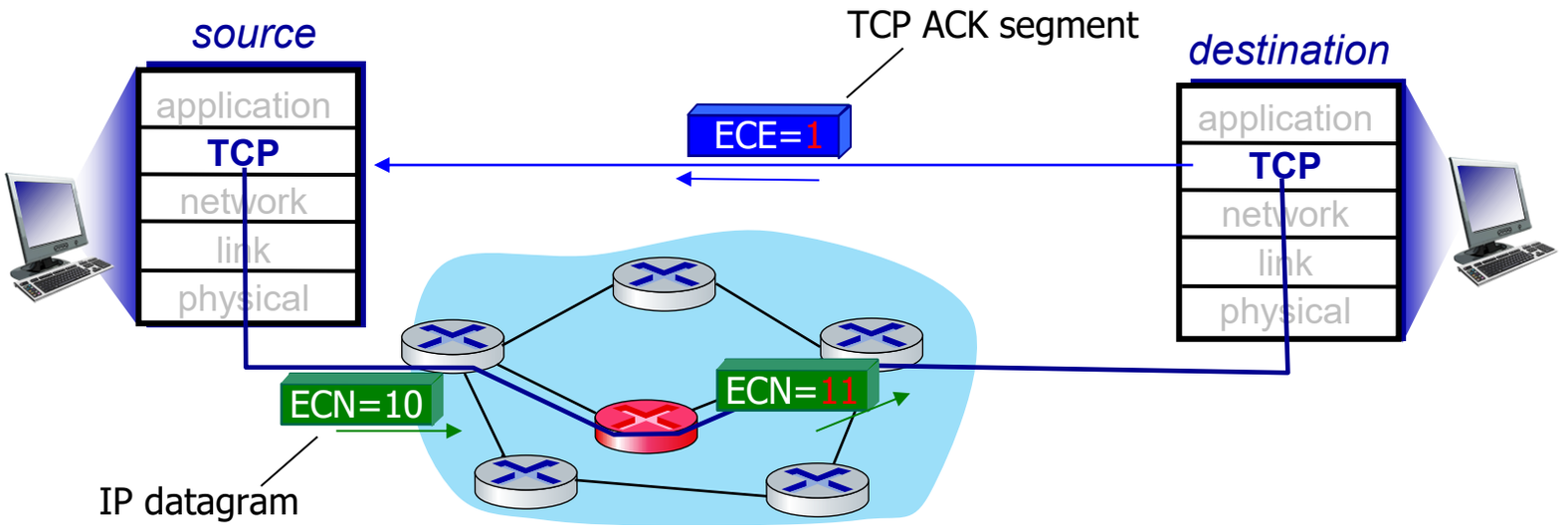
BBR (Bottleneck Bandwidth and RTT)

- regulates $n_{inflight}$
- at longer time intervals:
  - reduces $n_{inflight}$ to drain pipe, measure new *min_RTT*
- at shorter time intervals:
  - *acceleration:* increases sending rate, $n_{inflight}$, until reaches throughput plateau, (saturating available per-flow link capacity)
  - *cruising:* sends at the rate that network is delivering data, as evidenced through received ACKs.
  - *deceleration:* purposefully reduces $n_{inflight}$, decreasing queue pressure, looking for lower *min_RTT*

Explicit congestion notification (ECN)

TCP deployments often implement *network-assisted* congestion control:

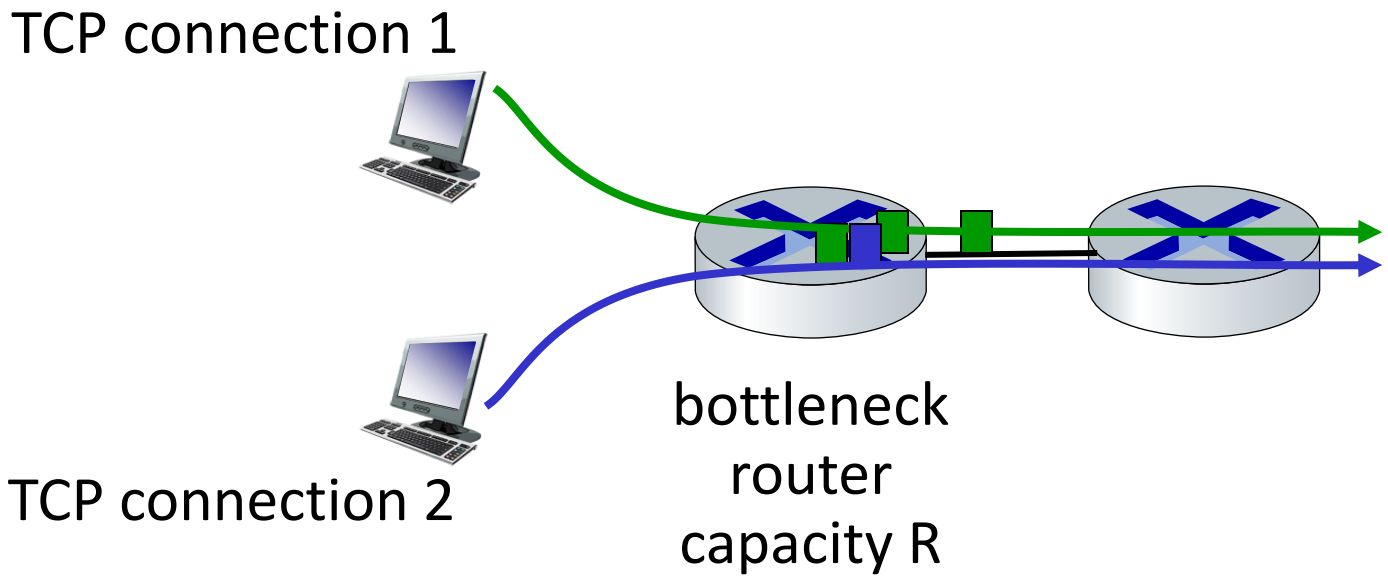
- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



Troy University Resources

TCP fairness

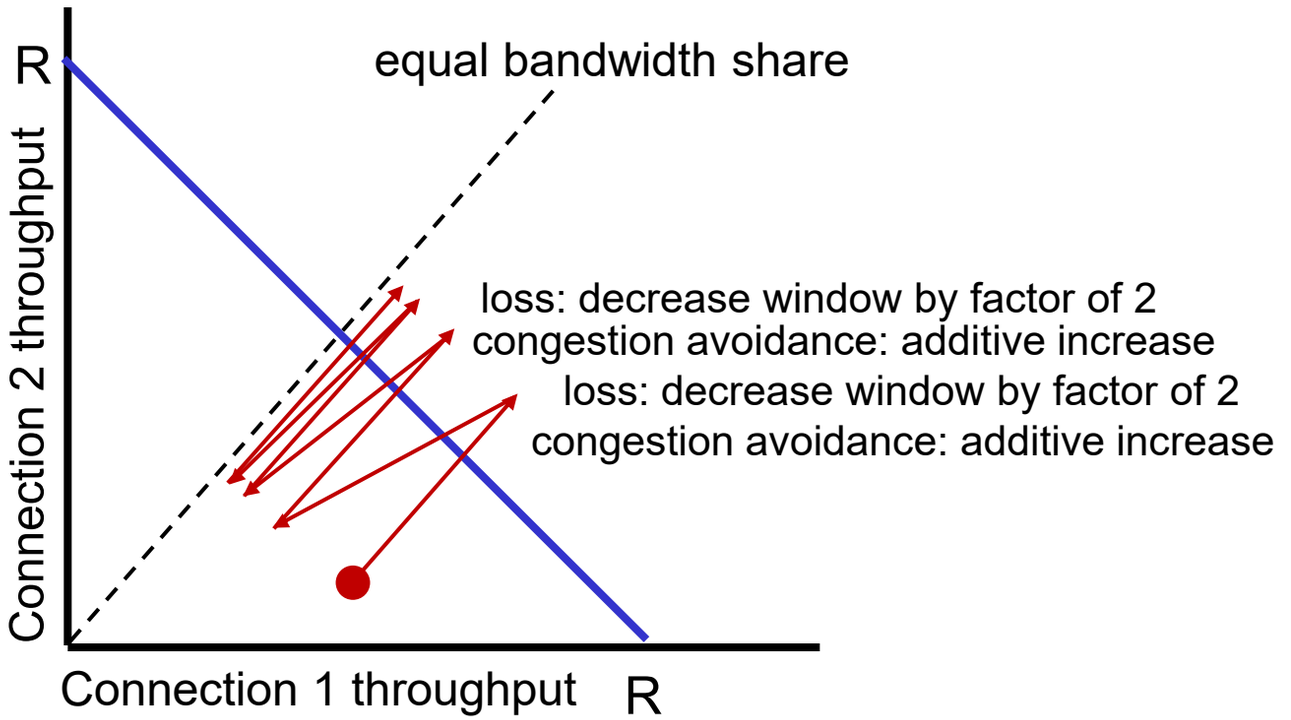
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Troy University Resources

Fairness: must all network apps be "fair"?

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no "Internet police" policing use of congestion control

Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



Troy University Resources

Evolving transport-layer functionality

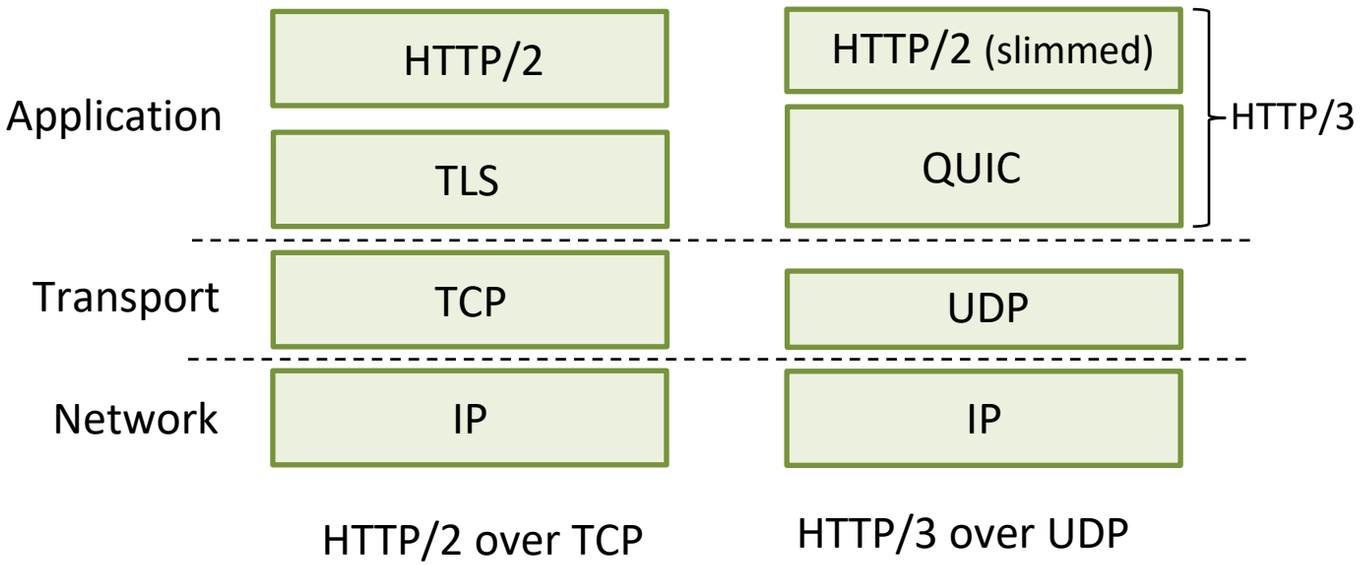
- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)



Troy University Resources

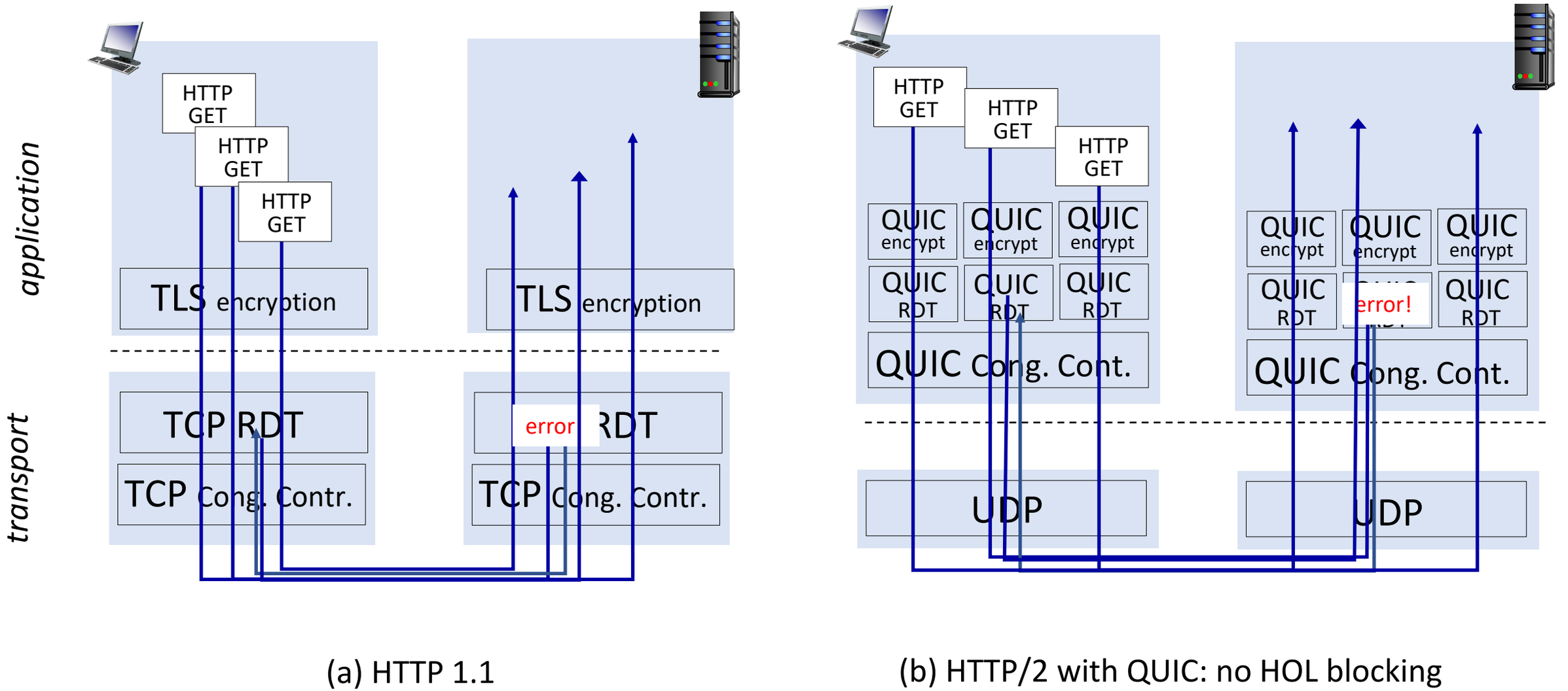
QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one (or even zero) RTT (recall HTTP3 discussion in Chapter 2)

- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

(b) HTTP/2 with QUIC: no HOL blocking

Troy University Resources

Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

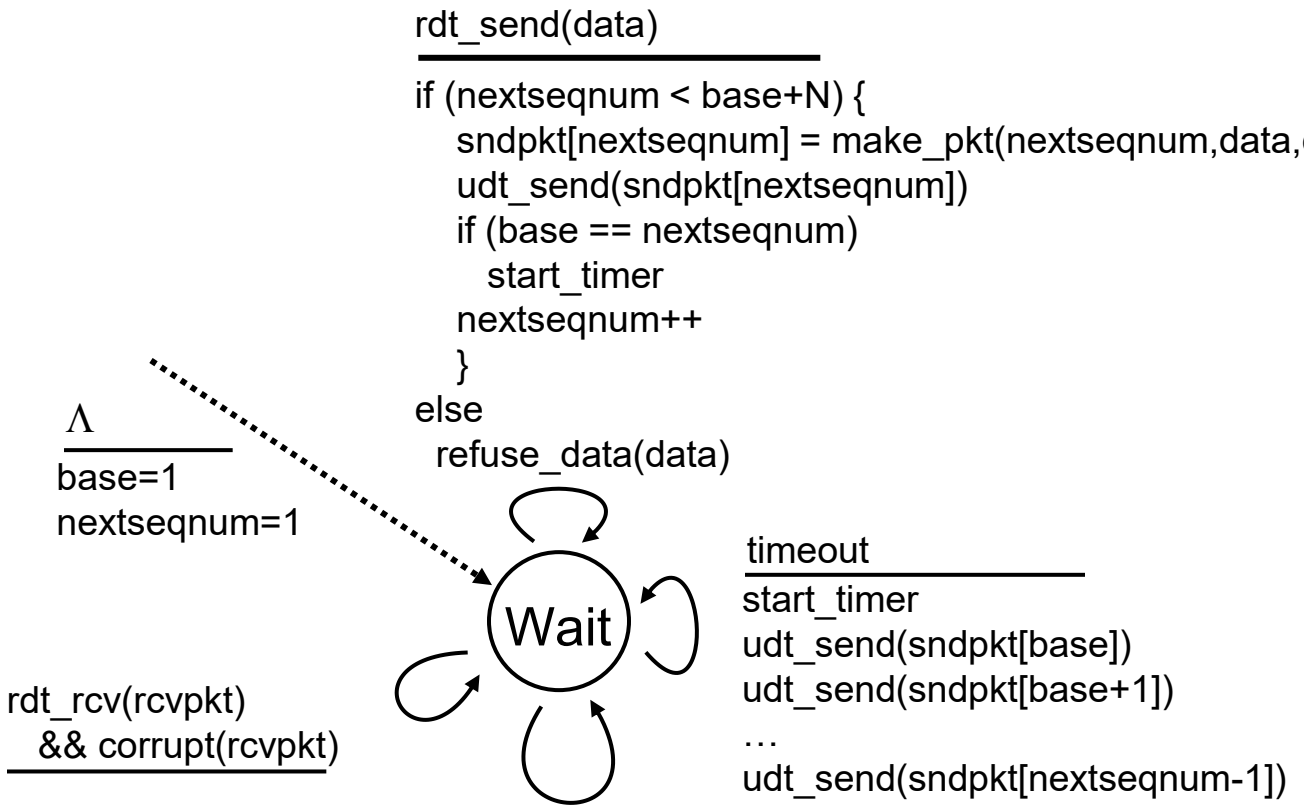
Up next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane

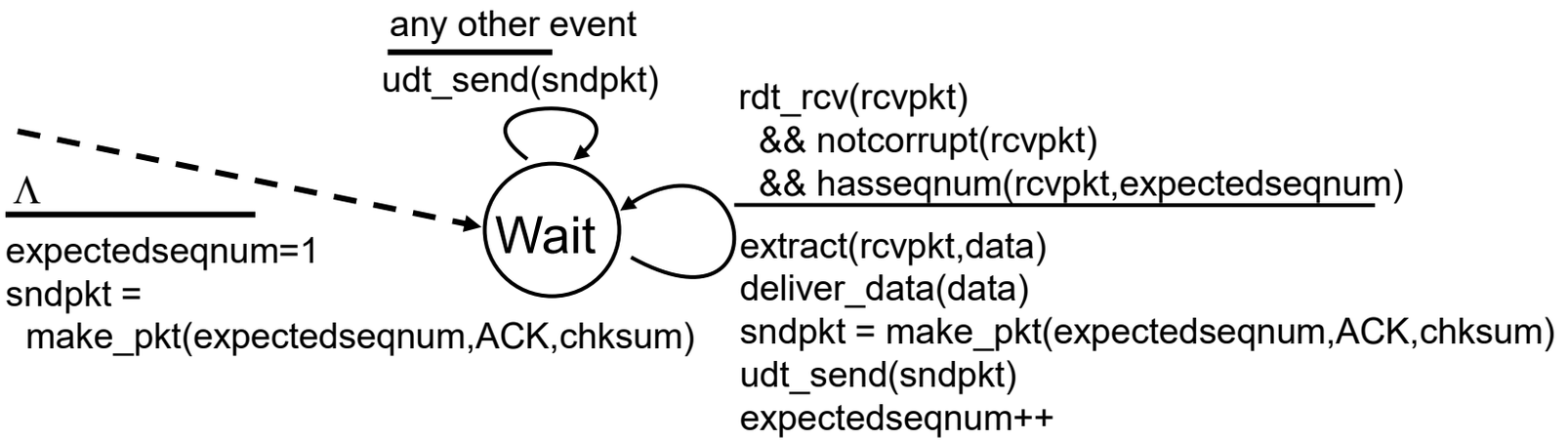
Additional Chapter 3 slides

Troy University Resources

Go-Back-N: sender extended FSM



Go-Back-N: receiver extended FSM



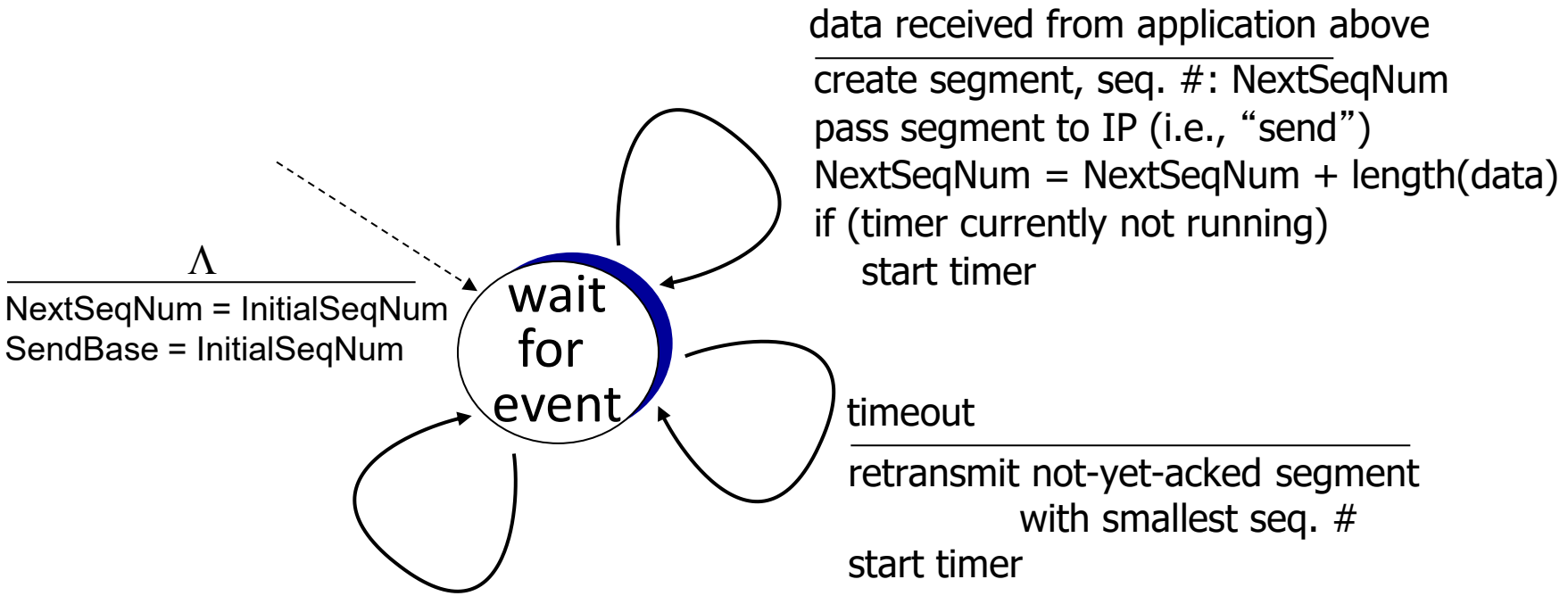
ACK-only: always send ACK for correctly-received packet with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

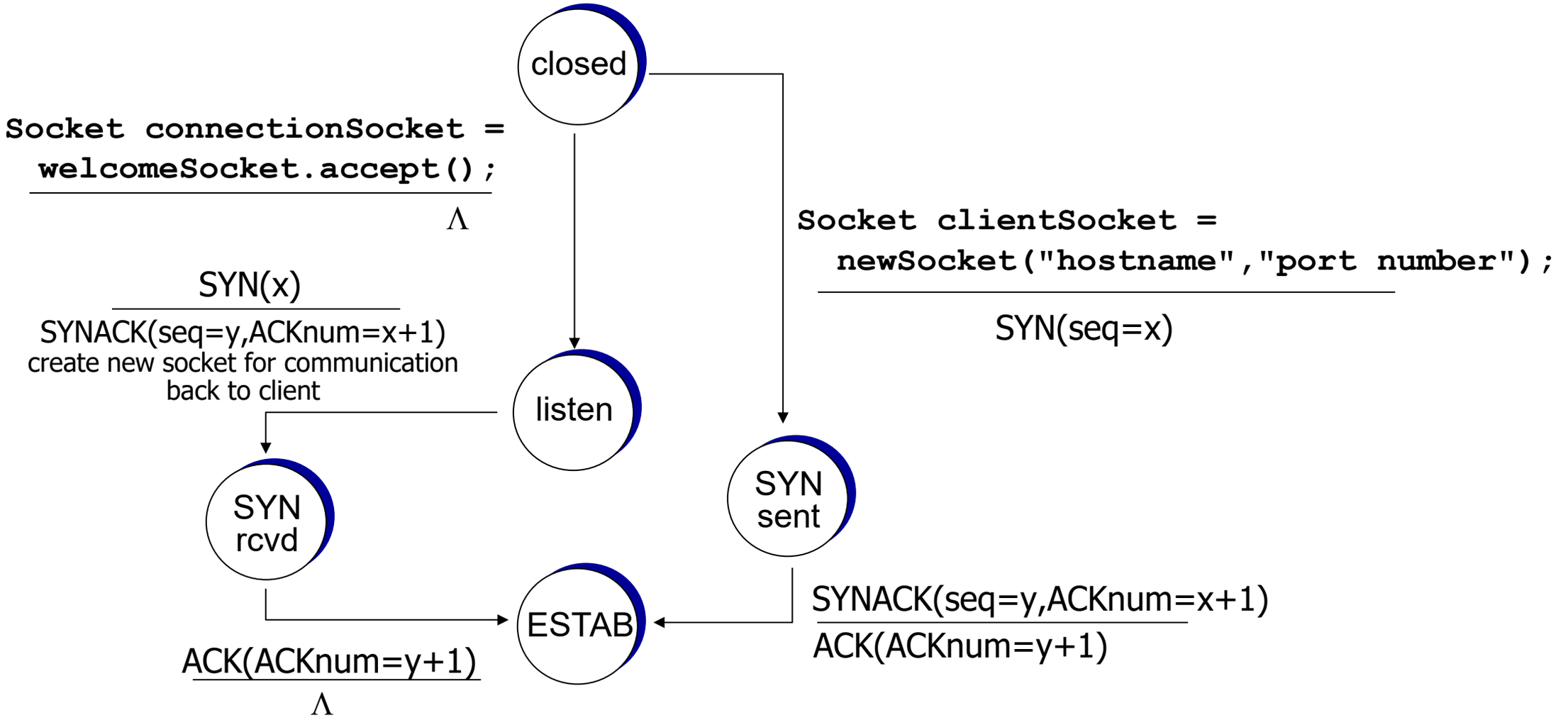
- out-of-order packet:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

Troy University Resources

TCP sender (simplified)

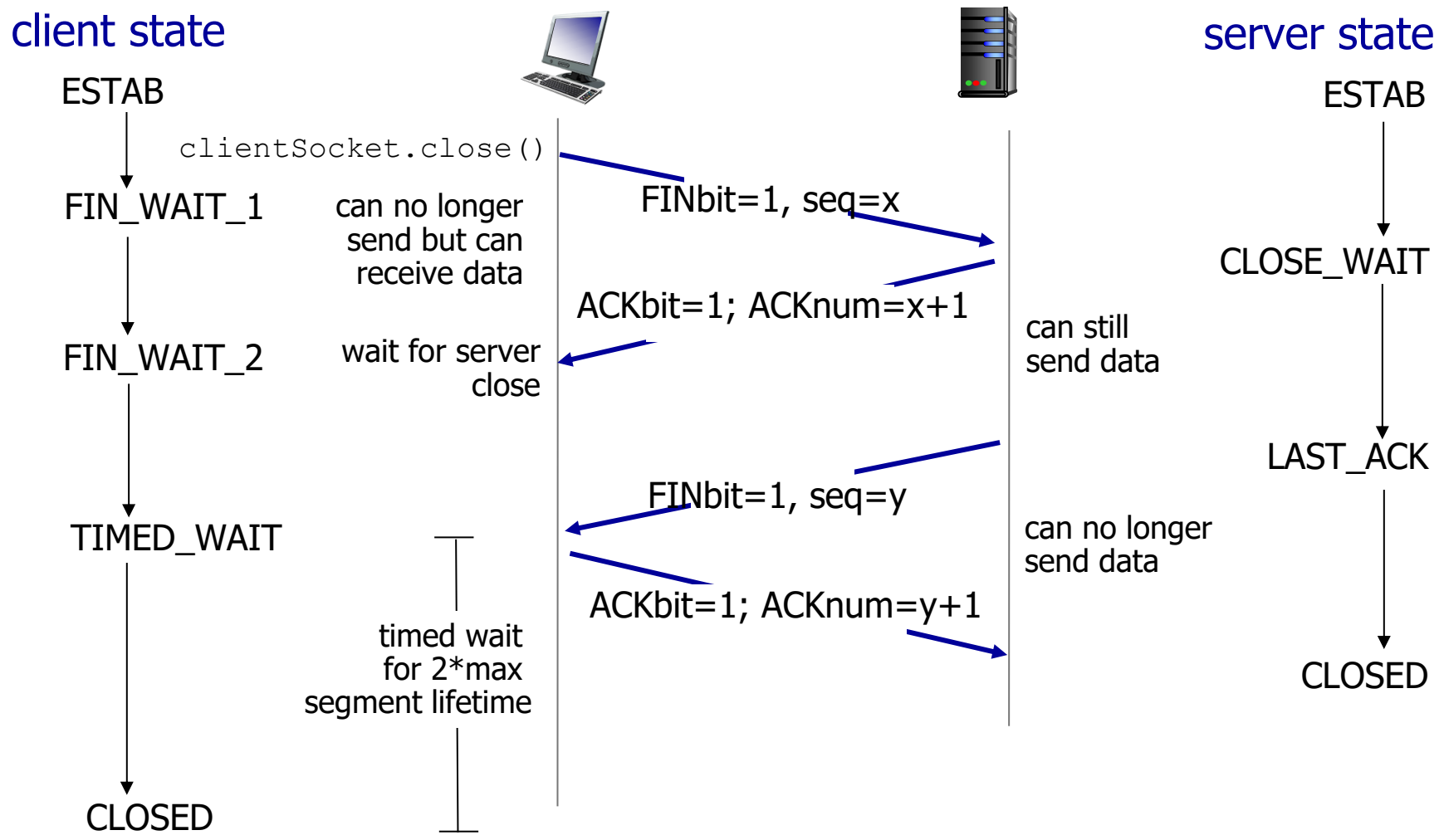


TCP 3-way handshake FSM



Troy University Resources

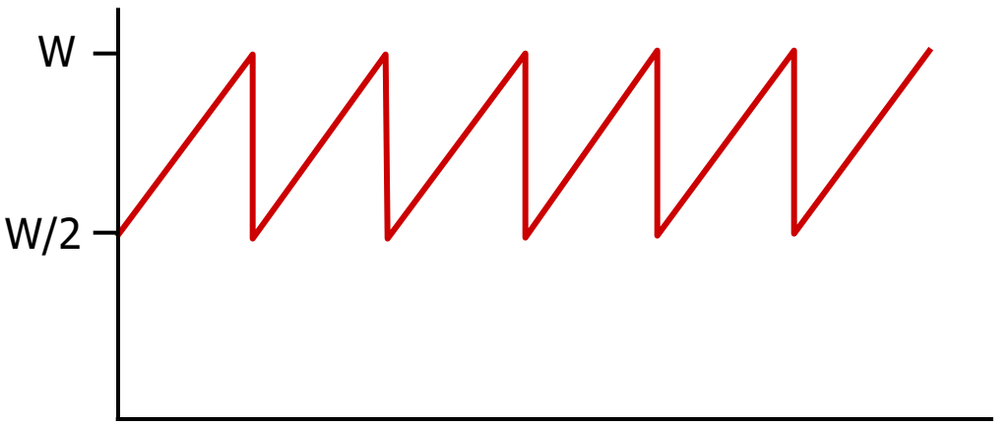
Closing a TCP connection



TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume there is always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\,\frac{W}{RTT}\ \text{bytes/sec}$$



TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ – *a very small loss rate!*

- versions of TCP for long, high-speed scenarios