Lecture 7 — Sockets

Prepared by Jeff Zarnett, taught by Seyed Majid Zahedi
jzarnett@uwaterloo.ca, smzahedi@uwaterloo.ca

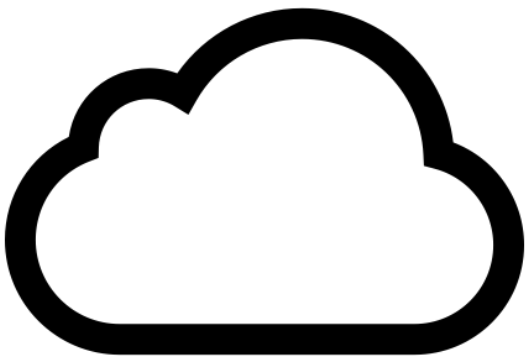
Department of Electrical and Computer Engineering
University of Waterloo

Former US Senator Ted Stevens

Network Communication

If two processes aren't on the same machine, we need to use the network.

The network is frequently portrayed as a mysterious cloud blob:



The Apple iCloud icon

The Socket

The socket API describes how to communicate over the network.

The socket is the concept for how to establish a communication channel.

There are two ways we can communicate: datagrams and connection streams.

Call or Text?

The connection stream is like a telephone call.

Both parties have to be on the line to communicate.

Datagram is like texting or sending a letter in the mail.

Datagrams are unidirectional and can get lost!

Sockets are Files

Much like everything else in UNIX, a socket is handled like a file.

To create a socket, we need the `sys/socket.h` header

```
int socket( int domain, int type, int protocol )
```

Domain: address format; `AF_INET` (IPv4)

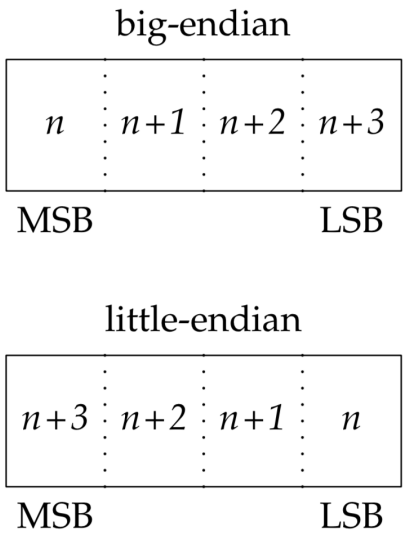
Type: what kind of data; `SOCK_DGRAM` or `SOCK_STREAM`

Protocol: how data is transported; `0` for default (TCP/IP).

Check the Boot of the Car for your Jumper

Speaking the same dialect is important.

Consider a 4-byte integer. Two possible organizations:



Network protocol specifies the use of big-endian!

Translate!

Included in the `arpa/inet.h` header are some functions to help us out.

Their use is advisable even if you're sure the system you are using is big-endian, because of portability of your code.

```
uint32_t htonl( uint32_t hostint32 ) /* Translate 4 byte int to network format */
uint16_t htons( uint16_t hostint16 ) /* Translate 2 byte int to network format */
uint32_t ntohl( uint32_t netint32 )  /* Translate 4 byte int to host format */
uint16_t ntohs( uint16_t netint16 ) /* Translate 2 byte int to host format */
```

An Internet Address

When we want to call someone, we have to put in their phone number.

If we want someone to call us, we need a phone number and we need to be ready to receive calls.

An internet address is represented by the following structure:

```
struct sockaddr_in {
  sa_family_t sin_family; /* Address family */
  in_port_t sin_port; /* Port number */
  struct in_addr sin_addr; /* IPv4 Address */
};
```

Initializing an Address

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons( 2520 );
addr.sin_addr.s_addr = htonl( INADDR_ANY );
```

AF_INET is IPv4.

You are probably quite familiar with how they look: 192.168.0.1

But if you want to go to uwaterloo.ca a translation to an IP address takes place.

Here we chose a constant value, INADDR_ANY
    Choose an address of the current computer.

Ports: Like Apartments!

Imagine your computer is then an apartment building; the port number is which apartment the connection is made with.

Different services (processes) are communicating over different ports.

No two processes can be using the same port at the same time.

By convention, ports with numbers below 1024 considered to be reserved for system services.

Example: `ssh` on port 22.

Address Lookup

Address Lookup

We probably only rarely use IP addresses directly; we use human-friendly names.

For example, you use `ssh username@ecelinux.uwaterloo.ca` and don't need to manually look up the IP address for the server.

Looking up hostnames and the like is somewhat complex (and not the focus), so we will just learn one method for doing this.

Many examples and older texts use the function `gethostbyname()`, but this is now deprecated.

Get Address Info!

The function is prototyped in in `netdb.h`:

```
int getaddrinfo(const char *node,     // e.g. "www.example.com" or IP
                const char *service,  // e.g. "http" or port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

`node`: hostname or IP address.

`service`: protocol or port number.

`hints`: used to restrict the kind of connection you want.

`res`: pointer to be updated with the result.

Look Up an Address Example

```
struct addrinfo hints;
struct addrinfo *serverinfo;  // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_INET;     // Choose IPv4
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

int result = getaddrinfo("www.example.com", "2520", &hints, &serverinfo);
if (result != 0) {
  return -1;
}
struct sockaddr_in * sain = (struct sockaddr_in*) serverinfo->ai_addr;
/* Do things with this */

freeaddrinfo( serverinfo );
```

Assuming that all went well, the `serverinfo` pointer is now pointing to a linked list of `struct sockaddr`.

Most of the time we just need the first result.

Uh, where am I again?

If we are interested in getting the structure for the local computer, we can manually initialize the `struct sockaddr_in` as we did earlier.

Or we can call `getaddrinfo()` with `NULL` as the `node` parameter.

It's possible to use `NULL` for the hints if you are willing to accept the defaults

To deallocate the information that has been allocated, use `freeaddrinfo()`.

If you find a fork in the road... keep it

Up until now what we've learned applies to both the client and server side.

Now the paths diverge.

If we are the client, we'd like to connect to a server.

This is the easier workflow. We just call `connect()`.

Connect

```
int connect( int sockfd, struct sockaddr *addr, socklen_t len);
```

`sockfd`: the socket file descriptor (the `int` we got back from the call to `socket`).

`addr`: address structure from our lookup.

`len`: size of the address structure.
Use either `sizeof` or `ai_addrlen`.

Connect Example

```
struct addrinfo hints;
struct addrinfo *res;
int sockfd;

memset(&hints, 0, sizeof( hints ));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.uwaterloo.ca", "80", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

int status = connect(sockfd, res->ai_addr, res->ai_addrlen);
```

The return value (`status`) tells us if we were successful.

0 indicates success.

Check codes by looking at the man pages:
`http://man7.org/linux/man-pages/man2/connect.2.html`

Success means we're ready to communicate!

Server: Bind, Listen, Accept

The overview of what steps the server is going to do is bind, listen, and accept.

The bind step is how we choose what port we are going to connect to.

The listen step is the part where we wait for connections from a client.

Then the last step is accept: establish the connection so we can start talking.

Bind, Listen, Accept

Bind

`bind()`: associate the socket with whatever port we want to use.

When the `ssh` daemon is available for connection, it's because it has bound itself to the port 22 using `bind`.

```
int socketfd = socket( AF_INET, SOCK_STREAM, 0 );
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons( 2520 );
addr.sin_addr.s_addr = htonl( INADDR_ANY );

bind( socketfd, (struct sockaddr*) &addr, sizeof( addr ));
```

With that done, we've acquired the resource of port 2520 for our use.

Client Side Port?

You'll notice also `bind()` did not happen on the client side.

This is because we don't care on the client side what the outgoing port number is.

So we can just skip that step, unless we have a reason to care.

Listen

`listen()`: int this step we wait for incoming connections.

This is the simplest step and you just call:

```
int listen(int sockfd, int backlog);
```

We listen on a socket that has been bound with `bind` and we'll allow a backlog up to `backlog` connections.

If the queue is full the server system will reject additional requests.

Acceptance is the Last Stage

So we've chosen a socket (got a phone number).

We've said we're ready to listen (our phone is turned on).

The next step is to `accept()` incoming `connect` requests (press the green icon).

```
int accept( int sockfd, struct sockaddr *addr, socklen_t *len );
```

The first parameter is, of course, the socket that we are listening to.

The second and third parameters are the information about the client. We allocate these, pass them in, and they are updated by the call to `accept`.

If we don't care at all about who the client is you can give in NULL.

Accept: “Always Two, There Are”

The return value is a new file descriptor which describes a new socket.

Further communication takes places over that socket (and not the original one).

The original socket is still used for accepting connections, and the new one is the socket used for communication with the client.

If `accept` is called and no requests are in the queue, the server is blocked until a request arrives. We simply wait for the connection.

Put the Pieces Together

```
struct sockaddr_in client_addr;
int client_addr_size = sizeof( struct sockaddr_in );
int newsockfd;

int socketfd = socket( AF_INET, SOCK_STREAM, 0 );
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons( 2520 );
server_addr.sin_addr.s_addr = htonl( INADDR_ANY );

bind( socketfd, (struct sockaddr*) &server_addr, sizeof( server_addr ));
listen( socketfd, 5 );
newsockfd = accept( socketfd, (struct sockaddr*) &client_addr, &client_addr_size );

/* Do something useful */

close( newsockfd );

/* Later when all is done */
close( socketfd );
```

Acceptance is Ongoing

Unless communication is a one-time thing, we call `accept` in some sort of loop.

We then are constantly accepting new connections and doing something useful with each, before going on to the next.

Blocked Number

We could save ourselves some trouble by not caring about the client address:

```
int newsockfd;

int socketfd = socket( AF_INET, SOCK_STREAM, 0 );
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons( 2520 );
server_addr.sin_addr.s_addr = htonl( INADDR_ANY );

bind( socketfd, (struct sockaddr*) &server_addr, sizeof( server_addr ));
listen( socketfd, 5 );
newsockfd = accept( socketfd, NULL, NULL );
/* Do something useful */

close( newsockfd );

/* Later when all is done */
close( socketfd );
```

Connection Established

We are finally ready for the client and server to communicate.

Connection Established

The client communicates using its original socket file descriptor.

The server communicates using the new file descriptor.

Likely you will move some of the boilerplate into your own function, e.g.:

```
int connect_to( const char* host, const char* port );
```

Next: let's actually communicate!