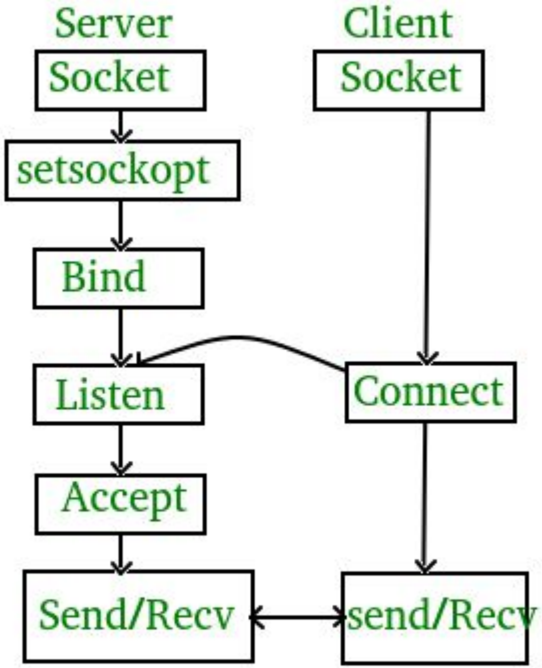
MONTANA
STATE UNIVERSITY

Socket Programming

Talking With The Network

Sockets

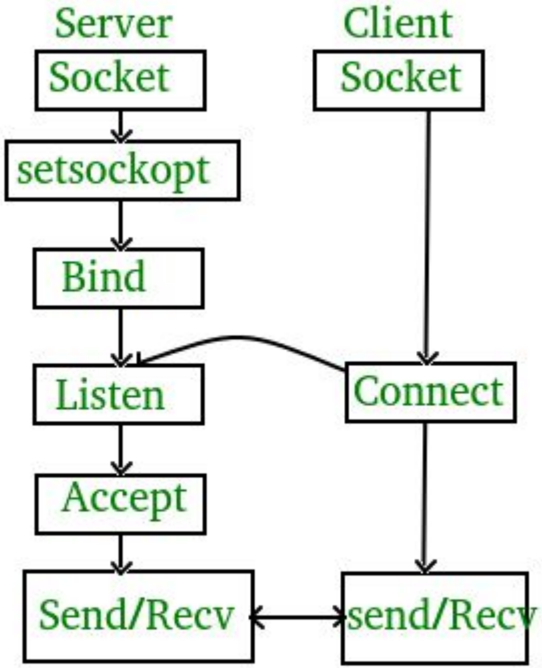
- Sockets provide an abstraction for communicating across networks
- Typically there are two use cases for sockets:
  - A client
  - A server
- C treats sockets as *File Descriptors* and the API is heavily influenced by File I/O



Sockets

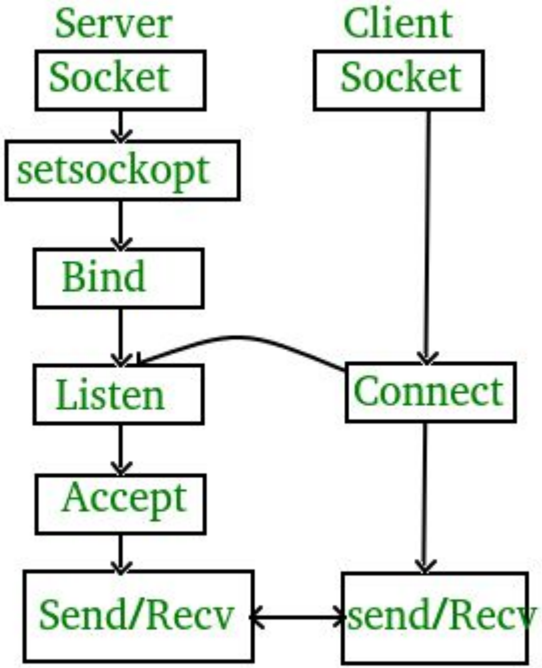
- This is in line with the UNIX philosophy:

  “Everything is a file”



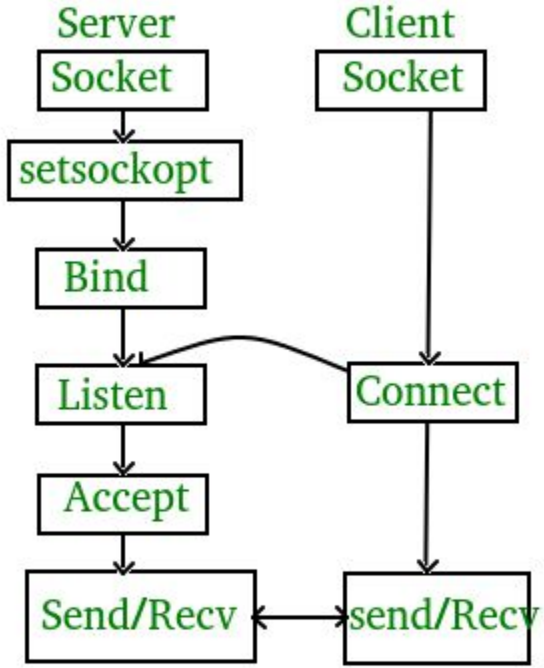
Sockets

- I have to warn you, the C socket API is *unfriendly*
- If you have done network programming in Java or Python or, especially, Go, it’s going to feel like banging rocks together
    - That’s C for you...



Client Sockets

- A sockets client is easier, so we will start with that
- Three operations:
  - Create the socket
  - Connect
  - Send/Receive



Client Sockets - Create

- To create a socket we first need to import
  - the sys/socket.h lib - provides the core socket functions
  - The netinet/in.h lib - provides some constants that make our code clearer

```c
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

void client_demo(){
    int socket_fd = socket(AF_INET,
            type: SOCK_STREAM,  protocol: IPPROTO_TCP);
    if (socket_fd == -1)
    {
        printf( format: "Could not create socket");
    }
```

Client Sockets - Create

- The next step is to create a socket file descriptor with the *socket* function call
- First argument is the domain
    - In this case we are using IPv4, known colloquially as “The Internet”
- Second argument is the mode
    - We are going to use a streaming, the most common mode

```c
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

void client_demo(){
    int socket_fd = socket(AF_INET,
            type: SOCK_STREAM,  protocol: IPPROTO_TCP);
    if (socket_fd == -1)
    {
        printf( format: "Could not create socket");
    }
```

Client Sockets - Create

- The third argument is the protocol
  - We are going to use TCP over IP, sometimes referred to TCP/IP
  - Another common protocol is UDP

```c
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

void client_demo(){
    int socket_fd = socket(AF_INET,
            type: SOCK_STREAM,  protocol: IPPROTO_TCP);
    if (socket_fd == -1)
    {
        printf( format: "Could not create socket");
    }
```

Client Sockets - Connect

- The next step is to connect to a server
- To do this we will need to create a data structure to specify the server
- *sockadder_in* struct - represents a socket address we want to connect to

```c
#include <arpa/inet.h>

struct sockaddr_in connecting_to;

// google.com
connecting_to.sin_addr.s_addr = inet_addr( cp: "74.125.235.20");
connecting_to.sin_family = AF_INET;
// NB no need to use htons, we are on linux only
connecting_to.sin_port = 80;
```

Client Sockets - Connect

- The first assignment uses the *inet_addr* function to fill out the address of the server from a string
- Note that we need to include the *arpa/inet.h* library to get the *inet_addr*() function

```
#include <arpa/inet.h>
struct sockaddr_in connecting_to;

// google.com
connecting_to.sin_addr.s_addr = inet_addr( cp: "74.125.235.20");
connecting_to.sin_family = AF_INET;
// we have to deal with endian-ness
connecting_to.sin_port = htons( hostshort: 80);
```

Client Sockets - Connect

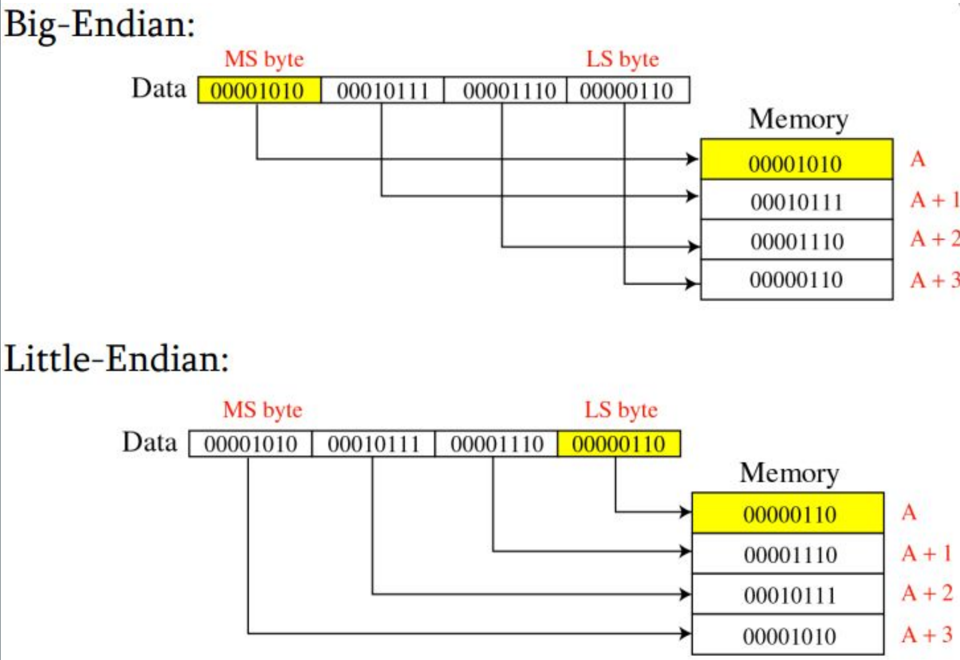
- The second line says that this is an internet address
- The third specifies the port but we have to call a funny htons function to reorder the bytes from little-endian (x86 standard) to big-endian (network standard)

```c
#include <arpa/inet.h>
struct sockaddr_in connecting_to;

// google.com
connecting_to.sin_addr.s_addr = inet_addr( cp: "74.125.235.20");
connecting_to.sin_family = AF_INET;
// we have to deal with endian-ness
connecting_to.sin_port = htons( hostshort: 80);
```

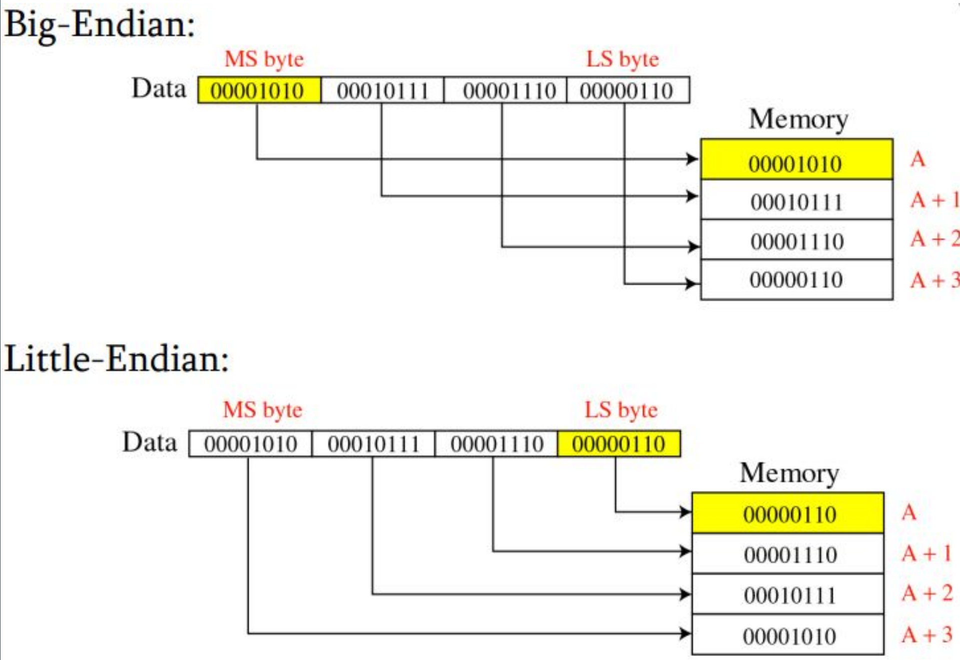
A Note on Byte Order

- It turns out that there are multiple ways to order bytes on a machine
- Consider a 32 bit or 4 byte value
- There are two common ways to order the bytes physically



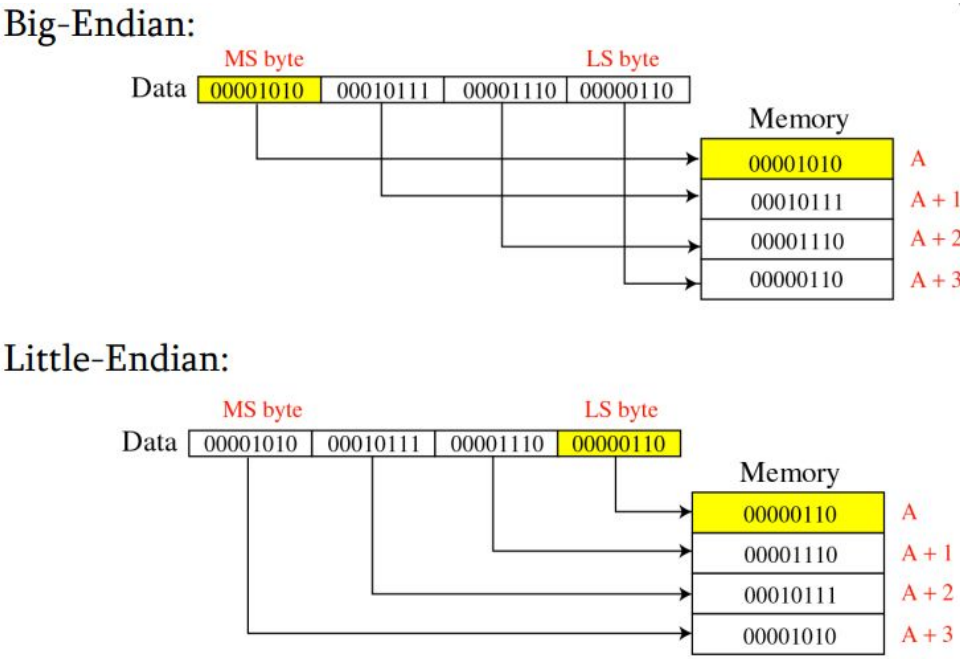
A Note on Byte Order

- The most logical (to me) is "Big Endian"
    - You put the most significant bytes on the right
    - Again, this is the byte layout *in memory*



A Note on Byte Order

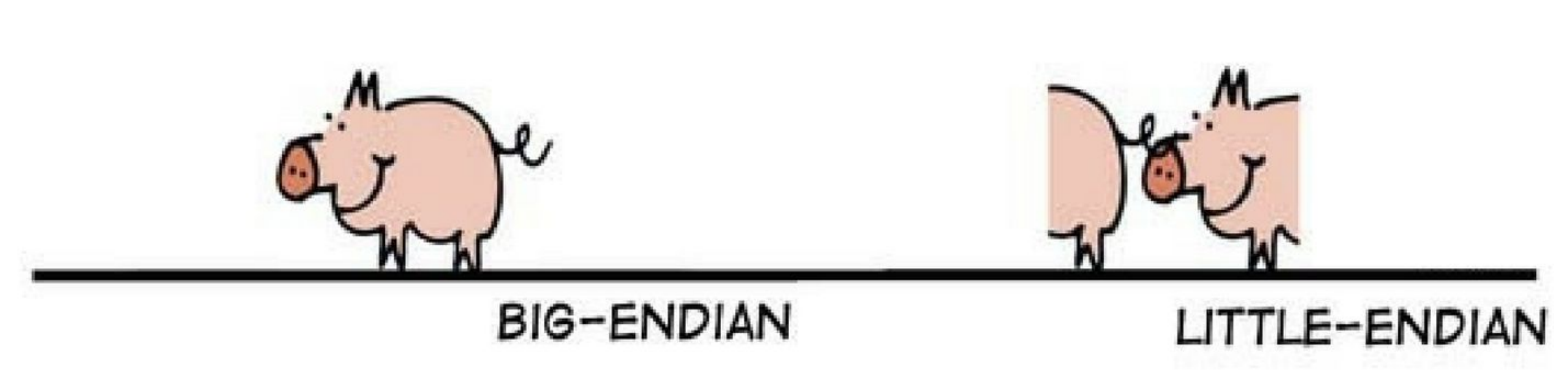
- Another option, maybe a bit weirder, is to place the *least significant* byte first in memory
- This is called *Little Endian*



Client Sockets - Connect

- Now we are ready to connect with the *connect* function!
- We pass in our file descriptor, the server we want to connect to and the size of server spec in bytes
  - Yeah, this is really crazy
  - That's just the way it is kids

```
//Connect to remote server
if (connect(socket_fd ,
        // we cast here because sockaddr_in
        // is structurally compatible
        // with sockaddr so it works out.
        // Insane, but this is the
        // recommended approach o_O
        (const struct sockaddr *) &connecting_to,
        sizeof(connecting_to)) < 0)
{
    puts( s: "Could not connect!\n");
} else {
    puts( s: "Connected!\n");
}
```


BIG-ENDIAN
LITTLE-ENDIAN

Client Sockets - Send/Recv

- Finally, we are ready to send some data and receive a response
- We send with the *send* function, which takes
    - The socket file descriptor
    - A byte buffer
    - The byte buffers length
    - Optional flags

```
//Send some data
char * message = "GET / HTTP/1.1\r\n\r\n";
if( send(socket_fd , message ,
        strlen(message) , flags: 0) < 0)
{
    puts( s: "Send failed");
} else {
    puts( s: "Data Sent!\n");
```

Client Sockets - Send/Recv

- And, if we got all that correct, we can now receive some data from our friends at google with the *recv()* function
- Note that we pass a fixed-size buffer into *recv*

```
//Receive some data
char buffer[2000];
if( recv(socket_fd, buffer , n: 2000 , flags: 0) < 0)
{
    puts( s: "Recieve failed");
} else {
    puts( s: "Reply received\n");
    puts(buffer);
}
```

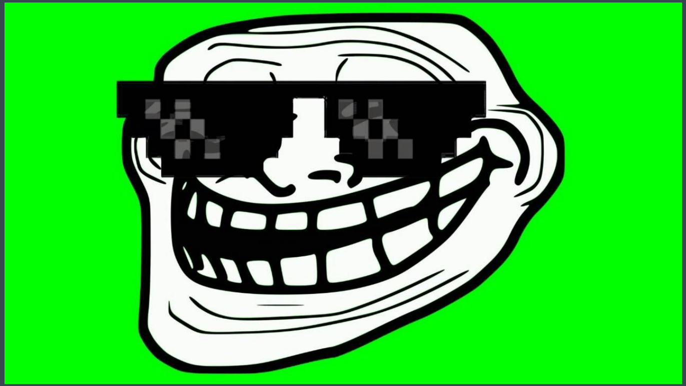
Client Sockets - Done!

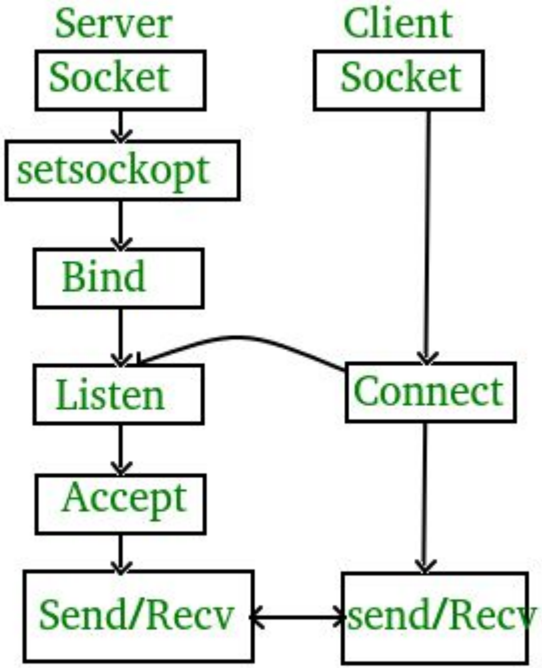
- Pretty crazy, eh?
- Yes, the C socket API is difficult
  - Just accept it and learn to live with it
- I’ll be honest: there is going to be a lot of copy and paste going on in the projects

```c
//Receive some data
char buffer[2000];
if( recv(socket_fd, buffer , n: 2000 , flags: 0) < 0)
{
    puts( s: "Recieve failed");
} else {
    puts( s: "Reply received\n");
    puts(buffer);
}
```

Server Sockets

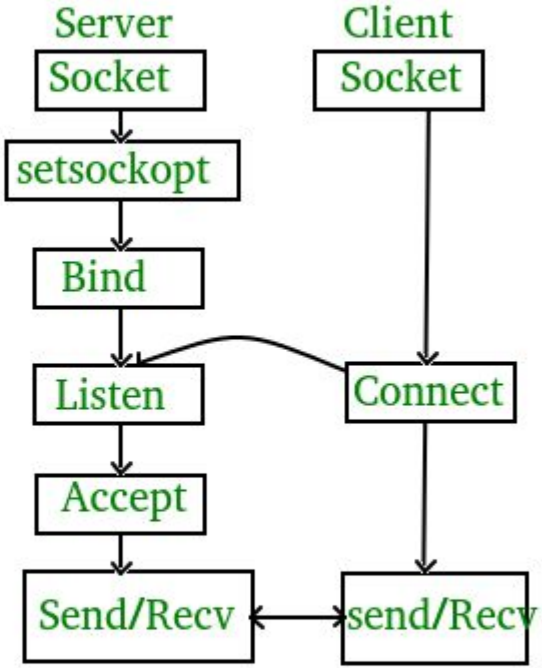
- So, you thought client sockets were bad, eh?
- Wait until you see server sockets





Server Sockets

- To create a server we are going to need to
  - Create a socket
  - Set any socket options we need
  - Configure our server struct
  - Bind it
  - Listen
  - Accept connections
  - Send/Receive
- Good times!



Server Sockets

- Good news! Creating the socket is identical to the client code
- Create a socket file descriptor with the appropriate protocols specified

```
int socket_fd = socket(AF_INET,
          type: SOCK_STREAM,
          protocol: IPPROTO_TCP);
if (socket_fd == -1)
{
    printf( format: "Could not create socket\n");
}
```

Server Sockets

- Now, let's set an option that will allow us to relaunch the program and still bind to the socket immediately
- If we don't do this, we will get bind failures (see 2 steps ahead) for a while
- All this to just tell C to allow us to rebind
  - Isn't C great?

```
int yes = 1;
setsockopt(server_socket_fd,
        SOL_SOCKET,
        SO_REUSEADDR, &yes, sizeof(yes));
```

Server Sockets

- Again, we create a *sockaddr_in* struct but we fill it in with server info rather than client info

```
struct sockaddr_in server;

// fill out the socket information
server.sin_family = AF_INET;
// bind the socket on all available interfaces
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons( hostshort: 9876 );
```

Server Sockets

- Next, we *bind* the socket to with the *bind()* function call
- This claims the port on the local interfaces, but does not start listening

```
struct sockaddr_in server;

// fill out the socket information
server.sin_family = AF_INET;
// bind the socket on all available interfaces
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons( hostshort: 9876 );
```

Server Sockets

- To listen, we unsurprisingly need to call the *listen()* function with our file descriptor
- The second argument is how many connections to queue before starting to refuse new connections

```
puts( s: "Bind worked!");
listen(socket_fd , n: 3);
```

Server Sockets

- Finally, we need to accept connections
- This will create a **new** socket file descriptor that we can use to communicate with a client
- Note that we pass in two in-out parameters
  - We won't be using them, we just want the socket file descriptor

```c
//Accept an incoming connection
puts( s: "Waiting for incoming connections...");
struct sockaddr_in client;
socklen_t size_from_connect;
int client_socket_fd =
        accept(server_socket_fd,
               (struct sockaddr *) &client,
               &size_from_connect);
```

Server Sockets

- With all that done, we can now write a message to the client that has connected with us

```
if (client_socket_fd < 0) {
    puts( s: "Bad client connect");
} else {
    char *message =
            "Thank you for coming, come again.\n\n";
    send(client_socket_fd, message,
            strlen(message), flags: 0);
}
```

Server Sockets

- Typically a server would accept in a loop
- You would also typically not do I/O in the accept loop
- You would fork another process or thread to do the communication
  - No other client can connect while the I/O is happening!

```
while((client_socket_fd = accept(server_socket_fd,
                   (struct sockaddr *) &client,
                   &size_from_connect)) > 0) {
    char message[100] = {0};
    sprintf(message,
            format: "Thank you for coming, come again - req %d\n\n",
            request_count++);
    send(client_socket_fd, message,
        strlen(message),  flags: 0);
    close(client_socket_fd);
}
```

Other APIs

- Not only is the regular C API pretty difficult to work with, but there are also perf issues with it
- Two APIs that address these issues are *poll()* and *select()*
  - We won't be covering these APIs, but if you end up getting into high performance socket work, you should be aware of them
- If you *really* want to learn network programming, I recommend Beej's Guide to Network Programming
  - https://beej.us/guide/bgnet/

Review

- C Sockets… yikes.
  - But useful: most of the software that powers the internet is written in C
- Clients sockets aren't too bad (if you don't do threading)
- Servers are a little worse
  - You will be implementing a server for the project, so rewatch that part of the lecture a few times
- It's Just Code™

MONTANA
STATE UNIVERSITY