Week 13 - Monday

CS222

Last time

- What did we talk about last time?
- Low level file I/O practice
- File systems

Questions?

Project 6

Quotes

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

Edsger Dijkstra

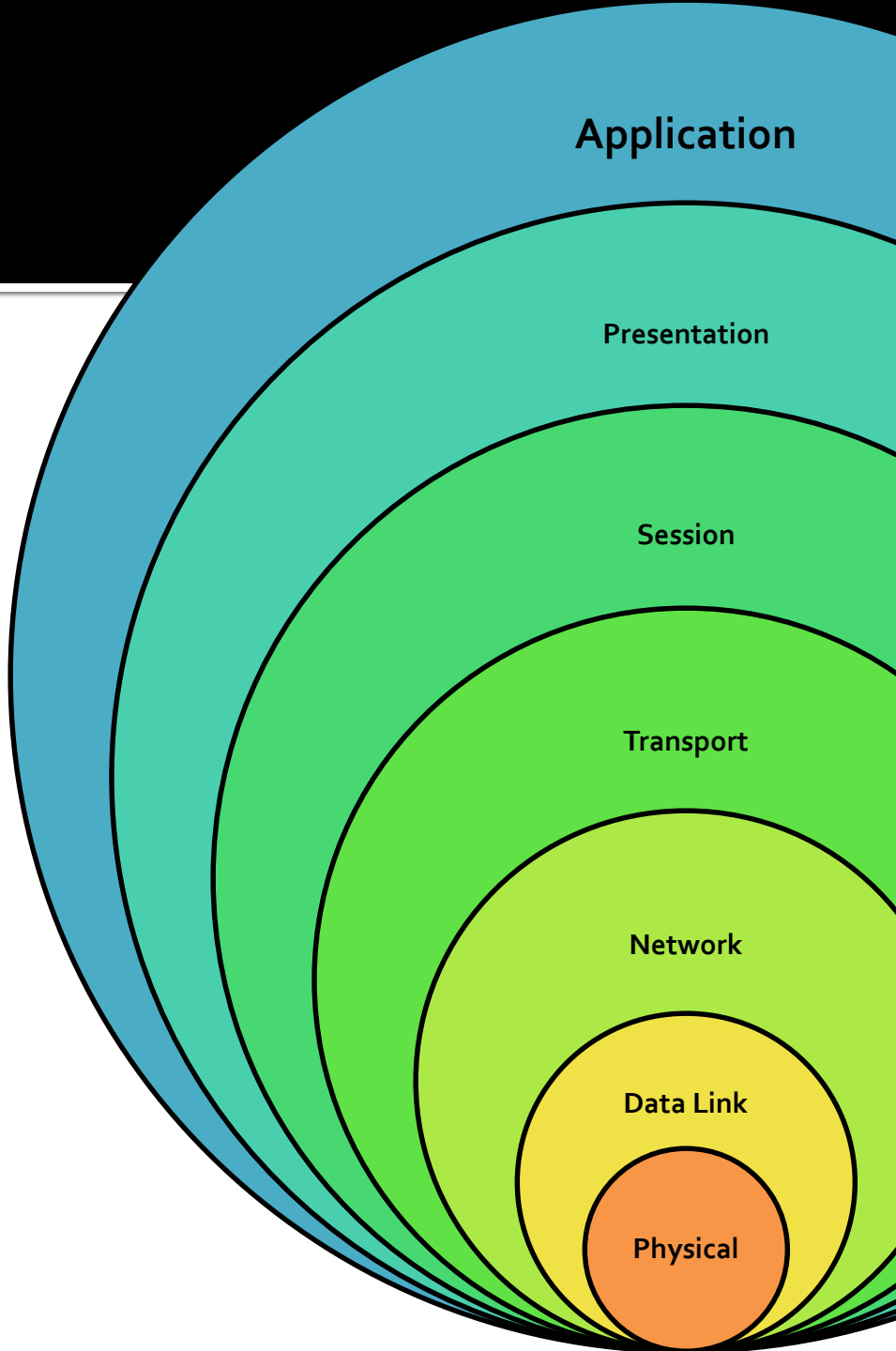
Networking

OSI seven layer model

- You can build layers of I/O on top of other layers
  - **`printf()`** is built on top of low level **`write()`** call
- The standard networking model is called the Open Systems Interconnection Reference Model
  - Also called the OSI model
  - Or the 7 layer model

Protocols

- There are many different communication protocols
- The OSI reference model is an idealized model of how different parts of communication can be abstracted into 7 layers
- Imagine that each layer is talking to another parallel layer called a **peer** on another computer
- Only the physical layer is a real connection between the two



Layers

- Not every layer is always used
- Sometimes user errors are referred to as Layer 8 problems

Layer	Name	Mnemonic	Activity	Example
7	**Application**	Away	User-level data	HTTP
6	**Presentation**	Pretzels	Data appearance, some encryption	SSL
5	**Session**	Salty	Sessions, sequencing, recovery	IPC and part of TCP
4	**Transport**	Throw	Flow control, end-to-end error detection	TCP
3	**Network**	Not	Routing, blocking into packets	IP
2	**Data Link**	Dare	Data delivery, packets into frames, transmission error recovery	Ethernet
1	**Physical**	Programmers	Physical communication, bit transmission	Electrons in copper

Physical layer

- There is where the rubber meets the road
- The actual protocols for exchanging bits as electronic signals happen at the physical layer
- At this level are things like RJ45 jacks and rules for interpreting voltages sent over copper
  - Or light pulses over fiber

Data link layer

- Ethernet is the most widely used example of the data layer
- Machines at this layer are identified by a 48-bit Media Access Control (MAC) address
- The Address Resolution Protocol (ARP) can be used for one machine to ask another for its MAC address
  - Try the **`arptables`** command in Linux
- Some routers allow a MAC address to be spoofed, but MAC addresses are intended to be unique and unchanging for a particular piece of hardware

Network layer

- The most common network layer protocol is Internet Protocol (IP)
- Each computer connected to the Internet should have a unique IP address
  - IPv4 is 32 bits written as four numbers from 0 – 255, separated by dots
  - IPv6 is 128 bits written as 8 groups of 4 hexadecimal digits
- We can use **`traceroute`** to see the path of hosts leading to some IP address

Transport layer

- There are two popular possibilities for the transport layer
- Transmission Control Protocol (TCP) provides reliability
  - Sequence numbers for out of order packets
  - Retransmission for packets that never arrive
- User Datagram Protocol (UDP) is simpler
  - Packets can arrive out of order or never show up
  - Many online games use UDP because speed is more important

Session layer

- This layer doesn't really exist in the TCP/IP model
- You can consider the setting up and tearing down a TCP connections as the session layer

Presentation layer

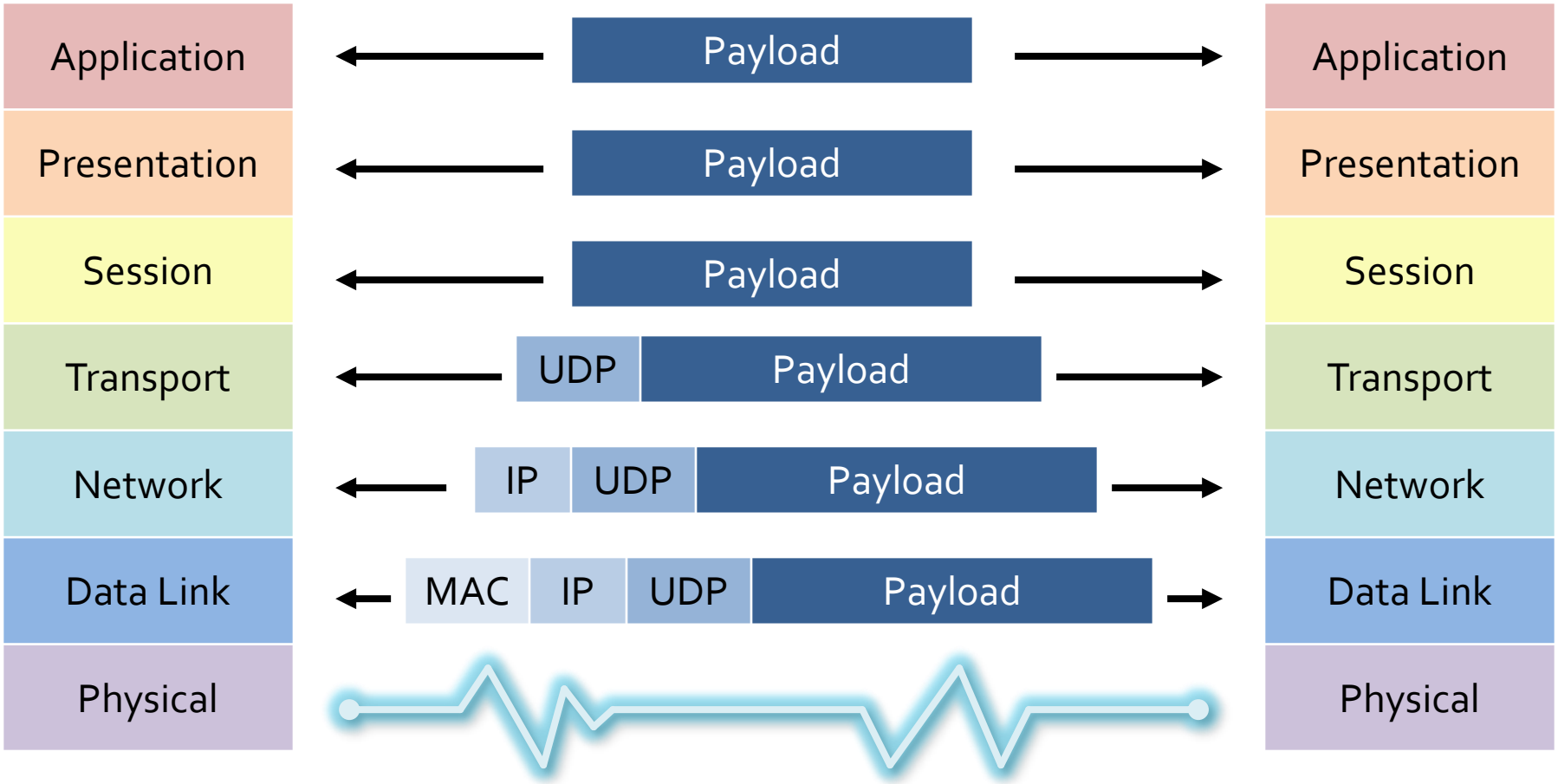
- The presentation layer is often optional
- It specifies how the data should appear
- This layer is responsible for character encoding (ASCII, UTF-8, etc.)
- MIME types are sometimes considered presentation layer issues
- Encryption and decryption can happen here

Application layer

- This is where the data is interpreted and used
- HTTP is an example of an application layer protocol
- A web browser takes the information delivered via HTTP and renders it
- Code you write deals a great deal with the application layer

Transparency

- The goal of the OSI model is to make lower layers transparent to upper ones



Mnemonics

- Seven layers is a lot to remember
- Mnemonics have been developed to help

Application	All	All	A	Away
Presentation	Pros	People	Powered-Down	Pretzels
Session	Search	Seem	System	Salty
Transport	Top	To	Transmits	Throw
Network	Notch	Need	No	Not
Data Link	Donut	Data	Data	Dare
Physical	Places	Processing	Packets	Programmers

TCP/IP

- The OSI model is sort of a sham
  - It was invented after the Internet was already in use
  - You don't need all layers
  - Some people think this categorization is not useful
- Most network communication uses TCP/IP
- We can view TCP/IP as four layers:

Layer	Action	Responsibilities	Protocol
Application	Prepare messages	User interaction	HTTP, FTP, etc.
Transport	Convert messages to packets	Sequencing, reliability, error correction	TCP or UDP
Internet	Convert packets to datagrams	Flow control, routing	IP
Physical	Transmit datagrams as bits	Data communication	

TCP/IP

- A TCP/IP connection between two hosts (computers) is defined by four things
  - Source IP
  - Source port
  - Destination IP
  - Destination port
- One machine can be connected to many other machines, but the port numbers keep it straight

Common port numbers

- Certain kinds of network communication are usually done on specific ports
  - **20** and **21**: File Transfer Protocol (FTP)
  - **22**: Secure Shell (SSH)
  - **23**: Telnet
  - **25**: Simple Mail Transfer Protocol (SMTP)
  - **53**: Domain Name System (DNS) service
  - **80**: Hypertext Transfer Protocol (HTTP)
  - **110**: Post Office Protocol (POP3)
  - **443**: HTTP Secure (HTTPS)

IP addresses

- Computers on the Internet have addresses, not names
- **Google.com** is actually [**`74.125.67.100`**]
- **Google.com** is called a **domain**
- The Domain Name System or DNS turns the name into an address

IPv4

- Old-style IP addresses are in this form:
  - `74.125.67.100`
- 4 numbers between 0 and 255, separated by dots
- That's a total of 256^4 = 4,294,967,296 addresses
- But there are 7 billion people on earth…

IPv6

- IPv6 are the new IP addresses that are beginning to be used by modern hardware
  - 8 groups of 4 hexadecimal digits each
  - **`2001:0db8:85a3:0000:0000:8a2e:0370:7334`**
  - 1 hexadecimal digit has 16 possibilities
  - How many different addresses is this?
  - $16^{32} = 2^{128} \approx 3.4 \times 10^{38}$ is enough to have 500 trillion addresses for every cell of every person's body on Earth
  - Will it be enough?!

Sockets

Sockets

- **Sockets** are the most basic way to send data over a network in C
- A socket is **one end** of a two-way communication link between two programs
  - Just like you can plug a phone into a socket in your wall (if you are living in 1980)
  - Both programs have to have a socket
  - And those sockets have to be connected to each other
- Sockets can be used to communicate within a computer, but we'll focus on Internet sockets

Includes

- There are a lot of includes you'll need to get your socket programming code working correctly
- You should always add the following:
  - `#include <netinet/in.h>`
  - `#include <netdb.h>`
  - `#include <sys/socket.h>`
  - `#include <sys/types.h>`
  - `#include <arpa/inet.h>`
  - `#include <unistd.h>`

socket()

- If you want to create a socket, you can call the **socket()** function
- The function takes a communication domain
  - Will always be **AF_INET** for IPv4 Internet communication
- It takes a type
  - **SOCK_STREAM** usually means TCP
  - **SOCK_DGRAM** usually means UDP
- It takes a protocol
  - Which will always be **0** for us
- It returns a file descriptor (an **int**)

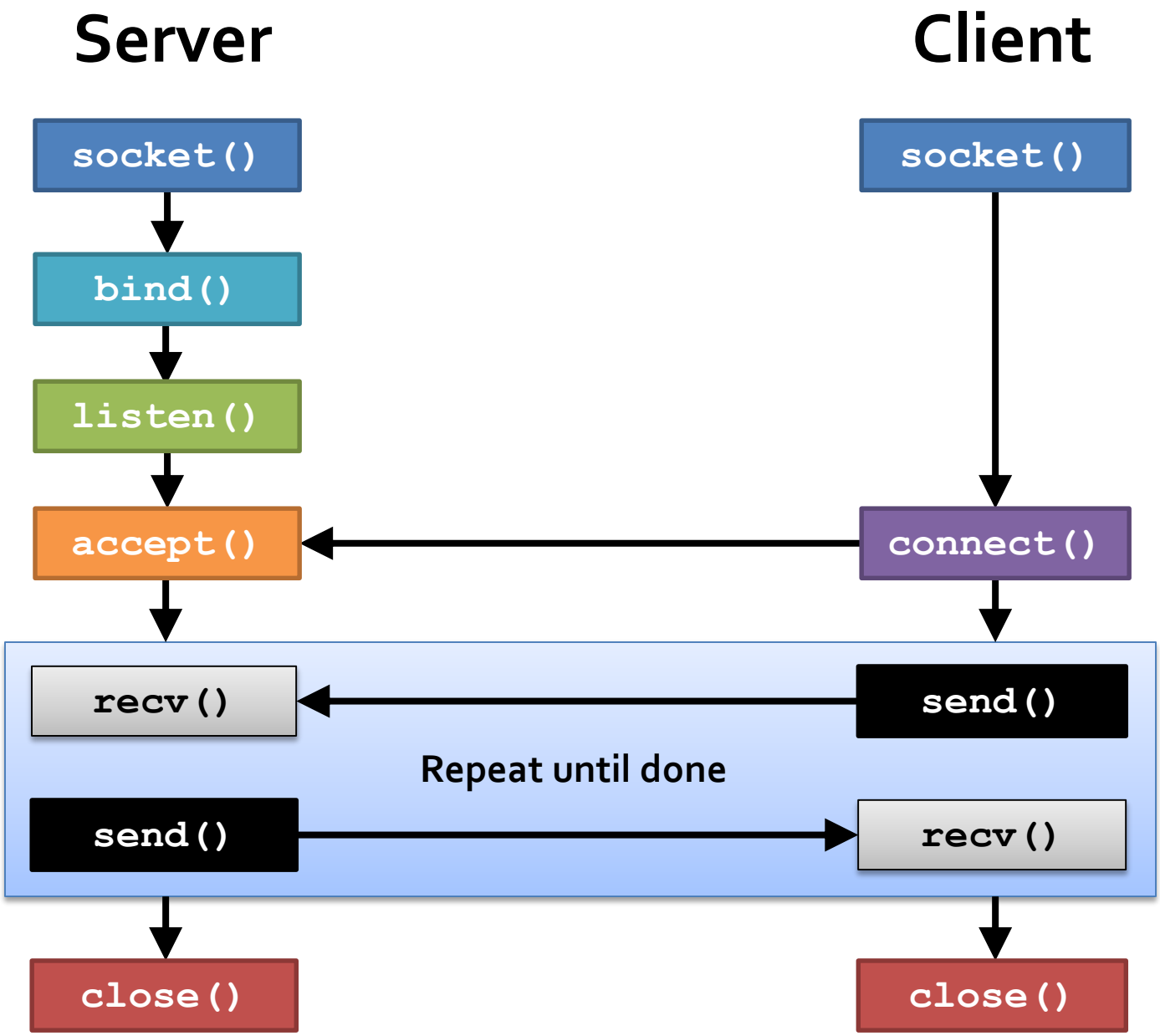
```
int sockFD = -1;
sockFD = socket(AF_INET, SOCK_STREAM, 0);
```

Now you've got a socket...

- What are you going to do with it?
- By themselves, they aren't useful
- You need to connect them together
- We're going to be interested in the following functions to work with sockets
  - `bind()`
  - `listen()`
  - `accept()`
  - `connect()`
- And also functions that are similar to the ones you know from low-level file I/O
  - `recv()`
  - `send()`
  - `close()`

Clients vs. servers

- Using sockets is usually associated with a client-server model
- A **server** is a process that sits around waiting for a connection
  - When it gets one, it can do sends and receives
- A **client** is a process that connects to a waiting server
  - Then it can do sends and receives
- Clients and servers are processes, not computers
  - You can have many client and server processes on a single machine


Server
Client
socket()
socket()
bind()
listen()
accept()
connect()
recv()
send()
Repeat until done
send()
recv()
close()
close()

Client

- We'll start with the client, since the code is simpler
- Assuming that a server is waiting for us to connect to it, we can do so with the **`connect()`** function
- It takes
  - A socket file descriptor
  - A pointer to a **`sockaddr`** structure
  - The size of the **`sockaddr`** structure
- It returns `-1` if it fails

```
connect(sockFD, (struct sockaddr *) &address,
      sizeof(address));
```

Address

- Where do we get a **sockaddr** structure?
- Actually, we use a **sockaddr_in** structure for Internet communication

```
struct sockaddr_in
{
        short sin_family;           //AF_INET or AF_INET6
        unsigned short sin_port;    //e.g htons(3490)
        struct in_addr sin_addr;    //see struct in_addr
        char sin_zero[8];           //zero this if you want
};

struct in_addr
{
        unsigned long s_addr;       //load with inet_pton()
};
```

`htons()` and `ntohs()`

- Is your machine little endian or big endian?
- Is the network you're connecting to little endian or big endian?
- You don't want to have to keep it straight
- But the port number you put in a **`sockaddr_in`** has to have the network endianness
- **`htons()`** takes a **`short`** from **h**ost endianness **to** **n**etwork endianness
- **`ntohs()`** takes a **`short`** from **n**etwork endianness **to** **h**ost endianness
- Both functions work correctly whether or not the network and the host have the same endianness

What about `in_addr`?

- The **`in_addr`** field in **`sockaddr_in`** is hard to get right as a human
- So, we fill it up with a call to **`inet_pton()`**
  - The name means "presentation to network", or human-readable to machine readable
- It takes
  - A communication domain (**`AF_INET`** or **`AF_INET6`**)
  - A string with an IP address
  - A pointer to an **`in_addr`** field

```
inet_pton(AF_INET, "173.194.43.0",
&(address.sin_addr));
```

Putting the address together

- We fill a **`sockaddr_in`** structure with
  - The communication domain
  - The correct endian port
  - The translated IP address
- We fill it with zeroes first, just in case

```
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = htons(80);
inet_pton(AF_INET, "173.194.43.0",
&(address.sin_addr));
```

Sending

- Once you've created your socket, set up your port and address, and called **`connect()`**, you can send data
  - Assuming there were no errors
  - Sending is very similar to writing to a file
- The **`send()`** function takes
  - The socket file descriptor
  - A pointer to the data you want to send
  - The number of bytes you want to send
  - Flags, which can be **0** for us
- It returns the number of bytes sent

```
char* message = "Flip mode is the squad!";
send(socketFD, message, strlen(message)+1, 0);
```

Receiving

- Or, once you're connected, you can also receive data
  - Receiving is very similar to reading from a file
- The **`recv()`** function takes
  - The socket file descriptor
  - A pointer to the data you want to receive
  - The size of your buffer
  - Flags, which can be **0** for us
- It returns the number of bytes received, or **0** if the connection is closed, or **-1** if there was an error

```
char message[100];
recv(socketFD, message, 100, 0);
```

Servers

- Sending and receiving are the same on servers, but setting up the socket is more complex
- Steps:
  1. Create a socket in the same way as a client
  2. Bind the socket to a port
  3. Set up the socket to listen for incoming connections
  4. Accept a connection

Bind

- Binding attaches a socket to a particular port at a particular IP address
  - You can give it a flag that automatically uses your local IP address, but it could be an issue if you have multiple IPs that refer to the same host
- Use the bind() function, which takes
  - A socket file descriptor
  - A sockaddr pointer (which will be a sockaddr_in pointer for us) giving the IP address and port
  - The length of the address

```
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = htons(80);
address.sin_addr.s_addr = INADDR_ANY;
bind(socketFD, (struct sockaddr*)&address,
sizeof(address));
```

`getaddrinfo()`

- If you don't want to fill out the **`sockaddr_in`** structure by hand, you can use the **`getaddrinfo()`** function to do it for you
- Unfortunately, you have to fill out a hints structure which is almost as bad
- But **`getaddrinfo()`** is good because it can do DNS lookup, allowing you to put in a domain name (www.google.com) instead of an IP address

Calling getaddrinfo()

- **getaddrinfo()** takes
  - A string giving an IP address or a domain name (or **NULL** if it's going to figure out your computer's IP for you)
  - A string giving the port number (e.g. **"80"**) or the service linked to a port number (e.g. **"http"**)
  - A pointer to an **addrinfo** structure that gives hints about the address
  - A pointer to a pointer to an **addrinfo** structure, which will end up pointing to a dynamically allocated linked list of addresses that fit the bill

```
struct addrinfo hints, *addresses;
memset(&hints, 0, sizeof(hints));
hints.ai_flag = AI_PASSIVE; //pick IP for me
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo(NULL, "80", &hints, &addresses);
```

Calling getaddrinfo() again

- The previous code showed how to call it to get an unspecified IP address for a local machine
- We could use it to get an IP that we specify as follows

```
struct addrinfo hints, *addresses;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.google.com", "http", &hints,
&addresses);
```

Listening

- After a server has bound a socket to an IP address and a port, it can listen on that port for incoming connections
- To set up listening, call the **`listen()`** function
- It takes
  - A socket file descriptor
  - The size of the queue that can be waiting to connect
- You can have many computers waiting to connect and handle them one at a time
- For our purpose, a queue of size 1 often makes sense

```
listen( socketFD, 1);
```

Accept

- Listening only sets up the socket for listening
- To actually make a connection with a client, the server has to call **`accept()`**
- It is a blocking call, so the server will wait until a client tries to connect
- It takes
  - A socket file descriptor
  - A pointer to a **`sockaddr`** structure that will be filled in with the address of the person connecting to you
  - A pointer to the length of the structure
- It returns a file descriptor for the client socket
- We will usually use a **`sockaddr_storage`** structure

```
struct sockaddr_storage otherAddress;
socklen_t otherSize = sizeof(otherAddress);
int otherSocket;
otherSocket = accept( socketFD, (struct
sockaddr *) &otherAddress, &otherSize);
```

setsockopt()

- The **setsockopt()** function allows us to set a few options on a socket
- The only one we care about is the **SO_REUSEADDR** option
- If a server crashes, it will have to wait for a timeout (a minute or so) to reconnect on the same port unless this option is set
  - A dead socket is taking up the port
- When working on Project 6, it's a good idea to use this so you don't get stuck waiting around

```
int value = 1; //1 to turn on port reuse
setsockopt(socketFD, SOL_SOCKET,
SO_REUSEADDR, &value, sizeof(value));
```

`send()` and `recv()`

- Last time, we suggested that you use **`send()`** and **`recv()`** for writing to and reading from sockets
- You can actually use **`write()`** and **`read()`**
- The difference is that **`send()`** and **`recv()`** have an extra parameter for flags
  - We provided **0** as the argument (no value)
- These flags control how the **`send()`** or **`recv()`** acts

Flags

Flag	Meaning for `send()`	Meaning for `recv()`
`MSG_DONTWAIT`	Nonblocking send. If the buffer is full, return **`EAGAIN`**.	Nonblocking receive. If no message is available, return **`EAGAIN`**.
`MSG_OOB`	Send a single out of band (high priority) byte	Receive a single out of band (high priority) byte
`MSG_PEEK`	**Invalid flag**	Read the data from the buffer, but don't remove it
`MSG_WAITALL`		Keep reading until you have received the maximum bytes you can hold
`MSG_MORE`	A series of messages will be packed into a single TCP packet	**Invalid flag**
`MSG_NOSIGNAL`	A send on a closed socket will not generate a signal	

Why do we cast to sockaddr*?

- This is the basic **sockaddr** used by socket functions:

```
struct sockaddr {
    unsigned short sa_family; //address family
    char sa_data[14];  //14 bytes of address
};
```

- We often need **sockaddr_in**:

```
struct sockaddr_in {
    short          sin_family;   // AF_INET
    unsigned short sin_port;     // e.g. htons(3490)
    struct in_addr sin_addr;     // 4 bytes
    char           sin_zero[8];  // zero this
};
```

- They start with the same bytes for family, we can cast without a problem
  - C has no inheritance, we can't use a child class

ABET Evaluations

Upcoming

Next time…

- More socket programming

Reminders

- Start working on Project 6
  - It's challenging!
- Pick teams immediately!
- Keep reading LPI Chapters 56, 57, 58, and 59