Week 11 - Friday

COMP 2400

Last time

- What did we talk about last time?
- Low-level file I/O
- Networking

Questions?

Project 5

Quotes

Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer.

Edsger Dijkstra

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 five layers:

Layer	Action	Responsibilities	Protocols
Application	Prepare messages	User interaction	HTTP, FTP, etc.
Transport	Convert messages to segments	Sequencing, reliability, error correction	TCP or UDP
Internet	Convert segments to packets	Flow control, routing	IP
Link	Convert packets to frames	Point-to-point communication between devices on the same network	Ethernet, Wi-Fi
Physical	Transmit frames 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 8 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?!

Netcat

- Netcat (**`nc`**) is a very useful tool for testing networking
- It allows you to interact with network communications through stdin and stdout
- You can run **`nc`** as either a client or a server

`nc` as a client

- We can run **`nc`** as a client, connecting to some waiting server:

```
nc google.com 80
```

- Then, we can type in a command that server is expecting

```
GET / HTTP/1.0
```

- We should see the webpage response from Google

nc as a server

- Alternatively, we can use **nc** as a server to see what a client does when it tries to connect
  - Which can be useful when trying to understand HTTP

```
nc -l 30000
```

- Now, we can type **127.0.0.1:30000** into the address bar of a web browser
  - **127.0.0.1** is a the special loopback IP address that means "this computer"
  - **30000** is the port that **nc** is listening on (in this case)

nc as both!

- We can even use **`nc`** as both a client and a server just for the hell of it
- In one terminal, start **`nc`** as a server:

```
nc -l 50000
```

- In another terminal, connect **`nc`** as a client to that server:

```
nc 127.0.0.1 50000
```

- Now, send stuff back and forth!

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>`

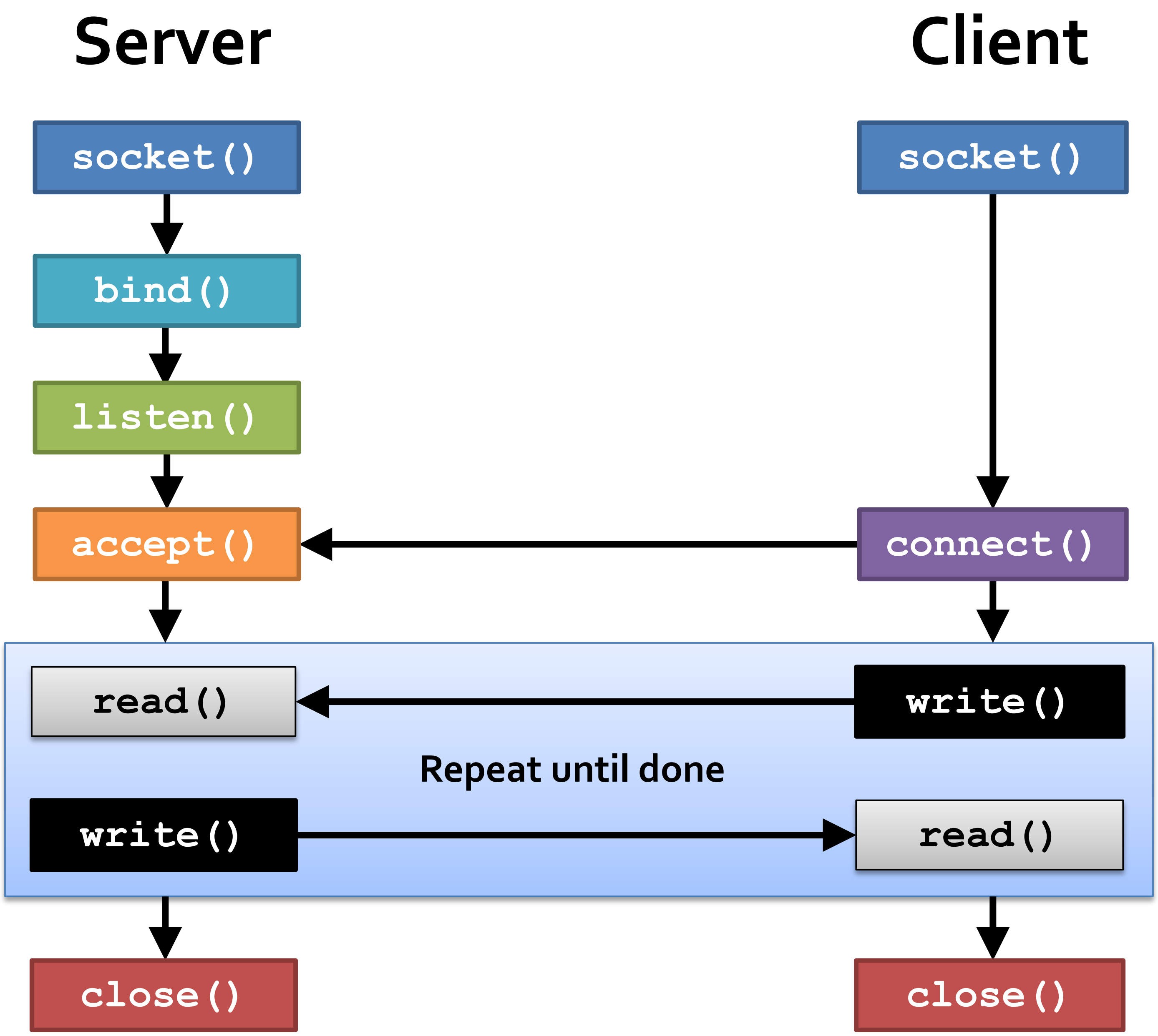
socket()

- 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 = 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 we can also use functions from low-level file I/O
  - **`read()`**
  - **`write()`**
  - **`close()`**
  - Note that different functions are needed to read and write in UDP, but we'll just be doing TCP


Server
Client
socket()
bind()
listen()
accept()
socket()
connect()
read()
write()
Repeat until done
write()
read()
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));
```

Making an address for a client

- 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 just like writing to a file
- The **`write()`** function takes
  - The socket file descriptor
  - A pointer to the data you want to send
  - The number of bytes you want to send
- It returns the number of bytes sent

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

Receiving

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

```
char message[100];
read(socketFD, message, 100);
```

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));
```

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 = 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

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

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

Example 1

- Let's make a client and connect it to **`nc`** acting as a server
- We'll just print everything we get to the screen

Example 2

- Let's make a server and connect to it with **`nc`**
- We'll just print everything we get to the screen

Ticket Out the Door

Upcoming

Next time…

- Finish networking
- File systems

Reminders

- Work on Project 5
- Read Chapters 14 and 15 of LPI