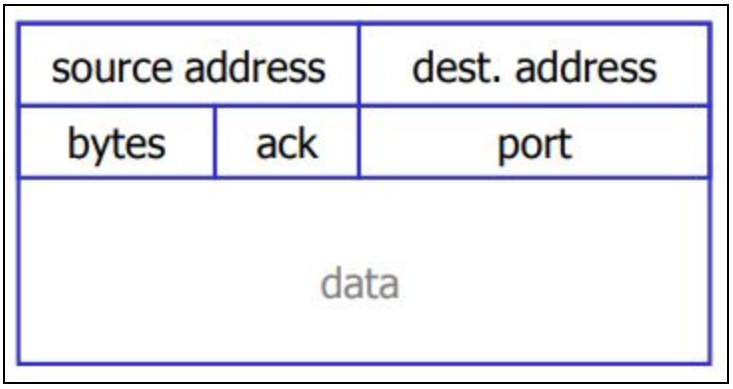
CSCB09 Week 10 Notes

1. **IP Address:**
    - Every computer on the Internet has an internet address, called its **IP (Internet Protocol) address**.
    - An IP address is 4 numbers separated by dots.
2. **Protocols:**
    - Computers use several layers of general protocols to communicate.
    - A **network protocol** is a set of established rules that dictates how to format, transmit and receive data so computer network devices can communicate regardless of the differences in their underlying infrastructures, designs or standards.
    - For example, HTTP is a high-level protocol specific to the web.
3. **TCP/IP:**
    - **TCP** stands for **Transmission Control Protocol**.
    - **TCP/IP (Transmission Control Protocol/Internet Protocol)** is a suite of communication protocols used to interconnect network devices on the internet. It can also be used as a communications protocol in a private network, such as an intranet or an extranet.
    - TCP/IP specifies how data is exchanged over the internet by providing end-to-end communications that identify how it should be broken into packets, addressed, transmitted, routed and received at the destination.
    - TCP/IP requires little central management, and it is designed to make networks reliable, with the ability to recover automatically from the failure of any device on the network.
      I.e. TCP/IP tells the computer how to package up the data.




    - The TCP defines how applications can create channels of communication across a network. It also manages how data is assembled into smaller **packets** before they are then transmitted over the internet and reassembled in the right order at the destination address.
    - A **packet** is the unit of data that is routed between an origin and a destination on any packet-switched network, including the internet.
    - The IP defines how to address and route each packet to make sure it reaches the right destination. Each gateway computer on the network checks this IP address to determine where to forward the message.
    - E.g. When any file is sent from one place to another on the internet, the TCP layer divides the file into chunks of an efficient size for routing. Each

of these packets is separately numbered and includes the internet address of the destination. The individual packets for a given file may travel different routes through the internet. When they have all arrived, they are reassembled into the original file by the TCP layer at the receiving end.

- TCP/IP uses the **client/server model of communication**.
- In the **Client-Server model of communication**, a client process wants to talk to a server process. First, the client must find server by doing a DNS lookup. Then, the client must find process on server by using ports. Finally the client must establish a connection so two processes can talk.
- Advantages of TCP/IP include:
    - It is non-proprietary and, as a result, is not controlled by any single company. Therefore, the internet protocol suite can be modified easily.
    - It is compatible with all operating systems, so it can communicate with any other system.
    - The internet protocol suite is also compatible with all types of computer hardware and networks.
- Sometimes a  packet might not arrive, because of traffic overload or bit corruption. In that case, the receiver asks for missing packets to be resent. While we want to send data as fast as possible, sending data too fast wastes resources. We can use **TCP Congestion Control** to solve this problem.

4. **TCP Congestion Control:**
    - Network congestion may occur when a sender overflows the network with too many packets. At the time of congestion, the network cannot handle this traffic properly.
    - What the TCP Congestion Control does is that it creates a window that allows some sent packets to be ack'd. As more packets are **ack'd**, it increases the capacity of the window. If a packet loss is discovered, it decreases the capacity of the window.
    - **Ack** is the name of a signal that data has been received successfully.
5. **Introduction to Sockets:**
    - Sockets allow communication between two different processes on the same or different machines using standard Unix file descriptors.
    - Similar to pipes, except sockets can be used between processes on different machines.
    - Sockets are built on top of the TCP layer.
    - A Unix Socket is used in a client-server application framework.
    - A **server** is a process that performs some functions on request from a client.
6. **Types of Sockets:**
    - There are two main categories of sockets:
        1. **UNIX domain:** Both processes are on the same machine.
        2. **INET domain:** The processes are on different machines.

- There are three main types of sockets:
    1. **SOCK_STREAM (Stream Sockets):** Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order, "A, B, C". These sockets use TCP for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
    2. **SOCK_DGRAM (Datagram Sockets):** Delivery in a networked environment is not guaranteed. They use UDP (User Datagram Protocol) and are connectionless because they don't need to have an open connection. Instead, you build a packet with the destination information and send it out.
    3. **SOCK_RAW (Raw Sockets):** These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.

7. **Addresses and Ports:**
    - A **socket pair** is the two endpoints of the connection.
    - An endpoint is identified by an IP address and a port.
    - IPv4 (IP version 4) addresses are four 8-bit numbers.
    - A **port** is an endpoint for communication.
    - Ports are identified on a server for each protocol and address, known as the **port number**.
    - We use ports because multiple processes can communicate with a single machine, so we need another identifier.
    - The port assignments to network services can be found using the command **cat /etc/services**.
    - Well-known ports are from 0 - 1023.
    - E.g.:
        - 80 = http
        - 22 = ssh
        - 23 = telnet
    - Registered ports are from 1024 - 49151.
    - E.g.:
        - 2709 = supermon
        - 26000 = quake
        - 3724 = world of warcraft
    - Dynamic (private) ports are from 49152 - 65535.

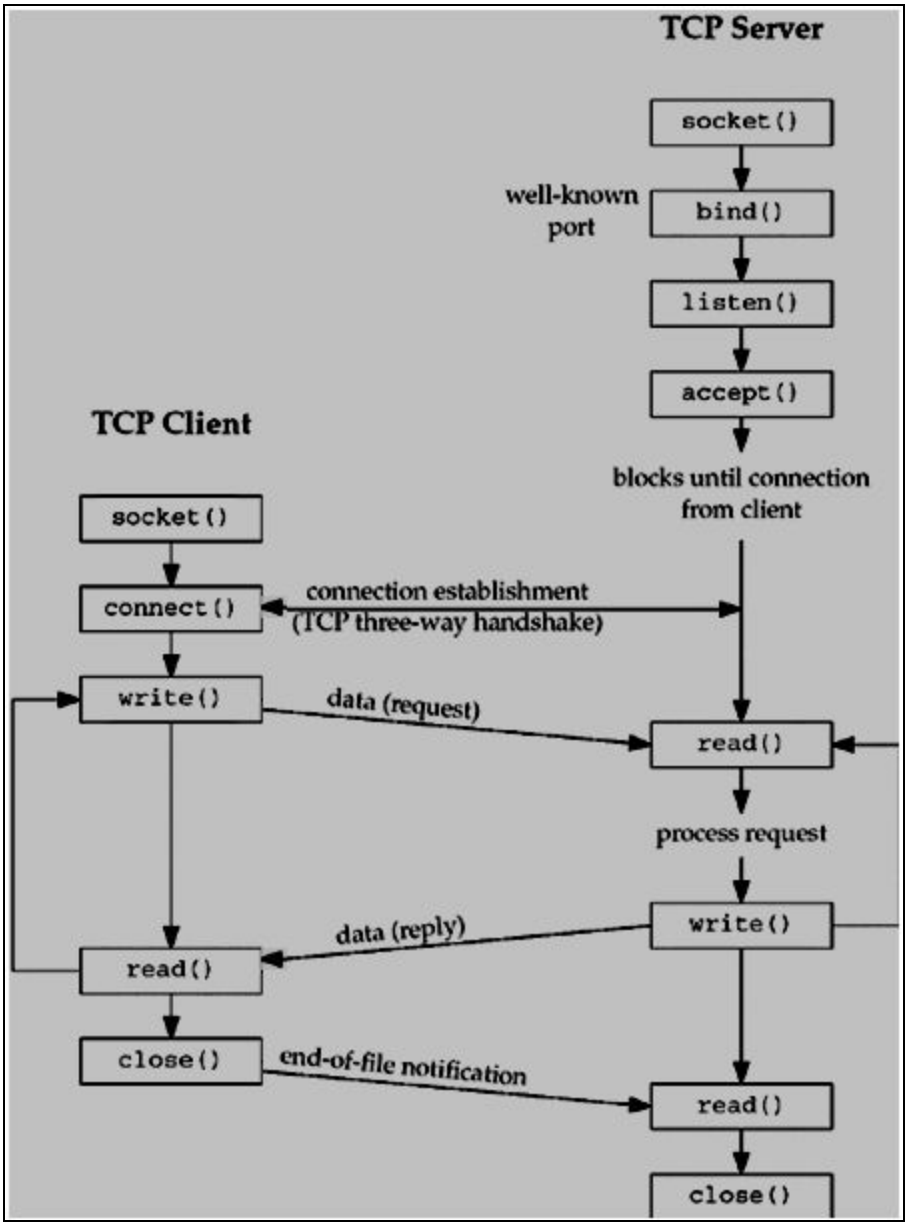
8. **Server Side:**
   - Need to use **#include <sys/socket.h>**
   - Steps:
     1. Create a socket: socket().
     2. Assign a name to a socket: bind().
     3. Establish a queue for connections: listen().
     4. Get a connection from the queue: accept().
   - **Syntax for socket():**
     - This creates a socket. It returns a socket descriptor, an integer (like a file-handle). If the return number is -1, then there was an error.
     - **int socket(family, type, protocol)**
     - **family:** Specifies protocol family:
       - PF_INET: IPv4
       - PF_LOCAL: Unix domain
     - **type:** The communication type.
       - SOCK_STREAM
       - SOCK_DGRAM
       - SOCK_RAW
     - **protocol:** Set to 0 except for RAW sockets.
   - **Syntax for bind():**
     - The bind function assigns a local protocol address to a socket. This function is called by TCP server only. This returns 0 if it successfully binds to the address, otherwise it returns -1 on error.
     - **int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);**
     - **Sockfd:** The number returned by socket().
     - **struct sockaddr_in {**
       **short int sin_family;**
       **unsigned short int sin_port;**
       **struct in_addr sin_addr;**
       **unsigned char sin_zero[8];**
       **};**

| **Attribute** | **Values** | **Description** |
|---|---|---|
| sin_family | PF_INET | Represents the protocol family. |
| sin_port | Service Port | A 16-bit port number in Network Byte Order. |
| sin_addr | IP Address | A 32-bit IP address in Network Byte Order. **Note:** sin_addr can be set to **INADDR_ANY** to communicate on any network interface. |
| sin_zero[8] | Filling | You just set this value to NULL as this is not being used. |

    - **Servaddr:** A pointer to struct sockaddr that contains the local IP address and port.
    - **Addrlen** should be set to sizeof(struct sockaddr).
- **Syntax for listen():**
    - After calling listen, a socket is ready to accept connections.
    - It prepares a queue in the kernel where partially completed connections wait to be accepted.
    - **int listen(int sockfd, int backlog)**
    - **Sockfd:** The number returned by socket().
    - **Backlog:** The maximum number of partially completed connections that the kernel should queue.
- **Syntax for accept():**
    - It blocks, waiting for a connection from the queue, creates a new connected socket, and returns a new descriptor, which refers to the TCP connection with the client.
    - At this point, connection is established between client and server, and they are ready to transfer data.
    - Reads and writes on the connection will use the socket returned by accept.
    - **int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);**
    - **Sockfd:** Is the listening socket.
    - **Cliaddr:** A pointer to struct sockaddr that contains client IP address and port.
    - **Addrlen** should be set to sizeof(struct sockaddr).

9. **Client Side:**
    - Need to use **#include <sys/socket.h>**
    - Steps:
        1. Create a socket: socket(). This is the same as the server side.
        2. Initiate a connection: connect().
    - **Syntax for connect():**
        - The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by servaddr. The server's address and port is specified in servaddr.
        - The kernel will choose a dynamic port and source IP address.
        - Returns 0 on success and -1 on failure setting errno.

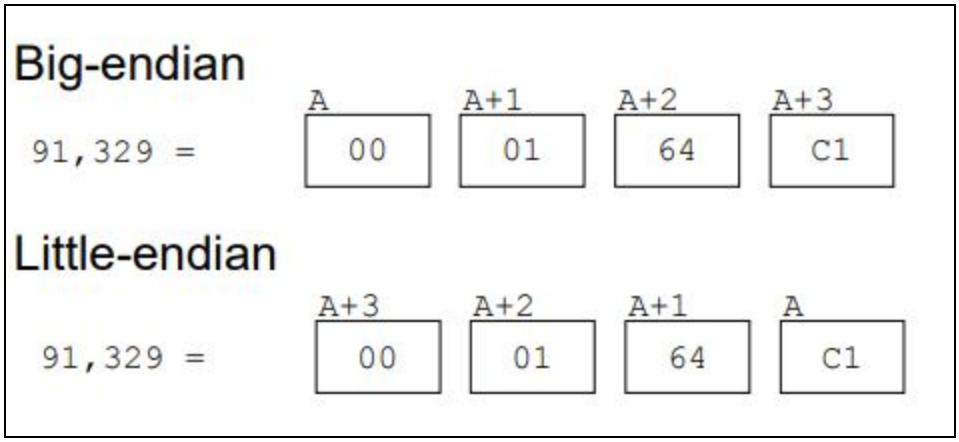
- Initiates the three-way handshake (The picture below).




- **int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);**
- **Sockfd:** The number returned by socket().
- **Servaddr:** A pointer to struct sockaddr that contains destination IP address and port.
- **Addrlen** should be set it to sizeof(struct sockaddr).

**10. Byte order:**

- Unfortunately, not all computers store the bytes that comprise a multibyte value in the same order. Consider a 16-bit internet that is made up of 2 bytes. There are two ways to store this value:
    1. **Little Endian:** In this scheme, a low-order byte is stored on the starting address (A) and a high-order byte is stored on the next address (A + 1).
       I.e. Little Endian byte ordering places the least significant byte first.
    2. **Big Endian:** In this scheme, a high-order byte is stored on the starting address (A) and a low-order byte is stored on the next address (A + 1).
       I.e. Big Endian byte ordering places the most significant byte first.




- Intel is little-endian, and Sparc is big-endian.
- To allow machines with different byte order conventions communicate with each other, we convert numbers to **network byte order** (big-endian) before we send them.
- There are functions provided to do this:

| **Function** | **Description** |
|---|---|
| unsigned long htonl(unsigned long) | This function converts 32-bit quantities from host byte order to network byte order. |
| unsigned short htons(unsigned short) | This function converts 16-bit quantities from host byte order to network byte order. |
| unsigned long ntohl(unsigned long) | This function converts 32-bit quantities from network byte order to host byte order. |
| unsigned short ntohs(unsigned short) | This function converts 16-bit quantities from network byte order to host byte order. |

## 11. Sending and Receiving Data:

- Read and write calls work on sockets, but sometimes we want more control.
- We can use the send function and the receive function instead.
- **Send():**
  - The send function is used to send data over stream sockets or connected datagram sockets.
  - Syntax: **ssize_t send(int fd, const void *buf, size_t len, int flags);**
  - This call returns the number of bytes sent out, otherwise it will return -1 on error.
  - **fd** is the socket descriptor returned by the socket function.
  - **buf** is a pointer to the data you want to send.
  - **len** is the length of the data you want to send (in bytes).
  - **flags:**
    - If flags==0, then send() works like write.
    - If flags is MSG_OOB, then it sends out-of-band data on sockets that support out-of-band communications. The significance and semantics of out-of-band data are protocol-specific.
    - If flags is MSG_DONTROUTE, then it doesn't include routing information in the message.
    - If flags is MSG_DONTWAIT, then it enables a non-blocking operation.
- **Recv():**
  - The recv function is used to receive data over stream sockets or connected datagram sockets.
  - Syntax: **ssize_t recv(int fd, void *buf, size_t len, int flags);**
  - This call returns the number of bytes read into the buffer, otherwise it will return -1 on error.
  - **fd** is the socket descriptor returned by the socket function.
  - **buf** is the buffer to read the information into.
  - **len** is the maximum length of the buffer.
  - **flags:**
    - If flags==0, then recv() works like read.
    - If flags is MSG_OOB, then it requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
    - If flags is MSG_WAITALL, then it requests that the function block until the full amount of data can be returned.
    - If flags is MSG_PEEK, then it peeks at an incoming message. The data is treated as unread and the next *recv*() or similar function shall still return this data.

12. **Close:**
    - The close function is used to close the communication between the client and the server. When finished using a socket, the socket should be closed.
    - Syntax: **int close(int socketfd)**
    - Socketfd is the file descriptor of the socket being closed.
    - This returns 0 on success, otherwise it returns -1 on error.
    - Closing a socket:
        - Closes a connection for SOCK_STREAM.
        - Frees up the port used by the socket.
13. **PF Vs AF:**
    - PF stands for **Protocol Family**. It refers to anything in the protocol, usually sockets and ports.
    - AF stands for **Address Family**. It refers to addresses from the internet, IP addresses specifically.