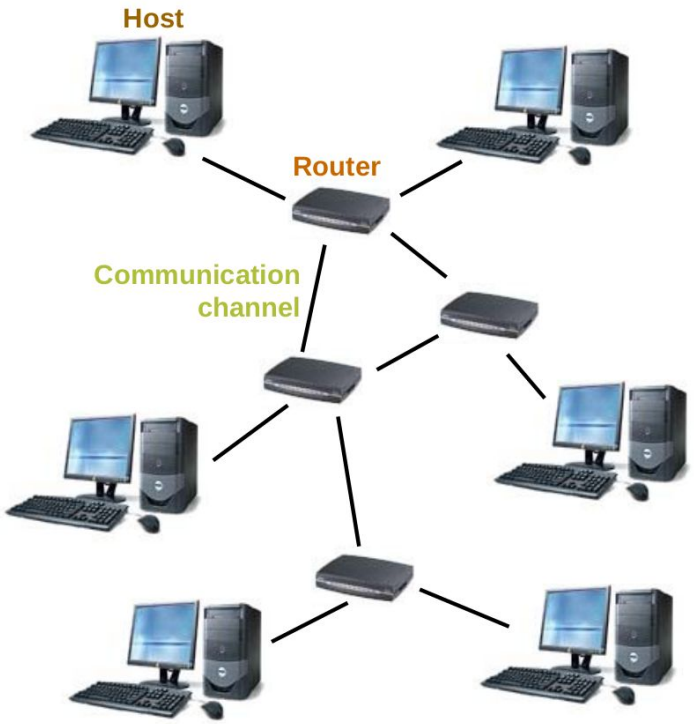
Recitation 11

Socket Programming in C

Introduction

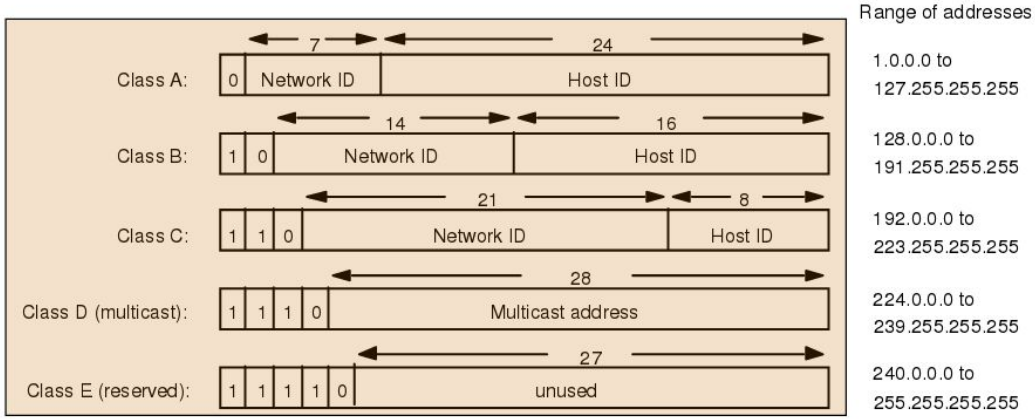
- Computer Network
  - hosts, routers, communication channels
- Hosts run applications
- Routers forward information
- Packets: sequence of bytes
  - contain control information
  - e.g. destination host
- Protocol is an agreement
  - meaning of packets
  - structure and size of packets

  e.g. Hypertext Transfer Protocol (HTTP)



Addresses-IPv4

- The 32 bits of an IPv4 address are broken into 4 octets, or 8 bit fields (0-255 value in decimal notation).
- For networks of different size,
  - the first one (for large networks) to three (for small networks) octets can be used to identify the network, while
  - the rest of the octets can be used to identify the node on the network.



TCP v/s UDP

1. TCP/IP is a suite of protocols used by devices to communicate over the Internet and most local networks. It is named after two of it's original protocols—the Transmission Control Protocol (TCP) and the Internet Protocol (IP).
2. TCP provides apps a way to deliver (and receive) an ordered and error-checked stream of information packets over the network.
3. The User Datagram Protocol (UDP) is used by apps to deliver a faster stream of information by doing away with error-checking.

What do they have in common?

1. Both TCP and UDP are protocols used for sending bits of data—known as packets—over the Internet. Both protocols build on top of the IP protocol. In other words, whether you're sending a packet via TCP or UDP, that packet is sent to an IP address. These packets are treated similarly, as they're forwarded from your computer to intermediary routers and on to the destination.

How does TCP work?

1. When you request a web page in your browser, your computer sends TCP packets to the web server's address, asking it to send the web page back to you. The web server responds by sending a stream of TCP packets, which your web browser stitches together to form the web page. When you click a link, sign in, post a comment, or do anything else, your web browser sends TCP packets to the server and the server sends TCP packets back.

How does TCP work?

2. TCP is all about reliability—packets sent with TCP are tracked so no data is lost or corrupted in transit. This is why file downloads don't become corrupted even if there are network hiccups. Of course, if the recipient is completely offline, your computer will give up and you'll see an error message saying it can't communicate with the remote host.

3. TCP achieves this in two ways. First, it orders packets by numbering them. Second, it error-checks by having the recipient send a response back to the sender saying that it has received the message. If the sender doesn't get a correct response, it can resend the packets to ensure the recipient receives them correctly.

How does UDP work?

1. The UDP protocol works similarly to TCP, but it throws out all the error-checking stuff. All the back-and-forth communication introduce latency, slowing things down.
2. When an app uses UDP, packets are just sent to the recipient. The sender doesn't wait to make sure the recipient received the packet—it just continues sending the next packets. If the recipient misses a few UDP packets here and there, they are just lost—the sender won't resend them. Losing all this overhead means the devices can communicate more quickly.

How does UDP work?

3. UDP is used when speed is desirable and error correction isn't necessary. For example, UDP is frequently used for live broadcasts and online games.

4. For example, let's say you're watching a live video stream, which are often broadcast using UDP instead of TCP. The server just sends a constant stream of UDP packets to computers watching. If you lose your connection for a few seconds, the video may freeze or get jumpy for a moment and then skip to the current bit of the broadcast. If you experience minor packet-loss, the video or audio may be distorted for a moment as the video continues to play without the missing data.

The Subnet Mask

The Subnet Mask

Part of an IP address identifies the *network.* The other part of the address identifies the *host.* A **subnet mask** is required to provide this distinction**:**

158.80.164.3 255.255.0.0

The above IP address has a subnet mask of *255.255.0.0*. The subnet mask follows two rules:

- If a binary bit is set to a **1** (or *on*) in a subnet mask, the corresponding bit in the address identifies the **network**.
- If a binary bit is set to a **0** (or *off*) in a subnet mask, the corresponding bit in the address identifies the **host**.

Looking at the above address and subnet mask in binary:

IP Address:	10011110.01010000.10100100.00000011
Subnet Mask:	11111111.11111111.00000000.00000000

The first 16 bits of the subnet mask are set to *1*. Thus, the first 16 bits of the address (*158.80*) identify the *network*. The last 16 bits of the subnet mask are set to *0*. Thus, the last 16 bits of the address (*164.3*) identify the unique *host* on that network.

The network portion of the subnet mask must be **contiguous**. For example, a subnet mask of *255.0.0.255* is not valid.

* * *

Subnet Mask (contd)

Hosts on the same logical network will have *identical* network addresses, and can communicate freely. For example, the following two hosts are on the same network:

Host A:	158.80.164.100 255.255.0.0
Host B:	158.80.164.101 255.255.0.0

Both share the same network address (*158.80*), which is determined by the *255.255.0.0* subnet mask. Hosts that are on *different* networks cannot communicate without an intermediating device. For example:

Host A:	158.80.164.100 255.255.0.0
Host B:	158.85.164.101 255.255.0.0

The subnet mask has remained the same, but the network addresses are now different (*158.80* and *158.85* respectively). Thus, the two hosts are *not* on the same network, and cannot communicate without a **router** between them. **Routing** is the process of forwarding packets from one network to another.

A trickier example

Host A:	158.80.1.1 255.248.0.0
Host B:	158.79.1.1 255.248.0.0

The specified subnet mask is now *255.248.0.0*, which doesn't fall cleanly on an octet boundary. To determine if these hosts are on separate networks, first convert everything to binary:

Host A Address:	10011110.01010000.00000001.00000001
Host B Address:	10011110.01001111.00000001.00000001
Subnet Mask:	11111111.11111000.00000000.00000000

Remember, the **1** (or **on)** bits in the subnet mask identify the *network* portion of the address. In this example, the first *13 bits* (the 8 bits of the first octet, and the first 5 bits of the second octet) identify the network. Looking at only the first 13 bits of each address:

Host A Address:	10011110.01010
Host B Address:	10011110.01001

Clearly, the network addresses are *not* identical. Thus, these two hosts are on separate networks, and require a router to communicate.

What Are the Default Subnet Masks?

Subnet and Broadcast Addresses

On *each* IP network, two host addresses are reserved for special use:

- The **subnet** (or **network**) address
- The **broadcast** address

Neither of these addresses can be assigned to an actual host.

The **subnet** address is used to identify **the network itself**. A routing table contains a list of known networks, and each network is identified by its subnet address. Subnet addresses contain **all 0 bits in the host portion** of the address.

For example, *192.168.1.0/24* is a subnet address. This can be determined by looking at the address and subnet mask in binary:

IP Address:	11000000.10101000.00000001.00000000
Subnet Mask:	11111111.11111111.11111111.00000000

Note that all host bits in the address are set to *0*.

Broadcast Address

The **broadcast** address identifies *all* hosts on a particular network. A packet sent to the broadcast address will be received and processed by every host on that network. Broadcast addresses contain **all 1 bits in the host portion** of the address.

For example, *192.168.1.255/24* is a broadcast address. Note that all host bits are set to *1:*

IP Address:	11000000.10101000.00000001.11111111
Subnet Mask:	11111111.11111111.11111111.00000000

Broadcasts are one of three types of IP packets:

- **Unicasts** are packets sent from one host to one other host
- **Multicasts** are packets sent from one host to a *group* of hosts
- **Broadcasts** are packets sent from one host to all other hosts on the local network

A router, by default, will **never forward** a multicast or broadcast packet from one interface to another.

A switch, by default, will forward a multicast or broadcast packet **out every port**, except for the port that originated the multicast or broadcast.

Subnetting

Subnetting is the process of creating new networks (or *subnets)* by **stealing bits** from the host portion of a subnet mask. There is one caveat: stealing bits from hosts creates **more** networks but **fewer** hosts per network.

Consider the following Class C network:

192.168.254.0

The default subnet mask for this network is 255.255.255.0. This single network can be segmented, or *subnetted*, into multiple networks. For example, assume a minimum of *10* new networks are required. Resolving this is possible using the following magical formula:

2^n

The exponent '**n**' identifies the number of bits to steal from the host portion of the subnet mask. The default Class C mask (255.255.255.0) looks as follows in binary:

11111111.1111111.1111111.00000000

There are a total of 24 bits set to *1*, which are used to identify the network. There are a total of 8 bits set to *0*, which are used to identify the host, and these host bits can be *stolen*.

Subnetting contd.

Stealing bits essentially involves changing host bits (set to *0* or *off*) in the subnet mask to network bits (set to *1* or *on*). Remember, network bits in a subnet mask **must always be contiguous** - skipping bits is not allowed.

Consider the result if three bits are stolen. Using the above formula:

$2^n = 2^3 = 8 =$ **8 new networks created**

However, a total of 8 new networks *does not* meet the original requirement of at least 10 networks. Consider the result if four bits are stolen:

$2^n = 2^4 = 16 =$ **16 new networks created**

A total of 16 new networks *does* meet the original requirement. Stealing four host bits results in the following *new* subnet mask:

11111111.11111111.11111111.11110000 = 255.255.255.240

Subnetting contd.

In the previous example, a Class C network was subnetted to create *16* new networks, using a subnet mask of *255.255.255.240* (or */28* in CIDR). Four bits were stolen in the subnet mask, leaving only four bits for hosts.

To determine the number of hosts this results in, for each of the new 16 networks, a slightly modified formula is required:

$$2^n - 2$$

Consider the result if four bits are available for hosts:

$2^n - 2 = 2^4 - 2 = 16 - 2 =$ **14 usable hosts per network**

Thus, subnetting a Class C network with a /28 mask creates 16 new networks, with 14 usable hosts per network.

Why is the formula for calculating usable hosts $2^n - 2$? Because it is **never possible** to assign a host an address with all *0* or all *1* bits in the *host* portion of the address**.** These are reserved for the subnet and broadcast addresses, respectively. Thus, every time a network is subnetted, useable host addresses are lost.

What are sockets?

- An Application Programming Interface (API) used for InterProcess Communications (IPC). [A well defined method of connecting two processes, locally or across a network]
- Protocol and Language Independent
- Often referred to as Berkeley Sockets or BSD Sockets

Connections and Associations

- In Socket terms a connections between two processes in called an association.
- An association can be abstractly defined as a 5-tuple which specifies the two processes and a method of communication. For example:
  - ***{protocol, local-addr, local-process, foreign-addr, foreign-process}***
- A half-association is a single "side" of an association (a 3-tuple)
  - ***{protocol, addr, process}***

- Socket creation and usage in C: (all in <sys/sockets.h>
  - socket()
  - socketpair()
  - bind()
  - listen()
  - accept()
  - connect()
  - read()
  - write()
  - close()

Create A Socket

- **int socket (int domain, int type, int protocol)**
  - Domain is AF_UNIX or AF_INET
    - AF_INET for processes on different or on the same host
    - AF_UNIX for processes on the same machine only
  - Type is the style of communication:
    - SOCK_STREAM: connection-oriented „stream“: no record boundaries, guaranteed delivery
    - SOCK_DGRAM: connection-less „datagram“: sending individual records, delivery is not guaranteed
  - Protocol allows the specification of the underlying protocol to be used;
    - AF_INET and stream goes with TCP
    - AF_INET and datagram goes with UDP
    - The sysem will chose the most appropriate when specifying 0
  - Our application uses AF_INET and SOCK_STREAM

Assigning a name to a socket

- **int bind(int sockfd, struct sockaddr *my_addr, socklen_t, addrlen)**
  - Assigns a name to a socket, so that it can be referenced by another process
  - In the AF_UNIX domain, a name is established using the file system
  - In the AF_INET domain a name consists of an internet (IP) address and a port number
  - Example: 132.159.32.1:80
  - Ports below 1024 are reserved

Connect to another socket

- **int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);**
  - Specify the local socket (sockfd) and the name of the remote socket
  - Remote socket must have been bound to that name
  - When connect succeeds, the connection is set up and you can read/write to the socked specified by sockfd

Setting up a server to accept connections

- Create (socket()) and assign name (bind())
- Change to listening mode
  - int listen(int s, int backlog)
  - backlog is the length of the queue where unhandled connection requests are placed
  - non-blocking
- Wait for connections
  - int accept (int s, struct sockaddr *addr, socklen_t *addrlen);
  - blocking
  - returns a *new* socket used to talk to client, client's address information is placed in name
  - The old socket is used to handle new connection requests

Read/Write Send/Receive

- ssize_t read (int fd, void *buf, size_t count);
- ssize_t write (int fd, const void *buf, size_t count);

- int send (int s, const void *msg, size_t len, int flags);
- int recv (int s, void *buf, size_t len, int flags);

- Also: int recv(int s, void *buf, size_t len, int flags);
  - Recvfrom, recvmsg, sendto sendmsg…