Reading For Next Three Lectures

- Textbook
  - The Client Server Programming Model - Web Servers
    - 2nd ed: 11.1-11.5
    - 1st ed: 12.1-12.5

IPC Basic Idea

- Communication for processes that don’t share memory
  - could be on same processor (shared physical, but not virtual) memory
  - could be on different processors connected by a network
  - same communication mechanism for both cases

- Unformatted data transfer
  - message payload is the data to be transferred from sender to receiver
  - sender assembles the payload as an array of bytes -- like a file block
  - receiver translates byte array back into programming-language types

- Asynchronous control transfer
  - send initiate sending message payload to receiving process, but do not wait
  - recv receive next available message, either blocking or not if no data waiting

- Naming
  - sender needs to name the receiving process
  - receiver needs to name something --- options?

Client-Server Model

- server is a process that
  - waits to receive network messages from clients
  - processes the message in some way
  - sends a response message back to client
  - client is a process that sends request messages to server

- client is a process that
  - sends requests to server and waits for response

- configuration
  - many clients, one server
  - server is often client for another server (e.g., browser, web-server, database)
Basic communication-endpoint naming

- Internet Protocol address (IP address)
  - 32-bit (IPv4) or 128-bit number
    - we write IPv4 addresses as four numbers divided by . – IPv6 is 8 divided by :
  - names machines **nodes** in an internet (there are many internets, more later)
  - same-machine communication sent to 127.0.0.1 (called **localhost**)

- Port
  - 16-bit number
  - names a process, unique to a single node
  - low numbers are privileged and for standard services (e.g., imap, smtp, httpd)

Addressing a message
- destination address is IP address and port number of target process
- source address is IP address and port number of sending process
- both are included as part of the **message header**

```
IP header       payload

msg: src dst ... byte 0
```

Simple example

- sending process
  - allocates message buffer for payload
    - copies payload data into buffer
    - issues send

- receiving process
  - issues recv to wait on port
    - copies payload data out of buffer and gets source address

Determining IP address and port number

- IP Address
  - usually use the **IP Domain Name** a hierarchical name string
    - e.g., cascade.cs.ubc.ca
  - translated to IP Address by the **Domain Name Service (DNS)**
    - a hierarchical name server that is distributed throughout internet
    - every node is configured with the IP address of a DNS node implemented by its ISP
    - ISP is internet service provider
    - first step in communication is to contact DNS to get IP address for domain name

- port number
  - some services resident on well-known ports
  - you could implement your own name server for ports
  - via a virtual connection using protocols like TCP

Communication Protocols (OSI model)

- a protocol is
  - a **specification** of message-header formats of handing of messages
  - an implementation of the specification

- layering of abstraction
  - several different protocols involved in sending a message
  - layered into a hierarchy

- the 7-layer OSI model (e.g., 802.11 web browsing)
  - application HTTP get and post etc. web-server messages
  - presentation TCP
  - session TCP
  - transport TCP connections, streams and reliable delivery
  - network routing IP routing using IP address and port #
  - data link LLC/MAC data framing and signalling to access airspace
  - physical PHY radio
Transport protocols

- **UDP**
  - send/receive datagrams
  - addressing is just IP address and port # as we have seen
  - best-effort transport
  - but, if any router queue in network is full, message will be dropped

- **TCP**
  - send/receive streams
  - addressing using virtual connection between two hosts
  - reliable, in-order delivery of stream packets
  - sending rate adapts to available bandwidth
  - reliability provided by software: receipt acknowledgement and retransmission

Routing packets in the Internet

- **What is the Internet**
  - collection of many thousands of networks bridged to each other
  - backbone routers connect each of these networks
    - routing protocol is called BGP (border gateway protocol)

- **Nodes are directly connected to a switch**
  - physical routing using, for example, ethernet or 802.11 MAC addresses
  - address resolution protocol (arp)
    - broadcast protocol to resolve IP addresses on local network
  - first step in sending an IP packet is to send it to your switch
  - last step in receiving a packet for switch to send it to destination

- **In the middle is IP and BGP routing**
  - each switch has set of output ports and a routing map
  - map lists which IP addresses are accessible on which port
  - packet is routed, step by step, eventually reaching edge router for target node
  - packets have a time-to-live counter, decremented at each step, to ensure they don’t travel forever in the Internet without reaching their destination
  - see the route using traceroute

Sockets

- **A socket is a OS abstraction for a communication endpoint**
- **The first step in communicating is to create a socket**
  - it is like opening a file (which returns a file descriptor)
  - socketDescriptor = socket ( domain, type, protocol )
  - sd = socket ( PF_INET, SOCK_STREAM, 0 )
  - sd = socket ( PF_BLUETOOTH, SOCK_STREAM, BTPROTO_FRCOMM )
  - sd = socket ( PF_INET, SOCK_DGRAM, 0 )

- **What happens next depends on send/recv or protocol**
  - basically, assign an network address to socket and then start communicating

- **To send via UDP**
  - create a destination address (ip address or domain name)
  - sendto (sd, buf, bufLen, destAddr, destAddrLen )

- **To receive via UDP**
  - create a receive address (port)
  - bind (sd, addr, addrLen)
  - recvfrom (sd, buf, bufLen, 0, srcAddr, srcAddrLen )
TCP Virtual Connections

- Designed for long-term flows of data between a pair of endpoints
  - Most traffic on internet is TCP — otherwise Internet would not work
  - Bob Metcalfe (Ethernet inventor) eats words “catastrophic collapse” in 1995

- Basic idea
  - In setup phase protocol picks send and receive port numbers for the flow
  - Sending application divides flow into messages
  - Sending OS (TCP) divides messages into packets, giving each a sequence number
  - Receiver sends ACK messages listing sequence numbers it is missing (roughly)
  - Sender retransmits these packets
  - Sender rate starts slow, gradually increases, slows when packet-loss rate too high

**Full Protocol Diagram**

**TCP Steps on Caller**

- Setup socket to send connection request
- Send connection-request packet
- Send / receive data on socket

```
struct sockaddr_in dstAddr;
unsigned long dstIP = htonl (0xAB112090);
memset (&dstAddr, 0, sizeof (dstAddr));
dstAddr.sin_family = PF_INET;
memcpy (&dstAddr.sin_addr.s_addr, &dstIP, sizeof (dstIP));
dstAddr.sin_port = htons (7891);
connect (so, (struct sockaddr *) &dstAddr, sizeof(dstAddr));
```

```
connect (so, (struct sockaddr *) &dstAddr, sizeof(dstAddr));
```

```
send (so, buf, length, 0);
length = recv (so, buf, length, 0);
```

Adapted from: Computer Systems: A Programmer's Perspective
TCP steps on Server

- setup address connection-listening address
  ```c
  struct sockaddr_in conAddr;
  memset(& conAddr,0,sizeof(conAddr));
  conAddr.sin_family = PF_INET;
  conAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  conAddr.sin_port = htons(7891);
  bind (so, (struct sockaddr *) & conAddr,sizeof(conAddr))
  ```

- setup socket to listen for connection requests
  ```c
  listen (so, maxNumberOfPendingRequestsQueued)
  ```

- block waiting for connection requests to arrive
  ```c
  struct sockaddr_in caller;
  int cl_len = sizeof (caller);
  int callerSo;
  callerSo = accept (so, (struct sockaddr *)&caller, &cl_len);
  ```

- send/recv messages to/from caller using callerSo socket

A few additional details

- purpose of bind step at server
  - each machine typically has multiple network interfaces
  - and so it might have multiple IP addresses
  - bind picks the one to be used for this session
  - bind also picks the connection-request port number (e.g., port 80)

- finding out who called
  ```c
  unsigned long callerIP = ntohl(caller.sin_addr.s_addr);
  unsigned short callerPort = ntohs(caller.sin_port);
  ```

- disconnecting
  ```c
  close (callerSo);
  ```

Complete Example (caller)

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

int main()
{
  int fd;
  fd = socket(PF_INET, SOCK_STREAM, 0);

  struct sockaddr_in remoteAddr;
  unsigned long remoteIP = htonl(0x7F000001);
  memset(&remoteAddr, 0, sizeof(remoteAddr));
  remoteAddr.sin_family = PF_INET;
  remoteAddr.sin_addr.s_addr = htonl(0x7F000001);
  remoteAddr.sin_port = htons(7891);

  struct sockaddr_in caller;
  int cl_len = sizeof (caller);
  int callerSo;
  callerSo = accept (so, (struct sockaddr *)&caller, &cl_len);

  unsigned long callerIP = ntohl(caller.sin_addr.s_addr);
  unsigned short callerPort = ntohs(caller.sin_port);

  close (callerSo);
}
```
if (connect(fd, (struct sockaddr *) &remoteAddr, sizeof(remoteAddr)) < 0) {
    perror("Connection failed");
} else {
    char *msg = "Hi there\n";
    time_t ltime;
    char buff[256];
    time(&ltime);
    write(fd, &ltime, sizeof(ltime));
    write(fd, msg, 10);
    int amt;
    amt = read(fd, buff, 256);
    while( amt > 0) {
        printf("Buffer %s", buff);
        amt = read(fd, buff, 256);
    }
}

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
int main()
{
    int fd;
    fd = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in callerAddr;
    memset(&callerAddr, 0, sizeof(callerAddr));
    callerAddr.sin_family = PF_INET;
    callerAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    callerAddr.sin_port = htons(7891);
    bind(fd, (struct sockaddr *) &callerAddr, sizeof(callerAddr));
    listen(fd, 4);
    struct sockaddr_in caller;
    while(1) {
        int cl_len = sizeof(caller);
        int callerFD;
        callerFD = accept(fd, (struct sockaddr *)&caller, &cl_len);
        char *buff[256];
        int amt;
        time_t rtime;
        recv(callerFD, &rtime, sizeof(time_t), 0);
        amt = recv(callerFD, buff, 256, 0);
        struct hostent *hp;
        hp = gethostbyaddr((char *) &caller.sin_addr.s_addr, sizeof(long), PF_INET);
        amt = snprintf(buff,256, "%s\n (\%x) \%x at %s", hp->h_name, ntohl(caller.sin_addr.s_addr), (long) ntohs(caller.sin_port), ctime(&rtime));
        send(callerFD, buff, amt + 1, 0); sleep(10);
        send(callerFD, "Bye\n", 5, 0);
        close(callerFD);
    }

    struct hostent *hp;
    hp = gethostbyaddr((char *) &caller.sin_addr.s_addr, sizeof(long), PF_INET);
    int cl_len = sizeof(caller);
    int callerFD;
    callerFD = accept(fd, (struct sockaddr *)&caller, &cl_len);
}
Other useful functions

- `inet_aton()`  string to network address
- `inet_ntoa()`  network address to string
- `gethostbyname()`  lookup host by IP domain name (get hostent)
- `gethostbyaddr()`  lookup host by IP address

A naive web server

1. wait for request
2. process request, read from file
3. wait for file read to complete
4. may repeat 2 & 3 several times
5. prepare reply and send
6. goto step 1

What is wrong?