1. Network Terminology

2. TCP/IP Network Layers
   - TCP/IP vs OSI

3. Socket Programming
   - Opening a Socket
   - Bind, Listen, Accept
   - Connect, Send, Recv
   - Assignment 1
   - UDP
   - Odds and Ends
Data Communication Networks

- Data (digital data - chunks of bits)
- Communication (exchange of data)
- Network (the facilitator)

For the purposes of this course:
- bits (chunks, streams, or packets) exchanged between computers;
- The Internet
Network Components

- Hosts; Network Infrastructure
- Postal Network: sender receiver; USPO
- Telephone network: caller and callee (subscribers); Telephone companies
- Mobile Telephone: subscribers; service provider.
- Internet participants
  - Host computers: Clients and servers
  - Infrastructure: ISPs, Backbone operators
  - Infrastructural elements: Links, routers
Network Scale

- Personal area networks (PAN)
- LAN (Local)
- MAN (Metropolitan)
- WAN (Wide)
- Most often just LAN and WAN
- PANs are expected to become prevalent soon
Network Terminology
TCP/IP Network Layers
Socket Programming

LAN

(a) Computer connected through Cable
(b) Multiple Computer connected in a ring through Cable
WAN - Interconnection of LANs
Type of Connection

- Connection oriented, Connectionless
- Connection oriented: Telephone (circuit switching)
- Connectionless: Snail mail, telegram
- Internet: Packet switching (connectionless)
- Virtual Connections are also possible in a connectionless network.
Router C makes a choice to forward packets to E and not to D.
Connectionless Services

- Datagram Subnet
- Virtual Circuit
Datagram Routing

A's table
initially later
A: - A: -
B: B B: B
C: C C: C
D: B D: B
E: C E: B
F: C F: B

C's table
A: A
B: A
C: -
D: D
E: E
F: E

E's table
A: C
B: D
C: C
D: D
E: -
F: F

Dest. Line
Virtual Circuit Subnet

- Process P1
- Process P2
- Process P3
- Carrier's equipment
- Router

A's table:
- H1: 1
- H3: 1

C's table:
- A: 1
- C: 1
- E: 1

E's table:
- C: 1
- F: 1
Structure of the Internet

- Regional ISP
- Backbone
- NAP
- Network Access Provider (POP)
- Corporate LAN
- Server farm
- Router
- Telephone system
- Client
- Server

Diagram illustrating the structure of the Internet.
Requirements

- Send a packet from one computer to another
- Send data from one process to another (most often running on different computers)
- The requirements are met by using different layers - each with a very specific responsibility.
- Each layer follows some rules - protocols
- First requirement: met using three layers: Physical, Datalink and Network
- Second requirement: two additional layers - application and transport
Network Layer Models

- TCP/IP Model
  - Application
  - Transport (TCP / UDP)
  - Network (IP)
  - Datalink / MAC (Medium Access Control)
  - Physical layer

- OSI (Open Systems Interconnection) Model
  - Adds two more layers between Application and Transport
  - Presentation Layer, Session Layer
Physical Layer

- Hardware for physically carrying data
- Over wires, or wireless links
- modems, ethernet/wifi card, etc.
Data Link Layer

- send a packet of bits from one computer to another when a direct connection exists between the computers
- Depends on the nature of the physical medium used
- Two broad categories
  - Physical medium is not shared: Data Link (DL) protocols
  - Physical medium is shared: Medium Access Control (MAC) protocols
    - Every computer that shares the physical medium should have a unique MAC layer address
    - Contention for channel access needs to be addressed
    - MAC = DL + MAC layer addressing + Regulating access to shared channel.
Network Layer

- To send a packet (Internet Protocol or IP packet) from one computer to another
- A unique *network address* (IP address) for every computer
- Routers relay packets over *multiple hops*
- Internet protocol (IP) - every computer has a unique IP address
Application and Transport Layer

- Protocols for applications/processes running on different computers to communicate with each other
- Client-server Application model
  - (Client establishes a connection with the server)
  - Client sends a request
  - Server responds
  - (Close connection)
- Some applications: E-mail, WWW, IM, FTP, File sharing, · · ·
- Transport layer
  - Provides a “connection” between processes running on different computers
  - Takes care of many “low-level details” for creating, maintaining and closing connections
  - Different processes running on the same computer should be differentiated by unique process addresses (port numbers)
Every computer connected to the Internet has a unique IP address.

Corresponding to each IP address, there may be many application processes willing to accept connection requests at some port number.

Servers *listen* - wait for incoming attempts to establish a connection (accept connection requests).

Clients *initiate* connection requests.

Once a connection is established, both client and server can *send* and *receive* (any number of) bytes.
Client Server Applications

- **Server**
  - Listen (at some port number)
  - If a connection attempt is sensed
    - accept connection request
    - process query (send response)
    - close connection

- **Client**
  - request connection
  - send query
  - receive response
  - close connection
Client Server Applications

**Server**
- 0  Listen at some port number
- 2  Accept connection request
- 4  Process Query
- 5  Send Response
- 7/8 Close Connection

**Client**
- 1  Request Connection
- 3  Send Query
- 6  Receive response
- 7/8 Close connection
Addressing servers and Clients

- Client needs to know IP address and port number of the server.
- Usually clients know only the domain name (for e.g., yahoo.com)
- DNS (domain name system/service) — an application that translates domain names to IP addresses (like 411 Directory service);
- Port number depends on the type of application (follow standards)
- Connection request made to IP address and port number of the server, and conveys IP address and port number of the client.
Sockets and Socket Programming

- Software library for network application development;
- Sockets are similar to file handles
- File handles bound to a file name (and path); Sockets bound to socket address;
- Socket address has two components — an IP address and a port number
- Two types of sockets - TCP (transport control protocol) and UDP (user datagram protocol)
- TCP sockets can be connected.
- The socket library provides various system calls like socket(), bind(), listen(), connect(), accept(), send(), recv(), close()
Socket (System) Calls

- `sd = socket(OPTIONS)`. Creates a socket. `sd` is a handle to the socket.
- `bind(FROMADDRESS)` (local socket address).
- `listen()`
- `connect(TOADDRESS)` (remote socket address).
- `newsd = accept()` (accept() returns a "connected" socket)
- `send(buffer, num_bytes)`.
- `recv(buffer, num_bytes)`.
- `close()`.

These functions are the *interfaces provided by the transport layer* (to the application layer above).
Transport Layer

- Tasks: Establish connections; reliable delivery of application data; differentiating between different communication instances using port numbers;
- Break up application data into chunks, add a TRANSPORT header to each chunk
- Transport packet = TRANSPORT header + application data chunk
- Send / receive transport packets to / from the network layer.
- Network (IP) layer does not guarantee reliable delivery of packets (packets can be lost, can arrive out of order).
Transport Layer

- Flow control: what happens if the sender has a high bandwidth connection and the receiver has a low bandwidth connection?
- Transport layer does not care about the location of the destination (that’s the job of the network layer).
Trans. Hdr (1546, 80, flags, sequence/ack no. ...)

Net. Hdr (120.5.246.47, 201.234.56.78, ...)

Trans. Packet = Trans Hdr + App Data

Net. Packet = Net Hdr + Trans Packet
Network Layer

- IP packets created by the sender — by adding IP header to transport packet (IP packet = IP header + transport packet)
- Header specifies IP address of sender and final destination
- IP packet sent to next hop;
- when a host receives an IP packet, the final destination can be
  - itself — accept and provide contents (transport packet) to higher layer;
  - a host that is directly reachable (one hop away), or
  - a host that is not directly reachable (may be multiple hops away)

- Next hop can be the final destination, or a router on the path to the final destination
Lower Layers

- DL / MAC layer
  - Enclose IP packet (to be sent to next hop) in a DL / MAC frame
  - MAC Frame = MAC Header + Payload (IP packet) + footer + framing flags at either end.
  - MAC header includes MAC addresses of sender and receiver (one hop away from sender)
  - MAC/DL footer for CRC (error detection)
  - Provide the MAC frame to hardware (Ethernet card, modem) using interfaces to the hardware (drivers)

- Physical Layer: Convert bits to electrical/electromagnetic signals and send them over the link
Network Layers - Functional Description

**AL**
Data Packet

**TL**
- Destination IP
- Destination Port
- TH: TCP HEADER
- Data
- TH: TCP HEADER
- Origin IP
- TH: TCP HEADER
- Origin Port

**NL**
- Next Stop DL Address
- NH: IP HEADER
- TH: TCP HEADER
- Data
- NH: IP HEADER
- TH: TCP HEADER
- Origin DL Address

**DL**
- DH: MAC HEADER
- NH: IP HEADER
- TH: TCP HEADER
- Data

**PL**
Conversion to Electrical Signals

**TH** - TCP HEADER, **NH** - IP HEADER, **DH** - MAC HEADER
OSI vs TCP/IP

**OSI Model**
- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

**TCP/IP Model**
- Application
- Transport
- Network
- Data Link
- Physical
Presentation Layer

- Syntax and semantics of information transmitted
- The same sequence of bytes can have different interpretations in different machines/CPUs — Big-endian vs little-endian
- All machines need to agree on some byte order (for representing 32-bit IP address and 16-bit port numbers)
- Network order, host order
- For TCP/IP translations are provided by some helper functions - htons(), htonl(), ntohs(), ntohl() included in the socket library.
Session Layer

- Establishing sessions
- Dialog control (who goes next?)
- Synchronization (continue seamlessly from previous connection)
- In TCP/IP model, dialog control and synchronization is the responsibility of the application layer
- Examples?
Most applications can be abstracted to a sequence of
- `connect()`, `send()`, `recv()` by client
- `accept()`, `recv()`, `send()` by server

Examples
- WWW (uses TCP)
- Telnet (uses TCP)
- FTP (uses TCP)
- SFTP (TCP or UDP)
- DNS (UDP)
- SMTP (Simple Mail Transfer Protocol) - TCP
Web Browser / Web Server

- Say “http://www.abc.com/def/page.html” entered in the browser address bar
- Browser parses string - separates domain name - “www.abc.com” and file name / path - “/def/page.html”
- DNS (uses a function call gethostbyname() ) to discover IP address from domain name
- Server port number 80 for HTTP
- Create a socket, send connection request to server
- Server accepts connection
- Client sends application data “HTTP 1.1 GET /def/page.html”
- Server sends page.html as response
- Client and server close connection.
Physical Layer

- Representation of signals
- Modulation / demodulation
- Multiplexing
DL / MAC Layer

- Framing
- error control (detection and correction)
- protocols for reliable delivery
- taking full advantage of available bandwidth (pipelining)
- Additional MAC layer for scenarios with shared medium
- addressing; resolving contention for channel
- We will look at several types of shared media (shared cable/wire, wireless, satellite link)
Network Layer

- Routing and Forwarding
- Collaborative; many intermediaries involved in deciding how packets will be relayed.
- Congestion control.
- IP specific features
Transport Layer

- Establishment and termination of connections
- Maintaining a reliable connection
- Flow control
- Making best use of available bandwidth (pipe-lining)
- Help network layer control congestion
- TCP and UDP
Opening a Socket

```
int socket(int, int, int);
Output: socket handle
Inputs: domain, type, protocol
TCP:

int sd;
sd = socket(AF_INET, SOCK_STREAM, 0);

UDP:

sd = socket(AF_INET, SOCK_DGRAM, 0);
```

Returns an integer (handle) sd or -1 on failure.
int bind(int, sockaddr *, int);
Inputs: socket handle, pointer to address structure, size of address structure

struct sockaddr {
  unsigned short sa_family;
  char sa_data[14];
}

struct sockaddr_in {
  short int sin_family;
  unsigned short sin_port;
  struct in_addr sin_addr;
  unsigned char sin_zero[8];
}

struct in_addr {unsigned long s_addr;}
Address Format

```c
unsigned short port = 4345;
sockaddr_in soaddr;
soaddr.sin_family = AF_INET;
soaddr.sin_port = htons(port);
memset(&(soaddr.sin_zero), 0, 8);

htons() - host-to-network byte order conversion for shorts
htons(), htonl(), ntohs(), ntohl()
```
Specifying IP Address

Automatically fill local IP Address

soaddr.sin_addr.s_addr = INADDR_ANY;

IP address specified as a string

char * IPaddr = "123.134.245.123";
inet_aton(IPaddr, &soaddr.sin_addr);

IP from domain name through a DNS query

hostent * h;
char * dname = "yahoo.com";
h = gethostbyname(dname);
soaddr.sin_addr = (struct in_addr *)h->h_addr;
Getting IP Address from Domain Name

```c
struct hostent * = gethostbyname(char *);

struct hostent {
    ... char ** h_addr_list;
}
#define h_addr h_addr_list[0];

h_addr is a pointer to a sequence of four characters
Cast h_addr to a `in_addr` pointer
Recall that `in_addr` is just `unsigned long`
```
int bind(int, sockaddr*, int);

int check = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));

bind() returns -1 on failure.
int listen(int, int);
Inputs : socket handle, BACKLOG

check = listen(sd,BACKLOG);

listen() returns -1 on error;
BACKLOG is the number of connection requests that can be queued
int accept(int, void *, int *);
Inputs: socket handle, pointer to client address, size of returned address.

sockaddr_in clientaddr;
int addressesize, newsd;
newsd = accept(sd, (void *)&clientaddr, &addressesize);
accept() returns -1 on error
int sd, addsize, backlog=10, newsd;
sockaddr_in serveraddr, clientaddr;
unsigned short port = 4349;
sd = socket(AF_INET, SOCK_STREAM, 0);
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
serveraddr.sin_addr.s_addr = INADDR_ANY;
memset(&(serveraddr.sin_zero), 0, 8);
check = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
check = listen(sd, backlog);
while(1) {
newsd = accept(sd, (void *) &clientaddr, &addsize);
//connection specific recv() and send()
//process request by
//recv() ing from newsd and
//send() ing using newsd;
}
close(sd);

Typically connection specific tasks performed in a new thread / process so that the server can go back to waiting on accept().
int connect(int, sockaddr *, int);
Returns -1 on failure
Inputs : socket handle, destination address, size of address.
int csd;
unsigned short port = 4349; // dest port
char * serverIP = "123.134.245.123"; // dest IP
sockaddr_in serveraddr;
csd = socket(AF_INET, SOCK_STREAM, 0);
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
inet_aton(serverIP, &serveraddr.sin_addr);
connect(csd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));

inet_aton() - converts character string "X.Y.Z.W" to unsigned long
Send and Receive

int send(int, const void *, int, int);
Output: Number of bytes sent, 0, or -1
Input: socket handle, message buffer, buffer length in bytes, and FLAGS

int recv(int, const void *, int, int);
Output: Number of bytes received, 0, or -1
Input: socket handle, message buffer, maximum buffer length in bytes, and FLAGS
Client with `Send()` and `Recv()`

```c
int csd, check, numbytes;
unsigned short port = 4349;
char * serverIP = "123.134.245.123";
char sbuf[256], rbuf[256];
sockaddr_in serveraddr;
//create application specific query in sbuf ...
csd = socket(AF_INET, SOCK_STREAM, 0);
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
inet_aton(serverIP, &serveraddr.sin_addr);
```
connect(csd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
numbytes = send(csd, (const void *)sbuf, 100, 0);
shutdown(csd, SHUT_WR);
numbytes = recv(csd, (const void *)rbuf, 256, 0);
close(csd);
//Process response (in rbuf) from server ...

shutdown(csd, SHUT_RD), shutdown(csd, SHUT_RDWR) can also be used. shutdown(csd, SHUT_RDWR) closes both reading and writing.
send() and recv() in Server

```c
char rbuf[256], sbuf[256];
int numbytes;
numbytes = recv(newsd, (const void *)rbuf, 256, 0);
//create application specific response in sbuf
numbytes = send(newsd, (const void *)sbuf, 200, 0);
close(newsd);
```
int main(int argc, char* argv[]) {
    int sd, newsd; char buf[256];
    struct sockaddr_in serveraddr, clientaddr;
    socklen_t sasize = sizeof(struct sockaddr_in);
    sd = socket(AF_INET, SOCK_STREAM, 0);
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(atoi(argv[1]));
    serveraddr.sin_addr.s_addr = INADDR_ANY;
    bind(sd, (struct sockaddr*)&serveraddr, sasize);
    listen(sd, 10);
    while(1) {
        newsd=accept(sd,(struct sockaddr*) &clientaddr,&sasize);
        recv(newsd, buf, 256, 0); send(newsd, "Hello", 6, 0);
        close(newsd); }
    close(sd); }

int main(int argc, char* argv[]) {
    int csd;
    char buf[256];
    struct sockaddr_in serveraddr;
    socklen_t sasize = sizeof(struct sockaddr_in); //16
    csd = socket((AF_INET, SOCK_STREAM, 0);
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(atoi(argv[2]));
    inet_aton(argv[1], &serveraddr.sin_addr);
    connect(csd, (struct sockaddr *)&serveraddr, sasize);
    send(csd, "Hi", 3, 0); shutdown(csd, SHUT_WR);
    recv(csd, buf, 255, 0);
    close(csd);
}
send() and recv() can be used only after *connection has been established*

Obviously, you cannot use send() and recv() for UDP
Use sendto() and recvfrom() instead

```c
int sendto(int sd, const void *msg, size_t len, int flags, const struct sockaddr *to, int len);
```

```c
int recvfrom(int sd, void *buf, size_t len, int flags, struct sockaddr *from, int *len);
```

For POSIX-2 compliance the last (6th) parameter in sendto() and recvfrom() should be socklen_t instead of int
UDP Client and Server

Server:
- \( sd = \) socket();
- bind(sd, address);
- recvfrom() - equivalent to listen() and accept()
- process request; create response
- sendto()

Client
- \( sdc = \) socket();
- create query
- sendto();
- recvfrom();
- process response
getpeername() tells you who is connected at the other end of the socket
getsockname() tells you who is connected at your end of the socket

```c
int getpeername(int sd, struct sockaddr *their_address, socklen_t *namelen);
int getsockname(int sd, struct sockaddr *my_address, socklen_t *namelen);
```

gethostname() - get host name (from /etc/hosts)

```c
int gethostname(char *name, size_t len);
```
#include <sys/socket.h> //socket, send, recv, bind, listen, 
//accept, getsockname, getpeername...
#include <netdb.h> //for hostent, gethostbyname()
#include <netinet/in.h> //definitions of protocols
#include <arpa/inet.h> //inet_ntoa, inet_aton etc
Libraries - Link Options

- Command line switches for gcc or g++
- gcc server.c -o server -lnsl -lsocket (for Solaris)
- gcc server.c -o server -lnsl -lsocket -lresolv
- gcc server.c -o server -lnsl -lsocket -lresolv -lxnet
- gcc server.c -o server (should do for Linux / MAC OS-X)
- gcc server.c (will result in a.out)
- chmod +x server (to make the output executable)
Socket Programming in Windows

```c
#include <winsock.h>
#include <netinet/in.h> // definitions of protocols
#include <arpa/inet.h> // inet_ntoa, inet_aton etc

int sd; ......

WORD wVersionRequested;
WSADATA wsaData;
wVersionRequested = MAKEWORD( 1, 1 );
WSAStartup(wVersionRequested, &wsaData) // returns 0
...... // on success

....
closesocket(sd); // for unix just close(sd);
WSACleanup();
}
```
Blocking vs Non Blocking sockets

- By default `accept()`, `recv()`, `recvfrom()` functions block
- For example `nb = recv(sd, buf, 256, 0)` would not return unless
  - 256 bytes have been received (`nb=256`), or
  - less than 256 bytes received (say `nb=100`), but the sender does not have anything to send anymore, or
  - sender closed connection (`nb=0`), or
  - an error occurs (`nb=-1`)
- Blocked sockets just wait till the transaction is “completed”
- Non-blocked sockets can return - for instance they can return with `nb=5`. It is up to the programmer to make sure that all bytes are received, by calling `recv` again to get the remaining bytes.
- Or call `recv` till 0 is received (or -1)
Setting a socket to non-blocking mode

```c
#include <fcntl.h>
...
sd = socket(SF_INET, SOCK_STREAM, 0);
fcntl(sd, F_SETFL, O_NONBLOCK);
...
```

If a socket is set to non blocking we have to periodically poll the socket to see if any bytes have been received / sent.

Why do we need non-blocking sockets?
Is there a better way?
select()

- int select(int, fd_set*, fd_set*, fd_set*, struct timeval*)
- maxfd - highest file descriptor - first input
- fd_set fdR, fdW, fdE (file descriptor sets - handles that have to be monitored for Reading, Writing and Exceptions)
- struct timeval - specifies timeout for select()
- FD_ZERO(), FD_SET(), FD_CLR(), FD_ISSET()
- Say we need to listen to a TCP socket and UDP socket.
- If we did not have select(), execution thread will wait at accept() for TCP socket and at recvfrom() for UDP socket
- select() needs to monitor two handles - TCP socket and UDP socket
- Assign both handles to fdR using FD_SET()
select()

```c
sdt = socket(. , SOCK_STREAM .) ; bind () ; listen ();
std = socket (. , SOCK_DGRAM , .) ; bind ();
int nfd = sdt > stdu ? sdt+1 : stdu+1;

for (; ; ) {
    fd_set fdR;
    FD_ZERO (&fdR);  // FD_ZERO (fd_set *)
    FD_SET (sdt, &fdR);  // FD_SET (int, fd_set *)
    FD_SET (std, &fdR);  // FD_CLR (int, fd_set *)
    select (nfd, &fdR, NULL, NULL, NULL);  // execution thread
    if (FD_ISSET (sdt, &fdR)) {}  // waits here
    if (FD_ISSET (std, &fdR)) {}  // waits here
}
```
- Windows sockets are used only for network programming
- `select()` can be used *only for sockets*