15-441: Computer Networking

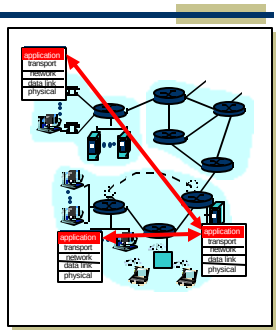
Lecture 3: Application Layer and Socket Programming

Lecture Overview

- Application layer
  - Client-server
  - Application requirements
- Background
  - TCP vs. UDP
  - Byte ordering
- Socket I/O
  - TCP/UDP server and client
  - I/O multiplexing

Applications and Application-Layer Protocols

- Application: communicating, distributed processes
  - Running in network hosts in "user space"
  - Exchange messages to implement app
  - e.g., email, file transfer, the Web
- Application-layer protocols
  - One "piece" of an app
  - Define messages exchanged by apps and actions taken
  - User services provided by lower layer protocols

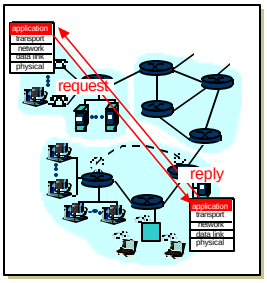


Client-Server Paradigm

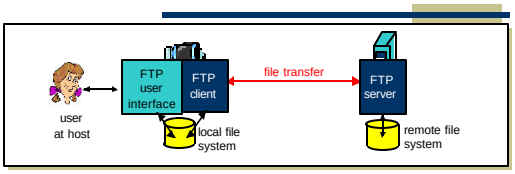
Typical network app has two pieces: *client* and *server*

Client:
- Initiates contact with server ("speaks first")
- Typically requests service from server,
- For Web, client is implemented in browser; for e-mail, in mail reader

Server:
- Provides requested service to client
- e.g., Web server sends requested Web page, mail server delivers e-mail



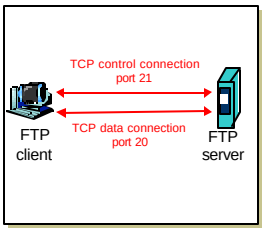
Ftp: The File Transfer Protocol



- Transfer file to/from remote host
- Client/server model
  - *Client:* side that initiates transfer (either to/from remote)
  - *Server:* remote host
- ftp: RFC 959
- ftp server: port 21

Ftp: Separate Control, Data Connections

- Ftp client contacts ftp server at port 21, specifying TCP as transport protocol
- Two parallel TCP connections opened:
  - Control: exchange commands, responses between client, server.
    "out of band control"
  - Data: file data to/from server
- Ftp server maintains "state": current directory, earlier authentication



Ftp Commands, Responses

Sample Commands:

- sent as ASCII text over control channel
- **`USER` *`username`***
- **`PASS` *`password`***
- **`LIST`** return list of files in current directory
- **`RETR filename`** retrieves (gets) file
- **`STOR filename`** stores (puts) file onto remote host

Sample Return Codes

- status code and phrase
- **`331 Username OK, password required`**
- **`125 data connection already open; transfer starting`**
- **`425 Can't open data connection`**
- **`452 Error writing file`**

What Transport Service Does an Application Need?

Data loss

- Some apps (e.g., audio) can tolerate some loss
- Other apps (e.g., file transfer, telnet) require 100% reliable data transfer

Timing

- Some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

Bandwidth

- Some apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"
- Other apps ("elastic apps") make use of whatever bandwidth they get

Transport Service Requirements of Common Apps

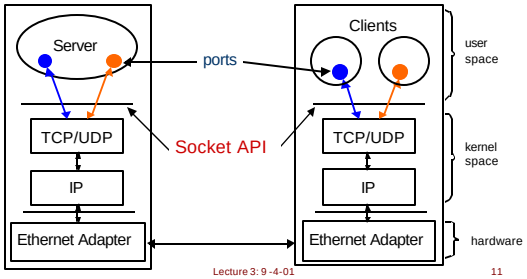
Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
web documents	no loss	elastic	no
real-time audio/ video	loss-tolerant	audio: 5Kb-1Mb video:10Kb-5Mb	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few Kbps	yes, 100's msec
financial apps	no loss	elastic	yes and no

Lecture Overview

- Application layer
  - Client-server
  - Application requirements
- Background
  - TCP vs. UDP
  - Byte ordering
- Socket I/O
  - TCP/UDP server and client
  - I/O multiplexing

Server and Client

Server and Client exchange messages over the network through a common Socket API



User Datagram Protocol(UDP): An Analogy

UDP	Postal Mail
• Single socket to receive messages • No guarantee of delivery • Not necessarily in-order delivery • Datagram – independent packets • Must address each packet	• Single mailbox to receive letters • Unreliable ☺ • Not necessarily in-order delivery • Letters sent independently • Must address each reply

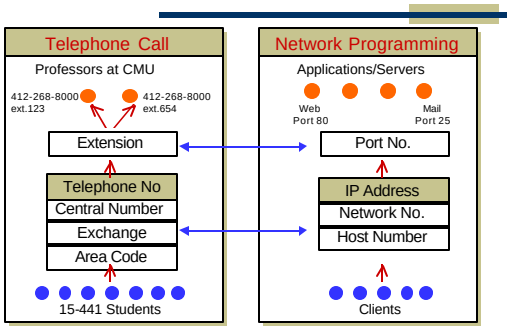
Example UDP applications
Multimedia, voice over IP

Transmission Control Protocol (TCP): An Analogy

TCP	Telephone Call
• Reliable – guarantee delivery • Byte stream – in-order delivery • Connection-oriented – single socket per connection • Setup connection followed by data transfer	• Guaranteed delivery • In-order delivery • Connection-oriented • Setup connection followed by conversation

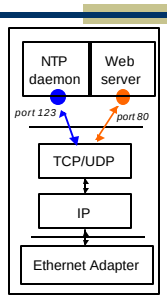
Example TCP applications
Web, Email, Telnet

Network Addressing Analogy



Concept of Port Numbers

- Port numbers are used to identify "entities" on a host
- Port numbers can be
  - Well-known (port 0-1023)
  - Dynamic or private (port 1024-65535)
- Servers/daemons usually use well-known ports
  - Any client can identify the server/service
  - HTTP = 80, FTP = 21, Telnet = 23, ...
  - /etc/service defines well-known ports
- Clients usually use dynamic ports
  - Assigned by the kernel at run time



Names and Addresses

- Each attachment point on Internet is given unique address
  - Based on location within network – like phone numbers
- Humans prefer to deal with names not addresses
  - DNS provides mapping of name to address
    - Whitepages of the Internet
  - Name based on administrative ownership of host

Internet Addressing Data Structure

```
#include <netinet/in.h>

/* Internet address structure */
struct in_addr {
        u_long s_addr;          /* 32-bit IPv4 address */
};                              /* network byte ordered */

/* Socket address, Internet style. */
struct sockaddr_in {
        u_char  sin_family;     /* Address Family */
        u_short sin_port;       /* UDP or TCP Port# */
                                /* network byte ordered */
        struct in_addr sin_addr; /* Internet Address */
        char    sin_zero[8];    /* unused */
};
```

- sin_family = AF_INET selects Internet address family

Byte Ordering

```
union {
   u_int32_t addr;  /* 4 bytes address */
   char c[4];
} un;
/* 128.2.194.95 */
un.addr = 0x8002c25f;
/* c[0] = ? */
```

	c[0]	c[1]	c[2]	c[3]
• Big Endian →	128	2	194	95
• Little Endian →	95	194	2	128

- Sun Solaris, PowerPC, ...
- i386, alpha, ...
- Network byte order = Big Endian

Byte Ordering Functions

- Converts between **host byte order** and **network byte order**
  - 'h' = host byte order
  - 'n' = network byte order
  - 'l' = long (4 bytes), converts IP addresses
  - 's' = short (2 bytes), converts port numbers

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int
hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int
netshort);
```


Lecture Overview

- Application layer
  - Client-server
  - Application requirements
- Background
  - TCP vs. UDP
  - Byte ordering
- Socket I/O
  - TCP/UDP server and client
  - I/O multiplexing

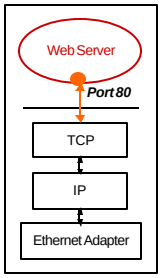
What is a Socket?

- A socket is a file descriptor that lets an application read/write data from/to the network

```
int fd;         /* socket descriptor */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) }
      perror("socket");
      exit(1);
}
```

- ***socket*** returns an integer (socket descriptor)
  - fd < 0 indicates that an error occurred
  - socket descriptors are similar to file descriptors
- AF_INET: associates a socket with the Internet protocol family
- SOCK_STREAM: selects the TCP protocol
- SOCK_DGRAM: selects the UDP protocol

TCP Server



- For example: web server
- **What does a *web server* need to do so that a *web client* can connect to it?**

Socket I/O: socket()

- Since web traffic uses TCP, the web server must create a socket of type SOCK_STREAM

```
int fd;         /* socket descriptor */

if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket");
      exit(1);
}
```

- ***socket*** returns an integer (**socket descriptor**)
  - ***fd*** < 0 indicates that an error occurred
- **AF_INET** associates a socket with the Internet protocol family
- **SOCK_STREAM selects the TCP protocol**

Socket I/O: bind()

- A ***socket*** can be bound to a ***port***

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;    /* used by bind() */

/* create the socket */

srv.sin_family = AF_INET; /* use the Internet addr family */

srv.sin_port = htons(80); /* bind socket 'fd' to port 80*/

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("bind"); exit(1);
}
```

- **Still not quite ready to communicate with a client...**

Socket I/O: listen()

- ***listen*** indicates that the server will accept a connection

```
int fd;                     /* socket descriptor */
struct sockaddr_in srv;     /* used by bind() */

/* 1) create the socket */
/* 2) bind the socket to a port */

if(listen(fd, 5) < 0) {
      perror("listen");
      exit(1);
}
```

- **Still not quite ready to communicate with a client...**

Socket I/O: accept()

- ***accept*** blocks waiting for a connection

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */
struct sockaddr_in cli;          /* used by accept() */
int newfd;                       /* returned by accept() */
int cli_len = sizeof(cli); /* used by accept() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
      perror("accept");   exit(1);
}
```

- ***accept*** returns a new socket (***newfd***) with the same properties as the original socket (***fd***)
  - ***newfd*** < 0 indicates that an error occurred

Socket I/O: accept() continued...

```
struct sockaddr_in cli;        /* used by accept() */
int newfd;                     /* returned by accept() */
int cli_len = sizeof(cli);     /* used by accept() */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
      perror("accept");
      exit(1);
}
```

- How does the server know which client it is?
  - **cli.sin_addr.s_addr** contains the client's ***IP address***
  - **cli.sin_port** contains the client's ***port number***
- Now the server can exchange data with the client by using ***read*** and ***write*** on the descriptor ***newfd***.
- Why does ***accept*** need to return a new descriptor?

Socket I/O: read()

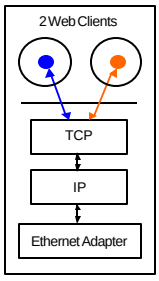
- ***read*** can be used with a socket
- ***read*** **blocks waiting for data from the client but does not guarantee that sizeof(buf) is read**

```
int fd;                          /* socket descriptor */
char buf[512];                   /* used by read() */
int nbytes;                      /* used by read() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept the incoming connection */

if((nbytes = read(newfd, buf, sizeof(buf))) < 0) {
      perror("read"); exit(1);
}
```


TCP Client

- For example: web client

- **How does a *web client* connect to a *web server*?**



Dealing with IP Addresses

- IP Addresses are commonly written as strings ("128.2.35.50"), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr("128.2.35.50");
if(srv.sin_addr.s_addr == (in_addr_t) -1) {
      fprintf(stderr, "inet_addr failed!\n"); exit(1);
}
```

Converting a numerical address to a string:

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0) {
      fprintf(stderr, "inet_ntoa failed!\n"); exit(1);
}
```

Translating Names to Addresses

- Gethostbyname provides interface to DNS
- Additional useful calls
  - Gethostbyaddr – returns `hostent` given sockaddr_in
  - Getservbyname
    - Used to get service description (typically port number)
    - Returns `servent` based on name

```
#include <netdb.h>

struct hostent *hp; /*ptr to host info for remote*/
struct sockaddr_in peeraddr;
char *name = "www.cs.cmu.edu";

peeraddr.sin_family = AF_INET;
hp = gethostbyname(name)
peeraddr.sin_addr.s_addr = ((struct in_addr*)(hp->h_addr))->s_addr;
```


Socket I/O: connect()

- ***connect*** allows a client to connect to a server...

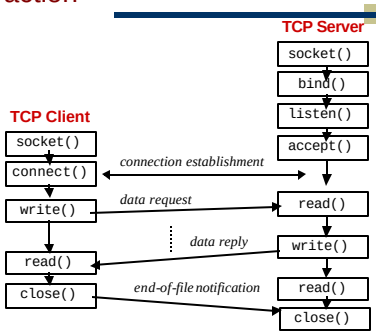
```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */

/* create the socket */

/* connect: use the Internet address family */
srv.sin_family = AF_INET;

/* connect: socket 'fd' to port 80 */
srv.sin_port = htons(80);

/* connect: connect to IP Address "128.2.35.50" */
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

if(connect(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("connect"); exit(1);
}
```


Socket I/O: write()

- ***write*** can be used with a socket

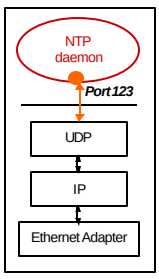
```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */
char buf[512];                   /* used by write() */
int nbytes;                      /* used by write() */

/* 1) create the socket */
/* 2) connect() to the server */

/* Example: A client could "write" a request to a server
*/
if((nbytes = write(fd, buf, sizeof(buf))) < 0) {
      perror("write");
      exit(1);
}
```


Review: TCP Client-Server Interaction



From UNIX Network Programming Volume 1, figure 4.1

UDP Server Example



- For example: NTP daemon
- **What does a *UDP server* need to do so that a *UDP client* can connect to it?**

Socket I/O: socket()

- The UDP server must create a **datagram** socket…

```
int fd;           /* socket descriptor */

if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror("socket");
      exit(1);
}
```

- ***socket*** returns an integer (**socket descriptor**)
  - ***fd*** < 0 indicates that an error occurred
- AF_INET: associates a socket with the Internet protocol family
- **SOCK_DGRAM: selects the UDP protocol**

Socket I/O: bind()

- A ***socket*** can be bound to a ***port***

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;    /* used by bind() */

/* create the socket */

/* bind: use the Internet address family */
srv.sin_family = AF_INET;

/* bind: socket 'fd' to port 80*/
srv.sin_port = htons(80);

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("bind"); exit(1);
}
```

- **Now the UDP server is ready to accept packets…**

Socket I/O: recvfrom()

- ***read*** does not provide the client's address to the UDP server

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */
struct sockaddr_in cli;          /* used by recvfrom() */
char buf[512];                   /* used by recvfrom() */
int cli_len = sizeof(cli);       /* used by recvfrom() */
int nbytes;                      /* used by recvfrom() */

/* 1) create the socket */
/* 2) bind to the socket */

nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
              (struct sockaddr*) &cli, &cli_len);
if(nbytes < 0) {
      perror("recvfrom"); exit(1);
}
```

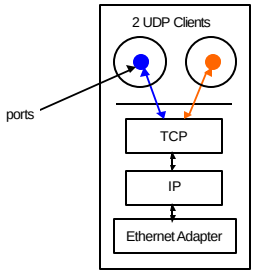

Socket I/O: recvfrom() continued...

```
nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
              (struct sockaddr*) cli, &cli_len);
```

- The actions performed by ***recvfrom***
  - returns the number of bytes read (***nbytes***)
  - copies ***nbytes*** of data into ***buf***
  - returns the address of the client (***cli***)
  - returns the length of ***cli*** (***cli_len***)
  - don't worry about flags

UDP Client Example

- **How does a *UDP client* communicate with a *UDP server*?**

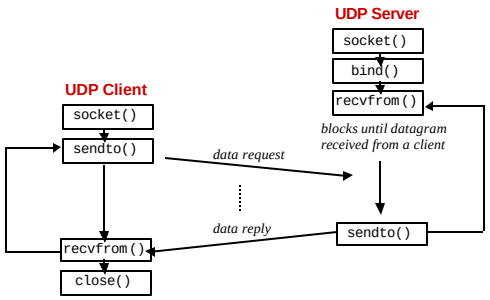


Socket I/O: sendto()

- ***write*** is not allowed
- Notice that the UDP client does not ***bind*** a port number
  - a port number is **dynamically assigned** when the first ***sendto*** is called

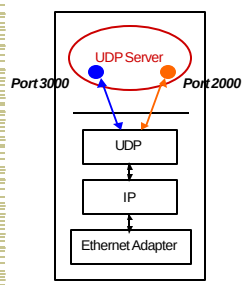
```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by sendto() */

/* 1) create the socket */

/* sendto: send data to IP Address "128.2.35.50" port 80 */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

nbytes = sendto(fd, buf, sizeof(buf), 0 /* flags */,
              (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
      perror("sendto");   exit(1);
}
```


Review: UDP Client-Server Interaction



from UNIX Network Programming Volume 1, figure 8.1

The UDP Server



- **How can the *UDP server* service multiple ports simultaneously?**

UDP Server: Servicing Two Ports

```
int s1;                         /* socket descriptor 1 */
int s2;                         /* socket descriptor 2 */

/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */

while(1) {
      recvfrom(s1, buf, sizeof(buf), ...);
      /* process buf */

      recvfrom(s2, buf, sizeof(buf), ...);
      /* process buf */
}
```

- **What problems does this code have?**

Socket I/O: select()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *fds);   /* clear the bit for fd in fds */
FD_ISSET(int fd, fd_set *fds); /* is the bit for fd in fds? */
FD_SET(int fd, fd_set *fds);   /* turn on the bit for fd in fds */
FD_ZERO(fd_set *fds);          /* clear all bits in fds */
```

- ***maxfds***: number of descriptors to be tested
  - descriptors (0, 1, ... maxfds-1) will be tested
- ***readfds***: a set of *fds* we want to check if data is available
  - returns a set of *fds* ready to read
  - if input argument is *NULL*, not interested in that condition
- ***writefds***: returns a set of *fds* ready to write
- ***exceptfds***: returns a set of *fds* with exception conditions

Socket I/O: select()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

struct timeval {
      long tv_sec;        /* seconds /
      long tv_usec;             /* microseconds */
}
```

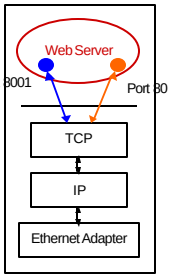
- ***timeout***
  - if NULL, wait forever and return only when one of the descriptors is ready for I/O
  - otherwise, wait up to a fixed amount of time specified by *timeout*
    - if we don't want to wait at all, create a timeout structure with timer value equal to 0
- Refer to the man page for more information

Socket I/O: select()

- ***select*** allows synchronous I/O multiplexing

```
int s1, s2;                     /* socket descriptors */
fd_set readfds;         /* used by select() */

/* create and bind s1 and s2 */
while(1) {
        FD_ZERO(&readfds);              /* initialize the fd set
*/
        FD_SET(s1, &readfds);   /* add s1 to the fd set */
        FD_SET(s2, &readfds);   /* add s2 to the fd set */

        if(select(s2+1, &readfds, 0, 0, 0) < 0) {
                perror("select");
                exit(1);
        }
        if(FD_ISSET(s1, &readfds)) {
                recvfrom(s1, buf, sizeof(buf), ...);
                /* process buf */
        }
        /* do the same for s2 */
}
```


More Details About a Web Server



- **How can a a *web server* manage multiple connections simultaneously?**

Socket I/O: select()

```
int fd, next=0;                          /* original socket */
int newfd[10];                    /* new socket descriptors */
while(1) {
       fd_set readfds;
       FD_ZERO(&readfds); FD_SET(fd, &readfds);

       /* Now use FD_SET to initialize other newfd's
          that have already been returned by accept() */

       select(maxfd+1, &readfds, 0, 0, 0);
       if(FD_ISSET(fd, &readfds)) {
              newfd[next++] = accept(fd, ...);
       }
       /* do the following for each descriptor newfd[n] */
       if(FD_ISSET(newfd[n], &readfds)) {
              read(newfd[n], buf, sizeof(buf));
              /* process data */
       }
}
```

- Now the web server can support multiple connections...

A Few Programming Notes: Representing Packets

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               Type                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |            Checksum           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Type: 4-byte integer
Length: 2-byte integer
Checksum: 2-byte integer
Address: 4-byte IP address
```


A Few Programming Notes: Building a Packet in a Buffer

```
struct packet {
      u_int32_t type;
      u_int16_t length;
      u_int16_t checksum;
      u_int32_t address;
};

/* ================================================ */
char buf[1024];
struct packet *pkt;

pkt = (struct packet*) buf;
pkt->type = htonl(1);
pkt->length = htons(2);
pkt->checksum = htons(3);
pkt->address = htonl(4);
```


Socket Programming References

- Man page
  - usage: man <function name>
- Textbook
  - Sections 2.6, 2.7
  - demo programs written in Java
- Unix Network Programming : Networking APIs: Sockets and XTI (Volume 1)
  - Section 2, 3, 4, 6, 8
  - ultimate socket programming bible!