Lecture 7

Berkeley Socket Programming

Berkeley Sockets

Unix Socket Programming FAQ

Beej's Guide to Network Programming

Metaphors

- Postal Service
  - Address
    - Name, Street, City, State, Zip Code
  - Return Address
  - Network of Post Offices
  - Local Post Offices
  - Lost Mail
  - Indeterminacy of Order
  - No confirmation of delivery for regular mail
  - No failed delivery notification

Metaphors (continued)

- Toll call from one hotel room to another (circa 1945), (or, London today)
  - call down to local hotel operator
  - tell her the area code and number to call of the remote hotel
  - hotel operator calls long distance operator, who is listening for incoming calls
  - long distance operator calls remote hotel
  - remote hotel operator picks up, as she has been listening for calls, and routes the call to your friend’s room
  - You and your friend are now talking directly

You Already Use Sockets

- echo (7), telnet (25), ftp (21), ssh (22)
  - telnet calcna.ab.ca echo
- cat /etc/services | grep [telnet | ssh | ftp | echo | etc.]
- daytime (13) (telnet time.mit.edu 13)
- email (SMTP) (port 25): (telnet direct to SMTP server)
  - telnet laime.cs.uchicago.edu 25
  - MAIL FROM: jcao@cs.uchicago.edu
  - RCPT TO: cspp51081@cs.uchicago.edu
  - DATA
  - [write something here, and end with a period on a line]
  - .
  - QUIT

The Fundamentals

- The Computer Systems Research Group (CSRG) at the University of California Berkeley gave birth to the Berkeley Socket API (along with its use of the TCP/IP protocol) with the 4.2BSD release in 1983.
  - A Socket is comprised of:
    - a 32-bit node address (IP address or FQDN)
    - a 16-bit port number (like 7, 21, 13242)
  - Example: 192.168.31.52:1051
    - The 192.168.31.52 host address is in “IPv4 dotted-quad” format, and is a decmial representation of the hex network address 0xc0a81f34

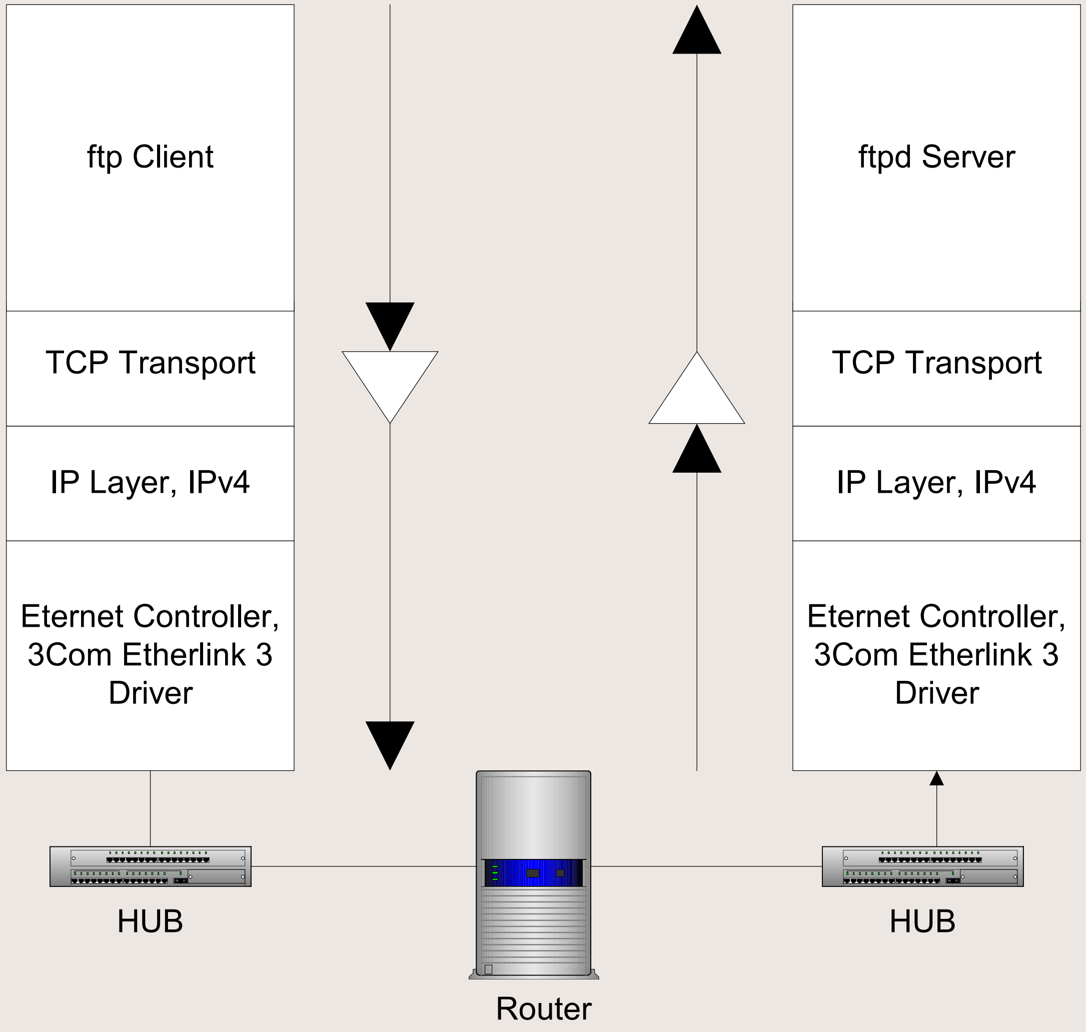
Port Assignments (less /etc/services)

- Ports 0 through 1023 are reserved, *priveledged* ports, defined by TCP and UDP well known port assignments
- Ports 1024 through 49151 are ports *registered* by the IANA (Internet Assigned Numbers Authority), and represent second tier common ports (socks (1080), WINS (1512), kermit (1649), https (443))
- Ports 49152 through 65535 are *ephemeral* ports, available for temporary client usage

Protocol Stacks

OSI Model (Tannenbaum, 1988)	Internet Protocol Suite
Application (Telnet, ftp, etc.)	Application (Telnet, ftp, etc.)
Presentation (MIDI, HTML, EBCDIC)	
Session (RPC, Netbios, Appletalk, DECnet)	
Transport (TCP, UDP)	Transport (TCP, UDP)
Network (IPv4, IPv6, IPX)	IP Layer (IPv4, IPv6)
Datalink (Ethernet, Token Ring, ATM, PPP)	Device Driver and Hardware (twisted pair, NIC)
Physical (V.24, 802.3, Ethernet RJ45)	

Protocol Communication



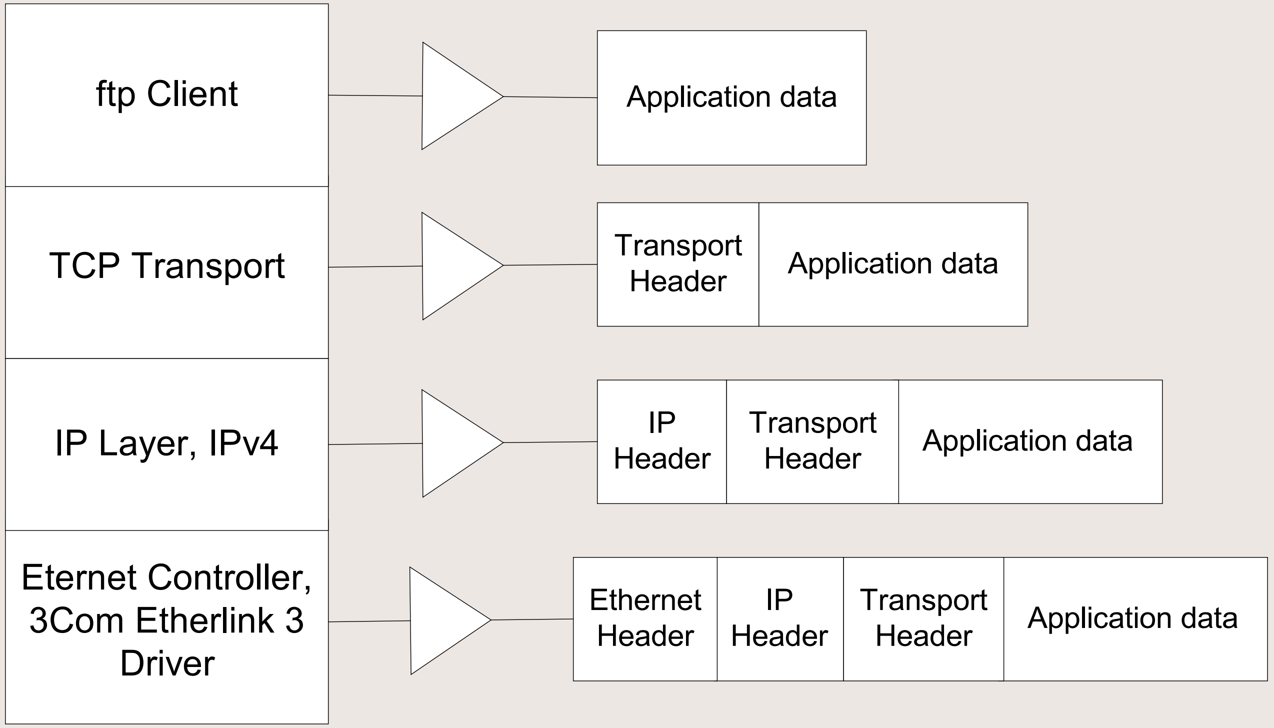
Common Protocols

Application	ICMP	UDP	TCP
Ping	✓		
Traceroute	✓		
DHCP		✓	
NTP		✓	
SNMP		✓	
SMTP			✓
Telnet			✓
FTP			✓
HTTP			✓
NNTP			✓
DNS		✓	✓
NFS		✓	✓
Sun RPC		✓	✓

ICMP: Internet Control Message Protocol
UDP: User Datagram Protocol
TCP: Transmission Control Protocol

Data Encapsulation

- Application puts data out through a socket
- Each successive layer wraps the received data with its own header:



The Hardware (Ethernet) Layer

- Responsible for transfering frames (units of data) between machines on the same physical network

Preamble (bit sequence)	Destination Address (192.32.65.1)	Source Address (192.32.63.5)	Packet type (magic number for protocol: 0x800 = IP, 0x6003 = Decnet, 0x809B = Appletalk)	Datagram (THE DATA) (up to 12k bits)	Cyclic Redundancy Check
64 bits	48 bits	48 bits	16 bits	variable	32 bits

The IP Layer

- The IP layer allows packets to be sent over gateways to machines not on the physical network
- Addresses used are IP addresses, 32-bit numbers divided into a network address (used for routing) and a host address
- The IP protocol is connectionless, implying:
  - gateways route discrete packets independently and irrespective of other packets
  - packets from one host to another may be routed differently (and may arrive at different times)
  - non-guaranteed delivery

IP Datagram Format

- Packets may be broken up, or *fragmented*, if original data is too large for a single packet (Maximum Transmission Unit is currently 12k bits, or 1500 Bytes)
- Packets have a Time To Live, number of seconds/rounds it can bounce around aimlessly among routers until it's killed

Preamble	Length of data	Fragmentation Information (if it's too big for an ethernet frame buffer)	Time To Live	Protocol (TCP, UDP)	Checksum	Source Address (192.32.63.5)	Destination Address (192.32.65.1)	Options	Datagram (THE DATA) (up to 12k bits)
2 bytes	2 bytes	4 bytes	1 byte	1 byte	2 bytes	4 bytes	4 bytes	variable	variable

The Transport Layer

- Unix has two common transports
  - User Datagram Protocol
    - record protocol
    - connectionless, broadcast
    - *Metaphor*: Postal Service
  - Transmission Control Protocol
    - byte stream protocol
    - direct connection-oriented
    - *Metaphor*: Phone Service circa 1945
      - "Sarah, this is Andy, get me Barney please."

The Transport Layer: UDP Protocol

- Connectionless, in that no long term connection exists between the client and server. A connection exists only long enough to deliver *a single packet* and then the connection is severed.
- No guaranteed delivery ("best effort")
- Fixed size boundaries, sent as a single "fire and forget message". Think *announcement*.
- No built-in acknowledgement of receipt

The Transport Layer: UDP Protocol

- No built-in order of delivery, random delivery
- Unreliable, since there is no acknowledgement of receipt, there is no way to know to resend a lost packet
- Does provide checksum to guarantee integrity of packet data
- *Fast* and Efficient

The Transport Layer: TCP Protocol

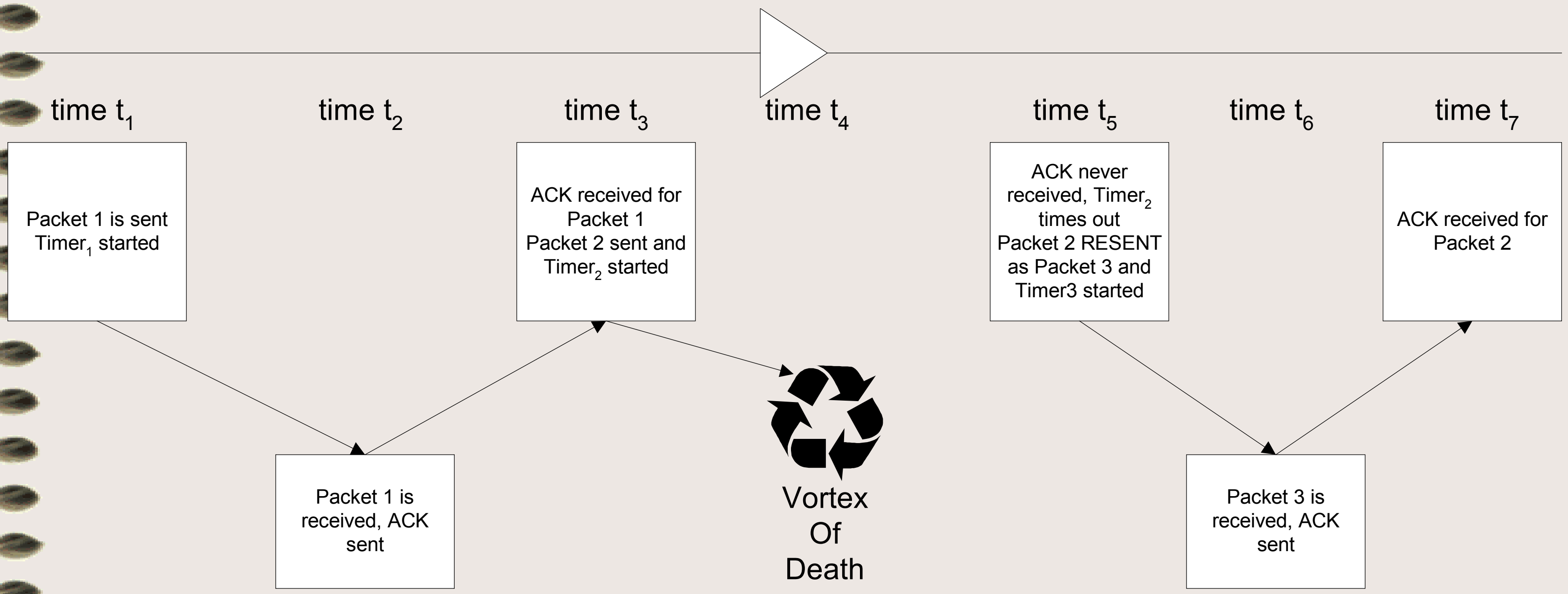
- TCP *guarantees delivery* of packets in order of transmission by offering acknowledgement and retransmission: it will automatically resend after a certain time if it does not receive an ACK
- TCP promises *sequenced delivery* to the application layer, by adding a sequence number to every packet. Packets are reordered by the receiving TCP layer before handing off to the application layer. This also aides in handling "duplicate" packets.

The Transport Layer: TCP Protocol

- Pure stream-oriented connection, it does not care about message boundaries
- A TCP connection is full duplex (bidirectional), so the same socket can be read and written to (cf. half duplex pipes)
- Provides a checksum that guarantees packet integrity

TCP's Positive Acknowledgement with Retransmission

- TCP offers acknowledgement and retransmission: it will automatically resend after a certain time if it does not receive an ACK
- TCP offers *flow control*, which uses a "sliding window" (in the TCP header) will allow a *limited* number of non-ACKs on the net during a given interval of time. This increases the overall bandwidth efficiency. This window is dynamically manged by the recipient TCP layer.


time t_1
time t_2
time t_3
time t_4
time t_5
time t_6
time t_7
Packet 1 is sent Timer$_1$ started
Packet 1 is received, ACK sent
ACK received for Packet 1 Packet 2 sent and Timer$_2$ started
Vortex Of Death
ACK never received, Timer$_2$ times out Packet 2 RESENT as Packet 3 and Timer3 started
Packet 3 is received, ACK sent
ACK received for Packet 2

How to Reuse Addresses

- Local ports are locked from rebinding for a period of time (usually a couple of minutes based on the TIME_WAIT state) after a process closes them. This is to ensure that a temporarily "lost" packet does not reappear, and then be delivered to a *reincarnation* of a listening server. But when coding and debugging a client server app, this is bothersome. The following code will turn this feature off:

```
int yes = 1;
server = socket(AF_INET, SOCK_STREAM, 0);

if (setsockopt(server, SOL_SOCKET,
    SO_REUSEADDR, &yes, sizeof(int)) < 0)
{
  perror("setsockopt SO_REUSEADDR");
  exit(1);
}
```

TCP Datagram Format

- Source and Destination addresses
- Sequence Number tells what byte offset within the overall data stream this segment applies
- Acknowledgement number lets the recipient set what packet in the sequence was received ok.

Source Port	Destination Port	Sequence Number	Acknowledgement Number	Flags	Window Size	Checksum	Urgent Pointer	Options	Datagram (THE DATA) (up to 12k bits)
2 bytes	2 bytes	4 bytes	4 bytes	2 bytes	2 bytes	2 bytes	2 bytes	variable	variable

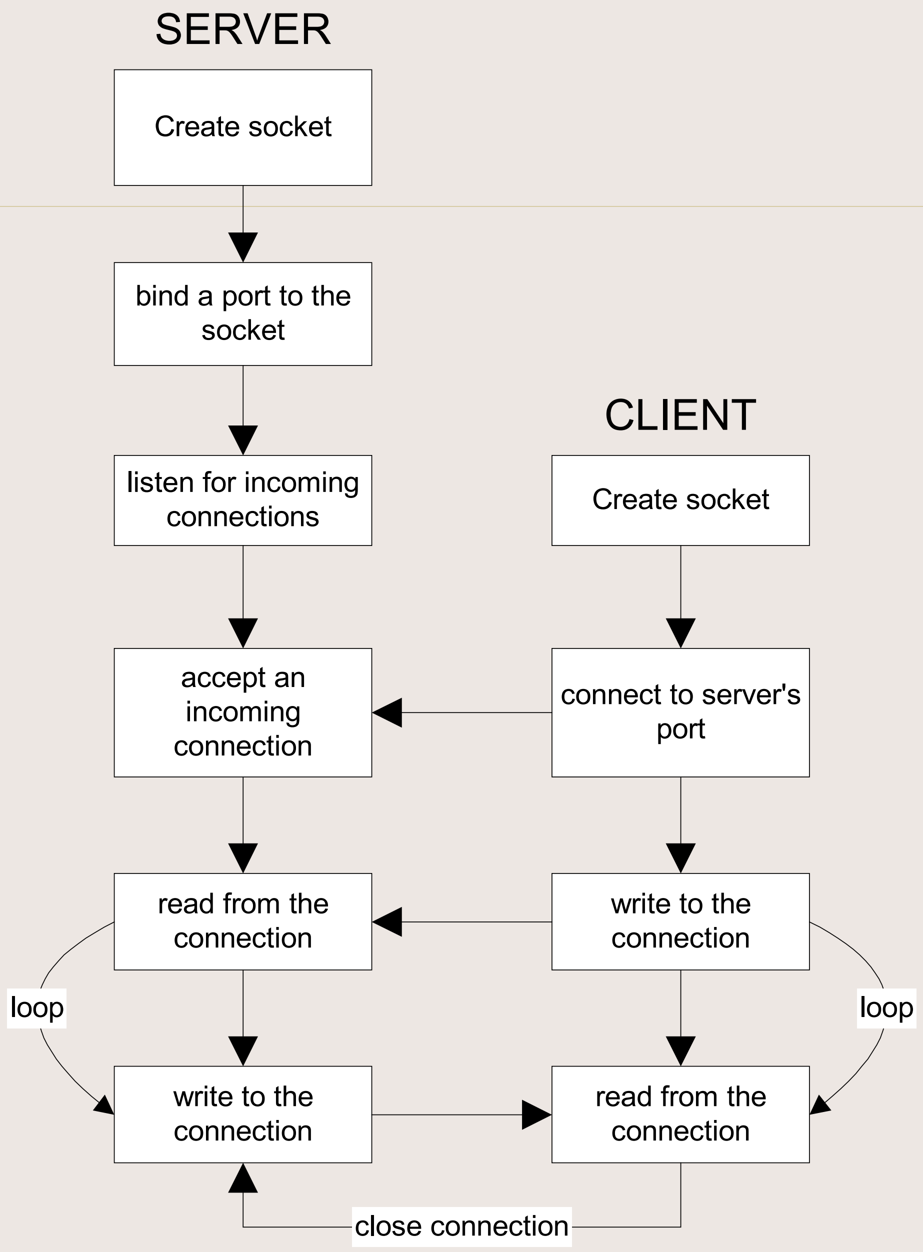
Socket Domain Families

- There are several significant socket domain families:
  - Internet Domain Sockets (AF_INET)
    - implemented via IP addresses and port numbers
  - Unix Domain Sockets (AF_UNIX)
    - implemented via filenames (think “named pipe”)
  - Novell IPX (AF_IPX)
  - AppleTalk DDS (AF_APPLETALK)
  - *Example*: ~mark/pub/51081/sockets/linux/socketpairs.c

Creating a Socket

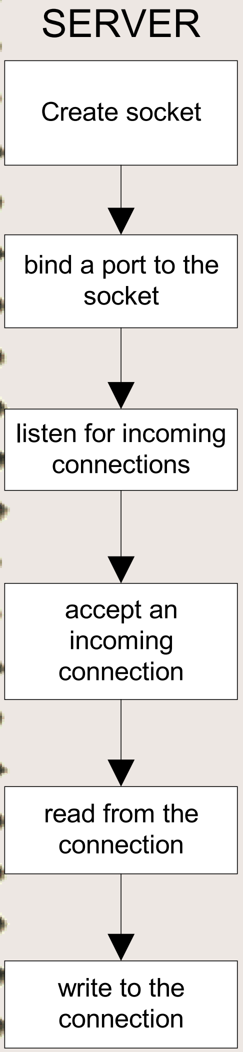
```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- domain is one of the *Address Families* (AF_INET, AF_UNIX, etc.)
- type defines the communication protocol semantics, usually defines either:
  - SOCK_STREAM: connection-oriented stream (TCP)
  - SOCK_DGRAM: connectionless, unreliable (UDP)
- protocol specifies a particular protocol, just set this to 0 to accept the default (PF_INET, PF_UNIX) based on the domain



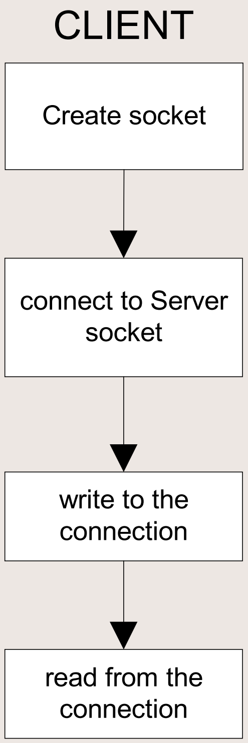
- Connection-oriented socket connections
- Client-Server view

Server Side Socket Details


SERVER
Create socket
bind a port to the socket
listen for incoming connections
accept an incoming connection
read from the connection
write to the connection

```
int socket(int domain, int type, int protocol)
sockfd = socket(PF_INET, SOCK_STREAM, 0);

int bind(int sockfd, struct sockaddr *server_addr, socklen_t length)
bind(sockfd, &server, sizeof(server));

int listen( int sockfd, int num_queued_requests)
listen( sockfd, 5);

int accept(int sockfd, struct sockaddr *incoming_address, socklen_t length)
newfd = accept(sockfd, &client, sizeof(client)); /* BLOCKS */

int read(int sockfd, void * buffer, size_t buffer_size)
read(newfd, buffer, sizeof(buffer));

int write(int sockfd, void * buffer, size_t buffer_size)
write(newfd, buffer, sizeof(buffer));
```

Client Side Socket Details



```
int socket(int domain, int type, int protocol)
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

```
int connect(int sockfd, struct sockaddr *server_address, socklen_t length)
connect(sockfd, &server, sizeof(server));
```

```
int write(int sockfd, void * buffer, size_t buffer_size)
write(sockfd, buffer, sizeof(buffer));
```

```
int read(int sockfd, void * buffer, size_t buffer_size)
read(sockfd, buffer, sizeof(buffer));
```

Setup for an Internet Domain Socket

```
struct sockaddr_in {
   sa_family_t sin_family;
   unsigned short int sin_port;
   struct in_addr sin_addr;
   unsigned char pad[...];
};
```

- sin_family is set to *Address Family* AF_INET
- sin_port is set to the port number you want to *bind* to
- sin_addr is set to the IP address of the machine you are binding to (struct in_addr is a wrapper struct for an unsigned long). INADDR_ANY supports all interfaces (since a given machine may have multiple interface cards)
- ignore padding

Setup for A Unix Domain Socket

```
struct sockaddr_un {
   sa_family_t sun_family;
   char sun_path[UNIX_PATH_MAX];
};
```

- sun_family is set to *Address Family* AF_UNIX
- sun_path is set to a UNIX *pathname* in the filesystem

Reading From and Writing To Stream Sockets

- Sockets, like everything else, are like *files*:
  - low level IO:
    - read() system call
    - write() system call
  - higher level IO:
    - int recv(int socket, char *buf, int len, int flags);
      - blocks on read
      - returns 0 when other connection has terminated
    - int send(int socket, char *buf, int len, int flags);
      - returns the number of bytes *actually sent*
    - where flags may be one of:
      - MSG_DONTROUTE (don’t route out of localnet)
      - MSG_OOB (out of band data (think *interruption*))
      - MSG_PEEK (examine, but don’t remove from stream)

Closing a Socket Session

- int close(int socket);
  - closes read/write IO, closes socket file descriptor
- int shutdown(int socketfd, int how);
  - where how is:
    - 0: no more receives allowed
    - 1: no more sends are allowed
    - 2: disables both receives and sends (but doesn't
      close the socket, use close() for that)
- *Example*: hangserver.c (hangman game)

Host and Network Byte Ordering

- Different computer architectures store numbers differently:
  - Little Endian architectures (like VAX, Intel) store the least significant byte first
    - This means that within a (2-byte) word, the *least* significant byte is stored *first*, that is, at the *lowest* byte address
  - Big Endian architectures (like Sun Sparc, Motorola 68000) store the most significant byte appearing first
    - This means that within a (2-byte) word, the *most* significant byte is stored *first*, that is, at the *lowest* byte address
- *examples: ~mark/pub/51081/byteorder/linux/endian.sh and ~mark/pub/51081/byteorder/solaris/endian.sh*

Why This Matters

- TCP/IP mandates that *big-endian* byte ordering be used for transmitting protocol information
- This means that *little-endian* machines will need to *convert* ip addresses and port numbers into *big-endian* form in order to communicate successfully
- Note that big-endian architectures don’t actually have to do anything, because they already meet the specification

What's To Be Done About It?

- Several functions are provided to allow you to easily convert between host and network byte ordering, and they are:
  - to translate 32-bit numbers (i.e. IP addresses):
    - unsigned long htonl(unsigned long hostlong);
    - unsigned long ntohl(unsigned long netlong);
  - to translate 16-bit numbers (i.e. Port numbers):
    - unsigned short htons(unsigned short hostshort);
    - unsigned short ntohs(unsigned short netshort);

UDP Clients and Servers

- Connectionless clients and servers create a socket using SOCK_DGRAM instead of SOCK_STREAM
- Connectionless servers do not call listen() or accept(), and *usually* do not call connect()
- Since connectionless communications lack a sustained connection, several methods are available that allow you to *specify a destination address with every call*:
  - sendto(sock, buffer, buflen, flags, to_addr, tolen);
  - recvfrom(sock, buffer, buflen, flags, from_addr, fromlen);
- *Examples: daytimeclient.c, mytalkserver.c, mytalkclient.c*

Servicing Multiple Clients

- Two main approaches:
  - forking with fork()
  - selecting with select()
- fork() approach forks a new process to handle each incoming client connection, essentially to act as a "miniserver" dedicated to each new client:
  - must worry about zombies created when parent loops back to accept() a new client (ignore SIG_CHILD signal)
  - inefficient
- A better approach would be to have a *single* process handle all incoming clients, without having to spawn separate child "server handlers". Enter select().

select()

int select(int numfiledescs, fd_set readfdsset, fd_set writefdsset, fd_set errorfdsset, struct timeval * timeout);

- The select() system call provides a way for a single server to wait until a set of network connections has data available for reading
- The advantage over fork() here is that no multiple processes are spawned
- The downside is that the single server must handle state management on its own for all its new clients

select() (continued)

- select() will return if *any* of the descriptors in readfdsset and writefdsset of file descriptors are ready for reading or writing, respectively, or, if any of the descriptors in errorfdsset are in an error condition
- The FD_SET(int fd, fd_set *set) function will add the file descriptor *fd* to the set *set*
- The FD_ISSET(int fd, fd_set *set) function will tell you if filedesc *fd* is in the modified set *set*
- select() returns the total number of descriptors in the modified sets
- If a client closes a socket whose file descriptor is in one of your watched sets, select() will return, and your next recv() will return 0, indicating the socket has been closed

Setting the timeval in select()

- If you set the timeout to 0, select() times out *immediately*
- If you set the timeout to NULL, select() will *never* time out, and will block indefinitely until a filedes is modified
- If you don't care about a particular file descriptor set, just set it to NULL in the call:

  ```
  select (max, &readfds, NULL, NULL, NULL);
  ```

  – Here we only care about reading, and we want to block indefinitely until we do have a file descriptor ready to be read
- *examples: multiserver.c, multiclient.c*

Miscellaneous Socket Functions

- int getpeername(int sockfd, struct sockaddr * addr, int *addrlen);
  - this tells you the *hostname* of the REMOTE connection
- int gethostname(char * hostname, size_t size);
  - this tells you the *hostname* of your LOCAL connection
- int inet_aton(const char * string_address, &(addr.sin_addr));
  - converts the const ip *string_address* ("192.168.3.1") into an acceptable numeric form
- addr.sin_addr = inet_addr("192.168.3.1");
  - does the same thing

More Miscellaneous Functions

- struct hostent *gethostbyname(const char *hostname);
  - Does a DNS lookup and returns a pointer to a hostent structure that contains the host name, aliases, address type (AF_INET, etc.), length, and an array of IP addresses for this host (hostent.h_addr_list[0] is usually the one)
    (cf. /etc/nsswitch.conf)

```
struct hostent {
 char *h_name; /*DNS host name*/
 char **h_aliases; /*alias list*/
 int h_addrtype; /* "AF_INET", etc*/
 int h_length; /* length of addr*/
 char **h_addr_list; /*list of IP adds*/
 };
```

And a Few More

struct servent * getservbyname(const char *servicename, const char *protocol)

struct servent * getservbyport(int port, const char *protocol)

- example:

  serventptr = getservbyname("daytime", "udp");

```
struct servent {
    char * s_name; /*official service name*/
    char **s_aliases; /* alias list */
    int s_port; /*port num*/
    char *s_proto; /* protocol: "tcp", "udp"*/
};
```