Socket Programming

By: Krishan Kumar Sethi

KIIT, Bhubaneshwar

Outline

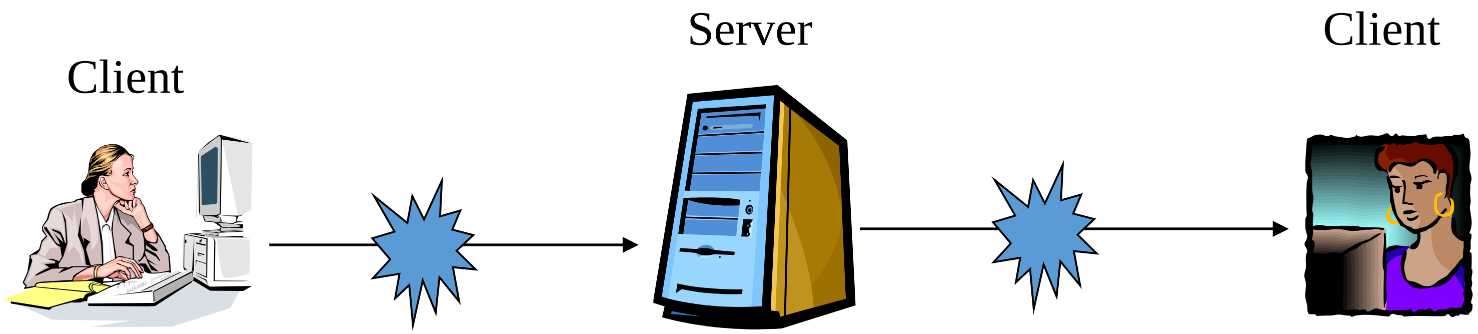
- Client Server Architecture
- Introduction to Socket
- IP address
- Ports
- Byte Ordering
- Connectionless service
- Connection oriented service
- Various library functions for socket programming

Client/Server Computing

- Some hosts (clients, typically desktop computers) are specialized to interact with users:
  - Gather input from users
  - Present information to users
- Other hosts (servers) are specialized to manage large data, process that data
- The Web is a good example: Client (Browser) & Server (HTTP server)

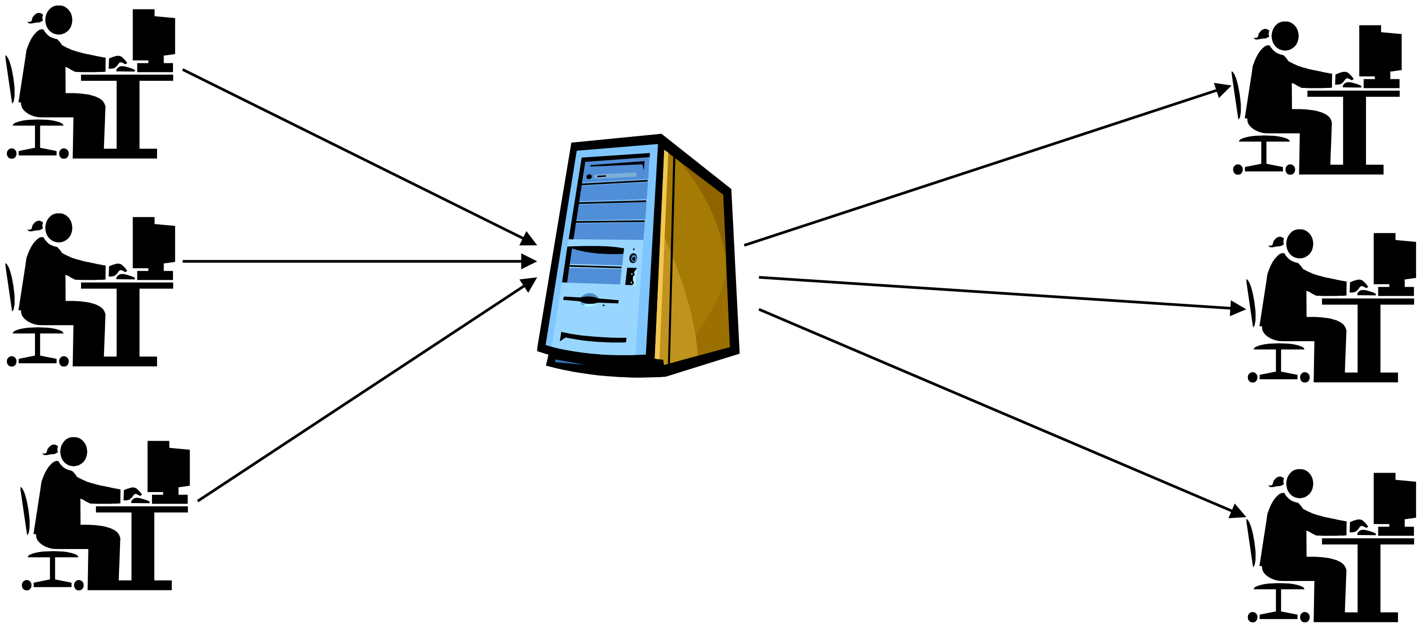
Client/Server Computing

- Other examples:
  - E-mail



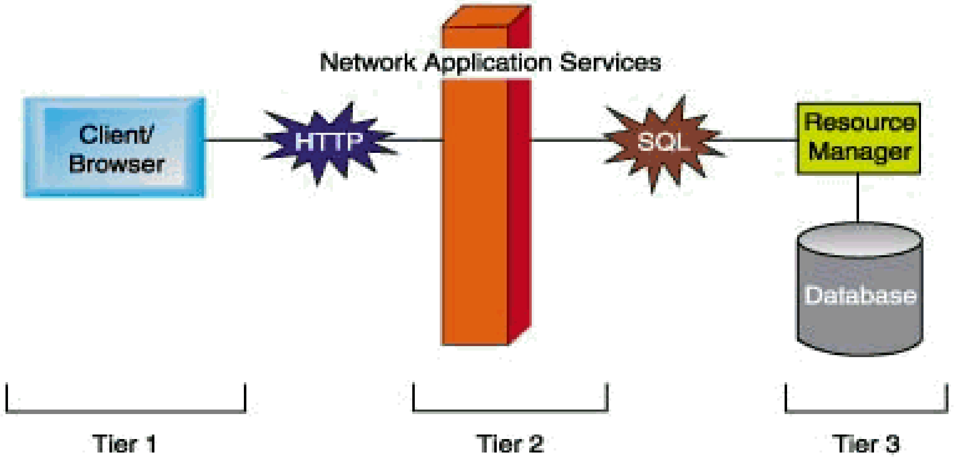
Client/Server Computing

- Other examples:
  - Chatroom



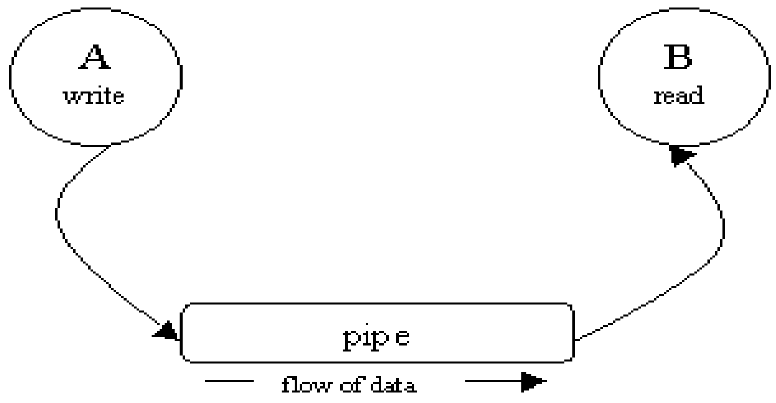
Tiered Client/Server Architecture

- 1-tier: single program
- 2-tier: client/server (e.g. the Web)
- 3-tier: application logic and databases on different servers (e.g. the Web with CGI and databases)



Client/Server Communication

- Two related processes on a single machine may communicate through a pipe



- A pipe is a pseudo-file that can be used to connect two processes together

Client/Server Communication

- Two UNRELATED processes may communicate through files (process A write to a file and process B reads from it)

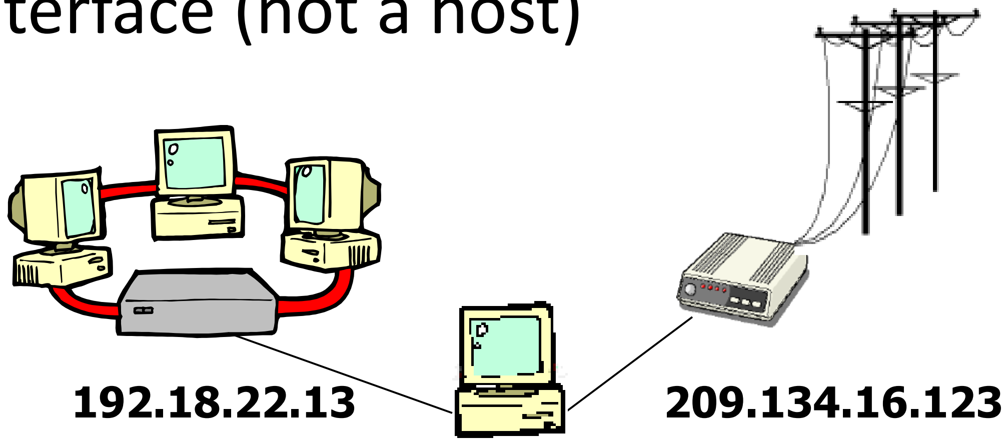
- But HOW two processes located on two different machines communicate? Solution: Berkeley sockets.

What are sockets

- A socket is an end point of a connection
- Or: the interface between user and network
- Two sockets must be connected before they can be used to transfer data (case of TCP)
- Message destinations are specified as socket addresses.
- Each socket address is a communication identifier:
  - Internet address
  - Port number
- The port number is an integer that is needed to distinguish between services running on the same machine
- A number of connections to choose from:
  - TCP, UDP
- Types of Sockets
  - SOCK_STREAM, SOCK_DGRAM

IP Address

- It is a logical address assigned to each machine to identify it uniquely in the computer network.
- 32-bit identifier
- Dotted-quad: 192.118.56.25
- www.google.com -> 167.208.101.28
- Identifies a host interface (not a host)



Ports

- it is used to identify a unique process/service on the server.
- 16 bits unsigned number. Total possible ports are (0- (2^{16} – 1)) = (0- 65535)
- Port numbers between 0 .. 1023 are reserved for well known applications.
- Some “well-known” ports:
  - 21: ftp
  - 23: telnet
  - 80: http
  - 161: snmp
- Check out /etc/services file for complete list of ports and services associated to those ports

Which transport protocol (TCP vs. UDP)

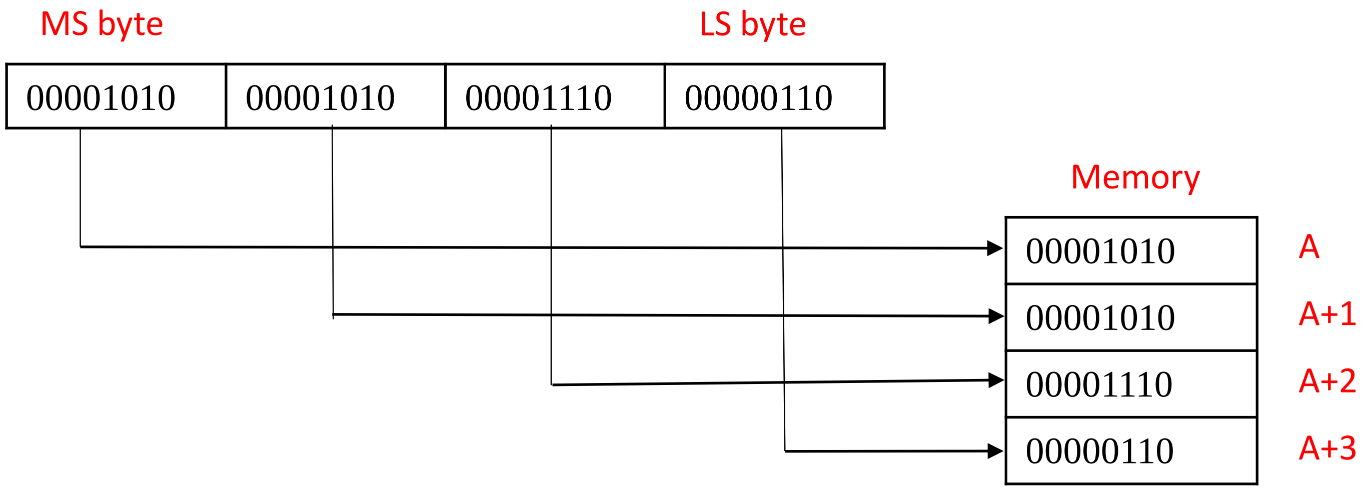
- TCP -- Transmission Control Protocol
- UDP -- User Datagram Protocol
- What should I use?
  - TCP is a ***reliable*** protocol, UDP is not
  - TCP is ***connection-oriented***, UDP is ***connectionless***
  - TCP incurs overheads, UDP incurs fewer overheads
  - UDP has a size limit of 64k, in TCP no limit

Byte Ordering/ Endianness

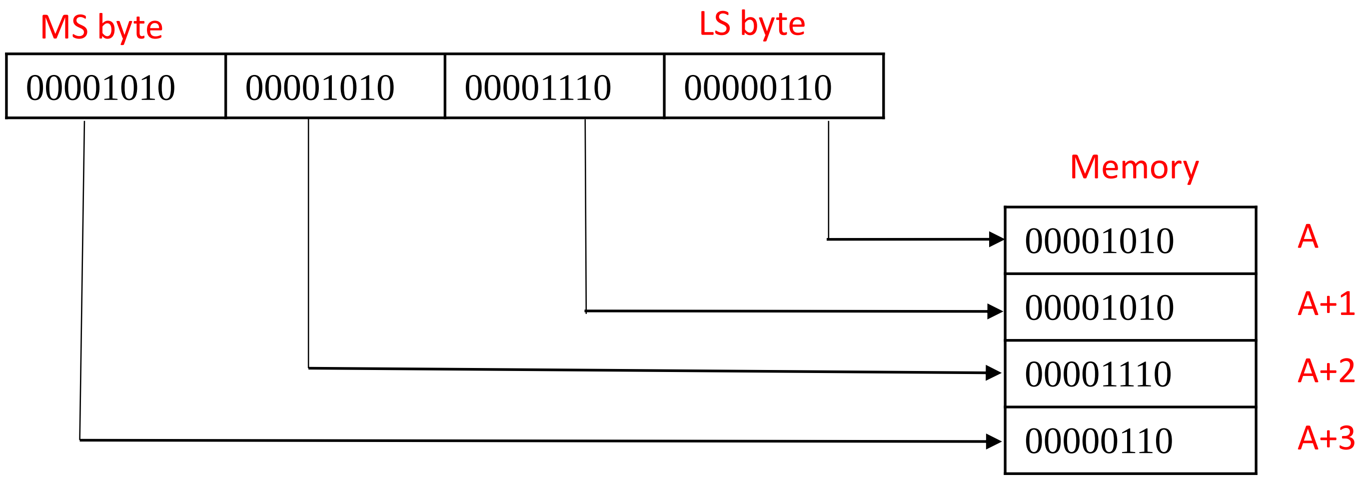
- In computing, endianness is the order or sequence of bytes of a word of digital data in computer memory.
- Types:
- Big-Endian (Network Byte Order): SUN, PPC MAC, Internet
  - High-order byte of the number is stored in memory at the lowest address.
  - Big End First
- Little-Endian: x86
  - Low-order byte of the number is stored in memory at the lowest address
  - Low End first
- Network stack (TCP/IP) expects Network Byte Order

Big Endian vs Little Endian

Big Endian:



Little Endian:



Example

- Variable X has 4 byte representation 0x01234567
- Address of X (&X) is given by 0x100
- Big Endian

			0x100	0x101	0x102	0x103			
			01	23	45	67			

- Little Endian

			0x100	0x101	0x102	0x103			
			67	45	23	01			

Conversions:

- htonl(): Translates an unsigned long integer into network byte order.
- htons(): Translates an unsigned short integer into network byte order.
- ntohl(): Translates an unsigned long integer into host byte order.
- ntohs(): Translates an unsigned short integer into host byte order.

Connectionless Service (UDP)

Server

1. Create transport endpoint: **socket()**
2. Assign transport endpoint an address: **bind()**
3. Wait for a packet to arrive: **recvfrom()**
4. Formulate reply (if any) and send: **sendto()**
5. Release transport endpoint: **close()**

Client

1. Create transport endpoint: **socket()**
2. Assign transport endpoint an address (optional): **bind()**
3. Determine address of server
4. Formulate message and send: **sendto()**
5. Wait for packet to arrive: **recvfrom()**
6. Release transport endpoint: **close()**

CONNECTION-ORIENTED SERVICE

Server

1. Create transport endpoint for incoming connection request: **socket()**
2. Assign transport endpoint an address: **bind( )**
3. Announce willing to accept connections: **listen( )**
4. Block and Wait for incoming request: **accept( )**
5. Wait for a packet to arrive: **recv ( )**
6. Formulate reply (if any) and send: **send( )**
7. Release transport endpoint: **close( )**

Client

1. Create transport endpoint: **socket( )**
2. Assign transport endpoint an address (optional): **bind( )**
3. Determine address of server
4. Connect to server: **connect( )**
4. Formulate message and send: **send ( )**
5. Wait for packet to arrive: **recv( )**
6. Release transport endpoint: **close( )**

socket()

- Create a socket, giving access to transport layer service.
- `Prototype: int socket(int family, int type, int protocol);`
- *family* is one of
  - AF_INET (IPv4), AF_INET6 (IPv6), AF_LOCAL (local Unix),
  - AF_ROUTE (access to routing tables), AF_KEY (new, for encryption)
- *type* is one of
  - SOCK_STREAM (TCP), SOCK_DGRAM (UDP)
- *protocol* is 0 (used for some raw socket options)
- upon success returns socket descriptor
  - Integer, like file descriptor
  - Return -1 if failure

bind()

- It is used to associate a socket with identifying address (IP + PORT).

- `Prototype: int bind(int sockfd, const struct sockaddr *myaddr, int addrlen);`

- *sockfd* is socket descriptor from `socket()`
- *myaddr* is a pointer to address struct that contains information about IP address and port.
  - if port is 0, then host will pick ephemeral port
- *addrlen* is length of the myaddr structure, i.e., sizeof(struct myaddr)
- returns 0 if ok, -1 on error
- Re-running a server may result in bind failure
  - Why? Socket still around in kernel using the port
  - Solution: Wait a minute or two or use function setsockopt() to clear the socket

Generic

- struct sockaddr

```
{         unsigned short sa_family;     /* Address family (e.g., AF_INET) */
          char sa_data[14];             /* Protocol-specific address information */
};
```

IP Specific

- struct sockaddr_in

```
{                   unsigned short sin_family;    /* Internet protocol (AF_INET) */
          unsigned short sin_port;      /* Port (16-bits) */
          struct in_addr sin_addr;      /* Internet address (32-bits) */
          char sin_zero[8];             /* Not used */
};
```

- sin_port and sin_addr must be in **network byte order.**

sockaddr	Family	Blob		
	2 bytes	2 bytes	4 bytes	8 bytes
sockaddr_in	Family	Port	Internet address	Not used

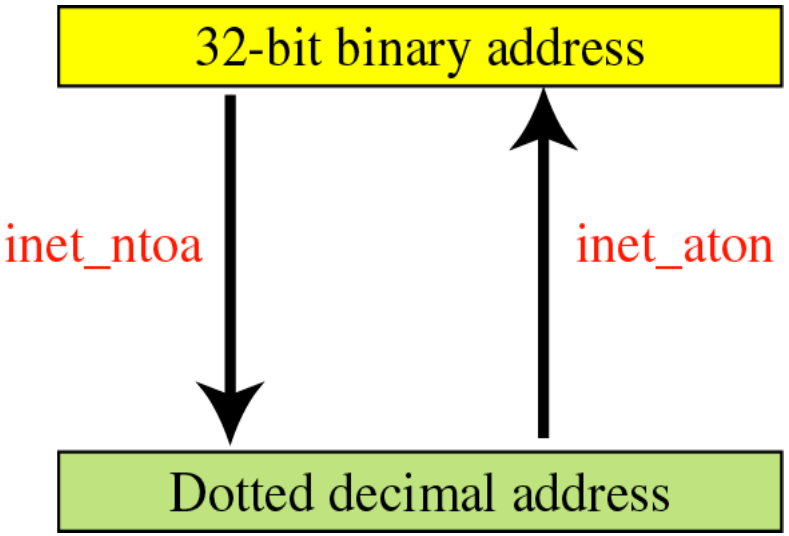
Address Transformation

int **inet_aton** (const char ***strptr*** , struct in_addr ***addrptr***) ;

char ***inet_ntoa** (struct in_addr ***inaddr***) ;

```
Example:
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);
inet_aton("10.0.0.5",&(my_addr.sin_addr));
memset(&(my_addr.sin_zero),'\0',8);

// To convert binary IP to string: inet_ntoa()
printf("%s", inet_ntoa(my_addr.sin_addr));
```



Byte-Manipulation Functions

- In network programming, we often need to initialize a field, copy the contents of one field to another, or compare the contents of two fields.
  - Cannot use string functions (strcpy, strcmp, …) which assume null character termination.

```
void *memset ( void *dest , int chr , int len ) ;

void *memcpy ( void *dest , const void *src , int len ) ;

int   memcmp ( const void *first , const void *second , int len ) ;
```

Common ways to IP address assign

1. my_addr.sin_addr.s_addr = INADDR_ANY; //use my IP adr (127.0.0.1)
2. my_addr.sin_addr.s_addr = inet_addr("10.0.0.1"); //assign a specific IP address

Example:

```
int sockfd;
struct sockaddr_in my_addr;
sockfd = socket(PF_INET, SOCK_STREAM, 0);
 my_addr.sin_family = AF_INET; // host byte order
my_addr.sin_port = htons(MYPORT); // short, network byte order
my_addr.sin_addr.s_addr = inet_addr("10.0.0.1");
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
/****** Code needs error checking. Don't forget to do that ****** /
```

sendto()

- Prototype: `int sendto(int sockfd, const void *msg, int len, int flags, const struct sockaddr *to, int tolen);`
  - sockfd: socket descriptor you want to send data to
  - msg is pointer to the data you want to send
  - len is the maximum length of the msg
  - *to* is a pointer to a struct sockaddr which contains the destination IP and port
  - *tolen* is sizeof(struct sockaddr)
  - Set flags to zero
  - Function returns the number of bytes actually sent or -1 on error

recvfrom()

- Prototype： `int recvfrom(int sockfd, void *buf, int len, int flags, struct sockaddr *from, int *fromlen);`
  - *sockfd: socket descriptor to read from*
  - *buf:* buffer to read the information from
  - *len:* maximum length of the buffer
  - *flags* set to zero
  - *from* is a pointer to a local struct sockaddr that will be filled with IP address and port of the originating machine
  - *fromlen* will contain length of address stored in *from*
  - *Returns the number of bytes received or -1 on error*

connect()

- Used by connection-oriented clients to connect to server
  - There must already be a socket bound to a connection-oriented service on the fd
  - There must already be a listening socket on the server
  - You pass in the address (IP address, and port number) of the server.
- Used by connectionless clients to specify a "default send to address"
  - Subsequent "writes" or "sends" don't have to specify a destination address
  - BUT, there really ISN'T any connection established… this is a bad choice of names!

- Prototype: `int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)`
- sockfd is the socket descriptor returned by socket()
- serv_addr is pointer to struct sockaddr that contains information on destination IP address and port
- addrlen is set to sizeof(struct sockaddr)
- client doesn't need `bind()`
  - OS will pick ephemeral port
- returns -1 on error

Example

```
#define DEST_IP "10.2.44.57"
#define DEST_PORT 5000
main(){
 int sockfd;
struct sockaddr_in dest_addr; // will hold the destination
addr sockfd = socket(PF_INET, SOCK_STREAM, 0);
dest_addr.sin_family = AF_INET; // host byte order
dest_ addr.sin_port = htons(DEST_PORT); // network byte order
dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
memset(&(dest_addr.sin_zero), '\0', 8); // zero the rest of the struct
connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
/******* Don't forget error checking ********/
```

listen()

- Change socket state for TCP server.

  Prototype: `int listen(int sockfd, int backlog);`
- sockfd is the socket file descriptor returned by socket()
- backlog is the number of connections allowed on the incoming queue
- listen() returns -1 on error
- Need to call bind() before you can listen()

accept()

- accept() gets the pending connection on the port you are listen()ing on.
- prototype: `int accept(int sockfd, struct sockaddr cliaddr, socklen_t *addrlen);`
- sockfd is the listening socket descriptor
- information about incoming connection is stored in addr which is a pointer to a local struct sockaddr_in
- addrlen is set to sizeof(struct sockaddr_in)
- accept() returns a new socket file descriptor to use for this accepted connection and -1 on error
- returns brand new descriptor, created by OS.
- Note: if create new process or thread, can create concurrent server

Example

```
 #include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
 #define MYPORT 3490 // the port users will be connecting to
 #define BACKLOG 10 // pending connections queue will hold
main(){
int sockfd, new_fd; // listen on sock_fd, new connection on new_fd
struct sockaddr_in my_addr; // my address information
struct sockaddr_in their_addr; // connector's address information
int sin_size; sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

Example (*cont..*)

```
my_addr.sin_family = AF_INET; // host byte order
my_addr.sin_port = htons(MYPORT); // short, network byte order
my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
// don't forget your error checking for these calls:
bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
listen(sockfd, BACKLOG);
sin_size = sizeof(struct sockaddr_in);
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
```

send() and recv()

- The two functions are for communicating over stream sockets or connected datagram sockets.

  Prototype: int send(int sockfd, const void *msg, int len, int flags);

- sockfd is the socket descriptor you want to send data to (got from accept())
- msg is a pointer to the data you want to send
- len is the length of that data in bytes
- set flags to 0 for now
- sent() returns the number of bytes actually sent (may be less than the number you told it to send) or -1 on error

- Example:

```
char *msg = “hello!”;
int len, bytes_sent;
………
len = strlen(msg);
bytes_sent = send(sockfd, msg, len 0);
```

send() and recv()

- Prototype: int recv(int sockfd, void *buf, int len, int flags);
  - sockfd is the socket descriptor to read from
  - buf is the buffer to read the information into
  - len is the maximum length of the buffer
  - set flags to 0 for now
  - recv() returns the number of bytes actually read into the buffer or -1 on error
  - If recv() returns 0, the remote side has closed connection on you

close()

Prototype: `int close(int sockfd);`

- Closes connection corresponding to the socket descriptor and frees the socket descriptor
- Will prevent any more sends and recieves

How to write + compile + run socket programs (Unix)

- Socket programs can be executed either in unix system using GCC compiler or windows using winsock library.
- Write a separate client side program and server side program in any editor program (gedit, vi, nano etc..).
- Compile: gcc client.c -o client (on client side)

  gcc server.c -o server (on server side)

- Run: ./client (on client side)

  ./server (on server side)