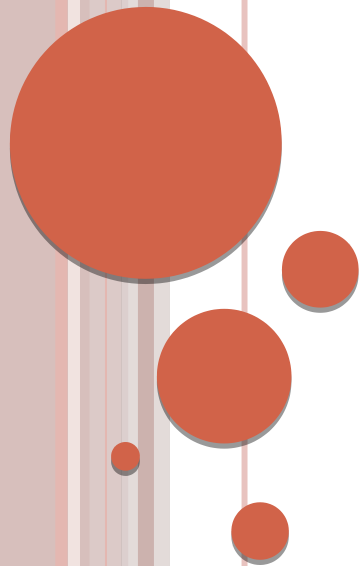


# **INTRODUCTION TO SOCKET PROGRAMMING**



# CLIENT SERVER MODEL

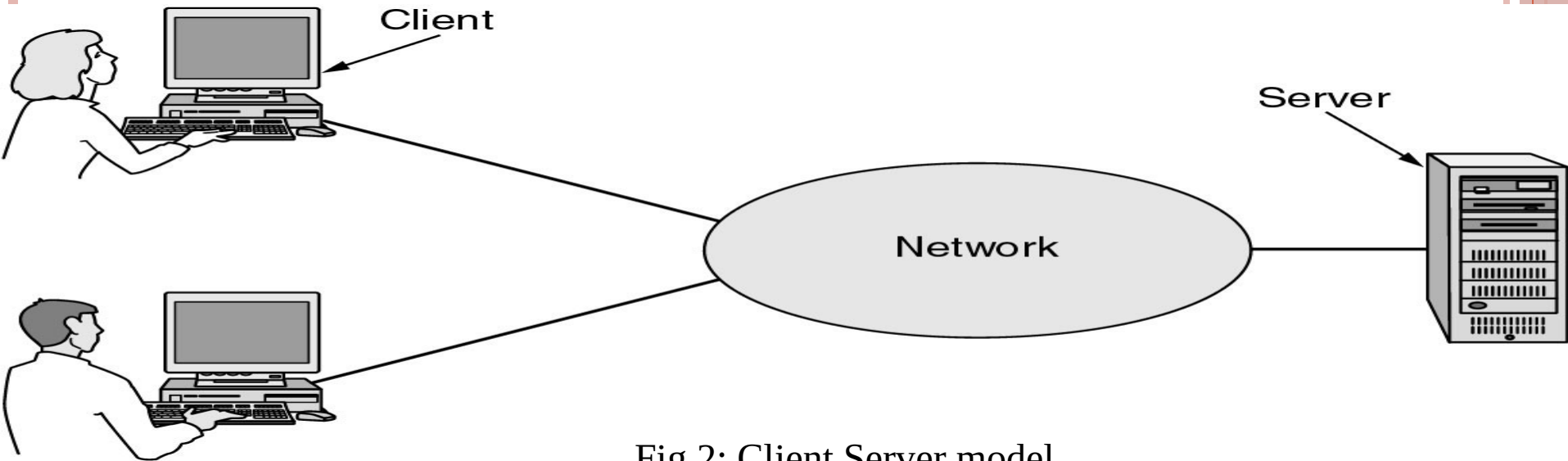


Fig 2: Client Server model

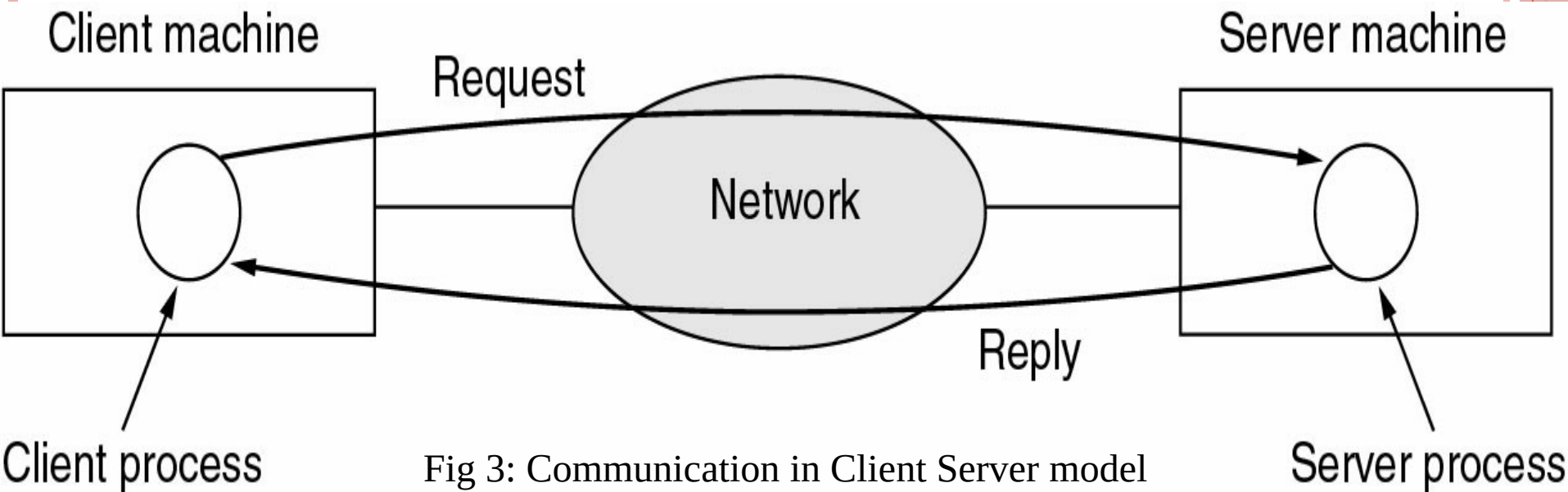


Fig 3: Communication in Client Server model

# WHAT IS COMPUTER NETWORKS?

- Computer network is a communication network in which a collection of computers are connected together to facilitate data exchange
- The connection between the computers can be wired or wireless.
- A computer network basically comprises of 5 components:
  - Sender
  - Receiver
  - Message
  - Transmission medium
  - Protocols

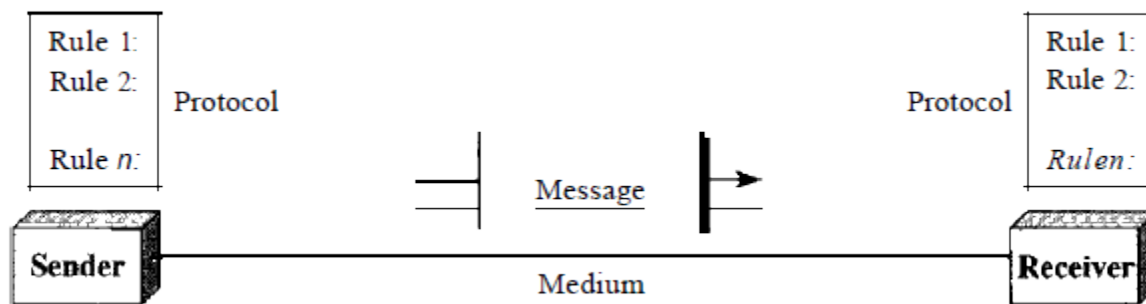


Fig 1: Components of data communication

# AN ANALOGY

## In the world of sockets.....


- Socket() – Endpoint for communication
- Bind() - Assign a unique telephone number.
- Listen() – Wait for a caller.
- Connect() - Dial a number.
- Accept() – Receive a call.
- Send(), Recv() – Talk.
- Close() – Hang up.



# SOCKET PROGRAMMING IN UNIX

What is “socket”?

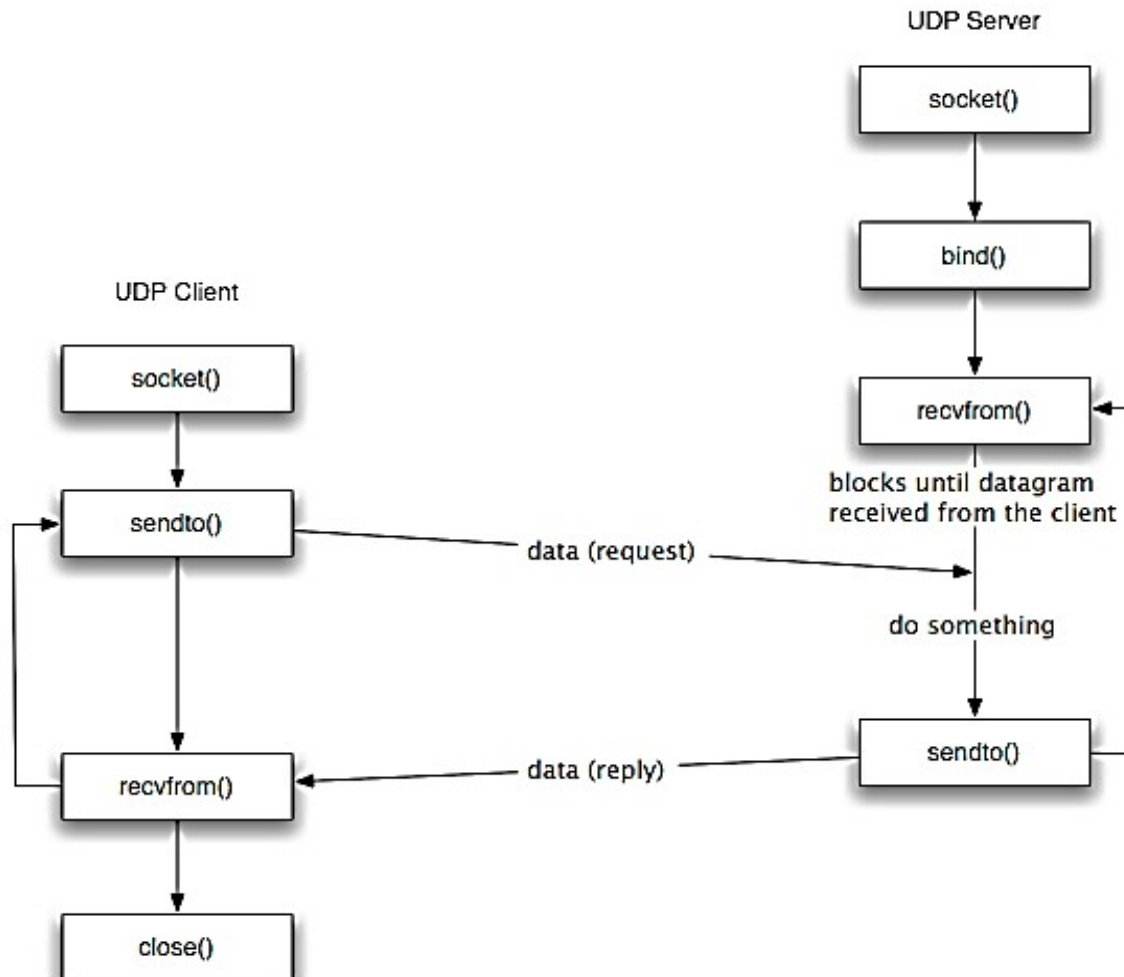
Socket Address = IP Address  
+ Port Number

- A socket is a **MECHANISM TO PROVIDE SERVICES** to the application program.
  - Using a socket, **two processes** can communicate with each other
  - A socket is **bi-directional** (full-duplex) transmission
  - A socket can be created **dynamically.(created any time)**
  - In operating systems, a socket is a structure(does not have methods)
  - A socket address is the combination of an IP address and a port number
- 

# SOCKET SYSTEM CALLS FOR CONNECTION-ORIENTED PROTOCOL



# SOCKET SYSTEM CALLS FOR CONNECTION-LESS PROTOCOL



- The server begins by carrying out a passive open as follows. (Server when it starts, opens the door for incoming requests but never initiates a service until it is requested to do so. This is called passive open.)
- The socket call creates a TCP socket.
- The bind call then binds the well-known port number of the server to the socket.
- The listen call turns the socket into a listening socket that can accept incoming connections from clients.
- The accept call puts the server process to sleep until the arrival of a client connection.





The left side of the slide features a series of vertical stripes in various shades of brown and tan. Overlaid on these stripes are several circles of different sizes, all in a reddish-brown color. The circles are arranged in a cluster, with the largest one at the top left and several smaller ones below and to its right.

# **ELEMENTARY SOCKET SYSTEM CALLS**

- The client does an active open(A client opens the communications channel using the IP address of the remote host and the well-known port address of the specific server program running on that machine).
- The socket call creates a socket on the client side, and the connect call establishes the TCP connection to the server with the specified destination socket address.
- The accept function at the server wakes up and returns the descriptor for the given connection namely, the source IP address, source port number, destination IP address, and destination port number.
- The client and server are now ready to exchange information.



Socket facilities are provided to programmers through system calls except that control is transferred to the OS kernel once the call is invoked.

To use these facilities, the header files `<sys/types.h>` and `<sys/socket.h>`

Before an application program can transfer any data it must first create an endpoint for communication by calling `socket`



# **“SOCKET”** SYSTEM CALL

## *DEFINITION AND PARAMETERS*

- To do network I/O, the first thing a process must do is create a socket by calling the socket system call, specifying the type of communication protocol desired (TCP, UDP etc.) as follows

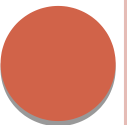
***int socket(int family, int type, int protocol);***

- This system call returns an integer which is (similar to file descriptor) a reference to the socket called the socket descriptor. On error, -1 is returned.



# BERKELEY SOCKETS

- **socklen\_t** and **size\_t** are same as **int**
- **u\_long** and **uint32\_t** are same as **long int**
- **u\_short** and **uint16\_t** are same as **short**



- `SOCK_RAW` – provides access to internal network interfaces and is available only to super-user.
- `SOCK_SEQPACKET` – Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
<code>SOCK_STREAM</code>	TCP SCTP	TCP SCTP	Yes		
<code>SOCK_DGRAM</code>	UDP	UDP	Yes		
<code>SOCK_SEQPACKET</code>	SCTP	SCTP	Yes		
<code>SOCK_RAW</code>	IPv4	IPv6		Yes	Yes



- ✂ Since we are interested in internet protocols the family parameter is set to AF\_INET.
- ✂ The type field takes the values SOCK\_STREAM which means a stream socket is created (used in the case of TCP) and SOCK\_DGRAM which means a datagram socket is created (used in the case of UDP).
- ✂ The protocol argument to the socket system call is typically set to 0 to indicate default protocol.
- ✂ The default protocol for SOCK\_STREAM with AF\_INET family is TCP.

### **Example**

```
if ( (sd = socket( AF_INET, SOCK_STREAM, 0 )) < 0 ) {  
    cout<<"Socket creation error";  
    exit(-1);  
}
```



## Protocol *family* constants for `socket` function.

<i>family</i>	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

## *type* of socket for `socket` function.

<i>type</i>	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

## *protocol* of sockets for `AF_INET` or `AF_INET6`.

<i>Protocol</i>	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol



# “**BIND**” SYSTEM CALL

## *WHY USE BIND?*

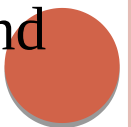
- Bind helps the **servers** to register themselves with the system. It tells the system that “any messages received at the particular IP address and the specified port be directed to me”.
- This is required in case of both connection-oriented (like TCP) and connectionless (like UDP) servers.
- A **connectionless client** needs to assure that the system assigns it some unique address, so that the other end (the server) has a valid return address to send its responses to. This is not required in the case of connection-oriented client.



# **“BIND”** SYSTEM CALL

## *DEFINITION AND PARAMETERS*

- This system call assigns a name to an unnamed socket –  
*int bind(int sockfd, struct sockaddr \*myaddr, socklen\_t addrlen);*
- The sockfd argument is nothing but the socket descriptor which is obtained after executing the socket system call.
- The second argument is a pointer to a protocol-specific address and the third argument is the size of the address structure passed as reference in the second argument.
- Since we are interested in Internet Protocols the second argument passed is of type sockaddr\_in.



# **“BIND”** SYSTEM CALL

## *DEFINITION AND PARAMETERS*

- This system call assigns a name to an unnamed socket –  
*int bind(int sockfd, struct sockaddr \*myaddr, socklen\_t addrlen);*
- The sockfd argument is nothing but the socket descriptor which is obtained after executing the socket system call.
- The second argument is a pointer to a protocol-specific address and the third argument is the size of the address structure passed as reference in the second argument.
- Since we are interested in Internet Protocols the second argument passed is of type sockaddr\_in.



# **“BIND”** SYSTEM CALL

## *THE SOCKADDR\_IN STRUCTURE*

- The structure sockaddr\_in (defined in <netinet/in.h>) is as follows –

```
struct in_addr {  
    u_long s_addr;  
};  
struct sockaddr_in {  
    short sin_family;  
    u_short sin_port;  
    struct in_addr sin_addr;  
    char sin_zero[8];  
};
```

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

*/\* unused \*/*



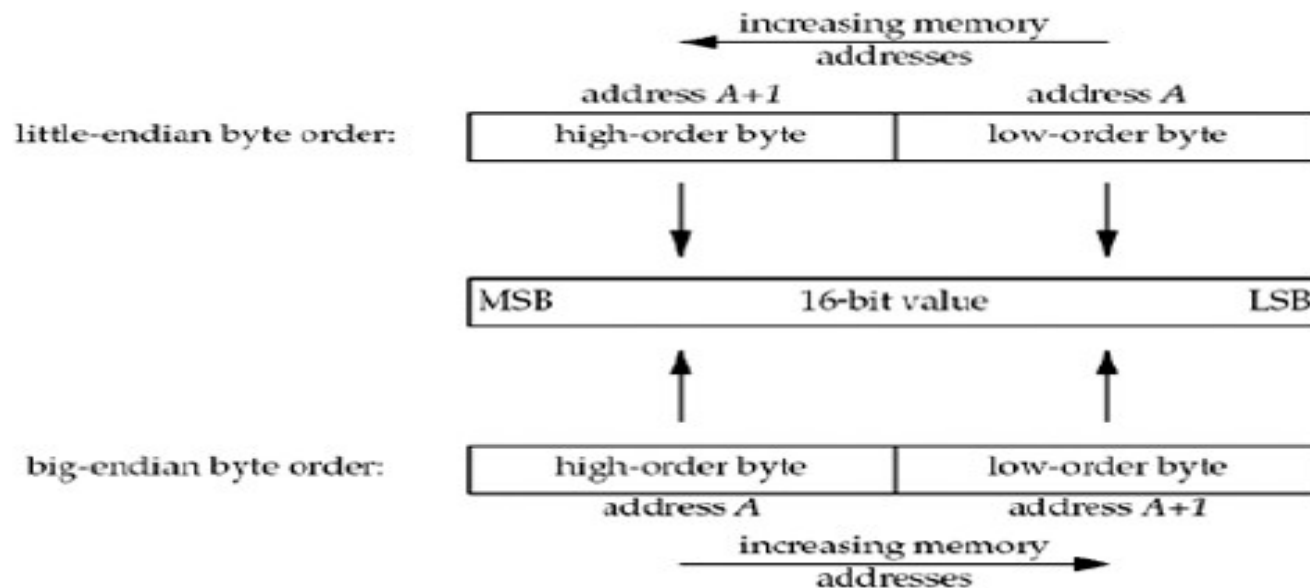
## **BIND SYSTEM CALL**

# *MORE ON SOCKADDR\_IN STRUCTURE...*

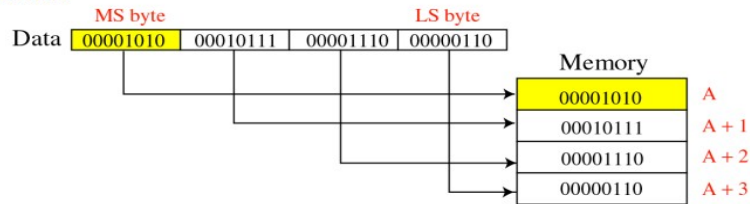
- The *sin\_family* field takes the value **AF\_INET** indicating that an internet family protocol is being used.
- The *sin\_port* field identifies the server/client application to the host where it is running.
- The *sin\_addr* structure has only one field *s\_addr* which identifies the server side IP address (32-bits) to which various clients can connect.
- The *sin\_port* and *sin\_addr* fields are network byte ordered i.e., a format that is permissible for a given network protocol.



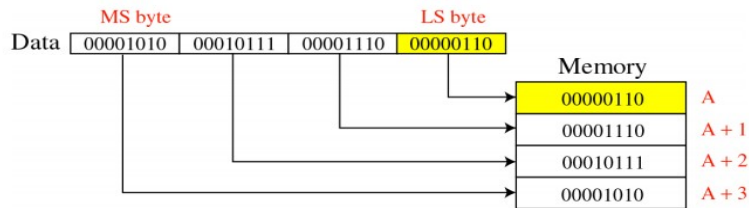
- Consider a 16-bit integer that is made up of 2 bytes. There are two ways to store the two bytes in memory: with the low-order byte at the starting address, known as *little-endian* byte order, or with the high-order byte at the starting address, known as *big-endian* byte order.
- So that machines with different byte order conventions can communicate, the Internet protocols specify a canonical byte order convention for data transmitted over the network. This is known as *network byte order* or big endian byte order.



■ Big-Endian:



■ Little-Endian:



BIG-ENDIAN BYTE ORDER IS  
ALSO REFERRED TO  
AS **NETWORK BYTE ORDER**

LITTLE-ENDIAN BYTE  
ORDER IS ALSO  
REFERRED TO  
AS **HOST BYTE ORDER**



# “**BIND**” SYSTEM CALL

## *BYTE ORDERING ROUTINES*

- The various byte ordering routines that provide the necessary conversion from host byte ordering to network byte ordering and vice versa are as follows –
  - uint32\_t htonl(uint32\_t hostlong);*
  - uint16\_t htons(uint16\_t hostshort);*
  - uint32\_t ntohl(uint32\_t netlong);*
  - uint16\_t ntohs(uint16\_t netshort);*
- **htonl** and **ntohl** convert from host-to-network and network-to-host byte order respectively and return a long integer value representing the converted ordering.





# **“BIND”** SYSTEM CALL

## *MORE ON BYTE ORDERING ROUTINES...*

- **htons** and **ntohs** convert from host-to-network and network-to-host byte order respectively and return a short integer value representing the converted ordering.
- **htonl** and **ntohl** can be used while converting network IP addresses which are passed to *sin\_addr.s\_addr* field of the **sockaddr\_in** structure.
- **htons** and **ntohs** can be used while converting the 16-bit port number that is passed to the *sin\_port* field of the **sockaddr\_in** structure.



# “**BIND**” SYSTEM CALL

## *ADDRESS CONVERSION ROUTINES*

- An internet address is usually written in dotted-decimal notation i.e., in the form of A.B.C.D. Thus it is necessary that this address be converted into proper format before passing the value to the *sin\_addr.s\_addr* field of the **sockaddr\_in** structure.
- The following functions provide the necessary address conversion –

*in\_addr\_t inet\_addr(const char \*cp);*

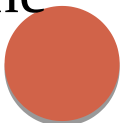
*char \*inet\_ntoa(struct in\_addr in);*

- The **inet\_addr** function converts the dotted-decimal address string into binary data in network byte order.

## “**BIND**” SYSTEM CALL

### *MORE ON ADDRESS CONVERSION ROUTINES...*

- If the input provided to **inet\_addr** is invalid an error occurs, which makes the function returns an address value where all the 32 bits are set to 1, which means 255.255.255.255 is returned if an error occurs.
- Thus for the **inet\_addr** function 255.255.255.255 is considered to be an error, but nevertheless it is a valid network (reserved) address. Thus the **inet\_addr** function fails in this regard. (A solution to this discussed shortly).
- The **inet\_ntoa** function does the reverse conversion i.e., it extracts the 32-bit internet address and converts it into the usual dotted-decimal notation in host byte order.



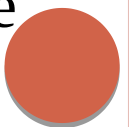
## “**BIND**” SYSTEM CALL

### *MORE ON ADDRESS CONVERSION ROUTINES...*

- The **inet\_addr** function is an obsolete interface to **inet\_aton** which provides a clearer way to indicate errors. The function prototype is as follows –

***int inet\_aton(const char \*cp, struct in\_addr \*inp);***

- **inet\_aton** converts the internet host address cp from the usual dotted-decimal notation into binary data and stores the converted address in the structure pointed to by inp.
- **inet\_aton** returns a non-zero value if the address provided is valid or else zero is returned, thus solving the problem posed by **inet\_addr**. Thus it is advised that **inet\_aton** be used instead of **inet\_addr**.



# “**BIND**” SYSTEM CALL


## *SOME MORE INFORMATION...*

- Since the second argument to the **bind** system call is of type pointer to **sockaddr** structure, but the network specific address structure **sockaddr\_in** is used, this (**sockaddr\_in**) structure has to be type-casted before passing its address to **bind**. Even though no errors are raised (a warning will be given though) the call fails.
- On success, 0 is returned. On error, -1 is returned by the **bind** system call.



# “**LISTEN**” SYSTEM CALL

## *DEFINITION AND PARAMETERS*

- This system call is used by a connection-oriented server (i.e., **SOCK\_STREAM** socket) to indicate that it is willing to receive connections from clients.
  - The function prototype is as follows –  
***int listen(int sockfd, int backlog);***
  - The ***sockfd*** field is nothing but the socket descriptor obtained after executing the **socket** system call.
  - The ***backlog*** parameter defines the maximum length of the queue of pending connections may grow to. This argument is usually specified as 5 which the maximum value currently allowed.
- 

# “ACCEPT” SYSTEM CALL

## *DEFINITION AND PARAMETERS*

- The **accept** system call extracts the first connection request on the queue of pending connections and creates a new connected socket with mostly the same properties as that of **sockfd**. This newly created socket's descriptor is returned and only this descriptor must be used for further communication with the intended client. The original socket descriptor **sockfd** remains unaffected by this call.
- The function prototype is as follows –

```
int accept(int sockfd, struct sockaddr *peer,  
socklen_t *addrlen);
```



## **“ACCEPT”** SYSTEM CALL *SOCKFD, PEER AND ADDRLLEN PARAMETERS...*

- The *sockfd* parameter is the socket descriptor that was created with **socket**, bound to a local address with **bind** and is listening for connections after **listen**.
- The argument *peer* is a pointer to a **sockaddr** structure. This structure will be filled in with the address of the peer entity (connecting entity or client). Since the **sockaddr\_in** structure is used (in the case of internet family protocols) the structure passed as the second argument to **accept** must be adequately type-casted.
- The *addrlen* is a value-result parameter whose value is initially set to contain the size of the structure pointed to by peer.



## “ACCEPT” SYSTEM CALL *PARAMETERS AND MORE...*

- On return the value in the **addrlen** parameter is updated so as to contain the actual length (in bytes) of the address returned.
- **accept** system call blocks the caller until a connection from a client is established.
- As mentioned earlier **accept** returns a non-negative integer that represents the descriptor for the accepted socket while **listen** returns 0 on success.
- Both **listen** and **accept** system calls return -1 if any error occurs.




# “CONNECT” SYSTEM CALL

## *DEFINITION AND PARAMETERS*

- A client process can establish a connection with a server using the **connect** system call as follows –

***int connect(int sockfd, const struct sockaddr \*servaddr, socklen\_t addrlen);***

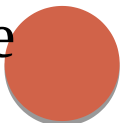
- The ***sockfd*** parameter is the socket descriptor that is returned by the **socket** system call (at the client side).
  - The ***servaddr*** parameter is the address of the server (IP address and port number of the server) to which the client wishes to connect.
  - ***addrlen*** is the length of the address structure pointed to by ***servaddr***.
- 

## “**CONNECT**” SYSTEM CALL *CONNECTION-ORIENTED VS. CONNECTIONLESS...*

- For connection-oriented protocols the **connect** system call results in the actual establishment of a connection between the client and the server.
- Specific parameters such as buffer size, amount of data to exchange between acknowledgements etc. might be agreed upon in the process of establishing the connection.
- A connectionless client can also use the **connect** system call. But here an actual establishment of connection between the server and client does not take place.
- In the case of a connectionless client, the *servaddr* is stored by the process in the local system so that all future data is directed to the address specified by *servaddr*.



## “**CONNECT**” SYSTEM CALL *CONNECTION-ORIENTED VS. CONNECTIONLESS...*

- And the client socket will receive only datagrams from the address specified by ***servaddr*** (in the case of a connectionless client).
  - Thus if the connectionless client wishes to receive datagrams from another host then the only thing that needs to be done is change the address in the ***servaddr*** address structure and use **connect** again.
  - The advantage of using **connect** with a connectionless client is that every time a datagram needs to be sent, the destination address need not be specified again and again (specially if there is large amount of data to be exchanged).
- 

## “CONNECT” SYSTEM CALL

### *SOME MORE INFORMATION...*

- If the connection or binding succeeds, 0 is returned. On error, -1 is returned.
- Note that all the system calls discussed so far under **Berkeley Sockets** can be used by including the following headers –

*<sys/types.h>*

*<sys/socket.h>*

- To use the system calls **send**, **sendto**, **recv** and **recvfrom** (discussed next) the above mentioned header files must be included in the socket program.



# “**SEND**” SYSTEM CALL

## *DEFINITION AND PARAMETERS*

- A connection-oriented client/server can use the **send** system call to exchange messages between each other. The function prototype is as follows –

***int send(int sockfd, const void \*msg, size\_t len, int flags);***

- **sockfd** is the socket descriptor returned by the **socket** system call in the case of the client and the new socket descriptor returned by the **accept** system call in the case of the server.
- **msg** is pointer to the message that has to be read or written and **len** is the number of bytes has to be read or written .
- The **flags** parameter is usually set to 0.



# “**RECV**” SYSTEM CALL

## *DEFINITION AND PARAMETERS*

- To receive messages from a connected socket a client/server can use the **recv** system call as follows –  
***int recv(int sockfd, void \*buf, size\_t len, int flags);***
- **sockfd** and **flags** have the same meaning as in the **send** system call.
- Here **buf** is nothing but the message that will be received by the client/server.
- The **len** parameter specifies the number of characters (in bytes) that can be read at once.
- The **recv** system call blocks until a message is received.



# **“CLOSE”** SYSTEM CALL *DEFINITION AND PARAMETERS*

- The Unix **close** system call is also used to close a socket.
- The function prototype is as follows –  
***int close(int fd);***
- Here ***fd*** refers to the socket descriptor.
- The **close** system call does not close the socket at once. Before closing it tries to send any queued data or any queued data to be sent is flushed.
- **The close** system call returns 0 on success and -1 on error.





# SENDTO()--UDP SOCKETS

- **int sendto(int socket, char \*buffer, int length, int flags, struct sockaddr \*destination\_address, int address\_size);**
- *For example:*

```
struct sockaddr_in sin;  
sin.sin_family = AF_INET;  
sin.sin_port = htons(12345);  
sin.sin_addr.s_addr = inet_addr("128.227.22.43");  
char *msg = "Hello, World";  
sendto(s, msg, strlen(msg)+1, 0, (struct sockaddr *)sin,  
      sizeof(sin));
```



# RECVFROM()--UDP SOCKETS

- **Int recvfrom(int socket, char \*buffer, int length, int flags, struct sockaddr \*sender\_address, int \*address\_size)**
- *For example:*

```
struct sockaddr_in sin;  
char msg[10000];  
int ret;  
int sin_length;  
sin_length = sizeof(sin);  
ret = recvfrom(s, msg, 10000, 0, (struct sockaddr *)sin,  
    &sin_length);  
printf("%d bytes received from %s (port %d)\n", ret,  
    inet_ntoa(sin.sin_addr), sin.sin_port);
```



## A Simple TCP server program

```
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
```

```
#define MAXSIZE 50
```



```
main()
{
    int sockfd,newsockfd,retval;
    socklen_t actualen;
    int recedbytes,sentbytes;
    struct sockaddr_in serveraddr,clientaddr;
    char buff[MAXSIZE];
    int a=0;

    sockfd=socket(AF_INET,SOCK_STREAM,0);
    if(sockfd==-1) {
        cout<<"\nSocket creation error";
        exit(-1); }
```



```
newsockfd=accept(sockfd,(struct sockaddr*)&clientaddr,&actua  
llen);  
if(newsockfd==-1) {  
    close(sockfd);  
    exit(0); }
```

```
recedbytes=recv(newsockfd,buff,sizeof(buff),0);  
if(recedbytes==-1) {  
    close(sockfd);  
    close(newsockfd);  
    exit(0); }
```

```
puts(buff);  
cout<<"\n";
```



```
serveraddr.sin_family=AF_INET;  
serveraddr.sin_port=htons(3387);  
serveraddr.sin_addr.s_addr=htonl(INADDR_ANY);
```

*/\*With IPv4, the *wildcard address* is specified by the constant INADDR\_ANY, whose value is normally 0. This tells the kernel to choose the IP address.\*/*

```
retval=bind(sockfd,(struct sockaddr*)&serveraddr,sizeof(serveraddr));  
if(retval==-1) {  
    cout<<"Binding error"; close(sockfd);  
    exit(0); }
```

```
retval=listen(sockfd,1);  
if(retval==-1) { close(sockfd);  
    exit(0); }  
actuellen=sizeof(clientaddr);
```



```
gets(buff);  
sentbytes=send(newsockfd,buff,sizeof(buff),0);  
if(sentbytes==-1) {  
    close(sockfd);  
    close(newsockfd);  
    exit(0);  
}
```

```
close(newsockfd);  
close(sockfd);  
}
```



Header	Functions defined
arpa/inet.h	inet functions: inet_aton, inet_ntoa, and inet_addr
unistd.h	System call: close, fork, exec
sys/socket.h	basic socket definitions recv, send
sys/types.h	basic system data types
netinet/in.h	sockaddr_in{} and other Internet definitions
sys/stat.h	for S_xxx file mode constants
fcntl.h	for nonblocking





Process specifies		Result
IP address	port	
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

