# DECCAN COLLEGE OF ENGINEERING & TECHNOLOGY

Dar-us-Salam, Hyderabad -500 001.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**CNP LAB**

B.E. VI Semester (CBCS)

**2018-2019**

# Lab Manual for the Academic Year 2018-2019

|  |  |  |
|---|---|---|
| **SUBJECT** | **:** | **CNP LAB** |
| **SUBJECT CODE** | **:** | **PC653CS** |
| **CLASS** | **:** | **B.E VI Semester (CBCS)** |
| **STREAM** | **:** | **COMPUTER SCIENCE AND ENGINEERING** |
| **INSTRUCTOR** | **:** | **Mr. C. ATHEEQ** |

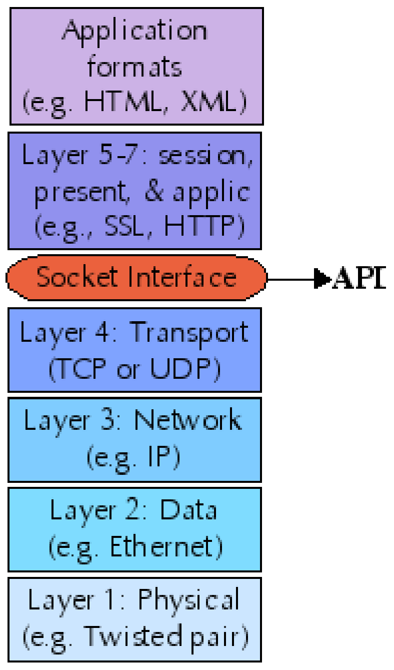
# INTRODUCTION

The Windows Sockets Specification has been built upon the Berkeley Sockets programming model which is the de facto standard for TCP/IP networking. It is intended to provide a high degree of familiarity for programmers who are used to program with sockets in UNIX and other environments, and to simplify the task of porting existing sockets-based source code. The Windows Sockets API is consistent with release 4.3 of the Berkeley Software Distribution (4.3BSD).

**Network**

A computer network is composed of a number of "network layers", each providing a different restriction and/or guarantee about the data at that layer. The protocols at each network layer generally have their own packet formats, and headers.

The seven traditional layers of a network are divided into two groups: upper layers and lower layers. The sockets interface provides a uniform API to lower layers of a network, and allows implementing upper layers within sockets application.


Application formats (e.g. HTML, XML)
Layer 5-7: session, present, & applic (e.g., SSL, HTTP)
Socket Interface
API
Layer 4: Transport (TCP or UDP)
Layer 3: Network (e.g. IP)
Layer 2: Data (e.g. Ethernet)
Layer 1: Physical (e.g. Twisted pair)

## Socket

1. The basic building block for communication is the socket. **A socket is an endpoint of communication** to which a name may be bound. Each socket in use has a type and an associated process. Sockets exist within communication domains.

2. A communication **domain is an abstraction** introduced **to bundle common properties of threads** communicating through sockets. Sockets normally exchange data only with sockets in the same domain.

3. The Windows Sockets facilities support a single communication domain: **The Internet Domain**, which is used by processes which communicate using the Internet Protocol Suite.

**Two types of sockets currently are available to a user:-**

1. A **STREAM** socket provides bi-directional, reliable, sequenced, and unduplicated flow of data without record boundaries.

2. A **DATAGRAM** socket supports bi-directional flow of data which is not promised to be sequenced, reliable, or unduplicated.

## PROTOCOLS

### IP, TCP and UDP

1. TCP is a stream protocol, while. TCP establishes a continuous open connection between a client and a server, over which bytes may be written--and correct order guaranteed--for the life of the connection. Hover, bytes written over TCP have no built-in structure, so higher-level protocols are required to delimit any data records and fields within the transmitted byte stream.

2. UDP is a datagram protocol does not require any connection be established between client and server; it simply transmits a message between addresses. A feature of UDP is that its packets are self-delimiting each datagram indicates exactly where it begins and ends. A possible disadvantage of UDP is that it provides no guarantee that packets will transmit in-order or not. Higher-level protocols built on top of UDP may, of course, provide handshaking and acknowledgements.

### Peers, Ports, Names, and Addresses

1. Beyond the protocol--TCP or UDP--there are two things a **Peer** (a client or server) needs to know about the machine it communicates with: An **IP Address** and a **Port**.

2. An **IP address** is a **32-bit data** value, usually represented for humans in "dotted quad" notation, e.g., 64.41.64.172.

3. A **Port** is a **16-bit data value**, usually simply represented as a number less than 65536--most often one in the tens or hundreds range. An IP address gets a packet *to* a machine, a port lets the machine decide which process/service (if any) to direct it to.

4. Classful addressing divides IPv4 Adresses into 5 classes: Class A, B, C, D, and E. where as port numbers have Reserved Port Numbers or Well-Known Port Numbers (1-3000).

## SYNTAXES

In network applications process can be divided into: a Client and a Server.

**Creating a socket**

#include <sys/types.h>

#include <sys/socket.h>

When you create a socket there are three main parameters:

**int socket (intdomain, int type, int protocol)**

**Description**

Socket creates an endpoint for communication and returns a descriptor. The domain parameter specifies a communication domain this selects the protocol family which will be used for communication. These families are defined in <sys/socket.h>. The currently understood formats include:

| **Name** | **Purpose** |
|---|---|
| AF_UNIX, PF_LOCAL | Local communication |
| AF_INET | IPv4 Internet protocols |
| AF_INET6 | IPv6 Internet protocols |
| AF_IPX | IPX - Novell protocols |

The socket has the indicated type, which specifies the communication semantics. Currently defined types are:

**SOCK_STREAM:** Provides sequenced, reliable, two-way, connection- based byte streams. An out-of-band data transmission mechanism may be supported.

**SOCK_DGRAM:**Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

**SOCK_SEQPACKET:** Provides a sequenced, reliable, two-way connection- based data transmission path for datagrams of fixed maximum length; a consumer is required to read anentire packet with each read system call.

**SOCK_RAW:** Provides raw network protocol access.

**SOCK_RDM:** Provides a reliable datagram layer that does not guarantee ordering.

**SOCK_PACKET:** Obsolete and should not be used in new programs.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. The protocol number to use is specific to the "communication domain" in which communication is to take place.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. They do not preserve record boundaries. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect call. Once connected, data may be transferred using read and write calls or some variant of the send and recv calls. When a session has been completed a close may be performed. Out-of-band data may also be transmitted as described in send and received as described in recv.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send calls. Datagrams are generally received with recvfrom, which returns the next datagram with its return address. SOCK_PACKET is an obsolete socket type to receive raw packets directly from the device driver.

**Return Value**

-1 is returned if an error occurs otherwise the return value is a descriptor referencing the socket.

**Example:**

**Sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP)**

Where'Sockfd' is the file descriptor returned by the socket function.
AF_INET is the address family.
SOCK_STREAM is the type of packet.
IPPROTO_TCP is the protocol.

**Binding a socket to a port and waiting for the connections**

**<u>Bind:</u>**

Bind will bind a name to socket,it gives the socket sockfd, the local address myaddr, myaddr is addrlen bytes long.Traditionallythis is called assigning a name to a socket. When a socket is created with sockect () it exists in a name space but has no name assigned. It is normally necessary to assign local address using bind before sock_stream socket may receive connection.

**int bind (intsockfd, (structsockaddr *)myaddr,socklen_taddrlen);**

The struct necessary to make socket works is the structsockaddr_in address; and then we have the follow lines to say to system the information about the socket.

**The type of socket**

address.sin_family = AF_INET /* use a internet domain */

**The IP used**

address.sin_addr.s_addr = INADDR_ANY /*use a specific IP of host*/

**The port used**

address.sin_port = htons (15000); /* use a specific port number */

And finally bind our port to the socket

bind(create_socket , (structsockaddr *)&address,sizeof(address));

**Return Value:**

On success ( ) is return, on error –1 is returned and errno is set to appropriately.

**Listen:**

Listen for connections on a socket

#include <sys/socket.h>

int listen(int s, int backlog);

**Description**

To accept connections, a socket is first created with socket, a willingness to accept incoming connections and a queue limit for incoming connections are specified with listen, and then the connections are accepted with accept. The **listen** call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.The backlog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that retries succeed.

Return Value On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

**Connect**: initiate a connection on a socket.The connect operation is used on the client side to identify and, possibly, start the connection to the server. The connect syntax is

#include <sys/types.h>

#include <sys/socket.h>

int connect(intsockfd, conststructsockaddr *serv_addr, sizeof(address));

**Description**

The file descriptor sockfd must refer to a socket. If the socket is of type SOCK_DGRAM then the serv_addr address isthe address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type SOCK_STREAM or SOCK_SEQPACKET, this call attempts to make a connection to another socket. The other socket is specified by serv_addr, which is an address (of length addrlen) in the communications space of the socket. Each communications space interprets the serv_addr parameter in its own way.

Generally, connection-based protocol sockets may successfully connect only once; connectionless protocols sockets may use connect multiple times to change their association. Connectionless sockets may dissolve the association by connecting to an address with the sa_family member of sockaddr set to AF_UNSPEC.

RETURN VALUE

If the connection or binding succeeds, zero is returned.

On error, -1 is returned, and errno is set appropriately.

**Accept:**it accepts a socket connection. And to finish we have to tell the server to accept a connection, using the accept() function. Accept is used with connection based sockets such as streams.

#include <sys/types.h>

#include <sys/socket.h>

int accept(int s, structsockaddr *addr, socklen_t *addrlen);

## DESCRIPTION

The accept function is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET and SOCK_RDM). It extracts the first connection request on the queue of pending connections, creates a new connected socket with mostly the same properties as s, and allocates a new file descriptor for the socket, which is returned. The newly created socket is no longer in the listening state. The original socket s is unaffected by this call. Note that any per file descriptor flags (everything that can be set with the F_SETFL fcntl, like non- blocking or async state) are not inherited across an accept.

In order to be notified of incoming connections on a socket, you can use select. A readable event will be delivered when a new connection is attempted and you may then call accept to get a socket for that connection. Alternatively, you can set the socket to deliver SIGIO when activity occurs on a socket;

There may not always be a connection waiting after a SIGIO is delivered or select or poll return a readability event because the connection might have been removed by an asynchronous network error or another thread before accept is called. If this happens then the call will block waiting for the next connection to arrive. To ensure that accept never blocks, the passed socket s needs to have the O_NONBLOCK flag set.

## RETURN VALUE

The call returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket

**RECV, RECVFROM:**receive a message from a socket

#include <sys/types.h>

#include <sys/socket.h>

intrecv(int s, void *buf, size_tlen, int flags);

intrecvfrom(int  s,  void  *buf,  size_tlen, int flags

Structsockaddr *from, socklen_t *fromlen;

**Description**

The recvfrom and recvmsg calls are used to receive messages from a socket, and may be used to receive data on socket whether or not it is connection-oriented.

If from is not NULL, and the socket is not connection-oriented, the source address of the message is filled in the argument fromlen is a value-result parameter, initialized to the size of the buffer associated with from, and modified on return to indicate the actual size of the address stored there.

The recv call is normally used only on a connected socket and is identical to recvfrom with a NULL from parameter.

**RETURN VALUE**

These calls return the number of bytes received, or -1 ifan error occurred.

**SENDTO, SEND:**send a message from a socket

#include <sys/types.h>

#include <sys/socket.h>

```
int send(int s, const void *msg, size_tlen, int flags);

intsendto(int s, const void *msg, size_tlen, intflags,conststructsockaddr *to, socklen_ttolen);
```

**Description**

Send, sendto are used to transmit a message to another socket. Send may be used only when the socket is in a connected state, while sendto may be used at any time. The address of the target is given by to with tolen specifying its size. The length of the message is given by len. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted. No indication of failure to deliver is implicit in a send. Locally detected errors are indicated by a return value of-1.

When the message does not fit into the send buffer of the socket, send normally blocks, unless the socket has been placed in non-blocking I/O mode. In non-blocking mode it would return EAGAIN in this case. The select call may be used to determine when it is possible to send more data.

**RETURN VALUE**

The calls return the number of characters sent, or -1 if an error occurred.

## Server Software

Servers can be classified based on:-

1. Servicing discipline (iterative or concurrent);
2. Communication methods (connection-oriented orconnectionless);
3. Maintenance of history (stateless or stateful).

### Iterative Server

An iterative server serves the requests one after the other.It is easier to design and implement. Iterative design is suitable when the service time for each request is small (because the mean response time is still acceptably small).It is suitable for simple services such as the TIME service.Iterative design is not suitable when the service time for arequest may be large.

**Example**

Two clients are using a file transfer service:

1. The 1st client requests to get a file of size 200 Mbytes,
2. The 2nd client requests to get a file of size 20 bytes.

If iterative design is used, the 2nd client has to wait a long time.

## Concurrent Server

A concurrent server serves multiple clients simultaneously.

Concurrency refers to either

1. real simultaneous computing (using multiple processors)
2. apparent simultaneous computing (by time sharing).Concurrent design is more difficult to implement, but it cangive a smaller response time when
3. Different clients require very different service time, orthe service requires significant I/O, or the server is executed on a computer with multipleprocessors.

## Connection-Oriented Servers

A connection-oriented server uses TCP for connection-oriented communication. TCP provides reliable transport.The servers need not do so. The servers are simpler. A server uses a separate socket for each connection.

## Connectionless Servers

A connectionless server uses UDP for connectionlesscommunication. UDP does not guarantee reliable transport.A connectionless server may need to realize reliabilitythrough timeout and retransmission.The server and client are relatively complicated.

## Stateless Server

A stateless server does not keep information about the on-going interactions with each client.

**Example**

Consider the service that a client can read a file storedon the server's computer.

1. Tomaintain stateless, each client's request must specifya file name, a position in the file, and the number ofbytes to read.
2. The server handles each request independently:
3. Open the specified file,
4. Seek to the specified position,
5. Read the specified number of bytes,
6. Send these bytes to the client,
7. Close the file.

Stateless servers are less efficient but they are more reliable they may be a good choice for UDP transport.

**Stateful Servers**

A stateful server keeps information about the status ongoing interactions with each client.

**Example**

Consider the file-reading service.

The server maintains the following information: File reading can be more efficient.

**Four possible types of server designs:**

1. Iterative, connection-oriented server.
2. Iterative, connectionless server.
3. Concurrent, connection-oriented server.
4. Concurrent, connectionless server.

Some iterative server use tcp because to send data reliably.

## Process and Thread

We can use processes and threads to realize concurrency ona processor via time sharing.

**Process:** A process is a fundamental unit of computation. The OShas to store various information about each process.

**Thread:**

1. Windows provides a second form of concurrentexecution known as threads of execution or threads.
2. Each thread must be associated with a single process.
3. A thread is executed independently of other threads.
4. All threads in a process share: (i) global variables and(ii) resources that the OS allocates to the process.
5. Each thread in a process has its own local variables.

For example, if multiple threads execute the following piece of code,

```
For (i=1; i<=10; i++)
Printf (“%d\n”, i);
```

Then each thread has its own index variable i.

## Remote Procedure Call (RPC)

Remote Procedure Call (RPC) provides a different paradigm for accessing network services. Instead of accessing remote services by sending and receiving messages, a client invokes services by making a local procedure call. The local procedure hides the details of the network communication.

When making a remote procedure call:

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

The main goal of RPC is to hide the existence of the network from a program. As a result, RPC doesn't quite fit into the OSI model:

1. The message-passing nature of network communication is hidden from the user. The user doesn't first open a connection, read and write data, and then close the connection. Indeed, clients often does not even know they are using the network!
2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often. For example, on (diskless) Sun workstations, every file access is made via an RPC.

RPC is especially well suited for client-server (e.g., query-response) interaction in which the flow of control alternates between the caller and callee. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

The following steps take place during an RPC:

1. A client invokes a *client stub* procedure, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub*marshalls* the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a *server stub*, which demarshalls the parameters and calls the desired server routine using the regular procedure call mechanism.

5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

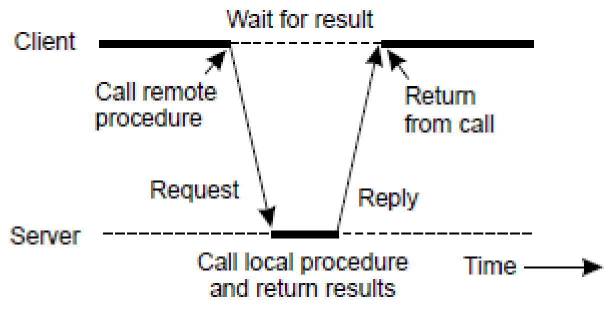


**Figure 2-2.** Principle of RPC between a client and server program.

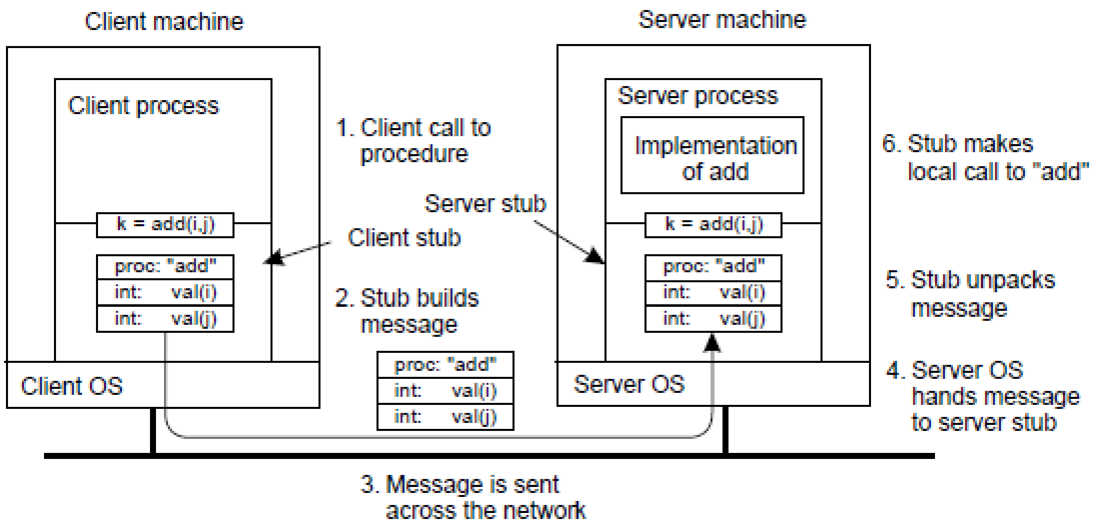


**Figure 2-3.** The steps involved in a doing a remote computation through RPC.

**Understanding of Arp, Telnet, Ftp, Finger, Traceroute, Ifconfig, Netstat, Ping Commands**

## i. ARP

arp - manipulate the system ARP cache

```
arp [-evn] [-H type] [-i if] -a [hostname]

arp [-v] [-i if] -d hostname [pub]

arp [-v] [-H type] [-i if] -s hostname hw_addr [temp]

arp  [-v]  [-H  type] [-i if] -s hostname hw_addr [netmask nm] pub

arp [-v] [-H type] [-i if] -Ds hostname ifa  [netmask  nm]  pub

arp [-vnD] [-H type] [-i if] -f [filename]
```

### DESCRIPTION

Arp manipulates the kernel's ARP cache in various ways. The primary options are clearing an address mapping entry and manually setting up one. For debugging purposes, the ARP program also allows a complete dump of the ARP cache.

#### OPTIONS

| | |
|---|---|
| -v, --verbose | Tell the user what is going on by being verbose. |
| -n, --numeric | shows numerical addresses instead of port or user names. |
| -H type, --hw-type type, -t type | When setting or reading the ARP cache, this optional parameter tells arp which class of entries it should check for. |

| | |
|---|---|
| -a [hostname], --display [hostname] | Shows the entries of the specified hosts. If the hostname parameter is not used, all entries will be displayed. The entries will be displayed in alternate (BSD) style. |
| -d hostname, --delete hostname | Remove any entry for the specified host. |
| -D, --use-device | Use the interface ifa's hardware address. |
| -e | shows the entries in default (Linux) style. |
| -i If, --device If | Select an interface. When dumping the ARP cache only entries matching the specified interface will be printed. |
| -s hostname hw_addr, --set hostname | Manually create an ARP address mapping entry forhost hostname with hardware address set to hw_addr class, but for most classes one can assume that the usual presentation can be used. |
| -f filename, --file filename | Similar to the -s option, only this time the address info is taken from file filename set up. The name of the data file is very often /etc/ethers, but this is not official. If no filename is specified /etc/ethers is used as default. |

## ii. TELNET

user interface to the TELNET protocol

telnet [-8EFKLacdfrx] [-X authtype] [-b hostalias] [-e escapechar]

[-k realm] [-l user] [-n tracefile] [host [port]]

### DESCRIPTION

The telnet command is used to communicate with another host using the TELNET protocol. If telnet is invoked without the host argument, it enters command mode, indicated by its prompt (telnet>). In this mode, it accepts and executes the commands listed below. If it is invoked with arguments, it performs an open command with those arguments.

#### OPTIONS

| | |
|---|---|
| -8 | Specifies an 8-bit data path. This causes an attempt to negotiate the TELNET BINARY option on both input and output. |
| -E | Stops any character from being recognized as an escape character. |
| -F | If Kerberos V5 authentication is being used, the -F option allows the local credentials to be forwarded to the remote system including any credentials that have already been forwarded into the local environment. |
| -K | Specifies no automatic login to the remote system. |
| -L | Specifies an 8-bit data path on output. This causes the BINARY option to be negotiated on output. |
| -X atype | disables the atype type of authentication. |
| -a | Attempt automatic login. Currently, this sends the user name via the USER variable of the ENVIRON option if supported by the remote system. |

| | |
|---|---|
| -b  hostalias | Uses bind(2) on the local socket to bind it to an aliased address or to the address of another interface than the one naturally chosen by connect(2). |
| -c | disables the reading of the user's .telnetrc file. |
| -d | sets the initial value of the debug toggle to TRUE. |
| -eescapechar | Sets the initial telnet escape character to escape char.  If escape char is omitted, then there will be no escape character. |
| -f | If Kerberos V5 authentication is being used, the -f option allows the local credentials to be forwarded to the remote system. |
| -k realm | If Kerberos authentication is being used, the -k option requests that telnet obtain tickets for the remote host in realm realm instead of the remote host's realm, as determined by krb_realmofhost(3). |
| -l user | this option implies   the -a option.  This option may also be used with the open command. |
| -n tracefile | Opens tracefile for recording trace information. |
| -r | Specifies a user interface similar to rlogin(1).  In this mode, the escape character is set to the tilde (~) character, unless modified by the -e option. |
| -x | turns on encryption of the data stream if possible. |
| host | Indicates the official name, an alias, or the Internet address of   a remote host. |
| port | Indicates a port number (address of an application).  If a number is not specified, the default telnet port is used.  When in rlogin mode, a line of the form ~.  Disconnects from the remote prompt. |

### iii. FINGER

finger - user information lookup program

finger [-lmsp] [user ...] [user@host ...]

**DESCRIPTION**

The finger displays information about the system users.

**OPTIONS**

| | |
|---|---|
| -s | Finger displays the user's login name, real name, terminal name and write status (as a ``*'' after the terminal name if write permission is denied), idle time, login time, office location and office phone number. |
| -l | produces a multi-line format displaying all of the information |
| -m | Prevent matching of user names. User is usually a login name; however, matching will also be done on the users' real names, unless the -m option is supplied. All name matching performed by finger is case insensitive. |

### iv. FTP

ftp - Internet file transfer program

ftp [-pinegvd] [host]

pftp [-inegvd] [host]

**DESCRIPTION**

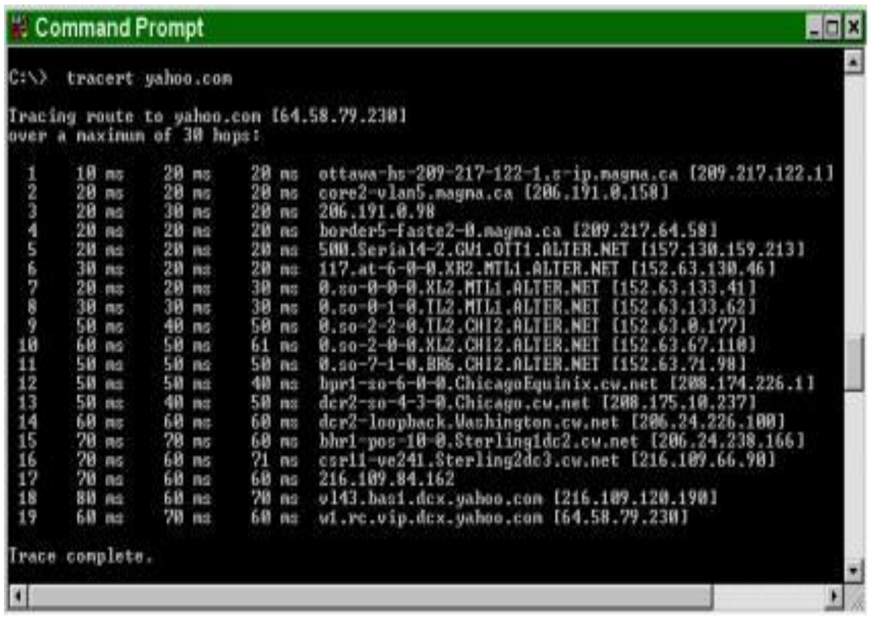
Ftp is the user interface to the Internet standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site. Options may be specified at the command line, or to the command interpreter.

**OPTIONS**

| | |
|---|---|
| -p | Use passive mode for data transfers. Allows use of ftp in environ¡ments where a firewall prevents connections from the outside world back to the client machine. |
| -i | turn off interactive prompting during multiple file transfers. |
| -n | Restrains ftp from attempting ``auto-login" upon initial connection. |
| -e | Disables command editing and history support, if it was compiled into the ftp executable. Otherwise, does nothing. |
| -g | Disables file name globing. |
| -v | verbose option forces ftp to show all responses from the remote server, as well as report on data transfer statistics. |
| -d | Enables debugging. The client host with which ftp is to communicate may be specified on the command line. |

## v. TRACEROUTE

The traceroute command traces the network path of Internet routers that packets take as they are forwarded from your computer to a destination address. The "length" of the network connection is indicated by the number of Internet routers in the traceroute path. Traceroutes can be useful to diagnose slow network connections. On a Windows computer, you can run a traceroute in an MSDOS or Command window by typing "tracert" followed by the domain name, for example as in "tracertwww.yahoo.com"



```
C:\>  tracert yahoo.com

Tracing route to yahoo.com [64.58.79.230]
over a maximum of 30 hops:

  1    10 ms    20 ms    20 ms  ottawa-hs-209-217-122-1.s-ip.magna.ca [209.217.122.1]
  2    20 ms    20 ms    20 ms  core2-vlan5.magna.ca [206.191.0.158]
  3    20 ms    30 ms    20 ms  206.191.0.98
  4    20 ms    20 ms    20 ms  border5-faste2-0.magna.ca [209.217.64.58]
  5    20 ms    20 ms    20 ms  500.Serial4-2.GW1.OTT1.ALTER.NET [157.130.159.213]
  6    30 ms    20 ms    20 ms  117.at-6-0-0.XR2.MTL1.ALTER.NET [152.63.130.46]
  7    20 ms    20 ms    30 ms  0.so-0-0-0.XL2.MTL1.ALTER.NET [152.63.133.41]
  8    30 ms    30 ms    30 ms  0.so-0-1-0.TL2.MTL1.ALTER.NET [152.63.133.62]
  9    50 ms    40 ms    50 ms  0.so-2-2-0.TL2.CHI2.ALTER.NET [152.63.0.177]
 10    60 ms    50 ms    61 ms  0.so-2-0-0.XL2.CHI2.ALTER.NET [152.63.67.110]
 11    50 ms    50 ms    50 ms  0.so-7-1-0.BR6.CHI2.ALTER.NET [152.63.71.98]
 12    50 ms    50 ms    40 ms  bpr1-so-6-0-0.ChicagoEquinix.cw.net [208.174.226.1]
 13    50 ms    40 ms    50 ms  dcr2-so-4-3-0.Chicago.cw.net [208.175.10.237]
 14    60 ms    60 ms    60 ms  dcr2-loopback.Washington.cw.net [206.24.226.100]
 15    70 ms    70 ms    60 ms  bhr1-pos-10-0.Sterling1dc2.cw.net [206.24.238.166]
 16    70 ms    60 ms    71 ms  csr11-ve241.Sterling2dc3.cw.net [216.109.66.90]
 17    70 ms    60 ms    60 ms  216.109.84.162
 18    80 ms    60 ms    70 ms  v143.bas1.dcx.yahoo.com [216.109.120.190]
 19    60 ms    70 ms    60 ms  w1.rc.vip.dcx.yahoo.com [64.58.79.230]

Trace complete.
```

**OPTIONS**

| Option | Definition |
|---|---|
| -m | Set the maximum Time To Live (TTL) for the trace, measured as the number of hosts the program will trace before ending, default of 30 |
| -q | Set the number of UDP packets to send for each setting, default of 3. |
| -w | Set the amount of seconds to wait for an answer from each host before giving up, default of 5 |
| -p | Specify the other host's invalid port address, default of 33434 |

Print the route packets take to network host.

## vi. NETSTAT

Netstat command displays the contents of various network-related data structures in various formats, depending on the options you select. The first form of the command displays a list of active sockets for each protocol. The second form selects one from among various other network data structures. The third form shows the state of the interfaces. The fourth form displays the routing table, the fifth form displays the multicast routing table, and the sixth form displays the state of DHCP on one or all interfaces.

**SYNTAX**

netstat [-a] [-n] [-v]

netstat [-g | -m | -p | -s | -f address_family ] [-n] [-P protocol]

netstat [ -i ] [ -I interface ] [ interval ]

netstat -r [-a] [-n] [-v ]

netstat -M [-n] [-s ]

netstat -D [ -I interface ]

| | |
|---|---|
| -a | Show the state of all sockets and all routing table entries; normally, sockets used by server processes are not shown and only interface, host, network, and default routes are shown. |
| -n | Show network addresses as numbers. netstat normally displays addresses as symbols. This option may be used with any of the display formats. |
| -v | Verbose. Show additional information for the sockets and the routing table. |
| -g | Show the multicast group memberships for all interfaces. |
| -m | Show the STREAMS statistics. |

| | |
|---|---|
| -p | Show the address resolution (ARP) tables. |
| -s | Show per-protocol statistics. When used with the -M option, show multicast routing statistics instead. |
| -i | Show the state of the interfaces that are used for TCP/IP traffic. |
| -r | Show the routing tables. |
| -M | Show the multicast routing tables. When used with the -s option, show multicast routing statistics instead. |
| -d | Show the state of all interfaces that are under Dynamic Host Configuration Protocol (DHCP) control. |
| -D | Show the status of DHCP configured interfaces. |
| -f address_family | imit statistics or address control block reports to those of the specified address_family, which can be one of:<br><br>inet For the AF_INET address family unix For the AF_UNIX address family |
| -P protocol | Limit display of statistics or state of all sockets to those applicable to protocol. |
| - I interface | Show the state of a particular interface. interface can be any valid interface such as ie0 or le0. |

## vii. IFCONFIG

The "ifconfig" command allows the operating system to setup network interfaces and allow the user to view information about the configured network interfaces.

**SYNTAX**

```
ifconfig [-L] [-m] interface [create] [address_family] [address[/prefixlength] [dest_address]]
        [parameters] ifconfig interface destroy
        ifconfig -a [-L] [-d] [-m] [-u] [address_family]
        ifconfig -l [-d] [-u] [address_family]
        ifconfig [-L] [-d] [-m] [-u] [-C]
```

| | |
|---|---|
| address | For the DARPA-Internet family, the address is either a host name present in the host name data base, or a DARPA Internet address expressed in the Internet standard ``dot notation". It is also possible to use the CIDR notation (also known as the slash notation) to include the netmask. That is, one can specify an address like 192.168.0.1/16. |
| addres_family | Specify the address family which affects interpretation of the remaining parameters. Since an interface can receive transmissions in differing protocols with different naming schemes, specifying the address family is recommended. The address or protocol families currently supported are ``inet", ``inet6", |
| dest_address | Specify the address of the correspondent on the other end of a point to point link. |
| interface | This parameter is a string of the form ``name unit", for example, ``en0". |

## viii. PING

Sends ICMP ECHO_REQUEST packets to network hosts.

**SYNTAX**

```
ping -s [-d] [-l] [-L] [-n] [-r] [-R] [-v] [ -i interface_address ] [-I interval] [-t ttl] host [packetsize]
        [count]
```

| | |
|---|---|
| -d | Set the SO_DEBUG socket option. |
| -l | Loose source route. Use this option in the IP header to send the packet to the given host and back again. Usually specified with the -R option. |
| -L | Turn off loopback of multicast packets. Normally, if there are members in the host group on the out- going interface, a copy of the multicast packets will be delivered to the local machine. |
| -n | Show network addresses as numbers. ping normally displays addresses as host names. |
| -r | Bypass the normal routing tables and send directly to a host on an attached network. If the host is not on a directly-attached network, an error is returned. This option can be used to ping a local host through an interface that has been dropped by the router daemon. |
| -R | Record route. Sets the IP record route option, which will store the route of the packet inside the IP header. The contents of the record |

Ping computerhope.com - Would ping the host computerhope.com to see if it is alive

## 3. Implementation of Concurrent Echo server using connection- oriented Socket System Calls.

**SERVER:**

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

main(int argc,char *argv[])

{

int sockfd,new_sockfd,rval,pid;

char buff1[100],buff2[100];

struct sockaddr_in server,client;

int len;

sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(sockfd==-1)

{

perror("\n sock_err\n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr("argv[1]");

server.sin_port=htons(atoi(argv[2]));

rval=bind(sockfd,(struct sockaddr*)&server,sizeof(server));

if(rval!=-1)
```

```
{
listen(sockfd,5);
new_sockfd=accept(sockfd,(struct sockaddr*)&client,&len);
if(new_sockfd!=-1)
{
pid=fork();
if(pid==0)
{
printf("\n child process executing\n");
printf("\n child process id is %d",getpid());
rval=recv(new_sockfd,buff1,100,0);
if(rval==-1)
{
perror("\n RECV_ERR\n");
exit(1);
}
else
{
printf("\n received message is %s\n",buff1);
}
rval=send(new_sockfd,buff1,sizeof(buff1),0);
if(rval!=-1)
{
printf("\n message sentto successfully\n");
}
else
{
perror("\n SEND_ERR\n");
exit(1);
}
}
else
{
printf("\nparents process\n");
printf("\nparents process id is %d\n");
exit(1);
}
}
else
{
perror("\nACCEPT_ERR\n");
exit(1);
}
}
else
{
perror("\nBIND_ERR\n");
close(sockfd);
```

**<u>CLIENT</u>:**

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

main(int argc,char *argv[1])

{

int sockfd,rval;

char buff1[100],buff2[100];

struct sockaddr_in server,client;

int len;

sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(sockfd==-1)

{

perror("\n SOCK_ERR\n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr("192.168.0.5");

server.sin_port=htons(5432);

rval=connect(sockfd,(struct sockaddr*)&server,sizeof(server));

if(rval!=-1)

{

printf("\nenter a message\n");
scanf("%s",buff1);
rval=send(sockfd,buff1,sizeof(buff1),0);
if(rval==-1)
{
perror("\nSEND_ERR\n");
exit(1);
}
rval=recv(sockfd,buff2,100,0);
if(rval!=-1)
{
printf("\n received message is %s \n",buff2);
}
else
{
perror("\n RECV_ERR\n");
exit(1);
}
}
else
{
printf("\n CONNECT_ERR");
exit(1);
}
}
```

**OUTPUT:**

**Server execution:**

```
cc 11s.c

./a.out 192.168.0.5 3456

Child process executing

Parent process

Parent process ID is 7273

Child process ID is 7947

Received message is hello

Message sent successfully
```

**Client execution**

```
cc 11c.c

./a.out 192.168.0.5 3456|

Enter a message

Hello

Received message is hello
```

## 4. Implementation of Concurrent Echo Server Using Connection- Less Socket System Calls.

**SERVER**

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

main(int argc,char *argv[])

{

int sockfd,rval,pid;

char buff1[100],buff2[100];

struct sockaddr_in server,client;

int len;

sockfd=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

if(sockfd==-1)

{

perror("\n SOCK_ERR\n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr(argv[1]);

server.sin_port=htons(atoi(argv[2]));

rval=bind(sockfd,(struct sockaddr *)&server,sizeof(server));

if(rval!=-1)
```

```
{
pid=fork();
if(pid==0)
{
printf("\n childprocess executing\n");
printf("\n childprocess ID is:%d\n",getpid());
len=sizeof(client);
rval=recvfrom(sockfd,buff1,20,0,(struct sockaddr *)&client,&len);
if(rval==-1)
{
perror("\n RECV_ERR\n");
exit(1);
}
else
{
printf("\n received message is:%s\n",buff1);
}
rval=sendto(sockfd,buff1,sizeof(buff1),0,(struct sockaddr *)&client,sizeof(client));
if(rval!=-1)
{
printf("\n message sent successfully\n");
}
else
{
perror("\n SEND_ERR\n");
exit(1);

}

}

else

printf("\n parent process\n");

printf("parent process ID is%d\n",getppid());

}

else

{

perror("\n BIND_ERR\n");

exit(1);

}

}
```

**CLIENT :**

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
main(int argc,char *argv[])
{
int sockfd,rval;
char buff1[100],buff2[100];
struct sockaddr_in server,client;
int len;
sockfd=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
if(sockfd==-1)
{
perror("\n SOCK_ERR\n");
exit(1);
}
server.sin_family=AF_INET;
server.sin_addr.s_addr=inet_addr(argv[1]);
server.sin_port=htons(atoi(argv[2]));
rval=bind(sockfd,(struct sockaddr *)&client,sizeof(client));
if(rval==-1)
{
perror("\n BIND_ERR\n");
exit(1);
}
printf("\n enter a message\n");
scanf("%S",buff1);
rval=sendto(sockfd,buff1,sizeof(buff1),0,(struct sockaddr *)&server,sizeof(server));
if(rval!=-1)
{
printf("\n message sent successfully\n");
}
else
{
perror("\n SEND_ERR\n");
exit(1);
}
len=sizeof(server);
rval=recvfrom(sockfd,buff1,20,0,(struct sockaddr *)&server,&len);
if(rval==-1)
{
perror("\n RECV_ERR\n");
exit(1);
}
else
{
printf("\n received message is %s\n",buff1);
}}
```

**OUTPUT:**

**Server execution**

```
cc 12s.c

./a.out 192.168.0.5  9566

Child process executing

Child process ID is 15572

Parent process

Parent process ID is 11736

Received message is  hellow

Message sent successfully
```

**Client execution**

```
cc 12c.c

./a.out 192.168.0.5  9566

Enter a message

Hellow

Message sent successfully.
```

## 5. Implementation of Iterative Echo Server Using Connection-Oriented Socket System Call

**SERVER:**

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

main(int argc,char *argv[])

{

int sockfd,new_sockfd,rval,pid;

char buff1[20],buff2[20];

struct sockaddr_in server,client;

socklen_t len;

sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(sockfd==-1)

{

perror("\n sock_ERR\n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr(argv[1]);

server.sin_port=htons(atoi(argv[2]));

rval=bind(sockfd,(struct sockaddr*)&server,sizeof(server));

if(rval!=-1)
```

```
{
listen(sockfd,5);
while(1)
{
new_sockfd=accept(sockfd,(struct sockaddr*)&client,&len);
if(new_sockfd!=-1)
{
rval=recv(new_sockfd,buff1,20,0);
if(rval==-1)
{
perror("\n RECV_ERR \n");
exit(1);
}
else
{
printf("\n recieved message is %s \n",buff1);
}
rval=send(new_sockfd,buff1,sizeof(buff1),0);
if(rval!=-1)
{
printf("\n message sent successfully \n");
}
else
{
perror("\n SEND_ERR \n");
exit(1);
}
}
else
{
perror("\n ACCEPT_ERR \n");
exit(1);
}
}
else
{
perror("\n BIND_ERR\n");
close(sockfd);
}
```

**CLIENT:**

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

main(int argc,char *argv[])

{

int sockfd,rval;

char buff1[20],buff2[20];

struct sockaddr_in server,client;

int len;

sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(sockfd==-1)

{

perror("\n SOCK_ERR \n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr(argv[1]);

server.sin_port=htons(atoi(argv[2]));

rval=connect(sockfd,(struct sockaddr*)&server,sizeof(server));

if(rval!=-1)

{

printf("\n enter a message\n");
scanf("%s",buff1);

rval=send(sockfd,buff1,sizeof(buff1),0);

if(rval==-1)

{

perror("\n SEND_ERR \n");

exit(1);

}

rval=recv(sockfd,buff2,sizeof(buff2),0);

if(rval!=-1)

{

printf("\n recieved message is %s \n",buff2);

}

else

{

perror("\n RECV_ERR \n");

exit(1);

}

}

else

{

printf("\n CONNECT_ERR \n");

exit(1);

}

}
```

**OUTPUT:**

**Server execution**

```
cc 13s.c

./a.out

Received message is hellow

Message sent successfully
```

**Client execution**

```
cc 13c.c

./a.out

Enter a message

Hellow

Received message is hellow
```

## 6. Implementation of Iterative Echo Server Using Connectionless Socket System Call.

**SERVER:**

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

main(int argc,char *argv[])

{

int sockfd,rval;

char buff1[100],buff2[100];

struct sockaddr_in server,client;

int len;

sockfd=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

if(sockfd==-1)

{

perror("\n sock_err\n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr(argv[1]);

server.sin_port=htons(atoi(argv[2]));

rval=bind(sockfd,(struct sockaddr*)&server,sizeof(server));

if(rval!=-1)
{

len=sizeof(client);

rval=recvfrom(sockfd,buff1,100,0,(struct sockaddr*)&client,&len);

if(rval==-1)

{

perror("\nRECV_ERR\n");

exit(1);

}

else

{

printf("\nreceived message is %s\n",buff1);

}

rval=sendto(sockfd,buff1,sizeof(buff1),0,(struct sockaddr*)&client,sizeof(client));

if(rval!=-1)

{

printf("\nmessage sent successfully\n");

}

else

{

perror("\nSEND_ERR\n");

exit(1);

}

}

}
```

**CLIENT:**

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
main(int argc,char *argv[])
{
int sockfd,rval;
char buff1[100],buff2[100];
struct sockaddr_in server,client;
int len;
sockfd=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
if(sockfd==-1)
{
perror("\nSOCK_ERR\n");
exit(1);
}
server.sin_family=AF_INET;
server.sin_addr.s_addr=inet_addr(argv[1]);
server.sin_port=htons(atoi(argv[2]));
rval=bind(sockfd,(struct sockaddr*)&client,sizeof(client));
if(rval==-1)
{
perror("\nBIND_ERR\n");
exit(1);
}
printf("\nenter a message\n");
scanf("%s",buff1);
rval=sendto(sockfd,buff1,sizeof(buff1),0,(struct sockaddr*)&server,sizeof(server));
if(rval!=-1)
{
printf("\nmessage sent successfully\n");
}
else
{
perror("\nSEND_ERR\n");
exit(1);
}
len=sizeof(server);
rval=recvfrom(sockfd,buff1,100,0,(struct sockaddr*)&server,&len);
if(rval==-1)
{
perror("\nRECV_ERR\n");
exit(1);
}
else
{
printf("\nreceived message is %s\n",buff1);
}}
```

**<u>OUTPUT</u>:**

**<u>Server execution</u>**

```
cc 14s.s

./a.out 192.168.0.5 5757

received message is hellow

message sent successfully
```

**<u>Client execution</u>**

```
cc 14c.c

./a.out 192.168.0.5 5757

Enter a message

Hellow

Message sent successfully
```

## 7. Implementation of time service as Connection Oriented Concurrent Server using Socket System Calls.

**SERVER**:

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

main(int argc,char *argv[])

{

int sockfd,new_sockfd,rval,pid;

char buff1[100],buff2[100];

struct sockaddr_in server,client;

int len;

struct timeval tv;

sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(sockfd==-1)

{

perror("\n sock_err\n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr(argv[1]);

server.sin_port=htons(atoi(argv[2]));
```

```
rval=bind(sockfd,(struct sockaddr*)&server,sizeof(server));

if(rval!=-1)

{

listen(sockfd,5);

new_sockfd=accept(sockfd,(struct sockaddr*)&client,&len);

if(new_sockfd!=-1)

{

pid=fork();

if(pid==0)

{

printf("\n child process executing\n");

printf("\n child process id is %d",getpid());

rval=recv(new_sockfd,buff1,100,0);

if(rval==-1)

{

perror("\n RECV_ERR\n");

exit(1);

}

else

{

printf("\n received message is %s\n",buff1);

}

rval=send(new_sockfd,buff1,sizeof(buff1),0);

if(rval!=-1)

{
```

```
printf("\n message sent successfully\n");
}
else
{
perror("\n SEND_ERR\n");
exit(1);
}
gettimeofday(&tv,NULL);
send(new_sockfd,(char *)tv.tv_sec,sizeof(tv.tv_sec),0);
}
else
{
printf("\n parents process\n");
printf("\n parents process id is %d\n", getppid());
exit(1);
}
}
else
{
perror("\nACCEPT_ERR\n");
exit(1);
}
}
else
{
```

```
perror("\nBIND_ERR\n");

close(sockfd);

}

}
```

**CLIENT:**

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

main(int argc,char *argv[1])

{

int sockfd,rval;

char buff1[100],buff2[100];

struct sockaddr_in server,client;

int len;

sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(sockfd==-1)

{

perror("\n SOCK_ERR\n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr(argv[1]);

server.sin_port=htons(atoi(argv[2]));

rval=connect(sockfd,(struct sockaddr*)&server,sizeof(server));

if(rval!=-1)

{
```

```
printf("\n enter a message\n");
scanf("%s",buff1);
rval=send(sockfd,buff1,sizeof(buff1),0);
if(rval==-1)
{
perror("\nSEND_ERR\n");
exit(1);
}
rval=recv(sockfd,buff2,100,0);
if(rval!=-1)
{
printf("\n received message is %s \n",buff2);
}
else
{
perror("\n RECV_ERR\n");
exit(1);
}
printf("time is % lu",buff2);
}
else
{
printf("\n CONNECT_ERR");
exit(1);
}}
```

**OUTPUT:**

**Server execution**

```
cc 15s.c

./a.out 192.168.0.5 5245

Child process executing

Parent process

Parent process ID is 11705

Child process is 17142

Received message is hellow

Message sent successfully
```

**Client execution**

```
cc 15c.c

./a.out  192.168.0.5  5245

Eneter a message

Hellow

Received message is hellow

Time is 3219263984
```

## 8. Implementation of Day Time Service as Connection Oriented Concurrent Server Using Socket System Calls.

<u>**SERVER**</u>**:**

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

main(int argc,char *argv[])

{

int sockfd,new_sockfd,rval,pid;

char buff1[100],buff2[100];

struct sockaddr_in server,client;

int len;

time_t t;

sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(sockfd==-1)

{

perror("\n sock_err\n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr(argv[1]);

server.sin_port=htons(atoi(argv[2]));
```

```
rval=bind(sockfd,(struct sockaddr*)&server,sizeof(server));
if(rval!=-1)
{
listen(sockfd,5);
new_sockfd=accept(sockfd,(struct sockaddr*)&client,&len);
if(new_sockfd!=-1)
{
pid=fork();
if(pid==0)
{
printf("\n child process executing\n");
printf("\n child process id is %d",getpid());
rval=recv(new_sockfd,buff1,100,0);
if(rval==-1)
{
perror("\n RECV_ERR\n");
exit(1);
}
else
{
printf("\n received message is %s\n",buff1);
}
t=time(NULL);
snprintf(buff1,sizeof(buff1),"%s",ctime(&t));
rval=send(new_sockfd,buff1,sizeof(buff1),0);
if(rval!=-1)
{
printf("\n message sent to successfully\n");
}
else
{
perror("\n SEND_ERR\n");
exit(1);
}
}
else
{
printf("\n parents process\n");
printf("\n parents process id is %d\n", getppid());
exit(1);
}
}
else
{
perror("\nACCEPT_ERR\n");
exit(1);
}
}
else
{
```

```
perror("\nBIND_ERR\n");
close(sockfd);
}
}
```

**<u>CLIENT</u>:**

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
main(int argc,char *argv[1])
{
int sockfd,rval;
char buff1[100],buff2[100];
struct sockaddr_in server,client;
int len;
sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
if(sockfd==-1)
{
perror("\n SOCK_ERR\n");
exit(1);
}
server.sin_family=AF_INET;
server.sin_addr.s_addr=inet_addr(argv[1]);
server.sin_port=htons(atoi(argv[2]));
```

```
rval=connect(sockfd,(struct sockaddr*)&server,sizeof(server));
if(rval!=-1)
{
printf("\nenter a message\n");
scanf("%s",buff1);
rval=send(sockfd,buff1,sizeof(buff1),0);
if(rval==-1)
{
perror("\nSEND_ERR\n");
exit(1);
}
rval=recv(sockfd,buff2,100,0);
if(rval!=-1)
{
printf("\n received message is %s \n",buff2);
}
else
{
perror("\n RECV_ERR\n");
exit(1);
}
}
else
{
printf("\n CONNECT_ERR");
exit(1);

}
```

**<u>OUTPUT:</u>**

**<u>Server execution</u>**

```
cc 16s.c

./a.out 192.168.0.5 6235

child process executing

parent process

parent process ID is 10756690

child process ID is 25130

recevied message is hellow

message sent successfully
```

**<u>client execution</u>**

```
cc 16c.c

./a.out  192.169.0.5 6235

enter a message

hellow

received message is
```

**Thu Feb 14 12.06:00 2013**

## 9. Remote Command Execution Using Any Shell Commands

**SERVER**

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
main(int argc,char *argv[])
{
int sockfd, new_sockfd, rval, fd;
char buff1[400],buff2[400];
struct sockaddr_in server, client;
int len;
sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
if(sockfd==-1)
{
perror("\nsock_error\n");
exit(1);
}
server.sin_family=AF_INET;
server.sin_addr.s_addr=inet_addr(argv[1]);
server.sin_port=htons(atoi(argv[2]));
rval=bind(sockfd,(struct sockaddr*)&server,sizeof(server));
```

```
if(rval!=-1)
{
listen(sockfd,5);
while(1)
{
len=sizeof(client);
new_sockfd=accept(sockfd,(struct sockaddr*)&client,&len);
if(new_sockfd!=-1)
{
rval=recv(new_sockfd,buff1,sizeof(buff1),0);
if(rval==-1)
{
perror("\n recev_error\n");
exit(1);
}
else
{
strcat(buff1,">file.txt");
system(buff1);
fd=open("file.txt",O_RDONLY,0666);
if(fd==-1)
{
printf("\n file_error\n");
exit(1);
}
read(fd,buff2,sizeof(buff2));
}
rval=send(new_sockfd,buff2,sizeof(buff2),0);
if(rval!=-1)
{
printf("\n message sent successfully!\n");
}
else
{
perror("\n send_error\n");
exit(1);
}
}
else
{
perror("\n accept_error");
exit(1);
}}}
else
{
perror("\n bind_error\n");
close(sockfd);
}
}
```

**CLIENT:**

```
#include <stdio.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <arpa/inet.h>

#include<netinet/in.h>

main(int argc, char *argv[]){

int sockfd,rval;

char buff1[400],buff2[400];

struct sockaddr_in server;

sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if(sockfd==-1)

{

perror("\nSOCK_ERR\n");

exit(1);

}

server.sin_family=AF_INET;

server.sin_addr.s_addr=inet_addr(argv[1]);

server.sin_port=htons(atoi(argv[2]));

rval=connect(sockfd,(struct sockaddr*)&server,sizeof(server));

if(rval!=-1)

{

printf("\n Enter any shell command: \n");

scanf("%s",buff1);

rval=send(sockfd,buff1,sizeof(buff1),0);
if(rval==-1)
{
perror("\nSEND_ERR\n");
exit(1);
}
rval=recv(sockfd,buff2,sizeof(buff2),0);
if(rval!=-1)
{
printf("\n received data is:%s\n", buff2);
}
else
{
printf("\n RECV_ERR\n");
}
}
else
{
perror("\nCONNECT_ERR\n");
}
}
```

**OUTPUT:**

**Server execution**

```
cc remotecommandexecutionserver.c
./a.out 192.168.0.5 5009
 message sent successfully!
```

**Client execution**

```
cc remotecommandexecutionclient.c
./a.out 192.168.0.5 5009


 Enter any shell command:
Ps
 received data is:  PID TTY          TIME CMD
 3152 pts/1    00:00:00 bash
 3229 pts/1    00:00:00 a.out
PID TTY          TIME CMD
152 pts/1    00:00:00 bash
229 pts/1    00:00:00 a.out
```

## 10.Demonstrate the Use of Advanced Socket System Calls

### i. <u>Getsockopt( )</u>

```
#include<stdio.h>

#include<unistd.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<sys/types.h>

int main(void)

{

int msg;

int len=sizeof(msg);

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

setsockopt(sfd,SOL_SOCKET,SO_SNDBUF,(char *)&msg,len);

printf("the source no %d\n",msg);

}
```

**<u>OUTPUT:</u>**

```
cc 19.c

/a.out

the source no 6270112
```

## ii. <u>Setsockopt( )</u>

```
#include<stdio.h>

#include<unistd.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<sys/types.h>

int main(void)

{

int msg=17221;

int len=sizeof(msg);

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

setsockopt(sfd,SOL_SOCKET,SO_SNDBUF,(char *)&msg,len);

printf("the source no %d\n",msg);

}
```

```
cc 18.c

./a.out

the source no 17221
```

**Select( )**

```
#include<stdio.h>

#include<sys/time.h>

#include<sys/types.h>

#include<unistd.h>

int main(void)

{

fd_set rfds;

struct timeval tv;

int retval;

FD_ZERO(&rfds);

FD_SET(0,&rfds);

tv.tv_sec=5;

tv.tv_usec=0;

retval=select(1,&rfds,NULL,NULL,&tv);

if(retval==-1)

perror("select()");

else if(retval)

printf("\n data is available now");

else

printf("\n no data within 5 sec");

return 0;

}
```

**<u>OUTPUT :</u>**

```
cc 21.c
./a.out
 no data within 5 sec


 ./a.out
Zafar
data is available now
```

iii. **<u>Readv( ) and Writev( )</u>**

**<u>Step 1</u>: create a file named file.txt**

**<u>Step 2</u>: Implementation of writev**

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include <netinet/in.h>
#include<sys/uio.h>
#include<fcntl.h>
main()
{
char s1[20],s2[20],s3[20],s4[20];
struct iovec iov[4];
int fd;
fd=open("file.txt",O_WRONLY);
printf("enter 1st string\n");
scanf("%s",&s1);
printf("enter 2nd string\n");
scanf("%s",&s2);
printf("enter 3rd string\n");
scanf("%s",&s3);
printf("enter 4th string\n");
scanf("%s",&s4);
iov[0].iov_base=s1;
iov[0].iov_len=sizeof(s1);
iov[1].iov_base=s2;
iov[1].iov_len=sizeof(s2);
iov[2].iov_base=s3;
iov[2].iov_len=sizeof(s3);
iov[3].iov_base=s4;
iov[3].iov_len=sizeof(s4);
writev(fd,(struct iovec*)&iov,4);
}
```

**OUTPUT:**

```
cc 17a.c
./a.out
enter 1st string
hello
enter 2nd string
my
enter 3rd string
nameis
enter 4th string
Ahsan
```

**<u>Step 3:</u> implementation of readv**

```
#include<sys/types.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include <netinet/in.h>
#include<sys/uio.h>
#include<fcntl.h>
main()
{
char s1[20],s2[20],s3[20],s4[20];
struct iovec iov[4];
int fd;
iov[0].iov_base=s1;
iov[0].iov_len=sizeof(s1);
iov[1].iov_base=s2;
iov[1].iov_len=sizeof(s2);
iov[2].iov_base=s3;
iov[2].iov_len=sizeof(s3);
iov[3].iov_base=s4;
iov[3].iov_len=sizeof(s4);
fd=open("file.txt",O_RDONLY);
readv(fd,(struct iovec*)&iov,4);
printf("first string is%s \n",s1);
printf("second string is %s\n",s2);
printf("third string is %s \n",s3);
printf("fourth string is %s\n",s4);
}
```

**OUTPUT:**

```
cc 17b.c
./a.out
first string ishello
second string is my
third string is nameis
fourth string is Ahsan
```

**iv. Getpeername( )**

```
#include <stdio.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <sys/types.h>

#define ERROR -1

main()

{

int s,k;

struct sockaddr_in server,addr;

socklen_t len;

s=socket(AF_INET,SOCK_STREAM,0);

server.sin_family=AF_INET;

inet_aton("192.168.0.5",&server.sin_addr);

server.sin_port=htons(80);

k=connect(s,(struct socaddr *)&server,sizeof(server));

if(k==0)

{

perror("connect");

exit(0);

}

len=sizeof(addr);

getpeername(s,(struct sockaddr *)&addr,&len);

printf("peer IP address: %s \n",inet_ntoa(addr.sin_addr));

printf("peer port: %d \n",ntohs(addr.sin_port));
}
```

**OUTPUT:**

```
cc 20.c

./a.out

peer IP address: 28.152.4.8

peer port: 29184
```

## 11.Implementation of Talker and Listener using Socket System Calls.

**SERVER**

```
#include<sys/socket.h>

#include<netinet/in.h>

#include<string.h>

#include<errno.h>

#include<stdlib.h>

short myport;

main(int argc,char *argv[])

{

int sockfd,newfd,nbyte,n,p;

struct sockaddr_in myaddr,claddr;

char buff[512] , buff1[512];

if(argc!=2)

{

perror("invalid no of arguments");

exit(1);

}

if((sockfd=socket(AF_INET,SOCK_DGRAM,0))==-1)

{

perror("unable to create socket" );

exit (1);

}

myport=atoi(argv[1]);
```

```
myaddr.sin_family=AF_INET;

myaddr.sin_port=htons(myport);

myaddr.sin_addr.s_addr=INADDR_ANY;

bzero(&(myaddr.sin_zero),8);

if((bind(sockfd,(struct sockaddr *)&myaddr,sizeof(myaddr)))== -1)

{

perror("bind error");

exit(1);

}

write(1,"\n waiting for Client",sizeof("\n Wait for Client"));

newfd=sizeof(claddr);

n=recvfrom(sockfd,buff1,sizeof(buff1),0,(struct sockaddr *)&claddr,&newfd);

if(n==-1)

{

perror("RECEIVE ERROR");

exit(1);

}

printf("\n CLIENT CONNECTED&..");

write(1,"\n Client :" ,10) ;

write(1,buff1,n);

// for(;;)

p=fork();

if(p<0)

{

perror("CHILD CANNOT BE CREATED&.");
exit(1);
}
if(p>0)
{
for(;;)
{
write(1,"\n Server :" ,10);
n=read(0,buff,512);
if((sendto(sockfd,buff,n,0,(struct sockaddr *)&claddr,newfd))==-1)
{
perror("SEND ERROR");
exit(1);
}
}
}
else
{
for(;;)
{
n=recvfrom(sockfd,buff1,sizeof(buff1),0,(struct sockaddr *)&claddr,&newfd);
write(1,"\rClient :",10);
write(1,buff1,n);
write(1,"\nServer : ",sizeof("\nServer:"));
}
}
```

```
}
```

**CLIENT:**

```
#include<stdio.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>
short myport,destport;
main(int argc,char *argv[])
{
int sockfd,newfd,nbyte,n,na,p,q;
struct sockaddr_in myaddr,servaddr;
char buff[512],buff1[512];
/*if(argc!=4)
{
perror("invalid no of arguments");
exit(1);
}*/
sockfd=socket(AF_INET,SOCK_DGRAM,0);
if(sockfd==-1)
{
perror("unable to create socket ");
exit (1);
}
```

```
myport=atoi(argv[1]);
myaddr.sin_family=AF_INET;
myaddr.sin_port=htons(myport);
myaddr.sin_addr.s_addr=inet_addr("192.168.0.5");
bzero(&(myaddr.sin_zero),8);
destport=atoi(argv[2]);
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(destport);
servaddr.sin_addr.s_addr=inet_addr("192.168.0.5");
bzero(&(myaddr.sin_zero),8);
q=bind(sockfd,(struct sockaddr *)&myaddr,sizeof(myaddr));
if(q==-1)
{
perror("bind error");
exit(1);
}
newfd=sizeof(servaddr);
write(1,"\n ENTER MESSAGE TO CONNECT TO THE SERVER&.. : ",57) ;
// for(;;)
p=fork();
if(p<0)
{
perror("CHILD CANNOT BE CREATED&.");
exit(1);
}
```

```
if(p>0)

{

for(;;)

{

write(1,"\n Client :" ,sizeof("\nClient : "));

na=read(0,buff,512);

if((sendto(sockfd,buff,na,0,(struct sockaddr *)&servaddr,newfd))==-1)

{

perror("SEND ERROR");

exit(1);

}

}

}

else

{

for(;;)

{

n=recvfrom(sockfd,buff1,sizeof(buff1),0,(struct sockaddr *)&servaddr,&newfd);

if(n==-1)

{

perror("RECEIVE ERROR");

exit(1);

}

write(1,"\rServer :",10);

write(1,buff1,n);
write(1,"\nClient : ",10);

}
```

**<u>OUTPUT</u>:**

**<u>Server execution</u>**

cc chats.c

./a.out **7723** waiting for Clie hello

**Client:** how are u

**Server:** were are u

**<u>Client execution</u>**

cc chatc.c

**./a.out  7731  7723 192.168.0.5**

ENTER MESSAGE TO CONNECT TO THE SERVER&:

Server : hello

Client : how are u

Server :were are u

**Note: -** 7723 (server’s port number)

7731 (client’s port number)

## 12.Implementation of Remote files Access using RPC.

**STEP 1:** vi file.x

```
program FILE_ACCESS{
version FILE_VERSION{
string getfile(string name)=1;
}=1;
}=0x31234567;
```

**STEP 2:**

```
$ rpcgen -a file.x
```

**STEP 3:** vi file_server.c

```
#include<stdio.h>
#include "file.h"
#include<rpc/rpc.h>
#define maxsize 500
FILE *fp;
char **
getfile_1_svc(char **argp, struct svc_req *rqstp)
{
static char * result;
static char buff[maxsize];
char temp[100];
char *file=*argp;

char *not_found="no such file bound";

printf("step 1");

fp=fopen(file,"r");

if(fp==NULL)

result=not_found;

else

{

printf("step2");

fgets(buff,sizeof(temp)-1,fp);

while(!feof(fp))

{

fgets(temp,sizeof(temp)-1,fp);

strcat(buff,temp);

}

result=&buff[0];

printf("step3");

}

return &result;

}
```

**<u>STEP 4:</u>**

```
cc file_server.c file_svc.c -o file_server

./file_server&

[2] 13479
```

**STEP 5: vi file_client.c**

```
#include<stdio.h>
#include "file.h"
#include<rpc/rpc.h>
void
file_access_1(char *host)
{
        CLIENT *clnt;
        char * *result_1;
        char * getfile_1_arg;
        char file[100];
#ifndef DEBUG
        clnt = clnt_create (host, FILE_ACCESS, FILE_VERSION, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }
#endif  /* DEBUG */
printf("enter a file name");
scanf("%s",&file);
getfile_1_arg=&file[0];
        result_1 = getfile_1(&getfile_1_arg, clnt);
        if (result_1 == (char **) NULL) {
                clnt_perror (clnt, "call failed");
        }
printf("the file is %s",*result_1);
#ifndef DEBUG
        clnt_destroy (clnt);
#endif   /* DEBUG */
}
int
main (int argc, char *argv[])
{
        char *host;

        if (argc < 2) {
                printf ("usage: %s server_host\n", argv[0]);
                exit (1);
        }
        host = argv[1];
        file_access_1 (host);
exit (0);
}
```

**<u>Step 6:</u>**

```
cc file_client.c file_clnt.c -o file_client

./file_client 192.168.0.5

enter a file name

file.x

the file is program FILE_ACCESS{

version FILE_VERSION{

string getfile(string name)=1;

}=1;

}=0x31234567;
```