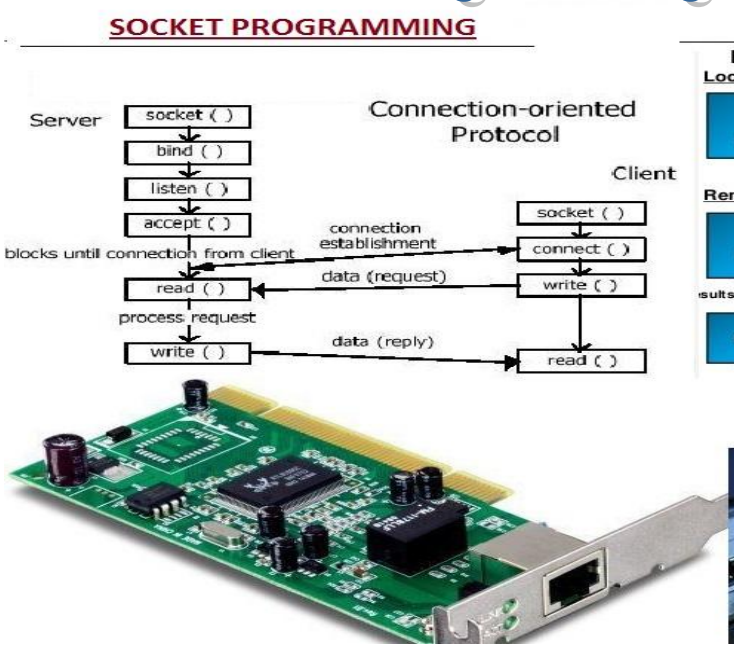
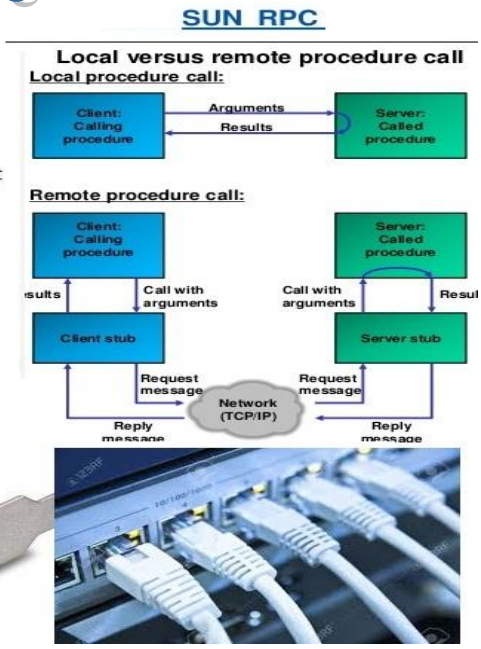
MUFFAKHAM JAH COLLEGE
OF
ENGINEERING AND TECHNOLOGY
(Affiliated Osmania University)
Banjara Hills, Hyderabad, Telangana State

INFORMATION TECHNOLOGY DEPARTMENT

Network Programming Lab Manual





1. Institution Vision

To be part of universal human quest for development and progress by contributing high calibre, ethical and socially responsible engineers who meet the global challenge of building modern society in harmony with nature.

2. Institution Mission

- To attain excellence in imparting technical education from the undergraduate through doctorate levels by adopting coherent and judiciously coordinated curricular and co-curricular programs
- To foster partnership with industry and government agencies through collaborative research and consultancy
- To nurture and strengthen auxiliary soft skills for overall development and improved employability in a multi-cultural work space
- To develop scientific temper and spirit of enquiry in order to harness the latent innovative talents
- To develop constructive attitude in students towards the task of nation building and empower them to become future leaders
- To nourish the entrepreneurial instincts of the students and hone their business acumen.
- To involve the students and the faculty in solving local community problems through economical and sustainable solutions.

3. Department vision

Fostering a bright technological future by enabling the students to function as leaders in software industry and serve as means of transformation to empower society through ITeS.

4. Department Mission

To create an ambience of academic excellence through state of art infrastructure and learner-centric pedagogy leading to employability in multi-disciplinary fields.

5. Program Educational Objectives

1. The Program Educational Objectives of Information Technology Program are as follows:
2. Graduates will demonstrate technical competence and leadership in their chosen fields of employment by identifying, formulating, analyzing and creating efficient IT solutions.
3. Graduates will communicate effectively as individuals or team members and be successful in varied working environment.
4. Graduates will demonstrate lifelong learning through continuing education and professional development.
5. Graduates will be successful in providing viable and sustainable solutions within societal, professional, environmental and ethical context.

6. Program Outcomes

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

7. Program Specific Outcomes

PSO1: Work as Software Engineers for providing solutions to real world problems using Structured, Object Oriented Programming languages and open source software.

PSO2: Function as Systems Engineer, Software Analyst and Tester for IT and ITeS.

8. INTRODUCTION TO NETWORK PROGRAMMING LAB

NETWORKING BASICS

Computer networking is the engineering discipline concerned with communication between computer systems or devices.

It is the practice of linking computing devices together with hardware and software that supports data communications across these devices.

KEY CONCEPTS AND TERMS

Packet A message or data unit that is transmitted between communicating processes.

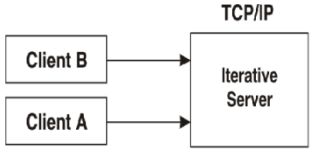
Host : A computer system that is accessed by a user working at a remote location. It is the remote process with which a process communicates. It may also be referred as Peer.

Channel: Communication path created by establishing a connection between endpoints.

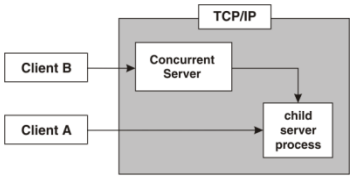
Network A group of two or more computer systems linked together

Server: In computer networking, a server is a computer designed to process requests and deliver data to other computers over a local network or the Internet.

- **Iterative servers**: This server knows ahead of time about how long it takes to handle each request & server process handles each request itself.



- **Concurrent servers:** The amount of work required to handle a request is unknown, so the server starts another process to handle each request.



Client: A Client is an application that runs on a personal computer or workstation and relies on a server to perform some operations.

Network Address: Network addresses give computers unique identities they can use to communicate with each other. Specifically, IP addresses and MAC addresses are used on most home and business networks.

Protocols: A Protocol is a convention or standard rules that enables and controls the connection, communication and data transfer between two computing endpoints.

Port An interface on a computer to which you can connect a device. It is a "logical connection place" and specifically, using the Internet's protocol, TCP/IP.

A port is a 16-bit number, used by the host-to-host protocol to identify to which higher-level protocol or application program (process) it must deliver incoming messages.

PORTS	**RANGE**
Well-known ports	1-1023
Ephemeral ports	1024-5000
User-defined ports	5001-65535

Connection: It defines the communication link between two processes.

Association: Association is used for 5 tuple that completely specifies the two processes that make up a connection.

{ Protocol, local-address, local-process, foreign-address, foreign- process}

The local address and foreign address specify the network ID & Host-ID of the local host and the foreign host in whatever format is specified by protocol suite.

The local process and foreign process are used to identify the specific processes on each system that are involved in a connection.

We also define Half association as either

{ ***protocol, local-address, local process}*** or **{** ***protocol, local-address, local process}***

which specify each half of a connection. This half association is called a Socket or transport address.

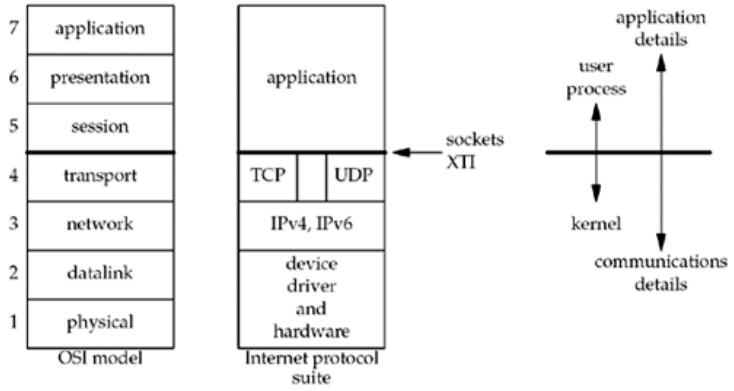
	Protocol	Local Address , Local Process	Foreign Address , Foreign Process
connection-oriented server	socket()	bind()	listen() accept()
connection-oriented client	socket()	connect()	
connectionless server	socket()	bind()	recvfrom()
connectionless client	socket()	bind()	sendto()

OSI Model

A common way to describe the layers in a network is to use the International Organization for Standardization (ISO) open systems interconnection (OSI) model for computer communications. This is a seven-layer model, which we show in Figure below along with the approximate mapping to the Internet protocol suite.

We consider the bottom two layers of the OSI model as the device driver and networking hardware that are supplied with the system. The network layer is handled by the IPv4 and IPv6 protocols. The transport layers that we can choose from are TCP and UDP

Layers in OSI model and Internet protocol suite.



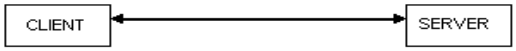
The upper three layers of the OSI model are combined into a single layer called the **application**. This is the Web client (browser) or whatever application we are using. With the Internet protocols, there is rarely any distinction between the upper three layers of the OSI model.

The sockets programming interfaces are interfaces from the upper three layers (the "application") into the transport layer. The sockets provide the interface from the upper three layers of the OSI model into the transport layer. There are two reasons for this design:

- The upper three layers handle all the details of the application and know little about the communication details. The lower four layers know little about the application, but handle all the communication details: sending data, waiting for acknowledgments, and so on.
- The second reason is that the upper three layers often form what is called a user process while the lower four layers are normally provided as part of the operating system (OS) kernel.

CLIENT-SERVER MODEL

Network applications can be divided into two process: a Client and a Server, with a communication link joining the two processes.

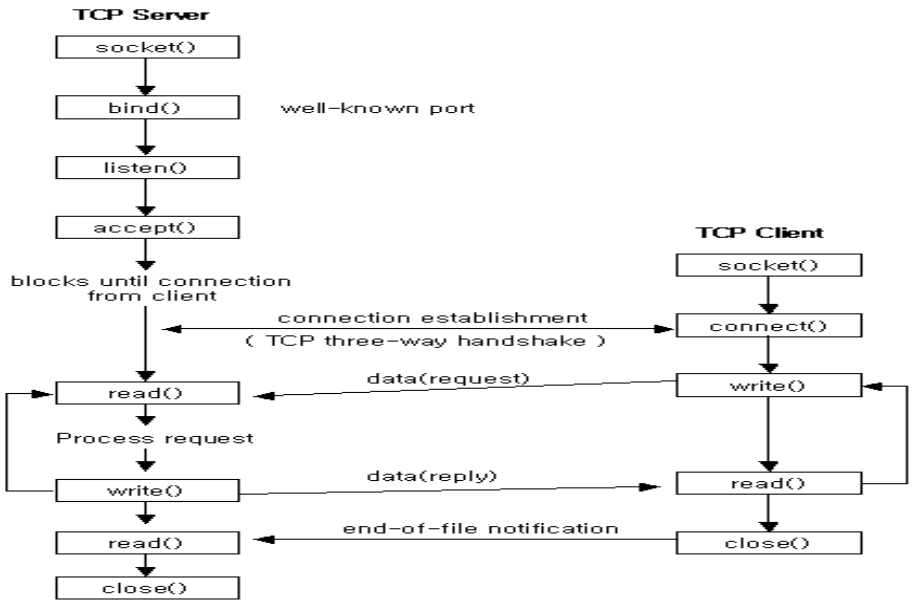


Normally, from Client-side it is one-one connection. From the Server Side, it is many-one connection.

The standard model for network applications is he Client-Sever model. A Server is a process that is waiting to be contacted by a Client process so that server can do something for the client.

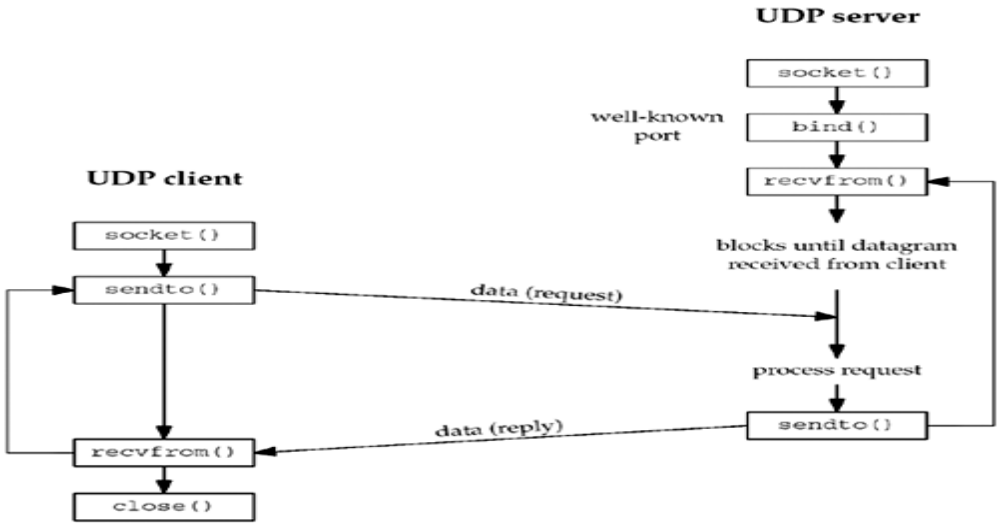
Typical BSD Sockets applications consist of two separate application level processes; one process (the **client**) requests a connection and the other process (the **server**) accepts it.

Socket functions for elementary TCP client/server in Connection-oriented Scenario



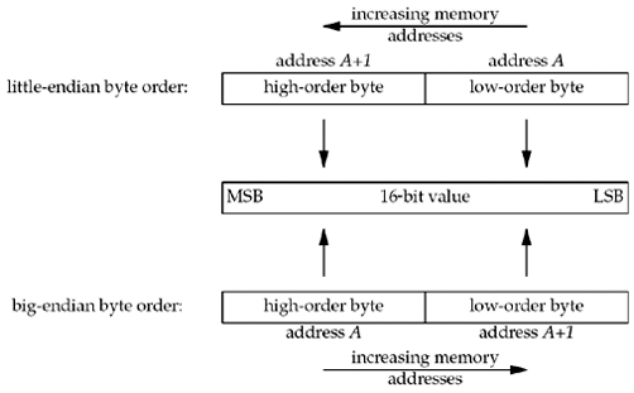
The server process creates a socket, binds an address to it, and sets up a mechanism (called a listen queue) for receiving connection requests. The client process creates a socket and requests a connection to the server process. Once the server process accepts a client process's request and establishes a connection, full-duplex (two-way) communication can occur between the two sockets.

Socket functions for elementary TCP client/server in Connection-less Scenario



Byte-Ordering Functions: 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.

Little-endian byte order and big-endian byte order for a 16-bit integer.



In this figure, we show increasing memory addresses going from right to left in the top, and from left to right in the bottom. We also show the most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit.

The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.

We refer to the byte ordering used by a given system as the *host byte order*. We must deal with these byte ordering differences as network programmers because networking protocols must specify a *network byte order*. Our concern is therefore converting between host byte order and network byte order. We use the following four functions to convert between these two byte orders.

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

unsigned long htonl(unsigned long hostlong);

unsigned short htons(unsigned short hostshort);

unsigned long ntohl(unsigned long netlong);

unsigned short ntohs(unsigned short netshort);
```

htons	host to network short
htonl	host to network long
ntohs	network to host short
ntohl	network to host long

Sockets Overview

The operating system includes the Berkeley Software Distribution (BSD) interprocess communication (IPC) facility known as *sockets*. Sockets are communication channels that enable unrelated processes to exchange data locally and across networks. A single socket is one end point of a two-way communication channel.

Sockets Overview: In the operating system, sockets have the following characteristics:

- A socket exists only as long as a process holds a descriptor referring to it.
- Sockets are referenced by file descriptors and have qualities similar to those of a character special device. Read, write, and select operations can be performed on sockets by using the appropriate subroutines.
- Sockets can be created in pairs, given names, or used to rendezvous with other sockets in a communication domain, accepting connections from these sockets or exchanging messages with them.

Sockets Background: Sockets were developed in response to the need for sophisticated interprocess facilities to meet the following goals:

- Provide access to communications networks such as the Internet.
- Enable communication between unrelated processes residing locally on a single host computer and residing remotely on multiple host machines.

Socket Facilities: Socket subroutines and network library subroutines provide the building blocks for IPC. An application program must perform the following basic functions to conduct IPC through the socket layer:

- Create and name sockets.
- Accept and make socket connections.
- Send and receive data.
- Shut down socket operations.

Socket Interface: The Socket interface provides a standard, well-documented approach to access kernel network resources.

Socket Header Files to be Included: Socket header files contain data definitions, structures, constants, macros, and options used by socket subroutines. An application program must include the appropriate header file to make use of structures or other information a particular socket subroutine requires. Commonly used socket header files are:

/usr/include/netinet/in.h	Defines Internet constants and structures.
/usr/include/netdb.h	Contains data definitions for socket subroutines.
/usr/include/sys/socket.h	Contains data definitions and socket structures.
/usr/include/sys/types.h	Contains data type definitions.
/usr/include/arpa.h	Contains definitions for internet operations.
/usr/include/sys/errno.h	Defines the **errno** values that are returned by drivers and other kernel-level code.

Internet address translation subroutines require the inclusion of the **inet.h** file. The **inet.h** file is located in the **/usr/include/arpa** directory.

Socket Addresses: Sockets can be named with an address so that processes can connect to them. Most socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of these structures begin with sockaddr_ and end with a unique suffix for each protocol suite.

Generic socket address structure: Many of the Networking system calls require a pointer to a socket address structure as an argument. Definition of this structure is in

#include<sys/socket.h>

```
struct sockaddr {
        unsigned short  sa_family;  /* address family : AF_xxx Value */
        char        sa_data[14]; /* up to 14 bytes of protocol-
                                        specific address */
    };
```

Internet Socket address structure: The protocol specific structure sockaddr_in is identical in size to generic structure which is 16 bytes.

#include <netinet/in.h>

```
struct sockaddr_in {
        short          sin_family;   /* AF_INET
        unsigned short   sin_port;    /* 16-bit port number */
                                            /* Network-byte ordered */
        struct in_addr   sin_addr;     /* 32-bit netid/hostid*/
                                            /* Network-byte ordered */
        char           sin_zero[8];  /* unused*/
   };

struct in_addr {
        unsigned long s_addr;  /* 32-bit netid/hostid */
                                    /* network byte ordered*/
   };
```

sin_zero is unused member, but we always set it to 0 when filling in one of these structures.

Socket address structures are used only on a given host: the structure itself is now communicated between different hosts, although certain fields (eg: IP Address & ports) are used for communication. *The protocol-specific structure **sockaddr_in** is identical in **size** to generic structure sockaddr which is **16 bytes**.

ELEMENTARY SOCKET SYSTEM CALLS

Socket() System Call: Creates an end point for communication and returns a descriptor.

Syntax

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

int socket ( int AddressFamily, int Type, int Protocol);
```

Description: The **socket** subroutine creates a socket in the specified *AddressFamily* and of the specified type. A protocol can be specified or assigned by the system. If the protocol is left unspecified (a value of 0), the system selects an appropriate protocol from those protocols in the address family that can be used to support the requested socket type.

The **socket** subroutine returns a descriptor (an integer) that can be used in later subroutines that operate on sockets.

Parameters

AddressFamily Specifies an address family with which addresses specified in later socket operations should be interpreted. Commonly used families are:

AF_UNIX
Denotes the Unix internal protocols

AF_INET
Denotes the Internet protocols.

AF_NS
Denotes the XEROX Network Systems protocol.

Type Specifies the semantics of communication. The operating system supports the following types:

SOCK_STREAM
Provides sequenced, two-way byte streams with a transmission mechanism for out-of-band data.

SOCK_DGRAM
Provides datagrams, which are connectionless messages of a fixed maximum length (usually short).

SOCK_RAW
Provides access to internal network protocols and interfaces. This type of socket is available only to the root user.

SOCK_SEQPACKET
Sequenced packet socket

Protocol Specifies a particular protocol to be used with the socket. Specifying the ***Protocol*** parameter of **0** causes the **socket** subroutine to select system's default for the combination of family and type.
IPROTO_TCP TCP Transport protocol
IPROTO_UDP UDP Transport protocol
IPROTO_SCTP SCTP Transport protocol

Return Values Upon successful completion, the **socket** subroutine returns an integer (the socket descriptor). It returns -1 on error.

Bind( ) System call: Binds a name to a socket.

Description: The **bind** subroutine assigns a *Name* parameter to an unnamed socket. It assigns a local protocol address to a socket.

Syntax

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

   int bind (int sockfd, struct sockaddr *myaddr, int addrlen);
```

sockfd is a socket descriptor returned by the `socket` function. The second argument is a pointer to a protocol specific address and third argument is size of this address structure.

There are 3 uses of bind:

a) Server registers their well-known address with a system. Both connection-oriented and connection-less servers need to do this before accepting client requests.
b) A Client can register a specific address for itself.
c) 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.

Return Values: Upon successful completion, the **bind** subroutine returns a value of 0. Otherwise, it returns a value of -1 to the calling program.

connect() System call:

The `connect` function is used by a TCP client to establish a connection with a TCP server.

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

   int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

sockfd is a socket descriptor returned by the `socket` function. The second and third arguments are a pointer to a socket address structure and its size. The socket address structure must contain the IP address and port number of the server.

Return Values: Upon successful completion, the **connect** subroutine returns a value of 0. Otherwise, it returns a value of -1 to the calling program.

listen() System call

This system call is used by a connection-oriented server to indicate that it is willing to receive connections.

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

        int listen (int sockfd, int backlog);
```

It is usually executed after both the socket and bind system calls, and immediately before accept system call. The *backlog* argument specifies how many connections requests can be queued by the system while it waits for the server to execute the accept system call.

Return values: Returns 0 if OK, -1 on error

accept() System call: The actual connection from some client process is waited for by having the server execute the accept system call.

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

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

accept takes the first connection request on the queue and creates another socket with the same properties as *sockfd*. If there are no connection requests pending, this call blocks the caller until one arrives. The cliaddr and addrlen arguments are used to return the protocol address of the connected peer process (the client). addrlen is called a value-result argument.

RETURN VALUES: This system call returns up to three values: an integer return code that is either a new socket descriptor or an error indication, the protocol address of the client process (through the cliaddr pointer), and the size of this address (through the addrlen pointer).

Send( ),sendto( ),recv( ) and recvfrom( ) system calls:

These system calls are similar to the standard `read` and `write` functions, but one additional argument is required.

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

int send(int sockfd, char *buff, int nbytes, int flags);

int sendto(int sockfd, char void *buff, int nbytes, int flags, struct sockaddr *to, int addrlen);

int recv(int sockfd, char *buff, int nbytes, int flags);

int recvfrom(int sockfd, char *buff, int nbytes, int flags, struct sockaddr *from, int *addrlen);
```

The first three arguments, *sockfd*, *buff* and *nbytes* are the same as the first three arguments to `read` and `write`. The flags argument is either 0 or is formed by logically OR'ing one or more of the constants.

MSG_OOB: Send or receive out-of-band data. This flag specifies that out-of-band data is being sent.

MSG_PEEK: Peek at incoming message (recv or recvfrom). This flag lets the caller look at the data that's available to be read, without having the system discard the data after recv or recvfrom returns.

`MSG_DONTROUTE`: This flag tells the kernel that the destination is on a locally attached network and not to perform a lookup of the routing table.

The to argument for `sendto` is a socket address structure containing the protocol address of where the data is to be sent. The size of this socket address structure is specified by addrlen. The `recvfrom` function fills in the socket address structure pointed to by from with the protocol address of who sent the datagram.

RETURN VALUES: All four system calls return the length of the data that was written or read as the value of the function. Otherwise it returns, -1 on error.

- The **network system calls** takes **two arguments**: the address of the generic sockaddr structure and the size of the protocol specific structure.

The caller must do is provide the address of protocl-specific structre as an argument, casting this pointer to a generic socket address structure.

From the kernel's perspective, another reason for using pointers to generic socket address structures as arguments is that the kernel must take the caller's pointer, cast it to a `struct sockaddr *`, and then look at the value of `sa_family` to determine the type of family.

Close( ) system call:

The normal Unix `close` function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>

int close (int sockfd);
```

VALUE RESULT-ARGUMENTS:

When a socket address structure is passed to any socket function, it is always passed by reference. That is, a pointer to the structure is passed. The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed: from the process to the kernel, or vice versa.

1. Three functions, `bind`, `connect`, and `sendto`, pass a socket address structure from the process to the kernel. One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure, as in

   ```
   struct sockaddr_in serv;
   /* fill in serv{} */
   connect (sockfd, (SA *) &serv, sizeof(serv));
   ```

   Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel. Figure shows this scenario.

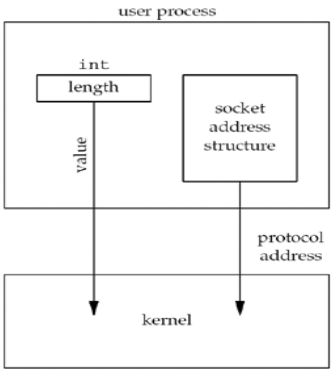


Figure: Socket address structure passed from process to kernel

2. Four functions, `accept`, `recvfrom`, `getsockname`, and `getpeername`, pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario. Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure, as in:

```
struct sockaddr_un  cli;   /* Unix domain */
  socklen_t  len;
  len = sizeof(cli);        /* len is a value */
  accept(unixfd, (SA *) &cli, &len);
  /* len may have changed */
```

The reason that the size changes from an integer to be a pointer to an integer is because the size is both a value when the function is called (it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in) and a result when the function returns (it tells the process how much information the kernel actually stored in the structure). This type of argument is called a value-result argument. Figure shows this scenario.

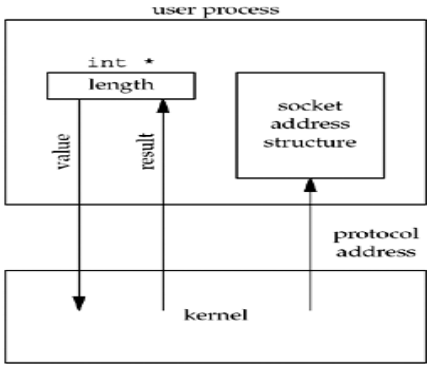


Figure. Socket address structure passed from kernel to process

9. Program 1: Familiarity with Lab environment and Client-Server model, Unix basic commands Understanding and using the following Networking Utility commands: Ifconfig, netstat, ping, arp, telnet, ftp, finger.

Program Objective:

Understanding and using of commands like ifconfig, netstat, ping, arp, telnet, ftp, finger, traceroute, whois

Program Description:

UNIX utilities are commands that, generally, perform a single task. It may be as simple as printing the date and time, or a complex as finding files that match many criteria throughout a directory hierarchy

IFCONFIG

The Unix command **`ifconfig`** (short for **i**nter**f**ace **config**urator) serves to configure and control TCP/IP network interfaces from a command line interface (CLI).

Common uses for `ifconfig` include setting an interface's IP address and netmask, and disabling or enabling a given interface.

NETSTAT

netstat (**net**work **stat**istics) is a command-line tool that displays network connections (both incoming and outgoing), routing tables, and a number of network interface statistics.

It is used for finding problems in the network and to determine the amount of traffic on the network as a performance measurement.

Parameters

Parameters used with this command must be prefixed with a hyphen (-) rather than a slash (/).

-a : Displays **a**ll active TCP connections and the TCP and UDP ports on which the computer is listening.

-e : Displays **e**thernet statistics, such as the number of bytes and packets sent and received. This parameter can be combined with **-s**.

-f : Displays **f**ully qualified domain names <FQDN> for foreign addresses.

-i : Displays network **i**nterfaces and their statistics (not available under Windows)

-n : Displays active TCP connections, however, addresses and port numbers are expressed numerically and no attempt is made to determine names.

-o : Displays active TCP connections and includes the process ID (PID) for each connection.

-p Linux: **P**rocess : Show which processes are using which sockets

PING

Ping is a computer network tool used to test whether a particular host is reachable across an IP network; it is also used to self test the network interface card of the computer, or as a speed test. It works by sending ICMP "echo request" packets to the target host and listening for ICMP "echo response" replies. Ping does not estimate the round-trip time, as it does not factor in the user's connection speed, but instead is used to record any packet loss, and print a statistical summary when finished.

The word *ping* is also frequently used as a verb or noun, where it is usually incorrectly used to refer to the round-trip time, or measuring the round-trip time.

ARP

In computer networking, the **Address Resolution Protocol** (**ARP**) is the method for finding a host's link layer (hardware) address when only its Internet Layer (IP) or some other Network Layer address is known.

ARP has been implemented in many types of networks; it is not an IP-only or Ethernet-only protocol. It can be used to resolve many different network layer protocol addresses to interface hardware addresses, although, due to the overwhelming prevalence of IPv4 and Ethernet, ARP is primarily used to translate IP addresses to Ethernet MAC addresses.

TELNET

Telnet (**Tel**ecommunication **net**work) is a network protocol used on the Internet or local area network (LAN) connections.

Typically, telnet provides access to a command-line interface on a remote machine.

The term *telnet* also refers to software which implements the client part of the protocol. Telnet clients are available for virtually all platforms

Protocol details:

Telnet is a client-server protocol, based on a reliable connection-oriented transport. Typically this protocol is used to establish a connection to TCP port 23

FTP

File Transfer Protocol (**FTP**):

FTP is a network protocol used to transfer data from one computer to another through a network such as the Internet.FTP is a file transfer protocol for exchanging and manipulating files over a TCP computer network. An FTP client may connect to an FTP server to manipulate files on that server.FTP runs over TCP. It defaults to listen on port 21 for incoming connections from FTP clients. A connection to this port from the FTP Client forms the control stream on which commands are passed from the FTP client to the FTP server and on occasion from the FTP server to the FTP client. FTP uses out-of-band control, which means it uses a separate connection for control and data. Thus, for the actual file transfer to take place, a different connection is required which is called the data stream.

FINGER:

In computer networking, the **Name/Finger protocol** and the **Finger user information protocol** are simple network protocols for the exchange of human-oriented status and user information.

TRACEROUTE:

traceroute is a computer network tool used to determine the route taken by packets across an IP network . An IPv6 variant, **traceroute6**, is also widely available.Traceroute is often used for network troubleshooting. By showing a list of routers traversed, it allows the user to identify the path taken to reach a particular destination on the network. This can help identify routing problems or firewalls that may be blocking access to a site. Traceroute is also used by penetration testers to gather information about network infrastructure and IP ranges around a given host. It can also be used when downloading data, and if there are multiple mirrors available for the same piece of data, one can trace each mirror to get a good idea of which mirror would be the fastest to use.

WHO IS:

WHOIS (pronounced "**who is**"; not an acronym) is a query/response protocol which is widely used for querying an official database in order to determine the owner of a domain name, an IP address, or an autonomous system number on the Internet. WHOIS lookups were traditionally made using a command line interface, but a number of simplified web-based tools now exist for looking up domain ownership details from different databases. WHOIS normally runs on TCP port 43.

The WHOIS system originated as a method that system administrators could use to look up information to contact other IP address or domain name administrators (almost like a "white pages").

10. Program 2: Socket Programming: Implementation of Connection-Oriented Service using standard ports.

i. Echo Service (7)
ii. Date and Time Service(13)
iii. Time of Day Service (37)
iv. Character generation(19)

Program Objective: Implementation of connection oriented service using standard port by the system calls .

Program Description The Standard port numbers are the port numbers that are reserved for assignment for use by the application end points that communicate using the Internet's Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). Each kind of application has a designated (and thus "well-known") port number.

In order to implement the standard ports we need to create an application for instance say client, which is going to invoke service which is established on the standard ports. The Client will be creating its socket endpoint and establish a connection with the standard server by specifiying the port number which has the defined service, for instance 7 for echo service.

STEPS:

Connection Oriented Implementation

Client

- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port of the Server, where it is providing service
- Establish connection to the Server using *connect()* system call.
- For echo server, send a message to the server to be echoed using *send()* system call.
- Receive the result of the request made to the server using *recv()* system call.

- Write the result thus obtained on the standard output.

Validation:

Sample Input: For port 37: Client sends an empty message

Sample Output: For **port 37**: Time elapsed since January 1st 1900 in seconds will be displayed.

”

11. Program 3: Implementation of Connection-Less Service using standard ports

i. Echo Service (7)

ii. Date and Time Service(13)

iii. Character generation(19)

Program Objective: Implementation of connectionless service using standard port by the system calls.

Program Description:

The Standard port numbers are the port numbers that are reserved for assignment for use by the application end points that communicate using the Internet's Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). Each kind of application has a designated (and thus "well-known") port number.

In order to implement the standard ports we need to create an application for instance say client, which is going to invoke service which is established on the standard ports. The Client will be creating its socket endpoint and establish a connection with the standard server by specifiying the port number which has the defined service, for instance 7 for echo service.

Connection less Implementation

Client:

- Include appropriate header files.
- Create a UDP Socket.
- Fill in the socket address structure (with server information)
- Specify the port of the Server, where it is providing service
- For echo server, send a message to the server to be echoed using *sendto()* system call.
- Receive the result of the request made to the server using *recvfrom()* system call.
- Write the result thus obtained on the standard output.

Validation:

Input: Client sends a message that will be echoed by the Server, say “Hello”.

Output: Server echoes the message back to the client i.e “Hello”

12. Program 4: Implementation of Connection-Oriented Iterative Echo-Server, date and time, character generation using user-defined ports

Program Objective Implementation of iterative echo server using both connection and connectionless socket system calls

Problem Definition: An iterative server knows ahead of time about how long it takes to handle each request & server process handles each request itself.

Problem Description: In order to implement the Iterative Service we need to create an application for instance say client, which will be invoking service which is established on the Iterative server working on a user-defined port. The Client will be creating its socket endpoint and establish a connection with the Iterative server by specifiying the port number similar to that of the Server

STEPS:

a) Connection Oriented Implementation

Server:

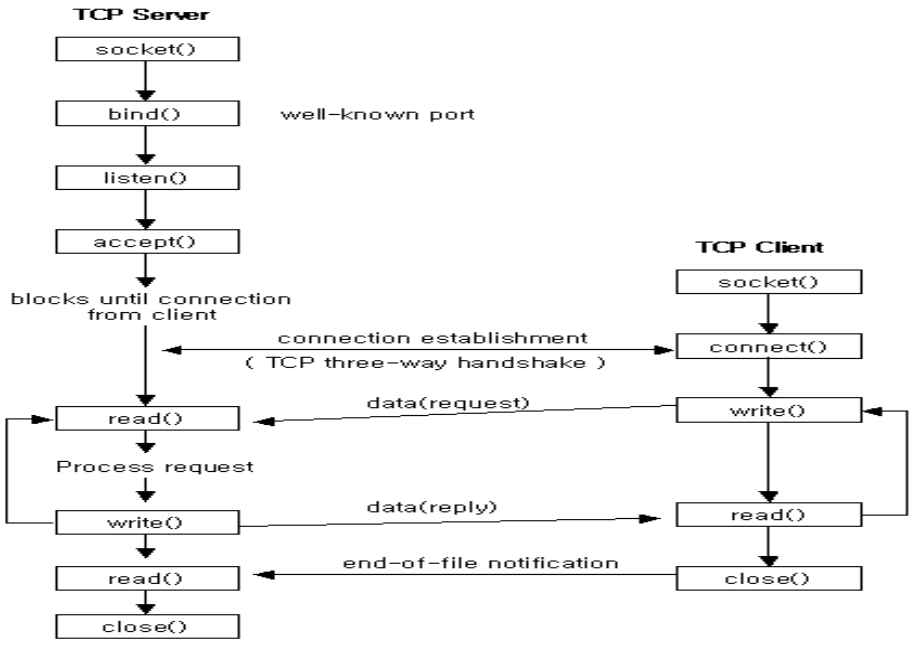
- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port where the service will be defined to be used by client.
- Bind the address and port using *bind()* system call.
- Server executes *listen()* system call to indicate its willingness to receive connections.
- Accept the next completed connection from the client process by using an *accept()* system call.
- Receive a message from the Client using *recv()* system call.
- Send the result of the request made by the client using *send()* system call.

Client

- Include appropriate header files.

- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port of the Server, where it is providing service
- Establish connection to the Server using *connect()* system call.
- For echo server, send a message to the server to be echoed using *send()* system call.
- Receive the result of the request made to the server using *recv()* system call.
- Write the result thus obtained on the standard output.

FLOW-CHART



Execution Procedure: Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj ⇨ $ ./obj& ⇨ $ cc client.c ⇨ $./a.out

Validation:

Sample Input: Client sends a message that will be echoed by the Server, say "Hello".

Sample Output: Server echoes the message back to the client i.e "Hello"

13. Program 5: Implementation of Connectionless Iterative Echo-server, date and time, character generation using user-defined ports.

Program Objective Implementation of iterative echo server using connectionless socket system calls

Problem Definition:

An iterative server knows ahead of time about how long it takes to handle each request & server process handles each request itself.

Problem Description:

In order to implement the Iterative Service we need to create an application for instance say client, which will be invoking service which is established on the Iterative server working on a user-defined port. The Client will be creating its socket endpoint and establish a connection with the Iterative server by specifying the port number similar to that of the Server

Connection less Implementation

Server:

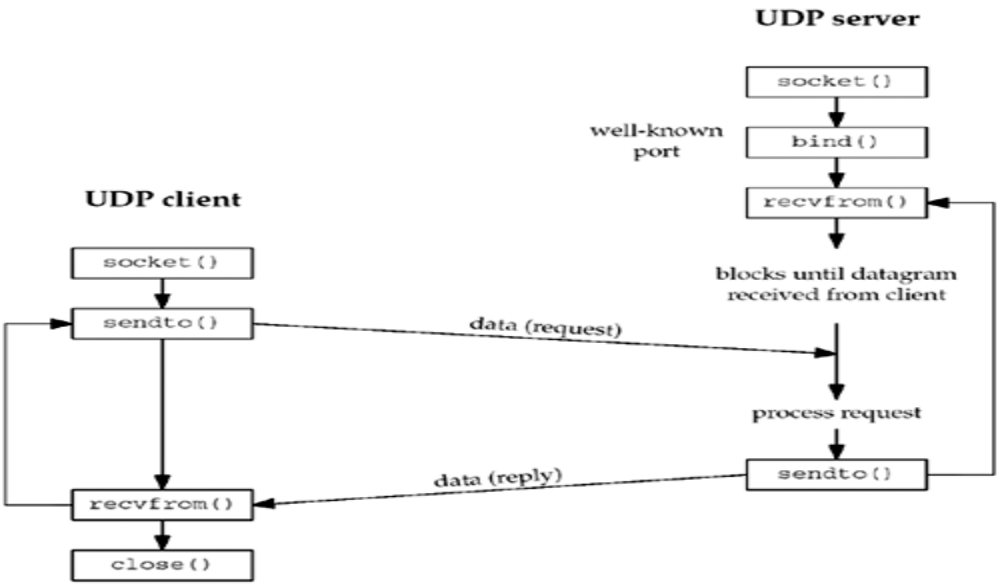
- Include appropriate header files.
- Create a UDP Socket.
- Fill in the socket address structure (with server information)
- Specify the port where the service will be defined to be used by client.
- Bind the address and port using *bind()* system call.
- Receive a message from the Client using *recvfrom()* system call.
- Send the result of the request made by the client using *sendto()* system call.

Client

- Include appropriate header files.
- Create a UDP Socket.
- Fill in the socket address structure (with server information)
- Specify the port of the Server, where it is providing service

- For echo server, send a message to the server to be echoed using *sendto()* system call.
- Receive the result of the request made to the server using *recvfrom()* system call.
- Write the result thus obtained on the standard output.

FLOW CHART



Execution Procedure:

Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj

$ ./obj&

$ cc client.c

$./a.out

Validation:

Sample Input: Client sends a message that will be echoed by the Server, say “Hello”.

Sample Output: Server echoes the message back to the client i.e “Hello”

14. Program 6: Implementation of Connection-Oriented Concurrent Echo-server, date and time, character generation using user-defined ports

Program Objective: Implementation of Concurrent echo server using both connection and connectionless socket system calls

Problem definition: The amount of work required to handle a request is unknown, so the server starts another process to handle each request.

Problem Description: In order to implement the Iterative Service we need to create an application for instance say client, which will be invoking service which is established on the Iterative server working on a user-defined port. The Client will be creating its socket endpoint and establish a connection with the Iterative server by specifiying the port number similar to that of the Server

STEPS:

a) Connection-Oriented Implementation:

Server:

- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port where the service will be defined to be used by client.
- Bind the address and port using *bind()* system call.
- Server executes *listen()* system call to indicate its willingness to receive connections.
- Accept the next completed connection from the client process by using an *accept()* system call.
- Create a new process (child process) using *fork()*, to handle the client request. The parent process will be waiting for new incoming connections.
- Receive a message from the Client using *recv()* system call.
- Send the result of the request made by the client using *send()* system call.

Client

- Include appropriate header files.

- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port of the Server, where it is providing service
- Establish connection to the Server using *connect()* system call.
- For echo server, send a message to the server to be echoed using *send()* system call.
- Receive the result of the request made to the server using *recv()* system call.
- Write the result thus obtained on the standard output.

a) Connection-less Implementation:

Server:

- Include appropriate header files.
- Create a UDP Socket.
- Fill in the socket address structure (with server information)
- Specify the port where the service will be defined to be used by client.
- Bind the address and port using *bind()* system call.
- Create a new process (child process) using *fork()*, to handle the client request. The parent process will be waiting for new incoming connections.
- Receive a message from the Client using *recvfrom()* system call.
- Send the result of the request made by the client using *sendto()* system call.

Client

- Include appropriate header files.
- Create a UDP Socket.
- Fill in the socket address structure (with server information)
- Specify the port of the Server, where it is providing service
- For echo server, send a message to the server to be echoed using *sendto()* system call.
- Receive the result of the request made to the server using *recvfrom()* system call.
- Write the result thus obtained on the standard output.

Execution Procedure:
Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj ⇨ $ ./obj&$ ⇨ cc client.c ⇨ $./a.out

Validation:

Sample Input: Client sends a message that will be echoed by the Server, say "Hello"

.**Sample Output:** Server echoes the message back to the client i.e "Hello"

15. Program 7: Program for connection-oriented Iterative Service in which server reverses the string sent by the client and sends it back

Problem Description: The problem can be implemented using sockets. General implementation steps are as follows:

Steps involved in writing the Server Process:

1. Create a socket using *socket( ) system call.*.
2. Bind server's address and port using *bind( ) system call.*
3. Convert the socket into a listening socket using *listen( ) sytem call.*
4. Wait for client connection to complete using *accept( ) system call.*
5. Receive the Client request using recv() system call which consist of the name of the command that is to be executed along with data parameters(if any)
6. The command is interpreted and executed.
7. On successful execution the result is passed back to the client by the server

Steps involved in writing the Client Process:

1.Create a socket.

2.Fill in the internet socket address structure (with server information).

3.Connect to server using *connect system call*.

4.The client passes the command and data parameters (if any) to the server.

5.Read the result sent by the server, write it to standard output.

6.Close the socket connection.

Execution Procedure:

Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj ⟹ $ ./obj& ⟹ $ cc client.c ⟹$./a.out

Validation:

Sample Input:

The Client sends the string "NPLAB"

Sample Output

The string will get back as reverse "BALNP"

16. Program 8: Program for connection-oriented Iterative service in which server changes the case of the strings sent by the client and sends back (Case Server).

Problem Description: The problem can be implemented using sockets. General implementation steps are as follows:

Steps involved in writing the Server Process:

1. Create a socket using *socket( ) system call.*.
2. Bind server's address and port using *bind( ) system call.*
3. Convert the socket into a listening socket using *listen( ) sytem call.*
4. Wait for client connection to complete using *accept( ) system call.*
5. Receive the Client request using recv() system call which consist of the name of the command that is to be executed along with data parameters(if any)
6. The command is interpreted and executed.
7. On successful execution the result is passed back to the client by the server

Steps involved in writing the Client Process:

1. Create a socket.
2. Fill in the internet socket address structure (with server information).
3. Connect to server using *connect system call*.
4. The client passes the command and data parameters (if any) to the server.
5. Read the result sent by the server, write it to standard output.
6. Close the socket connection.

Execution Procedure:

Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj ⇨ $ ./obj& ⇨ $ cc client.c ⇨$./a.out

Validation:

Sample Input:

The Client sends the string "HEllo"

Sample Output: The string will get back as "heLLO"

17. Program 9: Program for Connection-Oriented Iterative service in which server calculates the Net-salary of an Employee based on the following details sent by the client
i)basic-sal ii) hra iii) da iv) pt v) epf (net-sala=basic+hra+da-pt-epf).

Problem Description: The problem can be implemented using sockets. General implementation steps are as follows:

Steps involved in writing the Server Process:

1. Create a socket using *socket( ) system call.*.
2. Bind server's address and port using *bind( ) system call.*
3. Convert the socket into a listening socket using *listen( ) sytem call.*
4. Wait for client connection to complete using *accept( ) system call.*
5. Receive the Client request using recv() system call which consist of the name of the command that is to be executed along with data parameters(if any)
6. The command is interpreted and executed.
7. On successful execution the result is passed back to the client by the server

Steps involved in writing the Client Process:

1. Create a socket.
2. Fill in the internet socket address structure (with server information).
3. Connect to server using *connect system call*.
4. The client passes the command and data parameters (if any) to the server.
5. Read the result sent by the server, write it to standard output.
6. Close the socket connection.

Execution Procedure: Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj ⇨ $ ./obj& ⇨ $ cc client.c ⇨ $./a.out

Validation:

Sample Input: The Client sends the salary details 1000 2000 3000 500 500 100

Sample Output

Then it will return the complete salaray after calculation 4900

18. Program 10: Program for file access using sockets.

Problem Description: The problem can be implemented using sockets. General implementation steps are as follows:

Steps involved in writing the Server Process:

1. Create a socket using *socket( ) system call.*.
2. Bind server's address and port using *bind( ) system call.*
3. Convert the socket into a listening socket using *listen( ) sytem call.*
4. Wait for client connection to complete using *accept( ) system call.*
5. Receive the Client request using recv() system call which consist of the name of the command that is to be executed along with data parameters(if any)
6. The command is interpreted and executed.
7. On successful execution the result is passed back to the client by the server

Steps involved in writing the Client Process:

1. Create a socket.
2. Fill in the internet socket address structure (with server information).
3. Connect to server using *connect system call*.
4. The client passes the command and data parameters (if any) to the server.
5. Read the result sent by the server, write it to standard output.
6. Close the socket connection.

Execution Procedure:

Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj ⇨ $ ./obj& ⇨ $ cc client.c ⇨ $./a.out

Validation:

Sample Input: The Client sends a file which is present in current directory "sample.c"

Sample Output

Then it will return the content of the file "welcome to Nplab"

19. Program 11: Program for Remote Command Execution using sockets

Problem Definition - Remote command execution is when a process on a host causes a program to be executed on another host. Usually the invoking process wants to pass data to the remote program capture its output also.

Problem Description: The problem can be implemented using sockets. General implementation steps are as follows:

Steps involved in writing the Server Process:

1. Create a socket using *socket( ) system call.*.
2. Bind server's address and port using *bind( ) system call.*
3. Convert the socket into a listening socket using *listen( ) sytem call.*
4. Wait for client connection to complete using *accept( ) system call.*
5. Receive the Client request using recv() system call which consist of the name of the command that is to be executed along with data parameters(if any)
6. The command is interpreted and executed.
7. On successful execution the result is passed back to the client by the server

Steps involved in writing the Client Process:

1. Create a socket.
2. Fill in the internet socket address structure (with server information).
3. Connect to server using *connect system call*.
4. The client passes the command and data parameters (if any) to the server.
5. Read the result sent by the server, write it to standard output.
6. Close the socket connection.

Execution Procedure:

Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj ⟹ $ ./obj& ⟹ $ cc client.c ⟹ $./a.out

Validation:

Sample Input: The Client sends the name of the command to be executed, for instance pwd

Sample Output: /home/guest/it07001/networks

20. Program 12: Implementation of DNS

Problem Definition: The **Domain Name System** (**DNS**) is a hierarchical naming system for computers, services, or any resource participating in the Internet. The Domain Name System distributes the responsibility of assigning domain names and mapping those names to IP addresses

Problem Description:

The Client program sends a request containing domain-name to the server

STEPS:

Server:

- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port where the service will be defined to be used by client.
- Bind the address and port using *bind()* system call.
- Server executes *listen()* system call to indicate its willingness to receive connections.
- Accept the next completed connection from the client process by using an *accept()* system call.
- Receive the request from the Client using *recv()* system call.
- For the domain-name thus received from the Client, obtain the corresponding IP address using appropriate logic.
- Send the result (in the buffer) of the request made by the client using *send()* system call.

Client

- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port of the Server, where it is providing service
- To obtain the IP address for the domain name. Send request to the server consisting of the domain-name using *send()* system call.
- Receive the result of the request made to the server using *recv()* system call.

- Write the result thus obtained on the standard output.

Execution Procedure:

Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj

$ ./obj&

$ cc client.c

$./a.out

Validation:

Sample Input:

Client sends Domain-name asking for the IP address

For Example: mjcet.it.edu

Sample Output:

Server replies back with the IP Address that corresponds to the domain name.

The corresponding IP Address will be generated, for ex: 192.100.100.6

21. Program 13: Program to implement Web Server using sockets

Program Objective : Build Web Server using Sockets

Problem Definition: The Client will be requesting Web page to be accessed which resides at the Server side.

Problem Description:

A Web Page is the page which contains html tags that can be executed in a browser. An application process i.e Client will be requesting access to a web page to the Web Server. On getting such a request, Server will be responding with the page requested.

STEPS:

Server:

- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Bind the address and port using *bind()* system call.
- Server executes *listen()* system call to indicate its willingness to receive connections.
- Accept the next completed connection from the client process by using an *accept()* system call.
- Receive a message from the Client using *recv()* system call. The message will be actually the name of the web page requested by the client.
- Send the result of the request made by the client using *send()* system call.

Client

- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Establish connection to the Server using *connect()* system call.
- Make a request to web server with the web page that is residing at the server side and send a message containing this request using *send()* system call.
- Receive the result of the request made to the server using *recv()* system call.
- Write the result thus obtained on the standard output.

Validation:

Sample Input:

Enter the filename from the list:

web1.html

web2.html

web3.html

web4.html

Sampe Output:

<html>

<title> WEB1</title>

<body> This is WEB1.html </body>

</html>

22. Program 14: Advanced Socket System Calls : Programs to demonstrate the usage of Advanced socket system calls like getsockopt( ),setsockopt( ),getpeername ( ),getsockname( ),readv( ) and writev( ).

Problem Definition: Getting the various details associated with the socket by setting appropriate arguments in the Advanced socket system calls.

Problem Description:

getpeername - get the name of the peer socket

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

int getpeername(int socket, struct sockaddr *address,
                socklen_t *address_len);
```

The *getpeername()* function retrieves the peer address of the specified socket, stores this address in the **sockaddr** structure pointed to by the *address* argument, and stores the length of this address in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address will be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by *address* is unspecified.

getsockname - get the socket name

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

int getsockname(int socket, struct sockaddr *address,
                socklen_t *address_len);
```

The *getsockname()* function retrieves the locally-bound name of the specified socket, stores this address in the **sockaddr** structure pointed to by the *address* argument, and stores the length of this address in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address will be truncated.

If the socket has not been bound to a local name, the value stored in the object pointed to by *address* is unspecified.

STEPS:

- Include the header files
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Bind the Address and port using *bind()* system call.
- If **Socket name** is to be retrieved, then include *getsockname()* system call with appropriate options set. If **Peer address** of the specified socket is to be retrieved, then include *getpeername()* system call with appropriate options set.

Write the options thus obtained to the standard output.

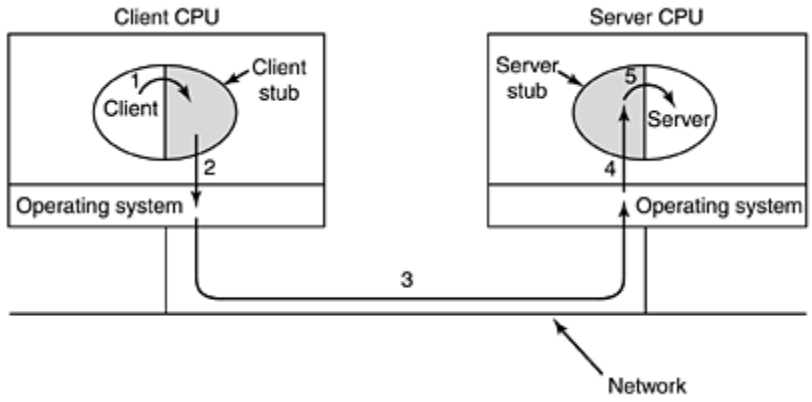
23. Program 15: Implementation of File access using RPC.

Program Objective : A file at the server is to be accessed using Remote Procedure calls.

Problem Description: - High-level programming through remote procedure calls (RPC) provides logical client-to-server communication for network application development - without the need to program most of the interface to the underlying network. With RPC, the client makes a ***remote procedure call*** that sends requests to the server, which calls a dispatch routine, performs the requested service, and sends back a reply before the returns to the client.

RPC does not require the caller to know about the underlying network (it looks similar to a call to a C routine).

RPC model - Figure below illustrates the basic form of network communication with the RPC (synchronous model).



The local function call model works as follows:

1. The caller places arguments to a procedure in a specific location(such as a result register).
2. The caller temporarily transfers control to the procedure.
3. When the caller gains control again, it obtains the results of the procedure from the specified location.
4. The caller then continues program.

The RPC is similar, in that one thread of control logically winds through two processes – that of the caller and that of the server.

1. The caller process sends a call message to the server process and blocks for a reply. The **call message** contains the parameters of the procedure and the **reply message** contains the procedure results.
2. When the caller receives the reply message, it gets the results of the procedure.
3. The caller process then continues executing.

On the **server side** a process is dormant – awaiting the arrival of a call message. When one arrives, the server process computes a reply that is then sent back to the client. After this the server again becomes dormant.

Writing RPC applications with the *rpcgen* protocol compiler - It accept a remote program interface definition written in RPC language (which is similar to C). It then produces C language output consisting Skeleton versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, a header file that contains common definitions, and optionally dispatch tables that the server uses to invoke routines that are based on authorization checks.

The **client skeleton** interface to the RPC library hides the network from its callers, and the **server skeleton** hides the network from the server procedures invoked by remote clients.

Header file to include: <rpc/rpc.h>

The client handle is created in the client program that is generated by rpcgen compiler. The RPC is called through the client. This request passes over the network and reaches server side wherein server stub calls this procedure. The RPC returns the contents of file to the server stub and it travels through the network to the client stub and back to client where it is displayed.

STEPS:

- Create the Specification file, a file with .x extension.
- Compile it using rpcgen compiler which creates the stubs and the client and server programs

Client:

- Specify the RPC header client.
- Using this Create Client handle and invoke a protocol (UDP).
- Call the remote procedure using the handle.
- On return ,display the file contents
- Destroy the client handle
- Stop

Server:

- Declare the static variables for result.
- Open the file for which request came from client
- Read its contents into a buffer.
- Return the buffer as result

Execution Procedure:

- rcpgen compiles the specification file for instance, file.x and generates client stub, server stub, client program and server program.

```
$ rpcgen –a file.x
$ ls
file_client.c   file.h   file_svc.c   Makefile.x file_clnt.c   file_server.c   file.x
```

- After Stub and programs are generated, Compile the server program and server stub. Similarly do for the Client program and Client.

```
$ cc file_server.c file_svc.c –o obj
$ ./obj&
[2]5030
$ cc file_client.c file_clnt.c
$ ./a.out 192.100.100.6
```

Validation:

Sample input:

The client requests access to a file and enters that filename for ex, file1

Sample Output:

This is file access

24. Program 16: Build a concurrent multithreaded file transfer server using threads

Program Objective: Getting the various details associated with the socket by setting appropriate arguments in the Advanced socket system calls.

Program Description:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void*), void *arg);
```

Description

The *pthread_create()* function is used to create a new thread, with attributes specified by *attr*, within a process. Upon successful completion, *pthread_create()* stores the ID of the created thread in the location referenced by *thread*.

The thread is created executing *start_routine* with *arg* as its sole argument.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Description

The *pthread_join()* function suspends execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated. When a *pthread_join()* returns successfully, the target thread has been terminated.

STEPS

Server:

- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port where the service will be defined to be used by client.
- Bind the address and port using *bind()* system call.
- Server executes *listen()* system call to indicate its willingness to receive connections.

- Accept the next completed connection from the client process by using an *accept()* system call.
- Receive the request from the Client using *recv()* system call.
- To transfer the file to the Client, create a subroutine implementing the logic of file transfer using pthread_create () function.
- Send the result of the request made by the client using *send()* system call.
- Close the Socket.

Client

- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Specify the port of the Server, where it is providing service
- Send the name of the file to be transferred, which is residing at the server using *send()* system call to the Server.
- Receive the result of the request made to the server using *recv()* system call.
- Write the result thus obtained on the standard output.

Execution Procedure:

Suppose, the server program is server.c and client program is client.c

First compile the Server program as,

$ cc server.c – o obj

$ ./obj&

$ cc client.c

$./a.out

Validation:

Sample Input:

Client sends file name. For ex: File1

Sample Output

Server transfers the requested page to the Client

25. Program 17: Implementation of concurrent chat server that allows current logged in users to communicate one with other

Program Objective: Determine the number of Users currently logged in and establish chat session with them.

Program Description:

The command that counts the number of users logged in is who |wc –l. Using this command, determine the number of users currently available for chat.

Steps

Server:

- Include appropriate header files.
- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Bind the address and port using *bind()* system call.
- Server executes *listen()* system call to indicate its willingness to receive connections.
- Accept the next completed connection from the client process by using an *accept()* system call.
- Receive a message from the Client using *recv()* system call.
- Send the reply of the message made by the client using *send()* system call.

Client

- Create a TCP Socket.
- Fill in the socket address structure (with server information)
- Establish connection to the Server using *connect()* system call.
- Send a chat message to the Server using *send()* system call.
- Receive the reply message made to the server using *recv()* system call.
- Write the result thus obtained on the standard output.

26. Program 18: Demonstration of Non-Blocking I/O

Program Objective: Generally, whenever an I/O operation that is requested on a socket cannot be completed without putting the process to sleep, we refer this to be blocking operation. To avoid this behavior we set the socket to be Non-blocking so that whenever when a requested operation cannot be completed an error is returned rather than getting blocked.

Problem Definition: Non-Blocking I/O allows the process to tell the kernel to notify it when a specified descriptor ready for I/O. It is called Signal-driven I/O. The notification from the kernel to the user process takes place with a signal, the SIGIO signal.

Problem Description:

Non-Blocking I/O allows the process to tell the kernel to notify it when a specified descriptor ready for I/O. It is called Signal-driven I/O. The notification from the kernel to the user process takes place with a signal, the SIGIO signal.

STEPS:

- The process must establish a handler for the SIGIO Signal. This is done by calling the signal system call.
- The process must set the process ID or the process group ID to receive the SIGIO Signals. This is done with the fcntl system call, with the F_SETOWN command.
- The process must enable asynchronous I/O using the fcntl system call, with the F_SETFL command and the FASYNC argument.

Sampe Output

I can do some other work till server respond through async signal
I can do some other work till server respond through async signal
I can do some other work till server respond through async signal
I can do some other work till server respond through async signal
I can do some other work till server respond through async signal
Msg echoed from server
--this is an nonblocking I/O--

27. Program 19: Implementation of Ping service

Program Objective :

Implementation of Ping service to check whether the server is alive or not.

Program Description :

PING stands for Packet InterNet Groper. It is often used to test the reachability of another site on the internet. The program sends and ICMP echo request message to a specified host and waits for a reply.

Figure below shows format of an **ICMP message**.

IP header	ICMP header	ICMP data

Icmp message.

The echo request and echo reply messages are only two of the 13 currently defined ICMP messages. The ICMP header structure which is defined in <netinet/ip_icmp.h> is as follows :

```
struct icmp {
u_char icmp_type; // type of message
u_char icmp_code; // type of sub code
u_short icmp_cksum; // ones complement cksum of struct
u_short icmp_id; // identifier
u_short icmp_seq; // sequence number;
char icmp_data[1]; // start of optional data
};
```

Steps:

- Include the header files
- Define the proto structure for Ipv4.
- Set the length of optional data
- Handle command line options
- Process host argument
- Create a *raw* socket
- Set socket receive buffer size
- Send first packet
- Infinite loop reading all ICMP messages
- Get pointer to ICMP header
- Check for ICMP echo reply
- Print all received ICMP messages
- Stop

ANNEXURE– I: Network Programming Laboratory – OU Syllabus

With effect from the Academic Year 2014-2015

BIT 382

NETWORK PROGRAMMING LAB

Instruction	3 Periods per week
Duration	3 Hours
University Examination	50 Marks
Sessional	25 Marks

1. Understanding and using of commands like ifconfig, netstst, ping, arp, telnet, ftp, finger, traceroute, whois etc.
2. Implementation of concurrent and iterative echo server using both connection and connectionless socket system calls.
3. Implementation of time and day time services using connection oriented socket system calls.
4. Implementation of ping service
5. Build a web server using sockets.
6. Implementation of remote command execution using socket system calls.
7. Demonstrate the use of advanced socket system calls.
8. Demonstrate the non blocking I/O.
9. Implementation of concurrent chart server that allows current logged in users to communicate one with other.
10. Implementation of file access using RPC.
11. Build a concurrent multithreaded file transfer server using threads.
12. Implementation of DNS.

Suggested Reading:

1.Douglas E.Comer,Hands-on Networking with Internet Technologies,Pearson Education.
2. W. Richard Stevens, Unix Network Programming, Prentice Hall/Pearson Education,2009.