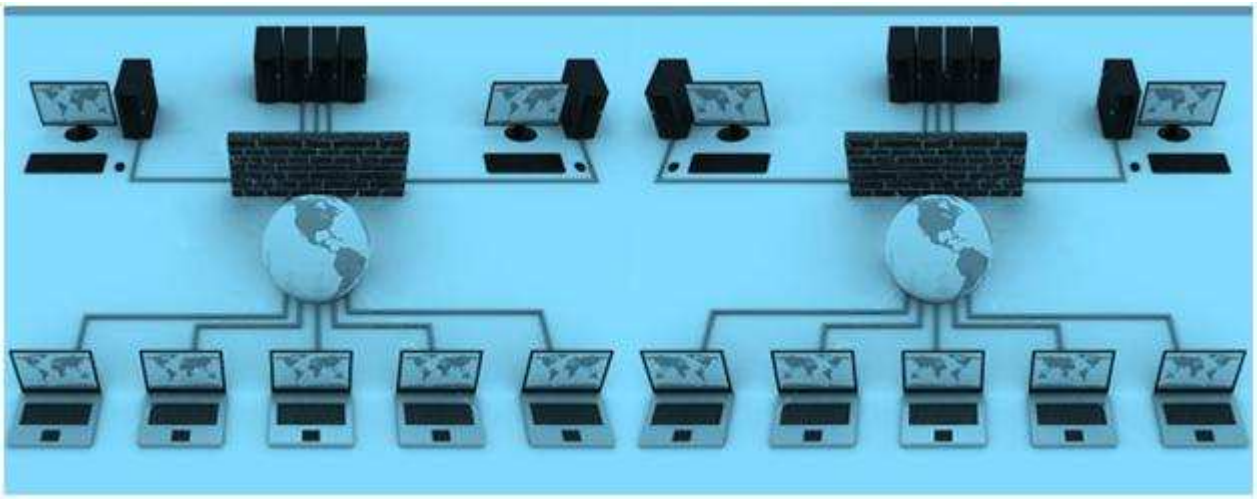
MUFFAKHAM JAH COLLEGE OF ENGINEERING AND TECHNOLOGY

Banjara Hills, Hyderabad, Telangana

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Network Programming Laboratory Manual



Academic Year 2016-2017

Table of Contents

I Contents

II Programs

Part I

Contents

1. Vision of the Institution

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. Mission of the Institution

- To attain excellence in imparting technical education from 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

To contribute competent computer science professionals to the global talent pool to meet the constantly evolving societal needs.

4. Department Mission

Mentoring students towards a successful professional career in a global environment through quality education and soft skills in order to meet the evolving societal needs.

5. Programme Education Objectives

1. Graduates will demonstrate technical skills and leadership in their chosen fields of employment by solving real time problems using current techniques and tools.

2. Graduates will communicate effectively as individuals or team members and be successful in the local and global cross cultural working environment.

3. Graduates will demonstrate lifelong learning through continuing education and professional development.

4. Graduates will be successful in providing viable and sustainable solutions within societal, professional, environmental and ethical contexts

6. Programme Outcomes

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **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.
4. **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.
5. **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.
6. **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.
7. **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.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **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.
11. **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.
12. **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

7. Programme Specific Outcomes

The graduates will be able to:

PSO1: Demonstrate understanding of the principles and working of the hardware and software aspects of computer systems.

PSO2: Use professional engineering practices, strategies and tactics for the development, operation and maintenance of software

PSO3: Provide effective and efficient real time solutions using acquired knowledge in various domains.

8. Inrtoduction to Network Basics

Introduction to Networking Basics

Laboratory Objective

Upon successful completion of this Lab the student will be able to:

- To understand the use of client/server architecture in application development.
- To understand and use elementary socket system calls and advanced socket system calls.
- To understand how to use TCP and UDP based sockets.

Computer Network: 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:

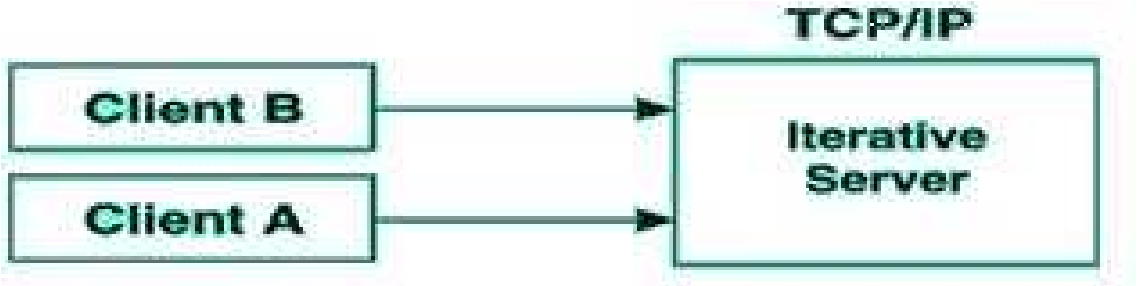
A group of two or more computer systems linked together

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.

1. **Iterative servers:**

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



2. **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 Addresses:

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.

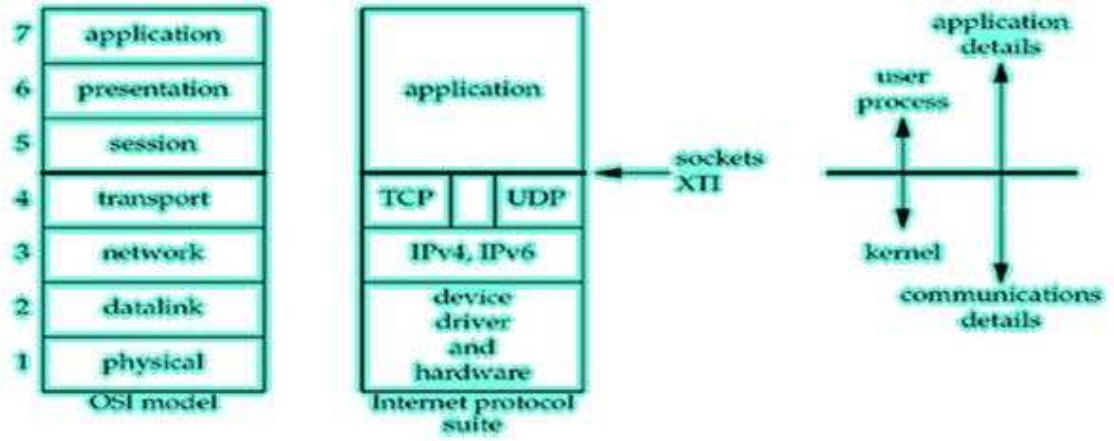
	Protocol	Local Address , Local Process	Foreign Address , Foreign Process
connection-oriented server	socket()	bind()	listen() accept()
connection-oriented client	socket()	connect()	
connectionless server	socket()	bind()	recvfrom()
connectionless client	socket()	bind()	sendto()

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.

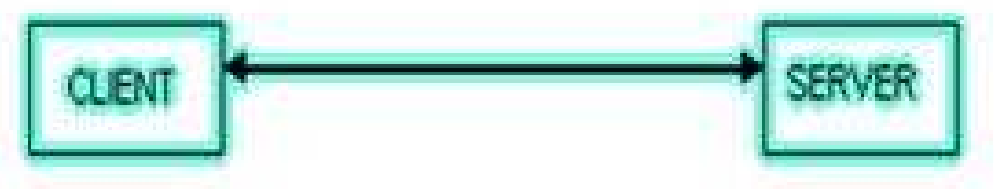
The sockets programming interfaces are interfaces from the upper three layers (the "application") into the transport layer. The sockets provide the interface from the upper three layers of the OSI model into the transport layer.

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

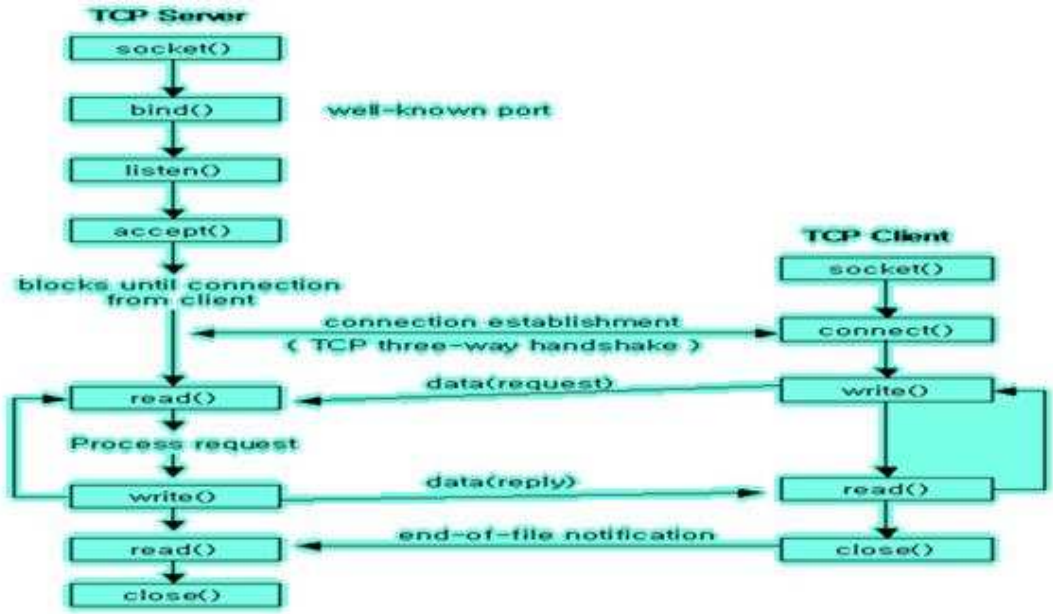
Network applications can be divided into two process: a Client and a Server, with a communication link joining the two processes.



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

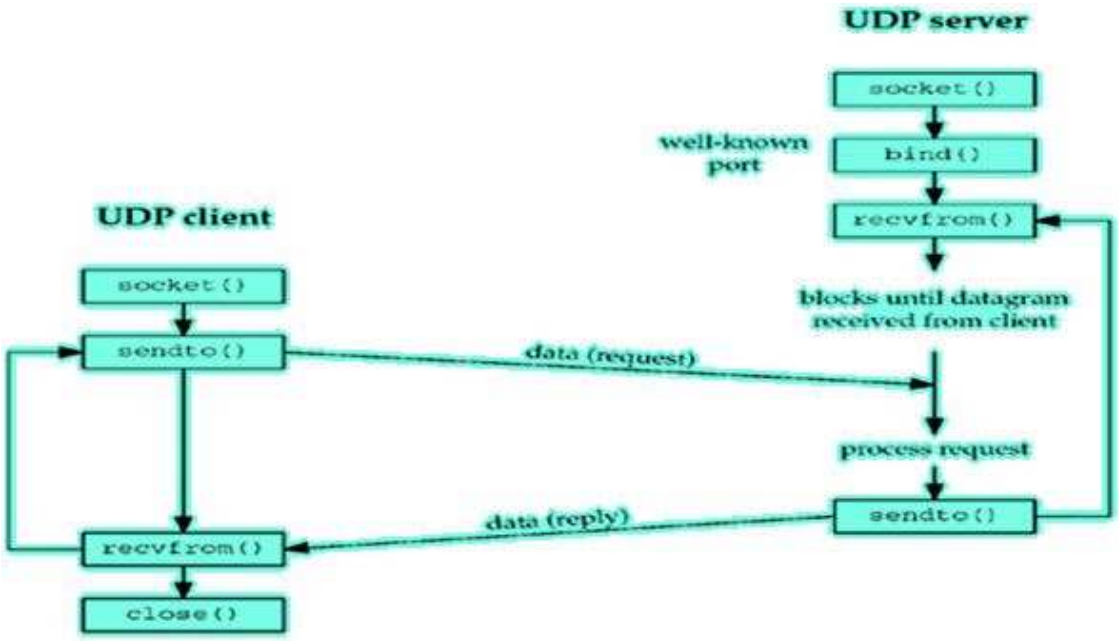
The standard model for network applications is he Client-Sever model. A Server is a process that is waiting to be contacted by a Client process so that server can do something for the client. Typical BSD Sockets applications consist of two separate application level processes; one process (the client) 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

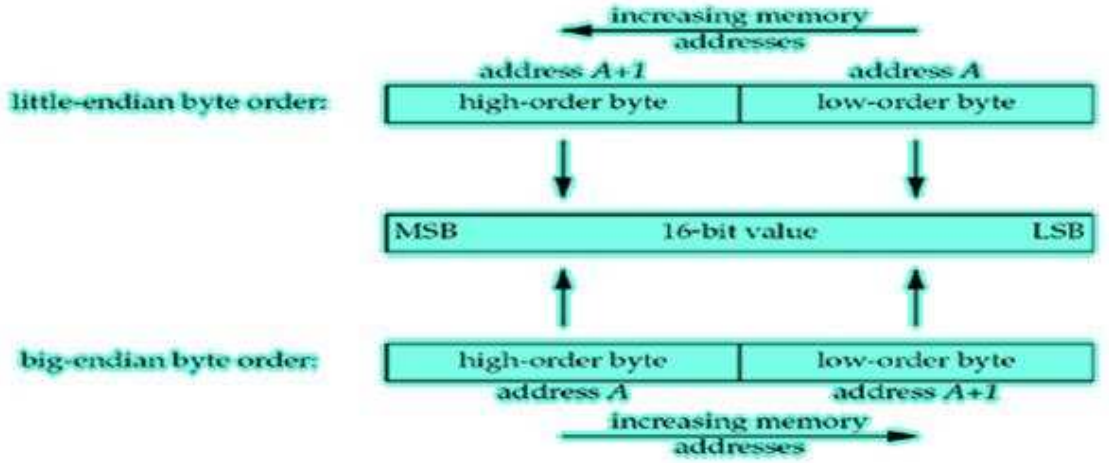
Consider a 16-bit integer that is made up of 2 bytes. There are two ways to store the two bytes in memory: with the low-order byte at the starting address, known as little-endian byte order, or with the high-order byte at the starting address, known as big-endian byte order.

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

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

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

Sockets Overview

The operating system includes the Berkeley Software Distribution (BSD) inter process 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.

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 protocolspecific 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:

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

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 ResultArguments

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.

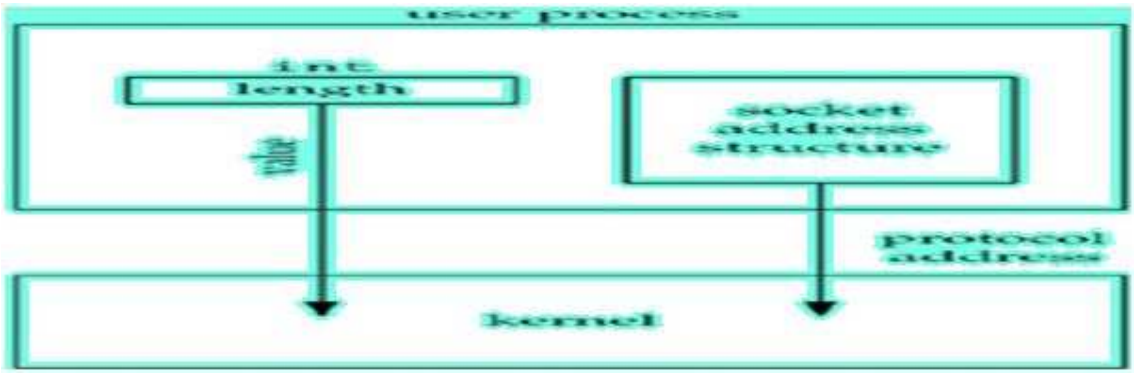


Figure: Socket address structure passed from process to kernel.

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 */
```

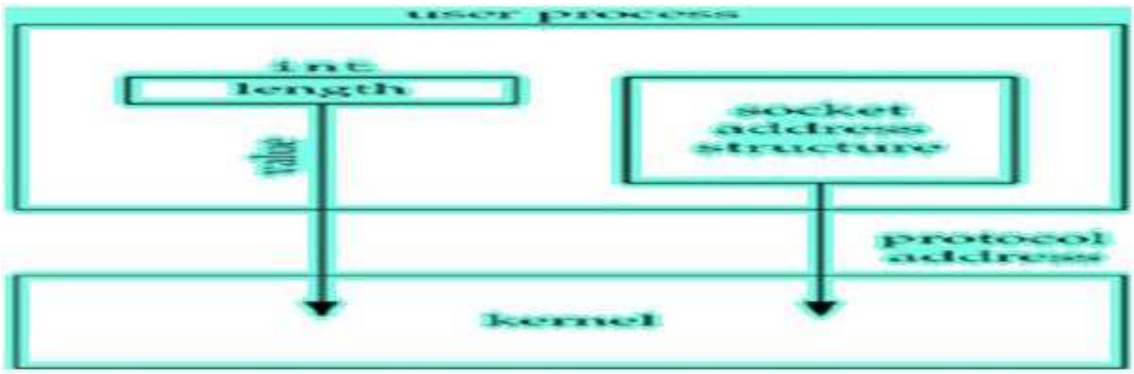


Figure shows this scenario.

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.

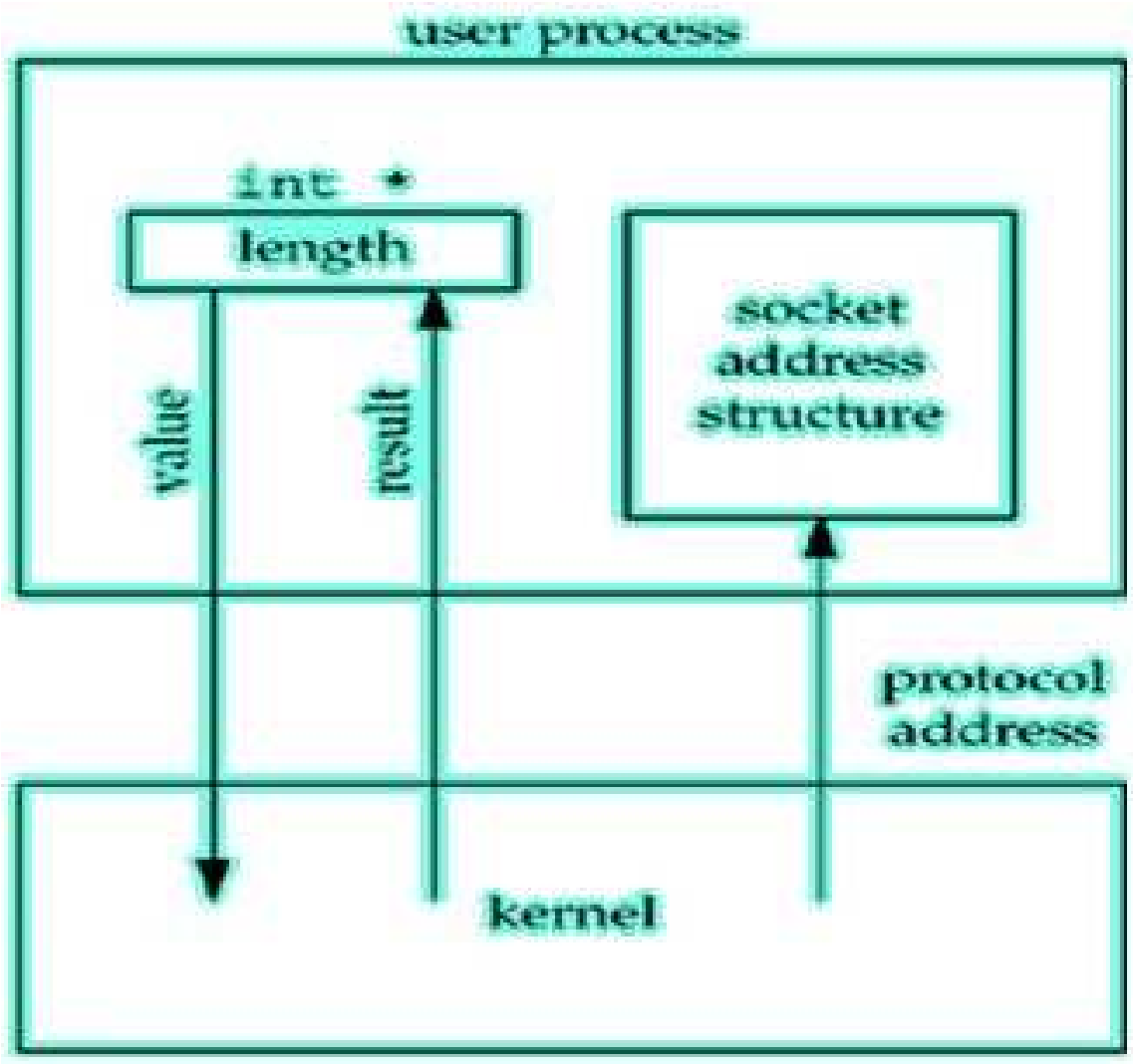
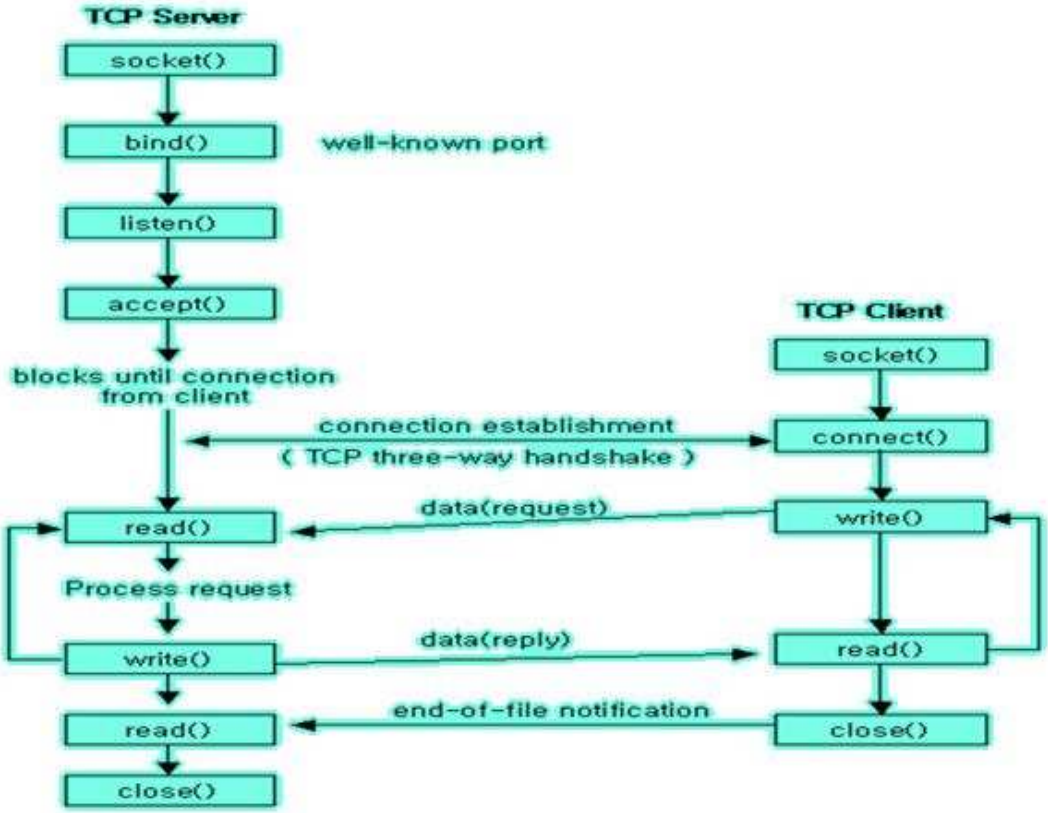


Figure. Socket address structure passed from kernel to process.

Figure shows this scenario.


TCP Server
socket()
bind()
well-known port
listen()
accept()
blocks until connection
from client
connection establishment
(TCP three-way handshake)
read()
Process request
write()
read()
close()
TCP Client
socket()
connect()
data(request)
write()
data(reply)
read()
end-of-file notification
close()

Part II

Programs

Program 1

Understanding Networking Commands

Problem Definition :-

Understanding and using the following commands ifconfig, netstat, ping, arp, telnet, tftp, ftp.

Problem Description: -

In this we will study few networking commands to gain an understanding of the commands and their usage along with their options

1. **ifconfig**

   Name

   ifconfig - configure a network interface

   Synopsis

   ifconfig [interface]

   ifconfig interface [aftype] options — address ...

   Description

   Ifconfig is used to configure the kernelresident network interfaces. It is used at boot time to set up interfaces as necessary.

   If no arguments are given, ifconfig displays the status of the currently active interfaces. If a single interface argument is given, it displays the status of the given interface only; if a single a argument is given, it displays the status of all interfaces, even those that are down. Otherwise, it configures an interface.

   Address Families

   If the first argument after the interface name is recognized as the name of a supported address family, that address family is used for decoding and displaying all protocol addresses. Currently supported address families include inet (TCP/IP, default), inet6 (IPv6), ax25 (AMPR Packet Radio), ddp (Appletalk Phase 2), ipx (Novell IPX) and netrom (AMPR Packet radio).

Options

interface:

The name of the interface. This is usually a driver name followed by a unit number, for example eth0 for the first Ethernet interface.

up:

This flag causes the interface to be activated

down:

This flag causes the driver for this interface to be shut down.

[-]arp:

Enable or disable the use of the ARP protocol on this interface.

[-]promisc:

Enable or disable the promiscuous mode of the interface

metric N:

This parameter sets the interface metric.

mtu N:

This parameter sets the Maximum Transfer Unit (MTU) of an interface.

dstaddr addr :

Set the remote IP address for a point-to-point link (such as PPP).

netmask addr :

Set the IP network mask for this interface.

add addr/prefixlen:

Add an IPv6 address to an interface.

del addr/prefixlen:

Remove an IPv6 address from an interface.

tunnel aa.bb.cc.dd:

Create a new SIT (IPv6-in-IPv4) device, tunneling to the given destination

io_addr addr :

Set the start address in I/O space for this device.

mem_start addr :

Set the start address for shared memory used by this device

media type:

Set the physical port or medium type to be used by the device

multicast :

Set the multicast flag on the interface.

address :

The IP address to be assigned to this interface.

2. netstat

Name

netstat -

Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships

Synopsis

```
netstat [address_family_options] [--tcp|-t] [--udp|-u]
        [--raw|-w] [--listening|-l] [--all|-a] [--numeric|-n]
[--numeric-hosts][--numeric-ports][--numeric-ports]
[--symbolic|-N] [--extend|-e[--extend|-e]] [--timers|-o]
[--program|-p] [--verbose|-v] [--continuous|-c]
[delay] netstat {--route|-r}
```

Description

Netstat prints information about the Linux networking subsystem. The type of information printed is controlled by the first argument, as follows:

```
--route , -r
```

Display the kernel routing tables.

`--groups , -g`

Display multicast group membership information for IPv4 and IPv6.

`--masquerade , -M`

Display a list of masqueraded connections.

Options

–verbose , -v:

Verbose output

–numeric , -n:

Show numerical addresses instead of trying to determine symbolic host, port or user names.

–numeric-hosts:

Shows numerical host addresses but does not affect the resolution of port or user names.

–numeric-ports:

Shows numerical port numbers but does not affect the resolution of host or user names.

–numeric-users:

Shows numerical user IDs but does not affect the resolution of host or port names.

-c, –continuous:

This will cause netstat to print the selected information every second continuously.

-e, –extend:

Display additional information. Use this option twice for maximum detail.

-o, –timers:

Include information related to networking timers.

-p, –program:

Show the PID and name of the program to which each socket belongs.

-l, –listening:

Show only listening sockets. (These are omitted by default.)

-a, –all:

Show both listening and non-listening sockets. Interfaces that are not marked

-F:

Print routing information from the FIB. (This is the default.)

-C:

Print routing information from the route cache.

3. **ping**

Name

ping, ping6 send ICMP ECHO_REQUEST to network hosts

Synopsis

```
            ping [ -LRUbdfnqrvVaAB] [ -c count] [ -i interval]
      [ -l preload][ -p pattern] [ -s packetsize]
[ -t ttl] [ -w deadline][ -F flowlabel]
[ -I interface] [ -M hint] [ -Q tos]
[ -S sndbuf] [ -T timestamp option]
[ -W timeout] [ hop ...]
destination
```

Description

ping uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams ("pings") have an IP and ICMP header, followed by a struct timeval and then an arbitrary number of "pad" bytes used to fill out the packet.

Options

-a:

Audible ping.

-A:

Adaptive ping.Allow pinging a broadcast address.

-B:

Do not allow ping to change source address of probes.

-c count:

Stop after sending count ECHO_REQUEST packets With deadline option, ping waits for count ECHO_REPLY packets, until the timeout expires.

-d:

Set the SO_DEBUG option on the socket being used.Essentially, this socket option is not used by

Linux kernel.

-F flow label:

Allocate and set 20 bit flow label on echo request packets.(Only ping6). If value is zero, kernel allocates random flow label.

-f:

Flood ping. For every ECHO_REQUEST sent a period "." is printed,while for ever ECHO_REPLY received a backspace is printed.

-i interval:

Wait interval seconds between sending each packet.

-L:

Suppress loopback of multicast packets.

-n:

Numeric output only. No attempt will be made to lookup symbolic names for host addresses.

-Q tos:

Set Quality of Service -related bits in ICMP datagrams. tos can be either decimal or hex number.

-q:

Quiet output. Nothing is displayed except the summary lines at startup time and when finished.

-r:

Bypass the normal routing tables and send directly to a host on an attached interface

-s packetsize:

Specifies the number of data bytes to be sent. The default is 56.

-S sndbuf:

Set socket sndbuf. If not specified, it is selected to buffer not more than one packet.

-t ttl:

Set the IP Time to Live.

-U:

Print full user-to-user latency (the old behaviour).

-v:

Verbose output.

-V:

Show version and exit.

-w deadline:

Specify a timeout, in seconds, before ping exits regardless of how many packets have been sent or received.

-W timeout:

Time to wait for a response, in seconds.

4. **arp**

Name

arp -

manipulate the system ARP cache

Synopsis

```
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 trying to determine symbolic host, port or user names.

-a [hostname]:

Shows the entries of the specified hosts.

-D, –use-device:

Use the interface ifa's hardware address.

-e :

Shows the entries in default (Linux) style.

5. **telnet**

Name telnet -

user interface to the TELNET protocol

Synopsis

```
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.

The options are as follows:

-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

-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

-e escapechar:

Sets the initial telnet escape character to escapechar If escapechar 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.

-l user:

When connecting to the remote system, if the remote system understands the ENVIRON option, then user will be sent to the remote system as the value for the variable 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.

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.

6. **tftp**

Name tftp -

IPv4 Trivial File Transfer Protocol client

Synopsis

```
tftp [ options... ] [host [port]] [-c command]
```

Description

Tftp is the user interface to the Internet **TFTP** (Trivial File Transfer Protocol), which allows users to transfer files to and from a remote machine. The remote host may be specified on the command line, in which case tftp uses host as the default host for future transfers.

Commands

Once **tftp** is running, it issues the prompt and recognizes the following commands:

? **command-name:**

Print help information.

ascii :

Shorthand for "mode ascii"

binary :

Shorthand for "mode binary"

connect host-name[port]:

Set the host (and optionally port for transfers. Note that the TFTP protocol, unlike the FTP protocol, does not maintain connections between transfers; thus, the connect command does not actually create a connection, but merely remembers what host is to be used for transfers. You do not have to use the connect command; the remote host can be specified as part of the get or put commands.

get filename :

get remotename localname

get file1 file2 ... fileN:

Get a file or set of files from the specified sources Source can be in one of two forms: a filename on the remote host, if the host has already been specified, or a string of the form hosts:filename to specify both a host and filename at the same time. If the latter form is used, the last hostname specified becomes the default for future transfers.

mode transfer-mode :

Set the mode for transfers; transfermode may be one of ascii or binary The default is ascii put file /put localfile remotefile /put file1 file2 ... fileN remote-directory : Put a file or set of files to the specified remote file or directory.

The destination can be in one of two forms:

A filename on the remote host, if the host has already been specified, or a string of the form

hosts:filename :

To specify both a host and filename at the same time. If the latter form is used, the hostname specified becomes the default for future transfers. If the remote-directory form is used, the remote host is assumed to be a **UNIX** machine.

```
quit:  Exit tftp An end of file also exits.
status :  Show current status.
trace :  Toggle packet tracing.
```

7. **ftp**

Name ftp -

Internet file transfer program

Synopsis

```
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.

-p:

Use passive mode for data transfers. Allows use of ftp in environments where a firewall prevents connections from the outside world back to the client machine.

-i:

Turns off interactive prompting during multiple file transfers.

-n:

Restrains ftp from attempting autologin 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 globbing.

-v:

Verbose option forces ftp to show all responses from the remote server, as well as report on data transfer statistics.

-d:

Enables debugging.

ascii:

Set the file transfer type to network ASCII This is the default type.

bell:

Arrange that a bell be sounded after each file transfer command is completed.

binary:

Set the file transfer type to support binary image transfer.

bye :

Terminate the FTP session with the remote server and exit ftp An end of file will also terminate the session and exit.

case :

Toggle remote computer file name case mapping during mget commands.

cd remote-directory:

Change the working directory on the remote machine to remote-directory

close:

Terminate the FTP session with the remote server, and return to the command interpreter. Any defined macros are erased.

cr:

Toggle carriage return stripping during ascii type file retrieval.

delete remote-file:

Delete the file remote-file on the remote machine.

disconnect:

A synonym for close

help[command]:

Print an informative message about the meaning of command If no argument is given, ftp prints a list of the known commands.

idle[seconds]:

Set the inactivity timer on the remote server to seconds seconds. If seconds is ommitted, the current inactivity timer is printed.

Program Validation

```
Input: \$ [command ]   {- options}
```

Output:

Displays the result related to the entered command.

Conclusion:

By using these commands we gain an understanding of networking basics commands.

Program 2(A)

Implementing Iterative Echo Server using Socket System calls

Problem Definition

To implement an iterative echo server using both connection-oriented and connection less Socket System Calls

Problem Description: -

We will implement echo service using connection-oriented (TCP) or connection less (UDP) where the client passes the message to the server and then the server passes the same message back to the client who then prints it. 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 userdefined 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.

Algorithm:

1. a) Connection Oriented Implementation of iterative server:

   Server:

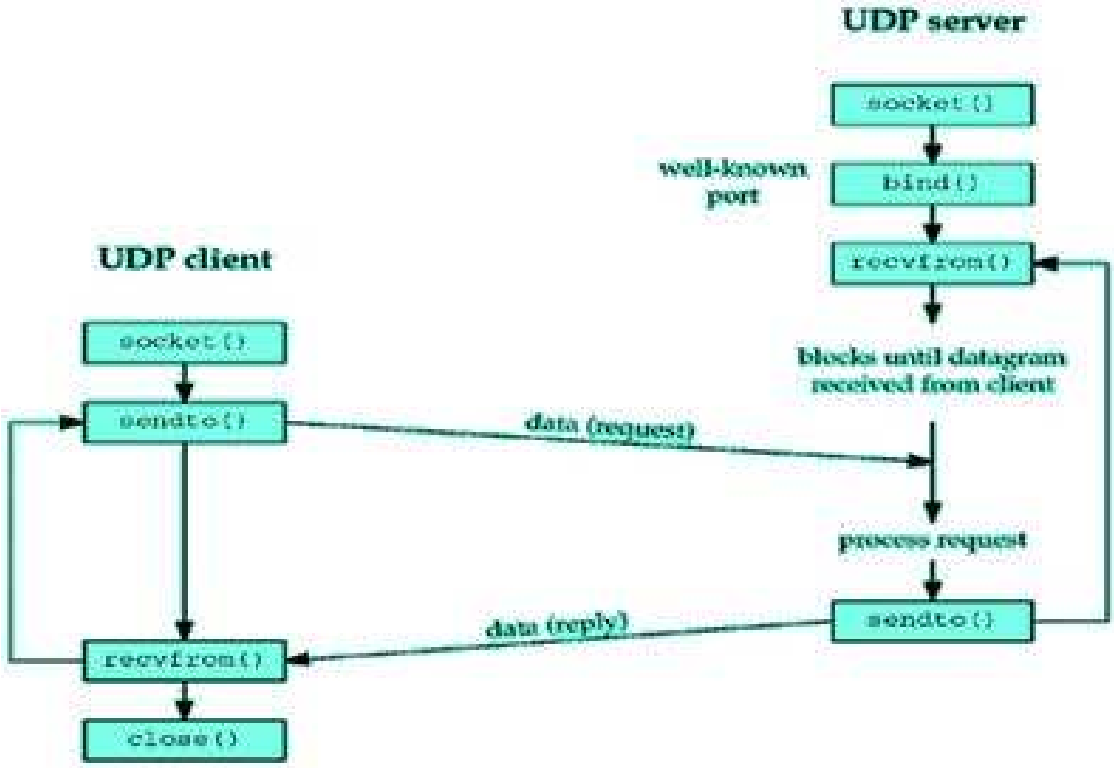
   - Include appropriate header files.
   - Create a TCP Socket.
   - Fill in the socket address structure (with server information)
   - Specify the port where the service will be defined to be used by client.
   - Bind the address and port using bind() system call.
   - Server executes listen() system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

FLOWCHART



2. **Connection less Implementation of iterative server:**

   Algorithm:

   Server:

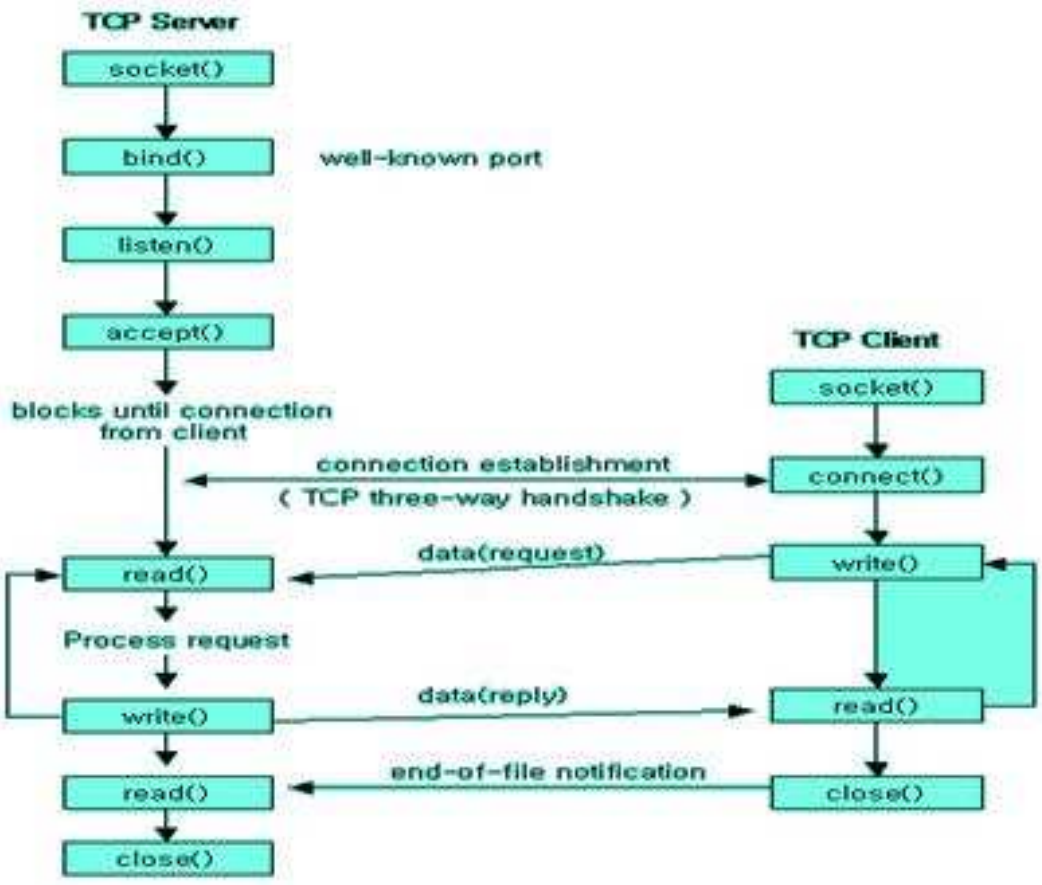
   - Include appropriate header files.
   - Create a UDP Socket.
   - Fill in the socket address structure (with server information)
   - Specify the port where the service will be defined to be used by client.
   - Bind the address and port using bind() system call.
   - Receive a message from the Client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



Program Validation

Input

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

First compile the Server program as,

/* cc is used for compiling the program and o filename is used to specify the compiled executable file name */

$ cc server.c o obj

/* & is used to execute the program in background */

$ ./obj&

/* Then compile the client program */

$ cc client.c

/* Then execute the client program. a.out is the default compiled executable file. To view its contents we use ./a.out */

$./a.out

Sample Input:

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

Output

Sample Output:

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

Conclusion

The C program to implement echo service was executed successfully using connection oriented /connection less iterative server using socket system calls.

Program 2(B)

Implementing Concurrent Echo Server using Socket System calls

Problem Definition

To implement a concurrent echo server using both connection-oriented and connection less Socket System Calls

Problem Description: -

We will implement echo service using connectionoriented (TCP) or connection less (UDP) where the client passes the message to the server a nd then the server passes the same message back to the client who then prints it. In order to implement the concurrent Service we need to create an application for instance say client, which will be invoking service which is established on the concurrent server working on a userdefined port. The Client will be creating its socket endpoint and establish a connection with the concurrent server by specifying the port number similar to that of the Server. The amount of work required to handle a request is unknown, so the server starts another process to handle each request.

Algorithm:

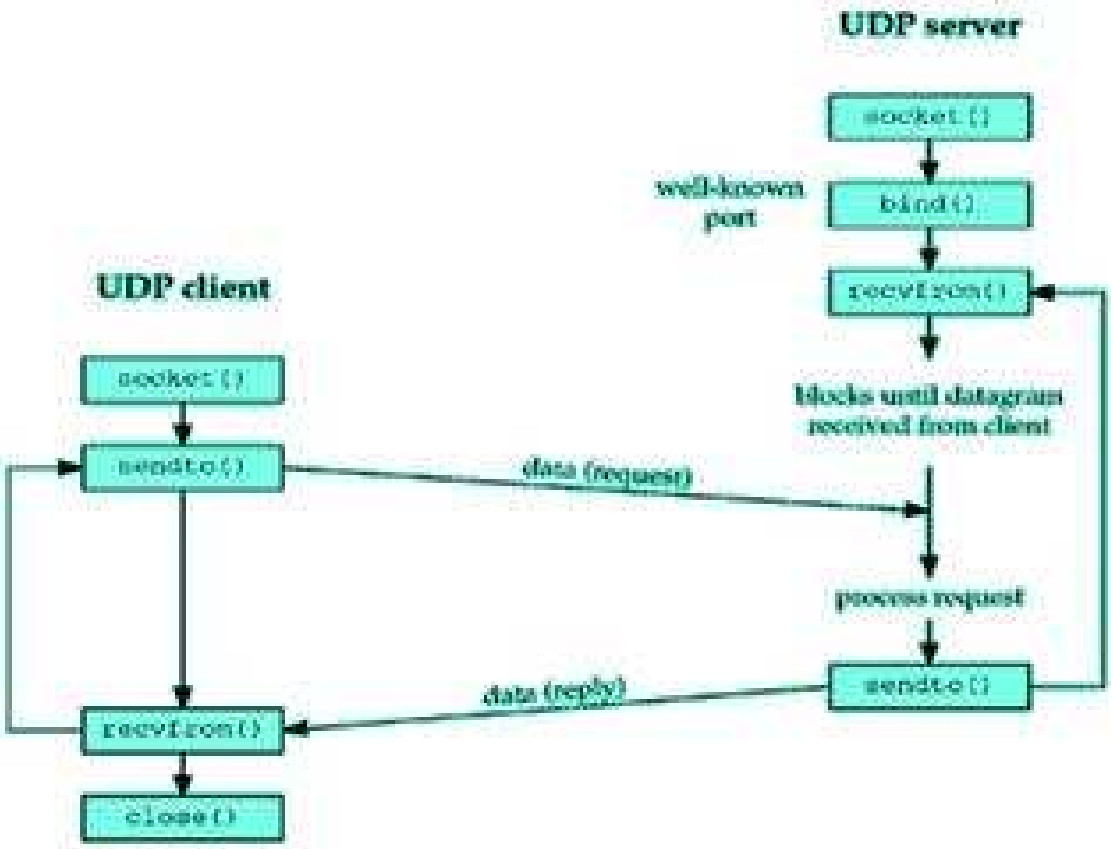
1. Connection Oriented Implementation of Concurrent server:

   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.
- **FLOW-CHART :**



2. **Connection less Implementation of Concurrent Server**

Algorithm:

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.

FLOW CHART:

Program Validation

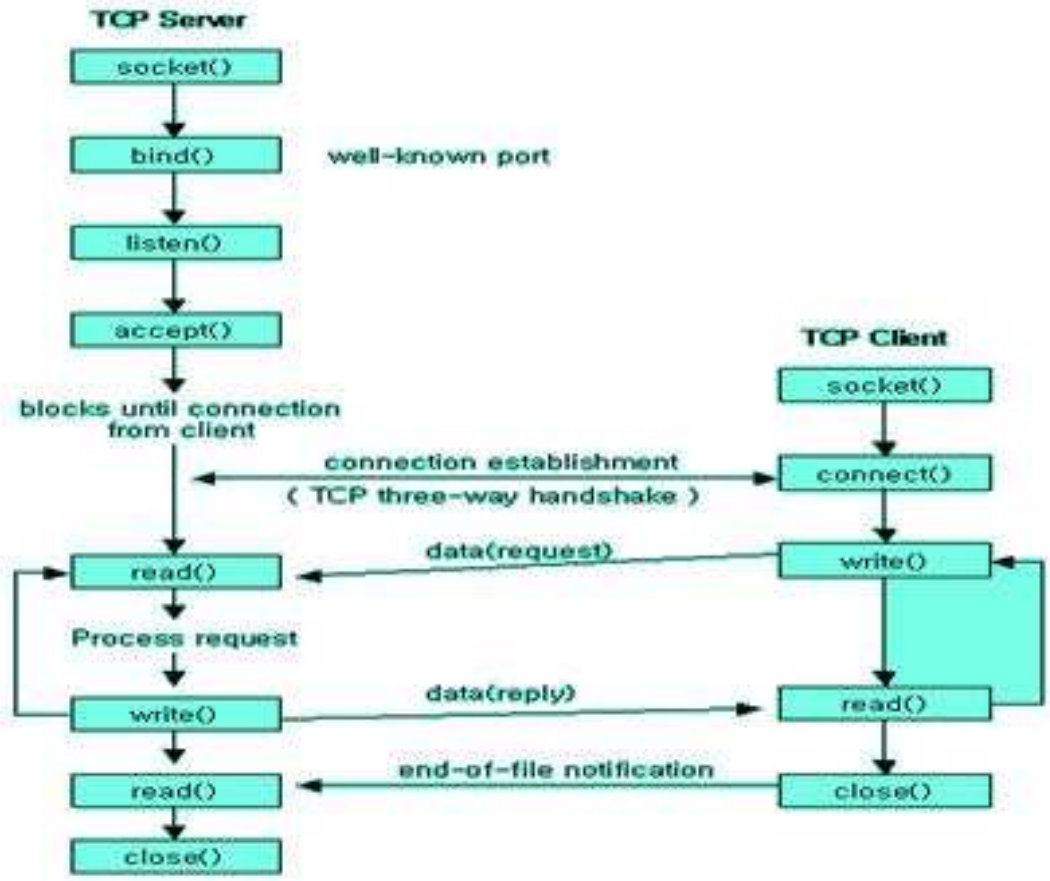
Input

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

First compile the Server program as,
/* cc is used for compiling the program and o filename is used to specify the compiled executable file name */

$ cc server.c - o obj

/* & is used to execute in background*/



$ ./obj&

/* Then compile the client program */

$ cc client.c

/* a.out is the default compiled executable file. To view its contents we use ./a.out */

$./a.out

Sample Input:

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

Output

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

Conclusion

The C program to implement echo service was executed successfully using connection oriented /connection less concurrent server using socket system calls.

Program 3

Implementing Time of the Day Service

Problem Definition

To implement time of the day service using connection oriented concurrent server using socket system calls

Problem Description: -

The client establishes a TCP connection with a server and the server simply sends back the current time and date in human readable form (e.g., the format got by executing date at prompt). To implement this we use "time" function of C.

Algorithm:

Server:

- Include appropriate header files.
- Create a TCP socket using socket( ) system call..
- Bind server's address and port using bind( ) system call.
- Convert the socket into a listening socket using listen( ) sytem call.
- Wait for client connection to complete using 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.
- Accept client connection and then call the time( ) to get current date and time which returns the time in time _t object format, then use

```
ctime( ) to convert time in
  string(human readable format).
```

- Send the result (time and date) using send() system call.
- After sending reply terminate connection (in child process).

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 with the Server using connect() system call.
- Send the request to the server 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.

Program Validation

Input

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

First compile the Server program as,

/* cc is used for compiling the program and –o filename is used to specify the compiled executable file name */

$ cc server.c – o obj

/* & is used to execute in background*/

$ ./obj&

/* Then compile the client program */

$ cc client.c

/* a.out is the default compiled executable file where output will be kept. To view its contents we use ./a.out */

$./a.out

Output

Sample Output:

Server gives daytime response to the client which client prints on screen

```
FRI MAR 12 14:27:52 2016
```

Conclusion

The C program to implement time of day service was executed successfully using connection oriented concurrent server using socket system calls.

Program 4

Build a concurrent multithreaded file transfer server

Problem Definition

To build a concurrent multithreaded file transfer server using threads using separate threads to allow server to handle multiple clients concurrently.

Problem Description: -

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

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void * (*start_routine)(void*),  void *arg);
```

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);
```

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.

Algorithm

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.

Program Validation

Input

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

First compile the Server program as,

/* cc is used for compiling the program and –o filename is used to specify the compiled executable file name */

$ cc server.c – o obj

/* & is used to execute in background*/

$ ./obj&

/* Then compile the client program */

$ cc client.c

/* a.out is the default compiled executable file. To view its contents we use ./a.out */

$./a.out

Sample Input:

Client sends file name. For ex: File1

Output

Sample Output:

Server transfers the requested page to the Client

Conclusion

The C program to build file transfer server was executed successfully using connection oriented server using socket system calls.

Program 5

Implementing Remote Program Execution

Problem Definition

To implement Remote Program Execution using socket system calls.

Program Description

Remote Program 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 to capture its output also.

Algorithm

Server:

- Create a socket using socket( ) system call..
- Bind server's address and port using bind( ) system call.
- Convert the socket into a listening socket using listen( ) sytem call.
- Wait for client connection to complete using accept( ) system call.
- Receive the Client request using recv() system call which consist of the name of the program that is to be executed along with data parameters(if any).
- The program name is handled using service() and then using system() we execute it and then using open() we open the program to copy its contents in a string and then we pass its contents to the client using send().

Client

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

Program Validation

Input

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

First compile the Server program as,

/* cc is used for compiling the program and –o filename is used to specify the compiled executable file name */

$ cc server.c – o obj

/* & is used to execute in background*/

$ ./obj&

/* Then compile the client program */

$ cc client.c

/* a.out is the default compiled executable file. To view its contents we use ./a.out */

$./a.out

Sample Input:

The Client sends the name of the program to be executed, for instance sample

Output

Sample Output:

Server executes the program and then transfers result back to the Client

Conclusion

The C program to implement remote program execution was successfully executed using socket system calls.

Program 5(B)

Implementing Demonstrate Advanced Socket System Calls

Problem Definition

To Demonstrate the usage of advanced socket system calls.

Program Description

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

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 locallybound 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.

Algorithm

- 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.

Program Validation

Input

Suppose, the program is getset.c

First compile the program as,

/* cc is used for compiling the program */

$ cc getset.c

/* a.out is the default compiled executable file. To view its contents we use ./a.out */

$./a.out

Output

Sample Output:

Program displays the socket name, peername and other socket options which are specified.

Conclusion

The C program to demonstrated the usage of advanced socket system calls was successfully executed using socket system calls.

Program 6

Implementing Concurrent Chat Server

Problem Definition

To implement a concurrent chat server that allows current logged in users to communicate with one another using socket system calls.

Problem Description

In this program, we try to determine the number of Users currently logged in and establish chat session with them. 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.

Algorithm

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.
- 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 or fgets().
- 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 or fgets().
- Write the result thus obtained on the standard output.

Program Validation

Input:

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

First compile the Server program as,

/* cc is used for compiling the program */

$ cc server.c

/* Then compile the client program in new tab*/

$ cc client.c

/* Then execute the server program first using a.out. (a.out is the default compiled executable file). To view its contents we use ./a.out */

$./a.out

/* Then execute the client program in the other tab using a.out. (a.out is the default compiled executable file). To view its contents we use ./a.out */

Sample Input:

The Client sends messages to the server and server sends messages to clients.

Output

Sample Output:

Server receives the messages which are sent by client and the Client receives the messages sent by server.

Conclusion

The C program to implement concurrent chat server was successfully executed using socket system calls.

Program 7

Remote File Access using RPC

Problem Definition

To implement Remote File Access using Remote Procedure Call.

A file at the server is to be accessed using Remote Procedure calls. High level programming through remote procedure calls (RPC) provides logical clienttoserver 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).

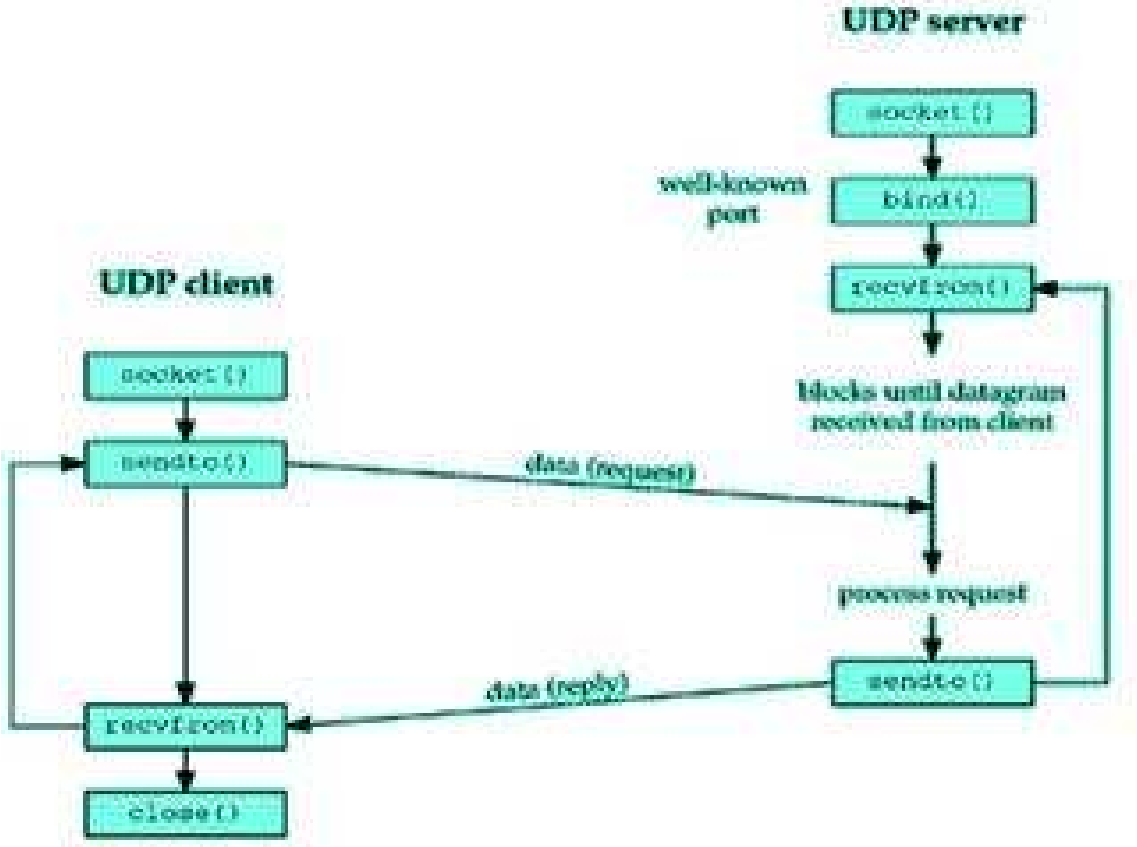
RPC model

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.

Algorithm

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

Program Validation

Execution Procedure:

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

/* compiling file.x using rpcgen */

$ rpcgen – a file.x

/* seeing the list of files which gets created using ls command */

$ ls

file _ client.c file.h file _ svc.c Makefile.x file _ clnt.c file _ server.c file.x

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

/*compiling server and server stub program */

/* cc is used for compiling the program and – o filename is used to specify the compiled executable file name */

$ cc file_ server.c file _ svc.c – o obj

/* & is used to execute in background*/

$ ./obj&

[2]5030

/*compiling client and client stub program */

$ cc file_client.c file_clnt.c

/* a.out is the default compiled executable file. To view its contents we use ./a.out */

$ ./a.out 192.100.100.6

Sample input:

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

Sample Output:

The output of the file is displayed on screen. Ex: This is file access

Conclusion :

The C program to implement Remote File Access was successfully executed using RPC.

Annexure – I

CS 381 NETWORK PROGRAMMING LAB

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

1. Understanding and using the following commands.
   Ifconfig, netstat, ping,arp, telnet, tftp, ftp.

2. Implementation of concurrent and iterative Echo server using both connection oriented and connectionless Socket System Calls.

3. Implementation of time of the day service as Connection Oriented Concurrent Server using Socket System Calls.

4. Build a concurrent Multithreaded File Transfer Server. Use separate Threads to allow the server to handle multiple clients concurrently.

5. Implementation of Remote Program execution using Socket system calls.
   Programs to demonstrate the usage of Advanced Socket System calls Like Getsockopt(),Setsockopt(), Select(), Readv(), getpeernanel(), Getsockname().
6. Implement a Concurrent Chat Server that allows currently logged in users to communicate with one another. Use Socket System calls.
7. Implementation of Remote files Access using RPC.