JAGANNATH GUPTA INSTITUTE OF ENGINEERING AND TECHNOLOGY

(Approved by AICTE and Affiliated to RTU, Kota)

LABORATORY MANUAL

(2019-2020)

UNPS LAB

IV Year & VIII Semester

Computer Science & Engineering

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDEX

SYLLABUS

8CS5 UNIX NETWORK PROGRAMMING & SIMULATION LAB

At the end of course, the students should be able to:

• Understand various distributions of Unix viz. BSD, POSIX etc.
• Write client/server applications involving unix sockets involving TCP or UDP involving iterative or concurrent server.
• Understand IPV4 & IPV6 interoperability issues
• Use fork( ) system call.
• Understand the network simulator NS2 and Simulate routing algorithm on NS2. (Available on http://www.isi.edu/nsnam/ns/).

Suggested Platform: For Socket Programming- Linux, For NS2 Any of Microsoft Windows or Linux (In case of Microsoft, Virtual environment cygwin will also be required).

Suggested Exercises

S.No. List of Experiments

1. Write two programs in C: hello_client and hello_server
    - The server listens for, and accepts, a single TCP connection; it reads all the data itcan from that connection, and prints it to the screen; then it closes the connection
    - The client connects to the server, sends the string "Hello, world!", then closes theconnection
2. Write an Echo_Client and Echo_server using TCP to estimate the round trip timefrom client to the server. The server should be such that it can accept multipleconnections at any given time.
3. Repeat Exercises 1 & 2 for UDP.
4. Repeat Exercise 2 with multiplexed I/O operations
5. Simulate Bellman-Ford Routing algorithm in NS2

List of Lab Experiments

Experiment No:-1	Introduction to Socket Programming
Experiment No:-2	Write a programs in C: hello_client (The server listens for, and accepts, a single TCP connection; it reads all the data it can from that connection, and prints it to the screen; then it closes the connection)
Experiment No:-3	Write a programs in C: hello_server (The client connects to the server, sends the string "Hello, world!", then closes the connection)
Experiment No:-4	Write a programs in C for TCP chat server
Experiment No:-5	Write a programs in C: hello_client (The server listens for, and accepts, a single UDP connection; it reads all the data it can from that connection, and prints it to the screen; then it closes the connection)
Experiment No:-6	Write a programs in C: hello_server (The client connects to the server, sends the string "Hello, world!", then closes the connection)
Experiment No:-7	Write a programs in C for UDP chat server
Experiment No:-8	Write an Echo_server using TCP to estimate the round trip time from client to the server. The server should be such that it can accept multiple connections at any given time , with multiplexed I/O operations
Experiment No:-9	Write an Echo_Client using UDP to estimate the round trip time from client to the server. The server should be such that it can accept multiple connections at any given time, with multiplexed I/O operations
Experiment No:-10	Program using fork( ) system call.
Experiment No:-11	Simulate Bellman-Ford Routing algorithm in NS2
Experiment No:-12	Simulation of sliding window protocol.
Experiment No:-13	Simulation of File transfer protocol.

Course Objectives & Course Outcomes

Course Objectives: The student should be made:

1. To educate them about UNIX environment.
2. To train them to simulate the routing protocols.
3. To make the students aware of the TCP and UDP Sockets.
4. To emphasis the students the importance of Network Programming.
5. To simulate chat application of their own

Course Outcomes: Upon Completion of the course, the students will be able to

1. To get an idea of how the process executes in UNIX.
2. To know the concept of inter process communication.
3. To implement the network programming in UNIX.
4. To make a client server communication through TCP and UDP protocols.
5. To expose on advanced socket programming, domain name system, http in UNIX environment.

Content Beyond Syllabus

Write a programs in C for UDP chat server Program using fork( ) system call. Simulation of sliding window protocol.
Simulation of File transfer protocol.

Experiment No.: 1

Aim: Introduction to Socket Programming

Keywords: sockets, client-server, network programming-socket functions, OSI layering, byte-ordering

Outline:

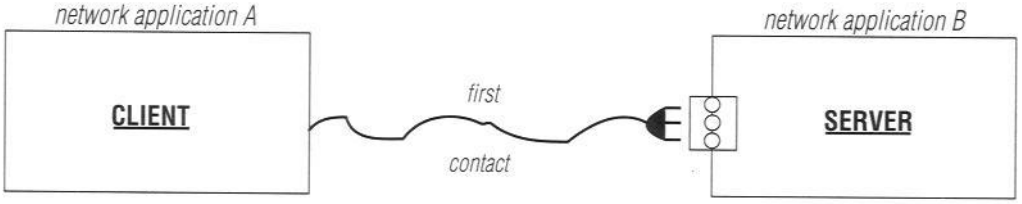
1.) Introduction
2.) The Client / Server Model
3.) The Socket Interface and Features of a TCP connection
4.) Byte Ordering
5.) Address Structures, Ports, Address conversion functions
6.) Outline of a TCP Server
7.) Outline of a TCP Client
8.) Client-Server communication outline
9.) Summary of Socket Functions

1.) Introduction

In this Lab you will be introduced to socket programming at a very elementary level. Specifically, we will focus on TCP socket connections which are a fundamental part of socket programming since they provide a connection oriented service with both flow and congestion control. What this means to the programmer is that a TCP connection provides a reliable connection over which data can be transferred with little effort required on the programmers part; TCP takes care of the reliability, flow control, congestion control for you. First the basic concepts will be discussed, then we will learn how to implement a simple TCP client and server.

2.) The Client / Server Model

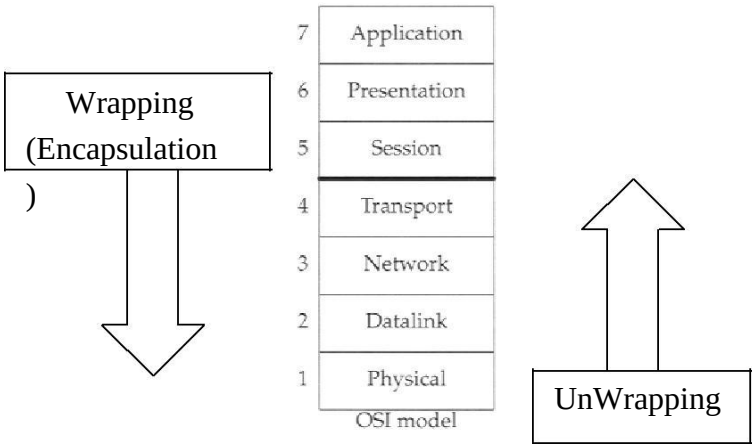
It is possible for two network applications to begin simultaneously, but it is impractical to require it. Therefore, it makes sense to design communicating network applications to perform complementary network operations in sequence, rather than simultaneously. The server executes first and waits toreceive; the client executes second and sends the first network packet to the server. After initial contact, either the client or the server is capable of sending and receiving data.



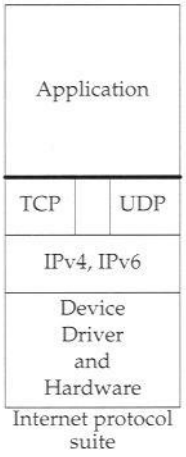
A client initiates communications to a server.

3.) The Socket Interface and Features of a TCP connection

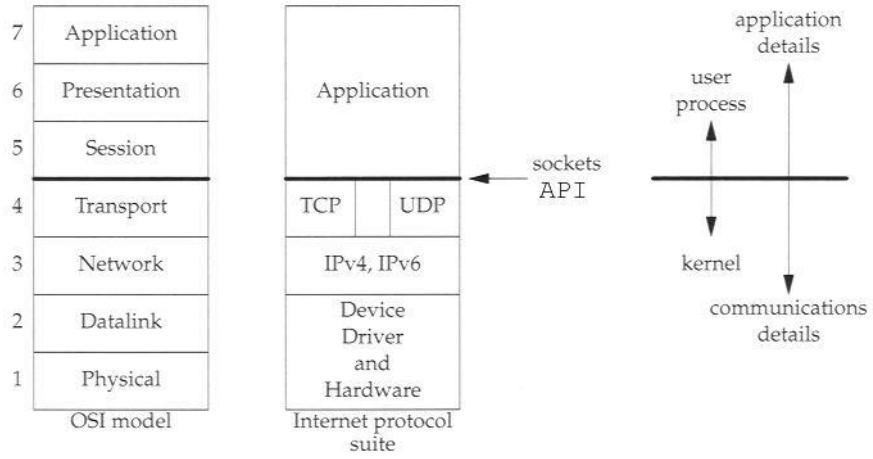
The OSI Layers:



The Internet Layers:



The Internet does not strictly obey the OSI model but rather merges several of the protocols layers together. Where is the socket programming interface in relation to the protocol stack?



Features of a TCP connection:

- Connection Oriented
- Reliability
  1. Handles lost packets
  2. Handles packet sequencing
  3. Handles duplicated packets
- Full Duplex
- Flow Control
- Congestion Control

TCP versus UDP as a Transport Layer Protocol:

TCP	UDP
Reliable, guaranteed	Unreliable. Instead, prompt delivery of packets.
Connection-oriented	Connectionless
Used in applications that require safety gurantee. (eg. file applications.)	Used in media applications. (eg. video or voice transmissions.)
Flow control, sequencing of packets, error-control.	No flow or sequence control, user must handle these manually.
Uses byte stream as unit of transfer. (stream sockets)	Uses datagrams as unit of transfer. (datagram sockets)
Allows to send multiple packets with a single ACK.	
Allows two-way data exchange, once the connection is established. (full-duplex)	Allows data to be transferred in one direction at once. (half-duplex)
e.g. Telnet uses stream sockets. (everything you write on one side appears exaclt in same order on the other side)	e.g. TFTP (trivial file transfer protocol) uses datagram sockets.

Sockets versus File I/O

Working with sockets is very similar to working with files. The socket() and accept() functions both return handles (file descriptor) and reads and writes to the sockets requires the use of these handles (file descriptors). In Linux, sockets and file descriptors also share the same file descriptor table. That is, if you open a file and it returns a file descriptor with value say 8, and then immediately open a socket, you will be given a file descriptor with value 9 to reference that socket. Even though sockets and files share the same file descriptor table, they are still very different. Sockets have addresses associated with them whereas files do not, notice that this distinguishes sockets form pipes, since pipes do not have addresses with which they associate. You cannot randomly access a socket like you can a file with lseek(). Sockets must be in the correct state to perform input or output.

File I/O	*Network I/O*
open a file	open a socket
	name the socket
	associate with another socket
read and write	send and receive between sockets
close the file	close the socket

4.) Byte Ordering

Port numbers and IP Addresses (both discussed next) are represented by multi-byte data types which are placed in packets for the purpose of routing and multiplexing. Port numbers are two bytes (16 bits) and IP4 addresses are 4 bytes (32 bits), and a problem arises when transferring multi-byte data types between different architectures. Say Host A uses a "*big-endian*" architecture and sends a packet across the network to Host B which uses a "*little-endian*" architecture. If Host B looks at the address to see if the packet is for him/her (choose a gender!), it will interpret the bytes in the opposite order and will wrongly conclude that it is not his/her packet. ***The Internet uses big-endian*** and we call it the *network-byte-order*, and it is really not important to know which method it uses since we have the following functions to convert host-byte-ordered values into network-byte-ordered values and vice versa:

To convert port numbers (16 bits):

Host -> Network

```
unit16_t  htons( uint16_t hostportnumber )
```

Network -> Host

```
unit16_t  ntohs( uint16_t netportnumber )
```

To convert IP4 Addresses (32 bits):

Host -> Network
unit32_t htonl(uint32_t hostportnumber)

Network -> Host
Unit32_t ntohl(uint32_t netportnumber)

5.) Address Structures, Ports, Address conversion functions

Overview of IP4 addresses:

IP4 addresses are 32 bits long. They are expressed commonly in what is known as dotted decimal notation. Each of the four bytes which makes up the 32 address are expressed as an integer value (0 – 255) and separated by a dot. For example, 138.23.44.2 is an example of an IP4 address in dotted decimal notation. There are conversion functions which convert a 32 bit address into a dotted decimal string and vice versa which will be discussed later.

Often times though the IP address is represented by a domain name, for example, hill.ucr.edu. Several functions described later will allow you to convert from one form to another (Magic provided by DNS!).

The importance of IP addresses follows from the fact that each host on the Internet has a unique IP address. Thus, although the Internet is made up of many networks of networks with many different types of architectures and transport mediums, it is the IP address which provides a cohesive structure so that at least theoretically, (there are routing issues involved as well), any two hosts on the Internet can communicate with each other.

Ports:

Sockets are UNIQUELY identified by Internet address, end-to-end protocol, and port number.
That is why when a socket is first created it is vital to match it with a valid IP address and a port number. In our labs we will basically be working with TCP sockets.

Ports are software objects to multiplex data between different applications. When a host receives a packet, it travels up the protocol stack and finally reaches the application layer. Now consider a user running an ftp client, a telnet client, and a web browser concurrently. To which application should the packet be delivered? Well part of the packet contains a value holding a port number, and it is this number which determines to which application the packet should be delivered.

So when a client first tries to contact a server, which port number should the client specify? For many common services, standard port numbers are defined.

Port	*Service Name, Alias*	*Description*
1	`tcpmux`	TCP port service multiplexer
7	`echo`	Echo server
9	`discard`	Like `/dev/null`
13	`daytime`	System's date/time
20	`ftp-data`	FTP data port
21	`ftp`	Main FTP connection
23	`telnet`	Telnet connection
25	`smtp, mail`	UNIX mail
37	`time, timeserver`	Time server
42	`nameserver`	Name resolution (DNS)
70	`gopher`	Text/menu information
79	`finger`	Current users
80	`www, http`	Web server

Ports 0 – 1023, are reserved and servers or clients that you create will not be able to **bind** to these ports unless you have root privilege.

Ports 1024 – 65535 are available for use by your programs, but beware other network applications maybe running and using these port numbers as well so do not make assumptions about the availability of specific port numbers. Make sure you read Stevens for more details about the available range of port numbers!

Address Structures:

Socket functions like connect(), accept(), and bind() require the use of specifically defined address structures to hold IP address information, port number, and protocol type. This can be one of the more confusing aspects of socket programming so it is necessary to clearly understand how to use the socket address structures. The difficulty is that you can use sockets to program network applications using different protocols. For example, we can use IP4, IP6, Unix local, etc. Here is the problem: Each different protocol uses a different address structure to hold its addressing information, yet they all use the same functions connect(), accept(), bind() etc. So how do we pass these different structures to a given socket function that requires an address structure? Well it may not be the way you would think it should be done and this is because sockets where developed a long time ago before things like a void pointer where features in C. So this is how it is done:

There is a generic address structure: struct sockaddr

This is the address structure which must be passed to all of the socket functions requiring an address structure. ***This means that you must type cast*** your specific protocol dependent address structure to the generic address structure when passing it to these socket functions.

Protocol specific address structures usually start with *sockaddr_* and end with a ***suffix*** depending on that protocol. For example:

struct sockaddr_in (IP4, think of in as internet)
struct sockaddr_in6 (IP6)
struct sockaddr_un (Unix local)
struct sockaddr_dl (Data link)

We will be only using the IP4 address structure: struct sockaddr_in.
So once we fill in this structure with the IP address, port number, etc we will pass this to one of our socket functions and we will need to type cast it to the generic address structure. For example:

```
struct sockaddr_in  myAddressStruct;

//Fill in the address information into myAddressStruct here, (will be explained in detail shortly)

connect(socket_file_descriptor, (struct sockaddr *) &myAddressStruct,
sizeof(myAddressStruct));
```

Here is how to fill in the sockaddr_in structure:

```
struct sockaddr_in{

        sa_family_t  sin_family          /*Address/Protocol Family*/ (we'll use PF_INET)
        unit16_t          sin_port       /* 16-bit Port number          --Network Byte Ordered--
*/
        struct in_addr       sin_addr    /*A struct for the 32        bit IP Address  */
        unsigned char sin_zero[8]        /*Just ignore this it        is just padding*/
};

struct in_addr{
        unit32_t         s_addr   /*32 bit IP Address   --Network Byte Ordered-- */
};
```

For the sa_family variable sin_family always use the constant: PF_INET or AF_INET
***Always initialize address structures with bzero() or memset() before filling them in ***
Make sure you use the byte ordering functions when necessary for the port and IP address variables otherwise there will be strange things a happening to your packets

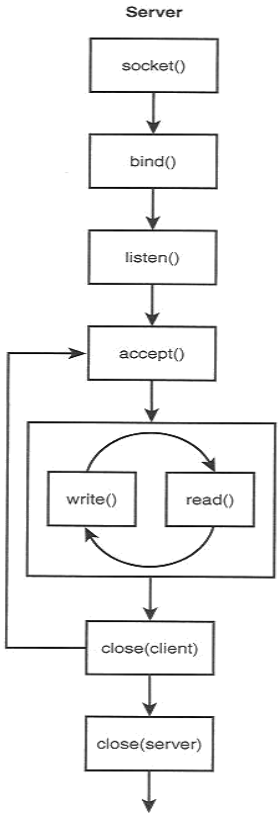
To convert a string dotted decimal IP4 address to a NETWORK BYTE ORDERED 32 bit value use the functions:

- inet_addr()
- inet_aton()

To convert a 32 bit NETWORK BYTE ORDERED to a IP4 dotted decimal string use:

- inet_ntoa()

6.) Outline of a TCP Server:



Step 1: Creating a socket:

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

Creating a socket is in some ways similar to opening a file. This function creates a file descriptor and returns it from the function call. You later use this file descriptor for reading, writing and using with other socket functions

Parameters:
family: AF_INET or PF_INET (These are the IP4 family)
type: SOCK_STREAM (for TCP) or SOCK_DGRAM (for UDP)
protocol: IPPROTO_TCP (for TCP) or IPPROTO_UDP (for UDP) or use 0

Step 2: Binding an address and port number

int bind(int socket_file_descriptor, const struct sockaddr * LocalAddress, socklen_t AddressLength);

We need to associate an IP address and port number to our application. A client that wants to connect to our server needs both of these details in order to connect to our server. Notice the difference between this function and the connect() function of the client. The connect function specifies a remote address that the client wants to connect to, while here, the server is specifying to the bind function a local IP address of one of its Network Interfaces and a local port number.

The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *bind()* is 0 for success and –1 for failure.

**Again make sure that you cast the structure as a generic address structure in this function **

You also do not need to find information about the IP addresses associated with the host you are working on. You can specify: INNADDR_ANY to the address structure and the bind function will use on of the available (there may be more than one) IP addresses. This ensures that connections to a specified port will be directed to this socket, regardless of which Internet address they are sent to. This is useful if host has multiple IP addresses, then it enables the user to specify which IP address will be b_nded to which port number.

Step 3: Listen for incoming connections

Binding is like waiting by a specific phone in your house, and Listening is waiting for it to ring.

int listen(int socket_file_descriptor, int backlog);

The backlog parameter can be read in Stevens' book. It is important in determining how many connections the server will connect with. Typical values for backlog are 5 – 10.

The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *listen()* is 0 for success and –1 for failure.

Step 4: Accepting a connection.

int accept (int socket_file_descriptor, struct sockaddr * ClientAddress, socklen_t *addrlen);

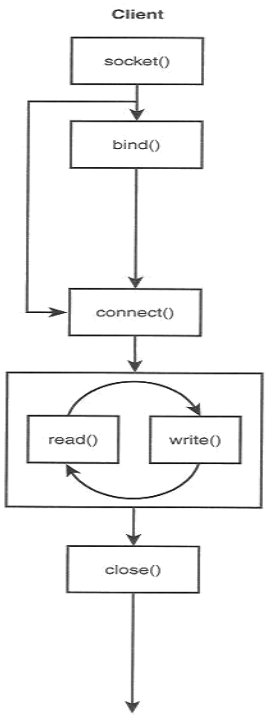
accept() returns a new socket file descriptor for the purpose of reading and writing to the client. The original file descriptor is used usually used for listening for new incoming connections. Servers will be discussed in much more detail in a later lab.

It dequeues the next connection request on the queue for this socket of the server. If queue is empty, this function blocks until a connection request arrives. (read the reference book *TCP/IP Implementation in C* for more details.)

Again, make sure you type cast to the generic socket address structure

Note that the last parameter is a pointer. You are not specifying the length, the kernel is and returning the value to your application, the same with the ClientAddress. After a connection with a client is established the address of the client must be made available to your server, otherwise how could you communicate back with the client? Therefore, the accept() function call fills in the address structure and length of the address structure for your use. Then accept() returns a new file descriptor, and it is this file descriptor with which you will read and write to the client.

7.) Outline of a TCP Client



Step 1: Create a socket : Same as in the server.

Step 2: Binding a socket: This is unnecessary for a client, what bind does is (and will be discussed indetail in the server section) is associate a port number to the application. If you skip this step with a TCP client, a temporary port number is automatically assigned, so it is just better to skip this step with the client.

Step 3: Connecting to a Server:

int connect(int socket_file_descriptor, const struct sockaddr *ServerAddress, socklen_t AddressLength);

Once you have created a socket and have filled in the address structure of the server you want to connect to, the next thing to do is to connect to that server. This is done with the connect function listed above.

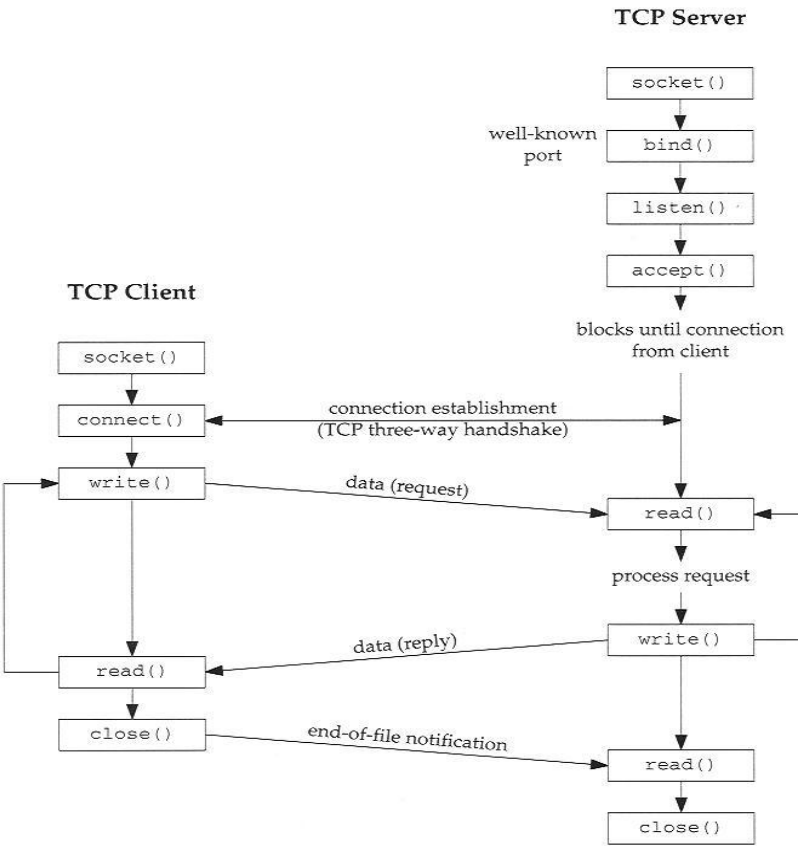
**This is one of the socket functions which requires an address structure so remember to type cast it to the generic socket structure when passing it to the second argument **

Connect performs the three-way handshake with the server and returns when the connection is established or an error occurs.

Once the connection is established you can begin reading and writing to the socket.

Step 4: Read and Writing to the socket will be discussed shortly
Step 5: Closing the socket will be discussed shortly

8.) Outline of a client-server network interaction:



Communication of 2 pairs via sockets necessitates existence of this 4-tuple:

- Local IP address
- Local Port#
- Foreign IP address
- Foreign Port#

!!!! When a server receives (accepts) the client's connection request => it *fork*s a copy of itself and lets the child handle the client. (make sure you remember these Operating Systems concepts)

Therefore on the server machine, listening socket is distinct from the connected socket.

read/write:These are the same functions you use with files but you can use them with sockets as well.However, it is extremely important you understand how they work so please read Stevens carefully to get a full understanding.

Writing to a socket:

int write(int file_descriptor, const void * buf, size_t message_length);

The return value is the number of bytes written, and –1 for failure. The number of bytes written may be less than the message_length. What this function does is transfer the data from you application to a buffer in the kernel on your machine, it does not directly transmit the data over the network. This is extremely important to understand otherwise you will end up with many headaches trying to debug your programs.

TCP is in complete control of sending the data and this is implemented inside the kernel. Due to network congestion or errors, TCP may not decide to send your data right away, even when the function call returns. TCP has an elaborate sliding window mechanism which you will learn about in class to control the rate at which data is sent. Read pages 48-49, 77-78 in Stevens very carefully.

Reading from a socket:

int read(int file_descriptor, char *buffer, size_t buffer_length);

The value returned is the number of bytes read which may not be buffer_length! It returns –1 for failure. As with write(), read() only transfers data from a buffer in the kernel to your application , you are not directly reading the byte stream from the remote host, but rather TCP is in control and buffers the data for your application.

Shutting down sockets:

After you are finished reading and writing to your socket you most call the close system call on the socket file descriptor just as you do on a normal file descriptor otherwise you waste system resources.

The close() function: int close(int filedescriptor);

The shutdown() function: You can also shutdown a socket in a partial way which is often used when forking off processes. You can shutdown the socket so that it won't send anymore or you could also shutdown the socket so that it won't read anymore as well. This function is not so important now but will be discussed in detail later. You can look at the man pages for a full description of this function.

12.) Summary of Functions

For specific and up-to-date information about each of the following functions, please use the online man pages and Steven's Unix Network Programming Vol. I.

Socket creation and destruction:
- socket()
- close()
- shutdown()

Client:
- connect()
- bind()

Server:
- accept()
- bind()
- listen()

Data Transfer:
- send()
- recv()
- write()
- read()

Miscellaneous:
- bzero()
- memset()

Host Information:
- uname()
- gethostbyname()
- gethostbyaddr()

Address Conversion:
- inet_aton()
- inet_addr()
- inet_ntoa()

Experiment No.: 2

Aim: Write a programs in C: hello_client (The server listens for, and accepts, a single TCP connection; it reads all the data it can from that connection, and prints it to the screen; then it closes the connection)

/* CLIENT PROGRAM FOR TCP CONNECTION */

```
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
void func(int sockfd)
{
   char buff[MAX];
   int n;
   for (;;) {
      bzero(buff, sizeof(buff));
      printf("Enter the string : ");
      n = 0;
      while ((buff[n++] = getchar()) != '\n')
         ;
      write(sockfd, buff, sizeof(buff));
      bzero(buff, sizeof(buff));
      read(sockfd, buff, sizeof(buff));
      printf("From Server : %s", buff);
      if ((strncmp(buff, "exit", 4)) == 0) {
         printf("Client Exit...\n");
         break;
      }
   }
}

int main()
{
   int sockfd, connfd;
   struct sockaddr_in servaddr, cli;

   // socket create and varification
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
    if (sockfd == -1) {
        printf("socket creation failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully created..\n");
    bzero(&servaddr, sizeof(servaddr));

    // assign IP, PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(PORT);

    // connect the client socket to server socket
    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
        printf("connection with the server failed...\n");
        exit(0);
    }
    else
        printf("connected to the server..\n");

    // function for chat
    func(sockfd);

    // close the socket
    close(sockfd);
}
```

Compilation –

Server side:
gcc server.c -o server
./server

Client side:
gcc client.c -o client
./client

Output –
Server side:

```
Socket successfully created..

Socket successfully binded..
```

```
Server listening..
server acccept the client...
From client: hi
    To client : hello
From client: exit
    To client : exit
Server Exit...
```

Client side:

```
Socket successfully created..
connected to the server..
Enter the string : hi
From Server : hello
Enter the string : exit
From Server : exit
Client Exit...
```

Resources

Text Books:

1. Unix n/w programming, Stevens

Reference Websites:

1.http://beej.us/guide/bgipc/output/html/multipage/unixsock.html

2.http://en.wikipedia.org/wiki/Berkeley_sockets

3. https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/

Experiment No.: 3

Aim: Write a programs in C: hello_server for TCP
(The client connects to the server, sends the string “Hello, world!”, then closes the connection)

//server

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr

// Function designed for chat between client and server.
void func(int sockfd)
{
   char buff[MAX];
   int n;
   // infinite loop for chat
   for (;;) {
      bzero(buff, MAX);

      // read the message from client and copy it in buffer
      read(sockfd, buff, sizeof(buff));
      // print buffer which contains the client contents
      printf("From client: %s\t To client : ", buff);
      bzero(buff, MAX);
      n = 0;
      // copy server message in the buffer
      while ((buff[n++] = getchar()) != '\n')
         ;

      // and send that buffer to client
      write(sockfd, buff, sizeof(buff));

      // if msg contains "Exit" then server exit and chat ended.
      if (strncmp("exit", buff, 4) == 0) {
         printf("Server Exit...\n");
         break;
      }
```

```
    }
}

// Driver function
int main()
{
    int sockfd, connfd, len;
    struct sockaddr_in servaddr, cli;

    // socket create and verification
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("socket creation failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully created..\n");
    bzero(&servaddr, sizeof(servaddr));

    // assign IP, PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    // Binding newly created socket to given IP and verification
    if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
        printf("socket bind failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully binded..\n");

    // Now server is ready to listen and verification
    if ((listen(sockfd, 5)) != 0) {
        printf("Listen failed...\n");
        exit(0);
    }
    else
        printf("Server listening..\n");
    len = sizeof(cli);

    // Accept the data packet from client and verification
    connfd = accept(sockfd, (SA*)&cli, &len);
    if (connfd < 0) {
        printf("server acccept failed...\n");
        exit(0);
```

```
    }
    else
        printf("server acccept the client...\n");

    // Function for chatting between client and server
    func(connfd);

    // After chatting close the socket
    close(sockfd);
}
```

Compilation –

Server side:
gcc server.c -o server
./server

Client side:
gcc client.c -o client
./client

Output –
Server side:

```
Socket successfully created..
Socket successfully binded..
Server listening..
server acccept the client...
From client: hi
     To client : hello
From client: exit
     To client : exit
Server Exit...
```

Client side:

```
Socket successfully created..
connected to the server..
```

```
Enter the string : hi
From Server : hello
Enter the string : exit
From Server : exit
Client Exit...
```

Resources

Text Books:

1. Unix n/w programming, Stevens

Reference Websites:

1.http://beej.us/guide/bgipc/output/html/multipage/unixsock.html

2.http://en.wikipedia.org/wiki/Berkeley_sockets

3. https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/

Experiment No.: 4

Aim: Write a program to implement TCP Chat Server.

// Program for chatappserver.c

```
#include<sys/socket.h>
#include<sys/types.h>
#include<stdio.h>
#include<arpa/inet.h>
#include<netinet/in.h>
#include<string.h>
#include<unistd.h>
#define SER_PORT 1200
int main()
{
int a,sersock,newsock,n;
char str[25],str2[25];
struct sockaddr_in seraddr;
struct sockaddr_in cliinfo;
socklen_t csize=sizeof(cliinfo);
seraddr.sin_family=AF_INET;
seraddr.sin_port=htons(SER_PORT);
seraddr.sin_addr.s_addr=htonl(INADDR_ANY);
if((sersock=socket(AF_INET,SOCK_STREAM,0))<0)
```

```
{
error("\n socket");
exit(0);
}
if(bind(sersock,(struct sockaddr *)&seraddr,sizeof(seraddr))<0)
{
error("\nBIND");
exit(0);
}
if(listen(sersock,1)<0)
{
error("\n LISTEN");
}
if((newsock=accept(sersock,(struct sockaddr *)&cliinfo,&csize))<0)
{
error("\n ACCEPT");
exit(0);
}
else
printf("\n now connected to %s\n",inet_ntoa(cliinfo.sin_addr));
read(newsock,str,sizeof(str));
do
{
```

```
printf("\n client msg:%s",str);
printf("\n server msg:");
scanf("%s",str2);
write(newsock,str2,sizeof(str2));
listen(newsock,1);
read(newsock,str,sizeof(str));
n=strcmp(str,"BYE");
a=strcmp(str2,"BYE");
}
while(n!=0||a!=0);
close(newsock);
close(sersock);
return 0;
}
```

// Program for chatappclient.c

```
#include<stdio.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<arpa/inet.h>
#include<netinet/in.h>
#include<unistd.h>
#define SER_PORT 1200
```

```
int main(int count,char*arg[])
{
int a,clisock;
char str[20],str2[20];
struct sockaddr_in cliaddr;
cliaddr.sin_port=htons(SER_PORT);
cliaddr.sin_family=AF_INET;
cliaddr.sin_addr.s_addr=inet_addr(arg[1]);
clisock=socket(AF_INET,SOCK_STREAM,0);
if(clisock<0)
{
perror("\n SOCKET");
exit(0);
}
if(connect(clisock,(struct sockaddr*)&cliaddr,sizeof(cliaddr))<0)
{
perror("\n CONNECT");
exit(0);
}
printf("\nclient connected to %s",arg[1]);
printf("\nCLIENT");
scanf("%s",&str);
if(write(clisock,str,sizeof(str))<0)
```

```
{
printf("\n data could not be sent");
}
do
{
listen(clisock,1);
read(clisock,str2,sizeof(str2));
printf("\nserver msg:%s",str2);
printf("\nclient msg:");
scanf("%s",&str);
a=strcmp(str2,"BYE");
write(clisock,str2,sizeof(str2));
}
while(a!=0);
close(clisock);
return 0;
}
```

Resources:

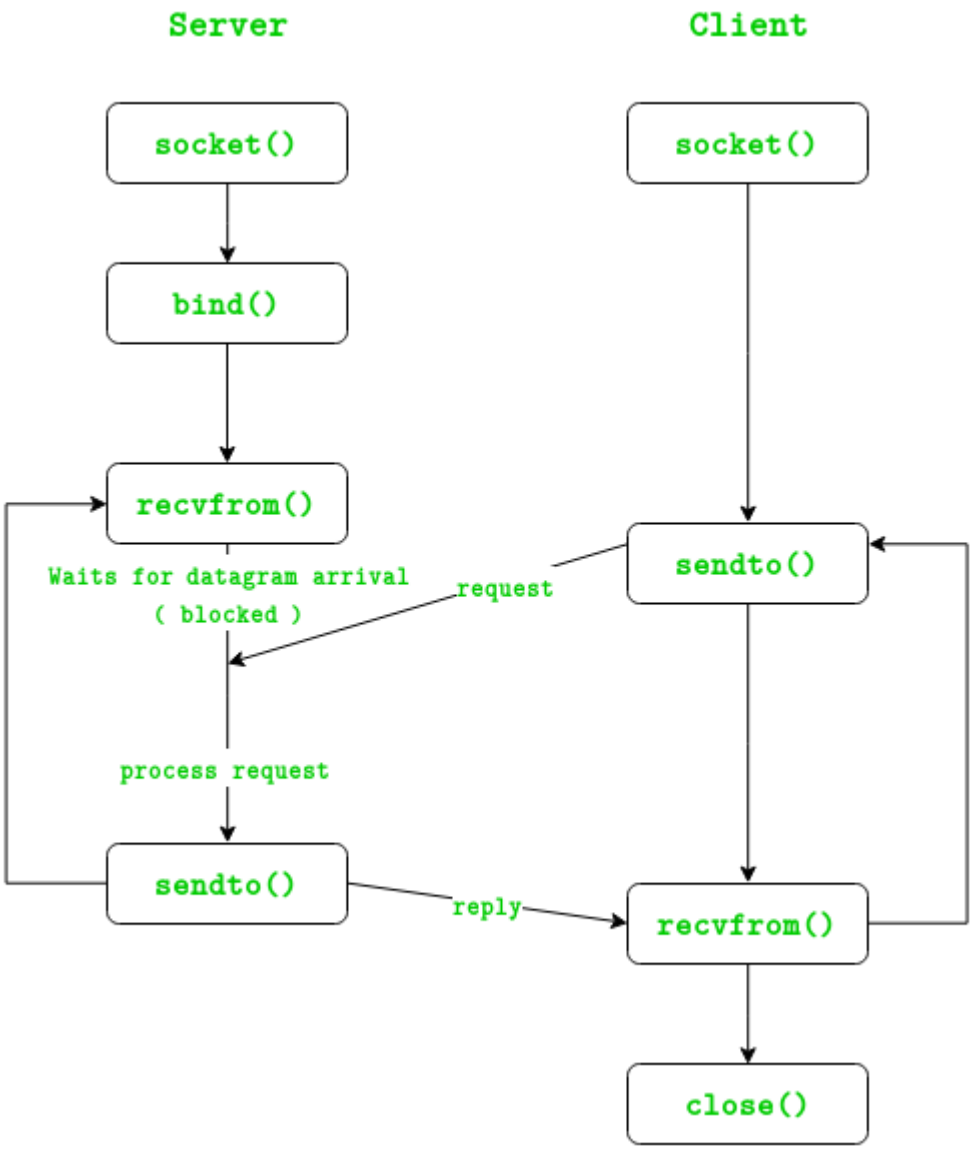
1. https://forgetcode.com/C/1202-Chat-app-TCP

Experiment No.: 5

Aim: Write a programs in C: hello_client (The server listens for, and accepts, a single UDP connection; it reads all the data it can from that connection, and prints it to the screen; then it closes the connection)

Theory

In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of sender which the server uses to send data to the correct client.



The entire process can be broken down into following steps :

UDP Server :

1. Create UDP socket.
2. Bind the socket to server address.
3. Wait until datagram packet arrives from client.
4. Process the datagram packet and send a reply to client.
5. Go back to Step 3.

UDP Client :

1. Create UDP socket.
2. Send message to server.
3. Wait until response from server is recieved.
4. Process reply and go back to step 2, if necessary.
5. Close socket descriptor and exit.

Necessary Functions :

int socket(int domain, int type, int protocol)

Creates an unbound socket in the specified domain.

Returns socket file descriptor.

Arguments :
domain – Specifies the communication
domain (AF_INET for IPv4/ AF_INET6 for IPv6)
type – Type of socket to be created
(SOCK_STREAM for TCP / SOCK_DGRAM for UDP)
protocol – Protocol to be used by socket.
0 means use default protocol for the address family.

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

Assigns address to the unbound socket.

Arguments :
sockfd – File descriptor of socket to be binded
addr – Structure in which address to be binded to is specified
addrlen – Size of *addr* structure

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,

const struct sockaddr *dest_addr, socklen_t addrlen)

Send a message on the socket

Arguments :
sockfd – File descriptor of socket
buf – Application buffer containing the data to be sent
len – Size of *buf* application buffer
flags – Bitwise OR of flags to modify socket behaviour
dest_addr – Structure containing address of destination
addrlen – Size of *dest_addr* structure

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,

struct sockaddr *src_addr, socklen_t *addrlen)

Receive a message from the socket.

Arguments :
sockfd – File descriptor of socket
buf – Application buffer in which to receive data
len – Size of *buf* application buffer
flags – Bitwise OR of flags to modify socket behaviour
src_addr – Structure containing source address is returned
addrlen – Variable in which size of *src_addr* structure is returned

int close(int fd)

Close a file descriptor

Arguments :
fd – File descriptor
In the below code, exchange of one hello message between server and client is shown to demonstrate the model.

```
// Client side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT     8080
#define MAXLINE 1024

// Driver code
int main() {
    int sockfd;
    char buffer[MAXLINE];
    char *hello = "Hello from client";
    struct sockaddr_in     servaddr;

    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }
```

```
    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;

    int n, len;

    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr,
            sizeof(servaddr));
    printf("Hello message sent.\n");

    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                MSG_WAITALL, (struct sockaddr *) &servaddr,
                &len);
    buffer[n] = '\0';
    printf("Server : %s\n", buffer);

    close(sockfd);
    return 0;
}
```

Output :

$ ./server

Client : Hello from client

Hello message sent.

$ ./client

Hello message sent.

Server : Hello from server

Text Books:

1. Unix n/w programming, Stevens

Reference Websites:

1.http://beej.us/guide/bgipc/output/html/multipage/unixsock.html

2.http://en.wikipedia.org/wiki/Berkeley_sockets

3. https://www.geeksforgeeks.org/udp-server-client-implementation-c/

Experiment No.: 6

Aim: Write a programs in C: hello_server
(The client connects to the server, sends the string “Hello, world!”, then closes the UDP connection)

```
// Server side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT     8080
#define MAXLINE 1024

// Driver code
int main() {
    int sockfd;
    char buffer[MAXLINE];
    char *hello = "Hello from server";
    struct sockaddr_in servaddr, cliaddr;

    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));

    // Filling server information
    servaddr.sin_family    = AF_INET; // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT);

    // Bind the socket with the server address
    if ( bind(sockfd, (const struct sockaddr *)&servaddr,
            sizeof(servaddr)) < 0 )
    {
        perror("bind failed");
```

```
        exit(EXIT_FAILURE);
    }

    int len, n;
    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                MSG_WAITALL, ( struct sockaddr *) &cliaddr,
                &len);
    buffer[n] = '\0';
    printf("Client : %s\n", buffer);
    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            len);
    printf("Hello message sent.\n");

    return 0;
}
```

Output :

```
$ ./server

Client : Hello from client

Hello message sent.

$ ./client

Hello message sent.

Server : Hello from server
```

Resources

Text Books:

1. Unix n/w programming, Stevens

Reference Websites:

1.http://beej.us/guide/bgipc/output/html/multipage/unixsock.html

2.http://en.wikipedia.org/wiki/Berkeley_sockets

3. https://www.geeksforgeeks.org/udp-server-client-implementation-c/

Experiment No.: 7

Aim: Write a program for UDP Chat Server

/* udpserver.c */

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

int main()
{
int sock;
int addr_len, bytes_read;
char recv_data[1024];
struct sockaddr_in server_addr , client_addr;


if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
perror("Socket");
exit(1);
        }

        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(5000);
        server_addr.sin_addr.s_addr = INADDR_ANY;
bzero(&(server_addr.sin_zero),8);


if (bind(sock,(struct sockaddr *)&server_addr,sizeof(struct sockaddr))==-1)
        {
perror("Bind");
exit(1);
        }

        addr_len = sizeof(struct sockaddr);
```

```
        printf("\nUDPServer Waiting for client on port 5000");
fflush(stdout);

while (1)
{

 bytes_read = recvfrom(sock,recv_data,1024,0,(struct sockaddr *)&client_addr, &addr_len);


          recv_data[bytes_read] = '\0';

printf("\n(%s,%d)said:",inet_ntoa(client_addr.sin_addr),ntohs(client_addr.sin_port));
printf("%s", recv_data);
        fflush(stdout);

      }
return 0;
}
```

/* CLIENT PROGRAM FOR UDP CONNECTION */

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

int main()
{
int sock;
struct sockaddr_in server_addr;
struct hostent *host;
char send_data[1024];

host= (struct hostent *) gethostbyname((char *)"127.0.0.1");


if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
perror("socket");
```

```
exit(1);
}

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(5000);
server_addr.sin_addr = *((struct in_addr *)host->h_addr);
bzero(&(server_addr.sin_zero),8);

while (1)
  {

printf("Type Something (q or Q to quit):");
gets(send_data);

if ((strcmp(send_data , "q") == 0) || strcmp(send_data , "Q") == 0)
break;

else
sendto(sock, send_data, strlen(send_data), 0,
          (struct sockaddr *)&server_addr, sizeof(struct sockaddr));

  }

}
```

Resources

Text Books:

1. Unix n/w programming, Stevens

Reference Websites:

1.http://beej.us/guide/bgipc/output/html/multipage/unixsock.html

2.http://en.wikipedia.org/wiki/Berkeley_sockets

Experiment No.: 8

Aim: Write an Echo_server using TCP to estimate the round trip time
from client to the server. The server should be such that it can accept multiple connections at any given time , with multiplexed I/O operations

Code for Client (For echo server)

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

#define MAXCOUNT 1024

int main(int argc, char* argv[])
{
int sfd;
char msg[MAXCOUNT];
char blanmsg[MAXCOUNT];
struct sockaddr_in saddr;

memset(&saddr,0,sizeof(saddr));
sfd = socket(AF_INET,SOCK_STREAM,0);
   saddr.sin_family = AF_INET;
   inet_pton(AF_INET,"127.0.0.1",&saddr.sin_addr);
   saddr.sin_port = htons(5004);

connect(sfd,(struct sockaddr*) &saddr, sizeof(saddr));
for(; ;) {
memset(msg,0,MAXCOUNT);
memset(blanmsg,0,MAXCOUNT);
fgets(msg,MAXCOUNT,stdin);
send(sfd,msg,strlen(msg),0);
recv(sfd,blanmsg,sizeof(blanmsg),0);
printf("%s",blanmsg);
fflush(stdout);
   }
exit(0);
}
```

Resources

Text Books:

1. Unix n/w programming, Stevens

Reference Websites:

1.http://beej.us/guide/bgipc/output/html/multipage/unixsock.html

2.http://en.wikipedia.org/wiki/Berkeley_sockets

Experiment No.: 9

Aim: Write an Echo_server using TCP to estimate the round trip time
from client to the server. The server should be such that it can accept multiple connections at any given time , with multiplexed I/O operations

Here is the code for the server:

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

#define MAXCOUNT 1024

int main(int argc, char* argv[])
{
int sfd,nsfd,n,i,cn;
char buf[MAXCOUNT];
socklen_t caddrlen;
struct sockaddr_in caddr,saddr; //Structs for Client and server Address in the Internet

sfd = socket(AF_INET,SOCK_STREAM,0);
memset(&saddr,0,sizeof(saddr)); //Clear the Server address structure

   saddr.sin_family = AF_INET; //Internet Address Family
   saddr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
   saddr.sin_port = htons(5004);

bind(sfd, (struct sockaddr*) &saddr,sizeof(saddr));
listen(sfd,1);

for(; ;) {
caddrlen = sizeof(caddr);
nsfd = accept(sfd,(struct sockaddr*) &caddr,&caddrlen);
cn = recv(nsfd,buf,sizeof(buf),0);
if(cn == 0) {
exit(0);
      }
```

Resources

Text Books:

1. Unix n/w programming, Stevens

Reference Websites:

1.http://beej.us/guide/bgipc/output/html/multipage/unixsock.html

2.http://en.wikipedia.org/wiki/Berkeley_sockets

Experiment No.: 10

Aim: Write a programs to show the functioning of fork() system call.

Algorithm:

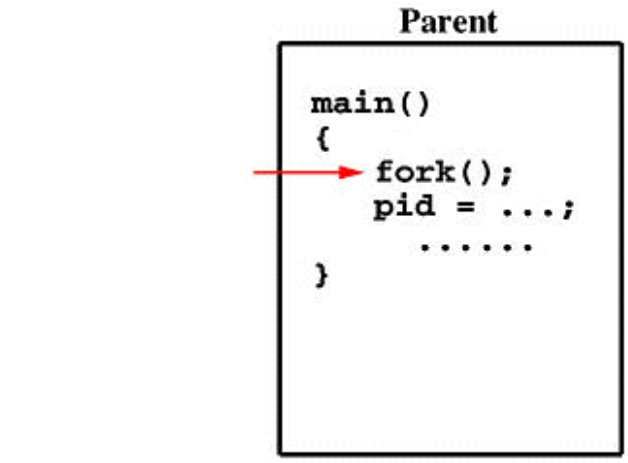
System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork()**, a simple test can tell which process is the child. **Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces**.
Let us take an example to make the above points clear. This example does not distinguish parent and the child processes.

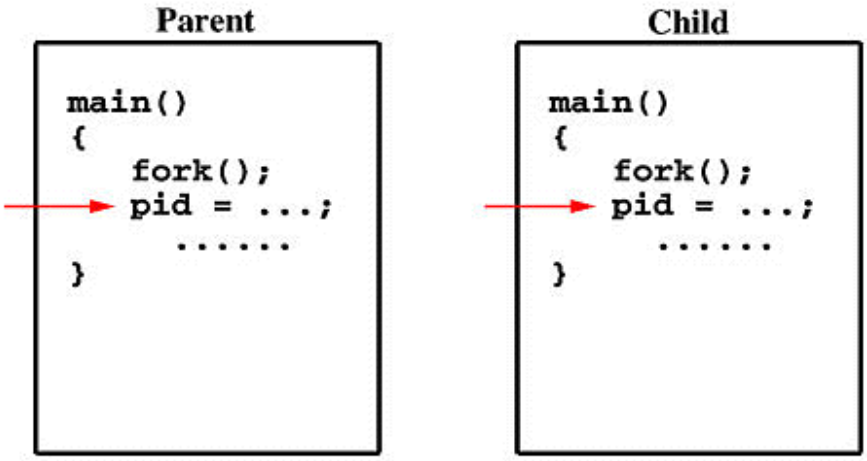
```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
    pid_t  pid;
int    i;
char   buf[BUF_SIZE];

fork();
pid = getpid();
for (i = 1; i <= MAX_COUNT; i++) {
sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
write(1, buf, strlen(buf));
    }
}
```

Suppose the above program executes up to the point of the call to **fork()** (marked in red color):



If the call to **fork()** is executed successfully, Unix will

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork()** call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call **fork()**. Since both processes have identical but separate address spaces, those variables initialized **before** the **fork()** call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by **fork()** calls will not be affected even though they have identical variable names.

What is the reason of using **write** rather than **printf**? It is because **printf()** is "buffered," meaning **printf()** will group the output of a process together. While buffering the output for the parent process, the child may also use **printf** to print out some information, which will also be buffered. As a result, since the output will not be send to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the "unbuffered" **write**.

If you run this program, you might see the following on the screen:

.....
This line is from pid 3456, value 13
This line is from pid 3456, value 14
.....
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
.....
This line is from pid 3456, value 21
This line is from pid 3456, value 22
.....

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

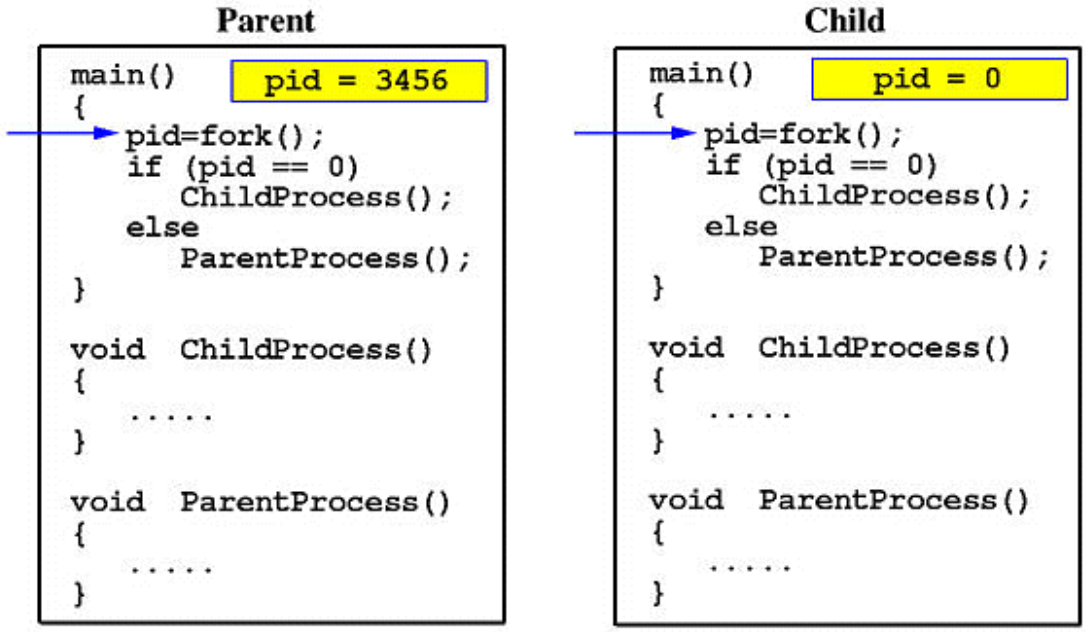
Consider one more simple example, which distinguishes the parent from the child. Click **here** to download this file **fork-02.c**.

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

#define   MAX_COUNT  200

void  ChildProcess(void);              /* child process prototype  */
void  ParentProcess(void);             /* parent process prototype */

void  main(void)
{
    pid_t  pid;

pid = fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}

void  ChildProcess(void)
{
int   i;

for (i = 1; i <= MAX_COUNT; i++)
printf("   This line is from child, value = %d\n", i);
printf("   *** Child process is done ***\n");
}

void  ParentProcess(void)
```

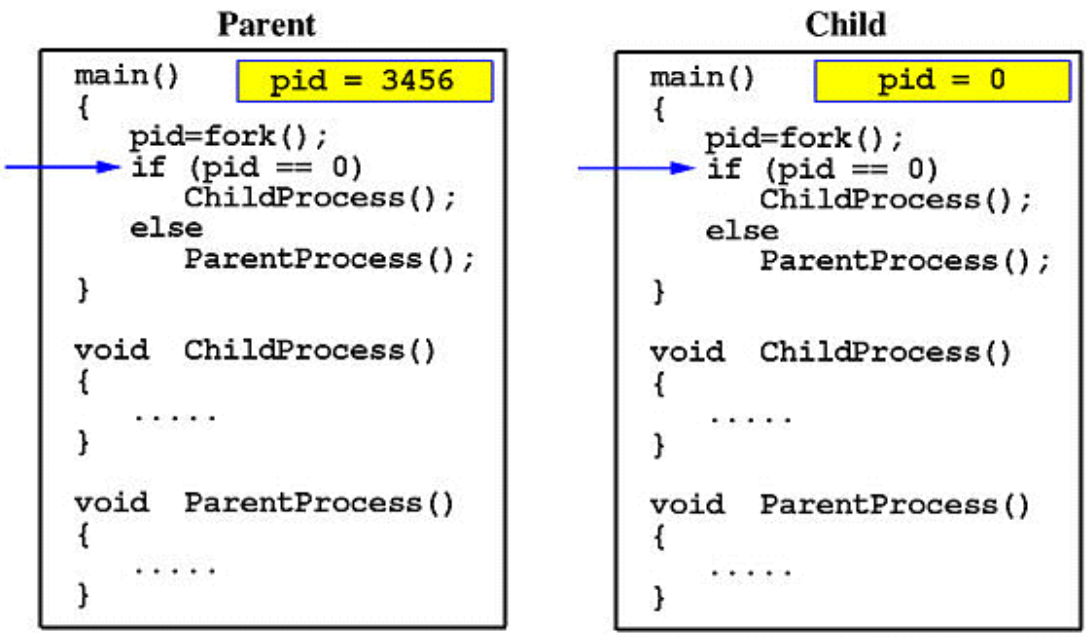
```
{
int   i;

for (i = 1; i <= MAX_COUNT; i++)
printf("This line is from parent, value = %d\n", i);
printf("*** Parent is done ***\n");
}
```

In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable**i**. For simplicity, **printf()** is used.

When the main program executes **fork()**, an identical copy of its address space, including the program and all data, is created. System call **fork()** returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable **pid**. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



Now both programs (*i.e.*, the parent and child) will execute independent of each other starting at the next statement:



In the parent, since **pid** is non-zero, it calls function **ParentProcess()**. On the other hand, the child has a zero **pid** and calls **ChildProcess()** as shown below:

Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process. Therefore, the value of **MAX_COUNT** should be large enough so that both processes will run for at least two or more time quanta. If the value of **MAX_COUNT** is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.

Resources

Text Books:

1. Unix n/w programming, Stevens

Reference Websites:

1.http://beej.us/guide/bgipc/output/html/multipage/unixsock.html

2.http://en.wikipedia.org/wiki/Berkeley_sockets

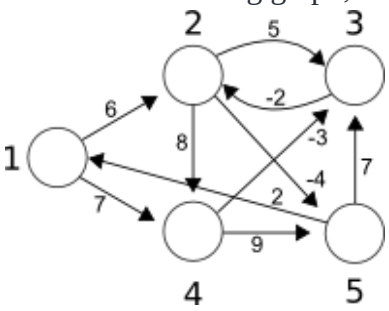
Experiment No.: 11

Aim: Program to simulate Bellman Ford Routing Algorithm

Algorithm:

The Problem

Given the following graph, calculate the length of the shortest path from **node 1** to **node 2**.



It's obvious that there's a direct route of length *6*, but take a look at path: *1 -> 4 -> 3 -> 2*. The length of the path is *7 – 3 – 2 = 2*, which is less than *6*. BTW, you don't need negative edge weights to get such a situation, but they do clarify the problem.

This also suggests a property of shortest path algorithms: to find the shortest path form *x*to *y*, you need to know, beforehand, the shortest paths to *y*'s neighbours. For this, you need to know the paths to *y*'s neighbours' neighbours… In the end, you must calculate the shortest path to the connected component of the graph in which *x* and *y* are found.

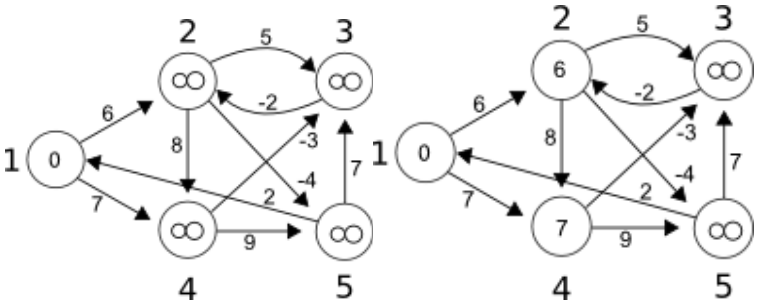
That said, you usually calculate **the shortest path to all nodes** and then pick the ones you're intrested in.

The Algorithm

The Bellman-Ford algorithm is one of the classic solutions to this problem. It calculates the shortest path to all nodes in the graph from a single source.

The basic idea is simple:
Start by considering that the shortest path to all nodes, less the source, is infinity. Mark the length of

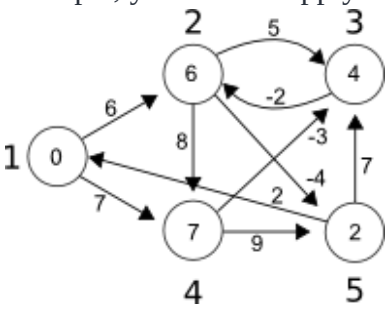
the path to the source as *0*:

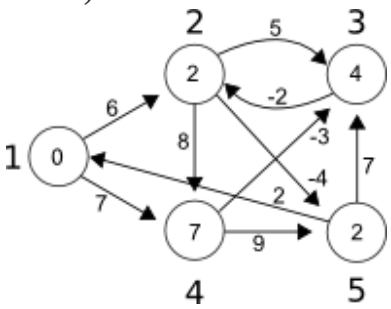


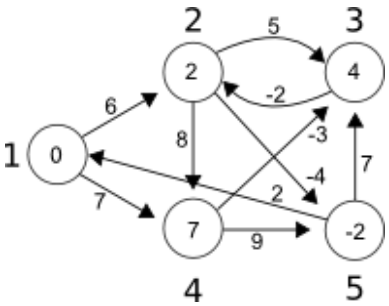
Take every edge and try to *relax* it:

Relaxing an edge means checking to see if the path to the node the edge is pointing to can't be shortened, and if so, doing it. In the above graph, by checking the **edge 1 -> 2** of length *6,* you find that the length of the shortest path to **node 1** plus the length of the**edge 1 -> 2** is less then infinity. So, you replace infinity in **node 2** with *6*. The same can be said for edge *1 -> 4* of length *7.* It's also worth noting that, practically, you can't relax the edges whose start has the shortest path of length infinity to it.

Now, you apply the previous step *n – 1* times, where n is the number of nodes in the graph. In this example, you have to apply it *4* times (that's *3* more times).







Here, **d[i]** is the shortest path to node **i**, **e** is the number of edges and **edges[i]** is the **i**-th edge.

It may not be obvious why this works, but take a look at what is certain after each step. After the first step, any path made up of at most *2* nodes will be optimal. After the step *2*, any path made up of at most *3* nodes will be optimal… After the *(n – 1)*-th step, any path made up of at most *n* nodes will be optimal.

The Programme

The following programme just puts the **bellman_ford** function into context. It runs in**O(VE)** time, so for the example graph it will do something on the lines of **5 * 9 = 45**relaxations. Keep in mind that this algorithm works quite well on graphs with few edges, but is very slow for dense graphs (graphs with almost n^2 edges)

```
#include <stdio.h>

typedef struct {
	int u, v, w;
} Edge;

int n; /* the number of nodes */
int e; /* the number of edges */
Edge edges[1024]; /* large enough for n <= 2^5=32 */
int d[32]; /* d[i] is the minimum distance from node s to node i */

#define INFINITY 10000

void printDist() {
	int i;

	printf("Distances:\n");

	for (i = 0; i < n; ++i)
		printf("to %d\t", i + 1);
	printf("\n");

	for (i = 0; i < n; ++i)
		printf("%d\t", d[i]);

	printf("\n\n");
}
```

```
void bellman_ford(int s) {
	int i, j;

	for (i = 0; i < n; ++i)
		d[i] = INFINITY;

	d[s] = 0;

	for (i = 0; i < n - 1; ++i)
		for (j = 0; j < e; ++j)
			if (d[edges[j].u] + edges[j].w < d[edges[j].v])
				d[edges[j].v] = d[edges[j].u] + edges[j].w;
}

int main(int argc, char *argv[]) {
	int i, j;
	int w;

	FILE *fin = fopen("dist.txt", "r");
	fscanf(fin, "%d", &n);
	e = 0;

	for (i = 0; i < n; ++i)
		for (j = 0; j < n; ++j) {
			fscanf(fin, "%d", &w);
			if (w != 0) {
				edges[e].u = i;
				edges[e].v = j;
				edges[e].w = w;
				++e;
			}
		}
	fclose(fin);

	/* printDist(); */

	bellman_ford(0);

	printDist();

	return 0;
```

}

And here's the input file used in the example (dist.txt):

```
5
0 6 0 7 0
0 0 5 8 -4
0 -2 0 0 0
0 0 -3 9 0
2 0 7 0 0
```

Experiment No.: 12

Aim: Program to simulate of sliding window protocol.

Algorithm: Step 1: Start the program.
Step 2: Include the necessary header files.
Step 3: To create the socket using socket() function.
Step 4: Enter the number of frames.
Step 5: And the corresponding message is send to receiver.
Step 6: Acknowledgement is received by receiver.
Step 7: If u send another message,ACK 2 message is received.
Step 8: Send the acknowledgement to sender.
Step 9: Print out with the necessary details.
Step 10: Stop the program.

SOURCE CODE:

Server:

```
#include<string.h>
#include<stdio.h>
#include<netdb.h>
#include<netinet/in.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<errno.h>
int main(int argc,char ** argv)
{
struct sockaddr_in saddr,caddr;
int r,len,ssid,csid,pid,pid1,i,n;
char wbuffer[1024],rbuffer[1024];
float c;
if(argc<2)
fprintf(stderr,"Port number not specified\n");
ssid=socket(AF_INET,SOCK_STREAM,0);
if(ssid<0)
perror("Socket failed\n");
bzero((char *)&saddr,sizeof(saddr));
saddr.sin_family=AF_INET;
saddr.sin_port=htons(atoi(argv[1]));
saddr.sin_addr.s_addr=INADDR_ANY;
if(bind(ssid,(struct sockaddr *)&saddr,sizeof(saddr))<0)
perror("Socket Bind\n");
listen(ssid,5);
len=sizeof(caddr);
csid=accept(ssid,(struct sockaddr *)&caddr,&len);
```

```
if(csid<0)
perror("Socket Accept\n");
fprintf(stdout,"TYPE MESSAGE TO CLIENT\n");
pid=fork();
if(pid==0)
{
while(1)
{
bzero(rbuffer,1024);
n=read(csid,rbuffer,1024);
if(n==0)
perror("Socket read\n");
else
fprintf(stdout,"MESSAGE FROM CLIENT: %s\n",rbuffer);
}
exit(0);
}
else
{
while(1)
{
bzero(wbuffer,1024);
fgets(wbuffer,1024,stdin);
n=write(csid,wbuffer,1024);
if(n==0)
perror("Socket Write");
}
}
return(0);
}
```

CLIENT:

```
#include<stdio.h>
#include<netdb.h>
#include<netinet/in.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<errno.h>
int main(int argc,char ** argv)
{
struct sockaddr_in saddr;
struct hostent *server;
int n,ssid,csid,pid,pi;
char wbuffer[1024],rbuffer[1024];
char str[15];
if(argc<3)
```

```
fprintf(stderr,"Parameter inadequate\n");
csid=socket(AF_INET,SOCK_STREAM,0);
if(csid<0)
perror("Socket Failed\n");
bzero((char *)&saddr,sizeof(saddr));
server=gethostbyname(argv[1]);
saddr.sin_family=AF_INET;
saddr.sin_port=htons(atoi(argv[2]));
bcopy((char *)server->h_addr,(char *)&saddr.sin_addr.s_addr,server->h_length);
ssid=connect(csid,(struct sockaddr *)&saddr,sizeof(saddr));
if(ssid<0)
perror("Socket Connect\n");
fprintf(stdout,"ENTER MESSAGE TO SERVER:\n");
pid=fork();
if(pid==0)
{
while(1)
{
bzero(wbuffer,1024);
fgets(wbuffer,1024,stdin);
n=write(csid,wbuffer,sizeof(wbuffer));
if(n==0)
perror("Socket Write");
}
exit(0);
}
else
{
while(1)
{
bzero(rbuffer,1024);
n=read(csid,rbuffer,sizeof(rbuffer));
if(n==0)
perror("Socket Read\n");
else
fprintf(stdout,"MESSAGE FROM SERVER: %s\n",rbuffer);
}
return(0);
}
}
```

OUTPUT:

SERVER:

```
[05mecse090@networkserver ~]$ cc chatserv.c
[05mecse090@networkserver ~]$./a.out 9898
```

TYPE MESSAGE TO CLIENT

MESSAGE FROM CLIENT: hi

hai

CLIENT:
[05mecse090@networkserver ~]$ cc chatcli.c
[05mecse090@networkserver ~]$cc chatcli.c
[05mecse090@networkserver ~]$ ./a.out 127.0.0.1
ENTER MESSAGE TO SERVER:
hi
MESSAGE FROM SERVER: hai
RESULT:
Thus the c program for the simulation of sliding window protocol has beenexecuted and the output is verified successfully.

Experiment No.: 13

Aim: Program to simulate file transfer protocol.

Algorithm:
Client
Step 1: start the program
Step 2: Declare the variables and structure for sockets
Step 3: And then get the port number
Step 4: Create a socket using socket functions
Step 5: The socket is binded at the specified port
Step 6: Using the object, the port and address are declared
Step 7: Get the source file and the destination file from the user
Step 8: Use the send command for sending the two strings
Step 9: Receive the bytes sent from the server
Step 10: Print it in the console
Step 11: Close the socket
Server
Step 1: Start the program
Step 2: Declare the variables and structure for sockets
Step 3: And then get the port number
Step 4: Create a socket using socket functions
Step 5: Use the connect command for socket connection
Step 6: Use bind option to bind the socket address
Step 7: Use accept command to receive the connection from the client
Step 8: Receive command from the client
Step 9: Send the file to the client socket
Step 10: Close the connection

PROGRAM:

SERVER:

```
#include<stdio.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<string.h>
#include<sys/socket.h>
int main()
{
int sd,nsd,i,port=1234;
char content[100]="\0",fname[100]="\0";
struct sockaddr_in ser,cli;
FILE *fp;
if((sd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))==-1)
```

```
{
printf("ERROR::SOCKET CREATION PROBLEM--CHECK THE PARAMETERS.\n");
return 0;
}
bzero((char *)&ser,sizeof(ser));
printf("THE PORT ADDRESS IS: %d\n",port);
ser.sin_family=AF_INET;
ser.sin_port=htons(port);
ser.sin_addr.s_addr=htonl(INADDR_ANY);
if(bind(sd,(struct sockaddr *)&ser,sizeof(ser))==-1)
{
printf("\nERROR::BINDING PROBLEM, PORT BUSY--PLEASE CSS IN THE SER AND
CLI\n");
return 0;
}
i=sizeof(cli);
listen(sd,1);
printf("\nSERVER MODULE\n");
printf("**********************\n");
nsd=accept(sd,(struct sockaddr *)&cli,&i);
if(nsd==-1)
{
printf("\nERROR::CLIENT ACCEPTIN PROBLEM--CHECK THE DEIPTOR
PARAMETER.\n\n");
return 0;
}
printf("\nCLIENT ACCEPTED");
i=recv(nsd,fname,30,0);
fname[i]='\0';
fp=fopen(fname,"rb");
while(1)
{
i=fread(&content,1,30,fp);
content[i]='\0';
send(nsd,content,30,0);
strcpy(content,"\0");
if(i<30)
break;
}
send(nsd,"EOF",4,0);
printf("\nFILE TRANSFERED TO DESTINATION\n\n");
fclose(fp);
close(sd);
close(nsd);
return 0;
}
```

CLIENT:

```
#include<stdio.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<string.h>
#include<sys/socket.h>
int main()
{
int sd,i,port=1234;
char content[100]="\0",fname[100]="\0",file[100]="\0";
struct sockaddr_in ser;
FILE *fp;
if((sd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))==-1)
{
printf("\nERROR::SOCKET CREATION PROBLEM--CHECK THE
PARAMETER.\n\n");
return 0;
}
bzero((char *)&ser,sizeof(ser));
printf("\nTHE PORT ADDRESS IS: %d\n",port);
ser.sin_family=AF_INET;
ser.sin_port=htons(port);
ser.sin_addr.s_addr=htonl(INADDR_ANY);
if(connect(sd,(struct sockaddr *)&ser,sizeof(ser))==-1)
{
printf("\nERROR::CANT CONNECT TO SERVER--CHECK PARAMETERS.\n\n");
return 0;
}
printf("\nTHIS IS THE CLIENT MODULE. THIS MODULE CAN ASK THE SERVER A
FILE");
printf("\n**********************************************************************\n\n");
printf("\nENTER THE PATHNAME OF SOURCE FILE::\n");
scanf("%s",fname);
printf("\nENTER THE PATHNAME OF DESTINATION FILE::\n");
scanf("%s",file);
send(sd,fname,30,0);
fp=fopen(file,"wb");
while(1)
{
i=recv(sd,content,30,0);
content[i]='\0';
if(!strcmp(content,"EOF"))
break;
//fwrite(&content,strlen(content),1,fp);
printf("%s",content);
strcpy(content,"\0");
```

```
}
printf("\n\nFILE RECEIVED\n\n");
fclose(fp);
close(sd);
return 0;
}
```

OUTPUT:
SERVER:

```
[3itb41@TELNET ~]$ cd ftpser.c
[3itb41@TELNET serv]$ ./a.out

THE PORT ADDRESS IS: 1234
SERVER MODULE
**********
CLIENT ACCEPTED
FILE TRANSFERED TO DESTINATION
```

CLIENT:

```
[3itb41@TELNET cli]$ cc ftpcli.c
[3itb41@TELNET cli]$ ./a.out
THE PORT ADDRESS IS: 1234
THIS IS THE CLIENT MODULE. THIS MODULE CAN ASK THE SERVER A FILE
**********
ENTER THE PATHNAME OF SOURCE FILE::
/home/3itb41/file1.html
ENTER THE PATHNAME OF DESTINATION FILE::
fp1.html
<HTML>
<BODY>
Hi
</BODY>
</HTML>
FILE RECEIVED
```

RESULT:
Thus the c program for transferring files from one machine to another machine using TCP is executed and the output is verified successfully.

Viva Questions

Q1. What is socket?
Q2. How does the race condition occur?
Q3. What is multiprogramming?
Q4. Name the seven layers of the OSI model?
Q5. What is the difference between TCP and UDP?
Q6. What does socket consists of?
Q7. What is firewall?
Q8. How do I monitor the activity of sockets?
Q9. What is the role of TCP protocol and IP PROTOCOL?
Q10. How should I choose a port number for my server?
Q11. What is DHCP?
Q12. What does routing work?
Q13.What is VPN?
Q14. How do I open a socket?
Q15. How do I create an input stream?
Q16. How do I close a socket?
Q17. What is SMTP?
Q18. What is echo server?
Q19. What this function bind() does?
Q20. What this function socket() does?
Q21. What is IP address?
Q22. What are network host names?
Q23. How to find a machine address?
Q24. Difference between ARP and RARP.
Q25. What is MAC address?
Q26. What is multicasting?
Q27. What is DNS?
Q28. What is RMI?
Q29. How does TCP handshaking works?
Q30. What is sliding window protocol?
Q31. What is the difference between a Null and a void pointer?
Q32. Can I connect two computers to internet using same line?