NATIONAL INSTITUTE OF TECHNOLOGY SILCHAR

Cachar, Assam

B.Tech. Vth Sem

Subject Code: CS-311

Subject Name: Computer Network Laboratory

Submitted By:

Name : Subhojit Ghimire

Sch. Id. : 1912160

Branch : CSE – B

List of Experiments

1. Write a Client-Server socket program to implement “TCP client server” architecture. (Description: Retrieve information (any text or system info) from TCP server to TCP client.)

2. Write a Client-Server socket program to implement “TCP chat server”. (Description: Two client systems connected to one central server for connection establishment, communication / chat have to be done in client machines.)

3. Write an “Echo Client” and “Echo Server” 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. (Description: Multiple clients connected at the same time to one server for connection establishment, server has to listen to all the clients, use I/O monitoring calls if required.)

4. Write an “Echo Client” and “Echo Server” 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. (Description: Multiple clients connected at the same time to one server for connection establishment, server has to listen to all the clients, use I/O monitoring calls if required.)

5. Write a program for “Connectionless Iterative Service” in which the server finds the factorial of a number sent by the client and sends it back.

6. Write a Client-Server socket program to implement “FTP server” using UDP connection. (Description: The file.txt may be there on the server side with some text; on the client request, the server has to send the text file file.txt and corresponding text has to print on client terminal.)

7. Write a program for “Remote Command Execution” using sockets. (Description: The client sends command(s) and data parameters (if required) to the server. The server will execute the command and send back the result to the client. Execute 3 commands.)

8. Write a program to implement “Web Server”. (Description: The Client will be requesting a web page to be accessed which resides at the Server side.)

Q.1.: Write a Client-Server Socket Program to implement "TCP Client Server" architecture. (Description: Retrieve information (any text or system info) from TCP Server to TCP Client.

AIM: TO IMPLEMENT "TCP CLIENT SERVER" ARCHITECTURE - C++

THEORY:

1. SOCKET: It is an Application Programming Interface (API) used for InterProcess Communications (IPC), a well-defined method of connecting two processes locally or across a network.
2. CLIENT-SERVER SOCKET: The server creates a socket, attaches it to a network port addresses, then waits for the client to contact it. The client-to-server data exchange takes place when a client connects to the server through a socket. The server performs the client's request and sends the reply back to the client.
3. TCP CLIENT SERVER: TCP (Transmission Control Protocol) is a transport layer in a networking service. The client in TCP/IP connection is the device that dials the phone and the server is the device that is listening in for calls to come in.

The entire process can be broken down as:

TCP SERVER: (i) Create TCP Socket

(ii) Bind the Socket to server address

(iii) Put the server socket in passive mode and wait for the client to approach.

(iv) When the connection is established, transfe data.

TCP CLIENT: (i) Create TCP Socket
(ii) Connect newly created client socket to server.
(iii) When the connection is established, receive data

CODE:

```
// TCP SERVER

# include <iostream>
# include <cstdlib>
# include <cstring>
# include <netdb.h>
# include <sys/socket.h>
# include <unistd.h>
// # include <netinet/in.h>

#define MAX 80
#define PORT 8080

using namespace std;

void errorExit (int errorSignal) {
        cout << "FAIL: ";
        if (errorSignal == 1)   cout << "SOCKET CREATING\n";
        else if (errorSignal == 2)   cout << "SOCKET BINDING\n";
        else if (errorSignal == 3)   cout << "SERVER LISTENING\n";
        else if (errorSignal == 4)   cout << "SERVER ACCEPTING\n";
        exit (0);
}
```

```
int main () {
    // Socket Creating
    int sockfd = socket (AF_INET, SOCK_STREAM, 0);
    Struct Sockaddr_in servaddr;
    if (sockfd != -1)
            cout << "SUCCESS: SOCKET CREATED" << endl;
    else
            errorExit (1);
    bzero (&servaddr, sizeof (servaddr));

    // Assigning IP and PORT
     servaddr.sin_family  = AF_INET;
     servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
     // [illegible] & servaddr.sin_addr.s_addr = inet_addr ("127.0.0.1");

    // BINDING NEWLY CREATED SOCKET TO GIVEN IP
    if (!(bind (sockfd, (struct sockaddr*)&servaddr,
                                sizeof (servaddr))))
            cout << "SUCCESS: SOCKET BINDED" << endl;
    else
            errorExit (2);

    // Server Ready to Listen
    struct  Sockaddr_in  cli;
    unsigned  int len = sizeof (cli);
    if  (!(listen (sockfd, 5)))
            cout << "SUCCESS: SERVER LISTENING" << endl;
    else
            errorExit (3);
```

```
// ACCEPTING THE DATA PACKET FROM CLIENT
int connfd = accept (sockfd, (struct sockaddr *)&cli,
                                        &len);
if ( connfd >= 0)
        cout << "SUCCESS : CLIENT ACCEPTED "<<endl;
else
        errorExit (4);

// MESSAGE EXCHANGE BETWEEN CLIENT AND SERVER
char buff [MAX];
while (true) {
        bzero (buff, MAX);
        read (connfd, buff, sizeof (buff));
        cout << "CLIENT MESSAGE:  "<<buff;
        bzero (buff, MAX);

        // write server message and send to client
        cout << "WRITE SERVER MESSAGE:  ";
        for (int ii= 0; (buff [ii] = getchar ()) != '\n'; ++ii);
        write (connfd, buff, sizeof (buff));
        if (!strncmp ("exit", buff, 4))
                break;
  }
  cout << "SERVER EXIT " << endl;
  close (sockfd);
  return 0;
}
```

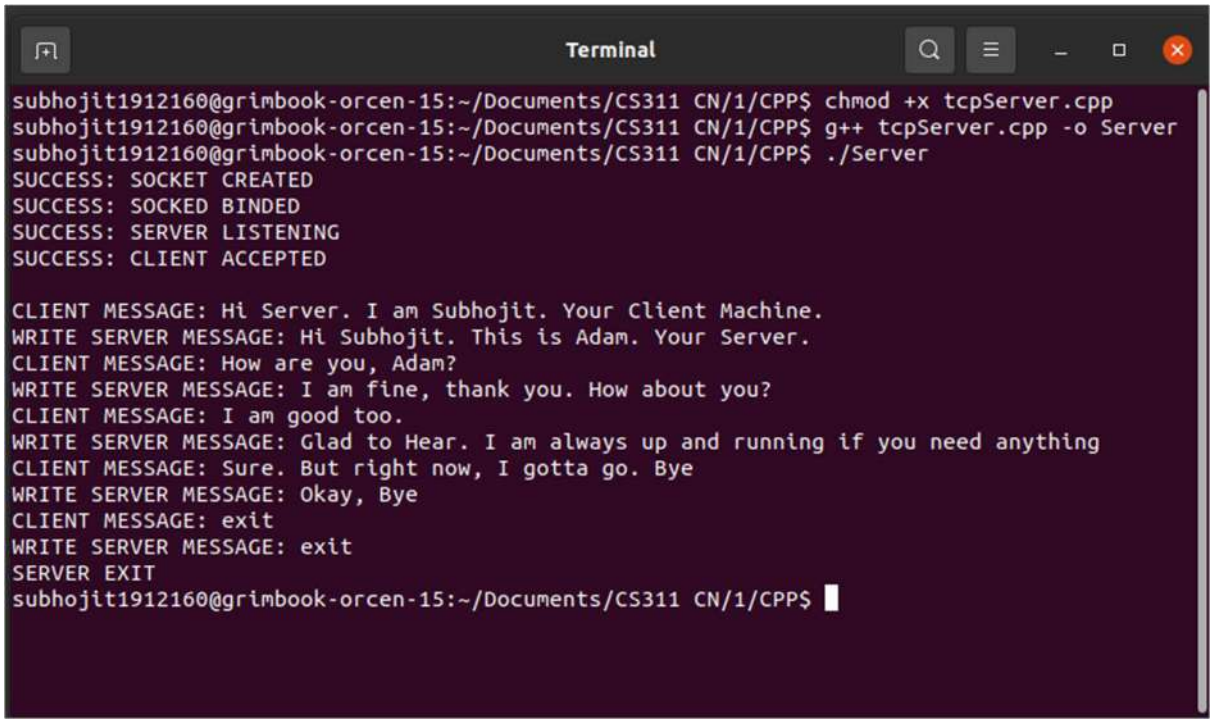
// TCP CLIENT

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <netdb.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define MAX 80
#define PORT 8080

using namespace std;

void errorExit (int errorSignal) {
        cout << "FAIL: ";
        if (errorSignal == 1)   cout << "SOCKET CREATING\n";
        else if (errorSignal == 2)   cout << "SERVER CONNECTION\n";
        exit (0);
}

int main () {
        // SOCKET CREATION
        int sockfd = socket (AF_INE, SOCK_STREAM, 0);
        struct sockaddr_in servaddr;
        if (sockfd != -1)   cout << "SUCCESS: SOCKET CREATED\n";
        else errorExit (1);
        bzero (&servaddr, sizeof (servaddr));
```

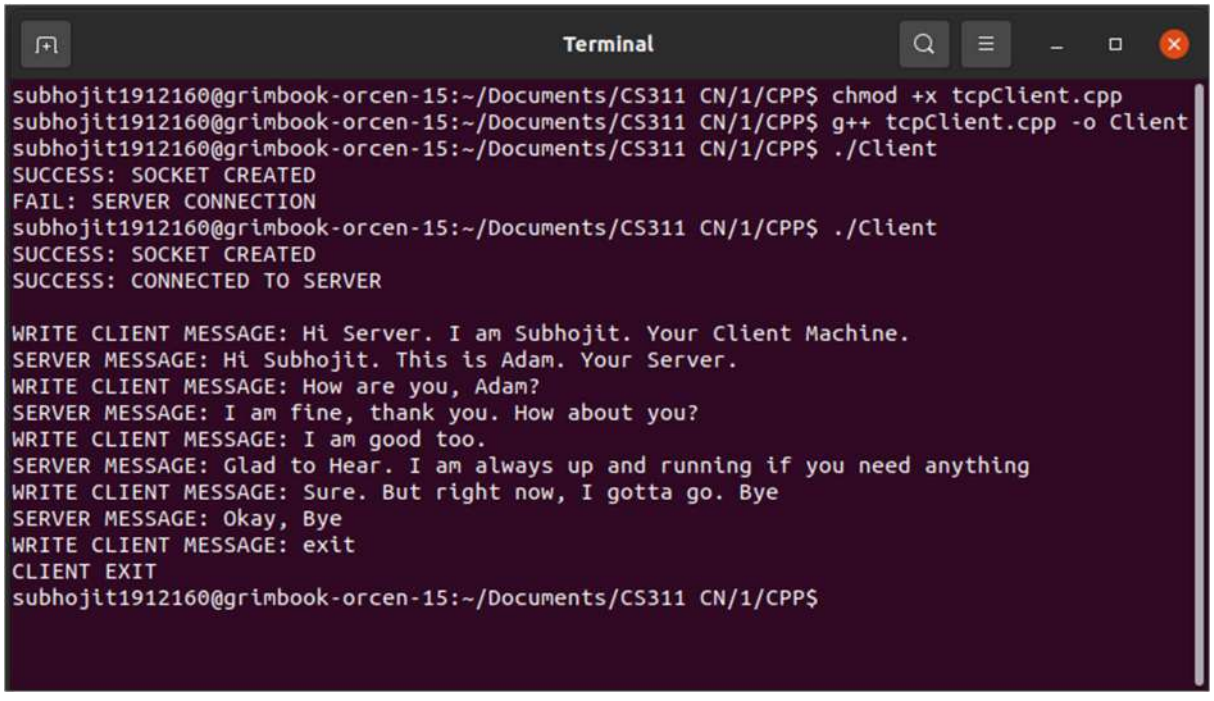
```
// ASSIGNING IP AND PORT
serv2addr.sin-family    = AF_INET;
serv2addr.sin-2ddr.s-2ddr = htonl (INADDR_ANY);
// serv2addr.sin-2ddr.s-2ddr = inet-2ddr ("127.0.0.1");
serv2addr.sin-port    = htons (PORT);

// CONNECTING TO SERVER SOCKET
if (! (connect (sockfd, (struct sock2addr*)&serv2addr,
                                    sizeof (serv2addr))))
        cout << "SUCCESS: CONNECTED TO SERVER \n";
else
        errorExit (2);

// MESSAGE EXCHANGE BETWEEN CLIENT AND SERVER.
char buff [MAX];
while (1) {
        bzero (buff, sizeof (buff));
        cout << "WRITE CLIENT MESSAGE:  ";
        for (int ii=0; (buff[ii]=getchar()) != '\n'; ++ii);
        write (sockfd, buff, sizeof (buff));
        if (!strncmp (buff, "exit", 4));
                break;
        bzero (buff, sizeof (buff));
        read (sockfd, buff, sizeof (buff));
        cout << "SERVER MESSAGE:  " << buff;
}
cout << "CLIENT EXIT \n";
close (sockfd);
return 0;
}
```

OUTPUT:

// TCP SERVER



// TCP CLIENT



Output Explanation:

Firstly, compile and run the TCP Server Program. This way, a server is established for any or all created clients to connect to. If there is no server, Client will fail to connect to any server. After running Server, compile and run the TCP Client Program. The created client will connect to the server. After the connection is successfully established between client and server, message exchange can take place. The messages sent by client will be visible to server, and the server can reply back with its own messages.

Q.20.: Write a Client-server Socket Program to implement "TCP Chat Server". (Description: Two client system connected to one central server for connection establishment, communication / chat have to be done in client machines.)

AIM: TO IMPLEMENT "TCP CHAT SERVER" USING C++

THEORY: 1. SOCKET: It is an Application Programming Interface (API) used for InterProcess Communications (IPC), a well-defined method of connecting two processes locally or across a network.

2. CLIENT-SERVER SOCKET: The server creates a socket, attaches it to a network port addresses, then waits for the client to contact it. The client-to-server data exchange takes place when a client connects to the server through a socket. The server performs the client's request and sends the reply back to the client.

3. TCP CHAT SERVER: Even the widely used messaging application like Snapchat, WhatsApp, Twitter, Instagram etc. use transmission control protocol. In TCP based chats, the exchanging of messages take place using TCP/IP connection layer. There is a secure server that is open to all the clients for connection. Two or more clients can chat with one another by passing message through this server.

The entire process can be broken down as:

TCP SERVER: (i) Create TCP Socket.
(ii) Bind the socket to server address.
(iii) Put the server on passive mode: It will wait for the client to approach the server to make a connection.
(iv) When a connection is established between client and server, wait for client to send message.
(v) Transmit any or all messages received by the server to all the clients currently connected to the server.

TCP CLIENT: (i) Create TCP Socket.
(ii) Connect newly created client socket to server.
(iii) When the connection is established, send and receive data.

CODE:

```
// TCP CHAT SERVER

#include <iostream>
#include <netdb.h>
#include <cstdlib>
#include <cstring>
#include <sys/socket.h>
#include <pthread.h>
#include <unistd.h>
```

```
#define PORT 8080

pthread_mutex_t mutex;
int clients [20];
int nn = 0;

using namespace std;

void sendToAll (char *msg, int curr) {
        pthread_mutex_lock (&mutex);
        for (int ii = 0; ii<nn ; ++ii)
                if (send (clients [ii], msg, strlen(msg),0)
                                < 0) {
                        cout<<" FAIL: MESSAGE SEND\n";
                        continue;
                }
        pthread_mutex_unlock (&mutex);
}

void *recvMsg (void *connfd) {
        int sockfd = *((int *) connfd);
        char msg [1000];
        int len;
        while ((len =recv (sockfd, msg, 1000, 0))>0){
                msg [len] = '\0';
                sendToAll (msg, sockfd);
        }
        return 0;
}
```

```
void errorExit (int errorSignal) {
        cout << "FAIL: ";
        if (errorSignal == 1)
            cout << "SOCKET CREATING \n";
        else if (errorSignal == 2)
            cout << "SOCKET BINDING \n";
        else if (errorSignal == 3)
            cout << "SERVER LISTENING \n";
        else if (errorSignal == 4)
            cout << "SERVER ACCEPTING \n";
        exit (0);
}

int main() {
    // SOCKET CREATION
    int sockfd = socket (AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in servaddr;
    if (sockfd != -1)
        cout << "SUCCESS: SOCKET CREATED \n";
    else
        errorExit (1);
    bzero (&servaddr, sizeof (servaddr));

    // ASSIGNING IP AND PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    // servaddr.sin_addr.s_addr = inet_addr ("127.0.0.1");
    servaddr.sin_port = htons (PORT);
```

```
// Binding NEWLY CREATED SOCKET TO GIVEN IP
if (!(bind (sockfd, (struct sockaddr*) &servaddr,
                    sizeof (servaddr))))
        cout << " SUCCESS: SERVER BINDED \n";
else
        errorExit (2);

// Server ready to listen
if (!listen (sockfd, 20))
        cout<<" SUCCESS: SERVER LISTENING "<<endl;
else    errorExit (3);

// Accepting CLIENTS
pthread_t recvt;
int connfd;
while (1) {
        if ((connfd = accept (sockfd, (struct sockaddr*)
                        NULL, NULL)) >=0) {
                pthread_mutex_lock (&mutex);
                clients [nn++] = connfd;
                // Creating Thread for each Client
                pthread_create (&recvt, NULL, &recvMsg, &connfd);
                pthread_mutex_unlock (&mutex);
        }
        else    errorExit (4);
}
close (sockfd);
return 0;
}
```

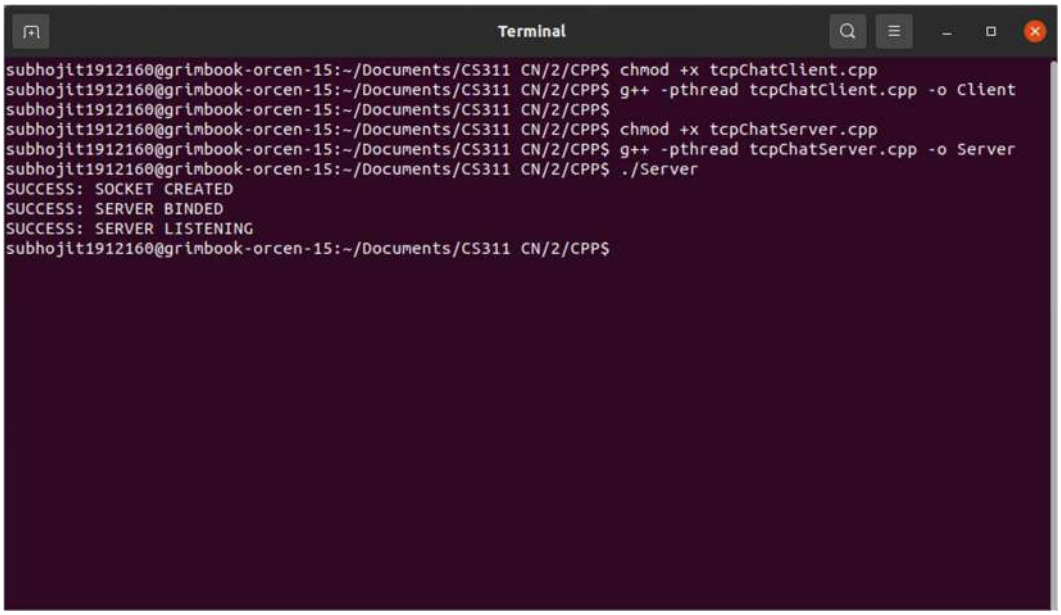
```
//TCP CHAT CLIENT

# include <iostream>
# include <cstdlib>
# include <cstring>
# include <netdb.h>
# include <arpa/inet.h>
# include <pthread.h>
# include <sys/socket.h>
# include <unistd.h>

# define PORT 8080
char msg[1000];

using namespace std;

void *recvMsg (void *connfd) {
        int sockfd = *((int *) connfd);
        int len;
        // Client Thread Always Ready to Get Message
        while ((len = recv (sockfd, msg, 1000, 0))>0){
                msg [len] = '\0';
                fputs (msg, stdout);
        }
        return 0;
}
```

```
void errorExit (int errorSignal) {
        cout << "FAIL: ";
        if ( errorSignal == 1)
            cout << " SOCKET CREATING\n";
        else if (errorSignal == 2)
            cout <<" SERVER CONNECTION\n";
        exit (0);
}

int main ( int argc , char *argv[ ]) {
    // Socket creation
    int sockfd = socket (AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in servaddr;
    if (sockfd != -1)
        cout<< " SUCCESS : SOCKET CREATED\n";
    else
        errorExit (1);
    bzero (&servaddr, sizeof (servaddr));

    // Assigning IP and PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    // servaddr.sin_addr.s_addr = inet_addr ("127.0.0.1");
    servaddr.sin_port = htons (PORT);

    // Connecting to Server Socket
    if(!(connect (sockfd, (struct sockaddr*)&servaddr,
                    sizeof (servaddr))))
        cout << "SUCCESS: CONNECTET TO SERVER\n";
    else errorExit (2);
```

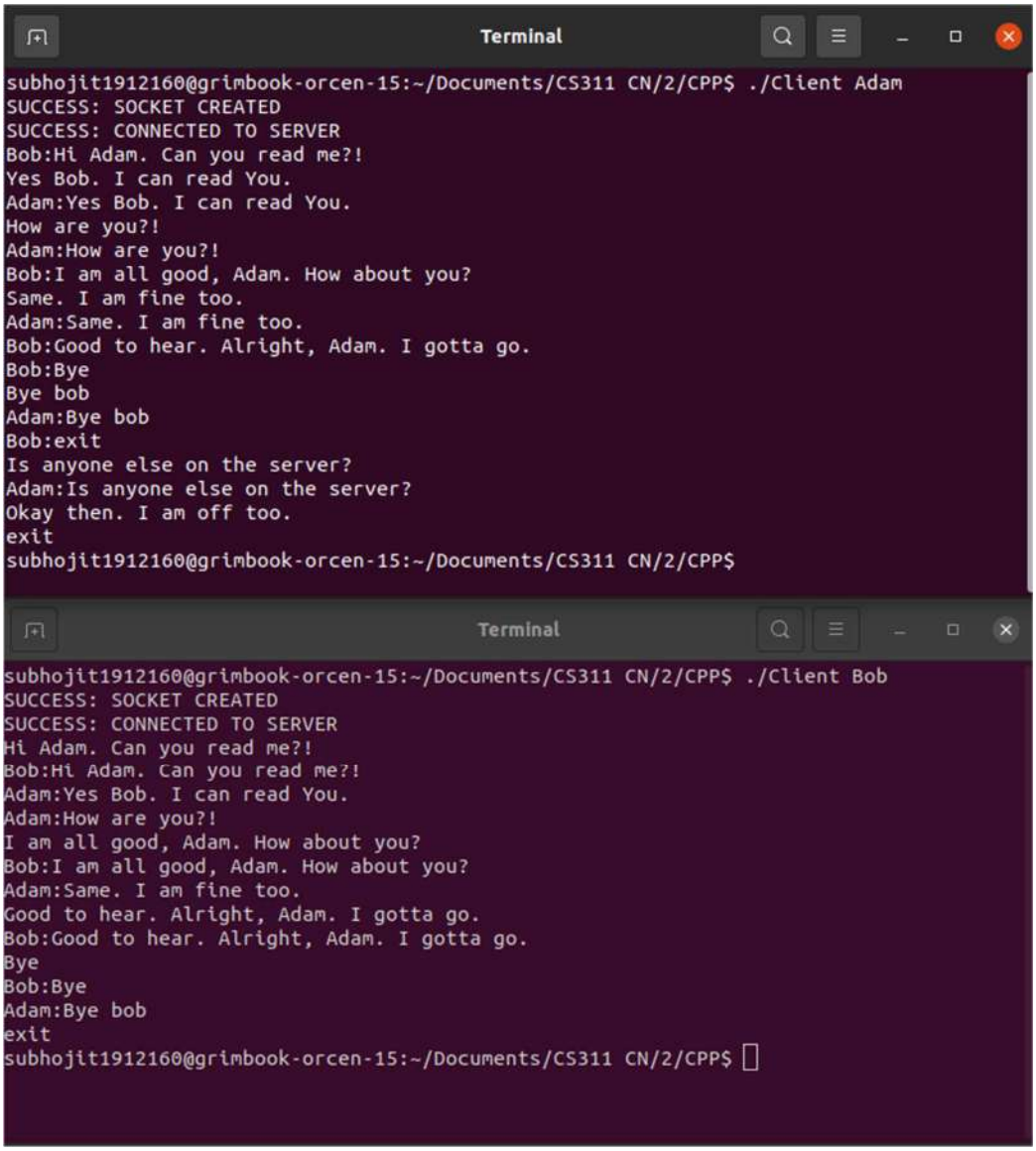
```
// Create client thread always ready for message receive
pthread_t recvt;
pthread_create (&recvt, NULL, &recvMsg, &sockfd);

// READY TO RECEIVE MESSAGE FROM CONSOLE
int len;
char sendMsg[1000], clientName[100];
strcpy (clientName, argv[1]);
while (fgets (msg, 500, stdin) > 0) {
        strcpy (sendMsg, clientName);
        strcat (sendMsg, ":");
        strcat (sendMsg, msg);
        len = write (sockfd, sendMsg, strlen(sendMsg));
        if (len < 0)
                cout << "FAIL: MESSAGE SEND\n";
        if (!strncmp ("exit", msg, 4))
                break;
}

// pthread_join (recvt, NULL);
close (sockfd);
return 0;
}
```

OUTPUT:

// TCP CHAT SERVER



// TCP CHAT CLIENT



OUTPUT EXPLANATION:

Firstly, the TCP Server Program is compiled and run. After the server is up and running, any or all clients created will be able to connect to the server. In TCP based Chat server, the server cannot, or rather, does not communicate with the clients. The clients communicate with one another. The server just acts as a secure medium to transmit the messages from one client to another. So, once the server is created, the TCP Client Program is now compiled and run. As many clients (or as many as the server allows) can run the compiled file, or the application file, and connect to the server and send messages to all the online clients , as well as receive the messages sent by other clients on the same server. After sometime, when the server overloads, it exits automatically, disconnecting everyone.

Q.3. Write a Socket Program to implement "Go-Back-N" Protocol using TCP.

AIM: TO IMPLEMENT "GO-BACK-N" PROTOCOL USING TCP IN C

THEORY: 1. GO-BACK-N PROTOCOL: It is a connection oriented transmission. It uses sliding window method for reliable and sequential delivery of data frames. This data link layer protocol provides for sending multiple frames before receiving the acknowledgement for the first frame. The Go-Back-N protocol works for both the sender and the receiver side to ensure reliable data transfer.

2. SLIDING WINDOW PROTOCOL: It is a feature-based data transmission protocols. Sliding window protocols are used where reliable in-order delivery of packets is required, such as in the Data Link layer (OSI model) as well as in Transmission Control Protocol (TCP). By placing limits on the number of packets that can be transmitted or received at any given time, a sliding window protocol allows an unlimited number of packets to be communicated using fixed-size sequence numbers.

3. The Sender: Better known as client, the sender has a sequence of frames to send. The sender starts by sending the first frame, initially consisting of the base and the next packet to send. While there are more packets to send and the value for the next

packet to send is smaller than the summation of base and window size, the sender sends the packet pointed by pointer of next packet to send variable, and also increments the pointer. Meanwhile, the base pointer is incremented after receiving acknowledgement packets from the receiver.

4. The Receiver: Better known as the server, the receiver only keeps track of the expected sequence number to receive next. There is no receive buffer; the out of order packets are simply discarded, and so are the corrupted packets. The receiver always sends the acknowledgement for the last in-order packet received upon reception of a new packet (whether successfully or unsuccessfully).

The entire operation can be broken down into following parts:

1. Connection Setup: First, the connection is set up with a 3-way handshake between the sender and the receiver.

2. Data Transmission: Once the connection is established, the data is sent in DATA packets by the sender. Each DATA packet is acknowledged by the receiver by sending an ACK packet.

3. Connection Teardown: Once the data transmission is completed, the connection is torn down by both the sender and the receiver by sending FIN packets.

CODE:

// TCP GBN SENDER AND RECEIVER.

// RECEIVER:

```
# include <iostream>
# include <sys/types.h>
# include <sys/socket.h>
# include <netinet/in.h>
# include <arpa/inet.h>
# include <cstring>
# include <time.h>
# include <cstdlib>
# include <unistd.h>

# define windowSize 3
# define frameNumber 50
# define lossRate 10
# define PORT 8080

using namespace std;

Char buffer [10];
Char timeOutMsg [] = "Time Out!";

Void errorExit (string errorSignal) {
        Cout << "FAIL: " << errorSignal << endl;
        exit (0);
}
```

```
void gbnGenerate (int ack) {
    int ii = 0;
    int jj, kk = ack, ll;
    while (kk > 0) {
        ++ii;
        kk = kk/10;
    }
    ll = ii--;
    while (ack > 0) {
        kk = ack % 10;
        buffer [ii--] = kk + 48;
        ack = ack / 10;
    }
    buffer[ll] = '\0';
}

int main () {
    int sockfd = socket (AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in gbnServAddr;
    if (sockfd != -1)
        cout << " SUCCESS: SOCKET CREATED \n";
    else
        errorExit (" SOCKET CREATING ");
    bzero (&gbnServAddr, sizeof (gbnServAddr));

    gbnServAddr.sin_family = AF_INET;
    gbnServAddr.sin_addr.s_addr = htonl (INADDR_ANY);
    gbnServAddr.sin_port = htons (PORT);
```

```
if (!(bind (sockfd, (struct sockaddr*) &gbnServAddr,
                    sizeof (gbnServAddr))))
        cout << "SUCCESS : SOCKET BINDED \n";
else
        errorExit ("SOCKET BINDING");

Struct sockaddr_in gbnClntAddr;
unsigned int cliAddrLen;
if (!(listen (sockfd, 5)))
        cout << " SUCCESS: SERVER LISTENING \n";
else
        errorExit ("SERVER LISTENING");
 cliAddrLen = sizeof (gbnClntAddr);

int connfd = accept (sockfd, (struct sockaddr *) &gbnClntAddr,
                                &cliAddrLen);
if (connfd >= 0)
        cout << "SUCCESS: CLIENT ACCEPTED \n\n";
else
        errorExit ("SERVER ACCEPTING");

unsigned int timeOut = (unsigned int) time (NULL);
srand (timeOut);
recv (connfd, buffer, sizeof (buffer), 0);

int buffValue = atoi (buffer);
int ii, frameDrop;
int ack = 1;
```

```
while (1) {
        for (ii = 0; ii < windowSize; ++ii) {
            recv (connfd, buffer, sizeof (buffer), 0);
            if (strncmp (buffer, timeOutMsg, 10) == 0)
                break;
        }
        ii = 0;
        while (ii < windowSize) {
            frameDrop = rand() % frameNumber;
            if (frameDrop < lossRate) {
                send (connfd, timeOutMsg, sizeof (
                                    timeOutMsg), 0);
                break;
            }
            else {
                gbnGenerate (ack);
                if (ack <= buffValue + 1) {
                    cout << "\nFRAME RECEIVED: ";
                    cout << buffer;
                    send (connfd, buffer, sizeof(buffer),0);
                }
                else break;
                ++ack;
            }
            if (ack > buffValue) break;
            ++ii;
        }
}
cout << " SUCCESS: SERVER EXIT\n"; close (sockfd); return 0;
```

// SENDER:

```
# include <iostream>
# include <sys/types.h>
# include <sys/socket.h>
# include <netinet/in.h>
# include <arpa/inet.h>
# include <cstring>
# include <time.h>
# include <cstdlib>
# include <unistd.h>

# define PORT 8080
# define windowSize 3

using namespace std;

char buffer [10];
char timeOutMsg [] = "Time Out!";

void errorExit (string errorSignal) {
        cout << "FAIL: " << errorSignal << endl;
        exit (0);
}
```

```
void gbnGenerate (int ack) {
    int ii = 0, jj, kk = ack, ll;
    while (kk > 0) {
        ++ii;
        kk = kk / 10;
    }
    ll = ii--;
    while (ack > 0) {
        kk = ack % 10;
        buffer [ii--] = kk + 48;
        ack = ack / 10;
    }
    buffer [ll] = '\0';
}

int main () {
    int sockfd = socket (AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in gbnServAddr;
    if (sockfd != -1)
        printf ("SUCCESS: SOCKET CREATED\n");
    else
        errorExit ("SOCKET CREATION");
    bzero (&gbnServAddr, sizeof (gbnServAddr));

    gbnServAddr.sin_family = AF_INET;
    gbnServAddr.sin_addr.s_addr = htonl (INADDR_ANY)
                                  // htonl (INADDR_ANY);
    gbnServAddr.sin_port = htons (PORT);
```

```
if (!(connect (sockfd, (struct sockaddr*) &gbnServAddr,
                                    sizeof (gbnServAddr))))
        printf ("SUCCESS: CONNECTED TO SERVER \n\n");
else
        errorExit ("SERVER CONNECTION");

int frameNumber;
cout << endl;
cout << "Enter the number of frames:  ";
cin >> frameNumber;
cout << endl;
gbnGenerate (frameNumber);
send (sockfd, buffer, sizeof (buffer), 0);

int ii, windowTemp, ack = 1, bufferValue = 0, gbN = 0;
while (1) {
    for (ii = 0; ii < windowSize; ++ii) {
        gbnGenerate (ack);
        send (sockfd, buffer, sizeof (buffer), 0);
        if (ack <= frameNumber) {
            cout << "FRAME SENT  " << ack << endl;
            ++ack;
        }
    }
    sleep (3);
    ii = 0;
    windowTemp = windowSize;
```

```
while (ii < windowSize) {
    recv (sockfd, buffer, sizeof (buffer), 0);
    bufferValue = atoi (buffer);
    if (strcmp (buffer, timeOutMsg) == 0) {
        gbN = ack - windowTemp;
        if ( gbN < frameNumber) {
            cout << "\n\n TIME OUT. RESENDING";
            cout << " FRAMES " << gbN << " ONWARDS";
            cout << endl;
        }
        sleep (3);
        break;
    }
    else {
        if (bufferValue <= frameNumber) {
            cout << "FRAME ACKNOWLEDGED. FRAME";
            cout << " SEQUENCE : " << buffer << endl;
            -- windowTemp;
        }
        else break;
    }
    if (bufferValue > frameNumber) break;
    ++ii;
}
if (windowTemp == 0 && ack > frameNumber) {
    send (sockfd, timeOutMsg, sizeof (timeOutMsg), 0);
    break;
}
```

```
        else {
            ack = ack - windowTemp;
            windowTemp = windowSize;
        }
        if (atoi(buffer) == frameNumber)
            break;
    }
    cout << " SUCCESS: CLIENT EXIT " << endl;
    close (sockfd);
    return 0;
}
```

// OUTPUT EXPLANATION:

The socket port in this Go-Back-N Protocol Implementation was pre-defined as 8080. The loss rate (ie, the chance to drop frame) was pre-taken as 10%. The total number of frames to be accepted was limited to 50, that means the receiver cannot accept more than 50 data frames from the sender.

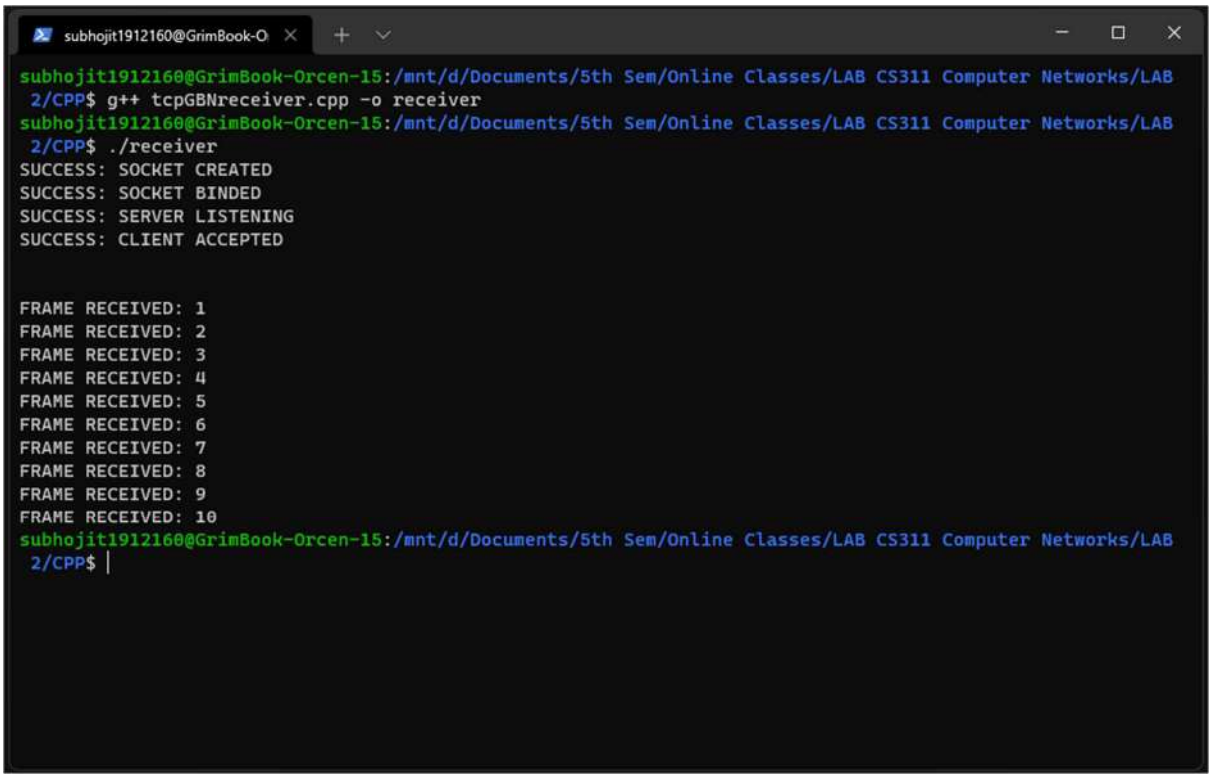
Firstly, the receiver programme is compiled and run. The we compile and run the sender programme. Once the connection is established between the sender and the receiver, we give in the number of data frames we wish to work with as the input in the sender terminal. Then we enter the window size. Window size ensures how many data packets are sent at a time. Here we have taken window size as 3, so we first send 3 data frames back to back, without awaiting the receiver's acknowledgement. After sending, we wait for

the receiver to acknowledge the data packets. If the receiver successfully receives the data packet, the server terminal updates with the frame Number it has received, and the sender terminal also updates with the "frame acknowledged" message. If the receiver fails to receive data; in our case, receiver suffers data drop for frame 3; the next three frames starting from frame 3 will be resent. The data drop is completely random, that means, there is a possibility that all the data frames will be received with no error at all, and also a propability all the data frames will suffer drop and have to be resent all over again. The process will continue until the receiver receives all the data frames and the sender receives acknowledgement for all the sent data packets, ensuring successful transmission of data packets.

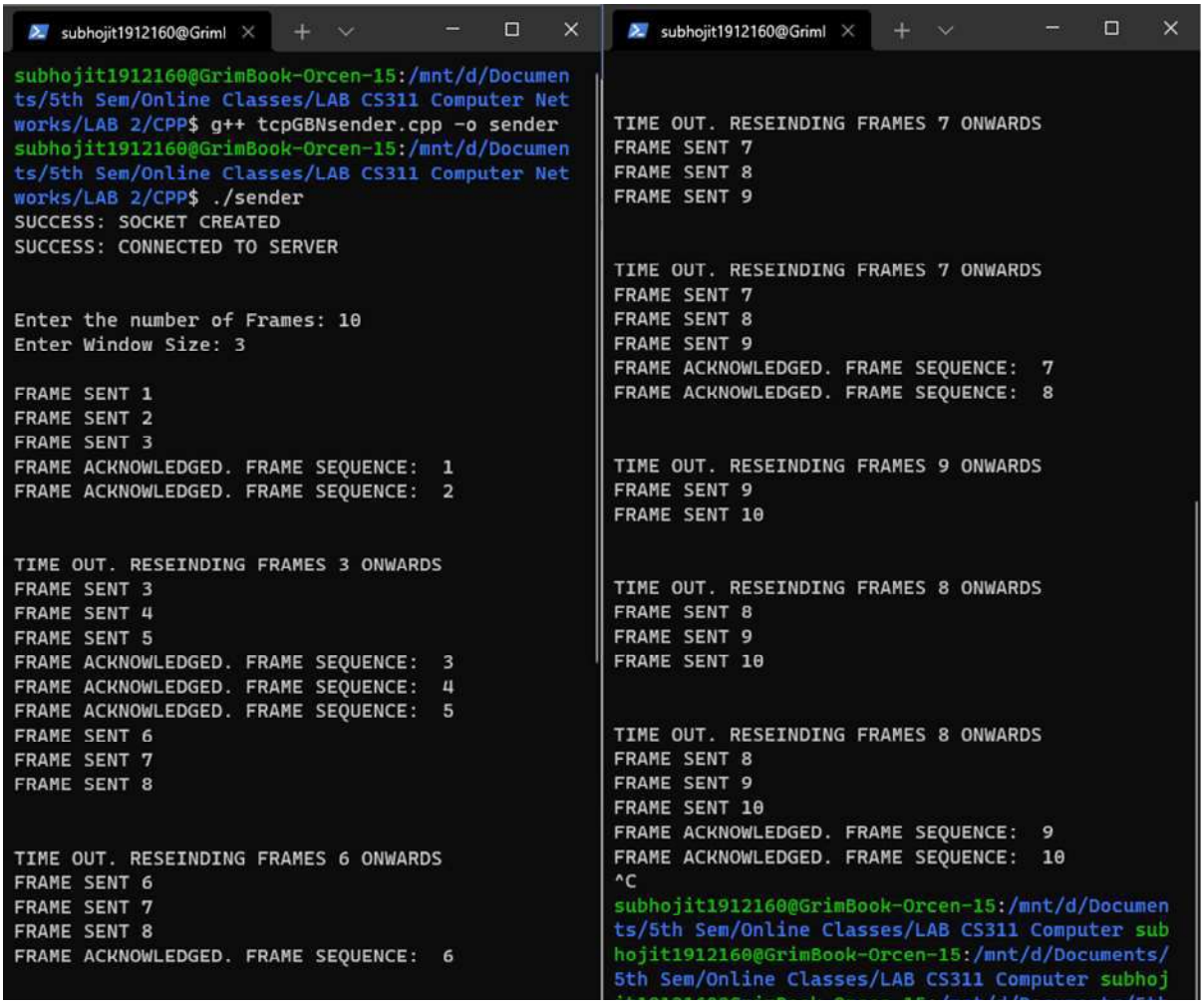
Finally, the receiver and the sender program will close the connection and terminate successfully, (in our case, I have forcefully terminated the terminal using Ctrl+C command due to unforeseen bug issues).

OUTPUT:

// TCP GO-BACK-N RECEIVER



// TCP Go-BACK-N SENDER



Q.4. Write an "Echo Client" and "Echo Server" using UDP to estimate the round trip time from client to server. The server should be such that it can accept multiple connections at any given time. (Description: Multiple clients connected at the same time to one server for connection establishment, server has to listen to all the clients, use I/O monitoring calls if required.)

AIM: TO IMPLEMENT "ECHO CLIENT SERVER" TO ESTIMATE THE ROUND TRIP TIME FROM CLIENT TO SERVER USING UDP IN CPP.

THEORY: 1. ECHO CLIENT SERVER: It is an application that allows a client and a server to connect so a client can send a message to the server and the server can receive the message and send, or echo, it back to the client.

2. UDP CLIENT SERVER: In UDP, the client does not form a connection with the server like in TCP. Instead, the client just sends a datagram. Similarly, the server does not need to 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.

3. ROUND TRIP TIME: The RTT is the duration in milliseconds (ms) taken by a network request to go from a starting point to a destination and back again to the starting point.

4. I/O MULTIPLEXING : It essentially means reading from or writing to multiple file descriptors simultaneously. I/O multiplexing is used when a client is handling multiple descriptors or multiple sockets, as well as when a server handles both TCP and UDP or handles multiple services and protocols.

The entire operation can be broken down as follows:

UDP ECHO CLIENT:

(i) A socket is created and binded.

(ii) The messages are sent from the user using sendto() function and the RTT start time is recorded.

(iii) The data from the server is received using recvfrom() and the RTT end time is recorded.

(iv) The final RTT time is displayed along with the echo message.

UDP ECHO SERVER:

(i) A socket is created and binded to an advertised port number.

(ii) An infinite loop is started to process the client requests for connections.

(iii) The process receives data from the client using recvfrom() function and echoes the same data using the sendto() function.

(iv) This server is capable of handling multiple clients automatically as UDP is a datagram based protocol and hence, no exclusion connection is required to a client.

CODE :

```
// ECHO SERVER

#include <iostream>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT 8080
#define MAXLINE 1024

using namespace std;

int main() {
    int sockfd;
    struct sockaddr_in servaddr, cliaddr;
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("FAIL: SOCKET CREATION \n");
        exit(EXIT_FAILURE);
    }
    cout << "SUCCESS: SOCKET CREATED \n";
    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));
```

```
servaddr.sin_family      = AF_INET ;
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port        = htons (PORT);

if (bind (sockfd, (const struct sockaddr *)&servaddr,
                                    sizeof (servaddr)) < 0 ) {
        perror ("FAIL: SERVER BINDING \n");
        exit (EXIT_FAILURE);
}
cout << " SUCCESS: SERVER BOUND \n";

cout << " SERVER LISTENING FOR MESSAGES TO ECHO BACK...\n";
char buffer [MAXLINE];
unsigned int len, nn;
len = sizeof (cliaddr);
while (1) {
        memset (buffer, 0, MAXLINE);
        nn = recvfrom (sockfd, (char *) buffer, MAXLINE,
             MSG_WAITALL, (struct sockaddr *)&cliaddr,
              &len );
        buffer[nn] = '\0' ;
        cout << "\n" << buffer << " : " ;
        memset (buffer, 0, MAXLINE);
        nn = recvfrom (sockfd, (char *) buffer, MAXLINE,
             MSG_WAITALL, (struct sockaddr *) &cliaddr,
             &len);
        buffer [nn] = '\0' ;
        cout << buffer;
```

```
        sendto (sockfd, (const char *) buffer, strlen(buffer),
                MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
                len);
        cout << " ECHO SENT.\n";
    }
    close (sockfd);
    return 0;
}
```

// ECHO CLIENT

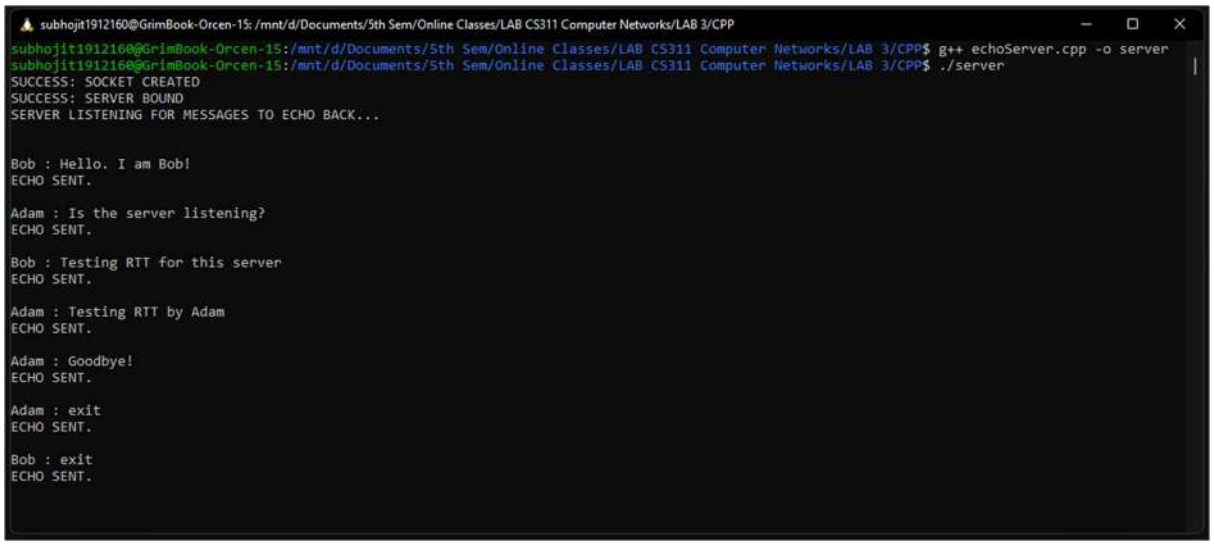
```
# include <iostream>
# include <cstdlib>
# include <unistd.h>
# include <cstring>
# include <sys/types.h>
# include <sys/socket.h>
# include <arpa/inet.h>
# include <netinet/in.h>
# include <ctime>

# define PORT 8080
# define MAXLINE 1024

using namespace std;
```

```
int main (int argc, char *argv[]) {
        clock_t start, end;
        double cpuTimeUsed;
        int sockfd;
        struct sockaddr_in servaddr;
        if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0))
                                                < 0) {
                perror ("FAIL: SOCKET CREATION\n");
                exit (EXIT_FAILURE);
        }
        cout << " SUCCESS: SOCKET CREATED\n\n";
        memset (&servaddr, 0, sizeof (servaddr));

        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons (PORT);
        servaddr.sin_addr.s_addr = INADDR_ANY;

        char buffer [MAXLINE];
        char msg [MAXLINE];
        char clientName [10];
        strcpy (clientName, argv [1]);
        cout << " SEND MESSAGES \n";

        unsigned int nn, len;
        while (1) {
                memset (msg, 0, MAXLINE);
                memset (buffer, 0, MAXLINE);
                cout << " > ";
                fgets (msg, MAXLINE, stdin);
```

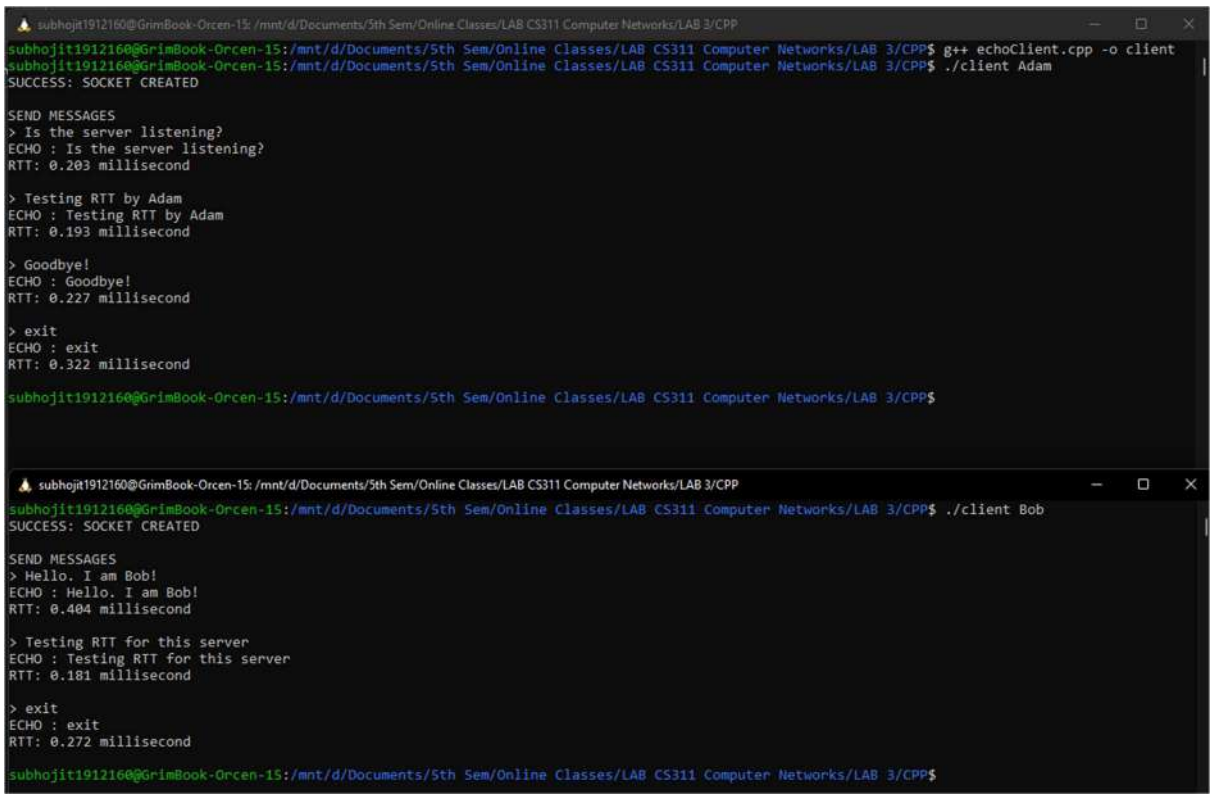
```
start = clock ();
sendto (sockfd, (const char *) clientName , strlen (clientNam
        , MSG-CONFIRM , (const struct sockaddr * ) &servaddr,
         sizeof (servaddr));
sendto ( sockfd, (const char *) msg, strlen (msg),
         MSG-CONFIRM, (const struct sockaddr * ) &servaddr,
         sizeof (servaddr));
nn = recvfrom ( sockfd, (char *) buffer , MAXLINE,
     MSG-WAITALL, (struct sockaddr *) &servaddr,
     &len );
buffer [nn] = '\0' ;
sleep (1);
end = clock();

cout <<" ECHO : " << buffer ;
cpuTimeUsed = ((double) (end-start)) | CLOCKS-PER-SEC;
cout << "RTT : " <<(cpuTimeUsed * 1000) <<" milliseconds";
cout << endl << endl;

if (strncmp (msg, "exit", 4) == 0)
        break;
}

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

OUTPUT:

// UDP ECHO SERVER



// UDP ECHO CLIENT



Output Explanation:

Firstly, the UDP Server programme is compiled and run. The server, after socket creation and binding, is open to all the incoming connections. Then, the UDP Client programme is compiled and run in two different terminals, each terminal acting as its own separate client. Either of the client can send message at any time; in the above screenshot, the client named “Bob” sends the first message which is echoed back and the round trip time for the message from client to server and back to client is displayed. Similarly, the client named “Adam” send their own message and the RTT for the message to transmit from the client to server and back to client is displayed. In the server terminal as well, every log of received message from all the client and echoed messages are recorded. In the end, the clients break the connection with the server by sending “exit” message, which terminates the processes.

Q.5. Write a program for "Connectionless Iterative Service" in which the server finds the factorial of a number sent by the client and sends it back.

[redacted]

AIM: TO IMPLEMENT "CONNECTIONLESS ITERATIVE SERVICE" TO FIND THE FACTORIAL OF A NUMBER SENT BY THE CLIENT USING UDP IN CPP.

THEORY: 1. UDP CLIENT SERVER: In UDP, the client does not form a connection with the server (hence, connectionless) like in TCP. Instead, the client just sends a datagram. Similarly, the server does not need to 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.

2. CONNECTIONLESS ITERATIVE SERVER: In this model, the server receives a request packet from UDP, processes the request and gives response to the UDP to send to the client. The packets are stored in queue and processed in order of arrival.

3. CONNECTIONLESS SERVICE: It is a technique that is used in data communications to send or transfer data or message at layer 4, i.e., Transport layer of Open System Interconnection model.

4. ITERATIVE SERVER: An iterative server processes request from clients in a serial manner; one connection is served and responded to before the server accepts a new client connection.

The entire operation can be broken down as follows:

CONNECTIONLESS ITERATIVE SERVICE SERVER:

(i) A socket is created and binded to an advertised port number.

(ii) An infinite loop is started to process the client requests for connections.

(iii) The process receives a number from the client using recvfrom() function and calculates the factorial for the received number, and sends it back to the client using sendto() function.

CONNECTIONLESS ITERATIVE SERVICE CLIENT:

(i) A socket is created and binded.

(ii) The number, whose factorial is to be found, is sent as a message from the user (client) using sendto() function.

(iii) The factorial value from the server is received using recvfrom() function and displayed.

CODE :

// CIS SERVER

```
# include <iostream>
# include <cstdlib.>
# include < cstring >
# include < unistd.h>
# include < sys/types.h >
# include < sys/socket.h>
# include < 3rpa/inet.h>
# include < netinet/in.h>

# define PORT 8080
# define MAXLINE 1024
char fact [MAXLINE];

using namespace std;

int multiply ( int xx, int res[] , int resSize) {
        int carry = 0;
        for (int ii = 0 ; ii < resSize ; ++ii ) {
            int prod = res [ii] * xx + carry ;
            res [ii] = prod % 10 ;
            carry = prod / 10 ;
        }
```

```
    while (carry) {
            res [resSize] = carry % 10;
            carry = carry / 10;
            ++resSize;
    }
    return resSize;
}

void factorial (int number) {
    int res [MAXLINE];
    char numToTxt [5];
    res [0] = 1;
    int resSize = 1;
    for (int xx = 2; xx <= number; ++xx)
        resSize = multiply (xx, res, resSize);
    for (int ii = resSize - 1; ii >= 0; --ii) {
        sprintf (numToTxt, "%d", res [ii]);
        strcat (fact, numToTxt);
    }
}

int main () {
    int sockfd;
    struct sockaddr_in servaddr, cliaddr;

    if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror ("FAIL: SOCKET CREATION\n");
        exit (EXIT_FAILURE);
    }
```

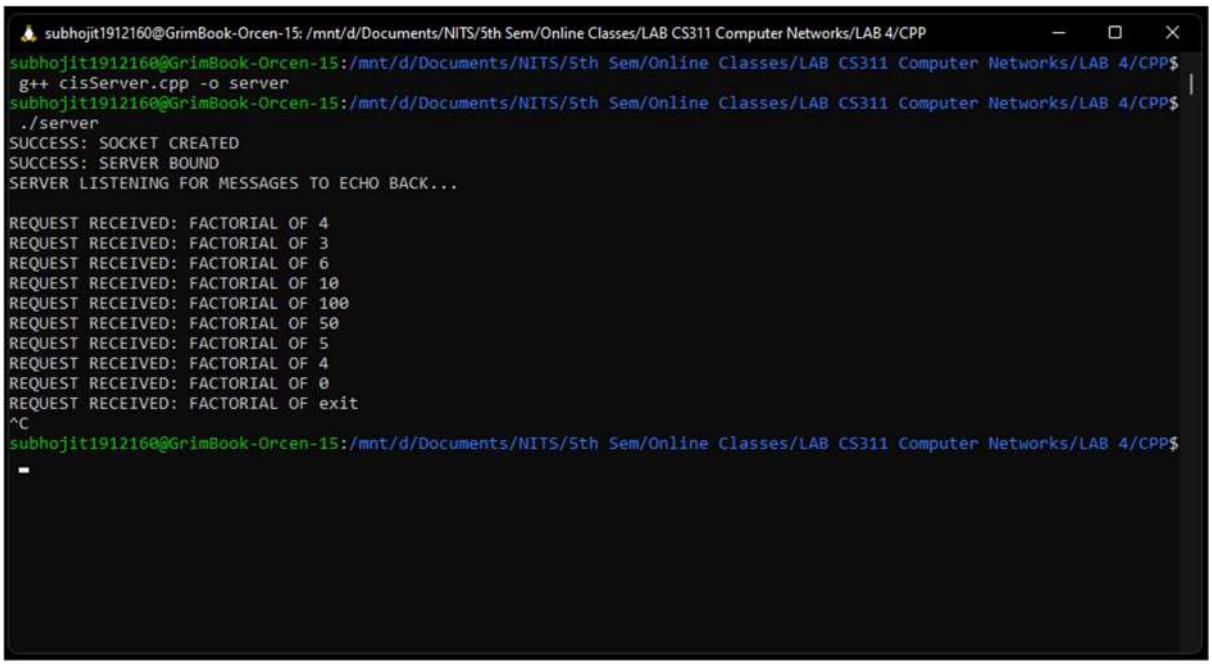
```
cout << " SUCCESS: SOCKET CREATED \n" ;

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

servaddr. sin-family = AF_INET ;
servaddr. sin-addr. S-addr = INADDR-ANY;
servaddr. sin-port = htons (PORT) ;

if (bind (sockfd, (const struct sockaddr *) & servaddr,
                    sizeof (servaddr)) < 0 ) {
        perror ("FAIL: SERVER BIND \n");
        exit (EXIT-FAILURE);
}
cout << "SUCCESS: SERVER BOUND \n" ;

cout <<" SERVER LISTENING FOR MESSAGES... \n\n";
char buffer [MAXLINE];
unsigned int len , nn;
len = sizeof (cliaddr) ;
while (1) {
        memset (buffer , 0 , MAXLINE) ;
        bzero (fact , MAXLINE);
        nn = recvfrom (sockfd, (char *) buffer,
                MAXLINE, MSG-WAITALL, (struct sockaddr*)
                & cliaddr , & len);
        buffer [nn] = '\0' ;
        cout << " REQUEST RECEIVED: FACTORIAL of " << buffer
        int num = atoi (buffer) ;
```

```
            factorial (num);
            bzero (buffer, MAXLINE);
            strcpy (buffer, fact);
            sendto (sockfd, (const char *) buffer, strlen (buffer),
                    MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
                    len);
        }
        close (sockfd);
        return 0;
}

// CIS CLIENT

# include <iostream>
# include <cstdlib>
# include <unistd.h>
# include <cstring>
# include <sys/types.h>
# include <sys/socket.h>
# include <arpa/inet.h>
# include <netinet/in.h>

# define PORT 8080
# define MAXLINE 1024

using namespace std;

int main () {
        int sockfd;
```

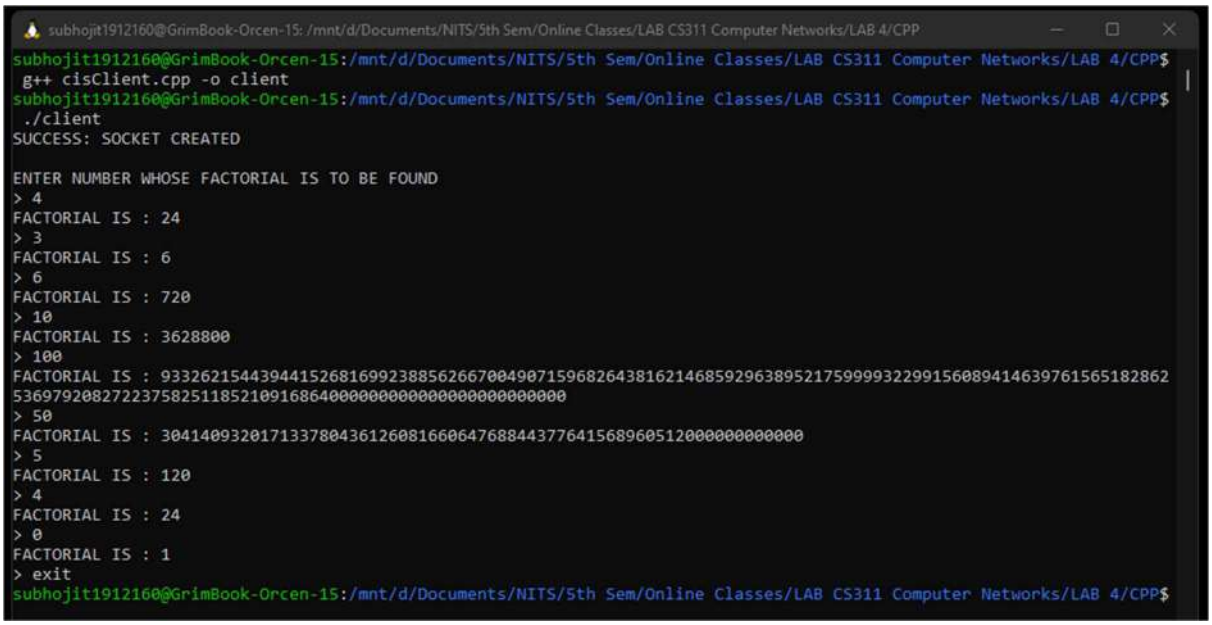
```
struct sockaddr_in servaddr;
if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror ("FAIL: SOCKET CREATION \n");
        exit (EXIT_FAILURE);
}
cout << "SUCCESS: SOCKET CREATED \n\n";
memset (&servaddr, 0, sizeof (servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons (PORT);
servaddr.sin_addr.s_addr = INADDR_ANY;
unsigned int nn, len;
char buffer [MAXLINE], msg[MAXLINE];
cout << " ENTER NUMBER WHOSE FACTORIAL IS TO BE FOUND ";
while (1) {
    memset (msg, 0, MAXLINE);
    memset (buffer, 0, MAXLINE);
    cout << "\n> ";
    fgets (msg, MAXLINE, stdin);
    sendto (sockfd, (const char *)msg, strlen (msg), MSG_CONFIRM,
            (const struct sockaddr *)&servaddr, sizeof (servaddr));
    nn = recvform (sockfd, (char *)buffer, MAXLINE, MSG_WAITALL,
         (struct sockaddr *) &servaddr, &len);
    buffer [nn] = '\0';
    if (strncmp (msg, "exit", 4) == 0)
        break;
    cout << "FACTORIAL IS: " <<buffer;
}
close (sockfd);
return 0;
}
```

OUTPUT:

// CONNECTIONLESS ITERATIVE SERVICE SERVER



// CONNECTIONLESS ITERATIVE SERVICE CLIENT



Output Explanation:

Firstly, the CIS Server is compiled and run, making it open to receive all types of connections. Then the CIS Client is compiled and run, connecting it to the CIS Server. Then, in the Client Terminal, the number for which the factorial is to be found is entered. The Server Terminal receives the request and calculates the factorial for the requested number. After the factorial is calculated, the server echoes back the calculated value (which is the factorial value) to the client, which is then displayed on the Client Terminal.

Q.6. Write a Client-Server socket program to implement "FTP Server" using UDP connection. (Description : The file.txt may be there on the server side with some text; on the client request, the server has to send the text file file.txt and corresponding text has to print on client terminal.)

AIM: TO IMPLEMENT "FTP SERVER" USING UDP CONNECTION IN CPP.

THEORY: 1. UDP CLIENT SERVER: In UDP, the client does not form a connection with the server like in TCP. Instead, the client just sends a datagram. Similarly, the server does not need to accept a connection and just waits and scans for datagrams to arrive. Datagrams, upon arrival, contain the address of the sender, along with the message, which the server uses to send data to the correct client.

2. FTP : A File Transfer Protocol (FTP) client is a software utility that establishes a connection between a host computer and a remote server, typically an FTP server. FTP refers to a group of rules that govern how computers transfer files from one system to another over the internet. FTP protocol uses TCP protocol for client server communication. However, there is another protocol called Trivial File Transfer Protocol (TFTP) which uses UDP protocol to transfer files. TFTP has minimal features and doesn't have authentication.

3. FTP SERVER: It holds the files and databases that are required to provide the services requested by clients.

4. FTP CLIENT: It is generally a personal computer used by an end user or a mobile device which is running the necessary software that is capable of requesting and receiving files over the internet from a FTP server.

The entire operation can be broken down as follows:

FTP CLIENT:

(i) A socket is created and binded.

(ii) The file name, whose content is to be read / retrieved, is entered as an input and sent from the user using sendto().

(iii) The content of the file whose name was sent is received using recvfrom() function and displayed.

FTP SERVER:

(i) A socket is created and binded to an advertised port number.

(ii) An infinite loop is started to process the client requests for connections.

(iii) The process receives a filename from the client using recvfrom() function and reads content from the file, and sends the content back to the client using sendto() function.

CODE :

```
// FTP UDP SERVER

# include <iostream>
# include <cstdlib>
# include <unistd.h>
# include <cstring>
# include <sys/types.h>
# include <sys/socket.h>
# include <arpa/inet.h>
# include <netinet/in.h>
# include <fstream>

# define PORT 8080
# define MAXLINE 1024
char fileContent [MAXLINE];

using namespace std;

int ftpFileProcess (char *fileName) {
        FILE *ff = fopen (fileName, "r");
        if (ff == NULL)
                return 0;
        int ii = 0;
        while (!feof (ff))
                fileContent [ii++] = getc (ff);
        fileContent [ii-1] = '\0';
```

```
    if (fileContent[0] == '\0')
        return 0;
    return 1;
}

int main () {
    int sockfd;
    Struct sockaddr_in servaddr, cliaddr;
    if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0) {
            perror ("FAIL: SOCKET CREATION\n");
            exit (EXIT_FAILURE);
    }
    cout << " SUCCESS: SOCKET CREATED \n";

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

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons (PORT);

    if (bind (sockfd, (const struct sockaddr *) &servaddr,
                sizeof (servaddr)) < 0 ) {
        perror ("FAIL: SERVER BIND \n");
        exit (EXIT_FAILURE);
    }
    Cout << " SUCCESS: SERVER BOUND \n";

    cout << " SERVER LISTENING FOR INCOMING MESSAGES...";
```

```
cout << endl << endl;
char buffer[MAXLINE];
unsigned int len, nn;
len = sizeof(cliaddr);
while (1) {
        memset (buffer, 0, MAXLINE);
        bzero (fileContent, MAXLINE);
        nn = recvfrom (sockfd, (char *) buffer, MAXLINE,
             MSG_WAITALL, (struct sockaddr *) &cliaddr, &len);
        buffer[nn] = '\0';
        cout << "REQUEST RECEIVED : OPENING FILE " << buffer;
        buffer[nn-1] = '\0';
        if (ftpFileProcess (buffer)) {
            memset (buffer, 0, MAXLINE);
            strcpy (buffer, fileContent);
        }
        else {
            memset (buffer, 0, MAXLINE);
            strcpy (buffer, "ERROR: FILE EITHER
                    EMPTY OR DOES NOT EXIST");
        }
        sendto (sockfd, (const char *) buffer, strlen(buffer),
            MSG_CONFIRM, (const struct sockaddr *)
            &cliaddr, len);
}
close (sockfd);
return 0;
}
```

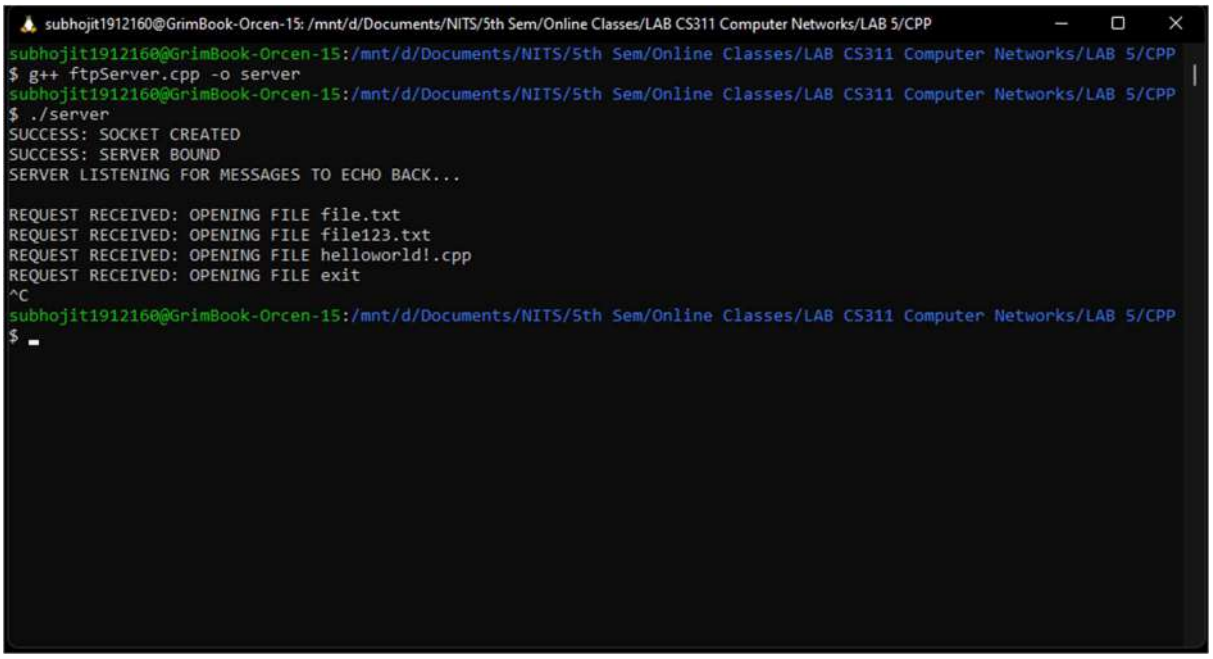
// FTP UDP CLIENT

```
# include <iostream>
# include <cstdlib>
# include <unistd.h>
# include <cstring>
# include <sys/types.h>
# include <sys/socket.h>
# include <arpa/inet.h>
# include <netinet/in.h>

# define PORT 8080
# define MAXLINE 1024

using namespace std;

int main() {
    int sockfd;
    struct sockaddr_in servaddr;
    if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror ("FAIL : SOCKET CREATION \n");
        exit (EXIT_FAILURE);
    }
    cout << " SUCCESS : SOCKET CREATED \n\n ";
    memset (&servaddr, 0, sizeof (servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons (PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;
```

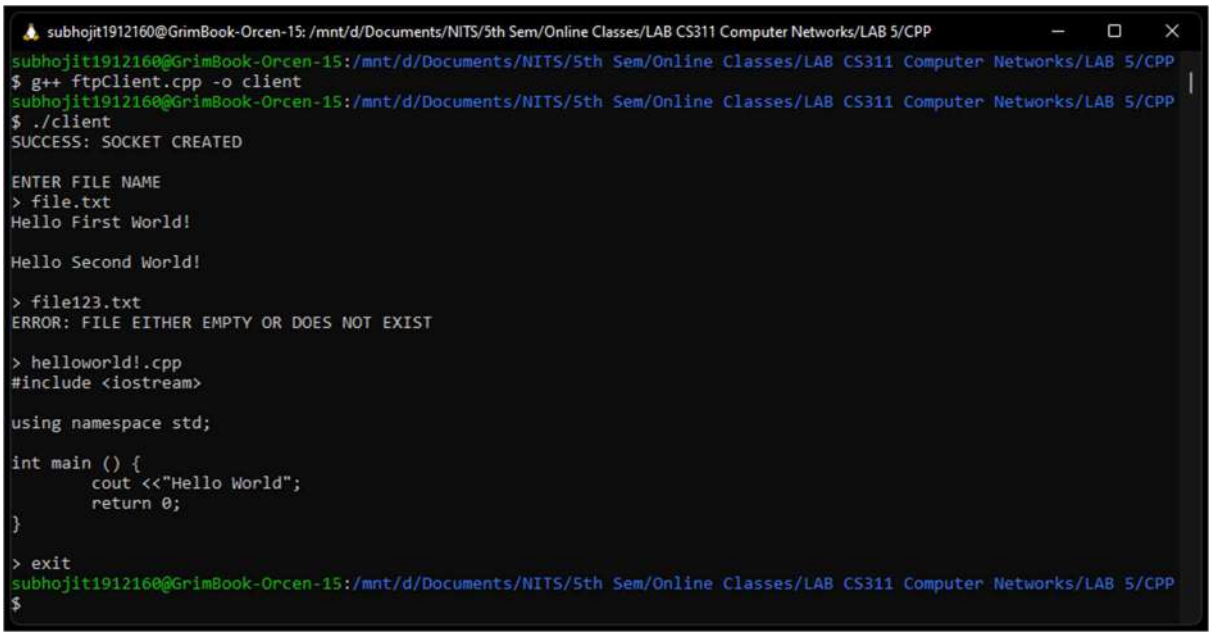
```
char buffer [MAXLINE];
char fileName [MAXLINE];
cout << "ENTER FILE NAME : " ;
unsigned int nn , len ;

while (1) {
    memset (fileName , 0, MAXLINE);
    memset (buffer, 0, MAXLINE) ;
    cout << "\n>  " ;
    fgets (fileName , MAXLINE, stdin) ;

    sendto (sockfd , (const char *) fileName , strlen (fileName),
            MSG-CONFIRM, (const struct sockaddr *) &servaddr,
            sizeof (servaddr));
    nn = recvfrom (sockfd , (char *) buffer, MAXLINE,
            MSG-WAITALL, (struct sockaddr *) &servaddr,
            &len) ;
    buffer[nn ] = '\0' ;
    if (strncmp (fileName , "exit" , 4) == 0)
            break;
    cout << buffer << endl;
}

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

OUTPUT:

// FTP SERVER



// FTP CLIENT



Output Explanation:

Firstly, two terminals are opened and FTP client and server programmes are compiled and run. The client terminal asks the user to enter file name for the file the user is looking to read. Once the file name is entered, the file name is sent as a message from client to server. The server accepts the file name as the buffer value and searches for the file in its database with the same name as the buffer message. If the file is found, then the server reads and stores the content of that file in a string and sends it back to the client, which is then displayed on the client terminal. If the file is not found, or the file is found but it is empty (i.e., even if the file is found but the content is not found), in such cases, server sends back an error message which is received by the client and is displayed on the client terminal. After the client is done with the interaction, the client will enter an “exit” command or message to terminate the connection.

Q.7. Write a program for "Remote Command Execution" using sockets. (Description: The client sends command(s) and data parameters (if required) to the server. The server will execute the command and send back the result to the client. Execute 3 commands.)

AIM: TO IMPLEMENT "REMOTE COMMAND EXECUTION" USING UDP IN CPP.

THEORY: 1. REMOTE COMMAND EXECUTION: Remote Command or Code Execution (RCE) is when external code is able to execute internal, operating-system level commands on a server from a distance. Also known as Arbitrary Code Execution, RCE is a concept that describes a form of cyber-attack in which the attacker can solely command the operation of another person's computing device or computer.

2. UDP CLIENT SERVER: In UDP, the client does not form a connection with the server like in TCP. Instead, the client just sends a datagram. Similarly, the server does not need to 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.

CODE:

// RCE SERVER

```
# include <iostream>
# include <cstdlib>
# include <cstring>
# include <unistd.h>
# include <sys/types.h>
# include <sys/socket.h>
# include <arpa/inet.h>
# include <netinet/in.h>
# include <fstream>

# define PORT 1194
# define MAXLINE 1024

using namespace std;

int main () {
    int sockfd;
    struct sockaddr_in servaddr, cliaddr;
    if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0){
        perror ("FAIL: SOCKET CREATION\n");
        exit (EXIT_FAILURE);
    }
    cout << "SUCCESS: SOCKET CREATED\n";
    memset (&servaddr, 0, sizeof (servaddr));
    memset (&cliaddr, 0, sizeof (cliaddr));
```

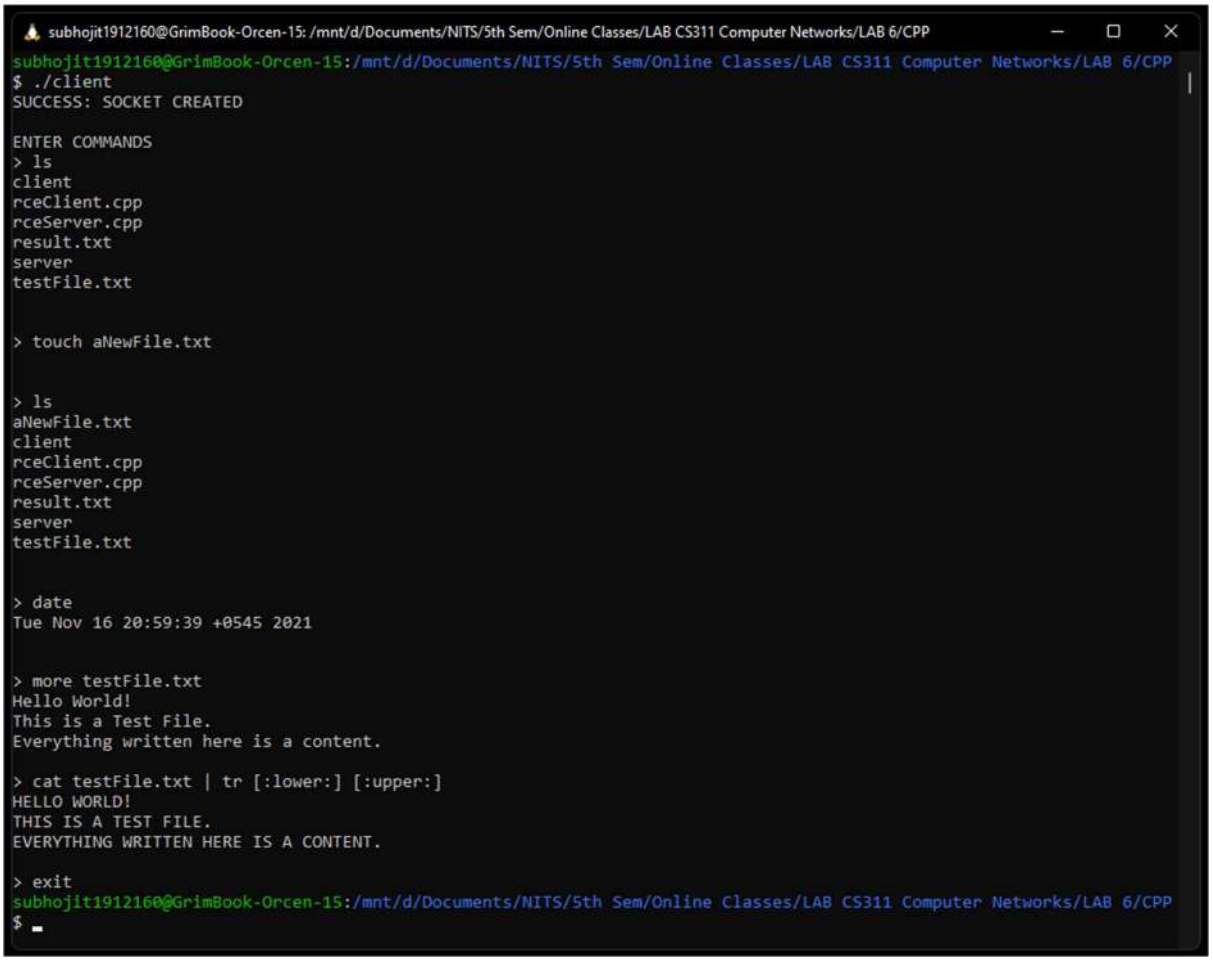
```
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port = htons (PORT);

if (bind (sockfd, (const struct sockaddr *) &servaddr,
        sizeof (servaddr)) < 0 ) {
    perror ("FAIL : SERVER BIND \n");
    exit ( EXIT_FAILURE );
}
cout << "SUCCESS : SERVER BOUND \n";

cout << "SERVER OPEN FOR INCOMING MESSAGES \n\n";
char buffer [MAXLINE];
char command [MAXLINE+15];
unsigned int len, nn;
len = sizeof (cliaddr);
while (1) {
    memset (buffer, 0, MAXLINE);
    bzero (command, MAXLINE);
    nn = recvfrom (sockfd, (char *) buffer,
        MAXLINE, MSG_WAITALL, (struct
        sockaddr *) &cliaddr, &len );
    buffer [nn] = '\0';
    cout << "REQUEST RECEIVED. EXECUTING
        COMMAND: " << buffer;
    strcpy (command, buffer);
    nn = strlen (command);
    command [nn-1] = '\0';
    strcat (command, " > result.txt ");
```

```
        system (command);
        FILE *ff = fopen ("result.txt", "r");
        bzero (buffer, MAXLINE);
        for (nn=0 ; !feof (ff) ; ++nn)
                buffer [nn] = getc (ff);
        buffer [nn-1] = '\0';
        cout << "SUCCESS: RESULT SENT BACK TO CLIENT";
        cout << endl << endl;
        sendto (sockfd, (const char *) buffer, strlen(buffer),
                MSG_CONFIRM, (const struct sockaddr *)
                &cliaddr, len);
    }
    close (sockfd);
    return 0;
}

// RCE CLIENT

# include <iostream>
# include <cstdlib>
# include <cstring>
# include <unistd.h>
# include <sys/types.h>
# include <sys/socket.h>
# include <arpa/inet.h>
# include <netinet/in.h>

# define PORT 1194
# define MAXLINE 1024
```

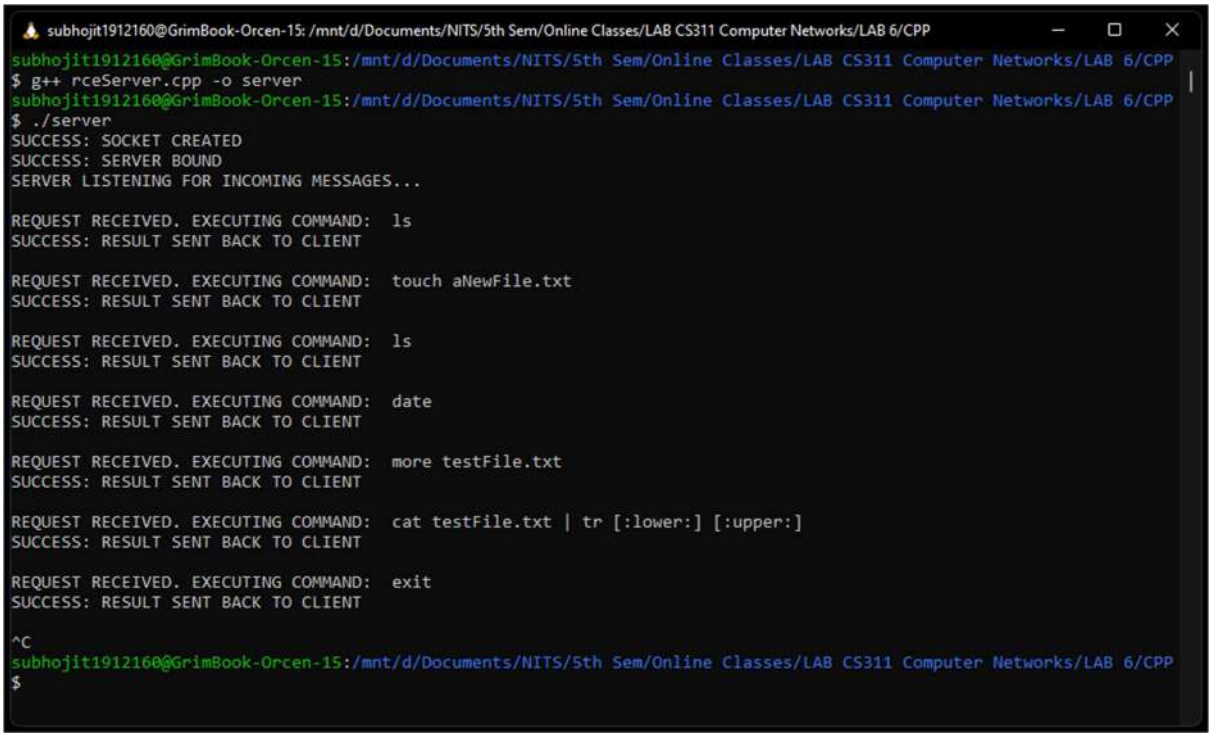
```
using namespace std;
int main () {
    int sockfd;
    struct sockaddr_in servaddr;
    if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) <0))
            exit (EXIT_FAILURE);
    cout << "SUCCESS : SOCKET CREATED \n\n";
    memset (&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons (PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;
    char buffer [MAXLINE], command [MAXLINE];
    cout << "ENTER COMMANDS";
    unsigned int nn, len;
    while (1) {
        memset (command, 0, MAXLINE);
        memset (buffer, 0, MAXLINE);
        cout << "\n> " ;
        fgets (command, MAXLINE, stdin);
        sendto (sockfd, (const char *) command, strlen
            (command), MSG_CONFIRM, (const struct
            sockaddr *) &servaddr, sizeof (servaddr));
        nn = recvfrom (sockfd, (char *) buffer, MAXLINE,
            MSG_WAITALL, (struct sockaddr *) &servaddr,
            &len);
        buffer [nn] = '\0';
        if (strncmp (command, "exit", 4) == 0)
            break;
        cout << buffer <<endl;
    }
    close (sockfd); return 0;
}
```

OUTPUT:

// RCE CLIENT



// RCE SERVER



Output Explanation:

Firstly, the Remote Command Execution Server and Client are compiled and run. After the connection is established, in the client terminal, the command to be executed is entered. This command is sent to the server as a message request. The server then runs the requested command, saves the result in buffer and sends the buffer back to client. The client then displays the buffer sent by the server, which is actually the result of the command sent by the client machine. For example, as seen in the output images above, the client sends “ls” as the command request. The server then executes “ls” command on its machine and sends back the result to client. The client then displays the result, which, in this case, is listing of all files present in the currently open folder location of the server. Similarly, client send the command “touch aNewFile.txt”. The server receives the request and executes the command. This “touch” command line utility is used to create a new file. So, in the third time, when the client requests to see all the listed files, the newly created file is also displayed, as it gets created in the second command. Similarly, the other commands are requested by the client, the server executed the command on its machine and sends the result back to the client, which is then displayed on the client terminal.

Q08. Write a program to implement "Web Server". (Description: The Client will be requesting a web page to be accessed which resides at the Server side.)

AIM: TO IMPLEMENT "WEB SERVER" USING TCP SOCKET IN CPP.

THEORY: 1. Web Server: A web server is computer software and underlying hardware that accepts requests via HTTP, the network protocol created to distribute web content, or its secure variant HTTPS.

2. TCP CLIENT SERVER: TCP (Transmission Control Protocol) is a transport layer in a networking service. The client in TCP/IP connection is the device that dials the phone and the server is the device that is listening in for the calls to come in.

CODE:

// WS SERVER

```
# include <iostream>
# include <cstdlib>
# include <cstring>
# include <netdb.h>
# include <netinet/in.h>
# include <sys/socket.h>
# include <sys/types.h>
# include <unistd.h>

# define MAX 1024
# define PORT 8080
using namespace std;

int main() {
    int sockfd = socket (AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in servaddr;
    if ( sockfd != -1 )
        cout << " SUCCESS: SOCKET CREATED\n";
    else {
        perror ("ERROR: SOCKET CREATION ");
        exit (EXIT_FAILURE);
    }
    bzero (&servaddr, sizeof (servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    servaddr.sin_port = htons (PORT);
```

```
if (!bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)))
        cout << "SUCCESS: SOCKET BINDED\n";
else {
    perror ("ERROR : SOCKET BINDING ");
    exit (EXIT_FAILURE);
}

struct sockaddr_in cli;
unsigned int len = sizeof (cli);
if (!(listen (sockfd, 5)))
        cout << "SUCCESS : SERVER LISTENING\n";
else {
    perror ("ERROR : SERVER LISTEN.");
    exit (EXIT_FAILURE);
}

int connfd = accept (sockfd, (struct sockaddr *)&cli, &len);
if ( connfd >= 0)
        cout << "SUCCESS : CLIENT ACCEPTED \n\n";
else {
        perror ("ERROR : CLIENT ACCEPTION");
        exit (EXIT_FAILURE);
}

char buffer[MAX], buff[MAX];
while (1) {
        bzero (buffer, MAX);
        bzero (buff, MAX);
        read (connfd, buffer, sizeof (buffer));
```

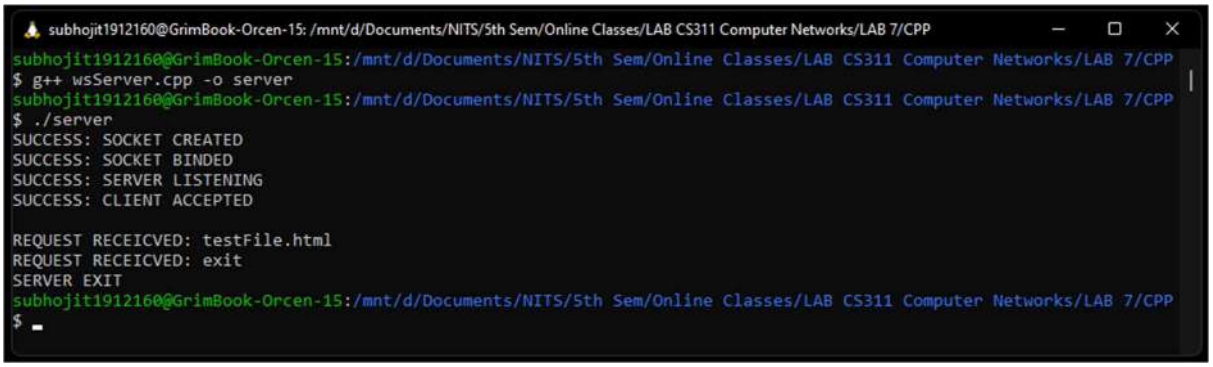
```
        strcpy (buff, buffer);
        buff [strlen (buff)-1] = '\0';
        cout << "REQUEST RECEIVED: " << buff <<endl;
        if (!strncmp (buff, "exit", 4))
                break;
        FILE *ff = fopen (buff, "r");
        while (!feof (ff)) {
                bzero (buffer, MAX);
                fgets (buffer, MAX, ff);
                write (connfd, buffer, sizeof (buffer));
        }
        fclose (ff);
        bzero (buffer, MAX);
        strcpy (buffer, "END");
        write (connfd, buffer, sizeof (buffer));
    }
    cout << "SERVER EXIT \n";
    close (sockfd);
    return 0;
}

// WS CLIENT

#include <iostream>
#include <cstdlib>
#include <cstring>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>
```

```
#define MAX 1024
# define PORT 8080

int main () {
    int sockfd = socket (AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in servaddr;
    if (sockfd != -1)
        cout << " SUCCESS: SOCKET CREATED\n";
    else {
        perror (" ERROR : SOCKET CREATION ");
        exit (EXIT-FAILURE);
    }
    bzero (&servaddr, sizeof (servaddr));

    servaddr.sin-family = AF-INET;
    servaddr.sin-addr.s-addr = htonl (INADDR_ANY);
    servaddr.sin-port = htons ( PORT);

    if (!(connect (sockfd, (struct sockaddr *) &servaddr,
        sizeof (servaddr))))
        cout << " SUCCESS : CONNECTE TO SERVER \n\n";
    else {
        perror (" ERROR: SERVER CONNECTION ");
        exit (EXIT_FAILURE);
    }

    cout << " ENTER FILENAME TO OPEN: ";
    char buffer [MAX], buff [MAX];
```

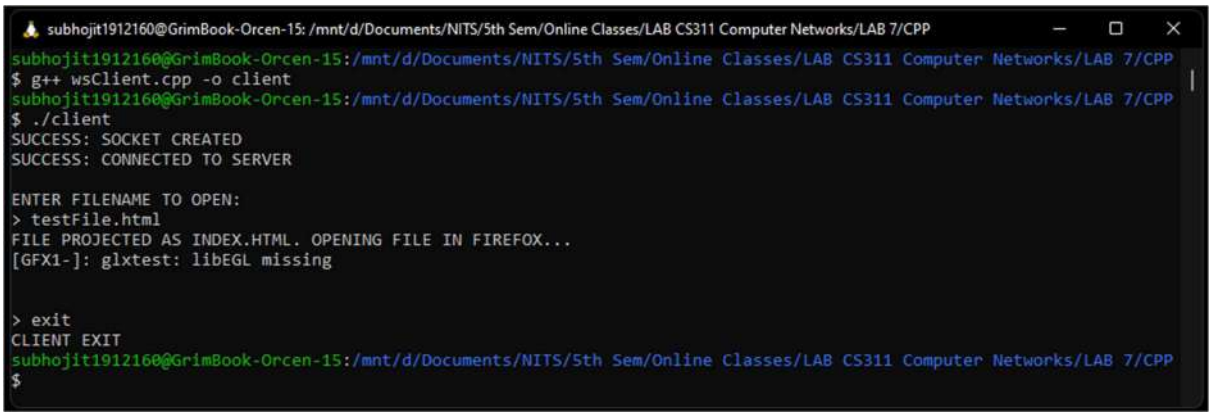
```
while (1) {
    bzero (buff, sizeof (buff));
    bzero (buffer, sizeof (buffer));
    cout << "\n> ";
    fgets (buff, MAX, stdin);
    write (sockfd, buff, sizeof (buff));
    if (!strncmp (buff, "exit", 4))
        break;
    FILE *ff = fopen ("index.html", "w");
    while (1) {
        bzero (buffer, sizeof (buffer));
        read (sockfd, buffer, sizeof (buffer));
        if (strncmp (buffer, "END", 3) == 0)
            break;
        fputs (buffer, ff);
    }
    fclose (ff);
    cout << "FILE PROJECTED AS INDEX.HTML.
            OPENING FILE IN FIREFOX...\n";
    system ("firefox index.html");
    int c = getchar ();
    system ("rm index.html");
}
cout << "CLIENT EXIT\n";
close (sockfd);
return 0;
}
```

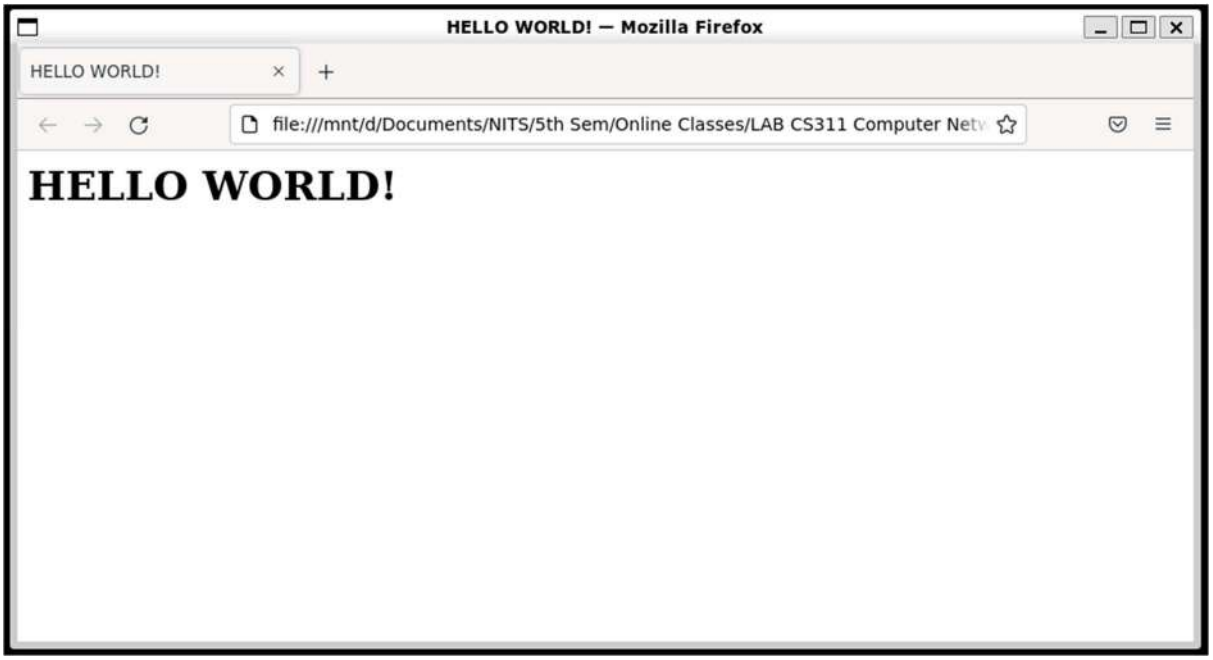
OUTPUT:

// WS SERVER



// WS CLIENT





Output Explanation:

The client requests a web-page from the server. The server receives the requests and responds accordingly by sending back the web-page resources so that the client can view the web-page on his/her side. The client program, after receiving the web-page, opens it on Firefox for the client to view. When the client is done, the “exit” command is sent to close the processes.