Title：Socket Programming

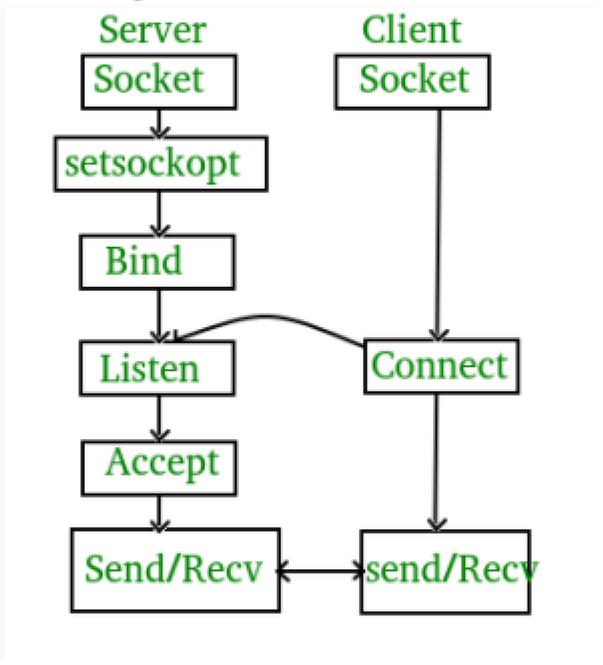
Name: Dikshya Kafle
Stu No: 2018380039
Class No: 10101801
Deadline: Dec 13th 2020

School of Computer Science and Engineering

■ Title: Socket programming

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

State diagram for server and client model



1 .Purpose:

Implement the mutual communication between two hosts based on TCP or UDP.

2.principle:

Principle: Communication procedure of TCP and UDP

Requirement:

a)Implement one-way data transmission. One sends data, and the other receives data.

b)Implement Client and Server send and receive data at the same time.

c)Try to transmit a media file and analyze the features of TCP/UDP

3.Client/Server program based on TCP:

bind() - is used to bind a socket and the ip address and port of a hosts.

listen() - is used in the connection-oriented server to listen to the connection of a client.

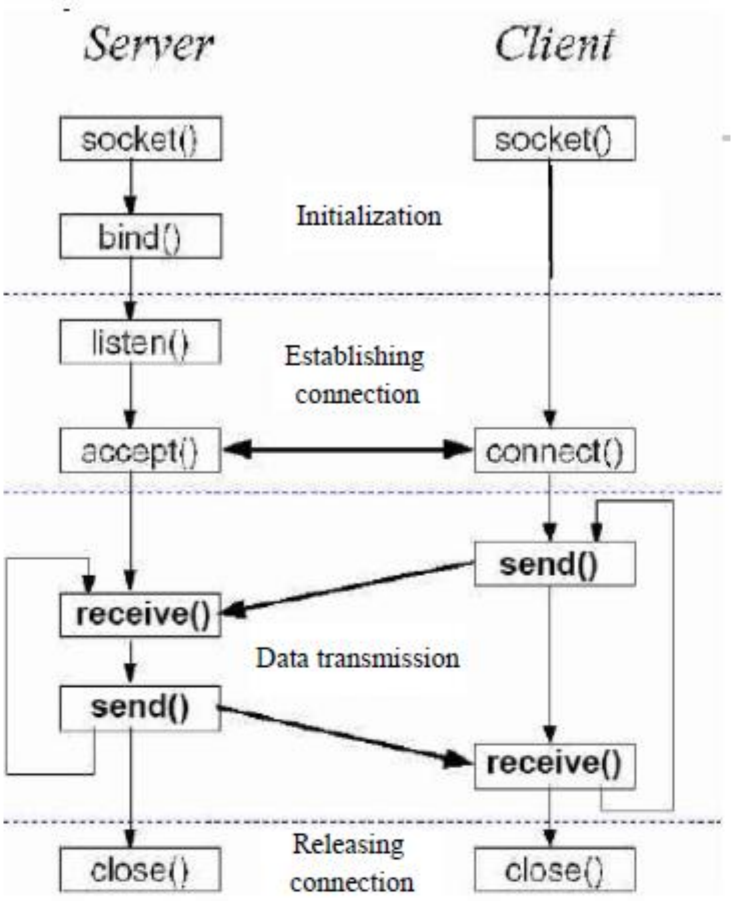
accept() - is responsible to fetch a connection from the array of established connection and bind it and s.

connect() - is used to create the connection to the specific port of a server.

send() - No matter processes in the client or processes in the server send the data to the other end of TCP by using send.

recv() - No matter processes in the client or processes in the server receive the data from the other end of

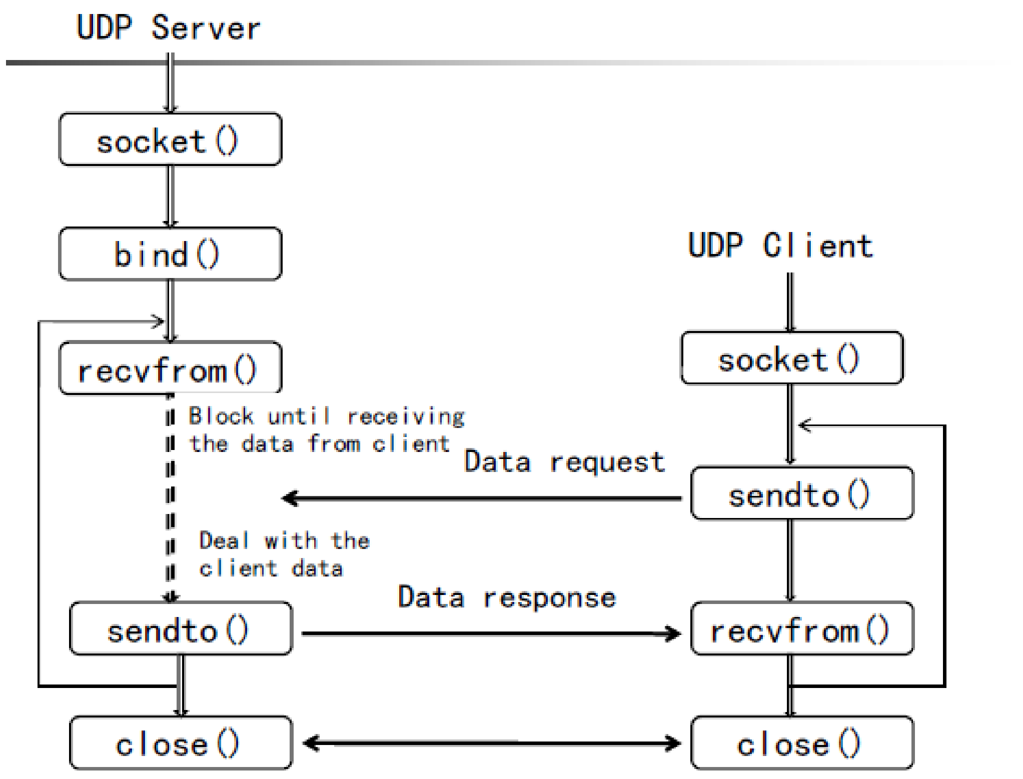
TCP by using recv.
closesocket() - closes the sockets.



Working of TCP Client- Server divides in two parts:
Server:
1. Create a socket.
2. Bind the socket to an address.
3. Listen for incoming connections.
4. Accept a connection. This call typically blocks until a client connects with the server.
5. Send and receive data to/from server.
6. Close the connection.

Client:
1. Create a socket.
2. Connect the socket to the address of the server.
3. Send and receive data from/to server.
4. Close the connection.

4.Client/Server program based on UDP



.5.Implementation details

TCP Server:

1. Initialize the Winsock DLL using WSAStartup function.
2. Fill server sock_addr structure.
3. Create a socket with the socket() function.
4. Bind the socket to an address using the bind() function.
5. Listen for connections with the listen() function.
6. Accept a connection with the accept() function system call. This call typically blocks until a client connects with the server.
7. Send and receive data by means of send() function and recv() function.
8. Close the connection by means of the CloseSocket() function.
9. Remove the Winsock DLL from WSACleanup() function.

TCP Client:

1. Initialize the Winsock DLL using WSAStartup function.
2. Fill server sock_addr structure.
3. Create a socket with the socket() function.
4. Connect to Server using connect() function.
5. Send and receive data by means of send() function and recv() function.

6. Close the connection by means of the CloseSocket() function.
7. Remove the Winsock DLL from WSACleanup() function.

1. **WSAStartup – initiate the use of Winsock DLL.**

   The parameters are:

   i.wVersionRequired – version of Winsock DLL. Current version is 2.2
   it is WORD type. So, we have to use MAKEWORD function to Convert this version into WORD type.
   Example- MAKEWORD(2,2).

   ii.lpWSAData – A pointer to WSADATA Structure. Basically, it receives the information for Windows Socket Creation.

   iii. WSAStartup API Link.

2. **sockaddr Structure**

   Fill the Socket Information with the help of sockaddr Structure. Below is the IPv4 sockaddr structure.
   We have to fill only 3 parameters of this structure:

   i. *sin_family:* tells about SOCKET Type.
   ii. *sin_port:*Provide port number.
   iii. *struct in_addr sin_addr*

3. **socket – creates socket for transport service provider.**

It creates socket. It transports data to end point node.
i. af – Address family.
ii. type- Tells about socket type.
iii. Protocol

4. **Bind – It provides local address to socket**

   This function basically used in Server-side process. It binds local address and port to socket. The bind parameters are:

   i.SOCKET s – Pass Socket descriptor. Whatever used in socket function.
   ii.sockaddr *addr – Pass sockaddr Structure variable. Which is a type of sockaddr pointer.
   iii.namelen – size of sockaddr structure.

5. **listen – listening for incoming Signal**

   The parameters are:

   i.SOCKET s – Pass Socket descriptor. Whatever used in socket function.
   ii.backlog – Maximum length of pending connection or how many connections can be allowed.
   Maximum connection allowed is 65535.

6. **accept –Accept the incoming Signal from Socket**

   The accept parameters are:

   i.SOCKET s – Pass Socket descriptor. Whatever used in socket function.

ii.sockaddr *addr – Pass sockaddr Structure variable. Which is a type of sockaddr pointer.
iii. *addrlen – size of sockaddr structure. It is pointer type.

7. **connect – Make a connection to Specified Socket.**

This function used in Client Side. The connect parameters are:
i.SOCKET s – Pass Socket descriptor. Whatever used in Socket function
ii.sockaddr *addr – Pass sockaddr Structure variable. Which is a type of sockaddr pointer.

8. **recv – Receive the incoming data from Socket**

This function mainly uses in TCP. The receive parameters are:
i. SOCKET s – Socket descriptor. Whatever used in accept function return value.
ii. *buf – Buffer.
iii. len – Buffer length.
iv. flags – flags for incoming stream. Default is Zero.

9. **send – Send data to a connected Socket**

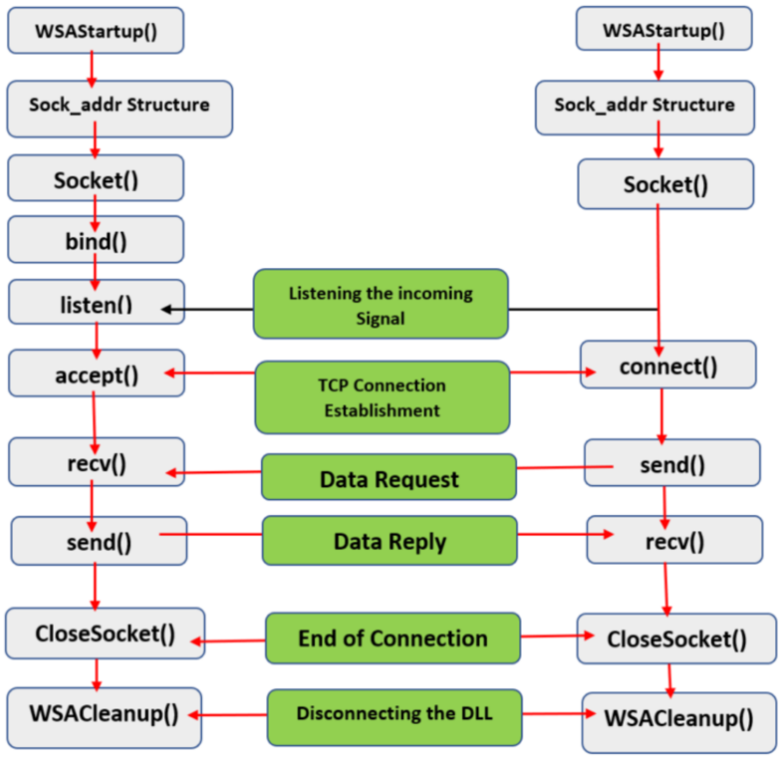
This function mainly uses in TCP. The sending Parameters are:
i. SOCKET s – Socket descriptor.
ii. *buf – Buffer. It is pointer type.
iii. len – Buffer length.
iv. flags – flags for sending data to Socket. Default is Zero.

10. **CloseSocket – Closing the Socket.**

This function used to close the existing open socket.

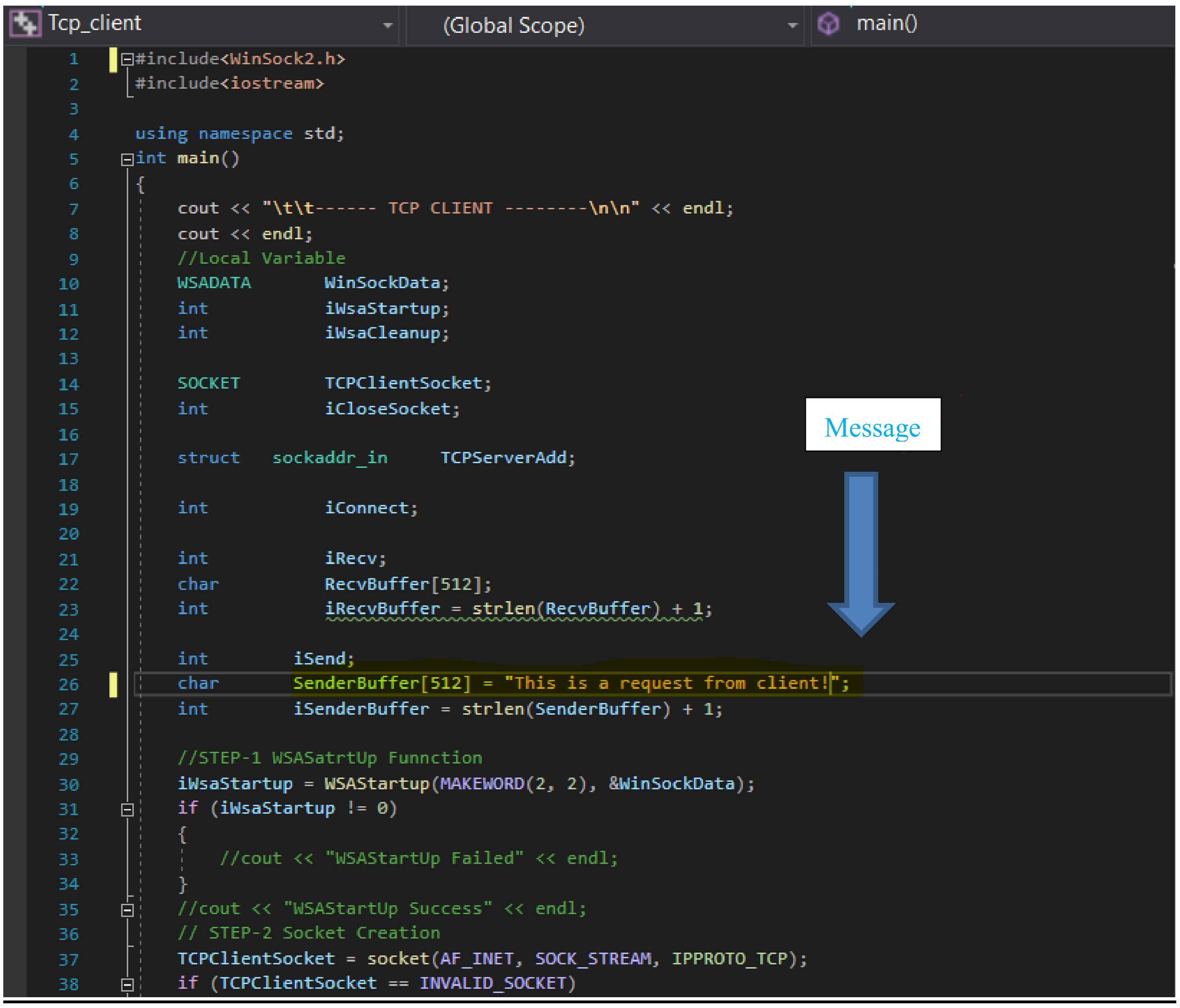
11. **WSACleanup –Terminate the use of Winsock DLL**

This function used to remove the Winsock DLL. It returns ZERO if it successes and returns SOCKET_ERROR if it fails.

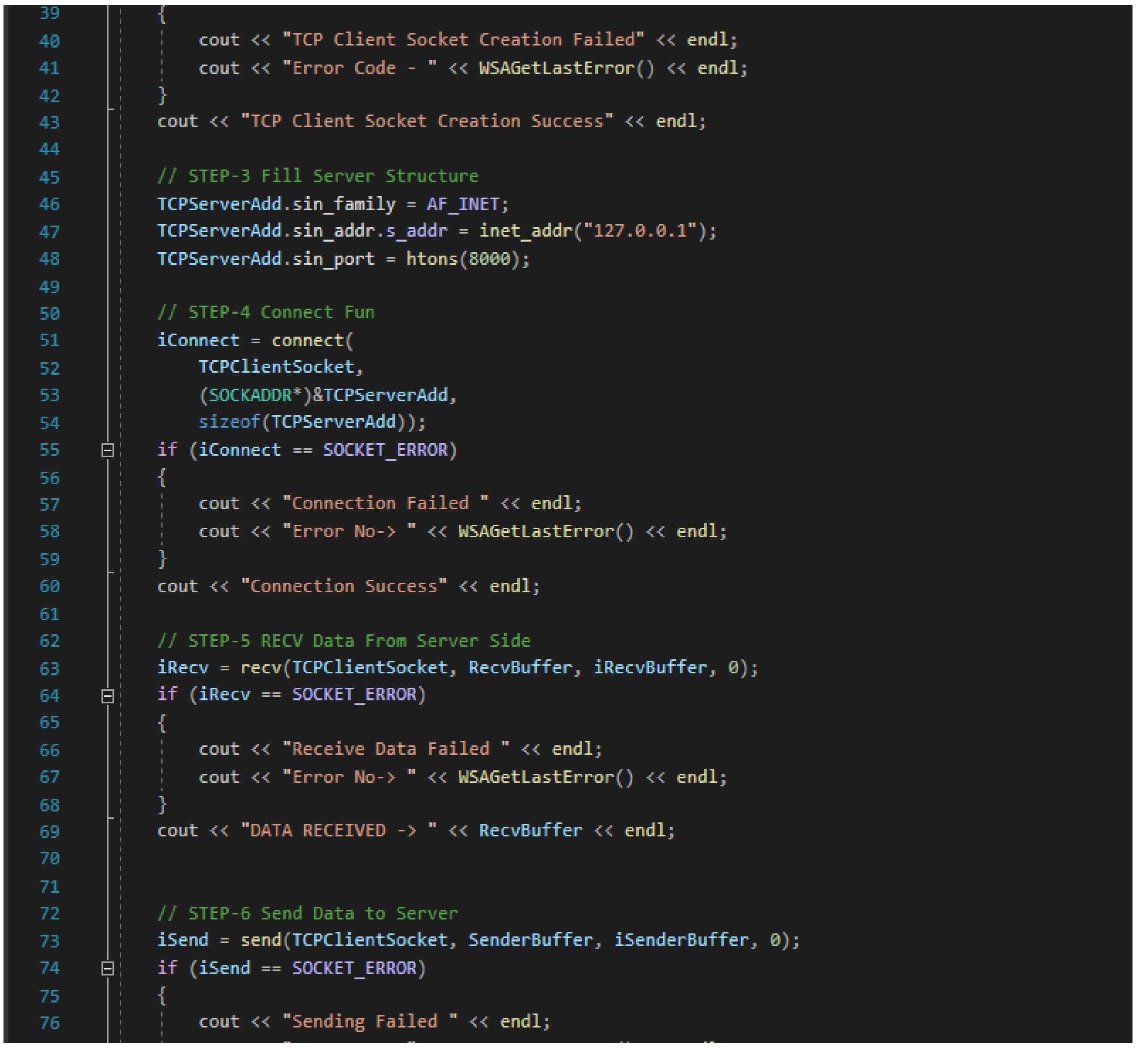


Results:

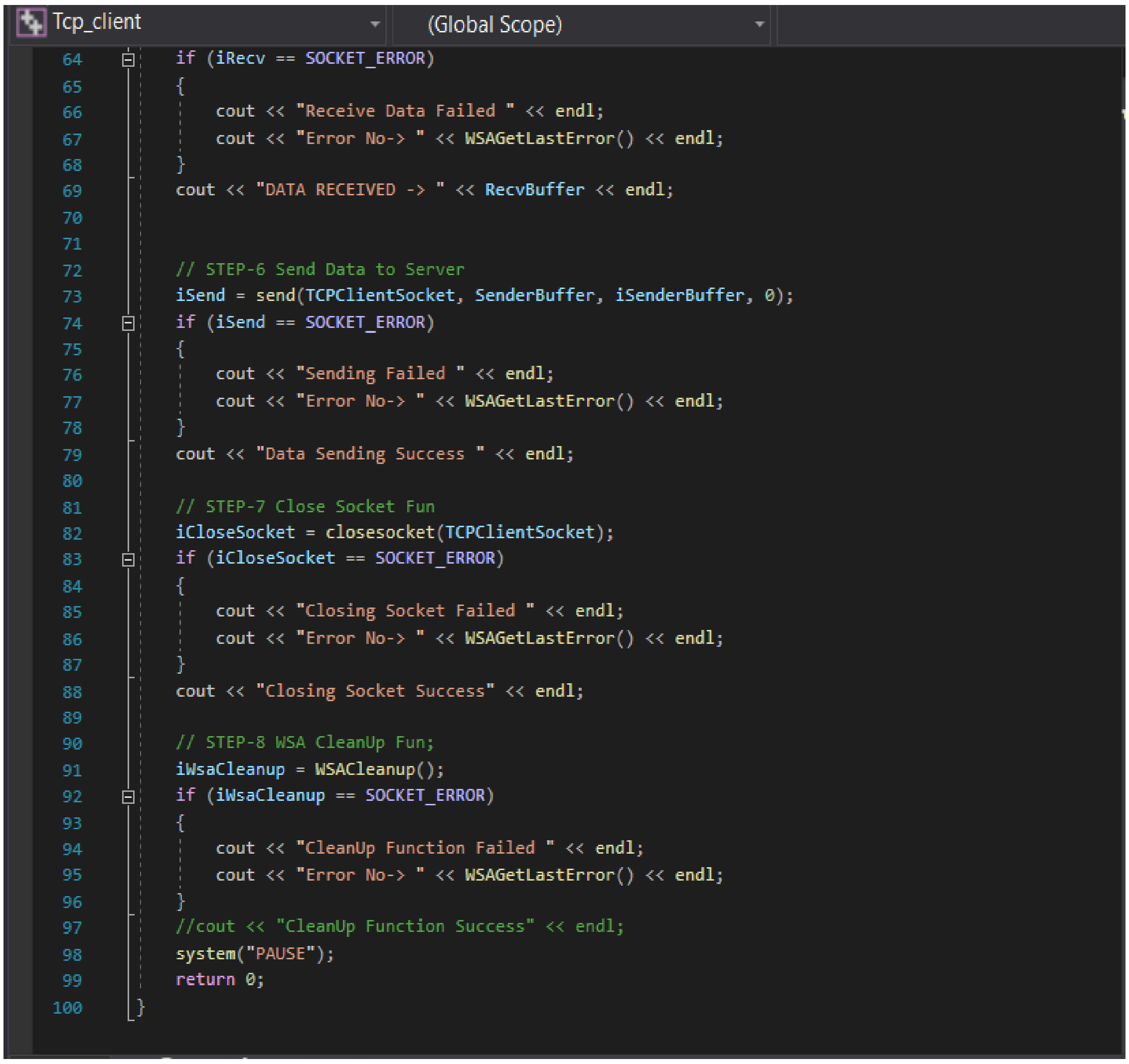
TCP_CLIENT:



```
#include<WinSock2.h>
#include<iostream>

using namespace std;
int main()
{
    cout << "\t\t------ TCP CLIENT --------\n\n" << endl;
    cout << endl;
    //Local Variable
    WSADATA        WinSockData;
    int            iWsaStartup;
    int            iWsaCleanup;

    SOCKET         TCPClientSocket;
    int            iCloseSocket;

    struct    sockaddr_in     TCPServerAdd;

    int            iConnect;

    int            iRecv;
    char           RecvBuffer[512];
    int            iRecvBuffer = strlen(RecvBuffer) + 1;

    int         iSend;
    char        SenderBuffer[512] = "This is a request from client!";
    int         iSenderBuffer = strlen(SenderBuffer) + 1;

    //STEP-1 WSAStartUp Funnction
    iWsaStartup = WSAStartup(MAKEWORD(2, 2), &WinSockData);
    if (iWsaStartup != 0)
    {
        //cout << "WSAStartUp Failed" << endl;
    }
    //cout << "WSAStartUp Success" << endl;
    // STEP-2 Socket Creation
    TCPClientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (TCPClientSocket == INVALID_SOCKET)
```

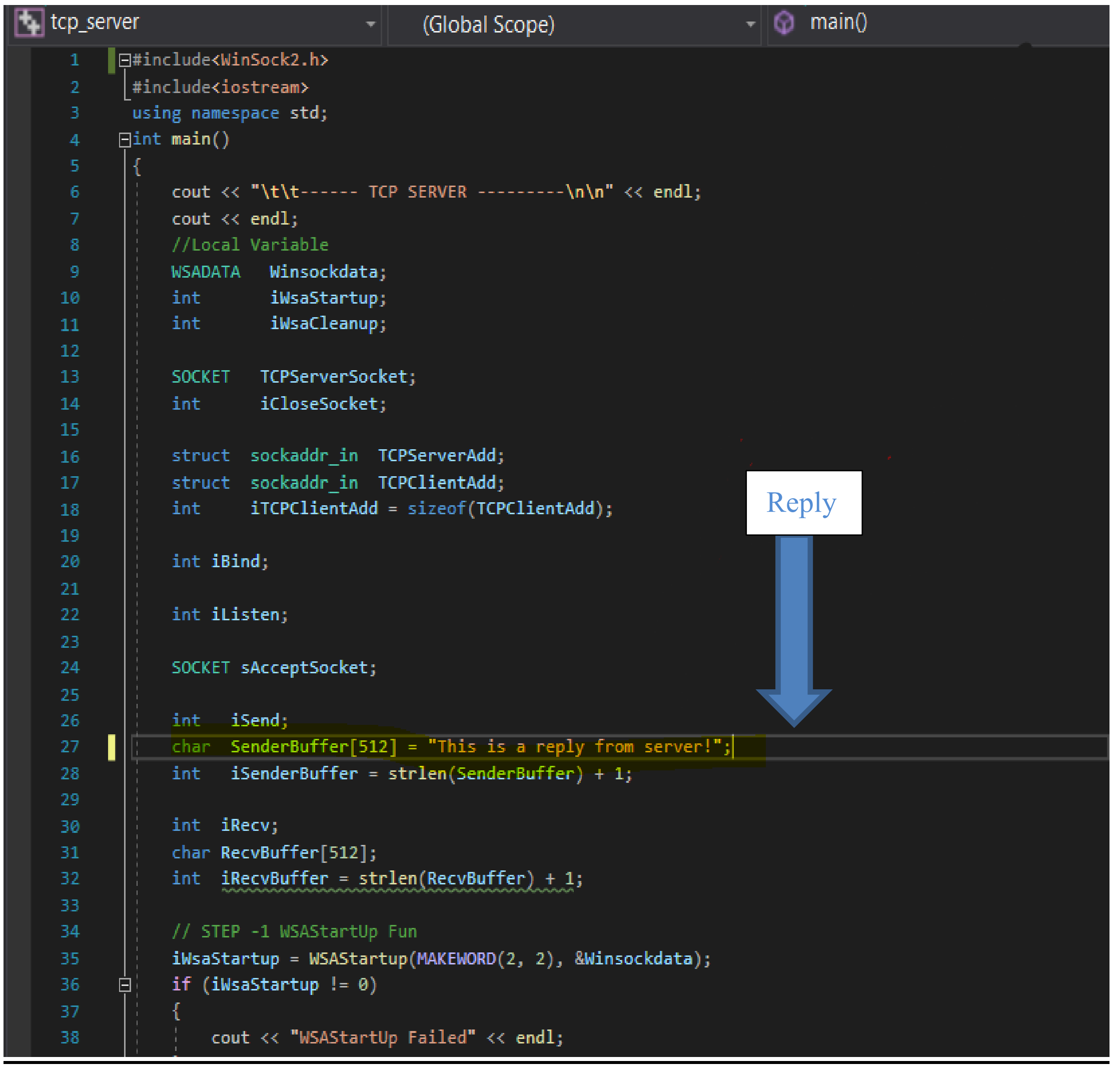


```
    {
        cout << "TCP Client Socket Creation Failed" << endl;
        cout << "Error Code - " << WSAGetLastError() << endl;
    }
    cout << "TCP Client Socket Creation Success" << endl;

    // STEP-3 Fill Server Structure
    TCPServerAdd.sin_family = AF_INET;
    TCPServerAdd.sin_addr.s_addr = inet_addr("127.0.0.1");
    TCPServerAdd.sin_port = htons(8000);

    // STEP-4 Connect Fun
    iConnect = connect(
        TCPClientSocket,
        (SOCKADDR*)&TCPServerAdd,
        sizeof(TCPServerAdd));
    if (iConnect == SOCKET_ERROR)
    {
        cout << "Connection Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    cout << "Connection Success" << endl;

    // STEP-5 RECV Data From Server Side
    iRecv = recv(TCPClientSocket, RecvBuffer, iRecvBuffer, 0);
    if (iRecv == SOCKET_ERROR)
    {
        cout << "Receive Data Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    cout << "DATA RECEIVED -> " << RecvBuffer << endl;


    // STEP-6 Send Data to Server
    iSend = send(TCPClientSocket, SenderBuffer, iSenderBuffer, 0);
    if (iSend == SOCKET_ERROR)
    {
        cout << "Sending Failed " << endl;
```

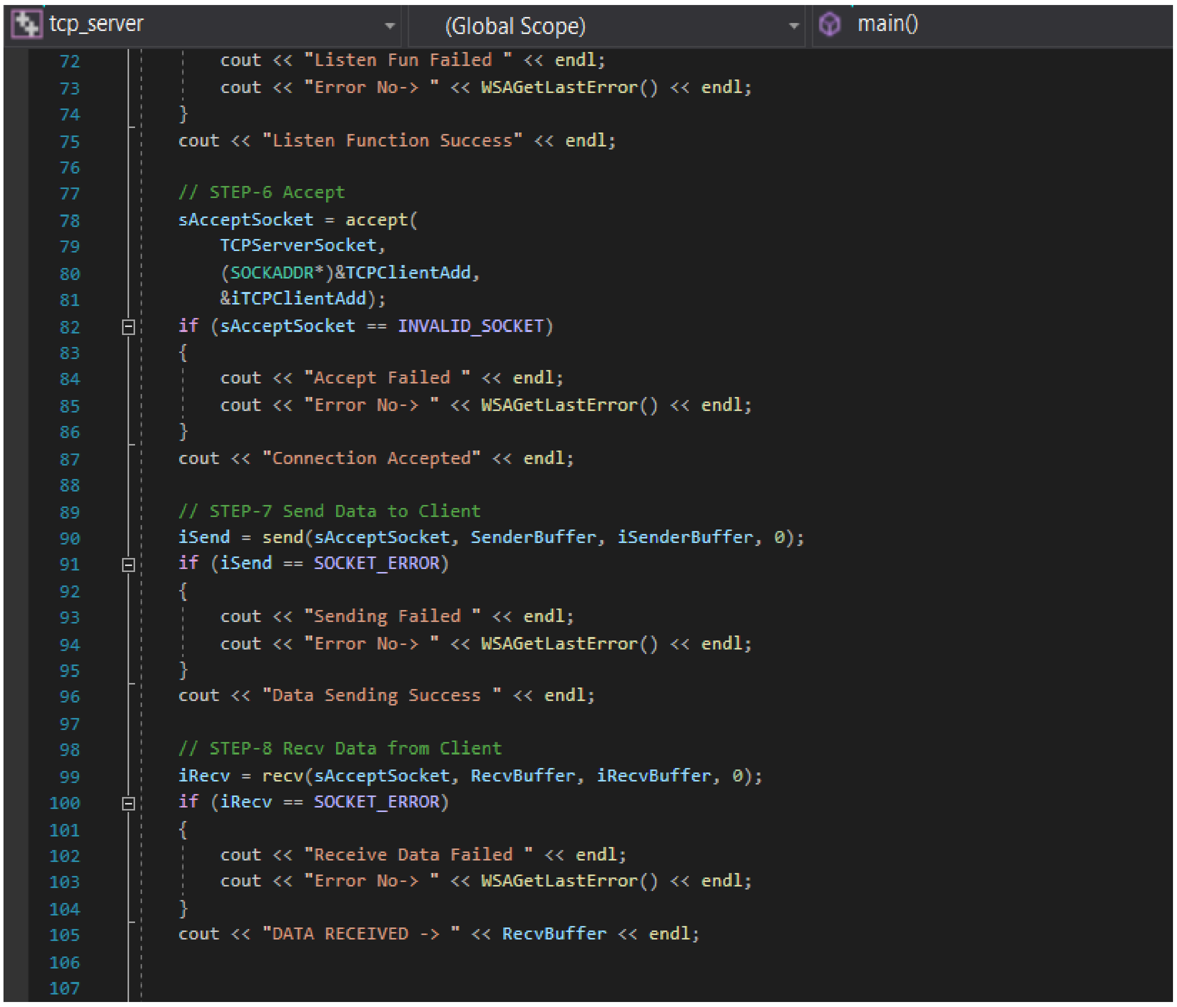


```
    if (iRecv == SOCKET_ERROR)
    {
        cout << "Receive Data Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    cout << "DATA RECEIVED -> " << RecvBuffer << endl;


    // STEP-6 Send Data to Server
    iSend = send(TCPClientSocket, SenderBuffer, iSenderBuffer, 0);
    if (iSend == SOCKET_ERROR)
    {
        cout << "Sending Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    cout << "Data Sending Success " << endl;

    // STEP-7 Close Socket Fun
    iCloseSocket = closesocket(TCPClientSocket);
    if (iCloseSocket == SOCKET_ERROR)
    {
        cout << "Closing Socket Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    cout << "Closing Socket Success" << endl;

    // STEP-8 WSA CleanUp Fun;
    iWsaCleanup = WSACleanup();
    if (iWsaCleanup == SOCKET_ERROR)
    {
        cout << "CleanUp Function Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    //cout << "CleanUp Function Success" << endl;
    system("PAUSE");
    return 0;
}
```

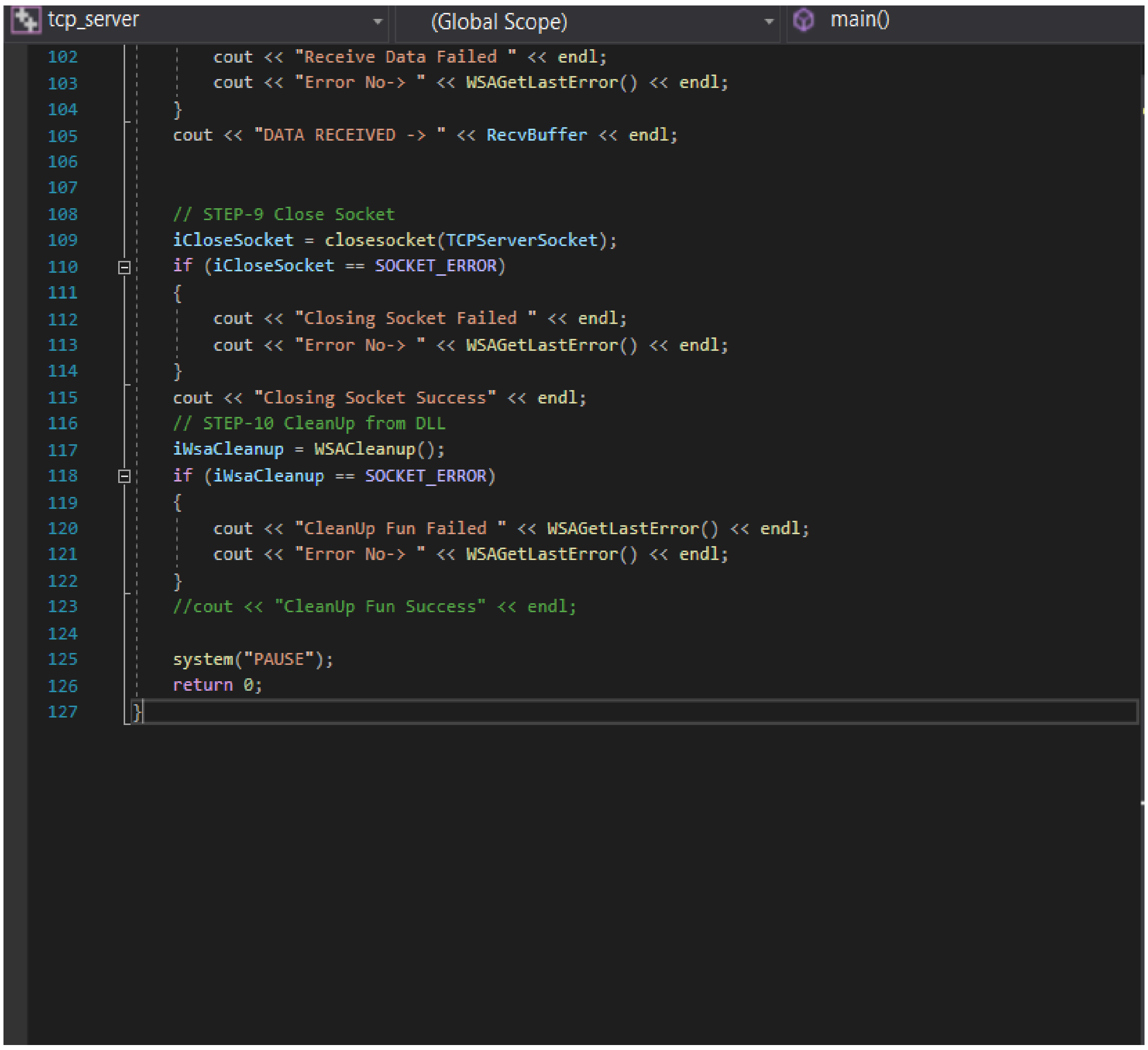
TCP SERVER:



```
#include<WinSock2.h>
#include<iostream>
using namespace std;
int main()
{
    cout << "\t\t------ TCP SERVER ---------\n\n" << endl;
    cout << endl;
    //Local Variable
    WSADATA   Winsockdata;
    int       iWsaStartup;
    int       iWsaCleanup;

    SOCKET   TCPServerSocket;
    int      iCloseSocket;

    struct  sockaddr_in  TCPServerAdd;
    struct  sockaddr_in  TCPClientAdd;
    int     iTCPClientAdd = sizeof(TCPClientAdd);

    int iBind;

    int iListen;

    SOCKET sAcceptSocket;

    int   iSend;
    char  SenderBuffer[512] = "This is a reply from server!";
    int   iSenderBuffer = strlen(SenderBuffer) + 1;

    int  iRecv;
    char RecvBuffer[512];
    int  iRecvBuffer = strlen(RecvBuffer) + 1;

    // STEP -1 WSAStartUp Fun
    iWsaStartup = WSAStartup(MAKEWORD(2, 2), &Winsockdata);
    if (iWsaStartup != 0)
    {
        cout << "WSAStartUp Failed" << endl;
```



```
        cout << "Listen Fun Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    cout << "Listen Function Success" << endl;

    // STEP-6 Accept
    sAcceptSocket = accept(
        TCPServerSocket,
        (SOCKADDR*)&TCPClientAdd,
        &iTCPClientAdd);
    if (sAcceptSocket == INVALID_SOCKET)
    {
        cout << "Accept Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    cout << "Connection Accepted" << endl;

    // STEP-7 Send Data to Client
    iSend = send(sAcceptSocket, SenderBuffer, iSenderBuffer, 0);
    if (iSend == SOCKET_ERROR)
    {
        cout << "Sending Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    cout << "Data Sending Success " << endl;

    // STEP-8 Recv Data from Client
    iRecv = recv(sAcceptSocket, RecvBuffer, iRecvBuffer, 0);
    if (iRecv == SOCKET_ERROR)
    {
        cout << "Receive Data Failed " << endl;
        cout << "Error No-> " << WSAGetLastError() << endl;
    }
    cout << "DATA RECEIVED -> " << RecvBuffer << endl;


```

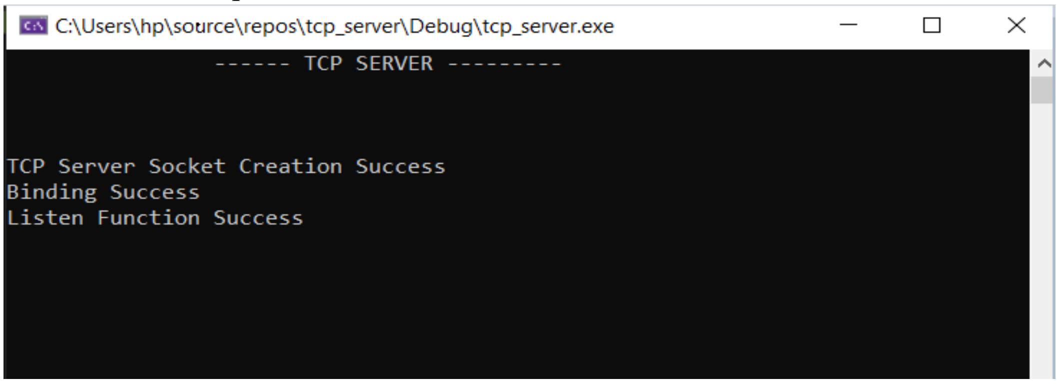


```
            cout << "Receive Data Failed " << endl;
            cout << "Error No-> " << WSAGetLastError() << endl;
        }
        cout << "DATA RECEIVED -> " << RecvBuffer << endl;


        // STEP-9 Close Socket
        iCloseSocket = closesocket(TCPServerSocket);
        if (iCloseSocket == SOCKET_ERROR)
        {
            cout << "Closing Socket Failed " << endl;
            cout << "Error No-> " << WSAGetLastError() << endl;
        }
        cout << "Closing Socket Success" << endl;
        // STEP-10 CleanUp from DLL
        iWsaCleanup = WSACleanup();
        if (iWsaCleanup == SOCKET_ERROR)
        {
            cout << "CleanUp Fun Failed " << WSAGetLastError() << endl;
            cout << "Error No-> " << WSAGetLastError() << endl;
        }
        //cout << "CleanUp Fun Success" << endl;

        system("PAUSE");
        return 0;
    }
```

Execution of the code:

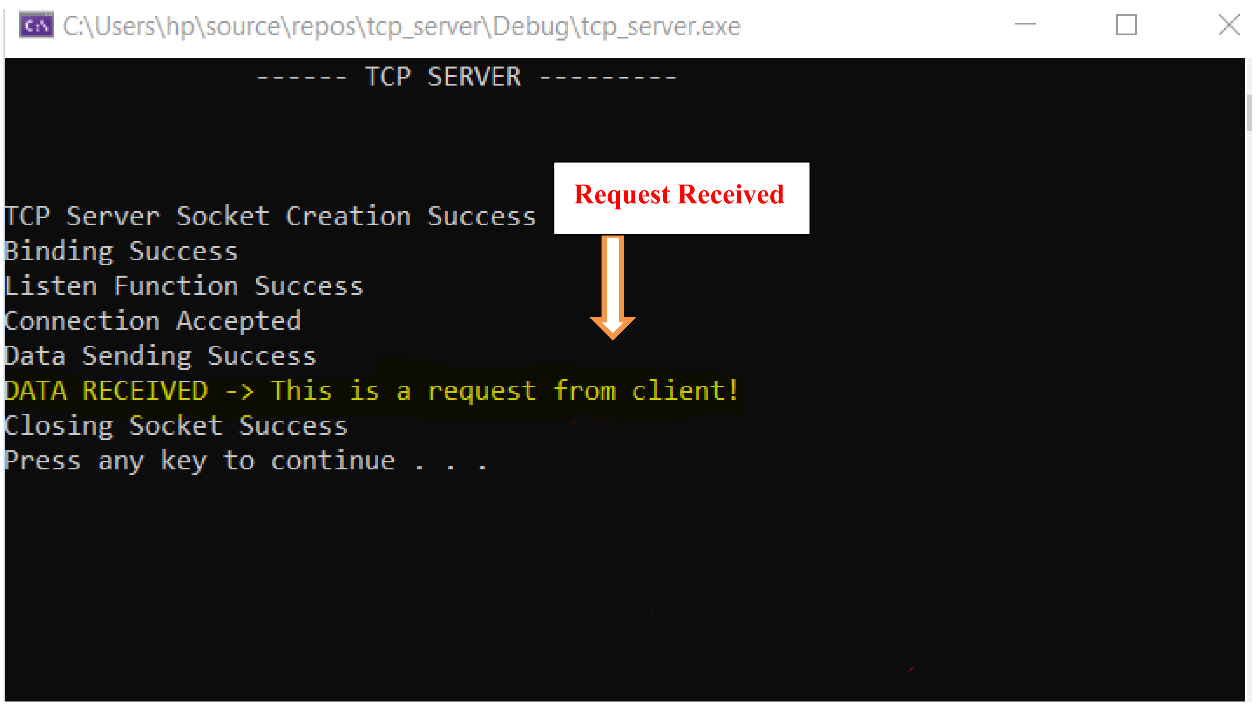
First we run the code of the TCP_server and then second we run the TCP_client. Until we run the tcp_client that will submit a request and wait for the server to answer, the tcp _server will wait for the request.

The server awaits a request:



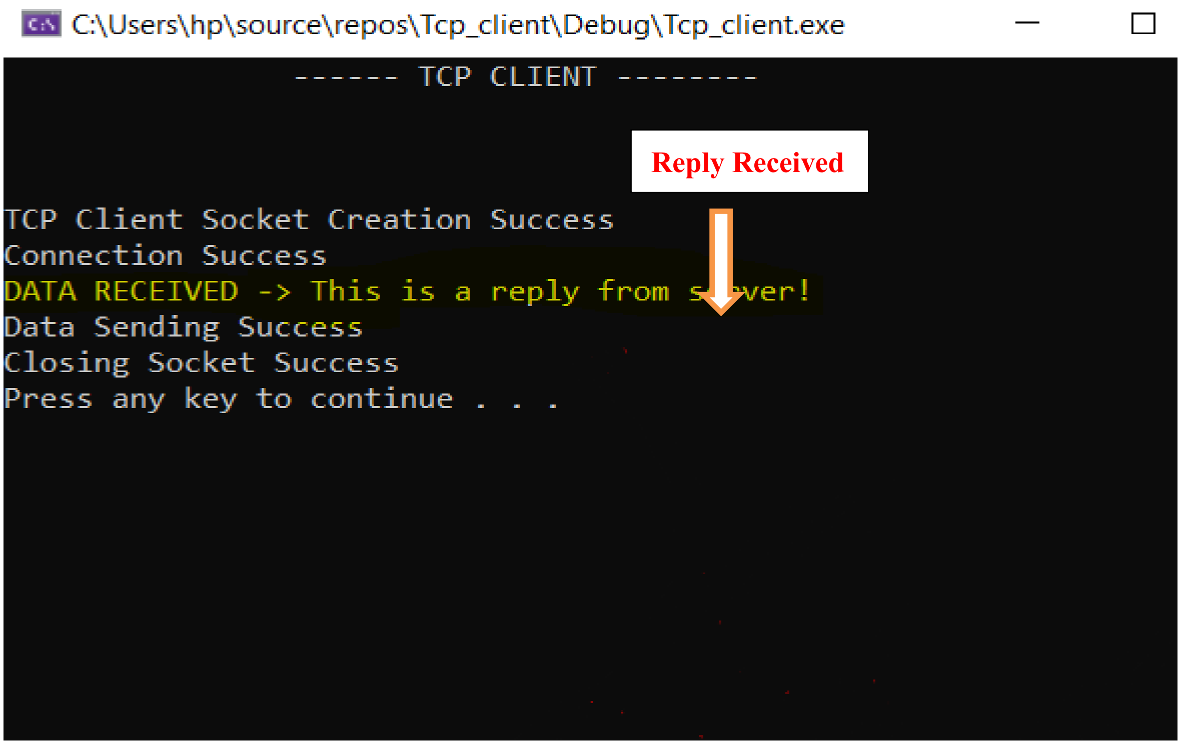
```
                ------ TCP SERVER ---------


TCP Server Socket Creation Success
Binding Success
Listen Function Success
```

Request Success:



```
                ------ TCP SERVER ---------


TCP Server Socket Creation Success
Binding Success
Listen Function Success
Connection Accepted
Data Sending Success
DATA RECEIVED -> This is a request from client!
Closing Socket Success
Press any key to continue . . .
```

Reply Success:



```
              ------ TCP CLIENT --------


TCP Client Socket Creation Success
Connection Success
DATA RECEIVED -> This is a reply from server!
Data Sending Success
Closing Socket Success
Press any key to continue . . .
```

Conclusion:

TCP is connection oriented, i.e., it creates a connection for the transmission to take place, and once the transfer is over that connection is terminated. **UDP** on the **other** hand is connectionless just like IP (Internet **Protocol**). Reliability: **TCP** sends an acknowledgement when it receives a packet.

TCP is connection oriented once a connection is established, data can be sent bidirectional. **UDP** is a simpler, connectionless Internet protocol. **UDP** is **faster than TCP**, and the simple reason is because its nonexistent acknowledge packet (ACK) that permits a continuous packet stream, instead of **TCP** that acknowledges a set of packets, calculated by using the **TCP** window size and round-trip time (RTT).

We have learned how we can implement the TCP and UDP protocols for sending text and other file via the protocols by implementing the procedures for both the sender and reciever. We have clearly understood how these protocols can be used to communicate both the server to the client and the same server-client pc(where data is transmitted to another device location).