

Jack Partridge-1702917

NetworkAppAssign2

by Jack Partridge

FILE	4111_JACK_PARTRIDGE_JACK_PARTRIDGE-1702917_NETWORKAPPASSIGN2_252504_510119945.DOCX (211.08K)		
TIME SUBMITTED	30-APR-2020 03:12PM (UTC+0100)	WORD COUNT	2370
SUBMISSION ID	125685177	CHARACTER COUNT	15865

Networked Application-Assignment 2

Jack Partridge – 1702918

April 10st, 2020

Contents

Introduction.....	1
Design & Implementation	1
TCP vs UDP.....	1
Creating the server	2
Creating the Client	4
Testing.....	5
Conclusion	6
References	7
Appendix	7

Introduction

Within this report, it aims to go over the chosen networked application and break down the stages of implementation along with the design. It will discuss the methods used and approaches to get the working solution. The chosen networked application is a multi-client TCP chat server using sockets [1] to send and receive low level data. This is where a server is running and will handle a multiple client connected at the same time.

Design & Implementation

This segment will discuss the code implementation along with what steps were taken to achieve the application. To begin we must first pick between UDP or TCP for our protocol approach [2]. After this, starting with the server side of programming followed by the client with the use of WinSock for our sockets.

TCP vs UDP.

The chosen protocol for the IRC/Chat server would be TCP over UDP.

TCP allows for the help of a three-way handshake and ensuring that packets of information are delivered. This meaning that the data is sent in order, making it reliable and data sent successfully.

UDP uses simple transmission without the use of a three-way handshake. Meaning the data isn't reliable, ordered and has no data integrity. Along with UDP it throws away all means of error checks and correction.

Creating the server

When starting on the server I had made a step required section to indicate the requirements and this was inside of the visual studio server.cpp file.

The steps are as followed:

1. Initialise Winsock
2. Create socket with address & keep track of socket descriptor.
3. Fill in a hint structure/Server address family.
4. Bind socket to IP address & port number.
5. Tell Winsock that the socket is listening.
6. Create a loop to wait for connections & broadcast a welcome message.
7. Close Socket,
8. Clean-up Winsock.

After this I ensured the use of multiple clients using Select() command along with macros/file descriptors.

File descriptors are listed below:

FD_CLR is used to Remove from a set.

FD_SET is used to Add to a set.

FD_ZERO is used to Clear a set.

FD_ISSET is used for incoming clients.

Firstly, we need to initialise Winsock and do error checking in case of failure. We will be using version 2.4 of Winsock.

```
WSADATA winSockData;  
WORD version = MAKEWORD(2, 4);  
  
int wsOk = WSAStartup(version, &winSockData);
```

Now we need to create the socket with address, that will listen and keep track of socket descriptor. We've gone with AF_INET internet here.

```
// Create the socket.  
SOCKET listening = socket(AF_INET, SOCK_STREAM, 0);
```

Once we have a socket that is listening, we then need to set-up the connection using address family. Going for AF_INET over PF_INET for the address family as AF identifier allow for all data structures. We use htons for byte order and INADDR_ANY so that it binds to all interfaces not just localhost.

```
//Configure settings for the server address.
sockaddr_in servAddr;
ZeroMemory(&servAddr, sizeof(servAddr));
servAddr.sin_family = AF_INET; //Address Family = Internet
servAddr.sin_addr.S_un.S_addr = INADDR_ANY; //Bind all interfaces
servAddr.sin_port = htons(54000); //Set port number (htons for byte order)
```

With the settings for our address family set up, we need to now bind the socket we created (listening) to the server address shown above.

```
//Bind the socket
int bindStatus = bind(listening, (sockaddr*)&servAddr, sizeof(servAddr));
```

Since the socket is set up with an address bound to it, we need to tell Winsock that this socket called listening is used to listen. Using SOMAXCONN or SOcket MAX CONNecTions it will allow of a maximum number of socket connections to be made.

```
listen(listening, SOMAXCONN); //SOMAXCONN to allow for more
```

We now ensure that we create a file descriptor(fd_set), Clear the set (FD_ZERO) and lastly add to set(FD_SET).

```
//Zero the set & add to the master set
fd_set masterSet;
FD_ZERO(&masterSet);

FD_SET(listening, &masterSet);
```

Now making a new file descriptor called copySet and make it equal to our masterSet we made previously.

```
fd_set copySet = masterSet;
```

We then need to accept if a client is listening, so we make a new socket for clients.

```
// Here will accept a new connection
SOCKET client = accept(listening, nullptr, nullptr);
```

We need to add the client socket to the master set by using FD_SET again.

```
// Add to master file set
FD_SET(client, &masterSet);
```

Using a buffer, we can send the clients messages across to each other.

```
char buffer[256];
ZeroMemory(buffer, sizeof(buffer));

int bytesRecived = recv(sock, buffer, sizeof(buffer), 0);
```

In order to know the difference between our clients we use their socket number as an id/name.

```
ss << "Client's Socket #" << sock << ": "
    << buffer << "\r\n";
string stringOut = ss.str();
```

Lastly, we use the send() function to send all the information such as the client socket # & the client's message.

```
if (outSocket != listening && outSocket != sock)
{
    // Output socket # to show different clients
    ostream ss;

    ss << "Client's Socket #" << sock << ": "
        << buffer << "\r\n";
    string stringOut = ss.str();

    send(outSocket, stringOut.c_str(), stringOut.size() + 1,
}
```

Now, we can test by running the server to see if it is successful.

```
WSAStartup() was successful!
Socket creation was successful!
Socket Bind was successful!
```

Creating the Client

Like the server, steps required section is inside of the client.cpp file.

The steps are as followed:

1. Initialise Winsock
2. Create socket with address & keep track of socket descriptor.
3. Fill in a hint structure/Server address family.
4. Connect to server
5. Loop to send and receive data.
6. Close Socket,
7. Clean-up Winsock.

This section will be shorter as steps one to three are the same as the server, so these stages will be left out.

Step 4 shows us connecting to the server using the address & port from the hint structure.

```
int connectionResult = connect(connectSocket, (sockaddr*)&servAddr, sizeof(servAddr));
```

We then use a do While loop. This will handle a string for userInput/response and a buffer to echo the client's response to all other clients.

```

char buffer[256]; //Buffer to echo client's response.
string userInput; //For the client's message.

do
{
    //Prompt user to enter a message
    getline(cin, userInput);
}

```

Do a check for if the userInput size is greater than zero or nothing and if so, we can then send client's response.

```

if (userInput.size() > 0) //check for user input
{
    //Send the text to server
    int sendMessage = send(connectSocket, userInput.c_str(), userInput.size() + 1, 0);

    if (sendMessage != SOCKET_ERROR)
    {
        //Wait for a response
        ZeroMemory(buffer, sizeof(buffer));
        int bytesReceived = recv(connectSocket, buffer, sizeof(buffer), 0);

        if (bytesReceived > 0)
        {
            //Echo response to console
            std::cout << "Server: " << string(buffer, 0, bytesReceived) << std::endl;
        }
    }
}

```

To finish off the client side of the programming we then close the connectSocket along with Winsock cleanup().

Testing

This segment will be going over the testing for the chat application. To showcase the features implemented. Using PuTTY [3] as a way to test multiple client(s) talking to each other was used.

Below shows the server running along with two client(s) connected and talking. The first client (picture on the left) is socket number 276 & the second user socket number is 300. These are used to determine the client's id.

The screenshot displays a Windows desktop environment. At the top, a window titled 'C:\Users\wackojacko357\source\repos\TCP server chat\Debug\TCP server chat.exe' shows the server's startup logs: 'WSAStartup() was successful!', 'Socket creation was successful!', and 'Socket Bind was successful!'. Below this, two PuTTY windows are open, both connected to '127.0.0.1'. The left PuTTY window, titled '127.0.0.1 - PuTTY', shows the server's output for client #276: 'Welcome to Jack's server!', 'Hello, world!', 'Client's Socket #276: Hello, user 300!', and 'oh, that was a test message!'. The right PuTTY window, also titled '127.0.0.1 - PuTTY', shows the server's output for client #300: 'Welcome to Jack's server!', 'Client's Socket #300: Hello, world!', 'Hello, user 300!', and 'Client's Socket #300: oh, that was a test message!'.

Next image is showing the server when the client's quit / leave from the server.

```
C:\Users\wackojacko357\source\repos\TCP s
WSAStartup() was successful!
Socket creation was successful!
Socket Bind was successful!

Client's Socket #300 has left!
Client's Socket #276 has left!
```

An option to have a list of emoji's from github [4] where the user can type was an idea thought of.

```
Enter Message: test
Server: ///////////////////////////////////
Welcome to Jack's server!
////////////////////////////////////

/Emoji
=====
Type ':smile: smily face!'
Type ':angry: angry face!'
Type ':laughing: laughing face!'
Type ':coffee: coffee!'
```

Attempts of adding the emojis were made but unsuccessful. Trying server side & client side adding emojis didn't work out. Future development would implement these.

Server.cpp

```
//string joinMessage = {
//"\n"
//"//////////////////////////////// \r\n"
//"Welcome to Jack's server! \r\n"
//"//////////////////////////////// \r\n"
//"For a drink type: :coffee:\r\n"
//"Laughing face type: :laughing:\r\n"
//"Smiley face type: :smile:\r\n"
//"Angry face type: :angry:\r\n"
//};
```

Client.cpp

```
if (userInput == "/Emoji")
{
    printf_s("===== \r\n");
    printf_s("Type ':smile: smily face!'\r\n");
    printf_s("Type ':angry: angry face!'\r\n");
    printf_s("Type ':laughing: laughing face!'\r\n");
    printf_s("Type ':coffee: coffee!'\r\n");
}
```

Conclusion

To conclude, using the correct protocol is important for the type of networked application that's intend on being designed and developed. Using one of them might have some advantages other another but both can be used simultaneously to develop an effect means of sending & receiving data. The chosen development of the IRC server

has the essential features required in order to handle multiple clients talking to one another. With nicknames in place to distinguish between each other.

References

- [1] "GeeksforGeeks," GeeksforGeeks, [Online]. Available: <https://www.geeksforgeeks.org/socket-programming-cc/>. [Accessed 10 April 2020].
- [2] "guru99," guru99, [Online]. Available: <https://www.guru99.com/tcp-vs-udp-understanding-the-difference.html>. [Accessed 10 April 2020].
- [3] S. Tatham, "Putty," Putty, 08 January 1999. [Online]. Available: <https://www.putty.org>. [Accessed 10 April 2020].
- [4] shalithasuranga, "Github," Github, [Online]. Available: <https://github.com/99xt/emojicpp>. [Accessed 10 April 2020].

Appendix

Server.cpp

```
////////////////////////////////////
//          TCP SOCKET CHAT APP          //
//    By Jack Partridge-1702918    //
////////////////////////////////////

////////////////////////////////////
//Steps Required (server)//
////////////////////////////////////
//1.  Init winsock
//2.  Create socket w/ address & keep track of sd.
//3.  Fill in a hint structure/Server address family.
//4.  Bind socket to ip address and port
//5.  Tell winsock the socket is for listening
//6.  While Loop: Wait for connection & broadcast message
//7.  Close socket
//8.  CleanUp winsock
////////////////////////////////////

        //Create multiple clients to a server
        //Use Select() command
        //With macros (File descriptors)
        //FD_CLR: remove from set
        //FD_SET: add to set
        //FD_ZERO: clear set
        //FD_ISSET: Incoming client

////////////////////////////////////
//User interface implementation
////////////////////////////////////
//Display user(s) / Client(s) names & or ip address
//Disconnect from the server (Close / QUIT)
//Send emoji(s) via switch statement list
////////////////////////////////////

////////////////////////////////////
//HEADERS
```

[illegible]

```

int bindStatus = bind(listening, (sockaddr*)&servAddr, sizeof(servAddr));

if (bindStatus == INVALID_SOCKET)
{
    std::cerr << "Unable to bind socket to local address! \r\n";
    exit(0);
}
else
    std::cout << "Socket Bind was successful! \r\n";

////////////////////////////////////
//5. Tell winsock the socket is for listening
////////////////////////////////////

listen(listening, SOMAXCONN); //SOMAXCONN to allow for more

//Zero the set & add to the master set
fd_set masterSet;
FD_ZERO(&masterSet);

FD_SET(listening, &masterSet);

//Clock setup for session time & ammount of data sent
auto start = std::chrono::system_clock::now();
int bytesRecived = 0;
int bytesWritten = 0;

////////////////////////////////////
//6. While Loop: Wait for connection & broadcast message
////////////////////////////////////

while (true)
{
    fd_set copySet = masterSet;

    int sockCount = select(0, &copySet, nullptr, nullptr, nullptr);

    for (int i = 0; i < sockCount; i++)
    {
        SOCKET sock = copySet.fd_array[i];

        // check if socket is listening. If so accept clients and send message
        if (sock == listening)
        {
            // Here will accept a new connection
            SOCKET client = accept(listening, nullptr, nullptr);

            //Check for invalid socket
            if (client == INVALID_SOCKET)
            {
                std::cerr << "Error accepting request from client!\r\n";
                exit(0);
            }
            else
            {
                // Create a join/welcome message to clients
                string joinMessage = {
                    ""
                    "//////////////////////////////// \r\n"
                    "Welcome to Jack's server! \r\n"
                    "//////////////////////////////// \r\n"
                };

                //string joinMessage = {
                //    "\n"
                //    "//////////////////////////////// \r\n"
            }

```

```

        // "Welcome to Jack's server! \r\n"
        // "//////////////////////////////////////// \r\n"
        // "For a drink type: :coffee:\r\n"
        // "Laughing face type: :laughing:\r\n"
        // "Smiley face type: :smile:\r\n"
        // "Angry face type: :angry:\r\n"
        // };

        printf_s("\r\n");

        // Send the join/Welcome message to clients
        send(client, joinMessage.c_str(), joinMessage.size() + 1, 0);
    }

    // Add to master file set
    FD_SET(client, &masterSet);

    if (sock <= 0)
    {
        auto end = std::chrono::system_clock::now();

        std::chrono::duration<double> elapsed_seconds = end - start;
        std::time_t end_time = std::chrono::system_clock::to_time_t(end);

        std::cout << "*****Session*****" << "\r\n";
        std::cout << "Bytes written: " << bytesWritten << " Bytes read: "
        << bytesReceived << "\r\n";
        std::cout << "Elapsed time: " << elapsed_seconds.count() << "
        secs" << "\r\n";

        std::cout << "*****" << "\r\n";
        std::cout << "Connection closed..." << "\r\n";
    }
}
else
{
    char buffer[256];
    ZeroMemory(buffer, sizeof(buffer));

    int bytesReceived = recv(sock, buffer, sizeof(buffer), 0);

    // Check if client sent bytes.
    if (bytesReceived <= 0)
    {
        closesocket(sock);
        FD_CLR(sock, &masterSet);
        cout << "Client's Socket #" << sock << " has left! \r\n";
    }
    else
    {
        for (int i = 0; i < masterSet.fd_count; i++)
        {
            SOCKET outSocket = masterSet.fd_array[i];

            if (outSocket != listening && outSocket != sock)
            {
                // Output socket # to show different clients
                ostringstream ss;

                ss << "Client's Socket #" << sock << ": "
                << buffer << "\r\n";
                string stringOut = ss.str();

                send(outSocket, stringOut.c_str(), stringOut.size()
                + 1, 0);
            }
        }
    }
}

```

```

    }
}

////////////////////////////////////
//7.   Close listening socket(s)
////////////////////////////////////

closesocket(listening);

////////////////////////////////////
//8.   CleanUp winsock
////////////////////////////////////

WSACleanup();

;
return 0;
}
////////////////////////////////////

```

Client.cpp

```

////////////////////////////////////
////      TCP SOCKET CHAT APP      ////
////   By Jack Partridge-1702918   ////
////////////////////////////////////

//Create multiple clients to a server
//Using the Select() command
//With macros (File descriptors)
//FD_CLR: remove from set
//FD_SET: add to set
//FD_ZERO: clear set
//FD_SET:

////////////////////////////////////
//Steps Required (Client)//
////////////////////////////////////
//1.   Init winsock
//2.   Create a socket
//3.   Fill in a hint structure
//4.   Connect to a server
//5.   D-While loop to send and receive data
//6.   Close socket
//7.   Clean up winsock
////////////////////////////////////

////////////////////////////////////
//HEADERS
#include <stdio.h>
#include <iostream>
#include <WS2tcpip.h>
#include <string>
#include "emoji.h"
////////////////////////////////////

////////////////////////////////////
#pragma comment (lib, "WS2_32.lib")
#pragma warning(disable : 4996) // _CRT_SECURE_NO_WARNINGS

using namespace std;
////////////////////////////////////

```

```

////////////////////////////////////
//Client Side
int main(int argc, char* argv[])
{
    string ipAddress = "127.0.0.1"; //IP Address of server
    int port = 54000; //Listening port no. of the server

    ///////////////////////////////////
    //1. Initialize winsock (Version 2.4)
    ///////////////////////////////////

    WSADATA winSockData;
    WORD version = MAKEWORD(2, 4);
    int wsResult = WSAStartup(version, &winSockData);

    if (wsResult != 0)
    {
        // Check if winsock failed.
        std::cerr << "Unable to init Winsock! Err # " << wsResult << std::endl;
        exit(0);
    }

    ///////////////////////////////////
    //2. Create a socket
    ///////////////////////////////////

    SOCKET connectSocket = socket(AF_INET, SOCK_STREAM, 0);

    // Check if couldn't create the socket
    if (connectSocket == INVALID_SOCKET)
    {
        std::cerr << "Unable to create socket! Err # "
                    << WSAGetLastError() << std::endl;
        WSACleanup();
        exit(0);
    }

    ///////////////////////////////////
    //3. Fill in a hint structure
    ///////////////////////////////////

    sockaddr_in servAddr; // Socket Address info
    ZeroMemory(&servAddr, sizeof(servAddr));
    servAddr.sin_family = AF_INET; // Address family Internet
    servAddr.sin_port = htons(port); // Port to connect
    inet_pton(AF_INET, ipAddress.c_str(), &servAddr.sin_addr); //Create socket

    ///////////////////////////////////
    //4. Connect to server
    ///////////////////////////////////

    int connectionResult = connect(connectSocket, (sockaddr*)&servAddr, sizeof(servAddr));

    // check if connection failed.
    if (connectionResult == SOCKET_ERROR)
    {
        std::cerr << "Unable to connect to server! Err # "
                    << WSAGetLastError() << std::endl;
        closesocket(connectSocket);
        WSACleanup();
        exit(0);
    }

    // Connect successful.
    printf_s("Type to join the server! \n");
    printf_s("===== \r\n");
    printf_s("Type '/Quit' or '/Leave' to disconnect from server \r\n");
    printf_s("Type /Emoji for Emoji List. \r\n");
    printf_s("===== \r\n");
}

```

```

printf_s("Enter Message: ");

////////////////////////////////////
//5.  D-While loop to send and receive data
////////////////////////////////////

char buffer[256]; //Buffer to echo client's response.
string userInput; //For the client's message.

do
{
    //Prompt user to enter a message
    getline(cin, userInput);

    if (userInput.size() > 0) //Check for user input
    {
        if (userInput == "/Quit" || userInput == "/Leave")
        {
            userInput.empty();
        }
        if (userInput == "/Emoji")
        {
            printf_s("===== \r\n");
            printf_s("Type ':smile: smily face!\r\n");
            printf_s("Type ':angry: angry face!\r\n");
            printf_s("Type ':laughing: laughing face!\r\n");
            printf_s("Type ':coffee: coffee!\r\n");
        }

        //Send the text to server
        int sendMessage = send(connectSocket, userInput.c_str(), userInput.size() + 1,
0);

        if (sendMessage != SOCKET_ERROR)
        {
            //Wait for a response
            ZeroMemory(buffer, sizeof(buffer));
            int bytesReceived = recv(connectSocket, buffer, sizeof(buffer), 0);

            if (bytesReceived > 0)
            {
                //Echo response to console
                std::cout << "Server: " << string(buffer, 0, bytesReceived) <<
"\r\n";
            }
        }
    }

} while (userInput.size() > 0);

////////////////////////////////////
//6.  Close everything
////////////////////////////////////

closesocket(connectSocket);
WSACleanup();

;

return 0;
}
////////////////////////////////////

```

Jack Partridge-1702917 NetworkAppAssign2

GRADEMARK REPORT

FINAL GRADE

62/100

GENERAL COMMENTS

Instructor

A good effort overall - I think the emojis would even have worked in an environment which supports them (i.e. non-console). A shame the server isn't multithreaded, this would've allowed you to implement some far more powerful functionality. Still, interesting to see you wrote your own low level code rather than use my wrapper functions - though nothing wrong with that!

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13