

Rhiannon Thomas
Network Multiplayer Games
Net Game Development
Lecturer: Mike Dacey

Table of Contents

Introduction	3
TCP	3
Implementation used in Visual Studio	3
UDP	3
Implementation used in Visual Studio	3
Multithreading	4
Testing	4
Test 1.....	4
Test 2.....	4
Conclusion	5
Evaluation	5
Bibliography	5
Appendix	6

Introduction

In this assignment a windows game will be created which will take the form of a client/server application. The game that has been chosen for this assignment is “Higher or Lower” which is a basic number guessing game between two players. Also, a number of networking principles, concepts and techniques have been demonstrated in this console application such as TCP and UDP.

TCP

TCP is short for Transmission Control Protocol which is a standard that defines how to establish and maintain a network conversation where application programs can exchange data [1]. TCP works with the IP, also known as the Internet Protocol, which defines how computers send packets of data to each other, which is a way of communicating between the server and the client. TCP is one of the most commonly used protocols in digital network communications [2]. TCP’s main job is to ensure end-to-end delivery of data between specific nodes and also the TCP transports and ensures that data is delivered to the correct destination [2].

Implementation used in Visual Studio

TCP has been used in the application as a Lobby for both players. In the client TCP, the users will be able to write a message and send it to the client. In this case, an option has been created where the user is able to click 1 to play the game and 2 to exit the lobby. The client will data which will run the game or exit the game.

In the server TCP, the users will receive the message from the client. The lobby page will be shown in the client whilst the server waits for a message from the client. When the player selects 1, the message will be received to the server and then outputs back to the client “Starting Game” to announce that the game is about to start. When the player selects 2, the message will be received to the server and then outputs back to the client “Exiting Lobby” to announce that user is exiting the lobby, the code for this is shown in the appendix below.

UDP

UDP is short for User Datagram Protocol which is an alternative communications protocol compared to the TCP used mostly for establishing low-latency and loss-tolerating connections between applications on the internet [3]. UDP enables process-to-process communication and sends messages called Datagrams which is considered the best way of communicating between nodes [3]. Also, UDP provides two services which are Port Numbers to help categorize different user requests and Checksum capability to verify that the data has arrived unbroken [3].

Implementation used in Visual Studio

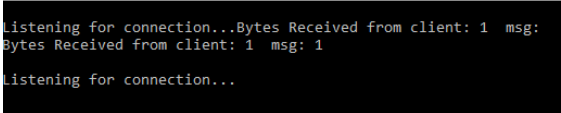
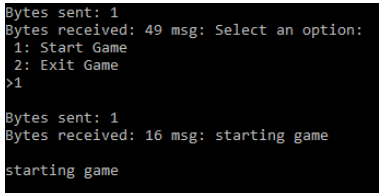
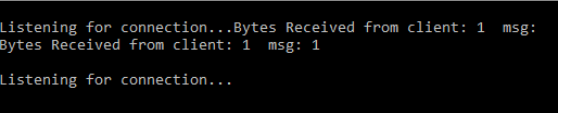
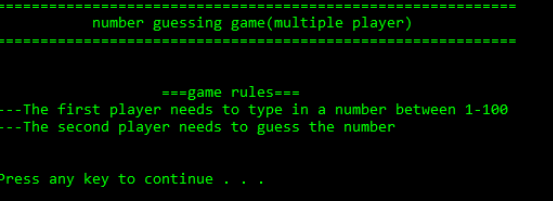
Parts of UDP has been used in the application. The main goal for the assignment is to be able to run the game in the UDP, unfortunately the game does not run on the UDP due to lack of time. However, the packets to communicate back and forth between the client and the server compiles correctly. A structure has been created called Character in both client and server so the client and server is able to send packets to each other. A class called “PackKey” has been implemented to be able to send the packet over to the client, then a class called “UnPackKey” has been implemented to be able to unpack the message from the server, the code for this is shown in the appendix below.

Multithreading

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and able to manage multiple requests by the same user without having to have multiple copies of the programming running in the computer [4]. Each user request in an application is kept track as a thread with a separate identity [4].

Multithreading has not been used in the application which has been implemented, however if multithreading was used each player would have been running on two separate threads and change some game logic so that player one and player two are both against an AI bot. The AI bot would randomly select a number and the two players would have to go against each other to guess the number the AI bot has chosen.

Testing

Test Number	Test Name	Expectation	Actual	Work? (Yes/No)	Screenshot
1	TCP	Player lobby should use a TCP server	Player lobby uses a TCP server	Yes	Server:  Client: 
2	UDP	Main game should run as a UDP server	Main game does not run as a UDP server	No	Server:  Client: 

Evaluation

In this assignment the networking techniques have been used and implemented onto a console application. This report explains what a TCP does and how it has been used in the console application which has been created and also with the UDP. For future improvements, more coding will be implemented so the game will run on the UDP and also create a multithread so that both players will play the game on separate threads.

References

- [1] M. Rouse, "TCP (Transmission Control Protocol)," [Online]. Available: <https://searchnetworking.techtarget.com/definition/TCP>.
- [2] "Transmission Control Protocol (TCP)," Techopedia, [Online]. Available: <https://www.techopedia.com/definition/5773/transmission-control-protocol-tcp>.
- [3] M. Rouse, "UDP (User Datagram Protocol)," Search Networking, [Online]. Available: <https://searchnetworking.techtarget.com/definition/UDP-User-Datagram-Protocol>.
- [4] M. Rouse, "multithreading," Tech Target, [Online]. Available: <https://whatis.techtarget.com/definition/multithreading>.

Bibliography

- [1] M. Rouse, "TCP (Transmission Control Protocol)," [Online]. Available: <https://searchnetworking.techtarget.com/definition/TCP>.
- [2] "Transmission Control Protocol (TCP)," Techopedia, [Online]. Available: <https://www.techopedia.com/definition/5773/transmission-control-protocol-tcp>.
- [3] M. Rouse, "UDP (User Datagram Protocol)," Search Networking, [Online]. Available: <https://searchnetworking.techtarget.com/definition/UDP-User-Datagram-Protocol>.
- [4] M. Rouse, "multithreading," Tech Target, [Online]. Available: <https://whatis.techtarget.com/definition/multithreading>.

Appendix

Server:

ClassGame.h

```

class ClassGame
{
    char restart = 'n';
    int game;
    int m1;
    char restart1 = 'n';
    int m2;
    int retrn;
    char restart2 = 'n';
    char restart3 = 'n';
    int hack;
    int v1;
    int n1;
    int n2;
    int n3;
    char restart4 = 'n';

public:
    ClassGame();
    ~ClassGame();
    void MultiPlayer();
};

```

ClassGame.cpp

```

#include "pch.h"
#include "ClassGame.h"

#include <iostream>
#include <string>
#include <cstdlib>
#include <time.h>
#include <windows.h>

using namespace std;

ClassGame::ClassGame()
{
}

ClassGame::~~ClassGame()
{
}

void ClassGame::MultiPlayer() {
    // Changes font colour to green
    system("color 0A");
    system("color 0A");

    int i = 1;
    restart = 'n';
}

```

```

system("cls");
cout << "===== " << endl;
cout << "          number guessing game(multiple player)" << endl;
cout << "=====\\n\\n" <<
endl;
cout << "          ===game rules===\\n---The first player needs to type
in a number bewten 1-100\\n---The second player needs to guess the number\\n\\n" << endl;
system("pause");

system("cls");
cout << "===== " << endl;
cout << "          number guessing game(multiple player)" << endl;
cout << "=====\\n\\n" <<
endl;
cout << "please enter the maximum of attempts: ";
cin >> v1;

do {
    system("cls");
    restart1 = 'n';
    cout << "===== "
<< endl;
    cout << "          number guessing game(multiple player)" << endl;
    cout <<
"=====\\n\\n" << endl;
    cout << "Player 1" << endl;
    cout << "-please enter a number between 0-100: ";
    cin >> n1;

    if (n1 > 100) {
        cout << "\\n\\n*****" <<
endl;
        cout << "ERROR\\nThe number is to big please re-enter" << endl;
        cout << "\\n\\n*****" <<
endl;
        system("pause");
        system("cls");
        restart1 = 'y';
    }

} while (restart1 == 'y');

system("cls");
cout << "===== " << endl;
cout << "          number guessing game(multiple player)" << endl;
cout << "=====\\n\\n" <<
endl;
cout << "player 2\\n" << endl;
cout << "type in the number you want to guess,type enter to start\\n" << endl;
cout << "          =====GOOD LUCK===== " << endl;
system("pause");

system("cls");
cout << "===== " << endl;
cout << "          number guessing game(multiple player)" << endl;
cout << "=====\\n\\n" <<
endl;

cout << "          ===gamestart===\\n\\n" << endl;
do {
    restart2 = 'n';

```


ClassTcpServer.cpp

```

#include "pch.h"
#include "ClassTCPserver.h"
#define _CRT_SECURE_NO_WARNINGS 1

#include <iostream>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>

using namespace std;

#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLen 512

// Initialize winsock
WSADATA wsaData;
char recvbuf[DEFAULT_BUFLen];
int iResult;
int recvbuflen = DEFAULT_BUFLen;
int iRcvdBytes = 1000;

struct addrinfo *addressList = NULL;
struct addrinfo *ptr = NULL;
struct addrinfo hints;

// Create a socket
SOCKET ListenSocket = INVALID_SOCKET;
SOCKET ClientSocket = INVALID_SOCKET;

ClassTCPserver::ClassTCPserver()
{
}

ClassTCPserver::~~ClassTCPserver()
{
}

// Sends message to the client
void ClassTCPserver::sendMessage(char* message)
{
    iResult = send(ClientSocket, message, (int)strlen(message) + 1, 0);
    shutdown(ClientSocket, SD_SEND); // Forces message to send
}

// Message recieved from client
char* ClassTCPserver::messageRecieved()
{
    //int iRcvdBytes = 1000;

    ClientSocket = accept(ListenSocket, NULL, NULL);
    iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);

    if (iResult > 0)
    {
        printf("Bytes Received from client: %d msg: %s \n", iResult, recvbuf);
        //iRcvdBytes = iResult;
    }
}

```

```

    }
    else if (iResult == 0)
    {
        printf("Client closing connection \n");
    }
    else
    {
        printf("recv failed %d \n", WSAGetLastError());
        WSACleanup();
    }
    return recvbuf;

    // Echo it back
    //char *sendbuf = recvbuf;
}

// Lobby
void ClassTCPserver::serverMain()
{
    char buff[256];
    char option;

    WSAStartup(MAKEWORD(2, 2), &wsaData);
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE; // Passive opens
    getaddrinfo(NULL, DEFAULT_PORT, &hints, &addressList);

    // Create a socket for connectong to the server
    ListenSocket = socket(addressList->ai_family, addressList->ai_socktype,
addressList->ai_protocol);
    BOOL optionValue = TRUE;
    int optionLength = sizeof(BOOL);
    setsockopt(ListenSocket, SOL_SOCKET, SO_REUSEADDR, (char*)&optionValue,
optionLength);

    // Set socket to listen on the port number provided (DEFAULT_PORT)
    bind(ListenSocket, addressList->ai_addr, (int)addressList->ai_addrlen);

    freeaddrinfo(addressList);

    listen(ListenSocket, SOMAXCONN);

    // Resolve the server address and port

    while (1) // Loop until key pressed
    {
        // Waits for a message from the client
        printf("\r\nListening for connection...");
        messageRecieved();

        // Outputs on screen the options
        strcpy(buff, "Select an option: \n 1: Start Game \n 2: Exit Game");
        sendMessage(buff);

        // convert to integer
        strcpy(buff, messageRecieved());
    }
}

```

```

        option = atoi(buff);

        if (option == 1)
        {
            // if 1 is pressed then send this message to the client
            strcpy(buff, "starting game \n");
            sendMessage(buff);
        }
        else if (option == 2)
        {
            // if 2 is pressed then send this message to the client
            strcpy(buff, "exiting lobby \n");
            sendMessage(buff);
        }
    }

    closesocket(ListenSocket);

    printf("\r\nHit key to exit");
    while (!_kbhit());

    WSACleanup();
}

```

Messages.h

```

#pragma once

typedef struct MESSAGE_PREAMBLE
{
    unsigned short code;           //type of message
    unsigned short length;        //Length of message
};

#define PRAMBLE_LENGTH 4          //4 bytes (2 x unsigned short)

//code values
#define MESSAGE_CODE_BULLET_FIRED 1
#define MESSAGE_CODE_RUBBISH      2

typedef struct MESSAGE_BULLET_FIRED
{
    int startX;
    int startY;
    int vectorX;
    int vectorY;
};

typedef struct MESSAGE_RUBBISH
{
    unsigned char username[20];
    unsigned long longvalue;
    unsigned short shortvalue;
    unsigned char bytevalue;
    int intValue;

    //some variable length data
}

```

```

        unsigned short dataLength;
        unsigned char data[100];           //You dont really need this
};

struct Character
{
    char Character;
};

```

Packing.h

```

#pragma once
#include <stdlib.h>

/*
MSG_STRUCT
An application may have many types of messages that need to be
sent/received. A good idea is to start each message
with a code to indicate the message structure being sent and
a length field containing the number of bytes in the message
structure.
*/
/*
Note: The packing of structures is dependant on the compiler,
fields may be aligned on word or long word boundaries.
The 'packing bytes' have to be removed by the packing process.
*/
typedef struct
{
    unsigned short code;
    unsigned short length;
    unsigned char username[20];
    unsigned long longValue;           // 32 bit
    unsigned short shortValue;        // 16 bit
    unsigned char byteValue;          // 8 bit
    int intValue;                     // 32 bit (signed)
    unsigned short datalen;
    unsigned char data[100];
} MSG_STRUCT;

// Macros for packing and unpacking the structure
// to a buffer.

// PACKING

#define PackBytes(bf, bytes, len) \
    memcpy(bf, bytes, len); \
    bf += len;

// PackLong is equivalent to htonl but writes the
// u_long into a buffer.
// Note: You could do the equivalent
// #define PackLong(bf, l) \
//     *((unsigned long *)bf) = htonl(l); \
//     bf += sizeof(unsigned long);
// However, this would not be portable, some processors
// would crash if bf was not word/long word aligned
// when trying to use it as a unsigned long.
// Note: the terms long and short in htonl, htons, etc,
// are historical artifacts from the Digital VAX BSD distribution.
// In this context, long = 32 bits and short = 16 bits

```

```

//
#define PackLong(bf, l) \
    *bf++ = (l >> 24) & 0xFF; \
    *bf++ = (l >> 16) & 0xFF; \
    *bf++ = (l >> 8) & 0xFF; \
    *bf++ = l & 0xFF;

#define PackShort(bf, s) \
    *bf++ = (s >> 8) & 0xFF; \
    *bf++ = s & 0xFF;

#define PackByte(bf, b) \
    *bf++ = b;

// UNPACKING

#define UnpackBytes(bf, bytes, len) \
    memcpy(bytes, bf, len); \
    bf += len;

// UnpackLong is equivalent to ntohl but reads the
// buffer into a u_long.
// Note: You could do the equivalent
// #define UnpackLong(bf, l) \
//     l = ntohl(*(unsigned long *)bf); \
//     bf += sizeof(unsigned long);
// However, this would not be portable, some processors
// would crash if bf was not word/long word aligned
// when trying to use it as a unsigned long.
//
#define UnpackLong(bf, l) \
    l = (*bf++ << 24); \
    l += (*bf++ << 16); \
    l += (*bf++ << 8); \
    l += *bf++;

#define UnpackShort(bf, s) \
    s = (*bf++ << 8); \
    s += *bf++;

#define UnpackByte(bf, b) \
    b = *bf++;

// Could use a code to distinguish between different
// types of data being sent e.g. bullet = 1, player = 2,
// new level = 3, etc.
#define MSGCODE 0x0001

// Function to pack the message structure into a
// provided buffer
unsigned short PackMsg(MSG_STRUCT *msg, unsigned char *buff)
{
    unsigned short msglen;
    unsigned char *lenptr;
    // unsigned char *startptr = buff;

    PackShort(buff, msg->code);
    lenptr = buff; // Add total length later
    buff += sizeof(unsigned short);
    PackBytes(buff, msg->username, 20);
    PackLong(buff, msg->longValue);
    PackShort(buff, msg->shortValue);
}

```

```

    PackByte(buff, msg->byteValue);
    PackLong(buff, msg->intValue);
    PackShort(buff, msg->datalen);
    PackBytes(buff, msg->data, msg->datalen);

    // Calculate the correct message length.
    // =====
    // You may hard code the length of the header
    // and add the extra data bytes (msg->datalen)
    // at the end.
    // 37 + msg->datalen
    // This is quite often the technique used,
    // but since we are incrementing the buffer
    // pointer each time we add a byte we can
    // simply calculate the length from the pointers.
    // Because they are unsigned char* pointers, subtracting
    // them gives us the difference in the addresses in bytes.
    //msglen = (unsigned short)(buff-startptr);

    // Here we add all the size of all the fields together
    // to make it clear what we want.
    msglen = sizeof(unsigned short)
        + sizeof(unsigned short)
        + sizeof(msg->username)
        + sizeof(unsigned long)
        + sizeof(unsigned short)
        + 1
        + sizeof(unsigned long) // int
        + sizeof(unsigned short)
        + msg->datalen;

    PackShort(lenptr, msglen);

    return msglen;
}

// Unpack the provided buffer into the provided
// message structure
void UnpackMsg(MSG_STRUCT *msg, unsigned char *buff)
{
    UnpackShort(buff, msg->code);
    UnpackShort(buff, msg->length);
    UnpackBytes(buff, msg->username, 20);
    UnpackLong(buff, msg->longValue);
    UnpackShort(buff, msg->shortValue);
    UnpackByte(buff, msg->byteValue);
    UnpackLong(buff, msg->intValue);
    UnpackShort(buff, msg->datalen);
    UnpackBytes(buff, msg->data, msg->datalen);
}

```

PackUnpack.h

```
#pragma once
```

```

#define PackBytes(bf, bytes, len) \
    memcpy(bf, bytes, len); \
    bf += len;

#define PackLong(bf, l) \
    *bf++ = (l >> 24) & 0xFF; \

```

```

    *bf++ = (l >> 16) & 0xFF; \
    *bf++ = (l >> 8) & 0xFF; \
    *bf++ = l & 0xFF;

#define PackShort(bf, s) \
    *bf++ = (s >> 8) & 0xFF; \
    *bf++ = s & 0xFF;

#define PackByte(bf, b) \
    *bf++ = b;

#define UnpackBytes(bf, bytes, len) \
    memcpy(bytes, bf, len); \
    bf += len;

#define UnpackLong(bf, l) \
    l = (*bf++ << 24); \
    l += (*bf++ << 16); \
    l += (*bf++ << 8); \
    l += *bf++;

#define UnpackShort(bf, s) \
    s = (*bf++ << 8); \
    s += *bf++;

#define UnpackByte(bf, b) \
    b = *bf++;

```

Source.cpp

```

#include "pch.h"
#include <stdio.h>
#include "ClassTCPserver.h"

ClassTCPserver server;

int main(int argc, char** argv)
{
    server.serverMain();
    return 0;
}

```

ClassUdp.h

```

#pragma once
#include "Messages.h"
#include "PackUnpack.h"
class classUdp
{
    char restart = 'n';
    int game;
    int m1;
    char restart1 = 'n';
    int m2;
    int retrn;
    char restart2 = 'n';
    char restart3 = 'n';
    int hack;
    int v1;
    int n1;
}

```

```

    int n2;
    int n3;
    char restart4 = 'n';

public:
    classUdp();
    ~classUdp();
    unsigned short packKey(Character *msg, unsigned char *buff);
    void MultiPlayer();
    int mainServerUDP();
};

```

ClassUdp.cpp

```

#include "pch.h"
#include "classUdp.h"

#include <iostream>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <string>
#include <cstdlib>
#include <time.h>
#include <windows.h>

using namespace std;

#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLen 512

classUdp::classUdp()
{
}

classUdp::~classUdp()
{
}

unsigned short classUdp::packKey(Character *msg, unsigned char *buff) {
    unsigned short msglen;
    unsigned char *lenptr;

    PackByte(buff, msg->Character);

    msglen = sizeof(msg->Character);

    return msglen;
}

// Calculates the data
void classUdp::MultiPlayer() {
    /*int i = 1;

    do {
        system("cls");

```

```

        restart1 = 'n';
        cout << "=====
<< endl;
        cout << "          number guessing game(multiple player)" << endl;
        cout <<
"=====\\n\\n" << endl;
        cout << "Player 1" << endl;
        cout << "-please enter a number betwen 0-100: ";
        cin >> n1;

        if (n1 > 100) {
            cout << "\\n\\n*****" <<
endl;
            cout << "ERROR\\nThe number is to big please re-enter" << endl;
            cout << "\\n\\n*****" <<
endl;
            system("pause");
            system("cls");
            restart1 = 'y';
        }

    } while (restart1 == 'y');

    system("cls");
    cout << "=====
    cout << "          number guessing game(multiple player)" << endl;
    cout << "=====\\n\\n" <<
endl;
    cout << "player 2\\n" << endl;
    cout << "type in the number you want to guess,type enter to start\\n" << endl;
    cout << "          =====GOOD LUCK=====
    cout << "          =====\\n\\n" <<
endl;

    cout << "          ===gamestart===\\n\\n" << endl;
    do {
        restart2 = 'n';
        cout << i;
        cout << ": ";
        cin >> n2;
        if (i == v1) {
            cout << "\\nsorry, you are out of attempts\\n\\n\\n" << endl;
            system("pause");
        }

        else if (n2 == n1) {
            cout << "You win!" << endl;
        }
        else if (n2 < n1) {
            cout << "bigger" << endl;
            i++;
            restart2 = 'y';
        }
        else if (n2 > n1) {
            cout << "smaller" << endl;
            i++;

```



```

        printf("socket failed: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    rx_addr.sin_family = AF_INET;
    rx_addr.sin_port = htons((u_short)atol(DEFAULT_PORT));
    rx_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    iResult = bind(UdpSocket, (SOCKADDR *)&rx_addr, sizeof(rx_addr));
    if (iResult == SOCKET_ERROR)
    {
        printf("bind failed: %d\n", WSAGetLastError());
        closesocket(UdpSocket);
        WSACleanup();
        return 1;
    }

    iResult = recvfrom(UdpSocket,
        recvbuf,
        recvbuflen,
        0,
        (SOCKADDR *)&sender_addr,
        &iSenderAddrSize);

    if (iResult == SOCKET_ERROR)
    {
        printf("recv failed: %d\n", WSAGetLastError());
        closesocket(UdpSocket);
        WSACleanup();
        return 1;
    }

    printf("Bytes received: %d Msg=%s\n", iResult, recvbuf);

    iResult = sendto(UdpSocket,
        recvbuf,
        iResult,
        0,                                     // flags - we never need
        them so zero them                      (SOCKADDR *)&sender_addr,
        iSenderAddrSize);

    if (iResult == SOCKET_ERROR)
    {
        printf("send failed: %d\n", WSAGetLastError());
        closesocket(UdpSocket);
        WSACleanup();
        return 1;
    }

    printf("Bytes Sent: %ld\n", iResult);

    // cleanup
    closesocket(UdpSocket);

    while (!_kbhit());

```

```

        WSACleanup();

        return 0;
}

```

psh.h

```

#ifndef PCH_H
#define PCH_H

#endif //PCH_H

```

psh.cpp

```

#include "pch.h"

```

Client:

ClassGame.h

```

#pragma once
class ClassGame
{
    char restart = 'n';
    int game;
    int m1;
    char restart1 = 'n';
    int m2;
    int retrn;
    char restart2 = 'n';
    char restart3 = 'n';
    int hack;
    int v1;
    int n1;
    int n2;
    int n3;
    char restart4 = 'n';

public:
    ClassGame();
    ~ClassGame();
    void MultiPlayer();
};

```

ClassTCPclient.h

```

#pragma once
class ClassTCPclient
{
    void sendMessage(char* message);
    char* messageRecieved();

public:
    ClassTCPclient();
    ~ClassTCPclient();
    void clientMain();
}

```

```
};
```

classUdp.h

```
#pragma once
#include "Messages.h"
#include "PackUnpack.h"
class classUdp
{
    char restart = 'n';
    int game;
    int m1;
    char restart1 = 'n';
    int m2;
    int retrn;
    char restart2 = 'n';
    char restart3 = 'n';
    int hack;
    int v1;
    int n1;
    int n2;
    int n3;
    char restart4 = 'n';

public:
    classUdp();
    ~classUdp();
    void MultiPlayer();
    unsigned short packKey(Character * msg, unsigned char * buff);
    void unPackKey(Character *msg, unsigned char* buff);
    int mainClientUDP();
};
```

Messages.h

```
#pragma once

typedef struct MESSAGE_PREAMBLE
{
    unsigned short code;           //type of message
    unsigned short length;         //Length of message
};

#define PRAMBLE_LENGTH 4          //4 bytes (2 x unsigned short)

//code values
#define MESSAGE_CODE_BULLET_FIRED 1
#define MESSAGE_CODE_RUBBISH      2

typedef struct MESSAGE_BULLET_FIRED
{
    int startX;
    int startY;
    int vectorX;
    int vectorY;
};

typedef struct MESSAGE_RUBBISH
{
    unsigned char username[20];
```

```

    unsigned long longvalue;
    unsigned short shortvalue;
    unsigned char bytevalue;
    int intValue;

    //some variable length data
    unsigned short dataLength;
    unsigned char data[100];           //You dont really need this
};

struct Character
{
    char Character;
};

```

Packing.h

```

#pragma once
#include <stdlib.h>

/*
MSG_STRUCT
An application may have many types of messages that need to be
sent/received. A good idea is to start each message
with a code to indicate the message structure being sent and
a length field containing the number of bytes in the message
structure.
*/
/*
Note: The packing of structures is dependant on the compiler,
fields may be aligned on word or long word boundaries.
The 'packing bytes' have to be removed by the packing process.
*/
typedef struct
{
    unsigned short code;
    unsigned short length;
    unsigned char username[20];
    unsigned long longValue;           // 32 bit
    unsigned short shortValue;        // 16 bit
    unsigned char byteValue;          // 8 bit
    int intValue;                     // 32 bit (signed)
    unsigned short datalen;
    unsigned char data[100];
} MSG_STRUCT;

// Macros for packing and unpacking the structure
// to a buffer.

// PACKING

#define PackBytes(bf, bytes, len) \
    memcpy(bf, bytes, len); \
    bf += len;

// PackLong is equivalent to htonl but writes the
// u_long into a buffer.
// Note: You could do the equivalent
// #define PackLong(bf, l) \
//     *((unsigned long *)bf) = htonl(l); \
//     bf += sizeof(unsigned long);

```

```

// However, this would not be portable, some processors
// would crash if bf was not word/long word aligned
// when trying to use it as a unsigned long.
// Note: the terms long and short in htonl, htons, etc,
// are historical artifacts from the Digital VAX BSD distribution.
// In this context, long = 32 bits and short = 16 bits
//
#define PackLong(bf, l) \
    *bf++ = (l >> 24) & 0xFF; \
    *bf++ = (l >> 16) & 0xFF; \
    *bf++ = (l >> 8) & 0xFF; \
    *bf++ = l & 0xFF;

#define PackShort(bf, s) \
    *bf++ = (s >> 8) & 0xFF; \
    *bf++ = s & 0xFF;

#define PackByte(bf, b) \
    *bf++ = b;

// UNPACKING

#define UnpackBytes(bf, bytes, len) \
    memcpy(bytes, bf, len); \
    bf += len;

// UnpackLong is equivalent to ntohl but reads the
// buffer into a u_long.
// Note: You could do the equivalent
// #define UnpackLong(bf, l) \
//     l = ntohl(*(unsigned long *)bf); \
//     bf += sizeof(unsigned long);
// However, this would not be portable, some processors
// would crash if bf was not word/long word aligned
// when trying to use it as a unsigned long.
//
#define UnpackLong(bf, l) \
    l = (*bf++ << 24); \
    l += (*bf++ << 16); \
    l += (*bf++ << 8); \
    l += *bf++;

#define UnpackShort(bf, s) \
    s = (*bf++ << 8); \
    s += *bf++;

#define UnpackByte(bf, b) \
    b = *bf++;

// Could use a code to distinguish between different
// types of data being sent e.g. bullet = 1, player = 2,
// new level = 3, etc.
#define MSGCODE 0x0001

// Function to pack the message structure into a
// provided buffer
unsigned short PackMsg(MSG_STRUCT *msg, unsigned char *buff)
{
    unsigned short msglen;
    unsigned char *lenptr;
    // unsigned char *startptr = buff;

```

```

PackShort(buff, msg->code);
lenptr = buff;      // Add total length later
buff += sizeof(unsigned short);
PackBytes(buff, msg->username, 20);
PackLong(buff, msg->longValue);
PackShort(buff, msg->shortValue);
PackByte(buff, msg->byteValue);
PackLong(buff, msg->intValue);
PackShort(buff, msg->datalen);
PackBytes(buff, msg->data, msg->datalen);

// Calculate the correct message length.
// =====
// You may hard code the length of the header
// and add the extra data bytes (msg->datalen)
// at the end.
// 37 + msg->datalen
// This is quite often the technique used,
// but since we are incrementing the buffer
// pointer each time we add a byte we can
// simply calculate the length from the pointers.
// Because they are unsigned char* pointers, subtracting
// them gives us the difference in the addresses in bytes.
//msglen = (unsigned short)(buff-startptr);

// Here we add all the size of all the fields together
// to make it clear what we want.
msglen = sizeof(unsigned short)
        + sizeof(unsigned short)
        + sizeof(msg->username)
        + sizeof(unsigned long)
        + sizeof(unsigned short)
        + 1
        + sizeof(unsigned long) // int
        + sizeof(unsigned short)
        + msg->datalen;

PackShort(lenptr, msglen);

return msglen;
}

// Unpack the provided buffer into the provided
// message structure
void UnpackMsg(MSG_STRUCT *msg, unsigned char *buff)
{
    UnpackShort(buff, msg->code);
    UnpackShort(buff, msg->length);
    UnpackBytes(buff, msg->username, 20);
    UnpackLong(buff, msg->longValue);
    UnpackShort(buff, msg->shortValue);
    UnpackByte(buff, msg->byteValue);
    UnpackLong(buff, msg->intValue);
    UnpackShort(buff, msg->datalen);
    UnpackBytes(buff, msg->data, msg->datalen);
}

```

PackUnpack.h

```

#pragma once

```

```

#define PackBytes(bf, bytes, len) \
memcpy(bf, bytes, len); \
bf += len;

#define PackLong(bf, l) \
    *bf++ = (l >> 24) & 0xFF; \
    *bf++ = (l >> 16) & 0xFF; \
    *bf++ = (l >> 8) & 0xFF; \
    *bf++ = l & 0xFF;

#define PackShort(bf, s) \
    *bf++ = (s >> 8) & 0xFF; \
    *bf++ = s & 0xFF;

#define PackByte(bf, b) \
    *bf++ = b;

#define UnpackBytes(bf, bytes, len) \
    memcpy(bytes, bf, len); \
    bf += len;

#define UnpackLong(bf, l) \
    l = (*bf++ << 24); \
    l += (*bf++ << 16); \
    l += (*bf++ << 8); \
    l += *bf++;

#define UnpackShort(bf, s) \
    s = (*bf++ << 8); \
    s += *bf++;

#define UnpackByte(bf, b) \
    b = *bf++;

```

Pch.h

```

#ifndef PCH_H
#define PCH_H

#endif //PCH_H

```

ClassGame.cpp

```

#include "pch.h"
#include "ClassGame.h"

#include <iostream>
#include <string>
#include <cstdlib>
#include <time.h>
#include <windows.h>

using namespace std;

ClassGame::ClassGame()
{
}

```

```

ClassGame::~~ClassGame()
{
}

void ClassGame:: MultiPlayer() {
    system("color 0A");
    int i = 1;
    system("color 0A");
    restart = 'n';
    system("cls");
    cout << "===== " << endl;
    cout << "          number guessing game(multiple player)" << endl;
    cout << "=====\\n\\n" <<
endl;
    cout << "          ===game rules===\\n---The first player needs to type
in a number between 1-100\\n---The second player needs to guess the number\\n\\n" <<
endl;
    system("pause");

    system("cls");
    cout << "===== " << endl;
    cout << "          number guessing game(multiple player)" << endl;
    cout << "=====\\n\\n" <<
endl;
    cout << "please enter the maximum of attempts: ";
    cin >> v1;

    do {
        system("cls");
        restart1 = 'n';
        cout << "===== "
<< endl;
        cout << "          number guessing game(multiple player)" << endl;
        cout <<
"=====\\n\\n" << endl;
        cout << "Player 1" << endl;
        cout << "-please enter a number between 0-100: ";
        cin >> n1;

        if (n1 > 100) {
            cout << "\\n\\n*****" <<
endl;
            cout << "ERROR\\nThe number is to big please re-enter" << endl;
            cout << "\\n\\n*****" <<
endl;
            system("pause");
            system("cls");
            restart1 = 'y';
        }

    } while (restart1 == 'y');

    system("cls");
    cout << "===== " << endl;
    cout << "          number guessing game(multiple player)" << endl;
    cout << "=====\\n\\n" <<
endl;
    cout << "player 2\\n" << endl;
    cout << "type in the number you want to guess,type enter to start\\n" << endl;
    cout << "          ===== GOOD LUCK =====" << endl;
}

```



```

#define _CRT_SECURE_NO_WARNINGS 1

#include <iostream>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include "ClassTCPClient.h"
#include "ClassGame.h"
using namespace std;

#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLen 512

WSADATA wsaData;
char recvbuf[DEFAULT_BUFLen];
int iResult;
int recvbuflen = DEFAULT_BUFLen;

struct addrinfo *addressList = NULL;
struct addrinfo *ptr = NULL;
struct addrinfo hints;
int iRcvdBytes;

const char *serverName;
SOCKET connectSocket = INVALID_SOCKET;
char sendbuf[1000];
ClassGame g;

ClassTCPClient::ClassTCPClient()
{
}

ClassTCPClient::~~ClassTCPClient()
{
}

// Sends a message to the server
void ClassTCPClient::sendMessage(char* message)
{
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    getaddrinfo(serverName, DEFAULT_PORT, &hints, &addressList);

    connectSocket = socket(addressList->ai_family, addressList->ai_socktype,
addressList->ai_protocol);

    //Active open
    connect(connectSocket, addressList->ai_addr, (int)addressList->ai_addrlen);

    iResult = send(connectSocket, message, (int)strlen(message), 0);

    freeaddrinfo(addressList);

    printf("Bytes sent: %ld \n", iResult);
}

```

```
}
```

```
char* ClassTCPclient::messageRecieved()
{
    //strcpy(recvbuf, "test");
    // Block waiting to receive response from server
    iResult = recv(connectSocket, recvbuf, recvbuflen, 0);

    // YOU must handle the streaming of the bytes being received!! e.g create a
    // buffer to store the until you get enough to unpack and use

    if (iResult > 0)
    {
        // iResult is the number of bytes received. Assumed NUL char is included
        // in recvbuf to terminate the string
        printf("Bytes received: %d msg: %s \n", iResult, recvbuf);
        iRcvdBytes = iResult;
    }
    else if (iResult == 0)
    {
        printf("Connection closed by server");
    }
    else
    {
        printf("recv failed: %d \n", WSAGetLastError());
    }
    return recvbuf;
}
```

```
// Lobby
void ClassTCPclient::clientMain()
{
    char buff[256];
    char message[256];
    serverName = "127.0.0.1";

    //strcpy(message, "hello");
    //printf(">");
    //scanf("%s", &buff); //message to send and receive

    WSASStartup(MAKEWORD(2, 2), &wsaData);
    char rty[1000];
    sprintf(rty, "%s %i", "hello", clock());
    strcpy(message, " ");
    sendMessage(message);
    strcpy(recvbuf, messageRecieved());
    while (1)
    {
        // Type in number
        printf(">");
        scanf("%s", &buff); //message to send and receive
        printf("\n");
        sendMessage((char*)buff);

        //messageRecieved();

        strcpy(recvbuf, messageRecieved());
    }
}
```

```

        printf("%s", recvbuf);

        // outpu sendbuf
        cin >> sendbuf;
        sendMessage(sendbuf);

        strcpy(buff, messageRecieved());

        printf("%s", buff);

        if (strcmp(sendbuf, "1") == 0)
        {
            // if 1 is selected then run game
            g.MultiPlayer();
        }
        else if (strcmp(sendbuf, "2") == 0)
        {
            // if 2 is selected then exit game
            exit(0);
        }
    }

    closesocket(connectSocket);

    printf("\r\nHit key to exit");
    while (!_kbhit());
    shutdown(connectSocket, SD_SEND);
    WSACleanup();
}

```

classUdp.cpp

```

#include "pch.h"
#include "ClassGame.h"
#include "classUdp.h"

#include <iostream>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <string>
#include <cstdlib>
#include <time.h>
#include <windows.h>

using namespace std;

#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLen 512

ClassGame guess;

classUdp::classUdp()
{
}

classUdp::~classUdp()

```

```

{
}

// Displays the data
void classUdp::MultiPlayer()
{
    /*system("color 0A");
    int i = 1;
    system("color 0A");
    restart = 'n';
    system("cls");
    cout << "=====
<< endl;
    cout << "          number guessing game(multiple player)" << endl;
    cout <<
"=====\\n\\n" << endl;
    cout << "          ===game rules===\\n---The first player needs
to type in a number bewten 1-100\\n---The second player needs to guess the number\\n\\n"
<< endl;
    system("pause");

    system("cls");
    cout << "=====
<< endl;
    cout << "          number guessing game(multiple player)" << endl;
    cout <<
"=====\\n\\n" << endl;
    cout << "please enter the maximum of attempts: ";
    cin >> v1;*/
}

// Client UDP
unsigned short classUdp::packKey(Character *msg, unsigned char *buff)
{
    unsigned short msglen;
    unsigned char *lenptr;

    PackByte(buff, msg->Character);

    msglen = sizeof(msg->Character);

    return msglen;
}

void classUdp::unPackKey(Character *msg, unsigned char* buff)
{
    UnpackByte(buff, msg->Character)
}

int classUdp::mainClientUDP()
{
    WSADATA    wsaData;
    char    recvbuf[DEFAULT_BUFLen];
    int    iResult;
    int    recvbuflen = DEFAULT_BUFLen;
    SOCKET UdpSocket = INVALID_SOCKET;

    struct addrinfo *addressList = NULL;
    struct addrinfo hints;

    iResult = WSASStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0)

```

```

{
    printf("WSAStartup failed: %d\n", iResult);
    return 1;
}

const char *sendbuf = "this is a test";

ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_protocol = IPPROTO_UDP;

iResult = getaddrinfo("127.0.0.1", DEFAULT_PORT, &hints, &addressList);
if (iResult != 0)
{
    printf("getaddrinfo failed: %d\n", iResult);
    WSACleanup();
    return 1;
}

if (addressList != NULL)
{
    UdpSocket = socket(addressList->ai_family,
                       addressList->ai_socktype,
                       addressList->ai_protocol);

    if (UdpSocket == INVALID_SOCKET)
    {
        printf("Error at socket(): %ld\n", WSAGetLastError());
        freeaddrinfo(addressList);
        WSACleanup();
        return 1;
    }

    iResult = sendto(UdpSocket,
                     sendbuf,
                     (int)strlen(sendbuf) + 1,
                     0,                                     // flags - we never need
them so zero them
                     addressList->ai_addr,
                     (int)addressList->ai_addrlen);

    if (iResult == SOCKET_ERROR)
    {
        printf("send failed: %d\n", WSAGetLastError());
        freeaddrinfo(addressList);
        closesocket(UdpSocket);
        WSACleanup();
        return 1;
    }
}

```

```

printf("Bytes Sent: %ld\n", iResult);

// For fun!!, lets find out the dynamic port number used.
//
sockaddr_in client_addr;
int iClientAddrLen = sizeof(client_addr);
iResult = getsockname(UdpSocket, (SOCKADDR *)&client_addr, &iClientAddrLen);
if (iResult == SOCKET_ERROR)
{
    printf("getsockname failed: %d\n", WSAGetLastError());
    freeaddrinfo(addressList);
    closesocket(UdpSocket);
    WSACleanup();
    return 1;
}

freeaddrinfo(addressList);

// Receive a datagram packet from the socket.
//
iResult = recvfrom(UdpSocket,
    recvbuf,
    recvbuflen,
    0, // flags we don't need, so set to zero
    NULL,
    NULL);

if (iResult == SOCKET_ERROR)
{
    printf("recv failed: %d\n", WSAGetLastError());
    closesocket(UdpSocket);
    WSACleanup();
    return 1;
}
printf("Bytes received: %d Msg=%s\n", iResult, recvbuf);

// cleanup
closesocket(UdpSocket);

while (!_kbhit());

WSACleanup();

return 0;
}

```

pch.cpp

```
#include "pch.h"
```

Source.cpp

```

#include "pch.h"
#include <stdio.h>
#include "ClassTCPClient.h"

```

```
ClassTCPClient client;
```

```
int main(int argc, char** argv)
{
    client.clientMain();
    return 0;
}
```

