

Network Multiplayer Games Report

by Evan Smith

FILE	586_EVAN_SMITH_NETWORK_MULTIPLAYER_GAMES_REPORT_178155_1996675596.DOCX (2.17M)		
TIME SUBMITTED	05-APR-2019 09:37PM (UTC+0100)	WORD COUNT	3359
SUBMISSION ID	104243611	CHARACTER COUNT	19732



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Network Multiplayer Games Assignment

By Evan Smith
Student Number: P146188
Lecturer: Mike Dacey

Contents

Table of Figures	3
1: Introduction	4
2: The Game	5
3: TCP, UDP and Multi-Threading	6
4: The Application	9
4.1: Planning the Application	9
4.2 Phase 1: TCP Connection	11
4.3 Phase 2: The UDP, The Game and The Multithread	11
4.4 Phase 3: Clean-up.....	11
5: Testing.....	12
5.1: TCP Testing.....	12
5.2: UDP Testing.....	13
5.3: Game	13
6: Conclusion.....	15
7: Personal Notes	16
8: References	17
8: Appendix	18
8.1 Server Side	18
8.1.1 MultiThread	18
8.1.2 Packing	20
8.1.3 TCPServer	22
8.1.4 TicTacToe	27
8.1.5 UDPServer	32

Table of Figures

Figure 1: TicTacToe	5
Figure 2: OSI and TCP/IP Reference Model.....	6
Figure 3 Diagram of UDP Datagram.....	7
Figure 4 System Sequence Diagram of the Application.....	9
Figure 5 Class Diagram of the application.....	10
Figure 6 Test Result of TCP	12
Figure 7 Simple of Diagram of Sending Game Messages.....	13
Figure 8 Test Result of Game and UDP	14

1: Introduction

The Network component in Video Games has been essential for decades, allowing players to compete with others around the world. This assignment requires the use of TCP/UDP servers as well as an element of Multi-Threading, one that will run the game known as Noughts and Crosses or in America, Tic Tac Toe. This report, details the practises of developing the game, as well as the network aspect of it, including the issues that were faced during the process and the solutions that were used to address these.

2: The Game

The game is called Noughts and Crosses, a simple, straight forward game, requiring two players to compete against each to get three of either X or O in a row to win. The justification behind the game is that it is easy and simple to implement, therefore a lot of development time can be allocated for the network aspect of this assignment. During development, there was a choice as to how the user would play the game. It was between using the move keys to move and select a specific cell within the board and enter their X or O or use enter a number from 1 – 9 and select a cell in the board. The justification to use the number driven input was purely to enable keeping the gameplay simple and easy.

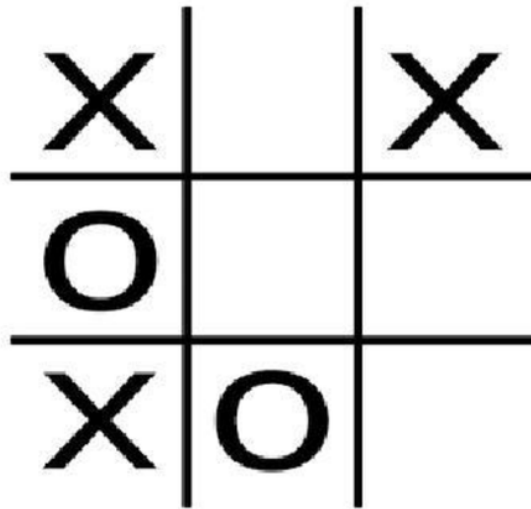


Figure 1: TicTacToe

During the process of implementing a simple Tic Tac Toe game, a tutorial was used. The Link can be found below which detailed a step by step process of how to implement it. Because the tutorial covered the fundamentals of the gameplay, the code had to be adjusted and modified to work in a network.

Link: <https://youtu.be/xwwl8TgkwgU>

3: TCP, UDP and Multi-Threading

The network aspect of the game is the main contention for this assignment. This section examines how the connection between the server and the client is made as well as the multi-threading aspect, such as critical sections and semaphores.

The assignment requires the usage of the TCP model for the network aspect that deals with information such as Player information, moves being counted and keeping the clients connected. The UDP model is used for the game which is jointly used with the multithreading component. The key difference between TCP and UDP is that UDP loses important information, while TCP does not, therefore TCP must be used for important information and UDP is used for the gameplay. However, it is important to note that there is an argument to make as to be made concerning the suitability of the network protocols specified. According to Saldana and Suznjevic [1], the “UDP is more suitable than TCP (Transmission Control Protocol) for real-time services”. The justification for this is that UDP does not expect any form of response from the receiver. However, TCP is used for games such as FPS (First Person Shooter) such as Call of Duty.

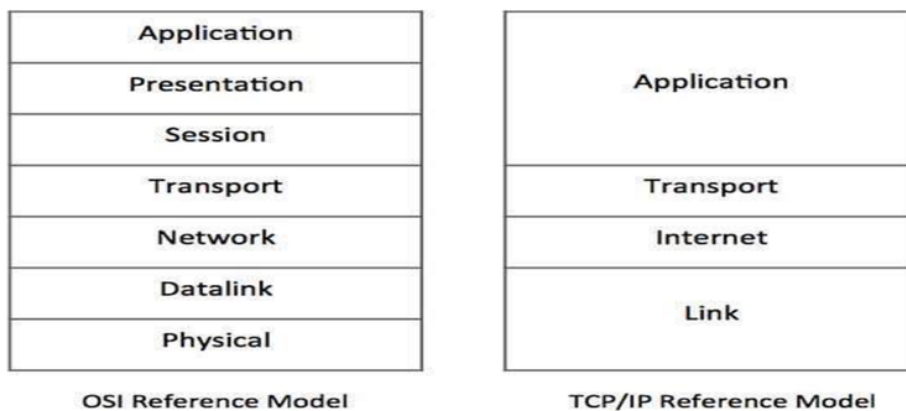


Figure 2: OSI and TCP/IP Reference Model

TCP (Transmission Control Protocol) is the model that is used for sending packets of data from the sender to the receiver. TCP is the opposite of the OSI layer model, however it only contains 4 layers instead of seven. TCP is designed as a client server model, whereas OSI is designed as a theoretical idea of a model that can be used in a network. UDP (User Datagram Protocol) is an alternative to the TCP as a connection less, loss tolerating for data packets method.

The main advantage of using TCP is that it is reliable to use, and the receiver is guaranteed to receive data packets. In an event where the server is shut down, the data packets that are queued to be sent, will be sent before the server can be closed. This means that no data packet will be lost. The second advantage is the flow of control, whereby the process of sending data is smooth and goes with the flow. The third advantage is how TCP divides and resembles messages that are divided itself into segments. The Transport Layer in the OSI layer model uses segments, dividing the data into segments, TCP then sends the data to the receiver. Once received, the data is reassembled from the receiver. Finally, TCP is connection- oriented, meaning that the session will not be interrupted if more connections are made. The problem with TCP is that it does not respect the record boundaries. Record Boundaries, by definition, is the use of rules surrounding the data being sent over the network. TCP sends Bytes and is considered a byte stream delivery service. However, applications want to send data, sequentially and TCP does not allow it. Another problem is that the process is slow, and performance is low.

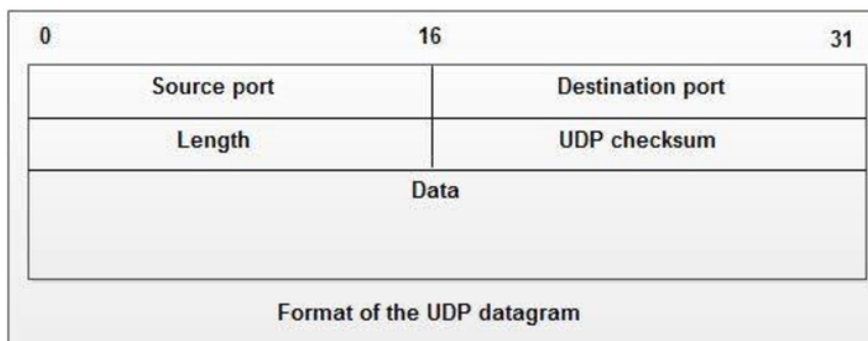


Figure 3 Diagram of UDP Datagram

UDP's biggest problem is the lack of reliability that the data sent would reach the receiver be received and forms the overall issue of lack of reliability. When working with information that is important, the majority of users do not trust UDP because data can be lost and neither the sender nor receiver will know about it. UDP does not use acknowledgements, whilst TCP does, for users to know if their data packet has been sent. It also does not have any form of flow control. UDP however does respect record boundaries and it also uses datagrams which is a different method of transmitting data from the sender to the receiver. It demonstrates the lack of restrictions when using UDP in comparison to the TCP protocol. Another advantage is that UDP is connectionless, which means that the process of sending information to the client is faster than TCP.

Multithreading can be defined as a way to utilize processors for different tasks, allocating one thread to a task each. It is used in Networking in the form of socket programming. There are five elements of multi-threading, however this assignment requires two of the five as seen below:

- 1) Critical Section
- 2) Semaphores
- 3) Mutex
- 4) Producer-Consumer
- 5) Multi Thread

Semaphores, designed by Dijkstra, who also developed the Dijkstra pathfinding algorithm in the 50's. The design of the semaphore is to maintain a count that is the value ranging from 0 to a maximum value. The counter is used to determine how many threads are active throughout execution time. Whenever a thread is completed and terminated, the counter will decrease. Semaphores are excellent for controlling shared resources that can be occupied by users. In practise, it limits the amount of threads that can be used to acquire the shared resource to make sure the threads can access them at the right time to avoid race conditions which is a substantial problem in multithreading. If all threads created acquired the shared resource, race condition will occur, and the tasks will not be completed.

In terms of the assignment, semaphores are used for the game as the design requires an input from the user. The semaphore will need to wait until a key has been entered and the game loop can be executed. The justification for this is down to the design of the game, if the game was designed where the game was updating throughout the process, semaphores would not be suitable. The alternative would be either a critical section or a mutex.

The UDP server side uses a multithread which a thread is created with an ID and assigned to run the UDP server until the game is finished. The technique used for this a multithread. This is one of the simpler choices to use as it will wait for a connection and a message in a form of a pack. Pack's are simply a data structure, designed to hold the contents of the message as well as the structure of the message. The justification for this is that there is no criteria that is required

4: The Application

The development process requires a few diagrams such as Class and networking diagrams such as TCP Protocols to further demonstrate how the protocols worked when implementing the TCP and UDP protocols, as well as the elements of multithreading. This section does include diagrams such as class diagrams that demonstrate how classes are structured.

4.1: Planning the Application

Before developing the application, there was a question of how the application should be designed. In this case, a System Sequence Diagram was created. This helps illustrate an idea of how the application will work and a diagram demonstrating this can be seen below.

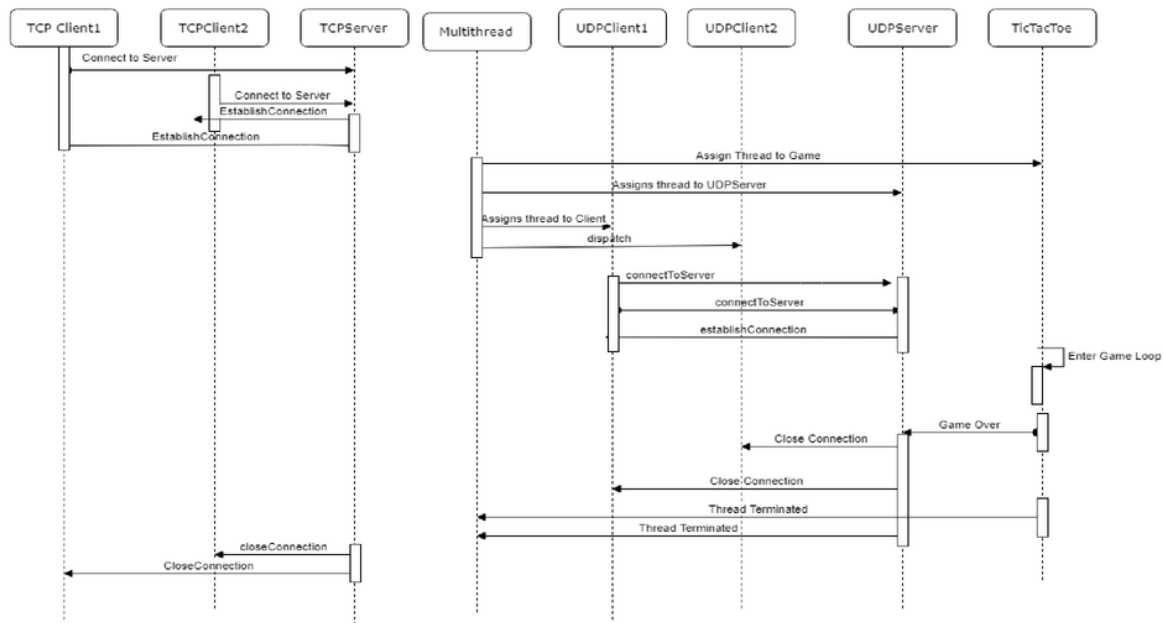


Figure 4 System Sequence Diagram of the Application

The diagram illustrates how the system will run and how each sequence works. There is another step in the planning process, which is to plan the construction of classes. The application, being developed with C++, requires classes as it is a more efficient way of developing code as well as maintaining clean code. Below is a diagram that illustrates how the classes will be designed and implemented.

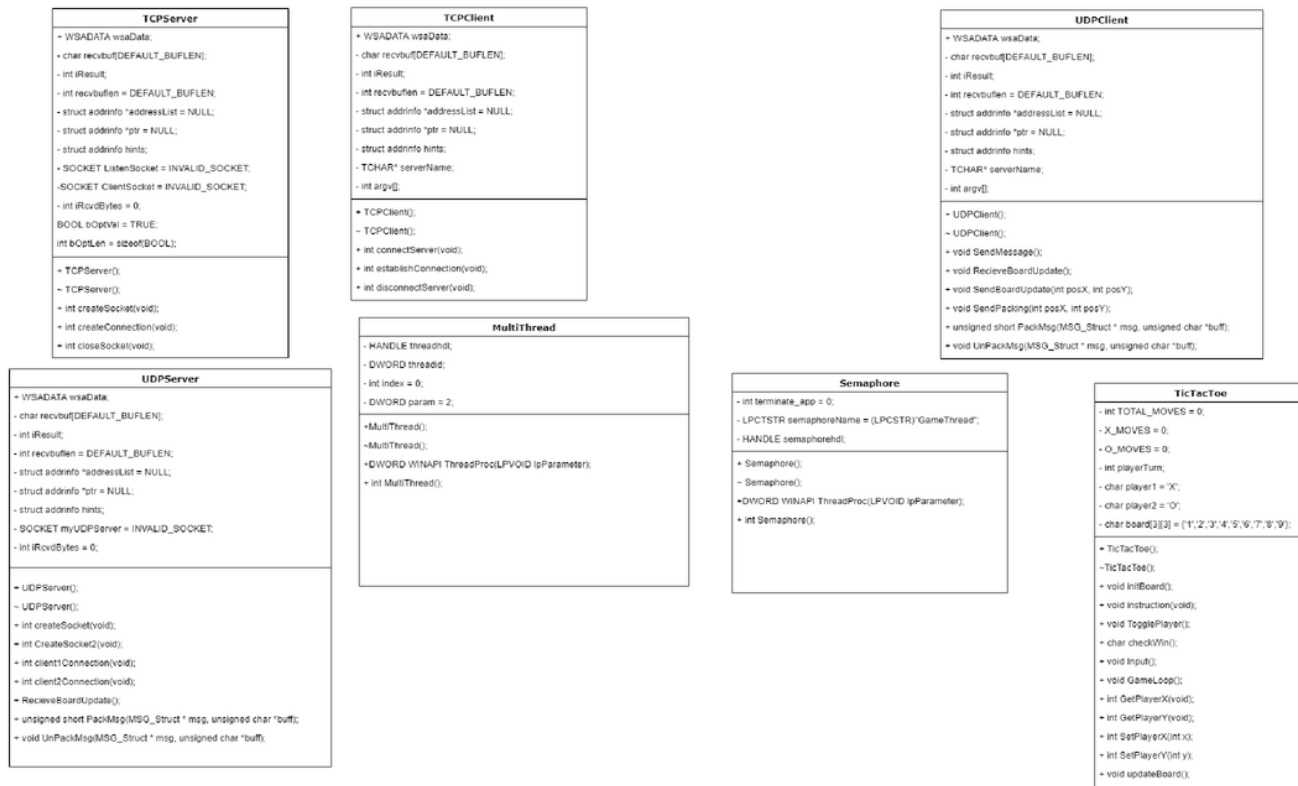


Figure 5 Class Diagram of the application

The application works can be described in the following steps:

- 1) Connection between TCP Server and Client is established.
- 2) UDP is used at which point, the multithreading also becomes active.
- 3) The game is launched.
- 4) When the game is finished, the UDP and Multi-threading is terminated.
- 5) TCP is also terminated.

4.2 Phase 1: TCP Connection

The client is required to enter their username and details which in turn, is the data type of a character. The server is executed first, listening for the first client. When the first client connects, the server will then listen for the second client. When two clients have connected, the first phase is completed and the transition to phase 2 will initiate. There is however one important aspect to note. When both clients have connected to the server, the IP address is also sent, allowing the user on the server side to view the IP address of both clients. When the server receives the IP address, this will be sent to the UDP server as to which phase 2 will take place. To send messages, Sprintf is used. Sprintf is a method that formats a string in a message. The Sprintf function in this case, relies on three parameters. The first of these is a buffer which is defined as a string buffer that contains the message with the size of the message. The second, is the contents of the message which could for example read as "Hello World". Lastly, the third parameter is the variable that is used in the message. For example, if the position of the player has been made, this will be sent to the server.

4.3 Phase 2: The UDP, The Game and The Multithread

When the TCP connection is established, and the IP address book is formed, the second phase is initiated. The multithread aspect is launched when a thread is created to contain the UDP server code. The multithread also applies to the game loop and happens after the two clients are connected to the UDP server. The choice of multithreading method for the game is a semaphore. Semaphores are, as discussed in the previous section, a synchronised object that waits for user input. The game also requires user input hence the justification for using the semaphore as it requires action from the user for the game to progress, otherwise, the game will wait for long periods of time. In terms of sequence, the Multithread for UDP is created first, leaving the server waiting for clients to join. When the clients joined, the semaphore is then created, initiating the game loop. The semaphore is active if the game loop continues. When the game loop concludes, the third phase then takes place which is the clean-up stage.

4.4 Phase 3: Clean-up

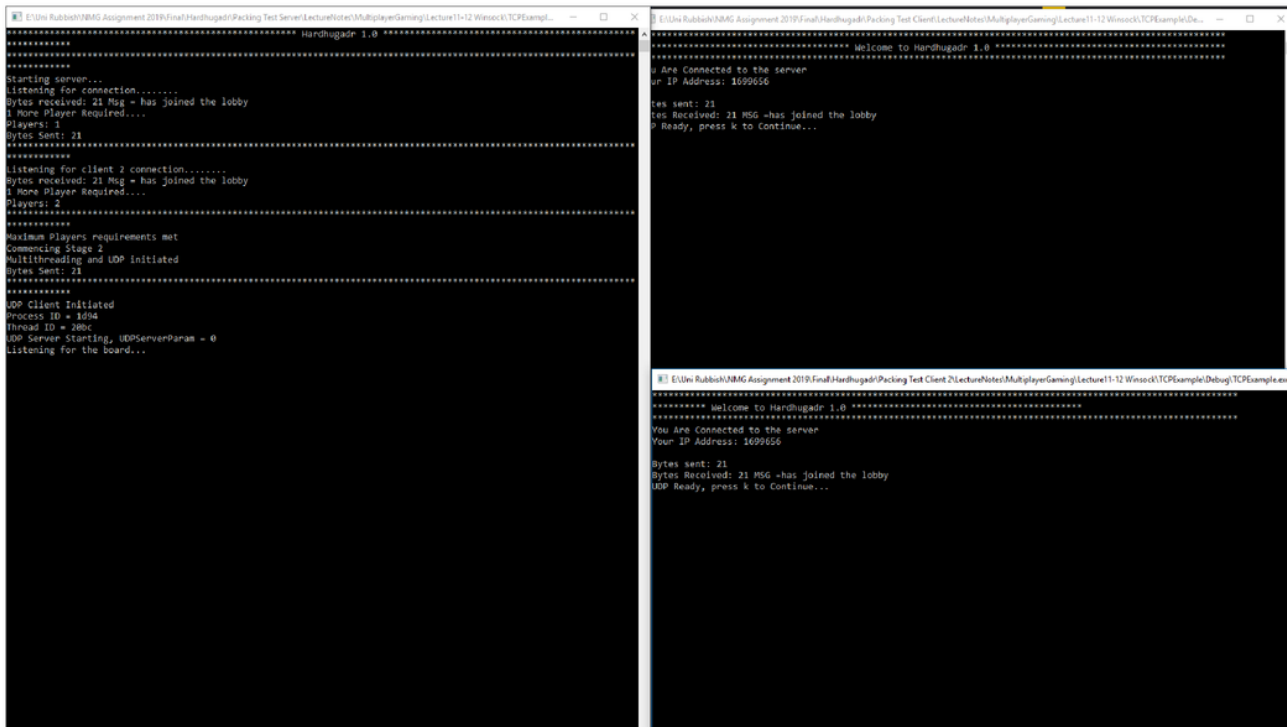
One of the important steps to take when dealing with networks and memory, is to clean-up after it has been used. There are several consequences that can happen whenever developers do not clean up after the usage of networks and memory. Firstly, Memory Leak, which is one of the traps that developers can fall into. Memory is heavily used in the practise of strings in networking, particularly sending messages to other clients and servers. When a memory leak happens, the resource that is not used is released which in good practise, the memory not used must be released. When the game loop is finished, the semaphore must be terminated when the process runs its course which the object is no longer used. The thread for the UDP server is then terminated, which means that all forms of memory are released. UDP Clients and TCP Servers and Clients are terminated, signalling the completion of the clean-up.

5: Testing

The previous section discusses the design and application of the assignment. With the implementation complete, testing is required to insure the outcome is achieved. This section of the report discusses the testing that occurred during the implementation, analysis of the results and possible improvements that can be made to the project.

5.1: TCP Testing

When testing the TCP, the test is to determine whether the client has successfully connected to the server and sends a message to ensure that the server has received the message.



```
HardHugadr 1.0
*****
Starting server...
Listening for connection.....
Bytes received: 21 MSG - has joined the lobby
1 More Player Required....
Players: 1
Bytes Sent: 21
*****
Listening for client 2 connection.....
Bytes received: 21 MSG - has joined the lobby
1 More Player Required....
Players: 2
*****
Maximum Players requirements met
Commencing Stage 2
Multithreading and UDP Initiated
Bytes Sent: 21
*****
UDP Client Initiated
Process ID = 1004
Thread ID = 200c
UDP Server Starting, UDPServerParam = 0
Listening for the board...

***** Welcome to HardHugadr 1.0 *****
You Are Connected to the server
Your IP Address: 1099056

Bytes sent: 21
Bytes Received: 21 MSG -has joined the lobby
UDP Ready, press k to Continue...
```

Figure 6 Test Result of TCP

The TCP result was a success, the client is connected to the server, in turn, the server obtains the address of the client. The message received is measured in bytes. The efficiency of the test is not necessary to perform.

5.2: UDP Testing

The UDP's test remains the same as the TCP's testing and determines if the player has connected to the UDP server and the number of bytes the server received from the client. The test was as predicted a success, the client joined the server, sending the message in the process. The test image can be seen in the Game Test below.

5.3: Game

The Game aspect does contain UDP elements such as sending messages to the server. For the messages, the only time UDP is used in the game is when the player chooses a cell.

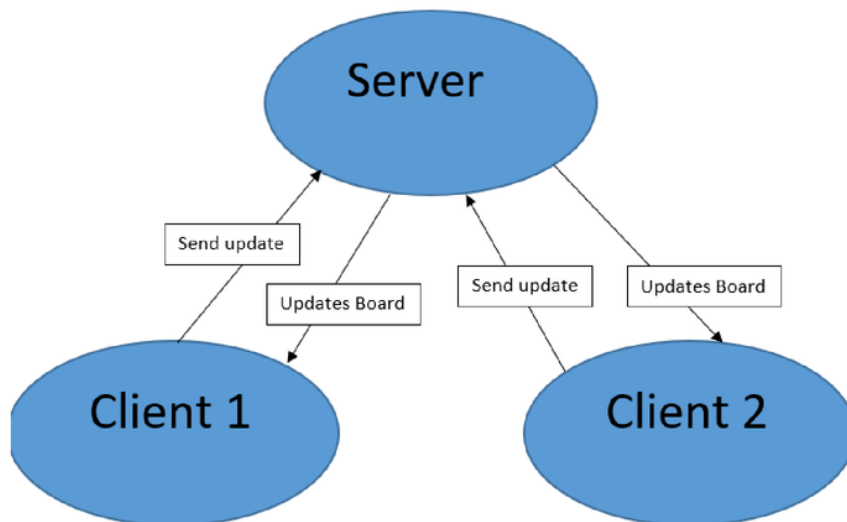


Figure 7 Simple of Diagram of Sending Game Messages

The diagram above demonstrates how messages are being sent between the three parties. When the first client enters a cell and marks 'X', the position of X and Y are stored and sent to the server where it updates the board on the server's side. Next, the server sends the change to the second client which will then update the board. To finish the loop, the second client repeats the same step as client one did and sends the change to the server, updating both the server and client one's board. This will keep going until the winner is determined or a draw occurs.

```

[Server Window]
Listening for the board...
*****Message received*****
Message received: 167 bytes code:1
User:Player 1
Player Position:X:-858993460 Player Position Y:-858993460
Player Turn: 1
dataLen:160
*****Message Sent*****
Response Sent: 167 Msg-
1 2 3
4 5 6
7 8 9
Bytes Sent: 167
Listening for the board...
*****Message received*****
Message received: 97 bytes code:1
User:Player 2
Player Position:X:-858993460 Player Position Y:-858993460
Player Turn: 2
dataLen:60
*****Message Sent*****
Response Sent: 97 Msg-
1 2 3
4 5 6
7 8 9
Bytes Sent: 97
Listening for the board...
*****Message received*****
Message received: 167 bytes code:1
User:Player 1
Player Position:X:-1 Player Position Y:-1
Player Turn: 1
dataLen:160
*****Message Sent*****
Response Sent: 167 Msg-
1 2 3
4 5 6
7 8 9
Bytes Sent: 167
Listening for the board...
*****Message received*****
Message received: 97 bytes code:1
User:Player 2
Player Position:X:0 Player Position Y:0
Player Turn: 2
dataLen:60
*****Message Sent*****
Response Sent: 97 Msg-
1 2 3
4 5 6
7 8 9
Bytes Sent: 97
Listening for the board...

[Client Window]
Player Position Y:-858993460
Player Turn: 2
dataLen:60
Bytes sent: 97
*****Message received*****
code:1
New Player Position:X:-858993460 New Player Position Y:-858993460
dataLen:60
5
1 2 3
4 5 6
7 8 9
Total Moves Made: 1
Total Moves Player 1 (X) Made: 0
Total Moves Player 2 (O) Made: 1
Message being sent: code:1
User:Player 2
Player Position:X:0
Player Position Y:0
Player Turn: 2
dataLen:60
Bytes sent: 97
*****Message received*****
code:1
New Player Position:X:0 New Player Position Y:0
dataLen:60
[Client Window]
Player Turn: 1
dataLen:160
Bytes received: 167 Msg-
1 2 3
4 5 6
7 8 9
Total Moves Made: 1
Total Moves Player 1 (X) Made: 1
Total Moves Player 2 (O) Made: 0
*****Message Sent*****
code:1
User:Player 1
Player Position:X:-1
Player Position Y:-1
dataLen:1
Bytes sent: 167
*****Message received*****
code:1
New Player Position:X:-1 New Player Position Y:-1
Player Turn: 1
dataLen:160
Bytes received: 167 Msg-

```

Figure 8 Test Result of Game and UDP

The technique used for this is Packing and Unpacking. This requires the use of a structure, defining fields such as the position of the player when they choose a cell on the board and the player's turn. When the first client enters their chosen cell, the position is recorded and placed into the structure, ready to be packed and send it to the server. The server receives it, unpacks the message and display the contents.

6: Conclusion

The report gives a detailed breakdown of how the application is constructed, as well as other theoretical aspects that give more technical insight during the practice of socket programming. The decisions made during the implementation phase have been explained as well as the advantages and disadvantages of each technique of that choice. The project's implementation was a success, with test results examined and recommendations for further improvement to the project specified. In a critical analysis, there are a few short comings that have been recognised towards the development cycle of the project. One of the short comings is to have the client's boards both updated when a move has been made. This does require the second client and the server to be listening, however the problem that will be encountered is that one of them has to listen as if two are listening, one of them will crash and the application will be shut down. If the problem occurred with the three applications on one machine, it is not a problem as you can see the board being updated. However, if the applications are spread on three different computers, the problem will occur as the clients will not see the current board.

Another short coming is that both applications must not have more than one move where the board is not updated. This is mainly due to the semaphore's implementation where it is waiting for a keyboard input. With the improvements noted, the project will be further improved upon in the future. With the debate regarding if TCP or UDP should be used for multiplayer games ongoing, it is certainly interesting to see both sides of the argument. However, with big games, dedicated servers are used due to the amount of information being sent between client and server.

7: Personal Notes

This section is dedicated to the notes that have been written in conjunction with the assignment. This explores developing the servers as well as the game which are the main focus of these notes. As well as that, these notes examined the theory in more detail. These notes are not subjected to marking and are purely for the interest of research.

The game by choice has to be easy to make, as the majority of the focus of the assignment should be on how data is being sent and what methods should be used. As from the last academic year, a maze game was used. When that is processed, the site will then highlight similarities to my original report submitted at that time, in turn, stating that I plagiarised my own report. As the site was developed by a third party, the logic here is eye rolling to say the least. So a new game had to be made, in turn, Tic Tac Toe was a perfect choice due to the easy and straightforward process of gameplay.

One of the main topics that kept appearing is the debate of the choice of TCP or UDP for games. While the majority of debates are in favour of TCP due to the benefit of reliability that messages will be sent to the receiver, the UDP aspect is not favoured as much due to its many issues. It is understandable, that the majority of games favour reliability due to the amount of information and while, it is very easy to dismiss UDP and favour TCP, it should not be ruled out from the onset. UDP has its benefits for other applications such as games. UDP, for example, can be used for real time applications due to the fact that it does not take a lot of time to process information unlike TCP.

8: References

- [1] M. S. Jose Saldana, "QoE and Latency Issues in Networked Games," in *Handbook of Digital Games and Entertainment Technologies*, Singapore, Springer, 2015, pp. 1 - 36.

8: Appendix

This section contains snippets of the code that was developed for the assignment.

8.1 Server Side

8.1.1 MultiThread

```
1  #pragma once
2  #include "stdafx.h"
3  #include "MultiThread.h"
4  // #include "TicTacToe.h"
5  #include "UDPServer.h"
6  //Game Thread Variables
7  static HANDLE ThreadHdl;
8  static DWORD ThreadId;
9  static int Index;
10 static DWORD param;
11
12 //UDP Server Thread Variables
13 static HANDLE UDPServerThreadHdl;
14 static DWORD UDPServerThreadId;
15 static int UDPServerIndex;
16 static DWORD UDPServerparam;
17
18
19 class MultiThread
20 {
21 public:
22     MultiThread();
23     ~MultiThread();
24     //static DWORD WINAPI GameThreadProc(LPVOID lpParameter);
25     static DWORD WINAPI UDPServerThreadProc(LPVOID SerlpParameter);
26     static int MultiThreadMain();
27 };
28
29
```

```

1  #include "stdafx.h"
2  #include "MultiThread.h"
3
4
5  MultiThread::MultiThread()
6  {
7  }
8
9
10 MultiThread::~MultiThread()
11 {
12 }
13
14 DWORD MultiThread::UDPServerThreadProc(LPVOID SerlpParameter)
15 {
16     UDPServerparam = (DWORD)SerlpParameter;
17     UDPServerIndex = 0;
18     UDPServer myUDPServer;
19
20     printf("UDP Server Starting, UDPServerParam = %d\r\n", UDPServerparam);
21
22     for (int i = 0; i < 9; i++)
23     {
24
25         ////////////////////////////////////////////
26         //Listen for any update messages for the the game board
27         ////////////////////////////////////////////
28         myUDPServer.ReceiveBoardUpdate();
29
30     }
31
32
33
34
35
36     return 0;
37 }
38
39 int MultiThread::MultiThreadMain()
40 {
41
42     printf("Process ID = %x\r\n", GetCurrentProcessId());
43     printf("Thread ID = %x\r\n", GetCurrentThreadId());
44
45
46     while (1)
47     {
48         ThreadHdl = CreateThread(NULL, 0, UDPServerThreadProc, (LPVOID)param, 0, &Threadid);
49         getchar();
50     }

```

```

int MultiThread::MultiThreadMain()
{
    printf("Process ID = %x\r\n", GetCurrentProcessId());
    printf("Thread ID = %x\r\n", GetCurrentThreadId());

    while (1)
    {
        ThreadHdl = CreateThread(NULL, 0, UDPServerThreadProc, (LPVOID)param, 0, &Threadid);
        getchar();
    }

    TerminateThread(ThreadHdl, 0);
    CloseHandle(ThreadHdl);
    printf("\r\n Terminating Threads \r\n");
    return 0;
}

```

8.1.2 Packing

Header file

```
#pragma once
#define MSGCODE 0x0001
class Packing
{
    //Create the data structure for the message
    //Create the macros required for packing the message
    typedef struct
    {
        unsigned short code;
        unsigned short length;
        unsigned char username[20];
        unsigned long longValue;
        unsigned short shortvalue;
        unsigned char byteValue;
        int intValue;
        unsigned short datalen;
        unsigned char data[100];
    }MSG_STRUCT;

    //Create the macros required for packing the message
    //Create the macros for unpacking the messages contents
    #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;

    // UNPACKING
    //Create the macros for unpacking the messages contents
    #define UnpackBytes(bf, bytes, len) \
        memcpy(bytes, bf, len); \
        bf += len;

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

```

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

// UNPACKING
//Create the macros for unpacking the messages contents
//Create the macros for unpacking the messages contents
#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++;

public:
    Packing();
    ~Packing();

//Create a method that will pack the contents of the message before sending to the user
//Create a method that will pack the contents of the message before sending to the user
    unsigned short PackMsg(MSG_STRUCT *msg, unsigned char *buff);
//Create a method that will pack the contents of the message before sending to the user
//Create a method that will pack the contents of the message before sending to the user
    void UnpackMSG(MSG_STRUCT *msg, unsigned char *buff);
};

```

.cpp

```

#include "stdafx.h"
#include "Packing.h"

Packing::Packing()
{
}

Packing::~Packing()
{
}

//This method is used to pack the message to send to the players during gametime
//This method is used to pack the message to send to the players during gametime
unsigned short Packing::PackMsg(MSG_STRUCT * msg, unsigned char * buff)
{
    unsigned short msglen;
    unsigned char *lenptr;

    PackShort(buff, msg->code);
    lenptr = buff;
    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);

    msglen = sizeof(unsigned short)
        + sizeof(unsigned short)
        + sizeof(msg->username)
        + sizeof(unsigned long)
        + sizeof(unsigned short)
        + 1
        + sizeof(unsigned long)
        + sizeof(unsigned short)
        + msg->datalen;

    PackShort(lenptr, msglen);
    return msglen;
}

```

```

}
}
//This method is used to unpack the message that is recieved from the players during game time
void Packing::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);
}

```

8.1.3 TCPServer

```

#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLen 512
class TCPServer3
{
private:
    WSADATA wsaData;
    WSADATA wsaData2;
    char   recvbuf[DEFAULT_BUFLen];
    char   recvbuf2[DEFAULT_BUFLen];
    int    iResult;
    int    iResult2;
    int    recvbuflen = DEFAULT_BUFLen;
    int    recvbuflen2 = DEFAULT_BUFLen;
    int    playerCount = 0;
    struct addrinfo *addressList = NULL;
    struct addrinfo *ptr = NULL;
    struct addrinfo hints;

    struct addrinfo *addressList2 = NULL;
    struct addrinfo *ptr2 = NULL;
    struct addrinfo hints2;

    char   userNameBuf[DEFAULT_BUFLen];
    char   userNameBuf2[DEFAULT_BUFLen];
    char   responseBuf[DEFAULT_BUFLen];
    //For player 1
    SOCKET ListenSocket1 = INVALID_SOCKET;
    //For Player 2
    SOCKET ListenSocket2 = INVALID_SOCKET;

    //For Player 1
    SOCKET ClientSocket1 = INVALID_SOCKET;
    //For Player 2
    SOCKET ClientSocket2 = INVALID_SOCKET;
    int    iRcvdBytes = 0;
    int    iRcvdBytes2 = 0;
    BOOL   bOptVal = TRUE;
    int    bOptLen = sizeof(BOOL);
    BOOL   bOptVal2 = TRUE;
    int    bOptLen2 = sizeof(BOOL);
}

```

```

public:
    TCPServer3();
    ~TCPServer3();
    //Create the sockets for Player 1
    int createSocket(void);
    //Create the sockets for Player 2
    int createSocket2(void);
    //Recieve player 1
    int recievePlayer1(void);
    //Recieve player 2
    int recievedPlayer2(void);
    //Create the connection
    int createConnection(void);
    //Close the socket
    int closeSocket(void);
};

```

.cpp

```

#include "stdafx.h"
#include "TCPServer3.h"

TCPServer3::TCPServer3()
{
}

TCPServer3::~TCPServer3()
{
}

//This method is to create the socket for the first player to connect to
//int TCPServer3::createSocket(void)
int TCPServer3::createSocket(void)
{
    iResult = WSASStartup(WKWORD(2, 2), &wsaData);

    if (iResult != 0)
    {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    printf("Starting server...\n\n");

    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;

    // Resolve the server address and port
    iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &addressList);

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

    // Create a SOCKET for connecting to server
    ListenSocket1 = socket(addressList->ai_family, addressList->ai_socktype, addressList->ai_protocol);

    if (ListenSocket1 == INVALID_SOCKET)
    {
        printf("socket failed: %d\n", WSAGetLastError());
        freeaddrinfo(addressList);
        WSACleanup();
        return 1;
    }

    setsockopt(ListenSocket1, SOL_SOCKET, SO_REUSEADDR, (char*)&bOptVal, bOptLen);

    if (setsockopt(ListenSocket1, SOL_SOCKET, SO_REUSEADDR, (char*)&bOptVal, bOptLen) == SOCKET_ERROR)
    {
        printf("setsockopt failed: %d\n", WSAGetLastError());
        closesocket(ListenSocket1);
        WSACleanup();
        return 1;
    }

    iResult = bind(ListenSocket1, addressList->ai_addr, (int)addressList->ai_addrlen);

    if (iResult == SOCKET_ERROR)
    {
        printf("bind failed: %d\n", WSAGetLastError());
        freeaddrinfo(addressList);
        closesocket(ListenSocket1);
        WSACleanup();
        return 1;
    }

    freeaddrinfo(addressList);
    recievePlayer1();
    sprintf(responseBuf, "%s", "Two Players are connected, Please Press a Key To begin", clock());
    //createConnection();
}

```

```

int TCPServer3::createConnection(void)
{
    printf("Listening for connection..... \r\n"); //listen for connection for the cl

    iResult = listen(ListenSocket1, SOMAXCONN);
    while (!_kbhit()) // Loop until key pressed
    {
        if (iResult == SOCKET_ERROR)
        {
            printf("listen failed: %d\n", WSAGetLastError());
            closesocket(ListenSocket1);
            WSACleanup();
            return 1;
        }

        ClientSocket1 = accept(ListenSocket1, NULL, NULL);
        if (ClientSocket1 == INVALID_SOCKET)
        {
            printf("accept failed: %d\n", WSAGetLastError());
            closesocket(ListenSocket1);
            WSACleanup();
            return 1;
        }

        do
        {
            //iResult = recv(ClientSocket, userNameBuf, recvbuflen, 0);
            iResult = recv(ClientSocket1, recvbuf, recvbuflen, 0);
            //Recieve the second player

            if (iResult > 0)
            {
                printf("Bytes received: %d Msg = %s\n", iResult, recvbuf);
                printf("1 More Player Required...\n");
                playerCount++; //Increase the player Count
                printf("Players: %d", playerCount);
                iRcvdBytes = iResult;
            }
            else if (iResult == 0)
                printf("Connection closing...\n");
            else
            {
                printf("recv failed: %d\n", WSAGetLastError());
                closesocket(ClientSocket1);
                WSACleanup();
                return 1;
            }
        }
    }
}

```

```

    }
    else if (iResult == 0)
        printf("Connection closing...\n");
    else
    {
        printf("recv failed: %d\n", WSAGetLastError());
        closesocket(ClientSocket1);
        WSACleanup();
        return 1;
    }

    if (playerCount == 2)
    {
        printf("*****\n");
        printf("Maximum Players requirements met \n");
        printf("Commencing Stage 2 \n");
        printf("Multithreading and UDP initiated \n");
    }
} while (iResult > 0);

// Echo the buffer back to the sender
iResult = send(ClientSocket1, rcvbuf, iRcvdBytes, 0);
if (iResult == SOCKET_ERROR)
{
    printf("send failed: %d\n", WSAGetLastError());
    closesocket(ClientSocket1);
    WSACleanup();
    return 1;
}

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

//iResult = shutdown(ClientSocket1, SD_SEND);
//iResult2 = shutdown(ClientSocket2, SD_SEND);
//if (iResult == SOCKET_ERROR)
//{
//    printf("shutdown failed: %d\n", WSAGetLastError());
//    closesocket(ClientSocket1);
//    WSACleanup();
//    return 1;
//}

// cleanup
//closesocket(ClientSocket);
}
//closesocket(ListenSocket);

printf("\n\nHit key to exit");
while (!kbhit());
//closeSocket();

```

8.1.4 TicTacToe

```
#pragma once
#include "stdafx.h"
#define BOARD_WIDTH 3
#define BOARD_HEIGHT 3

class TicTacToe
{
private:
    struct Player
    {
        int X;
        int Y;
        char player1 = 'X';
        char player2 = 'O';
    }player;

    int playerTurn;
    char player1 = 'X';
    char player2 = 'O';
    char board[3][3] = { '1', '2', '3', '4', '5', '6', '7', '8', '9' };
    bool update = false;
public:
    TicTacToe();
    ~TicTacToe();

    //Create the board
    void initBoard();
    //Check Win
    char checkWin();
    //Check whos turn it is
    int CheckTurn(int);
    //Game Loop
    void GameLoop();
    void updateBoard();
    // void SetChangePlayer2(char player2);
    int SetPlayerX(int x);
    int SetPlayerY(int y);
    int SetPlayerTurn(int playerTurn);
    char lastMove();
};
```

```
#include "stdafx.h"
#include "TicTacToe.h"

TicTacToe::TicTacToe()
{
}

TicTacToe::~TicTacToe()
{
}

//This method is to initialise board
void TicTacToe::initBoard()
{
    //system("CLS");
    updateBoard();

    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
    //lastMove();
}
```

```

////////////////////////////////////
//This method is to check if one of the players has won
////////////////////////////////////
char TicTacToe::checkWin()
{
    //////////////////////////////////////
    //Check if Player One Wins
    //////////////////////////////////////

    //////////////////////////////////////
    //Check Across
    //////////////////////////////////////
    if (board[0][0] == 'X' && board[0][1] == 'X' && board[0][2] == 'X')
    {
        return 'X';
    }

    if (board[1][0] == 'X' && board[1][1] == 'X' && board[1][2] == 'X')
    {
        return 'X';
    }

    if (board[2][0] == 'X' && board[2][1] == 'X' && board[2][2] == 'X')
    {
        return 'X';
    }

    //////////////////////////////////////
    //Down
    //////////////////////////////////////
    if (board[0][0] == 'X' && board[1][0] == 'X' && board[2][0] == 'X')
    {
        return 'X';
    }

    if (board[0][1] == 'X' && board[1][1] == 'X' && board[2][1] == 'X')
    {
        return 'X';
    }

    if (board[0][2] == 'X' && board[1][2] == 'X' && board[2][2] == 'X')
    {
        return 'X';
    }

    //////////////////////////////////////
    //Diagonal
    //////////////////////////////////////
    if (board[0][0] == 'X' && board[1][1] == 'X' && board[2][2] == 'X')
    {
        return 'X';
    }
}

```

```

}
//Diagonal
if (board[0][0] == 'X' && board[1][1] == 'X' && board[2][2] == 'X')
{
    return 'X';
}

if (board[2][0] == 'X' && board[1][1] == 'X' && board[0][2] == 'X')
{
    return 'X';
}

//check if Player 2 wins
//Check Across
if (board[0][0] == 'O' && board[0][1] == 'O' && board[0][2] == 'O')
{
    return 'O';
}

if (board[1][0] == 'O' && board[1][1] == 'O' && board[1][2] == 'O')
{
    return 'O';
}

if (board[2][0] == 'O' && board[2][1] == 'O' && board[2][2] == 'O')
{
    return 'O';
}

//Down
if (board[0][0] == 'O' && board[1][0] == 'O' && board[2][0] == 'O')
{
    return 'O';
}

if (board[0][1] == 'O' && board[1][1] == 'O' && board[2][1] == 'O')
{
    return 'O';
}

if (board[0][2] == 'O' && board[1][2] == 'O' && board[2][2] == 'O')

```

```

if (board[0][0] == 'O' && board[0][1] == 'O' && board[0][2] == 'O')
{
    return 'O';
}

if (board[1][0] == 'O' && board[1][1] == 'O' && board[1][2] == 'O')
{
    return 'O';
}

if (board[2][0] == 'O' && board[2][1] == 'O' && board[2][2] == 'O')
{
    return 'O';
}

//Down
if (board[0][0] == 'O' && board[1][0] == 'O' && board[2][0] == 'O')
{
    return 'O';
}

if (board[0][1] == 'O' && board[1][1] == 'O' && board[2][1] == 'O')
{
    return 'O';
}

if (board[0][2] == 'O' && board[1][2] == 'O' && board[2][2] == 'O')
{
    return 'O';
}

//Diagonal
if (board[0][0] == 'O' && board[1][1] == 'O' && board[2][2] == 'O')
{
    return 'O';
}

if (board[2][0] == 'O' && board[1][1] == 'O' && board[0][2] == 'O')
{
    return 'O';
}

return '/';

```

```

//This method is to check whose turn it is
//This method is to check whose turn it is
int TicTacToe::CheckTurn(int)
{
    if (playerTurn == 1)
    {
        playerTurn = 2;
    }
    else
    {
        playerTurn = 1;
    }
    return playerTurn;
}

//This method is to start the game loop which will be used in the semaphore
//This method is to start the game loop which will be used in the semaphore
void TicTacToe::GameLoop()
{
    initBoard();
    for (int i = 0; i < 9; i++)
    {
        initBoard();
        if (checkWin() == 'X')
        {
            cout << "Player 1 wins!\n";
            break;
        }
        if (checkWin() == 'O')
        {
            cout << "Player 2 wins!\n";
            break;
        }
    }
    system("pause");
}

```

```

//This method is to set the player X
//This method is to set the player Y
int TicTacToe::SetPlayerX(int x)
{
    player.X = x;
    //printf("X: %d\n", player.X);

    return player.X;
}

int TicTacToe::SetPlayerY(int y)
{
    //Get the position sent from the client and have the player store it
    player.Y = y;
    //printf("Y: %d\n", player.Y);

    return player.Y;
}

int TicTacToe::SetPlayerTurn(int Turn)
{
    playerTurn = Turn;

    return playerTurn;
}

//This method is to get the last move that was made
char TicTacToe::lastMove()
{
    printf("Last Move made: X: %i , Y: %i\n", player.X, player.Y);
    if (player.X && player.Y == 'X')
    {
        printf("Player 1 Made the last move (X)\n");
    }

    if (player.X && player.Y == '0')
    {
        printf("Player 2 Made the last move (0)\n");
    }

    return 0;
}

```

8.1.5 UDPServer

```
#pragma once
#include "stdafx.h"
#include "TicTacToe.h"
#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLen 512

class UDPServer
{
private:
    WSADATA wsaData;
    char recvbuf[DEFAULT_BUFLen];
    int iResult;
    int recvbuflen = DEFAULT_BUFLen;
    int iRcvdBytes = 0;
    sockaddr_in rx_addr;
    sockaddr_in sender_addr;
    int iSenderAddrSize = sizeof(sender_addr);
    struct addrinfo *addresslist = NULL;
    struct addrinfo *ptr = NULL;
    struct addrinfo hints;
    SOCKET myUDPServer = INVALID_SOCKET;
    TicTacToe myGame;
    int x;
    int y;

    WSADATA wsaData2;
    char recvbuf2[DEFAULT_BUFLen];
    int iResult2;
    int recvbuflen2 = DEFAULT_BUFLen;

    struct addrinfo *addresslist2 = NULL;
    struct addrinfo *ptr2 = NULL;
    struct addrinfo hints2;
    SOCKET myUDPServer2 = INVALID_SOCKET;

    int iRcvdBytes2 = 0;
    sockaddr_in rx_addr2;
    sockaddr_in sender_addr2;
    int iSenderAddrSize2 = sizeof(sender_addr2);
    char recvbuf3[1000];
    SOCKET myUDPServer3 = INVALID_SOCKET;
```

```

//Create the data structure for the message
//Create the macros required for packing the message
typedef struct
{
    unsigned short code;
    unsigned short length;
    unsigned char username[20];
    unsigned long longValue;
    unsigned short shortValue;
    unsigned char byteValue;
    int PlayerPositionX;
    int PlayerPositionY;
    int playerTurn;
    unsigned short datalen;
    unsigned char data[100];
}MSG_STRUCT;
unsigned short buflen;
//Create the macros required for packing the message
#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++;

public:
    UDPServer();
    ~UDPServer();

    //Create the sockets for Player 1
    int createSocket1(void);

    //Create the sockets for Player 2
    int createSocket2(void);

    //Create the socket connection for Player 1
    int client1Connection(void);

    //Create the sockets for Player 2
    int client2Connection(void);

    //Recieve Board Update during game time
    void ReceiveBoardUpdate();
    // void RecieveUpdate();
    // void RecieveUpdate2();

    //Create a method that will pack the contents of the message before sending to the user
    unsigned short PackMsg(MSG_STRUCT *msg, unsigned char *buff);

    //Create a method that will pack the contents of the message before sending to the user
    void UnpackMSG(MSG_STRUCT *msg, unsigned char *buff);
};

```

.cpp

```
#include "stdafx.h"
#include "UDPServer.h"

UDPServer::UDPServer()
{
}

UDPServer::~UDPServer()
{
}

//Create the sockets for Player 1
//Create the sockets for Player 2
int UDPServer::createSocket1(void)
{
    iResult = WSASStartup(MAKEWORD(2, 2), &wsaData);
    int iRcvdBytes = 0;
    sockaddr_in rx_addr;
    sockaddr_in sender_addr;
    int iSenderAddrSize = sizeof(sender_addr);
    myUDPServer = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP); //set the socket up
    printf("Listening for Client 1 Connection... \n");

    return 0;
}

//Create the sockets for Player 2
int UDPServer::createSocket2(void)
{
    iResult2 = WSASStartup(MAKEWORD(2, 2), &wsaData2);
    int iRcvdBytes2 = 0;
    sockaddr_in rx_addr2;
    sockaddr_in sender_addr2;
    int iSenderAddrSize2 = sizeof(sender_addr2);
    myUDPServer2 = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP); //set the socket up

    return 0;
}
```

```

//Create the connection for the first client
//Create the connection for the first client
int UDPServer::client1Connection()
{
    rx_addr.sin_family = AF_INET; //set the family
    rx_addr.sin_port = htons((u_short)atol(DEFAULT_PORT));
    rx_addr.sin_addr.s_addr = htonl(INADDR_ANY); //listen for any port address
    iResult = bind(myUDPServer, (SOCKADDR*)&rx_addr, sizeof(rx_addr));
    //Who is the sender

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

    printf("Bytes recieved: %d MSG = %s\n", iResult, recvbuf);
    printf("Player 1 Connected... \n");
    printf("Waiting For Second Client... \n");

    //Send to original sender
    iResult = sendto(myUDPServer, recvbuf, iResult, 0, (SOCKADDR*)&sender_addr, iSenderAddrSize);
    //Print how many bytes were sent to the reciever
    printf("Bytes Sent: %ld\n", iResult);
    printf("*****\n");
    //Close the socket to tidy up
    closesocket(myUDPServer);
    return 1;
}

//Create the connection for the second client
//Create the connection for the second client
int UDPServer::client2Connection()
{
    printf("*****\n");
    printf("Listening for Connection for Client 2...\n");

    rx_addr2.sin_family = AF_INET; //set the family
    rx_addr2.sin_port = htons((u_short)atol(DEFAULT_PORT));
    rx_addr2.sin_addr.s_addr = htonl(INADDR_ANY); //listen for any port address
    iResult2 = bind(myUDPServer2, (SOCKADDR*)&rx_addr2, sizeof(rx_addr2));
    //Who is the sender
    iResult2 = recvfrom(myUDPServer2, recvbuf2, recvbuflen2, 0, (SOCKADDR *)&sender_addr2, &iSenderAddrSize2);
    printf("Player 2 Connected... \n");
    printf("Bytes recieved: %d MSG = %s\n", iResult2, recvbuf2);
    iResult2 = sendto(myUDPServer2, recvbuf2, iResult2, 0, (SOCKADDR*)&sender_addr2, iSenderAddrSize2);
    //Print how many bytes were sent to the reciever
    printf("Bytes Sent: %ld\n", iResult2);
    printf("*****\n");
    closesocket(myUDPServer);
    return 1;
}

```

```

void UDPServer::ReceiveBoardUpdate()
{
    unsigned char sendbuff2[1000];
    SOCKET UDPServer2 = INVALID_SOCKET;
    char recvbuff2[DEFAULT_BUFLEN];
    iResult = WSASStartup(MAKEWORD(2, 2), &wsaData);
    int iRcvdBytes = 0;
    sockaddr_in rx_addr;
    sockaddr_in sender_addr;
    MSG_STRUCT msg;
    int iSenderAddrSize = sizeof(sender_addr);
    UDPServer2 = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP); //set the socket up
    printf("Listening for the board... \n");

    rx_addr.sin_family = AF_INET; //set the family
    rx_addr.sin_port = htons((u_short)atoi(DEFAULT_PORT));
    rx_addr.sin_addr.s_addr = htonl(INADDR_ANY); //listen for any port address

    iResult = bind(UDPServer2, (SOCKADDR*)&rx_addr, sizeof(rx_addr));
    iResult = recvfrom(UDPServer2, recvbuff2, recvbufflen, 0, (SOCKADDR *)&sender_addr, &iSenderAddrSize);

    UnpackMSG(&msg, (unsigned char*)recvbuff2);

    printf("*****Message received*****\n");
    printf("\n");
    printf("Message received: %d bytes code:%i\n User:%s\n Player Position X:%i Player Position Y:%i\n Player Turn: %i\n datalen:%i\n",
        iResult, msg.code,
        msg.username,
        msg.PlayerPositionX,
        msg.PlayerPositionY,
        msg.playerTurn,
        msg.datalen);
    printf("*****\n");
    //strcpy(recvbuff2, "Message Received, Updating Board\n");
    //iResult = strlen(recvbuff2) + 1;

    //pack the message
    buflen = PackMsg(&msg, sendbuff2);
    printf("*****Message Sent*****\n");
    printf("\n");
    //Send to original sender
    iResult = sendto(UDPServer2,
        recvbuff2,
        iResult,
        0, // flags - we never need them so zero them
        (SOCKADDR *)&sender_addr,
        iSenderAddrSize);

    printf("Response Sent: %d Msg-%s\n", iResult, recvbuff2);
    myGame.SetPlayerX(msg.PlayerPositionX);
    myGame.SetPlayerY(msg.PlayerPositionY);
    myGame.SetPlayerTurn(msg.playerTurn);
}

```

```

printf("Listening for the board... \n");

rx_addr.sin_family = AF_INET; //set the family
rx_addr.sin_port = htons((u_short)atoi(DEFAULT_PORT));
rx_addr.sin_addr.s_addr = htonl(INADDR_ANY); //listen for any port address

iResult = bind(UDPServer2, (SOCKADDR*)&rx_addr, sizeof(rx_addr));
iResult = recvfrom(UDPServer2, recvbuff2, recvbuflen, 0, (SOCKADDR *)&sender_addr, &iSenderAddrSize);

UnpackMsg(&msg, (unsigned char*)recvbuff2);

printf("*****Message received*****\n");
printf("\n");
printf("Message received: %d bytes code:%i\n User:%s\n Player Position:X:%i Player Position Y:%i\n Player Turn: %i\n datalen:%i\n",
    iResult, msg.code,
    msg.username,
    msg.PlayerPositionX,
    msg.PlayerPositionY,
    msg.playerTurn,
    msg.datalen);
printf("*****\n");
//strcpy(recvbuff2, "Message Received, Updating Board\n");
//iResult = strlen(recvbuff2) + 1;

//pack the message
buflen = PackMsg(&msg, sendbuff2);
printf("*****Message Sent*****\n");
printf("\n");
//Send to original sender
iResult = sendto(UDPServer2,
    recvbuff2,
    iResult,
    0,
    (SOCKADDR *)&sender_addr,
    iSenderAddrSize);

printf("Response Sent: %ld Msg=%s\n", iResult, recvbuff2);
myGame.SetPlayerX(msg.PlayerPositionX);
myGame.SetPlayerY(msg.PlayerPositionY);
myGame.SetPlayerTurn(msg.playerTurn);
myGame.initBoard();

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

printf("*****\n");
//Close the socket to tidy up
closesocket(UDPServer2);
WSACleanup();

return;
}

```

```

//Create a method that will pack the contents of the message before sending to the user
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
unsigned short UDPServer::PackMsg(MSG_STRUCT * msg, unsigned char * buff)
{
    unsigned short msglen;
    unsigned char *lenptr;

    PackShort(buff, msg->code);
    lenptr = buff;
    buff += sizeof(unsigned short);
    PackBytes(buff, msg->username, 20);
    PackLong(buff, msg->longValue);
    PackShort(buff, msg->shortValue);
    PackByte(buff, msg->byteValue);
    PackLong(buff, msg->PlayerPositionX);
    PackLong(buff, msg->PlayerPositionY);
    PackLong(buff, msg->playerTurn);
    PackShort(buff, msg->datalen);
    PackBytes(buff, msg->data, msg->datalen);

    msglen = sizeof(unsigned short)
        + sizeof(unsigned short)
        + sizeof(msg->username)
        + sizeof(unsigned long)
        + sizeof(unsigned short)
        + 1
        + sizeof(unsigned long)
        + sizeof(unsigned short)
        + msg->datalen;

    PackShort(lenptr, msglen);
    return msglen;
}

//Create a method that will pack the contents of the message before sending to the user
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void UDPServer::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->PlayerPositionX);
    UnpackLong(buff, msg->PlayerPositionY);
    UnpackLong(buff, msg->playerTurn);
    UnpackShort(buff, msg->datalen);
    UnpackBytes(buff, msg->data, msg->datalen);
}

```

Network Multiplayer Games Report

GRADEMARK REPORT

FINAL GRADE

65/100

GENERAL COMMENTS

Instructor

UDP Server (30) 22
TCP Server (30) 19
Process Sync (20) 14

Work Independently (20) 10

Report is quite detailed and you have even included UML design diagrams to aid description of the approach that you took.

The code uses both UDP and TCP and I can see the work the evidence of the work you put in throughout the year.

As noted in your report the game is based on a tutorial but you have had to integrate this into your networking code.

Multi threading has also been used with semaphores being applied.

I know you had many problems during this term, but you have produced a good assignment.

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

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39
