

FinalPracReportRJ

by Robert Jamieson

Submission date: 02-May-2023 09:48PM (UTC+0100)

Submission ID: 204401416

File name: 24101_Robert_Jamieson_FinalPracReportRJ_499749_76487353.docx (785.17K)

Word count: 1642

Character count: 8795

The University of Wales Trinity Saint David
School of Applied Computing
Wales Institute of Science and Art

**Socket Programming and Concurrency combined
assessment – Task 2**

Socket Programming and Concurrency (ACCA6028)
Robert Iwan Jamieson (2004728)
Date of submission: 02.05.2023

Contents

List of figures	3
Chapter 1 Introduction	4
Chapter 2 Implemented application features, design decisions and how it adheres to best practise	5
2.1 Core features of the application.....	5
2.1.1 NetworkCore	5
2.1.2 Messages.....	5
2.2 Further features of the application	5
2.2.1 Entering an appropriate username	5
2.2.2 Players take turns and idle.....	7
2.3 Attempted features of the application.....	8
Chapter 3 Conclusion	11
Chapter 4 References	12

List of figures

Figure 1: Screenshot revealing client side code regarding a user entering a username. Source: [1].	6
Figure 2: Screenshot revealing server side code regarding a user entering a username. Validation checking. Modified and inspired from source: [1].	6
Figure 3: Screenshot revealing client side code regarding users taking turns and idling. Modified and inspired from source [1].	7
Figure 4: Screenshot revealing client side code regarding users taking turns and idling. Source: [1].	7
Figure 5: Screenshot revealing server side code regarding users taking turns and idling. Source: [1].	8
Figure 6: Screenshot revealing server side code regarding users taking turns and idling. Source: [1].	8
Figure 7: Screenshot revealing client side code of the failed implementation. Modified and inspired from source [1].	8
Figure 8: Screenshot revealing server side code of the failed implementation. Modified and inspired from source [1].	9
Figure 9: Screenshot revealing console output of the failed implementation. Code used for the output of this console was modified and inspired from source [1].	9

Chapter 1 Introduction

IMPORTANT NOTE: Much of the implementation is credited to Dr Tim Bashford and his example work found via Moodle: <https://moodle.uwtsd.ac.uk/course/view.php?id=6605> [1].

The implementation has original code but also modified/inspired code. Where appropriate, this is highlighted in the implementation itself and within the figures throughout this report.

In this report, a networked treasure game that has been developed will be discussed. It allows two clients to communicate back and forth with each other via a server and thus enable them to play versus one another. The main features, design decisions and how they adhere to best practise will be discussed. A failed implementation will also be briefly discussed.

Chapter 2 Implemented application features, design decisions and how it adheres to best practise

The implemented network application is a turn based treasure game. For the game to begin, there must be 2 players connected to the server. The aim of the game is to guess the correct column and row that the treasure is hiding behind. Players take it in turn to guess.

2.1 Core features of the application

The server side and client side of the implementation contains many similarities, including much of the same networking principles that enables the program to function.

2.1.1 NetworkCore

The “NetworkCore” files are used in both the server and client side. These files have the core functionality which enable players to connect to the server. When the server first comes into contact with a player, a socket is created so both the server and player can continually communicate with each other over a network [2].

2.1.2 Messages

The “Messages” files contain many different ways for the server to send or reply to a player’s input. These are used throughout the “MainGame” files to constantly respond to both players.

For example, the server responds by accepting or rejecting a user’s username. If it accepts it, the player will be allowed to play the game. However, if the username is rejected, the player will be told why their input was rejected and be told to e.g., enter a unique username.

Message objects are able to be sent back and forth from the server to the players. This is thanks to the use of serialization and deserialization functionality [3].

2.2 Further features of the application

2.2.1 Entering an appropriate username

Before being allowed to play the game, players must first enter a username. However, it has been designed so that users must enter a unique username which is 3 or more characters long.

Client side (ChatAppNetworkClient.cpp):

```
while (true)
{
    //cout << "CTEST1" << endl;
    //Start by sending a connection object. Requires a string for the username.
    string username = GetStringFromConsole("Please enter username");

    ConnectMessage* cm = new ConnectMessage();
    cm->SetUsername(username);

    SendNetworkMessage(cm);

    delete cm;

    Message* m = GetNetworkMessage();

    switch (m->GetType())
    {
        case MessageType::accept:
            cout << ((AcceptMessage*)m)->GetAcceptMessage() << endl;
    }
```

Figure 1: Screenshot revealing client side code regarding a user entering a username. Source: [1].

From the client's side, they are asked to enter a username which is then sent to the server which either accepts or rejects the username.

Server side (Source.cpp):

```
//MODIFIED / INSPIRED
if (totalLobbyVists > 0 && user1 == chatter->GetChatterName() || chatter->GetChatterName().length() < 3)
{
    //Checking whether the username entered is unique or not.
    if (totalLobbyVists > 0 && user1 == chatter->GetChatterName())
    {
        cout << "TEST101" << endl;
        Message* reply = new RejectMessage("Invalid input. Enter a unique name.");
        ci->SendString(reply->Serialise());
        delete reply;
        delete chatter;
    }
    else
    {
        //Checking whether the name is long enough.
        //MODIFIED / INSPIRED
        cout << "TEST102" << endl;
        Message* reply = new RejectMessage("Invalid input. Enter a name with 3 characters or more..");
        ci->SendString(reply->Serialise());
        delete reply;
        delete chatter;
    }
}
```

Figure 2: Screenshot revealing server side code regarding a user entering a username. Validation checking. Modified and inspired from source: [1].

After receiving the username, the server decides whether or not it is appropriate. As long as the username is 3 or more characters and is unique, it will be accepted – otherwise it is rejected. It is good practise for this validation to be on the server side because if it was on the client side then a devious player may be able to circumvent the validations.

Another aspect of this game that this reasoning can be applied to is when players must enter a row and column. They must enter a row and column within the boundaries, otherwise the server will reject their input and ask them to input a valid row and column.

Checking other factors, such as the win condition is also done on the server side. This ensures that there is no way that a client can cheat by altering the win condition.

2.2.2 Players take turns and idle

The game is implemented in such a way that players take their turn, then wait idly for the other player to take their turn and can only then take their next turn.

Client side (ChatAppNetworkClient.cpp)

```
//MODIFIED / INSPIRED
if (((ChatState*)m)->GetState() == yourturn)
{
    while (true)
    {
        _col = GetIntFromConsole("Enter column (0-8)");
        _row = GetIntFromConsole("Enter row (0-8)");
    }
}
```

Figure 3: Screenshot revealing client side code regarding users taking turns and idling. Modified and inspired from source [1].

```
else if (((ChatState*)m)->GetState() == opponentturn)
{
    //Request an update

    cout << "It is your opponent's turn - please wait!" << endl;
    //Send a request - the return should be a grid.
    SendNetworkMessage(new RequestUpdate());
    Message* tm = GetNetworkMessage();

    if (tm->GetType() == disconnect)
    {
        cout << ((DisconnectMessage*)tm)->GetDisconnectInfo() << endl;
        return new ChatState(win);
    }
}
```

Figure 4: Screenshot revealing client side code regarding users taking turns and idling. Source: [1].

From the Client's perspective, when it is their turn, they can input a row and column. However, when it is not their turn they are simply presented with a message informing them of this and they cannot interact with the game at all.

Server side (ChatAppNetwork.cpp)

```
if (tChatter == currentChatter)
{
    lock_guard<mutex> lock(turnSync);

    while (true)
    {
        //valid messages here are typemessage and disconnect.
        Message* m = tChatter->GetNetworkMessage();
    }
}
```

Figure 5: Screenshot revealing server side code regarding users taking turns and idling. Source: [1].

```
//Wait for opponent's turn to be over.  
unique_lock<mutex> lockT(turnSync);  
turnChange.wait(lockT, [&] {return tChatter == ttt->GetCurrentChatter() || opponent->GetConnectionStatus() == disconnected; });  
lockT.unlock();
```

Figure 6: Screenshot revealing server side code regarding users taking turns and idling. Source: [1].

The opponent on the client side is always waiting for the server side to confirm that it is its turn. The server has 2 threads which it must deal with. One of which is passed through and triggers the mutex to lock, whilst the other one is idle once it reaches the locked mutex [4].

Once a player's turn is over, the mutex is unlocked but again immediately locked once the previously idling thread passes through. For this particular game, this is best practise because it ensures that the threads are synchronized [4] during the game. It prevents 2 threads from processing the same crucial code at the same time. This means that players cannot play concurrently – they must make their turn one after the other, as is designed.

2.3 Attempted features of the application

The implementation was originally supposed to be a chatting application; however it was unsuccessful. It was therefore made into a treasure game. Despite of this, the failed application built on top of example code [1] that originally asked the user for their username. Instead, the application was now designed to ask the user to send a message to the other user, repeatedly.

Client side (ChatAppNetworkClient.cpp)

```
//MODIFIED / INSPIRED  
string userMessage = GetStringFromConsole("Your message: ");  
  
ConnectMessage* cm = new ConnectMessage();  
cm->SetUserMessage(userMessage);  
  
SendNetworkMessage(cm);  
  
delete cm;
```

Figure 7: Screenshot revealing client side code of the failed implementation. Modified and inspired from source [1].

Server side (ChatAppNetwork.cpp)

```
//MODIFIED / INSPIRED
//Chatters type to each other one after the other
if (tChatter == currentChatter)
{
    tChatter->SendNetworkMessage(new ChatterMessage("User1 message: " + opponent->GetChatterMessage()));
    tChatter->SendNetworkMessage(new ChatState(yourturn));
}
else
{
    tChatter->SendNetworkMessage(new ChatterMessage("User2 message: " + opponent->GetChatterMessage()));
    tChatter->SendNetworkMessage(new ChatState(opponentturn));
}
break;
```

Figure 8: Screenshot revealing server side code of the failed implementation. Modified and inspired from source [1].

The aim of this application was so that whenever one client inputs their message, it is received by the other client and then that client inputs their message and the process repeats itself. That chat would go on indefinitely until someone disconnected.

The outputs from the failed implementation can be seen below:

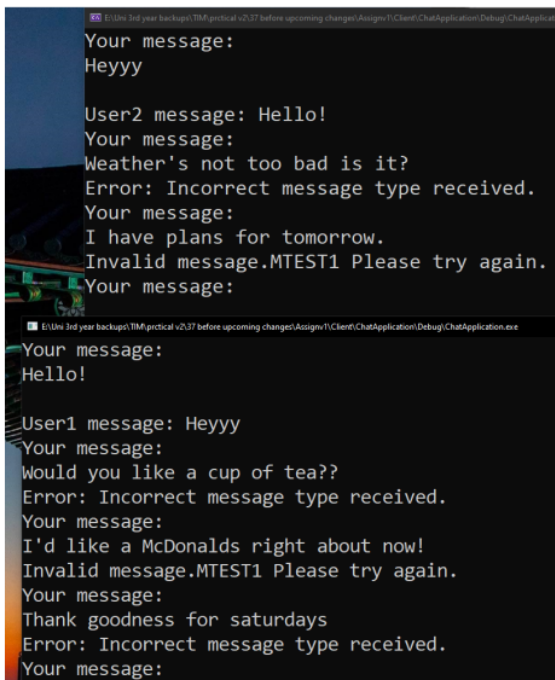


Figure 9: Screenshot revealing console output of the failed implementation. Code used for the output of this console was modified and inspired from source [1].

The design idea was that a message would be sent to the server from one client and that message would be immediately relayed to the other client. If this had worked, the next aspect

that would have been implemented is checking for inappropriate messages and thus either accepting or rejecting messages. The server would therefore have acted as a rule enforcer which would have been good practise – rather than checking for inappropriate language on the client side.

Unfortunately, the implementation of this was flawed and thus although the program executed, it did not execute as planned. Only the first message from both clients was successfully relayed to the other.

Chapter 3 Conclusion

In conclusion, a networked treasure game has been successfully developed. It adheres to best practise in some areas such as when validating a user input on the server rather than on the client side. This helps ensure no user cheats to win the game. The main features, design decisions and best practises for the implementation, including the main files of the game have been discussed. Furthermore, the features, design decisions and best practises have also been discussed for the failed implementation.

Chapter 4 References

- [1] T. Bashford, "L6 - Socket Programming and Concurrency (ACCA6028)," The University of Wales Trinity Saint David, [Online]. Available: <https://moodle.uwtsd.ac.uk/course/view.php?id=6605>. [Accessed 30 Apr 2023].
- [2] Oracle, "What Is a Socket?," Oracle, [Online]. Available: <https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html>. [Accessed 30 Apr 2023].
- [3] baeldung, "What Are Serialization and Deserialization in Programming?," baeldung, 24 Feb 2023. [Online]. Available: <https://www.baeldung.com/cs/serialization-deserialization>. [Accessed 30 Apr 2023].
- [4] cplusplus.com, "std::mutex::lock," cplusplus.com, [Online]. Available: <https://cplusplus.com/reference/mutex/mutex/lock/>. [Accessed 30 Apr 2023].

FinalPracReportRj

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

66/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12