

# Code\_report\_Leon\_Mathias

*by* Leon Mathias

---

**Submission date:** 02-May-2023 03:31PM (UTC+0100)

**Submission ID:** 204337563

**File name:** 24756\_Leon\_Mathias\_Code\_report\_Leon\_Mathias\_499749\_2111968856.docx (197.63K)

**Word count:** 1203

**Character count:** 5784

Leon Mathias

2013015

Computing (Software engineering)

Student University of Wales Trinity St David,  
Swansea, Wales

L6 - Socket Programming and Concurrency  
(ACCA6028)

Date of submission:

## Introduction

This report explains the features and design decisions of a networked Connect 4 game and how it adheres to best practice. The program allows two clients to play a game of Connect 4 with against each other by connecting to a server. The code used to create this networked connect 4 game is a combination of mine and Tim Bashford [1] code as well as code adapted from class work.

## Server and client connection

The code that can be seen in Figure 1 is the main on the server. This code initializes an `Inet4Address` object to listen on port 50018, creates a `ServerConnection` object to manage the incoming connections, and starts the server. A loop then accepts incoming client connections and starts a new thread to handle each client with the `ClientHandler` function. Starting a new thread to handle each client connection allows the server to handle multiple clients concurrently. Without this approach, the server would have to handle one client at a time. Each thread will handle one client connection, and the server can continue to accept new connections while existing threads handle their clients, allowing the server to handle many clients at once.

```
int main(int argc, char** argv)
{
    Inet4Address* address = new Inet4Address(50018);
    ServerConnection* server = new ServerConnection(address);

    server->StartServer(100);

    while (true)
    {
        ConnectionInstance* ci = server->AcceptClient();
        thread(ClientHandler, ci).detach();
    }

    system("pause");
}
```

Figure 1: Start server code

Figure 2 displays the main function of the client. It begins by creating an `Inet4Address` object with the server's IP address and port number and then creates a `ClientConnection` object using this address. The code then uses the `ClientConnection` object to connect to the server, which returns a `ConnectionInstance` object representing the connection between the client and the server. A `Connect4NetworkClient` object is then created using the `ConnectionInstance` object, and the `StartGame()` function is called to begin the game.

```

int main(int argc, char** argv)
{
    char ip[] = "127.0.0.1";

    INet4Address* address = new INet4Address(ip, 50018);

    ClientConnection* conn = new ClientConnection(address);

    ConnectionInstance* ci = conn->ConnectToServer();

    Connect4NetworkClient* nc = new Connect4NetworkClient(ci);

    nc->StartGame();

    system("pause");
}

```

Figure 2: Start client code

As seen in Figure 3 the socket created uses TCP which is used as it is a reliable protocol that ensures delivery of data. This is very important in a turn-based game such as Connect 4, this is because the game requires real-time communication between the players, as each move affects the available options for subsequent moves. By using TCP, the server can ensure that each player's moves are accurately transmitted to the other player. If a packet of data is lost or corrupted during transmission, TCP will automatically retransmit the data to ensure that it's received correctly. This helps prevent errors and ensures the game state is consistent between both players in the game, to avoid confusion or frustration.

```

void Connection::CreateSocket()
{
    sockfd = socket(address->GetSockaddrRef()->sin_family, SOCK_STREAM, IPPROTO_TCP);

    if (sockfd <= 0)
    {
        throw WSAGetLastError();
    }
}

```

Figure 3: Create socket

## Starting the game

In order for the game to begin 2 players must be connected and placed into the game. This is because Connect 4 is a two-player game one player should never be playing alone. But to allow a client to play, an accept message must be sent to each client. For this to work once the client has connected to the server, they are prompted to enter a username and a colour for the console. The username and colour are then sent back to the server, which checks to see if the username is 3 characters and unique in this instance and also checking if the colours sent are valid this can be seen in Figure 4. If any of these are true, then the server sends the client a reject message and prompts them to enter these details again. Only once valid details are entered will an accept message be sent allowing the client to join the game and the details will be stored in order to be used for displaying their name and their opponents name. Also displayed in Figure 4 is if the lobby is full as two clients are playing a reject message will be sent and the client will be disconnected from the server as a lobby to handle more than two clients has not been implemented.

```

if (player->GetPlayerName().length() != 3)
{
    Message* reply = new RejectMessage("Username must be 3 letters!");
    ci->SendString(reply->Serialize());
    delete reply;
    delete player;
}
else if (!regex_match(player->GetColour(), regex("[0-9A-F]{2}$")))
{
    Message* reply = new RejectMessage("Choose a valid colour");
    ci->SendString(reply->Serialize());
    delete reply;
    delete player;
}
else if (Lobby::Inst().GetGame()->IsUsernameTaken(player->GetPlayerName()))
{
    Message* reply = new RejectMessage("Sorry, the username is already taken.");
    ci->SendString(reply->Serialize());
    delete reply;
    delete player;
}
else if (!Lobby::Inst().GetGame()->AddPlayer(player))
{
    Message* reply = new RejectMessage("Sorry the game is full. Please try again later!");
    ci->SendString(reply->Serialize());
    delete reply;
    delete player;

    ci->Disconnect();
    break;
}

```

Figure 4: Detail Validation

As it is possible that a client will disconnect whilst the game is being played it is important to handle this. As this can cause the program to crash therefore the exception is to be caught and dealt with. It is also important as this is a two-player game, the other player must be given the win and informed of the disconnect. As seen in Figure 5 if a client disconnects on their turn the server swaps turn. Otherwise, the client still connected will be waiting for their turn which will never happen.

```

if (m == nullptr || m->GetType() == disconnect)
{
    tPlayer->SetDisconnected();
    HandleDisconnect(opponent);
}

void Connect4Network::HandleDisconnect(ConnectedPlayer* opponent)
{
    while (opponent->GetConnectionStatus() != disconnected)
    {
        if (c4->GetCurrentPlayer() != opponent)
        {
            c4->SwapTurn();
        }

        turnChange.notify_one();
    }
}

```

Figure 5: Handle disconnect

The game logic such as checking for victory and placing a counter, is held on the server rather than the client. The reasons for this are because if it was handled on the client side it becomes possible for users to manipulate the game state, leading to cheating or exploitation. It also ensures all clients receive the same game state, for example, that the board is displayed the same for both players. The game is only required to be updated once on the server rather than for each client.

## Mutex

To ensure only one player can make their move at a time, mutex is used to lock and unlock access to the turn of the game being played. As seen in Figure 6 when a player has their turn, the `lock_guard<mutex> lock(turnSync)` is used to lock the mutex. This prevents other threads from acquiring the lock and executing the code that accesses the turn of the game being played. Once a player has made their move, the `turnChange.notify_one()` function is called to signal the other thread that is waiting on the mutex that the turn has changed. This allows the other player to make their move. This is done either when the player has disconnected or made a valid move.

```
if (tPlayer == currentPlayer)
{
    lock_guard<mutex> lock(turnSync);
    while (true)
    {
        Message* m = tPlayer->GetNetworkMessage();
        if (m == nullptr || m->GetType() == disconnect)
        {
            c4->SwapTurn();
            turnChange.notify_one();
            return;
        }
        else if (m->GetType() == makemove)
        {
            if (c4->MakeMove(((MakeMove*)m)->GetCol()))
            {
                tPlayer->SendNetworkMessage(new BoardUpdate(c4->GetBoard()));
                turnChange.notify_one();
                break;
            }
            else
            {
                tPlayer->SendNetworkMessage(new RejectMessage("Please make a valid move."));
            }
        }
        else
        {
            tPlayer->SendNetworkMessage(new RejectMessage("Invalid message. Please try again."));
        }
    }
}
```

Figure 6: Lock Mutex

It is important to eventually unlock the mutex or waiting threads will be stuck forever. As can be seen in Figure 7, Once the other player has made their move or disconnected, the mutex is unlocked, allowing the current player to make their move. This is another reason why it is important to handle a client disconnecting.

```
unique_lock<mutex> lockT(turnSync);
turnChange.wait(lockT, [&] {return tPlayer == c4->GetCurrentPlayer() || opponent->GetConnectionStatus() == disconnected; });
lockT.unlock();
```

Figure 7: Unlock Mutex

## Use of messages

To send and retrieve any data or instructions from both the client and the server a message class was created which defined the different types of messages that could be sent. These include "connect", "disconnect", "accept", "reject", "requestupdate", "gameboard", "makemove", "gamestate", "opponentName", and "colour" some of which have been mentioned already such as reject message. All of these have their own class which are used to set the message and then get the

message from either side. An example is the “makemove” message which is used as seen in Figure 8 where the client sends a MakeMove to the server.

```
cout << "Please select column (1-7)" << endl;
int col = GetIntFromConsole();

SendNetworkMessage(new MakeMove(col));
Message* nm = GetNetworkMessage();
```

Figure 8: Send MakeMove

As seen in Figure 9 the server checks the message and as the type is makemove, the server then checks to ensure the move is valid. If it is, then it sends an updated version of the board to the player and changes turn.

```
else if (m->GetType() == makemove)
{
    if (c4->MakeMove(((MakeMove*)m)->GetCol()))
    {
        tPlayer->SendNetworkMessage(new BoardUpdate(c4->GetBoard()));
        turnChange.notify_one();
        break;
    }
    else
    {
        tPlayer->SendNetworkMessage(new RejectMessage("Please make a valid move."));
    }
}
```

Figure 9: Server receive MakeMove

## Bibliography

[1] Tim Bashford, “L6 - Socket Programming and Concurrency (ACCA6028)”, Url: <https://moodle.uwtsd.ac.uk/course/view.php?id=6605>

# Code\_report\_Leon\_Mathias

## GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

54/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6