

SP+C Final Report

by James Millns

FILE	6668_JAMES_MILLNS_SP_C_FINAL_REPORT_349477_1978227345.DOCX (1.86M)		
TIME SUBMITTED	17-MAY-2021 09:25PM (UTC+0100)	WORD COUNT	4971
SUBMISSION ID	153630295	CHARACTER COUNT	26203

Socket Programming and Concurrency

Name

James Edward Millns

Student No

1708455

Email

1708455@student.uwtsd.ac.uk

Course

BSC (Hons) Software Engineering

Module Name

Socket Programming and
Concurrency

Module Code

ACCA6028

Lecturer

Dr Tim Bashford

Table of Contents

Table of Contents	2
Table of Figures	3
1. Investigation – Task 1	5
1.1 Introduction	5
1.2 TCP	6
1.3 Transfer of Data	9
1.4 Synchronicity and A-Synchronicity	10
1.5 Interoperability.....	12
1.6 Conclusion	13
2. Implementation – Task 2	14
2.1 Requirements.....	14
2.2 Networked Application Files.....	15
2.3 Connect Four	16
2.4 Othello.....	25
2.5 Dots and Boxes.....	33
2.6 Conclusion	42
References	43

Table of Figures

Figure 1: TCP Header Structure [5].....	6
Figure 2: Three-Way Handshake TCP Connection Establishment [5].....	8
Figure 5: Shared NetworkCore files	15
Figure 6: Shared Message Files	15
Figure 3: Connect Four Folders	16
Figure 7: Connect Four Make Move Method.....	16
Figure 8: Connect Four Check for vertical win.	17
Figure 9: Connect Four check for horizontal win.	18
Figure 10: Connect Four check for forward diagonal win	19
Figure 11: Connect Four check for backwards diagonal win.	20
Figure 12: Connect Four Welcome Screen	21
Figure 13: Connect Four two players connected	21
Figure 14: Connect Four first two moves.....	22
Figure 15: Connect Four horizontal win.....	22
Figure 16: Connect Four vertical win.....	23
Figure 17: Connect Four forwards diagonal win.	23
Figure 18: Connect Four backwards diagonal win.	23
Figure 19: Connect Four draw.....	24
Figure 20: Othello folders	25
Figure 21: Othello Board Setup.....	25
Figure 22: Othello Make Move	26
Figure 23: Othello Check for neighbouring pieces	27
Figure 24: Othello flip pieces.....	28
Figure 25: Othello check for valid moves	29
Figure 26: Othello check for victory	30
Figure 27: Othello Welcome message	31
Figure 28: Othello two players connected	31
Figure 29: Othello first two moves	32
Figure 30: Othello completed game	32
Figure 31: Dots and Boxes folders	33
Figure 32: Dots and Boxes board setup	34
Figure 33: Dots and Boxes make move.	35

Figure 34: Dots and Boxes check for completed squares.	36
Figure 35: Dots and Boxes check for victory.	37
Figure 36: Dots and Boxes welcome message.	38
Figure 37: Dots and Boxes two players connected.	38
Figure 38: Dots and Boxes first two moves.	39
Figure 39: Dots and Boxes box being claimed	40
Figure 40: Dots and Boxes player 1 wins	41
Figure 41: Dots and Boxes draw	41

1. Investigation – Task 1

1.1 Introduction

This research report aims to investigate how communication over networks is established and maintained. The Transmission Control Protocol (TCP) and Internet Protocol (IP) are known to be the main protocols used in network communication and will be explored to find the methods used to guarantee delivery of data over a network.

The mechanics behind the transfer of data when regarding size will also be investigated looking at the Maximum Transfer Unit (MTU) of the Protocol Data Units (PDU) at their relative positions in the OSI model.

Synchronicity and Asynchronicity in network communication will also be investigated, and the applications in which they may appropriate for will be explored.

Finally, the meaning of interoperability within networks will be examined to find how it affects scalability and how to best achieve it in development.

1.2 TCP

The Transmission Control Protocol (TCP) is a set of standards that deal with the exchange of data between connected devices in a computer network. The protocol was published in a research paper in 1974 by Robert E. Kahn and Vinton G. Cerf [1] and was later standardised in 1981 in RFC 793 [2]. Since its standardisation, the protocol has received many extensions and improvements, but the core protocol has remained the same [3].

TCP operates by allowing two nodes in a computer network to create a consistent connection that allows for back-and-forth transmission of data. In the event of data loss through transmission, the protocol allows for automatic detection and correction, making TCP a reliable protocol [3]–[5].

TCP breaks down the information that needs to be communicated between two computer network nodes into segments which are called packets. Packets are the basic units of transmission and encapsulate the data within a digital envelope that has information of where it is going, where it is coming from, and what it should be containing alongside some other information [6]–[8]. The information about the transmission data is stored in a TCP header as shown in Figure 1.

TCP Header				
Bits	0-15			16-31
0	Source port			Destination port
32	Sequence number			
64	Acknowledgment number			
96	Offset	Reserved	Flags	Window size
128	Checksum			Urgent pointer
160	Options			

Figure 1: TCP Header Structure [5].

The reason for the information being broken down into packets is due to the issue of data corruption during transmission. This occurs when a binary bit is affected by the transmission process and flips the state, thereby not being the same data that was sent [5], [9], [10].

If the data was sent as one large blob, then if this corruption occurred, the whole blob would need to be sent again. Packets reduce this problem as if there is corruption, only a small amount of data needs to be resent. Corrupted data is detected by comparing the checksum value of the data when it was sent to the checksum value of the data on receipt, if there is a discrepancy then it means there was an error in transmission and a request for a resend is sent [3], [11]–[13].

For a TCP connection to be established, there needs to be two endpoints in the network that have unique IP addresses which serve as a unique identifier for the endpoint [3].

TCP connections are established through what is called a “three-way handshake”. This method is a solution to a problem of not being sure if the recipient has received the sent data. This issue is known as the “Two General’s problem”, whereby two generals who are in different geological positions are trying to synchronise and attack on their enemies [14]. The problem plays out in the following steps:

- General A wishes to attack the enemy at 12pm in a week, so sends a messenger to General B with the plans [14].
- General B receives the plans from the messenger, and wishes to confirm the receipt of said plans, otherwise General A will not know if the attack is going ahead, so General B sends a messenger out to confirm [14].
- General A receives the confirmation but needs to inform General B that they have received the confirmation, otherwise General B will not know that they acknowledged the confirmation, so send a messenger out to confirm the confirmation [14].
- These steps repeat indefinitely as there is no number of steps that can guarantee that each part knows with 100% certainty that the other is onboard with the plans [14].

This problem exists in the same way in network connections and is solved by the “three-way handshake”. This is where the client of the connection sends a sync request to the server, the server responds with a sync-acknowledgment, and the client sends an acknowledgment-acknowledgment. This is the end of the three-way handshake where the client assumes that the final acknowledgement has been received and proceeds with the connection [3]–[5].

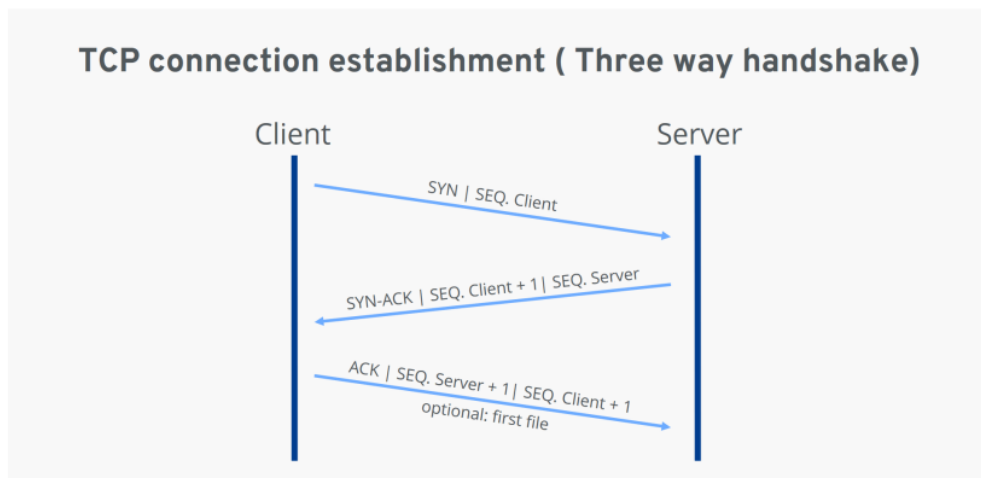


Figure 2: Three-Way Handshake TCP Connection Establishment [5].

1.3 Transfer of Data

The Maximum Transmission Unit (MTU) refers to the size of the largest Protocol Data Unit (PDU) that a network can communicate. When the MTU is larger, it means that the data being sent is cut into larger segments or packets, meaning that there is less overall overhead as there are less packets being sent through the network. This has the flipside of being more vulnerable to transmission corruption as larger packets will need to be sent in the event of corruption. Smaller MTU can reduce the delay on the network as corrupted packets are smaller and faster to resend, however they involve more overhead as there are more packets (making it less efficient and may increase network traffic) [15]–[17].

Even with the issue of larger MTU packets being more susceptible to corruption and the negatives of retransmission, there is still an overall positive effect on the TCP performance between two nodes on a network as there is higher efficiency meaning there is higher throughput of useful data [15], [16], [18].

The Open Systems Interconnection Model (OSI model) is a description of the characteristics and standards of communication functions between computing systems. The OSI model is broken down into seven layers: Application, Presentation, Session, Transfer, Network, Data-link and Physical [15]–[17].

Each layer has a PDU, the physical layer uses bits that are transmitted via hardware, the data-link layer uses series of bits (frames), the network layer uses packets, the transport layer then uses segments for the TCP, the session layer passes data to the network connection, the presentation then formats the data, and the application shows are transmits data from a software application [15]–[17].

Internet Protocol Fragmentation is a protocol that separates packets into smaller chunks (fragments) so they can be passed through a network link which has a smaller MTU than the initial packet size. Fragmented packets are then reassembled by the receiving host to then pass it to a higher protocol layer. This reassembly may occur in the receiving host or in an intermediate router like a Network Address Translation (NAT) which needs to reassemble the packets to translate the data [15]–[17].

1.4 Synchronicity and Asynchronicity

Synchronous data transmission through a network is a method of data transfer that streams a continuous signal of data which are accompanied by timing signals. The timing signals allow the transmitter and receiver to synchronise with each other [19]–[21].

Asynchronous data transmission through a network is a method of data transfer that sends data from the transmitter to the receiver using a flow control method. Instead of using timing signals, data is sent in chunks of a predefined size, usually 8 bits sandwiched by a start and stop bit, totalling in 10 bits [19]–[21].

Synchronous transmission requires users to wait until the data they are sending has been fully transferred before they can receive a response from the server, while Asynchronous transmission does not have this downside, allowing users to receive responses from the server at any time [19]–[21].

Synchronous transmission sends blocks of characters at a time through high-speed transmission lines while asynchronous transmission transmits the information character-by-character, making synchronous transmission fast but costly and asynchronous transmission slow but economical [19]–[21].

Synchronous transmission is suitable for network protocols that need to transfer large amounts of data seen in examples such as- Ethernet, SONET and Token-Ring [19]–[21].

Asynchronous transmission is suitable for network protocols that require instant responses like telephone lines and instant messaging [19]–[21].

When considering networking with external web services, there may be a significant amount of waiting involved due to service latency or network latency. Traditionally, web applications handle Hyper Text Transfer Protocol (HTTP) requests with application threads. Each concurrent request is assigned to an individual application thread (e.g. 20 concurrent requests using 20 threads), making the memory overhead larger and so makes it a poorly scalable protocol [21], [22].

To combat this issue, I/O dispatcher models are used, which instead offload the network requests onto a small number of worker threads. These worker threads use a select system call to handle a larger number of sockets more efficiently. Application threads are reserved to run application code, allowing for scenarios where they may have many concurrent requests which are divided into service calls and application execution where the small number worker threads deal with all the service call requests, and each application execution request is handled by an individual application thread. This model scales better than the previous model [21], [22].

1.5 Interoperability

Interoperability describes the ability for different programs or networks to operate and communicate with each other without the requirement for additional interfaces or tools. TCP/IP is a highly interoperable standard that is available to all operating systems, computers and networks [23]–[25].

Interoperability between systems requires the ability to trade, decipher and represent shared data in a way each system understands. This can be achieved with syntactic interoperability, whereby each system adopts a common data format and data structure protocols. This is then followed by semantic interoperability, whereby the addition of metadata connects data elements to a shared, pre-established standard [23], [26], [27].

The difference between integration and interoperability, is how the systems communicate. Integration is usually involving the combination of multiple applications into one uninterrupted system that sometimes requires the use of “middleware”. Interoperability is instead the immediate exchange of data between systems that communicate with each other directly in the same vocabulary, allowing it to present data as it was received [23], [26], [28].

Network Interoperability can be achieved by either requiring both systems or networks to conform to a shared protocol or standard, effectively making them speak the same language, or by providing an intermediate gateway that translates the different protocols or standards used [23], [25], [26], [28].

1.6 Conclusion

In conclusion, TCP operates by breaking down transmitted data into smaller packets that are sent over a network with metadata that guarantees the delivery by detecting whether any packets have been missed or corrupted in transit [4]. TCP uses a “three-way handshake” to establish and terminate connection sessions, which is a compromised solution to the “Two Generals’ Problem” [14].

The size of MTU affects the transfer of data, whereby larger MTU allow for more efficient and faster data transmission due to the reduction in metadata, but being more vulnerable to corruption slow down, while smaller MTU makes the transmission slower and less efficient due to the increase in metadata, while being more robust to corruption as the packets are smaller [16].

Synchronicity in network protocols refers to a continuous stream of data transmission between sender and receiver which is accompanied by timing signals which is used in scenarios like Ethernet, SONET and Token Ring [20].

A-synchronicity in network protocols refers to a non-continuous transfer of data that is controlled by flow control protocol. A-synchronicity is used in scenarios like telecommunications and instant messaging [20].

Interoperability in networked applications refers to the ability for two network entities to communicate with each other without the requirement for additional interfaces or tools. TCP/IP being a highly interoperable protocol that is available to all operating systems, computers and networks [25].

2. Implementation – Task 2

2.1 Requirements

Creating a networked application that enables two clients to communicate in order to play a multiplayer game requires an infrastructure that establishes a connection over a network and a messaging system that can be used for communication. There also needs to be game logic for the desired game. It is the aim of this project to implement the logic for three different board games, Connect Four, Othello, and Dots and Boxes. The student could not decide between the three, so chose to make all of them instead.

Connect Four Requirements [29]:

- Players can take turns placing disk into columns.
- Game only allows player to make legal moves.
- Game registers when win condition has been met.
- Game registers when game ends in draw.
- Console displays updated board every move to each player.

Othello Requirements [30]:

- Players can take turns placing disks onto board.
- Game only allows players to make legal moves.
- Game flips pieces as per Othello rules
- Game skips players turn if there are no legal moves to be made for them.
- Game registers when win condition has been met.
- Game registers when game ends in draw.
- Console displays updated board every move to each player.

Dots and Boxes Requirements [31]:

- Players can take turn drawing lines between dots.
- Game only allows players to make legal moves.
- Game claims boxes when a square is completed.
- Game gives player who just claimed a box an extra turn.
- Game registers when win condition has been met.
- Game registers when game ends in draw.
- Console displays updated board every move to each player.

2.2 Networked Application Files

To make the applications into multiplayer games, there needed to be infrastructure for communication between two clients. For each game, a solution was made for a server and another solution for a client. Both solutions would have the same communication infrastructure, where they would send contextual messages once a connection had been established.

The Messages and NetworkCore files shown in Figures 5 and 6 were created by Dr. T. Bashford and were given with permission to implement the functionality of a networked application.

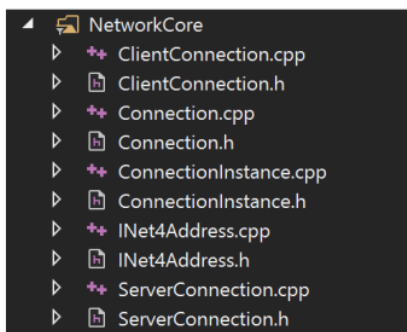


Figure 3: Shared NetworkCore files

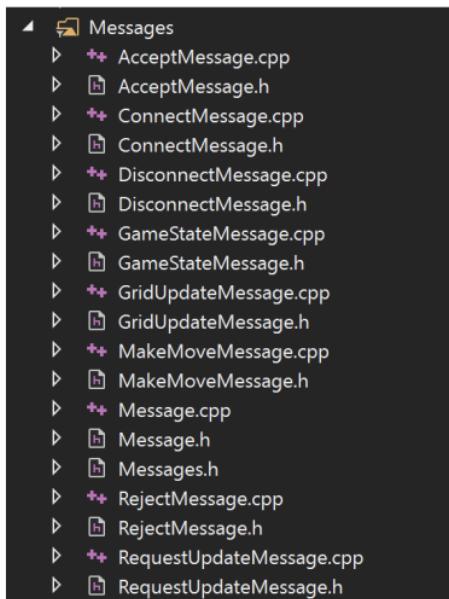


Figure 4: Shared Message Files

2.3 Connect Four

The connect four networked game is separated into a client and a server, where the server contains the game logic, and communicated the changes in game states, while the clients present the user-interface to the user and send moves to the server.

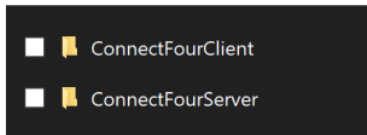


Figure 5: Connect Four Folders

The make move method takes a column, checks to make sure it is in bounds, checks to make sure the column is not already full, and then places the chip into the lowest row in that column.

```
bool ConnectFour::MakeMove(int column)
{
    //bounds check
    if (column < 0 || column >= COLS)
    {
        return false;
    }

    //full column check
    if (board[column].back() != SlotState::empty)
    {
        return false;
    }

    //Put chip into lowest row in column
    for (int row = 0; row < ROWS; row++)
    {
        if (board[column][row] == SlotState::empty)
        {
            board[column][row] = playerTurn;
            break;
        }
    }

    SwapTurn();

    return true;
}
```

Figure 6: Connect Four Make Move Method

The check for victory method has four possible win conditions to check. First it checks for a vertical connect four as shown in Figure 8, then checks for a horizontal connect four as shown in Figure 9.

```
// check columns
for (auto col : board)
{
    connected = 0;
    ss = SlotState::empty;
    for (auto row : col)
    {
        if (row == SlotState::empty)
        {
            ss = row;
            connected = 0;
        }
        else
        {
            if (ss == row)
            {
                connected++;
                if (connected == 4)
                {
                    return GetPlayer(ss);
                }
            }
            else
            {
                connected = 1;
            }
            ss = row;
        }
    }
}
```

Figure 7: Connect Four Check for vertical win.

```
// check rows
for (int row = 0; row < ROWS; row++)
{
    connected = 0;
    ss = SlotState::empty;
    for (int col = 0; col < COLS; col++)
    {
        if (board[col][row] == SlotState::empty)
        {
            ss = board[col][row];
            connected = 0;
        }
        else
        {
            if (ss == board[col][row])
            {
                connected++;
                if (connected == 4)
                {
                    return GetPlayer(ss);
                }
            }
            else
            {
                connected = 1;
            }
            ss = board[col][row];
        }
    }
}
```

Figure 8: Connect Four check for horizontal win.

To check for a diagonal win, there is first a method which sets the initial row and column position of the check pointer. Once this has been set, it uses a loop much like the vertical and horizontal checks but is instead increments two values.

This logic for checking diagonals is used for both forward and backward diagonals, where for backward diagonals, the initial starting points for each possible diagonal that could contain a connect four is different, and instead of incrementing the column index, it is instead decremented, as shown in Figure 11.

```
// check forward diagonals
for (int diagonals = 0; diagonals < 6; diagonals++)
{
    connected = 0;
    ss = SlotState::empty;
    int startCol, startRow;
    SetStartRowAndCol(diagonals, startCol, startRow, true);
    for (int col = startCol, row = startRow; row < ROWS && col < COLS; row++, col++)
    {
        if (board[col][row] == SlotState::empty)
        {
            ss = board[col][row];
            connected = 0;
        }
        else
        {
            if (ss == board[col][row])
            {
                connected++;
                if (connected == 4)
                {
                    return GetPlayer(ss);
                }
            }
            else
            {
                connected = 1;
            }
            ss = board[col][row];
        }
    }
}
```

Figure 9: Connect Four check for forward diagonal win

```
// check backward diagonals
for (int diagonals = 0; diagonals < 6; diagonals++)
{
    connected = 0;
    ss = SlotState::empty;
    int startCol, startRow;
    SetStartRowAndCol(diagonals, startCol, startRow, false);
    for (int col = startCol, row = startRow; col >= 0 && row < ROWS; row++, col--)
    {
        if (board[col][row] == SlotState::empty)
        {
            ss = board[col][row];
            connected = 0;
        }
        else
        {
            if (ss == board[col][row])
            {
                connected++;
                if (connected == 4)
                {
                    return GetPlayer(ss);
                }
            }
            else
            {
                connected = 1;
            }
            ss = board[col][row];
        }
    }
}
```

Figure 10: Connect Four check for backwards diagonal win.

When the client connects to the server, the user is asked to enter their name. When they do, they are presented with a list of the rules, with a display of the empty starting board, as shown in figure 12.

```

Please enter username
tim
Welcome
Welcome to Connect Four
Rules:
1. Players take turns dropping disks into the columns
2. Players cannot drop a disk into a column that is full
3. The aim is to get four of your disks in a row, column or diagonal
4. The player who connects four disks wins

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

 1 2 3 4 5 6 7

```

Figure 11: Connect Four Welcome Screen

When two players connect to the lobby and enter their names, the game starts, and the player who first connected is designated player 1, as shown in Figure 13.

```

Please enter username
james
Welcome
Welcome to Connect Four
Rules:
1. Players take turns dropping disks into the columns
2. Players cannot drop a disk into a column that is full
3. The aim is to get four of your disks in a row, column or diagonal
4. The player who connects four disks wins

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

 1 2 3 4 5 6 7
Please select column (1-7)

^ Please enter username
tim
Welcome
Welcome to Connect Four
Rules:
1. Players take turns dropping disks into the columns
2. Players cannot drop a disk into a column that is full
3. The aim is to get four of your disks in a row, column or diagonal
4. The player who connects four disks wins

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

 1 2 3 4 5 6 7
It is your opponent's turn - please wait!

```

Figure 12: Connect Four two players connected

Players then take turns making moves, dropping disks into the columns they choose until a win condition is met, or there are no valid moves left. Figure 14 shows the first two moves made by the connected players.

```

Please enter username
james
Welcome
Welcome to Connect Four
Rules:
1. Players take turns dropping disks into the columns
2. Players cannot drop a disk into a column that is full
3. The aim is to get four of your disks in a row, column or diagonal
4. The player who connects four disks wins

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

  1 2 3 4 5 6 7
Please select column (1-7)
4

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

  1 2 3 4 5 6 7
It is your opponent's turn - please wait!

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

  1 2 3 4 5 6 7
Please select column (1-7)
4

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

  1 2 3 4 5 6 7
Please select column (1-7)
4

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

  1 2 3 4 5 6 7
It is your opponent's turn - please wait!

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

  1 2 3 4 5 6 7
Please select column (1-7)
4

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

  1 2 3 4 5 6 7
It is your opponent's turn - please wait!
  
```

Figure 13: Connect Four first two moves

Figure 15 shows the case where a player wins by connected four in a row on the horizontal.

```

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

  1 2 3 4 5 6 7
Congratulations, you win!
Press any key to continue . . .

  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o
  o o o o o o o

  1 2 3 4 5 6 7
Sorry, you lose.
Press any key to continue . . .
  
```

Figure 14: Connect Four horizontal win.

Figure 16 shows the case where a player wins by connecting four disks on the vertical axis.



Figure 15: Connect Four vertical win.

Figure 17 shows the case where a player wins by connecting four disks on the forward diagonal axis.

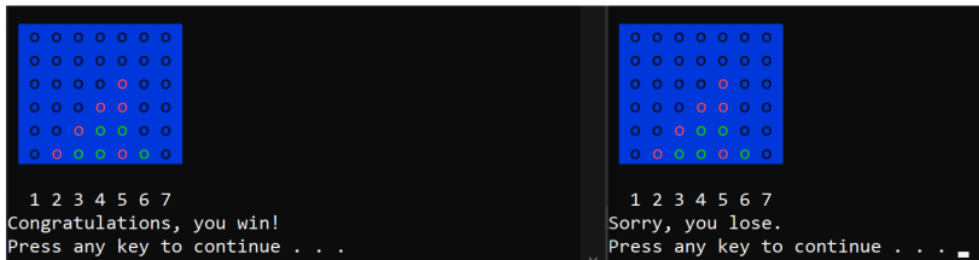


Figure 16: Connect Four forwards diagonal win.

Figure 18 shows the case where a player wins by connecting four disks on the backwards diagonal axis.

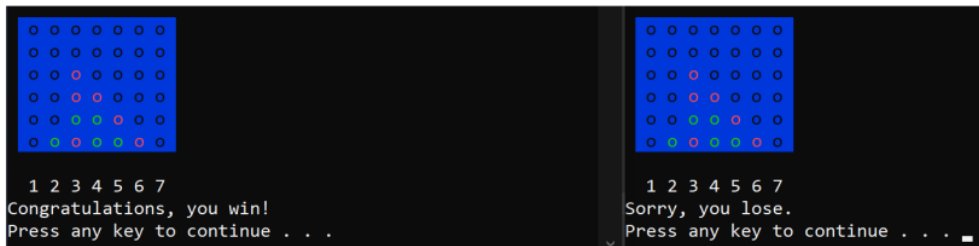


Figure 17: Connect Four backwards diagonal win.

Figure 19 shows the case where there have been no win conditions met but there are no remaining valid moves, making the game end in a draw.

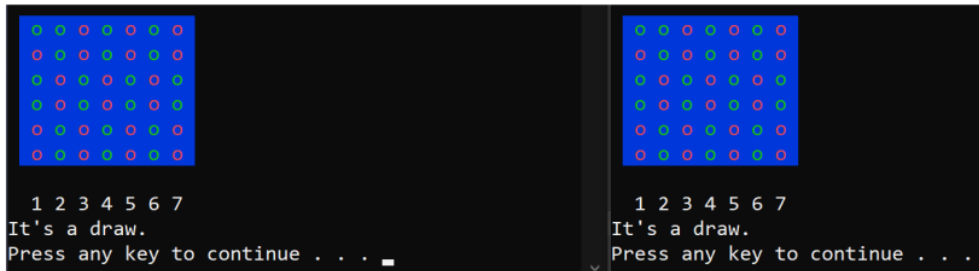


Figure 18: Connect Four draw

2.4 Othello

As with the connect four game, the Othello game is divided into a client and server. They operate in the same way where the server contains the game logic, and the client presents the user interface to the user.

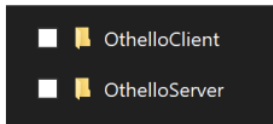


Figure 19: Othello folders

When the game is initialised, each square in the board is set to empty, with exception of the four central boxes, which are set to two player 1 chips and two player 2 chips in an alternating pattern.

```
Othello::Othello(ConnectedPlayer* p1, ConnectedPlayer* p2)
{
    for (int r = 0; r < ROWS; r++)
    {
        vector<SquareState> row;

        for (int c = 0; c < COLS; c++)
        {
            if ((c == 3 || c == 4) && (r == 3 || r == 4))
            {
                if (c == r)
                {
                    row.push_back(SquareState::player1);
                }
                else
                {
                    row.push_back(SquareState::player2);
                }
            }
            else
            {
                row.push_back(SquareState::empty);
            }
        }

        board.push_back(row);
    }

    this->p1 = p1;
    this->p2 = p2;

    playerTurn = SquareState::player1;
}
```

Figure 20: Othello Board Setup

The make move method first checks if the selected position is valid (is in bounds, is empty and has a neighbouring opponent piece).

```
bool Othello::MakeMove(int column, int row)
{
    //Bounds check
    if (column < 0 || column >= COLS)
    {
        return false;
    }

    if (row < 0 || row >= ROWS)
    {
        return false;
    }

    //Emptiness check
    if (board[row][column] != SquareState::empty)
    {
        return false;
    }

    // Check surrounding squares
    if (!CheckForNeighbouringPiece(row, column))
    {
        return false;
    }

    board[row][column] = playerTurn;
    flipPieces(row, column);
    SwapTurn();

    return true;
}
```

Figure 21: Othello Make Move

The method that checks for neighbouring pieces looks at the eight surrounding positions and returns true if it finds an opponent piece.

```
bool Othello::CheckForNeighbouringPiece(int row, int column)
{
    // set the bool to false to start
    bool hasAdjacentOpponentPiece = false;
    //set the opponent
    SquareState opponent;
    switch (playerTurn)
    {
        case player1:
            opponent = player2;
            break;
        case player2:
            opponent = player1;
            break;
        default:
            break;
    }

    // check every neighbour for an opponent piece
    // north square
    if (row + 1 < ROWS)
    {
        if (board[row + 1][column] == opponent)
        {
            hasAdjacentOpponentPiece = true;
        }
    }
    // north east square
    if ((row + 1 < ROWS)&&(column + 1 < COLS)) { ... }
    // east square
    if (column + 1 < COLS) { ... }
    // south east square
    if ((row - 1 >= 0) && (column + 1 < COLS)) { ... }
    // south square
    if (row - 1 >= 0) { ... }
    // south west square
    if ((row - 1 >= 0)&&(column - 1 >= 0)) { ... }
    // west square
    if (column - 1 >= 0) { ... }
    // north west square
    if ((row + 1 < ROWS)&&(column - 1 >= 0)) { ... }

    return hasAdjacentOpponentPiece;
}
```

Figure 22: Othello Check for neighbouring pieces

The flip pieces method shown in figure 24 checks straight lines in all eight directions. If it finds a piece that is of the current player in any of the directions, then it changes a flip trigger Boolean value that then loops through that direction, flipping each piece along the way until it reaches the found piece again.

```

void Othello::flipPieces(int row, int column)
{
    bool flipTrigger;

    // north
    if (row + 1 < ROWS)
    {
        flipTrigger = false;

        // loop through the squares in a straight line
        for (int n = row + 1; n < ROWS; n++)
        {
            // if an empty space is found, stop looping
            if (board[n][column] == SquareState::empty)
            {
                break;
            }

            // if a piece of the player is found, change the trigger so pieces can be flipped and exit loop
            if (board[n][column] == playerTurn)
            {
                flipTrigger = true;
                break;
            }
        }

        // if the flip trigger was activated in the last loop
        if (flipTrigger)
        {
            // loop through again
            for (int n = row + 1; n < ROWS; n++)
            {
                // when the piece is found, exit the loop
                if (board[n][column] == playerTurn)
                {
                    break;
                }
                else // otherwise flip the piece and move to the next
                {
                    board[n][column] = playerTurn;
                }
            }
        }
    }

    // north east
    if ((row + 1 < ROWS) && (column + 1 < COLS)) { ... }
    // east
    if (column + 1 < COLS) { ... }
    // south east
    if ((row - 1 >= 0) && (column + 1 < COLS)) { ... }
    // south
    if (row - 1 >= 0) { ... }
    // south west
    if ((row - 1 >= 0) && (column - 1 >= 0)) { ... }
    // west
    if (column - 1 >= 0) { ... }
    // north west
    if ((row + 1 < ROWS) && (column - 1 >= 0)) { ... }
}

```

Figure 23: Othello flip pieces

The valid move check method shown in figure 25 checks every square on the board, where it is looking for an empty square which has an opponent's piece in one of the eight neighbouring squares. This is to enforce the rule that players can only place their pieces adjacent to an opponent piece, and if they can't then their turn is skipped.

```
// A valid move only exists if there is an empty space with a neighbouring opponent piece
bool Othello::CheckForValidMoves()
{
    //set the flag to false to begin
    bool validMoveExists = false;
    //set the opponent
    SquareState opponent;
    switch (playerTurn)
    {
        case player1:
            opponent = player2;
            break;
        case player2:
            opponent = player1;
            break;
        default:
            break;
    }
    // loop through all the squares in the board
    for (int row = 0; row < ROWS; row++)
    {
        for (int col = 0; col < COLS; col++)
        {
            // if there is an empty space
            if (board[row][col] == SquareState::empty)
            {
                // check all the neighbouring squares
                // north square
                if (row + 1 < ROWS)
                {
                    if (board[row + 1][col] == opponent)
                    {
                        validMoveExists = true;
                    }
                }
                // north east square
                if ((row + 1 < ROWS) && (col + 1 < COLS)) { ... }
                // east square
                if (col + 1 < COLS) { ... }
                // south east square
                if ((row - 1 >= 0) && (col + 1 < COLS)) { ... }
                // south square
                if (row - 1 >= 0) { ... }
                // south west square
                if ((row - 1 >= 0) && (col - 1 >= 0)) { ... }
                // west square
                if (col - 1 >= 0) { ... }
                // north west square
                if ((row + 1 < ROWS) && (col - 1 >= 0)) { ... }
            }
        }
    }
    // if there is no valid move then swap the turn
    if (!validMoveExists)
    {
        SwapTurn();
    }
    return validMoveExists;
}
```

Figure 24: Othello check for valid moves

The check for victory method as shown in figure 26 only triggers when there are no remaining moves left. If there are no remaining moves, then the pieces are counted, and the player with the most is the winner, unless they are equal, then the game ends in a draw.

```
ConnectedPlayer* Othello::CheckForVictory()
{
    // if there are empty spaces, then skip victory check
    if (CheckForRemainingMoves())
    {
        return nullptr;
    }

    int p1Count = 0, p2Count = 0;

    // loop through all the squares on the board
    for (auto row : board)
    {
        for (auto col : row)
        {
            // count the number of p1 squares
            if (col == SquareState::player1)
            {
                p1Count++;
            }
            // count the number of p2 squares
            else
            {
                p2Count++;
            }
        }
    }

    // if p1 has more than p2, then p1 wins
    if (p1Count > p2Count)
    {
        return GetPlayer(SquareState::player1);
    }

    // if p2 has more than p1, then p2 wins
    if (p2Count > p1Count)
    {
        return GetPlayer(SquareState::player2);
    }

    // if neither has more than the other, then it is a draw
    return nullptr;
}
```

Figure 25: Othello check for victory

When the player connects to the server through the client, they are asked to enter their name. Once they do, they are then presented with the rules of the game and the starting board, as shown in figure 27.

```

Please enter username
tim
Welcome
Welcome to Othello
Rules:
1. You can only place a piece adjacent to an opponent piece
2. When a new piece connects in a straight line through
   to another piece of that kind, flip all the inbetween pieces
3. If there are no valid moves for a player to make, but there are still empty
   spaces, then their turn is skipped
4. The game ends when all squares are filled
5. The winner is the player at the end with the most pieces face up

0  1  2  3  4  5  6  7
0  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -
3  -  -  -  o  o  -  -
4  -  -  -  o  o  -  -
5  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -
0  1  2  3  4  5  6  7

```

Figure 26: Othello Welcome message

When two players connect to the lobby, the server starts the game, and the player who connected first is set to player one.

```

Please enter username
james
Welcome
Welcome to Othello
Rules:
1. You can only place a piece adjacent to an opponent piece
2. When a new piece connects in a straight line through
   to another piece of that kind, flip all the inbetween pieces
3. If there are no valid moves for a player to make, but there are still empty
   spaces, then their turn is skipped
4. The game ends when all squares are filled
5. The winner is the player at the end with the most pieces face up

0  1  2  3  4  5  6  7
0  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -
3  -  -  -  o  o  -  -
4  -  -  -  o  o  -  -
5  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -
0  1  2  3  4  5  6  7
Please select column (0-7)

Please enter username
tim
Welcome
Welcome to Othello
Rules:
1. You can only place a piece adjacent to an opponent piece
2. When a new piece connects in a straight line through
   to another piece of that kind, flip all the inbetween pieces
3. If there are no valid moves for a player to make, but there are still empty
   spaces, then their turn is skipped
4. The game ends when all squares are filled
5. The winner is the player at the end with the most pieces face up

0  1  2  3  4  5  6  7
0  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -
3  -  -  -  o  o  -  -
4  -  -  -  o  o  -  -
5  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -
0  1  2  3  4  5  6  7
It is your opponent's turn - please wait!

```

Figure 27: Othello two players connected

Each player takes turns entering board positions where they want to place their pieces.

Figure 29 shows the first two moves made by the connected players.

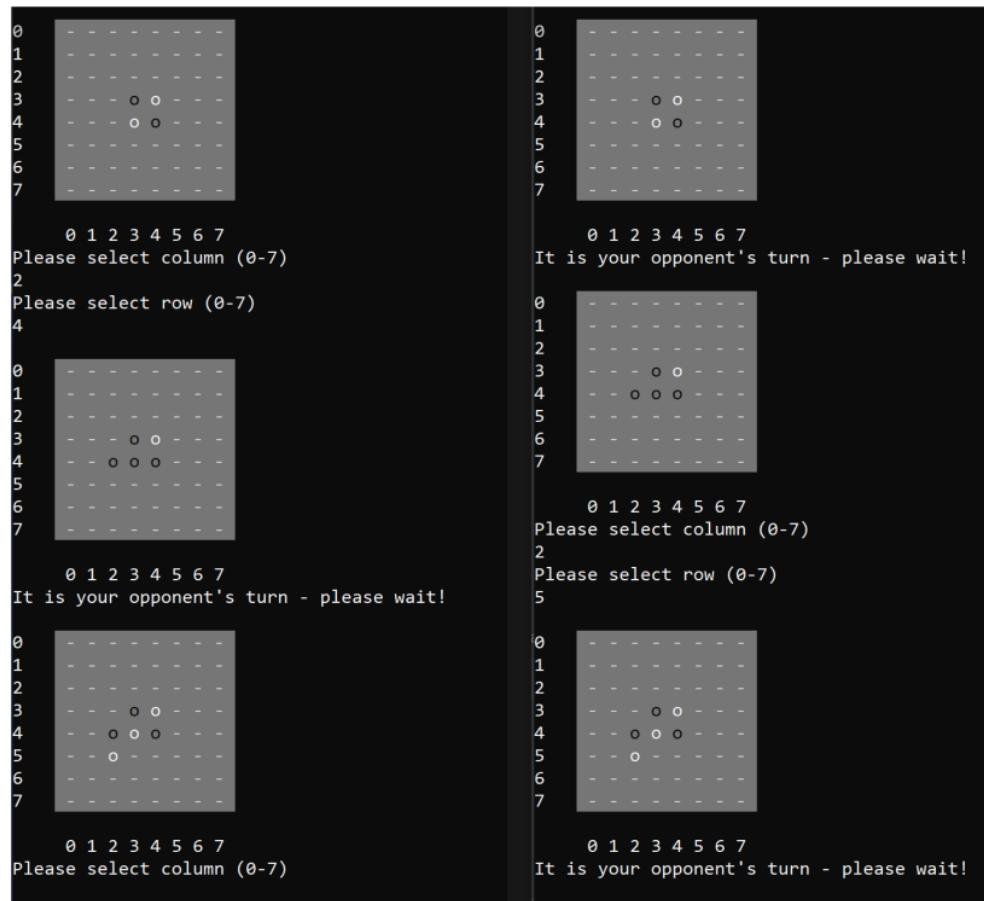


Figure 28: Othello first two moves

Figure 30 shows the result of a finished game, where player two emerged victorious.



Figure 29: Othello completed game

2.5 Dots and Boxes

As seen in the connect four and Othello games, Dots and Boxes has a client solution and server solution, where the server contains the game logic, and the client presents the user interface to the user and communicates moves to the server.

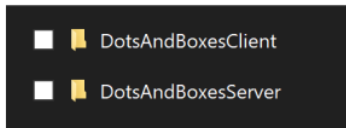


Figure 30: Dots and Boxes folders

The board is initialised in the game constructor as shown in figure 32 below.

- The dots of the board are placed into positions in the board where both the row and column are an even index.
- The empty spaces are placed into positions where the row and column are an odd index.
- The horizontal empty spaces are positions where the row is even and the column is odd.
- The vertical empty spaces are positions where the row is odd, and the column is even.

```

DotsAndBoxes::DotsAndBoxes(ConnectedPlayer* p1, ConnectedPlayer* p2)
{
    for (int r = 0; r < ROWS; r++)
    {
        vector<SquareState> row;

        for (int c = 0; c < COLS; c++)
        {
            if (r % 2 == 0 && c % 2 == 0)
            {
                row.push_back(SquareState::dot);
            }
            else if (!(r % 2 == 0 || c % 2 == 0))
            {
                row.push_back(SquareState::empty);
            }
            else if ((r % 2 == 0))
            {
                row.push_back(SquareState::hempty);
            }
            else if ((c % 2 == 0))
            {
                row.push_back(SquareState::vempty);
            }
        }

        board.push_back(row);
    }

    this->p1 = p1;
    this->p2 = p2;

    playerTurn = SquareState::player1;
}

```

Figure 31: Dots and Boxes board setup

The make move method shown in figure 33 below first checks whether the move is valid before continuing. If it is valid, then it changes the state of the chosen position based on its current position i.e. vertical empty space becomes vertical line and horizontal empty space becomes horizontal line.

```
bool DotsAndBoxes::MakeMove(int column, int row)
{
    //Bounds check
    if (column < 0 || column >= COLS)
    {
        return false;
    }
    if (row < 0 || row >= ROWS)
    {
        return false;
    }
    //Emptiness check
    switch (board[row][column])
    {
        case SquareState::vempty:
        case SquareState::hempty:
            break;
        default:
            return false;
    }
    // fill the line in
    switch (board[row][column])
    {
        case SquareState::vempty:
            board[row][column] = SquareState::vline;
            break;
        case SquareState::hempty:
            board[row][column] = SquareState::hline;
            break;
        default:
            break;
    }
    // check whether a square has been completed, if not then swap turn
    if (!CheckForCompletedSquares(row, column))
    {
        SwapTurn();
    }
    // When game ends, send notification to player by swapping turn
    if (!CheckForMovesRemaining())
    {
        SwapTurn();
    }
    return true;
}
```

Figure 32: Dots and Boxes make move.

The check for completed squares method as shown in figure 34 below checks whether a newly placed line completed a box. If it was a vertical line, then it checks the boxes to its right and left, if it was a horizontal line, then it checks the boxes above and below.

```
bool DotsAndBoxes::CheckForCompletedSquares(int row, int column)
{
    bool completedSquare = false;

    switch (board[row][column])
    {
        case vline:
            if (column - 1 >= 0)
            {
                if (board[row - 1][column - 1] == SquareState::hline &&
                    board[row + 1][column - 1] == SquareState::hline &&
                    board[row][column - 2] == SquareState::vline
                )
                {
                    board[row][column-1] = playerTurn;
                    completedSquare = true;
                }
            }
            if (column + 1 < COLS)
            {
                if (board[row - 1][column + 1] == SquareState::hline &&
                    board[row + 1][column + 1] == SquareState::hline &&
                    board[row][column + 2] == SquareState::vline
                )
                {
                    board[row][column+1] = playerTurn;
                    completedSquare = true;
                }
            }
            break;
        case hline:
            if (row - 1 >= 0)
            {
                if (board[row - 1][column - 1] == SquareState::vline &&
                    board[row - 1][column + 1] == SquareState::vline &&
                    board[row - 2][column] == SquareState::hline
                )
                {
                    board[row - 1][column] = playerTurn;
                    completedSquare = true;
                }
            }
            if (row + 1 < ROWS)
            {
                if (board[row + 1][column + 1] == SquareState::vline &&
                    board[row + 1][column - 1] == SquareState::vline &&
                    board[row + 2][column] == SquareState::hline
                )
                {
                    board[row+1][column] = playerTurn;
                    completedSquare = true;
                }
            }
            break;
        default:
            break;
    }

    return completedSquare;
}
```

Figure 33: Dots and Boxes check for completed squares.

The check for victory method as shown in figure 35 below first checks whether there are any remaining moves. If there are not, then it counts the claimed squares of each player respectively, and returns the player who has the most, or returns no player if they have tied.

```
ConnectedPlayer* DotsAndBoxes::CheckForVictory()
{
    if (CheckForMovesRemaining())
    {
        return nullptr;
    }

    int p1count = 0;
    int p2count = 0;

    for (auto r : board)
    {
        for (auto c : r)
        {
            switch (c)
            {
                case SquareState::player1:
                    p1count++;
                    break;
                case SquareState::player2:
                    p2count++;
                    break;
                default:
                    break;
            }
        }
    }

    if (p1count > p2count)
    {
        return GetPlayer(SquareState::player1);
    }
    if (p2count > p1count)
    {
        return GetPlayer(SquareState::player2);
    }
    return nullptr;
}
```

Figure 34: Dots and Boxes check for victory.

When a player connects to the server through the client, they are asked to enter their name. Once they have, they are then presented with a list of the rules an empty board.

```

Please enter username
james
Welcome to the server, james!
Welcome to Dots and Boxes
Rules:
1. Take turns in connecting dots with lines.
2. When a box is made, the player who made it claims it.
3. When a player claims a box, they take an extra turn.
4. The game ends when all the boxes have been claimed.
5. The player with the most claimed boxes wins.

  0  +  +  +  +  +
  1
  2  +  +  +  +  +
  3
  4  +  +  +  +  +
  5
  6  +  +  +  +  +
  7
  8  +  +  +  +  +

    0 1 2 3 4 5 6 7 8

```

Figure 35: Dots and Boxes welcome message.

When two players connect to the lobby, the server starts the game, and each player can then take turns drawing lines between the dots, as shown in figure 37 and 38.

```

Please enter username
james
Welcome to the server, james!
Welcome to Dots and Boxes
Rules:
1. Take turns in connecting dots with lines.
2. When a box is made, the player who made it claims it.
3. When a player claims a box, they take an extra turn.
4. The game ends when all the boxes have been claimed.
5. The player with the most claimed boxes wins.

  0  +  +  +  +  +
  1
  2  +  +  +  +  +
  3
  4  +  +  +  +  +
  5
  6  +  +  +  +  +
  7
  8  +  +  +  +  +

    0 1 2 3 4 5 6 7 8
Please select column (0-8)

Please enter username
rim
Welcome to the server, rim!
Welcome to Dots and Boxes
Rules:
1. Take turns in connecting dots with lines.
2. When a box is made, the player who made it claims it.
3. When a player claims a box, they take an extra turn.
4. The game ends when all the boxes have been claimed.
5. The player with the most claimed boxes wins.

  0  +  +  +  +  +
  1
  2  +  +  +  +  +
  3
  4  +  +  +  +  +
  5
  6  +  +  +  +  +
  7
  8  +  +  +  +  +

    0 1 2 3 4 5 6 7 8
It is your opponent's turn - please wait!

```

Figure 36: Dots and Boxes two players connected.

```

Please select column (0-8)
0
Please select row (0-8)
1
0  +  +  +  +  +
1  |
2  +  +  +  +  +
3
4  +  +  +  +  +
5
6  +  +  +  +  +
7
8  +  +  +  +  +

    0 1 2 3 4 5 6 7 8
It is your opponent's turn - please wait!

0  +---+  +  +  +
1  |
2  +  +  +  +  +
3
4  +  +  +  +  +
5
6  +  +  +  +  +
7
8  +  +  +  +  +

    0 1 2 3 4 5 6 7 8
Please select column (0-8)

It is your opponent's turn - please wait!
0  +  +  +  +  +
1  |
2  +  +  +  +  +
3
4  +  +  +  +  +
5
6  +  +  +  +  +
7
8  +  +  +  +  +

    0 1 2 3 4 5 6 7 8
Please select column (0-8)
1
Please select row (0-8)
0
0  +---+  +  +  +
1  |
2  +  +  +  +  +
3
4  +  +  +  +  +
5
6  +  +  +  +  +
7
8  +  +  +  +  +

    0 1 2 3 4 5 6 7 8
It is your opponent's turn - please wait!

```

Figure 37: Dots and Boxes first two moves.

When a player completes a box, they are awarded with an extra turn.

```

0 1 2 3 4 5 6 7 8
It is your opponent's turn - please wait!

0 +---+ + + +
1 |   |
2 + + + + +
3
4 + + + + +
5
6 + + + + +
7
8 + + + + +

0 1 2 3 4 5 6 7 8
Please select column (0-8)
2
Please select row (0-8)
1
0 +---+ + + +
1 |   |
2 + + + + +
3
4 + + + + +
5
6 + + + + +
7
8 + + + + +

0 1 2 3 4 5 6 7 8
It is your opponent's turn - please wait!

0 +---+ + + +
1 |   |
2 +---+ + + +
3 |
4 + + + + +
5
6 + + + + +
7
8 + + + + +

0 1 2 3 4 5 6 7 8
Please select column (0-8)

0 1 2 3 4 5 6 7 8
Please select row (0-8)
3
0 +---+ + + +
1 |   |
2 +---+ + + +
3 |
4 + + + + +
5
6 + + + + +
7
8 + + + + +

0 1 2 3 4 5 6 7 8
Please select column (0-8)

0 1 2 3 4 5 6 7 8
It is your opponent's turn - please wait!
  
```

Figure 38: Dots and Boxes box being claimed

Figures 40 and 41 below show two ending games states of two games, one showing the state where a player has won, and the other showing the state where the game ended in a tie.

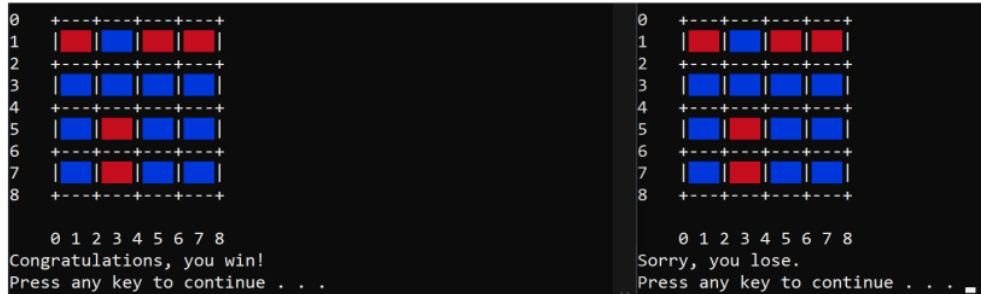


Figure 39: Dots and Boxes player 1 wins

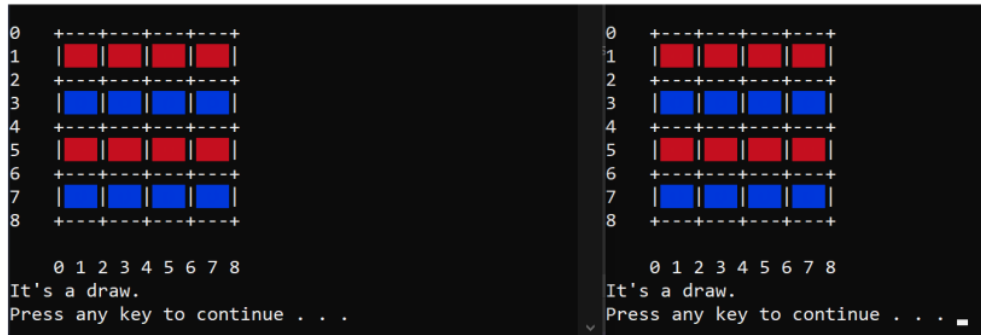


Figure 40: Dots and Boxes draw

2.6 Conclusion

In conclusion, the implementations of “Connect Four”, “Othello”, and “Dots and Boxes” successfully met the requirements established. The games can connect over a network and play the respective games with an opponent operating on a separate client. The clients display updated board states, and communicate moves to the server, which then applies the moves to the game. Each game successfully follows the respective game rules [29]–[31].

With further development, the three games could be combined into a game suite hub that allows players to select a game they wish to play and enter a lobby for the respective game. Also, a matchmaking lobby could be further developed, adding the ability for the server to run multiple instances of the game at one.

References

- [1] V. G. Cerf and R. E. Kahn, "A protocol for packet network intercommunication," *Comput. Commun. Rev.*, vol. 35, no. 2, pp. 71–82, 2005, doi: 10.1145/1064413.1064423.
- [2] "rfc793." <https://datatracker.ietf.org/doc/html/rfc793> (accessed May 11, 2021).
- [3] G. R. Wright, *TCP/IP illustrated*. 1995.
- [4] A. G. Blank, *TCP/IP JumpStart: Internet Protocol Basics*. 2006.
- [5] Ionos, "TCP: How the Transmission Control Protocol works - IONOS," 2020. <https://www.ionos.com/digitalguide/server/know-how/introduction-to-tcp/> (accessed May 11, 2021).
- [6] A. Mehdizadeh, M. Nagarajan, H. Harun, and M. Mohammadpoor, "Congestion Window Scaling Method to Optimize Delay in TCP/IP," *Wirel. Pers. Commun.*, vol. 101, no. 4, pp. 2227–2239, Aug. 2018, doi: 10.1007/s11277-018-5813-3.
- [7] H. Yuksel and Ö. Altunay, "Host-to-host TCP/IP connection over serial ports using visible light communication," *Phys. Commun.*, vol. 43, p. 101222, Dec. 2020, doi: 10.1016/j.phycom.2020.101222.
- [8] M. Urban and B. Tiemann, "{Sams} teach yourself {FreeBSD} in 24 hours," *Sams ; Pearson Educ. [distributor]*, p. xvii + 433, 2003.
- [9] A. Mejías, R. S. Herrera, M. A. Márquez, A. Calderón, I. González, and J. Andújar, "Easy handling of sensors and actuators over TCP/IP networks by open source hardware/software," *Sensors (Switzerland)*, vol. 17, no. 1, p. 94, Jan. 2017, doi: 10.3390/s17010094.
- [10] Q. Liu, Z. Xu, and Z. Li, "Implementation of hardware TCP/IP stack for DAQ systems with flexible data channel," *Electron. Lett.*, vol. 53, no. 8, pp. 530–532, Apr. 2017, doi: 10.1049/el.2016.4680.
- [11] F. A. Bender, "Delay dependent antiwindup for AQM in TCP/IP routers," *Asian J. Control*, vol. 16, no. 5, pp. 1511–1514, Sep. 2014, doi: 10.1002/asjc.810.
- [12] N. El Fezazi, F. El Haoussi, E. H. Tissir, F. A. Bender, and T. Alvarez, "Multiclass AQM on a TCP/IP router: A control theory approach," *Int. J. Robust Nonlinear*

Control, vol. 29, no. 6, pp. 1719–1733, Apr. 2019, doi: 10.1002/rnc.4463.

- [13] E. Baccarelli, P. G. Vinueza Naranjo, M. Shojafar, and M. Scarpiniti, “Q*: Energy and delay-efficient dynamic queue management in TCP/IP virtualized data centers,” *Comput. Commun.*, vol. 102, pp. 89–106, Apr. 2017, doi: 10.1016/j.comcom.2016.12.010.
- [14] “Two Generals’ Problem – Finematics.” <https://finematics.com/two-generals-problem/> (accessed May 11, 2021).
- [15] “Path MTU Discovery and Filtering ICMP.” <https://web.archive.org/web/20110809214618/http://www.znep.com/~marcs/mtu/index.html> (accessed May 11, 2021).
- [16] K. Moldeklev and P. Gunningberg, “How a Large ATM MTU Causes Deadlocks in TCP Data Transfers,” *IEEE/ACM Trans. Netw.*, vol. 3, no. 4, pp. 409–422, 1995, doi: 10.1109/90.413215.
- [17] “PDU (Protocol Data Unit) Definition.” <https://techterms.com/definition/pdu> (accessed May 11, 2021).
- [18] O. Sharon and Y. Alpert, “Physical and medium access control criteria for the optimal medium access control protocol data unit size in automatic repeat request-enabled connections in IEEE 802.16e/WiMAX systems,” *Int. J. Commun. Syst.*, vol. 27, no. 11, pp. 3292–3311, Apr. 2014, doi: 10.1002/dac.2542.
- [19] “What is Synchronous Transmission and Asynchronous Transmission? – Fosco Connect.” <https://www.fiberoptics4sale.com/blogs/archive-posts/95042950-what-is-synchronous-transmission-and-asynchronous-transmission> (accessed May 11, 2021).
- [20] “Difference between Synchronous and Asynchronous Transmission.” <https://www.guru99.com/difference-between-synchronous-and-asynchronous-transmission.html> (accessed May 11, 2021).
- [21] “Understanding Asynchronous I/O in Play Framework | mobiarch.” <https://mobiarch.wordpress.com/2014/11/13/understanding-asynchronous-io-in-play-framework/> (accessed May 11, 2021).

- [22] J. Boney, *Cisco IOS in a Nutshell*. 2001.
- [23] "What is Interoperability?" <https://www.computerhope.com/jargon/i/interoperability.htm> (accessed May 11, 2021).
- [24] F. B. Vernadat, "Technical, semantic and organizational issues of enterprise interoperability and networking," in *Annual Reviews in Control*, Apr. 2010, vol. 34, no. 1, pp. 139–144, doi: 10.1016/j.arcontrol.2010.02.009.
- [25] H. Panetto and J. Cecil, "Information systems for enterprise integration, interoperability and networking: Theory and applications," *Enterprise Information Systems*, vol. 7, no. 1. pp. 1–6, Feb. 2013, doi: 10.1080/17517575.2012.684802.
- [26] A. Tapiador and S. Hassan, "Understanding federation: An analytical framework for the interoperability of social networking sites," *arXiv*. May 16, 2018, Accessed: Mar. 23, 2021. [Online]. Available: <http://arxiv.org/abs/1805.06474>.
- [27] "What is Interoperability? Definition and FAQs | OmniSci." <https://www.omnisci.com/technical-glossary/interoperability> (accessed May 11, 2021).
- [28] Paulo Teixeira de Sousa and P. Stuckmann, "Telecommunication Network Interoperability," in *Telecommunication Systems and Technologies*, vol. II, no. 5, 2009, pp. 1–37.
- [29] Hasbro, "Connect Four online." <https://www.hasbro.com/common/documents/dad2614d1c4311ddb0b0800200c9a66/1EF6874419B9F36910222EB9858E8CB8.pdf> (accessed May 12, 2021).
- [30] "English :: World Othello Federation." <https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english> (accessed May 12, 2021).
- [31] M. Barile, "Dots and Boxes · GitHub," *MathWorld*. <https://mathworld.wolfram.com/DotsandBoxes.html> (accessed May 12, 2021).

SP+C Final Report

GRADEMARK REPORT

FINAL GRADE

92/100

GENERAL COMMENTS

Instructor

An excellent report, well written and structured with a extremely competent and coherent academic practice.

Three implementations showing your evolution through the process. Truthfully, any of these would have been suitable, so this has gone far above and beyond what would've been expected.

Really excellent work, James.

Report (100): 91

Implementation (100): 93

Module average: 92

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

