

TaylorHodges_Part2

by Taylor Hodges

Submission date: 02-May-2023 03:35PM (UTC+0100)

Submission ID: 204338591

File name: 23668_Taylor_Hodges_TaylorHodges_Part2_499749_1996035881.docx (575.37K)

Word count: 1680

Character count: 8569

Assignment 2 – report 1000 words

- The features and design decisions of your application
- including **how it adheres to best practice.**

Contents

Features	3
Sockets	3
Security	3
Messages.....	4
Serialise and unserialise.....	4
Threading and Synchronisation.....	5
Alternating turns.....	5
Pairs.....	5
Disconnected player.....	5
Input validation	6
Code snippets.....	7
Efficiency check for victory.....	7
Grid pairs.....	9
Desirable features and their implementation difficulties	10
Bibliography	11

List of Code Snippets

Code Snippet 1: [1] Socket creation and initialisation of a socket.	3
Code Snippet 2: [2] Blind connection of a socket, and use of the socket to connect to server.....	3
Code Snippet 3: [3] Closing of the socket.	3
Code Snippet 4: Make move function in server.	3
Code Snippet 5: Setup and sending of connection message.	4
Code Snippet 6: Serialise and unserialise of the board in the client.	4
Code Snippet 7: [4] use of a mutex during synchronisation.	5
Code Snippet 8: [5] Swap turns function in server.....	5
Code Snippet 9: Check for victory function on the server.	8
Code Snippet 10: Client function prints colour options.....	9
Code Snippet 11: client's print board function [2].....	9
Code Snippet 12: Integer validation.....	10

List of Tables

Table 1: Diagrams and pseudo code of the check for victory function logic.	7
Table 2: example setup of printing coloured-pieces.....	9

Features

Sockets

Sockets were used to establish a connection between server and clients. By supplying the socket with an address family 'sin_family', socket type 'SOCK_STREAM', and a protocol 'IPPROTO_TCP', it can connect to a client with IP address and port number to data exchange.

The connection

```
8 void Connection::InitialiseSocket(){
9     WSADATA wsaData;
10    if (int errorCode = WSStartup(MAKEWORD(2, 2), &wsaData) < 0){
11        throw WSAGetLastError();}
12
13    void Connection::CreateSocket(){
14        sockfd = socket(address->GetSockaddrRef()->sin_family, SOCK_STREAM, IPPROTO_TCP);
15        if (sockfd <= 0){
16            throw WSAGetLastError();}
```

Code Snippet 1: [1] Socket creation and initialisation of a socket.

```
7 void ClientConnection::BindConnection(){
8     if (int errorCode = connect(sockfd, (sockaddr*)&address->GetSockaddrRef(), (int)address->GetSocketLengthVal()) < 0){
9         throw WSAGetLastError();}
10
11    ConnectionInstance* ClientConnection::ConnectToServer(){
12        InitialiseSocket();
13        CreateSocket();
14        BindConnection();
15        return new ConnectionInstance(sockfd);
16    }
```

Code Snippet 2: [2] Bind connection of a socket, and use of the socket to connect to server.

```
99 void ConnectionInstance::Disconnect(){
100     closesocket(sock);}
```

Code Snippet 3: [3] Closing of the socket.

Security

The application holds all of the game logic and variable manipulation in the server as opposed to the client. This is done to ensure that the client does not have access to data where they could make amendments. For example, the user inputs a coordinate on the client side which is sent to the server for that move to be made; if this move was made client side, it could be possible for the client to make changes that they should not be able to, such as put their pieces on unavailable spaces and enable them to win unauthentically.

```
36 bool Connect4::MakeMove(int column){
37     if (column < 0 || column > 7){ //Bounds check
38         return false;}
39     int row = 5;
40     while (board[row][column].second != SquareState::empty && row >= 0) { //look for available row in column
41         row--; }
42     if (row == -1){
43         return false;}
44     board[row][column] = make_pair(GetCurrentPlayer()->GetColour(), playerTurn);
45     SwapTurn();
46     return true;
47 }
```

Code Snippet 4: Make move function in server.

Similarly, relevant gameplay functions are also out-of-reach from the players by locating them in the server. Such as check-for-victory function which checks for lines of four connected pieces and deems the corresponding player as the winner. The player does not have access to this code they may have the ability to make amendments and grant themselves the win without earning it, hence these functions are in the server and not client.

[See Code Snippet 9: Check for victory function on the server.]

Messages

Messages are used to send data back and forth between the server and client. Messages enable communications between connected clients and the server and are used to send grid updates, client details, client moves, etc.

```
38     username = GetStringFromConsole("Please enter username: ");
39     screen->PrintColourOp();
40     do{
41         colour = GetIntFromConsole("Please select colour(1-7): ");
42     } while (colour < 1 || colour > 7);
43     ConnectMessage* cm = new ConnectMessage();
44     cm->SetDetails(username, colour);
45     SendNetworkMessage(cm);
```

Code Snippet 5: Setup and sending of connection message.

Serialise and unserialise.

Messages are serialised, meaning multiple pieces of data can be appended in series and sent as one message. The receiver can then unserialise the message at its destination and separate the included variables in reverse for use. This is used on all messages.

```
14     void GridUpdate::Serialise(ostream& os){
15         Message::Serialise(os);
16         for (auto i = board.rbegin(); i != board.rend(); i++){
17             for (auto j = i->rbegin(); j != i->rend(); j++){
18                 os << j->first << endl << j->second << endl;}}
19
20     void GridUpdate::UnserialiseType(istream& is){
21         for (int i = 0; i < 6; i++) {
22             vector<pair<int, SquareState>> row;
23             for (int j = 0; j < 7; j++) {
24                 int colour;
25                 is >> colour;
26                 int ss;
27                 if (is >> ss) {
28                     row.push_back(make_pair(colour, static_cast<SquareState>(ss)));}
29             }
30             board.push_back(row);
31         }
32     }
```

Code Snippet 6: Serialise and unserialise of the board in the client.

Threading and Synchronisation

Each connecting player has their own thread so that they can connect individually. This enables both clients to join simultaneously and play without interruptions or bottlenecks.

These threads are synchronised so to prevent problems such as race conditions, deadlocks, starvation, etc, mutexes are used to halt multiple threads from passing.

```
123 //wait for opponent's turn to be over
124 unique_lock<mutex> lockT(turnSync);
125 turnChange.wait(lockT, [&] {return tPlayer == c4->GetCurrentPlayer() || opponent->GetConnectionStatus() == disconnected; });
126 lockT.unlock();
```

Code Snippet 7: [4] use of a mutex during synchronisation.

Alternating turns

When a turn begins, the client uses an enum to track which player's turn it is. When a turn ends, the player's alternate between 'your turn' and 'opponents turn'. Depending on which enum is found, depends which code is run for that user and if they can input a coordinate or wait for their opponent to take their turn.

```
49 void Connect4::SwapTurn(){
50     switch (playerTurn){
51     case player1:
52         playerTurn = player2;
53         return;
54     case player2:
55         playerTurn = player1;
56         return;
57     }
58 }
```

Code Snippet 8: [5] Swap turns function in server.

Pairs

The grid colours player's pieces based on their selected colour. This selected colour is sent to the server and processed into the grid as integer colour, and square state (empty, player-1, or player-2.) When the client receives a grid update, the grid is printed incorporating the colour on the pieces. This enables both clients to see both player's selected colours of the respective pieces in the grid.

[For use of pairs see Code Snippet 4: Make move function in server. and Code Snippet 6: Serialise and unserialise of the board in the client.] [For use of the integer colour of the pair, see Code Snippet 11: client's print board function .]

Disconnected player

If a player disconnects from the server, the current player is alternated and this remaining player is deemed the winner.

Input validation

When the user is asked to input an integer, the client ensures the input is in range of the stated options. For example, when the player selects a colour, their options are to select from the list (1-7), if the input is less than one or greater than seven, the player is asked to try again. [See Code Snippet 5: Setup and sending of connection message.]

It would be better practice to have improved validation, including checking for correct data type on the server, to better validate inputs with better security.

Code snippets

It should be noted that code was supplied by T. Bashford and delivered in the classroom, this involves a networked elements and structural significance of the program; this program was adapted for this project to be Connect-4 game and evolved where possible such as addition of the colour element and relevant messages, board size, structure, and serialise/unserialise and print formating, etc.

Efficiency check for victory

As there are 4×10^{12} combinations available on the board, there are very many different winning combinations. The writer has devised an efficient way to check for victory in this unpredictability.

The check for victory function attempts to thoroughly check all opportunities for victory on the board in the most efficient manner by sharing for loops where possible, and checking if the box is not empty before proceeding with a check for victory.

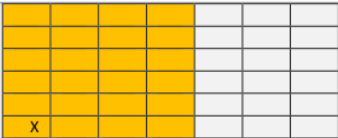
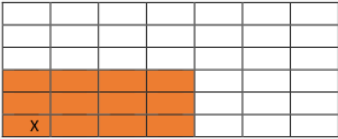
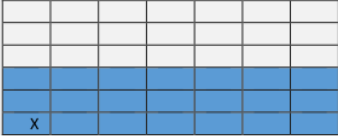
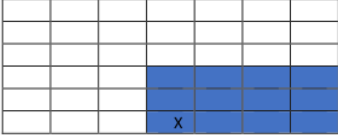
CHECK IF BOX IS NOT EMPTY, COORDINATE RANGE	CHECK METHOD FOR FOUR CONSECUTIVE PIECES FROM INITIAL CHECKED BOX
	Check horizontal – starting from X, check each box from the shaded area for square state ‘not empty.’ If so, check the next three boxes to the right.
	Check diagonal 1 - during the same for loops for efficiency, compete the same check. Check the next three boxes diagonal up and right.
	Check vertical – in the next section of code, compete the same check. Check the next three boxes above.
	Check diagonal 2 - during the same for loops for efficiency, compete the same check. Check the next three boxes diagonal up and left.

Table 1: Diagrams and pseudo code of the check for victory function logic.

```

63 ConnectedPlayer* Connect4::CheckForVictory(){
64     for (int y = 5; y >= 0; y--){
65         for (int x = 0; x < 4; x++){
66             // check horizontal (East)
67             if (board[y][x].second != SquareState::empty &&
68                 board[y][x] == board[y][x + 1] && board[y][x] == board[y][x + 2] && board[y][x] == board[y][x + 3]){
69                 return GetPlayer(board[y][x].second);
70             }
71             if (y > 2){
72                 // check diagonal, right incline (NE)
73                 if (board[y][x].second != SquareState::empty &&
74                     board[y][x] == board[y - 1][x + 1] && board[y][x] == board[y - 2][x + 2] && board[y][x] == board[y - 3][x + 3]){
75                     return GetPlayer(board[y][x].second);
76                 }
77             }
78         }
79         for (int x = 0; x <= 6; x++){
80             for (int y = 5; y > 2; y--){
81                 // check vertical (North)
82                 if (board[y][x].second != SquareState::empty &&
83                     board[y][x] == board[y - 1][x] && board[y][x] == board[y - 2][x] && board[y][x] == board[y - 3][x]){
84                     return GetPlayer(board[y][x].second);
85                 }
86                 if (x > 2){
87                     // check diagonal, left incline (NW)
88                     if (board[y][x].second != SquareState::empty &&
89                         board[y][x] == board[y - 1][x - 1] && board[y][x] == board[y - 2][x - 2] && board[y][x] == board[y - 3][x - 3]){
90                         return GetPlayer(board[y][x].second);
91                     }
92                 }
93             }
94         }
95     }
96     return nullptr; //no victory found.
97 }

```

Code Snippet 9: Check for victory function on the server.

Grid pairs

The grid is set up as a vector (rows) of vectors (boxes) of pairs, which hold the integer of colour and the state of the square (empty, player-1, or player-2.) The colour is set from the player's input, these integers relate to the syntax used to change the colour, meaning the colour integer is used directly in the cout without the need to create a switch case.

When printing the grid element to the screen, a switch case is used to determine the state of the square. If the state is player-1 or player-2, it means the square will hold a piece and therefore should be the respective player's colour.

The first part of the pair holds an integer (1-7). This integer is the same as the 'place value one' unit of the reference numbers: e.g. integer 2 for green is used in the syntax "\033[1:32m" predeceasing the desired coloured text, therefore the variable is inserted into the statement below as "C":

Set colour	Text	Return colour to default
\033[1:3Cm	X or O	\033[1;0m

Table 2: example setup of printing coloured-pieces.

```
16 void Screen::PrintColourOp()
17 {
18     cout << "Colour options-" << endl;
19     cout << "1 - Red" << endl << "2 - Green" << endl
20         << "3 - Yellow" << endl << "4 - Blue" << endl
21         << "5 - Magenta" << endl << "6 - Cyan" << endl
22         << "7 - White" << endl;
23     /* Red: 31
24        Green: 32
25        Yellow: 33
26        Blue: 34
27        Magenta: 35
28        Cyan: 36
29        White: 37*/
30 }
```

Code Snippet 10: Client function prints colour options.

```
186 void Connect4NetworkClient::PrintBoard(vector<vector<pair<int, SquareState>>> board){
187     screen->ClearScreen();
188     int row = 0;
189     cout << "| 0 | 1 | 2 | 3 | 4 | 5 | 6 |" << endl;
190     cout << "-----" << endl;
191     for (auto col : board){
192         row++;
193         for (auto square : col){
194             cout << "| ";
195             switch (square.second){
196                 case SquareState::empty:
197                     cout << " ";break;
198                 case SquareState::player1:
199                     cout << "\033[1;3" << square.first << "m" << "O" << "\033[1;0m ";break;
200                 case SquareState::player2:
201                     cout << "\033[1;3" << square.first << "m" << "X" << "\033[1;0m ";break;
202             }
203             cout << "| " << endl << "-----" << endl;
204         }
205     }
206     cout << endl;
207 }
```

Code Snippet 11: client's print board function [2].

Desirable features and their implementation difficulties

The following features were desired and the problems preventing their implementation.

- No play again feature – after completing a game, the results are printed and the game ends. After two players are connected, the server does not clear these connections, meaning no new game can commence.
 - In attempt to resolve this, the function 'PlayGame' was encapsulated in a while true loop. This kept the server running, but the server still saw the two client connections even after they had left. These remaining connections were attempted to be dropped, but results were unsuccessful.
 - To resolve this, the server would have to close these connections only once the clients have disconnected.
- Input validation – the client has some input validation for integer ranges, but the program does not deal well if the player's input is not an integer.
 - The developer created a validation while loop in the client to reject integers out of range.

```
85 | if (((GameState*)m)->GetState() == yourturn){
86 |     while (true){
87 |         int col;
88 |         do{
89 |             col = GetIntFromConsole(username + ", please select column (0-6): ");
90 |         } while (col < 0 || col > 6);
91 |         cout << endl;
92 |         SendNetworkMessage(new MakeMove(col));
93 |         Message* tm = GetNetworkMessage();
```

Code Snippet 12: Integer validation.

- This solution still relies on the client to input an integer data structure.
- The server would need implementation of a try function or a test of 'isdigit' to reject the input when input out of range or invalid data types and prompt for another attempt to prevent crashing.

Bibliography

- [1] T. Bashford, *Server/TicTacToeGame/Connection.cpp*.
- [2] T. Bashford, *Server/TicTacToeGame/ClientConnection.cpp*.
- [3] T. Bashford, *Server/TicTacToeGame/ConnectionInstance.cpp*.
- [4] T. Bashford, *Server/TicTacToeGame/TicTacToeNetwork.cpp*.
- [5] T. Bashford, *Server/TicTacToeGame/TicTacToeNetwork.cpp*.
- [6] T. Bashford, *Client/TicTacToeGame/TicTacToeNetworkClient.cpp*.
- [7] T. Bashford, *Server/TicTacToeGame/TicTacToe.cpp*.

FINAL GRADE

GENERAL COMMENTS

Instructor

70/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11