project for KIV/UPS

A TCP server-client game (Connect Four)

Jakub Šilhavý

A17B0362P

silhavyj@students.zcu.cz

17. 10. 2020

Contents

1   Assignment description

The whole assignment description is quite detailed and can be found at `http://home.zcu.cz/~ublm/vyuka/ups/Pozadavky2019.pdf`.

In short, each student was supposed to choose a game for at least two players and implement a server-client application based on the TCP protocol. As one of the requirements, the server was supposed to be written in `C/C++`, and the client, on the other hand, in a high-level programming language such as `Java` or `C#`. The client should also implement a graphical user interface (`GUI`) visualizing the game and giving the player more entertaining experiences playing it than a terminal would provide.

The whole application should be robust and able to handle multiple clients playing multiple games at the same time. Both sides of the application must follow a protocol, which the student was supposed to come up with, and appropriately react to unexpected situations. For example, if a client loses their connection while playing a game, they should be put back in the game once their connection to the server is re-established again, so they can continue playing where their connection went off.

If either of the sides finds out the other one is not following the protocol, it should recognize it, log it, so it could be analyzed later on as an incident, and appropriately respond to this event. For example, if it all happens on the server-side, the server may cut the client off for not following the protocol. As far as logs are concerned, both sides of the application should log everything of what is going on as they communicate with the other side, so it could be used for both the process of debugging and analyzing incidents.

When it comes to compilation, both parts of the project should be compiled using some standard compilation tools such as `make` for `C/C++`, and `Ant` or `Maven` for Java.

2   Game explanation

As mentioned in chapter 1, each student was supposed to choose a game for at least two players. As far as mine is concerned, I decided to go with a game called *Connect Four*, a full explanation of which can be found at `https://en.wikipedia.org/wiki/Connect_Four`.

2.1   Rules

Essentially, this game is pretty similar to well-known *Tic Tac Toe* with a couple of slight modifications. Whenever a player chooses a position on the grid, his *disk* drops all the way down as far as it can until it gets stuck either at the bottom of the grid or on the top of another previously-placed disk. You can think of a disk as an *X* or *O* in Tic Tac Toe. A player who connects four of his disks in any direction wins the game.

2.1.1   Grid

At the very beginning of the game, there is an empty grid made up by six columns and five rows as it is shown below.
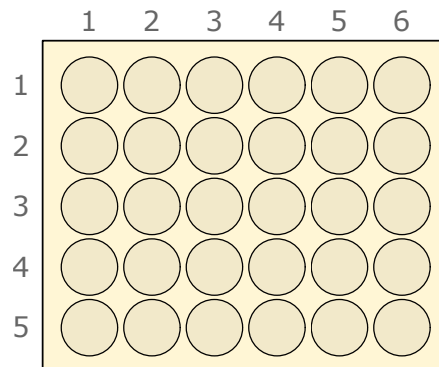

Figure 2.1: An empty grid (board) of the game

2.1.2   Players' turns

If the first player chooses to place their first disk at, for example, position 4 (we consider this to be the x position because the y position does not really

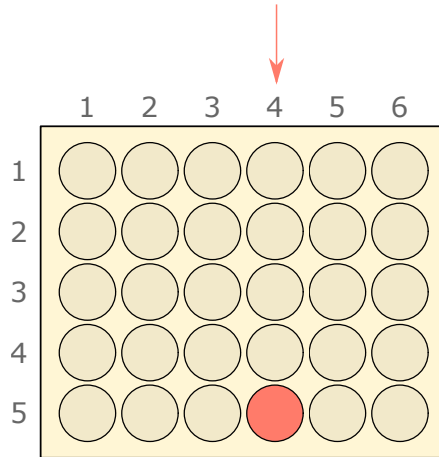matter as it falls all the way down anyway), the grid will look as it is shown in the picture down below.

Figure 2.2: The grid after the first player has already played

The other player has now the option of placing their disk wherever they want to, within x positions 1 and 6, of course. If they decide to copy their opponent's move and go with position 4, the disk will eventually lend on position [4;4] - right above their opponent's disk. Let us consider they do so, then the grid will look like it is shown below.
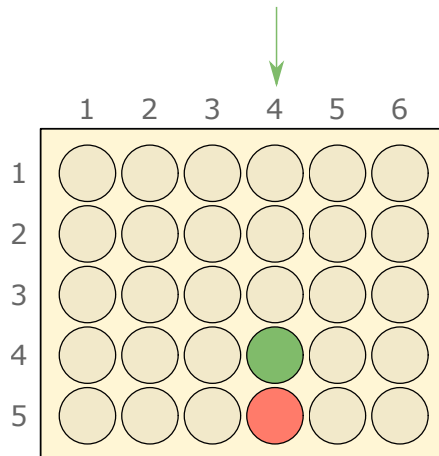
Figure 2.3: The grid after the other player has already played as well

2.1.3 End of the game

After a couple of turns, the game may finish in two different ways. Either one of the players wins the game, or the game ends in a draw, meaning there is no room for another disk on the grid, and yet neither of the players has won.
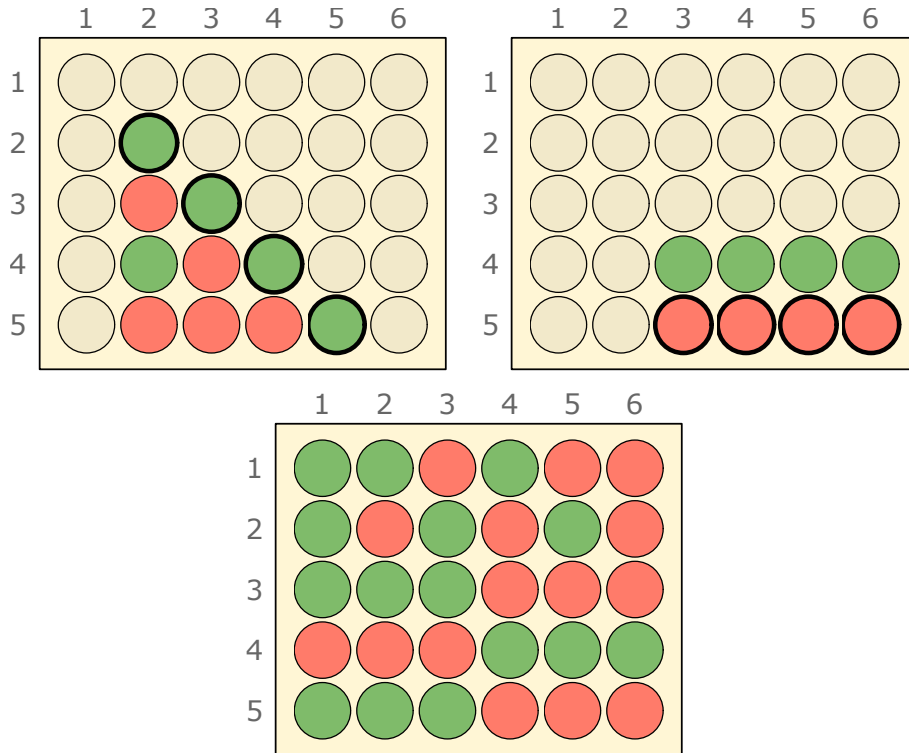

Figure 2.4: Different ways the game may end (winning disks - thicker edges)

In the top left picture above, the second player wins the game. The right top picture shows the opposite situation where the first player wins. The most interesting end of the game is the third picture where no one wins, and the game ends in a draw.

3   Communication protocol

The entire communication protocol can be divided into several different categories as it depends on the situation the client is currently in (playing a game, just connected to the server, ...). In the following sections, I will explain step by step how the protocol works as well as the way it deals with some unusual situations such as when someone tries to make the server crash.

Table 3.1: List of all the commands the client can send off to the server

Command	Description
`NICK <nick>`	sets the client's nickname
`RQ <nick>`	sends a game request to another client
`RQ_CANCELED <nick>`	cancels the game request
`RPL <nick> <YES/NO>`	replies to the game request (YES/NO)
`GAME_PLAY <x_position>`	plays the game (x position of the board)
`GAME_CANCELED`	cancels the game that is being played
`EXIT`	the clients leaves the server
`/PING`	pings the server
`/NICK`	returns the client's name
`/SATE`	returns the client's state
`/ALL_CLIENTS`	returns all active clients
`/HELP`	returns help

In the table above, there are all the commands the user can execute (send off to the server). However, they must know when they can use which commands, except for the commands starting with a slash and the command `EXIT` - those can be used at any time. The list of all the messages the client may receive from the server can be found here 3.6.

3.1   Message structure

For the structure of the protocol, every message must start with a protocol id, which in this case is `silhavyj`, following by the length of the message aligned up to four bytes. For instance, when a client joins the server and sets their nickname, the sent message looks like `silhavyj0009NICK Kuba`.

3.2 Connecting to the server

The user can connect to the server either via the client application, in which the protocol is already implemented, or via tools like `Netcat` or `PuTTY` (`https://www.putty.org/`). In case the user decides not to use the client application, they need to know how the protocol works and what they are supposed to do at all times. Otherwise, they risk to get themselves disconnected for not following the protocol. The server is strict on purpose to prevent any attempts to make it crash.

Once connected to the server, the user is supposed to enter the nick they want to present themselves on the server, which is supposed to be only one word. To do so, they have a limited amount of time of 10s. If they do not enter their nick within this period, they will be automatically cut off for not following the protocol.
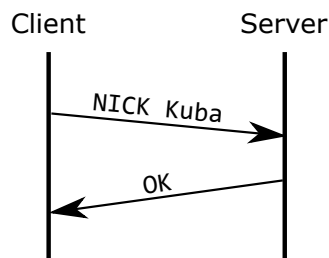

Figure 3.1: Entering the client's nick upon their connection to the server

The reason for this is the limited number of clients that can be connected to the server at a time, and some people might try to exploit this by connecting a lot of clients without a purpose. Therefore, if they want to connect to the server, they must know the protocol and enter their nick first. If the nick is entered successfully, the server will send back an acknowledge message, and the client will be moved into the lobby 3.3. If the user enters a nick that is already taken by another user, they will be automatically disconnected.

3.3 Lobby

The lobby is a state of client, in which they can either send a game request or receive one. The client is able to see which clients are currently available to play a game and which are already in a game, so they can decide to whom they want to send a game request. When a client is done playing a game, they will be put back into the lobby again.
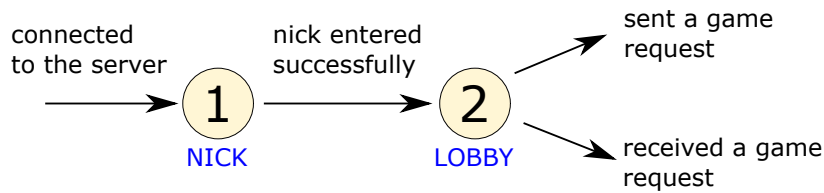
Figure 3.2: States the client has gone through so far

3.3.1 Sent a game request

When a client decides to send a game request to another client, the other client is moved on to a state (3.3.2) where they no longer can send nor receive a game request. All they can do is to either accept or reject the one they just received. Any attempt to do otherwise will lead to not following the protocol, and the client will be kicked off the server. More or less the same rules apply for the client who sent the game request - they can either wait for a reply or change their mind and cancel the request themselves.
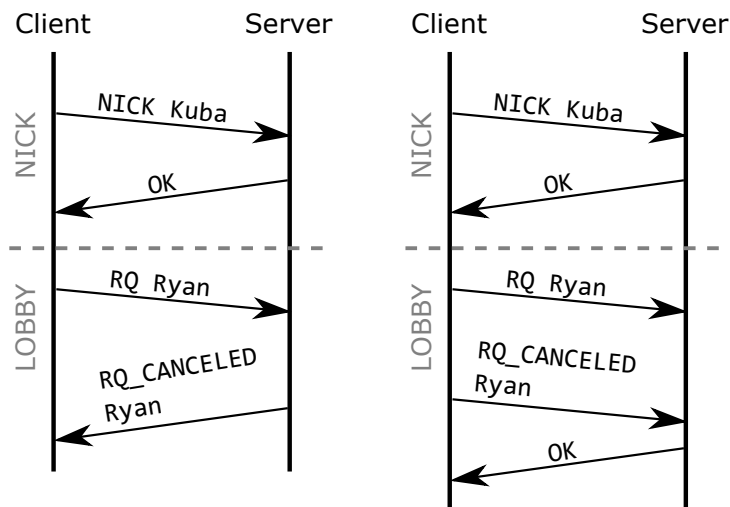

Figure 3.3: Examples of a client sending a game request

3.3.2 Received a game request

When a client receives a game request, they have, once again, a limited amount of time, within which they have to either accept the challenge or reject it. If they do not do so, the game request will be canceled automatically once the time of 30s is up. When the client gets into this state, they are not

allowed to do anything but these two actions as any attempt to do otherwise would get them kicked off the server for not following the protocol.
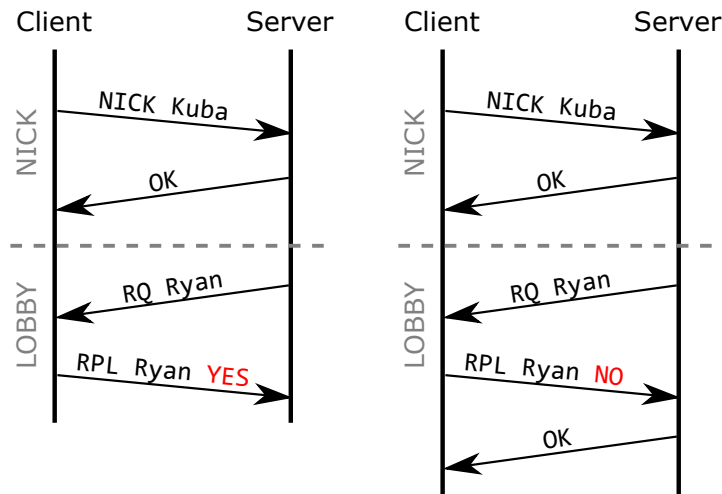
Figure 3.4: Examples of a client accepting/rejecting a game request

3.4 Playing a game

Once both of the clients agree on playing a game (the first one by sending a game request, and the other one by accepting it), they will be moved into a game room where they can play a game against each other. By now, all the other clients connected to the server have been already informed about these two clients being busy, meaning they cannot be sent a game request until they are both done playing the game.

The client who sent the game request starts first, and both play by the rules explained in chapter 2. At any time, either of the clients can decide to cancel the game for several reasons. For example, if they are no longer entertained, they can simply use command `GAME_CANCELED`, and both clients will be put back into the lobby (3.3).
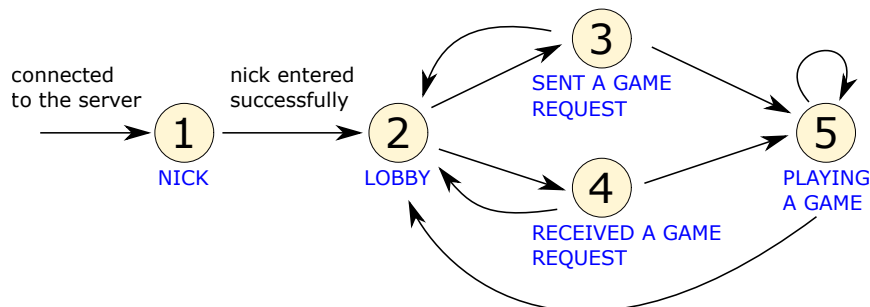

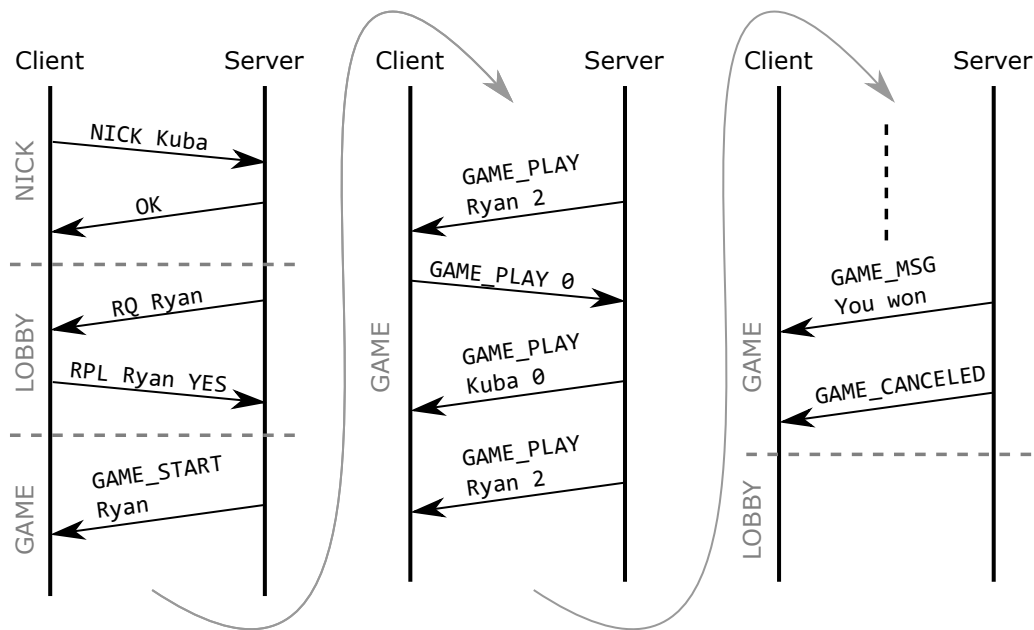Figure 3.5: States the client has gone through so far

Figure 3.6: Communication between the server and a client when playing a game

The game also sends informative messages to both clients, so they are updated on what is going on. For instance, if a client suddenly loses their connection (3.5), the other client will receive a message explaining what just happened. Another example is announcing the winner of the game or when a client is trying to play their turn but they are not up yet.

3.5   Lost connection while in a game

When a client is playing a game, their connection may go off and consequentially, they will be disconnected from the server. In that case, the server should recognize what just happened and notify the other player as well.

The procedure that was put in place to handle such events works the following way. If the client gets unintentionally disconnected, the other player (their opponent) will wait for 60s for the client to get re-connected back to the server. If they do not appear back online within that time, or the opponent changes their mind while waiting and cancels the game, the client will be treated as if the game ended in one of the standard ways (either of the player has won or the game was canceled on purpose).
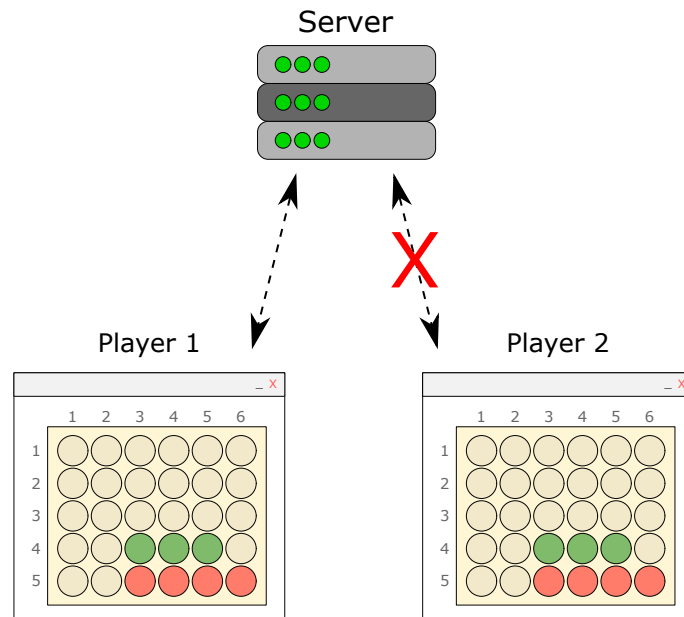
Figure 3.7: A player loses their connection while playing a game

If the client makes it back to the server in time, they will be put back in the game as it was when they went offline, so they can continue playing it. The condition for this is that the other player must stay online waiting for them to come back.

In order to find out if a client lost their connection, each client connected to the server is supposed to keep sending `PING` messages off to the server. If the server does not receive a `PING` message from the client for more than 6s, it automatically assumes the client lost their connection and will be treated accordingly. The same approach is used for the client in order to find out whether or not the server is still online. Therefore, if the user decides to connect to the server via `PuTTY`, they need to keep in mind the fact they have to send a `PING` message every 6s.
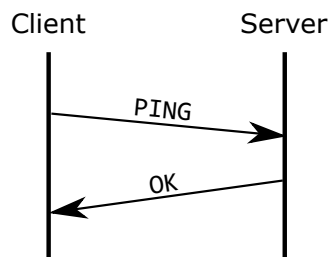

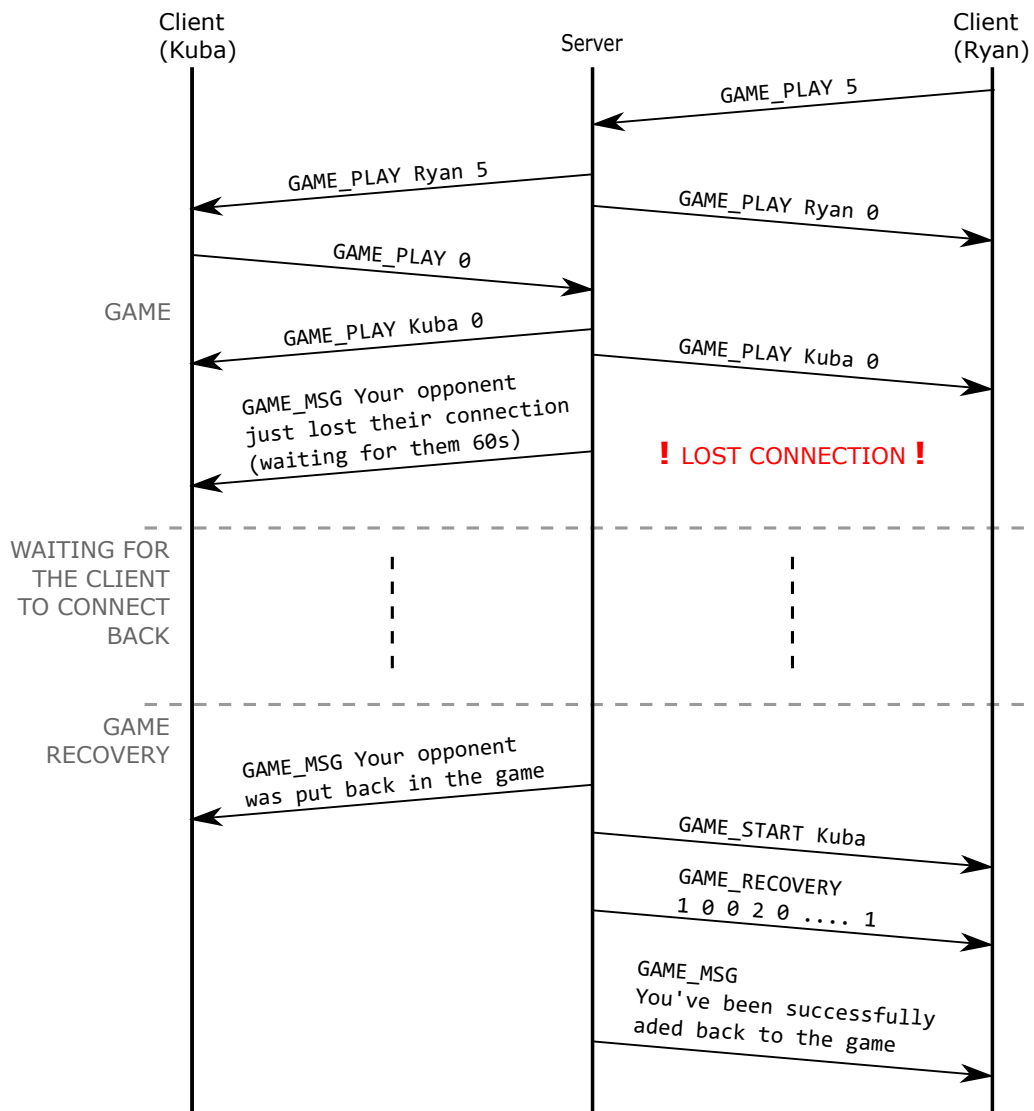Figure 3.8: Sending a `PING` message to the server

Figure 3.9: An example of game recovery when a client loses their connection

In the scenario shown above, one of the client (Ryan) loses their connection while playing a game. The game is recovered for both players as soon as they get connected back to the server.

3.6 Received messages from the server

Table 3.2: List of all the commands the client can receive from the server

Command	Description
`ADD_CLIENT <nick>`	adds a new online client
`REMOVE_CLIENT <nick>`	removes an online client
`GAME_PLAYER_STATE <nick> <ON/OFF>`	changes the state of the client
`RQ <nick>`	received a game request
`INVALID_PROTOCOL <msg>`	not following the protocol
`OK`	acknowledgment
`RQ_CANCELED <nick>`	cancels a game request
`GAME_START <nick>`	start of a new game
`GAME_PLAY <nick> <y> <x>`	places a disk on the board
`GAME_CANCELED`	cancels the game
`GAME_MSG <msg>`	sends a game message
`GAME_RESULT <msg>`	announces a result of the game
`GAME_WINNING_TAILS` `y1 x1 ...  y4 x4`	sends the list of winning disks (tails)
`GAME_RECOVERY` `0 1 0 2 ...  1`	recovers the state of the game

In the table above, the user can see all the possible message they may receive from the server while they are connected to it. Each message depends on the current state of the client (playing a game, received a game request, etc.). The list of all the commands the client can send off to the server can be found here 3.1.

4 Implementation

4.1 Sever

As required, the server was implemented using `C/C++`. The structure of the
server handling all clients connected to the server is shown in the picture
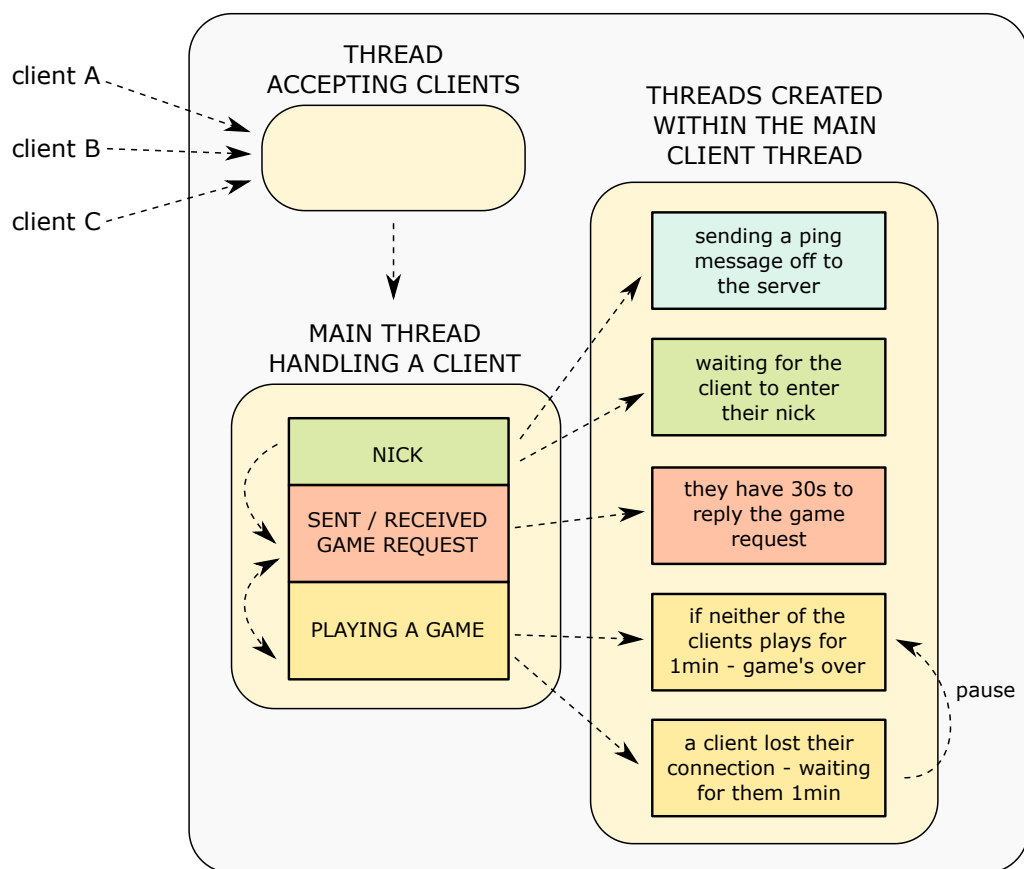below.

Figure 4.1: Structure of the server

When a new client gets connected, the server will create a thread, which
will take care of them. Within this thread, some other threads may be created
as well. For example, a countdown thread that checks if the client entered
their name within the predefined time (3.2) or a thread that deals with `PING`
messages (3.5).

As far the the `UML` class diagram is concerned, it is quite straight forward. When the application starts, it will first attempt to parse arguments the user may have passed on through the terminal, and then it will start the server that accepts new clients. The logic of the game itself as well as all the variables associated with each client was moved into a separate class in order to keep the whole program structured.
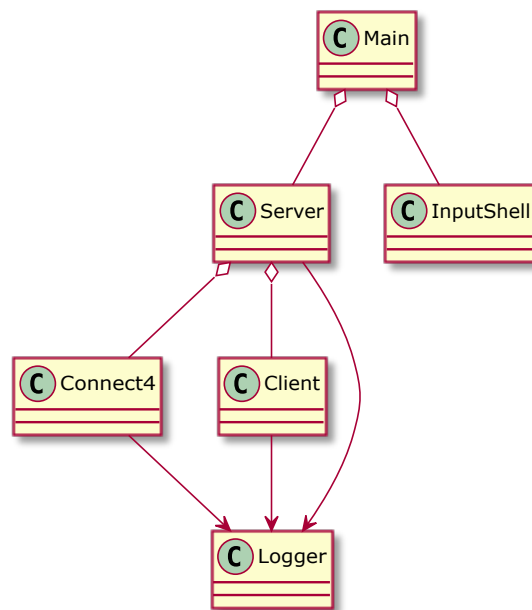

Figure 4.2: UML class diagram of the server

More detailed information on the implementation of the server, such as all the methods and their explanations, can be found in the documentation generated by `Doxygen` - `https://www.doxygen.nl/index.html`.

4.2   Client

The entire client-side of the project was implemented in `Java`, particularly `Java 8`. Since there was supposed to be a graphical user interface as well, I used the `JavaFX` library to meet this requirement. The who strucrure of the client can be seen down below.
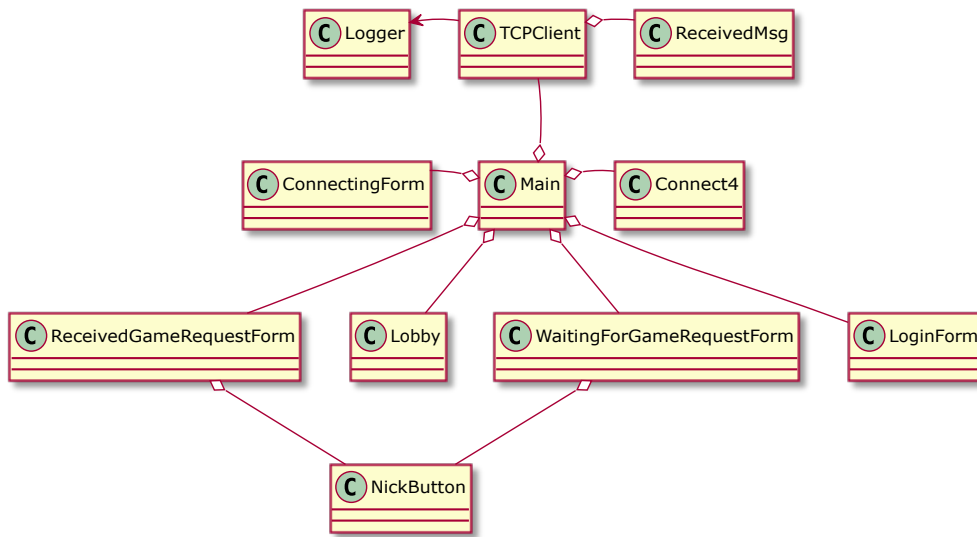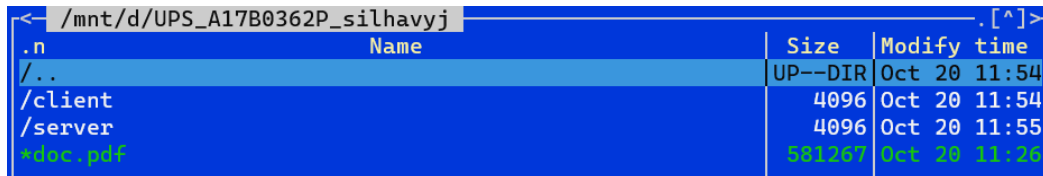
Figure 4.3: UML class diagram of the client

The approach I used for implementing the client is to split the application up into different forms appearing to the user as their current state changes along the way (playing a game, connecting to the server, and so on). The first "line" of classes in the UML diagram shown above is all about the TCP connection to the server, and the rest of them represent individual forms such as the lobby, the game itself, etc. The description of all of these classes can be found in the `Javadoc` documentation. Essentially, there are three threads involved in the program, where the first two take care of the TCP connection (accepting messages and sending off `PING` messages to make sure the server is still up) and the last one is the `JavaFX` application itself.
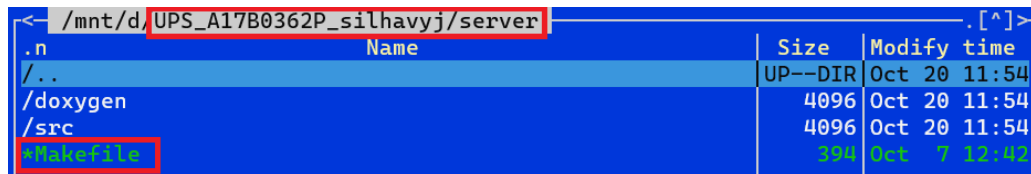
5 Compilation

Figure 5.1: Content of the project folder

5.1 Sever
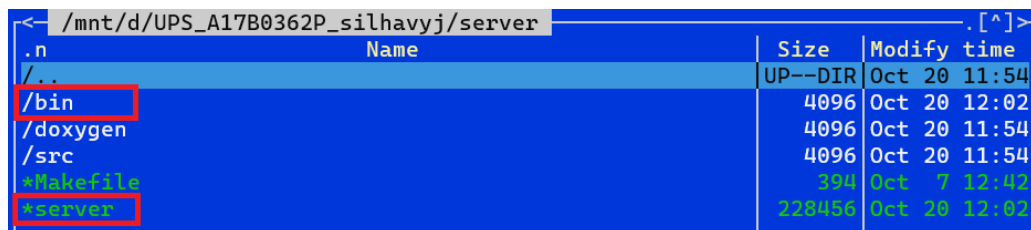
The compilation process is done by `Make` - `https://en.wikipedia.org/wiki/Make_(software)`.

Figure 5.2: Location of `Makefile`

All the user is supposed to do when they want to compile the server of the application is to go to folder `server`, where the `Makefile` is located, and simply run the command `make` from the terminal. Upon successful compilation, a folder `bin` will be created containing all the binary files along with an executable application located in the same folder as the `Makefile`.

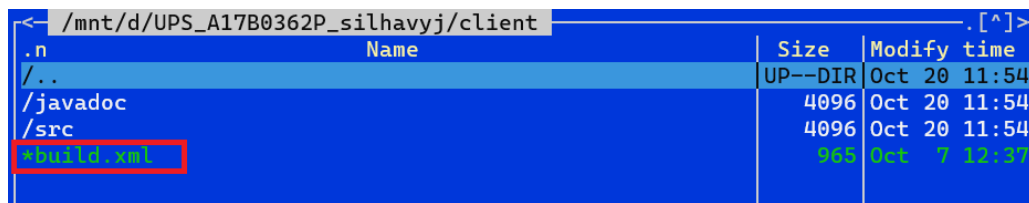Figure 5.3: Files and folders created by the compilation process

5.2 Client

The compilation process of the client does not differ so much from the compilation process of the server. The tool used in this case is called `Ant` and can be downloaded from `https://ant.apache.org/bindownload.cgi`. It is one of the standard ways of compiling and building `Java` applications. In order to create an executable file, first, we need to go to the `client` folder where the `build` file is located.
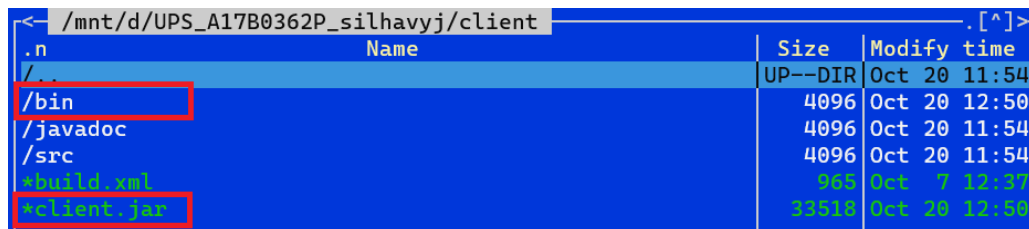

Figure 5.4: Location of `Makefile`

Once in the `client` folder, all we need to do is to run the command `ant` from the terminal. It is important to mention that the user must have `Java 8` installed on their machine. If the compilation is successful, a folder `bin` will be created along with an executable `jar` file located in the same folder as the file `build`.

Figure 5.5: Files and folders created by the compilation process

6 User manual

6.1 Server

Once the server has been compiled (5.1), the user can run it using the command `./server` from the terminal. Additionally, they have the option of changing both the default maximum number of clients and the port number by using the parameters shown in the table below.

Table 6.1: Parameters the user can use when booting up the server

Command	Description
`-p <port>`	changes the default port number
`-c <amount>`	changes the number of clients

However, the user dos not have to necessary use these parameters as there are default valuates defined within the source code. The default port number that the server runs on is 53333 and the default maximum number of client that can be connected to the server at a time is 10. An example of how the user may start the server could be `./server -p 4569 -c 5`

```
kuba@LenovoT540p:/mnt/d/UPS_A17B0362P_silhavyj/server$ ./server
[#41][20-10-2020_16-17-43][BOOTING_LOG] <[STARTING SERVER]>
[#42][20-10-2020_16-17-43][BOOTING_LOG] [port=53333 max clients=10]
[#51][20-10-2020_16-17-43][BOOTING_LOG] creating a socket file descriptor
[#59][20-10-2020_16-17-43][BOOTING_LOG] attaching the socket to the port
[#68][20-10-2020_16-17-43][BOOTING_LOG] binding the server
[#82][20-10-2020_16-17-43][BOOTING_LOG] setting listening
[#119][20-10-2020_16-17-43][BOOTING_LOG] <[ SERVER STARTED ]>
```

Figure 6.1: Starting the server

As soon as the user starts the server, it will start printing out logs of all that is currently happening. All the logs are stored on the disk (`server/log`) as well, so it could be analyzed later on if required.

Figure 6.2: folder containing all the log files

Figure 6.3: Log file named with the date-time when the server started

6.2 Client

The process of running the client is fairly similar to the process of running the server (6.1). The client can be run from the terminal using the following command java -jar client.jar. As soon as the command is executed, a login window will pop up (6.2.1).

6.2.1 Login Form

Figure 6.4: Login window of the client application

20

In the login window, the user is supposed to enter all the information necessary for a successful connection to the server - all these details can be seen in the picture above. The nick is supposed to be only one word, which the user wants to present themselves on the server. The IP address is supposed to be in an `IPv4` format and the port a non-negative integer less or equal than 65535. If any of these details is invalid, an alert message will pop up telling the user they did something wrong. If all three input fields have been filled out correctly when the user clicks on the `Connect` button, the client will start attempting to establish a connection to the server.
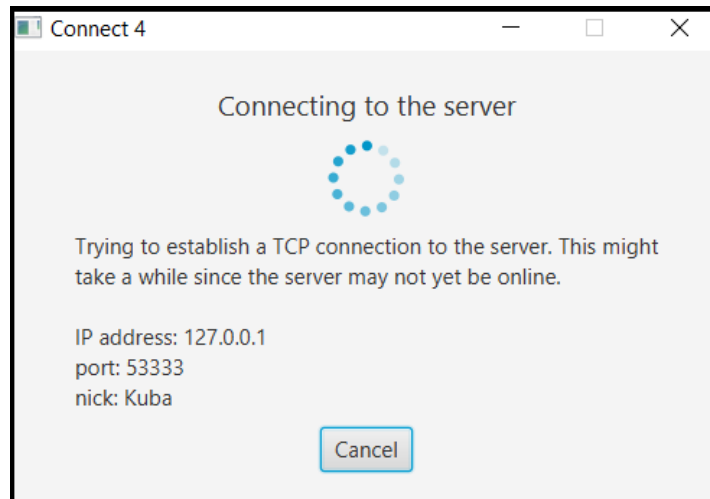
6.2.2 Connecting Form

Figure 6.5: Connecting to the server window

Once the connecting window appears, the user has the option of canceling the connection if they change their mind. Otherwise, they will be moved into the lobby (6.2.3) as soon as the connection has been successfully established. This window will also appear if the server goes offline - the client will automatically try to re-connect back to the server.

6.2.3 Lobby

The lobby is a "room" where the client can see other online clients along with their current statuses. If a client is available, the user can click on the button `Play` in order to send them a game request. If the client is already busy playing a game, their status is turned into an orange circle, and they cannot be sent a game request until they are done playing their current game.
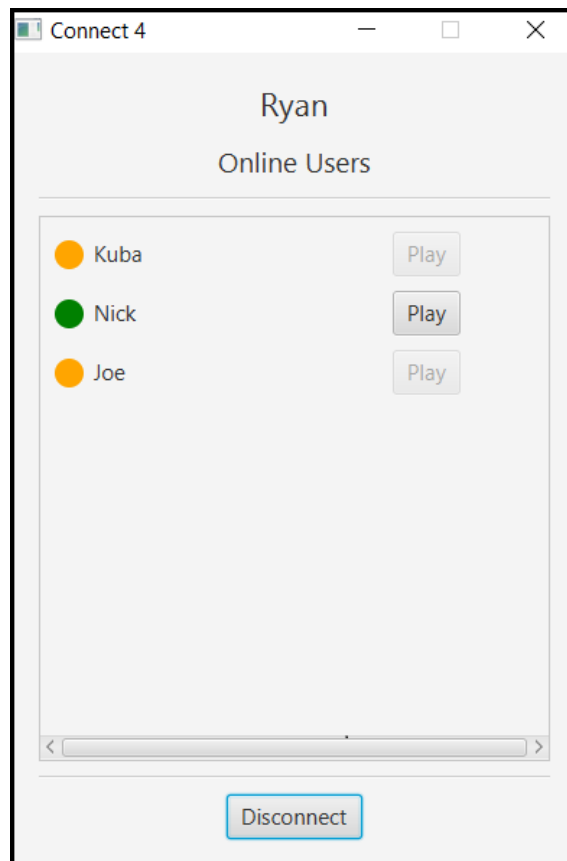

Figure 6.6: The lobby of the game

Once a game request is sent off to another client, the lobby window will disappear and a "waiting" window will show up instead (6.2.4).
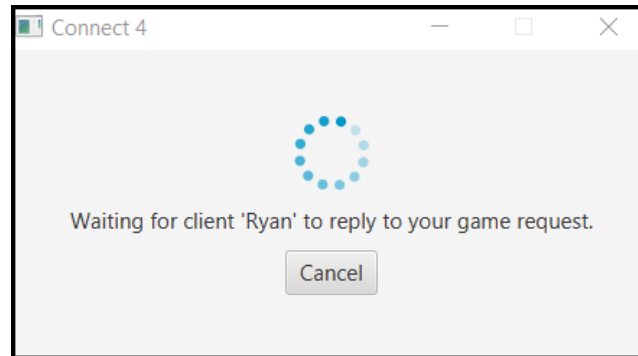
6.2.4 Sent/received a game request

Figure 6.7: Waiting for the other player to reply to to the game reqeust
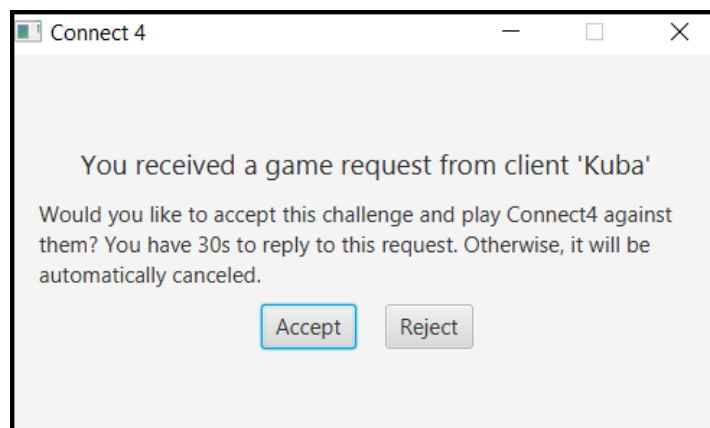

Figure 6.8: The user just received a game request from another client

In both cases, the user has the option to cancel the game request, whether they sent it or received it. If they both agree on playing a game, they will be moved into the game room where they play a game of Connect4 against each other (6.2.5).
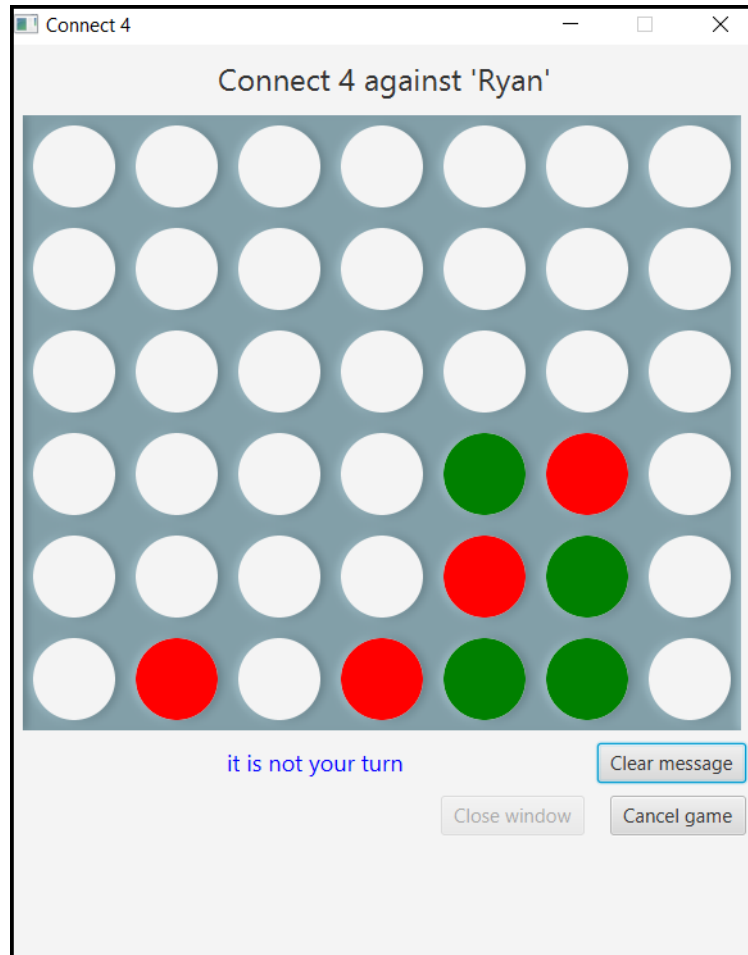
6.2.5 Game

Figure 6.9: The user is playing a game

When looking at the picture above, there is a couple of buttons that might need to be explained. Throughout the game, the user receives a bunch of messages, such as when they try to play and they are not up yet, and they may want to get rid of them once they have already read them. In order to do so, there is a button called `Clear message` that will clear the message displayed next to it. Right below it, there is a button used to cancel the game if the user does not want to play anymore. As soon as the game is canceled, the button `Close window` will be enabled allowing the user to close the window and get back into the lobby (6.2.3).

7 Conclusion

For the initial testing, the application was run only within one local network in order to test the basic functionality such as accepting connections from clients, splitting them up into different game rooms, and so on. Once, this part had proven to work properly, the testing process focused on making the server crash. This involved, for example, "spamming" the buttons in the user application or connecting to the server using `Netcat` and sending messages that did not follow the protocol (3). One particular command came in handy - `cat /dev/random | nc 127.0.0.1 53333`. This command was used in a script that simulated multiple invalid clients attempting to connect to the server at a time (a very simple `DDOS` attack). For testing the game recovery procedure, three computers were hooked up within the same network, one of which ran the server itself and the other two clients. Additionally, after all these tests had been passed, the server was hosted up using `Amazon Web Services https://aws.amazon.com/websites/`, so the application could be tested beyond one single `LAN`. The reason for this was to test out if the server handles different speed connections as well as the game recovery process when the server is not located "right next to all the clients". As a particular example, the server was located in the US, one player in the UK, and the other one in the Czech Republic. Both of the clients tried repeatedly turning their Wifi connection on and off in order to test the process of game recovery. All the tests were passed successfully, and the application met all the requirements mentioned in chapter 1.