Introduction to Computer Networking
Tic-tac-toe Project Report

Franck Duval HEUBA
Student ID: S227629

October 26, 2025

1 Introduction

This project involves the design and implementation of a **concurrent, text-based Tic-Tac-Toe game** using **Java TCP sockets**. The core task is to build a robust server that can manage multiple simultaneous clients and arbitrate games according to a well-defined protocol. The server must support two distinct modes: a single-player mode where a user competes against a bot with random move logic, and a multiplayer mode that pairs two clients for a head-to-head match. The client application will handle user input from the console and communicate with the server to play the game.

2 Software Architecture

There is six main classes in this project:

- **Client** class: this class represent a client
- A **Client** Can either a Connected Client who is using his terminal **TictactoeClient**
- Or a connected client on the server side **ServerClient**
- **ServerGame** is a game created by the server for two player either (ServerClient, Bot) or (ServerClient, ServerClient)
- **TictactoeServer** is the main server class which handle all the connections and create games
- **EventListener** interface is used to handle events between classes

First the **TictactoeServer** is started and listen for incoming connections. When a user want to play, a **TictactoeClient** is created to handle the communication with the server. The **TictactoeClient** will listen for messages from the server and handle them accordingly. The user can then choose to play against a bot or against another player then other instructions. When a client connect (server side) a new **ServerClient** is created to handle the communication with this client. The **ServerClient** will listen for messages from the client and handle them accordingly. When a client want to start a game, the **TictactoeServer** will create a new **ServerGame** for this client. If the client want to play against a bot, the **ServerGame** will set-up the internal bot of the **ServerGame** classes then start the game. If the client want to play against another client, the **TictactoeServer** will wait for another client to connect and then start the game.

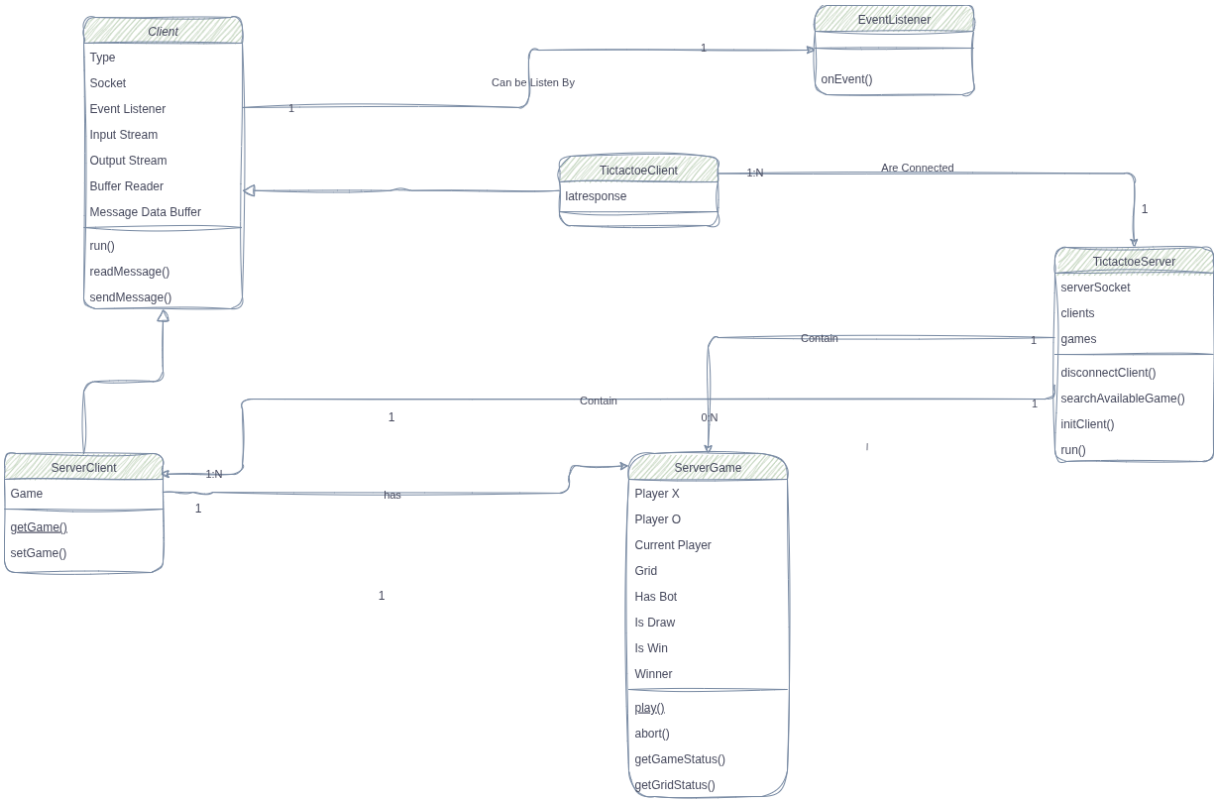

Figure 1: Software Architecture Diagram

Another very important classes but not present in the Diagram is **TictactoeResponse** which is used to handle the messages between the client and the server. A message send by the **TictactoeClient** will be received on the server side by the **ServerClient** and inversely. When a message is received, if the **Client(TictactoeClient or ServerClient)** possess an **EventListener** it will call the appropriate method to handle the message **onEvent(response)**. Any received message is a **response(text)** which is parsed with a **TictactoeResponse** object by his own constructor. The **TictactoeResponse** contains all data of the response.

3 Program logic

The logic of our **TictactoeServer** is quiet simple.

- The main loop is running until the server is stopped by a **Ctrl+C** command or by a defect in the code (which should not normally happen).
- Each iteration of the loop will check for new incoming connections.
- When a new connection is established, a new **ServerClient** is created to handle the communication with this client (**TictactoeClient**) and added to the client's list.
- When a **ServerClient** when a dedicted thread is created to manger all possible operatios with this client.
- A **EventListener** is attached to the **ServerClient** to handle all incoming messages from this client (the methode initClient).
- when a message is received from the client, the appropriate method of the **EventListener** is called to handle the message (onEvent).

- The text message is parsed with a **TictactoeResponse** object to extract all data of the message.
- Depending on the type of message, the appropriate action is taken (start game, make move, etc).
- When the client want to start a game, a new **ServerGame** is created for this client.
- A ServerGame is always added to the list of games in the server.

4 TCP Stream

When a **Client (TictactoeClient or ServerClient)** is created the started method is called. the main loop of the client is running until the client is disconnected. Using **BufferedReader**, which is created in the construcor with **InputStreamReader** and **InputStream**, the main loop of the client will call **readLine()** method to read a line from the buffer reader. When a line is received, $total\_message = temp + received$, where temp is a part of the message which have been received previously. Then we check if the *total_message* contains the end of message character `\r\n` If it does, if the end character is at the end of the message, we will split the *total_message*, then for each part we will give it to the **TictactoeResponse** constructor to parse it. If the end character is not at the end of the message, we split the message in multiple, the we parse each message with a **TictactoeResponse** constructor. excepted for the last part of the message which is stored in temp for the next iteration.

When a **TictactoeResponse** is created, the constructor will use two regex to validate the message. is follow the Tic Tac Toe protocol. The first regex is used to validate the command part of the message, and the second regex is used to validate the puzzle part of the message (if present).

if the received message is neither a valid command nor a valid puzzle, an **TictactoeResponseException** is thrown.

```
1    // command_regex
2    Pattern.compile("^([A-Z]+)(\\s+[A-Z]{2,})?(\\s+[A-Z]{2,})?(\\s+[A-Za
        -z]|\\s+\\-?[0-9]+)?(\\s+[A-Za-z]|\\s+\\-?[0-9]+)?\\s*\\r\\n(\\r
        \\n)?$");
3
4    // puzzle_regex
5    Pattern.compile("^([XxOo\\s]{3}\\r\\n){3}(([XxOo] WON|DRAW|OPPONENT
        QUIT)\\r\\n)?(\\r\\n)?$");
```

Listing 1: Response parsing in TictactoeResponse

5 Multi-threading

Yes my server server enable multiple games (single and multiplayer) to be played simultaneously without interfering with each other. A **ServerGame** is a game between two players (ServerClient, Bot) or (ServerClient, ServerClient). Each **ServerGame** and each client are store in a dedicted list in the **TictactoeServer** class.
A game is a shared object which can only by acceded by the two players (ServerClient) maximun. The main Thread (TictactoeServer) is alone to access to the list of clients and the list of games.

When a client is deconnected, **ServerClient** thrown an exception which is catched by the server object.

6 Robustness

My server is robust because it use the concept of exceptions to handle errors and atomicity. When an error occur, or a bad state, the appropriate exception is thrown by the current class, and catched by the class above. Each class is responsible to handle its own errors and to throw the appropriate exception. For example, when a **TictactoeResponse** is created, if parameters(a message from the client or the server) are not valid, a **TictactoeResponseException** is thrown. This exception is catched by the **ServerClient** or the **TictactoeClient** class which will handle the error accordingly (send an error message to the other side, disconnect the client, etc). This way, each class is responsible to handle its own errors and to throw the appropriate exception.

7 Conclusion

- The most challenging is to catch all possible errors and to handle them accordingly to the context.
- It was the first time I used mutti-threading in Java, so I had to learn how to use it properly.
- I learned how to use Java sockets to create a TCP server and client.
- I learned how to use regex to validate and parse messages.
- I learned how to use exceptions to handle errors and to make my code more robust