Distributed System Project - Server/Client Multiplayer Game

FEUP | CPD - 2022/23

Lia Vieira - up202005042

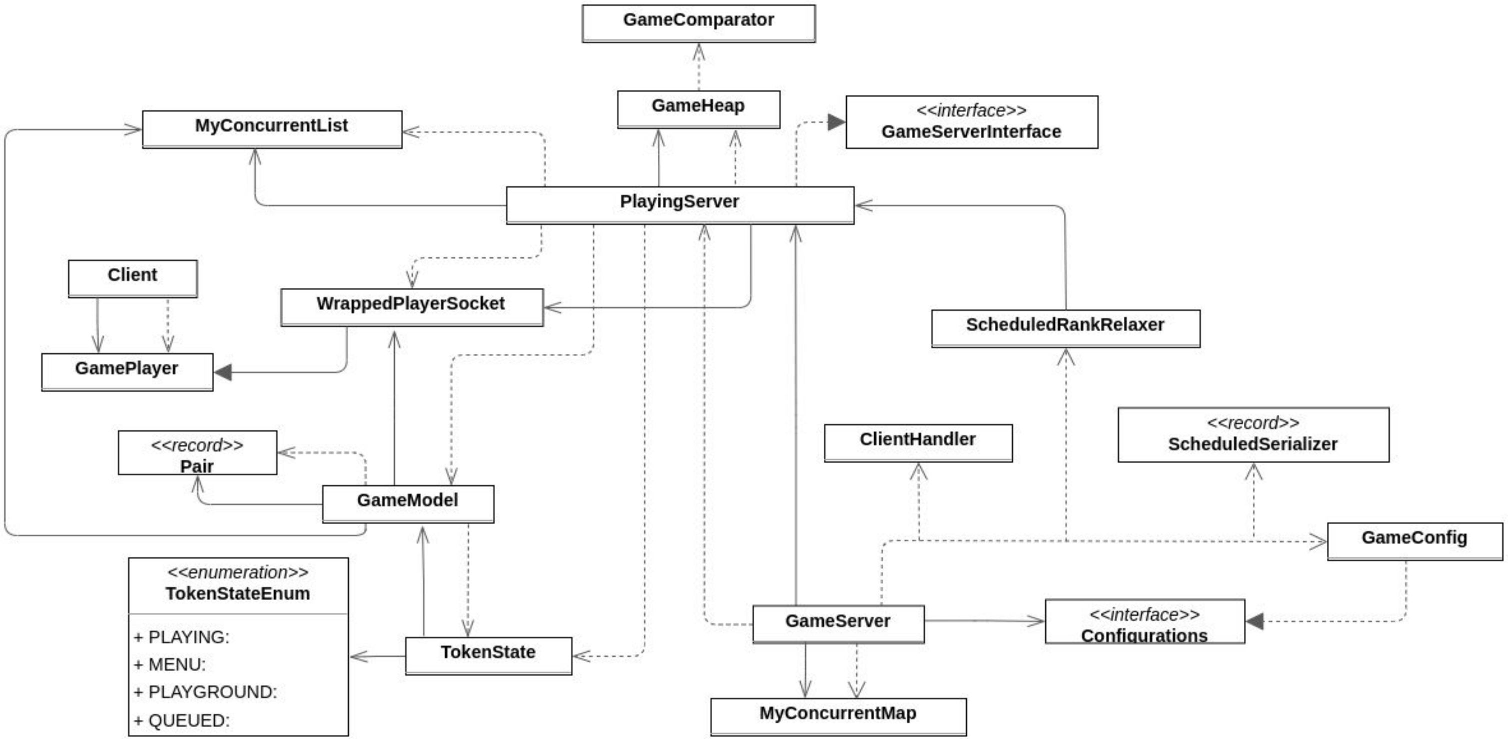
Marco André - up202004891

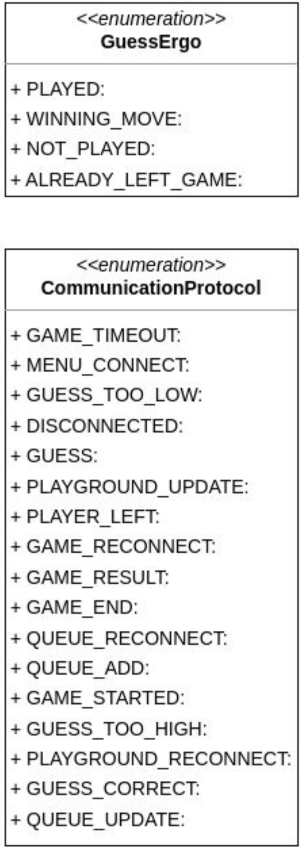
Ricardo Matos - up202007962

Project context and description

- This course project aims to achieve a **client-server multiplayer game** using **TCP** sockets in **Java**. The game must support authentication and a basic game server.
- The connection between them should follow our own communication protocol and **concurrency** should be taken into account with our own locking mechanism.
- Moreover, the developed solution should also be **fault tolerant**, preventing bad experiences from taking place.
- To facilitate the evaluation of the project, the group decided to create a simple **logger** that shows what's happening in the application.

Code Architecture - Class Diagram

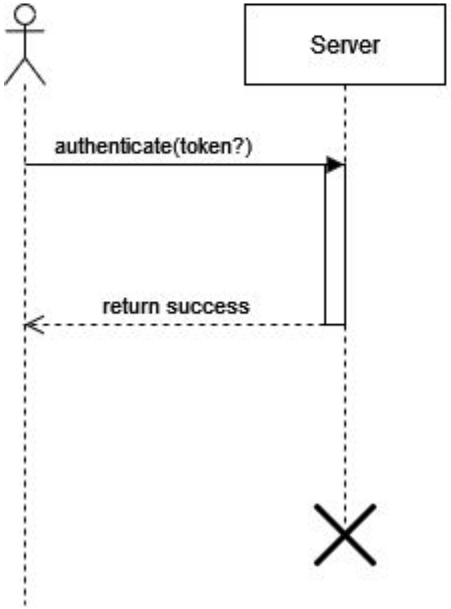


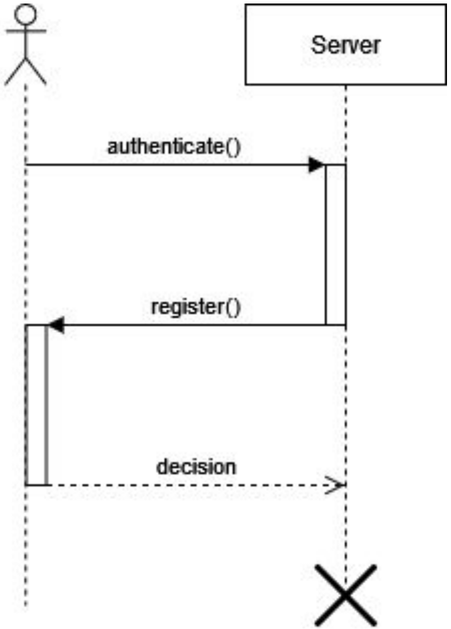


Note: Multiplicities and class contents were omitted for clarity sake

Communication Protocol - Sequence Diagrams

- User has the possibility to logout from the game
- Logout invalidates the token and removes the client state
- Token is **optional**, used to go back to a previous application state on a lost connection
- Token has a timespan



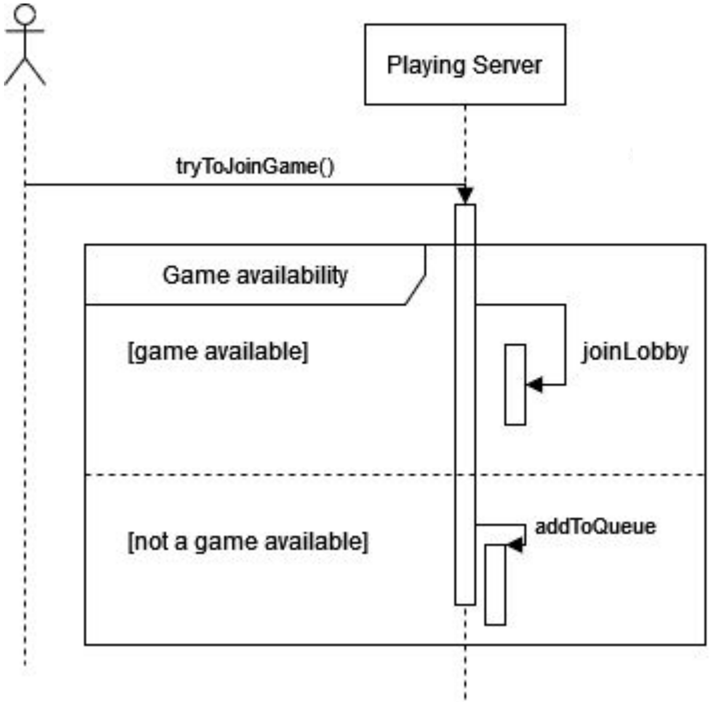


Communication Protocol

- **Schedular rank relaxer** - relaxes tolerances to the players in queue on rank mode to prevent starving of player with very different trophy count from all other currently active players

- Player in lobby receives updates on the current lobby occupation

- On game end, game is responsible to check queue and add to game. Possibly restarting the game

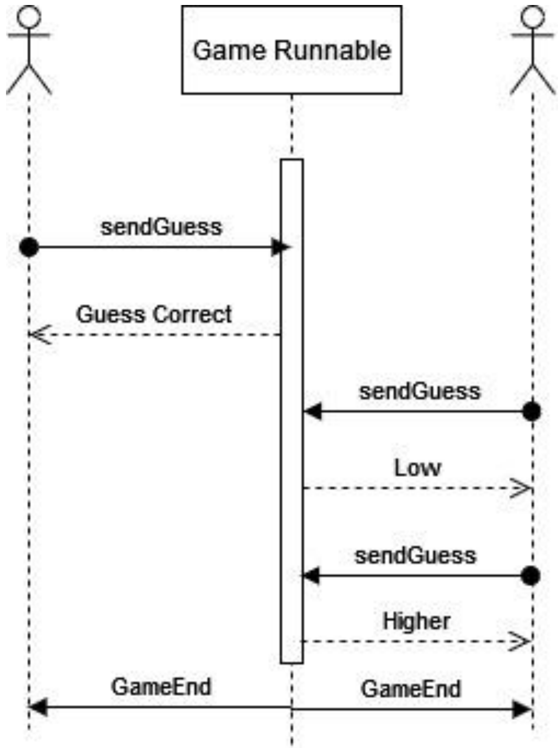


Communication Protocol

- On the right we can see a example of a game play

- A player can reconnect to the game after a failure on the connection

- If the game server stops working, the game can continue as if nothing happened. However, you can lose some progress, check **serverCacheInterval** config.properties



Communication Protocol

- For communication we developed our own *SocketUtils.java* on top of what Java provides;
- We use regular socket communication as well as non-blocking **NIO channels**;
- Connections have a **connection timeout** after a while;
- Finally, we use **RMI** (Remote Method Invocation) on the player side for entering the game;

Server Modes and Configurations

The server supports two different matching modes: simple and ranked mode.

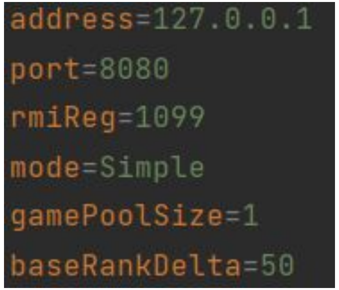
Simple Mode

- In this mode, a player is assigned to the first game in the heap (game with most players that stills hasn't started)
- Players in queue enter game **in order** of arrival

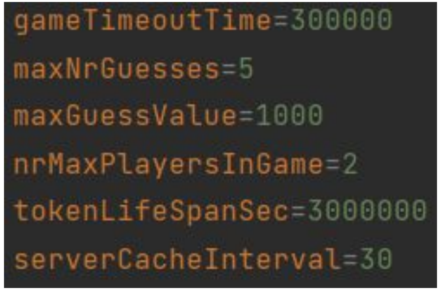
Rank mode

- Player is assigned to the first non-started game that is close enough in trophy range (has its rank tolerance met)
- Player in queue, have their tolerance relaxed over time to prevent **starvation**

Our game's configuration is all done on a single *config.properties* file (to the right).



```
address=127.0.0.1
port=8080
rmiReg=1099
mode=Simple
gamePoolSize=1
baseRankDelta=50
```

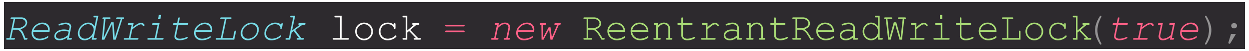


```
gameTimeoutTime=300000
maxNrGuesses=5
maxGuessValue=1000
nrMaxPlayersInGame=2
tokenLifeSpanSec=3000000
serverCacheInterval=30
```

Concurrency

All established requisites were met regarding concurrency access on the server side (as the client has no need for such measures).

We used *java.util.concurrent.locks* like shown below to manage all data structures and files on the server side:



```
ReadWriteLock lock = new ReentrantReadWriteLock(true);
```

Here are the most important uses of multiple threads in our project:

- The server has 5 fixed thread pools with a game each;
- Each new connected player is given by the server its own clientHandler;
- When playing, the gameServer cycles between all players to allow for a non-turn based experience;
- The client has a separate thread exclusively dedicated to detect if “exit” is typed by a user (in the queue to play);
- Scheduler thread to periodically serialize the current server state to a cache file;
- Scheduler thread to relax trophy delta over time to prevent player starvation;

Fault tolerance

The solution developed was built to be resilient to faults both on server and client side.

Server side

- Serialized object on small intervals and brought back on restart if cache file is present

Client side

- Token with customizable expiration time used for authentication on connection to server

If the client fails, it sends its token on a new authentication and the server will check if the player had previous status to recover and informs the client to move to that state.

Conclusion

- In conclusion, this project aimed to provide a robust client-server system for simple online gaming, allowing users to play with others.

- By implementing key functionalities such as authentication, matchmaking, fault tolerance and optimized concurrency, we intended to deliver a simple and intuitive game experience while ensuring system efficiency and scalability.

End

FEUP | CPD - 2022/23

Lia Vieira - up202005042

Marco André - up202004891

Ricardo Matos - up202007962