Distributed Systems Assignment

CPD, 2023/2024

8 April

1 Introduction

You were just hired for an internet online game company and your first task is to create a client-server system using TCP sockets in Java. The company has users that can authenticate with the system in order to play some text based game among them. A game is handled by a class and requires a given number of connected players to start. E.g. for **chess** two players are enough, but you should allow for any number of players. Lets assume size is n.

```
public class Game {
    private List<Socket> userSockets;

    public Game(int players, List<Socket> userSockets) {
        this.userSockets = userSockets;
    }

    public void start() {
        // Code to start the game
        System.out.println("Starting game with " + userSockets.size() + " players");
        ....
    }
}
```

Users connect to the system using some Java client code that allows users to first authenticate and then enter a queue for the game and finally play the game until it completes.

The server needs to accept and authenticate user connections, and make teams of n users for the game. Once the team is formed a thread can be requested from a thread pool (e.g. `Executors.newFixedThreadPool(5);`) to create and animate a new Game instance. The configuration of the thread pool defines the maximum number of games that can run concurrently in the server.

2 Barrier and Matchmaking

Your game server should provide two alternative operating modes when launching.

Simple In this simple mode the first n users that connect to the system are assigned to the first game instance and so forth for the next batches of n users.

Rank The system keeps a level per user that is based on the scores that user got in each game it played. (When a game ends, each player should have its level updated.) In rank mode, the matchmaking algorithm should try to create teams whose members have similar levels; the idea is to have teams of players with similar playing skills. The difference between the levels of the players in a team can be relaxed as time passes so that users do not stay forever waiting for a game.

3 Fault Tolerance

Your implementation should tolerate broken connections when users are queuing and waiting for the game to start. Try to devise a protocol between client and server that allows the clients to not lose their position in the game wait queue when resuming broken connections (Hint: Assign each player a token.)

4 Concurrency

This is one of the most important grading criteria of your assignment. Your design should strive for the following goals:

No race conditions in the access to shared data structures or in synchronization between threads. You cannot use thread safe implementations of the Java collection classes that belong to `java.utils.concurrent`. Instead, you should design your own implementations using the classes of `java.utils.concurrent.locks`

Minimize thread overheads i.e. avoid thread creation/termination overheads. In addition, you should minimize the number of threads, which is particularly important in Java, if you want to scale to hundreds of thousands of connections, because each Java thread is mapped to one kernel thread. (Hint: take a look at the multiplexed non blocking channels of `java.nio.channels`)

Avoid slow clients from bringing the system to a crawl (except in the game).

5 Game

You can implement whatever game you wish, but its weight in the final grade is residual. If you wish you can just emulate a round based game. For example, in each round, a player sends a message to the server, and the server replies with another message. After a few rounds, which may be random, the server may terminate the game and report back the outcome to each of the players. Furthermore, the server should also update the level of the players in the team.

6 User registration and authentication

You may provide a (sub) protocol for user registration. Furthermore, you may also persist the registration data in a file. Alternatively, you can provide a file with registration data.

You may want to decouple authentication from game playing. I.e. allow for a player to send its username and password once, and then playing a sequence of game instances.

7 Implementation and Submission

Your system must use TCP as the transport protocol.

You should implement this client-server system in Java SE 17 (or more recent). You cannot use packages that do not belong to Java SE.

You should use your CPD group repository at Gitlab@FEUP to submit the code as well as a README file with instructions about how to run the server and the clients. Submission deadline is at 18th May 2024.

8 Testing and Demonstration

It may be easier for testing and demonstration purposes, if you persist both the server and the client state in non-volatile memory. E.g., by terminating the client and restarting it, you can show that the server tolerates broken network connections.

Testing and demonstration will also be simpler, if your system tolerates broken connections. E.g., you can show registration by running the client once. Then you can show authentication, by running the client once more (with appropriate command line arguments). Then you can show queuing/waiting and game playing.

9 Grading Criteria

Your grade will depend mostly on your design choices, implementation and demonstration. The ceiling of your grade depends on the operating mode. If you demonstrate only **simple** mode operation, the ceiling of your grade is 16 points (in 20).