



UNIVERSIDADE DO MINHO

03-01-2018

# Relatório

Sistemas Distribuídos

**Trabalho elaborado por:**

Carlos Pedrosa - a77320

David Sousa - a78938

Manuel Sousa - a78869

**Grupo 20**

# Introdução

No âmbito da unidade curricular de Sistemas distribuídos foi proposto aos alunos que desenvolvessem um Matchmaking de um jogo online. Nesse sentido, o projeto tem como principal objetivo fazer com que haja um contacto mais prático no que toca a sistemas distribuídos.

Consideramos que o projeto se encontrava dividido em duas partes. Uma primeira parte em que se deve controlar os acessos aos diversos recursos partilhados e uma segunda parte que permitisse assegurar a presença de vários jogadores na aplicação. Assim, o projeto desenvolvido deverá simular corretamente a escolha de personagens por parte dos vários clientes.

## Conceção da solução:

O nosso projeto encontra-se dividido em algumas classes que pretendem dar resposta a todos os objetivos propostos.

- **Utilizador:**  
Contém toda a estrutura necessária à inicialização do utilizador.
- **GameServer:**  
Classe principal. Nesta classe é invocada a classe necessária a todos os procedimentos necessários que se prendem com o começo do jogo. Alguns métodos desta classe estão protegidos relativamente aos vários acessos pois estão escritos com a primitiva *synchronized*. A título exemplificativo poderíamos ter um jogador a fazer login ao mesmo tempo o que não deve ser possível pois iria se perder consistência.
- **Jogador**  
Classe que gere os clientes do sistema.
- **Jogo**  
Classe que implementa todos os mecanismos necessários ao decorrer de um jogo. É nesta classe que se trata por exemplo da parte em que o jogo “decorre” (gerando os vencedores automaticamente) e ainda é nesta classe que se faz o controlo do tempo proposto pelo enunciado.

```
this.players = new ArrayList<>(10);  
this.players.add(ordered.get(0).getUsername());  
this.players.add(ordered.get(1).getUsername());  
this.players.add(ordered.get(2).getUsername());  
this.players.add(ordered.get(8).getUsername());  
this.players.add(ordered.get(9).getUsername());  
this.players.add(ordered.get(3).getUsername());  
this.players.add(ordered.get(4).getUsername());  
this.players.add(ordered.get(5).getUsername());  
this.players.add(ordered.get(6).getUsername());  
this.players.add(ordered.get(7).getUsername());
```

Ordena os jogadores de acordo com o seu rank e coloca-os num ArrayList.

```

this.team1 = new HashMap<>();
this.team1.put(ordered.get(0).getUsername(), null);
this.team1.put(ordered.get(1).getUsername(), null);
this.team1.put(ordered.get(2).getUsername(), null);
this.team1.put(ordered.get(8).getUsername(), null);
this.team1.put(ordered.get(9).getUsername(), null);

```

```

this.team2 = new HashMap<>();
this.team2.put(ordered.get(3).getUsername(), null);
this.team2.put(ordered.get(4).getUsername(), null);
this.team2.put(ordered.get(5).getUsername(), null);
this.team2.put(ordered.get(6).getUsername(), null);
this.team2.put(ordered.get(7).getUsername(), null);

```

De forma a construir jogos que possuísem ranks justos para todos os jogadores, decidimos criar uma espécie de mediana com os jogadores existentes no HashMap. De notar que este método é sempre aplicado a duas equipas.

```

for(String st : this.buffs.keySet()) {
    buffs[p] = new Thread(new TalkToPlayers(this.buffs.get(st), s));
    buffs[p].start();
    p++;
}

```

```

Thread[] threads = new Thread[10];
int t = 0;
for(t = 0; t < 10; t++){
    BufferedReader bR = this.buffsIn.get(this.players.get(t));
    threads[t] = new Thread(new JogoIn(bR, this, this.players.get(t), t));
    threads[t].start();
}

```

```

try {
    while ( endTime - startTime <= 10000 ){
        endTime = System.currentTimeMillis();
    }

    boolean todosJogam = todosTemHero();
}

```

Cria as diferentes threads para permitir a comunicação entre os vários jogadores e o servidor.

### ➤ Jogoin

Permite ainda a gestão correta da escolha dos heróis. Deste modo, esta classe possui vários métodos sincronizados não permitindo que vários jogadores da mesma equipa escolham o mesmo herói. Para além disto, é ainda nesta classe que se faz a comunicação entre os vários jogadores e o servidor (mais intrinsecamente).

### ➤ Players

Classe criada apenas para fazer uma gestão correta dos mecanismos de concorrência relacionados com o jogador, pelo que guarda os *BufferedReader* e os *BufferedWriters* dos jogadores que ainda não se encontram em jogo.

### ➤ TalkToPlayers

Classe que trata da questão de escrita para os outros clientes de acordo com os seus *BufferedWriters*.

```
public class TalkToPlayers implements Runnable{

    private BufferedWriter buW;
    private String s;

}

    public TalkToPlayers(BufferedWriter buW, String s){
        this.buW = buW;
        this.s = s;
    }

    public void run(){
        try {
            this.buW.write(this.s);
            this.buW.newLine();
            this.buW.flush();
        } catch (IOException ex) {
            Logger.getLogger(TalkToPlayers.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

### ➤ UsersWannaPlay

Classe criada apenas para fazer uma gestão correta dos mecanismos de concorrência relacionados com o jogador que ainda não se encontram em jogo.

```
public class UsersWannaPlay {
    HashMap<Integer, List<String>> usersToPlay;
    HashMap<String, Utilizador> users;
    ReentrantLock lock;

    public UsersWannaPlay(HashMap<String, Utilizador> users) {
        this.usersToPlay = new HashMap<Integer, List<String>>();
        this.users = users;
        this.lock = new ReentrantLock();
    }

    public HashMap<Integer, List<String>> getUsersToPlay() {
        return this.usersToPlay;
    }

    public ReentrantLock getLock() {
        return this.lock;
    }
}
```

Criação de lock para garantir um acesso correto ao map e, deste modo, bloquear apenas esta classe em particular.

➤ **Worker**

Classe principal que invoca o menu e trata de invocar a classe jogo que será capaz de tratar os mecanismos necessários à resposta dos diferentes objetivos.

```
this.server.getUsersToPlay().getLock().lock();

if (this.server.getUsersToPlay().getLength(rank - 1) < 10 && this.server.getUsersToPlay().getLength(rank) < 10) {

    this.server.getUsersToPlay().addUserToPlay(rank, username);

    this.out.write("Infelizmente não existem jogadores suficientes para iniciar uma partida.\n" +
        "Aguarde...");
    this.out.newLine();
    this.out.flush();
}
```

É necessário usar lock pois pode existir concorrência no HashMap utilizado pois pode existir concorrência neste e queremos garantir que apenas se obtenha um estado consistente.

# Conclusão

De acordo com o objetivo do projeto, concluímos que este foi atingido. No entanto, no decorrer do trabalho deparamo-nos com diversos desafios que na sua grande maioria foram ultrapassados.

Apesar de todos os nossos esforços alguns desafios apenas foram ultrapassados parcialmente. De facto, o fator trabalho-tempo influenciou negativamente o trabalho. Assim, no decorrer do projeto a falta de primitivas que permitissem o término correto das threads por nós criadas revelaram-se uma barreira no que diz respeito à conceção do trabalho. Para além disto, o uso do método *readLine()* também por si só também conduziu a vários estados inconsistentes. No entanto, as funções básicas que envolvem a construção dos utilizadores, a correta simulação do jogo entre outras metas base foram atingidas.

Assim, à medida que o trabalho ia sendo concebido, fomos aprofundando os nossos conceitos no que diz respeito a threads, bem como a mecanismos de lock e unlock e ainda conceitos relativamente a Sockets TCP-IP obtidos aquando a construção de um cliente e de um servidor.