Universidade do Minho

LEI 3ºAno 2ºSemestre

Comunicações por Computador

Trabalho Prático Nº2

Desenho e implementação de um jogo distribuído na Internet

Carlos Morais a64306
Daniel Caldas a67716
Xavier Fernandes a55838

9 de Junho de 2015

Conteúdo

Resumo

O presente relatório destina-se à explicitação das decisões tomadas, no decorrer do desenvolvimento do jogo distribuído. Iremos abordar com extremo detalhe a **arquitetura** da solução implementada, explicar em que contexto e como foram utilizados os protocolos de comunicação que integram a nossa aplicação. Iremos numa fase mais terminal apresentar casos práticos da utilização da aplicação e finalmente apresentaremos as conclusões e trabalho futuro, resultantes do trabalho efetuado ao longo do projeto.

1 Introdução

No contexto da UC Comunicações por Computadores foi proposto ao grupo de trabalho que se desenvolvesse um jogo distribuído. O grupo escolheu a linguagem ***JAVA*** para desenvolvimento da aplicação. O jogo terá como objetivo principal o uso de protocolos de comunicação, nomeadamente **UDP**, na troca de dados entre clientes e servidor (por forma a simplificar o processo foi decido que para a comunicação entre servidores fosse utilizado o protocolo **TCP**). A nossa aplicação funciona num contexto local da nossa rede (N Clientes 1 Servidor). Ao longo deste relatório, serão certamente expostas as dificuldades que surgiram quando tentamos implementar uma aplicação usando o protocolo **UDP**.

2 Protocolo

Relativamente ao protocolo utilizado para troca de mensagens dentro da aplicação, não foram feitas alterações significativas relativamente à proposta feita no enunciado do projeto.

3 Arquitetura e detalhes de implementação

3.1 Nota introdutória

Nesta secção do nosso relatório, de apresentar a aquitetura das aplicações servidor e cliente. A essência ou esqueleto das aquiteturas para os dois lados da aplicação são idênticas, no entanto existem algumas diferenças pontuais que como iremos ver, pela sua natureza, nos obrigam a separar a explicação das duas partes.

3.2 Visão geral

Como é habitual numa aplicação distribuída, este projeto terá uma aplicação que irá correr no contexto do servidor (**QuizServer**) e uma aplicação que irá correr na máquina do cliente (**QuizClient**).

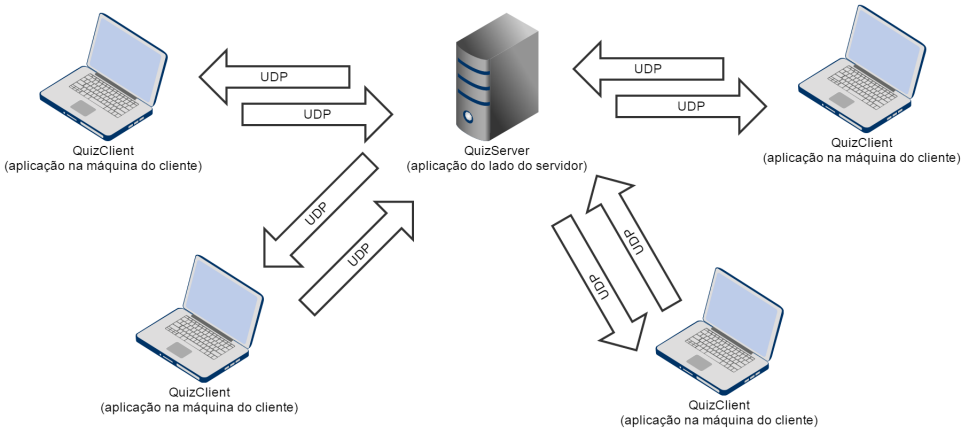

Figura 1: .

3.3 Servidor

Embora não seja explícito pelo número de packages, a aplicação servidor está dividida em três camadas lógicas: **dados**, ***middleware*** e **negócio**.

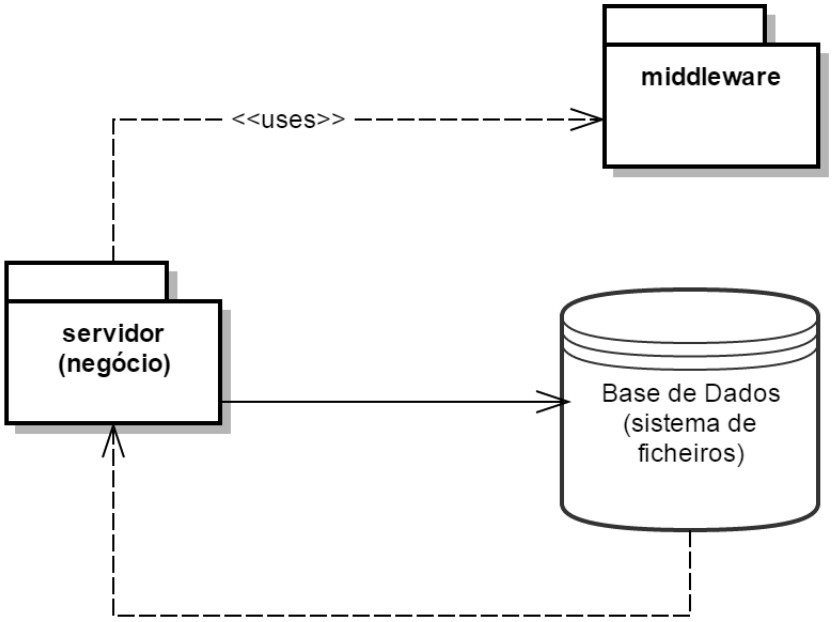

Figura 2: Arquitetura do código do lado do servidor.

- **Base de dados**: corresponde ao **sistema de ficheiros** já explicitado anteriormente neste relatório.
- **Negócio**: é fisicamente representado pelo *package* **server** , contém o código que faz gestão de recursos (Manager), procedimentos de atendimento dos clientes (*ClientStub*), gestão de desafios (*ChallengeThread*) entre outros que desempenham funções semelhantes.
- **Middleware**: é fisicamente representado pelo *package* **middleware** este package contém as funcionalidades que são partilhadas tanto pela aplicação servidor com pela aplicação cliente.

3.4 Middleware

Figura 3: Analogia de um puzzle com a função do package middleware.

Apesar de o nome das classes e funcionalidades serem semelhates dos dois lados da aplicação existem ligeiras alterações a nível dos métodos implementados (como iremos ver mais à frente neste relatório), mas é o package middleware que permite o encaixe (**merge**) lógico das aplicações, ou seja existe uma concordância entre aplicação cliente e servidor acerca de ***qual a língua que ambos falam***, isto é ambos têm de possuír mecanismos que permitem às aplicações comunicar segundo o protocolo de comunicação **UDP** e segundo o protocolo pré-definido para reconhecimento de pacotes de dados no contexto da aplicação. Se tivessemos de fazer uma analogia da função deste package com a realidade, diríamos que o package se trata da peça **cinzenta** do puzzle na figura.

4 Primeira abordagem - Servidor

Num projeto com dimensões consideráveis como é caso, é importante fazer-se a modelação do problema, orientado às tecnologias que iremos utilizar. Numa primeira abordagem iremos então tentar desenhar duas aplicações que troquem entre si pacotes **PDU**, uma correspondente ao **servidor** outra ao **cliente**, mas numa primeira fase dá-mos previlégio ao desenvolvimento da aplicação da parte do servidor, e à implementação de mecanismos de comunicação.

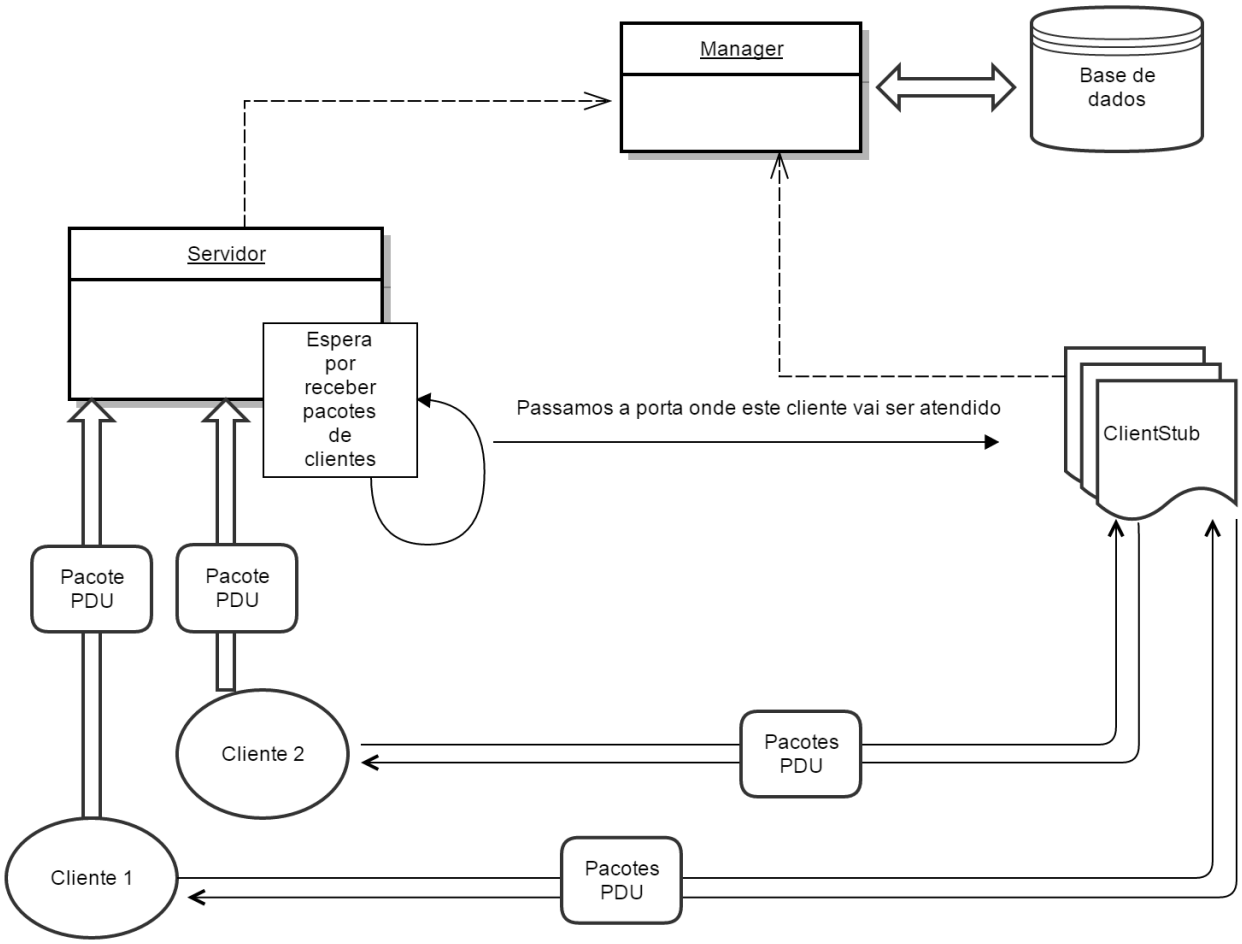

Figura 4: Esquematização da **arquitetura** base da aplicação e **fluxo**.

Com base no esquema da figura 4, podemos agora explicar com mais detalhe as funcionalidades até aqui implementadas, e as funções que desempenham cada um dos componentes do esquema.

4.1 Servidor

O servidor é então a entidade, portal para a aplicação do lado do servidor, isto é, no servidor estamos continuamente à escuta clientes que tentam comunicar com este (daí no esquema a seta que aponta para ele mesmo).

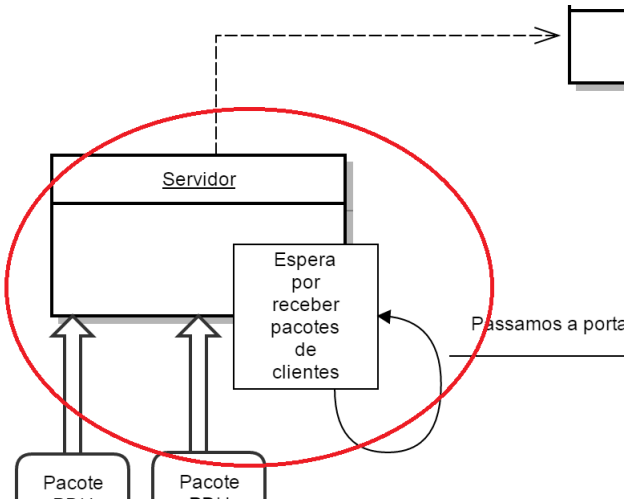

Figura 5: Servidor.

Sempre que um cliente envia o primeiro pacote de dados ao servidor o servidor simplesmente cria uma ***thread*** que irá tratar de atender esse mesmo cliente, deste modo o servidor fica então disponível para poder atender mais clientes. Veremos de seguida como se comporta essa thread dedicada a um cliente. De notar que para essa thread são passados os seguintes argumentos:

- **pacote PDU**: primeiro pacote PDU enviado pelo cliente assim como os seguintes que será processado no **ClientStub**;
- **porta**: a nova porta de atendimento, onde futuramente o cliente será atendido, isto no contexto da nova thread;
- ***manager***: este é o objeto partilhado (instanciado pelo servidor) que circula em todas as threads a correr no servidor, mais à frente iremos vê-la em detalhe.

4.1.1 Servidor.java

A classe que espelha as funcionalidades descritas na subsecção anterior é a classe **Servidor.java**.

```
16  public class Servidor {
17      // Variáveis de classe
18      private static Manager manager;
19
20      public static void main(String args[]) throws IOException, ClassNotFoundException{
21          int port = 9877;
22          int count = 0;
23
24          //manager = Manager.loadManager();
25          //if(manager==null)
26          manager = new Manager();
27
28          try{
29              DatagramSocket serverSocket = new DatagramSocket(port);
30              /*FICA A RECEBER NOVOS CLIENTES*/
31              while(true){
32                  count++;
33                  //em principio são mais que suficientes 1024 bytes
34                  byte[] receiveData = new byte[1024];
35                  DatagramPacket receivePacket = new DatagramPacket(receiveData,receiveData.length);
36                  serverSocket.receive(receivePacket);
37
38                  //verica que recebe o PDU HELLO
39                  byte[] dados = receivePacket.getData();
40                  PDU msg = (PDU)Utils.fromBytes(dados);
41
42                  if(msg.getTipo() == 1){
43                      // Lançar thread para atender cliente na qual são passados os parâmetros:
44                      // - receivePacket: Pacote de dados recebido (primeiro pacote enviado pelo respetivo cliente)
45                      // - port+count: Porta na qual esse cliente será atendido
46                      // - manager: Objeto partilhado do sistema que contém informação que pode vir a ser necessária
47                      ClienteStub c = new ClienteStub(receivePacket, port+count, manager);
```

Figura 6: Exercerto do código de **Servidor.java**.

Como podemos observar temos no servidor a instância do **objeto partilhado *Manager***, que é passado como parâmetro para todas as threads que atendem um cliente (**fig. 6 linha 47**). Podemos também ver a implementação do ciclo que faz com que o servidor fica continuamente à espera de clientes. A porta para atender o primeiro cliente será por defeito a porta passada como argumento na sua execução, este valor é incrementado de cada vez que um novo cliente estabelece conexão, sendo a porta do próxima cliente dada por *port+count* (**fig. 6 linha 47**).

4.2 PDU

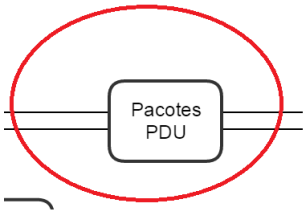

Figura 7: PDU.

Na figura 7 temos o componente PDU, que é o objeto que encapsula as mensagens trocadas entre servidore e clientes, com base numa estrutura pré-definida e separada por **lógicos de dados**. É neste componente que criamos os pacotes de dados, através da chamada de métodos visíveis do exterior que sendo invocados, alteram a estrutura interna da instância **PDU** por forma a criar o pacote de dados do tipo pretendido.

4.2.1 PDU.java

```
public class PDU implements Serializable {

    private int versao = 0; //acho que pode/deve ser atribuida aqui
    private int seg = 0; //acho que pode/deve ser atribuida aqui
    private int label;
    private int tipo;
    private int nCampos;
    private int tamanho;

    private byte[][][] buffer;

    public PDU(int label) {
        this.label = label;
        this.buffer = new byte[256][][];
        this.tamanho = 0;
        this.nCampos = 0;
    }
```

Figura 8: Excerto do código de **PDU.java**.

Na fig. 8 podemos observar a estrutra base comum a todo o pacote de dados PDU. Os campo base definidos permitem determinar algumas caracteristicas importantes da PDU.

4.2.2 Buffer

Para que a PDU possa tranportar informação foi implementado um buffer que é um array de array de arrays de bytes onde estão contidos os dados da mensagem enviada. Embora a implementação de um, byte [ ][ ][ ], possa numa primeira abordagem não fazer sentido, esta implementação permite que sejam guardados em bytes vários dados para diferentes campos de dados de uma forma organizada. A sua implementação permite assim, com a primeira posição definir um determinado campo, com a segunda posição definir uma lista valores e na terceira posição é onde vão os dados em bytes. Com isto é possível definir a posição 1 do buffer para representar nomes de jogadores, e guardar vários nomes. Por exemplo se quisermos enviar na mesma mensagem, os nomes "André André"e "Ana Malhoa"correspondentes a jogadores diferentes, a informação seria guardada da seuinte forma(em bytes) no PDU:

```
buffer[1][0] = "Andre Andre";
buffer[1][1] = "Ana Malhoa";
```

É assim possivel enviar vária informação na mesma PDU mantendo o protocolo inicialmente definido.
A PDU dispõe também de métodos que permitem adicionar/obter os respetivos dados. Existem vários métodos para cada campo de dados, é possível inserir e obter o nome de um jogador numa PDU, assim como inserir e obter uma lista dos nomes dos jogadores numa PDU.

4.3 ClientStub

Aqui já falamos apenas no contexto de um cliente. Esta será a principal **thread** que irá comunicar com o cliente, trocando pacotes PDU, e fazendo as operações de **marshalling** e **unmarshalling** dos

pacotes de dados, vem como a sua interpretação, pois um pacote de dados é criado com base no protocolo pré-estabelecido e por nós seguido.

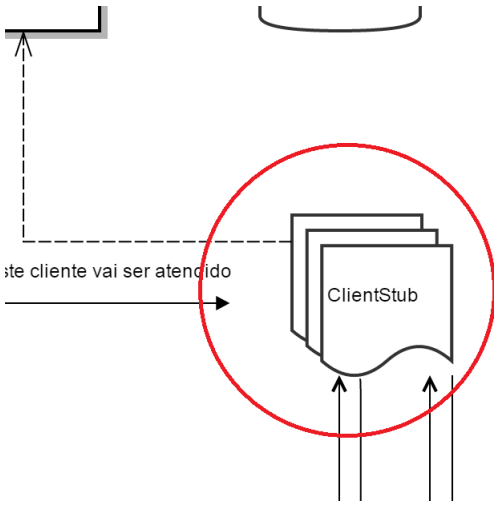

Figura 9: ClientStub.

Como iremos ver de seguida, para além de recebermos os pacotes de dados neste componente, o *ClientStub* tem também a função de fazer chamadas ao objeto partilhado, por forma a inserir ou pesquisar informação, acerca do utilizador diz respeito à ligação em causa.

4.3.1 ClientStub.java

A classe que espelha as funcionalidades descritas na subsecção anterior é a classe **ClientStub.java**.

```
22  public class ClienteStub extends Thread{
23      public static final int MAXSIZE = 1024;
24
25      private DatagramPacket receivePacket;
26      private boolean ok = true;
27      private InetAddress IPAdress;
28      private int MyPort;
29      private int ClPort;
30      private Manager manager;
31      private DatagramSocket serverSocket;
32      private String currentUser; //para saber qual o Id do Cliente a ser atendido
33
34      private Lock lock;
35
36      public ClienteStub(DatagramPacket r, int port, Manager manager){
37          this.receivePacket = r;
38          this.MyPort = port;
39          this.manager = manager;
40          this.currentUser = null;
41          this.lock = new ReentrantLock();
42      }
43
44      /**
45       * run() é executado sempre que é lançada nova thread ou enviado pacote ao utilizador a ser tratado na mesma.
46       */
47      @Override
48      public void run(){
49          PDU pdu;
50
51          try {
52              IPAdress = receivePacket.getAddress();
53              ClPort = receivePacket.getPort();
```

Figura 10: Exercerto do código de **ClientStub.java**.

Como podemos ver um ClientStub é sempre instanciado com a porta de atendimento, o objeto partilhado do servidor e o primeiro pacote recevido, a ser guardado numa variável auxiliar (**receivePacket**).

```
72                  //os metodos "do" são responsaveis por tratar os pedidos do Cliente
73                  switch (pdu.getTipo()){
74                      //cada mensagem tem um método para a tratar
75                      case 1:  //01 HELLO
76                          //faz sentido receber um HELLO aqui?
77                          this.doReplyOK(pdu.getLabel());
78                          break;
79                      case 2:  //02 REGISTER
80                          this.doRegister(pdu);
81                          break;
82                      case 3:  //03 LOGIN
83                          this.doLogin(pdu);
84                          break;
85                      case 4:  //04 LOGOUT
86                          this.doLogout(pdu);
87                          break;
88                      case 5: //05 QUIT
89                          this.doQuit(pdu);
90                          break;
91                      case 6: //06 END
92                          this.doEnd(pdu);
```

Figura 11: Decisão do procedimento em função do **tipo de pacote**.

Como podemos ver na fig. **??**, sempre que o stub receve um pacote, o procedimento a ser efetuado será determinado **em função do tipo de pacote**, desta forma podemos separar os pedidos e invocar os métodos em conformidade.

Para enviar mensagens para o Cliente atravéz do ClientStub, existe o seguinte método

```
public void sendReeply(PDU resposta);
```

4.3.2 Controlo de Fluxo

Durante alguns testes da aplicação verificamos que estavam a ser feitos muitos pedidos de retransmissão de PDU's dos desafios pelo Cliente. Após alguns testes, verificamos que o Cliente não conseguia processar as PDU's dos desafios à mesma velocidade que o Servidor os envia, o buffer de leitura do Cliente era rápidamente preenchido levando à perda de vários PDU's.

Para resolver o problema, bastou introduzir um pequeno sleep() no sendReeply() de forma a que o Servidor não fosse tão rápido a enviar PDU's para o mesmo Cliente e assim dar tempo a este de processar os PDU's enviados.

4.4 ChallengeThread

A Thread ChallengeThread é criada sempre que um pedido do tipo MAKECHALLENGE é aceite pelo servidor.

A ChallengeThread será reponsável por preparar e enviar toda a informação do desafio para os jogadores inscritos no mesmo, recerber e confirmar as suas respotas às questões e enviar o score do desafio.

Inicialmente a thread é adormecida, acordando à hora marcado para o inicio do desafio.

Quando chega à hora marcada, é feita a verificação do número de jogadores inscritos no desafio, para que se possa dar inicio ao desafio.

Uma vez que as questões a enviar não cabem num único PDU, estas têm de ser divididas em vários blocos. Para cada questão, são construidos um conjunto de PDU's a enviar para os jogadores.

Para que a thread seja capaz de responder eficientemente a pedidos de retransmissão, estes são mantidos em memória até ao final do desafio. Os PDU's são guardados num Map associados pela questão e bloco:

```
//Map<#Questao,Map<#Bloco,PDU>>
private Map<Integer, Map<Integer,PDU>> respostas;
```

sempre que haja um pedido de retransmissão para um determinado bloco de uma questão, apenas é necessário reenviar o PDU correspondente.

4.5 Manager

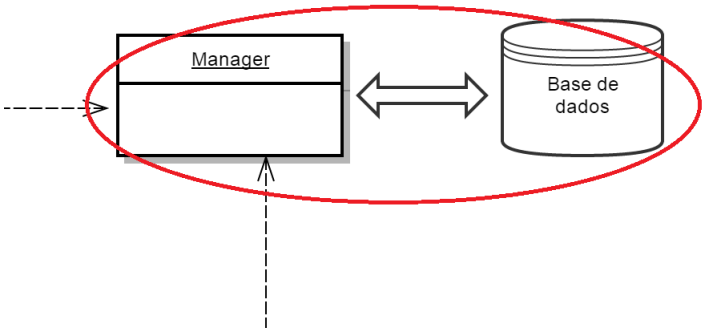

Figura 12: Manager.

Na figura 12 temos o componente da aplicação servidor que tem o acesso autêntico (ou se quisermos direto) aos dados, isto porque é sempre o Manager que efetua as alterações ou consultas sobre os dados armazenadas independentemente do cliente que efetua o pedido. Porque esta é uma zona crítica do nosso código no que diz respeito ao acesso aos dados, implementamos mecaniscmos de controlo de concorrência, **locks**, desta forma garantimos um acesso restrito aos dados por forma a que os mesmos estejam coerentes, evitando colisões e perda de dados.

4.5.1 Base de dados

A nossa base de dados no contexto deste projeto é constituída por um **pequeno sistema de ficheiros** armazenados na aplicação servidor (**apenas na aplicação servidor!**).

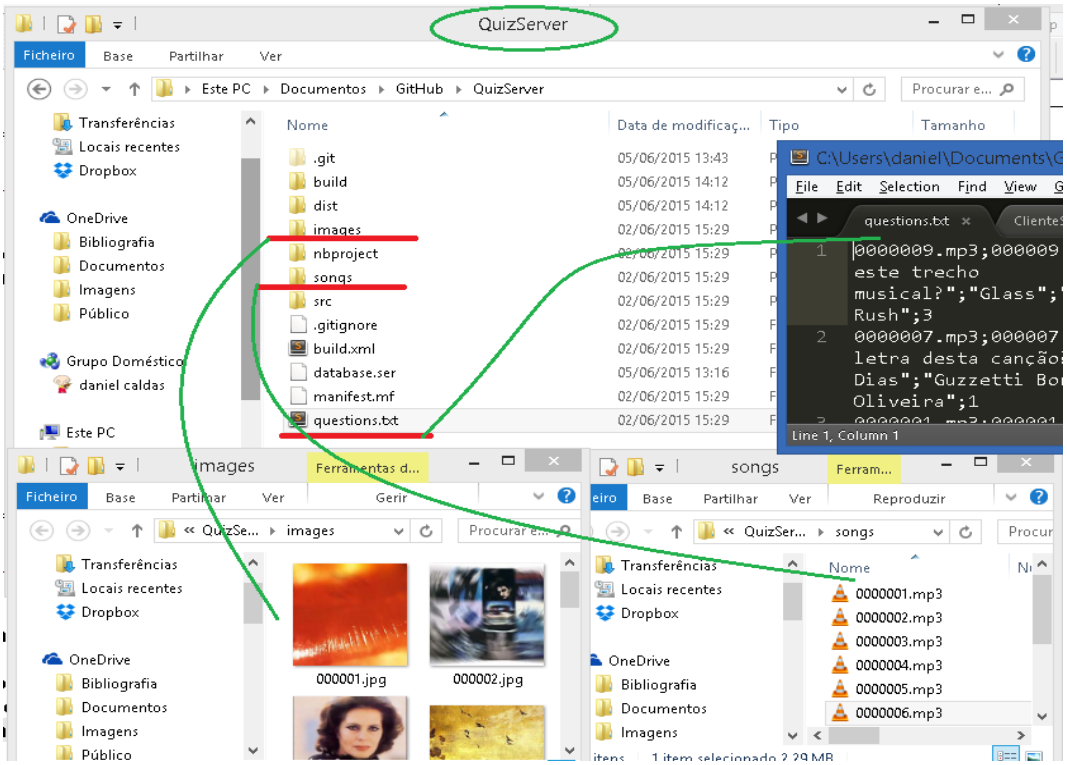

Figura 13: Organização dos ficheiros, base de dados.

4.5.2 Manager.java

Os métodos que realmente "*efetuam trabalho*" estão no nosso Manager. É aqui que acontece a autenticação de utilizadores, a geração de respostas de confirmação ou erro, a gestão de desafios tanto a longo prazo como no decorrer do próprio desafio.

```
35  public class Manager {
36
37      // Variáveis de instância
38      private int idChallengeGenerator;
39      private Map<String,User> users;
40      private Map<String,Challenge> challenges;
41      //para saber qual a Thread que esta a comunicar com o Cliente
42      private Map<String, ClienteStub> threads;
43      private List<Question> questions;//questoes
44      private Lock lock; // Lock para organizar acessos ao objeto partilhado
45
46      // Construtor
47      public Manager() {
48          this.idChallengeGenerator = 1000;
49          this.users = new HashMap<>();
50          this.challenges = new HashMap<>();
51          this.threads = new HashMap<String, ClienteStub>();
52          this.questions = new ArrayList<Question>();
53          this.loadQuestions(); // carrega as questoes para memoria
54          this.lock = new ReentrantLock();
55
56          //Utilizadores registados para teste
57          User us1 = new User("jcm","jcm","123");
58          this.users.put(us1.getUsername(), us1);
59          User us2 = new User("rc","rc","12345");
60          this.users.put(us2.getUsername(), us2);
61          User us3 = new User("carlos","jjm","12345");
62          this.users.put(us3.getUsername(), us3);
63
```

Figura 14: Exercerto do código de **Manager.java**.

Na fig. 14, podemos observar a estrutura do nosso Manager, em que guardamos toda informação acerca de:

- **Utilizadores** (*users*);
- **Desafios** (*challenges*);
- **Sessões de utilizadores ativos** (*threads*);
- **'Base de Dados" de questões** (*questions*);
- **Objeto de controlo de concorrência** (*lock*);

5 Primeira abordagem - Cliente

O Cliente é a parte da aplicação destinada aos jogadores. Deverá ser possível aos jogadores registarem-se e autenticarem-se no sistema. posteriormente deveram poder criar, listar e inscreverem-se em desafios e obter o score final do desafio assim o ranking global.

Todas as funcionalidaes impelentadas utilizam o PDU para poder comunicar com o servidor e receberem respostas a eventuais pedidos.

5.1 Connection

O Connection é responsável por inicar a comunicação com o Cliente, através do envio da mensagem com o tipo HELLO. Caso haja uma resposta positiva do servidor, a aplicação está então pronta a iniciar a troca de novas mensagens com o Servidor.

```
void boolean initCommunication();
```

O Connection é partilhado pela interface e o envio e receção de mensagens é feito sempre atravéz deste.

Para que o Cliente possa receber mensagens do Servidor a qualquer momento, e sem que lhe tenha sido feito um pedido é utilizada a Thread ReaderThread. A ReaderThread estará assim sempre à espera de mensagens do Servidor, adiocionando as novas mensagens ao Connetion, que será responsável por estas.

Sempre que seja necessário efetuar um pedido ao Servidor, deverá ser utilizado o seguinte método

```
public PDU getSendAndRecive(PDU msg);
```

este, é responsável por enviar uma mensagem ao Servidor, adormecendo (um certo período de tempo até à ocorrência de timeout) à espera da resposta. Quando uma nova mensagem é adicionada ao Connection, este é responsável por acordar todas as Threasd à espera de uma resposta. É feita a verificação através do valor da Label do PDU, para confirmar que se trata da resposta ao mesmo pedido. Caso não o seja, a Thread volta a adormecer.

5.2 Desafio

Sempre que no Connection é recebido uma mensagem sobre um desafio, é verificado se já existe alguma Thread criada para desafio. Se já existir uma Thread responsável pelo desafio então é adicionado o respetivo PDU à Thred, caso contrário será então criada uma GameThread e adicionado o PDU recebido.

5.3 GameThread

A GameThread é a Thread responsável pelo desenrolar de um desafio, esta Thread apenas é criada quando o cliente recebe uma primeira mensagem com dados sobre o jogo. Evita-se assim que o Cliente esteja á espera do Servidor para começar o jogo, evitando alguns problemas que podessem acontecer, como o Cliente e o Servidor não estarem sincronizados com a mesma hora.

Assim que a Thread é criada, o desafio está pronto a começar, sendo confirmado se jogador pretende iniciar o mesmo.

Para que a interface possa apresentar as questões é utilizado o método **getNextQuestion()**. Este método é responsável por retornar a próxima Questão para o jogo.

```
public Question getNextQuestion();
```

A parte mais critica do cliente encontra-se aqui. Para poder contruir a questão é verificado se já foram recebidos todos os pacotes da questão, para que seja possível voltar a juntar os dados que foram divididos pelo servidor.

Uma vez que existe alguma probabilidade de que alguns PDU's sejam perdidos durante o envio, **foi definido um timeout** para que podessem ser pedidos os pacotes não recebidos ao Servidor. Assim, sempre que a plicação fica um determinado tempo á espera de um determinado PDU, é feito um pedido de retransmissão ao Servidor. De notar que apenas é pedido de retransmisão do PDU em falta, não havendo necessidade de retransmitir toda a questão.

Assim que todos os blocos/PDU's da próxima questão foram recebidos do Srevidor os dados são reconstruidos, a questão está pronta a ser apresentada ao jogador.

6 Interface - Aspeto final

Nesta secção demonstramos o aspeto final da interface do nosso jogo distribuído.

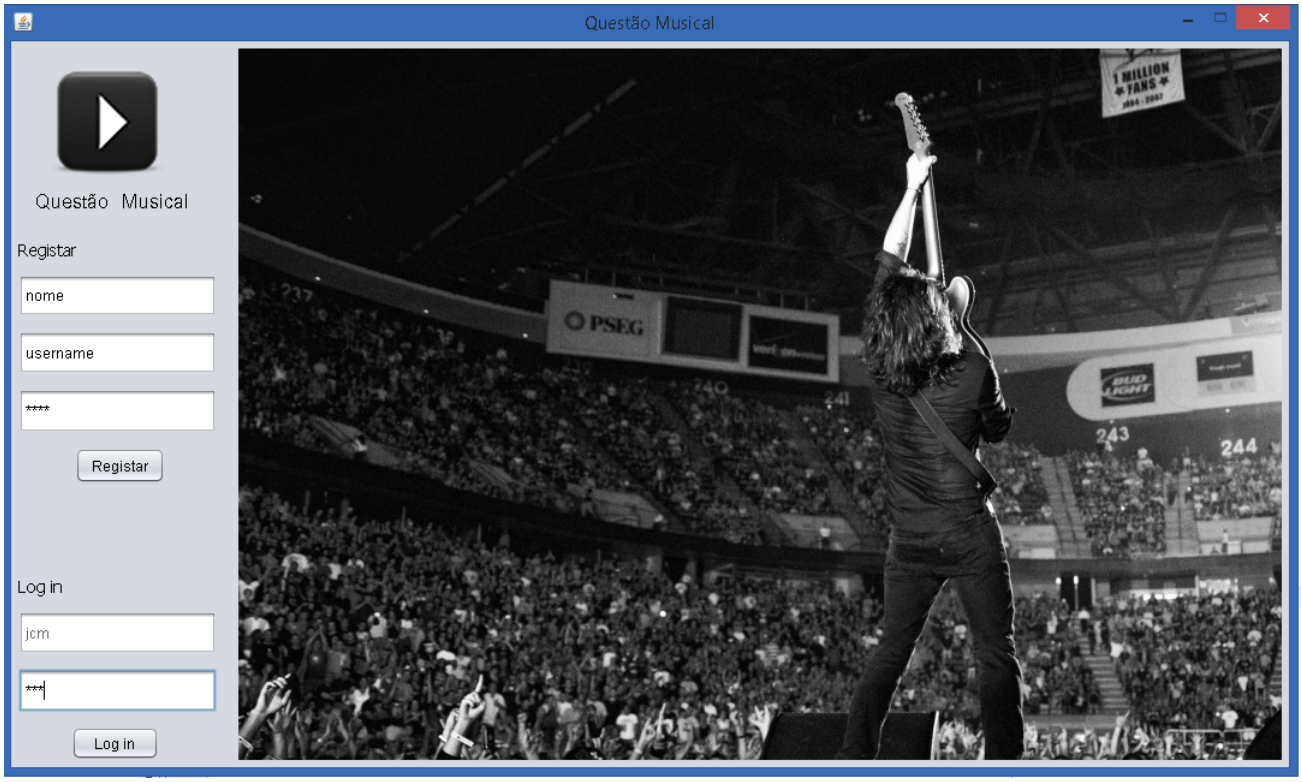

Figura 15: Esta é a janela inicial onde se desenrolam os processos tradicionais de autenticação e registo.

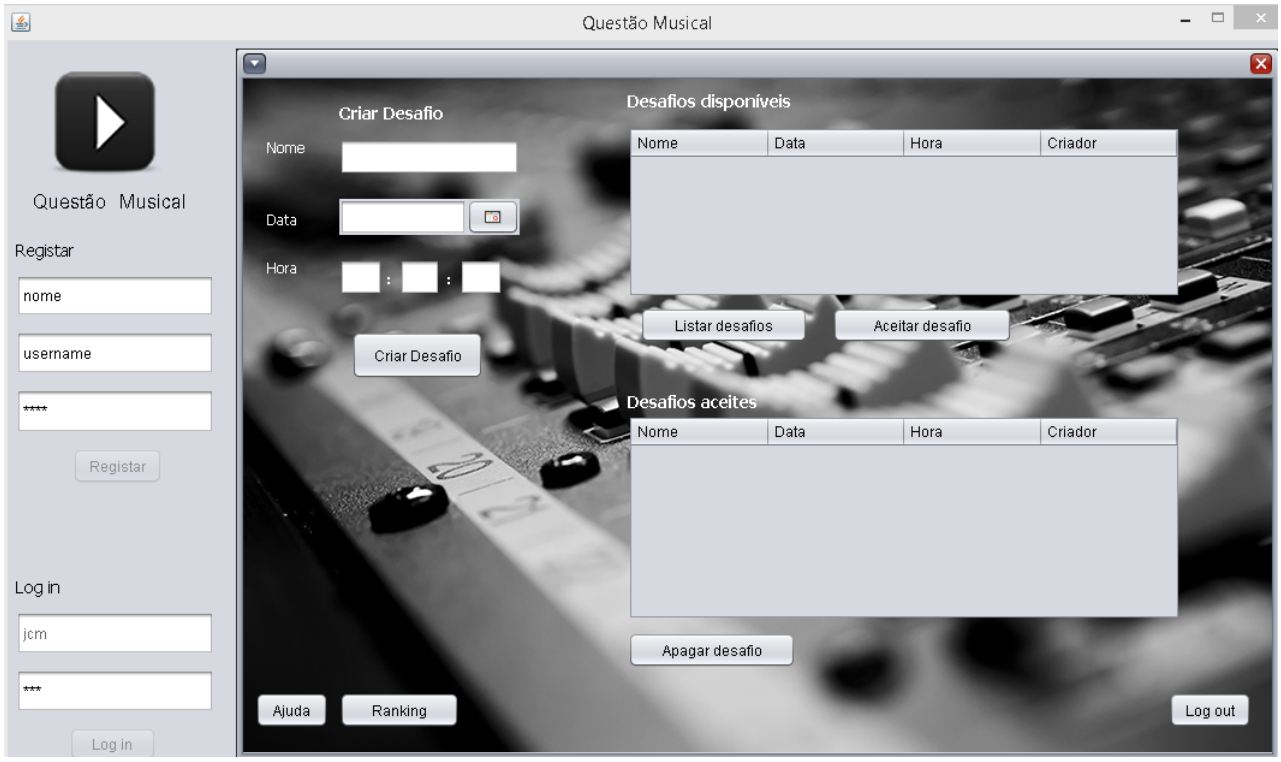

Figura 16: No perfil um utilizador pode criar desafios, visualizar e aceitar desafios propostos por outros, aceder ao ranking geral e ainda a uma secção de ajuda.

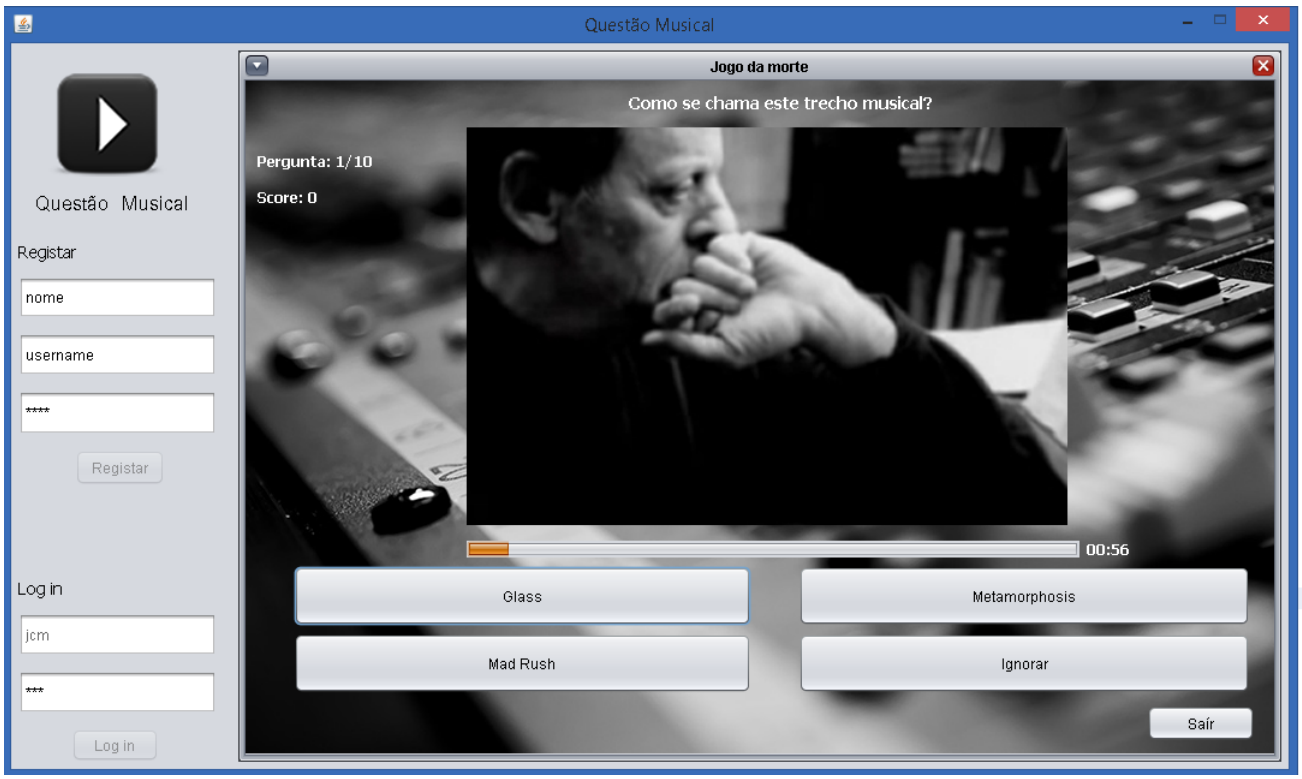

Figura 17: Exemplo de um desafio a decorrer.

Figura 18: Aqui o utilizador pode ficar a conhecer as regras do jogo.

Figura 19: Exemplo da funcionalidade de listar ranking numa pequena mensagem pop-up.

7 Conclusões e Trabalho Futuro

No desenvolvimento do projeto o grupo focou-se em implementar todas as funcionalidades da 1ªfase. As funcionalidades propostas foram devidamente implementadas introduzindo algumas melhorias na aplicação e obedecendo ao protolo base poposto. O projeto revelou-se demasiado grande, surgindo várias diculdades para a sua implentação. As dificuldades encontradas ajudaram também a uma melhor compreensão dos protolocos de encaminhamento e perceber a dificuldade da sua implementação.

Contudo, por falta de tempo não foi possível terminar a última fase. Uma vez que a maior parte do esforço de programação já se encontra desenvolvido, bastando algumas horas de trabalho para colocar a aplicação a funcionar com múltiplos Servidores, este é apresentado como trabalho futuro a ser desenvolvido.