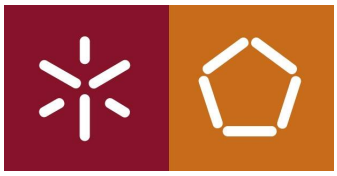

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistemas Distribuídos

Ricardo Costa	Guilherme Araújo	Gonçalo Esteves	Rui Oliveira
85851	a84527	A85731	A83610

1 Introdução

Neste trabalho, implementamos uma plataforma para partilha de ficheiros de música sob a forma de cliente/servidor em Java, intermediados por um servidor multithreaded também escrito em Java, recorrendo a comunicação via TCP. Primeiramente, começamos por implementar as classes e métodos necessários para haver comunicação cliente/servidor e, após isso, abordamos as propostas de funcionalidades oferecidas pela plataforma.

2 Cliente

Na classe **Cliente** são oferecidos todos os métodos que possibilitam a interação de um utilizador com os serviços da aplicação. Todas as funcionalidades são oferecidas por um servidor remoto, sendo por isso fundamental uma comunicação entre este cliente e esse mesmo serviço.

Para a funcionalidade adicional "Notificação de novas músicas" chegamos à conclusão de que criar dois clientes no método *main* desta classe seria a melhor opção, de forma a termos assim dois canais a trabalhar com o servidor em simultâneo através das sockets, um para tratar de comunicar com o servidor para a execução das funcionalidades que este oferece, e outro para receber notificações de publicações de novas músicas na plataforma. No entanto, temos consciência do facto de esta opção não ser a melhor em termos de eficiência, porém, consideramos ser a mais segura, no sentido em que conseguimos que não haja conflito de escritas e/ou leituras no momento em que estão a ser enviadas notificações aos utilizadores, tendo estes possibilidade de executar outras funcionalidades.

Apresentamos agora um exemplo de um método da classe Cliente que, neste caso, denomina-se *login*, responsável por comunicar com o servidor de modo a que um utilizador consiga aceder à aplicação.

```
// Cliente.java
final int ERR_DadosInvalidos = 101;
final int ERR_UtilizadorJaLogado = 103;
public void login(String user, String pass) throws
    DadosInvalidosException, UtilizadorJaLogado, IOException {
       this.out.writeUTF("1-" + user + "-" + pass);
       this.out.flush();
       int x = new Integer(this.in.readUTF());
       (x == ERR_DadosInvalidos)
           throw new DadosInvalidosException();
       else if (x == ERR_UtilizadorJaLogado)
           throw new UtilizadorJaLogado();
   }
```

2.1 Interface

A interação com o utilizador é através do uso comandos, e este, antes de executar qualquer operação, efetuar o login ou registar-se. Depois deste momento, poderá utilizar todas as funcionalidades do sistema.

1 - Publicar musica
2 - Procurar musica
3 - Descarregar musica
0 - Sair

3 Servidor

A classe do **Servidor** tem duas variáveis de instância, que representam uma lista de utilizadores e uma lista de músicas, e uma vez que as funcionalidades suportadas pelo servidor não envolvem o contacto destes dois grupos de informação, concluímos que, a nível de concorrência, esta divisão facilita a sincronização.

Devido à funcionalidade adicional abordada na classe **Cliente**, o servidor começar por executar uma instância da classe **NotificationSendThread**, que é informada sempre que uma música é publicada.

Este servidor é ainda administrado por um sistema de multi-thread, que permite o conctaco entre si e os clientes. Para isto, é criada uma thread quando um cliente se liga ao servidor que executa uma instância da classe **WorkThread**, capaz de atender um cliente.

A execução das mensagens recebidas é feita através de uma nova classe, chamada **Tarefa**, que é capaz de descodificar as mesmas e invocar o método local correspondente, enviando de seguida o resultado para o cliente.

Assim como na classe **Cliente** são criados dois clientes, de forma a um utilizador da aplicação poder comunicar através de dois canais com o servidor, nesta classe criamos duas sockets de modo a que a ligação entre as entidades referidas seja possível. Sendo que a primeira socket é responsável por tratar da comunicação do que já referimos anteriormente, através da thread que executa uma instância da classe **WorkThread**, a segunda trata de executar uma instância da classe **NotificationToClientThread**, que se encarrega de enviar a informação de músicas que sejam publicadas na plataforma.

Para realizarmos certas funcionalidades adicionais, como "Notificação de novas músicas" e "Limite de descargas", analisamos que o melhor seria a criação de classes auxiliares que usavam como variáveis de instância *ReentrantLock* e respetivas variáveis de condição.

3.1 Lista de Músicas

Para implementar um conjunto de músicas, utilizámos um *Map* que associa o seu código a uma **Música**. Todas as operações que modificam e acedem à estrutura são feitas em exclusão mútua nos objetos necessários.

Apresentamos agora um método desta classe, responsável por adicionar uma música ao *Map* que guarda o conjunto de músicas da plataforma.

```
// ListaMusicas.java
    private Map<Integer, Musica> musicas;
    private Integer id;
    private ReentrantLock lockLista;

    public int publicarMusica(String titulo, String interprete, int ano,
         String[] etiquetas){
        this.lockLista.lock();
        int id_Criado = this.id++;
        this.musicas.put(id_Criado, new Musica(titulo, interprete, ano,
            etiquetas, 0));
        this.lockLista.unlock();
        return id_Criado;
    }
```

Para determinar o ID da música a criar é desnecessário ter lock sobre qualquer elemento da lista de músicas, apenas sendo necessário adquirir o lock sobre a lista para o adicionar.

3.2 Utilizador

Ao contrário do conjunto anterior, para a lista de utilizadores, é também importante o controlo de concorrência sobre um utilizador da mesma. Como tal, criamos na classe **Utilizador** uma variável de instância denominada *lockUtilizador*, responsável por garantir exclusão mútua através de lock explícitos.

```
// Utilizador.java
    private ReentrantLock lockUtlizador;

    public void lock(){
        this.lockUtlizador.lock();
    }
    public void unlock(){
        this.lockUtlizador.unlock();
    }
```

Podemos ver a implementação do controlo de concorrência através do método *login* da classe **Servidor**, responsável por logar um utilizador ao sistema.

```
// Servidor.java
    private ListaUtilizadores utilizadores;

    protected Utilizador login(String user, String pass) throws
         UtilizadorJaLogado, DadosInvalidosException {
        if (!this.utilizadores.utilizadorValido(user, pass))
            throw new DadosInvalidosException();
        Utilizador u = this.utilizadores.getUtilizador(user);
        u.lock();
        if (u.isLogged()){
            u.unlock();
            throw new UtilizadorJaLogado();
        });
        u.setLogged(true);
        u.unlock();
        return u;
    }
```

Antes de verificarmos se o utilizador que quer dar login já está logado no mesmo, adquirimos o lock que garante a execução da região crítica em exclusão mútua.

Caso seja verificado que este não está logado, é invocado um método que atualiza o utilizador como logado.

Por fim, o lock é libertado e o objeto **Utilizador** obtido é retornado.

4 Conclusão

Em suma, através deste projeto, conseguimos pôr em prática os conceitos lecionados nas aulas de Sistemas Distribuídos. Estabelecer um protocolo universal para a troca de mensagens, conjugado com a possibilidade do servidor atender vários clientes em simultâneo, recorrendo para isso a multithreading e, consequentemente, a controlo de concorrência e esperas passivas, foi o principal meio para realizar com sucesso este trabalho. Concluímos então com a consciência de que todos os objetivos propostos foram atingidos com sucesso.