Sistemas Distribuídos

MIEI - 3º ANO - 1º SEMESTRE
UNIVERSIDADE DO MINHO

SOUNDCLOUD

Grupo 26

Bernardo Viseu
A74618

Renato Cruzinha
A75310

André Sousa
A74813

Braga, 5 de Janeiro de 2020

Conteúdo

1. Introdução

No âmbito da cadeira de Sistemas Distribuídos foi-nos proposta a elaboração de um trabalho prático cujo o tema imposta seria a implementação de um *SoundCloud*. Desta forma, este projeto tem como objetico a implementação de uma aplicação distribuída em que seja permitido o upload e download de ficheiros de música. Os utilizadores podem fazer upload de música para o Servidor, e por sua vez também podem fazer download de músicas para o seu dispositivo. Tudo isto está implementado usando um modelo cliente-servidor escrito em linguagem de programação JAVA, no qual os utilizadores devem poder interagir, intermediados por um servidor multi-threaded, e recorrendo a comunicação via sockets TCP.

2. Implementação

2.1 Utilizador

A classe **Utilizador** é a classe que representa qualquer utilizador do *SoundCloud*. Nela são guardados o *username* e a respectiva *password*.

2.2 Ficheiro

A classe **Ficheiro** é a classe que representa uma música. Esta classe guarda o *id* único, o *nome*, o *artista*, todas as *labels* com que a música pode ser encontrada numa pesquisa, o *ano* e o número de vezes que essa música foi descarregada *times_downloaded*.

2.3 ServerMessage

A classe **ServerMessage** representa as mensagens que o servidor envia para o utilizador, como resposta às acções. Veremos mais adiante o funcionamento da comunicação Servidor-Cliente onde esta classe tem um papel muito significativo.

2.4 SoundCloud

A classe **SoundCloud** é a classe responsável por armazenar toda a informação presente na aplicação num determinado momento. Nesta classe estão presentes todos os utilizadores registados, assim como o registo de todas as músicas que foram carregadas para o servidor. É também aqui que estão armazenadas todas as comunicações que foram efectuadas entre o servidor e um cliente, à excepção dos ficheiros de música partilhados, pois esses não ficam guardados em cache. Esta classe é partilhada por

todos os utilizadores da aplicação, o que implica que qualquer interacção com a nossa aplicação tenha necessariamente acesso a esta.

2.5 Menu

A classe **Menu** é responsável por apresentar os diferentes tipos de menus para o tipo de actividade diferente que um utilizador possa ter.

2.6 Cliente

A classe **Cliente** representa o executável que qualquer utilizador da nossa aplicação irá usar para desfrutar da mesma.

2.7 ClienteInput

A classe **ClienteInput** representa a classe responsável por, para cada cliente, informar o servidor da acção que o cliente pretende efectuar.

2.8 ClienteOutput

A classe **ClienteOutput** representa a classe responsável por receber do servidor informação para mostrar para o cliente.

2.9 Servidor

A classe **Servidor** representa o executável que permite manter o servidor ligado enquanto os utilizadores desfrutam da aplicação.

2.10 ServerRead

A classe **ServerRead** representa a classe responsável por interpretar a informação recebido do cliente, fazendo a respectiva acção.

2.11 ServerWrite

A classe **ServerWrite**, por sua vez, é responsável por, após o cliente executar uma acção, transmitir a respectiva informação para este.

2.12 Exceptions

As Exceptions são excepções enviadas quando acontecem situações indesejadas durante a execução da nossa aplicação. Existem na nossa aplicação as seguintes aplicações:

- PasswordIncorretaException
- UsernameInexistenteException
- UsernameTakenException

3. Comunicação Servidor/Cliente

Para cumprir com os requisitos do enunciado, nós implementamos um servidor e um cliente que comunicam via sockets (TCP). Quando um Cliente é inicializado, o Servidor aceita-o, e a comunicação entre ambos é estabelecida em torno de quatro threads, sendo que são duas para cada uma das partes. Uma das threads do Servidor tem como objectivo receber mensagens por parte do Cliente, enquanto que a outra têm como objectivo enviar mensagens para o Cliente. O mesmo funcionamento acontece do lado do Cliente, em que as duas threads existentes são para envio e recepção de mensagens vindas do Servidor. As mensagens que são enviadas entre o Cliente e Servidor são enviadas de duas maneiras distintas, dependendo da acção do Cliente. Para qualquer interacção orientada à linha, a comunicação é feita através de linhas de texto (Strings), mas para o caso do Cliente querer fazer uma transferência de ficheiro, quer download, quer upload, é necessário transformar esse ficheiro num array de bytes e nesse caso o Servidor/Cliente comunicam o envio desses bytes, através de um *DataInputStream* e de um *FileOutputStream*. A imagem que se segue clarifica melhor a nossa estruturação.

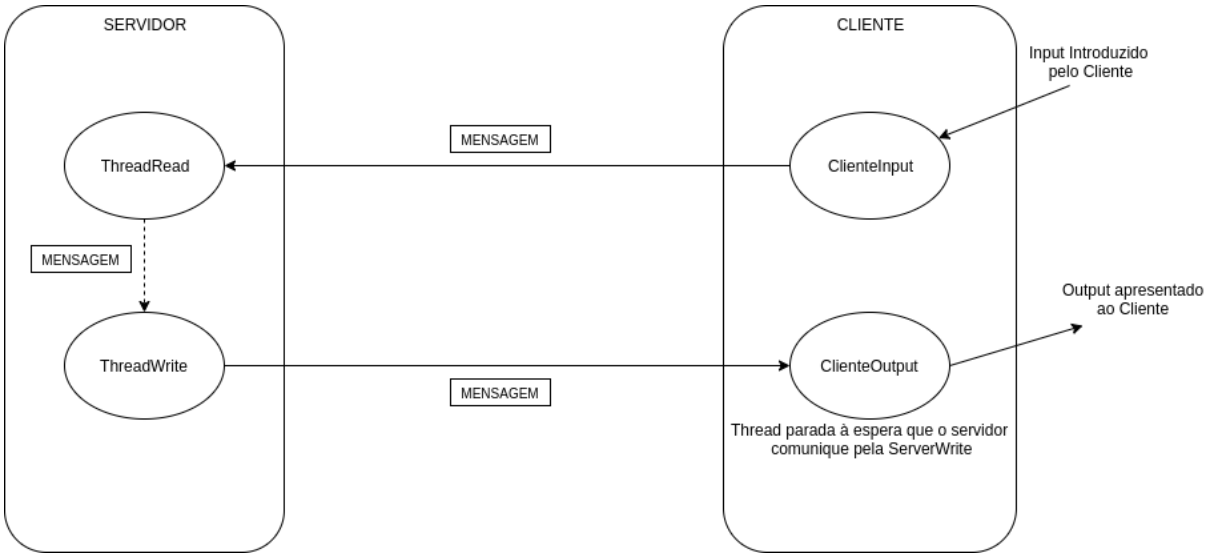

Figura 3.1: Esquema de comunicação Servidor/Cliente

A comunicação entre o ServerRead e o ServerWrite é feita indirectamente, ou seja, é implementada a classe ServerMessage, que guarda todas as mensagens referentes a um utilizador específico. No caso de ser um ficheiro de música, ele é guardado provisoriamente nesta classe até ser enviado, onde depois é removido, para não ficar em memória. Para além disso, para que a ServerWrite saiba quando pode enviar uma mensagem, possui uma *Condition*, onde fica em *await()* até que seja recebido um *signal()* para acordar, por parte do ServerRead, que quer enviar uma mensagem ao utilizador que este comunica. Qualquer ServerRead que o Servidor possui, visto ser uma por Cliente, pode mandar mensagens para qualquer ServerMessage que exista, que visava a permitir que acções de outros clientes no Servidor possam ser notificados sempre que fosse efectuado um upload.

4. Concorrência

O principal objectivo deste trabalho é controlar a transferência de ficheiros de música e o envio de mensagens para os diversos utilizadores, de maneira que o programa consiga lidar com a informação de vários clientes em simultâneo. Para isso, tivemos que utilizar mecanismos de controlo de concorrência, que permitam ao servidor transmitir uma imagem de coerência em resposta a todos os clientes.

No código por nós desenvolvido, nas classes ServerMessage e SoundCloud podemos verificar que existe partilha, por todas as threads do Servidor. O SoundCloud é composto por um conjunto de utilizadores, um conjunto de músicas e um conjunto de ServerMessage, para que seja possível haver comunicação entre os diferentes clientes e para que a informação num instante de tempo esteja armazenada de modo a ser acedida sem problemas. Para garantir que esta informação partilhada por todos esteja sempre actualizada quando é pretendida por cada utilizador, implementamos um sistema que faz com que só um utilizador possa aceder à informação partilhada por todos os utilizadores de cada vez.

Sendo assim a classe SoundCloud possui três *locks*, sendo um para a lista de utilizadores, outro para a lista de mensagens e, por fim para a lista de músicas, permitindo assim bloquear o acesso de vários utilizadores, enquanto um está a aceder aos objectos, e libertar o mesmo quando este acaba.

Na classe ServerMessage, também é utilizado um lock, para que seja bloqueado o acesso simultâneo dos utilizadores, permitindo que utilizadores adicionem uma mensagem ou um conjunto de mensagens em diferentes ServerMessage em simultâneo, contudo não se aplica no mesmo ServerMessage.

```
public Utilizador login(String username, String password, ServerMessage sm) thr
    this.lockUsers.lock();

    try{
        if(!this.users.containsKey(username)){
            throw new UsernameInexistenteException("Invalid username.");
        }
        else if(!this.users.get(username).getPassword().equals(password)){
                throw new PasswordIncorretaException("Invalid password.");
        }
    }
    finally{
        this.lockUsers.unlock();
    }

    this.lockMsgs.lock();
    try{
        if(this.user_messages.containsKey(username)){
            ServerMessage m = this.user_messages.get(username);

            String linha;
            while((linha = m.getMessage()) != null){
                sm.setMessage(linha, null);
            }
            this.user_messages.put(username, sm);
        }
    }
    finally{
        this.lockMsgs.unlock();
    }

    this.lockUsers.lock();

    try{
        return this.users.get(username);
    }
    finally{
        this.lockUsers.unlock();
    }
}
```

Figura 4.1: Exemplo de utilização dos diversos locks

Em suma, o programa por nós desenvolvido não deixa que dois utilizadores acedam a informação partilhada em simultâneo, permitindo assim que a informação esteja correta no momento em que é consultada.

5. Interface

A nossa aplicação apresenta uma interface muito simples, uma vez que o seu propósito é ser utilizado numa linha de comandos. Sendo assim são utilizados dois menus essenciais para a interacção com o Cliente.

5.1 Menu

Este é o Menu geral para qualquer Cliente que se conecte ao Servidor, onde pode efectuar o seu login, criar a sua conta, ou sair do programa.

Figura 5.1: Menu Inicial

5.2 Utilizador

Este é o Menu onde os utilizadores registados e com login efectuado podem interagir e usufruir da utilização da nossa aplicação. Aqui têm as opções de fazer download de músicas existentes no Servidor, fazer upload de músicas que ficarão disponíveis a todos os utilizadores e também a uma opção de procura das músicas guardadas no servidor, através de *labels*.

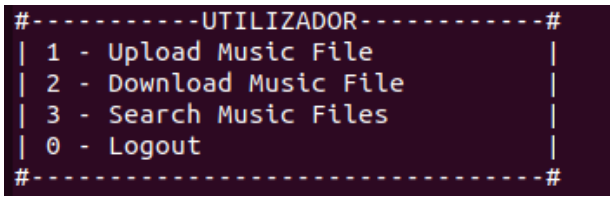

Figura 5.2: Menu do Utilizador Logged in

6. Conclusão

Com a realização deste trabalho pudemos aplicar todos os conceitos e o vasto conhecimento adquirido durante as aulas, pondo em prática grande parte daquilo que nos foi transmitido.

Foi nos proposto que fizemos um serviço de hospedagem de música num Servidor e que fosse possível haver comunicação entre o mesmo e um Cliente. Para além desta comunicação essencial, era importante implementar uma comunicação com transferência de ficheiros, para além da normal comunicação entre Servidor/Cliente. Foi neste ponto que encontramos o nosso tendão de Aquiles, pois deparámo-nos com um problema que era conseguimos transgredir da nossa comunicação implementada para a transferência de ficheiros. Para resolver este impasse tivemos de usar um método diferente para a comunicação, ou seja, em vez de usar-mos a comunicação via Strings, tivemos de implementar uma comunicação que usasse um array de bytes. Como tal, avançamos para a utilização de *DataInputStream* e *FileOutputStream*.

Com isto dito podemos afirmar que para uma futura melhor implementação poderia ser usar uma comunicação mais sólida, ou seja, em vez da actual comunicação de dois tipos, utilizar um método alternativo para o melhor funcionamento do projecto.