

**Universidade do Minho**

Escola de Engenharia

Mestrado Integrado em Engenharia Informática

**Unidade Curricular de**

**Sistemas Distribuídos**

Ano Letivo de 2019/2020

***SoundCloud***

**Grupo n.º 11:**

**Alexandre Reis da Costa, A78890**

**Cecília da Conceição de Oliveira Soares, A34900**

**Luís Filipe Simões de Abreu, A82888**

**Miriam Miranda Pinto, A42040**

5 de Janeiro, 2020

**Resumo**

Este relatório surge no âmbito do projeto da unidade curricular Sistemas Distribuídos, lecionada no 3.º ano do Mestrado Integrado em Engenharia Informática e pretende documentar o processo de desenvolvimento de uma plataforma de troca de ficheiros.

O objetivo desta plataforma é transferir ficheiros de música acompanhados de meta-informação, a qual permitirá aos utilizadores partilhar músicas. Para tanto, utilizamos a linguagem Java para criar uma solução cliente/servidor com multi-thread....

1. **Introdução**

Para a elaboração da referida plataforma começamos por criar a classe Cliente onde foram definidos os métodos considerados essenciais para um utilizador, nomeadamente a autenticação e registo do utilizador, etc.. A classe Cliente tem uma variável pertencente à classe ClienteStub, a qual faz estabelece a ligação entre a classe Cliente e Servidor, foi projetada para funcionar como um “tradutor” em ambos os sentidos.

Do lado do Servidor, temos um Servidor multi-thread, onde cada thread faz o tratamento dos pedidos de um cliente. Os pedidos que o utilizador pode ver executados com a nossa aplicação são: o registo de utilizador, autenticação do utilizador, upload de uma música, o download de uma música, procura de músicas e, por último, sair da aplicação.

Ademais, para limitarmos o número de operações simultâneas a serem efectuadas pelo sistema criamos uma “PoolThread” que nos fornece uma fila de tarefas a serem executadas, as quais serão executadas por ordem FIFO, e um conjunto de threads que serão responsáveis pela execução dessas mesmas tarefas. Este mecanismo também nos possibilitou limitar as descargas de ficheiros em simultâneo, conforme se explica mais adiante.

Acresce que, além de termos implementado as funcionalidades acima descritas, também garantimos que a transferência de um ficheiro não ocupa mais do que 1024 bytes em memória.

1. **Desenvolvimento**
   1. **Estrutura de Dados**

O nosso programa é composto por três pacotes (Server, Client e Exceptions) que agregam várias classes que cumprem finalidades distintas e que passaremos a explicar.

O pacote **Client** agrega as classes Cliente e ClienteStub. A primeira classe trata da interface com o utilizador e é o ponto de entrada do programa. Por outro lado, o ClienteStub estabelece a conexão com o servidor e “trata” a mensagem do utilizador antes de a enviar ao Servidor. Ademais, a referida classe receciona as respostas enviadas do lado do servidor e trata-as para serem apresentadas ao utilizador.

O pacote **Server** reúne dois pacotes (Dados e Pedidos), a interface *SoundCloud* e quatro classes que vão desenhar a arquitectura do nosso programa relativa ao funcionamento do servidor.

Em primeiro lugar, o pacote Dados tem todas as classes e métodos relativos às músicas e utilizadores que o servidor armazena. Já o pacote Pedidos agrega todos as classes e métodos referentes aos pedidos que o Servidor processa e qual o tratamento que dá a cada um deles através da invocação do método *executar* em cada um dos respetivos pedidos.

Em segundo lugar, a interface *SoundClound* contém os métodos implementados pelas nossas classes *ClienteStub* e *ServerHelper*, sendo que esses mesmos métodos implementam as funcionalidades da nossa plataforma de troca de ficheiros, designadamente a autenticação do utilizador, o *upload* e *download* de músicas, entre outras.

No que se refere às classes do pacote Server, estas são a classe Servidor que é o ponto de entrada do programa do lado do servidor e que aceita a conexão requerida pelo *Cliente*.

Além do Servidor, temos a classe Worker, que recebe os pedidos do Cliente e coloca-os em fila de espera até serem executados por uma das *threads* da nossa classe *ThreadPool*. Esta última classe implementa uma fila de pedidos e um array de *threads* que vão se encarregar de executar os diferentes pedidos.

No pacote Server, resta-nos referir a nossa classe *ServerHelper*. Esta classe implementa a interface *SoundCould* e é aquela que implementa a resposta aos pedidos do utilizador.

Por último, temos o último pacote, o *Exceptions* que agrega as diferentes classes de exceções do nosso programa.

* 1. **Descrição das Atividades**

O ponto de entrada por parte do utilizador é através da nossa classe Cliente. Esta classe, conforme já referimos anteriormente, disponibiliza uma interface com o utilizador que permite suportar as funcionalidades do nosso programa.

Esta classe contém uma variável de instância da classe *ClienteStub*, a qual irá funcionar como um “tradutor” entre Cliente e Servidor. Tal acontece porque o Cliente quando inicia o programa cria um *ClienteStub* passando como parâmetro ao construtor daquele o IP e o porto do servidor, sendo que é o *ClienteStub* que cria o socket para se conectar com o servidor.

Estabelecida a ligação com o Servidor, todas as mensagens escritas no terminal pelo utilizador são processadas pelo ClienteStub antes de serem enviadas ao Servidor para tratamento do pedido correspondente. Por sua vez, a resposta ao pedido do utilizador enviada pelo Servidor também é recebida pelo ClienteStub e só depois de tratada é que é entregue ao utilizador. Desta feita, para manter a linguagem padronizada entre o servidor e o ClienteStub, estabelecemos um protocolo de comunicação entre ambos, que será detalhado no título seguinte.

Iniciado o programa do lado do Cliente, resta-nos explicar como o mesmo se inicia do lado do Servidor.

Em primeiro lugar, a classe Servidor começa por criar um *serverSocket* que vai rececionar os pedidos de conexão por parte dos clientes e ao aceitá-los cria um *socket* específico para comunicar com cada um dos clientes. Ademais, por cada conexão estabelecida é criada uma *thread* (é criada uma *thread* por cliente), que irá executar a nossa classe *Worker*. Esta classe, recebe os pedidos do cliente e coloca-os em fila de espera até serem executados por uma *thread* da nossa classe *ThreadPool*.

No que se refere aos pedidos dos utilizadores, criamos uma super classe abstrata PedidoCliente que implementa os métodos genéricos que todas as classes de pedidos vão usar, sendo estes os métodos *espera*, *notificaEspera* e define a assinatura do método *executa*. O primeiro método obriga a que cada pedido de determinado cliente tem de ser satisfeito até que esse mesmo cliente possa enviar mais pedidos para a fila de espera de pedidos. Já o método *notificaEspera* avisa as threads de que o seu pedido foi já executado e, por isso, pode enviar mais pedidos. Por último, o método *executa*, tal como o nome indica, executa as diferentes acções.

Entretanto, a classe *ThreadPool* implementa uma fila de tarefas, com 30 posições, que armazena, em cada uma das suas posições, um pedido de um cliente conectado à plataforma, os quais serão executados, por ordem FIFO, por uma das *threads* de um conjunto de 10. O esquema de funcionamento desta classe está representado na figura 1.

Ademais, convém referir que esta classe foi implementada também para dar resposta ao limite de descargas que podem estar a decorrer em simultâneo. Desta feita, para garantirmos que não há mais do que MAX\_DOWNLOADS downloads a decorrer em simultâneo, verificamos qual o tipo de pedido que a *thread* irá executar e, caso seja um download, verificamos ainda se o número de downloads a decorrer são menores do que o valor MAX\_DOWNLOADS e, em caso afirmativo, é então executado o pedido. No entanto, se o valor do MAX\_DOWNLOADS já tiver sido atingido, a *thread* que selecionou o pedido de download que ultrapassa o referido valor espera até que um dos downloads que está em execução acabe, altura em que é diminuído o nosso contador de número de downloads que estão a decorrer em simultâneo e é enviado um *signal* à *thread* que estava à espera para processar o seu pedido de download. Com efeito, ao avisarmos uma única thread de que pode começar a executar a descarga do ficheiro, garantimos que todos os utilizadores obtêm os ficheiros e que terão de esperar apenas os pedidos de descarga que estão à sua frente. Acresce que, há justiça na obtenção da vez para poder dar resposta ao seu pedido de download porque a espera para obter o *lockDownload* segue a política FIFO.

Por último, resta referir que a classe *ServerHelper* implementa a execução de todas as funcionalidades da nossa plataforma *SoundCloud*. Vamos analisar sucintamente cada uma delas.

Primeiramente, o registo de um novo utilizador é feito somente se o nome escolhido não existir previamente armazenado no repositório de utilizadores do servidor.

Em segundo lugar, o login ou autenticação do nosso programa é feito se o utilizador digitou o seu nome e password e os mesmos forem iguais ao de um utilizador já guardado no servidor, o que significa que este foi previamente registado no sistema.

No que se refere à possibilidade do utilizador publicar um ficheiro de música, definimos que o utilizador terá de fornecer ao servidor, impreterivelmente, cinco campos (meta-dados da música), título, interprete, ano, género e o caminho do ficheiro da música. Caso contrário, o *upload* não é efectuado. Depois de fornecidos esses campos o *upload* é feito, garantindo que a transferência do ficheiro de música para o servidor é feita em *chuncks* de 1024 bytes (MAXSIZE), limitando o tamanho de cada pacote que é transferido. O mesmo acontece com o download dos ficheiros, sendo que para efetuar esta funcionalidade basta que o servidor tenha o Id da música, sendo este único. Ademais, sempre que um download de determinada música é efetuado, o mesmo é somado ao campo da música que contabiliza o número de vezes que a música foi descarregada.

Finalmente, a procura de uma música por determinada palavra é efetuada comparando cada um dos meta-dados de cada uma das músicas armazenadas pelo servidor com a palavra que o utilizador quer procurar. Caso essa palavra exista em algum dos campos dos meta-dados da música, esta é adicionada a uma lista que depois de completa será transformada numa *string* e passada para o *ClientSutb*. Já do lado do *ClienteStub*, este recebe a *string* e transforma-a numa lista para ser apresentada ao utilizador.

**Uma imagem com captura de ecrã

Descrição gerada automaticamenteFigura 1:** Esquema de funcionamento da Classe ThreadPool

* 1. **Protocolo de comunicação Cliente-Servidor**

O nosso protocolo de comunicação definiu como pedidos válidos e passíveis de serem processados como comandos inteligíveis, os que comecem pelas seguintes palavras: ‘registar’, ‘login’, ‘logout’, ‘procurarMusica’, ‘download’, ‘upload’ e ‘sair’.

Em primeiro lugar, para se registar, o utilizador deve escrever ‘registar’, o nome com que se quer registar no sistema e a palavra passe que quer que lhe seja atribuída. De notar que, caso já exista um utilizador registado com os mesmos parâmetros o registo é rejeitado.

No que se refere ao login, este é feito escrevendo no terminal ‘login’, o nome e a password com que o utilizador se registou.

Em terceiro lugar, para fazer logout basta escrever ‘logout’, sendo que o mesmo acontece com a opção ‘sair’. Estes dois pedidos fazem com que a conexão com o servidor se feche.

No que concerne ao pedido de um upload de uma música, o utilizador terá de escrever ‘upload’, o título da música a transferir, o seu intérprete, o seu ano, o género da música e o caminho para o ficheiro onde esta se encontra.

No que diz respeito ao pedido de download, o utilizador apenas terá de escrever ‘download’ e o Id da música que pretende transferir para a sua máquina. Para saber qual é esse número identificador, o utilizador poderá procurar os meta-dados dessa música na plataforma, sendo que para isso basta digitar ‘procurarMusica’ e uma qualquer etiqueta que identifique algum campo da música pretendida, por exemplo, ao digitar ‘procurarMusica Madonna’, receberá uma lista com os meta-dados, incluindo o identificador único, de todas as músicas que tenham a palavra Madonna em algum dos seus campos, nomeadamente no intérprete.

Ademais, convém notar que no nosso protocolo, nenhum dos meta-dados referentes às músicas constantes da plataforma podem conter espaços, porquanto as componentes do pedido que são digitadas pelo utilizador, bem como as que são enviadas pelo *ClienteStub* ao Servidor e vice-versa, são identificadas pelo espaço que existe entre elas. Por conseguinte, nenhum dos pedidos introduzidos pelo utilizador pode conter espaços entre palavras que pertencem ao mesmo comando. Assim, se o utilizador se quer registar com o nome Anabela Silva, deverá fazê-lo de forma alternativa, por exemplo, Anabela\_Silva. O mesmo acontecendo com a introdução do título da música ou o nome do intérprete.

De acordo com o protocolo de comunicação implementado, a resposta dada pelo servidor também é padronizada, sendo que utilizamos números inteiros para indicar ao *ClienteStub* o resultado do pedido efectuado. Assim, quando o pedido do cliente é satisfeito o servidor envia o número 1, ao passo que, quando o mesmo tem insucesso é enviado um 0. Ademais, temos outros valores que representam exceções, tal como o número 2 que indica que determinada música não existe.

1. **Como executar o programa**

Passos para executar o programa:

* Descompactar o programa e mover para a directoria criada;
* Compilar o programa com o comando mvn compile;
* Correr o servidor com o comando ./run-servidor.sh;
* Correr o cliente noutro terminal com o comando ./run-cliente.sh;
* Seguir as instruções da interface para executar as funcionalidades do sistema.
* Os logs se encontram nos ficheiros servidor-ssd.log e cliente-ssd.log, para o servidor e para o cliente, respectivamente.

1. **Conclusões**

Face ao exposto, podemos afirmar que atingimos os objetivos propostos, mas entendemos que existem ainda melhorias que no futuro podem ser implementadas.

No que se refere a objetivos futuros, as melhorias a introduzir seriam as notificações de músicas novas, tarefa que não foi por nós implementada por falta de tempo.

Além da tarefa adicional acima mencionada, tentamos ainda fazer uso do package *NIO*.\* onde temos a classe *Selector* que nos permitiria usar apenas uma *thread* em vez de várias para administrar múltiplos canais. Apesar dos nossos esforços não conseguimos, por falta de tempo, implementar a solução que havíamos delineado para o problema em mãos.

Acresce que, pensamos ainda em encapsular as mensagens que são enviadas pelo servidor em objetos, de forma a ser mais fácil tratá-las do lado do Cliente e de modo a termos uma solução mais elegante do que a atualmente implementada.

Em conclusão, apesar de termos as funcionalidades básicas do *SoundCloud* a funcionar, bem como algumas funcionalidades adicionais, entendemos que no futuro ainda há espaço para melhorias que iriam beneficiar a nossa plataforma.