Aplicação Cliente-Servidor para partilha de faixas de áudio

Relatório Final do Segundo Projeto

Sistemas Distribuídos
3º ano do Mestrado Integrado em Engenharia Informática e Computação

Elementos do Grupo sdis1718-t1g14:
Nádia de Sousa Varela de Carvalho – up201208223 – ei12047@fe.up.pt
Ricardo Miguel Carvalho – up201503717 – up201503717@fe.up.pt
Tiago Lascasas dos Santos – up201503616 – up201503616@fe.up.pt

28 de Maio de 2018

Índice

1. Introdução

A aplicação que desenvolvemos tem como objectivo providenciar um arquivo público e digital de música e outras faixas de áudio. Foi implementada usando uma arquitetura distribuída, com um ou mais servidores cujo objetivo é armazenar e tornar disponíveis as faixas, e vários clientes que, após autenticados, podem aceder e contribuir para esse arquivo.

A interface de utilizador é muito simples, pois pensamos mais nas funcionalidades que pudessem tornar esta aplicação mais útil na área de sistemas distribuídos, implementando os conhecimentos obtidos nas aulas teóricas e práticas da unidade curricular.

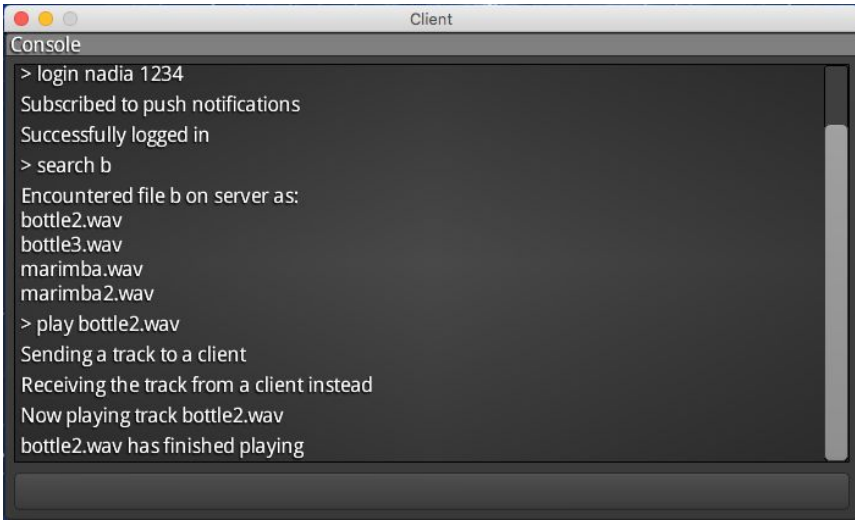

Imagem 1: Interface Gráfica do Cliente

Neste relatório iremos abordar, primeiramente, uma descrição da arquitetura do projeto, onde se descreve os componentes principais da aplicação e como se procede a sua interação, incluindo os protocolos utilizados na comunicação.

Numa segunda etapa, apresentamos uma especificação dos detalhes de implementação da aplicação, incluindo uma explicação mais pormenorizada da construção da arquitetura de cada componente, explicitando o contexto e as razões de uso das bibliotecas e frameworks quando for o caso.

Em seguida, refletimos sobre os principais problemas que tivemos de resolver, incluindo a segurança do sistema, o processo de tolerância a falhas, a implementação de notificações e a escalabilidade, explicitando as nossas decisões que permitiram tornar esses problemas em pontos fortes da nossa aplicação.

No final, teceremos conclusões sobre o estado do projeto, a sua utilidade e possíveis melhoramentos que possam ser efetuados no futuro.

2. Arquitetura

O sistema segue maioritariamente uma arquitetura Cliente-Servidor, com uma pequena componente Peer-to-peer.

Desta maneira, os dois principais módulos representam o cliente e o servidor, sendo que o primeiro possui um caso de uso que envolve uma arquitetura Peer-to-peer.

A componente do servidor é composto por um sistema distribuído em quatro máquinas. Este sistema, tolerante a falhas, implementa uma arquitetura baseada em Primary Backup, na qual um dos nós recebe pedidos do cliente enquanto que os outros replicam o que acontece no servidor principal. O servidor líder aceita conexões TCP dos clientes, escuta os seus pedidos, processa-os concorrentemente, possivelmente notificando os servidores de backup caso haja uma mudança no estado interno e finalmente responde ao cliente com o resultado do seu pedido. É totalmente concorrente e assíncrono. Tem também a capacidade de notificar todos os clientes autenticados, e proporciona comunicação segura tanto entre si e os clientes como entre si e os servidores de backup.

A componente do cliente é semelhante no que diz respeito à concorrência e assincronismo. O utilizador requisita pedidos ao cliente, o cliente envia-os para o servidor e aguarda resposta. Não é necessário obter a resposta a um pedido para se iniciar outro.

Existe um pedido em específico (pedir uma faixa de áudio enviada por um cliente que está online) que permite a utilização de uma componente Peer-to-peer. Neste caso, e de modo a não sobrelotar o servidor, o pedido é encaminhado para o cliente que possui a faixa de áudio, sendo que este a envia ao cliente que a requisitou. A Figura 2 exemplifica a arquitetura do sistema completo.

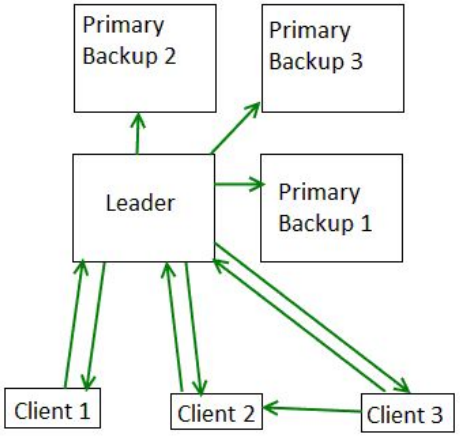

Imagem 2: Arquitetura global do sistema

3. Implementação

3.1. Componentes

3.1.1. Cliente

O cliente é uma aplicação multithreaded, emitindo pedidos do utilizador e recebendo as respostas de forma assíncrona. Vários pedidos podem ser realizados ao mesmo tempo, não havendo problemas de concorrência.

Quando um cliente inicia, tenta conectar-se a um dos servidores disponíveis no ficheiro servers.json usando uma conexão TCP com SSL. Se a conexão não for possível (normalmente porque o servidor não está online), passa para o próximo da lista, iterando até encontrar um servidor que lhe aceite o pedido. Se o servidor a que se conectar falhar e terminar a conexão, o processo volta a ser repetido até encontrar um novo servidor que esteja a aceitar pedidos. Este mecanismo de conexão é realizado por uma thread iniciada logo no início do programa chamada RequestConnect.

Existe uma consola que recolhe os pedidos do utilizador e os transforma em pedidos ao servidor. Cada pedido é realizado numa thread, havendo um tipo de thread por pedido. Estas threads (que estendem uma thread de pedido genérica chamada Request) processam os argumentos do utilizador e enviam uma mensagem para o servidor. Estas threads são lançadas a partir de uma classe chamada Service, sendo que cada método público pode ser invocado a partir da consola. Todas estas threads são controladas por um ThreadPoolExecutor.

As respostas do servidor são recebidas por uma thread especial chamada ServerListener, que está constantemente à escuta das respostas do servidor. Este "listener" lê bytes sequencialmente até encontrar um NULL, o que lhe indica que a mensagem está completa. De seguida, a mensagem é processada pelo método processMessage, que verifica se a mensagem é válida e, se não for, descarta-a. Por cada resposta existe também uma thread própria, desta vez derivadas de uma thread genérica chamada ResponseHandler. À semelhança das anteriores, são também controladas por um ThreadPoolExecutor.

Existe ainda uma outra thread, PeerListener, que é em tudo semelhante a ServerListener mas que escuta mensagens por parte de outros clientes. Apenas uma mensagem é processada aqui, mensagem essa que contém um ficheiro de áudio para reproduzir.

Todas as threads têm acesso a um singleton, ClientManager, que é thread-safe e que providencia dados e operações comuns a todas as threads.

Esta arquitetura está representada na seguinte figura:

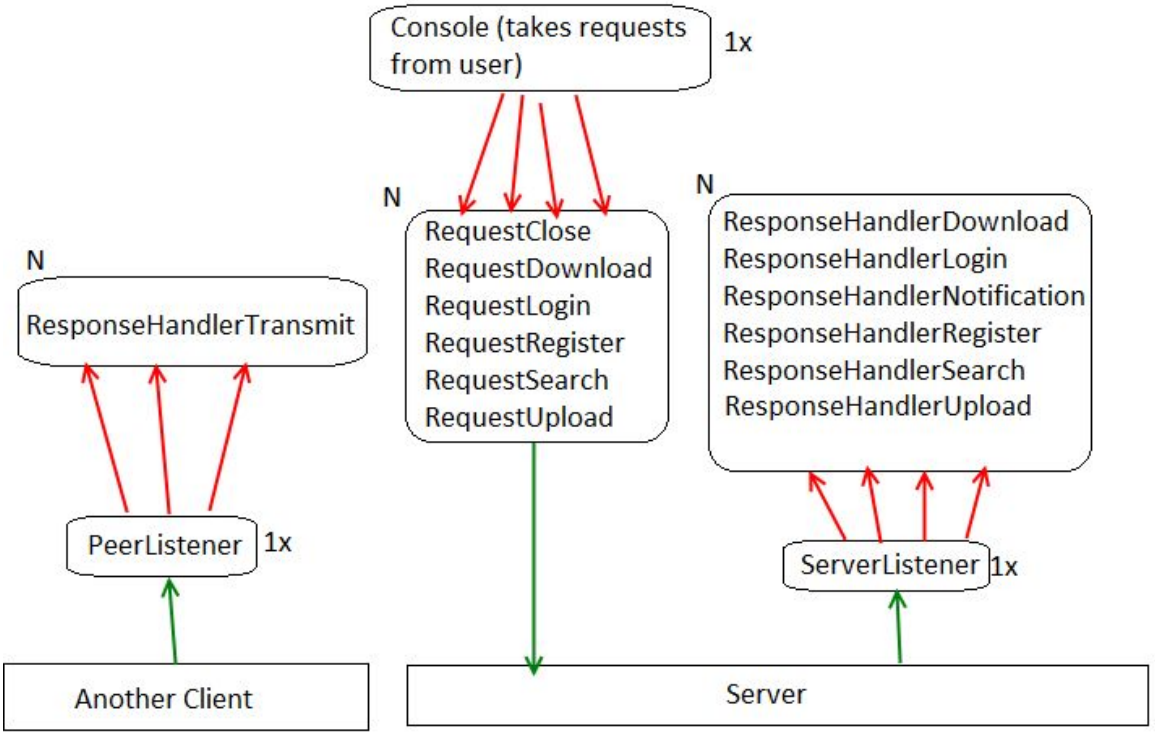

Imagem 3: Arquitetura multithreaded do cliente. Setas vermelhas: inicia uma ou mais threads. Setas verdes: direção das mensagens via TCP

Todos os pedidos e respostas são realizados a partir de mensagens passadas através de sockets TCP com SSL. O formato das mensagens são os seguintes:

3.1.1.1 Pedidos

Estas mensagens são enviadas do cliente para o servidor:

```
CLOSE
DOWNLOAD <username> <password> <title>
LOGIN <username> <password> <P2P port>
REGISTER <username> <password>
SEARCH <username> <password> <search term>
UPLOAD <username> <password> <title> <body>
```

Cada mensagem é terminada por um NULL e cada elemento é separado por um espaço. No caso de um dos elementos conter um NULL ou um espaço é realizado um encoding para Base64 desse elemento usando os métodos próprios do Java para esse efeito.

3.1.1.2 Respostas

Estas mensagens são enviadas do servidor para o cliente, e seguem as mesmas regras de formatação das anteriores. Status é um valor binário, em que 0 significa erro e 1 sucesso:

```
RES_CLOSE <status> <message>
RES_DOWNLOAD <status> [<title> <body>]
RES_LOGIN <status> <message>
RES_REGISTER <status> <message>
RES_SEARCH <status> <results>
RES_UPLOAD <status> <message>
TRANSMIT <peer port> <track> <peer ip>
NOTIF <content>
```

3.1.1.3 Outras

Esta mensagem é usado no mecanismo peer-to-peer de modo a transmitir uma faixa de áudio de um cliente para outro:

```
TRACK <title> <body>
```

3.1.2 Servidor

O servidor é, tal como o cliente, uma aplicação multithreaded que recebe, de forma assíncrona e paralela pedidos do utilizador, processa-os e emite uma resposta. A thread principal do servidor, Server, escuta pedidos de conexão TCP por parte dos clientes e, por cada pedido aceite, inicia uma nova thread para esse cliente chamada ClientListener. Esta thread existirá durante toda a sessão do utilizador, e ficará à escuta de mensagens por parte do mesmo. Segue exatamente a mesma lógica da thread ServerListener do cliente: lê bytes até encontrar um NULL, verifica se a mensagem é válida e, se for, inicia uma nova thread para tratar do pedido da mensagem. Novamente, existe um tipo de thread para cada pedido, sendo que todas estendem a thread abstrata Handler. Quando uma destas threads Handler acaba de processar o pedido, é emitida uma resposta para o cliente. Tal como no cliente, também aqui são usados ThreadPoolExecutors.

Se um pedido implicar a notificação dos clientes online, uma nova thread chamada Notifier é criada com o objetivo de notificar todos os clientes online exceto o cliente que provocou a notificação.

À semelhança do cliente, existe um singleton thread-safe chamado ServerManager que possui informação útil a todas as threads: informação sobre os clientes online que estão autenticados (útil para as notificações), informação sobre os ficheiros armazenados e quais os seus donos e por fim qual a porta que cada cliente usa para realizar conexões peer-to-peer. O registo dos utilizadores (username e password) e as faixas de áudio são guardadas de forma estável no sistema de ficheiros recorrendo ao mecanismo de serialização do Java. Uma thread chamada StateSaver guarda a informação usando este mecanismo de meio em meio segundo, sendo que os ficheiros em si são guardados diretamente no sistema de ficheiros.

No caso do peer-to-peer, o servidor verifica na lista de clientes autenticados se o cliente que enviou originalmente a faixa de áudio requisitada está online e, se estiver, manda-lhe uma mensagem de TRANSMIT. Esta mensagem, como especificada acima, inclui o endereço e a porta na qual o cliente que quer a faixa de áudio está à escuta de pedidos P2P.

Existem ainda outras classes e threads, mas essas serão explicadas posteriormente devido ao facto de abordarem problemas mais complicados.

A arquitetura do servidor está exemplificada na Figura 3.

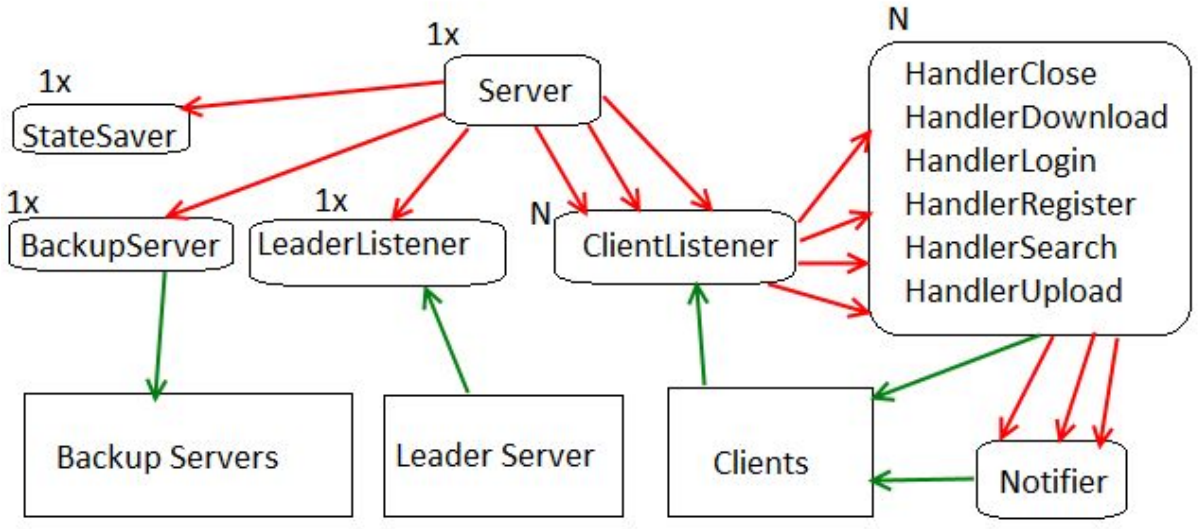

Figura 4: Arquitetura multithreaded do servidor. Setas vermelhas: inicia uma ou mais threads. Setas verdes: direção das mensagens via TCP

3.2. Bibliotecas e Frameworks

3.2.1. LibGDX

Os pedidos por parte do utilizador são realizados através de uma consola interativa construída usando o framework LibGDX. Este framework tem o objetivo principal de servir como uma base para o desenvolvimento de videojogos em Java, mas foi reaproveitado para este projeto de modo a produzir uma consola que separasse as linhas de input das de output e que suportasse a escrita concorrente de mensagens. Esta consola, contudo, não faz parte das funcionalidades básicas da biblioteca, visto que é um módulo produzido pela comunidade. O LibGDX foi também usado para reproduzir as faixas de áudio concorrentemente e para ler o ficheiro JSON previamente mencionado.

4. Informações Relevantes

4.1. Segurança

A segurança foi abordada usando SSL sobre TCP para a comunicação. Cada servidor tem a sua própria keystore, e tanto a comunicação entre clientes e servidores como a comunicação entre servidores e servidores fazem uso deste mecanismo. Na comunicação P2P tal não acontece, mas dado que não existem dados comprometidos nesta comunicação não existe nenhuma preocupação relevante em relação à segurança.

4.2. Tolerância a Falhas

Como mencionado anteriormente, foi implementado um mecanismo de tolerância a falhas. Este mecanismo envolve a replicação de servidores segundo um modelo de Primary Backup. Este modelo entende que existe um único servidor à escuta dos pedidos dos clientes e ao qual se chamará de leader. Este leader, por sua vez, estará conectado a um número de servidores auxiliares e idênticos, idealmente alojados em máquinas diferentes. No caso da nossa aplicação foi contemplado um número máximo total de 10 servidores, se bem que na prática apenas se pode usar 4 devido às keystores únicas que foram geradas para cada servidor.

Quando o leader recebe uma mensagem de um cliente, essa mensagem é reenviada para as réplicas de modo a ser replicada. A garantia de que a mensagem é entregue às réplicas é garantido pela conexão TCP. É também de notar que não vale a pena reenviar todas as mensagens para as réplicas; devem ser enviadas apenas aquelas que mudam de forma permanente o estado interno de um servidor. No contexto do nosso trabalho, essas mensagens são as de UPLOAD e REGISTER, que envolvem, respectivamente, o armazenamento de uma faixa de áudio e o registo de um utilizador. São também operações idempotentes, pelo que não haverá problema caso seja necessário fazer alguma retransmissão.

A nível de implementação, um servidor começa por decidir se é o leader ou um backup. Esta eleição é feita segundo o seguinte algoritmo:

1. o servidor tenta conectar a todos os servidores com ID inferior ao seu. Se conseguir conectar a um, então é backup;
2. Se não conseguir, espera uma quantidade de tempo proporcional ao seu ID e tenta conectar de novo, mas desta vez à gama inteira de servidores (independentemente do ID). Se conseguir conectar a algum, então é backup.
3. Se não conseguir, é leader e passa a poder aceitar conexões dos clientes assim como de quaisquer outros futuros servidores replicados.

Este mecanismo de eleição está no método electLeader na classe Server. Dependendo do resultado, runServer ou runBackup são chamadas.

runServer tem o comportamento descrito previamente, só que a esse comportamento se adiciona a conexão com as réplicas e a notificação de todas as réplicas sempre que uma mensagem relevante é recebida. A conexão com as rélocas é realizada através da thread BackupServer, que aceita conexões de réplicas e guarda-as no ServerManager para poderem ser futuramente notificadas. Essas notificações ocorrem no método notifyBackups de ServerManager, no qual todos os backups são notificados. Visto que algumas destas mensagens podem ser bastante grandes, é realizada concorrência na escrita para as sockets, sendo que todas as escritas ocorrem ao mesmo tempo. Uma thread "inline" é criada para esse efeito (linha 290). O processamento do pedido do cliente não avança enquanto todos os backups não forem notificados, o que contribui para a consistência entre réplicas. Por sua vez, as réplicas têm uma outra thread, LeaderListener, que escuta as notificações do leader. A implementação deste listener segue os mesmos modos dos usados para a interação cliente-servidor.

Sempre que uma réplica de backup falha, nada de especial ocorre, sendo que o leader simplesmente deixa de notificar essa réplica. Contudo, quando o leader falha, as réplicas correm todas o algoritmo de eleição ao mesmo tempo, até que uma delas é eleita leader. Essa réplica passará a aceitar conexões dos clientes (que por sua vez estão constantemente à procura do novo servidor depois do anterior falhar) e o funcionamento normal resume-se.

O único caso não totalmente contemplado é o caso de uma réplica ficar online a meio de uma execução normal. Neste caso seria necessário atualizar o estado interno da réplica com todas as informações que foram guardadas durante todo o tempo em que esteve offline. Tentou-se implementar um mecanismo de transferência do estado interno do leader para a nova réplica, mas por constrições de tempo tal não foi possível.

4.3. Implementação de Notificações

O mecanismo de notificações já foi previamente mencionado e brevemente explicado, mas segue-se uma explicação mais detalhada.

As notificações dos clientes ocorrem sempre que certas ações são despoletadas, e são enviadas para todos os utilizadores autenticados exceto para aquele que despoletou a ação. Estas ações são: utilizador fica online, utilizador fica offline e utilizador envia uma nova faixa de áudio para o servidor. Em cada uma das threads que processam pedidos deste tipo existe a criação de uma nova thread, chamada de Notifier, que percorre a lista de utilizadores autenticados e notifica-os, um a um, da ocorrência. Ao contrário das notificações entre servidores, estas mensagens são muito curtas e rápidas de transmitir, pelo que não se justifica ainda mais multithreading para o envio concorrente das notificações para todos os clientes.

4.4. Escalabilidade

A componente Peer-to-peer foi criada de modo a libertar alguma carga sobre o servidor, visto que se elimina a necessidade de transmitir ficheiros possivelmente grandes. Isto ajudaria a obter uma maior escalabilidade visto que permitiria ao servidor responder a mais pedidos do que aqueles que normalmente poderia, sendo que se está a eliminar a necessidade de mandar constantemente ficheiros grandes a todos os clientes que o pedirem.

5. Conclusão

Para finalizar, com este projeto conseguimos compreender melhor o quão importante é a escolha da estrutura de implementação e dos protocolos a utilizar, assim como que estratégias utilizar para resolver alguns problemas que este tipo de aplicações implica como, por exemplo, a necessidade de haver tolerância a falhas e também a garantia de segurança que é essencial numa aplicação como esta. Os desafios arquiteturais foram muito mais relevantes do que os desafios de implementação, o que é raro quando comparado com os outros projetos universitários que tivemos até agora.

Algumas melhorias que podem futuramente ser implementadas consistem em aplicar um serviço baseado em streaming, possivelmente integrado no sistema de notificações, poderá ser implementado, ao invés do actual sistema de download e só depois reproduzir, assim como a melhoria da interface do cliente para uma interface gráfica melhorada. Poderíamos também acrescentar a API do facebook para que os clientes pudessem ligar a sua conta a este serviço e, também, publicar no Facebook uma mensagem sempre que o cliente associado a essa conta faça upload de uma faixa de áudio. Por fim, e agora olhando para os aspectos mais teóricos de sistemas distribuídos, poderia ter sido implementado um sistema de replicação mais realista, passando por incluir um algoritmo de eleição de líder mais sofisticado e um melhor tratamento da consistência entre réplicas. Por fim, poder-se-ia considerar um modelo de replicação que também fosse escalável, com possivelmente vários servidores conectados a clientes em vez de apenas um.