Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática

Troca de ficheiros

A85227 João Pedro Rodrigues Azevedo
A85729 Paulo Jorge da Silva Araújo
A83719 Pedro Filipe da Costa Machado

Braga
Janeiro 2020

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

Este trabalho foi realizado no âmbito da unidade curricular de Sistemas Distribuídos do curso de Mestrado Integrado em Engenharia Informática.

Esta UC, assim como este trabalho prático, tinha como principal objetivo a utilização de conhecimentos Teóricos e Práticos relativos à conceção de sistemas capazes de operar num ambiente distribuído envolvendo interação com os mesmos através de conexões TCP, ponto a ponto, que em Java se designam por Sockets. Além disso, este projeto incide na manipulação de ficheiros, gestão de *threads*, memória e justiça entre pedidos concorrentes.

1.2 Descrição do problema

A ideia por detrás deste projeto está na criação de uma plataforma de troca de ficheiros (neste caso, músicas). Uma música será uma entidade deste sistema e terá associada à mesma o seu título, artista e um número variável de etiquetas para que a mesma possa, posteriormente, ser pesquisada.

Pretende-se também a implementação de funcionalidades como *Upload*, *Download* e gestão da congestão da memória causada pela transferência de ficheiros entre o cliente e o servidor.

Numa vista mais concorrente, será também valorizada a implementação de uma limitação de transferências e uma justiça relativa entre os clientes que esperam obter prioridade de transferência. Outras funcionalidades serão também valorizadas, como a apresentação de notificações no momento de upload de novas músicas.

Capítulo 2

Servidor

2.1 Arquitetura e funcionamento geral

O servidor é criado inicialmente usando a classe disponibilizada pelo Java designada por **Server Socket**, onde é explicitado a porta que deve escutar os pedidos dos clientes e o endereço pelo qual o servidor deve estar hospedado.

A classe que corre o programa servidor é a classe **ServerApp** onde são inicializadas as estruturas principais do sistema para guardar os dados na memória principal, como a classe **FileSharingSystem** que guarda os utilizadores e o conteúdo carregado no sistema e a inicialização da classe **Config** que armazena as definições principais ao funcionamento do servidor, classes estas que serão faladas com mais pormenor nas próximas secções..

```
Config cnf = new Config();

try { cnf.init(); } catch (Exception e) { GeneralMessage.show(0, "app", "error initializing config", true); return;}

FileSharingSystem fss = new FileSharingSystem();
```

Figura 2.1: Inicialização das estruturas principais.

Uma característica importante num ambiente cliente/servidor como o pedido está em permitir a aceitação de múltiplos clientes ao mesmo tempo e uma resposta paralela aos mesmos, sem que a demora de uns leve à demora da resposta do Servidor a outros. Temos portanto um *Worker* ao qual chamamos de **ServerWorker**, presente na classe **Server** e que é responsável por lidar com o *Input/Output* da informação obtida/enviada pelo *Socket* para o destino final, o Cliente. Esta classe é corrida em paralelo, com o auxílio de novas *Threads*. Temos então um servidor a correr num ciclo *"infinito"* no qual para cada cliente criámos um socket (ponto de conexão) em que o *IO* é gerido pelo **ServerWorker**.

```
public boolean start() {//running server
    ...
    while(true) {
        ...
        client_socket = server_socket.accept();//Um socket por cliente
        ...
        //Um ServerWorker para gestao dos pedidos do cliente
        try {
        sv_worker =
            new ServerWorker(client_socket, this.system,
                             this.config, this.downloadRequestBuffer);
```

```
            } catch(...){...}
            ...
            new Thread(sv_worker).start(); //multi-threaded ServerWorker
            ...
        }
        ...
}
```

Deste modo, a classe **ServerWorker** recebe o pedido do cliente sob a forma de uma *String* e faz *parsing* da mesma de modo a executar o pedido e recolhe a informação que possa vir no mesmo. Com a nossa implementação assume-se que todas as mensagens que chegam ao servidor estão formatadas da forma correta visto que a aplicação **Cliente** sabe como as enviar.

2.2 Estruturas Partilhadas

A estrutura que contém os dados e operações necessárias à gestão de Utilizadores e de Conteúdo (músicas) do sistema foi designada por **FileSharingSystem** (da *package* app.model) e traduz-se numa abstração de 2 outras classes, entre elas:

```
public class FileSharingSystem {

    private ContentDB content;
    private UsersDB users;

    ...

    public boolean authenticate(String name, String pass){...}
    public User get_user(String name) {...}
    public boolean register_user(String name, String password) {...}
    ...
    public List<Music> search(String tag) {...}
    ...
}
```

1. **UsersDB**: Esta classe armazena todos os Utilizadores do sistema e contém duas estruturas que serão partilhadas por todas as *threads*:

   ```
   private Map<String, User> users; //Users Registados
   private Map<String, User> current_authenticated_users;//Logged-in
   ```

   Todo o controlo de concorrência com *locks* e *synchronized* é feito nesta classe e na classe que guarda um utilizador individual, a classe **User**. Este controlo é feito por estas classes para que o Servidor não fique comprometido com a consistência dos dados de modo a apresentar a informação correta aos clientes.

2. **ContentDB**: Esta classe armazena todas as músicas do sistema indexadas pelo seu ID:

   ```
   private int lastID; //ultimo id de uma musica
   private Map<Integer, Music> musics; //musicas do sistema
   ...
   public List<Music> filter_tag(String tag){...}
   public int add_content(String title, String artist, int year,
   List<String> list_of_tags){...}
   ...
   ```

Obviamente, o acesso a estas estruturas também tem controlo de concorrência através do uso de *Locks* explícitos, nomeadamente, no acesso ao **lastID** que contém o identificador **único** da última música adicionada.

2.3 Funcionalidades principais

Todas as funcionalidades básicas da aplicação pedida foram implementadas. Iremos dar destaque àquelas que se revelaram mais complicadas:

- **Upload**: Tendo consciência que a memória principal do computador é, sobretudo, volátil, decidimos implementar, desde já, a transferência de ficheiros respeitanto a valorização 3 do enunciado em relação ao *MAX_SIZE* de bytes em memória, do lado do Cliente e do Servidor. Para isso, foi necessário dividir a transferência do ficheiro dos dois lados em *chunks* de tamanho máximo *MAX_SIZE* e enviar a *stream* de bytes do ficheiro por *chunks*, isto é, *arrays* com este formato - byte[] chunk = new byte[MAX_SIZE];

```
while ((count = fis.read(chunk)) > 0) {
    out_socket.println(Base64.getEncoder()
                                .encodeToString(Arrays
                                .copyOfRange(chunk, 0, count)));
    out_socket.flush();...}
```

  Onde "fis"é o **FileInputStream** do ficheiro, o "out_socket"corresponde ao **PrintWritter** para enviar os bytes para o socket e o "count"é o número de bytes lidos na iteração atual. De notar que o mecanismo de tradução de bytes para chars adotado foi o do java.util.Base64 e operações como **decode** e **encondeToString**. Deste modo, foi apenas necessário "replicar"o mecanismo de leitura do lado do servidor. De realçar que foi também enviado pelo *Socket* o número de bytes do ficheiro a transferir de modo a efetuar um simples *checksum* para verificar se o ficheiro foi ou não corrompido no envio.

- **Download (versão inicial)**: O download, numa versão inicial, segue a mesma lógica do Upload, sendo que o mecanismo de transferência, deixa de ser do Cliente para o Servidor e passa a ser o contrário.

- **Download (valorização)**: Esta versão do download, pedida no enunciado, revelou ser o mecanismo mais complicado de implementar neste projeto. Na implementação seguimos a política seguinte:

  1. Criação de um **BoundedBuffer**: Foi criada a classe **DownloadRequestsBuffer** (na package app.server.tools) que implementa um buffer onde todos os clientes que não ultrapassem o limite *MAX_DOWN* são aceites e os outros ficam em espera tendo para isso duas variáveis **Condition**:

     ```
     private Condition smallDownloadCondition;
     private Condition bigDownloadCondition;
     ```

  2. Divisão dos clientes em **2 partes**: Clientes com downloads "grandes"e clientes com downloads "pequenos". Aqui critério de divisão em grandes e pequenos está a cabo de uma fase de testes, mas foi considerado grande o ficheiro que ultrapassa-se os **51229 bytes**, no entanto, o grupo tem consiência que esse é um número irrealista, sendo usado apenas para testes.
  3. **Justiça relativa** entre clientes: Para garantir uma justiça mínima entre os clientes em espera nas Conditions, decidimos que o mais simples a fazer seria dar prioridade a 3 clientes com downloads pequenos e depois a 1 com download grande. Aqui a relação 3 para 1 garante que nunca temos muitos clientes seguidos com downloads

grandes, sendo a prioridades destes mais pequena. Com a falta de tempo, não conseguimos garantir que a ordem de chegada para obtenção de conteúdo seja garantida, visto que, o tempo para testes foi curto, tendo apenas testado o programa com **Thread.sleep(ms)**. No entanto, o conceito implementado é semelhante ao *Read/Write Lock* desenvolvido nas aulas práticas.

- **Outras Funcionalidades**: Foram desenvolvidas outras funcionalidades básicas como: Autenticação, Registo e Log-out de um Utilizador e a procura de músicas através da sua Tag. Nesta última, damos relevo à forma como os resultados são apresentados, no qual seria uma má implementação enviar uma string enorme com os resultados e ler tudo do lado do cliente pelo que decidimos enviar o número de linhas a ler e efetuar uma leitura/escrita sincronizada (linha-a-linha) do lado do Servidor/Cliente.

Capítulo 3

Cliente

3.1 Aplicação Cliente

Na aplicação Cliente optamos por não dar tanta liberdade de *input* por parte do utilizador, pelo que os comandos que são enviados ao servidor são formatados de acordo com a opção que o cliente escolhe:

```
StringBuilder mainmenuSB = new StringBuilder();

mainmenuSB.append("-------------------------------------------------\n");
mainmenuSB.append("                   Main Menu                     \n");
mainmenuSB.append("-------------------------------------------------\n");

mainmenuSB.append("e: Exit the system | a: Authenticate | r: Register\n");

mainmenuSB.append("-------------------------------------------------\n");
```

Figura 3.1: Método que apresenta o menu principal.

Na imagem vista acima, da classe **GUI** da package app.utils, podemos ver que o utilizador não insere comandos diretamente para o servidor mas sim, escolhe opções e insere o que lhe é pedido na consola. A classe **Input** da package app.utils controla o input recebido com métodos para ler Strings, Integers, etc...

Mas, claramente, a verificação da autenticação, procura de dados e acesso aos mesmos é toda feita pelo classe que os pode aceder, os **ServerWorker**, através de uma estrutura partilhada referida no capítulo anterior.

3.2 Outras Classes e Ferramentas

Aproveitamos esta secção para apresentar algumas das classes que pertencem ao mecanismo da aplicação, tanto do cliente como do servidor, e que nos ajudaram a efetuar *debugging*, a organizar e estruturar mais o nosso projeto.

- **Package app.utils**:

1. Config: Permitiu carregar um ficheiro "config.cnf"que guarda todas as variáveis globais do nosso projeto, como o tamanho máximo de download (MAX_DOWN), o *host* do servidor, a porta de escuta,...
2. GUI, GeneralMessage, DateAndTime, Input: Para interação com o utilizador e *debugging de erros*.

Capítulo 4

Conclusão

A topologia de Cliente/Servidor pedida neste problema e na grande maioria dos assuntos estudados nesta Unidade Curricular permitiu-nos criar sistemas capazes de responder concorrentemente a pedidos de clientes de dispositivos diferentes, identificados pelo seu IP com conexões de partilha de pacotes TCP.

A concorrência é, assim, o emblema desta UC, onde foi possível tratar acessos concorrentes a zonas críticas do programa com recurso à tão conhecida técnica de exclusão mútua com *Locks explícitos* do Java.

No entanto, nem tudo correu bem na realização deste projeto, pois tivemos diversas situações de threads a bloquear (deadlock) o que torna as situações de *debugging* mais complicadas, principalmente na implementação da valorização relativa ao limite de downloads, que envolvia a manipulação de muitas condições de espera e também garantir que o processo cliente não morre por estar à espera de resposta do servidor.

Por outro lado, não foi implementada a segunda valorização por falta de organização de tempo do nosso grupo, pelo que, numa implementação futura desta aplicação, tal funcionalidade seria bastante útil. Para implementar esta valorização temos em mente que tal poderia ser feito com o auxílio de um Socket extra por cliente, criado aquando o login efetuado com sucesso e estaria numa espera constante por respostas do servidor, no entanto, consideramos que esta solução rapidamente esgotaria o número de sockets abertos visto que teríamos 2 conexões ativas por cliente e a gestão de conexões seria um ponto importante a abordar. Uma outra alternativa, que alteraria, em grande maioria, a estrutura do nosso trabalho, seria estabelecer uma mensagem **fixa** entre o Cliente/Servidor, onde poderíamos incluir o *header*, o tamanho da mensagem, e outros campos de modo a padronizar a leitura e a escrita para o socket e, com isto, poderíamos enviar mensagens de notificação que poderiam intervalar com mensagens de envio de *chunks* de um tranferência, sem ter de criar um *socket* extra por cliente.

Concluindo, este trabalho foi dos mais interessantes que desenvolvemos neste curso, pelo que se trata dos assuntos que o grupo mais se interessa ao nível da ciência de computadores.