Comunicações por Computador

Relatório do Trabalho Prático nº2

Desenho e implementação de um sistema de partilha de conteúdos áudio

Diogo Constâncio (a70719), Diogo Meira (a71329), Ruben Santos (a70644)

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal

email: {a70719, a71329, [a70644}@alunos.uminho.pt](mailto:a70644%7d@alunos.uminho.pt)

**12 de Junho de 2016**

**Abstract.** A realização do trabalho foi feita em três fases. Neste relatório explicaremos o processo de desenvolvimento de cada uma destas fases.

* **1ª Fase**: Implementação de versões básicas do servidor e cliente em que seja possível utilizar o serviço de registo de clientes.
* **2ª Fase**: Implementação do sistema de descoberta de ficheiros de áudio, passando por consulta do servidor local e resposta sobre existência ou não do ficheiro. É esperada também a inclusão de mecanismos de PROBING e de transmissão de dados entre os clientes.
* **3ª Fase**: Implementação do funcionamento interdomínio, envolvendo o registo e interação entre servidores pertencentes a domínios distintos.

**Keywords: Protocolo UDP, Protocolo TCP, PROBING, Transferência de Dados**

1. Introdução

O objetivo principal deste trabalho prático consiste na implementação de um ambiente distribuído de troca de ficheiros, nomeadamente conteúdos áudio, entre utilizadores registados em servidores existentes em cada domínio de rede. Estes utilizadores poderão interagir diretamente entre os utilizadores, escolhendo com quem trocarão conteúdos, com base em indicadores de desempenho da rede.

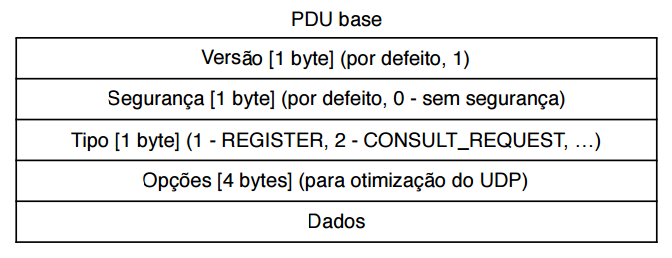
No caso de um ficheiro não estar disponível para ser partilhado num domínio, o acesso a esse ficheiro é concretizável recorrendo à comunicação entre servidores.

A transmissão de ficheiros é feita recorrendo ao protocolo UDP que se caracteriza pela falta de fiabilidade, ausência de mecanismos de controlo de fluxo e de congestão na rede. O controlo de erros neste protocolo é diminuto e opcional. No caso da transferência de um elevado volume de dados é conveniente desenvolver uma camada protocolar sobre UDP que permita a ordenação e recuperação de dados perdidos. As comunicações UDP neste contexto têm de ter funcionalidades acrescidas, ordenação e retransmissão.

É-nos imposto que a interação entre um servidor e os seus utilizadores seja efetuada pelo protocolo TCP, assim como a interação entre os vários servidores. O TCP caracteriza-se por ser um protoloco orientado à comunicação, o que significa que a conexão é mantida até as aplicações acabarem de trocar mensagens, numa comunicação ponto a ponto.

* 1. Funcionamento do serviço

Este trabalho prático impõe que seja utilizado o mesmo PDU base para todas as comunicações aplicacionais.



* Cada cliente tem de se registar no respetivo servidor enviando para isso um pedido REGISTER, com o seu ID, endereço IP e porta TCP onde irá receber consultas enviadas pelo servidor.
* O servidor da rede local mantém uma lista com todos os clientes ativos associados a ele e de outros servidores existentes em domínios diferentes.
* Uma consulta de um ficheiro específico implica o envio de uma mensagem do tipo CONSULT\_REQUEST para o servidor do cliente.
* A ação do servidor perante um *request* é o envio da mensagem CONSULT\_REQUEST a todos os clientes do seu domínio, aguardando uma resposta durante um período de tempo. Na eventualidade de nenhum dos clientes associados possuir o ficheiro, a consulta é enviada para outros servidores. Caso o ficheiro não exista em nenhum outro servidor, é enviada uma resposta ao cliente do tipo CONSULT\_RESPONSE informando a não existência do ficheiro (NOT\_FOUND).
* Cada cliente, ao receber a consulta do servidor responde com uma mensagem do tipo CONSULT\_RESPONSE, que contém o seu ID, endereço IP e porta UDP pela qual aceitará a comunicação com outros clientes.
* Caso o ficheiro seja encontrado, o servidor envia ao cliente que fez a consulta uma mensagem do tipo CONSULT\_RESPONSE (FOUND), contendo a lista de clientes que possuem o ficheiro.
* De maneira a tornar a transferência mais eficaz, o cliente envia uma mensagem do tipo PROBE\_REQUEST para os todos clientes que possuem o ficheiro.
* Após a definição do cliente que apresenta melhores condições para a transferência, o cliente que pretende o ficheiro envia uma mensagem do tipo REQUEST, contendo o nome do ficheiro.
* O cliente que recebe o REQUEST inicia o envio do ficheiro através de mensagens do tipo DATA.
* A descoberta por novos servidores é feita através de um servidor central que mantém o registo de todos os servidores ativos. O registo é feito seguindo o mesmo processo do registo dos clientes, obrigando para tal um atributo que permita diferenciar clientes de servidores.

Resumindo, cada tipo de mensagem tem um código identificador e tem diferentes tipos de parâmetros:

[1] REGISTER : Tipo (in/out) | Username | Password | IP | Porta

[2] CONSULT\_REQUEST: musica.extensao

[3] CONSULT\_RESPONSE: Tipo (FOUND/NOT\_FOUND) | NumeroHosts (nº clientes que possuem) | ID | IPs | Portas

[4] PROBE\_REQUEST

[5] PROBE\_RESPONSE: Timestamp

[6] DATA: Transferência feita por UDP, com tamanho máximo de pacote (48 kB), retransmissão de pacotes perdidos.

* 1. Mecanismo de PROBING

Tal como referido anteriormente, antes de iniciar a transferência, o cliente que tem interesse em obter o ficheiro envia uma mensagem do tipo PROBE para todos os outros clientes que possuam o ficheiro em questão, com vista a determinar qual deles apresenta menor latência de transmissão. Cada cliente que recebeu o pedido de PROBE responde de volta com uma mensagem do tipo PROBE\_RESPONSE, contendo o *timestamp* para calcular a latência aproximada para cada uma das possíveis fontes, decidindo para qual fará o REQUEST com base no que tiver menor *ping*.

1. Implementação

A aplicação está dividida em três *packages*, cliente, servidor e classes comuns a esses dois.

* 1. Client

Foram implementadas duas classes referentes ao cliente, Client e FileDB.

A classe **Client** fornece vários métodos que garantem o funcionamento conforme o pretendido.

* Escolha do servidor a qual se ligar da primeira vez, recorrendo ao método de probing.
* Registo de um novo cliente, com *username* e *password*.
* Efetuar login, com base num registo previamente realizado.
* Tratamento de *requests.*
* Efetuar logout da aplicação.

Bibliotecas usadas:

* java.io.\*;
* java.net.\*;
* java.util.Date;
* java.util.Iterator;

A classe **FileDB** foi implementada para guardar a lista de ficheiros do cliente associado.

* Criação da base de dados.
* Atualização da base de dados.

Bibliotecas usadas:

* java.io.File;
* java.util.ArrayList;
  1. Server

Relativamente ao Server foram implementadas seis classes sendo elas ClientHandler, Login, Main, Packet, Server e Users.

A classe ClientHandler foi definida para que o servidor possa lidar com os diferentes clientes que pretendam utilizar o serviço.

Para tal fornece métodos para tratar a informação recebida pelo cliente conforme o tipo de request efetuado pelo mesmo, handle. Este método recebe um *Packet* que pode ter um de quatro tipos entre registo, login, *consult request* e *probe request*, e conforme o tipo do *Packet* recebido age concordantemente.

Bibliotecas usadas:

* java.io.\*;
* java.net.\*;
* java.security.NoSuchAlgorithmException;
* java.util.ArrayList;
* java.util.Date;
* java.util.Iterator;
* java.util.concurrent.locks.ReentrantLock;
* java.util.logging.Level;
* java.util.logging.Logger;

Uma classe Login foi também criada para que o login de diferentes utilizadores pudesse ser efetuado na aplicação desenvolvida.

Oferece métodos para registar um utilizador através do seu *username* e *password*, eliminar um utilizador previamente registado, verificar se o cliente se encontra registado ou não, efetuar login, e todos os métodos auxiliares para o seu funcionamento correto.

Bibliotecas usadas:

* java.io.ByteArrayOutputStream;
* java.io.IOException;
* java.io.Serializable;
* java.security.MessageDigest;
* java.security.NoSuchAlgorithmException;
* java.security.SecureRandom;
* java.util.Arrays;
* java.util.HashMap;

A classe Main apenas contém um método que cria um novo utilizador, e cria e inicializa um servidor numa *thread*.

A classe Packet, já anteriormente referida, foi criada para que pudesse ser definida e armazenada a informação relativa ao *Packet* a ser enviado. Apenas contém construtores e respetivos *getters* e *setters*. Tem como variáveis o tipo, versão, segurança, dados e dados encriptados e por fim opções.

Bibliotecas usadas:

* javax.crypto.SealedObject;
* java.io.Serializable;

A classe Server tem um método pingHandler, que quando o server recebe um *Packet* do tipo *Probe Request*, guarda o instante em que o pedido foi recebido no ProResData, classe para guardar informação de *Packets* de *Probe Response*, para que seja possível calcular posteriormente o *ping*.

Adicionalmente, contém o método saveState que guarda o estado de execução do servidor. O método debuUI que vai imprimindo na consola do servidor informações sobre o que se está a passar.

Por fim foi implementado o método run, para correr o servidor e poder executar todas as funções necessárias, criando os *sockets* necessários e aguardando por conexões por parte dos clientes.

Bibliotecas usadas:

* java.lang.invoke.SerializedLambda;
* java.net.InetAddress;
* java.net.ServerSocket;
* java.net.Socket;
* java.net.SocketException;
* java.util.ArrayList;
* java.util.Date;
* java.util.HashMap;
* java.util.concurrent.locks.ReentrantLock;
* java.util.logging.Level;
* java.util.logging.Logger;

A classe Users foi criada para que se pudessem guardar vários utilizadores num TreeMap, associando a cada, um inteiro identificador.

Bibliotecas usadas:

* java.io.Serializable;
* java.util.TreeMap;
* java.util.concurrent.locks.Lock;
* java.util.concurrent.locks.ReentrantLock;

Por fim foi definida uma série de classes “comuns” ao cliente e ao servidor nomeadamente:

* FileData
* ConResData
* LoginData
* ProResData
* RegistarData
* ProReqData
* Data
* PacketTypes
* ConReqData
* User
* SerializerTodas estas classes contêm apenas construtores, e são usadas para guardar informação relativamente aos ficheiros*, packet ConectionResponse*, *login*, etc…

1. Fluxo da aplicação
2. **Servidor**

Dado que para efeitos de teste e facilidade de apresentação os servidores operam sobre a presunção de que o servidor mestre está em *localhost* o número da porta é usado para fazer a distinção entre o servidor mestre e secundário. Ambos os tipos de servidores operam uma *thread* para resposta a *pings*, uma *thread* para interação com o utilizador, e uma *thread* para persistência. Os servidores secundários registam-se com o servidor mestre através do envio de um *registerPacket*, e os seus endereços e portas são guardados. Ambos os tipos de servidores aceitam conceções na *thread* principal, criando uma nova *thread* para cada cliente. Periodicamente uma *thread* guarda num ficheiro os dados dos utilizadores, garantindo persistência.

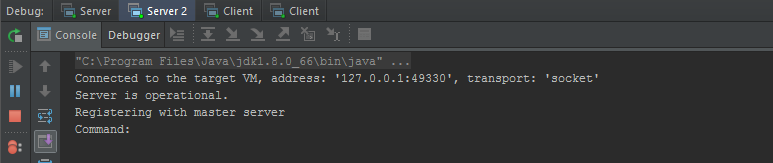
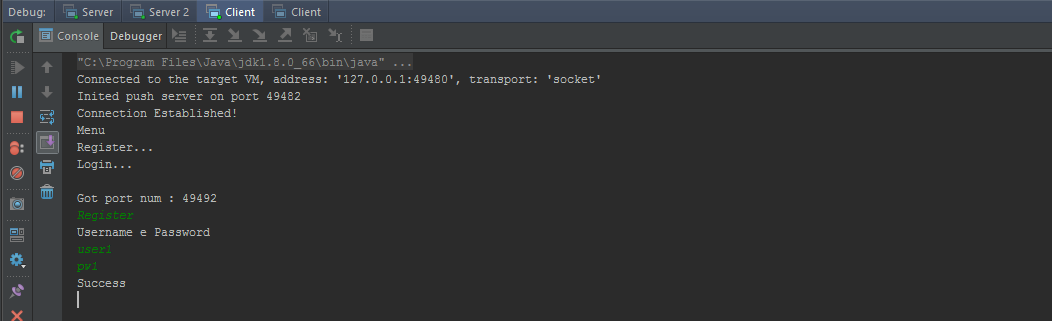
1. **Cliente**

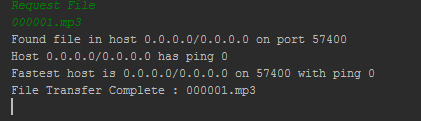
No início da execução o cliente contacta o servidor mestre, respondendo este com uma lista de todos os servidores registados até ao momento. De seguida o cliente estabelece ligação com cada um destes servidores e calcula o *ping*, escolhendo estabelecer ligação com aquele que apresente uma menor latência. De seguida a aplicação espera pelo *input* do utilizador, podendo este escolher efetuar o registo ou *login* no servidor a que se conectou. Depois de ter sido autenticada o utilizador pode escolher requisitar um ficheiro ao servido a que está ligado. Este servidor contacta todos os seus clientes, e no caso de algum possuir o ficheiro retorna ao utilizador uma lista de clientes com o ficheiro. Se nenhum dos seus utilizadores possuir o ficheiro o servidor contacta o servidor principal que encaminha o pedido para os restantes servidores. De seguida o cliente contacta todos os clientes que contêm o ficheiro, determina aquele com o menor *ping*, e inicia a transferência. A transferência usa por base UDP, fragmentando o ficheiro a ser enviado em pacotes do tamanho pretendido, enviando com cada pacote uma pequena *header* contendo informação sobre o número de sequencia do pacote enviado, informação sobre o ultimo pacote, e o número de *bytes* a ler no pacote, sendo que o cliente responde com um pequeno pacote contendo informações sobre o ultimo pacote lido, o número de *bytes* lidos, e sobre a correção dos dados. No caso de *timeouts* ou *packet loss* é reenviado o pacote em causa.

1. Testes e Resultados

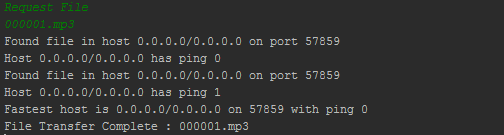
De seguida será mostrado um conjunto de capturas de ecrã que demonstram os diversos passos no funcionamento da aplicação

##### **Inicialização de servidor mestre**

1. **Inicialização de um servidor secundário**
2. **Cliente 1 estabelece ligação ao servidor mestre e regista-se (latências influenciadas para garantir ligação ao servidor pretendido)**
3. **O mesmo processo é repetido para o Cliente 2 com o servidor secundário**
4. **Cliente 2 requisita o ficheiro 000001.mp3, com o Cliente 1 a fazer uma transferência inter-servidor**



1. **Cliente 2 e 3 registam-se no Servidor mestre**
2. **Cliente 3 requisita o mesmo ficheiro**



1. **Cliente 3 requisita um ficheiro inexistente**



1. Conclusão

Após a implementação da solução para o problema proposto podemos verificar que o comportamento da aplicação é o esperado, em que se exigia uma otimização do protocolo UDP.

No que toca a trabalho futuro existem duas áreas onde podem ser feitas algumas alterações. Primeiro o mecanismo de transferência dos ficheiros com recurso a UDP poderia ser otimizado, por exemplo, enviando múltiplos pacotes sem confirmação explicita da receção do primeiro. Cada servidor mantém uma lista dos seus próprios utilizadores, como tal, em casos de anormalidade, o cliente pode vir a ser redirecionado para um servidor onde o seu registo não seja válido, então, dependendo do caso de uso em concreto, poderia ser justificada a criação de um mecanismo para a sincronização de credenciais entre servidores.