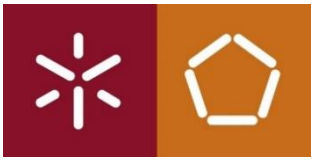
Universidade do Minho



Departamento de Informática

Comunicações por Computador

Trabalho Prático 2

A78485

Diogo Araújo

A78957

Diogo Nogueira

TransfereCC

Índice

1. Introdução

Transferências estão presentes em virtualmente todos os dispositivos conectados à Internet, como forma de comunicarem entre si. O propósito deste trabalho prático da Unidade Curricular de Comunicações por Computador é desenhar, estruturar e implementar um serviço de transferência de ficheiros que seja fiável e que ofereça um ramo de funcionalidades que suportem e corrijam as falhas de garantia existentes nas camadas mais inferiores da rede.

Segundo as instruções do enunciado do problema, foi implementado uma alternativa em que a mesma esteja a utilizar unidades de dados que caibam dentro dum datagrama UDP e que sejam enviados e/ou recebidos através de *sockets* UDP. Por cima desta ligação pré-estabelecida houve a implementação de métodos que nos fornecessem soluções para conseguir criar uma ligação fiável e capaz de lidar com vários erros, tais como perdas e duplicação de pacotes, mas também erros que o pacote pudesse ter.

Com tudo isto em mente, o projeto final consiste numa apresentação fiável, com controlo de conexão e receção ordenada capaz de enviar os datagramas UDP genéricos de forma a ter o melhor das funcionalidades presentes no protocolo TCP.

2. Especificação do Protocolo

2.1. Formato das mensagens protocolares (PDU)

Para aproveitar o melhor da linguagem Java e utilizar as *APIs* da linguagem, mais concretamente a classe *DatagramPacket*, que cria um pacote UDP pronto a ser enviado por um *socket* UDP, que neste caso foi implementado através da classe *DatagramSocket*.

Desta maneira foi pensado num *array* de 1004 *bytes* para ser o nosso *Protocol Data Unit* (PDU), no qual foi concebido a que os primeiros 4 bytes seriam o nosso cabeçalho e o restante, ou seja, 1000 *bytes*, seria para conter os dados vindos do ficheiro a transferir.

O formato da mensagem protocolar, concretamente, para o cabeçalho, teve em mente algo intrínseco à linguagem Java que tem os números inteiros definidos a 32 bits, ou seja, 4 bytes. Assim o cabeçalho tem o tamanho de 4 para conter os dados associados ao controlo de sequência e também o número de *ACKnowledge*, que são inteiros que podem ir até pouco mais de 2 milhões. Assim, a implementação funcionará para ficheiros inferiores a 2 MB o que é suficiente para demonstrar a fiabilidade e segurança do TransfereCC.

2.2. Interação protocolar

A comunicação entre as duas partes a correr o programa funciona através dos *sockets* UDP falados anteriormente que irão comunicar pacotes com os dados do ficheiro, mas também pacotes para controlar a sequência dos mesmos. Com tal no pensamento, definiu-se a porta UDP 7777 para envio e recebimento de pacotes de dados do ficheiro e também a porta UDP 9999 para enviar e receber pacotes ACK provenientes da gestão da sequencialidade dos pacotes.

3. Implementação

A aplicação está dividida em 4 módulos de fácil compreensão, sendo que temos as seguintes classes:

- Cliente, que se baseia na função de *upload* para o servidor, sendo o mesmo que envia os pacotes de dados e recebe os ACK respetivos.
- Servidor, que recebe os pacotes de dados e vai construindo o ficheiro que está a receber, enviando os ACK de forma sequencial análoga ao que está a receber.
- PacoteUDP: Onde se insere todo o código para a criação dos *arrays* de *bytes* e pacotes, mas também outros métodos que sejam utilizados nas restantes classes em comum.
- TransfereCMD: A classe onde se situa o *main* e na qual existe a inserção dos comandos GET ou PUT que iniciam a aplicação no modo cliente ou servidor.

Para começar no servidor houve umas ideias a ter na conceção do algoritmo a seguir, sendo que se assumiu a atribuição do nº de sequência em -1, para o incremento que acontecesse começasse corretamente no número de sequência 0. Dessa forma, como delineado anteriormente os números para os ACK seriam o último nº de sequência + 1000, ou seja, teoricamente igualando ao que o protocolo TCP faz com o *SEQ+1*. Como técnica final definiu-se que um pacote com o nº ACK de -5 seria enviado para o cliente como forma de confirmar que a transferência tinha sido concluída e terminada. Com estes parâmetros delineados, quando se recebe o primeiro pacote vindo do cliente, obtemos, através da leitura do *buffer/array* de *bytes*, os 4 primeiros *bytes* (cabeçalho) que nos fornecerão com o nº de sequência do pacote. Quando isso acontece, abre-se automaticamente o *stream* do ficheiro que irá alocar os *bytes* que estão a ser recebidos vindos do cliente. O servidor sempre que recebe um pacote ordenado vindo do cliente através do método criado *PacoteUDP.gerarPacoteACK* envia o ACK correspondente (nº SEQ+1000) e guarda também numa variável global de forma a saber qual o próximo pacote que deve receber e também qual o último que recebeu.

Por fim, sempre que acontece um erro ou duplicação no recebimento dos pacotes, volta-se a gerar um pacote ACK com o último nº de sequência de pacote que tínhamos recebido como forma de requisitar uma retransmissão do mesmo pelo Cliente.

É no Cliente que está a maioria do trabalho que é efetuado na nossa aplicação. Assumindo o PDU falado anteriormente foi pensada numa implementação protocolo *pipelined*, mais concretamente, uma janela deslizante com base em *Go-Back-N*. A conceção para tal foi a criação de uma *ArrayList* de 5 pacotes, que funcionaria como a nossa janela deslizante que tem como *buffer* 5 pacotes de dados. Dessa forma pode enviar no máximo estes cinco pacotes sem esperar por o ACK de cada um sequencialmente. Tal como o *Go-Back-N ARQ*, o nosso cliente tem um temporizador de 5 segundos para a base da janela, ou seja, o primeiro pacote enviado que ainda não tenha recebido o seu *acknowledgement*. Com todos estes algoritmos pensados, a implementação em Java passou-se por usar concorrência através da extensão da classe *Thread* e também a utilização da classe *Semaphore* para filtrar o acesso concorrente das várias *threads* de entrada e saída, quando existe duplicação e/ou perdas de pacotes na comunicação.

Quando o Cliente começa, abre os seus *sockets* de entrada e saída e corre as *threads* correspondentes também. Na *thread* de saída existe uma verificação inicial em que se compara o número de sequência e se vê se é menor que a *base da janela + 5000*, de forma a certificar se o pacote a ser enviado pertence à janela/*buffer* corrente e inicia-se o temporizador para o ACK do mesmo. Por conseguinte, sempre que existam dados do ficheiro ainda por enviar, faz-se a leitura utilizando a ajuda da classe *File* e *FileInputStream* que nos ajudaram a dividir o ficheiro nos vários pacotes de 1000 *bytes* de dados, e por fim colocar estes pacotes pré-feitos no *ArrayList* (janela deslizante de 5 pacotes). Por fim, quando o *stream* do ficheiro der -1, ou seja, não existem mais dados para enviar, o cliente envia um último pacote vazio, ou seja, apenas com o cabeçalho preenchido, mas com um *array* de dados vazio. Dessa forma, o servidor irá conseguir fazer o ACK final que foi supracitado.

De forma paralela a *thread* de entrada, responsável por receber os pacotes de *acknowledgement* vindos do servidor. No seu comportamento normal, acontece o recebimento do pacote, no qual se extrai o cabeçalho, ou seja, o nº de ACK. Com este número incrementa-se o nº do pacote que estava na base da janela deslizante dado que já se teve a confirmação e assim a janela pode "deslizar". Quando o nº de ACK recebido é o mesmo que a base da janela, assumiu-se que o pacote foi perdido ou duplicado na transmissão e dessa forma, volta-se a assumir como a base da janela o último nº de sequência enviado, ou seja, volta-se a tentar enviar a "janela deslizante" toda, seguindo a especificação do *Go-Back-N*.

Por fim, quando se recebe um pacote com o nº de ACK igual a -5, infere-se que o ficheiro foi enviado por completo e assim se termina a transferência de forma fiável, correta e capaz de suportar os possíveis e frequentes erros inerentes nas camadas mais baixas da rede.

4. Testes e Resultados

Usando o CORE como teste de referência, foi colocado na máquina virtual a correr a aplicação TransfereCC em duas máquinas da topologia de Comunicações por Computadores 2019. Utilizando o *Cliente1* como Cliente e o *Alfa* como servidor, foi possível testar com os erros, perdas e duplicações que a rede provoca na ligação e notar que a aplicação resolve estes problemas, lidando corretamente com a janela deslizante, os vários envios e a espera dos ACK correspondentes.

```
Servidor: Numero de sequencia recebido 165000
Servidor: ACK enviado 166000
Servidor: Numero de sequencia recebido 167000
Servidor: Ack duplicado enviado 165000
Servidor: Numero de sequencia recebido 168000
Servidor: Ack duplicado enviado 165000
Servidor: Numero de sequencia recebido 169000
Servidor: Ack duplicado enviado 165000
Servidor: Numero de sequencia recebido 166000
Servidor: ACK enviado 167000
Servidor: Numero de sequencia recebido 166000
Servidor: Ack duplicado enviado 166000
Servidor: Numero de sequencia recebido 167000
Servidor: ACK enviado 168000
Servidor: Numero de sequencia recebido 168000
Servidor: ACK enviado 169000
Servidor: Numero de sequencia recebido 169000
Servidor: ACK enviado 170000
Servidor: Numero de sequencia recebido 170000
Servidor: ACK final enviado -5
Servidor: Todos pacotes foram recebidos! Ficheiro recebido e criado!
Servidor: Socket de entrada fechado!
Servidor: Socket de saida fechado!
```

Figura 1 - Resultados da linha de comandos do Servidor

```
Cliente: ACK RECEBIDO 166000
Cliente: ACK RECEBIDO 166000
ACK duplicado.
Cliente: Numero de sequencia enviado 167000
Cliente: Numero de sequencia enviado 168000
Cliente: ACK RECEBIDO 165000
Cliente: Numero de sequencia enviado 169000
Cliente: ACK RECEBIDO 165000
ACK duplicado.
Cliente: ACK RECEBIDO 165000
ACK duplicado.
Cliente: Numero de sequencia enviado 166000
Cliente: ACK RECEBIDO 165000
ACK duplicado.
Cliente: Numero de sequencia enviado 166000
Cliente: ACK RECEBIDO 167000
Cliente: Numero de sequencia enviado 167000
Cliente: ACK RECEBIDO 166000
```

Figura 2 - Resultados da linha de comandos do Cliente

5. Conclusões

Este projeto tornou-se imperativo para pôr em prática os conhecimentos dados e lecionados na Unidade Curricular de Comunicação por Computador e explorar as maneiras próprias para tentar construir um sistema fiável por cima dum protocolo que não é orientado à conexão e tentar desta forma, colocar uma ordem e confiança sobre o mesmo.

Com o acumular de teoria explicando a complexidade do protocolo TCP foi nos possível criar uma aplicação com o essencial e de forma a aproveitar a velocidade e simplicidade dos datagramas por protocolo UDP.

Assim e concluindo o relatório para o segundo trabalho prático, houve a concretização das funcionalidades-chave pedidas pelo enunciado mas também uma liberdade para conseguir a melhor maneira de criar e modularizar uma boa aplicação Java capaz de satisfazer os nossos pedidos.