Trabalho Prático Nº2 – FolderFastSync

Alexandre Soares[a93267], Pedro Sousa[a93225], and Simão Cunha[a93262]

Universidade do Minho - Campus de Gualtar, R. da Universidade, 4710-057 Braga
Portugal

Comunicações por Computador (2021/2022) - PL1 - Grupo 6

Resumo O segundo trabalho prático da UC de Comunicações por Computador tem como objetivo principal fazer um programa que consiga sincronizar pastas sem necessitar de servidores ou conexão à internet. A sincronização das pastas deve ocorrer utilizando o protocolo `UDP` e a monitorização do sistema deve ser em `HTTP` usando o protocolo `TCP`.

Keywords: Sincronização · UDP · TCP · HTTP · Socket

1 Introdução

Nos dias que correm, a maioria dos utilizadores usa no diariamente um ou mais serviços de armazenamento remotos, quer por razões de segurança, quer por facilidade em ter os seus ficheiros disponíveis em mais que um computador devidamente sincronizados e atualizados.

Quando se trabalha em equipa, a necessidade de usar pastas partilhadas é também grande, sendo um dos serviços mais básicos e indispensáveis nos projetos e no trabalho em equipa. Embora existam já existam algumas soluções, foi-nos proposto para desenvolvermos um novo serviço de sincronização de pastas espontâneo e ultrarápido, que se baseie quase exclusivamente em protocolos de transporte não orientado à conexão, como por exemplo o `UDP`.

Neste relatório, iremos abordar os seguintes tópicos:

- Arquitetura da solução
- Implementação da solução
- Testes e resultados
- Conclusão e trabalho futuro

2 Arquitetura da solução

A aplicação que foi desenvolvida em `Java` verifica, em primeiro lugar, se os argumentos são válidos. Se forem, irá executar quatro *threads*: o `sender`, o `reciver` , o `logs` e o `httpServer`.

A *thread* responsável pelo `sender` vai enviar as mensagens e os ficheiros para o parceiro de sincronização. A *thread* que trata o `reciver` vai interpretar

as mensagens e escrever os ficheiros recebidos. A *thread* encarregue pelos `logs` vai escrevendo para o ficheiro *"logs.txt"* na pasta *"Logs"* os eventos principais da sincronização de pastas. A *thread* `httpServer` é responsável por devolver o estado do programa através de pedidos `HTTP GET`, isto é, mostra um dos estados abaixo listados, o `host`, o `peer` e a pasta a sincronizar.

As classes mais importantes para além das já referidas são `DirController` e `DirWatcher`. A classe `DirController` vai controlar o estado da aplicação que pode ser um dos seguintes:

1. PREAUTENTICATION: representa o estado antes da autenticação dos intervenientes e tenta descobrir se está alguém do outro lado;
2. SENDINGANSWER: mostra o estado em que está a ser envida a resposta à questão feita no estado anterior;
3. SENDINGCODE: mostra o estado em que está a ser enviado o código secreto definido pelos intervenientes na conexão;
4. VALIDATECODE: corresponde ao estado onde se confirma o segredo partilhado;
5. PRESYNC: representa o estado imediatamente a seguir à autenticação e antes de as pastas estarem sincronizadas;
6. TESTTIME: é o estado que envia a resposta do teste da verificação dos tempos;
7. SYNC: indica que as pastas estão sincronizadas;
8. SENDINGAV: representa o estado em vão ser enviados todos os avisos sobre os ficheiros que vão ser alterados;
9. SENDINGFL: mostra o estado em que estão a ser enviados pacotes com a informação escrita nos ficheiros;
10. RECIVINGAV: corresponde ao estado em que estão a ser recebidos os avisos dos ficheiros que vão ser enviados para esta máquina;
11. RECIVINGFL: indica que estão a ser recebidos os pacotes com a informação dos ficheiros que vão ser alterados;
12. ERRORAV: representa o estado em que houve uma perda de pacote de aviso e vai ser feito um pedido de reenvio do aviso em falta;
13. ERRORFL: mostra o estado em que foi perdido um pacote com a informação do ficheiro que vai ser alterado e o aviso feito para este ser reenviado;
14. WAITOKAV: corresponde ao estado posterior ao SENDINGAV onde a máquina fica à espera de confirmação da receção de todos os pacotes de aviso enviados;
15. WAITOKFL: é análogo ao anterior, mas relativo ao estado SENDINGFL e aos seus pacotes de informação de ficheiro.
16. OKAV: indica que todos os pacotes de aviso foram recebidos com sucesso;
17. OKFL: mostra que todos os pacotes de informação foram recebidos com êxito.

Para além disso, esta classe vai ter os métodos que as threads `reciver` e `sender` vão usar. A classe `DirWatcher` vai ficar responsável por observar as pastas a sincronizar e reportar qualquer alteração que aconteça nas mesmas ao `DirController`.

3 Especificação do protocolo

3.1 Formato das mensagens protocolares

A mensagens que vamos enviar têm um total de 5 KB(5120 bytes), onde os primeiros 4 bytes representam o *header* da mensagem que estão a ser escritos em `UTF-16`, logo vão ser necessárias 2 letras para os representar.

Vamos, de seguida, descrever todos os *headers* que podemos ter nas nossas mensagens:

1. AU: *header* que representa as mensagens de autenticação que possui um parâmetro extra(2bytes) para especificar se a mensagem é uma pergunta(Q), uma resposta(A), uma confirmação(C) ou uma negação(N). O corpo desta mensagem vai estar vazio.
2. PS: *header* que representa a mensagem que envia a palavra-passe no seu corpo para ser confirmada pela máquina que estamos conectados.
3. PR : *header* que mostra que a aplicação está num estado pré-sincronização e estas vão ser as mensagens trocadas para alcançar o estado *SYNC*.
4. AV: *header* que corresponde aos avisos e possui dois parâmetros adicionais, correspondentes ao *ID* do ficheiro atribuido por nós para sabermos diferenciar os ficheiros e ao total de partes que vão ser enviadas para construir o mesmo. No corpo da mensagem vai estar escrito o nome do ficheiro.
5. FL: *header* que corresponde à informação do ficheiro e possui dois parâmetros: *ID* do ficheiro e a parte do total do mesmo que estamos a enviar. No corpo de mensagem vai estar a informação contida nesse ficheiro.
6. ER: *header* que corresponde a uma mensagem de erro. Este tipo de mensagem tem dois parâmetros adicionais. Se se tratar deu um erro de uma mensagem de aviso, 4 bytes vão dizer "AV"e os próximos 4 vão ter o *ID* do ficheiro. Já se for um erro de uma mensagem do tipo FL, esta vai ter três parâmetros onde o primeiro corresponde a FL, o segundo ao *ID* do ficheiro e o terceiro à parte que foi perdida. O corpo da mensagem vai estar vazio em ambos os casos.
7. OK: *header* das mensagens de confirmação. Este *header* vai ter um parâmetro extra que vai indicar o tipo de confirmação, ou seja, AV ou FL. O tipo AV tem ainda outro campo que contém o total de mensagens do tipo recebidas para certificar que não foi perdida nenhuma. O corpo da mensagem vai estar vazio.

3.2 Interações

Este é o exemplo de uma transferência de ficheiros em que não houve nenhuma perda de pacotes. Primeiro são enviados os avisos que têm o nome do ficheiro e depois de receber a confirmação de envio são enviadas sequencialmente as partes dos mesmos e no final é enviada a confirmação por parte da *thread* a receber.

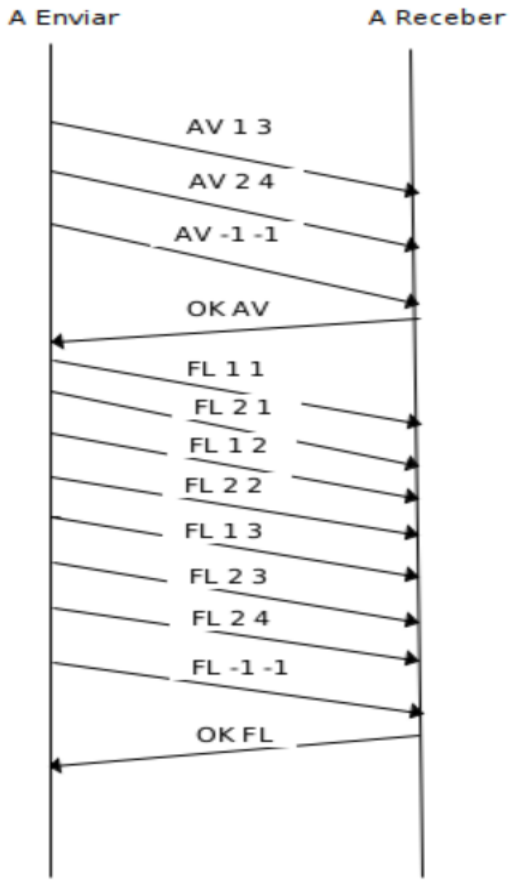

Figura 1. Tranferência de dois ficheiros sem pacotes perdidos

Este é o exemplo de uma transferência de ficheiros em que houve perda de pacotes. O método é análogo ao anterior até ao ponto em que se perde um pacote. Quando este é perdido volta-se atrás no envio dos pacotes até aquele que foi perdido e são reenviados esses pacotes. O mecanismo de perda funcionaria da mesma maneira se tivesse sido perdida uma mensagem de AV.

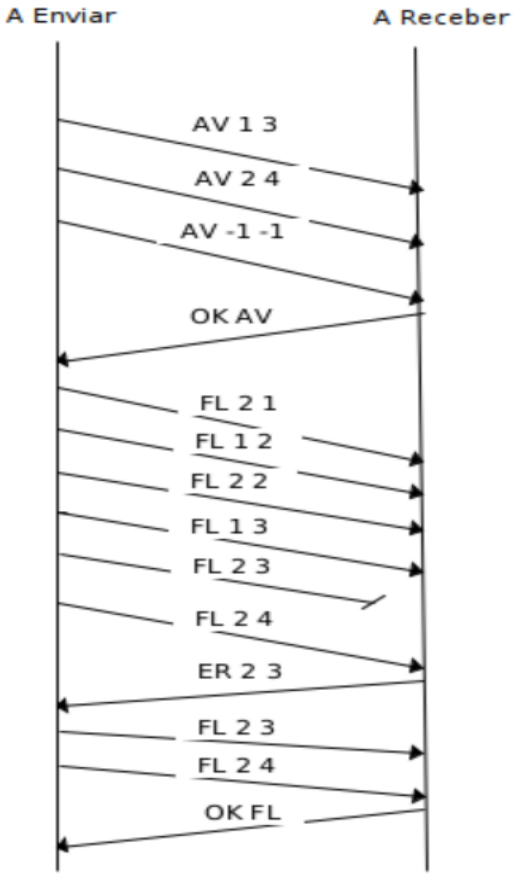

Figura 2. Tranferência de dois ficheiros com pacotes perdidos

4 Implementação

4.1 Detalhes

DirWatcher

A *thread* que controla a classe `DirWatcher` avisa o controlador das alterações que foram feitas à diretoria enviando-lhe uma lista com o nome dos ficheiros que foram alterados. Estas mudanças podem ser adicionar, remover e modificar.

Em termos de código, esta classe é muito simples, uma vez que usamos a biblioteca de funções chamada `java.nio.file` que tem o método *take* que retorna quais os ficheiros que foram alterados e qual foi o tipo de alteração que foi feita.

Por isso, bastou colocar a *thread* num *loop* infinito com uma espera que garante que o `Watcher` só vai procurar alterações na diretoria da pasta quando esta estiver sincronizada com a outra.

DirController

A classe `DirController` vai ter os métodos de mudança de estado, gestão de erros e dos ficheiros a transferir, leitura e escrita de ficheiros e interpretação de mensagens. O envio das mesmas é feito pela *thread* da classe `Sender` e a receção das mesmas é feita pela *thread* da classe `Reciver`. Vamos explicar os métodos referentes a cada um dos tópicos previamente enunciados:

1. `Mudança de estado`: Não temos métodos especifícos que provoquem a mudança do estado do programa. Este vai sendo alterado sempre que forem verificadas certas condições nos métodos desta classe.
2. `Gestão de erros`: A deteção de erros baseia-se em receber pacotes fora de ordem ou no caso do último pacote quando é enviada uma mensagem do tipo AV ou FL com os dois parâmetros com valor -1 e o último pacote não foi ainda recebido, ou seja, o sender envia a mensagem de confirmação de receção dos pacotes todos e o reciver ainda não recebeu o pacote final. Sempre que detetamos que houve uma perda de pacote esta é comunicada ao sender pelo reciver que se encarrega de reenviar o pacote.
3. `Gestão de ficheiros a transferir`: Depois de receber a lista de ficheiros a transferir vamos guardar num map o seu identificador e criamos um objeto da classe FileStat que contém informação sobre o ficheiro como o seu nome, o número de partes recebidas/enviadas(depende do lado da sincronização) e têm ainda uma input e output stream para ler e escrever para o ficheiro, respetivamente. Para enviar os pacotes usamos o método *round robin* passando uma vez por cada ficheiro. Os motivos que temos para usar este método são a facilidade de implementação e a possiblidade de melhoria futura para *weighted round robin*.
4. `Leitura e escrita de ficheiros`: Os métodos que fazem a leitura e a escrita dos ficheiros estão na classe FileStat e o controller limita-se a invocá-los. Mas, estas escritas e leituras são feitas usando input e output streams. Do lado do sender são lidos, normalmente, 5108 bytes(5120 é o tamanho total da mensagem menos 12 bytes do header) do ficheiro, o caso que são lidos menos é quando estamos a ler a última e está tem menos de 5108 bytes. Do lado do reciver, são lidos os bytes da mensagem e escritos no ficheiro através de uma output stream.
5. `Interpretação das mensagens`: Para interpretar as mensagens verificamos o campo do *header* e usamos o método correspondente que foi feito para tratar desse *header*.
6. `Controlo de fluxo/congestão`: O controller tambem faz um pequeno controlo de fluxo/congestão detetando que o numero de erros passou de um determinado valor, e caso aconteça, obriga o envio das menssagens a esperar um tempo entro o envio.

Sender e Reciver

O sender e o reciver são as classes que tratam do envio e receção de pacotes,respetivamente. Estes ainda dão timeout sempre que detectam que o outro lado tem problemas de conexão ou foi desligado.

Logs

A *thread* que controla a classe `Logs` está encarregue de escrever para um ficheiro toda a informação relevante sobre a sincronização das pastas. Em particular, a nossa classe `Logs` vai escrever para o ficheiro `logs.txt` qual vai ser o caminho da pasta e o IP do par da sincronização.

Quando houver uma mudança na diretoria vai ser escrito o estado RECIVINGAV ou SENDINGAV dependendo do lado da sincronização em que estamos e de seguida o estado RECIVINGFL ou SENDINGFL. No caso do primeiro estado vai escrever que ficheiros está a receber no caso do último estado vai ser escrito ainda o tamanho dos ficheiros, quanto tempo demorou a transferência e a sua velocidade em bits por segundo.

Em termos de código, esta classe é parecida com a `DirWatcher`, porque vai ter um *loop* infinito que vai ficar à espera que o estado seja diferente de SYNC e quando for um dos quatro referidos anteriormente vai escrever no ficheiro `logs.txt` a informação respetiva.

HTTPServidor

O nosso servidor `HTTP` vai ser executado numa *thread* do programa principal. Esta classe vai conter a classe `DirController` e o IP do *peer* de forma a poderem ser colocadas no ficheiro `HTML` e vai atender pedidos `HTTP GET` na porta 80. Este servidor possui invoca uma classe de *handler* (`MyHttpHandler`) que é responsável por enviar através de um `ObjectOutput` o conteúdo do ficheiro `HTML` que inclui o estado da aplicação, a pasta a sincronizar, o *host* e o seu *peer*. Além disso, é enviado o código 200 para indicar que o pedido foi bem sucedido.

4.2 Bibliotecas de funções

- **java.net.DatagramSocket**: Canais UDP e TCP
- **java.net.InetAddress**: Resolução de nomes e *IP*'s
- **java.io.***: Serialização dos dados
- **java.nio.file.***: Manipulação de ficheiros
- **com.sun.net.httpserver.***: Servidor HTTP
- **java.util.concurrent.locks.***: Controlo de concorrência

5 Testes e resultados

Para os cenários de teste, a equipa docente disponibilizou 3 ficheiros indicados abaixo:

– **tp2-folder1**: pasta com apenas um ficheiro de texto de 232KB, disponível em [1].

– **tp2-folder2**: pasta com 3 ficheiros de diferentes tipos, que ocupa aproximadamente 6,2MB, disponível em [2].

– **tp2-folder3**: pasta com 6 ficheiros de diferentes tipos, alguns iguais aos da pasta tp2-folder2 e que ocupa 38MB, disponível em [3].

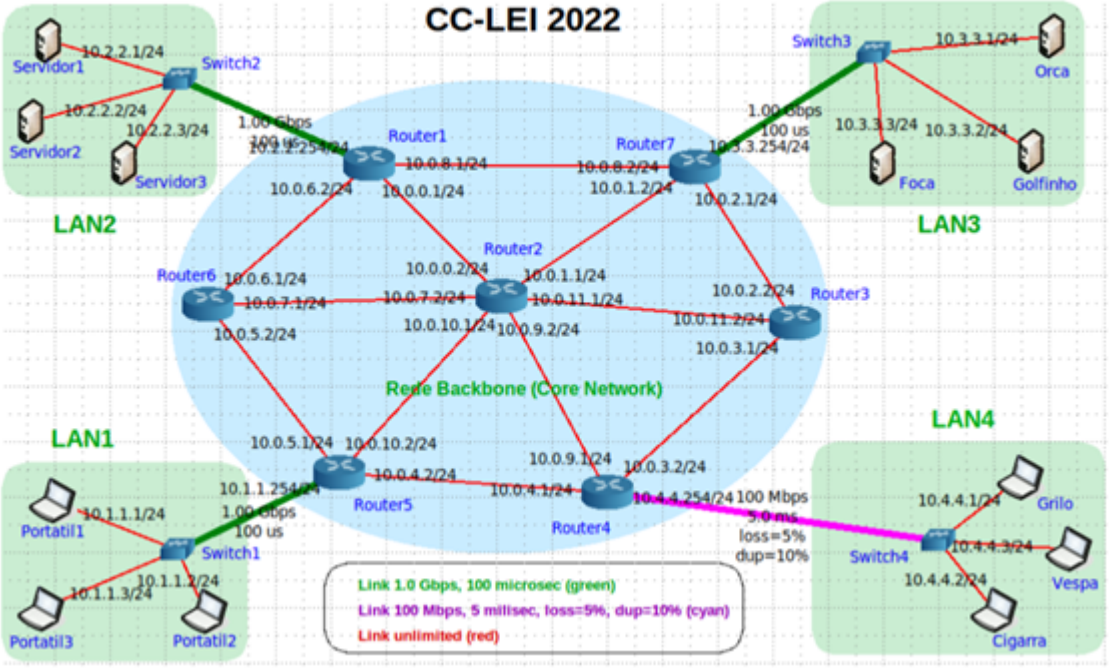

Figura 3. Topologia da rede da UC de Comunicações por Computador

[1] `wget http://marco.uminho.pt/disciplinas/CC-LEI/tp2-folder1.zip`
[2] `wget http://marco.uminho.pt/disciplinas/CC-LEI/tp2-folder2.zip`
[3] `wget http://marco.uminho.pt/disciplinas/CC-LEI/tp2-folder3.zip`

5.1 Cenário #1: Sincronizar a pasta *tp2-folder1* do *Servidor1* com uma pasta vazia *orca1* no servidor *Orca*

5.2 Cenário #2: Sincronizar a pasta *tp2-folder1* colocada no *Servidor1* com a pasta vazia *grilo1* no portátil *Grilo*

5.3 Cenário #3: Sincronizar a pasta *tp2-folder2* do *Servidor1* e a pasta vazia *orca2* no servidor *Orca*

5.4 Cenário #4: Sincronizar a pasta *tp2-folder2* colocada no *Servidor1* e com a pasta *tp2-folder3* colocada no servidor *Orca*

Cenários	Tempo de transferência(segundos)	Débito final (bits por segundo)
1	1.489	1262877.1
2	1.482	1268842.1
3	5.992	1069153.4
4		

Tabela 1. Registo de algumas estatísticas dos cenários de teste

Como a nossa implementação do PreSync assume que ambas as pastas vão ficar com os ficheiros da pasta que entrou em sincronização a mais tempo não conseguimos testar o cenário 4. Se as alterações na pasta tivessem ocorrido no estado de Sync já daria para fazer este tipo de sincronização.

6 Conclusões e trabalho futuro

Neste trabalho prático, abordámos a temática da sincronização rápida de pastas sem necessitar de servidores nem de internet com recurso ao protocolo UDP e da monitorização simples em HTTP sobre TCP.

Cumprimos todos os objetivos mínimos propostos pela equipa docente com êxito, mas assumimos que o estado de PreSync ia tranferir o conteúdo da pasta que estava à mais tempo na conexão para a mais nova o que, provavelmente, não é o melhor método.

Como sugestões para trabalho futuro, poderá ser interessante implementar os requisitos opcionais do enunciado do trabalho prático. Além disso também seria possível otimizar a nossa implementação com as seguintes mudanças: implementação de um round-robin pessado, que permitiria que os ficheiros mais pessados pudessem ter um maior prioridade na transferência de ficheiros, o que levaria a a tempos de tranferência menores, a melhoria do controlo de congestão/fluxo, podendo mudar dinamicamente o tempo de espera entre envios,

interpertando os erros e calculando o número de pacotes enviados sem dar erro, levando a esperas menores e a um tempo de tranferência menor.

Podia-se também melhorar o estado de PreSync, permitindo a troca de determinados ficheiros, evitando assim enviar a tranferência de pastas inteiras e evitar limpar a pasta que vai receber.