Universidade do Minho

Mestrado Integrado em Engenharia Informática
Licenciatura em Engenharia Informática

Comunicações por Computador

TP2 - FolderFastSync
Sincronização rápida de pastas em ambientes *serverless*

Grupo 7.6

Ana Canelas
(A93872)

Ana Henriques
(A93268)

Ana Murta
(A93284)

27 de janeiro de 2022

Conteúdo

1 Introdução

No âmbito da unidade curricular *Comunicações por Computador*, foi realizado este trabalho prático que consiste na implementação de uma aplicação de sincronização rápida de pastas sem necessitar de servidores nem de conectividade à Internet, designada por *Folder-FastSync*. Esta opera com dois protocolos: um de monitorização simples em HTTP sobre TCP e outro para a sincronização de ficheiros sobre UDP.

Esta aplicação deve ter algumas funcionalidades, tais como:

- O sist deve conseguir responder a pedidos HTTP GET devolvendo, no mínimo, o seu estado de funcionamento.
- Ser possível atender múltiplos pedidos em simultâneo nos dois sentidos.
- O sistema deve ser eficaz, ou seja, no final de uma operação de sincronização entre dois sistemas, as pastas de um e de outro tem de ser iguais.
- O sistema de ter um sistema de registo (logs) que vá indicando as operações que estão a ser executadas.
- O sistema deve definir e obter duas métricas de eficiência: tempo de transferência e débito final em bits por segundo, sendo que os valores devem ser registados no log.
- Autenticação dos parâmetros e verificação das condições de funcionamento, nomeadamente se possui acesso à pasta e à rede.

2 Arquitetura da solução

Para a resolução do problema apresentado foi necessário logo à partida estabelecer uma arquitetura adequada que conseguisse atender a todos os requisitos impostos. Deste modo, segue-se a arquitetura final implementada:

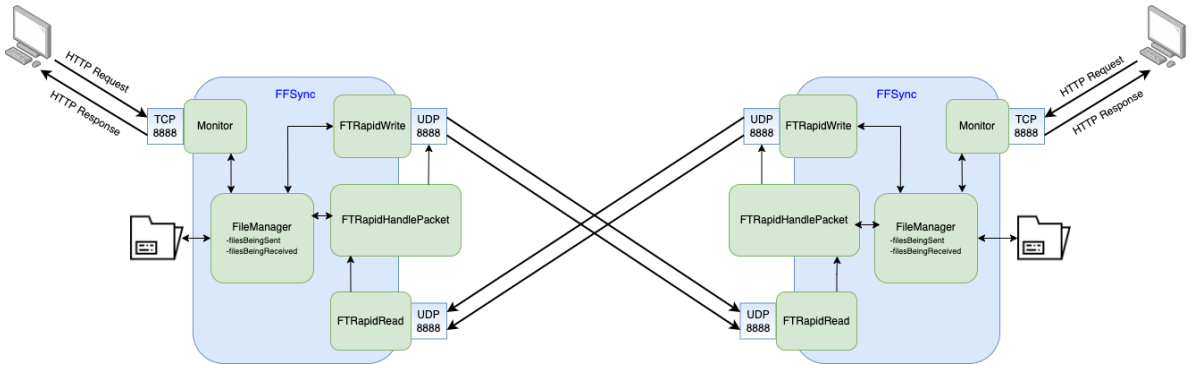

Figura 1: Arquitetura da Solução

Como podemos observar pelo diagrama, existem 5 módulos principais. Apenas um deles,o `Monitor` faz parte da monitorização e todos os outros são relativos ao protocolo FT-Rapid.

2.1 Monitorização

Para atender aos requisitos da monitorização, temos um modulo `Monitor`, que é uma Thread responsável por atender todos os pedidos `TCP` na porta 8888. Quando recebe um pedido `HTTP` no url base /, devolve uma resposta `HTTP` com o seu estado de funcionamento juntamente com uma lista de todos os ficheiros presentes na sua pasta. Para saber qual a sua lista de ficheiros terá de aceder ao módulo `FileManager`, o qual será de seguida falado.

2.2 Protocolo FTRapid

Os principais componentes do protocolo FTRapid são os módulos `FileManager` e `FTRapidHandlePacket`. O `FileManager` é responsável por todas as ações que estejam relacionadas com ficheiros. Este regista todos os ficheiros que estão a ser enviados e aqueles que estão a ser recebidos, coordenando assim o processo de sincronização. O módulo `FTRapidHandlePacket` é responsável por processar todos os pacotes recebidos e decidir qual o próximo passo a tomar (tipo de pacote a enviar, criar ficheiro, etc). É criada uma thread `FTRapidHandlePacket` por cada pacote recebido. Estas threads são geradas pelo modulo `FTRapidRead`, que é uma thread responsável por receber todos os pedidos `UDP` na porta 8888. Por último, temos o módulo `FTRapidWrite` que é responsável por periodicamente enviar um tipo de pacotes, os quais iniciam todo o processo de sincronização.

3 Especificação do Protocolo

O processo de construção deste protocolo baseou-se na tomada de decisões relativas a dois tópicos fundamentais: o formato das mensagens protocolares e as interações entre os nós, isto é, a forma como estas mensagens/pacotes são trocados.

Estes dois tópicos serão de seguida explicados, e no final serão justificadas todas as decisões tomadas.

3.1 Formato das mensagens protocolares

De forma a atender à especificidade do protocolo a implementar, tivemos de definir um formato para as mensagens/pacotes que serão trocadas entre os vários nós. Este formato é constituído por um conjunto de parâmetros.

```
PacketType type;
List<FileInfo> fileList;
List<FileInfo> requestFiles;
FileChunk fileChunk;
InetAddress endIP;
int endPort;
String secret;
```

Primeiro temos o parâmetro `PacketType` que define o tipo de pacote e consequentemente todo o resto da informação que conterá o pacote. Existem 4 tipos de pacote, cada um com uma ação/objetivo específico.

```
enum PacketType {
    FILE_LIST,
    REQUEST_FILES,
    FILE_CHUNK,
    CHUNK_ACK
}
```

Um pacote pode ser do tipo `FILE_LIST`, o qual tem como objetivo fornecer ao nó destino informação sobre o conjunto de ficheiros existentes no sistema do nó origem. Este pacote irá assim conter uma lista com a informação de cada ficheiro(`fileList`) presente no sistema do nó origem, sendo essa informação(`FileInfo`) constituída por um conjunto de parâmetros.

```
String name;
long size;
long lastModified;
```

Dentro destes encontrar-se-á o nome do ficheiro(`name`), incluído o seu formato, o tamanho do ficheiro(`size`) em bytes e, por fim, a data da última alteração(`lastModified`).

Se o tipo do pacote for `REQUEST_FILES`, significa que o nó origem pretende solicitar um ou vários ficheiros do nó destino. Assim, neste pacote será enviada uma lista(`requestFiles`) com a informação dos ficheiros a pedir ao nó destino. A informação de cada ficheiro é idêntica à anteriormente referida(`FileInfo`).

O pacote pode ainda ser do tipo `FILE_CHUNK`, o qual é utilizado quando o nó origem pretende enviar ao nó destino uma parte/chunk de um dado ficheiro. Este tipo de pacotes são enviados como resposta a um pacote do tipo `REQUEST_FILES`.

```
byte[] data;
FileInfo fileInfo;
int chunkSequenceNumber;
int numChunks;
```

Cada chunk(`FileChunk`) é constituído por um array de bytes(`data`), a informação(`fileInfo`) sobre o ficheiro ao qual esse chunk pertence, o número de sequência(`chunkSequenceNumber`) desse chunk, e, por último, o número total de chunks(`numChunks`) desse ficheiro.

Por último, um pacote pode ainda ser do tipo `CHUNK_ACK`, caso o nó origem pretenda dar confirmação de recebimento/acknowledge a um chunk recebido. Este tipo de pacote é sempre enviado depois de recebido um pacote do tipo `FILE_CHUNK`. Como informação, é enviada a mesma presente num chunk(`fileChunk`) referida anteriormente, excluindo o array de bytes.

Para além de toda esta informação específica a cada tipo de pacote, existem ainda três parâmetros que estão sempre presentes em todos os pacotes. Sendo estes o endereço IP do nó destino(`endIP`), a porta do nó destino(`endPort`) e o segredo(`secret`) em comum partilhado entre os nós.

3.2 Interações do *FTRapidProtocol*

Neste secção iremos explicar como é que ocorrem todas as interações entre os nós, ou seja, qual a sequência de troca de pacotes que permite atingir um estado de sincronização de pastas entre estes.

Quando um nó é iniciado, este fica de imediato apto a receber pacotes. O processo de sincronização é de seguida iniciado: este vai buscar a lista com a informação dos ficheiros presentes na sua pasta local e envia de imediato um pacote do tipo `FILE_LIST` ao nó destino. Repetindo este comportamento periodicamente.

Quando um nó recebe um pacote do tipo `FILE_LIST`, este compara a lista de ficheiros recebida com a lista de ficheiro do seu sistema local. Caso tenha algum ficheiro em falta ou algum dos seus ficheiros tenha uma data de alteração mais antiga que a do ficheiro do nó origem, a informação desses ficheiros é guardada e envida num pacote do tipo `REQUEST_FILES`.

Caso não hajam ficheiros que respeitem essas condições, significa que a sua pasta local se encontra sincronizada e o processo de sincronização termina.

Quando um nó recebe um pacote do tipo `REQUEST_FILES`, vai responder a esse pacote enviando os ficheiros requisitados. Para cada um dos ficheiros, em paralelo, irá gerar um conjunto de chunks e enviar cada um através de um pacote `FILE_CHUNK`. Regista localmente os ficheiros que esta a enviar, juntamente com os chunks gerados.

Quando um nó recebe um pacote do tipo `FILE_CHUNK`, guarda o conteúdo do chunk recebido e envia de resposta um pacote do tipo `CHUNK_ACK`. Quando recebe todos os chunks de um dado ficheiro, escreve esse ficheiro para o sistema e, se não tiver mais ficheiros para receber ou enviar, termina o processo de sincronização.

Quando um nó recebe um pacote do tipo `CHUNK_ACK`, regista o recebimento do ack no chunk correspondente. Até receber todos os acks relativos a um dado ficheiro que esta a enviar, vai reenviando os chunks para os quais não recebeu ack, tendo em conta um timeout definido. Quando recebe todos os acks relativos a um dado ficheiro, regista o envio do ficheiro e, se não tiver mais ficheiros para enviar ou receber, termina o processo de sincronização.

No diagrama seguinte temos um exemplo do processo de sincronização entre o computador 1 e o computador 2. Em que o computador 1 tem o ficheiro `file1.txt` na sua pasta e o computador 2 tem a pasta vazia.

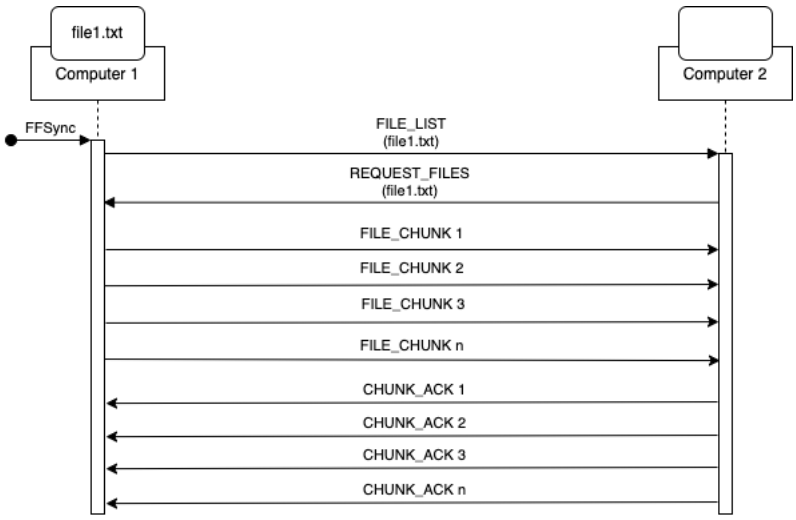

Figura 2: Diagrama de sequência do protocolo de sincronização FTRapid.

Neste diagrama podemos observar mais facilmente a sequência de pacotes anteriormente explicada. Caso algum pacote `FILE_CHUNK` fosse perdido, o `CHUNK_ACK` correspondente não chegaria ao computador 1, sendo o chunk reenviado depois de um timeout.

É de notar que a sequência dos pedidos também acontece no sentido contrário, isto é, o computador 2 também envia o pacote `FILE_LIST` ao computador 1. Nesse caso, como o computador 1 não tem nenhum ficheiro para receber, o processo terminaria por aí.

3.3 Decisões tomadas

A tomada de todas estas decisões foi feita tendo por base o facto de estarmos a trabalhar com `UDP`. `UDP` é um protocolo orientado a datagramas que não fornece confiabilidade da transmissão, e, como tal, apresenta três caraterísticas importantes. Não garante a recepção das mensagens (pode haver a perda de pacotes), não garante a ordem de transmissão (os pacotes podem ser recebidos numa diferente ordem da enviada) e pode haver a duplicação de pacotes.

Relativamente à primeira caraterística, que é aquela que nos pode trazer mais problemas, decidimos logo no início que uma boa estratégia para combater este problema seria tentar diminuir ao máximo o número de pacotes trocados, de forma a diminuir o número de pacotes possivelmente perdidos e consequentemente diminuir a probabilidade de haver problemas durante o processo de sincronização. Assim, decidimos que o processo de obtenção dos ficheiros do nó vizinho seria composto por apenas um pedido, o pacote `FILE_LIST`. Cada nó fica responsável por enviar periodicamente ao outro qual a sua lista de ficheiros, sem que esta lhe seja pedida. Assim, em vez de haver duas trocas (a requisição da lista e depois envio da lista) há apenas uma. A requisição dos ficheiros em falta também é feita apenas com um pedido, o pacote `REQUEST_FILES`. No caso de algum destes dois pacotes não chegar ao destino, como o processo de sincronização ainda se encontra no início, o problema não é grave, pois, como dito anteriormente, os pacotes do tipo `FILE_LIST` são enviados periodicamente por cada nó.

Relativamente ao envio dos ficheiros, devido ao seu tamanho, estes têm de ser enviados por partes/chunks em pacotes `FILE_CHUNK`, não havendo forma de reduzir o número de pacotes para além de aumentar o tamanho de cada um deles. Caso algum destes pacotes seja perdido, a situação é diferente da anterior, uma vez que já estamos a meio do processo de transferência, o problema já seria mais grave. Por esta razão decidimos adicionar outro tipo de pacote, `CHUNK_ACK`, que informa o nó origem que o chunk foi recebido. Assim, o processo de transferência não precisa de ser terminado a meio, e os chunks continuam a ser reenviados até todos terem chegado ao nó destino.

Relativamente às duas outras caraterísticas do protocolo UDP, isto é, a ordem de transmissão e a duplicação de pacotes, estas também se podem tornar num problema principalmente no que diz respeito aos pacotes `FILE_CHUNK`. Porém, estes foram combatidos através do uso do número de sequência de cada chunk presente no pacote, referido anteriormente, e ao nível da implementação com a persistência de um estado que nos permite saber quais os pacotes recebidos.

4 Implementação

Para a implementação da arquitetura proposta decidimos utilizar a linguagem de programação Java por já estarmos familiarizados com esta. Utilizamos duas livrarias importantes, `Java.net` para lidar com conexões `TCP` e `UDP` e `Java.io` para lidar com ficheiros, streams de dados e serialização de dados.

Olhando para o diagrama da arquitetura, observamos que temos 5 módulos, os quais se reflectem em 5 classes `Java`. De seguida iremos falar um pouco sobre elas e as restantes classes implementadas.

4.1 Parâmetros

O programa inicia-se na classe `FFSync`, onde inicialmente é feita uma validação dos parâmetros recebidos. O primeiro parâmetro tem de conter uma directoria válida da pasta a sincronizar. Caso a directoria seja válida mas não exista, esta será criada. O segundo parâmetro recebido será um IP válido do nó parceiro, que esteja apto a receber pedidos(fizemos uso do `InetAdress` para essa verificação). Apenas um IP será aceite, visto que a aplicação não foi desenvolvida para mais do que um nó parceiro. Por último, ainda é possível fornecer um segredo (string) partilhado.

De seguida, é gerada uma instância `FileManager` e são de imediato iniciadas 3 threads (`Monitor, FTRapidRead e FTRapidWrite`).

4.2 FileManager

`FileManager` é a classe que guarda todo o estado da aplicação.

```
private final File folder;
private final Map<String, Map<Integer,FileChunk>> filesBeingReceived;
private final Map<String, Map<Integer,FileChunk>> filesBeingSent;
private final String secret;
```

Tem acesso direto à pasta de sincronização (folder), tem dois `Map` para guardar as partes dos ficheiros que estão a ser recebidos assim como dos ficheiros que estão a ser enviados. Ambos estão indexados pelo nome do ficheiro, e o valor corresponde a outro `Map` indexado pelo número de sequência do chunk para a instância do `FileChunk` correspondente.

É aqui que ocorre a divisão dos ficheiros em partes/chunks. Junto com cada chunk é registado o seu número de sequência, o número total de chunks e o timestamp de criação(igual para todos os chunks do mesmo ficheiro). O tamanho de cada chunk é sempre o mesmo (1024 bytes) independentemente do tamanho do ficheiro, visto que não tivemos tempo suficiente para pensar numa estratégia que possibilitasse a chegada de vários ficheiros ao mesmo tempo.

Quando um ficheiro é enviado, é guardado em `filesBeingSent`, e sempre que é recebido um `CHUNK_ACK`, é registado na instância `FileChunk` correspondente.

Quando um ficheiro esta a ser recebido, cada chunk que chega é guardado em `filesBeingReceived`.

Quando o último chunk de um dado ficheiro é recebido, é calculado o tempo de descarga e o debito para esse ficheiro. O tempo é calculado fazendo a diferença entre o timestamp que esta registado num chunk(o timestamp é o mesmo em todos os chunks uma vez que são gerados ao mesmo tempo) com o timestamp atual. O debito é calculado dividindo o tamanho do ficheiro em bits pelo tempo calculado em segundos.

Assim, quando todos os chunks são recebidos, o `Map<Integer,FileChunk>` é ordenado pela key (número de sequência do chunk) e por essa ordem os chunks são escritos para disco criando/alterando o ficheiro.

O `secret` é o segredo partilhado que foi recebido (ou não) como parâmetro.

4.3 Monitor

A classe `Monitor` é uma thread responsável por atender pedidos `TCP` na porta 8888. Se o pedido recebido é um pedido `HTTP GET` em /, devolve uma simples página `HTML` com o estado da aplicação e uma tabela com informação dos ficheiros existentes.

4.4 FTRapidRead e FTRapidWrite

`FTRapidRead` é uma thread responsável por receber todos os pedidos `UDP` na porta 8888, desserializar os dados recebidos e iniciar uma thread `FTRapidHandlePacket`. `FTRapidWrite` é responsável por enviar pacotes do tipo `FILE_LIST` a cada 2 segundos.

4.5 FTRapidHandlePacket

Tal como referido, `FTRapidHandlePacket` é uma thread gerada para cada pacote recebido. Inicialmente é feita uma validação do segredo partilhado presente no pacote. Caso o segredo não seja igual, o pacote não é processado. Dependendo do tipo de pacote, uma ação será realizada, interagindo sempre com o `FileManager`.

Por exemplo, quando um pacote é do tipo `REQUEST_FILES`, para cada ficheiro é criada uma thread na qual são gerados os chunks correspondentes e de seguida enviados. Cada uma destas thread só termina quando tiverem sido recebidos acks para todos os chunks, até lá esta vai reenviando os chunks sem ack, segundo um timeout de 1 segundo.

4.6 FTRapidPacket

`FTRapidRead` é a classe que implementa os pacotes que serão transmitido. Todos os seus parâmetros foram anteriormente referidos. `FileInfo` e `FileChunk` são classes que poderão fazer parte dos pacotes, consoante o seu tipo.

4.7 Logs

Os logs da aplicação são registados num ficheiro `logs.txt`, o qual fica guardado na pasta (este ficheiro não é contabilizado para a sincronização).

São registados vários níveis de logging. Ao nível de pacotes, sempre que um pacote é recebido ou enviado é registado no log. São registados vários acções, como comparação das listas de ficheiros, divisão dos ficheiros em chunks, etc. Também é registado, tanto para o ficheiro log como para a consola, o envio completo de um ficheiro e o recebimento de um ficheiro, com o respetivo tempo de descarga e débito.

5 Testes e Resultados

Para testar a implementação do protocolo decidimos seguir o cenário de testes descrito no enunciado. No final, para verificar a sincronização das pastas utilizamos o comando `diff`, tendo em atenção em não incluir os ficheiros `logs.txt`.

5.1 Cenário 1

Depois de testado este cenário verificamos que a sincronização aconteceu como era esperado. O ficheiro `rfc7231.txt` foi transferido do `Servidor1` para `Orca`. Obtivemos um tempo de descarga de 1.7 segundos e um debito de 1091274 bits/segundo.

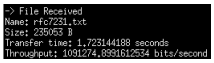

Figura 3: Resultados cenário 1.

5.2 Cenário 2

Depois de testado este cenário verificamos que a sincronização aconteceu como era esperado. O ficheiro `rfc7231.txt` foi transferido do `Servidor1` para `Grilo`. Obtivemos um tempo de descarga de 2.6 segundos e um debito de 725235 bits/segundo. Esta diminuição no débito deveu-se à perda de pacotes adicionada.

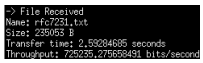

Figura 4: Resultados cenário 2.

5.3 Cenário 3

Depois de testado este cenário verificamos que a sincronização aconteceu como era esperado. Os ficheiro `topo.imn`, `rfc7231.txt` e `Chapter_3_v8.0.pptx` foram transferido do `Servidor1` para `Orca`. Obtivemos um tempo de descarga total de 27 segundos e um debito total de 1896492 bits/segundo (calculando somando os tamanhos de todos os ficheiros e dividindo pelo tempo de descarga do último ficheiro recebido).

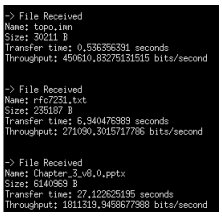

Figura 5: Resultados cenário 3.

5.4 Cenário 4

Depois de testado este cenário verificamos que a sincronização aconteceu como era esperado. Os ficheiro `topo.imn`, `rfc7231.txt` e `Chapter_3_v8.0.pptx` foram transferido do `Servidor1` para `Orca`, obtendo um tempo de descarga total de 78 segundos e um debito total de 658909 bits/segundo. Os ficheiros `CC-Topo-2022.imn`, `book.jpg`, `bootstrap-dist.zip` e `wireshark.tar.xz` foram transferidos do `Orca` para o `Servidor1`, obtendo um tempo de descarga total de 116 segundos e um debito total de 2285943 bits/segundo.

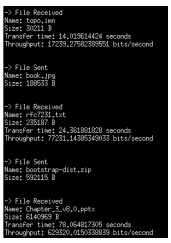

Figura 6: Resultados cenário 4 - Orca

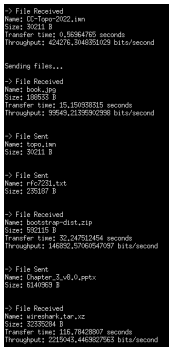

Figura 7: Resultados cenário 4 - Servidor1

6 Conclusão

Com a realização deste trabalho, podemos por em prática o conhecimento assimilado nas aulas teóricas no que diz respeito aos servidores de front e back-end. Acabámos, também, por aperfeiçoar e aprofundar outros conhecimentos e ferramentas dos quais tirámos proveito, como criar comunicações em *Java* através de *Java Sockets*.

Deste modo, a elaboração deste trabalho prático, por se mostrar diferente dos outros trabalhos práticos realizados para esta Unidade Curricular, revelou-se muito interessante, desafiando o grupo a ultrapassar as dificuldades enfrentadas ao longo da sua construção e a tomar decisões imprescindíveis para uma implementação mais eficiente. Acreditamos, ainda, que este trabalho prático nos enriqueceu com uma experiência que certamente será bastante útil para o futuro.

Um trabalho futuro seria a implementação de alguns dos requisitos opcionais, os quais não conseguimos concluir por falta de tempo.