FolderFastSync: Sincronização rápida de pastas em ambientes serverless

Gonçalo Braz[a93178], Luís Faria[a93209], and Rui Moreira[a93232]

Universidade do Minho, Braga
Departamento de Informática

Abstract. Desenvolvimento de uma aplicação de sincronização rápida de pastas sem necessitar de servidores nem de conectividade Internet, designada por FolderFastSync (FFSync)

Keywords: UDP · Protocol · Server · Client.

1 Introdução

Este trabalho foi realizado no âmbito da Unidade Curricular Comunicações por Computadores. Foi proposta a elaboração de um sistema de sincronização rápida entre pastas sem necessitar de servidores nem de conectividade Internet, chamada FolderFastSync(FFSync). Este sistema foi implementado recorrendo à linguagem de programação Java.

O protocolo desenvolvido cumpre os requisitos obrigatórios, que são:

- O sistema deve responder a pedidos HTTP GET devolvendo, no mínimo, o seu estado de funcionamento
- O sistema deve conseguir 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 devem ser exactamente 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. Os valores devem ser registados no *log*.
- Segurança: o sistema deve ter implementado um mecanismo simples autenticação mútua

Estes itens serão abordados nos capítulos seguintes bem como os requisitos extras implementados:

- Sincronização entre mais que dois *peers*

2 Arquitectura da Solução

O trabalho elaborado consistes maioritariamente por 3 pacotes principais, **Server**, **Messages** e **MetaData**, e a classe **FFSync**.

FFSync actualiza os meta dados dos ficheiros usando classes de **MetaData**, enviando de x em x tempos estes meta dados aos seus *peers* e, caso esteja desactualizado, pede também aos peers correspondentes os ficheiros mais recentes, recebendo-os através do **Server**.

Server receberá todos os tipos de **Messages**, tratando-os e respondendo de acordo. Toda esta comunicação mencionada ocorre através de UDP.

Além disso, **Server** ainda abre uma socket para pedidos TCP, de modo que apresente resposta para pedidos HTTP GET.

3 Especificação do Protocolo

3.1 Formato das mensagens protocolares

Definimos 5 tipos de mensagens, em que os seus campos serão explicitados nas próximas secções:

Table 1. Tipos de Mensagens

opcode	Tipo de Mensagem
0	Metadados
1	Pedido de Ficheiro
2	Dados
3	Ack
4	End

Todos os tipos de mensagem, tem os parâmetros *opcode*, utilizado para identificar o tipo de mensagem, e o valorHmac como um mecanismo simples autenticação mútua.

MetaDados Para cada ficheiro é enviada uma mensagem do tipo MetaDados que, para além de indicar o *opcode* e o valor Hmax, indica o nome do ficheiro (terminado por um delimitador para sabermos onde acaba o nome do ficheiro), a data de modificação e o tamanho do ficheiro.

Table 2. Parâmetros da Mensagem MetaDados

1	variavel com delimiter	8	8	20
opcode	NomeFicheiro	DataModificação	size	valorHmac

Pedido de Ficheiro Verificados os meta dados e tendo uma lista dos ficheiro que lhe fazem falta, começa por pedir um a um, o(s) ficheiro(s) ao parceiro utilizando este tipo de mensagem. Este tipo de mensagem apenas contém o nome de ficheiro para o seu parceiro saber qual ficheiro precisa de enviar.

Table 3. Parâmetros da Mensagem Pedido de Ficheiro

1	variavel com delimiter	20
opcode	NomeFicheiro	valorHmac

Dados Este tipo de mensagem indica: o identificador do ficheiro (que é um valor hash MD5 para não estarmos sempre a enviar o nomeFicheiro), o número de bloco, o tamanho dos dados que vão ser enviados e os dados.

Table 4. Parâmetros da Mensagem Dados

1	32	4	4	TamanhoDados	20
opcode	idFicheiro	numeroBloco	TamanhoDados	Dados	valorHmac

ACK Esta mensagem serve para confirmar se recebeu uma mensagem do tipo dados ou meta dados. O parâmetro de número de bloco é utilizado para garantir que o ack recebido após ser mandado um bloco de dados, condiz com o esperado.

Table 5. Parâmetros da Mensagem ACK

1	4	20
opcode	numeroBloco	valorHmac

End Este tipo de mensagem serve para controlo e é enviada por exemplo quando acabamos de enviar metadados ou dados.

Table 6. Parâmetros da Mensagem End

1	20
opcode	valorHmac

4 Interações

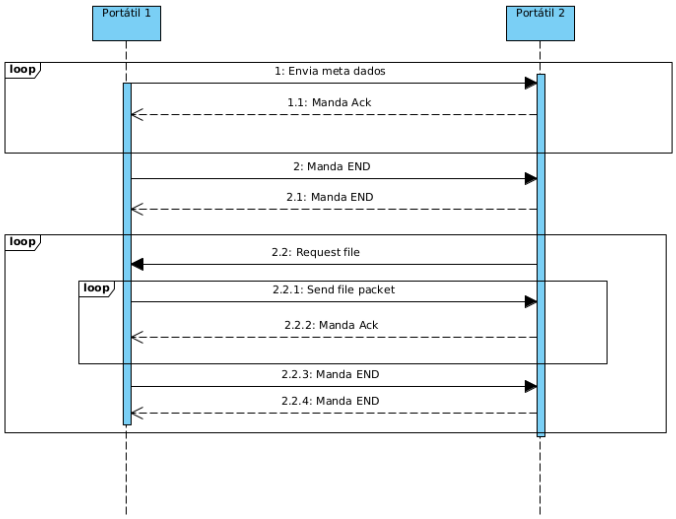

Fig. 1. Interações: Diagrama de sequência

O programa inicia-se calculando os metadados dos ficheiros dentro da pasta escolhida para sincronização. A partir daí, iniciam-se duas *threads* onde se abre um socket na porta 80 para receber pedidos UDP e TCP respectivamente.

O ouvinte UDP está constantemente à espera de receber mensagens. Recebida uma mensagem, será criada uma *thread worker* para a tratar.

A *thread worker* verifica o tipo de mensagem que recebeu (reconhecendo apenas as que foram anteriormente estudadas neste relatório) e, trata de comunicar com quem lhe enviou o pedido.

Por exemplo:

Num computador 1, uma *thread worker* recebe pedido de envio de ficheiro, enviando o primeiro bloco do ficheiro para a porta 80 de quem fez o pedido, recebido o primeiro ack, ele apercebe-se que foi enviado por uma outra porta (que não a 80), então o *worker* passa a enviar os blocos para esta nova porta.

Do outro lado, no computador que fez o pedido, um *worker* vai receber o primeiro bloco de dados, percebendo que não foi enviado pela porta 80, e manda um ack de confirmação para esta nova porta.

Assim, os *workers* passam a comunicar directamente entre eles até terminarem a tarefa que estão a realizar, transferência de ficheiro ou transferência de metadados.

O ouvinte TCP de igual modo está sempre á espera de receber mensagens do tipo *HTTP GET*, respondendo com as informações do estado do programa FFSync, tais como, os *peers* ou ficheiro a sincronizar, entre outras.

Depois de criados estes ouvintes, a aplicação entra num *loop* onde está constantemente, de x em x tempos, a enviar os seus metadados aos seus *peers*, e, caso nos seus metadados tenha metadados que indiquem que um *peer* tem um ficheiro mais atualizado, ele criará um uma *thread* para cada *peer* para pedir os respetivos ficheiros. Por último no ciclo, ele realiza uma atualização/limpeza dos metadados correspondentes ao seu diretório, eliminando metadados de ficheiros que já não existem ou, caso já tenha um ficheiro mais atualizado, muda o peer desse meta dado para null (indicando que ele tem mais atualizado).

5 Implementação

Este sistema foi implementado recorrendo ao Java 15, uma vez que era a linguagem com que os elementos do grupo estavam mais à vontade. Assim sendo, optou-se pela criação de 6 *packages*:

1. **Messages** , onde estão definidas as classes para todos os tipos de mensagens e para cada uma, um método que gera um *array* de bytes com todos os campos necessários para posteriormente enviar
2. **FileMeta**, que organiza todos os meta dados sobre os ficheiros que a aplicação conhece em um map. É a partir destes que a aplicação sabe que ficheiros estão desatualizados e a quem os pedir.
3. **Client**, classe que unicamente gera requests dos ficheiros mais atualizados aos peers.
4. **Server**, classe que trata de receber mensagens e comunicar com workers dos peers. Esta classe é quem manda e recebe ficheiros e recebe metadados.
5. **Http**, classe que inicia um listener constante de mensagens TCP, de modo a responder a pedidos HTTP GET.
6. **Security** , onde está definida a função que calcula o valor HMAC. O algoritmo que usamos foi o HMAC-SHA1. Basicamente os parceiros que se queiram conectar, têm que ter um ficheiro "key.txt" com a mesma chave. Calculamos o valor hmac com uma chave que temos num ficheiro "key.txt" e com os dados da mensagem enviada. Colocamos esse valor no pacote para enviar para o outro parceiro. Quando o seu parceiro recebe esse pacote, recalcula o valor com a chave que tem no seu ficheiro "key.txt" e com os dados que recebe, comparando este valor com o valor que recebeu. Se forem iguais, então aceita a mensagem.

6 Testes e resultados

Para testar a eficácia do nosso programa, realizá-mos os testes propostos no enunciado do tp2:

1. Sincronizar a pasta tp2-folder1 do Servidor1 com uma pasta vazia orca1 no servidor Orca. O programa deverá transferir correctamente o ficheiro de texto em falta. Obter tempo de descarga e débito real em bits/s.

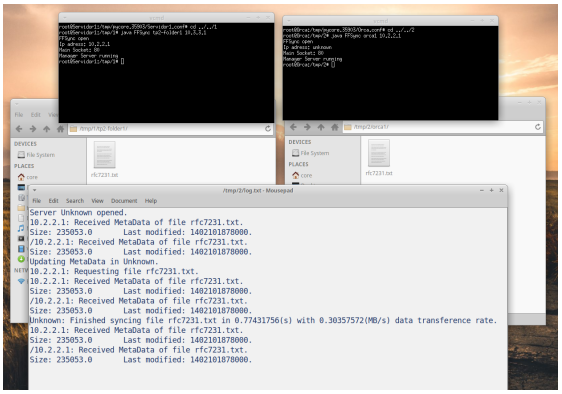

Fig. 2. Teste 1

2. Sincronizar a pasta tp2-folder1 colocada no Servidor1 com a pasta vazia grilo1 no portátil Grilo. No caminho há um link que introduz erros, nomeadamente pacotes perdidos e duplicados. O programa deverá transferir correctamente o ficheiro de texto em falta. Obter tempo de descarga e débito real em bits/s. Comprovar que os conteúdos foram transferidos correctamente.

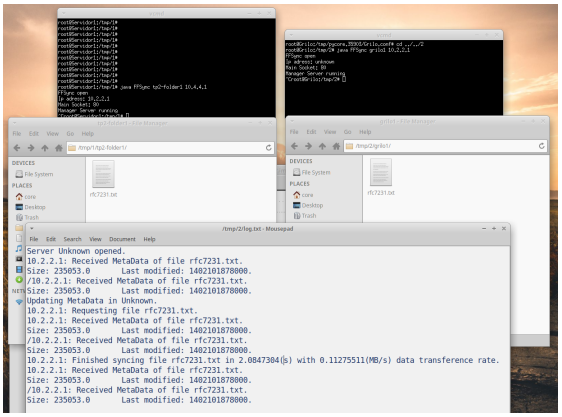

Fig. 3. Teste 2

3. Sincronizar a pasta tp2-folder2 do Servidor1 e a pasta vazia orca2 no servidor Orca. O programa deverá transferir correctamente os múltiplos ficheiros, em simultâneo. Obter tempo de descarga e débito real em bits/s

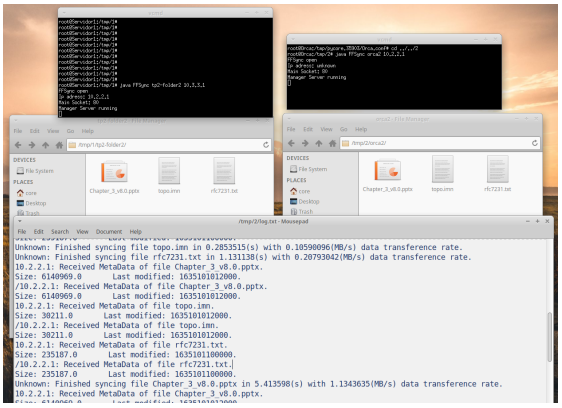

Fig. 4. Teste 3

4. Sincronizar a pasta tp2-folder2 colocada no Servidor1 e com a pasta tp2-folder3 colocada no servidor Orca. Ambas as pastas possuem inicialmente o conteúdo predefinido. Alguns ficheiros são comuns, outros não. Um deles possui uma ligeira diferente de tamanho e de data. O programa deverá conseguir sincronizar as duas pastas transferindo apenas os ficheiros necessários nos dois sentidos. Obter tempo de descarga e débito real em bits/s

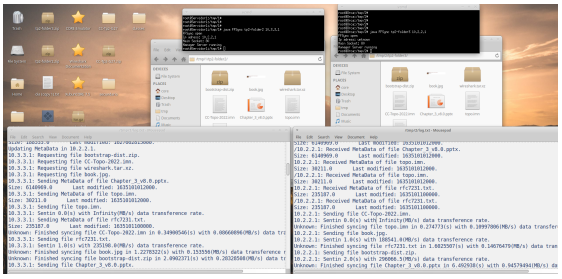

Fig. 5. Teste 4

5. Teste extra: sincronizar 3 pastas em simultâneo

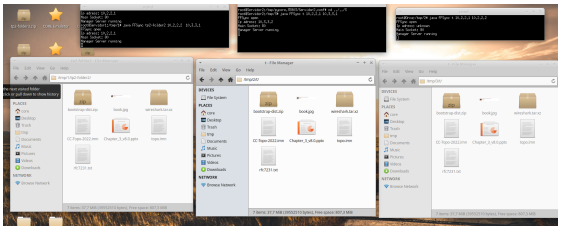

Fig. 6. Teste Extra

Table 7. Resultados dos testes efectuados

Cenário de Teste	File Transferido	Tempo de Transferência(s)	Débito Final(b/s)
1	rfc7231.txt	0.77431756	303575.72
2	rfc7231.txt	2.0847304	112755.11
3	topo.imn	0.2853515	105900.96
	rfc7231.txt	1.131138	207930.42
	Chapter_3_v8.0.pptx	5.413598	1134363.5
4 Serv1	CC-Topo-2022.imn	0.34900546	86608.96
	book.jpg	1.2278322	153556
	bootstrap-dist.zip	2.0902371	283285.08
	wireshark.tar.xz	19.504322	1657853
4 Orca	topo.imn	0.274773	109978.06
	rfc7231.txt	1.6025507	146764.79
	$\text{Chapter}_{3v}8.0.pptx$	6.492938	945794.94

7 Conclusões e trabalho futuro

Com a conclusão deste trabalho, o nosso grupo é agora capaz de reconhecer o trabalho envolvido na formulação e implementação de um protocolo de transporte. Além disso, através dos testes realizados no CORE, conseguimos uma melhor percepção do problema que a perda e duplicação de pacotes quando se usa UDP origina. Por isso mesmo observamos que o tempo de transferência para nodos como o grilo era muito superior a nodos na mesma LAN, uma possível solução para isto seria a implementação de uma janela deslizante para enviar os blocos, estilo TCP.

Quanto à implementação do programa, acreditamos ter realizado uma boa execução dos requisitos pedidos, sendo que este garante uma sincronização entre *peers* fiável, capaz de responder e receber respostas em paralelo e responder a pedidos HTTP GET.

References

1. https://www.infoworld.com/article/2853780/socket-programming-for-scalable-systems.htmlSocket programming
2. https://www.geeksforgeeks.org/md5-hash-in-java/md5 security
3. https://www.jetbrains.com/help/idea/convert-a-regular-project-into-a-maven-project.htmladd$_{m}aven_{s}upportMavenproject$
4. https://www.baeldung.com/java-hmacAlgorithm HMAC