

FolderFastSync: Sincronização rápida de pastas em ambientes serverless

Rui Monteiro^{1[a93179]}, Diogo Barbosa^{1[a93184]}, and Joaquim Roque^{1[a93310]}

Universidade do Minho, Braga
Departamento de Informática

Resumo Desenvolvimento de uma aplicação de sincronização rápida de pastas que não necessita de servidores nem de conectividade Internet, correndo em permanência um protocolo de monitorização simples em HTTP sobre TCP e outro desenvolvido de raiz para a sincronização de ficheiros sobre UDP.

Keywords: UDP · Protocol · HTTP · Folder · TCP.

1 Introdução

O presente projeto, desenvolvido na linguagem de programação Java, consistiu na implementação de uma aplicação de sincronização rápida de pastas sem necessitar de servidores nem de conectividade Internet, designada *FolderFastSync* (**FFSync**). Esta aplicação funciona indicando a pasta a sincronização e o sistema parceiro com quem se irá sincronizar, correndo, posteriormente, um protocolo de sincronização de ficheiros desenvolvido pelo grupo de trabalho que funciona sobre UDP. Para além deste protocolo, também existe um de monitorização simples em HTTP sobre TCP para permitir receber e responder a pedidos deste género.

2 Arquitetura da solução

A nossa solução arquitetural divide-se principalmente em quatro *packages* distintos, sendo eles: `udp`, `tcp`, `packet` e `utils`. Os *peers* comunicam entre si por UDP, definido nas classes **UDP_Handler.java**, **UDP_Listener.java** e **UDP_Sender.java** utilizando os pacotes definidos na classe **Packet.java** do *package* `packet`, sendo que a informação é adicionada aos *logs* nas classes responsáveis por enviar essa informação.

No que toca ao atendimento em TCP, serve para receber pedidos simples de HTTP e verificar o estado da aplicação, definidos principalmente nas classes **TCP_Listener.java** e **TCP_Handler** do *package* `tcp`.

Em relação ao *package* `utils`, contém algumas classes auxiliares ao funcionamento do projeto em geral.

3 Especificação do protocolo

3.1 Formato das mensagens protocolares

O protocolo desenvolvido pelo grupo de trabalho prevê os seguintes tipos de mensagens:

- **FILE_META** - Tem como função enviar os metadados e o HMAC, para verificar a integridade e segurança, de um ficheiro. Possui como parâmetros um *opcode* que identifica o tipo de pacote, um *md5Hash* que funciona como identificador único do ficheiro, o *nameSize* que serve para armazenar o tamanho do nome do ficheiro em questão, um campo *filename* que armazena o nome do ficheiro em si, um parâmetro *hasNext* que indica se existem mais ficheiros para enviar metadados após o envio deste pacote e, por fim, um campo *HMAC* que serve para verificar a integridade do pacote, sendo usado como palavra-passe partilhada entre os *peers*.
- **DATA_TRANSFER** - Serve para fazer o envio de *chunks* de *bytes* de um determinado ficheiro. Tem como campos um *opcode* que identifica o tipo de pacote, um *sequenceNumber* que representa o número de sequência para nos ser possível construir corretamente o ficheiro no destino, um parâmetro *md5Hash* que funciona como identificador único do ficheiro que estamos a transferir, um *hasNext* que indica se existem mais pacotes de dados relativos a este ficheiro, um campo *dataSize* que armazena o tamanho do campo que lhe sucede e, finalmente, um parâmetro *data* que contém os *bytes* do ficheiro propriamente ditos, que podem ser no máximo 1450.
- **ACK** - Funciona como um *acknowledgment* dos pacotes de DATA_TRANSFER, sendo feito de pacote a pacote. Possui um *opcode* que identifica o tipo de pacote, um *sequenceNumber* que representa o número de sequência para nos permitir verificar se os dados estão a ser recebidos pelo outro *peer*, um *md5Hash* que funciona como identificador único do ficheiro e um campo *HMAC* que serve como mecanismo simples de autenticação mútua.

Cada mensagem é serializada, anteriormente a ser enviada, pelo método *serialize* contida no ficheiro **Packet.java**. Posteriormente, ao receber a mensagem, o receptor deverá deserializá-la através do método *deserialize* do mesmo ficheiro, de forma a converter os *bytes* recebidos em pacotes de formato legível, proporcionando facilidade de uso.

O tamanho do *Packet* foi decidido pelo grupo de trabalho como 1472 *bytes*, no máximo, tendo em conta que o *header* do UDP tem um *overhead* de 8 *bytes* e o *header* IP tem um *overhead* de 20 *bytes*, a soma destes valores com o tamanho máximo do pacote corresponde a 1500 *bytes*, que é o maior tamanho da mensagem do nosso protocolo.

Para assegurar a segurança dos dados enviados usamos HMAC juntamente com SHA-1, que calculamos antes de enviar os pacotes de metadados e os *acks* e quando os recebemos voltamos a calculá-lo para compararmos ambos os valores, de modo a garantir a autenticação mútua com recurso ao uso de um segredo partilhado.

3.2 Interações

O programa inicia-se computando os pacotes de metadados dos ficheiros na pasta especificada. Este processo é cíclico (repete-se a cada 20 segundos).

O *HostA* envia um pacote indiferenciado de metadados para o endereço e *port* especificado. Nesse momento, este entra em *timeout*, aguardando que o *HostB* lhe envie um pacote de modo a garantir que este se encontra ativo. No momento em que o *HostB* é iniciado, este repete o mesmo processo descrito anteriormente e, a partir desse momento, ambos sabem que estão ativos e iniciam a troca de mensagens. No caso de não existirem metadados na diretoria local, o *Host* simplesmente um pacote sem significado para o outro *Host* de modo a assinalar que está ativo.

Uma vez concluída a transferência de metadados, cada *Host* calcula as diferenças entre os ficheiros na diretoria local e os da diretoria do *peer*. Aí iniciam-se as trocas de pacotes de dados (se necessário). Cada pacote de dados enviado necessita de ser confirmado pelo outro *Host* através de um *ACK*. No caso deste pacote de confirmação não ser enviado, o *Host* a enviar não atualiza o número de sequência e reenvia o mesmo pacote.

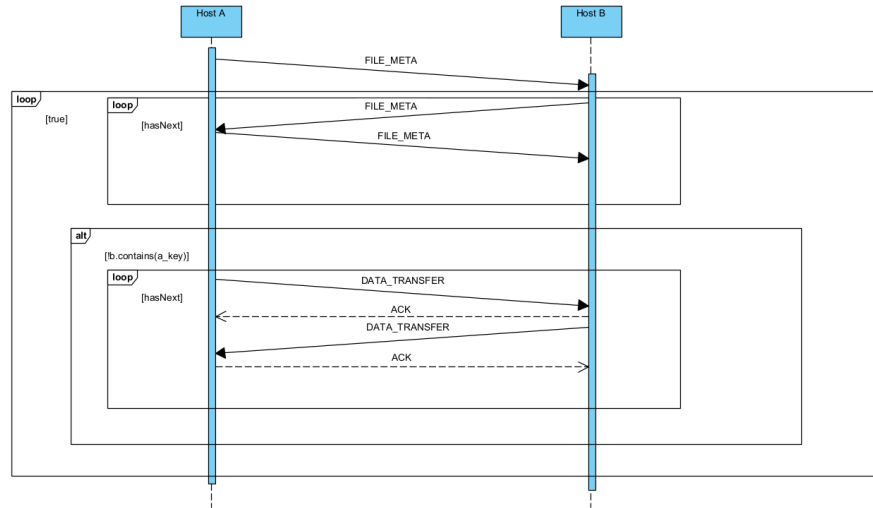


Figura 1. Interações: diagrama de sequência

4 Implementação

No que toca à implementação, temos 4 ficheiros principais:

- **Packet.java** - trata da implementação dos pacotes do nosso protocolo.

- **TCP_Handler.java** - trata do atendimento em TCP dos pedidos de HTTP.
- **FileTracker.java** - trata da monitorização dos metadados e dados, recebidos e enviados, e *logs*.
- **UDP_Handler.java** - trata do atendimento em UDP dos pacotes recebidos.
- **UDP_Sender.java** - trata do envio de pacotes de metadados e dados.

4.1 Packet.java

A classe *Packet* serve como abstração para os *DatagramPacket*. Isto permite maior flexibilidade e facilidade na comunicação entre os dois *peers*, pois evita manipular vetores de *bytes* diretamente, reduzindo a propensão a erros. Existem construtores distintos para os três tipos de pacote, recebendo como parâmetros os dados necessários. Existem ainda métodos de serialização e desserialização, essenciais para poder comunicar e interpretar os pacotes.

4.2 TCP_Handler.java

A classe *TCP_Handler* é instanciada aquando da receção de um pedido pelo *TCP_Listener*. Esta distingue três tipos de pedido:

- */* - apresenta informação sobre o protocolo desenvolvido.
- */logs* - lista os *logs* de registo no momento do pedido.
- */files* - lista os ficheiros atualmente na diretoria.

Outros pedidos resultarão em erro.

4.3 FileTracker.java

Esta classe é eventualmente a mais importante do protocolo, pois funciona como demultiplexador e monitorizador. É instanciada uma única vez no início do programa, na *main* e é partilhada por diversos outros objetos do programa. Esta contém estruturas (mapas) que associam a cada identificador único (calculado através de metadados) pacotes de metadados locais, metadados do *peer*, *acks* a receber, escritores para ficheiros e uma lista dos *logs*.

A monitorização da diretoria local é feita por uma classe interna que, a cada 20 segundos, atualiza o mapa de metadados locais adequadamente. Por outro lado, o mapa que trata da monitorização remota é atualizado aquando da receção de novos metadados. Estas duas estruturas permitem calcular as diferenças entre ambas as pastas e decidir quais os ficheiros a enviar. No momento em que esse cálculo é feito, é também atualizado o mapa dos *acks*, uma vez que já temos a noção que pacotes de confirmação iremos receber por parte do *peer*.

Os escritores para ficheiros são uma classe que permite escrever para ficheiros. Um pacote de dados recebido é escrito para o ficheiro adequado, de acordo com o identificador.

4.4 UDP_Handler.java

O *UDP_Handler* é instanciado pelo *UDP_Listener* assim que chega um pacote. A função desta classe é tratar os pacotes recebidos, comunicando os mesmos para o *FileTracker*. No caso de se tratar de um pacote de dados, é também enviado um *ack*.

4.5 UDP_Sender.java

O *UDP_Sender* é um único objeto responsável pelo envio de metadados e dados. Este está em execução constante, sendo que a cada 20 segundos inicia uma nova iteração de envio de pacotes. Começa por requisitar a lista de metadados de a enviar ao *FileTracker*, e de seguida a lista dos dados. Na primeira iteração, este começa por assumir que o *peer* não está ativo até receber confirmação externa que indique o contrário. Nesse caso, o *sender* envia um pacote e suspende-se de seguida. No caso da diretoria não conter qualquer ficheiro, o *Host* envia um pacote sem significado algum, servindo apenas para garantir que o *peer* reconhece que este está ativo.

5 Testes e resultados

Ao longo da realização deste projeto, fizemos vários testes com ficheiros diferentes, ou seja, com tamanhos variados e tipos diferentes, além de termos testado com números de ficheiros diferentes. No presente relatório apenas apresentaremos alguns testes que fizemos num ambiente *GNU – Linux*, sem recurso ao emulador *CORE* e à topologia disponibilizada nas aulas práticas, mas utilizando os ficheiros disponibilizados pelos professores para o efeito.

5.1 Teste 1

No primeiro teste o *peer1* (Orca) enviou uma mensagem a informar o *peer2* (Servidor1) que tem a pasta vazia e o *peer2* enviou um *FILE_META* que informa o outro *peer* que tem o ficheiro "rfc7231.txt". Após haver essa troca de informação, o *peer2* enviou o "rfc7231.txt" para *peer1* ficando assim sincronizado com ele.

```
Pacote recebido com número de sequência: 161
20:45:58.483 -> enviando ack 161
Pacote recebido com número de sequência: 162
20:45:58.490 -> enviando ack 162
rfc7231.txt foi recebido e guardado em 2.244 segundos a uma taxa de 104747.32620320855 bytes/s
```

Figura 2. Logs do Peer1 no fim da sincronização

```
Pacote enviado com número de sequência: 161
20:45:58.483 -> recebendo ack 161
Pacote enviado com número de sequência: 162
20:45:58.490 -> recebendo ack 162
rfc7231.txt foi enviado com sucesso
```

Figura 3. Logs do Peer2 no fim da sincronização

5.2 Teste 3

Tal como no Teste 1, no Teste 3 o *peer1* (Orca) enviou uma mensagem a informar o *peer2* (Servidor1) que a sua pasta está vazia e o mesmo enviou três mensagens *FILE_META*, uma para cada ficheiro que possui. De seguida, o *peer2* enviou ao *peer1* todos os ficheiros que o mesmo tem em falta. Dado este processo como finalizado, as pastas ficaram sincronizadas.

```
Pacote recebido com número de sequência: 160
20:55:10.219 -> enviando ack 160
Pacote recebido com número de sequência: 161
20:55:10.224 -> enviando ack 161
Pacote recebido com número de sequência: 162
20:55:10.229 -> enviando ack 162
rfc7231.txt foi recebido e guardado em 0.84 segundos a uma taxa de 279984.52380952385 bytes/s
```

Figura 4. Logs do Peer1 no fim da sincronização

```
Pacote enviado com número de sequência: 160
20:55:10.219 -> recebendo ack 160
Pacote enviado com número de sequência: 161
20:55:10.224 -> recebendo ack 161
Pacote enviado com número de sequência: 162
20:55:10.229 -> recebendo ack 162
rfc7231.txt foi enviado com sucesso
```

Figura 5. Logs do Peer2 no fim da sincronização

5.3 Teste 4

No Teste 4 ambos os *peers* enviaram mensagens *FILE_META* um ao outro simultaneamente, para ambos os lados possuírem a informação dos ficheiros um

do outro. De seguida, cada *peer* comparou os seus ficheiros com os do outro para determinar que ficheiros necessitam de ser enviados para que ambas as pastas estejam sincronizadas. Os *peers* enviam os ficheiros selecionados um ao outro, de maneira a ficarem sincronizados no final. No entanto, neste teste ao executar o comando *diff*, foi encontrada uma diferença entre o conteúdo do ficheiro "rfc7231.txt" de cada *peer*, visto que o nosso protocolo considera ficheiros com o mesmo nome como iguais.

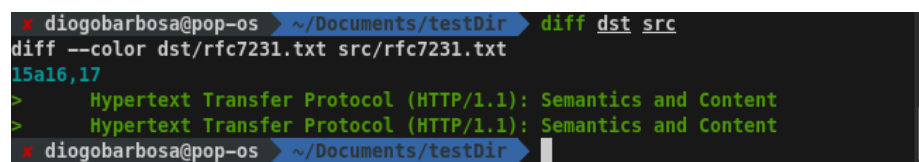
A terminal window with a dark background. The prompt is 'diogobarbosa@pop-os' followed by a blue arrow pointing to '~/.Documents/testDir'. The command 'diff dst src' is entered. The output shows 'diff --color dst/rfc7231.txt src/rfc7231.txt' followed by '15a16,17' in red. Then, two lines of green text are shown: '> Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content' and '> Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content'. The prompt is repeated at the bottom.

Figura 6. Terminal que mostra a execução do comando *diff* sobre as duas pastas no final da sincronização

6 Conclusão e trabalho futuro

Através da realização deste projeto conseguimos implementar em grande parte os requisitos obrigatórios de forma eficaz e, assim, aplicar os conhecimentos adquiridos ao longo do semestre nesta unidade curricular, apesar de não termos tido a unidade curricular de Redes de Computadores anteriormente, o que dificultou essa aprendizagem.

Sentimos dificuldades ao implementar uma forma de conseguir atender múltiplos pedidos em simultâneo nos dois sentidos e, lamentavelmente, não cumprimos com este requisito. Adicionalmente, fomos incapazes de adicionar qualquer dos requisitos opcionais sugeridos no enunciado.

Como trabalho futuro, seria necessário implementar envio de ficheiros de forma concorrente, corrigindo problemas de concorrência existentes. A classe *FileTracker* necessitaria de ser subdivida em várias classes, uma vez que esta já apresenta uma elevada complexidade. As funcionalidades de desmultiplexagem deviam ser exploradas com maior detalhe de modo a melhorar a *performance*, fazendo uso do conhecimento de programação concorrente lecionado na Unidade Curricular de Sistemas Distribuídos.