



UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

TP2: FolderFastSync: Sincronização rápida de  
pastas em ambientes serverless  
PL5 – Grupo 55

Francisco Peixoto (a84668)      Gonçalo Soares (a93286)  
Samuel Lira (a94166)

Dezembro 2021

# Conteúdo

<b>1</b>	<b>Introdução e Objetivos</b>	<b>ii</b>
<b>2</b>	<b>Arquitetura do Projeto</b>	<b>ii</b>
2.1	Síntese dos Requisitos Implementados . . . . .	ii
<b>3</b>	<b>Estratégia de Solução</b>	<b>iii</b>
3.1	Arquitetura da aplicação . . . . .	iii
3.1.1	Sender . . . . .	iii
3.1.2	Receiver . . . . .	iii
3.2	Início de ligação . . . . .	iii
3.3	Fim de ligação . . . . .	iii
3.4	Especificação do Protocolo . . . . .	iv
3.4.1	Header do pacote . . . . .	iv
3.4.2	Get . . . . .	v
3.4.3	Data . . . . .	vi
3.4.4	Ack . . . . .	vi
3.4.5	Metadata . . . . .	vi
3.4.6	Notify . . . . .	vi
3.4.7	Error . . . . .	vi
3.5	Controlo de erros . . . . .	vi
3.5.1	Acknowledgement . . . . .	vi
3.5.2	Stop and Wait protocol . . . . .	vi
3.5.3	Sliding Window protocol . . . . .	vii
3.6	Segurança: Encriptação . . . . .	vii
3.7	Ficheiros . . . . .	vii
3.7.1	Deteção de diferentes ficheiros na diretoria . . . . .	vii
3.7.2	Compressão de ficheiros aquando do envio . . . . .	vii
3.7.3	Envio e recepção de ficheiros . . . . .	vii
3.7.4	Folder Stats . . . . .	ix
3.8	Serviço de Monitorização . . . . .	x
<b>4</b>	<b>Testes e Resultados</b>	<b>xi</b>
<b>5</b>	<b>Bibliotecas Utilizadas</b>	<b>xi</b>
<b>6</b>	<b>Conclusão</b>	<b>xii</b>

# Lista de Figuras

1	Diagrama da Topologia da Aplicação FFSync . . . . .	ii
2	Pacote encriptado, sendo n um número de bytes não maior que o MTU usado para a camada de aplicação . . . . .	iv
3	Pacote não encriptado, em bytes . . . . .	iv
4	Diagrama sequencial do pedido de ficheiro . . . . .	v
5	Diagrama sequencial do pedido de ficheiro . . . . .	viii

6	Diagrama de Classes relativo ao FTRapid . . . . .	ix
7	Diagrama de Classes relativo ao serviço TCP . . . . .	x
8	Exemplo de um ficheiro de logs em que os nodos se ligam. . . . .	xi
9	Exemplo de um ficheiro de logs em que os nodos se sincronizam. . . . .	xi

## 1 Introdução e Objetivos

Este trabalho surgiu no âmbito da Unidade Curricular de *Comunicações por Computador* e tem como objetivo implementar uma aplicação de sincronização rápida de pastas sem necessitar de servidores nem de conectividade Internet, designada por FolderFastSync (FFSync).

A aplicação utiliza como parâmetros quer a pasta a sincronizar quer o sistema parceiro com quem se vai sincronizar.

A aplicação corre em permanência dois protocolos: um de monitorização simples em HTTP sobre TCP e outro, a desenvolver de raiz para a sincronização de ficheiros sobre UDP.

## 2 Arquitetura do Projeto

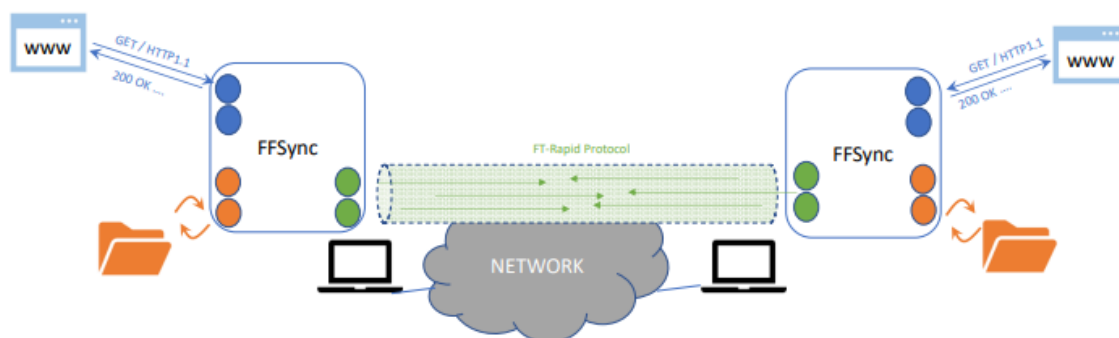


Figura 1: Esquema geral de funcionamento

Figura 1: Diagrama da Topologia da Aplicação FFSync

### 2.1 Síntese dos Requisitos Implementados

- O sistema consegue responder a pedidos HTTP GET devolvendo uma página em *HTML* relativa à homepage, que explicita todas as subpáginas presentes no serviço, nomeadamente */status* e */log*.
- O sistema consegue atender múltiplos pedidos em simultâneo nos dois sentidos.
- O sistema é eficaz, no sentido em que consegue cumprir corretamente a função de sincronização, de pastas e de subpastas.
- O sistema tem um registo (logs) que indica as operações no decorrer de sua execução. A informação a ser exibida descreve interações de envios de pacotes e conexões do serviço

a endereços. O ficheiro de Logs consegue ser lido em formato .txt ou consultado via browser pelo serviço TCP.

- Os valores relativos ao tempo de transferência e débito final em bits por segundo são registados nos Logs, como mostra a figura 9.
- No que diz respeito à segurança o sistema implementa um mecanismo simples autenticação mútua, seguindo o protocolo HMAC, e ainda um mecanismo de encriptação de pacotes usando o algoritmo *AES*, bem como compressão dos ficheiros a enviar por UDP.

### 3 Estratégia de Solução

Nesta secção do relatório pretendemos explicitar a arquitetura que o grupo desenvolveu para responder ao problema proposto.

#### 3.1 Arquitetura da aplicação

Como arquitetura desta aplicação *peer-to-peer*, temos dois componentes principais: o *sender* e *receiver*, que estabelecem a conexão entre os diversos nodos na rede, através de sockets UDP.

##### 3.1.1 Sender

O *sender* é a entidade responsável por mandar pedidos aos outros nodos conectados na rede. No início da aplicação, este envia um pedido para conhecer a estrutura dos ficheiros e sub-pastas da pasta escolhida para ser sincronizada. Após receber a resposta ao seu pedido, este analisa quais são as diferenças entre a sua pasta e a do outro nodo, e pede os ficheiros necessários para si e manda os metadados dos ficheiros necessários para o outro nodo.

##### 3.1.2 Receiver

O *receiver* tem como função receber pedidos e atendê-los. Podendo estes pedidos, ser de um ficheiro, metadados da diretoria ou até da recepção dos metadados dos ficheiros necessários para atualizar a diretoria.

#### 3.2 Início de ligação

Como início de ligação temos apenas o envio do pedido que, posteriormente, vai ser atendido pelo *receiver* do outro nodo. Ao receber o pedido, é enviado um **Acknowledgement** para confirmar a sua recepção. Depois é iniciado o processamento do pedido.

#### 3.3 Fim de ligação

De modo a terminar a ligação, o *sender* tem a informação de quantos pacotes lhe são enviados se for um ficheiro. Caso contrário, assume-se, de modo a simplificar, que o número de pacotes a ser enviado é 1 e que este não ultrapassa o *MTU* definido para a camada de aplicação. Isto, para dizer, que o *sender* saberá sempre o quantidade de pacotes que vai receber depois de ter efetuado o pedido, e, por isso, sabe quando deve terminar a ligação.

### 3.4 Especificação do Protocolo

Fala-se inicialmente sobre o *header* utilizado no protocolo, tal como as suas vantagens e desvantagens.

De seguida, apresenta-se os vários tipos de pacotes acompanhados com o **formato da sua mensagem** e a sua **função e significado dos campos**. Para os pacotes principais é apresentado um **diagrama temporal ilustrativo**.

#### 3.4.1 Header do pacote

Na solução desenvolvida existem dois *headers* em duas fases diferentes. Aquando do envio de um pacote, o seu *header* contém 4 bytes iniciais relativos ao tamanho do pacote de dados encriptados.

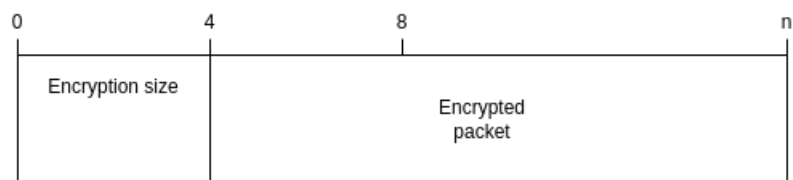


Figura 2: Pacote encriptado, sendo n um número de bytes não maior que o MTU usado para a camada de aplicação

Após a descriptação, o pacote fica com outro formato como podemos ver na Figura 5.

O *overhead* de 20 bytes corresponde ao código de hash do *HMAC Header*, que garante a autenticidade e integridade da mensagem através da confirmação de um hashcode gerado pelo algoritmo *SHA1* e uma *key* definida pelo grupo. O *Header* contém ainda um byte que define o tipo de mensagem enviada, o resto do pacote corresponde aos dados enviados.

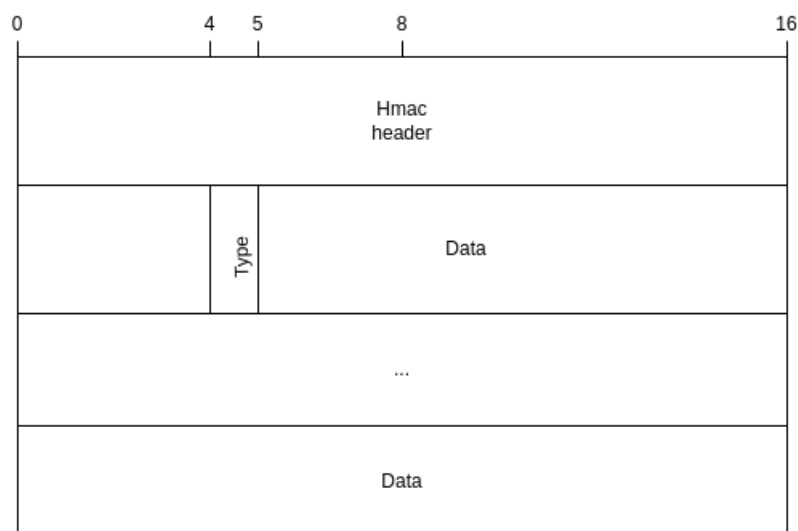


Figura 3: Pacote não encriptado, em bytes

### 3.4.2 Get

O pacote *Get* tem como objetivo fazer pedidos tanto de ficheiros como de meta-dados da diretoria que se pretende sincronizar.

O pacote é composto por dois campos booleanos. Um que confirma se se pretende meta-dados ou não e outro, relativo ao campo *root* que será *true* se for pretendido os dados referentes à diretoria toda. Como campo final, temos uma lista com os nodos de ficheiros, que será apenas utilizada caso o campo *root* seja falso.

Abaixo conseguimos ver um exemplo do uso do pacote *GET* para obter os meta-dados da diretoria de outro nodo.

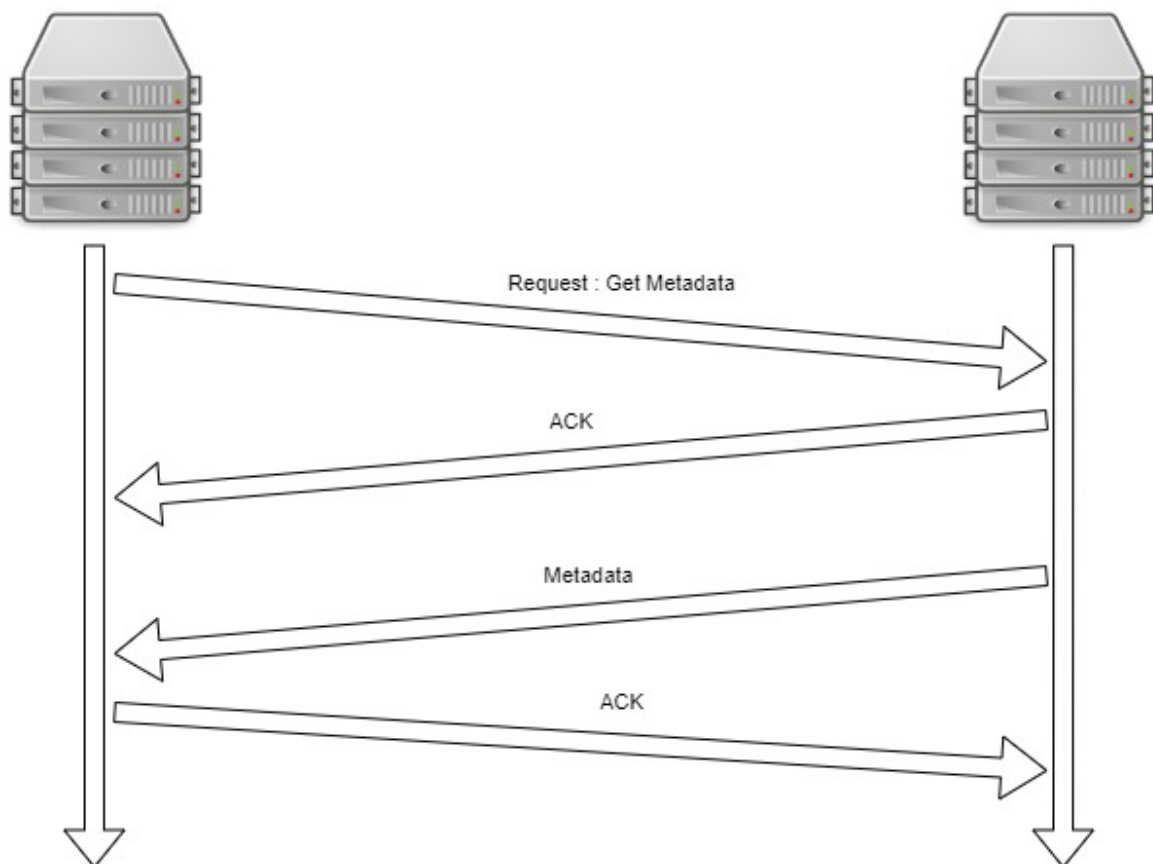


Figura 4: Diagrama sequencial do pedido de ficheiro

### 3.4.3 Data

Pacotes do tipo *Data* são utilizados para enviar *chunks* de ficheiros. E têm, como campos, o número do bloco e o *array* de bytes correspondentes aos dados do *chunk*.

### 3.4.4 Ack

O pacote *Ack* é utilizado para sinalizar um *acknowledgement*, ou seja, demonstrar que foi recebido um determinado pacote. O número do pacote que se pretende confirmar como recebido está no único campo deste pacote, que corresponde ao número de segmento.

### 3.4.5 Metadata

Os pacotes do tipo *Metadata* têm as seguintes características: pretendem representar os meta-dados pedidos através de um pacote do tipo *Get* (como podemos ver na figura 6) ou indicar a um nodo que este precisa de um certo conjunto de ficheiros. Como campo de dados, temos uma lista de pares do tipo {ficheiro/ultima data de modificação}, que, depois da fase de deserialização do pacote, é guardado como um *Map*.

### 3.4.6 Notify

De modo, a conseguir notificar os outros nodos de mudanças na pasta, foi criado o pacote *Notify*. A forma pensada de detetar as eventuais mudanças era através da implementação de *WatchKey's* nos ficheiros da diretoria desejada que notificariam o programa quando certos eventos ocorrem. Infelizmente, esta parte do projeto não foi realizada. Tendo sido implementado uma versão mais simplista, que a cada 60 segundos compara os seus meta-dados com os dos outros nodos. Com esta versão, este pacote não é usado. Porém os campos deste são análogos ao do pacote *Metadata*.

### 3.4.7 Error

Este pacote tem o objetivo de comunicar erros entre os diversos nodos. Esta parte não se encontra totalmente desenvolvida, porém pode ser usada em várias situações. Como, por exemplo, na receção de um pedido inválido, o *receiver* pode responder com este pacote.

## 3.5 Controlo de erros

### 3.5.1 Acknowledgement

Tal como referido atrás, o pacote *Ack* é usado para sinalizar a receção de um determinado pacote. Porém, ao longo do envio de pacotes, são usados dois tipos de *Ack's* diferentes. Um que sinaliza que **um** pacote foi recebido e outro chamado de **acumulativo** que comunica a receção de todos os pacotes que tenham o número de segmento igual ou menor ao do *Ack*.

### 3.5.2 Stop and Wait protocol

Neste projeto, é utilizado para o envio e receção de mensagens simples. O conceito por de trás deste protocolo é o de esperar sempre por uma confirmação de receção de pacote, antes de se enviar o proximo. É utilizado em todo o programa com exceção do envio e receção de ficheiros.

### 3.5.3 Sliding Window protocol

Devido à sua maior eficiência em relação ao *Stop and Wait protocol* este foi implementação numa das partes onde o que mais interessa é ser eficiente, no *download* e *upload* de ficheiros. A ideia básica deste protocolo, é a de se mandar um certo número de pacotes seguidos, esperar por um *Ack* e voltar a enviar pacotes a partir do número de sequência do *Ack* recebido.

## 3.6 Segurança: Encriptação

De modo a garantir a segurança no envio de pacotes de dados entre os nodos na rede, o sistema realiza encriptação dos pacotes aquando do envio. A encriptação recorre ao algoritmo "AES" e a uma chave gerada pelo grupo.

## 3.7 Ficheiros

### 3.7.1 Detecção de diferentes ficheiros na diretoria

De modo, a simplificar, são detetados diferentes ficheiros apenas através do nome e da data da ultima modificação.

### 3.7.2 Compressão de ficheiros aquando do envio

No envio dos ficheiros, de modo a não saturar a rede, realiza-se compressão antes da divisão do ficheiro em pacotes, desta forma garantimos que ficheiros grandes não causam um impacto tão acentuado no tráfego da rede. A compressão utiliza *GZIP*, uma biblioteca de Java.

### 3.7.3 Envio e recepção de ficheiros

Antes do envio é efetuada a compressão do ficheiro e a divisão em blocos (que não ultrapassem o tamanho do MTU da camada de aplicação). De seguida, é mandado um pacote *Ack* por parte do *receiver* (que atende os pedidos, neste caso não é quem recebe o ficheiro) a indicar a quantidade de pacotes que serão enviados para ser efetuado o *download* do ficheiro. No lado do nodo que recebe o ficheiro, o processo é análogo. A descrição deste processo todo, pode ser vista na figura abaixo.



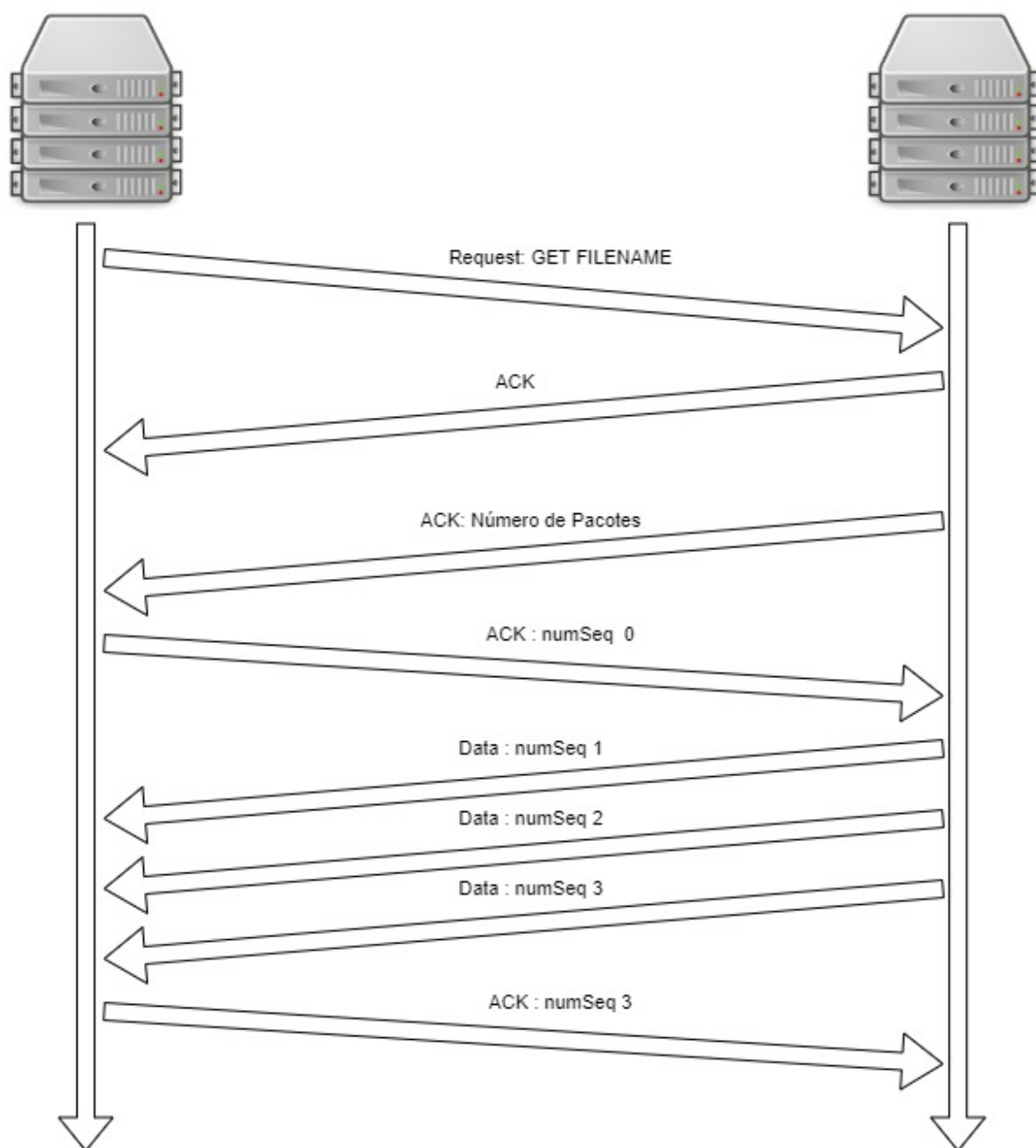


Figura 5: Diagrama sequencial do pedido de ficheiro

### 3.7.4 Folder Stats

A classe FolderStats contém exclusivamente o método estático metadata que é utilizado para percorrer a diretoria atual e criar um HashMap contendo o nome dos ficheiros/subpastas e o tempo da última modificação deste ficheiro/pasta, afim de facilitar a comparação dos ficheiros posteriormente durante a fase de decisão entre quais os ficheiros a serem enviados.

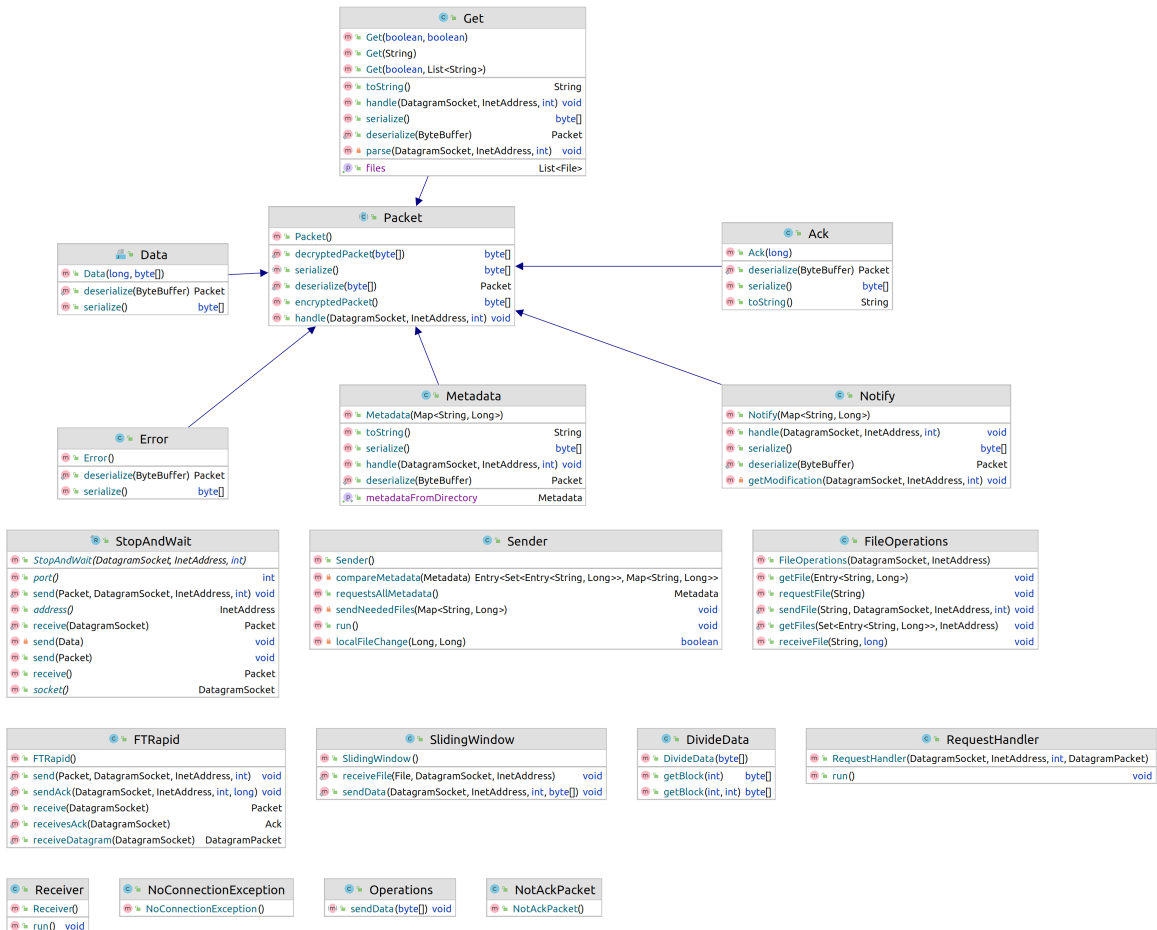


Figura 6: Diagrama de Classes relativo ao FTRapid

### 3.8 Serviço de Monitorização

Na implementação do serviço que corre sobre TCP, a abordagem foi cada nodo correr um servidor que responde a pedidos TCP na porta 5555.

O servidor devolve páginas HTML para 3 routes, "/" em que se colocou algumas informações sobre as páginas, "/log" onde o utilizador pode ver o conteúdo dos ficheiros de Logs (cada sessão cria um ficheiro de Logs) e por fim "/status" onde podemos visualizar vários dados básicos sobre todos os ficheiros sincronizados da máquina.

O servidor suporta vários *ClientHandler* a correr em *Threads* em paralelo.

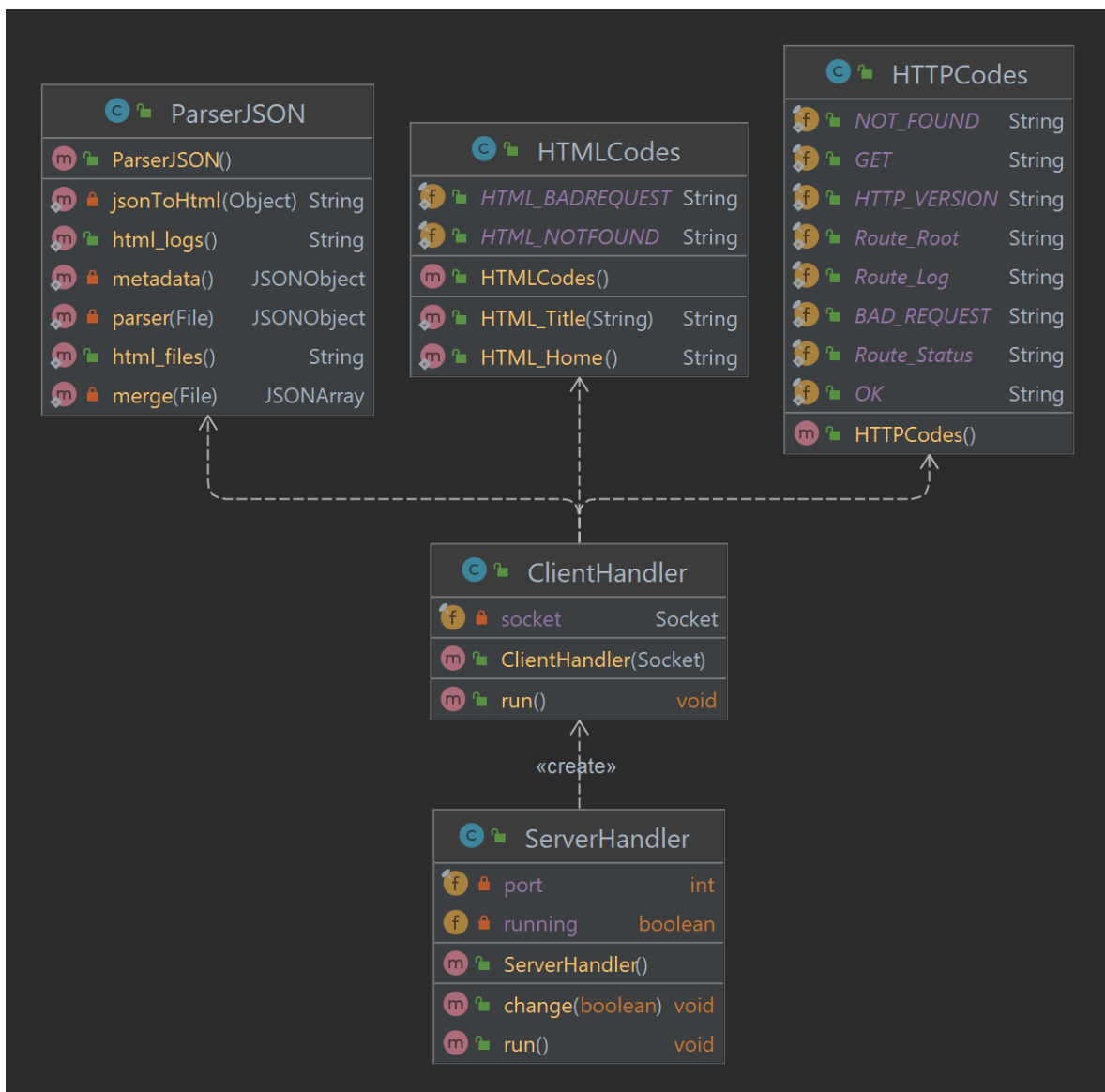
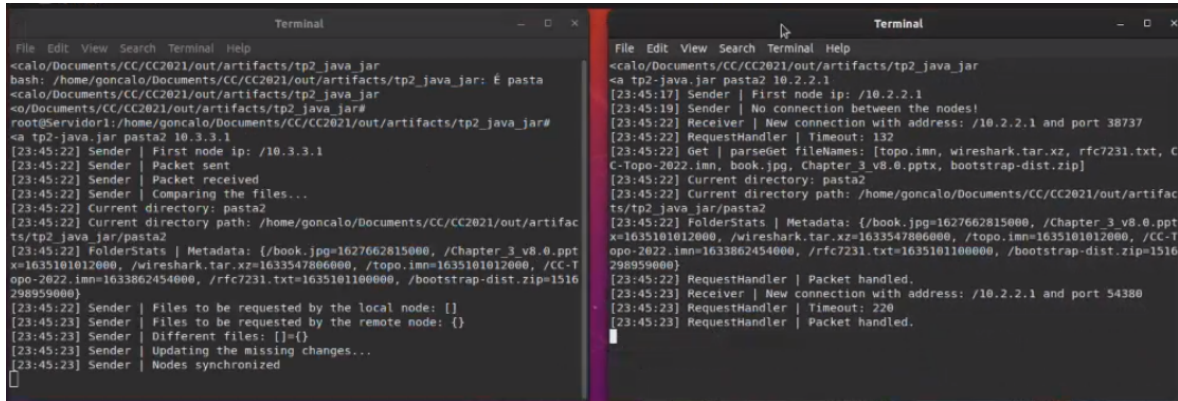


Figura 7: Diagrama de Classes relativo ao serviço TCP

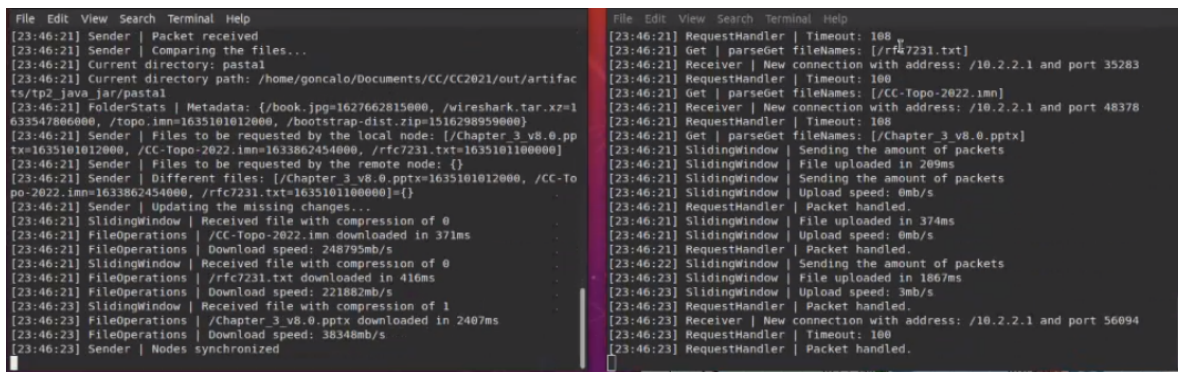
## 4 Testes e Resultados



```
File Edit View Search Terminal Help
<calo/Documents/CC/CC2021/out/artifacts/tp2_java.jar
bash: /home/goncalo/Documents/CC/CC2021/out/artifacts/tp2_java.jar: É pasta
<calo/Documents/CC/CC2021/out/artifacts/tp2_java.jar
<co/Documents/CC/CC2021/out/artifacts/tp2_java.jar#
root@Servidor1:~/home/goncalo/Documents/CC/CC2021/out/artifacts/tp2_java.jar#
<a tp2-java.jar pasta2 10.3.3.1
[23:45:22] Sender | First node ip: /10.3.3.1
[23:45:22] Sender | Packet sent
[23:45:22] Sender | Packet received
[23:45:22] Sender | Comparing the files...
[23:45:22] Current directory: pasta2
[23:45:22] Current directory path: /home/goncalo/Documents/CC/CC2021/out/artifac
ts/tp2_java.jar/pasta2
[23:45:22] FolderStats | Metadata: [/book.jpg=1627662815000, /Chapter_3_v8.0.ppt
x=1635101012000, /wireshark.tar.xz=1633547806000, /topo.imn=1635101012000, /CC-T
opo-2022.imn=1633862454000, /rfc7231.txt=1635101100000, /bootstrap-dist.zip=1516
298959000]
[23:45:22] Sender | Files to be requested by the local node: []
[23:45:23] Sender | Files to be requested by the remote node: {}
[23:45:23] Sender | Different files: []={}
[23:45:23] Sender | Updating the missing changes...
[23:45:23] Sender | Nodes synchronized

File Edit View Search Terminal Help
<calo/Documents/CC/CC2021/out/artifacts/tp2_java.jar
<a tp2-java.jar pasta2 10.2.2.1
[23:45:17] Sender | First node ip: /10.2.2.1
[23:45:19] Sender | No connection between the nodes!
[23:45:22] Receiver | New connection with address: /10.2.2.1 and port 38737
[23:45:22] RequestHandler | Timeout: 132
[23:45:22] Get | parseGet fileNames: [topo.imn, wireshark.tar.xz, rfc7231.txt, C
C-Topo-2022.imn, book.jpg, Chapter_3_v8.0.pptx, bootstrap-dist.zip]
[23:45:22] Current directory: pasta2
[23:45:22] Current directory path: /home/goncalo/Documents/CC/CC2021/out/artifac
ts/tp2_java.jar/pasta2
[23:45:22] FolderStats | Metadata: [/book.jpg=1627662815000, /Chapter_3_v8.0.ppt
x=1635101012000, /wireshark.tar.xz=1633547806000, /topo.imn=1635101012000, /CC-T
opo-2022.imn=1633862454000, /rfc7231.txt=1635101100000, /bootstrap-dist.zip=1516
298959000]
[23:45:22] RequestHandler | Packet handled.
[23:45:23] Receiver | New connection with address: /10.2.2.1 and port 54380
[23:45:23] RequestHandler | Timeout: 220
[23:45:23] RequestHandler | Packet handled.
```

Figura 8: Exemplo de um ficheiro de logs em que os nodos se ligam.



```
File Edit View Search Terminal Help
[23:46:21] Sender | Packet received
[23:46:21] Sender | Comparing the files...
[23:46:21] Current directory: pasta1
[23:46:21] Current directory path: /home/goncalo/Documents/CC/CC2021/out/artifac
ts/tp2_java.jar/pasta1
[23:46:21] FolderStats | Metadata: [/book.jpg=1627662815000, /wireshark.tar.xz=1
633547806000, /topo.imn=1635101012000, /bootstrap-dist.zip=1516298959000]
[23:46:21] Sender | Files to be requested by the local node: [/Chapter_3_v8.0.pp
tx=1635101012000, /CC-Topo-2022.imn=1633862454000, /rfc7231.txt=1635101100000]
[23:46:21] Sender | Files to be requested by the remote node: {}
[23:46:21] Sender | Different files: [/Chapter_3_v8.0.pptx=1635101012000, /CC-To
po-2022.imn=1633862454000, /rfc7231.txt=1635101100000]={}
[23:46:21] Sender | Updating the missing changes...
[23:46:21] SlidingWindow | Received file with compression of 0
[23:46:21] FileOperations | /CC-Topo-2022.imn downloaded in 371ms
[23:46:21] FileOperations | Download speed: 248795mb/s
[23:46:21] SlidingWindow | Received file with compression of 0
[23:46:21] FileOperations | /rfc7231.txt downloaded in 410ms
[23:46:21] FileOperations | Download speed: 221882mb/s
[23:46:21] SlidingWindow | Received file with compression of 1
[23:46:23] FileOperations | /Chapter_3_v8.0.pptx downloaded in 2407ms
[23:46:23] FileOperations | Download speed: 38348mb/s
[23:46:23] Sender | Nodes synchronized

File Edit View Search Terminal Help
[23:46:21] RequestHandler | Timeout: 100
[23:46:21] Get | parseGet fileNames: [/rfc7231.txt]
[23:46:21] Receiver | New connection with address: /10.2.2.1 and port 35283
[23:46:21] RequestHandler | Timeout: 100
[23:46:21] Get | parseGet fileNames: [/CC-Topo-2022.imn]
[23:46:21] Receiver | New connection with address: /10.2.2.1 and port 48378
[23:46:21] RequestHandler | Timeout: 100
[23:46:21] Get | parseGet fileNames: [/Chapter_3_v8.0.pptx]
[23:46:21] SlidingWindow | Sending the amount of packets
[23:46:21] SlidingWindow | File uploaded in 209ms
[23:46:21] SlidingWindow | Sending the amount of packets
[23:46:21] SlidingWindow | Upload speed: 0mb/s
[23:46:21] RequestHandler | Packet handled.
[23:46:21] SlidingWindow | File uploaded in 374ms
[23:46:21] SlidingWindow | Upload speed: 0mb/s
[23:46:21] RequestHandler | Packet handled.
[23:46:22] SlidingWindow | Sending the amount of packets
[23:46:23] SlidingWindow | File uploaded in 1867ms
[23:46:23] SlidingWindow | Upload speed: 3mb/s
[23:46:23] RequestHandler | Packet handled.
[23:46:23] Receiver | New connection with address: /10.2.2.1 and port 56094
[23:46:23] RequestHandler | Timeout: 100
[23:46:23] RequestHandler | Packet handled.
```

Figura 9: Exemplo de um ficheiro de logs em que os nodos se sincronizam.

## 5 Bibliotecas Utilizadas

**org.json.simple:** Biblioteca que oferece as ferramentas necessárias para criar e manipular Objetos JSON.

**javax.crypto:** Biblioteca utilizada para os métodos relacionados a encriptação e decriptação dos dados, bem como o processo de autenticação mútua usado pelo HMAC.

**java.nio.charset:** Biblioteca que define charsets, decoders e encoders que efetuam a tradução entre bytes e caracteres Unicode.

## 6 Conclusão

Ao longo da realização do trabalho prático tornou-se imperativo pôr em prática os conhecimentos adquiridos ao longo da Unidade Curricular de *Comunicações por Computador* de forma a conseguirmos implementar o nosso protocolo que corre sobre sockets UDP e ainda o serviço sobre TCP.

Em suma, o grupo faz um balanço positivo do trabalho realizado, apesar de algumas dificuldades a implementar o protocolo da forma que queríamos, pensamos ter feito um trabalho satisfatório que cumpre na totalidade os requisitos inicialmente propostos e ainda alguns adicionais.