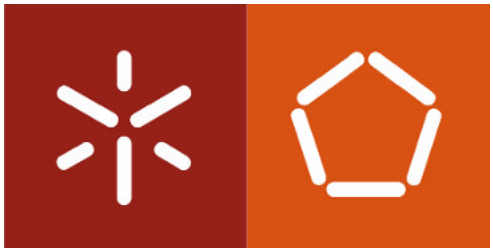

Universidade do Minho
Licenciatura em Engenharia Informática

Comunicações por Computador

TP2 - PL22

André Silva - A87958 João Nunes - A87972

Dezembro 2021

Conteúdo

1 Introdução

No âmbito do trabalho prático 2 da unidade curricular Comunicações por Computador foi pedido o desenvolvimento de uma aplicação de sincronização rápida de pastas, designada por *FolderFastSync* (**FFSync**). Para tal, foi necessária a implementação de todo o sistema pretendido com especial atenção à sincronização dos clientes que se tentam conectar. Essas comunicações entre clientes são todas tratadas através de um protocolo totalmente desenvolvido por nós, o **FTRapid**, que funciona sobre UDP. Além disso, o sistema ainda recebe e trata pedidos HTTP GET, via TCP.

2 Arquitetura da Solução

A arquitetura do projeto desenvolvido baseia-se na demonstração do funcionamento do sistema, quando dois clientes tentam sincronizar pastas e os respetivos ficheiros entre si.

A conexão entre os dois clientes tem de ser iniciada por eles mesmos, ou seja, o primeiro pede a ligação ao IP do outro e através do protocolo **FTRapid** vão ser realizados todos os procedimentos até a conexão ter sido estabelecida. De salientar, que esta ligação apenas é aceite se ambos tiverem a mesma palavra chave, tema que vai ser explicado mais pormenorizadamente nas secções seguintes. Depois de confirmada a conexão, vai ser iniciada a verificação dos ficheiros em falta e por sua vez a transferência dos ficheiros para o respetivo cliente.

Para todo o projeto foram criadas 5 classes, todas elas com papéis importantes para o bom funcionamento do sistema. Para a aplicação em si, **FFSync**, construímos uma classe que trata de todos os procedimentos necessários até alcançar o objetivo final, dedicámos uma classe para os pedidos HTTP GET e, por fim, para o protocolo **FTRapid** decidimos construir três classes (protocolo, pacote e informação do fragmento). De salientar, que a classe do protocolo manda e recebe todos os pacotes enviados pelos clientes, trata da verificação dos ficheiros em falta e da transferência e receção desses mesmos.

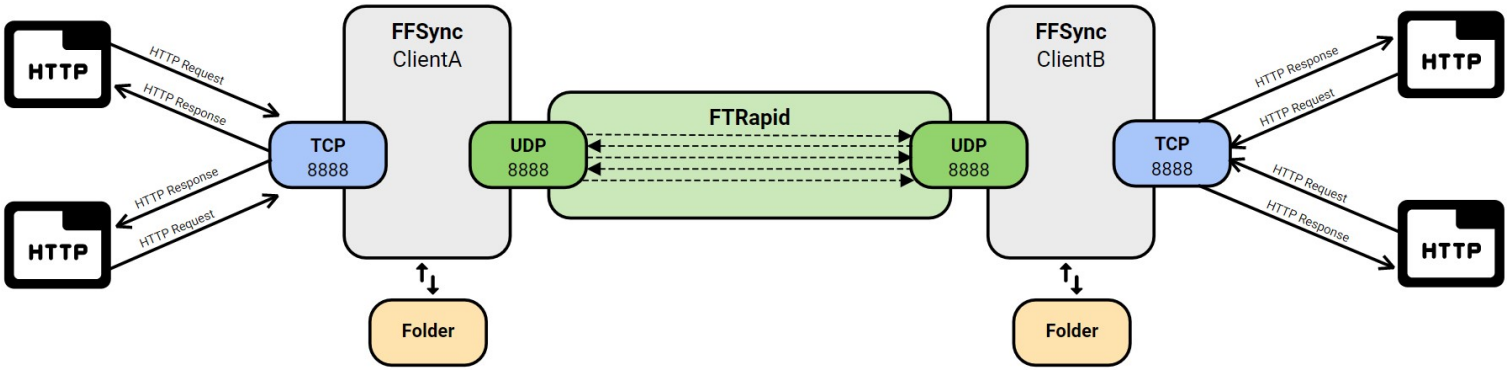

Figura 1. Arquitetura da solução

3 Especificação do Protocolo

3.1 Formato das mensagens protocolares

Cada mensagem trocada entre os clientes é empacotada com recurso à classe **FTRapid_packet** e transmitida no protocolo **FTRapid**, via UDP. Quando a mensagem a ser enviada é um fragmento de um ficheiro, empacotámos toda a informação do fragmento na classe **FTRapid_fragment**, esta que depois vai ser tratada como o conteúdo do pacote geral a ser enviado. Este pacote está formatado para que o protocolo consiga enviar, receber e manipular toda a informação que precisa.

Cada pacote é composto por 4 campos, estes vão ser explicados a seguir, sendo que o *header* possui 3. Para a transferência de ficheiros, utilizámos mais 3 campos que depois vão ser tratados como o conteúdo do pacote reconhecido pelo protocolo.

O tamanho máximo do pacote geral é de 2500 bytes, este que nos garante tamanho mais que suficiente para enviar qualquer mensagem pelo protocolo. Já o tamanho máximo para o fragmento de um ficheiro é de 2048 bytes, número que nos assegura tanto a velocidade como a segurança para o envio de fragmentos através do UDP, este foi decidido depois de vários testes a diferentes ficheiros de diversos tamanhos.

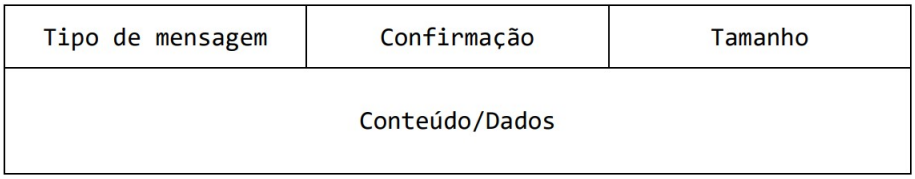

Figura 2. Estrutura do pacote geral, reconhecido pelo protocolo

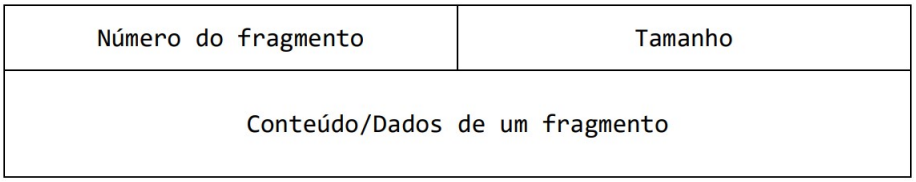

Figura 3. Estrutura do pacote respetivo a um fragmento

Os seguintes campos correspondem à estrutura do pacote geral, reconhecido pelo protocolo e que é usado para enviar e receber as mensagens necessárias entre clientes.

Tipo de mensagem: Inteiro que identifica o tipo de pacote que está a ser enviado, sendo que este pode ser uma destas 8 opções: início de conexão, conexão estabelecida, enviar a lista de ficheiros, pedir pelo ficheiro pretendido, enviar o número de fragmentos, enviar o fragmento, enviar o *acknowledgment* ou acabar a transferência.

Confirmação: Inteiro que indica que recebeu o tipo de mensagem que estava à espera.

Tamanho: Inteiro que indica o tamanho do pacote a ser enviado, incluindo o *header*.

Conteúdo/Dados: *Array* de bytes que guarda os dados que necessitam de ser enviados para o cliente.

Os próximos campos correspondem à estrutura do pacote relativo a um fragmento. Este pacote apenas é utilizado quando é necessária a fragmentação de um ficheiro, sendo depois transformado no conteúdo do pacote descrito acima e enviado para o cliente.

Número de fragmento: Inteiro que identifica o número do fragmento que está a ser enviado.

Tamanho: Inteiro que indica o tamanho do pacote a ser enviado, incluindo o *header*.

Conteúdo/Dados de um fragmento: *Array* de bytes que guarda o fragmento para ser enviado para o cliente.

3.2 Interações com o protocolo

As interações com o protocolo que vão ser explicadas a seguir, representam o caso de dois clientes estarem a tentar sincronizar uma pasta entre si. Destacámos que todas as receções e envios de pacotes através do protocolo são realizadas tendo em conta a probabilidade de perda e de duplicação dos mesmos. Para tal, são sempre verificados os tipos de mensagem e o protocolo espera no mínimo **3 vezes** o tempo de espera do respetivo *socket*.

Início da conexão entre clientes

Quando dois clientes tentam se conectar, cada um vai enviar um pacote com a palavra chave que inseriu. Se ambos os clientes receberem o pacote com a palavra chave igual, então vão enviar outro pacote a confirmar a receção e por fim a ligação vai ser estabelecida. No entanto, se não receberem o pacote vão enviá-lo de novo, até 3 vezes, na tentativa de estabelecer a ligação.

Acrescentámos que é nesta fase que é definida a primeira tarefa que cada cliente vai realizar, pois vai existir sempre um cliente que vai executar o programa primeiro que o outro e dessa forma conseguimos atribuir tarefas diferentes para cada um.

Sincronização de ficheiros

Na tentativa de obterem os ficheiros que estão em falta na pasta atual de cada cliente, é enviado para cada um, a lista de ficheiros da pasta que o outro cliente possui. Depois é feita a análise de quais os ficheiros que não estão na pasta atual, sendo que estes vão ser adicionados à lista de ficheiros em falta. Se por acaso houver algum ficheiro com o mesmo nome, é feita a comparação de datas de modificação e adicionado à lista de ficheiros em falta se o ficheiro do outro cliente estiver sido editado mais recentemente.

Receber ficheiros

Para receber os ficheiros em falta é primeiro enviado ao cliente que vai enviar, o ficheiro pretendido. Depois é recebido o número de fragmentos em que o ficheiro vai ser fragmentado, de forma a termos a certeza que recebemos todos os fragmentos necessários. Por último, são recebidos os fragmentos e enviados os respetivos *acknowledgments*. Caso não seja recebido nenhum fragmento durante o tempo máximo de espera do *socket*, a transferência desse vai acabar.

De salientar, que todos os fragmentos são guardados num Mapa que possui como *key* o número do fragmento e como *value* o conteúdo do mesmo. Se for recebido um fragmento repetido, não vai ser adicionado ao Mapa.

Enviar ficheiros

O envio de ficheiros é iniciado depois de ter sido recebido o pacote que vai identificar o ficheiro desejado pelo outro cliente. De seguida, é realizado o cálculo do número de fragmentos necessários para a transferência do ficheiro. Esse número, como dito na alínea acima, é enviado para o cliente. Por fim, vai ser enviado o fragmento esperando sempre pela receção do *acknowledgment* vindo do cliente. Caso não seja recebido, então o protocolo vai esperar até 5 vezes o tempo máximo de espera do *socket*, intercalando o reenvio do fragmento para o cliente, de modo a prevenir que não sejam enviados todos os fragmentos. No caso especial de nenhum *acknowledgment* ter sido recebido depois das 5 tentativas, o protocolo vai tentar mais 2 vezes o envio do mesmo fragmento, prevenindo que a má ligação entre clientes arruíne a transferência dos ficheiros.

Ainda são guardadas algumas estatísticas como, o número de fragmentos reenviados ou o número de *timeouts* do *socket* à espera de *acknowledgments*.

Acabar com a transferência de ficheiros

Depois do cliente ter recebido todos os ficheiros que tencionava, o protocolo envia para o outro cliente uma mensagem de tipo '10'. Tipo definido pelo protocolo de encerramento da transferência de ficheiros, permitindo a progressão da aplicação.

4 Implementação

A linguagem de programação optada foi **Java**, por ser uma linguagem que o grupo se sente confortável e confiante para a implementação da aplicação pedida. Um ponto também importante nesta decisão foi a extensa biblioteca de *packages* que esta linguagem nos oferece, proporcionando métodos indispensáveis, por exemplo, para a realização das conexões por UDP ou TCP.

4.1 FFSync

Classe principal da nossa aplicação que executa todos os passos até ao objetivo final (sincronização correta das pastas fornecidas pelos clientes). Invoca todos os métodos definidos no protocolo pela ordem correta de execução. Começa por criar o *socket* e disponibilizar uma *thread* para os pedidos HTTP GET, tema que vai ser explicado a seguir. Se a ligação for estabelecida pelo protocolo, vai ser obtida a lista de ficheiros que já possui e, através do protocolo, a lista de ficheiros em falta. De seguida, vai ser verificada qual a primeira tarefa (receção ou envio de ficheiros) a cumprir e vai executá-la. Caso a ligação não seja estabelecida a aplicação vai encerrar.

Autenticação

Introduzimos um mecanismo simples e eficaz que aumenta a camada de segurança na nossa aplicação. Mecanismo esse que utiliza uma palavra chave que apenas os clientes que desejam efetuar a sincronização de pastas devem ter. Por exemplo na ligação entre dois clientes, ambos têm de inserir, ao invocarem o programa, uma palavra chave que será utilizada para validar a ligação entre eles. Se a palavra chave estiver incorreta a ligação não será estabelecida.

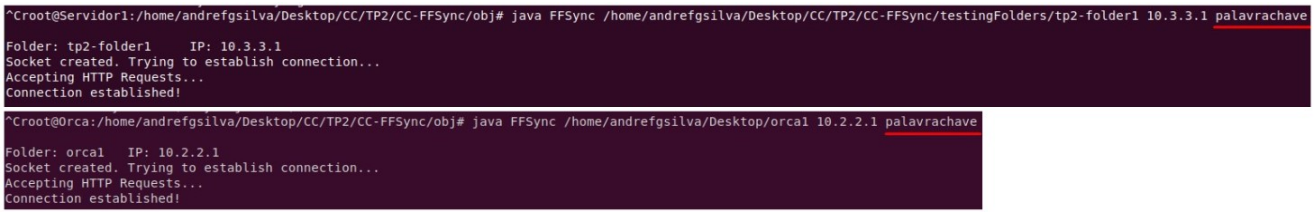

Figura 4. Autenticação de clientes com a mesma palavra chave

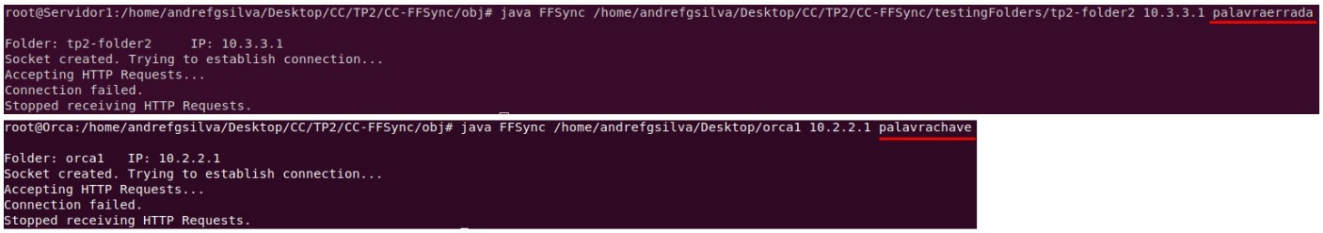

Figura 5. Autenticação de clientes com palavras chaves diferentes

Registo dos passos (*Logs*)

De forma a nunca deixarmos nenhum dos clientes sem saber o que o programa está a fazer, adotámos um sistema de mensagens que são imprimidas no ecrã do cliente. Incluímos, por exemplo, a criação do *socket* do cliente, a lista de ficheiros que tem em falta e o tempo e débito de uma certa transferência de um ficheiro. As mensagens mais relevantes são ainda guardadas numa variável para futuro acesso.

```
Folder: orca1   IP: 10.2.2.1
Socket created. Trying to establish connection...
Accepting HTTP Requests...
Connection established!
```

Figura 6. *Logs* para a conexão de clientes

```
Missing 1 files:  rfc7231.txt

---------- SENDING FILES ----------
No files to send.

---------- RECEIVING FILES --------
File pretended: rfc7231.txt
- Number of fragments: 115
- Received in 0.0s with debit of 1880424bits/s
- Transfer completed.
```

Figura 7. *Logs* para a sincronização e transferência

Envio e receção de ficheiros

É normal durante o envio de ficheiros de tamanhos maiores que ocorra a fragmentação desse ficheiro. E devido à existência de *packet loss* e de duplicados, foi necessária a implementação de um algoritmo para a verificação da chegada das mensagens protocolares com os fragmentos de um ficheiro ao cliente. Tudo isto é tratado, e como na secção anterior é explicado, pelo protocolo.

A classe do FFSync depois da receção, por parte do protocolo, de todos os fragmentos, vai construir o ficheiro através da iteração ordenada do Mapa que contém respetivamente, o número do fragmento e o conteúdo desse.

```
// Writing to file
FileOutputStream fos = new FileOutputStream(fileN);
for (int i = 0; i < fragmentsMap.size(); i++)
    fos.write(fragmentsMap.get(i));
int fileLength = (int) fileN.length();
```

Figura 8. Código relativo à construção de um ficheiro

4.2 FFSync_http

Classe que trata de todos os pedidos HTTP GET. Possui 4 variáveis, o IP do cliente, a tarefa atual do cliente, a lista de ficheiros atuais e uma variável para começar e acabar a receção dos pedidos.

Começa por criar um *socket* TCP, e espera por algum pedido. Depois de receber um pedido, constrói a página html. De salientar que o *socket* tem um tempo máximo de espera pequeno para verificar sempre se é para continuar no ciclo ou não, ou seja,

se é para continuar à espera de receber pedidos ou não. Verificação que é feita na variável descrita no início desta subsecção.

```
// While the socket in the FFSync is open, the http requests is also open
while(this.running) {
    Socket conn = null;
    // Defining a timeout to know when the FFSync socket is closed and the http requests needs to end
    try {
        ss.setSoTimeout(1000);
        conn = ss.accept();
    } catch(SocketTimeoutException e) {}
```

Figura 9. Ciclo que corre até a variável se tornar falsa

```
logs.put(countingLogs++, "Stopped receiving HTTP Requests.");
echo("Stopped receiving HTTP Requests.\n");
obj.setRunning(false);
udpSocket.close();
```

Figura 10. Mudança do valor da variável para falso no final da aplicação

5 Testes e Resultados

5.1 Teste 1

Sincronizar a pasta *tp2-folder1* do **Servidor1** com uma pasta vazia *orca1* no servidor **Orca**.

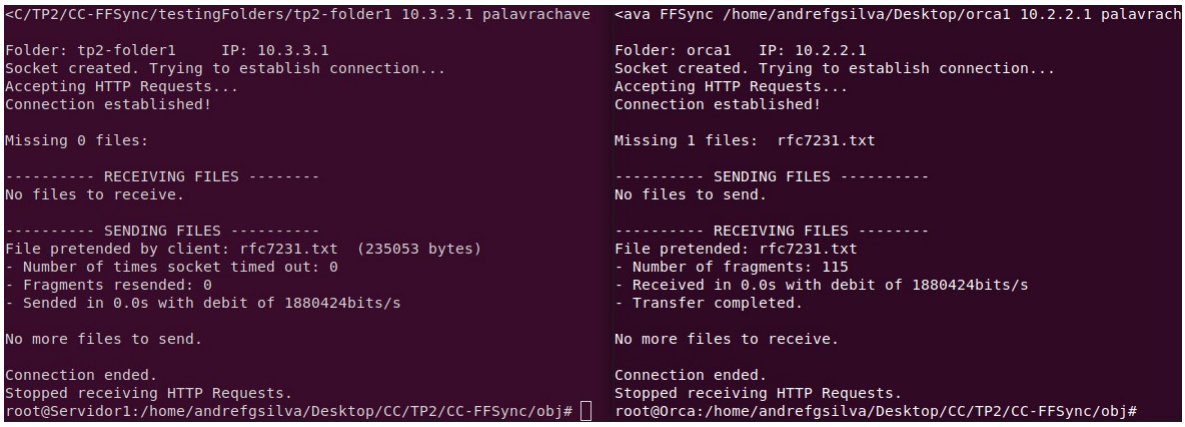

Figura 11. Execução do teste pelo Servidor1 (esquerda) e a Orca (direita)

5.2 Teste 2

Sincronizar a pasta *tp2-folder1* colocada no **Servidor1** com a pasta vazia *grilo1* no portátil **Grilo**.

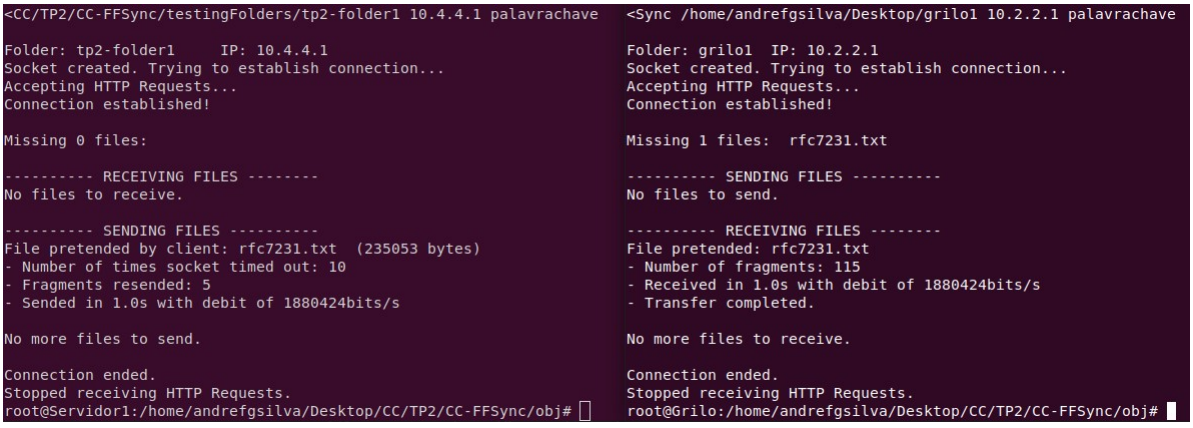

Figura 12. Execução do teste pelo Servidor1 (esquerda) e o Grilo (direita)

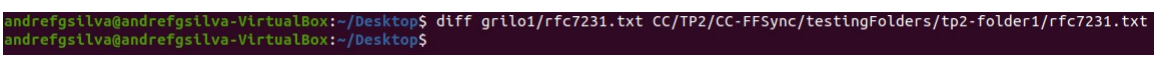

Figura 13. Verificação da igualdade dos ficheiros através do comando *diff*

5.3 Teste 3

Sincronizar a pasta *tp2-folder2* do **Servidor1** e a pasta vazia *orca2* no servidor **Orca**.

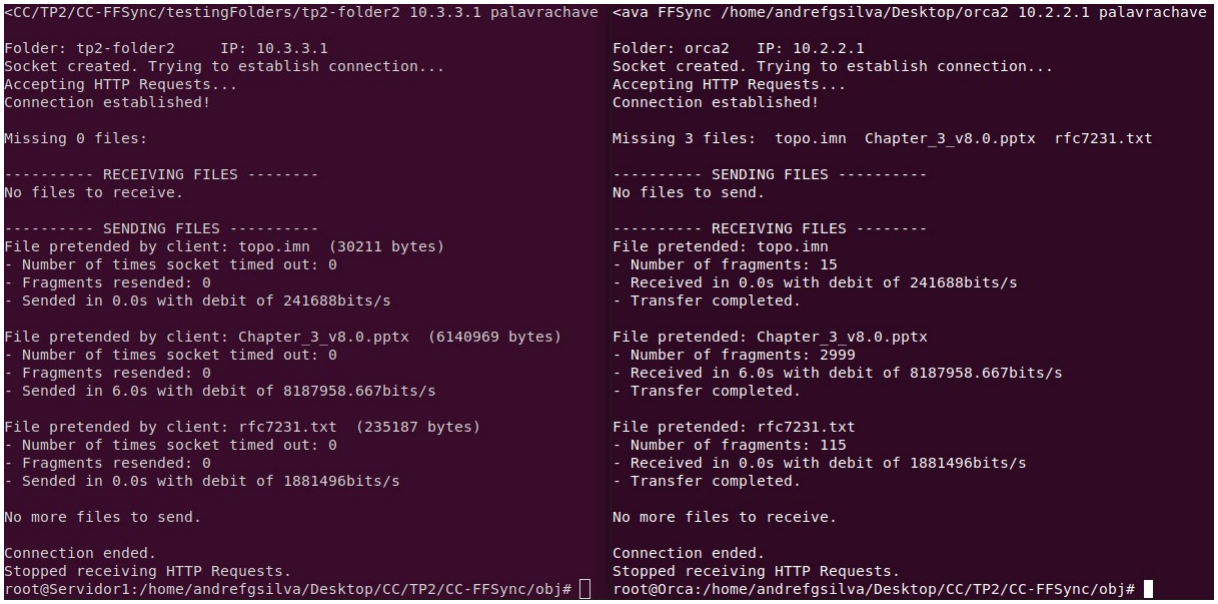

Figura 14. Execução do teste pelo Servidor1 (esquerda) e a Orca (direita)

5.4 Teste 4

Sincronizar a pasta *tp2-folder2* colocada no **Servidor1** com a pasta *tp2-folder3* colocada no servidor **Orca**.

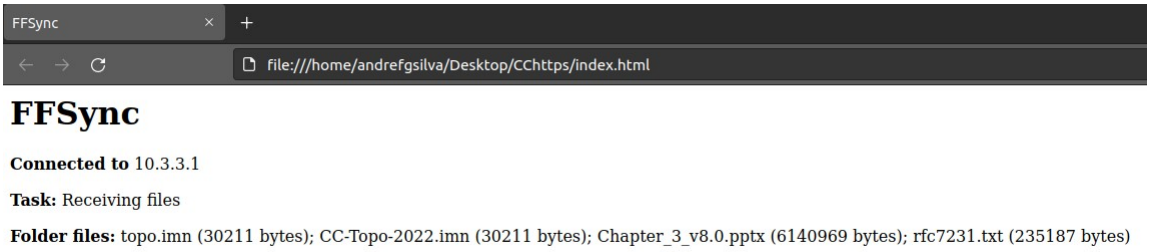

Figura 15. Pedido HTTP GET ao Servidor1

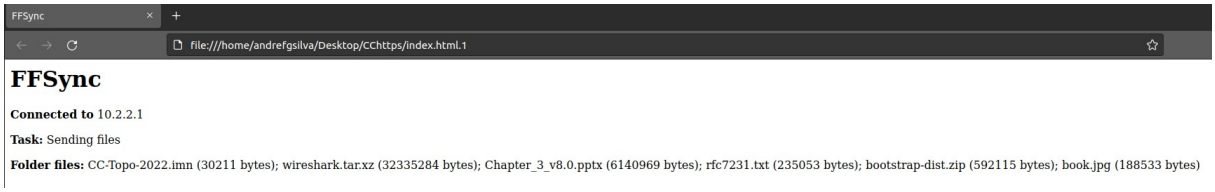

Figura 16. Pedido HTTP GET à Orca

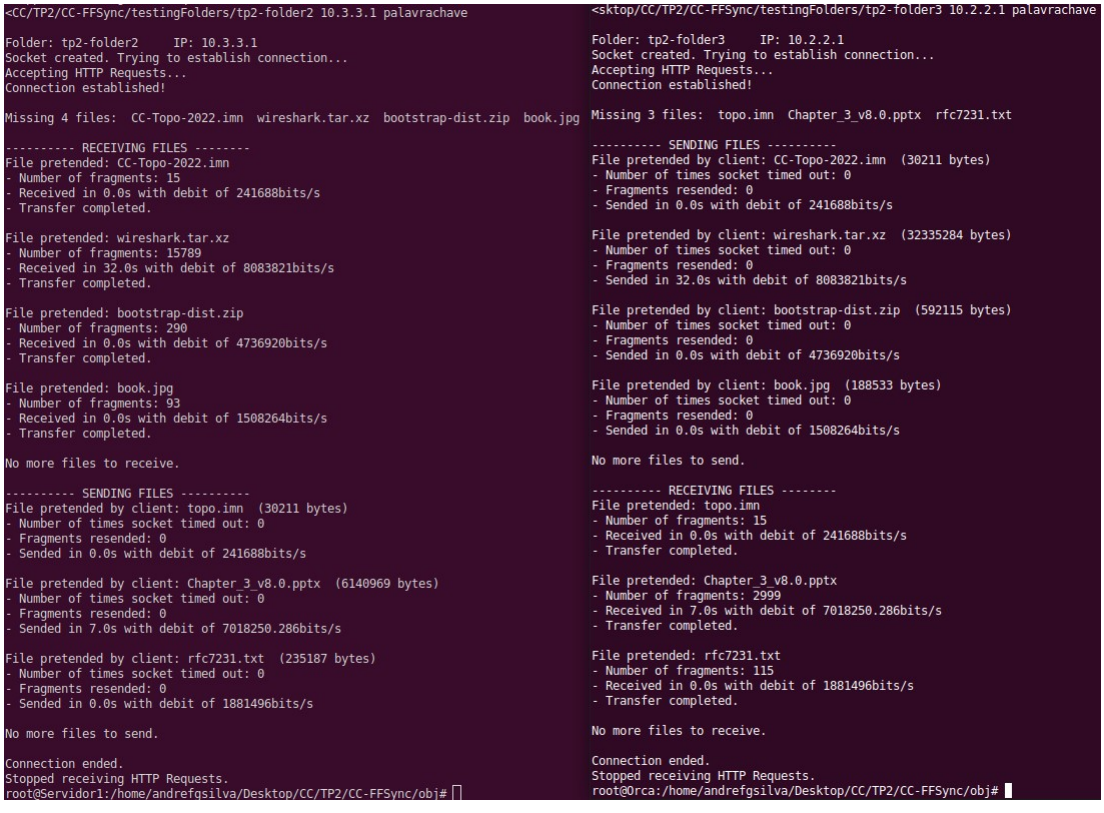

Figura 17. Execução do teste pelo Servidor1 (esquerda) e a Orca (direita)

5.5 Teste 5 - Extra

Sincronizar a pasta *tp2-folder2* colocada no **Servidor1** com a pasta *tp2-folder3* colocada no servidor **Grilo**.

```
<CC/TP2/CC-FFSync/testingFolders/tp2-folder2 10.4.4.1 palavrachave

Folder: tp2-folder2     IP: 10.4.4.1
Socket created. Trying to establish connection...
Accepting HTTP Requests...
Connection established!

Missing 6 files:  CC-Topo-2022.imn  wireshark.tar.xz  Chapter_3_v8.0.pptx  rfc7231.txt  bootstrap-di
st.zip  book.jpg

---------- RECEIVING FILES --------
File pretended: CC-Topo-2022.imn
- Number of fragments: 15
- Received in 0.0s with debit of 241688bits/s
- Transfer completed.

File pretended: wireshark.tar.xz
- Number of fragments: 15789
- Received in 436.0s with debit of 593307.963bits/s
- Transfer completed.

File pretended: Chapter_3_v8.0.pptx
- Number of fragments: 2999
- Received in 84.0s with debit of 584854.19bits/s
- Transfer completed.

File pretended: rfc7231.txt
- Number of fragments: 115
- Received in 3.0s with debit of 626808bits/s
- Transfer completed.

File pretended: bootstrap-dist.zip
- Number of fragments: 290
- Received in 7.0s with debit of 676702.857bits/s
- Transfer completed.

File pretended: book.jpg
- Number of fragments: 93
- Received in 1.0s with debit of 1508264bits/s
- Transfer completed.

No more files to receive.

---------- SENDING FILES ----------
File pretended by client: topo.imn  (30211 bytes)
- Number of times socket timed out: 2
- Fragments resended: 1
- Sended in 0.0s with debit of 241688bits/s

No more files to send.

Connection ended.
Stopped receiving HTTP Requests.
```

```
<ktop/CC/TP2/CC-FFSync/testingFolders/tp2-folder3 10.2.2.1 palavrachave

Folder: tp2-folder3     IP: 10.2.2.1
Socket created. Trying to establish connection...
Accepting HTTP Requests...
Connection established!

Missing 1 files:  topo.imn

---------- SENDING FILES ----------
File pretended by client: CC-Topo-2022.imn  (30211 bytes)
- Number of times socket timed out: 8
- Fragments resended: 4
- Sended in 0.0s with debit of 241688bits/s

File pretended by client: wireshark.tar.xz  (32335284 bytes)
- Number of times socket timed out: 2978
- Fragments resended: 1486
- Sended in 436.0s with debit of 593307.963bits/s

File pretended by client: Chapter_3_v8.0.pptx  (6140969 bytes)
- Number of times socket timed out: 598
- Fragments resended: 297
- Sended in 84.0s with debit of 584854.19bits/s

File pretended by client: rfc7231.txt  (235053 bytes)
- Number of times socket timed out: 27
- Fragments resended: 13
- Sended in 3.0s with debit of 626808bits/s

File pretended by client: bootstrap-dist.zip  (592115 bytes)
- Number of times socket timed out: 55
- Fragments resended: 27
- Sended in 7.0s with debit of 676702.857bits/s

File pretended by client: book.jpg  (188533 bytes)
- Number of times socket timed out: 10
- Fragments resended: 5
- Sended in 1.0s with debit of 1508264bits/s

No more files to send.

---------- RECEIVING FILES --------
File pretended: topo.imn
- Number of fragments: 15
- Received in 0.0s with debit of 241688bits/s
- Transfer completed.

No more files to receive.

Connection ended.
Stopped receiving HTTP Requests.
root@Grilo:/home/andrefgsilva/Desktop/CC/TP2/CC-FFSync/obj#
```

Figura 18. Execução do teste pelo Servidor1 (esquerda) e o Grilo (direita)

```
andrefgsilva@andrefgsilva-VirtualBox:~/Desktop/CC/TP2/CC-FFSync/testingFolders$ diff tp2-folder2/ tp2-folder3/
andrefgsilva@andrefgsilva-VirtualBox:~/Desktop/CC/TP2/CC-FFSync/testingFolders$
```

Figura 19. Verificação da igualdade dos ficheiros através do comando *diff*

6 Conclusões e Trabalho futuro

Com este trabalho concluído, o grupo considera que foi realizado com sucesso, satisfazendo as funcionalidades pedidas no enunciado. Apresentámos a seguir uma análise crítica do trabalho, e as possíveis melhorias para um trabalho futuro.

Considerámos um ponto positivo a autenticação simples implementada que garante a sincronização de ficheiros entre clientes que realmente queiram fazer a sincronização, prevenindo qualquer problema com clientes desconhecidos. A fragmentação foi construída de uma forma que só em casos em que a ligação é extremamente má, poderão haver problemas no envio dos fragmentos. Garantindo assim a transferência de ficheiros que apesar de não ser feita com vários ficheiros em simultâneo, conseguimos que o ficheiro seja enviado e recebido para ligações boas ou para ligações com problemas de *packet loss* e duplicados. Acrescentámos também a obtenção de uma página html simples e objetiva, através dos pedidos HTTP GET, como um ponto positivo no nosso projeto.

Como referimos em cima, não conseguimos implementar a transferência concorrente de ficheiros para um cliente, sendo este talvez o nosso ponto mais negativo no projeto. E que poderia ser realizado numa expansão futura. Também poderia ser adicionada a possibilidade de sincronização entre mais do que um cliente em simultâneo. Funcionalidades estas que nos garantiriam uma excelente melhoria no desempenho e na expansão da aplicação.