# Trabalho Prático 2: FolderFastSync

Joana Alves[a93290] and Jorge Vieira[a84240]

Universidade do Minho
Departamento de Informática
Comunicações por Computador

**Resumo** Neste trabalho, no âmbito da unidade curricular de Comunicações por Computador, pretende-se implementar uma aplicação de sincronização rápida de pastas, designada por ***FFSync*** *(FastFolderSync)*, sem necessitar de servidores nem de conectividade Internet. Para obter a máxima eficiência e velocidade de transferência, a aplicação irá executar a sincronização através de um protocolo baseado em *UDP (User Datagram Protocol)* desenvolvido pelo grupo, titulado de ***FT-Rapid***. No decorrer do mesmo, será também possível obter informações sobre o estado da transferência através de um pedido *HTTP* recorrendo ao protocolo *TCP*.

**Keywords:** UDP · HTTP · TCP · Folder Sync.

## 1 Introdução

Atualmente, a eficiência e fiabilidade dos sistemas são aspetos decisivos no momento de tomada de decisão do cliente. Um sistema super rápido e fiável é, naturalmente, favorecido relativamente a um sistema minimamente eficiente.

Assim sendo, é de extrema importância ter em consideração estes mesmos aspetos no desenvolvimento de *softwares* direcionados, por exemplo, para a área da conexão, pois esta é extremamente competitiva. No entanto, esta área apresenta algumas particularidades no que toca aos protocolos de transporte utilizados, como a incerteza da entrega da informação.

Um dos protocolos mais completos no suporte a perdas de informação e retransmissões é o protocolo orientado à conexão: *TCP (Transmission Control Protocol)*. Este garante um transporte de informação fiável, sem perdas de informação. No entanto, devido a todas as funcionalidades que integra, a sua velocidade de execução pode tornar-se algo desinteressante.

Em contraste, o protocolo *UDP* não é orientado à conexão, o que traz claras vantagens para sistemas de *streaming* ou execução em tempo real, devido à sua rapidez de transmissão.

Com o objetivo de eficiência em mente, foi-nos proposto desenvolver uma aplicação de sincronização rápida de pastas, usando, para isso, o protocolo não orientado à conexão: *UDP*. Neste relatório, vamos abordar as várias fases de conceção da aplicação, assim como a explicação detalhada do comportamento do protocolo ***FT-Rapid***.

## 2 Arquitetura da Solução

Para o desenho da arquitetura da aplicação, utilizamos o *software* ***Draw.io*** para facilitar o esboço do comportamento da solução. Como tal, apresentamos o esquema geral da aplicação onde notamos uma sequência de passos até ao começo da sincronização propriamente dita.

**Verificação de Argumentos:** A aplicação começa por verificar se a pasta fornecida como parâmetro existe na máquina. De seguida. começa a verificação dos *IPs* dos argumentos, testando se o seu formato é válido. Se algum dos parâmetros estiver inválido, a aplicação lança uma exceção e termina a sua execução.

**Thread HTTP:** Após a verificação de argumentos, a aplicação cria uma *thread HTTP* que ficará responsável pela receção de pedidos HTTP (através do protocolo TCP) na porta 80. Após receber pedidos, passa-os à classe auxiliar ***httpWorker*** que responde ao pedido com toda a informação sobre o estado da aplicação armazenada na classe ***InfoTransfer***.

**Thread Recetora:** De seguida, cria uma *thread* Recetora que estará à espera de conexões UDP na porta 80. O seu comportamento será mais detalhado no capítulo *Especificação do Protocolo*, no subcapítulo de *Interações*.

**Thread conexaoClientes:** Através dos *IPs* dados como argumento ao programa, anteriormente validados, são criados N *threads conexaoCliente* responsáveis pela conexão da máquina local com os mesmos. Mais uma vez, o seu comportamento será especificado no capítulo *Especificação do Protocolo*, no subcapítulo de *Interações*.

**Mensagens Protocolares:** Estas classes serão especificadas no capítulo *Especificação do Protocolo*, no subcapítulo de *Formato das Mensagens Protocolares*. São classes auxiliares no envio de mensagens do protocolo, diferindo no seu significado e conteúdo.

**Classes Auxiliares:** Como referido acima, a classe *InfoTransfer* é responsável pelo armazenamento de informação pertinente a ser apresentada na receção de um pedido HTTP. Já a classe *Log* apenas serve para escrever num ficheiro as informações, em tempo real, da sincronização. É criado um ficheiro para cada parceiro de conexão.

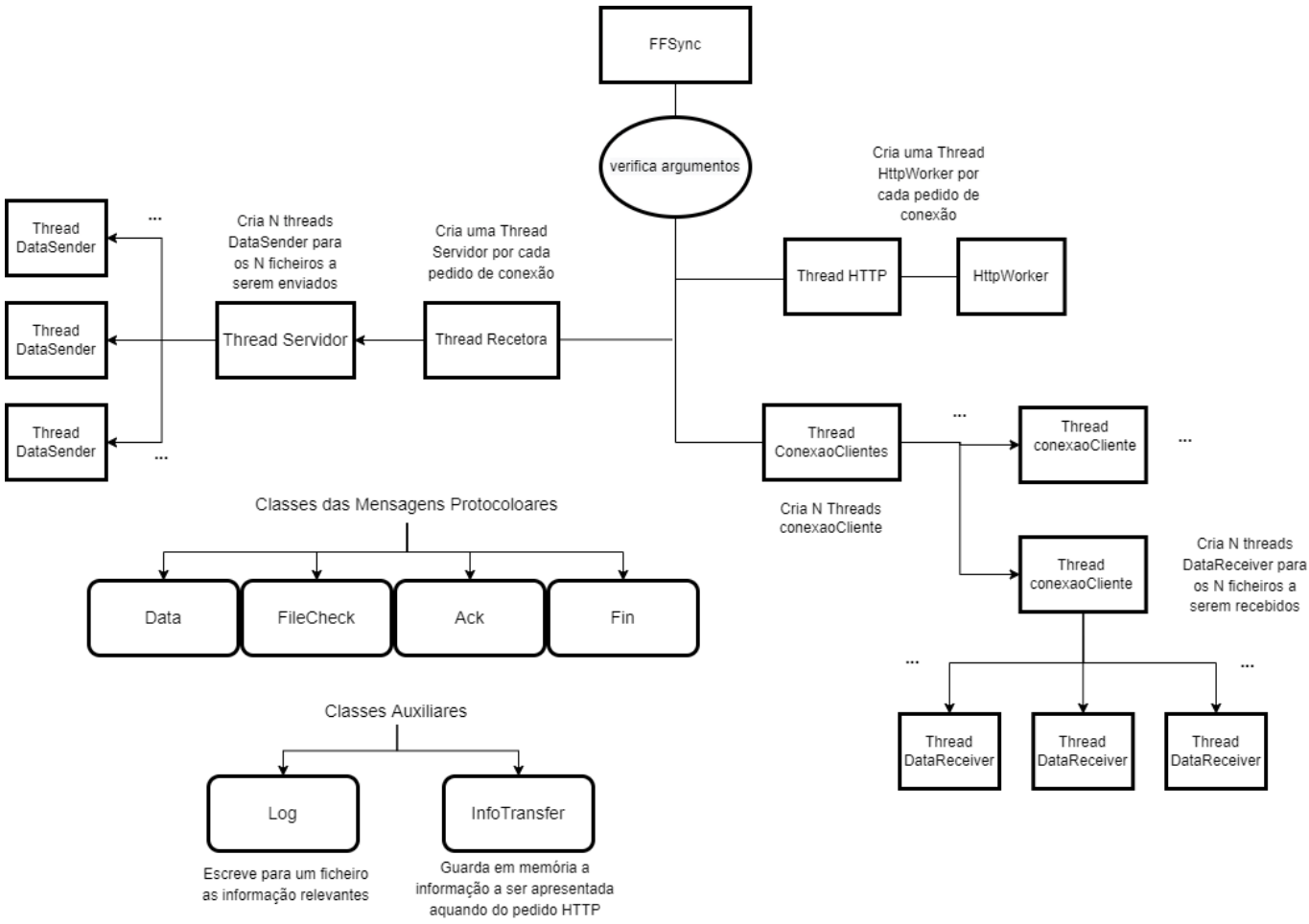



**Fig. 1.** Arquitetura Geral da Aplicação.

## 3 Especificação do Protocolo

### 3.1 Formato das Mensagens Protocolares

Para o desenvolvimento do protocolo, começamos pelos níveis mais baixos, como as mensagens protocolares, para obtermos bons alicerces para a construção da nossa base de informação necessária para o esboço final do protocolo a ser implementado. Como tal, definimos os cabeçalhos das diversas mensagens:

**Cabeçalho Standard** Este cabeçalho é comum a todos os tipos de pacotes criados. Tem um total de 14 *bytes*, sendo 8 destes para o *checksum* e 4 para o número de sequência (*SeqNum*) e diferencia o tipo de pacote com que estamos a lidar. Relativamente ao *byte T* (Tipo), este distinguirá dois tipos de pacotes: **Controlo** ou **Dados**. Já o *byte ST* (SubTipo) faz a distinção entre os vários subtipos de pacotes de controlo ou dados.

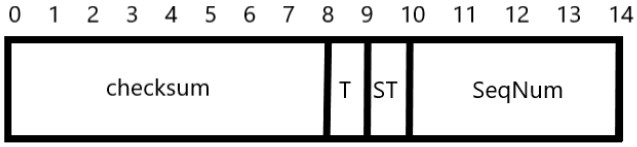
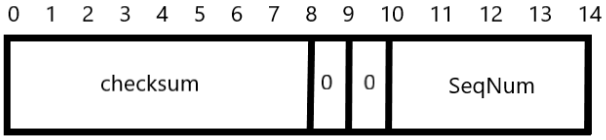
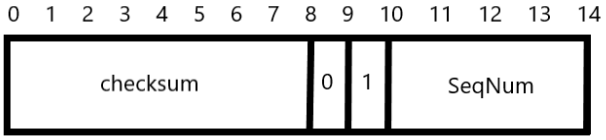



**Fig. 2.** Cabeçalho Standard.

**Pacote de CONTROLO** Este tipo de pacote contém dois subtipos, sendo estes: ***Ack*** e ***Fin***. O subtipo *Ack* apenas serve para informar o parceiro de sincronização da receção dos pacotes de dados através do número de sequência associado. O subtipo *Fin* serve para indicar ao parceiro de sincronização o término da conexão.




Pacote de Controlo [ACK]




Pacote de Controlo [FIN]

**Pacote de DADOS** Este tipo de pacote é o mais complexo, pois é responsável pela serialização de informação. Contém os 14 *bytes* do cabeçalho *standard* (ver Fig. 2), sendo $T$=1. Existem dois subtipos (***Data*** e ***FileCheck***) diferindo em conteúdo, tendo ambos um tamanho máximo de 1024 *bytes*.

O pacote [**DATA**] corresponde ao subtipo $ST$=0. Relativamente ao *Nº Stream*, é um inteiro (4 *bytes*) que identifica o ficheiro ao qual o pacote se refere, já o *byte L* indica se o pacote é o último da *stream*. Assim, ficam alocados 1005 *bytes* para os dados.

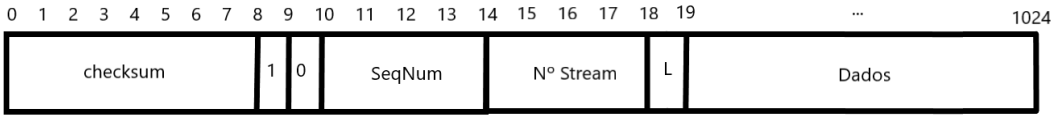



**Fig. 3.** Pacote de Dados [DATA].

O pacote de dados [**FILECHECK**] corresponde ao subtipo $ST$=1. Enquanto o pacote [DATA] envia conteúdo de ficheiros, o pacote [FILECHECK] envia apenas o nome dos ficheiros presentes na pasta alvo de sincronização. Como tal, por conter apenas os dados para além do cabeçalho *standard*, ficam disponíveis 1010 *bytes* para os mesmos.

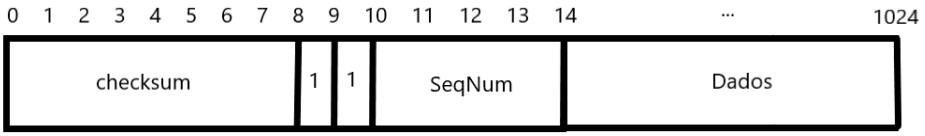



**Fig. 4.** Pacote de Dados [FILECHECK].

### 3.2 Interações

Para o desenvolvimento esquemático do protocolo, usamos como ferramenta auxiliar de esboço, o *software* ***Draw.io*** por ser bastante intuitivo na sua utilização.

Decidimos começar por esboçar um protocolo geral, através de um diagrama temporal, e, de seguida, especificar o seu comportamento nas diversas variantes de acontecimentos através de outros diagramas individuais. Como tal, para permitir uma melhor compreensão do protocolo, apresentamos o diagrama geral:

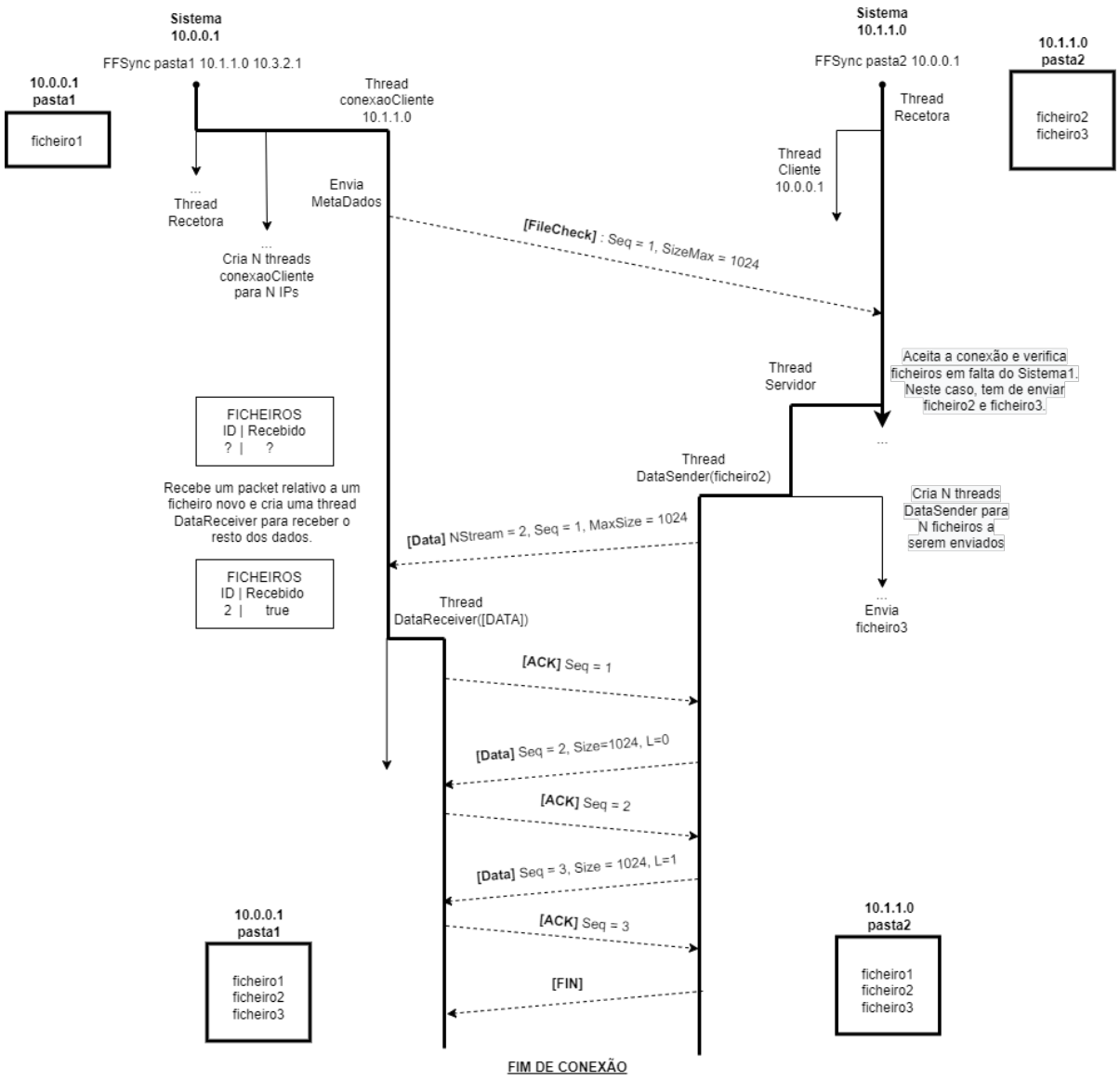



**Fig. 5.** Diagrama Protocolo Geral.

A aplicação é capaz de sincronizar com vários parceiros ao mesmo tempo, usando para cada conexão uma *thread* ***conexaoCliente*** responsável pela comunicação entre máquina local e parceiro de ligação.

Cada *thread* Cliente vai comunicar com uma *thread* ***Servidor*** do lado do parceiro de conexão. Assim, conseguimos atingir uma distribuição de comunicações paralelas necessárias para a multiplicidade de conexões em cada máquina.

***Thread* conexaoCliente:** É criada pelo programa na inicialização do mesmo destinada a ser o elo de comunicação entre a máquina local e os parceiros referenciados nos argumentos da aplicação.

***Thread* Servidor:** É criada pelo programa aquando de uma conexão nova, isto é, quando algum dos sistemas presentes nos argumentos se tenta conectar à máquina local.

**Início de Conexão Cliente:** Após a criação da *thread* Cliente, é feita uma tentativa de conexão com o parceiro enviando um ***packet*** do tipo [FILECHECK]. O pacote vai conter a informação da pasta dada como argumento ao programa. Caso não obtenha resposta por parte do parceiro, entra em modo *sleep* durante 5 segundos repetindo, no máximo, 3 vezes o processo de tentativa de comunicação.

**Início de Conexão Servidor:** Do lado do parceiro, quando recebe o *packet* [FILECHECK], o primeiro passo é, como referido acima, criar uma *thread* de comunicação para a conexão. De seguida, terá duas opções: enviar ficheiros ou não. Se **não precisar** de enviar dados porque o outro sistema já está sincronizado, envia um *packet* [ACK]. Caso contrário, se **precisar** de enviar dados, cria N *threads* ***DataSender*** para os N ficheiros a serem enviados.

***Thread* DataSender:** Esta é responsável pelo envio de um ficheiro em específico, fornecido como argumento pela *thread* Servidor. Começa por enviar um pacote [DATA] contendo apenas o nome do ficheiro, só depois enviando o seu conteúdo e tem o comportamento descrito na figura 7.

***Thread* DataReceiver:** A *thread* Cliente terá, em memória, uma lista de todos os códigos de *stream* dos pacotes [DATA] que tenha recebido até ao momento. Como tal, quando uma *thread* DataSender tenta comunicar com a *thread* Cliente, esta verifica se já recebeu aquele identificador. Se não o contiver na lista, cria uma *thread* DataReceiver para lidar com a comunicação futura do ficheiro e assim assegurar a receção paralela de múltiplos ficheiros.

De seguida, apresentamos os vários diagramas que ilustram o comportamento do protocolo em situações específicas:

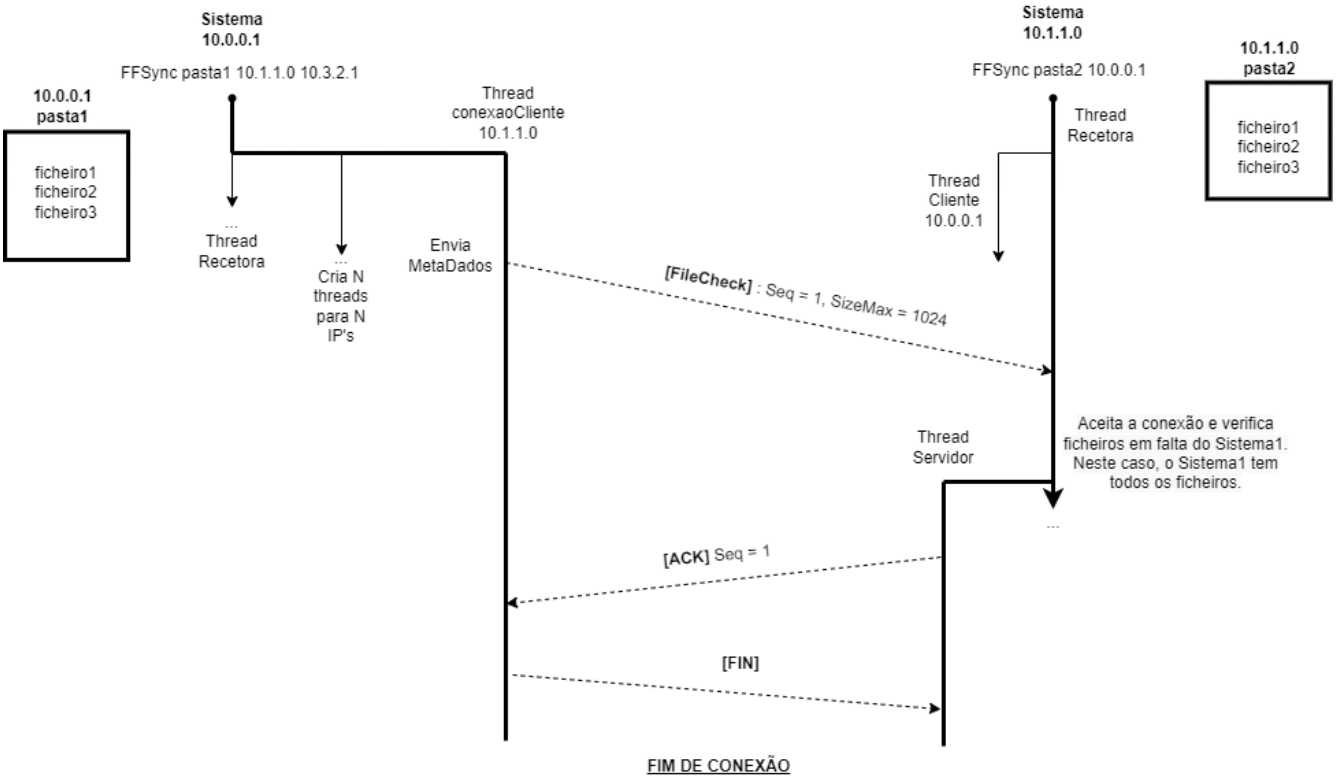



**Fig. 6.** Fluxo Alternativo - a máquina local não tem ficheiros a sincronizar.

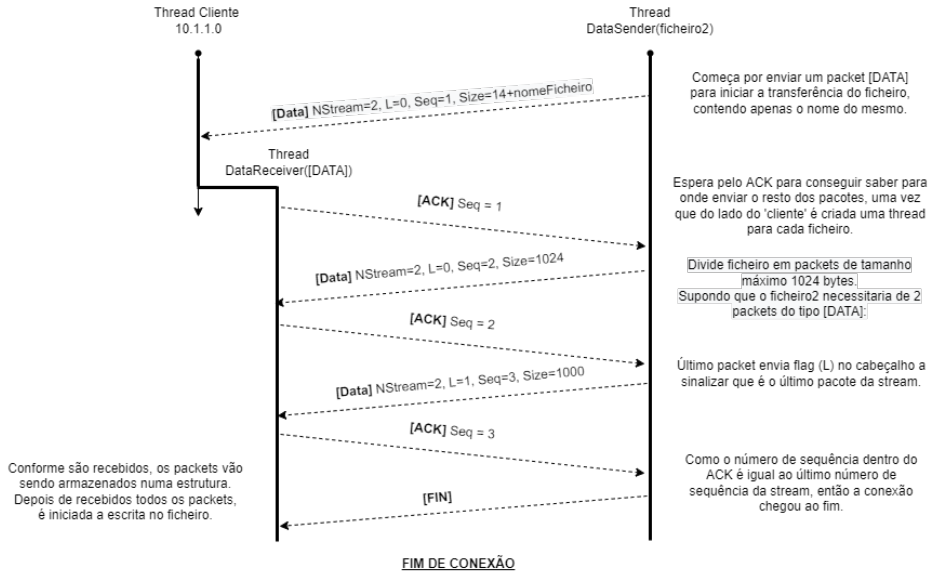



**Fig. 7.** Comportamento das *threads* DataSender e DataReceiver

Observando os esquemas do protocolo, conseguimos perceber que este se comporta à base de ***stop and wait***, não sendo muito eficaz em caso de perdas de pacotes e possíveis retransmissões dos mesmos. Como tal, decidimos evoluir um pouco o seu comportamento teórico, não o tendo aplicado no código da aplicação.

Assim, alteramos o comportamento das *threads* DataSender e DataReceiver. No caso da **DataSender**, esta irá enviar todos os pacotes não esperando pelo [ACK] de resposta. No final, quando enviar o último packet, ficará à espera do [ACK] a confirmar a receção de todos os pacotes. Do lado da **DataReceiver**, vai recebendo os pacotes e armazenando na estrutura. A partir deste momento poderão existir três cenários que serão descritos nas figuras que se seguem: recebe todos os pacotes, não recebe pacotes do meio da *stream* e/ou não recebe o último pacote a marcar o fim da *stream*.

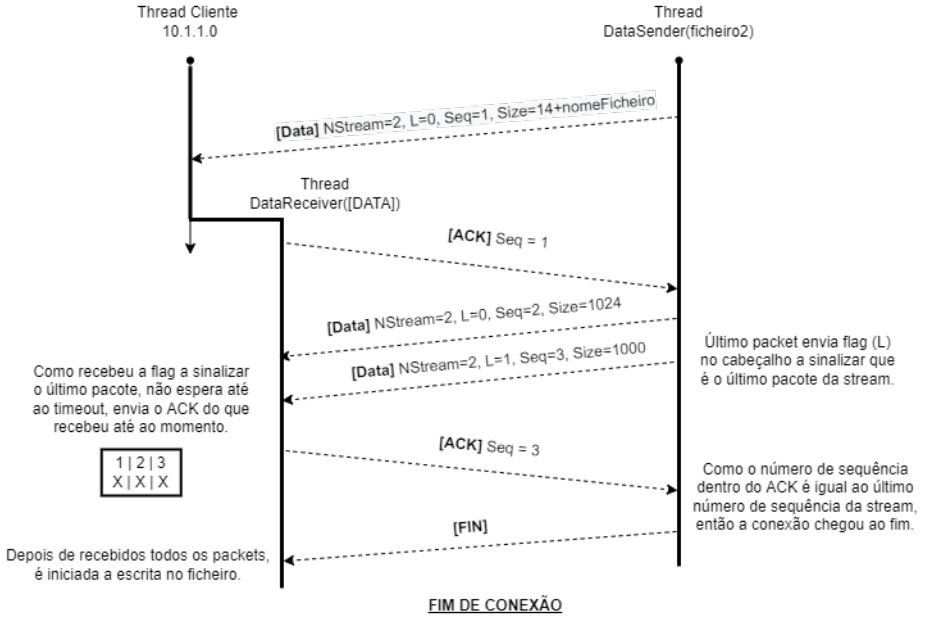



**Fig. 8.** Comportamento DataSender e DataReceiver - comportamento ideal.

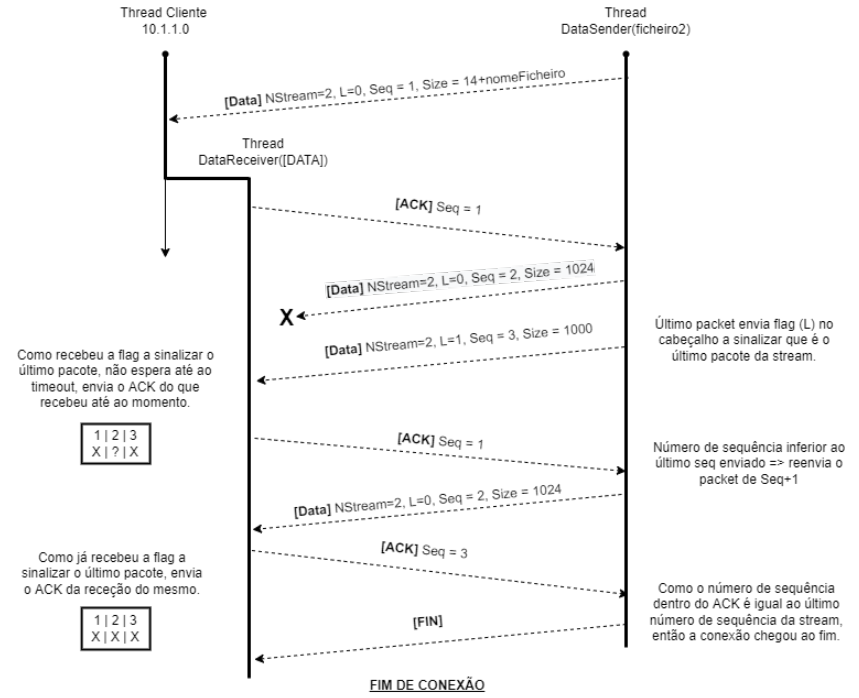



**Fig. 9.** Comportamento DataSender e DataReceiver - não receção de um pacote a meio da *stream*.

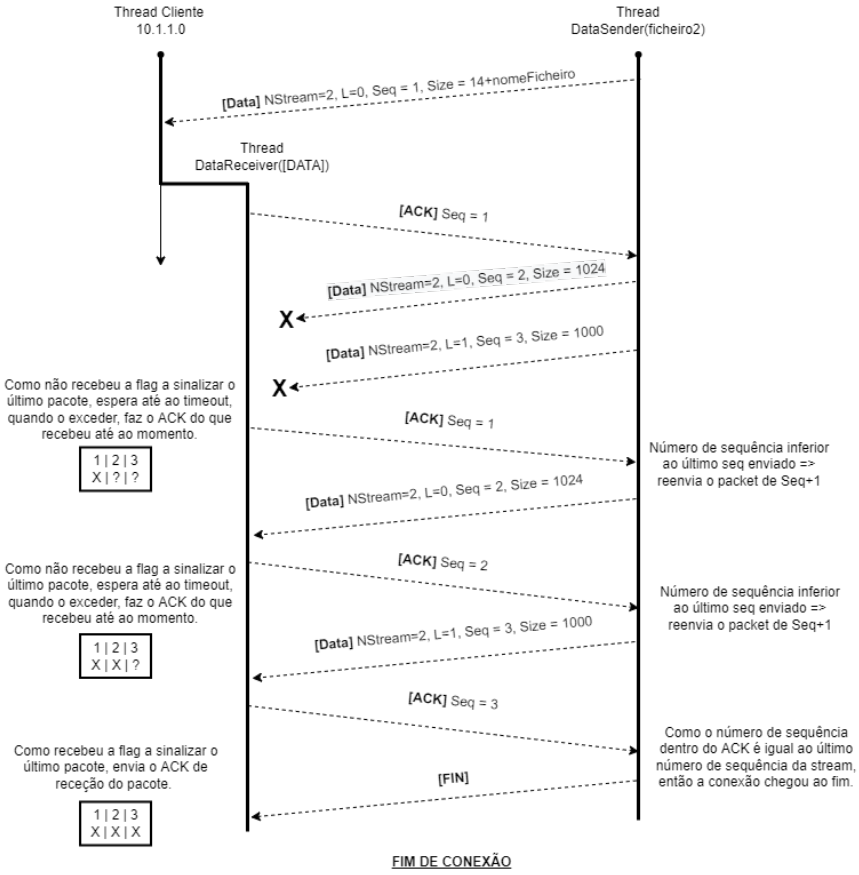



**Fig. 10.** Comportamento DataSender e DataReceiver - não receção do último pacote da *stream*.

Como podemos ver pelos esquemas, este novo protocolo, após rececionar o primeiro [ACK] de resposta, se necessitar de retransmitir pacotes, passa a um comportamento de ***stop and wait***. Assim, do lado da *thread* Cliente, esta faria uma verificação de todos os pacotes recebidos até ao momento e enviaria os pacotes [ACK] necessários, esperando pela resposta da *thread* Servidor. Portanto, a grande diferença entre o protocolo implementado e este evoluído seria o primeiro envio de todos os pacotes, onde não espera por resposta, passando toda a verificação para o "final" da transmissão.

## 4 Implementação

Para o desenvolvimento da aplicação escolhemos a linguagem de programação ***java***, uma vez que é a linguagem com a qual o grupo se sente mais confortável. Para a implementação do código, utilizamos o *software* **IntelliJ IDEA** devido à sua facilidade de utilização e ambiente de desenvolvimento prático.

Para as várias funcionalidades da aplicação foi necessário recorrer a diversas bibliotecas de funções do *java*. Assim, para os vários assuntos apresentamos a biblioteca escolhida.

**Threads:** Como a aplicação opera sobre *threads*, recorremos à biblioteca *java.lang.Thread*.

**Atendimento HTTP:** Para o atendimento de pedidos HTTP, utilizamos a biblioteca de funções *java.net.Socket* pois esta implemente o protocolo de transporte TCP.

**Sincronização:** Para a sincronização das pastas, usamos a biblioteca *java.net.DatagramSocket* uma vez que esta opera sobre o protocolo UDP.

**Packets:** Para a formatação da informação para *packets* recorremos à biblioteca *java.net.DatagramPacket*.

**IPs:** Para a obtenção e manuseamento de *IPs* utilizamos a biblioteca *java.net.InetAddress*.

**Estruturas de Dados:** No decorrer do projeto foram utilizadas diversas estruturas de dados para armazenar os vários tipos de informação. Como tal, foram maioritariamente utilizadas as seguintes bibliotecas: *java.util.Map, java.util.List, java.util.Set*. Para o manuseamento de listas foi necessário recorrer à biblioteca *java.util.Iterator*.

# 5 Testes e Resultados

Para testar o funcionamento da aplicação, utilizamos a topologia *CC-Topo-20-22.imn* fornecida pelos docentes no início do ano letivo. Assim, conseguimos testar a transferência de ficheiros numa simulação do mundo real. Começamos por testar com ficheiros aleatórios, passando de seguida para os ficheiros e cenários de teste aconselhados pelos docentes. Neste capítulo vamos apresentar os resultados dos vários cenários.

## 5.1 Ficheiros Aleatórios

Quando começamos a testar o funcionamento da transferência de ficheiros, testamos com ficheiros relativamente pequenos. Nos primeiros testes efetuamos a transferência de um só ficheiro de uma máquina para outra, evoluindo no número de ficheiros a serem trocados entre as mesmas. Todos os cenários de teste neste subcapítulo são feitos entre o nó **Servidor1** (pasta */home/core/miniTeste/*) e o nó **Golfinho** (pasta */home/core/sync/*).


**Fig. 11.** Resultado do envio do ficheiro 'Relatorio-g77-pl07.pdf' (Servidor1 para Golfinho).

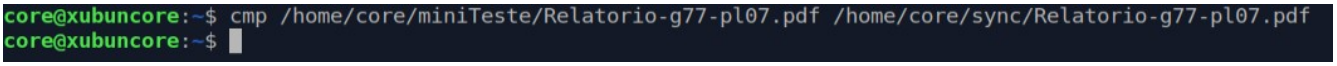



**Fig. 12.** Comparação entre ficheiro original e recebido.

## 5.2 Cenário de Teste (1)

Como obtivemos bons resultados no envio de múltiplos ficheiros concorrentes, passamos então aos cenários de teste definidos pelos docentes, começando pela sincronização da pasta *tp2-folder1* do nó Servidor1 com a pasta vazia *orca1* do nó Orca. A sincronização foi bem-sucedida, obtendo estes resultados:


**Fig. 13.** Resultado do envio do ficheiro 'rfc7231.txt' (Servidor1 para Orca).

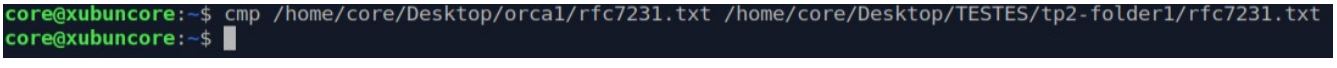



**Fig. 14.** Comparação entre ficheiro original e recebido.

## 5.3 Cenário de Teste (2)

Este cenário de teste visa verificar o tratamento de erros na transmissão de pacotes. Como o nosso programa trabalha sobre um método de **stop and wait**, este, quando existem problemas de receção de pacotes, espera indefinidamente, resultando numa "paragem" infinita. O tratamento de erros seria efetivamente melhor caso implementássemos o protocolo evoluído referido no capítulo de *Especificação do Protocolo*, no subcapítulo *Interações*.

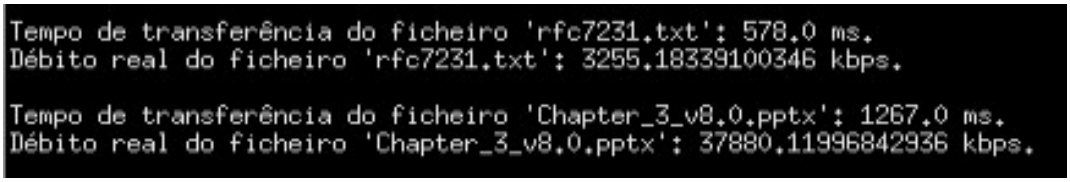



**Fig. 15.** Resultado dos ficheiros recebidos no nó Grilo.

Assim, podemos ver pelos resultados que apenas um dos três ficheiros é recebido na totalidade. O terceiro ficheiro (*topo.imn*), como não foi recebido nenhum packet de resposta a meio da transferência, ficou "retido" numa espera indefinida, não enviando [ACK] de confirmação.

No caso do ficheiro com extensão *pptx*, este não recebeu pacotes do meio da sequência, recebendo, no entanto, o último, o que significa que chegou a escrever no ficheiro mas não a informação completa, notando que a aplicação também não faz controlo do número de sequência dos pacotes, apenas esperando pela receção do último e escrevendo de seguida. O único ficheiro transferido na totalidade foi o *rfc7231.txt* como podemos comprovar pela Fig.16.


**Fig. 16.** Resultado dos ficheiros recebidos no nó Grilo.

### 5.4 Cenário de Teste (3)

Este cenário de teste consiste na sincronização da pasta *tp2-folder2* do nó Servidor1 com uma pasta inicialmente vazia no nó Orca. Neste exemplo é testado o envio concorrente de ficheiros num sentido, neste caso, do nó Servidor1 para o nó Orca. Como não existiram problemas de conexão, a transferência ocorreu sem grandes percalços, terminando com a pasta do nó Orca completamente sincronizada relativamente à pasta do Servidor1.

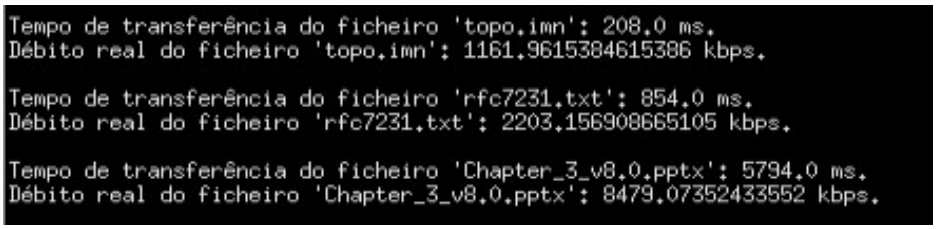



**Fig. 17.** Resultado do envio dos ficheiros (Servidor1 para Orca).

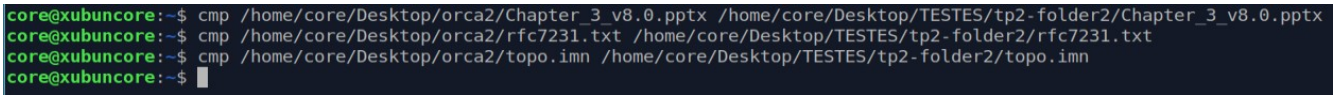



**Fig. 18.** Comparação entre ficheiros originais e recebidos.

Para comprovar que os ficheiros foram recebidos concorrentemente, apresentamos de seguida *prints* do ficheiro de *log* do nó Orca, onde é apresentada toda a informação da transferência sobre a perspetiva do nó que recebe os ficheiros. É de notar, que, apesar de no ficheiro *log* não ser evidente, os ficheiros do lado do Servidor1 foram também enviados concorrentemente através de *threads* DataSender.

```
[SYSTEM]: (11:50:19.378144681) SENT -> [FILECHECK] Size = 14  ;  Seq = 1
[SYSTEM]: Meta-Dados Enviados: []
[SYSTEM]: (11:50:19.578174893) (Thread-6) RECEIVED -> [DATA] Size = 39  ;  Seq = 1  ;  Last=0
[SYSTEM]: (11:50:19.583592848) (Thread-7) RECEIVED -> [DATA] Size = 28  ;  Seq = 1  ;  Last=0
[SYSTEM]: (Thread-7): encarregue do ficheiro 'topo.imn'.
[SYSTEM]: (11:50:19.573807315) (Thread-5) RECEIVED -> [DATA] Size = 31  ;  Seq = 1  ;  Last=0
[SYSTEM]: (Thread-5): encarregue do ficheiro 'rfc7231.txt'.
[SYSTEM]: (Thread-6): encarregue do ficheiro 'Chapter_3_v8.0.pptx'.
```

**Fig. 19.** Registo *log* ilustrando as várias *threads* responsáveis por receber os ficheiros.

### 5.5 Cenário de Teste (4)

Este cenário de teste verifica o envio simultâneo de ficheiros em ambos os sentidos, ou seja, sincronização simultânea da pasta *tp2-folder2* no nó Servidor1 com a pasta *tp2-folder3* no nó Orca. Como a nossa aplicação apenas verifica nomes de ficheiros e não a sua data de criação/modificação, os ficheiros com nome idêntico não são sincronizados, pois o programa assume que são idênticos.

```
Tempo de transferência do ficheiro 'topo.imn': 39.0 ms.
Débito real do ficheiro 'topo.imn': 6197.128205128205 kbps.
```

**Fig. 20.** Resultado do envio dos ficheiros para Orca.

```
Tempo de transferência do ficheiro 'CC-Topo-2022.imn': 111.0 ms.
Débito real do ficheiro 'CC-Topo-2022.imn': 2177.3693693693695 kbps.

Tempo de transferência do ficheiro 'book.jpg': 409.0 ms.
Débito real do ficheiro 'book.jpg': 3687.6870415647923 kbps.

Tempo de transferência do ficheiro 'bootstrap-dist.zip': 907.0 ms.
Débito real do ficheiro 'bootstrap-dist.zip': 5222.6240352811465 kbps.

Tempo de transferência do ficheiro 'wireshark.tar.xz': 12216.0 ms.
Débito real do ficheiro 'wireshark.tar.xz': 21175.69351669941 kbps.
```

**Fig. 21.** Resultado do envio dos ficheiros para Servidor1.

```
core@xubuncore:~$ cmp /home/core/Desktop/orca3/tp2-folder3/book.jpg /home/core/Desktop/servidor1/tp2-folder2/book.jpg
core@xubuncore:~$ cmp /home/core/Desktop/orca3/tp2-folder3/bootstrap-dist.zip /home/core/Desktop/servidor1/tp2-folder2/bootstrap-dist.zip
core@xubuncore:~$ cmp /home/core/Desktop/orca3/tp2-folder3/CC-Topo-2022.imn /home/core/Desktop/servidor1/tp2-folder2/CC-Topo-2022.imn
core@xubuncore:~$ cmp /home/core/Desktop/orca3/tp2-folder3/Chapter_3_v8.0.pptx /home/core/Desktop/servidor1/tp2-folder2/Chapter_3_v8.0.pptx
core@xubuncore:~$ cmp /home/core/Desktop/orca3/tp2-folder3/topo.imn /home/core/Desktop/servidor1/tp2-folder2/topo.imn
core@xubuncore:~$ cmp /home/core/Desktop/orca3/tp2-folder3/wireshark.tar.xz /home/core/Desktop/servidor1/tp2-folder2/wireshark.tar.xz
core@xubuncore:~$ cmp /home/core/Desktop/orca3/tp2-folder3/rfc7231.txt /home/core/Desktop/servidor1/tp2-folder2/rfc7231.txt
/home/core/Desktop/orca3/tp2-folder3/rfc7231.txt /home/core/Desktop/servidor1/tp2-folder2/rfc7231.txt differ: byte 457, line 16
core@xubuncore:~$
```

**Fig. 22.** Comparação entre ficheiros recebidos e enviados.

Como podemos observar pelos resultados do comando *cmp*, o ficheiro cujo nome existia em ambos os nós (*rfc7231.txt*) não foi sincronizado, resultado numa diferença de conteúdo.

## 6 Conclusões e Trabalho Futuro

Este trabalho proporcionou, sem dúvida, uma consolidação de conhecimentos sobre protocolos de aplicação e transporte, permitindo ao grupo obter uma nova visão sobre os mesmos.

Este projeto dividiu-se maioritariamente em duas fases: esboço do protocolo e implementação do mesmo. Relativamente ao **esboço do protocolo**, pensamos ter desenvolvido um relativamente funcional, estando cientes dos problemas inerentes à possível retransmissão de pacotes, tendo estes sido atenuados com a evolução esboçada a posteriori.

No que toca à **implementação**, esta não foi desenvolvida na sua totalidade, faltando alguns aspetos definidos no esboço do protocolo como: definição de *timeouts* e suporte a erros de transmissão. Para além disto, relativamente aos pontos obrigatórios, ficou em falta a segurança pois não implementamos um sistema de autenticação mútua, apenas tendo acrescentado uma funcionalidade que restringe as tentativas de conexão àquelas especificadas nos argumentos do programa. Apesar de não termos desenvolvido o ponto da autenticação, implementamos a funcionalidade extra de conexão entre múltiplos parceiros, isto é, a mesma máquina conseguir conectar-se a vários *IPs* concorrentemente.

Em termos de **trabalho futuro**, concluímos que planear um protocolo com antecedência e cuidado é deveras importante devido à facilidade de implementação do código seguindo um plano já traçado.

Em suma, todo o planeamento e desenvolvimento deste projeto foi útil e cativante, proporcionando uma boa abordagem à área da conexão e protocolos de transporte.