FolderFastSync: Sincronização rápida de pastas em ambiente serverless

Rúben Santos, João Martins, Jorge Lima
Licenciatura em Engenharia Informática
Universidade do Minho
Braga, 2021/2022

Abstract

O FolderFastSync (FFSync) surge no âmbito da Unidade Curricular de Comunicações por Computador. Esta é uma aplicação de sincronização rápida de pastas que não necessita de servidores nem de conectividade Internet para funcionar. O FFSync utiliza TCP para receber pedidos HTTP (estado da aplicação) e sincroniza as pastas sobre UDP (FT-Rapid). A linguagem de programação escolhida para implementar o sistema foi o Java.

Arquitetura da solução

De modo a resolver o problema proposto decidimos dividir o sistema nos seguintes *packages*:

1. *Main* (Inicia o FFSync).
2. *UI* (Interpretador).
3. *HTTP* (Sistema de logs).
4. *Listener* (Recebe pedidos de início de conexão).
5. SyncHandler (Gere pedidos de início de conexão: fase pré-guide).
6. *Transfers* (Gere fase pós-guide).
7. *FTRapid* (Envia/Recebe dados).
8. HistoryRecorder (Formato do sistema de logs).
9. Logs (Gerar estado do sistema).

O diagrama abaixo, é um diagrama de packages inspirado no princípio: The Acyclic Dependency Principle [Martin].
Este diagrama vem oferecer uma visão geral da organização do sistema e representa as dependências entre os diferentes *packages*.

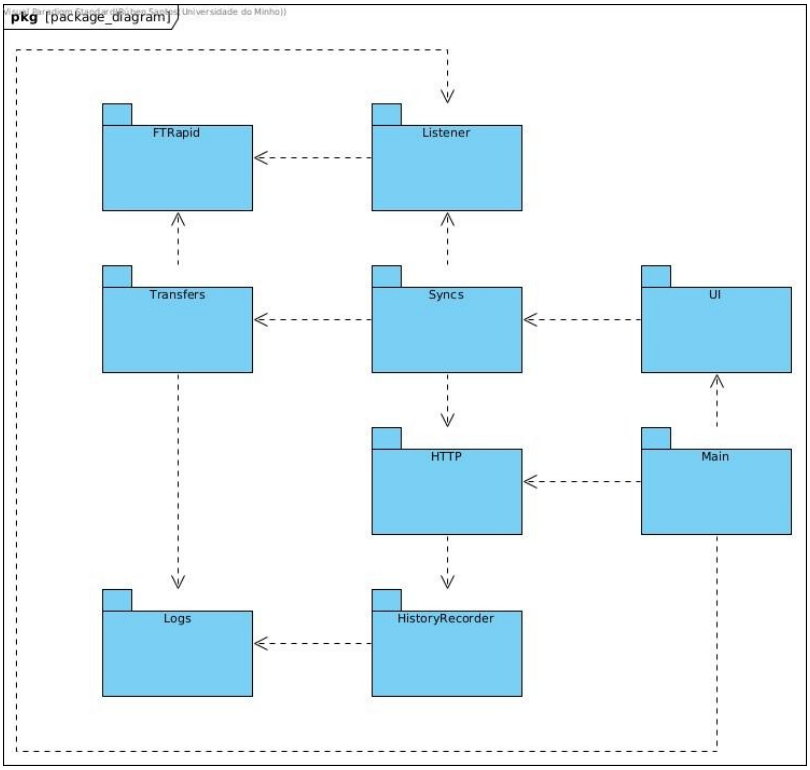

Figure 1: Diagrama de Packages (acíclico)

Especificação do Protocolo

Propósito

O *FT-Rapid* é um protocolo de sincronização de ficheiros *peer-to-peer* que utiliza UDP. Este protocolo foi desenhado com o único propósito de manter duas pastas, em dois *end systems* diferentes, sincronizadas, ou seja, com o mesmo conteúdo. Dado o foco do protocolo, o *FT-Rapid* foi desenhado para ser rápido e, acima de tudo, viável.

Visão Geral

Podemos identificar 3 momentos principais do fluxo do protocolo.

1. Uma fase inicial, *pré-rotule*, onde os pares chegam a acordo em relação a quem é o *peer superior* e quem é o *peer inferior*.
2. Uma fase *pré-Guide*, onde o sistema tenta chegar a acordo com o outro par acerca do ficheiro *Guide*.
3. Uma fase *pós-Guide*, onde os *peers* seguem o conteúdo do *Guide* acordado, ou seja, transferem os ficheiros que tem a transferir.

Fase 1: *pré-rotule*

Dada a natureza P2P deste protocolo, é necessária uma forma de ordenar os 2 pares, para que possa existir uma ordem de ideias, p. ex., qual é o par responsável por fazer o quê e quando. Desta forma, um dos pares assume o papel de "inferior" e o outro de "superior". Estes papeis são determinados de forma **aleatória**.

1. Cada sincronização começa com o envio de um pacote *INIT* para outro par.
   a) Este pacote é enviado para uma porta pré-definida (8000), onde são recebidos pedidos UDP e TCP. Chamamos a este componente do nosso protocolo, *Listener*.
   b) O *Listener* recebe, em UDP, pacotes *INIT* e *INIT_ACK*.
2. Após o envio do pacote *INIT*, é verificado o *buffer* de pacotes recebidos pelo *Listener*. Neste *buffer* podemos encontrar pacotes *INIT* e *INIT_ACK*.
   a) Caso encontrarmos um pacote *INIT*
      - Existe um par que pretende sincronizar uma pasta connosco e portanto temos de verificar se se trata da pasta que estamos a sincronizar de momento (comparar *filename*).
      - Caso seja a pasta que estamos a tentar sincronizar, vamos ler o inteiro aleatório que vem no pacote.
      - Caso o inteiro seja maior do que aquele que nós geramos, e enviamos anteriormente, passamos para o passo ***handle_inferior_random***.
      - Caso o inteiro seja menor do que aquele que nós geramos, e enviamos anteriormente, ignoramos o pacote.
      - Caso o inteiro seja igual ao que nós geramos, a sincronização é abortada.
   b) Caso encontrarmos um pacote *INIT_ACK*
      - Existe um par que está a tentar sincronizar uma pasta connosco e que tem um número aleatório inferior ao nosso.
      - Verificamos o conteúdo do pactote: se a pasta é a que estamos a sincronizar, passamos para o passo ***handle_superior_random***.

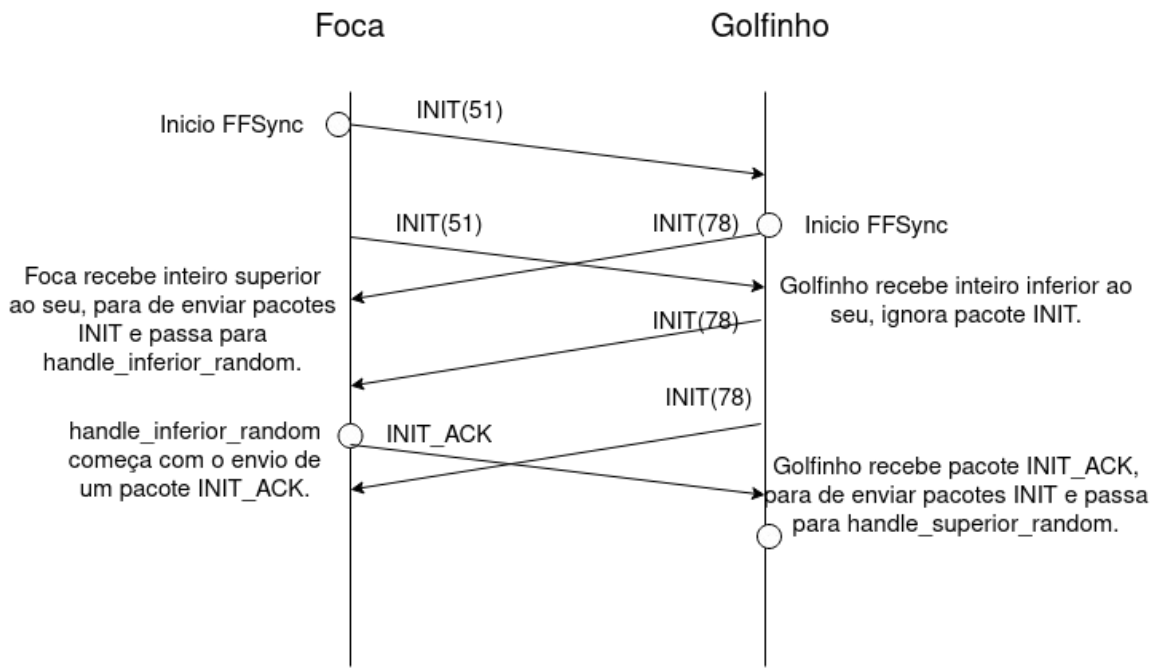

Figure 2: Diagrama Fase 1

Notas Diagrama: Ambos os pares enviam pacotes INIT até que recebam um pacote INIT com um número superior ou um pacote INIT_ACK. Os métodos mencionados no diagrama estão detalhados abaixo.

Fase 2: *pré-Guide*

Nesta fase os *peers* tem como objetivo chegar a acordo de um *Guide*, que é essencial para a fase de transferência de ficheiros.

1) ***handle_inferior_random***
    a. Vamos calcular os nossos *Logs*. Os *Logs* são um conjunto de triplos da forma <*filename*, *timestamps, checksum*>. Os timestamps marcam a hora a que os ficheiros foram modificados e o *checksum* é o resultado de aplicar o algoritmo CRC-32 ao respetivo ficheiro.
    b. Vamos enviar um pacote *INIT_ACK* para a porta do *Listener* do *peer superior* (8000) e esperar por um pacote *ACK*. Este pacote *ACK*, enviado pelo *peer superior*, é dirigido, não ao nosso *Listener,* mas sim à porta pela qual enviamos o *INIT_ACK* anterior. Ou seja, é recebido dentro do *handle_inferior_random*.
    c. Esta troca de pacotes é feita sequencialmente, sendo que esperamos pelo *ACK* do *peer superior* para seguir ao próximo passo. Caso o *ACK* não seja recebido, é enviado novamente um *INIT_ACK* e voltamos a esperar.
    d. Agora que ambos os pares estão prontos para começar a trocar informação mais relevante, vamos enviar os *Logs* para a porta do *peer superior* em

handle_superior_random, que obtemos a partir do *ACK* recebido anteriormente.

e. Após termos enviado os *Logs* com sucesso, vamos esperar pelo *Guide*.
   i. O *Guide* é calculado pelo par superior e consiste na lista de passos necessários para que ambas as pastas fiquem com o mesmo conteúdo.
   ii. Cada passo indica que ficheiro irá ser enviado e quem o vai enviar.

2. ***handle_superior_random***
   a. Após termos recebido um pacote *INIT_ACK*, vamos enviar um pacote *ACK* para a porta do *peer inferior* em *handle_inferior_random*, obtida a partir do pacote *INIT_ACK*.
   b. Depois de enviarmos o *ACK*, verificamos se o *peer inferior* está a enviar *Logs.* Caso não esteja (*timeout*), voltamos a enviar o pacote *ACK*.
   c. Recebemos os *Logs* do *peer* inferior.
   d. Calculamos o *Guide*, partindo dos nossos *Logs* e dos *Logs* recebidos.
   e. Enviamos o *Guide* para o par inferior.

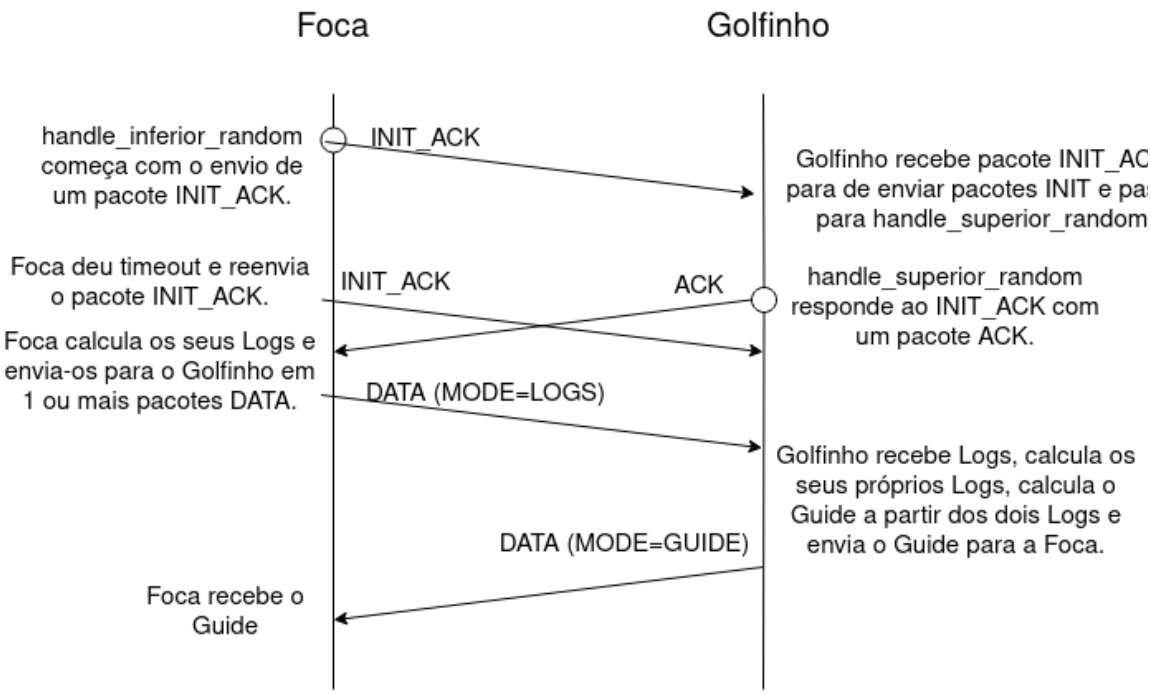

Figure 3: Diagrama Fase 2

Notas diagrama: O envio dos LOGS pode necessitar de mais do que um pacote DATA, sendo que o envio de Logs segue o método descrito em FTRapid-Data e não o que é apresentado neste diagrama. O diagrama está, portanto, simplificado.

Fase 3: *pós-Guide*

A partir deste ponto, cada *peer* tem acesso ao *Guide*, que contém a lista de ações a tomar para que ambas as pastas, nos dois pares, fiquem com o mesmo conteúdo. Ambos vão necessitar de iniciar os processos de transferência de todos os ficheiros.
Cada ficheiro a ser transferido vai ter direito a uma *thread*, sendo que existe um limite no número máximo de *threads* por sincronização (de momento 5 *threads* por sincronização).

Cada *peer* vai se encarregar de pedir os ficheiros que precisa e ouvir os pedidos do outro *peer*. Como é necessário fazer isto de maneira concorrente, vamos ter duas *threads*:

1. A *thread principal*, lê o *Guide* e pede ao outro *peer* os ficheiros de que necessita, criando uma thread para os receber.
2. A *thread listener*, que ouve pedidos, cria uma *thread* para responder aos pedidos de ficheiros por parte do outro *peer*. Só vai responder aos pedidos se o ficheiro requerido pelo outro *peer* constar na lista de pedidos que tem a receber. Caso o pedido seja validado, é criada uma thread para enviar o ficheiro.

Para aumentar a eficiência do sistema implementamos uma espera passiva em ambas as *threads*. A *thread principal*, assim que atingir o número máximo de *threads* disponíveis, vai esperar até que uma das transferências acabe. A *thread listener* vai *adormecer* sempre que a lista de pedidos que tem a receber está vazia. Todos os pedidos que receber nesta altura são considerados inválidos.

Para cada ficheiro, a *thread principal* vai percorrer o ciclo descrito abaixo:
thread principal:

1. Analisar o próximo ficheiro
    1. Se tem que receber o ficheiro, vai criar a *thread* que vai fazer a transferência.
    2. Se tem que enviar o ficheiro, vai incluí-lo na lista de ficheiros a receber e envia um sinal (*signalAll()*) para a *thread listener* acordar.
2. Verificar se o número de *threads* máximo foi atingido e, caso sim ficar à espera de um sinal que é dado sempre que uma *thread* de transferência (envio ou receção) acaba.
3. Voltar ao passo 1.

A *thread listener* vai percorrer o ciclo que está descrito abaixo:
thread listener:

1. Se não existem ficheiros sobre os quais está à espera de receber pedidos, espera por um sinal (enviado pela *thread principal*).
2. Aguarda por receber pedidos
    a) Se o pedido é valido (pacote *INIT_ACK* contém o filename esperado) inicia a *thread* que vai responder (enviar o ficheiro).

b) Se o pedido não é valido volta ao passo 2

3. Volta ao passo 1

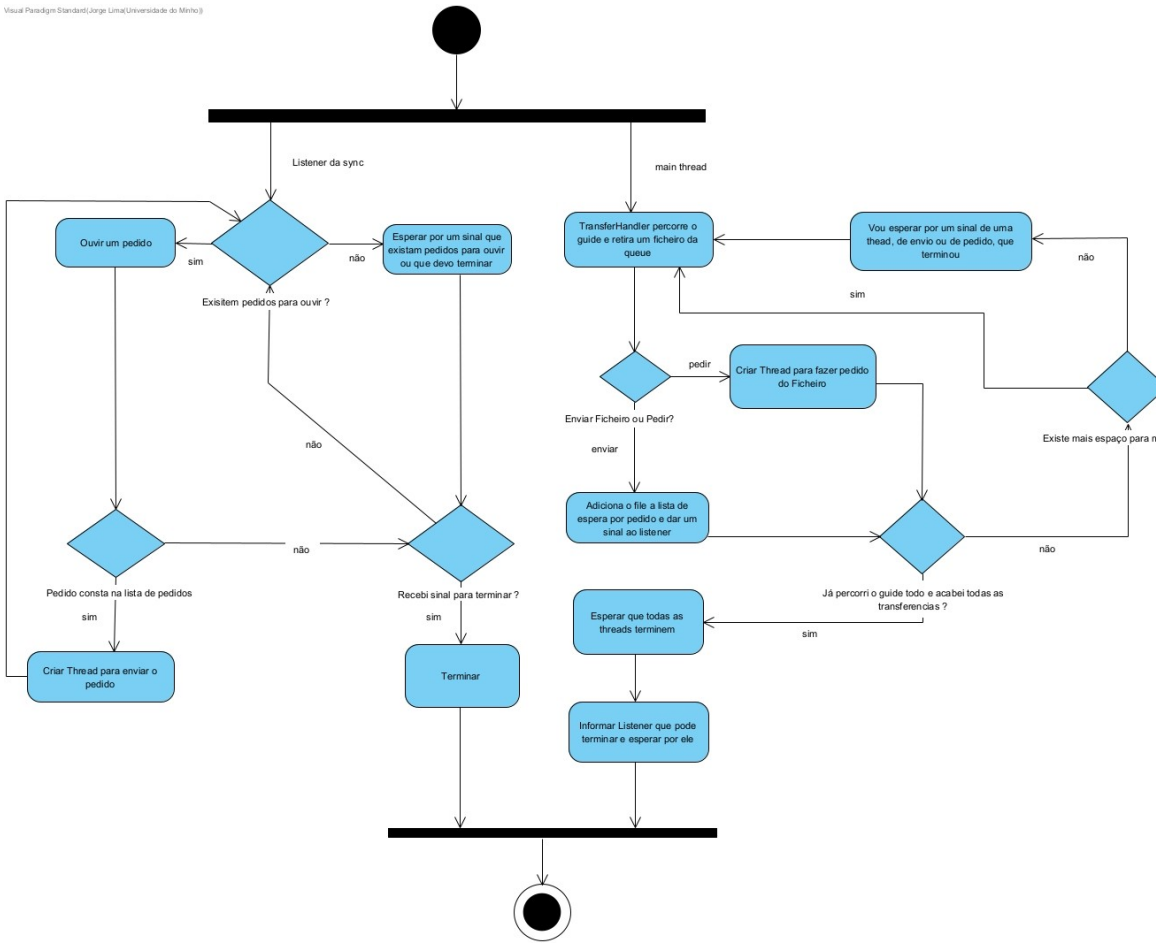

Figure 4: Diagrama de Atividade (Fase 3: Pós-Guide)

Nota Diagrama: Este diagrama surge para tentar simplificar a descrição textual apresentada acima. O diagrama tenta representar o comportamento do sistema numa fase *pós-Guide*, onde são criadas várias *threads* para receber e enviar ficheiros.

Quando a *thread principal* acabar de percorrer o *Guide*, envia um sinal à *theard listener* para terminar. Por fim, espera que todas as threads de envio/receção de ficheiros acabem e dá o processo como terminado.
O processo de envio/receção de dados é explicado em **FT-Rapid-Data**.

FT-Rapid-Data

O FT-Rapid-Data lida com a parte de enviar *Logs, Guides* e *Files*. Desta forma, um pacote *DATA* pode conter uma destes três tipos de informação diferentes.

O FT-Rapid-Data começa com o envio de um pacote *META*. Este pacote serve para indicar ao par de destino: o tipo de ficheiro que vai ser transferido (*Log, Guide* ou *File*), o seu tamanho e, caso estejamos a enviar um ficheiro, o seu nome.
Após a receção de um pacote *META*, é enviado um pacote *ACK*, caso aceitemos receber o ficheiro.
Após esta primeira troca de informação de controlo, o ficheiro a ser enviado é dividido nos respetivos pacotes *DATA*. O primeiro pacote, com número de sequência 0, é enviado. Antes de enviarmos o segundo pacote, com número de sequência 1, esperamos por um pacote ACK (***0@0***). Novamente, antes de enviarmos o terceiro pacote, com número de sequência 0, esperamos por um pacote ACK (***0@1***), e assim sucessivamente.
Por cada pacote *DATA* enviado, esperamos um determinado tempo por um pacote ACK, antes de enviarmos o pacote *DATA* novamente. O mesmo acontece com o envio de pacotes ACK, por parte do *peer* que recebe os dados.

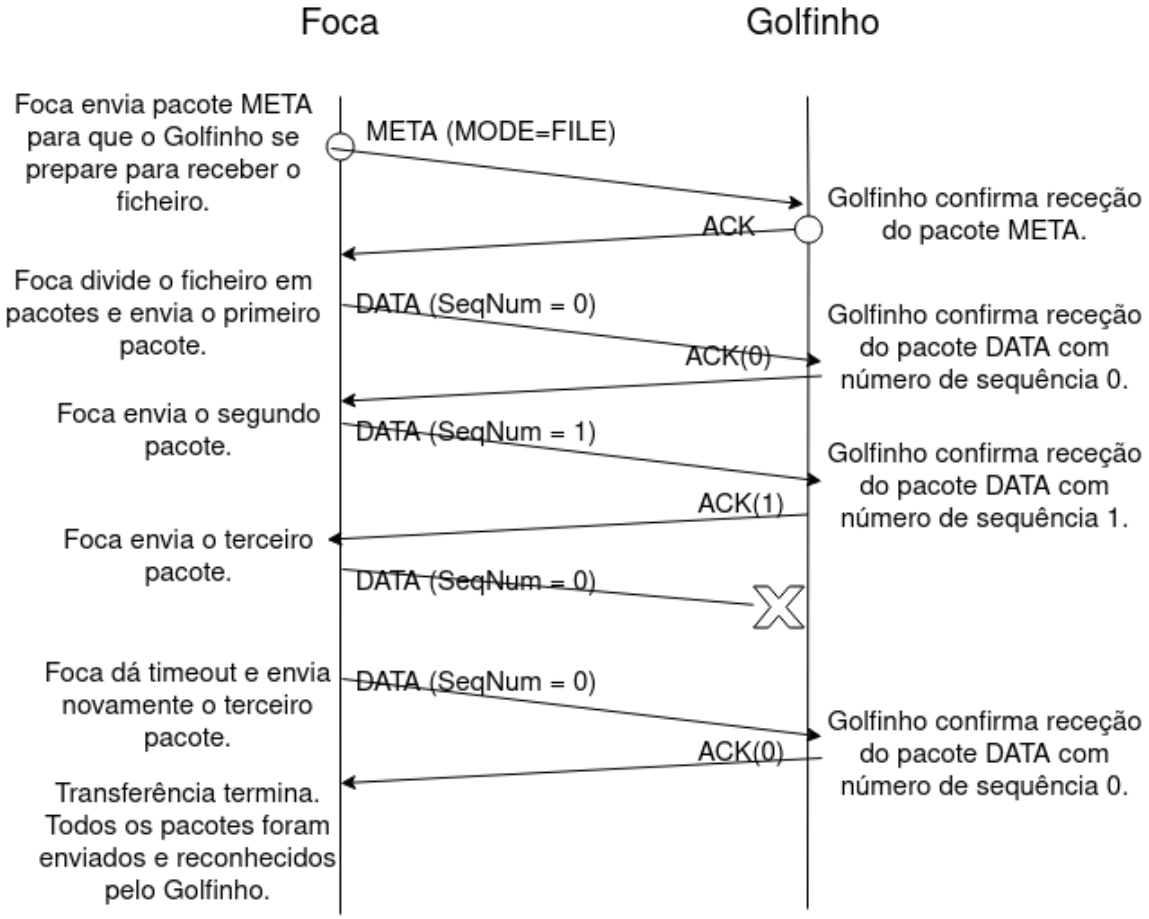

Figure 5: Diagrama Troca de Ficheiro

Notas Diagrama: O diagrama acima (stop-and-wait) representa a troca de um ficheiro entre a Foca e o Golfinho. É de realçar de que, para que seja enviado um pacote de dados,

o pacote anterior tem de ser reconhecido com o devido número de sequência. O último pacote ACK enviado pelo Golfinho pode não chegar ao destino. Neste caso, o Golfinho, como já recebeu o ficheiro, vai se embora e a Foca, como não recebe o pacote ACK, reenvia o último pacote DATA um número determinado de vezes, até que desiste e assume que correu tudo bem.

Pacotes FT-Rapid

Descrição dos pacotes do FT-Rapid. Os pacotes são identificados por um inteiro (4 bytes) e os conteúdos são delimitados pelo caractere '***@***'.

- INIT (***1@random@filename@***)
  - O pacote *INIT* contém um inteiro (4 bytes) "aleatório" e o nome da pasta que pretendemos manter sincronizada.
- INIT_ACK (***2@filename@***)
  - O pacote *INIT_ACK* contém apenas um nome de um ficheiro.
- ACK (***0@seqNum***)
  - O pacote *ACK* contém o número de sequência (inteiro de 4 bytes) do pacote que está a "reconhecer".
  - O número de sequência pode ser 0/1 caso seja relativo ao envio de um ficheiro ou então 2, caso seja relativo a um pacote de controlo (qualquer pacote exceto *DATA*).
- DATA (***4@seqNum@data***)
  - O pacote *DATA* contém o número de sequência (inteiro de 4 bytes) que o identifica assim como os respetivos dados.
- META (***3@mode@size@[filename@]***)
  - O pacote *META* contém o tipo de dados que está a enviar (***mode***: inteiro de 4 bytes), e o tamanho desses dados (***size***: inteiro 4 bytes).
  - Caso o pacote *META* anteceda o envio de um *File*, é incluído o nome desse ficheiro.
- INVALID
  - Este pacote é identificado com um código de: "-1". O pacote é utilizado quando são recebidos dados que não são pacotes FTRapid.

Estado do Sistema (*Logs* em *HTTP*)

Para podermos colocar a informação sobre as sincronizações no servidor *HTTP*, optamos por uma solução, que consiste na escrita e leitura de ficheiros binários. Isto consiste na criação de uma diretoria "*HistorySaved*" que vai conter um ficheiro por sincronização, com a informação de todos os ficheiros.

Para isto criamos um *package HistoryRecorder* que é responsável por guardar a informação num ficheiro. A classe *FileTransferHistory* contém a informação para cada

ficheiro dentro de uma determinada sincronização, nomeadamente, a última vez que este ficheiro foi transferido, o tempo que demorou essa transferência (em ***milissegundos***) e o débito final em bits por segundo, que é apresentado no servidor em ***Kbps***. Esta classe também contém um método *toHTML*, que coloca numa string o código html com toda a informação, para facilitar a sua transformação.
Para mais informação consultar a documentação em anexo.

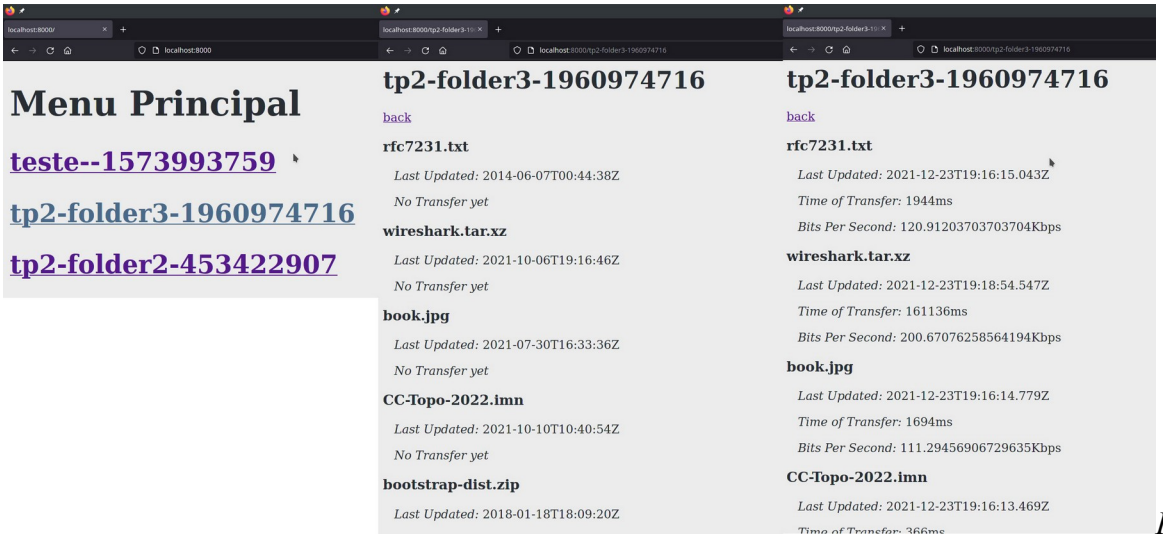

Figure 6: Preview dos Logs no servidor HTTP.

Nota: Os valores não são representativos dos tempos de transferência dos ficheiros. Para mais informação consulte a secção de Testes e Resultados.

Implementação

A documentação do código encontra-se disponível em anexo, sendo que nesse documento encontram-se todos os detalhes da implementação do nosso FFSync.
Estas são algumas das bibliotecas que utilizamos na nossa implementação:

1. *java.net*, mais concretamente as classes DatagramSocket, DatagramPacket, InetAdress, entre outras classes de Exceções desta biblioteca.
2. *java.nio* para operações sob ficheiros.
3. *java.util.Random* para gerar números aleatórios.
4. *java.util.concurrent* para locks e condições.
5. *java.util.zip* para calcular o Checksum de um ficheiro (CRC32).
6. *java.io* para ler/escrever no server HTTP.

Testes e Resultados

O FFSync foi executado em dois nós da topologia virtual CC-Topo-2022.imn, disponibilizada pela equipa docente.
O computador utilizado tem as seguintes características:

	Computador 1
Processador	Intel Core i7 5950HQ @
Memória RAM	16GB, DDR3L, 1600MHz
Sistema Operativo	Manjaro 21.2.0 (Kernel: Linux5.15.7-1-MANJARO)

Os ficheiros a serem enviados fazem parte da pasta *tp2-folder3* disponibilizada pela equipa docente e são:

1. bootstrap-dist.zip (592 115 bytes ≃ 578,2 KiB).
2. Chapter_3_v8.0.pptx (6 140 969 bytes ≃ 5,9 MiB).
3. wireshark.tar.xz (32 335 284 bytes ≃ 30,8 MiB).

Os ficheiros são enviados sempre a partir do nó do Portátil1 para o Golfinho, utilizando um tamanho do pacote de dados variável.

	File 1		File 2		File 3	
Tamanho Pacote (bytes)	*T* (ms)	Kbps	*T* (ms)	Kbps	*T* (ms)	Kbps
2048	1244	475.98	4715	1302.432	16555	1953.203
8192	474	1249.188	1372	4475.925	3242	9973.869
16384	816	7525.698	3766	157.227	5714	5658.9576
24576	456	1298.498	3922	1565.775	5214	6201.627

Escolhemos portanto um tamanho de pacote de 8192 bytes.

Conclusão e Trabalho Futuro

Para concluir este relatório, gostariamos de referir aquilo que poderia ter corrido melhor, nomeadamente:

1. Não conseguimos implementar o *encode*/*decode* de dados com uma chave simétrica partilhada pelos *peers*. Não conseguimos resolver a exceção *BadPaddingException*. Decidimos deixar o código em comentário na classe *FTRapidPacket*.
2. Não conseguimos implementar um mecanismo que verifique a integridade dos dados, apesar de tentarmos tivemos problemas semelhantes ao *encode*/*decode*.
3. O nosso programa está a sincronizar as pastas e a esperar 5 segundos até que volte a sincronizar. Esta espera poderia ser feita com uma condição, tal como fazemos no *package Transfers* para gerir as transferências de ficheiros. Para resolver esta

questão bastava colocarmos uma condição na classe *Listener* que era ativada sempre que recebia um pedido, tal como fizemos no *package Transfers*.

Referências

Fowler. UML Distilled (3rd ed).Addison-Wesley, 2004.
James F.Kurose. Computer Networking: A Top-Down Approach, 8th Edition.
https://www.geeksforgeeks.org/stop-and-wait-arq/
https://datatracker.ietf.org/doc/html/rfc1350/