Universidade do Minho

Licenciatura em Engenharia Informática

Comunicações Por Computador

Grupo 45

TP2: FolderFastSync - Sincronização rápida de pastas em ambientes serverless

Maria Eugénia Bessa Cunha (A93264)

Vicente Gonçalves Moreira (A93296)

Tânia Filipa Soares Teixeira (A89613)

Novembro 2021

1 Introdução

Nas aulas práticas da unidade curricular de Comunicações por Computador, foi-nos proposto o desenvolvimento de uma aplicação de sincronização de pastas entre clientes. A sincronização terá de ser realizada sobre um protocolo a definir sendo este sujeito a certos requirimentos como uma rápida performance, que por sua vez terá que ser segura para a sincronização de pastas entre utilizadores, sem recorrer à utilização de servidores externos.

Assim, esta aplicação - intitulada de *FFSync* - funcionará sobre a camada de transporte *UDP*, e irá implementar o uso do protocolo *FT-Rapid Protocol* para a comunicação e transferência de ficheiros.

Resumimos então, de seguida, as considerações, objetivos, especificações e desenvolvimento deste protocolo aplicacional.

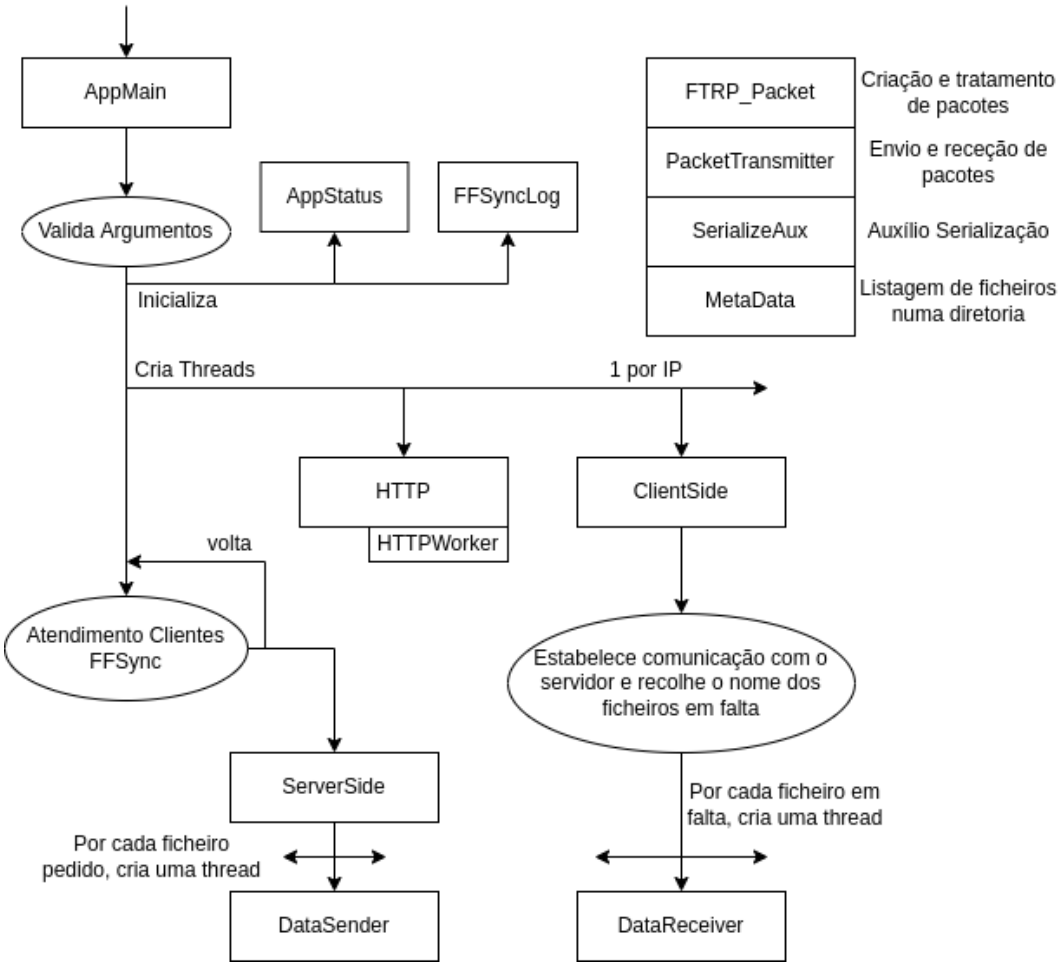
2 Arquitetura da Solução

Dada a particularidade *peer-to-peer* da aplicação, foi necessário desenvolver uma solução de forma a que vários clientes se pudessem coordenar e permitir assim a sincronização de ficheiros numa pasta.

Para este objetivo decidimos repartir as funcionalidades da aplicação em pares "Cliente-Servidor", onde a componente de Servidor da aplicação será responsável por fornecer todos os ficheiros a um Cliente de outra aplicação, sendo a componente do Cliente responsável por pedir, e recolher os ficheiros de um Servidor. Implementando esta solução, com o iniciar da aplicação, depois de verificar os argumentos fornecidos, esta terá de criar um número de Clientes igual ao número de *IP's* ao qual se pretende conectar. Depois disso, a aplicação entrará no modo de atendimento, no qual Clientes de outras aplicações se irão conectar e, depois de verificar os seus *IP's*, são criados Servidores que irão "servir"os Clientes correspondentes.

Apresentamos de seguida um esquema que representa o funcionamento desta solução. Neste, todos as componentes de formato rectangular representam Classes na aplicação e as componentes ovais ações realizadas. As classes *FTRPPacket, PacketTransmitter, SerializeAux* e *MetaData* são classes gerais, que são utilizadas pelas várias *Threads* ao longo da execução da aplicação.

Figura 1: Arquitetura FFSync



3 Especificação do Protocolo

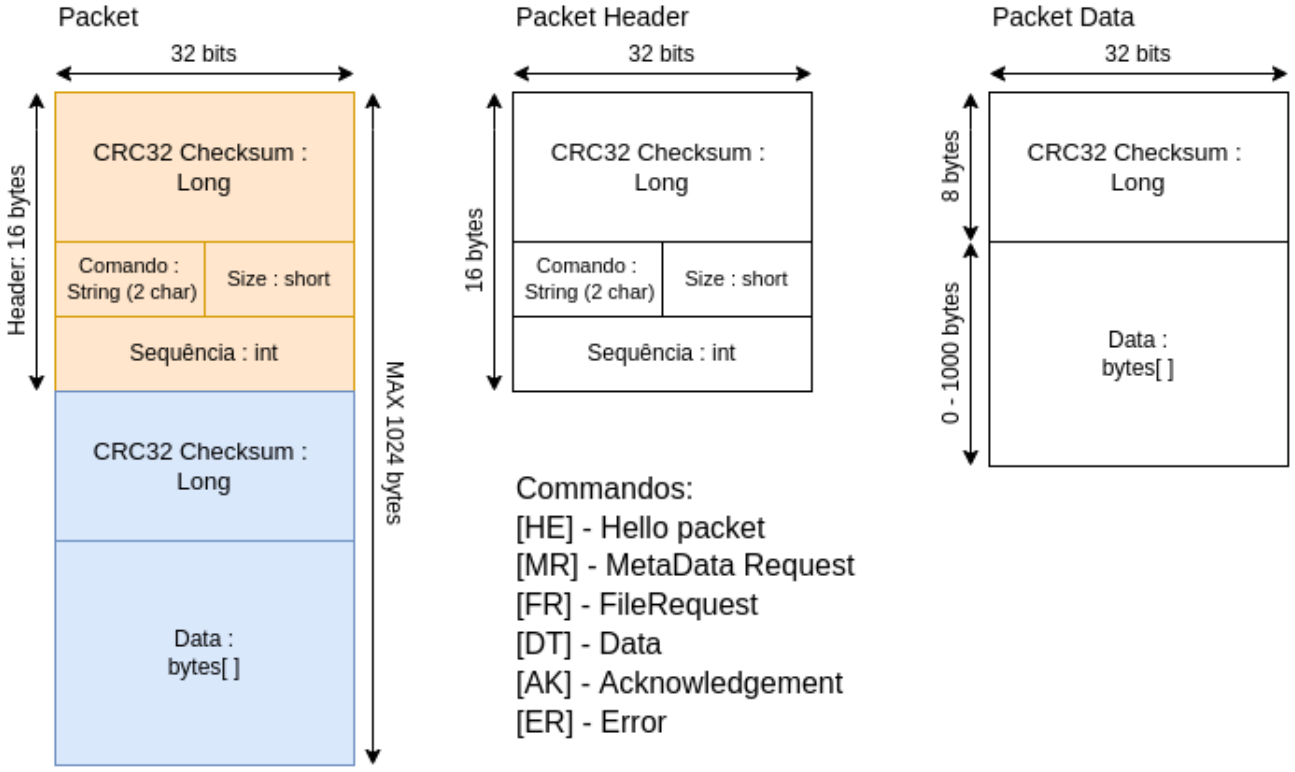
3.1 Mensagens Protocolares

Decidimos começar o desenvolvimento do protocolo por definir as nossas *packets*. Encontramos prático definir um cabeçalho de sintaxe fixa que funcionasse para todas as *packets* que fossem surgir ao longo do desenvolvimento do projeto. Cada uma destas inclui um comando, o seu tamanho total, o seu número de sequência (caso inutilizado, é definido com '1') e, por fim, uma *checksum* aplicada ao cabeçalho de forma a garantir a integridade deste. Desenvolvemos também uma sintaxe simples para transmissão de dados, incluindo apenas uma *checksum* extra, sendo os dados precedidos desta.

Definimos também um tamanho fixo máximo para as *packets* e, tendo em conta o limite comum das *MTU's* de 1500 *bytes*, optamos por um tamanho máximo de 1024 *bytes* ($1KiB$) por *packet*, tendo assim, na tranmissão de dados, um *overhead* de 24 *bytes* por cada 1000 *bytes* de dados transmitido.

3.2 Formato das mensagens protocolares

Figura 2: Packets FT-Rapid Protocol



3.3 Função e significado dos campos

3.3.1 CRC32 Checksum

Este valor é proveniente do método "getValue()"utilizado na biblioteca "java.util.zip.Checksum". Depois de toda a informação da *packet* ser serializada e inserida num *array* de *bytes*, utilizamos o método "update(byte[] b, int off, int len)"para criar a *checksum* da *packet*. (*Offset* de 8 *bytes* de forma a não incluir a *checksum* em si).

3.3.2 Comando

Este valor irá definir o comando a ser interpretado pelo recetor. Inicialmente decidimos utilizar um *unsigned short* com a intenção deste ser interpretado como 2 caracteres alfa-numéricos. No entanto durante a implementação descobrimos que seria mais simples e prático tratar estes dois caracteres como uma *String* e codificar de acordo com isso.

'HE' - Hello packet - Packet para estabelecimento de conexão
'MR' - Metadata Request - Pedido da informação da Metadata
'FR' - File Request - Pedido de ficheiro
'DT' - Data Packet - Dados
'ER' - Error - Erro de transmissão
'AK' - Acknowledgement - Transferência concluída

3.3.3 Size

Este valor irá definir o tamanho em *bytes* do pacote transmitido. A sua posição fixa no *packet* garante que este poderá ser sempre lido e interpretado. Encontramos a necessidade de incluir este valor para facilitar o uso de *checksums* assim como definir o tamanho de *packets* de dados, sendo também usado de modo a sinalizar o fim das transmissões de dados. Para este protocolo decidimos implementar um tamanho máximo de 1024 *bytes* por *packet*.

3.3.4 Sequência

Este campo tem como função controlar a ordem de chegada e leitura de uma quantidade de múltiplas *packets*, assim como confirmar e/ou indicar erros de transmissão de cada *packet*.

3.3.5 Dados

Neste protocolo, haverá apenas três situações onde a secção de dados será utilizada:

- Na transmissão de dados de ficheiros a serem sincronizados.
- Na transmissão dos 'MetaDados', pedidos pelo cliente. (É utilizado a *packet* [DT])
- Na transmissão de *File Request* [FR], onde o nome do ficheiro será convertido para binário.

Dado a unicidade desta *packet* (só pode ser enviada uma vez), criamos uma restrição ao programa, na qual nomes de ficheiros que ultrapassem a capacidade da secção dos dados, ou seja, maior que 1000 *bytes*, serão ignorados.

3.4 Protocolo de Comunicação

3.4.1 Cliente-Servidor

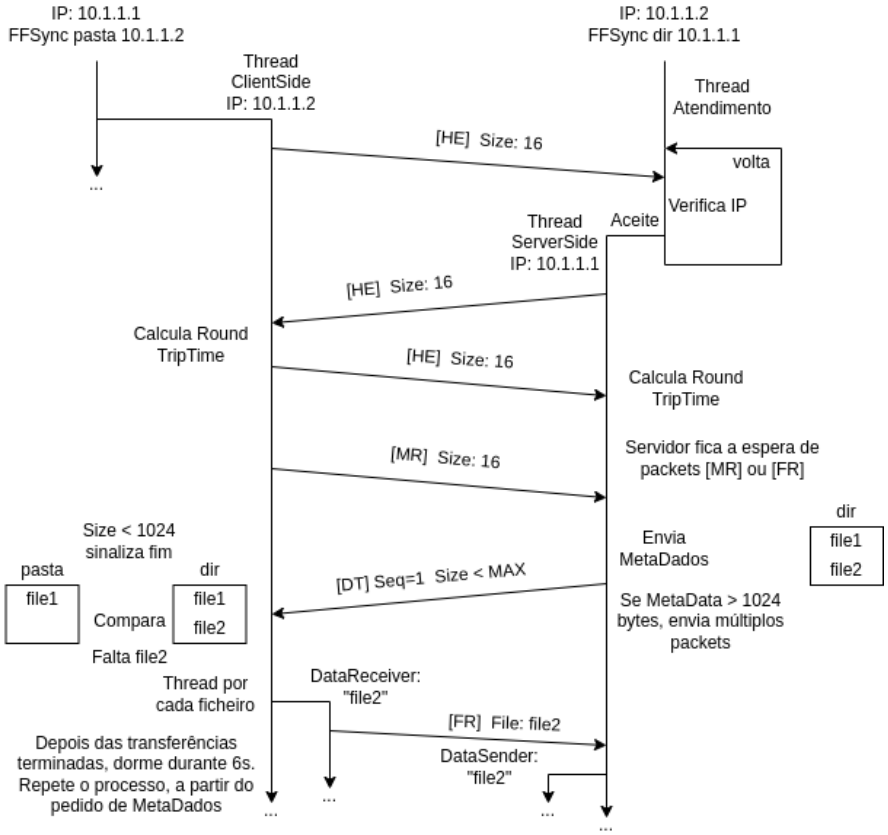
Para o protocolo de comunicação, desenvolvemos um protcolo com o objetivo de obter um comunicão rápida e eficaz, onde o cliente efetua "pedidos", recebendo respostas do servidor. Inicialmente é estabelecida a comunicação entre servidor e cliente, através de uma comunicação semelhante ao protocolo *TCP*, o cliente começa por enviar um pacote [HE] a cada $5s$, até obter uma resposta semelhante do Servidor, ao qual o cliente responde novamente com um pacote [HE]. Com esta troca de pacotes, tanto o cliente como o servidor calculam o *RoundTripTime* da comunicação.

De seguida, o cliente pode efetuar uma série de pedidos, como receber os 'Metadados' do Servidor [MR], ou pedir de um ficheiro [FR]. O Cliente realiza um *loop* onde começa por obter os 'Metadados' do Servidor, calcula os ficheiros em falta e cria as várias *Threads* necessárias para obter todos os ficheiros. Depois das transferências terminarem, volta ao início do *loop*.

Pacote [MR] - Caso o servidor receba um pedido de 'MetaDados', envia uma sequência de pacotes de dados com a informação dos seus 'MetaDados' atualizados, esta transferência não possui controlo de retransmissões dado que estes dados possuem dimensões reduzidas. Caso a transmissão não ocorra com sucesso, cabe ao Cliente efetuar um novo pedido.

Pacote [FR] - Para a transferência de ficheiros o Cliente começa por criar uma nova *Thread "DataReceiver"* que será responsável por contactar o servidor e efetuar o pedido do ficheiro. Quando o servidor recebe este pedido, este é reencaminhado para uma *Thread "DataSender"* que efetuará o envio do ficheiro. Tanto a *Thread "DataReceiver"*, como a *Thread "DataSender"* comunicam paralelamente, permitindo ao Cliente e Servidor efetuarem outras operações.

Figura 3: Protocolo Cliente-Servidor



3.4.2 Transferência de Ficheiros

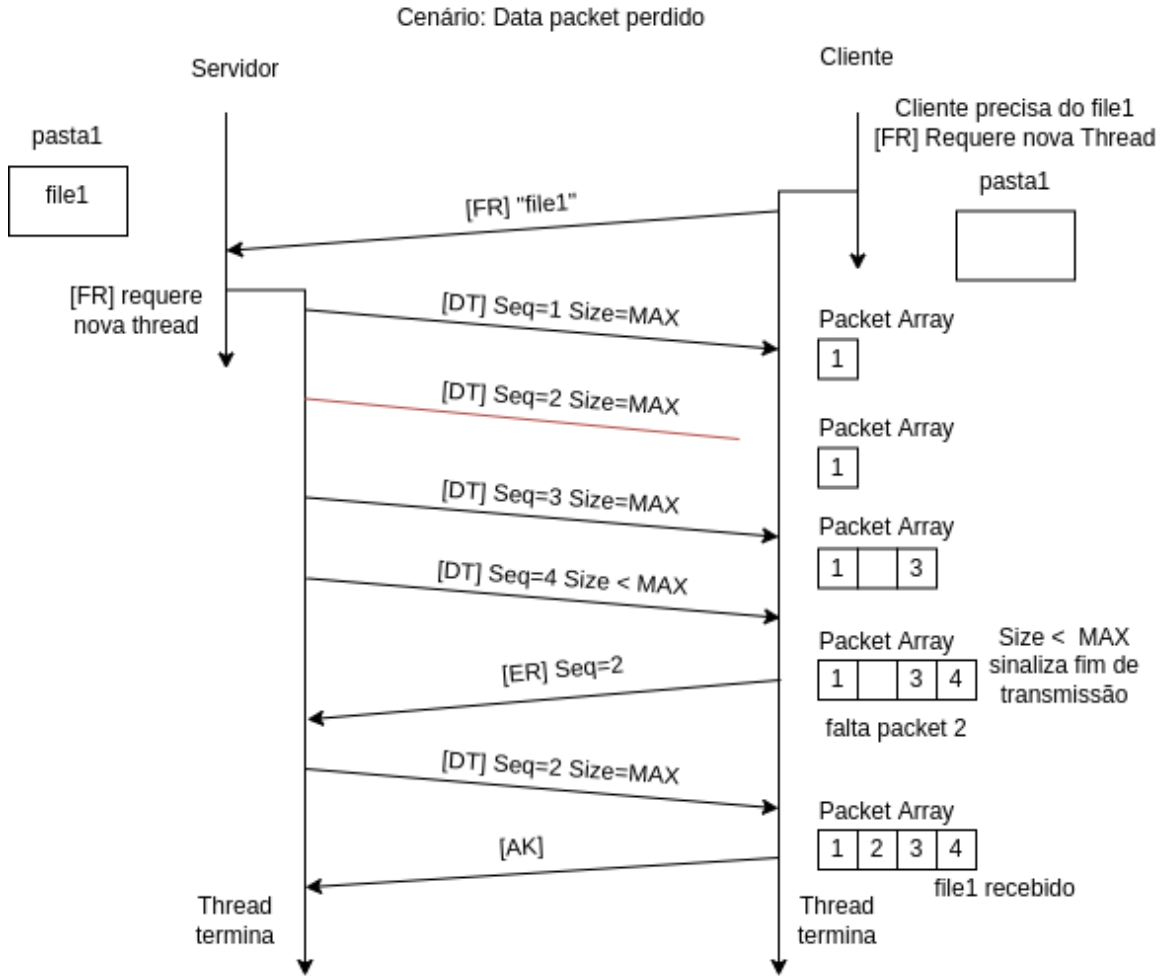
Para a transferência de ficheiros, começamos por implementar um sistema *"Stop and Wait"*. Apesar deste ser fácil de implementar e ser bem sucedido, decidimos que o ritmo de transmissão era inadequado, especialmente para ficheiros de maior dimensões.

Decidimos por isso modificar o protocolo de transmissão. Dado a verificação de um número baixo de pacotes perdidos na transmissão, optamos por escolher uma estratégia focada no controlo de erros e pacotes perdidos, ao invés de uma estratégia de "confirmação"de todos os pacotes transmitidos.

Deste modo, decidimos que a " *Thread "DataSender"* começaria por transmitir todos os pacotes de dados do ficheiro. Depois de terminada a transferência, sinalizada atráves do envio de um pacote com menos de 1024 *bytes*, este espera por respostas da Thread "DataReceiver"a confirmar a receção total do ficheiro, ou a pedir retransmissão de pacotes em falta.

Apresentamos de seguida um diagrama temporal com a implementação desta estratégia de protocolo.

Figura 4: Transferência com uma perda de pacote

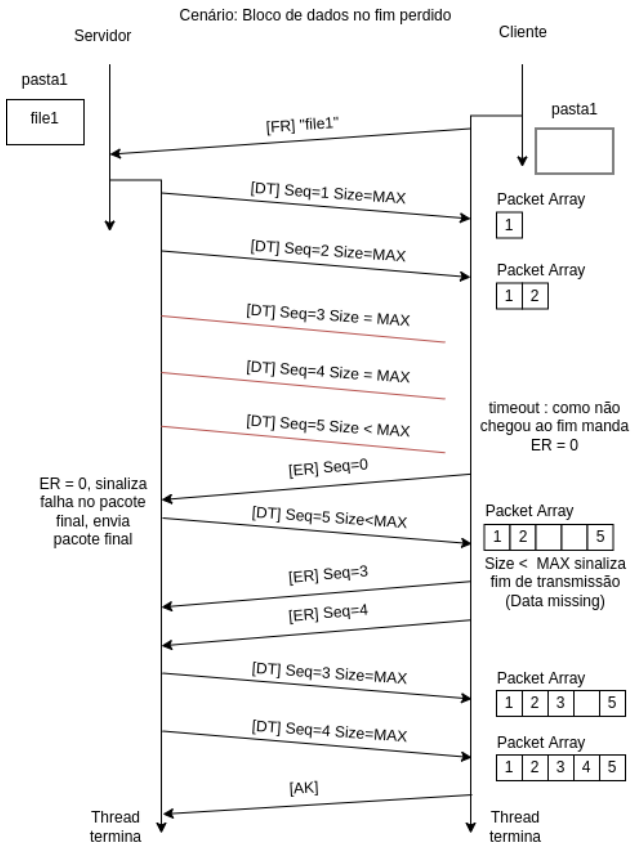


No entanto, esta solução apresentou dois problemas:

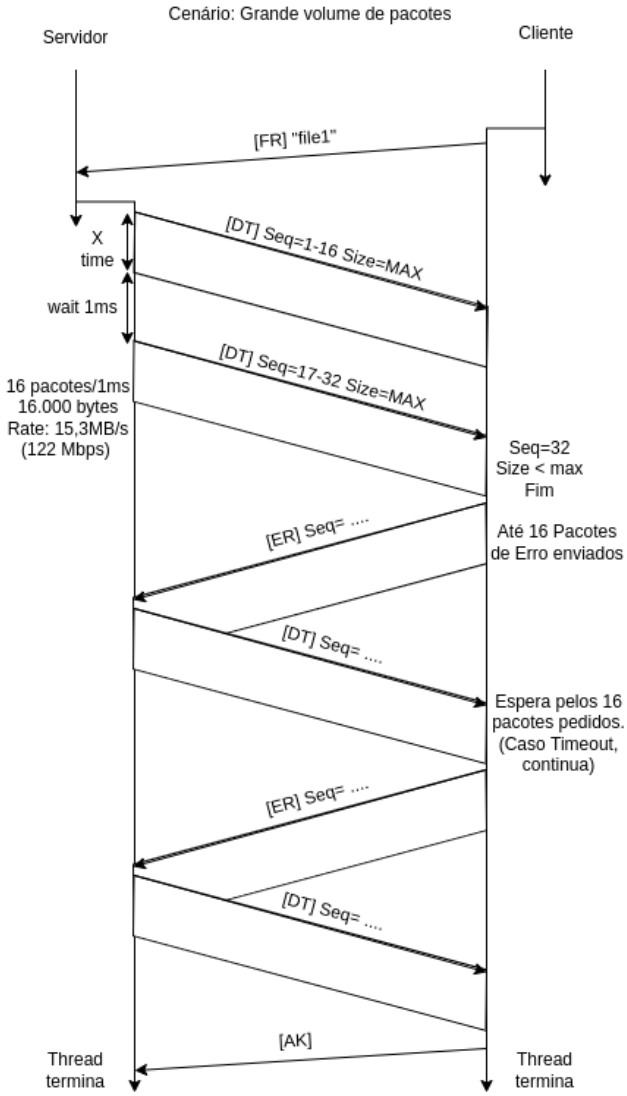
- Caso a transmissão do último pacote falhe, a *Thread "DataReceiver"* nunca irá avançar da fase de receção para a fase de controlo de pacotes em falta. Para resolver este problema, acrescentamos um tempo de *timeout* onde, caso este termine e o último pacote não tenha sido recebido, a *Thread "DataReceiver"* envia um pacote de erro com *Seq=0*, de forma a indicar um erro no último pacote. Depois deste ser retransmitido, a *Thread* prossegue normalmente para a fase de controlo de pacotes em falta.

- Com o envio sucessivo de múltiplos pacotes de dados e devido à necessidade de mais tempo de processamento na receção de pacotes comparado ao envio, encontramos que a *Thread "DataReceiver"* não conseguia acompanhar o ritmo da transferência, o que leva a um aumento significativo de perda de pacotes, os quais têm de ser retransmitidos e, por consequência, aumentam o tempo de transferência. Para a resolução deste problema limitamos o ritmo de envio de pacotes por parte da *Thread "DataSender"*, escolhendo enviar um bloco de 16 pacotes sucessivos, seguidos de uma espera de $1ms$, de forma a permitir que a *Thread* recetora conseguisse acompanhar.

Apresentamos dois diagramas temporais, correspondentes a estas duas situações:



Pacote Final Retransmitido (Pior Caso)



Lógica de transferência de grandes volumes

4 Funcionalidades e Implementação

4.1 Funcionalidades

A aplicação *FFSync*, para além das funcionalidades base requeridas, fornece outras tais como:

- Sincronização entre múltiplas aplicações em simultâneo.
- Sincronização de ficheiros mais recentes.
- Sincronização de diretorias e subdiretorias.
- Sincronização entre Sistemas Operativos Windows e Linux.
- Sincronização de ficheiros, a um ritmo de transferência até $8Mb/s$
- Encriptação na comunicação e transferência de dados.
- Atendimento *HTTP* dinâmico (Atualização automática), permitindo observar a informação e estado de cada componente da aplicação, assim como o estado de transferências atuais e antigas.

4.2 Implementação

4.2.1 FTRPPacket

Esta classe foi desenvolvida para criar um nível de abstração de forma a facilitar a criação, leitura, e uso dos pacotes do nosso protocolo. Para criar pacotes, são usados métodos estáticos que obrigam a utilização dos argumentos necessários que cada tipo de pacote requer de forma a evitar erros na escrita de código. Esta classe também é responsável pela sua serialização, deserealização, verificação de corrupções através da *checksum* e encriptação.

A encriptação utilizada no envio de *packets* é uma modificação da Cifra de César, onde aplicamos um "desvio"a cada *byte* a ser enviado na *packet*. Este desvio é calculado através de uma função sinusoidal '$y = 128 * sen(X * (pi * 16/512))$' onde X representa a posição do *byte* no seu *array*, quando serializado ($0 \leq X < 1024$). De forma a evitar o cálculo desta função para todos os *bytes* enviados/recebidos, esta é calculada um única vez na execução inicial da aplicação e os seus resultados são guardados numa constante.

4.2.2 PacketTransmitter

Esta classe é responsável por facilitar o uso das *Sockets UDP*, assim como fornecer funcionalidades extras para uma maior conveniência. Este permite criar *Sockets* de atendimento, assim como *sockets* "direcionadas", ou seja, o destino dos pacotes já é definido. Também é fornecido os métodos *"sendPacket"* e *"receivePacket"* que processam de forma automática os pacotes para o seu envio/receção, gravando estes no *log* caso o programador desejar.

O método *receivePacket* também tem uma funcionalidade extra que, depois de receber um pacote com sucesso, o *PacketTransmitter* fica "direcionado"para a porta e *IP* do emissor, de forma a permitir respostas. Esta funcionalidade é complementada com o método "makeCopy", que cria uma cópia do *PacketTransmitter* atual, apenas criando uma *socket* de receção diferente. Estas duas funcionalidades provam-se úteis quando uma *Thread* em atendimento recebe um pacote de um emissor, permitindo este criar rapidamente um *PacketTransmitte*r já direcionado ao emissor, sendo este entregue a uma *WorkerThread*.

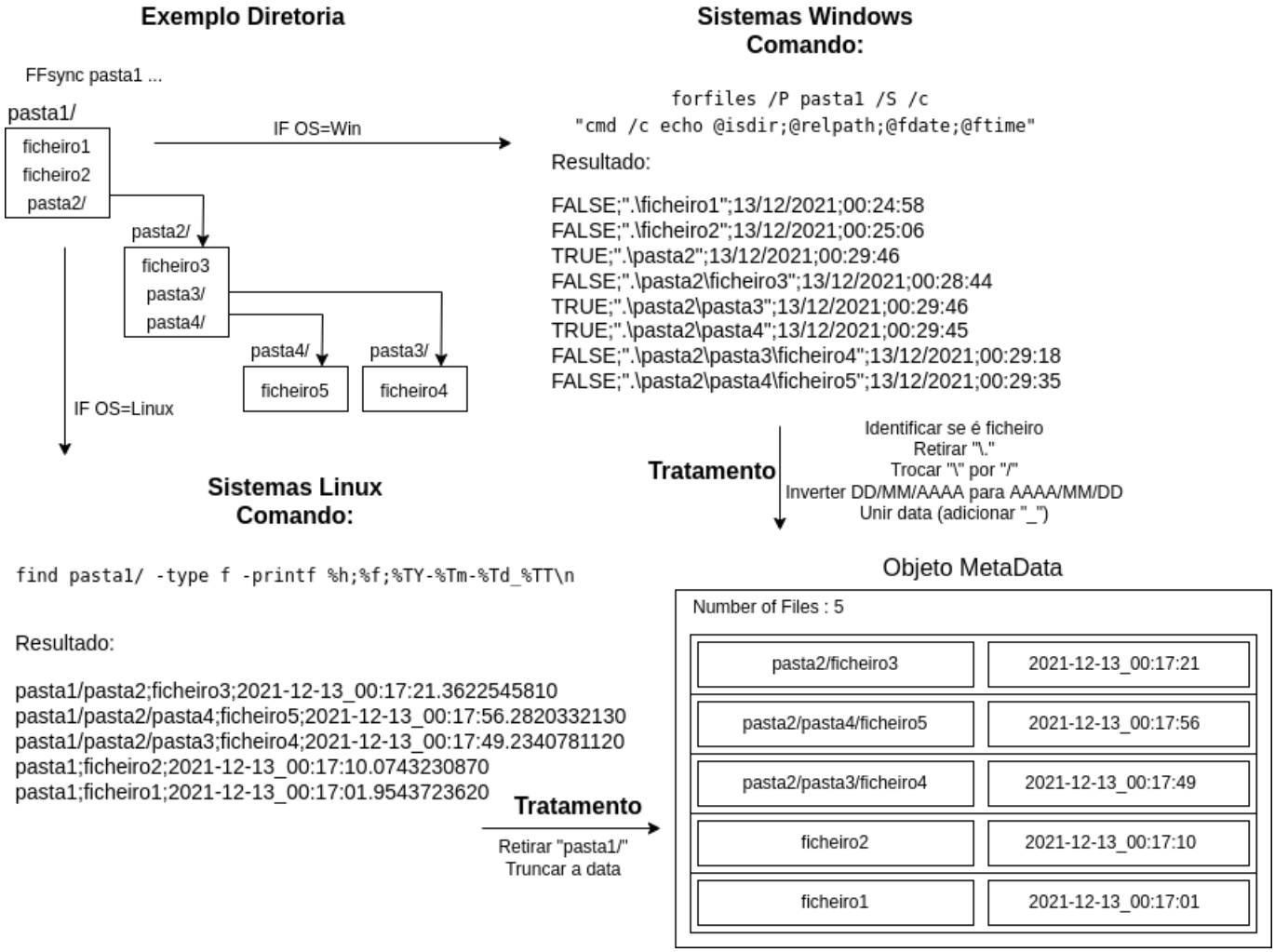
4.2.3 MetaData

A classe "MetaData"é utilizada extensivamente ao longo da execução da aplicação. Esta é responsável por recolher o nome e a data da última modificação de todos os ficheiros na diretoria e subdiretorias, estruturando essa informação. Tanto o Servidor como o Cliente atualizam os seus "MetaDados"utilizando o método "updateMetadata". Este método, dependendo do Sistema Operativo em uso, utiliza um dos comandos indicados para obter a informação da diretoria, esta depois é tratada conforme necessário.

Esta classe também possui métodos de serialização e deserialização, para facilitar o envio.

Apresentamos um diagrama exemplo, de forma a simplificar o processo ocorrido na atulização dos ficheiros numa diretoria:

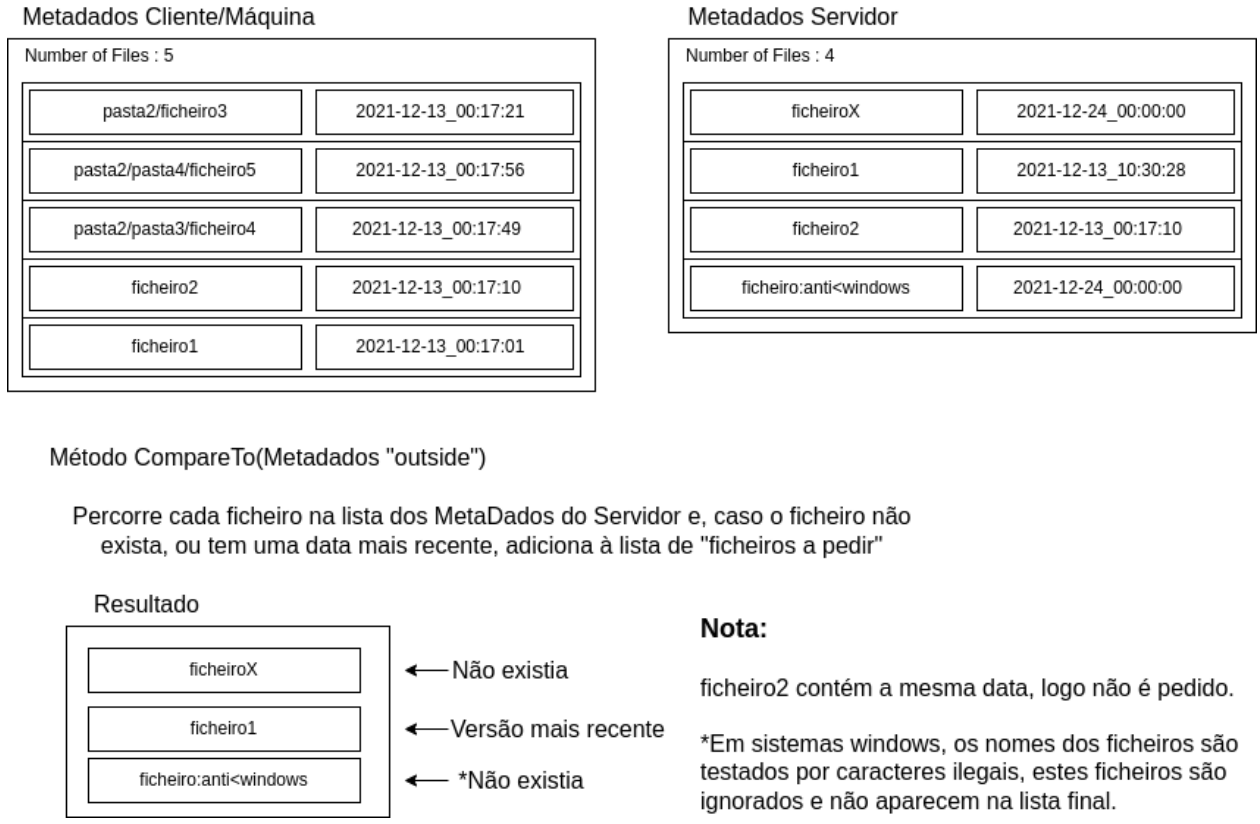
Figura 5: Update MetaData



Depois da informação da sua diretoria recolhida, assim como a obtenção da informação dos "Metadados"do Servidor, o Cliente necessita calcular os ficheiros em falta, para poder criar pedidos. Para isso, a classe "MetaData"oferece um método "compareTo"que permite comparar duas entidades "MetaDados", e devolver uma Lista de *Strings* com os nomes dos ficheiros em falta.

Apresentamos outro esquema em relação ao funcionamento deste método.

Figura 6: Compare MetaData



4.2.4 AppStatus

Esta classe permite o registo do estado de todas as *Threads* em funcionamento na aplicação, esta é composta por um *Map* de Chaves-*ConnectionInfo*. As Chaves são geradas e possuem um codificação única, dependendo da função da *Thread*. EX: *Thread* Cliente para *IP* 10.1.1.1 = /10.1.1.1-C; *Thread* Servidor para *IP* 10.1.1.1 (1ºFicheiro) = /10.1.1.1-S-DT1;

Já a entidade "ConnectionInfo"possui uma série de *Strings* como o nome da *Thread* responsável, estado da *Thread* e informação adicional.

4.2.5 HTTP

Esta classe tem um funcionamento relativamente simples. Esta apenas atende pedidos de clientes, criando uma *Thread HTTPWorker*, que irá ler o ficheiro *html* disponível e acrescentar a informação presente na entidade "AppStatus". Este ficheiro *html* transmitido contêm uma *flag* que indica ao recetor que o conteúdo tem que ser atualizado a cada $1s$. Esta atualização constante permite uma visualização dinânimca do estado da aplicação.

5 Testes e Resultados

Nesta secção apresentamos os vários testes que foram realizados para comprovar o funcionamento da aplicação. Para além da execução bem sucedida dos 4 testes requeridos no enunciado, também efetuamos testes extras de forma a testar a sincronização de subdiretorias, assim como a sincronização entre clientes de vários sistemas operativos.

Verificamos que, com a transferência de ficheiros mais pequenos, a métrica de transferência é inconsistente devido a vários *overheads*. Logo, decidimos realizar testes com ficheiros de grande volume para observar não só potenciais problemas na transferência destes ficheiros, mas como obter uma métrica mais estável. Para isso, utilizamos um ficheiro *ZIP* com cerca de $1Gb$ de tamanho. Realizamos vários testes entre máquinas em ambiente real e dentro da topologia fornecida pelos docentes. Para cada situação efetuamos 5 testes e calculamos a média para cada parâmetro obtido.

Apresentamos aqui a tabela com os resultados obtidos, assim como também a tabela das especificações de cada máquina utilizada.

Figura 7: Testes Realizados

Especificação das Máquinas Usadas				
	SO	Memória (Gb)	Processador	Interface NET (Max.Vel / Tipo)
PC1	Windows 10	16	Intel i5-8400 @2.80GHz	1Gbps (Ethernet)
PC2	Ubuntu 20.04.03 LTS	8	Intel i5-8300H @2.30GHz	1Gbps (Wifi)
PC3	Windows 10	16	Intel i7-1165G7 @2.80Ghz	144Mbp (Wifi)

Ficheiro Zip: Tamanho: 1Gb (1011181873 bytes) | Nº testes = 5				
	Tempo (s)	Ritmo (Mb/s)	Nº Packets Perdidos	Packet Loss (%)
PC1 -> PC2[1]	126.65	7.98	1350	0.13
PC2 -> PC1[1]	137.84	6.89	33368	3.3
PC2 (Topologia)[2]	78.92	9.84	2225	0.22
PC2 -> PC3[3]	143.34	5.53	230194	22.76

[1]PC1 e PC2 na mesma rede local.
[2]Topologia Core fornecida (Servidor1 -> Portatil1)
[3]PC2 e PC3 conectados através do serviço Hamachi. (Braga -> Póvoa de Lanhoso)

Podemos observar que a nossa aplicação não só é eficaz na sincronização de ficheiros, como também obtém ritmos de transferência satisfatórios. No entanto, como se pode verificar, existe uma limitação à velocidade da transferência, especialmente em máquinas com menor capacidade de processamento. Acreditamos que isto aconteça devido à maior necessidade de processamento na receção das pacotes, o que resulta num maior número de pacotes perdidos, obtendo ritmos mais baixos.

6 Conclusão

Assim, concluimos com sucesso o nosso projeto. Encontramo-nos satisfeitos com o protocolo desenvolvido, tendo atingido metas razoáveis, quer para a segurança, quer para a eficácia e rapidez de troca de ficheiros.

No entanto também reconhecemos que existe falta de controlo mais preciso no envio de pacotes sucessivos. A definição de *timeouts* fixos trouxe em algumas situações um aumento da ineficiência à aplicação. Futuramente, poder-se-ia desenvolver um sistema de controlo de fluxo simples, e implementar funções de receção mais eficientes com a experiência do funcionamento das *sockets UDP*.

Em suma, este projeto foi desafiante e proporcionou aos elementos do grupo uma melhor compreensão do funcionamento dos conteúdos leccionados, em particular, transmissão de dados por *sockets*.