Comunicações por Computador
Trabalho Prático 2: Gateway Aplicacional e Balanceador de Carga sofisticado para HTTP

Ana Rita Peixoto, Leonardo Marreiros, and Luís Pinto

University of Minho, Department of Informatics, 4710-057 Braga, Portugal
e-mail: {a89612,a89537,a89506}@alunos.uminho.pt

1 Introdução

No âmbito do trabalho prático 2 da UC Comunicações por Computador foi elaborado um *Gateway* aplicacional e balanceador de carga para pedidos HTTP. Deste modo, foi necessário implementar o *gateway HttpGw* e vários servidores, *FastFileSrv*, que possuem acesso a todos os ficheiros disponíveis para *download*. Estas entidades comunicam entre si através do protocolo implementado, *FSChunkProtocol* que funciona sobre UDP. Posteriormente, o *gateway* envia o ficheiro pedido (áudio, vídeo, texto ou imagem) de volta ao cliente, através do protocolo HTTP. A conexão entre o *gateway* e o cliente é feita com recurso ao protocolo TCP.

2 Arquitetura da solução

Para a resolução do TP2 e de forma a dar resposta aos requisitos propostos, o projeto foi implementando em **Java** recorrendo às suas funcionalidades e bibliotecas, por exemplo, nas comunicações TCP e UDP.

A solução implementada possui duas vertentes: Cliente <-> *Gateway* e *Gateway* <-> Servidores. A ligação entre Cliente e *Gateway* é estabelecida por parte do cliente com o envio de um pedido wget (ex: wget http://10.1.1.1:80/um-ficheiro-grande.pdf), assegurada por TCP. O gateway processa esse pedido, pedindo informação aos FastFileSrv e envia a resposta de volta ao cliente. A vertente mais importante da implementação foi o lado *Gateway* <-> FastFileSrv, agora recorrendo aos protocolos UDP e FSChunkProtocol. Desta forma, foi preciso implementar classes auxiliares como : FSChunkProtocol - responsável por materializar as ideias associadas ao protocolo a implementar, FastFileSRV - relativa aos FastFileServers e a sua execução, HttpGw - o *gateway* aplicacional que assegura as respostas aos clientes, Ping - para sinalizar a vivacidade de cada FastFileSrv ativo, TimeoutFS - para definir um timeout para as comunicações, twodParityCheck e ChecksumFS para detetar erros nos envios, entre outras também relevantes que explicaremos em maior detalhe na secção da implementação. A arquitetura proposta teve em conta a demanda de vários clientes e assumiu também que os todos os FastFileSrv tem acesso ao mesmo conteúdo podendo por isso a carga de um pedido de um dado cliente ser distribuída. A figura a seguir ilustra de uma forma abstrata mas explicativa a base da arquitetura implementada.

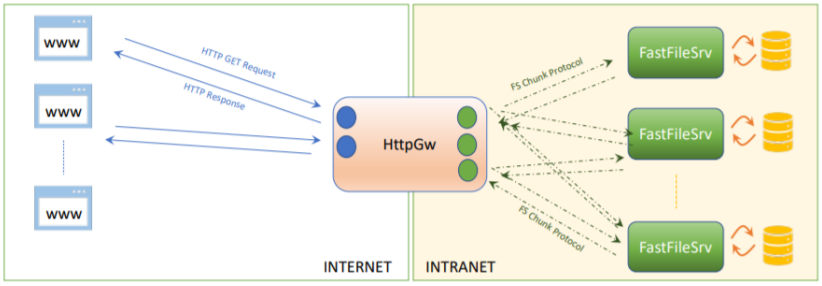

Figura 1: Esquema geral de funcionamento

Fig. 1: Base da arquitetura da solução

3 Especificação do protocolo

3.1. Formato das mensagens protocolares e Significado dos campos

Cada mensagem protocolar entre os FastFileSrv e o gateway é transmitida com recurso a UDP e empacotada num FSChunkProtocol. Desta forma foi preciso estabelecer quais os campos relevantes a considerar no protocolo para permitir o correto funcionamento da comunicação gateway<->fastfileservers. A seguir apresentamos a sintaxe do protocolo bem como a correspondente semântica.

Desta forma, foi necessário implementar na classe *FSChunkProtocol* diferentes variáveis que correspondem a diferentes campos do protocolo. O campo "filename" corresponde ao nome do ficheiro pedido, "payload" corresponde ao local onde são transmitidos todos os bytes da mensagem, "checksum" contém o código de controlo de erros, "offset" é útil no caso da fragmentação, "fragment" identifica o número do fragmento que está a ser transmitido, "lastFragment" é uma *flag* que identifica se se trata do último fragmento, "port" contém a porta à qual um dado *FastFileSrv* está conectado, "address" possui o endereço de um *FastFileSrv*, o campo "client" identifica qual o cliente em questão, e, por fim, o campo "type" permite distinguir diferentes tipos de mensagem, isto é, na comunicação *Gateway* -> *FastFileSrv* o tipo 1 identifica um pedido de um ficheiro e o tipo 2 identifica o pedido de apenas um fragmento. No espetro das transmissões *FastFileSrv* -> *Gateway* existem 3 tipos de mensagem: o tipo 1 identifica uma mensagem de autenticação, 2 remete para a transmissão de dados de um ficheiro, e as mensagens do tipo 3 correspondem às mensagens de *ping*.

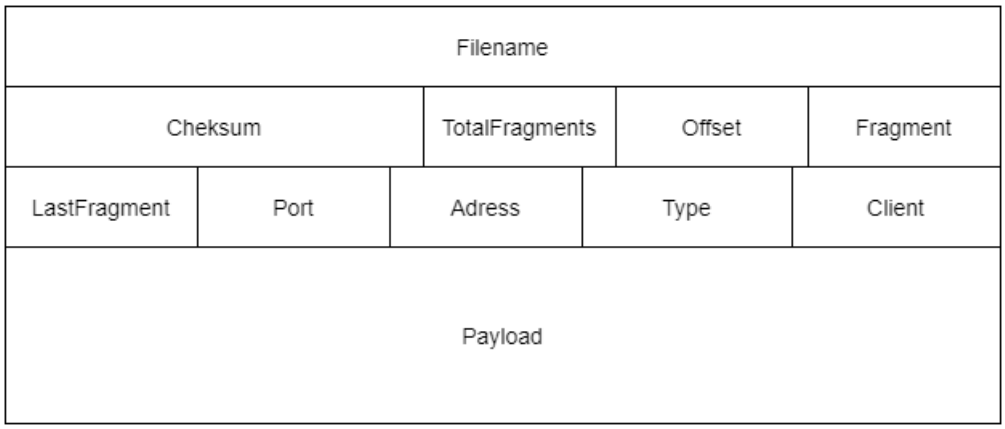

Fig. 2: Estrutura *FSChunkProtocol*

3.2. Diagrama temporal ilustrativo - Comportamento

Na figura seguinte é possível observar as interações do servidor para o cliente e a sua conexão. Além disso, também podemos ver o procedimento de um pedido por parte do cliente.

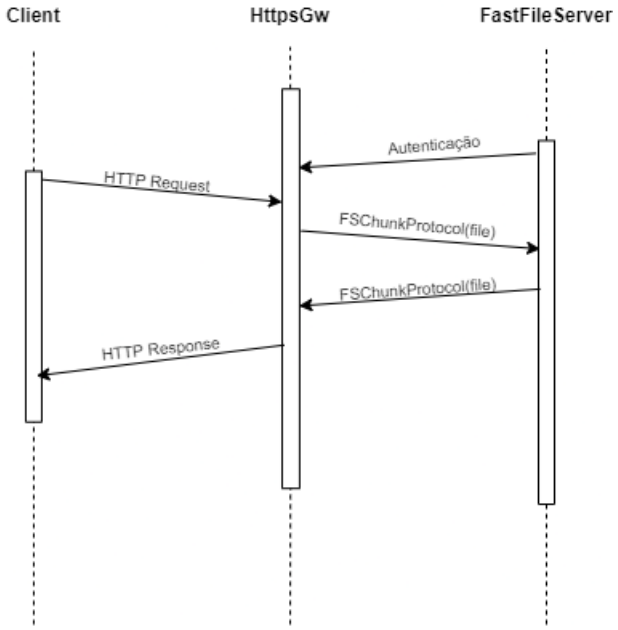

Fig. 3: Diagrama temporal interação com o *gateway*

4 Implementação

4.1. HttpGw - Gateway Aplicacional

De forma a responder aos pedidos provenientes dos clientes, foi necessário implementar um *gateway* aplicacional capaz de receber mensagens de clientes e servidores através de protocolos de aplicação distintos. Assim, foi implementada a classe HttpGw que está encarregue de escutar mensagens provenientes de clientes e de FastFileSrv, recorrendo ao protocolo TCP e UDP, respetivamente. Dentro desta classe foram implementadas outras classes, "TCPConnection" e "Packet". A primeira possui o propósito de armazenar informações relativas à conexão de cada cliente. A classe "Packet" guarda as informações relativas aos dados de um ficheiro pedido por um cliente. Além disso, também foram implementadas as classes "UDPListener" e "TCPListener" que correm dentro de *threads* distintas, de modo a permitir a escuta de mensagens de cada protocolo.

Conexão TCP - TCPListener

A classe encarregue de estabelecer conexões TCP e tratar de pedidos dos clientes é a classe "TCPListener", recorrendo à biblioteca "Socket" do Java de forma a efetuar a comunicação. Nesta classe é aberto um *socket* que está à escuta na porta 8080. Por cada novo cliente, é criada uma nova conexão e uma *thread*

para a tratar. Através desta conexão são enviados pacotes *Http Request* do cliente para o *gateway* e pacotes *Http Response* do *gateway* para o cliente. Deste modo, a classe está preparada para interpretar pedidos Http, efetuando o seu *parse* e extraindo o nome do ficheiro pretendido. Após obter o nome do ficheiro, verifica se algum *FastFileSrv* possui acesso àquele ficheiro. Em caso afirmativo, esse pedido é transmitido para a classe "UDPListener" que trata a outra conexão e do pedido. Em caso negativo, é enviada uma resposta Http sem conteúdo, recorrendo ao estado "204 No Content".

Conexão UDP - UDPListener

De modo a efetuar pedidos de ficheiro aos servidores recorreu-se à classe UDPListener, através do protocolo FSChunkProtocol. Esta classe está à escuta de pacotes FSChunk na porta 8888 e está encarregue de enviar os pedidos aos FastFileSrv e de receber pacotes. Para concretizar tal feito, recorreu-se ao uso de *threads*.

Esta classe necessita de ter FastFileSrv conectados de forma a enviar pedidos de ficheiros. Para isso, aguarda por mensagens de autenticação por parte dos servidores (tipo 1). Estas mensagens, além de conterem o *username* e *password* também contém o nome dos ficheiros que os servidores tem acesso. Após efetuar a validação dos dados de autenticação, esse servidor é armazenado na *pool* de servidores, e os ficheiros aos quais tem acesso são guardados numa estrutura. O *gateway* suporta **servidores FastFileSrv ilimitados**.

Quando um cliente envia um pedido Http, a *thread* encarregue de pedir os ficheiros aos servidores é despertada e efetua o pedido do ficheiro, na sua íntegra, **a um único servidor, selecionando o que está menos ocupado.** O critério de seleção baseou-se no número de pedidos concretizados pelos servidores, ou seja, o servidor menos ocupado é aquele que recebeu menos pedidos até ao momento. Paralelamente a esta *thread*, ocorre um processo de escuta respostas por novos pacotes UDP (FSChunkProtocol) na porta 8888. Deste modo, quando receber uma resposta (FSChunkProtocol), em *bytes*, deserializar-se-á o mesmo e em seguida procede-se ao armazenamento dos dados nele encapsulados numa estrutura. Caso o pacote recebido seja apenas um fragmento, então guarda-se esse fragmento e espera-se até a receção de todos os outros, para então enviar a resposta para o cliente. Por fim, antes do envio, é feita uma organização de todos os fragmentos recebidos tendo em conta o número do fragmento presente no FSChunkProtocol.

Para efetuar a comunicação entre estas duas classes TCPListener e UDPListener que executam em *threads* diferentes, foi necessário efetuar mecanismos de controlo de concorrência, com recurso a *locks* e variáveis de condição.

Retransmissão

De forma a ultrapassar o obstáculo que é a perda de pacotes na rede, implementamos mecanismos de retransmissão. O mecanismo implementado possui 2 casos: o caso em que o *gateway* recebe o último pacote e o caso em que não recebe. Para o primeiro caso, é efetuado um pedido de retransmissão quando recebe o **último pacote** mas existem falhas de pacotes intermédios. O segundo mecanismo para a retransmissão foi a implementação de um ***timeout*** (classe "TimeoutFS") que verifica de X em X tempo se existem pacotes em falta. Este mecanismo complementa o primeiro, na medida em que caso o ultimo pacote tenha sido enviado mas perdido ainda será efetuada a retransmissão devido a *timeout*. Em ambos os casos, são efetuados pedidos do tipo 2 aos servidores, que denotam o pedido por um único fragmento, para pedir individualmente cada fragmento em falta. Existe **distribuição de carga** para os *FastFileSrv* uma vez que cada fragmento é pedido ao servidor menos ocupado, e também **paralelismo** dado que os fragmentos a ser retransmitidos são pedidos aos múltiplos servidores.

4.2. FastFileSrv - Servidores

De modo a aceder aos ficheiros disponíveis para *download* e envia-los para o *gateway* foi implementada a classe "FastFileSrv".

Autenticação

Ao executar um FastFileSrv é indicado como argumento o endereço e a porta do servidor ao qual se vai conectar, assim como o *username* e a *password* necessários para se autenticarem (optamos por username : server e password : 1234 a titulo de exemplo). Para a autenticação no *gateway*, cada FastFileSrv envia inicialmente um pacote de autenticação (com *type* 1) onde contém os nomes de todos os ficheiros a que tem acesso, assim como os dados de autenticação (*username* e *password*).

Fragmentação

Quando o FastFileServer recebe um pedido é necessário ler o ficheiro, transforma-lo em *bytes* e por fim verificar se é necessário fragmentação. O valor de cada *chunk* é **5000 bytes** e foi calculado a partir da **média ponderada** relativamente aos testes feitos e ao tamanho dos ficheiros para *download*. Nos casos em que não há fragmentação, é simplesmente enviado um fragmento que contém todos os dados do ficheiro. No caso em que é necessário haver fragmentação de um ficheiro, tal como fora mencionado, procedeu-se à sua divisão em pacotes de **5000 bytes**, salvo o último fragmento que por norma não terá um valor exato de 5000 bytes mas sim o resto da divisão inteira do número total de bytes do ficheiro por 5000, posteriormente estes "chunks" serão enviados sequencialmente para o *gateway*.

Ping

De modo a fornecer informação ao *gateway* quanto aos **servidores ativos**, foi implementada a classe Ping. Esta classe é invocada em cada FastFileSrv e tem como função enviar mensagens periodicamente para o *gateway* de forma a informa-lo que ainda está ativo. Deste modo, quando o HttpGw deixa de receber mensagens de Ping de um dado servidor, considera que a conexão foi abaixo e o servidor já não está ativo. Para melhor ajuste a realidade de uma conexão ter ido efetivamente a baixo, estabelecemos que o servidor é considerado morto se falhar 3 Pings, contudo visto que se trata de um valor arbitrário podemos ajusta-lo consoante a instabilidade da rede.

4.3. Deteção de Erros

Nenhuma correção de erros ou confirmação de erros é feita pelo protocolo UDP. Este protocolo apenas se preocupa com a velocidade. Por esta razão, quando são enviados dados pela Internet, estes estão propensos à ocorrência de erros.

Desta forma, uma das funcionalidades que adicionamos ao FsChunkProtocol foi um mecanismo de deteção de erros através do método ***Two Dimensional Parity Check*** e *Cyclic Redundancy Code*, um tipo de ***checksum***.

Two Dimensional Parity Check

Os bits de verificação de paridade são calculados para cada linha e coluna, em seguida, ambos são enviados junto com os dados. Na extremidade de recepção, são comparados com os bits de paridade calculados nos dados recebidos.

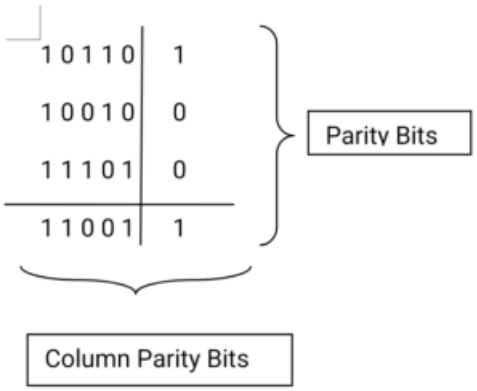

Fig. 4: Exemplo de 2dParityCheck

Como podemos verificar pelo exemplo acima, para cada linha é adicionado um bit de forma que o número de bits igual a 1 seja par. Do mesmo modo, é adicionada uma palavra no fim dos dados que corresponde a efetuar este método para cada coluna de bits.

Ora, quando é recebido um pacote, é verificada a existência de erros e, caso esteja algum bit errado, é possível detetar a sua localização e proceder à sua correção.

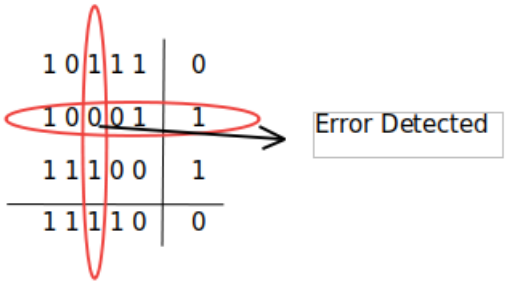

Fig. 5: Exemplo de deteção de erro

No entanto, apesar de ser um método simples de entender, há alguns casos em que mesmo que seja detetado um erro, é impossível precisar a sua localização; e outros casos raros onde não deteta erro mesmo que este tenha ocorrido.

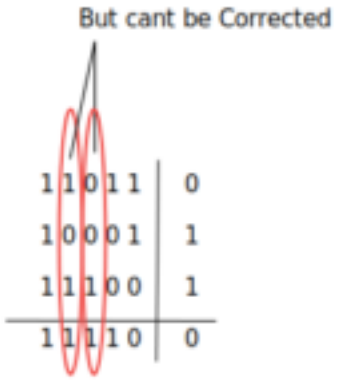

Fig. 6: Exemplo de deteção de erro mas impossibilidade de correção

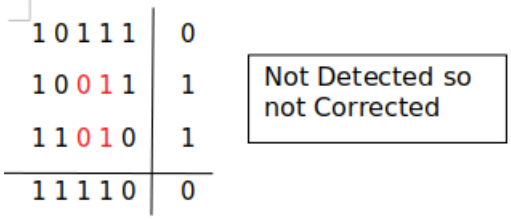

Fig. 7: Exemplo de ocorrência de erro sen deteção

Ainda assim, quando ocorre um erro que não pode ser corrigido, é efetuada a **retransmição** desse pacote pelo que realisticamente, o número de casos que não é corrigido será bastante pequeno.

Checksum

Além do *Two Dimensional Parity Check*, fizemos também uso da classe CRC-32 presente na *package* java.util.zip. Sempre que um pacote é enviado, é calculado o *checksum* dos seus dados e introduzido este resultado no *header* do FsChunkProtocol. Quando um pacote é recebido, é calculado o *checksum* do *payload* e é comparado com o *checksum* que veio no *header* desse pacote. Caso sejam diferentes, é sinal que ocorreu erro.

Na eventualidade de ser detetado um erro com este método, é efetuada a **retransmissão** desse pacote.

5 Testes e resultados

Teste 1 - Ficheiro sem fragmentação (1 Server)

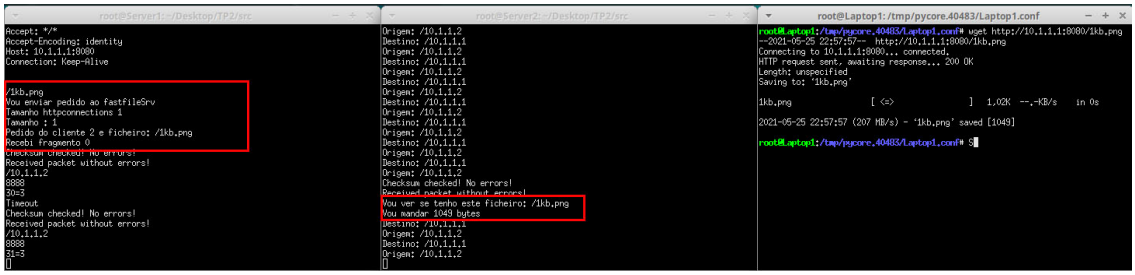

Fig. 8: Teste 1

Teste 2 - Ficheiro com fragmentação sem perdas (1 Server) - Não ocorre Retransmição

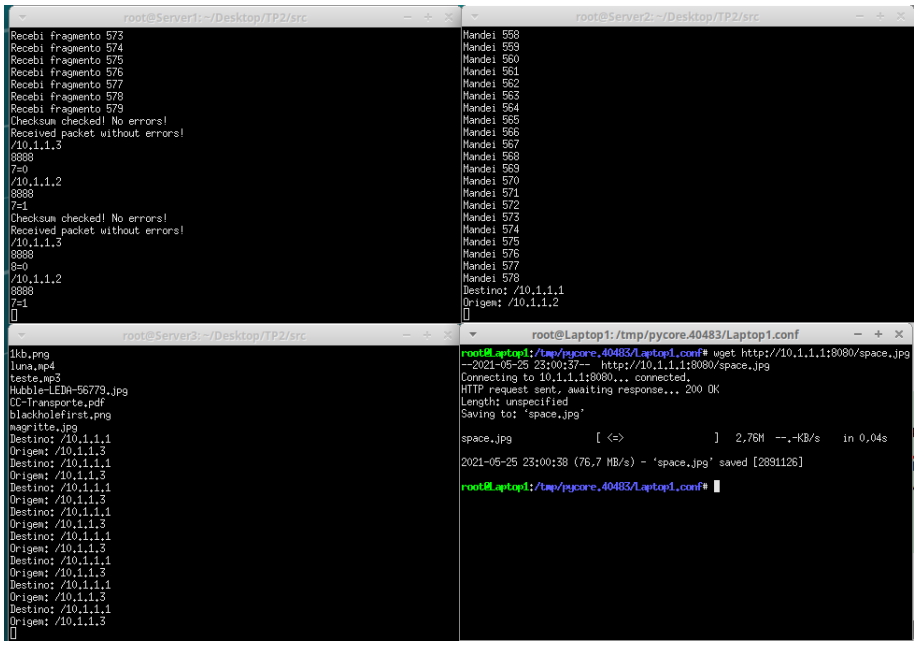

Fig. 9: Teste 2

Teste 3 - Ficheiro com fragmentação com perdas (1 Server) - Ocorre Retransmição

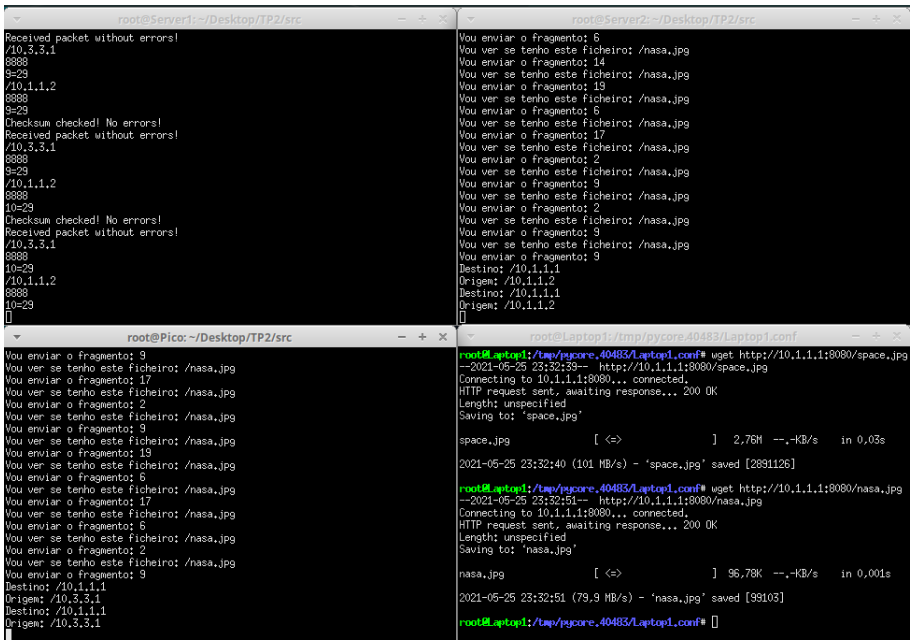

Fig. 10: Teste 3

6 Conclusões e trabalho futuro

Dado por concluído o presente relatório, apresentamos uma análise critica do trabalho realizado, os pontos fortes, as decisões mais importante, possíveis melhorias e considerações para trabalho futuro.

Comecemos pelos pontos fortes, dentro desta categoria consideramos como positivo o facto de poderem existir um número ilimitado de FastFileServer que por sua vez quando usados num numero consideravel possibilitam grande paralelismo, o controlo de acessos através da autenticação garante segurança isolando a rede interna de clientes desconhecidos, o balanceamento da carga contribui para um melhor desempenho, conseguimos lidar com vários clientes ao mesmo tempo e com a falha espontânea de servidores através dos metodos de Ping e Timeout, conseguimos ainda responder a qualquer tipo de ficheiro e independentemente do seu tamanho e condições da rede através de mecanismos de fragmentação e retransmissão, garantimos também a recessão ordenada e sem duplicações dos pacotes ao cliente, finalmente temos um metodo sofisticado e invulgar de deteção e correção de erros chamado twoDParityCheck. Existem também alguns aspetos que podiam ser melhorados, visto tratar-se de um ambiente académico não estamos a garantir a encriptação das mensagens e devido à solução adotada não achamos relevante o uso de metadados uma vez que o gateway sabe aquando da autenticação do server quais os ficheiros que este tem, o tamanho do payload poderia também não ser conhecido a priori pelo FastFileServer e, como não estamos a enviar a confirmação de autenticação do server pelo que, eventualmente, esta mensagem poderia ser perdida devido a *packet loss*, apesar de ser um cenário bastante raro.

No cômputo geral, achamos que o esforço aplicado durante a realização do trabalho é notável e consideramos o resultado final bastante satisfatório, no futuro de forma a aumentar o desempenho poderíamos paralelizar os pedidos, provenientes dos clientes, do gateway para vários FastFileServers e passar a execução sequencial dos pedidos nos FastFileServer para paralela, contudo assumimos que com um N suficiente de servidores a carga estará bem distribuída e irá simular um bom desempenho para uma taxa de pedidos aproximada de N.

Apesar de termos referido a deteção e correção de erros e destes terem sido codificados e estarem funcionais, a implementação destes métodos no FSChunkProtocol revelou-se bastante trabalhosa (principalmente o 2DParityCheck) uma vez que tinham de ser feitas conversões de bytes para bits e vice-versa pelo que decidimos que seria melhor entregar uma versão do trabalho sem estas funionalidades. Contudo, estamos confiantes que dado um pouco mais de tempo conseguiríamos implementar estes métodos.