Comunicação por Computadores

TP2 - Gateway Aplicacional e Balanceador de Carga sofisticado para HTTP

Grupo 9 - PL4

Ana Luísa Lira Tomé Carneiro - A89533
Henrique Manuel Ferreira da Silva Guimarães Ribeiro - A89582
Pedro Almeida Fernandes - A89574

Universidade do Minho, Departamento de Informática, Braga, Portugal

Resumo O seguinte trabalho realizado para a Unidade Curricular de Comunicação por Computadores tem como objetivo a criação de um *gateway* denominado de **HttpGw**, que com recurso ao protocolo **HTTP**, consiga responder a múltiplos pedidos ao mesmo tempo. Para obter respostas a estes pedidos, o *gateway* com recurso a um protocolo por nós desenvolvido (**Fs Chunk Protocol**) irá recorrer a um conjunto de servidores dinâmicos designados por **FastFileSrv** que contém os ficheiros pretendidos. Todo o programa implementado seguindo os objetivos anteriormente referidos, foi realizado em java e testado na topologia CORE fornecida pelos docentes.

Keywords: Gateway · UDP · HTTP.

1 Introdução

Os *gateways* têm diversas razões que motivam a sua existência, tais como a segurança através do isolamento da rede interna e do controlo de acesso, e desempenho devido a uma redução de interações e balanceamento de carga. Um *gateway* operando na camada 7 tem como função desencapsular dados de um protocolo e encapsular num outro, o que o torna muito flexível possibilitando fazer operações de múltiplas formas.

Deste modo foi-nos desafiado implementar um *gateway* (HttpGw) que faria a ponte entre clientes e os seus servidores de suporte FastFileSrv. Os clientes que pretendem receber determinados ficheiros necessitam de os pedir ao ao HttpGw através de um protocolo HTTP onde declaram o ficheiro que pretende receber. De seguida o *gateway* vai pedir esses mesmo ficheiros aos seus servidores através do uso de um pacote FS Chunk Protocol. Dependendo do tamanho do ficheiro pedido, os servidores irão enviar um ou mais pacotes deste mesmo tipo para o HttpGw, que no fim da sua receção fornece a estas uma confirmação da receção dos mesmos. Por fim o *gateway* desencapsula estes pacotes, junta-os na sua ordem correta e envia por fim ao cliente recorrendo ao protocolo HTTP.

2 Arquitetura da solução

A implementação da arquitetura passou por criar duas classes principais o HttpGw e o FastFileSrv. Dentro da primeira classe é disponibilizada a classe ClienteListenner que vai estabelecer a ligação HTTP com o cliente, a classe ServerUpdater que tem como objetivo estabelecer a ligação com os FastFileSrv e uma outra classe GwWorker que vai processar os pedidos HTTP REQUEST.

Na classe FastFileSrv é onde vão ser processados os pedidos vindos do *gateway* e onde o conteúdo do ficheiro pedido vai ser enviado de volta ao *gateway* quer este seja fragmentado ou não.

Existem ainda 2 classes que visam estabelecer as conecções UDP quer da parte do servidor (*Receiver*) quer da parte do *gateway* (*Transmitter*). Além disso, foi criado uma classe package que representa o pacote UDP que é enviado na rede entre o *gateway* e o servidor. Na figura abaixo encontra-se a representação das classes e como estas estabelecem ligação entre si.

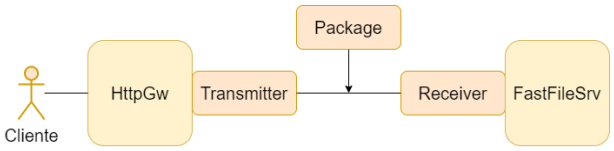

Figura 1: Demonstração Classes

2.1 Início de ligação

No HttpGw foi criado uma *thread* cujo objetivo é escutar a porta 12345 para estabelecer ligação com os FastFileSrv. Quando esta thread recebe um *DatagramPacket* vai ler o seu conteúdo e armazenar na pool de servidores (disponível no HttpGw) a porta especifica onde vão ser feitas as futuras conexões, o endereço onde se encontra o FastFileSrv e por fim se este se encontra disponível ou não para pedidos.

Este processo pode ser feito a qualquer momento sem que seja necessário restabelecer ligações com os servidores que já estejam presentes na pool.

Para processar os pedidos enviadas por HTTP REQUEST foi criado uma *thread* que tem como responsabilidade criar uma *thread* por cada pedido HTTP. Cada fio criado vai se conectar com um servidor disponível na pool de servidores.

2.2 Envio de pedidos

Após o inicio de ligação tanto o *gateway* como os fast file servers estão disponíveis para processar pedidos do cliente, sendo que cada servidor esta disponível para aceitar pedidos sequencialmente. Estes pedidos são enviados em formato HTTP REQUEST e tem como objectivo descarregar vários ficheiros que estejam disponíveis no computador. Foi assumido que todos os servidores tem acesso aos mesmos ficheiros e que estes existem e estão disponíveis para descarregar.

O *header* dos pedidos HTTP vão ser processados no *gateway* obtendo, no fim, o nome do ficheiro a descarregar. Este ficheiro será enviado ao fast file server através de um canal UDP utilizando o protocolo FS Chunk

Protocol. Assim, cada pedido é atribuído a um único servidor, a não ser que este se desconecte do gateway, e nesse caso o pedido é imediatamente entregue a um outro servidor. O conteúdo será lido pelo fast file server e será enviado de volta ao gateway que vai enviar em formato HTTP RESPONSE o conteúdo do ficheiro pedido.

2.3 Fragmentação

De modo a realizar a fragmentação de ficheiros de maiores dimensões decidimos no nosso protocolo usar campos que nos ajudassem nessa tarefa, entre os quais um boleano fragmentado, um inteiro resto, um inteiro *idPackage* e inteiro *offset*.

Do lado do *gateway* quando se recebe uma mensagem começa-se por inicializar um *array* de pacotes com o tamanho equivalente ao números de pacotes que irá receber, que de seguida é adicionado ao *map* pedidos que associa o *idPackage* corresponde ao identificador do pedido a cada *array* de pacotes. A cada pacote recebido recorrendo ao seu *offset* é possível adicionar o mesmo no lugar correto do *array*. Após um determinado tempo (*timeout*), caso verifique que o array dos pacotes tem "buracos", isto é, não recebeu todos os pacotes de um determinado ficheiro, o *gateway* irá enviar uma mensagem ao servidor pedindo os pacotes em falta. No caso de verificar que tudo ocorreu da melhor forma irá enviar um pacote ao servidor com o campo *ack* a 1, mostrando assim ao servidor que tudo chegou com sucesso.

Do lado do servidor ele consoante o tamanho do ficheiro, irá dividi-lo em pacotes de 60 000 *bytes* até chegar ao último pacote cujo tamanho vai ser o tamanho restante do ficheiro. Após o envio este vai ficar à espera de receber um pacote com *ack* postivo, caso isto não aconteça, ele vai ter em consideração o *offset* a partir do qual deve ler o ficheiro ler o identificador e reenviar o fragmento especifico de um determinado pedido identificado pelo *idPackage* .

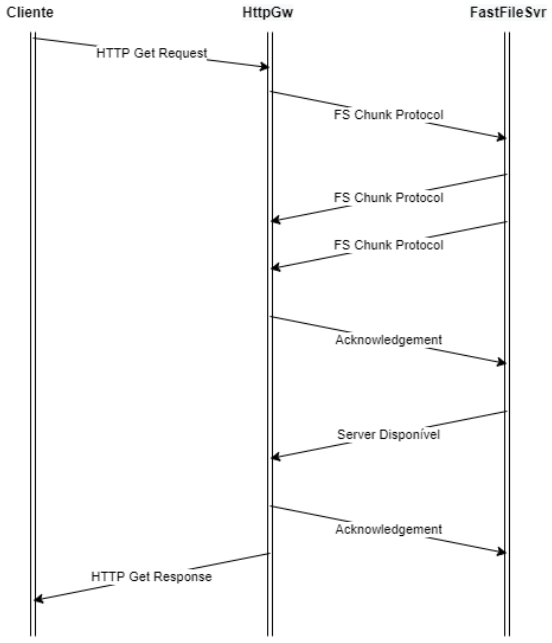

Figura 2: Envio de ficheiro composto por 2 fragmentos

2.4 Retransmição

Durante a reacção do conteúdo enviado pelo servidor ao *gateway* é possível que haja perdas ao longo de todo este processo de transmissão, sendo deste modo necessário retransmitir caso o *gateway* detete perdas nos pacotes que recebe.

Como forma de permitir retransmissão decidimos implementar uma *pool* que, à medida que o *gateway* recebe um fragmento, este vai ser adicionado a um *array* que contém todos o fragmentos recebidos de um mesmo *id* de pedido. Além disso, este procedimento é realizado com recurso ao *timeout*. Caso o primeiro fragmento não chegue ao *gateway* ao fim do *timeout*, este vai reenviar o ficheiro novamente ao *fast file server* que voltará a enviar o conteúdo total do ficheiro. Caso haja fragmentação então o *gateway* vai esperar que o servidor envie todos os fragmentos mesmo que este não estejam pela ordem correta até ao fim do *timeout*. No fim o *gateway* vai percorrer o *array* dos pacotes recebidos para aquele *id* de pedido e vai pedir a retransmissão dos pacotes que aí faltam. A imagem abaixo encontra-se a representação do funcionamento da retransmissão para o caso em que há fragmentação do pacote.

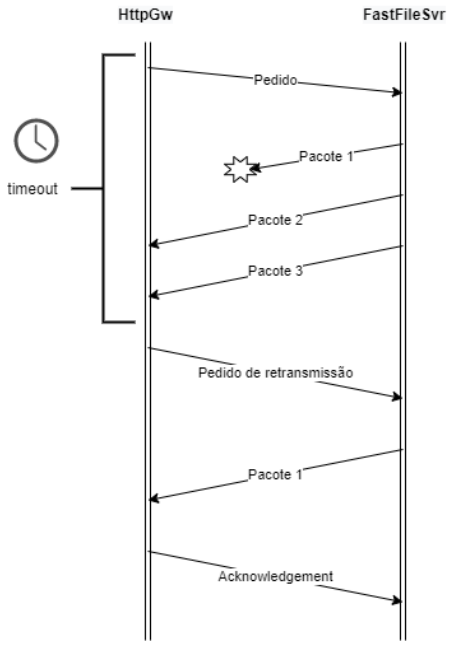

Figura 3: Falha na receção de pacotes

2.5 Controlo de fluxo em caso de Desconexão

Quando um servidor se desconecta do *gateway* a meio de uma transmissão é lançada uma exceção que será tratada de modo a restabelecer a conexão com outro servidor que esteja disponível. Após esse estabelecimento vamos voltar a pedir o ficheiro ao novo *Fast File Server* mantendo assim um controlo de fluxo em caso de desconexão. Com este procedimento conseguimos manter o paralelismo em caso de perda de conexão tornando todo o serviço eficiente e viável.

2.6 Paralelismo entre vários clientes

Como forma de permitir o pedido de ficheiros por vários clientes é necessário que haja vários servidores disponíveis ao mesmo tempo pois cada servidor só atende um cliente de cada vez. Assim desde que o número de servidores seja maior que o número de clientes é possível obter total paralelismo no sistema.

2.7 Fim de Pedido

Tendo já sido confirmada a integridade to ficheiro descarregado, é necessário descontinuar o processo de download e "libertar"o servidor na *pool* de servidores indisponíveis. Para este efeito é enviado um *acknowledgment* pelo cliente HttpGw ao servidor após o *gateway* receber um pacote "LIVRE"por parte do servidor. Assim, o servidor será atualizado na *pool* de servidores para o estado de disponível.

3 Especificação do protocolo

3.1 Formato das mensagens protocolares

O protocolo por nós desenvolvido (**FS Chunk Protocol**) funciona sobre UDP, é não orientado à conexão e não tem informação de estado. Este protocolo encontra-se materializado na classe *Package.java* contendo como seus atributos o seu cabeçalho e sua camada de dados.

Quanto ao cabeçalho existem duas *flags* fragmentado e *ack* que são representadas a partir de um bit a 0 ou 1 consoante esses predicados sejam verdadeiros ou falsos, três inteiros de controle (formados cada um por 4 *bytes*), o resto que informa no caso de uma fragmentação quantos pacotes faltam, o *idPackage* que contém um identificador do pedido e por fim o *offset*, também utilizado quando existe fragmentação para saber como "colar"os fragmentos no ficheiro pretendido. No total para cabeçalho são usados 12 *bytes* + 2 *bits* o que se reflete num total de 98 *bits*.

Sabendo que um pacote UDP contém 65 527 *bytes* de dados úteis decidimos usar então 60 000 *bytes* para o corpo de dados do nosso protocolo, de modo a tentar rentabilizar o espaço que obtemos a partir de nosso protocolo base. Decidimos fragmentar com 60 000 bytes para aproveitar o tamanho de dados do UDP. Poderíamos usar 512 bytes devido à fragmentação do *ip*, mas isso não é responsabilidade nossa.

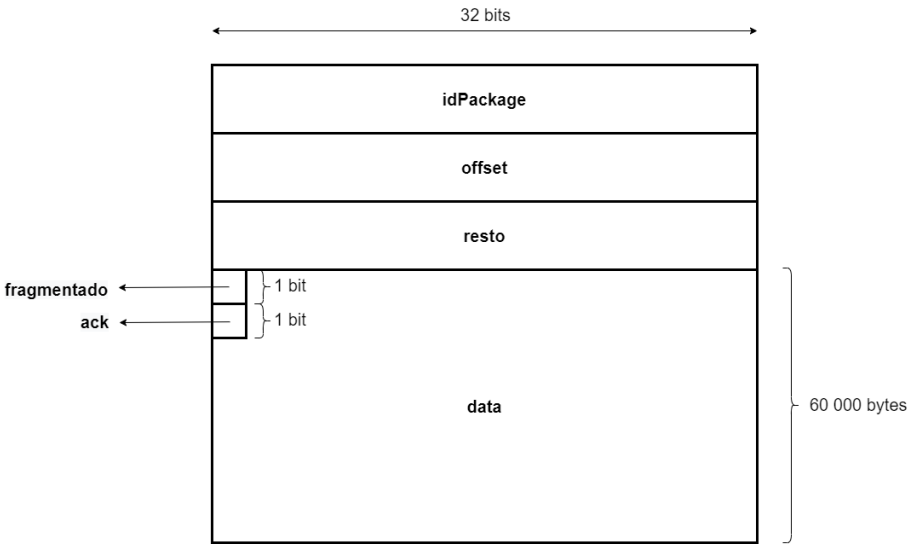

Figura 4: Constituição de um pacote FS Chunk Protocol

3.2 Interações

O nosso protocolo interage e estabelece a conexão entre o *gateway* e os seus servidores. Este pode conter no seu interior pedaços de ficheiros, *acknowledgments*, pedidos de ficheiros e pedidos de retransmissão. A interação entre estes dois começa com o pedido de um ficheiro do *gateway* a um servidor. De seguida este segundo envia os fragmentos ou o ficheiro completo dependendo do tamanho, e caso chegue tudo com sucesso o primeiro envia um *ack* ao segundo. Por fim o servidor envia um pedido a dizer que se encontra disponível para nova conexão e recebe depois um *ack* dessa mesma informação. Na imagem seguinte é possível ver graficamente o que foi dito anteriormente.

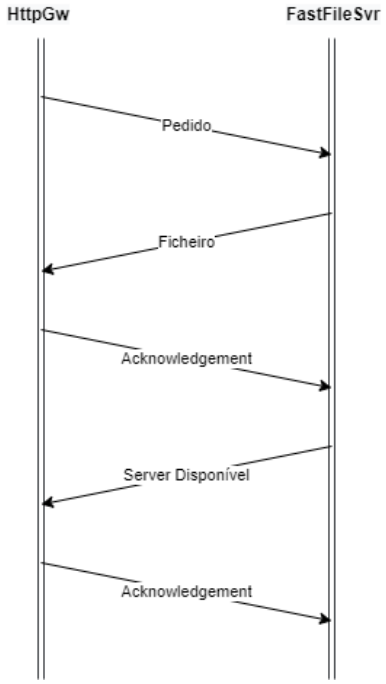

Figura 5: Interação entre *gateway* e servidor

4 Implementação

Para a implementação do trabalho foram utilizadas diversas bibliotecas pré existentes do java entre as quais:

- **java.io.IOException** : Para poder usar execeções em casos de *timeout* e de *crashes* de servidores;

- **java.util** : Foram usadas estruturas de dados como por exemplo *maps* e listas que foram implementados no HttGw para representar principalmente a pool de servidores e a pool de pedidos;

- **java.net** : Foram usados para manusear *sockets* UDP para transmissão da informação entre os fast file servers e o HttpGw;

- **java.lang.Thread.sleep** : Usado maioritariamente para testes ao programa.

4.1 Detalhes no Fs Chunk Protocol

No protocolo Fs Chunk implementado foi necessário dar valores distintos em cada campo como forma de representar e diferenciar as diversas transmissões durante todo o funcionamento do sistema.

- **Id Pedido** : O id do pedido representa o identificador único de cada pedido, contudo quando enviamos um pacote de conexão durante a conexão entre o gateway e o fast file server esse pacote tem com id o 0000. Já na altura do fim de transmissão quando um servidor envia o pacote a dizer que está disponível para aceitar outros pedidos o id do pedido passa a ser 9999.
- **Data** : O data representa o conteúdo do ficheiro que foi pedido ao *fast file server*, contudo este também funciona como mensagem que o *gateway* ou *fast file server* envia um para outro. Assim este conteúdo pode ser:
  - **PORT**: Pacote enviando durante a conexão entre o gateway e o fast file server com a porta onde o servidor se vai conectar para futuros pedidos vindos do *gateway*.
  - **RENVIO**: Caso seja um pacote para retransmissão de um ficheiro.
  - **ACK**: Após receber todo o conteúdo do ficheiro de forma ordenada é enviado um ACK para que servidor saiba que o ficheiro chegou ao seu destino de forma correta.
  - **LIVRE**: Pacote permite ao *gateway* obter a atualização da disponibilidade do servidor de ocupado para disponível.
  - **ACK LIVRE**: Pacote permite ao servidor saber que o *gateway* atualizou a disponibilidade do servidor.
  - **REENVIO LIVRE**: Caso seja um pacote para retransmissão de um pedido para atualização do estado do servidor.

5 Testes e resultados

Ao longo de todo o trabalho foram realizados diversos testes de modo a verificar o funcionamento do nosso programa. Na máquina nativa foram implementados alguns testes utilizando *Thread.sleep* que permitiram verificar o funcionamento das retransmissão ao longo do sistema. Como forma a realizar testes em contexto mais realista foi testado o código na topologia CORE. Em seguida encontram-se diversos testes realizados no CORE que comprovam com o o bom funcionamento do trabalho.

5.1 CORE

Teste 1 Neste teste temos um HttpGw a correr no *Server1* do Core e dois *Fast File Server*, um a correr a Venus e outro em Marte e dois clientes um no *laptop1* e outro no *laptop2*. Conseguimos ver que o *laptop1* pediu um ficheiro grande sendo este alocado ao servidor Marte. Durante o tempo em que Marte está ocupado com a transmissão do pedido do *laptop1*, o *laptop2* faz um pedido de um ficheiro pequeno sendo este alocado ao servidor Venus já que o servidor Marte está ocupado com a transmissão para o *laptop1*. Com este este conseguimos ver dois clientes a correrem em paralelo devido a existência de dois servidores. Na imagem a seguir encontra-se representado o teste realizado.

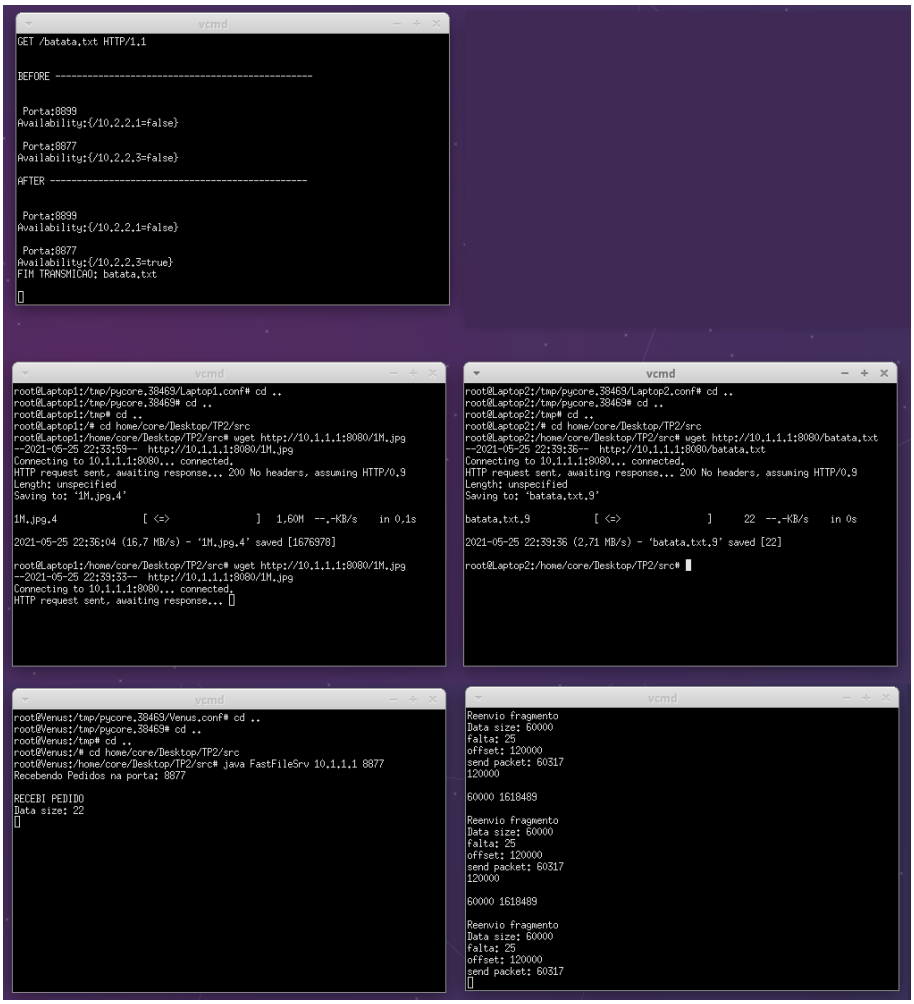

Figura 6: Teste 1

Teste 2 Neste teste temos um HttpGw a correr no *Server1* do Core e dois *Fast File Server*, um a correr a Venus e outro em Marte e um cliente no *laptop1*. Conseguimos ver que o *laptop1* pediu um ficheiro grande sendo

este alocado ao servidor Marte. Durante o tempo em que Marte está ocupado a transmissão é quebrada havendo perda de conexão tal como está no HttpGw "PERDA DE CONEXÃO". Logo após essa quebra a transmissão do ficheiro passa a ser alocado a um outro servidor disponível que neste caso é o servidor Venus. Na imagem a seguir encontra-se representado o teste realizado.

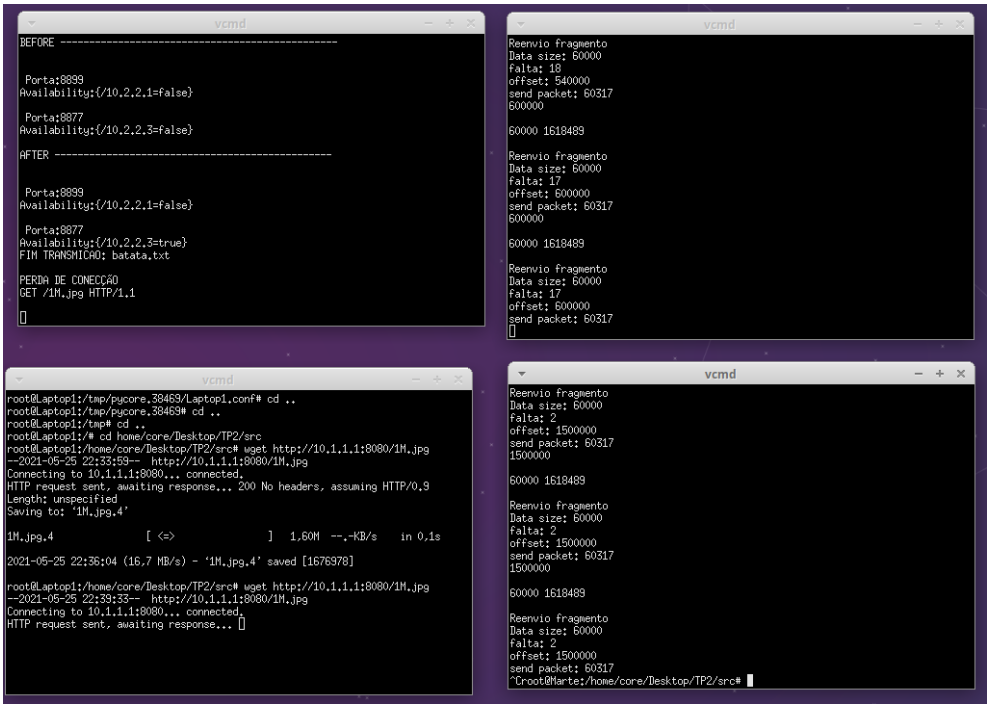

Figura 7: Teste 2

6 Conclusões e trabalho futuro

Dado por concluído o trabalho entendemos que realizamos com sucesso e de forma eficaz a tarefa proposta relativa à implementação de um *gateway* e de um protocolo para o satisfazer. O protocolo desenvolvido consegue não só enviar os ficheiros pretendidos quando tudo corre da melhor forma, mas consegue também recuperar de perdas e pacotes e também suceder à fragmentação dos ficheiros superiores à capacidade do protocolo com base em UDP elaborado.

No futuro poderíamos implementar diversas formas de segurança entre os clientes, gateway e os servidores, a partir do uso de autenticação para estabelecer conexões e também enviar os pacotes todos encriptados.

Em geral, consideramos que o balanço do trabalho é positivo, as dificuldades sentidas foram superadas e os requisitos básicos propostos cumpridos.

Bibliografia

[1] Kurose, James, Ross, Keith: Computer Networks - A Top-Down Approach. 7th edition, 2017

[2] Oracle DatagramSocket, https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html. Last accessed 25 May 2021

[3] The User Datagram Protocol (UDP), https://www.erg.abdn.ac.uk/users/gorry/course/inet-pages/udp.html: :text=A%20computer%20may%20send%20UDP,packets%20usually%20requires%20IP%20fragmentation.. Last accessed 24 May 2021