



Universidade do Minho
Escola de Engenharia

Departamento de Informática
Comunicações por Computador

Trabalho Prático nº 2

Gateway Aplicacional e Balanceador de Carga sofisticado para
HTTP

Ano Letivo 2020/2021

Grupo 1 - PL1

Ana Filipa Pereira A89589
Carolina Santejo A89500
Raquel Costa A89464

Conteúdo

1	Introdução	3
2	Arquitetura da Solução	4
3	Especificação do protocolo	5
3.1	Formato de Mensagens Protocolares (PDU)	5
3.2	Interações do <i>FShunkProtocol</i>	6
4	Implementação	8
4.1	Linguagem de Programação e Bibliotecas utilizadas	8
4.2	Autenticação de Servidores	8
4.3	Garantia de ordem na entrega das mensagens	8
4.4	Multiplexagem de clientes	9
4.5	Controlo de perdas	9
4.6	Pedidos HTTP GET e HTTP Response	11
5	Testes e Resultados	13
6	Conclusões e Trabalhos Futuros	16

1 Introdução

Neste trabalho, o desafio consistiu em implementar um gateway de aplicação, designado por HttpGw, que opera exclusivamente com o protocolo HTTP/1.1. Este gateway deveria ter algumas funcionalidades básicas tais como:

- Ser capaz de responder a múltiplos pedidos de clientes, recorrendo, para isso, a servidores designados por FastFileSrv e usando um protocolo específico denominado FShunkProtocol (que funciona sobre UDP);
- Os dados a disponibilizar aos clientes são ficheiros, podendo estes ser texto, áudio, imagem ou vídeo;
- Autenticação da origem dos pacotes transmitidos;
- Os servidores FastFileSrv servem todos os mesmos ficheiros, e o serviço não deve ser afetado, mesmo quando há servidores a ser arrancados ou parados;

Tendo isto em conta, neste relatório serão apresentados, de forma detalhada, a arquitetura da solução, o protocolo desenvolvido e os testes efetuados ao serviço desenvolvido.

2 Arquitetura da Solução

A arquitetura desenvolvida, divide-se em duas grandes partes, sendo a primeira a conexão dos servidores e dos clientes ao *HttpGw*, e a segunda o encaminhamento de *chunks* entre o *gateway* e os *FastFileSrv*.

Quando um cliente se conecta ao *gateway* é criada uma nova *thread* responsável pelo tratamento dos pedidos desse cliente e respectivas respostas. Por outro lado, quando um *server* se conecta a sua informação (endereço IP, ficheiros e porta) é guardada numa estrutura do *HttpGw*. Além disto, na classe *HttpGw*, existem duas *threads* em background, cuja função é ficar à espera de novos pedidos de conexão por parte dos clientes e dos servidores.

Por outro lado, quando o *gateway* envia a um *server* um pedido de ficheiro pode acontecer uma de duas situações: o *servidor* não possui esse ficheiro e lança-se uma exceção ou então o *servidor* contém o ficheiro requerido, enviando de volta para o *gateway*, um *ack* de recebimento de pedido bem como o primeiro pacote do ficheiro requerido.

Por último, a classe *ProtocolDemultiplexer* interceta os pacotes vindos dos vários servidores e organiza-os de forma a enviá-los para os respetivos clientes (cada *packet* contém o *id* do cliente a que é destinado).

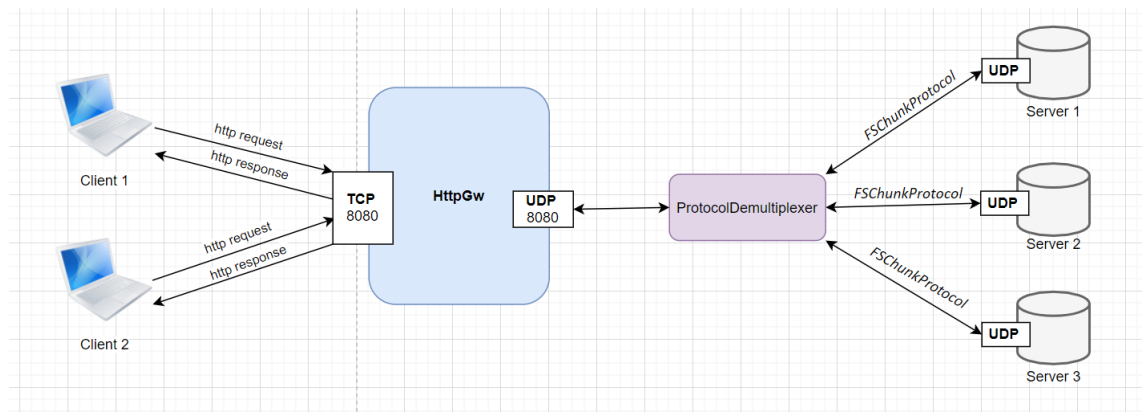


Figura 1: Arquitetura da Solução

3 Especificação do protocolo

3.1 Formato de Mensagens Protocolares (PDU)

Cada PDU é composto por 10 campos bem definidos sendo que o *header* contém 9 destes campos (todos os presentes na imagem abaixo exceto o conteúdo).

É de realçar que os campos correspondentes aos metadados (tamanho do ficheiro, nome do ficheiro, e data de modificação) e o número do pacote são opcionais.

O tamanho máximo de um PDU é 4096 bytes, no entanto o tamanho máximo do conteúdo é variável. Caso os campos dos metadados estejam preenchidos, o tamanho máximo do conteúdo a transmitir seja 321711 bits. Caso contrário, este valor aumenta para 321839 bits. Salienta-se que para o primeiro cálculo considerou-se um campo nome de ficheiro com tamanho de 32 bits.

Caso conteúdo a transmitir seja superior ao tamanho máximo do *payload*, ocorrerá fragmentação do mesmo.

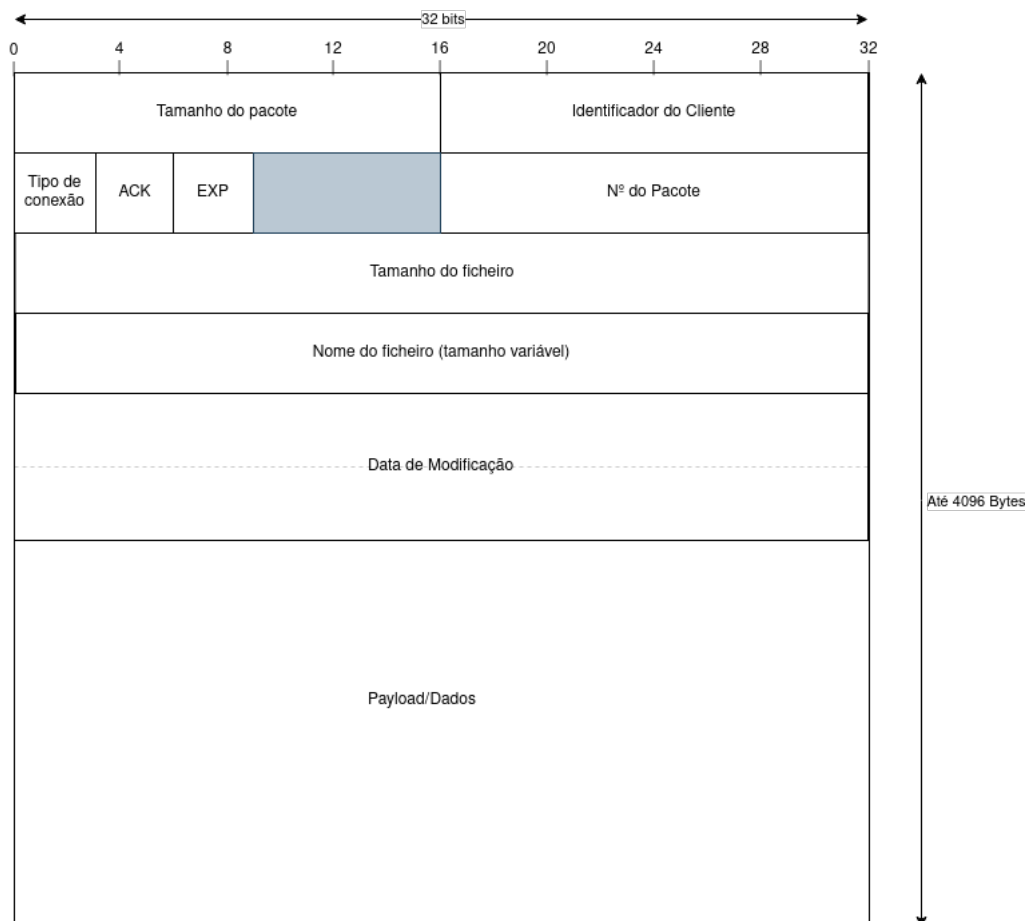


Figura 2: Packet

Tamanho do pacote: Campo com um tamanho máximo de 16 bits, que indica o tamanho do pacote a ser transportado, incluindo o header.

Identificador do Cliente: Campo com um tamanho máximo de 16 bits, que contém o id do cliente que fez o pedido que originou esse pacote. No início de conexão entre gateway e servidor, este último envia pacotes com id de cliente igual a -1.

Tamanho do pacote: Campo com um tamanho máximo de 3 bits, que indica o tipo de conexão a estabelecer, sendo que esta pode ser uma de seis opções: início de conexão, fim de conexão, retransmissão, pedido de dados, envio de dados ou envio de metadados.

Ack: Campo com um tamanho máximo de 3 bits, que indica que um pedido foi bem recebido. Exemplos de pedidos são, o pedido de início de conexão, de envio de dados ou de metadados.

Exp: Campo com um tamanho máximo de 3 bits, que indica quando uma exceção ocorre, como por exemplo, um ficheiro pedido não foi encontrado.

Número do pacote: Campo opcional, que indica a ordem com que os fragmentos originados devem ser organizados.

Tamanho do ficheiro: Campo com tamanho máximo de 32 bits que indica o tamanho total do ficheiro que está a ser transmitido.

Nome do ficheiro: Campo com tamanho variável, que indica o nome do ficheiro que se está a transmitir.

Data de modificação: Campo com um tamanho máximo de 64 bits, e que indica a data em que o ficheiro em questão foi modificado pela última vez.

Payload: Campo onde é armazenado o conteúdo a transmitir, ou parte dele, caso haja fragmentação.

3.2 Interações do *FSChunkProtocol*

Início de conexão

Quando um servidor se pretende conectar ao *gateway* envia-lhe um pedido de início de conexão, no qual envia o seu endereço, porta e nome de ficheiros. Caso o servidor não receba uma confirmação de que o pedido foi bem recebido, pede, no máximo de 3 vezes, a retransmissão do último pacote enviado pelo *HttpGw*.

Fim de conexão

Quando o servidor se pretende desconectar do serviço, envia ao *gateway* um pedido de fim de conexão. Caso o servidor receba a confirmação do recebimento do pedido por parte do *HttpGw*, então, desconecta-se e passa a estar inativo.

No entanto, caso o servidor não receba nenhuma confirmação do *gateway*, envia um pedido de retransmissão do último pacote enviado pelo *HttpGw*.

Retransmissão de dados

Algo relativamente comum no transporte de informação entre clientes e servidores é a perda de pacotes. De forma a diminuir o impacto desta situação, é implementado o mecanismo de retransmissão. Assim, quando um servidor está à espera de uma determinada informação do *gateway* e não a recebe, solicita que este a retransmita.

Pedido de dados

Sempre que um cliente solicita um ficheiro, é enviado um *HttpRequest* ao *gateway*, sendo que este efetua o parse, de forma a descobrir o nome do ficheiro. Posteriormente, o *HttpGw* seleciona um server que contenha o *file* desejado e envia-lhe um pedido de dados.

Envio de dados

Sempre o *gateway* efetua um pedido de dados a um servidor, verifica se aquilo que foi enviado pelo servidor corresponde à totalidade do ficheiro. Para isto, é feito um cálculo que determina quantos pacotes um dado ficheiro irá originar. Se o número de pacotes enviados pelo server for inferior aos que são esperados, é pedida a retransmissão de todos os pacotes.

Envio de metadados

O *gateway* pode solicitar ao servidor apenas os metadados de um ficheiro.

4 Implementação

4.1 Linguagem de Programação e Bibliotecas utilizadas

Como linguagem de programação optamos por *Java*, uma vez que é a linguagem na qual o grupo se sente mais confortável a trabalhar, além de ser aquela onde o grupo já tem experiência a trabalhar com *threads*, *sockets* e ainda com o modelo *Client-Server*.

É importante ainda referir que da biblioteca *java* foram importados alguns *packages* tais como o *net*, *util* e *io*. Do package *net* foram usadas as classes *DatagramPacket*, *DatagramSocket*, *InetAddress*, *NetworkInterface* e *UnknownHostException*, no package *util* as classes *concurrent.locks.Condition*, *concurrent.locks.Lock* e *concurrent.locks.ReentrantLock*, de forma a ser possível aplicar mecanismos de sincronização de *threads* quando as mesmas acedem a recursos partilhados. Além disso, do package *io*, foram utilizadas as classes *FileOutputStream*, *BufferedInputStream* e *BufferedOutputStream*.

4.2 Autenticação de Servidores

Para inicializar a conexão entre os *FastFileSrv* e o *Gateway* foi necessário efetuar uma autenticação entre as ambas partes. Para poder receber novos pedidos de conexão, o *HttpGateway* possui uma *thread* que está sempre à espera de novas mensagens. Quando um servidor se quer conectar, precisa de conhecer o IP e a porta do *Gateway* e efetuar um pedido de conexão. Ao efetuar este pedido, o servidor envia também as suas informações (*HostInfo*). Quando a mensagem é recebida, o *Gateway* guarda na estrutura *ServersConnected* as informações do novo servidor ao qual poderá posteriormente efetuar pedidos solicitados por clientes.

4.3 Garantia de ordem na entrega das mensagens

Para enviar ficheiros de tamanhos maiores do que o que um *chunk* permite, foi necessário efetuar a fragmentação do conteúdo por vários pacotes. Esta situação levou à possibilidade de haver problemas na ordem com que os pacotes de um determinado ficheiro são recebidos. Para isso foi utilizado o campo 'numero de pacote' que indica a sua posição na lista de todos os pacotes de um ficheiro.

Assim sendo, quando um servidor recebe um pedido de envio de ficheiro que existe no servidor, é calculado, em primeiro lugar, o tamanho máximo que o *payload* do pacote a enviar poderá ter, visto que este varia dependendo do espaço que o nome de ficheiro ocupa no cabeçalho. Sabendo esse valor e o tamanho total do ficheiro é assim possível calcular o número de pacotes a enviar para o *gateway*.

Por outro lado, o *gateway* terá de voltar a ordenar todos os pacotes recebidos (utilizando o campo 'numero de pacote') para assim poder criar o ficheiro correspondente. Previamente foi também necessário, com o tamanho do ficheiro e do *payload* máximo voltar a calcular o número de pacotes.

4.4 Multiplexagem de clientes

De modo a implementar a funcionalidade de o *gateway* poder responder a vários clientes em simultâneo foi necessário realizar a multiplexagem de clientes. Para que isto fosse possível, o *gateway* no início da sua execução inicializa uma *thread* que será responsável por intercetar todas as mensagens recebidas pelos servidores no socket UDP e organizá-las num buffer que agrupa pacotes de acordo com o id do cliente que está no seu cabeçalho.

Deste modo, cada vez que um cliente se conecta ao socket TCP, é inicializada uma thread responsável por responder ao seu pedido. Consequentemente, quando este efetua um pedido ao *gateway*, este encaminha-o diretamente para o servidor que lhe foi atribuído. Pelo contrário, para receber a resposta, a thread só pode receber as mensagens correspondentes ao seu id de cliente, logo é invocado o método *receive* da classe *protocolDemultiplexer* que acede ao buffer de todos os pacotes recebidos pelo *gateway* e retorna o primeiro da fila correspondente ao seu id. Se ainda não houver nenhum pacote na fila, é invocado o método *await* que vai esperar pela receção de novos *chunks*.

4.5 Controlo de perdas

De forma a evitar perdas de PDUs garantindo assim que todos chegam ao seu destino, é utilizado tanto o campo ACK do cabeçalho para indicar a receção correta de cada mensagem como também o tipo de mensagem 3 (Retransmissão) que caso não seja recebido um ack solicita a retransmissão do último pacote enviado. Para assumir que um ack não foi recebido, foi definido um tempo de timeout de 10 segundos e, se não for recebida a confirmação após este intervalo pode assim ser pedida a retransmissão do pacote.

Para o caso do pedido de conexão pelo servidor, se este não for recebido pelo *gateway* pode ser solicitada a retransmissão por parte do servidor no máximo 3 vezes e ainda assim caso não seja feita qualquer confirmação é cancelada a solicitação de conexão. Pode ser também solicitada a retransmissão de pacotes no caso do envio de ficheiros quando o *gateway* não recebe a totalidade dos pacotes correspondentes à fragmentação de um ficheiro.

```

    if (p.getAck() == 4 && p.getMessageType() == 5) {
        try {
            f = receiveNewFile(s, clientId, p);
            p = new Packet((short) clientId, 0, 5, 0);
            protocol.send(s, hostInfo, p);
        }
        catch (FragmentationPacketsNotReceived e) {
            p = new Packet((short) clientId, 3, 0, 0);
            protocol.send(s, hostInfo, p);
            p = receive(clientId);
            if (p.getAck() == 3 && p.getMessageType() == 5) {
                try {
                    f = receiveNewFile(s, clientId, p);
                    p = new Packet((short) clientId, 0, 5, 0);
                    protocol.send(s, hostInfo, p);
                }
                catch (FragmentationPacketsNotReceived ef) {
                    f = null;
                }
            }
        }
    }
}

```

Figura 3: Código referente ao envio de mensagem de retransmissão

```

Packet ack = receive(sock);
if (ack.getAck() == 5) {
    return 0;
}
else {
    if (ack.getMessageType() == 3) {
        for (int i = 0; i < numberOfPackets; i++) {
            packet = new Packet( clientId, 5,3,0,fI,buffers[i]);
            send(sock, gwInfo, packet);
            if (numberOfPackets>1) Thread.sleep(5); // Para enviar mais devagar
        }
        ack = receive(sock);
        if (ack.getAck() == 5) {
            return 0;
        }
    }
}
return -1;
}

```

Figura 4: Código referente à receção de mensagem de retransmissão

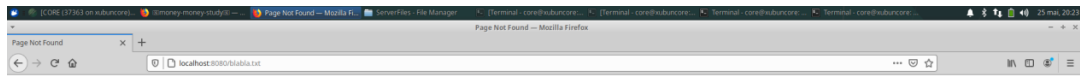
4.6 Pedidos HTTP GET e HTTP Response

Sendo o objetivo implementar um *Gateway* de aplicação que consiga operar exclusivamente com o protocolo HTTP/1.1, onde poderão ser enviados múltiplos objetos Web por cada ligação estabelecida entre o cliente e o servidor. Foi necessário implementar uma solução que eficientemente recebesse o pedido por parte do cliente/browser, fosse buscar o ficheiro pedido aos servidores, e que finalmente, retornasse uma resposta com o ficheiro ao cliente.

Portanto, em primeiro lugar, recorreu-se ao *parse* do HTTP *request* do cliente, de forma a obter-se o nome do ficheiro pedido. De seguida, através da implementação do algoritmo de *load balancing Round Robin*, foi possível distribuir os pedidos por cada servidor, isto é, escolher um servidor e pedir tudo ao mesmo sequencialmente, sendo que o *Round Robin* permite que os pedidos não sejam feitos sempre ao mesmo. Futuramente, pretende-se aplicar uma melhoria a este parâmetro tal como explicitado num dos seguintes tópicos. Posto isto, agora com o nome do ficheiro e o servidor escolhido, foi possível requisitar o ficheiro pedido pelo cliente ao servidor através dos protocolos e métodos implementados. Além disso, foi necessário também verificar o tipo de ficheiro (*.txt/.png* etc.) de modo a contruir o HTTP header com o Content-Type para indicar o tipo de media do conteúdo. Depois o conteúdo do ficheiro é passado para bytes de forma a conseguirmos enviar os dados solicitados ao cliente, e finalmente enviarmos uma resposta ao seu pedido, através de vários cabeçalhos HTTP construídos que possuem as informações necessárias a disponibilizar ao cliente, tal como: a data de envio da resposta, o conteúdo do ficheiro, e o tamanho do mesmo. Caso não seja encontrado o ficheiro pedido é lançada uma exceção, tal como para o caso de não haver servidores conectados, ou para qualquer outra interrupção que aconteça no sistema. No tratamento destas exceções são enviados HTTP responses do tipo "HTTP/1.1 404 File Not Found" ou "HTTP/1.1 502 Bad Gateway", de acordo com a correspondente exceção que tenha acontecido.

```
out.println("HTTP/1.1 200 OK");
out.println("Server: " + Address: " + server.address + "Port: " + server.port);
out.println("Date: " + new Date());
out.println("Content-type: " + content);
out.println("Content-length: " + fileLength);
out.println();
out.flush();
dataOut.write(fileData, 0, fileLength);
dataOut.flush();
```

Figura 5: Http Headers

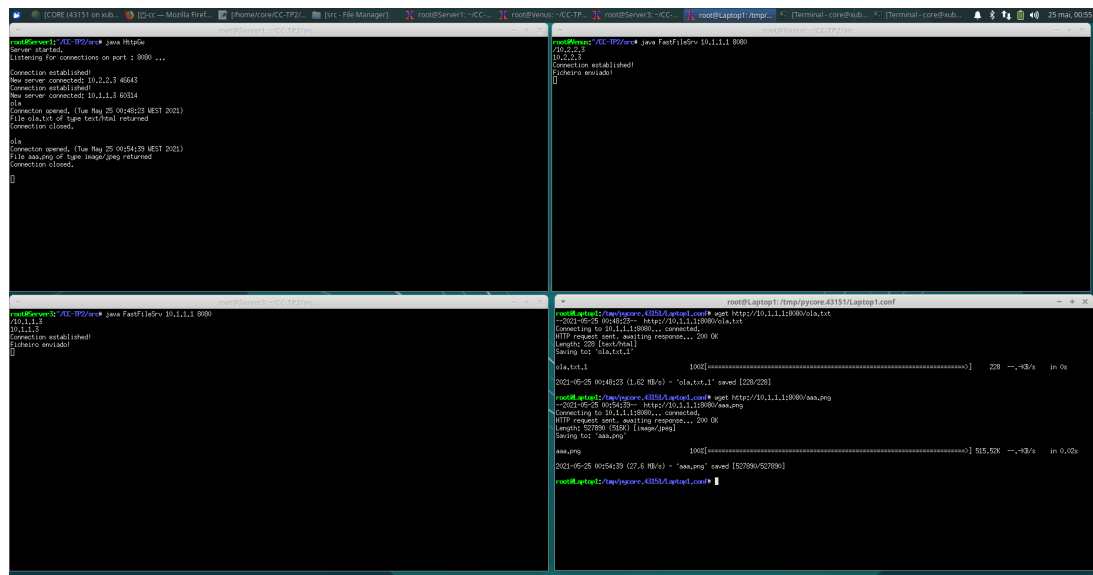


File Not Found
Sorry, but the file you were trying
to view does not exist.

Figura 6: Ficheiro pedido não existe

5 Testes e Resultados

De forma a testar a solução implementada foi utilizado o emulador Core, com a topologia fornecida pela equipa docente nas aulas práticas. Para tal, corremos um `HttpGateway` no nodo correspondente ao "Server1" da topologia, e ainda um ou mais `FastFileSrv` noutros nodos da topologia, tal como o "Server2" ou "Marte". De seguida nos nodos correspondentes ao "Laptop1" ou "Laptop3", foram feitos pedidos do tipo `wget` com o endereço `http` que contém o endereço, a porta e o ficheiro pretendido pelo cliente. Inicialmente, verificamos que estes testes permitiram-nos descobrir alguns erros na nossa implementação ao nível dos endereços de IP das interfaces dos hosts. Depois de corrigir, os testes foram realizados com sucesso. De seguida, apresenta-se alguns resultados obtidos durante a fase de testagem do trabalho prático.



```
root@Server1:~/02-IP2/ncv# java HttpGw
Server started.
Listening for connections on port 1 9090 ...
Connection established!
New server connected: 10.1.1.3 4043
Connection established!
New server connected: 10.1.1.3 5034
File object of type test/txt returned
Connection closed.
File object of type test/txt returned
Connection closed.
File object of type image.png returned
Connection closed.

root@Server2:~/02-IP2/ncv# java FastFileSrv 10.1.1.1 8080
10.1.1.3
Connection established!
File object returned

root@Laptop1:~/02-IP2/ncv# wget http://10.1.1.1:8080/vla.txt
--2021-05-25 00:54:39-- http://10.1.1.1:8080/vla.txt
Connecting to 10.1.1.1:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 205 [text/plain]
Saving to: 'vla.txt'

vla.txt 100%[=====] 205 --.-KB/s in 0s

2021-05-25 00:54:39 (1.62 MB/s) - 'vla.txt' saved [205/205]

root@Laptop1:~/02-IP2/ncv# wget http://10.1.1.1:8080/aaa.png
--2021-05-25 00:54:39-- http://10.1.1.1:8080/aaa.png
Connecting to 10.1.1.1:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 52789 (51KB) [image/png]
Saving to: 'aaa.png'

aaa.png 100%[=====] 515.53K --.-KB/s in 0.02s

2021-05-25 00:54:39 (27.6 MB/s) - 'aaa.png' saved [52789/52789]
root@Laptop1:~/02-IP2/ncv#
```

Figura 7: Teste realizado com 2 servidores e 1 cliente

```

root@Server1:/CC-TP2/src$ cd src/
root@Server1:/CC-TP2/src$ java HttpGw
Server started.
Listening for connections on port : 8080 ...

Connection established!
New server connected: 10.1.1.3 35076
Connection established!
New server connected: Mercurio 54111
Connection established!
New server connected: 10.2.2.3 48108
Connection opened. (Tue May 25 20:15:41 WEST 2021)
File ola.txt of type text/html returned
Connection closed.

Connection opened. (Tue May 25 20:15:55 WEST 2021)
File aaa.png of type image/jpeg returned
Connection closed.

Connection opened. (Tue May 25 20:16:11 WEST 2021)
File G19_Relatorio_Fase3.pdf of type application/pdf returned
Connection closed.

root@Laptop3:/tmp/pycore.37363/Laptop3.conf$
root@Laptop3:/tmp/pycore.37363/Laptop3.conf$ wget http://10.1.1.1:8080/ola.txt
--2021-05-25 20:15:41-- http://10.1.1.1:8080/ola.txt
Connecting to 10.1.1.1:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 228 [text/html]
Saving to: 'ola.txt'

ola.txt      100%[=====] 228 --KB/s  in 0s
2021-05-25 20:15:41 (744 KB/s) - 'ola.txt' saved [228/228]
root@Laptop3:/tmp/pycore.37363/Laptop3.conf$

root@Laptop2:/tmp/pycore.37363/Laptop2.conf$
root@Laptop2:/tmp/pycore.37363/Laptop2.conf$ wget http://10.1.1.1:8080/aaa.png
--2021-05-25 20:15:55-- http://10.1.1.1:8080/aaa.png
Connecting to 10.1.1.1:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 527890 (510K) [image/jpeg]
Saving to: 'aaa.png'

aaa.png      100%[=====] 515.52K --KB/s  in 0.02s
2021-05-25 20:15:55 (24.3 MB/s) - 'aaa.png' saved [527890/527890]
root@Laptop2:/tmp/pycore.37363/Laptop2.conf$

root@Laptop1:/tmp/pycore.37363/Laptop1.conf$
root@Laptop1:/tmp/pycore.37363/Laptop1.conf$ wget http://10.1.1.1:8080/G19_Relatorio_Fase3.pdf
--2021-05-25 20:16:11-- http://10.1.1.1:8080/G19_Relatorio_Fase3.pdf
Connecting to 10.1.1.1:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 158133 (1.2M) [application/pdf]
Saving to: 'G19_Relatorio_Fase3.pdf'

G19_Relatorio_Fase3 100%[=====] 1.53M --KB/s  in 0.02s
2021-05-25 20:16:11 (86.7 MB/s) - 'G19_Relatorio_Fase3.pdf' saved [158133/158133]
root@Laptop1:/tmp/pycore.37363/Laptop1.conf$

```

Figura 8: Teste realizado com 3 servidores e 3 clientes

Além dos testes especificados anteriormente, o sistema também foi testado fora do Core, obtendo os seguintes resultados:

```

core@xubuncore:~/CC-TP2/src$ cd CC-TP2/
core@xubuncore:~/CC-TP2$ cd src/
core@xubuncore:~/CC-TP2/src$ java HttpGw
Server started.
Listening for connections on port : 8080 ...

Connection established!
New server connected: 10.0.2.15 34039

core@xubuncore:~/CC-TP2$
core@xubuncore:~/CC-TP2$ cd CC-TP2/
core@xubuncore:~/CC-TP2$ cd src/
core@xubuncore:~/CC-TP2/src$ java FastFileSrv localhost 8080
Connection established!

```

Figura 9: Teste realizado com 1 servidor e 1 cliente fora do emulador Core

De seguida foi feito um Http Request através do browser produzindo o seguinte resultado:



Figura 10: Resultado Browser

De forma a testarmos o fim da conexão de um servidor, e da receção de uma notificação pelo HttpGateway, foi aplicado um *timeout()* de cinco segundos ao socket do servidor. Ao fim desse tempo verificamos que foi enviada uma mensagem com sucesso ao HttpGateway.

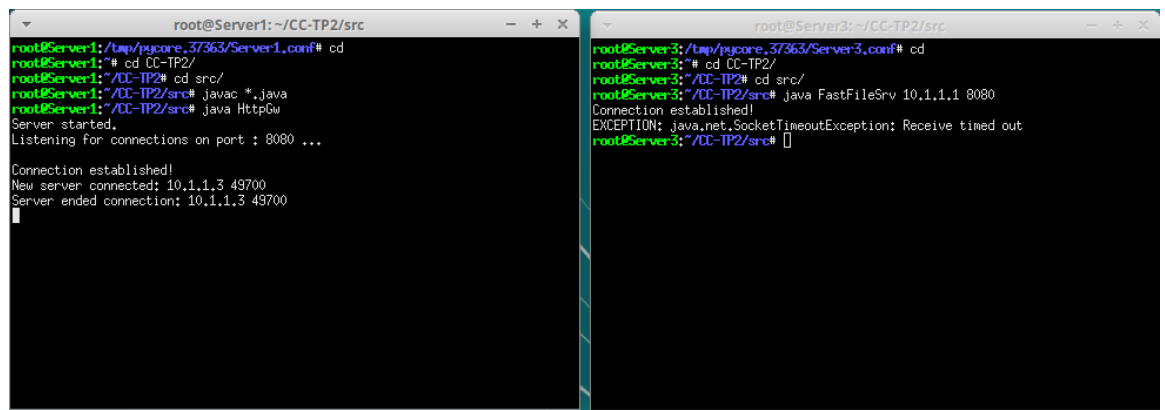


Figura 11: Fim de conexão do servidor

6 Conclusões e Trabalhos Futuros

O grupo considera que concluiu este segundo trabalho prático com sucesso, na medida em que foi desenvolvido um protocolo que satisfaz as funcionalidades pedidas no enunciado e já referidas neste relatório. No entanto, há espaço para melhorias, sendo que para trabalhos futuros, se poderia adicionar a encriptação dos PDU's, e um mecanismo de autenticação de servidores, uma vez que neste trabalho é permitido que qualquer servidor se conecte, deste que tenha conhecimento do protocolo. Seria, também, interessante, implementar a funcionalidade de controlo de congestão de tráfego e melhorar a nossa metodologia de controlo de erros. Outro factor que poderia ser mudado é o *load balancing*, sendo que em vez de pedir o ficheiro completo a um único servidor, pedir-se-ia vários *chunks* do ficheiro a vários servidores. Além disto, futuramente dever-se-ia adicionar uma funcionalidade que informa o *gateway* cada vez que é adicionado um novo ficheiro a um servidor.