

Gateway Aplicacional e Balanceador de Carga sofisticado para HTTP

João Correia^a, Rúben Cerqueira^b, and Tânia Rocha^c

^a A84414,

^b A89593,

^c A85176

25 de maio de 2021

Resumo

O seguinte relatório detalha a elaboração de uma aplicação ponte entre clientes e servidores na função de transferência de ficheiros. Esta aplicação mapeia os pedidos de cada cliente para o respetivo conteúdo solicitado, sendo este pedido aos servidores disponíveis e entregue ao respetivo cliente.

Keywords: *HttpGw*; *Protocolo TCP*; *FastFileSrv*; *Protocolo UDP*; *Protocolo HTTP*; *Ligações persistentes*

1 Introdução

No âmbito da unidade curricular de Comunicações por Computador foi proposto a implementação de um *gateway* de aplicação com o nome de **HttpGw**, cujo protocolo de operação é exclusivamente o protocolo HTTP/1.1 e tem como objetivo ter a capacidade de resposta a múltiplos pedidos, possivelmente simultâneos, de ficheiros a um determinado número de servidores considerados de alto desempenho designados por **FastFileSrv**. Estes últimos devem ter a capacidade de obter os metadados e os *chunks*, isto é, os blocos de dados do ficheiro pedido para enviar para o *gateway*. Esta comunicação é efetuada sobre **UDP**, enquanto os pedidos efetuados ao **HttpGw** ocorrem sobre **TCP**.

2 HTTPGW - Clientside

O gateway aplicacional, denominado por **HttpGw**, será responsável por receber pedidos **HTTP/1.1** vindo de vários clientes e será capaz de responder a cada um destes simultaneamente. Os pedidos dos clientes serão exclusivamente do tipo GET, isto é, cada cliente requisitará um ficheiro ao **HttpGw**, que este deverá depois devolver. Para tal ser possível, o gateway aplicacional terá

acesso a um conjunto de servidores denominados por **FastFileSrv**. A comunicação entre o gateway e os vários servidores de ficheiros é possibilitada através do protocolo **FSChunk**. Esta lógica é discutida em profundidade na secção 3.

A atual secção tem como propósito explicitar a forma como o HttpGw interage com o cliente, através de portas TCP que comunicam utilizando o protocolo HTTP e HTTPS.

Como tal, quando o HttpGw é iniciado, são criados dois sockets TCP:

- A porta 8080 estará responsável por responder a pedidos HTTP (ServerSocket)
- A porta 8081 estará responsável pelos pedidos HTTPS (SSLServerSocket)

2.1 GatewayWorker

Sempre que é estabelecida uma ligação TCP (que comunica usando o protocolo HTTP), entre o *gateway* e um *client* é criado um *GatewayWorker*, ou seja, uma thread responsável por gerir toda a comunicação entre ambas as entidades. Desta forma o *gateway* conseguirá atender um conjunto elevado de clientes. Várias outras medidas são aplicadas no gateway de forma a superar as limitações de hardware em que este está a correr, sendo até possível transferir ficheiros de tamanho superior à RAM livre no computador onde se encontra o gateway. Estas otimizações são apresentadas na secção 3.

O GatewayWorker irá fazer parse ao pedido do cliente, capturando os vários campos do pedido, dando especial atenção a dois campos: o ficheiro requisitado e o valor existente no campo *Connection*. Como o gateway suporta ligações persistentes, este campo será bastante importante, como explicado na subsecção 2.3.

Estado o parsing do pedido concluído, este é passado ao módulo FSChunk que irá primeiramente requisitar metadados do ficheiro a um servidor e, depois, caso exista, o requisite aos vários FastFileSrv's existentes. Este processo é explicado em detalhe nas secções 3 e 4.

Quando o pedido HTTP contém, no campo do ficheiro pedido, apenas *"/*", o gateway devolve o ficheiro *index.html*. Este ficheiro existe diretamente na diretoria do HttpGw, sendo enviado diretamente como resposta ao invés de requisitar metadados a um fastfilesrv e posteriormente requisitar chunks a vários servers, visto ser trabalho desnecessário.

2.2 Suporte do protocolo HTTPS

Para possibilitar o funcionamento do protocolo HTTPS foi utilizado um *self-signed certificate* isto é, um certificado SSL (Secure Socket Layer) que não foi assinado por uma entidade autorizada para tal, mas sim assinado pela mesma pessoa que o criou, utilizando o seu próprio *Root CA* (*Certification Authority Certificate*).

Por definição, um sistema não confia um certificado ssl que não foi assinado por uma entidade autorizada. Como tal, o Root CA Certificate deve ser adicionado ao conjunto de entidades confiadas pelo sistema operativo (para que aplicações como o *wget* e o *curl* confiem no certificado apresentado pelo HttpGw, visto que este está assinado pelo Root CA Certificate). Este CA Certificate deve também ser adicionado à lista de entidades confiadas pelo browser de forma a que seja possível estabelecer ligações HTTPS entre o browser e o gateway sem que o browser tente impedir o utilizador de aceder ao gateway.

Este certificado SSL utilizado pelo gateway encontra-se guardado dentro de uma *java keystore*, uma estrutura própria externa ao gateway responsável por guardar certificados e protegida por uma password. Esta password encontra-se guardada num ficheiro *.env* em vez de estar *hard-coded* no próprio código do gateway, de forma a garantir segurança e garantir também que o certificado não é usado para fins nefários.

2.3 Ligações persistentes

Como mencionado na secção anterior, o gateway aplicacional suporta pedidos HTTP/1.1 persistentes, sendo, portanto, possível reutilizar a mesma ligação TCP para vários pedidos de um cliente, ao invés de estabelecer uma nova por cada um, contornando o tempo gasto com o estabelecimento de uma.

Para a utilização de ligações persistentes, salienta-se a importância do cabeçalho **Connection** de um pedido HTTP. Este cabeçalho indica se a ligação deve ser reutilizada no final da transação atual. Se o cabeçalho tomar o valor **keep-alive**, a ligação será persistente, caso tome o valor **close**, a ligação não será reutilizada, sendo necessário estabelecer uma nova ligação para futuros pedidos vindos desse cliente.

O tempo definido para o timeout do socket TCP foi definido para 5 segundos, isto é, no final de uma resposta, caso o cliente faça um novo pedido dentro dos próximos 5 segundos a ligação será reutilizada. Caso contrário, o socket TCP é fechado e um novo deve ser aberto.

2.4 Chunked Transfer Encoding

A utilização de ligações persistentes causa dificuldades ao cliente, nomeadamente, ao tentar perceber onde acaba uma resposta HTTP e começa outra. Como tal, quando são utilizadas ligações persistentes, as respostas HTTP serão enviadas usando **Chunked Transfer Encoding**. Este é um mecanismo de *data streaming* que consiste em dividir a resposta HTTP em vários *chunks* que são enviados sequencialmente.

Neste tipo de respostas, deve-se adicionar o header **Transfer-Encoding: chunked**. O corpo da mensagem consiste portanto num conjunto de *chunks*. Cada chunk é precedido pelo

seu número de bytes representado em **hexadecimal**, seguido de um carácter de nova linha. São depois enviados os bytes do chunk correspondente também seguidos de um carácter de nova linha para assinalar o final do chunk. O chunk final de uma resposta é assinalado com um comprimento de 0 bytes e é composto apenas por um carácter de nova linha.

Segue um exemplo do corpo de uma resposta usando chunked encoding:

```
4\n (nº de bytes a enviar)
Wiki\n (chunk)
6\n (nº de bytes a enviar)
pedia \n (chunk)
E \n (nº de bytes a enviar - 10 bytes)
in \n (continuação do chunk)
\n
chunks.\n (chunk)
0\n (byte final - 0)
\n (chunk vazio final)
```

Esta resposta gerará o seguinte texto:

```
Wikipedia in

chunks.
```

3 HTTPGW - Serverside

Na lógica do servidor, mais concretamente do protocolo FSChunk, é possível fazer uma divisão deste em 2 funções: manipular pedidos de autenticação e desconexão de servidores e transferir os ficheiros requisitados pelos clientes.

Esta última função foi dividida em camadas, uma mais inferior relativa à comunicação entre o protocolo e os servidores, enquanto que a restante trata do processamento dos pedidos de busca à camada inferior e envio dos pedaços do ficheiro sequencialmente e ordenadamente ao cliente de modo a que este receba o ficheiro pedido rapidamente e fiavelmente.

3.1 Server Association Worker

Para uma aplicação de transferência de ficheiros são necessárias conexões a servidores para que estes consigam enviar os ficheiros pedidos pelo HTTPGW. Antes disso é crucial um mecanismo de autenticação e atendimento de pedidos de associação de servidores para a aplicação.

Como solução, é destinada uma thread que terá como única função, tratar de pedidos de servidores, ao que nomeamos de **Server Association Worker**. O fluxo de funcionamento é simples, quando um servidor se pretender associar à aplicação, esta thread receberá esse pedido, uma vez que está associada a um socket respetivo, e iniciará o processo de autenticação que será abordado com mais detalhe na secção 5. Depois da autenticação ser bem sucedida, a thread colocará a informação do servidor numa estrutura de dados própria de acesso rápido. Tal como servidores se podem associar à aplicação, os mesmos também terão possibilidade de desassociar, enviando o respetivo pedido, e posteriormente, a thread removerá a informação desse servidor da estrutura de dados da aplicação.

3.2 Protocolo FSChunk

Para que um pedido de um cliente seja concretizado é necessário um sistema rápido e organizado de modo a que não perca informação nem misture com os restantes pedidos concorrentes. Logo, foi criado o protocolo FSChunk, que funcionará concorrentemente para cada pedido. Cada vez que este protocolo é chamado, primeiramente, vai pedir a um servidor aleatório por meta informação sobre o ficheiro desejado.

Após obter as informações básicas sobre o ficheiro, é criado um ambiente adaptável, ou seja, é criado um número de fios de execução proporcional ao tamanho de ficheiro, de modo a que o trabalho seja dividido entre eles, promovendo a velocidade de processamento do pedido. Dito isto, para ficheiros maiores, a quantidade de threads será maior, dado que para ficheiros leves apenas uma thread é requisitada. Além disso é enviado um *chunk* contendo o cabeçalho da resposta HTML para o socket se o ficheiro existir, contendo informações como o tamanho do ficheiro e o seu tipo. Se o ficheiro pedido não existir é enviado o pacote ao cliente com a mensagem de erro 404, terminando assim o processo.

Após a criação do ambiente de transferência, a obtenção do ficheiro será dividida em offsets, igualmente distribuídos pelas várias threads criadas. Os fios de execução irão pedir ao servidor os offsets que lhes foram atribuídos, guardando o resultado numa estrutura de dados partilhada por todo o protocolo responsável pelo mesmo pedido. Essa estrutura constará num mapeamento de offset para informação em array de bytes. Ao mesmo tempo executará uma thread responsável por ler ordenadamente dessa estrutura de dados. Sempre que a mesma requisitar um offset da estrutura de dados, a thread recolherá a informação em bytes correspondente e enviará um

chunk para o socket de escrita para o cliente, apagando a entrada de seguida de modo a libertar memória do sistema. Este processo ocorrerá até o ficheiro ser completamente enviado para o socket, terminando o envio por chunks ao cliente, sendo enviada assim a resposta esperada pelo cliente.

3.3 FSChunk Worker

Cada pedido que o protocolo FSChunk faz que recorre a servidores é feito por meio de um FSChunk Worker. Este worker é responsável pela comunicação com os servidores e assim obter os pedaços ou metadados de ficheiros requisitados pelo protocolo FSChunk.

Cada vez que é feito um pedido de ficheiro, este worker irá preparar um socket UDP responsável para a comunicação com o servidor. Após a preparação do socket, no caso de se pretender obter chunks de ficheiros, é enviado um pacote com a flag GET, seguida pelo offset de leitura, o tamanho do chunk a requisitar e o nome do ficheiro, para um dos servidores disponíveis (ex: GET 400 1200 sample.txt). Na solicitação de metadados de ficheiros é enviado pacote com a flag INFO seguida do nome do ficheiro (ex: INFO sample.txt). O *worker* usa um método análogo ao Round Robin na seleção de servidores a comunicar. Este método consiste na transformação da estrutura de servidores numa lista que funcionará como uma fila. Cada vez que o worker precisará de obter um pedaço de um ficheiro de um servidor, selecionará o servidor que consta no início da fila, colocando-o no fim desta após a conclusão da comunicação. Este método visa prevenir a sobrecarga de trabalho dos respetivos servidores. Na questão de obtenção de metadados, como se tratam de pacotes pequenos, é apenas selecionado um servidor aleatório para a troca de dados. Uma vez que se trata de UDP, a fiabilidade é limitada, sendo assim responsabilidade da aplicação de garanti-la. Para solucionar este problema recorreu-se aos timeouts, ou seja, cada vez que é enviado um pacote para o servidor, o worker espera 2 segundos. Ao fim deste tempo, se não houver qualquer resposta do servidor, o pacote é enviado novamente, repetindo este procedimento até receber resposta do servidor. Permanecendo na questão da fiabilidade, mais propriamente do conteúdo, pensamos implementar uma funcionalidade de checksums para a identificação de pacotes corrompidos, mas como o objetivo do desenvolvimento desta aplicação incide na rapidez de transferência de ficheiros, decidimos não colocar esse overhead na troca de pacotes para não se representar num desaceleramento do processamento de pedidos.

4 FastFileSrv

Com o objetivo de tratar de pedidos de ficheiros, a implementação do **FastFileSrv** foi evoluindo com o propósito de receber pedidos simples aos quais o servidor responderia "à letra",

isto é, enviaria os bytes de um determinado ficheiro conforme recebido no pedido sem questionar de qualquer forma o mesmo.

4.1 Message Data

Numa fase inicial foi tomada em consideração dois tipos de pedidos que se destingem com uma *flag* no cabeçalho. Estas *flags* descrevem este mesmo tipo de pedido da seguinte forma:

- **INFO** - Aquando da presença desta *flag* no pedido, o servidor considera que foi efetuado um pedido de *metadados* de um determinado ficheiro, cujo respetivo nome é enviado neste mesmo pedido. Estes *metadados* correspondem à confirmação da existência do ficheiro e em caso afirmativo, o seu tamanho e o seu tipo (ex: INFO sample.txt).
- **GET** - No caso de pedidos com a *flag* GET, o servidor considera, à partida, que o ficheiro a obter existe, isto porque o **HttpGw** apenas deve efetuar o pedido após receber a confirmação da existência do ficheiro no pedido do tipo INFO. Daí estes pedidos contém, no cabeçalho, um *offset*, o tamanho do pacote a enviar e o nome do ficheiro (ex: GET 400 1200 sample.txt).

O *offset* corresponde à parte do ficheiro, em *bytes*, a partir do qual deve ter início o envio o pacote de *bytes* a ser enviado de volta. Este pacote terá exatamente o tamanho descrito em *size*.

Pequenas contribuições para a procura do ficheiro pedidos, passam pela implementação de mecanismos de confirmação da existência de um determinado ficheiro mesmo sem a extensão.

4.2 Request Handler

Com o objetivo de elevar a capacidade de resposta a pedidos e a rapidez de velocidade dos mesmos, foi implementado um sistema de concorrência a partir de uma *thread pool*. Esta *pool* permite também que não sejam gerados tantos recursos e posteriormente destruídos devido à capacidade de reutilização de *threads* que se encontrem livres.

Aquando cada pedido recebido do lado do *gateway*, o **FastFileSrv** utiliza uma *thread* que se encontre disponível e atribui o pedido para que o **RequestHandler** possa tratar do mesmo.

Por sua vez, o **RequestHandler** faz a análise do pedido efetuado através das *flags* nele presentes e efetua o seu tratamento conforme. Dependendo da *flag* é então feita uma busca

pelos metadados com auxílio à classe *Message Data* ou então é feita a leitura dos *bytes* desejados e enviado de volta para o *gateway*.

Em ambas as situações é necessário ter guardado os dados da porta e do endereço do *gateway*. Estes são armazenados aquando a construção da classe em cada *thread*.

4.3 Quitter

É importante que, quando um servidor do tipo **FastFileSrv** seja desligado, o *gateway* saiba do acontecimento e retire da sua lista de servidores disponíveis para fazer pedidos. Para tal foi implementada uma classe *Quitter* que, tal como a classe *Request Handler*, é concorrente e permite que seja associada uma *thread* a cada **FastFileSrv** que tem como função desligar o servidor, enviando a informação ao *gateway* de que se vai desligar. Em resposta o servidor responde com uma confirmação de que o **FastFileSrv** se pode desligar, permitindo assim, consistência e atualização da listagem dos servidores disponíveis para a recolha de ficheiros ou metadados.

5 Mecanismo de autenticação

A autenticação de FastFileSrv's no gateway aplicacional será construída em cima de encriptação RSA, como mencionado na secção 3. Quando o gateway é iniciado, é aberto um *port* UDP somente para autenticação de servidores. Também é gerado, pelo gateway, um par chave pública/chave privada, a ser utilizado para encriptação durante o processo de autenticação.

Para que a autenticação de um servidor seja válida, este deve apresentar um token secreto que o gateway irá comparar com o seu próprio. Caso sejam iguais a autenticação é autorizada. Tanto o gateway como o servidor guardam o seu secreto/token num ficheiro *.env* de forma a impedir que este seja hard-coded em código open-source.

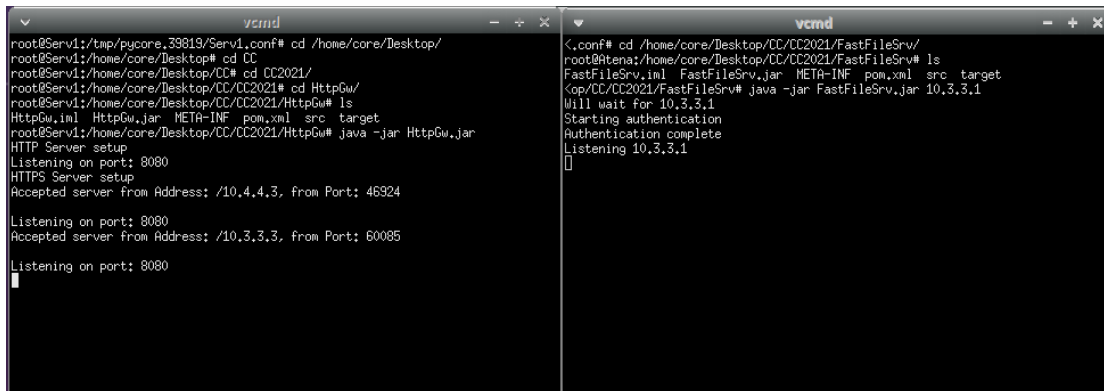
O processo de autenticação seguirá os presentes passos:

1. O FastFileSrv que pretenda iniciar o processo de autenticação envia um byte à porta UDP responsável pelo processo de autenticação no gateway;
2. HttpGw responde com a sua chave pública;
3. FastFileSrv lê o seu secreto do seu ficheiro *.env*, cifrando-o com a chave pública do gateway e envia-o;
4. HttpGw decifra a mensagem usando a sua chave privada;
5. HttpGw compara o secreto decifrado com o secreto existente no seu próprio ficheiro *.env*. Caso sejam iguais, o gateway adiciona o FastFileSrv à lista de servidores autorizados;

6. HttpGw responde ao FastFileSrv com o veredicto da autenticação: "Granted"/"Denied".

6 Testes e Resultados

A nível de testes na topologia *core*, foi inicializado um servidor **HttpGw** no servidor 1 da mesma, com o endereço 10.3.3.1. Foram também ligados dois servidores **FastFileSrv** noutros dois servidores e, para saberem a que endereço deviam iniciar a autenticação, foi descrito o endereço do *gateway* como argumento.

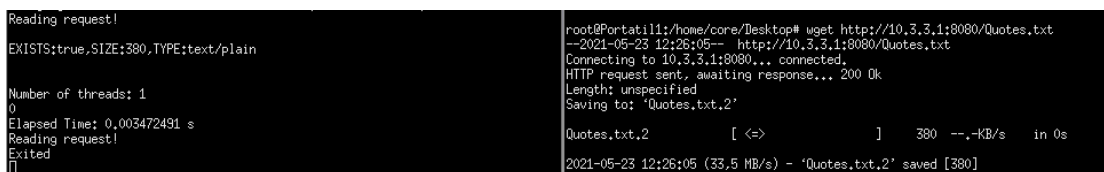


```
vcmd
root@Serv1:/tmp/pycore.39819/Serv1.conf# cd /home/core/Desktop/
root@Serv1:/home/core/Desktop# cd CC
root@Serv1:/home/core/Desktop/CC# cd CC2021/
root@Serv1:/home/core/Desktop/CC/CC2021# cd HttpGw/
root@Serv1:/home/core/Desktop/CC/CC2021/HttpGw# ls
HttpGw.iml HttpGw.jar META-INF pom.xml src target
root@Serv1:/home/core/Desktop/CC/CC2021/HttpGw# java -jar HttpGw.jar
HTTP Server setup
Listening on port: 8080
HTTPS Server setup
Accepted server from Address: /10.4.4.3, from Port: 46924
Listening on port: 8080
Accepted server from Address: /10.3.3.3, from Port: 60085
Listening on port: 8080

vcmd
root@Htena:/home/core/Desktop/CC/CC2021/FastFileSrv# ls
FastFileSrv.iml FastFileSrv.jar META-INF pom.xml src target
root@Htena:/home/core/Desktop/CC/CC2021/FastFileSrv# java -jar FastFileSrv.jar 10.3.3.1
Will wait for 10.3.3.1
Starting authentication
Authentication complete
Listening 10.3.3.1
```

Figura 1: Ligação do *gateway* e autenticação de um servidor *FastFileSrv*.

Para testar o download de ficheiros, foi pedida a transferência de apenas um ficheiro e, no teste seguinte, de dois ficheiros simultaneamente (figuras 2 e 3). Como podemos observar de seguida, os testes estão em bom funcionamento:



```
Reading request!
EXISTS:true,SIZE:380,TYPE:text/plain
Number of threads: 1
Elapsed Time: 0.003472491 s
Reading request!
Exited

root@Portatil1:/home/core/Desktop# wget http://10.3.3.1:8080/Quotes.txt
--2021-05-23 12:26:06-- http://10.3.3.1:8080/Quotes.txt
Connecting to 10.3.3.1:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified
Saving to: 'Quotes.txt.2'
Quotes.txt.2 [ <=> ] 380 --KB/s in 0s
2021-05-23 12:26:05 (33,5 MB/s) - 'Quotes.txt.2' saved [380]
```

Figura 2: Download de um ficheiro.

As figuras 4 e 5 ilustram a transferência de um ficheiro de 428MB utilizando tanto HTTP como HTTPS. Como em ambas as instâncias se utilizou **Chunked Transfer Encoding**, sabe-se que os valores apresentados pela ferramenta **wget** no que concerne as velocidades de transferência e tempo decorrido são verídicos e se referem ao tempo utilizado pelo gateway aplicacional, algo que não aconteceria noutros tipos de envio de dados por HTTP.

```
EXISTS:true,SIZE:380,TYPE:text/plain

Number of threads: 1
0
Elapsed Time: 0.009918909 s
Reading request!
Exited
New socket
Debug: got new client Socket[addr=10.1.1.1,port=53820,localport=8080]
Reading request!
EXISTS:true,SIZE:1445045,TYPE:text/plain

Number of threads: 1
141
Elapsed Time: 0.78587412 s
Reading request!
Exited
[]

<.1:8080/Quotes.txt && wget http://10.3.3.1:8080/test
--2021-05-23 12:27:06-- http://10.3.3.1:8080/Quotes.txt
Connecting to 10.3.3.1:8080... connected.
HTTP request sent, awaiting response... 200 Ok
Length: unspecified
Saving to: 'Quotes.txt.3'

Quotes.txt.3      [ <=>          ] 380 --.-KB/s   in 0.009s

2021-05-23 12:27:06 (43.2 KB/s) - 'Quotes.txt.3' saved [380]

--2021-05-23 12:27:06-- http://10.3.3.1:8080/test
Connecting to 10.3.3.1:8080... connected.
HTTP request sent, awaiting response... 200 Ok
Length: unspecified
Saving to: 'test'

test              [ <=>          ] 1.38M 1.76MB/s   in 0.8s

2021-05-23 12:27:07 (1.76 MB/s) - 'test' saved [1445045]
```

Figura 3: Download de 2 ficheiros simultaneamente.

```
jpcorreia@jpcorreia ~$ ./teste > wget http://localhost:8080/cbow_s50.txt
--2021-05-23 14:50:42-- http://localhost:8080/cbow_s50.txt
Resolving localhost (localhost)... :1, 127.0.0.1
Connecting to localhost (localhost)::1:8080... connected.
HTTP request sent, awaiting response... 200 Ok
Length: unspecified
Saving to: 'cbow_s50.txt.5'

cbow_s50.txt.5    [ <=>          ] 428,21M 125MB/s   in 3,7s

2021-05-23 14:50:45 (117 MB/s) - 'cbow_s50.txt.5' saved [449011567]
```

Figura 4: Download de um ficheiro de 428 MB via HTTP

```
jpcorreia@jpcorreia ~$ ./teste > wget https://localhost:8081/cbow_s50.txt
--2021-05-23 14:52:07-- https://localhost:8081/cbow_s50.txt
Loaded CA certificate '/etc/ssl/certs/ca-certificates.crt'
Resolving localhost (localhost)... :1, 127.0.0.1
Connecting to localhost (localhost)::1:8081... connected.
HTTP request sent, awaiting response... 200 Ok
Length: unspecified
Saving to: 'cbow_s50.txt'

cbow_s50.txt      [ <=>          ] 428,21M 108MB/s   in 4,1s

2021-05-23 14:52:12 (104 MB/s) - 'cbow_s50.txt' saved [449011567]
```

Figura 5: Download de um ficheiro de 428 MB via HTTPS

7 Conclusões e Trabalho Futuro

Tendo em conta os testes e resultados obtidos, infere-se que se obteve sucesso na realização deste trabalho prático. Não só foram cumpridos os requisitos obrigatórios como também implementamos todos os requisitos opcionais referidos no enunciado.

A realização do trabalho foi interessante na medida em que foi possível aprofundar e consolidar conhecimentos acerca de protocolos de transporte, mais concretamente UDP e TCP, bem como o protocolo HTTP e as suas exigências e particularidades.

Com esforço e trabalho em equipa, foi possível concretizar uma aplicação fiável e robusta, sendo a mais rápida possível na sua função de transferência de ficheiros e assim concluir o projeto pedido para o TP2.

Como proposta de melhoria futura, sugere-se a implementação de um mecanismo de inserção, atualização e remoção de ficheiros nos servidores FastFileSrv, através de pedidos HTTP POST, PUT e DELETE enviados pelo cliente.