

# Gateway Aplicacional e Balanceador de Carga sofisticado para HTTP

Sofia Santos<sup>1[a89615]</sup>, Rui Armada<sup>2,3[a90468]</sup>, and Ariana Lousada<sup>3[a87998]</sup>

Universidade do Minho, Braga  
Departamento de Informática

**Abstract.** Desenvolvimento de um *gateway* de aplicação que opere com o protocolo HTTP e que tenha capacidade de responder a vários pedidos de clientes distintos em simultâneo, recorrendo a uma *pool* dinâmica de servidores.

**Keywords:** HTTP · Gateway · Server · Client · Protocol.

## 1 Introdução

Este projeto, desenvolvido na linguagem de programação Python, consistiu na criação de um *gateway* de aplicação. Através deste *gateway*, um ou vários clientes são capazes de pedir ficheiros de qualquer tipo. Estes ficheiros estão localizados em um ou mais servidores, que comunicam com o *gateway* utilizando um protocolo desenvolvido pelo grupo de trabalho que funciona sobre UDP. Para além dos módulos constituintes do Python por defeito, recorreu-se ao módulo `aiohttp` que permite receber e responder a pedidos HTTP de forma assíncrona e paralela.

## 2 Arquitetura da solução

A solução desenvolvida pela equipa de trabalho consiste maioritariamente em três componentes distintos: servidor (`FastFileSrv.py`), *packets* (`Packet.py`) e *gateway* (`HttpGw.py`).

O *gateway* comunica com os servidores por UDP utilizando os pacotes especificados no ficheiro `Packet.py`. Por sua vez, os clientes comunicam com o *gateway* por UDP, através de pedidos HTTP. Desta forma, o *gateway* possui duas vias de comunicação.

## 3 Especificação do protocolo

### 3.1 Formato das mensagens protocolares

As mensagens enviadas com o protocolo desenvolvido pelo grupo de trabalho possuem cinco campos:

- type - define o tipo da mensagem (NEW\_CONNECTION, REQUEST\_FILE, FILE\_CHUNK ou FILE\_NOT\_FOUND\_ERROR);

- `chunkN` - especifica o número do *chunk*, se a mensagem for do tipo `REQUEST_FILE` ou `FILE_CHUNK`;
- `data` - contém os dados que a mensagem pretende enviar;
- `hasNext` - informa se o *chunk* enviado é o último de uma sequência;
- `md5` - *hash* MD5 dos dados contidos no pacote, usado para fins de verificação dos dados.

Antes de serem enviadas, as mensagens devem ser serializadas, isto é, convertidas para um formato binário, visto que os *sockets* comunicam de forma binária. Para isto, foi utilizado o módulo `pickle` do Python, que possibilita este tipo de conversão. Consequentemente, o recetor da mensagem deverá deserializar os bytes recebidos de forma a obter os pacotes em formato legível.

Foi decidido pela equipa de trabalho que cada *chunk* deveria ter 4098 bytes de tamanho, o que nos possibilita a ter a nosso dispor *chunks* de uma maior dimensão, que por sua vez permitem o envio de ficheiros com uma relativa rapidez. Contudo, não é possível utilizar *chunks* de maior tamanho, uma vez que o limite para um pacote UDP é de cerca de 65000 bytes e que este possui mais informação para além do *chunk*.

De forma a evitar a leitura de *chunks* corrompidos, geramos um *hash* usando o algoritmo MD5 antes e depois do envio de um pacote. Se o *hash* gerado após a receção do pacote for igual ao *hash* que veio junto com os dados, a probabilidade de ter havido corrupção de dados é extremamente reduzida. Este *hashing* pode ser também utilizado para criar uma ligação mais segura, apesar deste algoritmo não ser o mais adequado para esse fim. Se esse fosse o nosso objetivo principal, deveríamos usar um algoritmo como o SHA-3, por exemplo.

## 4 Implementação

A implementação do *gateway* e dos servidores encontra-se nos ficheiros `HttpGw.py` e `FastFileSrv.py`, respetivamente.

### 4.1 FastFileSrv.py

Ao executar este ficheiro, devemos especificar na linha de comandos o endereço e a porta do *gateway* ao qual este se deve ligar da seguinte forma:

```
> python FastFileSrv.py 192.168.1.100 80
```

Neste caso o *gateway* encontra-se no endereço 192.168.1.100 e executa na porta 80.

Inicialmente, o servidor irá enviar um pacote ao gateway com um pedido para estabelecer uma ligação (`SYNC`). Se o gateway receber este pedido e o aceitar, irá enviar um pacote vazio ao servidor, que atua como um *acknowledgment* (`ACK`).

Depois desta confirmação, o servidor entra num ciclo infinito, onde estará à espera de pedidos de ficheiros. Sempre que receber um pacote, irá verificar

se é um pacote do tipo `REQUEST_FILE` e se não for não irá fazer nada. Caso contrário, tenta abrir o ficheiro respetivo, caso ainda não esteja aberto, e procura pelo *chunk* pedido lendo pedaços do ficheiro. Quando chega ao pedaço "certo", envia esse pedaço dentro de um pacote. Caso o servidor não consiga encontrar o ficheiro pedido, irá enviar um pacote do tipo `FILE_NOT_FOUND_ERROR`.

Para além do *chunk*, o pacote enviado contém ainda o endereço do cliente que requisitou o ficheiro. Esta informação, que veio no pacote `REQUEST_FILE`, é necessária para o gateway saber a que cliente é que deve enviar o *chunk*.

## 4.2 HttpGw.py

A parte mais difícil de implementar no *gateway* é a capacidade de tratar de vários pedidos simultaneamente. Para resolver este problema, é utilizado o módulo `aiohttp` do Python, que nos ajuda a simplificar alguns destes aspetos. Com este módulo, apenas temos de definir o comportamento do gateway para cada tipo de HTTP Request recebido (parsing dos pedidos, envio de HTTP Response, etc.), enquanto que o módulo trata da assincronicidade.

Contudo, esta é apenas uma parte do problema, uma vez que ainda é necessário tratar dos pacotes que chegam via UDP. Para esta questão, foi desenvolvido um *demultiplexer*, que executa simultaneamente com o gateway num *thread* distinto e que trata da receção e organização dos pacotes recebidos. Por exemplo, se um pacote do tipo `NEW_CONNECTION` for recebido, adiciona o servidor que o enviou à lista de servidores conhecidos pelo gateway. Se receber um pacote do tipo `FILE_CHUNK` e se o servidor que o enviou estiver na lista de servidores conhecidos pelo gateway, o *demultiplexer* coloca os dados do pacote num dicionário, organizado de acordo com os clientes que pediram o ficheiro respetivo a esse pacote.

```
# The demultiplexer receives every UDP packet sent to the gateway and performs the required actions for each
def demultiplexer():
    while True:
        try:
            p, addr = s.recvfrom(CHUNK_SIZE * 2)
        except ConnectionResetError: # If one of the servers is unreachable, continue. This will be dealt with in another part of the program.
            continue

        packet : Packet = Packet.deserialize(p)
        print(f"Received packet from server {addr}.")

        if packet.type == PacketType.NEW_CONNECTION:
            with hosts_lock:
                known_hosts[addr] = 0
            print(f"Added server {addr} to list of known hosts.")
            s.sendto(b'',addr)
        else:
            if addr not in known_hosts:
                print(f"ERROR - Received packet from unknown server {addr}.")
            elif packet.type == PacketType.FILE_CHUNK:
                with files_lock:
                    if packet.md5 == hashlib.md5(packet.data).digest():
                        file_chunks[str(packet.data[:16],"utf-8").strip()] = (packet.data[16:],addr,packet.hasNext)
                        files_cond.notify_all()
            elif packet.type == PacketType.FILE_NOT_FOUND_ERROR:
                with hosts_lock:
                    known_hosts.pop(addr)
                print(f"ERROR - Server {addr} does not contain the requested file(s) - removing it from list of known hosts.")
```

Fig. 1. Demultiplexer

Na figura 1, `file_chunks` é um dicionário que associa a cada cliente o *chunk* mais recente recebido relativo ao ficheiro requisitado pelo mesmo, `files_lock` é o lock relativo a esse dicionário, `known_hosts` é um dicionário que associa a cada servidor um valor que corresponde à quantidade de *timeouts* verificados nesse servidor e `hosts_lock` é o seu lock. Para além disso, `s` é o socket usado pelo gateway para receber dados dos servidores.

Temos depois a função `handler`, que vai tratar de todos os HTTP Requests do tipo `"/name"`, onde `"name"` é o nome do ficheiro requisitado. Se o gateway estiver a operar no endereço 192.168.1.100e na porta 80, por exemplo, o comando `wget 192.168.1.100/file.txt` irá fazer com que esta função seja chamada com um argumento `"name"` igual a `"file.txt"`.

A função começa por registar o nome do ficheiro pedido e o endereço do cliente que o pediu. Depois, irá pedir aos servidores que conhece (por outras palavras, os servidores no dicionário `known_hosts`) pelos *chunks* que constituem o ficheiro. Foi decidido realizar estes pedidos de forma alternada, ou seja, a função irá percorrer a lista de servidores num ciclo e pedir um *chunk* de cada vez até ter recebido todos os *chunks*.

Depois de pedir um *chunk*, a função irá esperar até receber um sinal por parte do *demultiplexer* ou durante um tempo predefinido. Após esse tempo, que neste caso corresponde a 0.5 segundos, um valor definido pela equipa de trabalho, se ainda não tiver chegado o *chunk* pedido, considera-se que houve um *timeout* e o contador de *timeouts* do servidor respetivo é incrementado. Quando este valor chega a 5, o servidor é removido da lista dos servidores, visto que está a dar *timeout* demasiadas vezes. Também ocorre um *timeout* se o *chunk* que for recebido estiver corrompido, ou seja, se o seu valor de hash não corresponder ao que vinha no pacote. Tecnicamente estes dois erros são diferentes, isto é, um erro de dados corrompidos não é um erro de timeout, mas para efeitos de simplificação consideraram-se como semelhantes, visto que ambos levam à eliminação do servidor em causa.

Após a receção de todos os *chunks* (ou seja, depois de receber um *chunk* cujo valor de `hasNext` é falso), a função devolve uma HTTP Response com o ficheiro recebido em formato binário no corpo da Response.

Se o *gateway* não tiver associado a ele nenhum servidor, envia uma HTTP Response com o código 404, isto é, *File not Found*.

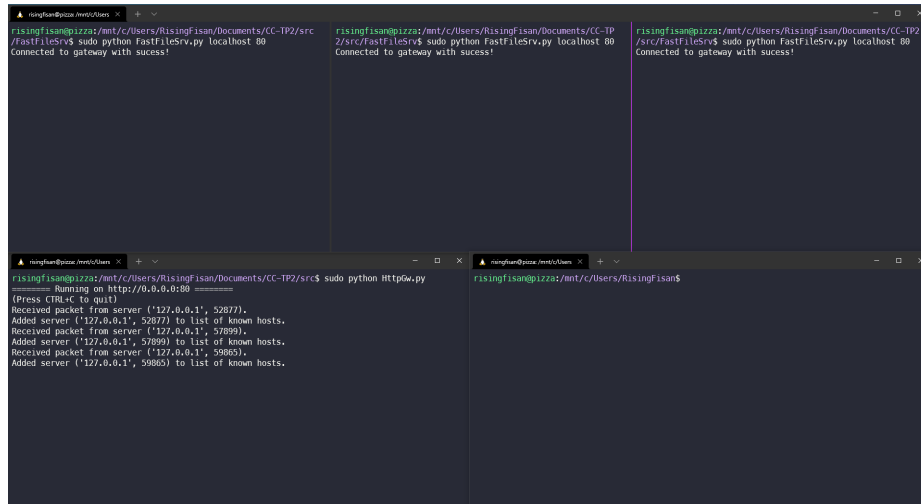
Para correr o gateway, podemos usar o comando:

```
> python HttpGw.py [endereço_IP] [porta]
```

onde os argumentos `endereço_IP` e `porta` (opcionais, com valores 0.0.0.0 e 80 por defeito) especificam o endereço e a porta pelos quais o gateway vai escutar por pedidos HTTP e pacotes UDP.

## 5 Testes e Resultados

Apesar de se terem realizado vários testes no total, irão apenas ser apresentados três de modo a manter este relatório curto e conciso. No primeiro teste, temos um *gateway* e três servidores, todos em execução num ambiente Linux.



```
risingfisan@piza:~/src$ sudo python fastfileSrv.py localhost 80
Connected to gateway with success!

risingfisan@piza:~/src$ sudo python fastfileSrv.py localhost 80
Connected to gateway with success!

risingfisan@piza:~/src$ sudo python fastfileSrv.py localhost 80
Connected to gateway with success!

risingfisan@piza:~/src$ sudo python httpgw.py
===== Running on http://0.0.0.0:80 =====
(Press CTRL+C to quit)
Received packet from server ('127.0.0.1', 52677).
Added server ('127.0.0.1', 52677) to list of known hosts.
Received packet from server ('127.0.0.1', 57899).
Added server ('127.0.0.1', 57899) to list of known hosts.
Received packet from server ('127.0.0.1', 59863).
Added server ('127.0.0.1', 59863) to list of known hosts.
```

Fig. 2. Primeiro teste - ambiente inicial.

O terminal inferior direito irá pedir o ficheiro `testfile.jpg`, uma imagem com um tamanho aproximado a 1 MB. Podemos ver na figura 3 o resultado.



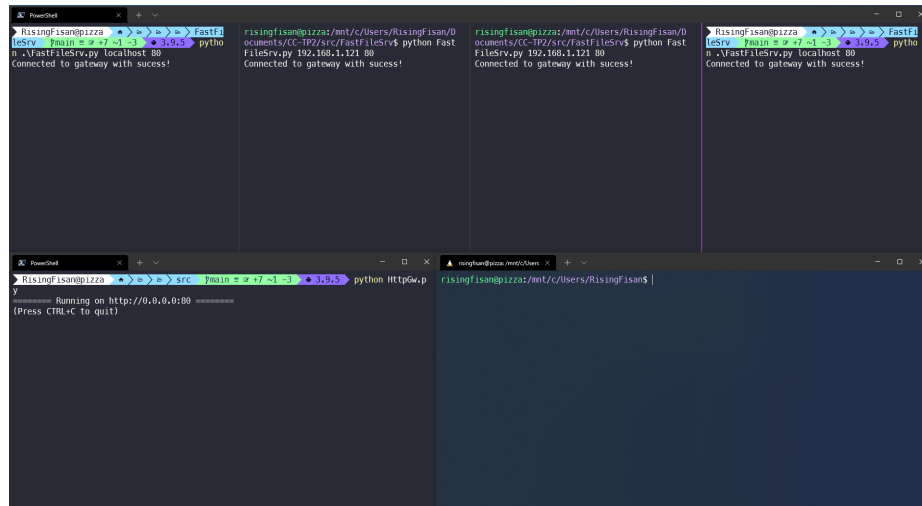


Fig. 4. Segundo teste - ambiente inicial.

Os terminais com a *shell* mais "colorida" estão a correr em Windows e os restantes em Linux.

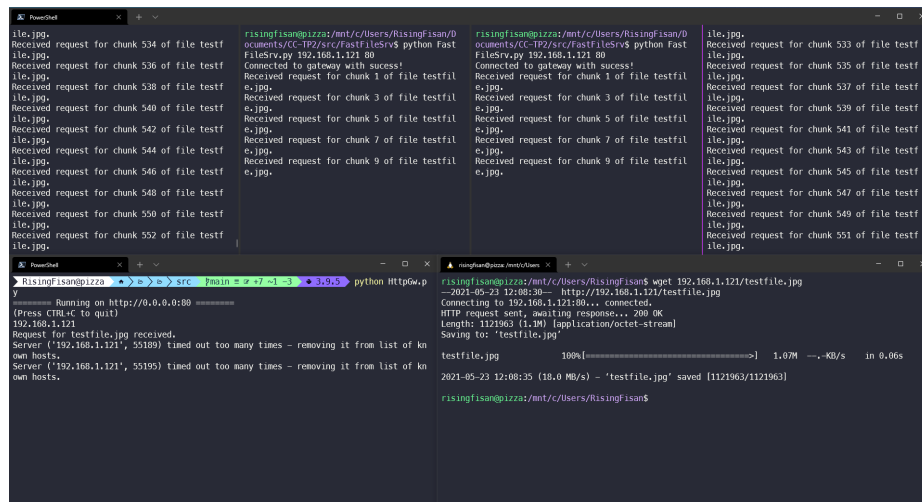


Fig. 5. Segundo teste - ambiente final.

É de fácil observação que, devido à limitação referida anteriormente, os pacotes com *chunks* enviados pelos servidores em Linux não chegam ao gateway (apesar dos pacotes de registo do servidor chegarem), ocorrendo assim *timeout*

desse dois servidores, que são removidos do sistema. Depois de serem removidos, os consequentes pedidos de *chunks* são efetuados aos dois servidores restantes. No final, conseguimos na mesma obter o ficheiro, por isso podemos ver que o *gateway* se consegue adaptar a estes problemas.

Num terceiro teste, tentámos pedir um ficheiro inexistente no sistema.

```

risingfisan@pizzaz:/mnt/c/Users/RisingFisan/Documents/CC-TP2/src
~/FastFileSrv$ sudo python FastFileSrv.py localhost 80
Connected to gateway with success!
Received request for chunk 0 of file testfile2.jpg.
ERROR - file testfile2.jpg not found.

risingfisan@pizzaz:/mnt/c/Users/RisingFisan/Documents/CC-TP2/src
~/FastFileSrv$ sudo python FastFileSrv.py localhost 80
Connected to gateway with success!
Received request for chunk 0 of file testfile2.jpg.
ERROR - file testfile2.jpg not found.

risingfisan@pizzaz:/mnt/c/Users/RisingFisan/Documents/CC-TP2/src
~/FastFileSrv$ sudo python FastFileSrv.py localhost 80
Connected to gateway with success!
Received request for chunk 0 of file testfile2.jpg.
ERROR - file testfile2.jpg not found.

risingfisan@pizzaz:/mnt/c/Users/RisingFisan/Documents/CC-TP2/src$ sudo python HttpGateway.py
Running on http://0.0.0.0:80
(Press CTRL+C to quit)
Received packet from server ('127.0.0.1', 49563).
Added server ('127.0.0.1', 49563) to list of known hosts.
Received packet from server ('127.0.0.1', 51677).
Added server ('127.0.0.1', 51677) to list of known hosts.
Received packet from server ('127.0.0.1', 33570).
Added server ('127.0.0.1', 33570) to list of known hosts.
Request for testfile2.jpg received.
Received packet from server ('127.0.0.1', 49563).
ERROR - Server ('127.0.0.1', 49563) does not contain the requested file(s) - removing it from
list of known hosts.
Received packet from server ('127.0.0.1', 51677).
ERROR - Server ('127.0.0.1', 51677) does not contain the requested file(s) - removing it from
list of known hosts.
Received packet from server ('127.0.0.1', 33570).
ERROR - Server ('127.0.0.1', 33570) does not contain the requested file(s) - removing it from
list of known hosts.
ERROR - could not get file testfile2.jpg - aborting!

risingfisan@pizzaz:/mnt/c/Users/RisingFisan$
-2021-05-23 12:41:31- http://localhost/testfile2.jpg
Resolving localhost [localhost]... 127.0.0.1
Connecting to localhost [localhost]:80... connected.
HTTP request sent, awaiting response... 404 Not Found
2021-05-23 12:41:32 ERROR 404: Not Found.

risingfisan@pizzaz:/mnt/c/Users/RisingFisan$

```

Fig. 6. Terceiro teste.

Aqui, podemos ver que os servidores comunicam ao *gateway* que não possuem o ficheiro pedido, sendo por isso removidos da lista de servidores. Depois de os remover, o *gateway* fica sem servidores disponíveis, enviando por isso uma HTTP Response com erro 404.

## 6 Conclusões e trabalho futuro

Com a realização deste trabalho fomos capazes de implementar de forma prática vários conceitos lecionados nas aulas de Comunicações por Computador, o que nos permitiu reconhecer a sua utilidade importância no mundo real. Alguns aspetos foram mais complicados e infelizmente não fomos capazes de cumprir os requisitos opcionais, como autenticação de servidores e capacidade de responder a pedidos HTTPS, mas de forma geral estamos satisfeitos com o que fomos capazes de conceber.