Trabalho Prático 2
Comunicações por Computador

José Magalhães, Manuel Carvalho e Paulo R. Pereira
a{85852,69856,86475}@alunos.uminho.pt
Grupo 1, PL6

Universidade do Minho, Braga, Portugal

Resumo Criação de um Gateway Aplicacional e Balanceador de Carga sofisticado para HTTP. Criação de um protocolo específico para este efeito que opere sobre UDP. Utilização de sockets TCP e UDP.

1 Introdução

No âmbito da unidade curricular de Comunicações por Computador foi proposto desenvolver um Gateway de aplicação, ***HttpGw***, que distríbua a carga, sofisticado para HTTP. Este opera exclusivamente com o protocolo HTTP/1.1 e é capaz de responder a múltiplos pedidos em simultâneo recorrendo, para isso, a uma *pool* dinâmica de N servidores - *FastFileSrv*.

A comunicação entre o *HttpGw* e os servidores de suporte, *FastFileSrv*, é feita de acordo com um protocolo especificamente desenhado para o efeito - *FSChunk Protocol*. Este funciona sobre UDP e não está orientado à conexão.

O objetivo principal é que a nossa solução consiga responder a vários pedidos de ficheiros, sendo estes efetuados por diversos clientes, estando a aplicação pronta para vários em simultâneo.

2 Especificação do protocolo *FSMessage*

2.1 Formato das mensagens protocolares (PDU)

De modo a garantir o bom funcionamento de todas as funcionalidades desejadas, foi necessário idealizar o protocolo *FSMessage* para funcionar sobre UDP.

Para além dos cabeçalhos habituais UDP, os pacotes utilizados na solução têm a seguinte constituição:

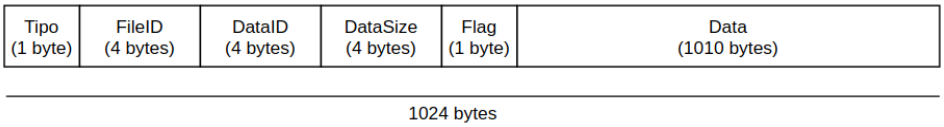

Figura 1. Pacote de Dados

De modo a tirar o máximo proveito do pacote definido, decidiu-se utilizar este para todos os tipos de comunicações, sendo que os respetivos valores dos vários campos variam consoante o tipo da mensagem em questão, conseguindo assim cobrir todas as necessidades sem nunca alterar o *overhead*.

Acrescenta-se em forma de nota que o tratamento do campo **Flag** é comum a todos os tipos de *payload*, tendo este o valor de **1** caso seja necessário ler o campo **Data**, e o valor de **0** caso contrário. Com esta verificação, conseguimos evitar operações e gasto de tempo desnecessários.

O pacote de dados pode tomar diferentes significados, consoante o seu **Tipo**:

– Se o valor de **Tipo** for **0**, então estamos perante o pedido de metadados de um ficheiro, por parte do *HttpGw*, ou a respetiva resposta, por parte do *FastFileServer*. Tanto o pedido como a resposta têm os campos do *payload* preenchidos consoante a necessidade, como vemos na imagem seguinte.

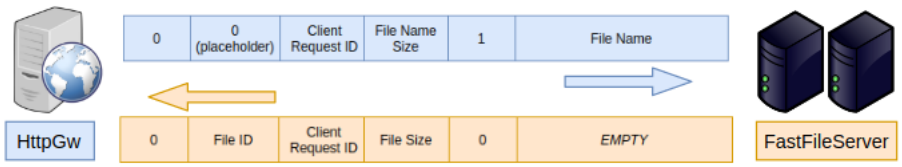

Figura 2. *Payloads* referentes ao Tipo 0

– Se estivermos perante o **Tipo 1**, então trata-se de um *FSChunk Protocol*, responsável por pedidos e respostas envolvendo os blocos de dados de um determinado ficheiro, blocos esses que têm a designação de *Chunks*. Realça-se que o *payload* com origem no *HttpGw* e destino aos *FastFileSrv* (representado de seguida, a azul) tem, no seu campo **DataSize**, um inteiro denominado por **Chunk Size**, que determina em quantos *bytes* o ficheiro vai ser dividido.

Através desta medida, garantimos total poder ao *HttpGw* em alterar, se assim pretender, em *runtime*, o número máximo de *bytes* em cada *Chunk*.

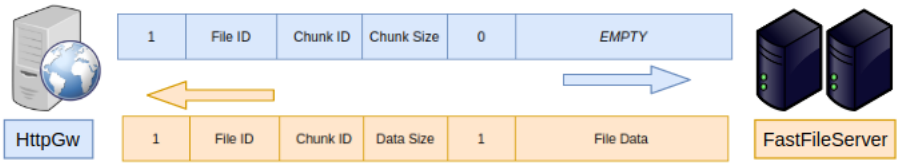

Figura 3. *Payloads* referentes ao Tipo 1

– Se **Tipo** tiver o valor **4**, então estamos perante um *Beacon*, pelo que este é enviado de forma recorrente de forma a mostrar a operacionalidade do *FastFileSrv* que o envia. Com este tipo de pacote, conseguimos garantir que um determinado *FastFileSrv* não seja considerado inativo, evitando assim a sua remoção na lista de utilizáveis, por parte do *HttpGw*.

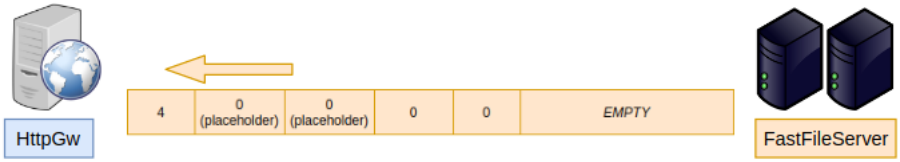

Figura 4. *Payloads* referentes ao Tipo 4

É possível verificar, então, que todos os pacotes presentes na solução estão programados de modo a terem o tamanho máximo de **1024 bytes** e que o valor de *overhead* associado aos mesmos é de **14 bytes**.

Acrescenta-se que o valor de tamanho máximo do protocolo foi cuidadosamente escolhido, sendo este inferior ao valor do MTU (*Maximum Transmission Unit*), evitando assim eventuais necessidades de fragmentação, precavendo possíveis erros durante esse mesmo processo. Após aplicada esta restrição, o valor **1024** foi o predileto pois é também múltiplo de **4** (bytes).

3 Implementação

A implementação da nossa solução foi feita utilizando a linguagem *Java*, devido à facilidade de manipulação das bibliotecas necessárias, bem como a existência

de grande variedade de métodos disponíveis, simplificando assim a execução dos vários componentes do sistema como por exemplo, toda a componente relativa aos *sockets*.

A fim de exemplificar todo este processo, de uma forma clara, foram então construídos dois diagramas de Classes - um referente ao ***HttpGw*** e outro ao ***FastFileServer*** - acompanhados de uma breve explicação de cada classe envolvente.

3.1 *HttpGateway*

- **ServerInfo:** Classe responsável por armazenar a informação relativa a um determinado *FastFileSrv*, tal como o seu *InetSocketAddress* e o seu respetivo *time_ms*, utilizado para tratar a inatividade.
- **PendingRequest:** Classe desenvolvida para representar um determinado pedido de ficheiro em espera no nosso sistema. Este, fica em *stand-by* à espera que lhe seja atribuído um ID. Uma vez definido, é transferido para a componente relativa aos ficheiros. A cada solicitação é atribuído um ID único.
- **RequestManager:** Todos os *PendingRequest* da aplicação são armazenados numa estrutura de dados, para posterior tratamento. Esta classe tem sobretudo a função de gerir todos os pedidos no sistema, tendo na sua definição todos os métodos de manipulação da referida estrutura de dados;
- **FileInfo:** Classe responsável por armazenar toda a informação relativa a um ficheiro. bem como toda a gestão dos *Chunks*, tanto no número que vamos ter, bem como efetuar *merge* de todos estes, a fim de proceder ao envio.
  Possui duas estruturas de dados: um *List*, para armazenar todos os *Chunks* relativos ao ficheiro que representa, bem como um *Map* responsável por conter, se for o caso, todos os Clientes (e respetivas informações) que pretendam o ficheiro em questão.
- **ServerManager:** Contém na sua definição uma estrutura de dados com todos os *FastFileSrv* disponíveis (e respetivos dados) para serem utilizados

pelo *HttpGw*. É a classe responsável por manipular o tempo de atividade dos servidores, para fins de controlo e, se for o caso, de proceder à remoção de um servidor da lista de disponíveis, se este passar demasiado tempo em inatividade. Por fim, é também nesta classe que é feita a escolha do *FastFileSrv* a usar.

- **FileManager:** Tal como o nome indica, é nesta classe que é feita a gestão dos ficheiros presentes no nosso sistema, através de uma estrutura de dados definida para esse efeito.
- **FSMessage:** Classe responsável por conter todos os métodos relativos à construção e leitura do protocolo *FSMessage*, já referido na secção anterior.
- **HttpGW:** Classe principal da nossa aplicação, onde é lançada uma *main Thread*, acompanhada por outras cinco *Threads*, com fins específicos:
  - *Main Thread*: Responsável por remover *FastFileSrv* inativos, reenviar pedido de metadados dos *PendingRequests* inativos e, por último, efetua o envio do conteúdo atual ao Cliente;
  - *Thread 1* - *timer*: É iniciado o contador de tempo para todos métodos que dependem deste, nomeadamente o controlo de inatividade dos servidores, ficheiros e, por fim, dos *PendingRequests*;
  - *Thread 2* - *parseUDPPayloads*: É feita a interpretação do pacote recebido, procedendo, consoante o tipo, ao tratamento devido dos dados;
  - *Thread 3* - *listenUDP*: Fica à escuta constantemente de pacotes UDP, provenientes dos *FastFileSrv*. Quando é recebido um pacote, então este é adicionado a uma FIFO (*first in, first out*) de *payloads*.
  - *Thread 4* - *listenTCP*: Fica à escuta constantemente por conexões TCP. Quando uma destas é aceite, é então criado o pacote UDP contendo o pedido de metadados do ficheiro pretendido, a fim de ser enviado a um determinado *FastFileSrv*;
  - *Thread 5* - *fetcher*: De modo a que a nossa aplicação não "fique presa"num ficheiro, definiu-se este método responsável por pedir, a cada um dos x *FastFileSrv*, x *Chunks*, podendo assim pedir blocos de dados de vários ficheiros ao mesmo tempo.

3.2 *FastFileServer*

- **FChunks:** Classe responsável por armazenar toda a informação relativamente aos demais *Chunks* de um determinado ficheiro. Possui na sua definição uma estrutura de dados usada para organizar estes por ordem;
- **FSMessage:** Já falada anteriormente;
- **FastFileServer:** Classe principal no que toca aos servidores. Aqui são definidos diversos métodos, tais como os referentes ao envio de *Chunks* e de metadados. Além disso, é lançada uma *Thread* principal, bem como outras quatro *Threads*, com fins específicos:
  - *Main Thread*: Responsável pela leitura e interpretação de pacotes UDP recebidos;
  - *Thread 1* - *listen*: Fica à escuta constantemente de pacotes UDP, provenientes dos *FastFileSrv*. Quando é recebido um pacote, então este é adicionado a uma estrutura FIFO de *payloads*;
  - *Thread 2* - *timer*: É iniciado o contador de tempo, de forma a controlar a inatividade dos ficheiros. Com isto, referimo-nos ao tempo desde que houve a receção do último pedido de *Chunk*;
  - *Thread 3* - *removeInactives*: De forma a evitar que um determinado ficheiro seja lido para memória em cada pedido, este é guardado num *buffer* temporário durante determinado tempo. A função da *thread* é de limitar o tempo que o ficheiro se encontra em memória;
  - *Thread 4* - *beaconer*: Responsável por, de x em x segundos, enviar um *Beacon* ao *HttpGw* de modo a mostrar que o servidor responsável pelo envio se encontra ativo pode ser utilizado.

Figura 5. Diagrama de Classes - *HttpGw*

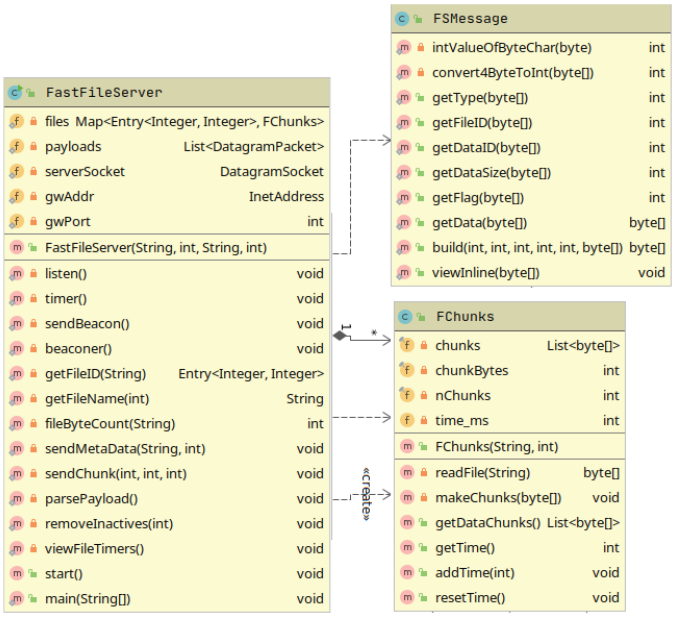

Figura 6. Diagrama de Classes - *FastFileServer*

3.3 Distribuição de carga pelos vários *FastFileServer*

Um dos objetivos do enunciado era o balanceamento da carga. Na nossa solução, supondo vários ficheiros em *queue*, são pedidos tantos *Chunks* de cada ficheiro quantos *FastFileServer* ativos no sistema. Ou seja, suponhamos que temos 5 pedidos de ficheiro e 4 servidores. Então, serão pedidos 4 *Chunks* de cada vez, de cada um dos 5 ficheiros, aos servidores. A ordem da escolha do servidor é feita através do maior tempo de inatividade, sendo escolhido, a cada momento, o que tem maior tempo.

3.4 Pedido de *FSChunks* em falta

O *HttpGw*, após pedir o último *Chunk*, relativo a um ficheiro, faz uma verificação de forma a saber se algum bloco de dados se encontra em falta. Caso o bloco não esteja completo, e visto encontrarem-se ordenados, então o *HttpGw* inicia um novo pedido de ficheiro, pedindo apenas os *Chunks* a partir do primeiro que se encontrar em falta, até o completar.

3.5 Remoção de servidores inativos

De x em x segundos, o *HttpGw* efetua uma verificação dos tempos de inatividade, calculados em tempo real, presentes em cada *FastFileServer*. Os servidores que tenham um tempo superior ao definido pelo *HttpGw* são removidos da lista dos disponíveis.

3.6 Envio parcelado ao Cliente

O ficheiro é enviado em parcelas ao Cliente, para evitar que, no caso de ficheiros pesados, passe muito tempo sem receber nenhuma resposta. O processo é controlado pelo tempo, ou seja, a cada x segundos, é verificada a integridade dos *Chunks* processados até à data. No caso de não haver falhas na ordem, então eles vão sendo enviados ao Cliente, através de uma *HTTP Reponse*.

3.7 Otimização para pedidos idênticos

Se, enquanto um ficheiro é carregado para memória, vários clientes o pretenderem, então, ao invés de carregar sucessivamente para cada pedido, é aproveitado então o que já se encontra em memória, enviando assim todos os *Chunks* do ficheiro para todos os clientes correspondentes.
É criada uma estrutura de dados, em tempo real, que armazena todos os clientes que pediram um determinado ficheiro. Quando terminar de enviar ao último cliente da estrutura de dados, então o ficheiro é eliminado da memória.

4 Testes e resultados

Para garantir o correto funcionamento da nossa solução, foram efetuados diversos testes na topologia *Core*, cobrindo grande parte das hipóteses.

Após gerar os respetivos executáveis:

```
javac -classpath <dir> <dir>/*.java
```

É possível então, estando nas respetivas diretorias, executar o *HttpGw* e N *FastFileServer* da seguinte forma:

```
java HttpGw
java FastFileServer <ipHttpGw> <portaGw> <ipServer>
```

Acrescenta-se em forma de nota que, de modo a facilitar a legibilidade das figuras, decidiu-se associar cores às entidades: ao *HttpGw* a cor verde, aos Clientes a cor azul e amarelo aos *FastFileServer*.

Inicialmente testou-se o funcionamento com **dois Clientes** no sistema, cada um a pedir um ficheiro diferente. Acrescenta-se que o momento da captura corresponde ao instante inicial de ambos os pedidos, pelo que é possível verificar o *HttpGw* a reconhecer tantos os servidores ativos, como os pedidos dos Clientes:

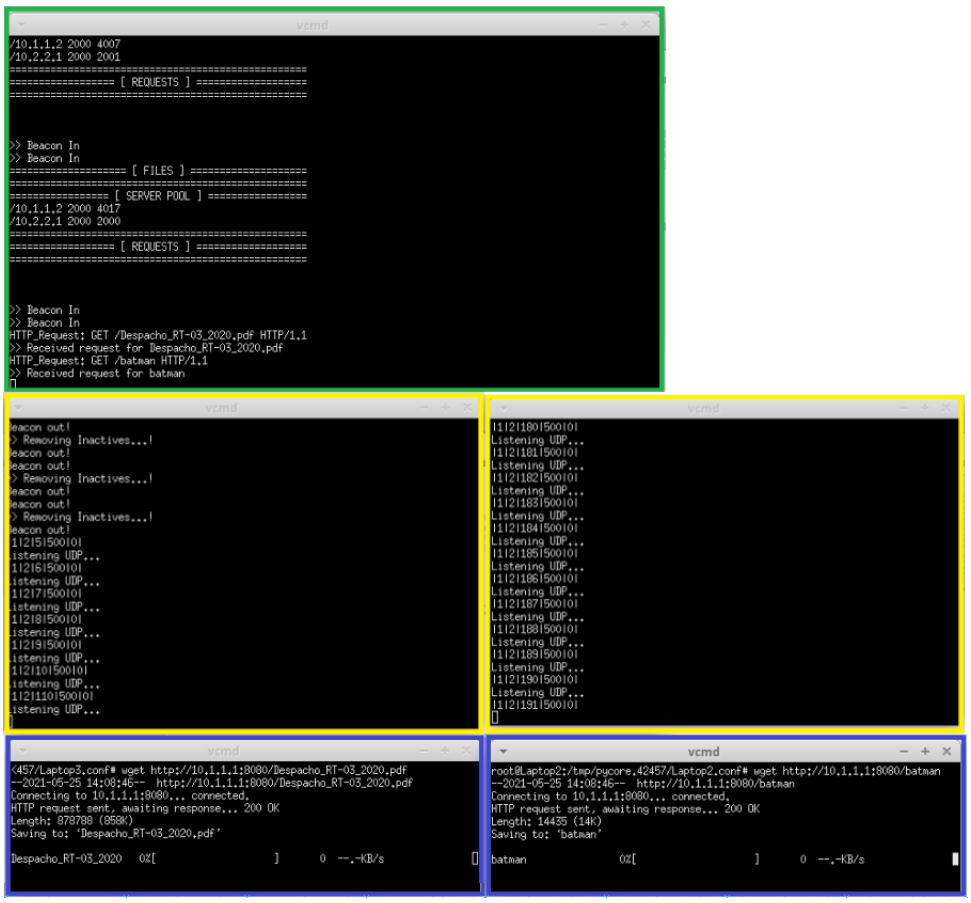

Figura 7. Teste com dois Clientes - Instante inicial

De seguida, é possível verificarmos o progresso das transferências após alguns instantes. Embora os dois pedidos tenham sido feitos ao mesmo tempo, vemos claramente a vantagem da distribuição de carga e de tratar de vários ficheiros ao mesmo tempo. Pois, com isto, o Cliente que pediu o ficheiro de tamanho substancialmente menor, vê o seu pedido respondido de forma célere.

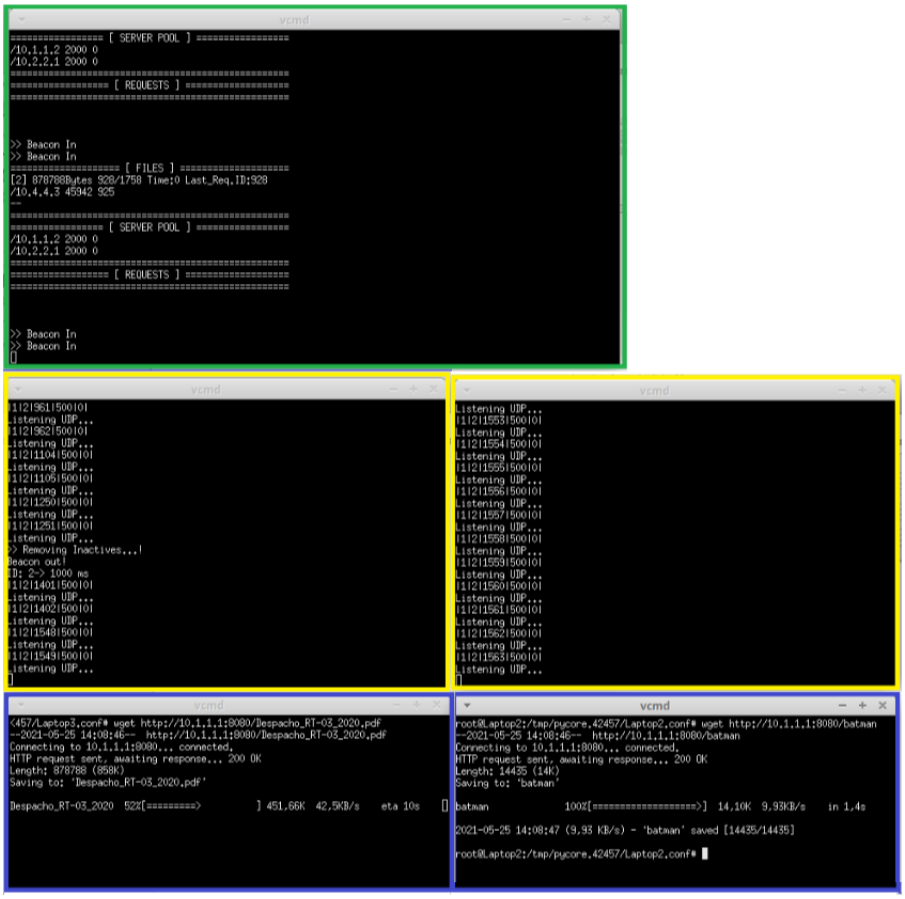

Figura 8. Teste com dois Clientes - Após algum tempo

Por fim, testou-se também, conforme pretendido pelo enunciado, efetuar, através do **mesmo Cliente**, **dois pedidos de ficheiro** distintos usando, para isso, três *FastFileServer*:

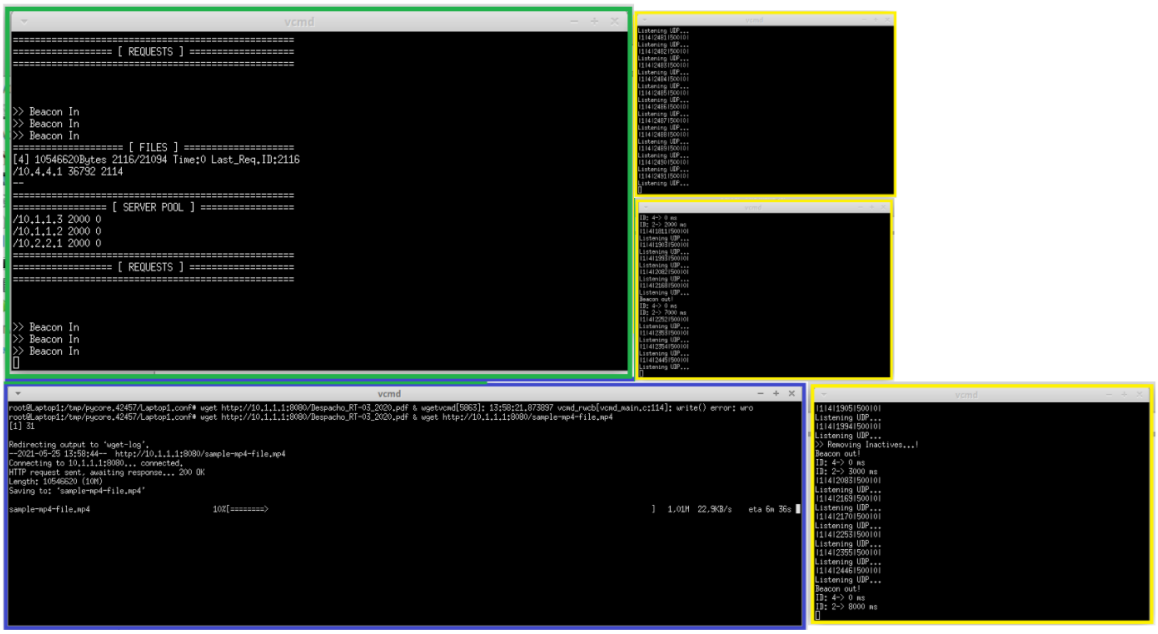

Figura 9. Teste com N pedidos no Cliente - Progresso do pedido

Posteriormente, após alguns instantes, a transferência ocorre dentro do pretendido, como se pode concluir após interpretação da *Http Response*:

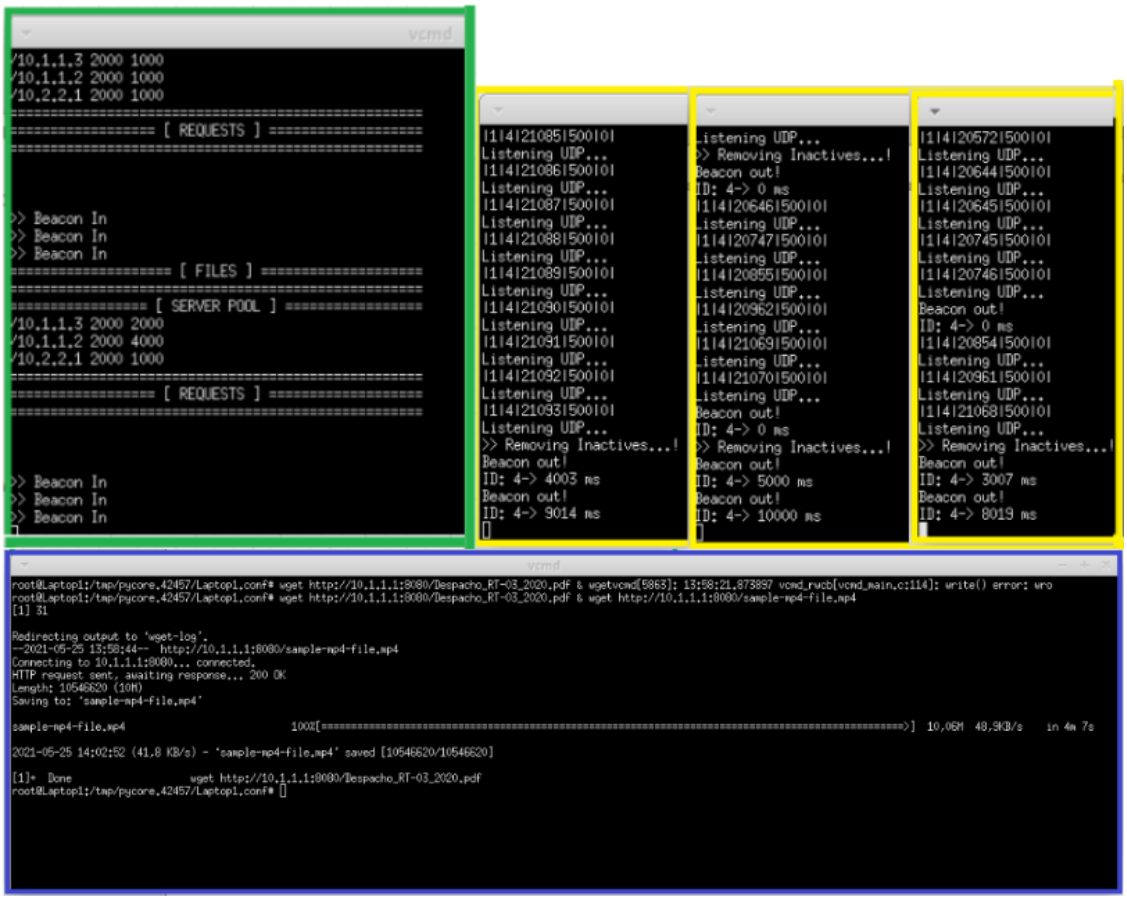

Figura 10. Teste com N pedidos no Cliente - Pedido concluído

Tendo em conta que o Cliente foi aberto, na topologia *Core*, no *Laptop1*, de forma a verificar se a transferência de ambos os ficheiros ocorreu de forma correta, basta abrir a pasta correspondente:

Figura 11. Conteúdo da Pasta *Laptop1.conf*

Por fim, e de forma a verificar a integridade dos ficheiros, abriu-se cada um deles (*Despacho_RT-03_2020.pdf* e *sample-mp4-file.mp4*) pelo que o resultado é, conforme esperado, positivo:

Figura 12. Verificação da integridade dos ficheiros transferidos

5 Conclusão

Este trabalho prático, além de extremamente desafiante, ajudou bastante a consolidar a matéria lecionada nas aulas teóricas. Reconhecemos que foi muito enriquecedor para o nosso coletivo, pois permitiu que cimentássemos todas as competências envolventes.

Assumimos que houve alguma dificuldade na ligação de todos os conceitos, pois estamos a falar de uma solução que envolve um grande número de componentes. Por um lado, foi necessária toda uma manipulação de sockets tanto TCP como UDP, cada um com as suas caraterísticas. Por outro, foi necessário transformar toda a arquitetura e requisitos da solução em código, andando sempre a par com a eficiência.

Em suma, o esforço foi grande com o intuito de garantir boas soluções para o enunciado proposto deixando, assim, um sentimento de objetivo cumprido.