ESR TP2: Serviço Over the Top para entrega de multimédia*

Gonçalo Santos, Pedro Sousa, and João Novais

Universidade do Minho, Braga, Braga, Portugal

Abstract. Neste relatório vamos abordar todas as principais decisões que tivemos que tomar, problemas que tivemos que enfrentar e conclusões a que chegamos no nosso trabalho prático 2 da Unidade Curricular de *Engenharia de Serviços em Rede*.

Keywords: TCP · UDP · *Streaming* · *Server* · RP · *Client* · *Node* · *Overlay Node* · *Underlay* · *Overlay*

1 Introdução

Para o segundo trabalho prático da UC de Engenharia de Serviços em Rede, o objetivo era implementar um serviço de streaming multimédia *Over the Top*. O trabalho baseia-se em construir uma rede Overlay (fictícia) em cima de uma rede Underlay (real) para montar uma bancada de transmissão de conteúdos multimédia que promova a eficiência e a otimização dos recursos para uma melhor experiência do utilizador. Serviços bastante conhecidos que serviram como exemplo foram a **Netflix** ou o **Hulu**.

2 Arquitetura da solução

A nossa arquitetura é constituída por vários módulos/diretorias para implementar as funcionalidades de cada um dos atores da nossa rede Overlay, da seguinte maneira:

```
esr2324
├── Client
├── Common
│   ├── Logger.java
│   └── (...)
├── Overlay_Node
├── Rendezvous_Point
├── Server
├── lib
├── topologia.imn
└── configBootstrapper.json
```

* Com o apoio da Universidade do Minho

A pasta *Common* inclui funcionalidades/propriedades partilhadas entre os vários atores da rede como por exemplo um **Logger** para poder manter um ficheiro de logs da atividade de cada ator.

Ainda existem os ficheiros *topologia.imn*, que guarda a topologia na qual o nosso serviço pode ser testado, e *configBootstrapper.json* que guarda a informação de configuração sobre a rede overlay para o **Bootstrapper**.

A pasta *lib* vai ser discutida na secção **Implementação**.

3 Especificação do(s) protocolo(s)

Para este trabalho achamos que o melhor a seguir seria utilizar dois protocolos diferentes da camada de transporte: o **TCP** e o **UDP**.

O TCP foi utilizado para assegurar o envio e receção dos pacotes de controlo e gestão da stream (tudo o que não é a transmissão do conteúdo da stream em si) e da rede *Overlay* em si (pedidos ao *Bootstrapper*, verificação se um nó/caminho ainda está "vivo", etc).

O UDP foi utilizado para transportar o conteúdo da *stream* (*frames* dos vídeos).

3.1 Formato das mensagens protocolares

TCP As mensagens protocolares mudam conforme o propósito que elas tinham que cumprir.

Existem vários tipos de mensagens para o protocolo TCP tais como:

- Pedido de stream por parte de um cliente;
- Pedido de fim de stream por parte de um cliente;
- Notifcação ao RP das streams disponíveis num servidor;
- Floods para a construção de caminhos entre clientes e RP;
- Pedidos relacionados com o *Bootstrapper* (pedido de informação ao *Bootstrapper*, envio de mudanças na informação e mensagem de remoção da rede, ambos enviados pelo *Bootstrapper*);
- Mensagens de *flood*;
- Resposta ao *flood* por parte do RP;
- *Liveness Check* por parte do Cliente;
- Notificação de final de *stream* por parte de um servidor;

UDP As mensagens protocolares têm sempre a mesma estrutura e esta é representada pela classe presente na pasta Common chamada *UDPDatagram*.

Os campos nesta classe são os seguintes:

- header_size: representa o tamanho fixo do cabeçalho da mensagem;
- payload_type: representa o tipo de conteúdo que está presente no payload (Mjpeg, MP4, etc., codificado em inteiro);
- sequence_number: representa o número da atual frame deste pacote;

- time_stamp: representa o time stamp relativo do vídeo (time_stamp = sequence_number * frame_period);
- streamName: representa o nome da stream (vídeo) que está a ser transmitida;
- payload: um array de bytes que contem o conteúdo da frame;
- payload_size: representa o tamanho do payload;

3.2 Interações

Nesta parte do relatório vamos abordar cronologicamente as interações que acontecem num cenário ideal na nossa solução.

Primeiramente é inicializado um nó da rede com a funcionalidade de *Bootstrapper*, que deve receber o seu ID e um ficheiro de configuração da rede, que ficará à escuta de mensagens dos diferentes nós da rede a pedir informação acerca do RP e dos seus próprios vizinhos, assim como verificar periodicamente por mudanças nesse mesmo ficheiro de configuração. Tirando essas funcionalidades, este nó tem um funcionamento idêntico aos restantes nós.

De seguida, é inicializado o RP e o(s) servidor(es), sendo que os servidores podem ser adicionados a qualquer altura. Os servidores, ao inicializarem, notificam o RP das streams disponíveis.

Posteriormente, o **flood** é iniciado pelo cliente para determinar os caminhos disponíveis entre este e o RP. Enquanto o cliente não receber nenhuma resposta ao flood, este espera até que receba como resposta pelo menos 1 caminho para utilizar.

Assim que o cliente tenha pelo menos 1 caminho, este inicia a segunda interação: **o cliente pergunta ao RP quais são as streams disponíveis para visualizar**. De notar que esta interação é realizada de forma direta entre o cliente e o RP.

Após todas estas interações, o cliente pode escolher uma das possíveis streams para visualizar. Ao escolher, o cliente dá inicio à próxima interação: **O cliente requisita o início de uma stream**. Esta interação é feita de nó a nó pelo atual melhor caminho do cliente para o RP (a gestão e armazenamento dos caminhos entre os clientes e o RP é feita em cada cliente, diminuindo o overhead no RP). A interação segue pelo caminho até que se chegue a um nó que já esteja a fazer a stream desejada ou até que chegue ao RP (no pior caso), com isto podemos implementar o **multicast**. Se algum nó intermédio já estiver a transmitir esta stream, este simplesmente passa também a transmitir a stream para o vizinho que lhe fez o pedido da mesma. Se o pedido chegar até ao RP, este realiza um pedido ao melhor servidor que tiver esta stream disponível para começar a realizar o streaming de uma nova stream. Com esta interação, dá-se assim o início à **transmissão do conteúdo da stream via UDP, nodo a nodo**.

Antes dessa transmissão, para que o RP possa determinar a qualquer altura qual é o Servidor que tem melhores condições para fazer a stream de um certo vídeo, **o RP estabelece comunicações periódicas com cada um dos Servidores ativos para testar as suas condições**.

A transmissão dos pacotes UDP com as frames dos vídeos é feita através da passagem do pacote desde o servidor até ao RP, depois desde o RP até ao

cliente pelo melhor caminho anteriormente definido pelo cliente (no pedido de streaming).

Enquanto todas essas interações ocorrem, cada cliente está em segundo plano a verificar se os caminhos que calculou ao fazer *flood* estão ainda ativos, fazendo também periodicamente *flood* (menos regularmente).

Com esta interação acabam assim todas as interações num cenário ideal na nossa solução.

4 Implementação

A nossa solução é constituída por 4 programas diferentes: **ONode**, **Server**, **RP** e **Client**, cada um representando um tipo de ator diferente na nossa rede.

Todos os programas foram escritos utilizando a linguagem de programação **Java**.

4.1 Detalhes, parâmetros, bibliotecas de funções, etc.

De forma a conseguir-mos fazer a análise do ficheiro de configuração do *Bootstrapper*, que está escrito em formato *json*, decidimos utilizar a biblioteca *org.json* [1].

Os programas inicializam-se da mesma forma, recebendo o endereço *ip* do *Bootstrapper*, à exceção do Server, que deve receber também a diretoria com as streams que esse servidor tem disponíveis para transmitir, e do ONode com funcionalidade de Bootstrapper, que deve receber o ip e o ficheiro de configuração como argumentos.

Todos estes programas aceitam ainda uma *flag* adicional para ativar o modo de *debug*, que faz com que todas as mensagens de *log* sejam mostradas no terminal, e não só nos ficheiros de *logging*. Essa *flag* é identificada por "-g".

4.2 *Client*

Criação e Gestão de Caminhos De forma a minimizar o fluxo na rede, ter uma entrega rápida de conteúdo e reduzir a carga de trabalho do RP, decidimos descentralizar a informação da árvore de encaminhamento de conteúdo, e decidimos que cada cliente é que seria responsável por isso. Desta forma, ao ser iniciado, um cliente faz o *flood* (após saber a informação dos seus vizinhos), de forma a que cheguem mensagens ao RP de todos os caminhos possíveis saídos do Cliente. O RP responde aos clientes com os caminhos possíveis, e estes guardam os mesmos.

Uma vez que há nós na rede que podem ser desligados e/ou adicionados, decidimos implementar um gestor de caminhos que periodicamente verifica se os caminhos atualmente usados ainda se encontram disponíveis, de 250 em 250 milissegundos, enviando pacotes de *Liveness Check* pelos mesmos. Caso algum não esteja, o servidor envia um pedido de fim de stream pelo caminho antigo (saltando os nodos que não consegues estabelecer conexão), e envia um pedido

de stream pelo novo melhor caminho. Caso não haja nenhum caminho, é feito um *flood*. Ele faz ainda, periodicamente, de 5000 em 5000 milissegundos, um *flood*, para permitir que, caso tenha recebido vizinhos novos, estes possam ser incluídos na lista de caminhos disponíveis.

GUI do Client Para tornar o nosso serviço apelativo, decidimos implementar no *Client* um *GUI* (Graphical User Interface). A GUI do cliente é simples: temos um menu principal, que é gerado ao executar o *Client* que mostra todas as possíveis streams para visualizar e um botão para poder sair da mesma.

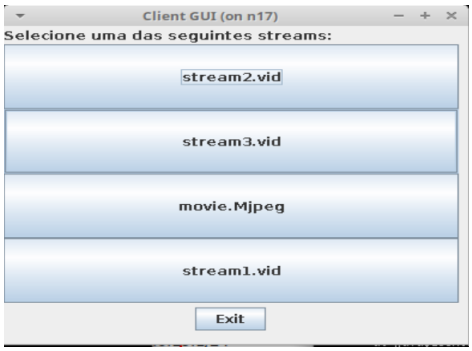

Fig. 1. Menu principal do Cliente

Este menu foi implementado na nossa solução através da classe **ClientGUI** da pasta Client.

Para além disto, a GUI do *Client* é também composta por uma outra janela onde é possível visualizar a stream em si.

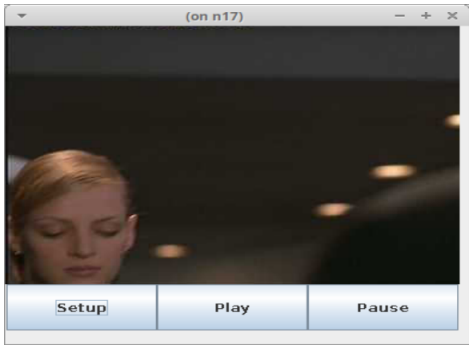

Fig. 2. Video player do cliente

Esta janela foi implementada na nossa solução através da classe **ClienteVideoPlayer** da pasta Client. Como é possível visualizar mais que uma stream num único Cliente, implementamos uma classe responsável por gerir os Video Players, sendo este o **ClientVideoManager** da pasta Client.

4.3 ONode

Gestão da Rede - modo *Bootstrapper* Tal como dito acima, ao iniciar no modo de *Bootstrapper* é necessário dar como argumento ao programa o ID do nó em questão e o ficheiro de configuração da rede, em *json*, com o seguinte formato:

```
{
    "RP": {
        "id": X,
        "ips": [
            "xxx.xxx.xxx.xxx",
            ...
        ]
    },
    "nodes": [
        {
            "id": X,
            "ips": [
                "xxx.xxx.xxx.xxx",
                ...
            ],
            "neighbours": [
                {
                    "id": X,
                    "ip": "xxx.xxx.xxx.xxx"
                },
                ...
            ]
        },
        ...
    ]
}
```

onde todos os id's são inteiros e os ip's são strings, sendo cada nó identificado pelos ip's de todas as suas interfaces, sendo essa informação utilizada para informar os nós que o contactem acerca da informação que eles precisem (id do RP, ip's do RP e os seus vizinhos).

De forma a suportar que fosse possível adicionar e remover nós da rede *overlay*, o *Bootstrapper* periodicamente vai reler o ficheiro de configuração e verificar se existem diferenças. Caso existam diferenças e os nós afetados o tenham contactado, o *Bootstrapper* avisa-os acerca dessas mudanças. Essa verificação também

acontece quando um nó com um IP não reconhecido pelo *Bootstrapper* o contacta, pelo que ele relê o ficheiro de configuração para verificar se este foi adicionado e, caso tenha sido, envia-lhe a informação, caso contrário informa-o que este não é válido e este desconecta-se. Para além de informar os nós conectados acerca das mudanças, também avisa nós existentes que tenham sido removidos da rede que isso aconteceu, e pede que eles se desconectem.

5 Limitações da Solução

5.1 Sinalização do final de uma stream

Uma das grandes dificuldades que enfrentamos foi encontrar uma maneira confiável e eficiente de sinalizar para todos os nós de um caminho entre o cliente e o RP o final de uma stream.

A primeira solução na qual pensamos foi adicionar uma flag ao último pacote UDP da stream mas isto seria pouco eficaz visto que o protocolo UDP não é confiável e este pacote poderia ficar perdido.

Outra solução na qual pensamos foi enviar um pacote TCP para sinalizar o final de uma stream. O problema desta abordagem é que ao enviar um pacote TCP não é garantido que este chega depois de todos os pacotes UDP. Assim, se o cliente terminasse a stream quando este pacote chegasse, poderia perder uma grande parte da stream.

A maneira como contornamos este problema foi assegurando que o pacote TCP de sinalização de fim de stream chegasse quase no fim da stream, sendo que se alguns frames fossem perdidos, seriam poucos. Assim, alteramos o comportamento dos servidores de maneira que estes mandassem os pacotes de frame ao ritmo do *frame_period* do vídeo, sendo este valor o tempo entre os frames (inverso dos frames por segundo).

Assim, não só resolvemos este problema, como também diminuímos o "trânsito" na rede (dado que os pacotes eram mandados com um throughput mais baixo), fazendo com que menos pacotes fossem perdidos.

5.2 Ordem de chegada dos pacotes UDP

Dado que o protocolo UDP não garante a ordem de chegada dos pacotes, poderia acontecer que o cliente reproduzisse o vídeo com a ordem dos frames errada, fornecendo uma má experiência ao cliente. Assim, criamos a classe *FrameQueue*, sendo esta responsável manter a ordem dos pacotes recebidos (interiormente usa uma Priority Queue para atingir este fim de maneira eficiente).

5.3 Tipos de vídeos que podem ser transmitidos

Outra limitação do nosso trabalho é o facto de a nossa solução apenas suportar vídeos do formato **Mjpeg** para serem transmitidos. Ainda tentamos implementar a transmissão de vídeos **MP4** mas tivemos problemas com librarias extra que era preciso utilizar para trabalhar com vídeos deste formato.

6 Testes e resultados

Os testes realizados e dispostos nesta secção seguem a topologia apresentada no enunciado, à exceção das figuras 4, 5 e 6, que são realizados numa topologia mais simples.

```
[2023-12-06 14:52:18] : Log File created
[2023-12-06 14:52:18] : Now Listening to TCP requests
[2023-12-06 14:52:18] : Listening on UDP:333
[2023-12-06 14:52:19] : Bootstrapper : Received neighbours request from 10.0.15.2
[2023-12-06 14:52:20] : Bootstrapper : Successfully sent neighbour information to 10.0.15.2
[2023-12-06 14:52:25] : Bootstrapper : Received neighbours request from 10.0.4.2
[2023-12-06 14:52:25] : Bootstrapper : Successfully sent neighbour information to 10.0.4.2
[2023-12-06 14:52:27] : Bootstrapper : Received neighbours request from 10.0.1.2
[2023-12-06 14:52:28] : Bootstrapper : Successfully sent neighbour information to 10.0.1.2
[2023-12-06 14:52:29] : Bootstrapper : Received neighbours request from 10.0.16.1
[2023-12-06 14:52:29] : Bootstrapper : Successfully sent neighbour information to 10.0.16.1
[2023-12-06 14:52:32] : Bootstrapper : Received neighbours request from 10.0.5.20
[2023-12-06 14:52:32] : Bootstrapper : Successfully sent neighbour information to 10.0.5.20
[2023-12-06 14:52:34] : Bootstrapper : Received neighbours request from 10.0.18.10
[2023-12-06 14:52:34] : Bootstrapper : Successfully sent neighbour information to 10.0.18.10
[2023-12-06 14:52:37] : Bootstrapper : Received neighbours request from 10.0.19.21
[2023-12-06 14:52:37] : Bootstrapper : Successfully sent neighbour information to 10.0.19.21
```

Fig. 3. Bootstrapper Logs

```
[2023-12-06 19:13:47] : Now Listening to TCP requests
[2023-12-06 19:13:47] : Listening on UDP:333
[2023-12-06 19:13:59] : Bootstrapper : Received neighbours request from 10.0.3.1
[2023-12-06 19:13:59] : Bootstrapper : Successfully sent neighbour information to 10.0.3.1
[2023-12-06 19:14:00] : Bootstrapper : Received neighbours request from 10.0.1.10
[2023-12-06 19:14:01] : Bootstrapper : Successfully sent neighbour information to 10.0.1.10
[2023-12-06 19:14:21] : Bootstrapper : Received neighbours request from 10.0.5.1
[2023-12-06 19:14:21] : Bootstrapper : Successfully sent neighbour information to 10.0.5.1
[2023-12-06 19:14:24] : Bootstrapper : Received neighbours request from 10.0.2.1
[2023-12-06 19:14:24] : Bootstrapper : Couldn't find Node with ip 10.0.2.1 on config file.. Checking changes.
[2023-12-06 19:14:24] : Bootstrapper : Node with ip 10.0.2.1 is not on the overlay network...
[2023-12-06 19:14:44] : Bootstrapper : Successfully sent new neighbour information to 10.0.4.2
[2023-12-06 19:14:55] : Bootstrapper : Received neighbours request from 10.0.2.1
[2023-12-06 19:14:55] : Bootstrapper : Successfully sent neighbour information to 10.0.2.1
[2023-12-06 19:15:14] : Bootstrapper : Successfully sent disconnect message to 10.0.2.1
[2023-12-06 19:15:14] : Bootstrapper : Successfully sent new neighbour information to 10.0.4.2
```

Fig. 4. Bootstrapper Logs quando há alterações na rede *overlay*

```
[2023-12-06 19:21:04] : Sending neighbour request to Bootstrapper
This node is not on the overlay network.. Shutting down
<ib/json-20231013.jar Overlay_Node/ONode 10.0.2.2 -g
[2023-12-06 19:21:37] : Sending neighbour request to Bootstrapper
[2023-12-06 19:21:37] : Log File created
[2023-12-06 19:21:37] : Received id, RP information and neighbours from Bootstra
pper
[2023-12-06 19:21:37] : Now Listening to TCP requests
[2023-12-06 19:21:37] : Listening on UDP:333
[2023-12-06 19:21:54] : Received REMOVED message from bootstrapper. Turning off.
```

Fig. 5. Logs de um nó quando não está na rede *overlay* e quando é removido da rede *overlay*

```
[2023-12-06 19:20:54] : Received id, RP information and neighbours from Bootstrapper
[2023-12-06 19:20:54] : Now Listening to TCP requests
[2023-12-06 19:20:54] : Listening on UDP:333
[2023-12-06 19:21:34] : Received topology changes message from Bootstrapper
[2023-12-06 19:21:34] : Received new id, RP information and neighbours from Bootstrapper
[2023-12-06 19:21:54] : Received topology changes message from Bootstrapper
[2023-12-06 19:21:55] : Received new id, RP information and neighbours from Bootstrapper
```

Fig. 6. Logs de quando um nó recebe informações novas quando há mudanças na rede *overlay*

```
[2023-12-06 14:52:19] : Sending neighbour request to Bootstrapper
[2023-12-06 14:52:19] : Log File created
[2023-12-06 14:52:19] : Received id, RP information and neighbours from Bootstrapper
[2023-12-06 14:52:20] : Now Listening to TCP requests
[2023-12-06 14:52:20] : Listening on UDP in Port 333
[2023-12-06 14:52:32] : Received available streams warning from server 10.0.5.20
[2023-12-06 14:52:32] : Adding available streams from server 10.0.5.20
[2023-12-06 14:52:34] : RP requesting ServerStreams from the 9 server!
[2023-12-06 14:52:34] : RP received the ServerStreams response and is going to update the server ranking!
[2023-12-06 14:52:34] : Received available streams warning from server 10.0.18.10
[2023-12-06 14:52:34] : Adding available streams from server 10.0.18.10
[2023-12-06 14:52:36] : RP requesting ServerStreams from the 1 server!
[2023-12-06 14:52:36] : RP received the ServerStreams response and is going to update the server ranking!
[2023-12-06 14:52:36] : RP requesting ServerStreams from the 9 server!
[2023-12-06 14:52:36] : RP received the ServerStreams response and is going to update the server ranking!
[2023-12-06 14:52:37] : Received flood message from 10.0.19.21
[2023-12-06 14:52:37] : Received available stream request from 10.0.19.21
[2023-12-06 14:52:37] : Sent flood response to client: 10.0.19.21
[2023-12-06 14:52:37] : Sent available streams to client 10.0.19.21
[2023-12-06 14:52:38] : RP requesting ServerStreams from the 1 server!
[2023-12-06 14:52:38] : RP received the ServerStreams response and is going to update the server ranking!
```

Fig. 7. Inicialização do RP

```
[2023-12-06 14:52:34] : Getting availabe Streams
[2023-12-06 14:52:34] : Sending neighbour request to Bootstrapper
[2023-12-06 14:52:34] : Log File created
[2023-12-06 14:52:34] : Received id, RP information and neighbours from Bootstrapper
[2023-12-06 14:52:34] : Notifying Rendezvous Point about the available streams
[2023-12-06 14:52:34] : Now Listening to TCP requests
[2023-12-06 14:52:36] : Received ServerStream request from the RP!
[2023-12-06 14:52:36] : Got a new ServerStream request from the RP!
[2023-12-06 14:52:36] : Notifying Rendezvous Point about the available streams
[2023-12-06 14:52:38] : Got a new ServerStream request from the RP!
[2023-12-06 14:52:38] : Notifying Rendezvous Point about the available streams
[2023-12-06 14:52:40] : Got a new ServerStream request from the RP!
[2023-12-06 14:52:40] : Notifying Rendezvous Point about the available streams
```

Fig. 8. Inicialização do Server

```
[2023-12-06 14:52:37] : Sending neighbour request to Bootstrapper
[2023-12-06 14:52:37] : Log File created
[2023-12-06 14:52:37] : Received id, RP information and neighbours from Bootstrapper
[2023-12-06 14:52:37] : Now Listening to TCP messages
[2023-12-06 14:52:37] : Listening on UDP:10.0.19.21:333
[2023-12-06 14:52:37] : Sent flood message to 10.0.19.1:333
[2023-12-06 14:52:37] : Path Manager started..
[2023-12-06 14:52:37] : Received flood response from RP: 10.0.19.1
[2023-12-06 14:52:42] : 5000 milliseconds have passed, a flood will occur
[2023-12-06 14:52:42] : Sent flood message to 10.0.19.1:333
[2023-12-06 14:52:42] : Received flood response from RP: 10.0.19.1
[2023-12-06 14:52:47] : 5000 milliseconds have passed, a flood will occur
[2023-12-06 14:52:47] : Sent flood message to 10.0.19.1:333
[2023-12-06 14:52:47] : Received flood response from RP: 10.0.19.1
[2023-12-06 14:52:52] : 5000 milliseconds have passed, a flood will occur
[2023-12-06 14:52:52] : Received flood response from RP: 10.0.19.1
```

Fig. 9. Inicialização do Client

```
[2023-12-06 14:57:27] : Client requesting stream: movie.Mjpeg
[2023-12-06 14:57:27] : Got an UDP packet!
[2023-12-06 14:57:27] : Set video period and started the timer
[2023-12-06 14:57:27] : Got an UDP packet!
[2023-12-06 14:57:27] : Got an UDP packet!
[2023-12-06 14:57:27] : Got an UDP packet!
[2023-12-06 14:57:27] : Got an UDP packet!
[2023-12-06 14:57:27] : Got an UDP packet!
[2023-12-06 14:57:27] : Got an UDP packet!
[2023-12-06 14:57:27] : Got an UDP packet!
[2023-12-06 14:57:27] : Got an UDP packet!
```

Fig. 10. Pedido de stream pelo Client

```
[2023-12-06 14:57:27] : Received video stream request from 10.0.19.21
[2023-12-06 14:57:27] : A client wants the stream: movie.Mjpeg!
[2023-12-06 14:57:27] : Received available stream request from 10.0.19.21
[2023-12-06 14:57:27] : Sent available streams to client 10.0.19.21
[2023-12-06 14:57:27] : Sent 5 UDP packets to Thread pool queue
[2023-12-06 14:57:27] : Thread pool worker 1 sent UDP packet to 10.0.19.21
[2023-12-06 14:57:27] : Thread pool worker 1 sent UDP packet to 10.0.19.21
[2023-12-06 14:57:27] : Thread pool worker 1 sent UDP packet to 10.0.19.21
[2023-12-06 14:57:27] : Thread pool worker 1 sent UDP packet to 10.0.19.21
[2023-12-06 14:57:27] : Thread pool worker 1 sent UDP packet to 10.0.19.21
[2023-12-06 14:57:27] : Thread pool worker 2 sent UDP packet to 10.0.19.21
```

Fig. 11. Pedido de stream chega ao RP

```
[2023-12-06 14:57:27] : Received Video Stream Request from 10.0.8.2
[2023-12-06 14:57:27] : Streaming 'movie.Mjpeg' through UDP!
[2023-12-06 14:57:29] : Got a new ServerStream request from the RP!
[2023-12-06 14:57:29] : Notifying Rendezvous Point about the available streams
[2023-12-06 14:57:31] : Got a new ServerStream request from the RP!
[2023-12-06 14:57:31] : Notifying Rendezvous Point about the available streams
[2023-12-06 14:57:33] : Got a new ServerStream request from the RP!
[2023-12-06 14:57:33] : Notifying Rendezvous Point about the available streams
[2023-12-06 14:57:35] : Got a new ServerStream request from the RP!
[2023-12-06 14:57:35] : Notifying Rendezvous Point about the available streams
[2023-12-06 14:57:37] : Got a new ServerStream request from the RP!
[2023-12-06 14:57:37] : Notifying Rendezvous Point about the available streams
[2023-12-06 14:57:39] : Got a new ServerStream request from the RP!
[2023-12-06 14:57:39] : Notifying Rendezvous Point about the available streams
[2023-12-06 14:57:40] : Sent 501 frames!
```

Fig. 12. Pedido de stream chega ao Server

7 Conclusões e trabalho futuro

Acabado o segundo trabalho prático da UC de Engenharia de Serviços em Rede, o nosso grupo conclui que este foi bastante desafiante mas que no final desenvolvemos um serviço do qual podemos dizer que estamos orgulhosos. O nosso serviço foi desde o início idealizado para uma possível inserção num contexto real, como é o caso de outras plataformas de streaming como a **Netflix** ou o **Hulu**, o que acabou por ter impacto em várias das decisões mais importantes que tivemos que tomar.

Este trabalho foi-nos extremamente valioso como método de aprendizagem na prática dos conceitos lecionados nas aulas da Unidade Curricular. Achamos que este tipo de trabalho tem um impacto positivo na aprendizagem do aluno.

Quanto ao trabalho futuro, existem várias funcionalidades que acabamos por não implementar, seja por falta de tempo ou por nível de complexidade, e outras que não ficaram implementadas da melhor maneira possível. Uma destas funcionalidades foi sem dúvida a *GUI* do cliente que, apesar de não ser o foco do trabalho, tem um importante papel numa plataforma de streaming no contexto da vida real. Outra funcionalidades seriam o que foi discutido na secção de **Limitações da Solução** entre outras.

References

1. Página github do módulo *org.json*, bluehttps://github.com/stleary/JSON-java.