Universidade do Minho
Escola de Engenharia

Trabalho Prático 2

Serviço Over the Top para Entrega de Multimédia

Engenharia de Serviços em Rede
Mestrado em Engenharia Informática

PL7 - Grupo 3

Gabriela Santos Ferreira da Cunha - pg53829

Millena de Freitas Santos - pg54107

Nuno Guilherme Cruz Varela - pg54117



dezembro, 2023

Conteúdo

1 Introdução

No âmbito do segundo trabalho prático da UC de Engenharia de Serviços em Rede foi-nos proposta a implementação de um serviço *over-the-top* para entrega de multimédia.

Os serviços *over-the-top* (OTT) são serviços implementados sobre a camada aplicacional, daí a denominação de *over-the-top*. Estes surgiram com o novo padrão de uso da Internet por parte do cliente que consiste no consumo de conteúdos de qualquer tipo, a todo o instante, continuamente e, muitas vezes, em tempo real. Este padrão colocou grandes desafios à infraestrutura IP de base que, apesar de não ter sido originalmente desenhada com esse requisito, tem sido adaptada com redes sofisticadas de entrega de conteúdos (CDNs) e serviços específicos, de modo a resolver a entrega massiva de conteúdos. Alguns exemplos comuns de serviços OTT são as plataformas de *streaming* de vídeo como a Netflix e a Amazon Prime Video.

2 Arquitetura da Solução

A nossa arquitetura consiste em 4 aplicações: o `Client`, o `Node`, o `RP` e o `Server`. Estas aplicações fazem uso de ambos os tipos de pacote para comunicarem entre si: `ControlPacket`, para controlo, e `RtpPacket`, para *streaming*. O servidor apenas estabelece contacto com o RP, uma vez que o *streaming* vai chegar aos nodos e clientes por meio deste último. Para além de receber os pacotes com o conteúdo a transmitir, o RP necessita também da classe `ServerInfo`, que regista as condições de entrega de um dado servidor, axuiliando na monitorização dos servidores. Por último, o servidor faz uso da classe `VideoStream`, exclusiva a assuntos relacionados com o *streaming*.

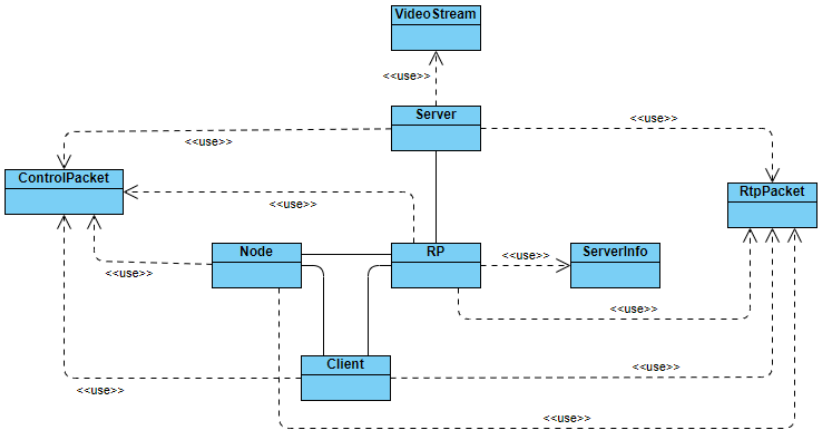

Figura 1: Arquitetura do sistema.

3 Especificação dos Protocolos

Perante as múltiplas interações que ocorrem na rede, necessitamos de um protocolo que uniformize a comunicação entre os componentes.

Em relação ao protocolo da camada de transporte, para *streaming*, escolhemos o protocolo UDP em detrimento do TCP. O protocolo TCP possui mecanismos de controlo de congestionamento e garantia de entrega integrados que se traduzem num maior *overhead* em cada pacote. O estabelecimento de conexão, envio de confirmações e retransmissões em caso de perda de pacotes introduzem uma maior latência na comunicação, algo que não é desejado em *streaming* de conteúdo, uma vez que a entrega contínua de dados é mais crítica do que garantir que todos os pacotes cheguem. A transmissão requer o mínimo de *delay* entre o envio da informação do servidor e a receção da mesma informação no cliente.

Para as mensagens de controlo, a escolha recaiu novamente sobre o UDP por uma questão de simplificação do processo. Nesta troca de informações, o *delay* não tem tanta importância mas é igualmente necessário minimizar a quantidade de *overhead* na rede, pelo que preferimos manter a utilização de uma comunicação orientada ao datagrama. Posteriormente, iremos verificar que grande parte das mensagens de controlo trocadas na rede não necessitam prontamente de uma resposta do destinatário, sendo que esta mensagem poderá ter de ser reencaminhada para que uma resposta seja formulada e enviada. No entanto, existem alguns casos particulares em que é essencial o envio de mensagens de confirmação para que a receção de um pacote seja confirmada. Coube ao grupo tratar da implementação deste procedimento desejado, uma vez que o UDP não fornece esse controlo incluído no TCP, como havíamos referido anteriormente.

3.1 Formato das Mensagens Protocolares

3.1.1 Pacote de Controlo

Relativamente à estrutura do pacote de controlo, utilizamos o seguinte cabeçalho com um comprimento fixo de 10 *bytes*.

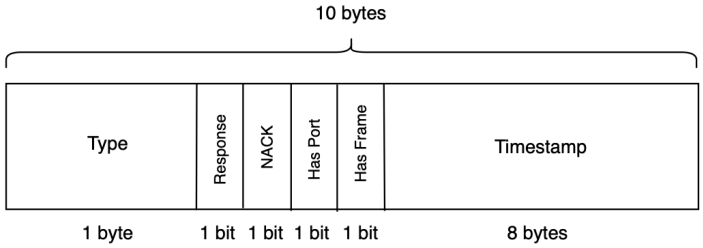

Figura 2: Estrutura do cabeçalho do pacote de controlo.

- ***Type*:** Inteiro que representa o tipo da mensagem enviada;

  Na seguinte tabela, indicamos os tipos de mensagens existentes no nosso projeto.

Tipo		Descrição
0	`NEIGHBOURS`	Pedido de vizinhos
1	`STATUS`	Obtenção das condições de entrega de um servidor
2	`PLAY`	Pedido de entrada na árvore
3	`LEAVE`	Pedido de saída da árvore
4	`POLLING`	Pedido de permanência na árvore

Tabela 1: Tipos de mensagem.

- ***Response*:** *Bit* que representa se a mensagem é um pedido (0) ou a resposta a esse pedido (1);
- ***NACK*:** *Bit* que representa se a mensagem se trata de um *negative ack*;
- ***Has Port*:** *Bit* que indica se a mensagem contém no *payload* a porta a ser utilizada para o envio do conteúdo;
- ***Has Frame*:** *Bit* que indica se a mensagem contém o *frame* a partir do qual o remetente quer receber o conteúdo no *payload*;
- ***Timestamp* :** *Double* que regista o *timestamp* de envio da mensagem;

  Este *timestamp* é utilizado na administração da árvore dinâmica para distinguir cronologicamente as confirmações dos possíveis caminhos provenientes dos pedidos de subscrição ou renovação na árvore. O cliente ao enviar um pedido de `PLAY` ou `POLLING`, regista o seu *timestamp* na mensagem. Este tempo é utilizado no caminho de volta do RP ao cliente, no qual cada nodo intermediário compara o *timestamp* recebido na mensagem de confirmação com o registado para o seu *parent* atual. Caso seja superior, substitui este *parent* pelo endereço IP do nodo que lhe reencaminhou esta mensagem. Caso contrário, envia-lhe um ACK negativo, que é propagado até ao RP, e todos os nodos que a recebem atualizam as suas árvores e os seus *parents*.

Em seguida, apresentamos a estrutura do *payload* da mensagem.

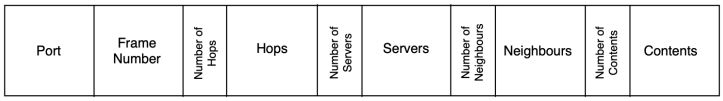

Figura 3: Estrutura do *payload* do pacote de controlo.

- ***Port*:** indica a porta pela qual o conteúdo será transmitido;
- ***Frame Number*:** indica a *frame* do conteúdo a ser transmitido;
- ***Number of Hops*:** indica a quantidade de *hops* a serem enviados no *payload*;
- ***Hops*:** contém todos os nodos do caminho entre o cliente e o RP. Este campo auxilia a evitar ciclos entre a propagação da mensagem;
- ***Number of Servers*:** indica a quantidade de servidores a serem enviados no *payload*;
- ***Servers*:** contém os IPs dos servidores;
- ***Number of Neighbours*:** indica a quantidade de *neighbours* a serem enviados no *payload*;
- **Neighbours:** contém todos os *neighbours* do nodo a receber a mensagem;
- ***Number of Contents*:** indica a quantidade de conteúdos a serem enviados no *payload*;
- ***Contents*:** contém o conteúdo a ser requisitado pelo cliente ou a lista de conteúdos que os servidores possuem para transmitir.

3.1.2 Pacote de *Streaming*

Relativamente ao pacote de *streaming*, este não sofreu alterações e mantém-se igual ao do código fornecido pela equipa docente.

O cabeçalho do pacote RTP tem um tamanho de 12 bytes e possui a estrutura representada na figura 4.

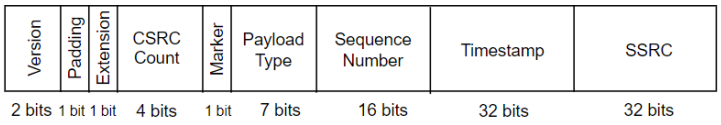

Figura 4: Estrutura do *cabeçalho* do pacote RTP.

- ***Version*:** indica a versão do protocolo RTP utilizada;
- ***Padding*:** indica se existe algum preenchimento no fim do pacote, para que este satisfaça um comprimento total múltiplo de 32 bits;
- ***Extension*:** indica se existe um *extension header* a seguir ao cabeçalho RTP;
- ***CSRC Count*:** indica o número de *contributing sources* presentes no pacote;

- ***Marker*:** *bit* utilizado para propósitos específicos;
- ***Payload Type*:** especifica o formato do *payload* para que este seja interpretado;
- ***Sequence Number*:** usado para detetar a perda de pacotes e restaurar a sequência de pacotes;
- ***SSRC*:** identificador que é atribuído a cada fonte de dados numa sessão RTP para distinguir uma fonte de outra. Cada participante deve ter um SSRC único.

3.2 Interações

3.2.1 Receção dos Vizinhos

Quando o RP arranca, este deverá contactar o *bootstrapper* a identificar-se, recebendo como resposta a lista dos vizinhos e servidores com os quais deverá comunicar e estabelecer conexões. Os restantes nodos deverão também identificar-se perante o *bootstrapper*, no entanto receberão apenas a lista de vizinhos na resposta, uma vez que estes nodos não vão precisar de comunicar diretamente com os servidores para receber conteúdo.

Como foi abordado anteriormente, existiriam pedidos que necessitariam de uma confirmação proveniente do destinatário. Como o pedido de vizinhos não se propaga e é uma mensagem de controlo, trata-se de uma situação em que a implementação do controlo de erros é desejável. Neste sentido, o nodo que faz o pedido espera sempre 2 segundos pela resposta. Caso o pedido seja feito e o *bootstrapper* ainda não tenha sido arrancado ou, por algum outro motivo, não seja possível estabelecer conexão com o *bootstrapper*, será enviada uma nova tentativa a cada 2 segundos, até que a resposta seja recebida.

3.2.2 Escolha do Servidor

Esta interação é feita diretamente entre o RP e os servidores. Quando o RP arranca, pretende-se que haja uma troca de mensagens de prova, isto é, uma mensagem do tipo `STATUS` entre o RP e os servidores, de modo a obter as condições de entrega de cada um. Assim, quando o RP recebe um pedido de `PLAY`, este poderá selecionar o melhor servidor naquele instante, pelo qual irá receber o conteúdo, de acordo com as métricas guardadas.

Para além disso, ao longo do *streaming* para os clientes pretende-se que a transmissão seja sempre rápida e sem falhas, mesmo que não haja nenhuma nova entrada na rede, ou seja, um novo pedido de `PLAY`. Como sabemos, a rede não é sempre estável e podem surgir *delays* ou *losses* nas ligações. Para isso, o RP, quando é iniciado, cria uma *thread* para executar o serviço de *tracking*. Este é responsável por, a cada segundo, enviar pedidos `STATUS` aos servidores, calcular as métricas de latência e *loss* e atualizar estes valores consoante cada

servidor, podendo vir a substituir o servidor corrente, ou seja, o melhor até ao momento.

3.2.3 Subscrição na Árvore

Assim que o cliente arranca, é enviada uma mensagem do tipo `PLAY`, assumindo-se que este quer, desde já, receber o conteúdo de *streaming*. Esta mensagem é propagada progressivamente por todos os vizinhos até que chegue a algum nodo que já esteja a transmitir o conteúdo pedido ou, no limite, até ao RP.

Para explicar mais detalhadamente a lógica implementada para a subscrição na árvore, selecionamos dois cenários.

Cenário 1

Neste cenário, ainda não há conteúdo a ser transmitido, portanto todos os pedidos chegam ao RP, conforme ilustrado na figura 5.

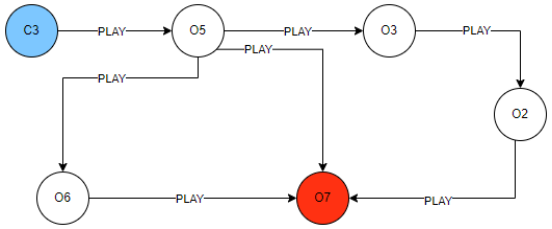

Figura 5: Primeiro cenário de subscrição na árvore.

Suponhamos que o primeiro pedido que chegou ao RP foi o pedido cujo remetente direto é o `O5` e, portanto, vamos analisar primeiramente este fluxo.

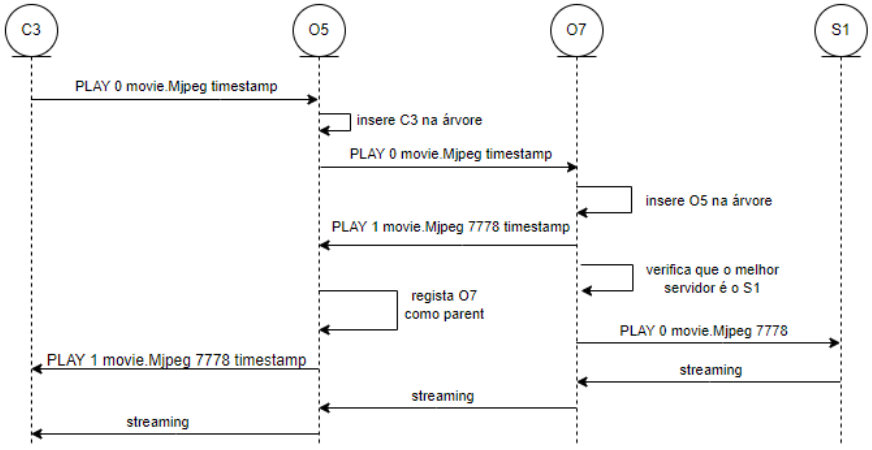

Figura 6: Cenário 1 - Caminho ótimo escolhido.

O O5 ao receber o pedido do cliente, insere o cliente na sua árvore. Em seguida, reencaminha este pedido a todos os seus vizinhos. O RP ao receber esta mensagem, insere o O5 na sua árvore, calcula a porta que será utilizada para o envio e receção do conteúdo pedido e responde com a mensagem recebida, sendo o seu *response code* alterado para 1 e acrescentada a porta no *header*.

O RP procede, então, à procura do melhor servidor de acordo com as métricas registadas. Ao selecionar o melhor, envia o mesmo pedido, relembrando que já contém a porta pela qual o servidor deve enviar o conteúdo. Com isto, há a criação de uma *thread* para executar o serviço de *streaming*.

Enquanto isto, o O5 recebeu a mensagem do RP e, portanto, regista o RP como seu *parent* na árvore, reencaminha a mensagem para o cliente, guarda a porta recebida como a que deve ser utilizada para este conteúdo e cria uma *thread* para executar o serviço de *streaming*.

O RP ainda receberá os pedidos provenientes dos nodos O6 e O2. Será exemplificado apenas o caso em que recebe do O6, visto que o cenário no qual recebe do O2 será análogo.

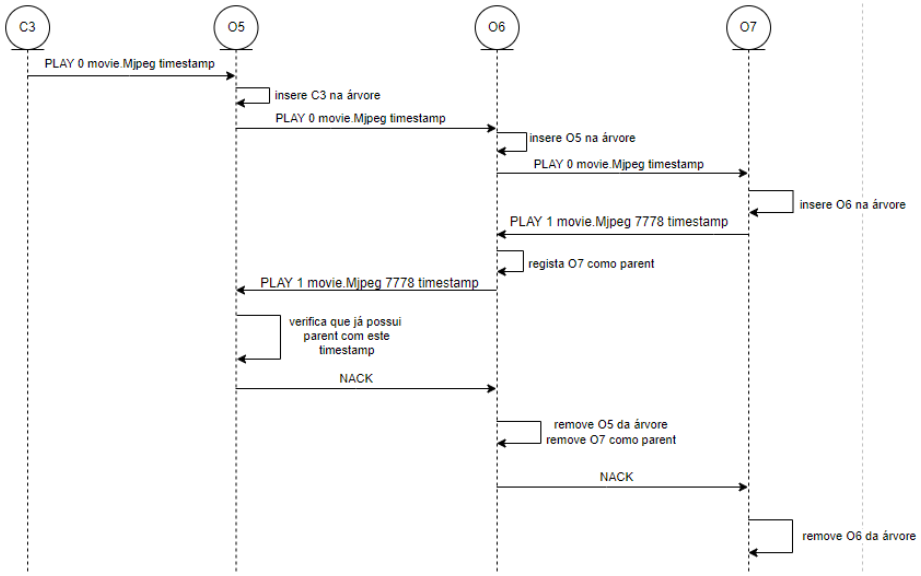

Figura 7: Cenário 1 - Restantes caminhos.

Neste fluxo, o O5 insere o cliente na sua árvore e reencaminha o pedido para o O6. Este insere o O5 na sua árvore e reencaminha a mensagem para o RP. Por sua vez, o RP insere o O6 na sua árvore e responde com a porta a ser utilizada. Neste caso, não precisa de calcular a porta pois esta já está registada para o conteúdo solicitado.

Ao receber a resposta, o O6 regista o RP como *parent* e reencaminha-a para o O5. Aquando da receção do pedido, este verifica que já possui um *parent* com o mesmo *timestamp* recebido e, portanto, envia uma mensagem NACK para o O6.

Ao receber este NACK, o O6 remove tanto o O5 da árvore quanto o RP como seu *parent* e reencaminha a mensagem ao RP. Este ao recebê-la, remove o O6 da sua árvore.

Cenário 2

Neste cenário, o O5 encontra-se a transmitir o conteúdo para o C3. O C1 envia um pedido de subscrição do mesmo conteúdo ao O3 e este reencaminha-o para os seus vizinhos.

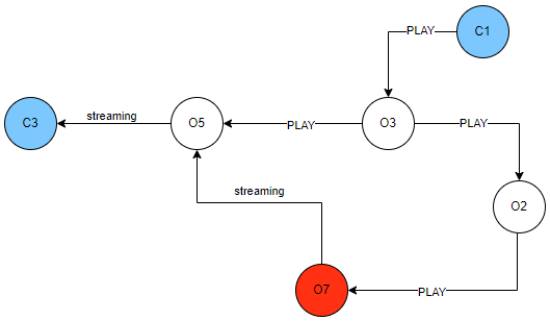

Figura 8: Segundo cenário de subscrição na árvore.

A figura 9 ilustra o fluxo referente ao O3 receber a confirmação do O5 antes das outras possíveis confirmações.

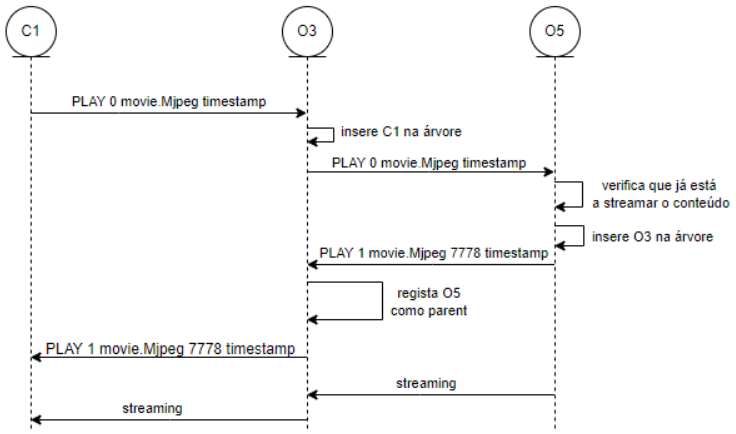

Figura 9: Cenário 2 - O3 recebe primeiro a confirmação do O5.

O O5 ao receber o pedido do O3 verifica que já se encontra a transmitir este conteúdo e, portanto, não reencaminha a mensagem para os seus vizinhos. Prossegue, portanto, a inserir o O3 na sua árvore e envia a mensagem de confirmação que contém a porta pela qual o conteúdo será transmitido.

O O3 ao receber esta mensagem, regista o O5 como o seu *parent*, reencaminha a confirmação para o cliente e cria uma *thread* para receber o *streaming*.

A figura 10 exemplifica o fluxo relacionado a quando o O3 recebe primeiramente a confirmação de outro nodo.

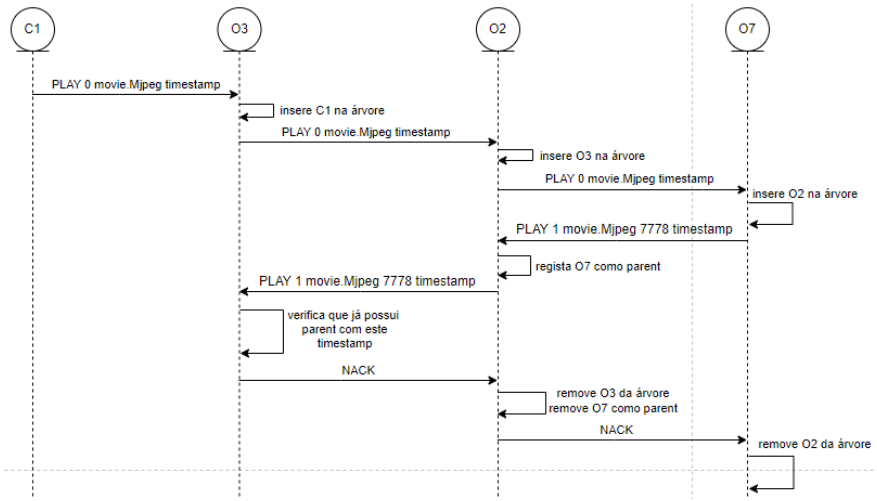

Figura 10: Cenário 2 - O3 recebe confirmação através de outro nodo.

Neste caso, o fluxo de ida até ao RP é análogo ao explicado anteriormente. Quando o O3 recebe a confirmação, verifica que já possui o O5 como *parent* e que os *timestamps* são iguais. Portanto, envia uma mensagem NACK de volta ao O2. Este remove o O3 da sua árvore e o O7 como *parent* e reencaminha a mensagem para o RP. Este, por fim, remove o O2 da sua árvore.

3.2.4 Saída da Árvore

Assim que o cliente já se encontra a receber conteúdo, o mesmo pode indicar que pretende deixar de o receber, enviando uma mensagem do tipo `LEAVE`.

Quando um nodo recebe um pedido de `LEAVE`, este procede a remover o remetente deste pedido da sua árvore e verifica se ainda há outros nodos associados ao conteúdo. No caso de ainda existir, não propaga esta mensagem. Caso contrário, reencaminha a mensagem ao seu *parent*.

O RP, ao receber este pedido, possui o mesmo comportamento explicado acima. Entretanto, caso não tenha mais interessados pelo conteúdo, propaga a mensagem ao servidor para que o *streaming* seja devidamente interrompido, minimizando o número de fluxos e garantindo a existência dos fluxos de transmissão de conteúdo estritamente necessários.

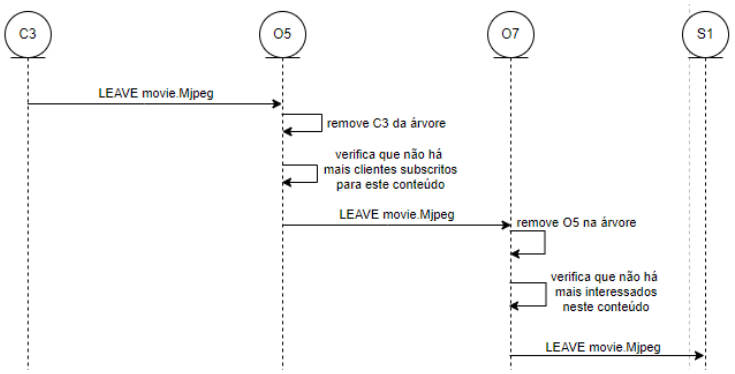

Figura 11: Leave - Fluxo relacionado à mensagem de `LEAVE`.

3.2.5 *Polling*

Suponhamos que o primeiro caminho ótimo selecionado para o cliente 3 foi `C3 -> O5 -> RP`. O cliente envia uma mensagem de *polling* que é reencaminhada com o mesmo comportamento do `PLAY` anteriormente esclarecido.

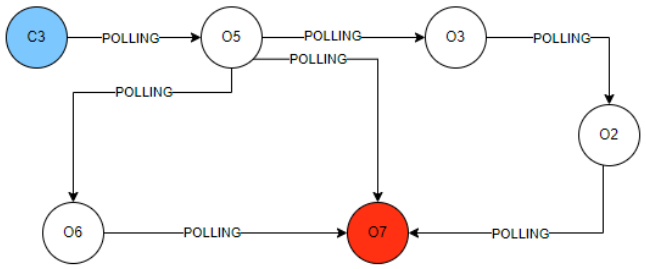

Figura 12: Cenário de propagação de uma mensagem de *polling*.

Caso o caminho ótimo permaneça igual, isto é, o RP receba primeiro o pedido proveniente do nodo `O5`, a árvore é equivalente. Admitamos então que o RP recebe primeiro o pedido relacionado ao fluxo ilustrado na figura 13, portanto, decorrente do nodo `O2`.

O RP envia uma mensagem de confirmação ao `O2` e esta é propagada até o nodo `O5`. Este verifica que o *parent* atual possui o *timestamp* inferior ao recebido neste instante e, portanto, substitui-o por `O3`.

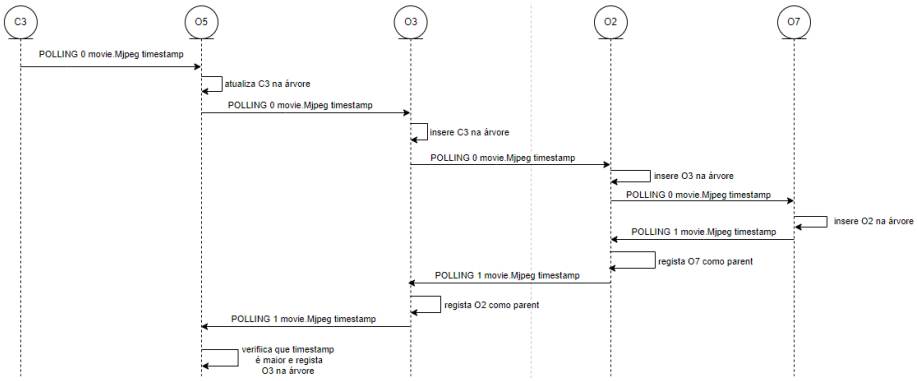

Figura 13: *Polling* - Novo caminho ótimo identificado.

A figura 14 demonstra o fluxo referente ao momento em que o RP recebe a mensagem de *polling* do `O5` após a atualização do caminho ótimo.

Neste caso, o RP envia a mensagem de confirmação ao `O5` e este verifica que já possui um *parent* com *timestamp* equivalente. Por conseguinte, envia um NACK para o RP e este remove o `O5` da sua árvore.

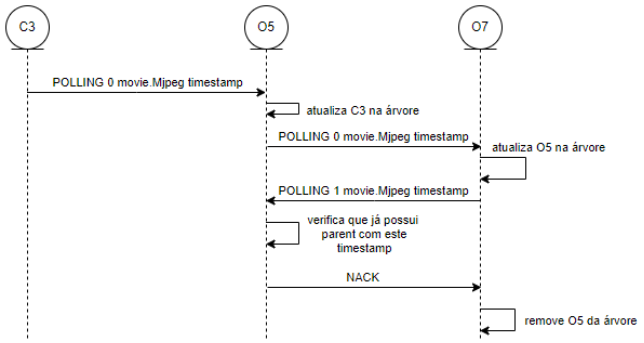

Figura 14: *Polling* - `O5` recebe confirmação do RP.

4 Implementação

4.1 *Bootstrapper*

A função do *bootstrapper* é fornecer as informações necessárias aos restantes nodos assim que estes são arrancados. Esta função pode ser atribuída a qualquer nodo com a *flag* `-file {file_path}`. Após o arranque deste nodo, deve ser feita a leitura e armazenamento da informação sobre os vizinhos de cada nodo e os servidores disponíveis.

Para que o *bootstrapper* tenha a informação necessária, escolhemos providenciá-la através de um ficheiro JSON, sendo esta diretamente traduzida num dicionário `nodes` armazenado na instância. Este ficheiro deverá ser composto por 3 objetos: um para indicar os nodos, outro para indicar os servidores e outro para as interfaces do RP. O primeiro objeto, por sua vez, é composto por vários objetos, correspondentes a cada nodo, que contêm as interfaces do nodo em questão e as interfaces dos seus vizinhos. O segundo objeto contém as interfaces dos servidores existentes na topologia pelas quais o RP deverá contactá-los. O terceiro objeto é útil para o *bootstrapper*, na medida em que é necessário verificar se o IP do remetente da mensagem pertence à lista de interfaces do RP para que seja ou não enviada a lista de servidores.

```
{
    "nodes": {
        "n1": {
            "interfaces": ["10.0.0.20"],
            "neighbours": ["10.0.0.1"]
        },
        "n2": {
            "interfaces": ["10.0.1.20"],
            "neighbours": ["10.0.1.1"]
        },
        "n3": {
            "interfaces": ["10.0.0.1", "10.0.5.2", "10.0.3.1"],
            "neighbours": ["10.0.5.1", "10.0.3.2"]
        },
        "n4": {
            "interfaces": ["10.0.1.1", "10.0.5.1", "10.0.2.1"],
            "neighbours": ["10.0.5.2", "10.0.2.2"]
        },
        "n5": {
            "interfaces": ["10.0.3.2", "10.0.2.2", "10.0.4.1"],
            "neighbours": ["10.0.3.1", "10.0.2.1", "10.0.4.10"]
        }
    },
    "servers": ["10.0.4.10"],
    "rp": ["10.0.3.2", "10.0.2.2", "10.0.4.1"]
}
```

Figura 15: Estrutura do ficheiro.

4.2 Nodo

Relativamente à implementação da aplicação que vai correr nos nodos *overlay*, optamos por desenvolver duas aplicações separadas - `Node` e `RP`.

O nodo começa a sua execução a contactar o *bootstrapper*, nodo responsável por conhecer toda a topologia *overlay*, que irá responder com todas as ligações possíveis do nodo. De notar que, em caso de falha, o nodo irá fazer sucessivos *retries* até obter uma resposta do *bootstrapper*.

Todo o comportamento normal do nodo é assegurado através de 2 serviços que são corridos em 2 *threads* disjuntas: serviço de controlo e serviço de *pruning*. O primeiro é responsável por receber todas as mensagens de controlo através de um *socket* associado à porta 7777. Este serviço vai invocar o respetivo *worker* de controlo, responsável por tratar de todo o comportamento do nodo. O nodo

possui ainda um serviço de *pruning* que vai consultar a estrutura com os últimos contactos dos clientes presentes na árvore e remover da árvore se não tiver obtido uma comunicação nos últimos 5 segundos. Para além destes serviços, ao longo da execução da aplicação, serão criadas *threads* responsáveis por receber um certo conteúdo numa determinada porta predefinida pelo RP e difundi-lo pelos nodos da sua árvore que o pretendem.

Este componentes possuem estruturas de dados cuidadosamente escolhidas de forma a possibilitar a implementação de toda a lógica referente à árvore e ao *streaming*.

- **Neighbours:** Uma lista que contém todos os vizinhos do nodo na topologia. Estes vizinhos são adquiridos através do contacto com o *bootstrapper*.
- **Tree:** Árvore dinâmica partilhada. Esta árvore regista o nodo para o qual o conteúdo será reenchaminhado e o *parent*, ou seja, o nodo com ligação direta em sentido ascendente (caminho até ao RP) em conjunto com um *timestamp*. Isto possibilita que o caminho ótimo possa ser recalculado e ajustado. Caso o caminho mude, os nodos propagam a mensagem de NACK para o seu *parent* até que esta alcance o RP.
- **Contacts:** Dicionário responsável por guardar o *timestamp* do último contacto feito pelo cliente. Desta forma, podemos realizar o *pruning*.
- **Ports:** Dicionário para registar as portas a serem utilizadas para a transmissão de cada conteúdo.

4.3 Rendezvous-Point (RP)

O RP trata-se de um tipo de nodo particular, uma vez que é responsável por receber em *unicast* o conteúdo a distribuir e propagá-lo pelo conjunto de nodos da árvore. Deste modo, este herda todas as funcionalidades dos restantes nodos e tem, ainda, algumas funcionalidades adicionais. Para representarmos o que foi dito, implementamos uma hierarquia, onde a classe `RP` é uma subclasse da classe `Node`.

O RP terá de correr ainda um serviço responsável por pedir e calcular as métricas dos servidores. As métricas são calculadas com base no *delay* e *loss* da ligação entre os servidores, atribuindo um peso de 70% e 30% respetivamente para o cálculo da métrica final. Para armazenar as informações associadas a cada um dos servidores, o RP faz uso da classe auxiliar `ServerInfo`.

O RP vai ser responsável por atribuir as portas a cada conteúdo na rede. Desta forma, nos pedidos de confirmação de subscrição enviará no *payload* do pacote a respetiva porta para o conteúdo pedido. Os nodos, ao receberem esta porta, irão registá-la na sua estrutura `ports` acima referida e, por consequente, criar a *thread* responsável por receber os pacotes de *streaming*.

Para além das estruturas de dados mencionadas, este componente possui uma adicional.

- **Servers:** Dicionário que regista todos os servidores disponíveis. Para cada servidor, guarda as métricas, os conteúdos que este possui e o *status* que diz se este servidor está a transmitir algum conteúdo nesse instante.

4.4 Servidor

A implementação do servidor teve como base o código fornecido. O servidor possui também uma *thread* onde corre o serviço de controlo e, assim que recebe um pedido do tipo `PLAY` proveniente do RP, é criada uma *thread* para o envio desse conteúdo. O servidor pode também receber mensagens do tipo `LEAVE`, onde deve interromper o envio do conteúdo em questão, e mensagens do tipo `STATUS`, onde deve responder com os conteúdos que está a transmitir nesse instante.

Em relação às estruturas de dados, o servidor contém um dicionário `videostreams` onde guarda informação relativa aos conteúdos que transmite, sendo a informação respetiva a cada conteúdo guardada na classe `VideoStream`.

4.5 Cliente

O cliente trata-se de outra aplicação devidamente adaptada do código fornecido. O cliente manifesta os seus pedidos na interface *tkinter* onde possui 3 botões: `PLAY`, `STOP` e `EXIT`. Assim que o cliente é inicializado, é desde logo enviado um pedido do tipo `PLAY`. Ao pausar a reprodução, clicando no botão `STOP`, o cliente envia um pedido do tipo `LEAVE`. Caso queira voltar a receber conteúdo, clicando no botão `PLAY`, é enviado um novo pedido de subscrição na árvore. Caso o cliente clique no botão `EXIT`, a janela da interface é destruída e é propagada uma mensagem do tipo `LEAVE`, assim como quando este pausa a reprodução.

Desde a sua inicialização, o cliente possui 2 *threads* a correr o serviço de controlo e o de *polling*. O serviço de controlo é responsável por todas as mensagens de controlo recebidas, nomeadamente de mensagens de confirmação da subscrição na árvore. Da mesma maneira que as outras aplicações, o cliente inicia uma *thread* para a receção de conteúdo, assim que recebe uma mensagem de confirmação de subscrição. O serviço de *polling* trata de manter o cliente na árvore através do envio de mensagens do tipo `POLLING`. Desta maneira, a subscrição do cliente na árvore não será removida pelo serviço de *pruning* dos nodos, visto que está constantemente a contactar os nodos. De outra forma, caso a aplicação do cliente tenha uma saída indesejada em que não envie uma mensagem `LEAVE` para ser retirado da árvore, o serviço de *pruning* dos nodos irá tratar deste erro, removendo-o da árvore.

5 Limitações da Solução

Independentemente de estarmos satisfeitos com a solução final, estamos conscientes de que esta sofre de algumas limitações.

Uma das limitações notáveis é a falta de suporte para o envio de outros formatos de vídeo, diferentes de MJPEG. Esta funcionalidade é, de facto, imprescindível para uma plataforma de *streaming*. Contudo, no contexto deste projeto, consideramos ser um requisito menos prioritário, pelo que acabou por não ser implementado. Para tal, poderíamos utilizar a biblioteca *cv2* para implementar o suporte a outros formatos existentes.

Outra limitação foi não conseguir obter outro vídeo com o formato MJPEG. Apesar disto, este projeto foi planeado e concebido para suportar o *streaming* de conteúdos diferentes, conforme ilustrado nas estruturas de dados definidas e explicado nas secções anteriores.

Em relação aos *locks*, identificamos que os *read and write locks* seriam a decisão mais prudente. Isto devido ao facto de existir um desbalanceamento significativo entre operações de *read* e *write* no qual a leitura se sobrepõe. Neste sentido, seria mais eficiente poder efetuar estas leituras em simultâneo.

Entretanto, a ausência destes *locks* na linguagem de programação escolhida levou-nos a utilizar *locks* comuns. Uma forma de ultrapassar esta limitação seria a implementação manual da lógica destes *locks* ou a utilização de uma biblioteca como *rwlock*, *readerswriterlock* ou *fasteners*. Porém, seria necessário despender um tempo suficiente de análise e estudo das bibliotecas para decidir qual abordagem seria a melhor entre a manual e a com recurso às bibliotecas, para além de qual biblioteca seria a mais eficaz.

6 Manual de Utilização

É possível executar todos os componentes em modo *debug* ao adicionar a *flag* `-d`.

6.1 Cliente

Para executar o cliente é necessário fornecer o endereço IP do *bootstrapper* e o conteúdo que deseja receber.

```
python3 client.py -b <bootstrapper-ip> -f <conteudo>
```

6.2 Nodo e RP

Para executar um nodo da topologia ou o RP há duas possibilidades. Caso este seja também o *bootstrapper*, é necessário fornecer o ficheiro JSON que contém a informação sobre a topologia.

6.2.1 Nodo

```
python3 node.py -b <bootstrapper-ip>
ou
python3 node.py -b <bootstrapper-ip> -f <bootstrapper-file>
```

6.2.2 RP

```
python3 rp.py -b <bootstrapper-ip>
ou
python3 rp.py -b <bootstrapper-ip> -f <bootstrapper-file>
```

6.3 Servidor

Para executar o servidor, é necessário fornecer os conteúdos que o servidor terá acesso para transmitir.

```
python3 server.py -videostreams <conteúdos>
```

7 Testes e Resultados

O teste contínuo em cada etapa do desenvolvimento desempenha um papel fundamental na identificação e correção de problemas, o que contribui para um ciclo de desenvolvimento mais eficiente e económico. Deste modo, inicialmente, construímos a topologia sugerida no enunciado, servindo esta como base para todos os testes efetuados durante o desenvolvimento do projeto, uma vez que pretendíamos testar o maior número de cenários possíveis, sendo eles esperados ou inesperados, por forma a manter a nossa aplicação robusta perante essas situações.

Todavia, consideramos que a topologia referida introduz complexidade na rede *overlay* que, apesar de ser necessária para alguns testes, deve ser evitada na apresentação de alguns cenários para uma melhor compreensão. Neste sentido, construímos também os cenários de teste sugeridos no enunciado para efetuar e apresentar alguns testes neste relatório. Nesta secção apresentaremos apenas alguns testes mais simples que envolvem poucos componentes e que o comportamento é facilmente compreendido pelos *logs*.

7.1 Pedido de Vizinhos

Para testar o pedido de vizinhos, utilizou-se o cenário 1. Para analisar o comportamento do cliente perante a falha na receção da resposta, arrancou-se primeiro o cliente e só depois o *bootstrapper*, passados alguns segundos.

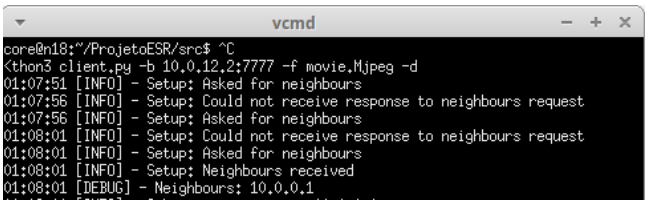

Figura 16: Comportamento do cliente.

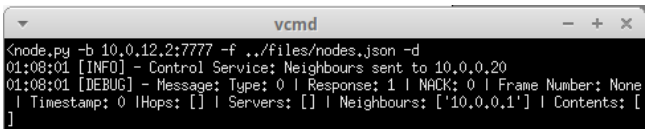

Figura 17: Comportamento do *bootstrapper*.

O *bootstrapper* foi iniciado 10 segundos após o cliente. Durante estes 10 segundos, o cliente efetuou 2 novos pedidos de vizinhos, como esperado, uma vez que caso não receba uma resposta, ele voltará a tentar 5 segundos depois. Assim que iniciamos o *bootstrapper*, podemos verificar que este recebeu o pedido com sucesso e respondeu com os vizinhos que foram também recebidos com sucesso no lado do cliente.

7.2 Serviço de *Pruning*

Para testar o serviço de *pruning*, utilizamos os nodos `C3`, `O5`, `O7` e `S1` da rede *overlay* base.

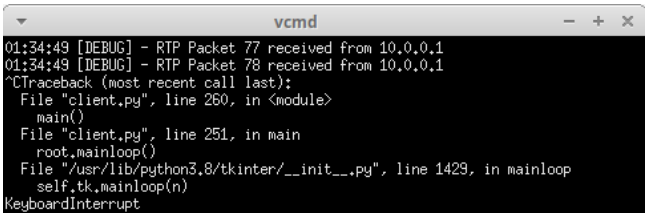

Figura 18: Comportamento do cliente.

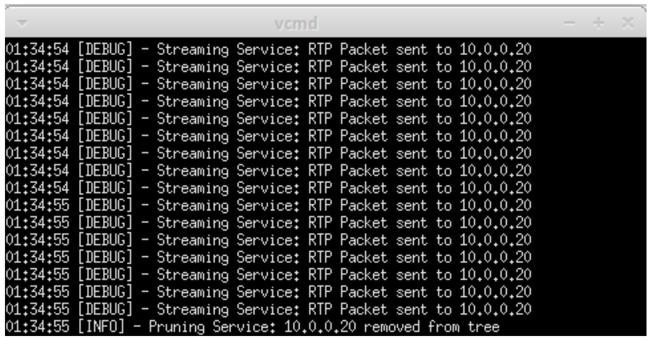

Figura 19: Comportamento do nodo intermediário.

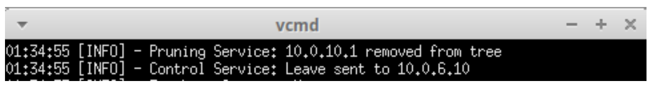

Figura 20: Comportamento do RP.

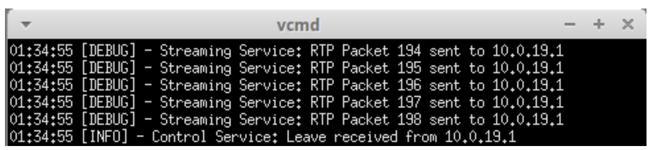

Figura 21: Comportamento do servidor.

Para analisar o comportamento dos diferentes componentes forçamos a interrupção da aplicação do cliente. Podemos verificar que alguns segundos depois, quando a *thread* do nodo intermediário responsável pelo serviço de *pruning* acordou para efetuar a respetiva ronda, removeu o cliente da sua árvore, uma vez que já não recebia contacto por parte deste há mais tempo que o tempo definido. Este nodo procede a propagar uma mensagem do tipo `LEAVE` no sentido ascendente, neste caso diretamente para o RP que é a sua ligação direta, que, ao receber esta mensagem, remove o `O5`, nodo intermediário, da sua árvore. Como verifica que não há mais clientes para o conteúdo em questão, o RP reencaminha a mensagem para o servidor para que este também interrompa a sua transmissão. Deste modo, concluímos que efetivamente reduzimos o número de fluxos, uma vez que tanto o nodo intermediário como o RP interrompem a sua receção e propagação do conteúdo, assim como o servidor, que deixa de transmitir o conteúdo.

7.3 *Streaming* para 2 clientes

Para testar o *streaming* para 2 clientes foi utilizado o cenário 2. Assim como visualizamos na figura abaixo, ambos os clientes estão a receber o conteúdo e a reproduzir o mesmo *frame*, como esperado e pretendido na implementação deste projeto.

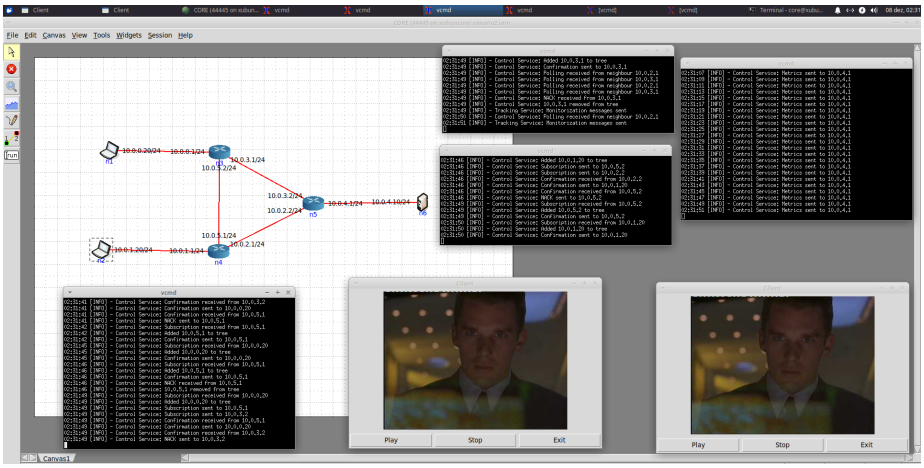

Figura 22: *Streaming* para 2 clientes.

8 Conclusão

Para concluir, acreditamos que a elaboração deste projeto nos fez ter uma melhor perceção do funcionamento do serviço *multicast* em rede, bem como do serviço de *streaming* que é algo muito utilizado no nosso dia a dia. Consideramos a implementação deste projeto desafiante dado o tempo disponível, sendo que existem algumas funcionalidades que eram do nosso interesse e acabaram por não ser implementadas. Este projeto abordou conceitos como *bootstrapper*, árvore dinâmica partilhada, *polling* e *pruning*. A execução deste projeto demonstrou a viabilidade prática dessas técnicas num contexto realista, desafiando e expandindo a nossa compreensão sobre o assunto.

A construção do *bootstrapper* revelou-se importante, garantindo a inicialização eficaz do serviço. A implementação da árvore dinâmica partilhada proporcionou uma estrutura flexível e escalável, essencial para o controlo de múltiplas conexões simultâneas. O mecanismo de *polling* foi implementado com sucesso, oferecendo um meio eficiente de manter a integridade e a atualização do serviço. Por fim, o processo de *pruning* demonstrou a sua eficácia na manutenção da eficiência do sistema, removendo elementos desnecessários.

De um modo geral, estamos satisfeitos com o trabalho desenvolvido e com a solução obtida, uma vez que fomos optando sempre pela estratégia mais complexa a cada etapa e terminamos com uma solução altamente preparada para receber novos clientes, conteúdos diferentes, novos servidores e novos fluxos.