Trabalho Prático #2 – Serviço *Over the Top* para entrega de multimédia

Simão Cunha[a93262], Tiago Silva[a93277], and Gonçalo Pereira[a93168]

Universidade do Minho - Campus de Gualtar, R. da Universidade, 4710-057 Braga Portugal

Engenharia de Serviços em Rede (2022/2023) - PL7 - Grupo 2

Resumo Este segundo trabalho prático surge no âmbito da UC de Engenharia de Serviços em Rede, onde nos foi proposto a elaboração de um protótipo de entrega de áudio/vídeo/texto com requisitos de tempo real, a partir de um servidor de conteúdos para um conjunto de N clientes. Este relatório começa com uma introdução, onde falamos sobre a motivação deste trabalho, passando pela arquitetura da nossa solução, pela especificação dos protocolos usados - formato das mensagens protocolares e interações entre entidades e pela implementação - realizada em Python. De seguida, temos uma secção de testes e resultados, onde exemplificámos o uso da nossa solução numa topologia criada no simulador Core e finalizámos com uma secção de conclusões e de trabalho futuro.

Keywords: Servidor, Cliente, Bootstrapper, RTP, Streaming

1 Introdução

Ao longo do último meio século de vida da Internet, observou-se uma evolução notável na mesma. Tem havido um consumo massivo de conteúdos multimédia em tempo real e, para satisfazer este desejo, surgiram serviços específicos, desenhados sobre a camada aplicacional, e por isso ditos Over the Top (OTT). São exemplos de serviços de streaming a Twitch e o Netflix. Neste trabalho, pretendemos conceber um protótipo de entrega de vídeo com requisitos de tempo real, a partir de um servidor de conteúdos para um conjunto de N clientes. De forma a pôr em prática a nossa solução, recorreremos ao emulador CORE para os testes - iremos apresentar mais detalhes na secção 5.

2 Arquitetura da solução

De forma a idealizar a solução para responder ao problema do enunciado do trabalho prático, utilizamos a sugestão da equipa docente e seguimos as etapas definidas.

Com isto, o primeiro passo foi a definição da linguagem de programação, onde escolhemos Python por dois motivos: temos alguma experiência em utilizar esta linguagem em outros projetos de outras unidades curriculares e é uma das linguagens fornecidas pelo código de exemplo para o *streaming*. De seguida, criamos apenas uma topologia de teste (*overlay* e *underlay*) no emulador CORE. No enunciado eram sugeridas três, mas decidimos incluir a mais complexa dessas (*figura 1 do enunciado*). E, para terminar esta etapa, decidimos que o protocolo de transporte na rede *overlay* será UDP, uma vez que garante uma melhor velocidade na entrega de pacotes do que o TCP.

Na próxima etapa, passamos para a idealização da construção da topologia *overlay*. Optámos pela estratégia que pressupõe a existência de um nó bootstrapper, que trata da configuração da topologia, e, lendo um ficheiro de texto (`config.txt`), sabe quais são os nós da topologia, assim como os vizinhos de todos.

A próxima etapa a efetuar é a do tratamento do serviço de *streaming*. Para tal, adotamos a versão em Python do código de exemplo fornecido pela equipa docente. Numa primeira instância, completamo-lo com as respostas RTCP a enviar do cliente para o servidor e completamos o módulo

do pacote RTP, tendo conseguido experimentar este serviço em questão. No entanto, este código permitia uma ligação TCP entre cliente e servidor, o que não é desejável para a implementação da nossa solução.

De seguida, passamos para a etapa da monitorização da rede *overlay*. Esta monitirização é feita no momento em que o servidor conhece o seu nodo vizinho, a partir do qual manda uma mensagem de *probe* que é difundida pela rede *overlay*. Esta mensagem de *probe* tem uma marca temporal do servidor que é usada em cada nodo para guardar as possiveis em rotas e tempo que estas demoram até ao servidor e vice-versa. Cada nodo guarda apenas as melhores rotas, e apenas envia adiante na topologia se houve atualização da sua rota. A monitorização é feita periodicamente a cada 30 segundos. A validade das rotas em cada node só tem 30 segundos, mas esta só atualizada se receber uma nova probe.

Depois, passamos a etapa da construção de rotas para a entrega de dados. Quando um cliente deseja receber *stream*, o pedido é enviado para o seu vizinho. Por cada nodo que o pedido passa é criado um fluxo. Desta forma, cada nodo terá um fluxo por cada cliente que pede e esse fluxo tem o próximo salto até ao cliente que o pacotes RTP terão que seguir. De notar que os fluxos são criados usando os melhores tempos guardados nas tabelas de rotas construídas no momento da monitorização.

A última etapa do enunciado incide na ativação e teste do servidor alternativo. Extendemos a monitorização implementada de modo a suportar mais do que 1 servidor.

3 Especificação do(s) protocolo(s)

3.1 Formato das mensagens protocolares

OlyPacket

Entidade que implementa mensagens de controlo da rede *overlay*. As mensagens são em formato string, com um tamanho máximo de 250 bytes, sendo que os campos da mensagem são separados por ";".

Num objeto `Olypacket` existem 3 campos: Type (o tipo da mensagem a ser enviada), Payload (dados da mensagem) e Padding (campo preenchido com zeros apenas para garantir o tamanho afixado para o pacote). Nós criamos 7 tipos de mensagens, sendo estes:

- `HELLO` → Mensagem usada para fazer a ligação com o bootstrapper.
- `HELLORESPONSE` → Mensagem que o bootstrapper usa de resposta ao Hello. Esta tem de payload os vizinhos do nodo, que fez o HELLO, na topologia, assim como o ip interface que ele é conhecido nessa topologia.
- `PROBE` → Mensagem de probe. Tem como payload uma marca temporal do servidor, número de saltos e o ip da interface de quem a enviou.
- `SETUP` → Mensagem usada pelos clients/nodes para informar da criação de fluxo de streaming.
- `PLAY` → Mensagem usada pelos clients/nodes para informar da abertura de fluxo de streaming.
- `PAUSE` → Mensagem usada pelos clients/nodes para informar do fecho de fluxo de streaming.
- `TEARDOWN` → Mensagem usada pelos clients/nodes para informar da remoção de fluxo de streaming.

Como só o as mensagens HELLORESPONSE e PROBE precisam de enviar dados, são as únicas que tem payload.

Todas as mensagens têm padding.

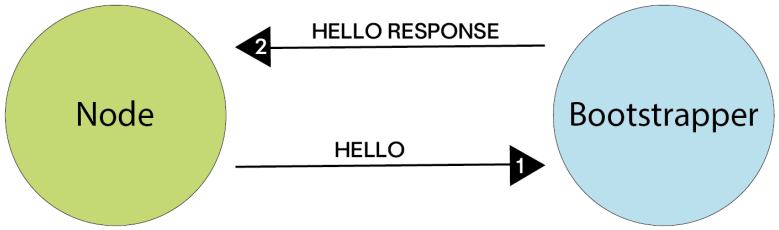
RtpPacket

Implementa as pacotes de streaming, ou seja, encapsula com um header RTP os bytes de um trecho do video a ser transmitido.

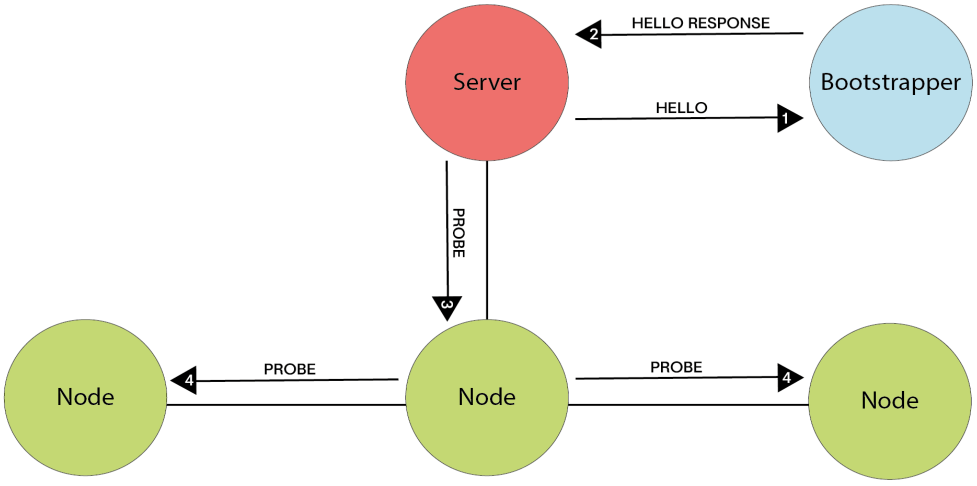
– `HEADER`: Campos de um cabeçalho rtp:
  - `Version` → Indica a versão do protocolo (2 bits)
  - `Padding` → Indica se existe bytes de enchimento no fim do pacote rtp (1 bit)
  - `Extension` → Indica a presença da extensão do header(1 bit)
  - `CSRC count` → Contêm o número de identificadores CSRC (4 bits)
  - `Marker` → Sinalizador (1 bit)
  - `Payload type` → Indica o formato do payload (7 bits)
  - `Sequence number` → Número de sequência dos pacotes, usado para o recetor detetar perda de pacotes (16 bits)
  - `Timestamp` → Marca temporal (32 bits)
  - `SSRC` → Identifica a origem de uma stream (32 bits)
  - `CSRC` → Enumera o conjunto das origem que contribuiram para a stream. Um por cada CSRC count. (32 bits)
– `PAYLOAD`:
  - Bytes de um trecho do video que está a ser transmitido.

3.2 Interações entre entidades

Nesta secção, iremos demonstrar o fluxo da nossa aplicação através de alguns diagramas.

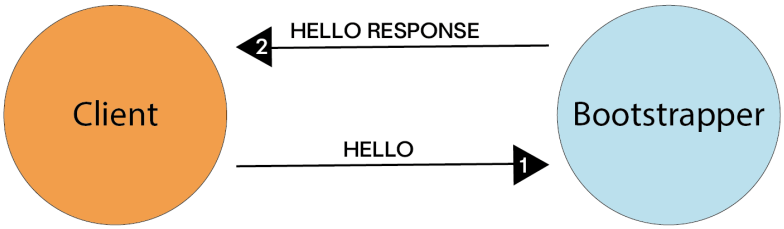


Neste primeiro diagrama, conseguimos entender como um nó se liga à topologia. Este envia um pacote de início de conexão (`Hello Packet`) ao *bootstrapper*, que responde-lhe com um pacote de resposta (`Hello Response Packet`).

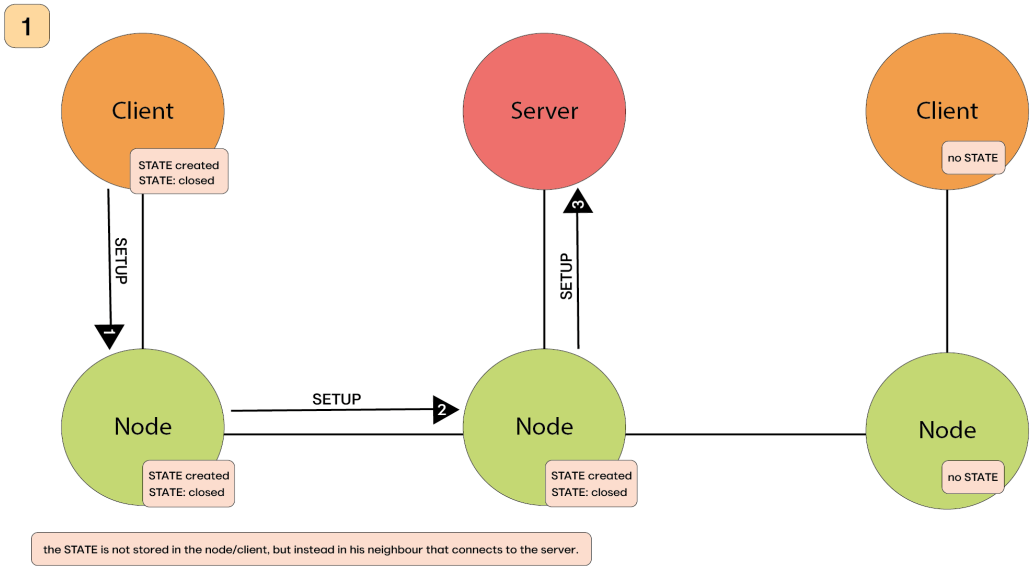


Aqui, é possível entender o processo de conexão do servidor à topologia. Procede da mesma forma que um nó e, recebendo de volta a informação dos seus nós vizinhos, envia um pacote de

probe apenas aos nós que estiverem ativos, de forma a estes saberem qual o caminho mais rápido até ele.

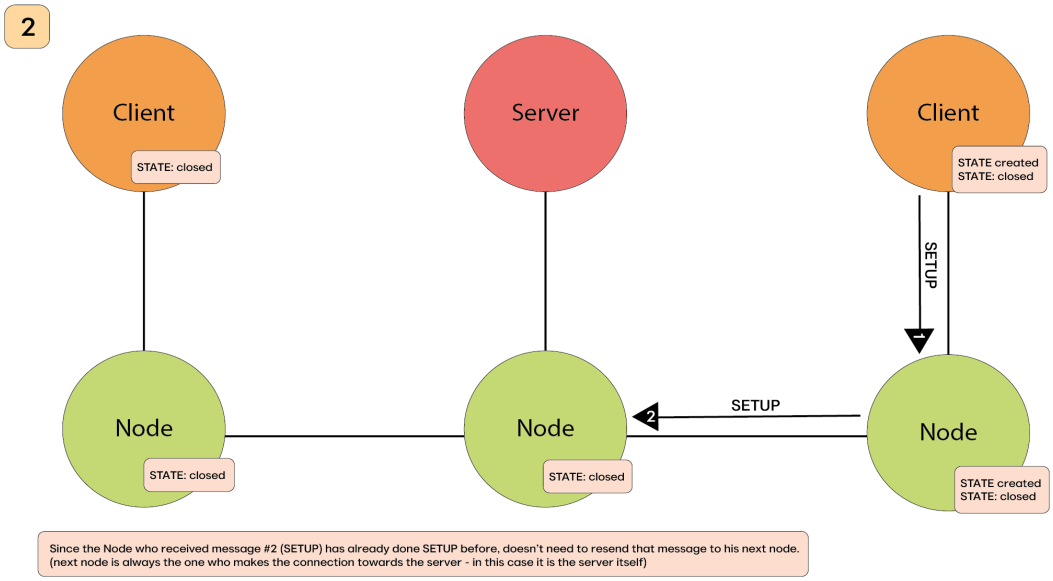


A ideia subjacente neste diagrama é semelhante ao diagrama de ligação de um nó mais acima descrito: cliente envia `Hello Packet` para informar que se quer conectar e o *bootstrapper* envia um pacote de resposta com os seus vizinhos.

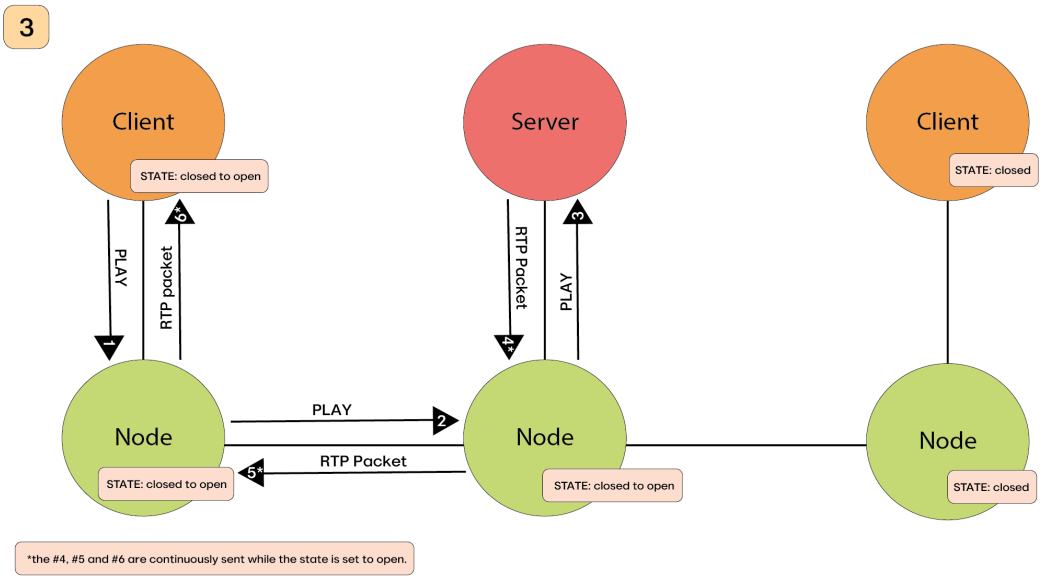


Nos próximos diagramas constarão 2 clientes, 3 nós e 1 servidor e explicaremos o que acontecerá quando cada um dos clientes manda mensagens do tipo `SETUP`, `PLAY`, `PAUSE` e `TEARDOWN`. Pressupomos que todas estas entidades já têm as suas ligações efetuadas da mesma forma que a explicada nos diagramas anteriores.

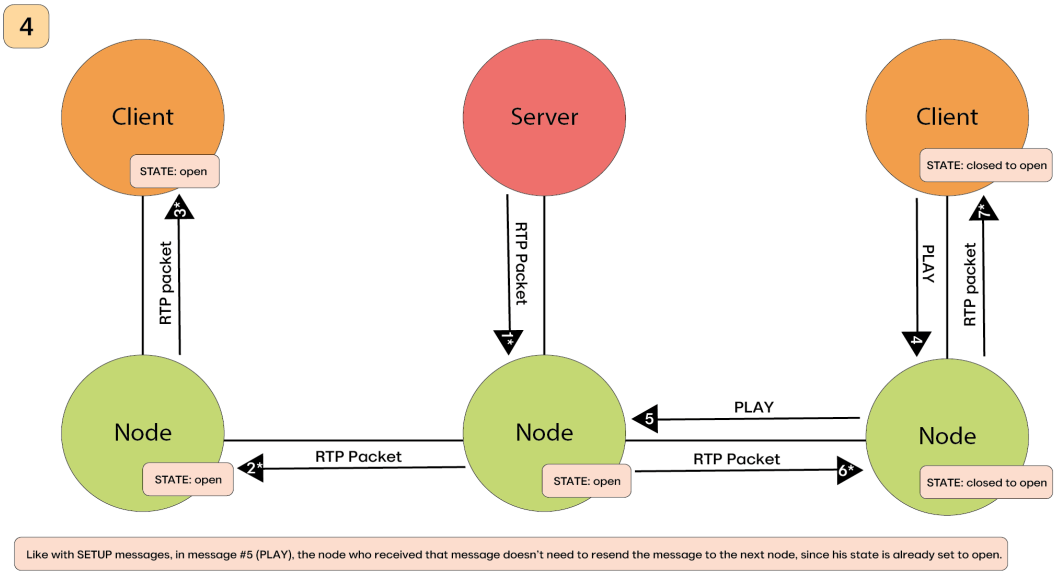
Neste primeiro diagrama, um dos clientes envia uma mensagem de `SETUP` que irá ser enviada para o seu nó vizinho que trata de o encaminhar até ao servidor, com o intuito de mostrar o interesse de ver a *stream* emitida pelo servidor. Tanto no cliente como nos nós participantes no envio da mensagem é criado um estado em que, neste momento, se encontra a fechado - ainda não recebe pacotes de *stream*.



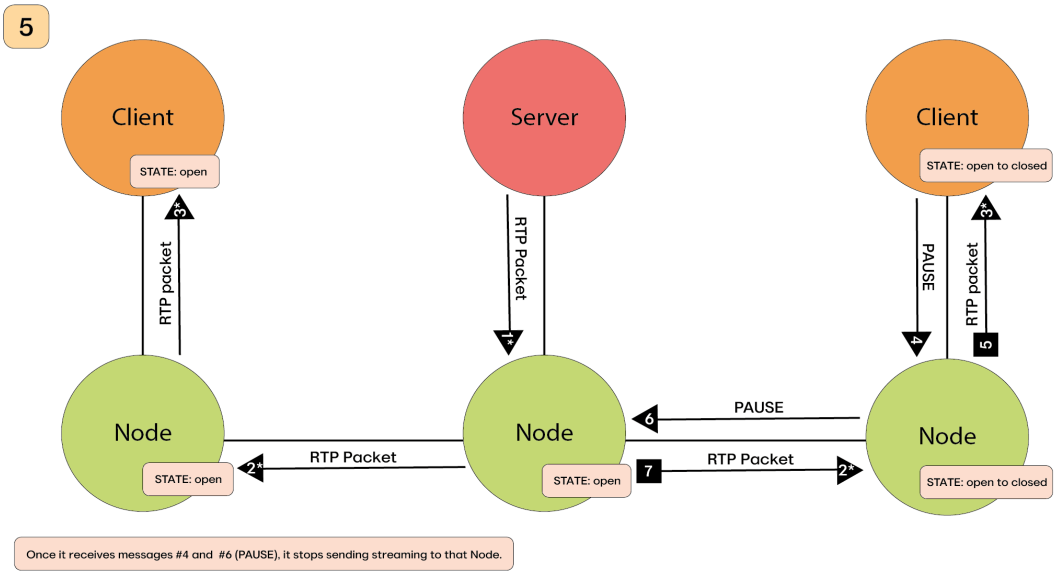
Neste caso, outro cliente também envia uma mensagem de `SETUP` da mesma forma que a explicada no diagrama anterior com uma pequena exceção: um dos nós, que também recebeu a mensagem de `SETUP` do primeiro cliente (e também recebeu deste novo cliente), já tem o seu estado criado - quando assim acontece, não cria novo estado nem informa os seus vizinhos (incluíndo o servidor).



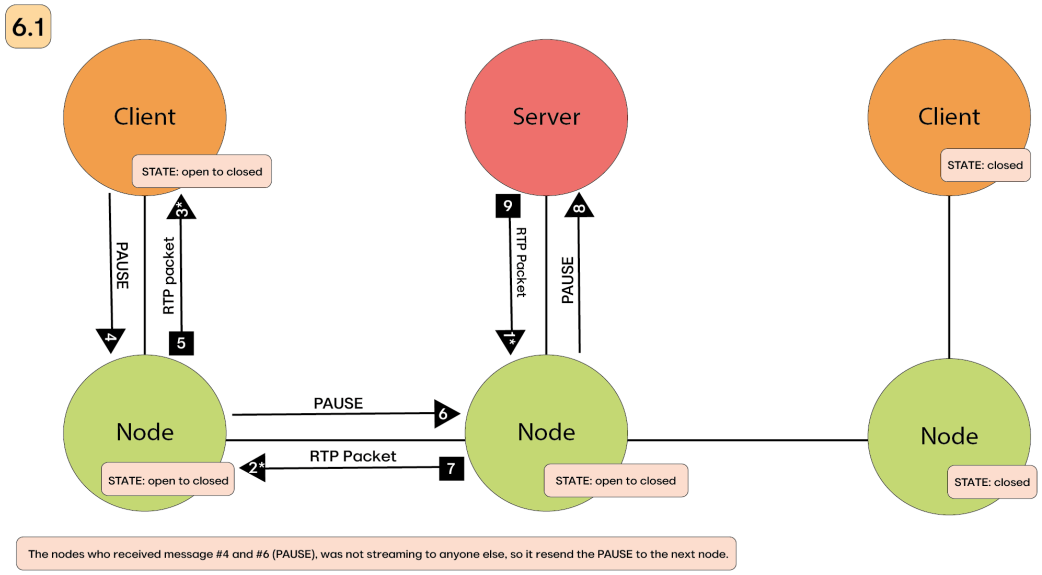
Nesta fase, ambos os clientes já enviaram as suas respetivas mensagens de `SETUP` e, neste diagrama, o primeiro cliente deseja receber a *stream*, enviando uma mensagem `PLAY` para os seus vizinhos (e o seu estado passa a ser `open`, tal como acontecerá em todos os nós que receberem uma mensagem deste tipo), que será reencaminhada até ao servidor (tal como observado na figura) ou até um dos nós que já contém pacotes de *stream* RTP. O servidor, depois de receber esta mensagem, começa a enviar pacotes RTP.



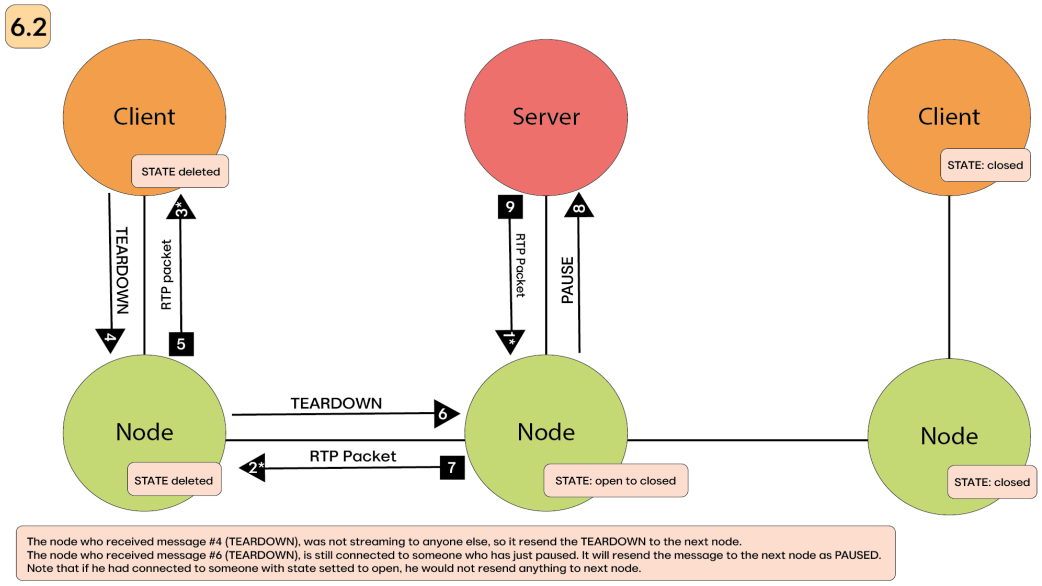
Já neste caso, o segundo cliente também deseja receber a *stream* e, de igual modo ao explicado para o envio das mensagens `SETUP`, envia a mensagem `PLAY` até a um nó que já recebe pacotes RTP, enviando-os para o cliente em questão.



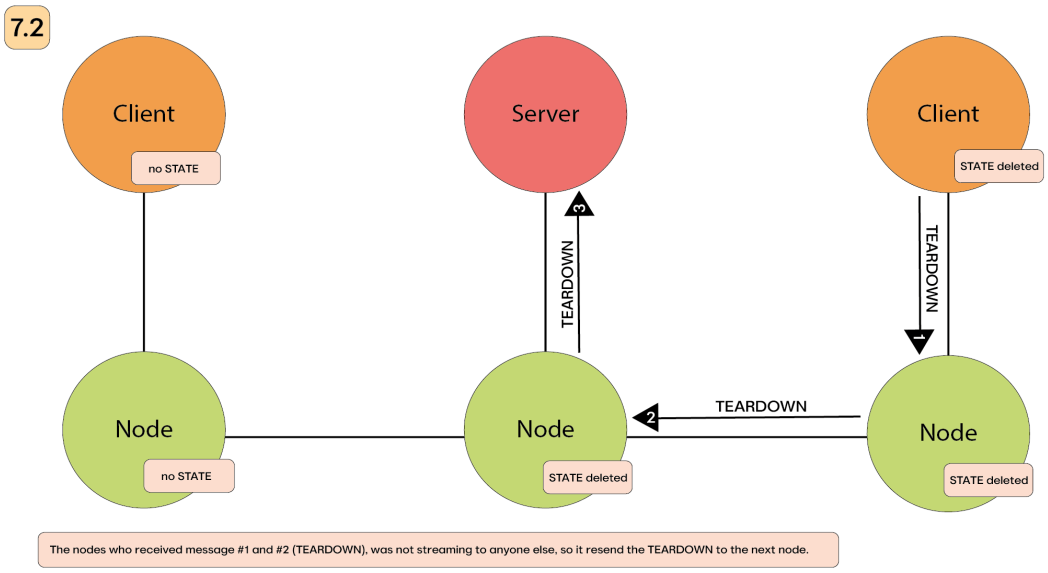
Seguindo do caso anterior, um dos clientes decide colocar a sua *stream* em pausa, enviando a mensagem de `PAUSE` ao seu vizinho que envia sucessivamente a nós pela topologia até ao servidor, fechando os seus fluxos de *streaming*. Como um dos nodos que recebeu a mensagem `PAUSE` ainda tem fluxos de *streaming* abertos de outros nodos/clientes, este nó simplesmente deixa de enviar pacote RTP até ao cliente que requisitou a pausa na *stream*.



De seguida, se o outro cliente também enviar a mensagem de `PAUSE`, podemos ver no nodo que faz a ligação com o servidor, já não existem mais fluxos de *streaming* abertos, enviando também ele a mensagem de `PAUSE` ao servidor.



Ainda continuando do cenário #5 (em que temos um cliente com o fluxo ligado e outro com o fluxo desligado), se o cliente com o fluxo ligado enviar a mensagem de `TEARDOWN` - que elimina o seu fluxo de *stream* - esta, quando chega ao nodo mais perto do servidor, como ainda possui um fluxo fechado, apenas envia a mensagem de `PAUSE` ao servidor, cainda no mesmo cenário que o #6.1. De notar que, se o fluxo do cliente do lado direito estivesse aberto, o nodo em baixo do servidor nem enviaria a mensagem de `PAUSE` pelas razões explicadas no cenário #5.



Concluindo, temos um cenário que sucede do #6.2, onde o outro cliente também envia a mensagem de `TEARDOWN`, destruindo com todos os fluxos de *streaming* da topologia, uma vez que mais nenhum cliente se encontra ligado.

4 Implementação

Tal como já foi dito anteriormente, a nossa solução foi escrita em Python, concretamente a versão 3.8.10. De forma a permitir a utilização do nosso programa, será necessário instalar algumas bibliotecas Python num ambiente Ubuntu tais como:

- `python3 -m pip install -upgrade pip`
- `python3 -m pip install -upgrade Pillow`
- ...

PIL	Usado para a construção da janela de *streaming*
socket	Usado para a comunicação entre nós
tkinter	Usado para pop-ups no lado do cliente
sys	Usado para aceder a argumentos recebidos pela linha de comandos
threading	Usado para permitir o manuseamento de vários clientes em simultâneo
os	Usado para a saída do programa no lado do cliente quando faz TEARDOWN
datetime	Usado para timestamps
time	Usado para timestamps
traceback	Usado para a visualização de exceções lançadas pelo programa

Tabela 1: Bibliotecas usadas no trabalho prático

De forma a implementar todas as ideias estabelecidas na secção 2, temos ao nosso dispor várias classes.

4.1 oNode.py

Este módulo implementa 4 modos de funcionamento diferentes:

Bootstrapper	Uso da flag `-bs` e passando como argumento o ficheiro de configuração da rede *overlay.*
Nodo	Uso da flag `-n` e passando como argumento o IP do bootstraper.
Cliente	Uso da flag `-c` e passando como argumento o IP do bootstraper.
Servidor	Uso da flag `-s` e passando como argumento do IP do bootstrapper e o ficheiro de video

4.2 Node.py

Módulo que trata de todo o funcionamento de um Nodo da topoligia.

4.3 Bootstrapper.py

Módulo que trata de todo o funcionamento do bootstrapper. Acedendo ao ficheiro de consiguração carrega para memória a constituição da rede overlay. A unica função do bootstraper é ficar à escuta de pedidos de ligação (HELLO packet) e responde com os vizinhos do nodo que se ligou (HELLO RESPONSE packet).

4.4 ClientLauncher.py

Módulo que trata de lançar o cliente. Começa por fazer um pedido ao bootrstrapper sobre o seu nodo vizinhos. Recebendo o seu nodo vizinho no HELLORESPONSE inicia uma *thread* com o objeto `Client`.

4.5 Client.py

Módulo que implementa a lógica do cliente. Responsável por enviar as mensagens de SETUP, PLAY, PAUSE e TEARDOWN e ainda receber os pacotes de stream.

4.6 ServerLauncher.py

Módulo que implementa a lógica do servidor. Começa pelo envio de uma mensagem de HELLO ao bootstraper, obtendo o seu nodo vizinho no conteúdo do HELLO RESPONSE. Sabendo o seu vizinho, o servidor difunde uma mensagem de *probe* e inicia uma *thread* com o objeto `ServerWorker`.

4.7 ServerWorker.py

Processa mensagens de controlo da rede *overlay* e implementa o envio de pacotes RTP.

4.8 OlyPacket.py

Classe que implementa a codificação e a descodificação de pacotes de controlo da rede *overlay.*

4.9 RtpPacket.py

Classe que implementa a codificação e a descodificação de pacotes de *stream* de conteúdos multimédia - é um dos módulos usados na íntegra do código de exemplo fornecido pela equipa docente.

4.10 VideoStream.py

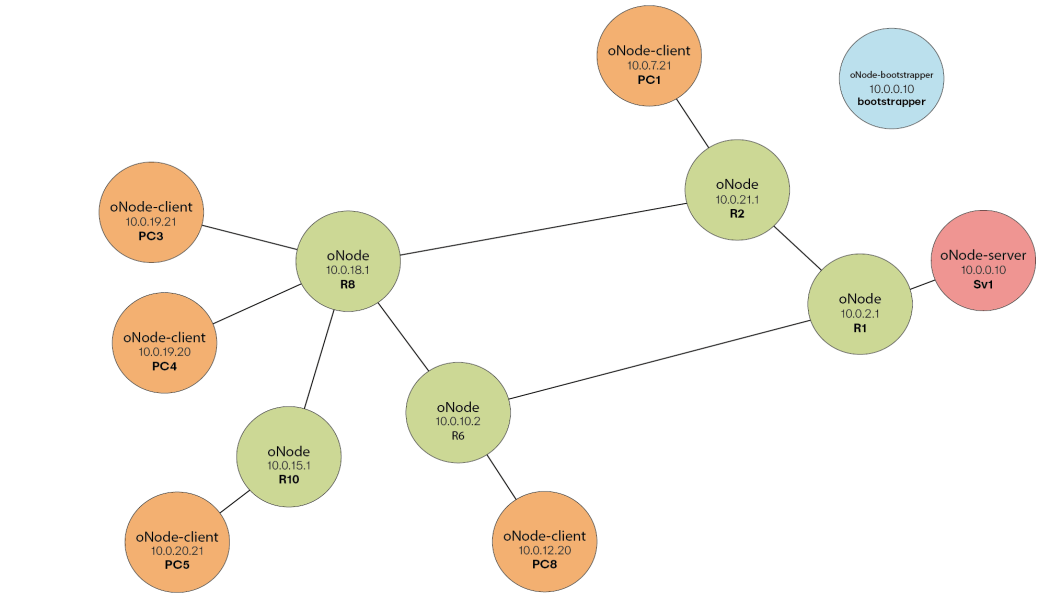
Classe que serve de suporte à classe `Serverworker` para realizar *stream* de video fazendo obtenção de frames do vídeo.

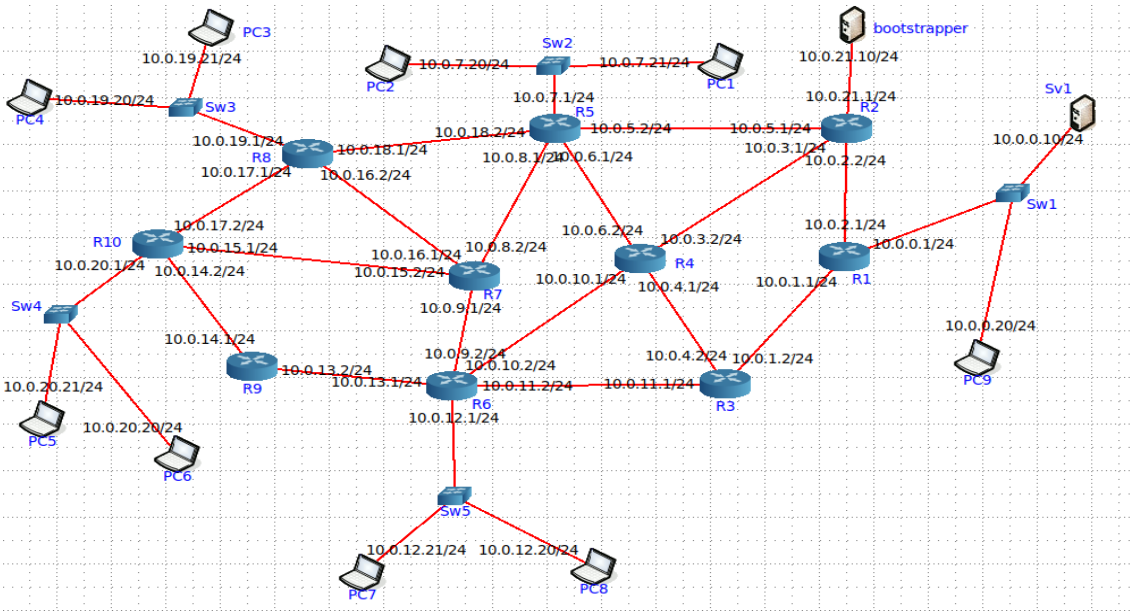
4.11 StreamsTable.py

Classe que trata da lógica da tabela dos fluxos de *streaming*.

5 Testes e resultados

Nesta secção, mostraremos como é que o nosso serviço de *streaming* se comporta num cenário da vida real. Para tal, utilizaremos o emulador CORE e criamos a seguinte topologia, surgindo, assim, uma topologia *overlay* e a topologia *underlay*, respetivamente:





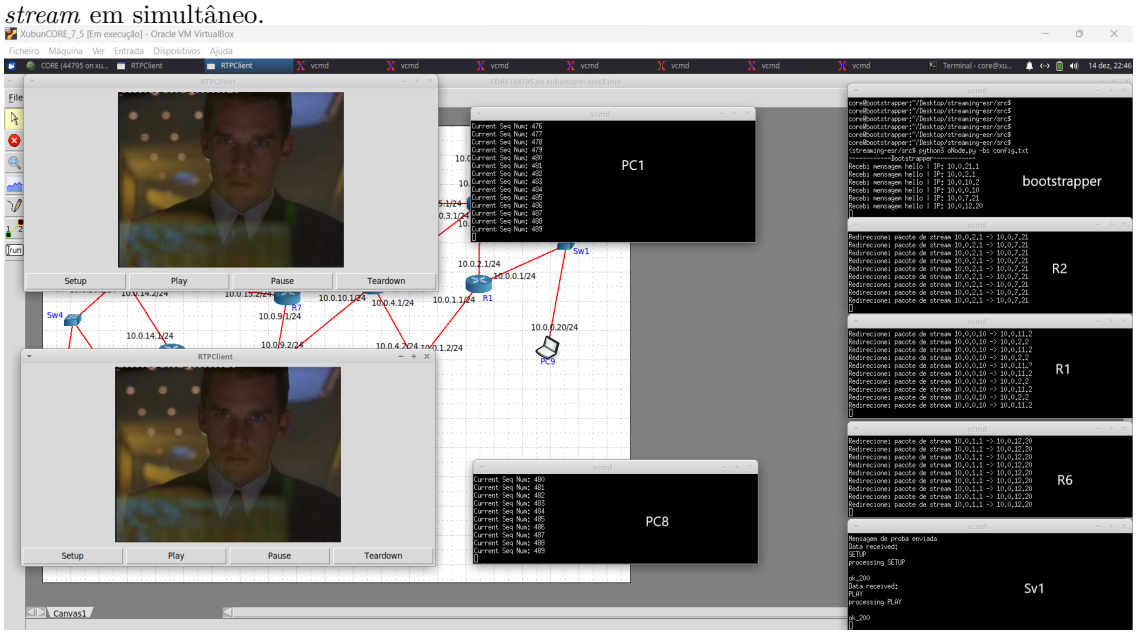
Além disto, criámos um ficheiro de texto com a configuração da rede *overlay*, onde representamos cada nó (incluindo servidor e clientes) e os seus vizinhos. Por exemplo, para o nó `R2` (com IP 10.0.21.1), escrevemos com um "#"o IP de `R2` e os seus vizinhos `R1`, `PC1` e `R8` da seguinte forma:

```
#10.0.21.1
10.0.7.21
10.0.2.1
10.0.18.1
```

De forma a podermos executar o programa, necessitamos de efetuar os seguintes passos:

1. Abrir o ficheiro `.imm` no emulador CORE com o comando: `sudo core-gui [ficheiro com a topologia]`
2. Executar a topologia
3. Executar o *bootstrapper*:
   - Abrir uma *shell*;
   - `su - core`
   - `cd [diretoria src do código]`
   - `python3 oNode.py -bs [ficheiro de configuração]`
4. Executar todos os nós a serem ativados na topologia (nós simultaneamente da tipologia *overlay* e *underlay*:
   - Abrir uma *shell*;
   - `su - core`
   - `cd [diretoria src do código]`
   - `python3 oNode.py -n [endereço IP do bootstrapper]`
5. Executar servidor:
   - Abrir uma *shell*;
   - `su - core`
   - `cd [diretoria src do código]`
   - `python3 oNode.py -s [endereço do bootstrapper] [path do vídeo]`
6. Executar cliente:
   - Abrir uma *shell*;
   - `su - core`
   - `export DISPLAY=:0.0`
   - `cd [diretoria src do código]`
   - `python3 oNode.py -c [endereço do bootstrapper]`

Assim, depois de executados os passos acima descritos, temos os clientes `PC1` e `PC8` a observar a *stream* em simultâneo.



Caso um dos clientes decida colocar em pausa, os outros clientes conseguem ver a *stream* normalmente e, caso volte a carregar no botão *Play*, esse cliente irá ver a *stream* com os pacotes que tiver a receber naquele momento. Por outras palavras, esse cliente não volta ao frame que estava a ver antes de carregar no botão *Pause*. Uma vez que é difícil para o leitor perceber estes acontecimentos com imagens, decidimos não as colocar neste relatório.

6 Conclusões e trabalho futuro

A realização deste trabalho prático ofereceu-nos a oportunidade de aprofundarmos os nossos conhecimentos relativamente ao funcionamento de um protocolo de *streaming*, assim como a comunicação entre diferentes clientes e servidores e a transmissão de dados através de pacotes RTP.

Cremos que atingimos etapas sugeridas no enunciado do trabalho prático com sucesso, embora não tenhamos tratado das etapa opcionais de tolerância a falhas dos nodos da rede *overlay* e da configuração automática para topologias distintas.

Uma das dificuldades sentidas neste projeto foi entender que este trabalho prático assemelha-se ao processo de *streaming* em plataformas como a Twitch e não como o YouTube: a nossa primeira abordagem consistia no servidor enviar os pacotes RTP para os diversos clientes e cada um assistia à sua *stream* independentemente dos outros clientes. Ou seja, um cliente colocava a sua *stream* em pausa e quando voltasse a fazer *play* assistiria a partir do frame em que estava a ver antes de colocar em *pause*. No entanto, depois de discutirmos ideias com o professor do nosso turno prático, entendemos que, em traços gerais, o servidor enviava pacotes RTP e cada cliente, ao fazer *pause* e, mais tarde, *play*, assistia ao frame que o servidor estaria a transmitir naquele momento.

Como trabalho futuro, consideramos que a implementação das etapas que não conseguimos implementar iria enriquecer consideravelmente a nossa solução do serviço *Over the Top* para entrega de multimédia.