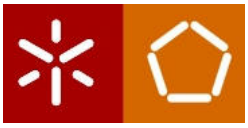

Universidade do Minho
Departamento de Informática

Engenharia de Serviços em Rede

Trabalho Prático 1

Grupo 86

Gonçalo Costa - PG55944 Lara Pereira - PG57884 Marta Rodrigues - PG55982

Índice

1. Contextualização

O consumo de multimédia em tempo real através da Internet apresenta desafios significativos para as infraestruturas de rede que utilizamos hoje em dia. Neste projeto, pretendemos implementar um protótipo de um serviço OTT (Over-The-Top) para *streaming* de vídeo, semelhante a Netflix ou Twitch.

2. Arquitetura da solução

De forma a começarmos o desenvolvimento tivemos de decidir o tipo de arquitetura a seguir entre as propostas pelo grupo docente. Devido ao curto prazo o grupo decidiu optar por uma estratégia centralizada, reconhecendo a inviabilidade da sua prática no mundo real.

Existindo um servidor central e um Bootstrapper a correr em paralelo no mesmo *host*.

- **Servidor:** Responsável por lidar com as *streams*, desde pedidos ao envio das *streams* locais
- **BootStrapper:** Gestão da topologia e das ligações, informando cada nodo da opção mais viável a qual deve pedir a *stream* e de possiveis alterações que aconteçam durante a execução do programa, ou seja, ele tem o conhecimento de toda a topologia, tornando assim uma arquitetura centralizada.

Já cada nó irá ter a responsabilidade de trocar mensagens entre si e os seus vizinhos e informar a latência da sua ligação ao *Bootstrapper*, para que o mesmo calcule os melhores caminhos possíveis.

Os clientes apenas escolhem o melhor PoP e fazem o pedido da *stream* desejada e rodam criam a interface para assisir à mesma.

2.1. Topologia

A topologia seguinte foi preparada atendendo aos requisitos indicados, nomeadamente, a presença de um servidor capaz de servir conteúdo (BootServer), a existência de três pontos de presença (n9, n10 e n11), mais de cinco pontos intermédios com redundância de caminhos e o número de clientes é superior ao número de pontos de presença.

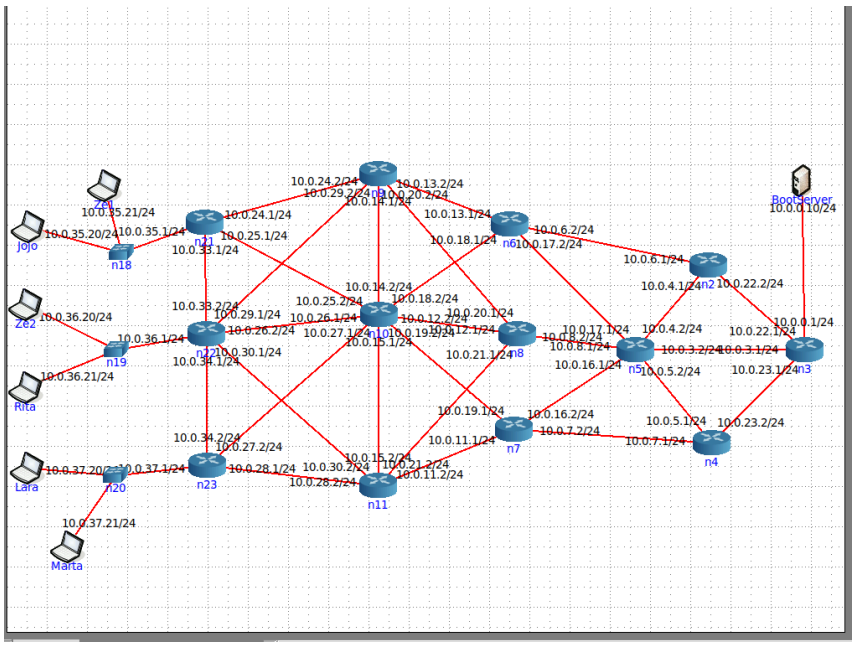

Figura 1: Topologia de rede preparada

3. Importantes considerações

3.1. Protocolo de Transporte UDP

O protocolo de transporte escolhido para a rede deste serviço de transmissão foi o UDP (User Datagram Protocol), em detrimento do TCP (Transmission Control Protocol). Comparando os dois protocolos, verificamos que o primeiro adequa-se às características valorizadas por um utilizador de numa aplicação de *streaming* de vídeo em tempo real:

- A ausência de conexões e de confirmações de entrega permite maior velocidade na transmissão dos *frames* de vídeo e diminuição da latência, principalmente em redes com múltiplos clientes, para aumentar o desempenho da aplicação e melhorar a experiência de visualização do vídeo;
- O protocolo RTP (Real-time Transport Protocol), amplamente usado em transmissão de dados multimédia em tempo real, está projetado para ser executado sobre UDP;
- Apesar de não existirem garantias na entrega dos pacotes, podemos mitigar este problema através da utilização de *timestamps* e números de sequência nos pacotes, assegurando a reprodução do vídeo de forma ordenada e suave para os clientes;
- A retransmissão de pacotes perdidos utilizada no protocolo TCP não seria adequado para esta aplicação, porque aumentaria as interrupções na reprodução do vídeo, sendo preferível para a experiência de visualização do vídeo ignorar pacotes perdidos e a ausência de *frames* no vídeo.

3.2. Mensagens

De forma a simplificar a troca de mensagens não foi desenvolvido nenhum protocolo extremamente complexo, a monitorização da latência e tolerância a faltas nas instruções mais importantes é feita enviando mensagens vazia, pois a partir da mesma já é possível calcular latência entre envio e resposta e obter os *addresses* dos *hosts* de origem. Quando é preciso a troca de informações no caso de *streams* simples elas são codificadas diretamente e estruturas mais complexas são usados dicionários compatíveis com o formato JSON. Facilitando a codificação por parte do remetente e a descodificação por parte do destinatário. Em alguns casos em que os mesmos *sockets* foram usafos para receber dois tipos de mensagens diferentes, especialmente na parte da monitorização, usou-se uma palavra identificadora para distinguir a instrução.

4. Etapa 1 – Construção da Topologia *Overlay* e *Underlay*

4.1. Configuração da rede

A camada de rede *Underlay*, que dita a infraestrutura física da rede, foi definida através de um ficheiro JSON pré-configurado. Este ficheiro está estruturado como um dicionário, onde cada chave é o endereço IP de um nó da rede e o valor associado é um objeto com as seguintes informações:

- **ID:** identificador único do nó na rede (por exemplo, n2, n3, etc.);
- **Interfaces:** lista de endereços IP que o nó utiliza para se conectar à restante rede;
- **Vizinhos:** lista de outros nós com os quais o nó está conectado diretamente, bem como, a banda larga da respetiva conexão.

A partir da leitura deste ficheiro, o serviço conseguirá obter uma representação da rede através da construção de um grafo bidirecional, onde cada nó representa um ponto físico da rede e as arestas entre eles representam as respetivas conexões físicas entre os nós. A cada aresta está associado um custo, correspondente à banda larga da ligação que simboliza.

Por outro lado, para a construção da camada *Overlay* sobre a rede *Underlay*, foi adotada uma abordagem baseada num controlador que regista os nós da rede e informa-os da sua lista de vizinhos, com base

no conhecimento que tem de toda a rede proveniente do ficheiro de configuração como explicado anteriormente.

4.2. Implementação do oNode.py

O código que implementa um nó da rede encontra-se no ficheiro `oNode.py`. O principal papel dos nós é o de intermediários da distribuição efetuada ao longo da rede. Ao executar este programa, o nó regista-se imediatamente no controlador e obtém o seu nó-pai, sendo esse obtido através do algoritmo *dijkstra* realizado pelo *BootStrapper*. A arvore de caminhos é construída, após a realização do 3 *dijkstras*, um para cada ponto de acesso, também graças ao *dijkstra* garantimos que um nó nunca poderá ter mais que um pai.

Após isso, o nó fica à escuta de requisições de vídeos até que uma seja solicitada. Quando isto acontece, o nó redireciona o fluxo desse vídeo para os nós interessados. Para lidar com cada requisição de *stream*, uma *thread* dedicada é inicializada mal o módulo inicie. Uma *thread* também é inicializada mais tarde para processar e retransmitir os pacotes de *stream*

O nó possui uma estrutura essencial ao trabalho, um dicionário cuja estrutura é a seguinte:

```
for video, port in Streams_List.items():
            self.stream_dick[video] = {
                "running": False,
                "thread": None,
                "port": port,
                "nodes_interested": set()
            }
```

Ao receber um pedido de video cada nó armazena na sua estrutura o estado desse mesmo video, se ele já o possui, a thread e a porta que está associado e os ips de nós/clientes interessados.

Os pacotes são reencaminhados para o nó que solicitou o vídeo através da porta configurada. Se o vídeo já não estiver a ser transmitido no mesmo (running == False), o nó solicita a respetiva *stream* ao nó pai e acrescenta o nó que pediu a stream aos interessados.

Além disso de forma a ter uma pequena tolerância a falhas, nas conexões mais importantes, ele antes de iniciar a instrução pedida, ele espera primeiro que o seu *host* de destino, seja um cliente ou um nó, lhe responda. Isto acontece por exemplo, no pedido da *stream*.

4.3. Implementação do Servidor e BootStrapper

O objetivo principal do Servidor é funcionar como uma central que gere a comunicação entre os nós e os clientes que solicitam o serviço de *streaming*, bem como, fornecer os vídeos solicitados através de *streams*, além de enviar as informações dos PoPs ao clientes que se conectam. As suas funcionalidades estão implementadas no ficheiro `BootServer.py`. O Bootstrapper funciona no mesmo módulo em uma thread separada.

O controlador utiliza um *socket* UDP para cada uma das seguintes tarefas: registo dos nós, registo dos clientes, gestão dos *Points Of Presence* e gestão das requisições de *stream* de vídeo. Para além disto, são criadas *threads* para a inicialização do servidor, o registo dos nós, o registo dos clientes, e a gestão das requisições de *stream* de vídeo de forma a que clientes e nós diferentes possam ser criados sem perturbar o desempenho do programa. É também criada uma *thread* por cada vídeo disponível no servidor, para que possa ser enviado o fluxo para os clientes que o solicitam, através dos devidos

nós. O envio é gerido por um objeto `ServerWorker`, associado a cada vídeo disponível, presente no ficheiro `ServerWorker.py`.

É importante realçar que quando o BootServer.py é inicializado as *streams* dos vídeos também são inicializadas de forma a simular uma *stream* como acontece no mundo real, a transmissão dos mesmo só acontece quando o Cliente fizer o pedido.

- **Registo dos nós:** O controlador fica à escuta de mensagens dos nós que se pretendem conectar à rede e regista-os, funcionando como um *bootstrapper* para os nós, recebendo dos mesmos o seu id. Assim, o controlador consegue manter a lista dos nós ativos na rede. Como resposta, o controlador envia aos nós o respetivo nó-pai, acedendo a um dicionário de nós ativos que possui com key o seu id de forma a que os look-ups sejam o mais eficiente possível.
- **Registo dos clientes:** O controlador fica à escuta de mensagens dos clientes que se pretendem assistir a um dado vídeo, recebendo do mesmo o seu Nome e a Stream desejada. Caso o cliente não tenha estabelecido contacto com controlador anteriormente, este regista-o numa lista de clientes. Como resposta, o controlador envia aos clientes a lista de Points of Presence (PoPs), para que este possa escolher a partir de onde irá consumir o seu conteúdo.
- **Gestão das requisições de *stream*:** O controlador fica à escuta de requisições de vídeos dos nós. Quando um vídeo é solicitado, o servidor verifica se este se encontra na sua lista de *streams* e, se estiver, inicia o envio do mesmo para o nó que o pediu até ao PoP escolhido pelo Cliente.

5. Etapa 2 – Cliente `oClient`

A aplicação `oClient` é o cliente final que requer ver o conteúdo de vídeo e solicita a respetiva *stream* ao Points of Presence (PoPs) escolhido (este procedimento será explicado em mais detalhe na fase complementar dinâmica).

As suas funções incluem:

- **Enviar as devidas informações ao Servidor:** Para que o Servidor possa ter a noção de que clientes existem.
- **Fazer pedido da Stream:** A um dos PoP escolhido mediante aquele que tiver menor latência.
- **Receber os pacotes RTP:** O cliente conecta-se à rede e recebe os pacotes vindos de um PoP.
- **Processar os pacotes de vídeo:** Trata da descodificação dos pacotes e do armazenamento dos *frames* em ficheiros de cache temporários
- **Exibir o vídeo na interface gráfica:** Atualiza a interface gráfica com os *frames* recebido e permite iniciar e parar a reprodução

Na sua conexão com o servidor, o cliente recebe uma lista de PoPs disponíveis para *streaming* e inicializa as suas latências a infinito. De seguida, mede a latência para cada PoP através de comandos do recurso da função `time.time()`. Esta métrica permite-lhe escolher qual o PoP com menor latência e, consequentemente, decidir com quem irá estabelecer o fluxo de transmissão.

Assim, quando o cliente pretende ver um dado vídeo, envia uma solicitação do PoP selecionado a indicar que deseja receber o *streaming* do ficheiro de vídeo. Neste momento, o cliente inicia o programa `ClientWorker`, definido no ficheiro `clientWorker.py`, para tratar da reprodução e exibição do *stream* recebido.

Quando o Cliente recebe uma *stream* ele começa a ver a mesma a partir do *frame* que se encontrava quando iniciou o pedido, não do inicio.

Além disso de forma a ter uma pequena tolerância a falhas, nas conexões mais importantes, ele antes de iniciar a instrução pedida, ele espera primeiro que o seu *host* de destino, seja um cliente ou um nó, lhe responda. Isto acontece por exemplo, no pedido da *stream*.

6. Etapa 3 – Serviço de *Streaming*

6.1. Pacotes RTP

O `RtpPacket.py` implementa a classe `RtpPacket`, usada para criar, codificar, decodificar e manipular pacotes RTP (Real-time Transport Protocol). Este ficheiro define a estrutura dos pacotes RTP e fornece métodos para processar os dados de cabeçalho e carga útil do pacote. O cabeçalho contém 12 bytes e tem a seguinte constituição:

`version`	Versão do protocolo RTP
`padding`	Indica se o pacote tem *padding*
`extension`	Indica se o pacote tem uma extensão
`cc`	Número de fontes de controlo (Csrc count)
`seqnum`	Número de sequência do pacote RTP (usado para identificar a ordem dos pacotes)
`marker`	Indica eventos específicos, como o final de uma transmissão
`pt`	Tipo de *payload*
`ssrc`	Identificador da fonte de sincronização (identifica a origem da transmissão)

A codificação do cabeçalho é feita através do método `encode`, bit a bit, utilizando operações de deslocamento e máscara:

- O primeiro byte (`header[0]`) contém os campos `version`, `padding`, `extension`, e o número de fontes de controlo;
- O segundo byte (`header[1]`) contém o `marker` e o tipo de `payload`;
- Os próximos 2 bytes (`header[2]` e `header[3]`) representam o número de sequência;
- Os próximos 4 bytes (`header[4]` a `header[7]`) representam o `timestamp`;
- Os últimos 4 bytes (`header[8]` a `header[11]`) representam o SSRC.

A carga útil do pacote vem de seguida e corresponde ao ficheiro multimédia que se pretende transmitir. A descodificação deste pacote é realizada com o método `decode`.

6.2. *Stream* de vídeo

O ficheiro `VideoStream.py` implementa a classe `VideoStream`, usada para carregar um ficheiro de vídeo e fornecer os respetivos *frames* de forma sequencial.

Após o vídeo ser carregado, é usado um objeto `cv2.VideoCapture` para abrir o ficheiro e permitir a leitura dos respetivos *frames*. Cada *frame* é codificado para o formato JPEG que é, por sua vez, convertido para bytes para poder ser enviado pela rede. Um contador global armazena o *frame* atual do vídeo.

Importante denotar que utilizamos o código fornecido pelos docentes para realizar o *streaming* por isso não vamos entrar muito em detalhes nesta secção, apesar disso apenas alteramos a forma como as conexões e pedidos eram feitos para usar UDP em vez do enorme protocolo usado pelos docentes, sendo assim mais simples e eficiente.

7. Etapa 4 – Construção das Árvores de Distribuição e Etapas Complementares

7.1. Implementação Dinâmica

Além dos papeis ditos anteriormente, iremos expor o papel de cada componente na dinamização do projeto

7.1.1. oNode

Responsável pela comunicação entre nós, capaz assim de monitorizar a latência e retransmitir as *streams*.

Sockets de monitorização são criados para cada vizinho do nó, onde há as trocas de mensagens de monitorização de 3 em 3 segundos, quando a mensagem é enviada o tempo atual da mesma é armazenado e quando recebe a resposta é calculado o novo tempo, sendo a diferença dos dois, a latência das ligações, funcionando à semelhança de um comando *ping*. De forma a lidar com nós não ativos, caso o nó não obtenha resposta, ele tenta reenviar de novo, caso mesmo assim não obtenha nenhuma resposta, o nó considera a latência dessa ligação, como infinito, simbolizando que o nó está em baixo.

À semelhança da *thread* dedicada à requisição de *streams*, *threads* como a responsável pelas mensagem de monitorização e de vizinhos são inicializadas também no inicio do módulo.

Além de ser responsável pelo fluxo de video do Servidor até ao cliente, os nós também são responsáveis por atualizar o estado da rede e permitir uma maior eficiência da *stream*.

`bootstrapper_socket`	Responsável por receber os parents do Bootstrapper
`socket_request`	Responsável pelos pedidos de stream
`socket_stream`	Responsável por redirecionar o pedido de stream para o seu parent
`socket_stream`	Responsável por receber mensagens de interrupção de stream
`timestamp_socket`	Responsável por receber as mensagens de latência do cliente
`socket_monitor`	Responsável por receber as mensagens de latência dos nodos
`socket_parent`	Responsável por receber o pai atualizado do nodo

7.1.2. Servidor e Bootstrapper

O servidor acima de tudo é responsável pelo registo dos clientes, e pela transmissão do video, usando uma *thread* para cada uma dessas funções, sendo cada transmissão de vídeo gerida por uma *thread* associado ao ServerStream, como explicado anteriormente.

Ou seja em termos de dinamismo, o servidor não desempenha grande papel.

`client_socket`	Responsável por registar os clientes e enviar os PoPs
`socket_request`	Responsável pelos pedidos de stream
`socket_stream`	Responsável por redirecionar o pedido de stream para o seu parent

Já o BootStrapper tem um papel maior, responsável por gerir toda a topologia e receber todas as informações dos nós, seja latências ou conexões novas.

`server_node_bootstrapper`	Responsável por registar os nós que se conectam e lhes enviar o pai

Paralelamente o Bootstrapper irá de 3 em 3 segundos rodar um *dijkstra* para recalcular o novo melhor caminho atual da rede e informar dos novos pais aos nós e ficar há espera de conexões de novas conexões de nós. De forma a que os nós calculem a latência o Bootstrapper também é responsável de enviar os vizinhos aos nós. Uma *thread* dedicada é criada e usada para registar os novos valores de latência no grafo. Como dito anteriormente caso algum nó não responda, ele é dado como inativo, e a sua latência será infinita.

Havendo alguma alteração na arvore a *thread* inicialmente dita também está responsável de informar os nós cujos pais foram afetados para fazerem o pedido da *stream* ao pai correto.

7.1.3. oClient

Além de receber os Pops do Servidor, o Cliente é responsável por contactar esses Pops e decidir qual o melhor a que deve pedir a *stream*, há semelhança do oNode, a monitorização da latência dos PoPs também é feita calculando o *delay* entre o tempo de envio e resposta de uma mensagem.

Numa conexão inicial o melhor PoP é calculado e usado para um primeiro pedido de *stream* e o *streaming* é portanto inicializado. Após isso o cliente monitoriza constantemente o estado dos PoPs, de 5 em 5 segundo a latência será recalculada e caso o melhor PoP mude, uma mensagem para a interrupção de stream é feita e é feito um repedido de stream, agora ao novo melhor PoP. Tanto a monitorização como a receção e descodificação da *stream* são feitas em *threads* em paralelo.

`socket_client`	Responsável por receber os PoPs do Servidor
`socket_stream`	Responsável pelos pedidos de stream
`socket_monitor`	Responsável por medir a latência com os PoPS
`socket_pop`	Responsável por interromper a stream

8. Conclusão

Fomos capazes de atingir a transmissão de um mesmo video em simultâneo em dois clientes diferentes, assim como dois vídeos em simultâneo em clientes diferentes. Assim como a adaptabilidade para o cliente optar pelo melhor PoP, mesmo ele alterando a meio e que a *stream* seja feita pelo melhor caminho no lado do CDN, aquele com latência menor, e que a mesma se mantenha quando um nó essencial vá a baixo.

Achamos que os objetivos principais do projeto foram atingidos, reconhecendo as possíveis melhorias que podem ser feitas e alguns objetivos não atingidos devidos à falta de tempo para o mesmo.

De trabalho futuro, os pontos essências a se trabalhar seria melhorar e acrescentar métricas de calculo da latência, de momento apenas uma latência daquele momento é calculada levando ao programa a estar sujeito a que devido e um pequeno *spike* na *latencia* daquela rede, a melhor possa não ser escolhida, de forma enfrentar isso, o ideal seria a partir de *pings* sucessivos, calcular a média da latência ao longo do tempo e assim ter uma melhor noção da real latência dessa ligação, assim como considerar outras métricas, por exemplo *packet loss* e outro ponto a melhorar seria um cliente poder, na mesma *tab* de aplicação, ver mais que uma *stream*. Sem mencionar de implementar uma arquitetura descentralizada levando a uma maior segurança e independência da rede, uma abordagem necessária caso a aplicação tivesse de ser lançada num contexto real.