Trabalho Prático Nº3 –

Eduardo Benjamim Lopes Coelho, Henrique Gabriel dos Santos Neto e Irenel Lopo da Silva
{pg47164, pg47238, pg42644}@alunos.uminho.pt

Universidade do Minho

Resumo Neste trabalho pretende-se conceber 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. Para tal, um conjunto de nós pode ser usado no reenvio dos dados, como intermediários, formando entre si uma rede de overlay aplicacional, cuja criação e manutenção deve estar otimizada para a missão de entregar os conteúdos de forma mais eficiente, com o menor atraso e a largura de banda necessária. A forma como o overlay aplicacional se constitui e se organiza é determinante para a qualidade de serviço que é capaz de suportar.

1 Introdução

A comunicação em grupo surgiu como um dos desenvolvimentos mais importantes na Internet. Videoconferência, distribuição de multimédia, jogos online e educação à distância são hoje alguns dos aplicativos mais populares da Internet, que geram grandes volumes de tráfego. Para oferecer suporte a esses aplicativos, a comunicação multicast confiável é um pré-requisito.

2 Arquitectura da Solução

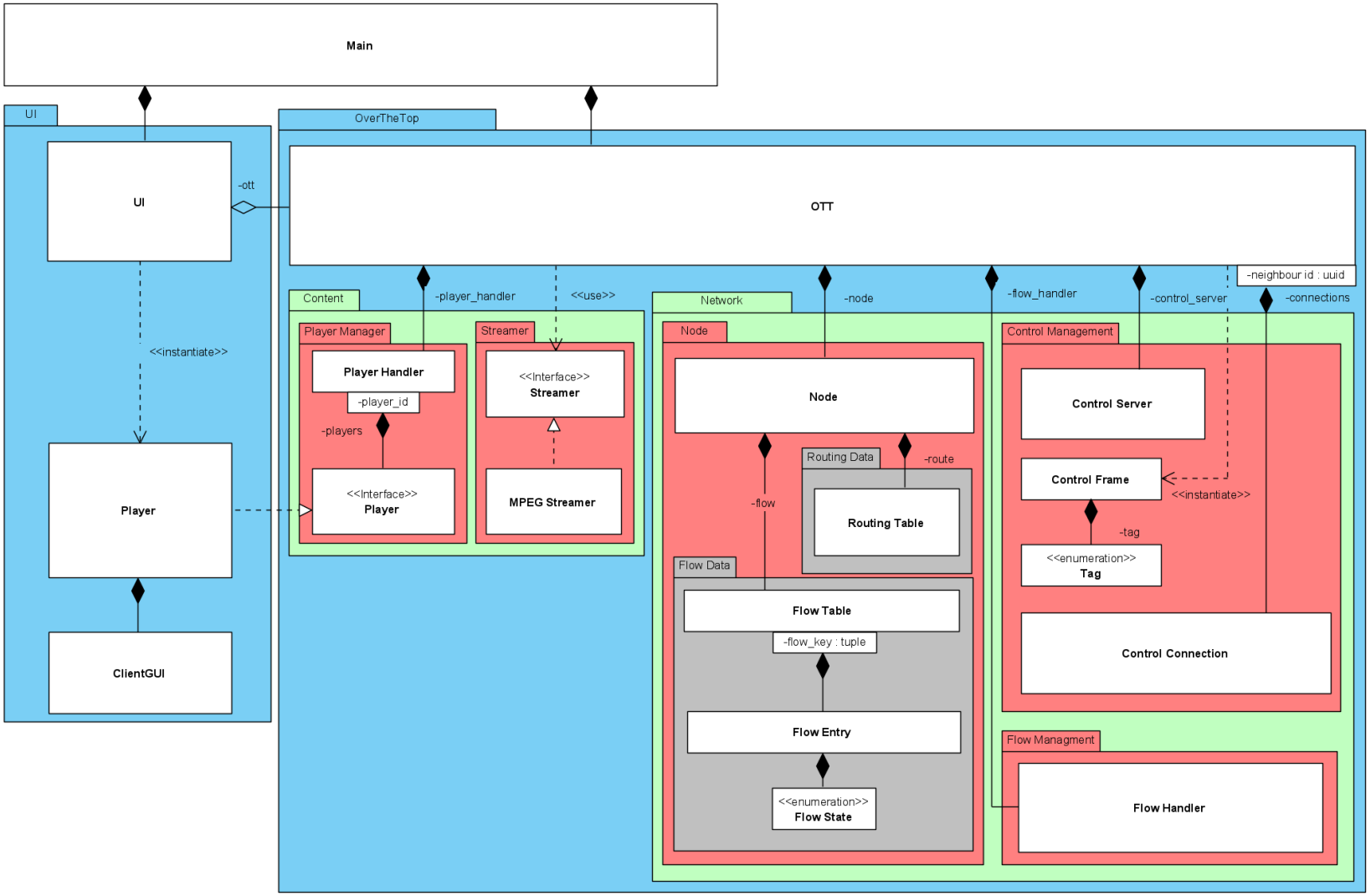


Figura 1. Diagrama de Classes

2.1 Main

Classe que inicializa o serviço. Para além de realizar o parse dos argumentos do programa, contém uma instância da classe OTT e uma instância do UI.

neighbours	[NEIGHBOURS, ...]	Vizinhos aos quais este nodo terá de se conectar (Formato: *Endereço IP* : *Porta*, Exemplo 127.0.0.1 : 8000).
-p / –port	PORT	A porta na qual este nodo funcionará (opcional, predefinição é 8000).
-a / –address	ADDRESS	O endereço a partir do qual este nodo funcionará (opcional, predefinição é a interface local).
-n / –name	NAME	O nome deste nodo (opcional).
-src / –source	[FILE, ...]	As fontes dos fluxos. Podem ser adicionados mais tarde, através do UI.
-dbg / –debug		Ativa o modo de debug, o que significa que serão vísiveis mensagens de debug no terminal.
-cli / –client		Executa o serviço num modo de cliente improvisado, que tenta se conectar apenas a um dos endereços fornecidos. As solicitações de conexão por terceiros não serão aceitas, mas o serviço ainda pode encaminhar fluxos ao se conectar manualmente a outros nós através do UI.
-nogui / –nogui		Executa a aplicação sem UI.

2.2 UI

Existem duas classes distintas para a implementação da interface gráfica do utilizador. A classe *UI* implementa a interface geral, onde o utilizador pode iniciar a transmissão de fluxos, aceder a fluxos existentes, visualizar o estado das conexões a nodos vizinhos e conectar-se / desconectar-se a outros nodos. A classe *PlayerUI* implementa a funcionalidade de visualização de fluxos. Para a implementação destas duas classes, foi utilizada a biblioteca *Tkinter* do *Python*.

UI

Tal como foi referido anteriormente, esta classe implementa uma interface gráfica que disponibiliza a maior parte das funcionalidades da aplicação. Esta interface consiste em duas secções distintas. À esquerda estão presentes todos os fluxos disponíveis na rede *overlay*. A partir daqui é possível não só, visualizar qualquer um dos fluxos (simultaneamente), como também iniciar uma nova transmissão de um ficheiro multimédia à escolha (ao clicar no botão de nova transmissão, o utilizador pode inserir o caminho para o ficheiro que quer transmitir). Para além disso, através da cor do nome do fluxo, é possível inferir o uso que este nodo lhe está a dar. Por exemplo, se este nodo estiver a visualizar o fluxo, o nome aparecerá a azul, enquanto que se estiver apenas a transmitir para outros nodos, aparece a verde. Se não estiver a fazer nenhuma destas duas funções, aparece a preto.

À direita estão presentes todos os vizinhos que estão conectados a este nodo. Caso estes vizinhos tenham um nome associado, o nome aparecerá antes do identificador. Neste caso, a cor do texto também tem significado. Caso esteja a laranja escuro, significa que esse vizinho está *unreachable* na rede *overlay* e que o sistema está de momento a tentar recuperar a conexão. Para além disso, é possível conectar-se a outros vizinhos, clicando no botão de adicionar e inserindo o endereço IP e porta do vizinho ao qual se pretende conectar. Por último, também é possível desconectar-se de qualquer vizinho, clicando no botão de eliminar a conexão.

Todas as mudanças que ocorrem na rede, tanto aos fluxos como aos nodos, são atualizadas automaticamente na interface gráfica do utilizador, o que garante que estão sempre visíveis informações reais e atuais da rede.

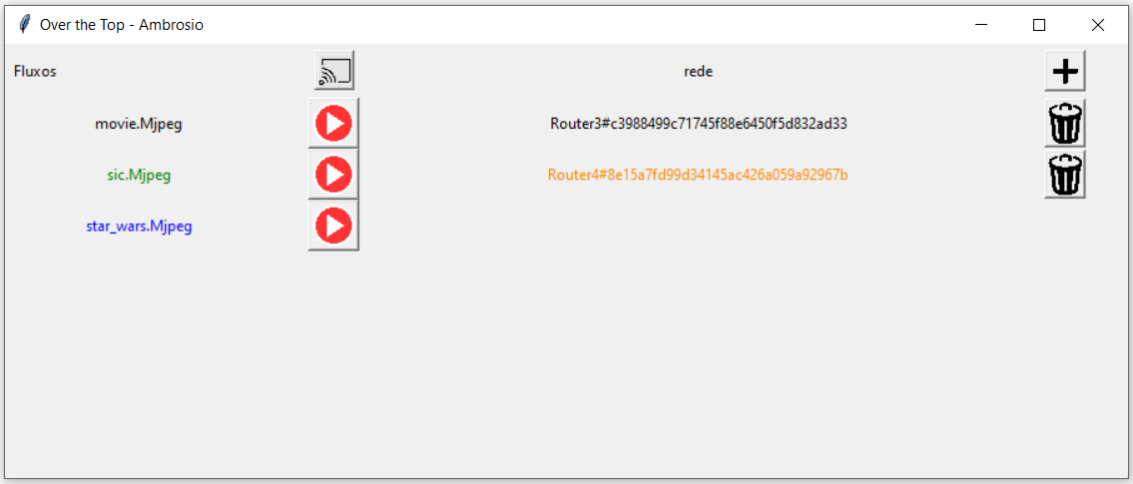


Figura 2. UI

PlayerUI Esta classe implementa o *player* de vídeo da nossa aplicação. Ao clicar no botão de *play* na interface gráfica, o *OTT* inicializa uma nova instância do *PlayerUI* para gerir a reprodução desse fluxo. Deste modo, cada nodo pode reproduzir múltiplos fluxos simultâneamente.

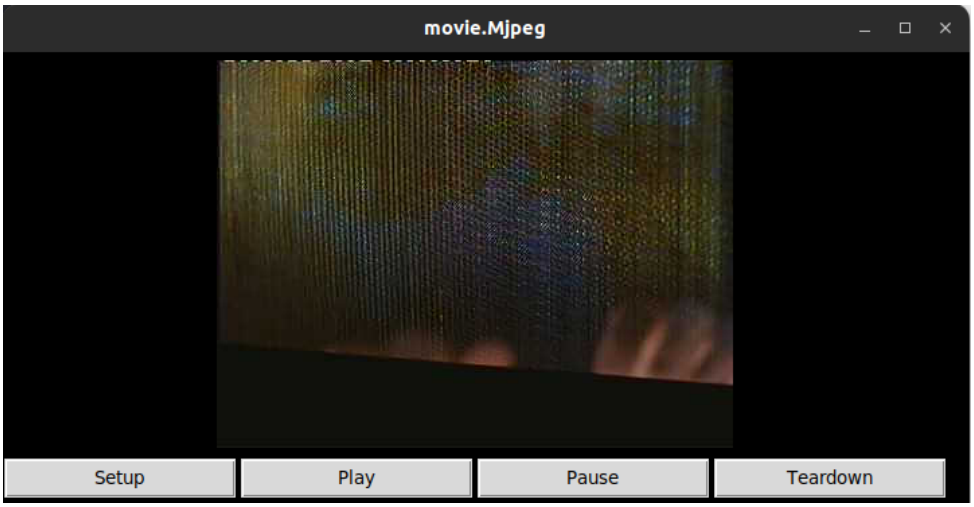
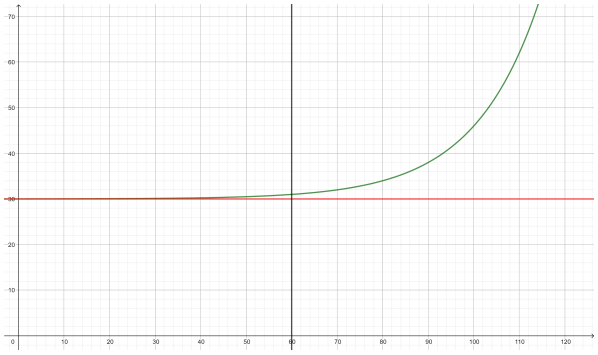


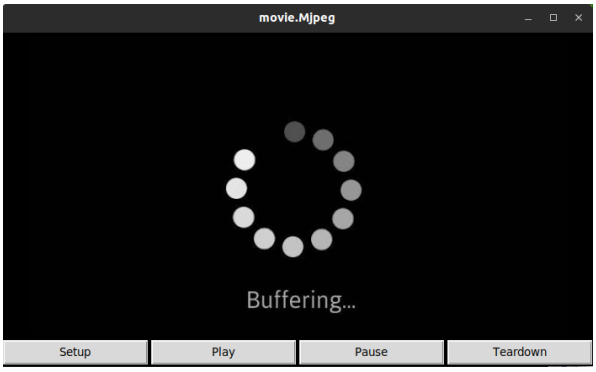
Figura 3. PlayerUI - Reprodução de vídeo

Sempre que chega uma nova trama de fluxo ao *OTT*, este chunk é inserido no *frame buffer* do *PlayerUI* correspondente a esse fluxo. Por sua vez, o *PlayerUI* remove do *frame buffer* um frame de cada vez e atualiza o ecrã. A taxa de atualização de ecrã (*framerate*) base é 30 frames por segundo. No entanto, esta taxa é dinâmica, na medida em que, adapta-se à quantidade de frames que estão no *frame buffer*. A função que representa a relação entre a quantidade de frames que estão no *buffer* e a *framerate* do player está definida, a verde, na figura 2.2. Por predefinição, o ponto em que a *framerate* começa

a aumentar é quando o tamanho do *buffer* é 60. Este detalhe permite a adaptação da reprodução do vídeo à rede e evita problemas tais como o overflow do *buffer* e de atrasos em relação ao instante atual da transmissão.



Curva



PlayerUI - Buffering

Content

Player Manager Esta classe é responsável por gerir uma lista de instâncias de classes que implementam a interface *PlayerInterface* que passaremos a referir como *players*. Cada *player* é responsável pela reprodução de um fluxo. O *Player_Handler* redireciona os *chunks* dos fluxos que chegam ao *ott* para os *players* correspondentes e permite adicionar e remover *players*. Adicionalmente também controla o estado deste conforme a disponibilidade do fluxo, sendo que se este ficar indisponível, terminará o *player* com recurso ao método *stop* que este apresenta.

Streamer Quando um nodo decide iniciar a transmissão do conteúdo multimédia de um ficheiro, é verificada qual a extensão desse ficheiro e o *ott* instância a classe que implementa a interface*Streamer* e que faça a transmissão de ficheiros dessa extensão. Atualmente, a única extensão suportada pela aplicação é a extensão *Mjpeg*, pelo que apenas existirão instâncias da classe *MPEG_Streamer*. No entanto, acreditámos que, com esta interface, a aplicação torna-se mais escalável, uma vez que é simples adicionar classes responsáveis pela transmissão de conteúdos com extensões diferentes. Após ser instanciada a classe responsável pela transmissão, pode ser chamado o método *next_chunk* que devolve o próximo *chunk* do conteúdo a ser transmitido.

Network

Node

Flow Data O ficheiro *Flow_Data* contém toda a lógica relacionada com a tabela dos fluxos. Para tal, utiliza a classe *Flow_Table* que representa a tabela e a classe *Flow_Entry* que contém a informação de uma entrada da tabela.

Em conjunto estas classes implementam uma tabela cujas chave principal consistes na composição da identificação do fluxo e da origem dele. A cada chave é associada uma entrada, que é constituída pelo estado do fluxo (definido pela classe *Flow State*) e uma lista de destinos que usam o nodo atual como rota.

Existem três estados que um fluxo pode ter:

- ACTIVE - Indica que o nodo atual está a consumir este fluxo. Durante este estado, o nodo pode adicionalmente estar a encaminhar fluxo.
- STREAMING - Indica que o nodo atual está apenas a transmitir fluxo.
- HOLD - Indica que o nodo atual não está transferir tramas relativas a este fluxo.

Para além destes três estados, encontra-se definido um valor *INVALID* que simplesmente causa o lançamento de exceções na estrutura interna do programa.

A lista de destinos contém unicamente os consumidores que usam este nó como intermediário. Tendo isto em conta, alterações das arvores de transferencias de conteúdos podem ser diretamente calculadas a partir da alterações na rede de *overlay*.

Routing Data O ficheiro *Routing Data* contém toda a lógica relacionada com a tabela de *routing*, necessária para calcular rotas de comunicação entre nodos. Para tal, utiliza a classe *Routing_Table* que implementa um algoritmo de *routing* baseado em vetores de distância, tal como o protocolo *RIP*. É uma tabela de duas entradas: nodos vizinhos e nodos destino. Cada valor da tabela representa o custo de comunicar com o nodo destino dessa coluna, a partir do nodo vizinho dessa linha.

Node Esta classe implementa a lógica de negócio interna a cada nó da rede. Desta forma a classe apresenta informação única ao nó (como a sua identificação) e implementa as interações principais entre os módulos *Routing Data* e *Flow Data*. Desta forma, operações que afetam estas duas frentes são controladas por esta classe de forma a que não ocorram erros nem que surgem estados errados como rotas inválidas, ou fluxos degradados.

Control Managment

Control Server A classe *Control_Server* é instanciada numa *thread* do *ott* e é responsável pelo estabelecimento de conexões com outros nodos. Sempre que um nodo se tenta conectar, o *Control_Server* cria uma nova instância da classe *Control_Connection* que é responsável pela gestão da comunicação *TCP* que ocorre entre os dois nodos.

Control Connection A classe *Control_Connection* é responsável pela comunicação *TCP* entre dois nodos. Todas as mensagens de controlo (mensagens que não contêm *chunks* de conteúdo multimédia) são controladas por esta classe, utilizando o protocolo *TCP*. Embora o sistema *OTT* esteja unicamente preparado para interagir com mensagens do tipo *Control_Frame*, esta classe permite a transferência de qualquer estrutura de dados pelas *data streams* do protocolo *TCP*.

Control Frame Esta classe é responsável pela criação de mensagens de controlo para serem enviada por *TCP* para nodos vizinhos, através de instâncias de *Control_Connection*. Estas mensagens são constituídas por uma *tag*, que especifica o tipo de mensagem, e a *data*, que contém o conteúdo da mensagem.

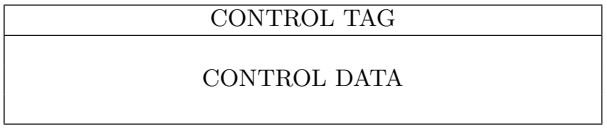
Flow Management Neste ficheiro foi implementada a classe *Flow_Handler*. Esta classe funciona como o intermediário entre o programa e a camada *UDP*. Todas as tramas recebidas pelos *dispatchers* são colocadas num *buffer* à espera de serem processadas pelo programa. Esta abordagem evita perdas de dados quando existem picos de informação, isto é, quando chegam demasiados dados em pouquíssimo tempo. Para além dos *dispatchers*, existem 1 ou mais *forwarders* que enviam as tramas que são colocadas no *buffer* de *output* para os vizinhos. Esta implementação tem a vantagem de tornar o tratamento das tramas que precisam de ser enviadas a vizinhos independente do *socket* de comunicação. Para além disso, torna a aplicação mais escalável, na medida em que facilita a adição de funcionalidades tais como o controlo de congestionamento.

OTT A classe *OTT* é a fachada principal do serviço desenvolvido. Assim, esta classe fornece uma interface para a lógica de negócio à interface gráfica. Para além disso, o *OTT* pode ser visto como uma ponte entre as interfaces externas, tais como as conexõesm a interface de utilizador e o *node* em si, através de *handlers* e métodos que processam quaisquer mudanças que ocorrem na rede. Na verdade, esta classe interpreta todas as mensagens de controlo que chegam a este nodo, realizando todas as ações relevantes a essas mensagens e gerando mensagens de resposta. Para além disso, o *OTT* também responde às chamadas da interface gráfica do utilizador. Como esta classe é responsável por tantas funcionalidades diferentes, é necessário haver gestão de várias *threads* que funcionam como trabalhadores com funções específicas. Por exemplo, o trabalhador *flow processor* é responsável pelo *forwarding* de tramas de fluxo. São também instanciadas sobre as classes *Flow_Handler* e *Control_Connection* os trabalhadores responsáveis pelas funcionalidades explicadas anteriormente. Já para a funcionalidade de *streaming* é inicializado o trabalhador *flow_streamer* que instancia a classe *Streamer*. Por último, para gerir as conexões com os vizinhos, temos o trabalhador *Doctor* que é responsável por situações como reconexão de vizinhos e o trabalhador *connection_worker* que gere as conexões aos nodos vizinhos.

3 Especificação do(s) protocolo(s)

3.1 Mensagens de controlo

As mensagens de controlo são enviadas em conexões TCP e são constituídas por uma *Tag* correspondente ao tipo e por uma estrutura de dados especifica ao tipo a que pertence.



Distance Vector Array

Esta mensagem, que possui a *Tag DISTANCE_VECTOR* é transmitida entre dois nós vizinhos e contém os destinos acessíveis por um nó, associados ao custo da rota que ele oferece. Este vetor nunca contem rotas provenientes do seu destinatário de forma a não provocar divergências no algoritmo de *routing*. A partir disto, cada nó constrói uma tabela de *routing*, baseada em todas as tramas dos seus vizinhos, que associa cada nodo da rede e cada vizinho ao custo da rota que estes estabelecem. Por fim, a tabela é condensada num vetor global de melhores rotas que é usado pelos algoritmos de *fowarding*. Adicionalmente, se este vetor apresentar mudanças à topologia definida anteriormente, este é filtrado e transmitido a cada vizinho que ainda se encontrem desatualizados. Este processo é extremamente semelhante ao protocolo *Routing Information Protocol* (*RIP*) que opera da mesma forma, porém relativo ao *routing* das rede *IP*.

Adicionalmente esta trama também pode provocar atualização nas árvores de transferência de fluxo. Quando é calculado um vector distancia global, o algoritmo desenvolvido está programado para caracterizar as mudanças realizada a cada nó alterado conforme três categorias.

A primeira categoria são as mudanças leves (*light*), e são constituídas por novas rotas na rede ou por mudanças benéficas ou intransigentes às árvores de melhores rotas. Neste caso as tabelas de fluxo mantêm-se intactas.

A segunda categoria reflecte as mudanças pesadas (*heavy)*, e são constituídas pelas mudanças que pioram o acesso a um dado conjunto de nodos. Nestes casos, devido a não podermos assumir que a rede convergiu, os registos de consumo dos nós são limpos da tabela de fluxos, sendo necessário enviar novos pedidos pelos consumidores.

A terceira categoria corresponde às mudanças críticas (*critical*), e são constituídas pelas rotas que foram perdidas. Como o nome indica, estas mudanças são delicadas e implicam, não só a remoção

dos consumidores do nó associado, mas também a remoção de todas as entradas de fluxo que este disponibilizava à rede de conteúdo.

Nestas duas últimas categorias, é da responsabilidade dos nós consumidores realizar a recuperação (de forma automática) dos fluxos que perderam, conforme o resultado destas mudanças. Se ainda lhes for possível aceder ao conteúdo, estes procedem à sua recuperação através de novos pedidos, o que pode ser transparente ao utilizador devido ao *buffer* que o *player* contém. Caso o acesso seja perdido por completo ao fluxo, o nó fecha o *player* aberto para este e elimina os registos deste.

Anúncio/Colecção de Fluxos

Estas mensagens, designadas pelas *Tags FLOW_COLLECTION, FLOW_ANNOUNCE*, tem o propósito de comunicar aos nós a lista de fluxos que estes podem requisitar da rede. A propagação acontece de forma epidemémica, e começa com a realização de novas conexões ou quando um nó cede um fluxo à distribuição. Por fim, ao receber uma destas tramas de um vizinho, o nó regista-a na sua tabela de fluxo e, se este possuir entradas de fluxo novas, o nó prosseguirá a transmitir novas tramas de anúncio/colecção contendo apenas estes fluxos novos que descobriu.

Remoção de Fluxo

Esta mensagem, caracterizada pela *Tag FLOW_WITHDRAW*, apresenta o comportamento oposto às tramas de *Anúncio/Colecção*, ou seja, indica a remoção de um dado conteúdo por parte de um cliente. Esta trama é apenas transmitida quando o fornecedor do conteúdo indica, deliberadamente, que não disponibiliza mais o fluxo, não sendo necessária em outras situações de remoção, como perda de conexão com o nó fornecedor. Semelhante às duas tramas anteriores, a propagação desta trama acontece de forma epidémica, e só se espalha para os vizinho se o nó atual possuir o fluxo descrito por esta trama.

Pedido de Fluxo

Os pedidos de transmissão, designados por tramas *FLOW_REQUEST*, são mensagens ponto a ponto que indicam que um dado consumidor deseja aceder ao fluxo de um dado fornecedor. Esta mensagem é gerada em duas situações, quando o utilizador pede acesso ao fluxo pela interface do *OTT*, ou quando a serviço está a tentar recuperar uma rota de fluxo alterada.

Cancelamento de Fluxo

Estas mensagens, designadas pelas tramas *FLOW_CANCEL*, possuem um comportamento oposto à mensagem anterior, ou seja, indica que o consumidor especificado já não pretende aceder ao fluxo explícito na mensagem. Esta mensagem é gerada pelo próprio consumidor, podendo ser gerada pelo próprio utilizador (ao fechar o *player* do fluxo) ou pela gestão de fluxos interna à lógica de negócio.

Autenticação

Dado às características base do *OTT*, e com o objetivo de possibilitar várias instâncias do mesmo serviço na mesma máquina, não é possível identificar os nodos apenas pelo seu endereço ou porta. Desta forma, durante a abertura de uma conexão existe um processo de autenticação que envolve a troca de duas mensagens de autenticação, designadas pela *Tag AUTHENTICATION*. Estas tramas são constituídas pelo identificador único universal do nodo (uuid), e pela interface geral dos servidores de fluxo e de controlo a que estão associados. Adicionalmente, também é comunicada informação opcional ao serviço, como o nome que o vizinho tem.

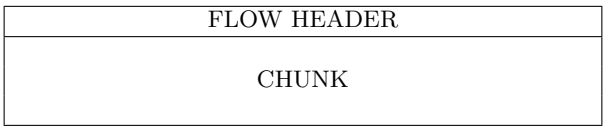
Autenticação Necessária

Esta mensagem, descrita pela *Tag AUTHENTICATION_REQUIRED* é enviada pelo serviço, enquanto este não lhe comunicar a sua informação única, através de uma mensagem de autenticação.

Estas mensagens são enviadas cada vez que o serviço recebe uma mensagem antes do processo de autenticação estar completo. Após um certo número de tentativas (por predefinição são 3), o serviço fecha comunicações com o par envolvido.

3.2 Formato dos chunks dos fluxos

O serviço implementado abstrai as mensagens de fluxo. Na verdade, uma mensagem de fluxo consiste no encapsulamento de um *chunk* especificado pelo *streamer* com um cabeçalho de transporte de fluxo. Este cabeçalho possui a chave do fluxo e os destinos a que se destina. Ao receber uma mensagem de fluxo, o serviço divide o *chunk* pelos seus vizinhos, conforme os *gateways* dos destinos especificados no cabeçalho. Por sua vez, as mensagens resultantes possuem cabeçalhos atualizados com apenas os destinos relevantes ao *gateway* a que se destina. Caso o nodo atual esteja presente nos destinos, o chunk é adicionalmente encaminhado para o gestor de *players* local.



As mensagens são enviadas com recurso ao protocolo de transporte *UDP*. Não foi implementado nenhum mecanismo de confirmação que a mensagem chegou ao devido destino intacta, uma vez que, a nosso ver, esta viabilidade pode ser sacrificada para permitir uma rápida transferência de conteúdos entre os nós e consequentemente uma melhor qualidade de serviço, visto que, neste caso de *streaming*, é irrelevante que exista controlo e retransmissão de mensagens de fluxo já que o atraso provocado por estes processos tornaria a trama irrelevante.

Chunk do Mjpeg Streamer

Na implementação atual do *Mjpeg Streamer*, um *chunk* corresponde a uma frame. Desta forma a secção de *chunk* será o numero da *frame* seguido da *frame* em si.

4 Implementação

Logging Para gerar informação coerente sobre o estado do programa na consola/ficheiros, foi usada um modulo presente nas bibliotecas padrão do python chamado *logging*. Este modulo permite configurar um ou mais *loggers* que seguem um formato especificado na sua criação e categoriza as mensagens em 5 niveis (Critical, Error, Warning, Info, Debug). Por fim para escrever uma mensagem no *logger* basta chamar o método correspondente ao nível desejado e fornecer a mensagem como argumento. O modulo possibilita adicionar novo níveis e interação com mecanismos *in-built* do python, como a *trace stack*.

ArgParse Esta classe está presente nas bibliotecas padrão do *python* e possibilita uma fácil especificação, gestão e consulta dos argumentos de consola.

ThreadPoolExecutor Esta classe faz parte da biblioteca *concurrent.futures*, fornecida nas instalações padrão do python e adiciona funcionalidades de Thread Pools ao mecanismos de concorrência já existentes no python.

Tkinter Para implementar a interface gráfica da aplicação, decidimos utilizar a biblioteca *Tkinter* do Python. Esta decisão deve-se ao facto de o código disponibilizado pelos professores fazer uso desta mesma biblioteca. Deste modo, evitamos problemas de compatibilidade entre a janela de reprodução de conteúdo e a janela principal da aplicação.

5 Testes e resultados

5.1 Pré-requisitos

Em debian (ubuntu, xubuntu ...):

```
sudo apt-get install python3-pil python3-pil.imagetk
```

De modo a testar a nossa implementação, criámos 3 cenários distintos:

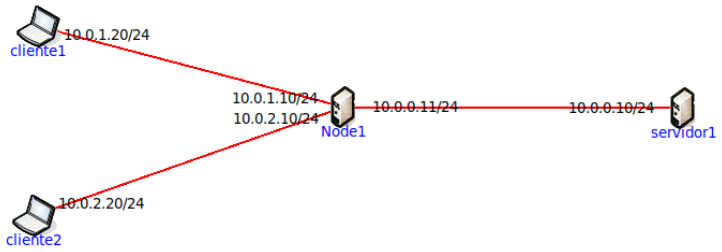


Figura 4. Cenário 1

Nesta topologia estão presentes dois clientes (esquerda), um servidor (direita) e um nodo intermédio que só está a redirecionar fluxos. Iniciando uma transmissão de um fluxo *Movie.mjpeg* no *servidor1* e inicializando a reprodução do mesmo fluxo nos clientes *cliente1* e *cliente2* verificámos que o fluxo foi replicado no *node1* com sucesso e reproduzido nos dois clientes sem problemas. Parando de assistir nos dois clientes, o *node1* deixou de redirecionar fluxos uma vez que o próprio *servidor1* deixou de transmitir conteúdo visto que não tem nenhum cliente a assistir ao seu fluxo.

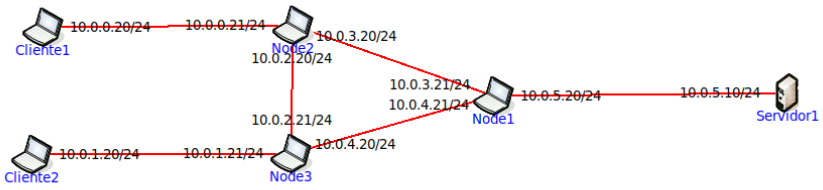
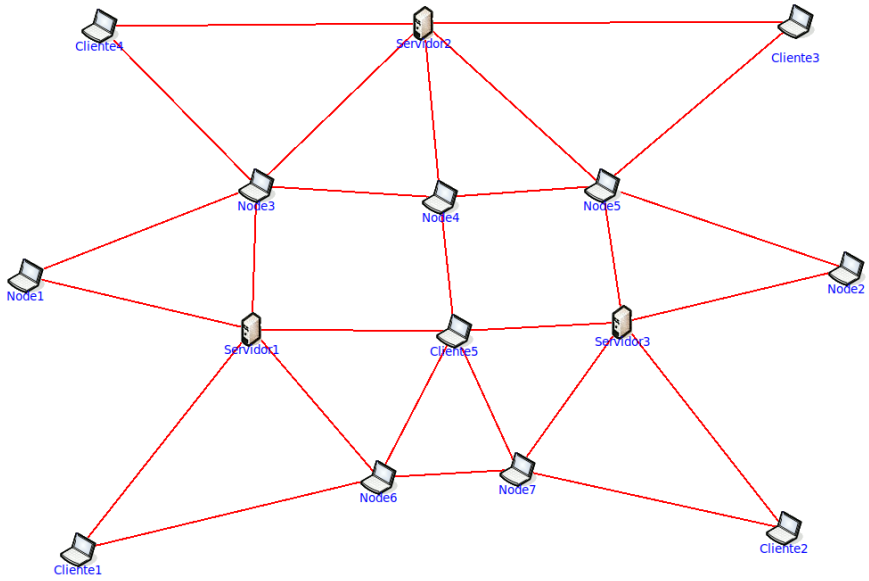


Figura 5. Cenário 2

Nesta topologia, temos 2 clientes (esquerda), 1 servidor (direita) e 3 nodos intermédios que só estão a redireccionar fluxos. Colocando o *servidor1* a disponibilizar um fluxo, se assistirmos a esse mesmo

fluxo no *cliente1* e no *cliente2* podemos verificar que a rota de entrega do conteúdo é a seguinte: o conteúdo é enviado para o *node1* onde é redirecionado para o *node2*, onde é replicado para o *3* e *Cliente1*, sendo redirecionado finalmente a partir do *node3* para o *Cliente2*.



55

Figura 6. Cenário 3

Este cenário, mais complexo que os anteriores tem os seguintes objetivos: testar a adaptação das rotas de entrega de fluxos quando ocorrem mudanças na rede *overlay*, testar a transmissão de múltiplos fluxos e testar a transmissão de um só fluxo mas com várias origens diferentes.

Transmissão de um fluxo com origens diferentes - Ao se disponibilizar um fluxo a partir do *servidor2* e *servidor3*, se inicializarmos a reprodução desse mesmo fluxo no *cliente2* podemos verificar que é o *servidor3* que está a enviar o conteúdo. Ora, como podemos observar, esta é a escolha correta, uma vez que o *servidor3* encontra-se a um salto do *cliente2*, enquanto que o *servidor2* está a 3 saltos.

Transmissão de múltiplos fluxos - Se inicializarmos uma transmissão de um fluxo distintos em cada um dos servidores, verificámos que podemos inicializar a reprodução dos mesmos em cada um dos clientes sem qualquer problema e, até mesmo, a reprodução dos 3 fluxos em simultâneo num só cliente. Se inicializarmos a transmissão dos 3 fluxos distintos num só servidor, por exemplo no *servidor2*, verificámos que podemos assitir aos 3 fluxos em qualquer um dos clientes sem qualquer problema.

Adaptação das rotas de entrega de fluxos a mudanças na rede - Se inicializarmos uma transmissão de um fluxo no *servidor2* e tentarmos assistir o conteúdo deste fluxo no *cliente5* verificamos que a rota de entrega é a seguinte: o *servidor2* envia o fluxo para o *node4* e este redireciona-o para o *cliente5*. No entanto, se o *node4* ficar sem conectividade ao resto da rede *overlay*, verificámos que a rota de entrega de fluxo se adapta para o seguinte: o *servidor2* envia o conteúdo para o *node5* que, por sua vez redireciona-o para o *servidor3* que, por último, redireciona o conteúdo para o *cliente5*. Este é o comportamento pretendido já que, a rota de entrega de fluxo foi capaz de se adaptar a uma falha na rede.

6 Conclusões e trabalho futuro

Este projeto permitiu a consolidação dos conceitos lecionados na unidade curricular *Engenharia de Serviços em Rede*, através da implementação de um serviço de *streaming* de conteúdos multimédia, com base numa rede de overlay aplicacional. Um dos maiores desafios deste trabalho foi a criação de um algoritmo de criação e gestão de rotas dos fluxos, que implicou a implementação de diversas mensagens de controlo que teriam de ser enviadas na rede. No entanto, acreditamos que conseguimos chegar a uma solução que permite entregar os conteúdos multimédia de forma eficiente, com poucos atrasos e baixo uso de largura de banda. Uma melhoria que poderá ser feita à aplicação no futuro é tornar a construção da topologia overlay automática, na medida em que não seja necessário conectar os nodos de forma manual. Outra grande dificuldade que ultrapassámos neste trabalho foi o controlo de erros na rede overlay. Por exemplo, permitir que a rede se adapte a desconexões e conexões de nodos, de modo à entrega de fluxos ser dinâmica e adaptar-se a quaisquer mudanças que ocorram. Em suma, para além de termos sido capazes de cumprir todos os objetivos deste projeto, ainda implementámos algumas funcionalidades extra tais como a transmissão de múltiplos fluxos na rede e a reprodução de múltimos conteúdos num só nodo.