TP2 - Serviço *Over-the-Top* para Entrega de Multimédia

Rúben Silva , Pedro Martins e Tomás Campinho

Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal
Engenharia de Serviços em Rede

{pg57900, pg57894, pg57742}@alunos.uminho.pt

1. Introdução

A evolução da internet trouxe um aumento significativo no consumo de conteúdos digitais em tempo real, exigindo infraestruturas capazes de lidar com desafios como congestionamento e atrasos. Serviços de streaming em tempo real como *Youtube Live* e *Twitch*, que operam em redes ***Over-the-Top (OTT)***, dependem de soluções eficientes para garantir a qualidade de entrega. Este relatório descreve as várias etapas de desenvolvimento de um protótipo de rede overlay aplicacional para otimizar a entrega de conteúdos através de uma infraestrutura baseada em ***Content Delivery Networks (CDNs)***.

2. Arquitetura da Solução

Neste trabalho, pretende-se criar um sistema de entrega multimédia em tempo real, partindo de um ou mais servidores de streaming para um conjunto de clientes. Os servidores são o cérebro de toda a operação, sendo responsáveis por definir os caminhos a percorrer e o funcionamento do sistema. Estes recebem pedidos de conexão e determinam a melhor rota para garantir que os pacotes de *streaming* cheguem ao destino. Os nós intermédios são aplicações com **capacidade *full duplex***, encarregues de encaminhar os dados para os destinos necessários, considerando apenas as interfaces dos vizinhos adjacentes. Por fim, os clientes são **aplicações de linha de comando (*CLI*)** com capacidade para visualizar vídeos e monitorizar todo o tráfego gerado. Tudo isto é integrado numa topologia mais complexa, de forma a demonstrar as capacidades do projeto em questão. Para a implementação de isto tudo foi **utilizada a linguagem de programação *Python*** que oferece um bom nível de abstração e dispõe de uma vasta gama de bibliotecas que facilitam a manipulação de tudo o que for necessário. Dentro das bibliotecas, consta ***OpenCV* para o encoding e leitura de *packets* e *sockets***, entre outras mais. Além disso, esta é a linguagem com a qual os membros do grupo se sentem mais familiarizados e confortáveis a usar. Na seguinte imagem temos a nossa topologia ***overlay*** e ***underlay***:

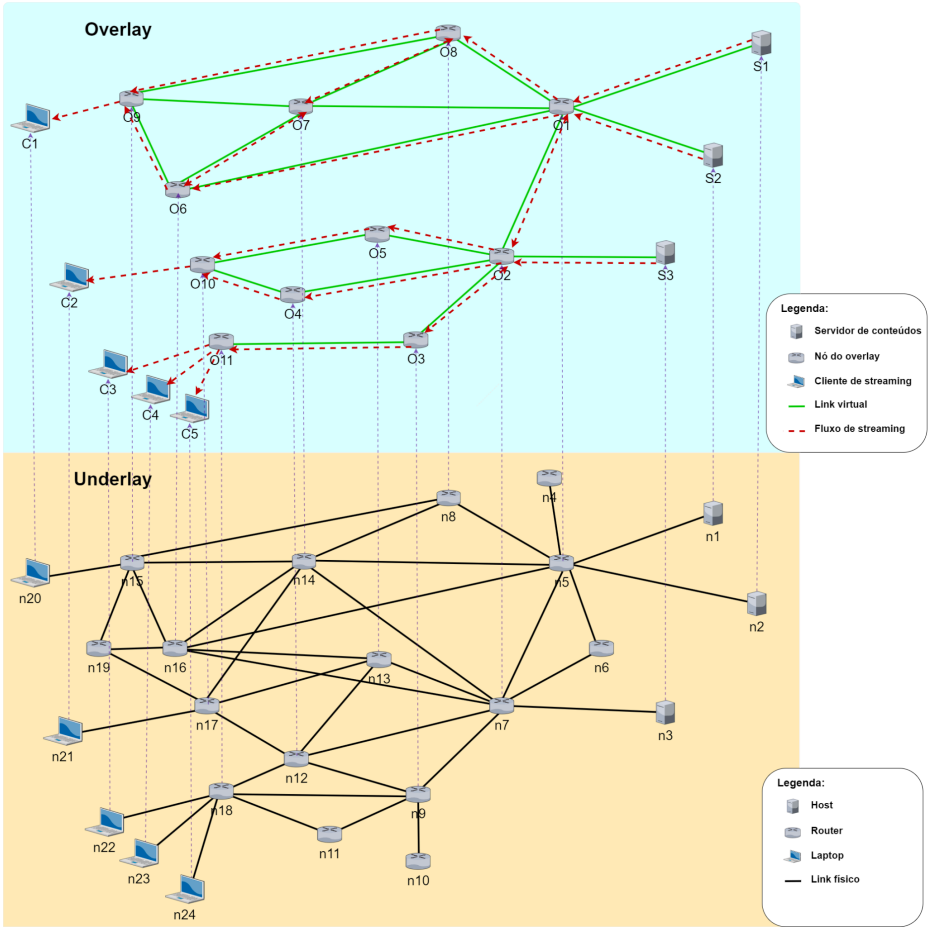

Figura 1. Arquitetura do sistema

A estratégia definida para a construção da rede *overlay* baseia-se na utilização de um **ficheiro de configuração em formato *JSON*** que contém a informação necessária para cada componente, ***Points of Presence* e os vizinhos de cada nodo**. As secções seguintes detalham os argumentos exigidos por cada componente, bem como a estrutura esperada dos ficheiros de configuração.

2.1. Nodos e Servidores

Para um **nodo** saber onde estão os seus vizinhos, ao iniciar envia uma mensagem a um servidor ligado especificado por argumentos: ***python3 oNode <IPServidor>***

O servidor pode ser qualquer um, visto que todos conhecem a topologia, tem é de estar ligado para conseguir comunicar com os vizinhos.

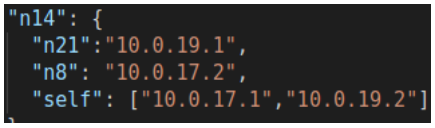

Figura 2. Exemplo da configuração de um nodo

Na figura anterior conseguimos visualizar o que o **nodo** irá receber. Sabemos que as interfaces dos **nodos adjacentes** são os que se encontram no **n21** e no **n8**. O ***self*** é um parâmetro para o servidor reconhecer o nodo quando este pede informações sobre os seus vizinhos.

2.2. Clientes

Para um **cliente** saber quais os ***PoPs* (*Points of Presence)*** possíveis também receberá uma estrutura idêntica. Tendo este de ser iniciado com: ***python3 oClient <ipServidor>***. A lógica de implementação é análoga. Na figura seguinte temos um excerto da informação recebida:

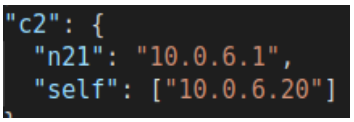

Figura 3. Exemplo da configuração de um cliente

3. Especificação de Protocolos

O protocolo aplicacional escolhido para o streaming foi o ***UDP (User Datagram Protocol)***, utilizando uma implementação nova, feita de raíz sem nenhum esqueleto.

Para o controlo da perda de pacotes, a nossa rede tem a capacidade de meter uma chave no protocolo que obrigatoriamente tem de a receber, garantindo assim que a resposta chegou, evitando a perda de dados importantes.

A transmissão dos conteúdos para os vários clientes, através dos diferentes componentes da nossa infraestrutura, exigiu o **desenvolvimento de um protocolo aplicacional de controlo** que permitisse gerir o envio de conteúdos pela rede. Tendo em conta a natureza do projeto, considerámos que a opção mais adequada seria implementar este protocolo de controlo utilizando o **protocolo de transporte *UDP***.

Neste contexto, o protocolo base que suporta a **funcionalidade principal do nosso serviço** (transmissão de conteúdos) é composto por **três etapas principais**:

1. Analisar a rede atual e decidir as rotas do conteúdo.
2. Realizar a transmissão do conteúdo.
3. Finalizar a transmissão do conteúdo.

Além do protocolo responsável pela transmissão de conteúdos, o nosso serviço inclui outros protocolos destinados à monitorização dos servidores de conteúdos, gestão de erros e falhas, bem como à administração de alterações na topologia da rede, abrangendo a entrada e saída de nós.

3.1. Formato das mensagens protocolares

De uma forma abstrata, o nosso projeto funciona dentro de um paradigma bem definido:

<Tipo>:<id> <Comando> <Mensagem>

Tipo : Cliente, Servidor ou *Node*, podendo ter ou não um *id* à frente, isto é para mensagens com sensibilidade a perda de pacotes, caso exista *id*. Esse mesmo tem de receber uma resposta com o respectivo *id* garantindo que a resposta acontece.

Comando :

- *Request_PoP* -> Cliente pede os *PoPs* respectivos ao servidor;
- *Response_PoP* -> Servidor responde com os *PoPs*;
- *PING* -> Cliente envia *PING* aos *PoPs*;
- *PONG* -> Cada *PoP* responde com *PONG* e o tempo atual para calcular o *rtt*;
- *Request_Videos* -> Cliente pede a lista de vídeos disponíveis;
- *Response_Videos* -> Servidor envía a lista de vídeos;
- *Request_Stream* -> Cliente pede para assistir uma das *streams* disponíveis;
- *Response_Stream* -> Servidor concorda em enviar a *stream* e dá-lhe a *frame-rate* do vídeo para o leitor do cliente;
- *StreamLogOff* -> Cliente desconecta-se da *stream*;
- *Request_Vizinho* -> *Node* pede os vizinhos ao Servidor;
- *Response_Vizinho* -> Servidor entrega os vizinhos;
- *Pathfinder* -> Comando associado à inundação controlada. (Explicação mais detalhada ao longo deste relatório);

Mensagem : Em formato de *string*, onde podem incluir novos comandos para processamento posterior.

Em caso de mensagens com sensibilidade a resposta, temos um exemplo de *request* e *response*:

Ex.Request : Cliente:17 Request_PoP cliente

Ex.Response: Servidor Response_PoP 10.0.2.2
Servidor Response_PoP 10.0.3.2
Servidor Response_PoP done 17

A última mensagem é o que dirá ao cliente que foi tudo entregue e mostra-lhe o *id* do *request*.

3.2. Interações

Nesta Secção constam exemplos de interações entre as diversas aplicações do sistema e a sua funcionalidade das quais achamos importantes:

3.2.1. Cálculo entre Cliente e *PoP* (*PING/PONG*)

Este é um exemplo simples de comunicação ponto a ponto, com uma mensagem enviada e uma resposta recebida. Existem várias funções deste tipo, como *Request_PoP/Response_PoP* e *Request_Vizinho/Response_Vizinho*, entre outras. Estas funções permitem a troca de dados de forma simples e são utilizadas de forma esporádica. Neste caso específico, o cliente mede o *RTT* (*Round-Trip Time*) entre os *PoPs* que lhe fornecem acesso à rede.

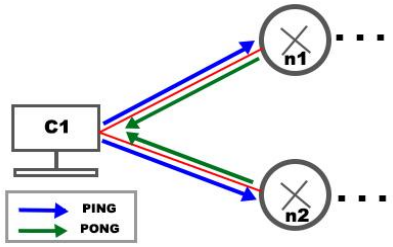

Figura 4. Cálculo entre Client e PoP

3.2.2. Transmissão de Vídeo

Este é um exemplo de como a maioria das comunicações são feitas na nossa implementação. Na figura abaixo ilustramos que já existe um *path* bem definido por parte do servidor e que é o menos custoso, visto que o caminho de cima possui mais Latência. Este tipo de comunicação serve para monitorar toda a rede para saber os melhores caminhos e entregar conteúdos pela topologia.
Neste caso, o cliente previamente já tinha escolhido um vídeo, e o servidor já tinha feito uma procura da melhor rota. Após isto os pacotes são entregues seguindo o *path* definido pelo servidor.

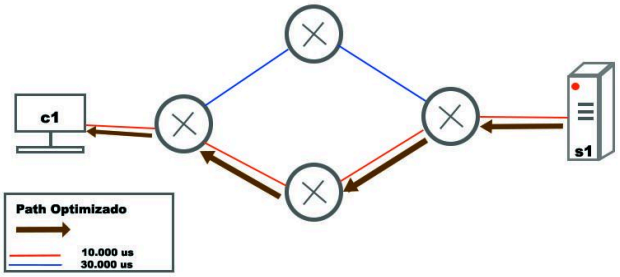

Figura 5. Transmissão de Vídeo

3.2.3. Procura de Melhor Caminho (*BFS*)

Para encontrar o melhor caminho, o servidor de 10 em 10 segundos liberta na rede um comando (*pathfinder*) em que o seu objetivo é, em cada nodo, percorrer todos os seus nodos vizinhos (sem voltar atrás), e com isto tentar encontrar o cliente que consta o payload da mensagem. Após encontrar o cliente, este último envia a mensagem de volta pela rede e exatamente pelo *path* correspondente a essa iteração do ***BFS* (*Breadth First Search)*** para o servidor com o tempo de chegada marcado. O servidor irá receber várias dessas rotas e irá constantemente escolher a que tem menor tempo, e se existir tempos iguais, a que tem menor quantidade de saltos. Caso este ***pathfinder*** não encontre nenhum cliente marcado previamente (como podemos ver na seta preta vertical), então não irá retornar, evitando com que o servidor processe algo desnecessário.

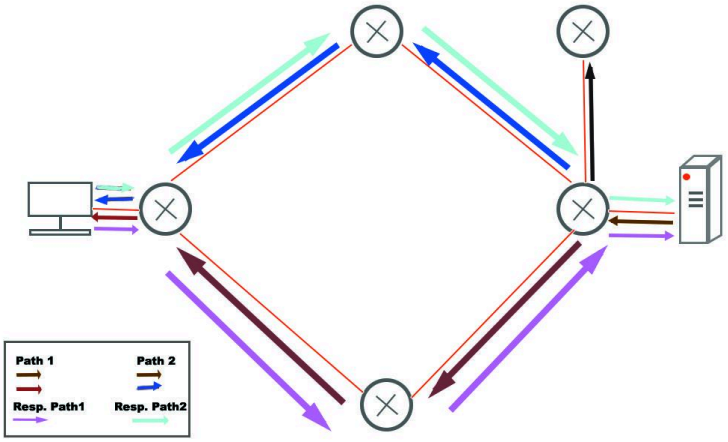

Figura 6. Procura de Melhor Caminho

4. Implementação

Relativamente ao processo de implementação foi seguida a lógica entregue pelo enunciado, portanto, ao longo desta secção, iremos explicar as nossas decisões guiando-nos pelo próprio.

4.1. Etapa 0: Preparação das Atividades

Como não foi usado nenhum esqueleto ao longo deste projeto, foi necessário nesta etapa criar um servidor e um cliente *full duplex*, com as suas interfaces a serem usadas para processos de *debugging* ao longo do projeto. Todos os programas ao longo das próximas etapas constam com o básico *Ouvir* e *Enviar* em *multithreading*.

4.2. Etapa 1: Construção da Topologia *Overlay* e *Underlay*

Nesta etapa foi construída toda a topologia, tanto *underlay*, como *overlay*. O underlay foi criado no *CORE* e o *overlay* é representado a partir de um ficheiro *JSON* que contém os Points of Presences de todos os clientes e que contém todas as interfaces adjacentes necessárias de cada nodo. O funcionamento do programa todo não necessita de ter o *overlay* todo ligado podendo ser só ligado o fundamental para o *streaming* existir.

A aplicação *oNode* é uma aplicação que simplesmente tem como objetivo redirecionar conteúdo ao longo da *CDN*. Foi optado por deixar o servidor entregar os dados da topologia ao iniciarmos os clientes e os nodos.

4.3. Etapa 2: Construção do *oClient*

Nesta etapa foi construído o *oClient* baseado no cliente da **Etapa 0**. Foi aumentado as funcionalidades do *oClient* consoante o enunciado, como fazer *RTT* ao *PoP* mais recente. Pode agora pedir ao servidor para enviar os *PoPs* e conta com a capacidade de pedir conteúdos não recebidos.

Foi melhorado o servidor para conseguir suportar este novo cliente, sendo que todas as comunicações são *UDP.*

4.4. Etapa 3: Serviço de *Streaming*

Na etapa 3, para conseguirmos fazer o *stream* dos conteúdos foi usado a biblioteca *OpenCV* (*Open Computer Vision Library - Google*). Esta biblioteca permite-nos dar *load* a todos os vídeos disponíveis em pastas para respectivos *buffers*, deu-nos a possibilidade de transformar cada *frame* em *JPG* e reduzir-lhe a qualidade, visto que o *UDP* tem um limite máximo teórico de 65507 *bytes*. Também foi necessário usar a biblioteca *pickle* para serializar os pacotes com informações necessárias, sendo que estas não são *UTF-8*.

O processo de *streaming* acontece com o servidor a ligar e a dar *load* aos *buffers* todos, o cliente pede a lista de vídeos disponíveis e escolhe um. Após a escolha, a primeira mensagem do servidor é a confirmar que está disponível e qual a *frame-rate* do vídeo. Após isto, o cliente entra na lista de clientes que querem ver vídeos e receber os *pacotes*.

Cada vídeo trabalha numa *thread* isolada evitando assim perdas de performance.
Na parte do cliente, é ativada uma thread que irá permitir ver o vídeo usando na mesma o *OpenCV*. Também é calculado o *Packet Loss* simplesmente incrementando sempre que o cliente recebe um pacote e com a numeração do último pacote enviado. Existem funcionalidades básicas como sair da stream “q” e pausar a stream “p”.

Por último, foi criada a capacidade nas três aplicações de entregar e receber conteúdo pela topologia, em vez de ser diretamente, sendo este o primeiro passo para a próxima etapa.

4.5. Etapa 4: Construção das Árvores de Distribuição

Esta última etapa principal foi a parte mais complexa a nível técnico. Para conseguirmos encontrar sempre o melhor path recorremos ao clássico algoritmo ***BFS (Breadth First Search)***. O objetivo deste algoritmo, com árvores construídas por fonte ativa, é simples: chegar ao cliente com o destino marcado, caso isto aconteça, o cliente responde com o mesmo *path* registado no *payload* da mensagem. O comando que permitiu isto foi a *Pathfinder* que tinha uma estrutura bem diferente:

Servidor Pathfinder:next_<nextIp>:
visited_<ip1>/.../<ipn>:currentpath_<ip1>/.../<ipn>:destination_<ipDest>

e a resposta do cliente é:

Cliente Path:<Todo o conteúdo do currentPath> time

Estas duas mensagens permitem dizer ao servidor o tempo de cada *path* e a quantidade de saltos necessários contando o número de *IPs* no *currentPath*.
Esta função é executada no servidor de 10 em 10 segundos ou quando algum cliente conecta-se. Com isto foi atingido um *Self Healing* das rotas ao longo de toda a rede.

4.6. Etapa Complementar 1: Monitorização da Rede Overlay

Esta etapa acabou por ser feita na **Etapa 4**, com a utilização do ***BFS***.

4.7. Etapa Complementar 2: Definição do Método de Recuperação de Falhas

Como na etapa anterior, esta também acaba por ser parcialmente feita, pois a partir do ***BFS***, se algum nodo se conectar e for vantajoso para algum cliente, o servidor irá descobrir e mudar a rota. O mesmo acontece caso haja nodos a desligarem-se. Este encontrará sempre a melhor rota.

4.8. Funcionalidades Adicionais

A nossa topologia pode suportar vários servidores distintos ao mesmo tempo, cada um entregando o seu próprio conteúdo.
O programa também facilmente suporta várias topologias, bastando fornecer os ficheiros underlay (*.imn*) e overlay (*.json*), desde que estes estejam bem definidos.

5. Testes e Resultados

Para testar e obter resultados do nosso projeto foi usada a seguinte topologia:

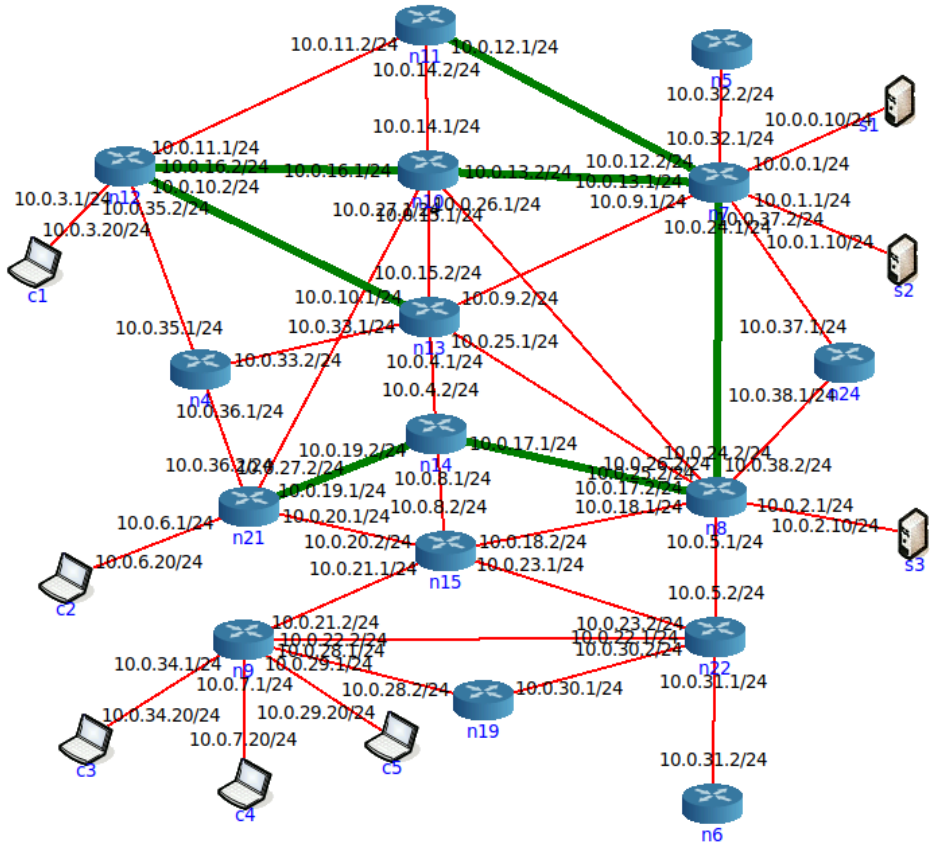

Figura 7. Topologia de Testes

Velocidades de Links Físicos:

Vermelho -> 10.000 us (10 ms)
Verde -> 1.000.000 us (1000 ms)

5.1. Transmissão de Vídeo para mais que um Cliente

No teste abaixo podemos ver três clientes, dois a ver a mesma *stream* e outro a ver uma stream diferente, tudo do mesmo servidor:

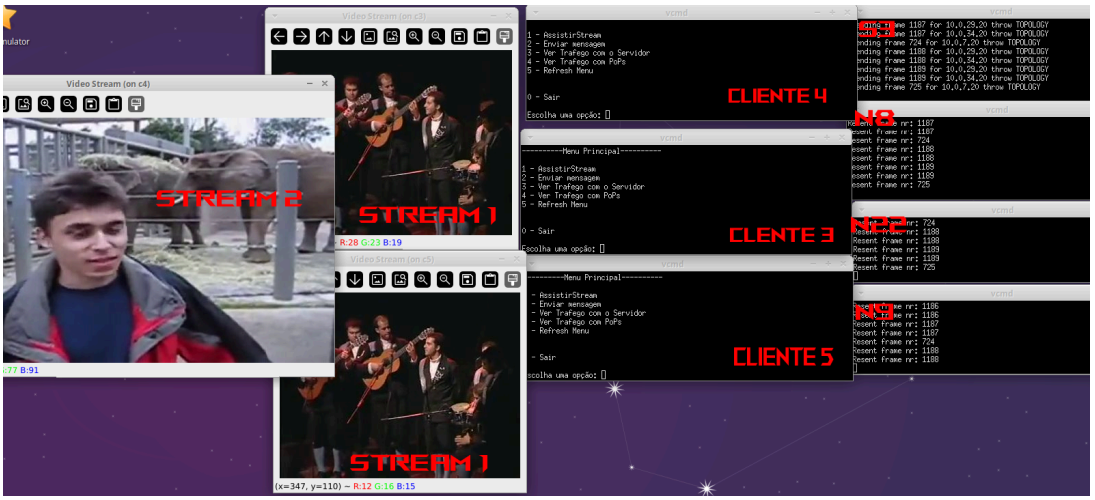

Figura 8. Teste de Transmissão de vídeo para mais que um cliente

Como podemos observar, os nodos estão a redirecionar pacotes com numeração diferente, mostrando assim que duas *streams* distintas estão a fluir por eles e conseguimos ver os dois clientes a ver a mesma *stream*.

5.2. Tratamento de Falhas + Descobrir o Melhor Caminho

No teste abaixo iremos mostrar duas funcionalidades, inicialmente no teste da esquerda, o *path* de cima é o que está a funcionar, após o abate desse terminal, o debaixo substitui:

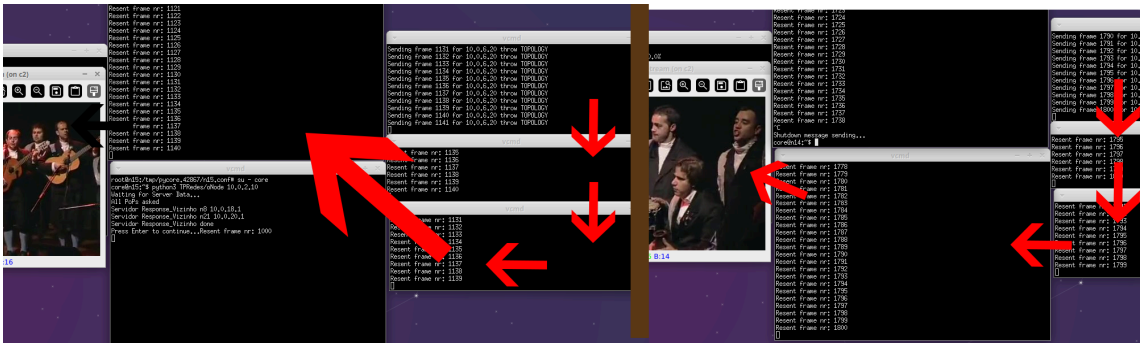

Figura 9. Teste de Tratamento de Falhas

No teste seguinte, demonstrado na próxima figura, temos a escolha de um caminho bastante complexo fazendo um “Z” por meio de um caminho, em que alguns deles têm latências bem superior aos outros (parte superior da nossa topologia):

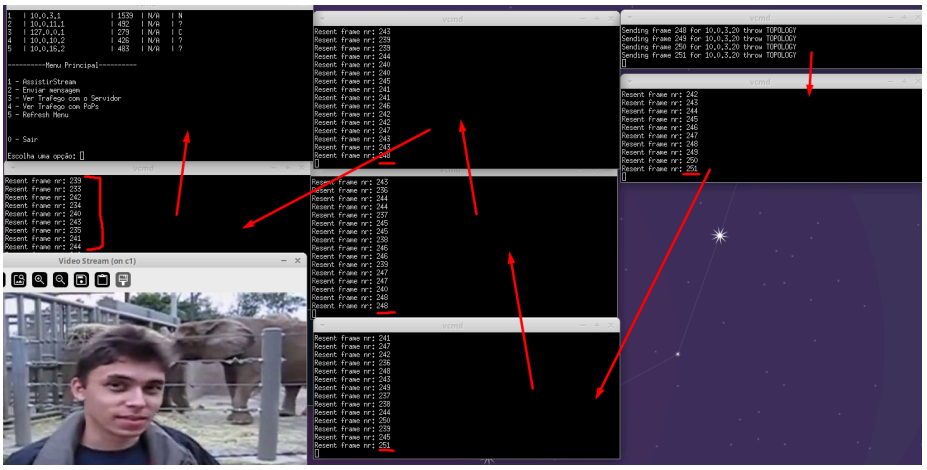

Figura 10. Teste de Descobrir o Melhor Caminho

5.3. Múltiplos Servidores a usar a mesma Topologia

Na próxima figura é possível visualizar o servidor a usar a topologia, para isso iremos usar o servidor 2 e 3 para servir conteúdo para o cliente 3 e 4 respetivamente:

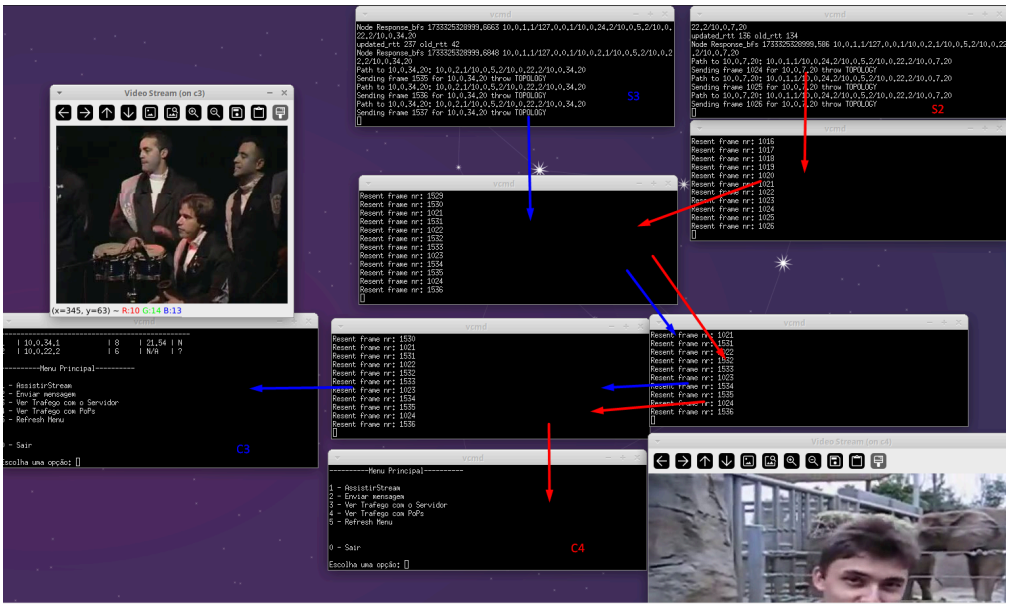

Figura 11. Teste de Múltiplo Servidores a usar a mesma Topologia

6. Conclusões e Trabalho Futuro

Concluindo, acreditamos que conseguimos implementar com sucesso todos os aspetos essenciais previstos para um protótipo de um serviço Over-the-Top (OTT) de *streaming* em tempo real. Além disso, fomos capazes de incorporar as funcionalidades sugeridas nas etapas complementares.

No entanto, identificamos como um maior desafio o tratamento adequado da situação de "morte catastrófica" de nodos. Assim, para um trabalho futuro, poderíamos aprimorar a arquitetura do programa, de forma a lidar melhor com este cenário específico.

Por fim, consideramos que alcançamos os objetivos definidos no enunciado. Reconhecemos, entretanto, que existem oportunidades para refinamentos e uma possível adição de outras funcionalidades, para melhorar a nossa solução.