Engenharia de Serviços em Rede

TP3

Trabalho realizado por:

Carlos Miguel Luzia de Carvalho

Paulo Silva Sousa

Rui Emanuel Gomes Vieira

PG47092
Carlos Carvalho

PG47556
Paulo Sousa

PG47635
Rui Vieira

Grupo 25
2021/2022
Universidade do Minho

Conteúdo

1 Introdução

Como terceiro projeto na Unidade Curricular de Engenharia de Serviços em Rede foi nos proposto 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, assim a forma como o *overlay* aplicacional se constitui e organiza é determinante para a qualidade de serviço que é capaz de suportar cabendo ao grupo tomar determinadas decisões nessa definição.

Este projeto tem a cima de tudo que cumprir um determinado número mínimo de requisitos sendo estes:

- Um nó da rede *overlay* é um programa (aplicação) que se executa manualmente numa máquina da rede IP de suporte;
- Os nós da rede *overlay* podem ser criados e destruídos dinamicamente (início e fim de execução do software);
- Quando se junta à rede *overlay*, um nó deve estabelecer e manter conexões de transporte (TCP ou UDP) com X nós da rede, que serão seus vizinhos;
- Ao iniciar a execução, o nó da rede tem de conhecer pelo menos um outro nó da rede, que pode ser o servidor de conteúdos, que o ajuda a obter a sua lista de X vizinhos;
- Os nós da rede *overlay* precisam de construir e manter uma tabela de rotas (uma por fluxo); como só há um fluxo, apenas uma rota (Fluxo, Custo, Origem, Destinos, Estado);
- Os nós da rede *overlay* reenviam todos os dados de acordo com essa tabela;
- Os nós da rede *overlay* reenviam todos os dados de acordo com essa tabela;
- Em caso de não saber o que fazer aos pacotes, reenviar por inundação controlada, para todos;

Na perspectiva de cumprir todos os requisitos anteriormente enunciados o grupo decidiu então desenvolver o projeto em **Java** sendo o seu desenvolvimento explicado nos restantes tópicos deste relatório.

2 Arquitetura da Solução

Neste trabalhos tivemos a necessidade de criar diversas classes cada uma com um diferente propósito, assim passamos a explicar de seguida o propósito adjacente a cada uma destas classes.

2.1 AddressingTable

Classe responsável por criar e atualizar as tabelas de endereçamento dos diferentes nodos, como também criar os ficheiros "log"com o uso do comando "ping".

Nesta classe temos:

- *Map<String, Map<Integer, Boolean» table* - Map com as rotas a que o nodo transmite com um segundo map que contém os ids das diversas streams associados a um boolean (que representa se essa rota pretende receber a stream com esse id)
- *Map<Integer, Boolean> isClientStream* - Map com o id das streams associados a um boolean (que representa se o próprio nodo pretende apresentar a stream com esse id)
- *Map<Integer, Boolean> neighbours* - Map com o ip dos vizinhos e um boolean que representa se este se encontra conectado na rede
- *Set<String> neighbours_temp* - Set com os vizinhos temporários, adicionados quando uma rota deste nodo sai da aplicação e as rotas deste que saiu não têm vizinhos com rota alternativa disponível
- *String ip* - Ip do nodo
- *String sender* - Ip do nodo anterior a este na rota
- *String senderSender* - Ip do nodo anterior ao sender na rota
- *int hops* - Número de saltos até ao nodo
- *int numStreams* - Número de streams disponíveis

2.2 Bootstrapper

Será nesta classe que o ficheiros de *bootstraper* respetivo com a topologia a usar será interpretado e onde futuramente os nodos do *overlay* terão acesso aos seus vizinhos

Nesta classe temos:

- *Map <String, BootstrapperCollumn> bootstrapper* - Map com o ip dos nodos associados a uma class BootstrapperCollumn
- *ReentrantLock lock* - Lock para controlo de concorrência
- *Condition full* - Condição que nos permite deixar o servidor em espera até metade dos nodos da topologia entrar na rede (pedir vizinhos)

2.2.1 BootstrapperCollumn

Nesta classe guardamos a informação dos vizinhos de um nodo e se estes já foram pedidos pelo nodo em questão.

Nesta class temos:

- *Set<String> neighbours* - Set com os ips dos vizinhos
- *boolean visited* - boolean que representa se os vizinhos já foram pedidos pelo nodo ou não

2.3 ClientDisplay

Classe em que é definido todo o *display* a que o cliente terá acesso ao pedir o acesso uma dada *stream* com as suas respetivas utilidades. Estas utilidades são o *play*, que corre a *stream* e o *stop*, que informa que o cliente quer parar de assistir à *stream*.

2.4 ClientStream

2.5 Ott

Classe principal do nosso programa que é usada para iniciar o serviço de *stream* nos nodos podendo este serem servidores, clientes ou apenas nodos do *overlay*. Dependendo do que o nodo é este será iniciado com diferentes funções.

2.6 OttBeaconSender

Classe responsável pelo envio de beacons do servidor para o resto da rede *overlay* para, assim, verificar se nenhum dos nodos Ott se desligou sem aviso prévio.

2.7 OttReceiverTCP

Classe responsável por decidir o que fazer dependendo do pacote recebido num dado Ott por TCP.

2.8 OttSenderTCP

Classe responsável pelo envio dos pacotes por TCP.

2.9 OttStream

Classe que recebe os pacotes da *stream* e os distribui pela rede.

2.10 Packet

Classe onde estão definidos e são tratados os pacotes enviados por TCP, nesta definição temos então uma *string* correspondente ao IP da source, uma outra correspondente ao IP do destino, um *integer* correspondente ao tipo do pacote (sendo este tipo uma questão de organização para os respetivos pacotes e ainda um array de bytes onde efetivamente irá a informação do pacote.

Nesta classe temos:

- *String source* - Ip do nodo da *source* do pacote
- *String destination* - Ip do nodo do destinatário do pacote

- *int type* - Número que identifica o tipo do pacote
- *byte[] data* - Dados que o pacote transporta

2.11 PacketQueue

Classe responsável pela criação de uma Queue de pacotes, armazenando assim os pacotes para envio ou recebidos por TCP numa lista ligada, tornando mais fácil o seu tratamento.

2.12 RTPpacket

Classe onde estão definidos e tratados os pacotes enviados por UDP, estes contém o vídeo a ser stremado para os clientes.

2.13 RTPqueue

Esta classe é muito similar à PacketQueue, sendo que serve para o armazenamento dos RTPpackets para envio ou a receber em UDP.

2.14 ServerFload

Classe responsável por realizar o fload de rotas para a topologia. Esta é realizada quando metade dos nodos entram na topologia e de 20 em 20 segundos cajo existam alterações.

2.15 ServerReceiverTCP

Classe responsável pelo tratamento de pacotes recebidos por TCP por parte do servidor, tendo assim em conta o tipo do pacote que está a receber e efetuando de seguida as instruções necessárias tendo em conta o pacote recebido.

2.16 ServerSenderTCP

Classe responsável pela resposta dos servidores aos Otts aquando se dão alterações na rede.

2.17 ServerSenderUDP

Classe responsável por invocar a ServerStream para um determinado id de stream.

2.18 ServerStream

Classe responsável pelo envio em UDP dos pacotes de *stream* para os respetivos clientes.

2.19 VideoStream

´

Classe responsável pela leitura do video do ficheiro e por devidi-lo em frames para ser enviado para a Stream.

3 Especificação dos Protocolos

Neste trabalho, optamos por usar dois tipos de protocolos, TCP para a comunicação entre o Server e os Otts no que toca a gestão da Rede e UDP para o envio da stream do Server para os clientes.

3.1 TCP

Relativamente aos pacotes enviados por TCP criamos identificativos de tipo para o pacote permitindo assim ao recetor desse pacote compreender que alterações aconteceram ou que ações é que ele tem de tomar.

Assim, será explicitado na tabela em baixo todos os tipos possíveis de pacotes a enviar.

Tipo	Responsabilidade
1	Este tipo de pacotes é enviado por um Ott ao Server de forma a pedir a este que lhe envie uma pacote com a informação sobre os seus vizinhos
2	Este tipo de pacote é enviado do Server ao Ott que lhe pediu a lista de vizinhos, com a lista pedida informando também que ainda não está a enviar pacotes de vídeo
3	Este tipo de pacote é enviado do Server ao Ott que lhe pediu a lista de vizinhos, com a lista pedida informando também já está a enviar pacotes de vídeo
4	Este pacote é enviado de um Ott para os seus respetivos vizinhos como forma de pedido
5	Pacote é enviado de um Ott para os seus vizinhos a dizer que é um caminho possível
6	Pacote enviado ao remetente do pacote to tipo 5 assinalando assim que aceita o caminho como ideal
8	Pacote enviado que informa a atualização da rota
9	Pacote enviado de um nodo para os seus vizinhos que emite o facto que o mesmo irá sair
11	Pacote enviado ao vizinho que faz parte da sua rota ótima que pretende transmitir a *stream*
12	Pacote enviado ao vizinho que faz parte da sua rota ótima que não pretende mais a transmissão da *stream*
13	Pacote enviado de um nodo para os vizinhos que possuam uma rota com este para apagar as rotas, este é espalhado por toda a rede *overlay*
14	Pacote enviado de um nodo para avisar o servidor a avisar que irá deixar de existir
15	Pacote que será enviado pela rede para ser feito o *ping* dos vários nodos do *overlay*, ou seja, a escrita para ficheiro de log
16	Pacote enviado de um dos nodos para o servidor a pedir a execução de um novo *fload*
17	Pacote enviado com o *fload* sem redicerionamento
18	Pacote que sinaliza o envio de um *beacon*
19	Pacote enviado para remover os vizinhos temporários que tem em comum com o nodo que envia o pacote
20	Pacote enviado para acrescentar o nodo como vizinho temporário

Tabela 1: Tipos de pacotes presentes no protocolo TCP

3.2 UDP

Os pacotes enviados por UDP são relativos aos pacotes relacionados com as várias *streams* que o servidor irá transmitir aos clientes que as quiserem.

4 Implementação

Nesta secção vamos descrever todos os processos e a forma como implementamos as funcionalidades do nosso programa, nesta descrição estarão inseridas as formas como definimos a **Topologia de Rede** e fazemos a **Gestão e Controlo do Fluxo**.

4.1 Detalhes e interações

4.1.1 Pedido de vizinhos

Para a definição da rede overlay escolhemos uma estratégia baseada num controlador, ou seja temos um **bootstrapper** construído manualmente por nós que tem todos os nodos da rede e os seus respetivos vizinhos, estando construído da seguinte forma:

ip_nodo : ip_vizinho1, ... ,ip_vizinhoN

Primeiramente, o Servidor entra na rede e carrega toda a informação do ficheiro *bootstrapper* para a classe *Bootstrapper*. Após carregar os vizinhos para essa classe vai buscar os seus vizinhos e guarda num *Map* na *AddressingTable*.

Quando um nodo entra na rede envia um pacote do tipo 1 ao servidor (cujo ip é dado como argumento). Ao receber este pacote o servidor envia um pacote do tipo 2 com os vizinhos caso não tenha sido feito fload ou um pacote do tipo 3 com os vizinhos caso já tenha sido feito o fload.

4.1.2 Fload

Quando vão entrando nodos na rede, a classe *Boostrapper* contabiliza quando entram na rede e enviam um *signal()* para uma condição.

Já a classe *ServerFload* chama a função *full()* que tem a mesma condição em *await()*, em que apenas sai desta condição se metade dos nodos entrarem na rede. Quando a condição sai do ciclo esta classe avança para o primeiro fload. Após o primeiro, a thread fica num ciclo em que fica em *sleep* durante 20 segundos e depois volta a executar o fload se houve alterações na rede (entraram ou sairam nodos).

Para o fload, primeiramente o servidor envia um pacote do tipo 13 aos seus vizinhos e vizinhos temporários para as tabelas serem limpas (remover número de saltos, sender, senderSender e Map com encaminhamentos). Cada nodo que recebe este pacote envia-o primeiro aos seus vizinhos e vizinhos temporários e só depois limpa as suas tabelas.

Após as tabelas estarem limpas, o servidor envia um pacote do tipo 5 aos seus vizinhos e vizinhos temporários com o número de saltos 1. Ao receber este pacote, o nodo confirma se o número de saltos é menor do que o que encontra atualmente na tabela. Se este for o caso, o nodo atualiza o seu sender, senderSender e número de saltos. Além disso, se já tiver um caminho de outro nodo, avisa-o que quer deixar de receber stream dele (pacote do tipo 12) e para remover a rota da sua tabela (pacote do tipo 8). Posteriormente, notifica o nodo do novo caminho para o adicionar à sua tabela de rotas (pacote do tipo 6) e que streams pretende receber (pacote do tipo 11). Por último, envia a todos os vizinhos (exceto o que lhe envia a rota) e vizinhos temporários um novo pacote do tipo 5 com o número de saltos incrementados por 1.

4.1.3 Adicionar Nodo

Quando um nodo entra na rede faz um pedido de vizinhos ao servidor como está descrito no ponto 4.1.1.

Se receber um pacote do tipo 2 fica à espera do fload do servidor.

Caso receba um pacote do tipo 3 vai enviar aos seus vizinhos um pacote do tipo 16 a pedir rotas para transmissão. Além disso, envia também aos seus vizinhos um pacote do tipo 19 a avisar que podem remover os vizinhos temporários que tem em comum com este.

Um nodo ao receber um pacote do tipo 16 envia de volta ao nodo que entrou um pacote do tipo 17 com o número de saltos da rota. O nodo ao receber um pacote do tipo 17 executa o mesmo procedimento que num pacote do tipo 5, com a diferença que o pacote não é propagado aos vizinhos deste.

Este método permite um estabelecimento rápido de rotas, embora não sejam mais eficientes. Estas rotas serão depois repostas pelo fload periódico do servidor.

4.1.4 Remover Nodo

Um nodo pode sair da rede de duas formas, através da escrita no terminal do comando *exit*, em que este nodo envia um pacote do tipo 9 a todos as suas rotas e envia um pacote do tipo 8 ao nodo que lhe envia stream para este o remover da tabela de rotas, ou através de *CONTROL+C*, em que o processo é lidado pelo envio de beacons como vai ser explicado mais à frente. Em qualquer das formas, seja recebendo um pacote do tipo 9 ou não conseguindo estabelecer ligação no caso dos beacons, a situação é lidada da mesma forma.

Primeiramente, o nodo envia um pacote do tipo 13 às suas rotas para limpar as tabelas e depois limpa as suas próprias rotas.

Após isso, verifica se tem algum nodo com que pode estabelecer ligação, pegando em todos os vizinhos e removendo do lista o nodo que lhe envia caminho e os nodos com que tem rota. No caso de não ter nenhum a que se possa ligar, vai enviar um pacote do tipo 4 para o senderSender (nodo que tem o sender do nodo atual na sua tabela de rotas) a pedir caminho, um pacote do tipo 14 que vai sendo enviado até ao servidor a avisar que houve alterações na rede e um pacote do tipo 20 para adicionar o nodo aos vizinhos temporários.

Ao receber um pacote do tipo 4, o nodo envia de volta um pacote do tipo 5 com o número de saltos. A partir daqui é feito um *"mini-fload"*, em que é feito um fload como explicado no tópico 4.1.2 mas apenas a partir do nodo que saiu, não na rede toda.

Este método permite um estabelecimento rápido de rotas, embora não sejam mais eficientes. Estas rotas serão depois repostas pelo fload periódico do servidor.

4.1.5 Pedir Stream

Quando um cliente pede para receber stream escreve no terminal o número da stream que pretende visualizar. Assim, é enviado um pacote do tipo 11 ao seu sender com o id da stream pretendida. O nodo ao receber este pacote, se já estiver a receber a stream pedida, apenas vai alterar o *Map* das streams na *AddressingTable* e a thread *OttStream* começa a enviar stream para esse nodo. Caso ainda não esteja a receber a stream pedida, vai enviar outra vez um pacote do tipo 11 ao seu sender até chegar ao servidor e adiciona da mesma forma ao *Map* das streams.

Para parar de receber a stream, o cliente apenas tem de carregar no botão stop. Deste modo é enviado um pacote do tipo 12 e o processo é realizado de modo inverso.

4.1.6 Envio de Beacons

De modo a verificar se as ligações entre um nodo e os seus vizinhos está ativa utilizamos a thread *OttBeaconSender*. Esta thread está num ciclo infinito que faz *sleep()* de 400 milissegundos e depois envia um beacon para todos os seus vizinhos.

Se o beacon for bem entregue não é feito nada. Se não for bem entregue, temos três casos. Se for o sender dele, o tratamento deste está descrito no tópico 4.1.4. Se for uma rota, remove-o da tabela. Por último, se não for nenhum dos dois, marca o vizinho como *false* no *Map* de vizinhos.

4.1.7 Ping

Quando se pretende ver a informação das tabelas de rotas escreve-se *ping* no terminal do servidor. Este vai enviar um pacote do tipo 15 a todas as suas rotas e executa a função *ping()*. Cada nodo que recebe este pacote faz o mesmo procedimento que o servidor.

A função *ping()* vai escrever num ficheiro log o seu ip, o seu sender, número de hops, que stream pretende receber e o as suas rotas (junto também com que streams pretendem receber).

4.2 Bibliotecas de funções

Para a elaboração do trabalho foi imprescindível o uso de diversas bibliotecas de Java existentes que nos auxiliaram de forma positiva no desenvolvimento do projeto.

Assim, as bibliotecas que empregamos foram as seguintes:

- java.io.*
- java.nio.*
- java.util.*
- javax.swing.*
- java.awt.*
- java.net.*

5 Testes e Resultados

De forma a realizar os diversos testes necessários para compreender dar resposta à totalidade dos requisitos deste projeto, decidimos usar a topologia apresentada a baixo.

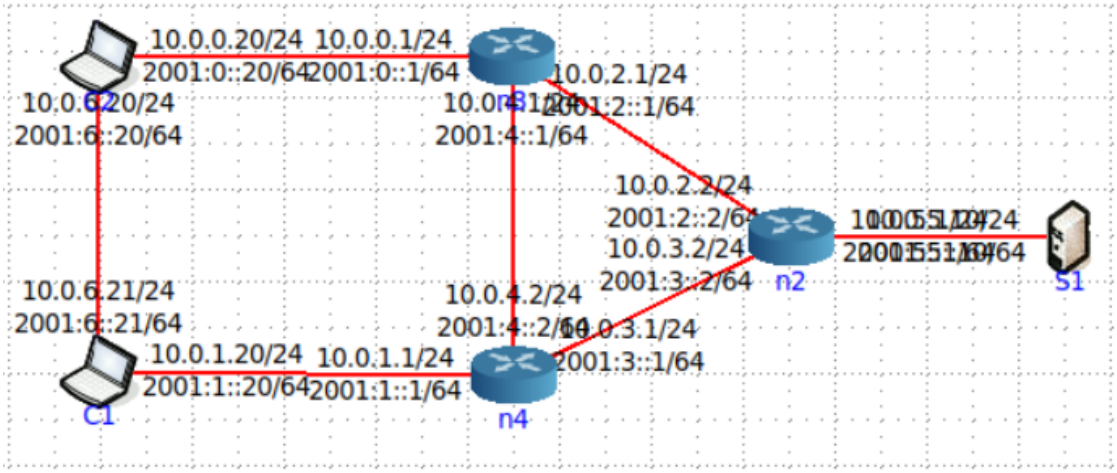

Figura 1: Topologia de Teste

È de reparar nesta topologia de de teste que acabamos por cirar uma ligação entre os dois clientes de forma a que eles possa enviar entre eles também.

Para de melhor forma mostrarmos as features do nosso projeto fizemos uma testagem sequencial do que acontece em determinadas situações, assim, começámos por ligar o Servidor S1, o nodo n2 o nodo n3 e os clientes C2 e C1.

Dentro deste esquema, vamos pedir stream 1 e 2 no Cliente C2.

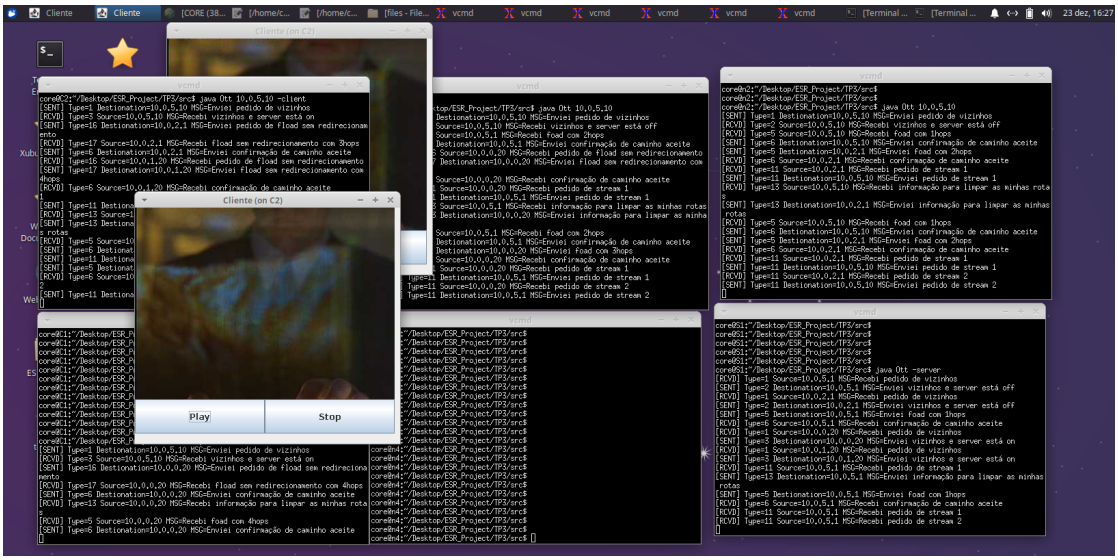

Figura 2: Cliente C2 com várias streams simultâneas

De seguida pedimos stream no cliente C1 de forma a comprovar que ambos os clientes recebem as streams.

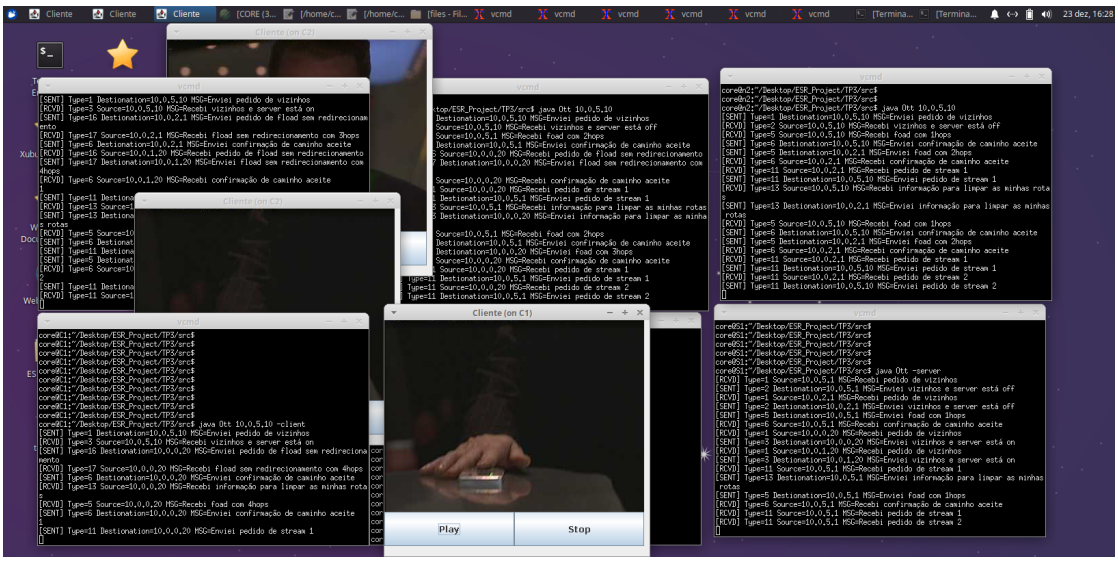

Figura 3: Cliente C2 com várias streams simultâneas e C1 com uma stream

Depois de confirmar que ambos os clientes estão a receber stream vamos fechar o nodo n3 de forma a verificar que ambas as streams continuam a reproduzir e que o C2 se consegue ligar ao nodo n2 uma vez que nenhum dos seus vizinhos tem conexão ao servidor. De notar que o nodo n3 foi interrompido/mandado a baixo ou seja sem sem qualquer aviso.

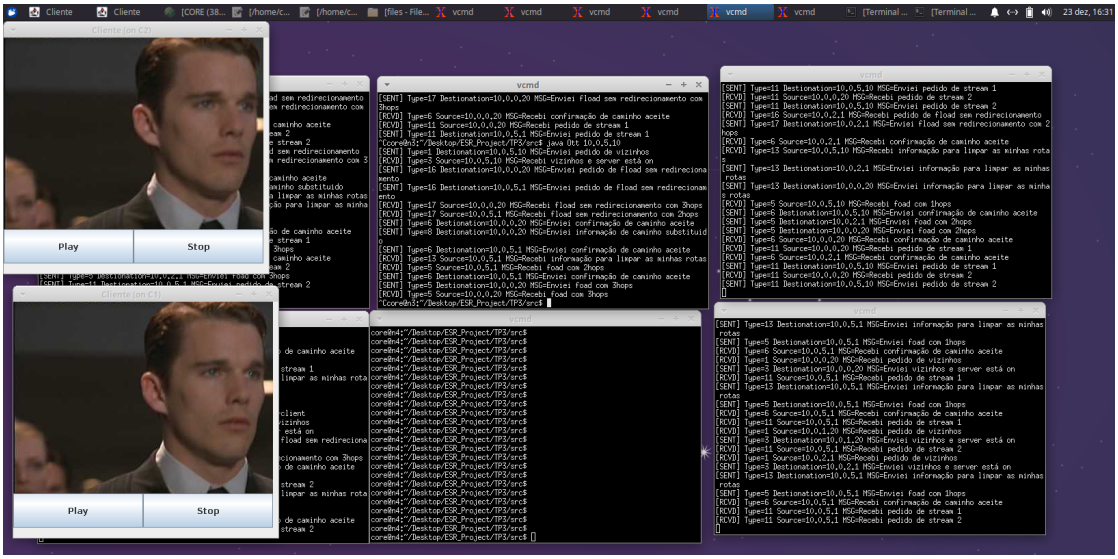

Figura 4: Nodo 3 sai sem aviso, Streams continuam em reprodução

Abaixo vamos demonstrar o mesmo teste mas com um safe exit, ou seja o nodo n3 avisa que vai sair.

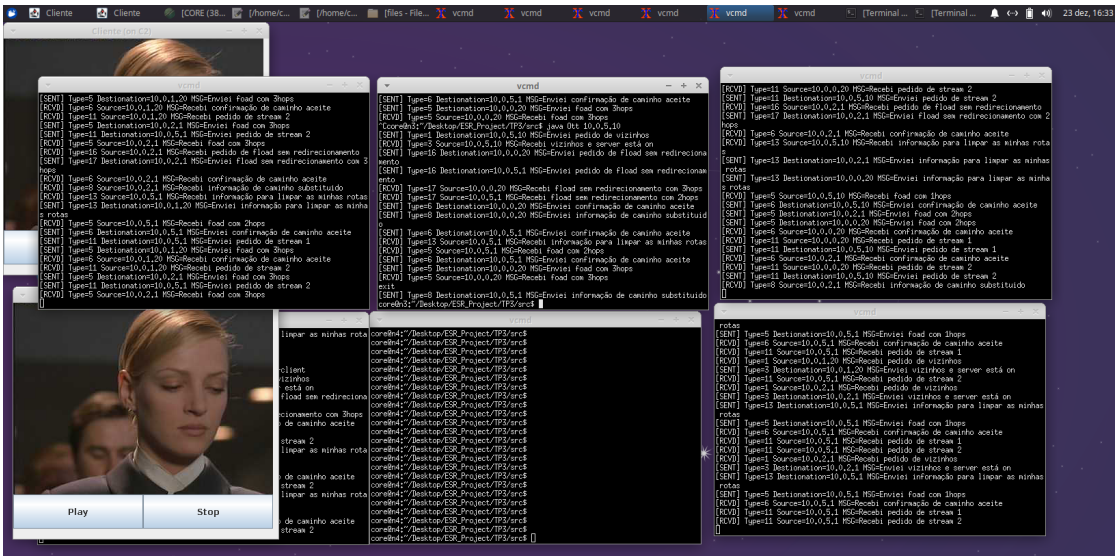

Figura 5: Nodo 3 avisa que vai sair, Streams continuam em reprodução

De seguida, vamos ligar o nodo n4, e vamos verificar que o sistema vai preferir a rota s1>n2>n4>C1>C2 uma vez que anteriormente C2 estava ligado directamente a n2, não estando este na sua lista de vizinhos. Como n4 é um vizinho de n2 e C1 o sistema vai preferir o caminho que tem em conta os vizinhos existentes e não o caminho forçado ente C2 e n2.

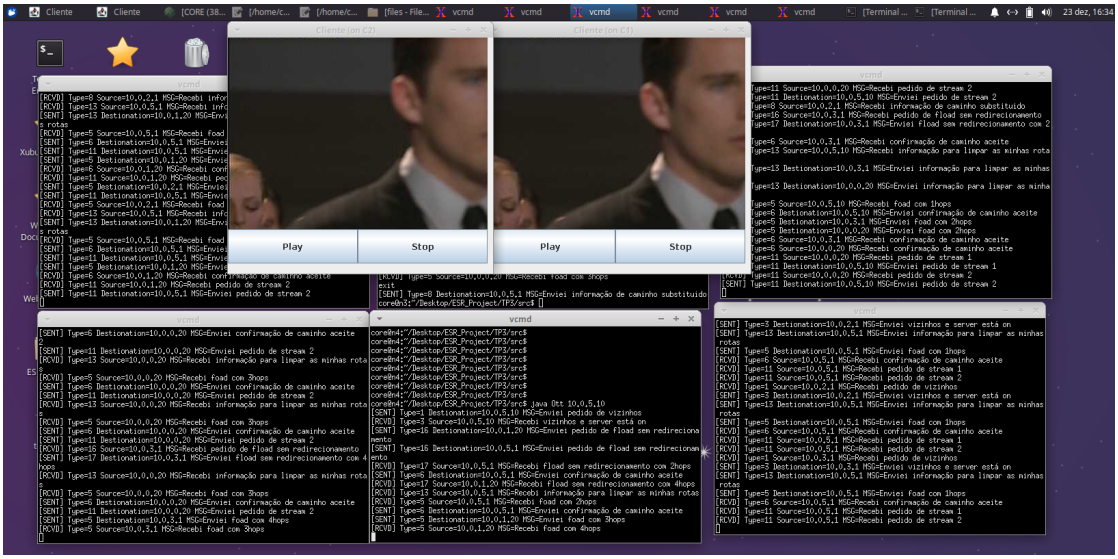

Figura 6: Alteração de Rotas quando Ligado N4, Stream em Reprodução

De forma a demonstrar que o sistema escolhe entre várias rotas as duas próximas imagens vão ser bastante elucidativas, na primeira poderemos ver todos os nodos da rede ligados, estando C1 e C2 ligados com aos vizinhos que lhes fornecem os melhores caminhos possíveis. Já na segunda o nodo n4 é desligado e o cliente C1 acaba por tomar um caminho alternativo ligando-se ao C2.

É de notar que nenhuma das streams deixa de reproduzir.

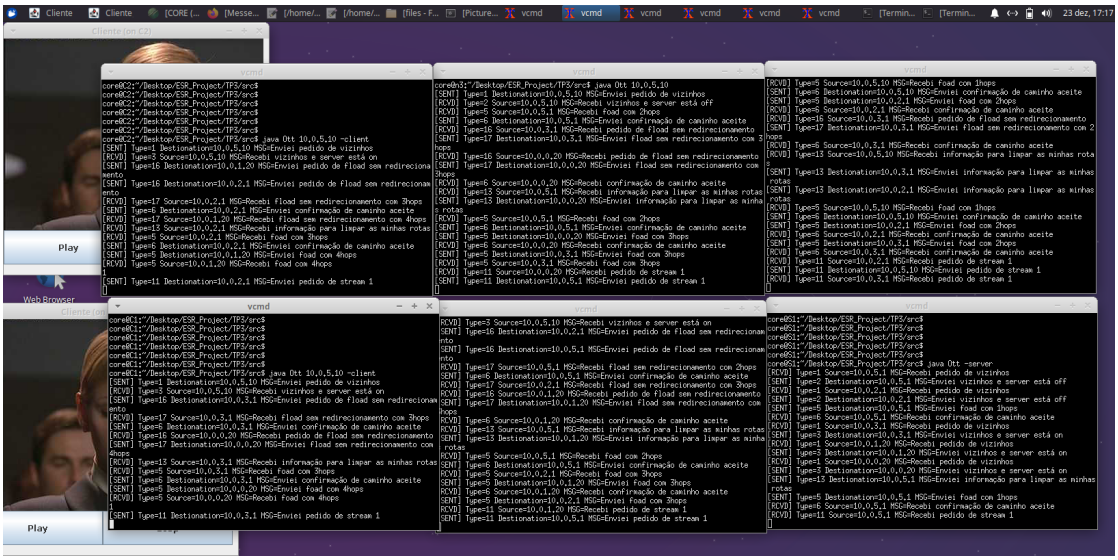

Figura 7: Topologia Toda ligada

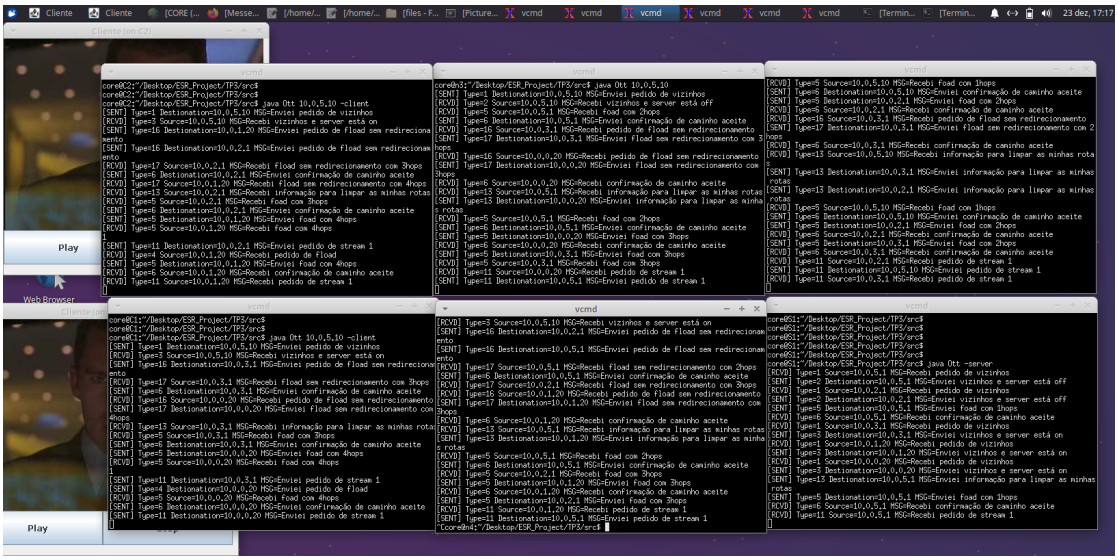

Figura 8: N4 desligado, nova rota para C1

6 Conclusão

Neste trabalho foi nos proposto conceber um protótipo de entrega áudio/vídeo/texto com requisitos de tempo real, a partir de um servidor de conteúdos para um conjunto N de clientes, tendo isto em conta o grupo conclui que, apesar de ter encontrado diversas adversidades, conseguiu ultrapassa-las e realizar este projecto alcançando todos os objectivos a que nos propusemos.

Consideramos ainda que, com a elaboração deste trabalho prático da UC de Engenharia de Serviços em Rede, consolidamos bem o conhecimento adquirido no decorrer da licenciatura como também da própria UC.