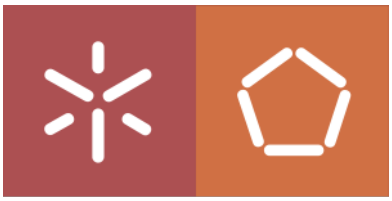
Engenharia de Serviços em Rede

TP2 - Serviço Over the Top para entrega de multimédia

Autores:
João Paulo Sousa Mendes (pg50483)
João Silva Torres (pg50499)
José Diogo Martins Vieira (pg50518)

December 15, 2022



Universidade do Minho
Escola de Engenharia

1 Introdução

No âmbito do segundo trabalho prático da UC Engenharia de Serviçoes em Rede foi-nos proposto desenvolver um programa com o objetivo de efetuar **multicast aplicacional** de forma mais eficiente e otimizada possível para melhor qualidade. Assim sendo, foi necessário considerar diferentes etapas para suportar toda a lógica do programa.

Usando primariamente o emulador CORE como bancada de teste, e uma ou mais topologias de teste, 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.

Numa fase inicial, definimos qual a linguagem que iriamos utilizar. Escolhemos Java, visto que é uma linguagem em que o grupo se encontra familiarizado. Foram também criados cenários de teste, bem como a definição do protocolo de transporte.

Ao longo deste relatório iremos explicar pormenorizadamente todas as etapas e estratégias utilizadas durante o desenvolvimento do projeto.

Numa fase final do relatório faremos não só uma análise do trabalho, bem como do grupo.

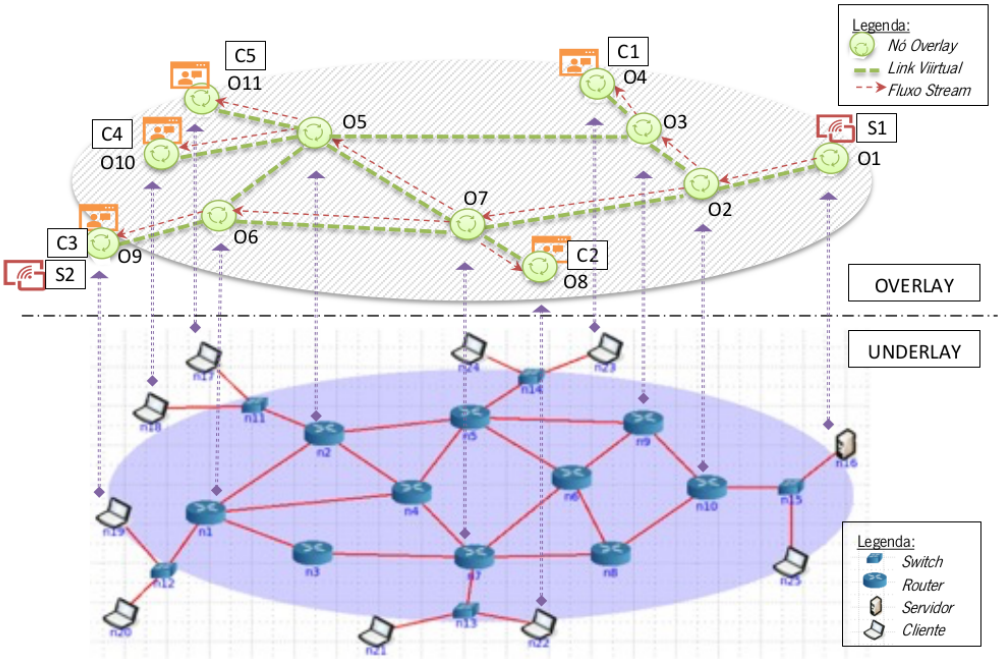


Figure 1: Visão geral de um serviço OTT sobre uma infraestrutura IP

2 Arquitetura da solução

Segundo o enunciado do projeto temos de implementar um protocolo de *streaming*. Assim sendo, utilizamos os dois protocolos de transporte: UDP e TCP.

O UDP foi utilizado para o funcionamento de *streaming* apesar de sabermos que tem certos riscos, relativamente a perda de pacotes. No entanto, é mais rápido em termos de envio dos mesmos e não afetará significamente a qualidade de serviço. Por outro lado, utilizamos o protocolo TCP para a sincronização da rede Overlay. Ademais, como se pode ver na imagem que segue, criamos classes específicas para o Cliente, Servidor, ServidorAdicional e Node, sendo estas as classes por onde o streaming é feito. Para além disso, elaboramos uma classe, Metrica, responsável por descobrir qual a melhor rota para o melhor servidor através do tempo e do número de saltos. Aproveitamos por base o código fornecido pela equipa docente na elaboração de algumas classes. Ao longo do relatório explicamos melhor o funcionamento de cada um mais detalhadamente.

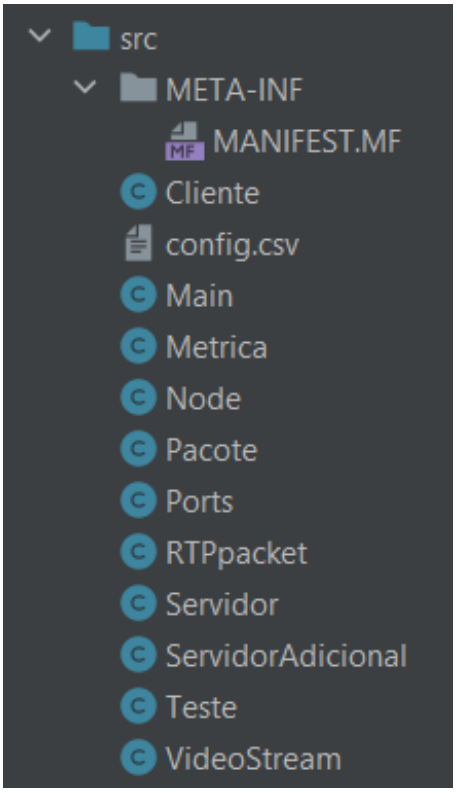


Figure 2: Arquitetura da solução

3 Especificação do protocolo

De acordo com os requisitos estabelecidos no enunciado do trabalho, foi necessário criar protocolos aplicacionais de controlo/sincronização e um protocolo de streaming. Por isso surgem as classes Pacote e RTPPacket, onde são implementadas as mensagens protocolares de controlo e de streaming, respetivamente.

Pacote

- origem : InetAddress;
- saltos: Int;
- conteudo : Int;
- timestamp : Long.

O propósito dos diversos atributos numerados anteriormente, depende do serviço a que esteja sujeito. Os serviços serão descritos mais adiante no relatório bem como o seu funcionamento de acordo com os atributos.

RTPPacket

Utilizado para a transmissão de conteúdos de *streaming* pela topologia. Esta classe foi fornecida pela equipa docente, de forma a apoiar a realização do trabalho. A sua estrutura não foi alterada, continuando com as suas funcionalidades de origem.

3.1 Interações

De seguida, através do diagrama de sequência, podemos observar uma ordem de envio de mensagens protocolares na topologia

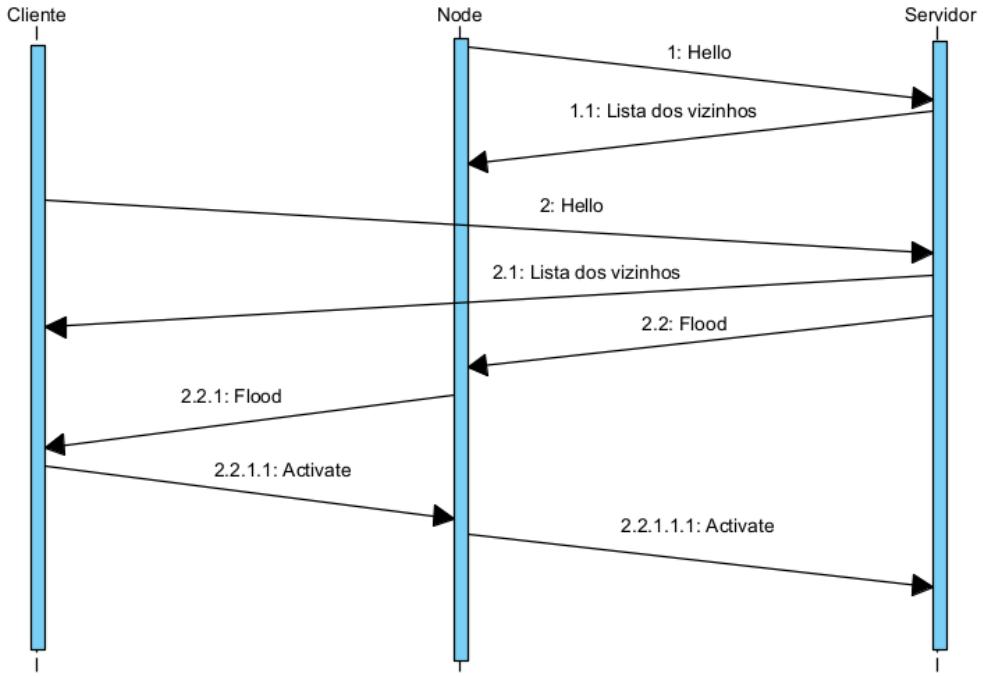


Figure 3: Interações Pacote

4 Implementação

4.1 Etapa 1 - Construção da Topologia Overlay

Para a construção da rede overlay baseamo-nos num construtor. Isto é, ao executar o programa, indicamos apenas um nó para o arranque da rede que começa por fazer a leitura de um ficheiro configuração, que será construído manualmente.

Assim sendo, o *bootstrapper* (servidor) irá ler o ficheiro de configuração que contém toda a informação acerca do overlay, e vai estruturar essa informação de modo a saber todos os nodos da rede e os respetivos vizinhos, assim como os nodos ativos.

Este servidor principal que será o *bootstrapper*, terá que ser o primeiro a ser executado, sendo executado com a *flag* "-S" e o respetivo ficheiro de configuração com a informação.

Após isto, irá ficar à espera que os restantes membros do overlay se registem. Este registo é feito através de TCP. Cada membro do overlay, podendo este ser um Cliente, Nodo ou Servidor Adicional será executado com as *flags*: "-C", "-N" e "-S2", respetivamente, seguido de um argumento que será comum entre estes, o IP do Servidor (*bootstrapper*).

Cada membro, ao executar o programa, conecta-se com o Servidor e envia uma mensagem de "Hello" para anunciar a sua chegada. O Servidor atualiza o estado deste nodo para ativo e devolve a lista dos seus vizinhos.

É assim mantido um Socket de cada membro do overlay com o servidor, até que este abandone o overlay. E no servidor é mantida uma thread para cada conexão. Esta thread fica sempre à escuta até a conexão morrer.

Quando a conexão morre, sabemos que o nodo em questão abandonou a rede. O Servidor atualiza assim o seu estado e avisa também os vizinhos desse nodo que este já não se encontra ativo.

4.2 Etapa 2 - Serviço de Streaming

Para a elaboração do Serviço de *Streaming* utilizamos como base o código fornecido pela equipa docente.

Os Servidores são os responsáveis pela leitura do ficheiro de vídeo e enviam os pacotes RTP para os nodos, que são seus vizinhos, caso haja algum cliente a pedir vídeo. Esta parte da ativação das rotas será explicada posteriormente. O nodo recebe assim os pacotes RTP e envia para um ou mais clientes/nodos caso as rotas destes estejam ativas.

Por fim, o cliente recebe o pacote e apresenta a imagem na janela de vídeo. O vídeo de *streaming* está em constante *loop*, e deixa de ser entregue pelos Servidores quando não há clientes ativos.

4.3 Etapa 3 - Monitorização da Rede Overlay

A monitorização da rede é feita através do serviço *flood*. Este utiliza a classe Pacote para conter toda a informação necessária:

- origem: relativo ao servidor que fornece a *stream*;
- saltos: referente ao número de saltos entre os nodos;
- conteúdo: referente ao número de sequência do flood;
- timestamp: referente ao momento que é enviado o flood.

O serviço de *flood* inicia no Servidor, onde este envia um Pacote para os vizinhos com toda a informação explicada anteriormente, onde o saltos inicia a zero. Os nodos estão ativamente à espera deste pacote, e ao receberem, vão verificar a origem e o conteúdo. Se já tiverem recebido um pacote com esta combinação, não vão enviar mais. Caso seja a primeira vez a receber esta combinação, o pacote irá ser enviado para os vizinhos do nodo, com exceção do nodo emissor, atualizando apenas o campo **saltos** que será incrementado em 1. Este serviço está constantemente a ser repetido com um intervalo de tempo de descanso no Servidor, alterando apenas o número de sequência que incrementa em 1.

4.4 Etapa 4 - Construção de Rotas para a Entrega de Dados

Na sequência da etapa anterior, os nodos vão guardar informação relativamente aos custos de cada rota para depois conseguirem perceber qual a melhor rota para o *streaming*. Para tal, após receber a mensagem de *flood*, irá ser criada uma instância da classe Metrica. Esta classe vai ter como atributos:

- nrSaltos : referente ao campo do Pacote;
- duration_ms : a diferença entre o timestamp de envio e o tempo atual, que dá a duração do envio;
- address : Referente ao vizinho que enviou;
- nrSequencia : referente ao campo do Pacote.

Esta classe vai implementar a interface **Comparable<Metrica>**, para definirmos assim qual a melhor rota.

Para tal, vamos primeiro comparar a **duration_ms**. Caso esta se encontre num range de 50 ms irá ser decidido, através dos saltos, que quanto menor, melhor. Caso não se encontre no range, será decidido pela menor **duration_ms**. Assim cada nodo irá ter um map de custos, que será definido da seguinte forma:

- Map<InetAddress, Map<InetAddress, Metrica» costMap;

Para cada Servidor, está associado um map que para cada vizinho fará corresponder a métrica. Assim, para o cálculo do vizinho, que será o melhor fornecedor, calculamos o melhor para cada Servidor. De seguida, calculamos o melhor entre Servidores para saber assim, qual o vizinho e qual o Servidor que irá ser o melhor fornecedor. Este map de custos é atualizado assim no $flood$.

Quando o cliente entra, irá receber os pacotes de $flood$ que permite construir o map de custos, para saber qual o melhor vizinho e servidor para pedir stream. Quando tem essa informação envia o pacote de ativação para o nodo.

Cada nodo terá um serviço *activate* que estará sempre à espera para receber pacotes.

Ao receber, este calcula qual o melhor fornecedor e envia o pedido de ativação para o tal. Este processo repete-se até chegar ao Servidor que ao receber o pedido de ativação irá começar o envio dos pacotes de *streaming*.

4.5 Etapa 4.5 - Método de recuperação de falhas

Para complementar a etapa anterior, adicionamos uma *thread* para cada nodo, responsável por verificar se houve alterações relativamente ao vizinho melhor fornecedor. Neste processo, elimina-se no map de custos, as métricas de vizinhos que não enviem o *flood* há algum tempo, ou de algum servidor que também já não se receba há algum tempo. Para tal, temos um map:

- Map<InetAddress, Instant> lastTime

Que vai associar a um IP a última vez que nos enviou um *flood*. Este IP tanto pode ser um vizinho como um Servidor.

Após retirar as métricas necessárias, é calculado o melhor fornecedor como foi explicado na etapa anterior.

Caso este fornecedor tenha mudado, irá ser enviado um pacote de desativação ao fornecedor antigo e também se envia um pacote de ativação ao novo fornecedor.

4.6 Etapa 5 - Ativação e Teste do Servidor Alternativo

O objetivo desta etapa é forçar a ativação do servidor alternativo de forma transparente ao cliente quando as condições na entrega da *stream* se degradam. Para tal, criamos uma nova classe Servidor Adicional que vai dar *extends* da classe Servidor. A grande diferença entre estas duas classes é que o Servidor será o *bootstrapper*, enquanto o Servidor Adicional fará a leitura do ficheiro de vídeo e o envio destes pacotes, assim como o *flood* e a ativação.

De maneira a tornar este Servidor Adicional o mais parecido com o Servidor, é necessário haver algum tipo de sincronização entre as duas *streams*, para não haver uma diferença significativa para o cliente quando troca de Servidor. Para tal, após o Servidor Adicional anunciar a sua presença ao Servidor irá enviar uma mensagem através do Socket TCP com "Start" e espera metade do período de tempo que a resposta com a lista de vizinhos demorou, para começarem as duas em instantes semelhantes.

5 Testes

Por fim, vamos abordar alguns cenários de teste realizados na topologia, que se encontra na figura a seguir, para testar todas as funcionalidades implementadas de forma sequencial, analisando se o resultado obtido era o esperado.

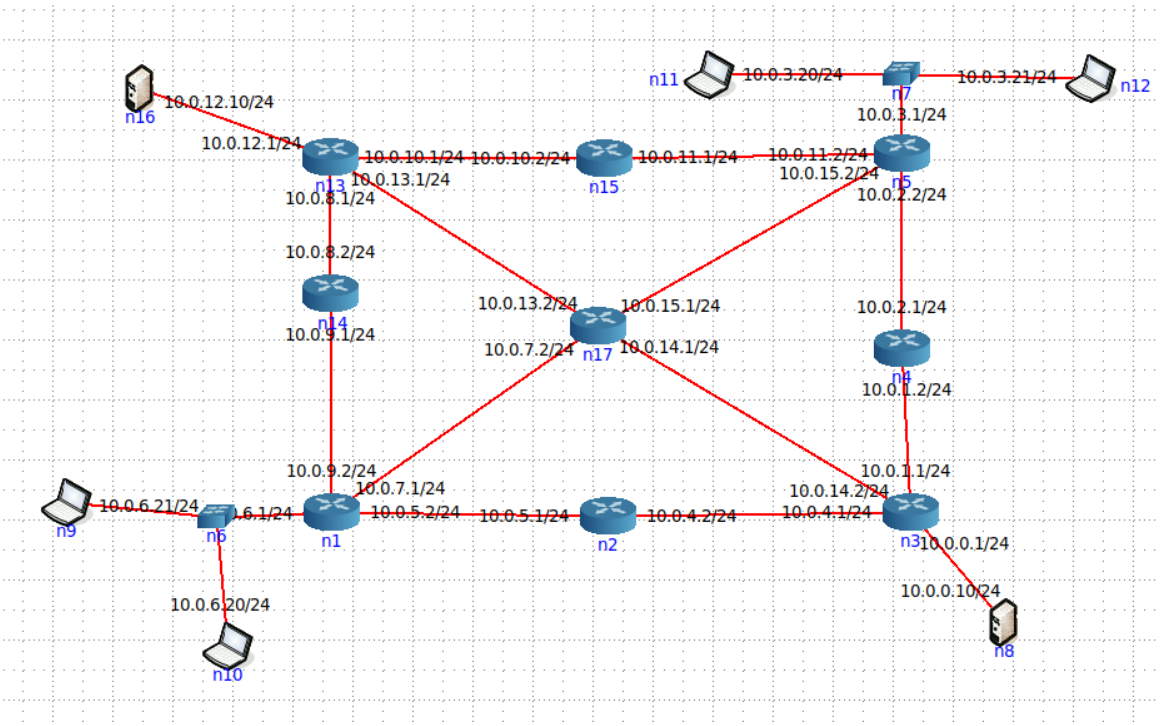


Figure 4: Topologia Underlay

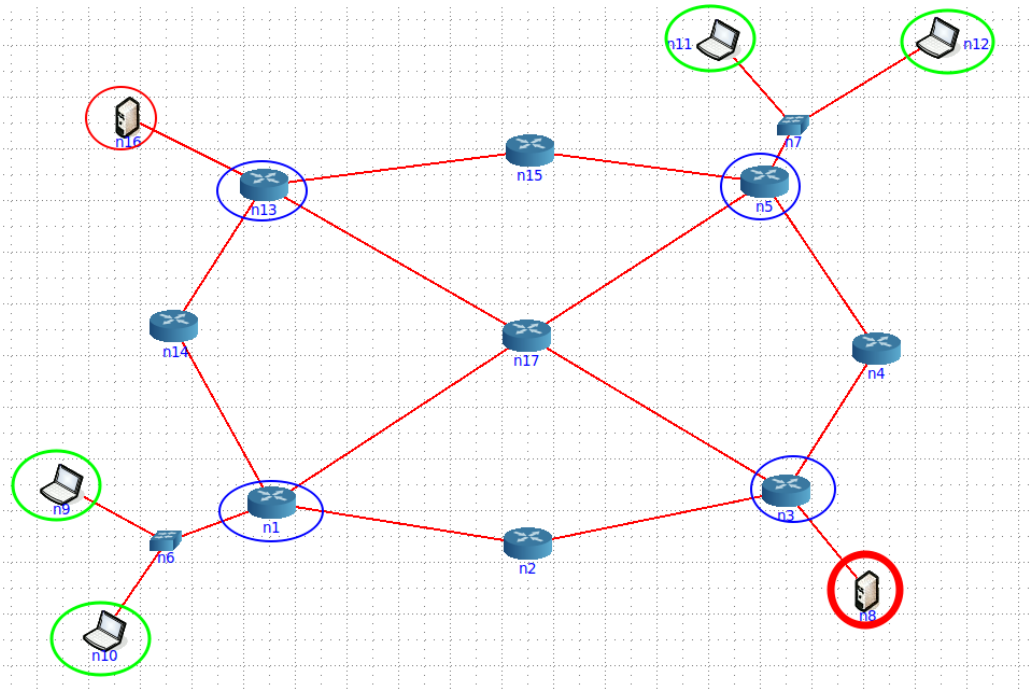


Figure 5: Topologia Overlay

Cenário 1 - Acrescentar um atraso excessivo numa das ligações da rede underlay

Sem atrasos na rede *underlay*, verificámos que a rota para entregar o conteúdo ao cliente n9, vai do Servidor Principal(**n8**), passa pelos nodos **n3**, **n1** e, por fim, no cliente **n9**.

Ao adicionar um atraso entre os nodos n1 e n2 na rede underlay, a rota para entregar o conteúdo ao cliente n9 mudou.

Agora evita passar por aquele atraso adicionado na rede underlay e, por isso, vai do Servidor Adicional (**n16**), passa pelos nodos **n13**, **n1** e, por fim, no cliente **n9**.

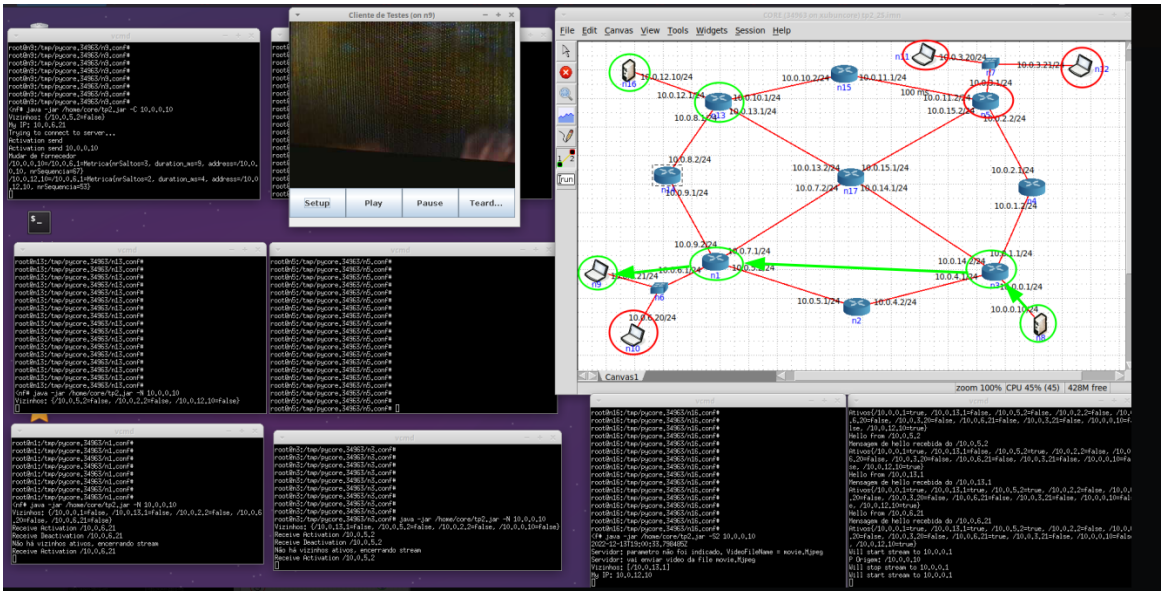


Figure 6: Topologia Overlay sem atraso na rede

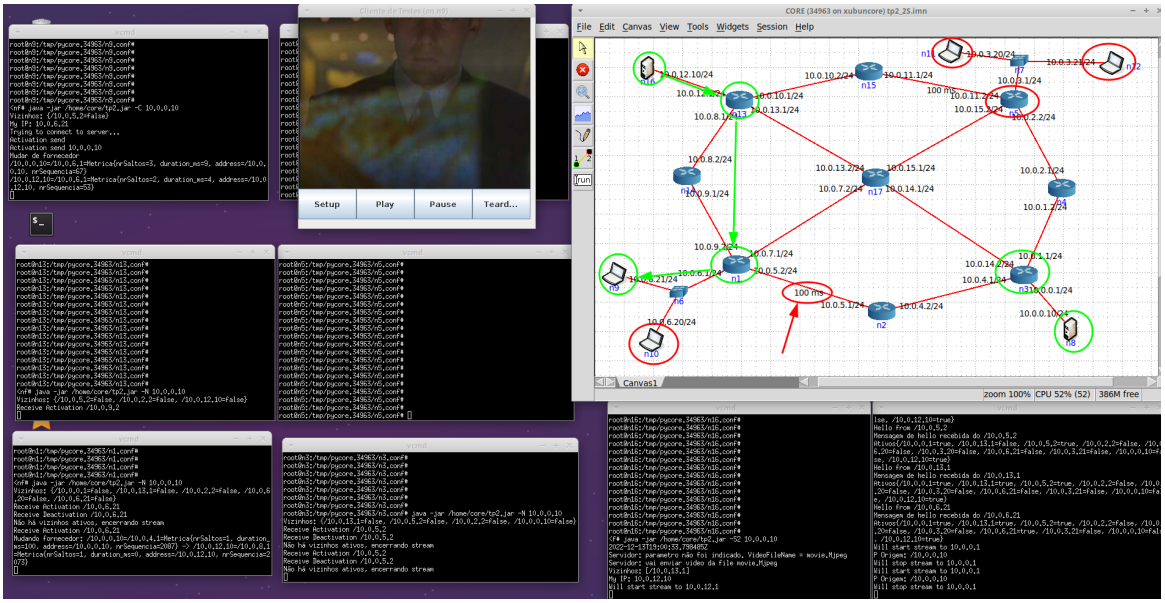


Figure 7: Topologia Overlay com atraso na rede

Cenário 2- Cliente n9 e n11 pedem stream

Quando n9 e n11 entram na rede vão ter de esperar para que o servidor realize um *flood* na rede de forma a que os custos sejam atualizados para receber os clientes. De seguida, os clientes realizam o *activate* para que consigam receber a *stream* de dois servidores diferentes, uma vez que depende do custo.

Na figura que se segue encontra-se realizado no core, onde podemos ver as 2 streams nos 2 clientes distintos, tal como é pretendido.

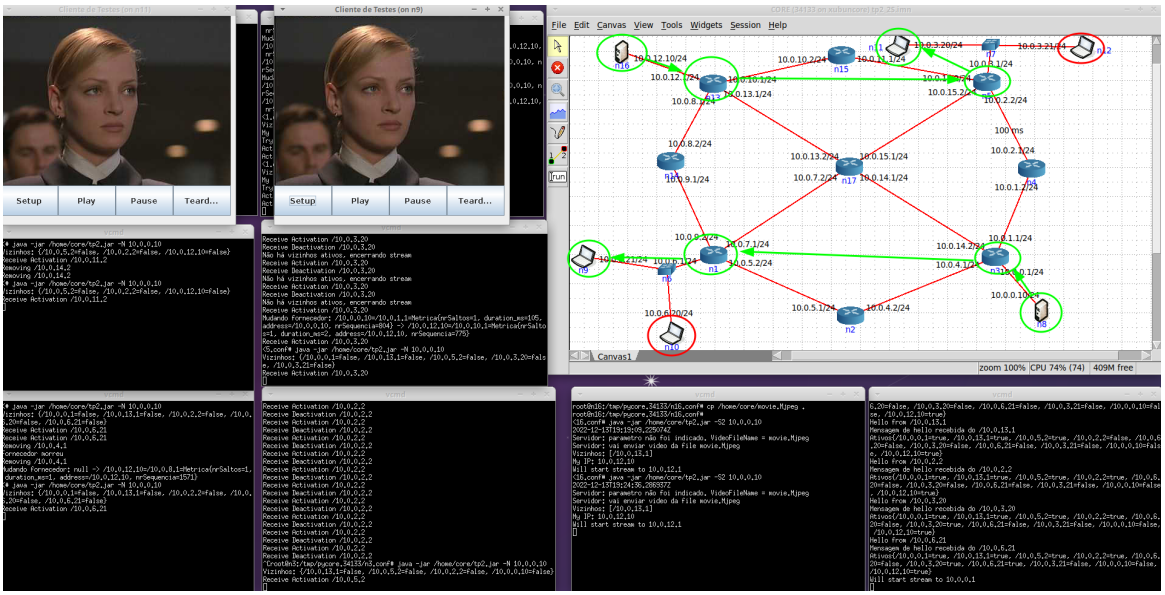


Figure 8: Clientes distintos pedem stream

Cenário 3- Nodo n13 sai da rede

Quando o nodo n13 sai da rede faz com que deixe de ser possível receber a stream proveniente do Servidor Adicional (n16). O cliente n11, que anteriormente recebia a stream pela rota de overlay (n16-n13-n4-n11), passa a deixar de ser possível.

Assim, o conteúdo passa a ser fornecida pelo servidor principal, e como existe um atraso na rede underlay entre os nodos n4 e n5, a melhor rota passa a ser (n8-n3-n1-n5-n11).

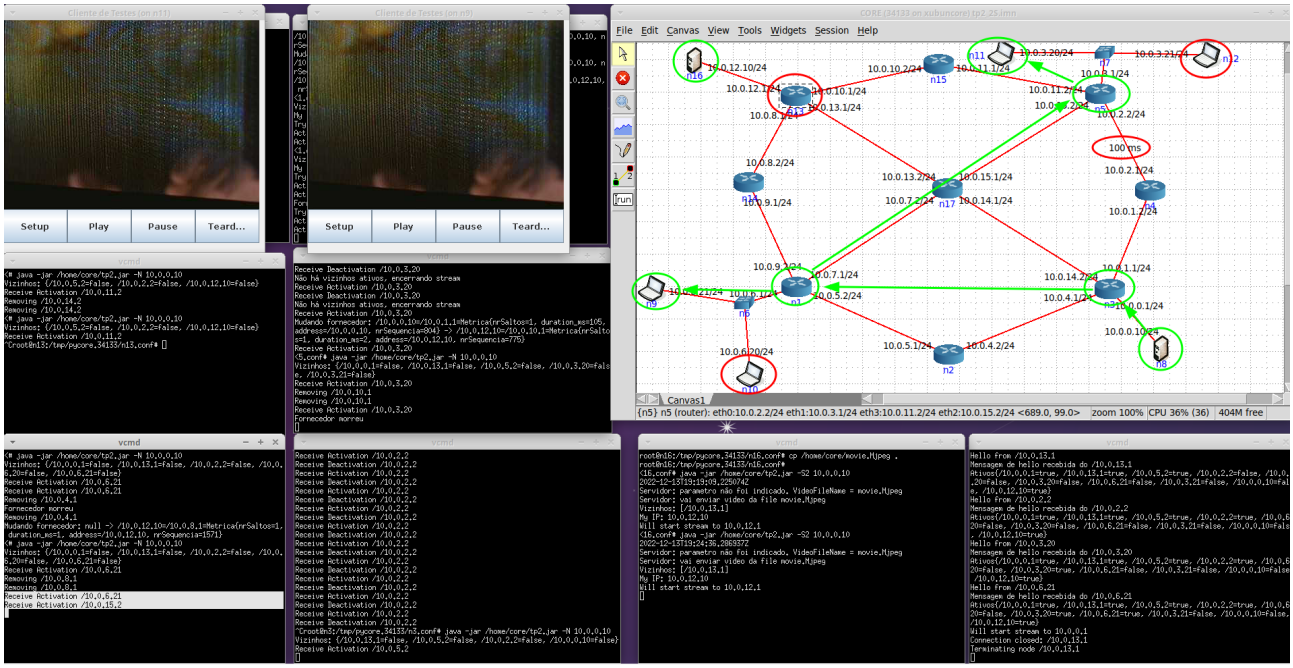


Figure 9: Nodo sai da rede

Cenário 4- Cliente n9 sai da rede

Esta situação acontece quando o n9 deixa de querer a *stream*, por isso este sai da rede, acabando por fechar a janela do vídeo. O n1 deixa de fornecer conteúdo ao n9 por ele ter saído e passa a fornecer apenas ao n5.

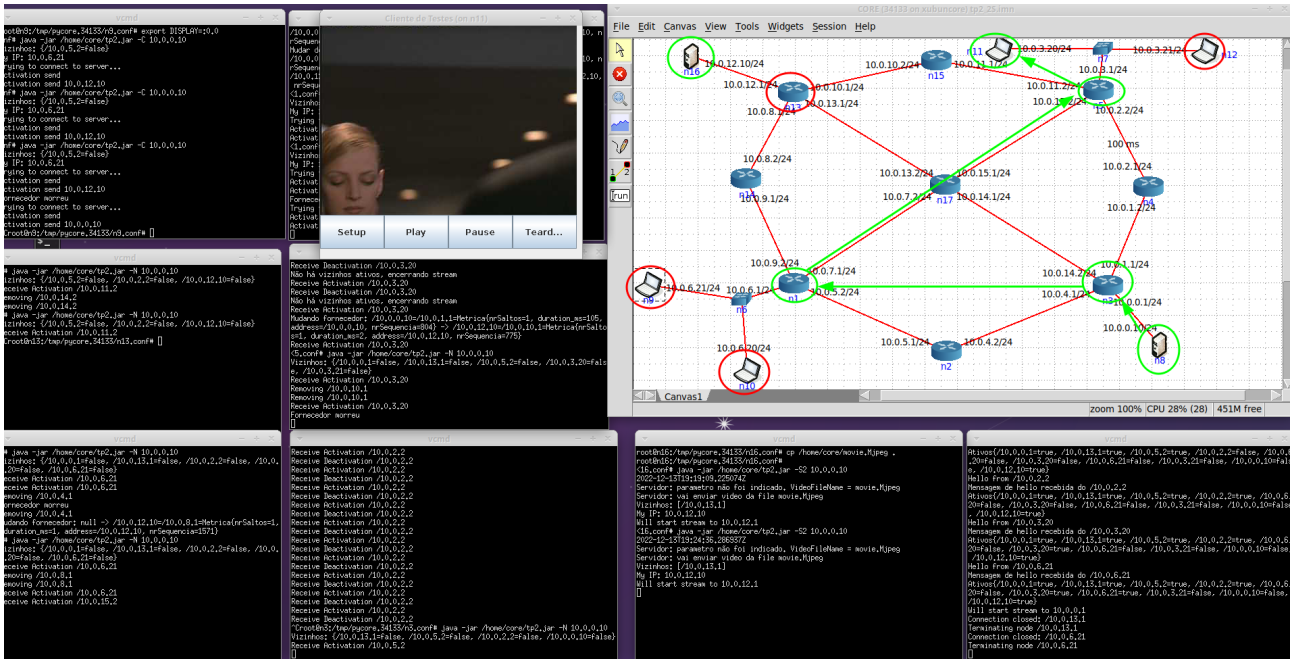


Figure 10: Cliente sai da rede

Cenário 5- Nodo n13 volta a entrar na rede overlay

Quando o n13 volta a entrar na rede, as rotas vão ser novamente atualizadas, pois o custo desta rota é inferior ao da que passa por n3, n1 e n5 até chegar ao cliente. Após atualizar o flood das rotas, o n13 já consegue manter a *stream*.

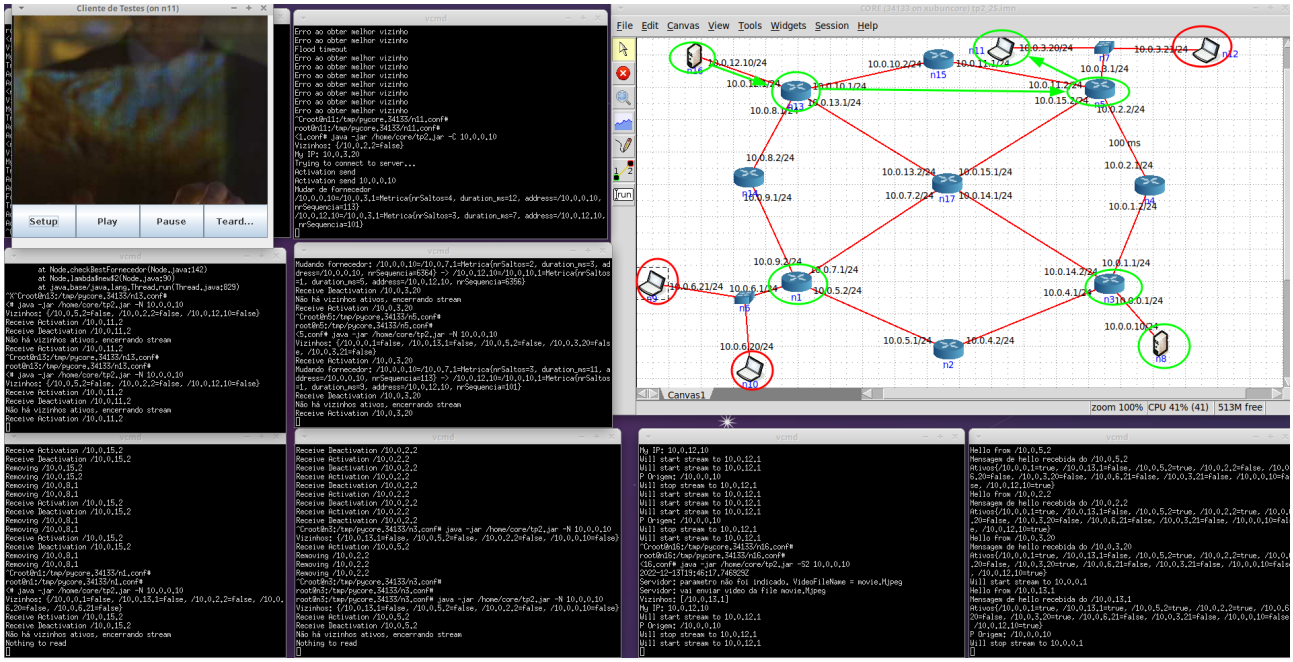


Figure 11: Nodo volta a entrar na rede

6 Conclusões e Trabalho futuro

Através da realização deste trabalho prático tivemos a oportunidade de desenvolver novas aptidões e também consolidar a matéria lecionada nas aulas teóricas relativamente ao *flooding*.

Numa fase inicial, tivemos alguma dificuldade em definir quais as melhores estratégias a ser utilizadas para uma maior eficiência do nosso projeto. No entanto, através de pesquisas e recorrendo à ajuda dos docentes conseguimos superar esta dificuldade e consideramos que o projeto está satisfatório.

Enquanto grupo, conseguimos distribuir bem o trabalho entre todos. Ajudamos-nos mutuamente e, de forma geral, o grupo teve um aproveitamento positivo.

Concluindo, pensamos ter alcançado os objetivos propostos para este trabalho.