ESR - TP2
Serviço Over the Top para entrega de multimédia

Diogo Pires[PG50334], Marco Sampaio[PG47447], and Miguel Martins[PG50655]

Departamento de Informática, Universidade do Minho - Campus de Gualtar, R. da Universidade, 4710-057 Braga
sec@di.uminho.pt
https://www.di.uminho.pt/

Abstract. Projeto final da Unidade Curricular de Engenharia de Serviços em Rede (ESR), cujo principal objetivo passa por conceber um serviço *Over the Top* para entrega de multimédia. Este relatório explora todo o processo, desde a solução conceptual desenvolvida numa primeira fase até à sua implementação. Procura-se também analisar o produto final, tendo em conta a inevitável comparação com os serviços de *streaming* já existentes, e as funcionalidades que os mesmos implementam.

Keywords: *Overlay* · *Underlay* · *Thread* · *Socket* · Topologia · *Streaming* · Servidor · Cliente · Nodo · *Bootstraper*

1 Introdução

Ao longo do último século notou-se um grande desenvolvimento na área da comunicação ponto-a-ponto em tempo real. Neste sentido, o grupo foi desafiado a desenvolver uma solução de um serviço *Over the Top* para entrega de multimédia.

Um serviço multimédia deste género pode usar uma rede *overlay* configurada e gerida para contornar os problemas de congestão e limitação de recursos da rede de suporte, entregando os *media* ao cliente final em tempo real e sem perda de qualidade. Temos alguns exemplos bem conhecidos como a *Netflix* ou o *Holu*.

Deste modo, para responder aos objetivos propostos pela equipa docente, foi desenvolvido um protótipo de entrega de ficheiros de vídeo, enviados pelo servidor para um dado conjunto de N clientes, usando nodos intermédios, formando entre si uma rede de *overlay* aplicacional. Além disso, existência de um controlador *bootstrapper*, permite ao Servidor ser auxiliado no monitorização da topologia.

Neste relatório estão descritos de forma sucinta alguns dos aspetos principais para perceber o que grupo implementou neste projeto: É descrita a arquitetura da solução e o processo de conceptualização que levou à mesma, bem como o protocolo que rege a comunicação entre as várias partes. Apresentam-se também algumas caraterísticas da implementação propriamente dita, e os testes realizados, com os respetivos resultados.

2 Arquitetura da solução

A solução concebida teve por base as estratégias propostas pela equipa docente para as diferentes etapas. Para cada uma das etapas tentamos primeiramente escolher das propostas colocadas quais seriam as mais adequadas para o objetivo final do grupo. Dado que, desde uma fase muito precoce o grupo havia definido que seria interessante conceber uma aplicação dinâmica e com tolerância a falhas, acabamos por optar pelas sugestões que iam mais de encontro a esse objetivo.

Além disso, na definição dos aspetos mais pessoais da solução, nomeadamente a nível protocolar, o grupo tentou sempre manter a par o docente do turno prático, de forma a ter sempre uma opinião externa que permitisse validar as nossas decisões, evitando a implementação de algo inútil ou que pudesse levar a problemas.

O seguinte Diagrama de Pacotes representa o cerne do programa que o grupo concebeu, sendo que o grupo considera que o mesmo consegue dar uma visão bastante intuitiva de como esses pacotes se relacionam na arquitetura final da solução.

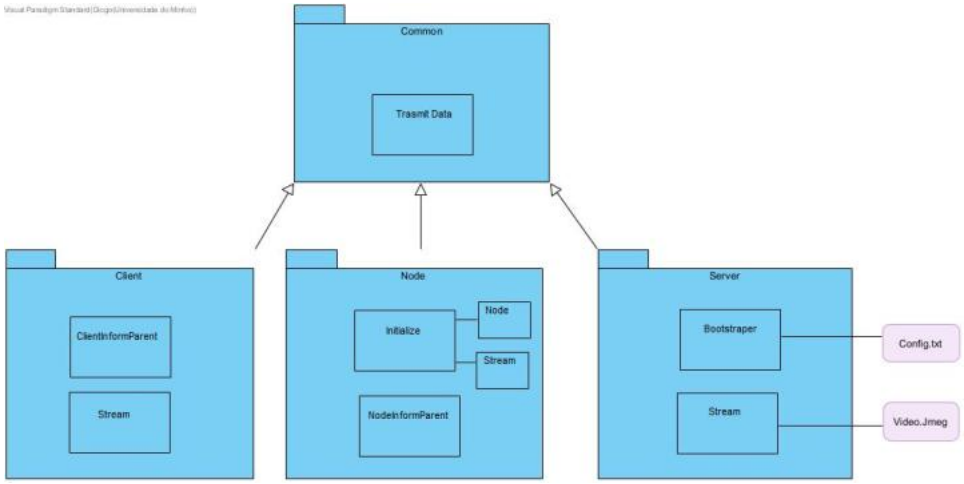

Fig. 1. Diagrama de Pacotes

Dado que este projeto nos foi apresentado como um conjunto de etapas correlacionadas, consideramos interessante apresentar a conceptualização concebida em cada uma dessas etapas, bem como a solução obtida.

2.1 Decisões por etapas

ETAPA 0: Preparação das atividades

Este projeto foi desenhado com a linguagem de programação Java, utiliza o emulador CORE como bancada de teste, está preparado para uma ou mais topologias e opera sob o protocolo de transporte UDP. As rotas definidas entre o Servidor e o Cliente são geradas através de um algortimo baseado no algoritmo de Prim.

ETAPA 1: Construção da topologia *Overlay*

Optamos por escolher a segunda estratégia apresentada no enunciado (abordagem baseada num controlador), visto que esta é não só uma solução mais dinâmica, como também facilita bastante o processo de testagem, pois os nodos ao conectarem-se não necessitam de indicar os seus vizinhos.

Assim sendo, ao executar um nodo, o programa recebe como nodo de contacto o *bootstrapper* e consequencialmente envia-lhe uma mensagem de conexão(*"hello message"*), o bootstrapper reencaminha-lhe de seguida os seus vizinhos ativos. A partir daqui o nodo em questão comunica com os seus nodos vizinhos, desta forma retiramos o valor do *delay* entre nodos e número de saltos do nodo até ao servidor, até que toda essa informação chega ao servidor através da rede, ficando este com conhecimento total das conexões de nodos e atualizando a rede sempre com os melhores caminhos entre um nodo e o servidor. De seguida, o *bootstrapper* comunica ao resto da topologia a árvore com os melhores caminhos.

ETAPA 2: Serviço de *Streaming Overlay*

O serviço de *streaming* é enviado pelo servidor, reencaminhado pelos nodos, e apresentado pelos clientes. Quando um cliente se junta à rede, começa a enviar mensagens a pedir a *stream*, e essas mensagens são reencaminhadas dos filhos para os pais na árvore, até chegar ao servidor. Por outro lado, o servidor só envia a *stream* se receber mensagens a informá-lo que algum cliente está ligado. Quando existe pelo menos um cliente ligado, o servidor envia para o primeiro nodo o conteúdo da *stream*. Depois, cada nodo reencaminha o conteúdo apenas para os filhos interessados, isto é, os que enviaram pedidos de *stream*.

Quanto às possibilidades de *media*, dado que esse foi um ponto da implementação construído numa fase mais próxima da data de entrega, o grupo acabou por não escolher a opção mais dinâmica, optando pela estratégia 1 desta etapa, adaptando o código fornecido pela equipa docente.

ETAPA 3: Monitorização da Rede *Overlay*

A monitorização da rede é conseguida através de mensagens do tipo *StillAlives* entre pais e filhos. Quando um nodo deixa de receber *StillAlives* do nodo

pai, envia mensagem ao *bootstrapper* avisando que nodo pai ficou desconectado. De seguida, o *bootstrapper* atualiza a árvore, e reenvia a árvore de conexões aos nodos da rede ativos.

ETAPA 4: Construção de Rotas para a Entrega de Dados

Para a construção de rotas, nodos utilizámos o conjunto de nodos ativos. A comunicação da árvore para os filhos é em formato XML, e essa mensagem é decomposta em cada nodo. Assim, cada nodo só envia para os filhos a parte que lhes corresponde. Utilizamos o formato XML por ser mais fácil separar por grupos de nodos. Com estas informações, quando o nodo recebe a stream, já sabe para quem deve reenviar o seu conteúdo, filtrando pelos filhos interessados. Para além disso, também envia para os seus filhos mensagens do tipo *StillAlive*.

ETAPA 5: Ativação e Teste do Servidor Alternativo

O servidor alternativo recebe informações do servidor principal, e recebe pedidos dos nodos quando a ligação entre eles e o servidor perde qualidade. Os métodos utilizados por este servidor encontram-se implementados na classe do servidor principal, pois possuem muitas semelhanças. Por ainda se encontrar em fase de testes, não o consideramos como totalmente implementado.

EXTRA: Definição do método de recuperação de falhas

As falhas que analisamos podem ter duas origens, num nó, ou no servidor. Quando um nodo falha, o seu filho deixa de receber mensagens do tipo *StillAlive*, e então envia mensagem ao *bootstrapper* a dizer que o seu pai se desconectou da rede. De seguida, o *bootstrapper* recalcula a árvore dos melhores caminhos, e envia para a rede a árvore dos caminhos atualizada. Caso o nodo que o liga ao servidor principal era o único, então envia mensagem ao secundário a pedir conexão.

3 Especificação do(s) protocolo(s)

3.1 Formato das mensagens protocolares

No tratamento do protocolo utilizamos a Classe *MessageAndType* para lidar com os pacotes enviados/recebidos, esta classe possui o inteiro *msgType* que distingue todo o tipo de mensagens abaixo descritas:

- **Hello Message** - Constituída pelo seu tipo de mensagem, ip e porta de destino.
- **Resposta ao Hello Message** - Constituída com um array[bytes] com os vizinhos do nodo que enviou a mensagem, ip e porta de destino.

- **Timestamp Message** - Constituída pelo seu tipo de mensagem, o instante em que foi enviada, ip e porta de destino.
- **Connection Message** - Constituída por uma array[bytes] com uma conexeção entre nodos, ip e porta de destino.
- **Still Alive Message** - Constituída pelo seu tipo de mensagem, o tempo da criação desta mensagem, ip e porta de destino.
- **Want Stream Message** - Constituída pelo seu tipo de mensagem e porta de utilizada para *streaming*.
- **Stream - RTP packet Message** - Constuído por campos usualmente associados a um Pacote, importante de salientar o uso do *TimeStamp* que nos dita o instante do pacote associado ao vídeo enviado e o *Sequence Number* que nos permite o controlo da chegada de Pacotes.

3.2 Interações

De forma a facilitar a interpretação do nosso programa dividimos as interações por pares de entidades, como apresentado abaixo, segue também um diagrama de fluxos que demonstra toda a interação desde o cliente até ao servidor.

3.3 Nodo - Servidor

- **Hello Message** - Mensagem enviada pelo nodo inicializado ao *boot* como modo de inicializar o nodo na rede.
- **Timestamp Message** - Mensagem enviada pelo nodo ao Servidor com o **CurrentTime** em que é inicializado, o servidor é atualizado com esta conexão entre eles com o respetivo *delay*.
- **Connection Message** - Mensagem que é enviada por um nodo com a conexão entre dois nodos quaisquer que vai ser guardada na rede. Surge após o *TimeStamp*
- **Want Stream Message** - Mensagem enviada pelo Cliente para o seu pai, e reenviada pela rede até ao servidor.

3.4 Nodo - Nodo

- **Timestamp Message** - Mensagem enviada pelo nodo a cada vizinho com o **CurrentTime** em que é enviada, o vizinho calcula o *delay* desta mensagem e envia pela rede esta conexão.
- **Connection Message** - Mensagem que é enviada por um nodo com a conexão entre dois nodos quaisquer que vai ser enviada pela rede.
- **Still Alive Message** - Mensagem enviada repetitivamente por um nodo a indicar de que ainda está ativo. Esta mensagem só deixa de ser enviada quando o nodo se desliga.
- **Want Stream Message** - Mensagem recebida do Cliente para que um nodo a comunique ao servidor nodo-a-nodo.

3.5 Servidor - Nodo

- **Resposta ao Hello Message** - Mensagem enviada pelo *boot* ao nodo com os seus vizinhos ativos na rede, para que este comece a ter conhecimento da rede.
- **Resposta ao Connection Message** - O servidor recalcula a árvore, os nodos recebem o *xml* com os seus novos filhos.
- **Resposta ao Want Stream Message** - O servidor começa a enviar a *stream*.
- **Stream - RTP packet Message** - nao sei onde colocar

3.6 Cliente - Nodo

- **Want Stream Message** - Mensagem enviada pelo Cliente para que um nodo a comunique pela rede até ao servidor.

.

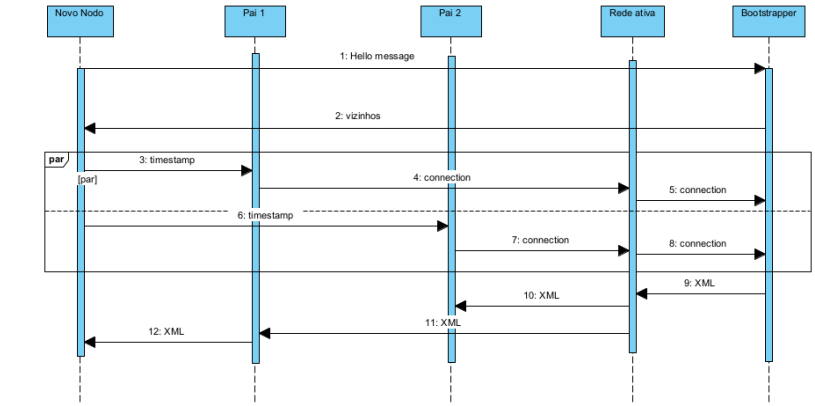

Fig. 2. Diagrama de Fluxo

4 Implementação

4.1 Nodo

Para garantir melhor qualidade da *stream*, a sua transmissão não é interrompida por mensagens de controlo de rede. Deste modo a implementação do nodo pode ser dividida em 2 fases:

- **Inicialização do Nodo** - Esta fase retrata o modo de arranque do nodo, que recebe como argumentos, o endereço IP do *bootstrapper* e o seu respetivo IP. Nesta primeira etapa, o nodo comunica unicamente com o *bootstrapper* , de modo a que, a rede seja atualizada com este novo nodo ativo. Para além disso, neste ponto, o nodo inicializado recebe o conjunto de vizinhos que estão ativos na rede.
- **Comunicação Nodo-a-Nodo** - Depois de inicializado o nodo começa a comunicação Filho-Pai com constantes trocas de mensagens facilitando o controlo dos nodos na rede, isto é, no caso de algum nodo se desconectar a rede é rapidamente atualizada. Neste segmento, auxiliado pela *Thread NodeInformParent*, trata-se também do *handling* de todo o tipo de mensagens que o nodo poderá receber por parte de um nodo vizinho e, consequentemente de como processar essa informação.

4.2 Cliente

Na implementação do cliente, decidimos armazenar alguns frames antes de começar a apresentar o conteúdo. Assim, possíveis falhas na transmissão podem ser ocultadas pelos frames que ainda estão em buffer. Para além disso, criámos dois modos de apresentar a stream quando o utilizador despausa o vídeo: continua onde parou, ou volta para o ponto atual da stream, ignorando o que se passou enquanto pausou o vídeo. Por outro lado, decidimos que cada cliente só se pode conectar a um nodo, para que menos mensagens de *wantStream* circulem na rede, em vez de cada cliente a mandar uma.

- **Thread da conexão** - Utilizamos um Timer, que acorda em intervalos de tempos iguais, e envia uma mensagem a pedir stream para o nodo pai.
- **Thread que recebe mensagens** - Na classe *StreamWindow*, o socket do cliente recebe os vários *frames* recebidos, e guarda-os numa queue que é FIFO (*First In, First Out*). Decidimos usar uma estrutura FIFO porque ao apresentar os vídeos, vai sempre buscar o *frame* mais antigo. Se a stream tiver sido pausada pelo utilizador, e estivermos no modo de voltar para a stream atual, trocamos os frames mais antigos pelos novos.
- **Thread que apresenta a *stream*** - Esta thread apenas retira da queue em intervalos de tempo constantes (igual ao frame rate do vídeo), e atualiza a janela de apresentação. Também é responsável por gerir os botões de pausa e continuar a *stream*.

4.3 Servidor

- ***Thread* da conexão** - Esta thread tem a missão de ir atualizando a rede sempre que um nodo se conecta/desconecta.
- ***Thread* da *Stream*** - Esta thread é responsável pela leitura do vídeo a partir do ficheiro, por *frames*, quando há clientes interessados. Assim, de X em X tempo, verifica se há clientes interessados, e se não há, fica à espera, com um *sleep*. Quando há clientes interessados, envia a stream em loop para o filho, que a reencaminha.

4.4 *Bootstrapper*

O *Bootstrapper* é o elemento que permite ao Servidor possuir conhecimento da topologia completa, bem como dos nodos que se encontram ativos e o melhor caminho para cada um deles, tendo como ponto de partida o servidor.

Ele começa, inicialmente por efetuar a leitura do ficheiro de configuração e guardar a topologia completa do *overlay*.

Seguidamente, à medida que um novo elemento (nodo, cliente ou servidor alternativo) se vai juntando à topologia, o *Bootstrapper* vai atualizando uma estrutura (sub-topologia) que possui, que guarda os elementos da topologia ativos e com este calcula a árvore dos melhores caminhos, utilizando o Algoritmo de *Primm*.

Ficheiro de configuração

Na primeira etapa do desenvolvimento da solução, foi necessária a construção da topologia *overlay*. Definimos como estratégia a construção de um ficheiro de configuração com a informação de quem é vizinho de quem na rede, seguindo a estrutura a seguir apresentada como exemplo:

```
s1: 10.0.0.10 ; n1
n1: 10.0.0.1 ; s1 , n2 , n3
n2: 10.0.0.2 ; n1, n3, c2, n5
n3: 10.0.0.3 ; n1, n2, n4, n5
n4: 10.0.0.4 ; n3, n5
n5: 10.0.0.5 ; n3, n4, c1, n2
c1: 10.0.0.6 ; n5
c2: 10.0.0.7 ; n2
```

Fig. 3. Ficheiro de configuração da topologia *overlay*

Como se pode verificar, em cada linha está representado o nome do nodo em questão, o seu respetivo endereço IP e os nodos a quem se liga na topologia. A partir deste ficheiro o *bootstrapper* irá ter conhecimento de toda a rede, guardando em memória os dados apresentados neste mesmo ficheiro.

Representação da topologia

A topologia é representada por 3 estruturas principais, que permitem ao Servidor, usando o *Bootstraper* como intermediário, ter um conhecimento bastante completo da rede:

– **Topologia completa**: Lida diretamente do ficheiro de configuração da rede, contém todos os nodos ativos e inativos que a rede *overlay* pode conter - esta é a topologia usada como teste no *CORE*;

– **Topologia dos nodos ativos** - Representam um subconjunto da topologia anterior, são os nodos inicializados e posteriormente usados para calcular a árvore dos melhores caminhos;

– **Árvore dos melhores caminhos**: Relaciona os nodos ativos entre si, num grafo do tipo árvore. A raiz dessa árvore é o servidor, e as folhas são os clientes. É utilizada para definir o caminho mais consistente.

Assumindo, que todos os nodos incluídos no ficheiro de configuração anterior já estão ativos, a figura 4 representa não só a topologia completa, como também a topologia dos nodos ativos.

De forma a ilustrar como o algoritmo de *Primm* funcionaria para o caso em questão, optamos por colocar valores representativos do peso das arestas, sendo que a figura 5 já representa uma árvore com os melhores caminhos.

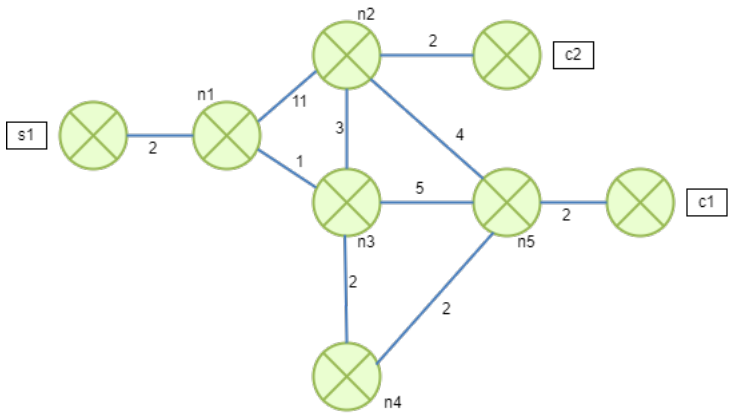

Fig. 4. Topologia dos nodos ativos com representação abstrata de distâncias.

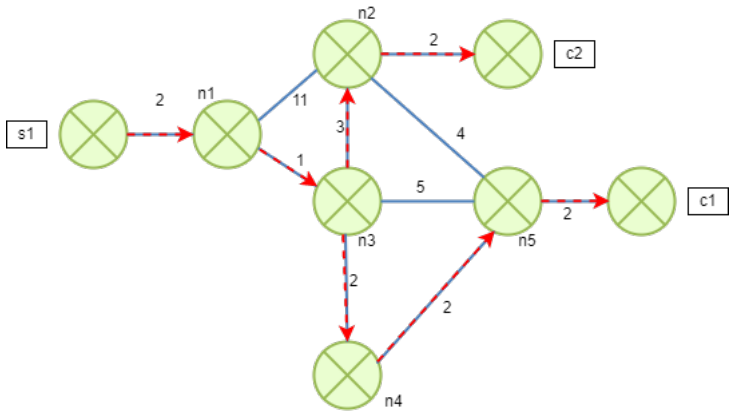

Fig. 5. Topologia com os melhores caminhos, calculados recorrendo ao Algoritmo de *Primm*.

Nesta última figura, podemos observar setas de um nodo para outro. O grupo acabou por ir usando o conceito de **pai** e **filho**. Esta é uma relação entre dois nodos da árvore, sendo que o pai é o nodo mais próximo do servidor, e o filho o mais afastado, cujo caminho até ao servidor passa inevitavelmente pelo pai.

4.5 Conhecimento da rede ao nível do *Bootstrapper*

- Reação a falhas (um nodo desligar-se) - O nodo filho manda mensagem ao *bootstrapper* a dizer que perdeu o pai. O *bootstrapper* recalcula a árvore dos melhores caminhos e reenvia;
- Novos nodos - É enviada uma mensagem ao bootstrapper. e todo o processo está já descrito na parte das interações;
- Recálculo da melhor árvore - É utilizado o algoritmo de *Primm*, sempre que surge ou desaparece um nodo na tipologia, ou quando uma conexão altera muito o seu tempo de ligação;

Por outro lado, o servidor alternativo tem apenas conhecimento dos nodos ativos, e quando for preciso, calcula a árvore dos melhores caminhos. Desta forma, prevenimos que o cálculo custoso da árvore seja realizado vezes desnecessárias.

4.6 Bibliotecas de funções

(Explicar a utilidade a nivel da aplicação)

- *XML Parser* - Para que os ficheiros processem mais depressa o ficheiro enviado pelo servidor sobre a árvore dos melhores caminhos;
- *Datagram Socket* - Para enviar informações entre as diferentes entidades da topologia;

- *Thread* - Para permitir executar operações em paralelo;
- Timer e timerTask - Também para permitir executar operações, mas mais simples, em paralelo, e separadas por intervalos de tempo iguais;
- *JFrame, JButton, JPanel (Java Swing* - Para apresentar a *stream* ao utilizador.
- *RTP packet fornecido pelos professores* - Utilizado para enviar a stream ao longo da topologia.
- *etc*

5 Testes e resultados

Antes de dar por concluída esta exposição do que o grupo fez no projeto, é importante demonstrar os resultados obtidos testando no CORE, visto que o mesmo é uma ferramenta bastante útil para simular redes virtuais.

Além de efetuar testes nas 2 topologias sugeridas no enunciado, o grupo também achou pertinente demonstrar como se comporta o programa quando uma rede maior é simulada (cenário extra).

5.1 Cenário 1

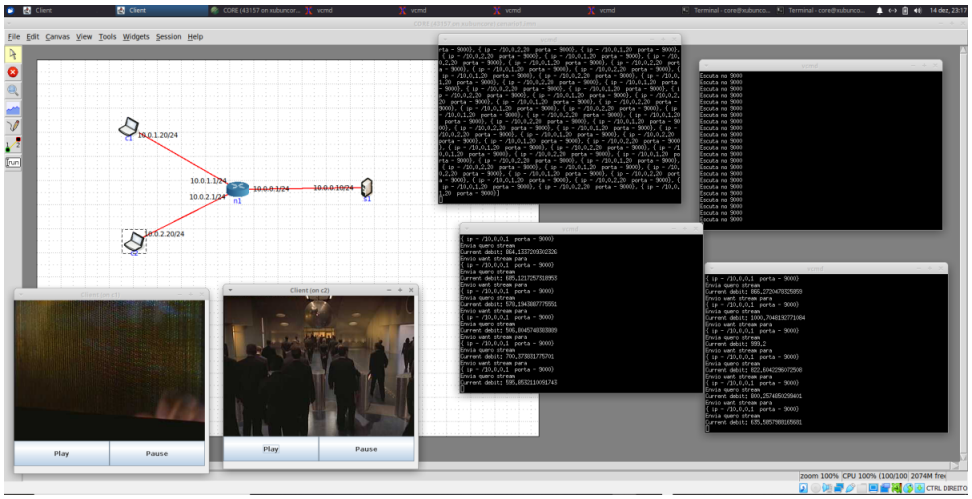

Fig. 6. Cenário 1: Overlay com 4 nós(um servidor, dois clientes e um nodo intermédio).

5.2 Cenário 2

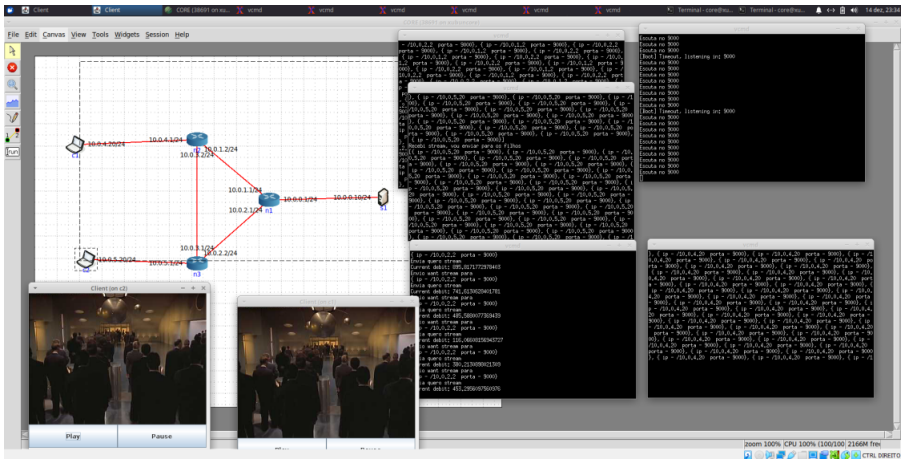

Fig. 7. Cenário 1: Overlay com 6 nós(um servidor, dois clientes e três nodos intermédio).

6 Conclusões e trabalho futuro

Em modo de conclusão tiramos algumas ilações, desde já, alguns pontos que gostaríamos de ter implementado e que deixamos aqui como trabalho futuro:

- **Comunicação da árvore aos nodos ativos por parte do *bootstrapper*** . Neste caso, o *bootstrapper* envia a árvore completa aos nodos, uma solução mais eficiente seria um protocolo que enviasse apenas a aresta alterada.
- **Existir mais que um servidor**, e eles decidirem entre si, sem qualquer intervenção do utilizador, qual seria o principal. Essa escolha podia ter dois critérios, o *minmax* dos custos da ligação (maximizando a qualidade da stream), ou um mais fácil de implementar, comparando os ips. Desta última forma, não temos em conta a qualidade da transmissão, mas pelo menos iriam alternar entre si caso um falhasse, aumentando a tolerância a falhas.
- **Ficheiro de logs**;
- **Ser possível transmitir mais que uma stream**. Para isso, a mensagem de *wantStream* teria também qual o conteúdo, e cada nodo armazenaria os filhos interessados e agrupados por conteúdo pretendido. Por outro lado, o servidor enviaria apenas as *streams* que tivessem clientes interessados, e no pacote RTP o campo ssrc teria qual a *stream* que contém. Para esta funcionalidade funcionar, seria preciso que o servidor comunicasse aos clientes quais as streams disponíveis, e o cliente responderia qual era a pretendida.

- **Tratamento do pacote RTP** - No protocolo definido pela equipa todas as mensagens são verificadas pelo tipo, por outro lado, as mensagens RTP não seguem este raciocínio. Caso o nosso programa fosse para *deployment*, esta verificação teria de ser alterada.
- **Alterar forma de leitura do ficheiro**.

Assim sendo, o grupo considera-se satisfeito com o trabalho realizado, tendo implemenatado parcialmente todos requisitos presentes no enunciado.

References

1. Kurose, J. (2016). Computer Networking: A Top-Down Approach (7th edition.). Pearson.
2. Exemplo de código do livro anterior: https://marco.uminho.pt/disciplinas/ESR/ProgEx.zip
3. Prim's Algorithm from programiz: https://www.programiz.com/dsa/prim-algorithm