

Serviço *Over the Top* para entrega de multimédia

Ivo Baixo, José Magalhães e Paulo R. Pereira

{pg47271, pg47355, pg47554}@alunos.uminho.pt

PL28

MEI, Engenharia de Serviços em Rede, Universidade do Minho, Portugal

22 de dezembro de 2021

Resumo. Conceção de um protótipo *Over the Top* (OTT) de entrega de áudio/vídeo/texto com requisitos de tempo real, a partir de um servidor de conteúdos para um conjunto de N clientes. Adequação do protótipo de modo a promover a eficiência e otimização de recursos para melhor qualidade de experiência do utilizador. Serviço OTT assente em cima do protocolo de transporte UDP.

1. Introdução

No âmbito da unidade curricular de Engenharia de Serviços em Rede foi proposto desenvolver um serviço *Over the Top* (OTT) para entrega de multimédia. Este, desenhado sobre a camada aplicacional, tem como objetivo principal usar uma rede *overlay* aplicacional devidamente configurada e gerida de modo a contornar os problemas de congestão e limitação de recursos da rede de suporte, garantindo assim uma entrega de conteúdos em tempo real e sem perda de qualidade aos diversos clientes finais.

A criação deste serviço assenta no facto de um conjunto de nós poder ser usado no reenvio de dados, como intermediários, formando entre si uma rede de *overlay* aplicacional pelo que, toda a comunicação interna desta funciona sobre UDP e está de acordo com dois protocolos (*OTTpacket* e *RTPpacket*), desenhados especificamente para propósitos distintos, como será visto mais à frente.

Usando o CORE como plataforma de testes, bem como várias topologias de teste, foi, para cada uma destas, concebido um *overlay* aplicacional sobre a respetiva infraestrutura IP. Este serviço OTT foi idealizado de forma ponderada e cirúrgica pois, visto estarmos a lidar com infraestruturas IP que podem conter N componentes, entre eles *Switch*, *Router* e Clientes, a sua constituição e organização é determinante para a qualidade de serviço que este é capaz de suportar.

2. Arquitetura da solução

O sistema encontra-se dividido essencialmente em três partes: servidor, nós *overlay* e cliente.

O servidor, por um lado, comunica com os nós intermédios para lhes enviar as informações dos seus vizinhos, bem como a *stream*. Os nós *overlay*, por outro lado, comunicam exaustivamente entre si de forma a terem noção do estado dos seus vizinhos e trocarem todas as informações necessárias - fazem a ponte entre o servidor e o cliente. Já os clientes, a terceira parte desta solução, comunicam diretamente com os nós intermédios para consumirem uma determinada *stream* multimédia.

Desta forma temos uma arquitetura de *multicasting*, onde o servidor apenas precisa de enviar um único fluxo *stream* por nó adjacente, independentemente do número de clientes, respondendo assim de forma positiva à questão da escalabilidade. Além disto, apresentando esta arquitetura, o servidor também não necessita de receber pedidos por cada cliente que peça a *stream*, o que evita comunicações excessivas, primando a eficiência.

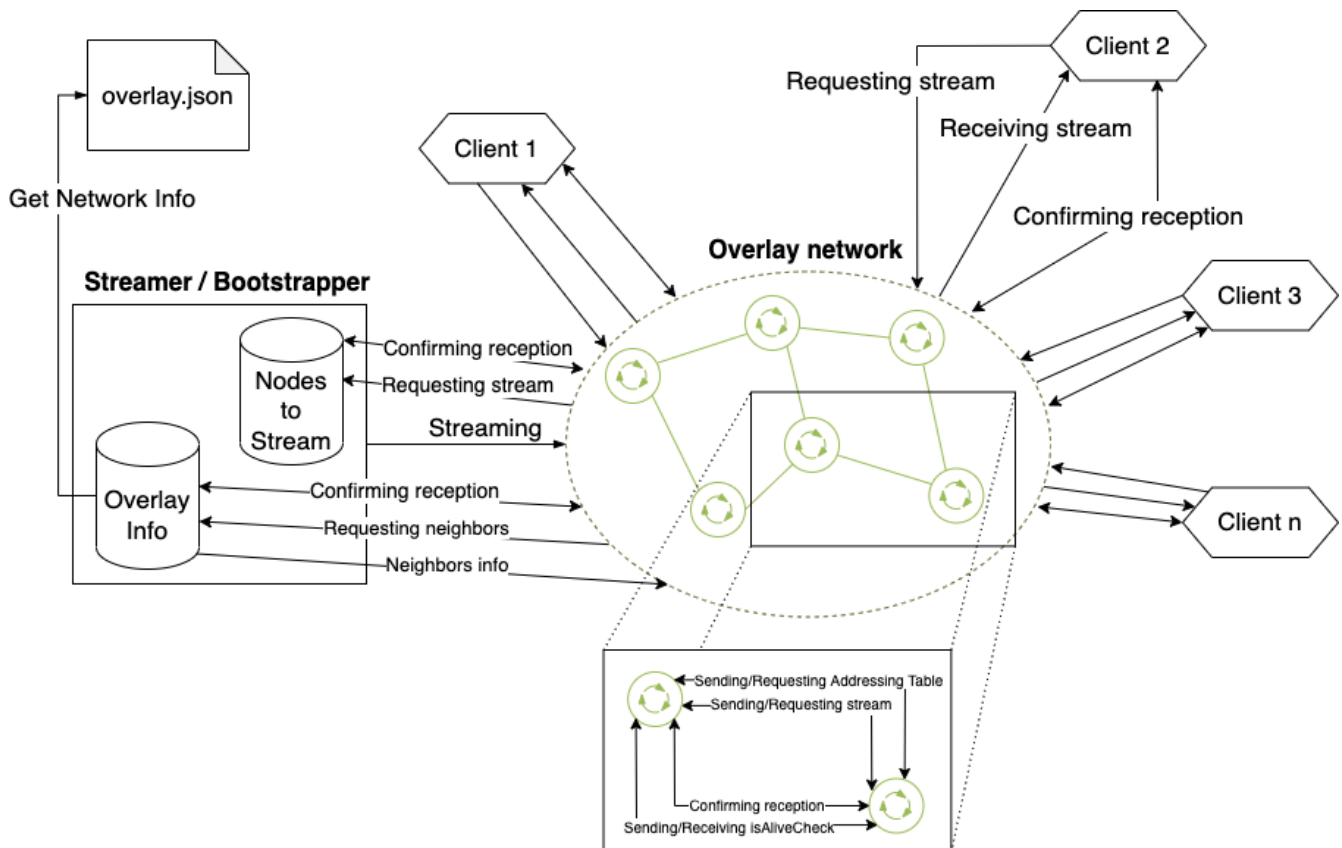


Figura 1: Arquitetura da Solução

3. Especificação dos protocolos

De modo a garantir o bom funcionamento de todas as funcionalidade desejadas, foi necessário idealizar dois tipos de protocolos, ambos a funcionar sobre UDP: *OttPacket* e *RTPpacket*. Estes possuem funções distintas na solução: o primeiro corresponde a um protocolo aplicacional de controlo, sendo que o segundo é responsável por toda a componente protocolar de *streaming*.

3.1. Protocolo aplicacional de controlo - *OttPacket*

3.1.1. Formato das mensagens protocolares

Desenhado de origem, este protocolo tem como missão suportar todo o fluxo de mensagens e comunicações que acontece dentro da rede *overlay*, desde pedidos de *stream* até envio de tabelas de endereçamento. Este tem a seguinte constituição padrão:

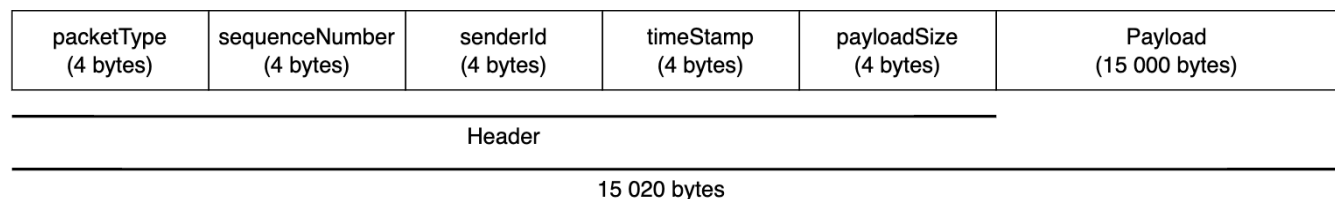


Figura 2: *OttPacket* - mensagem protocolar

Analisando ao pormenor cada campo:

- **Header:**
 - **packetType:** Tipo de pacote. Este pode tomar diferentes significados, consoante o seu tipo, como veremos posteriormente;
 - **sequenceNumber:** Usado para detetar perdas. Os números de sequência incrementam numa unidade por cada pacote RTP transmitido;
 - **senderId:** Corresponde ao ID do nó responsável pelo envio do pacote em questão;
 - **timeStamp:** Usado para colocar os pacotes de áudio e vídeo de entrada na ordem de tempo correta (*playout delay compensation*);
 - **payloadSize:** Tamanho, em *bytes*, do campo *Payload*.
- **Payload:** Campo responsável por guardar a informação, em *bytes*, que se pretende enviar.

3.1.2. Interações

O campo *packetType*, como já foi dito, é responsável por armazenar o "motivo" do pacote em questão. Este pode tomar vários valores, cada um com um propósito diferente, e podem ser interpretados sabendo também que tipo de "percurso" estão a fazer, conforme se verifica de seguida:

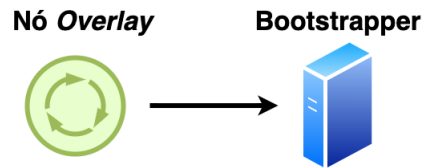


Figura 3: Sentido do envio do pacote de Tipo 0

- **Tipo 0:** Estamos perante um pedido de conhecimento de vizinhos. Quando um nó *overlay* se liga à rede, este envia um pedido do tipo **0** ao *Bootstrapper* a pedir para conhecer os seus vizinhos;

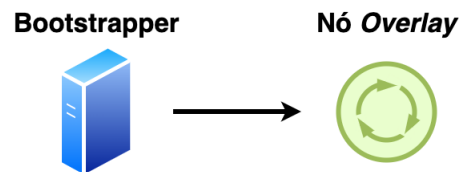


Figura 4: Sentido do envio do pacote de Tipo 1

- **Tipo 1:** Corresponde ao envio da respetiva tabela de vizinhos, por parte do *Bootstrapper*, aos nó(s) que efetuaram o pedido;

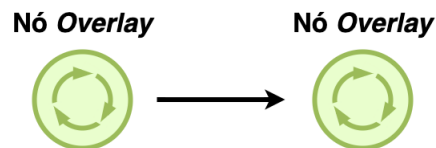


Figura 5: Sentido do envio dos pacotes dos Tipos 2,3,4,5,6 e 7

- **Tipo 2:** Pacote que contém a tabela de endereçamento do vizinho, enviada pelo próprio, ao nó em questão a fim de ser atualizada;
- **Tipo 3:** Estamos perante um pedido da tabela de endereçamento de um determinado vizinho;
- **Tipo 4:** Usado para verificar se um determinado nó vizinho está vivo;

- **Tipo 5:** Confirmação de que o nó em questão está vivo;
- **Tipo 6:** Pedido de *stream* de um determinado nó aos seus vizinhos;
- **Tipo 7:** Usado como confirmação de que um determinado pacote foi recebido pelo nó em questão.

3.2. Protocolo *streaming* - *RTPpacket*

3.2.1. Formato das mensagens protocolares

Adaptado do código fornecido pela equipa docente, este protocolo tem uma única função: o envio da *stream* propriamente dita, através dos respetivos *frames*. Este tem a seguinte constituição padrão:

packetType (4 bytes)	sequenceNumber (4 bytes)	senderId (4 bytes)	timeStamp (4 bytes)	payloadSize (4 bytes)	Padding (4 bytes)	Extension (4 bytes)	CC (4 bytes)	Marker (4 bytes)	Srcr (4 bytes)	Payload (15 000 bytes)
Header										
15 040 bytes										

Figura 6: *RTPpacket* - mensagem protocolar

Dos diversos campos acima ilustrados correspondentes ao *Header*, os primeiros cinco já foram referidos na subsecção anterior e os restantes cinco foram fornecidos pela equipa docente e, por isso, não se achou relevante mencionar a sua função. Para o campo *Payload* aplicou-se o mesmo critério.

3.2.2. Interações

Este protocolo foi adaptado e, dessa forma, restringe-se apenas a um tipo de pacote pois consegue, com este, cumprir a sua única função - tratar da componente *streaming* propriamente dita.



Figura 7: Sentidos do processo de envio de um pacote Tipo 26

- **Tipo 26:** Pacote que contém a *stream*. Foi escolhido este número pois, segundo a documentação, é o correspondente ao *RTP Payload Format for JPEG-compressed Video*.

4. Implementação

4.1. Decisão da linguagem de programação e protocolos de suporte

Para desenvolver este projeto, o grupo optou por utilizar a linguagem de programação *Java*, visto esta ser a mais conveniente para todos os elementos. Para que a *performance* do sistema desenvolvido fosse a melhor possível, o grupo decidiu utilizar para o envio de mensagens entre máquinas (quer de *stream* ou não), o protocolo **UDP**. Contudo, para garantir um bom funcionamento do sistema, desenvolveu-se um algoritmo para garantir que as mensagens que não fossem de *streaming* fossem entregues, uma vez que estas estavam a ser enviadas pelo protocolo **UDP** que não fornece nenhum tipo de garantia de entrega.

4.2. Construção da topologia *overlay*

Para a construção da topologia *overlay*, foi seguida uma abordagem baseada num *bootstrapper*, de forma a permitir que um nó *overlay* corra sem que este tenha *a priori* conhecimento dos seus vizinhos, apenas precisa de saber comunicar com o *bootstrapper*. Foi decidido ter como *Bootstrapper* o *streamer*, visto que este tem de estar ativo para que todo o sistema funcione. Assim, quando um nó *overlay* se liga, começa por pedir ao *Bootstrapper* informação sobre os seus vizinhos. Este, por sua vez, consulta um ficheiro **JSON** onde se encontra armazenada toda a informação da topologia, e envia ao nó *overlay* toda a informação necessária para este conseguir contactar os seus vizinhos.

4.3. Construção das rotas de fluxo

Quando um nó *overlay* se liga, começa por enviar aos seus vizinhos a sua tabela de endereçamento. Estes, atualizam as suas tabelas de endereçamento com a informação nova fornecida pelo nó que se ligou e, caso tenham obtido nova informação, enviam-na aos seus vizinhos. Acontece, então, um *flooding* controlado da rede, onde após algumas mensagens todos os nós vão ficar a conhecer o novo nó que entrou.

É importante referir que a tabela de endereçamento de um nó *X* vai ter, para cada nó ativo *Y* na rede, uma lista de caminhos possíveis para esse nó, com o respetivo custo associado. Essa lista de caminhos tem no máximo um número de entradas igual ao número de vizinhos do nó *X*, isto é, para cada nó vizinho de *X* é guardado o melhor caminho pelo qual se pode alcançar cada nó *Y*.

4.4. Algoritmo de update das tabelas de endereçamento

Cada nó *overlay* contém uma tabela de endereçamento, onde guarda, para cada nó 'destino', uma lista de nós vizinhos apelidados de 'próximo salto'. Para cada um destes, é também guardado o respetivo custo (número de saltos necessários para alcançar o nó destino através daquele próximo nó).

Quando um nó *overlay* se liga, começa por receber do *Bootstrapper* a informação relativa aos seus vizinhos, construindo assim uma tabela de endereçamento cujas únicas entradas vão ser como chegar a si mesmo (com custo 0) e aos seus vizinhos (com custo 1). Seguidamente, vai enviar a sua tabela aos seus vizinhos.

Quando um nó recebe uma tabela de endereçamento, vai compará-la com a sua e, caso a tabela recebida contenha novas informações, vai atualizar a sua própria tabela adicionando estas informações e, de seguida, vai enviar a sua nova tabela atualizada aos seus vizinhos. Contudo, se não aprender nada com a tabela recebida, o nó não vai atualizar a sua tabela e portanto não vai enviá-la aos seus vizinhos, havendo então um **flooding** controlado pela rede. Assim, quando um novo nó *overlay* se liga, vai desencadear na rede um conjunto de *OttPackets* do tipo **2** que vai fazer com que todos os nós da rede *overlay* passem a ter as novas informações que este trouxe.

4.5. Streaming

Quando um cliente faz um pedido de *streaming*, este não é enviado ao *streamer*, mas sim ao seu vizinho mais próximo deste, evitando-se assim uma sobrecarga do *streamer* com pedidos de *streaming* de cada cliente. Assim, o cliente envia um *streaming request* ao seu vizinho com menor custo (usou-se como métrica o número de saltos) para alcançar o servidor. Por sua vez, este verifica se já está a receber a *stream* e, caso não esteja, vai pedir ao seu vizinho mais próximo do servidor. No pior caso, o pedido do cliente vai "chegar" ao *streamer*, porém, este cenário vai ser muito raro, pois implica que nenhum dos nós entre o cliente e o *streamer* esteja a receber a *stream*.

4.6. Recalculo de rotas

Para que o sistema desenvolvido fosse capaz de, em *run time*, mudar de rota de *streaming* (caso a anterior deixasse de dar por algum nó *overlay* ir abaixo), cada nó faz, periodicamente, uma verificação de que os seus vizinhos estão vivos. Esta verificação consiste em mandar um pedido *isAliveCheck* - tipo 4 e esperar um determinado tempo por uma resposta. Caso a resposta nunca chegue, o processo é repetido mais duas vezes. Se ao fim dos três pedidos não existir uma resposta, esse vizinho vai ser

marcado como morto. Desta maneira, quando um nó *overlay* X está a receber a *stream* de um nó *overlay* Y e este deixa de responder às verificações, o nó X vai pedir a *stream* ao seguinte vizinho que esteja mais próximo do *streamer*.

4.7. Garantia de entrega de pacotes UDP (*OttPacket*)

Para os nós *overlay* comunicarem entre si, como já foi referido anteriormente, escolheu-se utilizar pacotes UDP para que esta comunicação fosse mais rápida. Assim, surgiu a necessidade de garantir que estes pacotes eram de facto entregues e que não se perdiam para os *OttPackets* do tipo **2**, **3** e **6**. Quando um nó *overlay* recebe um pacote de um destes tipos, ele envia uma resposta de confirmação, isto é, um pacote de tipo **7**.

O campo *SequenceNumber*, presente no *Header* de cada pacote *OttPacket*, serve como identificador do pacote, sendo que cada pacote enviado por um determinado nó *overlay* tem um *SequenceNumber* único. Quando uma resposta de confirmação é enviada, esta vai com o *SequenceNumber* do pacote ao qual está a confirmar a receção. Desta forma, um nó *overlay* pode estar à espera da confirmação de N mensagens, não ficando preso à espera da confirmação de um só *OttPacket*. De forma análoga ao *isAliveCheck* referido na secção anterior, de cada vez que um nó envia um *OttPacket*, espera um determinado tempo por uma confirmação e, se esta não chegar, volta a reenviar o mesmo *OttPacket*. Se, após a terceira tentativa de envio não receber confirmação, vai desistir, registando o nó *overlay* destinatário do pacote como *offline*.

4.8. Funcionamento de um nó *overlay*

Um nó *overlay* pode ser um cliente que queira assistir a uma *stream*, um *streamer* que esteja a fornecer a *stream* ou um nó normal, que faz a ligação entre os outros dois tipos.

4.8.1. Streamer

O *Streamer*, tal como os nós *Overlay* normais, tem uma lista de nós para os quais está continuamente a enviar o mesmo vídeo em *loop*. Além disso, encontra-se sempre à escuta de novos pedidos quer de *streaming* (tipo 6), quer de novos a ligarem-se (tipo 0).

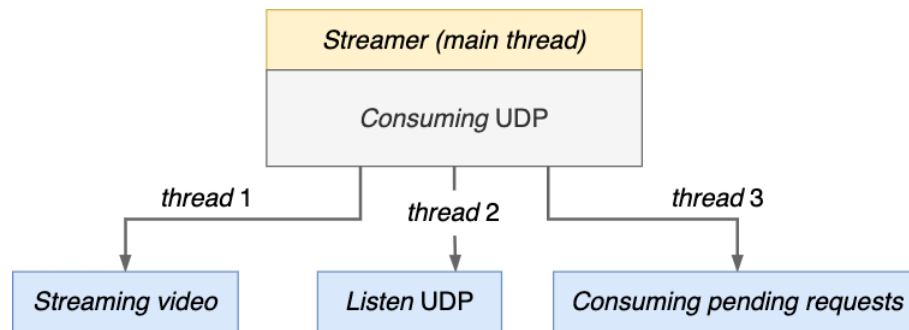


Figura 8: Estrutura do Streamer

4.8.2. Nó Intermédio

Um nó normal trata de fazer chegar a *stream* a quem lhe pede, sempre que possível. Quando recebe um pedido do tipo **6** e não se encontra a receber a *stream*, pede-a então ao seu vizinho mais próximo do *Streamer*. Além disso, verifica periodicamente se os seus vizinhos estão vivos para quando necessário seja capaz de fazer um recálculo das rotas de *streaming*.

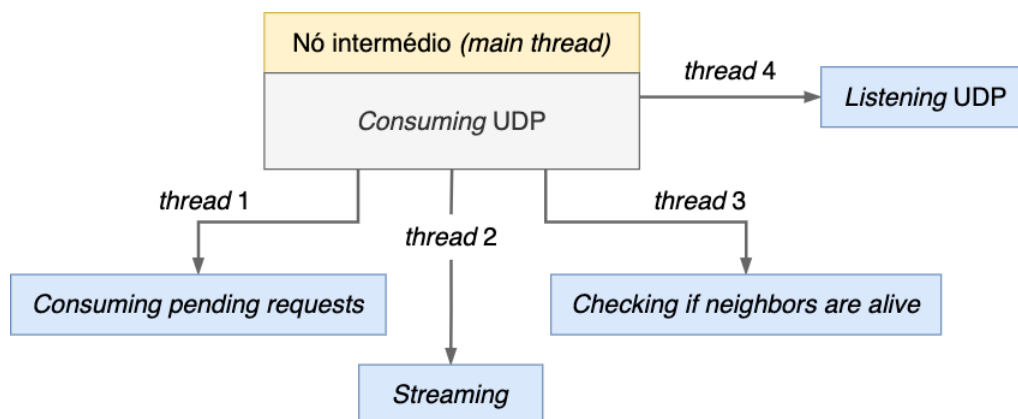


Figura 9: Estrutura de um nó intermédio

4.8.3. Cliente

Um cliente, quando se liga trata logo que possível de requisitar a *stream* ao seu vizinho mais próximo do *Streamer*.

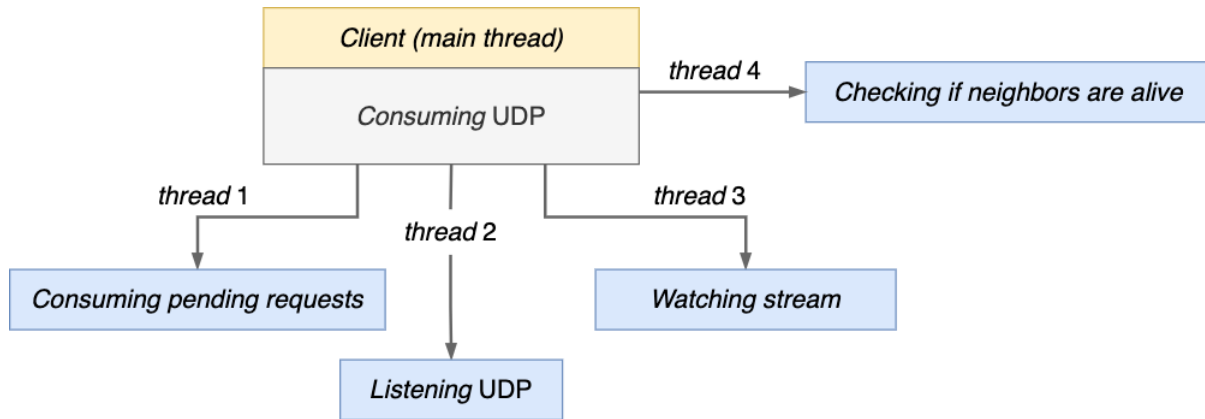


Figura 10: Estrutura de um cliente

4.9. Fluxo das Comunicações do Sistema

De forma a tornar claro todo o processo de troca de mensagens entre as diversas entidades do sistema ao longo do tempo, desenvolveu-se o seguinte diagrama.

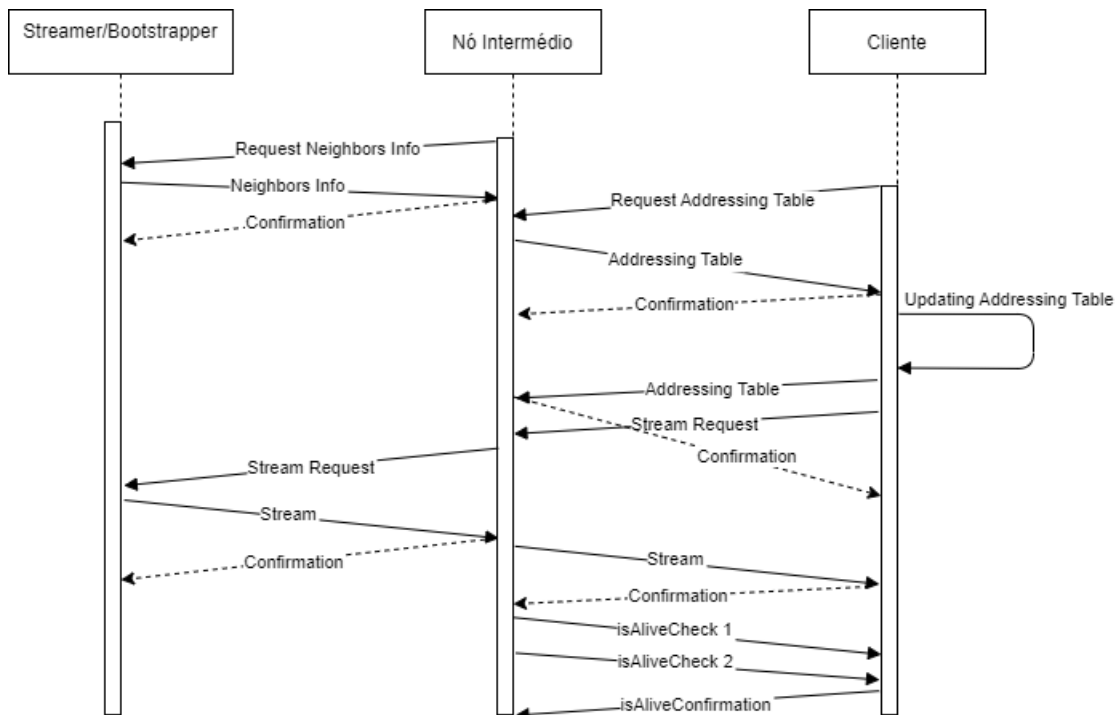


Figura 11: Diagrama temporal de troca de mensagens entre as diversas entidades do sistema

5. Testes e resultados

Para garantir e pôr à prova o correto funcionamento da nossa solução, foram efetuados diversos testes no emulador CORE, usados para cobrir um grande número de casos relevantes.

Tendo em conta o grande número de componentes das diversas topologias, bem como a obrigatoriedade, por parte do CORE, no uso de certos comandos necessários à execução, optou-se por encurtar ao máximo os passos para a realização dos testes, através da criação de algumas *scripts bash*. Esta medida permitiu que estes possam ser feitos de forma mais célere e automática e que possíveis erros de escrita sejam evitados.

- No nó *overlay* correspondente ao *Streamer/Bootstrapper*:

```
su - core // necessário para uso de GUI
bash setup.sh // script responsável pela reposição de diretorias
bash javac.sh // script responsável pela criação dos executáveis
bash run.sh // script responsável pelo "run" do programa
```

- Nos restantes nós *overlay*:

```
su - core
bash setup.sh
bash run.sh ... // depende do tipo de nó
```

- Caso seja nó intermédio:

```
bash run.sh "IPnó:Portanó:idenó"
```

- Caso seja nó Cliente:

```
bash run.sh "IPnó:Portanó:idenó" "-c"
```

Acrescenta-se em forma de nota que, para efeitos de legibilidade do documento, criou-se uma secção Anexos que contém o resultado final da execução dos três cenários, acompanhado das respetivas *bash*.

5.1. Cenário 1: *overlay* com 4 nós

Para este cenário foi desenhada uma rede *overlay* com 4 nós: um servidor, dois clientes e um nó intermédio, conforme demonstrado na figura seguinte:

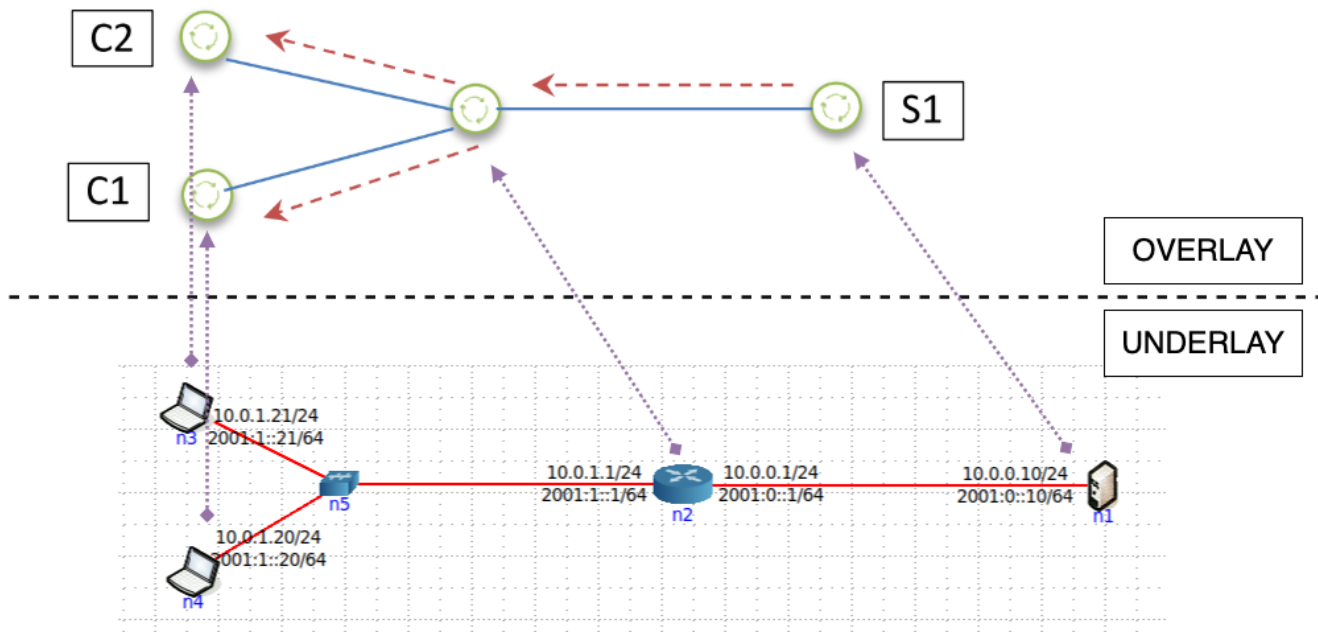


Figura 12: *Overlay* com 4 nós

Após executar a nossa solução, facilmente se verifica o bom funcionamento da mesma, visto que os dois clientes estão, como suposto, a consumir uma *stream* multimédia enviada pelo servidor/*streamer*. O resultado final desta execução pode ser consultado em A.

5.2. Cenário 2: *overlay* com 6 nós

Neste cenário foi desenhada uma rede *overlay* com 6 nós: um servidor, dois clientes e 3 nós intermédios, conforme demonstrado na figura seguinte:

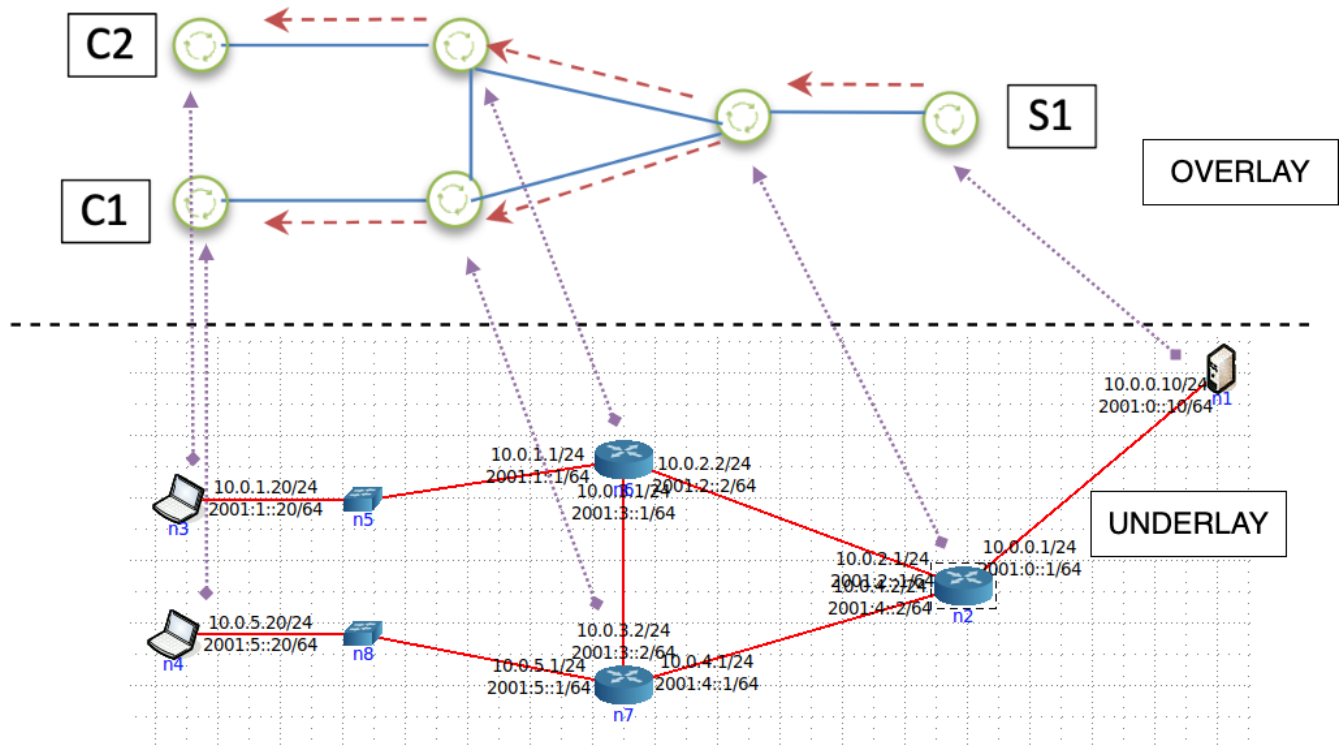


Figura 13: Overlay com 6 nós

Após executar a nossa solução, facilmente se verifica o bom funcionamento da mesma, visto que os dois clientes estão, como suposto, a consumir uma *stream* multimédia enviada pelo servidor/*streamer* sendo que esta obrigatoriamente tem que passar por todos os nós intermédios - o que acontece. O resultado final desta execução pode ser consultado em B.

5.3. Cenário 3: overlay com 11 nós

Por fim, e com base no enunciado prático, foi adaptada uma rede *overlay* com 11 nós: um servidor, 5 clientes e 5 nós intermédios.

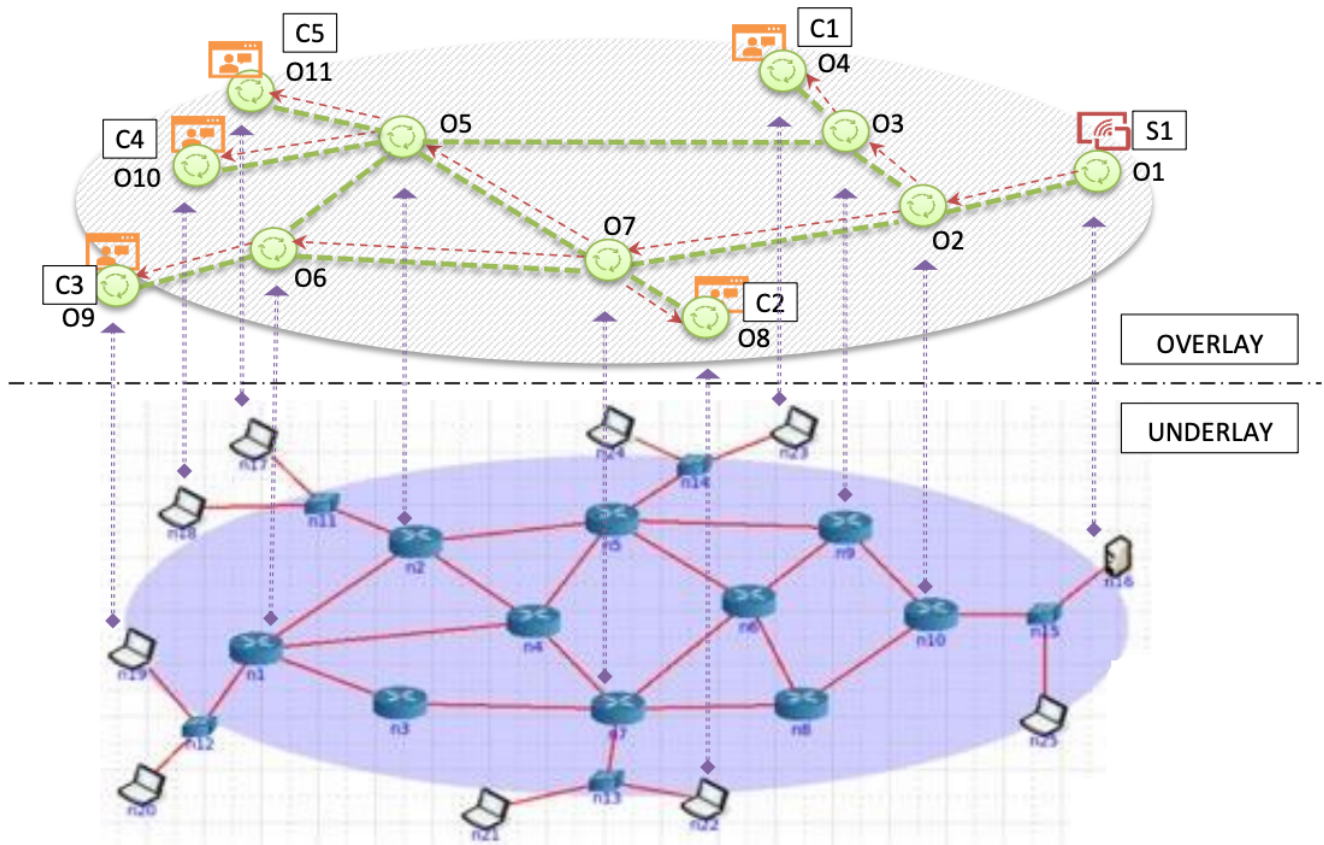


Figura 14: Overlay com 11 nós

Depois de executar a solução proposta, facilmente se verifica o bom funcionamento da mesma, visto que todos os 5 clientes estão, como suposto, a consumir uma *stream* multimédia enviada pelo servidor/*streamer* sendo que, neste caso, esta obrigatoriamente tem que passar por todos os nós intermédios - o que acontece pois, caso contrário, teríamos clientes sem *stream*. O resultado final desta execução pode ser consultado em C.

6. Conclusões

De um modo geral, o sentimento que passa é que este trabalho prático, além de extremamente desafiante, ajudou bastante a consolidar a matéria lecionada nas aulas teóricas. Reconhecemos que foi muito enriquecedor para o nosso coletivo, pois permitiu que cimentássemos todas as competências envolvidas - parte delas já abordadas no trabalho prático anterior - bem como a aplicação destas num caso prático real, como a criação de um serviço *Over the Top* sobre uma determinada infraestrutura IP.


Assumimos que houve alguma dificuldade na ligação de todos os conceitos, pois estamos a falar de uma solução que envolve um grande número de componentes. Além disso, foi também necessário transformar toda a arquitetura e requisitos da solução em código, sempre a pensar na eficiência. Então, para a implementação propriamente dita, foi precisa toda uma manipulação de *Sockets*, aliada a uma concorrência praticamente perfeita (*Threads*) pois, de outra forma, não seria possível obter os resultados pretendidos.

Todas as decisões tomadas pelo grupo, desde as estratégias escolhidas até traços fundamentais da implementação - como qual seria a melhor resposta para, por exemplo, efetuar, se necessário, o recálculo de rotas - foram sempre discutidas entre todos, pelo que o grupo acha que a solução final apresenta um conjunto de resoluções bastante válidas, o que foi comprovado pelo sucesso de todos os testes efetuados.

Em suma, o esforço foi grande com o intuito de garantir boas soluções para o enunciado proposto deixando, assim, um sentimento de objetivo cumprido.

A. Cenário 1

Node: 3 (on n3)



Setup


Play

Pause

Close

Total Bytes Received: 19485000
Packet Lost Rate: 2,6
Data Rate: 0,01 bytes/s

Node: 4 (on n4)



Setup

Play

Pause

Close

Total Bytes Received: 13485000
Packet Lost Rate: 1,8
Data Rate: 0,01 bytes/s

Boostrapper / Streamer

Node 2

```
Servidor: vai enviar video do file movie.Mpeg
>>> Packet received from IP: /10.0.0.1 port: 8000
A new neighbors request has been made.

>>> Sent packet to IP: /10.0.0.1 port: 8000 type: 1
>>> Sending repeated packet to: /10.0.0.1 type: 1
>>> Packet received from IP: /10.0.0.1 port: 8000
Received confirmation, removing request 1!
>>> Packet received from IP: /10.0.0.1 port: 8000
>>> Packet received from IP: /10.0.0.1 port: 8080
A new neighbors request has been made.

>>> Sent packet to IP: /10.0.0.1 port: 8080 type: 1
>>> Packet received from IP: /10.0.0.1 port: 8080
Received confirmation, removing request 2!
>>> Packet received from IP: /10.0.0.1 port: 8000
Received stream request from node 2
>>> Packet received from IP: /10.0.0.1 port: 8090
A new neighbors request has been made.

>>> Sent packet to IP: /10.0.0.1 port: 8090 type: 1
>>> Packet received from IP: /10.0.0.1 port: 8090
Received confirmation, removing request 3!
```

Node 3

```
type: 7
Node 3: sent packet to IP: /10.0.0.1 port: 8000 type: 3
destingNodeid: 1
destingNodeid: 2
destingNodeid: 3
destingNodeid: 4
cost=2, is0n=true]]
->Requesting stream to node 2
->Node 3: sent packet to IP: /10.0.0.1 port: 8000 type: 6
->Node 3: sent confirmation packet to IP: /10.0.0.1 port: 8000 type: 7
->Received confirmation, removing request 2!
->Node 3: received packet from IP: /10.0.0.1 port: 8000
node: 2T
type: 7
->Did not update addressing table.
->Node 3: sent confirmation packet to IP: /10.0.0.1 port: 8000 type: 7
->Received confirmation, removing request 3!
->Requesting stream to node 2
->Node 3: received packet from IP: /10.0.0.1 port: 8000
node: 2T
type: 7
->Node 3: sent packet to IP: /10.0.0.1 port: 8000 type: 6
->Received confirmation, removing request 4!
Play Button pressed !
```

Node 4

```
destingNodeid: 1
destingNodeid: 2
destingNodeid: 3
destingNodeid: 4
cost=2, is0n=true]]
->Requesting stream to node 2
->Node 4: received packet from IP: /10.0.0.1 port: 8000
node: 2T
type: 7
->Node 4: sent packet to IP: /10.0.0.1 port: 8000 type: 6
->Node 4: sent confirmation packet to IP: /10.0.0.1 port: 8000 type: 7
->Received confirmation, removing request 2!
->Node 4: did not update addressing table.
->Node 4: sent confirmation packet to IP: /10.0.0.1 port: 8000 type: 7
->Did not update addressing table.
->Node 4: sent confirmation packet to IP: /10.0.0.1 port: 8000 type: 7
->Received confirmation, removing request 3!
->Requesting stream to node 2
->Node 4: received packet from IP: /10.0.0.1 port: 8000
node: 2T
type: 7
->Node 4: sent packet to IP: /10.0.0.1 port: 8000 type: 6
->Received confirmation, removing request 4!
Play Button pressed !
```

zoom 100% CPU 29% (29) 225M free

B. Cenário 2


[illegible]

C. Cenário 3

Streamer (on n16)

Send frame #208


Node 4 (on n23)



SetupPlayPauseClose

Total Bytes Received: 127590000
Packet Lost Rate: 17,01
Data Rate: 0,08 bytes/s

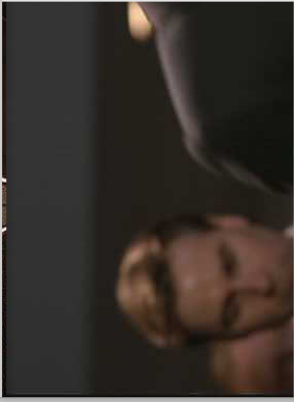
Node 8 (on n22)



SetupPlayPauseClose

Total Bytes Received: 79950000
Packet Lost Rate: 10,66
Data Rate: 0,05 bytes/s

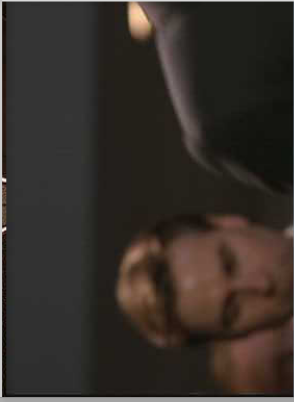
Node 9 (on n19)



SetupPlayPauseClose

Total Bytes Received: 64995000
Packet Lost Rate: 8,66
Data Rate: 0,04 bytes/s

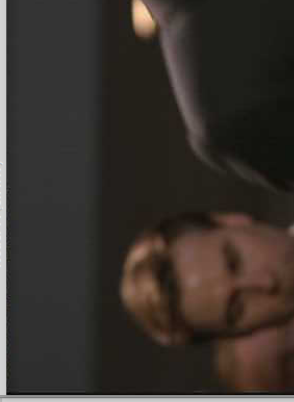
Node 10 (on n18)



SetupPlayPauseClose

Total Bytes Received: 47265000
Packet Lost Rate: 6,3
Data Rate: 0,03 bytes/s

Node 11 (on n17)



SetupPlayPauseClose

Total Bytes Received: 38655000
Packet Lost Rate: 5,15
Data Rate: 0,02 bytes/s

Streamer/Bootstrapper.mn

```
>> Sent packet to IP: /10.0.18.20 port: 8094 type: 1
>> Packet received from IP: /10.0.18.20 Port: 8094
Received confirmation, removing request 71
>> Packet received from IP: /10.0.17.20 Port: 8095
A new neighbors request has been made.

>> Sent packet to IP: /10.0.17.20 port: 8095 type: 1
Sending repeated packet to: /10.0.17.20 type: 1
>> Packet received from IP: /10.0.17.20 Port: 8095
Received confirmation, removing request 81
>> Packet received from IP: /10.0.19.21 Port: 8096
A new neighbors request has been made.

>> Sent packet to IP: /10.0.19.21 port: 8096 type: 1
>> Packet received from IP: /10.0.19.21 Port: 8096
Received confirmation, removing request 101
>> Packet received from IP: /10.0.19.20 Port: 8097
A new neighbors request has been made.

>> Sent packet to IP: /10.0.19.20 port: 8097 type: 1
>> Packet received from IP: /10.0.19.20 Port: 8097
Received confirmation, removing request 101
```

n6.pid

n6.xy

n7

n9.pid

n9.xy

n10

n17.pid

n17.xy

n18

n20.pid

n20.xy

n21

n23.pid

n23.xy

n24