Universidade do Minho
Escola de Engenharia
Mestrado em Engenharia Informática

Unidade Curricular de Engenharia de Serviços em Rede

Ano Letivo de 2024/2025

Trabalho Prático Nº2

Serviço *over-the-top* para entrega de multimédia

Carlos Ribeiro
PG55926

Diogo Matos
PG55934

Júlio Pinto
PG57883

4 dezembro, 2024

ESR

Índice

1. Introdução **1**

2. Arquitetura da Solução **1**

3. Especificação dos protocolos **2**
3.1. Protocolo de *streaming* 2
3.2. Protocolo de controlo 3
3.2.1. Campos da mensagem protocolar 3
3.2.1.1. `NodeType` 3
3.2.1.2. `MsgType` 3
3.2.1.3. `RSP` 4
3.2.1.4. `FWD` 4
3.2.1.5. `BCK` 4
3.2.1.6. `TimeStamp` 4
3.2.1.7. `Id` 4
3.2.1.8. `Hash` 4
3.2.1.9. `PayloadSize` 4
3.2.2. Tipos de mensagens 4
3.2.2.1. `Error` 4
3.2.2.2. `Ok` 4
3.2.2.3. `Hello` 4
3.2.2.4. `IWantStream` 5
3.2.2.5. `ConnFlood` 5
3.2.2.6. `AcceptedConn` 5
3.2.2.7. `TimeHop` 5

4. Implementação **5**
4.1. Tabelas de encaminhamento 5
4.2. Entrada de um cliente na rede 6
4.3. Escolha da conexão por parte do cliente 7
4.4. Dinamismo da rede 8
4.5. Tolerância a falhas 8
4.5.1. Problemas com esta conceção 9
4.5.1.1. Dependências 9
4.5.1.2. Estratégia de mitigação implementada 10
4.5.1.3. Degeneração da rede 10

5. Conclusões **11**

1. Introdução

O crescente consumo de conteúdos multimédia em tempo real na Internet, impulsionado por serviços *Over-The-Top* (OTT) como o *Netflix* ou o *HBO*, apresenta desafios significativos às infraestruturas de rede tradicionais. Estes serviços, ao utilizarem redes *overlay* aplicacionais otimizadas, permitem contornar limitações de largura de banda e congestão, garantindo uma experiência de qualidade aos utilizadores finais.

No âmbito deste trabalho prático, propõe-se o desenvolvimento de um protótipo de serviço de entrega de vídeo em tempo real, explorando arquiteturas de rede CDN (*Content Distribution Network*) e mecanismos de otimização para transmissão eficiente, de baixa latência e à prova de falhas.

Neste relatório, exploram-se as decisões arquiteturais que estiveram na base da nossa solução, assim como a especificação dos protocolos desenvolvidos, todos os detalhes de implementação relevantes e ainda os resultados obtidos.

2. Arquitetura da Solução

Para a escolha da arquitetura a utilizar no nosso projeto, tínhamos essencialmente duas opções viáveis: a utilização de um *bootstrapper*, responsável por conhecer e monitorizar toda a rede, ou o desenvolvimento de uma solução distribuída, onde os nós se organizam autonomamente. Depois de alguma reflexão, achamos que o desenvolvimento de uma arquitetura distribuída seria um desafio interessante para conceber um sistema robusto e relativamente simples, para além de apresentar diversas vantagens em relação à solução do *bootstrapper*.

Num contexto à escala internacional, a utilização de um *bootstrapper* central e único levantaria problemas quanto ao *overhead* da rede, assim como atrasos potencialmente significativos na criação e atualização de rotas, por exemplo, assumindo uma transmissão de vídeos a $30\ \mathrm{fps}$, a janela de atuação mantendo a qualidade do serviço seria de $33\ \mathrm{ms}$. Tentar resolver estes problemas com a introdução de redundância ao nível dos *bootstrapers* introduziria uma necessidade adicional de consistência entre o estado dos mesmos e o da rede e ainda entre as diferentes instâncias.

Assim, em vez de utilizar um *bootstrapper*, optamos por implementar um **protocolo de controlo distribuído**, baseado em **TCP**, para que os nós da rede comuniquem efetivamente entre si e troquem informações relevantes de forma confiável. Já a responsabilidade, que seria assumida pelo *bootstrapper*, de informar os clientes dos endereços dos POPs (*Points Of Presence*), ou seja a funcionalidade DNS, é assumida pelo Servidor nesta arquitetura (exploramos a forma como tal acontece em maior detalhe mais à frente). O *bootstrapper* poderia também surtir a função de monitorizador da rede (quanto a métricas de rede e à topologia do *overlay*), no entanto, manter o estado deste a "concordar" com o estado real da rede pareceu-nos um problema demasiado complicado e não vantajoso.

A funcionalidade de *streaming* é assegurada pelo protocolo **RTP**, que funciona sobre UDP, com um pequeno cabeçalho.

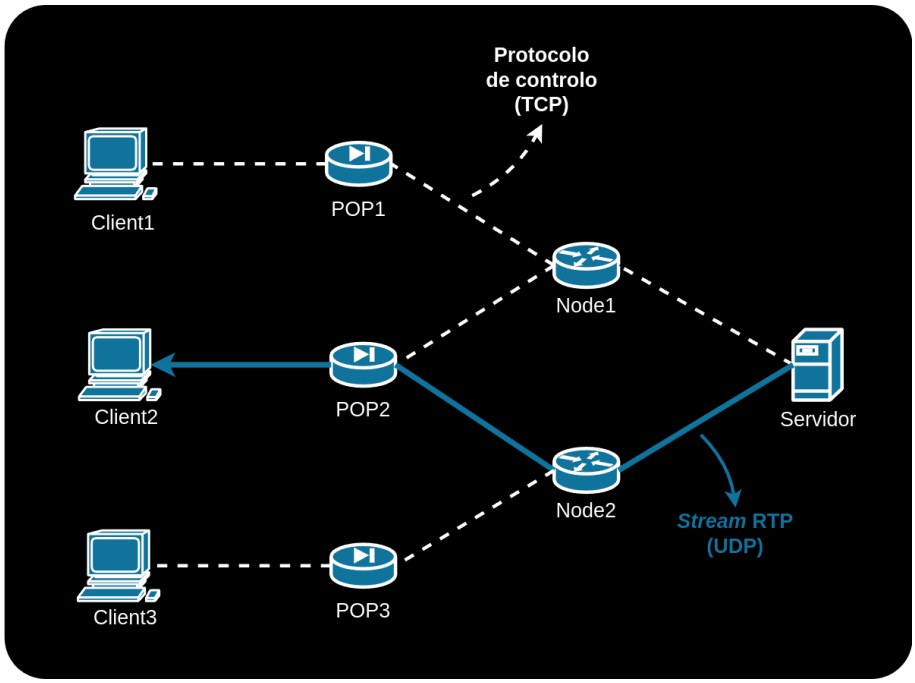

Figura 1: Ilustração da Arquitetura da Solução

3. Especificação dos protocolos

3.1. Protocolo de *streaming*

O protocolo de *streaming* é responsável por transportar a transmissão de vídeo e a informação necessária para a identificar.

No processo de envio de uma transmissão, o servidor lê pacotes de uma fonte contendo *frames* de um ficheiro de vídeo e "embrulha" cada *frame* num cabeçalho que contém um identificador da transmissão sob o formato de uma `string` e o seu comprimento, sob o formato de um inteiro. Segue-se uma ilustração de um pacote deste protocolo.

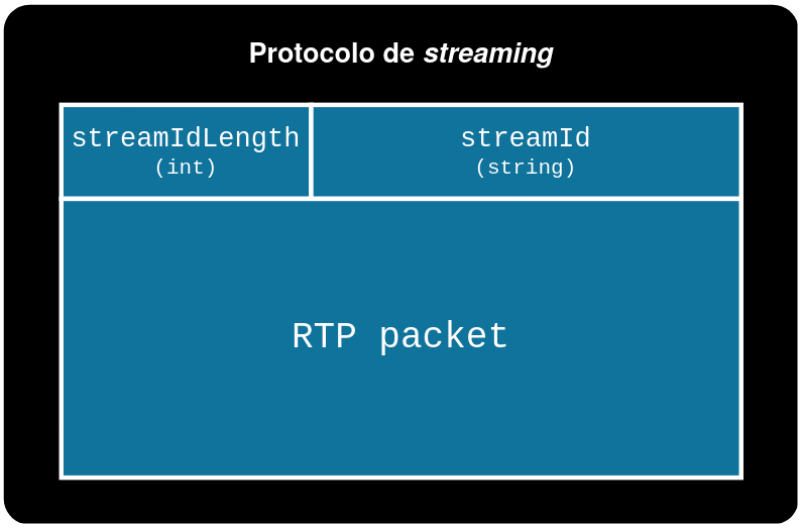

Figura 2: Ilustração do Protocolo de *Streaming*

Estes pacotes passam pela rede de *overlay* e são "desembrulhados" pelos nodos, para saberem para onde encaminhar, voltando depois a embrulhá-los no processo.

Por sua vez, os clientes recebem estes pacotes, "desembrulham-nos" e verificam que o *sequence number*, parte do pacote RTP, é superior ao do último recebido de modo a assegurar a ordem dos mesmos, caso isso não se verifique o pacote é descartado.

3.2. Protocolo de controlo

O protocolo de controlo é responsável por transportar informação útil à comunicação entre nós da rede.

As mensagens de controlo são trocadas entre os nodos de *overlay*, sobre TCP, tomando proveito de algumas características deste protocolo base:

- Confiabilidade;
- Comunicação *duplex*;
- Facilidade de identificação da desconexão de um interlocutor.

No entanto, utilizar TCP trás algumas consequências que podem ser consideradas desvantagens:

- *Overhead* protocolar, tanto ao nível de cada pacote individual, como ao nível da rede (*Syns*, *Acks*, etc);
- O protocolo é utilizado para medir *delays* entre nodos, no entanto, como o protocolo de *streaming* assenta sobre UDP, os *delays* em TCP serão invariavelmente superiores, não refletindo de forma fidedigna o estado da rede.

Segue-se uma ilustração do formato do protocolo:

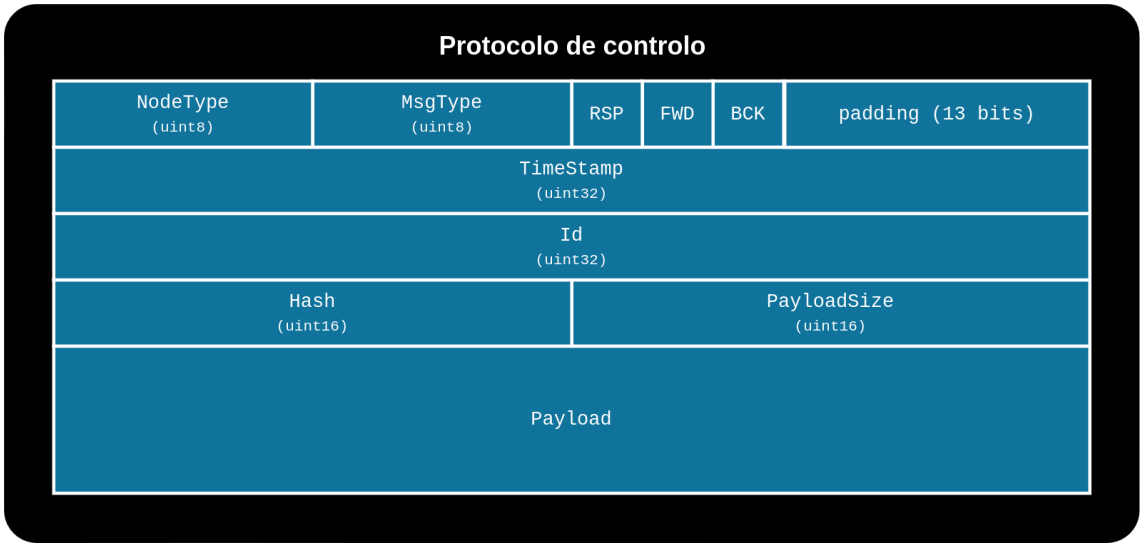

Figura 3: Ilustração do Protocolo de Controlo

3.2.1. Campos da mensagem protocolar

3.2.1.1. NodeType

Inteiro que estabelece o tipo de nó que envia o pacote, os valores possíveis são:

- **1**: Servidor
- **2**: Cliente
- **4**: Nodo
- **8**: *Point of presence*

3.2.1.2. MsgType

Inteiro que indica o tipo de mensagem que o protocolo comunica, pode tomar os seguintes valores:

- **MsgError**
- **Hello**
- **ConnFlood**
- **TimeHop**

- **`MsgOk`**
- **`IWantStream`**
- **`AcceptedConn`**

Cada tipo de mensagem pode, ou não, ser acompanhado por um *payload* específico. A função e *payload* relativo a cada mensagem são especificados em Secção 3.2.2.

3.2.1.3. `RSP`

Flag booleana "*Response*" que indica que um pacote é uma resposta a uma mensagem.

3.2.1.4. `FWD`

Flag booleana "*Forward*" que indica se uma mensagem deve ser propagada no sentido dos descendentes do nodo atual.

3.2.1.5. `BCK`

Flag booleana "*Backward*" que indica se uma mensagem deve ser propagada no sentido dos ascendentes do nodo atual.

3.2.1.6. `TimeStamp`

Timestamp utilizada para transportar informação temporal, utilizada para calcular intervalos de tempo entre dois acontecimentos, como por exemplo o RTT (*Round Trip Time*) entre dois nodos da rede ou o tempo de chegada de um pacote originário de um *flooding*.

3.2.1.7. `Id`

Identificador da mensagem, útil por exemplo para identificar uma resposta a uma outra mensagem.

3.2.1.8. `Hash`

Espaço dedicado ao envio de uma *hash*, de modo a garantir a integridade dos dados do *payload*. Não utilizado atualmente mas introduzido como uma forma de "*future proofing*".

3.2.1.9. `PayloadSize`

Indica o tamanho em *bytes* do *payload* do pacote.

3.2.2. Tipos de mensagens

3.2.2.1. `Error`

Comunica uma mensagem de erro, *payload* composta por uma *string* que indica o texto descritivo do erro.

```
type body struct{
  errorMsg string
}
```

3.2.2.2. `Ok`

Comunica uma confirmação positiva relativa a algum processo, pedido ou como resposta a uma outra mensagem. Não possui um *payload*.

3.2.2.3. `Hello`

Enviada aos nodos passados como vizinhos na iniciação do nodo de modo a iniciar a comunicação com os mesmos. O campo `NodeType` do cabeçalho ajuda o recetor a perceber como tratar esta conexão de acordo com o tipo de nodo. Não possui um *payload*.

3.2.2.4. `IWantStream`

Utilizada por um cliente para pedir um conteúdo ao servidor, admitindo uma resposta `Ok` ou `Error`. O *payload* é composto pela *string* que identifica o conteúdo a ser pedido.

```
type body struct{
  content string
}
```

3.2.2.5. `ConnFlood`

Comunica que a mensagem atual faz parte de um processo de *flood*, que permite criar e atualizar as tabelas de encaminhamento dos nodos intermédios da rede de *overlay*. O *payload* inclui toda a informação necessária à criação/atualização dessas mesmas tabelas.

```
type body struct {
  ConnId    string
  ClientIp  IP
  TotalHops int
  SplitAt   int
  TotalTime int
  }
```

3.2.2.6. `AcceptedConn`

Utilizada para comunicar que a variável `InUse`, cuja responsabilidade é descrita em Secção 4.1, deve ser alterada nas tabelas de encaminhamento da rede de *overlay*, uma vez que uma nova conexão foi aceite.

```
type body struct {
  connId  string
}
```

3.2.2.7. `TimeHop`

Utilizada para descobrir o RTT entre dois nodos, adjacentes numa árvore de distribuição. Não possui um *payload*.

4. Implementação

4.1. Tabelas de encaminhamento

Cada nodo, incluindo o servidor, guarda uma tabela de encaminhamento representada num *array thread safe* de *"routing entries"*, segue-se um exemplo de uma possível tabela:

From	To	ConnId	InUse	TotalHops	HopsSinceSplit	LastHopTime (ms)	TimeFromSource (ms)
10.0.0.1	-	[a.mp4]−1-2	True	2	1	5	12
-	10.0.0.2	[a.mp4]−1-2-1	True	-	-	-	-
-	10.0.0.3	[a.mp4]−1-2-2	False	-	-	-	-

Tabela 1: Tabela exemplar de encaminhamento de um nodo da rede

Esta tabela mapearia o seguinte excerto da topologia de rede:

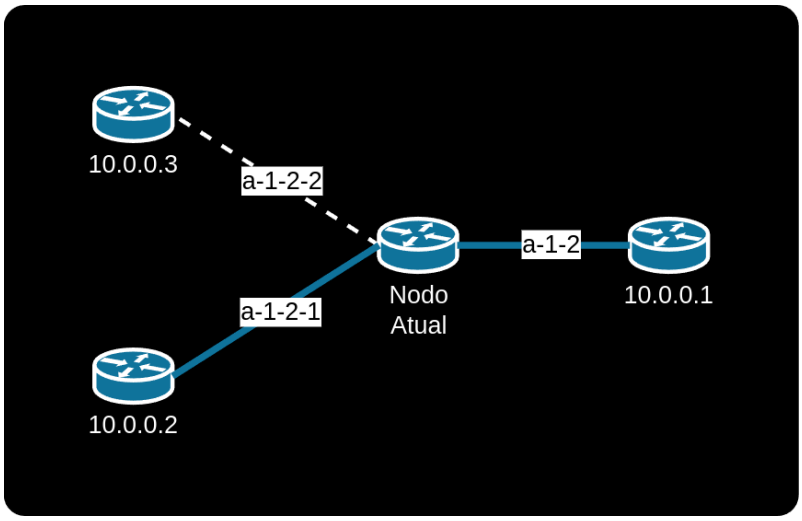

Figura 4: Excerto da topologia de rede, de acordo com a tabela de encaminhamento

Na tabela, nunca uma linha tem as colunas `From` e `To` preenchidas ao mesmo tempo. Optou-se por poder ter conexões não utilizadas num dado momento (`InUse = False`), já que isso permite atualizar dinamicamente o nodo fonte case este se torne indisponível, por qualquer razão, sendo possível simplesmente pedir o conteúdo ao IP `From` de uma outra conexão que esteja atualmente desligada.

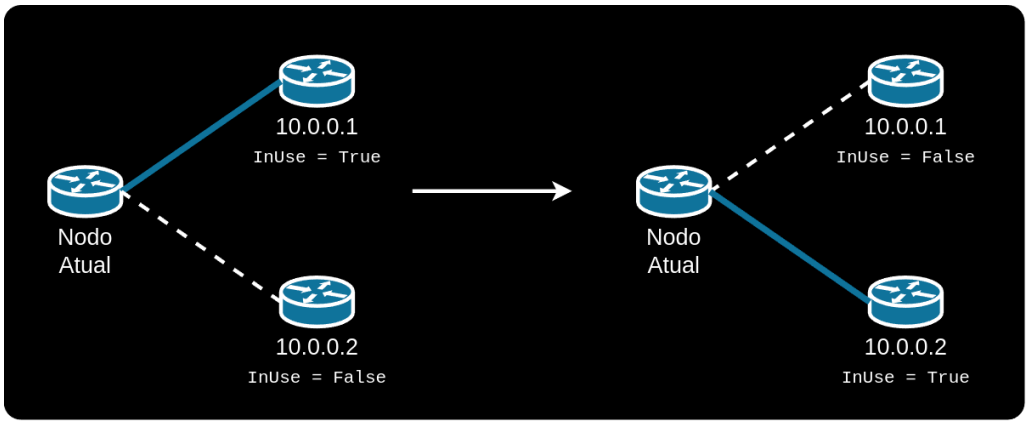

Figura 5: Ilustração do processo de mudança dinâmica do nodo fonte

4.2. Entrada de um cliente na rede

1. Cliente é inicializado tomando como argumento o conteúdo que pretende ver, e o endereço IP do servidor;
2. Cliente envia pedido `IWantStream` ao servidor;
3. O Servidor responde com `Ok` se possuir o vídeo;
4. O Servidor começa uma inundação da rede, enviando uma mensagem `ConnFlood`, com o nome `[<conteudo>]-N`, e `TotalTime = 0` a todos os seu vizinhos (`N` começa em 1, sendo incrementado por cada nodo a que é enviada a mensagem);
5. Cada Nodo, ao receber uma mensagem de `ConnFlood`:
    5.1. Verifica que o pacote que recebeu não se refere a uma conexão que possa já ter passado por ele;
    5.2. Envia um pedido `TimeHop` ao emissor da mensagem, com o `timestamp` do seu relógio;
    5.3. Este é refletido pelo recetor, efetivamente permitindo ao Nodo calcular o RTT entre eles;
    5.4. Uma entrada é adicionada à tabela de encaminhamento do Nodo, com toda a informação que existe no pacote, marcada com `InUse = False`
    5.5. Se o nodo for um:

- **Nodo intermédio**: A mensagem é enviada a todos os Nodos vizinhos, acrescentando metade do RTT calculado e um sufixo;
- ***Point of presence***: A mensagem é enviada apenas ao cliente cujo o IP consta no corpo da mensagem;
- **Cliente**: O Cliente escolhe aceitar ou não a *stream*.

6. O Cliente recebes várias mensagens `ConnFlood`;
7. O Cliente responde à mais rápida com um pacote `AcceptedConn`;
8. Este pacote é propagado para trás até ao Servidor (inclusive), alterando para `InUse = True` as entradas relativas a si e os seus antecessores pelo caminho;
9. Caso o Servidor não estivesse já a servir esse conteúdo, começa a servi-lo.

4.3. Escolha da conexão por parte do cliente

Por simplicidade, e considerando que esta funcionalidade foi implementada antes da monitorização de métricas de rede existir, o cliente aceita simplesmente a primeira conexão que recebe.

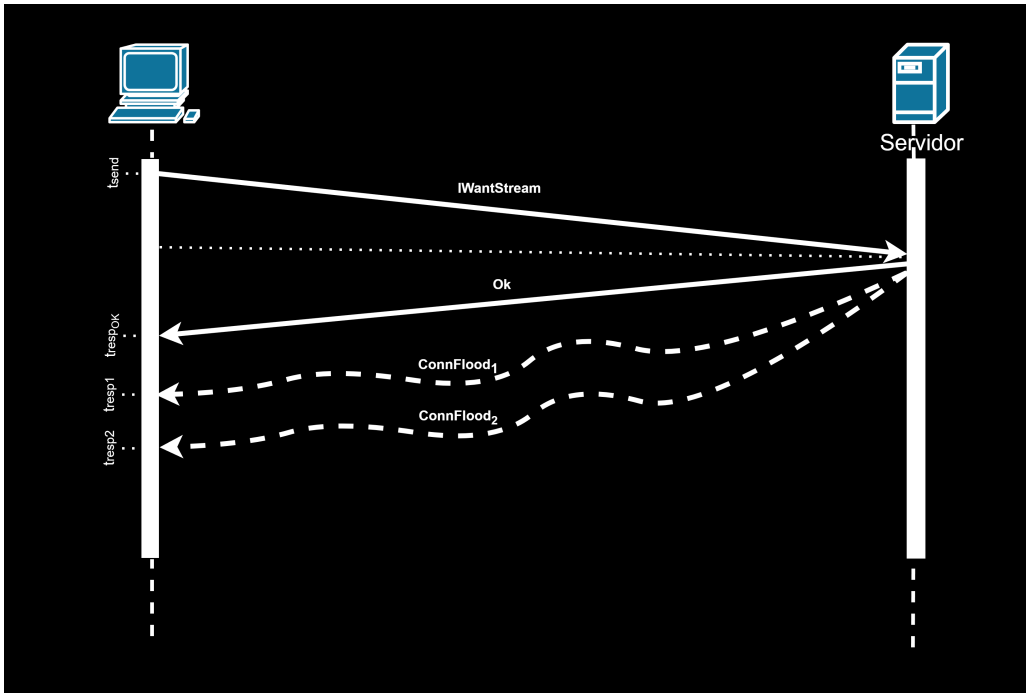

Figura 6: Ilustração do processo de conexão ao servidor por parte do cliente

Esta abordagem permite comparar os tempos na direção Servidor $\rightarrow$ Cliente, que é o que queremos otimizar, uma vez que para todas as mensagens o tempo de envio é o mesmo variando apenas o tempo da resposta.

No entanto, outra métrica que é importante otimizar é a utilização da rede, de grosso modo, o cliente deveria favorecer conexões que bifurcam "tarde", minimizando o campo `HopsSinceSplit`, não comprometendo a experiência de utilização.

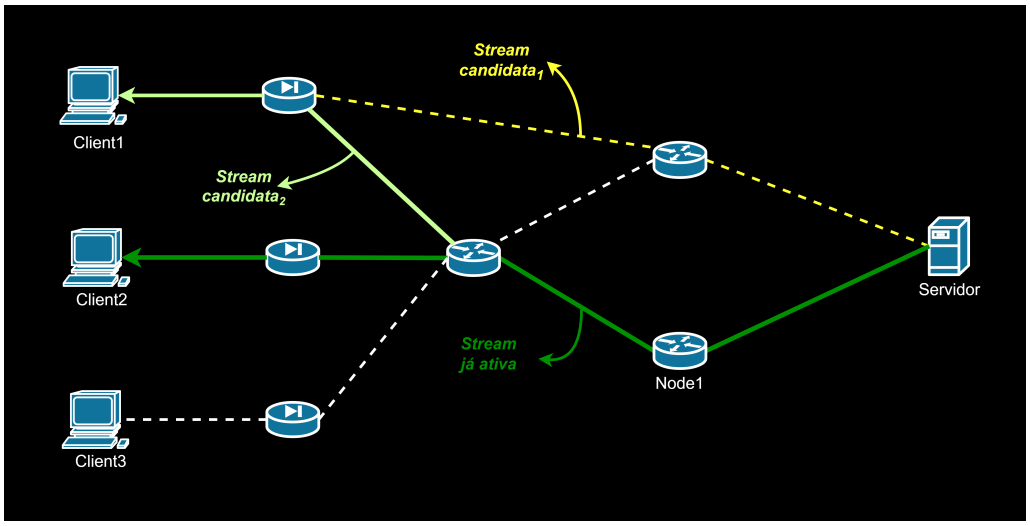

Figura 7: Exemplo de escolha de uma conexão por parte de um cliente

Na figura acima, o "Cliente1", que se juntou à rede, deve escolher entre a "*Stream* candidata" 1 ou 2. Se estas tiverem *delays* comparáveis, deve escolher a "*Stream* candidata 2" porque na rede, num dado instante, por cada *frame* do vídeo, haverá 2 saltos da *frame* na rede a mais, ao passo que se tivesse escolhido a "*Stream* candidata 1" haveria 3.

4.4. Dinamismo da rede

Quando um novo nodo entra na rede, anuncia-se aos seus vizinhos e fica *idle* até que haja uma *flood* que passe por ele, quando isto acontece, junta-se a uma árvore de distribuição de um ou mais conteúdos. Para garantir que a rede se mantém atualizada, o Servidor envia pacotes de `ConnFlood` periodicamente, estes passam por todos os nodos, incluindo os que se juntaram à rede mais recentemente.

4.5. Tolerância a falhas

Conceber uma forma de manter este sistema resiliente de forma autónoma revelou-se uma tarefa quase Herculeana.

Quando um Nodo deteta que o Nodo que lhe está a servir conteúdo morre, liga-se a um outro nodo que possa servir de fonte, com a *flag* `InUse` a falso, enviando um pacote `AcceptedConn` para este.

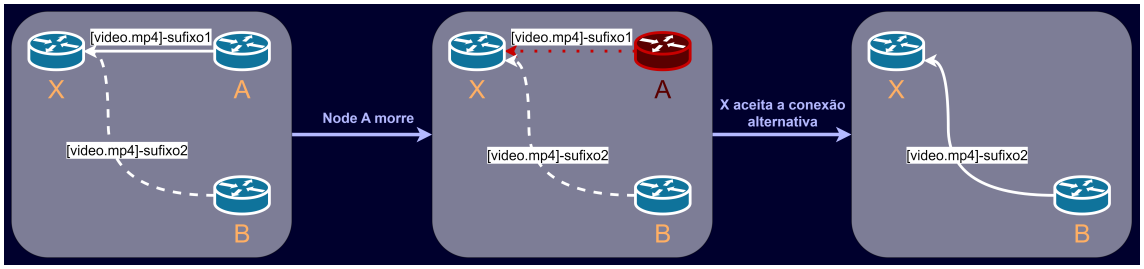

Figura 8: Recuperação de falha simples

Para tal, o Nodo precisa de garantir que tem pelo menos duas fontes para cada conteúdo, quando deteta que só tem uma, entre em **estado de pânico** e tenta arranjar um "contacto de emergência". Faz isto pedindo ao nodo fonte que tem atualmente que lhe envie o IP da sua fonte, ou seja, procura conhecer o nodo pai do seu nodo fonte enviando-lhe uma mensagem de tipo `Hello`, estabelecendo assim uma conexão entre os dois. Da próxima vez que haja um *flood*, este passa por essa ligação, dando aso a uma conexão alternativa. Este *flood* pode ser começado periodicamente pelo servidor, ou pedido através de alguma mensagem com *backtracking* até o servidor.

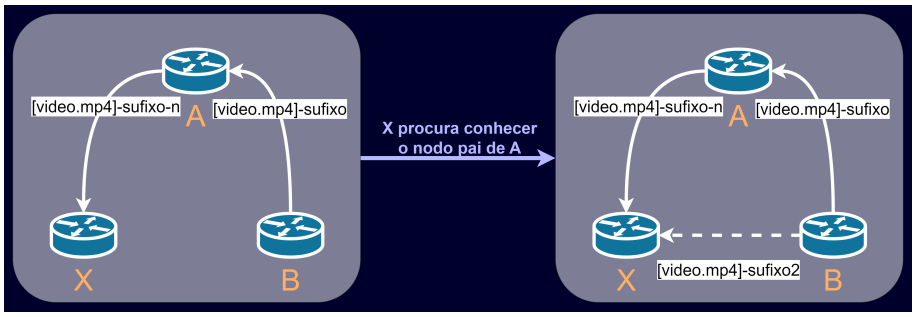

Figura 9: Estratégia de Recuperação

4.5.1. Problemas com esta conceção

4.5.1.1. Dependências

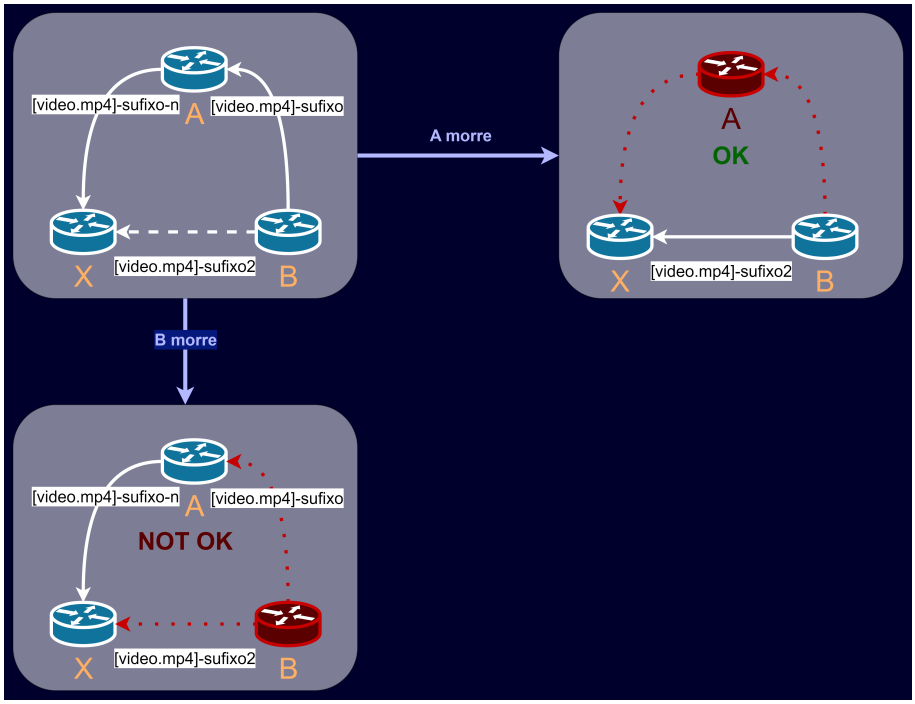

Figura 10: Problemas da conexão a duas fontes (“Pai” e “Filho”)

A abordagem de ligar a uma outra conexão disponível (A_1) quando uma morre (A_2) sofre se A_1 depender de A_2, isto é detetável, basta que A_1 seja prefixo de A_2.

No entanto, registando o estado acima como **estado de pânico**, acabaríamos com uma situação em que X contacta todos os seus antecedentes até chegar ao servidor, podendo acabar por degenerar a rede para simplesmente múltiplos *unicasts* a surgir do Servidor.

Sabendo isto, se assumirmos uma chance p de qualquer nodo falhar:

$$\text{Stream falhar} \Leftrightarrow A \text{ falhar} \wedge B \text{ falhar}$$
$$P(A \text{ falhar}) = P(B \text{ falhar}) = p$$

Pareceria que $P(\text{Stream falhar}) = P(A \text{ falhar}) \times P(B \text{ falhar})$, no entanto:

$$P(A \text{ falhar}|_{B \text{ falhou}}) = 1$$
$$\text{logo}$$
$$P(\text{Stream falhar}) = p$$

A menos que $P(B \text{ falhar}) < P(A \text{ falhar})$, coisa que assumimos no caso em que B é o Servidor, esta estratégia não acrescenta resiliência. O ideal seria que B fosse mais resiliente que A, ou que os eventos $(B \text{ falhar})$ e $(A \text{ falhar})$ fossem **independentes**.

Uma forma de manter os eventos acima referidos independentes seria ter redundância dos Nodos de níveis mais próximos ao servidor, uma vez que estes são os nodos mais críticos, isto deve ser assegurado pelo técnico de rede responsável pelo sistema.

4.5.1.2. Estratégia de mitigação implementada

From	To	ConnId	InUse	TotalHops	HopsSinceSplit	LastHopTime (ms)	TimeFromSource (ms)
B	-	[A.mp4]−1-1	True	2	1	5	12
A	-	[A.mp4]−1-2-1	False	-	-	-	-
-	C	[A.mp4]−1-1-1	True	-	-	-	-

Tabela 2: Tabela de encaminhamento do Nodo X

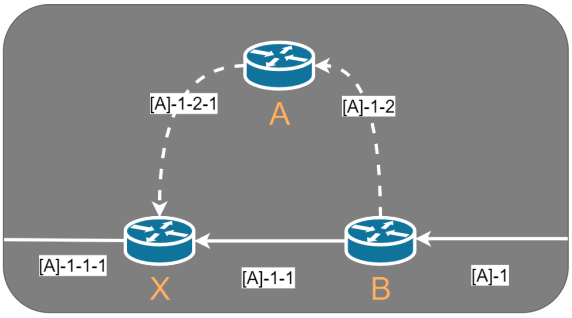

Figura 11: Representação do excerto de topologia de rede descrita pela tabela de encaminhamento

Para mitigar esta falsa impressão de resiliência, quando o nodo X deteta a morte de B, apercebe-se que `[A]-1-2-1` tem como prefixo `[A]-1`, que é "pai" de `[A]-1-1` pelo que a conexão `[A]-1-2-1` não é um bom candidato para substituir `[A]-1-1`. Tornado este caso também um **caso de pânico**.

4.5.1.3. Degeneração da rede

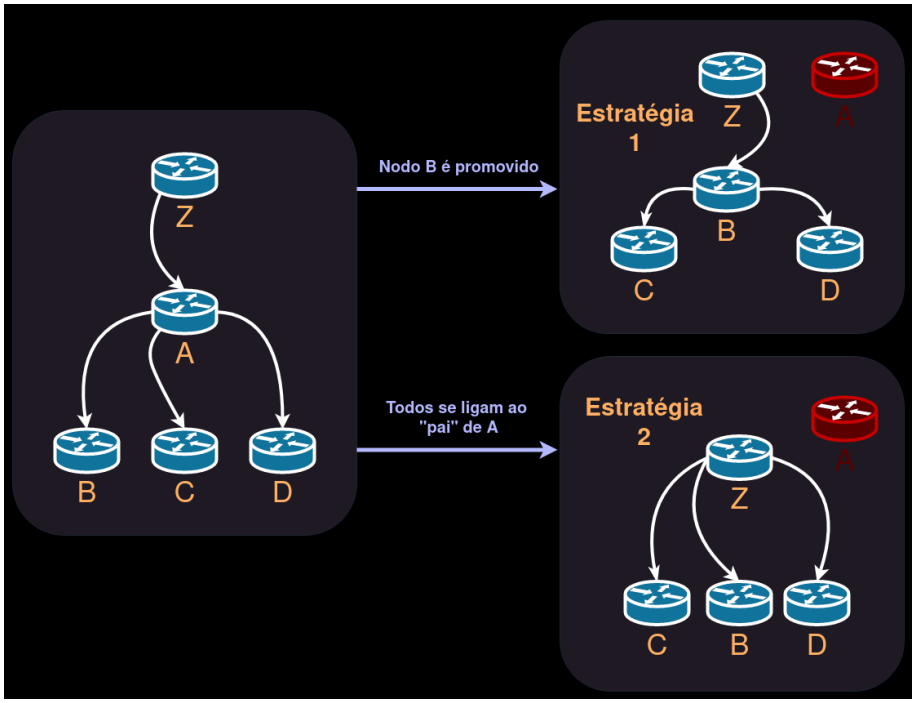

Figura 12: Ilustração do problema de degeneração da rede

Escolhendo a Estratégia 2 (delineada nos pontos acima), poderia potencialmente resultar numa degeneração da rede em que todos os Nodos têm o Servidor como contacto de emergência. Além disso, pode fazer com que o Nodo Z fique sobrecarregado com descendentes.

A Estratégia 1, onde B é promovido à posição previamente ocupada por A, seria uma melhor forma de resolver este problema.

Pensamos em implementar a Estratégia 1, fazendo com que o "contacto de emergência" no caso de falha de uma fonte (A), seja na mesma uma das suas fontes (Z), no entanto, quando Z recebe um pedido de conexão por B,C e D (por causa da falha de A), aceita "ajudar" apenas o primeiro (B), respondendo aos outros pedidos com o IP deste.

5. Conclusões

O desenvolvimento deste projeto permitiu explorar de forma prática as complexidades inerentes à conceção e implementação de um serviço *over-the-top* (OTT) para entrega de multimédia em tempo real. A arquitetura distribuída adotada revelou-se eficiente na mitigação de problemas associados à centralização, como a latência e a inconsistência de estado entre nodos. Para além disso, o uso de protocolos específicos, como o RTP para *streaming* e o protocolo de controlo sobre TCP, proporcionou uma base robusta para uma comunicação fiável e eficiente entre os elementos da rede.

Com a implementação obtida, conseguimos concretizar o envio de *N* vídeos para *N* clientes, tal como, uma implementação rudimentar da tolerância a falhas e algum dinamismo na rede.

Apesar dos avanços alcançados, identificaram-se algumas limitações na tolerância a falhas e na resiliência da rede, nomeadamente em situações de dependência cíclica entre nodos e no comportamento degenerativo em caso de falhas em cadeia. Estes aspetos apontam para a necessidade de investigações adicionais sobre estratégias de redundância e independência de eventos de falha.

Para trabalho futuro, sugere-se a implementação de mecanismos mais avançados de monitorização, recuperação de falhas e melhoria na gestão dinâmica das tabelas de encaminhamento. Adicionalmente, explorar a integração de técnicas de balanceamento de carga e otimização de latência poderia potenciar ainda mais a escalabilidade e eficiência do sistema.

Este projeto contribuiu para o entendimento das complexidades de sistemas OTT, destacando as potencialidades e os desafios no desenvolvimento de soluções robustas e escaláveis para a entrega de conteúdos multimédia.