Engenharia de Serviços em Rede

TP2

Daniel Pereira	**Francisco Ferreira**	**Rui Lopes**
PG55928	PG55942	PG56009

Introdução

Este relatório tem como objetivo apresentar o trabalho prático desenvolvido durante a unidade curricular de Engenharia de Serviços em Rede. O trabalho consistiu no desenvolvimento de um serviço *over the top* para a entrega de conteúdos multimédia em tempo real. O relatório terá como objetivo apresentar a arquitetura do sistema, a sua descrição, implementação e as decisões tomadas durante o desenvolvimento do mesmo.

Uma das liberdades dadas pela equipa docente foi a da escolha da linguagem de programação para o desenvolvimento do projeto. O grupo optou por Rust. Esta escolha permitiu-nos desenvolver um sistema muito performante e segur, sem grandes esforços.

Arquitetura

Tal como mencionado anteriormente, neste trabalho pretende-se a criação de um sistema de entrega de conteúdos multimédia em tempo real, a partir de um, ou mais, servidores de *streaming* para um dado conjunto de clientes. Este sistema deverá estar assente em cima de uma rede *overlay* própria.

Para isso, um **cliente** deverá escolher um ***point of presence***[1] como ponto de acesso à rede de entrega. Este *point of presence* estará, presumivelmente, a receber conteúdos por parte da dita rede. Sendo os mesmos conteúdos enviados, por estes *point of presence*, via *unicast* para os clientes interessados.

A chave para um bom funcionamento do sistema como um todo está na monitorização e manutenção da rede de entrega, composta por ***nodes***. Estes, devem formar, entre si, uma árvore de distribuição ótima[2] para a devida entrega dos conteúdos. Os ***nodes*** e os **point of presence** rodam a mesma aplicação, sem diferenças, só que o **point of presence** está acessível, na rede, pelos clientes.

O fluxo das *streams*, propriamente ditas, está ao encargo do (ou dos) **servidor**. Uma vez que se pretende um serviço em tempo real (e não *on demand*) parte-se do princípio de que o servidor estará sempre a enviar os *bytes* codificados das *streams* existentes e a propagar para a rede, quando necessário[3]. Estes *bytes* são, então, depois enviados para a rede de entrega, navegando através da árvore de distribuição.

As possíveis ligações de cada um dos nodos é ditada a partir do **bootstrapper**. Este contém um ficheiro de configuração com a lista de vizinhos de todos os nodos. Quando iniciado este programa, é aberto um *socket* por onde os **nodes** e os **servidores** se podem conectar, obtendo informação dos seus vizinhos. De notar que o **bootstrapper** só serve para a inicialização dos nodos/servidor e não é usado para mais nada.

De seguida, apresentamos um diagrama arquitetural genérico que tem como objetivo apresentar de que forma os diferentes componentes do sistema se dispõem e comunicam entre si.

[1] Essa escolha e a forma como é feita será abordada noutro capítulo.
[2] Baseando-se sempre num conjunto de métricas bem definidas.
[3] Por necessário entenda-se existirem clientes interessados numa determinada *stream*.

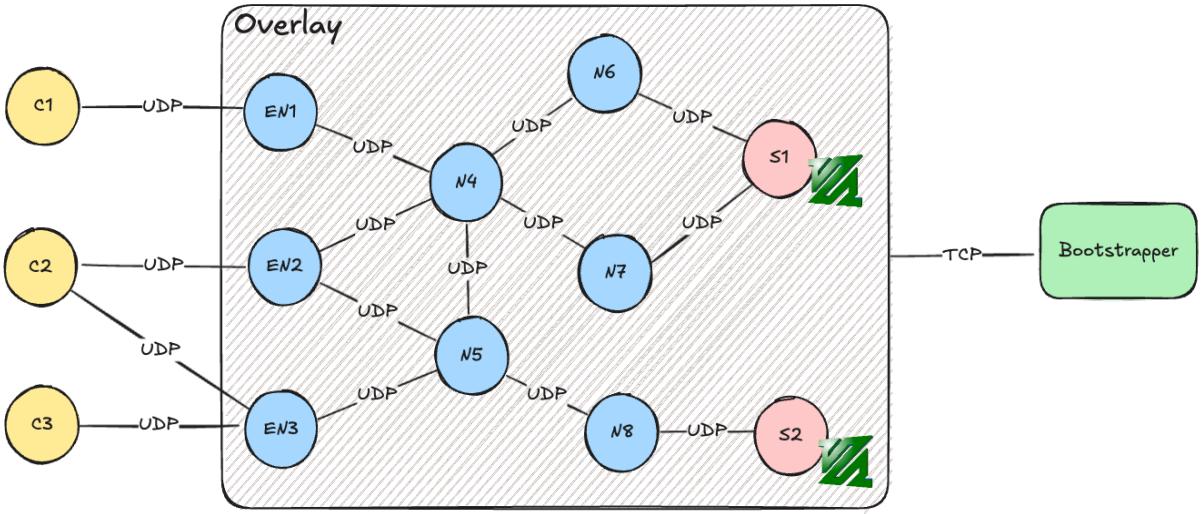

Figura 1: Diagrama arquitetural do sistema

É possível observar que o protocolo de transporte utilizado em praticamente todo o sistema é o UDP. Esta escolha baseia-se, principalmente, na performance que um protocolo como estes nos poderá dar, se em comparação com o TCP.

De facto, dentro da rede de entrega, a utilização de um protocolo como o TCP não traria qualquer vantagem, a não ser para pacotes de controlo e de metadados trocados entre os nodos. Para esses casos em que é necessário assegurar que os pacotes chegam, efetivamente, ao destino, existe a noção de pacotes *reliable*, e o seu oposto, pacotes *unreliable*. Entraremos em mais detalhe numa secção mais em baixo.

Construção da Topologia Overlay

Para a construção da topologia *overlay* o grupo decidiu optar pela abordagem baseada num controlador, o ***bootstrapper***.

Apesar desta abordagem "centralizada" para a entrada de novos *nodes*, a recuperação de falhas é feita de forma autónoma pela rede de entrega[4] e não depende, de forma alguma, do ***bootstrapper***, processo este que será abordado noutro capítulo.

Essencialmente, quando num dado nó da rede *underlay* é instanciado um programa do tipo *node* este passa a fazer parte da *overlay*. Esta entrada começa com o contacto, por parte do *node*, ao dito *bootstrapper*. Em resposta, o *bootstrapper* envia uma lista com os vizinhos do dado *node*, caso este exista na topologia para a qual o *bootstrapper* se encontra configurado no momento.

Uma vez que a comunicação realizada com o *bootstrapper* é extremamente simples e baseia-se na troca de dois pacotes, o grupo optou por utilizar TCP.

Árvore de Distribuição Ótima

Esta etapa é essencial para o bom funcionamento da rede de entrega. O objetivo aqui passa por encontrar a árvore que maximiza, em termos de algumas métricas[5], a performance da distribuição dos conteúdos na rede completa.

Para tal, o grupo decidiu utilizar um algoritmo baseado no célebre algoritmo de Prim. Dado um grafo pesado não direcionado, este algoritmo é capaz de encontrar a *minimum spanning tree*, que é um *subset* do grafo que conecta todos os nós, sem ciclos, e com o menor peso total possível.

Após a topologia *overlay* se encontrar num estado em que possa ser considerada usável, começa a etapa de construção da árvore de distribuição ótima. Esta etapa pode ser divida nos seguintes passos:

[4] Ou seja, é uma rede *self-healing*.
[5] Estas métricas serão abordadas, com mais detalhe, noutro capítulo.

1. O servidor começa por enviar, para os seus vizinhos, um pacote, nomeado como *flood*;
2. Os vizinhos recebem o pacote e registam de quem o receberam (consideram estes *nodes* como provedor de *streams*), atualizam as métricas existentes e propagam o pacote para os restantes vizinhos (exceto de quem o receberam);
3. O processo de *flooding* repete-se até ao pacote não poder ser mais propagado.

Durante o passo 2., o *node* que recebe o pacote atualiza, no seu estado interno, dados para calcular o melhor *node* a quem solicitar *streams* no futuro. Formando, assim, um dos ramos da dita árvore.

Para evitar que existam, de todo, ciclos aquando deste *flooding*, o pacote de *flood* contém uma lista com os identificadores únicos dos *nodes* por onde já passou[6]. Assim, caso um *node* receba um pacote que contém o seu identificador, descarta-o.

Finalmente, é de notar que a árvore de distribuição ótima não se irá manter, indubitavelmente, constante durante o tempo de vida da *overlay*[7] . Assim, o grupo optou por criar um sistema de monitorização da rede capaz de alterar a árvore usada em *runtime*. Este sistema irá ser abordado no capítulo que lhe concerne.

Bootstrapper

Este componente é muito simples, consistindo num programa cujo objetivo é receber conexões de *nodes*/servidores e responder imediatamente com uma lista dos respetivos vizinhos a partir do IP de contacto.

Assim, o pacote de resposta a tais conexões é o seguinte:

BootstrapperNeighboursResponse		
Campo	**Tipo**	**Descrição**
`neighbours`	`Vec<IpAddr>`	Lista com os endereços IP dos vizinhos
`id`	`u64`	Identificador único do *node*/servidor

Tabela 1: Pacote enviado do *bootstrapper* para um *node* que acabou de entrar na rede

Os detalhes de serialização dos pacotes serão descritos numa secção abaixo.

O mapeamento de um *node* para os seus vizinhos encontra-se num ficheiro de configuração TOML passado como argumento ao programa:

```toml
# neighbours.toml
"127.0.0.1" = ["127.0.0.2", "127.0.0.3"]
"127.0.0.2" = ["127.0.0.1", "127.0.0.4"]
"127.0.0.3" = ["127.0.0.1", "127.0.0.4"]
"127.0.0.4" = ["127.0.0.2", "127.0.0.3"]
```

Listing 1: Exemplo de ficheiro de configuração do *bootstrapper*

Quando o *node*/servidor recebe a resposta, pode prosseguir na sua inicialização, onde também fecha conexão com o *bootstrapper*.

[6]Este identificador único é dado pelo ***bootstrapper***. Entrando em detalhes de implementação, este identificador é apenas a *hash* do IP do *node*.

[7]Isto pois, a rede está exposta a nuances externas que podem fazer variar fatores como a latência entre dois *nodes*.

Cliente

Este componente é a ponte entre o utilizador real e o resto do sistema. O mesmo é responsável por listar e reproduzir *streams* disponíveis na rede.

O cliente recebe por parâmetro a lista de *point of presences* que se pode conectar. Quando o programa do cliente inicia, conecta-se a esses *point of presence* e aguarda pela resposta da lista de *streams* disponíveis, onde o cliente pode selecionar uma ou mais para visualizar.

Após a seleção de uma *stream* a visualizar, o cliente contacta o melhor *point of presence* para solicitar a *stream*. De realçar que é possível que um mesmo cliente visualize várias *streams* ao mesmo tempo e enviadas por *point of presence* diferentes.

Escolha do Point of Presence

A escolha de qual *point of presence* se ligar é baseada numa constante monitorização do estado da rede até aos diversos *point of presence*[8], por iniciativa do cliente.

O *point of presence* contactado aproveita para responder com a lista de vídeos disponíveis para consumo. Esta lista é enviada, desde o servidor, até aos *point of presence* através do pacote de *flood* mencionado anteriormente. Esta decisão permite que o sistema consiga lidar, facilmente, com a adição de *streams* em *runtime*, que abordaremos mais tarde.

O *point of presence* também envia o *timestamp* de quando o pacote de *ping* chegou, para possibilitar o cálculo bi-direcional do *ping*. A separação do tempo de ida e o tempo de volta só é usado para efeitos de visualização, visto que em todos os cálculos é usado o RTT.

A métrica aqui utilizada para a escolha do *point of presence* é a média do *round trip time* (RTT) num dado intervalo de tempo. Métrica que vai sendo calculada à medida que os clientes enviam *pings* para os diversos *point of presence*.

Uma vez que estamos a usar UDP, temos de lidar com possíveis perdas de pcoates. A forma como isso impacta o uso do RTT para a seleção do melhor *point of presence* será explicada numa secção abaixo.

De seguida, encontra-se o pacote enviado por parte de um cliente para cada um dos *point of presence* existentes.

ClientPing		
Campo	**Tipo**	**Descrição**
`sequence_number`	`u64`	Utilizado para fazer *match* entre o pedido e a resposta
`requested_videos`	`Vec<u8>`	Lista de vídeos pedidos atuais do cliente

Tabela 2: Pacote enviado de um cliente para um *point of presence* para monitorar a ligação

E, agora, a resposta enviada pelo *point of presence* contactado.

VideoList		
Campo	**Tipo**	**Descrição**
`sequence_number`	`u64`	Utilizado para fazer *match* entre o pedido e a resposta
`videos`	`Vec<(u8, String)>`	Lista de pares identificador da *stream*, nome da *stream*
`created_at`	`SystemTime`	*Timestamp* de criação do pacote, para a medição one-way do *ping*

Tabela 3: Pacote enviado de um *point of presence* em resposta a um ClientPing

[8] A lista de *point of presence* é passada como argumento ao programa do cliente.

Troca de Point of Presence

Quando um *point of presence* é dado como irresponsivo, todas as *streams* providenciadas por ele são reencaminhadas para o melhor *point of presence* atualmente.

Caso não existam *points of presence* que consigam providenciar o vídeo, o vídeo é colocado numa fila à espera. Na altura em que voltar a existir um *point of presence* que consiga reproduzir o vídeo (ou o *point of presence* morto voltar à vida), o cliente pede o vídeo a esse *point of presence* e continua a reprodução normal do vídeo.

Interface Gráfica

Como forma de dar uma melhor experiência ao utilizador final, o grupo decidiu desenvolver uma interface gráfica. Esta interface demonstra a lista de *streams* disponíveis para consumo, tal como opções para começar ou parar uma dita *stream*. Possui ainda informação relevante relacionada aos *point of presence* conhecidos pelo cliente.

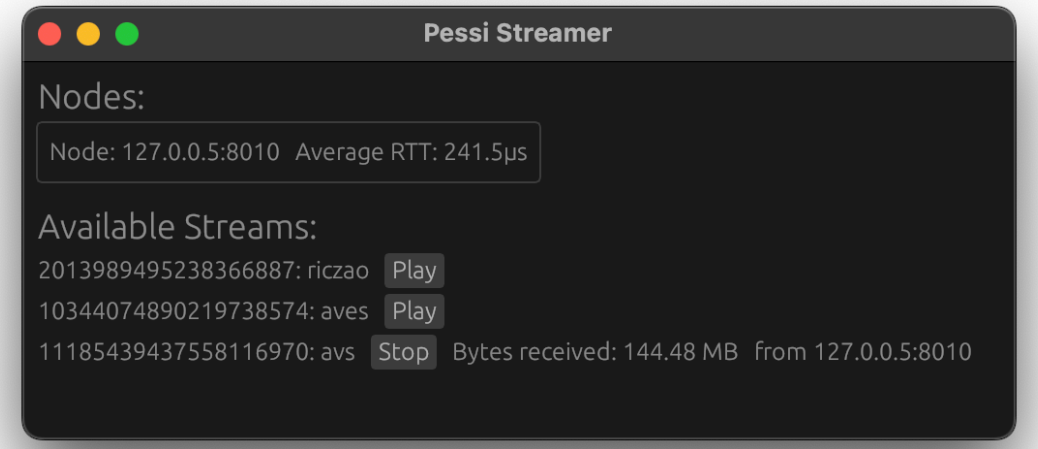

Figura 2: Interface gráfica presente no componente do cliente

A interface é responsável por criar uma instância de um *player* capaz de mostrar a *stream*, quando clicado no botão “Play”. Quando a *stream* está a ser reproduzida, o utilizador pode simplesmente fechar o *player* ou clicar no botão “Stop”, que aparecerá no mesmo lugar do botão “Play”.

Node

Este é, muito provavelmente, o componente mais importante do sistema. O mesmo tem como objetivo integrar a rede de distribuição, reencaminhado os pacotes de *streaming* necessários.

Essencialmente, este componente possui dois papéis: receber pedidos de *streams* e devolver respostas com as respetivas *streams*. Fazendo, assim, com que a árvore de distribuição ótima construída seja corretamente utilizada.

Point of Presence

Um *point of presence* pode ser entendido como um *node* que se encontra na fronteira entre os clientes e o resto da rede. São estes *nodes* que são dados a conhecer (o seu IP) aos clientes quando estes querem consumir os conteúdos multimédia.

Portanto, toda a funcionalidade entre um *point of presence* e um *node* é exatamente igual, tratando-se, efetivamente, do mesmo programa.

Quando, por parte de um cliente, um dado *point of presence* recebe um pedido de *stream* existem dois cenários possíveis:

1. O mesmo já se encontra a enviar pacotes dessa *stream* para outros clientes e, neste caso, cria um fluxo de dados e responde diretamente ao cliente solicitante;
2. O *point of presence* não possui a *stream* e, portanto, solicita ao seu melhor parente (ramo na árvore de distribuição ótima) que lhe envie a *stream*.

O processo em 2. ocorre recursivamente até que algures exista um *node* que já possui a *stream*. No pior dos casos, este *node* seria o servidor. Desta forma, todos os ramos da árvore, por onde o pedido passou, são ativados para a *stream* que foi solicitada. Esta ativação faz com que, obviamente, seja desnecessário consultar a rede toda quando uma mesma *stream* for, futuramente, solicitada por outros clientes.

RequestVideo		
Campo	**Tipo**	**Descrição**
id	u64	Identificador da *stream*

Tabela 4: Pacote enviado por parte de um cliente para iniciar uma *stream*

No caso em que um cliente deseja parar de visualizar a sua *stream* existem, também, dois cenários possíveis:

1. O *point of presence* ainda transmite a dita *stream* para outros clientes e, no caso, apenas desativa o fluxo criado para o cliente que deseja parar;
2. Não existem outros clientes interessados na *stream* e, portanto, é enviado um pedido de paragem para o seu parente.

Novamente, o processo em 2. ocorre recursivamente até que algures exista um *node* interessado na *stream*. No pior dos casos, este *node* seria o servidor.

StopVideo		
Campo	**Tipo**	**Descrição**
id	u64	Identificador da *stream*

Tabela 5: Pacote enviado por parte de um cliente para parar uma *stream*

De realçar, ainda, que tal como no caso do cliente, onde existe uma monitorização constante dos *point of presence* existentes, aqui, de certa forma, também acontece. Essencialmente, caso um *point of pre-*

sence detete que não recebe um *ping* de um cliente interessado há pelo menos três intervalos de *ping*[9], assume o cliente como morto, desativando os fluxos criados e, caso não existam outros clientes interessados nas ditas *streams*, envia um pedido de paragem para o seu parente, que se processa exatamente como mencionado anteriormente.

Servidor

O servidor é o componente responsável por gerar os pacotes de streaming e enviá-los para a rede de entrega. Este componente é o único que tem acesso direto às *streams* e é o responsável por propagar para a rede. O servidor é, também, o que envia pacotes de *flood* periodicamente, como objetivo de manter a árvore de distribuição atualizada.

Streaming

Para suporte de *streaming* de vídeos em tempo real com alta qualidade, o grupo optou pelo uso do `ffmpeg` para a conversão de vídeos em qualquer formato (automaticamente detetados pelo `ffmpeg`) para o formato `h264`, em vídeo, e `aac` em áudio. Os *bytes* gerados pelos *codecs* são encapsulados em pacotes `MPEG-TS`.

Esta instância do `ffmpeg` é criada no início do programa, para todos os vídeos presentes na pasta de vídeos. Esta pasta de vídeos é passada como parâmetro ao programa.

Ainda, para cada vídeo, é gerado um identificador numérico único, para evitar que seja necessário enviar o nome da *stream* em cada pacote de vídeo, gastando assim menos *bytes* na rede. Este identificador é gerado a partir do nome do vídeo, sendo um número de 64 bits. É de notar que é possível haver colisões de identificadores (estamos a falar de UUIDs), mas a probabilidade de tal acontecer é infinitamente baixa.

A instância do `ffmpeg` é então criada com o seguinte comando:

```
ffmpeg -re -stream_loop -1 -i "<video_path>" -c:v "<codec_video>" -b:v 8M -c:a aac -f mpegts
"<send_to_path>"
```

Listing 2: Comando para criação da instância do `ffmpeg`

A flag `-re` é usada para reproduzir o vídeo em tempo real, a flag `-stream_loop -1` é usada para repetir o vídeo indefinidamente, a flag `-i` é usada para indicar o caminho para o vídeo, `-c:v` é usado para indicar o codec de vídeo, `-b:v` é usado para indicar a *bitrate* do vídeo, `-c:a` é usado para indicar o codec de áudio e `-f` é usado para indicar o formato de saída.

O comando possui ainda vários parâmetros alteráveis:
- `<video_path>` é o caminho para o ficheiro do vídeo a ser transmitido;
- `<codec_video>` é o codec de vídeo a ser usado;
- `<send_to_path>` é o caminho para onde o vídeo é enviado.

O codec do vídeo é automaticamente detetado pelo programa, onde é sempre escolhido um codec `h264` que use sempre aceleração de hardware, caso esteja disponível. O programa executa o comando `ffmpeg -encoders` para obter a lista de codecs disponíveis e escolhe o primeiro codec presente nesta lista: “hevc_videotoolbox” (aceleração macOS), “h264_nvenc” (aceleração NVIDIA), “hevc_amf” (aceleração AMD), “h264” (aceleração CPU), “libx264” (aceleração CPU).

O `<send_to_path>` é o caminho para onde o vídeo é enviado. Este caminho será um *socket* UDP que o servidor cria para receber os pacotes de streaming. Desta forma, o servidor recebe os pacotes de vídeo já particionados, para serem enviados via UDP, respeitando o MTU, não tendo de fazer trabalho para acumular *bytes* do vídeo. A cada pacote recebido, o servidor envia o pacote encapsulado com a identificação da *stream* para os *nodos* que pediram a *stream*.

[9] Valor atualmente definido para 3 segundos, 1 segundo por cada *ping*, mas facilmente configurável.

Múltiplos servidores

A nossa arquitetura também possibilita a existência de múltiplos servidores. Cada servidor é responsável por um conjunto de vídeos, podendo ser o mesmo conjunto de vídeos que outro servidor ou não. Como cada *node* tem a lista de vídeos que pode reproduzir, ao receber o pacote de *flood* de um segundo servidor, apenas adiciona os vídeos desse servidor à sua lista de vídeos, e que pode reproduzir aquele vídeo a partir do *node* que lhe enviou o pacote de *flood*.

Adição de vídeos em runtime

Como funcionalidade extra, o grupo decidiu implementar a capacidade de adicionar *streams* durante a execução do servidor, eliminando a necessidade de interromper qualquer transmissão que esteja a decorrer para tal.

Esta funcionalidade foi integrada de forma natural, uma vez que o servidor já propagava para a rede quais as *streams* disponíveis aquando do *flooding*. *Flooding* este que se processa de forma contínua no tempo, dando aso ao sistema de monitorização da rede que será abordado com mais detalhe num próximo capítulo.

Monitorização da Rede

A grande vantagem de possuir um sistema de monitorização da rede contínuo no tempo é o facto da árvore de distribuição ótima se ir alterando de acordo com as condições da rede. Além disso, esta monitorização permite-nos, também, a deteção de falhas por parte de *nodes* integrantes da rede de entrega. Esta deteção irá ser abordada com mais detalhe num próximo capítulo.

Ainda, o grupo chegou a uma solução tão simples quanto um pacote, apelidado de *flood*, que é enviado periodicamente para a rede com origem no servidor. Este pacote é enviado para cada um dos vizinhos de cada *node* até chegar aos *point of presence*.

Este pacote contém informação relevante para a monitorização da rede, como o número de saltos que o pacote já deu, o *timestamp* de criação do pacote no servidor para cálculo do delay, a lista de vídeos disponíveis para consumo e a lista de *nodes* por onde o pacote já passou.

FloodPacket		
Campo	**Tipo**	**Descrição**
`created_at_server_time`	`SystemTime`	*Timestamp* de criação do pacote no servidor
`videos_available`	`Vec<(u64, String)>`	Lista de pares identificador da *stream*, nome da *stream* de vídeos que o nodo que enviou este pacote tem disponíveis
`visited_nodes`	`Vec<u64>`	Lista de identificadores únicos dos *nodes* por onde o pacote já passou
`my_parents`	`Vec<SocketAddr>`	Lista de endereços dos *nodes* pais do *node* que enviou o pacote

Tabela 6: Pacote enviado pelo servidor para a rede para monitorização

Este pacote é propagado para os vizinhos do *node* que o recebeu, exceto para o *node* que enviou o pacote. O pacote não é propagado caso o identificador do *node* atual já esteja na lista de `visited_nodes`, para evitar ciclos.

Fiabilidade com UDP

Uma vez que o grupo decidiu utilizar o protocolo de transporte UDP em praticamente todo o sistema, tornou-se desafiante a implementação de uma solução suficientemente genérica e performante.

Para tal, o grupo optou por implementar uma abstração em cima do *socket* UDP. Esta abstração tem como noção a de pacotes *reliable* e *unreliable*. Todas as entidades do sistema (exceto o *bootstrapper*) usam esta abstração para comunicação.

Os pacotes *reliable* são pacotes que são enviados e que o emissor espera uma resposta, um *Ack*. Caso não receba esse *Ack* num tempo determinado[10], o pacote é reenviado, até um número de tentativas[11], onde caso esse número seja atingido, o pacote é considerado perdido e o emissor do pacote é notificado.

Os pacotes *unreliable* são pacotes que são enviados e esquecidos, como é no UDP.

Todos os pacotes, exceto os pacotes de vídeo, são enviados de forma *reliable*. Como o protocolo *mpeg-ts* é resiliente a falhas, não é necessário retransmitir pacotes de vídeo, sendo assim então enviados de forma *unreliable*.

Algoritmo de Escolha

Tanto os clientes quanto os *nodes* precisam considerar qual é o melhor *node* para contactar ao solicitar os pacotes de uma *stream*.

[10] O valor determinado para este tempo foi de 100ms, para haver um bom equilíbrio entre rapidez de resposta e evitar uso desnecessário da rede.

[11] Valor determinado de 5 tentativas, pelas mesmas razões da escolha anterior.

Para isso, é levado em conta o RTT entre os vários vizinhos (ou *point of presence* no caso do cliente). Caso existam falhas na rede, como o pacote será retransmitido, o RTT será maior, e então o vizinho dificilmente será escolhido como melhor nó.

Desta forma, nos *nodes* o melhor *node* é escolhido a partir destes critérios:
1. Escolher os *nodes* que a média dos últimos 10 RTTs esteja entre 30% da menor média de RTT;
2. Entre estes *nodes* escolher o que tem menos *hops* até ao servidor;
3. Se o número de *hops* for igual, escolher o que tem menos vídeos pedidos atualmente;
4. Se o número de vídeos pedidos for igual, escolher o que tem menos *streams* que pode enviar.

No caso dos clientes, é o melhor *point of presence* é escolhido a partir dos mesmos críterios, excluindo o passo 3. já que os clientes não têm acesso ao número de *hops* que um *point of presence* tem até ao servidor.

Recuperação de Falhas

Tanto o cliente como os *nodes* rodam uma tarefa de provisionamento de conexões.

Nos *nodes*, quando um vizinho lhe envia um pacote de *flood*, esse vizinho é considerado como um *node parent*. A partir deste ponto, é esperado que o *node parent* lhe envie periodicamente pacotes de *flood*.

Caso um *node* não receba um pacote de *flood* de um *node parent* durante um intervalo de tempo determinado[12], o *node* considera o *node parent* como `Unresponsive`, e são feitos os procedimentos esclarecidos nas secções seguintes, de acordo com o tipo de mortes.

Caso um *node* não faça *Ack* do pacote de *flood* enviado, o *node* que enviou o pacote também considera o *node* que não fez *Ack* como `Unresponsive`.

Caso um *node* não receba um pacote de vídeo que era suposto receber, em 500ms, tal como o cliente, o *node* reenvia o pacote de pedido de vídeo. Isto serve para garantir que, caso o *node* que lhe está a enviar o vídeo tenha reiniciado ou perdido o estado, o *node* que pediu o vídeo consiga recuperar o estado.

Mortes

Para exemplificar o procedimento em caso de mortes simples e mortes complexas, usaremos esta tipologia como exemplo:

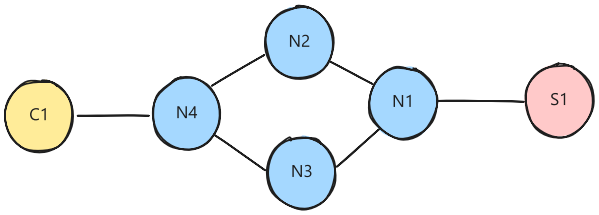

Figura 3: Topologia de exemplo

Nesta topologia, assume-se que o cliente C_1 está a pedir uma *stream* ao servidor S_1 a partir dos *nodes* $N_4 \rightarrow N_3 \rightarrow N_1 \rightarrow S_1$.

Mortes Simples

Assumindo agora que o N_3 morre, o N_4 vai detetar que o N_3 não lhe está a enviar pacotes de *flood* e considera-o como `Unresponsive`. O N_4 vai então pedir ao N_2 para lhe enviar a *stream* que estava a receber do N_3. O N_2 vai então pedir ao N_1 para lhe enviar a *stream* que estava a receber do N_3. O N_1 como já estava a receber a *stream* do S, vai então começar a enviar a *stream* para o N_2, que vai enviar para o N_4, que vai enviar para o cliente. O N_1 também vai detetar que o N_3 não lhe está a enviar pacotes de *Ack* para o pacote de *flood*, considera-o como `Unresponsive` e para de lhe enviar os pacotes de vídeo.

[12] Valor determinado de 3 segundos, um equilíbrio entre deteção rápida para evitar interrupções nos vídeos do cliente e não marcar *nodes* não suficientemente lentos como mortos.

Mortes Catastróficas

Partindo do estado final mostrado na secção anterior, onde o N_3 está morto, podemos agora matar o N_2 para simular uma morte complexa.

Matando o N_2, o mesmo procedimento acontece no N_1 descrito anteriormente, mas agora o N_4 não tem vizinhos disponíveis para pedir o vídeo. Como o N_4 sabe que os parentes do N_3 é o N_1, o N_4 irá enviar o pacote de `NewNeighbour` para o N_1. O N_1 irá adicionar o N_4 à lista dos seus vizinhos. O N_4 adiciona as *streams* perdidas a uma lista de *streams* pendentes. Quando o N_1 receber um pacote de *flood* do S_1, agora como o N_4 é seu vizinho, irá enviar-lhe o pacote de *flood*. O N_4 recebendo esse pacote de *flood* vai então pedir a *stream* ao N_1 e o N_1 vai enviar a *stream* para o N_4.

Agora, se o N_1 morrer também, o N_4 tem a lista de parentes do N_1 a partir do pacote de *flood* e então poderá repetir o mesmo algoritmo.

Caso um *node* morto volte à vida, os *nodes* vizinhos detetam que ele voltou à vida e declaram-no como `Responsive` novamente. Nenhum vídeo é reencaminhado e nenhuma conexão é refeita. No exemplo anterior, se o N_2 voltar à vida, o N_1 continuará a mandar pacotes de *flood* para o N_4 visto ainda pertencer à lista de vizinhos dele. Só no caso de algum *node* morrer ou algum cliente pedir um novo vídeo, é que o *nodo* que voltou à vida pode ser usado.

Adições de vizinhos em *runtime*

Caso um *node* não inicie corretamente, os vizinhos desse *node* declararão o *node* como `Unresponsive`. O *node* assim que iniciar, irá reencaminhar o pacote de *flood* para os vizinhos, que irão declarar o *node* como `Responsive` novamente, podendo assim pertencer à rede normalmente.

Segurança na Rede

Um fator que o grupo não explorou o suficiente foi a segurança na rede. A segurança é um fator muito importante num cenário real, onde pacotes podem ser facilmente intercetados e alterados. A arquitetura é altamente baseada nos IPs que vão nos pacotes UDPs e isto é um problema de segurança, já que estes podem ser facilmente alterados, a partir de um agente malicioso.

Num cenário real, seria necessário implementar um sistema de autenticação e encriptação dos pacotes, para garantir que os pacotes são enviados por quem dizem ser e que não são alterados durante o envio. Isto poderia ser feito a partir da migração do uso de MPEG-TS para o HLS, em que a *stream* seria enviada a partir de HTTPS, garantindo a autenticação e encriptação dos pacotes. Entre os *nodes*, o mesmo HTTPS poderia ser usado, ou então um sistema de autenticação e encriptação próprio.

Como esta UC não engloba estes temas, o grupo não entrou muito a fundo neste tópico. No entanto, existe uma *branch* (não contendo os *commits* mais recentes) onde os pacotes são encriptados a partir de uma chave simétrica partilhada entre os *nodes*. Não incluímos na solução final porque não é resiliente a *replay attacks*, não tem autenticação nem integridade, não tem *forward secrecy*, e só prejudicaria a performance do sistema com pouco benefício ganho.

Protocolo e serialização de pacotes

Para a comunicação entre os diversos componentes do sistema, foi necessário optar por um protocolo de comunicação extremamente eficiente e de fácil implementação. Para tal, o grupo decidiu utilizar a *crate*[13] bincode, que é uma biblioteca de serialização binária para Rust. Esta *crate*, juntamente com o *serde*[14], permitiu com que o código de serialização e desserialização de pacotes fosse gerado em *compile-time* a partir das definições dos *enums* e *structs*.

A especificação da serialização de *structs* e *enums* para *bytes* com vários exemplos está presente em https://github.com/bincode-org/bincode/blob/trunk/docs/spec.md.

Todos os pacotes estão definidos no ficheiro pessi-streamer/common/src/packet.rs. Estes pacotes, como são anotados pelo *macro* `#[Serialize, Deserialize]`, será gerado o código de serialização e desserialização automaticamente.

Estes pacotes, como são enviados a partir da abstração do *socket* UDP, serão ainda encapsulados no pacote definido em: pessi-streamer/common/src/reliable.rs#L41.

Isto permite-nos ter um protocolo binário extremamente eficiente com muito pouco esforço.

Conclusão

Para concluir, o grupo desenvolveu um sistema de distribuição de *streams* multimédia em tempo real, com capacidade de adição de *streams* em *runtime*, monitorização da rede, recuperação de falhas e uma interface gráfica para o utilizador final. As *streams* são reproduzidas em altissima qualidade. O sistema é altamente escalável e performante, com a possibilidade de adicionar múltiplos servidores e *point of presence*.

A escolha da linguagem Rust foi uma escolha muito acertada. Seja pela facilidade da serialização de dados num formato binário eficiente, seja pelo sistema de tipos concreto que faz com que se evite erros comuns durante o desenvolvimento ou seja pela facilidade de programar certos tipos de *patterns* de programação concorrente, o grupo irá continuar a usá-la em projetos futuros.

Com isto, o grupo achou que este trabalho foi um sucesso, tendo cumprido todos os objetivos propostos e tendo ultrapassado as expectativas iniciais.

[13] O termo *crate* é usado em Rust para referir uma biblioteca.
[14] O *serde* é uma *crate* que permite a serialização e desserialização de dados para formatos genéricos em Rust.