November 24, 2024

Relatório ESR - Fase 2 - Grupo 25

Ivan Ribeiro (pg55950)
Duarte Ribeiro (pg55938)
Diogo Marques (pg55931)

Sumário

1) Introdução

Este relatório centra-se no desenvolvimento de um serviço de *streaming* em tempo real com recurso a *Multicast* a nível aplicacional. Iremos abordar a arquitetura da solução, as decisões tomadas na mesma, e a consequente implementação e resultados da mesma.

2) Arquitetura da solução

De modo a cumprir todos os requisitos necessários, a arquitetura precisava de algum grau de complexidade, no entanto de se manter relativamente simples de implementar.

2.1) Arranque dos elementos da rede

De modo a um elemento poder fazer parte da rede, tanto para gerar, reproduzir ou transmitir vídeo, é necessário que este conheça quais são os dispositivos aos quais se tem de conectar. Sendo esta uma rede relativamente estática, onde conhecemos previamente com quem devemos falar é possível definir as ligações da rede, que depois os membros da rede poderão então consumir quando são ligados.

Para esse efeito, optamos pela criação de um nó **Bootstrapper**, que irá ter acesso a um ficheiro *JSON* onde estão detalhadas todas as possíveis ligações de todos os possíveis membros da rede. Quando um novo dispositivo é iniciado, contacta o *Bootstrapper* (cujo IP é fornecido como argumento), de modo a conhecer todos os seus possíveis vizinhos, e tenta iniciar uma conexão com os mesmos.

2.2) Servidor

Recebendo como argumentos de inicialização o IP do *Bootstrapper*, a pasta onde os conteúdos de *stream* se encontram, e o seu nome (de modo a informar o Bootstrapper quem é), o servidor, depois de contactar o Bootstrapper, irá conectar-se aos seu nós vizinhos e começar a propagar pacotes de *flood* com os conteúdos que ele possui, assim como uma timestamp de quando enviou esse pacote.

Quando um nó vizinho ao servidor lhe pede uma *stream*, este transmite a mesma para esse nó.

2.3) Nó

2.3.1) Inicialização

Recebendo como argumentos de inicialização o IP do *Bootstrapper* e o seu nome, o nó começa por contactar o **Bootstrapper** de modo a saber quais são os seus vizinhos e tenta conectar-se aos mesmos. Caso algum nó não esteja contactável é assumido que o mesmo simplesmente ainda não foi inicializado.

2.3.2) Ligação

Optamos por escolher TCP para todas as ligações entre nós (e entre nós e servidor). Realizamos esta escolha pois estas ligações não mudam frequentemente nem são breves, logo o *overhead* de estabelecer conexão é relativamente insignificante, e o controlo de falhas que este tipo de conexão nos confere simplifica bastante a implementação do sistema, ainda que haja uma pequena perda de performance em comparação com UDP.

2.3.3) Pacotes de Controlo

2.3.3.1) Receção

Após o nó inicializar as conexões com os vizinhos, assumindo que existe um servidor ativo e os nós até ao mesmo estejam ativos, este nó começará a receber as mensagens de *flood* do

servidor através dos seus nós vizinhos (ou do servidor se for vizinho do mesmo). Este nó irá então utilizar a *timestamp* presente na mesma para calcular o tempo que este pacote demorou a chegar do servidor até si, e registar este tempo numa tabela, por via de média ponderada[1]. O nó irá escutar por este tipo de pacotes de todos os seus vizinhos e registar o tempo médio do servidor até si por todos os caminhos que receba esses pacotes (possíveis nós pai).

2.3.3.2) Transmissão

Depois de registar o tempo de chegada do pacote, o nó envia então o mesmo para todos os seus vizinhos, exceto o vizinho que lhe enviou este mesmo pacote, propagando-o e permitindo que este seja avaliado como um possível pai pelos seus vizinhos. No entanto, esta abordagem cria um problema óbvio. Se eu tenho O1,O2 e O3 ligados em triângulo, e O1 receber um pacote, irá propagá-lo para O2[2], que o propagará para O3, que consequentemente o propagará para O1 visto que não o recebeu de O1, e O1 o propagará para O2 pela mesma razão, e por aí em diante, criando um ciclo infinito de propagação do mesmo pacote ente nós da rede.

Para além da *timestamp* e do nome do servidor que inicialmente criou o pacote, este obviamente possui as *streams* que o servidor transmite, mas também todos os nós intermédios por onde este pacote já passou. Deste modo, se um nó recebe um pacote com a sua assinatura, quer dizer que já o propagou no passado e não o vai voltar a propagar, parando aí o ciclo de transmissão.

2.3.4) Transmissão de conteúdo

2.3.4.1) Requerer conteúdo

Quando um cliente manifesta querer uma *stream* a um *edge node*, este consulta a sua tabela de possíveis pais e seleciona o seu vizinho mais rápido a alcançar o servidor com esse conteúdo. De seguida, envia a esse servidor um pedido desse conteúdo. O nodo vizinho, que para este conteúdo em específico passou a ser seu pai, realiza o mesmo processo para obter o seu pai, subindo na árvore até a um nodo vizinho do servidor, que lhe pede o conteúdo. A *stream* é então propagada no sentido inverso dos pedidos, até ao *edge node*. Este finalmente entrega o conteúdo ao cliente que o requisitou.

Se a algum momento algum nó intermédio, pelas estatísticas obtidas na receção de mensagens de *flooding*, determinar que existe um caminho mais rápido o suficiente, poderá trocar o seu pai para essa determinada *stream*. Deste modo, a árvore de transmissão de vídeo altera-se dinamicamente consoante as condições de rede.

2.3.4.2) Terminar transmissão

Quando um cliente termina de ver a transmissão, envia um pedido de cancelamento ao seu *edge node*. Este nó irá então avaliar se ainda existe outro consumidor desse conteúdo a quem o deve continuar a fornecer. Se for esse o caso, apenas deixa de transmitir a *stream* ao cliente que já não o deseja. Caso contrário, enviará um pedido ao seu nó pai a pedir o término de transmissão para si. Este nó pai irá realizar a mesma avaliação que o seu filho, e realizar as mesmas ações para as mesmas condições. Esta cadeia de cancelamento continuará enquanto este cliente for o único consumidor que dependia desse nó para fornecer conteúdo.

2.3.5) Morte Simples

Quando algum nó morre inesperadamente, os seus vizinhos são de certa forma automaticamente notificados, dada a natureza das ligações TCP, que terminam imediatamente a conexão no lado

[1] Atualmente cada novo pacote tem um peso de 20% sobre a métrica de tempo médio e o valor existente 80%

[2] E para O3 também, mas foi ignorado neste exemplo por questão de simplicidade

de leitura em caso da morte do lado de escrita. Sendo assim, quando um nó morre, todos os vizinhos têm conhecimento imediato desse facto, o que lhes permite descartar esse nó como válido. Dada a nossa constante monitorização da performance de outros nós vizinhos, se este nó nos fornecia conteúdo conseguimos facilmente pedir de novo o conteúdo em questão, agora ao (até então) 2º melhor nó vizinho.

Deste modo, as mortes simples são quase automaticamente mitigadas pela forma que a rede está desenhada, ao custo da perda de no máximo alguns segundos da *stream* (no caso deste cliente ser o único consumidor neste ramo, pois a cadeia de distribuição até ao servidor muito provavelmente terá de ser totalmente reposta).

2.3.6) Morte Catastrófica

No caso da rede não possuir caminhos alternativos numa secção, o cenário da morte dessa ligação única é considerada complexa, pois o fluxo de dados não pode simplesmente ser reposto por outro caminho existente.

Quando um nó deteta que se encontra numa situação de risco de morte complexa, ou seja, quando apenas recebe mensagens de controlo para um dado conteúdo por apenas um nó vizinho (o que implica que a morte desse dado vizinho corta o acesso ao conteúdo), pede a esse mesmo vizinho quem são os pais deste que lhe fornecem o conteúdo procurado (grandfather request). Na resposta virão todos os avós para esse conteúdo. No caso do pai morrer para esse conteúdo, o nó filho irá então ligar-se a todos os avós que partilhem esse conteúdo, essencialmente substituindo o antigo pai na rede.

2.3.6.1) Morte de avós após Grandfather Request

Esta abordagem, de informar quais são os avós no momento que a ligação tem um elo único cria um problema. Imaginando que existe uma secção do overlay A->B->C->D, sem caminhos alternativos, e por causa disso tanto o nó C e D pediram os seus avós e obtiveram A e B respetivamente. Num cenário onde o nó B é o primeiro a morrer, C conecta-se a A e a rede funciona normalmente. No entanto, em caso da morte de C, D tentará conectar-se a B, mas não irá conseguir porque este nó está morto e a rede deixa de funcionar.

A solução encontrada para este problema foi que sempre que um nó pai morre, o filho avisa os netos que houve uma mudança nos nós avós.

2.3.7) Adição Complexa (A qualquer momento)

Ao contrário de um cenário onde todos os nodos são inicializados antes de qualquer transmissão de vídeo acontecer, a adição complexa é quando alguns nós da overlay não são inicializados imediatamente, mas sim num momento arbitrário. É de esperar que depois da adição, se este nó se relevar como um caminho mais rápido para um determinado destino, que o conteúdo passe a fluir por este nodo.

Na nossa implementação esta funcionalidade é conseguida devido ao flooding constante da rede, que permite que um caminho que até então nao existia passe a ficar registado, e se o mesmo for mais rápido a monitorização do tempo que provém da partilha desses pacotes revele esse caminho como mais rápido.

2.4) Cliente

Após o cliente arrancar e pedir ao bootstrapper os edge nodes alcançáveis, o cliente começa imediatamente a medir a qualidade da conexão aos seus edge nodes. O utilizador poderá então escolher o edge node de onde quer o seu conteúdo, tendo em conta a qualidade da ligação.

Depois de escolher o edge node, receberá uma lista dos conteúdos disponíveis que poderá então selecionar. O conteúdo será então transmitido pelo servidor, recebido pelo cliente e reproduzido no mesmo, até ao cliente parar a transmissão. É possível parar a transmissão e trocar o conteúdo a ser reproduzido.

3) Especificação dos protocolos

3.1) Dispositivos - Bootstrapper - TCP

Entre qualquer dispositivo e o bootstrapper escolhemos usar TCP para as comunicações. Visto que esta ligação apenas é realizada uma vez no startup do dispositivo, o overhead desse estabelecimento não é muito relevante, e a adoção do TCP simplifica a implementação do protocolo.

3.1.1) BootstrapperPacket

Todas as comunicações de e para o bootstrapper são feitas usando este pacote, que possui:

- Nome do dispositivo
- JSONObject

O nome do dispositivo é utilizado no pedido, de modo ao mesmo se poder identificar, enquanto o JSONObject contém um campo com todos os nós vizinhos/edge nodes acessíveis a este dispositivo. Optamos pela utilização de um JSONObject para este efeito porque esta informação é armazenada num ficheiro JSON e lida pelo Bootstrapper, e isto permite utilizar o objeto original como resposta sem necessidade de manipulação de dados extra, simplificando o desenvolvimento. A utilização de um JSONObject também permite a adição de campos extra no futuro se necessário sem alteração do código existente.

3.2) Overlay - TCP Encriptado

Pareceu-nos a escolha correta usar TCP para as conexões entre os nós do overlay, visto que este abstrai preocupações de controlo de transmissão que simplifica o desenvolvimento, e porque permite ter noção de conexão - podemos então usar a quebra de conexão como sinal de morte de um vizinho.

Por segurança, decidimos também implementar encriptação das comunicações no overlay com uma simples chave AES. Isto previne, ainda que de forma muito rudimentar, que um ator mal intencionado se insira artificialmente na rede Overlay e se faça passar por um nodo, alterando maliciosamente as mensagens de flood e/ou alterando o conteúdo de streaming que a atravessa. Este tipo de ataques fica assim prevenido, enquanto a chave se mantiver secreta claro. É sempre possível implementar um sistema mais robusto onde a chave não é estática, no entanto sentimos que isso está fora do escopo deste trabalho.

3.2.1) FloodControlPacket

Os pacotes usados para flood, como referido acima, têm a seguinte estrutura:

- Nome do servidor: string usada para identificar cada servidor, de modo a permitir vários servidores a fornecer diferentes vídeos
- Timestamp: criado pelo servidor, representa instante em que o flood se inicia
- Lista de visitados: lista de strings (nomes do servidor), em que se registam os nodos por onde o pacote passou
- Lista de vídeos: lista de strings com os nomes dos vídeos disponíveis no servidor que originou este flood

Estes são enviados de cada servidor num intervalo de tempo fixo, atualmente de 10s, e são propagados pela rede.

3.2.2) GrandfatherControlPacket

Pacote usado para pedir/enviar listas de avós de um nodo, com o seguinte formato:

- Tipo: Enum (int) que identifica como Request ou Reply
- Nome do vídeo: para que se possa responder com os avós que têm disponível um determinado vídeo
- Lista de avós: lista de IPs referentes aos avós

3.2.3) VideoControlPacket

Este pacote permite obter funcionalidades de controlo do vídeo. Assim, foi estruturado da seguinte forma:

- Tipo: pode ser Request (pedir vídeo), Cancel, Reply (dados do vídeo)
- Video: ID do vídeo (neste caso o nome)
- Dados: byte[] com dados do vídeo, quando aplicável

3.3) Edge Node - Cliente - UDP

Para as comunicações entre o cliente e o edge node, decidimos utilizar UDP, pois deixa de haver overhead de conexão, controlo de transmissão e as respetiva penalizações em performance. Para além disto, na transmissão do video em si, a retransmissão não faz sentido visto ser um video real-time e o facto de ser escolhido um encoding que permite aguentar perdas.

3.3.1) VideoControlPacket

Usado para controlar quais vídeos o cliente recebe, podendo ser usado para pedir e parar (cancelar) o video.

Assim, este pacote tem apenas o tipo (Request ou Cancel) e o nome do video.

3.3.2) VideoListPacket

Usado para pedidos de lista de vídeos existentes:

- Tipo: Request ou Reply
- Video: ID do vídeo
- Lista de vídeos: lista de strings com os vídeos disponíveis para o nodo que recebeu o pedido

3.3.3) UDPPacket

Os pacotes UDP possuem um Tipo e um ID, de forma a estabelecer uma relação entre cada pacote UDP e o seu respetivo ACK.

3.3.4) LinkPacket

Sendo construído em cima do UDPPacket, este pacote é enviado do cliente para o edge node de forma a verificar a latência e a loss. Para tal, o cliente verifica quanto tempo e quantas tentativas foram necessárias ate que o ACK respetivo fosse recebido.

3.3.5) AckPacket

Usado para confirmar a receção de um UDPPacket.

3.3.6) VideoPacket - ffmpeg

Este pacote não é gerido por nós, mas apenas representa aquilo que é gerado pelo streaming do ffmpeg, a ser enviado do edge node para o cliente.

4) Implementação

4.1) Escolha de linguagem

Um momento crucial para o desenvolvimento deste projeto é a escolha de linguagem para a implementação do projeto. A equipa estava indecisa entre C++, Python e Java, as linguagens que cada um dos membros se sentia mais confortável. Acabamos por escolher Java pois é uma linguagem muito completa, com várias bibliotecas que nos seriam úteis e simplificariam o desenvolvimento do projeto.

4.2) Bootstrapper

Usado para o setup inicial da rede. Os seus dados estão armazenados num ficheiro json com o seguinte formato:

```
{
    "server1": { // servidor precisa de saber vizinhos (filhos)
        "neighbours": [
            "10.0.4.1",
            "10.0.3.1"
        ]
    },

    "client1": { // cliente precisa de saber edge nodes
        "neighbours": [
            "10.0.0.1"
        ]
    },

    "O1": { // nodos precisam de saber os vizinhos (pais e filhos)
        "neighbours": [
            "10.0.4.10",
            "10.0.5.2",
            "10.0.8.1"
        ],
        "edge": false // podemos tambem especificar se se trata de um edge node

    },
    ...
}
```

Através deste, ao iniciar os nodos, estes conseguem ter noção dos nodos que existem a sua volta, e dos endereços IP aos quais se devem tentar conectar.

4.3) Servidor

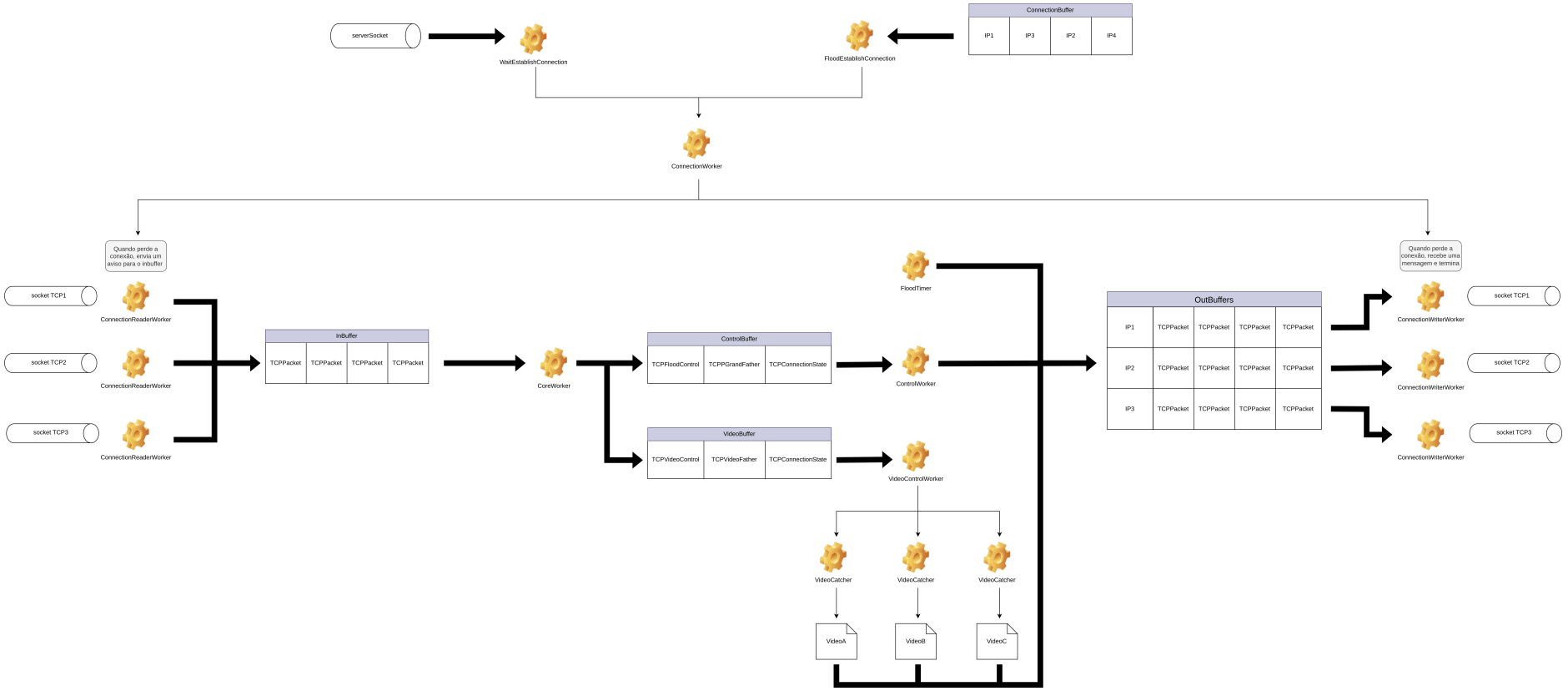


Figura 1: Diagrama de funcionamento do Servidor

4.3.1) Startup

Ao iniciar, o servidor irá conectar-se ao IP do Bootstrapper (dado como argumento do programa) e ficar a conhecer os nodos vizinhos e conecta-se aos mesmos. Analisa quais são os vídeos presentes na pasta passada como argumento e regista-os. Começa então a enviar pacotes de *flood* a informar que possui esses vídeos em intervalos constantes.

4.3.2) Transmissão de vídeo

Assim que recebe um pedido de conteúdo, o servidor inicia a transmissão do vídeo em questão. **A *stream* de todos os conteúdos não é iniciada automaticamente visto a plataforma CORE é executada num ambiente virtualizado com recursos limitados, o que iria causar problemas com vários vídeos na pasta**. No entanto é importante frisar que **após um vídeo ser pedido pela primeira vez, o servidor não para de o ler, sendo o playback do mesmo contínuo após o primeiro pedido**. No entanto, alterar a implementação para o servidor começar a ler os vídeos no *startup* para um caso de uso mais realista é bastante simples.

A leitura do vídeo para transmissão é realizada com recurso ao ffmpeg. Este é iniciado e lê o conteúdo do vídeo[3] para o stdout. O conteúdo do stdout deste processo é então capturado pelo servidor e enviado para os nós que o desejem.

4.4) Nó

4.4.1) *Startup*

Quando um nó liga, contacta o bootstrapper para saber os seus vizinhos e conecta-se aos mesmos, se estes estiverem disponíveis. Se existe uma ligação pela overlay desde um servidor até este nó, este começará a receber pacotes de flood do mesmo e a criar tabela de melhores pais.

4.4.2) *Streaming*

Quando o nó recebe de um nó vizinho um pedido de conteúdo, procura na tabela qual o seu melhor pai para esse conteúdo e pede-lhe o mesmo. Esse pai realiza o mesmo processo, o avô e todos os antecessores também até ao servidor. O conteúdo flui então em sentido inverso, do servidor ao nó em questão, que depois entrega o conteúdo ao nó que o requisitou. Se outro nó vizinho entretanto requere o mesmo conteúdo, visto que este nó já o está a receber, apenas precisa encaminhar o conteúdo que já recebe para outro nó.

No entanto para isto ser possível é necessário o nó contabilizar quantos nós estão a pedir um dado conteúdo. Isto porque quando um dos nós já não deseja o conteúdo temos de apenas cancelar a entrega de conteúdo para esse nodo visto que outro filho ainda precisa do conteúdo. Só quando todos os nós filho deixam de precisar do conteúdo em questão é que este nó informa o seu pai que já não precisa do conteúdo.

4.5) Edge Node

Um edge node funciona de forma semelhante a um nó, mas com a capacidade de interagir com clientes e entregar-lhes conteúdo.

[3]Não realiza playback do vídeo mas sim lê os bytes do mesmo

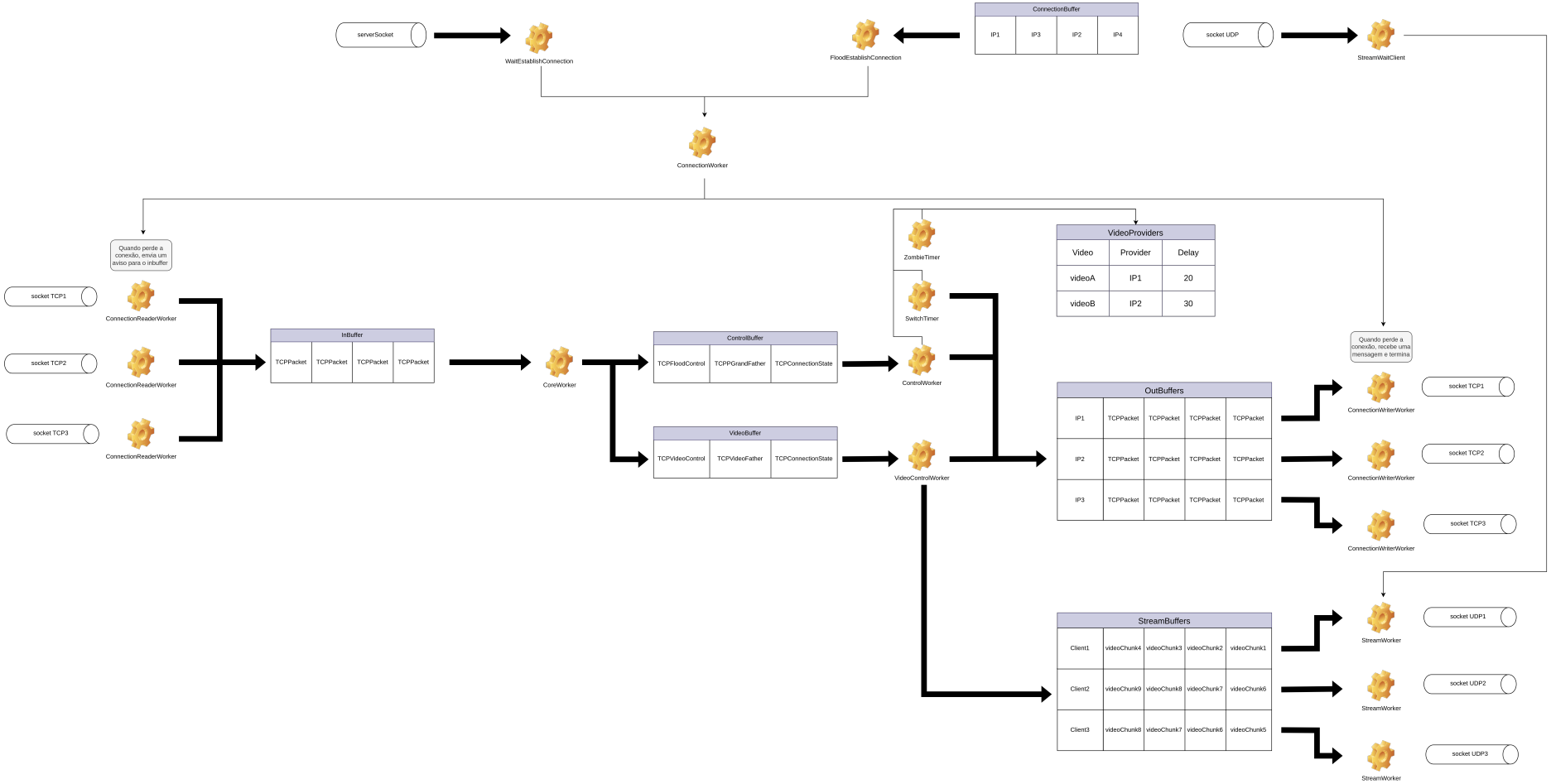


Figura 2: Diagrama de funcionamento do Edge Node

4.5.1) Cliente pede conteúdo

Quando um cliente pede conteúdo, o processo de obtenção do mesmo é igual ao de um nó. No entanto, ao receber o mesmo, o nó entrega o conteúdo ao cliente pela utilização do ffmpeg. O ffmpeg suporta a transmissão do conteúdo por UDP. O edge node, ao receber o conteúdo do servidor, lê o mesmo usando o ffmpeg e transmite para o cliente por UDP. Deste modo, não precisamos implementar nada para a transmissão do conteúdo entre edge node e client.

4.6) Cliente

4.6.1) Conexão a edge node

Quando o cliente inicia, começa a monitorizar a velocidade/qualidade de conexão aos seus edge nodes. É apresentada então uma tabela ao utilizador com as métricas de qualidade que permitem ao utilizador ver qual é o melhor nodo. O cliente pode então escolher qual é o nodo a quem se quer conectar.

4.6.2) Visualização de stream

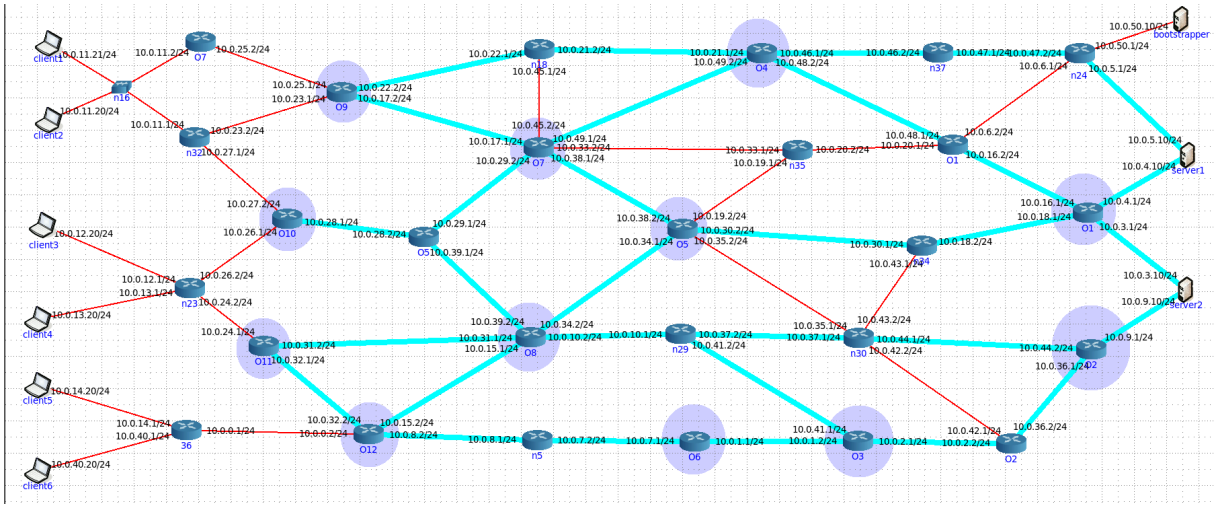
Após a conexão a um edge node, serão apresentados todos os conteúdos alcançáveis. O cliente pode então requerer o conteúdo que deseja ver. Uma instância do ffmpeg (ffplay) é aberta à escuta na porta para onde o edge node está a transmitir, reproduzindo assim diretamente o conteúdo.

4.6.3) Cancelamento de stream

A qualquer momento o cliente pode terminar a stream e escolher outra stream para visualizar.

5) Testes e resultados

Para realização de testes foi utilizada a seguinte topologia:



5.1) Features funcionais

5.1.1) Vários servidores de vídeo

A rede pode ter mais do que um servidor, com vídeos iguais ou diferentes.

5.1.2) Servidores de vídeo adicionados dinamicamente

Os servidores de vídeo podem ser adicionados enquanto rede já esta a funcionar.

5.1.3) Troca de provider durante a transmissão

Nos nodos do overlay, enquanto transmitem video, se o provider atual for lento e existirem opções mais rapidas, a conexão será automaticamente trocada para o provider mais rápido.

5.1.4) Medir qualidade dos links

Esta feature funciona tanto do cliente para o edge node (medindo perdas e latência) como entre nodos do overlay, usando apenas a latência. Neste ultimo, assumimos que perdas e congestionamento iriam acabar por se refletir em latência ao utilizar a socket TCP.

5.1.5) *Fork* da transmissão

Quando um nodo do overlay esta a transmitir video para um outro nodo, pode fazer *fork* da transmissão e passar a enviar para um ou mais outros nodos.

5.1.6) Mortes simples

Caso ocorram mortes simples, os nodos simplesmente conectam-se ao próximo melhor provider daquele video, continuando a transmissão de video.

5.1.7) Mortes complexas

Caso ocorram mortes complexas, os nodos estabelecem conexão com todos os avôs (pai do seu ultimo pai), passando a considerá-los como providers.

5.1.8) Encriptação dos pacotes TCP

O tráfego do overlay é corretamente encriptado como descrevemos anteriormente.

5.2) Features com problemas

Infelizmente não nos foi possível implementar as seguintes funcionalidades. No entanto, estas representam detalhes de implementação e poderiam ser atingidas mudando apenas o código do cliente.

5.2.1) Trocar de edge node durante a stream

Não é possível que o cliente troque de edge node enquanto a stream esta a decorrer.

5.2.2) Escolha automática de edge node

Neste momento, o cliente escolhe manualmente a qual edge node se conectar. No entanto, tem disponível métricas como latência e percentagem de perdas, que lhe permitem fazer uma decisão informada.

6) Conclusões e trabalho futuro

Durante a conceção e implementação da nossa rede de transmissão de conteúdo em tempo real com recurso a Multicast a nível aplicacional, aprendemos bastante sobre as dificuldades e benefícios de um sistema como este, tal como as suas desvantagens.

Este sistema permite que apenas um servidor (com ajuda de alguns nós intermédios) seja capaz de entregar um mesmo conteúdo a potencialmente milhares de clientes ao mesmo tempo, algo que uma única máquina seria incapaz de fazer se tivesse de enviar o conteúdo de forma individual a cada cliente.

No entanto, isto requer que todos os clientes necessitem de exatamente o mesmo conteúdo ao mesmo tempo, algo que no mundo real nem sempre se verifica. Atualmente, esta funcionalidade tem mais potencial para a visualização de eventos ao vivo (livestreaming), que é precisamente o que pretendemos emular na elaboração deste trabalho. A implementação deste sistema

necessitou de alguns cuidados, tal como a forma como recuperávamos de perdas de nodos, como descobríamos qual era o caminho ótimo para o conteúdo e até mesmo qual a melhor forma de transmitir o vídeo entre nós. Todos estes obstáculos foram oportunidades de aprendizagem que nos permitiram aprender e crescer bastante as nossas competências neste ramo.

No entanto, nada consegue ser completamente perfeito. Ainda sentimos que o nosso trabalho possui alguns pontos de melhoria. Podíamos, por exemplo, tornar a escolha de um edge node automática, fazer com que o cliente troque de edge node automaticamente caso uma alternativa melhor exista e tornar a interface do cliente numa interface gráfica. No geral, estamos bastante satisfeitos com o trabalho desenvolvido.