Trabalho Prático Nº2 – Serviço Over-the-top para entrega de multimédia

João Magalhães Jorge Rodrigues Rodrigo Gomes[1]

Universidade do Minho

Resumo O crescimento exponencial do consumo de multimédia baseado na internet tem desafiado as infraestruturas de rede tradicionais, exigindo soluções inovadoras para a entrega eficiente de conteúdos. Este artigo apresenta a conceção e implementação de um serviço de *streaming Over-the-Top* (OTT) que utiliza uma rede de sobreposição da camada aplicacional para otimizar a distribuição de conteúdos. Através da criação de uma rede de distribuição de conteúdos (CDN) dinâmica e adaptativa, a solução proposta aborda os desafios fundamentais de escalabilidade, utilização de recursos e desempenho de *streaming* em tempo real. A implementação centra-se na construção de uma topologia de sobreposição flexível com estratégias de encaminhamento inteligentes e dinâmicas, permitindo que múltiplos clientes recebam conteúdo de *streaming* através dos caminhos de rede mais eficientes. Os resultados experimentais demonstram a capacidade do sistema em minimizar o tráfego de rede e proporcionar uma experiência de *streaming* robusta em diferentes condições de rede.

1 Introdução

A paisagem da comunicação por internet sofreu uma transformação profunda ao longo dos últimos tempos, passando de um modelo de comunicação essencialmente ponto-a-ponto para um ecossistema centrado em conteúdos, caracterizado pelo consumo contínuo e em tempo real de multimédia. Esta mudança de paradigma coloca desafios significativos à infraestrutura IP subjacente, que não foi originalmente concebida para suportar a entrega massiva de conteúdos de alta qualidade. Serviços de *streaming* contemporâneos como Netflix e Hulu foram os pioneiros das técnicas de *streaming* Over-the-Top (OTT), criando redes de sobreposição na camada aplicacional que operam acima de protocolos de transporte tradicionais como TCP e UDP. Estas redes permitem uma entrega de conteúdos mais eficiente, encaminhando dinamicamente fluxos multimédia através de caminhos otimizados, contornando a congestão de rede e limitações de recursos. A investigação que se apresenta propõe uma abordagem inovadora à entrega de conteúdos OTT, focando-se nos seguintes objetivos fundamentais: conceber uma rede de sobreposição flexível e escalável na camada aplicacional, implementar mecanismos de encaminhamento inteligentes para minimizar a utilização de recursos de rede, desenvolver um serviço de *streaming* robusto capaz de se adaptar a condições de rede em mudança e criar um sistema que

possa distribuir eficientemente conteúdos para múltiplos clientes em simultâneo. A solução proposta aborda desafios críticos no *streaming* multimédia moderno, nomeadamente as limitações de escalabilidade dos modelos tradicionais de *streaming* Cliente-Servidor, a utilização ineficiente de recursos de rede, a variabilidade no desempenho de rede e a necessidade de entrega de conteúdos em tempo real com degradação mínima de qualidade. Através da construção de uma rede de sobreposição dinâmica com estratégias de encaminhamento sofisticadas, a nossa implementação demonstra um caminho potencial para serviços de *streaming* multimédia mais eficientes e adaptativos. As secções subsequentes detalharão o design arquitetural, as especificações de protocolo, as estratégias de implementação e a validação experimental do sistema de *streaming* OTT proposto.

2 Arquitetura da Solução

A arquitetura apresentada ilustra a interação entre os principais componentes do sistema: *Clientes*, *Nodes*, *POP* (*Points of Presence*), *Servidor*, e o *Bootstrapper*. Cada componente desempenha um papel específico para viabilizar o fluxo de mensagens de controlo e dados, garantindo a transmissão de vídeo de maneira eficiente e escalável.

Cliente: Os clientes são responsáveis por iniciar sessões de vídeo, representadas pelas *VideoSessions*, que gerem a reprodução do conteúdo. As sessões utilizam uma outra classe não representada, *VideoStream*, para acessar os *frames* dos vídeos a serem enviados.

Nodes e POP: Os *Nodes* atuam como intermediários na rede, encaminhando mensagens de controlo e dados entre os clientes, *POP's* e o servidor. A distinção entre *Node* e *POP* reside na configuração, que altera o comportamento na comunicação.

Todas as mensagens são controladas através do módulo *control_protocol_pb2.py*.

Servidor: O servidor controla as transmissões de vídeo. Para cada vídeo requisitado, cria um *ServerWorker* que processa as solicitações dos clientes. Caso uma sessão para um vídeo já esteja ativa, o mesmo *ServerWorker* é reutilizado.

Bootstrapper: Atua como uma entidade central para fornecer aos componentes informações sobre os seus vizinhos, essencial para o funcionamento da rede.

A arquitetura suporta escalabilidade e garante eficiência na transmissão de dados através de uma combinação de mensagens de controlo e pacotes de dados multimédia.

A solução foi projetada para ser escalável, robusta e eficiente. A arquitetura do sistema é baseada em uma comunicação cliente-servidor, podendo ter intermediários (*Nodes/Pop's*), utilizando protocolos TCP para controlo e UDP para o envio de dados de forma eficiente e controlo no caso de ser comunicação entre Cliente-Pop.

A arquitetura geral pode ser visualizada na Figura 1.

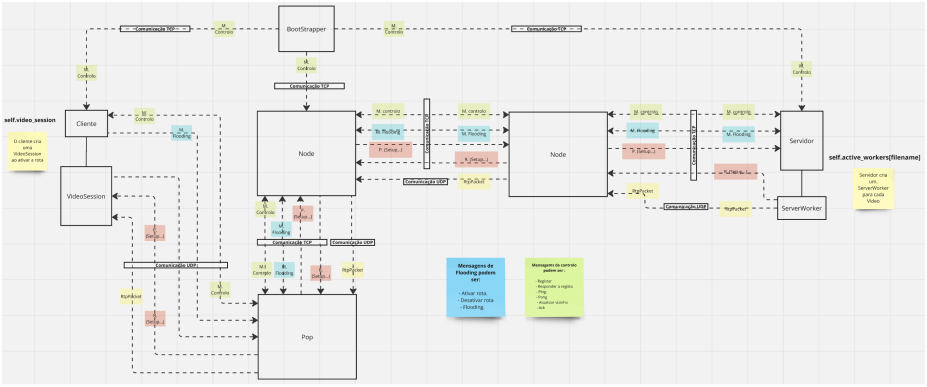

Figura 1. Arquitetura da solução proposta

3 Especificação do(s) Protocolo(s)

Nesta secção, detalhamos os formatos das mensagens protocolares utilizadas na comunicação entre os componentes do sistema. Estas mensagens são fundamentais para garantir o correto funcionamento das funcionalidades, como monitorização, manutenção da rede e transmissão de dados.

3.1 Formato das Mensagens Protocolares

1. **Mensagem de Registo e Atualização de Vizinhos**
   Após o registo de um qualquer elemento da rede, exceto o *Bootstraper*, no sistema, o *Bootstrapper* envia uma mensagem contendo a lista de vizinhos desse componente. O formato da mensagem é o seguinte:

```
{
  "response": {
    "type": "REGISTER_RESPONSE",       // Tipo de resposta, indicando o registo de vizin
    "neighbors": [                     // Lista de vizinhos ativos
      {
        "node_id": "string",           // Identificador único do nó vizinho
        "node_ip": "string",           // Endereço IP do nó vizinho
        "control_port": "integer",     // Porta utilizada para comunicações de controlo
        "data_port": "integer",        // Porta utilizada para transferência de dados
        "node_type": "string",         // Tipo de nó (e.g., "node", "pop")
        "rtsp_port": "integer"         // Porta RTSP para controlo de transmissão de víd
      }
    ]
  }
}
```

Nesta mensagem de registo vão informações do vizinho como o seu *id* e *ip* a porta de controlo onde irá receber as futuras mensagens de controlo, a porta de dados onde irá receber os pedidos do vídeo e os pacotes e o seu tipo (*node*, *server* ou *pop*)

2. **Mensagem ACK**
   Os *Nodes* e *POP's* e *Servidor* utilizam mensagens de Ping/Pong para monitorizar a conectividade com seus vizinhos. No entanto, a monitorização entre Cliente-Pop foi restringida a comunicação UDP, logo foi criada uma mensagem de *Ack* com o mesmo intuito.
   Segue se o formato da mensagem:

```
{
  "ack_message": {
    "type": "ACK",                    // Tipo de mensagem, indicando que é um ACK
    "node_ip": "string",              // Endereço IP do PoP que envia a mensagem
    "node_id": "string",              // Identificador único do PoP que envia a mensagem
    "accumulated_time": "float"       // Melhor tempo acumulado desde o servidor até o Po
  }
}
```

   Nesta mensagem vão os campos *id*, *ip* do *PoP* e o melhor tempo acumulado desde o servidor até ao próprio de maneira a que, o cliente ao receber este *ACK*, possa futuramente escolher qual é o melhor.

3. **Mensagem Flooding**
   O servidor, a cada intervalo de tempo definido, envia uma mensagem de *Flooding* para atualizar as tabelas de routing em toda a rede. Essas mensagens propagam-se de maneira controlada até os POPs fazendo com que, no fim, todos os intermediários da rede tenham nas suas tabelas as rotas necessárias para chegar até ao servidor. Segue se o formato da mensagem:

```
{
  "flooding_message": {
    "type": "FLOODING_UPDATE",        // Tipo de mensagem, indicando atualização de floo
    "source_id": "string",            // Identificador único do servidor que envia a men
    "source_ip": "string",            // Endereço IP do servidor que envia a mensagem
    "stream_ids": ["string"],         // Lista de identificadores dos vídeos disponíveis
    "route_state": "active",          // Estado atual da rota (neste caso, ativa)
    "control_port": "integer",        // Porta de controlo do servidor
    "rtsp_port": "integer"            // Porta RTSP do servidor para controlo de streami
  }
}
```

   Na mensagem de *flooding*, vai o *id*, *ip* de quem envia, neste exemplo a mensagem está a ser enviado do servidor, uma lista de filmes disponíveis na rede, o estado inicialmente ativo.

4. **Mensagem Ativação**
   Quando um cliente solicita um vídeo, é enviada uma mensagem de ativação de rota ao melhor *PoP*, previamente escolhido. Cada interveniente na rede encaminha a mensagem através da sua melhor entrada até chegar ao servidor.

```
{
  "activate_message": {
    "type": "ACTIVATE_ROUTE",          // Tipo de mensagem, indicando ativação de rota
    "stream_ids": ["string"],          // Identificador do vídeo para o qual a rota será
    "source_ip": "string",             // Endereço IP do nó que envia a mensagem
    "rtp_port": "integer",             // Porta RTP para envio dos pacotes do vídeo
    "rtsp_port": "integer"             // Porta RTSP para comunicação de controlo de stre
  }
}
```

   Nesta mensagem, segue se os campos *ip* o nome do video para qual a rota vai ser ativa e a *rtp_port* por onde serão enviados os pacotes do vídeo.

3.2 Interações

Inicialmente, como o *BootStrapper* tem acesso a todos os elementos da rede e aos que estão atualmente ativos (armazenados no dicionário *self.nodes*), ao conectar-se à rede, cada novo nó recebe a lista de vizinhos ativos para atualizar as suas informações.

À medida que os *nodes*, o servidor e os clientes vão se conectando e recebendo as informações dos seus vizinhos, o processo de monitorização é iniciado através das mensagens de *ping/pong* e dos *ACK's*, como já mencionado. Além da sua função principal de monitorização, estas mensagens também transportam o tempo acumulado desde o servidor até ao nó que as envia. Isso permite que cada *node* determine qual dos seus vizinhos oferece o caminho mais rápido para alcançar o servidor.

Após receber a informação dos vizinhos ativos, o servidor envia uma mensagem de *flooding*, que será encaminhada de forma controlada pelos *nodes* até chegar aos *Pop's*.

Quando o cliente se conecta à rede e recebe a lista dos seus vizinhos, e posteriormente conhece o tempo necessário para alcançar o servidor a partir de cada um, ele consegue determinar o melhor *Pop* para o envio de uma mensagem de ativação. Essa mensagem será reencaminhada pelas melhores rotas, de acordo com as tabelas de encaminhamento de cada *node* que a receber.

Com a rota ativa, o cliente pode enviar os pedidos relacionados ao vídeo.

4 Implementação

A implementação foi feita utilizando a linguagem Python, utilizando diversas bibliotecas e módulos para implementar as funcionalidades desejadas. Segue-se uma descrição das principais:

4.1 Bibliotecas Nativas do Python

1. **socket**: Utilizada para a criação e manipulação de conexões TCP e UDP, fundamentais para a comunicação entre os diferentes componentes da rede, como clientes, *nodes*, *Pop's* e servidor.
2. **threading**: Empregue para realizar operações concorrentes, permitindo que várias tarefas, como envio e receção de mensagens, sejam executadas simultaneamente.
3. **time**: Usada para operações relacionadas a temporização, como pausas (*sleep*) ou medição de tempo, útil no envio periódico de *pings* e mensagens de *flooding*.

4.2 Bibliotecas para Protocolo de Controlo

1. **control_protocol_pb2**: Inclui as classes geradas pelo *Protocol Buffer*, uma tecnologia utilizada para serializar as mensagens do protocolo de controle (*ControlMessage* e *FloodingMessage*). Estas mensagens definem a interação entre os componentes da rede.
2. **ControlMessage**: Define as mensagens de controlo, como *ping*, *ack* e atualizações de vizinhos.
3. **FloodingMessage**: Especifica as mensagens de *flooding*, usadas para atualização de tabelas de encaminhamento nos *nodes*, ativação e desativação de rotas.
4. **google.protobuf**: Bibliotecas relacionadas ao *Protocol Buffer*, utilizadas para processar e construir mensagens protocolares de forma eficiente.

Para além das bibliotecas anteriores foram utilizadas bibliotecas gráficas, como o *tkinter*, para criar interfaces que permitam a interação visual com o cliente, e o *Pillow* para processar *frames* de vídeo exibidos. Na manipulação de vídeos, o *OpenCV (cv2)* é empregue para leitura e processamento dos fluxos, enquanto a classe *VideoStream* gere os *frames* individuais para transmissão. Já o *RtpPacket* cuida do empacotamento dos dados de vídeo em pacotes RTP, utilizados na transmissão via UDP.

4.3 Detalhes de Implementação

Vantagens TCP :

- Confiabilidade: TCP garante entrega de pacotes, retransmitindo-os em caso de perda.

- Controle de fluxo e congestionamento: TCP ajusta dinamicamente a taxa de transmissão para evitar sobrecarregar a rede.

- Sequência: Mensagens chegam na ordem correta, o que é fundamental para aplicações que dependem de integridade de dados.

Desvantagens TCP :

- Maior latência: O controle de fluxo, *ACKs* e os retransmissões adicionam atraso.
- Overhead: O cabeçalho TCP é mais complexo, aumentando o custo de processamento e largura de banda.

Vantagens UDP :

- Baixa latência: Ideal para aplicações em tempo real, já que não há confirmações ou retransmissões.
- Menor overhead: Cabeçalho simples e menor consumo de recursos.
- Flexibilidade: Permite transmissão de dados para múltiplos recetores (multicast/broadcast).

Desvantagens UDP :

- Confiabilidade: Não garante entrega de pacotes nem a sua ordem.
- Controlo de fluxo: Risco de sobrecarga em redes congestionadas.
- Maior complexidade na aplicação: A lógica de verificação e recuperação de pacotes perdidos tem que ser implementada manualmente.

Na nossa implementação optamos por utilizar TCP para monitorização e pedidos de vídeo e assim, garantiamos que as mensagens de controle chegavam corretamente, sem perdas, e na sequência correta. Isso é crucial, pois qualquer inconsistência nas mensagens de controle poderia comprometer a integridade da rede ou a ativação de uma rota.

Relativamente à transmissão dos frames de vídeo, o UDP foi a escolha ideal devido à sua baixa latência. Como os frames são enviados constantemente, o impacto da perda de alguns pacotes é mínimo e não justifica o overhead associado à retransmissão no TCP. Essa escolha assegura que a experiência de reprodução de vídeo permaneça fluida.

A escolha para a comunicação UDP entre Cliente-Pop, foi uma restrição imposta pelos docentes.

5 Testes e Resultados

Foram realizados testes para avaliar o desempenho do sistema de comunicação, bem como a funcionalidade e eficácia dos protocolos implementados, de modo a garantir a correta implementação dos objetivos a que nos propusemos. Assim sendo, passamos a apresentar a topologia de teste, bem como alguns dos vários cenários testados.

5.1 Topologia de teste

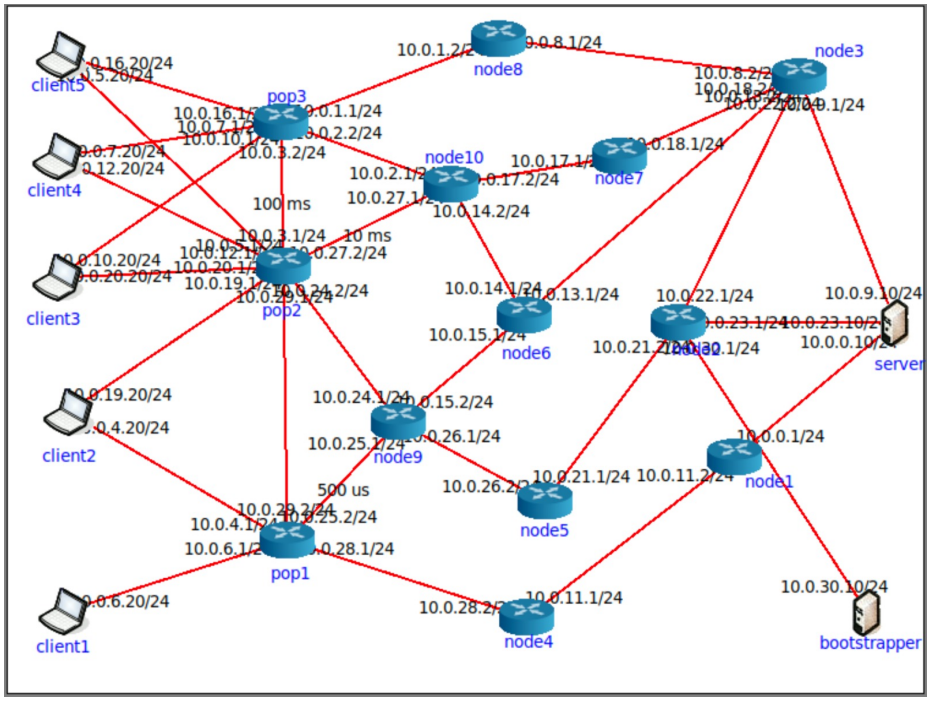

Figura 2. Topologia de rede utilizada para teste de cenários.

De modo a testar diversos cenários de transmissão, utilizamos a topologia visível na figura 2, composta por 1 *BootStrapper*, 1 Servidor, 10 nós intermédios, 3 POPs e 5 clientes.

5.2 Cenários de teste

Tendo os nós da camada de *Overlay* ligados, bem como o *Bootstrapper* e o Servidor, o processo de determinação de melhores rotas é iniciado, ficando registado nas tabelas de encaminhamento de cada *router* a melhor rota até ao servidor. A partir daqui, os pontos de presença conseguem informar clientes que se liguem a eles dos tempos desde os mesmos ao servidor. Corremos então os seguintes cenários e obtivemos os consequentes resultados:

1. Clientes 3 e 4 pedem um determinado vídeo, a título de exemplo, video1, encontrando-se ativos os nós 3, 7 e 10, e os POPs 2 e 3:

Ambos os clientes escolhem o melhor *POP* para realizarem o pedido do vídeo, com base nos tempos fornecidos pelos mesmos (neste caso, *pop2*). É ativada a

melhor rota com base nas tabelas de roteamento dos nós e o vídeo passa a fluir desde o servidor aos clientes por *node3 - node7 - node10 - pop2*. É ainda de mencionar que a pausa ou paragem de uma *stream* não influencia *streams* que estejam a ocorrer em simultâneo e a retoma ocorrerá no ponto em que se encontrar a *stream* no(s) restante(s) cliente(s).

2. É adicionado um *delay* nos *links* entre o *pop2* e os clientes a ele conectados:

Através dos nossos mecanismos de monitorização de rede, mencionados no decorrer do relatório, face a uma perda significativa de performance, ocorre a desativação do fluxo de stream pelo *pop2* e a stream passa a chegar aos clientes pelo *pop3*, o qual passa a possuir um melhor tempo.

3. O *node8* é ativado:

Passa a ser detetada uma melhor rota até ao *pop3*, com base em RTT, e a stream passa a fluir por *node3 - node8 - pop3*, sendo desativado o fluxo anterior desde o *pop3* até ao *node3*.

4. O *node8* é desativado:

Terminando o programa no *node8*, é detetada a sua inatividade pelos nós vizinhos, os quais desativam as rotas que utilizam este nó. De seguida, consultando as suas tabelas de roteamento e os tempos das possíveis rotas alternativas, é escolhida uma nova melhor rota, sendo esta ativada, culminando na passagem da *stream* por este novo caminho.

Com base em cenários de teste deste género, foi-nos permitido assegurar a correta implementação das funcionalidades do nosso projeto, como, por exemplo, monitorização de rede *Overlay*, árvores de distribuição dinâmicas, tanto na adição como remoção de nós, e monitorização ao nível do cliente (assegurados nos cenários demonstrados), bem como *stream* multi-cliente e serviço *multi-stream*.

6 Conclusões e Trabalho Futuro

Este trabalho apresentou uma solução de comunicação eficiente entre sistemas distribuídos, utilizando os protocolos RTSP e RTP. O sistema foi projetado para ser escalável e robusto, com base nas especificações dos protocolos.

Para o trabalho futuro, é sugerido explorar técnicas avançadas de correção de erros, como FEC (Forward Error Correction), para melhorar a taxa de entrega de pacotes em redes com alta latência ou perda de pacotes. Além disso, melhorias no gerenciamento de múltiplas conexões simultâneas e a implementação de um mecanismo de qualidade de serviço (QoS) podem ser exploradas para aumentar ainda mais a robustez e a performance do sistema.

Agradecimentos

Gostaríamos de agradecer à nossa instituição pelo suporte durante o desenvolvimento deste trabalho.