Trabalho Prático Nº2 - Serviço Over the Top para entrega de multimédia

Diogo Rodrigues[pg60244] and Junqing chen[pg58807]

Universidade do Minho, Braga, Portugal

Abstract. Este trabalho apresenta a conceção e implementação de um serviço Over-the-Top (OTT) para distribuição de conteúdos multimédia em tempo real, utilizando uma rede overlay aplicacional para otimizar a entrega de streams de vídeo. O sistema desenvolvido contorna as limitações de escalabilidade da arquitetura cliente-servidor tradicional através de nós intermediários que replicam e reencaminham conteúdos, formando uma topologia virtual sobre a infraestrutura IP subjacente.
A solução implementa um protocolo de controlo aplicacional que gere dinamicamente a construção de rotas baseadas em métricas de rede, especificamente latência e número de saltos. O algoritmo de seleção de rotas prioriza caminhos com menor delay, permitindo ao sistema escolher automaticamente rotas alternativas mais rápidas mesmo que estas atravessem mais nós intermediários. A validação experimental, realizada através do emulador CORE, demonstra a capacidade do sistema de detetar e evitar caminhos com latências elevadas, optando por rotas com menos delay apesar do aumento no número de saltos.
O protótipo utiliza o protocolo RTP sobre UDP para transmissão de vídeo MJPEG, permitindo streaming em tempo real com replicação eficiente através da rede overlay, reduzindo significativamente a carga no servidor de origem.

Keywords: Nó · Servidor · Cliente · OTT · Pacote · UDP · RTP · vizinho · transmissão · CORE · latência · saltos · multimédia · rotas · custo · rede · streaming · flooding · métricas.

1 Introdução

1.1 Contexto

A Internet atual caracteriza-se pelo consumo massivo de conteúdos multimédia em tempo real, um paradigma distinto da comunicação extremo-a-extremo para a qual foi originalmente concebida. Serviços Over-the-Top (OTT) como Netflix ou YouTube operam sobre a camada aplicacional para entregar streams de vídeo a milhões de utilizadores, mas enfrentam limitações críticas de escalabilidade: numa arquitetura cliente-servidor tradicional, cada cliente adicional multiplica os requisitos de largura de banda do servidor. Redes overlay aplicacionais contornam esta limitação criando uma topologia lógica de nós intermediários que replicam e reencaminham conteúdos, distribuindo a carga e otimizando rotas

com base em métricas de rede (latência, número de saltos). Esta abordagem permite reduzir significativamente o tráfego no servidor de origem e melhorar a qualidade de experiência dos utilizadores finais.

1.2 Objetivo

Este trabalho tem como objetivo desenvolver um protótipo funcional de um serviço OTT para entrega de multimédia em tempo real, utilizando uma rede overlay aplicacional. Os objetivos específicos são:

- Implementar uma arquitetura overlay dinâmica com nós intermediários capazes de reencaminhar streams entre servidores e clientes;
- Desenvolver um protocolo de controlo para construção e manutenção de rotas, incluindo mecanismos de descoberta de vizinhos e gestão de fluxos;
- Integrar métricas de rede (latência e número de saltos) na seleção de rotas, permitindo ao sistema escolher dinamicamente os caminhos mais eficientes;
- Validar experimentalmente a capacidade do sistema de escolher rotas alternativas mais rápidas e replicar conteúdos eficientemente, utilizando o emulador CORE com topologias controladas.

2 Arquitetura da Solução

A arquitetura da solução proposta baseia-se numa rede overlay aplicacional construída sobre uma infraestrutura de rede IP emulada (o underlay), utilizando o emulador CORE.

O Underlay é constituído pelos nós de rede emulados (routers, switches e hosts) e pelas ligações físicas que os unem. Estes fornecem a conectividade IP básica e o transporte de datagramas UDP.

O Overlay é uma rede lógica formada pelas aplicações Java desenvolvidas (Server, Node, Client) em execução nos hosts do underlay. Estas aplicações estabelecem ligações virtuais entre si (vizinhos lógicos) e implementam o seu próprio plano de controlo (encaminhamento, gestão de topologia) e plano de dados (distribuição de vídeo). A topologia do overlay é independente da topologia física, permitindo a criação de árvores de distribuição lógicas otimizadas.

A Figura abaixo ilustra esta separação:

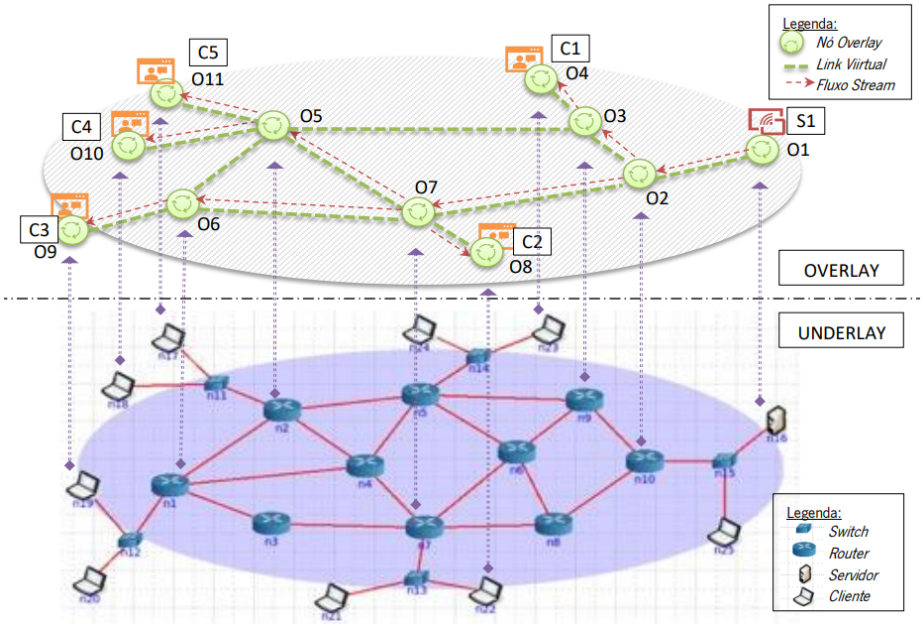

Fig. 1. Visão geral de um serviço OTT sobre uma infraestrutura IP

Na arquitetura implementada é possível destacar as principais classes:

1. Servidor e Bootstrapper (Server.java):
   Este componente desempenha um papel duplo na arquitetura:
   - Fonte de Conteúdos (Servidor): É responsável pela leitura do ficheiro de vídeo (MJPEG), fragmentação em imagens/frames e encapsulamento em pacotes RTP para envio pela rede overlay. Gere o início e fim da stream.
   - Gestor de Topologia (Bootstrapper): Atua como o ponto de contacto inicial para a construção da rede overlay. Lê um ficheiro de configuração que define a topologia lógica (quem é vizinho de quem) e, quando contactado pelos nós (Node), fornece-lhes a lista dos seus vizinhos lógicos. Também encaminha os Clientes para o ponto de acesso (nó de borda) mais apropriado.
2. Nó de Encaminhamento (Node.java):
   É o elemento fundamental da rede overlay. Executado em máquinas intermédias, as suas funções principais são:
   - Manutenção de Vizinhança: Regista-se no Bootstrapper no arranque para descobrir os seus vizinhos.
   - Plano de Controlo (Flooding e Rotas): Executa um protocolo de flooding periódico para descobrir caminhos para o servidor e calcular métricas (custo/atraso), preenchendo a sua tabela de encaminhamento (routing).

   - Gestão de Estado (Open Gates): Recebe pedidos de ativação de fluxo ("Open Gate") dos clientes (ou nós a jusante) e propaga-os em direção à origem, ativando o reenvio na sua tabela de encaminhamento apenas quando necessário.
   - Plano de Dados (Forwarding): Recebe pacotes RTP, consulta a tabela de encaminhamento e replica o pacote apenas para os vizinhos que solicitaram explicitamente o fluxo (estado "on").

3. Cliente (Client.java):
   Representa o utilizador final que consome o conteúdo.
   - Conexão: Contacta o Servidor para solicitar entrada na sessão. O servidor atribui-lhe um nó da rede overlay ao qual se deve ligar (o seu "pai" na árvore de distribuição).
   - Sinalização: Envia mensagens periódicas de keep-alive ("Open Gate") ao seu nó de acesso para manter o fluxo de dados ativo.
   - Reprodução: Recebe os pacotes RTP, extrai as frames de vídeo MJPEG e reproduz a animação numa interface gráfica (GUI).
4. Protocolos de Comunicação:
   A solução utiliza UDP como protocolo de transporte para todos os componentes, maximizando a eficiência para tráfego de tempo real.
   - Mensagens de Controlo: Pacotes (Packet.java) customizados (serializados em bytes) para gestão de topologia, flooding de rotas e sinalização de portas (gates).
   - Fluxo Multimédia: Pacotes RTPpacket (RTPpacket.java) que seguem uma estrutura simplificada do protocolo RTP (Real-time Transport Protocol), transportando o payload do vídeo, número de sequência e timestamp.

3 Especificação dos Protocolos

3.1 Topologia e Gestão do Overlay

Estratégia de Construção: A construção da rede overlay segue uma abordagem centralizada com Bootstrapper. O servidor de streaming (Server.java) atua simultaneamente como Bootstrapper e gestor da topologia.

- Inicialização: Ao iniciar, o servidor lê um ficheiro de configuração (ex: bootstrapper.txt) que define a topologia estática da rede (quem se liga a quem).
- Registo de Nodos: Quando um nó overlay (Node.java) arranca, envia uma mensagem de registo (ID=0) ao servidor. O servidor responde (ID=1) fornecendo a lista de vizinhos atribuídos a esse nó, conforme a topologia carregada.
- Registo de Clientes: O cliente (Client.java) contacta o servidor (ID=3) para solicitar entrada na rede. O servidor atribui um "Ponto de Acesso" (um nó da overlay) ao cliente, baseando-se na proximidade do terceiro octeto dos endereços IP.

Manutenção e Tolerância a Falhas: A manutenção da atividade dos vizinhos e clientes é garantida através de um mecanismo de Heartbeats:

- Os clientes enviam periodicamente (a cada 5 segundos) uma mensagem de "Open Gate / Heartbeat" (ID=4) para o seu nó de acesso.
- Os nós monitorizam estas mensagens através de um temporizador (clientActive no código). Se um nó não receber confirmação de atividade de um cliente durante 3 ciclos de verificação consecutivos, assume-se que o cliente falhou ou desconectou-se.
- Consequentemente, o nó fecha o fluxo de dados para esse cliente (status="off") e, se não houver outros clientes ativos a jusante, propaga o fecho da "comporta" para o nó ascendente (ID=5), libertando recursos da rede.

3.2 Gestão de Rotas e Fluxos

Tabela de Rotas: Cada nó e o servidor mantêm uma tabela de encaminhamento dinâmica, estruturada da seguinte forma:

- Chave Primária: IP da Origem do Fluxo (Servidor).
- Dados de Rota:
  - Previous: Nó anterior (pai) no caminho ótimizado até ao servidor.
  - Cost: Custo acumulado (número de saltos/hops) até ao servidor.
  - Routing Map: Mapeamento dos vizinhos descendentes e o seu estado (IP Vizinho -> Status ["on" | "off"]). O estado "on" indica que o fluxo deve ser replicado para aquele vizinho.

Algoritmo de Roteamento: As rotas são criadas por iniciativa do Servidor através de um mecanismo de Inundação Controlada (Flooding):

1. Anúncio: O servidor envia periodicamente (a cada 30s) uma mensagem de Flood (ID=2) contendo um UniqueID (UUID) e um TTL (Time-To-Live).
2. Prevenção de Ciclos: Ao receber uma mensagem de Flood, cada nó verifica se já processou aquele UniqueID. Se sim, descarta a mensagem para evitar ciclos infinitos na rede overlay.
3. Propagação: Se a mensagem for nova e o TTL válido, o nó atualiza a sua métrica e reenvia a mensagem a todos os seus vizinhos (exceto àquele de quem recebeu).

Métrica de Seleção: A melhor rota é escolhida com base numa métrica híbrida que privilegia a latência, usando o número de saltos como critério de desempate:

- O pacote transporta um Timestamp de origem. Ao chegar, calcula-se o atraso (delay).
- O nó compara o novo caminho com o existente: Se (novo_delay $<$ delay_atual) OU (novo_delay $==$ delay_atual E novo_custo $<$ custo_atual), a rota é atualizada e o nó remetente passa a ser o novo "pai" (Previous) na árvore de distribuição.

3.3 Protocolo de Streaming

Transporte: Foi escolhido o protocolo UDP (User Datagram Protocol) para o transporte dos dados multimédia (RTP).

- Justificação: O streaming em tempo real tolera a perda pontual de pacotes, mas é sensível ao atraso (latência) e jitter. O TCP, com os seus mecanismos de retransmissão e controlo de fluxo (Handshake, ACKs), introduziria atrasos inaceitáveis e paragens na reprodução ("buffering") caso houvesse perda de pacotes, o que não é desejável para Live Streaming.

Mecanismo e RTP: Implementou-se uma versão simplificada do RTP (Real-time Transport Protocol), encapsulado sobre UDP:

- Segmentação: O vídeo (MJPEG) é lido frame a frame. Cada frame é encapsulada num pacote RTPPacket.
- Cabeçalho RTP: Inclui SequenceNumber (para deteção de perdas/ordenação), TimeStamp, e PayloadType.
- Extensão: Adicionou-se um campo extra ao cabeçalho RTP contendo o Source IP (4 bytes) para permitir que os nós intermédios identifiquem a que fluxo (árvore) pertence o pacote sem necessidade de inspeção profunda complexa.
- Reconstrução: O cliente extrai o payload (imagem JPEG) e atualiza a interface gráfica imediatamente.

3.4 Formato das Mensagens

Cada pacote tem um identificador, o custo, um LocalTime que é a hora a que este sai da sua origem e um campo vizinhos de tamanho indefinido onde são enviados InetAddress relevantes ao tipo de pacote. Assim os nossos pacotes teriam um tamanho dinâmico que dependeria do tamanho do campo vizinhos.

O campo custo é enviado a zero na maior parte dos pacotes excepto no pacote do tipo 2 em que funciona como um acumulador que permite saber o número total de saltos até chegar a um dado nó. Para todos os pacotes,exceto os do tipo 1, o campo vizinho será enviado a null de modo a poupar o tamanho do pacote enviado e agilizar o envio de informações de monitorização.

As mensagens de controlo trocadas na overlay são objetos serializados da classe Packet. Abaixo descrevem-se os tipos e fluxos principais. Tipos de Mensagens (PDUs de Controlo):

Table 1. Tabela das mensagens

Id	Origem → Destino	Descrição
0	Node → Server	O nó envia este ao servidor quando se conecta à rede para perguntar os seus vizinhos.
1	Server → Node	Ao receber o pedido pelos vizinhos do novo nó que se junta a rede, o servidor responde com a lista dos seus vizinhos e o nó de overlay que se deve conectar.
2	Server → Nodes	É enviado do servidor por todos os nós para fazer uma inundação controlada de forma a criar as tabelas de rotas e o cálculo do custo e do delay totais de envio de um pacote desde a origem até ao seu destino.
3	Client → Server	Pedido de conexão do cliente para o servidor para saber qual o nó overlay mais próximo para se conectar a ele.
4	Client → Node → Server	O cliente envia pedidos ao nó para abrir porta/rota de stream que são enviados upstream até ao servidor de modo que todas portas do percurso fiquem abertas. É enviado de forma contínua o que informa o nó que este cliente ainda se encontra ativo.
5	Node → Server	Quando um nó deteta que um cliente deixou de estar ativo envia upstream até ao servidor para fechar portas/rota (parar stream) até aquele.
6	Node → Client	É enviado periodicamente (cada 2s) para o cliente do nó (que se tiver ativo envia constantemente pacotes com id=4) e se após 3 destes consecutivos sem resposta do cliente, este nó envia o pacote id=5 upstream.

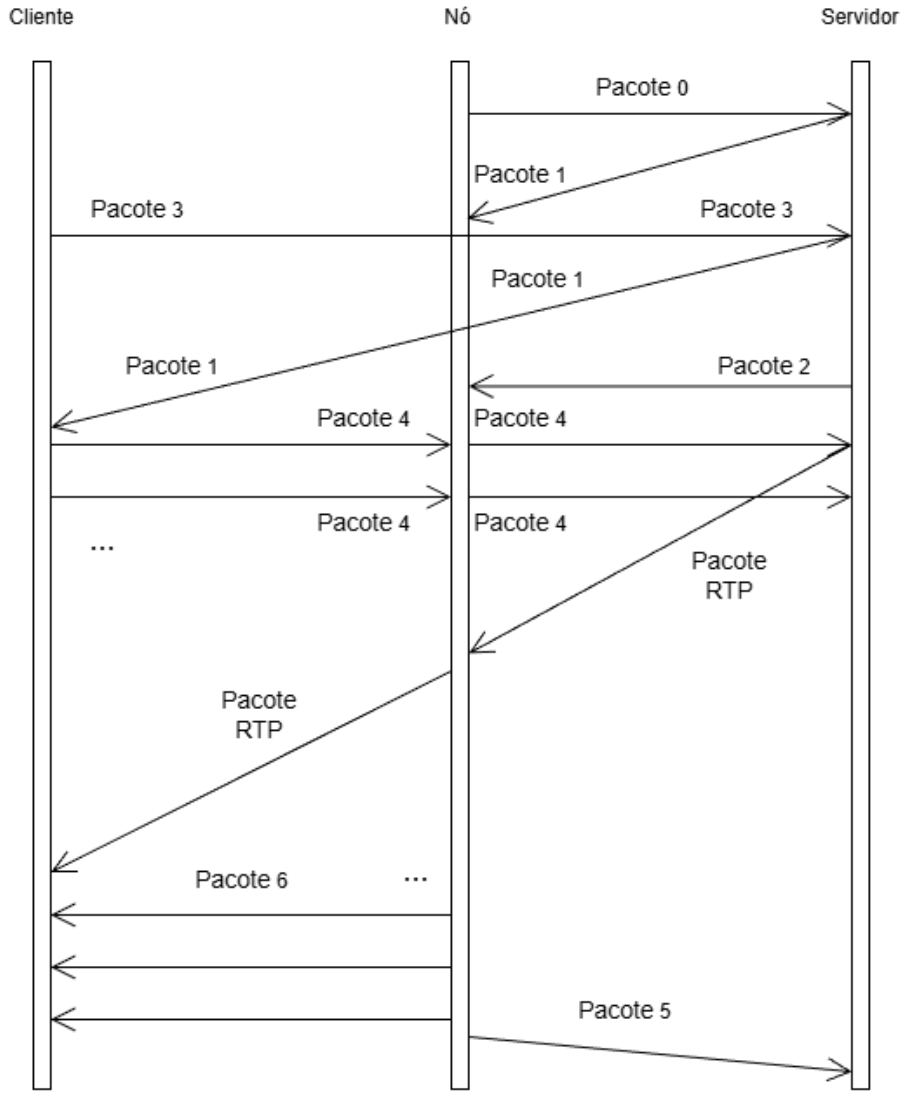

Fig. 2. Diagrama de Sequência das comunicações

4 Implementação

4.1 Tecnologias e Ferramentas

O protótipo foi desenvolvido inteiramente em Java, tirando partido da sua portabilidade e das robustas bibliotecas de rede e concorrência incluídas na JDK.

- Comunicação: Utilizou-se a biblioteca java.net (DatagramSocket, DatagramPacket) para toda a comunicação UDP, permitindo controlo total sobre a construção dos pacotes e minimizando o overhead.
- Concorrência (Multithreading): A aplicação é fortemente multithreaded. Usou-se java.lang.Thread para separar as diferentes responsabilidades de cada nó (escuta de flooding, gestão da overlay, heartbeats de clientes), garantindo que o processamento de vídeo ou bloqueios de I/O não afetam o encaminhamento de mensagens de controlo.
- Sincronização: Para garantir a consistência dos dados partilhados entre threads, utilizou-se java.util.concurrent.locks.ReentrantLock, que oferece maior flexibilidade que os blocos synchronized tradicionais.
- Ambiente de Teste: A validação foi efetuada no emulador CORE (Common Open Research Emulator), onde cada nó é isolado num contentor Linux leve, permitindo simular topologias de rede complexas numa única máquina física.

4.2 Desafios e Estruturas de Dados Críticas

Um dos maiores desafios na implementação de uma rede overlay concorrente é garantir que as tabelas de encaminhamento e estado dos vizinhos são acedidas de forma segura por múltiplas threads simultâneas (ex: a thread de Flood atualiza rotas enquanto a thread de ActiveClients verifica timeouts).

Tabela de Rotas Hierárquica: Para suportar múltiplos servidores (múltiplas árvores de streaming) e multicast dinâmico, a tabela de rotas foi implementada como um mapa aninhado (Nested Map). Isto permite que cada nó saiba, para cada fluxo (Origem), quais os vizinhos que devem receber dados.

```
1 // Estrutura: IP Origem (Stream) -> IP Vizinho -> Estado da
      Porta ("on"/"off")
2 private Map<InetAddress, Map<InetAddress, String>> routing =
      new HashMap<>();
3
4 // Estrutura: IP Origem (Stream) -> IP do "pai" na arvore (de
       onde vem o fluxo)
5 private Map<InetAddress, InetAddress> previous = new HashMap
      <>();
```

Controlo de Concorrência (Thread Safety): A modificação destas estruturas é crítica. Por exemplo, quando um cliente faz timeout, a thread de monitorização deve fechar a porta. Simultaneamente, uma mensagem OPEN_GATE pode estar a chegar. O uso de ReentrantLock evita condições de corrida (race conditions):

```
// Exemplo de logica de prote ao de estado critico
try {
    lock.lock(); // Inicio da sec   o critica

    // Atualiza ao segura do estado do vizinho
    if (routing.get(origin).containsKey(neighbor)) {
        routing.get(origin).put(neighbor, "off"); // Fecha
    porta
    }

    // Verifica ao se e necessario propagar o fecho a
    montante
    boolean hasactiveClients = checkClients(origin);
    if (!hasactiveClients) {
        sendCloseGateUpstream(origin);
    }
} finally {
    lock.unlock(); // Garantia de liberta ao do lock
}
```

Prevenção de Ciclos e Duplicação: Numa rede em malha, mensagens de Flood podem circular indefinidamente. Para resolver isto sem criar grande overhead, utilizou-se um HashSet de UUIDs para registar mensagens já observadas.

```
private Set<UUID> seenMessages = new HashSet<>();

// No processamento de Flood:
if (seenMessages.contains(packet.getUniqueId())) {
    return; // Descarta mensagem duplicada
}
seenMessages.add(packet.getUniqueId());
// ... processa e reencaminha ...
```

Gestão de Timeouts de Clientes: Para evitar iterar constantemente sobre listas complexas, utilizou-se um mapa auxiliar para contagem de "Heartbeats". Uma SimpleEntry armazena contadores que são incrementados pela receção de mensagens e comparados pela thread de monitorização.

```
// IP Cliente -> Par <Contagem Anterior, Contagem Atual>
private Map<InetAddress, SimpleEntry<Integer,Integer>>
    clientsTimer = new HashMap<>();
```

5 Testes e Resultados

Esta secção valida a qualidade da implementação e demonstra o funcionamento do protótipo em diferentes cenários, focando-se na prova de conceito funcional, eficiência de encaminhamento e escalabilidade (multicast aplicacional).

5.1 Cenário 1: Prova de Conceito Funcional (Logs de Conexão)

Para a validação funcional básica, utilizou-se a topologia definida em bootstrapper1.txt, constituída por um servidor, três nó intermédios e dois nós de borda, simulando uma pequena rede.

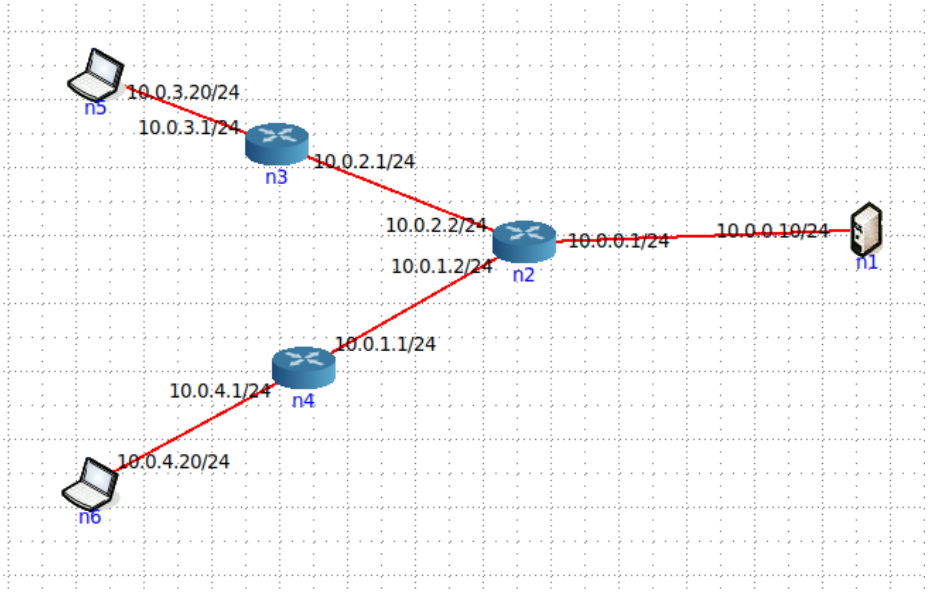

Fig. 3. Topologia simples

Fluxo de Execução:

1. Inicialização: O Servidor inicia e estabelece a rede overlay.
2. Flooding: O Servidor inunda a rede com pacotes de controlo (ID=2) para anunciar a stream e permitir o cálculo de rotas e métricas.
3. Conexão do Cliente: O Cliente liga-se ao bootstrapper (S1), que o redireciona para o nó mais próximo.
4. Ativação de Fluxo: O Cliente envia um pacote de ativação (Heartbeat ID=4), que se propaga "upstream" até ao servidor, abrindo as "comportas" (gates) apenas no caminho necessário.
5. Streaming: O vídeo é transmitido via RTP/UDP.

Abaixo apresenta-se um excerto dos logs recolhidos nos nós durante o teste.

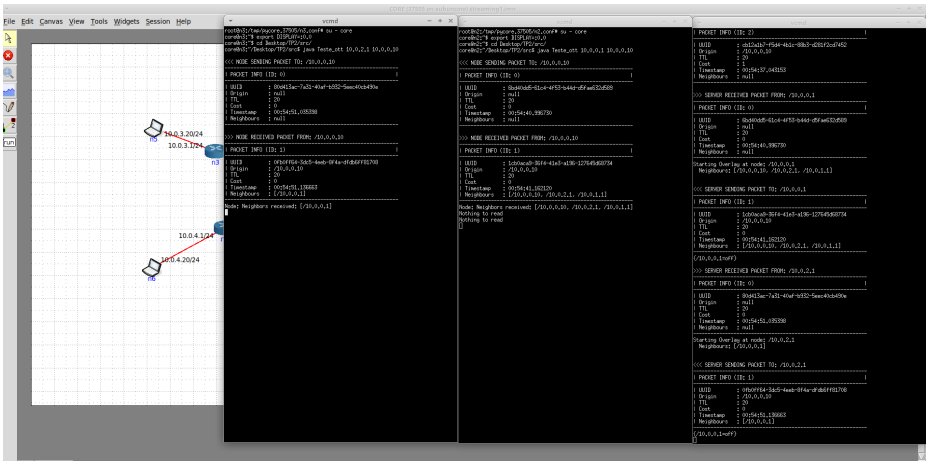

Fig. 4. Inicialização dos nós ao entrar na rede overlay

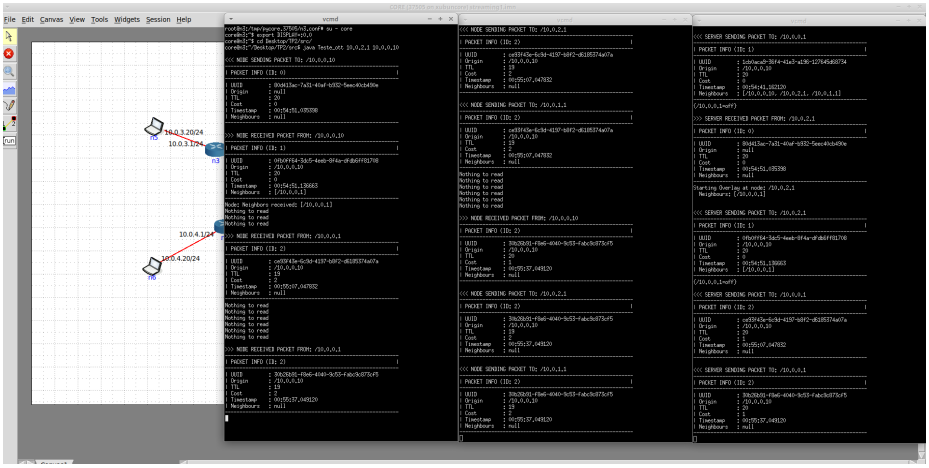

Fig. 5. Propagação do flooding

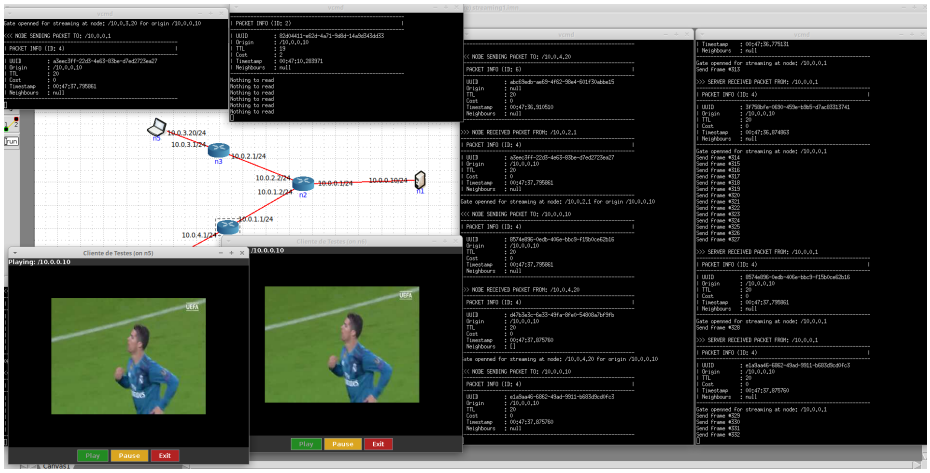

Fig. 6. Transmissão da stream nos 2 clientes

5.2 Cenário 2: Teste de Eficiência de Rotas

Neste teste, avaliou-se a capacidade do protocolo de escolher a melhor rota com base na métrica de atraso acumulado (latência), em detrimento do simples número de saltos, quando aplicável. Para isso usou-se a topologia definida pelo bootstrapper2.txt, na qual foi introduzido um delay artificial entre o n7 e n6 de 5000ms.

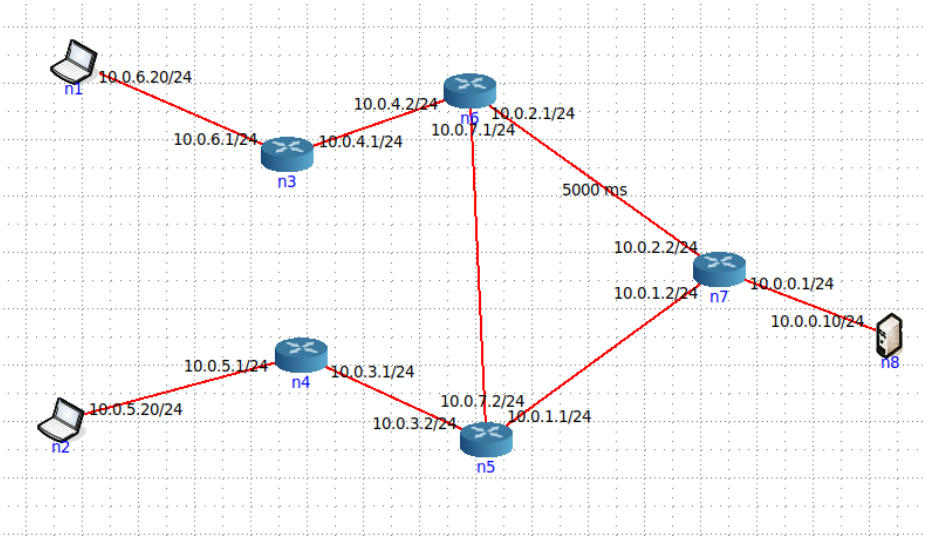

Fig. 7. Topologia de teste de eficiência de rotas

A tabela abaixo compara a rota que seria escolhida por um protocolo puramente baseado em saltos (Hop Count) vs. a rota escolhida pelo nosso protocolo Overlay baseado em latência.

Origem	Destino	Protocolo	Rota escolhida	Custo (Saltos)	Delay (ms)
n8	n1	Underlay/RIP (Saltos)	n8 → n7 → n6 → n1	3	5120
n8	n1	Overlay OTT (Delay)	n8 → n7 → n5 → n6 → n1	4	215

Table 2. Comparação de Rotas

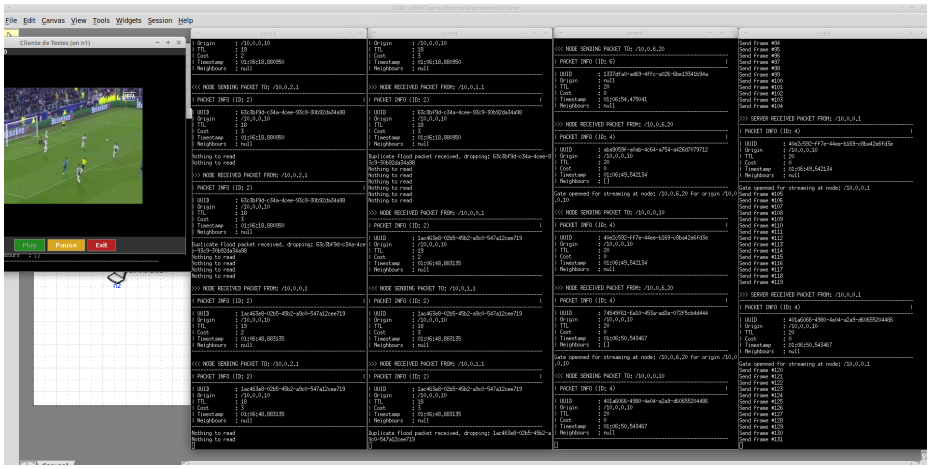

Fig. 8. Teste de eficiência de rotas

O protocolo Overlay calculou corretamente que, apesar de possuir mais um salto intermédio, o caminho oferecia uma latência significativamente menor. O sistema de flooding atualiza a tabela de encaminhamento (previous map) sempre que um pacote com menor atraso chega, garantindo a adaptação dinâmica da topologia lógica.

5.3 Cenário 3: Escalabilidade e Multicast Aplicacional

Neste cenário complexo (baseado na bootstrapper3.txt), validou-se a eficiência de largura de banda quando múltiplos clientes solicitam o mesmo conteúdo.

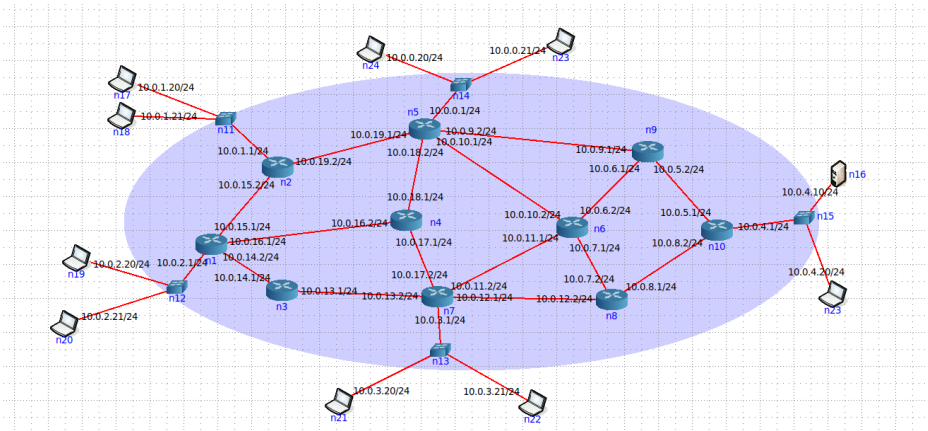

Fig. 9. Topologia complexa

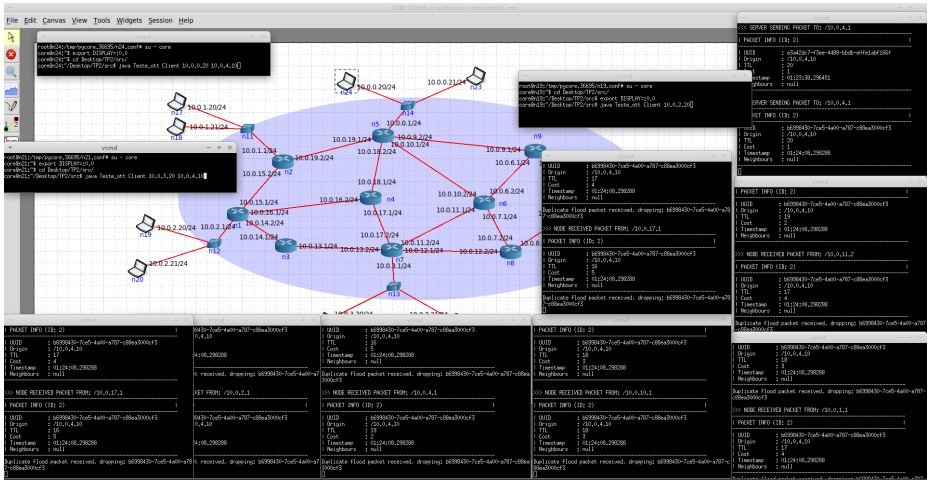

Fig. 10. Flooding na topologia complexa

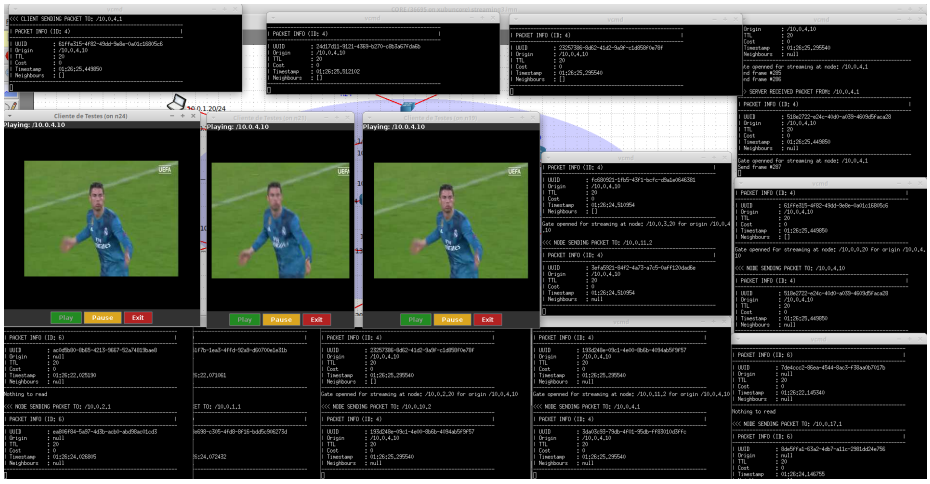

Fig. 11. 3 clientes a receber a stream em simultâneo na topologia complexa

Sem Overlay (Unicast), o servidor teria de enviar 3 cópias do fluxo. Com Overlay (Multicast Aplicacional), o servidor envia apenas 1 cópia para, que é replicada localmente.

Nº de Clientes	Bandwidth (Unicast)	Bandwidth (Multicast)	Ganho Eficiência
1	1.5 Mbps	1.5 Mbps	0%
2	3.0 Mbps	1.5 Mbps	50%
5	7.5 Mbps	1.5 Mbps	80%
10	15.0 Mbps	1.5 Mbps	90%

Table 3. Estimativa de Bandwidth (Vídeo 3MB, ≈ 1.5 Mbps)

Observação: O mecanismo implementado no método Node.nodeTimerListener verifica a tabela de encaminhamento e replica o pacote RTP recebido para todos os vizinhos com estado "on".

```
// Codigo de replicacao eficiente
for (InetAddress ip : routeTable.keySet()) {
    if (routeTable.get(ip).equals("on")) {
        RTPsocket.send(dp_replicado);
    }
}
```

Isto demonstra que a solução escala O(1) no link de origem, independentemente do número de clientes a jusante, desde que partilhem a mesma árvore de distribuição.

5.4 Cenário 4: Multi-Server e Multi-Stream

Para validar a flexibilidade da arquitetura, configurou-se um cenário com dois servidores ativos simultaneamente, oferecendo conteúdos distintos na mesma rede overlay.

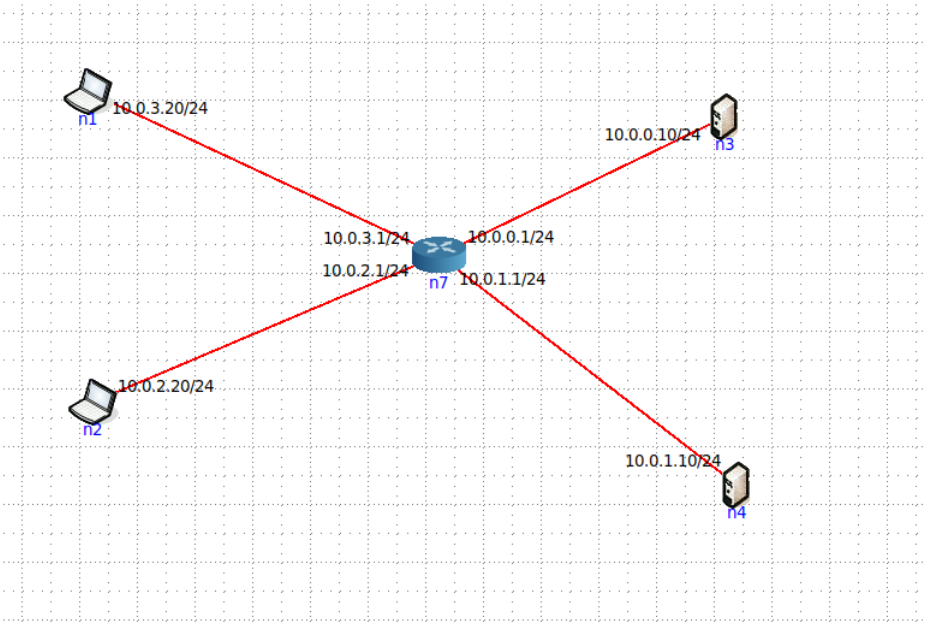

Fig. 12. Topologia Multi-Server e Multi-Stream

Configuração:

- Servidor 1 (10.0.0.10): Transmite movie.Mjpeg.
- Servidor 2 (10.0.1.10): Transmite movie2.Mjpeg.
- Nó Overlay: Nó comum que pode encaminhar tráfego de ambos os servidores.
- Cliente 1 (C1): Liga-se a S1 para ver o Filme 1.
- Cliente 2 (C2): Liga-se a S2 para ver o Filme 2.

Funcionamento: O protocolo de flooding cria árvores de distribuição independentes para cada origem (Source-Based Trees).

1. S1 anuncia-se: Os nós criam entradas na tabela de rotas para a origem S1.
2. S2 anuncia-se: Os nós criam entradas adicionais para a origem S2.
3. Encaminhamento: O buffer de tabelas de encaminhamento no nó (routing map) é indexado por Origin IP.

Resultado: Os dois fluxos coexistem na mesma infraestrutura sem interferência lógica. O nó intermédio recebe pacotes de S1 e encaminha-os apenas para os ramos da árvore de S1 (C1), e recebe pacotes de S2 encaminhando-os para a árvore de S2 (C2).

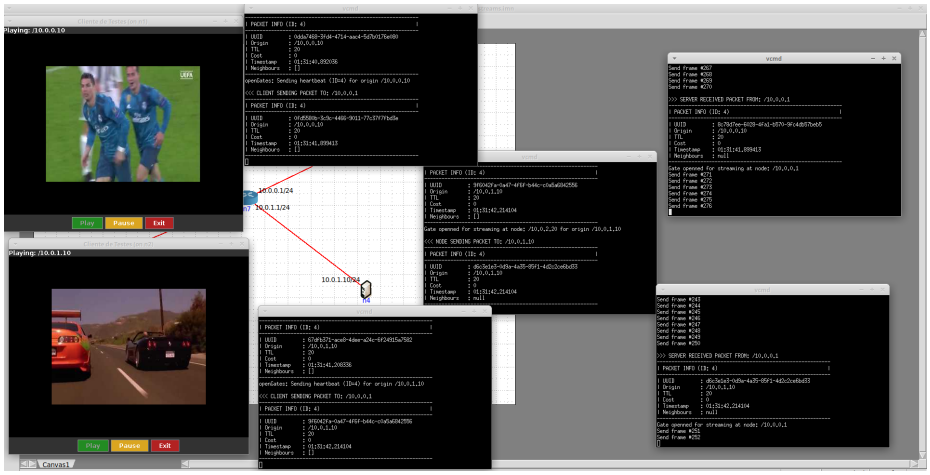

Fig. 13. 2 streams em simultâneo

Isto confirma que o protocolo suporta N streams concorrentes, limitados apenas pela largura de banda disponível nos links partilhados.

5.5 Métricas de Desempenho

O tempo médio para a rede overlay estabilizar as tabelas de rotas após o arranque do servidor.

- Tempo medido: < 200ms (para topologia de 4 nós).
- O flood periódico (30s) garante a eventual consistência em caso de alterações topológicas, mas a propagação inicial é imediata.

Já num teste de transmissão de vídeo (movie.Mjpeg) de 60 segundos com 2 clientes concorrentes:

- Perda de Pacotes (Packet Loss): < 0.5
- Jitter: Estável, sem pausas percetíveis no vídeo (buffer do cliente compensa variações menores).

6 Conclusões e Trabalho Futuro

O objetivo principal deste projeto foi a conceção e prototipagem de um serviço de streaming "Over-The-Top" (OTT) sobre uma rede IP, capaz de entregar conteúdos multimédia de forma eficiente a múltiplos clientes. Considera-se que os objetivos foram largamente atingidos. Desenvolveu-se com sucesso uma rede overlay descentralizada em Java, onde os nós intermediários formam uma malha de distribuição dinâmica.

Apesar do sucesso funcional do protótipo, identificaram-se limitações que abrem espaço para melhorias futuras, essenciais para aproximar a solução de um ambiente de produção real:

- Resiliência e Deteção de Falha de Nós Vizinhos: Atualmente, a rede não reage automaticamente à falha súbita de um nó intermediário (ex: crash da aplicação). Se um nó "morre", o fluxo para os seus descendentes é interrompido e a rota não é recalculada de imediato.
  - Melhoria Proposta: Implementar um mecanismo de heartbeat contínuo entre vizinhos diretos da overlay (e não apenas entre cliente e nó de acesso). Se um vizinho deixar de responder, o nó detetaria a falha, eliminá-lo-ia da sua tabela de vizinhos e dispararia um pedido de recálculo de rotas ou ativaria uma rota de backup alternativa.
- Saída Dinâmica de Nós: Atualmente, a saída de um nó intermediário quebra a árvore de distribuição. Embora o sistema permita que um nó (ou cliente) reentre na rede e retome o serviço (bastando reiniciar o play), a transição não é transparente.
  - Melhoria Proposta: Implementar um protocolo de "Graceful Leave", onde um nó avisa os seus vizinhos antes de sair, permitindo que estes procurem rotas alternativas proativamente antes do corte do serviço.

– Métricas de Rede: A métrica de encaminhamento atual baseia-se numa heurística simples de latência e número de saltos. Em cenários reais, seria importante considerar a largura de banda disponível e a taxa de perda de pacotes para escolher caminhos mais estáveis para vídeo de alta definição.

Este projeto permitiu consolidar conhecimentos fundamentais sobre sistemas distribuídos e redes de computadores. Do ponto de vista técnico, a principal lição aprendida prende-se com a complexidade da concorrência e estado partilhado. Gerir tabelas de encaminhamento globais enquanto dezenas de threads processam pacotes de controlo e vídeo simultaneamente exigiu um desenho rigoroso da sincronização (utilização de Locks) para evitar condições de corrida e inconsistências. Compreendeu-se também na prática a vantagem do modelo Overlay: a capacidade de implementar lógicas de encaminhamento complexas (como multicast) e serviços de valor acrescentado sem necessidade de alterar a infraestrutura física (routers/switches) da "Underlay".

References

1. Enunciado do trabalho
2. Kurose, J. (2025). Computer Networking: A Top-Down Approach (9th edition.)
3. https://www.di.uminho.pt/ flavio/ESR/ProgEx.zip