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

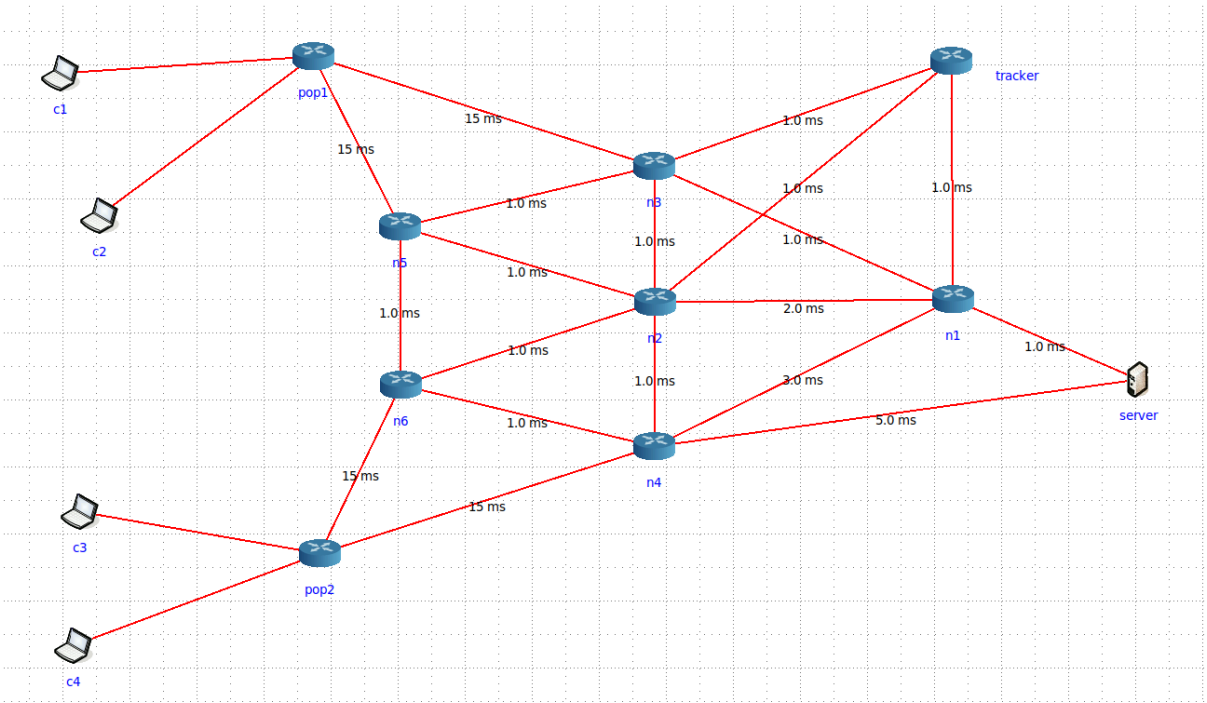
PL35

pg57879 João Rodrigues
pg57890 Mateus Lemos

Introdução

Neste projeto, foi desenvolvido um protótipo de serviço Over-the-top (OTT) para entrega de vídeo em tempo real, com base numa arquitetura simples, mas funcional, capaz de atender aos requisitos estabelecidos. Usando o emulador CORE, foi projetada uma rede overlay aplicacional que suporta a distribuição eficiente de conteúdos por meio de árvores simuladas de multicast, implementadas com unicasts específicos, e pontos de presença (PoPs). A solução priorizou a simplicidade na implementação, mantendo a capacidade de servir múltiplos vídeos a múltiplos clientes, minimizando atrasos e garantindo uma experiência de qualidade. Este relatório descreve a abordagem adotada, os principais desafios e os resultados alcançados.

Arquitetura da Solução



Para a realização do nosso projeto, optámos por uma topologia simples, mas que cumprisse todos os requisitos descritos no enunciado. A arquitetura escolhida inclui quatro **Clients**, dois **Points of Presence**, cinco **Tree Nodes** e um **ContentServer**.

Nesta arquitetura, todos os nós possuem um papel atribuído, não existindo routers sem um papel definido. Contudo, a aplicação funcionaria da mesma forma, mesmo que houvesse routers sem papel atribuído.

Para que o **Client** consiga receber o vídeo, não é necessário que todos os nós estejam em execução; basta que exista uma rota que ligue o **Client** ao **ContentServer**.

Especificação do(s) Protocolo(s) e Implementação

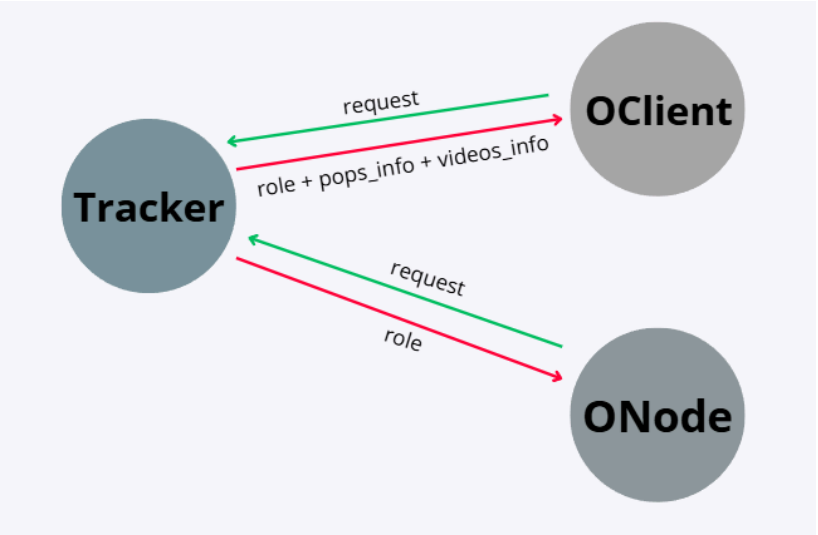
Inicialização e Funcionamento do Tracker:

O Tracker aguarda pedidos das máquinas que executam o OClient ou o ONode e fornece as informações necessárias para o correto funcionamento do sistema.

- Para os ONodes, o Tracker informa apenas o respetivo papel (ContentServer, TreeNode ou PoP).
- Para os OClients, o Tracker atribui o papel de Unknown, fornece a lista de PoPs (os seus endereços IP) e a lista dos vídeos disponíveis para visualização.

Sendo o Tracker o ponto de entrada no sistema para todos os componentes, é imprescindível que seja o primeiro elemento a ser inicializado.

A comunicação entre o Tracker e os restantes componentes do sistema realiza-se através de sockets UDP.



Inicialização e Pedido de Vídeo do OClient:

Como descrito anteriormente, o **OClient** conecta-se ao **Tracker**, que lhe fornece informações essenciais, como os endereços IP dos Points of Presence (PoPs) e a lista de vídeos disponíveis.

Com base nesta informação, o sistema aguarda que o utilizador selecione, no terminal da máquina, o vídeo que deseja assistir. Após a seleção, o **OClient** realiza um ping a cada um dos PoPs para determinar qual deles pode transmitir o vídeo com maior rapidez. Identificado o PoP mais adequado, o **OClient** envia-lhe uma mensagem de pedido para o vídeo selecionado e aguarda a receção do mesmo através de um socket UDP.

Inicialização e Funcionamento do Point of Presence (PoP):

Após a sua inicialização e a receção do respetivo papel atribuído pelo **Tracker**, o PoP entra em modo de espera por mensagens no seu socket UDP. Inicialmente, receberá uma mensagem do **OClient** solicitando o streaming de um vídeo. Ao receber esta mensagem, o PoP encaminha-a para todos os seus vizinhos na rede overlay.

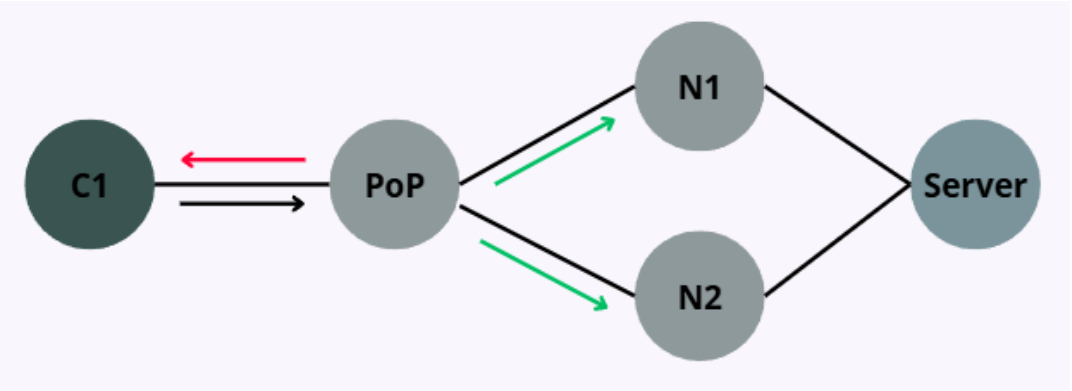
Esta mensagem é essencial para calcular a melhor rota para entregar o conteúdo ao **OClient**.

A estrutura da mensagem segue o formato:
`send video {video_name} to {pop_ip} via {route}`.

Inicialização e Funcionamento do Tree Nodes:

De forma semelhante aos PoPs, os **TreeNodes** permanecem à espera de mensagens no seu socket UDP. Quando recebem uma mensagem, propagam-na para todas as suas conexões na rede overlay, acrescentando o seu IP ao campo **route**.

Devem apenas assegurar que não redirecionam mensagens que já contenham o seu IP no campo **route**, de modo a evitar a criação de ciclos infinitos.



A figura anterior ilustra a propagação da mensagem que permite calcular a melhor rota para a transmissão do vídeo. O diagrama demonstra a chegada da mensagem ao **PoP**, enviada pelo **Client**, e a subsequente propagação do **PoP** para todas as suas conexões na rede overlay. Entre estas conexões inclui-se o próprio **Client** que enviou a mensagem, sendo esta descartada pelo **Client**, uma vez que o seu IP já consta na rota atual.

No final, o **Server** deverá receber duas rotas:

- N1 -> PoP -> C1
- N2 -> PoP -> C1

O **Server** selecionará para envio a rota correspondente à mensagem que chegar primeiro, dado que esta representa o percurso mais rápido (no sentido inverso).

Funcionamento do Content Server:

Após receber o seu papel definido pelo **Tracker**, o **ContentServer** aguarda mensagens contendo as rotas para o envio de vídeos. Com estas mensagens, o cálculo da melhor rota para servir um vídeo a um cliente torna-se um processo relativamente simples. O **ContentServer** regista a rota associada à primeira mensagem recebida para o par (cliente, vídeo), armazenando-a num dicionário no formato:

`(client_ip, video_name) -> route`.

Desta forma, o servidor mantém as melhores rotas para entregar os vídeos aos respetivos clientes.

Após determinar a rota, o **ContentServer** envia uma mensagem para cada IP da rota com a informação do próximo salto (**next hop**), garantindo que os nós sabem para onde encaminhar os pacotes de vídeo à medida que os recebem.

Para simular o multicast do vídeo, o **ContentServer** analisa as rotas existentes e apenas atualiza as tabelas de encaminhamento dos nós que ainda não estejam a transmitir para o

destino correto. Este mecanismo permite criar bifurcações em fluxos já existentes, evitando redundâncias.

Depois de atualizar as tabelas de encaminhamento dos nós envolvidos na propagação do vídeo, o **ContentServer** envia, caso necessário, os pacotes de vídeo para o primeiro nó da rota. Se já estiver a transmitir o vídeo para esse nó, não realiza um novo envio. Para esta verificação, utiliza um conjunto (**set**) que regista as transmissões ativas, associando cada vídeo ao IP de destino. Este mecanismo reforça a emulação do multicast, evitando duplicações desnecessárias.

```python
def send_video(client_ip, client_socket, video_path, client_port):
    cap = cv2.VideoCapture(video_path)
    if not cap.isOpened():
        print("Error: Unable to open video.")
        return

    print(f"Sending video to {client_ip}...")

    chunk_size = 1024
    try:
        while True:
            ret, frame = cap.read()
            if not ret:
                cap.set(cv2.CAP_PROP_POS_FRAMES, 0)
                continue

            _, encoded_frame = cv2.imencode('.jpg', frame)
            encoded_frame_bytes = encoded_frame.tobytes()
            frame_size = len(encoded_frame_bytes)

            # Dividir e enviar o frame em chunks
            for i in range(0, frame_size, chunk_size):
                packet = encoded_frame_bytes[i:i + chunk_size]
                client_socket.sendto(packet, (client_ip, client_port))

            time.sleep(0.03)
    except Exception as e:
        print(f"Error during video transmission: {e}")
    finally:
        cap.release()
```

Funcionamento dos PoPs e dos Tree Nodes no Encaminhamento dos pacotes:

Após receberem as tabelas de encaminhamento fornecidas pelo **ContentServer**, os **PoPs** e os **TreeNodes** ficam responsáveis por garantir a entrega correta dos pacotes de vídeo. Quando um pacote chega a um destes nós, eles consultam a sua tabela de encaminhamento e reencaminham o pacote para os endereços indicados como próximos saltos na rota.

Este processo garante que os pacotes de vídeo seguem o percurso previamente definido, otimizando a entrega ao cliente final. Além disso, ao trabalhar com tabelas de encaminhamento bem estruturadas, os **PoPs** e os **TreeNodes** evitam redundâncias e mantêm a eficiência da rede, assegurando que cada nó apenas reencaminha pacotes quando necessário.

```python
def handle_video_packet(client_socket, data, addr, routing_table, own_ip, socket_port):
    # Processa pacotes de vídeo e encaminha usando o routing_table
    for (client_ip, video_name) in routing_table:
        next_hop = routing_table[(client_ip, video_name)]
        if next_hop != own_ip:
            client_socket.sendto(data, (next_hop, socket_port))
            print(f"Forwarded video packet for {video_name} to {next_hop} to port {socket_port}")
```

Funcionamento dos OClient no recebimento dos Pacotes de Vídeo:

O processo de receção dos pacotes de vídeo envolve a utilização de duas funções principais. A primeira, `receive_video`, gerencia a receção dos pacotes através de um socket UDP, processando-os para exibir o vídeo em tempo real. A segunda função, `decode_frames`, é responsável por identificar e decodificar os frames a partir dos dados recebidos, assegurando que os pacotes de vídeo sejam corretamente processados e exibidos. Este conjunto de funções permite a recepção contínua e a visualização do vídeo enquanto os pacotes são transmitidos.

```python
def receive_video(client_ip, client_port):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    client_socket.bind((client_ip, client_port))  # Escuta na porta correta
    print(f"Listening for video on {client_ip}:{client_port}")

    cv2.namedWindow("Video Stream", cv2.WINDOW_NORMAL)
    video_data_buffer = b""

    try:
        while True:
            data, addr = client_socket.recvfrom(4096)
            print(f"Received {len(data)} bytes from {addr}")
            video_data_buffer += data

            # Decodifica frames
            frame, video_data_buffer = decode_frames(video_data_buffer)
            if frame is not None:
                cv2.imshow("Video Stream", frame)
                if cv2.waitKey(1) & 0xFF == ord('q'):  # Saída com tecla 'q'
                    print("Exiting video stream...")
                    break
            else:
                print("No complete frame yet. Waiting for more data...")

    except Exception as e:
        print(f"Error receiving video: {e}")
    finally:
        cv2.destroyAllWindows()
        client_socket.close()
        print("Socket closed.")
```

Esta função é responsável por receber pacotes de vídeo através de um socket UDP. Ao ser iniciada, escuta na porta especificada e exibe os frames do vídeo numa janela chamada "Video Stream". À medida que os pacotes são recebidos, os dados são acumulados num buffer e processados para reconstruir frames completos, que são exibidos em tempo real. A função permite sair do stream pressionando a tecla 'q' e garante o encerramento correto da conexão e da janela no final.

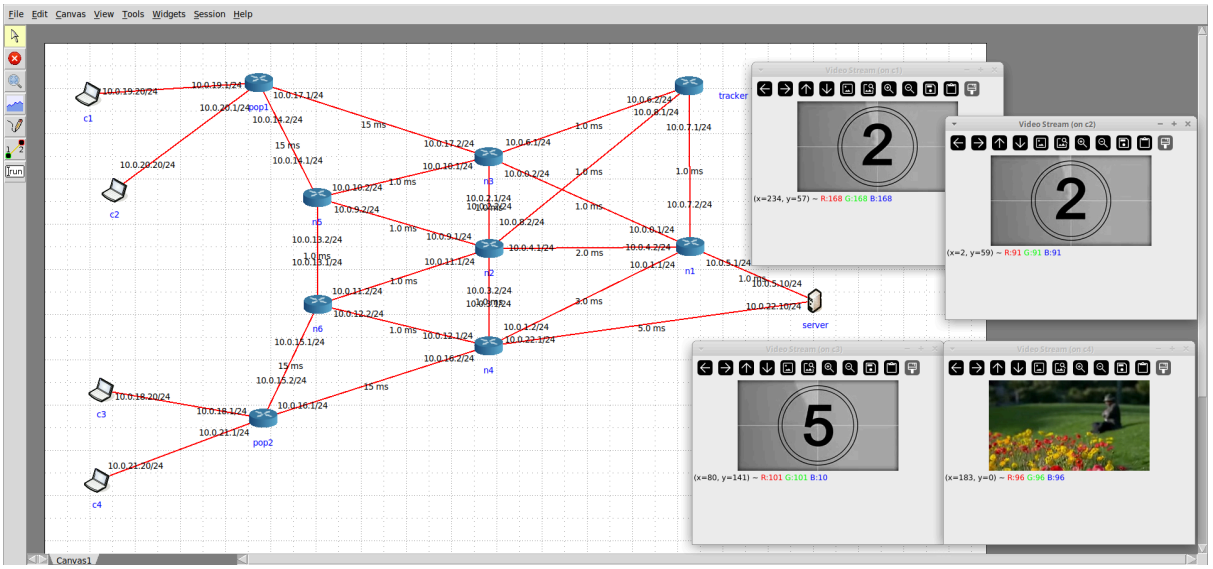
```python
def decode_frames(buffer):
    """
    Decodes JPEG frames from the buffer using FRAME delimiters.
    """
    FRAME_START = b'\xff\xd8'  # JPEG start marker
    FRAME_END = b'\xff\xd9'    # JPEG end marker

    start_idx = buffer.find(FRAME_START)
    end_idx = buffer.find(FRAME_END, start_idx)

    if start_idx != -1 and end_idx != -1:
        try:
            frame_data = buffer[start_idx:end_idx + len(FRAME_END)]
            frame = cv2.imdecode(np.frombuffer(frame_data, dtype=np.uint8), cv2.IMREAD_COLOR)
            if frame is not None:
                remaining_buffer = buffer[end_idx + len(FRAME_END):]
                return frame, remaining_buffer
        except Exception as e:
            print(f"Error decoding frame: {e}")

    # If no complete frame is found, return None and the buffer unchanged
    return None, buffer
```

A função `decode_frames` processa o buffer de dados recebidos para identificar e decodificar frames JPEG. Utiliza os marcadores de início e fim de frames (definidos por `FRAME_START` e `FRAME_END`) para localizar os dados correspondentes a um frame completo no buffer. Após identificar um frame válido, converte-o para uma imagem utilizando a biblioteca OpenCV e devolve o frame juntamente com o restante buffer não processado. Se nenhum frame completo for encontrado, retorna `None` e o buffer original.

Testes e Resultados

Problema Encontrado - Transmissão de Múltiplos Vídeos Simultaneamente:

Conforme descrito no relatório até o momento, a nossa aplicação já permitia o streaming sem falhas de um vídeo para múltiplos clientes. O problema surge quando se tenta transmitir mais de um vídeo na rede. Como os pacotes não possuem qualquer identificação, todos os pacotes, tanto os do vídeo correto como de outros vídeos em transmissão, são encaminhados pelos nós até aos clientes. Isso faz com que o cliente não consiga distinguir quais pacotes pertencem ao vídeo que está a receber.



Como podemos observar na imagem anterior, os **Client1** e **Client2**, que estão ligados ao mesmo **PoP** e requisitaram o mesmo vídeo, visualizam-no como esperado, no mesmo momento, uma vez que estão a receber os mesmos pacotes.

No entanto, nos **Client3** e **Client4**, ocorre uma contaminação dos pacotes de vídeo, com o **Client3** a consumir o vídeo solicitado pelo **Client4** e o **Client4** a consumir o vídeo solicitado pelo **Client3**.

A razão pela qual o vídeo visualizado pelos **Client1** e **Client2** não está sincronizado com o **Client3** é que, como não existe nenhum nó comum na rota de envio dos pacotes, o **Server** inicia uma nova transmissão de vídeo.

Primeira Tentativa de Solução - Implementação de Cabeçalhos nos Pacotes :

Como primeira tentativa de solução, tentamos implementar cabeçalhos nos pacotes de vídeo, de forma que cada cabeçalho contivesse a informação sobre o nome do vídeo a que o pacote pertence. Isso permitiria identificar os pacotes correspondentes ao vídeo requisitado, possibilitando o descarte dos pacotes restantes.

No entanto, após várias tentativas de implementação, nunca conseguimos realizar corretamente este processo. Embora os clientes conseguissem identificar os pacotes corretos, os frames estavam sempre mal formados. Acreditamos que o problema tenha sido causado pela adição de um cabeçalho aos pacotes de vídeo, o que corrompeu os dados do payload. Deste modo, decidimos abandonar esta solução e procurar uma alternativa.

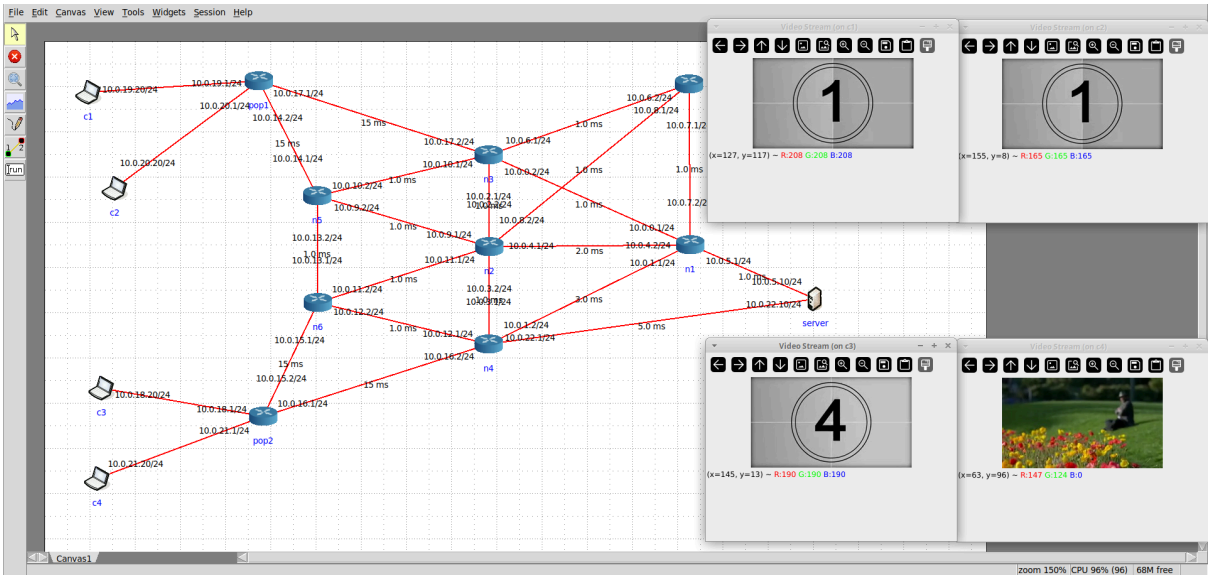
Segunda Tentativa de Solução - Utilização de um Socket para Cada Vídeo:

Após refletirmos sobre como poderíamos adaptar a nossa solução para permitir a transmissão de múltiplos vídeos simultaneamente, identificámos que o problema principal estava na mistura de pacotes de vídeos diferentes. Para resolver esta questão, decidimos criar sockets específicos para cada vídeo.

Anteriormente, tanto as mensagens de controlo como os pacotes de vídeo eram enviados e recebidos através da porta 6000. Agora, reservamos esta porta exclusivamente para as mensagens de controlo e criamos novos sockets para o envio de vídeo, um para cada vídeo.

Embora esta solução pudesse ser melhorada, devido ao tempo limitado disponível para resolver o problema, optámos por implementá-la de forma estática. Assim, a porta **6001** foi atribuída ao **video1_avc.mp4** e a porta **6002** ao **video_2_avc.mp4**.

Apesar de não ser a nossa primeira tentativa de implementação nem a solução ideal, estamos satisfeitos com o resultado, uma vez que permitiu cumprir o requisito mínimo de disponibilizar mais de um vídeo simultaneamente, sem alterar substancialmente o funcionamento do código.



Com base apenas nesta imagem, não é possível identificar diferenças em relação à solução anterior, que apresentava problemas na transmissão de múltiplos vídeos simultaneamente, uma vez que se trata apenas de uma captura de ecrã. No entanto, esta nova implementação garante que o vídeo desejado seja transmitido e visualizado corretamente no destino pretendido, conforme o esperado.

Conclusões e Trabalho Futuro

De forma geral, acreditamos ter cumprido os requisitos do projeto. No entanto, gostaríamos de ter feito mais e de ter entregue funcionalidades adicionais, uma vez que já tínhamos um plano definido para a sua implementação. Devido a alguns atrasos nas funcionalidades anteriores, que consideramos mais importantes para a entrega, algumas das funcionalidades planeadas acabaram por não ser implementadas:

Recuperação de Falha de um Nó na Rota de Transmissão: A nossa intenção era implementar um temporizador (timer) que, caso passasse um determinado tempo desde a receção do último pacote de vídeo, marcaria a rota atual como falhada. Nesse caso, um novo pedido de vídeo seria enviado, com a mesma estrutura do pedido inicial, permitindo que a nossa implementação fosse capaz de recuperar da falha.

Configuração Automática para Topologias Distintas: Atualmente, para que a nossa solução funcione, é necessário um ficheiro chamado nodes.txt, que contém todas as informações sobre a topologia. No entanto, este ficheiro poderia ser facilmente gerado de forma automática, através de um script que analisasse o ficheiro da topologia **.imn** e criasse o ficheiro **.txt** correspondente.

Topologia mais Complexa: Gostaríamos de ter criado uma topologia onde nem todos os nós fossem ONodes. A nossa intenção era implementar alguns nós intermédios, embora tal não tivesse impacto no funcionamento da aplicação.