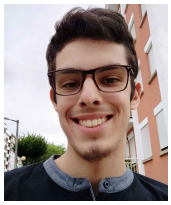
Universidade do Minho
Escola de Engenharia
Mestrado em Engenharia Informática

Unidade Curricular de Engenharia de Serviços em Rede

Ano Letivo de 2024/2025

TP2 – Serviço Over-the-top para entrega de multimédia
Grupo PL84


Flávio Silva (PG57539)


Inês Castro (PG57550)


Paula Marques (PG50676)

ESR

Introdução

Este projeto visa conceber um protótipo para entrega de vídeo com requisitos de tempo real, utilizando principalmente o emulador CORE como bancada de testes. O objetivo é transmitir conteúdos de um servidor para um conjunto de clientes, com foco na otimização da entrega dos dados.

A topologia será dividida entre a rede CDN (Content Delivery Network) e a rede de acesso. Dentro da rede CDN, um conjunto de nós atuará como intermediários no reenvio dos dados, formando uma rede de *overlay* aplicacional. A criação e manutenção dessa rede *overlay* devem ser otimizadas para garantir a entrega eficiente dos conteúdos, minimizando o atraso, número de saltos e a largura de banda necessária.

A forma como a rede *overlay* é constituída e organizada será determinante para a qualidade de serviço (QoS) que o sistema poderá suportar. Os nós localizados na fronteira entre a rede de acesso e a rede CDN são denominados *Points of Presence* (PoPs) e representam os pontos de contato entre os clientes e a rede CDN. Uma vez que o conteúdo sai da rede CDN, ele é enviado via *unicast* para os clientes que desejam visualizá-lo.

Inicialização

Linguagem escolhida

Decidimos utilizar o **Python**, pois todos os membros da equipe já possuíam familiaridade com a linguagem, além da facilidade de integração com pacotes que poderão ser úteis nas etapas futuras.

Topologia Utilizada

Para a topologia, optamos pelo exemplo descrito no enunciado, que inclui 4 clientes, 3 Pontos de Presença e 1 Host.

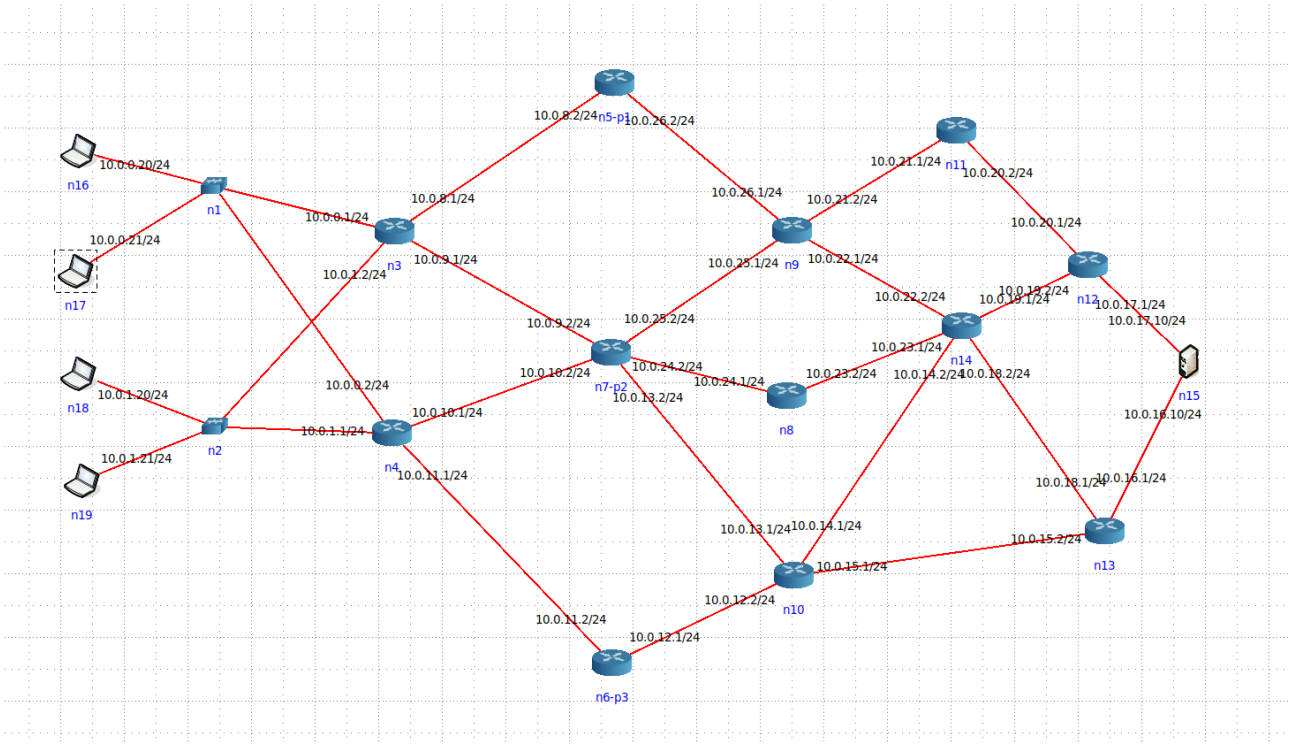


Figura 1: Topologia Final

Protocolo Utilizado na Rede Overlay

O protocolo escolhido para o *overlay* foi o **UDP**, devido à sua eficiência na transmissão de pacotes, especialmente em cenários que priorizam a velocidade em detrimento da confiabilidade. Por ser um protocolo sem conexão, o UDP permite uma comunicação mais rápida, o que o torna ideal para o nosso cenário de teste, onde a baixa latência e o alto desempenho são fatores críticos.

A única exceção será no **bootstrapper**, que utilizará o protocolo **TCP** para receber mensagens dos nós e enviar respostas de forma confiável.

Bootstrapper.py

Para implementar a rede *overlay*, decidimos introduzir um *bootstrapper* no mesmo nó onde o servidor está localizado. Este *bootstrapper* será executado em um processo ou função separado, utilizando uma porta distinta (5000). A sua principal função será criar um mapa que relaciona cada nó com seus vizinhos, incluindo o servidor.

```
neighbours = {
    'server' : ['10.0.16.1', '10.0.17.1'],
    'n1': ['10.0.0.20','10.0.1.20','10.0.7.1','10.0.5.1'],
    'n2': [ '10.0.5.2', '10.0.3.2',  '10.0.4.2','10.0.6.1'],
    'n3': ['10.0.8.2', '10.0.7.2','10.0.6.2','10.0.9.2'],
    'n4': [ '10.0.10.2', '10.0.4.1','10.0.5.2','10.0.11.2'],
    'n5': [ '10.0.8.1', '10.0.26.1'],
    'n6': [ '10.0.12.2', '10.0.11.1'],
    'n7': [ '10.0.25.1', '10.0.24.1', '10.0.9.1','10.0.10.1','10.0.13.1'],
    'n8': [ '10.0.23.1', '10.0.24.2',],
    'n9': ['10.0.21.1', '10.0.26.2','10.0.22.2','10.0.25.2'],
    'n10': ['10.0.15.2', '10.0.12.1',  '10.0.14.2','10.0.13.2'],
    'n11': ['10.0.20.1', '10.0.21.2'],
    'n12': ['10.0.19.1', '10.0.17.10', '10.0.20.2'],
    'n13': ['10.0.18.2', '10.0.16.10', '10.0.15.1'],
    'n14': ['10.0.19.2', '10.0.18.1', '10.0.14.1', '10.0.23.2', '10.0.22.1'],
    'n15': ['10.0.16.1', '10.0.17.1'],
    'n16': ['10.0.0.20', '10.0.0.1'],
    'n17': ['10.0.1.20', '10.0.1.1'],
    'n18': ['10.0.2.20', '10.0.2.1'],
    'n19': ['10.0.3.20', '10.0.3.1']
}
```

Ao iniciar, o *bootstrapper* aguardará conexões dos nós e do servidor, especificamente uma mensagem no formato `"NEIGHBORS {node_id}"`. Dessa forma, o *bootstrapper* poderá consultar o mapa, associar o *id* recebido aos vizinhos correspondentes e enviar essa lista de volta ao nó correspondente.

```
class Bootstrapper:
    def __init__(self):
        # Create socket
        bootstrapper = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        bootstrapper.bind(('0.0.0.0', 5000))

        print('Bootstrapper listening for connections!')
        try:
            while True:
                bootstrapper.listen()
                conn, addr = bootstrapper.accept()
                threading.Thread(target= self.handler, args=(conn, addr)).start()
        finally:
            bootstrapper.close()

    # Bootstrapper connection handler
    def handler(self, connection, address):
        ip = str(address[0])
        print(f"[INFO] {ip} connection started.")

        data = connection.recv(1024).decode('utf-8')

        if data.startswith('NEIGHBOURS'):

            # Return node neighbours
            _, node_id = data.split()
            response = pickle.dumps(neighbours[node_id])

        connection.send(response)
```

```
        print(f"Response sent to {ip}.")

        connection.close()
        print(f"{ip} connection closed.")
```

oNode.py

Para inicializar cada nó, criamos uma função chamada `oNode.py`, que recebe como parâmetro o nome ou *id* do nó que está sendo inicializado.

Em seguida, o nó solicita os seus vizinhos ao *bootstrapper* e aguarda requisições de outros nós na porta 6000.

Caso receba uma requisição do servidor para a construção da árvore (`"BUILDTREE"`), esta incluirá informações da árvore à medida que passa pelos nós, como o *current_flow*, o *flow_jump* (número de saltos), a latência do fluxo, bem como os vídeos ou *streams* disponíveis no servidor.

Com essas informações, o nó verificará se a árvore de distribuição construída até o momento é a mais eficiente.

Caso isso ocorra, o nó enviará essas atualizações aos seus vizinhos e atualizar a sua lista de clients para cada stream, garantindo que cada nó possua as informações mais recentes sobre a árvore de distribuição, as *streams* disponíveis, bem como os nós interessados nelas.

```
    # Handler for clients
    def handler(self, data, address):
        msg = data.decode('utf-8')
        if msg.startswith('BUILDTREE'):

            # Save better flow and send to neighbours
            ip = address[0]
            _, current_flood, t, latency, jump, streams = msg.split(':')

            self.streams_list = ast.literal_eval(streams)

            current_parent = self.flow_parent

            total_latency = float(latency) + time.time() - float(t)

            if total_latency < self.flow_latency or (total_latency == self.flow_latency
    and int(jump) + 1 <= self.jump) or current_flood > self.flow_current_flood:
                self.flow_jump = int(jump) + 1
                self.flow_parent = ip
                self.flow_latency = round(total_latency, 5)
                print(Back.BLUE + f'Arvore de distribuição construída: {self.flow_parent}
    - {self.flow_latency} - {self.flow_jump}' + Style.RESET_ALL)
            self.build_distribution_tree(ip)

            if not current_parent == self.flow_parent:
                # Cancela as streams vindas do pai :')
                for stream in self.streams:
                    self.server.sendto(f'NOSTREAM {stream}'.encode(), (current_parent,
    6000))

                # Pede as streams necessárias ao novo pai :)
                for stream in self.streams:
                    self.server.sendto(f'STREAM {stream}'.encode(), (current_parent,
    6000))
                ...
```

O Nodo também incluirá uma função que será executada em uma *thread* separada, a qual ficará a ouvir na porta 7000 por *streams*. Quando uma *stream* for recebida, ela será reencaminhada para os vários nós interessados.

```
    # Retransmit received streams packets
    def passthrough_streams(self):
        streams = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        streams.bind(('0.0.0.0', 7000))
        try:
            while True:
                data, addr = streams.recvfrom(2200)
                self.send_packet(streams, data)
        finally:
            self.server.close()

    # Send packet to clients
    def send_packet(self, socket, data):
        packet = pickle.loads(data)
        stream_id = packet['id']
        for client in self.streams[stream_id]:
            socket.sendto(data, (client, 7000))
```

Caso o cliente deseje saber a latência do nó atual (caso este seja um Ponto de Presença), ao receber uma mensagem `"PING"`, o nó enviará essa informação de volta ao cliente.

```
    elif msg.startswith('PING'):

            print(Back.BLUE + f"Received from {address[0]}: PING" + Style.RESET_ALL)
            msg = f'LATENCY:{self.flow_latency}'.encode()
            self.server.sendto(msg, address)
```

oServer.py

Após a inicialização de cada um dos nós, o servidor será inicializado. Ele irá conter informações como os *IPs* dos pontos de presença, os seus vizinhos e os videos disponíveis.

```
    # List of points of presence
pops = ['10.0.8.2', '10.0.9.2', '10.0.11.2']

class Server:
    def __init__(self):
        # List of neighbours
        self.neighbours = []

        # List of videos available
        self.videos = []

        # List of streams
        # Key:stream & Value:list of clients
        self.streams = {}

        # Create server socket
        self.server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.server.bind(('0.0.0.0', 6000))

        # RUN!!!
        self.get_neighbours_from_bootstrapper()
        self.make_list_of_videos()
        threading.Thread(target=self.build_distribution_tree).start()
        self.stream_videos()

        print(Back.LIGHTBLUE_EX + 'Server listening for requests.' + Style.RESET_ALL)
        try:
```

```
            while True:
                data, addr = self.server.recvfrom(1024)
                threading.Thread(target=self.handler, args=(data, addr)).start()
        finally:
            self.server.close()
```

Após obter seus vizinhos a partir do *bootstrapper*, o servidor criará uma lista de vídeos a partir da pasta `videos` e enviará aos seus vizinhos uma mensagem para a construção da árvore a cada 5 segundos. Esta mensagem incluirá a lista de streams disponíveis também como as informações da Árvore de Distribuição, que vão sendo atualizados por cada nodo que percorrem. Esse processo garante que a árvore de distribuição esteja regularmente atualizada e os nodos regularmente informados.

```
    # Build distribution tree - UDP
    # every 5 seconds
    def build_distribution_tree(self):
        current_flood = 1
        while(True):
            print(Back.LIGHTBLUE_EX + 'Building distribution trees.' + Style.RESET_ALL)
            for neighbour in self.neighbours:
                message = str.encode(f'BUILDTREE:{current_flood}:{time.time()}:0:0:{self.videos}') # . : horario : latência : saltos
                self.server.sendto(message, (neighbour, 6000))
            current_flood += 1
            time.sleep(5)

    # Make list of videos available to stream
    def make_list_of_videos(self):
        folder_path = './videos'
        try:
            self.videos = [file for file in os.listdir(folder_path) if os.path.isfile(os.path.join(folder_path, file)) and file != '.gitkeep']
        except Exception as e:
            print(f"An error occurred: {e}")
            self.videos = []
```

O servidor também ficará a ouvir na porta 6000 por mensagens dos nós e dos clientes. Ao receber uma mensagem `"STREAM"` ou `"NOSTREAM"`, ele irá adicionar ou remover o vizinho, de acordo com a mensagem recebida, da lista de clientes interessados na *stream* correspondente.

O servidor também receberá mensagens `"POPS"` do cliente quando este pede a lista de Pontos de Presença. Além disso, ao receber a mensagem `"PARENT"`, o servidor retornará o nó pai correspondente, que, neste caso, é o próprio servidor.

```
    # Handler for clients
    def handler(self, data, address):
        msg = data.decode('utf-8')
        if msg.startswith('STREAM'):

            # Adiciona o cliente à lista de clientes de uma stream requisitada
            client = str(address[0])
            _, stream_id = msg.split()
            if client not in self.streams[stream_id]:
                self.streams[stream_id].append(client)

        elif msg.startswith('NOSTREAM'):

            # Remove o cliente da lista de clientes de uma stream
            client = str(address[0])
            _, stream_id = msg.split()
            if stream_id in self.streams:
                self.streams[stream_id].remove(client)

        elif msg.startswith('POPS'):
```

```
21
22              # Devolve a lista de pops ao cliente
23              response = pickle.dumps(pops)
24              self.server.sendto(response, address)
25
26          elif msg.startswith('PARENT'):
27
28              # Devolve o IP do nodo pai - Neste caso devolve o próprio servidor
29              response = 'SERVER'.encode()
30              self.server.sendto(response, address)
```

oClient.py

Após a inicialização da topologia e da Árvore de Distribuição, o cliente será ativado sempre que desejar iniciar uma transmissão. Nesse momento, ele obterá a lista de Pontos de Presença (POPs) do servidor e enviará uma mensagem `"PING"` a cada um deles. O objetivo é medir regularmente a latência de cada POP, identificando aquele com o melhor desempenho. O POP selecionado será então armazenado em memória para futuras transmissões.

```
1  def __init__(self):
2          self.pops = [] # Lista pontos de presença
3          self.pop = '' # Ponto de presença a ser usado
4          self.timeout = 3 # Tempo para timeout em segundos
5
6          self.stream_choosen = ''
7          self.streams_list = []
8
9          self.get_points_of_presence()
10         threading.Thread(target=self.monitor_points_of_presence).start()
11         self.get_list_of_streams()
12
13         # Escolhe uma stream da lista de streams disponiveis
14         print('Escolha a stream:')
15         for stream in self.streams_list:
16             print(stream)
17         self.stream_choosen = input()
18
19         # Display stream
20         self.request_stream()
21         self.display_stream()
```

```
1      ### Monitor the Points of presence
2      def monitor_points_of_presence(self):
3          s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4          s.bind(('0.0.0.0', 6000))
5          s.settimeout(1)
6          while True:
7              valores = {}
8              for pop in self.pops:
9                  try:
10                     msg = str.encode('PING')
11                     start = time.time()
12                     s.sendto(msg, (pop, 6000))
13                     response = s.recv(1024)
14                     response = response.decode()
15                     if response:
16                         _, latency = response.split(':')
17                         end = time.time()
18                         volta = start - end
19                         valores[pop] = (round(volta + float(latency), 5))
20                 except socket.timeout:
21                     print(Back.LIGHTYELLOW_EX + 'Timeout - Reenvio de pedido PING.' +
    Style.RESET_ALL)
```

```
                continue

            menor = 999
            current_pop = self.pop
            for pop in valores:
                if valores[pop] < menor:
                    self.pop = pop
                    menor = valores[pop]
                    print(f'Ponto de Presença Modificado {self.pop}')

            if not current_pop == self.pop:
                self.cancel_stream() # Cancela stream vinda do pop atual
                self.request_stream() # Pede a stream ao novo pop

            time.sleep(60)
```

Após identificar o melhor Ponto de Presença (POP), o cliente enviará uma mensagem `"LISTSTREAMS"` para pedir a lista de *streams* disponíveis nesse ponto. Após receber a resposta, a lista será exibida no terminal, permitindo que o usuário escolha a *stream* desejada.

```
    # Get list of streams available to play from the Points of Presence
    def get_list_of_streams(self):
        pop_conn = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        pop_conn.settimeout(self.timeout)

        while self.pop == '':
            time.sleep(1)

        while True:
            try:
                # Envia mensagem
                message = str.encode('LISTSTREAMS')
                pop_conn.sendto(message, (self.pop, 6000))

                # Recebe lista de streams
                self.streams_list = pickle.loads(pop_conn.recv(1024))
                print(Back.GREEN + f'Lista de streams obtida com sucesso.')
                pop_conn.close()
                break
            except socket.timeout:
                print(Back.LIGHTYELLOW_EX + 'Timeout - Reenvio de pedido de lista de streams.' + Style.RESET_ALL)
                continue
            except:
                print(Back.RED + 'Servidor não está a atender pedidos.' + Style.RESET_ALL)
                pop_conn.close()
                break
```

Após o cliente selecionar a *stream* pretendida, enviará um pedido ao Ponto de Presença (POP) no formato da mensagem `"STREAM {self.stream_chosen}"`.

```
    # Request stream
    def request_stream(self):
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.bind(('0.0.0.0', 6000))

        message = str.encode(f'STREAM {self.stream_choosen}')
        sock.sendto(message, (self.pop, 6000))
```

Ao enviar a mensagem, o cliente será adicionado à lista de participantes da stream selecionada. Em seguida, ele começará a exibir a transmissão, permanecendo em escuta na porta 7000 para receber os pacotes de dados da stream correspondente. O vídeo será decodificado com o auxílio do pacote `FFmpeg`, que, por meio do comando ffplay, processa os dados de vídeo brutos e os exibe em uma janela em tempo real.

```
        # Display video from server (POP)
    def display_stream(self):
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.bind(('0.0.0.0', 7000))

        ffplay = subprocess.Popen(
            ['ffplay', '-i', 'pipe:0', '-hide_banner'],
            stdin=subprocess.PIPE,
            stderr=subprocess.DEVNULL)

        try:
            while True:
                data, _ = sock.recvfrom(2200)

                packet = pickle.loads(data)
                video = packet['data']

                ffplay.stdin.write(video)
                ffplay.stdin.flush()
        except:
            print(Back.RED + f'Erro showing stream' + Style.RESET_ALL)
        finally:
            sock.close()
            ffplay.stdin.close()
            ffplay.wait()
            self.cancel_stream()
```

Solução Final

Para resumir a nossa solução final, foi desenvolvida uma rede *overlay* que é constantemente monitorada e atualizada, garantindo a adaptação dinâmica da rede à medida que as condições de conexão variam, seja por piora ou melhora. Cada nó da rede mantém informações sobre a árvore de distribuição, incluindo dados essenciais como o número de saltos, as *streams* disponíveis e a lista de clientes/nós interessados em cada *stream*. Além disso, cada cliente realiza uma monitorização contínua dos Pontos de Presença (PoPs), avaliando regularmente a latência de cada um. Esse processo visa identificar o ponto ideal para estabelecer a conexão e acessar o conteúdo de maneira otimizada.

Inicialmente, planejamos utilizar o código-base fornecido pelo professor, que tinha como objetivo processar os *requests* e gerenciar a transmissão de vídeo em pacotes utilizando o protocolo RTP. Contudo, enfrentamos problemas de compatibilidade entre a implementação da nossa topologia/árvore e o código original fornecido. Tal incompatibilidade nos motivou a buscar uma solução alternativa que fosse mais simples e eficiente para enviar os pacotes de vídeo entre os nós e processá-los no cliente.

Dessa forma, optamos pela utilização do FFmpeg, uma ferramenta reconhecidamente robusta e prática. A sua capacidade de processar vídeos com alta eficiência, especialmente em transmissões via UDP, revelou-se fundamental para atender às nossas necessidades. Além disso, o uso do FFmpeg simplificou significativamente o processamento dos pacotes de vídeo, permitindo que focássemos no desenvolvimento e na integração da lógica da topologia/árvore, sem comprometer o desempenho ou a qualidade da transmissão.

A rede *overlay* foi projetada para suportar dois ou mais clientes simultaneamente, permitindo que cada um visualize uma *stream* de forma independente e sem qualquer comprometimento na qualidade do serviço. Dessa forma, o sistema consegue atender múltiplos usuários de forma eficiente, mantendo o desempenho e a qualidade de entrega de vídeo em tempo real.

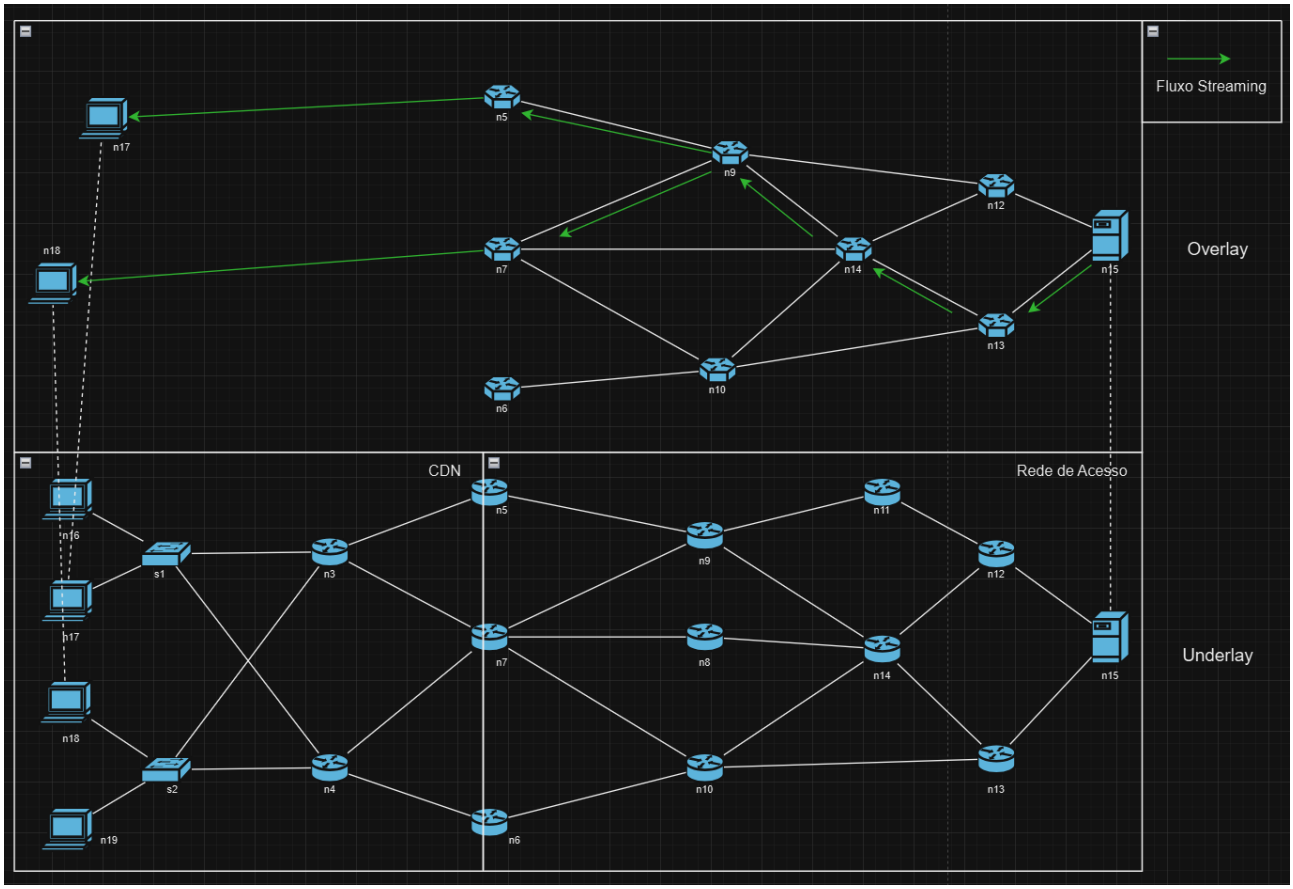


Figura 2: Overlay Final

Conclusão e Trabalho Futuro

Foi possível demonstrar a viabilidade de uma rede *overlay* aplicada à entrega de vídeos em tempo real, com foco na eficiência, redução de latência e na redução do menor número de fluxos possíveis. Contudo, algumas áreas do código podem ainda ser desenvolvidas para melhorar ainda mais a eficiencia e a confiança da transmissão de pacotes em tempo real.

Durante o desenvolvimento e testes do protótipo, alguns desafios e limitações foram identificados. Primeiramente, o sistema atual foi implementado com suporte exclusivo para o formato MJPEG. Isso limita a flexibilidade do protótipo, já que não oferece suporte a formatos de vídeo mais universais, como H.264 ou MP4/MOV, que são amplamente utilizados em sistemas de entrega de vídeo.

Também foi observado um erro ao solicitar uma *stream* no cliente após já ter sido feito um *request* para uma *stream* diferente. Esse erro resulta na mistura dos frames do primeiro vídeo com os do segundo.

Além disso, foram identificados erros aleatórios ao utilizar a porta 6000, como as mensagens de erro "address already in use"e "Broken Pipe". Embora esses erros não afetem diretamente o desempenho da transmissão, eles podem gerar confusão para o cliente que realiza o *request*, uma vez que o erro é exibido no terminal.

Também identificámos a ausência de mecanismos para recuperar a *stream* no caso de falha de um dos nós.

No trabalho futuro, será fundamental focar na implementação de suporte a novos formatos de vídeo, na correção dos erros observados e na otimização geral do sistema para garantir uma entrega de vídeo em tempo real mais eficiente e confiável, adaptando-se às diversas condições de rede e aos diferentes requisitos dos clientes.