Over-the-Top Service for Multimedia Delivery

Work Assignment 2 - Network Services Engineering - Group 83

5th of December 2023

Guilherme Sampaio
PG53851

Miguel Neiva
PG53997

João Sousa
PG50500

1. Introduction

In the ever-evolving landscape of Network Services Engineering, the implementation of Over-the-Top (OTT) services has emerged as a pivotal area of focus, particularly in the realm of real-time multimedia streaming. This second group assignment delves into the intricate design and execution of an OTT service capable of seamlessly transmitting audio and video content from one or more servers to a network of interconnected nodes, ultimately catering to diverse end clients.

The fundamental objective of this project lies in the development of a robust architecture that not only facilitates the transmission of multimedia content but also ensures its real-time delivery across a distributed network infrastructure. This necessitates a meticulous consideration of network protocols, data transfer mechanisms, and system optimization to uphold the quality and reliability of the user experience.

Therefore, this report aims to elucidate the comprehensive process involved in designing and implementing the application **mcast**. By dissecting the steps taken from conceptualization to execution, we seek to highlight the complexities and nuances involved in creating a service capable of facilitating the transmission of multimedia content to end users in real time.

2. Running the Application

The application comprises four main executables that can be generated by executing the command `make` in the terminal at the project's root directory. This command generates four executables within the `build/` directory, which are essential to run the project:

1. File **`node.run`**, launches either a Rendezvous or overlay Node based on information obtained from the Bootstrapper node.

   ```
   > build/node.run -bootstrap <addr:port_of_bootstrap>
   ```

2. File **`server.run`**, executes the Server. Running the server requires specifying its operational address and configuration file indicating the available content.

   ```
   > build/server.run -address <addr:port_of_server> -config <path_to_config>
   ```

3. File **`bootstrap.run`** executes the Bootstrapper node, responsible for providing essencial information to the other network nodes.

   ```
   > build/bootstrap.run -address <addr_port_of_bootstrap> -config <path_to_config>
   ```

4. File **`client.run`**, by executing this file and providing the content we desire, and the nearest neighbour address, we execute a request for the explicitated content.

   ```
   > build/client.run -content <content_name> -neighbour <addr:port_of_neighbour>
   ```

Note that running the nodes requires executing the Bootstrap node first, as both the rendezvous and overlay nodes depend on it for essential information.

3. Application Architecture

In this chapter, we aim to discuss the architecture of the application, discussing structural components, design patterns, and language considerations pivotal to its functionality.

3.1. Language and Design Pattern

The language selection plays a critical role on the application robustness and scalability. The two options that we had in mind were either **Go** or **Python**, ultimately choosing Go due to its better features and performance in socket programming over Python, which lacks this support in socket API's, performance and scalability.

Go's standard library supports essential networking protocols required for data streaming and its `goroutines` facilitate the simultaneous task handling, vital for real-time multimedia streaming. This, while, being simple to write and maintainable (ensuring scalable code) and being way more performant than Python.

As for the design pattern, Go by itself pushes the idea to use Concurrency-Oriented patterns (due to its nature), besides that, we employed structure composition as well as the functional options pattern in the initialization of structures.

3.2. System Components

Each element within the application operates as a distinct node, each exhibiting unique characteristics essential for consideration. This chapter delves into the specific responsibilities of each component and elucidates their behavior during service operation.

- **Overlay Nodes**: Functioning as the disseminators of content within the network, overlay nodes ensure the establishment of a diffusion tree, facilitating stream delivery to requesting clients by accommodating diverse requests.

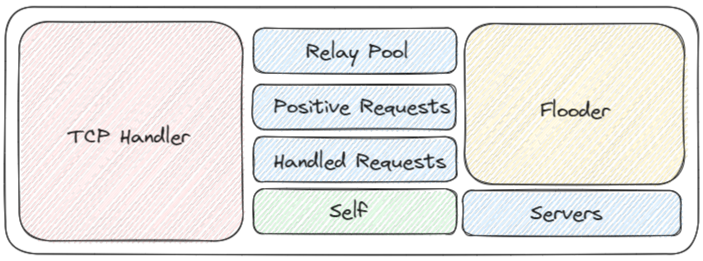
  

  Figure 1: Composition of the overlay Nodes and Rendezvous.

  Figure 1, depicts an outline of both an overlay node and a rendezvous node, highlighting their structural composition. The overlay node employs the *Flooder* and the *Positive* Requests, distinguishing it from the rendezvous node.

- **Rendezvous Node**: Singular in its presence within the network, the rendezvous node engages with servers in the absence of other content-streaming nodes. Requests converge on the rendezvous node as a final resort, typically when the requesting node shares adjacency with the rendezvous or when it marks the initial request for a specific content.

  Similar in structure to the overlay node (as illustrated in Figure 1), the rendezvous node functions in accordance with the same architectural blueprint.

- **Servers**: A pivotal component, servers form the cornerstone of content delivery by facilitating communication with the rendezvous point. They are responsible for asserting content streams and have the capacity to stream diverse forms of multimedia, albeit exclusively to the rendezvous point.

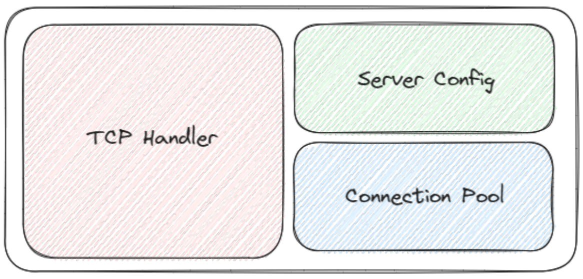


Figure 2: Composition of the Server node.

Figure 2, provides an insight into the internal structure of a Server node, highlighting the reusable nature of certain components within.

- **Bootstrapper**: Operating as a distinctive node, the bootstrapper communicates with every other node in the network, disseminating essential information concerning their positioning within the overlay. It furnishes details such as node type, neighboring nodes, and their respective operational `IPv4` addresses.

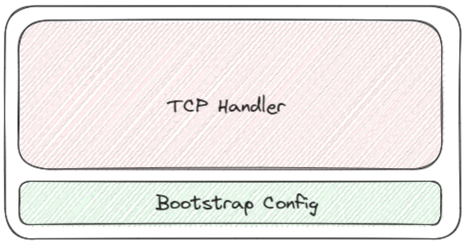


Figure 3: Composition of the Bootstrap node.

Figure 3, delineates the configuration and structure of a Bootstrap node, characterized by a request listener and a configuration framework containing vital network information.

Note: The intricates of how each of the components work are discussed in the Section 5.

4. Protocol Specifications & Communication

4.1. Protocols

The application employs three distinct communication protocols to facilitate various interactions within the network:

The Node Protocol serves as the primary communication service, enabling interactions among nodes and clients. Meanwhile, the Server Protocol specifically governs communication between the Rendezvous node and Servers. It orchestrates the communication flow and data exchange between the Rendezvous node and individual Servers operating within the network.

Additionally, the Bootstrap Protocol plays a secondary role within the system. While it is utilized, its primary functions or details won't be expounded upon in this context.

4.1.1. The Node Protocol

The Node Protocol represents a communication protocol facilitating interaction between Nodes, encompassing both Rendezvous and overlay Nodes and Clients.

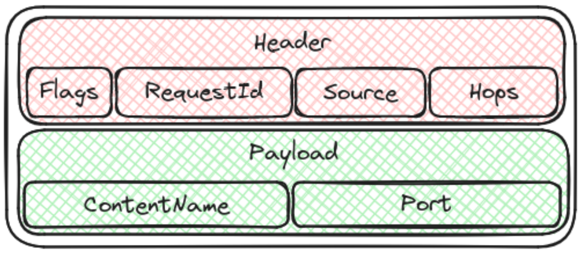


Figure 4: Node Packet.

As depicted in **Figure 4**, the protocol structure comprises two main sections: the Header and the Payload. The Header section contains critical metadata essential for every request. This metadata includes the packet flag, a unique request identification denoted by a `UUID`, the packet's source information, and the count of hops it traversed to fulfill its assignment. This information is crucial for both discovery requests and stream requests, ensuring effective communication and task completion within the network.

In the context of the Node Protocol, the Payload section provides additional pertinent information. It includes:

1. **Content Name:** This field denotes the specific content initially requested by a client. It serves as a reference to identify and locate the desired content within the network.
2. **Port Field:** This field specifies the port where a Node should be attentive or listen for a stream. It indicates the designated port at which the Node must be prepared to receive or handle the streaming content, enabling efficient communication and transmission within the network.

Flag	Origin	Description
`DISC`	`Node` `Client`	Signifies the initial phase of discovering the pathway to a node capable of streaming or holding the requested content.
`FOUND`	`Node`	Indicates that the requested content's stream has been located or found at a specific node within the network.
`MISS`	`Server`	If from nodes, it indicates that the requested content does not exist in that particular node. If from the rendezvous point, it signifies the absence of the requested content in the entire network.
`PORT`	`Node`	Specifies the port through which the content will be streamed or transmitted.
`STREAM`	`Node` `Client`	Signifies the request for streaming a specific content, initiating the streaming process for the requested content.

Table 1: Flags specification for the Node Protocol.

4.1.2. The Server Protocol

The Server Protocol is a communication protocol enabling interaction between Servers and the Rendezvous Node.

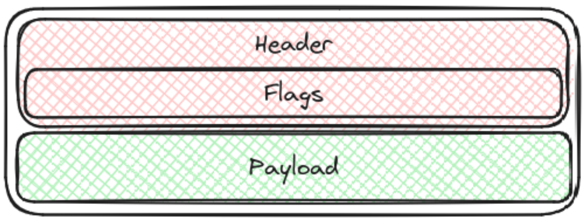


Figure 5: Packet with a generic payload.

As depicted in **Figure 5**, this protocol's packet structure is notably simpler compared to previous protocols. It primarily consists of a Header, containing essential flags required for specific operations, and a generic payload capable of accommodating various data types. This simplified structure enables efficient communication between Servers and the Rendezvous Node, offering flexibility in handling diverse data formats within the protocol.

Flag	**Origin**	**Description**
`WAKE`	`Rendezvous`	Serves as the initial flag prompting the server to provide its content list upon request.
`CONT`	`Rendezvous`	Contains information detailing the available content list along with their associated information.
`CSND`	`Server`	It specifies the port from which the server will commence streaming content in response to a request.
`STOP`	`Rendezvous`	Denotes the action of terminating an ongoing content stream from the server.
`OK`	`Rendezvous`	Signifies the receipt of the CSND flag from the server, confirming successful acknowledgment at the rendezvous point.
`PING`	`Rendezvous` `Server`	Utilized for calculating metrics, assessing network performance.

Table 2: Flags specification for the Server Protocol.

4.2. Service Communication

With our established protocols outlined in section Section 4.1, we can now delve into the operational behaviors of each component within the application and elucidate their inter-component communication, ensuring the attainment of expected outcomes.

Our approach involved the utilization of two distinct socket types:

1. **`UDP Sockets`** were employed for content delivery purposes, primarily for transmitting multimedia data. For instance, the transmission of video from a Server to a Rendezvous node exclusively utilized `UDP`. In scenarios involving video playback, complete video data transmission might not always be necessary, contingent upon the playback settings. Therefore, ensuring the arrival of packets holds a lower priority within these communication instances.
2. **`TCP Sockets`** were employed for querying various aspects of the service. For instance, the process of requesting content from a Rendezvous node to a Server relied entirely on `TCP`. Typically, steps of this nature hold critical significance for the service, warranting an assurance that requests culminate successfully without any possibility of failure.

Each service component is full-duplex, meaning that act as both a server and a client for every other node on the network.

As for the inner communication between nodes we went with a three-step approach. Firstly, we aim to discover the path to the stream by defusing `DISC` requests. Once the previous step is finished, we follow up with a `STREAM` request and finalize with a `STOP` request once we (the client) stops consuming the content being delivered. The detailed specification of the communication goes as follows:

4.2.1. The Discovery Phase

The flow of the initial phase of content discovery is depicted in the diagram represented by **Figure 6**.

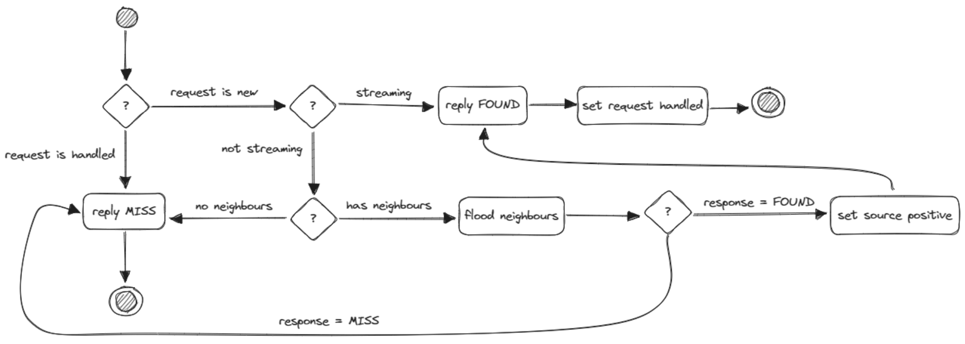


Figure 6: Activity diagram for the first step.

This process involves several sequential steps:

The node initiates by verifying if it has previously handled a request associated with the same `request_id`. If such handling has occurred, the node responds with a `MISS` message, signifying a duplicate request. Subsequently, the node checks whether it is currently streaming the requested content. If the content is being streamed, the node responds with a `FOUND` message, indicating the availability of the sought-after content.

Upon confirming the existence of neighboring nodes, the node employs a flooding mechanism by transmitting identical discovery packets to its neighbors. This mechanism serves to propagate the request throughout the network. However, it is crucial to note that the flooding process dictates that only one response is permitted to be returned.

Finally, by receiving a positive response as a result of the **flooding** process, the application stores the address from the node that provided the affirmative reply. This approach serves the purpose of maintaining the states of the request. Consequently, when transmitting a `STREAM` message, the stored address information allows each node to discern the appropriate destination to redirect the message. This storage mechanism ensures that the dissemination of `STREAM` messages occurs efficiently and accurately, directing the content to the node that initially confirmed its availability during the content discovery phase.

When a Rendezvous node verifies the availability of content on a Server, it consistently responds with a `FOUND` message, confirming the content's presence. Conversely, if the content is unavailable on the Server, the Rendezvous node responds with a `MISS` message, indicating the absence of the requested content, aligning with the expected response protocol.

4.2.2. The Streaming Phase

The flow of the second phase, representing the streaming request, is illustrated by **Figure 7**.

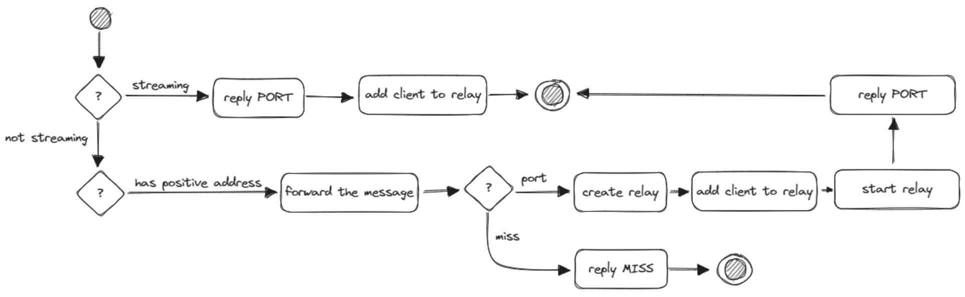


Figure 7: Activity diagram for the second step.

Having established the path leading to the designated streaming key-node during the content discovery phase, the subsequent process involves following the pre-stored addresses acquired from prior positive responses.

The initial step for the node involves checking whether it is actively streaming the requested content. This verification is crucial as there might have been subsequent requests for the same content between the initial query and this evaluation.

If the node is streaming the content, the process is straightforward. The node adds the client's address to its relay system and responds to the request with a `PORT` message. This `PORT` message indicates the specific port to which the client should listen in order to receive the transmission.

In cases where the node is not streaming the content, a **follow** process is initiated. Similar to the flooding mechanism, this follow process is recursive in nature, involving the forwarding of the received `STREAM` message to the previously registered address (the one that provided a positive response during the content discovery phase).

If the forwarded message returns a `PORT` response, the node proceeds to establish a new relay for the content. The origin of this relay is set to the node's loopback address (`0.0.0.0:<received_port>`), and the client's address is added to the relay. Subsequently, the node initiates the relay and responds to the client with a `PORT` message, indicating the designated port for content reception.

Certainly, within the system architecture, each request possesses the potential to eventually reach a Rendezvous node, considering its specialized role. Diverging from the typical follow process employed by other nodes, the Rendezvous node opts for a different approach.

Upon receiving a request, the Rendezvous node bypasses the follow process and directly communicates with the Server associated with the requested content. The Server responds by indicating the specific port through a `PORT` message, providing instructions to the Rendezvous node regarding where to listen for the transmission.

Subsequently, the Rendezvous node proceeds similarly to a standard overlay node. It initializes communication by listening on the designated port, allowing the subsequent operations to mirror the standard procedures observed within the system for content transmission and relay to requesting clients.

4.2.3. The Stopping Phase

The diagram denoted as **Figure 8** delineates the final step in the process: when a client ceases listening to the ongoing transmission, a `STOP` message is dispatched to the nearest neighbor.

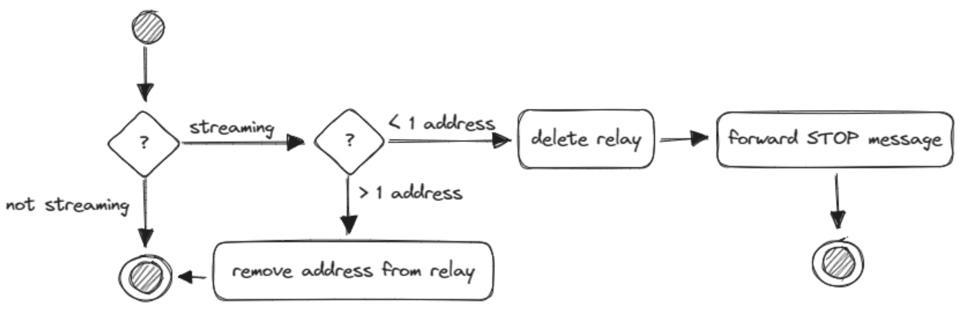


Figure 8: Activity diagram for the third step.

Upon receiving a `STOP` request, the node initiates a sequence of checks. Initially, it verifies whether it has already ceased streaming the content, employing a similar rationale as in prior stages. If the node has already stopped streaming, no further action is required, and the node concludes its processing without additional confirmation or follow-up messages.

However, if the node is still in the streaming state, two scenarios may arise. If the node is streaming to one or more addresses, it proceeds to remove the address of the client's source from the relay. Conversely, if the node is streaming solely to the client making the stop request, the relay responsible for the transmission is deleted entirely. Subsequently, the node employs the aforementioned follow process to forward the `STOP` packet, ensuring that the stream ceases effectively. This approach guarantees a comprehensive halt to the stream, even in scenarios where a single client is receiving the content.

In scenarios where a request reaches the Rendezvous point and the relay exclusively contains one registered address (implying a singular client receiving the transmission), the Rendezvous node executes a definitive termination of the transmission. To achieve this, the Rendezvous node dispatches a `TEARDOWN` message directly to the pertinent Server responsible for the ongoing transmission. This action effectively prompts the Server to cease the transmission entirely, concluding the streaming process for the sole client associated with the relay at that particular moment.

5. Implementation

In terms of technology stack, the application was developed using `Go 1.21.1`. We leveraged the `google/uuid` package for generating unique request IDs, but the implementation predominantly relied on standard libraries provided by Go, harnessing its native functionalities and components to create an efficient and functional network architecture.

In the implementation details of the application, several key components play a vital role in ensuring its functionality. In the following sections we dive into which components are key for the application.

5.1. The Relay

The Relay, a pivotal component within the system, plays a critical role in enabling nodes to transmit content via `UDP` to multiple addresses simultaneously. Each relay is associated with a specific type of content, which necessitates the implementation of the `RelayDb` structure to manage and track individual relays.

The `Relay` structure comprises two primary components:

1. **Origin and UDP Listener**: This component designates the origin of the content and includes the associated `UDP` listener. Typically, the listener operates on the loopback address, allowing nodes, functioning as routers, to manage requests arriving from various interfaces.
2. **List of Addresses**: The relay system maintains a list of addresses to which the content is being relayed. This list enables seamless control over the destinations for content transmission, facilitating actions such as addition, removal, or verification of addresses.

Regarding relay deletion, due to the external execution of the main streaming loop within a distinct `goroutine` (thread), the system utilizes Go's channels to signal and prompt the routine to halt and subsequently close the listener. This approach ensures an orderly cessation of the relay's operations by effectively signaling the streaming routine to stop and allowing for the closure of associated resources, such as the listener.

5.2. The Streamer

The implementation of the streamer component was a challenging task, particularly concerning video streaming functionality, which posed significant time and testing challenges.

Initially, the approach involved transmitting individual video frames, which encountered a setback due to its dependence on specific video formats. Variations in frame sizes across different formats made it impractical as we aimed to facilitate the streaming of any video regardless of format.

A subsequent attempt involved utilizing the Real-Time Protocol (`RTP`) for streaming. This approach utilized `ffmpeg` to generate `RTP` packets, yet it necessitated the separate transmission of video and audio streams, leading to heightened complexity within the application.

Ultimately, we settled on employing `MPEG Transport Streams (MPEG-TS)`. This choice leveraged `ffmpeg` to multiplex both video and audio into standardized 188-byte packets. Encoding video into a transport stream eliminated concerns about various video formats, and the `MPEG-TS` muxer efficiently managed the integration of video and audio streams within a single packet. However, encoding the video into a transport stream could incur a time overhead, although this could be mitigated by pre-encoding videos before initiating the service.

For playback, utilizing a format recognized by `ffmpeg` enabled the use of `ffplay` for receiving, decoding, and displaying both video and audio components of the stream. This setup provided a comprehensive solution for seamless playback of the streamed content.

Similar to the Relay, a Streamer is linked to specific content and is predominantly utilized within another structural entity known as the StreamingPool. As depicted in the architecture diagram **Figure 2**, the Server employs this structure to facilitate connections with the Rendezvous node and manages individual Streamers associated with different content streams.

While the primary function of the Server involves communication with the Rendezvous node, its implementation architecture permits the Server to concurrently stream content to multiple clients. This capability is achieved by sharing the Streamer structure among each active connection, allowing the Server to manage multiple streams efficiently. Consequently, the Server can handle multiple client connections simultaneously, utilizing the Streamer structure associated with each content stream for transmitting content to the respective clients.

5.3. The Flooder

The Flooder, an integral component present in every overlay Node, serves a fundamental purpose in disseminating information across the network. Its primary objective is to propagate a message to multiple nodes within the network.

The Flooder employs a specific strategy where it forwards a message and awaits responses from various nodes. It crucially considers only the initial positive response (`FOUND`) received from any node. This positive response is then forwarded as the Flooder's response to the requesting node. However, if no nodes respond with a `FOUND` message, signifying the presence of the requested content, the Flooder responds with a `MISS`. This `MISS` response indicates the absence of the requested content within the network.

This mechanism ensures certain key aspects:
1. In scenarios where multiple `FOUND` messages are received, the system only accounts for the first response. This approach optimizes the diffusion tree by selecting the response with the least latency, thus streamlining the propagation process.
2. If no positive responses are received, the system guarantees an outcome by responding with a `MISS` message, providing clarity regarding the content's absence within the network. This ensures that the requesting node receives a definitive response even in the absence of positive replies.

6. Solution Limits

It appears that our solution successfully encompasses steps 1 to 5 of the assignment, focusing on in-sync streaming, minimizing network overhead, content Server monitoring, and implementing functional teardown mechanisms. However, the implementation omitted step 6 due to limitations stemming from inadequate planning.

While steps 1 to 5 were executed effectively, the consideration for failure recovery mechanisms and accommodating new nodes entering the network presented challenges within the application architecture. These functionalities were deemed infeasible to integrate due to the structural design of the application, posing constraints on incorporating such features effectively.

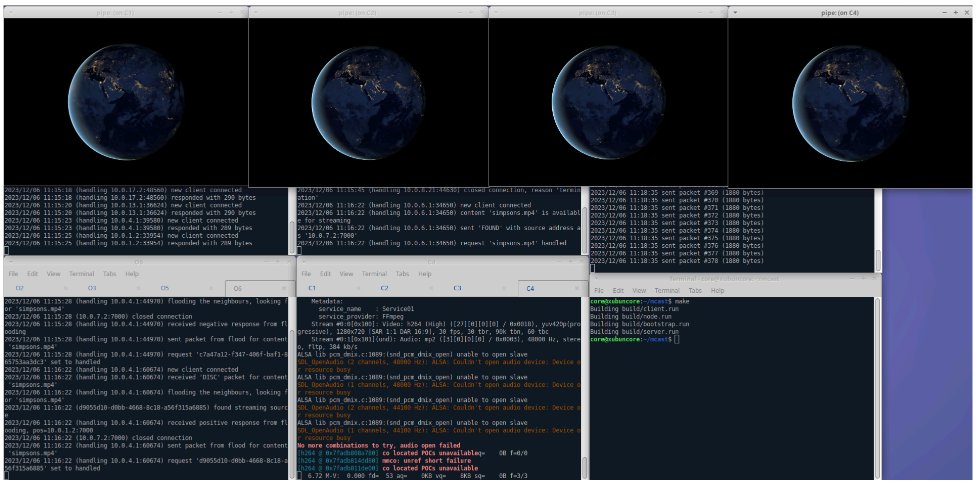
7. Tests and Results

Regarding the testing phase, our aim is to highlight the tests that best showcase the implemented functionalities. This includes scenarios like streaming to multiple clients concurrently (beyond two clients), managing streams of diverse content types simultaneously, evaluating the teardown process, and examining the effectiveness of server monitoring mechanisms.

Every test was executed on a virtual machine with 4 cores and `4GB` of `RAM` on the Core Emulator, each topology used can be found on the `resources/topologies` directory of the project, as well as each bootstrapper configuration file.

7.1. Streaming To Multiple Clients

During this test, we utilized the topology named `big_chungus` (same initial topology presented in the work assignment), which can be accessed at 'resources/topologies/big_chungus', as our foundational framework. This topology comprises two Servers, four overlay Nodes, one Rendezvous node, and four Clients. The primary objective of this test was to evaluate the system's capability to concurrently stream identical content to all four existing clients within this network configuration.



```
2023/12/06 11:15:18 (handling 10.0.17.2:48560) new client connected
2023/12/06 11:15:18 (handling 10.0.17.2:48560) responded with 290 bytes
2023/12/06 11:15:20 (handling 10.0.13.1:36624) new client connected
2023/12/06 11:15:20 (handling 10.0.13.1:36624) responded with 290 bytes
2023/12/06 11:15:23 (handling 10.0.4.1:39580) new client connected
2023/12/06 11:15:23 (handling 10.0.4.1:39580) responded with 289 bytes
2023/12/06 11:15:25 (handling 10.0.1.2:33954) new client connected
2023/12/06 11:15:25 (handling 10.0.1.2:33954) responded with 289 bytes
```

```
2023/12/06 11:15:45 (handling 10.0.8.21:44630) closed connection, reason 'termin
ation'
2023/12/06 11:16:22 (handling 10.0.6.1:34650) new client connected
2023/12/06 11:16:22 (handling 10.0.6.1:34650) content 'simpsons.mp4' is availabl
e for streaming
2023/12/06 11:16:22 (handling 10.0.6.1:34650) sent 'FOUND' with source address a
s '10.0.7.2:7000'
2023/12/06 11:16:22 (handling 10.0.6.1:34650) request 'simpsons.mp4' handled
```

```
2023/12/06 11:18:35 sent packet #369 (1880 bytes)
2023/12/06 11:18:35 sent packet #370 (1880 bytes)
2023/12/06 11:18:35 sent packet #371 (1880 bytes)
2023/12/06 11:18:35 sent packet #372 (1880 bytes)
2023/12/06 11:18:35 sent packet #373 (1880 bytes)
2023/12/06 11:18:35 sent packet #374 (1880 bytes)
2023/12/06 11:18:35 sent packet #375 (1880 bytes)
2023/12/06 11:18:35 sent packet #376 (1880 bytes)
2023/12/06 11:18:35 sent packet #377 (1880 bytes)
2023/12/06 11:18:35 sent packet #378 (1880 bytes)
```

```
2023/12/06 11:15:28 (handling 10.0.4.1:44970) flooding the neighbours, looking f
or 'simpsons.mp4'
2023/12/06 11:15:28 (10.0.7.2:7000) closed connection
2023/12/06 11:15:28 (handling 10.0.4.1:44970) received negative response from fl
ooding
2023/12/06 11:15:28 (handling 10.0.4.1:44970) sent packet from flood for content
 'simpsons.mp4'
2023/12/06 11:15:28 (handling 10.0.4.1:44970) request 'c7a47a12-f347-406f-baf1-8
65753aa3dc3' set to handled
2023/12/06 11:16:22 (handling 10.0.4.1:60674) new client connected
2023/12/06 11:16:22 (handling 10.0.4.1:60674) received 'DISC' packet for content
 'simpsons.mp4'
2023/12/06 11:16:22 (handling 10.0.4.1:60674) flooding the neighbours, looking f
or 'simpsons.mp4'
2023/12/06 11:16:22 (d9055d10-d0bb-4668-8c18-a56f315a6885) found streaming sourc
e
2023/12/06 11:16:22 (handling 10.0.4.1:60674) received positive response from fl
ooding, pos=10.0.1.2:7000
2023/12/06 11:16:22 (10.0.7.2:7000) closed connection
2023/12/06 11:16:22 (handling 10.0.4.1:60674) sent packet from flood for content
 'simpsons.mp4'
2023/12/06 11:16:22 (handling 10.0.4.1:60674) request 'd9055d10-d0bb-4668-8c18-a
56f315a6885' set to handled
```

```
    Metadata:
      service_name    : Service01
      service_provider: FFmpeg
    Stream #0:0[0x100]: Video: h264 (High) ([27][0][0][0] / 0x001B), yuv420p(pro
gressive), 1280x720 [SAR 1:1 DAR 16:9], 30 fps, 30 tbr, 90k tbn, 60 tbc
    Stream #0:1[0x101](und): Audio: mp2 ([3][0][0][0] / 0x0003), 48000 Hz, stere
o, fltp, 384 kb/s
ALSA lib pcm_dmix.c:1089:(snd_pcm_dmix_open) unable to open slave
SDL_OpenAudio (2 channels, 48000 Hz): ALSA: Couldn't open audio device: Device o
r resource busy
ALSA lib pcm_dmix.c:1089:(snd_pcm_dmix_open) unable to open slave
SDL_OpenAudio (1 channels, 48000 Hz): ALSA: Couldn't open audio device: Device o
r resource busy
ALSA lib pcm_dmix.c:1089:(snd_pcm_dmix_open) unable to open slave
SDL_OpenAudio (2 channels, 44100 Hz): ALSA: Couldn't open audio device: Device o
r resource busy
ALSA lib pcm_dmix.c:1089:(snd_pcm_dmix_open) unable to open slave
SDL_OpenAudio (1 channels, 44100 Hz): ALSA: Couldn't open audio device: Device o
r resource busy
No more combinations to try, audio open failed
[h264 @ 0x7fadb808a780] co located POCs unavailableq=    0B f=0/0
[h264 @ 0x7fadb814dd80] mmco: unref short failure
[h264 @ 0x7fadb811de00] co located POCs unavailable
   6.72 M-V:  0.000 fd=  53 aq=    0KB vq=    8KB sq=    0B f=3/3
```

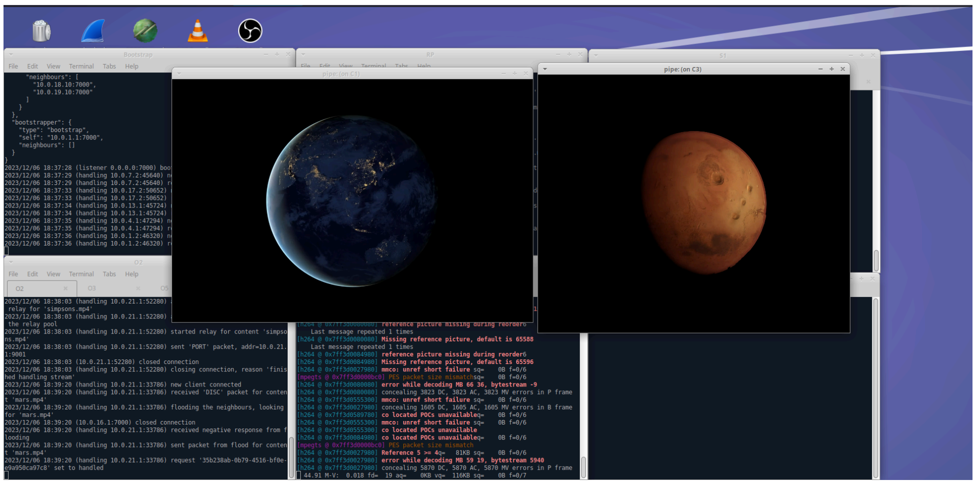
```
core@xubuncore:~/mcast$ make
Building build/client.run
Building build/node.run
Building build/bootstrap.run
Building build/server.run
core@xubuncore:~/mcast$
```

Figure 9: Streaming to more than two clients.

Verifying the accuracy of the execution demonstrated in **Figure 9** is challenging due to the substantial volume of log entries resulting from the connection of four clients. However, an observation is evident: all four videos are playing the same content simultaneously, indicating successful and synchronized playback across the four clients.

7.2. Streaming Different Contents

Similar to the previous test scenario, we employed the `big_chungus` topology, maintaining the identical count of Nodes, Servers, and Clients. However, the objective of this test differed, focusing on assessing the system's capability to handle the simultaneous streaming of two distinct videos.



```
    "neighbours": [
      "10.0.18.10:7000",
      "10.0.19.10:7000"
    ]
   }
  },
  "bootstrapper": {
    "type": "bootstrap",
    "self": "10.0.1.1:7000",
    "neighbours": []
  }
}
2023/12/06 18:37:28 (listener 0.0.0.0:7000) boo
2023/12/06 18:37:29 (handling 10.0.7.2:45640) n
2023/12/06 18:37:29 (handling 10.0.7.2:45640) r
2023/12/06 18:37:33 (handling 10.0.17.2:50652)
2023/12/06 18:37:33 (handling 10.0.17.2:50652)
2023/12/06 18:37:34 (handling 10.0.13.1:45724)
2023/12/06 18:37:34 (handling 10.0.13.1:45724)
2023/12/06 18:37:35 (handling 10.0.4.1:47294) n
2023/12/06 18:37:35 (handling 10.0.4.1:47294) r
2023/12/06 18:37:36 (handling 10.0.1.2:46320) n
2023/12/06 18:37:36 (handling 10.0.1.2:46320) r
```

```
2023/12/06 18:38:03 (handling 10.0.21.1:52280)
 relay for 'simpsons.mp4'
2023/12/06 18:38:03 (handling 10.0.21.1:52280)
 the relay pool
2023/12/06 18:38:03 (handling 10.0.21.1:52280) started relay for content 'simpso
ns.mp4'
2023/12/06 18:38:03 (handling 10.0.21.1:52280) sent 'PORT' packet, addr=10.0.21.
1:9001
2023/12/06 18:38:03 (10.0.21.1:52280) closed connection
2023/12/06 18:38:03 (handling 10.0.21.1:52280) closing connection, reason 'finis
hed handling stream'
2023/12/06 18:39:20 (handling 10.0.21.1:33786) new client connected
2023/12/06 18:39:20 (handling 10.0.21.1:33786) received 'DISC' packet for conten
t 'mars.mp4'
2023/12/06 18:39:20 (handling 10.0.21.1:33786) flooding the neighbours, looking
for 'mars.mp4'
2023/12/06 18:39:20 (10.0.16.1:7000) closed connection
2023/12/06 18:39:20 (handling 10.0.21.1:33786) received negative response from f
looding
2023/12/06 18:39:20 (handling 10.0.21.1:33786) sent packet from flood for conten
t 'mars.mp4'
2023/12/06 18:39:20 (handling 10.0.21.1:33786) request '35b238ab-8b79-4516-bf0e-
e9a950ca97c8' set to handled
```

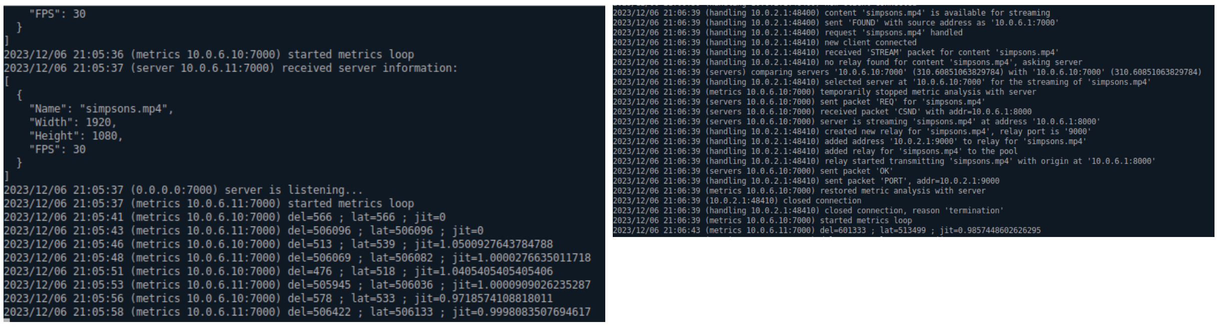
```
    Last message repeated 1 times
[h264 @ 0x7ff3d0080080] Missing reference picture, default is 65588
    Last message repeated 1 times
[h264 @ 0x7ff3d0084980] reference picture missing during reorder6
[h264 @ 0x7ff3d0084980] Missing reference picture, default is 65596
[h264 @ 0x7ff3d0027980] mmco: unref short failure sq=    0B f=0/6
[mpegts @ 0x7ff3d0000bc0] PES packet size mismatchsq=    0B f=0/6
[h264 @ 0x7ff3d0080080] error while decoding MB 66 36, bytestream -9
[h264 @ 0x7ff3d0080080] concealing 3823 DC, 3823 AC, 3823 MV errors in P frame
[h264 @ 0x7ff3d0555300] mmco: unref short failure sq=    0B f=0/6
[h264 @ 0x7ff3d0027980] concealing 1605 DC, 1605 AC, 1605 MV errors in B frame
[h264 @ 0x7ff3d0589780] co located POCs unavailableq=    0B f=0/6
[h264 @ 0x7ff3d0555300] mmco: unref short failure sq=    0B f=0/6
[h264 @ 0x7ff3d0555300] co located POCs unavailable
[h264 @ 0x7ff3d0084980] co located POCs unavailableq=    0B f=0/6
[mpegts @ 0x7ff3d0000bc0] PES packet size mismatch
[h264 @ 0x7ff3d0027980] Reference 5 >= 4q=   81KB sq=    0B f=0/6
[h264 @ 0x7ff3d0027980] error while decoding MB 59 19, bytestream 5940
[h264 @ 0x7ff3d0027980] concealing 5870 DC, 5870 AC, 5870 MV errors in P frame
 44.91 M-V:  0.018 fd=  19 aq=    0KB vq=  116KB sq=    0B f=0/7
```

Figure 10: Streaming different videos.

Indeed, both videos, representing Earth and Mars, are visibly playing simultaneously side by side, affirming the successful execution of the test scenario. However, due to the substantial volume of logs generated during this process, attempting to comprehensively understand or explain the intricate details step by step would not be feasible.

7.3. Server Monitoring

In the conducted test utilizing a simplified topology found at `/resources/topologies/simple_1`, consisting of two Nodes, one Rendezvous, and two Servers, the Server monitoring aspect emerges as a critical factor in optimizing the selection of streaming Servers. The test involved contrasting the performance of two Servers: the first located at `10.0.6.10` with unrestricted bandwidth, and the second at `10.0.6.11` constrained to `256kbps` bandwidth and a `500ms` delay.



```
    "FPS": 30
  }
]
2023/12/06 21:05:36 (metrics 10.0.6.10:7000) started metrics loop
2023/12/06 21:05:37 (server 10.0.6.11:7000) received server information:
[
  {
    "Name": "simpsons.mp4",
    "Width": 1920,
    "Height": 1080,
    "FPS": 30
  }
]
2023/12/06 21:05:37 (0.0.0.0:7000) server is listening...
2023/12/06 21:05:37 (metrics 10.0.6.11:7000) started metrics loop
2023/12/06 21:05:41 (metrics 10.0.6.10:7000) del=566 ; lat=566 ; jit=0
2023/12/06 21:05:43 (metrics 10.0.6.11:7000) del=506096 ; lat=506096 ; jit=0
2023/12/06 21:05:46 (metrics 10.0.6.10:7000) del=513 ; lat=539 ; jit=1.0500927643784788
2023/12/06 21:05:48 (metrics 10.0.6.11:7000) del=506069 ; lat=506082 ; jit=1.0000276635011718
2023/12/06 21:05:51 (metrics 10.0.6.10:7000) del=476 ; lat=518 ; jit=1.0405405405405406
2023/12/06 21:05:53 (metrics 10.0.6.11:7000) del=505945 ; lat=506036 ; jit=1.0000909026235287
2023/12/06 21:05:56 (metrics 10.0.6.10:7000) del=578 ; lat=533 ; jit=0.9718574108818011
2023/12/06 21:05:58 (metrics 10.0.6.11:7000) del=506422 ; lat=506133 ; jit=0.9998083507694617
```

```
2023/12/06 21:06:39 (handling 10.0.2.1:48400) content 'simpsons.mp4' is available for streaming
2023/12/06 21:06:39 (handling 10.0.2.1:48400) sent 'FOUND' with source address as '10.0.6.1:7000'
2023/12/06 21:06:39 (handling 10.0.2.1:48400) request 'simpsons.mp4' handled
2023/12/06 21:06:39 (handling 10.0.2.1:48410) new client connected
2023/12/06 21:06:39 (handling 10.0.2.1:48410) received 'STREAM' packet for content 'simpsons.mp4'
2023/12/06 21:06:39 (handling 10.0.2.1:48410) no relay found for content 'simpsons.mp4', asking server
2023/12/06 21:06:39 (servers) comparing servers '10.0.6.10:7000' (310.60851063829784) with '10.0.6.10:7000' (310.60851063829784)
2023/12/06 21:06:39 (handling 10.0.2.1:48410) selected server at '10.0.6.10:7000' for the streaming of 'simpsons.mp4'
2023/12/06 21:06:39 (metrics 10.0.6.10:7000) temporarily stopped metric analysis with server
2023/12/06 21:06:39 (servers 10.0.6.10:7000) sent packet 'REQ' for 'simpsons.mp4'
2023/12/06 21:06:39 (servers 10.0.6.10:7000) received packet 'CSND' with addr=10.0.6.1:8000
2023/12/06 21:06:39 (servers 10.0.6.10:7000) server is streaming 'simpsons.mp4' at address '10.0.6.1:8000'
2023/12/06 21:06:39 (handling 10.0.2.1:48410) created new relay for 'simpsons.mp4', relay port is '9000'
2023/12/06 21:06:39 (handling 10.0.2.1:48410) added address '10.0.2.1:9000' to relay for 'simpsons.mp4'
2023/12/06 21:06:39 (handling 10.0.2.1:48410) added relay for 'simpsons.mp4' to the pool
2023/12/06 21:06:39 (handling 10.0.2.1:48410) relay started transmitting 'simpsons.mp4' with origin at '10.0.6.1:8000'
2023/12/06 21:06:39 (servers 10.0.6.10:7000) sent packet 'OK'
2023/12/06 21:06:39 (handling 10.0.2.1:48410) sent packet 'PORT', addr=10.0.2.1:9000
2023/12/06 21:06:39 (metrics 10.0.6.10:7000) restored metric analysis with server
2023/12/06 21:06:39 (10.0.2.1:48410) closed connection
2023/12/06 21:06:39 (handling 10.0.2.1:48410) closed connection, reason 'termination'
2023/12/06 21:06:39 (metrics 10.0.6.10:7000) started metrics loop
2023/12/06 21:06:43 (metrics 10.0.6.11:7000) del=601333 ; lat=513499 ; jit=0.9857448602626295
```

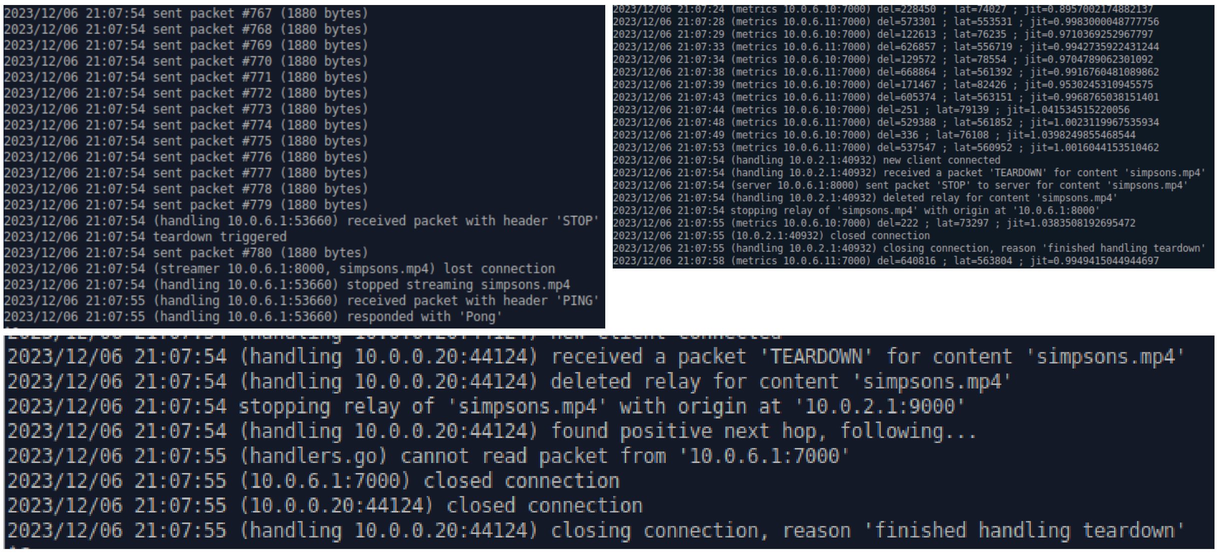
Figure 11: Logs generated by the Rendezvous Node when measuring metrics.

Examining the image on the right side (**Figure 11**), we observe the Rendezvous Node's automatic initiation of metrics measurement upon startup. Metrics including delay (the duration between a request and its corresponding response), latency (the average delay time), and jitter (the variance in latency) are measured in microseconds. As anticipated, the metrics for the Server at `10.0.6.11` exhibit significantly higher values compared to the other Server.

In the image on the left side, following a stream request, the Rendezvous Node strategically selects the server at `10.0.6.10` due to its inferior metric coefficient (less is better). Calculating the metrics involves assigning weights to latency (60%) and jitter (40%) to compute a metric coefficient aiding in the optimal selection of the most efficient streaming Server.

7.4. Teardown Process

In the conducted test, utilizing the same topology as previously mentioned, two experiments were carried out to examine the teardown process under different scenarios: one involving a single client receiving the transmission, and the other with multiple clients listening.



```
2023/12/06 21:07:54 sent packet #767 (1880 bytes)
2023/12/06 21:07:54 sent packet #768 (1880 bytes)
2023/12/06 21:07:54 sent packet #769 (1880 bytes)
2023/12/06 21:07:54 sent packet #770 (1880 bytes)
2023/12/06 21:07:54 sent packet #771 (1880 bytes)
2023/12/06 21:07:54 sent packet #772 (1880 bytes)
2023/12/06 21:07:54 sent packet #773 (1880 bytes)
2023/12/06 21:07:54 sent packet #774 (1880 bytes)
2023/12/06 21:07:54 sent packet #775 (1880 bytes)
2023/12/06 21:07:54 sent packet #776 (1880 bytes)
2023/12/06 21:07:54 sent packet #777 (1880 bytes)
2023/12/06 21:07:54 sent packet #778 (1880 bytes)
2023/12/06 21:07:54 sent packet #779 (1880 bytes)
2023/12/06 21:07:54 (handling 10.0.6.1:53660) received packet with header 'STOP'
2023/12/06 21:07:54 teardown triggered
2023/12/06 21:07:54 sent packet #780 (1880 bytes)
2023/12/06 21:07:54 (streamer 10.0.6.1:8000, simpsons.mp4) lost connection
2023/12/06 21:07:54 (handling 10.0.6.1:53660) stopped streaming simpsons.mp4
2023/12/06 21:07:55 (handling 10.0.6.1:53660) received packet with header 'PING'
2023/12/06 21:07:55 (handling 10.0.6.1:53660) responded with 'Pong'
```

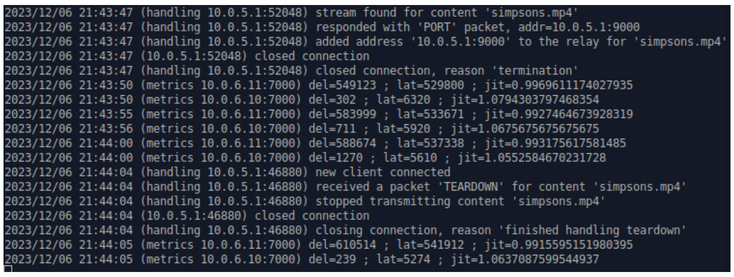
```
2023/12/06 21:07:24 (metrics 10.0.6.10:7000) del=228450 ; lat=74027 ; jit=0.8957002174882137
2023/12/06 21:07:28 (metrics 10.0.6.11:7000) del=573301 ; lat=553531 ; jit=0.9983000048777756
2023/12/06 21:07:29 (metrics 10.0.6.10:7000) del=122613 ; lat=76235 ; jit=0.9710369252967797
2023/12/06 21:07:33 (metrics 10.0.6.11:7000) del=626857 ; lat=556719 ; jit=0.9942735922431244
2023/12/06 21:07:34 (metrics 10.0.6.10:7000) del=129572 ; lat=78554 ; jit=0.9704789062301092
2023/12/06 21:07:38 (metrics 10.0.6.11:7000) del=668864 ; lat=561392 ; jit=0.9916760481089862
2023/12/06 21:07:39 (metrics 10.0.6.10:7000) del=171467 ; lat=82426 ; jit=0.9530245310945575
2023/12/06 21:07:43 (metrics 10.0.6.11:7000) del=605374 ; lat=563151 ; jit=0.9968765038151401
2023/12/06 21:07:44 (metrics 10.0.6.10:7000) del=251 ; lat=79139 ; jit=1.041534515220056
2023/12/06 21:07:48 (metrics 10.0.6.11:7000) del=529388 ; lat=561852 ; jit=1.0023119967535934
2023/12/06 21:07:49 (metrics 10.0.6.10:7000) del=336 ; lat=76108 ; jit=1.0398249855468544
2023/12/06 21:07:53 (metrics 10.0.6.11:7000) del=537547 ; lat=560952 ; jit=1.0016044153510462
2023/12/06 21:07:54 (handling 10.0.2.1:40932) new client connected
2023/12/06 21:07:54 (handling 10.0.2.1:40932) received a packet 'TEARDOWN' for content 'simpsons.mp4'
2023/12/06 21:07:54 (server 10.0.6.1:8000) sent packet 'STOP' to server for content 'simpsons.mp4'
2023/12/06 21:07:54 (handling 10.0.2.1:40932) deleted relay for content 'simpsons.mp4'
2023/12/06 21:07:54 stopping relay of 'simpsons.mp4' with origin at '10.0.6.1:8000'
2023/12/06 21:07:55 (metrics 10.0.6.10:7000) del=222 ; lat=73297 ; jit=1.0383508192695472
2023/12/06 21:07:55 (10.0.2.1:40932) closed connection
2023/12/06 21:07:55 (handling 10.0.2.1:40932) closing connection, reason 'finished handling teardown'
2023/12/06 21:07:58 (metrics 10.0.6.11:7000) del=640816 ; lat=563804 ; jit=0.9949415044944697
```

```
2023/12/06 21:07:54 (handling 10.0.0.20:44124) received a packet 'TEARDOWN' for content 'simpsons.mp4'
2023/12/06 21:07:54 (handling 10.0.0.20:44124) deleted relay for content 'simpsons.mp4'
2023/12/06 21:07:54 stopping relay of 'simpsons.mp4' with origin at '10.0.2.1:9000'
2023/12/06 21:07:54 (handling 10.0.0.20:44124) found positive next hop, following...
2023/12/06 21:07:55 (handlers.go) cannot read packet from '10.0.6.1:7000'
2023/12/06 21:07:55 (10.0.6.1:7000) closed connection
2023/12/06 21:07:55 (10.0.0.20:44124) closed connection
2023/12/06 21:07:55 (handling 10.0.0.20:44124) closing connection, reason 'finished handling teardown'
```

Figure 12: Logs generated by a teardown message.

As for the first experiment, the illustration **Figure 12** depicts the nodes of the topology as terminals, allowing inspection of their respective logs. In this representation, the streaming Server log is situated on the top-right, the Rendezvous Node log on the top-left, and the Node forwarding the transmission from the Rendezvous to the Client at the bottom.

Analyzing the logs, the teardown process unfolds as described in Section 4.2.3 for the given circumstances (only one client listening to the stream).



```
2023/12/06 21:43:47 (handling 10.0.5.1:52048) stream found for content 'simpsons.mp4'
2023/12/06 21:43:47 (handling 10.0.5.1:52048) responded with 'PORT' packet, addr=10.0.5.1:9000
2023/12/06 21:43:47 (handling 10.0.5.1:52048) added address '10.0.5.1:9000' to the relay for 'simpsons.mp4'
2023/12/06 21:43:47 (10.0.5.1:52048) closed connection
2023/12/06 21:43:47 (handling 10.0.5.1:52048) closed connection, reason 'termination'
2023/12/06 21:43:50 (metrics 10.0.6.11:7000) del=549123 ; lat=529800 ; jit=0.9969611174027935
2023/12/06 21:43:50 (metrics 10.0.6.10:7000) del=302 ; lat=6320 ; jit=1.0794303797468354
2023/12/06 21:43:55 (metrics 10.0.6.11:7000) del=583999 ; lat=533671 ; jit=0.9927464673928319
2023/12/06 21:43:56 (metrics 10.0.6.10:7000) del=711 ; lat=5920 ; jit=1.0675675675675675
2023/12/06 21:44:00 (metrics 10.0.6.11:7000) del=588674 ; lat=537338 ; jit=0.993175617581485
2023/12/06 21:44:00 (metrics 10.0.6.10:7000) del=1270 ; lat=5610 ; jit=1.0552584670231728
2023/12/06 21:44:04 (handling 10.0.5.1:46880) new client connected
2023/12/06 21:44:04 (handling 10.0.5.1:46880) received a packet 'TEARDOWN' for content 'simpsons.mp4'
2023/12/06 21:44:04 (handling 10.0.5.1:46880) stopped transmitting content 'simpsons.mp4'
2023/12/06 21:44:04 (10.0.5.1:46880) closed connection
2023/12/06 21:44:04 (handling 10.0.5.1:46880) closing connection, reason 'finished handling teardown'
2023/12/06 21:44:05 (metrics 10.0.6.11:7000) del=610514 ; lat=541912 ; jit=0.9915595151980395
2023/12/06 21:44:05 (metrics 10.0.6.10:7000) del=239 ; lat=5274 ; jit=1.0637087599544937
```

Figure 13: Streaming to more than two clients.

Figure 13 depicts the logs when a teardown message was processed by the Rendezvous node (that was streaming to two clients). As we can see, it also went by Section 4.2.3 since it only removed the client abandonning the streaming tree.

8. Conclusion

Our OTT service implementation successfully achieved real-time multimedia content delivery, showcasing synchronized streaming to multiple clients, diverse content handling, and robust teardown processes. Leveraging Go's capabilities, we designed a scalable architecture with distinct nodes and effective communication protocols.

Despite its achievements, limitations in adapting to dynamic network changes and implementing fault tolerance mechanisms surfaced. Future enhancements could focus on these aspects to bolster scalability and resilience within evolving network landscapes.