UCLouvain

epl

École polytechnique de Louvain

Low-latency live video streaming for mass-interactive experiences

Author: **Louis ADRIAENS**
Supervisors: **Benoît MACQ, Sébastien LUGAN**
Reader: **Ramin SADRE**
Academic year 2020–2021
Master [120] in Electrical Engineering

Acknowledgements

I would like to thank my supervisors Benoît Macq and Sébastien Lugan for their advice and help throughout this work. I also want to thank M. Lionel Dawson for his participation and sound advice. They put me on the right track at the beginning of my effort, and it helped me to go through until the end.

I thank also my reader professor Ramin Sadre for taking the time to judge my thesis.

Finally and most importantly, I want to thank my family for all the love and support they gave me during all my years at university.

Abstract

The demand for ever more performing live video streaming only increases with every new release of technology in this fast-evolving industry. In this thesis, we explore the multiple ways of deploying a live video streaming system that would allow several cinema rooms to play a multiplayer crowd interactive game. The goal is to design a model that would allow the exchange of multiple streams, including video, between the participating theaters, with very low latency requirements and real-time monitoring and adaptation to the network conditions. In order to comply with the low latency demands, the new standard JPEG XS is given for the compression of the video. The literature detailing the different possible topologies and components of such a system made us choose a Selective Forwarding Unit (SFU) architecture. Based on the low complexity and low latency promises of JPEG XS, we placed encoders in the SFU to adapt to the network quality. We then proposed an implementation of the selected model and analyzed whether our model is the most adapted to answer the initial problem. We come to the conclusion that the principle of the SFU is indeed the best suited for the redirection of live streams between clients, but that the results are insufficient to validate the use of transcoding in the central server.

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Problem statement . . . 2
- 1.3 Structure of the thesis . . . 4

2 Theoretical concepts **5**
- 2.1 Video streaming . . . 5
  - 2.1.1 Coding . . . 7
  - 2.1.2 Packetization . . . 9
  - 2.1.3 Network layer . . . 11
  - 2.1.4 Jitter buffer . . . 13
- 2.2 Video network topologies . . . 14
  - 2.2.1 Peer-to-Peer system . . . 14
  - 2.2.2 Client-Server system . . . 15
- 2.3 Content Delivery Networks . . . 18
  - 2.3.1 Mode of operation . . . 19
  - 2.3.2 Hybrid architectures . . . 21

3 Model description **23**
- 3.1 System architecture . . . 23
- 3.2 Transmission chain . . . 25
  - 3.2.1 Adaptive bitrate coding for video . . . 26
  - 3.2.2 Other data streams . . . 29
- 3.3 Signaling . . . 30
  - 3.3.1 Background . . . 31
  - 3.3.2 Dynamic pipeline building . . . 31
- 3.4 Scalability . . . 32

4 Implementation **35**
4.1 Implementation tool: Gstreamer . . . 35
4.2 Implementation . . . 37
4.2.1 The signaling component . . . 37
4.2.2 Sequence of operations . . . 38
4.2.3 Gstreamer elements . . . 39
4.2.4 Adaptive bitrate algorithm . . . 44
4.2.5 Limitations of the implementation . . . 45

5 Evaluation **47**
5.1 Environment . . . 47
5.2 Results of the tests . . . 48
5.2.1 CPU usage . . . 48
5.2.2 Latency . . . 52
5.3 Discussion of the results . . . 55

6 Conclusion **57**
6.1 Review of the guidelines . . . 58
6.2 Areas for improvement . . . 59

Bibliography **61**

A Graphics of the implemented pipelines **69**
A.1 Full pipelines of the implementation . . . 69
A.1.1 Sending pipeline of the client . . . 69
A.1.2 Receiving pipeline of the client . . . 75
A.1.3 Pipeline in the SFU, with encoders . . . 80
A.2 Pipelines used in measurements . . . 86
A.2.1 CPU measurements . . . 86
A.2.2 Latency measurements . . . 86

List of Figures

2.1 Model of our transmission chain 6
2.2 RTP header format 10
2.3 RTCP packet format 11
2.4 Full-mesh topology 15
2.5 Multipoint Control Unit (MCU) topology 16
2.6 Selective Forwarding Unit (SFU) topology 17
2.7 Principle of simulcast. [1] 18
2.8 Principle of Scalable Video Coding. [2] 18
2.9 Main system components of a Content Delivery Network 20

3.1 Sending client's video transmission chain 25
3.2 Receiving client's video transmission chain 25
3.3 Transmission chain for the video stream in the SFU, communication between 3 clients 28
3.4 Full arrangement of the transmission chains in the SFU for a session between three theaters 30
3.5 Signaling phase leading to the creation of a new media pipeline in the SFU. 32

4.1 Entities that make up the Gstreamer framework [3] 36
4.2 Entities of the system and the communication paths between them. 38
4.3 Sequence diagram, beginning of the session, for 2 clients. 39
4.4 The Gstreamer elements in charge of the transmission: the *rtpbin*, *udpsink* and *udpsrc*. 40
4.5 Structure of the way the clients are stored in the memory of the SFU. 40
4.6 Structure of the way the other peers of the session are stored in the clients' memory. 41
4.7 The dynamic building of the pipelines and the problem encountered. 42

4.8 Substitute solution, with 3 clients in the session. 43

5.1 Graphs of the CPU load over time for an SFU with transcoding. . 50
5.2 *htop* showing the CPU load for an SFU with transcoding. 50
5.3 Graphs of the CPU load over time for an SFU without transcoding. 51
5.4 Latency test: same video stream displayed at the sender and at the receiver. 53
5.5 Example of the ouptut of *markout* on the log. This is for the *rtph264depay* element. 54
5.6 *markout* error when multiple branches present. 54
5.7 Output of *mark* plugin for the RTP encapsulation element, *rtph264pay*. 54
5.8 Output of *mark* plugin for the H264 decoder, *avedc_h264*. 55

A.1 Sending pipeline of the client, part 1 70
A.2 Sending pipeline of the client, part 2 71
A.3 Sending pipeline of the client, part 3 72
A.4 Sending pipeline of the client, part 4 73
A.5 Sending pipeline of the client, part 5 74
A.6 Receiving pipeline of the client, part 1 76
A.7 Receiving pipeline of the client, part 2 77
A.8 Receiving pipeline of the client, part 3 78
A.9 Receiving pipeline of the client, part 4 79
A.10 Pipeline inside the SFU, with encoders, part 1 81
A.11 Pipeline inside the SFU, with encoders, part 2 82
A.12 Pipeline inside the SFU, with encoders, part 3 83
A.13 Pipeline inside the SFU, with encoders, part 4 84
A.14 Pipeline inside the SFU, with encoders, part 5 85

Chapter 1

Introduction

1.1 Context

In a world at hands with a historic pandemic, live video streaming is everywhere. People use it for work, for entertainment, for keeping in touch with each other, and more. Streaming consists in the transmission of video in a continuous fashion from a provider to an end user over the Internet, while the word "live" implies that the events being streamed are happening in real time, without first being recorded and stored. COVID-19 has increased its use enormously, forcing industries to an important transition to live streaming systems to organize work from home, and to follow events like sports, political interviews, award shows like the Oscars, etc. Because of that and because of the rising amount of High-Definition (HD) and now Ultra-High-Definition (UHD) video streaming, the part of Internet traffic occupied by video transmission has kept increasing over the years and is predicted to take up to 82% of all Internet traffic by 2022 [4][5].

Another field, which has been using live video transmission for a long time, is also experiencing a rise in popularity over the past year: the field of online video gaming. It is declined in many forms and types, from mobile gaming like *Clash of Clans* to console first-person shooters. In addition to playing the actual game, young people of age 18-25 now prefer to watch video games rather than watching TV or movies [6], the leading platform for this activity, Twitch, registering an increase of 50 % of time spent watching gaming content just between March and April 2020 [7].

Being so popular, online video games have high expectations from users in terms of performances and experience. When playing, the main requirement is high responsiveness from the game. This means that the end-to-end delay between the player and the gaming server must be as low as possible. On the other hand, when watching a streamed gameplay, the quality and the continuity

of the video, *i.e.* no loading or buffering time, are of higher importance. All the more so since video quality keeps increasing with the years. Indeed, we now see for example the arrival of technologies like HDR (High Dynamic Range) and 8K being commercialized and even available on YouTube videos. And with it, viewers become more aware and demanding about video quality than they used to be. However, transmitting video of higher quality over the Internet requires a very high amount of bandwidth resources. To solve this limited bandwidth problem, costly network improvements are being made and new, more advanced compression algorithms keep being published every year. Still, some areas of the world do not have high-speed Internet capable of handling these high video qualities, and viewers located in these areas are then unable to enjoy such new high-quality video. Therefore, modern video streaming systems must be efficient in terms of bandwidth management. In addition, they must be capable of adapting the quality of the streamed content to the condition of the network. Added to high video quality demands, platforms like Twitch also include a *chat* to allow for interaction between streamers and viewers. So, delay must also be minimized for such applications.

In the last years, new gaming situations and technologies have been under research. Among them, multi-user systems, where an important number of users play together as one. This has evolved over the time into crowd or mass interaction computing, for which a "crowd" of a very large number of users is recognized as a coherent entity engaging into a single activity with a large infrastructure or interactive system [8]. The Belgian *spin-off* Skemmi [9] has implemented that concept by having a full crowd of spectators in a cinema room interacting together with a video game projected on the big screen. In its next step, Skemmi achieved a multiplayer crowd interactive system when they managed to have three cinema rooms competing against each other in a race for the release of the Disney Pixar movie *Cars 3*.

1.2 Problem statement

The live crowd interaction games of Skemmi include the live streaming of a video of the spectators in the room as well as the transmission of streams related to the progress of the game, like the data captured by the movement sensors in the place. The communication between the different endpoints that are the cinema rooms have the same requirements of low latency, jitter, and good image quality as any video gaming or streaming system described in the previous section. In this thesis, we want to study how to achieve such mass interactive gaming and build a system capable of completing that task. To be more precise, the system has the following expectations:

- **(Very) low latency**: To provide a satisfying experience to the spectators, interactive gaming relies on very low latency. The game being projected in a cinema, a good picture quality is also expected to be received from every other participating theater. In short, the system must comply with important latency and bandwidth constraints at the same time. Naturally, we are able to impact both delay and image quality at the level of the encoder, as it is an element where a trade-off between compression complexity (and therefore latency) and compression ratio (and therefore image quality) can be found. And so, for this purpose, we are given brand new JPEG XS encoders, a new compression standard ordered by the JPEG Committee and co-created by Belgian company intoPIX [10], which was conceived to offer the best quality as well as the lowest latency and complexity in terms of hardware and software.

- **Multiple simultaneous video streams**: We want to be able to have multiple endpoints playing the game, so each cinema room should be able to broadcast video streams to each other connected node and to receive multiple video stream per another connected node.

- **Long distance**: Since cinema rooms are rarely located very close to each other, the system should be compatible with a deployment on a long distance of typically several hundreds of kilometers between each endpoint.

- **Real-time adaptation to the network condition**: Computer networks have properties that are susceptible to change at any time, be it the bandwidth, the delay, the jitter, or other. In consequence, the system should be able to automatically adapt to the degradation or improvement of the quality of the network, as we do not want the game to stall or to buffer in-play.

- **Statistics collection and real-time monitoring**: Finally, we want to be able to collect meaningful statistics about the live video streams in order to provide real-time quality assessment of the streams to the operator.

These five points will be our guidelines for this work. On all these accounts, our global objective is to explore the different techniques available in the streaming industry in order to propose an architecture that optimizes the live streaming of video streams captured simultaneously in multiple theaters. This way, the system should provide the players in the theaters real-time video feedback of the other players in the other cinema rooms.

In order to fulfill the above requirements, we must make the right choices of network topology and of transmission chain for the streaming system. The use of

the JPEG XS codec is one of them. We will, however, not delve into the intrinsic mechanics of the encoder in order to improve the transmission and will use it as a black box for the remainder of this piece of work. Many other papers have been issued researching algorithms that optimize the compression of video frames for different situations and this is not one of them. Moreover, we will consider our structure to be closed and will not include any discussion about the security of the communication either. However, it should not be difficult to add a security layer to our proposed architecture.

1.3 Structure of the thesis

We will start our research in Chapter 2 by a state-of-the-art of real-time video streaming technologies. In this chapter, we examine all the different elements necessary for building a live streaming system, exploring both video manipulation mechanisms and system topologies. These elements are necessary to understand the model that we design in Chapter 3. In this chapter, we describe the choices that we made in the design of our live streaming system and how they help us achieve the goals set at the beginning. Then, in Chapter 4, we present how we implement this architecture and explain the details and limitations of this implementation. Finally, in Chapter 5, we verify the good functioning of our solution and present some performance results to identify potential improvements and possibilities for the future. By the end of this thesis, readers should gain knowledge about the different architectures and techniques available for streaming video in real time between two and more participants, and find out what performance may be expected from such architectures with the implementation proposed in this work.

Chapter 2

Theoretical concepts

This chapter presents the state-of-the-art of video streaming. Before venturing into the topologies that we could use for our case study, an outline of the components of a streaming transmission chain is necessary. Indeed, video captured by a camera or any other video source needs to be processed before being sent on to a network, and then needs to be processed again in order to be displayed at the receiver. We will discuss in Section 2.1 the different elements playing a part in this manipulation of the video stream and see how they impact the performances of the transmission. Then we will review in Section 2.2 the existing architectures allowing for video exchange between multiple parties and compare their characteristics. This will allow us to choose the most suitable topology for our problem. Finally, in Section 2.3, we will complete the chapter by explaining the state-of-the-art of Content Delivery Networks (CDNs) and how they are used to distribute content all over the globe.

2.1 Video streaming

Our study case involves two-way communications between theaters that are exchanging multiple streams, including a video stream. Every video transmission chain consists of the elements shown on Figure 2.1. The figure shows the essential blocks needed to transform the stream of raw video captured at the source into a stream that is adapted to be sent, received, and read by the recipient. Every element of Figure 2.1 contributes to a greater or lesser extent to the end-to-end delay of the system, which is the time elapsed from when a video image is captured by a video source at the sender side until when it is displayed on a monitor at the receiver side [11]. In this section, we will review these elements and discuss how they participate in the end-to-end delay of the communication, addressing at the

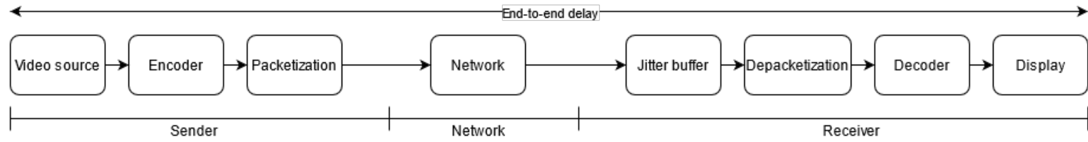


Figure 2.1: Model of our transmission chain

same time some key concepts that we will need in order to design our system.

The latency brought by an element is a measure of time, and is therefore usually measured in milliseconds. However, when talking about video, latency can also be measured in terms of *frames* or *lines*. Indeed, each element of Figure 2.1 can be seen as a buffer that stores a certain number of frames or lines of a video stream before processing them. Therefore, the greater the buffer, the greater the delay introduced, as the first bits of data flowing in an element must wait until the necessary amount of frames or lines is present in the component in order to be processed. The link between the number of frames and the time in milliseconds is expressed by the *frame rate* of the video stream. Depending on the resolution, each frame holds a certain number of lines that therefore also represent a certain amount of time each. The amount of data buffering required goes from a few pixels, to several lines, or even to several whole frames, depending on the task performed by the piece. It might appear that reducing buffering as much as possible would achieve very low latency, but in reality we must often find a trade-off between the processing speed and the transmission speed, and factors like video quality, hardware load, and more. But in the end, latency can be expressed either in time units or in image units and we will use both interchangeably in this work.

We consider here a simple sender-receiver architecture. In all generality, the chain starts with a video source consisting of a camera capturing a certain number of images per second which then outputs a stream of uncompressed digital images at the corresponding frame rate. Capturing the video flow adds a constant delay of a few milliseconds due to the camera refresh process and circuitry [12]. The uncompressed video frames output by the video source obviously have the best quality possible, but it comes at the expense of a very high bitrate, unsuitable for transmission. Therefore, the frames are then compressed by an encoder before being prepared for transmission in what we can call the "*packetization*" block. There, the stream is segmented and put in the chosen streaming protocol and transfer protocol payload. It is then ready to be sent over the network link, which is represented by the "*network*" block. The network layer is the main source of uncertainty of the chain, corrupting the stream with unknown and always changing delay, packet loss and jitter. This is partly corrected at the receiver using a jitter buffer, before the depacketization and decoding can take place.

The decoder outputs a stream of raw video frames, identical or not to the one created by the video source, depending on the coding standard and potential noise added during the transmission. The last element called "*display*" represents all the various display processing needed depending on the display technology. It adds another constant latency of a few milliseconds to the end-to-end delay.

2.1.1 Coding

The encoder takes as input the stream of raw video frames captured by the video source and compresses it in order to have a stream with lower data rate, suitable to be sent over the network. Compression requires important computing resources. Higher compression gains usually mean a higher complexity algorithm and a higher load on the hardware used for the encoding. It is also a major source of latency and jitter in the system. Therefore, when choosing a compression mechanism, a trade-off between the compression ratio and the delay must be found based on the application. The encoder also incorporates a buffer to ensure that compressed frames are output at the same rate as they enter the encoder, which can add some small delay too.

We can discern different types of encoding algorithms based on the structure of the video stream data, which can be cut into Groups of Pictures (GOPs) consisting of several frames. Within a GOP, the frames can have temporal dependencies with each other, which results in three frame types: Intra (I), Predicted (P) and Bidirectional (B). An I frame is fully encoded alone, while P frames are encoded based on the previous ones, and B frames rely on previous and on future frames. I frames are therefore heavier than P frames, which are heavier than B frames. Depending on the GOP structure, we can have intra or inter-coding. Intra-coding only uses I frames, while inter-coding uses unidirectional or bidirectional prediction, thus being able to achieve higher compression gains. However, for a low-delay streaming application, we can see that using bidirectional prediction-based inter-coding adds a delay of at least a full frame, which is unacceptable. In such a case, only intra-coding or unidirectional inter-coding is recommended.

The second important parameter for our application is good image quality after the decoder. For this point, we can differentiate lossless and lossy codecs. With a lossless standard, the decoder is capable of reconstructing the exact same signal as the one encoded. We also use the term *full transparency*. On the other hand, some information is lost during lossy compression, so the decoder only outputs an approximation of the original signal. Lossy compression can however achieve *visually lossless* transparency, that is when the difference between the original and reconstructed signals isn't detectable to the human eye, for much higher compression ratios than lossless compression.

Popular compression standards include JPEG2000 [13], AVC/H.264 [14],

HEVC/H.265 [15], AV1 [16] and VP9 [17]. Much research has been dedicated to comparing the performances of video codecs [18] and their use in low delay applications [19]. On this topic, the most popular implementations of AVC and HEVC, x264 [20] and x265 [21], provide a "zero latency" option, which sets parameters of the encoder and decoder to minimize the coding delay. In [19], Minopoulos *et al.* analyze video codecs performances for real-time transmission and come to the conclusion that each new codec brings better compression gain but at the expense of increased complexity cost that makes them impractical for very low latency applications. They prompt future video compression algorithms to focus on low complexity in order to reduce the encoding and decoding time. This need for a lightweight codec has stimulated the JPEG Committee to develop and standardize JPEG XS, a brand new codec which is ideal to optimize the live video streaming for this work.

JPEG XS

JPEG XS was standardized by the JPEG Committee in 2019. It is a new lightweight encoding system that was designed to preserve the advantages of uncompressed video and to be used wherever uncompressed video is usually employed today [22]. Indeed, uncompressed video is used in many applications because of its low latency and great picture quality implications, but with always rising image resolutions and the arrival of technologies like High-Dynamic Range (HDR), the bandwidth and memory space required in raw video applications are too high to be used realistically.

In a study case like ours where low latency and picture quality are critical, the usage of JPEG XS is indeed very convenient as show its characteristics. It uses Discrete Wavelength Transform (DWT)-based intra-coding, which offers very low latency, from a few lines down to a fraction of a line [23]. It was also designed to have a low implementation complexity to reduce power consumption compared to other video codecs, as well as to be inter-operable, *i.e.* it can be supported by all kinds of different popular platforms (CPU, GPU, FPGA or ASIC). That way, a stream encoded in real-time on a given platform with a certain degree of parallelization can be decoded in real-time as well on any other platform. The codec also offers full transparency for compression ratios up to 10:1 and visually lossless quality for ratios up to 20:1 but can still obtain higher gains. Finally, JPEG XS is robust, meaning that it can be encoded and decoded up to ten times without any quality degradation [24].

All in all, JPEG XS allows us to already advance on our research for the right architecture thanks to its advantages with respect to our goal. Moreover, it allows for precise bitrate control to accurately match the available bandwidth, an important topic which we will develop in Section 2.1.3.

2.1.2 Packetization

Next, the compressed pictures need to be sent on to the network layer. For that purpose, the data stream is cut into blocks that are then encapsulated into streaming protocol and transport protocol packets. It is represented by the "*packetization*" block on Figure 2.1. To be more precise, this task involves segmenting the video stream into blocks of a certain size and adding streaming protocol metadata to it, which includes payload type identification, sequence numbers, timestamps, and more depending on the chosen protocol. The packet is then encapsulated with one or more transport protocols' headers to allow for the packet to be transmitted through the different network layers until it arrives at the receiver. These protocols are typically the Internet protocol (IP) and the User Datagram Protocol (UDP) or the Transmission Control Protocol (TCP).

As for any other element, this step adds some delay to the ensemble, which is a fixed packetization delay. This delay is controllable as it depends on the size of the payload. Indeed, the smaller the payload, the smaller the size of the buffer and the smaller the delay. However, a smaller packet size introduces an increased overhead to the system, as it has to perform the encapsulation task more often for a fixed data rate. So, choosing the size of the payload of the transport protocol has an influence on the performances of the system. In most protocols there is a maximum value that can be set, which is called the Maximum Transmission Unit (MTU) or the Maximum Segment Size (MSS) for TCP.

Of course, on the other side of the channel there is a depacketization step to remove the headers and keep the payload part only. It simply is the reverse of the packetization task and in consequence the latency introduced is in correlation with the packetization delay added at the sender [25].

Streaming protocol: RTP and RTCP

In most multimedia real-time communication applications, the protocol in which the data is encapsulated is the real-time transport protocol (RTP) [26]. It provides end-to-end network transportation and allows monitoring of the data delivery. It consists of two parts that carry out a task each: the Real-Time transport Protocol for the delivery of the data itself, and the Real-Time Control Protocol (RTCP) for monitoring the quality of service and to transport information about the ongoing session.

To begin exchanging data with RTP, endpoints must acquire their counterparts' pair of transport addresses, which consists of an IP address and a pair of adjacent UDP ports, one for RTP packets and the other for RTCP packets. This describes what is called an RTP session. In a multimedia exchange, it is recommended that each different data stream is transmitted through a different RTP session with

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronization source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.2: RTP header format

specific RTCP packets, so an endpoint can participate in multiple RTP sessions at the same time and differentiates them by reception of the different sessions on separate transport addresses.

RTP is a text-based protocol. An RTP packet consists of a header followed by the payload where the media data is placed. The header has the format shown on Figure 2.2 and can be modified with extensions. To quickly summarize the important values, the payload type (PT) field indicates what type of media is contained in the RTP packet and its encoding. The payload type can also be specified with signaling. Next, the sequence number increments by one for each packet formed, and so can be used for packet identification and reordering as well as for detecting packet loss at the receiver. The timestamp field for video is the capture time of the first frame and allows for timing reconstruction as the number increases monotonically. It also enables inter-stream synchronization by pairing the RTP timestamp with a Network Time Protocol (NTP) timestamp present in the RTCP Sender Reports (SRs). Doing this establishes a mapping to a reference clock shared by all the sessions which allows streams from separate RTP sessions to be synchronized, for example audio and video for lip syncing. Finally, the synchronization source (SSRC) identifies with a random 32-bit number an RTP stream source within an RTP session.

The RTCP packets provide periodic feedback on the quality of the data transmission and can be separated into different roles. There are Sender Reports (SRs) providing transmission and reception statistics from participants that are active senders, like the number of bytes sent and the timestamps for synchronization. Then, Receiver Reports (RRs) give reception statistics about the packet loss, the inter-arrival jitter, timestamps, the Last Sender Report arrival time (LSR), the Delay since the Last Sender Report arrived (DLSR), and more. The full format of an RTCP SR packet can be seen on Figure 2.3. An RTCP RR packet follows the same structure, but without the sender information part. This feedback can be

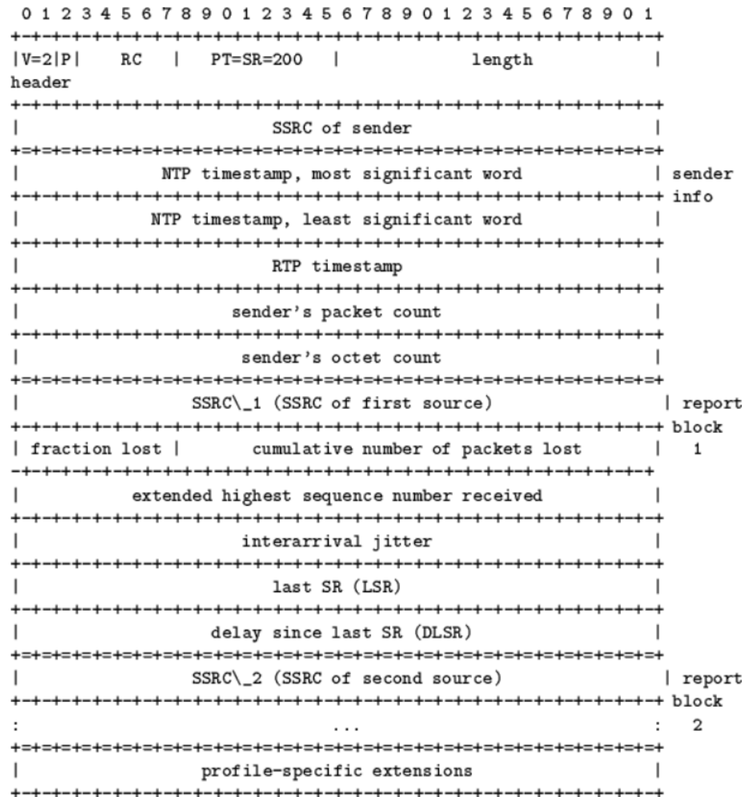


Figure 2.3: RTCP packet format

used in many applications, including congestion control and packet loss correction or retransmission. The LSR and DLSR notably allow the sender to compute the network Round-Trip Time (RTT). A server typically allocates 5% of the bandwidth to RTCP traffic, but this can vary depending on the payload type.

2.1.3 Network layer

Transmission over the network comes with all its ways of disrupting the delivery. It has limited bandwidth and introduces latency, packet loss, jitter, all of which can change at any time. The latency in this layer can be described as coming from two factors [12][25]. First, simply sending the packet at a certain target transmission rate takes some time, depending on the bitrate and on the size of the packet. Second, the propagation time it takes for packets to reach the receiver. It depends on the geographical distance the signal has to travel to arrive to its next stop as well as on the distance in the network, *i.e.* the number of hops and servers the packets must go through before its final destination, all of which potentially add some transmission, processing and queueing delays to the end-to-end delay.

Regarding transport protocols in the network layer, RTP and RTCP both run on top of UDP (TCP is possible but is not prevalent), which brings some advantages and disadvantages. In contrast to TCP (for Transmission Control Protocol), UDP (User Datagram Protocol) is an unreliable service which doesn't guarantee the orderly arrival of the packets at the recipients. Moreover, UDP is a push-based protocol, which poses some security problems. So, some networks have firewalls and Network Address Translators (NATs [27]) blocking the arrival of UDP packets. But protocols exist to bypass these firewalls. The Interactive Connectivity Establishment (ICE [28]) has for goal to find the way for two peers to talk to each other as directly as possible. Then the Session Traversal Utilities for NAT (STUN [29]) protocol can be used to try to connect the endpoints, as it allows to find out the original address and ports of NAT clients. Finally, we can potentially use the Traversal Using Relays around NAT (TURN [30]) protocol to go through a relay router in order to bypass the firewalls if it is needed. On the other hand, UDP being a push-based protocol and not acknowledging the arrival of every packet implies better timeliness and less jitter and therefore explains why real-time communication protocols prefer UDP over TCP.

Congestion control and stream repair algorithms

In order to avoid bottlenecks in link channels during transmission, addressing congestion control is critical to any network application like ours. Unlike TCP which is a reliable protocol and therefore takes care of congestion control, UDP itself does not avoid congestion. Congestion control measures must be implemented at the application level or in the network. And so there exists many mechanisms and protocols to produce an estimation of the available transfer bandwidth, which is then used to alter the behavior of the flow of the data by configuring an appropriate target sending bitrate to avoid overflowing the channel.

State-of-the-art algorithms include the Google congestion control (GCC [31]), which has been implemented in mainstream browsers like Chrome and Firefox. Since our application isn't browser based, an estimation of the available bandwidth can simply be found using RTCP packets. An example among RTCP messages is the Receiver-Estimated Maximum Bitrate (REMB [32]) extension to feedback messages. These signal to the endpoints the estimated total available bandwidth for a session. With REMB feedback messages, the receiver communicates to the sender of an RTP video stream an estimation of the available capacity based on variations in frame inter-arrival times. An alternative is the Transport-wide Congestion Control (TCC [33]) RTP header extension and RTCP messages, that also enable congestion control to be performed at the transport level on the sender's side.

Once an estimation of the bandwidth is known, the information can be commu-

nicated back to the encoder to adapt the encoding bitrate to match the available bandwidth of the channel. Other parameters can be changed to the video stream outside of the encoder in order to change the sending bitrate, like the frame rate and the resolution of the video.

If no extension like REMB or TCC can be used, it is still possible to adapt to the network by using an adaptive bitrate algorithm. For that, information contained within RTCP messages can be used to increase or decrease the sending rate based on the throughput of the channel. More exactly, RTCP messages contain a packet count of RTP packets sent and received. They can therefore be used to calculate the rate of successful packet delivery over the channel. If that rate is deemed high enough, we can say that more bandwidth is available for the transmission, and it can then be communicated to the sender to transmit the stream at a higher bitrate that before. Similarly, when too many RTP packets are lost, it is interpreted as the sending bitrate having surpassed the available bandwidth, and the sending peer can then be told to decrease that bitrate in order to reduce the packet loss. There exists different ways to increase and decrease the transmission bitrate (linearly, quadratically, etc.), each forming a different algorithm. Adaptive bitrate algorithms can also be based on different parameters than just the throughput of the channel, like the CPU usage of the peers, the jitter buffer level...

Even though congestion control is performed, packets still can be lost during transmission, so stream repair algorithms must be employed [34]. Packet loss can be detected using the sequence numbers and timestamps in the RTP header. A possibility is to re-transmit the packet, using Negative Acknowledgement (NACK [35]) messages to signal the sender that a packet has not been received. This implies keeping the stream stored in a cache memory for a few Round-Trip Times (RTTs) in case re-transmission is needed. Alternatively, for video, the decoder can judge that re-transmission is unneeded and apply error concealment techniques to the stream. In addition to that, techniques like Forward Error Correction (FEC [36]) can be used. Here, the idea is to add redundancy to the video stream, so that the receiver is able to detect and correct errors without re-transmission, but at the cost of a lower bandwidth for the sent data since some of it is redundant.

2.1.4 Jitter buffer

Finally, there is a jitter buffer placed for each stream at the receiver's side to compensate for the jitter and packet loss that appears randomly during the transmission. This element represents one of the most important sources of delay in the chain. Indeed, the buffer must be large enough to allow for re-transmission in order to reduce packet loss. As said earlier, a large buffer will increase substantially the end-to-end latency since packets potentially have to wait for several RTTs in

it. Therefore, state-of-the-art applications use adaptive jitter buffers that keep the size and the load of the buffer as small as possible in order to minimize the delay, while allowing it to grow too based on the real-time delay variation [37].

2.2 Video network topologies

The previous considerations allow us to already envision the different possibilities available to improve the timeliness and responsiveness to network conditions for our application. However, the environment was a simple communication from a sender to a receiver, and we still have to address some of the challenges presented in the introduction, like scalability and long-distance communication. Such matters are better answered by reviewing the possible architectures to give to our system, which we will do in this section.

The RFC7667 publication details all the topologies used in environments based on RTP [38], but anyhow the two main choices at hand when we consider the communication is either a Peer-to-Peer architecture or a Client-Server architecture. The latter can further be divided by looking at the number of streams exchanged with the central server and the processing done by the said server. A first possibility is for the server to receive all the video streams from the participants and mix them together to create a single new video stream that is then sent back to every client. We call this a Multipoint Control Unit (MCU). The second possibility is rather to have the central server simply copy and re-route the received streams to the participants. This is called a Selective Forwarding Unit (SFU). In the next two sections, 2.2.1 and 2.2.2, we expand on these topologies and weigh the pros and cons for each of them.

2.2.1 Peer-to-Peer system

In a peer-to-peer (P2P) architecture, the media is directly exchanged between the participants. Each peer encodes separately the video and sends it through a separate RTP session to every other connected peer. The resulting architecture is also called a *full-mesh topology*. An example for five peers is shown on Figure 2.4.

The clear advantage of this architecture is the absence of any kind of media relay server. In consequence, the latency is minimized as there is no queueing and processing delay at the server. Moreover, the architecture presents a cheap and easy to deploy solution since no central infrastructure needs to be built. However, each peer has to encode the video to stream multiple times, which is very heavy on computing resources for the endpoints. And as the number of peers increase, so does the processing at the user's side. Moreover, sending streams to multiple recipients (if there are n peers, each peer must send and receive $n-1$

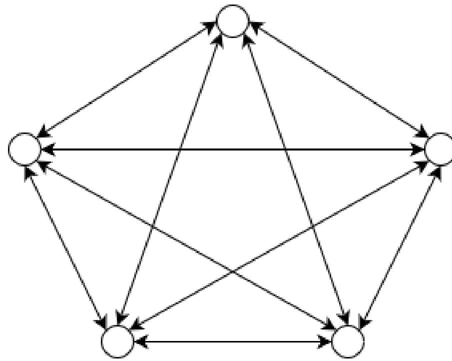


Figure 2.4: Full-mesh topology

streams) takes a lot of bandwidth which, besides, is usually limited for typical endpoint users, especially uplink bandwidth. In short, P2P systems do not have a good scalability. Finally, the absence of a central unit makes it difficult to monitor the exchanges between the peers and thus to implement good congestion control. Moreover, this also makes our "collecting statistics" requirement from the introduction difficult to implement.

In the end, these constraints have made the deployment of P2P solutions difficult in the video streaming industry, even though P2P is still a widely used technology in other domains, for example file sharing technologies like torrenting.

2.2.2 Client-Server system

The evident counterpart to P2P is the use of an intermediate media server, producing what is called a *star topology*. This approach reduces the client upstream flow and therefore improves the scalability of the architecture. The client-server system can be implemented in multiple forms, depending on the number of streams and on the media processing performed at the server's side. The two most popular architectures in videoconferencing technologies are presented hereunder.

Multipoint Control Unit

A Multipoint Control Unit (MCU) collects all the RTP streams published by the clients, decodes and mixes them to create a new image. The newly formed video is then re-encoded up to the quality that meets the bandwidth requirements for every connection, forming a single common stream that is then sent back to the clients. An example with four clients is shown in Figure 2.5.

Directly, we can compare to P2P and remark that the participants only need to send and receive one single stream, thus effectively reducing the bandwidth used. They also only have to encode and decode one video stream, so the processing

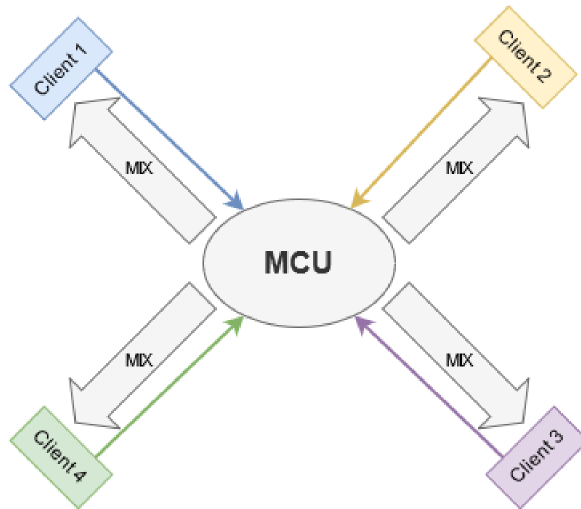


Figure 2.5: Multipoint Control Unit (MCU) topology

demands on the clients are also effectively reduced. Congestion control is also much easier, as the MCU can configure its encoding bitrate to adapt to each peer's network condition. On the other hand, the computational cost that was at the peer's side in P2P has now moved to the server: the MCU has to mix and transcode multiple streams which is very computationally intensive. Compared to P2P, the end-to-end delay is obviously greatly increased due to all the processing at the MCU, which notably requires having a jitter buffer for decoding, a great source of latency.

In addition, when we compare MCUs to other client-server architectures, flexibility for the clients as well as for the server is very low for this system. Indeed, the endpoints receiving only one pre-composed video stream means, for example, that they cannot choose to watch the video of only one other participant, or mute another one, etc. They receive the composed video and cannot change its parameters. Finally, the video quality of the result stream received by every one might be damaged due to the mixing and transcoding. All in all, an MCU is a solution which is light on the bandwidth, but heavy on the server load.

Selective Forwarding Unit

The alternative to MCUs, instead of mixing the videos together, is simply to re-route the stream from one client to every other one via a Selective Forwarding Unit (SFU), as shown on Figure 2.6. Here, the clients only send their streams once, to the server which then forwards them to one or more receivers without re-encoding them, while analyzing the media control packets and using that information to manage the flow of the streams.

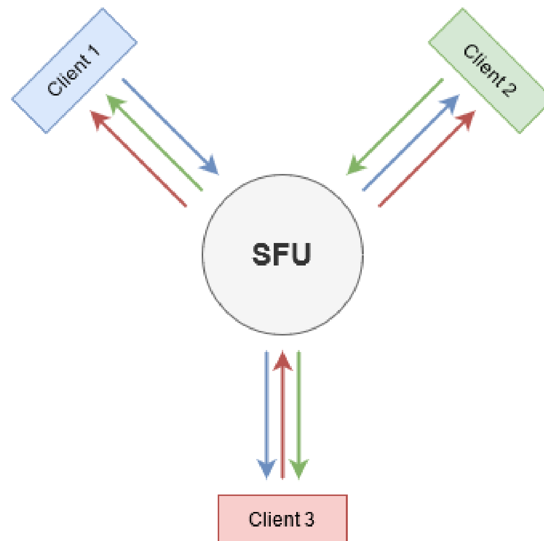


Figure 2.6: Selective Forwarding Unit (SFU) topology

Compared to an MCU, an SFU does not need to transcode the received videos, which reduces end-to-end delay and processing at the server's side, as it only "copies" the received streams and redistributes them. This is also an architecture that provides more flexibility. Indeed, endpoints receive separate RTP streams for each participant and can therefore personalize what they want to do with them. However, this also means that the computational cost is increased for the clients in order to decode the potentially numerous received videos.

Another point is that the downlink bandwidth requirement for SFU clients is increased in order to receive every stream. In the same line, congestion control is also more complex, as the SFU does not control the encoding bitrate of the videos it receives. So, congestion control solutions must be found on the application level. When there is a high number of participants, a first solution in order to deal with the high bandwidth demands and with changing network conditions is to stop the re-direction of some selected streams at the server. Only a subset of the RTP streams received are then forwarded which reduces the sending bitrate, and the server can possibly enable the stopped streams later on when network conditions allow it. In [39], Grozev *et al.* propose a solution for a videoconference setting, where they develop a dominant speaker identification algorithm whose streams will be transmitted to all other participants, while the others are not.

Several other solutions have been proposed in the literature. One of them is simulcast, where each participant encodes the video in different qualities and then uploads them all to the SFU in separate RTP streams. The SFU can then select which stream to forward to the other clients based on the network quality

Figure 2.7: Principle of simulcast. [1]

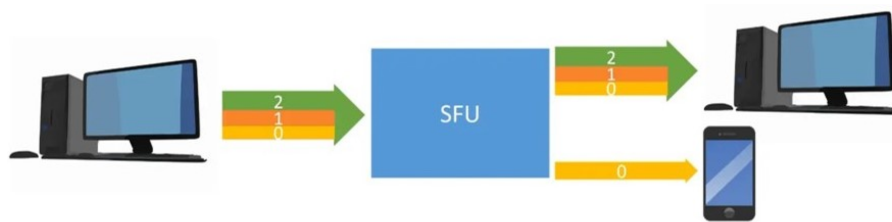


Figure 2.8: Principle of Scalable Video Coding. [2]

or on what the client decides. In [40], the sender encodes its video in up to three different but fixed qualities. In [41], the sending peers also use a limited number of encoders, but the SFU also periodically recomputes the set of encoding bitrates and communicates it back to the senders, who can then adapt the encoding parameters.

Another solution resides in Scalable Video Coding (SVC) standards. Here, the video source is coded in multiple layers, each with a higher resolution and quality level. These layers are transmitted as distinct bitstream components to the central server. Each additional layer adds a better quality to the video. The SFU can then select the number of layers to send to each receiving peer in order to adapt to the network rate [42]. For now, however, SVC is only available to a certain number of video codecs [43][44].

2.3 Content Delivery Networks

Using only a single server acting as an MCU or SFU still has a limited scalability. Indeed, the server itself also has limited computing resources and bandwidth. So, if multiple exchange sessions with many participants are being held at the same time, the central server becomes a bottleneck on bandwidth and might get overloaded. Moreover, a single server will have difficulties meeting the long distance requirement set in the introduction. Indeed, clients that are distant will experience increased latency and worsened connection quality which will result in degraded picture quality at the receiver, due to packet loss or if the server adapts the video encoding to the network. Finally, a unique server suffers from a Single Point of Failure (SPOF) configuration.

A first alternative is the use of a data center, a large set of interconnected servers housed in a dedicated building. This allows the system to have a better scalability by using load balancing techniques to spread the amount of load and processing needed to multiple servers in the data center. However, it still doesn't answer the long distance and low response time requirements because it still is a centralized architecture where one data center covers a potentially too wide geographical area to serve the clients correctly [45].

Another solution is to have multiple servers physically distributed around the world, so that the users can connect to the server that is closest to him. Such networks of dedicated servers deployed at different locations on the Internet in order to distribute some media are called Content Delivery Networks (CDNs) [46]. The idea is to replicate the media that users want to stream to a set of edge servers and then redirect the clients to a local edge server based on a redirection algorithm. This way, they can access the content with lower latency and higher data rates than if it was from a single, distant server.

Thus, the goal of a CDN is to offload the traffic originated to the content provider's infrastructure, and to make the content widely available to end-users with high performance by its widely deployed servers.

2.3.1 Mode of operation

The correct functioning of a CDN for live streaming relies on three main mechanisms. First, a video source provides some media content to the CDN, which then has to push the video streams to other edge servers of its network, so that clients from anywhere on the Internet can access the video rapidly. However, CDNs do not replicate a video to *every* edge server possible because of the memory and transfer bandwidth it would take. So an algorithm of *content placement* must be followed, that answers to the question of where and how to place the copies of the stream among the servers. For live streaming, the replicated content should be pushed to chosen edge servers following some load balancing decision, in order not to saturate any server already under heavy burden. Another question raised is how to place the edge servers on the Internet. Intuitively, there should be many edge servers carefully located to be as close as possible to a large number of clients. So, in the end we have servers widely deployed around the world with content carefully placed in some of them in order to be easily accessed by the masses everywhere.

Secondly, the CDN must find a path from the point where the content is entered to all the edge servers at the other end. For that, CDNs servers can be grouped into three categories [47]. First, there is the entrypoint, which directly receives the stream from the source. Then, the video must be moved to the subset of edge servers chosen to hold a replicate of the media. The entrypoint himself

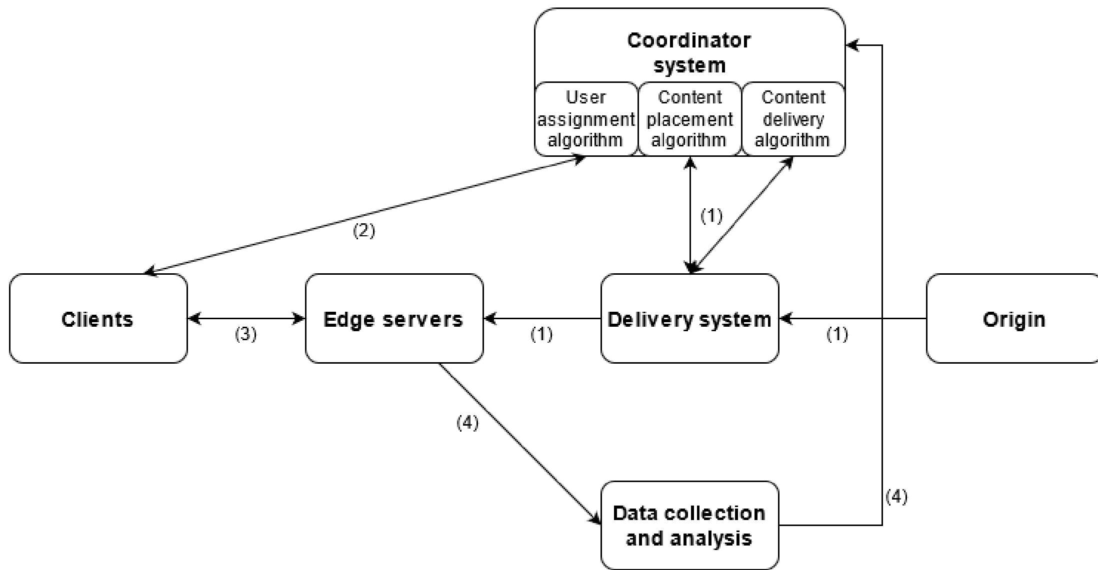


Figure 2.9: Main system components of a Content Delivery Network

is not able to do to it on its own, so an intermediate layer of servers called the reflectors is used to forward and multiply the received data to the edge servers, which constitute the last type, responsible for delivering the video to the endpoints. The second algorithm is thus *content delivery*, *i.e.* finding a way to distribute the live video from the entrypoint to the edge server in an optimized way. To address this matter, CDNs construct and maintain distribution trees or overlay networks through the CDN to constantly push the live stream from the entrypoint to the clients. Much research has been made on how to build the best overlay distribution tree based on resources used, latency, and more. An example of such an algorithm can be found in [48].

Third, the CDN needs a *user redirection* algorithm, to redirect the clients who want to receive the live content to an edge server holding this content based on some selection strategy. The selection of the most appropriate edge server is the center of the algorithm. It depends on the network proximity between the server and the receiver and on load balancing between edge servers. After that, the question is how to redirect the user to the chosen server. The main technique is DNS redirection, used by Akamai, the worldwide leader in the CDN industry. With DNS redirection, the IP addresses of edge servers holding a copy of the stream is returned by DNS servers of the CDN. An alternative to it is anycasting, notably used by CloudFlare. With anycasting, the rerouting is done by the Border Gateway Protocol (BGP) based on its own notion of best path [49].

To summarize, let us look at the essential components of a CDN as shown on Figure 2.9 [46][47]. The numbers seen on the figure describe the following

operations:

(1) The origin server is an entrypoint of some media content in the CDN. In order for this content to be distributed at the edge servers, the origin server gives it to the delivery system composed of the various reflectors, which replicate the content and pass it down to the chosen edge servers. Reflectors typically store the video in a cache memory too, in case it is needed in another edge server. The delivery is done following a content placement algorithm and a content delivery algorithm as described above, and that is all managed by a central coordinator system.

(2) Clients wanting to receive the content are redirected by the coordinator system from the origin server to an edge server that is closer to them following the user redirection algorithm. The assignment to the server is made based on collected data about the network and server conditions.

(3) The edge servers deliver the content to the clients.

(4) At the same time, data is collected from the edge servers and processed by another system that uses it for analytics, monitoring, reporting and billing. Some of this information is transferred to the coordinator system to help for future user assignments and content placement.

To conclude, a CDN offers wide distribution of content, therefore improving the scalability of any distribution application. It also provides distributed hosting capacity as well as redundancy with its caching capacities. However, it is very costly to implement and to manage, as it requires installing and running a high number of servers around the world.

2.3.2 Hybrid architectures

In order to get best of the solutions for large-scale video distribution, research has been made to develop hybrid architectures, using both P2P and Client-Server architectures. A first implementation would be switching from P2P to a server-based topology when the number of clients gets too high for the full-mesh topology to be correctly functioning [50]. Alternative research goes further by proposing hybrid P2P-CDN architectures, where the CDN provides for scalability while P2P between nearby clients ensures shorter delay times [51].

Chapter 3

Model description

Based on the concepts presented in the previous chapter, we are now more able to wisely pick the right system components in order to fulfill our set of requirements. As a reminder, we want to investigate how to build a live streaming system between cinema rooms exchanging several streams including a video stream and a sensor-related data stream for gaming. The emphasis on the latter should be very low delay, while the video flow must keep a high quality in order to be projected in a theater. Moreover, the global system has to be low-latency, has to support long-distance communication between a potentially high number of endpoints and lastly has to be able to adapt to the network conditions and to monitor the exchange in real-time.

In this chapter, we present the theoretical model that we propose to answer the demands just exposed. We start by choosing the topology best suited to our needs in Section 3.1. Then, in Section 3.2, we describe the transmission chain that we are going to implement in this work. Then, we will address the question of signaling with Section 3.3. Finally, we talk about how the system can be extended using the concept of a CDN in Section 3.4.

3.1 System architecture

We must select the architecture we want to give to our system. As we have seen in Chapter 2, a full-mesh topology suffers from requiring large amounts of uplink bandwidth and processing power at each endpoint. Moreover, statistics are difficult to collect due to the lack of a centralized structure. Notably due to this latter point, the Peer-to-Peer architecture must be discarded in favor of a Client-Server architecture. More exactly, a Selective Forwarding Unit-like architecture (SFU) is the most adapted in our case. Indeed, an SFU allows to

monitor the state of the transmission and to collect data. We can even possibly choose to record and store an exchange session if we want. Moreover, mixing the video streams as is done in an MCU isn't quite adapted to a gaming situation. The SFU provides much more flexibility to the participants to choose what they want to do with each received stream. For example, this allows flexibility for the design of a user interface fit for gaming. Server-centered architectures can be limited in scale due to the limited bandwidth and processing that can be done with a single server. However, we can scale the system to wider audiences by incorporating more than one SFU in the system, and thus cover a wider area too. The main disadvantage of an SFU is that managing the use of bandwidth at the server's side can become quite complex, but we can still find strategies to adapt the data processing to the different users. In the end, an SFU presents advantages that P2P and MCU do not have, and therefore we make the choice to study this model in this thesis.

MCUs and SFUs have already been implemented in commercial applications, mainly in videoconferencing environments using the WebRTC technology. For example, Jitsi has developed the Jitsi Videobridge as its own version of an SFU [52] and other companies like Kurento [53] and Janus [54] have been known for implementing such platforms as well.

In our study case, the goal is to transmit at least two RTP streams belonging to different mediums (a video stream and a gaming data stream minimum). We will therefore need to open separate RTP sessions for each medium [26], so there is an independent set of RTP sessions between every endpoint and the server. The SFU can then forward all or some of the streams that it receives to all or some of the clients connected.

With each RTP session is associated a two-way flow of RTCP control packets that also needs to be handled by the SFU. Among those, we first have the RTCP Sending Reports (SRs) that allow to identify the senders with an SSRC and to get information about the RTP streams that the SFU receives, like the number of bytes and packets sent, the NTP timestamp for synchronization, and more. Then, the RTCP Receiver Reports (RRs) can be processed to collect information about the exchange, like the number of packets received. The information that can be found in RTCP packets was shown on Figure 2.3. Both these reports can also include specific extensions like Negative Acknowledgment (NACK) and Receiver Estimated Maximum Bitrate (REMB). In the first case, if the stream is still available in some cache memory in the server, the missing packet reported by the receiver can be re-transmitted. In the latter case, the information about the estimated bandwidth can be extracted from the REMB packet and be used for a number of applications, like adapting the sending bitrate to the network condition.

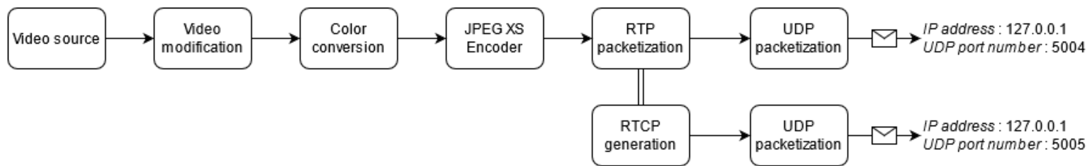


Figure 3.1: Sending client's video transmission chain

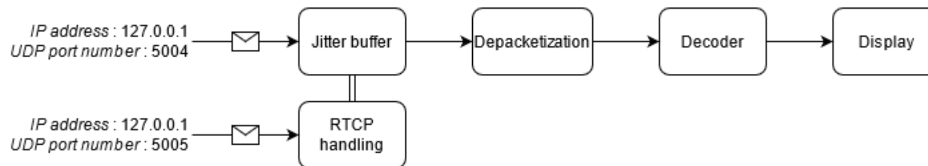


Figure 3.2: Receiving client's video transmission chain

3.2 Transmission chain

Now that a centralized architecture has been chosen, let us look at how the system components are organized inside the different entities. First of all, let us consider the manipulation and transmission of the captured video. The transmission chain for the clients essentially follows the same patterns as was shown on Figure 2.1. We separate the sender's and receiver's sides and reproduce the chains on Figure 3.1 and 3.2 for the RTP video stream.

On the sending peer's side, we add two optional but potentially useful elements compared to the basic chain of Figure 2.1. First, there is the "*video modification*" block to represent the potential transformations given to the raw video for various reasons. This notably includes choosing a different frame rate as well as rescaling to a different resolution, both operations having a significant impact on the bitrate of the video stream. Then, we add a "*color conversion*" block which is responsible for adapting the frame format if it is needed in the case where the encoder doesn't accept the format captured by the video source. A typical example is video captured with an RGB format that needs to be converted to YUV format in order to use chroma subsampling at the encoder. We can also see on Figure 3.1 the RTP stream being sent to an even port number while the associated RTCP stream is sent to the adjacent odd-numbered UDP port.

Concerning the receiver's side, color conversion might need to be performed too, depending on the display device. Otherwise the chain of operations remains the same as described in Chapter 2. We have to mention, however, that the chains of Figure 3.1 and 3.2 are only intended for sending and receiving one RTP stream, yet we know that the theaters in our system will have to emit at least two RTP streams and to receive multiple video streams from all the other participants. Moreover, the clients send and receive this data flow at the same time. So, in the

end, we know that there must be several of those chains working in parallel at every client's side.

The interesting part now is how to design the chain inside the SFU. This will depend on how we want to implement the congestion control for our system, and we expand on this subject in the next section.

3.2.1 Adaptive bitrate coding for video

Implementing congestion control is the main challenge when designing a Selective Forwarding Unit. Indeed, the ways to adapt the bitrate of the data flow are limited. The video parameters that impact it directly are the frame rate, the resolution and the encoding bitrate.

If the SFU only forwards the streams it collects, the only way to change the bandwidth taken by the transmission is at the sender's side. First by modifying the frame rate or the resolution of the video in the "*video modification*" block, then by encoding at the right bitrate at the encoder. However, setting these parameters at the client's side brings a problem. Indeed, this means that in order for every participant to receive the video from every other peer, we must estimate the bandwidths of all outgoing streams from the SFU, keep the least capable path and communicate it back to every emitting client. This results in peers having a perfectly good connection with the SFU that suffer from videos encoded in worse quality when they have enough capacity for better. There is the possibility to transfer only a subset of the received streams [39], but badly-linked users will still handicap the image quality of others.

Therefore, congestion control in an SFU should be user-adapted. That is why techniques like simulcast and scalable video coding are very popular and have been researched extensively for this situation. However, these study cases make use of more computation-demanding and lossy codecs like AVC/H.264 or VP9. That is where the use of the lightweight and visually lossless codec JPEG XS becomes very convenient. We can therefore use its robustness, low complexity and very low latency promises to implement a rather simple model containing encoders and decoders in the SFU.

In view of the previous arguments, we can recognize that the SFU's internal structure will be composed of components of both Figure 3.1 and 3.2, as the server receives, copies, and emits RTP streams. A first solution could be for the client to transmit its video stream uncompressed and have the server encode it multiple times in order to comply with each bandwidth requirement. But we know that the bitrate flow of raw video is too high for transmission, so a JPEG XS encoder is still needed at the sending client's side. This means that in order to modify the bitrate of the media flow at the SFU, transcoding needs to take place. In the end, the model that we propose has the emitting peer encoding its video stream with

a high image quality and sending it to the SFU. The SFU decodes the stream then duplicates it and transcodes each copy to a bitrate adapted to the copy's intended network transmission channel.

The adaptive encoding can be done either based on founding the bitrate capacity of the link using REMB packets exchanged with the receiving clients or by following an adaptive bitrate algorithm.

The resulting transmission chain for re-routing an incoming video stream to two clients is shown on Figure 3.3. In short, the SFU processes an incoming video stream in the following way:

1. The incoming RTP and RTCP packets are received by the SFU. This one can then read fields in the packet headers in order to recognize the type of packet and to count the number of packets arrived. Statistics like bitrate, packet rate, packet loss and jitter can then be calculated.

2. Next, the video stream is reconstructed in the jitter buffer. This is done with the help of the RTCP Sending Reports from the sending peer. At the same time, RTCP Receiver Reports are generated and are sent back, including potential NACK retransmission requests for stream repair. There really is a two-way exchange of RTCP reports and feedback packets between the SFU and every client.

3. The video data is extracted from the payload and decoded.

4. The stream is duplicated in sufficient number by copying it in new buffers.

5. Each video can then be manipulated and re-encoded by a JPEG XS encoder to satisfy the bandwidth constraints of every receiver. The encoding bitrate isn't the only parameter to affect the sending bitrate. That is why we leave the "*video modification*" block in Figure 3.3, to potentially modify the frame rate or frame resolution of the video. Indeed, if the stream sent to the SFU is a 4K UHD (3840×2160) 60 fps video for example, dropping the frame rate from 60 to 30 fps or changing the resolution to Full HD (1920×1080) decreases significantly the bandwidth taken by the stream and the quality is still very acceptable to be projected in a cinema.

6. The video is encapsulated back into transport protocol packets in order to leave the SFU. The packets are stored for a short time in a cache memory in case they need to be re-transmitted later.

7. The RTP packets are counted and sent. Again, RTCP packets of all sorts are exchanged along the media transmission. From these RTCP packets, either REMB or adaptive bitrate algorithms can be implemented, which

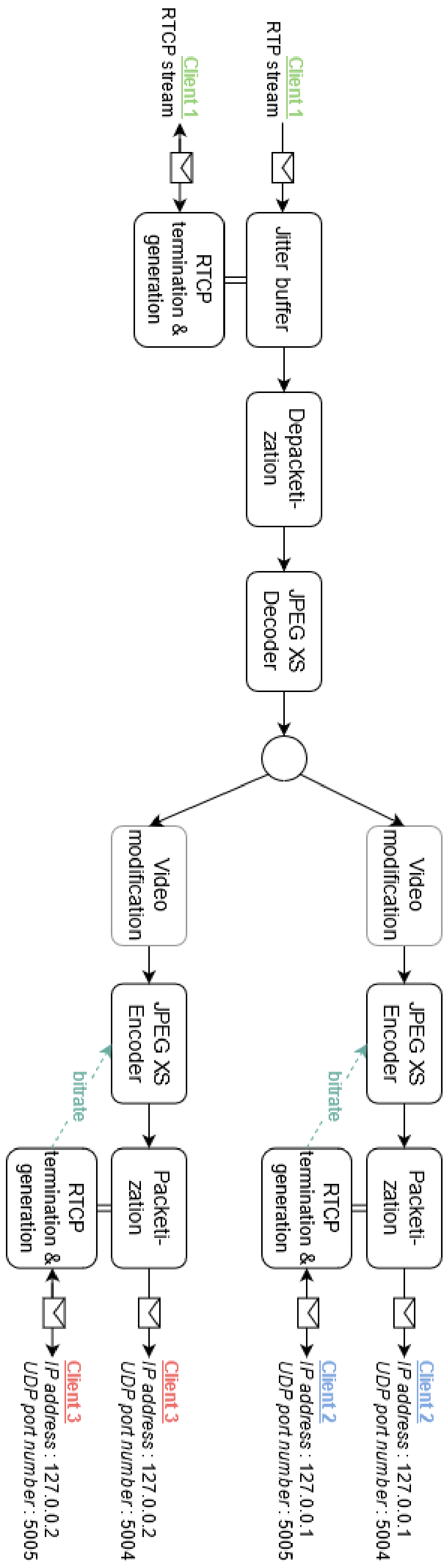


Figure 3.3: Transmission chain for the video stream in the SFU, communication between 3 clients

can be used to adapt the re-encoding bitrate of the data flow. Naturally, the re-encoding bitrate should be maximized with respect to the allocated bandwidth, in order to keep the video frames at their highest quality.

To sum up, we obtain a structure capable of re-directing an incoming video flow to multiple clients and of optimizing the frame quality to the network condition for every client.

3.2.2 Other data streams

Looking at the internal structure of the SFU for video streams, we can already tell that it is heavily resource-demanding, with multiple decoding and encoding operations taking place in the server. This reduces the scalability of the SFU, as the server might quickly overload. Moreover, the transcoding requires the presence of a jitter buffer, adding important delay to the transmission. Against that, we count on the low-complexity and low-latency functions of the JPEG XS standard.

However, when we consider other data streams that are not video-related, the chain of operations greatly simplifies itself. We recall that at least a motion sensor-captured data stream flows in parallel to the video stream and must also be transmitted from each peer to every other. We will refer to this stream as "the gaming stream" in the following paragraphs.

Naturally, there is no JPEG XS compression involved with this stream. Whatever the compression mechanism used for this data stream, if any, it will use very little bandwidth compared to video. In fact, even compressed, video takes most of the bandwidth, using at least three times as much data than an audio stream in videoconference applications [55]. Moreover, we can argue that the safe arrival of the gaming stream is of higher priority than that of high-quality video. Indeed, this data flow keeps the game running while the video stream doesn't. If the network quality does drop below a point where there is no possibility to transmit the gaming stream, the game simply cannot be played anymore. Therefore, the video stream should be the only one to adapt to bandwidth degradation. In the worst-case scenario, the spectators will not see the opposing cinema rooms, but will still be able to play against them. Besides, we already proposed a solution to adapt the video stream to the network quality. In consequence, there is no need to implement congestion control for the gaming stream as there is already for the video stream.

In addition, the focus on this data flow was ultra-low latency. Avoiding all transcoding and other modification of media at the server means that we can discard the jitter buffer, encoding-decoding and depacketization steps of Figure 3.3. In the end, there is little processing done by the SFU on the gaming stream. First, there is the duplication of the RTP stream and the changes in RTP header

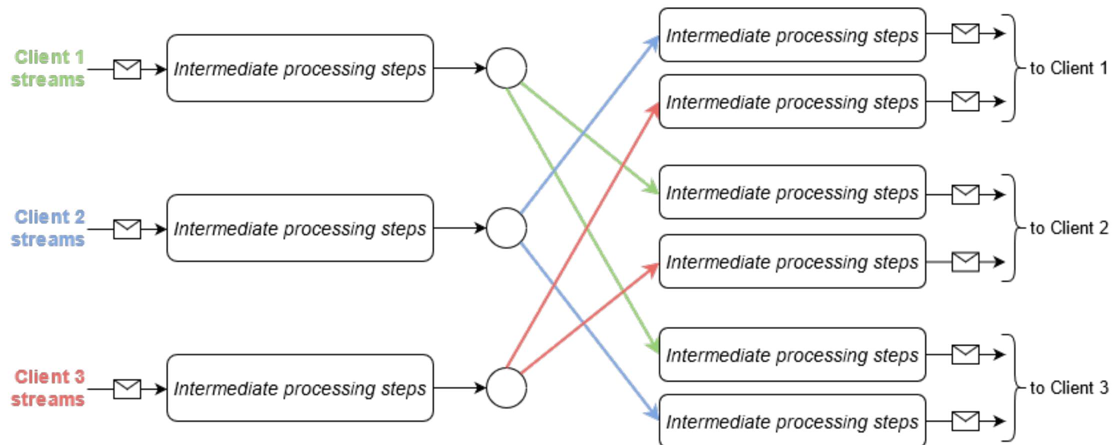


Figure 3.4: Full arrangement of the transmission chains in the SFU for a session between three theaters

fields that go with it; then there is computation of statistics like packet loss and jitter; and finally, the generation and termination of RTCP packets, notably for retransmission requests if needed. The latency is greatly reduced compared to the structure for the video stream. At the end of the day, for the gaming stream we genuinely have a theoretical SFU that only forwards its incoming streams to clients. Therefore it recovers its very low-latency and good-scalability properties.

We show on Figure 3.4 a simplified view of how the transmission chains are arranged in the SFU for RTP communication between 3 users. Each of the pipelines represented consists of a video stream and at least another data stream in parallel. The intermediate processing steps are as shown on Figure 3.3 for the video stream, while they are almost nonexistent for sensor-captured data flow as explained in the previous paragraph. More RTP streams can be added in parallel to these, for example to add an audio signal to the communication. This is the model we will implement with Gstreamer in Chapter 4.

3.3 Signaling

The pipelines of Figure 3.4 allow for communication between three clients. How can more clients be added from the system? Intuitively, we simply need to build more pipelines inside the SFU, or remove some when theaters leave the gaming session. In order to achieve this, we need a mechanism that will add and remove clients to the session and that will therefore manage the media transmission going in and out of the SFU. This is the role of the signaling system. In this section we start by giving a background on signaling principles and protocols and then we

will see how we use it to manage the communication in an SFU.

3.3.1 Background

When two or more endpoints want to set up a connection between them, they first need a mechanism to exchange information about them and the streaming session they are about to start. This mechanism is called "signaling" and goes through a server called the "signaling server".

Before any media exchange can be made through the SFU, the participating peers use the signaling server to negotiate the parameters of the session. They exchange their IP addresses and available UDP ports, as well as information about the session settings (the transport protocol, the encryption key, etc.) and about the media they are about to send or receive (for example, the codec and the resolution for video). This information can be communicated through messages described with the Session Description Protocol (SDP [56]) and transported with the help of a signaling protocol. Most popular among those are the Session Initiation Protocol (SIP [57]) and the Extensible Messaging and Presence Protocol (XMPP [58]). With these messages, the signaling server is able to allocate connection channels between the clients and the server, potentially using the ICE protocol, and therefore to start the transmission. After that, it keeps exchanging signaling messages periodically with everyone in order to be updated about the state of the streaming session. Finally, it is also in charge of ending all communication once the session is finished [41].

To sum up, the signaling component of the system is responsible for creating, managing and terminating the live streaming session by connecting the different entities to each other and allowing them to exchange their media flows.

3.3.2 Dynamic pipeline building

In this piece of work, we therefore also need to include a signaling server to add and remove clients to the gaming session. We can utilize a simple WebSocket-based protocol to exchange messages between the clients and the signaling server. Caution, no media stream passes through the entity, it is only used to set up the video exchange between the clients and the SFU. In this section, we show the train of operation of adding a new participant to the streaming session and how it is handled by the signaling server and the media server.

This simple process can be summed up with Figure 3.5 [59]. Three entities are represented on this figure: a client, the signaling server and the SFU. The client and signaling server first connect to begin the session establishing procedure by using a message sent via the signaling protocol. The signaling server processes it and communicates to the SFU the coordinates of the client. The SFU can then

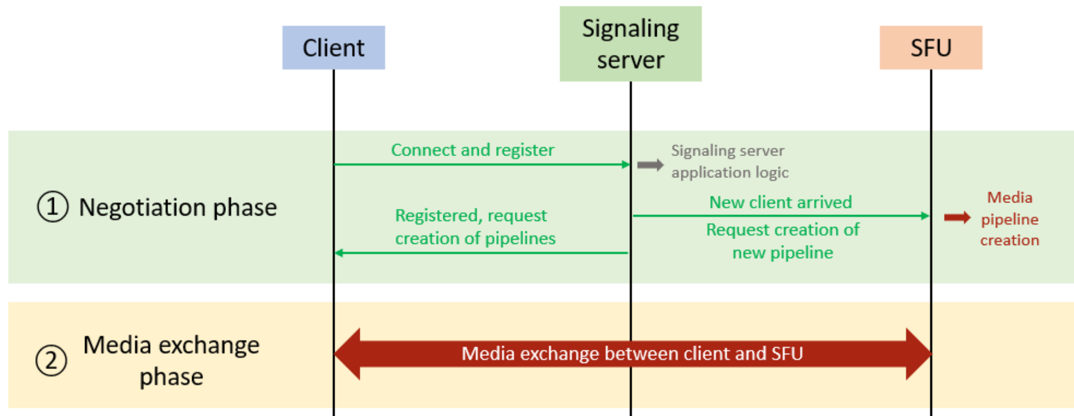


Figure 3.5: Signaling phase leading to the creation of a new media pipeline in the SFU.

build a new transmission chains for the processing of the media that it will receive. Visually with Figure 3.4, the SFU dynamically adds a new pipeline to those already present and builds every additional internal processing steps necessary to receive and redirect the content from the new peer. Moreover, he must enable the other, already present participants' contents to be sent to the new client as well. Once the pipeline and other elements successfully created, the media negotiation phase is over and the SFU and the clients are effectively set up for transmitting their various RTP streams to each other and the proper media exchange phase can start.

3.4 Scalability

We presently have an effective system composed of one SFU redirecting content originating from any number of clients. This is the model we will implement in the next chapter. However, we recall that a single SFU still has scalability issues preventing it from accepting a high number of clients, and even though we will not implement any kind of CDN or network-of-server architecture for this piece of work, we can still research how these concepts can help increase the scalability of our architecture.

We recall that CDNs allow to make live content available to thousands of users at the same time thanks to its numerous edge servers. However, a full commercial CDN deployment isn't needed to reach the goals of this work, as it is very costly to implement. Moreover, state-of-the-art CDNs aren't exactly fit for interactive applications like ours. Indeed, they typically use pull-based transport protocols like MPEG DASH, which isn't as quick to execute as RTP. Furthermore, the edge

servers of a typical CDN are not configured to process media but simply to cache it until it is requested [45].

However, they do give us the solution to the scalability issue, simply by building a network of several SFUs. By having a network of distributed SFUs, each sending endpoint would send and receive RTP streams from the SFU closest to him, while the server network coordinator takes care of delivering the streams received by different servers to all receiving participants of the streaming session.

The implementation of a distributed SFU network brings considerable complexity to the model. The correct functioning of such a network requires using concepts taken from CDN operations. The system needs a coordinator server implementing a user redirection algorithm as well as a content distribution algorithm to deliver the media streams from one end of the network to the other. On closer examination, we realize that we can group together the role of a CDN coordinator component and that of a signaling component. Indeed, the latter is also a governing entity that manages an exchange session by making the connections between client and server. Therefore, we end with a network of distributed SFUs governed by a centralized signaling/coordinator server which has the following roles:

- Selecting one or several SFUs to assign to a given set of peers, and updating its choice when there is a new coming or departing peer.

- Creating the transmission channels between clients and servers.

- Keeping the state of the streaming session updated with communication with the clients based on the signaling protocol used.

- Coordinating the delivery of the live streaming content through the network.

The advantages of such a system include an improved scalability and a better connection quality between clients and servers, meaning that better quality videos can be transmitted. The latency is also reduced since clients are connected to a local server. However, if one RTP stream has to go through several SFUs to get to its destination, the delays due to the several hops can increase the end-to-end latency of the system. Even more so if media processing is to be done at the SFU. Our own SFU implementation for video, as described in Section 3.2.1, suffers greatly from this because of the heavy processing done in the SFU. A solution could be to bypass the transcoding operation when the next hop on the path of the stream is another SFU. The transcoding operation would only be applied when the video quality must be reduced in order to be correctly transmitted. Another disadvantage is the increased complexity of implementing such a system, due notably to the content distribution algorithm that must be implemented and to the connection quality between SFUs, which must also be taken into account.

Such networks of processing servers have been the subject of a lot of research, and there exist commercial implementations in the market. In [60], Grozev *et al.* investigate the use of cascaded SFUs within the Jitsi environment and propose an server selection algorithm based on proximity and load balancing. They use Jitsi's focus component, called Jifoco [61], as the coordinator of the session and group the different SFUs into regional subdivisions governed by a unique Jifoco, thus employing several such signaling servers. In [62], the same strategy is followed, and the authors mention the needed "service" that redirects clients to the right signaling server, which then selects an SFU from the network. Granda *et al.* [63], as for them, implement a network system of MCUs where the peers are gathered in clusters around one MCU, and the MCUs are connected to each other in a full-mesh fashion.

Chapter 4

Implementation

We want to evaluate the performances of our transcoding Selective Forwarding Unit. To achieve that, we implement the model described in Chapter 3 using the Gstreamer framework [64]. In this chapter, we first describe the main features of Gstreamer and why it is a tool adapted to our needs in Section 4.1. Then, in Section 4.2, we present an overview of our Gstreamer implementation.

4.1 Implementation tool: Gstreamer

Gstreamer is a framework that helps programmers to create streaming media applications. Since 2002, it has been included in the GNOME desktop environment, an international project that focusses on developing software frameworks in order for them to be used in end-user applications.

Gstreamer is based on objects called *pipelines*, which in turn are made of *elements* or plugins, through which the data flows. These can be seen in an example shown on Figure 4.1. We can see on the image that the Gstreamer elements are linked to each other to form the pipeline, and are used to handle the audio, video, or any other kind of multimedia data flows. Each element performs a fixed task and nothing more. They also have properties that can be chosen in order to modify the element's behavior. For example with encoders, properties allow to set the bitrate, the number of reference frames or to choose a preset of such parameters. The framework then makes it easy to select the positions of the elements in order to build the desired pipeline. Gstreamer thus relies on the arrangement and selection of elements, and can build even high-end multimedia applications in this way.

The plugins are either given by the framework or can be written by the programmer if he needs a specific task to be executed by an element. The available

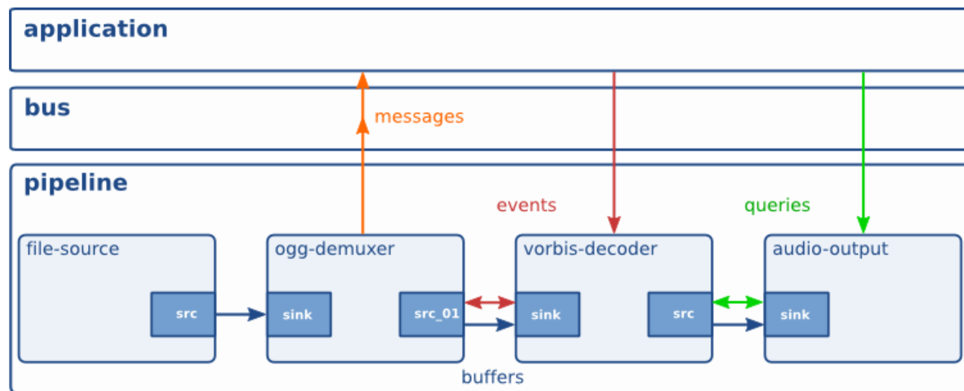


Figure 4.1: Entities that make up the Gstreamer framework [3]

elements are numerous, including transcoders to and from a high number of formats (audio and video, example: MP3, Ogg, H.264, VP8, ...), mixers, demuxers, reading and writing streams from/to files, and many more. An advantage of Gstreamer is that it also allows application designers to build new and customized elements and to plug them in pipelines in order to get the desired stream manipulation. As an example, a JPEG XS encoder isn't included in the Gstreamer libraries, and therefore a plugin acting as such must be written in order for JPEG XS to be used in a streaming application. This is not the subject of this work though.

As can be seen on Figure 4.1, elements can fulfil different roles, either creating, manipulating or consuming the data flow, depending on the *pads* that are associated with them. Pads are objects, shown as little blue rectangles inside the elements in Figure 4.1, through which the data flows in or out of a plugin. Selecting and linking the right pads together is thus a second mean of designing our desired pipeline. In the pipeline, the streamed data flows from one element to the next one in the form of buffers.

Finally, Gstreamer allows to communicate and exchange data with the pipeline thanks to the different signals that are exchanged between the elements, and coming from or addressed to the pipeline. Each element has several signals that it can emit when specific conditions are met. A designer can program its code in order to react to the apparition of a signal by calling on a callback function once the signal is emitted. This is very useful as it allows for example to modify a pipeline dynamically when an event that triggers the signal under watch occurs.

All in all, we can see that implementing the SFU structure presented in Chapter 3 is feasible using Gstreamer by linking elements together in order to create the transmission chains of the peers and of the server. For example with the video transmission, we are now able to build the pipeline represented on Figure 3.3 using Gstreamer elements. Next, in order to construct a fully working SFU, we

will build several such pipelines and arrange them as shown on Figure 3.4. To do that, Gstreamer allows to build structures in several programming languages like C and Python. We will then be able to test our implementation in a realistic environment.

4.2 Implementation

Our implementation is composed of 3 pieces of code: one for the media server, one for the signaling server and one for the client side. The SFU and the client-side codes are written in C, with the help of Gstreamer libraries that allow us to build and manipulate the pipelines that we need. These libraries are well-documented for the C language on the website of Gstreamer. And so, we wrote two `.c` files using a basic text editor on Linux, which we then compiled with GCC in the Linux command line. In the next sections, we describe in more details the functioning and the sequence of operations that takes place in our implementation. All the entities are represented on Figure 4.2, communicating following the model described in Figure 3.5.

4.2.1 The signaling component

In addition to the `.c` files, we wrote a basic signaling server which has to be able to exchange messages with all the other entities and is in charge of controlling the central server. For the communication with the signaling server, we settled on the use of WebSockets to pass the messages, as it is a popular protocol that allows fast and bi-directional sending of requests and responses in real-time. WebSockets also allow to push data to the recipient as events occur, thus providing a way to react to these events in real time. Today, the most popular choice of programming language for writing a signaling server is Node.js, which is an open-source JavaScript environment that supports the use of JavaScript to write code for the server side of applications. It also possesses popular WebSocket libraries like *ws* [65] to create and manipulate WebSocket connections. That is why we chose to write the signaling server using JavaScript, thus giving us a `signalingserver.js` file to execute.

The *ws* library allows to create *WebSocketServers*, which is actually an HTTP server that can be accessed with a simple URL address and a port, and is open to any connection attempt made towards it. This URL address and port thus have to be communicated to the other entities beforehand. In a realistic setting, where the data is exchanged over Internet networks, this is either done by giving a fixed public address to the server, sometimes involving the purchase of a domain name, or by dynamic DNS redirection. In our study case, we gave a fixed address to the

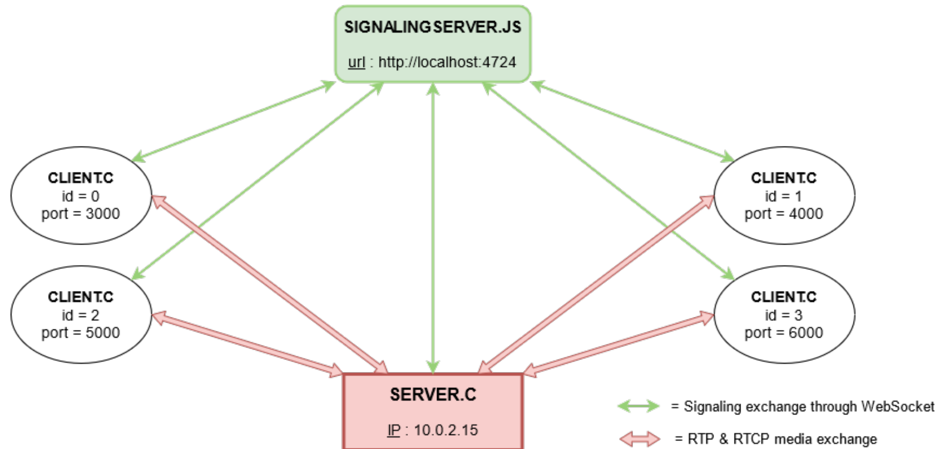


Figure 4.2: Entities of the system and the communication paths between them.

signaling server, simply because we mainly launched the signaling server on an address *http://localhost:port*.

To use WebSockets in the `server.c` and `client.c` files, we import the *libsoup* library which is an HTTP client/server library for GNOME that includes WebSocket methods. The fact that it is also incorporated in the GNOME project in the same way that Gstreamer is, means that both integrate well with each other and that the GLib methods of the GNOME environment can be used with both libraries. As an example, the method that is used to attach a callback function to a signal under watch works in the same way for *libsoup* signals as it works for Gstreamer. We can therefore use this property to watch for the signal that is emitted when a message is received through the WebSocket and execute certain actions depending on its content.

The messages that we exchange via the WebSockets follow the JSON format, a text data format that allows for easy deciphering on either side of the socket.

4.2.2 Sequence of operations

Let us review the sequence of operations that takes place in our implementation. This will allow us to touch on how the different tasks of the model described in Chapter 3 are realized.

We start by launching the signaling server, as it is the element controlling the media exchange session. It begins by creating a *WebSocketServer* and waits for other entities to connect. We can then launch the SFU with `server.c`, which directly establishes the connection to the signaling server thanks to the functions of the *libsoup* library. Once this handshake over, both servers can exchange messages with each other via the WebSocket that links them. The SFU then does

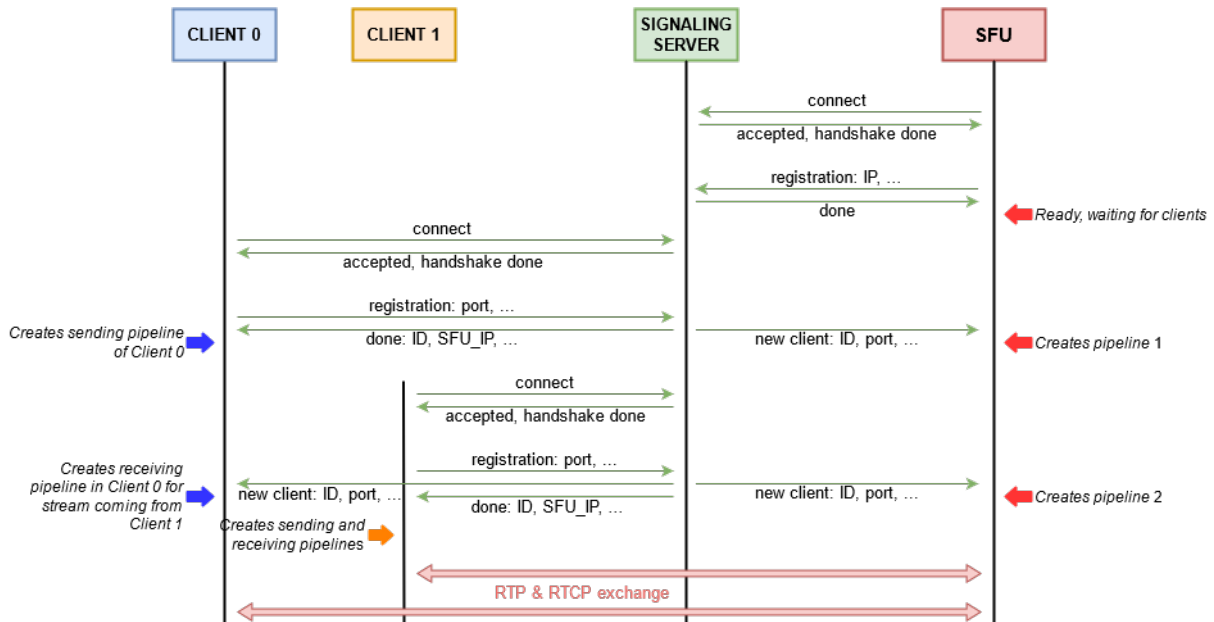


Figure 4.3: Sequence diagram, beginning of the session, for 2 clients.

so, and sends useful information about itself that can be shared later with the clients, like its IP address. The signaling server stores this information and in this way registers the presence of the SFU in its memory.

A number of `client.c` can then be launched, and each connects and registers itself with the signaling server in the same way as the SFU did. Each one is assigned a unique ID number to identify them. When a new client enters the session, its arrival is communicated by the signaling servers to every other entity already present in the conference, along with the useful information that is sent to him during the registration process. The other clients and the SFU can then all adapt and build media pipelines in their memory, as was described on Figure 3.5, and the video transmission with the new client can take place. This process is summed up on Figure 4.3.

4.2.3 Gstreamer elements

Quels sont les pads, properties, et autre des éléments utilisés?

Storage of the elements

In this implementation using Gstreamer, we end the pipelines with elements that take care of sending/receiving the UDP-wrapped RTP streams. They are represented on Figure 4.4, and consist of *rtpbins*, *udpsinks* and *udpsrcs*. These

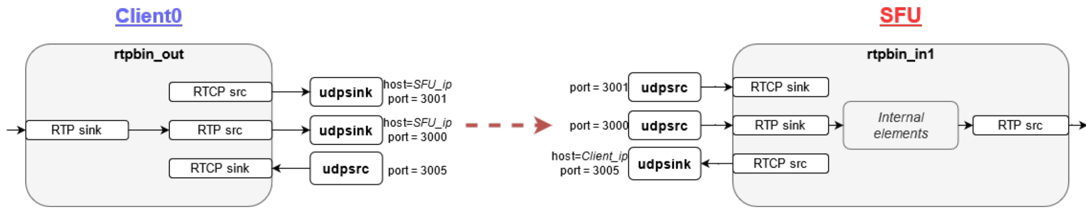


Figure 4.4: The Gstreamer elements in charge of the transmission: the *rtpbin*, *udpsink* and *udpsrc*.

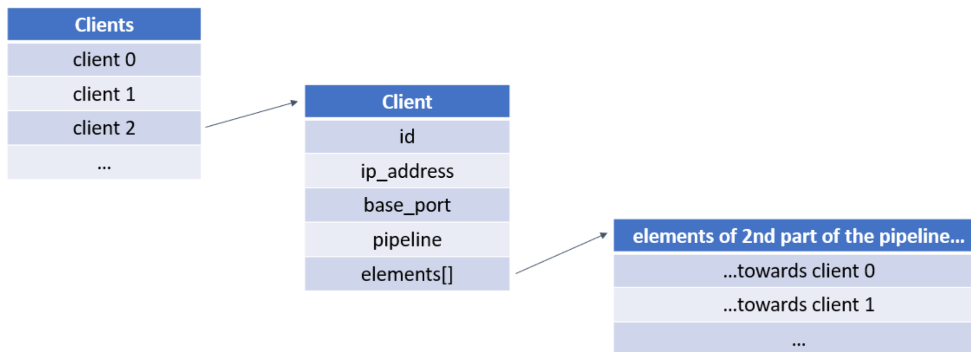


Figure 4.5: Structure of the way the clients are stored in the memory of the SFU.

last two are responsible for the UDP transfer, while the *rtpbin* contains several elements in itself that take care of the RTP packets, including: generation and termination of RTCP messages, collection of statistics, storage in case of NACK, and a jitterbuffer and a demuxer at the receiver's side. Between the SFU and the clients, several such *rtpbin*s pairs are thus created in order to exchange all the different streams of the application.

In order to send/receive the streams with RTP and UDP, the *udpsink*s and *udpsrc*s only need two pieces of information: a destination IP address and a destination UDP port. UDP then allows to send the data in a *push*-fashion, resulting in a reduced latency. The receiver must then collect the data at the same UDP port that the stream was transmitted towards by the sender.

This logic implies that the clients and the SFU must be synchronized on the ports at which they send and listen to the different transmissions. To achieve that, clients must give a base port number from which all other ports used will be found. This and other individual information regarding the client must be stored in memory by the signaling server and the SFU. This information consists of the ID number that is assigned to it by the signaling server, its IP address, and thus the port number that he chooses.

This data is stored at the SFU in a structure *Client*, as represented in Figure

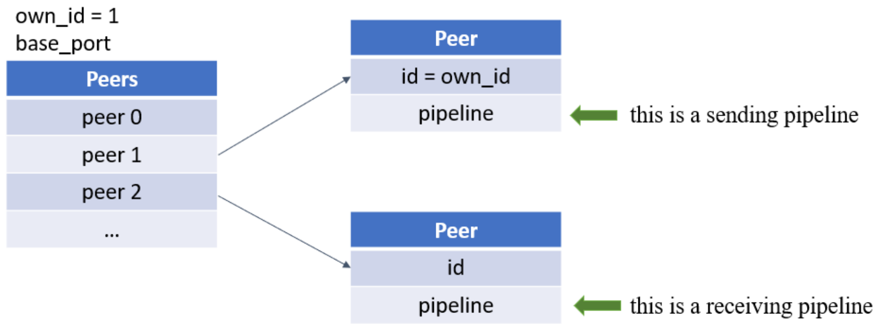
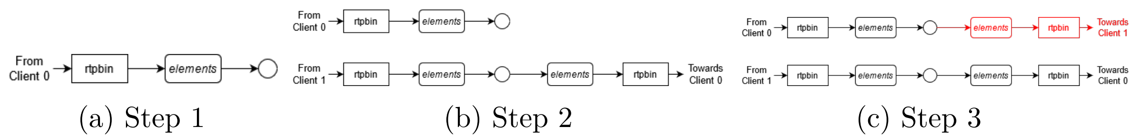


Figure 4.6: Structure of the way the other peers of the session are stored in the clients' memory.

4.5. Once registered in this structure, the SFU can start building a Gstreamer pipeline that is also stored in *Client*, representing the sending transmission chain of Figure 3.3, up to the point where the data is split in several branches. Indeed, to build the second part, we need other clients to join and to communicate their IP addresses and ports. Once that happens with a second client coming in, the SFU can complete the first pipeline by building a branch that redirects the stream towards the second client, as well as building a full new pipeline transmitting the data from the second client to the first client. In the end, we arrive at the architecture shown on Figure 3.4. Simply by knowing the ID number and base port of every client present, the SFU and the clients themselves are able to deduce the different ports corresponding to the different streams and are therefore always synchronized on where to send and to listen for data.

At the client's side, the same process is followed by storing the IDs of every other peer in a structure called *Peer*, as shown on Figure 4.6. A sending pipeline is always present in order to share the client's own media, looking like the transmission chain of Figure 3.1. Then, when a new participant joins the session, the client builds a receiving pipeline that operates like the chain of Figure 3.2.

In order to make the synchronization of ports work, we assign the IDs such that its value is always between zero and the maximum amount of clients that the session can support. The number is chosen starting from zero and increases as the peers connect to the signaling server. If a client leaves the session, its ID is given to the next incoming peer. In addition to the unique ID that is given to him, a client provides a base port to where he sends its data flow. Knowing both these pieces of information, the SFU can calculate the destination ports where it redirects the streams of another client with the result of the subtraction of their two IDs. It then multiplies the result by a constant that it adds to the base port to find the destination port. The clients follow the same formula and thus know the right ports to listen to.



(a) Step 1 (b) Step 2 (c) Step 3

Figure 4.7: The dynamic building of the pipelines and the problem encountered.

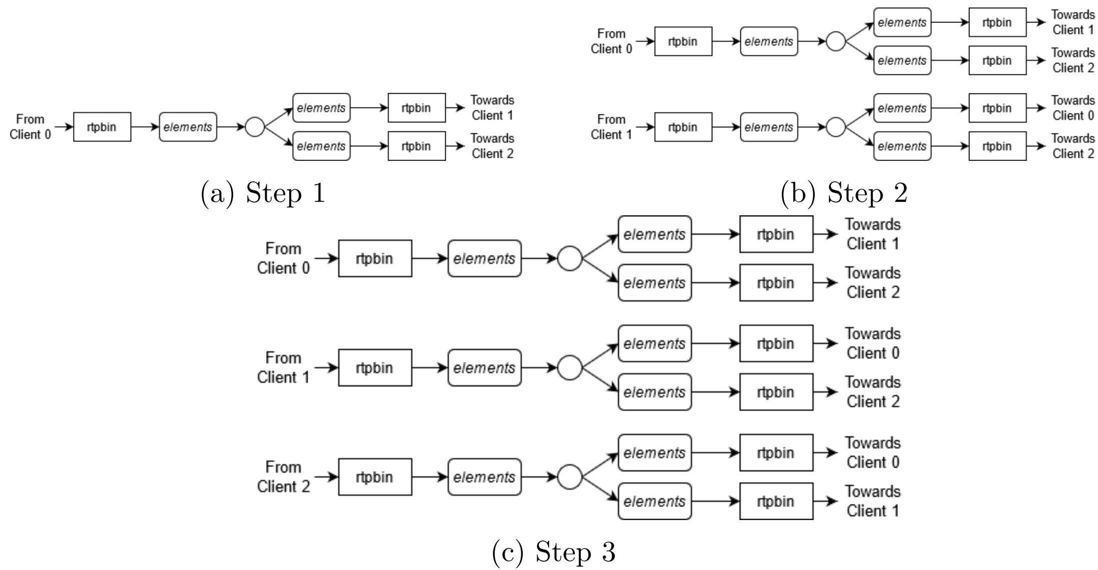
As a client should be able to leave the session freely, the system must be able to delete any of these pipelines at any time. That is why these objects must be kept in a known place in memory in order to be accessed whenever we want to destroy one of them. Therefore, the SFU must also store for every client all the elements that build the redirecting branches, as is represented by the third table of Figure 4.5. The same is done in the client's implementation, for every receiving pipeline related to another peer. In that way, when a client leaves, the signaling server sends a message to all the entities in the session, which can then delete the exact pipelines and branches that are used for this client off their memory.

The fact that we can access any point of the pipelines at almost any time is also very important for the implementation of the adaptive bitrate feature. Indeed, the re-encoding of the stream takes place in the second part of the pipeline, in order to be adapted to every recipient, and is thus present in the *elements* structure of Figure 4.5. We can then access that encoder at any time and change its parameters in order to adapt the stream.

The problem encountered

Unfortunately, we did not manage to complete this exact implementation the way we wanted because of a problem that prevented us from correctly constructing the pipelines in the dynamic fashion described above. This problem is represented in three steps in Figure 1. Let us consider the beginning of a session with a first client, Client 0, joining the signaling server. The latter transmits the information to the SFU, which can then react by building the first part of the pipeline corresponding to this peer, as shown in Figure 4.7a, step 1. In our original idea, the SFU waits to build the branch that constitutes the second part, since no other peer is present yet, and therefore we do not know the parameters to configure the *rtpbin*, *udpsink* and *udpsrc* of the branch. Once the first part constructed, this pipeline is stored in the second table of Figure 4.5, and the SFU can wait for another client.

Next, a second client, Client 1, joins the session. The SFU, knowing that Client 0 is already present, can immediately build Client 1's entire pipeline. It declares the new pipeline, adds the various elements to it, links them together and stores all these objects in the different arrays in its memory, as shown on Figure 4.7b, step 2.



(a) Step 1 (b) Step 2

(c) Step 3

Figure 4.8: Substitute solution, with 3 clients in the session.

The problem arises when we go back to Client 0 and fetch its pipeline in order to add to it the elements of the branch towards the second client. Indeed, at the time when we try adding these elements to the pipeline, this one is simply not recognized as a Gstreamer object by the program anymore. The functions and methods of the Gstreamer libraries can therefore no longer be applied to this pipeline and we cannot add or change anything about it anymore.

The way around

This issue forced us to reconsider the way the pipelines in the SFU are to be constructed, and a solution had to be found quickly in order to have a functioning architecture. In the code, we use the same function, called `add_client_to_pipeline()`, to build both the branches of Client 0 and Client 1. But this function only seems to work when called upon inside the function `create_pipeline()`, that is in charge of declaring and constructing the first part of the pipeline, as is the case for Client 1.

Therefore, in a substitute implementation, we will only make calls to `add_client_to_pipeline()` inside the function `create_pipeline()`. This means however that we have to build all the secondary branches in a row, even if there is no other client in the conference. This is shown on Figure 4.8, presenting the situation in case of a conference between 3 peers. We see that the SFU must immediately build the entire pipelines from the receiving *rtpbin* to the several sending *rtpbin*s as soon as a client joins the session.

This backup solution certainly has its shortcomings. Mainly, the number of secondary branches that will be built at the arrival of a client must be set before launching the session. And until the right number of clients is connected to the SFU, some branches will serve no purpose and waste some important server resources. However, once the conference reaches the maximum capacity set before its start, the architecture, pipelines and the way the data flows and is redistributed by the SFU is exactly the same as was intended in our original implementation. To operate with the best performances, this alternate solution must therefore only be used at maximum capacity. The main difference between the two solutions is thus that the exact number of participants must be known before launching the system.

The signaling server, which works perfectly fine, can also be used in a very similar way as he was originally, even though its task as the entity controlling the dynamic addition of clients is somewhat not as important now, since we know from the beginning how many there will be. The way the information is stored at the SFU's and at the client's side hasn't changed, and neither has the structure of the pipelines. While this failure of correctly implementing the dynamic addition and removal of clients is clearly a missed goal of our work, we can still verify the correct functioning of the SFU architecture on this backup solution and see how the pipelines and the SFU architecture itself perform. By setting the established number of participants from the beginning, and by varying that number, we get the same arrangements of pipelines as if the model and first intended implementation described above was successfully running.

4.2.4 Adaptive bitrate algorithm

To adapt the bitrate, manipulation of the RTCP packets is needed. Fortunately, the *rtpbin*s of Gstreamer present a number of methods and signals that help in this regard.

First, we start by saying that we did not use the REMB packets method could in our implementation. Indeed, REMB is an extension of the traditional RTCP packets, but unfortunately no official Gstreamer library allows to add or manipulate them easily. Some plugins and functions have been written by commercial open source projects, Kurento and WebRTC to name a few, but they are a part of these infrastructures and removing only the part that interested us in order to implement the protocol ourselves would have taken too much time and was therefore not an option.

Instead, it is easier to simply remove the information necessary to implement an adaptive bit rate algorithm from the RTCP packets received during the exchanges between *rtpbin*s. By using an RTCP detection signal on Gstreamer, it is possible to call a function as soon as RTCP is transmitted or received from an *rtpbin*.

Thanks to this, we can retrieve the statistics of an RTP session between the server and a client. On the server's side (the sender), we can extract the number of RTP packets transmitted since the last RTCP message. At the same time, at the client's side, the reception of this message and the response from the receiver allow us to know the value of the packet loss. We can extract this information and communicate it back to the SFU. Since it is a communication that contains no media, it has to be done via the signaling server. This one just acts as a relay here and redirects the message directly to the SFU. By communicating the number of packets received to the server, this one can decide whether or not the difference between the packets sent and received is small enough to increase the bit rate of the encoding.

This system therefore requires finding thresholds of lost packets above or below which the bitrate must be modified. There are naturally multiple ways to increase and decrease the bitrate. We chose simplicity by increasing or decreasing the parameter by a constant amount every time.

Unfortunately, we quickly realized that the issue that we encountered in the building of the pipelines finds its way in this part of the implementation too. Indeed, as the pipeline was, once the element in charge of re-encoding the video stream stored in the memory while the program builds or takes care of something else, it is later no longer considered as an element by the program. We can thus not use the appropriate Gstreamer functions that allow to modify the parameters of an element. So even though we were able to fetch and compare both counts of sent and received RTP packets, while this problem still lingers we just change the bitrate parameter of the encoder. Therefore, we regrettably will not be able to test the adaptive bitrate algorithm as we wanted.

4.2.5 Limitations of the implementation

Admittedly, this implementation is a simplified version of the theoretical model presented in Chapter 3. It lacks several important elements before being adapted for deployment and commercial use. Obviously, the problem that we could not solve plays an important role in those limitations, which was already discussed in the previous sections. But a couple of other features are not implemented in this work. They are listed here.

First of all, the fact that the Gstreamer elements that we use in the pipelines only send the RTP streams thanks the IP address and UDP port of the recipient limits importantly the features available for the transmission. First, there is no security implemented at all, no encryption, no message authentication, and no user authentication. Therefore, this implementation should only be run in a closed environment, as was established in the introduction. However, adding a security layer to the system shouldn't pose many problems, as there exists Gstreamer

elements implementing security protocols such as the Secure Real-time Transport Protocol (SRTP). Also, running in a closed environment allows us to avoid any firewall problem that might arise in real networks. In consequence, ICE, STUN and TURN protocols are not employed here. These will be needed, however, if the system is to be run in real networking conditions, with servers and clients located at different and potentially distant places on the network.

Next, in the same vein, this implementation comprises only one media relay server, the SFU in `server.c`. We did not implement any multi-server architecture, as it was not the objective as told in Section 3.4. Interested readers can refer to [60] for research and implementation of networks of SFUs.

Secondly, as we were developing our implementation, the Gstreamer plugin of a JPEG XS encoder was not available for us to include in our pipelines. Instead, we thus resorted to x264 encoders, the popular encoder of the H264 standard. This standard is still widely used and adapted to real-time applications thanks to its lower end-to-end latency compared to other popular standards, at the expense of lower compression gains.

Finally, the reader can remark that only the implementation of video-manipulating pipelines are addressed in this chapter. This is because other media streams, including the gaming stream, follow exactly the same logic of implementation as the one described here. Naturally, the difference will reside in the fact that no video coding is employed in other situations. Therefore, these elements can be removed from the pipelines of the gaming stream. As encoding and decoding is the main cause of demand for CPU usage, and requires a jitter buffer which is one of the main source of latency, pipelines that do not require any of them can only perform faster and with better scalability.

Chapter 5

Evaluation

In this chapter, we test the Gstreamer implementation described in the previous chapter in order to assess its functioning and performances. We first describe the environment in which the tests are made in Section 5.1. Next, we present results of the tests on our program. We first have a look at the CPU utilization of the SFU in Section 5.2.1. We then investigate the latency within the SFU in Section 5.2.2. We close the chapter on a global discussion of our results and what they mean with respect to our model in Section 5.3.

5.1 Environment

All this project was developed on a single computer, using an Ubuntu guest on a VirtualBox machine, which runs with 8 GB of dedicated memory and 3 processor cores. It is through this virtual machine that the code was written and tested. Since we only had at our disposal this single PC for running our program, we were constrained to do internal tests only. Meaning, all the pieces of code (`signalingserver.js`, `server.c` and any number of `client.c`) had to be launched from a different command line on this same virtual machine.

This limits us a bit as it prevents us from doing full-scale tests of the effectiveness of our implementation. We are instead constrained to do less realistic tests. Indeed, given that we work on a single machine, we only have one destination IP address which is the internal IP address of the local host virtual machine. As a result, the transmissions through the *udpsink*s and *udpsrc*s are quasi-instantaneous.

Nevertheless, we can still perform measurements on the local pipelines and the results are presented in the following section.

5.2 Results of the tests

First of all, we used a Gstreamer debug tool that allows to output graph files of the pipelines created. This allows us to visualize every element of the pipeline, as well as the internal elements inside bins like *rtpbin*. Before getting into specific performance tests, the reader can find the full pipeline graphs of the pipelines of our implementation for the regular use of our implementation in Appendix A.

In the following sections, we present the measurements we made on our implementation.

5.2.1 CPU usage

CPU utilization is vital parameter to our system, and this is true on both sides of the channel. If the processor is overused, there are obviously less clients that can participate in the exchange session, which increases infrastructure costs as more servers are needed for more scalability. An overloaded CPU also negatively impacts the encoding and slows down the response time of the system globally.

Here, we can measure the effect on the CPUs of the multiple encoders at the SFU and compare it to encoder-less solutions such as proposed for the gaming stream or for simulcast. To obtain that, we ran a number of SFU pipelines in parallel while recording the CPU loads, first on an SFU containing encoders and decoders and then on an SFU that justs duplicates the data from the input *rtpbin* to the output *rtpbin*s. By choosing the number of running SFU pipelines, we can present the results as a function of the number of clients present in the session.

Experiment setup

We want to isolate the load of the SFU only to see its performance. Since we work on a single machine, that means that clients have to be removed from the simulation, so that their participation in the system load is not taken into account. Therefore, we had to slightly modify the SFU pipeline. Since no client sends a video stream to the input *rtpin*, we replaced it with a source playing a file containing a video already encoded with the H264 codec. This video is test sample taken from [66] with a 1080p resolution and a framerate of 25 fps. The video lasts about ten minutes. We then made the SFU run several times with a varying number of pipelines in order to simulate the number of clients in the session. In this way, we recover the optimized model from Chapter 3.

In Linux, we can measure the CPU load by looking at the load averages. These are composed of three values representing the average system load calculated over a given period of time of 1, 5 and 15 minutes. More exactly, they represent the average number of processes in the "ready for running" queue for these periods

of time. These values must be compared with the number of cores the virtual machine possesses. In our case, there are three processor cores in our virtual machine. Therefore, a value of load average lower than 3.00 indicates that CPUs were idle some of the time, with no processes waiting for CPU time over the period of time; a value equal to 3.00 means that the CPUs were fully utilized; and a value superior to 3.00 indicates that the CPUs were overloaded, with processes waiting in the "ready for running" queue.

The load averages can be seen in Linux using various tools, popular examples being *uptime* and *htop*. However, we want to measure the evolution of the SFU load average as a function of the number of clients. We therefore employed a tool called *sysstat* that monitors system performance and usage activity of for Linux. *sysstat* possesses features that collect and historize performance and activity data, and so we can use it to record load averages over time.

Results

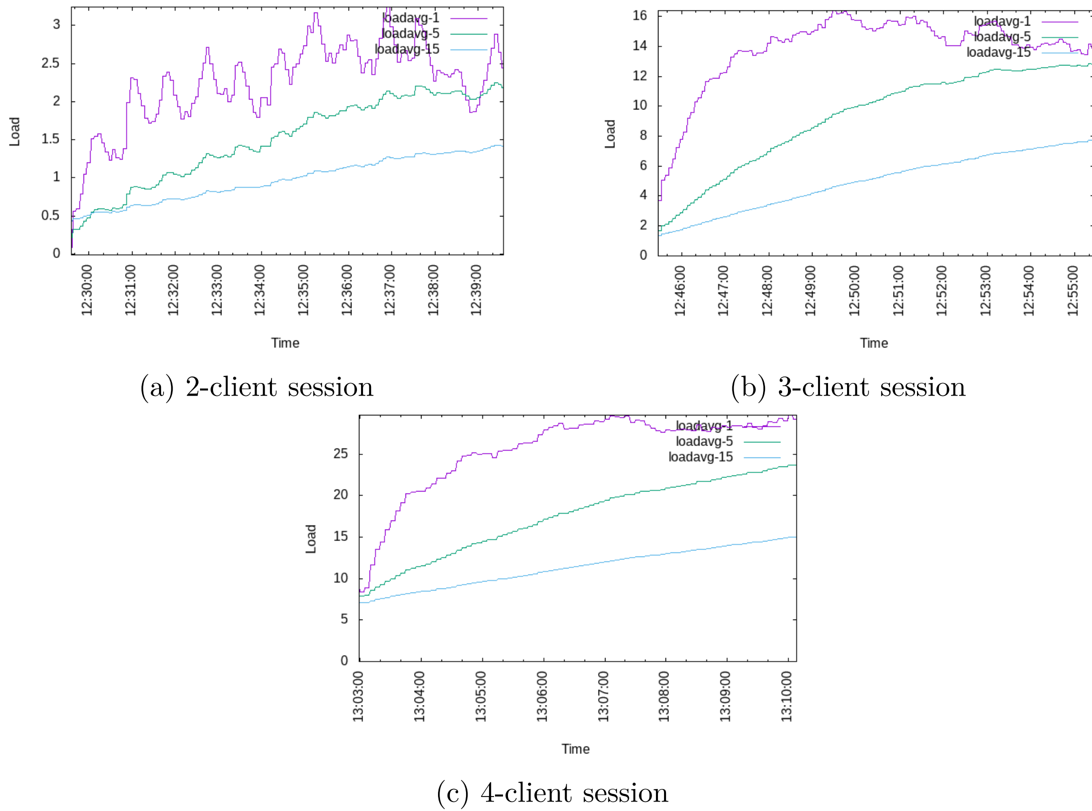
In first place, we do the simulation in the case of an SFU containing decoders and encoders. We first launch the server ready to re-route for 2 clients (thus building 2 pipelines), then 3 clients, and finally 4 clients, and we let *sysstat* run for the duration of the video. We save the values in a file and plot them on a graph over time. The results are shown in Figure 5.1. In Figure 5.2, we present screenshots of what *htop* shows at some point during the operation in each case.

On the graphs of the load averages, we are mostly interested in the results over one minute. Indeed, the video being only ten minutes long, the 5-minute and 15-minute averages do not have the time to adjust to the average number of processes in the queue.

We see on Figure 5.1a that for two clients, a transcoding SFU has a load average between 2.00 and 3.00. In this case, a 3-core CPU, like our computer, is therefore mostly idle and is able to handle the load of the SFU. With two pipelines running in parallel, this SFU contains two H264 decoders and two encoders running at the same time. On Figure 5.2a, we indeed observer that the three cores are being used at around 60%.

Next, we increase the number of clients to three and observe the results on Figure 5.1b. We observe that the load average in this case jumps to a value between 13.00 and 16.00. This is much larger than the 3 CPUs of our virtual machine can handle, and indeed we see on Figure 5.2b that the processors are fully employed. In this case, there are three decoders and six x264 encoders running concurrently.

Finally, Figure 5.1c shows what the loads are when the number of clients is four. There are now four decoders and twelve encoders running in the pipelines of the SFU. We see that the load average settles around 27.00 to 30.00 processes.

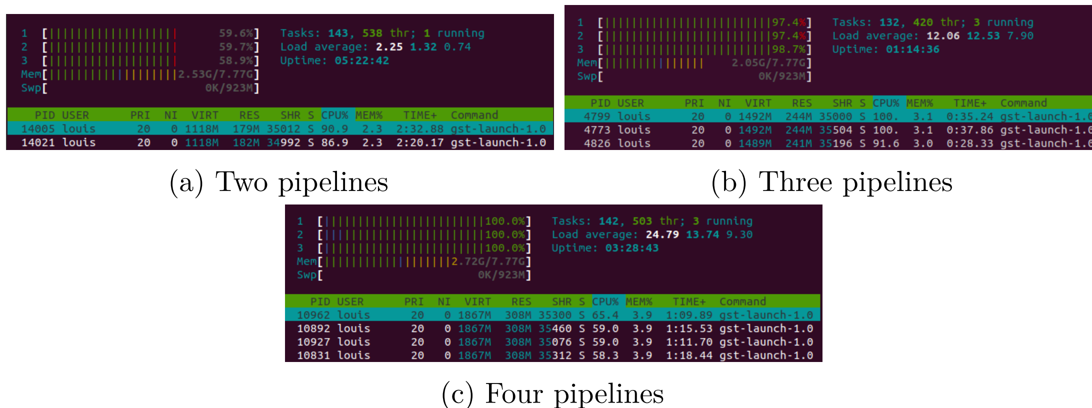


(a) 2-client session

(b) 3-client session

(c) 4-client session

Figure 5.1: Graphs of the CPU load over time for an SFU with transcoding.

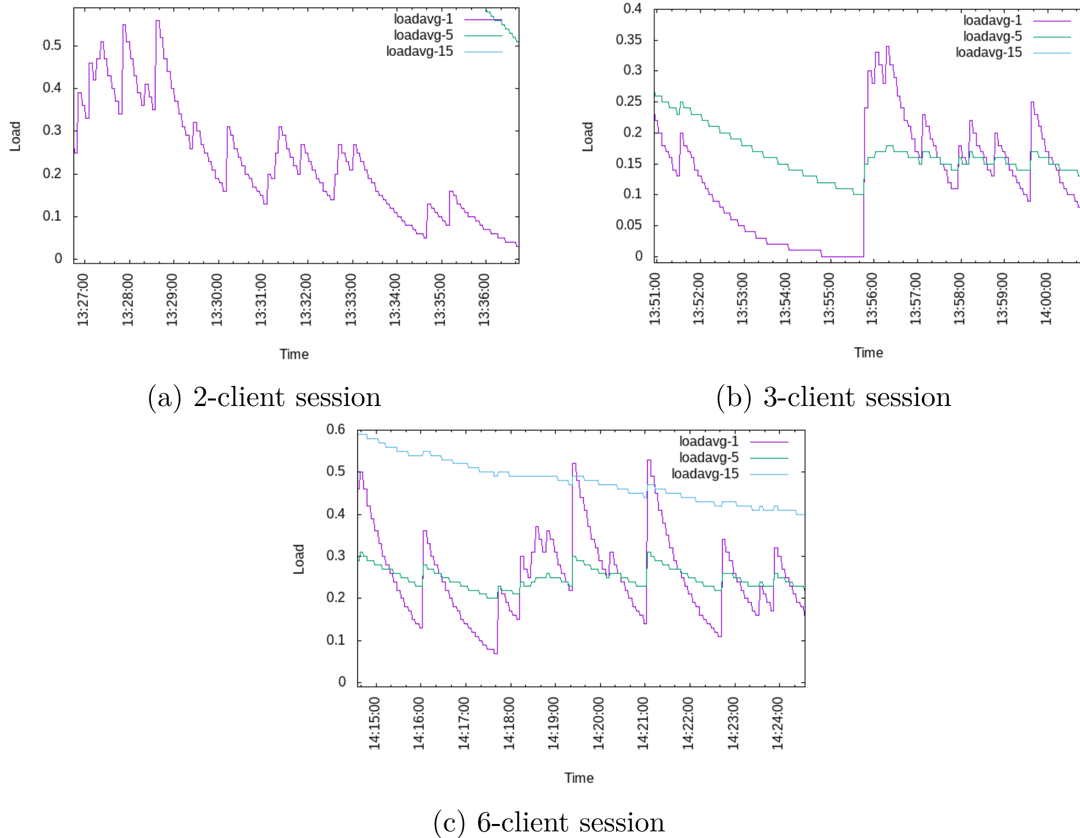


(a) Two pipelines

(b) Three pipelines

(c) Four pipelines

Figure 5.2: *htop* showing the CPU load for an SFU with transcoding.



(a) 2-client session

(b) 3-client session

(c) 6-client session

Figure 5.3: Graphs of the CPU load over time for an SFU without transcoding.

We didn't go further than that, as the virtual machine ws reaching the maximum of its performances. Even *sysstat* itself had trouble monitoring the loads in real time.

There aren't enough results to deduce a formula of any kind for the CPU load average of the SFU. However, we know that for N clients connected to it, there are N decoders and $N * (N - 1)$ encoders that do the transcoding. If each of them requires the same amount of CPU resources, we can therefore expect the load average to grow quadratically with the number of clients.

Next, we can compare these results with the ones obtained without any transcoding performed at the SFU. This is the case when we transmit the gaming stream, or for example in the case of simulcast. We again present three measures carried out with *sysstat* over the duration of our test video. The results are shown on Figure 5.3.

As we ran the tests one after the other, we can see that the 5 and 15-minute load averagse value are still affected by the previous tests. They therefore do not appear on the first graph.

When no transcoding is done at the SFU, the pipelines in the SFU between the *rtpbin*s only consist of the element that duplicates the data, called a *tee*. This requires very little processing resources, as we can see on all three subfigures that the load average of these is around 0.30. We first simulated with 2 and 3 clients, then, seeing no significant difference between the two, we launched a session with 6 peers with again no significant rise of the load average.

Comparing these results with those of the transcoding SFU, we observe that decoders and encoders are indeed very heavy on computing resources, and are the main components that influence the load average of the CPUs.

5.2.2 Latency

Having a low latency is one of the main guideline of this project, as it is of capital importance for the transmission of the gaming stream. As explained in Chapter 2, the main sources of latency in the pipelines are the network, meaning the distance between the peers and its quality, and the jitter buffer.

Here, we cannot measure the latency introduced by the network, as the transmissions are almost instantaneous. However, we can still get the overall latency of the pipelines using different techniques.

Indeed, measuring the latency can be done in multiple different ways. First, a common technique is by transmitting a video of a stopwatch with millisecond precision. By subtracting the time shown at the receiver from the original time at the sender we get the end-to-end delay between the two. This technique, however, depends a lot on the refresh rates of the capturing device and of the monitor.

We added this technique to our implementation in order to measure the latency. To do that, we inserted a *tee* in the sending client's pipeline in order to duplicate the video stream. We can then display the video on the sending peer's side, and compare it with the video that is received and displayed at the recipient, after having gone through the SFU. To help us compare the two videos, we add a timer in the top-left corner of the video. We were able to do this with the encoder-less SFU, and the result is shown on Figure 5.4. In the encoder-less SFU, the only potential sources of delay to the data flow are the input and output *rtpbin*s and the duplication of the stream.

We see on Figure 5.4 that both clocks indicate the same time. Therefore, the delay between the sending and receiving clients is very close to zero. This is of course due to the test being made in a closed environment between entities that share the same IP address. But it means that for the gaming stream, which is the stream actually intended to be transmitted through the encoder-less SFU, the only cause of latency will come from the network. Indeed, as explained in Chapter 3, for the gaming stream the relay through the SFU does not go through any

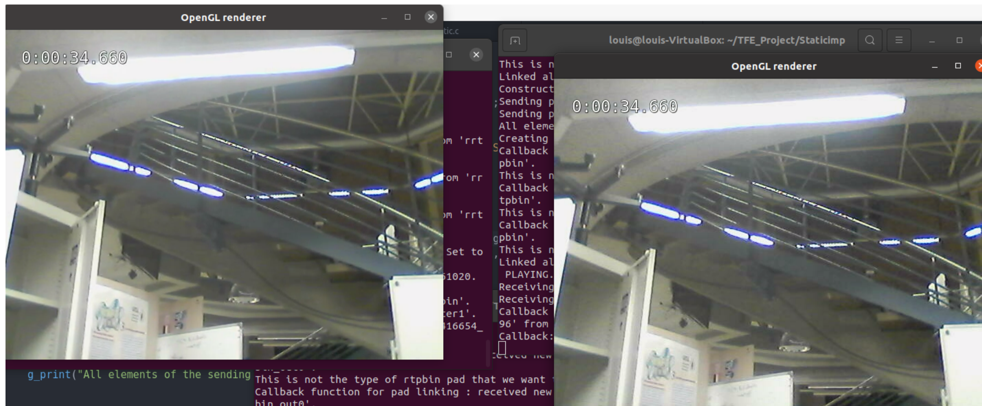


Figure 5.4: Latency test: same video stream displayed at the sender and at the receiver.

important delay-inducing element, as a jitter buffer isn't needed and no encoders are employed.

However, this method doesn't show good accuracy because of the rendering time and the loads that it adds to the CPU. In fact, we could only make this test work with the encoder-less SFU and with a livestream of the integrated webcam on our computer because it is of SD quality. Once we tried upgrading the quality or switching to the transcoding SFU, the sum of the processor resources demanded from all the encoders plus the renderers made the display of the video slow and with unconstant speed, such that this method could not be applied anymore.

Another technique to measure latency would be to get the time when a certain buffer arrives at the source or sink pad of an element. We found a Gstreamer plugin [67] that allows to do just that as it measure the time that a buffer makes to get from one point in the pipeline to another downstream. It consists of two elements, *markin* and *markout*, that must be placed around the section of pipeline under study. *markin* places a tag on every buffer that goes through it with a timestamp, and this one is then read by *markout* which can then compare it to the actual time and outputs the difference on a log, as can be seen on Figure 5.5.

We re-used the pipelines that we made for the measures on the CPU. However, while running the tests we realized that the plugin had trouble with the *tee* element, and that pipelines with multiples branches returned nothing on the log, as wee can see on Figure 5.6. In the end, we thus used the plugin to measure the delay introduced element by element in the SFU. This allows us to compare the measure with the theory of latency that was presented in Chapter 2.

First, let us take a look at the desencapsulation of the stream from the RTP protocol, which is represented in Gstreamer by the element *rtph264depay*, on Figure 5.5. We see that the process adds absolutely no latency to the data flow. In

```
704 0:00:03.939751649 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
705 0:00:03.952744243 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
706 0:00:03.976109985 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
707 0:00:03.986706645 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
708 0:00:03.997423646 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
709 0:00:04.015512571 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
710 0:00:04.032662412 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
711 0:00:04.042360984 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
712 0:00:04.053932238 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
713 0:00:04.069601232 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
714 0:00:04.084280232 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
715 0:00:04.100001898 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
716 0:00:04.110489417 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
717 0:00:04.119629058 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
718 0:00:04.127071854 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
719 0:00:04.145996766 14829 0x55be75c57360 DEBUG    markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
```

Figure 5.5: Example of the ouptut of *markout* on the log. This is for the *rtph264depay* element.

```
markout gstmarkout.c:162:gst_markout_transform_ip:<markout0> no marking metadata found
markout gstmarkout.c:162:gst_markout_transform_ip:<markout0> no marking metadata found
markout gstmarkout.c:162:gst_markout_transform_ip:<markout0> no marking metadata found
markout gstmarkout.c:162:gst_markout_transform_ip:<markout0> no marking metadata found
```

Figure 5.6: *markout* error when multiple branches present.

Chapter 2, we found in the literature that putting the stream in the RTP payload added a small latency that depended on the size of the buffers. Indeed, bigger buffers would have to wait to be full before being processed, while diminishing the buffer size to a minimum would add too much overhead to the operation as the encapsulation would have to be done too many times per unit time. It appears that Gstreamer is set with an ideal buffer size and that the small latency is actually totally negligible, being less than a millisecond. In the same way, on Figure 5.7, the encapsulation in the RTP protocol does not take any time either.

Next, we place the *mark* elements around the H264 decoder. The result is shown on Figure 5.8. We can see that there is indeed latency introduced by the element, taking value between 80 milliseconds and up to 290 milliseconds. This is an expected delay, even though the values are important for an H264 decoder. This is why JPEG XS was introduced in this piece of work, so that video coding could be adapted to real-time applications.

```
errlogpay.log
~/TFE_Project
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 0ms
```

Figure 5.7: Output of *mark* plugin for the RTP encapsulation element, *rtph264pay*.

```
errlog_avdec.log
~/TFE_Project
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 280ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 83ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 157ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 242ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 84ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 163ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 125ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 283ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 82ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 164ms
markout gstmarkout.c:170:gst_markout_transform_ip:<markout0> Mark Duration: 122ms
```

Figure 5.8: Output of *mark* plugin for the H264 decoder, *avedc_h264*.

5.3 Discussion of the results

The results presented in this chapter were restrained due to the environment on which the tests were made. However, they confirm what was presented during the literature research of Chapter 2. Let us review them.

First, we can say that the CPU measurements can be accepted thanks to the unit we have chosen to represent them. Indeed, the load averages allow us to have an idea of the number of cores and the CPU speed needed to run a certain program. For example, we know from our results that we need about five times as much CPU power to run an SFU that will accept 3 clients without overloading the CPUs. This result allows us to know what to expect when deploying this system.

However, this last example is quite representative of a definite flaw in our model, which is the computing load that is required to run an SFU with so many encoders. This flaw had already been announced in the Chapter 3, but the results presented here only confirm it, by showing that a single computer cannot accept more than 2 clients.

As deduced in the previous section, this overload of the CPUs comes almost exclusively from the numerous encoders and decoders present on the server. Transcoding has been introduced in the project in order to be able to adapt the quality of the stream to the network quality of each client. Unfortunately, this feature will be expensive when deploying the system, since it will require machines powerful enough to allow for large sessions if we want to realize large "multi-player" theater games.

Then, the results from the latency measures unfortunately do not teach us much. Indeed, with only the session internal to a single computer, we were not able to measure total end-to-end latency that this system would give in a realistic environment. However, as was explained in the previous section, we could take off a couple of claims from the few results that we had. First, let us say the timer measure made in this test is probably not exactly accurate. Indeed, even with an internal environment setup like ours, a sub-millisecond delay is unlikely

with the various steps that the stream must go through. This is likely due to the low-quality rendering from the program. However, we repeated this test numerous times and never found a big difference. We can deduce that this measure confirms again that the encoder-less SFU architecture possesses the advantages described in the theorical part compared to its other alternatives.

Then, the second measure of latency made on the H264 decoder also confirms that the transcoding server will add an important delay to the video stream. This delay can be reduced however by choosing the right parameters on elements of the pipeline, and by using JPEG XS instead of H264.

That makes two arguments against the addition of decoders and encoders in the SFU. This was to be expected, as the presence of video transcoding makes the SFU lose its advantages over an alternative like the MCU. Moreover, it employs more encoders. The addition of transcoding was made in order to have congestion control at the central server, at task that was announced to by difficult to implement in the research. However, we can put on hold the inclination to refuse our proposed model, since we have not seen yet how JPEG XS can change the results presented here. As explained in Chapter 2, JPEG-XS is supposed to be very fast and very lightweight, in order to be employed on drones for examples. Such advantages compared to H264 will make the architecture that we proposed less expensive to deploy, and will reduce the latency on the video. However, for the encoder-less SFU, the results indeed confirm what was presented in the theory, and we believe that it is the best architecture to transmit the gaming stream from cinema to cinema.

Chapter 6

Conclusion

The goal of this thesis was to explore the existing technologies and architectures that can be incorporated into a live video streaming system in order to find a solution to the initial problem set in the introduction. This problem was to find a way to distribute and exchange video streams between movie theaters so that live crowd interaction games could be installed in cinema rooms. Then, the proposed model had to be implemented with the Gstreamer framework, in order to be able to use it to realize some performance tests.

The model had to meet five main guidelines. First, very low latency between the theaters, especially for a sensor-based gaming stream. This was required in order for the game to react as fast as possible to the actions of the in-game audience. Second, the system had to be able to support multiple simultaneous video streams, thus sessions between more than two cinemas. Third, as theaters can be located hundreds of kilometers away from each other, the system had to be capable of providing for long distance communication. Next, the system had to be prepared to adapt to the network state in real time. Finally, statistics on the live video streams must be available to collect in order to monitor the quality of the session in real time.

To address this problem, we started this project by studying the literature on existing live-streaming technologies. This allowed us to identify the different components that are a necessary part of such a system as well as their theoretical impact on its performance. In particular, we were able to research which parameters we were interested in when choosing a video coding standard using the principles of JPEG XS, the standard imposed to our model. We were then able to consider several backend architectures, and we concluded that the Selective Forwarding Unit (SFU) concept was the most suitable for our purpose.

Once we had decided on the ideal model, we then turned to its implementation with a program written on Linux. In order to develop this application, we used in

the Gstreamer library, which allowed us to implement the different transmission chains established for the model in Section 3.2. However, this step of the work did not go without some difficulties. Indeed, we had to change path at the last moment on our original intended implementation in order to obtain concrete results. Nevertheless, the obtained solution is equivalent to the original one under some conditions, which allowed us to finally conduct some measurements on this model.

6.1 Review of the guidelines

- **Low latency:** For a centralized architecture, an SFU offers the best in terms of latency regarding the gaming stream. Indeed, this one is simply re-routed from a client to the others without undergoing any modification at the central server, only duplication and possible storage. On the contrary, the video stream undergoes several manipulations due to the transcoding at the SFU level. With the Gstreamer implementation, we could see thanks to the measurements of the Chapter 5 that the encoders add some latency to the SFU, but the results show us especially that the number of encoders will have a more significant impact on the resources used by the server.
- **Multiple simultaneous video streams:** The SFU accepts the addition and removal of as many clients as its computing resources can support. In order to add and remove these clients dynamically, we have written a signaling server that communicates with the entities present in the conference and thus controls its progress. Our implementation therefore accepts several clients. Nevertheless, as explained in the Chapter 4, difficulties encountered during the writing of the code compelled us to modify the way the SFU reacts to the addition of a client. This forces the program to know from the beginning the number of participants in the session in order to operate with the best performance.
- **Long distance:** The concept of CDN allowed us to consider a network of SFUs or edge servers to allow for the best possible delivery of the streams to the clients. For that, each client would be redirected to a server that is most appropriate for him, and the network of SFUs would push the content down to the client. However, this part of the requirements was beyond the scope of a Gstreamer implementation in C language.
- **Real-time adaptation to the network condition:** This point is the most problematic when choosing an SFU topology. We have seen during our literature research that congestion control for such an architecture must be

explicitly implemented, and research is still being done on the best way to do so. Solutions such as simulcast are notably interesting because they do not require transcoding at the SFU. In our case, we have decided to integrate the encoders into the central server, in order to be able to re-encode the video streams in a quality that is best suited to the quality of the network towards the recipient. However, this requires a plan to adapt the bitrate to the available bandwidth. Among the possible solutions, we can use the RTCP packets extensions called REMB, which provide the sender with an estimate of the bandwidth. Otherwise, we can implement an adaptive bitrate algorithm, which is the choice we made for our implementation. However, the same problem we encountered in the construction of the pipelines prevents us from even being able to modify the bitrate of the encoders, so we could not test our implementation in order to find the best algorithm.

- **Statistics collection and real-time monitoring:** The architecture of a central server allows to validate this point because the statistics can be gathered in a single entity. For the implementation of the adaptive bitrate, we are able to extract statistics from the *rtpbin* elements of the pipelines.

We can deduce from these points that the model described in Chapter 3 correctly meets the requirements made in the introduction. Nevertheless, it is at the implementation level that the problem lies. Indeed, the difficulties encountered on this point prevented us from obtaining a satisfactory application. As a result, the implementation has some limitations compared to the selected model. The small amount of tests performed prevents us from fully acknowledging our model as the best suited to this problem. More tests need to be done with JPEG XS encoders in order for our solution to be validated. The results however are very positive when they concern a encoder-less SFU, and we do believe it is the best suited architecture for this problem.

6.2 Areas for improvement

In the first time, the obvious improvement would be to complete the intended implementation and to perform more tests on it. The problem we encountered caused us to lose a lot of time and prevents us from bringing this piece of work to a satisfactory closure. We are fairly certain that the issue we encountered might not have been such an obstacle to a more experienced programmer. More features discussed in the theory should also be added to the the implementation in order to make it suited for deployment and commercial use.

The backup solution we found is still employable, but even with the initial model the performance from the point of view of the resources used could be

improved thanks to a better management of the memory in our programs. Indeed, the way we show in Figures 4.5 and 4.6 imply to encode from the beginning a predefined number of maximum admissible clients in the session. And if that number is large, the amount of objects created in the *elements* array is the square of that number. A better way could be to allocate memory dynamically in order to create the pipelines in a better way. All in all, we can say that this implementation may not be ideal to have the best performance that the theoretical model proposes, but it is still a good basis.

As predicted in Chapter 2 and confirmed in Chapter 5, the disadvantage of this architecture is the heavy processing at the SFU. In order to reduce it, a first solution would be to use the actual JPEG XS streamers. Otherwise, other ways of implementing the congestion control in an SFU must be pursued. A solution could be to combine our SFU with the technique of selecting a subset of streams that will not be re-transmitted. In order to effectively reduce processing, the data flow should be stopped before the video is decoded in the SFU. Another way to avoid transcoding could be to communicate the bandwidth estimation found thanks to the REMB packets or the adaptive bitrate algorithm all the way back to the entrypoint of the video stream in the SFU. If the link to the receiving peer has enough capacity for the original signal sent, then this signal could be redirected directly to the user without decoding it, as the transcoding operation doesn't change the bitrate at all in that case. In other words, find a way to bypass the transcoding in the SFU.

Bibliography

[1] Simulcast. http://webrtcglossary.com/simulcast/, September 2014. (accessed 2021-08-11).

[2] SVC. https://webrtcglossary.com/svc/, August 2014. (accessed 2021-08-11).

[3] Foundations. https://gstreamer.freedesktop.org/documentation/application-development/introduction/basics.html?gi-language=c. (accessed 2021-05-21).

[4] The State of Live Streaming in 2021. https://www.streamingmedia.com/Articles/ReadArticle.aspx?ArticleID=146325, April 2021. (accessed 2021-04-28).

[5] Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. (accessed 2021-04-28).

[6] THE STATE OF ONLINE GAMING – 2020. https://www.limelight.com/resources/white-paper/state-of-online-gaming-2020/. (accessed 2021-04-28).

[7] Nina Goetzen. Twitch viewership will grow by 26.2% this year, faster than previously forecast. https://www.businessinsider.com/twitch-viewership-grows-faster-than-previously-forecast-2020-9. (accessed 2021-04-28).

[8] Jean-Yves Lionel Lawson, Jean Vanderdonckt, and Radu-Daniel Vatavu. Mass-Computer Interaction for Thousands of Users and Beyond. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing*

Systems, CHI EA '18, pages 1–6, New York, NY, USA, April 2018. Association for Computing Machinery.

[9] SKEMMI® - Live Crowd Interaction. http://www.skemmi.com/. (accessed 2021-04-29).

[10] intoPIX TICO-XS | JPEG XS IP-cores & SDKs (ISO/IEC 21122). https://www.intopix.com/jpeg-xs. (accessed 2021-04-30).

[11] M. Baldi and Y. Ofek. End-to-end delay analysis of videoconferencing over packet-switched networks. *IEEE/ACM Transactions on Networking*, 8(4):479–492, August 2000.

[12] Christoph Bachhuber, Eckehard Steinbach, Martin Freundl, and Martin Reisslein. On the Minimization of Glass-to-Glass and Glass-to-Algorithm Delay in Video Communication. *IEEE Transactions on Multimedia*, 20(1):238–252, January 2018.

[13] 14:00-17:00. ISO/IEC 15444-1:2019. https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/83/78321.html. (accessed 2021-04-15).

[14] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[15] 14:00-17:00. ISO/IEC 23008-2:2013. https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/54/35424.html. (accessed 2021-04-15).

[16] Get Started with AV1 & AVIF. http://aomedia.org/av1-features/get-started/. (accessed 2021-04-15).

[17] The WebM Project | VP9 Video Codec Summary. https://www.webmproject.org/vp9/. (accessed 2021-04-15).

[18] Thorsten Laude, Yeremia Gunawan Adhisantoso, Jan Voges, Marco Munderloh, and Jörn Ostermann. A Comprehensive Video Codec Comparison. *APSIPA Transactions on Signal and Information Processing*, 8, 2019/ed.

[19] G. Minopoulos, V. A. Memos, K. E. Psannis, and Y. Ishibashi. Comparison of Video Codecs Performance for Real-Time Transmission. In *2020 2nd International Conference on Computer Communication and the Internet (ICCCI)*, pages 110–114, June 2020.

[20] X264, the best H.264/AVC encoder - VideoLAN. https://www.videolan.org/developers/x264.html. (accessed 2021-05-15).

[21] X265, the free H.265/HEVC encoder - VideoLAN. https://www.videolan.org/developers/x265.html. (accessed 2021-05-15).

[22] 14:00-17:00. ISO/IEC 21122-1:2019. https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/45/74535.html. (accessed 2021-04-30).

[23] Antonin Descampe, Joachim Keinert, Thomas Richter, Siegfried Fößel, and Gaël Rouvroy. JPEG XS, a new standard for visually lossless low-latency lightweight image compression. In *Applications of Digital Image Processing XL*, volume 10396, page 103960M. International Society for Optics and Photonics, September 2017.

[24] JPEG - JPEG XS. https://jpeg.org/jpegxs/. (accessed 2021-04-30).

[25] Miroslav Voznak and Michal Halas. Delay Variation Model with RTP Flows Behavior in Accordance with M/D/1 Kendall's Notation. *Advances in Electrical and Electronic Engineering*, 8(5):124–129, May 2011.

[26] V. Jacobson, R. Frederick, S. Casner, and H. Schulzrinne. RTP: A Transport Protocol for Real-Time Applications. https://tools.ietf.org/html/rfc3550, July 2003. (accessed 2021-04-13).

[27] Pyda Srisuresh and Matt Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. https://tools.ietf.org/html/rfc2663. (accessed 2021-05-04).

[28] Jonathan Rosenberg and Christer Holmberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. https://tools.ietf.org/html/rfc8445. (accessed 2021-05-04).

[29] Philip Matthews, Rohan Mahy, Dan Wing, Marc Petit-Huguenin, Jonathan Rosenberg, and Gonzalo Salgueiro. Session Traversal Utilities for NAT (STUN). https://tools.ietf.org/html/rfc8489. (accessed 2021-05-04).

[30] Philip Matthews, Alan Johnston, Jonathan Rosenberg, and Tirumaleswar Reddy. K. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). https://tools.ietf.org/html/rfc8656. (accessed 2021-05-04).

[31] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Congestion Control for Web Real-Time Communication. *IEEE/ACM Transactions on Networking*, 25(5):2629–2642, October 2017.

[32] H. Alvestrand. RTCP message for Receiver Estimated Maximum Bitrate. `https://tools.ietf.org/id/draft-alvestrand-rmcat-remb-03.html`, October 2013. (accessed 2021-05-04).

[33] Magnus Flodman, Erik Sprang, and Stefan Holmer. RTP Extensions for Transport-wide Congestion Control. `https://tools.ietf.org/html/draft-holmer-rmcat-transport-wide-cc-extensions-01`. (accessed 2021-05-04).

[34] Varun Singh. *Protocols and Algorithms for Adaptive Multimedia Systems*. Aalto University, 2015.

[35] Rfc4585. `https://datatracker.ietf.org/doc/html/rfc4585`. (accessed 2021-05-15).

[36] Rfc5109. `https://datatracker.ietf.org/doc/html/rfc5109`. (accessed 2021-05-15).

[37] Kevin M. McNeill, Mingkuan Liu, and Jeffrey J. Rodriguez. An Adaptive Jitter Buffer Play-Out Scheme to Improve VoIP Quality in Wireless Networks. In *MILCOM 2006 - 2006 IEEE Military Communications Conference*, pages 1–5, October 2006.

[38] Magnus Westerlund and Stephan Wenger. RTP Topologies. `https://tools.ietf.org/html/rfc7667`. (accessed 2021-05-04).

[39] Boris Grozev, Lyubomir Marinov, Varun Singh, and Emil Ivov. Last N: Relevance-based selectivity for forwarding video in multimedia conferences. In *Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '15, pages 19–24, New York, NY, USA, March 2015. Association for Computing Machinery.

[40] Boris Grozev, George Politis, Emil Ivov, Thomas Noel, and Varun Singh. Experimental Evaluation of Simulcast for WebRTC. *IEEE Communications Standards Magazine*, 1(2):52–59, 2017.

[41] Stefano Petrangeli, Dries Pauwels, Jeroen van der Hooft, Matúš Žiak, Jürgen Slowack, Tim Wauters, and Filip De Turck. A scalable WebRTC-based framework for remote video collaboration applications. *Multimedia Tools and Applications*, 78(6):7419–7452, March 2019.

[42] Alexandros Eleftheriadis, M. Reha Civanlar, and Ofer Shapiro. Multipoint videoconferencing with scalable video coding. *Journal of Zhejiang University-SCIENCE A*, 7(5):696–705, May 2006.

[43] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the Scalable Video Coding Extension of the H.264/AVC Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120, September 2007.

[44] Philipp Helle, Haricharan Lakshman, Mischa Siekmann, Jan Stegemann, Tobias Hinz, Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. A Scalable Video Coding Extension of HEVC. In *2013 Data Compression Conference*, pages 201–210, March 2013.

[45] Maha Abdallah, Carsten Griwodz, Kuan-Ta Chen, Gwendal Simon, Pin-Chun Wang, and Cheng-Hsin Hsu. Delay-Sensitive Video Computing in the Cloud: A Survey. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 14(3s):54:1–54:29, June 2018.

[46] Gang Peng. CDN: Content Distribution Network. *arXiv:cs/0411069*, November 2004.

[47] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai network: A platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, August 2010.

[48] Konstantin Andreev, Bruce M. Maggs, Adam Meyerson, and Ramesh K. Sitaraman. Designing overlay multicast networks for streaming. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, pages 149–158, New York, NY, USA, June 2003. Association for Computing Machinery.

[49] Matt Calder, Ashley Flavel, Ethan Katz-Bassett, Ratul Mahajan, and Jitendra Padhye. Analyzing the Performance of an Anycast CDN. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 531–537, New York, NY, USA, October 2015. Association for Computing Machinery.

[50] George Politis, Boris Grozev, Paweł Domas, Emil Ivov, and Thomas Noël. Experimental Evaluation of Dynamic Switching between One-on-One and Group Video Calling. In *2018 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–7, October 2018.

[51] Tran Thi Thu Ha, Jinsul Kim, and Jiseung Nam. Design and Deployment of Low-Delay Hybrid CDN–P2P Architecture for Live Video Streaming Over the Web. *Wireless Personal Communications*, 94(3):513–525, June 2017.

[52] Jitsi Videobridge | Video Conferencing for Developers. `https://jitsi.org/jitsi-videobridge/`. (accessed 2021-05-17).

[53] Kurento. `https://www.kurento.org/`. (accessed 2021-05-17).

[54] Janus WebRTC Server: About Janus. `https://janus.conf.meetecho.com/index.html`. (accessed 2021-05-18).

[55] Monitor and Manage your Data When Using Skype. `https://www.xplornet.com/support/getting-started/managing-data-using-skype/`. (accessed 2021-08-18).

[56] Rfc8866. `https://datatracker.ietf.org/doc/html/rfc8866`. (accessed 2021-05-20).

[57] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Rfc3261. `https://datatracker.ietf.org/doc/html/rfc3261`. (accessed 2021-05-20).

[58] Rfc6120. `https://datatracker.ietf.org/doc/html/rfc6120`. (accessed 2021-05-20).

[59] Writing Kurento Applications — Kurento 6.16.1-dev documentation. `https://doc-kurento.readthedocs.io/en/latest/user/writing_applications.html`. (accessed 2021-05-20).

[60] Boris Grozev, George Politis, Emil Ivov, and Thomas Noel. Considerations for deploying a geographically distributed video conferencing system. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 357–361, January 2018.

[61] Jitsi/jicofo. `https://github.com/jitsi/jicofo`, May 2021. (accessed 2021-05-20).

[62] Vamis Xhagjika, Òscar Divorra Escoda, Leandro Navarro, and Vladimir Vlassov. Load and Video Performance Patterns of a Cloud Based WebRTC Architecture. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 739–744, May 2017.

[63] Juan C. Granda, Pelayo Nuño, Francisco J. Suárez, and Daniel F. García. Overlay network based on WebRTC for interactive multimedia communications. In *2015 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pages 1–5, July 2015.

[64] GStreamer: Open source multimedia framework. `https://gstreamer.freedesktop.org/`. (accessed 2021-05-10).

[65] Ws: A Node.js WebSocket library. `https://github.com/websockets/ws`, August 2021. (accessed 2021-08-08).

[66] Samples - Official Kodi Wiki. `https://kodi.wiki/view/Samples`. (accessed 2021-08-22).

[67] trawn3333. Trawn3333/gstreamer_timestamp_marking. `https://github.com/trawn3333/gstreamer_timestamp_marking`, August 2021. (accessed 2021-08-22).

Appendix A

Graphics of the implemented pipelines

A.1 Full pipelines of the implementation

A.1.1 Sending pipeline of the client

This is the pipeline built in our implementation for sending the video from the client to the SFU. Here, the video is given by a test source, but it could be any camera or video file needed for the application. We can see how the *rtpbin* sends one RTP streams, one RTCP stream and receives one RTCP stream.

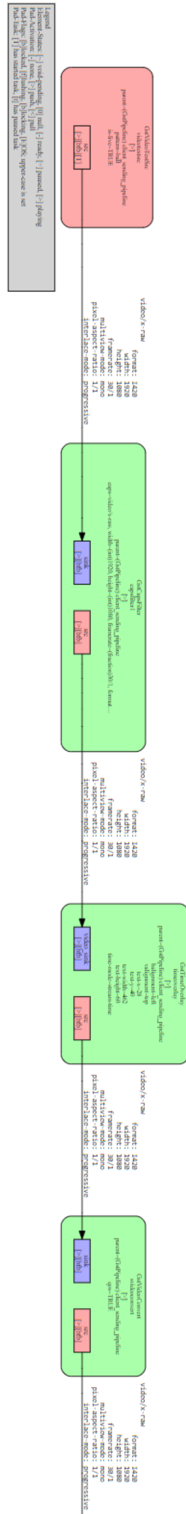


Figure A.1: Sending pipeline of the client, part 1

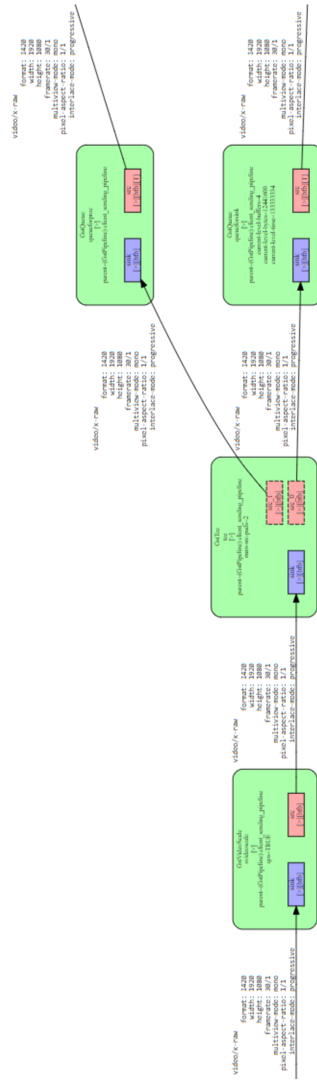


Figure A.2: Sending pipeline of the client, part 2

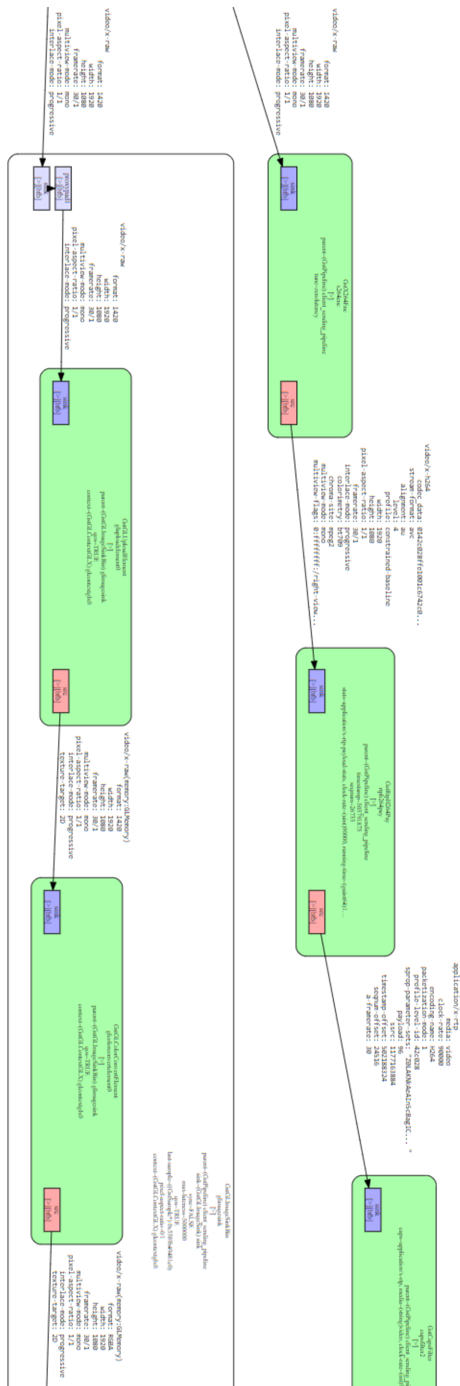


Figure A.3: Sending pipeline of the client, part 3

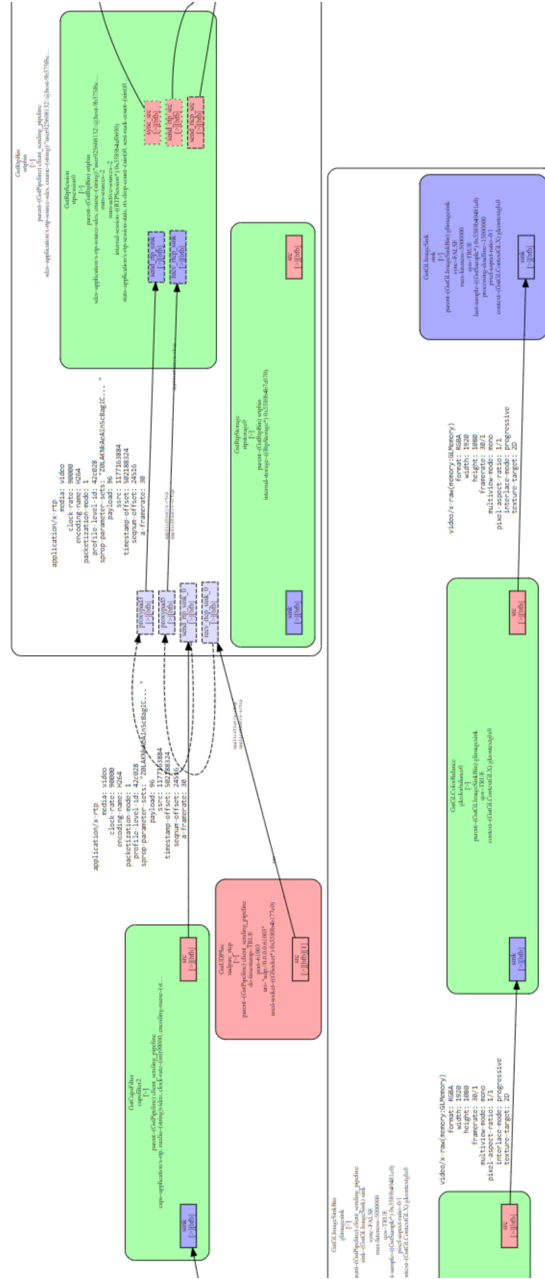


Figure A.4: Sending pipeline of the client, part 4

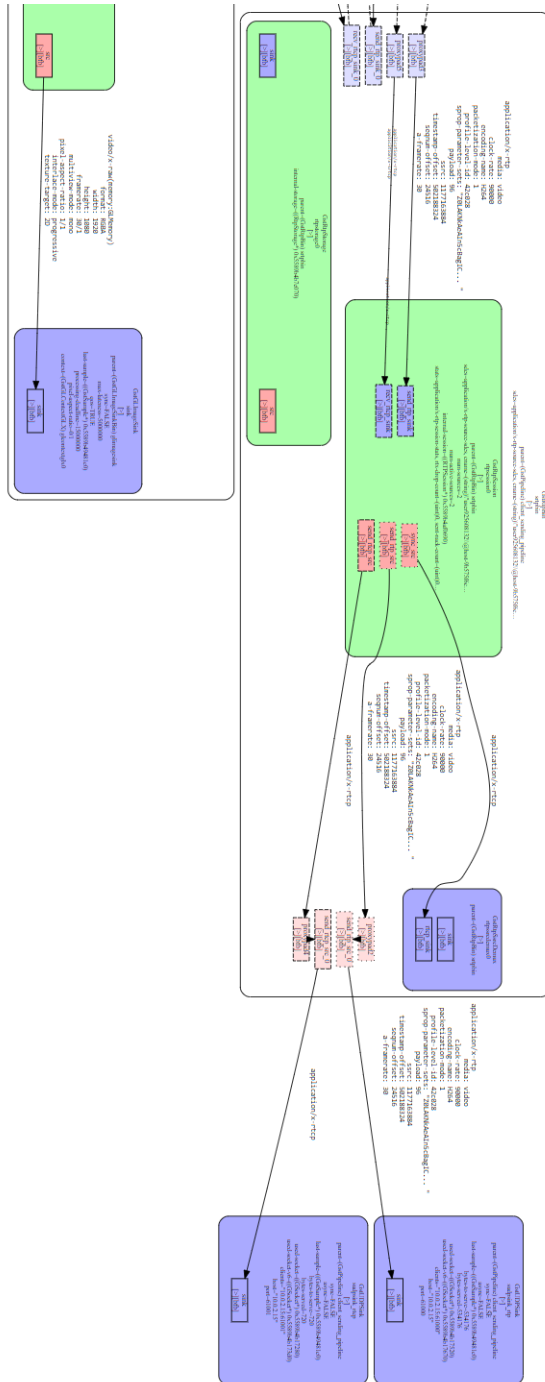


Figure A.5: Sending pipeline of the client, part 5

A.1.2 Receiving pipeline of the client

This is the pipeline the client uses to receive a video stream coming from the SFU and to play the video. Here, we can see the difference between a sending *rtpbin* from the previous Section and a receiving *rtpbin* hereunder. The latter contains notably a jitter buffer element, as well as a demuxer if multiplexing is performed.

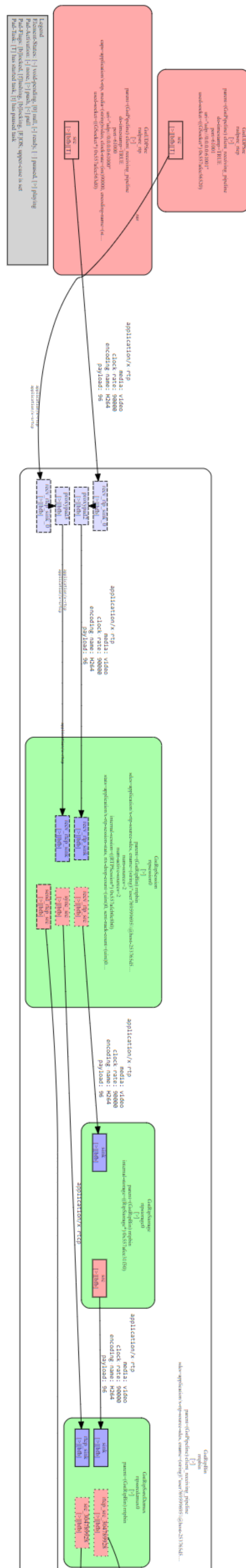


Figure A.6: Receiving pipeline of the client, part 1

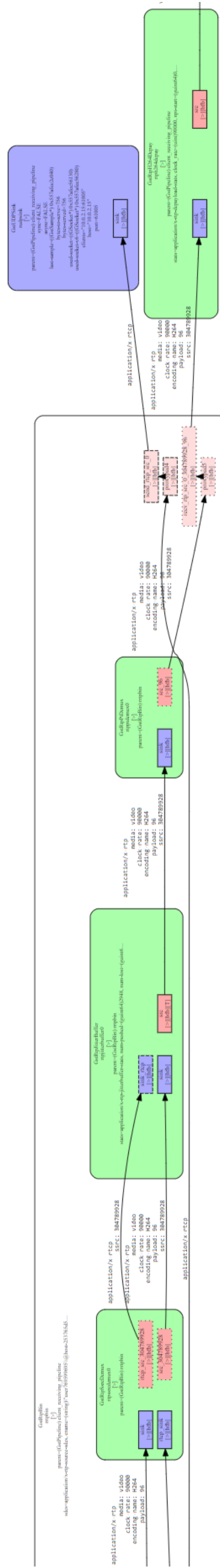


Figure A.7: Receiving pipeline of the client, part 2

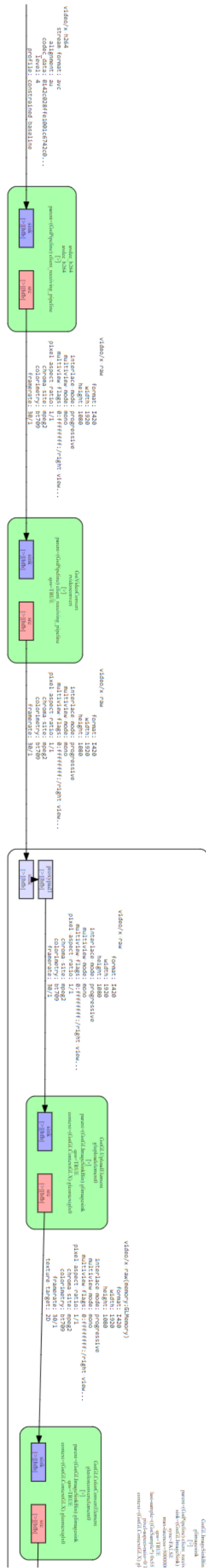


Figure A.8: Receiving pipeline of the client, part 3

Figure A.9: Receiving pipeline of the client, part 4

A.1.3 Pipeline in the SFU, with encoders

This is the pipeline inside the transcoding SFU in case of a session between 2 clients.

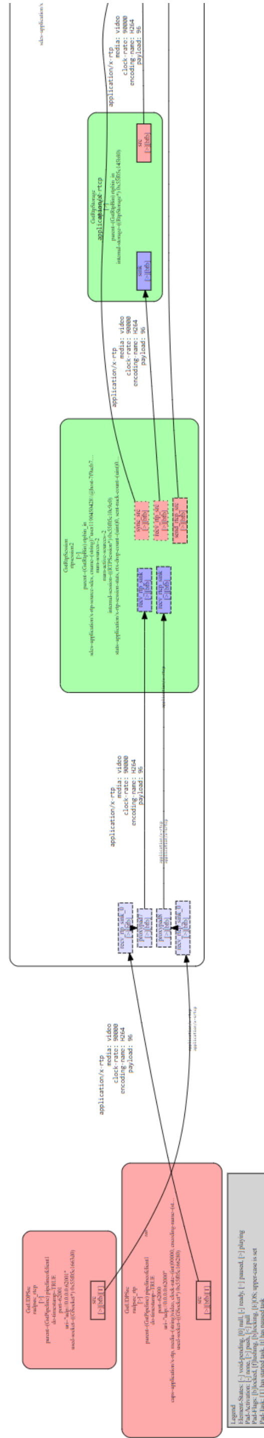


Figure A.10: Pipeline inside the SFU, with encoders, part 1

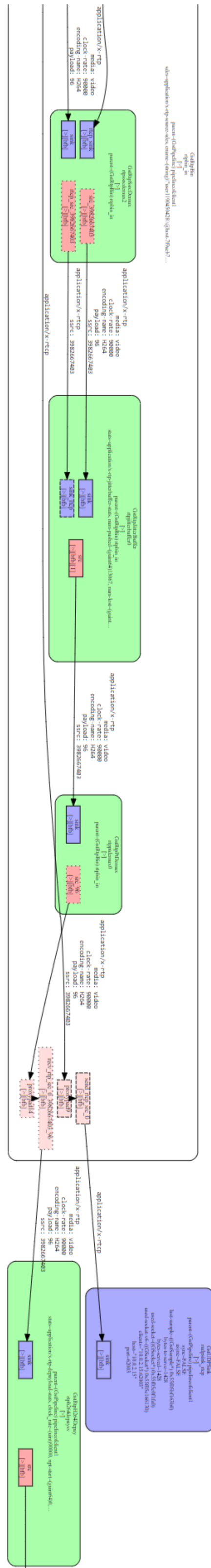


Figure A.11: Pipeline inside the SFU, with encoders, part 2

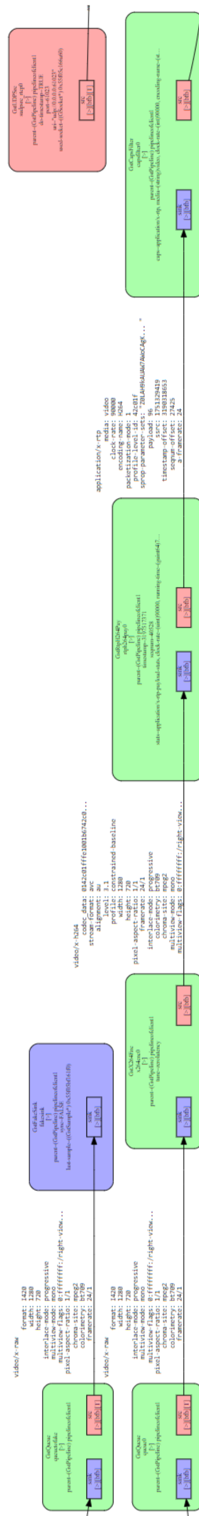


Figure A.12: Pipeline inside the SFU, with encoders, part 3

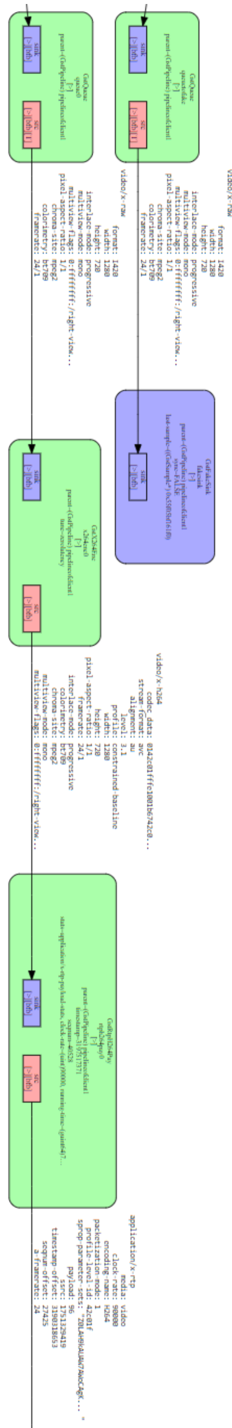


Figure A.13: Pipeline inside the SFU, with encoders, part 4

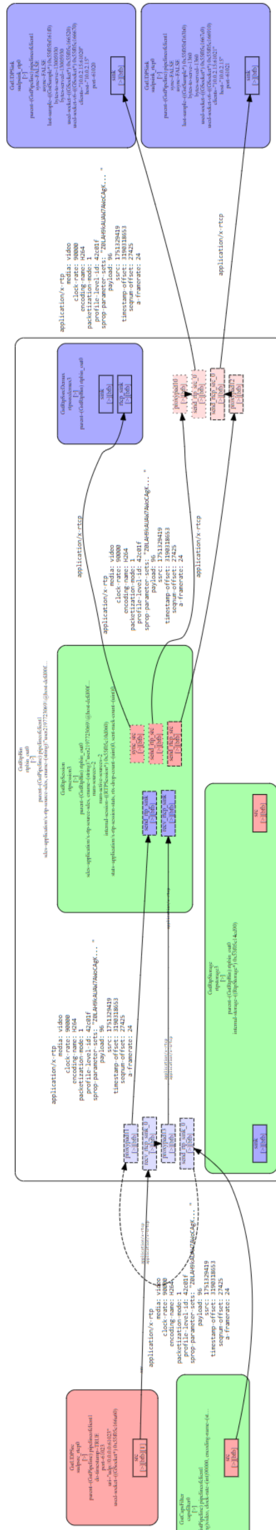


Figure A.14: Pipeline inside the SFU, with encoders, part 5

A.2 Pipelines used in measurements

A.2.1 CPU measurements

A.2.2 Latency measurements

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**