LOW LATENCY ADAPTIVE VIDEO ENCODING

Andi Domi
Student number: 41259

Master of Science Thesis
Supervisor: Sébastien Lafond
Second Supervisor: Annamari Soini
Åbo Akademi University
Faculty of Science and Engineering
Embedded Systems Laboratory
May 2019

ABSTRACT

This thesis presents a prototype method for streaming video data over large networks implemented in the high level programming language Python. The objective is to develop a software which would allow video footage to be streamed from a remote drone to a controller with a latency of less than 200 milliseconds and which would be able to deliver a satisfactory image quality in relation to the available bandwidth through different control algorithms and techniques. The proposed solution decreases the possible delays between the controller and the drone, making the navigation more secure while providing better maneuverability. The thesis details the state of known technologies implemented for the same purpose and conducts research on their advantages and disadvantages, while using these technologies as the base to develop a solution by removing some of the disadvantages encountered. In this work different methods were used to increase the coverage of the bandwidth by dynamically allocating the video bitrate to the given bandwidth level using the GStreamer framework implemented in a small-board computer, an Nvidia TX2, as the main mechanism for the purpose. Consequently two adaptive bitrate algorithms were created and implemented to tackle this problem. Finally the solution was tested under two custom bandwidth functions with the help of the ROBOT framework and a Linksys DIR-809 router.

The analysis of the results shows that achieving real time video streaming with variable bitrate depending on the bandwidth level is possible. Moreover, these results also detail the inability of the two algorithms to achieve on their own a satisfactory bitrate and streaming quality in relation to the different bandwidth behavior, making the usage of the two conjoined algorithms a possible better solution.

Keywords: Adaptive Bitrate, Video Streaming, Low Latency, Embedded Board

CONTENTS

LIST OF FIGURES

ACRONYMS

UAV Unmanned aerial vehicle

ABR Adaptive Bitrate Streaming

SVC Scalable Video Coding

RTP Real Time Protocol

UDP User Datagram Protocol

RTPC Real Time Control Protocol

TCP Transmission Control Protocol

ISP Internet Service Provider

FPS Frame per second

HEVC High-Efficiency Video Coding

HTTP Hyper Text Transfer Protocol

URL Uniform Resource Locator

DRM Digital Rights Management

AES Advanced Encryption Standard

HTTPS Hyper Text Transfer Protocol Secure

QoS Quality of Service

CNAME Canonical Name

SSRC Synchronization Source Identifier

PT Payload Type

SN Sequence Number

ML Machine Learning

MSS Maximum Segment Size

SSTHRESH Slow Start Threshold

RWND Receiver's Window

API Application Programming Interface

MSE Mean Square Error

PSTNR Peak Signal-To-Noise Ratio

SSIM Structural Similarity

PAE Peak Absolute Error

RMSE Root Mean Square Error

1 INTRODUCTION

1.1 Field of interest

Drones, also called unmanned aerial vehicles (UAVs), are vehicles with no human pilot on board which are either controlled remotely by an operator or are self-autonomous via computer software. In today's world, they are becoming progressively popular not just for military purposes, where their services have been used for years, but also for a wide range of different operations. They are used by scientists to collect data in dangerous environments where it would be otherwise impossible for humans to collect the information needed and where satellites cannot provide enough details for the given purpose. They are deployed for the study of hurricanes with the use of swarm technology implemented in drones to help create a model about the complex flow of a hurricane, or in law enforcement, where they are used to survey possible suspects' locations and to create a plan of action in case of a hostile situation. However, they are also used for commercial purposes, with drone delivery currently being tested in several locations giving impressive results in light of them being totally autonomous. Also, drones are used for recreational purposes like drone racing, capturing live events, and a myriad of other applications. One thing that brings together the diverse uses of drones is their ability to navigate through different landscapes without the need for an onboard pilot making them extremely versatile and safe. Remote controller navigation is achieved by streaming visual footage of the drone location to the operator or to a software which then computes the needed data through different video processing technologies. This navigation technology uses a direct linked connection between the operator and the drone, without third-party software or hardware, so it would be able to receive and send data between the two. This wireless connection is normally held over a 2.4 and 5.8 GHz WiFi frequency band [9] [10] which provides a low latency video output of the drone video footage which is directly correlated to a low latency navigation input by the operator, two features with an extremely high priority. Having

a direct linked connection also brings some disadvantages from which we can mention a limited connection range, where the operator should normally stay in a range of a maximum of 92 meters [9] [10] from the drone to be able to maintain a stable connection. With the increasing usage and different purposes, we see the need for a method to remotely control the drones over larger distances. As a result, drone navigation technology is shifting from end-to-end connection to a larger and broader network, the internet. While this solves the main problem of remotely controlling a drone over a wider distance, this also brings more challenges, such as the increase in the latency of the connection, and unpredictable bandwidth which varies greatly over time. This alteration in the connection latency is translated to an increase in the response time of the drone to the operator input but also an increase in drone video footage output, which increases the general latency of the navigation of the drone with unpredictable results. While in some autonomous vehicles low latency may not be of a high priority, in other implementations like autonomous cars and drone racing it is extremely important to have a fast response to the different environmental variables as part of the navigation system. The remote operation of the drones is divided into two parts, one of which is remotely controlling the drone according to the input of the pilot, but for the purpose of this thesis we will focus more on the streaming of the video footage from the drone to the end user side of the navigation.

1.2 Streaming over internet

For us to be able to stream multimedia data from a moving object over unstable computer networks, such as over the internet, we need to adapt the bitrate of the video footage to the available bandwidth level conditions. This is done using a technique called Adaptive Bitrate Streaming (ABR). ABR is a method used in streaming multimedia data over computer networks to adopt the video quality to the available bandwidth conditions. These algorithms can run either on the operator (client) side or on a drone (server) and by monitoring different variables such as packets drop, latency, bandwidth, CPU load etc. They take adequate measures to alter the bitrate of the video stream. The majority of ABR algorithm streams require the use of an encoder and decoder which can encode a single video source into multiple streams of different bitrates, then each stream is divided into smaller segments, usually between 2 to 10 seconds. The client, in the beginning, is made aware of the available bitrates with the

help of a special file, and after assessing the download speed of each segment, usually starting from the segment present in the lowest bitrate, compares it with the bitrate of the video source. Based on the difference between the two it requests segments from either the next higher bitrate in case of the difference being small, or a lower bitrate in case of the difference being bigger as can be seen in Figure 1.1.

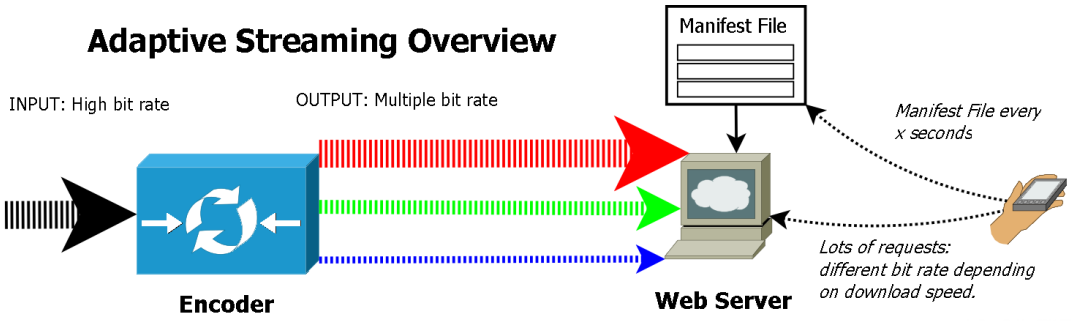

Figure 1.1: Adaptive Bitrate Streaming overview [1]

Different technologies have already implemented several versions of adaptive video streaming like MPEG-Dash, HLS, SVC, RTP etc. but they have their limitations. The majority of them were created to tackle another field of video streaming, on-demand video streaming, as per 2018 almost 58% of all downstream traffic on the internet is video data with streaming giants like Netflix and YouTube holding the top two of the three spots for global application traffic share [11].

The currently implemented technologies are based on almost the same principle. They divide the video stream in chunks of different quality attributes and then, depending on the bandwidth level of the connection, select the most appropriate quality for the given connection and proceed to the streaming phase. Some use complex algorithms which divide a high-quality video stream in different data layers, with each layer holding certain parts of the quality, and then try to deliver as many layers as possible to the client so that when they arrive and cluster on top of each other they are able to improve the quality of the video proportionally to the layers received, as seen in the Scalable Video Coding (SVC) implementation [12]. Another approach is to have a specific type of packets which holds information about the video stream sent to the client. Upon receiving this particular packet the client sends back to the server detailed statistics which are in turn used by it to adapt the video quality to the connection. This method is used in Real Time Protocol (RTP) implementation technology. RTP typically wraps User Datagram Protocol (UDP) packets [13], assigning each of them with a unique

identifying number to be able to retrieve statistics about the video stream. The packets are then sent to the client for statistical analysis by counting the packet identifiers and the time of delivery of each packet. The RTP protocol is almost always conjoined with Real Time Control Protocol (RTCP) to retrieve the statistical analysis. RTCP works by creating a Transmission Control Protocol (TCP) connection to be able to retrieve the statistical information packets from the client.

While these technologies have individual benefits they also have their limitations, linked to the specific field they are applied to when tackling the specific problems they are used for. For example, MPEG-dash tries to tackle the scalability problem [14] and because of that, it creates several chunks of video data with different bitrates for the same given video file, which in turns breaks the bandwidth interval to quality ration into ten parts not providing a satisfactory quality for the available bandwidth. Other advanced technologies like SVC are not widespread for their difficulty to be implemented and for the lack of support for HEVC. Lastly, protocol specific implementations like RTP are not well accepted by some Internet Service Providers (ISP), which puts the packets behind a firewall not allowing them to be sent to the end user.

1.3 Objectives of the thesis

The objective of this Master's thesis is to implement and assess a prototype method to adapt a video stream between the server and the client in such a way that it will maximize the quality of the video stream footage depending on the available network bandwidth by maintaining a real-time low latency of fewer than 200 milliseconds under the highest possible resolution of 3840x2160 pixels (4K) and a frame rate of a minimum of 30 frames per second (FPS). This is done by using the state of the art video compression codec, HEVC (High-Efficiency Video Coding), also known as H.265, implemented on an embedded board with the hardware capability to compute the video footage at the given latency. The embedded board also provides a custom-built encoder named "envenc" to encode the video footage under the non-functional requirements of this thesis [15]. This thesis takes inspiration from the concepts implemented in RTP and RTPC video streaming protocols in a high-level programming language such as Python. This is done to use any transport protocol for data delivery in order to be able to send the video stream over several network filters, like firewalls, by keeping to a minimum the non-data packets sent across the network. The result is a more efficient

usage of the bandwidth by not encapsulating the data packets under any wrapper but by streaming unadulterated UDP packets. The solution we are proposing is implemented by having two separate synchronized threads on the server and the client side, one of which prepares and streams the video data and the other one which computes the statistical analysis of the packets and computes the available bandwidth and the bitrate on which the encoder should operate upon. The thread responsible for the statistical analysis of the packets creates a report of the bandwidth by counting the number of packets sent and received from the transmitter to the receiver and then opens TCP stream for the statistical packets to be delivered. Depending on the difference between the number of packets sent and received we can create a general understanding of the maximum bandwidth level of the network. Different techniques are used to ensure the delivery of the data packets responsible for the statistical analysis and the synchronization of the threads between the server and the client. Also, two different custom ABR's are implemented to improve the streaming performance to handle different bandwidth level by giving the maximum bitrate to bandwidth ratio, which results in better perceived video quality. These algorithms are then put under several rigorous tests analyzing their performance down to the pixel levels of the streamed video frames, with the help of specialized algorithm which try to emulate human vision by giving a quantifiable score to the video streaming quality, a score which is comparable to the average perceived quality by a human being.

1.4 Thesis structure

Chapter 2 provides background knowledge on the definition of ABR's and their different implementations. We take a closer look at each implementation, its purpose, the mechanism behind its implementation, and the different advantages and disadvantages each of them offers.

In Chapter 3 we provide our own implementation for the ABR. Firstly, we analyze the concept behind the software from a high-level architecture perspective, and how the non-functional requirements are implemented into code. Then we analyze and explain the different algorithms used as the core of our ABR implementation and try to make assumptions on their behavior under different bandwidth levels.

Chapter 4 presents the testing ground for the software. We explain how we designed our testing environment for the proposed solution and the different technolo-

gies used. Also, we give a clear representation of the bandwidth levels our proposed solution will be tested upon so we can analyze the different algorithms. We try to keep this evaluation as objective as possible using several methods to evaluate the overall streaming performance of the video by tools well accepted by the research community.

In Chapter 5 we give the evaluation for the different experiments conducted in this thesis. Last but not least, we explain the next possible step in the implementation of the proposed solution by proposing a clear overview of future work.

2 Related Work

This chapter introduces the concept of ABR in regard to different implementations. Conventional video streaming methods commonly deliver a fixed bitrate of the video stream, therefore not giving the best video quality for the given bandwidth. While a different type of ABR attempts to send a satisfactory bitrate for the amount of bandwidth, its implementation is difficult and requires additional encoders and protocols to encapsulate the stream. This encapsulation may result in a sub-optimal streaming experience as the protocols may not be accepted by the different layers of the network.

2.1 What is Adaptive bitrate

Adaptive Bitrate Streaming is a technique used in streaming multimedia over computer networks to adapt the video quality to the available bandwidth conditions. This algorithm can run either on the server or on the client side, and monitors different variables such as packets drops, latency, bandwidth, CPU load, etc. to compute the output bitrate for the given maximum bandwidth level of the video stream. The majority of the ABR algorithms require the use of an encoder and decoder which can encode a single video source into multiple bitrates, then each bitrate is divided into smaller segments. At the beginning, the client is made aware of different available bitrates with the help of a manifest file, and after assessing the download speed of each segment, usually starting from the one present in the lowest bitrate, compares it with the bitrate of the video source. Based on the difference between the two packet counts it requests segments from either the next higher bitrate in case of the difference being small, or a lower bitrate in case of the difference being bigger, as can be seen in the Figure 2.1.

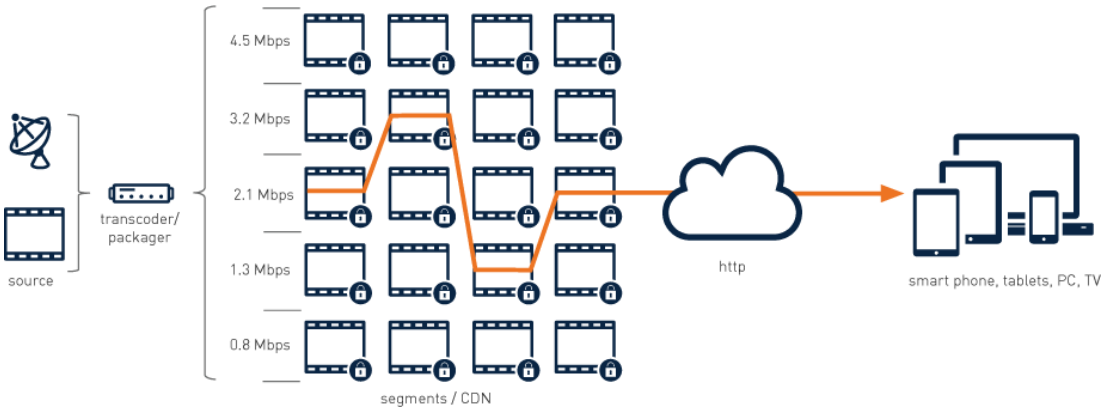

Figure 2.1: Adaptive streaming in action. [2]

2.2 Different Implementations

Different implementation solutions are available such as HLS, MPEG-DASH, Adobe HTTP Dynamic Streaming, Microsoft smooth streaming etc. All these adaptive streaming technologies follow almost the same core principle [16]. They generate different versions of a video file in different quality, spatial resolution etc. and then divide these versions into segments, usually between 2 to 10 seconds, which are then provided by a web server through a Hypertext Transfer Protocol (HTTP) request to the client. The information of different versions and relationships between them is encapsulated in the manifest file which is provided to the client at the very beginning of the streaming session. The manifest file describes all the different qualities of the media content and the individual segments of each quality with HTTP Uniform Resource Locators (URLs). This structure provides the binding of the segments to the bitrate among other things , e.g., start time, duration of segments etc. As a consequence, each client will first request the manifest that contains the temporal and structural information for the media content, and based on that information, it will start requesting the individual segments that fit best the current bandwidth level. The adaption of the quality or spatial resolution to the bandwidth is done on the transmitter side for each segment. Depending on the available bandwidth of the client, the server can switch to a higher bitrate segment or a lower one if the bandwidth decreases.

2.2.1 MPEG-DASH

MPEG-DASH is the only adaptive bitrate HTTP-based streaming solution adopted as an international standard. To be able to play the content, the client obtains first an MPD file which can be acquired from different transport routes such as HTTP, email, thumb drive, etc. The file is then parsed to be able to learn the program timing, resolution, minimum and maximum bandwidth, media types, media-content availability, and also the existence of various encoded alternatives of multimedia components, digital rights management (DRM), media-component locations on the network, and other characteristics.

With the information provided, the client selects the appropriate encoded stream and then starts it by fetching the segments using HTTP GET requests. After the receiver has pre-loaded the stream data into a buffer, it continues fetching the subsequent segments of the video while monitoring fluctuations in the bandwidth. Depending on its throughput, it can then decide to adapt the video stream to match the available bandwidth by fetching segments of different bitrates as presented in Figure 2.2.

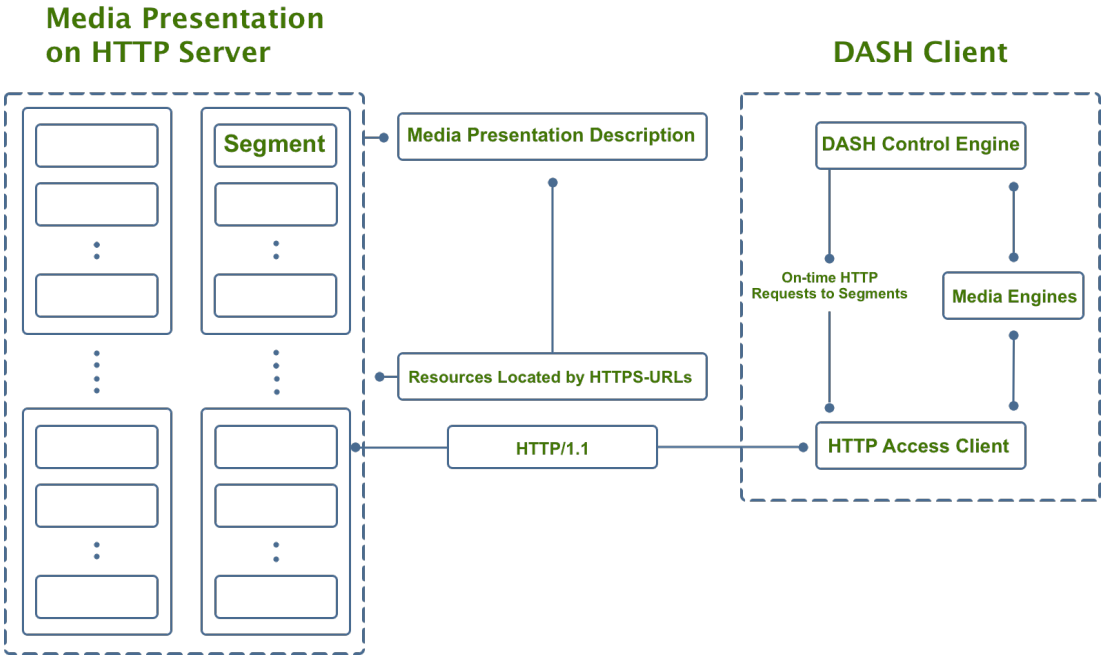

Figure 2.2: MPEG-DASH Scope. [3]

2.2.2 HLS

HTTP Live Streaming, also known as HLS, is a widely used adaptive HTTP-based streaming protocol available as IETF Internet Draft and implemented and patented by APPLE Inc. as part of its software suite. In its core, HLS works like all adaptive streaming technologies. It encodes the source into multiple files at different data rates and it divides them into short segments, usually 5-10 seconds long. These are loaded onto an HTTP server along with a manifest file, with the extension "M3U8" that provides information to the client about additional manifest files as shown in Figure 2.3.

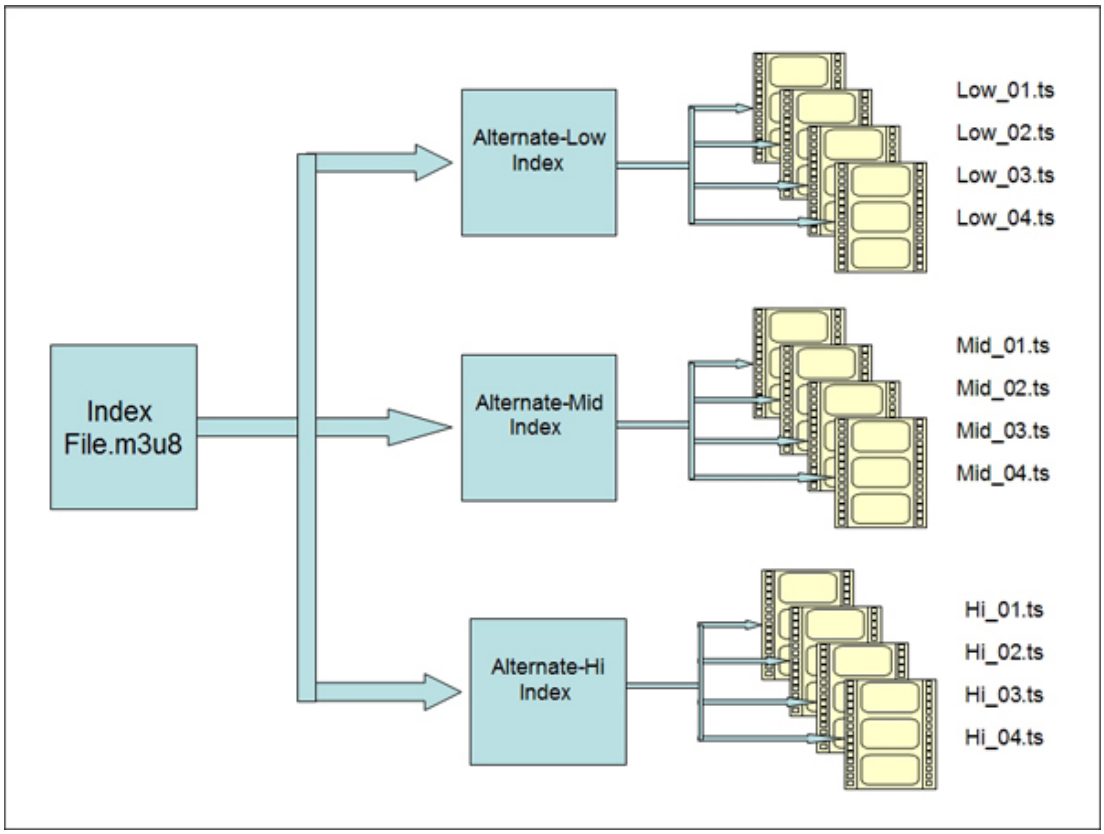

Figure 2.3: HLS Scope. [4]

The encoding protocol used is normally Advanced Video Coding, also known as H.264 for video, and MP3, AAC, EC-3, or AC-3 for audio. The video/audio stream must be segmented in a MPEG-2 Transport stream with an extension of "ts". The files are then deployed into an HTTP server. The .M3U8 manifest files are constantly updated with the location of different available streams and file chunks. The client then

requests and downloads the file resources, assembling them into a continuous flow video. The client first downloads the index file through a URL and then the several available media files. Then it assembles the sequences to allow the continued display to the user. HLS is equipped with Advanced Encryption Standard (AES) encryption mechanism and a secure-key distribution method, using Hypertext Transfer Protocol Secure (HTTPS) which uses either an HTTP cookie or device-specific authentication, two methods which combined provide a digital rights management system.

2.2.3 SVC

SVC is an extension of the H.264/MPEG-4 AVC standard providing scalability at a bitstream level with a considerable increase in decoder complexity relative to single-layer H.264/MPEG-4 AVC. The SVC codec adapts to sub-par network connections by dropping bitstream subsets or packets in order to reduce the frame rate, bandwidth overload, and resolution of a picture, which prevents the picture from losing the integrity by not being able to be properly formed. For example, a receiver would receive only the base layer or bitstream if the bandwidth level is low, while a high bandwidth client would receive both, the base layer and bitstream subset or enhancement layer, resulting in a higher quality video. This is possible because a subset bitstream can signify a smaller screen (lower spatial resolution), or a lower frame rate (lower temporal resolution), or a lower quality video signal to the bitstream it is derived from as illustrated in Figure 2.4.

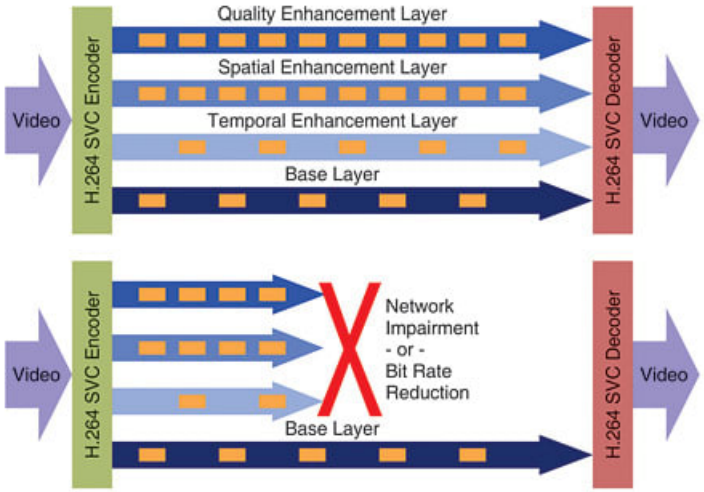

Figure 2.4: SVC Layers. [5]

The following modalities are possible:

- Temporal scalability: refers to the possibility to represent video streams with different frame rates by numerous bitstreams.
- Spatial (picture size) scalability: the video is coded in multiple resolutions. Videos of higher resolutions are responsible for enhancing the lower resolution layer through prediction.
- Fidelity scalability: the video is coded at a single spatial resolution but at different qualities. The data and decoded samples of lower qualities can be used to predict data or samples of higher qualities in order to reduce the bit rate to code the higher qualities.
- Combined scalability: a combination of the three scalability modalities described above.

  SVC enables forward compatibility for older hardware: the same bitstream can be consumed by basic hardware which can only decode a low-resolution subset (i.e. 720p or 1080i), while more advanced hardware will be able to decode high-quality video stream (1080p).

2.2.4 RTP

RTP is a real-time end-to-end transport protocol for delivering audio and video over the network. RTP works by encapsulating UDP packets on its own data wrapper packet. UDP, which is also a transport protocol in its own, is a connection less transport layer protocol which allows sending packets (datagrams) to other hosts without guaranteeing the delivery, ordering, or duplicate protection of the packets. For this reason, RTP adds real end-to-end delivery services on UDP, service that include sequence numbering, time-stamping, and delivery monitoring as part of its payload identification. The minimum header size of RTP is 12 bytes, with a maximum size of 76, depending on the optional fields used. [17]

RTP itself does not provide any means to ensure Quality of Service (QoS), timely delivery, or sequenced packet as it does not assume the network to be reliable and deliver packets in sequence, as can be seen in the protocol's header scheme Figure 2.5.

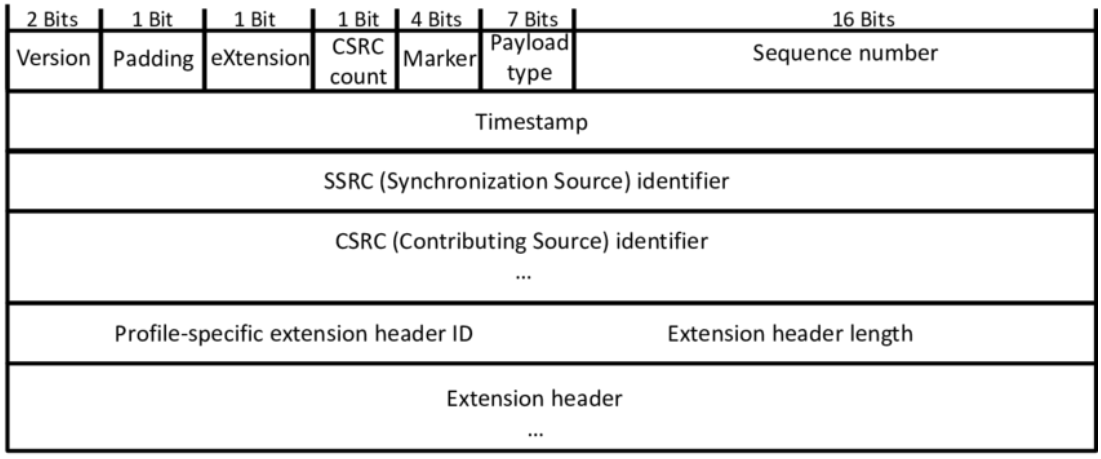

Figure 2.5: RTP header and extension. [6]

For this reason, RTP is normally used together with another protocol called RTCP, to monitor the quality of service and to convey information about the participants in an on-going session. This latter aspect of RTCP may be sufficient for "loosely controlled" sessions, i.e., where there is no explicit membership control and set-up, but it is not necessarily intended to support all of an application's control communication requirements.

RTCP is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets.

RTCP performs four functions:

1. The primary function is to provide feedback on the QoS of the data distributed.

2. RTCP carries a constant transport-level identifier called the canonical name (CNAME). The Synchronization Source Identifier (SSRC) may change because of a conflict and because of that, the receivers use CNAME to keep track of the different participants.

3. It is required from the first two functions that all the participants send RTCP packets. For RTP to be able to scale up and reach a large number of participants the rate must be controlled.

4. The fourth function is to rely on minimal information for session control. Because of that, there is a limit of 5 seconds by which time the rate of the session control information will be sent. Using a high bandwidth rate this limit may be lower, but this is not advised.

RTCP provides a way to connect and synchronize different media streams that have come from the same sender. This is done by providing the necessary details of

the connection, such as statistics or control information, using a dedicated channel, also called Out-of-band, to send such information as illustrated in Figure 2.6.

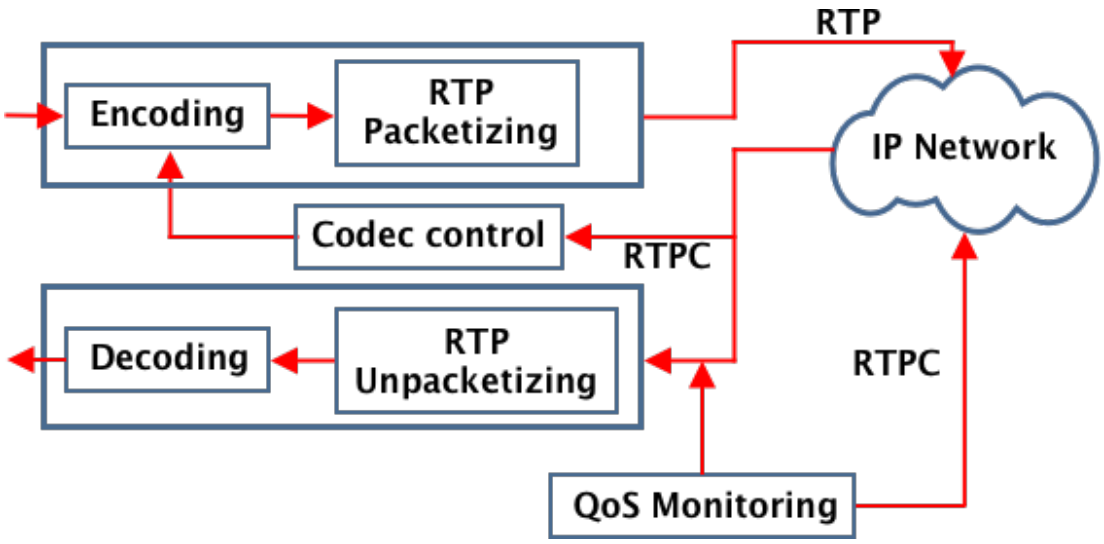

Figure 2.6: RTPC Scope. [7]

2.3 Drawbacks of the existing technologies

While these streaming technologies are the current state of the art for ABR, they all have their own drawbacks. ABR technologies which are based on HTTP are more complex than other adaptive bit rate technologies.

Another area where HTTP-based adaptive streaming solutions are not optimized is the ability to have the most optimal bitrate for the available bandwidth, because of the division of the video source into fixed stream quality.

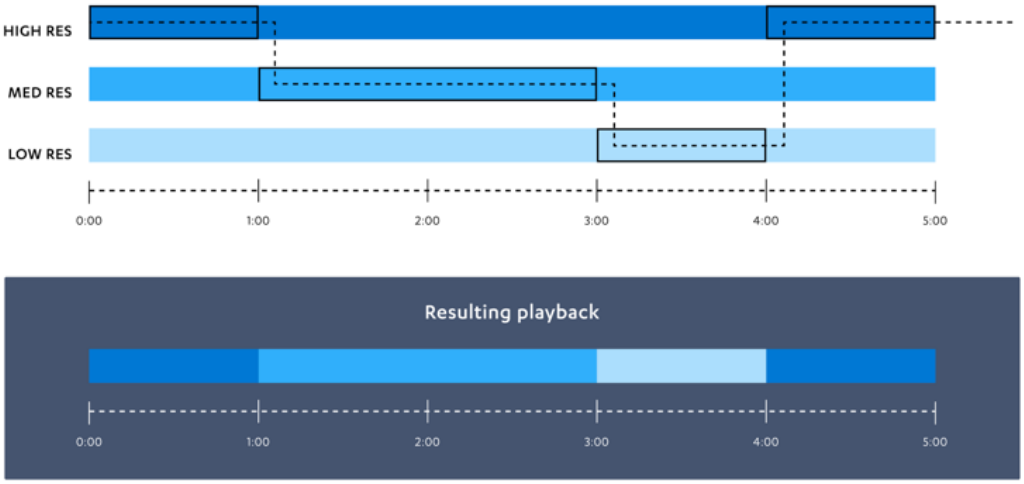

Figure 2.7: The normal response of ABR algorithms under variable bandwidth [2]

As shown in Figure 2.7, the quality of the video will not change if the bandwidth level is in between the three thresholds of the bandwidth. This means that a large segment of the bandwidth capable of delivering a higher bitrate is not utilized.

An even more complex and new implementation like SVC has a poor performance delivering a satisfactory bitrate for the available bandwidth level, as its implementation is really complex, and as to date there is no consolidated implementation of SVC technology for the H.265 video compression protocol, which would, in theory, decrease the bandwidth usage while maintaining the same bitrate level as another type of encoding.

Also, segmenting a stream in different layers with several types of bitrate and qualities results in an increase in the required computational capacity. While this may not be of a high priority for non-live video streaming where this segmentation can be done prior to the streaming, this is not a feasible option when the resources at our disposal are restricted, for example in a drone, where navigation is done by live streaming the drone footage.

Another issue with HTTP streaming implementations is the management of the DMRs as there is no universal way of delivering content which is time-sensitive or restricted. With no single clearly defined or open standard for the digital rights management used in the above methods, there is no compatible way of delivering restricted or time-sensitive content to all the participants.

Last but not least, issues also exist with RTP or other non-HTTP-based streaming implementations. Normally, streaming to different participants is done behind different

layers of networks, with each layer having a different node. Each node in the network may put a filter to discard RTP packets, which will make the transportation of the packets difficult. This is fairly seen in internet mobile network technologies which will be integrated into the drone to have access to the internet. RTP permits omitting different fields in the header, which allows the creation of a header shorter than 12 bytes; a minimal version of RTP of two bytes can be constructed, only containing a payload type (PT) and a sequence number (SN). This choice may come with some disadvantages from the which we can mention the difficulty to synchronize audio and video (cross-media synchronization) if we omit the time-stamp field. Also, a great part of the RTCP functionality would have to be altered as it depends on the time-stamp header and a long sequence number field for loss-statistics jitter computation and synchronization [18].

3 PROPOSED SOLUTION

As mentioned in Chapter 2, RTP is not used extensively in today's streaming services because of the restricting network filters which may be applied by the servers but also for the overhead RTCP adds to the stream for QoS analysis. For this reason, we decided to build an implementation of RTCP while still using RTP as the base for the streaming protocol. We will still be using RTP as the main streaming protocol for testing our implementation of RTCP because RTP is already a consolidated platform for video streaming and can be well implemented in our solution. Further in Chapter 5 we detail a new way in which we can remove RTP from our proposed solution by using UDP with specialized markers which will behave as RTP QoS analysis tags. Our solution will also be protocol agnostic. This is another reason we are trying to implement it on RTP to show that a future implementation of this solution with an unspecified protocol is possible.

3.1 Overview of the solution approach

Each RTP packet has a header, for encapsulating UDP packets in its protocol. Through this encapsulation, with the help of RTCP, we are able to have QoS over our desired stream. Our implementation uses the same approach as RTCP but without relying on RTP and RTCP packet headers. To remove the RTP packets header, and therefore the packet header of RTCP, using only UDP packets while still maintaining the QoS analysis, we are counting the packets on the source and destination side.

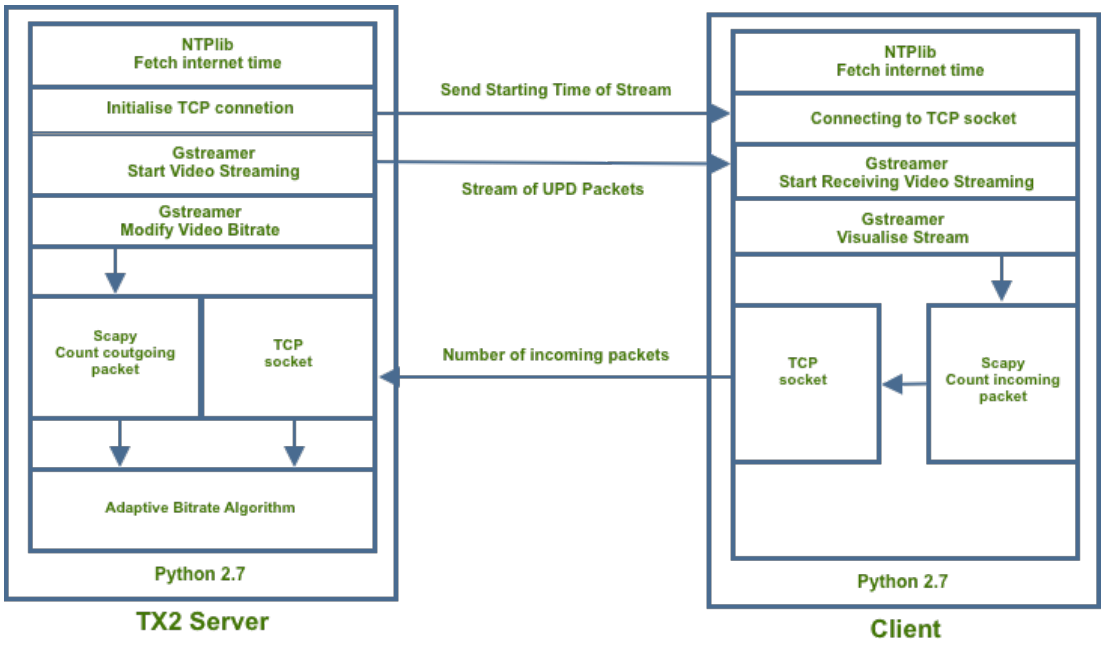

Figure 3.1: Software architecture

The high-level representation of the proposed architecture is shown in Figure 3.1. The solution is implemented in a Linux environment where the server is an embedded Nvidia TX2 board and the client a generic PC. The packet counting mechanism is achieved by using a Python library named SCAPY[19], implemented on both, the server and client side, and this counting mechanism analyzes a specific port where the video stream is broadcast to count the outgoing and incoming packages as seen in the sequence diagram Figure 3.2 .

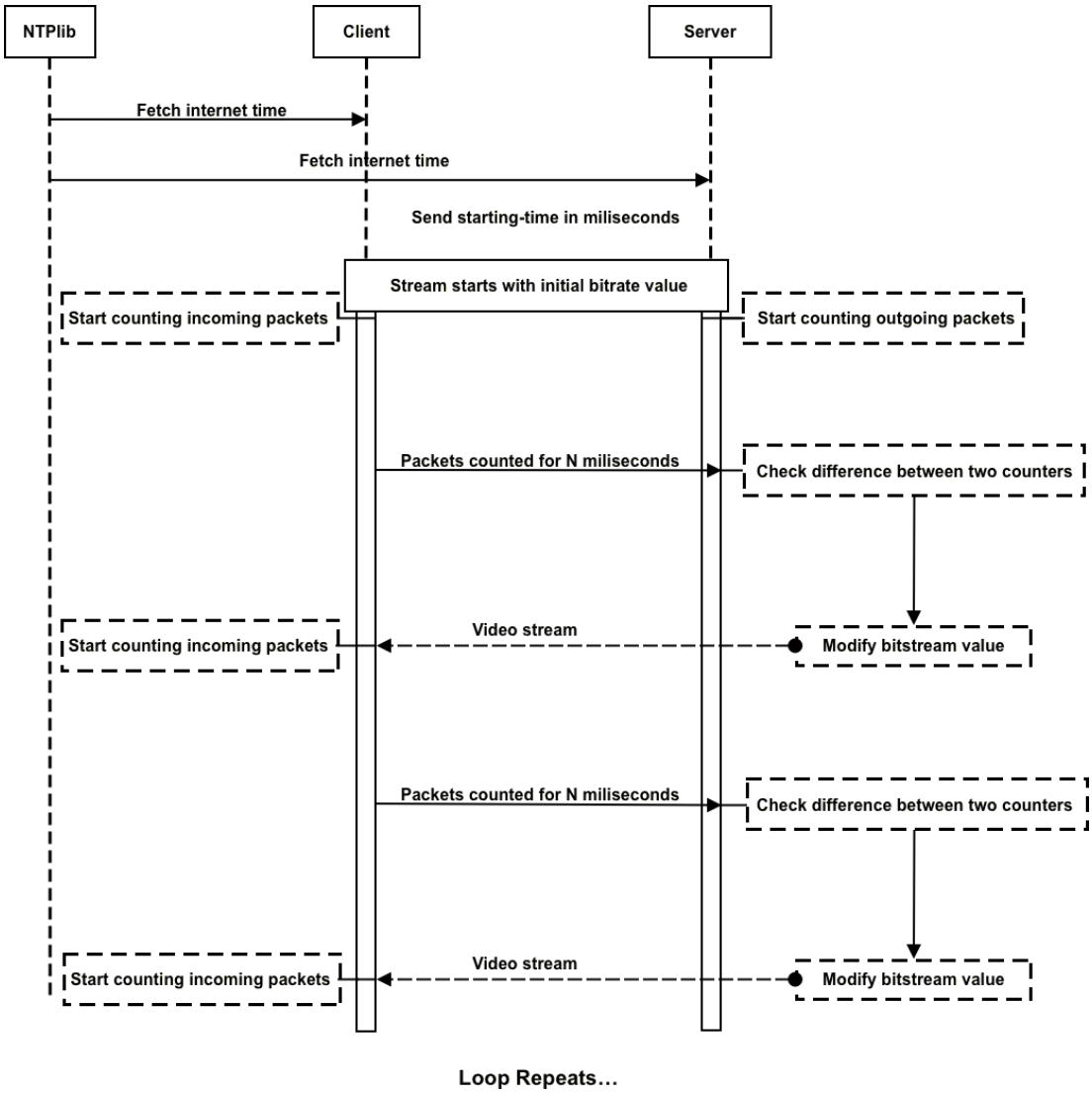

Figure 3.2: Sequence diagram

Another socket is then open to be able to send the details about the received packets, by the client to the server. The bitrate is then adapted based on the difference between the number of packets sent and received. If the difference between the two counters is under a certain level, the bitrate is increased until the difference crosses the specified level. To be able to synchronize the counting of the packets at the same time interval a synchronization mechanism is also used that starts the application and sends the QoS report at a chosen interval of milliseconds. This is done by implementing a Python library name NTPlib [20].

This mechanism allows us to have a better streaming quality for the given band-

width because we can manipulate the bitrate, and different variables which impact the streaming quality, to a more precise level as shown in Figure 3.3

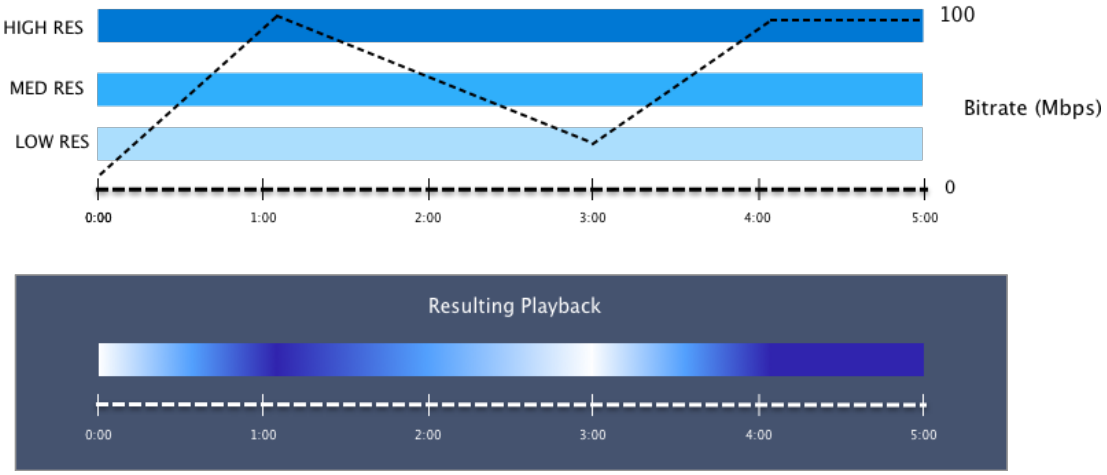

Figure 3.3: The response of dynamic ABR algorithm under variable bandwidth

As we can see, our solution does not have a fixed bitrate for different bandwidth levels like in the case of other ABR implementations. In the base of the bandwidth we have a more optimal bitrate of the video stream, a bitrate that is translated to a video stream quality that looks more of a gradient playback quality.

3.2 Software Design

The standard UML diagram of the application is divided into 3 classes.

The transmitter and receiver UML diagrams are the same in regard to their specific purposes. There are main three functions, supported by several secondary ones, each having a particular role in the application. The _$init$_ function creates the necessary GStreamer configuration variables to be able to create the stream. The stream is then initialized in the run function. The purpose of this function is to start the synchronization process, creating the initialization of the stream, and then counting the sent and received packages. The main difference between the server and the client application is in the GStreamer configuration variables and the supporting functions. The server run function also has the duty to create the TCP QoS stream to connect to the client. Once the connection is achieved, the client sends the statistical data, in our case the number of packets it counted, to the server where the ABR algorithm is implemented and the

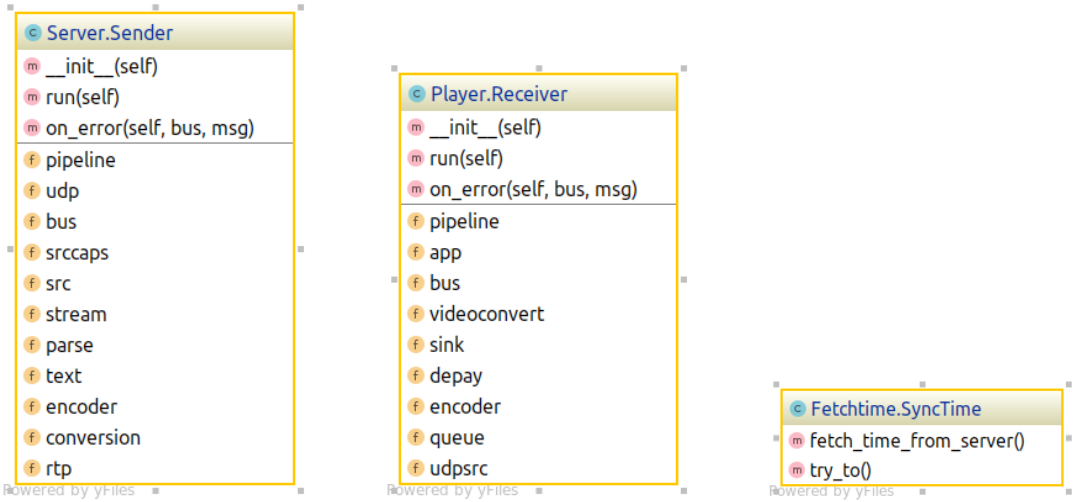

Figure 3.4: Standard UML diagram.

bitrate is changed. The last main function $on\_error()$ serves in case of a connection error between the two for log purposes.

3.3 Implementation of Gstreamer *Nvidia TX2*

To be able to stream the live data we implemented the GStreamer under a Python wrapper to create an RTP stream. GStreamer was chosen as it is the only framework able to manipulate the Nvidia codec's bitrate [15] and because it is a well accepted and adopted library for streaming purposes. The entire software was implemented in Python because of the vast libraries available, especially for the ease of use of the packet manipulation libraries and for managing Gstreamer script elements on a high level of abstraction. Last but not least, Python was chosen because different frameworks for Machine Learning (ML) algorithms are mostly written and implemented in Python [21] and these may be used to create a better ABR algorithm.

A GStreamer script should be visualized as a pipeline as shown in Figure 3.5. Each part of the script is divided into elements. We should think of GStreamer elements as black boxes. The data is created in the source element and ends in the sink one. Each element is composed of two parts, an input pad and an output pad. For example, we may have a decoder element which accepts a given stream of data and encodes it in a given format.

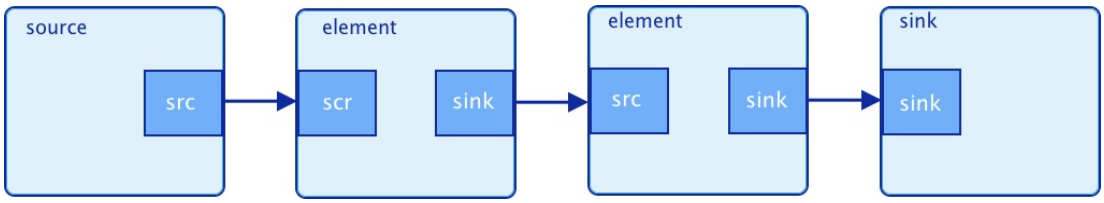

Figure 3.5: Elements of a Gstreamer pipeline

For the purpose of our solution we created a pipeline with the following sequence:

- **Transmitter**
  1. Generate the video data
  2. Encode the video data
  3. Split the data into smaller packets
  4. Send the packets through RTP transport protocol
- **Receiver**
  1. Receive the packets from the network
  2. Put together the packets into video data
  3. Decode the video data
  4. Visualise the video

To do this we use the following Gstreamer pipeline elements for both the transmitter and receiver:

1. ***nvcamerasrc***

   This is a source element for generating stream data in the form of images which will then be used by the following elements as a source.

2. ***nvvidconv***

   The element is used to convert the video in a given rotation, in our case rotation level 6 which translates to 90 degrees. It should be specified that this is an Nvidia proprietary element and is not available in the main source code repository of GStreamer.

3. ***CapsFilter*** We can visualize Capsfilter as a setting required by some elements of the pipeline to connect with each other. In our case this is a setting for the camera resolution, framerate ,etc. as shown on Figure 3.6.

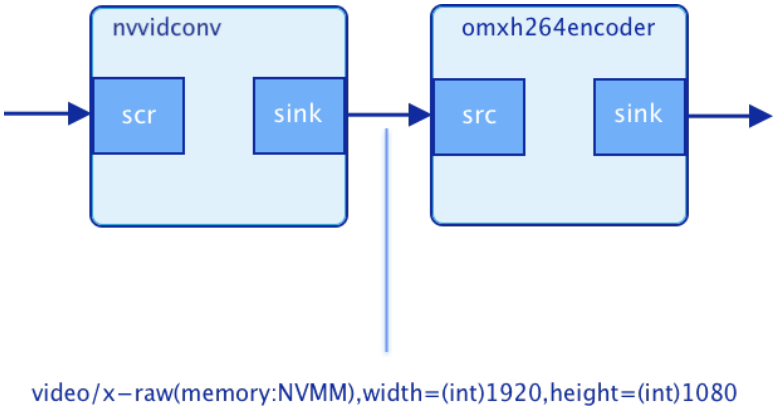

Figure 3.6: Caps of a Gstreamer element

4. ***omxh264enc / omxh265enc***

   This element is a custom encoder used by and proprietary of the embedded Nvidia board to encode the video images in H.264 and H.265 compression format. A parameter is given to the element to make the bitrate variable by giving a default bitrate and a control rate level, which enables the continuous changing of the bitrate.

5. ***h264parse / h265parse***

   This element is necessary for extracting missing information and, if needed, splitting it into packets and/or transforming packet format. The purpose of parsing is to understand the stream format and to signal the format of the stream to the upcoming element. It is also used to convert one H.264/H.265 stream from one H.264/H.265 format to another without the use of encoding. [22]

6. ***rtph264pay / rtph265pay***

   This element has a similar function to the h264parse element. It converts the stream in a given format but only by packetizing RTP packets, adding an RTP payload.

7. ***udpsink***

   This element is necessary to transport the given RTP stream from the transmitter to the receiver, using the UDP transport protocol.

8. ***udpsrc***

   This is used as the data source of the receiver to obtain the data, provided by the transmitter. The data is obtained through a port specified by the transmitter.

9. ***rtph264depay / rtph265depay***

   The purpose of this element is to extract H.264/H.265 video from the RTP packet that can be then used downstream by the pipeline.

10. ***queue***

    Until one of the limits, specified by the properties max-size-time, max-size-buffer and/or max-size-bytes, is reached the data is queued. If a thread attempts to push more data into the queue it will be blocked until more space becomes available. The queue creates a new thread on the source pad to separate the processing on the source and sink pad. [23]

11. ***avdec_h264 / omxh265dec*** Both elements are used to decode the video stream in their respective formats, H.264 and H.265, but the omxh265dec can only be used on the embedded board, as currently there is no decoder available on GStreamer.

12. ***nvoverlaysink / xvimagesink***

    This is the last element of the streaming pipeline used to visualize the video. This element can also only be used on the embedded board.

The GStreamer pipeline script and a visual representation for the transmitter, based on H.264 video compression standard, is shown in Figure 3.7 and Listing 3.1

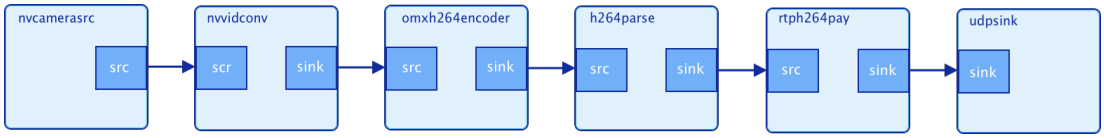

Figure 3.7: GStreamer transmitter pipeline

```
gst-launch-1.0 nvcamerasrc fpsRange="30 30" intent=3 ! nvvidconv flip-method=6 ! '
    video/x-raw(memory:NVMM), width=(int)1920, height=(int)1080, format=(string)I420,
     framerate=(fraction)30/1' ! omxh264enc control-rate=2 bitrate=4000000 ! 'video/x
    -h264, stream-format=(string)byte-stream' ! h264parse ! rtph264pay mtu=1400 !
    udpsink host=$CLIENT_IP port=5000 sync=false async=false
```

Listing 3.1: Transmitter pipeline

A visual representation of the receiver pipeline script is shown in the figure:

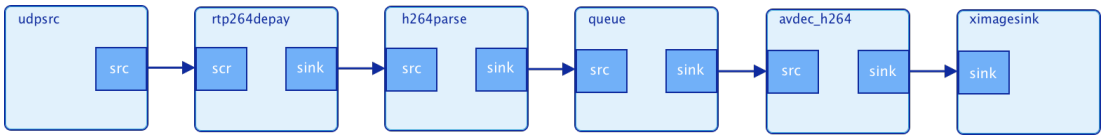

Figure 3.8: GStreamer receiver pipeline

```
gst-launch-1.0 udpsrc port=5000 ! application/x-rtp,encoding-name=H264,payload=96 !
    rtph264depay ! h264parse ! queue ! avdec_h264 ! xvimagesink sync=false async=
    false -e
```

Listing 3.2: Receiver pipeline

We also create an H.265 stream for the application. A stream which is visually represented in the GStreamer pipeline script for the transmitter is shown in Figure 3.9.

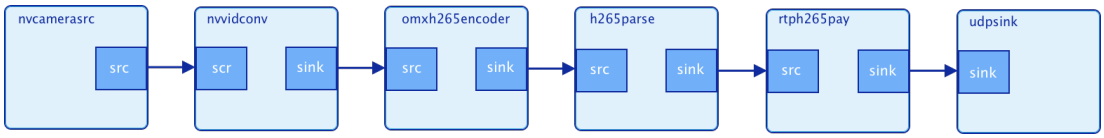

Figure 3.9: GStreamer transmitter pipeline

```
gst-launch-1.0 nvcamerasrc fpsRange="30 30" intent=3 ! nvvidconv flip-method=6 ! '
    video/x-raw(memory:NVMM), width=(int)1920, height=(int)1080, format=(string)I420,
     framerate=(fraction)30/1' ! omxh265enc low-latency=1 control-rate=2 bitrate
    =4000000 ! 'video/x-h265, stream-format=(string)byte-stream' ! h265parse !
    rtph265pay mtu=1400 ! udpsink host=$CLIENT_IP port=5000 sync=false async=false
```

Listing 3.3: Transmitter pipeline

A visual representation of the receiver pipeline script is shown in the figure:

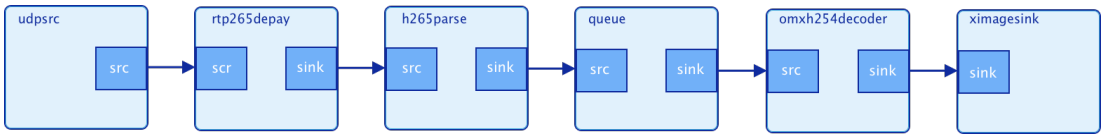

Figure 3.10: GStreamer receiver pipeline

```
gst-launch-1.0 udpsrc port=5000 ! application/x-rtp,encoding-name=H265,payload=96 !
    rtph265depay ! h265parse ! queue ! omxh265dec ! nvoverlaysink sync=false async=
    false -e
```

Listing 3.4: Receiver pipeline

As can be seen in the Listing 3.1 and 3.3, the omxh264enc / omxh265enc encoder has a property named ***control-rate*** set to the value 2 which modifies the video quality based only on the bitrate. Normally, to control the quality of a video several parameters would be modified, like frames dropped in the video, the type of frames that compose the video resolution, etc. but thanks to this property the encoder tweaks this parameter in the background in such a way that the output video stream equals the inputted bitrate in the parameter ***bitrate*** that in our case is set to 4000000 bits.

For the bitrate to be changed in the code implementation without the need to restart the stream, the two parameters of the omxh264enc / omxh265enc encoder explained earlier should be present and also a GStreamer loop-function named GLib.MainLoop().run() should be called. This function on every iteration gives us the possibility to change the bitrate without having (the need) to restart the stream. However, because we already have a loop-function created by SCAPY and AppScheduler we removed it and substituted it with their equivalent as seen in the Appendix A.1. It should be noted that both encoding compression algorithms H.264 and H.265 are shown in this high-level visualization of the GStreamer elements, but only the H.264 compression is implemented because the H.265 standard can only work between two TX2 boards, hence the omission of the implementation.

3.4 Adaptive Bitrate Algorithms

For this thesis, two algorithms are presented, approaching the problem in two different implementations. Each algorithm uses a different formula and method to achieve, in its scope, a satisfactory video quality with respect to the available bandwidth. The

first algorithm uses a linear approach where it computes the difference between the number of packets transmitted and sent, changing the bandwidth accordingly through multiplying the difference with a given variable, without taking in consideration the past history of the transmission. The second algorithm, on the other hand, uses the past history of the difference between the sent and received packets to compute the bitrate of the video stream. If the past transmissions are successful the algorithm increases the speed exponentially. The difference between these algorithms is only seen in the increase of the bitrate level, but both algorithms decrease the bitrate using the same method. If the difference between the sent and received packets is bigger than our specified level, 10 packets, the video bitrate is automatically changed to suit the current bandwidth level.

3.4.1 Decrease Algorithm

Before detailing the two algorithms which increase the bitrate of the video stream we will focus on the decrease algorithm formula that is common to both. Because we cannot decrease the bitrate of the video stream to a level that is not on a par with the bandwidth level at that exact moment, the formula is an Identity as shown in the Equation 3.1.

$$Y = X \tag{3.1}$$

where ***Y*** is the bandwidth level and ***X*** is the resulting bitrate for that given level. The ***X*** variable is computed in case the number of packets sent is bigger by a difference of more than 10 pakets as this would mean that, because of the decreasing bandwidth level of the connection, the missing packets could not be sent and were dropped.

3.4.2 Linear Algorithm

The first algorithm uses a linear formula (3.2) where the difference between the number of packets sent and received is used to calculate the output bandwidth by multiplying it by a variable obtained by different laboratory tests.

$$Y = Z + ((X * (-1)) + 10) * 1000 \tag{3.2}$$

Y represents the output bandwidth, ***Z*** the current bandwidth, and ***X*** the difference

in packets sent and received. As can be seen, there are also three fixed constants in this formula. The first constant is a multiplier needed to convert the difference in packets into a negative value if the difference is greater than the second constant, in our case 10. This is done because of the high probability of having some false negatives in the packets lost count which, based on prior testing, is typically under 10 packets. The third constant is used as a multiplier to increase the bandwidth by a certain value greater than the packet difference. This constant can be adjusted based on the bandwidth fluctuations of the connection as seen in Chapter 4. A representation of the algorithm output can be seen in Figure 3.11.

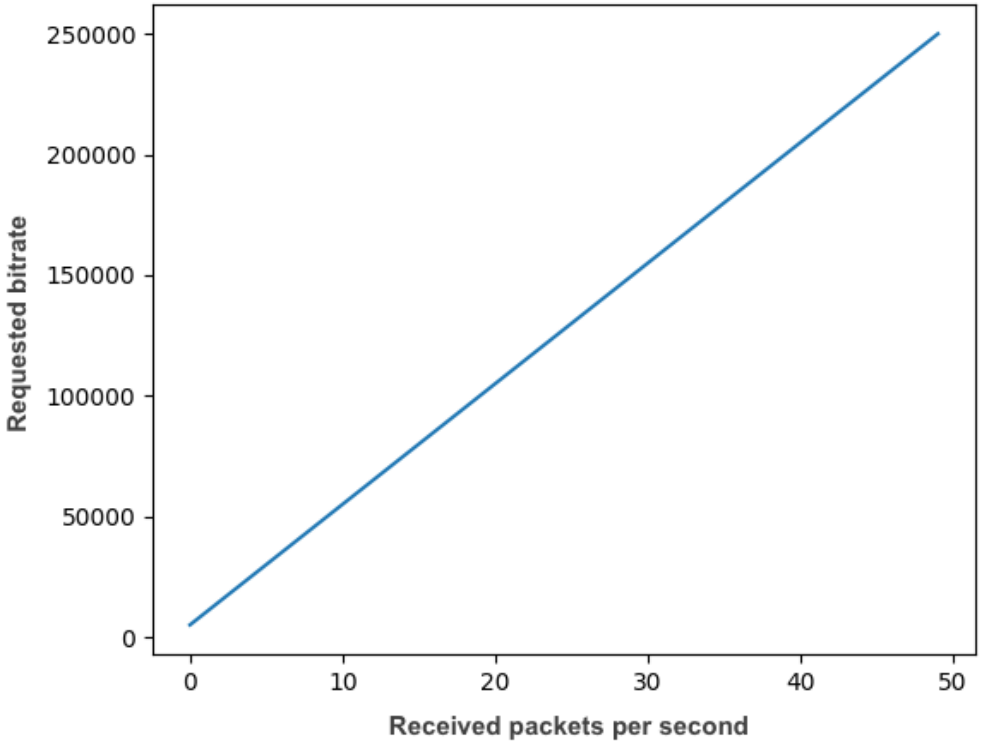

Figure 3.11: Linear Algorithm

This algorithm depends heavily on the current packet count difference without taking into consideration the past packets and this brings, in theory, several advantages and disadvantages which should be considered. The following assumptions are then made:

Advantages Assumption

Fast and predictable in terms of speed:

Because of the straightforward linear approach, the adaptive bitrate algorithm should respond particularly fast to achieve the satisfactory video streaming quality in relation to the available bandwidth. It should not introduce delays doing the necessary calculation, as it does not need extra time to compute past packets into the formula. Also, because the constants are given to the algorithm it is really predictable, hence making it ideal for analyzing the bandwidth immediately after the time this algorithm has been executed.

Disadvantages

Arbitrary speed:

What should make this algorithm fast and predictable is, at the same time, what could make it slow. Because of the unpredictable terrain in which the drone will navigate we can assume an unstable connection with a different bandwidth interval. If the network has a high capacity bandwidth the constant we provided to increase the bitrate may be too small for the bitrate to grow as fast as the bandwidth allows. This is also true for the opposite. If the constant is too large for the bandwidth it may surpass the maximum bandwidth, resulting in the algorithm trying to find the maximum bandwidth by increasing and lowering the bitrate over different iterations, making it slower.

Bad performance in bandwidth saturation:

This also brings us to another negative effect from which this algorithm may suffer. When the maximum bandwidth is found, it is hard to maintain a stable video bitrate over every iteration of the difference between packets sent and received. This would happen because it is highly unlikely to locate the exact ceiling of the bandwidth because of the constant in the equation. If the ceiling is not located in this value, the algorithm will be in a loop, trying to achieve the maximum bandwidth. Since in real life the bandwidth is always changing, this is a notable point which should be taken into consideration for stable networks. Not maintaining the bitrate in a stable state will introduce jitter to the video if the bitrate difference before and after a bandwidth drop is high.

3.4.3 Slow Start Algorithm

The second algorithm is introduced to mitigate some of the disadvantages of the linear one. This algorithm takes into consideration, to some extent, the past packet counter and tries to increase the video bitrate in a smoother, slower, way. The inspiration for this algorithm comes from the slow start TCP algorithm of TCP congestion control method [24] which is a congestion control strategy used by TCP.

The slow start algorithm limits the amount of data that can pass through in the beginning of a new established connection, to ramp it progressively until the carrying capacity of the bandwidth is reached. An important part of the slow start is the initial congestion window which puts a limit on how much data can be transmitted at the beginning of the connection. Initially, the slow-start algorithm begins with a small window size of a Maximum Segment Size (MSS) of either 1, 2, 4, or 10. The congestion window value will be increased by each acknowledgment received doubling the window size each time a round-trip is performed. The algorithm will increase the transition rate until a loss is detected, in case the Slow Start Threshold value (SSTHRESH) or a limiting Receiver's Window (RWND) are reached. In case of a loss event TCP also reduces the load on the network, assuming the loss was due to network congestion. To adapt the TCP congestion algorithm to our case we modified the formula as shown in Figure (3.3) :

$$Y = X^4 \tag{3.3}$$

The output bandwidth ***Y*** is equal to a variable ***X*** to the power of four. The variable in our case and for the purpose of the implementation is fixed at a certain value through the evaluation tests in Chapter 4, but it can be adjusted in case of bandwidth fluctuations as explained in more detail in Chapter 5. A representation of the algorithm can be seen in Figure 3.12.

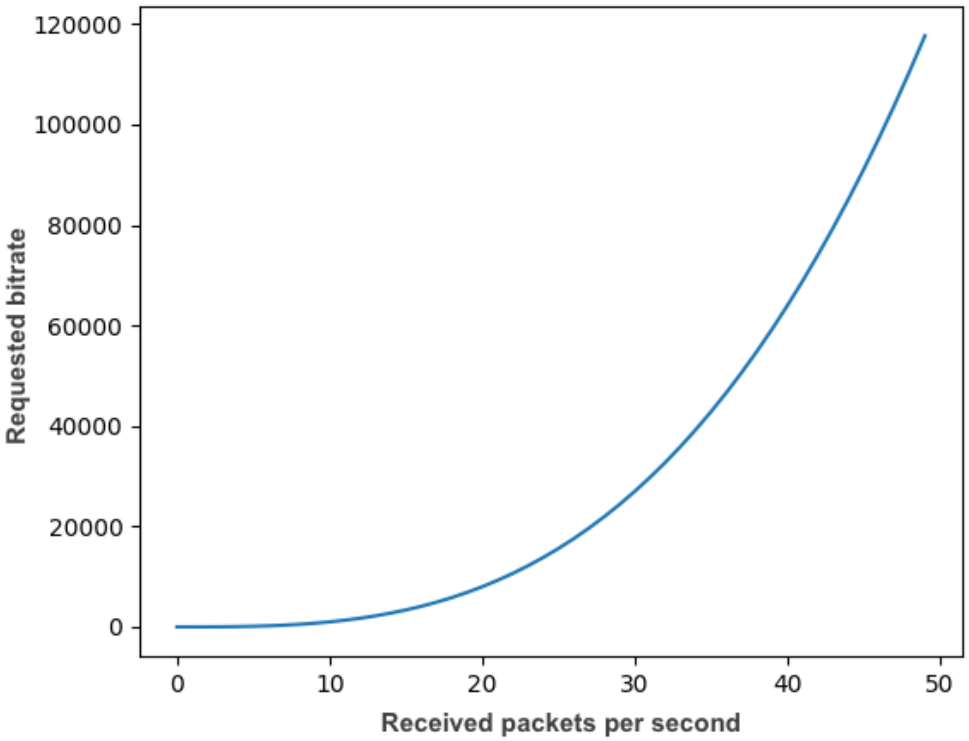

Figure 3.12: Exponential Algorithm

This brings us, in theory, several advantages and disadvantages.

Advantages

Good performance on bandwidth saturation:

The main idea of the slow start algorithm is to maximize the bandwidth threshold by maintaining good video quality even though the bandwidth itself is extremely variable. To overcome the second disadvantage of the linear algorithm, once the bandwidth falls, we try to increase the bitrate by a small amount, in the beginning, to be sure we didn't reach the maximum level, then we increment the speed by which the bitrate increases exponentially until we hit the maximum bandwidth threshold. The small increase in bitrate allows us to maintain the maximum level of the bandwidth by not introducing dramatic drops in the bitrate which would result in increased jitter of the video.

Fast to reach the saturation point on a stable network:

In case of a stable network, this algorithm should, theoretically, be faster than its linear counterpart. Assuming no packet drops occur the bitrate will increase exponentially, reaching the maximum level at a higher speed. The speed of this algorithm is especially high if the bandwidth has a great capacity, as being exponential will make the alrogorithm reach the maximum level of the bandwidth faster.

Disadvantages

Slow when the bandwidth is extremely variable:

The ideal case scenario for this algorithm is for the bandwidth to be stable through long enough not to introduce extreme drops, and variable when at the maximum capacity. But this is hardly a real-life scenario. From time to time we can see some packets drop because of unforeseen events like ISP's firewall, network problems, etc. which will stop this algorithm from increasing its speed and, because of the slow start, the more drops we have the slower this algorithm can be to reach the maximum level of the bandwidth.

4 Results Analysis

To evaluate our solution we need to design a testing environment from which we can manipulate the streaming bandwidth available between the server and the client. The approach we implemented, and the one which allowed us to have a good degree of bandwidth manipulation, is to create a laboratory environment. One non-functional requirement of our proposed solution, as explained in Chapter 3, is to be able to assess the level of the bandwidth even behind an operating system's firewall. This is required in case of a firewall rule targeting the same port as our application uses, which would impair the video streaming.

For this reason, we used a Python library named Scapy which would allow us to estimate the bandwidth level, even behind a firewall, by counting the packets on the network adapter level. This would mean that we will not need to reset the bitrate of the video when we change the video stream port, making the stream already adapted to the available bandwidth. The implementation of such a library, on the other hand, restricted us on the possible ways we could create the testing environment. This is because of the inability to use network shaper software as it works almost like a firewall, allowing only a certain number of packets to go through a certain port.

Another approach we could have implemented was to create a virtual machine, with a host operating system, able to receive the video stream. The virtual machine would then be used as a sandbox, creating a fake network adapter for Scapy to work upon. But after some preliminary testing, we had to exclude this option because of the substantial delay introduced by the host operating system to render the video frames upon receiving them.

4.1 Evaluation Environment

To be able to fulfill the non-functional requirements we decided to create a laboratory environment with the help of a network router which would be used as a communic-

ation link between the receiver and the transmitter. As a non-functional requirement, we needed a router which would allow us to manipulate the bandwidth without having the need to restart and close the stream connection. Our choice was *Linksys DIR-809* [25] as it fulfilled this requirement. In order to use this function we had to create our own software to be able to change the bandwidth values in a dynamic way, because the router did not provide an Application Programming Interface (API) we could implement to achieve such a task. To have an API-like approach for our tests we used a framework called ROBOT [26], originally developed by Nokia to test different software, which with the help of a library called Selenium [8] allows us to manipulate the administration interface of the router by automating the input of the variables on the web interface of the router as shown in Figure 4.1.

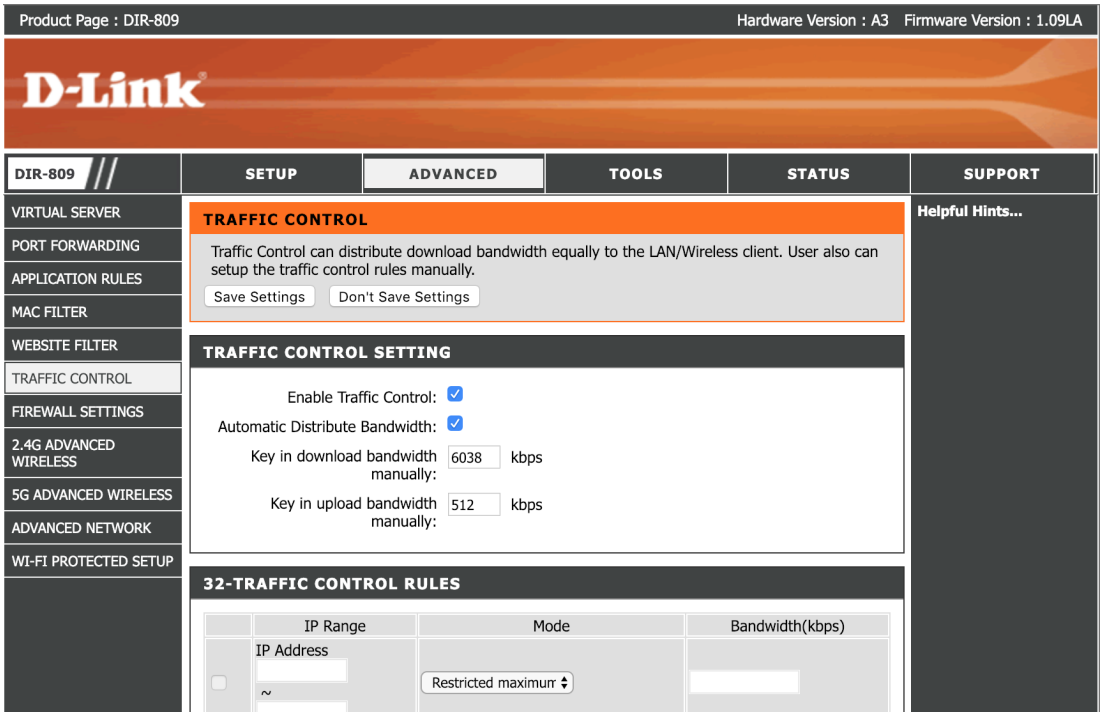

Figure 4.1: Linksys router web interface

ROBOT allowed us to write easy-to-understand tests with a human-like syntax, but also to have the possibility to use the SeleniumLibrary which is a ROBOT web testing library that utilizes Selenium tool. Selenium is an automated testing suite of web-based applications for a wide range of browsers and platforms. Selenium's WebDriver tool makes calls to the browser using each browser's native support for automation, allowing us to call and navigate any web page with the help of pre-made scripts, in

our case coded in Python, as in an API-like approach. A representation of the testing environment architecture is shown in Figure 4.2.

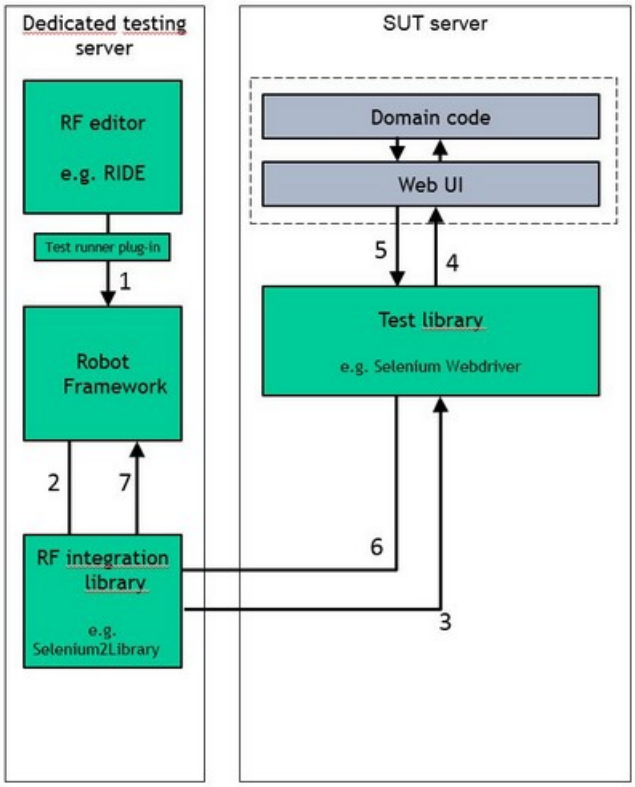

Figure 4.2: Testing environment architecture [8]

To achieve different test cases we created two bandwidth manipulation tests, each defined in a Python function linked to the ROBOT tests. The first one makes the bandwidth variable to a small degree through a sinusoidal function. It follows a simple formula to create the bitrate modification function.

$$Y = sin(Z) * A \tag{4.1}$$

Y represents the output bitrate which is created by the sinusoidal function of the variable Z multiplied by the constant A to give the bandwidth in megabits per second level as seen in 4.3.

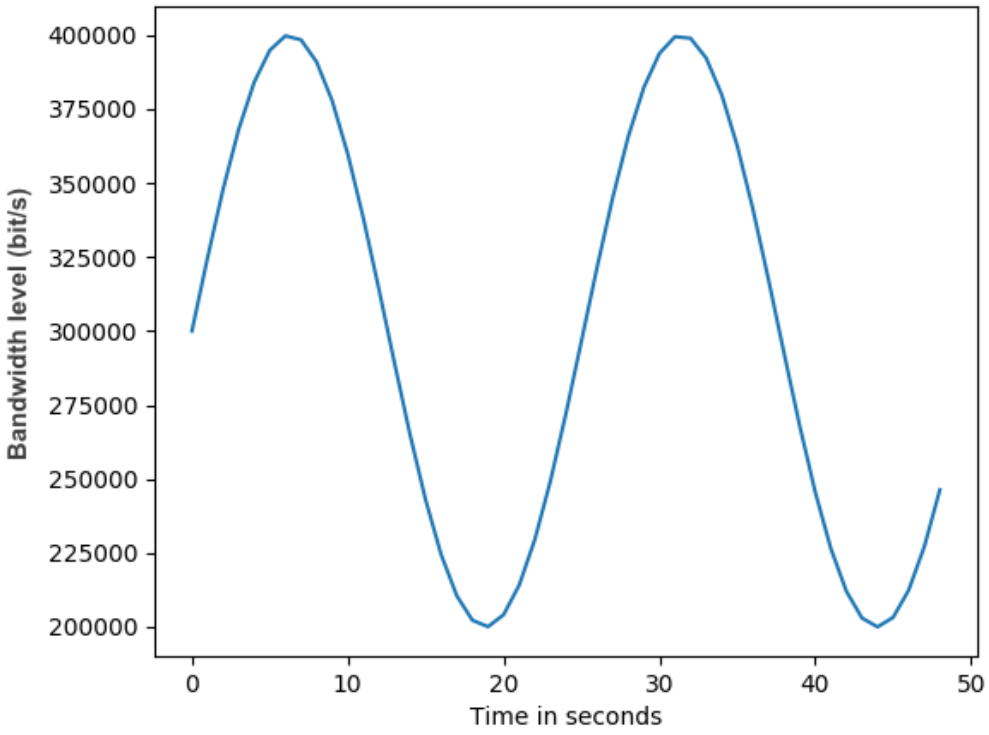

Figure 4.3: Bandwidth modification function

This test is conducted to test the fluctuations of the bandwidth by increasing and decreasing it in a smooth way.

In the second test, we wanted to simulate a real environment by abruptly increasing and decreasing the bandwidth to simulate an unstable real-life connection. For this purpose, we parsed a dataset with wireless bandwidth traces released at the MMSys 2013 conference in a paper entitled “Commute Path Bandwidth Traces from 3G Networks: Analysis and Applications“ [27]. For the purpose of our thesis, the route from Ljansbakken to Jernbanetorget was chosen, from the provided dataset, because of the variations in the bandwidth level which were without many extreme increases and drops compared to the other routes on the dataset which would make any type of stream quite unstable. A representation of the bandwidth levels which we used for our test is shown in Figure 4.4.

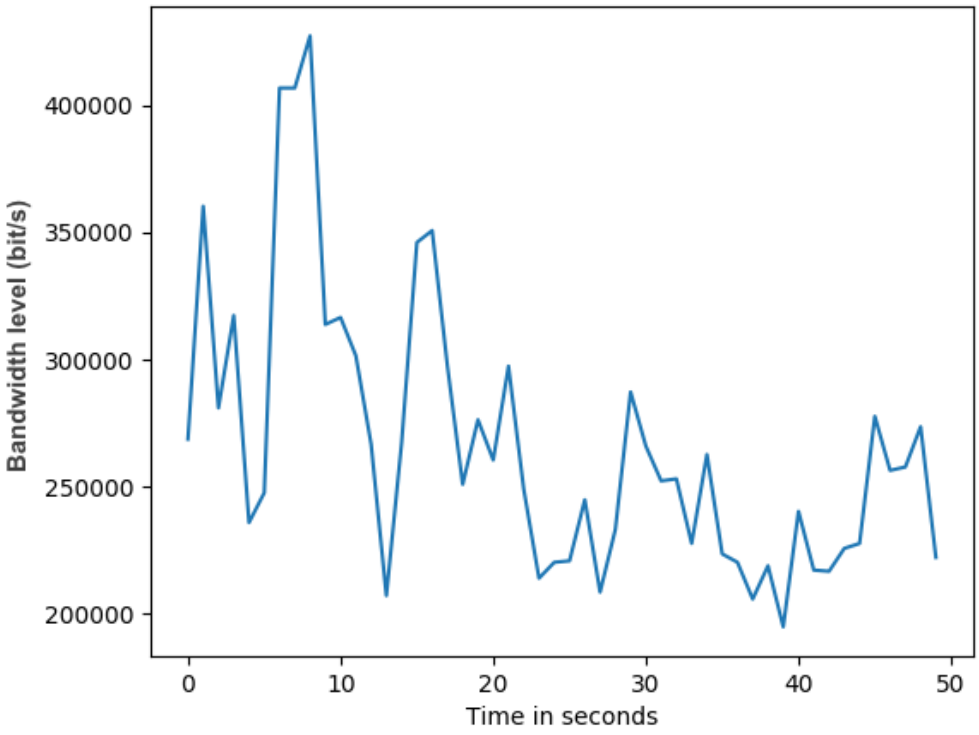

Figure 4.4: Bandwidth variability dataset

4.2 Evaluation Metrics

After implementing the the evaluation environment we need a way to evaluate the quality of the video streamed. For this thesis we used different methods, and while we tried to keep the subjectivity to a minimum that was not always possible because of the nature of the test itself. The evaluation criteria used are divided into three sections which are described below.

4.2.1 Picture Quality

Picture quality refers to the frame at a given time in the video stream which is then compared to the original frame of the video before it was streamed by the transmitter. This is done to understand the compression ratio of the frame as well as how well the frame was constructed withstanding missing data packets and how different the streamed frame is from the original. To be as objective as possible in the picture

quality test we compared the same frame at the end of both streamings, from the transmitter side and from the receiver side, and created an array of different tests used by the research community [28] [29], tests that we tried to create with different Python libraries and software such as ImageMagick [30], SSIM-PIL [31], etc. These testing algorithms are shown in the following table:

Picture Quality Statistics

Algorithm	Analysis Type	Additional Information
PAE	Peak Absolute Error	Within a channel, for 3D color spac
PSNR	Peak Signal to noise ratio	The ratio of mean square difference to the maximum mean square
MSE	Mean absolute error	Average squared error distance
RMSE	Square root mean square error	The standard deviation of the residuals (prediction errors)
SSIM	Structural Similarity Index	Image quality degradation caused by processing

We choose three different algorithms to assess the picture quality.

Mean Square Error (MSE) calculates the average squared difference between actual and uncompressed pixel values. Because it measures errors, values close to 0 are better.

Peak signal-to-noise ratio (PSNR) calculates the peak signal-to-noise ratio between the uncompressed image and the modified one. It is derived from MSE, and higher PSNR values mean lower mean square errors between the two images. If the PSNR value is low it implies a low quality of the compressed image.

Structural Similarity (SSIM) combines luminance, contrast, and local image structure into a quality score, where structures are patterns of pixel intensities. SSIM quality metric is close to a subjective quality score because it mimics the way humans perceive structures [32].

SSIM calculates a decimal value between -1 and 1 where 1 means that the images are identical even though in most cases a score is given on the interval [0, 1], where values closer to 0 represent no structural similarity [33]. SSIM can be negative when the local image structure is inverted [34].

We use different algorithms directly derived from MSE like Peak Absolute Error (PAE) and Root Mean Square Error (RMSE) which, as the name implies are different metrics evaluated concerning the same algorithm.

4.2.2 Video and Streaming Quality

Video quality refers to the general video quality of the stream. While the picture quality tests take into account just one frame at the given time, video quality evaluation focuses on the general construction of the video on the receiver end. An important aspect is

also the general FPS of the video. Because of the missing UDP packets, GStreamer may drop some incomplete frames. For this reason, a counter was put in the video file for each frame and then analyzed for a given second.

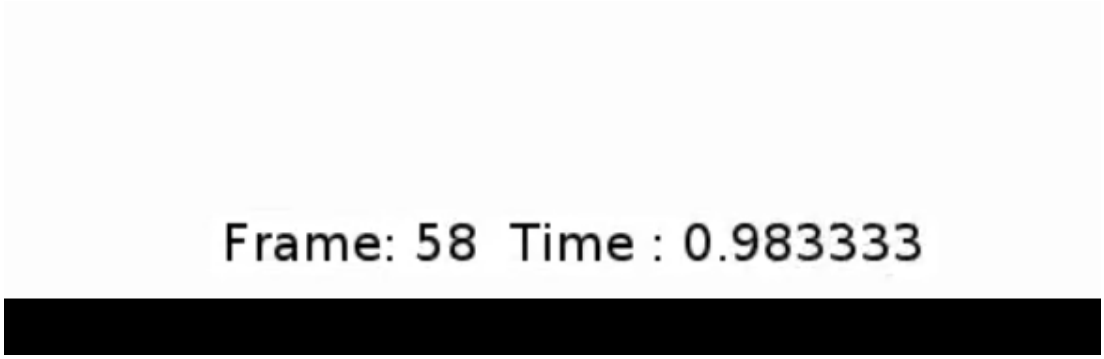

Figure 4.5: Example of FPS analysis for 1s

The video played by the receiver is saved and afterwards the number of frames are counted to create an average counter of the frames dropped and the lag in the video.

Another important test is the general latency of the video stream. To test it we inputted a timestamp to the video stream which starts to play at the exact same moment we start the stream over the network. At a given time a screenshot of the playing video at both ends is taken and the different timestamp labels are compared.

Figure 4.6: Example of latency analysis

The difference between the two counters will show the lag of the video stream as seen in Figure 4.6. This is done at different times of the video and at different levels of

bandwidth to evaluate the video streaming latency.

4.3 Laboratory Analysis

4.3.1 Sinusoidal bandwidth function

To evaluate the behavior of the two algorithms under a controlled bandwidth level we create our own functions as explained in Chapter 3. In the first case, we created a function which creates a steady increase and decrease of the bandwidth level in a sinusoidal approach as shown in Figure 4.3. After running our software for 50 seconds under such a function we collected and normalized the data and then proceeded to create a plot for both algorithms.

Linear Algorithm

To be able to have a satisfactory bitrate for the bandwidth we exchanged the constant 1000 of the controller formula [3.2] for the fixed number 18500. This constant is used to modify the angle of the linear algorithm function line and is derived from empirical research of the bandwidth. To be able to find an adequate fitting line to increasing segment of the bandwidth function we tried several constants, each incremented by 4600 as shown in Figure 4.7.

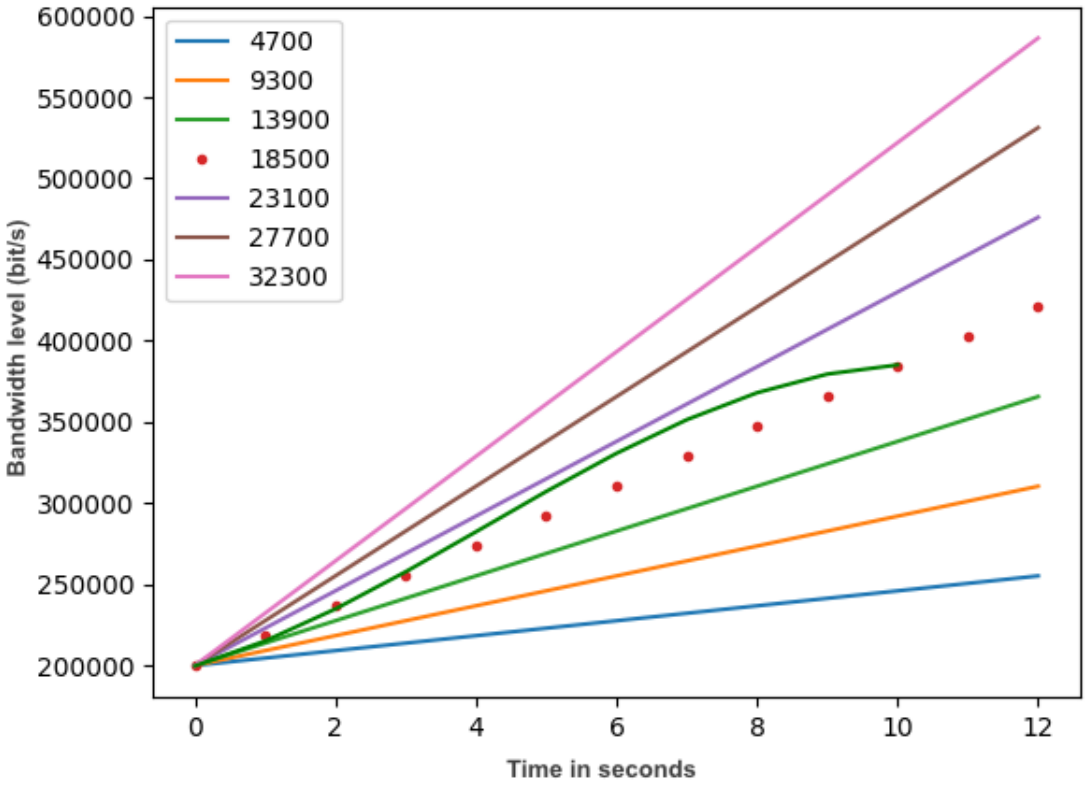

Figure 4.7: Empirical research of linear algorithm variable

As we can see the constant that fits the best the bandwidth function is the one labeled with red dots as it will not surpass the bandwidth functions and also will be closer to it compared with the other constants. We did not create more empirical test because the constant would not be transferable to a real-life bandwidth function as shown later in this chapter, but if a better precision of the constant is needed a lower increment may be implemented.

The plotted data relative to the bandwidth with the constant chosen is shown in Figure 4.8:

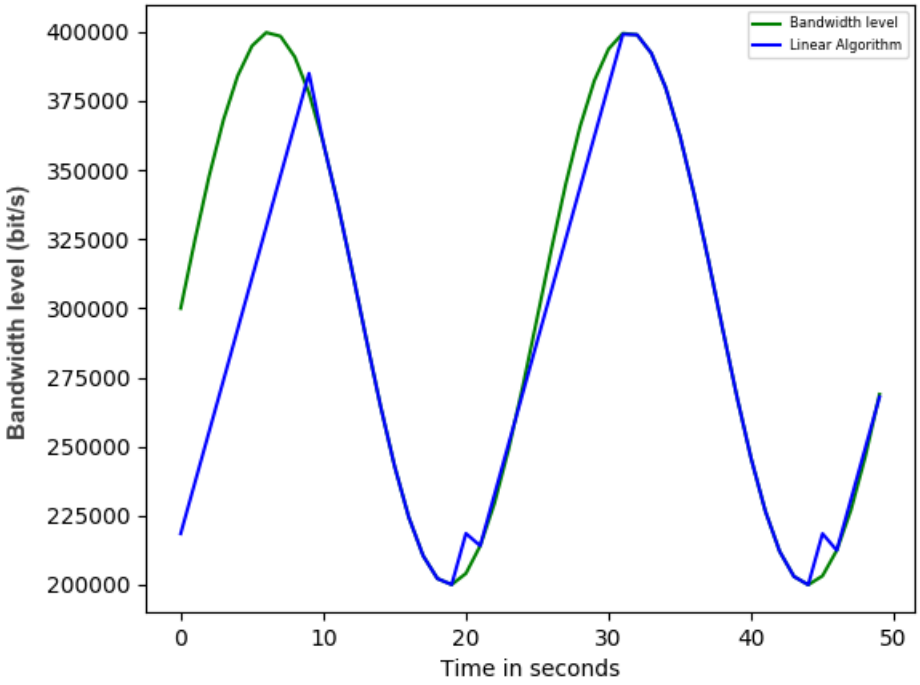

Figure 4.8: Plotted data of linear algorithm under sinusoidal bandwidth levels

As can be seen, the linear algorithm performed well, covering the majority of the bandwidth level. As per the assumptions made in Chapter 3, the algorithm was indeed fast and predictable. Knowing the current level of the bandwidth we could estimate the bitrate of the video according to the algorithm. However, it has to be noted that the speed by which the algorithm achieved the maximum bandwidth level is related to the constant variable which was modified to fit this specific bandwidth function and for this reason, the assumption of the disadvantage of the arbitrary speed still remains. Another disadvantage of the linear algorithm can be seen at the 10, 20, and 45 seconds of the stream. Because of the linear speed, the algorithm tried to increase the speed at the same level while the bandwidth was stable, which made the algorithm change the bitrate more than the bandwidth saturation point. This change resulted in a perceived lag by the client, with an average of 20 frames lost in the 10 seconds and up to 30 frames lost in the 20 and 45 seconds. Such behavior proved that the performance of the bitrate to the bandwidth saturation was indeed low, not only when the bandwidth level achieved its maximum and remained constant for a short amount of time but also on the lower spectrum, proving the disadvantage assumption.

Slow Start Algorithm

To be able to fit the bandwidth function as well as possible, the constant **X** of the equation 3.3 was set to 5. The data collected gave this result:

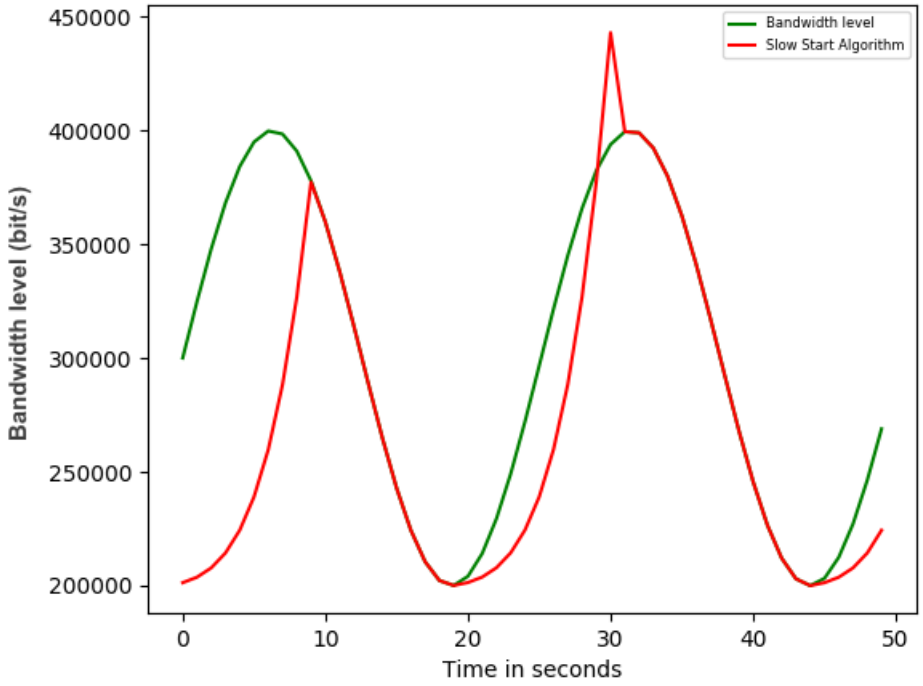

Figure 4.9: Plotted data of slow start algorithm under sinusoidal bandwidth levels

The first thing we notice is that the algorithm did not achieve the best possible bitrate to the bandwidth ratio. This can be seen especially from the beginning of the stream to the 10 seconds. Because of the nature of the algorithm, it takes time to increase the bitrate, hence not reaching the maximum of the bandwidth, disproving the assumption that the algorithm should be fast to reach the bandwidth saturation point. This can also be seen further at 30 seconds. Because the algorithm was growing exponentially it reached a level that went well beyond the bandwidth maximum. This resulted in a lag of 1 second with almost 70 frames lost at that point. An interesting part is seen in the lower part of the bandwidth function where the bandwidth level is decreasing and maintains, for a short amount of time, an almost stable level. At this point, the algorithm maintains a stable bitrate level relative to the saturation point of the bandwidth. This behavior in the maximum and minimum bandwidth levels made

us partially disprove the advantage assumption of good performance on bandwidth saturation, as the algorithm behaves well on the lower spectrum of the bandwidth but poorly on the higher one.

4.3.2 Real dataset bandwidth function

Thanks to the known sinusoidal function of the bandwidth and preliminary tests, we could tweak the variables of the ABR algorithms that would fit such a bandwidth function to give satisfactory results. But in a real case scenario, the connection is never on desired levels or stable enough. For this reason, we decided to try both algorithms, with the variables set by the sinusoidal bandwidth function, on a real-life dataset with data recorded from a moving car over a 3G network.

Linear Algorithm

The linear algorithm with variable bandwidth gave satisfactory results by covering the majority of the bandwidth function as can be seen in the Figure 4.10 .

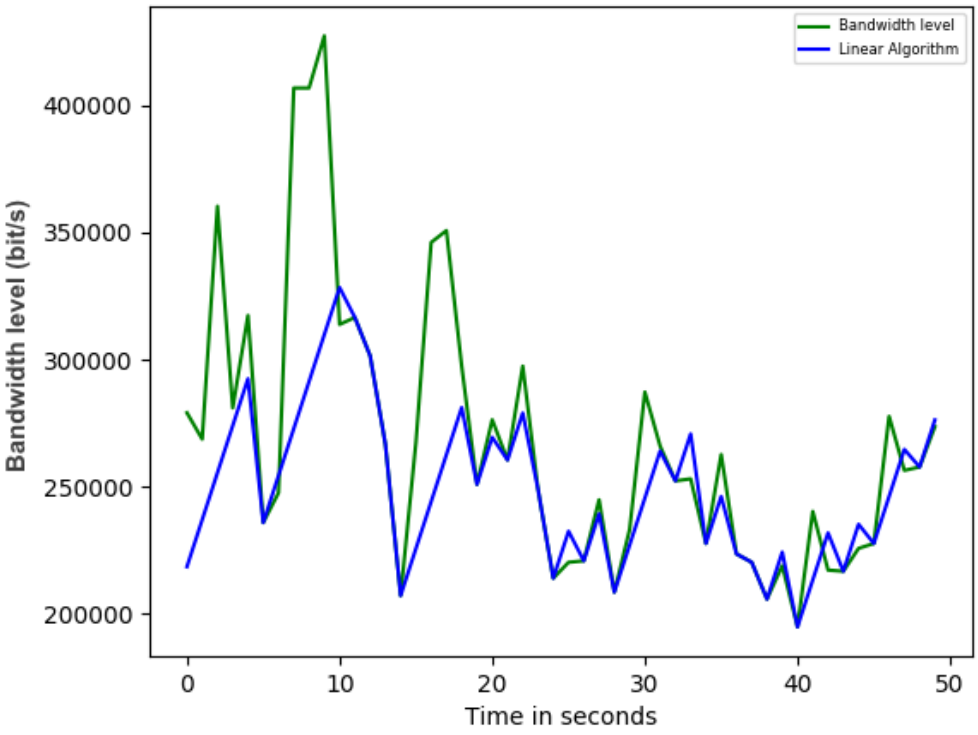

Figure 4.10: Plotted data of linear algorithm under variable bandwidth levels

While the algorithm did not always reach the maximum bandwidth possible, as seen from the 5 to the 10 seconds, it still managed to cover the majority of the bandwidth levels. While on second 25, 32 and 44 we see some frame drops which are translated to a maximum of 200 ms lag in the video stream. The real surprise was the behavior of the bandwidth in the majority of the bandwidth saturation levels. Because of the high fluctuation of the network, the bandwidth did not remain stable enough for the algorithm to overestimate the bitrate. While this does not disprove the disadvantage of bad performance at the saturation level we see that this rarely happens in real cases. It also has to be stated that the increase of the bitrate, relative to the bandwidth level, was indeed fast even without knowing the bandwidth function a priori, hence proving once more the assumption of the algorithm being fast to reach the saturation point.

Slow Start Algorithm

On analyzing the slow start algorithm in a real-life environment, we see a sudden change in the expected behavior on a very variable bandwidth as can be seen in the Figure 4.11.

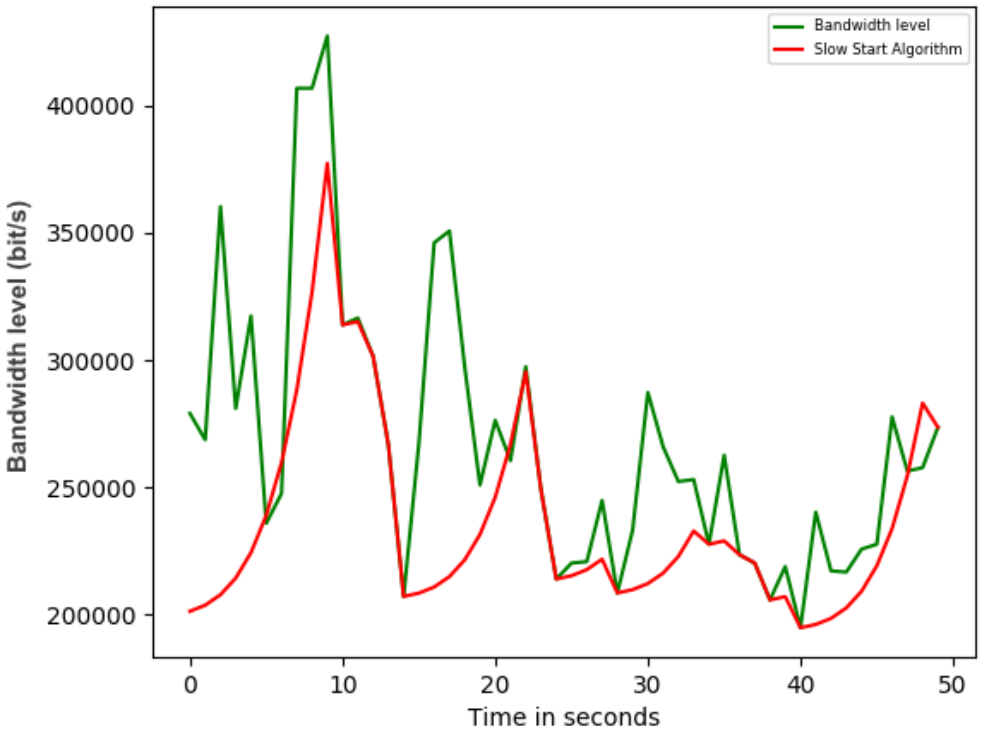

Figure 4.11: Plotted data of slow start algorithm under variable bandwidth levels

The algorithm did not fully cover the majority of the available bandwidth as seen previously in the sinusoidal bandwidth function. This can be noted in the seconds between 12 to 20, 28 to 30, and 40 to 47. However, the slow start algorithm did indeed reach a higher bitrate than its counterpart, as seen in the seconds from 6 to 9 and 20 to 22. While the stream was smoother (with a single lag happening at the second 49) the picture quality levels were not at a par with those of the linear algorithm. The algorithm did also reach a good performance on the lower levels by maintaining a stable connection and having fewer general frame drops compared to the linear algorithm because of the not-sudden changes in the bitrate level.

4.3.3 Conclusions

Picture Quality

To be able to have a clear understanding of the picture quality of the video stream we took different samples across the stream at the receiver and transmitter side. These samples were collected mainly when we encountered big differences between the two different algorithms. These samples were then tested under different quality assessment algorithms as explained earlier.

The first assessment was done on the behavior of the linear and slow start algorithms with the sinusoidal bandwidth function. We took four different frames at four different timestamps and analyzed the difference between the picture output of the linear and the slow algorithm in relation to the sinusoidal bandwidth function.

As can be seen in Figure 4.12, under the sinusoidal bandwidth function, the linear algorithm surpassed its counterpart in delivering the more satisfactory bitrate for the given bandwidth.

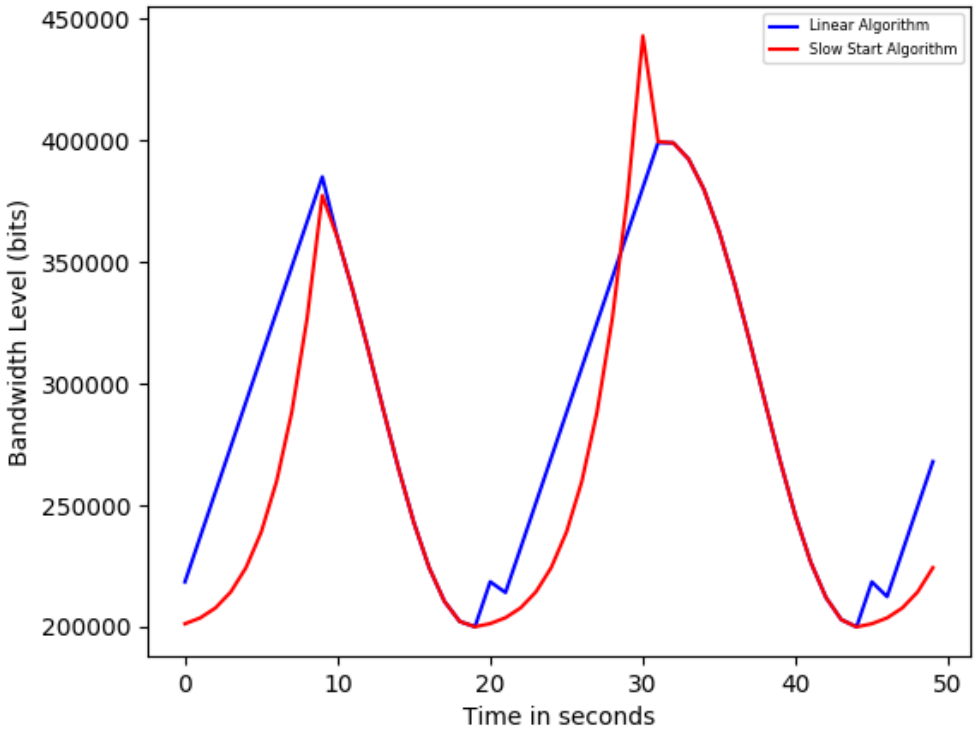

Figure 4.12: Plotted data of linear and slow start algorithms under variable bandwidth levels

This was perceived as a better picture quality as shown by the picture quality metrics in the table.

Sinusoidal Bandwidth Function						
Timestamp Tested	Bandwidth Level	Bitrate Output	MSE	RMSE	PSNR	SSIM
Linear Algorithm						
00:05	399749	329500	0.00164698	0.040583	27.8331	0.8015942052425324
00:22	272058	269607	0.00213008	0.0461528	26.716	0.7623592820124696
00:48	268888	267955	0.00216061	0.0464824	26.6542	0.7610025098709223
Slow Start Algorithm						
00:05	399749	238995	0.105379	0.324622	9.77244	0.5456502490399835
00:22	272058	224425	0.105112	0.32421	9.78348	0.5534908761915731
00:48	268888	224355	0.105112	0.32421	9.78348	0.5534908761915731

As we can see at the 5, 22, and 48 second, the MSE and RMSE are smaller in the linear algorithm compared to the slow start which is translated to a better perceived picture quality, as in both cases the lower the value the better. A better perceived quality can also be seen in the higher levels of PSNR and in the value of SSIM which is close to 1. These values of the picture quality algorithms in the analysis of the linear

algorithm compared to the slow start one showed that in almost all cases the picture quality was better for the linear algorithm.

This picture quality ratio of the linear algorithm compared to the slow start one is reversed under the variable bandwidth function. While the linear algorithm did, in fact, maintain a better overall picture quality, the slow start algorithm did reach a higher picture quality on different occasions especially when the bandwidth levels were increasing, as can be seen in the Figure 4.13.

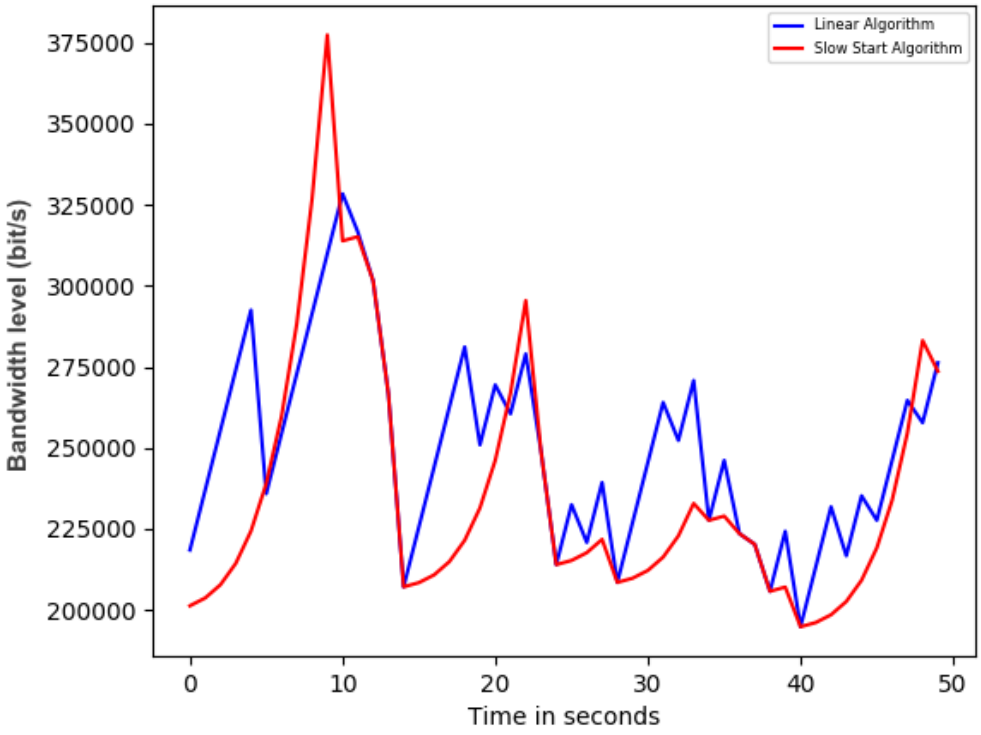

Figure 4.13: Plotted data of linear and slow start algorithm under variable bandwidth levels

As shown in the above table, the better picture to quality ratio is also true for the second bandwidth function.

Variable Bandwidth Function						
Timestamp Tested	Bandwidth Level	Bitrate Output	MSE	RMSE	PSNR	SSIM
Linear Algorithm						
00:08	406864	272864	0.105866	0.325371	9.75242	0.5336111550736803
00:17	346104	244136	0.104555	0.32335	9.80655	0.5606178375584953
00:32	265960	264004	0.104987	0.324017	9.78865	0.5571627618817595
00:41	240308	213324	0.104366	0.323058	9.8144	0.5714530271389546
Slow Start Algorithm						
00:08	406864	288292	0.105838	0.325328	9.75356	0.537687282032009
00:17	346104	210833	0.104555	0.32335	9.80655	0.5606178375584953
00:32	265960	216297	0.106042	0.32564	9.74524	0.5373872389932631
00:41	240308	196120	0.104914	0.323905	9.79166	0.55936811775337

In the above table, we can see that at the 8 and 17 second the slow start algorithm was inferior to the linear one, even though by a small margin. This is shown by the higher values of MSE and RMSE and lower values of PSNR and SSIM compared to the ones for the linear algorithm which means that the slow start algorithm has a lower perceived quality of the picture. This is reversed at the 32 and 41 seconds where the values of MSE and RMSE show that the slow start algorithm is outperforming the linear one. This is also seen in higher values of PSNR and values of SSIM closer to 1 compared to the linear algorithm. It should be noted that even though we see differences in the bitrate of the two algorithm this is not always translated to a large difference in the picture quality assessment. This may happen because of different variables in the encoding of the frame, such as an over-compression of the frame, while still keeping the same perceived quality. This happens especially when the picture has patterns which are easy to compress, such as a large portion of the picture being a single color. In both cases, the linear algorithm generally performed better in relation to the picture quality, with the exception of some cases as seen when the bandwidth level was increasing. While we only sampled parts that were indeed different, it also has to be stated that the differences were noticeable on an increasing level of bandwidth. In case of a decreasing bandwidth level, the algorithms performed almost the same because they use the same logic to decrease the bandwidth.

Streaming quality

As we predicted earlier, each algorithm introduced its own delay into the video processing. Latency-wise, the delay was unnoticeable, with an average of 170 milliseconds introduced by the encoding and decoding of the stream through the entire

stream, complying with the non-functional requirement of the prototype.

A noticeable delay was introduced when we look at the frame drops for each algorithm. Under the sinusoidal bandwidth function, the linear algorithm introduced two delays at the video stream in the form of a lag by almost 300 milliseconds at the seconds 20 and 45, with an average of 20 frames dropped. The slow start algorithm had a better performance on the total number of introduced lags, with only one lag produced during the stream, but the numbers of frames dropped were more substantial, with 70 frames on that particular occasion. The bandwidth function per se did not introduce noticeable lag as the decrease in the bandwidth level was managed by the two algorithms in a proper manner.

In the second laboratory test conducted we could see the same behavior of the two algorithms. The linear algorithm introduced a number of three delays at the seconds 25, 33, 41 and 43 compared to the slow start which introduced only one at the 48 seconds. But the difference in latency introduced by the linear algorithm was not noticeable enough with an average of 100 milliseconds compared to the slow start algorithm with 150 milliseconds, a difference that could be also seen in the overall perceived delay of the stream in the form of lags. Because the linear algorithm achieved a better coverage of the bandwidth, its streaming suffered from the sudden drops of the bandwidth, which resulted in more perceived lags in the stream as happened in ten occasions. While the slow start algorithm always maintained a smoother increase in the bitrate levels, we could only notice this behavior at four different occasions. There was a large difference between the two algorithms in the maximum level of the lag, as the slow start algorithm had almost 60% more lag than the linear one.

As we have seen in this analysis, we do not have a single silver bullet for different levels of bandwidth. While the linear algorithm did in fact cover more bandwidth than the slow start one, it did introduce more lag to the overall video stream but performed as expected in decreasing levels of bandwidth. While the slow start algorithm had better performance in the average streaming of the video data, it did suffer from low performance in increasing levels of the bandwidth.

5 Implementation Analysis and Future Work

5.1 Implementation analysis

The goal of this thesis was the creation of a video streaming solution capable of changing the quality of the stream by modifying the bitrate of the video under different bandwidth conditions. The solution had to have a latency of 200 ms at maximum and be capable of bypassing existing firewall rules imposed by different internet service providers. This solution would be integrated into drones which would cover long distances and therefore need a larger connection than a local network as part of their navigation system.

In the beginning of the project several existing adaptive bitrate technologies were considered and analyzed for their qualities, streaming properties, drawbacks, implementation difficulty, and several other factors which would satisfy the quality of the stream and fulfill the nonfunctional requirements. Nonetheless, after close inspection, we could not isolate an ABR technology capable of delivering what our strict constraint required. For this reason we decided to implement our own solution partially based on the RTP and RTCP protocols. To make real-time video streaming possible even in high resolution, such as 3840 x 2160 pixels, an embedded board (Nvidia Tx2) was chosen. The board is capable of encoding video data in the requested resolution but also capable of handling H.265 encoding for a lower bandwidth footprint of the video stream.

To implement a substitute for the existing protocols we created a streaming platform using the GStreamer framework as the core of the application. Several elements and variables of this framework were evaluated to create the video stream in a high level programming language such as Python. GStreamer was also a nonfunctional requirement as the embedded board was capable to encode video in the H.265 standard

only under this framework.

To make it possible to adapt the video bitrate to the bandwidth level without applying an existing protocol which might be filtered by a firewall, the data needed to be sent in UDP and TCP packets. To accomplish the task, Scapy was chosen and implemented as part of the Python plugin library.

To evaluate the software we decided to create our own laboratory environment, because the existing network-shaper solutions were not compatible with the Scapy library. For this reason, we decided to build a web browser automation system for an existing router which did not allow API calls. To make the testing ground as intuitive and easy to operate as possible, we implemented the testing software with ROBOT framework and Selenium.

Two main bandwidth functions were tested, one of which was artificially created and the other one used a real life dataset of different levels of bandwidth under a 3G network.

To make the adaption of the video quality to the bandwidth as fast and responsive as possible, two adaptive bitrate algorithms were implemented and then analyzed and discussed under such a testing environment. We saw the difference in behavior under different bandwidth functions, and their similarities; how the linear algorithm performed well on covering the majority of the bandwidth level while under-performing on the lower specter of the bandwidth. Also we saw how, on the contrary, the slow start algorithm performed exceptionally well when the bandwidth level had a decreasing trend, but gave not satisfactory results for an increasing bandwidth function. This makes us believe that a combination of the two algorithms would be a better solution for exploiting their strong points. In case of an increasing bandwidth level we could switch the algorithm to the linear one until we see a decrease on which we could again use the slow start.

5.2 Future work

Our solution is only a prototype of a full implementation of the presented solution. While our implementation can be used to adapt the bitrate to available bandwidth it is still based on the RTP protocol. To be able have a video stream which can pass through different network firewalls we need to remove RTP and for this reason we might need to create an implementation of it which would only use UDP packets at its base. The

role of RTP is to provide a means to analyze jitter compensation and packet loss, but most importantly, out of order delivery which is common during UDP video streaming. To be able to simulate this behavior the payload of the UDP packet could be used as a wrapper to be injected with custom data as shown in Figure 5.1:

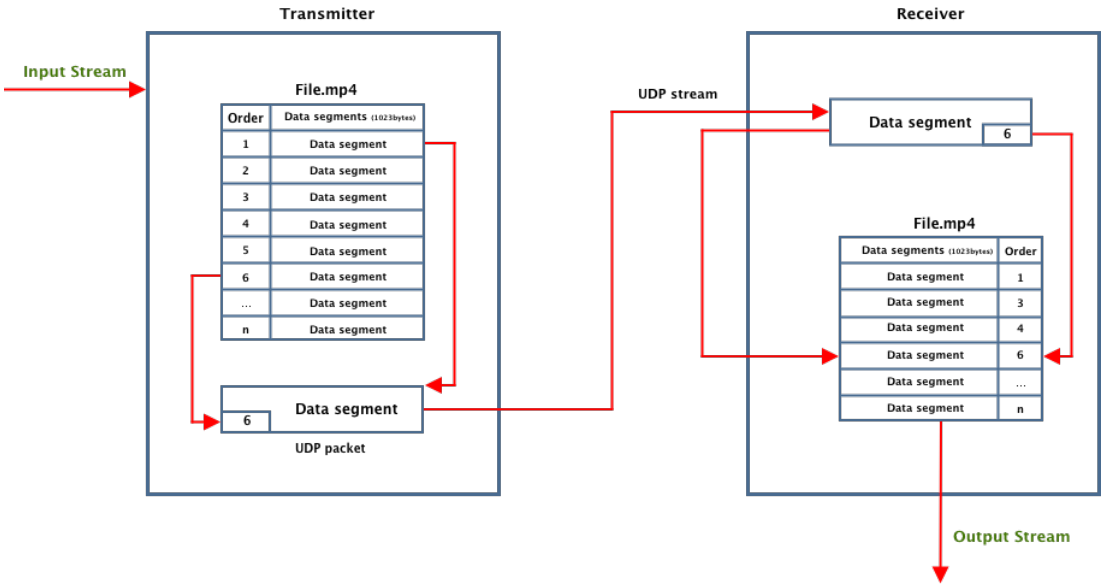

Figure 5.1: Construction and deconstruction of a timestamp segment

We could construct video data from the input stream by joining together the video data with a timestamp data segment. The payload could then be streamed over the receiver where it could be reconstructed by maintaining the original order of the packets. The timestamp could be used to compute jitter analysis and out-of-order delivery, and in conjunction with our previous RTCP re-implementation. As the last step to make the application fully operational, an advanced encryption algorithm and a mechanism which would allow us to change the port number of the video stream could be added, which could certainly give all the advantages of RTP and RTCP without the drawbacks.

BIBLIOGRAPHY

[1] Wikipedia contributors. Adaptive streaming overview, 2011. [Online; accessed 28 July 2011]. URL: https://commons.wikimedia.org/wiki/File:Adaptive_streaming_overview_daseddon_2011_07_28.png#/media/File:Adaptive_streaming_overview_daseddon_2011_07_28.png.

[2] Adaptive streaming. [Online; accessed 1 January 2019]. URL: https://opentv.nagra.com/player/adaptive-streaming.

[3] Wikipedia contributors. Mpeg-dash, an overview. URL: https://1yy04i3k9fyt3vqjsf2mv610yvm-wpengine.netdna-ssl.com/wp-content/uploads/2015/07/EDC_DASH-80.png.

[4] Jan Ozer. What is hls (http live streaming), 2011. [Online; accessed 14 Octover 2011]. URL: http://www.streamingmedia.com/Images/ArticleImages/ArticleImage.11612.jpg.

[5] Wes Simpson. Scaling up for scalable video coding, 2011. [Online; accessed 5 April 2011]. URL: https://www.tvtechnology.com/.image/c_limit%2Ccs_srgb%2Cfl_progressive%2Cq_auto:good%2Cw_450/MTUzNzQwNTMwODQyMDg1MTQ0/image-placeholder-title.jpg.

[6] A new tool to test the ip network performance. [Online; accessed 9 January 2019]. URL: https://www.researchgate.net/figure/RTP-header_fig4_315479711.

[7] Rtp, rtcp. [Online; accessed 10 January 2019]. URL: https://m.blog.naver.com/thorong/70147853857.

[8] Selenium home page. [Online; accessed January 2019]. URL: https://www.seleniumhq.org.

[9] Bradley Mitchell. The range of a typical wi-fi network. URL: https://www.lifewire.com/range-of-typical-wifi-network-816564.

[10] All you need to know about frequencies on which drones operate. URL: https://www.jammer-store.com/drones-frequencies.html.

[11] Cam Cullen. Sandvine releases 2018 global internet phenomena report, 2018. [Online; accessed 2 October 2018]. URL: https://www.sandvine.com/press-releases/sandvine-releases-2018-global-internet-phenomena-report.

[12] F.A. López-Fuentes. P2p video streaming strategies based on scalable video coding. *Journal of Applied Research and Technology*, 13(1):113 – 124, 2015. URL: http://www.sciencedirect.com/science/article/pii/S1665642315300109, doi:https://doi.org/10.1016/S1665-6423(15)30010-9.

[13] Rtp: A transport protocol for real-time applications. RFC 3550, RFC Editor, July 2003. URL: https://tools.ietf.org/html/rfc3550.

[14] Emmanuel Thomas. Enhancing mpeg dash performance via server and network assistance, 2017. [Online; accessed 29 March 2018]. URL: https://www.ibc.org/delivery/enhancing-mpeg-dash-performance-via-server-and-network-assistance/1027.article.

[15] NVIDIA. *ACCELERATED GSTREAMER USER GUIDE*. NVIDIA. URL: https://developer.download.nvidia.com/embedded/L4T/r28_Release_v2.0/DP/Docs/Jetson_TX1_and_TX2_Accelerated_GStreamer_User_Guide.pdf.

[16] Christopher Mueller. Mpeg-dash vs. apple hls vs. microsoft smooth streaming vs. adobe hds, 2015. [Online; accessed 29 March 2015]. URL: https://bitmovin.com/mpeg-dash-vs-apple-hls-vs-microsoft-smooth-streaming-vs-adobe-hds/.

[17] Rtp: A transport protocol for real-time applications. RFC 3550, RFC Editor, July 2003. URL: https://tools.ietf.org/html/rfc3550#section-5.1.

[18] Some frequently asked questions about rtp. [Online; accessed 8 January 2019]. URL: https://www.cs.columbia.edu/~hgs/rtp/faq.html#lite.

[19] Philippe Biondi and the Scapy community Revision. *Scapy User-Guide*. URL: https://scapy.readthedocs.io/en/latest/.

[20] Charles-Francois Natali. *Ntplib*. URL: https://pypi.org/project/ntplib/.

[21] The most popular language for machine learning is ... [Online; accessed January 2018]. URL: https://www.ibm.com/developerworks/community/blogs/jfp/entry/What_Language_Is_Best_For_Machine_Learning_And_Data_Science?lang=en.

[22] Gsth264parser. Documentation. URL: https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-bad-libs/html/gst-plugins-bad-libs-h264parser.html#gst-plugins-bad-libs-h264parser.description.

[23] queue. Documentation. URL: https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer-plugins/html/gstreamer-plugins-queue.html.

[24] Tcp congestion control. RFC 5681, RFC Editor, January 2019. URL: https://tools.ietf.org/html/rfc5681.

[25] *Wireless AC750 Dual Band Router*. [Online; accessed February 2019]. URL: https://eu.dlink.com/uk/en/-/media/consumer_products/dir/dir-809/manual/dir_809_a3_manual_v1_01_eu.pdf.

[26] Robot home page. [Online; accessed January 2019]. URL: https://robotframework.org.

[27] Haakon Riiser Paul Vigmostad Carsten Griwodz Pål Halvorsen. Commute path bandwidth traces from 3g networks: Analysis and applications.

[28] Z. Wang. Applications of objective image quality assessment methods [applications corner]. *IEEE Signal Processing Magazine*, 28(6):137–142, Nov 2011. doi:10.1109/MSP.2011.942295.

[29] J. D. Ruikar, A. K. Sinha, and S. Chaudhury. Image quality assessment algorithms: Study and performance comparison. In *2014 International Conference on Electronics and Communication Systems (ICECS)*, pages 1–4, Feb 2014. doi:10.1109/ECS.2014.6892744.

[30] Imagemagic home page. [Online; accessed January 2019]. URL: https://www.imagemagick.org.

[31] Ssim-pil home page. [Online; accessed January 2019]. URL: https://pypi.org/project/SSIM-PIL/.

[32] Image quality metrics. [Online; accessed January 2019]. URL: https://www.mathworks.com/help/images/image-quality-metrics.html.

[33] Richard Dosselmann and Xue Dong Yang. A comprehensive assessment of the structural similarity index. *Signal, Image and Video Processing*, 5:81–91, 01 2010. doi:10.1007/s11760-009-0144-1.

[34] Thrasyvoulos N. Pappas Alan C. Brooks, Xiaonan Zhao. Structural similarity quality metrics in a coding context: exploring the space of realistic distortions. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 17:i –8, 2008.

A Appendix A

A.1 GStreamer server - example

An example of the construction of a GStreamer pipeline and the concatenation of the different elements is given below:

```
#!/usr/bin/env python2
import os
import socket
import time
import logging
import gi



gi.require_version('Gst', '1.0')
from gi.repository import GObject, Gst
from gi.repository import GLib
# scheduler library for the application
from apscheduler.schedulers.blocking import BlockingScheduler
# scrap packet to count them library
from scapy.all import *
# fetch O-time class
from Fetchtime import SyncTime
# required for the scheduler
logging.basicConfig()
GObject.threads_init()
Gst.init(None)

# address of the client
TCP_IP = ''
TCP_PORT = 3333
BUFFER_SIZE = 1024  # Normally 64, less for faster response

# fetch online time
SyncTime.try_to()


class Sender:
```

```
  def __init__(self):
      # Create GStreamer pipeline
      self.pipeline = Gst.Pipeline()

      # Create bus to get events from GStreamer pipeline
      self.bus = self.pipeline.get_bus()
      self.bus.add_signal_watch()
      self.bus.connect('message::error', self.on_error)

      # source
      self.src = Gst.ElementFactory.make('nvcamerasrc', None)
      self.src.set_property('fpsRange', "30 30")
      self.src.set_property('intent', 3)

      # video
      #self.srccaps = Gst.Caps.from_string(
          #"video/x-raw(memory:NVMM), width=(int)800, height=(int)800, format=(
              string)I420, framerate=(fraction)30/1")
      self.srccaps = Gst.Caps.from_string(
          "video/x-raw(memory:NVMM), width=(int)800, height=(int)800, format=(
              string)I420, framerate=(fraction)30/1")

      # conversion
      self.conversion = Gst.ElementFactory.make('nvvidconv', None)
      self.conversion.set_property('flip-method', 6)

      # encoder
      self.encoder = Gst.ElementFactory.make('omxh264enc', None)
      # self.encoder.set_property('low-latency', 1)
      self.encoder.set_property('control-rate', 2)
      self.encoder.set_property('bitrate', 40000)
      print(self.encoder.get_property('bitrate'))

      # stream
      self.stream = Gst.Caps.from_string("video/x-h264, stream-format=(string)byte-
          stream")
      self.rtp = Gst.ElementFactory.make('rtph264pay', None)
# for controlling the size of the rtp packets
      #self.rtp.set_property('mtu',100)
      self.parse = Gst.ElementFactory.make('h264parse', None)
      self.udp = Gst.ElementFactory.make('udpsink', None)
      self.udp.set_property('host', '192.168.11.35')
      self.udp.set_property('port', 5001)
      self.udp.set_property('auto-multicast', False)
      # add time
      self.text = Gst.ElementFactory.make('textoverlay', None)
      self.text.set_property('text', "Time:")


      # Add elements to the pipeline
      self.pipeline.add(self.src)
      self.pipeline.add(self.conversion)
```

```
        self.pipeline.add(self.encoder)
        self.pipeline.add(self.parse)
        self.pipeline.add(self.rtp)
        self.pipeline.add(self.udp)

    # we add the text overlay here
        self.pipeline.add(self.text)


        # link them together
        self.src.link_filtered(self.conversion, self.srccaps)
        self.conversion.link_filtered(self.encoder, self.srccaps)
        self.encoder.link_filtered(self.parse, self.stream)
        self.parse.link(self.rtp)
        self.rtp.link(self.udp)



      # run the program
      def run(self):


        # create socket and accept connections
        def connect(host, port):
            address = (host, port)
            server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            server_socket.bind(address)
            server_socket.listen(5)
            print "Listening for client . . ."
            global conn
            conn, address = server_socket.accept()
            print "Connected to client at ", address

        connect(TCP_IP, TCP_PORT)



        # Start Gstreamer playing
        self.pipeline.set_state(Gst.State.PLAYING)


        # take the time for synch
        time_start = conn.recv(BUFFER_SIZE)
        print "h"
        print " The program will start to monitor the packets at: ", time_start

        # method to count the packets
        def count_udp():
            # initialize and reset the count var
            global count
            count = 0
```

```
# function called for every packet
# count how many packets are filtered
def pkt_callback(pakets):
    global count
    count += 1

# conf for raspberry pi
packets = sniff(lfilter=pkt_callback, filter='udp and host 192.168.11.48
    and port 5001', store=0, timeout=1)




# method to get the last packet in case of a concatination when the
    servers is down
def last_packet(received):
    final_regex=re.findall('\d+',received)[-1]
    final_regex=int(final_regex)
    return final_regex

# receive packet
rec_packets=last_packet(conn.recv(BUFFER_SIZE))

# if packet is 0
if not rec_packets:
    # wait for a reconnection
    connect(TCP_IP, TCP_PORT)
    time_start = conn.recv(BUFFER_SIZE)

    # start everything
    schedule_run.add_job(start_scheduler, 'date', run_date=time_start,
        args=[time_start])
    rec_packets = '0'


    # the difference between packets sent and received
difference_sent_received_packets = count - int(rec_packets)


# method to lower the bitrate when the bandwidth lowers
# todo: just a friendly reminder for you to actually use this

def lowerBitrate(created, received,bitrate):
    diffpacket=created-received
    diffPercentage=diffpacket/created
    diffBitrate=bitrate*diffPercentage
    return diffBitrate

# test the output
print str(count) + " - " + str(rec_packets) + " = " + str(
    difference_sent_received_packets)
```

```
            bitrate2 = self.encoder.get_property('bitrate')

            rec_packets = int(rec_packets)
            print "difference:", difference_sent_received_packets


            count = int(count)
            count = abs(count)
            print count

            if (difference_sent_received_packets < count*0.2) and (bitrate2
                <=4294967295) :
                print difference_sent_received_packets - count*0.2
                bitrate2= bitrate2+100000

            elif (difference_sent_received_packets > count*0.2) and (bitrate2 >
                200000):
                #bitrate2 = bitrate2 - 200000
                bitrate2=lowerBitrate(count,rec_packets,bitrate2)

            bitrate2=int(bitrate2)
            bitrate2=abs(bitrate2)
            self.encoder.set_property('bitrate', bitrate2)
            gititi = self.encoder.get_property('bitrate')
            print " Current bitrate:" + str(gititi)
            # recursive running of the count_udp method
            schedule_run.add_job(count_udp)

        # create schedule object
        schedule_run = BlockingScheduler()

        # function to start for the first time count_udp method
        def start_scheduler(datetime):
            print "Program started in", datetime
            schedule_run.add_job(count_udp)

        # start everything
        schedule_run.add_job(start_scheduler, 'date', run_date=time_start, args=[
            time_start])
        schedule_run.start()



    def on_error(self, bus, msg):
        print('on_error():', msg.parse_error())


if __name__ == '__main__':
    sender = Sender()
    sender.run()
```