TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

OSMAN FIRAT AKANDERE 166812IVSM

TELEMETRY ON ROBOT OPERATING SYSTEM BASED SELF-DRIVING VEHICLES

Master's thesis

Supervisor: Priit Järv
MSc.

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

OSMAN FIRAT AKANDERE 166812IVSM

Telemeetria operatsioonisüsteemil ROS põhinevatele isesõitvatele autodele

Magistritöö

Juhendaja:	Priit Järv
	MSc.

Tallinn 2019

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Osman Firat Akandere

13.05.2019

Abstract

The trend of building autonomous vehicles is constantly increasing. Many commercial car producers and educational institutions are building their own self-driving vehicles.

It is crucial for the development teams to analyze the data, that the car uses for decision making, in real time while the car is running to troubleshoot issues. Although this seems to be similar to video streaming like Internet television, it is actually quite challenging due to the limited resources of the car and also bandwidth of car's Internet connection can change depending of the car's location.

We analyzed various methods of video and data streaming in this paper to find the most suitable method for a self-driving vehicle with a cellular connection. Later, we made experiments on our proposal and finally we implemented a telemetry system for Tallinn University of Technology's self-driving bus and made a demo client that can be viewed from web browsers.

This thesis is written in English and is 40 pages long, including 8 chapters and 23 figures.

Annotatsioon

Autonoomsete sõidukite välja töötamine on pidevalt laienemas. Mitmed autotootjad ja haridusasutused ehitavad oma isesõitvaid sõidukeid.

Sõidukite arendusmeeskondade jaoks on oluline reaalajas jälgida andmeid, mida sõiduk kasutab otsuste tegemiseks, et arenduse käigus probleeme lahendada. Ehkki sarnased lahendused on käigus näiteks Interneti vahendusel televisiooni edastamisel, on see ülesanne raskendatud auto piiratud ressursside ja sidekanalite mahu tõttu. Näiteks sõltub auto Internetiühenduse ribalaius auto asukohast.

Töös analüüsitakse mitmeid video- ja andmeedastusmeetodeid, et leida sobivaim lahendus mobiilsidet kasutava isesõitva sõiduki jaoks. Leitud meetodeid katsetatakse ning implementeeritakse telemeetria lahendus Tallinna Tehnikaülikooli isesõitvale bussile. Lahendus sisaldab näidisrakendusena veebiklienti, mis võimaldab reaalajas edastatavaid andmeid näha.

Väitekiri on inglise keeles ja on 40 lehekülge pikk. Väitekiri sisaldab 8 peatükki ja 23 joonist.

List of abbreviations and terms

IEEE	The Institute of Electrical and Electronics Engineers
TTU	Tallinn University of Technology
Lidar	Light Detection and Ranging
GPS	Global Positioning System
HSDPA	High Speed Downlink Packet Access
MBPS	Megabits per second
GPU	Graphical processing unit
CPU	Central processing unit
4G	4th generation of broadband cellular network technology
LTE	Long-Term Evolution
HAS	HTTP Adaptive Streaming
RTT	Round-trip time
3D	3 dimensional
ROS	Robot operating system
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
RTP	Real-time Transport Protocol
RTCP	RTP Control Protocol
SSRC	Synchronization source identifier
KBPS	Kilobits per second

Contents

1 Introduction **10**

1.1 Problem Statement 11

1.2 Contribution 12

1.3 Organization of the Thesis 12

2 Literature Overview **14**

3 Background **17**

3.1 ROS (Robot Operating System) 17

3.2 Autoware 17

3.3 Iseauto 18

4 Methodology **19**

4.1 Software 19

4.1.1 GStreamer 19

4.1.2 Latency Clock 19

4.1.3 Janus 20

4.1.4 RViz 20

4.1.5 RViz Camera Stream 20

4.1.6 OpenCV 21

4.1.7 Python 21

4.2 Protocols and Techniques 21

4.2.1 Transport Protocol 21

4.2.2 Stream Synchronization 23

4.2.3 Delivery of Stream 24

4.3 Implementation 25

5 Experiments **28**

5.1 Experiment Setup 28

5.2 Results 28

5.2.1 Effect of FPS on Latency 28

5.2.2 Limitation of Resources 30

5.2.3 Simultaneous Streaming without Synchronization 31

5.2.4 Simultaneous Streaming with Synchronization 33

6 Discussion **36**

7 Future Work **37**

8 Summary **38**

List of Figures

1 A photography of Iseauto. 12
2 Demo of Web UI with synthetic data. 13
3 The diagram of HTTP adaptive streaming. 15
4 A frame of a video with latency clock overlay. 20
5 Diagram of a TCP packet node. 22
6 Diagram of a UDP datagram node. 22
7 Sequence diagram of a TCP communication. 22
8 Sequence diagram of a UDP communication. 23
9 Diagram of an RTP. 24
10a Unicast video delivery 26
10b Multicast video delivery 26
11 Telemetry system that runs in the vehicle. 26
12 Telemetry system that runs in the server. 27
13 Latency comparison of 30 FPS and 60 FPS video streaming. 29
14 Latency comparison of 30 FPS and 60 FPS video streaming. 29
15 Latency of 1080P media with 60 FPS. 31
16 Latency of 2 streams with 1080P media with 30 FPS. 31
17 Latency comparison of two streams with 768P media with 30 FPS. . . . 32
18 Latency comparison of two streams with 480P 60FPS and 1080P 30FPS. 32
19 Latency comparison of three simultaneous stream with 768P, 30FPS and 512 kbps. 33
20 Latency comparison of two simultaneous stream with 768P, 30FPS and 512 kbps 33
21 Latency comparison of two streams with 480P 60FPS and 1080P 30FPS. 34
22 Latency comparison of three simultaneous stream with 768P, 30FPS and 512 kbps. 35

1 Introduction

The development of self-driving vehicles and autonomous cars has increased notably during the recent years. According to the predictions of The Institute of Electrical and Electronics Engineers (IEEE), 75% of the vehicles that are in the traffic will be fully autonomous by 2040 [1]. Therefore, many of the car manufacturers and educational institutions are developing their autonomous vehicles. These vehicles are equipped with several extra sensors, compared to the ordinary vehicles, so that it can compensate the requirement for a driver. These sensors can be listed as:

- LiDAR (Light Detection and Ranging): A LiDAR scanner sends a laser pulse to a point in space and measures the time span of the reflection. By changing the direction of laser rapidly, a LiDAR scanner can gather data and create a 3D cloud-point of the surrounding area.
- Camera: The images from cameras can be used to identify the objects and the location of the vehicle by using image processing techniques.
- GPS (Global Positioning System): A GPS sends signals to satellites and calculates the latitude and longitude based on how long it takes for the response to arrive from the satellites.

It is crucial for the developers to be able to debug these sensor data from the vehicle, so that they can troubleshoot the issues. One of the most demanded debugging and analyzing technique is visualizing and watching the sensor data from vehicle in real time. Although it seems to be just a data transmission and media streaming task, it has its own challenges.

A big portion of these vehicles are connected to the Internet via cellular modems. Although, the current technologies for the cellular connectivity, such as HSDPA or 4G, are promising by having upload speed up to 8 mbps, the speed depends on the distance of the car to the service provider. This problem brings the bandwidth issue. The data that is transmitted from the vehicle should not exceed the available connection speed. Therefore, the transmitted data should still be understandable but as small as possible.

Self-driving vehicles uses graphical card heavily for image processing. Moreover, there are many calculations done per second to analyze the data from sensors so that the vehicle can find its route and understand the obstacles. These calculations are CPU demanding. Therefore, the resources for the telemetry are limited.

1.1 Problem Statement

Tallinn University of Technology is one of the institutes that develops their own self-driving car successfully. The vehicle is named *Iseauto* and can be seen in Figure 1. The vehicle has 6 people capacity and is equipped with various sensors, such as cameras, lidars and short-distance radars [2]. The car is using Robot Operating System (ROS) and AutoWare as its core software. All sensor data can be acquired by making requests to ROS' so-called *topics* which can be described as a message bus with a name.

Each ROS topic has a unique path as identification. The message data can be read with this unique path, either through command line or with ROS' libraries for programming languages. But still, the data is not ready for stream and the topic does not push new data to the requester. Therefore, the data has to be requested with a frequency. Text data, such as GPS location, speed of the vehicle or the heading can be directly transmitted over network. However the lidar and camera data requires processing of data for visualization.

The Lidar data is structured in a data type called *Point Cloud* or *PointCloud2* as it's called in ROS. A point cloud is an array of objects where each object represents a point in three-dimensional space with several other attributes. Depending on the lidar hardware, the point cloud data of a single moment can get too large to be transmitted over 4G network.

The camera data is provided by ROS' *Image* data type. This data is held in an array as a mathematical representation of the camera output to make analyses and image processing over it. However, this data should be transformed and encoded with a media encoder before streaming over network. Furthermore, streaming camera output frame-by-frame is not efficient, therefore a local buffer needs to be created.

There are multiple cameras and multiple lidar sensors in the vehicle and all data from these sensors are crucial. Therefore, streaming multiple media simultaneously is required and these streams should be synchronized to minimize the latency among the streams.

Figure 1: A photography of Iseauto.

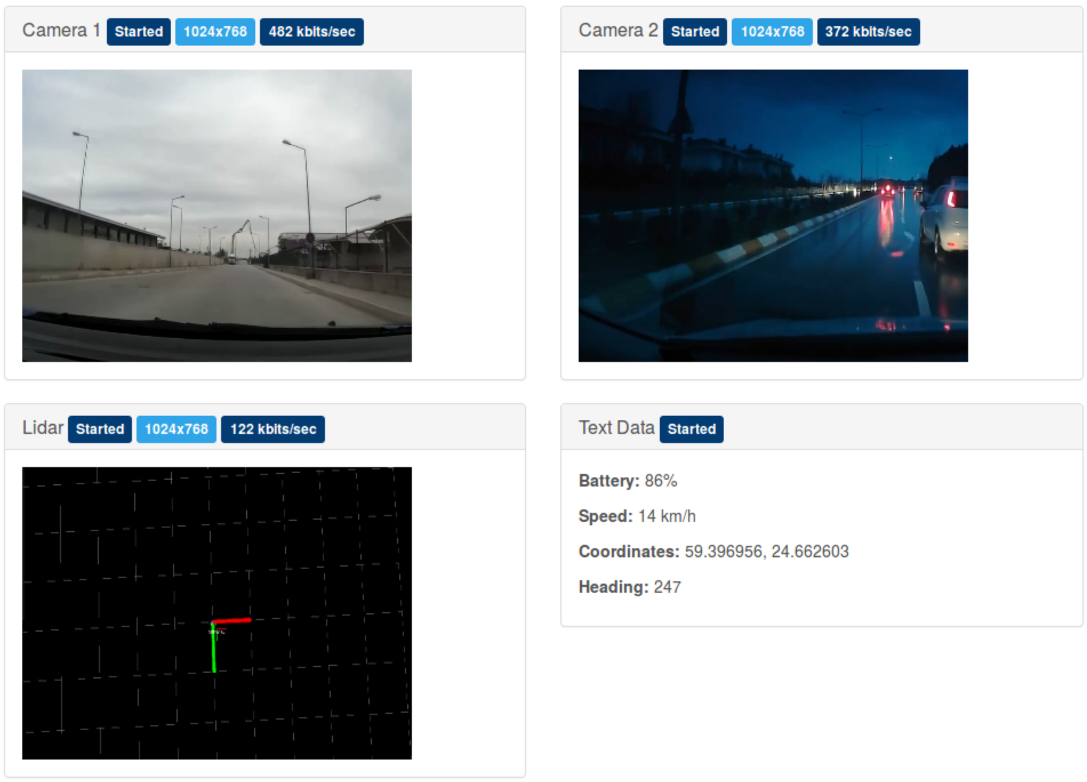
1.2 Contribution

In this paper, we reviewed the recent methods of media streaming techniques and their efficiencies under 4G network connection. We proposed a solution that can make streams from ROS topics with camera and text-based output and showed that it's possible to integrate our system with other ROS plugins in order to locally visualize the lidar data and then stream it with the proposed solution. Later, we made experiments with different configuration sets to analyze the system's performance. Finally, we prepared a demo where the sensor data can be viewed from web browsers. A screen shot from the demo can be seen in Figure 2.

1.3 Organization of the Thesis

This thesis is divided into following chapters:

- **Literature Overview:** This chapter gives an overview of the literatures that has been reviewed for this thesis.
- **Background:** This chapter gives a brief information about sensors and the software used in the vehicle's computer.
- **Methodology:** This chapter introduces the techniques and softwares that are available for this thesis and criticizes them to find the most suitable one for the current



Battery: 86%

Speed: 14 km/h

Coordinates: 59.396956, 24.662603

Heading: 247

Figure 2: Demo of Web UI with synthetic data.

problem.

- **Experiments:** The results of the experiments with different configuration are presented and analyzed.
- **Discussion:** The overall results of the experiments have been discussed to better understanding.
- **Future Work:** Potential improvements about the proposed system are suggested.
- **Summary:** The outcome of this thesis is described.

2 Literature Overview

There have been numerous study on delivering video stream since the first decades of Internet. However, live video stream is in its early stages. The delivery of real time video is used in domains such as, entertainment platforms, video conferencing, Internet televisions and online learning platforms [3].

The transmission of a live video stream is different than the data communication because the delayed data in video stream will be dropped and is not useful for the video decoder [4]. TCP/IP can introduce delays with its retransmission and it is not acceptable for live video streaming. Therefore, UDP is usually chosen as the transportation protocol for video/audio streaming [5].

Multicast real-time video streaming uses point-to-multipoint transmission which can help the sender to have bandwidth efficiency. However, this bandwidth efficiency comes with a cost. Since the bitrate of the stream is the same for all receivers, the multicast systems cannot meet the requirements of all receivers, with varying connection qualities, in terms of quality of service. Therefore, the receivers with bandwidth lower than the stream's, can experience packet loss or delay.

To solve the technical issue above, [3] proposes a framework that consist of the main components, *congestion control* and *error control*.

1. *Congestion Control:* Excessive loss and delay are usually caused by the network congestion. Therefore, the framework proposes a congestion control to prevent packet loss and delay. One of the congestion control mechanism is called *rate control* [6]. Rate control tries to adjust bitrate of the video stream depending on the available bandwidth in order to decrease the packet loss and delay.

2. *Error Control:* Although the congestion control tries to decrease packet loss, it is still unavoidable. Error control tries to avoid glitches in the video in case of a packet loss.

To solve the issue with clients with varying bandwidth, HTTP adaptive streaming is proposed [7, 8]. As it can be seen in Figure 3, the input video is segmented into multiple streams by encoding it with multiple quality levels where each segment is usually 2-10 seconds [9]. Although there has been many HAS approaches proposed, there are

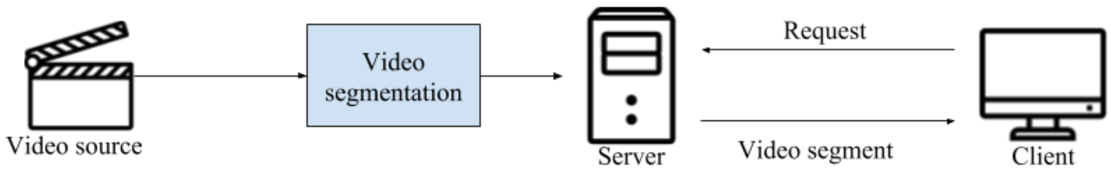


Figure 3: The diagram of HTTP adaptive streaming.

still handicaps. The clients still experience freezing in 27% of the video sessions [10], especially if the client's network has rapid bandwidth changes.

[9] proposes an approach with HEVC encoding to solve the issue above. H.265/HEVC is a video encoding method that provides twice the compression efficiency of the previous encoding method H.264/AVC [11]. It also proposes to decrease the segment length to sub-second so that download time of an individual segment gets shorter and eventually response to the bandwidth changes gets rapid. Unfortunately, this means the time spent for round-trip time (RTT) increases since every segment is retrieved with an HTTP request. This is a problem for clients with cellular connectivity since RTT is usually between 33ms to 857ms with cellular connection. Therefore, [9] proposes to use HTTP/2 server push to eliminate RTT. The server push is a new feature announced in HTTP/2 that allows the server to push data to client without a request from the client [12, 13]. As a result of this approach, the video quality is increased by 7.5%, the clients have experienced lower freeze time by 50.4% and the delay is reduced compared to HTTP/1.1 [9].

Although the proposition in [9] improves the packet delay and video quality issues, it's still weak for packet loss. To solve this issue, the Priority Encoding Transmission was introduced in [14], which prioritizes encoding segments and protects by different channel codes based on their priority.

Using LiDAR (Light Detection And Ranging) scanners for mapping the terrain elevation has been quite popular with the demand of high accurate data [15]. An aircraft-mounted LiDAR scanner can produce 10 points per meter square with 15 centimeter elevation accuracy [16], which means millions of points for a couple of hundreds of meter square area.

Rendering LiDAR data is not easy considering the amount of data for a single environment. Unlike the polygon models where the textures are used to represent the surfaces between the vertices of the polygons, LiDAR data is visualized with millions of colored points. Another issue is that the points may get overlapped which can cause overdrawing. When the user zooms out the drawing, the number of points remains the same but per area increases and may cause overlapping.

A progressive real-time rendering technique is proposed in [17] to solve this issue. The proposed technique only renders an amount of points that the GPU can hold in its memory. The method limits the number of points, so that it consumes less resources.

The remote visualization of LiDAR data is not easy due to large file size. [16] proposes an optionally distributed web server which can send the points for a chosen detail.

3 Background

3.1 ROS (Robot Operating System)

Robot Operating System is an open-source software system designed for building robots. Although it is not an operating system itself, it is designed to meet the expectations from an operating system, such as; low-level device control, hardware abstraction, message-passing among processes and package management. It also provides tools and bindings for several programming languages to build and run the code across multiple computers [18].

Robot Operating System has three levels of concepts. Filesystem level, Computation Graph level and Community level. The filesystem concept covers ROS resources that requires disk involvement. Computation Graph level is a data processing concept where ROS processes are connected with peer-to-peer network. ROS Community level concepts define the resources that allow third party communities to exchange information. Only Computation Graph level is related to the work that is done in this thesis. This level implements *Nodes* and *Topics* concepts, which are the interest of this work.

A *Node* is a process that performs computation. Each node is responsible for one task and then multiple nodes are combined together into a graph by communicating with each other. The communication is done by *Topics*. *Topics* are named buses where each node publish messages.

3.2 Autoware

Iseauto uses Autoware as its core controlling software. It is an open source software that is built on top of Robot Operating System (ROS) [19]. Autoware is an "all-in-one" software for self-driving vehicles. It is primarily developed for urban cities but can also cover highways, freeways and geofenced areas.

The system in this thesis is actually a *Node* that uses topics provided by Autoware which publishes information about camera outputs, lidar data and several other data about the car, such as, speed, heading and location.

3.3 Iseauto

Iseauto is the first self-driving vehicle project in Estonia that is developed by Tallinn University of Technology (TTU) and Silberauto AS. The car is limited to 10-20km/h due to the fact that it is designed to operate inside the campus area. The goal of the university with this project is to increase competence in the field and provide interesting projects for the engineering students. The car's computer is running Ubuntu version 16.04 as the operating system and uses the following sensors to observe the environment and for localisation, mapping and navigation [20]:

- LiDAR Velodyne VLP-16 x2,
- Cameras x8,
- Ultrasonic distance sensors x16,
- A short-distance radar,
- An IMU sensor,
- An RTK-GNSS.

4 Methodology

In this section, we briefly describe the tools and techniques and explain why we chose them.

4.1 Software

4.1.1 GStreamer

GStreamer is a popular and widely used, pipeline-based multimedia framework engine developed for Linux. It's developed in C programming language and based on the Glib Object System. The GStreamer's pipeline-based design allows the developers to create various kinds of multimedia tools, including video or audio encoders, media players, audio or video playback, streaming or capturing video or audio from web camera, microphone or desktop. GStreamer currently supports FreeBSD, NetBSD, OpenSolaris, Android, Mac OS X and Microsoft Windows and provides bindings for programming languages such as C, Python, Perl, Vala, GNU Guile and Ruby. GStreamer is licensed under GNU Lesser General Public License.

GStreamer is used to run our experiments and stream data from the autonomous vehicle. GStreamer is chosen for these purposes because it is a low level library and allows any kind of property to be set. Moreover, its command line tool helps rapid prototyping without need to code.

4.1.2 Latency Clock

Latency clock is a Gstreamer overlay plugin that is developed by William Manley in 2016. It provides two GStreamer pipeline elements, *timestampoverlay* and *timeoverlayparse*. *timestampoverlay* draws a timestamp onto video with black and white squares that are 8 pixels. The timestamp is in 64-bit nanoseconds and it is not human readable. Therefore, *timeoverlayparse* parses the timestamp and calculates the delay. It's recommended to have the same network time protocol (NTP) server synchronized on the both client and server. Latency clock is used to measure the latency of the streams in this paper. A sample frame from a video with timestamp overlay can be seen from Figure 4.

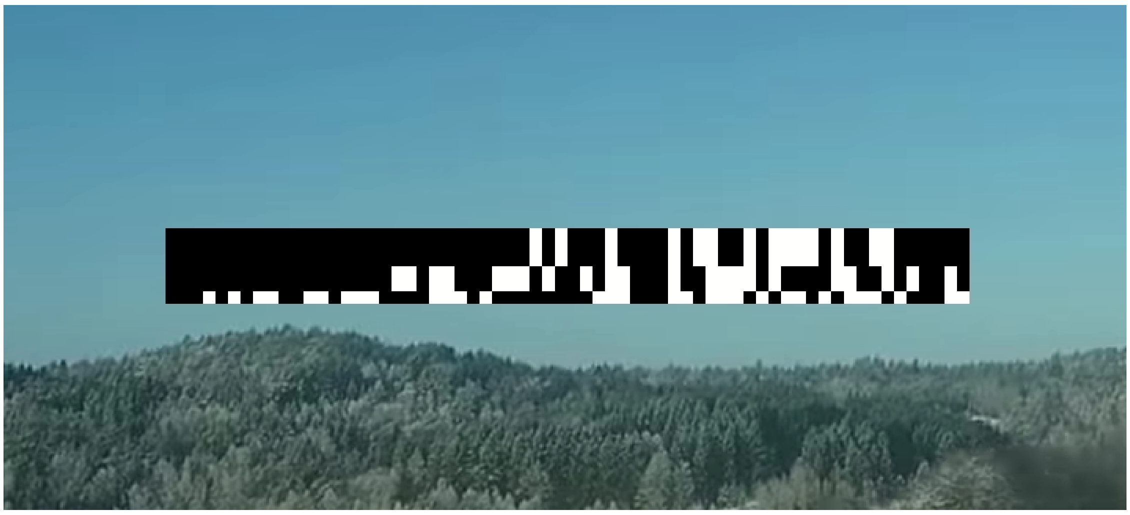


Figure 4: A frame of a video with latency clock overlay.

4.1.3 Janus

Janus is a WebRTC server for general purposes, created by MeetEcho. It's developed with C language and has plugin system. The core software only provides media communication with browser, relaying RTP/RTCP, message exchange with JSON. It also provides JavaScript, RESTful and websocket API. We use Janus server to stream data from vehicle and use its JavaScript library for the web page where the users can see the visualization of data.

4.1.4 RViz

RViz stands for ROS visualization and is a 3D visualization tool for displaying state information and sensor data from ROS. It can show real-time visualization of the sensor values that are streamed in ROS topics, including camera, sonar or LiDAR data as well. RViz is used for visualizing LiDAR data from the vehicle and then stream it to Janus through our ROS streaming node.

4.1.5 RViz Camera Stream

RViz Camera Stream is a plugin for RViz. It adds a camera element to RViz which can be placed in RViz's viewpoint. The camera element can be placed anywhere in the three dimensional space and its view angle and zoom can be configured. The view from this camera element is then published through a ROS topic and multiple camera elements can be placed as well. The RViz Camera Stream is used in this project to connect visualization of Lidar data where the visualization is done locally by RViz.

4.1.6 OpenCV

OpenCV (Open Source computer vision) is a library that is developed for real-time computer vision and machine learning. It is BSD-licensed and provides bindings for C++, Python, Java, MATLAB and supports Windows, Linux Android and Mac OS operating systems.

ROS' *Image* data type does not provide output that is ready to be buffered for video streaming but it provides a bridge for OpenCV. OpenCV is used in this project for transcoding Image data to another digital data that is acceptable for video streaming.

4.1.7 Python

The software tools that are used provides bindings for C++ and Python programming languages. Although C++ is a more robust and rapid programming language compared to Python, Python is chosen because its dynamically-typed design helps to create prototypes faster, change them easily and write readable code.

4.2 Protocols and Techniques

4.2.1 Transport Protocol

The widely-used lower-layer transport protocols for media streaming includes UDP and TCP. Both of the protocols have similar functionalities. UDP and TCP can multiplex streams from multiple data sources. Both of the protocols introduces an error control by adding a checksum to each packet. If a data corruption is detected on receiver-side, UDP discards the packet so that the corrupted packet does not end in upper layer. The diagram of a UDP datagram node can be seen in Figure 6. TCP protocol, on the other hand, retransmits the corrupted packet to ensure that the receiver gets all the packets correctly. Therefore, TCP provides more reliable transmission compared to UDP. Moreover, TCP implements a flow control that prevents overflowing the receiver's buffer and a congestion control that prevents network congestion due to sending too many packets at once, meanwhile UDP does not support both of these mechanisms[21]. The diagram of a TCP packet node can be seen in Figure 5.

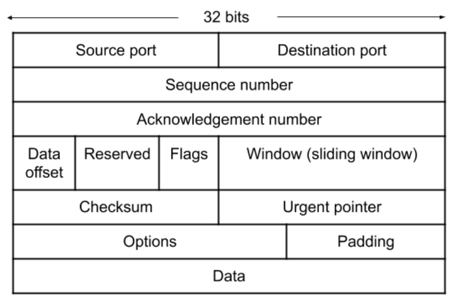


Figure 5: Diagram of a TCP packet node.

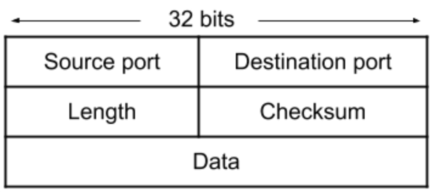


Figure 6: Diagram of a UDP datagram node.

As expected, TCP may introduce delays due to the retransmissions in case of data corruption. Furthermore, TCP provides packet delivery guarantee by sending acknowledgement requests in both ways, which adds additional delay, compared to UDP which does not guarantee packet delivery. Sequences of TCP and UDP communication can be seen in Figure 7 and 8 respectively.

UDP is chosen for this work for minimum latency over TCP and the modern decoders have fault correction for lost bits.

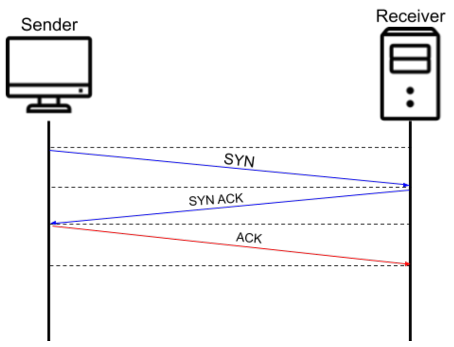


Figure 7: Sequence diagram of a TCP communication.

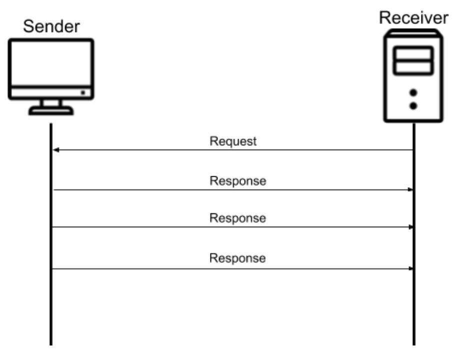


Figure 8: Sequence diagram of a UDP communication.

4.2.2 Stream Synchronization

A certain amount of delay is unavoidable while media streaming. Majority of the delay is caused by the encoding and transporting the stream and the delay amount can vary over time. Although this does not affect the quality of service for a single media stream as long as the delay stays in acceptable range.

However, the varying delay can be a problem when there are multiple media streams present from different cameras of the vehicle. The streams viewed by the developers should be synchronized to make healthy analysis. UDP itself is not capable of providing such mechanism. Therefore, RTP is used as an upper-layer on top of UDP. RTP is an internet protocol that provides extra functionalities for real-time media streaming [21].

The functionalities provided by RTP can be listed [21]:

- *Time-stamping:* RTP adds timestamp to the packets which can be used to synchronize multiple stream.
- *Sequence numbering:* UDP does not guarantee for the packages to be delivered in order, therefore the packages may arrive out of order. RTP adds sequence number to the payload so the receiver can use it to order packets. The sequence number can also be used to detect packet loss.
- *Payload type identification:* RTP adds a field to its header to identify the payload type (video, audio etc.).
- *Source identification:* RTP adds a field to its header, called synchronization source

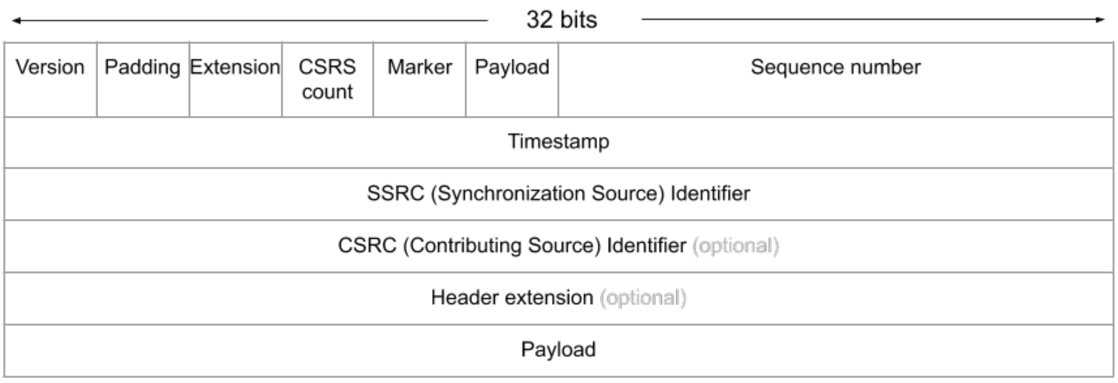

32 bits
Version
Padding
Extension
CSRS count
Marker
Payload
Sequence number
Timestamp
SSRC (Synchronization Source) Identifier
CSRC (Contributing Source) Identifier (optional)
Header extension (optional)
Payload

Figure 9: Diagram of an RTP.

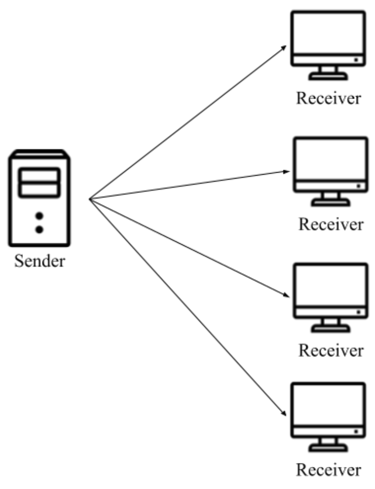
identifier (SSRC), to specify the source of the packet. This is important to distinguish which packet is provided by which media source.

Although RTP provides additional functionalities on top of UDP to synchronize multiple streams, it is still not capable of controlling it. RTCP is a control protocol that is designed to be used with RTP. RTCP provides bi-directional feedback system where both sides send report to each other periodically. Reports can include information about the packet loss statistics, packet arrival time and delay. Sender and receiver can use these reports to synchronize multiple streams by adjusting the transmission rate and using buffers. The diagram of an RTP node can be seen in Figure 9.

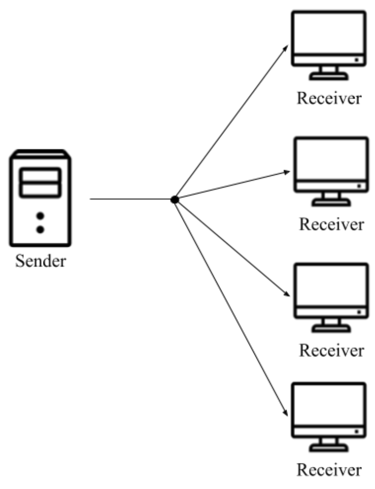
4.2.3 Delivery of Stream

The delivery of real-time stream can be categorized into two, unicast delivery and multicast delivery. The unicast delivery of real-time video streaming can be described as point-to-point delivery. Only one receiver and one sender are involved during streaming. On the other hand, multicast delivery is point-to-multipoint system, there's only one sender but multiple receivers.

The advantage of unicast system is that it can provide quality of service to the receivers in a heterogeneous network environment. The sender can use different encoding and transport protocols as well as various video properties (such as dimensions, bitrate, fps etc.) for each receiver based on their network or hardware system. However, this comes with a cost. Since there's an individual stream per receiver, resource usage increases



(a) Unicast video delivery



(b) Multicast video delivery

proportional to the number of receivers due to re-encoding of media per receiver. Likely, bandwidth usage increases per receiver as well.

On the other hand, multicast delivery is efficient in bandwidth and resource usage, since single encoding is done, regardless of receiver number and media is streamed only once. The disadvantage of multicast is that the receiver is expected to meet required resources. First, the network connection of the receiver should be enough to meet the incoming bitrate. Second, the hardware and software should be capable enough to decode and display the stream in real time.

The receiver of the stream in this project is the development team of the vehicle, therefore high diversity of network and hardware capabilities are not expected. Moreover, performance and bandwidth efficiency are more important than receiver's quality of service. Therefore, multi-cast delivery of stream is used in this solution.

4.3 Implementation

The system has been developed in Python. ROS' Python binding libraries, GStreamer and OpenCV2 packages are used as dependencies. *GstAppSrc* element from GStreamer's base library is used to create a local buffer and inject media into it.

GstAppSrc emits three signals, *need-data*, *enough-data* and *seek-data*. need-data signal is emitted when the local buffer is about to get out of data and enough-data is emitted

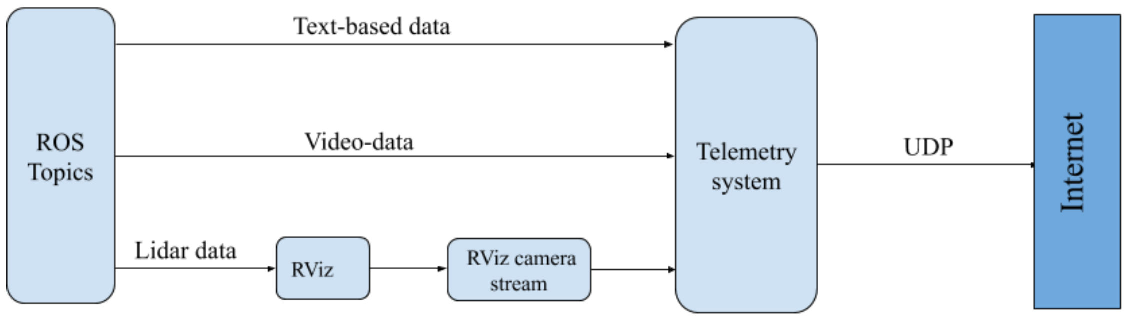


Figure 11: Telemetry system that runs in the vehicle.

when the local buffer is full and cannot accept any more data. seek-data is emitted to change the offset but it is not a concern of this project.

A flag, where its value is changed with *need-data* and *enough-data* signals, has been set to indicate whether the system should push new frames to the local buffer. Specified ROS topic with *Image* data type is subscribed. The data is converted into OpenCV2 object by using the OpenCV bridge library provided by ROS.

The data is then, either pushed to the local buffer with JPEG encoding or omitted, depending on the flag that indicates the status of the local buffer. Later, the buffer is resized by the provided dimensions and frame rate per second, followed by encoding and sending the stream through UDP to the server.

The scheme of the system that runs in the car can be seen from Figure 11. All the information from sensors are published from ROS topics. RViz is used to visualize the lidar data and then published with *Image* data type by RViz Camera Stream plugin.

All these data are processed in Telemetry system and then sent to the server. The scheme of the server can be seen in Figure 12.

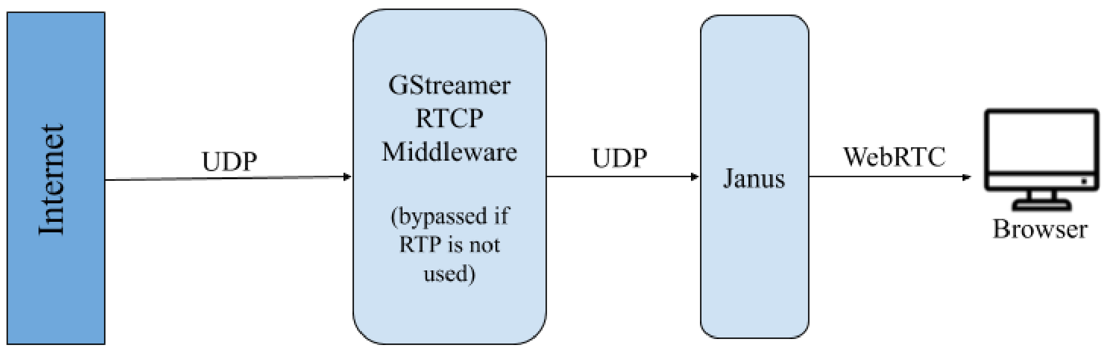


Figure 12: Telemetry system that runs in the server.

Although Janus' core has the infrastructure to provide RTP/RTCP support, the *streaming* plugin, which does WebRTC streaming, does not handle such protocol. Therefore, a middleware is implemented with GStreamer to handle RTCP and then forward the stream to Janus locally.

5 Experiments

5.1 Experiment Setup

In order to test the implementations, we needed to run them in real environment to measure the latency in a 4G cellular network with various configurations and multiple cameras.

A number of experiments have been set with Iseauto, using its own real cameras and 4G connection. The tests were configured to run with various FPS values (30 and 60), various screen resolutions (960x480, 1024x768, 1920x1080) in different bitrates (512kbps, 1024kbps, 2048kbps and 4096kbps). All tests were run with multiple cameras with and without RTP/RTCP to inspect the delay added for stream synchronization by RTP and also the latency between multiple media streamed simultaneously without RTP.

Media is streamed for 3 minutes in each test. Latency clock is used to capture the latency of the streams. This includes the delay from encoding and from transportation over UDP. Latency is measured for each frame and accumulated per each 5000 milliseconds by average to draw charts.

The goals of the experiments are to validate the synchronization of the media streams and the verification of the proposed solution and see how far the vehicle can handle real-time video encoding.

5.2 Results

5.2.1 Effect of FPS on Latency

One of the configurable value in the software is FPS. FPS (frame per second) defines how many frames there are per second in a video. A higher FPS provides a smoother video experience and a video with too low FPS can appear like slide show as the only a small amount of frames appears for each second. The disadvantage of high FPS is the fact that the encoder needs to process more frames per second.

We have run several tests where the only variable is FPS value and compared the results to see how FPS affects the latency in the system.

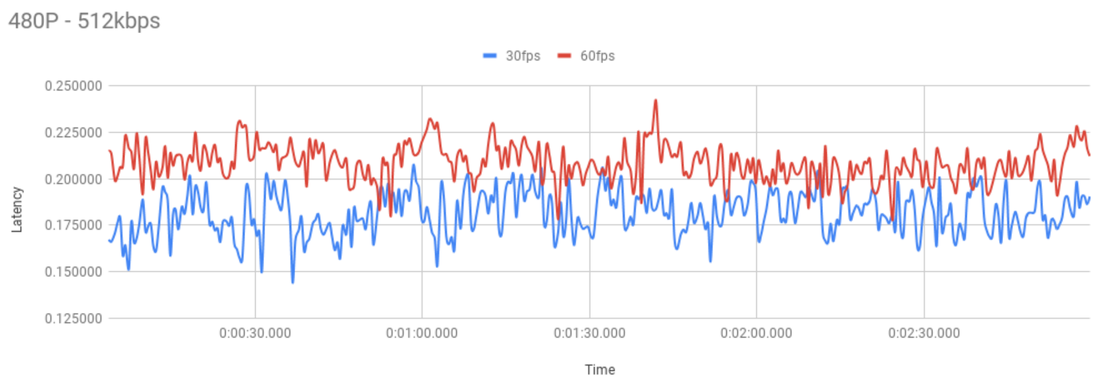


Figure 13: Latency comparison of 30 FPS and 60 FPS video streaming.

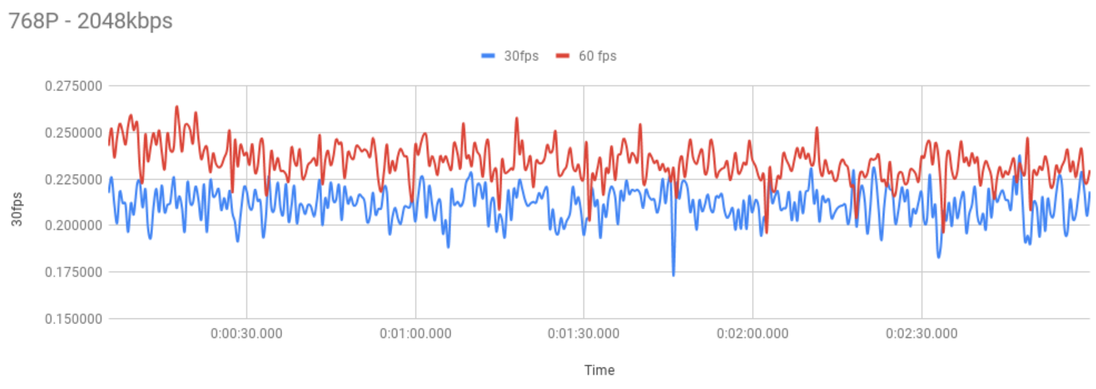


Figure 14: Latency comparison of 30 FPS and 60 FPS video streaming.

Figure 13 shows the comparison of latency where one media is streamed with

- 960x480 resolution,
- 512 kbps bandwidth
- and 30 frame per second

and another media with

- 960x480 resolution,
- 512 kbps bandwidth,
- and 60 frame per second

Another comparison of FPS but with different resolution and bitrate is shown in Figure 14. One media is streamed with

- 1024x768 resolution,
- 2048 kbps bandwidth,
- and 30 frame per second

and another media with

- 1024x768 resolution,
- 2048 kbps bandwidth,
- and 60 frame per second.

These two line charts show that increasing the demand on encoding by doubling the frame per second escalates the latency slightly.

5.2.2 Limitation of Resources

Figure 15 shows the latency where the video is streamed with 1920x1080 resolution with 60 frames per second and 1024 kbps bandwidth. As it can be seen, one of the configuration is causing a bottleneck and increments the delay over time drastically.

In Figure 14, we can see that 4G network can handle twice the bandwidth and in Figure 16, it can be observed that a bottleneck does not occur with two simultaneous media streams with 1920x1080 resolution with 30 frame per second and 2048kbps bandwidth each.

The cause of it can be explained by the fact that 1080P encoding with 60 frame per second is higher than the processor's single core can handle. Therefore, no more tests were done with this configuration as it does not generate healthy data.

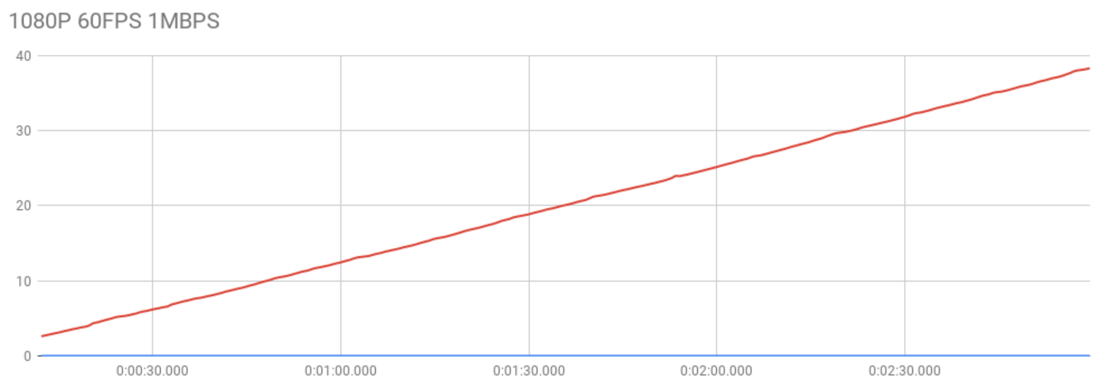


Figure 15: Latency of 1080P media with 60 FPS.

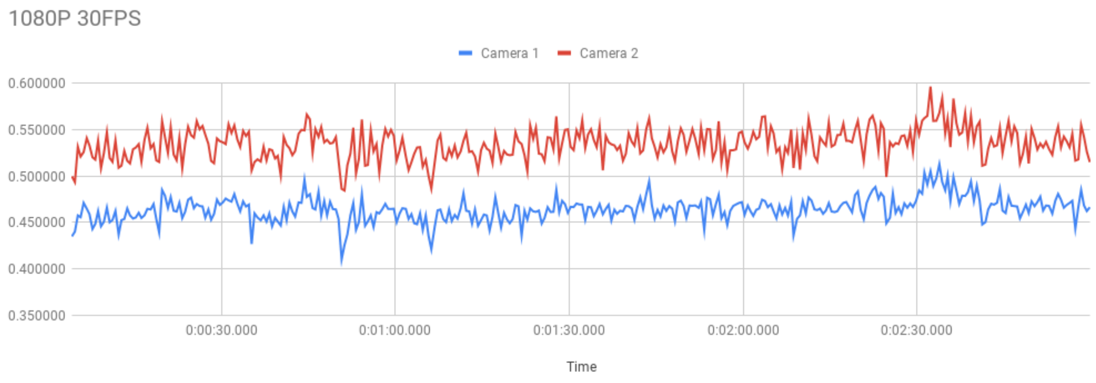


Figure 16: Latency of 2 streams with 1080P media with 30 FPS.

5.2.3 Simultaneous Streaming without Synchronization

A simultaneous streams of two media with 1024x768 resolution, 30 FPS and 512 kbps bandwidth is presented in Figure 17. RTP/RTCP synchronization is not used during the streams. Although it's fraction of a second, the latency difference between two streams is visible in the figure.

The latency difference may increase significantly if the two media are streamed with different configuration due to different encoding time and bandwidth. Figure 18 shows a latency comparison of two streams, where one media is streamed with

- 960x480 resolution,
- 60 frame per second,
- and 512 kbps bandwidth

and another media with

- 1920x1080 resolution,
- 30 frame per second,
- and 1024 kbps bandwidth.

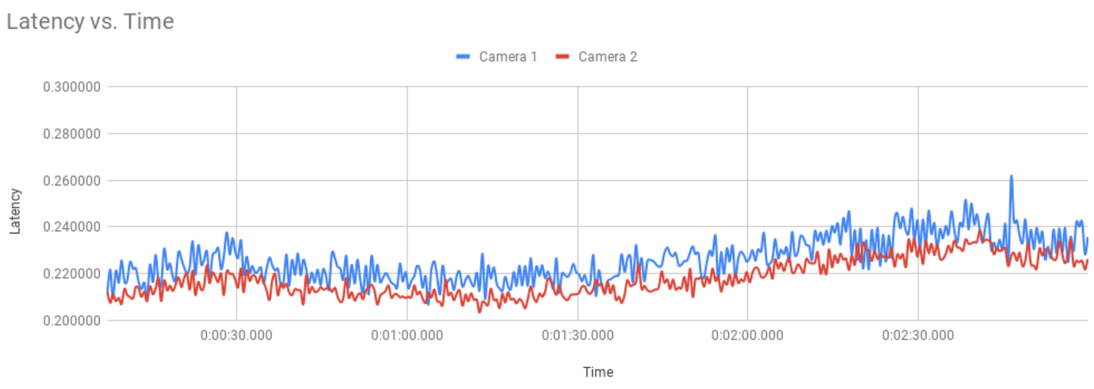


Figure 17: Latency comparison of two streams with 768P media with 30 FPS.

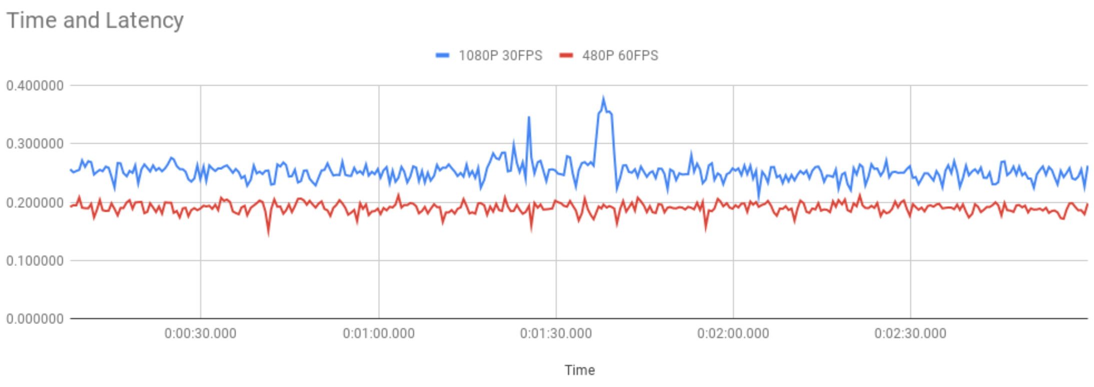


Figure 18: Latency comparison of two streams with 480P 60FPS and 1080P 30FPS.

Figure 19 shows another test with 3 cameras where each stream has 1024x768 resolution, 30 frame per second and 512 kbps bandwidth. As it can be seen, since they use the same configuration, most of the time the latency difference is low but there are time spans where it jumps more than 300 milliseconds.

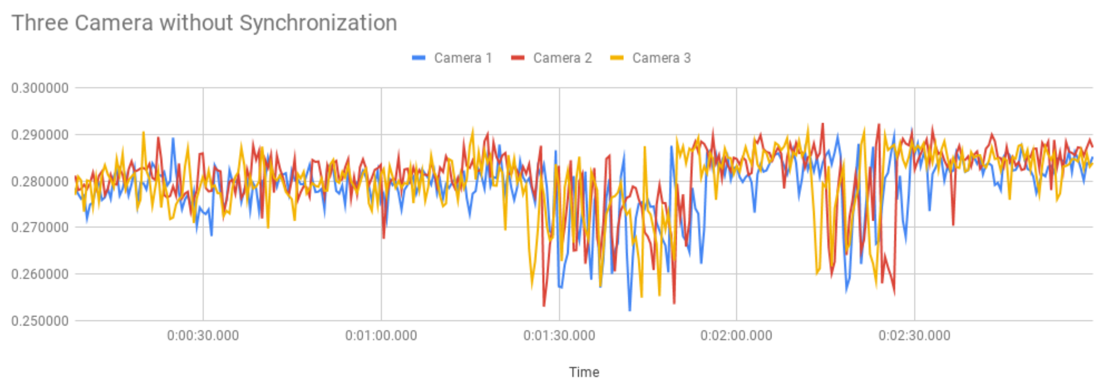


Figure 19: Latency comparison of three simultaneous stream with 768P, 30FPS and 512 kbps.

5.2.4 Simultaneous Streaming with Synchronization

The same simultaneous tests were repeated with RTP/RTCP protocols. The configuration for *rtpbin* element of GStreamer is as follows:

- **latency:** 200 ms buffer in jitterbuffers.
- **drop-on-latency**: false, jitterbuffer is allowed to exceed the given latency.
- **rtcp-sync-interval**: 0, always synchronize.

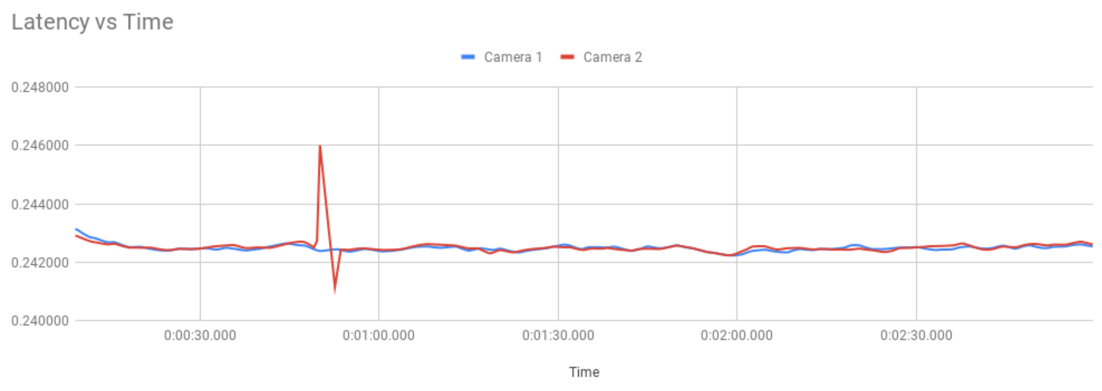


Figure 20: Latency comparison of two simultaneous stream with 768P, 30FPS and 512 kbps

Figure 20 shows a simultaneous streams of two 1024x768 resolution with 30 FPS and 512 kbps bandwidth where RTP/RTCP synchronization is used. Unlike Figure 17, where the latency difference was more than 10 milliseconds due to that the synchronization was not used, the latency difference with synchronization is less than 1 millisecond.

Figure 21 shows the latency comparison of two streams with one media with

- 960x480 resolution,
- 60 frame per second,
- and 512 kbps bandwidth

and another media with

- 1920x1080 resolution,
- 30 frame per second,
- and 1024 kbps bandwidth.

Compared to Figure 18, the latency difference is not visible in the line chart and less than 1 millisecond.

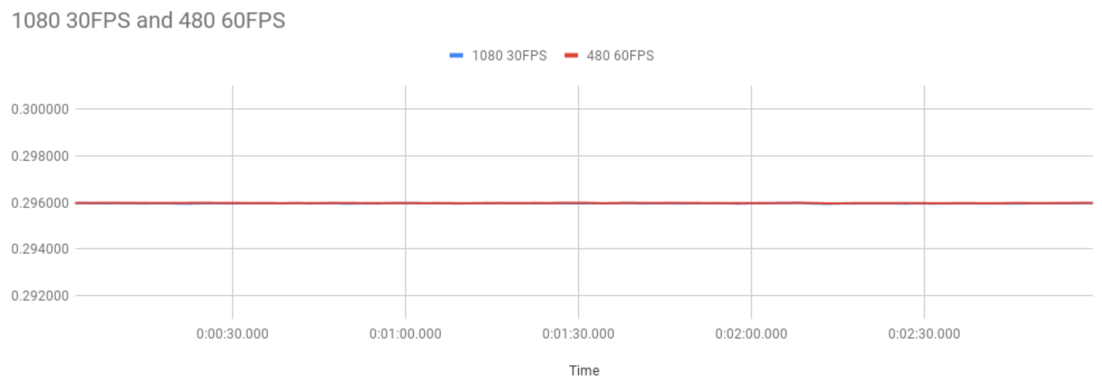


Figure 21: Latency comparison of two streams with 480P 60FPS and 1080P 30FPS.

Figure 22 shows the latency difference where three camera outputs are streamed with 1024x768 resolution, 30 frames per second and 512 kbps bandwidth for each stream. The latency difference is again less than 1 millisecond and barely visible in the line chart.

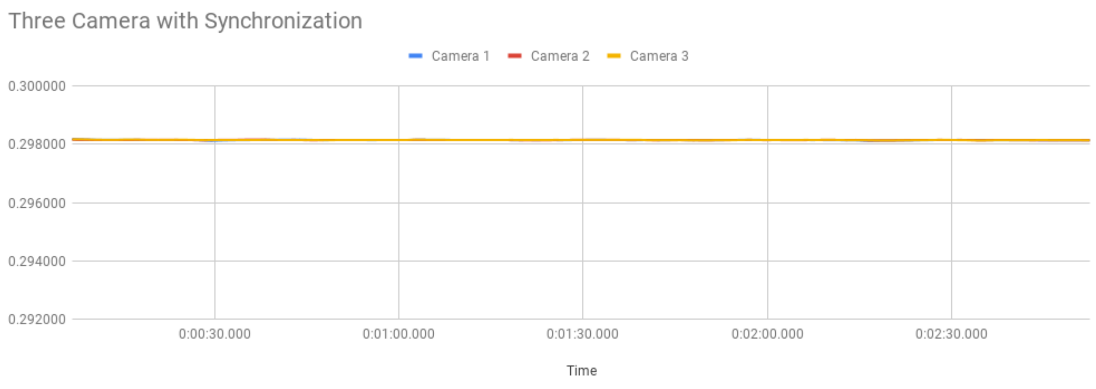


Figure 22: Latency comparison of three simultaneous stream with 768P, 30FPS and 512 kbps.

6 Discussion

Experiments have been conducted to verify that the solution provided can meet the purpose of this thesis. These experiments were done to show the latency changes with different configurations. The results have shown that the system can effectively stream media with 4G cellular connection.

Although lower latency occurs while streaming multiple videos simultaneously without synchronization, the latency difference can be misleading and may increase if the media streams have different configurations. Therefore, synchronization should be preferred when multiple videos are streamed simultaneously.

Also the system can be integrated with other ROS plugins, as long as a topic with *Image* data type is provided. The system only requires GStreamer 1.0 installed in the system as a dependency and does not require any additional hardware.

The system can be extended or modified thanks to its GStreamer-based structure, the encoding configurations can be changed or other encoders can be used. Moreover, the transportation protocol can be changed with other supported protocols if needed, such as TCP or HLS.

7 Future Work

The program can be re-written in C language rather than Python to provide a robust and faster system. As shown from the results, RTP/RTCP provides synchronization among multiple simultaneous media streams, but Janus plugin does not support RTP stream at the moment and an intermediate GStreamer pipeline is used as a bridge. Janus' streaming plugin can be extended to provide a native solution.

To bring genericness to the system, a CPU-demanding encoder is used in this project. However, there are hardware accelerated GStreamer encoders available. GStreamer is officially providing acceleration API for Intel® devices [22]. NVIDIA is also providing accelerated solution in Tegra® Linux Driver Package that is available for NVIDIA® Jetson AGX Xavier™ devices. The system can be extended to use one of these APIs to enhance the encoding performance depending on the hardware used.

Another improvement would be a bridge from OpenCV to GStreamer buffer. In the current system, first OpenCV encodes the frame to JPEG and it's then decoded again by GStreamer before adding the frame to its buffer. OpenCV can be extended to provide an output that can be directly inserted to GStreamer's buffer to eliminate the unnecessary encoding and decoding steps.

8 Summary

The purpose of this thesis is to provide a telemetry solution for autonomous self-driving cars that is running Robot Operating System as its core. The system uses GStreamer as the core video encoding and media streaming tool and it can be integrated with ROS plug-ins to provide stream for sensors.

The system does not require additional hardware and is capable of running Linux and Mac operating systems. In addition to ROS libraries, it only requires GStreamer installed in the operating system.

The system is able to stream multiple media and alphanumeric data simultaneously and it optionally provides RTP/RTCP if the synchronization of simultaneous streams is required.

Experiments that were consist of bandwidth stress tests, hardware stress tests and latency tests have been conducted to show the system's performance with different configurations under 4G cellular network. Results from the experiments showed that the system can achieve its purpose.

References

[1] R. Read, "Ieee says that 75% of vehicles will be autonomous by 2040," Sep 2012.

[2] "Iseauto r&d web page." http://iseauto.ttu.ee/en/rd/. Accessed: 2019-05-11.

[3] Y.-Q. Z. Dapeng Wu, Yiwei Thoms Hou, "Transporting real-time video over the internet: challenges and approaches," *Proceedings of the IEEE*, vol. 88, pp. 1855–1877, Dec 2000.

[4] A. Majumda, D. G. Sachs, I. V. Kozintsev, K. Ramchandran, and M. M. Yeung, "Multicast and unicast real-time video streaming over wireless lans," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, pp. 524–534, June 2002.

[5] Dapeng Wu, Y. T. Hou, Wenwu Zhu, Hung-Ju Lee, Tihao Chiang, Ya-Qin Zhang, and H. J. Chao, "On end-to-end architecture for transporting mpeg-4 video over the internet," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 10, pp. 923–941, Sep. 2000.

[6] J.-C. Bolot and T. Turletti, "Experience with control mechanisms for packet video in the internet," *SIGCOMM Comput. Commun. Rev.*, vol. 28, pp. 4–15, Jan. 1998.

[7] A. Balachandran, A. Campbell, and M. Kounavis, "Active filters: Delivering scaled media to mobile devices," pp. 125 – 134, 06 1997.

[8] S. McCanne, V. Jacobson, and M. Vetterli, "Receiver-driven layered multicast," *SIGCOMM Comput. Commun. Rev.*, vol. 26, pp. 117–130, Aug. 1996.

[9] J. van der Hooft, S. Petrangeli, T. Wauters, R. Huysegems, P. R. Alface, T. Bostoen, and F. De Turck, "Http/2-based adaptive streaming of hevc video over 4g/lte networks," *IEEE Communications Letters*, vol. 20, pp. 2177–2180, Nov 2016.

[10] Conviva, "Viewer experience report," 2015.

[11] G. J. Sullivan, J. Ohm, W. Han, and T. Wiegand, "Overview of the high efficiency video coding (hevc) standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, pp. 1649–1668, Dec 2012.

[12] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)." RFC 7540, May 2015.

[13] R. Huysegems, J. van der Hooft, T. Bostoen, P. Rondao Alface, S. Petrangeli, T. Wauters, and F. De Turck, "Http/2-based methods to improve the live experience of adaptive streaming," in *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, (New York, NY, USA), pp. 541–550, ACM, 2015.

[14] A. Albanese, J. Blomer, J. Edmonds, M. Luby, and M. Sudan, "Priority encoding transmission," *IEEE Transactions on Information Theory*, vol. 42, pp. 1737–1744, Nov 1996.

[15] C. Mondello, G. Hepner, and R. Medina, "Asprs ten-year remote sensing industry forecast - phase v," *Photogrammetric Engineering and Remote Sensing*, vol. 74, pp. 1297–1305, 11 2008.

[16] M. Kuder and B. Žalik, "Web-based lidar visualization with point-based rendering," in *2011 Seventh International Conference on Signal Image Technology Internet-Based Systems*, pp. 38–45, Nov 2011.

[17] M. Schütz and M. Wimmer, "Progressive real-time rendering of unprocessed point clouds," Aug. 2018. Poster presented at ACM SIGGRAPH 2018 (2018-08-12–2018-08-16).

[18] "Ros official web page." https://www.ros.org/. Accessed: 2019-05-11.

[19] "Autoware github page." https://github.com/autowarefoundation/autoware. Accessed: 2019-05-11.

[20] A. Rassõlkin, R. Sell, and M. Leier, "Development case study of the first estonian self-driving car, iseauto," *Electrical, Control and Communication Engineering*, vol. 14, pp. 81–88, 07 2018.

[21] Dapeng Wu, Y. T. Hou, Wenwu Zhu, Ya-Qin Zhang, and J. M. Peha, "Streaming video over the internet: approaches and directions," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, pp. 282–300, March 2001.

[22] V. Jaquez, "Gstreamer-vaapi hardware-accelerated encoding and decoding on intel® hardware," *GStreamer Conference*, oct 2015.