TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JOONA LAINE

LOW LATENCY HIGH-DEFINITION VIDEO STREAMING FOR REAL-TIME TELEOPERATION PLATFORM

Master of Science thesis

Examiner: Prof. Atanas Gotchev
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 7th December 2016

ABSTRACT

JOONA LAINE: LOW LATENCY HIGH-DEFINITION VIDEO STREAMING FOR REAL-TIME TELEOPERATION PLATFORM
Tampere University of Technology
Master of Science thesis, 42 pages, 0 Appendix pages
May 2017
Master's Degree Programme in Information Technology
Major: Signal Processing
Examiner: Prof. Atanas Gotchev
Keywords: teleoperation, real-time, low latency, GStreamer, H.264

Teleoperation is the remote controlling of machines using a real-time video stream to support the controlling decisions. The key components of a teleoperation system are video streaming through a network to enable the control, low enough latency to ensure real-time control and latency - bandwidth - resolution balance of the streaming system. In general, a low latency means high bandwidth consumption and the used resolution relates also straight to the bandwidth. Using the modern video coding method H.264/AVC allows for the reduction of bandwidth and latency by selecting a suitable H.264 profile.

This thesis studies the possibility and effect of maximizing the usage of a graphics processing unit (GPU) in the streaming pipeline of a teleoperation platform and presents measurements to show the impact on the streaming latency. An open source multimedia framework GStreamer is used in the pipeline construction of the platform. The thesis presents the creation of two teleoperation platforms and examines their features. Latency measurements between an existing system and one of the developed systems are compared and results discussed. The results show that employing a GPU in the streaming pipeline greatly improves the performance of the system and allows the streaming of multiple simultaneous low latency high resolution video streams.

TIIVISTELMÄ

JOONA LAINE: Matalan latenssin korkealaatuinen videon suoratoisto reaaliaikaiseen teleoperointialustaan
Tampereen teknillinen yliopisto
Diplomityö, 42 sivua, 0 liitesivua
Toukokuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Signaalinkäsittely
Tarkastaja: Prof. Atanas Gotchev
Avainsanat: teleoperointi, reaaliaikaisuus, matala latenssi, GStreamer, H.264

Teleoperointi on koneiden kauko-ohjausta, jossa hyödynnetään reaaliaikaista videon suoratoistoa ohjauspäätösten tukena. Teleoperointijärjestelmän keskeisiä komponentteja ovat verkon läpi tapahtuva videon suoratoisto ohjauksen mahdollistamiseksi, riittävän matala latenssi varmistamaan reaaliaikainen ohjaus sekä latenssin, kaistankäytön ja resoluution tasapaino suoratoistojärjestelmässä. Yleisesti ottaen matala latenssi tarkoittaa korkeaa kaistankäyttöä, johon käytetty resoluutio suoraan vaikuttaa. Modernin videokoodekin H.264/AVC käytöllä voidaan vaikuttaa kaistankäyttöön ja latenssiin valitsemalla sopiva H.264 profiili.

Työssä tutkitaan näytönohjaimen (GPU) käytön maksimoinnin mahdollisuutta ja vaikutusta teleoperointialustan suoratoistoputkessa ja esitetään mittaustulokset ja käytön vaikutus suoratoiston latenssiin. Alustan suoratoistoputken rakentamisessa käytetään avoimen lähdekoodin viitekehystä GStreameria. Työ esittelee kahden teleoperointialustan kehitystyön ja tarkastelee niiden ominaisuuksia. Työssä vertaillaan jo olemassa olevan ja toisen työssä kehitetyn alustan latenssimittauksia ja pohditaan saatuja tuloksia. Tulokset vahvistavat, että näytönohjaimen käyttö parantaa suuresti suoratoistojärjestelmän performanssia ja mahdollistaa usean samanaikaisen matalan latenssin sekä korkean resoluution videon suoratoiston.

PREFACE

This thesis was written at OptoFidelity Oy. The thesis was done as a result of developing a new version of the OptoFidelity OptoMon teleoperation platform for industrial environment. I want to thank my superior Lasse Lepistö for giving me this opportunity to work with the project and learn a ton of programming.

I also want to thank my supervisor professor Atanas Gotchev for supervising and taking it also his personal agenda to help me get this work finished. Especially I owe my endless gratitude to my colleagues Juha Leino and Kari Mäkelä who both acted as mentors and provided invaluable support and advice during the development process. I also want to thank Janne Honkakorpi and especially Rebekah Rousi for their valuable advice on spelling and structuring of the thesis.

Finally, I want to thank my family and friends and especially my wife Maria, who has been supporting me tirelessly throughout this process in every aspect imaginable.

Tampere, 22.05.2017

Joona Laine

TABLE OF CONTENTS

1. Introduction 1

2. Background 3

  2.1 Teleoperation 3

  2.2 Video streaming 4

  2.3 GStreamer 9

  2.4 Platform requirements and goals 13

3. Platform development 15

  3.1 Development using Qt 15

  3.2 Development using Glib 23

4. Platform evaluation 28

  4.1 Latency benchmark 28

  4.2 OptoMon v1 latency measurements 31

  4.3 OptoMon v2 latency measurements 33

  4.4 Comparison between platforms 35

5. Discussion 39

6. Conclusions 41

Bibliography 43

LIST OF FIGURES

2.1 Differences of image resolutions 7

2.2 Graph illustration of end-to-end latency 8

2.3 A simple GStreamer pipeline 10

2.4 An advanced GStreamer pipeline 11

3.1 H.264 profiles 18

3.2 Qt platform workflow 19

3.3 Architectural overview of the Qt platform 21

3.4 Architectural overview of the Glib platform 24

3.5 Workflow of the Glib platform 25

3.6 Workflow of the Stream Manager 26

4.1 Display setup used in latency measurements 29

4.2 Camera and LED target setup used in latency measurements 30

4.3 Latency measurements using camera Q1765-LE and MJPEG 35

4.4 Latency measurements using camera Q3709-PVE and MJPEG for OptoMon v1, H.264 for OptoMon v2 36

4.5 Latency measurements using camera Q1765-LE and H.264 37

4.6 Latency measurements using camera Q3709-PVE and H.264 38

LIST OF TABLES

4.1 Benchmark latencies measured from Axis Live View 31

4.2 Latency measurements of OptoMon v1 platform 32

4.3 Latency measurements of four simultaneous streams using OptoMon v1 platform 32

4.4 Latency measurements of OptoMon v2 platform 33

4.5 Performance measurements of OptoMon v2 platform 34

LIST OF ABBREVIATIONS AND TERMS

API	Application Programming Interface
AVC	Advanced Video Coding
CPU	Central Processing Unit
FHD	Full High-Definition
fps	Frames Per Second
GPU	Graphics Processing Unit
GUI	Graphical User Interface
H.264	ITU-T naming convention for video codecs, see AVC
HD	High-Definition
HTTP	Hypertext Transfer Protocol
HW	Hardware
JPEG	Joint Photographic Experts Group
LED	Light Emitting Diode
MB/s	Megabytes per second
MPEG-4	Moving Picture Experts Group number 4
ms	millisecond
NAL	Network Abstraction Layer
NTP	Network Time Protocol
OpenGL	Open Graphics Language
POE	Power Over Ethernet
PPS	Picture Parameter Set
RTCP	Real-Time Control Protocol
RTP	Real-Time Transfer Protocol
RTSP	Real-time Session Protocol
SD	Standard Definition
SDK	Software Development Kit
SPS	Sequence Parameter Set
SW	Software
TCP	Transfer Control Protocol
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
UHD	Ultra High-Definition
VAAPI	Video Accelerated API
VM	Video Multimeter

1. INTRODUCTION

We live in a world of automation. We already have lawnmowers and vacuum cleaner bots that do not need a human operator. We also have unmanned aerial vehicles (UAVs) in the military field. Even some consumer grade drones have features like automatic return to home on low battery or when out of reach of the controller. In addition, there is a lot of research going on about autonomously driving vehicles [1, 23, 33]. Many of these vehicles greatly benefit from camera-based sensory input and some can only be operated by using a video stream. In this case we are talking about teleoperation.

At the core of any teleoperation system there is a camera or a number of cameras. The videos from these cameras are streamed to the operator who makes controlling decisions based on the video streams. In order for this to work properly and accurately, the system should allow the operator to get up-to-date information and feedback on the controls he or she makes on the machine or device being operated. In order to give the operator as much information as possible about the field conditions, it is possible to use high-definition (HD) video streams. A key term in teleoperation is latency; the time difference between the actions in the real world and what is shown on the display at the operator's end.

This combination of cameras, possibly high-definitions cameras, real-time video streaming and low latency requirements poses a lot of challenges to the teleoperation system. There are many companies offering different possibilities to overcome these challenges. Some companies, such as NanoCosmos, Wowza and StreamBox, offer streaming solutions that do not quite satisfy the low-latency part and are more suited to streaming videos to massive amounts of people with a latency of a couple of seconds. On top of these there are some that offer a lower, close to one second end-to-end latency for the video streams. Companies such as The Streaming Company, Phoenix P2P and Unreal Streaming Technologies fall in to this category. Then there are those who can offer a low enough latency to allow real-time teleoperation.

These are companies such as Ittiam Systems, CoreEl technologies, IPX communications and OptoFidelity. Some of the companies offer purely software solutions such as their own customized platforms, whereas some of the systems are cloud-based streaming services. Among the companies there are also those who offer highly optimized hardware for video encoding, decoding and transcoding to offer the lowest possible latency for varying environments and devices.

OptoFidelity Oy has developed a multimedia platform for making complex monitoring and teleoperation applications for their customers. The platform is called OptoMon and consists of several self-built libraries used by the applications to enable video streaming and rendering. The development of the platform was originally started in 2008 when it was used for in monitoring industrial processes. From 2010 onwards it was introduced in the teleoperation environment. The platform is now facing multiple challenges. For this reason the system is now being remade.

From a customer perspective a teleoperation system is a matter of cost, latency and bandwidth. With a low-cost software solution you probably get adequate latency with relatively low bandwidth consumption. Whereas, with a high-cost hardware and software solution you get dedicated hardware and possibly very low latency, the drawback is that the price of extremely low latency is very high bandwidth. What if there was a software-based solution that could offer the best of the two; low latency and low bandwidth consumption at a reasonably low price? This current thesis presents the development of a system that is intended to possess low latency, require low bandwidth and be relatively inexpensive.

This document is structured as follows. Chapter 2 covers the background information, including the problem setting and main concepts of the thesis. In Chapter 3 we look at the two developed solutions to the problem. Chapter 4 includes an evaluation of the solutions presented earlier. In Chapter 5 we discuss the significance of the work. Finally, Chapter 6 includes conclusions and presents the future outline of the work.

2. BACKGROUND

In this chapter, the core concepts of the study are presented. First, the meaning of teleoperation is defined. Then, video streaming and the concepts involved in streaming is looked at. Next, GStreamer and the way it enables creating video streaming pipelines is explained. Finally, the main goals of the thesis are presented.

2.1 Teleoperation

Teleoperation is the remote controlling of machines and is used in situations where the robot or machine is unable to perform a task, subsequently necessitating the guidance of a human operator [26]. In the context of this thesis, teleoperation is used in the exact aforementioned way. For most of the time the machine operates autonomously, yet there are some scenarios in which it is unable to fulfill its task. This can be due to bad weather conditions if the machine is outdoors, or it can be a task that requires extreme precision or human judgement. The way teleoperation is made possible is through a video stream from the controlled machine to the operator that is further away from the machine. There might not even be a direct line of sight between the two. In this case, the video stream can be the only feedback the operator gets from operating the machine.

In a situation described above, it is of utmost importance that the feedback is appropriately controlled. There is a high chance of hazard if the feedback from a manoeuver comes too late to the operator. Even a couple hundreds of milliseconds might be too much for precision demanding tasks. The ultimate goal in teleoperation could be to enable the controls to feel as if the operator was present at the scene. This would mean zero latency in the operation. However, in true life situations, where teleoperation is used, this is usually not the case. Luckily latency can be reduced to a degree where real-time teleoperation is possible.

2.2 Video streaming

Streaming is defined as digital distribution of audio or video material in real-time [3]. Usually this happens over a network, which can be wired, wireless or both. The used network infrastructure affects the amount of latency experienced when streaming and the available bandwidth of the network affects, e.g., the quality of the streamed media. In this thesis the streaming is concentrated on video and audio is not considered.

Streaming a video has multiple steps. The most important being, from the point of view of this thesis, the encoding of the video, transmission over a network, decoding and finally rendering. The encoding usually happens in the camera. Encoding is a process where the video data is transformed into another form. The raw video data that is read from the sensor of a camera would be impractical to transmit as it is. Thus, encoding methods have been developed to transform the data into a more compact form without changing the content, or if necessary, changing it as little as possible. The encoded stream is not such that it could be viewed as it is, but needs to be decoded first. A corresponding decoder to the utilised encoder must be used. Encoding is also referred to as compression, simply because it compresses the data.

The main video compression technique in OptoMon is the MPEG-4 Advanced Video Codec (AVC), or more commonly known as H.264. It is one of the most used and best performing video codecs today [21] and has therefore been chosen to be used in OptoMon as well. Also, since most of the cameras employed by the users are capable of encoding video streams with H.264, it is a logical choice. The current OptoMon implementation also supports motion JPEG (MJPEG) encoded video streams due to legacy cameras. The use of MJPEG in encoding and decoding has the benefit of decreasing latency by being less complex than H.264. However, the downside of this is its much larger resulting bitstream. This is extremely noticeable when simultaneously streaming multiple streams or HD streams.

Decoding

The decoding of a video stream is one of the key steps in the streaming pipeline. It might introduce a great deal of latency or missing frames to the stream if not handled appropriately. The used network architecture has subsequently a large impact on the latency. However, we are not considering it here as a problem, since it is up to the user to manage the network. The main problem with the current implementation

in OptoMon is that the decoder is solely based on software decoding. This means that the decoding is performed among all the other processing steps in the system and the computing unit used is the central processing unit (CPU).

The proposed method for solving the issues related to decoding is to use the graphics processing unit (GPU) of the computer for decoding. It is expected to dramatically increase the overall performance of the system due to the fact that multiple simultaneous streams can be decoded parallel on the GPU. In addition, each individual stream can be decoded in a parallel fashion due to the parallel architecture of the GPU and the advanced decoding algorithms.[6, 34]

Frames per second

Frames per second (fps) is a property of a video describing the smoothness of motion in the video. It specifies how many frames are displayed in one second. The more frames are shown to the human eye, the less it sees difference between adjacent frames and the sequence of frames give the impression of moving objects. By showing 10 to 12 fps to the human eye make the brain think it sees motion [29]. The higher the fps value, the smoother rapid movements appear.

The problem with OptoMon in some environments and hardware is that to ensure the latency stays within limits, the fps of the stream has been forced to be decreased. This has the effect that motion appears jerky. The jerkiness affects the experience in a negative way, since it makes it harder to follow the stream. The key point with fps really is in the operation part of teleoperation. It can make it really hard to operate a machine when the feedback from the video stream is lacking information. It is expected that the use of a GPU in the pipeline will help with this issue.

Real-time streaming protocol

The transmission of video data is usually performed by streaming it over a network. There are a number of ways this can be made possible and the two commonly used methods are hypertext transfer protocol (HTTP) and real-time streaming protocol (RTSP). OptoMon uses RTSP whenever possible and within RTSP there is a user datagram protocol (UDP) based real-time transport protocol (RTP) delivery. RTSP is the command protocol. The actual video data is not transferred over RTSP, but over RTP. RTP in turn is a thin protocol sent over UDP or transport control protocol (TCP) [31]. The use of UDP is preferred due to its nature of performing

connectionless communication. This means, that by using UDP there are no confirmation messages ensuring that all packets have arrived at their destination. In the case of real-time streaming this is not a problem, since we aim for minimum latency and packets arriving late would be discarded anyway. TCP works in a different way by making sure that all packets arrive at their destination. This is achieved by retransmitting lost packets. TCP is capable of introducing more latency to the streaming in challenging network conditions and is more suitable for less time critical communication, e.g. Youtube streaming.

Visual quality

Visual quality of a video can be understood in a multitude of ways. When talking about the quality of a video people might refer to its resolution, fps, smoothness of motion or visual artifacts in the playback. In this thesis the interest is mostly in the resolution of the video. The sufficiency of the visual quality of the video stream is defined by the user. In addition, the purpose of the video can critically affect the quality requirements. There are no strict guidelines when it comes to quality. The main focus is on the reliability of the stream; that a captured frame is displayed as soon as possible on the display. The settings of the streams can be adjusted to an appropriate level of visual quality depending on the user's requirements, environment and available hardware.

The problem in OptoMon is that the used hardware is unable to decode and render high resolution streams. A hardware update might help with the issue, but it has not been investigated and is out of the scope of this thesis. OptoMon is currently capable of streaming multiple standard definition (SD) quality streams, but has severe problems with high-definition (HD) quality. The customers have been adopting modern camera technology which means they want to increase the resolution of the streams.

HD video is considered to be a resolution of 1280 by 720 pixels per frame. The numbers refer to the horizontal and vertical number of pixels in each frame, respectively. This resolution is also called a 720p resolution and analogously 1080p is called Full-HD (FHD) resolution. With an aspect ratio of 16:9, which is often used in images, video and monitors, gives FHD frame a horizontal pixel count of 1920. The differences are displayed in Figure 2.1.

The more pixels there are in a frame the harder it is to encode. This increased

Figure 2.1 *An illustration of image resolutions to the image quality, details and area. (Source: https://www.worldeyecam.com/4-HD-1080P-Security-Dome-HD-CVI-DVR-Kit-for-Business-Professional-Grade.html. Image by WorldEyeCam Inc.)*

amount of data in turn takes up more bandwidth from the network and is harder to decode. Processing becomes even more demanding with numerous streams running simultaneously. As proposed in earlier sections, the use of a GPU plays a key role regarding the increasing resolution and data flow.

Until recently, the resolution in the industry-grade camera's video frames has been up to SD, or 480p, but the new cameras and technology has brought HD and FHD to the teleoperation field. OptoMon works adequately with SD streams, but has severe difficulties with HD and FHD. The main motivation with the new OptoMon is to enable streaming of multiple FHD streams, while keeping steady and low latency.

Latency

Latency refers to the time it takes from the moment a camera captures an image to the moment it is transferred through a network and rendered on a screen. Therefore, we talk about end-to-end latency. Figure 2.2 illustrates the different components each affecting the total latency of a video streaming system. The aim is to enhance the real-time monitoring and control of machinery as if the operator were actually

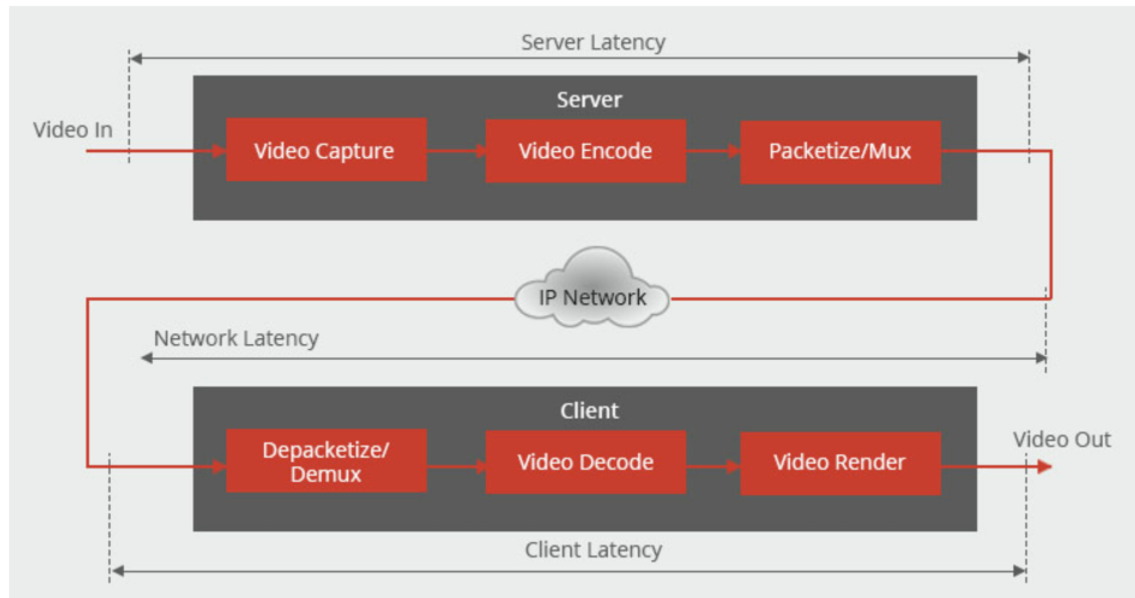

***Figure* 2.2** *An illustration of end-to-end latency in video streaming. (Source: http://www.ittiam.com/key-technologies/low-latency-video. Image by Ittiam Systems.)*

present in the cabin of he controlled machine. This is why it is important to try to maintain low latency. However, even more important is to know what the latency is and whether it is too high. There are two ways latency is being monitored in OptoMon. The first is the company's own product Video Multimeter (VM). The VM works by flashing an light emitting diode (LED) light with a known frequency. The light is positioned in a way that it is captured by the camera. There is an optical fibre on the display positioned where the blinking LED is shown. The device can then measure the end-to-end latency of the system, because it knows when the LED is lit in the real world and when the LED is lit on the display.

Measuring latency using the VM is a useful method in the laboratory environment and development phase, however it is not suitable for the field, since the machines are usually tens or even hundreds of meters away from the displays. The other way to measure the latency, which is feasible also in the field, is using a timestamp overlayed on the video. The timestamp is inserted onto every frame containing the time of capture, preferably in milliseconds, since latency should be determined in milliseconds. When the timestamped frame is transferred over the network the received and decoded frame's timestamp can be compared to the time of the computer. This requires that the server on the camera and the client computer are

synchronized using, e.g., network time protocol (NTP).

Keeping up a steady, low latency in the stream is the main motivation of OptoMon. In recent tests, it has been noticed that OptoMon is unable to keep the latency as low as it would need to be. This is apparent with streams from HD cameras, but also noticeable with multiple streams from SD cameras. It is the result of OptoMon not being able to process all the data in the streams. The system needs to buffer each stream in order to be able to process them. This in turn, increases the latency.

The means by which to reduce this behaviour is to use the GPU in the decoding to free up the resources of the CPU and share the processing workload. In the current systems, the GPU is not used in processing in any way. This means that significant performance upgrades can be gained by putting it to use. Another way to help with keeping up low latency is to change the currently used software components.

2.3 GStreamer

GStreamer is an open source multimedia framework for creating streaming media applications. Its foundations are at the Oregon Graduate Institute's video pipeline and ideas adopted from DirectShow. GStreamer allows for arbitrary pipeline construction which makes it an ideal tool when building any multimedia related applications. It is also extendable, meaning anyone can write their own plugins. This is a highly valued feature from the perspective of this thesis. [8]

At the hearth of GStreamer are the plugin packages, named the core, base, good, bad and ugly. In addition, there are also other packages, e.g. ,*Video Accelerated API (VAAPI)* and *libav*, which contain many encoders and decoders for audio and video formats to name some. The core and base packages are required to perform any basic media processing and usually some other packages are also needed depending on the requirements of the application. The packages contain plugins which in turn may have several elements. In this thesis, there are plugins and elements being used from all packages, except the ugly.

Before any actual platform development could be started there had to be a form of technology evaluation. This was necessary to ensure that the chosen multimedia framework was capable of performing the tasks it would need to do. It was also the appropriate time to learn how the whole framework functioned. After some

Figure 2.3 *An illustration of a simple GStreamer pipeline with a source element, one filter and a sink.*

preliminary pipeline construction and testing it was decided that the GStreamer pipeline was a suitable way to manage the video streams from multiple cameras.

Pipeline

The processing in GStreamer takes place in a pipeline. The pipeline is constructed from a number of elements and at minimum the pipeline needs a source and a compatible sink. The source provides data to the pipeline and can be, e.g., a file or a live source, such as a camera. The sink at the end of the pipeline ends the dataflow and outputs the stream e.g. into a file or renders it on a display. Between these two there can be a number of other elements, often called filters, which take data in from an upstream element, modify or otherwise process it and pass it on to a downstream element. A filter element might not do anything with the data in which case it just passes it through. Figure 2.3 shows an example pipeline. As can be seen, the data from the source flows downstream to the filter element and continues to the sink.

In an application there might only by one pipeline. This can be suitable when the pipeline has a relatively simple task, such as reading a source, processing it somehow and then displaying it. It is also possible to have multiple pipelines running at the same time. In the case of this thesis it was practical to have each video stream in its own pipeline. The simple reason is that this way the individual pipelines do not interfere with each other.

A real example of a pipeline is shown in Figure 2.4. The function of each element in the pipeline is explained in the following sections.

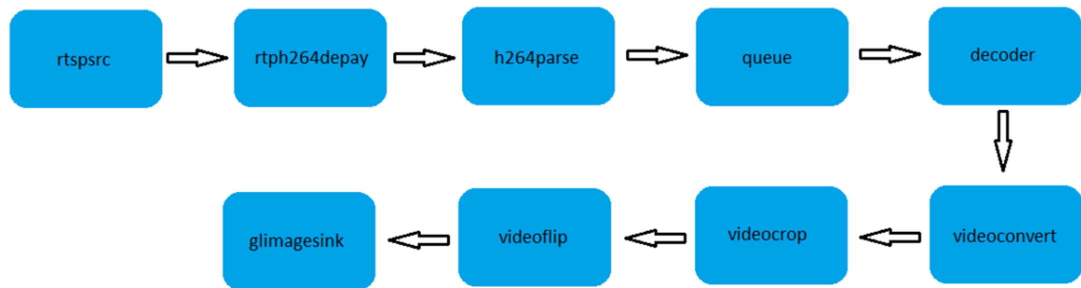

Figure 2.4 *An illustration of a real GStreamer pipeline with a source element, multiple filter elements and a sink.*

rtspsrc

The dataflow in this pipeline starts from the source. The source element is an *rtspsrc*, because the network protocol used in the streaming is RTSP. If we would use, e.g., HTTP we would need to have another source element and make some other modifications to the pipeline. The purpose of the source is to request a stream from the camera. This can be provided with additional parameters, such as the latency and whether to drop the connection in case of too much latency. The source negotiates the streaming details with the camera, such as the used transfer protocol and utilised ports. Once this is finished, the data starts streaming from the camera to the source element. The camera itself is usually pre-configured to some certain settings regarding the encoding method, resolution and compression of the stream. [13]

rtph264depay

The next element in the pipeline is a filter, *rtph264depay*, which by nature is a depayer object. The purpose of this element is to remove the RTP payload from the coming bitstream and pass on the rest of the data to the next element. [12]

h264parse

The *h264parse* element is used to parse the incoming bitstream in both H.264 and AnnexB standards. It provides the following element with a decodable H.264 bitstream that has the Network Abstraction Layer (NAL) units ordered the correct

way. The payload data in H.264 is transferred using NAL units and there are a number of different NAL units for different purposes; the coded slices contain encoded frame data, whereas, Sequence Parameter Set (SPS) and Picture Parameter Set (PPS) contain parameters related to the whole video stream or certain slices and macroblocks. [9, 30]

queue

The *queue* element is used to buffer the bitstream to allow the decoder to decode the stream with all of the needed slices and macroblocks required for the next frame. It also launches a new thread for the rest of the pipeline that continues from its source pad. The queue can be set to buffer a certain maximum amount of data and can be set into a leaky state using its properties. [11]

decoder

The decoder makes the most difference in the pipeline, since it does most of the work in transforming the encoded bitstream into renderable frames. The decoder is not specified here for the reason that there are two different decoders in use; the software decoder *avdec_h264* and the hardware accelerated decoder *vaapidecode*. [10, 14]

videoconvert

The *videoconvert* element is used to convert the stream type when the *vaapidecode* element is used in the decoding. The *videocrop* element is not compatible with *vaapidecode* which is why *videoconvert* is needed. When using *avdec_h264* this conversion is not needed and *videoconvert* only passes on the buffers it receives. [16]

videocrop

There are situations where the area of the frame needs to be altered. In practice, this means cutting away a certain amount of rows or columns from a frame. The *videocrop* element is used for that. It has four properties for cropping a frame from top, right, left and bottom of a frame. After cropping, the *videocrop* element passes on a cropped frame to the next element in the pipeline. [17]

videoflip

The *videoflip* element can be used to perform a flipping or rotating operation to a

frame. The element can perform the rotation in 90 degree increments and flipping along vertical, horizontal and both diagonal axes. [18]

glimagesink

The last element in the pipeline is the sink. In this thesis we used the *glimagesink*, which is an Open Graphics Language (OpenGL) enabled element. This means that the rendering in OptoMon is performed on the GPU even if all the other processing blocks were still implemented using the CPU. One of the reasons the *glimagesink* was selected was in fact the OpenGL capability. In addition, it was also the best performing sink from the selection of sinks that were tested; the other possibilities being *ximagesink*, *xvimagesink* and *vaapisink*. [7, 19, 20, 15]

2.4 Platform requirements and goals

The main requirement for the platform is to be able to stream multiple FHD streams simultaneously. Accomplishing this requirement will be a great benefit for the platform, since it is the largest weakness in the existing platform. In addition, the latency of the streams needs to be within reasonable limits that suit real-time teleopration. The new platform should perform equally or better than the existing platform. Further, the platform is to be independent of proprietary platforms and support GPU plugins in its streaming pipeline.

Dependency with Axis' DirectShow plugin

OptoMon's existing media streaming pipelines are based on the DirectShow multimedia API. DirectShow is made and maintained by Microsoft and it is a part of the Windows software development kit (SDK). OptoMon utilizes a DirectShow plugin made by Axis Communications for the DirectShow multimedia API since the cameras used by the consumers are often Axis cameras. The aforementioned plugin is used to input the streams from the cameras into a DirectShow pipeline. The problem with the plugin is that it is no longer officially supported by Axis. Instead, Axis offers a new component to replace the old one, except the two are not interchangeable since the new Axis Media Control ActiveX component works in a different way to the old [2].

The main issue with the plugin is that it leaks memory [5]. In addition, there are no fixes to be seen by Axis since the plugin is not officially supported anymore. On top

of this, although the plugin works most of the time, it is not capable of streaming multiple HD video streams. To solve these issues we decided to replace the plugin and DirectShow with a better supported and performing multimedia solution.

Lack of GPU plugins in DirectShow API

The main motivation for creating a new generation of OptoMon was to get a better performing and flexible platform. With HD and beyond resolution in cameras it becomes necessary to talk about using GPUs in the processing. DirectShow does not offer such a solution out of the box. It would have meant considerable amounts of programming to achieve these desired goals and DirectShow might not have been the best environment in which to place that effort.

In addition, the very first task in this project was to determine the technology that would allow for the accomplishment of goals with OptoMon. Thus, GStreamer was chosen to be that technology. The main undesirable features of DirectShow were its lack of support for GPUs, dependency of a proprietary platform and difficulty to modify existing plugins, and plugin creation.

3. PLATFORM DEVELOPMENT

In this chapter the development of the new platform is described. The development work is split into two parts. In the first part a version of the platform that was developed using the Qt framework is presented. In addition, its design principles and an analysis of its features and short-comings is revealed. In the second part an enhanced platform developed using the Glib framework is presented. Also, a description of its architecture and problems solved from the Qt-based version are discussed.

3.1 Development using Qt

The first version of the new platform was decided to be built around Qt. Qt is a cross-platform application development framework [27] and provides tools to make a graphical user interface (GUI) for our application. GStreamer also has a wrapper for Qt that allows for the integration of streaming pipelines into the application.

Design principle

A list of basic functionality was defined for the platform and consisted of the following:

- A way to configure the stream parameters
- Starting a stream
- Closing a stream
- Cropping the frame of a stream
- Rotating the frame of a stream

```xml
<?xml version="1.0" encoding="utf-8"?>
<ViewConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Streams>
    <Stream ID="Axis_Q1765-LE_Stream" CameraID="88" Type="Video"
        Device="Axis">
      <Host>110.120.130.140</Host>
      <Username>user</Username>
      <Password>pswd</Password>
      <Codec>H264</Codec>
      <TransmitType>UnicastRTP</TransmitType>
      <UseVaapi>true</UseVaapi>
    </Stream>
  </Streams>
  <StreamStyles>
    <StreamStyle ID="BasicStyle">
      <TopCrop>0</TopCrop>
      <RightCrop>0</RightCrop>
      <BottomCrop>0</BottomCrop>
      <LeftCrop>0</LeftCrop>
      <Rotation>0</Rotation>
    </StreamStyle>
  </StreamStyles>
  <StreamPanels>
    <StreamPanel ID="Axis_Q1765-LE_StreamPanel"
        StreamRef="Axis_Q1765-LE_Stream" StreamStyleRef="BasicStyle" />
  </StreamPanels>
  <Views>
    <View ID="Axis_Q1765-LE" DestinationArea="VideoPanel">
      <CompositePanel ID="Fill" Weight="100" Orientation="Horizontal">
        <VideoPanel ID="Left" Weight="100"
                StreamPanelRef="Axis_Q1765-LE_StreamPanel" />
      </CompositePanel>
    </View>
  </Views>
</ViewConfiguration>
```

Program 3.1 *An example of a configuration file. The file configures one stream with one style and one streampanel in one view.*

By using these basic features it is possible to set up a stream, close it down and modify the frames captured by a camera. One important feature of the platform is the ability to freely configure the used streams and set required parameters. For

this, a scheme, independent of the source of the configuration data, was made. The configuration could be read, e.g., from a database or a file. An example of such configuration is shown in Program 3.1.

The key information in the configuration are the streams, stream styles, stream panels and views. A stream specifies the address of the camera, its username and password, which encoding method to use, transmission type and whether to use hardware acceleration in decoding. The stream style specifies how to crop the frames and what kind of rotation should be applied. The stream panels are the individual streams that are rendered on the screen. The stream panel combines a specific stream with a specific style. Views are compositions of streams that are configured in a specified layout. In the example one stream is configured in the view, however more complex views are possible. For instance, a quad view, where four streams are ordered in a two-by-two grid can be created.

The ability to configure the used decoder was added to this version of the platform. There are two possibilities for a decoder; the software decoder *avdec_h264* and the hardware accelerated decoder *vaapidecode*. The choice for the decoder is not trivial, since the software driver enabling hardware acceleration supports all except one H.264 profile; the baseline profile. This makes choosing the decoder tricky, since many of the cameras used by the customers are models so outdated that they do not support other profiles. The newer cameras have usually the following list of supported H.264 profiles:

- Constrained Baseline
- Main
- High

A stream having any of the above profile could be efficiently decoded using the *vaapidecode* element. A selection of H.264 profiles is shown in Figure 3.1. As can be seen, different profiles use different sets of tools in the encoding process and the decoder has to be able decode the stream using the same tools.

As discussed earlier in Section 2.3, the sink in the pipeline is the element that is used to render the video frames on the display. In OptoMon, every individual stream from a camera would have its own pipeline and the pipeline would have a target window on which to render the frames.

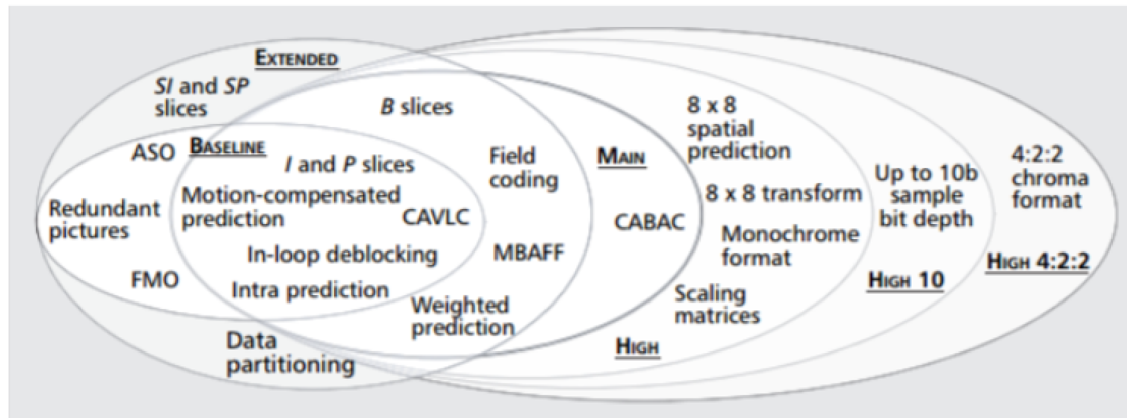

Figure 3.1 *A selection of H.264 profiles and coding tools used by the different profiles.[25]*

On the application side, there would usually be more than one video stream playing at the same time. This means that on the platform side an equal amount of streams would have to exist. It made sense to have a stream object for each stream that had a reference to a specific window on the GUI upon which to render. The Stream class that handled the pipeline construction and all GStreamer related activities was inherited from a Qt widget that would also perform the rendering. A block diagram of the architecture is shown in Figure 3.2.

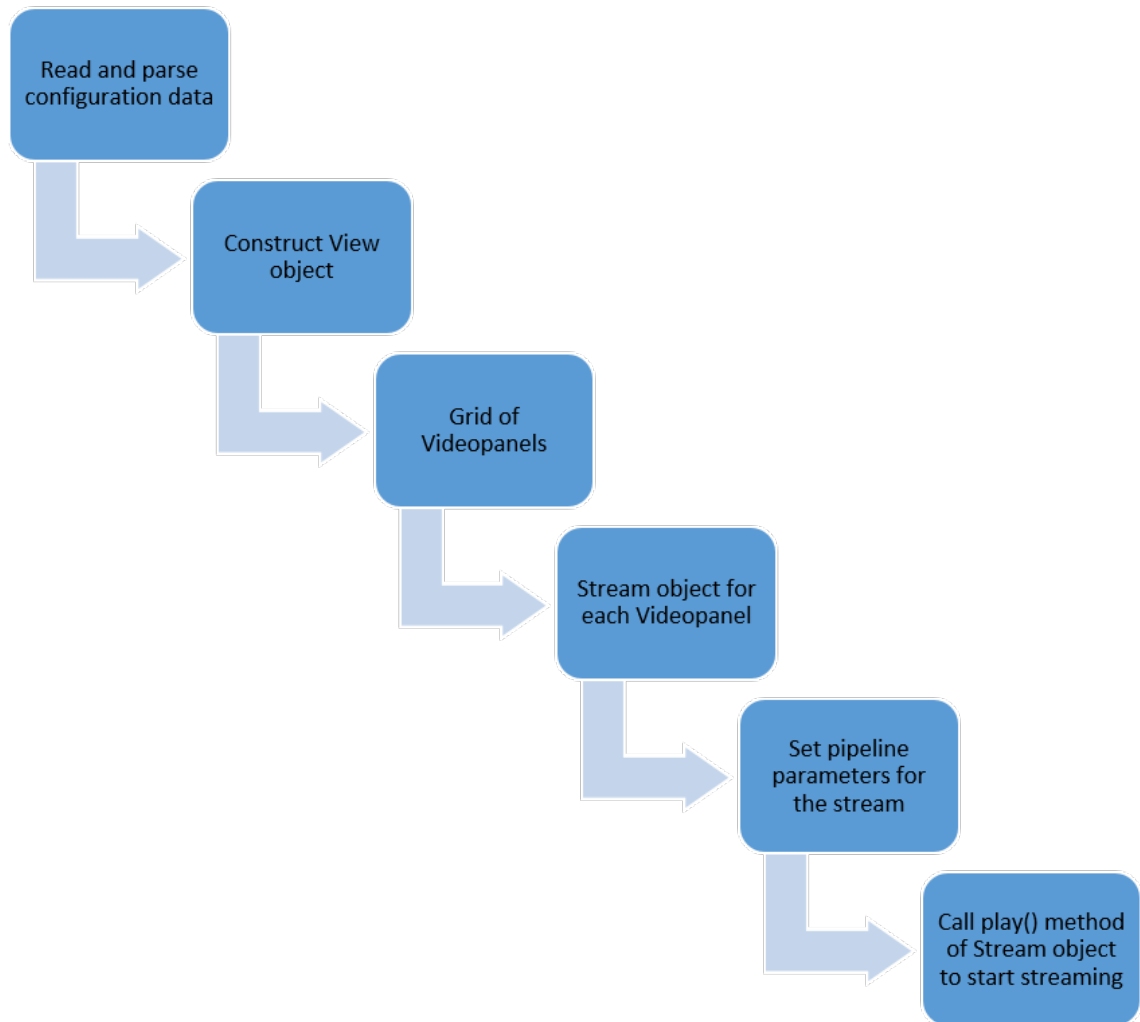

Figure *3.2 Platform workflow from configuration to streaming.*

As shows in the diagram, the configuration is read and parsed from the configuration data. Based on the configuration a view object is created that holds all the streams

in a layout specified by the configuration. A grid layout was chosen due to its versatility when configuring arbitrary 2D layouts. A stream object is created for every videopanel object in the layout and the stream parameters are set according to the configuration. Lastly, the play()-method of each stream object is utilised to start the streaming. The platform connects to the cameras and employs the GStreamer pipeline to stream and render to a specific location on the display. The application may allow the user to be able to change the views and the platform will open and close streams according to the configuration. The architecture of the platform is presented in Figure 3.3.

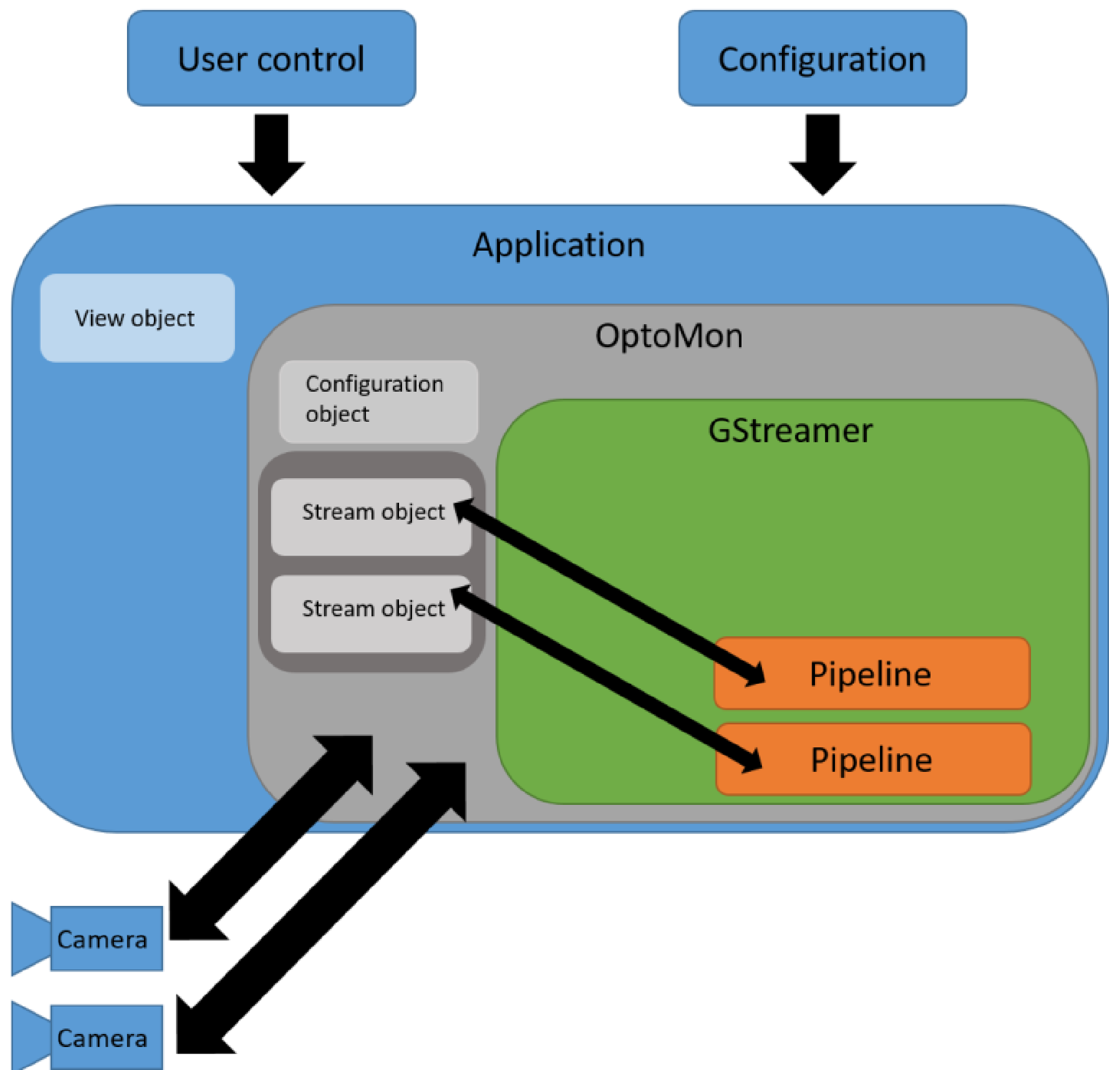

Figure 3.3 *Architectural overview of the Qt-based platform.*

In the figure, the different modules and their relation in the platform can be examined.

Problems with the design

As was stated earlier, the Qt version of OptoMon had issues. The severity of these issues ultimately lead to the rewriting of the whole platform from scratch. The main issues related to this first version were the seemingly random crashes during

view switching in the application. A view in the context of this thesis is a layout configuration of windows upon which each would have a stream playing. A fairly common view for the users was a quad view. At the time the application switches a view it first needs to stop the playback of each stream one by one, then destroy the Qt widgets representing the windows in the view, each containing a GStreamer pipeline, and finally construct the new view with new widgets. After the new windows had been created the application would configure each new stream with their corresponding parameters and start each of the new pipelines.

The issue in this process was that there was practically no way of knowing the timing of each of these steps. Ultimately, there were some steps that needed to be made in a specific order. First, the streaming would need to be stopped. For this the client computer would send a STOP message using RTCP to the server; the server being the camera. After getting acknowledgement and reply message from the server to the STOP message, the client computer would send a TEARDOWN message, which would tell the server to tear down the streaming pipeline. After the TEARDOWN, the server would reply to the client with an OK message. The opening and closing of the streams would fail sometimes, because of thread deadlocks. A deadlock is a situation where one thread wants to access a resource, but that resource is locked by another thread that has been suspended. This leads to the resource being locked indefinitely and that no thread can access it, leading to a deadlock situation [24].

The deadlock situation would not always crash execution and it would seem that everything is working as expected. In reality, the deadlock would reserve an X server connection from the operating system. The X server is an essential part of the X window system, also called X11, on UNIX-like operating system such as Linux. The X11 has an X server on a machine which works between the computer hardware; the graphics card, mouse and keyboard, and the applications which on a GUI have their own X client. Basically having multiple windows open on Linux means having multiple X client connections to the X server [28, 22]. The X11 has a maximum capacity of connections since it was discovered by trial and error that connecting the 257th window resulted in the system becoming unresponsive. The maximum amount of connections can be changed, but since the code severely misbehaved, the appropriate decision was not to try and go around it, but to fix it.

3.2 Development using Glib

As discussed earlier, the Qt version was not a successful attempt in creating a multimedia streaming platform. One of the issues was the outdated and incomplete Qt wrapper for GStreamer and while replacing it with another wrapper could have been an option, it was decided not use any wrappers. Instead, the same programming environment used for making GStreamer would be used, namely the Glib library for C language. Glib is a general-purpose utility library providing a cross-platform interface for application making [22]. Essentially it provides C programming language with a C++ type of functionality, such as object-oriented programming scheme, threading and many other useful features not found in plain C language.

One important reason for choosing C and Glib was the need to make a unique functionality to the platform. In GStreamer context this would mean making our own plugins and elements. The GStreamer elements are made using C and Glib. Therefore, is was a perfect match to learn and use Glib while making OptoMon too. Further, since GStreamer is an open source platform, modifications to the existing plugins can easily be made. This would require knowlegde of the development of GStreamer itself. There is one downside with Glib and that is the amount of boilerplate code that has to be made for classes in comparison with other modern programming languages, such as C#.

Design principle

For this enhanced version of OptoMon the whole implementation had to be rewritten from scratch. That is why the existing functionality, including the decoder selection, was decided to be implemented first from the Qt version. Additionally, there was the need to make OptoMon into a library that could more easily be maintained and deployed into more than one application. This lead to the rethinking of the architecture, and realization that instead of providing stream objects to the application, OptoMon should manage the streams by itself. The platform would be used to ask for a stream based on a configuration object and a target window on which to render. The application would not even need to know anything about the streams. It would be enough to return a "yes" or "no" answer to the stream request in regards to whether or not it could be done. An overview of the architecture is presented in Figure 3.4.

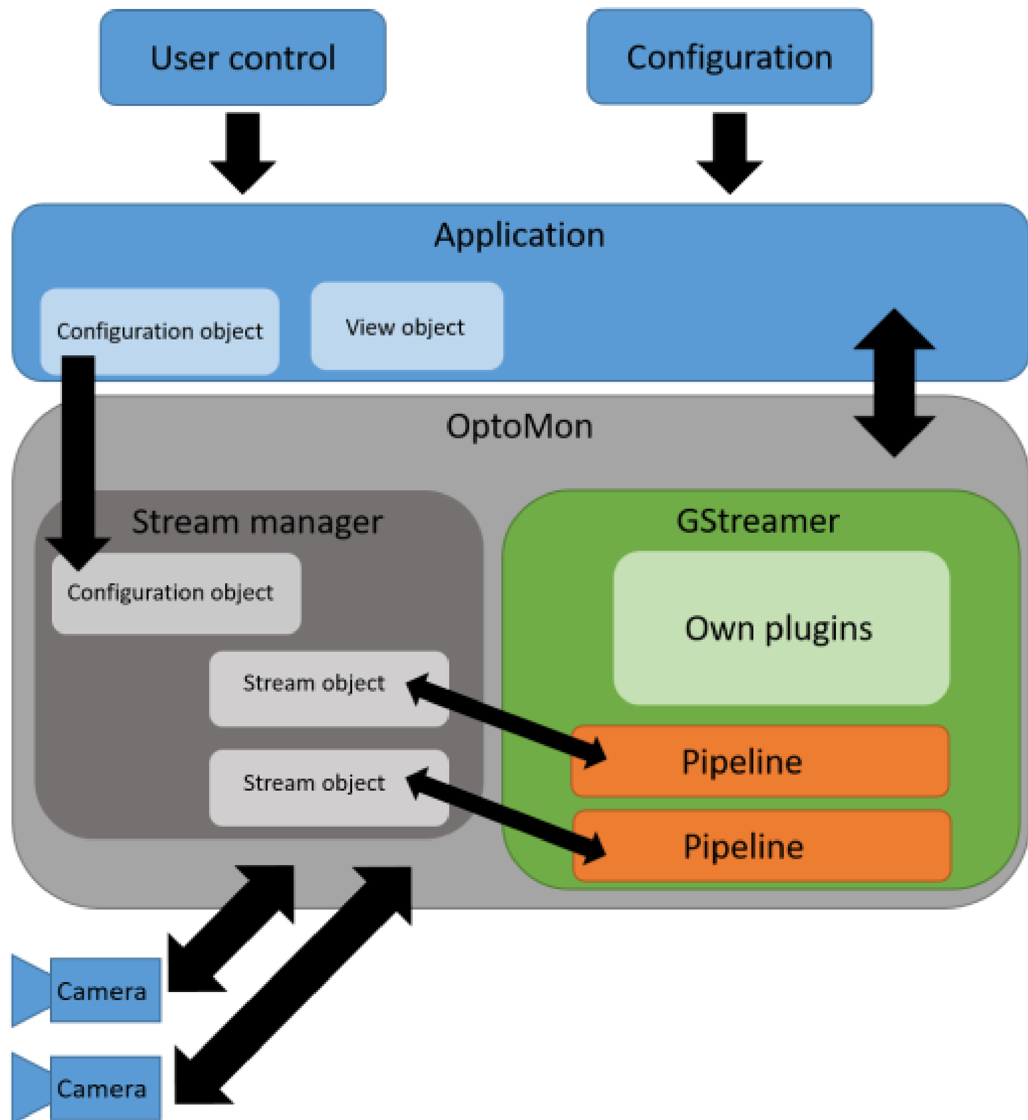

Figure 3.4 *Architectural overview of the Glib-based platform.*

Comparing the architectures presented in Figures 3.3 and 3.4 it can be noticed that the latter has had a few changes. The most obvious change is the separation of application and platform. The configuration scheme is the same as with the earlier platform, however, in addition to the application's configuration object there is also a separate configuration object for the Stream Manager on the platform. A place for self-made plugins for the GStreamer framework has been added also.

The workflow of the new platform is shown in Figure 3.5.

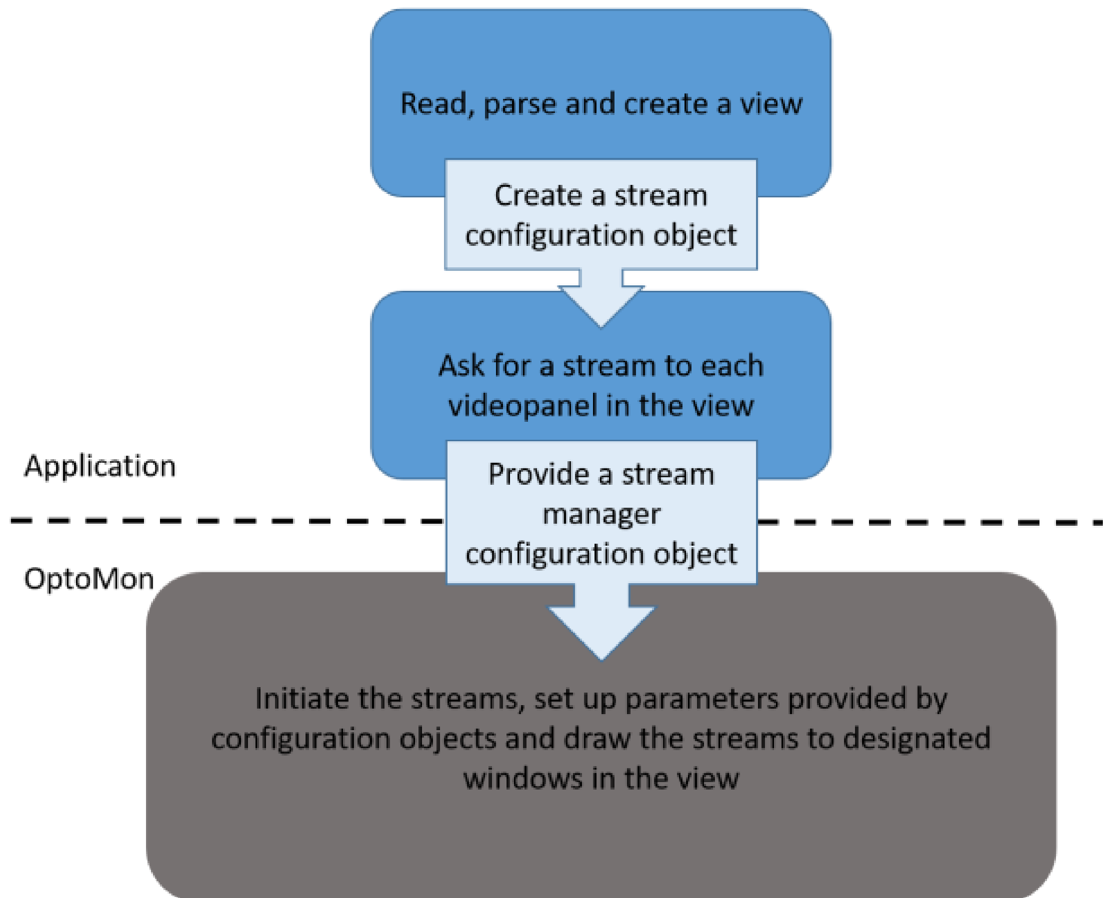

Figure 3.5 *Workflow of the Glib platform version.*

In the architecture the application reads, parses and creates a configuration object. The application takes care of creating a view and requesting a stream for each of the video panels in the view. The application provides a Stream Manager Configuration object to OptoMon containing the information of the requested stream and the window ID of the target window. The Manager object in the platform takes care of the streams, configures a stream for the application and draws the stream on to the provided window. On view change the application asks the manager to stop the streaming of a particular stream and provides a new Stream Manager Configuration object. The Manager object takes care of the stopping and starting of the new

stream.

The workflow of the Stream Manager module is shown in Figure 3.6.

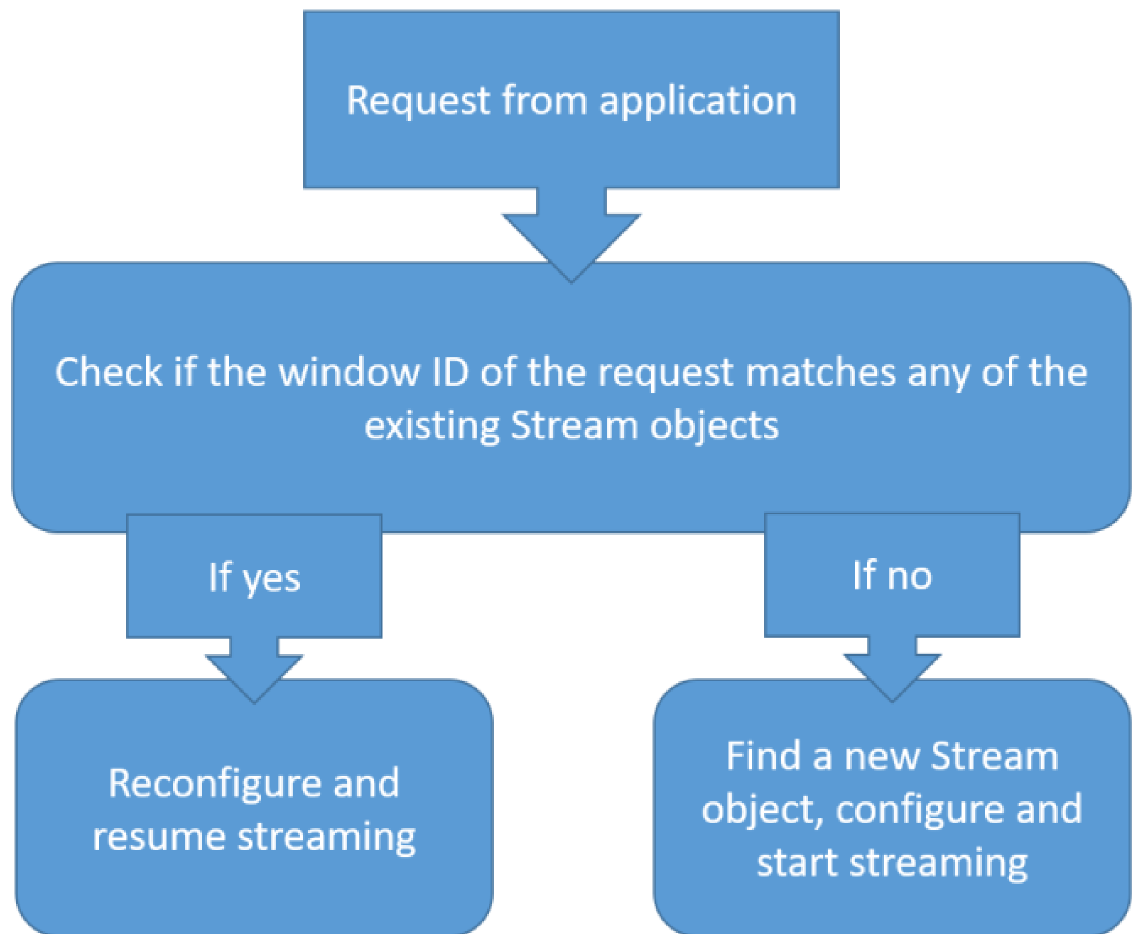

Figure 3.6 *Workflow of the Stream Manager module.*

The application requests a stream from the manager, which first checks if the requested stream is designated to an existing window ID. If it is, the manager stops, reconfigures and resumes the streaming using the same Stream object. Otherwise, the manager checks if it already has an idle Stream object available for use. If it does, then it is used, however if not, the manager will create a new Stream object.

After the Stream object has been determined the manager will configure it according to the request and start streaming. From the application point of view the stream change is faster and does not exhibit a blinking effect if the view configuration stays the same. That is, the same window IDs are used in the view. The deployed Stream objects are put to an active stream's list, while the stopped and displaced streams are put to a stream pool for later use. In this architecture no stream is destroyed and new streams are always drawn from the pool by default.

At this stage it would be the application's responsibility to close the stream. Later there could be a signaling from the application window to OptoMon, e.g., about the closing of a window, while OptoMon could manage the closing of the stream on its own. Using this new architecture and programming tools the absence of crashing on stream change was noticed. In addition, a clear separation of platform and application code was acquired, which made it much easier to manage the code and functionality.

The same configuration scheme was implemented in C using Glib, simply because there was no need to change it. It is flexible, allowing for complex view configurations and extendable with the possibility of adding new parameters to stream and view configurations.

Benefits of the design

A clear benefit of the new design was the stream handling procedure which fixed the crashing and deadlock issues experienced with the Qt version. The separation of platform and application code made it easier to maintain both code bases and develop multiple OptoMon-powered applications. There was no longer a need to update changes made to OptoMon in multiple projects, but updating the library would be enough.

A major step forward was also the increased knowledge of GStreamer via the use of Glib which allowed us to better understand how it works. This was also beneficial considering the fact that there might be a need to debug and create self-made GStreamer plugins in the future. In addition, learning Glib made it possible to dive deeper in GStreamer and its concepts, such as the bus, messages and pads [8].

4. PLATFORM EVALUATION

In this chapter the new platform is evaluated and its suitability for replacing our existing platform examined. At first a benchmarking method is presented. It will be used as the main target for our platforms. The evaluation in this thesis is performed by analyzing the measured latencies in the streams and the resolution of the streams. A low latency is better and having the latency as a function of resolution gives us a sense of how much the increased amount of data affects the streaming performance. The device used in the measurement is the OptoFidelity Video Multimeter, the company's own video performance analyzer. The measurements for the benchmark will be shown first. After that, the performance metrics of the old OptoMon will be shown. Then, the measurement metrics of the new platform will be presented. In the end, there will be a comparison of the old and new platforms.

4.1 Latency benchmark

We used two different camera models in the testing of the platforms. Both are Axis' cameras and the models are Q1765-LE and Q3709-PVE. They both produce up to FHD video, H.264 encoded bitstream and the Q3709-PVE also allows us to test the limits of our platform with its UHD resolution support. The cameras were connected to a Cisco Gigabit power over ethernet (POE) switch, since both of the cameras take their power from the Ethernet port. The cameras were configured to the same network as our computer that was running the OptoMon-powered test application.

Figure 4.1 *Latency measurement setup of the display using the OptoFidelity Video Multimeter.*

Figure 4.2 *Latency measurement setup of the camera and LED targer using the OptoFidelity Video Multimeter.*

The benchmark for our measurements is the Axis Live View web server running on each camera. Using this tool enables us to stream a high quality, low latency video stream and measure its latency. The test setup is presented in Figure 4.1 and Figure 4.2. In Figure 4.1 the Video Multimeter is measuring the latency from the display. The optical fibre on the display is connected to the Video Multimeter allowing continuous measurement. The LED shown in Figure 4.2 is blinking at a known threshold. The camera is capturing the blinking and the resulting video stream is presented on the display where the latency is being measured.

The stream from Live View is an MJPEG stream which makes it propitious in the latency sense since each frame is processed in a minimal way when compared to, e.g., H.264. MJPEG stream causes a larger load on the network than H.264, but since we only have one stream at a time the impact of network on latency can be neglected. The benchmark measurement are displayed in Table 4.1.

Table 4.1 *Benchmark latencies measured from Axis Live View*

Camera	Codec	Resolution	Latency (ms)
Q1765-LE	MJPEG	FHD	180 ± 24
Q3709-PVE	MJPEG	FHD	235 ± 18

As can be seen from the table, there are significant differences between cameras which makes choosing a camera not self-evident. The average latency has a rather large variation, but the difference between the latencies from these two cameras is apparent. We are using the latency of a FHD stream as a benchmark for all stream resolutions from both versions of OptoMon, since it is expected that with lower than FHD resolution the latency is less than the benchmark and with higher resolution it will be greater.

4.2 OptoMon v1 latency measurements

We measured the latencies of OptoMon v1 using the same cameras as in our benchmarking measurements. We used both cameras on three different resolutions, in addition to the Q3709-PVE with UHD, and with two different codecs; the MJPEG and H.264. The results are shown in Table 4.2.

As can be seen from the table, the MJPEG streams from Q1765-LE perform well compared to the benchmark up to HD resolution, but with a larger resolution the latency increases dramatically. Whereas, the MJPEG streams from Q3709-PVE perform exceptionally well when compared to the benchmark; the latency is clearly shorter. The latencies from Q1765-LE using H.264 are not as good as with MJPEG on SD and HD resolutions, but on FHD they are the same within the measurement accuracy. On the Q3709-PVE the H.264 streams are more stable, but introduce a larger latency compared to benchmark. Surprisingly, the measurements from Q3709-PVE are similar within the limits of measurement accuracy. It was not possible to measure any latencies on UHD, since the stream had a lot of missing frames and the motion was very jerky. This makes measuring the latency impossible using the video multimeter.

OptoMon v1 performance

Additionally, measurements were performed on a configuration of four streams play-

Table 4.2 Latency measurements of OptoMon v1 platform

Camera	Codec	Resolution	Latency (ms)
Q1765-LE	MJPEG	SD	164 ± 15
		HD	174 ± 15
		FHD	262 ± 15
	H.264	SD	208 ± 15
		HD	218 ± 15
		FHD	265 ± 15
Q3709-PVE	MJPEG	SD	193 ± 15
		HD	207 ± 15
		FHD	216 ± 15
		UHD	N/A
	H.264	SD	250 ± 15
		HD	243 ± 15
		FHD	256 ± 15
		UHD	N/A

ing at the same time and measured the latencies as above from one of the four streams. The results of this quick test are shown in Table 4.3.

Table 4.3 Latency measurements of four simultaneous streams using OptoMon v1 platform

Camera	Codec	Resolution	Latency (ms)
Q1765-LE	H.264	SD	196 ± 15
		HD	206 ± 15
		FHD	N/A

These tests were performed to verify our experiences with higher resolutions for the fact that OptoMon v1 was not able to stream multiple FHD streams at the same time. In addition, they were performed to later prove the point that OptoMon v2 was in fact more efficient when processing multiple high resolution streams at once. As can be seen from the measurements in Table 4.3 the SD and HD latencies are measurable and in fact rather good results, but using FHD resolution the latencies were not being able to be measured at all. The streams had multiple seconds of latency, something which could be determined simply by looking at the streams. The video multimeter was not able to measure the latency in this case, because the latency was more than the measurement lights blinking frequency.

4.3 OptoMon v2 latency measurements

The measurements for OptoMon v2 were performed in the same way as with OptoMon v1 with the addition of hardware decoded streams for both MJPEG and H.264 codecs. It was not possible to make any MJPEG measurements using Q3709-PVE, since GStreamer was not able to produce a reliable pipeline. The measurement results are presented in Table 4.4.

Table 4.4 *Latency measurements of OptoMon v2 platform*

Camera	Codec	Resolution	Latency SW decoded (ms)	Latency HW decoded (ms)
Q1765-LE	MJPEG	SD	165 ± 15	232 ± 15
		HD	170 ± 15	214 ± 15
		FHD	192 ± 15	203 ± 15
	H.264	SD	130 ± 15	210 ± 15
		HD	145 ± 15	215 ± 15
		FHD	185 ± 15	245 ± 15
Q3709-PVE	H.264	SD	207 ± 15	243 ± 15
		HD	215 ± 15	248 ± 15
		FHD	228 ± 15	258 ± 15
		UHD	N/A	312 ± 15

As can be seen from Table 4.4, similar or even better results than the benchmark can be reached using OptoMon v2. This requires selecting a proper camera and codec combination, but it can be clearly seen that it is possible. The MJPEG measurements are similar with OptoMon v1 with the exception that the measurement made using FHD resolution does not spike in the same way it did for OptoMon v1. The FHD MJPEG measurement is still a little larger than the benchmark, but it is very close. The hardware accelerated decoding results seem peculiar using MJPEG, since it is apparent that the latency is decreasing as the resolution is increasing, while the opposite would seem like the right behaviour. It is unclear as to why this happened, but although the trend would be to decrease as a function of resolution, it is not believed that this is in fact the case. There might be an issue in the hardware accelerated JPEG decoder or it could be an issue with the measuring equipment.

The H.264 measurements are equal or better than the benchmark using software decoding for both cameras. The H.264 measurements using hardware accelerated decoding result in a slightly longer latency overall, but compared to the benchmark they are very close with Q1765-LE within the limits of measurement accuracy. For

Q3709-PVE the hardware accelerated decoding latencies are longer than the benchmark on all resolutions. In addition to all earlier measurements, the latency of an UHD stream is being able to be measured. This was possible only using hardware accelerated decoding. The latency is a lot more than the other measurements which can be partly explained with the greatly increased amount of data to be processed by a pipeline in both encoding and decoding ends and in the transfer media.

OptoMon v2 performance

One of the key aspects of making a new version of OptoMon was to have a more powerful platform for multiple simultaneous streams. Now that the latencies using the system had been measured, it was time to evaluate the performance and put it to a more serious test. Performance for OptoMon v2 was examined using the operating system's System Monitor applet, which is the equivalent of Windows' Task Manager on Linux. We were particularly interested in the CPU usage of the system during streaming. All the measurements were performed using 16 simultaneous streams each presenting a view with varying motion. The used cameras were the same two cameras as earlier and the used codec was H.264 with 25 fps and variable bitrate. The measurements are presented in Table 4.5.

Table 4.5 *Performance measurements of OptoMon v2 platform*

Stream setting	HW decoded CPU usage %	HW decoded CPU peaks %	SW decoded CPU usage %	SW decoded CPU peaks %
16 x HD	20	-	45	65
16 x FHD	20	-	60	80
4 x FHD + 9 x 4K	35	-	95	-
16 x 4K	35	-	95	-

As can be seen from Table 4.5, the HW decoded streams put the CPU on a much more lighter load compared to the SW decoded streams. In addition, using SW decoding the CPU usage peaks during rapid motion resulting in missing frames, jerkiness of motion or other visual artifacts in the video playback. The dashes in the table represent a non existing value. For example, HW decoded streams did not exhibit any peaks even during rapid motion. On the other hand, when UHD streams were in use the CPU usage was already at its maximum so no peaks could be made visible. The results prove that hardware accelerated decoding is clearly the only way to enable high resolution streaming using multiple video sources.

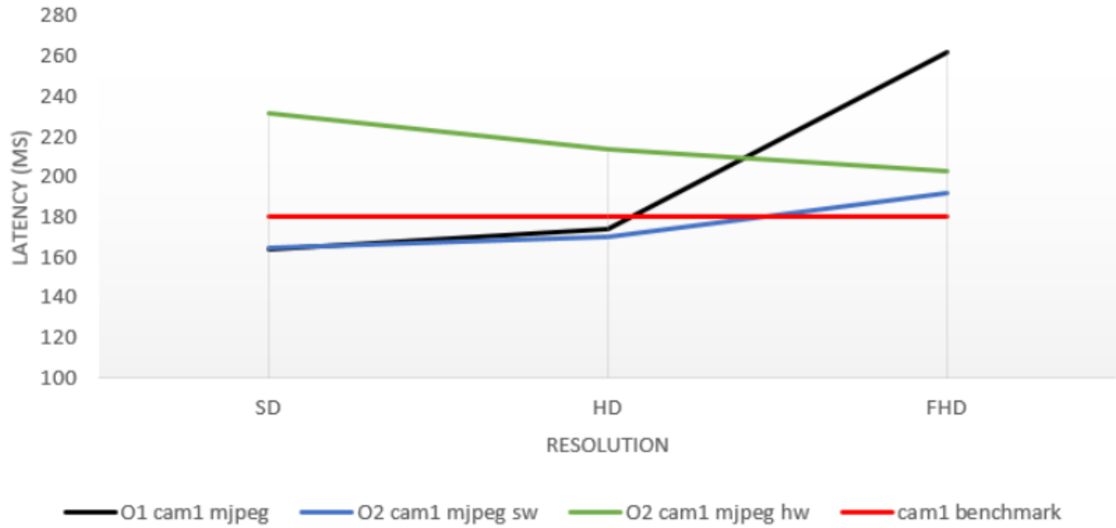

Figure 4.3 *Latency measurements of both OptoMon platforms using camera Q1765-LE and MJPEG.*

One interesting observation during these performance measurements was that the network became the bottleneck in the system. From Table 4.5 it can be seen that even though the amount of data to be streamed increased from four FHD and nine UHD streams to 16 UHD streams, the CPU usage did not change. Even more interesting was that the CPU usage did not increase while the CPU clearly was capable of extra processing. The network load was measured to around 72 MB/s in both cases, indicating that the network was in fact limiting the streaming.

4.4 Comparison between platforms

Now that measurements have been presented and observations made, it is time for a more thorough comparison and analysis of each platform's capabilities. By examining the tables and figures in this chapter it is noticed that OptoMon v1 can barely keep up with the benchmark. As Figure 4.3 and Figure 4.4 show, it is able to compete with the benchmark on HD and lower resolution on MJPEG, but other than that it starts to lag behind in latency. There is one exception to this, which is using the Q3709-PVE and MJPEG. However, an even comparison cannot be made,

since OptoMon v2 could not initiate a stream using the same configuration. This could on the other hand be seen as one of the few pros with OptoMon v1 over v2. Compared to the single streaming latency measurements, a similar behavior can be seen from the performance measurements using multiple streams as FHD multistreaming was not even possible.

Figure 4.4 *Latency measurements of both OptoMon platforms using camera Q3709-PVE and MJPEG for OptoMon v1 and H.264 for OptoMon v2.*

On the contrary, OptoMon v2 is similar or better in the latency sense on H.264. As Figure 4.5 and Figure 4.6 show, using software decoding OptoMon v2 is better than OptoMon v1 on all resolutions. Hardware accelerated streams using OptoMon v2 are comparable with those of OptoMon v1 using H.264. Due to the problem with OptoMon v2 MJPEG measurements, as seen in figure 4.3, it is hard to say whether the hardware accelerated measurements are better than those of OptoMon v1, but the software decoded streams apper to be better.

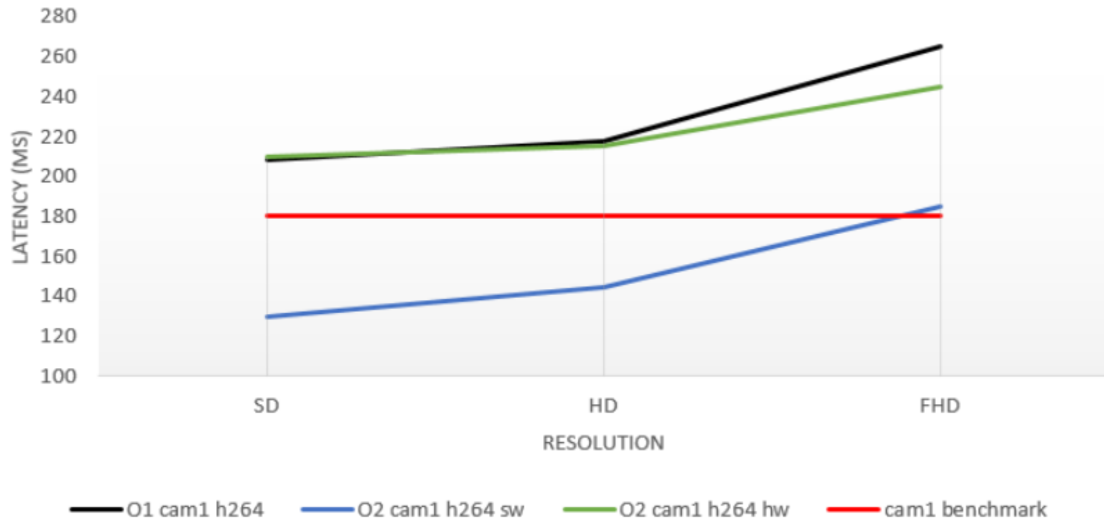

Figure 4.5 *Latency measurements of both OptoMon platforms using camera Q1765-LE and H.264.*

The true difference between the two platforms comes from the multistream test measurements. As stated above, OptoMon v1 was not able to stream four FHD streams. A massive difference was found with OptoMon v2 using 16 FHD streams on both software and hardware decoding. In addition to this, OptoMon v2 has the ability to decode a number of FHD streams plus a number of UHD streams using hardware decoding, something that is unimaginable for OptoMon v1. The actual number of UHD streams was not possible to determine based on the measurements, since the used network architecture became the bottleneck in the testing system. Based on the CPU usage on our measurements there is clearly capacity, since 16 FHD streams used around 20% of the CPU's resources.

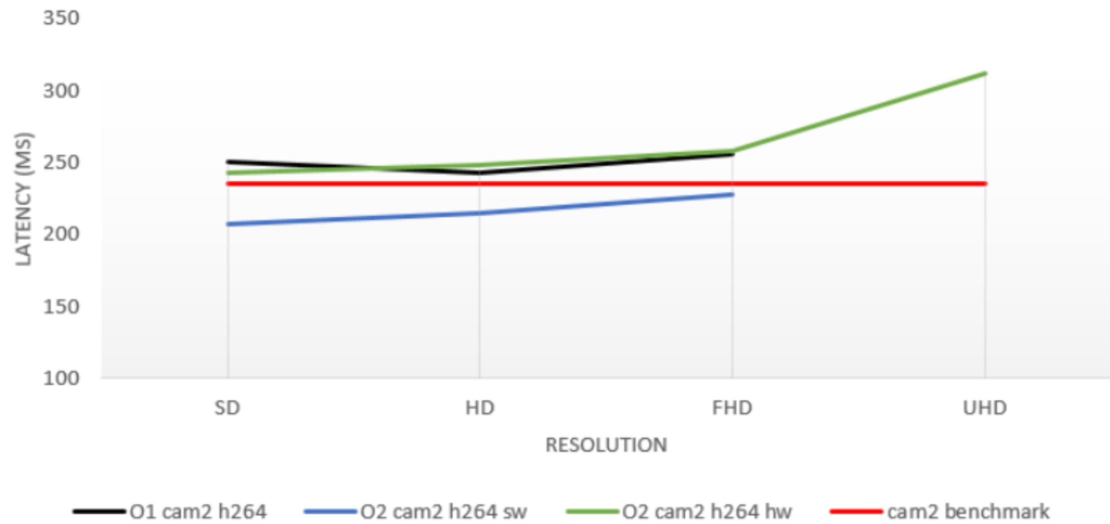

Figure 4.6 *Latency measurements of both OptoMon platforms using camera Q3709-PVE and H.264.*

However, some areas of the new platform are not developed to the same level as with the existing platform. For example, the old platform supports overlay graphics on top of the streams. The overlay graphics allow augmenting the streamed videos with additional information, such as the real-time latency or system specific textual and graphical information. Also, the old platform includes a method for calculating the real-time latency of the stream using timestamps on the video frames. There is also a feature on the old platform that allows stacking the video or graphics streams in an arbitrary configuration. This feature permits overlaying streams on top of each other and adds a third dimensions for the configurability of the platform.

5. DISCUSSION

The goal of this thesis was to develop a new and modern version of the OptoFidelity OptoMon teleoperation platform by solving the issues identified in the existing implementation. The core issues were:

- Decoding of multiple simultaneous streams as only a software decoder was used
- HD quality streams
- Latency with multiple streams
- Limited fps in some environments
- Dependency of Axis multimedia components for DirectShow API
- Dependency of DirectShow API in pipeline construction
- The platform had been developed over time and not necessarily designed to address all the issues it was now facing.

The development of a new platform has been presented in this thesis and that by using the new OptoMon v2 platform it is possible to stream and decode multiple simultaneous streams, even if only a software decoder was used. The platform receives a massive performance upgrade when employing hardware accelerated decoding in the streams. Also, stream quality issues have been solved by making it possible to stream UHD resolution streams. It was discovered that the used decoders did not offer any possibility to affect the way the decoding is performed. In the case of GPU decoding it would be beneficial to have better access to the GPU's resources and affect on its utilization. In addition, we do not have full access to the used cameras and their encoding parameters. At least not with the Axis' cameras.

Latency issues have been solved in the previously problematic case of multiple streams in OptoMon v1. In addition, by carefully choosing an appropriate camera and video codec it has been shown that it is possible to achieve even shorter latencies using the new platform when compared to the old. Furthermore, the fps is no longer an issue, as all measurements featured a full 25 fps, which is the maximum fps available from the used cameras. In comparison with an existing video streaming solution by Ittiam [4, 32], OptoMon v2 cannot achieve as low latencies as the Ittiam solution. It is able to provide an end-to-end latency of 70 ms on 1080p video with 60 fps, while OptoMon is capable of offering a stream with 185 ms of latency using 25 fps and 1080p video. Clearly OptoMon is not able to compete with Ittiam in this area. However, the Ittiam solution is built upon a highly customized hardware solution that is based on a Texas Instruments digital signal processing (DSP) chip. OptoMon, on the other hand, is using a common microprocessor and a separate GPU.

Perhaps the most limiting factors of the old platform were the dependencies and limitations of third party software components and APIs. Namely the Axis multimedia component used in communicating with the cameras and the Microsoft DirectShow API used in pipeline construction. By deploying the open source multimedia framework GStreamer, there is no need for the components provided by Axis, since GStreamer is capable of communicating with the cameras using standardized protocols, such as RTP, RTSP and RTCP. Additionally, the platform is no longer dependent on Axis' cameras, but can use practically any IP cameras given that they fulfill the platform's and the consumer's requirements. GStreamer also enables getting rid of the DirectShow pipeline, which was missing many plugins that were desperately needed. The most critical ones were the hardware accelerated decoding and maximized utilization of the GPU resources in the pipeline.

Using the new platform there are much less limitations in the future, since the platform is based on an open source multimedia framework, which is constantly developed by an active community. If there is something needed in the future it can be relatively effortlessly implemented, or if any bugs are found from the existing code it will be possible to fix them and not being bound to wait for somebody else to do it. In conclusion, the means of achieving real-time video streaming for a teleoperation platform have been successfully created to answer and solve the earlier set needs.

6. CONCLUSIONS

It is clearly possible to reach low latencies when streaming high resolution content in an IP network by using only a software decoder in the streaming pipelines. However, it is a lot more effictive from a computing perspective to use a hardware accelerated decoder which allows for the streaming of either higher resolution streams, or more streams of the same resolution at the same time. As stated at the beginning of this thesis, the users had many problems that were due to the use of DirectShow API in the pipelines. Another way of achieving the same or better results have been presented in the latency sense, a lot better results in efficiency sense and complete freedom from any ties to any software that could not be altered. In this respect, the problem presented in this thesis has been solved.

It was predicted at the beginning of the work that the use of a GPU in the streaming pipeline would allow for the achievement of the main goals of this development. However, it came as a surprise that the latencies were poorer using low resolutions on the GPU than on the CPU. It was also predicted that decoding the same stream with the GPU would result in a lower latency than by using the CPU in the decoding. For higher resolutions this is in fact true, but it was discovered that the lowest latencies for sub-HD resolutions were in fact achieved using a CPU decoder.

These results imply that OptoMon is capable of providing real-time streaming without using any additional hardware components, such as specific encoders or decoders. Using these may allow for a shorter latency, but they instantly add complexity and cost for the overall system. OptoMon is free of these while offering good-enough latency for real-time teleoperation at a lower cost.

Future work

The work presented in this thesis does not mean that the development of the platform has finished. A basis for future development has been merely created. One of the first future additions to the platform will be a latency monitoring mechanism.

Currently, there is no way of monitoring the latency in real-time on the field. However, we have made some initial testing with the timestamping method as discussed in Chapter 2.2. The method was not robust enough, since it only works on certain text fonts. In the future a more robust latency measurement method will be need to be developed. For this, the use of optical character recognition (OCR) algorithms could be experimented to help in recognizing the timestamps.

Another future development subject is adding information on the streams by overlaying graphics. Information such as stream statistics, real-time latency and guidance for the operator such as wind speed and direction indicators. The graphics should utilize the GPU as much as possible and will likely require their own graphics pipeline. There is a likely need for both 2D and 3D graphics meaning that the graphics processing will consume computing resources from the streams. However, the graphics should not under any circumstances interfere or radically increase the stream latency.

BIBLIOGRAPHY

[1] N. H. Amer, H. Zamzuri, K. Hudha, and Z. A. Kadir, “Modelling and control strategies in path tracking control for autonomous ground vehicles: A review of state of the art and challenges,” *Journal of Intelligent & Robotic Systems*, vol. 86, no. 2, pp. 225–254, 2017. [Online]. Available: http://dx.doi.org/10.1007/s10846-016-0442-0

[2] Axis Communications AB, “Windows development, axis media contol,” [online], http://www.axis.com/fi/en/support/developer-support/windows-development, January 2017.

[3] D. Chandler and R. Munday, “A dictionary of media and communication,” 2016.

[4] A. Charbonnier, “Ittiam unveils 60ms ultra low-latency streaming systems for high definition video,” 2009. [Online]. Available: https://www.ittiam.com/newsroom/news/2009-2/ittiam-unveils-60ms-ultra-low-latency-streaming-systems-for-high-definition-video

[5] V. Šor, S. N. Srirama, and N. Salnikov-Tarnovski, “Memory leak detection in plumbr,” *Software: Practice and Experience*, vol. 45, pp. 1307–1330, 2014.

[6] H. Deng, C. Deng, and J. Li, “Gpu-based real-time decoding technique for high-definition videos,” *2012 Eighth International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)*, 2012.

[7] GStreamer community, “glimagesink documentation,” [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-bad-plugins/html/gst-plugins-bad-plugins-glimagesink.html, January 2017.

[8] GStreamer community, “Gstreamer documentation,” [online], https://gstreamer.freedesktop.org/documentation/, January 2017.

[9] GStreamer community, “h264parse documentation,” [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-bad-libs/html/gst-plugins-bad-libs-h264parser.html, January 2017.

[10] GStreamer community, “libav avdec_h264 documentation,” [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-libav-plugins/html/gst-libav-plugins-plugin-libav.html, January 2017.

[11] GStreamer community, "queue documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer-plugins/html/gstreamer-plugins-queue.html, January 2017.

[12] GStreamer community, "rtph264depay documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good-plugins/html/gst-plugins-good-plugins-rtph264depay.html, January 2017.

[13] GStreamer community, "rtspsrc documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good-plugins/html/gst-plugins-good-plugins-rtspsrc.html, January 2017.

[14] GStreamer community, "vaapidecode documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer-vaapi-plugins/html/gstreamer-vaapi-plugins-vaapidecode.html, January 2017.

[15] GStreamer community, "vaapisink documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer-vaapi-plugins/html/gstreamer-vaapi-plugins-vaapisink.html, February 2017.

[16] GStreamer community, "videoconvert documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-base-plugins/html/gst-plugins-base-plugins-videoconvert.html, January 2017.

[17] GStreamer community, "videocrop documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good-plugins/html/gst-plugins-good-plugins-videocrop.html, January 2017.

[18] GStreamer community, "videoflip documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good-plugins/html/gst-plugins-good-plugins-videoflip.html, January 2017.

[19] GStreamer community, "ximagesink documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-base-plugins/html/gst-plugins-base-plugins-ximagesink.html, February 2017.

[20] GStreamer community, "xvimagesink documentation," [online], https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-base-plugins/html/gst-plugins-base-plugins-xvimagesink.html, February 2017.

[21] B. Juurlink, M. Alvarez-Mesa, C. C. Chi, A. Azevedo, C. Meenderinck, and A. Ramirez, *Scalable parallel programming applied to H.264/AVC decoding.* Springer, 2012.

[22] A. Krause, *Foundations of GTK+ Development.* Apress, 2007.

[23] Q. Li, L. Chen, M. Li, S. L. Shaw, and A. NÃ¼chter, "A sensor-fusion drivable-region and lane-detection system for autonomous vehicle navigation in challenging road scenarios," *IEEE Transactions on Vehicular Technology*, vol. 63, no. 2, pp. 540–555, Feb 2014.

[24] M. Ljumovic, *C++ Multithreading Cookbook.* Packt Publishing, Birmingham, GB, 2014.

[25] D. Marpe, T. Wiegand, and G. J. Sullivan, "The h.264/mpeg4 advanced video coding standard and its applications," *IEEE Communications Magazine*, pp. 134–143, 2006.

[26] M. Mihelj and J. Podobnik, *Teleoperation.* Dordrecht: Springer Netherlands, 2012, pp. 161–178. [Online]. Available: http://dx.doi.org/10.1007/978-94-007-5718-9_9

[27] D. Molkentin, *Book of Qt 4 : The Art of Building Qt Applications.* No Starch Press, 2007. [Online]. Available: http://site.ebrary.com/lib/ttyk/detail.action?docID=10202508

[28] V. Quercia and T. O'Reilly, *X Window System User's Guide for X11 R3 and R4*, ser. The Definitive Guides to the X Window System, vol. 3. Sebastopol, CA: O'Reilly & Associates, 1990.

[29] P. Read and M.-P. Mayer, *Restoration of Motion Picture Film*, ser. Conservation and Museology. Elsevier Science, 2000, pp. 22–24.

[30] I. E. Richardson and P. Anthony, *H.264 Advanced Video Compression Standard.* Wiley, 2010. [Online]. Available: http://site.ebrary.com/lib/ttyk/detail.action?docID=10392946

[31] M. Syme and P. Goldie, "Optimizing network performance with content switching; server, firewall, and cache load balancing," 12 2003, copyright - Copyright Book News, Inc. Dec 2003; Last updated - 2010-06-06. [Online]. Available: http://search.proquest.com/docview/200137064?accountid=27303

[32] I. Systems, "Ittiam unveils a full hd low latency streaming solution with a glass-to-glass latency of less than 70ms at nab 2013," Apr 02 2013, copyright - Copyright PR Newswire Association LLC Apr 2, 2013; Last updated - 2013-04-02. [Online]. Available: https://search.proquest.com/docview/1322262002?accountid=27303

[33] J. Vilca, L. Adouane, and Y. Mezouar, "Optimal multi-criteria waypoint selection for autonomous vehicle navigation in structured environment," *Journal of Intelligent & Robotic Systems*, vol. 82, no. 2, pp. 301–324, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10846-015-0223-1

[34] B. Wand, M. Alvarez-Mesa, C. C. Chi, and B. Juurlink, "Parallel h.264/avc motion compensation for gpus using opencl," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, pp. 525 – 531, 2014.