Edge Computing for Mixed Reality

Blandad virtuell verklighet med stöd av edge computing

Johan Lindqvist

Supervisor: Klervie Toczé
Examiner: Simin Nadjm-Tehrani
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Johan Lindqvist
Abstract

Mixed reality, or augmented reality, where the real and the virtual worlds are combined, has seen an increase in interest in recent years with the release of tools like Google ARCore and Apple ARkit. Edge computing, where the distributed computing resources are located near the end device at the edge of the network, is a paradigm that enables offloading of computing tasks with latency requirements to dedicated servers.

This thesis studies how edge computing can be used to bring mixed reality capabilities to mobile end devices that lack native support for that. It presents a working prototype for delivering mixed reality, evaluates the different technologies in it in relation to stability, responsiveness and resource usage, and studies the requirements on the end and edge devices.

The experimental evaluation revealed that transmission time is the most significant chunk of end-to-end latency for the developed application. Reducing that delay will have a significant impact on future deployments of such systems.

The thesis also presents other bottlenecks and best practices found during the prototype’s development, and how to proceed from here.
Acknowledgments

I want to give a very big thank you to my supervisor, Klervie Toczé, for all the help and feedback throughout the work on this thesis. The many discussions about the different aspects of the thesis improved it tremendously. I would like to thank my examiner, professor Simin Nadjm-Tehrani, for all the support, as well as the opportunity to do this thesis. I would also like to thank my opponent Oskar Gustafsson for valuable comments and suggestions for improvements.

Finally, I must express my great gratitude to my parents and my wife, Tina, for providing me with support and encouragement throughout the work for this thesis. This thesis would not have been possible without them.
# Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures vii
List of Tables viii

1 Introduction 1
1.1 Motivation ............................................. 1
1.2 Aim .................................................. 2
1.3 Problem definition ...................................... 2
1.4 Approach .............................................. 3
1.5 Structure .............................................. 3

2 Background 4
2.1 Platforms ............................................. 4
2.2 Edge computing ....................................... 5
2.3 Computer video encoding .............................. 6
2.4 Video streaming ....................................... 7
2.5 Computer vision ....................................... 9
2.6 Related works ........................................ 13

3 The Two Prototypes 14
3.1 Local prototype ....................................... 14
3.2 Edge prototype ....................................... 15
3.3 Network communication for the edge prototype ...... 19
3.4 Video streaming for the edge prototype ............... 21
3.5 Mixed reality for the edge prototype ................. 23

4 Evaluation 27
4.1 Overview .............................................. 27
4.2 Resource usage ....................................... 29
4.3 Responsiveness ....................................... 33
4.4 Stability ............................................... 38

5 Discussion 42
5.1 Method ............................................... 42
5.2 Resource usage ....................................... 43
5.3 Responsiveness ....................................... 43
5.4 Stability ............................................... 44
## List of Figures

2.1 GStreamer pipeline .................................................. 10

3.1 Image from the local prototype, displaying mixed reality content ......................... 15
3.2 An overview of the edge prototype ..................................... 16
3.3 Image from the end device running the edge prototype, displaying mixed reality content ............................................. 17
3.4 The components on the edge device and how they interact ...................................... 17
3.5 The components on the end device, arrows denoting interaction between the components .................................................. 18
3.6 The end device’s application’s start menu .................................. 19
3.7 Example of communication in the Edge Prototype ............................................... 20
3.8 Close-up view of the mixed reality graphics ........................................... 25

4.1 The reference video used for testing ........................................ 29
4.2 Average tracked point cloud features for different streaming methods and bitrates ..... 31
4.3 Processing power usage on the edge device for the different configurations, with and without the Mixed reality component enabled ........................................ 33
4.4 Processing power usage on the end device for the two prototypes .......................... 34
4.5 Time measured for adding virtual objects for the two prototypes ............................ 35
4.6 Round-trip times for the two prototypes ........................................ 36
4.7 The edge device’s components involved in the processing of a video frame ............... 37
4.8 The observed processing times on the edge device ............................................. 38
4.9 Frames per second displayed on the end device running the edge prototype .............. 40
4.10 Frames per second displayed on the end device running the local prototype .......... 40
4.11 Frames per second received by the edge device .............................................. 41
4.12 Percentage not processed, of the number of frames arrived at the edge device .......... 41

5.1 Example of where a broken keyframe has led to also subsequent video frames being incorrect ........................................... 45

A.1 Edge prototype camera calibration as seen on the end device ............................. 57
List of Tables

2.1 The seven layers of the OSI model. ................................................. 8
3.1 The evaluated video streaming and playback libraries for the end device. ......................... 22
3.2 The evaluated SLAM frameworks. ........................................................................ 24
4.1 The four evaluated edge prototype configurations. ............................................ 28
4.2 Required bitrates for H.264 transmission from the end device. ............................... 32
Mixed reality (MR), where real-life and virtual worlds are merged to produce new worlds, is a field in computer science that has seen a lot of increased interest in recent decades [18]. The increase has, to a large extent, been due to improvements in processing power in mobile devices [17]. Leading mobile technology companies such as Ericsson have presented frameworks, proof of concepts, or fully working products that utilise their core competence in developing mobile solutions[63]. In 2017 both Google and Apple released frameworks to allow developers to take advantage of the full power of their mobile platforms for this use case.

MR is an extension of the term augmented reality (AR), and often interconnected with it, as can be seen in the names of MR frameworks like Google’s ARCore and Apple’s ARkit. While AR centres around augmenting elements onto the real-world environment, for example having labels drawn on top of recognised human faces displayed on a mobile device’s screen, MR centres around displaying virtual content interlined in the real-world scene. The man and the machine exist in the same environment, and it can for, a person unfamiliar with the systems, sometimes be hard to differentiate between the real and the virtual objects. It is thus vital for the computing platform to have a good understanding of the surroundings to be able to render and display the mixed content accurately, which requires extensive processing power [29].

1.1 Motivation

There are limits for what is currently feasible to do when it comes to MR on mobile devices. This centres around the two most significant limitations, processing power and battery life. As of May 2019, Google’s ARCore and Apple’s ARKit only support a limited number of devices in their respective mobile eco-systems. Even on the supported devices calculating and displaying MR content involves a lot of computing power and the processing consumes battery power fairly quickly, a significant obstacle for the broad adoption of mobile MR. It could thus be beneficial to offload some of the processing to a dedicated server. Edge Computing is a distributed computing paradigm that focuses on having the computing resources at the edge of the network, close to the end devices. Thus it is justified to investigate how the edge computing paradigm can benefit the field of MR.

There are several similarities between edge computing and cloud computing, where the computing resources often are in huge data-centres in locations far from the end devices.
Cloud computing excels in certain aspects, including elasticity, where the available computing power can increase or decrease quickly to meet demand. However, there are limitations to the quality of services that can be provided with it. With edge computing, the proximity of the edge devices to the end devices makes it possible also to offload calculations where the results lose value if the latency is too high.

Having such an edge device do some of the calculations for a service could decrease the processing power requirements on mobile devices. It is possible to have on an edge device, processing power far exceeding that on even the fastest mobile device, especially with parallel computing platforms such as Nvidia’s CUDA. Notwithstanding that it is easier to scale a solution by adding more powerful hardware to the edge devices than it is to upgrade the mobile devices [64].

1.2 Aim

The goal of this thesis is to investigate the usage of edge computing for providing MR on mobile devices. This use will have the edge device process the mobile devices’ camera data and create the MR content image. There are many different ways for how to to create and let the user interact with MR. This includes both hardware equipment and how to present the virtual objects. The use-case for MR to be studied in this thesis was defined as follows:

The user holds in their hand a mobile device, with a camera attached to its back. The mobile device displays the image received by the camera. By moving the device around, the user can let the software running in the device get an understanding of the surroundings, with structures and objects. When the software has received enough information to be able to create a three-dimensional map of the surroundings, it draws a visualisation of the map on the device’s display. After the map has been created, the user should be able to press on the screen to insert a three-dimensional virtual object into the device’s representation of the observed scene. When the user triggers that action, the software identifies flat surfaces that can be used to place the virtual object and ensures that the virtual object’s position is not obstructed by other structures, physical or virtual. After an object has been added to the map, it is drawn on the screen with its original observed real-world coordinate and rotation, even when the camera device moves. The virtual object shall keep its observed real-world position and rotation, also when the user has walked away and come back.

1.3 Problem definition

This thesis will try to answer the following two questions:

- Which level of resources will be needed at the edge device to support an MR application? How close can it come to the performance and timeliness of an application for a modern phone with native support for this technology?

- What are the bottlenecks and encountered issues when creating such a solution, and what are the best ways to overcome them?
1.4 Approach

The majority of the thesis work involved the creation of two prototypes for displaying MR content on an Android device, from hereafter called the end device.

1. Local Computing – The end device uses Google ARCore to generate a point cloud map of the surrounding structures. With the point cloud it receives from ARCore, the end device locally draws MR graphics and display them to the user. The graphics consist of the observed surroundings, the point cloud and virtual objects in the form of 3D objects.

2. Edge Computing – The end device offloads the MR processing to an edge server by transmitting over a wireless network its camera video feed. On the edge device, the point cloud of the surroundings is calculated and used to draw MR image graphics. The graphics consist of the observed surroundings, the point cloud and virtual objects in the form of custom 3D objects. An image feed with MR graphics is sent to the end device and displayed to the user.

Most of this thesis focuses on the edge computing prototype. The local computing prototype has optimised native support and none of the bottlenecks involved with server-side computing. It is only used as a reference, for the studying of the targeted, but maybe not reachable, goal for an edge computing prototype.

The work for the implementation of the prototypes included writing software for the mobile end device as well as software for the edge device server. To be able to give a good answer to the question about bottlenecks and encountered issues, as well as evaluate different frameworks and technologies, decisions were made and time was spent on making the system modular with loose coupling between the different parts.

The prototypes, implemented as part of the thesis work, were evaluated using the following criteria:

1. Resource usage. How much bandwidth is required? How much processing resources are needed to achieve the results? The usage includes both that on the edge node as well as on the mobile end device.

2. Responsiveness. How long does it take for the system to act and respond to the user’s interaction?

3. Stability. Does the system performance or functionality degrade over time or due to other factors?

1.5 Structure

The thesis is divided into six chapters, with one appendix.

- Chapter 2 gives a background of what edge computing is as well as an overview of the technologies used during this thesis work.
- Chapter 3 describes the two prototypes and the design process behind them.
- Chapter 4 describes how the prototypes were evaluated and tested and what results those tests gave.
- Chapter 5 discusses the work.
- Chapter 6 contains the conclusion to the questions outlined in this chapter.
2 Background

This chapter covers many aspects of the theory referred to in the subsequent chapters of the thesis. The chapter starts by describing the two platforms being studied for the prototypes. It follows with an overview of edge computing. It then gives a broad overview of video streaming, the fields of computer vision and mixed reality, and the third-party libraries used for these purposes in the prototypes. It ends by giving an overview of research related to this thesis.

2.1 Platforms

This section describes the platforms, consisting of two frameworks interesting for developing the prototypes. For the edge device, the Qt framework is described, and for the end device, the Android SDK with NDK is described.

2.1.1 Qt

Qt [67] is an open-source and cross-platform application development framework, to be used for developing software in C++ and JavaScript. The framework includes a large number of classes and components for software development. Among the parts available include support for networking, graphical widget-based interfaces, multimedia, 3D graphics and concurrent programming. All components are with some minor exceptions available on all Qt’s supported platforms, which include Windows, Linux, MacOS, Android and IOS.

One of the main differences between Qt and other similar frameworks is the way messages and events are passed between classes, instances or threads. Qt uses a concept called Signals-and-slots, which is similar to the Observer design pattern [26]. A Qt slot can be connected and listen to several signals, and a signal can be broadcast to several slots. As all Qt classes inherit from the base Qt class, all Qt instances can be used this way. When integrating the Qt framework, the programmer can also let any of the existing classes inherit from the Qt base class and mark functions as signals or slots, to let them communicate the same way with each other as well as the surrounding Qt platform. For example, the instances of the standard Qt class for text input boxes in graphical user interfaces always send a predefined signal when text has been added. Any slot in another class and thread may subscribe to that signal. The signals and slots are connected dynamically by the programmer, who does not need to modify any of the two classes involved to adapt them to the new coupling.
This mechanism is part of the Meta-Object System, which in addition to signals and slots also manages other runtime information for the system. The C++ source code to support it is automatically generated during the compilation phase by a preprocessor tool. Having the Meta-Object System managing the communication with signals and slots also simplifies concurrent programming as processing can be moved to executable threads without the need of writing a separate event loop, with the executable threads waking up from sleep to act on signals before going back to sleep. Any instance of a class that is inherited from the standard Qt base class can be moved to a thread, and will after that have all signal and slot connections act asynchronously, meaning the processing for signals it receives will run in its separate thread. The default behaviour for signals to instances in the same thread is synchronous calls, executed immediately. The programmer can also set specific inter-thread connections to be synchronous, leading to the slot function being executed in the signal emitting instance’s thread. The mechanism thus allows the programmer to connect instances and threads without having to worry about thread safety or having to update existing class implementations.

2.1.2 Android SDK and NDK

The operating system Android [16] is, to a large extent, developed by Google. The company also owns the trademark and decides what is put into the platform. Google provides a software development kit, referred to as Android API, with classes that third-party applications can use to interact with the operating system. The SDK has official support for the programming languages Java and Kotlin and is built on top of the languages’ standard libraries. The Android devices are backwards compatible such that modern devices can run apps developed for early versions of Android as long as they were written to use the official API. While a lot of new functionality and frameworks have been added to the operating system with every new Android release, few phone vendors provide upgrades for phones that are older than two years old. This has led Google to backport some later released technologies and APIs in the form of support libraries, making them available on older versions of Android. There are, however, many limits in this support. The list of APIs that have not been fully backported includes the camera API Camera2, with more functionality than the original camera API Camera, and the MediaCodec API, which can be used to access hardware-supported media encoders and decoders. Both of these APIs are used for the edge computing prototype.

Google provides the Android Native Development Kit (NDK) to enable the use of native code in Android applications. Native code, in this case, means code that has been compiled directly to the hardware platform. This code is often written in C or C++. NDK builds upon the Java Native Interface (JNI) standard which enables communication between Java functions running native code, with the extension that native code now can use the same APIs as of Java apps and call Android OS code. It also has support for converting object types between the two. With the NDK an application can have the main module written in Java and have the computation-intensive graphics processing parts written in low-level C++.

2.2 Edge computing

Cloud computing is a technology that, for more than two decades, has changed the landscape of data storage and manipulation [8]. The cornerstone of cloud computing is that the server is in an unknown place relative to the user and many times also to the service provider. The physical locations of the servers are in many cases, far away from the end device.

New servers for a service can with cloud computing be put online within minutes with a few mouse clicks when user activity demands it, without any noticeable effects for the user. The service provider does not have to estimate how popular a new service will be and how many servers need to be installed before launching it. This changing of capacity is called elasticity, with the ability to dynamically increase or decrease computing power to meet demand. With cloud computing, the capacity can appear infinite [39].
With the elasticity also comes some side-effects. The major one is the latency in communications between the end device and the cloud server. While many cloud providers give their customers the option to decide in which regions their servers shall be, these regions can span large distances. Users from one country might be directed to a data centre on a different continent [55] even though the big cloud service providers have multiple data centres in a neighbouring country. With the cloud service providers’ server software enabling live migration, the virtual machines running a user’s sessions can move from one data centre to another without the service provider nor the end-user noticing it [45] [10]. This makes cloud computing less than ideal for some use cases, including real-time video analysis, computer gaming and driver assistance systems in automobiles [51] [44].

Edge computing [54] tries to bridge that gap, by moving the service provider’s servers closer to the end-user at the edge of the big network that is the Internet. Instead of being in a big data centre, the infrastructure node the clients connects to might be as close as the local cellular network base station or in the apartment building’s basement. Having the service provider’s computer, called the edge device, at a closer distance from the end user’s end device brings with it several benefits. One of these is more predictable and reliable communication between the two parties, including lower latency. With fewer infrastructure nodes involved in the traffic, the bandwidth may also be higher and the packet delivery less prone to jitter. Edge computing enables more data processing that today runs on the user’s device to be moved to the edge device. Intel claims that mobile edge computing and its low latency is one of the core features for 5G mobile networks [31]. Edge computing is not meant to replace the cloud but to complement it. Today many networks already have proxy servers to cache various static content. With edge computing, these are extended to have the ability to do more advanced computing tasks, which can include real-time video, voice recognition or automotive navigation. By having the computation done on an edge node instead of the end device, the hardware and software in the latter can be simpler.

2.3 Computer video encoding

Video is today one of the core drivers of Internet usage [3]. The popularity of watching video has led to the invention of many different technologies for compressing and transmitting moving images, and there are many different standards in extensive use today [23]. This section gives an overview of video compression and describes the video encoding standards VP8, MJPEG and H.264. All three compression standards were considered during this thesis work. They operate independently of each other, and can all be used to compress regular video.

2.3.1 Video compression

With only minor loss in image quality, it is now possible to use a modern video compression standard such as H.264 to compress down a raw data video with the resolution 1280x720 pixels, from 22 Megabit to 166 Kilobit per frame. In the context of video compression, the tasks of compressing and decompressing a video are often called encoding and decoding. Key elements of video compression are keyframes and P-frames. A keyframe contains all the information needed to reconstruct an image, while the P-frames only include the changes from the previous image frames. Many video compression standards try to minimise the use of keyframes, to increase the video compression rate [23].
2.3.2 MJPEG

Motion JPEG (MJPEG) is one of the most uncomplicated forms of video streaming that also compresses the video to a smaller byte size. Every image frame is compressed independently of each other using standard JPEG lossy compression, and the resulting files are transmitted as a long stream. The limitation with MJPEG is that no information is shared between different frames, everything is thrown away when the frame is ready, and each frame acts as a new keyframe. This limitation can, at the same time, be beneficial when there are interruptions or data corruption in the video transmission. If the content for an image frame is invalid or not received, the player only has to wait for the next valid frame to resume playback. Transmission can also be simplified when the transmitter can push out images as soon as they arrive.

2.3.3 H.264

MPEG-4 Part 10, Advanced Video Coding (AVC) or H.264 as it is more commonly known, is as of 2019 one of the most frequently used formats for compressing video [50]. H.264 uses several advanced mechanisms to compress image data and has a high compression ratio. As the power consumption needed both for encoding and decoding H.264 content using software processing is high, many devices that support video playback now have hardware-level support for it. This includes Android devices that are to get Google’s approval and be able to run the company’s applications, for which hardware-level support for H.264 encoding and decoding is listed as requirements.

2.3.4 VP8

VP8 [59] is a format put forward by Google, to replace the patent-bound format H.264 as the format of choice for streaming on the web including at Google’s YouTube. It shares many similarities to H.264, with the major difference being the lack of license fees. As with H.264, Android devices that are to run Google’s software need to have hardware-level support for VP8 encoding and decoding.

2.4 Video streaming

When a video has been encoded, there are several methods to transmit it to the receiving device. This thesis will only involve streaming media, where the video can be played while being received. The network protocols TCP and UDP were considered for the video transmission communication part in this thesis. When transmitting H.264 over TCP the protocol MPEG-TS can be used, and when transmitting H.264 over UDP, the protocol RTP can be used. This section describes these different technologies, as well as gives a background in network technology by giving an overview of the OSI model. This section also describes the GStreamer multimedia framework, which facilitates the application of both video compression, decompression, transmission and retrieval.

2.4.1 The OSI model

The International Organization for Standardization’s Open Systems Interconnection (OSI) model is often used to differentiate and group different parts of the network stack [19]. The standard groups the different network protocols into seven different abstraction layers, seen in Table 2.1. It starts at the top layer with the application-specific protocols on one device and then goes down to the lowest level with the physical connection such as cables or radio transmissions before going back up again on the other connected device until it reaches the application layer. Protocols that follow the OSI model are loosely coupled and written for
2.4. Video streaming

| 1. | Physical layer |
| 2. | Data link layer |
| 3. | Network layer |
| 4. | Transport layer |
| 5. | Session layer |
| 6. | Presentation layer |
| 7. | Application layer |

Table 2.1: The seven layers of the OSI model.

interchangeability. An application layer protocol describing communication for a specific service should work independently of the lower layer protocols. The work for the edge computing prototype in this thesis focuses on the application layer and the transport layer.

2.4.2 Transmission Control Protocol

TCP[47] is a transport layer protocol in the OSI model, and the most common such protocol used on the Internet. The server and the connecting client starts by agreeing to create a session for the transmission before any data payload is sent between them. Both parties use that session for sending and receiving data, and it is kept open until any party decides to close it. All sent data packets have a unique id number, and content verification checksums are included for all data. For every packet received correctly, verified by comparing the checksums, the receiver sends an acknowledgement back to the transmitter. The software implementations on both communication devices ensure that corrupt or lost packets are resent and that the correctly received packets are assembled in the correct order based on their id number. As long as the communication pipe is kept open between the two devices, one can be sure that all packets will eventually arrive. Thus it is advantageous for use in cases where it is important that the data arrives intact and the data has value also if it arrives a bit late. It is used for data communications on the World Wide Web, via the Application layer HTTP protocol, and as such in wide use in mobile devices.

2.4.3 User Datagram Protocol

UDP is a transport layer protocol in the OSI model. With UDP, the sender does not keep any trace of the data packets after they have been sent. The communicating parties do not need to communicate before a packet is sent and there is no built-in method for ensuring that packets are received correctly, in the correct order or that the receiving party can process them. There are optional verification checksums that most platforms today implement but in general the corrupted packets are just dropped. The receiver has to use a higher layer protocol to signal that back to the sender [58]. The lack of overhead is the main benefit of UDP compared to TCP. Glitches in the network communication do not mean that old data will get stuck in the network pipeline before it can be delivered. It also supports broadcasting, where the same packets are sent to multiple devices. UDP is thus suitable when the user wants to send a lot of data fast, and it is not necessary to re-transmit it because old data has little value. Examples of such use cases are live video feeds and action video games [48].
2.4.4 Real-time Transport Protocol

RTP is an application layer protocol in the OSI model, to facilitate real-time streaming of media content such as audio and video. The RTP packets can include application-specific stream packet data such as timestamps, packet numbers and stream metadata. RTP is often used with the RTP Control Protocol (RTCP), with which the receiver can send back to the transmitting device reports about stream jitter, out-of-order delivery of packets as well as packet loss. The server can then use this information to decide if it should resend content or change the streaming format, including the bitrate. In most cases, RTP streams are sent over UDP, with RTCP sent over TCP [57].

2.4.5 MPEG Transport Stream

MPEG Transport Stream is not defined as a protocol in the OSI model but as part of the MPEG-2 standard, to be used for wrapping different types of streamable media content. As part of the MPEG-2 standard, it is also a standard container for digital television streams, where it is commonly used to wrap H.264. It is designed to be used for one-way broadcast transmission over unreliable connections, such as radio transmissions, but can also be used when transmitting content over the Internet, over UDP and TCP. Being designed for digital television, the standard consists of several different methods to include streams and metadata, but as those are optional, the container format can also be lightweight and only include the most necessary packet data [34].

2.4.6 GStreamer

GStreamer [28] is an open-source framework used for processing video and audio, including encoding and decoding, and with official support for multiple platforms including Linux and Android. The elements in the GStreamer ecosystem follow defined interfaces and can be put together to form a pipeline, by creating plugin elements and connecting them manually or by specifying their relationship via text to have GStreamer create the instances. Elements can do tasks such as decoding, encoding, transforming and displaying content. This makes it easy for a developer to define an application-specific pipeline by connecting several different plugins as well as later with some adjustments also, for example, replace one encoder with another. GStreamer also has good support for configuring all codecs and streaming elements with interfaces they provide, including local hardware accelerated codecs where available, but as GStreamer only acts as an interface to the elements, it is not able to describe how to connect them and what settings are appropriate beyond regular debug information. The organisation behind the framework develops and provides, both as open source and bundled with the framework, several such elements and codecs. By being plugin-based, the framework can also be extended by external parties to add support for different media formats and hardware, leading to it being a cornerstone in many devices that have video capabilities.

One example pipeline can be seen in Figure 2.1. This pipeline, to be run on Android, retrieves the image from the device’s camera and sends it to three different sub-pipelines. The first displays the image on the screen, the second one sends the retrieved pixel data to be processed by callback functions, and the third compress the video using H.264 and sends it via RTP and UDP to a receiving device. With this example, all the elements, except the UDP transmitter, are run with their default configurations.

2.5 Computer vision

Computer vision deals with how computer algorithms can be used to analyse and gain information from images. With today’s computers’ high processing power, it is now possible to process real-time video streams and accomplish tasks such as automatic road lane track-
2.5. Computer vision

```bash
ahcsrc !
tee name=t t. ! videoconvert ! autovideosink t. !
tee name=p p. ! appsink p. !
videoconvert ! x264enc ! rtph264pay !
udpsink host=192.168.0.3 port=39200
```

Figure 2.1: GStreamer pipeline

ing, face recognition and three-dimensional representations of the observed structures. This increase in services has been most noticeable in augmented reality, surveillance and security systems for cars.

This section describes the theory involved, and also the two computer vision libraries that are used in this thesis for creating mixed reality. The edge computing prototype uses ORB-SLAM2, and the local computing prototype uses Google ARCore.

2.5.1 Mixed reality

Mixed reality (MR) [18] [29] is in the broad sense the mixing of the physical reality with virtual reality. Where virtual reality means stepping into a completely virtual environment, for example by putting on virtual reality glasses where one’s impressions come entirely from the computer’s creation, with MR the virtual world has a connection to the real world and never leaves it [68]. The term is often interchangeable with augmented reality (AR), which has been a focus area in several sciences for half a century [12]. Examples of implementations of AR that are in wide use today are mobile applications that allow the user to point the device’s camera to the sky and have the device display information about the visible aeroplanes [2], and software in cars that use infrared cameras during night-time to detect pedestrians and animals and use markers on the dashboards to highlight their locations for the driver [1].

With MR, the virtual elements are not just to highlight aspects. Instead, they are part of the scene, and to the observer can look the same as the portrayed three-dimensional real-world objects. The objects can also act in a real-life way by interacting with the surrounding real-world objects and observing the same rules in terms of physical borders and gravity. By doing this, the perceived differences between the virtual objects and the real-world objects can be minor, and it can be difficult for the observer to recognise which is which.

2.5.2 SLAM

Simultaneous localisation and mapping (SLAM) [24] deals with the task of creating and updating a map of the sensor device’s surroundings and tracking the sensor’s position relative to this map when it moves around. The resulting three-dimensional map is often modelled as a three-dimensional point cloud. SLAM is often a cornerstone of mixed reality, as to be able to insert content into an environment one has to both create a map of the surroundings to see where these new attributes should be placed, as well as localise oneself within this map [37].

Many different input sensors can be used for SLAM, including depth detecting cameras, stereo cameras and single passive cameras such as the ones found in mobile phones. In this thesis, we will only cover keyframe-based monocular SLAM, where the system’s input is taken from individual images retrieved from one camera with no depth-sensing capabilities.

Keyframe-based monocular SLAM shares many core aspects with the technique Structure from Motion (SfM) [66]. With SfM, two or more 2D images from one or several cameras under movement can be used to calculate a 3D representation of the observed environment and its structures. The foundation of SfM is identifying image features and points that are shared between two or more images. With the differences in 2D positions between images, algorithms such as bundle adjustment can be used to get the extra dimension. Identifying and keeping track of all these feature points is a significant task in SfM. With many images in the set, taken
from many different parts of the scene, this task can be challenging to solve in real-time. Many
SLAM algorithms use different techniques to speed up this processing, including clustering the
gathered data into scenes.

SLAM has seen a lot of interest mainly for use in autonomous vehicles and other robots,
where it is used to guide the systems in both known and unknown environments. In recent
years it has also seen increasing use within mobile phones and AR applications.

The algorithms for creating the three-dimensional point clouds can either be considered
dense, semi-dense or sparse, regarding how much input data they use and how much they
interpolate from the observed points to generate the output. In the case of monocular SLAM
with regular camera sensors, sparse mapping can include significant points like corners. Semi-
dense mapping also includes whole edges, and dense mapping’s goal is to include all points
in the observed structures’ surfaces. As the computing power needed to create a point cloud
from sensor data increases with point cloud density, most SLAM frameworks used for AR use
sparse mapping, including Google ARCore.

2.5.3 ORB-SLAM2
ORB-SLAM2 \cite{OrbSLAM2} is an open-source real-time SLAM library developed at Spain’s Uni-
versidad de Zaragoza. While it has not been actively developed since 2017, there is a large
community around it that has documented its limitations and provides workarounds. It in-
cludes support for monocular, stereo and depth detecting cameras and is written to be run
on Linux. It has since the release been the focus for several projects whose goal have been to
extend it with support for new computer vision technologies and computing platforms. Stan-
dard ORB-SLAM2 does not support offloading matrix calculation to external hardware, and
everything is processed by the main CPU. It uses OpenCV extensively for image filtering and
analysis. OpenCV \cite{OpenCV} is an open-source library with many algorithms and functions related
to computer vision \cite{OpenCV}.

In this thesis, only ORB-SLAM2 in monocular mode was evaluated. In this mode, ORB-
SLAM2 as input uses regular camera images whose significant feature points are detected and
described as vector representations. For this extraction, it uses the OpenCV’s ORB (Oriented
FAST and Rotated BRIEF detector), which scans the image for significant feature points,
often those with big changes in contrast such as corners on objects. It converts each point to
a multi-vector descriptor. These feature descriptors are not directly mapped to the original
image content, and the feature descriptor for a marked point shall be the same even if the
image is rotated, scaled or tilted.

These calculated feature descriptors, with their positions in the image, are fed into a bundle
adjustment algorithm to calculate and update the local point cloud and get an estimation of
the camera’s movement, position and direction. Detected sets of feature descriptors close to
each other are stored as keyframes to be used for coming comparisons and calculations.

The localisation part in a SLAM system needs to take care of three situations, listed below:

1. Tracking, where the camera’s position and rotation are known, and the camera is moving
   relative to those.
2. Global localisation, where the point cloud has been created, but the system does not
   know the camera’s position and rotation in it.
3. Re-localization, where the assumed position and rotation of the camera in the point
   cloud were found to be incorrect, and the correct ones need to be determined.

To be able to manage all three situations, the feature descriptors have to be also kept
for keyframes outside the local environment. Running ORB-SLAM2 over a large scene can
lead to a large set of stored multi-vector descriptors and a lot of computation needed when
finding matches. For faster lookup ORB-SLAM2 groups the keyframes into a large scene
graph, and stores the multi-vector feature descriptors using Bag of Words (BoW) [25]. BoW is a data representation that converts the descriptors into natural language and makes it easy to compare them to find matches. The conversion is done using a large vocabulary set model that has been taken from machine learning classifiers running over an extensive set of images. The BoW set included with ORB-SLAM2 is around 145 Mb and needs to be loaded before the system can start. Thus when new feature descriptors are detected, the system can fairly fast bring up the old keyframes and find the correct position. When the system has detected that it has returned to a location it has already been in but identifies that the relative pose was wrong, indicating that the system had under- or over-estimated how fast the camera had moved in the previous segments, it adjusts the overall point cloud to eliminate this drift.

The software architecture of ORB-SLAM2 consists of three perpetually running threads, in addition to an optional graphics drawing thread for displaying the collected data including point clouds and possible augmented graphics. These threads are listed below.

1. Tracking. This thread is fed images from a camera or video. For each image frame, it detects the image features via ORB and uses the extracted image feature descriptors to calculate where the camera is relative to the previously detected position. When it determines that enough unique features are present in an image, it saves the group of found image descriptors as a keyframe.

2. Local Mapping. This thread is fed new keyframes from the tracking thread when they are ready and uses them to calculate a map for the local environment. This map is calculated using bundle adjustment, run over the set of current keyframes. This thread is also responsible for deciding which keyframes from the local set should be stored in the system graph.

3. Loop Closing. This thread is fed data from the local mapping module and uses it to detect loops. Loops are when the device has returned to a previously known position within a local set. When a loop has been found, a new temporary thread is created to run over the whole set of collected keyframes and use the new information to adjust it for drift. Without this step, it is possible that incorrectly calculated distances can make the system think that the position it just returned to is in a completely new position.

ORB-SLAM2’s overall design makes it able to do real-time SLAM on a slower personal computer and when the system is fed input of low quality. Due to the SLAM algorithms only using the image feature descriptors taken from OpenCV’s ORB, only the high contrast features found by ORB can later exist in the point cloud. The calculated point cloud is sparse. ORB-SLAM2, like all other pure monocular SLAM systems [20], does not do any estimation of the physical distance between the points in the point cloud so to add this functionality a reference object with known dimensions needs to be added to the scene. As it internally uses OpenCV’s camera model based on camera matrices, it can compensate for many different cameras and lenses. It has even been shown to work with extremely wide-angle fisheye cameras [30] as long as the camera model is adjusted.

2.5.4 ARCore

Google ARCore [43] is an Android framework for developing AR or MR applications, first released in 2017. The framework is based on the work done for the framework Tango, which Google released the first version of in 2014. Where Tango only supported a few devices and came packed with the device by the vendor, Google ARCore is provided as an external Android package and not included in the operating system. To be able to run the framework a device must run Android 7.0 or higher and have passed Google’s certification process.

---

1 Google Developers: ARCore - Supported Devices https://developers.google.com/ar/discover/supported-devices
last requirement leads to the current situation where many Android devices lack support for ARCore, despite many of them running Android 7.0 or higher.

The framework internally manages all created point clouds and other data for AR, and only high-level APIs are available to third-party developers. The point clouds that are provided by these APIs are sparse. ARCore relies on camera images as well as the device’s Inertial Measurement Unit (IMU) \(^{[27]}\) to create the point clouds and use image-feature detection to recognise reference points. Being image-feature based, ARCore also suffers from the same issues as other similar SLAM libraries in that it struggles to create point clouds when pointed towards flat surfaces with few significant features. The resolution provided for the camera image depends on the device and is according to Google on most devices 1920x1080 pixels\(^{2}\). The camera update speed is as of version 1.9 released in the first half of 2019 set to 30 Hertz\(^{3}\), also on phones that have cameras with higher framerates.

### 2.6 Related works

Indirectly many studies of mobile augmented and mixed reality over the past decades have included research into offloading large parts of the calculations to servers \(^{[12]}\). The need for this has been a consequence of mobile devices for a long time having little computing power \(^{[29]}\). These cases for offloading include the first practical application of AR using mobile devices, the Touring Machine presented in 1997, which overlaid virtual 3D objects on camera image feeds \(^{[21]}\). Today with the increased performance in mobile devices, related research can be seen for AR using small wearable devices such as smartwatches, which offload processing to phones \(^{[61]}\). One of the earliest studies for doing mobile MR was done in 2004 by Skrypnyk and Lowe \(^{[62]}\), who used a local server to do the calculations related to scene modelling and insertion of virtual objects. However, the studies have rarely touched upon the aspect of edge computing, and the questions about transmission and processing between the devices that this thesis tries to answer.

There have been studies into cloud-focused AR toolkits, such as CloudRidAR \(^{[33]}\). Where the CloudRidAR paper describes the problem from a high level and uses offloaded image processing to position elements on the screen, with graphics rendered on the end device, this thesis focuses on more advanced MR, and graphics rendered on the edge device.

Much research has been done in the field of video streaming, including in relation to edge computing \(^{[11]}\). An aspect of video streaming that MR using edge computing shares some aspects with is cloud-based gaming \(^{[60]}\). With cloud-based gaming, the fast-moving 3D graphics are created on a cloud server and sent with low latency to the end user’s device using compressed video. Examples of this include the platform GamingAnywhere, which is a working implementation that can be tested \(^{[32]}\). In the case of cloud-gaming, the primary data traffic is one-way, to the thin client — a difference to the scope of this thesis, where the communication involves both devices simultaneously transmitting video graphics.

While many organisations and companies including the European Telecommunications Standards Institute \(^{[31]}\) and Ericsson \(^{[63]}\) say that AR and MR is part of the group of services that will benefit the most from the power of Edge Computing, not much research has focused on MR use cases specifically. With the broad scope of AR and MR, a large part of the research has focused on AR and less computing-intensive technologies like tracking of faces or objects \(^{[17]}\). No research has been found that tries to compare the performance of a modern native mobile framework like ARCore with the one running on an edge device.

---


\(^{3}\) Google ARCore Issue Tracker: Textures and device pose at 60Hz [https://github.com/google-ar/arcore-android-sdk/issues/604](https://github.com/google-ar/arcore-android-sdk/issues/604)
This chapter describes the two prototypes created as part of this thesis, one with local computing and one with edge computing. The chapter starts with how the prototypes were created, and describes the local prototype followed by the edge prototype. The edge prototype section describes its two parts, an Android application to be run on an end device, and Linux software to be run on an edge device. The later parts of the chapter have been divided into three aspects of the edge prototype. These aspects are network communication, video streaming and MR.

The method used for this thesis was the development and empirical study of two prototypes in the field of mixed reality. It followed the approach described in Section 1.4. First, the local prototype was developed, to be used as a reference. After this, the development of the edge prototype was done with an incremental method. All the source code written for this thesis can be downloaded from Linköping University’s web site\(^1\),\(^2\).

The edge prototype’s necessary tasks were listed and drawn as an overall system design. The overall system design was further detailed into loosely coupled components, as detailed in Section 3.2.2 and Section 3.2.3. The guiding principle for the design was modularity. The components could be studied and implemented independently using the defined interfaces. For some of the interfaces, different components with different technologies were implemented for evaluation. This evaluation can be elaborated in Chapter 4.

### 3.1 Local prototype

The local prototype implements the use case described in Section 1.2. It uses Google ARCore 1.8 and was developed to be run on an LG G6 running Android 8.0. The majority of the source code for the prototypes was taken from the tutorials and reference material Google provides, with changes only made to allow the framework’s behaviour and performance to be studied. These changes are described in Chapter 4. The MR content consists of a three-dimensional Android mascot provided by Google and is placed on flat surfaces calculated from the point cloud. The graphics are rendered with hardware acceleration using OpenGL ES 2.0. An image of the local prototype displaying mixed reality can be seen in figure Figure 3.1.

\(^1\)https://gitlab.liu.se/ida-rtslab/public-code/2019_mrleo_server
\(^2\)https://gitlab.liu.se/ida-rtslab/public-code/2019_mrleo_client
3.2 Edge prototype

This section includes an overview of the prototype, with all the components and how they interact with each other, as well as detailing the code components developed for the edge device and the end device.

3.2.1 Overview

The development and testing of the edge prototype were done with one Linux PC acting as the edge device and one Android mobile phone acting as the end device. Figure 3.2 shows an overview of the edge prototype. It shows the main components, as well as how data is sent between them.

For this thesis, the method for offloading mixed reality processing using edge computing have the end device and the edge device both produce and consume graphics. Thereby, a large part of the prototype involves the transferring of video. From the end device, the camera images are converted into a video stream and sent to the edge device, where the received images are processed. The generated mixed reality graphics are converted to a video stream sent to the end device, where they are displayed to the user. The end device software does not do any analysis of the content of the images it receives from the camera; it only encodes it as video and sends it to the edge device.

Both devices have an overall manager instance, which has the overall responsibility for configuring the different components and handles the communication with the other device. This design choice facilitated the testing, removing the need to change the source code when changing the test configurations. The two devices’ managers communicate with each other using the Network Data Link, described in Section 3.3. The data link can also be used to transfer action commands, status reports, or image feeds, but all of those are received by the device’s manager. Before any video is streamed, or data is processed, the two device managers confirm with each other what configuration to use for the session. The other components in the two devices are loosely coupled, meaning that for example the video receiver and streamers in a device do not communicate with each other.

The Mixed reality component is responsible for creating the MR graphics, and package them as images to be sent to the end device. It creates a point cloud mapping of the observed environment and determines the end device’s position and rotation in it, called its pose. The
three-dimensional point cloud is updated and extended when more images arrive, and the pose is updated. When a point cloud has been created, and the end device’s pose in it is known, a three-dimensional graphics objects can be added to it. When to add an object is decided by the user, with a command sent to the edge device, but the calculation of where to put it in the currently visible scene is done by the edge device. The graphics object can be removed by the user but is otherwise kept for the remainder of the session.

The part of the Mixed reality component responsible for drawing the graphics, further described in Section 3.5.3, gets sent the last received camera image, and, if available, the end device’s pose and the graphics object with the point cloud and added objects. The graphics drawer creates mixed reality by drawing the graphics object with the point cloud on top of the camera image. Using the calculated pose, the graphics object is drawn in their correct places in the image in reference to the observed real world. No graphics are added if the end device’s pose cannot be calculated or the objects are outside the visible scene. This includes both the point cloud and the added objects. The camera image is always included in the MR graphics and sent from the Mixed reality component, even if no virtual objects were drawn. The Mixed reality component is further outlined in Section 3.5.

When the end device disconnects from the edge device, all the resources allocated for the session is freed. No data is stored permanently by the edge device or is shared between connected end devices or sessions. The used configuration is stored in the end device software and to be displayed as the predefined configuration for the next session, but no other data is saved.

An image displaying the edge prototype running on the end device, with virtual objects visible, can be seen in Figure 3.3. The following two sections describe in further detail the designs of the edge and end device’s software.

### 3.2.2 Edge device component of the edge prototype

The development of the edge device software was done in C++ with the use of the Qt framework, version 5.9. The rationale behind this choice was extensive support in the platform for integrating different third-party open-source libraries. Personal experience with the Qt framework also removed the learning phase. All the code was compiled with GCC 7.4, which was the default compiler in Ubuntu 18.04, the operating system available during the development.
Figure 3.3: Image from the end device running the edge prototype, displaying mixed reality content

Figure 3.4 describes the components on the edge device, as well as how they send data to each other. This is a more detailed view of the right-hand side of the overview in Figure 3.2.

![Diagram of the Edge Prototype on the Edge Device]

Figure 3.4: The components on the edge device and how they interact.

Each of the components included in Figure 3.4 is initialised as a new instance when a client connects, and runs in a separate processor thread. The components are all loosely coupled, and the Manager’s task is to connect them, using the configuration it receives from the end device. The network components are outlined in Section 3.3, the video components are outlined in Section 3.4 and the Mixed reality interface, Graphics and SLAM components are outlined in Section 3.5.

With concurrent programming, where many processing threads are working on the same task, comes several issues that need to be handled by the system. These include ensuring...
that multiple threads are not writing and reading to the same memory at the same time, without having to make unnecessary data copying. It also includes data management, to ensure that allocated resources such as memory are released when no longer needed. To solve these issues, the edge prototype uses several of the mechanisms in the Qt framework. The software uses Qt’s signals and slots as its mechanism both for communication within and between threads and class instances. To improve performance, no larger data structures such as images are copied when sent between components but instead passed using Qt’s thread-safe smart pointers. These pointers allow references to data to be moved freely between threads, ensure that nobody removes the data while someone else wants to read it, and also that the allocated memory is freed when the last using component no longer uses it. Qt’s signal and slots, in default mode, use a data queue when sending data between threads. This configuration increases latency if frames arrive faster than can be processed. To bypass this queue and ensure that threads and objects can skip older frames in the queue, and always work on the latest image frame, some mechanisms had to be implemented on top of the Qt signal and slots. These mechanisms are described in Appendix A.

The system was written to be able to be started via the system console and run without a graphical user interface. It is configured via parameters sent to it via the system console, and it is possible to enable a graphical user interface as well as access cameras connected directly to the edge device. For further details about this part, and see Appendix A.

3.2.3 End device component of the edge prototype

The end device software to be run on Android has parts written both in Java and C and uses the Android API as well as Android NDK. The version used for building and other support functions was Android API 28, the latest available release. The application requires that the device runs Android 6.0 or later, due to the need for support for specific API interfaces to interact with the device’s hardware, including the camera and media encoding and decoding functionality. The native code written in C was necessary to integrate the GStreamer video framework, which only is provided as a native library and has no API functions exposed directly via Java. Compatibility issues between GStreamer and later versions of Android NDK lead to the use of Android NDK release 18.

![The Edge Prototype on the End Device](image)

Figure 3.5: The components on the end device, arrows denoting interaction between the components.
3.3. Network communication for the edge prototype

The overall architecture can be seen in Figure 3.5. When the users start the application, the first part they see is the Start Menu. This screen is seen in Figure 3.6. In the start menu, the users set the configuration to be used, including if a video is to be used as input instead of the device’s camera. It is also possible to start the ARCore-based local prototype from this menu.

![Mixed Reality On the Edge](image)

<table>
<thead>
<tr>
<th>Mixed Reality On the Edge</th>
<th>Edge node IP address: 192.168.0.3 Edge node port number: 39200</th>
</tr>
</thead>
<tbody>
<tr>
<td>The end device’s method for image transmission.</td>
<td><strong>H.264 over UDP (with SW encoder)</strong></td>
</tr>
<tr>
<td>The edge device’s method for image transmission.</td>
<td><strong>MJPEG over TCP</strong></td>
</tr>
<tr>
<td>End device’s bitrate (Kbit/s).</td>
<td>4000</td>
</tr>
<tr>
<td>Edge device’s bitrate (Kbit/s).</td>
<td>4000</td>
</tr>
</tbody>
</table>

Figure 3.6: The end device’s application’s start menu.

The chosen configuration is sent from the start menu to the app’s manager instance. As the name suggests, the manager manages all components in the system, as well as all the communication with the edge device, which is done via the network data link. It is also responsible for managing the user interactions it receives from the graphical user interface. Based on the configuration it received from the start menu, and having received additional configuration from the edge device, it sets up and configures one of the two H.264 video streamers and, depending on the video streaming format used by the edge device, configures either an H.264 or MJPEG receiver. Video on the end device is further detailed in Section 3.4.

The other components only communicate with the manager, not with each other. The video receiver only decodes video frames and displays them in the user interface area it got from the manager instance; it does not communicate with the user interface.

The graphical user interface created by the manager is seen in Figure 3.3. It has buttons for informing the edge device when to add or remove virtual objects, a menu for changing the camera configuration, as well as two video graphics displays, one smaller showing the image feed received from the camera or the selected video, and one bigger showing the video feed received from the edge device.

3.3 Network communication for the edge prototype

The edge prototype needs to include a network component to allow communication between the end devices and edge devices. The communication component supports both the transferring of commands, configurations as well as larger files in both directions. The component thus involves both devices, with implementations on both.

The Qt framework and Android API provide software interfaces to allow devices to communicate with each other using TCP and UDP, called sockets. In the OSI model, this is the transport layer. On top of that, the prototype needs to have an application layer, to turn the streams of bytes into meaningful data. This layer needs to have both a well-defined protocol.
and, on both the sending and receiving devices, software implementations that correspond to
the protocol. For the sending device, this includes how to format the data to be sent, and for
the receiver how to divide the received data streams into segments for further processing by
the system.

Internet sites for open-source software, including GitHub, were surveyed to try to find
existing open-source and cross-platform libraries that could be used for this purpose. Key
aspects looked for were the ability to provide more features than the sockets already available
in the platforms, and both the support for low-latency communication, which was needed for
the user interaction, as well as support for sending configurations and files. No Java and C++
open-source library was found that satisfied these requirements. Instead, a complete network
component was designed for the edge prototype, with a custom application layer protocol. It
was implemented for both devices, built on top of the TCP and UDP sockets.

Figure 3.7 shows typical communication between the devices running the edge prototype.
Only MJPEG images are sent using this network component, and sent as pure JPEG files.
Other types of video streams are sent using the video components described in Section 3.4.

![Figure 3.7: Example of communication in the Edge Prototype.](image)

The network protocol is very lightweight and easy to interpret, an effect of the limited
number of types of files and messages that needed to be sent between the two devices in the
edge prototype. Internally in both devices’ software, all segments that are to be sent to the
communication layer are treated conceptionally as independent files. All communication is
identified by session-id numbers, to allow both TCP and UDP to be used for any content
within the same session. By default, only TCP is used for commands and configuration while
both UDP and TCP can be used for transferring larger files such as images. When using UDP
the file’s data is divided into new UDP datagram packets with the user’s defined maximum
packet size, which may be higher or lower than the maximum packet size set by the devices or
the infrastructures nodes involved.

The commands and configuration data structures are sent as JSON, a text-based format
that is good for transmitting key-value pairs and has API support in both Qt and Android.
The file data structures used for image data may also contain additional metadata attributes.

One of the limitations with TCP is that if the available bandwidth drops, the sender might
continue to send data at the previous speed, not knowing that it is not being received. This
sending can build up a queue of data that, when it is later received and processed by the receiver, already is too old. With ordinary sockets, the implementation can support limiting the sending bitrate but has no feedback for how the underlying platform network socket process the packets or if there’s a queue in the hardware’s underlying transmission layer. A mechanism was therefore added to the edge device’s software to overcome this limitation, by allowing it to recognise if the receiving end device has received the segments sent over TCP. The mechanism builds upon the low-level support in Qt for the platform’s network sockets, where it receives information about how many bytes have been transmitted. In the edge prototype, this is used in combination with the buffering mechanism described in Section 3.2.2, to ensure that only the latest images are sent to the receiving device. Nothing similar was found in the Android API, and a corresponding functionality for transmissions from the end device could not be implemented.

3.4 Video streaming for the edge prototype

This section describes how video streaming was implemented for the edge prototype. Both devices needed to both encode and stream video as well as decode the received streams. Low-latency streaming was a goal for both parties, to ensure responsiveness in the edge prototype. A goal with the streaming was to make use of both device’s hardware-accelerated encoders and decoders, which are included by the device vendor to improve overall device performance and energy consumption. This section starts with outlining the video formats used, then describes how video support was implemented on the two devices.

3.4.1 Video formats

Three different video streaming formats were selected to be investigated for this thesis. These three are listed below, with the reason for why they were chosen to be investigated.

- VP8 and H.264 were included since they are in the list of media formats Android devices have to include encoders for to be allowed to run Google’s software [7].
- MJPEG was included on the list due to being deemed easy to implement on top of the existing communications protocol.

VP8 was found to have low support outside of the task of streaming to web browser clients. No open-source projects could be found that used the Android devices’ VP8 hardware-accelerated encoder, besides some simple proof of concept that had not been updated for several years. With the format found to be sharing many similarities with H.264[59], the format was not investigated further.

H.264 was found to have broad support in third-party frameworks on both platforms, including the support for hardware-accelerated encoders. Support for streaming and playback of H.264 was therefore implemented for both devices. MPEG-TS container is used for video sent over TCP, and RTP is used for video sent over UDP.

MJPEG streaming of images from the edge device to the end device was found to have good support in the network communication component and its network protocol, described in Section 3.3. Support for streaming from the edge device using MJPEG was therefore implemented. The implementation of streaming from the end device using MJPEG encountered issues, which prevented working support. How the two streaming formats were implemented for the two devices, and the encountered issues, are outlined below.
3.4 Video streaming for the edge prototype

3.4.2 Video on the edge device

On the edge device, the video streamer and receiver are run in separate threads, and both receive and provide data to the image processing module. The mechanism for Qt’s signals and slots described in Section 3.2.2 was used for both streaming formats, to ensure that only the last available video image frames are processed, encoded or transmitted.

MJPEG streaming from the edge device was implemented using the network communication layer described in Section 3.3. Which JPEG encoder is used depends on the image source type, and for Qt’s image objects, it uses Qt’s non-hardware accelerated JPEG encoder.

The two most commonly used media streaming frameworks, GStreamer[28] and FFmpeg[22], were investigated for the purpose of streaming and decoding of H.264 video and decoding of MJPEG video. Both frameworks support both formats, as well as contain a large number of plugins and filters for manipulating video content and encoding, decoding and transcoding media. From the onset, there was no noticeable difference in capabilities between the frameworks. It was eventually decided to use GStreamer due to better support for the Qt framework, subjectively a more user-friendly way to create playback and streaming pipelines. The integration of the GStreamer framework, to allow streaming of H.264 over TCP and RTP over UDP, is described in Appendix A.

3.4.3 Video on the end device

Four open-source video streaming libraries for Android were studied for the other tasks, playback and streaming of H.264. They were chosen based on their popularity in the open-source community. The four are listed in Table 3.1. The list includes libraries that can be used for video streaming, video playback or both.

<table>
<thead>
<tr>
<th>Name</th>
<th>H.264</th>
<th>MJPEG</th>
<th>Accelerated encoders used by default</th>
</tr>
</thead>
<tbody>
<tr>
<td>GStreamer [28]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>FFmpeg [22]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>libstreaming [38]</td>
<td>Only streaming</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>rtmp-rtsp-stream-client-java [52]</td>
<td>Only streaming</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3.1: The evaluated video streaming and playback libraries for the end device.

The work done for getting video encoding and streaming implemented focused on hardware-accelerated support, to make use of the device’s support chips. Therefore the primary focus was put on the two libraries that support hardware-accelerated H.264 encoders by default.

Libstreaming is one of the most popular libraries for streaming video on Android, being one of the first released full-featured open-source libraries for this. It has not seen much development in recent years. It supports hardware-accelerated H.264 encoding through Android’s MediaCodec API and streaming of it via RTP over UDP. Issues were encountered when testing it with the available Android device, and it was thus not further investigated.

The library rtmp-rtsp-stream-client-java is a modern library with the first release in 2017 and at the time of the development in this project (in 2019) still under active development. It supports hardware-accelerated H.264 encoding through Android’s MediaCodec API and streaming of it via RTP over UDP. It is designed for being a client for video streaming servers with several streaming-control related features implemented. Adding this functionality would bring unnecessary overhead to the prototype, which only needs direct communication between two devices. Instead, the edge prototype includes a modified version of the library, which later
3.5 Mixed reality for the edge prototype

This section describes how the edge device's Mixed reality component was developed. The component does the processing for simultaneous localisation and mapping of the surroundings, to both generate a point cloud of the structures and determine the device's position and rotation in the model. With that information, the component is able to draw MR graphics, with both the original received image, the generated point cloud as well as three-dimensional virtual objects. In Figure 3.4, the overview of the edge device components, the Mixed reality component is divided into the components Mixed reality interface, SLAM and Mixed reality graphics.

3.5.1 Mixed reality interface

An integration layer was developed to act as a boundary and handle all communication to and from the SLAM and Mixed reality graphics components, to enable the testing of different frameworks and implementations. The other modules in the edge device software managing tasks like video transmission and retrieval are thus very loosely coupled to the framework in use and can run independently in separate processing threads. To ensure that the integration layer did not affect performance when connected to the video streaming modules, an echo image module was implemented that sends back the original image without transforming it. Internally the image processing supports both the use of Qt's QImage class, OpenCV's Mat library as well as byte streams of JPEG images, to allow the testing, evaluation and extension of different libraries.

To allow the mixed reality related software, running on the edge device, to be tested independently of the end device, modules were written to allow input from both local webcams.
connected via USB to the edge device and local files with images stored on the server. This
was also extended to have the local image feed replacing the input from the end device so that
the MR content sent back to the devices are created using the edge devices local camera. This
option is configurable by the console parameters. During the integration step, a windowed
graphical user interface was added to the edge device to provide local debugging capabilities
without the need to have an end device connected. The interface outputs the same images
that the system sends to the video transmitting module.

Many SLAM libraries, including ORB-SLAM2, require a camera matrix to be provided by
the user. This camera matrix details the attributes for the device’s camera and its lens and
ensures that lens diffraction does not affect the calculations. The integration layer contains
a mechanism to let the user calculate the device’s camera model, which can then be stored
on the end device and provided the next time the user connects. Further details of this
implementation can be found in Appendix A.

3.5.2 Simultaneous localisation and mapping

SLAM is an area that has gotten a lot of interest over the past years. There exist several
frameworks, so no focus was spent on developing a new framework. That work came instead
to be the integration of an existing one and modify it to fit the outlined requirements for the
edge prototype.

The requirements for the framework to be investigated for this thesis were that they were
available as open-source and that they supported monocular SLAM, i.e. the input being one
camera without depth sensors. The frameworks were found doing an online survey on Google
Scholar and GitHub. The found frameworks are listed in Table 3.2.

The four frameworks are listed below, grouped by how many points are in the resulting
point cloud.

1. PTAM and ORB-SLAM2 are indirect SLAM, picking out the distinct features with the
most contrast in images and from that produce sparse point clouds.

2. LSD-SLAM is direct SLAM, taking into consideration more points in the images than
only point features, including observed edges, and from that produce semi-dense point
clouds.

3. CNN-SLAM is a new type of SLAM that utilises machine learning. The monocular
camera input is fed to a convolutional neural network that produces predicted depth
maps. While the depth maps are predictions and not of the same quality as depth maps
taken from a depth-sensing camera, these predictions can give enough estimates to be
useful for SLAM. The algorithms in CNN-SLAM can combine the estimated depth maps
with image-feature analysis of the original camera images and produce a dense point
cloud.
Since our baseline for comparison to a local application running a framework like ARCore, which provides sparse point clouds, the additional computing power that would be spent to create denser point clouds would give no benefit. A sparse point cloud provides enough accuracy and information to be able to insert and display virtual objects. Thus only PTAM and ORB-SLAM2 were investigated further.

Both PTAM and ORB-SLAM2 follow the same parallel tracking and mapping method, introduced first with PTAM in 2007. There are several differences between them. For image feature detection, PTAM uses the FAST method, while ORB-SLAM2 uses the faster and more capable ORB. ORB-SLAM2 supports loop closing and has performance optimisations, including the use of fast Bag of Words to lookup keyframes. PTAM has not seen any development by its original creators after the release of its source code by Oxford University in 2008, with some of the external libraries losing support for modern platforms or been made obsolete by modern alternatives. ORB-SLAM2 has not seen much development by its original creator in recent years either, but a large user community has documented all the limitations and found workarounds. It also benefits from being developed in 2016 and 2017, with that it is easy to run on a modern Linux platform such as Ubuntu 18.04. Therefore ORB-SLAM2 was found to be the most appropriate framework.

Frameworks that use movement and acceleration data from the device’s Inertial Measurement Unit (IMU) were not considered, meaning that for example, ETH Zürich’s visual-inertial framework Maplab was excluded from the investigation. Although most phones have such sensors, adding such a requirement for transmission of that data from the end device to the edge device would have made the solution more complex and was not necessary for the prototype’s simple MR use cases.

Some modifications of ORB-SLAM2 were done for this thesis work. This include changes to the loop closing and the loading of the Bag of Words vocabulary data set. These modifications are described in Appendix A.

### 3.5.3 Mixed reality graphics

A component for drawing MR content was developed for the edge device. It uses headless rendering, which draws three-dimensional graphics off-screen and removes the need for a graphical user interface on the edge device. All virtual graphics are drawn on top of the original image sent from the end device. The original image is not modified in any other way. A detailed view of the drawn MR content can be seen in Figure 3.8, which displays the point cloud and a 3D graphics on top of a real-world wood table.

![Figure 3.8: Close-up view of the mixed reality graphics.](image-url)
The graphics framework supports rendering advanced objects, and the end device can use the network component to configure what object should be drawn. For testing purposes, a simple hardcoded object was added, consisting of seven polygons with four vertices each. The object was designed to look like a house with a ground surface and with different colours for all surfaces. The house has the benefit that it is easy to see the direction and rotation for the added object.

The graphics component processes the point cloud it receives from the SLAM framework and draws the visible points as green dots. This 2D drawing uses ORB-SLAM2’s internal functions, which in turn call OpenCV’s image library. There was experimental support for displaying all detected point cloud points in a rotating 3D view, but that was dropped due to performance issues.

When a point cloud has been created, it is possible to add a virtual object to it. The algorithm used for choosing which positions and rotations to use when inserting the objects in a point cloud is similar to the algorithm in the local prototype described in Section 3.1. The algorithm was taken from an open-source application developed by the team behind ORB-SLAM2. It takes the point cloud and tries to find points whose positions share a flat surface. The algorithm can lead to false positives if the camera is facing uneven objects, but it works well under controlled circumstances when the user makes sure that the camera is facing a flat surface.

To render these virtual objects the graphics component uses the OpenGL API\(^3\). The headless rendering of OpenGL 3D graphics is done with the open-source library Off-Screen Mesa\(^4\), which renders everything using the main CPU. For further implementation details related to the 3D graphics renderer, see Appendix A.

---

\(^3\)https://www.opengl.org
\(^4\)https://www.mesa3d.org/osmesa.html
The two prototypes described in Chapter 3 were evaluated based on the three aspects listed in Section 1.4, which were responsiveness, stability and resource usage. Four different configurations of the edge prototype were tested and compared. This chapter starts by introducing how the tests were done. The last four sections describe the three aspects tested, and which metrics were used for the tests. Each section includes the results retrieved from its tests, with a brief analysis of them. Further discussion about the results is included in Chapter 5.

4.1 Overview

This section gives an overview of how the tests were done, with the configuration and test data shared between all tests.

4.1.1 Metrics

In total, seven different metrics were used in the evaluation of the edge prototype. These are listed below.

- **Resource usage**
  - Required network bandwidth, the lowest video bitrate possible for the SLAM system to function.
  - Processing power usage, measured on both devices.

- **Responsiveness**
  - Round-trip time for an image frame from the camera.
  - How much time it took to add a virtual object to the mixed reality scene.
  - How much time the server spent on processing an image.

- **Stability**
  - The number of received frames, measured on both devices.
  - The percentage of the received frames that the edge device was able to process.
4.1. Overview

<table>
<thead>
<tr>
<th>Configuration</th>
<th>End Device Streamer</th>
<th>Edge Device Format</th>
<th>Transport Protocol</th>
<th>Jitter Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.264 over UDP with acceleration</td>
<td>MediaCodec</td>
<td>H.264</td>
<td>UDP</td>
<td>No</td>
</tr>
<tr>
<td>H.264 over UDP without acceleration</td>
<td>GStreamer</td>
<td>H.264</td>
<td>UDP</td>
<td>Yes</td>
</tr>
<tr>
<td>H.264 over TCP</td>
<td>GStreamer</td>
<td>H.264</td>
<td>TCP</td>
<td>No</td>
</tr>
<tr>
<td>MJPEG and H.264 over TCP</td>
<td>GStreamer</td>
<td>MJPEG</td>
<td>TCP</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.1: The four evaluated edge prototype configurations.

4.1.2 Test environment

An LG G6 smartphone running Android 8.0.0 was used as the end device for the tests. LG G6, released in early 2017, contains a Qualcomm Snapdragon 821 system-on-a-chip with a quad-core CPU and 4 Gb RAM. All third-party applications on the end device were uninstalled, and the internal applications that could be disabled were disabled. A Lenovo Thinkpad T450s PC running Ubuntu 18.04 was used as the edge device for the tests. The T450s has an Intel Core i5-5200U CPU and 12 Gb RAM. No other application, scheduled tasks or other services were running in the background while the tests were done. All the edge computing measurements were done over a local network set up using a Compal CH7486e router disconnected from the Internet. The edge device was connected to the network using Gigabit Ethernet (1000 Mbit/sec), and the end device was connected using an 802.11n wireless network. 802.11n has a theoretical maximum bandwidth of 600 Mbit/sec, but practical performance is often significantly lower due to signal interference, and signal noise [69]. The devices were placed within one meter from the network gateway, and the same positions were used for all tests.

4.1.3 Edge prototype configurations

The end device only supports streaming of H.264 video, with two different streamers. The edge device supports streaming of MJPEG, using the custom Network Component, and H.264 using GStreamer. For the evaluation, four different configurations for the edge and end devices were tested. The configurations are listed in Table 4.1. All four configurations ran all tests, and for each configuration and test, the same setup was run 30 times, to filter out noise. Both the edge and end devices were restarted between all the tests, to ensure that they all were run under identical premises. For all tests, the camera resolution was set to 640x480, the only resolution supported by GStreamer’s camera component on the device used for testing.

Stability issues were observed during the tests with GStreamer streaming H.264 over UDP, where network jitter in the test environment leads to incorrectly decoded image frames. These incorrect frames being sent to the Mixed reality component generated unpredictable results. To solve this issue, the GStreamer network jitter reducing element Rtpjitterbuffer was included on the edge device for that configuration. Due to compatibility issues between the MediaCodec library and the jitter reducing element, it could not be included when streaming with hardware-acceleration. Similar network issues that had been observed with GStreamer streaming without Rtpjitterbuffer were not observed with the MediaCodec library. It is unclear what caused this difference in behaviour between the streamers. More details about network jitter reducing element can be found in Appendix A.
4.1.4 Test data for the edge prototype

To ensure reproducibility and remove the need to do all the tests manually, they were done using video playback functionality on the end device. The video playback mechanism acted as a camera and fed the decoded video frames into the standard end device processing pipeline. All measurements of time in the different tests were using timers with nanosecond resolution, with the measurements for presentation purposes were rounded to the nearest millisecond. See Appendix A for information about how video playback and the timers were implemented.

One reference video was recorded using the LG G6 with its standard video recording software. The video, with a duration of 53 seconds and with 30 frames per second, was filmed in a well-lit indoor environment, portraying a white table with a plant, a banana, a laptop and a coffee cup. The video was downscaled to the resolution 640x480 using a PC and stored on the end device’s local memory. Images from the reference video can be seen in Figure 4.1.

Figure 4.1: The reference video used for testing.

4.1.5 Tests of the local prototype

While the edge prototype could be tested using video playback, to ensure reproducibility, this was not possible for the local prototype. There is no ability in the ARCore API for using a custom video, or another type of custom software object, as input to the point cloud mapping calculations. It was tested if it was possible to do the tests by having the device’s camera pointed to a display that had the reference video playing, but ARCore was not able to create any point clouds. It is a strong indication that a large part of ARCore’s functionality relies on knowing how the device is moved, information that it retrieves from the device’s Inertial Measurement Unit. All the tests of the local prototype, therefore, had to be done by manually moving the device. They can therefore not be directly compared to the results from the edge prototype, but are included as a representative of the current state.

4.2 Resource usage

This section contains the two aspects that were evaluated for the prototypes’ resource usage. The first aspect was which network bandwidth was required for the edge prototype to work reliably. The second aspect was how much processor power the two devices used when running the software.
4.2. Resource usage

4.2.1 Required bandwidth

The first resource usage that was considered was how much network bandwidth was needed by the edge prototype configurations. In the edge prototype, a major part of the network traffic is caused by video streaming between the devices. From the end device, a camera feed is sent to the edge device, and in return, it gets back a stream with mixed reality graphics. The four tested edge prototype configurations have three different methods for streaming video from the end device. The methods all used H.264, with the difference between them that one sent the video over TCP and two sent it over UDP, with one of the latter used hardware-acceleration for the encoding.

The part in the edge prototype that determined the required bandwidth, or bitrate, for the end device’s video streaming was found to be the ORB-SLAM2 library. ORB-SLAM2 uses OpenCV’s ORB to analyse the input images and detect image features. These image feature descriptors are then used both to create a point cloud map using bundle adjusting and to track and calculate the device’s current position and movement. The images sent to the Mixed reality component, therefore, need to be of high enough quality that the feature detectors can detect and track points over multiple images.

With the type of lossy compression used in H.264, lowering the bitrate worsens the image quality. How the many different features of H.264 are implemented and configured in the encoders can also have consequences on what encoded image is produced for a given bitrate. If the bitrate is too low, and the image compressed too much, it might make the SLAM system unusable. The effects of sending low-quality input to the SLAM component can include it being unable to track the device’s position, calculate an incorrect position, being unable to create a point cloud map or adding false points and structures into the map [71].

There is no corresponding minimum bandwidth for the streaming from the edge device to the end device. No computer-based image analysis is done on the end device, and having a low bitrate on the edge device’s stream only affects the image displayed to the end-user. What users require regarding image quality was out of scope for this thesis.

The metric for the test was to measure, on average, how many image features in each video frame could be matched to a point in a calculated point cloud. It thus measured both if the SLAM system had been able to create a point cloud and also how many image feature points could be used in the received images to determine the position and extend the point cloud. The detection was done on the edge device, without any images sent back to the end device. Bitrates with an interval of 500 kbit/s were measured, from 500 kbit/s up until the level where it was observed that raising it higher did not improve the measured values.

Results

The results from these measurements can be found in Figure 4.2. The MediaCodec library displayed some unpredicted behaviour, where it in some cases sent the original H.264 encoded reference video, without decoding it and encoding it to a new bitrate. It is unclear why this happened, as both the decoding and encoding was managed by the underlying Android platform and LG device software. It was only observed for lower bitrates. When using the live camera input, this behaviour was not observed. This prevented reliable testing, and as such, no results could be produced for that method.

Measured values of 0 meant that the SLAM component had not been able to create any point cloud at all during that test run, i.e. the image quality was too low for it to be able to detect enough features. It can be seen in the graphs that for the reference video when increasing the bitrate, the number of average tracked image feature points eventually averaged around 340, which indicates that this was close to the number of features available in the reference video source. When the edge device received these high-quality video image frames, the SLAM component was able to create the point cloud already after around 1300 milliseconds, and continue to keep it stable throughout the test run. This maximum was observed in all tests.
when streaming with GStreamer over TCP with the bitrate of 2000 kbit/s. When streaming with GStreamer over UDP at a bitrate of 2000 Kbit/s, a majority of the tests were not able to find any point cloud. Instead for UDP, the bitrate had to be increased to 4000 kbit/s before the point cloud, and tracking was found to be stable. The MediaCodec library was also found to be reliable at 4000 kbit/s.

While some tests of the streamers with lower bitrates resulted in many feature points being found, the measurements fluctuated a lot. None of the tests with GStreamer over UDP set to bitrates lower than 2000 resulted in a point cloud being created. Increasing the bitrate more produced slite more feature points, to a maximum average of 350, but it did not improve the point cloud stability, nor the time until a point cloud had been found.

The difference in required bandwidth between TCP and UDP was a factor of two. Where the UDP configuration made every frame a keyframe, with the TCP configuration, only significant scene changes lead to new keyframes, significantly lowering the needed bitrate to produce the same image quality. GStreamer was configured to encode in the fastest mode, to prioritise encoding speed over image quality. It was not investigated how setting the GStreamer encoding to a slower encoding mode would affect the required bitrate.

Based on the results from this test, it was decided that the bitrates listed in Table 4.2 were the minimum required bandwidth for the edge prototype for TCP and UDP respectively.
### 4.2. Resource usage

#### Table 4.2:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Bitrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>2000 kbit/s</td>
</tr>
<tr>
<td>UDP</td>
<td>4000 kbit/s</td>
</tr>
</tbody>
</table>

Table 4.2: Required bitrates for H.264 transmission from the end device.

#### 4.2.2 Processing power usage

The four test configurations outlined in Section 4.1.3 were evaluated in regards to how much CPU processing power they consumed while running their respective software.

Processing usage on the edge device was measured using Ubuntu’s system service `top`, and a custom-written shell script to retrieve and store the value once every second. The system service lists the CPU usage as a percentage of a CPU core’s total processing capacity, so for the Intel Core i5-5200U CPU used for the tests, which to the operating system presents four CPU cores, the maximum value is 400%. Tests were also run when the mixed reality framework was disabled. In these tests, where the end device’s image frame was sent back immediately without any altering by the mixed reality interface, the only major processing tasks on the edge device were the video receiver and video streamer.

Processing power usage on the end device was measured using Android Studio’s `Android Profiler`\(^1\). This tool measures the usage of the connected device’s processors and gives it as a percentage with the maximum value of 100%. The processing power usage on the end device was only measured when the edge device had the Mixed reality component activated. The video stream bitrates for the different configurations were the ones listed in Table 4.2, with streams from the edge device and from the end devices configured to the same value.

#### Results

The measured results from the edge device can be seen in Figure 4.3. The processing power usage on the edge device was around the same for all configurations. That it never reached the maximum 400%, i.e. using all available processing power, was due to only three threads consuming most of the resources. These three threads were the graphics renderer thread and the ORB-SLAM2 instance’s two threads for tracking and the local mapping. It is a clear indication that it is the per-core CPU processing power that determines the performance, as a single thread can only occupy one CPU core, i.e. 100%. Only increasing the number of cores, with the same per-core performance, would still lead to similar numbers.

The measured results from the end device can be seen in Figure 4.4. Android Profiler only displays the app’s resources, and it can thus be misleading to determine how much CPU power is used for the different modes. The native libraries like GStreamer may create processing threads that are not included in the overall image. The Profiler consumes a lot of processing resources on the device it is running on, and it was observed that the received frame rate dropped by around 40% when it was running. It is unclear why the hardware-accelerated H.264 encoder in the MediaCodec Library consumes almost as much processing power as the GStreamer based encoder running on the CPU. One likely reason is that it also includes a CPU-based H.264 decoder, and also that the RTP packetisation of H.264 is done by the library using Java and not native C++ code. It is unclear how much the total processing power usage would be if also the receiver was to use MediaCodec, or if it had a more optimised source code.

\(^{1}\text{https://developer.android.com/studio/profile/android-profiler}\)
4.3 Responsiveness

The system’s responsiveness, or time required by the system to act on the input, was evaluated in regards to three aspects. These aspects were:

- The time before an added virtual object was visible.
- The total round-trip time for a camera image.
- How much time was spent for each frame in the edge device.

They are outlined in the three sections below. For each aspect, all four configurations described in Section 4.1.3 were tested. The video stream bitrates for the different configurations were the ones listed in Table 4.2, with streams from the edge device and from the end devices configured to the same value.

The edge prototype includes a ping mechanism, to measure the time for the packet transportation between the devices. The end device sends short packets over UDP, and the edge device when it receives them immediately responds with an acknowledgement over UDP. The round-trip times for these ping messages were logged by the end device during the tests and were measured to be between three and eight milliseconds.

4.3.1 Adding a virtual object

The aspect of the communication layer’s and the mixed reality system’s responsiveness was evaluated, in regards to the time needed by the system, after a point cloud had been generated,
to add and display a virtual object. On the end device measurements were made of the time that had passed from the pressing of the GUI’s Add Mixed Reality object button until a generated image had been received containing a virtual object. To ensure reproducibility, an automatic trigger was set to trigger the button programmatically 15000 milliseconds after the test session had started.

To enable the end device to detect when the sometimes small virtual objects drawn by the edge device were visible on its screen, two separate mechanisms had to be added to its H.264 and MJPEG receivers. For the MJPEG receiver, the image file segments used in the communication stack was extended on both platforms to include support for a four bytes metadata attribute, which the edge device set to whether virtual objects were included in the image or not. For H.264 streaming this mechanism was initially planned to be based on H.264’s Supplemental enhancement information (SEI) standard for attaching metadata to the video images frames, but this could not be used due to lack of support in the GStreamer encoder. Instead, it was based on having the edge server set a couple of pixels’ colour in the top left part of the generated image with metadata, to inform the image receiver of the overall image content. On the end device, the GStreamer pipeline was modified to include a function to check these pixel value. For the tests the colour pink was chosen, as it was not present in any of the frames in the video and the end device thus by, checking for just a single pixel, could determine if the image contained the virtual object. When measured on both devices, the overhead for adding and reading this metadata was too low to be measurable. For implementation details, see Appendix A. When the end device’s MJPEG or H.264 receivers had detected this metadata value in the received image frames, they sent a signal to the time logging functions to have the timestamp recorded.

For the local prototype, it was implemented with timers in the application’s graphics stack, which received data from ARCore and rendered mixed reality. These timers measured the time from the user interface button for adding content had been pressed, until the graphics renderer had drawn the object to the screen.

**Results**

The results can be seen in Figure 4.5. The local prototype can not be used as a direct comparison, as its implementation and behaviour differ, and it is only included for reference. In the edge prototype, the end device sends the commands to add elements separately from the video streaming, using the Network Data Link over TCP. As soon as the short text-based command arrives, the virtual objects are added to the point cloud and included in the following
mixed reality images frames. For both prototypes, the insertion and drawing of virtual objects is the last step in the Mixed reality component’s processing pipeline, before the image frame is sent to the video streamer, or as with the local prototype immediately displayed to the screen. This was most noticeable with the local prototype, where it was possible to have an addition-time as low as seven milliseconds.

Seen in the graph is that the numbers for the configuration H.264 over TCP stick out. There is not a clear explanation for this, but it can be caused by issues in the GStreamer pipelines in the edge device’s video streamer or end device’s video receiver. It can also be due to how TCP packets are treated in the local network and in the devices, or by drifting, packets being stuck in the transportation layer and preventing later packets from arriving in time. The latency being caused by network issues is somewhat disproved by the measurements from configuration H.264 and MJPEG over TCP, which transmits MJPEG over TCP using the Network Data Link. The numbers for this was the lowest of the four edge prototype configurations, which indicates that the latency issues seen with the other TCP configuration were due to issues in GStreamer TCP pipelines.

While H.264 over UDP with acceleration and H.264 over UDP without acceleration had the same configuration for streaming from the edge device, the more variations in the measurements of the former indicate that MediaCodec Library’s streaming caused issues on the edge device. This is further discussed in Section 4.4.2.

![Figure 4.5: Time measured for adding virtual objects for the two prototypes.](image)

### 4.3.2 Round-trip time

The whole system’s responsiveness was evaluated using measurements of round-trip times for individual camera frames. This method measured how long it took from an image being visible by the end device’s camera, until it had been processed by the edge device and returned, and was displayed on the device’s screen.

Time was spent on developing a mechanism to do this automatically, using the reference video input, but due to problems with the Android device’s internal libraries used for decoding and parsing the video, as well as ARCore’s lack of video input ability, this could only be achieved with the GStreamer streamer. When tested with GStreamer, there were still strong indications that the mechanism added latency into the streaming process.

Instead, a GoPro Hero 7 Black camera was used for the measurements. The GoPro camera has a frame rate of 240 frames per second, i.e. an interval of 4.16 milliseconds between frames. A setup was constructed with a computer display alternating between different colours. The GoPro camera filmed both the end device’s screen as well as the computer monitor, and the
4.3. Responsiveness

video could later be exported as individual images using the tool FFmpeg. By manually looking at the video frames, the round-trip time could be calculated by counting the number of frames from the change was only visible on the computer display until it was visible also on the end device. With 240 frames per second, fifty frames meant that the round-trip time was 208 milliseconds.

Results

The measurements are in Figure 4.6. The local prototype can not be used as a direct comparison, as its implementation and behaviour differ, and it is only included for reference. None of the edge prototype configurations came near the local prototype’s round-trip times.

The edge prototype configurations have measured times around twice as long as the times seen in Figure 4.5. One exception is H.264 with MJPEG over TCP, which had the lowest measured times for the edge prototype configurations for adding virtual objects, but here is only the third lowest. One explanation is that the time for streaming with H.264 over TCP from the end device, included in this graph, adds enough time to make the total round-trip time longer than the configurations transmitting H.264 over UDP. The TCP round-trip times vary a lot, which indicates that there is room for improvements in the GStreamer based H.264 TCP streaming mechanism, in addition to the issues mentioned in Section 4.3.1, these also affect the streaming from the end device to the edge device.

![Figure 4.6: Round-trip times for the two prototypes.](image)

4.3.3 Edge device processing time

The edge device’s responsiveness was also evaluated, with regards to how much time was spent on processing a video frame before it could be returned to the end device. This used the same setup as the previous tests, with the end device streaming the reference video to the edge device using the different configurations. Time measurements were then taken inside the edge device when the image frame crossed from one component to a sub-component, or vice-versa. The pipeline with the components involved in the processing can be seen in Figure 4.7.

In total, three timers were used. The first timer measured how much time the frame spent inside the Graphics Rendering component; from the time the component received it from the mixed reality interface until it was ready to be sent back. The second timer measured how
4.3. Responsiveness

Figure 4.7: The edge device’s components involved in the processing of a video frame.

much time the frame spent inside the Mixed reality component, from the moment it got the frame from the system until it was ready to be sent back. The third timer measured how much time the frame spent inside the Edge Device System. From it was retrieved by the video receiver, and including the time it has to wait for the mixed reality framework to be ready until it has been processed and is sent to the video transmitter. It was not possible to get access to the processing times in the GStreamer video receiver and transmitter at the edge device, and they are not included in the system measurements.

Measurements were also taken of the Edge Device System when the Mixed reality component was disabled and just sent back the received image to the end device. This was done to show the overall system performance when there are not any delays or gridlocks caused by the mixed reality framework in the processing pipeline.

Results

The measurements can be seen in Figure 4.8. The measurements follow the structure in Figure 4.7. The leftmost measurement is time a decoded image frame spent in the Edge Device System. To the right of that is measurements of the part, of the previous time, that was spent in the Mixed reality component. To the right of that is the measurements of the part of the previous time that was spent inside the Graphics Rendering Component. The right-most measurements in the graph are the time the decoded frames spent in the Edge Device System when the Mixed reality component was turned off.

Only frames that were sent back to the end device are included, and not frames where long processing times in the Mixed reality component caused them to be replaced in the waiting queue by later arrived frames. Average measured processing time per frame for the system was 73 milliseconds. The Mixed reality component took on average 62 milliseconds, and the graphics rendering on average 27 milliseconds. The video decoding times could not be measured, and there was no observed difference in the processing times between the different video streaming configurations.

The variation in the measurements was significant, with some frames taking twice the average time. This was most noticeable when the camera moved fast, as it was seen that ORB-SLAM2 had to do more calculations to track the camera and this lead to longer processing times. The longer processing times seen for the whole system compared to the Mixed reality component were caused by frames spending time in the wait queue before the Mixed reality component was ready to process them. With 30 frames per second, the SLAM component had 33.3 milliseconds to process them, to not cause any propagated delays. As can be seen in the graphs, the processing time was most of the time much longer. The processing time added by the graphics render was close to that limit. This in part a consequence of the edge device lacking support for hardware-accelerated headless OpenGL 3D rendering, forcing everything to be rendered with the slower CPU-based renderer.
The measured processing times for when the mixed reality module was turned off are too low to be seen. They were between zero and ten milliseconds, with 90% of the measurements less than two milliseconds. When put in the perspective of Figure 4.6, it can be seen that only a small chunk of each image frame’s processing time in the edge prototype is spent on creating mixed reality in the edge server. A much bigger part (around 90%) of the time is spent on transferring the images between the devices.

![Figure 4.8: The observed processing times on the edge device](image)

### 4.4 Stability

The prototypes’ stability, i.e. their ability to provide consistent results over time, was evaluated using received and displayed frames per second and how many of the arrived image frames that were not processed by the edge device. The video stream bitrates for the different configurations were the ones listed in Table 4.2, with streams from the edge device and from the end device configured to the same value.

#### 4.4.1 Frames per second

The first method measured the stability and performance in regards to how many frames per second were processed by the devices. This included both the number of frames received by the edge device and the number of frames displayed to the user by the end device.

For the edge prototype, the video receiver on the edge device, as well as both video receivers on the end device, were modified to log when new image frames arrived. For the edge device, this used the timer mechanism described in Section 4.3.3. For the end device’s H.264 receiver this measurement used GStreamer’s *pad probe* functionality, by attaching a callback function to the element displaying the graphics and using an array data structure with one bucket per second. For the end device’s MJPEG receiver, a similar mechanism was implemented in the communication stack to log received image data segments. For the local prototype, this was measured by having the local graphics stack log when new frames were drawn.

**Results**

The results have been divided into three figures. The measurements of the frames per second displayed on the end devices when running the edge prototype are included in Figure 4.9. The measurements for the frames per seconds displayed when running the local prototype are included in Figure 4.10.
The measurements for the frames per second received by the edge device are included in Figure 4.11. When it came to pushing images to the edge server, all configurations and streaming methods were able to push around 29 frames per second. There is a big difference for all configurations between the number of frames received by the edge device and the number of frames displayed on the end device. This is especially noticeable with the MediaCodec streamer, which had the highest frame rate as measured on the edge device. However, it also had the lowest frame rate when measuring the images sent back from the edge server. The reason for this is further discussed in the next section. The local prototype averaged a frame rate of around 28 frames per second.

The measured frames per second when the edge device did not generate mixed reality were still lower than 28 frames per second. It is not completely clear why this happened, but the likely reason is that they were dropped by the device’s video streamers and receivers. The GStreamer streaming pipelines on both devices were configured to use a one place input buffer, and drop the frame in the buffer if a new frame arrived.

4.4.2 Frames not processed

The Mixed reality component in the edge device contains the buffer mechanism described in Section 3.2.2, which prevents gridlocks in the processing queue, by removing image frames that are still waiting to be processed when a new frame arrives. The different edge prototype configurations were evaluated in regards to this. This was measured by counting how many frames arrived to the system from the video receiver, and counting how many frames were processed. A high number of dropped frames is an indication that the edge device might not have the processing power to process all frames. It can also be an indication that while the measured video transmission might be 30 frames per second, and the frames arrive in order, they arrive with high jitter where some frames arrive very close to each other and others arrive with some significant delays.

Results

The measurements can be seen in Figure 4.12. As can be seen, the number of processed frames during the test cases varied between the edge prototype configurations. What could be seen in the logs was that the dropped frames did not happen at a steady pace. Instead, it could be one frame per two seconds, and then suddenly 15 dropped frames when the SLAM tracking got stuck on a frame after something that triggered more processing, including fast camera movement or video streaming issues. For the two configurations where the video was streamed UDP, MediaCodec Library’s measured values stick out compared to GStreamer’s. This is an indication that there were issues in the implementation or configuration with either the end device’s video streamer or the edge device’s UDP H.264 video receiver.
4.4. Stability

**H.264 over UDP with Hardware-Acceleration**

**Without Mixed Reality**

**With Mixed Reality**

**H.264 over UDP without Hardware-Acceleration**

**Without Mixed Reality**

**With Mixed Reality**

**H.264 over TCP**

**Without Mixed Reality**

**With Mixed Reality**

**H.264 and MJPEG over TCP**

**Without Mixed Reality**

**With Mixed Reality**

Figure 4.9: Frames per second displayed on the end device running the edge prototype.

Figure 4.10: Frames per second displayed on the end device running the local prototype.
4.4. Stability

Figure 4.11: Frames per second received by the edge device

Figure 4.12: Percentage not processed, of the number of frames arrived at the edge device.
This chapter contains discussions about the thesis work, both concerning the results of the evaluation in the previous chapter but also in perspective about the method and the design choices. The two research questions defined in Chapter 1 are:

- Which level of resources will be needed at the edge device to support an MR application? How close can it come to the performance and timeliness of an application for a modern phone with native support for this technology?
- What are the bottlenecks and encountered issues when creating such a solution, and what are the best ways to overcome them?

5.1 Method

The thesis’s method consisted of the active development of an edge computing prototype. The development was iterative, verifying what worked and what did not, with a modular design of the edge prototype. This proved valuable for the understanding of the bottlenecks as when the work started, the lack of related research made it unclear where the most significant issues would be. It was very early on found that a large chunk of the issues was related to video streaming, and thus the thesis came to put significant focus on this aspect, and its impact on the performance evaluation.

All development and testing were done with a laptop containing a CPU that was part of Intel’s low voltage mobile CPU product group. Having a low-powered CPU was a benefit to the thesis as it made it more visible where the performance-related bottlenecks were. The forcing of non-hardware accelerated video codecs, graphics renderers and SLAM algorithms also reduces the dependence on specific hardware. This makes it easier for other people to expand and verify these findings. All the source code written for this thesis can be downloaded from Linköping University’s web site\(^1\).\(^2\).

\(^1\)https://gitlab.liu.se/ida-rtslab/public-code/2019_mrleo_server
\(^2\)https://gitlab.liu.se/ida-rtslab/public-code/2019_mrleo_client
5.2 Resource usage

As was shown in the evaluation, the edge prototype consumes significant amounts of processing power, on both the end device and the edge device. The lack of CPU power on the edge device was shown to cause longer response times and dropped frames when the edge device did not have enough resources to process the frames in time.

ORB-SLAM2 doing all the calculations using the regular CPU proved to be a major bottleneck. Studies into using Nvidia’s parallel computing platform CUDA to offload processing in ORB-SLAM2 to the computer’s graphics card were able to increase the performance by 33% \[13\]. Two forks of ORB-SLAM2 have been published\(^3\)\(^4\) with additions that make it possible to run the SLAM framework on Nvidia’s CUDA powered computing platform Jetson\[46\]. With the figures published by the developers of the forks, the SLAM framework’s performance increased between 50% and 200%, when using Nvidia Jetson TX1, compared to running it on a high-end PC. Doing these types of accelerations were not possible with the limit in time for this project and the lack of such hardware, but it is clear that support for this is preferable for the prototype.

5.3 Responsiveness

To be able to analyse how the added latency caused by the video transmissions affects the prototype, one needs to determine how long latency is acceptable to the user. With high-speed gaming, it has been observed that 100 milliseconds is a limit \[60\]. For mixed reality projected inside VR glasses, it has been argued that it needs to be under 20 milliseconds, to prevent motion sickness \[17\]. As seen in the measurements of the round-trip times, the time from the camera had seen an object until it was visible on the end device’s screen, this level of responsiveness could not be achieved with the edge prototype.

The four different test configurations for the edge prototype had average round-trip times between 380 milliseconds and 800 milliseconds, with the lowest one when using the hardware-accelerated MediaCodec Library. The majority of the time for each image frame was spent in transmission between the devices. The only other processing involved, the mixed reality processing, only contributed a minor part, 73 milliseconds per frame, on average.

5.3.1 Hardware-accelerated codecs

The non-accelerated GStreamer codec had noticeably longer round-trip times than the hardware-accelerated MediaCodec Library. When using MediaCodec Library for encoding on the end device, it still had to use a non-accelerated codec on the edge device for encoding the H.264 stream. On both devices, non-accelerated codecs were used for decoding the streams. It is unclear how big the performance increase would be if all four codecs in use were hardware-accelerated. It nonetheless should have a high priority for further work on lowering the latency.

5.3.2 Custom streaming pipelines

As could be seen in Figure 4.6, there was also a significant difference in round-trip times between GStreamer streaming over UDP and TCP. Despite the encoding and decoding pipelines being equal in other aspects, the TCP round-trip times were on average 330 milliseconds longer than with UDP. TCP round-trip times also varied greatly and could go up to 1200 milliseconds.

There was a big difference in transmission latency between using H.264 with GStreamer and using MJPEG with the custom network protocol, despite both being sent over TCP. This

\(^3\)http://yunchih.github.io/ORB-SLAM2-GPU2016-final
\(^4\)http://github.com/hoangthien94/ORB_SLAM2_CUDA
difference indicates that a significant processing overhead was in the GStreamer components, despite the configuration work that was done to speed up the processing. The edge prototype uses GStreamer’s built-in elements for TCP transmission, and they probably caused the delays. As described in 2.4.6, it is possible to extend GStreamer to use customised video streaming components. However, time was not available for that during the work on this thesis. Further studies regarding lowering the latency should involve investigating a more customised streaming pipeline. This streaming pipeline can include both custom streaming formats and custom implementations, which can minimise the overhead and focus on latency.

5.3.3 Less advanced compression

With the measurements for the time before an added object was visible, seen in Figure 4.5, it was shown that the fastest method, on average, was MJPEG over TCP, with 79 milliseconds delay. This was 49 milliseconds faster than the second-fastest method, H.264 over UDP. With the configuration that used MJPEG over TCP for streaming from the edge device, it was observed that the 30 frames per second, with the resolution 640x480, lead to an average bitrate of 7080 Kbit/s. This bitrate was around twice the one for H.264 over UDP. Despite the high bitrate, there was no triggering of the edge device’s network component’s transmission gridlock mechanism, described in Section 3.3. That it was not activated is likely due to the high bandwidth available in the testing environment. For situations where high bandwidth is available, edge computing for mixed reality benefits strongly from less computing-intensive formats.

5.4 Stability

There were many issues related to the stability of the video streaming encountered during the development. This despite all testing being done on a local network with the edge device connected via Ethernet and the end device connected wirelessly and being placed within a meter from the access point.

5.4.1 SLAM

In the edge prototype, glitches with the video streamed from the edge device to the end device only cause user annoyance. However, transmission issues when images are sent from the end device to the edge device might severely impact the computer vision algorithms in the SLAM system and leave it in an unusable state. These risks became obvious when UDP streaming was first implemented in the prototype, and packet loss and corrupt packets often led to incorrect images. These images made the SLAM tracker lose its pose and the virtual content end up in strange positions on the screen. The video streaming issues were aggravated by ORB-SLAM2 being found to be prone to lose the tracked pose if the camera moved too fast, or if there were severe packet loss for more than a couple of frames.

After having lost the pose, the SLAM framework’s re-localisation can take some time. The time will be longer if the end device during the interrupted transmission has moved to a position not previously visited. The gap makes the algorithm treat the new section as a completely distinct location, until it has received enough image features to connect the new keyframes to the previous set.

Studies have shown that while ORB-SLAM2 works well when the input images are of high quality, direct monocular SLAM frameworks such as LSD-SLAM seems to work better in situations where the input images may also contain incorrect data [71]. There was, however, no time available during this thesis to evaluate a second SLAM framework, to compare it to ORB-SLAM2.

The processing of the image frames on the edge device during the evaluation was shown to induce minor video stuttering. The instability in the framerate was caused by it not being
5.4. Stability

Figure 5.1: Example of where a broken keyframe has led to also subsequent video frames being incorrect.

able to process the frames in time, causing frames to be dropped. Having a processing time lower than the frame arrival interval would mean that the server would be ready to process the image as soon as it arrives. With 30 frames per second, the maximum processing time per frame would be 33.3 milliseconds.

5.4.2 Video streaming over UDP

For stability reasons, the video streamers when transmitting from the end device over UDP, feeding data into the SLAM framework, needed to have mechanisms added to reduce the number of incorrect frames. As was seen, also minor glitches in the video input could cause objects to be positioned incorrectly or lead to an unpredicted state.

These mechanisms include having them configured to make every frame a keyframe. Otherwise, packet loss or network jitter could make several frames defective until a correct keyframe had been received. When the B-frames only update minor parts of the image, as described in Section 2.3.1, most of the image remains incorrect. Example of where a corrupt keyframe has led to the next-coming non-keyframes frames having incorrect content can be seen in Figure 5.1. In this screenshot, the displayed image looks mostly grey, caused by the broken keyframe. However, the subsequent B-frames have changed minor parts, making it possible to see the outline of the laptop and the coffee cup. These incorrect images can make the SLAM framework think that the device has moved to an entirely new scene. As seen with GStreamer, it also needed an RTP jitter buffer to reduce the packet loss.

The parts added to the edge prototype for video streaming over UDP were shown to increase the bandwidth requirements significantly. Where streaming over TCP required 2000 Kbit/s, UDP required twice that, 4000 Kbit/s. More unstable network connections than the one used in the evaluation would require even more added mechanisms to minimise the number, if not remove, incorrect frames. These added mechanisms lead to higher bitrate usage and higher latency, reducing some of the benefits otherwise seen with streaming over a one-way protocol such as UDP.

5.4.3 Video streaming over TCP

While not being affected by data corruption or packet loss, the TCP connections in the prototype were noticed to be unstable in other regards. TCP could sometimes lead to severe transmission latencies and round-trip times. The edge prototype lacked from not having any video streaming control mechanism, which could send information back to the receiver about the received streams. Such a mechanism, similar to RTPCP, would significantly improve the stability by preventing severe transmission drift and gridlocks caused by data packets filling up buffers.

It could also be worth investigation to use a more advanced transport protocol, built on top of UDP but with some TCP-like features such as guaranteed delivery and packet
synchronisation. One such protocol that is available is ENet\(^5\). With that, one would be able to customise the time intervals when packets are considered valid and prevent severe packet delays and bottlenecks. It is unclear how well that would work in a practical environment and would require the writing of custom video streaming components for both the streaming and the receiving devices.

5.5 Development of the edge prototype

The design choice to use Qt for the edge device software proved to be beneficial for the thesis work. It supported thread-safe concurrent programming and easy integration of the third-party frameworks. It also supported the incremental development process, and simplified the process of early function testing as well as developing the components independently of each other.

It was noticed during the development of the end device software that Android devices’ behaviour differs between both vendors and operating system versions. These behaviour differences include not only the lower-level systems such as media codecs but also such minor aspects such as how the graphics are drawn. Many aspects are hidden behind the Android API, making troubleshooting more difficult.

It can be argued that for an Android application like this, which acts only as a consumer, with the analysis and graphics processing done remotely, it would be beneficial to write it entirely in native code using Android NDK. This method would allow it to have a more optimised code, with fewer aspects hidden behind the Android API.

5.6 Image quality for mixed reality

The edge prototype’s frame image resolution of 640x480 can seem minuscule when compared to the local prototype’s and ARCore’s 1920x1080, with its 6.75 times number of pixels.

The computational time for all the algorithms in the edge prototype, except the SLAM framework’s local mapping, depends on the number of pixels. Increasing also the edge prototype’s resolution for the tested scenarios would, therefore, affect the performance substantially, increasing the latency and decreasing the frame-rates.

The difference in image quality was also affected by the video encoding, which was lossy for both the streaming from the end device and the edge device. This degradation was aggravated by the encoders being set to the fastest possible encoding mode, which sacrificed image quality. The time available prevented surveys regarding acceptable image quality for this thesis.

5.7 The work in a wider context

In this thesis, it is assumed that all video data is sent unencrypted to an external server. There are significant security and privacy concerns related to this. For all data that is transported over the wireless network, there also needs to be a security mechanism to prevent eavesdropping. While edge computing has several benefits compared to cloud computing when it comes to user integrity, it does not altogether remove the risks.

6 Conclusion

This chapter will try to give the answers to the questions asked at the beginning of this thesis, in 1.3, and also lists how the work can proceed from here.

6.1 Performance

As seen in the thesis, it was possible to create, with the support of freely available opensource frameworks, an edge computing-based application that provides mixed reality to end devices, without having requirements on native mixed reality support. However, the edge prototype was not able to match an application for a modern phone with native support for technologies such as Google ARCore in terms of responsiveness. The configuration of the edge prototype with the lowest per-frame latency had an average of 344 milliseconds, more than twice as long as the ARCore reference application, where it was on average 139 milliseconds. The biggest bottleneck for performance was video streaming, which caused the majority of the per-frame latency in the edge prototype. The mixed reality related processing on the edge device was measured to require, on average, 73 milliseconds per frame.

6.2 Resources at the edge

Affecting the edge prototype’s video streaming performance was the lack of support for hardware-acceleration, forcing most processing, on both devices, to be done by their respective main CPUs. The prototype’s only limited support for such acceleration was on the end device, where it was seen to lead to shorter processing times than the non-accelerated alternatives. This lack of hardware-acceleration was also noticeable with the SLAM framework, used for generating mixed reality, which required the majority of the edge device’s CPU power. The SLAM framework’s processing times can be improved by offloading graphics processing to dedicated hardware using the CUDA platform. Similarly, the processing times for the 3D graphics rendering, which now had to be done by the main CPU, would improve if the prototype were to have access to a dedicated graphics card.

It was seen that the edge prototype configurations where video was streamed over UDP had significantly lower round-trip times than the configurations where video was streamed over TCP. The required bandwidths for the video streaming from the end device to the edge device
were found to be, for image pixel resolution 640x480, 2000 Kbit/s when over TCP and 4000 Kbit/s when over UDP. The higher bitrate for the video streams sent over UDP was needed to make the transmission reliable. For the video transmission from the edge device to the end device, there were no similar requirements.

6.3 Bottlenecks and encountered issues

It was observed that corrupt video frames could cause the mixed reality system in the edge device to act unpredictably. For this reason, streaming with UDP will require the implementation of mechanisms to ensure reliable transmission and remove the risk of corrupt video frames being used by the edge device. The alternative, to use TCP, will require mechanisms to detect and prevent latency drift.

It was seen that the advanced streaming framework, used for H.264 streaming, added significant latency compared to using simpler modules to stream more bandwidth requiring JPEG image feeds. It is therefore advisable to develop a custom video streaming system, with a focus on responsiveness.

Having the edge device running a standard Linux distribution is preferred as most research into SLAM, computer vision and mixed reality are done using this platform, making all the necessary frameworks available. Developing the server with Qt can be recommended as it was shown to have good parallel processing support necessary to integrate the different frameworks. For a complete system that is to support generic Android devices, much of the development task will likely involve troubleshooting vendor-specific issues.

6.4 Future work

This section includes suggestions and recommendations for future studies in this field.

6.4.1 Hardware-acceleration

The strongest recommendation is to investigate how to accelerate the different processing using hardware chips, on both devices, to offload the processing from the main CPU. The investigation should include how to use hardware-accelerated codecs for video streaming, as well as how to use CUDA or a similar platform to speed up the tasks related to mixed reality. It could also be interesting to see how the performance would react on running the edge device software on a dedicated graphics processing platform such as Nvidia Jetson.

6.4.2 Use the IMU

While the scope for this thesis was limited to the use case described in Section 1.2, one key difference between the local and edge prototypes is the ability to measure how distances in the point cloud correspond to distances in the real world. ARCore can provide this by measuring distances using the device’s Inertial Measurement Unit [27]. If the end device is to send IMU measurements to the edge device, it would also be possible to use SLAM frameworks that take advantage of these measurements, such as Maplab [56].

6.4.3 Mixed mode

In the edge prototype, all calculations and graphics drawing was done on the edge device. With mixed mode, SLAM and other processing can be done on the edge device, with graphics drawing done on the end device. The response times from the edge device would be lowered, if only object coordinates are sent back to the end device. It would also allow higher image quality, as only lower resolution images would need to be transmitted, and higher resolutions could be kept on the end device to be displayed.
Bibliography


[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


[34] Yuozo Iano and Marcus Pereira Magri. “How to architect an IPTV system”. In: Eighth International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS ’07). IEEE, June 2007. DOI: 10.1109/wiamis.2007.49.


A Implementation Details

This appendix contains information about the implementation of the prototype. They were too detailed to be included in chapter 3 but are still included in this appendix, to the benefit of a person doing similar development.

A.1 End device

The application was developed using Android Studio, the integrated development environment provided by Google. The compilation and building stages were managed using the build automation system Gradle, which is the default method when developing Android applications using Android Studio.

The application requires Android 6.0, which was released along with Android API 23 in October 2015. In a survey [5] Google did during the first seven days of May 2019, 74.8% of all Android devices in use had Android 6.0 or higher and 38.7% had Android 8.0 or higher.

The version of GStreamer used was 1.14.4, released in October 2018. This version and the libraries it provides for Android only support the compiler GCC. Google removed support for GCC in the NDK with release 18, released in September 2018. This lead to NDK releases 17 being used. The older versions are provided by Google "for informational and archival use only" [6].

During development, it was observed that it was not possible to use the native code debugger included in NDK release 17 to get debugging information for code that interacted with the GStreamer library when running on the LG G6 Android device. No crashes were observed for the same source code running in the official Android emulator. This limitation meant that instead, all debugging had to be done using log messages. There were no observed compatibility issues between the surrounding build system and NDK release 17 when running the application in normal mode. In April 2019 GStreamer 1.16 was released, containing support for NDK release 18. However, at that time NDK release 18 had already been made obsolete and unsupported by Google. When GStreamer 1.16 was evaluated running on NDK release 18, it displayed the same crashes when starting the debug mode.
A.2 Edge Device

There was initially in the software stack support for an Apple computer running MacOS, to be used for the evaluation. After issues were encountered regarding the support amongst third-party open source libraries, where several lacked official support, the support for the MacOS platform had to be abandoned. All third party libraries used for the edge prototype on the edge device have strong support for Ubuntu 18.04. No other platform was evaluated or used for development.

During testing, it was found that there were major performance issues when running OpenCV compiled in a debug mode, during which additional debug symbols are included in the executable and some of the compiler’s code optimisations are turned off. For this reason, it was only evaluated when running in release mode configuration.

The software uses Qt’s signals and slots as its mechanism both for intra-thread as well as inter-thread communication between instances. The default mode for Qt’s signal and slot connections between threads are asynchronous connections, where signals are put in a queue and processed by the receiver in the order they arrived. This is very thread-safe but could cause delays for the edge prototype, as in the typical case only the last arrived mixed image should be processed. If a slot queue gets full, due to some task taking longer than expected, the consuming process has no information that it is processing outdated data and would keep processing it until it had finished processing the whole queue. Thus, a mechanism was created to ensure system-wide synchronisation of data so that every component always works on the last input. The components are connected twice, first synchronously and then asynchronously. The synchronous call is executed first, without any queuing, and puts a reference to the latest data at the first position in the receiving component’s queue. Thread safety is controlled manually with semaphores to ensure that two threads do not access and manipulate the same buffer.

This is a common way to handle synchronisation in multi-threaded systems, where threads are always running, and the class’s developer defines when to sleep and wake it. However, with Qt’s managed threads and the system’s loose coupling, it would be impractical to have these direct connections between classes, or an internal event loop. To overcome this limitation with Qt’s managed threads, an asynchronous call gets sent immediately after the synchronous call. The asynchronous call acts as a signal to the receiver that its buffer references have been set and can be processed. As Qt’s signals are sent in the order the connections were made, there is no risk that the asynchronous call arrives before the buffer has been set.

A.3 GStreamer

Issues with network packet loss and jitter were observed when streaming using UDP, where RTP packets that arrived late or out of order caused issues for the video decoder. To reduce these issues, a GStreamer element was added to the video receiver pipelines that reads the RTP packets’ headers and ensures that they are delivered in the correct order to the video decoder.

This element, called \textit{rtpjitterbuffer}, has the potential to affect the system’s responsiveness. It may induce a slight delay in the video receiver, as it waits at most 200 milliseconds until it has received all packets for a frame. The end-user selects if it should be enabled or not using the end device application’s start menu.

Severe issues were observed when streaming using the MediaCodec Library and edge device’s video receiver included the rtpjitterbuffer element. The received frame rates dropped by around 40%, and caused very choppy playback. It is unclear what caused that, if something was missing in the RTP packets’ headers or interpreted wrong by the receiver, and there was no time available to investigate it further.

On both devices the open source codecs \textit{x264enc} and \textit{avdec_h264} are used for the encoding and decoding of H.264 streams. Both these codecs are fully executed on the main CPU.
Changes were made from the default configuration of the encoders to improve the prototype’s responsiveness. The adapted configuration heavily prioritise encoding speed and low transmission latency over image quality and thus require higher bitrates to achieve the same image quality as had been achieved with the slower default mode. To reduce issues with unstable connections, the encoders are configured to make every frame a keyframe when streaming over UDP. It was observed that when running the encoders in default mode, with two seconds between every keyframe, a broken keyframe caused by packet loss or data corruption could mean that all thirty to sixty successive frames contained incorrect image data. This had the potential to leave the SLAM framework in an unpredictable state.

Substantial work was put into having GStreamer on Android be able to use the same accelerated codecs that are used by the MediaCodec Library described in 3.4.3, which offload encoding and decoding to support-chips.

The device platform vendor provides interfaces to the device’s hardware via the Android MediaCodec API, and GStreamer has support through the amc framework for including these codecs.

The Android device used for this study have processors from Qualcomm in the form of a Snapdragon 821, and that processor includes the hardware-accelerated H.264 encoder OMX.qcom.video.encoder.avc. While during the development of the prototypes, this codec could be included and accessed, there were too many issues preventing it from working properly. There was little documentation about what the codec wanted in the form of data and parameters, what data it produces, and what classes it could be connected to. The integration, therefore, had to be left unfinished.

There is support in GStreamer for accessing hardware-accelerated media codecs when it is running on Linux. This access is done through the Video Acceleration API. Similar to the codecs made available via MediaCodec the VA-API codecs have their specific configuration parameters and running modes. Despite initial plans to evaluate and integrate these codecs, there was not enough time available for this task.

The ahcsrc camera module is used for image feed input from the Android device’s camera. While ahcsrc’s support for features was limited, especially compared to the Camera2 API, and it only supported two image resolutions, 320x240 and 640x480, it fulfilled the requirement for this thesis.

During the evaluation, when streaming from the end device over UDP with GStreamer, significant issues were encountered that related to packet loss or packets out of order. It is unclear why that happened, and despite investigations, the root cause could not be found. Similar issues were observed with the MJPEG UDP stack when having the packet size set too low, with it being stable at around 2000 bytes per packet. It was possible to set the UDP packet size to a custom value by setting parameters in GStreamer’s rtph264pay element, but it did not improve the situation. While GStreamer’s rtjpjitterbuffer improved the transmission quality considerably, there were still some outstanding issues even with it enabled. This was noticed as some packets images were corrupted or arrived late, causing flickering. In-depth investigation of the video packets was out of scope for this thesis. These severe packet loss issues were not observed with the modified version of rtmp-rtsp-stream-client-java which did not have it enabled due to the drop in video frames. The opposite was found for the GStreamer UDP streaming, which had a lot of corrupted packets flickering but no reduction in video frame rate when running the device.

### A.4 Camera calibration

Digital cameras can differ when it comes what type of input sensor they have, how the input is processed, and what lens is mounted, and what focal length it has. All these attributes change the optical properties, changing how a scene is projected to the final image. When the things being portrayed move or the camera is moved the relative distance from the old to the
new position in the image will differ between different objects based on their distance to the camera as their old and new image locations. This issue is an often important aspect when it comes to computer vision. Especially for structure from motion and SLAM, it is crucial to know the relationship between different parts of the images and how the received image from a camera corresponds to the actual world it portrays [41].

An often applied method to solve this problem is to use a camera matrix, that when calculated for a camera can be applied to all images from that camera taken with the same focal length. In the simplest case, a simple projection matrix can be used, but more advanced systems often have matrices that also adjust for deformation and distortions in the projection. One common method to calculate the camera matrix is to use Zhang’s method [15] and take several photos from different angles of a flat reference image of a pattern, for example, a chessboard. Knowing the properties of the pattern and the parts corresponding positions in the input images, the corresponding image matrix can be deducted. The process of calculating the matrix is called camera calibration.

Having a system to automatise camera calibration removes potential bottlenecks for the SLAM algorithm and makes it possible to test using new phones without having to add their parameters manually. The prototype, therefore, has the ability to do camera calibration on the end device, with the processing offloaded to the edge device. The edge device uses OpenCV’s function for this, as they are well-tested and give predictable results [70]. OpenCV’s calibration mechanism uses Zhang’s method and multiple photos with different angles of a paper with a black and white chessboard pattern with nine rows and six columns. How camera calibration is done in the edge prototype can be seen in Figure A.1.

![Figure A.1: Edge prototype camera calibration as seen on the end device.](image)

While switching calibration mode on and off and selecting which images to use for the calibration are done by the user via the end device, all calculations are done on the edge server. When the edge device has set the session to calibration mode, the received image stream that is typically used for the mixed reality framework is instead sent to OpenCV’s functions to search for the chessboard pattern. When all nine times six squares have been found they are marked with circles and sent back to the end device, to identify that the images in view can be added to the calibration set. When images are tagged to be added, the locations of the squares in pixel coordinates are added to the list of observation. When the user leaves the calibration mode, all the locations are sent to OpenCV’s functions and if enough data is available will result in a camera calibration matrix. This calibration matrix is sent to the end device to be stored locally in the device’s settings file. When the client...
connects to the server the next time, the camera matrix is sent as configuration parameters and loaded by the mixed reality framework.

A.5 Functional testing

Due to the modular nature of the edge computing stack, several different methods could be developed to allow functional testing, where the module’s functionality could be verified independently of each other. As having a graphical user interface running on the edge device affects performance and adds dependencies on the surrounding desktop environment, these parts are disabled by default and has to be included using a Qt preprocessor directive during the software compilation phase.

- The edge device can run locally a mock client that acts as an end device and as video frames sends pictures stored on the server.
- The edge device can run a separate mock client similar to the one mentioned above locally, but where the simulated end device video comes from a webcam connected to the same server.
- The end device can instead of the camera use a video file as its input, to be transcoded to the chosen format and sent to the edge device.
- Video both from the end device and the corresponding mixed reality output graphics can be displayed locally on the edge device in a graphical user interface windows.
- The edge device can ignore the video stream from the end device, which can be disabled, and instead for that client use as input a camera connected to the same computer to allow functional testing of video transmission from the edge device to the end device.
- There is a limited functionality custom camera streaming module in the end device. It has very limited functionality due to work being shifted to the external video streaming frameworks instead.

The video playback mechanism on the end device feeds the decoded frames into the standard encoding and display processing pipeline, bypassing the camera. For the MediaCodec library, the video decoding and playback were implemented using the MediaCodec decoder, which then sends the data directly to its encoder internally. This means that the source image feed is not visible in the GUI. The LG G6’s MediaCodec implementation uses for H.264 material the hardware-accelerated decoder `omx.qcom.video.decoder.avc`. For the GStreamer module, decoding and playback were implemented using the `decodebin` element. This automatically selects and configures the appropriate elements for the selected material, and on the LG G6, using the same decoder as MediaCodec.

A.6 Modifications of ORB-SLAM2

When a client connects to the system, a new ORB-SLAM2 instance is set up with resources allocated, with the instance running in a Qt managed thread separated from the surrounding system. There can be several instances of ORB-SLAM2 running concurrently in the system, without them communicating with each other. As described in 2.5.3 ORB-SLAM2 is designed to use as much available CPU power as possible, and the SLAM instance itself spawn up to three new processing threads for different subsystems. The SLAM system’s tracker mechanism and the graphics drawing module were written to be frame-based, where their threads are idle when they’re not processing any data and are woken up when new data arrives. This may potentially cause slightly lower throughput speed but has the benefit of reducing the amount of processing overhead.
One hypothesis during the early phase of the thesis work was that the loop closing mechanism was not needed for the outlined use case, where the observed scenes are small, and the camera does not travel long distances. Therefore the loop closing mechanism and its separate execution thread were modified to make it possible to turn them off. During functional testing, it was however seen that having the loop closing mechanism disabled lead to a degrading of the tracker’s robustness. When the tracker lost its pose because of corrupted frames or other issues, it could lead to situations where it was completely unable to re-localise itself. The loop closing mechanism was, therefore kept.

To lower the total memory usage on the server, the system was changed to have the end device disconnects immediately when it is application is paused by the user. When the end device disconnects the SLAM instance’s threads are stopped, and all the resources tied to that instance are freed. When the end device connects a second time, a new fresh SLAM session is started. No data is saved locally on the edge device between sessions.

As ORB-SLAM2 was developed to run as a single instance, some changes also had to be made to the start-up phase. Before ORB-SLAM2 can be used, it has to initialise the Bag of Words data model used for converting the feature descriptors into natural language. This initialisation requires the loading of a large vocabulary set with pre-calculated classifiers taken from reference images. For this thesis, the reference set included with ORB-SLAM2 with the size 145 Mb was used, and the loading of it took around ten seconds on the laptop used for this thesis. The open-source community has written patches to have the vocabulary file in a binary representation to load the loading time, but when tested, this still added a couple of seconds of loading time. Instead, the management of Bag of Words was modified to have them in a pool, which is initialised, then the edge device is started. When an end device connects, it is assigned an available Bag of Words from the pool, and when the device disconnects the instance is restored and returned to the pool. This significantly lowered ORB-SLAM2’s start-up time when a device connects, which now is under one second. Tests were done to have the end devices concurrently share the same Bag of Words, which while allowing them to detect image features already seen by other devices more quickly, also caused unpredictable crashes when two devices simultaneously accessed the same area.

A.7 Graphics rendering

The rendering part of ORB-SLAM2 is very tightly integrated with the Pangolin library, which acts as a wrapper for OpenGL and takes the matrix and poses calculated from the SLAM system and converts them to 3D coordinates. On Lenovo laptop used for development, it was not possible to get headless rendering working with Pangolin. The computer’s graphics subsystem and Linux software had very limited support for EGL, which Pangolin uses for headless support. As the issues were in the hardware-near device driver stack, it was not possible to work around them. This lead to work being spent on creating a new graphics stack for the toolkit using the Qt framework’s 3D graphics framework Qt 3D, which has the benefit of being officially supported on multiple platforms including Windows, Android, MacOS and Linux with hardware acceleration on all of them. However, due to issues encountered when integrating it with ORB-SLAM2, with the conversion between different internal formats and data structures as well as encountered performance issues, this solution was eventually scrapped.

Instead, a new graphics stack was built using OSMesa. OSMesa, which stands for Off-Screen Mesa, is a part of the Mesa 3D Graphics Library which is an open-source implementation of the OpenGL API. OSMesa may use hardware-accelerated graphics rendering where available but has automatic silent fallback modes for software rendering. On the development platform software, rendering was used. The code in use was taken from Pangolin and wrappers were written to support OSMesa. The Pangolin support is still available and can be enabled using a Qt build directive, for testing on platforms where it is supported. The switch of li-
libraries also lead to the plans to utilise Qt’s cross-platform support to also support Mac OS and Windows, to be scrapped as getting the software linked to OSMesa on those platforms requires manual adjustments.

While Pangolin previously had support for Android that support was broken with when the library’s architecture was rewritten in early 2018. Android lacks support for OpenGL and instead supports OpenGL ES, which although shares similarities have a completely different API, and this for Pangolin would have meant rewriting code. Compiling OSMesa or the on-screen rendering Mesa 3D for Android requires AOSP and low-level hardware support from the vendor. As the platform was an LG G6, this was thus not investigated. Instead, time was spent developing a graphics stack. Qt 3D which has the benefit of being officially supported on multiple platforms including Windows, MacOS, Linux and Android with hardware acceleration on all of them. Due to issues encountered when integrating it with ORB-SLAM2, with the conversion between different internal formats and data structures as well as encountered performance issues, this solution was eventually scrapped.

A.8 Measurements

For all time measurements on the end device the Android system’s internal high-resolution clock SystemClock¹ was used, which provides the time since bootup in nanoseconds. For all measurements on the edge device, the timestamps were taken using the Qt framework’s high-resolution timer QElapsedTimer², which gives the time since it was started in nanoseconds. For both platforms, the measurements in nanoseconds were for presentational purposes rounded down to milliseconds.

The to analyse the image frames that were received by the decoding pipeline, GStreamer’s pad probe³ callback functions were used. These can be used to read the received image frame content without adding much overhead while it is being processed.

During these tests the GStreamer H.264 transmitter was modified by replacing the element that displays the transmitting content on the screen with an appsink⁴ element, which was set to parse the image for pixels with the colour bright pink. Similar to the pad probe, this element added little processing overhead when it was set to check for the colour pink. On the edge server functionality had to be added to the mixed reality framework also to have it detect the colour pink, to have it set the MJPEG metadata field when transmitting the content back to the device and also to remove the possibility that virtual reality content was added to the image. Also on the edge server, the overhead was extremely minor, as only a couple of pixels was enough to determine the status.

While efforts were spent on trying to get the MediaCodec transmitter to detect when specific colours or other signals were available in the source video file, the system under testing lacked such functionality. It was possible to attach listeners and retrieve pixel data also from the MediaCodec encoder by creating Android class instances that could be plugged into the system and receive data, but time was not available to get a working implementation.

Thus for the MediaCodec library, the round trip time was measured manually using flickering lights and the device’s screen being recorded by a high-frequency camera of type Rollei 530. The camera device records in 240 Hz and thus has a theoretical resolution of 4.16 milliseconds.

---

²https://doc.qt.io/qt-5.9/qelapsedtimer.html