UCLouvain

epl

École polytechnique de Louvain

A benchmarking platform for wireless virtual reality video compression and transmission

Author: **Martin Danhier**
Supervisor: **Benoît Macq**
Readers: **Jean-Didier Legat, Karim El Khoury, Corentin Damman, Antoine Legrand**
Academic year 2022–2023
Master [120] in Computer Science

Abstract

Through the reconciliation of mobility and graphics quality, wireless virtual reality has become a promising technology used in a multitude of applications, ranging from casual gaming and social applications to job training and architectural planning. However, the limitations of wireless networks introduce a diverse set of challenges, particularly the latency caused by the additional processing involved. To overcome these challenges, it is important to have a clear overview of the wireless virtual reality system as well as an in-depth understanding of the end-to-end video transmission pipeline. In this master thesis, a plug-and-play open-source wireless virtual reality benchmarking platform is developed, with the objective to be able to perform fine-grained measurements directly within a complete pipeline. This platform can be used to detect critical bottlenecks and anomalies within the system, and evaluate the performance of optimization techniques. In addition, the platform facilitates the choice of optimal parameters for a specific application depending on hardware configuration. The source code of this work is publicly available and should be seen as a first step to encourage the use of open-source benchmarking tools within the virtual reality community.

Contents

1 Acknowledgments **3**

2 Introduction **4**

3 Overview of virtual reality **6**
3.1 Definition and applications 6
3.2 Achieving VR 8
3.2.1 Contributing factors 8
3.2.2 Enabling technologies 10
3.3 Wireless virtual reality 14
3.3.1 Challenges 15
3.3.2 State of the art 15

4 Implementation of a benchmark wireless VR link **17**
4.1 Objectives 17
4.2 Planning and testing 19
4.2.1 Frame retrieval 19
4.2.2 Inter-process communication 20
4.2.3 Format conversion 20
4.2.4 Video encoding 22
4.2.5 Video transmission 23
4.2.6 Data transmission 26
4.2.7 Video decoding 29
4.2.8 Frame rendering 29
4.2.9 Presenting and interface with VR client 30
4.2.10 Measurements 30
4.3 Architecture 31
4.3.1 Common library 32
4.3.2 Driver 36
4.3.3 Server 38
4.3.4 Client 40

5 Results and discussion **43**
5.1 Testing setup 43
5.2 Delay analysis and optimization 44
5.2.1 Evaluation of optimization techniques 46
5.2.2 Additional observations 47
5.3 Bitrate analysis 48

6 Conclusion **50**

Chapter 1

Acknowledgments

This work wouldn't have been possible without the invaluable help of many people. In particular, I would like to express my sincere gratitude to:

- **Benoît Macq**, my supervisor, for his helpful direction and regular feedback.
- **Karim El Khoury**, teaching assistant at UCLouvain, for his extensive availability throughout the past year to answer my questions, and for daily meetings during the last few weeks.
- **Jean-Didier Legat** and **Lucas El Raghibi**, respectively professor and research assistant at UCLouvain, as well as **Pascal Pellegrin** and **Julien Verecken**, hardware engineers at intoPIX, who were also present during regular follow-up meetings and provided useful advice.
- **Jean-Didier Legat**, again, **Corentin Damman**, software engineer at intoPIX, and **Antoine Legrand**, UCLouvain PhD student, who have accepted to be members of the jury.
- **intoPIX**, and in particular the software team, for their welcome and helpful technical help.
- My friends **Clément Delzotti**, **Guillaume Nizet**, **Vincent Higginson** and **Yannis Fraselle**, as well as my family, for their precious feedback and support.

Chapter 2

Introduction

The virtual reality (VR) medium has become increasingly popular in recent years due to the high level of immersion it provides. Through the use of a head-mounted display (HMD), VR systems enable users to immerse themselves in virtual environments. Though originally developed for casual gaming, the technology has been used in a variety of applications, such as tourism [1, 2], education [3], industrial training [4], or healthcare [5, 6].

Among the various types of existing VR content, such as 360° videos or remote control, 3D-rendered applications pose the greatest challenge due to their demanding requirements in terms of latency and computational resources. In these applications, frames are rendered in real-time by a 3D game engine, taking into account the position and orientation of the user's head. These frames can then be displayed on the HMD using one of three different methods, each offering two of the three main guarantees: high-mobility, high-latency and high-quality graphics.

The first method is the stand-alone rendering of frames directly on the VR headset, guaranteeing high mobility and low-latency. However, the computational limitations of mobile hardware reduce the achievable level of graphics quality. The second method, wired offloading, defers frame rendering to a high-end server via a high-capacity cable. This allows for high-quality graphics and low latency, but the presence of a cable restrains the user's mobility. Lastly, wireless offloading instead uses wireless networks, such as Wi-Fi, to stream compressed rendered frames from the server to the headset. This third method, also referred to as wireless virtual reality (WVR), combines the graphics quality of wired offloading with the mobility of stand-alone rendering. However, the inherent bandwidth limitations and unreliability of wireless networks introduce variable levels of latency. Due to the physical limitations of the other approaches, WVR therefore appears as one of the most promising directions for the achievement of an ideal system that would

provide all three types of guarantee. For this reason, the study of optimization techniques for WVR is of great interest.

The realization of low-latency wireless VR presents two major challenges: video compression and wireless transmission. Video compression needs to achieve high frame rates and low bitrates while maintaining satisfactory image quality to justify the use of offloading. Wireless transmission, on the other hand, must be resilient to packet loss and jitter without adding significant delays. Several researchers have addressed these challenges by separately analyzing their impact on the system [7, 8, 9]. Additionally, some studies performed benchmarks of a complete WVR pipeline using commercial wireless VR streaming programs (referred to as WVR links) [10, 11], providing valuable insights into how these constraints interact with each other in real-world scenarios. However, the black-box nature of commercial links limits the precision of collected metrics and complicates the measurement setup.

In this master thesis, a plug-and-play wireless virtual reality benchmarking link was developed. This platform is able to collect fine-grained latency and image quality measurements of the execution of a real-world WVR application, allowing for the detection of major bottlenecks and anomalies in the pipeline. These measurements offer valuable insights for the development of optimization techniques. The proposed software was also released as open-source to allow for further improvements in future work, and enable its use for the evaluation of future research. The source code of this work is publicly available on GitHub[1].

This document is organized as follows. The first chapter provides context about the virtual reality field, the major challenges it faces and the state of the art of wireless virtual reality research. The second chapter then details the implementation process of the proposed benchmarking platform, including major design choices and the final architecture. Finally, the third chapter demonstrates the use of the benchmarking platform for optimization through the evaluation of a real-world WVR application.

[1] `https://github.com/martin-danhier/wireless-vr-bridge`

Chapter 3

Overview of virtual reality

This chapter intends to provide context about the virtual reality (VR) field. To begin with, the first sections defines the VR medium and the major types of contents that it includes. Then, the technical challenges that are faced in the realization of VR are explored, as well as the key enabling technologies that are used to address these issues. Finally, the field of wireless virtual reality, which is the primary focus of this master thesis, is discussed in greater depth.

3.1 Definition and applications

Virtual reality (VR) can be defined as a medium through which users experience the sense of being in a virtual environment. [12] This sense, known as *telepresence*, is typically achieved using a head-mounted display (HMD) to display stereoscopic images directly in front of the user's eyes. The HMD takes into account head position and orientation, while lenses enable the eyes to focus, replicating the visual experience of the virtual world [12, 13]. Additionally, input and movements are gathered, with a precision varying from head-only to full-body tracking depending on the desired level of interactivity. Most applications, however, only use head and hand tracking, using a combination of cameras and wireless motion controllers. Tracking cameras can either be integrated in the headset itself (inside-out tracking) or be located on the sides of the play area to cover it by triangulation (outside-in tracking) [14]. The complete set of devices contributing to the experience is called a VR system. There exist a large variety of VR systems on the consumer market, ranging from stand-alone mobile headsets such as the Meta Quest 2[1] and the Pico 4[2] to high-end systems requiring a powerful computer, such as the Valve Index[3] or

[1] https://www.meta.com/be/fr/quest/products/quest-2/
[2] https://www.picoxr.com/fr/products/pico4
[3] https://www.valvesoftware.com/fr/index/

the Varyo XR-3[4].

Virtual reality is a medium that supports a variety of content formats, with diverse use cases, advantages and constraints.

- **360° video**: audiovisual-only experience, such as 360° videos projected on a sphere around the user. The only required tracking data is thus the head rotation (3 degrees of freedom, or 3-DOF). This simple projection allows 360° videos to be represented in a regular video format and enables their straightforward integration in existing video editing software and distribution platforms. Additionally, the low amount of required tracking data makes 360° videos usable on smartphones with cheap head supports such as the (now discontinued) Google Cardboard[5], reducing the entry cost of VR and improving its accessibility. Moreover, the passive nature of videos allow them to be pre-loaded, removing the need for intensive real-time computations. However, due to the lack of positional tracking data, head translation can break the sense of telepresence and induce motion sickness, as the world appears to translate alongside the head. Also, since a video frame represents the complete projection sphere, most pixels are invisible. In consequence, extremely high resolutions are required for a high graphical fidelity. Due to its accessibility, 360° videos have become increasingly popular for virtual tourism [15, 1], education [3] and 3D movies or live streams [16, 17].

- **Teleoperation**: real-time 360° video live stream used in combination with specialized input devices that directly impact the received image. This content type is typically used to control robotic devices, such as drones [2], manufacturing machines [18] or surgical machines [19]. The use of VR in this domain makes the operation feel more natural. It also permits a better precision by scaling down human movements. For example, a robotic arm is able to move by half a millimeter when the pilot moves their arm by 10 centimeters. Moreover, VR allows a user to see an image in greater detail due to the large field of view it provides. Visual elements can also be superposed to the original video feed to convey useful information to the pilot [5], using augmented reality (AR) techniques. However, on top of the other drawbacks of 360° video mentioned previously, teleoperation video introduces an ultra-low-latency requirement due to the presence of input. Moreover, input must be extremely reliable, as a failure can have drastic consequences on human lives (in the case of surgery) or expensive equipment

[4]https://varjo.com/products/xr-3/

[5]https://web.archive.org/web/20191002000104/https://vr.google.com/cardboard/

(in the cases of manufacturing and drone piloting). Dedicated VR systems often need to be designed for this content type.

- **3D-rendered environment**: frames are rendered in real-time using a 3D game engine. This allows the use of rotational and positional head tracking (6 degrees of freedom, or 6-DOF) and a high level of interactivity with the virtual world, creating a high sense of telepresence, at the cost of a low-latency requirement. Also, unlike 360° videos, only the visible area is rendered, allowing for a better use of image resolution. However, rendering realistic frames requires high computational power and dedicated VR hardware, which increases the entry cost of 3D-rendered content. This format includes applications ranging from casual gaming and social applications [20, 21] to job training [4] and architectural projects visualizations [22].

This work solely focuses on 3D-rendered content, as it is the most commonly used type on consumer VR systems. The real-time requirement also introduce a large set of challenges that makes this content interesting to study, as discussed in the following sections.

3.2 Achieving VR

3.2.1 Contributing factors

As previously stated, virtual reality is primarily defined in terms of subjective user experience (UX). To be viable, a VR system must successfully manage to provide the user with a sense of telepresence. This means that the user should feel an illusion of being present in the virtual environment, rather than simply observing a video. The virtual objects should appear tangible, as if they were physically existing next to the user. Therefore, it is valuable to explore the technical elements that influence the user experience and the realization of a successful VR system. In particular, there are various factors observable by the user that directly affect telepresence and the overall user experience [23]:

- **Latency**: the delay between user actions (such as movements or button presses) and perceivable visual, auditory or sensory feedback. The most commonly used metric is the motion-to-photon latency.
- **Frame rate**: the frequency of displayed frames, typically measured in frames per second (FPS) or in inter-frame delay, the amount of time elapsed between two frames.
- **Resolution**: the dimensions in pixels of a video frame.

- **Binocular field of view** (FOV): the angle of stereoscopic image visible by both eyes, measured in degrees.
- **Rendered image quality**: the subjective aesthetical quality of rendered images, in terms of level of detail or realism.
- **Decoded image quality**: the amount of visible compression artefacts that can appear if the image is compressed between rendering and display. Common metrics include peak signal-to-noise ratio (PSNR) or structural similarity (SSIM).
- **Optical quality**: the amount of optical artefacts induced by the display and the lenses. Among others, these include chromatic aberrations and the *screen door* effect.
- **Tracking accuracy**: the amount of error between the tracking pose assumed by a displayed frame and the actual physical pose at the moment of display.
- **Mobility**: the extent of freedom the user has in their movements, including the size of the play area and the presence of any obstructive elements that can hinder movements, such as cables or bulky headsets.

These metrics play multiple roles in enhancing telepresence and improving the overall quality of the user experience. Latency, frame rate and tracking accuracy influence the fluidity of the experience, determining how effectively the virtual environment can realistically respond to user actions in real-time.

Moreover, resolution, field of view and rendered image quality impact the overall pixel density and level of graphical details. For example, high resolutions are particularly important due to the close proximity of the display to the user's eyes, which can magnify the appearance of pixels compared to a regular monitor. In addition to providing a more pleasant visual experience, pixel density also contributes to the legibility of text and reduces eye strain.

Lastly, decoded image quality, optical quality, tracking accuracy and mobility are essential for maintaining a stable and uninterrupted immersion, as visual artefacts and physical obstructions represent distracting events that can disrupt the user's focus and draw their attention away from the virtual world.

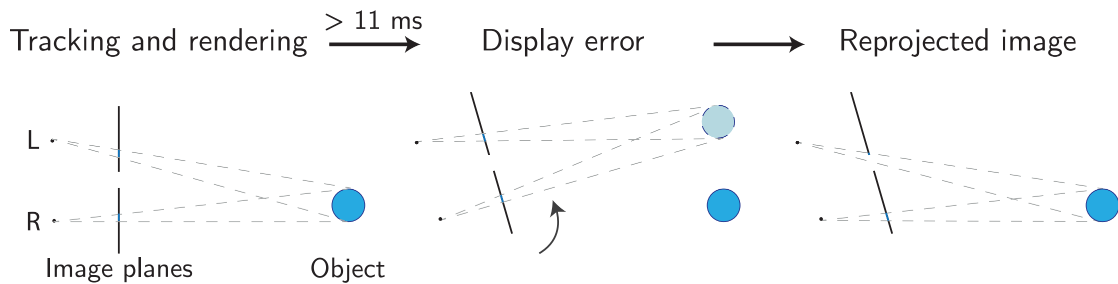

Figure 3.1: Example of misalignment caused by rendering latency during head rotation, and use of asynchronous reprojection to correct the error.

3.2.2 Enabling technologies

To optimize these factors and make the realization of functional VR systems possible, several technologies were developed over the last decade.

Asynchronous reprojection

One of the most critical challenges that needed to be addressed was the issue of low tracking accuracy for head movements. A certain amount of time is required to render a frame, typically at least 11 milliseconds at a frame rate of 90 frames per second. During this time, the user's head continues to move, resulting in a disparity between the head pose used for rendering and the actual pose at the moment of display. This poses a significant problem, as even the slightest offset in alignment makes objects appear at a different location than the intended one. For example, Figure 3.1 illustrates the misalignment that is created when rotating the head from right to left. At the beginning, the head is not moving, there is no alignment error and objects thus appear at the intended location. When rotating the head, the rendering pose is shifted to the right, as it is based on a past head orientation. This causes the objects to be displayed more on the right that they should. When the head stops rotating, the error disappears and objects appear at their correct location again. Overall, this makes the world appear to follow the movements of the head at a degree that depends on the rotation speed. This effect immediately breaks the illusion of telepresence, as objects in a realistic environment would never rotate alongside the head. Moreover, the unnatural motion of the environment and the lack of other referential than the VR image can cause severe motion sickness.

Asynchronous reprojection is a technique that aims to address this issue by reprojecting frames to their correct locations just before displaying them, thus preserving object alignement. Originally introduced by Oculus researchers as the

asynchronous time warping (ATW) algorithm for head rotation [24], variants have been developed over the years, such as asynchronous space warp (ASW)[6], application space warp (AppSW)[7] or motion smoothing[8]. In addition to correcting the rotational misalignment of the frames, these newer versions are able to use additional application data, such as a depth image, to compensate positional errors. Furthermore, they are also able to extrapolate new frames when the application is not able to maintain a consistent frame rate, consequently halving the required frame rate of the application.

All in all, the use of asynchronous reprojection greatly relaxes latency constraints, allowing the VR experience to be more resilient to high latency and delay variation. With these techniques, low-latency is no longer a requirement for a viable headset, and only impacts input responsiveness. This makes reprojection one of the most important enabling technologies of wireless virtual reality, which introduces high and unpredictable latencies due to the unreliable nature of wireless networks.

Rendering offloading

Rendering a VR frame is a computationally expensive task, as it involves higher resolutions and frame rates than standard gaming. For example, for the Meta Quest 2 VR system, a frame typically has a resolution of 2880×1584 pixels for a frame rate between 72 and 120 FPS, whereas regular gaming typically only requires 1920×1080 pixels and 60 FPS. Moreover, due to the increased immersion and the life-like scale provided by VR, the lack of details is more noticeable by the user. Hence, high quality meshes and textures are required, as well as a large overall density of objects in 3D scenes.

Consequently, a common technique consists in offloading the execution of the application to a high-end computer, called the server, allowing for high-quality graphics and expensive physics simulation. The offloading can either be realized to a local PC (edge computing) or to a remote cloud server (cloud gaming).

Multiple offloading modes exist, each providing a different set of constraints and being adapted to different applications:

- **Fully offloaded (wired)**: the headset only contains the HMD and head tracking sensors. All computations are performed on the server, which then transmits finished frames via a cable. This approach allows for the

[6] https://developer.oculus.com/blog/asynchronous-spacewarp/
[7] https://developer.oculus.com/blog/introducing-application-spacewarp/
[8] https://store.steampowered.com/news/app/250820/view/2898585530113853534

lowest latency due to the reliable nature of cables, and the high available computational power enables the highest possible image quality. The offloaded approach also permits the use of reliable outside-in tracking sensors, such as lighthouses or infra-red cameras. For these reasons, it was the first viable approach for 3D-rendered virtual reality and is still the method of choice for ultra high-resolution headsets such as the Varyo XR-3[9]. However, the requirement of an expensive computer greatly reduces accessibility for the general public. Moreover, cables also come with a large set of problems. First, they can hinder the user's movements and reduce the level of immersion. They also have a limited range, constraining the maximum size of the play area. Finally, and more importantly, cables represent a single point of failure (SPOF) that accumulates stress due to the intense motions inherent to VR applications, and is often the cause of breaking for a headset.

- **Partially offloaded (wired/wireless)**: additionally to the HMD and sensors, the headset contains additional hardware to run an operating system and executes all tasks of the VR environment, such as hand tracking and compositing. Only the execution and rendering of the application are then deferred to the server, allowing the transmission to optionally be realized wirelessly. The absence of a cable solves the problems mentioned above, at the cost of introducing new challenges due to the unreliable nature of wireless networks. Other mentioned drawbacks of the offloaded approach are still present, such as the requirement of an expensive computer. Wireless partial offloading also typically requires video compression, which can damage quality depending on the used codec.

- **Stand-alone**: no offloading. The headset is a complete mobile device, akin to a smartphone, which is able to execute the whole environment and application without the need of an external server. This makes the VR system more accessible and portable, at the cost of a much lower computational power, limiting the achievable level of graphical detail. Most stand-alone systems are also able to function as a partially offloaded system, enabling users to choose the preferred method based on the application at hand.

As mentioned above and summarized in Table 3.1, stand-alone systems have the benefit of being able to switch between partial offloading and stand-alone modes of operation depending on the requirements of the application in terms of mobility, graphical quality and latency. In particular, each of the three solutions provides guarantees for two of the three criterion: wired provides quality and low-latency,

[9] `https://varjo.com/products/xr-3/`

Offloading	Latency	Graphics	Mobility
Full	Very low	Very high	Constrained
Partial wired	Very low	Very high	Constrained
Partial wireless	Variable	High	Unconstrained
Stand-alone	Very low	Low	Unconstrained

Table 3.1: Properties of offloading modes in terms of latency, graphics quality and mobility.

wireless provides quality and mobility, and stand-alone provides mobility and low-latency. The third axis remains best-effort and should be optimized as much as possible, the ideal being a system achieving all three types of guarantees.

Virtual reality software stack

Despite the existence of a wide range of virtual reality systems, the industry itself remains relatively small with a limited user base. To reduce development costs and avoid complicated ports from one system to another, effort has been made to streamline the creation of cross-platform VR applications. This involves dividing the software into three primary layers, and using standardized application programming interfaces (APIs) for cross-platform communication between layers.

The **application** is the actual VR game or software of interest being executed by the user. Frames are typically rendered with a 3D rendering engine powered by a graphics library such as Vulkan[10], Direct3D[11] or OpenGL[12]. The OpenXR[13] API developed by the Khronos Group allows the application to interface with underlying layers in a cross-platform manner.

The **compositor** handles the VR environment itself. The main role of this layer is compositing, i.e performing additional processing on the frames received from the application, such as the addition of menus or play area boundaries. A compositor also typically uses algorithms to enhance user experience, such as asynchronous reprojection or input prediction.

The **VR system driver** handles the low level and device-dependent task of presenting received frames to the HMD and calculating tracking poses. Interface

[10] https://www.vulkan.org/
[11] https://learn.microsoft.com/en-us/windows/win32/direct3d
[12] https://www.khronos.org/opengl/
[13] https://www.khronos.org/openxr/

with compositors is typically realized through a compositor-specific driver API.

3.3 Wireless virtual reality

In the previous sections, virtual reality was defined and the enabling technologies that address the major challenges in its realization were discussed. As detailed above, stand-alone VR systems represent one of the most promising type of VR hardware, due to their ability to optionally function as offloaded systems depending on the application at hand. In particular, a stand-alone VR system is able to function in three main modes of operation: stand-alone, wired offloading and wireless offloading. Each mode provides two out of the three main guarantees, which are low-latency, graphical quality and mobility. The third axis is best-effort, and an ideal system would achieve all three types of guarantees. Attention should therefore be put in the optimization of this third axis.

By essence, mobility in a wired system will always be more constrained than in a wireless system. Also, graphical quality in a mobile stand-alone system will likely always be lower than a desktop computer with no space, weight or power limitation. Moreover, with the use of low-latency video compression codecs such as HEVC [25] and dedicated Wi-Fi 6 routers, wireless VR systems already reach acceptable levels of latency to satisfy the base requirements for telepresence. Hence, optimizing the latency and decoded image quality of wireless offloaded VR appears to be the most promising direction to achieve a system providing all types of guarantees. For this reason, this work primarily focuses on wireless partially offloaded 3D-rendered virtual reality content, that will be referred to as wireless virtual reality (WVR) for simplicity.

Wireless virtual reality can be achieved through the use of a special program, referred to as a **WVR link** in this document, that sits between the server and client compositors. In the server-to-client (downlink) direction, the link streams audio and video frames, as well as haptics events (such as controller vibrations). Additionally, tracking and input data are transferred in the client-to-server (uplink) direction. Examples of links include Meta Airlink[14], AirLightVR[15] (ALVR) and Virtual Desktop[16].

[14]`https://www.meta.com/fr-fr/help/quest/articles/headsets-and-accessories/oculus-link/connect-with-air-link/`

[15]`https://github.com/alvr-org/ALVR`

[16]`https://www.vrdesktop.net/`

3.3.1 Challenges

On top of regular VR challenges, such as high computational cost for rendering, the implementation of a WVR link faces additional challenges caused by the limitations of wireless networks.

- **Limited packet size**: the maximal transmission unit (MTU) of IP networks is usually limited to approximately 1500 bytes, creating the need for packetization and depacketization, which can introduce additional overhead.
- **Limited bandwidth**: wireless networks typically provide a much lower bandwidth than cables. For example, an HDMI 2.1 cable can reach a bandwidth up-to 48 Gbit/s [26]. In contrast, a Wi-Fi 6 router theoretically provides around 1.5 Gbit/s, though on average, most users are only able to reach a bandwidth of 92 Mbit/s [27]. Additionally, a typical raw VR RGBA stream of 90FPS at a resolution of 2880×1584 pixels requires a throughput of 13.138 Gbps. Therefore, this limitation makes impossible the raw transmission of video frames, and introduces the need for video compression.
- **Packet loss**: wireless networks are susceptible to high packet loss rates due to various factors, including multipath interference, collisions, or weak signal strength [28]. These packet losses can have detrimental effects on the transmission of frame data, resulting in image degradation or even complete decoding failure. Various mechanisms can be used to mitigate these issues. For example, forward error correction (FEC) can be used to reconstruct lost packets by introducing redundant data to transmitted frames. Another possible solution is the use of a network protocol that is able to retransmit lost packets.
- **Transmission delay**: the time taken by packets to arrive at destination can vary significantly. This introduces additional latency as well as out-of-order arrival of packets, causing jitter. Consequently, mechanisms need to be implemented to reorder packets on arrival.

In summary, the limitations of wireless networks create the need for additional processing steps both before and after frame transmission. These processing tasks introduce additional latency due to the time required for their execution. It is thus valuable to investigate methods for optimizing these processing steps.

3.3.2 State of the art

Several researchers have addressed this challenge by isolating one constraint and analyzing its impact on the system. Žádník et al. [7] have explored techniques and

limitations of ultra-low latency video compression using standardized video codecs such as HEVC [25] and VVC [29]. They suggest that, in wireless scenarios, the unpredictability of wireless networks rather than the computational complexity of video compression is the main latency bottleneck. Tan et al. [8] have proposed three methods to minimize motion-feedback latency and jitter in bidirectional streams over WiFi in the context of WVR, most notably by limiting the aggregation size of downlink frame data transmission.

Given that these constraints also impact one another, it is also interesting to analyze these constraints in a complete WVR pipeline. This is usually done by running benchmarks on a standalone headset connected wirelessly to a rendering server using a link program that handles both video compression and streaming. Zhao et al. [10] have done a series of network measurements on cloud-VR using the Virtual Desktop[17] link. Salehi et al. [11] have performed similar measurements in the context of edge computing, using the open source link AirLightVR[18] (ALVR). In both cases, the researchers gathered their measurements by sniffing packets using Wireshark[19] and analyzing the network trace.

As pre-existing links are primarily designed for consumer use, it can be unpractical to collect detailed measurements with them. In the next chapter, a wireless virtual reality benchmarking platform is proposed. This platform was designed with benchmarking in mind, and is able to collect fine-grained measurements directly within a complete pipeline.

[17] `https://www.vrdesktop.net/`
[18] `https://github.com/alvr-org/ALVR`
[19] `https://www.wireshark.org/`

Chapter 4

Implementation of a benchmark wireless VR link

The implementation of a wireless VR link represented a significant task, facing numerous challenges ranging from protocol and library choices to architecture design. To maintain a coherent architecture, save development time and find optimal solutions, a structured approach and rigorous planning was required. This chapter aims to detail the implementation process followed during the realization of this work. First, an overview of the objectives and major implementation steps is given. Then, the main challenges that had to be solved are discussed, and the architecture design is explained, before detailing the technical aspects of the implementation itself.

4.1 Objectives

Before discussing technical implementation details, it was important to clearly state the objectives that the final platform should satisfy. These objectives, which are listed below, were then taken into consideration throughout the whole implementation, as they directly impacted the decisions that were made.

- **Functional WVR link**: the platform should be usable as a basic WVR link. It should thus be able to transmit video frames rendered on a powerful machine – called the server – to a standalone and less powerful Android WVR headset – called the client. In the opposite direction, the client should stream tracking information, allowing the user to look around and move in the 3D scene. However, audio streaming and controller support are left as future work. Also, this link should only exist as a low-level cable substitute, and should not handle the actual rendering. In other words, the implementation

should simply receive frames from a VR compositor, as if it was a regular tethered headset.

- **Fine-grained measurements**: the platform should be able to collect measurements in various important parts of the pipeline, including timestamps, network data and frame captures. These measurements should have a minimum runtime cost to allow for realistic results.
- **Easy to use**: the system should allow for easy collection of measurements over a large variety of configurations.
- **Extensible**: the implementation should be designed to allow the straightforward definition of alternative implementations in future works.
- **Open-source**: the source code of the benchmarking platform should be made publicly accessible to enable extension and use by the community.
- **Simple**: as the realization of WVR link represents a significant amount of work, the implementation should be kept simple in order to be feasible by a single developer.

In order to structure development planning and achieve these objectives, the implementation process was divided into several major milestones.

1. **Planning and testing**: the various issues that were faced in the realization of a wireless link were evaluated and solved. In particular, solutions had to be found for frame retrieval and presentation, tracking collection, video and data transportation. The chosen solutions directly influenced the direction taken for the architecture.
2. **Architecture design**: the software architecture was designed, including the overall structure of threads and classes, as well as the execution processes.
3. **Implementation of a basic WVR link**: this architecture was implemented in order to achieve a basic video-only link.
4. **Implementation of a measurement system**: the measurement system was then introduced, and a Python module for data processing was developed.
5. **Optimization**: using measurements collected in a real-world VR application, major implementation issues were localized and addressed.

4.2 Planning and testing

Before being able to design the architecture of the software, it was first necessary to list all major sub-problems that had to be solved. For each one, the issue was discussed, possible solutions were investigated and one of the alternatives was chosen according to the project objectives. This process is described in the next sections, following the path taken by a frame from the client application to the client HMD.

4.2.1 Frame retrieval

As explained in the previous chapters, the VR environment is typically handled by a VR compositor, which acts a compatibility layer with applications and performs additional processing on rendered frames. In order to be compatible with existing VR applications, the WVR link should therefore retrieve frames from such a compositor. This is typically achieved through the creation of a driver using a compositor-specific API.

SteamVR appeared as the most promising choice of compositor, as it allows for the creation of drivers with the OpenVR API, enabling the development of custom DIY headsets, or it the present case, custom WVR links. In contrast, most other compositors only support devices from a specific brand. Moreover, numerous open-source examples of OpenVR drivers are available, simplifying the development process.

An OpenVR driver is a shared C++ library that is loaded by the SteamVR compositor on launch. It comprises several components:

- A **server driver** that handles initialization and event handling.
- A **tracked device driver** per connected device, which typically include the headset and motion controllers. Each device driver has special capabilities depending on the nature of the device. In the present case, the headset driver is able to function as a virtual display, enabling the retrieval of rendered frames from the CPU instead of their direct transfer to a connected wired headset.

However, various limitations are caused by the shared library nature of OpenVR drivers. First, it needs to be loaded by SteamVR, making debugging difficult, as only debug logs and crash dumps are available. Moreover, due to a SteamVR security measure, the driver is automatically disabled when the process crashes. It

then needs to be manually enabled again, slowing down development. Furthermore, the driver initialization is synchronous: OpenVR expects the immediate creation of a device driver with the correct specifications. In a WVR link, however, connections are asynchronous: the server starts disconnected and a client connects at a later point. Finally, the whole SteamVR process has to be restarted in order to restart the VR environment, preventing iterative executions with varying configurations.

To avoid these limitations, the driver was designed as a separate process from the server. The driver only handles direct interface with SteamVR, and all computations, such as video encoding and transmission, are deferred to the server process, which is a regular executable. This approach has multiple benefits. First, it allows the VR environment to be restarted automatically without needing to reconnect with the client, simplifying benchmarking. Moreover, the driver is able to only activate when the server process is running, enabling the use of SteamVR with other installed VR systems and links.

4.2.2 Inter-process communication

This division of the server into two processes introduced a new problem to solve: inter-process communication (IPC). Both the server and the driver should be able to detect if the other is running, as well as exchange information, including settings, states, frames, tracking data and measurements. This transfer should be as low-latency as possible.

IPC can be realized using multiple techniques, such as pipes, sockets or shared memory. Venkataraman et al. [30] identified shared memory as the fastest form of IPC, due to the lack of unnecessary copies, achieving high throughput and low latency. However, synchronization needs to be manually performed by the applications using mutexes or semaphores.

Message passing can also be used to signal the other process when a specific operation has been performed on the shared memory. This allows the receiver to wait for a specific event and execute dedicated handling code. For example, a dedicated server thread is able to wait on the "New frame" event, and begin processing with minimal latency.

4.2.3 Format conversion

The image received from SteamVR is a packed RGBA image, allocated on the GPU. In this format, each pixel is represented by a 32-bit value, composed of 8-bit red, green, blue and alpha channels, as depicted by Figure 4.1. Padding can also

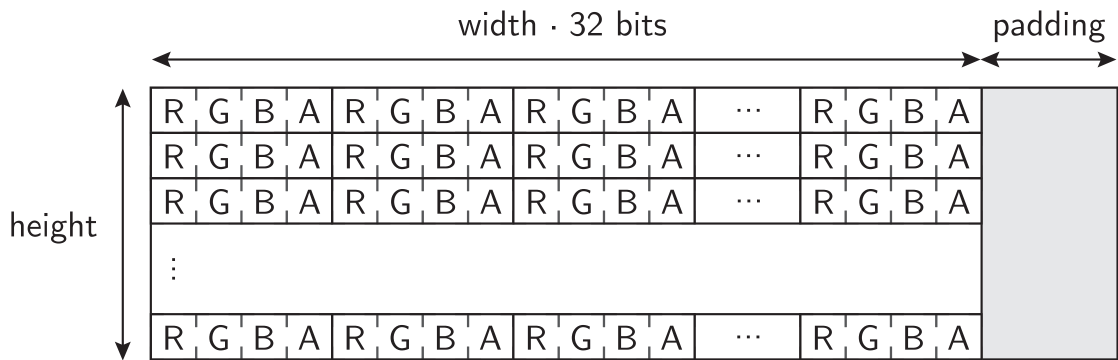

Figure 4.1: Packed 32 bits-per-pixel RGBA image format.

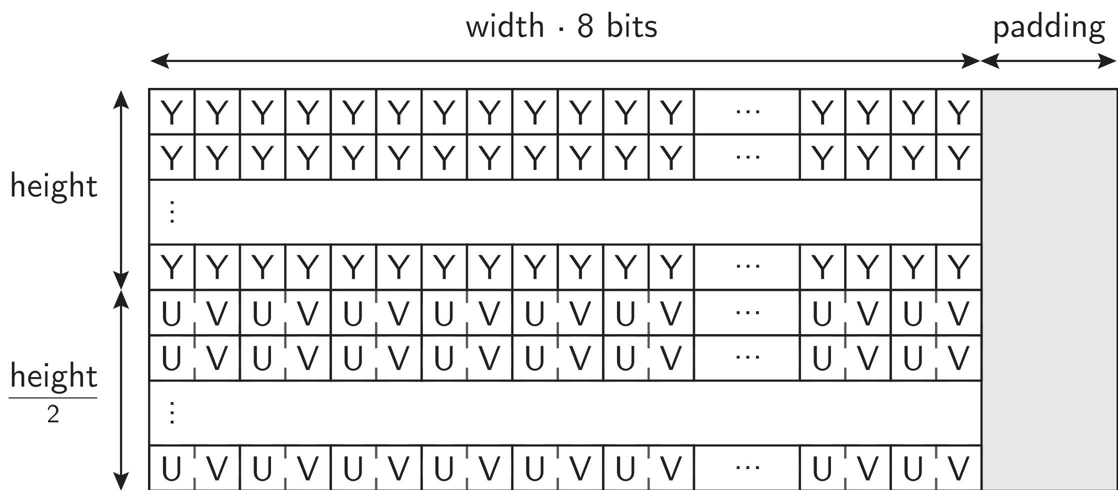

Figure 4.2: Semi-planar 12 bits-per-pixel NV12 image format, composed of a full-size 8 bits-per-pixel Y plane and a quarter-size 16 bits-per-pixel UV plane.

be inserted at the end of each line to align them on cache blocks and improve performance. However, most video encoders, including Nvidia NVENC, require the source image to be in a YUV format. This means that each pixel is instead represented using a combination of luminance (Y) and chrominance (U and V) values. Such a format allows for a higher compression of U and V values, as human vision is more sensible to luminance than chrominance.

In this work, the NV12 format was used, as described by the DirectX Graphics Infrastructure (DXGI) documentation[1]. The NV12 format, which is detailed in Figure 4.2, is a semi-planar 12 bits-per-pixel YUV 4:2:0 format, meaning that the image is composed of two planes:

- A **luma (Y) plane**, the luminance (grayscale) part of the image. Each pixel is a 8-bit Y value.
- A **chroma (UV) plane**, the chrominance (color) part of the image, downsampled to half the width and half the height of the original image. Each pixel is a 16-bit value, composed of interleaved 8-bit U and V values.

Before being sent to the encoder, the image thus needs to be converted. To do this, a compute shader can be used to directly perform the conversion on the GPU, which is more efficient due to increased parallelism.

4.2.4 Video encoding

As previously discussed, the low bandwidths of wireless networks make the raw transmission of video frames impossible. This creates the need for high-efficiency video compression that is able to transmit high-quality and high-frame-rate video content while maintaining low-latency and low-bandwidth consumption. Therefore, the selection of a suitable video codec is crucial. Moreover, the execution of the encoder and decoder plays a important role in the overall latency, data throughput and error tolerance requirements of the system. To make informed decisions regarding video codecs, it is thus valuable to compare them by running benchmarks with varying parameters. Hence, the benchmarking platform should support as many video codecs as possible, as well as provide intuitive means to incorporate support for new codecs as future work.

The video compression landscape comprises a wide range of libraries. Some libraries, such as x264[2], provide dedicated encoders fine-tuned for a specific codec,

[1] `https://learn.microsoft.com/en-us/windows/win32/api/dxgiformat/ne-dxgiformat-dxgi_format`

[2] `https://www.videolan.org/developers/x264.html`

offering intuitive and optimized use. On the other hand, there are generic libraries, such as FFMPEG AVCodec[3], which provide universal support to a variety of encoders and decoders through an unified interface. Due to the requirements of simplicity and wide codec support, a generic library appeared as the best solution for the benchmarking platform.

However, despite providing access to a number of encoders, some are not supported. Hence, the benchmarking platform was also designed with a plug-in system that enables the integration of additional dedicated codecs. The structure of this system is detailed in a further section.

4.2.5 Video transmission

Another issue that had to be discussed was the transmission of compressed video stream. Even though video encoding allows for a significant reduction in bandwidth requirements, the other network constraints need to be considered. Due to the limited maximal packet size, compressed video frames need to be split into small fragments. These fragments may arrive out-of-order, or even be lost entirely. The large amount of transferred data also increases the risks of congestion, which can further deteriorate the network capabilities. This creates the need for a protocol that is able to reorder packets, recover lost packets and perform congestion control. A variety of protocols can be used for this purpose, the two main candidates being TCP and RTP. The following sections detail their principles and advantages, after which the final choice of TCP for the benchmarking platform is explained.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) [31] is a protocol designed for the reliable transfer of a stream of data. It guarantees the ordered reception of packets through the use of sequence numbers. Additionally, received data is acknowledged (ACK) by the receiver, allowing for the retransmission of lost packets by the sender. Moreover, TCP provides congestion control to avoid network overloading and minimize packet loss. Due to its widespread use for the last 30 years and its built-in implementation in most operating systems, TCP represents the simplest and most stable option for reliable data transfer. However, it can introduce latency, as the strictly ordered reception of packets prevents a packet from being submitted until all preceding ones are received. This can prove to be an issue for low-latency video streaming, as in some cases, it is preferable to drop an old frame to avoid delaying more recent ones. This issue can be avoided to an extent by preventively dropping aged frames,

[3] `https://wiki.videolan.org/Documentation:Modules/avcodec/`

or by using a dedicated high bandwidth router to minimize packet loss and allow fast data retransmission.

Real-Time Protocol (RTP)

The Real-Time Protocol (RTP) [32] is an application-layer protocol designed for audio and video streaming. It is typically used over the User Datagram Protocol (UDP), a simple and unreliable packet-based transport protocol [33]. With RTP, frames are divided into smaller packets composed of a RTP header and a codec-specific payload. Most notably, the header carries a timestamp and a sequence number to allow the reordering of packet and synchronization. The payload section is typically defined separately for each codec in a standard describing the way frames should be divided and reconstructed. For example, the payload format for HEVC video is defined in RFC 7798 [34]. RTP is typically used in combination with the Real-Time Control Protocol (RTCP) which is used to provide feedback on the quality of reception for rate control.

RTP allows for an ordered, synchronized and low-latency reception of frames with rate control through RTCP. However, it is vulnerable to packet losses that can greatly deteriorate the quality of the received frame, or even completely prevent its decoding, depending on the error resilience of the used codec. This issue is particularly problematic in lossy wireless networks. In order to mitigate this issue, a retransmission system for RTP was proposed [35], but other techniques are possible, such as forward error correction (FEC).

Protocol choice

In order to further inform the protocol decision, existing WVR links were investigated using network traces captured using the Wireshark[4] application. ALVR provides the user with a choice between TCP, UDP or throttled UDP. Meta Airlink uses a simple TCP transmission, while still achieving telepresence and a high quality of experience. However, delay can accumulate on wireless routers with a limited bandwidth.

Moreover, tests were conducted using a simple custom implementation of RTP. Though efficient locally, the amount of frames drops on a wireless network prevented any frame from being decoded, due to the loss of essential codec headers. The performances of this custom implementation could be improved through the addition of a network shaper that would delay packets to maintain a maximum bitrate and avoid congestion. RTCP rate control and retransmission of vital headers

[4] `https://www.wireshark.org/`

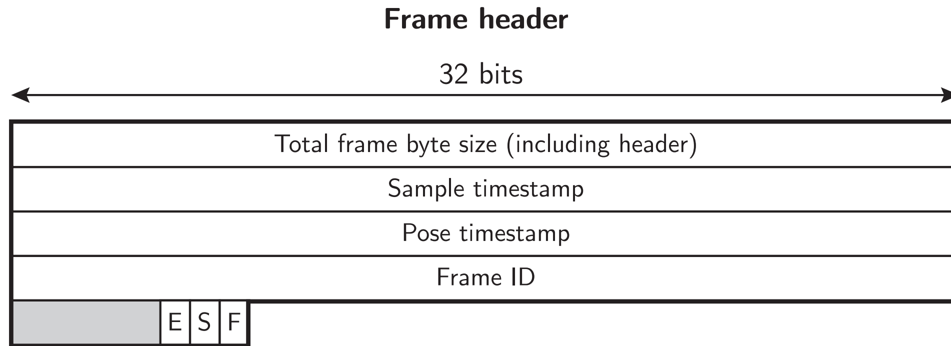

Figure 4.3: Frame header, carries the total size for stream splitting and other synchronization information. A flag byte can be used to mark the frame as end-of-stream (E), save (S) or end-of-frame (F).

should also improve the reliability of the implementation.

Another downside of RTP is that frame packetization is codec-dependant, as each payload format is defined separately. This increases the complexity of the support of a wide variety of codecs. TCP, however, only requires a simple header containing the size and timestamp of each frame. The reliable and ordered delivery of packets allows the frame data to be sent without further processing.

Hence, TCP was chosen for the benchmarking platform, as it provides satisfactory performances on a dedicated local network for a much simpler implementation than RTP. The finalization of the RTP implementation is thus left as future work, and would allow the use of the benchmarking platform for protocol comparisons.

Packet format

TCP is a stream-based protocol. From the above layers, transmitted data can be interpreted as a continuous stream of data, with no clear division between packet. This enables efficient transfer of frame data without the need for complex packetization and fragmentation. However, a mechanism has to be implemented to split the different incoming frames in the stream. For this purpose, a simple frame header was designed, depicted in Figure 4.3.

In addition to the frame size, the header also carries additional information. Firstly, various timestamps are provided, using a synchronized 90 KHz clock as described in the RTP standard. In particular, the sample timestamp indicates the time at which the frame was submitted to the driver, and the pose timestamp indicates the estimated display time of the tracking pose used for rendering. This

allows for client-side synchronization and frame reprojection, as is detailed in a later section. The frame identifier is also provided in order to simplify measurement collection. Finally, boolean flags can be applied to the header. The *end-of-stream* flag marks the last frame of the video stream, after which the client should stop rendering. The *save* flag indicates that the client should save the presented frame and send it back to the server. This is used for image quality measurements, such as PSNR or ISSN calculations. The *end-of-frame* flag can be used to mark the end of a frame if the frame is divided into multiple fragments.

4.2.6 Data transmission

Besides the transmission of video and audio frame data, four other data streams are present in a standard WVR link:

- **Control stream**: server discovery, session establishment and shutdown, benchmark configuration and transfer of measurements. Needs to be reliable and ordered, but doesn't have strong latency requirements.
- **Tracking stream**: stream of head and hand poses, i.e orientation, position and field of view. Needs to be low-latency, but packet loss and reordering are tolerated.
- **Input stream**: stream of button presses. Needs to be low-latency and reliable, but reordering is tolerable.
- **Haptics stream**: stream of head and controller vibrations. Needs to be low-latency and reliable, but reordering is tolerable.

In the basic video-only implementation, only control and tracking streams are required. However, the other streams were also considered in the design phase to simplify their eventual implementation as future work.

Unlike video frames, these streams only transmit a small amount of data, which simplified the choice of transport layer protocol. TCP appeared to be the best choice for the control, input and haptics streams, due to its reliability. For the tracking stream, UDP is sufficient, as tracking packets are often abundant and can afford to be dropped. Moreover, tracking packets are timestamped, allowing the filtering of out-of-order packets.

An application layer protocol also had to be designed for these data-based streams. This protocol should allow the client to discover available servers on the local network, establish a session, and allow the exchange of actual WVR

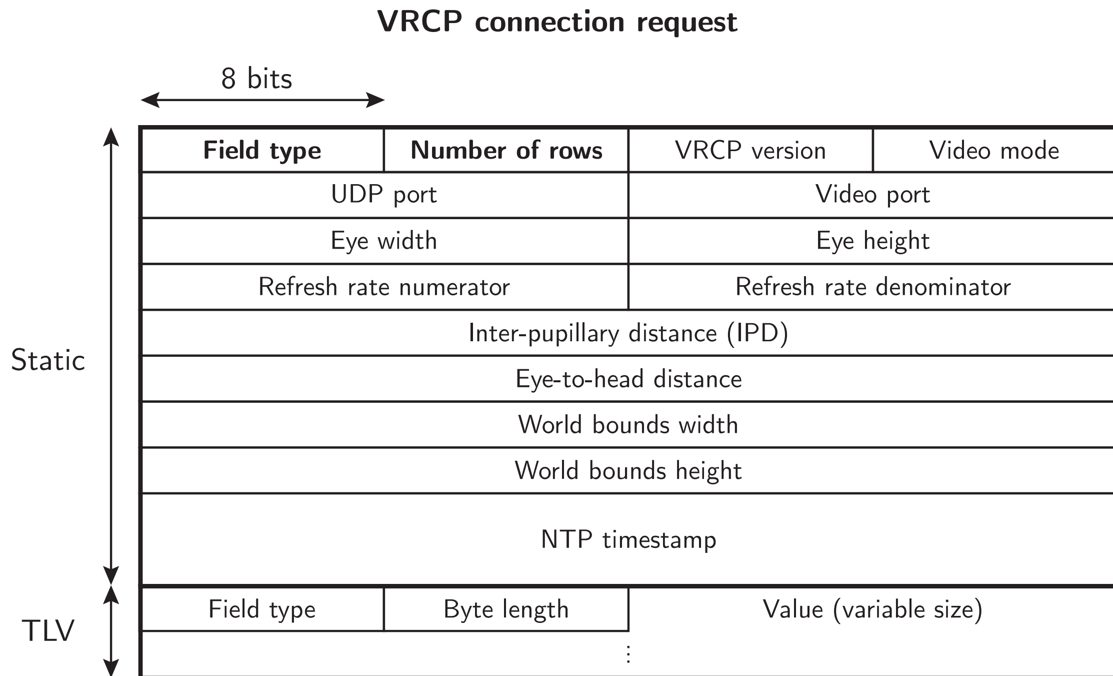

Figure 4.4: VRCP connection request structure. Carries the specification, configuration and capabilities of a client VR system. The start of the packet is a static structure determined by the **field type**. It is followed by several 32-bits-aligned type-length-value (TLV) fields carrying data of variable length, including system name, manufacturer name and the list of supported video codecs. The **number of rows** field indicates the total number of 32-bit rows in the packet, including TLV fields.

application data such as input or tracking. For this purpose, a simple binary protocol was designed, internally named **Virtual Reality Control Protocol (VRCP)**.

VRCP packet format

The VRCP packet format was designed to be compact, efficiently processed, and interchangeably usable over both TCP and UDP. A packet is composed of a succession of independent fields, allowing easy depacketization in TCP, where there is no clear division between packets. In UDP, it also enables to combine multiple fields in a single datagram, allowing for a better network utilization. Each field begins with a common two-byte header, indicating the field type and length. Each field type is associated with a compact and static binary structure, simplifying processing. Field length is expressed in terms of 4-bytes rows, allowing fields of up

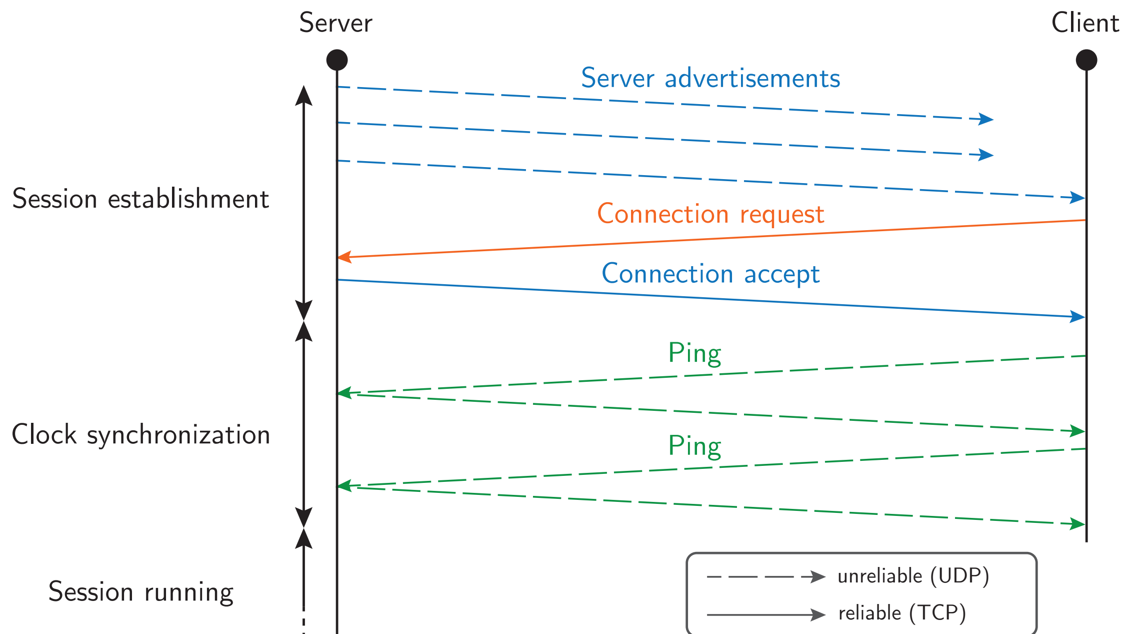

Figure 4.5: VRCP connection process. The server broadcasts advertisements until a client sends a connection request. If it is compatible, the server accepts the connection, otherwise rejects it with an error. The connection phase is followed by a synchronization phase to mitigate clock error.

to 1020 bytes. If the length of a field isn't a multiple of 4, padding is inserted at the end. In rare cases, such as strings or lists, field values cannot be expressed with a static structure and are instead represented as type-length-value (TLV). As an example, the VRCP connection request packet is detailed in Figure 4.4. It contains various information about the client hardware specifications (such as resolution, refresh rate or play area size), configuration (such as video protocol, socket ports and synchronization timestamp) and capabilities (such as a list of supported video codecs).

Session establishment

Besides being a format for VR application data packets, VRCP is also used for the establishment of the WVR session. Upon launch, the server repeatedly broadcasts a *server advertisement* on all local networks and on a known port. This message contains the ports of the different data streams mentioned above, as well as server

capabilities, such as the supported video codecs. If a client receives a server advertisement, it can request the creation of a WVR session by sending a *connection request*. Similarly, this request contains the specifications and capabilities of the VR application, including the frame rate, resolution, inter-pupillary distance (IPD) and supported codecs. The server then checks its compatibility with the client, and if compatible, confirms the session with a *connection accept* message. Otherwise, a *connection reject* is sent with an error. Once this connection phase is completed, parties can exchange VRCP messages, either reliably (over TCP) or unreliably (over UDP). The connection process is illustrated in Figure 4.5.

If a dedicated router is used for wireless transmission, the client may be disconnected from the Internet, preventing it from accessing NTP servers for clock synchronization. As a result, after several days of disconnection, the client system clock can differ from the server's by a margin of up to 15 minutes. To correct this error and increase the precision of measurements, the connection phase is succeeded by a synchronization phase. The client repeatedly sends ping requests over UDP to the server, which responds as quickly as possible with a timestamp of the current time. Upon reception of the response, the client is able to compute the round-trip-time (RTT) of the packet and estimate the time at which the server timestamp was measured. With this information, it is able to measure and correct the clock error between the server and client.

4.2.7 Video decoding

Once the video frame is received by the client, it needs to be decoded. Similarly to the encoding, a generic library can be used to easily support a large variety of codecs. However, tests showed that AVCodec doesn't provide support for Android hardware decoders, resulting in slow decoding times. Instead, the built-in Android MediaCodec[5] library was used.

4.2.8 Frame rendering

The decoded NV12 frame then needs to be converted back into RGBA format and displayed on the screen. This is easily achieved using the OpenGL graphics library. The image is copied in a GPU texture then projected on a full-screen quad, converting each pixel into RGBA at the same time. The downsampling of the UV plane can be compensated by the use of texture samplers. With this approach, the color of a rendered pixel is chosen based on the value of the nearest pixel of the texture, using relative coordinates.

[5] `https://developer.android.com/reference/android/media/MediaCodec`

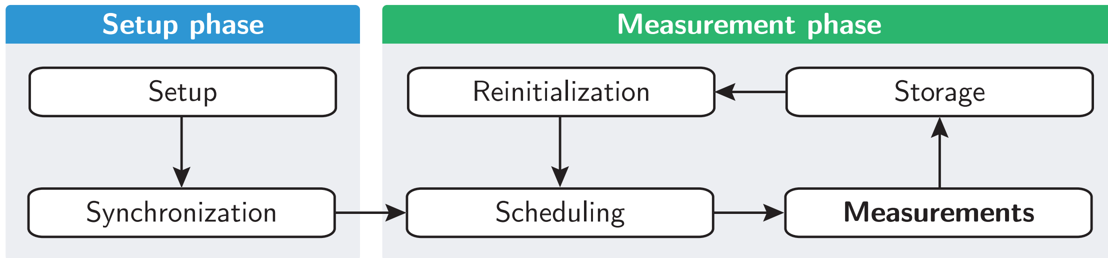

Figure 4.6: Benchmarking execution flow, composed of an initial setup phase and an iterative measurement phase.

4.2.9 Presenting and interface with VR client

The client side of the WVR link is a simple stand-alone VR application. Hence, it can be written using the standard OpenXR API, which would allow for straightforward ports of the client to other stand-alone systems. In particular, OpenXR has multiple uses:

- **Tracking**: retrieve head and controller poses, which are computed using a prediction model to mitigate the impact of latency. The expected display time is provided with the tracking data.
- **Frame submission**: acquire swapchain images for rendering, and submit rendered frames to the compositor. The expected display time of the pose used for rendering is also submitted alongside the frame, allowing the compositor to perform asynchronous reprojection.
- **Events**: handle VR events such as button presses or session termination, and trigger events such as controller vibrations.

4.2.10 Measurements

Finally, a system had to be designed for measurement collection. As mentioned earlier, the objective is to allow the collection of fine-grained measurements, including timestamps of all important stages of the pipeline, the amount of data sent and received by network sockets, as well as tracking and frame information. To increase the user-friendliness of the platform, diverse configurations – called benchmark passes – can be specified using command-line arguments provided to the server executable. The server then iterates over these passes, configures the client and driver accordingly, and collects measurements.

Figure 4.6 describes this execution flow as a state machine. At the start of the execution, the client connects to the server and synchronizes its clock

using the VRCP protocol described above. Once all components are ready, the measurement phase begins. The server communicates the current configuration to the client, schedules a measurement time window, and initiates the video pipeline. During this window, measurements are stored in local data structures, called measurement buckets. At the end of the test window, the server gathers and stores the measurement data in comma-separated values (CSV) files for straightforward processing. Finally, the system restarts and repeats the process with the next configuration.

4.3 Architecture

After having discussed the major issues to solve, the actual software architecture was designed. As mentioned earlier, the link is composed of three separate processes: a driver library, a server executable and a client application. A common library was also created to contain shared logic between processes.

The program was realized in the C++ programming language, as it is used by some dependencies, including OpenVR, D3D11 and Android Media Codec. Other dependencies, such as OpenXR, OpenGL or AVCodec, provide a C API instead. C++ also enables to structure the program in classes and namespaces, which is useful for large-scope project. Moreover, class inheritance allows for an easily extensible encoding system.

Several principles were applied in the design of the architecture. Firstly, it should allow the straightforward addition of alternative implementations of class interfaces. This permits the definition of platform-specific implementations and the test of different approaches as future work. This multi-implementation property can be implemented using two different approaches, depending on the use case:

- The **pointer-to-implementation** (pImpl) technique[6]: public classes only contain a pointer to an opaque data structure defined in the implementation-specific source file. Multiple implementations can be defined, using preprocessor definitions to only compile one of them at a time. This technique is useful to define platform-specific or library-specific implementations. It is also efficient, as it doesn't use virtual methods. Moreover, it allows for the separation of implementation details from header files, which accelerates compile times and enforces encapsulation.
- **Class inheritance**: implementations are defined as subclasses of an interface. The shared methods are declared as virtual, allowing runtime polymorphism

[6] `https://en.cppreference.com/w/cpp/language/pimpl`

by looking up the location of the implementation in a table, at the cost of a slightly slower execution. This technique is particularly useful for the simple definition of video encoders and decoders, as well as enabling runtime codec selection depending on benchmark settings and server capabilities.

Another important principle for the architecture design was the wrapping of platform-specific library access in common interfaces, allowing calling code to function independently of the operating system. For example, OS-specific network sockets were wrapped in the `TCPSocket` and `UDPSocket` classes. Therefore, the `VRCPSocket` and `VideoSocket` can be implemented for both Windows and Android with the same code.

The following sections detail the end architecture and the execution flow of the common library and the three processes. The final source code is publicly available on GitHub[7]. It contains 84 C++ files, for a total of approximately 13200 lines of code, excluding blank lines, comments, and tests.

4.3.1 Common library

The common library contains a collection of utilities used in multiple processes. It includes abstractions for inter-process communication, video compression, and the network protocols mentioned earlier. Additionally, it contains miscellaneous utilities and data structures. These components were designed independently of each other, reducing complexity and allowing them to be thoroughly tested without the need of a complete VR system.

Codec module system

To allow runtime codec selection, all the video compression and packetization logic was designed using the class inheritance technique. Four main interfaces were defined: `IVideoEncoder`, `IVideoDecoder`, `IPacketizer` and `IDepacketizer`. To add support for a new codec, opaque subclasses have to be defined for each of these interfaces. Factory functions must then be implemented, allowing the creation of a specific implementation. Finally, these factories are stored in a *module* alongside the codec name and identifier. The list of available module is then loaded by the server and the client at the beginning of the execution. In addition to built-in modules, external shared libraries in the executable directory are loaded if their file name begins with the "`wvb_module_`" prefix. The structure of a typical module is detailed in Figure 4.7.

[7] `https://github.com/martin-danhier/wireless-vr-bridge`

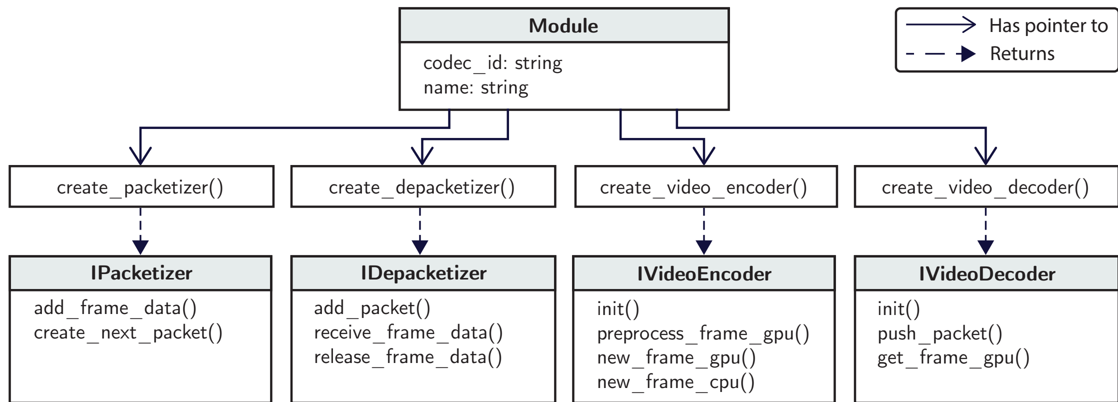

Figure 4.7: Architecture of a module. The `Module` structure contains pointers to factory functions that can be used to create opaque objects implementing video packetization and compression interfaces.

The built-in modules were implemented with the library and protocol choices previously mentioned. For video compression, the generic nature of AVCodec (for encoding) and Android Media Codec (for decoding) enabled multiple modules to be created with the same implementation, by simply varying creation parameters in the factory functions. This work principally focused on the H.264 [36] and H.265 [25] codecs, using hardware-accelerated Nvidia NVENC encoders and Qualcomm decoders. However, support for VP9, AV1 and VVC [37] could easily be added in the future, depending on the available hardware encoders on the server and client machines.

Video packetization is also greatly simplified by the use of the TCP protocol, using the simple header format defined in Section 4.2.5. This allows the `SimplePacketizer` and `SimpleDepacketizer` to be used with all codecs. However, the packetization was still included in the module system to simplify the implementation of RTP as future work, which will require codec-specific packet formats.

Network abstractions

As they are used in both the server and the client, the network protocols used in the WVR link were also defined in the common library. For each protocol, an abstracted socket was created, handling specific session and (de)-packetization logic.

The lowest-level sockets are the `TCPSocket` and `UDPSocket`, which are simple wrappers around OS-specific transport-layer sockets in order to bring them under

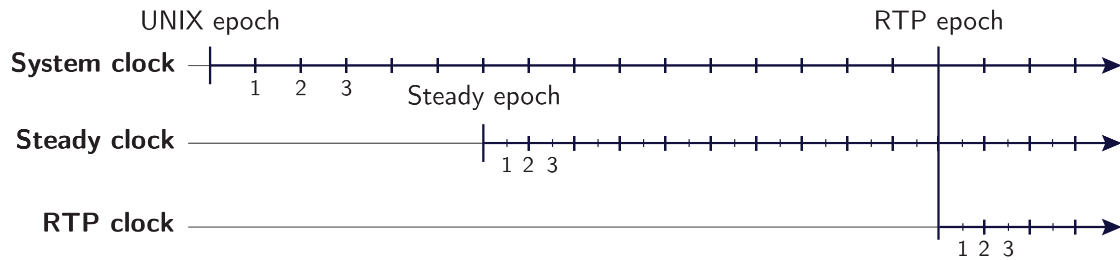

Figure 4.8: Relation of the RTP clock with the system and steady clocks. The system clock has a fixed epoch and low tick rate. The steady clock has an variable epoch but higher tick rate. The RTP clock represents the RTP epoch using both OS clocks to enable NTP epoch calculation with the system clock, while preserving a high tick rate.

a unified cross-platform interface. These low-level sockets were then used to implement the application-layer protocols for video and data transfer.

The `VRCPSocket` handles the virtual reality control protocol for data transfer defined in Section 4.2.6. It handles the complete handshake logic, from server discovery to session negotiation. Once connected, it provides methods to transmit VRCP messages, either reliably using TCP, or unreliably using UDP. On the receiver side, fields are automatically split based on the number of rows, enabling the straightforward iteration of received messages.

The `VideoSocket` handles the transfer of encoded video frames from the server to the client. The underlying protocol is chosen at compile time using preprocessor definitions. As mentioned before, only TCP was used in this master thesis, in which mode the simple packetizer and depacketizer are always used. However, in prevision for future work, a UDP mode was also implemented, using instead the packetization provided by the active codec module.

Synchronized clock

Numerous shared timestamps are measured across the program. They are used for various purposes: frame and tracking pose synchronization, latency measurements in benchmarks, and measurement time window scheduling. It is consequently important for these timestamps to be synchronized between the server and client, i.e for the same time value to represent the same point in time. An error between the two devices could indeed cause problems, most notably in the precision of measured network delays.

For this purpose, the timestamps were represented using the synchronized clock specifications detailed in the RTP RFC [32], implemented in the `RTPClock` class. Time values are counted in 90KHz ticks since an epoch shared between all components. The first tick is called the RTP epoch in the implementation. It corresponds to an arbitrary time point before the start of the VRCP session, but no more than 13.25 hours in the past, as RTP timestamps are represented with a 32 bits value.

In most systems, time values can be derived from two major operating system clocks: the system clock and the steady clock. The system clock counts the time elapsed since the UNIX epoch, the 1st of January 1970. This is useful as a global reference time to synchronize devices. The steady clock, on the other hand, does not have a fixed epoch, typically counting the time elapsed since boot. However, it can provide a higher tick rate than the system clock, and is thus used by the `RTPClock` to compute the main tick values. The relation of these three clocks is illustrated in Figure 4.8.

To allow inter-device synchronization, the `RTPClock` keeps track of two reference timestamps, representing the RTP epoch with each of the OS clocks. At the beginning of the execution, the client arbitrarily chooses an initial RTP epoch and computes the two associated timestamps. Using the system clock, a network-time protocol (NTP) epoch is computed, i.e the number of seconds elapsed since the 1st of January 1900. This value is then transmitted alongside the VRCP connection request to the server, which is able to adjust its own clocks, synchronizing the RTP time. Finally, during the synchronization phase, the client is able to shift its RTP epoch to compensate the error that could still exist due to system clock imprecisions.

Inter-process communication

The shared memory and inter-process events detailed in Section 4.2.2 were implemented on Windows using shared RAM file mapping, named mutexes, and named events. In order to send a piece of data from one process to the other, the sender locks the mutex, writes the new data in the shared memory then signals an event depending on the modification. Each event is able to function either in blocking or non-blocking modes. In blocking mode, the receiver waits on the event in a dedicated thread, allowing the data to be processed as soon as possible. In non-blocking mode, the event is periodically checked in an event looped, which is useful to avoid dedicating threads to events that do not require low latencies. Events represent each action that can be performed on the shared memory, including server/driver state change, new client specifications, new available frame, frame finished, new

tracking and new benchmark data.

Measurements

Finally, dedicated data structures were designed for the storage of measurements. These structures, called *measurement buckets*, are declined into three variants, for each process of the application. To minimize the execution impact of benchmarks, which could affect the results, all the arrays contained in buckets are pre-allocated. The data structures also directly handle the scheduled time windows, only saving provided values when necessary. Lastly, buckets contain utilities for the storage of saved values in CSV format.

4.3.2 Driver

As detailed in Section 4.2.1, three main components contribute to the execution of the driver process: the SteamVR compositor, the server driver and the HMD device driver. This execution is detailed in Figure 4.9. On launch, the driver signals its presence in the shared memory, requesting client specifications. If the server is running and connected to a client, its event loop will briefly respond with the desired information, allowing the driver to configure a device driver. Otherwise, the request times out and the driver exits with an error. In this case, SteamVR will try to fall back on other drivers, such as other links or tethered headsets.

If the initialization succeeds, SteamVR starts the VR environment and begins rendering frames. For each frame, when the scheduling of rendering calls is complete, the `Present` function of the HMD driver is called with a Direct3D shared texture handle. This function first synchronizes the frame to avoid exceeding the configured frame rate, then forwards the shared handle to the server. The subsequent `WaitForPresent` call then blocks the driver until the server has finished reading the shared memory. It is interesting to note that a frame can be rendered while the previous one is still being processed by the server, as SteamVR uses triple buffering, alternatively using one of three possible frame textures.

At each frame, an event handling function of the server driver is also called to process instructions received from the server. This function notably triggers the scheduling and transfer of measurements at each benchmark pass. It also shuts down the VR environment when the server is no longer ready to receive frame, for example if the VRCP connection is severed, or between each benchmark pass.

Finally, a simple thread waits on the *new tracking data* inter-process event and updates the current pose accordingly, ensuring low processing latency. This allows

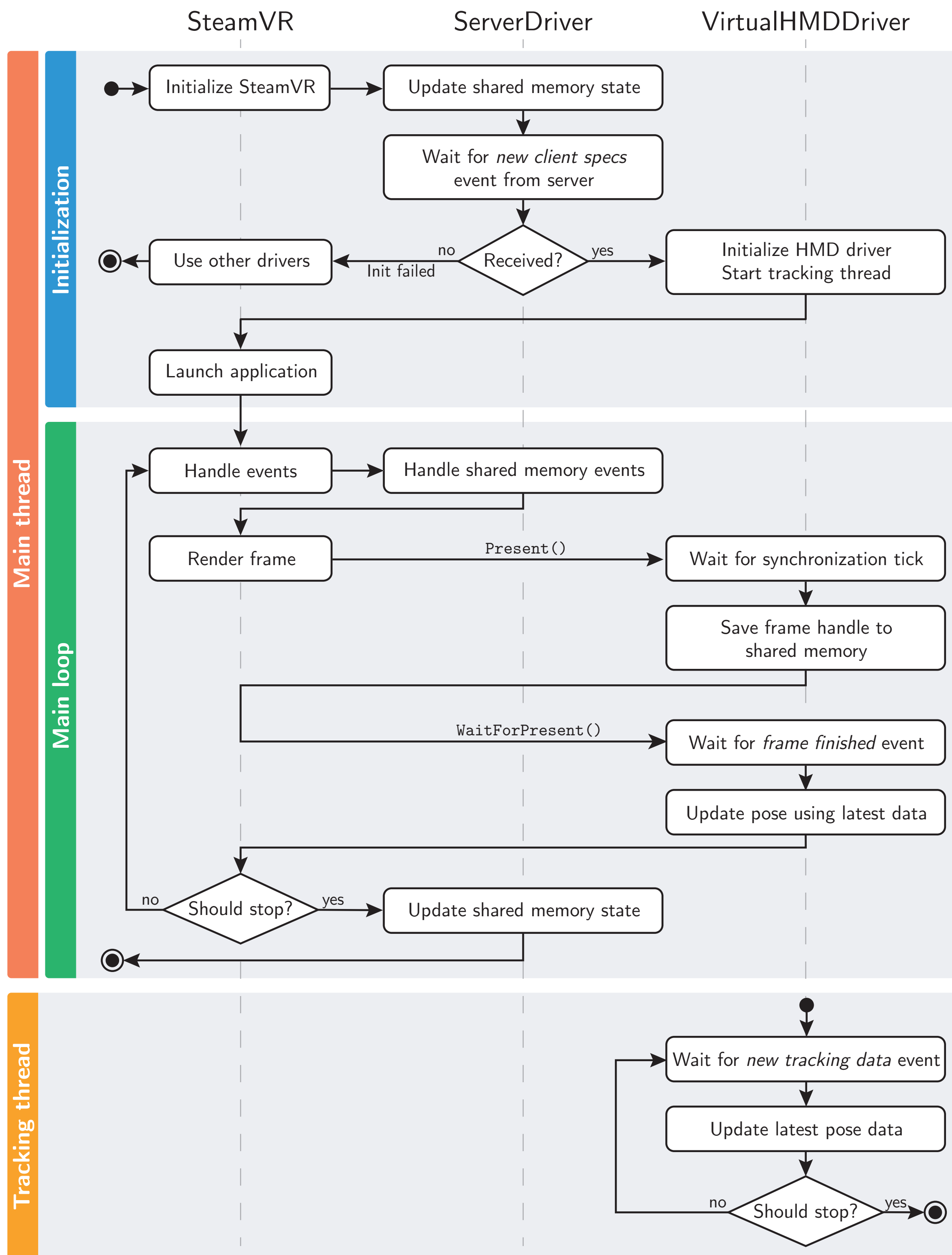

Figure 4.9: Execution flow of the two threads composing the driver process, and the role of each of the three major components in this execution.

each frame to be rendered with the most up-to-date tracking information.

4.3.3 Server

The server executable acts as a bridge between the driver and the client. It has multiple roles, including server discovery on the network, video encoding and packetization, orchestration of benchmark passes and the storage of measurements. The server is divided into two main components, each running in a separate thread, as depicted in Figure 4.10.

- The `Server`, the main class of the executable. It functions as an event-based state machine. In the main thread, a simple event loop is running, handling inter-process events and received VRCP packets.
- The `VideoPipeline`, a class dedicated to video compression and transmission. In a dedicated worker thread, a two-phase processing loop is running. The first phase consists of the retrieval and encoding of the frame. When a new frame is received from the driver, the shared texture is acquired and synchronized using the Direct3D 11 library. This raw image is then forwarded to the encoder module for optional pre-processing (such as format conversion) and encoding. The frame information is then pushed at the back of a first-in first-out (FIFO) queue. This approach is required by some encoders that introduce delay in the encoding in order to simultaneously process multiple frames. This means that a frame can sometimes only finish encoding after several subsequent frames have been pushed. The second phase consists of the retrieval of an encoded frame from the front of the queue, and its transmission to the client using the `VideoSocket` described above. Finally, measurements and frame captures are saved, depending on the scheduled measurement time window.

The server state machine is detailed in Figure 4.11. It contains the following states:

- **Disconnected**: while in this state, VRCP advertisements are regularly broadcasted on all local networks, enabling clients to discover the server.
- **Synchronizing clocks**: once a client connects, the server starts answering VRCP pings with as little latency as possible, until the VRCP *sync finished* message is received.
- **Ready**: when entering this state, the video pipeline and the encoder are initialized. Then, the server launches SteamVR to start the driver.

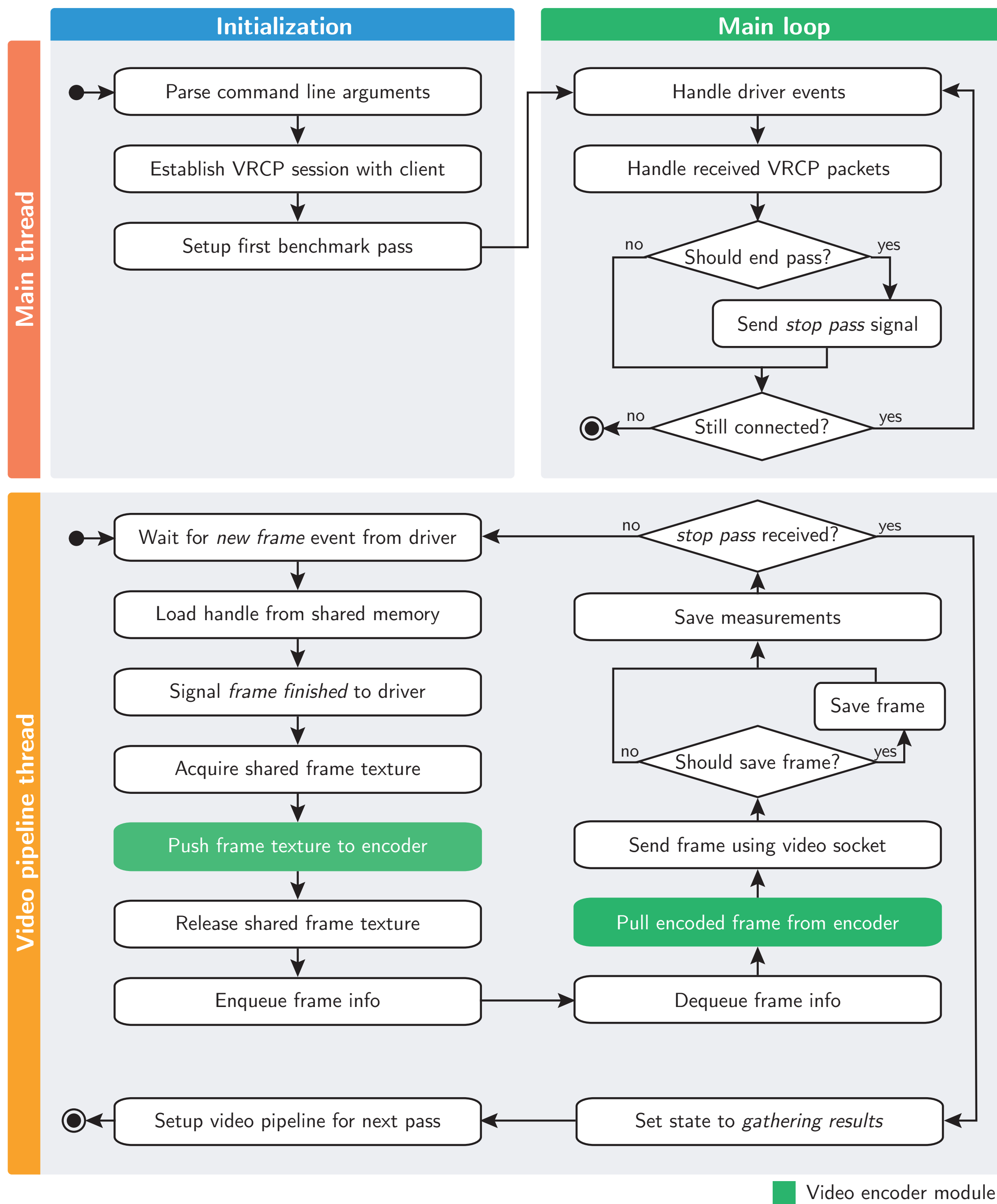

Figure 4.10: Execution flow of the two threads composing the server process, also highlighting the execution steps that are deferred to the video encoder module.

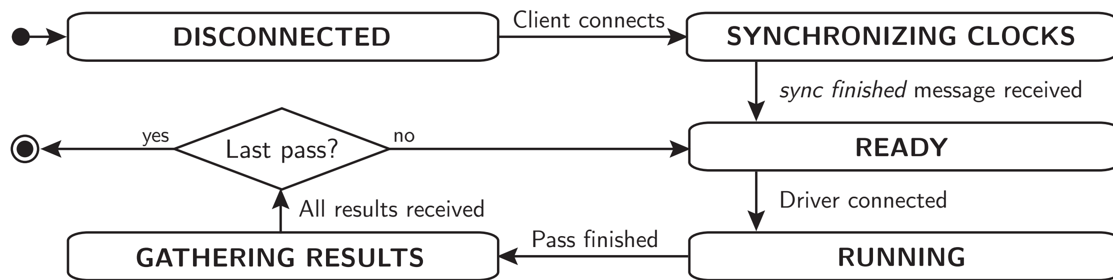

Figure 4.11: Server state machine, implementing the high-level benchmarking state machine detailed in Figure 4.6.

- **Running**: once the driver is ready and the VR application is running, the video pipeline worker thread is started and the link begins actively streaming VR frames and tracking data between the server and client. When entering this state, a measurement time window is scheduled and shared with both the driver and the client. During this time window, the collected data is saved in the `ServerMeasurementBucket`.
- **Gathering results**: at the end of the measurement window, the streaming stops and the server receives the data measured by the driver and the client. Once all the data is received, the measurements are stored in a CSV file, after which the system is restarted if another benchmark pass should be executed.

4.3.4 Client

At the time of writing this document, all stand-alone systems are mobile devices similar to smartphones, running on the Android operating system. Consequently, the base of the client VR application is a regular Java Android application, targeted at the Meta Quest 2 VR system. However, using the Android Native Development Kit[8], most of the logic can be developed in native C++ as a shared library. Even though the application itself is device-specific, the client library can therefore be easily ported to other systems.

The client library itself functions similarly to the server process. It is divided into two components, each executing in a separate thread, as depicted in Figure 4.12:

- The `Client` class is an event-based state machine equivalent to the one of the server.

[8] `https://developer.android.com/ndk/`

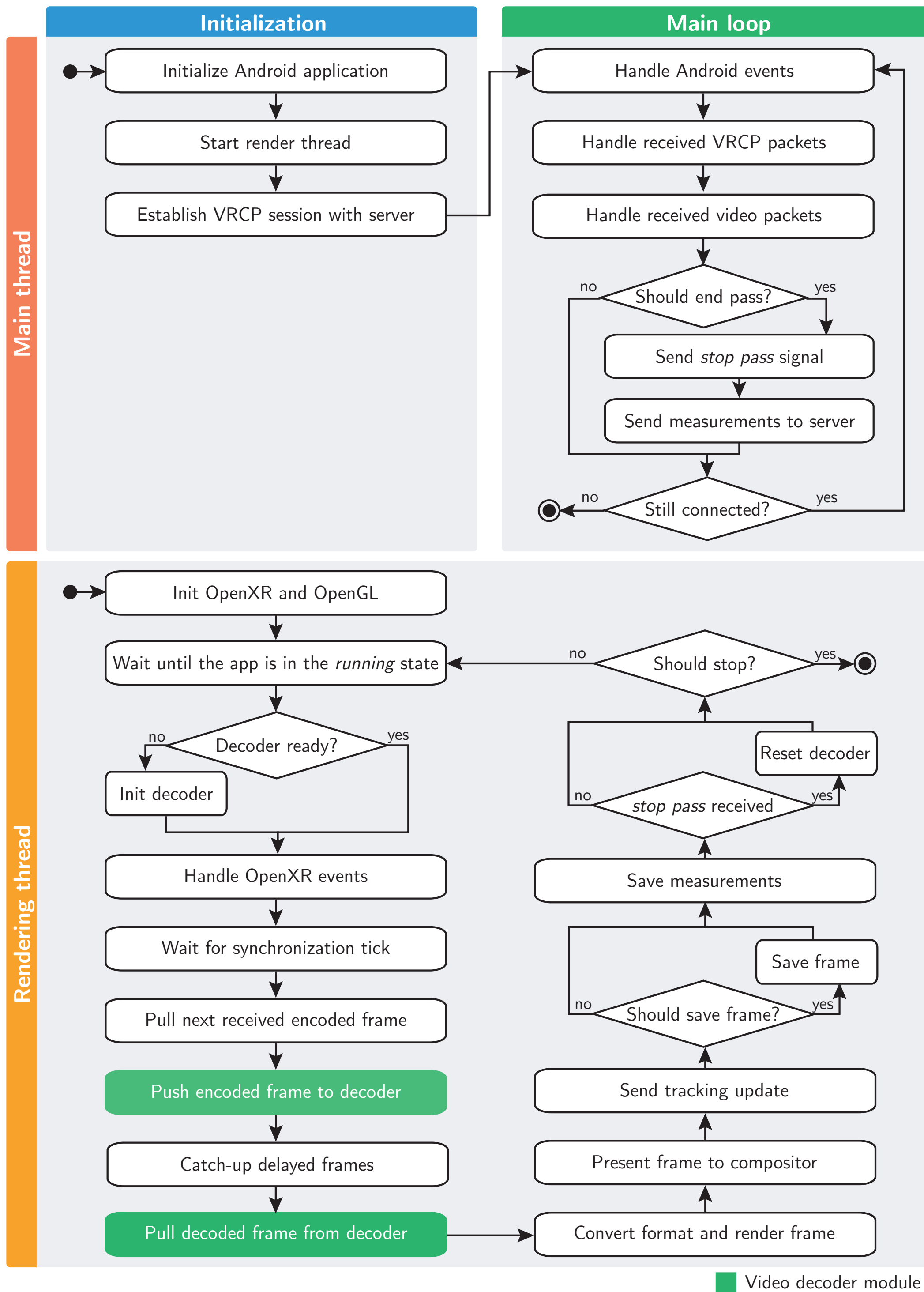

Figure 4.12: Execution flow of the two threads composing the client application, also highlighting the execution steps that are deferred to the video decoder module.

- The `VRSystem` handles the interface with the VR compositor with OpenXR, as well as the video pipeline. Iteratively, the render loop waits for a synchronization tick to ensure a stable frame rate. Then, all received frames present in the depacketizer are forwarded to the decoder. Optionally, some frames can also be dropped to reduce accumulated delay, which is one of the optimization techniques that are discussed in greater detail in the next chapter. The next decoded frame is then retrieved and rendered on the screen as a full-screen texture, before being presented to the compositor alongside the tracking pose used for rendering, enabling asynchronous reprojection. Finally, a tracking update is sent, and measurements are saved if needed.

Chapter 5

Results and discussion

The benchmarking platform detailed in the previous chapter is able to collect fine-grained measurements that provide useful insight on the performance of the wireless link. In this chapter, two series of measurements are discussed. In the first one, the causes of latency are evaluated using a frame-by-frame decomposition of each stage of the video pipeline. These measurements enable the identification of major bottlenecks and anomalies, allowing for the iterative development of optimization techniques. In the second experiment, the impact of video compression bitrate on image quality and frame rate is analyzed.

5.1 Testing setup

These measurements were conducted using a Meta Quest 2[1] headset as a client and a desktop computer as a server. The server ran on Windows 11 with a RTX 3070 GPU, an i7-9700KF CPU, and 16GB of RAM. Additionally, two different wireless local area networks (WLAN) were used in these tests:

- **Fast network**: TP-Link IEEE 802.11ax (Wi-Fi 6) router[2] disconnected from the Internet and entirely dedicated to the WVR link. This router is also located in the same room as the VR headset, maximizing signal strength and minimizing reflections or interferences. This network thus reflects an ideal scenario.
- **Slow network**: regular home network created using a Technicolor IEEE 802.11ac (Wi-Fi 5) router provided by the VOO Internet service provider[3].

[1] `https://www.meta.com/quest/products/quest-2/`
[2] `https://www.tp-link.com/fr/home-networking/wifi-router/archer-ax55/`
[3] `https://www.voo.be/en/internet/wifi-modem`

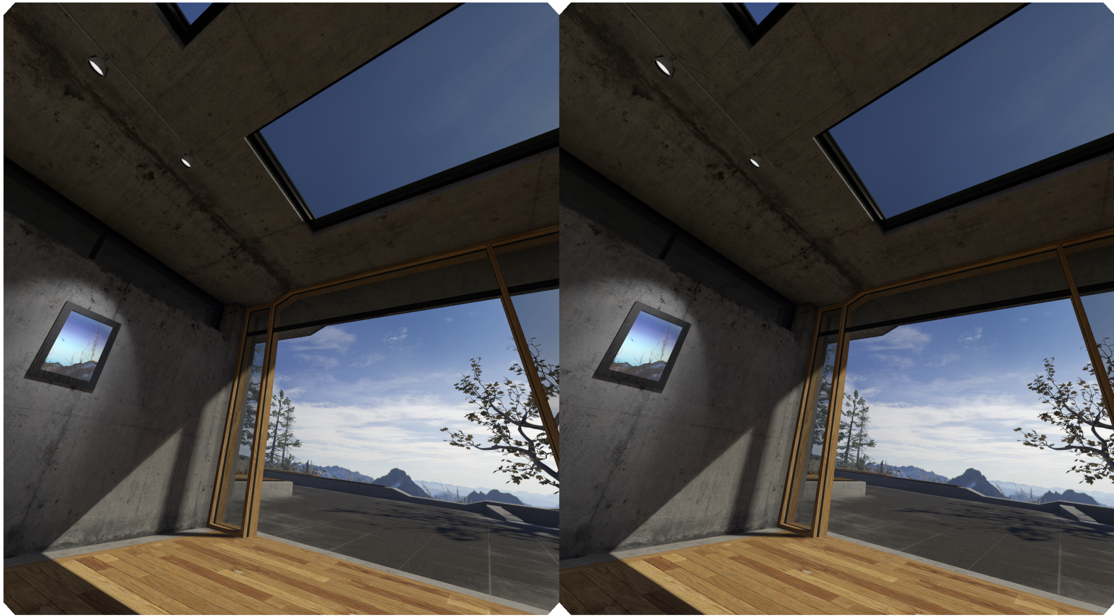

Figure 5.1: Example of rendered VR frame in the SteamVR Home application.

This router is located on a separate floor and is simultaneously used by other devices, recreating a typical home scenario.

In both cases, the server machine was connected to the router via a cable in order to minimize reliance on unstable wireless transmissions.

Regarding video encoding, all measurements were performed using the HEVC codec [25] with the NVENC hardware encoder and a Qualcomm hardware decoder. The encoder was configured using the default preset, a group of picture (GOP) size of 0, no overlapping frames, and no reordering delay. The transmitted video was of 2880×1584 pixels in size with a frame rate of 90 frames per second (FPS). The images were rendered in the SteamVR Home application. An example of rendered frame is depicted in Figure 5.1.

5.2 Delay analysis and optimization

Figure 5.2 shows the decomposition of each stage of the video pipeline as a parallel timeline. In this experiment, a single execution was measured with a bitrate of 100 Mbps during a 4-second window. Starting from a baseline depicted in 5.2A, each graph introduces an optimization technique aiming to address an issue observed in the previous graph. The following paragraphs discuss each of these configurations,

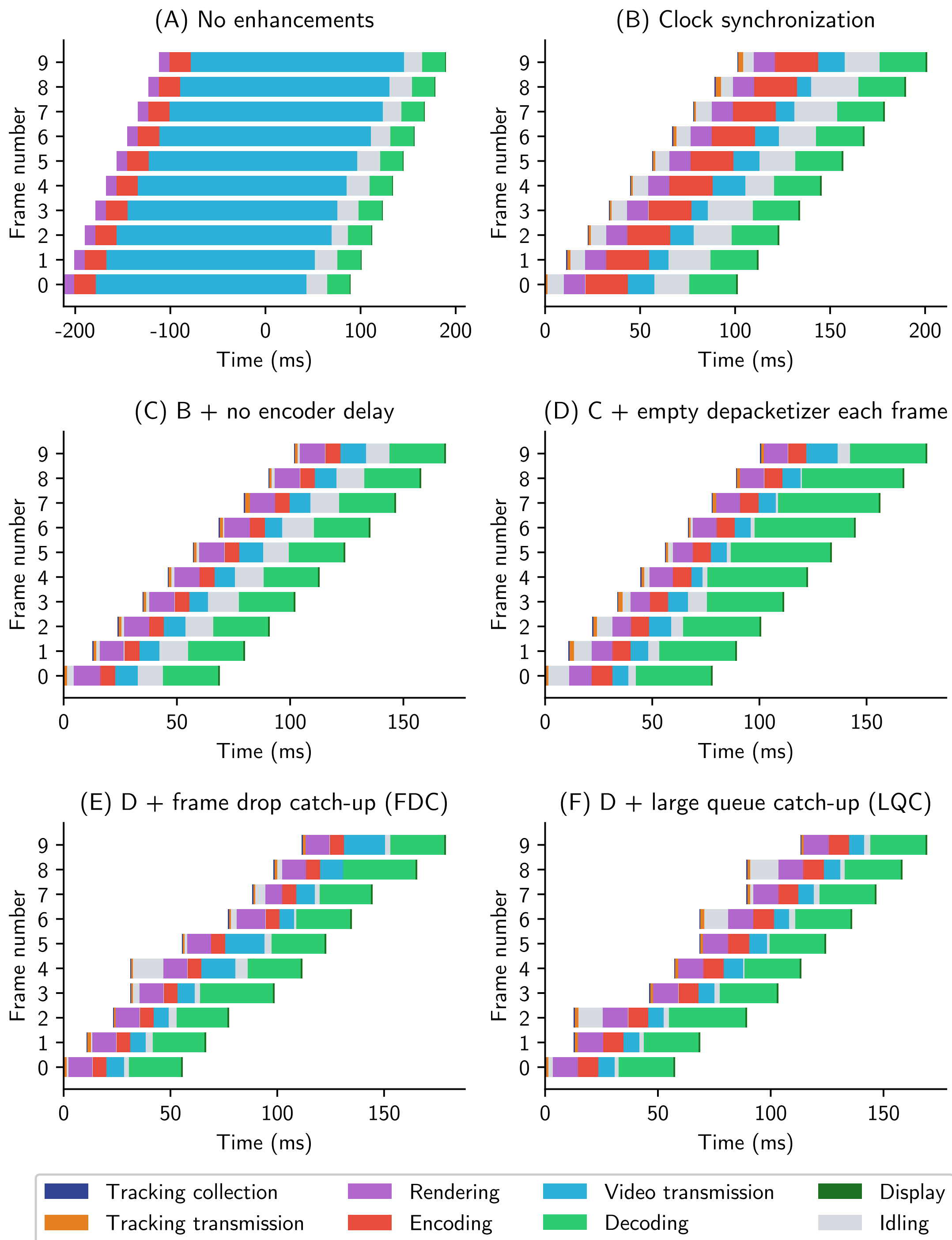

Figure 5.2: Frame-by-frame timeline highlighting each stage of the video pipeline, detailing the parallel execution of the first ten frames. Starting from an unoptimized baseline in (A), each graph introduces an additional optimization technique.

demonstrating the use of measurement collection for optimization.

5.2.1 Evaluation of optimization techniques

Figure 5.2A was obtained from measurements conducted with a basic pipeline, without any optimization nor synchronization. On the graphs, the time t=0 represents to moment when the tracking data of the first frame was collected. However, based on the measured timestamps, all frames appear to have been rendered and encoded at negative time points. This is logically impossible since tracking data would have needed to be received before it was collected. The root cause of this inaccuracy was identified to be a clock error, where the client system clock appeared to be offset by 200 milliseconds from the server clock.

In Figure 5.2B, the VRCP synchronization algorithms detailed in the previous chapter were introduced. This fixed the clock error, allowing the data to be interpreted. However, the observed total motion-to-photon latency was 100ms, which is a high value that can hinder the sense of telepresence. To minimize this latency, the detailed decomposition of each frame was used to locate the major bottlenecks to optimize. First, a significant amount of time appeared to have been spent in the video encoding and decoding stages. In addition, each frame seemed to idle for a duration between 15 and 25 milliseconds before entering the decoder.

In Figure 5.2C, the encoding time was significantly reduced by forcing the encoder to perform sub-frame encoding. By default, the NVENC encoder introduces a delay of two frames for video encoding. This means that after a frame has been pushed to the encoding queue, two more frames need to be pushed before the encoded version of the first frame can be pulled. Such a delay usually allows for higher frame rates, as multiple frames can be processed at the same time. However, for real-time applications, the induced latency is unacceptable. Sub-frame encoding forces the encoder to only work on a single frame at a given time, and return it as soon as it is finished. This parameter change reduced the encoding time from two inter-frame delays (22.2 milliseconds) to approximately 6 milliseconds.

In Figure 5.2D, the idling time before decoding was addressed. In the basic video pipeline, at each render loop iteration, a single frame is pulled from the video socket depacketizer queue and sent to the decoder. Therefore, delay can accumulate in the queue if new frames arrive at a faster rate than they are processed. In the configuration D, the render loop was modified to systematically send every received frames to the decoder at each iteration, significantly reducing the observed idling time. However, this also moved the delay-accumulation prob-

lem to the decoder. Delay can indeed accumulate when frames arrive at a faster rate than they are displayed, or if the decoding of a frame is unable to finish on time.

Two *queue catch-up* techniques were developed in the aim to address this delay-accumulation problem. These techniques attempt to minimize the size of the decoder queue by dropping outdated decoded frames, allowing frames to be displayed at the expected time point.

The first approach, referred to as *frame drop catch-up* (FDC), was introduced in Figure 5.2E. When the decoder is unable to finish decoding a frame in time, such as frame 3, FDC attempts to pull one additional frame in subsequent render loop iterations. This aims to drop the late frame and prevent it from affecting the presentation time of future frames. On the graph, a delayed frame is dropped when presenting frame 6, restoring the initial latency.

The second approach, *large queue catch-up* (LQC), is evaluated in Figure 5.2F. Instead of triggering frame drops when a frame is delayed, LQC simply pulls all completed frames when the queue size exceed a specific threshold. This ensures a minimal decoder queue delay.

5.2.2 Additional observations

As demonstrated in the previous section, the study of these graphs is extremely valuable for the development of latency and stability optimization techniques. In addition to the problems discussed above, several other issues can be observed on these graphs, and could be addressed in future works.

First, a comparison of 5.2B and 5.2C shows that there is a variable idling time between the reception of tracking data and the start of rendering. This variation is caused by the synchronization ticks awaited by the driver and the client in their render loops to ensure a maximal frame rate. These ticks are not synchronized together, creating an idling delay between 0 and 11.1 milliseconds. To address this issue, a synchronization mechanism could be introduced in order to dynamically update the driver synchronization ticks depending on the observed delay.

Moreover, tracking loss are regularly observed, causing old tracking data to be re-used and creating spikes in the motion-to-photon latency. For example, in graph 5.2F, tracking data was lost in frames 2, 6 and 8. This could be mitigated by sending tracking update at a higher rate, consequently increasing the probability

of a recent tracking update to arrive at the driver at each frame.

Finally, decoding time remains one of the major bottlenecks, despite the evaluated countermeasures. Alternative parameters, decoders or video codecs could be experimented, to ideally reach a sub-frame decoding delay.

5.3 Bitrate analysis

Figure 5.3 shows the impact of encoding bitrate on image quality, frame drop proportion and frame rate. In this experiment, for each evaluated bitrate value, these metrics were taken from thirty frame captures during three executions over a 4-second window. These measurements were repeated on both the slow and fast networks.

We can see that the amount of frame drops increases with the bitrate, especially above a threshold that depends on the capacity of the wireless network. Frame drops also appear to directly impact the observed frame rate, which fail to reach the target frame rate above the network threshold. The results also show that image quality is proportional to the bitrate, but that there is no difference between the two networks. This is easily explained by the reliability of the TCP protocol, which guarantees the intact reception of transmitted frames.

This experiment allows us to find the optimal bitrate settings for the wireless network at hand, allowing for high image quality while still reaching the target frame rate. For example, for the fast network, the optimal bitrate setting is 140Mbps. For the slow network, the optimum is instead 40Mbps. This also highlights the important role of the used WLAN in the quality of the wireless VR connection.

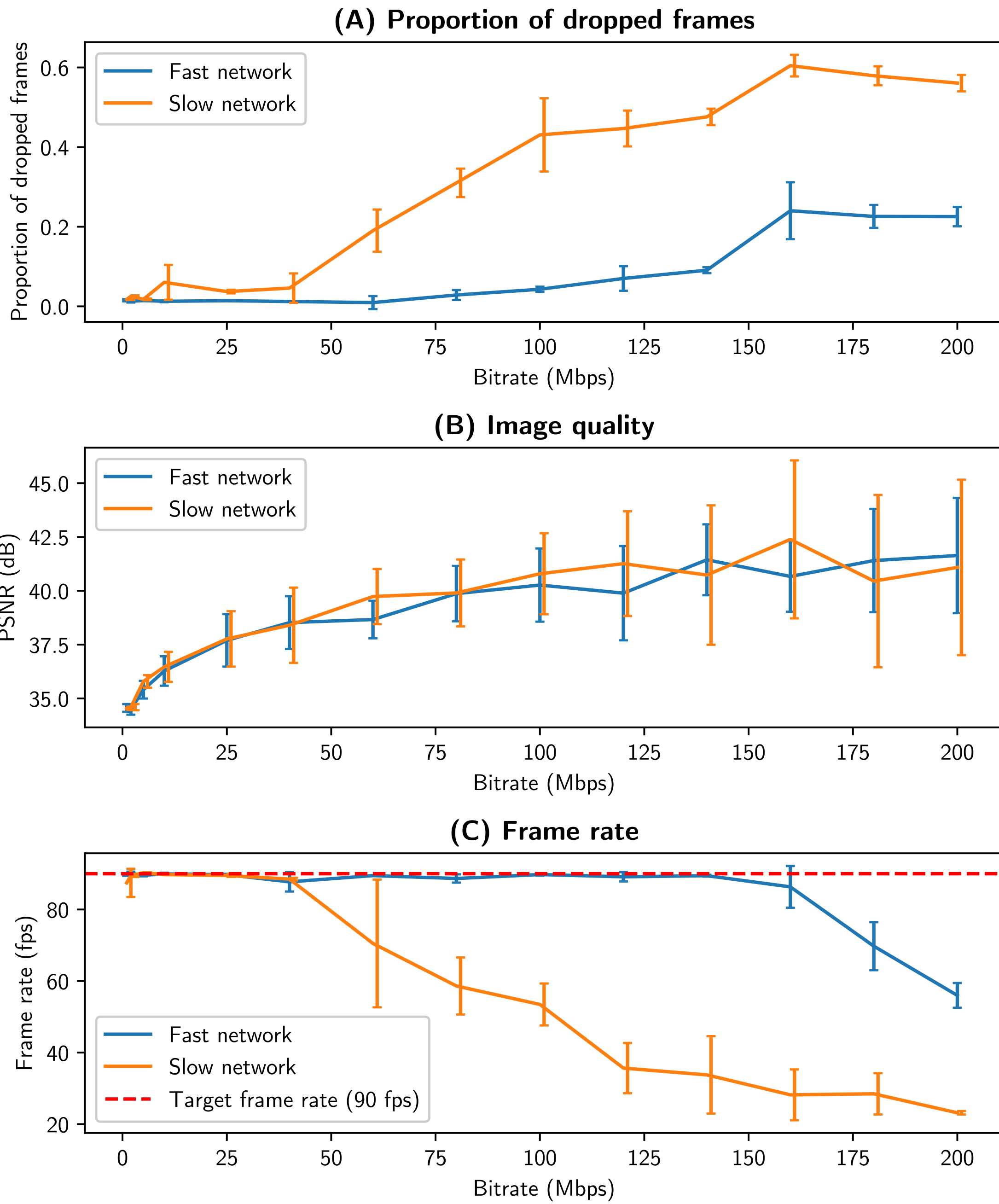

Figure 5.3: (A) Frame drops, (B) peak signal-to-noise ratio (PSNR) and (C) frame rate evolution with respect to bitrate increment, measured on two different wireless networks with varying capabilities.

Chapter 6

Conclusion

In this master thesis, a plug-and-play open-source wireless VR benchmarking platform was developed. This platform is able to link a stand-alone VR headset to a high-end computer in order to stream actual VR applications. This enables the collection of fine-grained latency and image quality measurements in a real-world scenario.

Through evaluation, we showed that the use of precise measurements in the video pipeline allows to solve technical and UX constraints. The detection of anomalies, such as late tracking data and frame drops, in addition to the identification of the major bottlenecks, such as decoding time, enables the detection of critical issues in the video pipeline. Countermeasures can then be developed and evaluated using benchmarks. In practice, the analysis of collected measurements allowed us to significantly reduce the observed motion-to-photon latency from 100 to 50 milliseconds. Additionally, measurements also enable the user to select the most appropriate parameter values for the application and network at hand in order to achieve an optimal user experience.

Moreover, the open-source nature of the proposed platform will enable the future integration of further features, such as motion controller inputs, sound support, the RTP protocol, additional metrics, as well as more advanced video algorithms, such as fixed foveated rendering [38]. Future work can also address the remaining latency problems by implementing dynamic driver synchronization ticks and higher tracking rates, as well as comparing the performance of other video codecs to minimize decoding delays.

Bibliography

[1] Shalini Talwar et al. "Digitalization and sustainability: virtual reality tourism in a post pandemic world". In: *Journal of Sustainable Tourism* (2022), pp. 1–28. DOI: 10.1080/09669582.2022.2029870.

[2] David Mirk and Helmut Hlavacs. "Virtual tourism with drones". In: *Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*. DroNet '15. 2015, pp. 45–50. DOI: 10.1145/2750675.2750681.

[3] Jorge Reyna. "The potential of 360-degree videos for teaching, learning and research". In: *INTED2018 proceedings*. 2018, pp. 1448–1454. DOI: 10.21125/inted.2018.0247.

[4] Nirit Gavish et al. "Evaluating virtual reality and augmented reality training for industrial maintenance and assembly tasks". In: *Interactive Learning Environments* 23.6 (2015), pp. 778–798. DOI: 10.1080/10494820.2013.815221.

[5] Mathilde R Desselle et al. "Augmented and virtual reality in surgery". In: *Computing in Science & Engineering* 22.3 (2020), pp. 18–26. DOI: 10.1109/MCSE.2020.2972822.

[6] Lucas El Raghibi et al. "Virtual reality can mediate the learning phase of upper limb prostheses supporting a better- informed selection process". In: *Journal on Multimodal User Interfaces* 17.1 (2022), pp. 33–46. DOI: 10.1007/s12193-022-00400-7.

[7] Jakub Žádník et al. "Image and video coding techniques for ultra-low latency". In: *ACM Computing Surveys* 54.11 (2022), pp. 1–35. DOI: 10.1145/3512342.

[8] Dang Tai Tan, Sanghyun Kim, and Ji-Hoon Yun. "Enhancement of motion feedback latency for wireless virtual reality in IEEE 802.11 wlans". In: *2019 IEEE Global Communications Conference (GLOBECOM)*. 2019, pp. 1–6. DOI: 10.1109/globecom38437.2019.9013507.

[9] Fenghe Hu et al. “Cellular-connected wireless virtual reality: Requirements, challenges, and solutions”. In: *IEEE Communications Magazine* 58.5 (2020), pp. 105–111. DOI: 10.1109/mcom.001.1900511.

[10] Sihao Zhao et al. “Virtual reality gaming on the cloud: A reality check”. In: *2021 IEEE Global Communications Conference (GLOBECOM)*. 2021, pp. 1–6. DOI: 10.1109/globecom46510.2021.9685808.

[11] Yen-Chun Li et al. “Performance measurements on a Cloud VR Gaming Platform”. In: *Proceedings of the 1st Workshop on Quality of Experience (QoE) in Visual Multimedia Applications*. 2020, pp. 37–45. DOI: 10.1145/3423328.3423497.

[12] Jonathan Steuer, Frank Biocca, Mark R Levy, et al. “Defining virtual reality: Dimensions determining telepresence”. In: *Communication in the age of virtual reality* 33 (1995), pp. 37–39. DOI: 10.1111/j.1460-2466.1992.tb00812.x.

[13] Philipp A Rauschnabel et al. “What is XR? Towards a framework for augmented and virtual reality”. In: *Computers in Human Behavior* 133 (2022), p. 107289. DOI: 10.1016/j.chb.2022.107289.

[14] Daniel Eger Passos and Bernhard Jung. “Measuring the accuracy of inside-out tracking in XR devices using a high-precision robotic arm”. In: *HCI International 2020-Posters: 22nd International Conference, HCII 2020, Copenhagen, Denmark, July 19–24, 2020, Proceedings, Part I*. 2020, pp. 19–26. DOI: 10.1007/978-3-030-50726-8_3.

[15] Ting Yang et al. “The impact of a 360 virtual tour on the reduction of psychological stress caused by COVID-19”. In: *Technology in Society* 64 (2021), p. 101514. DOI: 10.1016/j.techsoc.2020.101514.

[16] Jun Yi, Shiqing Luo, and Zhisheng Yan. “A measurement study of YouTube 360 live video streaming”. In: *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. 2019, pp. 49–54. DOI: 10.1145/3304112.3325613.

[17] Di Lei and Sae-Hoon Kim. “Application of wireless virtual reality perception and simulation technology in film and television animation”. In: *Journal of Sensors* 2021 (2021), pp. 1–12. DOI: 10.1155/2021/5041832.

[18] Jeffrey I Lipton, Aidan J Fay, and Daniela Rus. “Baxter’s homunculus: Virtual reality spaces for teleoperation in manufacturing”. In: *IEEE Robotics and Automation Letters* 3.1 (2017), pp. 179–186. DOI: 10.1109/LRA.2017.2737046.

[19] Gianluca Vadalà et al. "Robotic spine surgery and augmented reality systems: a state of the art". In: *Neurospine* 17.1 (2020), p. 88. DOI: 10.14245/ns.2040060.030.

[20] Abraham G Campbell et al. "Uses of virtual reality for communication in financial services: A case study on comparing different telepresence interfaces: Virtual reality compared to video conferencing". In: *Advances in Information and Communication: Proceedings of the 2019 Future of Information and Communication Conference (FICC), Volume 1*. 2020, pp. 463–481. DOI: 10.1007/978-3-030-12388-8_33.

[21] Frank Steinicke, Nale Lehmann-Willenbrock, and Annika Luisa Meinecke. "A first pilot study to compare virtual group meetings using video conferences and (immersive) virtual reality". In: *Proceedings of the 2020 ACM Symposium on Spatial User Interaction*. 2020, pp. 1–2. DOI: 10.1145/3385959.3422699.

[22] Michelle E Portman, Asya Natapov, and Dafna Fisher-Gewirtzman. "To go where no man has gone before: Virtual reality in architecture, landscape architecture and environmental planning". In: *Computers, Environment and Urban Systems* 54 (2015), pp. 376–384. DOI: 10.1016/j.compenvurbsys.2015.05.001.

[23] Christoph Anthes et al. "State of the art of virtual reality technology". In: *2016 IEEE aerospace conference*. 2016, pp. 1–19. DOI: 10.1109/AERO.2016.7500674.

[24] J. M. van Waveren. "The asynchronous time warp for virtual reality on consumer hardware". In: *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*. 2016, pp. 37–46. DOI: 10.1145/2993369.2993375.

[25] Gary J. Sullivan et al. "Overview of the high efficiency video coding (HEVC) standard". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (2012), pp. 1649–1668. DOI: 10.1109/tcsvt.2012.2221191.

[26] High-Definition Multimedia Interface. *HDMI 2.1a Specification Overview*. 2021. URL: https://www.hdmi.org/spec/hdmi2_1 (visited on 06/04/2023).

[27] Cisco. *Cisco Annual Internet Report (2018-2023)*. 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf (visited on 06/04/2023).

[28] Daniel Aguayo et al. "Link-level measurements from an 802.11 b mesh network". In: *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*. 2004, pp. 121–132. DOI: 10.1145/1030194.1015482.

[29] Benjamin Bross et al. “Overview of the versatile video coding (VVC) standard and its applications”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 31.10 (2021), pp. 3736–3764. DOI: 10.1109/tcsvt.2021.3101953.

[30] Aditya Venkataraman and Kishore Kumar Jagadeesha. *Evaluation of inter-process communication mechanisms*. Technical report. 2015.

[31] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293.

[32] Henning Schulzrinne et al. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. July 2003. DOI: 10.17487/RFC3550.

[33] Jonathan B. Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768.

[34] Ye-Kui Wang et al. *RTP Payload Format for High Efficiency Video Coding (HEVC)*. RFC 7798. Mar. 2016. DOI: 10.17487/RFC7798.

[35] David Leon et al. *RTP Retransmission Payload Format*. RFC 4588. July 2006. DOI: 10.17487/RFC4588.

[36] Thomas Wiegand et al. “Overview of the H. 264/AVC video coding standard”. In: *IEEE Transactions on circuits and systems for video technology* 13.7 (2003), pp. 560–576. DOI: 10.1109/TCSVT.2003.815165.

[37] Ticao Zhang and Shiwen Mao. “An overview of emerging video coding standards”. In: *GetMobile: Mobile Computing and Communications* 22.4 (2019), pp. 13–20. DOI: 10.1145/3325867.3325873.

[38] Rachel Albert et al. “Latency requirements for foveated rendering in virtual reality”. In: *ACM Transactions on Applied Perception* 14.4 (2017), pp. 1–13. DOI: 10.1145/3127589.

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