

Bachelorthesis

**Framework for delivering VR contents over
wireless network**

Kevin Woschny
17.11.2023

Supervisors:

Prof. Dr. Jian-Jia Chen

Zahra Valipour Dehnoo

Technische Universität Dortmund

Fakultät für Informatik

Lehrstuhl Informatik 12 (Eingebettete Systeme)

<http://ls12-www.cs.tu-dortmund.de>

Contents

1	Introduction	1
1.1	Motivation	1
2	Frameworks	3
2.1	OpenUVR	3
2.2	NS-3	4
2.3	5G-LENA	5
2.4	Docker	5
2.5	Dockerfiles	5
2.6	Docker Images	5
2.7	Docker Containers	6
3	Integration	7
3.1	Installing OpenUVR on host PC	8
3.2	trouble shooting FFmpeg	8
3.3	Unreal Engine	9
3.4	Modifying Unreal Engine	11
3.5	Quake	12
3.6	Modifying Quake	15
3.7	Installation of the receiving side	16
3.8	NS-3	17
3.9	NS-3 TapBridge	18
3.10	NS-3 File Descriptor NetDevice	19
3.11	NS-3 Simulation setup	21
3.12	NS-3 Simulation with a WiFi setup	23
3.13	NS-3 Simulation with a 5G Channel	23
3.14	Docker Container Setup	24
4	Measurements and evaluation	28
5	Conclusion	30
	List of Figures	31

List of Source Codes	32
Bibliography	33

1 Introduction

1.1 Motivation

Virtual Reality (VR) provides an immersive way to consume computer generated content. With the increasing computing power of personal computers many people do have access to the required hardware that is needed to provide an immersive experience in Virtual Reality. There are three common design ways for Virtual Reality systems: tethered devices, wireless devices without a host PC and wireless devices with a host PC.

A VR-system with a tethered design consists of a PC with high computing power and a low-performance head-device. The PC and wire, that connects the head-device to the PC, are capable to provide the performance to experience the virtual world in a high quality while maintaining a low latency. Image quality and latency are both crucial factors in user experience, especially how immersive the experience is. The big disadvantage of this setup is that the wire restrains the user's movement. It creates a tripping hazard for the person using the headset and they need to be aware of the wire and its current position at any given moment. These restrictions reduce the immersive experience of VR which is this main selling point. The other two VR systems solve this problem by using a wireless setup at the cost of latency and image quality.

Wireless devices without a host PC refer to a head-mounted device where all the hardware is located to generate content. One type of these all-in-one devices are powered by a smartphone which provides the necessary computing power to render content, have sensors to track the head movement and a screen to display the images. The other type are All-in-one devices where all components are built into the headset and no additional smartphone or PC is needed. Both implantation of this VR setup enables the user to freely move around with a headset on and the only restrain is given by the surrounding environment. But without a host PC and its high computing power these wireless devices can only rely on their own low-performance hardware. This resolves in a reduced image quality and limits the games and applications the user can run compared to the PC powered VR systems.

Wireless devices with a host PC bring the advantages of the two previous setups together: high image quality while keeping the freedom of a wireless setup. The content is generated on a PC which provides the high performance necessary for a high image quality. After the content is generated on the host PC it is sent over a wireless connection to the VR

headset. The headset only needs low computing power to prepare the images to display them. The problem that comes with this method is that delivering the content wirelessly introduces a new source of latency. While high image quality and free movement lay the ground for an immersive VR experience, latency can have major negative impact on user experience. Wireless devices with a host PC bring the need to be "able to deliver VR content to the user in 20 MS to avoid motion sickness in real-time gaming" [1]. There are many different types of approaches to reduce latency. One effective wireless VR system is OpenUVR. It is an open-source framework that achieves a pleasing user experience without the need for a special hardware setup. OpenUVR optimizes the VR Datapath and network stack "to reduce VR application delays to 14.32 ms"[1]. The impressive performance of OpenUVR makes it a great starting point to create a framework to evaluate the latency of VR applications within different network scenarios.

2 Frameworks

Next all frameworks used in the thesis will be presented and a little background for them is given.

2.1 OpenUVR

OpenUVR is an open-source Framework for interactive, real-time VR applications. To deliver the best user experience OpenUVR "adopt an untethered, wireless-network-based architecture to transfer VR content between the user and the content generator" [1]. The main barrier for this architecture "lies with a mismatch between the bandwidth demand for high-quality visual content and the sustainable bandwidth provided by the underlying wireless VR links" [1]. Even though there is compression that could reduce the required bandwidth, it also significantly increases the latency of VR DataPath. The latency of the VR framework needs to be as low as possible for the best user experience. OpenUVR "resolves the threeway trade-off between latency, bandwidth, and UX" [1]. The datapath of OpenUVR reduces unnecessary "data exchanges between memory spaces, data transformations between system modules/stacks, and data buffering between system component" [1].

OpenUVR uses a host PC to render the VR content and encode it to send it over the network to the receiving side. The host PC provides the necessary computing power to render the content in a quality to ensure a flawless user experience. In order to achieve a low latency transmission of the VR content, OpenUVR API functions are directly invoked in the game engine. With this approach OpenUVR does not create another process. This allows OpenUVR to work within the same context as the game engine and eliminates unnecessary memory copies and context switches, improving the efficiency of communication between OpenUVR and the game, creating the VR content.

To leverage from this idea, OpenUVR API functions need to be called within the games code. With which every game OpenUVR runs with, needs to be altered by adding the OpenUVR methods calls in the right place in the game engine code. On one side this enhances the performance of OpenUVR. On the other side it can make installing OpenUVR by hand a time-consuming, efforted and unconfined use for the average VR User, especially if conditions are changing in the game through an update or other things. When updates

alter the section of code where OpenUVR adds its functions, it may change the way where and how OpenUVR is bound into the game.

After capturing the VR content it needs to be encoded to be sent to Raspberry Pi, which is in control of the head mounted display (MUD). OpenUVR uses the libraries of FFmpeg to encode the captured content. FFmpeg is an open-source software used for editing multimedia data. It provides a collection of libraries and tools that enables the user to record, convert and stream audio and video. FFmpeg provides the functionality to encode, decode and transcode video and audio content. Encoding video content provides a compressed, more manageable format of the raw video content. OpenUVR uses these FFmpeg libraries to encode its content to reduce the amount of data that needs to be send over the Raspberry Pi and with that reduces the latency of transmission.

To further speed up the process of encoding CUDA can be enabled for OpenUVR and FFmpeg. CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model from NVIDIA. CUDA allows to use NVIDIA graphics cards to be used for general-purpose processing. Essentially, CUDA enables OpenUVR to use the computing power of the GPU beyond just for graphics, but CUDA is only available for NVIDIA GPUs.

On the receiving side OpenUVR operates on a Raspberry Pi. The Raspberry Pi has the task to decode the delivered VR content and display it via the MUD. It has enough computing power for this job while adding just a few grams to the MUD, guaranteeing it will not interfere with the user experience by increasing the weight too much of the head worn setup. OpenUVR needs to be installed on the Raspberry Pi and then started with the same settings as on the sending side of OpenUVR.

OpenUVR was designed with the idea to run without the need for a special setup. It operates and was tested on an average Wi-Fi setup as it is found in most households. While millimeter-wave (mmWave) wireless technologies greatly help the trade-off between latency and bandwidth, the downside of the need for special hardware and limitation of moment, through risk of losing connecting by walking outside of the transmission beam, led to the decision against mmWave.

2.2 NS-3

NS-3 stands for network simulator 3. It is an open-source network simulator tool used for simulating real-world computer networks. These networks can be defined through Scripts in C++ and run from the command line. For most network scenarios it would need multiple PC and route to test them and NS-3. Setting up these topologies can be time and cost intensive. Even if the needed resources are available, most of the times it is not the best idea to build such a network just for testing and experimentation. NS-3 provides the ability to virtual test these networks with just the need of a single PC. NS-

3 uses nodes to represent fundamental entities within the simulation. These nodes can represent PC, router, switches or any other network related device. In order to simulate the behavior of the network nodes can be configured with all kinds of network stacks, protocols, applications, etc. Between these nodes NS-3 allows the user to create point-to-point, wireless, CSMA and more connects between two or more nodes, which can be used to mimic LAN, WIFI or other real world connection setups. These connections can be created manually or with the help of different helper classes. Using the helper classes, it is possible to connect NS-3 to networks and application outside of the simulation.

2.3 5G-LENA

5G-LENA is an additional module for NS-3. It is a New Radio (NR) network simulator designed as a module for NS-3 and can be installed to extend the abilities of NS-3. It was developed especially for studying and testing the ability of various aspects of 5G networks, allowing the user to test the performance and function of 5G networks in the environment of NS-3. 5G-LENA adds various functions to NS-3 enabling to create a 5G connection between nodes in new written scripts or extending scripts already written for NS-3.

2.4 Docker

Docker is an open-source software platform that is used to create, develop and run applications in containers. Containers are a way to isolate software from the rest of the operating system while providing all necessary dependencies to run the software. This makes it possible to run an application across a variety of environments without the need to adjust the application for the underlying environments.

2.5 Dockerfiles

Dockerfiles are text files with a set of instructions that define how the Docker image is supposed to be build. Most Dockerfiles use other images as a starting point and add the necessary dependency, libraries and source code to create an alter or new image.

2.6 Docker Images

An Image is a template to create a Container. It contains a set of instructions for Docker to build a Docker Container. Images can be either build from a Dockerfile or can be pulled from a repository.

2.7 Docker Containers

Docker containers are running instances of a docker image. In contrary to Docker images, which are read-only, containers are running and interactable instances of an image. It runs the software and library defined in the Dockerfile and builds from the corresponding images creating a container that runs the desired application separated and isolated from the host operating system.

3 Integration

To measure the latency of OpenUVR in different network scenarios, the presented softwares will be used to integrate OpenUVR with NS-3. Using the ability of NS-3 to simulate various networks together with Docker, to isolate the application, will be used to run OpenUVR and route the data of it through NS-3. NS-3 and 5G-lena will allow to create different networks to test and run OpenUVR in, without the need for the required hardware and setting it up. Reaching this setup will need the following steps to be completed:

- Building OpenUVR
- Installing a game compatible with OpenUVR
- Modifying the games code to run OpenUVR
- Installing OpenUVR on a Raspberry Pi
- Creating a Docker Container with the game and OpenUVR
- Creating a simulated network in NS-3
- Connecting NS-3 and the Docker Container

3.1 Installing OpenUVR on host PC

To install the sending side of OpenUVR, it first needs to be built from source. Following the instruction from the OpenUVR GitHub, the first step is to clone the repository to the host PC and to install all required packages with "sudo apt-get install". Using the same command, a PC-wide FFmpeg installation needs to be done.

With "git submodule init git submodule update" additional the repositories of FFmpeg and Quake are cloned into the folder of OpenUVR. The include git submodule of FFmpeg is set to track version 4.0. Before FFmpeg is built for OpenUVR a few extra steps are necessary. The file "bgr0-ffmpeg.patch", which is include in the OpenUVR GitHub, needs to be applied with the command "git apply". This will change a few lines in the FFmpeg source code to enable support for CUDA encoding in RGBA format, which is disabled by default in version 4.0 in FFmpeg. For this encoding additional dependencies are required. The OpenUVR GitHub describes how to clone and install these required files. After these steps it is possible to build FFmpeg. Using the make file from the OpenUVR GitHub, the command "make ffmpeg" builds a version of FFmpeg, which can be used to build OpenUVR. The build ffmpeg files are placed in a folder in OpenUVR/sending. Building FFmpeg is required since OpenUVR relies on the created FFmpeg libraries to encode the VR content. With the FFmpeg files in place, OpenUVR can be build and installed with the two commands "make" and "sudo make install". This will first compile OpenUVR in the "OpenUVR/sending" folder and then copy the necessary files into "/usr/" to shared them with other programs.

3.2 trouble shooting FFmpeg

Trying to build the included FFmpeg version on Ubuntu 20.04 failed. While executing "make FFmpeg" an error was thrown "nvenc requested but not found".

Nvenc is an Nvidia feature for hardware based encoding, enabling to use the GPU instead of the CPU to encode video content. Searching online for this error did not provide any solutions. This error might be caused by compatibility issues with FFmpeg and the Nvidia graphic card or CUDA driver.

Switching to the newest version of FFmpeg resolved the problem.

3.3 Unreal Engine

OpenUVR needs a game to run with. The OpenUVR GitHub suggest "Unreal Tournament". It is a game from Epic games, developed on Unreal Engine 4. Development was started in 2014. It was possible to play the game through early access, but the game was never finished. It received the last major update in 2017 and in 2018 Epic Games announced the development of the game has been stopped. The games source files can be accessed through the Epic Games GitHub. The repositories of Epic Games are not publicly available, to join their GitHub repositories it is necessary to request permission. For the access an Epic account and a GitHub account are needed. After linking the two accounts the Epic Games GitHub can be accessed.

With this access it is possible to clone the Unreal Tournament GitHub. The Git clone command will download all files from the repository. It includes the project files of Unreal Tournament as well as the Unreal Engine version it is based on. Unreal Tournament is based on version 4.12 preview. The next steps to build and run the game is to execute the three commands `./Setup.sh`, `./GenerateProjectFiles.sh` and `make`. This will check if all requirements are satisfied to compile the game, like right compiler version or required external libraries, and then download additional content from Epic games. When everything is in place Unreal Tournament can be build.

If errors occur in this process, they can be hard to track down. The documentation for installing Unreal Engine and Unreal Tournament on Unix systems is limited, since there is just a small number of users for this operating system. There is a forum for Unreal Engine but posts in the forum often have cross-references to the official Unreal Engine wiki. The wiki went offline three years ago, which can make it hard to follow solutions for problems on the forum.

Trying to install Unreal Tournament resulted in an error while executing the command `./Setup.sh`. The error was "The remote server returned an error: (403) Forbidden. (WebException)". The setup script was unable to fetch the dependencies for Unreal Tournament from the Epic Games servers. A few months earlier this year there was a "disruption of service impacting Unreal Engine Users on GitHub"[2]. This resulted in the need to replace the "Commit.gitsdeps.xml" file from the downloaded repositories to resolve this issue. Epic games updated most of the GitHubs for Unreal Engines or provided a way to manually download them for older versions, but the inactive repository of Unreal Tournament did not receive an update or replacement file.

Replacing the xml of Unreal Tournament with an update file of an Unreal engine repository makes the error disappear, but now the setup script only downloads the dependency of the engine. Running the generate Project files script reveals that the dependencies for Unreal Tournament are missing and this approach does not lead a way to install Unreal

Tournament. Unable to obtain the necessary file of Unreal Tournament the idea was to just run OpenUVR with an Example Project of a working Unreal Engine version.

OpenUVR was built for UE version 4.12 preview. The first engine version that was tried to replace Unreal Tournament is UE 4.12. The differences between the released 4.12 version and the preview one used in Unreal Tournament should be minimal. This would guarantee that OpenUVR could be installed close to the way it is described on the OpenUVR GitHub, but version 4.12 requires the right version of Clang. Clang is a compiler for C and C++ and used by Unreal Engine to build the engine. The setup script checks if Clang is installed and which version of it. Clang can be installed with "apt install", on Ubuntu 20.04 version 10 will be installed. To build UE 4.12 it is required to have a version between 3.5 and 3.8. With any other version the script will abort the installation. Clang 3.8 cannot be obtained with "apt install" on Ubuntu 20.04. Clang 3.8 was released 2016, 7 years ago, making it quite outdated. There are prebuild versions of Clang to download, for Clang 3.8 there is a prebuild for Ubuntu 16.04. Running this build of Clang on Ubuntu 20.04 did not work. One other way to install Clang is to get the source files from the GitHub and build Clang from source. Unfortunately, this did not work out either. The build stopped with several error messages. It is possible to remove the restriction which Clang version is used to build UE 4.12. Editing a file called "LinuxToolChain" makes it possible to use Clang 10. Just adding Clang 10 to the desired Clang version in the part of the code where the Clang version is checked. With this modification the two scripts "Setup" and "GenerateProjectFiles" will run through without an issue, but then trying to build it with "make" results in a great number of errors. The restriction of Clang version has a reason and cannot be by passed.

Due to the fact that UE 4.12 cannot be installed properly, a switch to a newer version of Unreal engine was tried. After Unreal Engine 4.25, UE switch from using OpenGL as graphic API to Vulkan. OpenUVR uses OpenGL as the point to call its function from within the game and capture the frames, it needs to send to the receiving side. Unreal engine 4.25 seems to be the newest version that could be used for running OpenUVR, without the need for major change in the way how OpenUVR is integrated into the Unreal engine. The setup and installation of UE 4.25 can be done without any problems. After confirming that the installation was successful by starting UE4Editor, it was time to modify the Engine code to call OpenUVR.

3.4 Modifying Unreal Engine

OpenUVR needs that Unreal Engine uses OpenGL, but by default UE 4.25 already uses Vulkan. It can be changed by editing the file "BaseEngine.ini" in the folder "UnrealEngine/Engine/Config". That is done, by uncommenting "+TargetedRHIs=GLSL-430" in line 2214. After this UE4Editor can be started with the parameter "-opengl4", forcing it to use OpenGL.

The next step is to make the change that are listed in the file "ue4gitdiff". The file is included in the OpenUVR GitHub. This can be done by hand or with "git apply". The first step of the file needs to be done by hand, because the target file of this step has slightly changed in UE 4.25. After doing this step manually and deleting it from the "ue4gitdiff" file, the rest can be done with "git apply". Doing this will create a C Sharp build script that defines the directories and external libraries needed to run OpenUVR. Next step is to add the OpenUVR function to the engine code. It can be done by following the instructions of the OpenUVR GitHub. The last thing to do is to copy the compiled OpenUVR and FFmpeg libraries to allow Unreal Engine to use them and recompile UE4Editor.

Doing that will show that the C Sharp build script created by the "ue4gitdiff" file is outdated. In one version after UE 4.12 some of the syntax for this build files had changed. Some of the function names and class declaration had changed. Rewriting this file to work with UE 4.25 solved the error that was throw while recompiling UE4Editor. After the recompile UE4Editor can still be started, but OpenUVR does not start, stating that an Error occurred. The specific reason for this error is unclear, but the changed Unreal Engine, changed FFmpeg version or the alter build script might be the problem.

3.5 Quake

Given the recurring problems with OpenUVR and Unreal Engine, ioQuake3 was chosen as a replacement. IoQuake3 was the first game OpenUVR was modified with. The OpenUVR GitHub stats that even it is the first game OpenUVR run with it is less preferable than Unreal Tournament, but given the problems with it, ioQuake3 was still tested. Following the documentation on the OpenUVR GitHub, Quake can be cloned into the OpenUVR direction with the command "git submodule init git submodule update". The Makefile, which is responsible for building Quake, needs some changes to specify the location of OpenUVR, FFmpeg and other libraries. After that it is already time to modify the games code to use OpenUVR. The specified point for this is the file "sdl-glimp.c". In this file four functions need to be declared:

```
1  setup_openuvr_nocuda() send_openuvr_nocuda() setup_openuvr_cuda()  
    send_openuvr_cuda()
```

With the first two OpenUVR can be run without CUDA and the other two are used to run it with CUDA. The OpenUVR GitHub describes how to set them up. At the end of the function "GLimp-SetMode()" one of the created setup functions needs to be called. This will ensure that OpenUVR starts when Quake is started. Also, one of the send functions needs to be called right before "SDL-GL-SwapWindow()". After this function is executed, the next frame will be displayed. Calling OpenUVR before "SDL-GL-SwapWindow()" will make OpenUVR start processing the frame from the buffer before it is displayed on the host machine.

The last thing left to be done is built the game, but trying this with the functions specified in the OpenUVR GitHub will not result in a successful build. A few methods used in the four declared functions have been altered since the instructions have been written. Adjusting the parameters of the used methods will solve this issue.

```
1 void setup_openuvr_nocuda(void)
2 {
3
4     buf = malloc(4 * glConfig.vidWidth * glConfig.vidHeight);
5     ouvr_ctx = openuvr_alloc_context(OPENUVR_ENCODER_H264,
6         OPENUVR_NETWORKUDP, buf, 0); // null f r int pbo
7     openuvr_init_thread_continuous(ouvr_ctx);
8 }
9
10 void send_openuvr_nocuda(void)
11 {
12
13     glReadPixels(0, 0, glConfig.vidWidth, glConfig.vidHeight, GL_RGBA,
14         GL_UNSIGNED_BYTE, buf);
15 }
16
17 void setup_openuvr_cuda(void)
18 {
19
20     GLuint pbo;
21     glGenBuffers(1, &pbo);
22     glBindBuffer(GL_PIXEL_PACK_BUFFER, pbo);
23     glBufferData(GL_PIXEL_PACK_BUFFER, glConfig.vidWidth*glConfig.
24         vidHeight*4, 0, GL_DYNAMIC_COPY);
25     glReadPixels(0, 0, glConfig.vidWidth, glConfig.vidHeight, GL_RGBA,
26         GL_UNSIGNED_BYTE, 0);
27     ouvr_ctx = openuvr_alloc_context(OPENUVR_ENCODER_H264_CUDA,
28         OPENUVR_NETWORKUDP, 0, &pbo); //null buff
29     openuvr_cuda_copy(ouvr_ctx);
30     openuvr_init_thread_continuous(ouvr_ctx);
31 }
32
33 void send_openuvr_cuda(void)
34 {
35
36     glReadPixels(0, 0, glConfig.vidWidth, glConfig.vidHeight, GL_RGBA,
37         GL_UNSIGNED_BYTE, 0);
38     // technically should be called, but seems to be optional, and
39     // should be investigated:
40     openuvr_cuda_copy(ouvr_ctx);
41 }
```


OpenUVR is now built into Quake, in order for it to work the Ip addresses of the Host PC and Raspberry Pi need to be set in the files of OpenUVR files, otherwise OpenUVR will throw an error when starting Quake. At this time the used port should also be set. OpenUVR tries to ask for a port through the terminal when the games start, but it was not possible to make an input. The same Ips and ports need to be set on the receiving side.

Even though the source files of Quake are open source, not all required files of the game are. With the files from the Quake GitHub, it is only possible to build the base game. To run Quake additional files are needed. They can be acquired from the Quake CD or some other source and need to be placed into the compiled game folder.

With everything done Quake can be started for the first time with OpenUVR. The game will ask for a CD-key but can be tested without one. When the sending and receiving side are both started with the same transport protocol and encoding or decoding type, the Raspberry Pi should display an image. On the receiving side the transport protocol and encoding type are set in the setup functions declared in "sdl-glimp.c". On the receiving side the parameters are set when OpenUVR is started with the command:

```
1 sudo ./openuvr <encoding_type> <network_type>
```

Even if both sides are started correctly, the image is going to be probably fragmented. By default, Quake starts with a resolution of 640x480, and the receiving side expects an image of the resolution 1920x1080. Adjusting the resolution of Quake in the setting resulted in a crash of Quake. This in some way was caused by OpenUVR, it is possible to make this change in a Quake game without OpenUVR. To still get the right resolution an "Autoexec" is used. It is a file with a set of game console commands that is exuded every time the game is started. The "Autoexec" can also be used to set a costume resolution. It is used to set the resolution to 1920x1080 and enable the game clock. It is a timer in the top right corner of the game that counts up when a game is started and will be used to measurements. These changes remove the fragments on the Raspberry Pi side and deliver a working instance of Quake with OpenUVR.

```
1 //Autoexec
2
3 seta r_customHeight "1080"
4 seta r_customWidth "1920"
5 seta cg_DrawTimer "1"
6 seta sv_pure 0
7 seta Com_maxFPS 200
```

3.6 Modifying Quake

The game clock in Quake only displays seconds, but for the measurement milliseconds are needed. For this it is needed to modify the code of the game clock to add milliseconds. The code for the game clock is located in the file "cg-draw.c". The function is called "CG_DrawTimer", it subtracts the time the game was started from the current time. The result is given in milliseconds and then calculated into seconds and minutes. The result will be displayed as a string with the function "CG_DrawBigString". To add support for milliseconds the calculated minutes and seconds need to be subtracted from the time difference leaving behind the milliseconds. The calculated milliseconds then need to be added to the string.

```

1 static float CG_DrawTimer( float y ) {
2     char      *s;
3     int        w;
4     int        mins, seconds, tens;
5     int        msec, msec1, msec2;
6
7     msec = cg.time - cgs.levelStartTime;
8
9     seconds = msec / 1000;
10    mins = seconds / 60;
11    seconds -= mins * 60;
12    tens = seconds / 10;
13    seconds -= tens * 10;
14    msec -= seconds * 1000;
15    msec -= tens*10*1000;
16    msec -= mins*60*1000;
17    msec2 = msec/100;
18    msec -= msec2 * 100;
19    msec1 = msec/10;
20    msec -= msec1*10;
21
22    s = va( "%i:%i%i:%i%i%i", mins, tens, seconds, msec2, msec1, msec);
23    w = CG_DrawStrlen( s ) * BIGCHAR.WIDTH;
24
25    CG_DrawBigString( 635 - w, y + 2, s, 1.0F);
26
27    return y + BIGCHAR.HEIGHT + 4;
28 }

```

Compiling the game again will work without an error, but the made change will not appear in the game. Even though the code will be successfully completed it will not be used by Quake. This part of code is included in the files of the Quake CD. When Quake is built

it creates a build folder where all compiled files are placed and copied just the ones that are necessary and used to the installation directory. In order to use the modified, a folder needs to be created in the Quake installation directory. The name of the folder can be chosen freely, in this case it was called "mymod". Into this folder the modified files need to be put. This is done by copying the "vm" folder from the build folder into the create folder. Now the name of the folder can be specified in the starting command of Quake to use the modified files.



Figure 3.1: Quake new game clock with ms

```
1 sudo LD_LIBRARY_PATH=/home/kev/OpenUVR/sending/ffmpeg_build/lib ~/bin/  
   ioquake3/ioquake3.x86_64 +set fs_game mymod
```

This command will start Quake with OpenUVR and the modified game files and confirm that everything has been successfully installed.

3.7 Installation of the receiving side

To install OpenUVR on the Raspberry Pi, the instruction from the OpenUVR GitHub can be followed with the addition to alter the IPs and Port to match the sending side.

3.8 NS-3

NS-3 can be used to build various network scenarios, in this case the goal is to use it to test the performance of OpenUVR under different network scenarios. The general goal is to setup up four nodes creating three subnets. The middle subnet will be used for testing and its parameters will be modified to resemble the network scenario that will be tested. The right and left subnets both connect one of the middle nodes to an outside node. One of the outside nodes will be used to connect to a Docker container and the other will be used to connect to the host PC network device. This should enable a connection from the docker container through the left subnet then the middle subnet to the right subnet, which is connected to the host PC network device and therefore connected to the local network. This should enable a connection from the Docker container to the Raspberry Pi, while sending all data through the simulate NS-3 network.

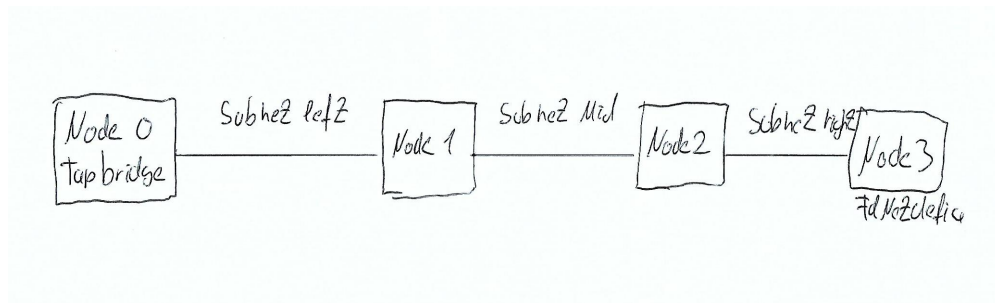


Figure 3.2: subnets of ns-3 simulation

3.9 NS-3 TapBridge

A TapBridge in NS-3 is used to connect a real internet host with the NS-3 simulation. A real host can be all devices that can use a Tun/Tap device. So, a real host can actually be a virtual machine (VM) like a docker container or other programs that allow to create a VM. In NS-3 a TapBridge is a network bridge that uses a Tap device to connect a node with a real host. The node to which the TapBridge is connected is called "GhostNode", since its only purpose is to forward the data that comes to it.

A real network device is a physical device that sends packets around through ethernet cables, while a Tap is a completely virtual interface. It is used to simulate the physical connections of a real network device. Despite being complete virtual a Tap device acts and behaves like a real network device and can be used with network protocols like IPv4, IPv6 and more.

It is called a Tapbridge, because it basically connects the inputs and outputs of an NS-3 net devices and Tap device together. Allowing for communication to travel from the Tap into the simulation via the Ghostnode that holds the Tapbridge.

There are three modes a Tapbridge can be operated in. While all of them basically have the same functionality and behave the same, they differ in the way they are setup and which devices can interact with them. The modes of a TapBridge can be ConfigureLocal, UseLocal and UseBridge and their name already indicate their behavior. While in ConfigureLocal mode the TapBridge is in charge of configuring the Tap, in the UseLocal and UseBridge the tapbridge adapts to the given Tap devices configured by the user.

To connect it with the Docker container, the Tapbridge will be used in UseBridge mode. A Tap device needs to be manually created and named. It can be done with a series of commands via the terminal. First the Tap needs to be setup and connected to the docker container and then the NS-3 simulation will be started. If the Tapbridge is correctly configured in the code of the simulation, it will connect with the tap device.

```
1 TapBridgeHelper tapBridge (interfacesPL.GetAddress (0));
2 tapBridge.SetAttribute ("Mode", StringValue (mode));
3 tapBridge.SetAttribute ("DeviceName", StringValue (tapName));
4 tapBridge.Install (nodesL.Get (0), devicesL.Get (0));
```

In the simulation used for testing OpenUVR, the Tapbridge is located in node 0. This is the left outside node. This will connect the container to the left subnet and the rest of the simulation. The Tapbridge is created by using the TapBridgeHelper class. The helper class is used to make it easier to build a Tapbridge. Frist the helper class is given the IP-adress of the node the TapBridge will be installed on. This IP-address is used as default gateway for the created Bridges. Then the mode of the Bridge is set, UseBridge in this case, and the name of the Tap is set. The name that was set needs to be the name of the Tap devices that is manual created in the terminal. The last step is to install the

Tapbridge to the node. In order to work the Tap and the Docker container need to be assigned IP-addresses in the same subnet as the Ghostnode. In this mode, the rest of the configuration must be done in the terminal and in the Docker container.

3.10 NS-3 File Descriptor NetDevice

The TapBridge creates a connection between NS-3 and the Docker container, but for OpenUVR to work the simulated network still needs to be connected to the Local network or more precisely to the Raspberry Pi running the receiving side.

The File Descriptor NetDevice is provided by the FdNetDevice class. This module can be used to read and write traffic via a File Descriptor. The File Descriptor can for example interact with a Tap device or a raw socket to read the output of this devices or write data to these devices. For this class there are also helpers to setup the File descriptor and link it with a NS-3 node. The FdNetDeviceHelper is the base helper, using this helper only installs all necessary things to a node that is given to the helper. This helper will not create or setup the File Descriptor. It must be done manually by the user himself.

An easier way to create and setup the File Descriptor is to use the Emu or Tap helper. The Helpers of them are called EmuFdNetDeviceHelper and TapFdNetDeviceHelper. They can be used to do most of the required creation, setup and installation. The Tap helper uses, like the name suggest, a tap device to be able to forward data from the NS-3 simulation to the local network.

The EmuFdNetDeviceHelper on the other hand creates a raw socket to the physical devices of the host machine and setup the corresponding Socket Descriptor. Using this network device, the NS-3 simulation can read and write frames to the underlying physical devices of the host machine and establish connection to the internet.

In the simulations for OpenUVR the EmuFdNetDevice sits on node four. It is the outside node of the right subnet. To setup the EmuFdNetDevice, first the EmuFdNetDeviceHelper gets created and then the devices name is set. This has to match the name of the Ethernet network interface of the host machine the EmuFdNetDevice is supposed to work with. In many cases of Linux distributions, it will be called "eth0", but the name may be different and can be looked up with a command like "ifconfig". After setting the right name to the "deviceName" attribute, the helper can be used to install the FdNetDevice to the right node. Like mentioned before in this case the device gets installed on the right node of the right subnet. The EmuFdNetDevice needs to have a different MAC address then the network interface of the host machine. The NS-3 helper uses MAC spoofing to alter the MAC address of all traffic leaving the NS-3 simulation through the FdNetDevice. The MAC address can be set manually or with a helper. The user needs to ensure that it is alright to use MAC spoofing in his network and that the spoofed MAC address do not

conflict with anything else on the network like other simulations also using this device. The line

```
1 device->SetAttribute("Address", Mac48AddressValue(Mac48Address::
    Allocate()));
```

will be used to give the Emu devices a MAC address generated by NS-3. This simulation will be the only one running on the network and no other device should conflict with the generated MAC address.

To be able to use the devices to send packets to the local network a IPv4 interface needs to be added and configured. For the IPv4 interface the attributes "localIp" and "localMask" need to be set. "localIp" needs to be an IP address that matches the subnet of the host machine and is not used by any other devices in the network. The attribute "localMask" needs to represent the subnet mask of the host machine. The next thing is to allow forwarding in the interface used by the FdNetDevices by setting it with the following line:

```
1 ipv4->SetForwarding(interface, true);
```

Last thing to do is to set the gateway of the FdNetDevice. The gateway of the FdNetDevice has to be the same as the gateway of the host machine.

With this configuration it is possible to ping other devices in the local network from the NS-3 node, since it has an IP address that matches the subnet of the local network. Trying to ping from one of the other three nodes of the simulation resulted in 100% packet loss. The ping reached their destination but got lost on the way back. There was no route to the subnets of the simulation in the router of the local network. To change this a static route has to be added to the router of the local network. If a packets destination is one of the subnets of the simulation the IP address of the FdNetdevices is the gateway for this packet. With this configuration of the router all nodes of the simulation can ping devices of the local network or the internet.

3.11 NS-3 Simulation setup

To test NS-3 a simulation will be used that has been setup with four nodes. Between the middle nodes there will be the configuration that will be tested. Then the middle subnet will be connected to the outside nodes. One of them will have a TapBridge installed to connect to the Docker container, the other node will have the EmuFdNetDevice to enable communication with the local network and the internet. A NS-3 simulation can be setup in a single c++ file. The file should be placed in the "ns3/scratch" folder and can be run with the command

```
1 ./ns3 run nameOfFile
```

The command can be executed in the terminal from the NS-3 folder. The command-line option "--enable-sudo" might be necessary for the FdNetdevice.

At the beginning of C++ file all libraries are included. They are for example necessary for all the helpers that are used to setup the simulation. Then some functions can be defined that can be used to do ping test or map the network and its routing table. These functions can be created and used for testing and debugging or to run the simulation with simulated data traffic, but there are not necessary for this setup. The only necessary function is the main function, where all nodes are created, all connects are setup and all NS-3 Netdevices are installed on to the nodes.

The first thing in the main function are the variables used for setting up the TapBridge and FdNetdevices. If the simulation should be run on a different host PC, they need to be altered to match naming and configurations of the host machine and its local network. The variables "mode" and "tapname" are for the Tapbridge. "mode" should not be changed unless a different setup for the connection with a VM is desired and "tapname" does not need to be changed either, since the TapBridge is used in UseBridge mode the Tap devices and name of the Tap devices are set and created manually. If for some reason another name for the Tap is picked, the variable "tapName" needs to be adjusted. The next Variables are for the FdNetdevice. "deviceName" should represent the network interface name of the Host machine. It is named something along the likes of "eth0" or "enpXXX" and should be adjusted to the current PC. The "localaddress" is the address the FdNetdevice gets set to and it needs to be free and in the subnet of the host. The "localGateway" variable must hold the IPv4 address of the gateway of the local network and will be used for the setup of FdNetdevices.

Then the three subnets and its nodes are created. First the middle subnet is created, for testing this network was setup as a "CSMA" devices. It models a simple bus network similar to Ethernet. "CSMA" stands for Carrier Sense Multi Access with Collision Detection. NS-3 simulates only a part of this protocol, collision cannot happen with this device in NS-3. It is simple to setup and simulate and therefore used for testing the whole setup. For the middle subnet two nodes are created and added to the middle node container.

Node containers are used to manage a group of nodes, a node container behaves similar to an array. Every node inside the node container gets an index, starting at zero and ending at $n-1$ and nodes can be accessed with their index. With a CSMA helper the CSMA channel is created, and two attributes need to be set. One being the Data Rate the channel operates at, and the other is the delay of the channel. After these two are set the CSMA channel can be installed to the node container, adding all nodes of the container to the channel. Then the IPv4 interface is added to all nodes of the container. For this interface a base needs to be set, consisting of an IPv4 address and the subnet mask of the channel. For the middle subnet it is "(10.1.1.0, 255.255.255.0)". This means the first node of the node container will get the address "10.1.1.1" and the other will have "10.1.1.2".

For the left and right subnet, the setup is similar. Both of these subnets will be setup as CSMA channel and will not be changed for consistency. Their purpose is to ensure that the FdNetdevice and Tapbridge work. These two network devices are not compatible with every type of channel, for example the TapBridge cannot be used directly with a peer-to-peer setup. The left and right subnet ensure that the middle subnet can be swapped out without a need to worry about these restrictions. For the setup of the left subnet one new node is created and the node with the index 0 from middle node container is taken to create the left node container. A CSMA channel is created and installed on to the left node container. The node with the index 0 has the IP address "192.168.10.1" and the other one has the IP address "192.168.10.2" on this interface and the IP address "10.1.1.1" on the other interface and being the gateway between the two subnets. The node with the single interface gets the Tapbridge to connect the Docker container. The right subnet gets setup analogue to this. One new node is created, and the other is taken from the middle subnet. A CSMA channel between them is created. One node is the gateway between the middle and right subnet with the IP addresses of "10.1.1.2" and "192.168.11.1" and the other node has the IP address of "192.168.11.2" and got the FdNetdevice installed. Then function is used to populate the routing tables of all nodes with:

```
1 Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

With this communication is possible between all nodes in the simulation. The last thing to do is to set a default route to the nodes to allow traffic to find its way out of the simulation through the Fdnetdevice.

3.12 NS-3 Simulation with a WiFi setup

To run this setup with another channel the CSMA one, simply the channel of the middle node container needs to be changed. This can be for example be replaced with a Wi-Fi channel. The IP address of the nodes stay the same and also the default route. To create the channel there is again a helper for this. The protocol and the Wi-Fi name need to be set and then the nodes get added to the channel. These parameters are left at their default setting and can be changed later for testing. The last thing to do for the Wi-Fi connection is to add a position of the nodes with a mobility helper. The position then can be set in a coordination system with x,y and z parameters in meters. The coordintes will determine how far apart they are and will influence the performance of the Wi-Fi channel.

3.13 NS-3 Simulation with a 5G Channel

The idea was to use the 5G-LENA to create a 5G setup in the Middel channel. There are multiple examples included in the 5G module showing how to setup and use the module. The setup of a 5G channel and its nodes can only be achieved with helper classes. One node act as a base station and the other nodes called "Ue" are connected to the base station via a mmWave channel. There are examples that show that it is possible to connect subnets to the base station and the ping the Ue nodes for the the subnets, but the Ue nodes are always endpoints in the example and do not act as gateways to other subnets. If a ghostnode is connected to a Ue node, the Ue node can ping both the base station and the ghostnode. But the base station and ghost node cannot successfully ping each other. Even if static routing is setup in the base station, Ue node and ghost node to allow this route, a ping between the two was not possible. When the helper creates the base station multiple nodes are created and their routing is automatically set, and they act as the base station. The packet of the ghost node gets somewhere lost in the nodes base station. In the google groups forum of NS-3 there are entries from people with the same routing, in this setup, problem but no solution has been posted. In addition to that there are entries on the forum to the performance of 5G-LENA in real-time simulation. The models of 5G-Lena have a higher complexity and therefor perform poorly in real-time simulation.

3.14 Docker Container Setup

The Docker container will have Quake and OpenUVR installed. For this a Dockerfile is used. The File needs to be placed into the OpenUVR folder and replace the Dockerfile from the OpenUVR GitHub. Inside the OpenUVR folder should be the sending folder, with the OpenUVR source files and FFmpeg, and the Quake folder with the cloned files of the IOquake3 GitHub. The setup of these two should be tested before running the Dockerfile to make debugging easier. If everything runs, the IP addresses set in the OpenUVR files need to be changed to represent the IP address the Docker container will get. In this NS-3 setup the container will be connected to the left subnet and given the IP address "192.168.10.6". The changes to the Quake source code, adding OpenUVR and changing cg-draw, can also be done and tested before the Docker container is created. One folder needs to be added to OpenUVR directory. It should be named "cd" and contain the "pk3" files needed to run Quake. These files are not included in the GitHub of Quake and can for example get from the Quake3 CD. The "autoexec" can also be placed inside the CD folder. The file can be modified if other Quake console comments need to be executed before the start of the game, but for this setup it only sets a costume resolution to "1920 x 1080" and enable the game clock. These are all the things which need to be included in the OpenUVR folder to run the Dockerfile.

The Dockerfile is based on a CUDA image that can be used to create a docker container running Ubuntu and Cuda installed. Then the libraries needed for OpenUVR and Quake are installed and the OpenUVR folder with Quake, sending side and the CD folder are copied to the docker image. Then FFmpeg and OpenUVR are build and installed. When this is done, Quake gets build, installed and then the pk3 files are copied into the Quake folder. The docker image can be built from the Docker Docker file with the command:

```
1 docker build -t nameOfImage .
```

When the Docker images are created a container can be started using it. In order to execute Quake in a container, it needs a display for the graphical output. The following commands can be used to allow docker to use the display of the host machine, but other ways are also possible.

```
1 sudo xhost +
2
3 export DISPLAY=:0.0
```

Then the Docker container can be started with:

```
1 docker run -it --env DISPLAY=unix$DISPLAY --privileged --volume /tmp/.
  X11-unix:/tmp/.X11-unix --name NameOfContainer --network none --
  runtime=nvidia --gpus all NameOfImage
```

This will start the container based on the image that was created from the Dockerfile and attach it to the console. When container is up and running, it is time to create a Tap device with the IP address of "192.168.10.5" and connect it to the docker container. Then the NS-3 has to be started to establish a connect from the Docker container to the local network and the Raspberry Pi. Then Quake can be started in the container with the command:

```
1 sudo LD_LIBRARY_PATH=/OpenUVR/sending/ffmpeg-build/lib /home/root/bin/  
ioquake3/ioquake3.x86_64 +set fs_game mymod
```

This starts the game in the container with the modified game clock. The game then should be displayed on the display of the host machine and on the Raspberry Pi OpenUVR needs to be started with:

```
1 sudo ./openuvr <encoding_type> <network_type>
```

After all these steps the game should be displayed on the host machine and the Raspberry Pi.

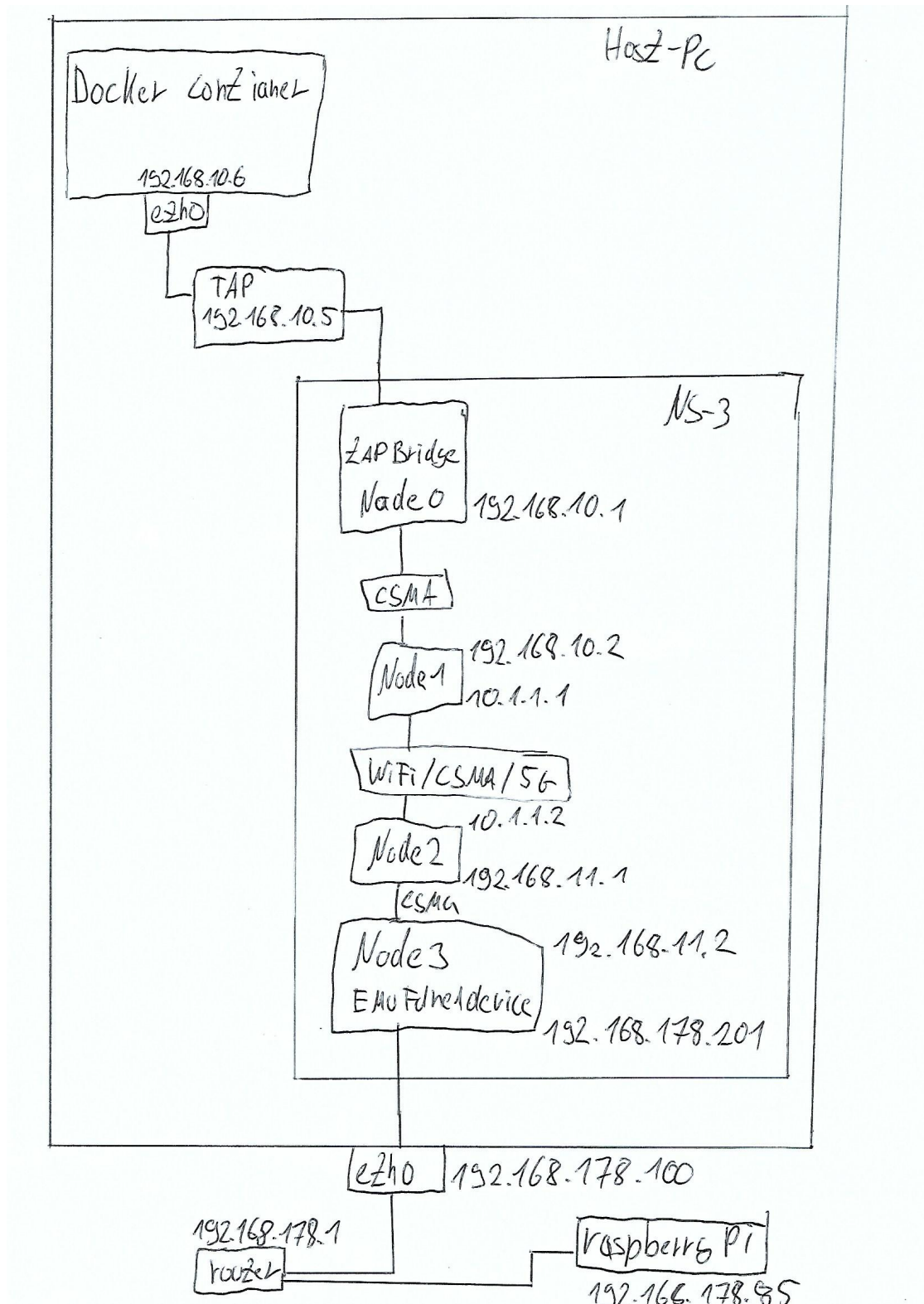


Figure 3.3: Complete network setup




Aktiv	Netzwerk 	Subnetzmaske 	Gateway 
<input type="checkbox"/>	10.1.1.0	255.255.255.0	192.168.178.201
<input type="checkbox"/>	192.168.10.0	255.255.255.0	192.168.178.201
<input type="checkbox"/>	192.168.11.0	255.255.255.0	192.168.178.201

Figure 3.4: added static routes in router

4 Measurements and evaluation

For the measurements a similar setup like described in the OpenUVR paper was used. Two monitors were setup next to each other. One displaying the host machine and the other one displaying the Raspberry Pi. They both are connected to a router via Ethernet to reduce latency in the connection outside the simulation. Then OpenUVR, the NS-3 simulation and the Docker container are started up like described before and then the game should be displayed on both displays one instance from the Docker container on the host machine and the other on the Raspberry Pi. The modified game clock of Quake should be displayed on both displays. Then a photo can be taken of the setup and the time difference between the two clocks on the photo can be calculated. The time difference is the latency of the whole setup and could be compared to results of different NS-3 simulations. However, in this setup the image of the Raspberry Pi is heavily corrupted. In the current state is not possible to collect any representative data from this setup.

In OpenUVR an I-Frame is send, followed by multiple P-Frames. An I-Frame is a complete frame from the game, while P-Frames only has information about the changes between the previous and the next frame. If a P or I frame gets dropped all following frames will be corrupted. OpenUVR should adjust itself to the connection it is using, but the latency and packet loss via NS-3 might be too high to enable a stable connection.



Figure 4.1: testing setup



Figure 4.2: Raspberry Pi image quality

5 Conclusion

Even though it is possible to create a simulation with NS-3 that OpenUVR runs with, this setup does not deliver representative data to collect and evaluate OpenUVR in different network scenarios. The main reason for this probably is the slow simulation speed of NS-3. A host machine with higher computing power might deliver better results, but even then, NS-3 might be too big of a bottleneck to make this approach work. Another approach might be to give all OpenUVR packets a timestamp and calculate the latency with them. Still the packetloss probaly distorts the results. The best thing to do would be to fully simulate the scenario or setup a real network. The realtime simulation might not be the best approach for these measurements.

List of Figures

3.1	Quake new game clock with ms	16
3.2	subnets of ns-3 simulation	17
3.3	Complete network setup	26
3.4	added static routes in router	27
4.1	testing setup	29
4.2	Raspberry Pi image quality	29

List of Source Codes

[GitHub/bachelorsthesis](#)

[GitHub/OpenUVR](#)

[GitHub/FFmpeg](#)

[GitHub/ioq3](#)

Bibliography

- [1] A. Rohloff, Z. Allen, K.-M. Lin, J. Okrend, C. Nie, Y.-C. Liu, and H.-W. Tseng. Openuvr: an open-source system framework for untethered virtual reality applications, 2021.
- [2] A. Schade. Upcoming disruption of service impacting unreal engine users on github. <https://forums.unrealengine.com/t/upcoming-disruption-of-service-impacting-unreal-engine-users-on-github/1155880?page=3>, 2023.