Remote video monitoring system based on VR

Yangzhou University

A thesis submitted for the partial fulfillment of the requirements for the degree of Master of Science in Engineering Technology Electronics-ICT

Sebastiaan Poesen en Marijn Ferrari

2016-2017

Contents

List of Figures

List of Tables

Abstract

The goal of this project is to develop a remote video monitoring system based on virtual reality technology. The multi-channel video streams can be accessed through a smart phone and shown on VR-glasses. In this case the smart phone is acting as a client to receive the multi-channel video streams. The video streams are encoded in H.264 packets and transmitted via the real-time transfer protocol (RTP). According to the gesture, the user can choose the various video channels to display on the VR-glasses. At the same time the camera's field of vision follows the user's head movement. Due to the time limitation, the gesture recognition and VR-compatibility had to be discarded from the project.

Chapter 1

Introduction

Firstly, there will be a brief description of each chapter. Then this chapter introduces the concept of Virtual Reality, explain its increasing importance to the society and why it is implemented in this project. Secondly the objectives are recited based on the given requirements. Finally the materials and methods to achieve these objectives are discussed.

1.1 Outline

In this paragraph the content of each chapter will briefly be discussed.

Chapter 1: Introduction

The first chapter gives a short introduction about Virtual Reality. Next it summarizes the objectives and the hardware plus the software used to develop the system.

Chapter 2: Theory

The second chapter will give a theoretical background on H.264 and the RTP-protocol. A good understanding of the H.264 compression method is needed to set the various parameters in the encoding program.

Chapter 3: Framework

In this chapter an overview of the system is given. It is divided in a software and a hardware part. It will not go in detail, just offer basic understanding of the system's structure.

Chapter 4: Modules

The fourth chapter consists of 4 sections: the H.264 video encoder running on the ORANGEPI, the H.264 receiver-streamer running on the NANOPI, the servo control with the server program running on the NANOPI and clients on the ORANGEPI and smartphone and finally the Android application which receives/plays the video data and sends its device orientation to the server.

Chapter 6: Conclusion

Summary and future improvements regarding the project.

1.2 Virtual Reality

Nowadays everyone has heard of the concept of virtual reality (VR), not to be confused with augmented reality (AR). Virtual reality recreates a real-life environment or situation while AR adds virtual elements to the existing reality. Although VR has been around for a couple of decades, it is only in the last couple of years that it has steadily increased in popularity [1]. In the beginning the main focus used to be gaming and entertainment. Today it has found its way into other branches of the industry. That is what this project is about, using VR in a functional way instead of entertainment. Imagine you can lose all the monitors, keyboards, controls in a standard security-setup. The only thing you need is your smartphone and a VR-glasses, no need for security rooms because the device is portable. The smartphone simply connects to a server, the server acts as a buffer for each video stream. The user can choose a particular stream using a set of predefined gestures. Additionally the camera moves according to the user's head movement so in combination with the VR-goggles it gives the feeling as if the user is in the room.[2].

1.3 Objectives

The initial system requirements contain four parts:

- Dynamic image acquisition and encoding unit with a resolution 640x480, at the same time the CPU can change the camera angle in two degrees of freedom (pan and tilt) by controlling two servo motors
- Four channel video convergence and transmission unit: streaming media server using RTP control to transmit multi-channel H.264 video

- Gesture recognition unit uploads the result to the smartphone via Bluetooth
- Smart phone client receives and decodes the video stream that can be watched through VR-glasses, according to the gesture, the device selects the corresponding channel

1.4 Materials and methods

The tools used to achieve the objectives discussed in the previous paragraph can be divided in hardware and software/libraries.

1.4.1 Hardware

The hardware used in the development of the system has been made available by Yangzhou University. Following boards or modules are used:

- Orange Pi Lite
- Nano Pi M2
- PCA9685

Orange Pi Lite

The ORANGEPI is an open-source single-board computer. The board is equipped with an ARM Cortex-A7 Quad-core processor and uses the AllWinner H3 SoC. It only supports internet connection via Wifi, no Ethernet ports are available. It can run Android Lubuntu, Debian or Raspbian, in this case Debian is installed as the operating system. The ORANGEPI is used to acquire and encode the image data because the chip supports hardware H.264-encoding. The board is released mid 2016, hence the community is still small in comparison to the Raspberry Pi community. This definitely prolonged the development of the system because there is little support online available [3].

Nano Pi M2

The NANOPI is also an open-source ARM board released by *FriendlyARM*. It is equipped with a SAMSUNG CORTEX-A9 S5P4418 and runs a Debian distribution of Linux. In size it is only two thirds the size of the Raspberry Pi. Compared to the ORANGEPI this board is well supported and is easily operable. The NANOPI operates as the buffer between the ORANGEPI and the smartphone-client. It receives and retransmits the video data and serves the smartphone's orientation coordinates to the ORANGEPI [4].

PCA9685

The PCA9685 is a 12-bit PWM/servo driver which can generate up to 16 signals and is controlled via I2C. Each output has its own 12-bit resolution fixed frequency individual PWM controller. All the outputs operate at the same PWM frequency which is programmable from 40 Hz to 1000 Hz. In servo applications the frequency is typically between 50-60 Hz, in this system it operates at a frequency of 60 Hz.

The software is developed on a laptop with standard performance running Linux as OS. To test the Android application a Samsung Galaxy S5 and Samsung Galaxy S7 are used.

1.4.2 Software/Libraries

All the applications except the Android application are written in C(++). The Android application is written in Java. Following libraries are used to implement the objectives described in Section 1.3:

- **Video4Linux2** was created to standardize interfaces related to TV and radio. This library makes it possible to communicate with a large range of devices without requiring the necessary knowledge about the specific drivers [5].
- **Vencoder** is part of the Allwinner media codec library. Implementing a range of hardware encoders and decoders.
- **Live555**, recommended by professor Zhou, is an open-source library for multimedia streaming. This library, written in C++, can be used for a wide range of applications. Such as streaming and receiving of a large number of audio and video codecs, supporting RTP, RTSP, RTCP and SIP protocols. Live555 can run on multiple operating systems, each using a specific compiler. To deploy the source code on the ORANGEPI lite and NANOPI M2, arm-linux-gcc is used to crosscompile.
- **Android Streaming Client** (ASC) (and Efflux): is a library provided on GitHub. The library supports RTP over UDP as transport protocol and decodes H.264 image data. ASC makes use of the *efflux*-library to create the underlying RTP-session.

- **WiringOP** is a library provided by *Zhao Lei*, that can be found on GitHub. It is a modified version of WiringPi, compatible with the ORANGEPI, used to access the GPIO's.

LIVE555 is used to stream content from the ORANGEPI to the NANOPI as well as from NANOPI to the smartphone-client.

Chapter 2

Theory

The images from the USB-camera are coded by the CPU using the H.264 video compression standard. Next the coded stream is wrapped in RTP-packets and sent over WIFI to the server.

2.1 H.264-compression

H.264, also known as MPEG-4 AVC (Moving Picture Experts Group - 4 Advanced Video Coding), is one of the more popular video encoding codecs for the time being. Despite the fact that every year, researchers come up with many new and innovative compression techniques, video encoding applications keep using a set of standard compression methods[6]. These standards improve the interoperability between different manufacturers of encoders and decoders. In 2013 the successor of H.264 was published, H.265/HEVC (High Efficiency Video Coding). In the scope of this project H.264 is used for the reason that the ORANGEPI's chip provides hardware H.264-encoding. In addition to this, the H.264 compression is better documented and supported than its successor [7].

2.1.1 Profiles and levels

The main goal when developing H.264 was to create a relatively simple compression method with the potential to choose from different profiles and levels, respectively sets of algorithmic features and performance classes. The H.264 compression method contains seven profiles, each of which is designed for a specific class of applications.

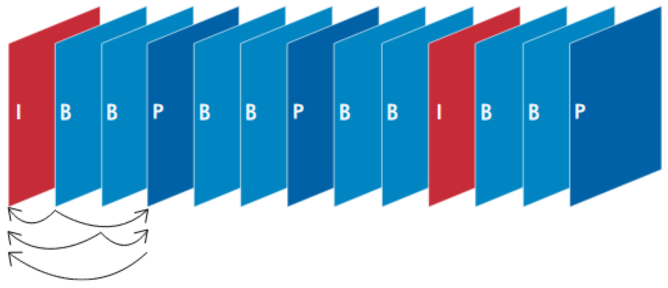


Figure 2.1: Sequence A-,B- and P-frames. Arrows point to which frames the P-and B-frames refer.

2.1.2 Frame types

Different profiles use different types of frames. There are three types of frames:

- I-frame
- P-frame
- B-frame

An I-frame (intra frame) can be decoded independently without any reference to other frames. Every pixel has to be encoded so it consumes much more bits, however it is less likely to cause artifacts in the image.

A P-frame (predictive inter frame) references to previous I/P-frames. They require less bits than I-frames but are sensitive to transmission errors because they rely on earlier P and I frames.

A B-frame (bi-predictive inter frame) is similar to a P-frame except that beside it references to previous frames it also makes reference to future frames. So it makes reference in both directions.

When a videos stream is received, the decoder always needs to start the decoding process with an I-frame. P/B-frames must be decoded with their corresponding key frame, see Figure 2.1. The *baseline* profile only uses I and P frames.

2.1.3 Image compression

Video data can be compressed within an image frame (intra-frame) and between a series of frames (inter-frame). Intra-frame (IAF) compression relies on eliminating redundant information from the image. This will affect the image resolution. On the other hand there are inter-frame (IRF) methods, ie. difference coding. This coding method compares different frames and only encodes the data that has changed in respect to the reference frame and thus reduces the amount of pixel values that need to be encoded. To further compress the image this method can be used on blocks of pixels (macro-blocks) instead of individual pixels. There is one major problem with this method, if there is a lot of motion in a video the data will not be significantly reduced. To solve this, methods like block-based motion compensation are used. Logically, two consecutive frames contain a lot of data which is the same. This method divides a frame into separate macro-blocks and then looks for a matching block in a preceding reference frame. If there is a match, instead of coding the entire macro-block, only the position of the block is encoded as shown in 2.2. It takes less bits to code the motion vector than the actual content of the block.

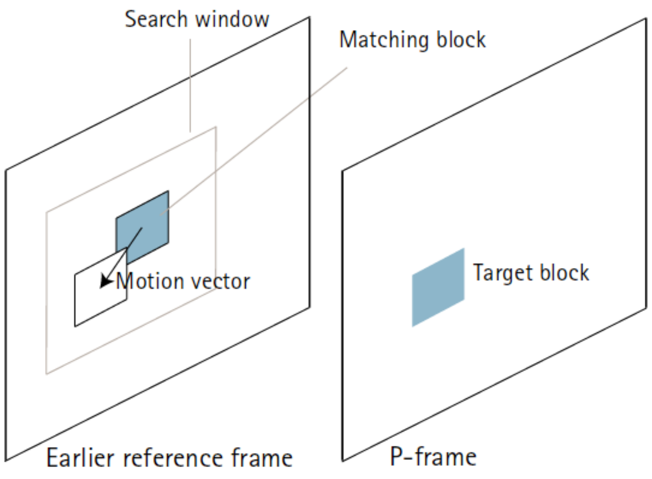


Figure 2.2: The motion vector determined by looking at the preceding frame.

2.1.4 Improvements H.264

H.264 introduced an improved intra prediction scheme for encoding I-frames. It reduces the size of I-frames even more by searching for matching pixel values in the already encoded block. Extrapolating the encoded pixel values that already have been encoded, it reduces the bit size significantly, see Figure 2.3. The I-frames of an H.264 stream have a much smaller size than the key-frames of a Motion JPEG(Joint Photographic Experts Group) stream. Besides the new intra prediction scheme, the block-based motion compensation has also been improved. The encoder is able to look for matching blocks in specific areas of reference frames. Also the block size and shape can be defined to improve the match. If no match is found in a reference frame, intra-coded macro-blocks are used. The high flexibility of the H.264-compression algorithm helps to compress the video in the most efficient way for specific applications.

The H.264 encoding is handled by a library available on *Github*. The library provides an H.264-encoder that uses the system-on-chip to encode the image data received from the camera. Initially it has been written to be used on the ORANGEPI PC but it is also running on the ORANGEPI LITE [8].

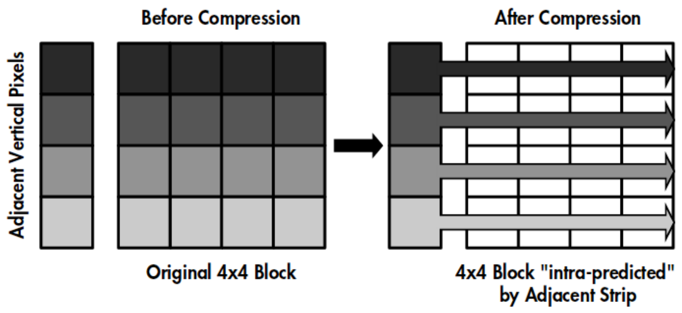


Figure 2.3: Intra prediction technique

2.2 RTP-protocol

The H.264-encoded frames are packed and sent via RTP, it is a standard specified in RFC3550[1], for a detailed explanation it is advised to read this standard. The real-time transfer protocol (RTP), as the name suggests, provides data transmission, such as audio and video, over the network in real-time. The RTP-payload format for H.264 packets is specified in RFC6184(Request for Comments) [2]. This paper will not elaborate further on this subject because it is out of the scope of this project. The entire RTP-part is handled by the LIVE555-library which will be explained in the next paragraph.

[1]https://tools.ietf.org/html/rfc3550
[2]https://tools.ietf.org/html/rfc6184

Chapter 3

Framework

The system can be divided into three parts: ORANGEPI with camera-setup, NANOPI-server and the smartphone-client. The ORANGEPI and the smartphone both act as clients to the NANOPI which operates as a buffer for the video streams. The ORANGEPI collects the image data from the camera and encodes it into H.264 packets. Using the LIVE555 library the packets are sent over RTP to the NANOPI. The NANOPI receives and retransmits the RTP packets to the smartphone on which the video stream can be viewed. The use of the NANOPI as a server, was one of the original requirements of this project. When the orangepi, NANOPI and the smartphone are connected to a single network, the use of the NANOPI is not necessary. It can be used to save multiple streams. At the same time the phone's orientation is determined by the gyroscope sensor [9]. A custom UDP-protocol sends the control-data to the NANOPI on which runs a server that saves the last received orientation. The ORANGEPI can send a *GET*-message to the server and he will reply with the orientation of the smartphone. Currently the ORANGEPI does not provide any PWM-pins hence the signal is generated by a PCA9685[1] controlled via I2C. The complete framework is shown in Figure 3.1. A remark, this overview shows two ORANGEPI's connected to the NANOPI. Due to the limited time, the use of two cameras is not supported yet.

3.1 Software overview

So far, the hardware part of the framework is described. This section will briefly introduce the different software and libraries that are implemented in the system. Chapter 4 will discuss the different software applications in more detail. Figure 3.2 provides a basic overview. Also the software structure can be separated in three parts: software running

[1]Adafruit 16-Channel 12-bit PWM-Servo driver

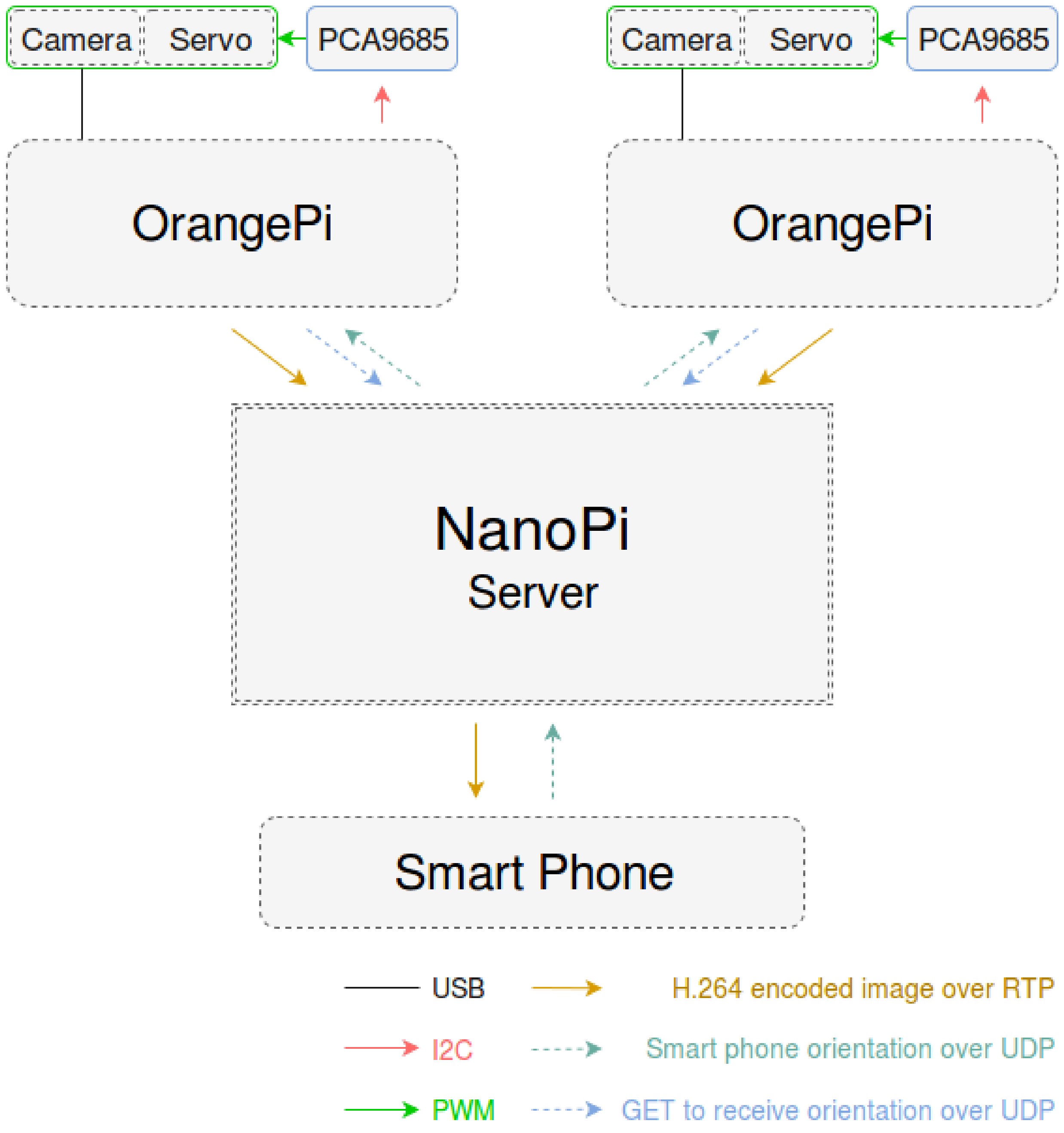

Camera
Servo
PCA9685
Camera
Servo
PCA9685
OrangePi
OrangePi
NanoPi
Server
Smart Phone
USB
H.264 encoded image over RTP
I2C
Smart phone orientation over UDP
PWM
GET to receive orientation over UDP

Figure 3.1: Framework of system

on the Orange Pi Lite, Nano Pi M2 and the smartphone application. Furthermore the applications on each device can be divided into two groups. One group handles the video streaming and the other one controls the angle of the camera with the servo motors.

Orange Pi Lite

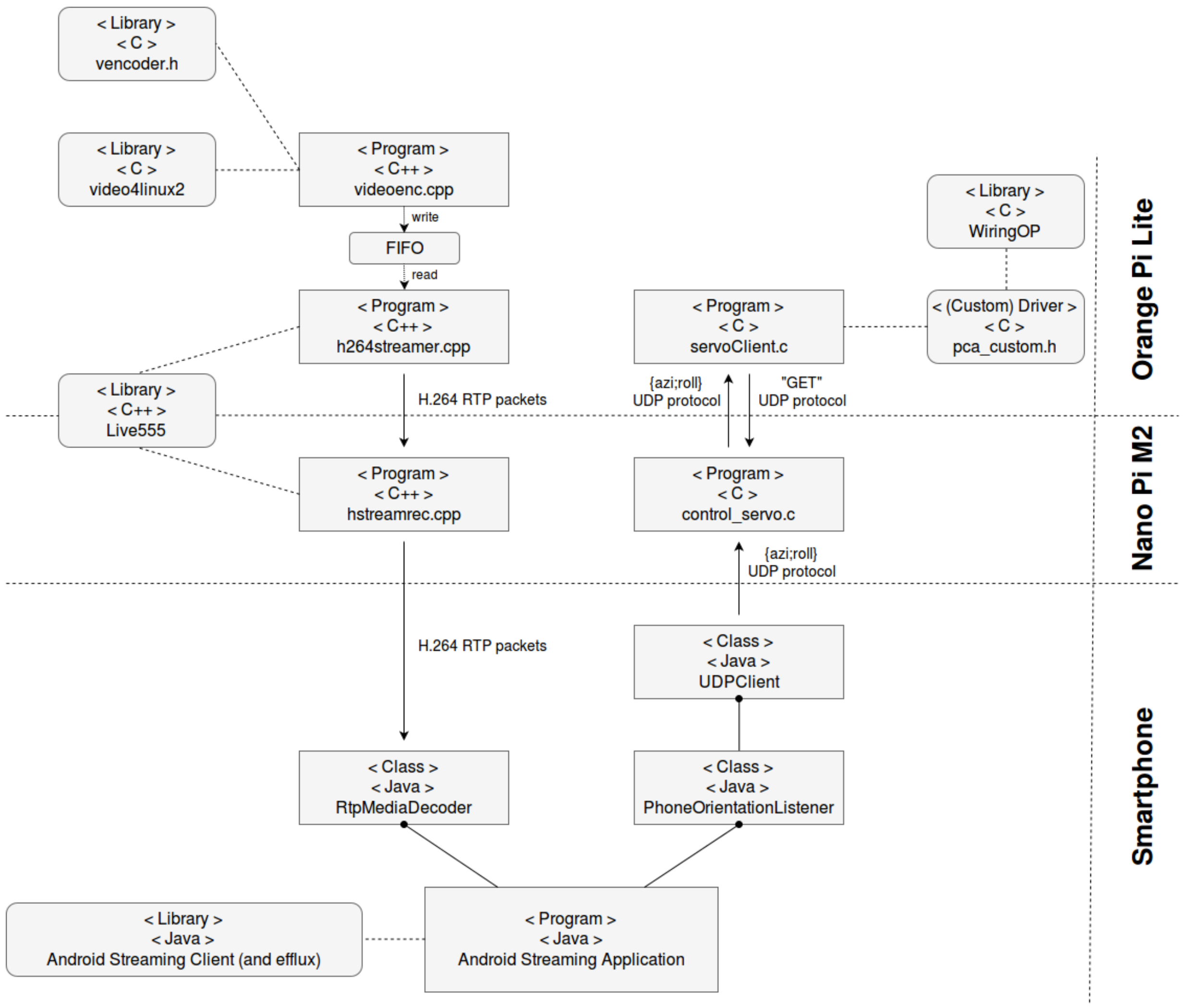
- videoenc.cpp: program to acquire and encode image data from the USB-camera. It uses the video4linux2 and, vencoder.h library
- h264streamer.cpp: streams H.264 via RTP over UDP.
- servoClient.c: sends a message to the server and the NANOPI will send the latest orientation of the smartphone.

Nano Pi M2

- hstreamrec.cpp: receives the H.264 RTP packets from the h264streamer.cpp, the data is written to a buffer and the streamer-thread reads and streams the video data. The streamer-thread and h264streamer.cpp work exactly the same with the only difference that the program is rewritten to a thread.
- control_servo.c: awaits for messages from both ORANGEPI and smartphone, replies with coordinates on receive of a certain message(GET).

Smartphone

- Android streaming application: the device only runs one application but calls different objects, the *RtpMediaDecoder* object handles the decoding, unpacking and visualization of the video stream while the *PhoneOrientationListener* and *UDPClient* acquire and send the device's orientation to the server.



< Library > < C > vencoder.h
< Library > < C > video4linux2
< Program > < C++ > videoenc.cpp
write
FIFO
read
< Program > < C++ > h264streamer.cpp
< Library > < C++ > Live555
H.264 RTP packets
< Program > < C > servoClient.c
< Library > < C > WiringOP
< (Custom) Driver > < C > pca_custom.h
{azi;roll} UDP protocol
"GET" UDP protocol
Orange Pi Lite
< Program > < C++ > hstreamrec.cpp
< Program > < C > control_servo.c
{azi;roll} UDP protocol
Nano Pi M2
H.264 RTP packets
< Class > < Java > UDPClient
< Class > < Java > RtpMediaDecoder
< Class > < Java > PhoneOrientationListener
< Library > < Java > Android Streaming Client (and efflux)
< Program > < Java > Android Streaming Application
Smartphone

Figure 3.2: Software overview of the system

Chapter 4

Modules

4.1 H.264 video encoder

As mentioned before the ORANGEPI LITE makes use of an Allwinner H3 processor with an embedded H.264-encoder. The following library [**Rosimildo**] implements the encoder as well as Video4linux2. Video4Linux2 is used to initialize the camera and to obtain the frames. The following paragraphs describe the encoder library and the parameters.

4.1.1 Framework

First of all the configuration parameters for the encoder and the camera are set. The YUV420 colorspace is required for most compression standards. YUV splits the color information in a luminace and two chrominance components. The most important component is luminance(Y), therefore it should have the highest sample rate. The chrominance components can be sampled at half the rate without a noticeable difference. YUV420 samples the U- and V- components with half the Y-rate in both the horizontal and vertical direction. This results in a lowered bandwidth with an almost unnoticeable difference in quality, hence the preference of this colorspace. Another colorspace, YUV444 samples the three components with an equal rate [10].

After the initialization, three threads are started, namely the encoder-, the camera- ant the out_writer-thread. The following diagram describes the flow of the program. The camera-thread will wait until *Camera.c:CameraGetOneframe* returns, after a timeout or until the time per frame expires. After the return a callback is executed. If no frame is queued the encoder thread will wait until the callback unlocks the wait in this thread. *Videoenc.cpp:processInBuffer* will encode the frame and queues it in the buffer of the

Streamwriter. The *Streamwriter* will wait until data is available. Firstly the length will be written to the FIFO and then the buffer itself. Otherwise the FIFO-reader, in *DeviceSource*, reads the entire buffer. The frames will be incorrectly segmented. As a result, the FIFO overflows and the reading process slows down.

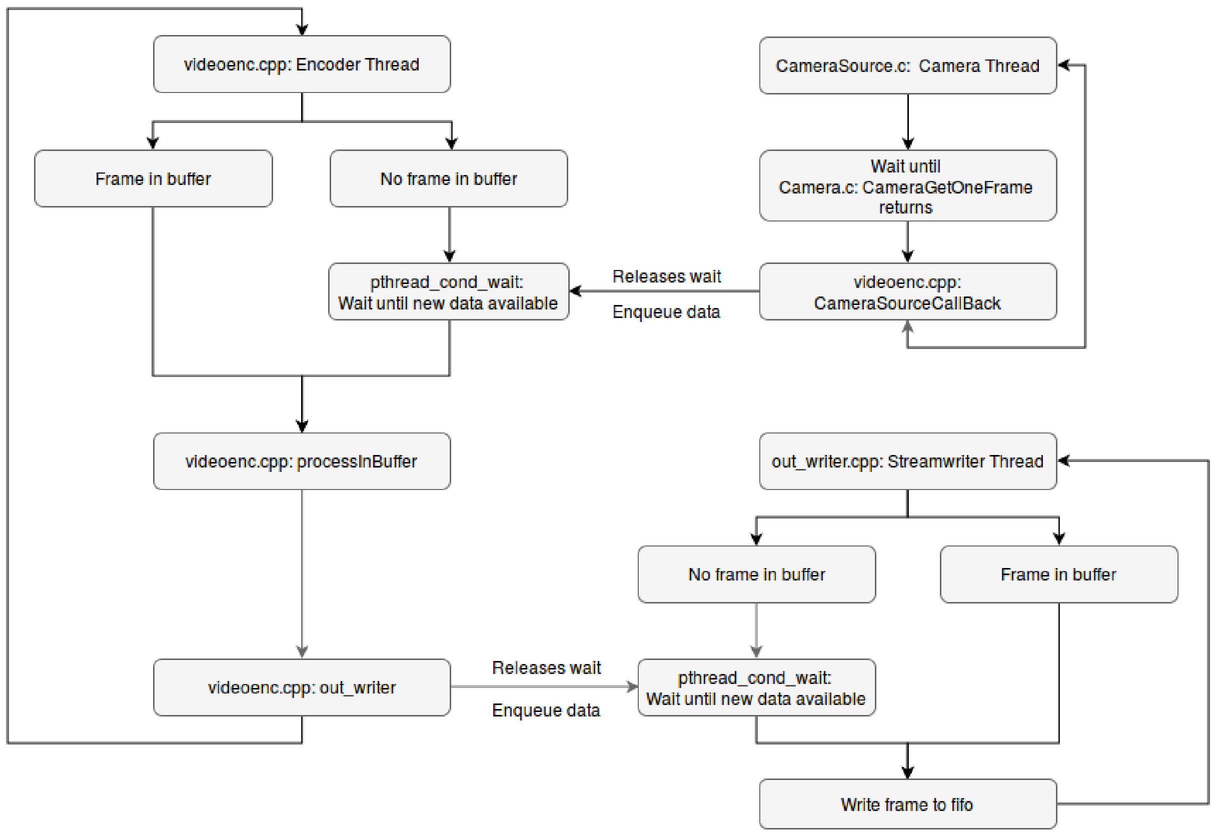

videoenc.cpp: Encoder Thread
Frame in buffer
No frame in buffer
pthread_cond_wait: Wait until new data available
Releases wait
Enqueue data
CameraSource.c: Camera Thread
Wait until Camera.c: CameraGetOneFrame returns
videoenc.cpp: CameraSourceCallBack
videoenc.cpp: processInBuffer
out_writer.cpp: Streamwriter Thread
No frame in buffer
Frame in buffer
videoenc.cpp: out_writer
Releases wait
Enqueue data
pthread_cond_wait: Wait until new data available
Write frame to fifo

Figure 4.1: Flowchart H.264 Encoder.

4.1.2 Parameters

Profile

The VENCODER.H library provides three profiles (baseline, main, high). Baseline is the most primitive profile of the three. As mentioned earlier, B-frames are not implemented. The main- and high profile use the CABAC (Context-adaptive binary arithmetic coding) algorithm, instead of CAVLC (Context-adaptive variable-length coding), to encode the frames[1]. CABAC is more efficient then CAVLC. The increased complexity involves an

[1]Main- and highline also support CAVLC.

improvement of video quality and compression, also a increase of en- and decoding time [11].

Quantization level and mode

The quantization level (Q_p) is inversely proportional to the quality of the encoded image. Qp controls the amount of spacial detail preserved. If Q_p is 0 almost all the details are maintained. When Q_p increases the quality drops and some distortion is introduced [12].

There are two modes: constant quality (CQP) and constant bitrate (CBR). CQP will encode all images with the same level of detail, in other words the quantization level is constant. CBR tries to keep the bitrate constant by varying the quantization levels between a minimum (Q_{min}) and a maximum (Q_{max}). B-frames are based on the previous frame. Logically if much motion between frames occurs, the size will increase. To limit the variation in bitrate, the quality is lowered. CBR was not implemented properly in the encoder library, therefore CQP-mode is used. Regarding this project, there is not much difference between the modes, because the limited motion prevents strong bitrate fluctuations [**H264RateControl** , 13]

Keyframe interval

One frame in every key frame interval is an I-frame. As mentioned before, B- and P-frames are based on the previous key frame. During a stream, there is a possibility that the key frame is not properly received. Logically, this interval can not be to large, otherwise the stream will be distorted for a significant amount of time.

Level

The level provides a set of constrains, according to the maximum specifications of the decoder. The bitrate and the decoding speed (concerning macroblocks) are limited. This results in an maximum frame rate and resolution. This project makes use of Androids phones capable of playing 1080p at 30 frames/s, well under the limit of the specifications of the ORANGEPI LITE and the network. Therefore, the preset level of 3.1 is maintained.

4.1.3 Determination parameters

This paragraph discus the chosen settings for the encoder.

Bandwidth(kB/s)			
Key frame interval	Baseline	Main	High
1	146.98	131.56	125.83
2	92.09	81.98	78.99
3	70.05	59.50	57.4
5	47.67	42.58	40.33
10	34.96	34.84	33.34
25	28.28	26.11	23.16
100	22.80	20.72	20.33

Table 4.1: Bandwidth in function of the key frame interval and profiles.

Profile and key frames

The following paragraph explains the differences between the three profiles in this project. An experiment has been conduced with a fixed Q_p of 40 and a frame rate of 20 frames/s. The results are shown in table 4.1.There is an noticeable difference in bandwidth[2] between the profiles. Table 4.2 shows the average frame size in function of the profiles. The relatively large difference between baseline and main/high is because the use of CABAC instead of CAVLC encoding. A small difference in quality can be noticed between baseline and main/high. The delay of high-profile is +-0.5s longer compared to baseline and main (3s). Due to the increased complexity, the encoding and decoding time increases.

Main-profile is preferred because the slightly increased quality with a decrease in bandwidth compared to baseline. High-profile is not compatible with older devices. In this case, the disadvantages outweigh the advantages.

When the key frame interval increases, logically the bandwidth will increase. The video stream also becomes smoother because of the relative increase of reference frames. In order to keep the distortion time to a minimum, caused by an incorrect received key frame, the interval has to be limited. Both 3 and 5 are acceptable values for a large range of frame rates. An interval of 5 is preferred due to a difference in bandwidth of 28,4%.

Frame rate

The frame rate depends on the speed on which LIVE555 can process each frame. The time needed is independent of the size of the frames. There is also no distinction between B

[2]The bandwidth is measured using the nload package

Average frame size (Bytes)			
	Baseline	Main	High
Keyframe	5608.03	4601.15	4558.17
B/P-frame	125.53	114.14	113.82

Table 4.2: Average frame size in function of the profiles.

and I frames, so the key frame interval has no influence. To conclude, if the frame rate becomes lager then 25 frames/s a delay tends to build up. Therefore the maximum frame rate is 25 frames/s.

Quantization level and frame rate

The lower limit depends on the size of the buffer that can be processed in LIVE555 and the available bandwidth. Due to the limitations of this library and the ORANGEPI LITE a buffer size of 32768 Bytes was chosen, a larger buffer results in a bad quality stream. After some experiments a lower limit of 25 was determined. The actual quantization level depends on the available bandwidth.

Table 4.3 represents the bandwidth in function of the quantization and the frame rate with an key frame interval of 5. Screen shots from the stream are showed from figure 4.2 until 4.7. There is a large difference in bandwidth when to video is encode using a Q_p of 25 and 30, the quality is nearly the same. The quality is judged based on visual appearance on the Android phone. Therefore a quantization level of 25 is not recommended. The following images show a slight decrease in quality accompanied by a justifiable difference in bandwidth. The decrease in bandwidth between a Q_p of 45 and 50 is not worth the quality drop.

The bandwidth in function of the frame rate approaches a linear relation. The average frame size, for key- and reference frames is constant. As mentioned before, each frame is sent individually. Therefore an increase in frame rate results in an increase of frequency in which RTP-packets are send.

To determine the actual frame rate and the quantization level, a trade off has to be made based on the available bandwidth.

Bandwidth(kB/s)					
Framerate(frames/s)	5	10	15	20	25
Quantization level					
25	80.1	160.68	226.62	304.23	370.13
30	41,65	83.33	110.15	145.21	183.95
35	18.65	36.32	51.35	63.12	79.18
40	12.36	23.13	31.54	41.65	50.13
45	10.11	18.43	23.51	28.74	33.58
50	8.25	14.68	20.99	25.85	30.65

Table 4.3: Bandwidth in function of the frame rate and the quantization level.

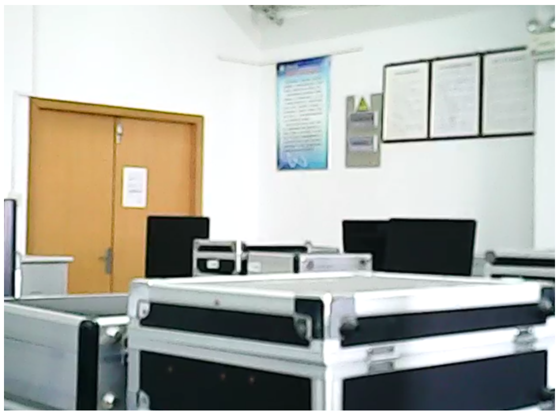


Figure 4.2: Q_p of 25

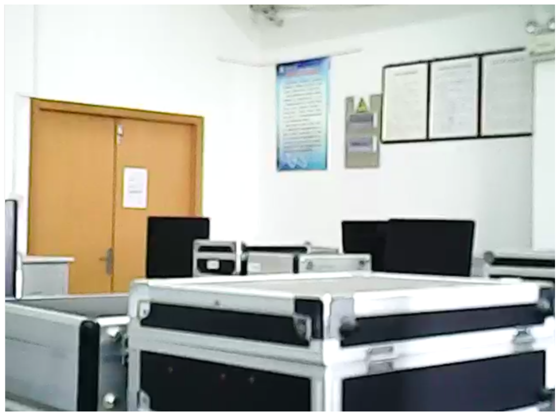


Figure 4.3: Q_p of 30

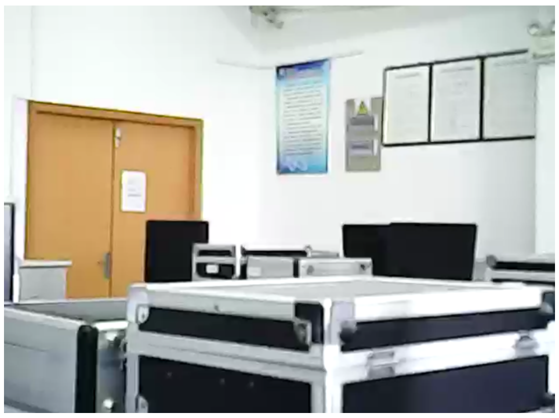


Figure 4.4: Q_p of 35

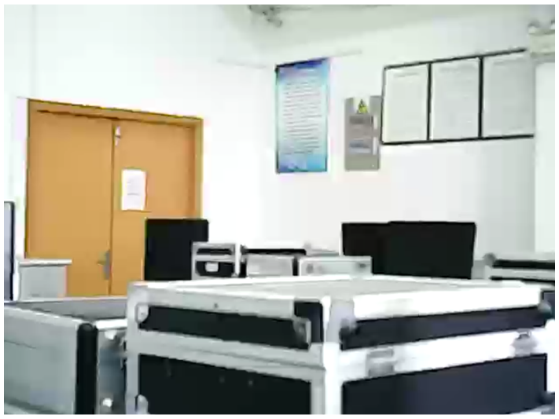


Figure 4.5: Q_p of 40

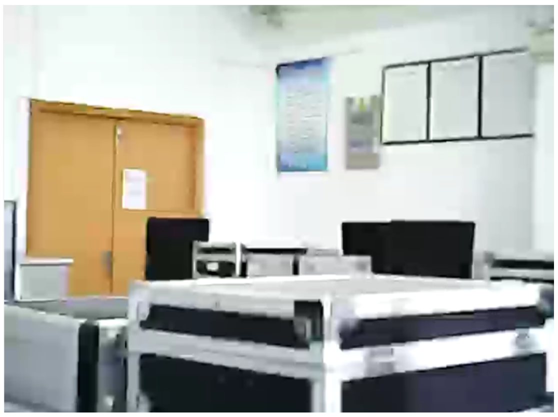


Figure 4.6: Q_p of 45

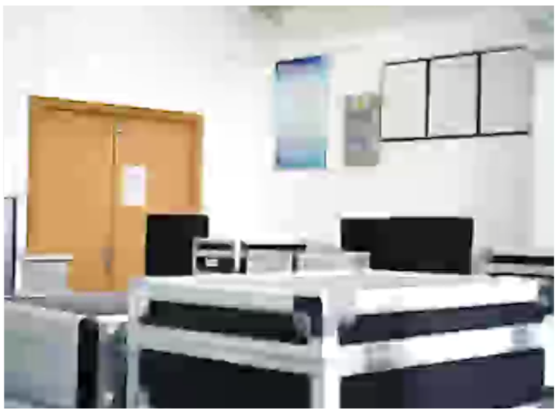


Figure 4.7: Q_p of 50

4.2 H.264 receiver-streamer

This section will explain de *hstreamrec.cpp* that runs on the NANOPI, see Figure 3.2. The program consists of two parts, the streamer-part and the receiver-part, both are based on the test programs offered by the LIVE555-library. First there will be a brief introduction to the LIVE555-library, the most important classes will be discussed. Next the changes that are made in the streamer- and receiver test programs, provided by LIVE555, will be explained. The final part in the section describes the flow in the *hstreamrec.cpp* program and gives an example of how to start a new streamer-receiver pair.

4.2.1 Live555

LIVE555 is an open-source library for multimedia streaming, programmed in C++ (more information see Section 1.4.2). Both streamer and receiver are based on the example programs offered by the library. The streamer is based on *testH264VideoStreamer* and the receiver on the *testMPEG1or2VideoReceiver*. Initially both applications were developed as stand-alone programs but later they are rewritten to run inside a streamer-thread and receiver-thread. So it is possible to receive and stream multiple concurrent streams. A necessary remark, LIVE555 assumes a single thread of execution. It is not built with the intention to be run in different threads. However there are several ways to make the code run in concurrent threads. The first option is to run the library in one thread, and all the other threads communicate with the library only via sockets, by setting global *flag variables*, or by calling *event-triggers*. The second option, and also the option used in the application, is to initialize *UsageEnvironment* and *TaskScheduler* objects for every thread. In the FAQ found on the website of LIVE555 do not recommend to use this configuration, a multi-process configuration is recommended rather than the multi-threaded environment used in this project. Until now no problem has established regarding this issue.[14]

Classes

This paragraph describes the most important classes and libraries used in LIVE555 regarding this project.

- UsageEnvironment-class and TaskScheduler-class
  Accommodates the printing of warning and error messages, scheduling deferred and asynchronous events.
- Groupsock-library
  Provides all the network interfaces and sockets. For example, Groupsock is used to create the sockets, packing RTP-packets with the appropriate header and sending them afterwards.
- LiveMedia-library
  Support for handling a large variery of audio- and videocodecs, both for streaming and receiving a certain codec.
- testH264VideoStreamer.cpp
  The application reads from a H.264 elementary stream video file (video.h264), and streams it using RTP multicast, besides it also provides a built-in RTSP server
- testMPEG1or2VideoReceiver.cpp
  This program reads incoming MPEG-encoded RTP packets and writes the output to "stdout" or a video file

4.2.2 Streamer

The streamer-thread is based on the *testH264VideoStreamer* program provided by LIVE555. A couple of modifications are made to achieve the requirements.

Unicast instead of multicast

In the example program from LIVE555 the custom function *chooseRandomIPv4SSMAddress()* from the class *GroupsockHelper* is used to generate a random multicast IP-address. Hence every device on the network could listen to the broadcast. Replacing this by the standard *inet_addr()*-function found in the *inet.h*-library permits to choose our own unicast

IP-address. For the moment every IP-address in the application is hard-coded so it is necessary to pick a particular IP-addresses within the network.

Only RTP transmission

Next to the RTP-session, the test program also starts a RTSP-server. The RTSP-server can be opened in a web browser to view the stream but it is not relevant in this project.

Buffer-parameter

There is the option to choose the maximum size of outgoing packets. The value has no direct influence on the quality of the video or the speed of transmission. As long as the size of it is large enough to contain the captured frames, the size does not matter. Several tests pointed out that it should not be smaller than the arbitrary value of 50000.

Implementation DeviceSource

The original streaming program is only able to stream data from a static file. It opens the file as a byte-stream file source then a filter is applied to break up the H.264 video elementary stream into NAL-units. Finally the NAL-units are passed to the *H264VideoRTPSink* which streams them to a specific port of an IP-address on the network. To stream a live video feed the class *ByteStreamFileSource* can not be used, LIVE555 provides a template for a MediaSource encapsulating a video input device, called DeviceSource. The provided class is modified so it is able to read directly from a FIFO (first-in-first-out). The received stream is written into a FIFO, DeviceSource reads from it, fills up a buffer and flushes the buffer into the *FramedSource*, as is shown in Figure 4.8.

Multiple concurrent streams

The final requirement in the streamer-program is that multiple streams run simultaneously. Section 4.2.1 explains the different methods available to achieve this goal. At this moment it is only possible to stream multiple files and not live video streams. The reason is that a static file uses *ByteStreamFileSource* instead of *DeviceSource*. The *DeviceSource* class implements a global variable and thus is not thread safe. Each thread creates its own *TaskScheduler*- and *UsageEnvironment* variable.

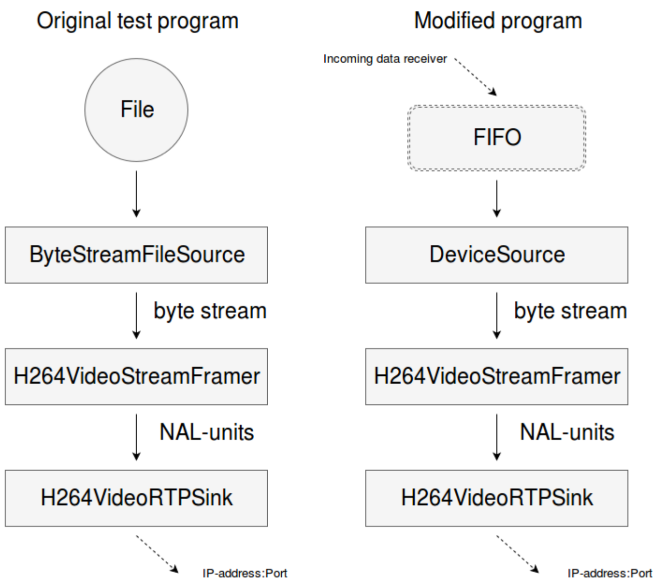


Figure 4.8: Flow inside streamer program

4.2.3 Receiver

LIVE555 does not provide any H.264 video receivers but it contains the classes necessary to build one. The example program *testMPEG1or2VideoReceiver* forms the base of the receiver thread. With some modifications in this class and the *FileSink*-class, it is possible to receive an H.264 stream over RTP.

H264VideoFileSink instead of FileSink

The original receiver only uses a FileSink object to write the extracted frame to a file. When an H.264 stream is to be received the sink needs to add *Source-Specific Parameters* (SPS) in the header preceding the first frame. The SPS define characteristics of the stream, ie. which profile is used. Also each NAL-unit has to begin with the start code: 0x00000001. In order to do this the *H264VideoFileSink* is called, this function is derived from *H264or265FileSink* which is derived from *FileSink* itself, as shown below.

$$FileSink \leftarrow H264or265FileSink \leftarrow H264FileSink$$

Initially the FileSink class is developed to write the received content to a file or the console. Instead of writing to a file, the FileSink is modified to write to a FIFO out of which the DeviceSource can read, as noted in Section 4.2.2.

Other minor modifications

- unicast instead of multicast

  Sometimes the IP-address to receive a unicast stream changes. For unknown reasons it differs between *0.0.0.0* and the IP-address of the host itself

- only receives RTP

  The example receiver also receives multimedia over RTCP, this is not necessary

- *H264RTPSource* class instead of *MPEG1or2RTPSource*

Due to the time limit, the requirement to receive multiple streams at once is not accomplished.

4.2.4 hstreamrec.cpp

Sections 4.2.2 and 4.2.3 describe the modifications in the streamer and the receiver test program. The *hstreamrec* merges both parts together, each running concurrently in their own thread. This section will explain how both threads are called, work together and describe in detail the flow of the application using the code snippet in Appendix A in combination with Figures 4.9 and 4.10.

main method

The first function call of the main method is *config_servers*. This function initializes the receiver-and streamer-struct. The structs hold the name of the FIFO and a port number. The variables are defined inside a struct so they can be passed to a thread as an argument. Next in line are the receiver-and streamer-threads, the threads call respectively the *init_new_receiver* method and the *init_new_streamer* method. Both methods will be described in following paragraphs. The main method ends with a *pthread_join()*-function call. It waits for the specified thread to terminate, otherwise the application will stop when the main-thread is finished.

init_new_receiver

This method will receive the incoming RTP-packets with H.264 encoded image data and write the data inta a FIFO. Figure 4.10 shows an overview of the flow inside this method. First the *TaskScheduler* and *UsageEnvironment* objects will be created. They organize

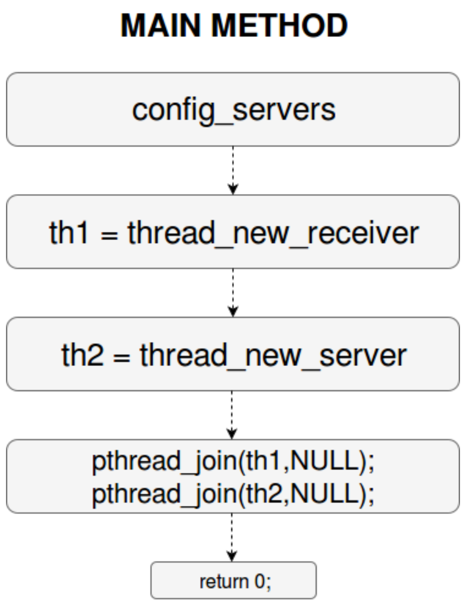


Figure 4.9: Function calls in main method

the chronological execution of events and print warning/error messages throughout the program. The *sessionAdressStr* variable is the IP-address where the receiver listens to. In this case it is declared: 0.0.0.0 because the ORANGEPI streams directly to the NANOPI. Next an H264VideoFileSink is created, this is a derived class of *FileSink* which writes the received data to the FIFO. Finally *rtpGroupsock* and *H264VideoRTPSource* work together to handle the underlying RTP-connection and unpack the received RTP packets.

init_new_server

The structure of this method is very similar to the one of the receiver (Figure 4.10). Here also, the *TaskScheduler* and *UsageEnvironment* are created. The IP-address of the device where it streams to is declared in *destinationAdressStr*. The *rtpGroupsock* handles the underlying RTP-connection and *H264VideoRTPSink* encapsulates the frames in RTP-packets. *DeviceSource* obtains the frame data by reading the FIFO, a complete diagram is given in Appendix B. It also adds a start code prefix for the NAL-unit if necessary and transfers it to a shared buffer. *H264VideoStreamFramer* separates the stream of frames into NAL-units and keeps them in order to maintain the dataflow. In this project only complete frames are read from the buffer. The data is passed to and processed by *H264or5Fragmenter*, an inner class of *H264or5VideoRTPSink*. During processing the following possibilities occur; The buffer contains a new NAL unit. When it is small enough to be send in one RTP-pack, it is delivered entirely to the *RTPSink*. When it is too large, the first fragmented unit(FU)

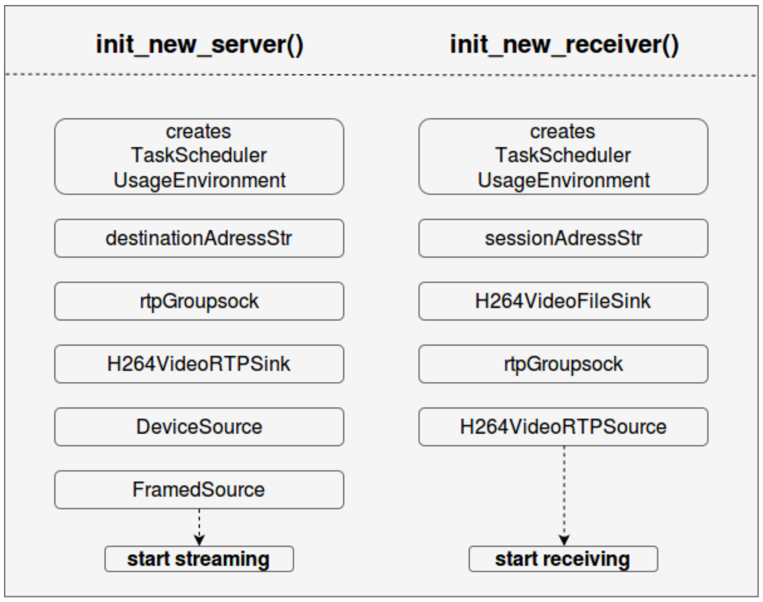


Figure 4.10: Flow inside streamer-and receiver function

is delivered to the sink. The existing NAL header is overwritten by a new NAL header and 1 Byte is added as the FU header. The third and last possibility, the next FU is delivered, two extra preceding header Bytes are added (NAL and FU).

MultiframedRTPSink will handle the packing and sending using *Groupsocket*. If necessary, *H264or5Fragmenter* is recalled, to transfer the next fragmented unit. After a complete frame is send, *FramesSource::afterGetting(this)* is called. This will execute *DeviceSource::doGetNextFrame()*.

4.3 Servo motor control

The servo motor attached to the camera has to move according to the user's head movement. The server, written in C, runs on the NANOPI. The communication between server and client is handled by the User Datagram Protocol (UDP). UDP is chosen as communication method because reliable data transmission is not important as the smartphone sends its orientation 10-50 times a second. Hence data loss will not be noticed. The server receives the control information from the smartphone and needs to update the ORANGEPI accordingly, all of this is shown in Figure 3.1. A custom protocol is developed, the smartphone sends its azimuth and roll in the following order:

$$\{azimuth; roll\}$$

Section 4.4 explains the the method to obtain the device's orientation and send it to the server as it is implemented in the smartphone application. This section will discuss the server-program on the NANOPI and client-program running on the ORANGEPI.[3]

4.3.1 Server-side

The server is constantly listening to incoming messages on port number 6000. When a message is received, it will first be checked using following regular expression: $\{[0-9]*;[0-9]*\}$. If the message satisfies this requirement the whole string will be saved in the control info variable. The server will not reply the smartphone-client when receiving the orientation coordinates. If the message contains the string *GET*, the server will reply the control info variable to the ORANGEPI-client. Every other message that does not apply to one of these two requirements will be discarded.

4.3.2 Client-side

The previous paragraph explains how the ORANGEPI obtains the orientation of the smartphone. This section describes the program to set the position of the servo. According to the datasheet the ORANGEPI is provided with 1 PWM-pin but for unknown reasons it is inaccessible. Hence there is no access to the PWM-pin and two PWM-pins are needed to control the pan-tilt motors, the PCA9685 is used to generate the PWM-signals.

PCA9685

The PCA9685 (see Section 1.4.1) has 16 channels that can generate their own PWM-signals. Each channel is controlled by four registers. Channel 0 for example, is handled by registers 6-9, as shown in Table 4.4. Register 6 and 7, respectively LED0_ON_L and LED0_ON_H, define when the signal transitions from low to high. Most of the times these registers are set to 0. Register 8 and 9, respectively LED0_OFF_L and LED0_OFF_H, define when the signal transitions from high to low. The latter registers are set to different values to change the servo to a specific position. Every channel has a 12-bit resolution so the four most significant bits of LED0_ON_H and LED0_OFF_H are not used. The same conditions apply for the other 14 channels.

PCA9685-library
The ORANGEPI supports no existing C-libraries to operate this module. Hence a sim-

[3]http://www.binarytides.com/programming-udp-sockets-c-linux

Table 4.4: Register summary PCA9685 (incomplete)

Register#(decimal)	Register# (hex)	D7	D6	D5	D4	D3	D2	D1	D0	Name	Type	Function
0	0	0	0	0	0	0	0	0	0	MODE1	read/write	Mode register 1
1	1	0	0	0	0	0	0	0	1	MODE2	read/write	Mode register 2
2	2	0	0	0	0	0	0	1	0	SUBADR1	read/write	I2C-bus subaddress 1
3	3	0	0	0	0	0	0	1	1	SUBADR2	read/write	I2C-bus subaddress 2
4	4	0	0	0	0	0	1	0	0	SUBADR3	read/write	I2C-bus subaddress 3
5	5	0	0	0	0	0	1	0	1	ALLCALLADR	read/write	LED All Call I2C-bus address
6	6	0	0	0	0	0	1	1	0	LED0_ON_L	read/write	LED0 output and brightness control byte 0
7	7	0	0	0	0	0	1	1	1	LED0_ON_H	read/write	LED0 output and brightness control byte 1
8	8	0	0	0	0	1	0	0	0	LED0_OFF_L	read/write	LED0 output and brightness control byte 2
9	9	0	0	0	0	1	0	0	1	LED0_OFF_H	read/write	LED0 output and brightness control byte 3
10	0A	0	0	0	0	1	0	1	0	LED1_ON_L	read/write	LED1 output and brightness control byte 0
11	0B	0	0	0	0	1	0	1	1	LED1_ON_H	read/write	LED1 output and brightness control byte 1
12	0C	0	0	0	0	1	1	0	0	LED1_OFF_L	read/write	LED1 output and brightness control byte 2
13	0D	0	0	0	0	1	1	0	1	LED1_OFF_H	read/write	LED1 output and brightness control byte 3
...	...				...					...	...	...
254	FE	1	1	1	1	1	1	1	0	PRE_SCALE[1]	read/write	prescaler for output frequency
255	FF	1	1	1	1	1	1	1	1	TestMode[2]	read/write	defines the test mode to be entered

ple library is written in C, based on the python library provided by the GitHub-page of *adafruit*.[4] To access the GPIO's and I2C communication of the ORANGEPI, the WiringOP-library of *zhaolei* is used.[5] WiringOP, also provided on GitHub, is a modified version of WiringPi, compatible with the ORANGEPI. The developed PCA9685-library contains two main functions:

1. set PWM frequency

```
1   int setfrequency(int bus, int freq){
2   float prescaler = 25000000/(4096*freq) - 1;
3   printf("prescaler value = %i\n", (int)prescaler);
4   wiringPiI2CWriteReg8(bus,0x00,16); //sleep bit is set to 1
5   wiringPiI2CWriteReg8(bus,0xFE, (int)prescaler);
6   wiringPiI2CWriteReg8(bus,0x00,0);
7   }
```

   On start-up it is necessary to initialize the PWM frequency. The clock frequency is 25 MHz, by setting a prescaler it is possible to choose different clock rates. In the case of servo-control, the value is chosen between 50-60 Hz. In line 4, the sleep-bit *D4* is set to 1 before writing to the prescaler register, see Table 4.4. This is necessary because it is impossible to write to the prescaler register while the oscillator is turned on. After the prescale value is set, the sleep-bit is set back to 0 again.

2. set frequency PWM controller

[4]https://github.com/adafruit/Adafruit_Python_PCA9685
[5]https://github.com/zhaolei/WiringOP

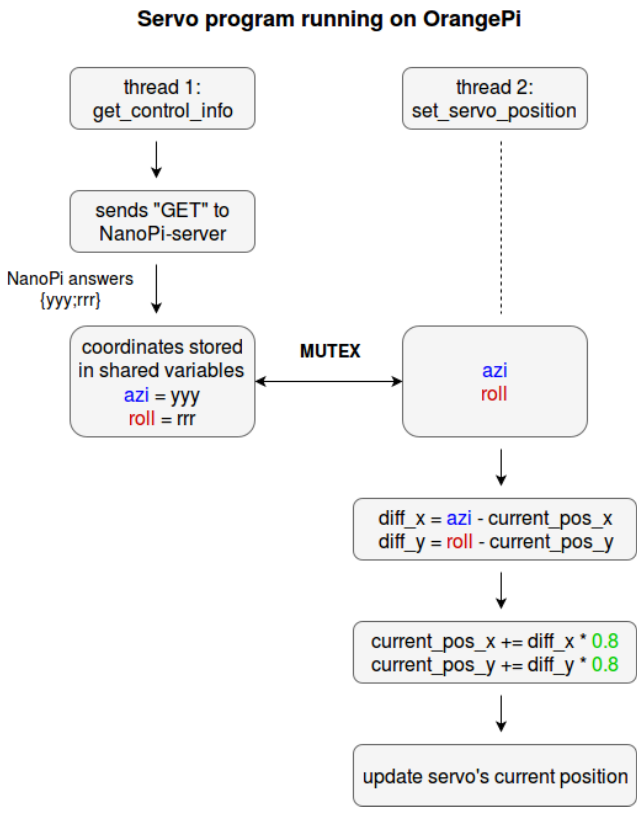
```c
int writepwm(int bus, int pin, int on=0, int off){
wiringPiI2CWriteReg8(bus, LED_ON_L+4*pin , on & 0xFF);
wiringPiI2CWriteReg8(bus, LED_ON_H+4*pin , on >> 8);
wiringPiI2CWriteReg8(bus, LED_OFF_L+4*pin , off & 0xFF);
wiringPiI2CWriteReg8(bus, LED_OFF_H+4*pin , off >> 8);

return 0;
}
```

The duty-cycle of a specific channel is modified by setting 4-registers as is explained in Section 4.3.2. For LED_OFF_L, an AND-operation between *off* and OxFF is used to keep only the 8 least significant bits. The 4 most significant bits, to set register LED_OFF_H, are obtained by shifting *off* 8 bits to the right. The minimum pulse length is 150 out of 4096 and the maximum is 600 out of 4096. LED_ON_L and LED_ON_H are set to 0 by default.

OrangePi client application

The servo client application, also written in C, makes use of the library described in the previous paragraph to communicate with the PCA9685 and control the servo's position. The client obtains the smartphone's orientation by sending the "GET" message to the server. Figure 4.11 shows the flow of the client-program. The constructor initializes two threads, one to get the orientation and one to control the PCA9685 to generate the correct PWM signal. Thread 1 is really straight-forward, every 40 ms it sends the message to the server, the server replies the orientation. The orientation is then stored in shared memory and locked/unlocked using a mutex. Next every 40 ms thread 2 will lock the mutex and read the variables. To smoothen the movement of the servo motor, a technique called easing/ramping is used.[6] The technique is really simple to implement and delivers neat results. First the difference between the current position of the servo and the received coordinates is calculated. Thereafter the difference multiplied with an easing parameter is added to the current position. In the program a value of 0.8 is chosen for this parameter. If the value is too high, it will have no effect and result in shaky movement. On the other hand a too low value results in a constant delay. Both threads execute a while loop that runs every 40 ms. Better results (smoother movement) are achieved if the loops run every 20 ms but then the program crashes, due to the *wiringPiI2CWriteReg8* that takes too long to execute.

[6]http://lab.guilhermemartins.net/2009/08/21/filtering-servo-movements/



Servo program running on OrangePi
thread 1: get_control_info
thread 2: set_servo_position
sends "GET" to NanoPi-server
NanoPi answers {yyy;rrr}
coordinates stored in shared variables azi = yyy roll = rrr
MUTEX
azi roll
diff_x = azi - current_pos_x
diff_y = roll - current_pos_y
current_pos_x += diff_x * 0.8
current_pos_y += diff_y * 0.8
update servo's current position

Figure 4.11: Flowchart of servo position application

4.4 Android application

The smartphone running the android application serves as front-end client in the setup. The android application consists of two parts. One part gathers and unpacks the RTP-packets, decodes the H.264 content and plays it. The other part retrieves the orientation of the phone and sends it over UDP to the server, using a protocol described in Section 4.3. Both parts will be discussed separately in following paragraphs.

4.4.1 Video streaming client

The real-time video data is played by *Android Streaming Client* (ASC), a library provided on GitHub.[15] The library only supports RTP over UDP as transport protocol and decodes H.264 image data. ASC makes use of the *efflux*-library to create the underlying RTP-session. There are two options to handle package arrival:

- **time-window**, which uses an RTP buffer that collects packets for a certain amount of time and pushes them upstream in the right order and at a fixed rate. It is handled by two threads, one thread to collect the incoming packets and the other one to deliver them upstream in the right order.

- **min-delay**, each packet received by the RTP buffer is transfered directly for processing. The packets are only transfered for processing when they are the ones being expected. If a received packet is newer than the one expected it will be stored for later use. A threshold determines how long packets are kept before being discarded.

The time-window method results in bad quality video, many frames are dropped and the image contains blocky artifacts hence min-delay is used. In the configuration file different parameters regarding the decoder can be set. In this case the most important ones are:

- RECEIVE_BUFFER_SIZE_BYTES = 100000 : size of the buffer
- BUFFER_TYPE = min-delay : type of buffer
- NODELAY_TIMEOUT = 1000 : (in ms) maximum delay to keep incoming packets

The library is really easy to implement. In the *OnCreate*-method of the main activity an *RtpMediaDecoder* object is created and started ()as shown in the code below). The decoded video is played in the surface view assigned to the *RtpMediaDecoder*.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
...
// create an RtpMediaDecoder with the surface view where you want
// the video to be shown
RtpMediaDecoder rtpMediaDecoder = new RtpMediaDecoder(surfaceView);
// start it
rtpMediaDecoder.start();
...
}
```

Initially the library could only listen to one port. So the ACR is slightly modified in a way the user can choose between multiple video streams. A *spinner*, populated with different streams, is used to change the port where the application is listening to. A *Stream*-object only contains a name, a port number and its corresponding getters/setters. The application consists of one activity, as shown in Figure 4.12. On the left the *spinner* where the stream can be chosen and on the right there is the surface view in which the video is played. Below the *spinner*, the phone's relative orientation coordinates are displayed. How they are determined will be discussed in the next paragraph.

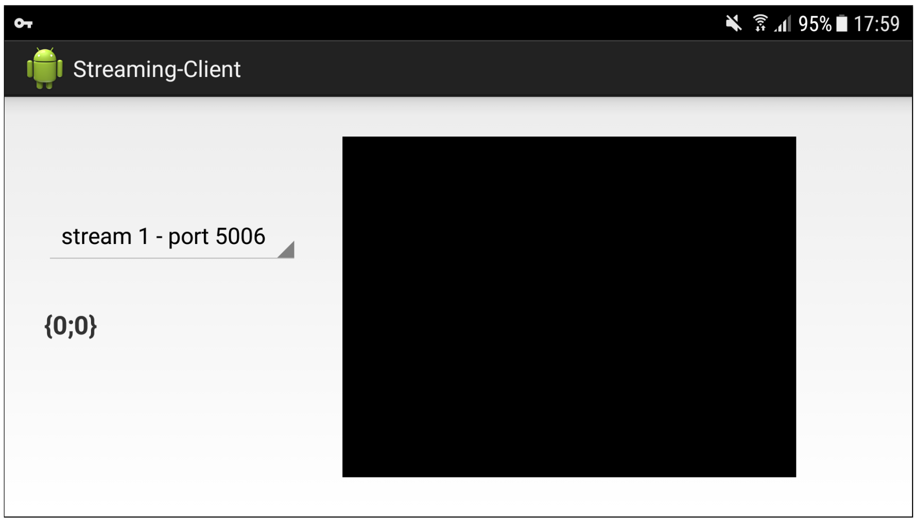


Figure 4.12: Screenshot application home screen

Both *efflux* and *Android Streaming Client* are licensed under the Apache License, Version 2.0.

4.4.2 Orientation listener and sender

The other function of the android application is to determine its orientation and send it over UDP to the NANOPI-server. In the *OnCreate* method of the main activity a *PhoneOrientationListener* (POL) is created. In the constructor of this object a UDP connection with the server and sensor listener are initialized. On change, the sensor values are updated and send to the server.

$$MainActivity \rightarrow PhoneOrientationListener \rightarrow UDPClient$$

Sensor listener

The device's position relative to the device's frame of reference is determined by using the motion sensors of the android device.[16] The sensor TYPE_ROTATION_VECTOR is used in this case. It returns an array of sensor values for each SensorEvent. During a single sensor event the sensor returns an array consisting of the four following values:

- SensorEvent.values[0]: Rotation vector component along the x axis ($x * sin(\theta/2)$)
- SensorEvent.values[1]: Rotation vector component along the y axis ($y * sin(\theta/2)$)
- SensorEvent.values[2]: Rotation vector component along the z axis ($z * sin(\theta/2)$)

- SensorEvent.values[3]: Scalar component of the rotation vector ($cos(\theta/2)$)

It is possible to calculate the orientation of the device out of the rotation vector in simply 2 steps:

1. **getRotationMatrixFromVector(float[] R, float[] rotationVector)**, helper function to convert rotation vector to a rotation matrix
2. **getOrientation(float[] R, float[] values)**, computes the device's orientation based on the rotation matrix

The latter function returns following values:

- values[0]: *Azimuth*, angle of rotation about the z-axis, represents the angle between the y-axis and the magnetic north pole
- values[1]: *Pitch*, angle of rotation about the x-axis
- values[2]: *Roll*, angle of rotation about the y-axis

Only *azimuth* and *roll* are used in this case, Figure 4.13 illustrates the coordinate system of the device.

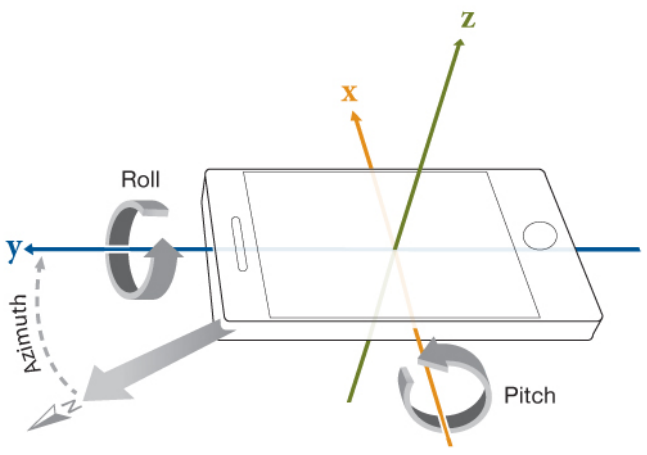


Figure 4.13: Azimuth, pitch and roll of device

The class *SensorManager* registers the motion sensor, the data delay can be set.[17] The data delay specifies the sample period of the sensor, the default-setting is 200 ms, to smoothly control the servo movement the SENSOR_GAME_DELAY with a sample period of 20 ms is chosen. Initially the sensor values were fluctuating so an averaging filter is applied. The last 10 values are kept in an array and the average of the values in the array is calculated. By pressing the volume down button the sensors get re-calibrated.

UDP client

The *azimuth* and *roll* are used to control the servo movement. These values are transmitted using the custom UDP protocol discussed in Section **??**. In the constructor of *PhoneOrientationListener*, the *UDPClient* is alled. It needs three arguments: IP-address and port number of server and the amount of messages per second (= 50). After initialization the UDP client simply waits for the *sendMessages()* method to be called. On the left side of Figure 4.12 the coordinates are displayed in a TextView which contains an OnClick-listener, when clicked the *sender*-variable in *PhoneOrientationListener* is toggled *true* or *false*. The message rate per second is set to 50 so every 20 ms the azimuth and roll are sent to the server.

Chapter 5

Conclusion

The first part of this chapter will summarize the implementation, achieved goals and give remarks where needed. The second part elaborates about future improvements or other approaches.

5.1 Summary

The main goal of the project is accomplished. Successfully capture and encode the video data from the USB camera. Send it over RTP to the NANOPI-server and send the live feed, again over RTP, to the smartphone-client. The smartphone receives, decodes and displays the video on screen. Additionally the device orientation is acquired and transmitted via UDP to the NANOPI, the ORANGEPI retrieves the coordinates from the server and controls the servo's position with the use of the PCA9685 servo/PWM. The big parts in the project are solved with the use of libraries, such as LIVE555, *efflux*, *Android Streaming Client*, etc. The challenge was to connect all the separate programs and let them work together as one entity.

5.2 Future improvements

This section proposes several improvements that can be made in the future. Due to the time limitation the system does not satisfy all the requirements. Following changes can be made:

- Multiple concurrent camera streams instead of one. The NANOPI is able to set up multiple streams by calling the streamer-thread several times. This is not the case for the receiver-thread because it still uses a global variable to pass the name of the FIFO to the *FileSink*. If this variable is eliminated multiple camera streams are possible.

- The video stream is shown in 2D on the smartphone, a future requirement is to display it in 3D using the VR-goggles.
- The current framework streams to the NANOPI and the NANOPI to the smartphone-client. So the same video data is streamed two times inside the network. An interesting modification could be that the ORANGEPI'S stream directly to the smartphone without the NANOPI as buffer to save bandwidth.
- When multiple streams are implemented the user can choose the channel by different gestures. The user wears a glove with a gesture recognition unit attached to it. Additional functionalities like zooming, adjust contrast/brightness,... can be implemented too.

Appendix A

hstreamrec.cpp code snippet

```
...

H264VideoStreamFramer* videoSource;
RTPSink* videoSink;

struct serv_struct {
   unsigned short port;
   char const *stream;
};

struct serv_struct args_s1, args_s2;

//h264-receiver initialize//
void afterPlayingRec(void* clientData); // forward
void init_new_receiver(unsigned short port_num, char const* fifoName); //
   forward

struct rec_struct{
   unsigned short port;
   char const* stream;
};

struct rec_struct args_r1, args_r2;

// A structure to hold the state of the current session.
// It is used in the "afterPlaying()" function to clean up the session.
struct sessionState_t {
   RTPSource* source;
   MediaSink* sink;
```

```
} sessionState;


void config_servers(){
    args_s1.port = 5008;
    args_s1.stream = "stream1";
   mknod("stream1", S_IFIFO | 0666, 0);

    args_s2.port = 5010;
    args_s2.stream = "stream2";
   mknod("stream2", S_IFIFO | 0666, 0);
}

void* thread_new_server(void *arguments) {
    struct serv_struct *args = (struct serv_struct *)arguments;
    init_new_server(args -> port, args -> stream);
    pthread_exit(NULL);
}

void* thread_new_receiver(void *arguments){
    struct rec_struct *args = (struct rec_struct *)arguments;
    init_new_receiver(args -> port, args -> stream);
    pthread_exit(NULL);
}

int main(int argc, char const *argv[])
{
    config_servers();

    pthread_t th2;
    int id_th2;
    id_th2 = pthread_create(&th2, NULL, thread_new_receiver, (void*) &args_r1
        );

    sleep(2);
    //server thread//
    pthread_t th1;
    int id_th1;
    id_th1 = pthread_create(&th1, NULL, thread_new_server, (void*) &args_s1);

    pthread_join(th1,NULL);
    pthread_join(th2,NULL);

    return 0;
}

void init_new_receiver(unsigned short port_num, char const* fifoName){
    TaskScheduler* scheduler = BasicTaskScheduler::createNew();
    UsageEnvironment* env = BasicUsageEnvironment::createNew(*scheduler);

    // Create the data sink for 'stdout':
    sessionState.sink = H264VideoFileSink::createNew(*env, fifoName, "test",
        NULL, 50000, False);
    // Note: The string "stdout" is handled as a special case.
    // A real file name could have been used instead.

    // Create 'groupsocks' for RTP
    char const* sessionAddressStr = "0.0.0.0";
    // Note: If the session is unicast rather than multicast,
    // then replace this string with "0.0.0.0"

    const unsigned char ttl = 7;
    const unsigned short rtpPortNum = port_num;

    struct in_addr sessionAddress;
    sessionAddress.s_addr = our_inet_addr(sessionAddressStr);

    const Port rtpPort(rtpPortNum);

    Groupsock rtpGroupsock(*env, sessionAddress, rtpPort, ttl);

    sessionState.source = H264VideoRTPSource::createNew(*env, &rtpGroupsock
        ,96,50000);

    // Finally, start receiving
    *env << "Start receiving...\n";
    sessionState.sink->startPlaying(*sessionState.source, afterPlayingRec,
        NULL);

    env->taskScheduler().doEventLoop();
}

void init_new_server(unsigned short port_num, char const *vid_file){
    TaskScheduler* scheduler = BasicTaskScheduler::createNew();
    UsageEnvironment *env = BasicUsageEnvironment::createNew(*scheduler);

    char const* destinationAddressStr = "192.168.199.139";
    const unsigned char ttl = 255;

    struct in_addr destinationAddress;
    destinationAddress.s_addr = our_inet_addr(destinationAddressStr);

    const Port rtpPort(port_num);

    *env << "rtp://" << destinationAddressStr << ":" << port_num << "\n";

    Groupsock rtpGroupsock(*env, destinationAddress, rtpPort, ttl);

    OutPacketBuffer::maxSize = 50000;
    videoSink = H264VideoRTPSink::createNew(*env, &rtpGroupsock, 96);

    *env << "Beginning streaming...\n";
    play(vid_file, env);

    env->taskScheduler().doEventLoop();
}

...

void play(char const *vid_file, UsageEnvironment *env) {
    DeviceSource* devS = DeviceSource::createNew(*env,vid_file);
    if (devS == NULL)
    {

    *env << "Unable to read from\"" << "Buffer"
    << "\" as a byte-stream source\n";
    exit(1);
    }

    FramedSource* videoES = devS;

    // Create a framer for the Video Elementary Stream:
    videoSource = H264VideoStreamFramer::createNew(*env, videoES, False);

    // Finally, start playing:
    *env << "Beginning to read from file...\n";
    videoSink->startPlaying(*videoSource, afterPlaying, videoSink);
}
```

Appendix B

streamer: DeviceSource

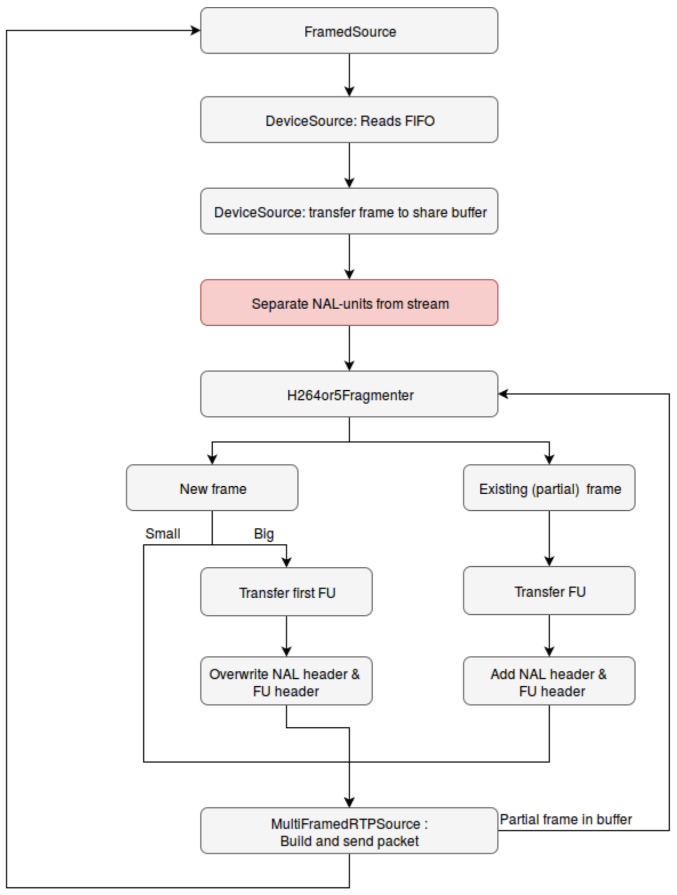

FramedSource
DeviceSource: Reads FIFO
DeviceSource: transfer frame to share buffer
Separate NAL-units from stream
H264or5Fragmenter
New frame
Existing (partial) frame
Small
Big
Transfer first FU
Transfer FU
Overwrite NAL header & FU header
Add NAL header & FU header
MultiFramedRTPSource : Build and send packet
Partial frame in buffer

Figure B.1: How DeviceSource reads from FIFO and streams the content

Bibliography

[1] Lindsay. *Virtual reality vs. Augmented reality.* [online]. Oct. 2015. URL: `http://www.augment.com/blog/virtual-reality-vs-augmented-reality/`.

[2] Don Reisinger. *Why virtual reality is about to go mainstream.* [online]. Oct. 2015. URL: `http://fortune.com/2015/10/07/virtual-reality-mainstream/`.

[3] Orange Pi. *Orange Pi Lite.* [online]. URL: `http://orangepi.com/orange-pi-lite`.

[4] FriendlyArm. *Nano Pi M2.* [online]. URL: `http://wiki.friendlyarm.com/wiki/index.php/NanoPi_M2`.

[5] Corbet. *The Video4Linux2 API: an introduction.* [online]. Oct. 2006. URL: `https://lwn.net/Articles/203924/`.

[6] Iain E. Richardson. *The H.264 Advanced Video Compression Standard.* John Wiley & Sons, 2011.

[7] MulticoreWare Inc. *HEVC / H.265 Explained.* [online]. URL: `http://x265.org/hevc-h265/`.

[8] LSI logic. *H.264/MPEG-4 AVC Video Compression Tutorial.* [online]. Wite Paper. URL: `http://web.cs.ucla.edu/classes/fall03/cs218/paper/H.264_MPEG4_Tutorial.pdf`.

[9] google android. *Position Sensors.* [online]. URL: `https://developer.android.com/guide/topics/sensors/sensors_position.html`.

[10] Sensoray. *Color Spaces in Frame Grabbers: RGB vs. YUV.* [online]. URL: `http://www.sensoray.com/support/appnotes/frame_grabber_capture_modes.htm`.

[11] Ben Balser. *Compressor: H.264 Profiles and Entropy Modes.* [online]. Sept. 2014. URL: `https://www.macprovideo.com/hub/final-cut/compressor-h264-profiles-entropy-modes`.

[12] PixelTools. *Rate Control and H.264.* [online]. URL: `http://www.pixeltools.com/rate_control_paper.html`.

[13] Fabio Sonnati. *FFmpeg the swiss army knife of Internet Streaming part III*. [online]. July 2012. URL: `https://sonnati.wordpress.com/2011/08/19/ffmpeg-%e2%80%93-the-swiss-army-knife-of-internet-streaming-%e2%80%93-part-iii`.

[14] Inc. Live Networks. *FAQ Live555*. [online]. URL: `http://www.live555.com/liveMedia/faq.html#threads`.

[15] Julian Cerruti Ayelen Chavez. *Android Streaming Client*. [online]. URL: `https://github.com/ekumenlabs/AndroidStreamingClient`.

[16] google android. *Motion Sensors*. [online]. URL: `https://developer.android.com/guide/topics/sensors/sensors_motion.html`.

[17] google android. *SensorManager*. [online]. URL: `https://developer.android.com/reference/android/hardware/SensorManager.html`.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Remote video monitoring system based on VR

Richting: **master in de industriële wetenschappen: elektronica-ICT**
Jaar: **2017**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Poesen, Sebastiaan **Ferrari, Marijn**

Datum: **26/06/2017**