

TECHNISCHE UNIVERSITEIT EINDHOVEN  
Department of Mathematics and Computer Science

MASTER'S THESIS

Volund

A research vehicle for networked video streaming

by

R. Meijer

Supervisor: dr. J. J. Lukkien

Eindhoven, March 2004

---

Copyright © 2003 Ralph Meijer

# Contents

<b>Preface</b>	<b>ix</b>
<b>Introduction</b>	<b>xi</b>
<b>1 Project Environment</b>	<b>1</b>
1.1 QOSIH . . . . .	1
1.1.1 TCP-MM . . . . .	1
1.2 KISS . . . . .	2
1.3 Introduction into Scalable Video . . . . .	2
1.4 Project experimentation setup . . . . .	3
<b>2 Requirements</b>	<b>5</b>
2.1 Scenarios . . . . .	5
2.1.1 Scenario 1 . . . . .	5
2.1.2 Scenario 2 . . . . .	6
2.1.3 Scenario 3 . . . . .	6
2.2 Software Requirements . . . . .	6
2.2.1 Scenario 1 . . . . .	6
2.2.2 Scenario 2 . . . . .	7
2.2.3 Scenario 3 . . . . .	7
<b>3 Streaming MPEG using RTP</b>	<b>9</b>
3.1 Introduction into the MPEG encoding . . . . .	9
3.1.1 Video Bitstream Structure . . . . .	9
3.1.2 Video Encoding . . . . .	10
3.2 Introduction into RTP . . . . .	12
3.3 Introduction into RFC 2250 . . . . .	12
3.3.1 Packetisation . . . . .	13
3.3.2 Recombination . . . . .	14
3.3.3 Loss handling . . . . .	14
3.4 Streaming Scalable Video . . . . .	14
3.4.1 Synchronisation . . . . .	14
3.4.2 Recombination and data loss . . . . .	15

---

<b>4</b>	<b>Architecture</b>	<b>17</b>
4.1	Introduction . . . . .	17
4.2	System Architecture . . . . .	17
4.3	Architectural Pattern . . . . .	17
4.4	Pipes and Filters . . . . .	18
4.5	Pipelines & Components . . . . .	19
4.5.1	Characterisation of a Component . . . . .	19
4.5.2	One layer . . . . .	20
4.5.2.1	File source . . . . .	21
4.5.2.2	Packetiser . . . . .	22
4.5.2.3	RTP Wrapper . . . . .	22
4.5.2.4	UDP Sink . . . . .	23
4.5.2.5	UDP Source . . . . .	23
4.5.2.6	RTP Unwrapper . . . . .	23
4.5.2.7	MPEG Decoder . . . . .	24
4.5.2.8	Display . . . . .	24
4.5.3	One layer with Feedback . . . . .	25
4.5.3.1	Packet Filter . . . . .	25
4.5.3.2	Filter Control . . . . .	26
4.5.3.3	Stream Monitor . . . . .	27
4.5.3.4	Remarks. . . . .	27
4.5.4	Multiple layers . . . . .	27
4.5.4.1	Adder . . . . .	28
4.5.4.2	RTP Unwrapper . . . . .	30
4.5.4.3	Remarks . . . . .	31
4.5.5	Using Modified TCP . . . . .	32
<b>5</b>	<b>Design &amp; Implementation</b>	<b>35</b>
5.1	Media Processing Framework . . . . .	35
5.2	Introduction into GStreamer . . . . .	36
5.3	The GStreamer Model . . . . .	37
5.3.1	Element states . . . . .	39
5.3.2	Execution model . . . . .	39
5.3.2.1	The Basic Scheduler . . . . .	39
5.3.2.2	The Optimal Scheduler . . . . .	40
5.4	Existing GStreamer Elements . . . . .	41
5.5	Changed Elements . . . . .	41
5.6	New Elements . . . . .	42
5.6.1	rtpmpegec . . . . .	42
5.6.2	rtpmpegpars . . . . .	43
5.6.3	add . . . . .	43
5.7	Building the pipeline . . . . .	44

---

<b>6 Findings</b>	<b>47</b>
6.1 Handling MPEG . . . . .	47
6.2 Using GStreamer . . . . .	47
6.3 Using RFC 2250 . . . . .	48
6.4 Reusability . . . . .	48
<b>7 Conclusions</b>	<b>49</b>
<b>8 Future work</b>	<b>51</b>
<b>Bibliography</b>	<b>53</b>



# List of Figures

<b>1 Project Environment</b>	
1.1 SNR scalable video . . . . .	3
1.2 Experimentation Setup . . . . .	4
<b>2 Requirements</b>	
2.1 Scenario 1 situation sketch . . . . .	5
<b>3 Streaming MPEG using RTP</b>	
3.1 MPEG terms . . . . .	10
3.2 MPEG frame dependencies . . . . .	11
3.3 Packetisation . . . . .	13
3.4 Input/output behavior of a recombinator . . . . .	15
<b>4 Architecture</b>	
4.1 Graphical representation of components . . . . .	20
4.2 Data Flow Diagram for one video stream . . . . .	21
4.3 Data Flow Diagram for one video stream . . . . .	25
4.4 CSP Pseudo Code for Packet Filter . . . . .	26
4.5 Data Flow Diagram for two video streams without feedback, receiver side . . . . .	28
4.6 Flow chart of the Adder component . . . . .	30
4.7 Data Flow Diagram for three video stream, receiver side from just before MPEG decoding . . . . .	32
4.8 Streaming one video stream using TCP. . . . .	33
<b>5 Design &amp; Implementation</b>	
5.1 GStreamer pipeline for sending two layers of scalable video . . . . .	45
5.2 GStreamer pipeline for receiving and displaying two layers of scalable video . . . . .	45



# Preface

This thesis is the result of my project work for the Master's degree in Computer science at the Eindhoven University of Technology (TU/e). The project was carried out as part of the QOSIH work in the Software Architectures Group within Philips Research, Eindhoven.

I wish to thank all people who's help in creating this thesis is greatly appreciated. I thank Peter van der Stok, Wim van der Linden and Michael van Hartkamp from Philips Research for supervising my work while there, Johan Lukkien for his supervision from TU/e, Peter van den Hamer for all his tough questions and support, the members of the KISS project and the many people in the Software Architectures group for their suggestions and stimulating atmosphere.

Special thanks go out to Michael and Johan for reviewing this thesis, and finally to my parents, parents-in-law and especially Irma Buelens for encouraging and supporting me in every way they possibly could.



# Introduction

Volund is the name of the God of smiths and supreme craftsmanship in Norse mythology. It was chosen as the name for the system that was build as part of the Master's project carried out as part of the QOSIH work in the Software Architectures group within Philips Research. The system is to be used as a research vehicle with which prototypes for networked video streaming can be *forged*.

QOSIH stands for Quality of Service in In-Home Digital Networks. The aim of the project by that name is (from the project description):

To increase the number of applications that can run with satisfactory results for the user on an in-home heterogeneous wireless and wired digital network. A consistent management of the network attributes and a consistent decion making over all network nodes are necessary. The project concentrates on bandwidth management decisions.

Quality of Service can be achieved in several ways. You can use a reservation and priority mechanism at the network layer for example. You can also do things at a higher level, which we aim to do. Applying QoS schemes frequently is a binary process; either you get data or you don't at some point.

Volund is meant to research QoS for data over wireless connections. We would like to have hands-on knowledge on what involves doing QoS for such an environment. The data used to be transmitted by Volund will be Scalable Video. This makes it possible to remove the binary nature of most QoS schemes by layering some original video stream and transmitting these layers separately over the network, possibly using some of the network level QoS services available.

**Chapter 1, "Project Environment"** of this document describes the environment in which this Master's project was carried out. It gives a quick overview of QOSIH and related projects, gives an introduction to scalable video, and describes the experimentation setup. **Chapter 2, "Requirements"** states the scenarios used as a guideline to build Volund along with the software requirements that follow from these scenarios. Then, **Chapter 3, "Streaming MPEG using RTP"** gives a detailed description of the formats and protocols used for encoding and transporting scalable video.

Using the software requirements, **Chapter 4, "Architecture"** presents the chosen architecture, followed by the design and implementation described in **Chapter 5, "Design & Implementation"**. Finally, chapters **Chapter 6, "Findings"**, **Chapter 7, "Conclusions"** and **Chapter 8, "Future work"** sum up the findings, conclusions and possible future work.



# Chapter 1

## Project Environment

### 1.1 QOSIH

Volund is meant as research vehicle for the QOSIH project. QOSIH stands for Quality of Service in In Home Digital Networks and is carried out in the Software Architectures (SwA) group in the Information and Software Technology (IST) sector of Philips Research.

QOSIH's focus is to examine problems in, and find solutions to the transmission of streaming data, in particular video, in digital networks in home situations. Video streams are by current standards high in bandwidth use, and the emergence of wireless networks makes that even more evident.

MPEG video with a resolution of 720x480 uses 2 - 8 Mbit/s. Existing wireless network technologies in the IEEE wireless standards family 802.11, commonly known as WiFi have relatively limited bandwidth. For example, IEEE 802.11b has a practical maximum throughput of 5.5 Mbit/s in ideal situations. This bandwidth is easily saturated with streaming video. Certainly when there are multiple users of the same network, Quality of Service for such video streams becomes an issue. QOSIH tries to find solutions that lead to a better user experience for video streaming in this kind of environment.

#### 1.1.1 TCP-MM

Streaming of video can be done using several technologies. The focus for Volund is to use RTP over UDP, but there are also TCP based solutions for streaming video. Standard TCP depends on retransmissions to guarantee that all data that has been sent is also received, and can be delivered in the correct order. Data that has been received is acknowledged, and if something is missing, it will be resent. This detection and retransmission of data, however, takes time and can cause latency problems.

One solution that is examined inside QOSIH, is to alter the handling of TCP connections (not the actual wire protocol), such that retransmissions only take place when there is a high chance that the data can be processed in time on the receiving end. This modified TCP is called TCP-MM, where MM stands for Multi Media.

One of the scenarios of Volund will accommodate this solution.

## 1.2 KISS

QOSIH also contributes to the European OZONE project. Another contributor to this project is KISS, carried out in another group inside IST.

Where QOSIH is concentrated on the Network part of Quality of Service, KISS is also concerned with the Terminal QoS. A device handling video streams has certain constraints as well in terms of for example processing power and memory.

For this Terminal QoS, research is being done on how to best split up video into multiple layers. Tests are being performed on the effect of difference in quality between the layers, and how much data is stored in each layer. Each of these layers are transported using MPEG streams, and streams produced during these tests can be used in Volund, so we have real data to work with.

KISS integrates output from other projects in one common demonstrator for OZONE, and is using a part of Volund for the party responsible for sending a video stream. The receiving side is being implemented by themselves, because of platform issues. But since most of the work is similar, we have been able to give each other tips and get insight in the work of the other team.

## 1.3 Introduction into Scalable Video

Scalable video consists of  $n$  layers (we will use  $n=3$ ); one baselayer and  $n-1$  enhancement layers with increasing quality and non-decreasing bit rate (the number of bits used to in the encoding of a sequence of pictures per time unit).

Scalable Video is not limited to a particular video encoding, but in the rest of this document we will assume MPEG 2 as the video encoding in the layers. For the input, the so-called original video, we assume a sequence of pictures in a raw data format that can be directly used as input to the encoders. This data format is also expected to be used for the output of the decoders and it should be easy to make computations on the pixel level with this format. An example of such a format is YUV 4:2:0.

There are several ways to layer a video stream. Two of them are Signal Noise Ratio (SNR) Scalability and Spatial Scalability. In SNR scalability, the quality of the different layers is varied by changing the used bit rate for encoding the picture sequence. For example when using three layer, the bit rates for encoding the different layers can be choosen as 1Mbit/s for the base layer, 1Mbit/s for the first enhancement layer, and 2Mbit/s for the second enhancement layer.

Spatial scalability on the other hand varies the picture size (resolution) of the pictures being encoded. If our original video stream uses a picture resolution of 720x576 (called standard definition, or SD), the base layer could be choosen to be encoded using a scaled down version by a factor of 4 (180x144), and the first enhancement layer at half of the original size (360x288). The second enhancement layer would use the original size.

We have used SNR scalability in this project, and the following is description of how the layers in SNR scalability are created from a picture sequence, and subsequently merged back to one sequence sequence.

The baselayer is a low bit rate encoding of the original video stream. To make the first enhancement layer, this stream is decoded again and subtracted (pixel-wise) from the original, unencoded, stream. The result is then encoded with a higher bit rate, to be used as the second layer. The second enhancement layer is subsequently decoded and subtracted from the original stream and encoded with a still higher bit rate. This is the third layer. The operations that have to be performed on a stream are represented in [Figure 1.1](#).

To produce a picture sequence from the different layers, at least the base layer is needed. Then for every enhancement, the pictures from all lower quality layers are needed.

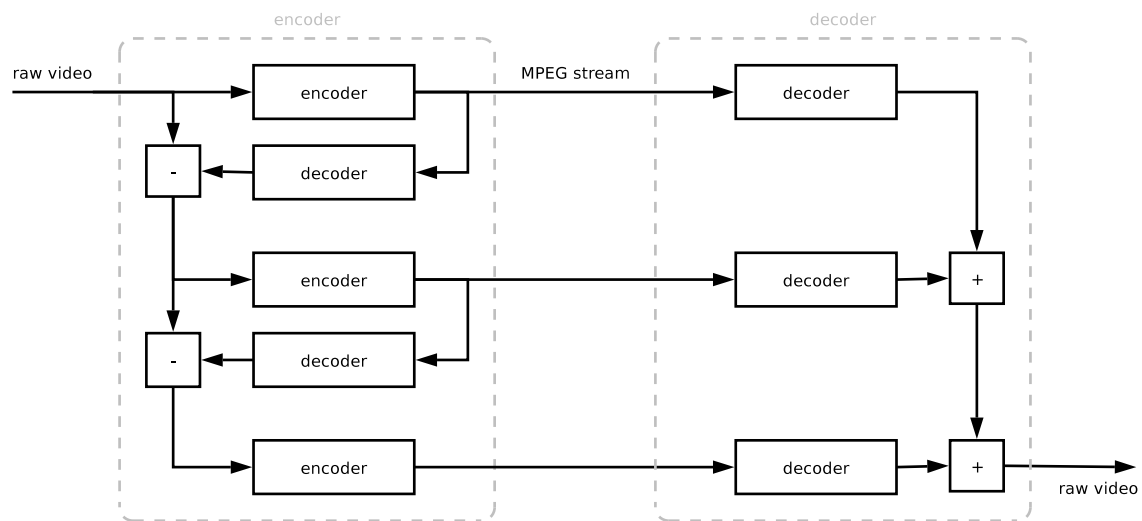


Figure 1.1: SNR scalable video

## 1.4 Project experimentation setup

QOSIH has a number of computers for evaluating the possible solutions they come up with. The setup consists of: one machine governing common services like file serving, internet access and name resolution, four desktop machines with both wired as well as wireless interfaces, and a local wireless access point (as shown in [Figure 1.2](#)).

The wired infrastructure and common services provide a development platform, whereas the wireless facilities are used for the actual experiments.

The setup was not in all aspects ready to be used. One particular problem was getting the wireless PCMCIA cards working. During the project, considerable time was spent installing and configuring all desktops with Debian (GNU/Linux).

During the project, it became clear that none of the three original desktop machines had enough processing power to simultaneously decode multiple MPEG streams, and so an extra 2GHz machine was ordered and installed.

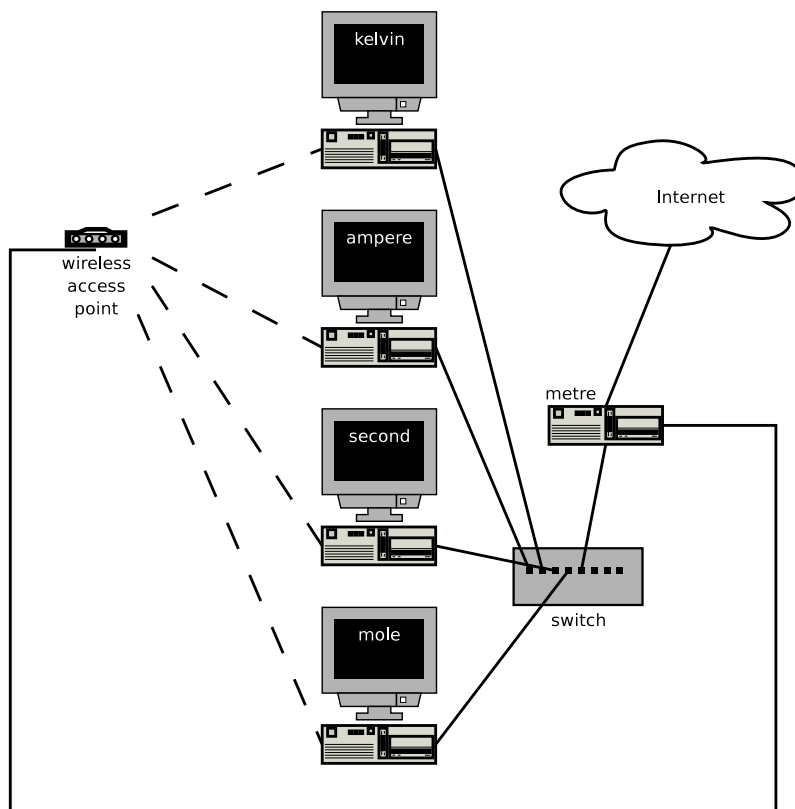


Figure 1.2: Experimentation Setup

## Chapter 2

# Requirements

To capture the typical use of Volund, three technical scenarios have been created. From these scenarios, requirements to our system have been derived. The choice of technical scenarios over end-user oriented scenarios is because the users of Volund are not end-users. Volund is a research vehicle and the “client” has mostly technical requirements for using it.

## 2.1 Scenarios

### 2.1.1 Scenario 1

The goal in this scenario is to stream a video over a wireless link from one terminal to another. The system has a feedback loop that informs the sending side of the quality of the data that arrives on the receiving end by requesting a number of layers. The actual strategy that determines how many layers are actually requested is not part of the scenario.

The wireless protocol used is IEEE 802.11b and the actual video layers are sent over this wireless link using RTP, the Real-time Transport protocol. Each layer uses one so-called *RTP session*, and we call the set of RTP sessions belonging to the same video a *bundle*. RTP will be explained further in [Section 3.2](#).

The video data is to be encoded as MPEG 2 Elementary Streams, described in [Section 3.1](#).

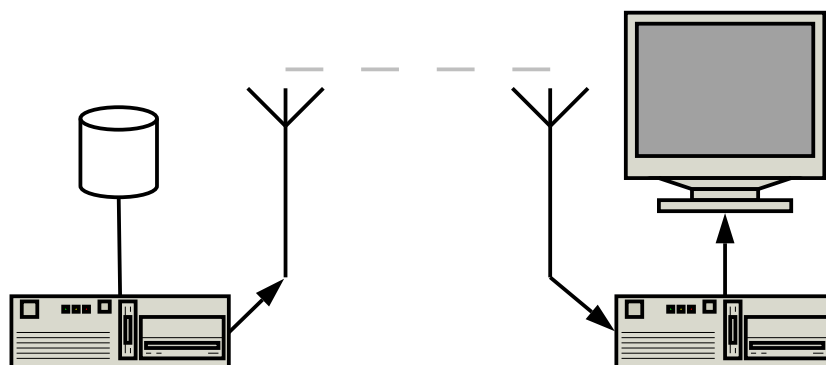


Figure 2.1: Scenario 1 situation sketch

[Figure 2.1](#) depicts the following scenario:

- A bundle is streamed to another party via IEEE 802.11b. It is assumed, there is no other traffic in the wireless network. Initially, the video is sent using three layers and the format of the payload is MPEG 2 Elementary Stream.
- A source of interference is introduced, causing the available bandwidth for the bundle to decrease. This interference can be devices like a microwave that operate in the same frequency range, but also other IEEE 802.11b traffic that is not managed by this application.
- The receiving entity discovers that it is experiencing packet loss, and decides that the other party should send a smaller amount of streams in the bundle. It informs the sending party of this decision via an unspecified protocol (out of band).
- The sending party receives the message and conforms to the request.
- After a while, the receiving party notices that no packets are lost anymore, and tells the sending party to start using more layers again.
- The sending party receives the message and conforms to the request.

### 2.1.2 Scenario 2

Someone wants to do research on Quality of Service of streaming scalable video over wireless connections. Because he is also concerned with terminal constraints in terms of processing power, he wants to do the decoding of the MPEG fragments in the different layers sequentially, depending on whether there is still time to decode the data in such a layer before the resulting (combined) picture must be displayed.

### 2.1.3 Scenario 3

Someone wants to do research on the use of a modified TCP protocol for streaming scalable video streams (see [Section 1.1.1](#)). The modifications on TCP are done in the Operating System, and the researcher wants to use Volund for streaming scalable video using this modified TCP implementation.

## 2.2 Software Requirements

These are the requirements following from the scenarios in [Section 2.1](#). In this document, all MPEG video streams are MPEG 2 Elementary Streams.

### 2.2.1 Scenario 1

1. It **MUST** be possible to transmit a video sequence using Scalable Video.  
It **MUST** be possible to split up a video in a number of layers. There is one base layer and several enhancement layers. The base layer is a lower quality version of the original. Each enhancement layer contains additional information, to increase the quality of the video when recombined with the baselayer and so on.
2. It **MUST** be possible to encode a video layer as an MPEG 2 Elementary Stream.
3. It **MUST** be possible to encapsulate MPEG video streams using RTP.
4. The application **MUST** be able to send RTP streams over UDP/IP.

5. The application **MUST** be able to use IEEE 802.11b for transporting RTP streams.
6. It **MUST** be possible to transmit the groups of RTP packets that make up one picture at a rate that, on average, matches the frame rate of the original video.
7. It **MUST** be possible to start the sending of those layers that are allowed to be transmitted simultaneously.
8. It **SHOULD** be possible to control the maximum bandwidth to be used for sending each layer.
9. It **MUST** be possible to recombine the payloads of the RTP streams in one bundle on the receiving end.
  - (a) It **MUST** be possible to decode each MPEG stream to a sequence of bitmaps .
  - (b) It **MUST** be possible to match up the pictures in the different layers carrying the same timestamp and merge them into one picture with the heighest quality possible.
  - (c) This process **MUST** be able to deal with packet loss.
10. It **MUST** be possible to dynamically control how many layers will be transmitted.
11. The control of the number of layers to be send **SHOULD** be remotely (over a network) accessible.
12. There **MUST** be a way to monitor the reception quality in terms of packet loss and lateness.
13. It **MUST** be possible to decide the number of layers to be sent, based on the reception quality.
14. It **SHOULD** be possible to simulate interference by other traffic or devices operating in the same frequency range as the wireless transport.

### **2.2.2 Scenario 2**

1. It **SHOULD** be possible to send and receive scalable video, like in scenario 1.
2. It **SHOULD** be possible to control which decode step is executed when.
3. It **SHOULD** be possible to skip the decoding of a part of the stream (because of lack of time).
4. It **SHOULD** be possible to combine the gathered statistics from scenario 1, with information about terminal QOS to control the number of layers being sent.

### **2.2.3 Scenario 3**

1. It **SHOULD** be possible to use TCP in stead of UDP to stream scalable video between two parties.
2. It **SHOULD** be possible to have the data pulled from the TCP socket no faster than the frame rate for retransmissions to work.



## Chapter 3

# Streaming MPEG using RTP

### 3.1 Introduction into the MPEG encoding

MPEG 1 audio and video compression was developed for storing and distributing digital audio and video [VD3]. MPEG 2 extends MPEG 1 by, for example, allowing a wider range of bit rates, better audio facilities and more advanced motion compensation. The basic format is still the same, however.

In MPEG 1 there exist four types of bit streams. Each of the first three, the Video Bitstream, the Audio Bitstream and the Data Bitstream is also referred to as Elementary Stream. The Data Bitstream can hold arbitrary data, defined by the user, such as subtitles. The fourth type of bit stream is called System Bitstream. In a System Bitstream, multiple Elementary Streams can be multiplexed into one bitstream, for example to combine the video and audio belonging to one and the same program in a single bit stream.

MPEG 2 extends MPEG 1 to cover a wider range of applications. For example, it supports up to five MPEG 1 compatible audio channels, but also several non backwards compatible audio extensions like Dolby Digital 5.1. On the video side it has six different profiles for supporting higher bitrates for higher resolution like in High Definition TV (HDTV).

Where MPEG 1 has the System Bitstream, MPEG 2 defines two methods for multiplexing Elementary Streams. The first method is Transport Stream and is used in applications where data loss is likely to occur. Small data packets are used to handle data loss robustly. One Transport Stream can contain multiple programs.

The second method is called Program Stream. It is aimed at applications where data loss is not likely to occur, such as storage on DVDs. It uses longer packets than Transport streams, and only Elementary Streams belonging to one program can be multiplexed in a Program Stream.

In Volund we only consider Video Bitstreams, also referred to by their general name Elementary Streams.

#### 3.1.1 Video Bitstream Structure

Video bitstreams have a hierarchical structure consisting of seven layers:

- Video Sequence
- Sequence
- Group of Pictures (GOP)

- Picture
- Slice
- Macroblock
- Block

In the bitstream Sequences, GOPs, Pictures and Slices all have a header followed by the contents of the lower layer, including their headers. A Video Sequence consists of one or more Sequences, terminated by an end of sequence marker. All other layers end at the beginning of a new header of the same layer or a higher layer.

Next, a quick run through of the most important data in the different headers.

The Sequence Header contains the size (width, height) and picture rate of the video. The GOP header then contains a time code, and a field that flags whether there are motion vectors referring to the previous GOP. The Picture Header then contains a temporal reference and the type of picture. The make-up of the Slices and lower layers is not described in this document. It is enough to know that Pictures are made up of one or more Slices.

Figure 3.1 shows the terms mentioned above related to each other.

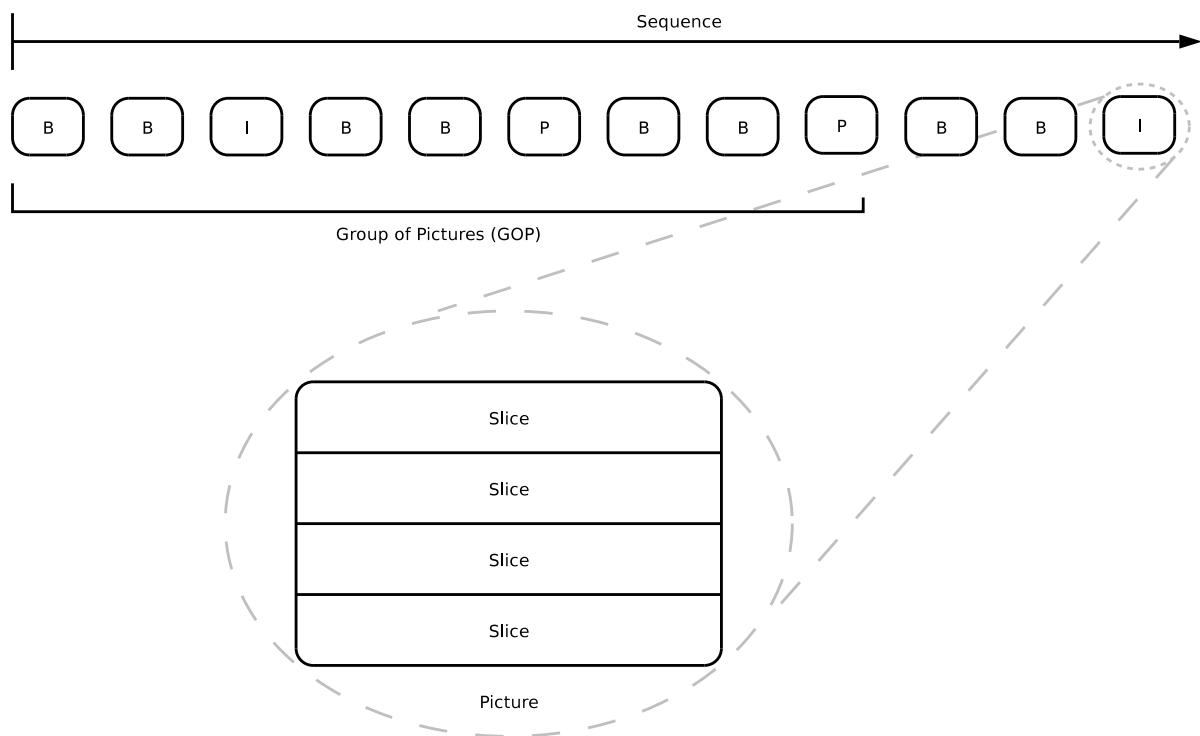


Figure 3.1: MPEG terms

### 3.1.2 Video Encoding

MPEG 1 defines four pictures types to encode a video sequence. Of those, the three described below are also used in MPEG 2. The fourth one (D) is not used in streaming applications.



## 3.2 Introduction into RTP

RTP is the Real-time Transport Protocol [RFC1889]. It offers end-to-end network transport functions for transmitting real-time data, like video, over existing network services.

RTP caters for payload type identification, sequence numbering and timestamping. RTP relies on the underlying network to move RTP packets between two end points. Although it can be used with other network protocols, the most common used protocol for the underlying network is UDP (User Datagram Protocol) on top of IP (the Internet Protocol).

UDP/IP provides the concept of a *transport address*, the combination of a network address and a port that identifies a transport-level endpoint. For this, respectively an IP address and UDP port are used. We will only look into unicast transmissions. An RTP session is the association of a set of participants communicating using RTP, defined by the combination of their transport addresses.

An RTP packet, a data packet consisting of RTP header information and the payload data, is typically sent as one packet in the underlying protocol, but multiple RTP packets maybe be contained if the encapsulation method for the underlying protocol allows for this. In this case an RTP packet is contained in one UDP packet. The headers in an RTP packet contain, among others, the following information:

**Payload type** Identifies the format of the RTP payload. A number of these are defined in so-called profiles [RFC1890] and we use the one for MPEG1/2 video.

**Sequence number** This number increases by one for each RTP packet being sent and is used for determining the order of the packets and detecting packet loss.

**Timestamp** Contains the time to which the first octet in the payload logically belongs in the sampled data. The clock frequency depends on the profile used.

**Synchronisation Source (SSRC)** A number that uniquely identifies the source of a stream of RTP packets withing the current session and determines the context in which the sequence numbers and timing information is sent.

## 3.3 Introduction into RFC 2250

First, a protocol as defined by the Internet Engineering Task Force (IETF), RFC 2250, will be explained. Then, how this RFC can be used in Volund.

RFC 2250 [RFC2250] describes an RTP payload format for transmitting MPEG 1 or 2 video. It covers MPEG video and audio Elementary Streams, and also for MPEG 1 System and MPEG 2 Transport and Program Streams. We will only look into transporting video Elementary Streams.

MPEG Elementary Streams were designed with low loss in mind. Also, there is large amount of dependency between data in the bitstream. Data loss in one part of the bitstream can have large effects, because correctly transmitted data may not be usable because it depends on data that has been lost. The RFC offers an encapsulation of the MPEG bitstream to make certain recovery strategies possible.

### 3.3.1 Packetisation

The RFC offers a packetisation scheme that describes how to split up the bitstream into chunks that fit in an RTP packet. It adds a so-called MPEG Specific RTP Header to these chunks for easier handling on the receiving side. This is especially useful for detecting and recovering from packet loss. Every packet must be smaller than the Maximum Transfer Unit of the underlying physical network. For Ethernet like networks, this is usually 1500 bytes.

The fragmentation follows these rules:

- The MPEG Video Sequence Header, when present, will always be at the beginning of an RTP Payload.
- An MPEG GOP Header, when present, will be at the beginning of an RTP Payload, or be preceded by a Video Sequence Header.
- An MPEG Picture Header, when present, will be at the beginning of an RTP Payload, or be preceded by a GOP Header.

The beginning of a slice must either be the first data in a packet (after any headers) or must follow after some integral number of slices in a packet. Also, every header must be completely contained within the packet. This means that, to fit the largest possible header, an the RTP payload has a minimum size of 261 bytes.

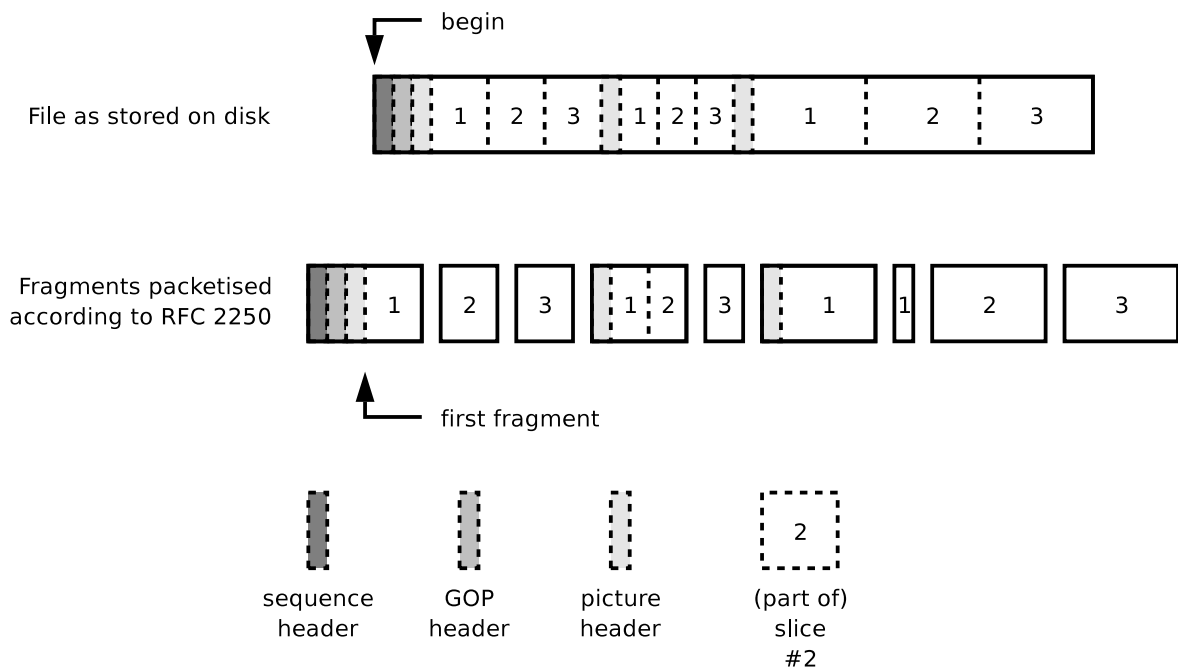


Figure 3.3: Packetisation

**Figure 3.3** shows an example MPEG file and the fragments resulting from the packetisation according to the rules above. These fragments are the payload of the RTP packets. The example file shows a sequence (starting with a sequence header) containing one GOP (starting with a GOP header) and three pictures. Each picture starts with a picture header and contains three slices. What we see is that each header is at the beginning of a fragment, and that each new slice starts

a new fragment. However, a fragment may contain multiple (complete) slices (slices #1 and #2 of the second picture), but if a slice does not fit in one fragment, it is spread out over multiple fragments (slice #1 from the third picture).

### 3.3.2 Recombination

To be able to view the stream again on the receiving side, the RTP packets need to be accumulated and processed. Assuming no data loss, concatenating the payload of the RTP packets (in order), would yield the original MPEG stream again. To order the RTP packets, the RTP sequence number can be used. Since these sequence numbers increase by one for each next fragment in the MPEG bitstream, plainly ordering the packets by the sequence number is sufficient.

### 3.3.3 Loss handling

RFC 2250 suggests a number of strategies for error recovery and resynchronisation. Using the information in the MPEG specific RTP headers, you can detect what was in the packets that were lost at the receiving end. By keeping track of the temporal references for each picture that is received, along with their picture type, you can detect whether a GOP or Picture Header was lost.

A picture in MPEG is made up of one or more *slices*, and a slice is intended to be the unit of recovery in case of data loss. Every MPEG decoder will skip to the beginning of the next slice when it encounters an error in the stream. RFC 2250 caters for this by setting flags on each RTP packet when the payload it contains starts with the first byte of a slice, and when the payload ends with the last byte of a slice.

So, when a gap is detected in the RTP sequence numbers by the receiving party, further processing of the payload can be suspended until a move to a known state, like the beginning of the next slice, can be made. The data received up to that point, which has not been processed, is discarded.

RFC 2250 further explains how lost header information can be reconstructed from earlier received headers and RTP timestamps.

## 3.4 Streaming Scalable Video

Volund is specifically meant for handling the streaming of Scalable (MPEG) Video using RTP. In this case, each layer of this Scalable Video is sent in its own RTP stream. Although the most part of RFC 2250 can be used for this, RFC 2250 is concerned with sending only one, independent, video stream.

In transporting Scalable Video, however, this assumption is not correct. Although the payload in each RTP stream that represents a layer in the Scalable Video is in itself a valid MPEG stream, there are higher level dependencies between the layers, that require special attention regarding synchronisation and error handling.

### 3.4.1 Synchronisation

The data in the original raw picture sequence, is split up in a number of layers at the picture level. That is, each picture in the original sequence corresponds to a picture in each of the layers of the Scalable Video MPEG Streams. Their only connection from then on, is the display timestamp of the picture. So, when synchronising the different layers on the receiving end, the timestamp of the pictures in the picture sequence after decoding the MPEG stream is used to match the pictures and combine them into one picture again, suitable for display or storage.

### 3.4.2 Recombination and data loss

The recombination of the pictures is the inverse operation of the process that created the respective pictures in the layers, but since data loss is to be expected, it can happen that not all of the pictures in their respective layers have been received correctly. This means only a part of the pictures can be recombined. For example, when using three layers, if, for the corresponding picture in the original video, the picture in the base layer and the picture in the second enhancement layer were received, but not the picture in the first enhancement layer, only the picture in the base layer can be used.

Also, since we are working at the picture level, the assumption of having the slice as unit of recovery is not valid. If, for example, a picture in the original video sequence is made up of four slices, the MPEG streams of the layers, will also have four slice per corresponding picture. Say all slices of the picture in the baselayer were correctly received, but in the first enhancement layer, only the first two, the image resulting from the merging of these two layers would have the first half of the picture in higher quality than the second half.

This is probably very noticeable to the end user, and therefore, it is probably better to only use the base layer in this case. Although it is of lower quality, this is probably less disturbing. The process is shown in [Figure 3.4](#).

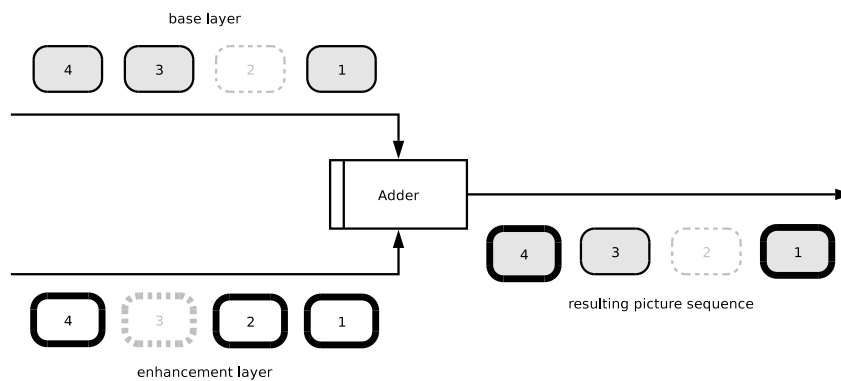


Figure 3.4: Input/output behavior of a recombinator

With RFC 2250, the MPEG specific RTP headers can be used to reconstruct picture and GOP headers when lost, so that at least some of the slices can be decoded. For example, if a picture header is lost, one could skip to the next slice, and use the MPEG specific headers from its RTP packet to reconstruct the missing information. However, since we use the picture as our unit of recovery for working with scalable video, we will collect all the data belonging to one picture (before decoding), and only process a picture when we have received all data for that picture, including the header, and refrain from any header reconstruction.



# Chapter 4

## Architecture

### 4.1 Introduction

The requirements derived from the scenarios show overlap. For example, in all scenarios, files have to be read, MPEG data has to be decoded, and pictures have to be displayed. Since the goal is to construct one system that can satisfy all scenarios, we would like to identify functional blocks that can be combined in a configuration to form an application that corresponds to one of the scenarios.

Also, we would like to reuse the functional blocks for future applications, not covered by the current scenarios, by writing new functional blocks for the new functionality, leaving the existing ones intact. The new application would be a configuration of the extended set of functional blocks.

We distinguish between an overall system architecture and an architecture for the software that encompasses most of the functionality.

In this chapter, the system architecture will be presented shortly, then, in more detail, the software architecture for creating functional blocks, connecting them, and executing such a configuration. Alternatives will be described shortly, with a short reasoning on the choice for this architecture. Later in the chapter, the blocks needed for the scenarios will be identified and specified.

### 4.2 System Architecture

The system to be built consists of two machines, connected via a network, with an operating system (OS). This operating system has a kernel that takes care of the execution of user software, and providing interfaces with peripherals as the network interface card, a harddisk for storage and a display device.

The kernel is equipped with pre-emptive multitasking for the semi-parallel execution of user programs in so-called processes. Choosing which process should be given opportunity to execute is called scheduling. The operating system also provides a way for processes to divide their tasks over multiple threads (sometimes called lightweight processes), again using a pre-emptive multitasking scheduler, with the difference that these threads can access all of the process' resources.

### 4.3 Architectural Pattern

In determining an overall software architecture for a system that fits its requirements closest, it is good to look at so-called *architectural patterns* [POSA1]. Such patterns form the fundamental

structure for software systems. The architectural pattern that closest matches the requirements for Volund is the *Pipes and Filters* pattern.

The system to be built *is* meant for processing streaming data. To take the simplified version of the first scenario (just one layer), we want to read a file, split it in smaller parts, extract data from that *stream* to make a sequence of network packets, send those packets over a network, receive them on the other side, make it into a stream of video data again, decode it into a sequence of pictures, and display it. Each of those steps are incremental in their processing as they do not require all of the input to be read before an output can be done.

Also, some of the steps should be interchangeable. Like in the second scenario, in stead of using UDP as the transport protocol, we would like to use a modified TCP protocol.

Besides processing data streams, the Pipes and Filters pattern's benefits are: flexibility by filter exchange, flexibility by recombination, reuse of filter components, rapid prototyping of pipelines and efficiency by parallel processing. These benefits largely overlap the demands to the system to be built, so it seems the Pipes and Filters architectural pattern is really suited to form the basis of the system.

## 4.4 Pipes and Filters

As said before, the Pipes and Filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a *filter* component and data is passed through *pipes* between adjacent filters.

Three types of components can be identified. A *data source* provides the input of the system (e.g. a file). On the other end, the output flows into a *data sink* (e.g. a file or the screen). In between there are filter components. These perform a specific transformation step. The output of one component is the input of another using a pipe, consuming and delivering data incrementally to achieve low latency and parallel processing. A sequence of connected components is called a *processing pipeline*. Variants of this pattern allow for filters with more than one input or more than one output. Processing can then be setup as a directed graph.

Filters can be passive or active. A passive filter either has its output data pulled by the following pipeline filter, or have its input date pushed from the preceding filter. Active filters, on the other hand, execute a loop that actively pulls its input and pushes its output.

In a particular configuration of processing pipeline, components can be easily exchanged without having to change the others. Recombination of components makes it possible to build families of related systems. New applications can easily be prototyped by making a new configuration of needed components.

A very well known example of the use of Pipes and Filters is the uniform pipe mechanism in UNIX<sup>TM</sup> operating systems. UNIX<sup>TM</sup> features small tools like **cat**, **sort** and **tee** that can be combined using this pipe mechanism with a simple shell command. Each of these commands is run in a separate process.

The Pipes and Filters pattern also has drawbacks compared to other architectural patterns. For example, sharing state information is usually expensive or inflexible. All sharing must either take place by communicating using the pipes, increasing the amount of data flowing in the pipeline. It also creates dependencies between components, reducing to possibility of recombination with other components. Efficiency gain by parallel processing is often an illusion, for example because data transfer cost is relatively high compared to computations cost in a filter.

## 4.5 Pipelines & Components

Now we have chosen the Pipes and Filters pattern as our architectural pattern for the system, we need to identify all components in the system and describe them in detail. Each component will have its own function, with defined input and output and their relation, and communication behavior. These components will be part of a processing pipeline that matches the scenarios.

The components will be described using a model that will be defined in the next subsection. After that follows the description of the pipelines and the components according to the model. Some of the components in the pipelines are described in more detail in a separate section, because they require more prior knowledge and are more specific for the purpose of Volund.

### 4.5.1 Characterisation of a Component

A component is fully characterised by three aspects defined below: its input and output *ports*, the relation between inputs and outputs, and its *communication behaviour*. A component in our model is an abstraction of the component in the Pipes and Filters pattern, and makes these three aspects explicit.

Ports are the endpoints of a pipe. A component has as many ports as it shares pipes with other components. A port can be either active or passive. Active means that the component initiates the communication, it either *pulls* data from a pipe via an input port, or *pushes* data to a pipe via an output port.

The most common component is a filter with one passive input port and one active output port. This is also called a passive filter. For illustration, we will use an example passive filter called *increaser* that consumes and produces integer values, increasing the input values by one, using one input port *c* and one output port *d*.

The *I/O relations* state how the outputs on each output port are related to the inputs from the component's input ports. This is also where the transformation algorithm is specified. For our *increaser*, the I/O relations are simple: each consumed input  $c(i)$  (with  $c(i)$  being the  $i$ th input on port *c*) is increased by 1, so that the output  $d(i) = c(i) + 1$ .

Finally, the communication behaviour specifies how and when inputs are acquired and outputs are produced. Besides the activeness or passiveness of ports, it also says when communications take place relative to each other. Our example filter, like most passive filter components, acquires one input, computes the next output value and outputs that, so that corresponding inputs and outputs (according to the I/O relations) are interleaved.

The communication behaviour in this model is specified as regular expression of the input and output actions. Such an action names the communication port by name, followed by the type of action. A *!* denotes a write action, a *?* denotes a read action. Sequential composition is denoted by *;* and *,* denotes parallel composition.

A sequence of actions can be grouped into a subexpression by putting it in parentheses. Alternatives are separated by a *|* and repetitions by adding a *\** at the end of a (sub)expression.

The communication behaviour for our example is specified as:

$$(c?; d!)*$$

This means that one input is read, followed by one output. These both actions are repeated, in that order, indefinitely.

Each component will also be graphically represented as in [Figure 4.1](#). We have 4 kinds of component: a data source, a data sink, a filter with a one-to-one I/O relation and communication behaviour and a filter that accepts input values until it has gathered enough data to produce an output. The first three are represented as a rectangle with rounded corners, the last one as a rectangle with an extra vertical line on the left side (which suggests internal buffering). The pipes

connected to a component are represented with arrows that show the direction of the data flow. These are attached to the ports of the component, which are represented as circles. Active and passive ports are represented as respectively filled and hollow circles attached to the component.

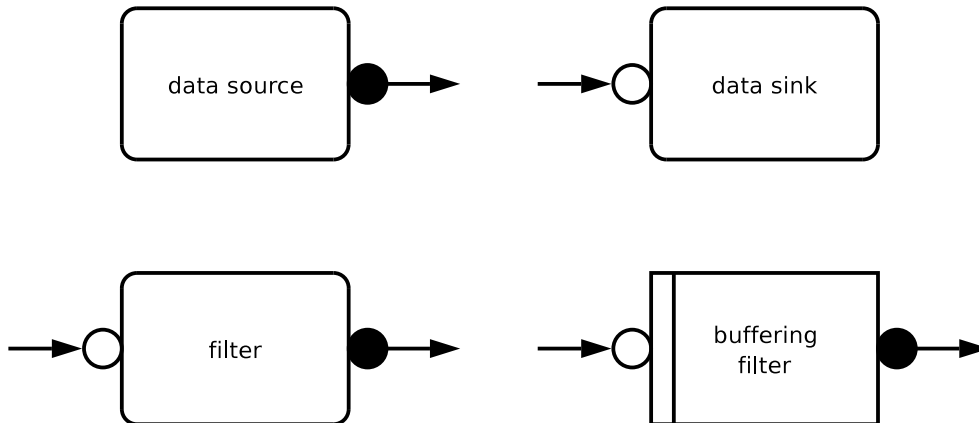


Figure 4.1: Graphical representation of components

Our example is represented by the graphic marked `filter` in [Figure 4.1](#), and we see that our input port is passive and our output port is active: we wait for a value to arrive, compute the new value, and actively push this value as our output. The following pseudo code show this:

```
proc Increaser( c: ? int, d: ! int) =
| [ var i: int
|   *(await(probe(c?))
|     ; c?i
|     ; d!i+1
|   )
| ]
```

In the following sections, each of the scenarios is represented as a pipeline of components and these components are defined with the aspects described above.

The pipelines are shown as Data Flow Diagrams and use the same graphics as for the components, but do not show the ports. The pipes are represented again as arrows, along with a characterisation of the data flowing through the pipe. Thick arrows represent out-of-band data flows. That is, data not managed as part of the Pipes and Filters pattern, but for example network traffic. Dotted arrows represent signaling between components, outside the normal flow of data.

## 4.5.2 One layer

Scenario 1 has many requirements, so we first start by identifying the components of a simplified version of this scenario. In this simplified version only one video layer, read from a file, will be transported using RTP to another party. No feedback on the reception quality will be communicated. [Figure 4.2](#) shows the components involved. For this scenario, we assume that UDP packets that arrive on the receiving side will be in the same order as transmitted. UDP itself has no concept or packet order, so this is a limitation on the UDP implementation and how the network behaves. However, it simplifies our design.

The diagram can be divided in two parts. The left part is the sending side, and the right part is the receiving side. Each side is one processing pipeline, both connected via a UDP socket. This UDP socket is not really part of the processing pipeline, but is abstracted as a pair of components:

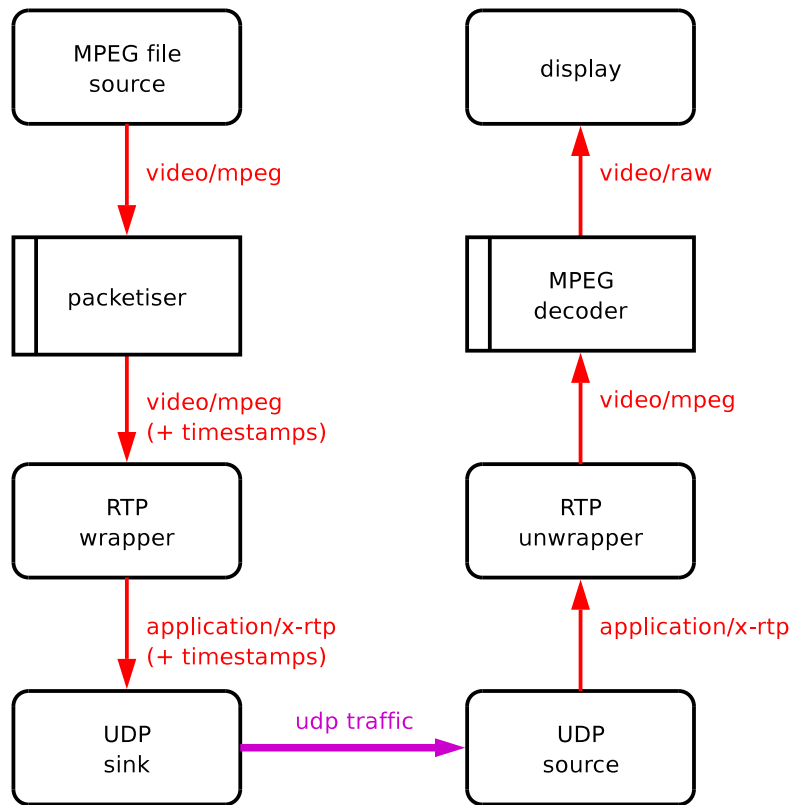


Figure 4.2: Data Flow Diagram for one video stream

a data sink on the sending side, and a data source on the receiving side. These two components interact with their Operating System to communicate with each other.

In the top left corner, an MPEG file is read from disk and streamed into the system. Then, a packetiser fragments the stream into packets with certain properties that fit in the payload of an RTP packet. It is described separately in [Chapter 3, “Streaming MPEG using RTP”](#).

After packetising, RTP packets can be created by adding RTP headers. This is done by the RTP wrapper. Finally, for the sending side, the stream of RTP packets goes into the UDP sink, that sends out the packets in UDP datagrams using the Operating System.

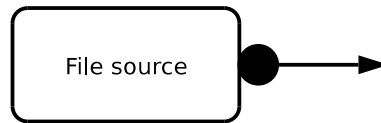
In the bottom right corner, the UDP source receives datagrams from the Operating System, extracts the RTP packets and sends them on to the RTP unwrapper.

The RTP unwrapper removes the RTP headers, and outputs an MPEG stream. This stream is then decoded by an MPEG decoder, and shown to the user using the display component.

We now have eight different components, specified in the following sections.

#### 4.5.2.1 File source

The file source component is a data source with one output port `o`. It interfaces with the Operating System to read a file from disk and push out equal sized data fragments (sequences of bytes) from the file in the order the file is read. Thus, the output port is active.



The fragments are of small size (e.g. 1024 bytes) because of the low latency requirement. Filters have to process their data incrementally and feeding the data in small pieces assists in this.

Each output value is one fragment (of configurable size) of the file, and the output values have the same order as the data is found in the file. The output behaviour is:

$(o!)*$

#### 4.5.2.2 Packetiser

The Packetiser is a filtering component that repacketises its MPEG input stream. The rules of packetising the incoming MPEG stream are described in [Section 3.3.1](#).

It has one passive input port  $i$  and one active output port  $o$ . Once enough MPEG data has been read for a new output, it pushes that value on its output port. This means that it does not necessarily do an output upon each input. It buffers the inputs internally as long as more data is required.



The input values are fragments of MPEG (sequences of bytes without constraints on the size, but preferably small enough to achieve low latency), and the output values is a structure of the display timestamp, MPEG specific RTP headers and RTP payload. We call this structure `preRTP`.

The display timestamp specifies the logical time since the beginning of the video data, for the picture to which the first byte of the RTP payload belongs. The RTP payload is a fragment of the MPEG input stream using the rules for packetisation. The MPEG specific RTP headers contain metadata about the payload for easier handling later on without having to parse the MPEG data. The format of these headers is described in RFC 2250.

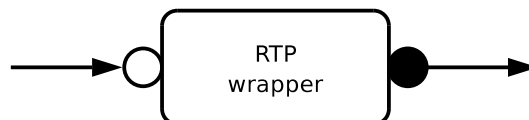
The display timestamp and MPEG specific RTP headers are needed to create an RTP packet. They are generated here to avoid having to reparse the MPEG data in RTP Wrapper.

Each input value is an MPEG fragment, and each output value is a `preRTP` structure. The communication behaviour of the packetiser is described by this expression:

$(i?; i?*; o!; o!)*$

#### 4.5.2.3 RTP Wrapper

The RTP Wrapper has one passive input port  $i$  and one active output port  $o$ . The input values are `preRTP` structures (consisting of a timestamp, MPEG specific RTP headers and a fragment of MPEG) as created by the Packetiser component. The output values are RTP packets that contain the input MPEG data as payload and has the RTP headers set.



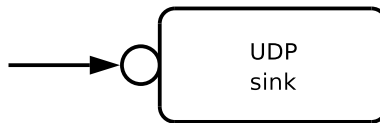
The RTP timestamp is created from the display timestamp in the input structure. The MPEG specific RTP headers in the same structure, are copied as is. Other headers that need to be set are the RTP version, the payload type, the SSRC, and the sequence number, as mentioned in [Section 3.2](#).

The communication behaviour of the RTP Wrapper is described by the following expression. Each input value is a `preRTP` structure, and each output value an RTP packet.

$$(i?; o!)*$$

#### 4.5.2.4 UDP Sink

The UDP Sink has one passive input port `i` and uses the incoming data packets to send them out using an UDP socket.



Each incoming data packet corresponds with one UDP datagram. It is expected that the resulting UDP packets do not exceed the MTU of the network layer. The destination IP address and port are configurable.

The communication behaviour of the UDP Sink is described by the following expression. Each input is a sequence of bytes, which will be the payload of one UDP datagram.

$$(i?)*$$

#### 4.5.2.5 UDP Source

The UDP Source interacts with the Operating System to listen to a (configurable) UDP port and push received UDP packets on its only output port `o`. Each UDP packet corresponds with one data packet in the pipe.

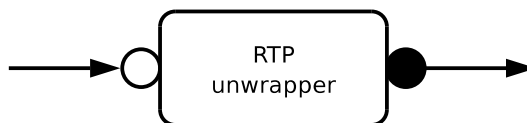


The communication behaviour of the UDP Source is described by the following expression. Each output value is a sequence of bytes with the payload of the UDP datagram.

$$(o!)*$$

#### 4.5.2.6 RTP Unwrapper

The RTP Unwrapper component has one passive input port `i` and one active output port `o`. It receives RTP packets as input values and extracts the MPEG payload. This MPEG fragment is then pushed on the output port.



The communication behaviour of the RTP Unwrapper is described by the following expression. Each input value is an RTP packet, and each output value is a fragment of MPEG data.

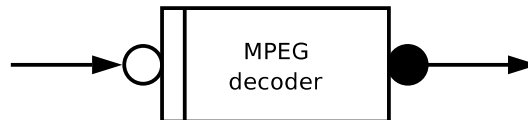
$$(i?; o!)*$$

This design of the RTP Unwrapper makes no assumptions on how the input RTP packets have been packetised. It does not use the MPEG specific RTP headers or the RTP headers. It just extracts the payload and sends it on. An MPEG decoder further down the pipeline only uses the MPEG data, and does not need the extra meta data to decode the stream.

In the multi layer case, however, the RTP Unwrapper needs some modifications, and will use the meta data in the RTP headers and MPEG Specific RTP Headers for handling loss. The final design is thus explained in [Section 4.5.4.2](#).

#### 4.5.2.7 MPEG Decoder

The MPEG Decoder takes an incoming MPEG stream and decodes it into a stream of bitmap pictures. It has one passive input port *i* and one active output port *o*. The input values are fragments of an MPEG stream (in the same order as the original file) and the decoder consumes as many fragments as it needs to decode the next picture. The output values are complete pictures.



If the input stream has missing or faulty data, the output pictures can be distorted or even missing.

In [Section 3.1.2](#), we saw that in MPEG streams B pictures are dependent on I and P pictures. In transport order B pictures are preceded by the pictures they depend on.

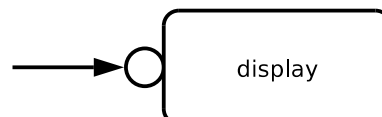
For example, I pictures can always be decoded independently of other pictures. Since the dependent B pictures that precede them in display order, come after the I picture in the stream, the MPEG decoder has to wait with outputting the decoded I frame, until it has decoded and output the B pictures that depend on it. That way, the output pictures are in the display order.

The communication behaviour is described with the following expression. Each input value is a fragment of MPEG data, each output value is a YUV 4:2:0 picture.

$$(i?; i?*; o!; o?)*$$

#### 4.5.2.8 Display

This data sink component has one, passive, input port *i* that receives a sequence of raw video images and displays each image in a window immediately. Each input is a complete image.



The communication behaviour is characterised by the following expression. Each input value is a YUV 4:2:0 picture.

$$(i?)*$$

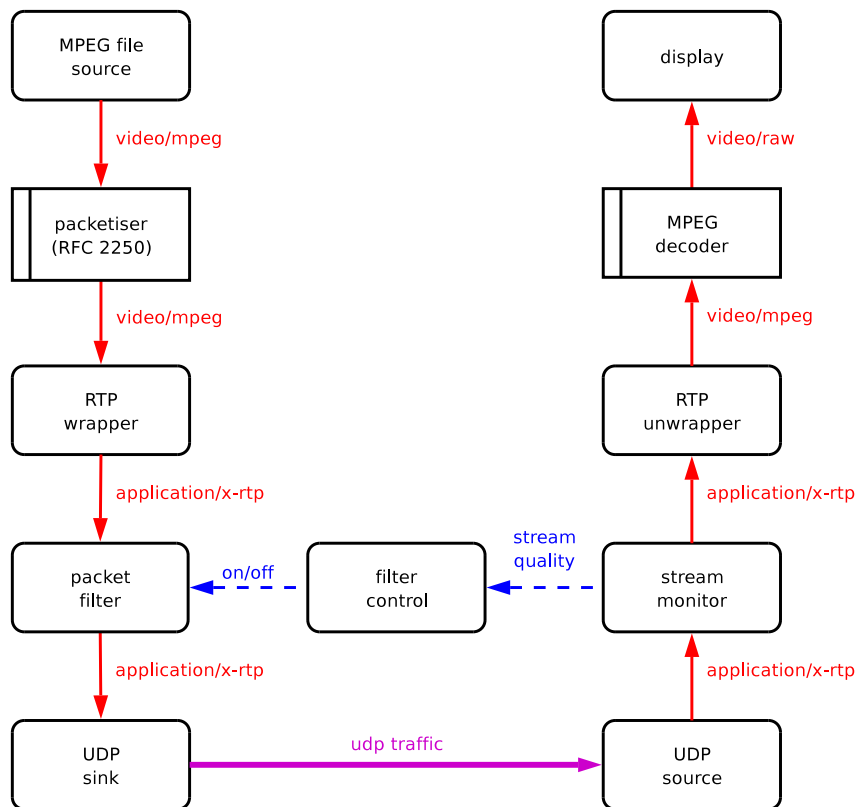


Figure 4.3: Data Flow Diagram for one video stream

### 4.5.3 One layer with Feedback

Figure 4.3 shows an extended version of the pipeline from the previous section. It now has a reception quality feedback mechanism, that monitors what has been received and controls if it is still desirable to send the data in the first place.

This feedback mechanism is useful for the multilayer pipeline described later, but presenting it for one stream makes it clearer what needs to be done. For example, when many packets are lost and no good quality images can be produced, sending nothing at all can make the reception of other layers better, because less data is sent on behalf of the whole application.

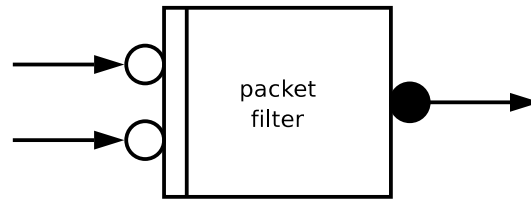
The new components are the Packet Filter that controls whether to pass on incoming data, the Stream Monitor that keeps statistics on incoming packets, the Filter Control that uses the statistics from Stream Monitor to decide whether to let the sending side send out its data.

We still assume the incoming packets on the receiving side are in the same order as transmitted.

The following sections describe the extra components in more detail. The last section contains some remarks on this pipeline.

#### 4.5.3.1 Packet Filter

This component has two input ports `ci` and `di` and one output port `do`. The component has two states: `pass-through` and `block`. Every time a new boolean value arrives on the control channel `ci`, it sets the state to `pass-through` for a `true` value, and `block` for a `false` value.



If the state is `pass-through`, each value received on input port `di` is directly pushed on output port `do`. If the state is `block`, the component discards all input values arriving on port `di`.

Figure 4.4 shows the process in pseudo code.

```

proc PacketFilter( di: ? pack, do: ! pack, ci ? bool) =
| [ var s: bool, pct: pack
| s := true
; *(await(probe(ci?) ∨ probe(di?))
; if probe(ci?) then ci?s fi
; if probe(di?) then
di?pct
; if s then di!pct fi
fi
)
] |

```

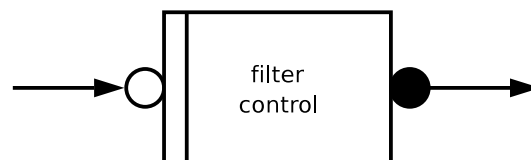
Figure 4.4: CSP Pseudo Code for Packet Filter

The communication behaviour of the Packet Filter component is described by the following expression. The input values on the `di` input port, and the output values on the `do` are sequences of bytes. Each input value on the `ci` input port is a boolean.

$$((ci?|di?)*; do!)*$$

#### 4.5.3.2 Filter Control

The Filter Control component receives quality information from Stream Monitor and decides whether the sending side should send data or not. How the input information is structured and processed is one of the research topics which Volund is a vehicle for. Here we abstract its function as a component with one passive input port `i` and one active output port `o`. Each output value is a boolean.



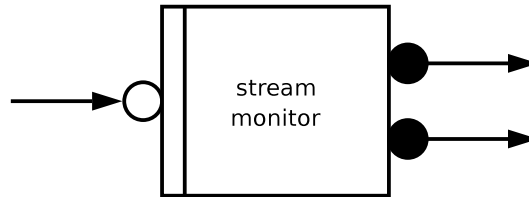
Since this is an abstraction, implementers should decide what kind of inputs will be used, and how they are structured. Here we used one input port, but it is likely more than one source of data is used to make a decision on whether data should be sent or not. Also, one could decide to use one component that controls multiple Packet Filter components, thus having more than one output ports.

The input values are structures that describe the statistics on the reception quality as produced by Stream Monitor.

### 4.5.3.3 Stream Monitor

The Stream Monitor has one passive input  $di$  and two active output ports  $do$  and  $so$ . It collects statistics on the incoming RTP packets and outputs structures with these statistics on output  $so$ . It pushes the RTP packets unmodified on output port  $do$ .

Just like the Filter Control component, this component is subject to research. What information is being collected and how the statistics are structured is to be defined outside this document. Based on the incoming RTP packets it is able to count received packets, lost packets and packets that arrived too late to be of use further down the pipeline.



Without more explicit knowledge on the function of this component, the communication behavior of the Stream Monitor is best described with the following expression:

$$(di?; do!, so!)*$$

### 4.5.3.4 Remarks.

- The Packet Filter component can also be put between Packetiser and RTP Wrapper or between MPEG File Source and Packetiser. For this instance of the pipeline, the choice is made to be the current position.
- Although this pipeline provides for *monitoring* dataloss, it does not take this loss into account for further processing. As stated before, the MPEG decoder can produce distorted images when it does not receive all data needed for decoding. In view of the use of this pipeline, it would be more useful to gather all RTP packets belonging to the same MPEG picture, and only send the MPEG data on when the data is complete.

Collecting all RTP packets belonging to one picture makes sure only complete images are produced by the MPEG decoder. Of course this should also take into account the relation between pictures in an MPEG bitstream regarding dependencies. We will introduce a solution to this in the multiple layer case.

## 4.5.4 Multiple layers

Now we have the components and pipelines for streaming one regular video stream over the network, we can move to the scalable video case.

As explained in [Section 3.4](#) each layer of the scalable video is sent in its own RTP stream. Each of these streams are also using MPEG encoding, although the enhancement layers really only contain the image enhancement information. Many of the components needed for the single stream case, can be reused for the multi stream case.

There are different algorithms for layered video. We use SNR scalable video (see [Section 1.3](#)), of which the encoding algorithm is such that corresponding pictures (after decoding) can be merged into one resulting picture, without dependencies on following or preceding pictures.

For the sending side, we use the exact setup for each layer, but multiplied by the number of layers. So for two layers, we run two identical pipelines like the left side of [Figure 4.2](#) which are

started simultaneously. The only difference is in the file each of them reads and the destination port number for the UDP socket<sup>1</sup>.

On the receiving side, the layers need to be combined again to finally output one sequence of pictures. Figure 4.5 shows the pipeline for two layers. There is one new component Adder and the RTP Unwrapper component is modified to address the issues raised in Section 4.5.3.4.

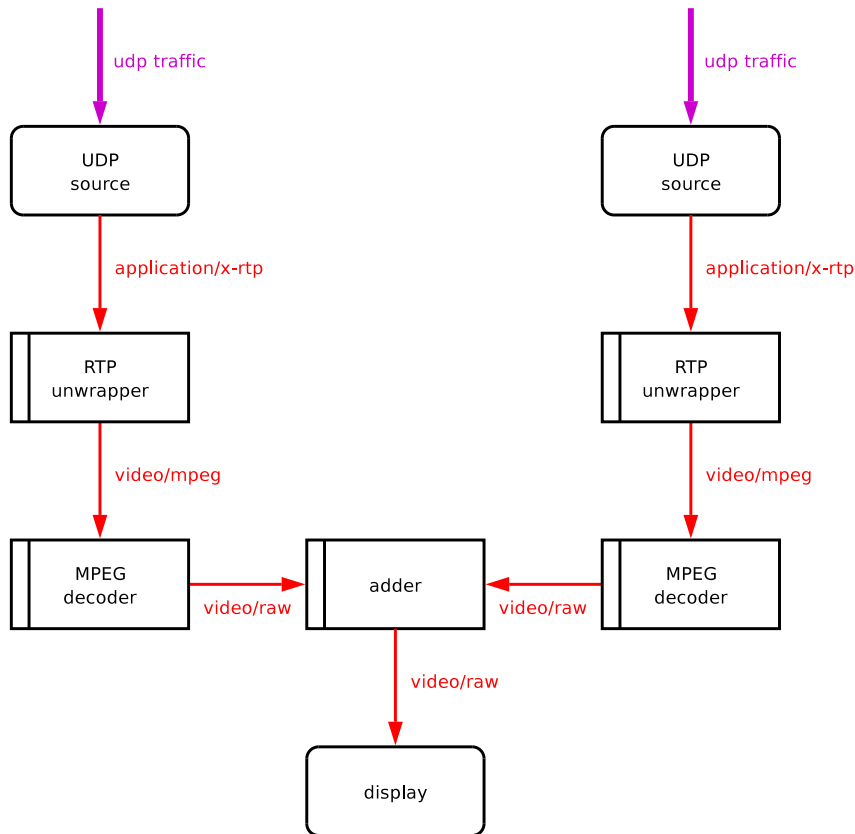
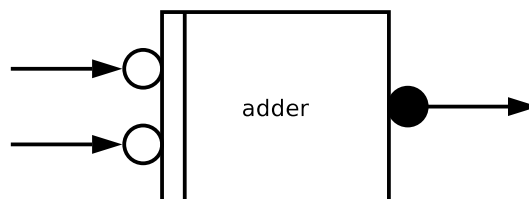


Figure 4.5: Data Flow Diagram for two video streams without feedback, receiver side

On the top of the figure, the UDP traffic from the two sending pipelines flows into the receiving UDP source components. Then, from each of the RTP streams the MPEG data is extracted and decoded. The resulting picture sequences are finally merged into one combined picture sequence by the Adder component and output by the Display component.

#### 4.5.4.1 Adder

The Adder component has two passive input ports *i1* and *i2* and one active output port *o*. It matches the incoming inputs, which are video images, according to their timestamp and merge them into one video image that is then pushed on the output port.



<sup>1</sup>How these parameters are set, is up to the application that controls the execution of the pipelines

The Adder component has two functions: to synchronise the input picture sequences (as described in [Section 3.4.1](#)) and recombine corresponding pictures from its inputs into one picture (as described in [Section 3.4.2](#)).

The input sequences are synchronised using timestamps. Each picture carries such a timestamp as metadata, and corresponding pictures have the same timestamp.

The recombination is the reverse operation of the process of splitting a picture sequence over two layers: a base layer and an enhancement layer. The pictures from one of its input ports represent the base layer and are regular pictures, the pictures from the other input port represent the enhancement layer. The latter pictures contain the difference between the original pictures and the pictures from the base layer. Combining the corresponding pictures in each layer results in pictures with a higher quality than just the base layer.

The format of the pictures is chosen to be YUV 4:2:0 because the available method of generating layers assumes this format. Addition then is a simple operation: adding each corresponding byte from each image and subtracting 128. If the result is smaller than zero, it is set to zero. If the result is larger than 255, it is set to 255. The resulting picture, also in YUV 4:2:0 format represents an image from input port  $i_1$ , enhanced with data from input port  $i_2$ .

The input ports are passive. This means that the component needs to buffer the inputs until the matching picture from the other port is received. It can also happen that pictures are missing from one of the layers because of data loss earlier in the pipeline. We want an input/output behaviour as visualised in [Figure 3.4](#). Therefore, the component is equipped with two queues,  $q_1$  and  $q_2$ , one for each input port. All inputs are expected to be in chronological order (according to their timestamp), and on each input, the following process, visualised by the flow chart in [Figure 4.6](#), is executed:

1. Append the input on the tail of the ports' queue (we assume inputs are ordered, see [Section 4.5.2](#)).
2. If one of the queues is empty, exit.
3. Looking at the timestamp of the pictures at the head of each queue ( $t_1$  and  $t_2$ ), do one of the following:
  - If  $t_1 = t_2$ , remove both pictures from their respective queues, combine them and output the result.
  - If  $t_1 > t_2$ , the base layer is ahead of the enhancement layer. This means the picture that corresponded to the picture from the enhancement layer has not been (and will not be) received (we assume the inputs are ordered). Therefore, the enhancement data has no use. We remove the picture from  $q_2$ , discard it, and go back to step 2.
  - If  $t_1 < t_2$ , the enhancement layer is ahead of the base layer. Because the inputs are ordered, we conclude the picture with the enhancement data has not been received. We remove the picture from the head of  $q_1$ , push it on the output port (instead of an enhanced picture), and go back to step 2.

The communication behavior of the Adder component is described with the following expression:

$$((i_1?; o!*) | (i_2?; o!*))^*$$

From the expression we see that we can have an infinite amount of inputs before any output occurs. More specifically, if no inputs are received on one of the channels, no outputs will occur. This could be countered by for example introducing queues with a limited size or by comparing the timestamps of incoming pictures with some local clock and dropping pictures that are “too old”.

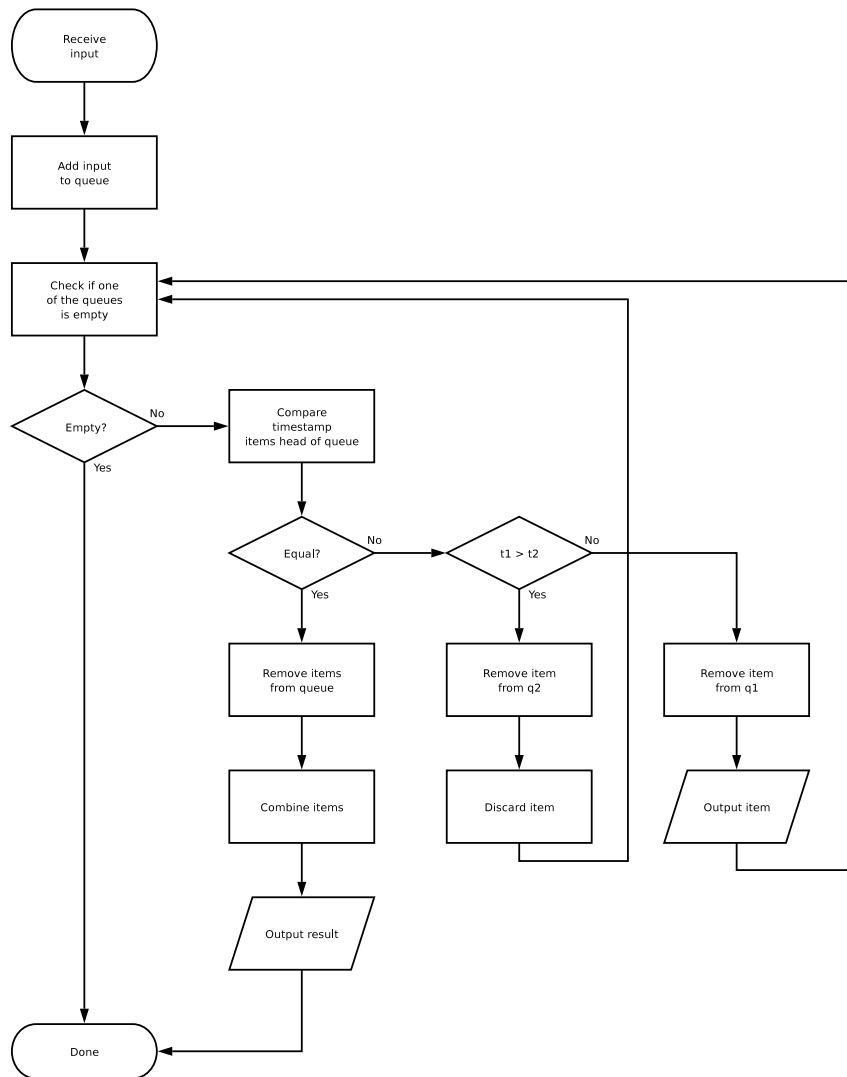


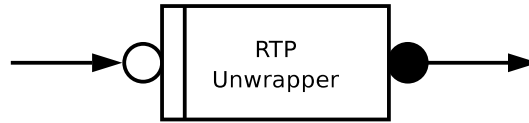
Figure 4.6: Flow chart of the Adder component

#### 4.5.4.2 RTP Unwrapper

In [Section 4.5.3.4](#) it is mentioned that one MPEG picture is likely to be spread among various RTP packets. If one of those RTP packets is lost, the picture resulting from decoding the MPEG data from the other RTP packets of the same MPEG picture, will be damaged.

To counter this, we modify the RTP Unwrapper component to collect all RTP packets that contain the data for the same MPEG picture and concatenate the extracted MPEG data into one MPEG fragment that is only output if all data has been received. This ensures that only complete MPEG pictures are decoded and output by the MPEG decoder.

This means the communication behaviour of the RTP Unwrapper changes, and the graphical representation of the component also needs to be adjusted to the fact that the component now buffers inputs. The ports remain the same.



The matching communication behaviour is described by the following expression:

$$(i?; i?*; o!)*$$

The gathering of the RTP packets belonging to the same MPEG picture is done using the information in the RTP headers. The component keeps a buffer containing the MPEG data of current picture received so far. All packets with the same RTP timestamp belong to the same MPEG picture.

The RTP sequence number is used to detect gaps in the RTP stream. When such a gap is detected, the data in the buffer is discarded and all incoming RTP packets are also discarded until the beginning of a new picture is detected. As soon as the data for one MPEG picture is completely acquired, this MPEG fragment is pushed on the output port.

Detecting the beginning and end of the MPEG picture is difficult with the headers defined by RFC 2250. Although it defines markers for the beginning and end of *slices* in the MPEG specific RTP headers, it does not have such markers for pictures. These begin and end of slice markers can be used to also detect the beginning and end of pictures, since the beginning of a picture coincides with the beginning of slice and the end of a picture with the end of a slice. But you would have to find out which slice in the picture you are dealing with to determine where they are in the picture. This information is not in the MPEG specific RTP headers, so scanning the MPEG data is necessary to do this. And even if you do this, you have to remember how many slices are in one picture, since this varies with the picture resolution (specified in the MPEG sequence header).

Another approach is to modify the Packetiser component on the sending side to put begin and end of picture markers in the MPEG specific RTP headers. There are five unused bits in these headers, so we define bits 3 and 4 of the first byte to contain this information, respectively holding the begin of picture and end of picture marker. The RTP Unwrapper then uses these bits to gather all RTP packets with the same timestamp, beginning with the packet that has the begin of picture marker set, and ending with the packet having the end of picture marker set.

#### CAUTION



This modification of the protocol defined by RFC 2250 makes it non-standard. However, since the used bits are reserved, compliant implementations of RFC 2250 should simply ignore these bits and function normally.

#### 4.5.4.3 Remarks

**More than two layers** The pipeline presented above only addresses scalable video using two layers. To use more than two layers, there are multiple solutions.

The Adder component could be modified to have any number of inputs and combine them incrementally. The algorithm to determine which pictures to select from the queues and merge them into one resulting picture is more complex than the two layer case.

Another solution is to use the components we already have and deploy multiple adder components, so that the output of each adder is an input of the next adder. [Figure 4.7](#) shows the three

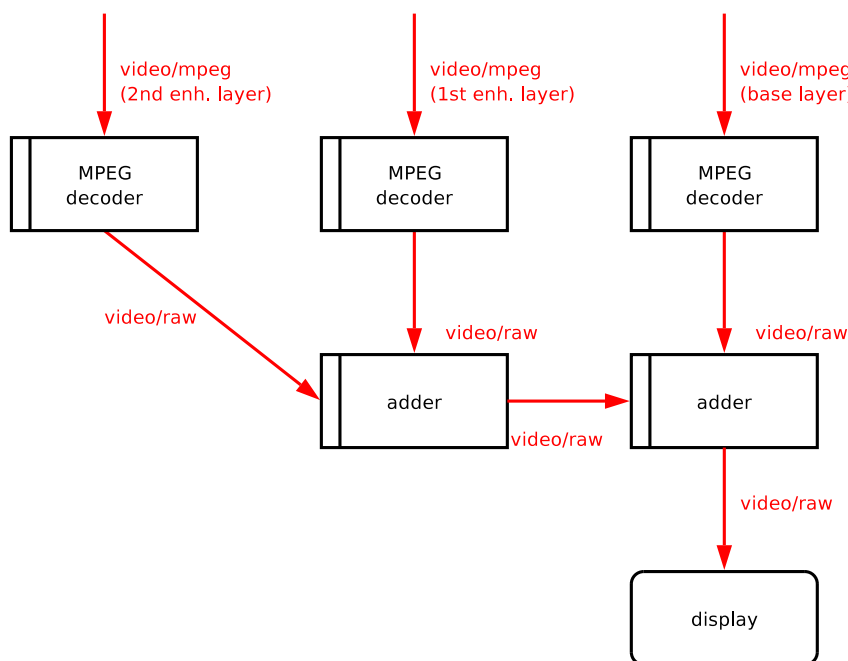


Figure 4.7: Data Flow Diagram for three video stream, receiver side from just before MPEG decoding

layer case using this approach, showing the pipeline after the MPEG stream has been extracted from the RTP streams.

The right stream is the base layer, the middle stream the first enhancement layer, and the left stream is the second enhancement layer. The first adder (on the right), merges the base layer and first enhancement layer, and the result is added to the second enhancement layer.

As we have seen in the description of the Adder component, the synchronisation of the pictures is done using the timestamps that accompany the pictures as meta data. The combined picture also carries this timestamp as meta data, and can thus be used to synchronise the result after the first enhancement with the second enhancement data.

This approach shows the strength of using the Pipes and Filters pattern. With the same set of components new or extended applications can be easily created.

### 4.5.5 Using Modified TCP

The second scenario (Section 2.1.2) describes using another transport protocol in stead of UDP: a modified TCP. Since all the modifications to TCP are transparent for the rest of the (distributed) application, we can reuse most of the components we described in Section 4.5.2.

The only thing different from the first scenario (the simplified case using one layer without feedback) is the use of TCP. We can simply replace the UDP Sink and UDP Source components with their TCP equivalents. This is shown in Figure 4.8.

The implementation of the pair of TCP components should be such that they accept RTP packets on the one side and output RTP packets again on the other side. Since TCP is not datagram oriented like UDP, but uses the abstraction of a continuous byte stream, the RTP packets need to be *framed* somehow so that the receiving side can detect the beginning and end of an RTP packet. Possible data loss complicates this. This makes the implementation of this set of components

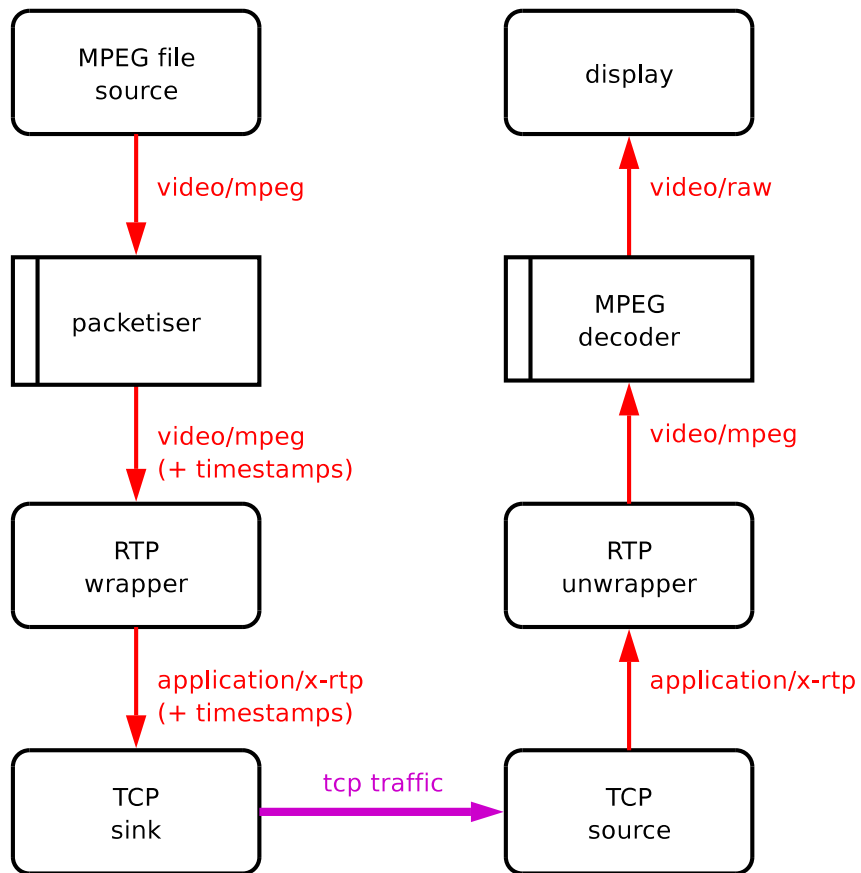


Figure 4.8: Streaming one video stream using TCP.

non-trivial. Also, this component should output its data at the last possible moment, to allow for retransmissions, when there is still time.

All TCP details and modifications wanted by the researcher can be contained in this pair of TCP components, while reusing the rest of the components.

The approach of exchanging the pair of UDP components for their TCP counterparts, shows the fast prototyping and reusability strengths of the Pipes and Filters architectural pattern.



## Chapter 5

# Design & Implementation

The architecture laid out in [Chapter 4, “Architecture”](#) gives a clear path to a design and implementation of Volund. However, we can choose to design and implement this architecture completely from scratch, or build on existing software. Choosing an existing *Media Processing Framework* that matches the ideas presented in the architecture description reduces the implementation effort significantly.

This chapter will describe what a Media Processing Framework entails, together with a number of examples of implementations of such frameworks. After that, the features of and strengths of the chosen framework, GStreamer, will be discussed, and why it is suitable as basis for Volund, how the model presented in [Section 4.5.1](#) maps to GStreamer, and how the components can be implemented using GStreamer. Also, issues surrounding the use of GStreamer, and other findings will be given in the last section of this chapter.

### 5.1 Media Processing Framework

Implementing the Pipes & Filters pattern can be done as a number of objects, connected to each other in the main program by connecting their interfaces at compile time. With some effort, maybe even at runtime. However, one could also think of implementing a more flexible environment, where components can be connected in arbitrary graphs dynamically.

If the components in such an environment would also have a more or less generic interface, adding new components for increased functionality can be much easier.

Several of such environments exist, targetted at media processing, like our system. Such a *Media Processing Framework* typically have a *plugin* system for easy addition of components, an execution mechanism that allows for the execution of the processing pipelines. It manages the flow of data between the components and usually provide a set of *basic components*. Basic components can be components that interact with the Operating System (files, displaying images) or make debugging easier.

Using such a framework as a basis for our system greatly reduces the work to be done because instead of having to build everything from scratch, you can focus on the most important part: implementing the filters and combine them into a processing pipeline that suits our requirements.

Examples of these frameworks are TSSA (TriMedia Streaming Software Architecture, for embedded software using on-chip media processing, by Philips), QuickTime (for desktop media processing on MacOS and also MS Windows, by Apple), and GStreamer (media processing framework of choice for the GNOME desktop environment, used in a number of different applications). All of these are targetted at different platforms, being specifically used on certain hardware or operating system.

One of the constraints of the QOSIH team was to have a system that runs on GNU/Linux systems. Since GStreamer is being developed for the GNOME desktop environment, running on GNU/Linux, FreeBSD and a number of other UNIX like operating systems, the choice was made to use GStreamer as the basis for building Volund.

## 5.2 Introduction into GStreamer

What is GStreamer? From the GStreamer website<sup>1</sup>:

GStreamer is framework for creating streaming media applications. It allows for the construction of graphs of media-handling components, ranging from simple mp3 playback to complex audio (mixing) and video (non-linear editing) processing.

GStreamer is written in the C programming language using an Object Oriented library called GLib. GStreamer comes with the following features:

**Functional blocks** The basic building block in GStreamer is called *Element*. Examples are data sources, data sinks and filter components. An element can also take the form of a container of elements called *Bin*. GStreamer has two standard Bins: *Thread* and *Pipeline*. The most used Bin is a Pipeline. Because the containers are also elements, Bins can contain other bins, with the toplevel Bin being a Pipeline. Elements are connected to eachother by linking their input and output ports called *Pads*, which are unidirectional. By linking elements, arbitrary graphs of functional blocks can be created to form an application.

**Plugins** Most of GStreamer's functionality is implemented in shared libraries known as *plugins*. GStreamer's core only contains some very basic functions, mostly for registering and loading plugins, and providing the fundamentals of all classes from which the classes in the plugins derive. A plugin derives from the `GstPlugin` class, which allows for the registering of pluggable objects called *plugin features*. The most common type of plugin feature is an element, and a plugin implements an element by subclassing `GstElement` and registering this new class as one of the plugin's features.

A plugin can contain any number of elements. Usually similiar elements are grouped in one plugin, for example because they share a common external library. Besides elements, other things can also be implemented in a plugin. A nice example of this is a scheduler, which manages how and when elements are activated and data flows between them.

GStreamer has an extensive number of existing plugins available. The basic GStreamer framework comes with a plugin of basic elements. These include a file source, a file sink, a tee filter (for 1-to-N *pipe fitting*), an aggregator filter (for N-to-1 pipe fitting) and a set of debugging elements called fake source and fake sink.

Then there is a large set of plugins that are part of the `gst-plugins` package. These plugins implement audio and video codecs, effects, converters, de-/multiplexers, and data sources and sinks interacting with network, sound or display devices.

With GStreamer plugins there is a generic interface for wrapping existing media processing libraries (e.g. audio and video codecs), or hardware like video input cards and build new applications. GStreamer plugins can be loaded when setting up the pipeline, but also *while running*.

---

<sup>1</sup><http://www.gstreamer.net/>

**Type system** All data flowing between components is marked with a data type (using MIME types). When two elements are connected using their pads, it is important that they communicate using data types known to both. To see if two pads are compatible, each pad describes its capabilities.

A *capability* is a combination of a MIME type (eg. `audio/mp3` or `audio/raw`) and some properties about that type (in the case of `audio/raw` for example the sample rate and width). Pads with matching capabilities can be linked. To determine which data type to use in communication, both pads exchange their capabilities and a choice is made by a process called *feature negotiation*.

**Debugging system** The debugging system provided by GStreamer enables developers to monitor everything that goes on in the system. It reports on plugin loading, capabilities negotiation, connecting pads, pipeline states, data flow, etc. Also, using special elements like fake sink and fake source, the developer can see exactly when data packets are transferred and what is in them.

The debugging system is invaluable for creating new filters and applications.

Using an existing framework has many advantages over writing custom code from scratch, provided its model fits the desired architecture. GStreamer is not targeted at one particular application, but made to be used in any media streaming application, with the following design goals: minimal overhead, high performance, low latency, extensibility with a clean and simple programming interface.

Writing such a generalised framework is a lot of work. Being able to use it as-is, instead of a customised framework for just one application, greatly jump-starts development. It allows for reuse, recombination of functional blocks and fast prototyping. Also, having so many plugins available ready to be used makes it possible to focus on the actual problem at hand, without having to reinvent wheels. The components used for Volund are discussed in a later section.

## 5.3 The GStreamer Model

Like any system, GStreamer has a model of how components interact with each other. This section will present the GStreamer model and the next section will attempt to map the model laid out in the Architecture chapter to GStreamer's.

GStreamer's architecture is built around the Pipes and Filters architectural pattern described in [Section 4.3](#) and adheres to the description of Media Processing Framework from [Section 5.1](#). Just like in the model from the Architecture chapter, the basic idea for creating applications with GStreamer is to connect functional components (filters, sinks and sources) in a graph to form a pipeline (note that a pipeline in GStreamer is not necessarily linear, but can be an arbitrary graph of connected components). So, to have a good match between the two models, we need to compare the ports, I/O relations, and communication behaviour for the components in both models.

The components from our model are implemented in GStreamer elements. Elements have unidirectional ports called pads. Input ports are called *sink pads*, and output ports are called *source pads*. Pads can be linked to form pipes. Using these pipes data is transferred in data structures called *Buffers*. Each element defines functions which process the data going through that element, how and when these functions are called is determined by the scheduler discussed in [Section 5.3.2](#).

A GStreamer element has two types of elements:

**chain based** A chain-based element defines a so-called *chain function* for each sink pad. Every time a new input buffer is available for a certain pad, this chain function is called with a pointer to the receiving pad object and the buffer object. Since the chain function can determine which pad got an input, so multiple pads can share the same chain function. A chain function looks like below:

```
static void
chain_function (GstPad *pad, GstBuffer *buffer)
{
    GstBuffer *outbuffer;

    ....
    // process the buffer, create a new outbuffer
    ...

    gst_pad_push (srcpad, outbuffer);
}
```

The task of the chain function is to process the incoming buffer and possibly do an output on a source pad, or, if the element is a data sink, do something else, like displaying an image on the user's screen.

As this function is called whenever there is a value available, this can be compared to a passive input port as in our architectural model.

Chain-based elements are mostly used for, but not limited to, elements that have a one-to-one relation between their input and output communication behaviour.

**loop based** A loop-based element defines a *loop function*. This function is called continuously and runs indefinitely. This is shown by the following pseudo-code for a loop-based element with one sink pad and one source pad:

```
GstBuffer *buffer, *outbuffer;

static void
loop_function () {
    buffer = gst_pad_pull (sinkpad);
    ...
    // process buffer, create outbuffer
    while (!done) {
        ....
        // optionally request another buffer
        buffer = gst_pad_pull (sinkpad);
        ....
    }
    ...
    gst_pad_push (srcpad, outbuffer);
}

while (1) {
    loop_function();
}
```

Buffers can be requested element's sink pads using `gst_pad_pull()` whenever it needs them. Loop-based elements are comparable to filter from our model, having active input ports and active output ports.

Loop-based elements are mostly used when an element has multiple input ports or when more than one input action is required before an output can be done.

Output actions are always active, by calling the function `gst_pad_push()`. There is no way to call a function of another element to make it produce a value directly.

### 5.3.1 Element states

A GStreamer element (so also a complete pipeline) can be in four different states `NULL`, `READY`, `PAUSED` and `PLAYING`.

The `NULL` is the initial state when elements are created. Also, going to this states enables elements to reset themselves and free up resources. The `READY` state is used let elements initiate themselves to get ready to start processing data. For example, this is where a file source will open the file to be read. The `PAUSED` state is to temporarily stop data flowing through a playing pipeline.

When a pipeline (and its elements) is in the `PLAYING` state, data is flowing through the pipeline. Going from `NULL` to `PLAYING` always goes via the `READY` state.

### 5.3.2 Execution model

The subsystem in GStreamer responsible for calling the chain and loop functions mentioned above, is called the scheduler. The scheduler takes care of the data flows and activating the elements that can process a buffer. Schedulers in GStreamer are not pre-emptive; they can only do context switches in certain places. For chain-based elements, this is before and after a call to its chain function, for loop-based elements before and after each (repetitive) call to its loop function. Also, inside the calls to `gst_pad_pull()` and `gst_pad_push()` context switches can be performed. From the calling element's point of view, it just takes a long time before these calls return.

In GStreamer schedulers are also plugins, and by default GStreamer comes with two schedulers: the *basic* and the *optimal*. The basic scheduler (`basic`) is a simple, complete but unoptimised scheduler. The optimal scheduler (`opt`), on the other hand, is optimised for media processing, and thus somewhat more complex.

#### 5.3.2.1 The Basic Scheduler

The basic scheduler makes use of a form of light-weight, user space threads called *co-threads*. Every element is run in a co-thread, and the scheduler controls when each co-thread is being executed.

When one element pushes data, a context switch is made to the next element that handles that piece of data. And also, when an element requires a new input (by using `gst_pad_pull()`) execution of the requesting element is suspended and the scheduler switches to the element upstream.

This method of scheduling can cause much data to be queued between elements, when one element has produced data that is not processed yet by the following element. This queued up data is stored by the scheduler using so-called *bufpens*, holding pointers to buffer objects. This queued up data causes a higher latency in the pipeline. Also, for larger pipelines, the co-threads can be a burden for the system, because of the context switches.

### 5.3.2.2 The Optimal Scheduler

The optimal scheduler is a optimized scheduler that uses knowledge about what kind of elements are connected to eachother. It makes shortcuts in the execution mechanism by letting certain elements run together in one co-thread, and have these elements control eachother.

For example, when a pipeline with only chain-based filters is run, and one filter actively pushes a buffer using `gst_pad_push()`, that call is a direct call to the chain function of the following filter. The following is an example using filters 1 and 2, where the sink pad of filter 1 is linked to the source pad of filter 2. In pseudo code:

```
GstPad *sinkpad1, *srcpad2;

void
gst_pad_push (GstPad *pad, GstBuffer *buffer) {
    ...

    if (pad == sinkpad1) {
        chain_function2 (srcpad2, buffer);
        return;
    }
    ...
}

static void
chain_function1 (GstPad *pad, GstBuffer *buffer)
{
    GstBuffer *outbuffer;

    ....
    // process the buffer, create a new outbuffer
    ...

    gst_pad_push (srcpad, outbuffer);
}

static void
chain_function2 (GstPad *pad, GstBuffer *buffer)
{
    GstBuffer *outbuffer;

    ....
    // process the buffer, create a new outbuffer
    ...

    gst_pad_push (srcpad, outbuffer);
}
```

Because no context switch has to be made, this can lower the latency in the pipeline.

Unfortunately, the implementation of this scheduler is not complete. One of the problems is with elements with multiple inputs and/or outputs, these are not handled correctly by the current implementation of the optimal scheduler.

## 5.4 Existing GStreamer Elements

GStreamer already provides a lot of elements ready to be used by our system. Most of these are simple chain-based elements, which consume one buffer, process them into a new buffer, and push this buffer on its source pad. This is also called a simple filter and the passive components from our model can be easily mapped on a chain-based element. Also, most data sources and data sinks are respectively active on their output and passive on their input ports, just like in our model.

The table in [Table 5.1](#) shows the mapping of the components from our model to available GStreamer elements. The GStreamer elements behave like the components in our model and can be used as is, with the exception of `xvideosink`.

Component	GStreamer Element
File source	<code>filesrc</code>
UDP sink	<code>udpsink</code>
UDP source	<code>udpsrc</code>
MPEG decoder	<code>mpeg2dec</code>
Display sink	<code>xvideosink</code>

Table 5.1: Components from the architectural model mapped to GStreamer elements

The `xvideosink` element accepts buffers that contain images in RGB format, which is different from the display component in our model. However, the `mpeg2dec` element outputs images using the YUV 4:2:0 format, and is therefore not compatible with the `xvideosink` component. Fortunately, GStreamer comes with the `colorspace` element that translates images between different colorspaces. Using capabilities negotiation, it finds out which formats are used and required. Putting this element between the `mpeg2dec` and `xvideosink` solves the incompatibility of formats.

Using the above components, we can not build any of the pipelines for Volund yet. We can however, build a simple application that displays MPEG files locally. For fast prototyping and testing during development, GStreamer comes with a command line utility to construct and execute pipelines of elements. Each element can have parameters that influence their behaviour. For example, the `filesrc` element needs a *location* to specify which file has to be read. A command for playing an MPEG file named `movie.es` looks like this:

```
gst-launch filesrc location=movie.es ! mpeg2dec ! colorspace ! xvideosink
```

Elements are connected using the `!` character, and each element name is optionally followed by its parameters. Issuing this command results in a new window being opened and the video from the file being displayed.

## 5.5 Changed Elements

The `Packetiser` component as defined in our model, was not available as such in GStreamer. However, in the `mpeg-stream` plugin, there exists an `rfc2250enc` element.

Unfortunately, the code for the `rfc2250enc` element was incomplete and had a few bugs related to a changed API in GStreamer. In its original form, it only packetised the input MPEG stream according to the rules defined in RFC2250. The packetiser in our model, also outputs the MPEG specific RTP headers and timestamp information.

Leaving the `rfc2250enc` element as it is and adding a new element that generates the necessary meta data was considered, but that new element would have to parse MPEG all over again. The choice was made to modify the `rfc2550enc` to match the description in our model.

The original output format of the element was fragments of MPEG and this was changed to a structure with the following structure: 32 bits timestamp, 32 bits MPEG specific RTP header (as defined by RFC2250), followed by an MPEG fragment. This element fills in the timestamp and MPEG specific RTP header information using the information in the MPEG stream.

The timestamp is the time on a 90Khz clock of the display time of picture to which the first byte of the output packet belongs. It is calculated from the Group of Pictures' display timestamp and the Temporal Reference field in the MPEG Picture header of the picture to which the packet belongs. Since the beginning of a picture is always at the beginning of an RTP packet, there is only one possible timestamp for the sequence of packets belonging to one picture.

The other RTP and MPEG Specific RTP fields are filled using information from the MPEG picture header and the occurrence of the slice and sequence headers. Additionally, the extra fields for marking the beginning and end of a picture, as explained in [Section 4.5.4.2](#) are generated by this element.

#### CAUTION



Although RFC2250 defines optional header fields for storing MPEG 2 specific meta data, this is not implemented in this element. They are useful for reconstructing missing header information, but as explained in [Section 3.4.2](#) we only process pictures when we received all data on this picture, and have no use for header reconstruction.

## 5.6 New Elements

To make the simplified pipelines for the first scenario, i.e. without the quality feedback loop, we still need a couple of elements which are not in GStreamer: RTP wrapper, RTP unwrapper and Adder.

GStreamer has an RTP plugin, which has a pair of elements for wrapping and unwrapping raw audio in RTP packets. For doing the same for MPEG, this plugin was extended with two elements that correspond to the RTP Wrapper and RTP Unwrapper from the architectural model: `rtpmpegenc` and `rtpmpegparse`.

The add element was written in a new plugin using a plugin template provided by the GStreamer team.

### 5.6.1 `rtpmpegenc`

This element works closely with the `rfc2250enc` element described earlier. It takes the output structures to create RTP packets out of them. The timestamp is copied into the RTP timestamp header, and the rest of the structure is copied as is into the RTP payload (including the MPEG specific header information, since they are already in the correct format).

Furthermore, the other RTP fields have to be filled in. These are, among others, the RTP sequence number and the content type. The RTP plugin already provides these functions, since they are the same as for wrapping raw audio, and can be reused.

The output is a sequence of RTP packets.

## 5.6.2 rtpmpegpars

The `rtpmpegpars` element takes a sequence of RTP packets as its input and outputs the MPEG fragments from the RTP payload.

To make sure that an MPEG decoder does not output garbled pictures because of the loss of RTP packets, this element is designed to only output MPEG fragments that make up the data for a complete picture. To do this, it collects all RTP packets that carry the data for one picture. It detects gaps in the RTP sequence numbers, and each time such a gap is detected, it discards the queued up data for the current picture, and skips all incoming RTP packets until the beginning of a new picture is detected. This information is acquired from the begin of picture marker in the extended MPEG specific RTP headers.

When an end of picture marker is detected, the queued up MPEG data for the current picture is output as one buffer.

## 5.6.3 add

The `add` element, in the plugin with the same name, is an implementation of the Adder component described in the Architecture chapter (see [Section 4.5.4.1](#)). It has two input pads `sink1` and `sink2` and one source pad `src`, that correspond to the `i1`, `i2` and `o` ports of the model. The input and output buffers are all complete pictures in the YUV 4:2:0 format.

As this element has two inputs, the suggestion from the GStreamer documentation was taken to implement this element using a loop function (as opposed to a chain function). Unfortunately, this did not work as expected.

The first try was to implement the element without handling data loss. The loop function would pull an input from both sink pads (in sequence) and add the two buffers to create a new picture which was then output. Assuming no data loss, matching pictures would arrive at the element in the same order: the 5th picture from one layer, would also be the 5th input for that pad, and the same for the other layer. It was assumed the loop function of one instance of this element would not be called more than once in parallel, and that the input/output behaviour would be like this:

$$(i1?; i2?; o!)*$$

However, this proved not to be the case. Experiments showed that the loop function was called more than once in parallel and that pictures with different timestamps were merged, resulting in an unpredictable, and unusable, output sequence. At this time, the GStreamer documentation on the implementation of elements with multiple inputs was close to non-existing, so it could not be determined if this behaviour was to be expected.

## NOTE



After the implementation phase was stopped, the GStreamer team updated its documentation. More explanation of how the scheduling in GStreamer works was given, as described in [Section 5.3.2](#). According to this documentation, as expected, the loop function of an instance of an element should only be called after a previous call to this function has ended. Also, when a context switch is made from within this function to another element, and the current element is switched back in, the function should be resumed at the same point as where the first context switch occurred.

One of the comments in the documentation was that the default, optimal, scheduler cannot handle elements with multiple inputs well. Combined with the documentation about how the behaviour should be, this remark explains the observations of unexpected I/O behavior. The basic scheduler should not exhibit these problems, but this was not tested with the loop-based implementation.

The second try was to reimplement the element using a chain-based function that stored the input from the one sink pad until the input of the other sink pad was also received, and then merging the two into a new output picture.

The chain-based approach did also not quite work as expected, and the cause was thought to be in the scheduler, so the basic scheduler was tried. As explained in the note above, this hypothesis turned out to be correct: the basic scheduler did give the expected results with the chain-based approach.

Subsequently, the implementation was expanded to use queues for both sink pads, as described in the architectural model of the Adder component. The element uses timestamps to match up the pictures, and these timestamps are transported in the buffers' meta data timestamp field. This requires that the incoming pictures have this timestamp set correctly, and this is done by the MPEG decoder provided with GStreamer. The decoder calculates the display timestamp of each picture using the GOP time information and the picture's temporal reference information that are both stored in the MPEG stream.

Using the basic scheduler, this element did not show the anomalies we encountered with the loop-based version, or when using the optimal scheduler.

## 5.7 Building the pipeline

Now we have all components needed to build the pipelines for transporting and decoding two layers of scalable video without a feedback loop, as described in [Section 4.5.4](#) using the flow diagram in [Figure 4.5](#). There is one pipeline running in a GStreamer instance on the sending machine, pictured in [Figure 5.1](#), and one pipeline running on the receiving machine, pictured in [Figure 5.2](#).

Running this pipeline uncovered a problem: the UDP traffic flowing between the two GStreamer instances was bursty. Close examination of the UDP packets learned that all RTP packets were sent in groups, and these groups represented exactly three MPEG pictures. The hypothesis was that GStreamer uses the buffer timestamps to schedule elements. Again lacking documentation in this area, the cause had to be determined by reading the GStreamer source.

It turns out that the `udpsink` uses the buffer timestamps to regulate the traffic by using a *GStream Clock*. An element can define a wake-up time for itself, and then waits for this time to

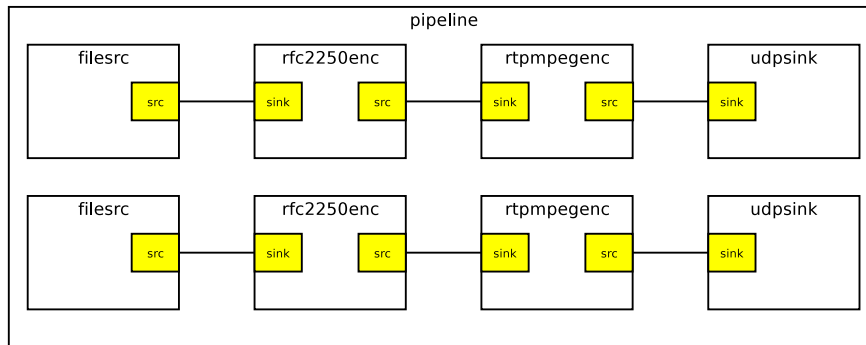


Figure 5.1: GStreamer pipeline for sending two layers of scalable video

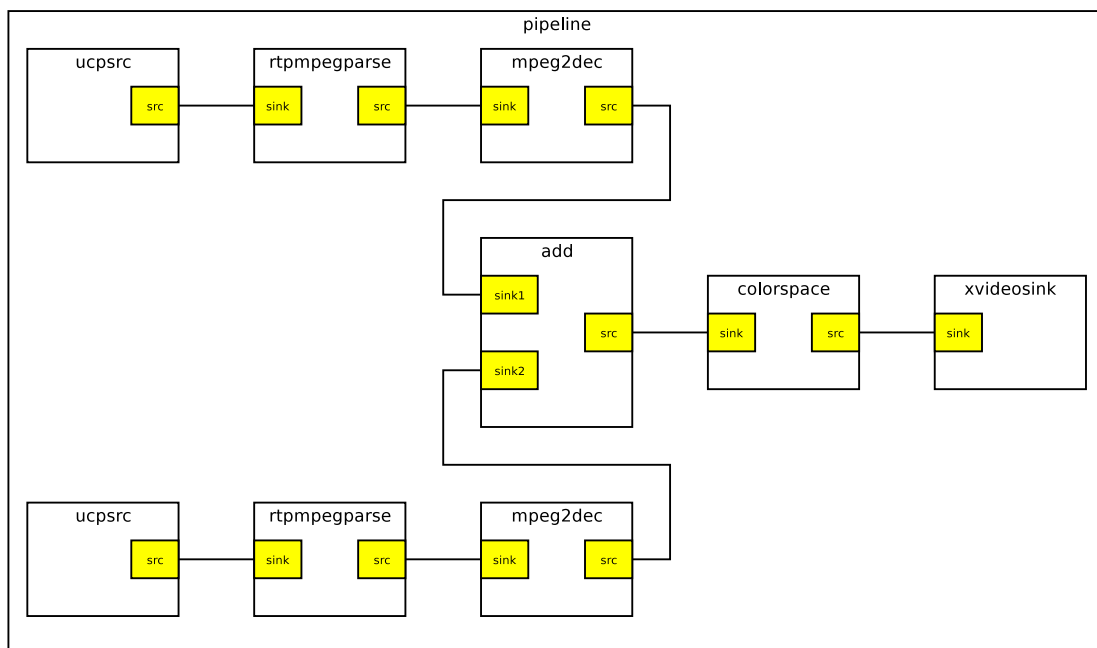


Figure 5.2: GStreamer pipeline for receiving and displaying two layers of scalable video

arrive by calling the function `gst_element_clock_wait()`. This function is a context switching point for the scheduler, and when the chosen time has arrived, the function returns. An element can now use the buffer timestamp as its target wake-up time. For example, when two buffers containing RTP packets, received by the `udpsink` element, with the second buffer timestamp 20ms later than the timestamp of the first buffer, the second packet is sent 20ms later than the first.

Having used the `rtpl16enc` element (which is for transporting raw audio over RTP) as starting point for implementing this element, the `rtmpegenc` uses the same buffer timestamp for outgoing buffers as the RTP timestamp. This buffer timestamp is generated by the `rfc2250enc` and copied verbatim.

However, since in MPEG streams the pictures are not in display order (see [Section 3.1.2](#)), the RTP timestamps are not monotonically increasing for subsequent packets. For example, using a framerate of 25 pictures per second and an GOP structure (in transport order) of I-B-B-P-B-B-P-B-B, the corresponding timestamps of the pictures in milliseconds are 80, 0, 40, 200, 120, 160, 320, 240, 280.

The way the clocks are used (looking at the source), the `udpsink` element first receives buffers with timestamp 80. Since these are the first buffers the element receives, it uses this timestamp as reference point, and the buffers are sent out as UDP packets immediately. Then buffers with timestamp 0 come in. This is *before* the timestamp of the current reference clock, so these buffers are also sent out immediately since they appear to be late. The same happens for the buffers with timestamp 40. The next batch of buffers have timestamp 20, which is still a bit away in the future, and the element waits until that time is reached and starts sending out packets again.

This results in unnecessary high bursts in the traffic. For MPEG streams with a higher bitrate this means that the bursts overflow the UDP receivers' buffers if they are not emptied quick enough (especially with slower machines), or even causes packets to be dropped before they arrive at the receiving party.

So the hypothesis proved to be wrong. GStreamer does not by default use the buffer timestamps to schedule elements when transferring a buffer through a pipe, but elements can themselves suspend their execution until a certain time is reached, in this case the time in buffer timestamps.

To counter this behaviour, the `rfc2250enc` component was modified to generate a fake buffer timestamp, representing the target time for sending out the RTP packet. This time is calculated from the GOP time information in the MPEG stream and the position of the picture in the current GOP (in transport order). This solves the problem.

# Chapter 6

## Findings

### 6.1 Handling MPEG

We choose to use MPEG as the encoding for transporting scalable video. This means we have to encode the original sequence of pictures using an MPEG encoder. And this the same number of times as the number of required layers.

MPEG's coding is designed so that decoding is computationally much cheaper than encoding. However, encoding an MPEG stream is computationally so expensive, and we quickly discovered that it is unfeasible to produce scalable video streams encoded in MPEG in real-time on a recent PC (2 GHz Pentium IV CPU) without specialised hardware encoders, even for small resolutions like QCIF (176x144).

That decoding MPEG streams is less expensive, does not mean it is an easy task. We found that realtime decoding two layers of MPEG in SD resolution (720x576) can be done on a 2 GHz Pentium IV, without special hardware. However, because we wanted low latency and do not buffer incoming packets, decoding three layers introduces delays in the processing pipeline, such that the Operating Systems' UDP buffers overflow because they are not emptied fast enough. This in turn leads to data loss.

### 6.2 Using GStreamer

GStreamer proves to be a well thought out platform to build media processing applications on. In the desktop applications area, especially for GNOME, more and more media applications are based on GStreamer. Examples include Rhythmbox, a music player modeled after iTunes, and a plugin for Nautilus (GNOME's file manager) to create thumbnails for video files.

However, GStreamer is still very much in flux. Our work has been based on a snapshot of the 0.7 development branch from March 2003, and recently the GStreamer team released the 0.7.4 developers release. Between those two versions, there have been API changes in the codebase, especially concerning video processing. The stable branch is now at 0.6.5, and has a number of API differences with the 0.7 development branch.

During the implementation phase of the project, the documentation that was available was of good quality, though incomplete. The relatively easier tasks of programming an application using GStreamer and writing simple (one-to-one) filters are well documented. However, for this project we needed to implement more difficult filters and also have to know more about how scheduling and clocks work. The current documentation has more information about this, but is still lacking (e.g. in the area of how clocks work).

Having the source code of both the GStreamer system as the plugins available helped a bit to overcome this, but this takes time. Another possibility is communicating with the GStreamer developers. However, as with most Open Source development communities, it is expected that when you ask for support or even code changes, you give them something in return (in terms of bug reports, fixes, documentation, implementation ideas or even code). This proved to be hard to do from within a corporate setting. After the implementation phase of Volund it became known that bug fixes may be submitted to Open Source projects as long as they do not supply added functionality.

## 6.3 Using RFC 2250

Although RFC 2250 is a good protocol for transporting MPEG video over RTP, it has a number of details in it that are not useful for transporting scalable video in MPEG encoding. RFC 2250 suggests ways to reconstruct lost header information and assumes a slice as the unit of recovery. For scalable video, the most logical unit of recovery is a picture, and reconstructing headers offers no value for lost video data in our application. Also, the MPEG 2 specific information that can be placed in the MPEG specific RTP headers is not needed.

Because of the assumption of a slice as the unit of recovery, there is no simple way to detect the beginning and end of a picture. Without extending the protocol with markers for those (as we have done in [Section 4.5.4.2](#), the receiver would discard more data than necessary before being able to continue processing after detecting data loss.

## 6.4 Reusability

One of the assumptions of using scalable video is that the base layer has a good probability of getting transported to the receiving party. A way to insure this is prioritising the different layers and give the base layer a higher priority. This should be done at lower layers in the protocol stack (like the physical link layer), and was therefore outside the scope of this project. However, the GStreamer elements coded for this project should be usable in experiments in this area with little to no modification.

# Chapter 7

## Conclusions

**Pure RFC 2250 is not optimised for scalable video** Using the proposed extensions to RFC 2250, adding the begin and end of picture markers, will enable scalable video to be processed more efficiently.

**The Pipes and Filter pattern is an adequate pattern for building Volund.** Since Volund is all about a series of well-defined operations on data streams, the Pipes and Filters architectural pattern is good choice. It proved possible to define the operations as filter components and exploiting the advantages of this pattern in terms of reusability and recombination.

**Conceptual model fits GStreamer.** All components described in the architectural model using the Pipes and Filters pattern, proved to either have ready available counterparts in GStreamer or be relatively easy to implement as new GStreamer elements.

**Using GStreamer cuts down on development time and increases flexibility.** GStreamer is developed as a generic media processing framework, and is therefore not particularly tied to a certain application (as opposed to custom code for just the given scenarios). This shows in the great number of ready available filters like decoders, encoders, de/multiplexers, transformations using a large number of media types. Also, having such extensive debugging facilities helps development of new elements. Without this genericity, it would be less likely that future research prototypes are implementable in Volund. However, good documentation, or a good understanding of the internals is necessary for rapid development, and fortunately, the GStreamer documentation has been improving, although slowly.

**Handling scalable video without dedicated hardware is troublesome.** As mentioned in the Findings chapter, encoding and decoding is computationally expensive, and the use of special hardware encoders and decoders would be a great advantage. Commercial, off the shelf hardware, in the form of enhanced graphics cards with on-board MPEG 2 encoders and decoders should be usable for this by wrapping their software interface in GStreamer elements.

**The other two scenarios are implementable in Volund** As already described in the Architecture chapter, the scenario of using an adapted TCP protocol should be implementable by replacing the UDP sink and source elements by TCP counterparts. The other scenario, that tries to take advantage of only decoding MPEG when the result is usable, is also implementable

in GStreamer, possibly by inserting special elements in the pipeline that selectively filter out data, or by using signalling between components using GStreamer events. Another approach would be to implement a special scheduler with the desired behaviour, but this will require more research (or documentation) about the internal workings of schedulers in GStreamer. The adder component, provided the addition is still possible on a frame-to-frame basis, can be used unmodified.

**GStreamer documentation is (still) lacking.** Although recently the GStreamer documentation has been updated to shed more light on the more advanced topics in development of GStreamer plugins, it is still troublesome. However, building simple filters and applications that use already written filters is well documented and easy to do.

**GStreamer is well suited to base Volund on.** Taking the above in account, and having shown that scenario 1 (without feedback loop) has been successfully implemented in GStreamer, and has proven to be also usable for other projects (see [Section 1.2](#)), GStreamer has been a good choice for building Volund. It enables the researcher to focus on the tasks at hand by using the existing framework and available plugins.

## Chapter 8

# Future work

**Enhance current solution to handle missing reference pictures.** The solution presented here does not take missing MPEG reference pictures into account. For example, if an I frame has been damaged and discarded, all following pictures (B and P) that depend on it, should also be discarded. Otherwise, the output of the encoder gives faulty pictures. This could be done by detecting which pictures were lost using the RTP and MPEG specific header information as described in RFC 2250 and developing an algorithm for dealing with the loss.

**Add signalling to the pipeline.** When the RTP Unwrapper knows what data will not be sent to the decoder, it could send GStreamer events upstream to the adder component, so that it can use this information instead of having to wait until the next picture arrives. This minimises buffering and lowers latency.

**Add feedback loop to scenario 1.** The stream monitor and filter control component would need to be implemented. The stream blocker is present as the `identity` element. It provides a property that defines the amount of traffic (in percentages) to be allowed to go through. This property can be altered while a GStreamer pipeline is running.

**Port implemented elements to current development version of GStreamer.** Since a number of API and other changes have occurred in the development branch since the version our implementation is based on, it would be good to port these components to work with the most recent development version. Future stable releases will be based on this version, and it would be nice to be able to use for example elements wrapping hardware decoders, which are currently in development.

**Allow session initiation using UPnP** Currently, all addressing information has to be configured up front in the sender and receiver pipelines. Being able to use UPnP to do this dynamically would be a nice way to use and experiment with the UPnP work inside Philips Research.



# Bibliography

- [POSA1] *Pattern-Oriented Software Architecture*, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, 1996, Wiley, ISBN: 0 471 95869 7. [4.3](#)
- [RFC1889] *RFC 1889: RTP: A Transport Protocol for Real-Time Applications*, H. Schulzrinne, Internet Engineering Task Force. [3.2](#)
- [RFC1890] *RTP Profile for Audio and Video Conferences with Minimal Control*, H. Schulzrinne, Internet Engineering Task Force. [3.2](#)
- [RFC2250] *RFC 2250: RTP Payload Format for MPEG1/MPEG2 Video*, D. Hoffman, Internet Engineering Task Force. [3.3](#)
- [VD3] *Video Demystified*, Keith Jack, 2001, LLH Publications, ISBN: 1 878707 56 6. [3.1](#)