Video Streams in a Computing Grid

STEFHAN KOEHLER

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:139
VIDEO STREAMS IN A COMPUTING GRID
Master of Science Thesis

Author:
Stephan Koehler

Examiner:
Prof. V. Vlassov
KTH, Stockholm, Sweden

Supervisor:
J. Remmerswaal
Tricode Business Integrators BV,
Veenendaal, the Netherlands

Royal Institute of Technology
Kungliga Tekniska Högskolan
Stockholm, Sweden

Version: 1.0 final
Date: 2010-06-09
Abstract

The growth of online video services such as YouTube enabled a new broadcasting medium for video. Similarly, consumer television is moving from analog to digital distribution of video content. Being able to manipulate the video stream by integrating a video or image overlay while streaming could enable a personalized video stream for each viewer. This master thesis explores the digital video domain to understand how streaming video can be efficiently modified, and designs and implements a prototype system for distributed video modification and streaming.

This thesis starts by examining standards and protocols related to video coding, formats and network distribution. To support multiple concurrent video streams to users, a distributed data and compute grid is used to create a scalable system for video streaming. Several (commercial) products are examined to find that GigaSpaces provides the optimal features for implementing the prototype. Furthermore third party libraries like libavcodec by FFmpeg and JBoss Netty are selected for respectively video coding and network streaming. The prototype design is then formulated including the design choices, the functionality in terms of user stories, the components that will make up the system and the flow of events in the system. Finally, the implementation is described followed by an evaluation of the fault tolerance, throughput, scalability and configuration. The evaluation shows that the prototype is fault tolerant and its throughput scales both vertically and horizontally.

Intended audience

This thesis focuses on topics in the area of general computer science and network technology. It is therefore assumed that the reader has knowledge of basic concepts and techniques in these areas. More specifically this report focuses on topics related to digital video and distributed computer systems. Knowledge in these areas is helpful but not required.
## Contents

1 Introduction

1.1 Background .................................................. 1
1.2 Problem statement ........................................... 1
   1.2.1 Functional requirements ............................... 2
   1.2.2 Non-functional requirements ......................... 2
   1.2.3 Roles .................................................. 2
1.3 Approach .................................................... 2
1.4 Related work ............................................... 3
1.5 Thesis outline .............................................. 3

2 Media protocols and standards ................................ 5

2.1 Video coding standards ...................................... 5
   2.1.1 MPEG-2 .................................................. 5
   2.1.2 MPEG-4 .................................................. 5
   2.1.3 H.263 .................................................. 6
   2.1.4 H.264 .................................................. 6
   2.1.5 Windows Media Video .................................. 6
   2.1.6 Open standards ........................................ 7
   2.1.7 Performance ............................................ 7

2.2 Video container standards ................................... 8

2.3 Network streaming protocols ................................ 9
   2.3.1 Properties .............................................. 9
   2.3.2 Challenges ............................................ 9
   2.3.3 Protocols ............................................. 10

2.4 Protocol stack .............................................. 13

3 Distributed middleware platforms ................................ 15

3.1 Peer-to-Peer platforms ...................................... 15

3.2 Cloud and Grid platforms .................................. 15
   3.2.1 JBOSS Infinispan ...................................... 16
   3.2.2 GridGain .............................................. 17
   3.2.3 Oracle Coherence ..................................... 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.4 GigaSpaces</td>
<td>18</td>
</tr>
<tr>
<td>3.2.5 IBM WebSphere eXtreme Scale</td>
<td>18</td>
</tr>
<tr>
<td>3.2.6 Terracotta</td>
<td>19</td>
</tr>
<tr>
<td>3.2.7 Gemstone GemFire</td>
<td>19</td>
</tr>
<tr>
<td>3.2.8 Hadoop</td>
<td>20</td>
</tr>
<tr>
<td>3.2.9 Globus Toolkit</td>
<td>20</td>
</tr>
<tr>
<td>3.2.10 Comparison</td>
<td>21</td>
</tr>
<tr>
<td>4 Selection &amp; design</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Design goals</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Supported media and protocols</td>
<td>23</td>
</tr>
<tr>
<td>4.3 Third party components</td>
<td>24</td>
</tr>
<tr>
<td>4.3.1 Media codec libraries</td>
<td>24</td>
</tr>
<tr>
<td>4.3.2 Media coding hardware</td>
<td>24</td>
</tr>
<tr>
<td>4.3.3 Network protocol libraries</td>
<td>24</td>
</tr>
<tr>
<td>4.3.4 Media frameworks</td>
<td>25</td>
</tr>
<tr>
<td>4.4 Distributed middleware</td>
<td>26</td>
</tr>
<tr>
<td>4.4.1 GigaSpaces overview</td>
<td>26</td>
</tr>
<tr>
<td>4.5 Prototype design</td>
<td>28</td>
</tr>
<tr>
<td>4.5.1 User stories</td>
<td>28</td>
</tr>
<tr>
<td>4.5.2 Components</td>
<td>29</td>
</tr>
<tr>
<td>4.5.3 Events</td>
<td>33</td>
</tr>
<tr>
<td>4.5.4 Deployment</td>
<td>39</td>
</tr>
<tr>
<td>5 Result</td>
<td>41</td>
</tr>
<tr>
<td>5.1 Process</td>
<td>41</td>
</tr>
<tr>
<td>5.2 Prototype</td>
<td>42</td>
</tr>
<tr>
<td>5.2.1 Component overview</td>
<td>43</td>
</tr>
<tr>
<td>5.2.2 ContentLoader</td>
<td>44</td>
</tr>
<tr>
<td>5.2.3 Filters</td>
<td>45</td>
</tr>
<tr>
<td>5.2.4 Director</td>
<td>45</td>
</tr>
<tr>
<td>5.2.5 FilterService</td>
<td>46</td>
</tr>
<tr>
<td>5.2.6 HTTP Server</td>
<td>46</td>
</tr>
<tr>
<td>5.2.7 Web module</td>
<td>47</td>
</tr>
<tr>
<td>5.2.8 Statistics</td>
<td>47</td>
</tr>
<tr>
<td>5.2.9 Summary</td>
<td>48</td>
</tr>
<tr>
<td>5.2.10 User stories: sequence &amp; deployment</td>
<td>49</td>
</tr>
<tr>
<td>6 Evaluation</td>
<td>53</td>
</tr>
<tr>
<td>6.1 Setup</td>
<td>53</td>
</tr>
<tr>
<td>6.1.1 Grid</td>
<td>53</td>
</tr>
<tr>
<td>6.1.2 Content</td>
<td>54</td>
</tr>
</tbody>
</table>
Contents

6.1.3 Test execution ........................................... 54
6.2 Functionality ............................................. 55
6.3 Fault tolerance ............................................. 55
  6.3.1 Failing of a partition ................................. 55
  6.3.2 Failing of the web module ......................... 55
6.4 Maximum concurrent requests ......................... 56
6.5 Scalability ................................................ 58
  6.5.1 Vertical scalability ................................. 58
  6.5.2 Horizontal scalability ............................ 60
6.6 StreamPackets ........................................... 61
6.7 Summary .................................................. 63

7 Conclusion ................................................. 65
  7.1 Future work .............................................. 66

Appendix A: List of classes .............................. 73
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The streaming media protocol stack.</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>UML Component diagram of the prototype system.</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Components of the prototype system.</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Sequence diagram for user story 1.</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Sequence diagram for user story 2. Here illustrated using HTTP.</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Sequence diagram for user story 3. Here illustrated using HTTP.</td>
<td>35</td>
</tr>
<tr>
<td>4.6</td>
<td>Sequence diagram for user story 4. Here illustrated using HTTP.</td>
<td>36</td>
</tr>
<tr>
<td>4.7</td>
<td>Sequence diagram for user story 5.</td>
<td>37</td>
</tr>
<tr>
<td>4.8</td>
<td>Sequence diagram for user story 6.</td>
<td>37</td>
</tr>
<tr>
<td>4.9</td>
<td>Sequence diagram for user story 7.</td>
<td>38</td>
</tr>
<tr>
<td>4.10</td>
<td>Sequence diagram for user story 8.</td>
<td>38</td>
</tr>
<tr>
<td>4.11</td>
<td>UML Deployment diagram for the prototype system.</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>UML Component diagram of the prototype system as implemented.</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Components of the prototype system as implemented.</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>ContentLoader writing multiple StreamPackets upon request.</td>
<td>44</td>
</tr>
<tr>
<td>5.4</td>
<td>A SessionStreamHandler checks each StreamPacket against the Filter (left). It then either forwards them to the HTTP server or issues modification by creating a FilterRequest and a FilterPacketRequest (right).</td>
<td>45</td>
</tr>
<tr>
<td>5.5</td>
<td>Two FilterService Instances each modifying a StreamPacket.</td>
<td>46</td>
</tr>
<tr>
<td>5.6</td>
<td>An HTTP Server taking StreamPackets from the space and sending the data over HTTP to the user.</td>
<td>46</td>
</tr>
<tr>
<td>5.7</td>
<td>The Map/Reduce approach to collecting statistical events.</td>
<td>47</td>
</tr>
<tr>
<td>5.8</td>
<td>Sequence diagram for user story one as implemented.</td>
<td>49</td>
</tr>
<tr>
<td>5.9</td>
<td>Sequence diagram for user story two as implemented.</td>
<td>50</td>
</tr>
<tr>
<td>5.10</td>
<td>Sequence diagram for user story three as implemented.</td>
<td>51</td>
</tr>
<tr>
<td>5.11</td>
<td>Sequence diagram for user story four as implemented.</td>
<td>51</td>
</tr>
<tr>
<td>5.12</td>
<td>Sequence diagram for user story six as implemented.</td>
<td>52</td>
</tr>
<tr>
<td>5.13</td>
<td>The updated UML Deployment diagram.</td>
<td>52</td>
</tr>
<tr>
<td>6.1</td>
<td>Mean (●) and standard deviation (⊥) CPU utilization (left) and memory usage (right) during maximum concurrent request evaluation.</td>
<td>57</td>
</tr>
</tbody>
</table>
6.2 Vertical scalability of the prototype system. ................................. 59
6.3 Mean ($\bullet$) and standard deviation ($\perp$) CPU utilization (left) and memory usage (right) during vertical scalability evaluation. ................................. 59
6.4 Horizontal scalability of the prototype system. .............................. 60
6.5 Throughput (left) and modification rate (right) when varying the number of keyframes in a StreamPacket. .................................................. 61
6.6 Mean ($\bullet$) and standard deviation ($\perp$) CPU utilization (left) and memory usage (right) when varying the number of keyframes in a StreamPacket. ................. 63
List of Tables

2.1 Common container formats for multimedia (adopted from [68]). ... 8

3.1 Comparison table or GRID middleware products. ... 21

6.1 The Amazon EC2 instance types used in the evaluation. ... 53
6.2 The video streams used in the evaluation. ... 54
6.3 Results of the maximum concurrent request evaluation. ... 56
6.4 Results of the vertical scalability evaluation. ... 58
6.5 Results of the horizontal scalability evaluation. ... 60
6.6 Results of the StreamPacket evaluation. ... 61
6.7 Profile of the modification process. All values display the mean of the measurements. ... 62
List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamically Linked Library</td>
</tr>
<tr>
<td>DVB</td>
<td>Digital Video Broadcasting</td>
</tr>
<tr>
<td>FEC</td>
<td>Forward Error Correction</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose computations on a Graphical Processing Unit</td>
</tr>
<tr>
<td>GSC</td>
<td>Grid Service Container</td>
</tr>
<tr>
<td>GSM</td>
<td>Grid Service Manager</td>
</tr>
<tr>
<td>H.263</td>
<td>Video coding standard developed by ITU</td>
</tr>
<tr>
<td>H.264</td>
<td>Video coding standard developed by ITU and MPEG</td>
</tr>
<tr>
<td>HD</td>
<td>High Definition</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>ITU-T</td>
<td>Telecommunication Standardization Sector, part of the International Telecommunication Union</td>
</tr>
<tr>
<td>JMF</td>
<td>Java Media Framework</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Picture Expert Group</td>
</tr>
<tr>
<td>MPEG-2</td>
<td>Video coding standard by MPEG</td>
</tr>
<tr>
<td>MPEG-4</td>
<td>Video coding standard by MPEG</td>
</tr>
<tr>
<td>NIO</td>
<td>Java New I/O package</td>
</tr>
<tr>
<td>PU</td>
<td>GigaSpaces Processing Unit</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-Time Transport Protocol</td>
</tr>
<tr>
<td>RTCP</td>
<td>RTP Control Protocol</td>
</tr>
<tr>
<td>RTMP</td>
<td>Real-Time Messaging Protocol</td>
</tr>
<tr>
<td>RTSP</td>
<td>Real-Time Streaming Protocol</td>
</tr>
<tr>
<td>SIP</td>
<td>Session Initiation Protocol</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreements</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td>VC-1</td>
<td>Video coding standard developed by Microsoft</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VoD</td>
<td>Video-on-Demand</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>WMV</td>
<td>Windows Media Video</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter presents a brief introduction to the subject of the thesis. First, the background and motivation for this project is given in section 1.1, followed by the problem statement in section 1.2 and an approach to the problem in section 1.3. Then, the related work on the subject is explored in section 1.4 and finally the outline of this document is given in section 1.5.

1.1 Background

Streaming video services continue to grow in number and users. The success of YouTube and the transition from analog to digital video broadcasting indicate that digital streaming of media will have a prominent role in media distribution. The ability to manipulate video streams could enable users to see a personalized version of the media stream. Manipulating video streams involves the aspects of video coding, (real-time) network distribution, and high-performance (distributed) computing.

This thesis will look into these aspects to gain a better understanding of the concepts of video streaming and distributed computing in order to develop a prototype system that is able to manipulate video streams on demand.

1.2 Problem statement

The problem for this thesis is the following:

_How can individual video streams be enhanced with an integrated video or image overlay at run-time using distributed middleware?_

The goal of the thesis is to study the areas of video streaming and distributed middleware platforms, in order to design and implement prototype software running on middleware that can modify video streams on demand. The prototype can be extended to forward the modified streams to a client and/or modify streams by inserting specific content per client.
1.2.1 Functional requirements

The prototype has the following functional requirements:

- Modify an incoming video stream, inserting a second video or image on top of the video content.
- The prototype implements a streaming server that can stream the modified content to the user.
- The prototype can modify the video differently for each client.
- The prototype has to be implemented on a distributed, scalable middleware platform.
- The prototype has to be implemented as software.
- The prototype has to detect faulty nodes and be able to restart or relocate components automatically (fault-tolerance).

1.2.2 Non-functional requirements

The prototype has the following non-functional requirements:

- Modifying a video stream has to happen on a best-effort basis: The prototype should have an upper bound to the number of active streams to ensure sufficient throughput.
- Performance should scale efficiently when adding nodes (scalability).

1.2.3 Roles

The prototype should implement two roles. The user role will be allowed viewing of the modified streams. The administrator role is an extension of the user role with the addition of the ability to add content to the prototype, define modification specifications and view statistics of the prototype.

1.3 Approach

The thesis work is divided in three phases:

1. Research / study phase.
2. Design phase.
3. Implementation phase.

During the research phase the following topics will be investigated:

- The different methods of manipulating video content.
- The different streaming protocols.
- The different video compression algorithms.
- The different distributed middleware platforms.

During this phase, all related work in these areas will be studied to give a solid foundation of knowledge required to design a prototype.
In the design phase design choices will be made based on the research phase. The middleware platform, protocols and external libraries are selected, using the results from the research phase. Then, a design for a prototype will be constructed and modeled.

The implementation phase will focus on implementing the prototype. Evaluating (parts of) the implementation can also take place during this phase.

### 1.4 Related work

A lot of work has been done in the field of distributed media streaming. Most efforts however focus on a Peer-to-Peer overlay network to deliver media. This is also referred to as peercasting. Zhang et al. use a data driven overlay to create a streaming application [87]. Nguyen et al. created a framework that utilized multiple senders to achieve high throughput and resilience [37, 38]. Baccichet et al. utilize Peer-to-Peer multicast to achieve low delay video delivery [4]. Many more Peer-to-Peer (research) systems and products are available, some are also mentioned in section 3.1.

Companies such as YouTube [86] and Innovid [27] modify the video content on the client side, enhancing the video player to use an overlay to display additional content over the video.

Older related work on distributed video streaming by Mungee et al. [36] and Schantz et al. [46] focus on utilizing CORBA based middleware to deliver streaming media with Quality of Service control.

Engelhardtse et al. look at JavaSpaces to create adaptive distributed systems [11] and Batheja et al. use JavaSpaces for a framework that targets coarse-grained parallelism [5]. These works inspired on how to use JavaSpaces in this project.

This thesis investigates video overlay and modification from a server side perspective using a distributed data and compute grid, opposed to the Peer-to-Peer and client side approaches described above. The on-demand and per user modification requires significant performance demands, such that scalability should be an important property of the result. This creates a rather unique and challenging research topic.

### 1.5 Thesis outline

The rest of this document is organized as follows: Chapter 2 describes the current standards and protocols in video content delivery. Chapter 3 covers the grid middleware platforms currently on the market. Chapter 4 covers the selection of third party elements, the design choices made and a description of the design of the prototype. Chapter 5 describes the prototype system as implemented, the implementation process and any problems encountered. Chapter 6 contains an evaluation of the prototype. Finally, the thesis work is concluded in Chapter 7.
Chapter 2

Media protocols and standards

Media protocols and standards are used to represent, transport or otherwise handle media. They are used widely in for example online video services, digital video broadcasting or portable media such as DVD or Blu-ray Disc. This chapter will investigate all aspects of video streaming to provide the knowledge required to work with video streaming.

2.1 Video coding standards

Video coding standards are used to compress video data, reducing the bitrate (the amount of bits per second of video). They also ensure interoperability across different platforms. Video coding is done using codecs. Codec stands for enCOder and DECoder. An encoder takes raw video data and runs a compression algorithm to reduce the size and produce a bitstream according the standard. The decoder takes the compressed stream and runs a decompression algorithm to restore the video content. The video coding standards only specify the bitstream syntax and the process of decoding this stream, providing no constraints on implementation details. The most common and recent standards are discussed in this section.

2.1.1 MPEG-2

MPEG-2 is a coding standard developed by the Moving Picture Experts Group (MPEG) in 1996 [75]. MPEG 2 is used as a mandatory codec for DVD and Blu-ray Discs [67], and is the standard for digital video broadcasting [70].

2.1.2 MPEG-4

MPEG-4 (or MPEG-4 part 2) is a coding standard developed in 1998, and includes many features of MPEG-2 [76]. It was initially developed as a codec for low bitrate video communications, but later expanded to support higher bitrates. An extension for scalability\(^1\) for MPEG-4 has also been proposed [44].

\(^1\)A scalable coding mechanism creates multiple substreams. A base stream with coarse visual quality, that can be improved by applying the other substreams. This is useful for adapting the video to varying bitrates.
2.1.3 H.263

H.263 is a coding standard developed by the International Telecommunication Union (ITU-T) and was introduced in 1996. It is based on the H.261 codec, developed earlier by ITU-T [39]. Like MPEG-4, H.263 was also developed for low bitrate video communications. Two new versions were developed after the initial version, introducing extensions to the standard [72]. H.263 is used in the majority of online video services including YouTube [69].

2.1.4 H.264

MPEG and ITU-T collaborated on a new coding standard labelled H.264. It is also referred to as H.264 / AVC or MPEG-4 part 10 [73]. H.264 inherited several properties from MPEG-4 and H.263. It has several profiles (modes of operation) that result in different quality and bitrate levels. The profiles can be categorized as:

- Baseline - intended for low bitrate applications.
- Main - intended for mainstream consumer applications.
- Extended - intended for streaming video.
- High - Intended to deliver high video quality.

Variations and extensions on these profiles were later introduced.

The H.264 standard distinguishes a Video Coding Layer (VCL) and a Network Abstraction Layer (NAL) [66]. The VCL contains the compressed video data, while the NAL formats the VCL and adds header information for use of different transport media.

H.264 is introduced in several areas as a standard for coding High-Definition (HD) video. Online video services that support HD video use H.264 [69], and H.264 is selected as the standard for broadcasting HD video to televisions [70]. It is also one of the three codecs that must be supported by Blu-ray Disc devices [67].

2.1.5 Windows Media Video

Windows Media Video (WMV) is a set of audio and video coding algorithms developed by Microsoft in 2003 [81]. Windows Media Video was later standardized as VC-1 in 2006 [80]. Like H.264, VC-1 has profiles that vary the quality and bitrate of the video content [34]:

- Simple - 96 to 384 Kb/s.
- Main - 2 to 20 Mb/s.
- Advanced - 2 to 135 Mb/s.

VC-1 is one of the three mandatory standards for devices that support Blu-ray Discs [67].
2.1.6 Open standards

Several open coding standard exists that are free from license fees when used.

**Dirac**  Dirac is an open and royalty free coding standard developed by BBC Research [71]. It is developed to compete with H.264 and VC-1. Dirac isn’t in widespread use in the public domain yet.

**Theora**  Like Dirac, Theora [83] is an open standard. Theora is developed by the Xiph.org Foundation and is a transformation of the VP3 coding standard by On2.

**VP8**  The WebM project proposes a new open standard for video (streaming) in the new HTML5 specification. This includes the WebM file format (see section 2.2) and the new VP8 coding standard [43].

2.1.7 Performance

Performance comparisons show that the newest codecs outperform older ones in terms of coding gain (the number of bits/s compared to the image quality). When comparing H.264 to MPEG-2, H.264 has an average coding gain of 50% over MPEG-2 [30]. The same paper shows that H.264 has a coding gain over H.263: on average 47% for low bitrate coding, and on average 24% for high bitrate coding.

A comparison between WMV and H.264 shows that the standards perform roughly the same, and states that WMV has a lower computational complexity [53].
2.2 Video container standards

Video containers hold meta data about the media it holds. They function as a wrapper to provide a complete file. Commonly used containers are listed in table 2.1.

<table>
<thead>
<tr>
<th>Name (File extension)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced Systems For-</td>
<td>Developed by Microsoft, supports a wide range of video and audio codecs, although H.264 is not fully supported.</td>
</tr>
<tr>
<td>mat (.asf, .wma, .wmv)</td>
<td></td>
</tr>
<tr>
<td>Audio Video Interleave ( .avi)</td>
<td>Also developed by Microsoft, has been extended to support a wide range of video and audio codecs.</td>
</tr>
<tr>
<td>Flash Video (.flv,.f4v)</td>
<td>Developed by Adobe, supports H.263 (.flv) and H.264 (.f4v) primarily, along with a few audio formats. Used by most online video streaming services [69].</td>
</tr>
<tr>
<td>Matroska (.mkv,.mka)</td>
<td>Open container standard. Can contain almost all audio and video codings.</td>
</tr>
<tr>
<td>MPEG Program Stream (.ps)</td>
<td>MPEG container for containing MPEG-2 and MPEG-4 video streams. Used on DVDs.</td>
</tr>
<tr>
<td>MPEG Transport Stream (.ts)</td>
<td>MPEG container format for transporting video over unreliable communication. Supports all MPEG codec standards, including H.264. MPEG TS is used as a communication protocol for broadcasting digital television [77].</td>
</tr>
<tr>
<td>OGG (.ogg)</td>
<td>Open container standard.</td>
</tr>
<tr>
<td>Quicktime (.mov,.qt)</td>
<td>Developed by Apple, supports various popular audio and video codecs.</td>
</tr>
<tr>
<td>WebM (.webm)</td>
<td>Open container standard by Google et. al., extended from Matroska. Intended to be used with HTML5.</td>
</tr>
</tbody>
</table>

Table 2.1: Common container formats for multimedia (adopted from [68]).

A special file format is the Flash (.swf) format developed by Adobe. Flash is a scriptable, interactive format used to deliver media over the Internet [57, p. 9]. It also supports integration
of a video file or a video stream [57, p. 235]. Flash files are downloaded when the web page is accessed, and played using a Flash player component.

### 2.3 Network streaming protocols

Network streaming protocols can have certain properties to be used for transporting video. Also they must overcome certain challenges to deliver the content in an acceptable manner. In this section these properties and challenges are first discussed, followed by a study of the current video streaming protocols.

#### 2.3.1 Properties

This section briefly discusses properties of network streaming of video.

**Communication form** When transporting video, there are multiple communication forms to use. Unicast communicates from one source to one destination. This is common in video-on-demand (VoD) services, where each individual user requests specific video content at a specific time. Multicast communication sends the same content from one source to many (but not all) destinations. Broadcast however sends the same content from one source to all destinations. This is used for example when a live sport event is broadcasted.

**Communication channels** The channels from a source to destinations can have different characteristics, such as bandwidth, loss, or latency. A static channel keeps these characteristics at more or less the same level over time, but in a dynamic channel these characteristics can vary greatly over time. A network streaming protocol will need to be able to work on dynamic channels.

**Network type** Channels have an underlying network type: packet-switched or circuit-switched. Packet switched networks have a varying delay and order of packet transportation. Circuit-switched networks have a constant delay and order, but can suffer from data corruption.

**Bitrate** Video can have a constant or variable bitrate. Transport protocols have to support both types.

#### 2.3.2 Challenges

This section briefly discusses how the challenges of transporting video content can be overcome.

**Varying data rate** To overcome the dynamic network conditions, video can be transported using either TCP or UDP. TCP is a reliable protocol, using retransmission to ensure all data is received. This retransmission can greatly increase wait time, and decreases throughput. It is therefore not ideal for streaming large amounts of video data. UDP works on a best effort basis, which greatly benefits throughput. However, packets can be dropped so some form of error concealment has to be used.
Buffering on the receiver side can improve the video playback quality. Using a buffer small variations in bandwidth will not hinder playback of the video. Buffering is a widely used and inexpensive technique.

**Errors**  
Lost video data results in errors in the video content. Errors can be handled in various ways. A lost section can be retransmitted, or Forward Error Correction (FEC) can be used. FEC adds redundant packets that can be used to restore the original packets at the destination. This generates some overhead, but does not require the return channel that retransmission uses.

Two additional mechanisms are Error Resilient Coding and Error Concealment. Error Resilient Coding provides some robustness to data loss into the video stream added by the codec. Error Concealment is a receiver side technique that uses algorithms to recover lost video information from the received video data, such that the viewer is unlikely to see that there was an error.

### 2.3.3 Protocols

There are several standardized protocols used to transport video to users. These protocols can be separated into Internet streaming protocols and digital television protocols. This section gives an overview of all protocols currently in use in both domains.

#### Internet streaming protocols

Communicating video content over the Internet can be done in three approaches. A video file can be downloaded as a whole before viewing, the video can be streamed, or a combination of these approaches can be used called progressive downloading.

**Progressive downloading**  
Progressive downloading of video is commonly used in online video services such as YouTube [86], and uses HTTP to download the video data. The client player may start playback of the video file while it is being downloaded. HTTP is a request/response protocol commonly used to transport web content. HTTP transfers packets over TCP, making it a reliable protocol. The recent version of HTTP supports the transfer of large files, and the ability to request a portion of a file. This is known as a range request [32], and enables seeking in a file.

**Real-time protocols**  
Protocols used for streaming video over the Internet in real-time can be classified in three classes [82]:

1. **Network protocols** - The IP protocol (multicast and unicast) is used for basic network functions.

2. **Transport protocols** - Including UDP and TCP, and real-time protocols (RTP [47] and RTMP [58]).

3. **Session protocols** - Session management protocols such as the Session Initiation Protocol (SIP) [45] and the Real-Time Streaming Protocol (RTSP) [48].
The transport protocols Real-Time Transport Protocol (RTP) and RTP Control Protocol (RTCP) work on top of UDP and TCP. RTP is intended to transport real-time data, such as audio and video data, to two or more participants. The RTP header has fields that enable the following features [47):

- **Payload type** - Identifies the data type of the RTP packet’s payload.
- **Sequence numbering** - Used to restore the correct order of packets at the receiver and detect packet loss.
- **Time stamp** - Allows applications to synchronize media streams.
- **Synchronization Source** - An unique identifier for the source of the stream.

RTP is accompanied by the RTCP protocol that primarily monitors the quality of service of RTP in an ongoing session. The protocol tries to use not more than 5% of the total bandwidth used by RTP [82]. RTCP performs four functions [47]:

- **Feedback on data distribution quality.**
- **A canonical name and description of the source.**
- **Scaling of RTCP messages (≤ 5%) to the number of participants in the session.**
- **(Optional) Provide minimal session control information.**

RTSP is a session management protocol that can establish and control streams of audio and video [48]. It can be used to start, stop and pause the streams. Additionally, RTSP can retrieve media or add media to an existing session [82], and decide on the transport protocol to use. RTSP uses a Universal Resource Locator (URL) contained in a session description file, which can additionally contain the encoding, addresses and ports and language of the media.

SIP, like RTSP, is an application layer protocol to manage multimedia sessions [45] between one or more parties. Both are text based protocols that share similarities with HTTP. SIP supports user mobility, identifies the availability and capabilities of the user, and can transfer, modify, and terminate sessions.

The Real-Time Messaging Protocol (RTMP) [58] by Adobe is another real-time protocol for transporting media. It uses TCP for reliable communication and uses several dedicated channels for video, audio and control messages. RTMP also supports remote procedure calls. Currently MPEG layer 3 audio and Flash Video are the only supported formats of RTMP. All protocols described in this section can be combined to create a complete streaming service [82]. Internet Protocol television (IPTV) [74] uses these protocols to offer television services.
Digital television protocols

For digital television the Digital Video Broadcasting (DVB) protocols are used in Europe [70]. Similar standards exist that are used in other countries. This set contains four different protocols for different physical communication channels:

- DVB-S - Satellite reception.
- DVB-T - Terrestrial (radio frequency) reception.
- DVB-C - Cable reception.
- DVB-H - Reception on handheld devices, using terrestrial reception.

These protocols mainly differ in their way of handling physical constraints of their different communication channels.

MPEG Transport Stream DVB protocols use the MPEG Transport Stream (MPEG-TS) container to transport video data [70]. A MPEG-TS can contain multiple video streams multiplexed together allowing viewers to select video streams from one universal broadcasted MPEG-TS. Additionally, a MPEG-TS contains a Program Association Table that lists all the video streams available in the transport stream. Each program has an entry in the Program Map Table with a description of the program. Finally, at least each 100ms a Program Clock Reference is transmitted to allow the receiver to synchronize (audio with video) streams [28].

DVB uses the MPEG-TS in combination with the MPEG-2 codec for normal television broadcasts and the H.264 codec for High Definition video content [70].

---

2Each video stream is referred to as a program in the MPEG-TS specification.
2.4 Protocol stack

As an overview, figure 2.1 shows the protocol stack for streaming media.

![Diagram of the streaming media protocol stack]

**Figure 2.1:** The streaming media protocol stack.
Chapter 3

Distributed middleware platforms

Distributed middleware can be utilized as a platform for applications that want to benefit from the properties of distributed computing. Because video coding and processing requires substantial computation power, this chapter focuses on this property, listing some of the (commercial) platforms for distributed middleware in the areas of Peer-to-Peer, Cloud- and Grid computing. This evaluation of platforms can provide the knowledge required to select the most suitable platform for the prototype.

3.1 Peer-to-Peer platforms

Peer-to-Peer systems are popular for file sharing and are increasingly utilized for media streaming. Several research systems have been developed that can provide (live) streams [62, 85, 24, 42] or Video-on-Demand [84, 9], as well as commercial systems [26, 2, 59, 56, 22]. These systems use an overlay network of individual peers that can request or stream content to each other, acting as both a client and a streaming server. This approach also avoids the need for a large central storage of video data, as the data is partitioned over all the peers, as well as the need for a large server bandwidth. Peer-to-Peer systems inherently are fault-tolerant and scalable. These systems however assume static video content. In order to support dynamic modification of video content a large amount of computational power is needed. When using a Peer-to-Peer system, individual peers would become largely burdened with these heavy computations. Furthermore, communicating video data between a large set of peers requires substantial network bandwidth.

3.2 Cloud and Grid platforms

In the area of cloud- and grid computing several concepts apply, which will be defined below:

- **Computing grid**: Set of CPU’s on which tasks or jobs are distributed to gain high parallel execution throughput.
- **Data grid**: Controls a large set of distributed and/or replicated data.
- **In-memory data grid**: Data is stored in memory, possibly divided over several partitions.
• **Colocation**: Logic and data are located in the same process or Virtual Machine (VM). This reduces latency as network calls are eliminated.

• **Grid computing**: The combination of a computing and data grid.

• **Cloud computing**: The combination of a datacenter with an API. The API manages the nodes in the grid, exposing a single image to users.

• **Service Level Agreements**: Service Level Agreements (SLA) state a set of requirements that must be met for the entire grid. Some platforms use SLAs to dynamically and automatically start or stop nodes in the grid under varying load.

• **Map/Reduce** [8]: A programming model for processing or handling large data sets. A Map function is applied to a large number of individual data items in parallel, followed by the execution of a single reduce function that gathers the results from the Map operations and applies the defined reduce function on these results.

Common products/frameworks used are:

• Spring [52] - Java enterprise application development framework.

• Hibernate [25] - Persistence and query service.

### 3.2.1 JBOSS Infinispan

The JBOSS Infinispan project [13] is an open source implementation of an in-memory data grid, written in Java. It utilizes a special cache data structure that can be distributed using a peer-to-peer architecture. This provides a scalable and concurrent data-grid with three modes of operation:

• **Distributed**: The entries are distributed over the grid nodes with a configurable number of backups.

• **Replicated**: Every node in the grid has the same entries in the cache. Limits scaling of the cache.

• **Invalidation**: Every node has a local cache, where entries are invalidated after a period of time or when another node changes the entry. Nodes will look in the local cache before doing a remote lookup.

JBOSS Infinispan evolved from the JBOSS Cache project [12], which implements a distributed cache. The cache can be backed by a database in synchronous (write-through) and asynchronous mode (write-behind). On the roadmap Infinispan is scheduled to support queries on the cache and indexing, as well as more programming languages, including Python, Ruby, C and PHP. Also on the roadmap is the support of passing a Runnable object around the grid to be executed somewhere in the grid and collect the results of the computation centrally, implementing the Map/Reduce paradigm.

Being a rather new product, it lacks some functionality. There is no control of data placement (routing) in the grid, and the data grid can only be configured to support one type of transaction which is defined on the grid level. This results in very limited use of transactions, because only one transaction can work on the grid at any time.
3.2.2 GridGain

GridGain [21] is a cloud development platform. It is written in Java and fully open source. GridGain provides a computing grid, and is usable with other data grids or (distributed) caches such as JBOSS Cache (described in section 3.2.1) and Oracle Coherence (section 3.2.3). The Map/Reduce model is the core of the GridGain functionality, furthermore providing failover management for fault tolerance, and a management console for managing nodes and showing statistics.

GridGain is designed to integrate with a number of other platforms such as application servers, test-, communication-, and monitoring frameworks, listed below. The list of products below is adopted from [20]:

- JUnit - Test framework.
- JBoss - Application server.
- Glassfish - Application server.
- AspectJ - Aspect orient programming framework.
- Spring.
- Weblogic - Application server.
- Mule - Enterprise service bus provider.
- JXInsight - Monitoring solution.
- GigaSpaces - see section 3.2.4.

3.2.3 Oracle Coherence

Coherence is a product line from Oracle that provides an in-memory data grid [41]. The data grid provides replicated, distributed data management and caching, using a peer-to-peer protocol. This makes the data grid fault tolerant and linear (horizontally) scalable. Combined with an (Oracle) application server a computing grid can be constructed and monitored. Some features of the in-memory data grid are:

- Queries on data stored in the grid.
- Remote execution, executing close to where data is located.
- Supporting for indexing.
- Load balancing of data over the partitions.
- Support for transactional consistency.
- Database integration.

Remote execution is done via agents. Agents provide colocation (targeted execution), parallel execution (including Map/Reduce), querying the entire grid, or a combination of these functions. On top of that agents can do aggregations of data on the entire grid. Furthermore, listeners can be defined for event notification. Coherence has a Java, .NET and C++ edition.
3.2.4 GigaSpaces

GigaSpaces is an application server based on an in-memory Data Grid [6]. The product extends the JavaSpaces [35] API for distributed applications and is centered around this data grid, called the Space. Some important features are:

- The Space provides a distributed data storage.
- Support for several grid topologies, providing load balancing and replication.
- Data in the grid can be queried using a subset of SQL.
- Remote access to the Space.
- Database support for persistence.
- Remote execution, providing data locality.
- Full control of data placement (routing) in the grid.
- Supports Map/Reduce model.
- Support for transactions.
- Support for messaging API’s and event notification.

In combination with the data grid, the application server provides a framework to develop a compute grid that works together with the Space. The compute grid can process events, located on the Space, using Processing units. A Processing unit is an isolated piece of business logic, and this shared nothing approach\(^1\) [50] allows for linear scaling of computing units.

On top of these features, GigaSpaces has full support for Spring and Hibernate, and comes with a built in web server to process web requests. The roadmap for 2010 indicated support for REST. GigaSpaces supports Java, .Net and C++ natively. The GigaSpaces grid manager can be configured to use SLAs.

3.2.5 IBM WebSphere eXtreme Scale

WebSphere is the application server product line from IBM. WebSphere eXtreme Scale is an in-memory data grid implementation in Java that can be used with an application server [31]. Some features of the data grid are:

- Scalability, moving data partitions in the grid when nodes are added.
- Partitioning and replication of data over the grid.
- Support for relational object storages, SQL like queries.
- Distributed code execution through agent, support Map/Reduce.
- Remoting supported using application server.
- Transaction support with optimistic and pessimistic locking.
- Good support for persistence/databases with several modes of operation.
- Support for Spring.
- Support for other languages through (HTTP) REST data service.

\(^1\)In a shared nothing system, a single node does not rely on any shared data or resource, removing contention for that data or resource.
• Management and statistic supported, but no graphical interface.

WebSphere eXtreme Scale uses a so called Catalog Manager to manage the data in the partitions and balance the load on each node. Messaging is not supported in the data grid, and routing is handled by the Catalog Manager without an interface for the developer to enable specific data placement on the grid.

3.2.6 Terracotta

Terracotta [61] is a scaling solution for Java. It uses a distributed cache called EHCache [60]. EHCache is implemented in Java with an open source version, and a rich set of features:

• Can be used as a memory or disk cache.
• Replication in synchronous and asynchronous mode.
• REST and SOAP API for remote access.
• Monitoring support.
• Support for third party/open source applications like:
  – Spring.
  – Hibernate.
  – Google App Engine.
  – Tomcat web server.

Terracotta adds job scheduling on the grid, as well as web server support and web sessions. EHCache is fully open source, and Terracota has both an open source and commercial version.

3.2.7 Gemstone GemFire

Gemstone provides a in-memory data grid product called GemFire [17]. GemFire distributes data across several partitions for horizontal scalability, where each partition can have a backup instance [18]. Persistence is supported through synchronous or asynchronous writes to a database or file system. The grid supports different topologies, and supports messaging and data update notifications in a reliable asynchronous way. Messages can be queued and delivered at a later time when a node has failed and recovers. Data in the grid can be queried using the Object Query Language standard. Parallel execution of tasks on the data grid provides the Map/Reduce model, and colocation of data is supported. A unique feature is the support for the grid to cluster nodes across a Wide Area Network (WAN). GemFire supports Java, C# and C++ natively.
3.2.8 Hadoop

Hadoop is a distributed computing project by the Apache Software Foundation [65]. Hadoop provides a Map/Reduce model combined with a distributed file system. Hadoop is divided into several sub-projects:

- **Core**: Components and interfaces for distributed file systems and general I/O.
- **AVRO**: Serialization system, allows cross language procedure calls.
- **Map/Reduce**: Execution environment for the Map/Reduce model.
- **HDFS**: Hadoop Distributed Filesystem.
- **ZooKeeper**: Coordination service for distributed systems.
- **HBase**: Distributed database using HDFS.
- **Pig**: Data flow language for exploring large datasets.
- **Hive**: Distributed data warehouse, provides SQL for querying data stored in the HDFS.
- **Chuckwa**: Distributed data analysis using Map/Reduce.

These projects combined provide building block for creating a scalable and reliable distributed system where Map/Reduce applications are the main focus. Hadoop does not have an in-memory data grid, but instead uses HDFS for data storage. HDFS is designed to provide relatively fast read access to very large files, and run on unreliable commodity hardware. However, HDFS is not very suitable for low latency access and using lots of small files. Furthermore, only one writer can write to a file simultaneously. Hadoop is open source and written in Java, but provides bindings to C++, Python and Ruby, as well as the ability to integrate native libraries.

3.2.9 Globus Toolkit

Globus Toolkit is a grid platform maintained by the Globus Alliance. The toolkit is open source and used in several large scale grid infrastructures [15]. Globus Toolkit is modular with several modules for security, resource management, data discovery, etc. These modules are implemented as services, and Globus Toolkit is compliant with web services and SOAP. Execution management is done by the Grid Resource and Allocation Management (GRAM) module, and data placement and locality is done by the GridFTP module. GridFTP can provide reliable and high speed data movement over memory or disk. Replication of data is managed by the Data Replication Service (DRS), combined with the Replication Location Service (RLS). User applications run in containers that have access to the Globus modules, and can expose themselves as webservers. Globus Toolkit provides containers for Java, C and Python code.
### 3.2.10 Comparison

This section presents a comparison table of the products described in the previous sections. Most common features are listed.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data grid</th>
<th>Compute grid</th>
<th>Map/Reduce</th>
<th>Remote access</th>
<th>Web-Server</th>
<th>REST</th>
<th>SOAP</th>
<th>Event notification</th>
<th>Queries</th>
<th>Transactions</th>
<th>Persistence</th>
<th>Spring</th>
<th>Auto. Recovery</th>
<th>SLA support</th>
<th>Management UI</th>
<th>Open source</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>JBOSS Infinispan</td>
<td>✓</td>
<td>*</td>
<td>***</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java****</td>
</tr>
<tr>
<td>GridGain</td>
<td>*</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java</td>
</tr>
<tr>
<td>Oracle Coherence</td>
<td>✓</td>
<td>*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java, .NET, C++</td>
</tr>
<tr>
<td>GigaSpaces</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java, .NET, C++</td>
</tr>
<tr>
<td>IBM WebSphere</td>
<td>✓</td>
<td>*</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java****</td>
</tr>
<tr>
<td>Terracotta</td>
<td>✓</td>
<td>✓</td>
<td>*</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java****</td>
</tr>
<tr>
<td>GemFire</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java, C#, C++</td>
</tr>
<tr>
<td>Hadoop</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java, C++, Ruby, Python</td>
</tr>
<tr>
<td>Globus Toolkit</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java, C, Python</td>
</tr>
</tbody>
</table>

**Table 3.1: Comparison table or GRID middleware products.**

- * Support for third party product, or other product from the same vendor.
- ** Limited support.
- *** This feature is scheduled on the product’s roadmap.
- **** This product has an open source component.
- ***** More programming languages are scheduled to be supported.
- ****** Support for additional languages through REST interface.
Chapter 4

Selection & design

This chapter focuses on the design choices made in the design process and the initial design of the prototype system. It covers selection of the middleware platform, media coding algorithms, media containers and network protocols, along with any external libraries that may ease the development of the prototype. This chapter then presents the user stories that reflect the functionality of the system, the components that make up the system and the flow of events and action that correspond to each user story.

4.1 Design goals

The design goals for the prototype system can be derived from the (non-) functional requirements specified in section 1.2. The prototype system has to modify incoming video content, this has to happen on a best effort basis, using scalable and fault tolerant middleware. Furthermore, a set of media standards and protocols have to be supported by the prototype. The design goals should serve as guidelines during selection and design.

Video processing is a computational intensive task. Therefore, the prototype has to aim for high performance and high throughput. To achieve this in a distributed system, the design should focus on parallelism, no or little lock- or resource contention and fast access to data (data locality).

4.2 Supported media and protocols

The wide range of video audio and image formats and codecs available forces the prototype to initially support a subset of formats. Online video services commonly use Flash video [69], which is based on H.263. It is therefore useful for a video service to support at least Flash video. As described in section 2.3, there are two categories of streaming protocols: Internet and digital television. Given that this design is for a software platform, the Internet protocols are easiest to support. Therefore the prototype will initially support the HTTP protocol, and is optionally extended with the RTP/RTSP protocols if time permits. RTMP is a proprietary format and therefore has the least preference.
4.3 Third party components

In order to support the media and network protocols, the use of (open source) libraries or hardware can benefit the development of the prototype.

4.3.1 Media codec libraries

Several libraries exist that implement media codecs in software. The largest open source library available is libavcodec by the FFmpeg project [14]. Written in C, it implements a large set of codecs including MPEG-2, MPEG-4, H.263, VC-1, Dirac and H.264 (decoding only). Other libraries are:

- libx264 [64] - Open source encoding library for H.264.
- CoreAVC [7] - Commercial H.264 decoder, can use graphic card acceleration.
- Badaboom [10] - Commercial media converter, supports a wide range of codecs and can use graphic card acceleration.

For use in the prototype, libavcodec would be the most complete solution, as it support all necessary codecs, is free, open source, and platform independent.

4.3.2 Media coding hardware

Hardware solutions also exist specifically designed for high performance media coding. One example of such hardware is hardware encoders and decoders from On2 [40]. Hardware solutions can significantly improve coding speeds, but are expensive to consider for use in the prototype. Another alternative is the use of General Purpose computations on a Graphical Processing Unit (GPGPU [19]) through one of the special code libraries or extensions. This enables programmers to do computations on the Graphical Processing Unit hardware, enabling a significant speedup due to the highly parallel structure. Video coding can be parallelized easily, but although some codec software supports GPGPU, most current software codec libraries do not support GPGPU acceleration.

4.3.3 Network protocol libraries

Several network protocol libraries exist, supporting several languages. This section focuses on Java libraries, as the distributed middleware that will be used supports Java, which will lead to the easiest integration in the system. For HTTP several Java implementations exist. Java has its default network implementation in the java.net package. Furthermore, there is the option to create Java servlets or use the Spring web controller component [51] to handle HTTP requests. However, this prototype has the characteristic that the video file is not completely available when a user makes a request. To enable a fast response, streaming over HTTP must start as soon as possible, even when the video file is not yet completely loaded or modified. This results in the extra requirement that the library used must support long running responses. Typically, Java Servlet implementations and Spring MVC are not intended for long running responses. An alternative is
to use a framework based on Java New I/O (NIO). These frameworks provide more flexibility to handle network requests. Current frameworks are:

- Grizzly [3].
- Apache MINA [16].
- xSocket [1].
- JBOSS Netty [29].
- NIO Framework [54].

Each framework provides an abstraction for Java NIO, and support HTTP. All frameworks should support long running and non blocking responses, but on top of this, Netty provides framework support for the transfer of files in chunks, which aligns perfectly with the special requirement. Other frameworks would require more work to enable chunked HTTP responses. Furthermore, Netty is well documented, up to date and performs better than the other frameworks [33].

RTP implementations exists in different forms. There are several C/C++ libraries available (see [78], External links), and the Java Media Framework described below in section 4.3.4 also supports RTP. As with HTTP, RTP could also be implemented on top of a Java NIO framework.

### 4.3.4 Media frameworks

There exists a set of media frameworks that intend to simplify the development of software with media components. A very popular framework is GStreamer [23]. GStreamer is an open-source framework written in C, and intends to be used with different plugins. Plugins could then support the media coding, media formats or network protocols. Using these plugins a flow of components can be created where output of one component serves as input for the next component. GStreamer is widely used in media applications (see [23], Applications). A special Java framework for media distribution is developed by Sun, called Java Media Framework (JMF) [55]. JMF supports streaming protocols such as RTP, making it very suitable to implement a streaming server. However, JMF has not been maintained by Sun for a very long time, and is therefore very outdated.
4.4 Distributed middleware

The distributed middleware to be used has a few requirements based on the previous section. First of all it has to provide a compute grid to perform the video modifications and a data grid to store video data. Another requirement is the support for C/C++ besides Java, as there are no Java codec libraries available and the best alternative is the libavcodec C library. Finally, the middleware needs to provide scalability and fault-tolerance for the application.

When looking at these criteria, possible candidates are GigaSpaces and Hadoop. Other middleware requires additional products to provide computing grid, or do not support C++. Hadoop’s focus on Map/Reduce may be limiting, as there may be a need for custom services accessing the data grid. Another critical benefit of GigaSpaces is the automatic failure detection and recovery, providing the prototype with fault-tolerance.

4.4.1 GigaSpaces overview

This section provides a short overview of important aspects of GigaSpaces.

GigaSpaces has different topologies of data grids (spaces) that can be deployed, listed below:

- Replicated - Two or more space instances that have (a)synchronous replication.
- Partitioned - Two or more space instances divide data, forming individual partitions. load balancing is obtained by (custom) routing of objects.
- Master-Local Cache - Clients have a local space that acts as a cache. The cache is filled upon request with data from the master space.
- Local view - Clients have a local space, and define a template. Data matching the template is pushed from the master to the local space.

Systems can combine many different topologies in order to optimize data locality. This, together with the customizable routing of data, is a very important feature for the prototype. It allows the system to keep video data local on the same node, avoiding the transfer of generally large amounts of video data when processing. It also allows scalability by simply adding more partitions.

Space instances can run in the same Virtual Machine (embedded space) as the business logic components, or in their own VM (remote space). An embedded space allows fast access to the process running in the same VM, while a remote space is easily accessible for all components in the system. Each VM, running in a container called a Grid Service Container (GSC) is in contact with the Grid Service Manager (GSM). This management process provides fault tolerance by automatic restart of components upon detection of a failure of one of the GSCs. Besides a restart, a backup replication of the partition may immediately take over resulting in minimal downtime.

Business logic components (called Processing Units, or PU) consists of a set of event handlers that can read and write objects to either the global space or their local space partition. These objects
can be data objects or events objects, with possible notification or polling of these events to/by a handler. This provides communication through event objects containing all necessary data, and can result in asynchronous and event-driven architectures. Processing Units are intended to work on their individual set of objects, enabling shared nothing processing. These PUs are therefore ideal to process the bulk of video data, as each PU can work on its own piece of video data in parallel without needing data from other partitions. Events could be used to notify PUs that there is work to be done.

To conclude, the possibilities of GigaSpaces described above match with the design goals defined in section 4.1, and therefore make GigaSpaces a good choice. The data grid provides both data locality and load balancing through object routing. Furthermore, the prototype system could easily scale by adding more partitions. Optimal parallelism can be achieved using Processing Units with a shared nothing approach, eliminating locks and data contention. The fault tolerance requirement is fully met by the use of the automatic failure recovery of the Grid Service Manager. On top of that, other features may prove to be valuable for the prototype, such as the integrated webserver, the ability to execute code directly in the space partitions and Map/Reduce support.
4.5 Prototype design

The design of the prototype is presented in this section. First the functionality of the system is expressed using user stories. Then several components are defined and finally the flow of events for each user story is presented.

4.5.1 User stories

User stories are used in agile software development to formulate requirements on a non-technical level. A user story formulates the required functionality in several sentences from the perspective of a system’s user. This approach can be used as a contract between a customer and a developer, leaving technical details out of scope. For the prototype, the following list of user stories was compiled:

1. A user must be able to login and see a list of videos that is presented on a website.
2. A user must be able to request a video, using HTTP or RTP, from the list. This video is then streamed to the user.
3. A user must be able to see a modified version of the original video, based on a filter, where the video:
   - either shows an image in a predefined section as a overlay
   - or shows another video in a predefined section as a overlay
   - or is interrupted by another video for a short time

Filters are used in the system to specify sections of a video that can be modified, the coordinates of the sections, and the start and end time.
4. A user must be able to abort the video, such that the system can properly close the session.
5. An administrator must be able to login and define a filter.
6. An administrator must be able to see statistics of the system, presented on a website.
7. An administrator must be able to specify an external location where the system has to retrieve references to the content to be used in the sections defined in the filter.
8. An administrator must be able to specify the location of the video content.

The server side support for each user story is developed and tested separately, resulting in incremental development of features and requirements of the prototype.
4.5.2 Components

Based on the user stories, the features of the middleware and the properties of the external libraries that will be used, a set of components can be defined that make up the system. This section first presents the component diagram of the system and then describes each individual component.

Figure 4.1 is the UML component diagram for the prototype system. It shows the relations between each component. The SpaceProxy interface is the connection that remote components have with the entire partitioned space. The components that are local to a space partition only have a connection to their local partition.

![Component Diagram](image)

**Figure 4.1**: UML Component diagram of the prototype system.

Figure 4.2 on the next page tries to better illustrate how the components are distributed. It shows the partitioned space topology divided over $n$ physical machines.
Figure 4.2 shows that the director will be executing directly in a space partition. It also shows which components will be implemented as Processing Units. Processing Units are separated from the data grid but execute in the same VM, such that they still have fast access to data. For more details on deployment see section 4.5.4.

The use of individual partitions that modify the entire stream locally for each session ensures optimal performance through data locality on each partition and shared-nothing processing in the components, such that components can efficiently process data in parallel. This also allows parallel processing of multiple sessions on a single partition. This design also enables scalability of the performance by simply adding more partitions: when the performance of the system is insufficient, the partitioned approach allows the system to scale the performance up by adding more partitions deployed on more nodes. Correct routing of sessions ensures load balancing over the available partitions.

**HTTP Server** The HTTP Server will handle all HTTP traffic between a client and the system. It will present the website with the login page, and stream a (modified) video file to the user over HTTP. Normally, a HTTP response is constructed passing the entire video file in the response. This response is then transmitted as a sequence of individual TCP packets to the client, each containing a portion of the file. In the prototype system video streams will be modified dynamically while streaming, which usually takes some processing time. In order to minimize the delay for the user,
the server should start streaming as soon as possible. The HTTP Server must therefore be able to
start streaming blocks of video data even when the video file is not yet complete. To do this the
HTTP Server will need to correctly assemble the TCP packets before the video file is complete.
To facilitate the need for the fine grained control over the transmission, Netty has to be used to
implement the transmission algorithms for the prototype. The standard Java network classes could
also be used, but have more implementation overhead compared to a specialized framework. The
HTTP Server will have an embedded space that serves as a buffer for blocks of video data that are
waiting to be transmitted over HTTP. The buffer is filled by the Director in correct order.

**RTP Server**  The RTP Server will present the modified stream using the RTP protocol to the
user. Additionally the server uses RTCP and optionally RTSP for management of the connection.
Because RTP is a real-time streaming protocol, the real-time assembly of the video stream does
not create a problem. Like the HTTP Server, this component also has an embedded space that
buffers blocks of video data, waiting to be send over RTP. This buffer has to be adequately filled at
all times to avoid it running out of video data to send. Implementation of the RTP/RTCP or RTSP
can initially be done using the JMF framework. However it may also be implemented using Netty
to remove the dependence on an outdated framework.

**FilterProxy**  The FilterProxy serves as an interface to an external platform or component. This
proxy is intended to provide the Director with the content or content locations of media to be used
to modify the original stream. This proxy can have multiple implementations of its interface to
communicate with several different external components.

**Admin web console**  The admin web console will implement an HTTP Server to present the
administration web pages. It also gathers statistics from the system and displays them on a web-
page. Statistic events are pushed from each component in the system to the embedded space
asynchronously, where they are processed by the admin component. Here an easier solution can
be used to handle HTTP requests, such as a Java servlet or a Spring web controller. Since GigaS-
paces integrates Spring, using the Spring web controller would be the most logical choice.

**Director**  The Director\(^1\) component will run inside the partitioned space. A Director will start
each session, and track all states in the session. The Director is responsible for managing all the
events related to the video data. It tracks the video timestamps that flow from the content loader
into the space, and determines whether a block of video data has to be modified by looking at the
filter. Furthermore it tracks the size of the buffers of the assigned HTTP or RTP server and adjusts
the flow of data accordingly. The Director is the center of control for the final stream that will be
sent over the network.

---

\(^1\)In the filming industry, a Director determines how the end result will look, he/she will advise actors and controls
which takes are used in the end product. The Director component will play a similar role in the system.
**FilterService**  The FilterService is responsible for modifying blocks of video data. It receives events from the Director with attached data and instructions, and uses the functions in the FilterLibrary to process the event. The FilterService is implemented as a polling Processing Unit, periodically polling the space partition for new events. It will be written in C++ to allow the use of the C functions in the FilterLibrary. A FilterService runs independent of any session, so two instances of a FilterService can work on a video block belonging to the same session independently, allowing a high degree of concurrency.

**FilterLibrary**  The FilterLibrary is a Dynamically Linked Library (DLL) containing several functions related to video modification. The library can split video data into individual video, audio and subtitle streams, it can decode and encode video, merge video frames together, etc. It uses the libavcodec library for these tasks.

**ContentLoader**  The ContentLoader is a Processing Unit that loads video files from the content repository into the space partition. Different implementations of a ContentLoader can exist for different types of repositories, but initially the prototype will contain a content loader that reads video files from the harddisk. The ContentLoader parses the video file and writes blocks of video data into the space partition, separating the file at correct frame intervals. The content repository can be duplicated at each individual machine to avoid having to send large amounts of video data over the network. An optimization can optionally be done by loading the first $x$ seconds or minutes of each video into the space partition. This enables the Director to immediately start streaming instead of waiting for the ContentLoader to load the first blocks of video. The feasibility of this optimization depends on the amount of memory available in the physical machines, and the number of videos available, as this optimization may consume a large amount of memory.
4.5.3 Events

Each user story corresponds to a different flow of events in the system. This section presents the event flows of each user story. UML sequence diagrams are used to illustrate the sequence of events in this section. Note that UML sequence diagrams are not ideal for event driven architectures, and that each sequence is executed asynchronously. Furthermore, there are no actual method calls. Instead components react and communicate with event objects written to and taken from their space partition.

**Figure 4.3:** Sequence diagram for user story 1.

**Story 1:** A user must be able to to see a list of videos that is presented on a website.

In this story, a HTTP request to login and request a list of all available video files is handled. First the HTTP Server checks the credentials of the user. Then it sends the request for the list of videos to the space. The Director marks the start of the session and creates the session object. Followed by forwarding the request to the ContentLoader who in turn lists all his entries in the repository. The resulting list is forwarded to the HTTP Server and then send back to the user.
Story 2: A user must be able to request a video, using HTTP or RTP, from the list. This video is then streamed to the user.

Story 2 handles the normal streaming of a video, without any modifications. Upon request for a stream, a StreamRequest is written to the correct partition. This request is then picked up by the Director, who in turn updates the session and starts streaming. It starts requesting StreamPackets (a small part of the video file) from the ContentLoader. Each StreamPacket is logged by the Director and then forwarded to the HTTP or RTP server, who in turn streams the data from the packet out.
Story 3: A user must be able to see a modified version of the original video, based on a filter, where the video:

- either shows an image in a predefined section as a overlay
- or shows another video in a predefined section as a overlay
- or is interrupted by another video for a short time.

Filters are used in the system to specify sections of a video that can be modified, the coordinates of the sections, and the start and end time.

Like story 2, story 3 streams a video to the user over HTTP or RTP. However, this video may be modified on the fly. To do the modification, the Director checks the filter for each StreamPacket that it receives from the ContentLoader. If necessary, the Director issues a FilterPacketRequest for the content that has to be used along the original video in the modification. The Director then compiles a FilterRequest for the FilterService and writes it to the space, after which it is consumed and the FilterService constructs a modified version of the original StreamPacket.
Story 4: A user must be able to abort the video, such that the system can properly close the session.

When a user closes the connection, the session for that user will have to be closed at all components. The HTTP or RTP server will start by sending a ConnectionClosed event to the space. The correct Director will be notified of this event, starts closing the session and issues a CloseContent command to the content Loader. In the mean time the Director can safely clear all FilterRequest events from the space. When the ContentLoader is done, it writes back a ContentClosed event. The Director then finalizes the closing by removing all StreamPacket objects in the space, and notifying the HTTP or RTP server that the session is closed. The HTTP or RTP server in turn can safely clear it’s send buffer for that session. Closing the session has to be done in a synchronous way to be sure that no component writes events for a closing session where the space has been cleared of events, leaving events of a closed session in the space.

Figure 4.6: Sequence diagram for user story 4. Here illustrated using HTTP.
Story 5: An administrator must be able to login and define a filter.
Administrators will have the ability to login and define a filter to be used by a Director. An administrator first logs in and requests the webpage from the Admin web console, and then uses HTTP POST to send the filter details back. The Admin web console creates a Filter object and writes it to all space partitions so that all Directors have access to the new filter in their local partition.

Story 6: An administrator must be able to see statistics of the system, presented on a website.
Statistics are pushed to the Admin web console asynchronously. Upon a HTTP request for statistics, the Admin web console returns a snapshot of its internal statistical view. Figure 4.8 shows an the Director sending a StatisticsEvent to the Admin web console. The statistics event is placed in the statistics buffer and then consumed by the admin console. Upon request from the Administrator, all statistics can be compiled and send back.
Story 7: An administrator must be able to specify an external location where the system has to retrieve references to the content to be used in the sections defined in the filter. The administrator submits the location URI to the console, which in turn writes a ProxyLocation-Event to any space partition. The Filter proxy is then notified by the space and connects to the new location.

Story 8: An administrator must be able to specify the location of the video content. In story 8, much like in story 7, the new location of the Content Repository is written to all space partitions and all ContentLoaders at all partitions are notified by the space.
4.5.4 Deployment

Deployment of the prototype can be distributed over several nodes. There are the individual space partitions with the components that are attached it. The components that have a remote connection may run on a separate node. As mentioned in section 4.4, GigaSpaces uses Grid Service Containers as execution environment. All GSCs are managed by the Grid Service Manager. No replication or persistence will be used as the overhead for these mechanisms may negatively affect performance. Upon failure all data is lost, but a user can simply restart their session when the system is restored. Figure 4.11 shows a UML deployment diagram, illustrating a possible deployment of the prototype. The Grid Service Manager runs on a separate node. Business logic components run in a dedicated Grid Service Container. The individual space partitions also run in their own execution environment. Note that this is just one possible deployment. Other setups are also possible, for example where the GSM runs on the same node as the remote components and might even be duplicated on another node to tolerate failure of a GSM.

![UML Deployment diagram for the prototype system.](image-url)
Chapter 5

Result

This chapter will present the prototype system as it is implemented. The development process is discussed, followed by an overview of the system components as it is implemented. Then, the implementation of each component is covered.

5.1 Process

The development of the prototype system was done using the agile development process Scrum [79]. In Scrum, software development is done in time bound iterations. Each iteration (called a sprint) a number of tasks is selected that, when properly implemented, make a demonstratable result. All tasks are listed in what is called a Product Backlog, which is basically a large list of all tasks currently known to be applicable.

The Scrum method is more extensive than described above, but for this thesis a simplified version was used. A sprint was defined to be two weeks in length, as opposed to the more common one month length. The product backlog was constructed from the user stories described in section 4.5.1. Some user stories required more work than others, where some stories were broken down over two or more sprints. Each sprint was concluded with a demonstration and a reflection looking into what went well and what went wrong. The development process was supported by a definition of done, stating the requirements for tasks to be marked as completed. This definition includes the desired level of code quality, the coverage of documentation and unit and integration tests.

Reflecting upon the development process, the real challenge when using Scrum is to be able to clearly define tasks. This is greatly dependant on the quality of the design, as each deviation of the design during development may have effect on the time to complete the functionality for the sprint. Related to this is the ability to estimate the time it takes to implement a task, and complete all requirements for the task to be done. When the time estimate of tasks are incorrect, the sprint can either be finished prematurely or tasks may be unfinished at the end of the sprint.
5.2 Prototype

The prototype currently implements four full user stories: number 1, 2, 3 and 4, and some parts of user story 6. The first four user stories are the core functionality of the prototype and user story 6 is the support for statistics. The prototype allows users to login and see the list of videos available from the content repository (story 1). The original stream can be seen on a client side player displayed on the web interface, streamed using HTTP (story 2). In addition, the original stream can be modified by overlaying an image or other video before it is streamed to the user (story 3). Users can abort the stream gracefully, aborting the modification and streaming in the prototype (story 4). Users can also request a different stream while the previous stream is still playing. Finally, statistics on the throughput are collected during streaming (story 6).

The prototype is implemented in a total of 79 Java classes, of which 56 are prototype classes or interfaces and 23 are unit test classes. In total, 51 unit tests were written to test the Java components. All Java classes combined contain 4,887 lines of code. The JBoss Netty framework is used in the HTTP server component as a network layer abstraction. The HTTP server consists of the logic to communicate with the space partition and correctly use Netty’s API to handle HTTP requests and responses.

The FilterService consists of 7 C++ classes, containing 1,029 lines of code. The FilterLibrary includes libavcodec as an external library. Only the encoding and decoding functionality of libavcodec is used. Parsing file formats is done by the FilterLibrary. The logic of the FilterLibrary is spread over 10 code files and 10 header files, containing 1,864 lines of code. Testing of the FilterService and FilterLibrary was done manually.

All components implemented use the GigaSpaces API to communicate with the space partitions and correctly annotate classes to be used in the GigaSpaces platform. For more details, see Appendix A for a short description of each prototype class or function.

The rest of this section describes all the components of the prototype and the flow of events related to streaming.
5.2.1 Component overview

During development, some changes to the layout of components were done. The individual sections of each component cover more details of these changes. Figure 5.1 and 5.2 below illustrate the new component layout.

Figure 5.1: UML Component diagram of the prototype system as implemented.

Figure 5.2: Components of the prototype system as implemented.
5.2.2 ContentLoader

The ContentLoader is responsible for writing video content into the space. The video is divided into small blocks called StreamPackets. The handling of a request for a StreamPacket was altered to be able to request more than one StreamPacket at a time. This enables the ContentLoader to read multiple sequential StreamPackets from disk. As a result, IO seek operations are reduced because alternating between requests for different streams is reduced. See figure 5.3 for an illustration.

StreamPackets themselves turned out different than designed, their data must start with keyframe data and end on the frame before the next keyframe, with \( n \) keyframes in between. This approach is taken because of the way video coding works. In coded video, frame decoding starts with a keyframe containing the entire picture data. The next \( x \) frames (called intraframes) contain fewer data but together with the keyframe restore the complete frame. The keyframe and following intraframes together form what is called a group of pictures, and usually consists of 10 to 12 frames. This is why it is essential when dividing video frames, the individual blocks must start with a keyframe. Audio frames are interleaved with the video frames and do not depend on each other.

Each StreamPacket is marked with a sequence number. Sequencing has to be done because when modifying the StreamPacket, its size in bytes changes and therefore the offset to the next StreamPacket is no longer equal to the offset of the current StreamPacket plus its size. Also, StreamPackets are provided with meta data such as the number of frames, the timestamps of the frames, etc, and state information.

Finally, the ContentLoader now keeps all the files in the repository open during operation. Requests for StreamPackets calculate the position where to start reading from based on the request, therefore needing no state information per file or per session. This means that the sequence for user story four changes slightly: Notifying the ContentLoader that content should be closed is no longer necessary because the ContentLoader is stateless.

Figure 5.3: ContentLoader writing multiple StreamPackets upon request.
5.2.3 Filters

The creation of Filters as described in user story five has not yet been implemented. The prototype does however use Filter objects during streaming to determine if a StreamPacket should be modified. A number of Filters are predefined and written to each space partition upon deployment. Filters are assigned to sessions upon creation of the session by the Director. The Director queries the FilterProxy to request which Filter should be used for the session. The prototype has a dummy implementation of the FilterProxy that randomly selects a Filter for a session, such that the prototype can be used standalone.

5.2.4 Director

The responsibility of the Director is to keep track of the state of sessions and StreamPackets. To do this, several handlers are running in each space partition that respond to certain events or data changes. The component diagram (figure 5.2) shows all handlers implemented. The most important one is the SessionStreamHandler. This handler keeps taking active streaming sessions from the partition. It requests new StreamPackets when there are too few of them in the space for that session (figure 5.3), and registers this progress in the session state. When done, it writes the session back to the partition.

![Component Diagram](image)

**Figure 5.4:** A SessionStreamHandler checks each StreamPacket against the Filter (left). It then either forwards them to the HTTP server or issues modification by creating a FilterRequest and a FilterPacketRequest (right).

Figure 5.4 shows new StreamPackets being checked against the Filter in order to determine if they are to be modified by the FilterService, updating the state of the StreamPacket and request the ContentLoader for the data to modify the StreamPacket with. A FilterRequest is created specifying how and when the StreamPacket is to be modified. If StreamPackets are not to be modified or modification is complete, their state is marked for output over HTTP and the progress is set in the session.
5.2.5 FilterService

The FilterService consumes StreamPackets and modifies them according to the FilterRequest, using the FilterPacket that was requested earlier. Sessions are selected at random to enable a more or less equal distribution of processing. StreamPackets are ordered per session such that the StreamPacket with the lowest sequence number is processed first. The FilterLibrary is then used as described in section 4.5.2.

Figure 5.5 shows how first the ContentLoader writes the FilterPackets, followed by each FilterService modifying a StreamPacket and then writing the modified StreamPacket back.

5.2.6 HTTP Server

The HTTP Server is implemented using the JBoss Netty framework. The HTTP Server is integrated directly with the space partition (see figure 5.2). This was done to provide fast access to the data in the space partition, and eliminating the large network traffic in the grid when a single HTTP server would be used on a separate machine. At the same time this improves the performance scalability because each stream is now fully handled on a single partition, including streaming from the partition to the user.

Each instance of the HTTP Server binds to its own port on the host machine, making it possible to deploy multiple instances on the same machine. For each connection a separate handler thread is started, and this thread takes care of servicing the HTTP request. Figure 5.6 shows the final stage of the life cycle of a StreamPacket. The HTTP server takes the StreamPackets in sequence from the partition, streaming out their content in individual TCP packets 8192 bytes in size until the StreamPacket with the last sequence is reached.
Load balancing between the HTTP server instances of each partition is done by GigaSpaces object routing mechanism. The routing of each session is done based on the session id. All other objects and events in the space partitions also belong to a session and therefore also route using the session id. This results in the sessions being equally distributed over the partitions, and therefore equal distribution between HTTP Servers.

5.2.7 Web module

The user interface with the prototype is implemented as a website. The website serves to have users select a video that they would like to see, and then present it using an embedded video player. The web module is implemented using the built-in web container of GigaSpaces, and is separated from the HTTP server. This is done because of the simplicity of presenting a web page using the integrated web container, and the components can now also be individually scaled. The web module can display three different pages, a login page, a page with the available video streams and a page with a video player. It confirms the users identity by matching the credentials against data in the space. User entries are inserted in the space upon deployment, but this could easily be replaced by a database. The list of streams is fetched from the ContentLoader as described in section 4.5.3. Finally the last page holds a video player. For the prototype the JWPlayer [63] was selected. This is a standard open source Flash Video player that supports streaming video over HTTP. This player is initialized with the link to the HTTP Server that will stream the actual content.

5.2.8 Statistics

The support for statistics collection is not implemented as originally specified in the design. Instead of using an embedded space as a statistics buffer, a Map/Reduce approach is used. This is done to reduce the effort other components have to make to log statistical events. Also, viewing statistics on a web page is not yet implemented. Currently a log file is generated.

Figure 5.7 shows the process of collecting statistical events. All modules write their statistics to their local space partitions. The statistics module itself has a remote connection to the space. At certain intervals the module issues a distributed task. One task is written to each space partition (step 1) where the task col-

Figure 5.7: The Map/Reduce approach to collecting statistical events.
lects all statistics events locally, and performs sorting or another arithmetics (step 2). The result of each task is then reduced at the statistics module, where it is further sorted and logged (step 3). This approach needs very little resources from the other modules, as writing statistical events to the local space partition is very fast. The processing is done at the statistics module which, because of the remote space connection, could be running on a separate machine.

5.2.9 Summary

To summarize, above descriptions and illustrations give an overview of how the StreamPackets flow from component to component before finally being streamed out. The design has been altered on many fronts during the implementation phase. This was done either to improve the performance or to work around limitations of either the design or GigaSpaces. The challenge was to keep the flow of events as asynchronous and stateless as possible. This was accomplished by having each component aim for sharednothing processing: the ContentLoader can process requests in any order without keeping any state. The FilterService gets all the required data from the StreamPacket, the FilterPacket and the FilterRequest to produce a modified StreamPacket. This allows multiple FilterServices to work in parallel and independently of each other without locking. The HTTP server was placed local to the partition, and simply takes StreamPackets in sequential order and sends out their data.

Finally, because of their asynchronous and sharednothing nature each component can scale individually if needed. This was already shown for the FilterService, but can also apply to the other components. Multiple ContentLoaders could exists side by side in a single partition, as well as multiple HTTP servers and even multiple instances of the Director handlers.
5.2.10 User stories: sequence & deployment

During implementation the components changed slightly from the original design. This has an effect on the sequence of events for each implemented user story. This section presents the sequence diagrams for each implemented user story, to reflect the changes during implementation. The deployment of the components was changed slightly as well, which can be seen in the deployment diagram in figure 5.13.

Figure 5.8: Sequence diagram for user story one as implemented.
Figure 5.9: Sequence diagram for user story two as implemented.
Figure 5.10: Sequence diagram for user story three as implemented.

Figure 5.11: Sequence diagram for user story four as implemented.
Figure 5.12: Sequence diagram for user story six as implemented.

Figure 5.13: The updated UML Deployment diagram.
Chapter 6

Evaluation

In this chapter, the prototype will be evaluated. First, the experiment setup is given. Then, the maximum number of concurrent requests and scalability of the system is evaluated to see its performance and if it meets the non-functional requirement for scalability. Finally, the effects of varying the size of StreamPackets on the system are measured.

6.1 Setup

This section describes the setup of the evaluation. It covers the grid platform specifications, the content used in the prototype and a description of the test parameters.

6.1.1 Grid

Evaluation is carried out using the Amazon EC2 grid [49]. This is a virtual computing grid where users can launch grid nodes on-demand. For the evaluation the following systems were used:

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU</th>
<th># of cores</th>
<th>Frequency</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.large</td>
<td>Intel Xeon E5430</td>
<td>2</td>
<td>2.66 Ghz</td>
<td>7.5 GB</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>Intel Xeon E5430</td>
<td>4</td>
<td>2.67 Ghz</td>
<td>15 GB</td>
</tr>
<tr>
<td>m2.4xlarge</td>
<td>Intel Xeon E5550</td>
<td>8</td>
<td>2.67 Ghz</td>
<td>68.4 GB</td>
</tr>
</tbody>
</table>

**Table 6.1:** The Amazon EC2 instance types used in the evaluation.

Each system runs Windows Server 2008, Datacenter edition and GigaSpaces version 7.1 4300 GA.
6.1.2 Content

The content used in this evaluation was selected to provide a representation of video content commonly displayed on the Internet. This content is usually short (up to 10-15 minutes), and has a smaller resolution. Table 6.2 presents the properties of each video stream.

<table>
<thead>
<tr>
<th>Video</th>
<th>Bitrate (KB/s)</th>
<th>Length (min)</th>
<th>Keyframes</th>
<th>Size (bytes)</th>
<th>Resolution</th>
<th>Format</th>
<th>Codec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aquarium.flv</td>
<td>195.3</td>
<td>4.33</td>
<td>683</td>
<td>52,588,594</td>
<td>640x360</td>
<td>FLV</td>
<td>H.263</td>
</tr>
<tr>
<td>Demolition.flv</td>
<td>195.3</td>
<td>0.48</td>
<td>119</td>
<td>5,789,235</td>
<td>640x360</td>
<td>FLV</td>
<td>H.263</td>
</tr>
<tr>
<td>Education.flv</td>
<td>195.3</td>
<td>15.26</td>
<td>2316</td>
<td>99,219,645</td>
<td>640x360</td>
<td>FLV</td>
<td>H.263</td>
</tr>
<tr>
<td>Trailer.flv</td>
<td>195.3</td>
<td>2.30</td>
<td>320</td>
<td>21,288,245</td>
<td>640x360</td>
<td>FLV</td>
<td>H.263</td>
</tr>
<tr>
<td>Magnetism.flv</td>
<td>195.3</td>
<td>4.11</td>
<td>668</td>
<td>48,828,795</td>
<td>640x360</td>
<td>FLV</td>
<td>H.263</td>
</tr>
<tr>
<td>Volcano.flv</td>
<td>195.3</td>
<td>8.00</td>
<td>1201</td>
<td>83,245,158</td>
<td>640x360</td>
<td>FLV</td>
<td>H.263</td>
</tr>
</tbody>
</table>

Table 6.2: The video streams used in the evaluation.

This content was selected to provide a random representation of video streams. The content used to modify these streams consists of five different images in Bitmap format with a resolution of 100x100 and five short videos. The videos have a length of 15, 30(2x) and 60(2x) seconds with a resolution of 320x178. Each StreamPacket holds video data with 5 keyframes. The FilterProxy returns a specific filter for each video stream. These filters are set to modify between 10 and 25% of the original video stream. Note that the percentage of modification greatly affects performance, as modifying content is the most resource intensive, and is therefore kept fairly constant.

6.1.3 Test execution

Unless otherwise specified, the evaluation tests were executed as follows: execution lasts for 15 minutes. The test client starts concurrent requests with a one second interval. Once a request is completely serviced, a new request is sent after a one second interval. The test client runs on a separate EC2 instance to make a large network bandwidth available.

An extra instance is used to host the GigaSpaces management console as well as the components that are singletons and/or remotely connected to the space. This is done to make the full resources of each physical machine available to the partition it hosts.
6.2 Functionality

During development of the prototype unit tests were created to test each component and class functionally according the requirements of the user story. On top of that, upon completion of a user story the functionality was tested manually to confirm the correct behaviour from the user’s perspective. As mentioned in section 5.2, some user stories were not completed entirely or missing some elements. In these cases the completed functionality was evaluated to make sure that the user could still experience the greater part of the user story.

6.3 Fault tolerance

One of the non-functional requirements of the prototype is fault tolerance. Fault tolerance for the prototype is provided by the GigaSpaces platform. This platform registers the Grid Service Containers when they are started, and detects any failure. The Grid Service Manager always tries to deploy the number of partitions specified in the Service Level Agreements of the application. If a GSC is idle, the GSM will deploy a partition to the idle GSC when the active GSC fails.

To use this fault tolerance in the prototype, the prototype needs to be able to route new requests to the correct partition when one partition fails. Existing sessions on the failed partition are lost, and the prototype should properly redirect the user to the login page such that a new session can be started. To evaluate this, manual tests were executed with two partitions to verify that the routing of requests and sessions were correct. Two containers were used to deploy two partitions. One container hosts the web and statistics module. One extra container was idle that will take over the partition or modules that were deployed on the failing container.

6.3.1 Failing of a partition

When a partition fails, the sessions of the users that were stored in the failed partition are lost. Therefore the user has to be redirected to the login page such that a new session can be created. In the tests a session was created by logging in, showing the list of video streams. The container holding that session was then destroyed and the GSM deployed a new partition on the idle container. When refreshing the page, the prototype correctly redirected to the login page and the streams could be viewed on the recovered partition. When the partition fails while streaming, the user can refresh the page or press the back button of the browser to be redirected to the login page.

6.3.2 Failing of the web module

The web module can be hosted on a separate container or on the same container as a partition. The web module itself is stateless, all state is kept in the partitions. This means that when a web module fails normal operation is immediately restored when the GSM deploys the module again on an idle container. When the web module is hosted on a container also holding a partition, and this container fails, the web module can be relocated to any other running container.
6.4 Maximum concurrent requests

To measure the capabilities of the prototype system, the maximum number of concurrent requests will be measured on a single node. In order to provide a certain quality of service to the user, a minimum data throughput has to be found. At the same time this evaluation can provide the upper performance bound of the prototype in terms of the maximum active streams. The configuration for these tests uses the m1.large Amazon instance hosting a single partition with one FilterService instance. The number of concurrent requests was increased each test execution up to eight concurrent requests.

<table>
<thead>
<tr>
<th>Conc. Requests</th>
<th>Throughput Mean (KB/s)</th>
<th>KB/s /req.</th>
<th>Total requests</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,199</td>
<td>2,199</td>
<td>21</td>
<td>1,979,431,902</td>
</tr>
<tr>
<td>2</td>
<td>2,366</td>
<td>1,183</td>
<td>33</td>
<td>2,130,097,544</td>
</tr>
<tr>
<td>3</td>
<td>2,233</td>
<td>744</td>
<td>32</td>
<td>2,009,774,250</td>
</tr>
<tr>
<td>4</td>
<td>2,559</td>
<td>639</td>
<td>34</td>
<td>2,303,606,392</td>
</tr>
<tr>
<td>5</td>
<td>3,316</td>
<td>663</td>
<td>35</td>
<td>2,846,823,011</td>
</tr>
<tr>
<td>6</td>
<td>2,725</td>
<td>454</td>
<td>31</td>
<td>2,452,816,468</td>
</tr>
<tr>
<td>7</td>
<td>2,297</td>
<td>328</td>
<td>37</td>
<td>2,067,561,198</td>
</tr>
<tr>
<td>8</td>
<td>2,651</td>
<td>331</td>
<td>31</td>
<td>2,386,495,948</td>
</tr>
</tbody>
</table>

Table 6.3: Results of the maximum concurrent request evaluation.

Table 6.3 lists the results for the maximum concurrent request evaluation. It shows that the data throughput remains fairly constant when increasing the number of concurrent requests. To find the minimum throughput per request, and respectively the maximum number of concurrent requests, the minimum quality of service has to be determined. For the client to view the video stream without hickups or delays, given sufficient client resources, the incoming data rate has to be equal or greater then the bitrate.

The throughput when streaming over HTTP fluctuates heavily because portions of the stream that do not need to be modified can be streamed out very fast, while modifying each StreamPacket provides significant delays in the stream’s throughput. To tolerate these fluctuations, the minimum average data rate should be increased.

One approach is to set the minimum average throughput to $x$ times the bitrate of the video. When using $x = 2$ that would result in a minimum average throughput of $390.6(195.3 \times 2)$. Looking at table 6.3, the m1.large instance can handle up to 6 concurrent requests.

When looking at the instance resources in figure 6.1, the CPU utilization shows some fluctuation and a minor increase. When the number of concurrent requests is low there might be times where there are no modifications to be done for any of the requests and the FilterService (the most CPU

---

1The current implementation reads StreamPackets linearly from disk and then decides if it should be modified. Other approaches may be used to try and stabilize the throughput, for example loading and modifying StreamPackets in advance.
intensive component) is idle, decreasing the average CPU load. The slight memory increase is due to the increasing number of StreamPackets stored in the partition for each concurrent request. Notice also the large fluctuation in-memory usage, which may be caused by all the StreamPackets flowing through the system and kept in memory even when they are sent out. The Java garbage collector then removes all these StreamPackets from memory, resulting in a jigsaw pattern of memory usage.

Looking at the throughput and resource usage, the optimum number of concurrent requests for this configuration is between four to six. Making optimal use of system resources while providing a good quality of service to users.
Chapter 6. Evaluation

6.5 Scalability

When looking at scalability in distributed systems, two types can be defined: vertical scalability and horizontal scalability. Vertical scalability means adding more resources to a single node of the system. Horizontal scalability in turn means adding more nodes to the system.

6.5.1 Vertical scalability

To measure vertical scalability of the system, resources on a single node in the system are increased. For this evaluation, all three instance types of Amazon EC2 are used. The number of FilterService instances is increased up to the number of CPU cores of the instance, exploiting the parallel computation power. This should result in an optimal configuration for each instance. Following from the maximum concurrent request evaluation, four concurrent requests are used to provide enough work for each FilterService.

<table>
<thead>
<tr>
<th>EC2 instance</th>
<th>FilterService #</th>
<th>Throughput Mean (KB/s)</th>
<th>Total requests</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.large</td>
<td>1</td>
<td>2,559</td>
<td>34</td>
<td>2,303,606,392</td>
</tr>
<tr>
<td>m1.large</td>
<td>2</td>
<td>3,052</td>
<td>43</td>
<td>2,746,850,404</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>1</td>
<td>3,313</td>
<td>39</td>
<td>2,981,059,816</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>2</td>
<td>4,555</td>
<td>51</td>
<td>4,099,733,784</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>4</td>
<td>4,329</td>
<td>48</td>
<td>3,896,400,231</td>
</tr>
<tr>
<td>m2.4xlarge</td>
<td>1</td>
<td>5,052</td>
<td>64</td>
<td>4,546,711,050</td>
</tr>
<tr>
<td>m2.4xlarge</td>
<td>2</td>
<td>7,671</td>
<td>90</td>
<td>6,904,368,212</td>
</tr>
<tr>
<td>m2.4xlarge</td>
<td>4</td>
<td>9,774</td>
<td>108</td>
<td>8,796,211,906</td>
</tr>
<tr>
<td>m2.4xlarge</td>
<td>8</td>
<td>7,834</td>
<td>97</td>
<td>7,058,711,618</td>
</tr>
</tbody>
</table>

Table 6.4: Results of the vertical scalability evaluation.

Table 6.4 and figure 6.2 show the results of the vertical scalability evaluation. The m1.large instance shows a slight increase in throughput when increasing the number of FilterService instances. The m1.xlarge instance also shows increasing throughput when increasing the number of FilterService instances, but the performance is slightly worse when the number of FilterService instances is equal to the number of cores. The same trend is showing when increasing the number of FilterService instances on the m2.4xlarge EC2 instance. This trend might be the effect of Amdahl’s Law. The flow of StreamPackets in the system is sequential, limiting the maximum speedup of parallelization. Another reason might be that the four concurrent requests do not generate enough work for the FilterService instances.

The vertical scalability with respect to increasing the resources (using a more powerful EC2 instance) is clearly visible. For any given number of FilterService instances, the throughput increases when executing on a more powerful EC2 instance. For example when two FilterService instances are active, the average throughput increases from 3,052 KB/s to 4,555 KB/s when going
Figure 6.2: Vertical scalability of the prototype system.

from a m1.large to a m1.xlarge instance. In turn, the average throughput increases from 4,555 KB/s to 7,671 KB/s when going from a m1.xlarge to a m2.4xlarge instance.

Figure 6.3: Mean (●) and standard deviation (⊤) CPU utilization (left) and memory usage (right) during vertical scalability evaluation.

Figure 6.3 illustrates the resource usage during the vertical scalability tests. Average CPU usage increases when increasing the number of FilterService instances, due to more efficient use of the multiprocessor instances. The trend of the average CPU utilization supports the suspicion that
there is not enough work available for all the FilterService instances, as the average CPU usage decreases when the number of FilterService instances is equal to the number of cores.

6.5.2 Horizontal scalability

Horizontal scalability is the scalability of the system when increasing the number of nodes. For this evaluation up to eight m1.large EC2 instances are used. The number of concurrent requests is equal to four times the number of instances to provide more or less equal load on each partition during each test.

<table>
<thead>
<tr>
<th>Partitions</th>
<th>Conc. requests</th>
<th>Throughput Mean (KB/s)</th>
<th>Total requests</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>2.559</td>
<td>34</td>
<td>2,303,606,392</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>5.190</td>
<td>65</td>
<td>4,671,789,683</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>9.070</td>
<td>117</td>
<td>8,163,098,617</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>17.272</td>
<td>235</td>
<td>15,545,249,906</td>
</tr>
</tbody>
</table>

Table 6.5: Results of the horizontal scalability evaluation.

Figure 6.4 is a plot of the results of the horizontal scalability test which are listed in table 6.5. The almost straight line indicates almost linear scalability. This also confirms that each partition can work more or less independently and in parallel. The remote singleton components such as the web module, the FilterProxy and the statistics collection that may be single points of contention have very little impact on the scalability of the system. This is probably because they are used very infrequently during streaming and execute mostly on their own physical machine.
To conclude, the non-functional requirement of scalability of the system is met for both vertical and horizontal scalability.

### 6.6 StreamPackets

In this section the effect of varying the size of a StreamPacket is evaluated. Each StreamPacket holds a portion of video data with a certain number of keyframes. See section 5.2.2 for more details. The number of keyframes is varied resulting in a varied StreamPacket size. This may affect the throughput and modification time of StreamPackets. Tests were carried out on a m1.large EC2 instance with four concurrent requests.

<table>
<thead>
<tr>
<th>Keyframes</th>
<th>Throughput Mean (KB/s)</th>
<th>Size (bytes) Mean</th>
<th>Mod. time (ms.) Mean</th>
<th>Mod. rate Mean (KB/s)</th>
<th>Total modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.244</td>
<td>67,054</td>
<td>35,453</td>
<td>56</td>
<td>1197</td>
</tr>
<tr>
<td>2</td>
<td>1.335</td>
<td>148,605</td>
<td>70,752</td>
<td>116</td>
<td>1281</td>
</tr>
<tr>
<td>5</td>
<td>2.457</td>
<td>360,909</td>
<td>171,473</td>
<td>282</td>
<td>1280</td>
</tr>
<tr>
<td>10</td>
<td>2.885</td>
<td>641,054</td>
<td>307,717</td>
<td>652</td>
<td>982</td>
</tr>
<tr>
<td>15</td>
<td>2.619</td>
<td>1,055,545</td>
<td>521,683</td>
<td>1,185</td>
<td>890</td>
</tr>
<tr>
<td>20</td>
<td>2.238</td>
<td>1,372,109</td>
<td>610,618</td>
<td>1,079</td>
<td>748</td>
</tr>
</tbody>
</table>

**Table 6.6:** Results of the StreamPacket evaluation.

![Figure 6.5](image-url): Throughput (left) and modification rate (right) when varying the number of keyframes in a StreamPacket.

The plots in figure 6.5 illustrate the effect of increasing the size of StreamPackets on the system’s throughput and modification rate. There exists an optimal throughput at around 640 KB of data.
(10 keyframes) per StreamPacket and an optimal modification rate at 2 to 5 keyframes per StreamPacket (roughly 150 to 360 KB). These results may be the result of conflicting preferences of components in the system.

On the one hand, there is the ContentLoader who is most effective in loading large chunks of data into the space partition, as this reduces seek times on the harddisk. On top of that, larger StreamPackets enable more coarse grained parallel execution of the individual components.

On the other hand the FilterService may benefit from smaller StreamPackets, whose implementation requires lots of memory (copy) operations. For example, when a single frame is modified, it has to be reinserted in the correct position of the StreamPackets data. To do this all data after this frame has to be moved as the original frame is not likely to be of the same size as the modified frame. When the streams are larger in size these operations may negatively affect performance of the FilterService. To see the effect of each operation during the modification process, the execution times of each operation was measured. The results are listed in table 6.7.

<table>
<thead>
<tr>
<th>Keyframes</th>
<th>Mod. Time (ms.)</th>
<th>Initialization</th>
<th>Parsing</th>
<th>Decoding</th>
<th>Merging</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56</td>
<td>7.72 (13.79%)</td>
<td>0.74 (1.33%)</td>
<td>8.58 (15.33%)</td>
<td>3.87 (6.91%)</td>
</tr>
<tr>
<td>2</td>
<td>116</td>
<td>13.13 (11.32%)</td>
<td>1.26 (1.09%)</td>
<td>20.12 (17.35%)</td>
<td>7.99 (6.89%)</td>
</tr>
<tr>
<td>5</td>
<td>282</td>
<td>15.49 (5.49%)</td>
<td>1.69 (0.60%)</td>
<td>54.48 (19.32%)</td>
<td>19.32 (6.85%)</td>
</tr>
<tr>
<td>10</td>
<td>652</td>
<td>15.78 (2.42%)</td>
<td>1.87 (0.29%)</td>
<td>99.99 (15.34%)</td>
<td>35.98 (5.52%)</td>
</tr>
<tr>
<td>15</td>
<td>1,185</td>
<td>21.09 (1.78%)</td>
<td>2.65 (0.22%)</td>
<td>147.76 (12.47%)</td>
<td>60.93 (5.14%)</td>
</tr>
<tr>
<td>20</td>
<td>1,832</td>
<td>20.46 (1.12%)</td>
<td>3.11 (0.17%)</td>
<td>148.10 (8.08%)</td>
<td>82.07 (4.48%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keyframes</th>
<th>Mod. Time (ms.)</th>
<th>Encoding</th>
<th>Overwriting</th>
<th>Misc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56</td>
<td>24.22 (43.27%)</td>
<td>0.78 (1.40%)</td>
<td>10.07 (17.95%)</td>
</tr>
<tr>
<td>2</td>
<td>116</td>
<td>51.91 (44.75%)</td>
<td>4.27 (3.67%)</td>
<td>17.34 (14.95%)</td>
</tr>
<tr>
<td>5</td>
<td>282</td>
<td>135.32 (47.99%)</td>
<td>32.31 (11.46%)</td>
<td>23.37 (8.28%)</td>
</tr>
<tr>
<td>10</td>
<td>652</td>
<td>232.76 (35.70%)</td>
<td>216.98 (33.28%)</td>
<td>48.62 (7.46%)</td>
</tr>
<tr>
<td>15</td>
<td>1,185</td>
<td>368.77 (31.12%)</td>
<td>531.24 (44.83%)</td>
<td>53.09 (4.48%)</td>
</tr>
<tr>
<td>20</td>
<td>1,832</td>
<td>462.99 (25.27%)</td>
<td>1,057.28 (57.71%)</td>
<td>57.98 (3.16%)</td>
</tr>
</tbody>
</table>

**Table 6.7:** Profile of the modification process. All values display the mean of the measurements.

Table 6.7 confirms the large increase in the overwriting process of modification, as can be seen from the steep increase of 1.4% at 1 keyframe per packet up to almost 58% at 20 keyframes per packet. A different approach to overwriting modified frames in the result buffer may result in slightly better performance, but the amounts of memory that have to be copied or moved will still be present. Table 6.7 also shows the most processing time is used for encoding and decoding.
Looking at the CPU and memory usage in figure 6.6, there is no significant effect of the variation in the StreamPacket size. The metadata stored in a StreamPacket is obviously comparably small to the actual stream data, such that it does not cause increased memory usage when using small StreamPackets.

Overall, using 5 keyframes per StreamPacket seems to be optimal. It provides decent throughput and modification rates (table 6.6) due to the high percentages of time spent decoding and encoding.

### 6.7 Summary

This evaluation showed that GigaSpaces handles failing nodes by relocating partitions or components. The prototype can redirect user such that they can restart the stream when their partition failed. The evaluation of the maximum number of concurrent requests showed the performance of a single partition, and can be used to determine the maximum number of active streams with a minimum throughput per request. The vertical scalability evaluation showed that the prototype scales vertically when using the same component configuration, and this scalability is improved when optimizing the number of active FilterServices. Horizontal scalability evaluation showed that the prototype scales horizontally when adding nodes and partitions. Finally, increasing the size of the StreamPackets affects the throughput and modification rate of the prototype.
Chapter 7

Conclusion

This thesis explored the aspects of digital video streaming to develop a prototype system for distributed video modification and streaming.

In today’s digital video streaming, the MPEG standards, H.263 and the upcoming H.264 are the most popular video compression algorithms. Most online video services use H.263 in combination with the Flash video container format, while digital television standards use the MPEG-TS format in combination with MPEG-2 or H.264 coding. Network distribution of video is generally done using HTTP for pseudo streaming, RTP and RTSP for real-time of live streaming while digital video distribution is done using one of the DVB standards. Several data grid products were evaluated based on their set of supported features. The majority of products focus on an in-memory data grid, providing a distributed storage or distributed cache mechanism for fast access, replication, Map/Reduce support and possible persistence by (a)synchronous writes to a database.

A design was created for the prototype system that can enhance video streams for individual users, based on the requirements formulated in the introduction. External libraries were investigated to handle support for different aspects of the system. The codec library libavcodec provided the best set of supported features for coding video data. The Netty New I/O framework provides support for fine grained control over network streaming, support for HTTP and the possibility to easily implement the RTP protocol. From all the distributed data grid products, GigaSpaces offers the most useful set of features, specifically the native support for C++, its topologies and the computational features.

The prototype focused on HTTP streaming using Flash video, similar to many online video services. The components of the system are centered around a (in-memory) data partition, which holds data per user session. Video is split into chunks, making sure that each chunk can be individually (de)compressed by correctly grouping keyframes and intraframes. These chunks are written in the data partition where it is determined if the chunk’s video data is to be modified. All components have a shared nothing approach to processing, allowing concurrent processing and optimal scalability.
Evaluation was performed using Amazon EC2 instances and a test client that simulated user behavior. Manual evaluation showed that the prototype can tolerate failure by using the automatic failure recovery of GigaSpaces. It showed that the throughput of the prototype can scale both vertically and horizontally when a fixed number of concurrent streams are processed. Evaluation also showed the optimal size video chunks, being five keyframes per chunk.

### 7.1 Future work

The prototype can be further enhanced and optimized. The incompleted user stories could be implemented in the future. Support for more video codecs and formats can be added, as well as support for more streaming protocols (RTSP/RTP). The prototype might also use video coding hardware or (GP)GPU accelerated video codecs to improve performance. Support for different types of video content repositories could be added, such as loading video from a database. A different approach to overwriting modified frames in larger StreamPackets could be designed and implemented. Automatic replication of video content over all the nodes may be added to simplify distribution and synchronization of the video streams in the content repositories of each node.

Different approaches to handling video chunks in the data partition can be investigated, such as:

- Loading video chunks that have to be modified in advance, such that modification can start at the same time as streaming starts.
- Have certain chunks of video loaded at all times, for example the first $x$ seconds of video for a quick start of streaming to the user.
- Modified chunks could be kept in the partition for later use when the same chunk is modified often and in the same way.

Self management of the prototype could be added, such that it can automatically scale up and down based on performance demands. This can be extended by automatic allocation or deallocation of cloud resources.

A MapReduce oriented approach to the modification of a single stream could be also be an interesting topic. For example mapping the video chunks to different nodes where they are modified, and reduced by assembling the correct video stream again. Finally, this work could inspire systems that stream or broadcast other data which have to be enhanced or generated at real-time and/or on a per user basis.
Bibliography


[46] Richard E. Schantz, Joseph P. Loyall, Craig Rodrigues, Douglas C. Schmidt, Yamuna Krishnamurthy, and Irfan Pyarali. Flexible and adaptive qos control for distributed real-time and


Appendix A: List of classes

This appendix gives a short description of all classes and functions implemented for the prototype.

Common classes

The common classes are all classes that interact with the space partitions and are used by multiple components. They have to be annotated as such and provide methods for accessing their members. Furthermore they provide an unique Id and a routing method to determine the partition they should reside in.

Filter  Holds multiple FilterEntry objects. A Filter specifies timestamps and frame position and size where a stream can be modified.

FilterEntry  Specifies a start and end timestamp for a single modification. It also holds one or more FilterZone objects.

FilterZone  Defines the x and y coordinates, width and height of a rectangle where overlay content (image or video) can be placed over the original video frame.

FilterPacket  Holds image or video data that is used to modify the original stream. Holds the byte data as well as relevant meta data.

FilterPacketRequest  Event class used to request a FilterPacket from the ContentLoader.

FilterRequest  A request event for the FilterService for modification of a StreamPacket. Specifies the Id of the StreamPacket and FilterPacket and all details from the Filter.

ListRequest  Request event for the content loader for a list of all available video streams.

ListResponse  Response event that contains all available video streams. Is to be published to the user by the Web component.

Session  Contains all relevant data for a user session, including the StreamProgress of the current active stream.
**StartSession**  Event that requests a new Session to be created.

**Storyboard**  Holds the one to one mappings of a FilterEntry to a StoryBoardEntry and contains the Filter to be used for a given Session.

**StoryboardEntry**  Specifies which content is to be shown in a specific FilterEntry.

**StoryboardRequest**  Request event for the FilterProxy for a Storyboard for the given Session.

**StoryboardResponse**  Response event that contains the Storyboard for the requested Session.

**EndStream**  Indicates a video stream has been completely send out to the client.

**StreamPacket**  Holds a potion of video data along with video header information and meta data such as the sequence, the offset in bytes and state information.

**StreamPacketRequest**  Request event that requests one or more StreamPacket objects from the ContentLoader.

**StreamPacketOffsetComparator**  Comparator for sorting StreamPacket objects based on the offset.

**StreamPacketSequenceComparator**  Comparator for sorting StreamPacket objects based on the sequence.

**StreamPacketTimestampComparator**  Comparator for sorting StreamPacket objects based on timestamps.

**StreamProgress**  Tracks which parts of a video stream have been processed.

**StreamRequest**  Indicates that streaming is to be started.

**ThroughputSample**  Holds statistics on the throughput for a period of time.
ContentLoader classes

This section lists all classes that make up the ContentLoader.

**FilterPacketRequestHandler**  Periodically polls the partition for FilterPacketRequest events. It then uses the ContentRepository to write the FilterPacket back to the partition.

**ListRequestHandler**  Is notified when a ListRequest is written to the partition. It uses the ContentRepository to compile a ListResponse.

**StreamPacketRequestHandler**  Periodically polls the partition for StreamPacketRequest events. It then uses the ContentRepository to write the StreamPacket objects back to the partition.

**ContentRepository**  Interface for a content repository. Defines methods for getting a list of content and setting the content location, calculating sequence numbers and creating StreamPacket and FilterPacket instances.

**FileRepository**  ContentRepository implementation for reading files from the filesystem.

**ParserFactory**  Factory class that determines the format of a stream and creates a parser for the format.

**StreamParser**  Interface for a parser for a given video format. Defines methods for (key)frame based seeking, reading individual Frames and compiling the StreamFormat.

**FlashVideoParser**  StreamParser implementation for the Flash video format.

**ImageParser**  StreamParser implementation for reading complete images.

**Frame**  Represents a frame of data in a video stream.

**FlashVideoFrame**  Extends Frame. Represents a frame of Flash video data.
FilterProxy classes

This section lists all classes that make up the FilterProxy.

**FilterProxy** Interface for a external component that specifies overlay content.

**DummyFilterProxy** Dummy implementation of the FilterProxy interface for test purposes.

**StoryBoardRequestHandler** Is notified when a StoryBoardRequest is written to any partition. Compiles a StoryBoard and writes a StoryBoardResponse back.

HTTP server classes

This section lists all classes that make up the HTTP server.

**HttpStreamHandler** A JBoss Netty Handler implementation to handle HTTP requests for a stream. Writes a StreamRequest to the partition to initiate the flow of StreamPackets.

**HttpStreamPipelineFactory** JBoss Netty pipeline factory. Specifies a stack of handlers for the NIO channel.

**HttpStreamServer** Initializes the HttpStreamPipelineFactory, a thread pool for the handlers and sets the correct IP address and port.

**SpaceProxy** Proxy object for HttpStreamHandler threads to connect to the space partition.

**ChunkedByteData** Assigned by a HttpStreamHandler to a NIO Channel. Writes data in chunks from a StreamPacket to the channel and requests the next StreamPacket until the stream is completely processed.
Appendix. Appendix A: List of classes

**Director classes**

This section lists all classes that make up the Director.

**EndStreamHandler** Processes EndStream events by (re)setting the state of the session and clearing the partition from all objects related to that Session.

**ListResponseHandler** Is notified when a ListResponse is written to the partition and updates the Session and forwards it to the Web Module+.

**SessionStreamHandler** Periodically polls the partition for Session that are actively streaming. Requests new StreamPacket objects if there are too few in the partition, issues FilterRequest events when a newly arrived StreamPacket is to be modified. Marks StreamPacket for output and tracks the progress in the StreamProgress.

**StartSessionHandler** Is notified when a StartSession is written to the partition and initiates a Session.

**StreamRequestHandler** Is notified when a StreamRequest is written to the partition and prepares a Session for streaming.

**Statistics module classes**

This section lists all classes that make up the Statistics module.

**ThroughputLogger** Logging class that periodically collects ThroughputSamples from each partition using the ThroughputSampler.

**ThroughputSampler** Periodic timer task that issues a ThroughputCollector task.

**ThroughputCollector** Distributed task, executed in each partition that contains a map and reduce method. The map method collects the ThroughputSample from the partition and the reduce method sorts the results from the map operation.
Web module

This section lists all classes that make up the web module.

**LoginController** Handles the display of the login page and the login procedure.

**StreamController** Handles the display of the list of streams and the video player.

**LoginForm** Data object holding data to be displayed on the login page

**StreamForm** Data object holding data to be displayed on the video player page.

**StreamListForm** Data object holding data to be displayed on the stream list page.

**LoginValidator** Authenticates a user during the login procedure.

FilterService C++ class list

This section lists all classes that make up the FilterService.

**FilterPacket** C++ version of a FilterPacket

**FilterPacketSerializer** (Un)marshals FilterPacket objects from the partition.

**FilterRequest** C++ version of a FilterRequest

**FilterRequestSerializer** (Un)marshals FilterRequest objects from the partition.

**StreamPacket** C++ version of a StreamPacket

**StreamPacketSerializer** (Un)marshals StreamPacket objects from the partition.

**FilterService** Periodically polls the partition for FilterRequest events. Consumes the StreamPacket and FilterPacket and issues the modification in the FilterLibrary. Writes the result back to the partition in the StreamPacket.
Appendix A: List of classes

FilterLibrary C function list

This section lists all functions in the FilterLibrary DLL.

FilterLibrary(.h / .c)

- **modify** Exposed function of the DLL. Takes data buffers with StreamPacket and FilterPacket data and modification details. Delegates to either modifyOverlayImage or modifyOverlayVideo.

OverlayImage(.h / .c)

- **modifyOverlayImage** Iterates over all frames in the original stream buffer to overlay an image on the frames. Issues decoding, merging, encoding and overwriting for each frame if applicable.

OverlayVideo(.h / .c)

- **modifyOverlayVideo** Iterates over all frames in the original stream buffer to overlay a video on the frames. Issues decoding, merging, encoding and overwriting for each frame if applicable.

FlvParser(.h / .c)

- **fillFlvContext** Parses the flash video data to find properties relevant for initializing the codecs.
- **nextFlvVideoFrame** Retrieves the next frame from the buffer in a frame struct.
- **flvSkipTo** Skips to the frame at the given offset.
- **flvSkipToKeyFrame** Skips to the first keyframe before the given offset.
- **setFlvBuffer** Creates a buffer holding a frame according the flv standard from a frame struct.

BmpParser(.h / .c)

- **fillBmpContext** Parses the bitmap data to find properties relevant for initializing the codecs.
- **getBmpImageFrame** Retrieves the next frame from the buffer in a frame struct.

Decoder(.h / .c)

- **initDecoder** Opens a decoder using libavcodec.
- **decode** Decode a single frame using libavcodec.
- **closeDecoder** Closes the decoder.
Encoder(.h/.c)

- **initEncoder** Opens an encoder using libavcodec.
- **encode** Encode a single frame using libavcodec.
- **closeEncoder** Closes the encoder.

Merger(.h/.c)

- **merge** Merges two frames using one frame as an overlay.
- **convert** Converts the dimensions or colorspaces of a single frame.

Util(.h/.c)

- **overwrite** Overwrites a section of a buffer. Used to overwrite an original frame with the new modified frame.
- **checkTimestamp** Checks the timestamp of a frame against the modification timestamps to determine if the frame is to be modified.

Log(.h/.c)

- **logError** Logs an error message to the console.
- **logWarn** Logs a warning message to the console.
- **logInfo** Logs an info message to the console.
- **logDebug** Logs a debug message to the console.