COBURG UNIVERSITY
of applied sciences and arts

Coburg University of Applied Sciences and Arts

Faculty of Electrical Engineering and Computer Science

Degree Program: Computer Science (Bc)

Bachelor Thesis

Video-Streaming and Video-Annotation for case workers

Lukas Höllein

Submission Date: 09 February 2019

Examiner:

Prof. Dr.-Ing. Quirin Meyer, Coburg University

Abstract

1. English Version

Insurance companies must be able to provide their case workers with a complete digital workflow to watch and annotate videos in an event of a liability case. At HUK-COBURG no such workflow currently exists. This bachelor's thesis explores different approaches to stream videos to a case worker's office and to create a video player client. Furthermore, we develop necessary annotation tools, such as snapshots, comments and image processing functions. Besides the theoretical comparison of different frameworks and concepts, we build a fully functional client/server prototype containing all those features. For that prototype, we use "HTTP Pseudo Streaming" and "MPEG-DASH" as video streaming protocols in a video streaming server and develop a video player client as Java Swing application with the aid of "VLCJ" and "LibVLC".

2. German Version

Versicherungsunternehmen müssen ihren Sachbearbeitern ein System zur Verfügung stellen, in dem diese Videos anschauen und annotieren können, wenn ein Schadenfall bearbeitet werden muss. In der HUK-COBURG existiert so ein automatisiertes und voll digitalisiertes System noch nicht. Diese Bachelorarbeit untersucht verschiedene Ansätze, um Videos zum Arbeitsplatz eines Sachbearbeiters zu streamen und diese in einem Video-Player anzuzeigen. Außerdem werden Funktionen entwickelt, die zur Annotation eines Videos benötigt werden. Dies sind zum Beispiel Einzelbild-Aufnahmen, Kommentare und Bildverarbeitungswerkzeuge. Neben dem theoretischen Vergleich verschiedener Frameworks und Techniken, wurde ein voll funktionsfähiger Client/Server – Prototyp entwickelt, der diese Features enthält. Dieser Prototyp wurde mithilfe von „HTTP Pseudo Streaming“ und „MPEG-DASH“ als Streaming Protokolle für den Server entwickelt und stellt einen Video-Player Client bereit, der als Java Swing Applikation mithilfe von "VLCJ" und „LibVLC“ entworfen wurde.

Table of Contents

List of Figures

List of Tables

List of Code

List of Abbreviations

AE	Anwendungsentwicklung
API	Application Programming Interface
BGH	Bundesgerichtshof
DASH	Dynamic Adaptive Streaming over HTTP
DMS	Document Management System
HLS	HTTP Live Streaming
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HUK-COBURG	Haftpflicht-Unterstützungs-Kasse kraftfahrender Beamter Deutschlands a.G.
JAR	Java Archive
JMF	Java Media Framework
JNA	Java Native Access
JPEG	Joint Photographic Experts Group File Interchange Format
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
mbps	Megabit per second
MPD	Media Presentation Description
MPEG	Motion Pictures Expert Group
PDF	Portable Document Format
QoE	Quality of Experience
RGB	Red-Green-Blue
RTCP	Real-Time Control Protocol
RTP	Real-Time Protocol
RTSP	Real-Time Streaming Protocol

TCP	Transmission Control Protocol
TIFF	Tagged Image File Format
UDP	User Datagram Protocol
UI	User Interface
URL	Uniform Resource Locator

1 Introduction

This chapter introduces the context and general topic of the bachelor's thesis.

1.1 Context

The HUK-COBURG (Haftpflicht-Unterstützungs-Kasse kraftfahrender Beamter Deutschlands a.G.) issued and supervised this bachelor's thesis. It is the leading car insurance company for private households in Germany. The company was founded in 1933 in Erfurt and built up a broad and comprehensive supply of insurances, since then. By now, it acquired more than 11 million customers and hires about 10.000 employees [1].

At HUK-COBURG we are developing software in the department "Anwendungsentwicklung" (AE, German for "Application Development"). That department is divided into smaller areas and groups. One of those groups is responsible for document management, which controls and develops the digital archive used for all types of documents associated with the company and its customers. Among these types of documents there are digitalized correspondence, email traffic and other so called "input channels". In event of a liability case, pictures or videos of an accident are sent to the company, as well. We received more than 52 million documents of all types in the year 2016 [2].

More than thousand case workers use the document management system (DMS) every day to work off their cases. The documents can be accessed by and loaded into a specialized "Image Viewer" which allows the case workers to inspect and annotate documents for further processing or finalization of a task.

1.2 Problem and Goal

In an event of a liability case, for example a car accident, not only do we receive pictures of the scene, but customers are increasingly sending videos to us, as well. In the year 2017 about 300 videos arrived at our document management system each month [2]. This figure is likely to increase in the next few years, especially since the Bundesgerichtshof (BGH) – the highest court in the German ordinary jurisdiction system – declared dashcam recordings as valid evidence in 2018 [3]. Therefore, case workers need to access these videos and annotate them in the same way, as they annotate and process pictures in the "Image Viewer" nowadays. Currently, no complete digital workflow exists, that allows case workers to look at those videos.

Instead, they need to request a video, wait until a USB-drive containing it is sent to their office and then take that drive to an external work station to watch the video.

It is the goal of the company to develop a complete digital workflow for receiving, watching and annotating videos and to include it in the existing document management system. We break down this goal into the following objectives:

1. **Finding a suitable video container and codec:** Document management systems need to preserve documents for a long time due to legal reasons, most of the time for ten years [4]. Not only must the DMS be capable of saving videos, but it also needs to play back the videos years later. Therefore, it is one objective to find a good and sustainable video format, which most likely can still be played back many years later.
2. **Choosing a video streaming protocol:** Case workers will access videos from their work stations and transfer them from a video streaming server each time they issue a request. Due to the large file size of videos, they cannot be transferred completely in a single request in an adequate time. All well-known multimedia platforms, like YouTube or Netflix, use video streaming techniques to compensate for this problem by sending chunks of video files. This allows a user to watch the video while it is still loading. Multiple video streaming protocols exist and finding the best for our goal is one objective.
3. **Implementing a video player in the existing "Image Viewer" client architecture:** Today, case workers use one integrated Java Swing system to work off liability cases as well as other day-to-day routines. We must integrate our video player into that system as well. Therefore, finding suitable frameworks that are compatible to the existing application and developing a video player with these frameworks is one objective. Furthermore, the video player must provide annotation tools, which can be used by a case worker.
4. **Extending the existing archive- and server-system:** Currently, the DMS system cannot save and distribute video-based files. We must implement a new service at the server-side systems of the DMS, which can access videos from a database and stream them to the requesting clients.

1.3 Scope of Work

In this bachelor's thesis, we cover the objectives 2 and 3 completely. We analyze and compare multiple video streaming protocols under the following categories:

- compatibility to different video players and video codecs
- worldwide distribution and usability
- bandwidth limitations and performance

We compare multiple frameworks for building a video player, as well, under the following categories:

- compatibility to the existing "Image Viewer" architecture
- ability to use different video codecs and video streaming protocols
- ability to implement different annotation features

It is not an objective of this bachelor's thesis to suggest one or more video codecs used for archiving regarding the first requirement listed in "Problem and Goal". However, video codecs will play a significant part in choosing a video streaming protocol and a video player framework as suggested previously in this chapter.

Furthermore, extending the existing server-side architecture of the DMS is also not an objective. Moreover, we develop a prototype server application, that covers all aspects needed to implement a video streaming server. That prototype can then later be used to extend the existing DMS. It is important to note, that finding a suitable database system to store video files is therefore also no task in this thesis.

1.4 Thesis structure

In chapter two "Theoretical Background", we cover basic knowledge regarding document management systems, video codecs and formats, video streaming protocols and image processing features.

After that, we list detailed requirements for the video player in chapter three, that will define how a case worker wants to integrate videos in his day-to-day routines.

We compare streaming protocols and video player frameworks in the subsequent chapters. Each chapter concludes with a recommendation.

The next chapters then describe the development of the prototype. In chapter six “Prototype Development”, we will explain the general design decisions and architecture design of the prototype. In chapter seven “Testing” we will cover various aspects of prototype testing, for example bandwidth tests and video player feature tests. Finally, in chapter eight “Evaluation” we summarize the case worker representatives’ rating of our prototype regarding the correct implementation of their requirements.

In the final chapter “Conclusion and Prospect” we summarize the findings of the thesis and provide a prospect of the complete integration of videos in the DMS and future work.

2 Theoretical Background

In the following chapter we cover basic knowledge and concepts that are necessary for this bachelor's thesis. All other chapters reference the explanations in this chapter.

2.1 Document Management Systems

Document management systems (DMS) are used as basic infrastructure for documents of any kind, for example for contracts. The main purpose of a DMS is to provide ways of archiving documents that makes them unchangeable and accessible over a long period of time. Paper-based documents are digitalized and saved in a DMS, while electronic documents, like e-mail or pictures, are converted into a chosen archive format, for example TIFF, JPEG or PDF/A. Besides archiving documents, a DMS also provides ways to convert, index, access, export and view documents [5, p. 3ff.].

At HUK-COBURG we built our own DMS, which is shown in figure 1.

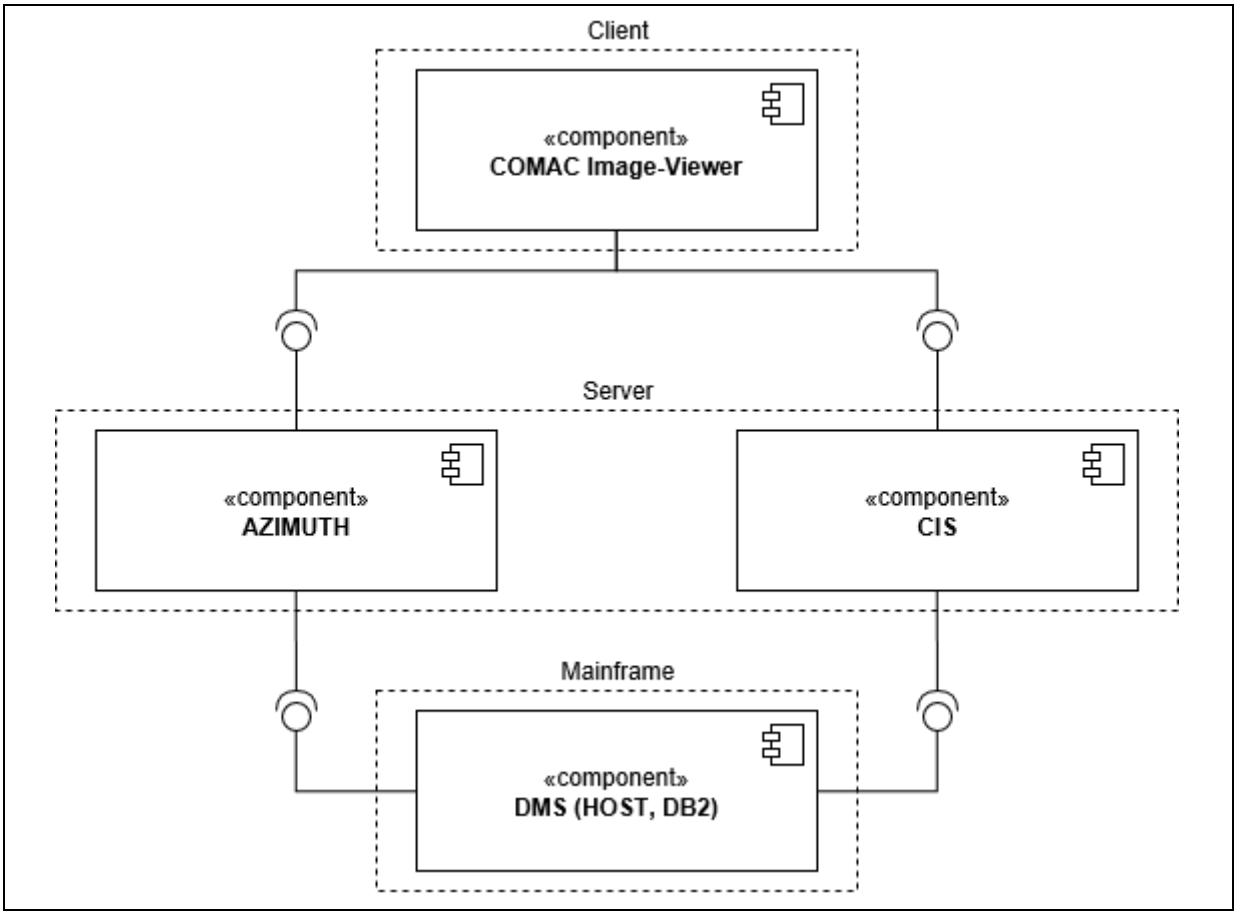

Figure 1: Component Diagram of the DMS at HUK-COBURG

Case workers interact with the “COMAC Image-Viewer” by loading documents into it. Currently, we use two independent server-side components, “AZIMUTH” and “CIS”, to handle requests issued by the client. These components access the main DMS component, which is hosted on a mainframe. The DMS component stores document data in a database (DB2) and provides basic modules to access them.

Figure 2 shows a typical usage of the “COMAC Image-Viewer”. It displays several pages of a document and renders annotations (yellow, green) on top of it.

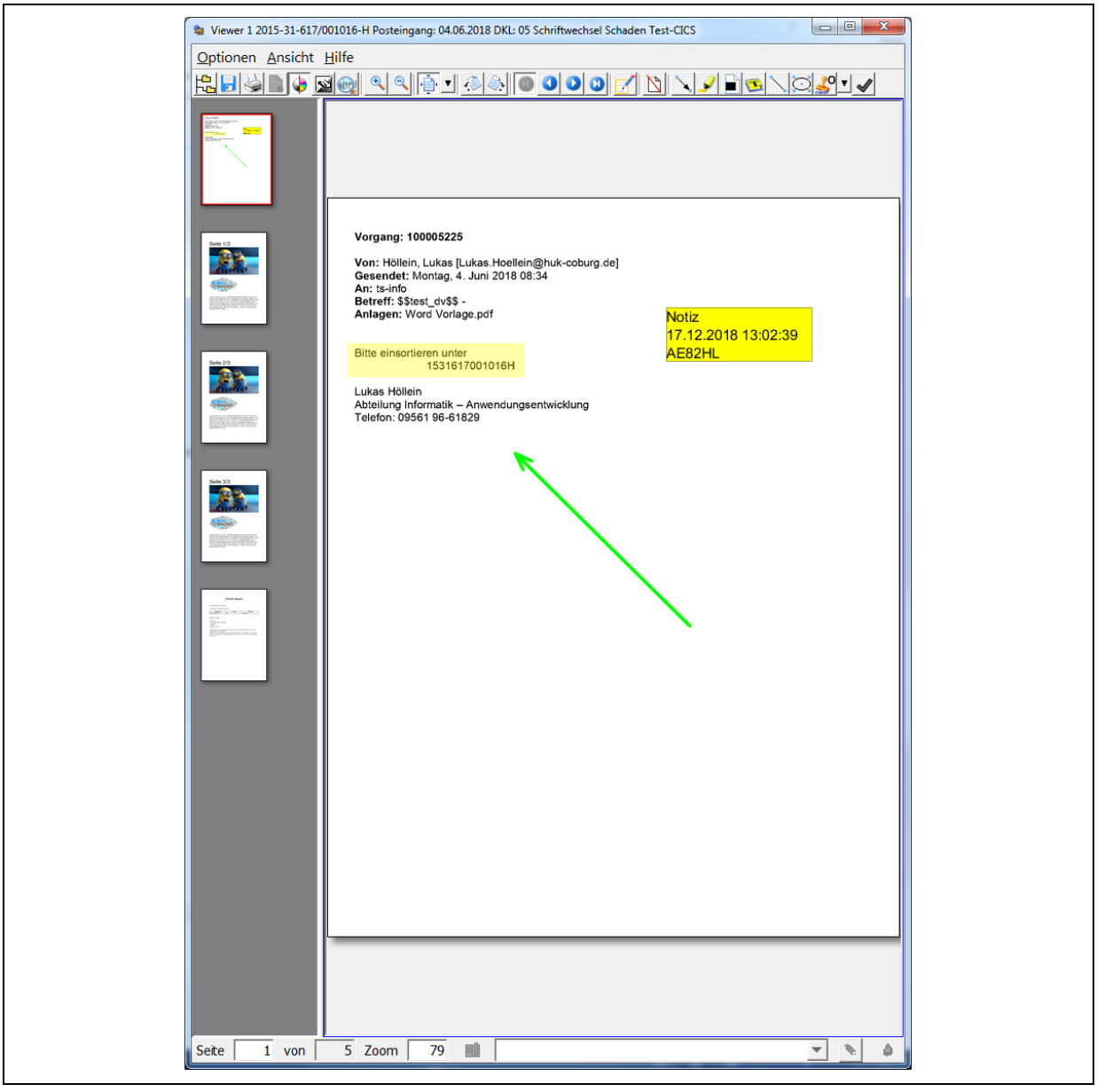

Figure 2: COMAC Image-Viewer example usage

2.2 Video Codecs and Containers

In an abstract sense, a video is a sequence of multiple images, which might be coupled with an audio track. When taking a closer look, multiple different approaches exist, which try to compress the raw video file – that is the sequence of multiple images – as efficiently as possible. Such an approach is necessary, because raw video files would be far too large to be handled by video players. These approaches are called video codecs [6].

Like any compression method, a video codec *decodes* the compressed data into a sequence of images and *encodes* a sequence of images into compressed data. A video codec can compress data *lossless* or *lossy*, depending on the type of the chosen codec. The more data is compressed by a lossy video codec, the more image quality is lost. This rate can often be configured in the encoding process and is called bitrate. It defines how many data is stored in one second of video [7].

Well-known lossy video codecs are “H.264” [8] and its successor “H.265” [9], both owned and developed by the “Motion Pictures Expert Group” (MPEG). Other modern codecs are the free, web-video focused codec “VP9” by “Google” [10] and the emerging open-source alternative “AV1”, created by “AOMedia” [11]. The most popular lossless video codecs are “H.264 lossless” and again “AV1”. The following figure shows the usage distribution of different video codecs in the year 2016.

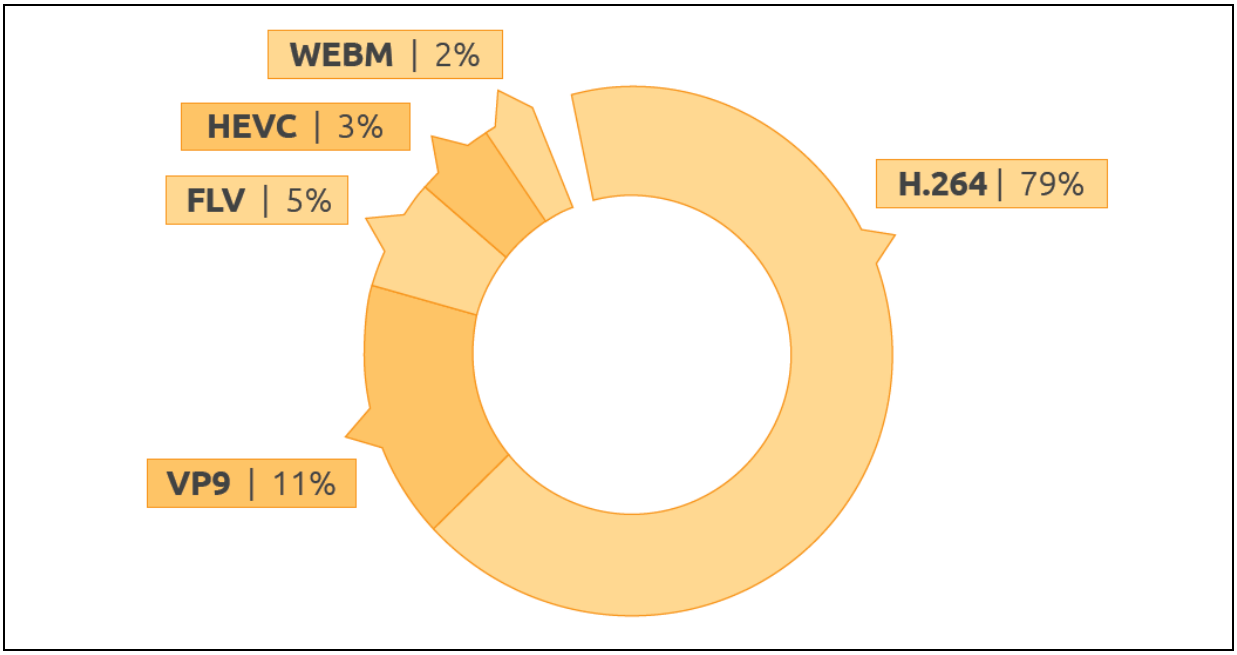

Figure 3: Usage distribution of video codecs [12]

A video container packages all types of compressed data into a combined structure. Among these types are the video codec data, as well as audio codec data and other metadata information. Video containers are often used as file extension property for videos, too. Well-known video containers are “MP4”, “WebM” and “MOV” [7].

Together, a video codec and container define the format of the video file. A video player must understand both the codec and container to achieve a successful decoding process and thus, video playback. Due to the large amount of available video codecs, most of the video players are limited in their ability to decode only specific codecs. Therefore, knowing which codecs are used is important to a real-world video application [7].

Another important characteristic of video codecs is their licensing model. When using a codec, one must obtain permission to generally use the codec and to use the programs that can encode or decode that codec. As seen in figure 3, the codec “H.264”, while being old, is still used by most of the end-users’ devices. This codec is not royalty-free for some use-cases [13]. Modern video codecs, like “VP9” or “AV1” are royalty-free, which is one of the key factors why they gain more and more market share in recent times [14]. According to Bakoski et al. [14], the need for a “competitive, royalty-free video codec” lead to the development of “AV1”.

2.3 Video Streaming Protocols

Many different approaches exist to stream video files. This chapter introduces the need for advanced streaming techniques and summarizes main concepts of different video streaming protocols. We do not compare the protocols in this chapter, but rather list their key characteristics.

Video traffic will take up 82% of the total amount of IP traffic by 2021 [15]. It is therefore necessary to provide videos in a bandwidth-friendly way to satisfy “Quality of Experience” (QoE) goals of users (e.g. how smoothly a video is playing) [16]. Different video streaming protocols differ greatly in their ability to optimally use bandwidth [17]. Other key factors to identify video streaming protocols are the underlying application-layer and network-layer protocols and the protocols’ requirements at the video files that it streams [17].

A further characteristic of streaming protocols is the context in which they are used. On the one hand, a user can access a video “on-demand”, which means streaming an already existing video file. Watching movies or video sequences on video portals is a classic example of this

context. On the other hand, a user can also watch live-streams, which means viewing a video that is not already finished. Big sports events are examples of that context.

Since we require only on-demand video streaming, we only introduce streaming protocols capable of this streaming technique.

The most primitive on-demand video streaming protocol is to download the entire video file completely through HTTP before beginning playback and therefore handle video files the same as any other static web resource, like regular HTML files. The biggest disadvantage of this technique is the need to download the video completely and to open it as a local file before watching. This technique is referred to as "HTTP downloading" [18].

A more advanced technique is called "HTTP Pseudo Streaming", sometimes also called "HTTP Progressive Download" [18]. The ability to request byte ranges of a resource, which was introduced in HTTP/1.1 [19], enables a client to request parts of a video file and start video playback while the remaining parts are still being loaded [17].

To compensate for bandwidth changes and to use the bandwidth even better, "Dynamic Adaptive Streaming over HTTP" (DASH) was proposed as a standard for video streaming by MPEG [17]. A video player switches seamlessly between multiple available video qualities, allowing for smoother video playback depending on the current bandwidth capabilities [17]. Besides "MPEG-DASH", other adaptive streaming protocols like "HLS" from Apple or "HDS" from Adobe are used, as well [20].

A different approach for video streaming is to abandon the HTTP protocol completely and use another application-layer protocol. The protocol RTSP is an example of that approach. Due to not utilizing HTTP and its well-known port, an RTSP streaming server and client always need to be able to connect via custom ports through firewalls [21].

2.4 Image Processing Features

This chapter briefly introduces the mathematical concepts to change the brightness, contrast and saturation of an image. The explanations are adapted from Burger and Burge [22, p. 60ff.] and Birchfield [23, p. 78ff.].

An image is a 2-dimensional array of pixels where each pixel is associated with a color value defined in a color space. A well-known color space is the RGB color space. The proposed image processing features fall in the group of point transformations. A point transformation

changes the color value of each pixel independently from other pixels and without changing the location of the pixel within the 2D array.

A change of brightness in an image gets expressed as a point transformation that adds a constant value or vector b to each pixel. The equation $I'(x,y) = I(x,y) + b$ adds a constant vector b to the pixel $I(x,y)$ in the image, where x and y identify the pixel's position inside the 2-dimensional image-array I [23, p. 78].

A change of contrast in an image gets expressed as a point transformation that multiplies a constant scalar or vector-valued factor c to each pixel. The equation $I'(x,y) = I(x,y) * c$ multiplies a constant scalar or vector-valued factor c to the pixel $I(x,y)$ in the image, where x and y identify the pixel's position [23, p. 78].

The saturation describes the "purity" of a color. It defines "the degree of difference between the color and neutral gray" [23, p. 411]. As a result, desaturating an image completely results in a grayscale image, while oversaturating results in unnatural colors. Complex saturation functions are applied to achieve this behavior.

Figure 4 visualizes all image processing features that we described.

Figure 4: Original (left), brighter and sharper (middle) and desaturated picture (right)

3 Requirements

In the following chapter, we list all requirements provided by case worker representatives concerning a video player integration. This will define, how case workers want to interact with videos in their day-to-day routines. While most requirements are mainly functional and describe general use cases, some non-functional requirements will be listed, as well. We obtained all requirements through conducting an interview with representatives of different divisions in the company.

3.1 Functional Requirements

We obtained the following functional requirements as user stories. A table describes each user story by listing it together with its acceptance criteria. Additionally, we provide a visual representation of some user stories for better comprehension.

Table 1 lists the first user story.

User Story	As a user, I want to access videos through the “Image Viewer” client by pressing on a “play” button so that I can watch the video.
Acceptance criteria	When I press on a “play” button, the video player will become visible in an independent window.
	Videos are represented to me in the “Image Viewer” as a normal page, which shows a “play” button on top of it.
	Each page, that is representing a video, is showing me a thumbnail of the video.
	When I press on a “play” button, the video will start at the time of the thumbnail.

Table 1: User-Story 1

Figure 5 visualizes the requirement explained in the first user story. It shows a video thumbnail as page in the “Image Viewer” together with a play-button.

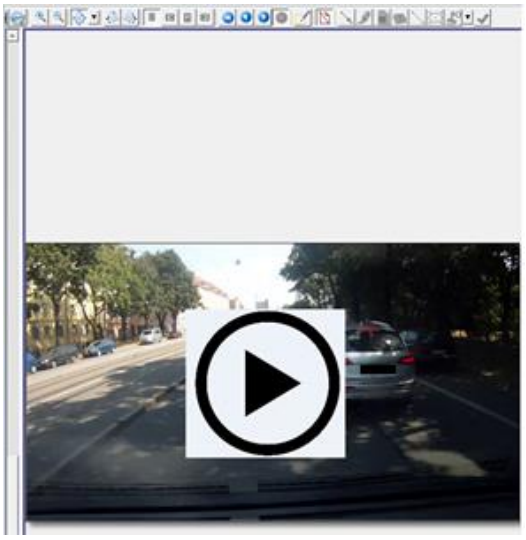

Figure 5: Play-Button on a page in the Image-Viewer

Table 2 lists the second user story, which describes general video controls present in many video players.

User Story	As a user, I want to control video playback through basic functions by pressing on dedicated buttons or sliders on a toolbar so that I can analyze the video comprehensively.
Acceptance criteria	When I press on a “Play / Pause” button, video playback stops or continues
	When I press on a “Next-Frame” button, the video plays back frame-by-frame.
	When I press on a “Rewind” button, the video can be rewound.
	A timeline suggests me the current and remaining playback time. I can change the playback time by clicking on that timeline.
	When I press on a “full screen” button, the video player toggles using all the available screen size.
	I can alter the playback speed to be slower or faster.
	I can activate sound playback. Per default, sound is muted.
	I can change and reset the brightness, contrast and saturation of the video.
	I can use an interactive zoom-lens that zooms in on a selected area of the video. I can adjust the zoom-level.

Table 2: User-Story 2

We list the next user story in table 3.

User Story	As a user, I want to create comments about the video so that I can write down and save useful information.
Acceptance criteria	Comments are listed next to the video in a comment feed, that I can expand or collapse.
	When I write a comment, each one is associated with the current playback time of the video.
	When I press on a comment, the video jumps to the associated playback time (comment acts as “bookmark”).
	When I write a comment, each one adds and displays my user information and the creation date.
	I can delete comments.
	When I reach the associated playback time of a comment, its border color changes to red as a signal that I reached this comment.

Table 3: User-Story 3

Figure 6 visualizes, how a comment feed shall look like according to the third user story. It contains two example comments with a creation date, the user that wrote the comment and the playback time of the video. The current playback time of the video is at thirteen seconds, which is why the corresponding comment is highlighted in red.

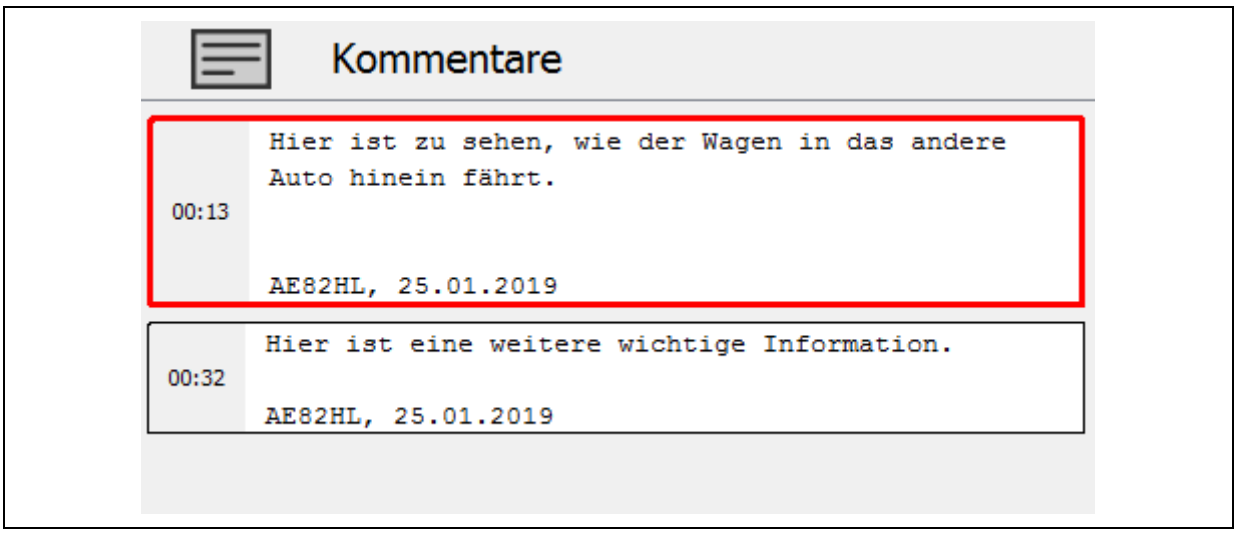

Figure 6: Example of a comment feed

Table 4 contains the next user story.

User Story	As a user, I want to transfer snapshots of the video into the “Image Viewer” so that I can extract and annotate important pictures.
Acceptance criteria	I can press on a “snapshot” button.
	When I take a snapshot, the video pauses at the exact time of the snapshot.
	After I take a snapshot, it becomes visible as an extra page in the “Image Viewer” application.
	I can annotate the extra page in the “Image Viewer” the same way as every other page.
	The extra page displays a “play” button, which will play back the video from the time the snapshot was taken when I press on its button.
	A snapshot will always be taken with the best video quality, even if I play back the video with a lower quality.

Table 4: User-Story 4

The left column of figure 7 shows multiple snapshots of an example video in the “Image Viewer”. One of them is being annotated on the right with the tools offered by the “Image Viewer”. Each snapshot references the video at the time of its creation.

Figure 7: Snapshots of a video inside the “Image Viewer”

Table 5 describes the last user story.

User Story	As a user, I want to choose the quality of the video so that I can influence the buffering time.
Acceptance criteria	A dynamic list of all available qualities for a video is displayed to me in the toolbar.
	When I choose a new quality, the video changes to that quality immediately.

Table 5: User-Story 5

Figure 8 shows a dropdown-menu of three available video qualities, which satisfies the fifth user story.

Figure 8: Dropdown-Menu of available video qualities

We combined all user stories in a use-case-diagram in figure 9. Its main actor is the user, who interacts with the video player and the “Image Viewer” and thus triggers one of the user stories listed above. The “Image Viewer” is a technical actor, which is involved in some of the use cases.

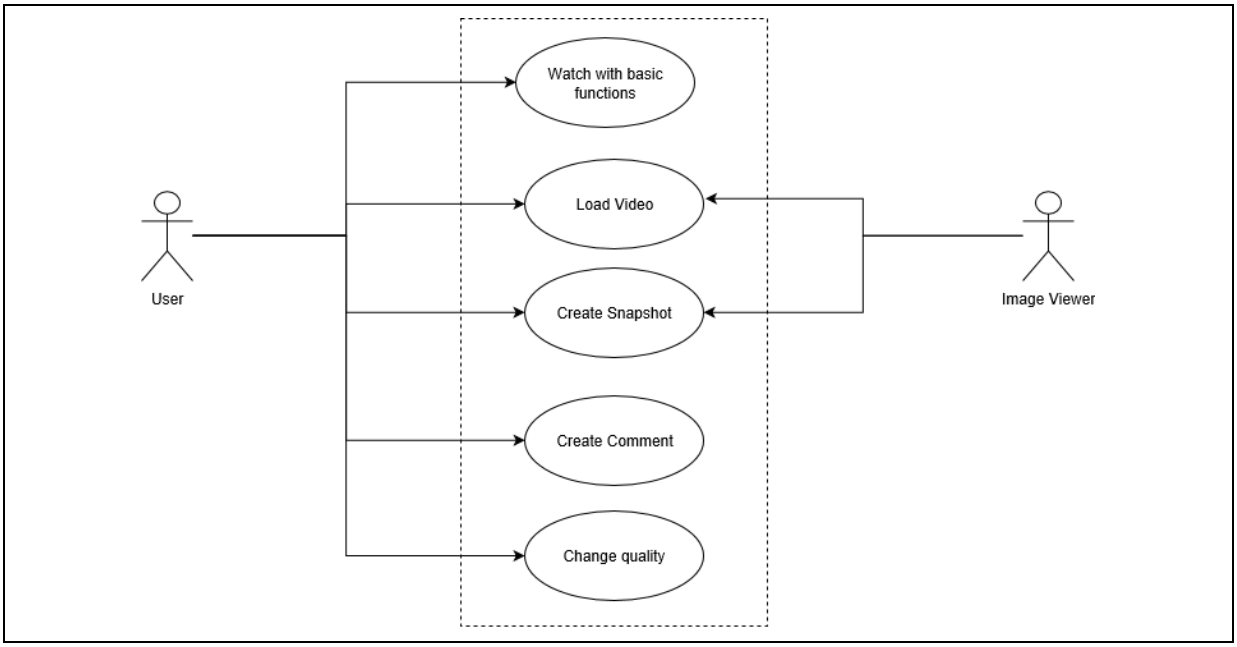

Figure 9: Use Case Diagram

3.2 Non-Functional Requirements

We obtained the following non-functional requirements. They are listed with short explanation texts, instead of specifying them with complex methods, making them straightforward to understand.

1. **Look-And-Feel:** The video player shall look similar to the existing “Image Viewer” client. The “Image Viewer” client is a Java Swing application with classic UI elements inherited from Java Swing. Figure 2 shows these UI elements. To achieve a good integration of the video player in the existing infrastructure, as described in “Problem and Goal”, it is required, that the video player looks and feels the same as the “Image Viewer”, when using it.
2. **Usability:** The functions of the video player are easy to understand. Therefore, tooltip texts are provided for each action in the toolbar. Interactions between the video player and the “Image Viewer”, like taking a snapshot, are easy to execute and do not cause noticeable delays.
3. **Buffering Time:** The video can be watched with acceptable buffering delays. A user is informed comprehensibly, if the video is buffering. The user can take actions, like choosing a lower video quality, to reduce waiting times, due to buffering.
4. **Reliability:** The video player shall function properly even after multiple different videos from different documents are loaded into it. The user is not confronted with system errors, but receives meaningful error messages, if errors occur.

4 Comparison of Streaming Protocols

In the following chapter, we compare streaming protocols against different categories. We conclude with a recommendation of one or more streaming protocols suited for use in our video player application.

The categories of comparison are:

- **Compatibility to different video players and video codecs:** As stated in chapter 2.2 “Video Codecs and Containers”, many different video codecs exist and not every video player can decode every video codec. We will investigate, whether video streaming protocols restrict the usage of video codecs or video players.
- **Worldwide distribution and usability:** Some streaming protocols are more widespread and feature-rich than others. We will examine the state-of-the-art of specific streaming protocols and whether a usage in our application is reasonable. Especially, we check the protocol’s ability to perform seeking operations, that is skipping from the current playback position to a completely different position. After all, we want to build our application on top of well-known, established streaming standards.
- **Bandwidth limitations and performance:** As stated in chapter 2.3 “Video Streaming Protocols”, some streaming protocols better utilize bandwidth availability than others. At HUK-COBURG, different branch offices are connected to our main computing center with different bandwidth throughput. It is essential for our video application, to provide a reliable solution to all branch offices, as stated in chapter 3.2 “Non-Functional Requirements”. Consequently, we will compare streaming protocols against their ability to utilize bandwidth and to deal with bandwidth limitations. Especially, we investigate the ability of each protocol to change the video quality as a measure to compensate for a lower bandwidth.

In the following, we compare the video streaming protocols “RTSP”, “HTTP Pseudo Streaming” and “MPEG-DASH” against the proposed categories. We choose these three, because they represent all types of video streaming protocols, which currently exist: streaming without usage of the HTTP protocol, streaming with simple HTTP functionalities and advanced streaming techniques.

4.1 RTSP

The “Real-time Streaming Protocol” (RTSP) was developed in 1998 to control streaming servers and clients. Typically, it utilizes the “Real-time Transport Protocol” (RTP) and the “Real-time Control Protocol” (RTCP) for delivery of streaming data. It offers a basic request-response message workflow, similar to HTTP, to setup, play, pause and terminate a streaming session. The actual video data is then sent via the selected transport protocol, typically either TCP or UDP [21], [24].

4.1.1 Compatibility

In the following, we list all results regarding compatibility of RTSP.

- RTSP limits the supported video codecs, due to its usage of RTP. While most of the popular video codecs, like H.264 and VP9 are supported, this might not be the case for all video codecs [25].
- A video player must support the RTSP protocol to receive such video streams. Nowadays, many video players do no longer support the RTSP protocol. For example, modern web-browsers like Chrome or Firefox offer no native support for RTSP [26]. While some video players, like “VLC Media Player” support this streaming protocol, it is no longer used for general purpose video platforms, but instead for special use cases like IP camera live-streaming [27].

4.1.2 Usability

In the following, we list all results regarding usability of RTSP.

- RTSP is not commonly used for video platforms, but rather for live-streaming in some special use cases, as stated previously. A contributing factor is the need to unlock special ports used by the protocol. In contrast, HTTP as standard protocol for the world-wide web does not require any special port unlocks, because its ports are unlocked by default on nearly any computer, anyway [28].
- Additionally, performing seeking operations in videos streamed by RTSP is often not working smoothly [28].

4.1.3 Performance

In the following, we list all results regarding performance of RTSP.

- Bandwidth limitations affect video streaming via RTSP very much, leading to dropped frames and distorted video images more quickly than through other HTTP-based protocols [29].
- A change of video quality is possible with RTSP, but it requires the video player to terminate the existing streaming session and build up a new one, resulting in a lot of overhead work [21].

4.2 HTTP Pseudo Streaming

"HTTP Pseudo Streaming" utilizes the possibility to request parts of a resource from a web server. A video player repeatedly issues so called "partial content requests", which leads to the partial transfer of a video file, as already described in chapter 2.3 "Video Streaming Protocols".

Code 1 shows an example request for this behavior. A resource is requested via standard HTTP, but in addition a "Range" attribute is specified in the HTTP header.

```
GET /video.mp4
HTTP/1.1 Host: videos.my-company.com
Range: bytes=0-1023
```

Code 1: HTTP Partial Content Request

Code 2 displays a typical response for the request issued in Code 1. The header attributes "Content-Range" and "Content-Length" specify, which and how much bytes are returned and how many are totally available for the requested resource.

```
HTTP/1.1
206 Partial Content
Content-Range: bytes 0-1023/146515
Content-Length: 1024
...
(binary content)
```

Code 2: HTTP Partial Content Response

A video player requests the next part of a resource as soon as it needs to buffer the next part of the video. This process continues until either the video is streamed completely, or the video playback is stopped.

4.2.1 Compatibility

In the following, we list all results regarding compatibility of "HTTP Pseudo Streaming".

- "HTTP Pseudo Streaming" does not limit the video codecs, which can be used. Receiving just a normal HTTP request, the streaming server does not even need to know, that it is responding with video content.
- Many video players support the HTTP protocol. Therefore, the only remaining limitation for a video player is the ability to decode the binary content it receives, but not the decoding of the protocol's response.

4.2.2 Usability

In the following, we list all results regarding usability of "HTTP Pseudo Streaming".

- No firewall problems emerge through the usage of HTTP, as we described earlier.
- The streaming protocol can handle seeking operations without problem by simply requesting a corresponding byte range, which does not need to follow up on the previous byte range. A video player needs to convert the new time-based position into a byte range for this concept to work. A so called "MOOV Atom" is used in many video containers, for example in "MP4", which stores metadata like framerate, bitrate, duration and size for the whole video file. As soon as a video player receives this "MOOV Atom", it can convert a time-based position into a byte range by using the provided information. Therefore, it is essential for this protocol to receive the "MOOV Atom" at the beginning of a new streaming process. The easiest way to achieve this, is to store the metadata information at the beginning of a video file. Usually, a video player requests the beginning of a video file with its first byte range request, thus receiving the required information [30].
- A video player does not store a copy of the streamed video file on its local file system when using "HTTP Pseudo Streaming". Instead, only a configurable amount of video cache remains in the computers volatile memory, hence making it more secure. A downside of this technique is the need to re-download parts of a video, which were al-

ready streamed, but no longer reside in the computer's memory. This problem can be solved by increasing the video cache size of the video player client accordingly. Furthermore, by applying meaningful caching techniques at the streaming server, we can ensure a quicker access to recently streamed video files [31].

- A video player can decide whether to request multiple short byte ranges or one large byte range. Requesting multiple short byte ranges results in more requests being issued, while one large byte range needs to be transferred in chunks in one HTTP response.

4.2.3 Performance

In the following, we list all results regarding performance of "HTTP Pseudo Streaming".

- In general, the protocol can utilize bandwidth limitations better than RTSP, resulting in less dropped frames and distorted images [29].
- Choosing another video quality can be performed quickly, due to HTTP being stateless. No overhead work, like closing a session, needs to be performed. Instead, the next byte range can simply be requested from a lower-quality resource. The only new information needed, is the "MOOV Atom" of the new resource.
- The protocol itself offers no mechanisms to switch between qualities automatically. Instead, a user must select another quality manually or a video player must guess the right quality somehow depending on current bandwidth capabilities.
- In general, "HTTP Pseudo Streaming" does not waste bandwidth, because only byte ranges are requested, instead of downloading a video completely. When video playback is stopped in the middle of a video, the video player simply stops requesting new byte ranges. The only "unused" data is received in the last byte range request before interruption, which might not be watched completely. Nevertheless, since a video player defines how big a byte range request will be, this still can lead to bad bandwidth utilization, when requesting large ranges and stopping video playback in the middle of it [17].

4.3 MPEG-DASH

"Dynamic Adaptive Streaming via HTTP" by MPEG is based on the same byte range request technique as "HTTP Pseudo Streaming". In addition to it, a metadata file, the so called

"MPD" (Media Presentation Description) file, lists all available video qualities and their bitrates. Receiving this information at the very beginning of a streaming process, a DASH-based video player knows exactly which qualities are available at which bandwidth capability. Additionally, such a video player implements multiple selection mechanisms, which define when to choose which video quality (e.g. switch to a lower quality as soon as bandwidth begins to drop, try to always display the best quality even when buffering becomes more unreliable,).

Furthermore, the streaming server holds one or more so called "segments" of the original video file as separate files. A video player receives information about the number of available segments in the MPD file, as well. This mechanism evolves the byte range concept used in "HTTP Pseudo Streaming" by creating small, separate files, instead of only using byte range requests to the same, large file [17].

The selection mechanisms and the usage of multiple segments allow the video player to seamlessly switch from one video quality to another without the video stopping playback [17]. This characteristic is called "adaption" and represents a whole class of "adaptive" video streaming protocols, with "MPEG-DASH" being one of them. We show an example of switching between qualities in figure 10.

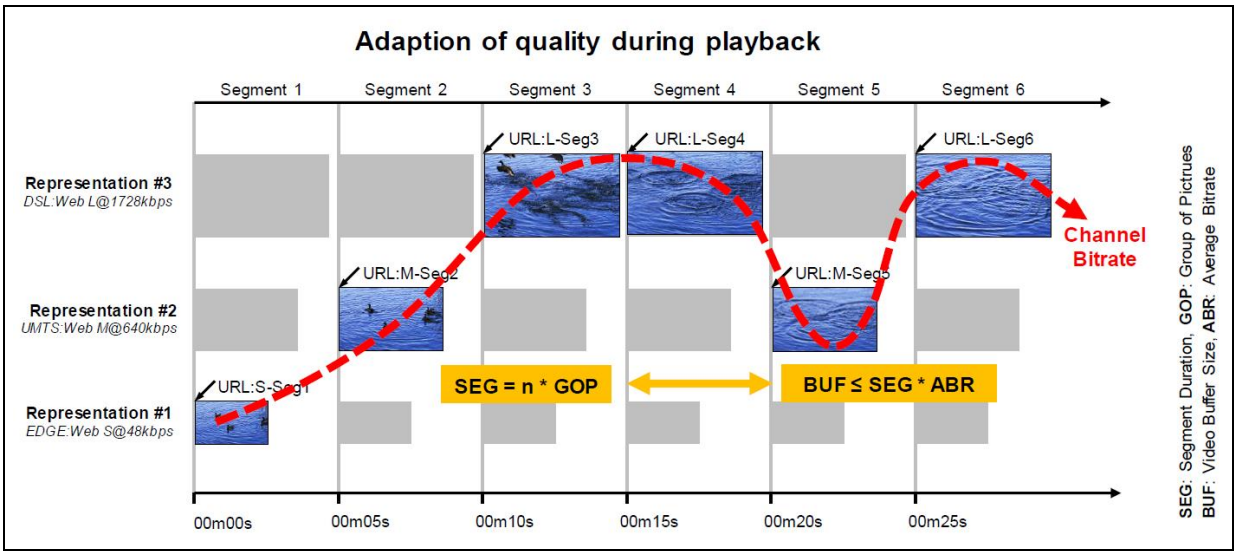

Figure 10: Example of quality switching through MPEG-DASH [32]

In figure 10, we stream a video starting with segment 1 in the lowest representation. The video player realizes that it needed only half of the playback time of the first segment to load it completely. Thus, it switches to a higher quality. Later, the video player cannot obtain the fourth segment completely in time, so it switches back down to a lower quality.

We put an example MPD file in the appendix section, which lists all concepts that we introduced in this chapter.

4.3.1 Compatibility

In the following, we list all results regarding compatibility of "MPEG-DASH".

- Being HTTP-based, this protocol does not limit video codecs, as well.
- Since MPEG-DASH was declared as video streaming standard, more and more video players implemented the protocol. But still, many players do not understand the protocol, which can be explained by the many requirements to the video player, when using the protocol: a video player needs to interpret the "MPD" file, it needs to know at least one selection mechanism and, depending on the number of segments, it needs to switch the requested resources frequently [17].

4.3.2 Usability

In the following, we list all results regarding usability of "MPEG-DASH".

- Many video platforms have adopted "MPEG-DASH" as their primary protocol, making it a rising standard for video streaming [20].
- Being HTTP-based, no firewall problems exist.
- Seeking operations are possible the same way as for "HTTP Pseudo Streaming", when a "MOOV Atom" is included at the beginning of a video file [30].
- The protocol does not store any video segments on its local file system. Instead, the same principles as for "HTTP Pseudo Streaming" apply.
- It requires a lot of effort for a streaming server to be capable of streaming via this protocol. A video file must be provided in different qualities and an "MPD" file needs to be generated, as well. Either this gets done for each video file previously to streaming requests, resulting in duplicate storage of multiple video qualities, or a streaming server must transcode a lower video quality in real-time, making the server application way more cost- and resource-intensive.
- The automatic quality switching can be irritating to end-users. A sudden loss of video quality due to bandwidth problems might not always be desirable. Instead, waiting a longer time for buffering to finish, might be the better solution. A user must manually select a higher quality again, if a loss of quality is not acceptable.

4.3.3 Performance

In the following, we list all results regarding performance of "MPEG-DASH".

- In general, the protocol can utilize bandwidth limitations better than RTSP, resulting in less dropped frames and distorted images. [29].
- Switching between video qualities automatically ensures the best utilization of available bandwidth and results in the smoothest possible video playback of all available streaming protocols. Buffering pauses occur rarely when using this protocol.
- This protocol diminishes the waste of bandwidth even more compared to "HTTP Pseudo Streaming". When multiple small segments of a video file are provided, a video player has no possibility of requesting too large byte ranges of a file, which could get discarded by not being watched. This eliminates the bandwidth problem present in "HTTP Pseudo Streaming".

4.4 Recommendation

In table 6 we rate the protocols regarding their fulfillment of the three proposed categories in the three levels "bad", "average" and "good".

	RTSP	HTTP Pseudo Streaming	MPEG-DASH
Compatibility	bad	good	good
Usability	bad	good	average
Performance	bad	average	good

Table 6: Comparison of video streaming protocols

RTSP does not satisfy any category in an adequate degree. Therefore, it is not recommended.

We consider "HTTP Pseudo Streaming" to be more usable than "MPEG-DASH", due to the large effort needed for servers and video players to support "MPEG-DASH" and the possibility of getting confused by automatic quality changes. Then again, "MPEG-DASH" deals with bandwidth limitations to a better degree than "HTTP Pseudo Streaming" does. Both protocols do not limit video codec support and are implemented in the vast majority of video players.

We believe, that it is easier to implement "HTTP Pseudo Streaming" as first streaming protocol in an actual video application. It provides all necessary features to the end user and can

still be configured manually to use a lower quality, if buffering happens too often. Therefore, we recommend it as first protocol to use.

When the infrastructure for a video application is up and running, we anticipate that adding "MPEG-DASH" step-by-step can bring additional benefits and should be considered, since it is becoming the standard in many video platforms. We recommend a combination of an "automatic" quality option and the possibility to select manual quality levels, as currently deployed in YouTube, when adding "MPEG-DASH" support.

5 Comparison of Video Player Frameworks

In this chapter, we compare three video player frameworks against different categories. We conclude with a recommendation of one video player framework suited for use in our video player application.

The categories of comparison are:

- **Compatibility to the existing "Image Viewer" architecture:** According to the requirements introduced in chapter three, the video player shall be opened from within the "Image Viewer". Therefore, custom communication between the "Image Viewer" and the video player framework must be possible. It is not desirable to show videos in a standalone browser application, because then a user would be confronted with a complete different user experience and environment. Instead, a video player framework must be integrated into a Java application.
- **Ability to use different video codecs and video streaming protocols:** A video player framework should support as many video codecs as possible. Because we do not know, whether we will use a specialized video codec at some time in the future, this ensures maximum compatibility to any video that we receive. Furthermore, a video player framework must at least be capable of playing videos through "HTTP Pseudo Streaming", with "MPEG-DASH" being desirable, as well.
- **Ability to implement different annotation features:** All annotation features introduced in chapter 3.1 "Functional requirements" need to be implemented. The easier this is possible with a specific video player framework the better. Expandability is also a key factor in this category, as new requirements might arise in the future.

In the following we compare the video player frameworks "VLCJ", "JavaFX" and "JxBrowser". All of them can be integrated in Java applications, because they are either based on Java completely or offer a JNA (Java Native Access) integration of native libraries. We do not consider pure web-based video player frameworks in a standalone browser application, like "dash.js" due to the downsides mentioned in the first category. Furthermore, we do not consider the Java-based video player framework "JMF" (Java Media Framework) because it is outdated and deprecated by its modern alternative "JavaFX".

5.1 VLCJ

VLCJ is a Java-based wrapper around the libraries used by “VLC Media Player”. These libraries, as well as its famous video player, are developed by the non-profit organization “VideoLAN” and are open-sourced. VLCJ developed a JNA-API to access the native libraries of “VLC Media Player”, which are called “LibVLC”. The Java-Wrapper can utilize nearly all features provided by the native libraries through that approach [33].

In addition to an API speaking directly to the native libraries, VLCJ provides multiple custom APIs. These provide a higher-level abstraction to access basic video player functionalities like loading a video file or seeking a time-based value. A custom Java-based video player can be developed easily with these high-level APIs. VLCJ provides a demo implementation of a custom video player that utilizes the provided APIs. In figure 11 we show that demo implementation [33].

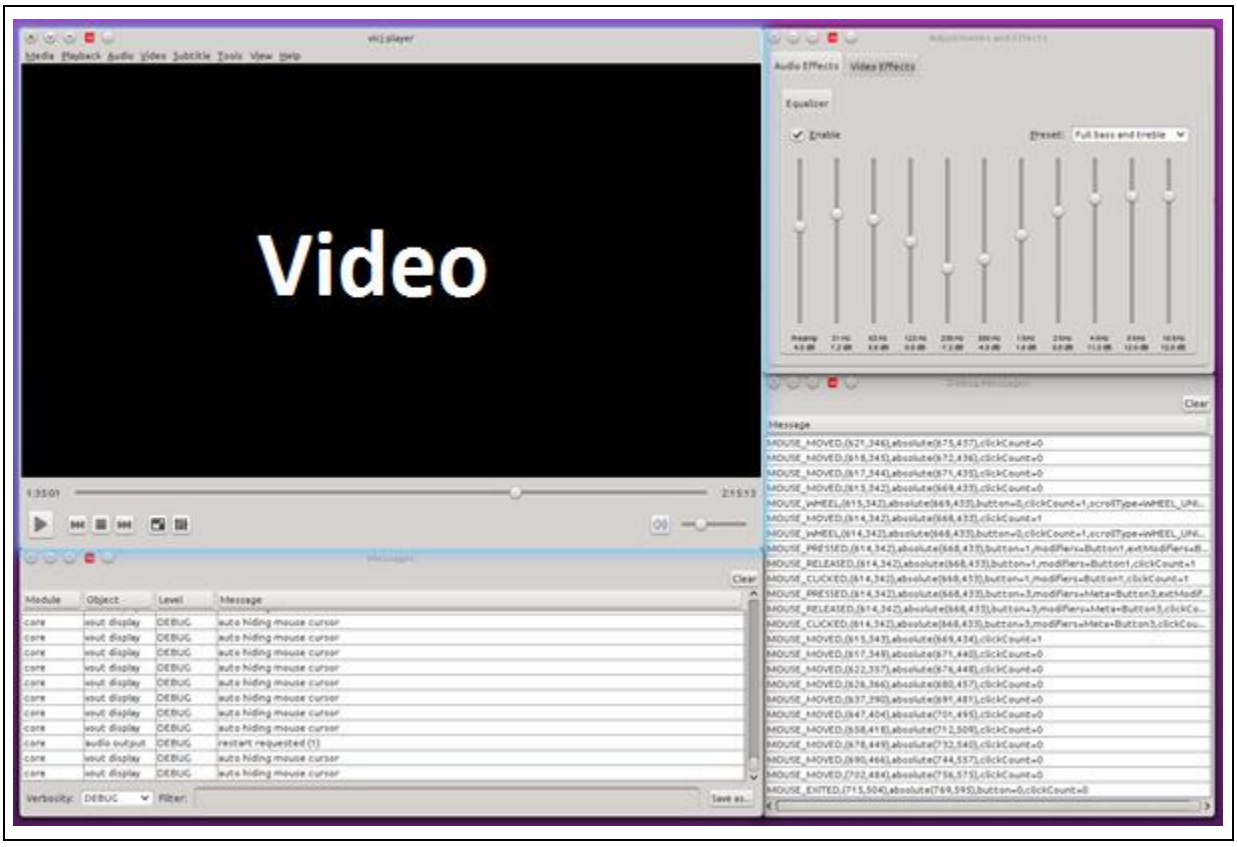

Figure 11: VLCJ‘s demo video player [33]

Many features of the native “VLC Media Player” are implemented in the demo video player, as well.

5.1.1 Compatibility

In the following, we list all results regarding compatibility of "VLCJ".

- VLCJ provides a video player component, that can be embedded into a "Java AWT Window", a "Java-Swing JFrame" and a "JavaFX" application [33]. Thus, it supports all major Java UI component frameworks. As a result, we can implement our own Java-Swing video player UI with an integrated video player component showing the actual video content. This enables us to create an application that looks and feels just like the existing "Image Viewer" application.
- We can implement all required features of our video player application, like a "play/pause" button or a volume slider, by creating our own Java-Swing toolbar with buttons and sliders. These UI elements can call a native function provided by the VLCJ wrapper API that performs the required action. Thus, we can create all controls to look and feel like the "Image Viewer" application.

5.1.2 Support of Codecs and Streaming Protocols

In the following, we list all results regarding support of video codecs and video streaming protocols of "VLCJ".

- Being based on VLC's native libraries, VLCJ supports many video codecs. In fact, the "VLC Media Player" is known for its ability to play "every" video codec and offers support for all popular and state-of-the-art video codecs, as well as less popular ones [34]. Once "LibVLC" supports a new video codec, the changed or additional native libraries can simply be updated in the library package used by VLCJ. Thus, video codec support is of no concern when using VLCJ.
- VLCJ supports the HTTP protocol, as well as the MPEG-DASH protocol. Therefore, the two streaming protocols that we recommended in the previous chapter are both supported by this video player framework.

5.1.3 Annotation Features Implementation

In the following, we list all results regarding the implementation of annotation features with "VLCJ".

- The native libraries provide many features that we need for our video player application. Among those are the ability to change brightness, contrast and saturation, a zoom function and the possibility to take snapshots. The VLCJ API provides meaningful methods to access these functionalities. More basic features like changing playback speed, volume control or frame-by-frame playback are offered by the VLCJ API, as well. Thus, many of the required features are available out-of-the-box with VLCJ [33].
- When a feature is not available through API calls to the native libraries, VLCJ offers a way to implement a custom feature, too. By default, the native libraries perform the video decoding process by utilizing fast hardware decoding methods. That approach provides only rendered images to the Java-based video player component. Another available option is a software-based image rendering process. VLCJ implements a second video player component, which receives each frame from the native libraries as byte array instead of a completely rendered image. The video player component needs to render the image itself when using this method. This approach, while requiring more effort, gives us the possibility to manipulate each frame on a byte-level abstraction. This allows us to implement any annotation feature, that might come up and is not supported through native library calls directly [33].

5.2 JavaFX

JavaFX is a purely Java-based graphics framework and the successor of Java-Swing [35]. It offers its own implementation of a video player through the “MediaPlayer” class [36]. We need no additional native libraries with this approach, as all relevant files and programs, like decoders, are distributed with Java itself and the JavaFX package, respectively. Like with VLCJ, we can develop a custom video player by creating a JavaFX application that uses the “MediaPlayer” class. In figure 12, we show a sample video player application developed with JavaFX.

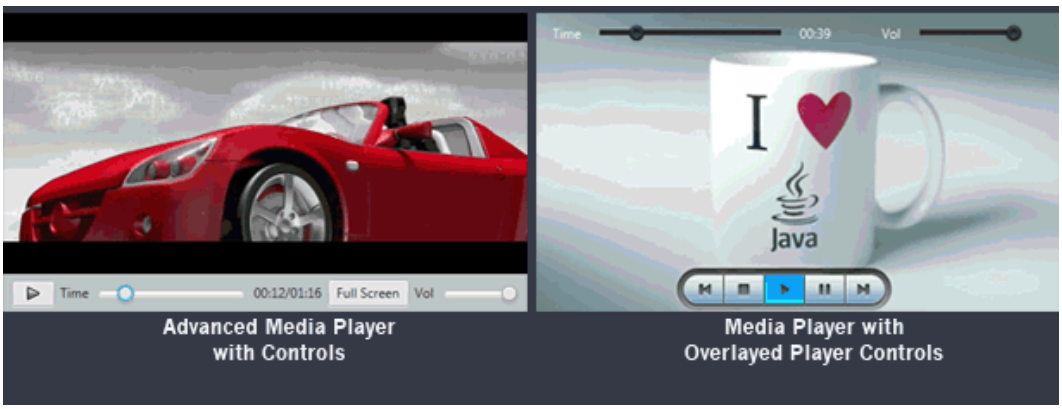

Figure 12: Demo video player application with JavaFX [37]

The figure shows, that JavaFX is also highly customizable regarding custom controls and UI-design of a video player.

5.2.1 Compatibility

In the following, we list all results regarding compatibility of "JavaFX".

- We can develop a video player with this framework by only using JavaFX and its provided UI functionalities. Therefore, the video player can be easily integrated into the existing "Image Viewer" application, as both are Java-based and because Java-Swing and JavaFX are compatible.
- When comparing figure 11 with the "Image Viewer" that we presented in figure 2, we can quickly discover differences in the "Look and Feel" of both UIs. These differences come from the usage of the two different UI frameworks, Java-Swing on the one hand and JavaFX on the other hand. It would be very time-intensive and nearly impossible to make JavaFX look like Java-Swing.

5.2.2 Support of Codecs and Streaming Protocols

In the following, we list all results regarding support of video codecs and video streaming protocols of "JavaFX".

- "JavaFX" has limited support for different video codecs. It only supports "H.264" and "VP6", an ancestor of the modern codec "VP9", among some other less popular codecs [36].
- "JavaFX" supports the HTTP protocol, but not "MPEG-DASH" as streaming protocols. Instead of "MPEG-DASH", it provides an implementation of the "HTTP Live

Streaming" (HLS) protocol, which is an older and non-standard adaptive streaming protocol [36].

5.2.3 Annotation Features Implementation

In the following, we list all results regarding the implementation of annotation features with "JavaFX".

- The "MediaPlayer" class does not provide required annotation features, like zooming or taking a snapshot, directly. However, being a pure JavaFX application, many possibilities exist to implement such features manually. Like VLCJ, a JavaFX-based video player has access to the underlying data either as rendered image or as raw byte array and can therefore manipulate the data accordingly to implement required features like brightness or contrast adjustments [36].

5.3 JxBrowser

"JxBrowser" is a Chromium-based browser that offers integration methods into Java-Swing or JavaFX applications [38]. Chromium is the underlying open-source engine used by the "Google Chrome" browser to process, render and display HTML content [39]. "JxBrowser" enables a Java-based application to display modern HTML5 web-applications. This allows us to use HTML as our video player framework, which will be rendered by "JxBrowser" [40]. The following code example shows an integration of "JxBrowser" into a simple Java-Swing application.

```
Browser browser = new Browser();
BrowserView view = new BrowserView(browser);
JFrame frame = new JFrame();

frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
frame.add(view, BorderLayout.CENTER);
frame.setSize(700, 500);
frame.setLocationRelativeTo(null);
frame.setVisible(true);

browser.loadURL("http://www.video-test.org");
```

Code 3: JxBrowser Video example [40]

We can add the browser as Java-Swing component to our application. Then we can load a website by specifying its URL. If the website offers video content through basic HTML, the browser component will display that video.

5.3.1 Compatibility

In the following, we list all results regarding compatibility of "JxBrowser".

- Due to its easy integration into a Java-Swing application, as seen in code 3, the video player framework "JxBrowser" is compatible to our "Image Viewer" application. The framework allows us to start the video player application from within the "Image Viewer" by loading a specific URL. Furthermore, the video player appears in a dedicated Java-Swing application window, instead of being opened in regular standalone browser application.
- We need to define the actual video player application through a web-application based on HTML and JavaScript. We only visit that web-page with "JxBrowser". Therefore, the "Look and Feel" of the video player application is likely to differ from the "Image Viewer" application.

5.3.2 Support of Codecs and Streaming Protocols

In the following, we list all results regarding support of video codecs and video streaming protocols of "JxBrowser".

- "JxBrowser" limits the supported video codecs due to its usage of "Chromium". The "Chromium" browser engine does not support the video codecs "H.264" and "H.265" out-of-the-box due to the licensing model of these codecs. We need to obtain a license for these codecs first and then contact the team behind "JxBrowser" to provide us with a custom build that supports these codecs, as well [41].
- HTML5 provides a "video" tag that can render videos referenced through HTTP [42]. Therefore, it supports the video streaming protocol "HTTP Pseudo Streaming". Furthermore, multiple JavaScript libraries exist, that implement the "MPEG-DASH" protocol, for example "dash.js" [43]. When a website provides such a JavaScript library, this protocol is supported, as well.

5.3.3 Annotation Features Implementation

In the following, we list all results regarding the implementation of annotation features with "JxBrowser".

- "JxBrowser" supports all basic player-functionalities, like skipping to a different time, brightness and contrast adjustments and many more, through its usage of the "Chromium" engine and the resulting possibility of HTML and JavaScript rendering. It is up to the web-application to provide a video player that supports these functionalities. Many JavaScript frameworks exist, that provide advanced video players, for example "dash.js" [43]. Therefore, support of basic annotation features is of no concern with this approach.
- As soon as communication between the "Image Viewer" application and the video player application occurs, it is up to "JxBrowser" whether a specific feature is supported. The ability to take snapshots and to transfer them into the "Image Viewer" application is one example of such a communication. "JxBrowser" offers an option to take screenshots of a website [44]. However, extensibility of features that fall into this category is less likely.

5.4 Recommendation

In table 7, we rate the video player frameworks regarding their fulfillment of the three proposed categories in the three levels "bad", "average" and "good".

	VLCJ	JavaFX	JxBrowser
Compatibility	good	average	average
Protocol & Codec Support	good	bad	average
Annotation Features	good	average	good

Table 7: Comparison of video player frameworks

"VLCJ" offers the best compatibility due to its possibility to "look and feel" like the "Image Viewer" application. It allows us to create a toolbar with basic Java-Swing UI elements and to integrate the video player component into a Java-Swing window, as well. On the other side,

"JavaFX" and "JxBrowser", while being easily integrated into Java-Swing applications, both "look and feel" differently compared to the "Image Viewer" application. Buttons and sliders on toolbars used in these frameworks will never look like Java-Swing UI elements, but rather like typical JavaFX- or HTML-UI elements.

"VLCJ" offers the best support of video codecs and video streaming protocols, as well. It supports nearly any video codec with a certainty to be compatible to newer video codecs in the future due to its native libraries. Furthermore, it supports both required video streaming protocols. "JavaFX" does not support any modern video codec and does not understand the "MPEG-DASH" protocol, thus failing this category. "JxBrowser" utilizes modern technologies through the usage of the "Chromium" web-engine. It is equally certain, that the "Chromium" project supports upcoming open-source video codecs and video streaming protocols in the future. However, it does not support the most used video codec "H.264" and its successor "H.265" out-of-the-box.

Finally, "VLCJ" provides good support for our required annotation features. It already implements many functionalities and offers a good way to extend on the provided functionalities through its APIs. We can extend "JavaFX" to support our annotation features, as well. However, it supports few functionalities out-of-the-box, resulting in a worse rating. Annotation features will be of no concern with the usage of "JxBrowser" due to its modern and state-of-the-art support of HTML5 and JavaScript. There are already many frameworks available that offer implementations of advanced video player functionalities.

To summarize it, we conclude that "VLCJ" is the best choice for our video player application. It fulfills all categories to a satisfying degree and is easy to use. We therefore recommend it as video player framework for usage in our application.

6 Prototype Development

We describe the development of a prototype video player application in this chapter. A video player client and a corresponding video streaming server communicate in this prototype. Furthermore, we implement a basic integration of the video player client into the “Image Viewer” application. Our prototype is based on the two technologies that we recommended in the previous chapters: "HTTP Pseudo Streaming” as main video streaming protocol, “MPEG-DASH” as a second option and “VLCJ” as video player framework.

6.1 Architecture

We introduce the general architecture of the whole prototype in this chapter. We give a more detailed look into each component later in the corresponding chapters.

We divide the prototype into two main components: the case worker’s PC that acts as client and where videos can be watched and the video streaming server that stores all videos and provides access through a defined streaming endpoint. Figure 13 illustrates this architecture.

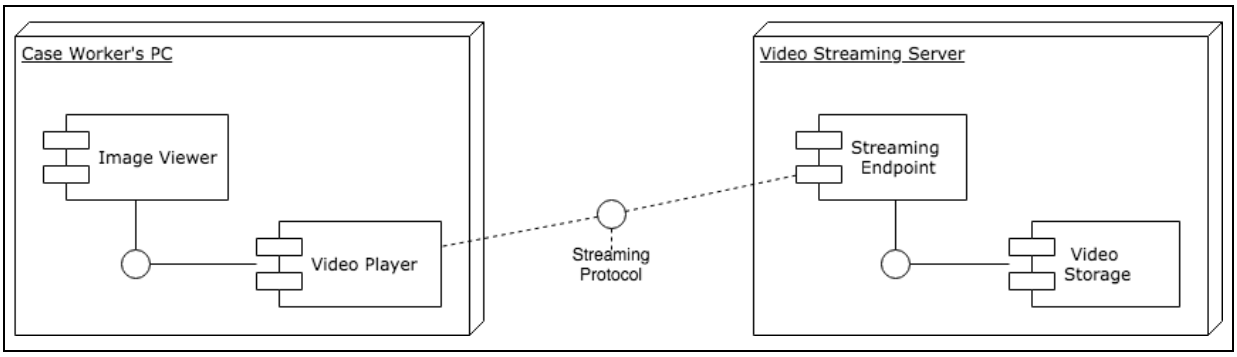

Figure 13: Prototype Component Diagram

Communication occurs between all four modules. The video player and the streaming endpoint communicate via one of the two chosen video streaming protocols, either “HTTP Pseudo Streaming” or “MPEG-DASH”. We transfer video data through a network in this path. A Java interface defines the communication between the video player and the “Image Viewer”, which happens in the same JVM. The streaming endpoint references a requested video file on its own file storage instead of accessing a database.

6.2 Server

We explore the video streaming server component with more detail in this section. The component is deployable as a single `jar` file combined with a folder, which contains sample video files, on any server system, for example on a Linux server. The following code snippet illustrates the structure of the component.

```
%PROTOTYPE_HOME%
├── streaming_server.jar
├── run_server.sh
└── videos/
    ├── sample1.mp4
    ├── sample2.mp4
    └── dash/
        ├── sample1.mpd
        ├── sample1_seg1.m4s
        └── sample1_seg2.m4s
```

Code 4: Prototype's server component directory structure

As illustrated in code 4, the `jar` file and the corresponding shell script to start the component are in the same directory as a `videos` directory, which contains sample video files that can be streamed by this prototype component.

To start the component, we can execute the provided shell script. It contains the following command, which will start the streaming server as background process:

```
java -jar streaming_server.jar &
```

Code 5: Start command for the prototype's server component

After successfully starting the server component via the command in code 5, it runs as single process on the server. It can receive multiple, parallel streaming requests through the defined streaming endpoint and will answer with the requested resource from the `videos` directory or with a "Not Found" error message, if the video is not found in the `videos` directory.

6.2.1 Streaming Endpoint

The "streaming endpoint" module defines the URLs and HTTP methods that need to be called by a client to stream a video file. Thus, it acts like a normal webserver. We use "Spring Boot" with an embedded "Apache Tomcat" webserver as main framework to host the streaming

endpoint [45]. The streaming endpoint provides multiple URLs to receive streaming requests. Each endpoint provides a slightly different implementation of a streaming protocol. We chose this approach to test different possibilities to implement streaming protocols. These implementations are accessible under the following URLs:

Protocol	URL Definition and Example	Description
HTTP Pseudo Streaming	`/videos/stream/<id>` `/videos/stream/1`	Provides files that are named `video<id>.mp4`. This enables us to issue test requests easily and fast.
HTTP Pseudo Streaming	`/videos/stream?name=` `/videos/stream?name=sample.mp4`	Provides any file that the streaming endpoint has stored. The client must know the exact name.
MPEG-DASH and HTTP Pseudo Streaming	`/videos/stream2/<path>` `/videos/stream2/dash/sample1.mpd`	Processes whole paths and tries to find the file relative to the `videos` directory. This allows clients to stream via MPEG-DASH, as well.
HTTP Pseudo Streaming	`/videos/stream3/<name>` `/videos/stream3/sample.mp4`	A different implementation of the HTTP Pseudo Streaming protocol that also processes file names.
HTTP Pseudo Streaming	`/videos/stream?c=&q=&d=&buf=&bandwidth=` `/videos/stream?c=h264&q=high&d=short&buf=20480&bandwidth=10`	Sophisticated implementation that allows to choose codecs, qualities, bandwidth in Mbps and buffer size of either a short or a long video sample.

Table 8: Prototype Streaming Endpoint URLs

As seen in table 8, we implement three different versions of the HTTP Pseudo Streaming protocol, that are available under `/videos/stream`, `/videos/stream2` and `/videos/stream3`.

The first one is the main implementation that proved to work best. It is based on an open-source implementation of partial content requests [46]. This allows us to modify the code that streams video files at any time, as we have full control over it.

We developed the second implementation from scratch as well to process partial content requests. This implementation proved less reliable than the first implementation by causing sudden stops or stutters in video playback on the client side. We developed support for MPEG-DASH in this implementation, because we could develop the required file path resolution with the least effort. We could transfer this support to the main implementation without any problems.

The third implementation is based on the "Spring Content" framework completely. We do not have to write and maintain any code in this implementation, but rather provide the framework only with a root path that hosts all video files. This implementation allows us to only stream video files from a local file storage, which is why it is no suitable solution for an actual video streaming server.

We put pseudo code of the implementations in the appendix section. The complete codebase is too large to be put in this document.

6.2.2 Video Storage

As already mentioned in chapter 1.3 "Scope of Work", it is no task in this bachelor's thesis to create a database solution for storing video files. Instead, we developed our streaming server to search for sample video files on its local file system, as mentioned in the directory structure in code 4.

A productive solution for video storage needs to be connected to our existing DMS for storage of index information for each video file. This index information contains metadata like the associated insurance id or liability case, for which the video is provided. Furthermore, a video storage must provide ways to read the video file in parts, rather than completely. This is required due to the principle of partial content requests used by "HTTP Pseudo Streaming" and MPEG-DASH.

Our proprietary solution for video storage provides complete video files in the `videos` directory. Furthermore, we placed segmented video files and a corresponding MPD file suitable for streaming via the MPEG-DASH protocol in the `videos/dash` directory. Code 4 illustrates this structure.

6.3 Client

We explore the video player component and its integration into the "Image Viewer" application with more detail in this chapter. The following white-box diagram of the video player component introduces its general architecture.

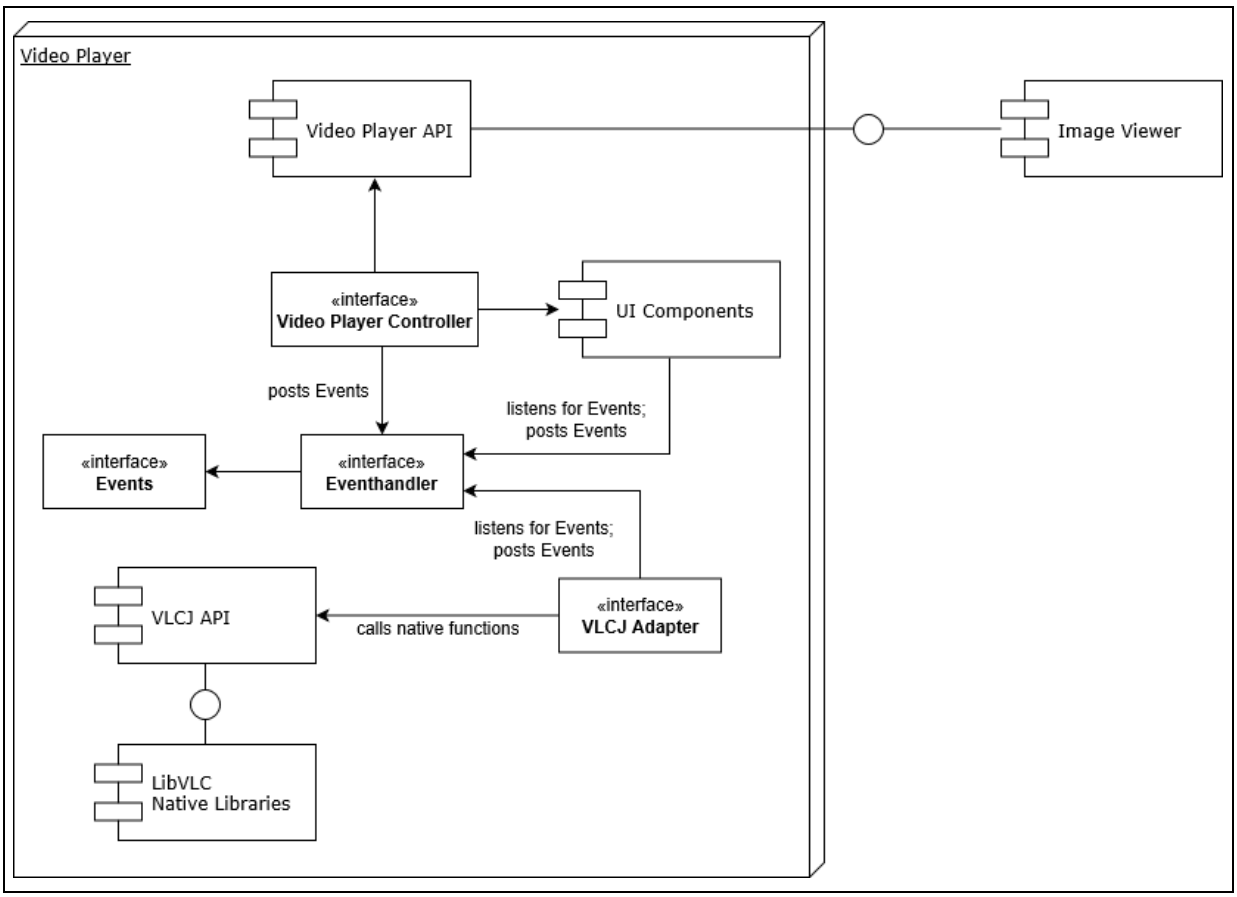

Figure 14: Video Player Component Diagram

As shown in figure 14, the video player component can be grouped in several layers. The bottom layer consists of the native libraries provided by "LibVLC". These are used and adapted by VLCJ, which provides its own APIs to communicate with the native libraries.

The next layer includes the whole implementation of the video player component. At its core, we chose an event-driven approach to distribute and execute events of any kind. This design

is adapted from the VLCJ demo player [33]. We list all events in the appendix section. This provides a good overview of all actions that can be performed. Every event contains information that is provided by some component and gets distributed to all components that are interested in receiving such an event. The receiver either uses the information to perform a specific action (e.g. change the volume to the specified value) or to indicate the information visually to the user (e.g. update the current playback time on a time slider). The `EventHandler` interface distributes all events to previously registered components.

The `VLCJAdapter` listens on all Events that request an action to be performed by the native libraries or by VLCJ. Pausing or resuming playback, taking a snapshot or changing the brightness value are some examples of such actions. The `VLCJAdapter` is the central interface to the VLCJ API and provides ways to handle these Events for all components. Hence, we need to deal with VLCJ APIs only at one place, namely at this interface.

The `VideoPlayerController` implements our own "Video Player API" that we introduce in a subsequent chapter. This component executes general commands to start or change video playback. Loading a new video or changing its quality are examples of such commands.

Multiple UI components exist that create parts of the whole video player. A subsequent chapter will introduce the general architecture of these components.

The already mentioned "Video Player API" is the top layer of the whole component. It is the sole API that is exported and therefore used by the "Image Viewer" application.

6.3.1 Video Player API

This chapter introduces the "Video Player API" that we developed. It defines the usage of the component by other applications, for example by the "Image Viewer". Figure 15 visualizes the API design.

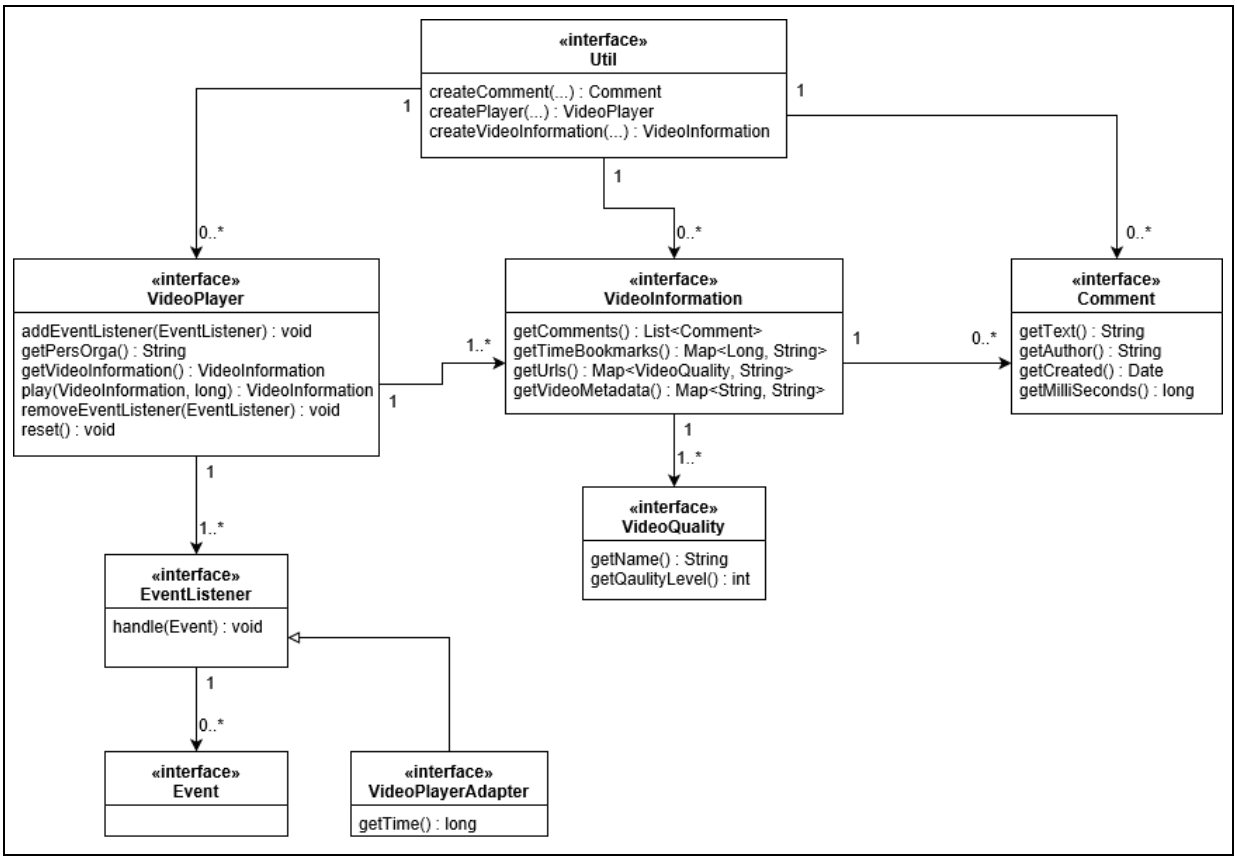

Figure 15: Video Player API

The `VideoPlayer` interface is the main contact point for external users. It provides a `play` method that allows to play a new video in the component. Furthermore, applications can register themselves as `EventListeners` to receive notifications about new or deleted `Comments` or `Snapshots`.

The `VideoInformation` interface is the data structure that holds all necessary information about a video needed by the `VideoPlayer` to load it. The following code sample shows such a data structure in a JSON format.

```
{
"video.qualities" : {
   "HIGH" : "/videos/stream?c=h264&q=high&d=short",
   "MEDIUM" : "/videos/stream?c=h264&q=medium&d=short",
   "LOW" : "/videos/stream?c=h264&q=low&d=short"
},
"video.markers" : [{"startTime" : 2000, "comment":"Wichtig"}],
"video.comments" :
  [
    {
```

```
      "time"   : 1000,
      "text"   : "hier passiert ein Unfall",
      "author" : "%PERS_ORGA_SAMPLE%",
      "created": "2018-11-15-19:14:30.123"
    },
    {
      "time"   : 2000,
      "text"   : "hier auch",
      "author" : "%PERS_ORGA_SAMPLE%",
      "created": "2018-11-15-19:33:30.123"
    }
  ],
"video.metaData" : {
    "VNR" : "123456789X",
    "Eingangsdatum" : "2018-11-01-00:00:00.000"
}
}
```

Code 6: Sample `VideoInformation` data structure

Code 6 shows that the `VideoInformation` data structure contains the following four parts. In `video.qualities` we provide URL paths to every available video quality. This required object allows the video player to start streaming a video. In the optional object `video.markers` we provide additional start times from which the video can be played back. All times when snapshots are taken are examples of such times. In the optional object `video.comments` we provide all available comments for the video. Finally, in the optional object `video.metaData` we provide additional information of any kind, for example an insurance id.

A user of our Video Player API can simply create a `VideoPlayer` by using the `Util` class.

Code 7 shows an example of creating a `VideoPlayer` through this approach.

```
VideoPlayer videoPlayer = Util.createPlayer("PERS-ORGA");
```

Code 7: Creating of a `VideoPlayer` object

The method called in code 7 searches for available implementations of the `VideoPlayer` interface in its classpath and returns the first available one. We created a `VLCJVideoPlayer` implementation that is based on the VLCJ framework.

6.3.2 UI Components

This chapter introduces the UI of the video player client. We developed the UI with Java Swing to meet the requirement to “look and feel” like the “Image Viewer” application. Figure 16 illustrates the complete video player UI.

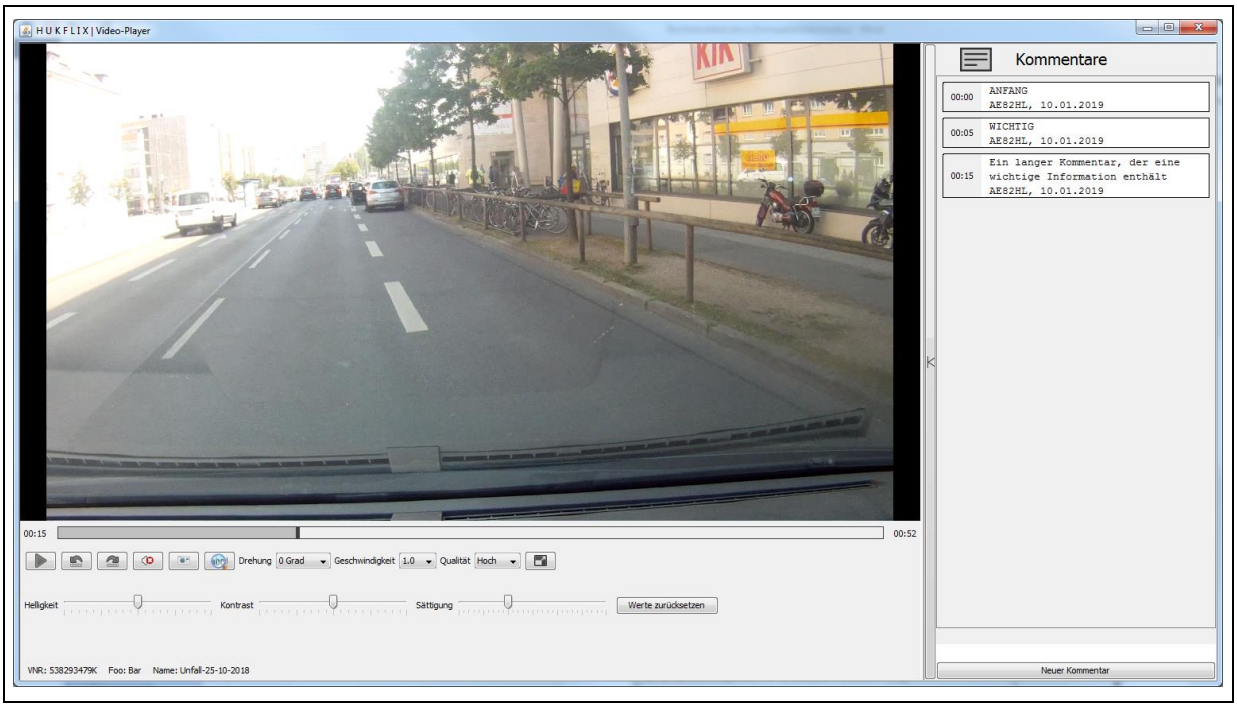

Figure 16: UI of the video player client

The UI consists of several smaller components that are grouped together. We show the smaller components in figure 17.

The first component is the video component itself. It is based on the implementation provided by VLCJ and shows the actual video. When resizing the video player window, this component might add black borders to the left and right or to the top and bottom of the component, depending on the aspect ratio of the video file. We offer no possibilities to change the aspect ratio of the video.

The second component contains all comments that a case worker wrote. The comment feed is collapsible completely, thus making the whole video player more compact if desired. Each comment highlights the time of the video that it references. The video will jump to that time when clicking on a comment.

The third component contains all controls and status information of the video player. Among these are labels that show the elapsed and total time and a time bar. All required controls are

grouped in a toolbar underneath that: a “play/pause” button, “skip forwards” and “skip backwards” buttons, a “volume control” button and slider, “snapshot” and “zoom-lens” buttons, “rotation”, “playback speed” and “video quality” dropdown menus and a “full screen” button. Sliders and a reset button to control the brightness, contrast and saturation of the video are provided underneath the toolbar for those general controls. Finally, a status bar visualizes all metadata information that are received in the `VideoInformation` data structure.

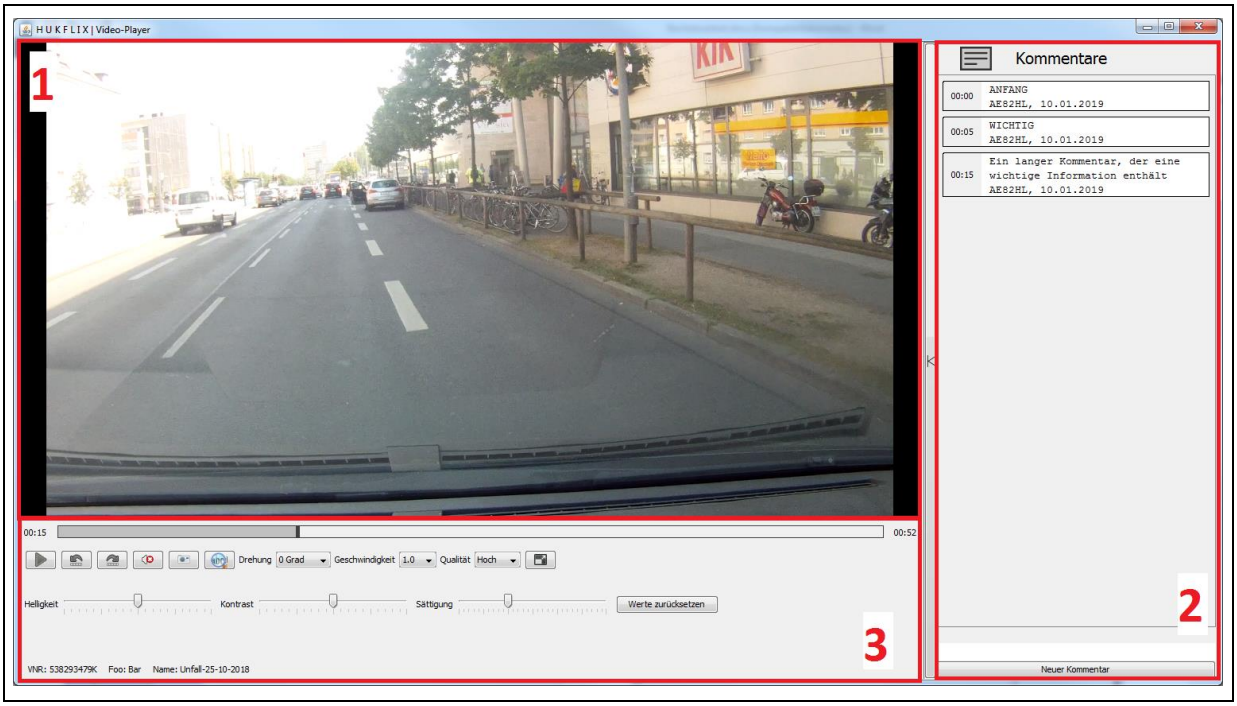

Figure 17: UI components of the video player

We created each component separately and bounded them together in a window component. We do not provide code samples of the Java Swing components in this document, because they would be too large.

6.3.3 Annotation Feature Implementation

This chapter gives insight in some noteworthy implementations of features and controls in the video player client.

6.3.3.1 Zoom-Lens Implementation

LibVLC provides a zoom-lens functionality in its native libraries. However, the VLCJ API does not implement a Java method to access this feature directly. As already mentioned in the

comparison of the VLCJ framework, we can implement our own features, if the VLCJ API does not provide methods for them.

A way to access the functions from LibVLC directly is to use the command-line arguments that are provided by LibVLC [47]. Code 8 shows the command-line argument to start the zoom-lens feature directly.

```
--video-filter=magnify
```

Code 8: Command-Line argument to start the zoom-lens feature

We then only need to provide this command-line argument to LibVLC when instantiating the native libraries. Code 9 shows this step.

```
var f = new MediaPlayerFactory("—-video-filter=magnify");
```

Code 9: Instantiate LibVLC with command-line arguments

The `MediaPlayerFactory` by VLCJ forwards custom command-line arguments directly to the native libraries of LibVLC. Therefore, when using this option, we can use the zoom-lens functionality.

Figure 18 shows the zoom-lens feature activated in the video player client.

Figure 18: Zoom-Lens feature of the video player client

The whole video gets displayed in the upper-left corner of figure 17 along with a rectangle that shows the current selection that is zoomed into. The main component shows that rectangle zoomed in. A user can change the zoom-level and position of the selection by moving the slider or rectangle.

6.3.3.2 Hardware-based vs. Software-based Rendering

VLCJ implements two possibilities to render the video frames. Either we use the `EmbeddedMediaPlayer` component, which leaves rendering completely to LibVLC and only receives the finished images. Or we use the `DirectMediaPlayer` component, which provides us with a byte-array of each video frame and leaves rendering an image out of that information to us.

The first approach is usually faster and requires less computing power because LibVLC utilizes hardware-based rendering mechanisms to render the video frames, whereas rendering out of a Java program is completely software-based and cannot utilize those mechanisms.

Besides the worse performance, we need to perform additional work to identify the video's "chroma subsampling scheme" when using software-based rendering. This scheme defines what "color space" is used by the video codec in each video frame. For example, a video frame can be defined in an RGB-based, YUV-based or grayscale color space. This information is required for interpretation of the byte-array as pixels with respective colors [23, p. 420ff.]. While many software-based rendering frameworks can determine the correct color space out-of-the-box, we still need to choose such a suited framework for usage in our Java application. When using the hardware-based rendering approach, LibVLC determines the used color space itself, thus eliminating the additional requirement.

To summarize, we should always use the hardware-based rendering option when possible. Only when a specific feature is neither available through VLCJ or through LibVLC, we need to use the software-based rendering approach.

VLCJ offers methods to change the brightness, saturation and contrast values of the video. Therefore, the requirement to use software-based rendering does currently not exist, because we can implement all features with hardware-based methods. Code 10 illustrates the commands that will change the brightness, saturation and contrast of the video.

```
MediaPlayer m = new EmbeddedMediaPlayer();
m.setAdjustVideo(true); // enable image processing features
```

```
m.setBrightness(1.5f); // between 0.0f and 2.0f. Default 1.0f
m.setContrast(0.5f); // between 0.0f and 2.0f. Default 1.0f
m.setSaturation(2.5f); between 0.0f and 3.0f. Default 1.0f
```

Code 10: Commands to change brightness, contrast and saturation of a video

6.3.3.3 Limitations

There exist some limitations in the combination of specific features. This chapter lists all limitations that we currently identified.

1. It is not possible to use the zoom-lens and to rotate the video by arbitrary angles. LibVLC uses the `--video-filter` command-line argument for both features. It is not possible to combine multiple video filter options [47].
2. It is not possible to use image processing features like contrast or brightness changes and to take a snapshot of such videos. LibVLC crashes when performing this action. We also observed this behavior in the "VLC Media Player" directly, thus signaling that this is a bug or restriction in LibVLC itself. As a result, we always deactivate image processing features automatically, when taking a snapshot.

6.3.3.4 LibVLC Plugins Cache

The native libraries reside in a `plugins` folder that is referenced by VLCJ when creating a video player instance. In this folder, a file `plugins.dat` exists that holds metadata information about the location of the native libraries. When moving the whole LibVLC folder to another destination, we invalidate the metadata information in this file. As a result, warning messages are printed to the console each time when creating a new video player instance. Code 11 shows an example warning message.

```
[00000000002de430] main libvlc error: stale plugins cache:
modified ...\libVLC\plugins\visualization\libvisual_plugin.dll
```

Code 11: Plugins-Cache example error message

We can remove these messages by generating a new `plugins.dat` file at the final destination of the LibVLC folder. Code 12 shows the required command to generate this file.

```
vlc-cache-gen.exe path\to\libVLC\plugins
```

Code 12: Generate a new Plugins-Cache

The binary executable `vlc-cache-gen.exe` that is used in code 12 is provided when installing “VLC Media Player” and LibVLC.

6.3.3.5 Buffer Overlay

When streaming a video, we load the file in chunks as already described in the chapters 4.2 and 4.3 regarding the protocols „HTTP Pseudo Streaming“ and „MPEG-DASH“. It can happen that the video player runs out of available chunks, thus resulting in a playback stop.

In such an event, we want to represent the user with a meaningful estimation of how long it will take until playback will restart again. LibVLC emits events containing the percentage of how many data is already buffered for the first chunk after playback stop. We describe this event as `BufferingEvent` in the appendix section. It does not emit these events, when the video player still has enough buffered video chunks available. Thus it is not possible to create a dynamic representation of the buffered video chunks on the time bar. However, we can show the percentage value when the video stops. Figure 18 shows how we visualize this information.

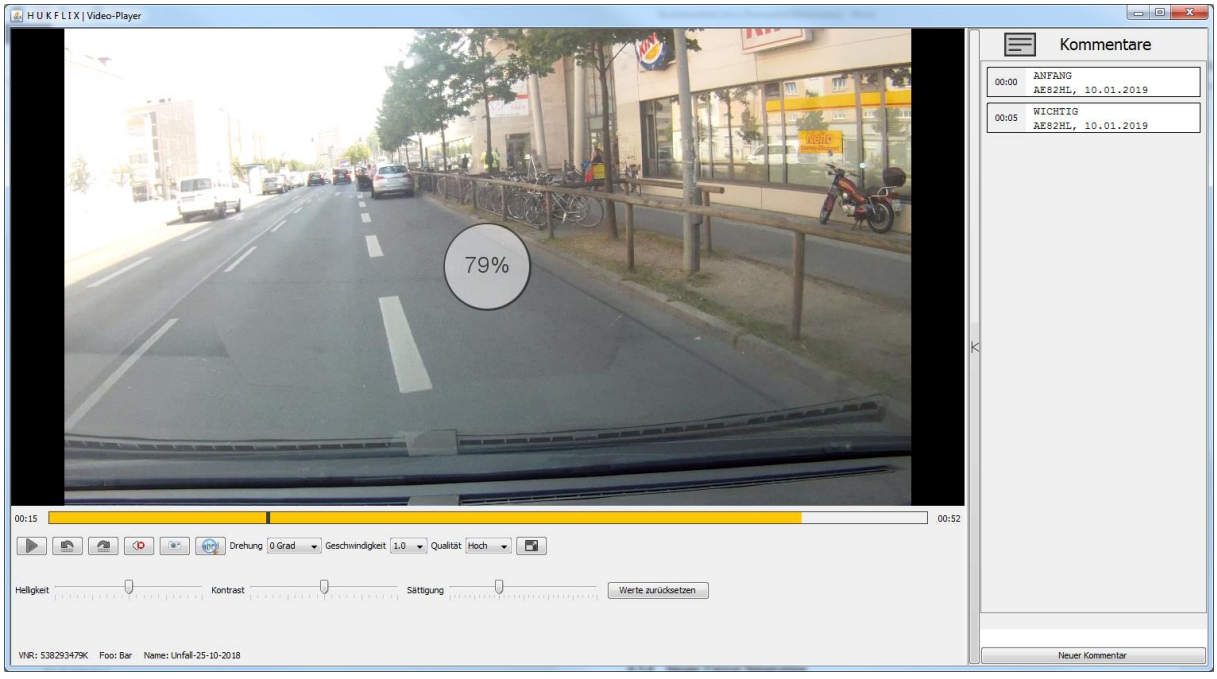

Figure 19: Buffer Overlay of the video player client

Figure 19 shows a yellow progress bar on the time bar that visualizes the percentage, here 79% are visualized. Additionally, we draw a transparent circle on top of the video component containing that percentage as well. The progress bar and the overlay update as soon as a new `BufferingEvent` is emitted by LibVLC.

The buffer overlay on top of the video component is always placed in the middle of the component, even when resizing the component. Furthermore, the font size is adapted accordingly to always fit inside of the circle. Figure 20 shows a smaller video player with the buffer overlay in the middle of the video component

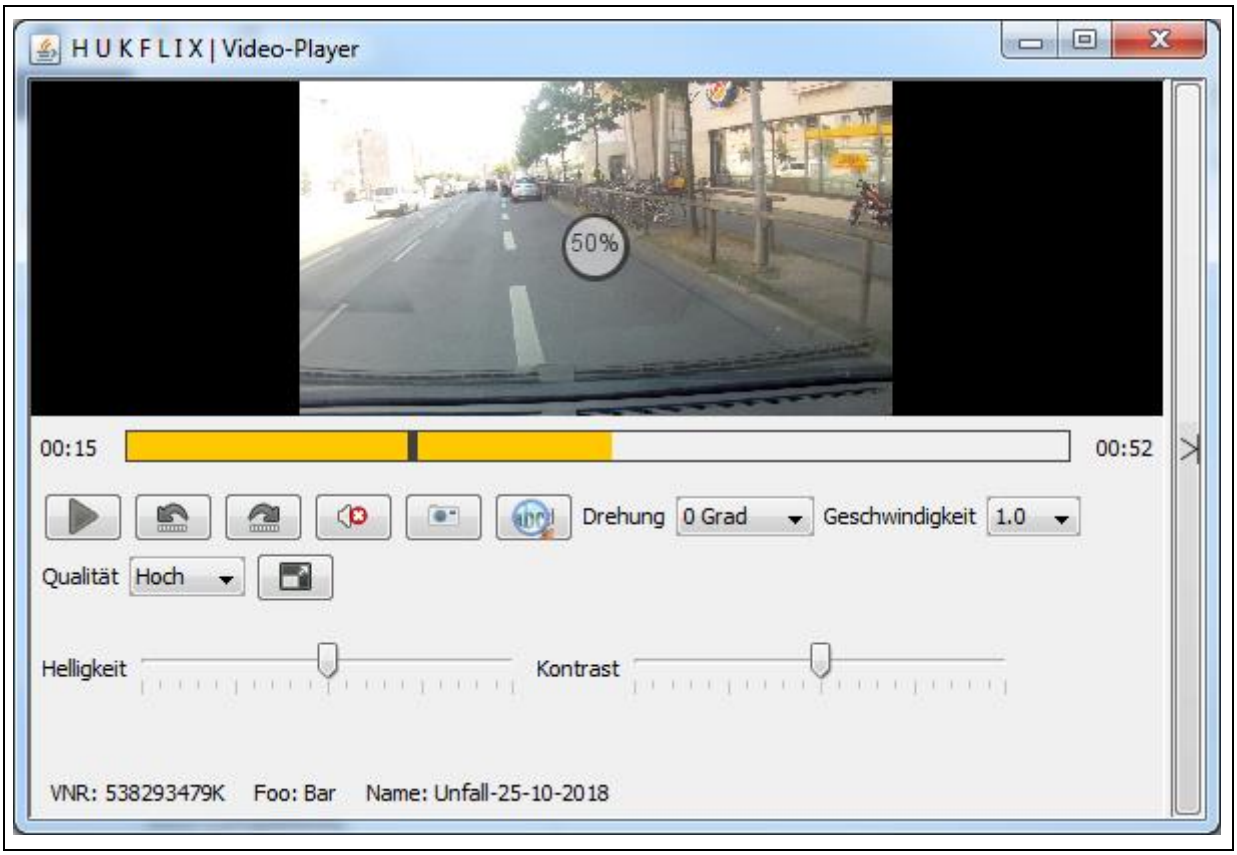

Figure 20: Smaller Buffer Overlay component in the middle of the video component

To implement this feature, we use linear interpolation and linear scaling to convert the middle position in the default video resolution to the middle position in the current size of the video component [22, p. 569ff.]. This is necessary, because a user can change the size of the video component, resulting in black borders and a changed coordinate room.

We put the implementation of the buffer overlay in the appendix section.

6.3.3.6 Snapshot of Best Quality

We always take a snapshot of the best video quality. When a user selects a lower video quality and presses the snapshot button, then we open a second video player in the background that loads the best quality at exactly the time of the snapshot. As soon as the first frame has loaded we take the snapshot. This takes longer than taking a snapshot of the already loaded video es-

pecially with low bandwidth capabilities, but it ensures that only good quality snapshots are transferred to the “Image Viewer” application.

6.3.4 Image Viewer Integration

This chapter describes the integration of the video client component into the “Image Viewer” application. The “Image Viewer” uses the proposed Video Player API to load videos and to receive and save new snapshots and comments. The following sequence diagram illustrates a typical process of interaction when watching and annotating videos.

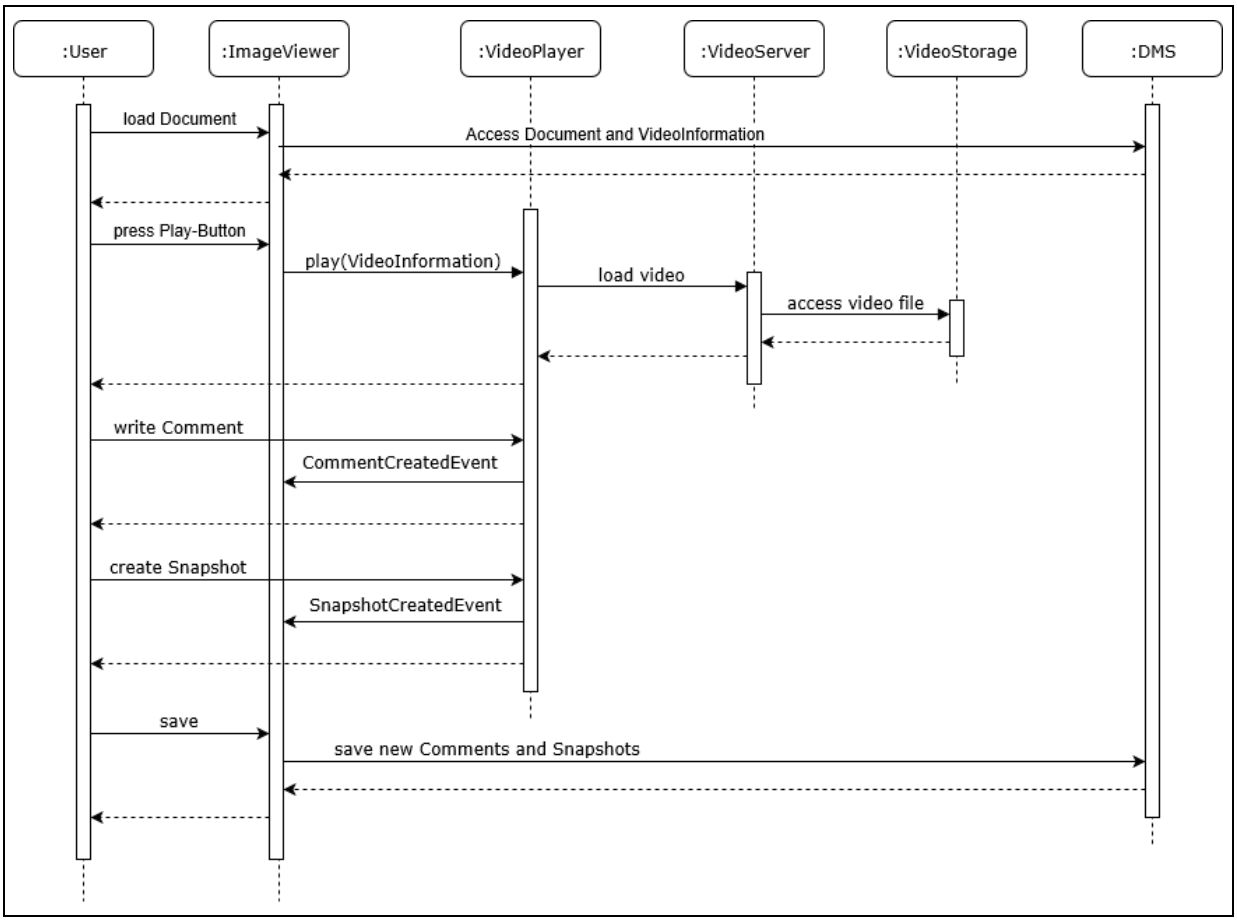

Figure 21: Sequence Diagram Image Viewer Integration

As illustrated in figure 21, first a user loads a document from our DMS. When that document contains a video, it has a `VideoInformation` object embedded into it. As a result, we draw a “play” button on top of the corresponding pages of the “Image Viewer”. When pressing on that button, the “Image Viewer” will call the `play` method of the Video Player API. The video player will then start to load the video and the user can watch it.

Later, the user might write a comment or create a snapshot of the video. Either way, the video player emits a corresponding event to the “Image Viewer”. When the user takes a snapshot, the “Image Viewer” will add a new page to its document that shows the snapshot with a corresponding “play” button, as described in 3.1 “Functional Requirements”.

Finally, a user can save changes to the DMS. In our video context, these changes are new or deleted comments or new snapshots. This action is currently not implemented, as already suggested in 1.3 “Scope of Work”.

Figure 22 shows the placement of a prototype “play” button on top of corresponding pages in the “Image Viewer” application. While its look and position are not final, it is working functionally by opening the video player as suggested in figure 20.

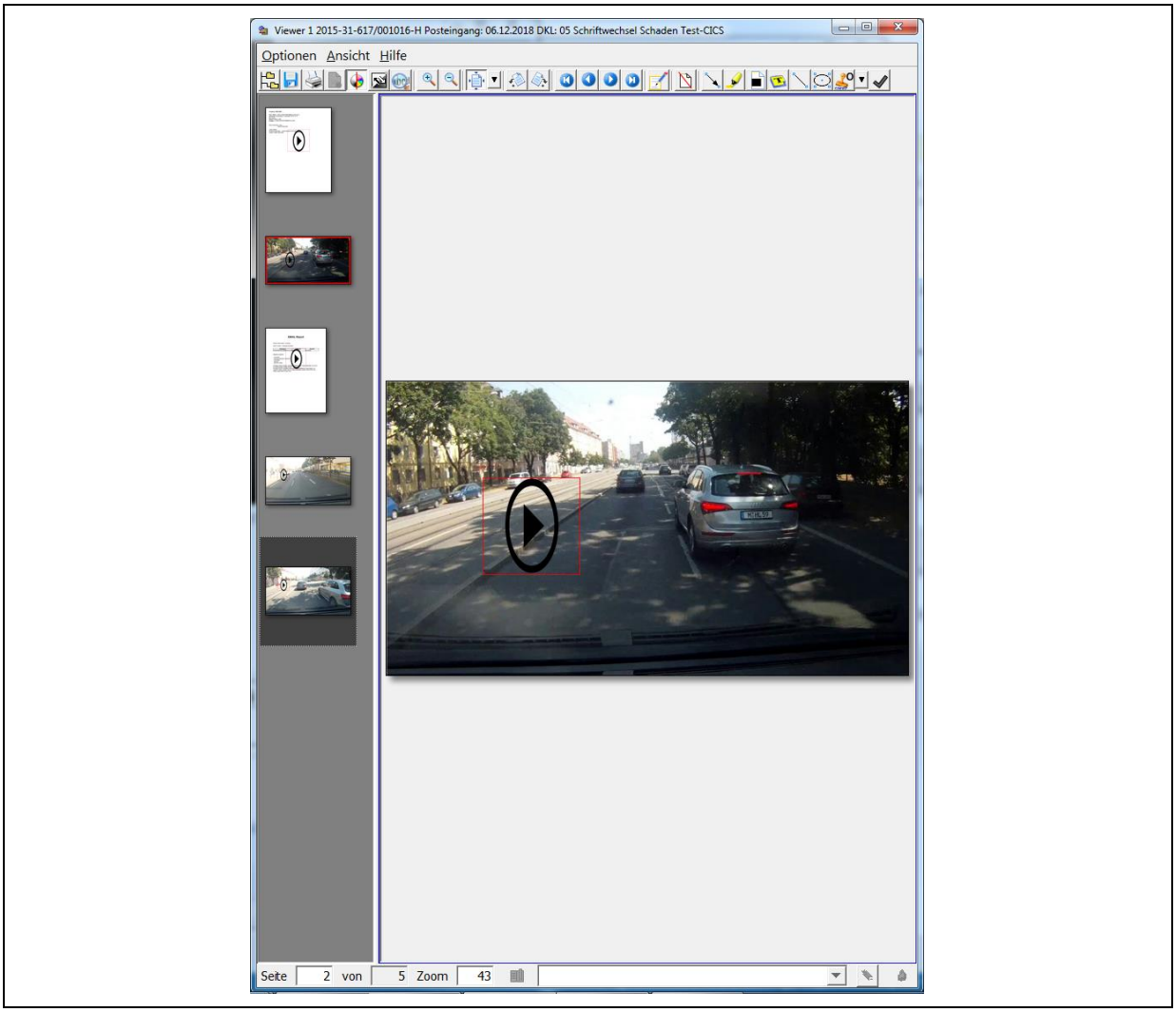

Figure 22: “Image Viewer” integration with a “play” button

We do not introduce the code necessary to achieve the integration into the “Image Viewer” application, as it would be too large for this document.

7 Testing

This chapter describes all test cases that we performed to ensure the correctness of the program and to verify the fulfillment of the requirements.

7.1 Client Test

We created multiple video player clients to simulate different test scenarios. The test scenarios are typical system-level or integration-level tests and therefore test the general functionality of a feature. We did not create unit tests due to the video player client being heavily UI-based. Instead, we always tested if the feature (e.g. using the zoom-lens) works correctly when already called from the UI components by a user manually. Through this approach we could verify that all features work correctly from a high-level black-box view.

In the following we introduce the different test clients.

7.1.1 Standalone Video Player

A single `jar` file combined with the necessary LibVLC native libraries allows us to start the video player client directly. We load a predefined video file that is available in three different video qualities with this test player. We can use this client to test the correctness of all features on multiple systems. A typical use case is to start the video player and then manually test a selection of features or all features.

7.1.2 Bandwidth Test Player

A single `jar` file combined with the necessary LibVLC native libraries allows us to start a video player used for bandwidth testing. It offers a list of available sample videos grouped with a bandwidth limitation. This allows us to test how good a specific video codec in a specific quality can be streamed with different bandwidth capabilities. We verified this through subjective quality assessment during video playback, because this is a reliable method to compare the “Quality of Experience” (QoE) with different settings [16]. Figure 23 shows the selection menu for different bandwidth limitations, here we have the option to simulate no bandwidth limitation, a bandwidth of 9 mbps and a bandwidth of 29 mbps.

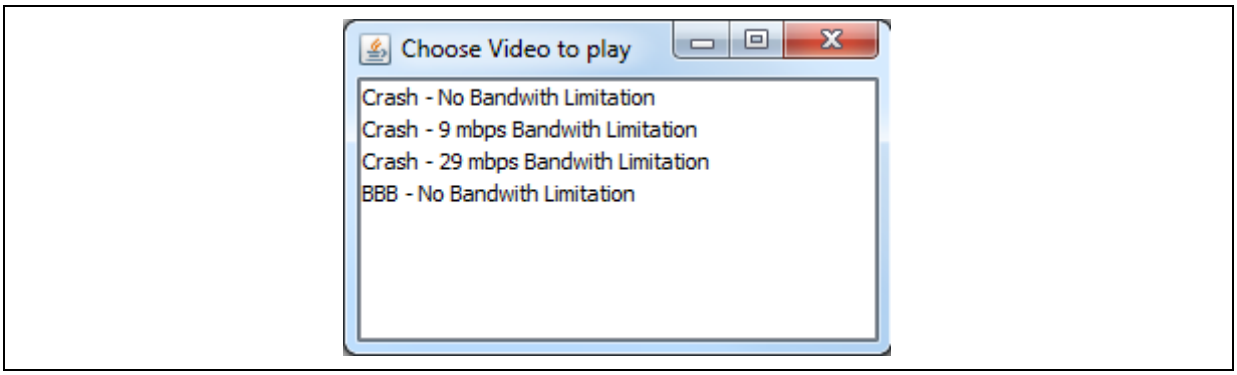

Figure 23: Bandwidth Test Player

We list results of the bandwidth tests in the subsequent chapter 7.3 “Network Test”.

7.1.3 Image Viewer Integration Distribution

We created a special version of the “Image Viewer” application that integrates the prototype video player. This version can be distributed to any case worker’s PC. We used it for evaluation of the prototype by case worker representatives and to test whether the integration in the “Image Viewer” works. We list results of the evaluation by case worker representatives in chapter 8 “Evaluation”.

7.2 Codec Test

We encoded sample videos with different codecs in different qualities to test whether the video player can play them and to test how bandwidth limitations affect each codec. We encoded a short video (duration: 50 seconds) and a longer video (duration: 5 minutes) with the codecs “H.264”, “H.265” and “VP9” and in the resolutions “640x360” (low), “1280x720” (medium) and “1920x1080” (high).

Encoding the videos with different codecs can be performed with the open-source tool “FFmpeg” [48]. The command to convert a video into a different resolution is shown in code 13.

```
ffmpeg -i input.mp4 -vf scale=-1:360 output.mp4
```

Code 13: Command to convert a video to a different resolution

The command in code 13 specifies the new resolution where `scale=-1:360` signals that the video shall be converted to a height of `360` pixels and to an according width that does not change the scale of the origin video.

A video gets transcoded to a different codec through applying one of the following commands.

```
// to VP9
ffmpeg -i input.mp4 -c:v libvpx-vp9 -crf 31 -b:v 0 output.webm


// to H.265
ffmpeg -i input -c:v libx265 -crf 28 -c:a aac -b:a 128k output.mp4


// to H.264
ffmpeg -i input.avi -c:v libx264 -preset slow -crf 23 -c:a copy output.mp4
```

Code 14: Commands to transcode a video to a different codec

Code 14 lists three different commands to convert to the three proposed codecs.

The video player can play all three of these codecs. We list results of the bandwidth test with the different video resolutions in the next chapter.

Additionally to the three codecs, we created sample video files suited for streaming via the "MPEG-DASH" protocol. The tool "mp4box" can convert a regular video file into segmented files and it creates the required MPD file [49]. The following command creates such a file structure.

```
mp4box -dash 4000 -frag 4000 -rap -segment-name segment_ high.mp4 middle.mp4 low.mp4
```

Code 15: Command for creation of MPEG-DASH video files

Code 15 creates a MPD file and several segmented video files with each segment containing 4000 milliseconds of the video. We provide the same video in three different qualities, thus each segment is represented in three different qualities as well. The MPD file knows about all three qualities and therefore enables the adaptive quality change of "MPEG-DASH".

We verified that the video player can play videos provided through this method. It can also switch between different qualities depending on the bandwidth capabilities.

7.3 Network Test

We tested the different codecs and qualities, which we proposed in 7.2 "Codec Test", in regard to our bandwidth limitations in different branch offices at HUK-COBURG. Some branch

offices use a bandwidth throughput of 10 mbps or 30 mbps, which are the lowest levels that we currently use. Hence, we tested how much bandwidth gets consumed by one video playback. Table 9 shows the results for this test. We only tested the video qualities "low" and "high". We measured the consumed bandwidth at its peak while playing a 50 second video clip on one PC.

	360p ("low")	1080p ("high")
H.264	1,6 – 1,75 mbps	22 mbps
H.265	1 – 1,05 mbps	6,5 – 7 mbps
VP9	0,8 – 1,05 mbps	19 mbps

Table 9: Network Test: Bandwidth Throughput

Table 9 shows that the necessary bandwidth throughput differs greatly depending on the video codec. It also shows that the encoding process influences the bandwidth throughput for each codec. Looking at "VP9", its consumed bandwidth at 1080p is significantly higher than that of "H.265", while both require equally much bandwidth at 360p. Wrong encoding settings resulted in that higher bandwidth throughput. It is therefore important to choose the right encoding settings to ensure the best bandwidth/quality ratio for each codec. The statistic in figure 24 shows that both "H.265" and "VP9" are about 50% more efficient concerning bitrate than "H264" is. As a result, streaming a video encoded with "H.264" consumes about 50% more bandwidth than streaming a video with equal quality encoded with "H.265" or "VP9".

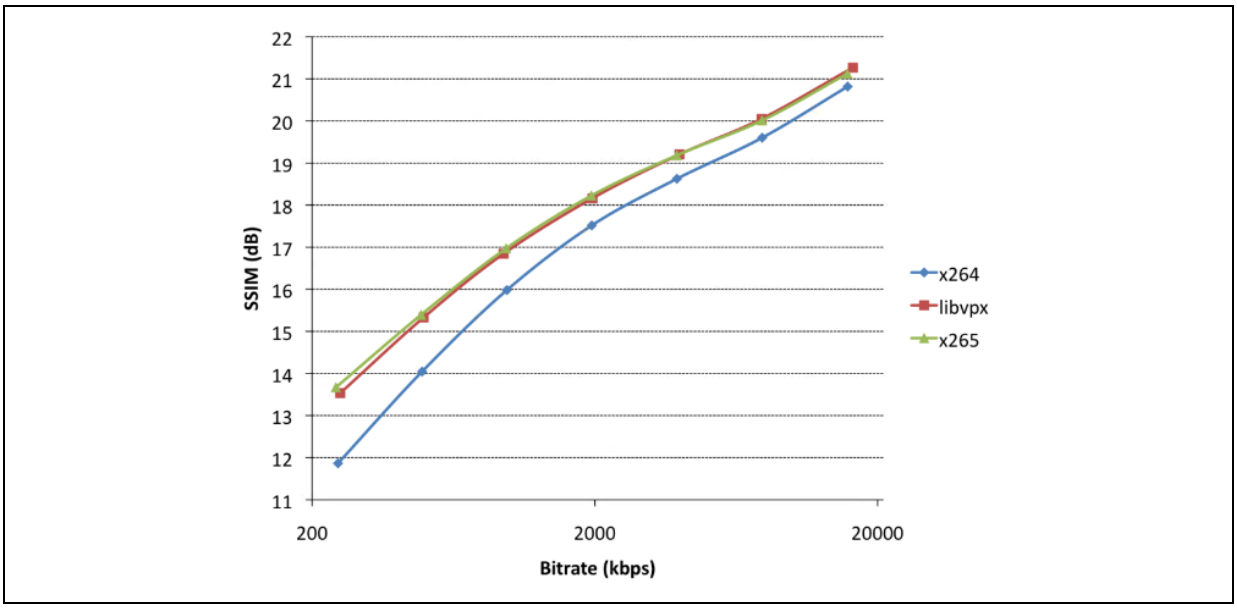

Figure 24: "Bitrate per Quality" measure for the codecs "H.264", "H.265" and "VP9" [50]

Furthermore, we tested how many videos can be watched simultaneously encoded in the "H.264" codec. Table 10 shows the results for this test.

Video-Count	Bandwidth	360p ("low")	720p ("medium")	1080p ("high")
1-2	10 mbps	OK	OK	-
	30 mbps	OK	OK	OK
4-5	10 mbps	OK	-	-
	30 mbps	OK	OK	-

Table 10: Network Test: Simultaneous Playback

Table 10 shows that a high quality "H.264" video cannot be watched fluently with 10 mbps. When watching 4 to 5 video simultaneously, a medium quality "H.264" video cannot be watched fluently with 10 mbps, as well. Furthermore, 4 to 5 high quality "H.264" videos cannot be watched simultaneously with 30 mbps.

A conclusion of both tests in table 9 and 10 is that high quality videos might not always be watched smoothly with low bandwidth capabilities. On the other side, low and medium qualities can still be watched smoothly most of the time. Furthermore, a difference between medium and high quality videos is only visible when watching a video in "full screen" mode. As a result, we think that the proposed bandwidth limitations are of no big concern for case workers, because they can switch the video quality manually. When a high quality must be provided, the visual buffer overlay that we introduced in 6.3.3.5 "Buffer Overlay" allows a case worker to estimate the waiting time, additionally.

8 Evaluation

In this chapter we describe the results of the prototype's evaluation that case worker representatives performed. Two representatives of the division concerning liability cases and five representatives of different divisions specialized on one area of insurance performed the evaluation. It states whether a functional or non-functional requirement is fulfilled. Table 11 shows the results for the functional requirements.

Functional Requirement	Fulfilled
"Image Viewer" Integration ("Play Button")	Yes
Basic Video Player Functions	Yes
Comments	Yes
Snapshots	Yes
Video Quality	Yes

Table 11: Evaluation of functional requirements

All user stories that we proposed in chapter 3.1 "Functional Requirements" are fulfilled and work as expected. The evaluation in table 11 does not respect non-functional requirements like usability, but only the functional correctness of the features.

Table 12 shows the results for the non-functional requirements and provides explanations when a requirement is not fulfilled.

Non-Functional Requirement	Fulfilled	Explanation
Look and Feel	Yes	
Usability	No	We developed a prototype and therefore did not aim to fulfill this category fully. Most of the features are usable already, but some aspects miss: • We do not yet provide tooltip texts for the toolbar. • The "Play/Pause" action is not triggered when pressing inside the video.
Buffering Time	No	Choosing the video quality and having the possibility to let

		"MPEG-DASH" choose the quality is rated as a good solution for buffering time. However, the case worker representatives asked for the possibility to download a video completely as another option. This would eliminate buffering during play-back by requiring an initial waiting time.
Reliability	Not tested	This requirement needs to be tested in a long-time test case.

Table 12: Evaluation of non-functional requirements

We can implement the missing usability features, which we proposed in table 12, into the prototype. However, we need to investigate the possibility to download the video completely, because this would require the video to be temporarily saved onto the case worker's file system. Security and data privacy issues might arise when implementing this feature.

Overall, the initial reaction of case workers to our prototype is positive. It provides them with the required toolset to work off liability cases that contain videos in a modern and integrated workflow. Therefore, it increases their efficiency because case workers no longer need to request a video through an extensive manual process and no longer need to take it to a separate work station. Furthermore, case workers simply need such a tool because they already receive videos and need a way to quickly access and annotate them.

9 Conclusion and Prospect

This chapter summarizes the findings of this bachelor's thesis and provides a brief prospect of further work.

Chapter 1.2 "Problem and Goal" proposes four objectives that need to be completed in order to integrate videos in the digital workflow of a case worker's day-to-day routines. This bachelor's thesis investigated two of these four objectives in-depth: choosing a video streaming protocol and creating a video player client with the aid of a video player framework. We provided some insight into the other two objectives as well: video codecs played a significant role in choosing the video streaming protocols and we developed a prototype server-side architecture for streaming.

After comparing streaming protocols and video player frameworks, we recommended the use of "HTTP Pseudo Streaming" and "MPEG-DASH" as protocols and "VLCJ" as video player framework. Furthermore, the prototype fulfills all functional requirements and can be extended easily to meet all non-functional requirements, as well.

In order to achieve a complete integration of videos in our DMS, we need to extend the server-side component by providing a database solution and by connecting it to our existing DMS. Furthermore, the "Image Viewer" application needs to implement a "save" function for modified video content, like comments and snapshots on top of that. Further investigation is necessary to decide whether one or multiple video codecs are suited for our DMS or whether we should restrict video codecs at all.

When these aspects are implemented and the integration is therefore complete, we can investigate advanced features of the video player client. For example, providing object tracking features to detect a car accident automatically is such an advanced feature. At HUK-COBURG we will conduct a follow-up project to this bachelor's thesis to implement the missing pieces of a digital workflow for videos in our DMS.

When our prototype is fully integrated in the DMS, it provides many benefits to case workers. First, they do no longer need to request videos manually and watch them on standalone work stations. Instead, they can access videos quickly at any time by using the programs that they already use. Secondly, they can use advanced features like zooming, image processing or frame-by-frame playback to better analyze videos. Finally, they can create comments, bookmarks and snapshots persistently, which enables them to write down findings for later usage or for finalization of a task.

References

[1] HUK-COBURG, "Firmengeschichte," aus dem Intranet (aufgerufen am: 26.10.2017), 2017.

[2] HUK-COBURG, "Kennzahl: Dokumenteneingänge gesamt," HUK-COBURG, Coburg, 2016, 2017.

[3] *VI ZR 233/17: Verwertbarkeit von Dashcam-Aufnahmen als Beweismittel im Unfallhaftpflichtprozess*, 2018.

[4] Gesamtverband der Deutschen Versicherungswirtschaft e. V., "Aufbewahrungspflichten und Aufbewahrungsgrundsätze für Geschäftsunterlagen von Versicherungsunternehmen," Gesamtverband der Deutschen Versicherungswirtschaft e. V., Berlin, 2010.

[5] K. Angerhausen, D. Dr. Weiß, M. Dörflein and C. Schneider, Software im Vergleich Dokumenten-Management - Dokumenten-Management- und Archivierungs-Systeme, München: Oxygon Verlag, 2003.

[6] R. MANTEL, "Video File Formats, Codecs, and Containers Explained," 9 1 2018. [Online]. Available: https://www.techsmith.com/blog/video-file-formats/. [Accessed 26 12 2018].

[7] L. Case, "All About Video Codecs and Containers," 14 12 2010. [Online]. Available: https://www.techhive.com/article/213612/all_about_video_codecs_and_containers.html?page=2. [Accessed 26 12 2018].

[8] J. Ostermann and e. al., "Video coding with H.264/AVC: tools, performance, and complexity," *IEEE Circuits and Systems magazine 4.1*, pp. 7-28, 2004.

[9] G. J. Sullivan and e. al., "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions on circuits and systems for video technology 22.12*, pp. 1649-1668, 2012.

[10] D. Mukherjee and e. al., "The latest open-source video codec VP9 - An overview and preliminary results," *Picture Coding Symposium (PCS)*, pp. 390-393, 2013.

[11] Y. Chen and e. al., "An Overview of Core Coding Tools in the AV1 Video Codec," *Picture Coding Symposium (PCS)*, pp. 24-27, 2018.

[12] Encoding.com, "Global Media Format Report," encoding.com, San Francisco, 2017.

[13] MPEG-LA, "AVC Patent Portfolio License Briefing," 21 05 2018. [Online]. Available: http://www.mpegla.com/main/programs/AVC/Documents/avcweb.pdf. [Accessed 26 12 2018].

[14] J. Bakoski, M. Frost and A. Grange, "The internet needs a competitive, royalty-free video codec," Google Inc., Mountain View, USA, 2017.

[15] Cisco, "Cisco Visual Networking Index," 8 6 2017. [Online]. Available: https://newsroom.cisco.com/press-release-content?type=webcontent&articleId=1853168. [Accessed 27 12 2018].

[16] K. Singh, Y. Hadjadj-Aoul and G. Rubino, "Quality of Experience estimation for Adaptive HTTP/TCP video streaming using H.264/AVC," in *CCNC - IEEE Consumer Communications Networking Conference*, Las Vegas, United States, 2012.

[17] T. Stockhammer, "Dynamic adaptive streaming over HTTP--: standards and design principles," *Proceedings of the second annual ACM conference on Multimedia systems*, pp. 133-144, 2011.

[18] M. Michalos, S. P. Kessanidis and S. L. Nalmpantis, "Dynamic Adaptive Streaming over HTTP," *Journal of Engineering Science & Technology Review*, 2012.

[19] R. Fielding and e. al., "Hypertext transfer protocol--HTTP/1.1," 1999.

[20] J. Ozer, "Bitmovin Survey: HLS Still Dominates, but DASH is on the Rise," streamingmedia.com, 11 09 2017. [Online]. Available: http://www.streamingmedia.com/Articles/News/Online-Video-News/Bitmovin-Survey-HLS-Still-Dominates-but-DASH-is-on-the-Rise-120419.aspx. [Accessed 14 1 2019].

[21] H. Schulzrinne, A. Rao and R. Lanphier, "Real time streaming protocol (RTSP)," 1998.

[22] W. Burger and M. J. Burge, Digitale Bildverarbeitung - Eine algorithmische Einführung mit Java, 3. ed., Berlin: Springer, 2015.

[23] S. Birchfield, Image Processing and Analysis, Clemson University: CENGAGE Learning, 2016.

[24] A. Durresi and R. Jain, "RTP, RTCP, and RTSP - Internet Protocols for Real-Time Multimedia Communication," in *The Industrial Information Technology Handbook*, Louisiana, CRC Press LLC, 2005, pp. 28.1 - 28.11.

[25] Wikipedia, "RTP audio video profile," 06 11 2018. [Online]. Available: https://en.wikipedia.org/wiki/RTP_audio_video_profile. [Accessed 20 12 2018].

[26] T. Dias, "How to stream RTSP live video in Firefox and Chrome now that the VLC plugin is not supported anymore?," StackOverflow, 12 12 2015. [Online]. Available: https://stackoverflow.com/questions/33080899/how-to-stream-rtsp-live-video-in-firefox-and-chrome-now-that-the-vlc-plugin-is-n. [Accessed 27 12 2018].

[27] D. Chu, C.-h. Jiang, Z.-b. Hao and W. Jiang, "The Design and Implementation of Video Surveillance System Based on H.264, SIP, RTP/RTCP and RTSP," *2013 Sixth International Symposium on Computational Intelligence and Design*, pp. 39-43, 28 10 2013.

[28] R. Denis-Courmont, "Video on Demand: RTSP vs HTTP," 08 07 2009. [Online]. Available: https://www.remlab.net/op/vod.shtml. [Accessed 27 12 2018].

[29] S. Laine and I. Hakala, "H.264 QoS and Application Performance with Different Streaming Protocols," ResearchGate, Kokkola, 2015.

[30] M. Levkov, "Understanding the MPEG-4 Movie Atom," 18 10 2010. [Online]. Available: https://www.adobe.com/devnet/video/articles/mp4_movie_atom.html. [Accessed 27 12 2018].

[31] S. DHAGE and B. B. MESHRAM, "Buffer management scheme for video-on-demand (VoD) System," *2012 International Conference on Information and Computer Networks*

(ICICN 2012) IPCSIT, pp. 77-82, 2012.

[32] IRT, "International Intensive Seminar Adaptive Streaming & MPEG DASH," IRT, München, 2018.

[33] Caprica, "VLCJ," 2016. [Online]. Available: https://github.com/caprica/vlcj. [Accessed 29 12 2018].

[34] VideoLAN, "VLC Features," [Online]. Available: https://www.videolan.org/vlc/features.html. [Accessed 29 12 2018].

[35] OpenJFX, "JavaFX 11," 2018. [Online]. Available: https://openjfx.io/. [Accessed 29 12 2018].

[36] Oracle, "JavaDoc: JavaFX Media," 2014. [Online]. Available: https://docs.oracle.com/javafx/2/api/javafx/scene/media/package-summary.html. [Accessed 29 12 2018].

[37] Oracle, "Introduction to JavaFX Media," 2014. [Online]. Available: https://docs.oracle.com/javase/8/javafx/media-tutorial/overview.htm. [Accessed 29 12 2018].

[38] TeamDev Ltd., "JxBrowser," 2018. [Online]. Available: https://www.teamdev.com/jxbrowser. [Accessed 2 1 2019].

[39] The Chromium Project, "The Chromium Projects," 2018. [Online]. Available: https://www.chromium.org/Home. [Accessed 2 1 2019].

[40] TeamDev, "JxBrowser: HTML5 Video," 6 6 2016. [Online]. Available: https://jxbrowser.support.teamdev.com/support/solutions/articles/9000074414-html5-video. [Accessed 2 1 2019].

[41] TeamDev, "JxBrowser: MP3/MP4/H.264," 2 10 2017. [Online]. Available: https://jxbrowser.support.teamdev.com/support/solutions/articles/9000013050-mp3-mp4-h-264. [Accessed 2 1 2019].

[42] W3Schools, "HTML5 Video," 2018. [Online]. Available: https://www.w3schools.com/html/html5_video.asp. [Accessed 2 1 2019].

[43] DASH-IF, "Dash.js," 2018. [Online]. Available: https://dashif.org/dash.js/. [Accessed 2 1 2019].

[44] TeamDev, "JxBrowser: Taking Screenshots," 2018. [Online]. Available: https://jxbrowser.support.teamdev.com/support/solutions/articles/9000012877-taking-screenshots. [Accessed 2 1 2019].

[45] Pivotal Software, "Spring Boot," 2019. [Online]. Available: https://spring.io/projects/spring-boot. [Accessed 5 1 2019].

[46] davinkevin, "MultiPartFileSender," 15 2 2015. [Online]. Available: https://github.com/davinkevin/Podcast-Server/blob/d927d9b8cb9ea1268af74316cd20b7192ca92da7/src/main/java/lan/dk/podcastserver/utils/multipart/MultipartFileSender.java. [Accessed 5 1 2019].

[47] VideoLAN, "VLC command-line help," 14 3 2018. [Online]. Available:

https://wiki.videolan.org/VLC_command-line_help/. [Accessed 10 1 2019].

[48] FFmpeg, "FFmpeg," 6 11 2018. [Online]. Available: https://www.ffmpeg.org/. [Accessed 11 01 2019].

[49] GPAC, "MP4Box," 2018. [Online]. Available: https://gpac.wp.imt.fr/mp4box/. [Accessed 11 1 2019].

[50] R. S. Bultje, "VP9 encoding/decoding performance vs. HEVC/H.264," 28 09 2015. [Online]. Available: https://blogs.gnome.org/rbultje/2015/09/28/vp9-encodingdecoding-performance-vs-hevch-264/. [Accessed 29 01 2019].

[51] "MPEG-DASH / Media Source demo," 27 8 2012. [Online]. Available: http://dash-mse-test.appspot.com/media.html. [Accessed 27 12 2018].

Glossary

Image Viewer	The client used by case workers to look at and annotate documents of any kind. It is used in their day-to-day routines to work off liability cases or other contract-related work of customers. The application is integrated in our DMS, as suggested in chapter 2.1 “Document Management Systems”.
Quality of Experience (QoE)	A measurement to rate how smoothly videos can play back in the perspective of an end-user. It does not consider factors like bandwidth usage or other technical factors, but does only consider the experience of a user while watching a video. The QoE can be obtained through manual assessment or automated tests [16].
Video Annotation	In the context of this bachelor’s thesis we define the annotation process for videos to be one of the following actions: • add/remove a comment • take a snapshot • zooming the video • changing brightness, contrast or saturation In general, annotation features help a case worker to work off a liability case by leaving notes, highlighting important aspects or investigating a moment in-depth.

Appendix A 1. **Ehrenwörtliche Erklärung**

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Titel

selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie nicht an anderer Stelle als Prüfungsarbeit vorgelegt habe.

Ort

Datum

Unterschrift

Appendix A 2. **MPD File Example**

We put an example MPD file used for the streaming protocol “MPEG-DASH” in this chapter. Code 16 contains all concepts needed for “MPEG-DASH” as introduced in chapter 4.3.

```
<MPD xmlns="urn:mpeg:DASH:schema:MPD:2011" mediaPresenta-
tionDuration="PT0H3M1.63S" minBufferTime="PT1.5S" pro-
files="urn:mpeg:dash:profile:isoff-on-demand:2011"
type="static">
  <Period duration="PT0H3M1.63S" start="PT0S">
    <AdaptationSet>
      <ContentComponent contentType="video" id="1" />
      <Representation bandwidth="4190760" codecs="avc1.640028"
height="1080" id="1" mimeType="video/mp4" width="1920">
        <BaseURL>car-20120827-89.mp4</BaseURL>
        <SegmentBase indexRange="674-1149">
          <Initialization range="0-673" />
        </SegmentBase>
      </Representation>
      <Representation bandwidth="2073921" codecs="avc1.4d401f"
height="720" id="2" mimeType="video/mp4" width="1280">
        <BaseURL>car-20120827-88.mp4</BaseURL>
        <SegmentBase indexRange="708-1183">
          <Initialization range="0-707" />
        </SegmentBase>
      </Representation>
      <Representation bandwidth="869460" codecs="avc1.4d401e"
height="480" id="3" mimeType="video/mp4" width="854">
        <BaseURL>car-20120827-87.mp4</BaseURL>
        <SegmentBase indexRange="708-1183">
          <Initialization range="0-707" />
        </SegmentBase>
      </Representation>
      <Representation bandwidth="686521" codecs="avc1.4d401e"
height="360" id="4" mimeType="video/mp4" width="640">
        <BaseURL>car-20120827-86.mp4</BaseURL>
        <SegmentBase indexRange="708-1183">
          <Initialization range="0-707" />
```

```
        </SegmentBase>
      </Representation>
      <Representation bandwidth="264835" codecs="avc1.4d4015"
height="240" id="5" mimeType="video/mp4" width="426">
        <BaseURL>car-20120827-85.mp4</BaseURL>
        <SegmentBase indexRange="672-1147">
          <Initialization range="0-671" />
        </SegmentBase>
      </Representation>
      <Representation bandwidth="100000" codecs="avc1.4d4015"
height="144" id="5" mimeType="video/mp4" width="256">
        <BaseURL>car-20120827-160.mp4</BaseURL>
        <SegmentBase indexRange="671-1146">
          <Initialization range="0-670" />
        </SegmentBase>
      </Representation>
    </AdaptationSet>
    <AdaptationSet>
      <ContentComponent contentType="audio" id="2" />
      <Representation bandwidth="127236" codecs="mp4a.40.2"
id="6" mimeType="audio/mp4" numChannels="2" sam-
pleRate="44100">
        <BaseURL>car-20120827-8c.mp4</BaseURL>
        <SegmentBase indexRange="592-851">
          <Initialization range="0-591" />
        </SegmentBase>
      </Representation>
      <Representation bandwidth="255236" codecs="mp4a.40.2"
id="7" mimeType="audio/mp4" numChannels="2" sam-
pleRate="44100">
        <BaseURL>car-20120827-8d.mp4</BaseURL>
        <SegmentBase indexRange="592-851">
          <Initialization range="0-591" />
        </SegmentBase>
      </Representation>
      <Representation bandwidth="31749" codecs="mp4a.40.5"
id="8" mimeType="audio/mp4" numChannels="1" sam-
pleRate="22050">
```

```
        <BaseURL>car-20120827-8b.mp4</BaseURL>
        <SegmentBase indexRange="592-851">
          <Initialization range="0-591" />
        </SegmentBase>
      </Representation>
    </AdaptationSet>
  </Period>
</MPD>
```

Code 16: Example MPD file [51]

Appendix A 3. **Prototype Streaming Server Code**

The following code examples show pseudo code of the streaming server component that we introduce in chapter 6.2.

Code 17 shows the streaming endpoint definition, which is based on the "Spring MVC" pattern. It defines, which java methods will process which HTTP requests under which URLs.

```
@RestController
@RequestMapping("/videos")
public class VideoController {

@GetMapping
@RequestMapping("/stream/{id}")
public ResponseEntity<InputStreamResource> getVideo(...){}

@GetMapping
@RequestMapping("/stream")
public void getVideoAsStreamWithQuery(
  @RequestParam(value="c", required=false) String codec,
  @RequestParam(value="q", required=false) String quality,
  @RequestParam(value="d", required=false) String duration,
  @RequestParam(value="buf", required=false,
                defaultValue="20480") String bufferSize,
  @RequestParam(value="bandwith", required=false,
                defaultValue="-1") String bandwith,
  @RequestParam(value="name", required=false) String fileName,
  HttpServletRequest request, HttpServletResponse response){}

@GetMapping
@RequestMapping("/stream2/{id}")
public void getVideoAsStream2(...){}

@StoreRestResource(path="videos/stream3")
public interface VideoStore extends Store<String> {}
```

Code 17: Definition of the prototype streaming endpoint

In code 18 we illustrate the main concept of sending data in chunks to the client. This implements the concept of partial content requests in "HTTP Pseudo Streaming".

```
int bytesRead;
```

```
int bytesLeft = contentLength;
ByteBuffer buffer = ByteBuffer.allocate(BUFFER_LENGTH);


try (SeekableByteChannel input = Files.newByteChannel(video,
                                    StandardOpenOption.READ);
     OutputStream output = response.getOutputStream()) {

  input.position(start); // byte range start position

  while ((bytesRead=input.read(buffer))!=-1 && bytesLeft>0) {
    buffer.clear();
    output.write(buffer.array(), 0, bytesLeft < bytesRead
                                  ? bytesLeft : bytesRead);
    bytesLeft -= bytesRead;
  }
}
```

Code 18: Partial Content Request Code Implementation

Appendix A 4. **Events in the Video Player Component**

Table 13 shows all events that are distributed by the `EventHandler` of the video player component. It either contains an action that needs to be performed by the receiving component or indicates new information about video playback.

Event Name	Description
`BrightnessEvent`	Change the brightness of the video to another value (0.0 - 2.0)
`BufferingEvent`	Indicates the percentage of the next video chunk that is already streamed. BufferingEvents are only distributed if the video needs to stop and wait for the next video chunk to be loaded completely.
`ContrastEvent`	Change the contrast of the video to another value (0.0 – 2.0)
`CreateCommentEvent`	Add a comment to the video. The comment is contained in the event.
`DeleteCommentEvent`	Delete a comment from the video. The comment is contained in the event.
`ErrorEvent`	An error occurred. The error is contained in the event.
`FinishedEvent`	Indicates that the playback of the video finished.
`FullscreenEvent`	Enter or leave the full screen mode of the video.
`MuteEvent`	Mute or unmute the sound.
`NextFrameEvent`	Show the next frame of the video (will stop video playback).
`PauseEvent`	Stop video playback.
`PersOrgaEvent`	Update the personal organization information of the user that is using the video player.
`PlaybackQuality-Event`	Change the quality of the video to the specified level.
`PlaybackSpeedEvent`	Change the speed of the video playback to the specified level (0.25x, 0.5x, 1x, 2x, 4x)

PlayerClosingEvent	Indicates that the video player window will be hidden.
PlayEvent	Continue video playback from the current position.
PositionEvent	Indicates the new position (percentage-based) in the video (for example 35% of the video is currently watched).
RequestScreenshot-Event	Take a snapshot from the current frame with the best possible quality.
ResetEvent	Reset the player to its default values that it uses when it is started for the first time.
RotationEvent	Rotate the video by 0, 90, 180 or 270 degrees.
SaturationEvent	Change the saturation of the video to another value (0.0 - 3.0)
ScreenshotTakenE-vent	Indicates that a snapshot was taken. The snapshot is contained in the event.
TakeScreenshotOf-CurrentQualityEvent	Take a snapshot from the current frame with the current quality.
TimeEvent	Indicates the new time (in milliseconds) in the video (for example video playback is currently at 12345 ms).
ToggleComment-VisibilityEvent	Show or hide the comment feed.
TogglePlayingEvent	Play or pause video playback.
VideoDimension-ChangedEvent	Indicates that the native resolution of the video changed.
VideoInformation-Event	The video player streams a new video. The event contains information about the video (URL, qualities, comments, …).
VideoLengthChanged-Event	The total duration of the video (in milliseconds) changed.
VolumeEvent	The volume value of sound playback changed (0-100)
ZoomLensEvent	Activate or deactivate the zoom lens.

Table 13: All events of the video player component

Appendix A 5. **Buffer Overlay Implementation**

Code 19 shows the implementation of the buffer overlay component as described in chapter "6.3.3.5".

```
@Override
public void paint(Graphics g) {
super.paint(g);
Graphics2D g2 = (Graphics2D)g;
if(videoDimension != null && bufferPercentage < 100) {
  int w = videoSurface.getWidth();
  int h = videoSurface.getHeight();
  int bufferCircleRadius = videoDimension.width / 10;
  Point upperLeftCirclePoint =
  new Point(videoDimension.width/2 - bufferCircleRadius / 2,
            videoDimension.height/2 - bufferCircleRadius / 2);


  // linear scaling from VideoDimension (e.g. 1920 x 1080) to
  // VideoSurface (e.g. 1000 x 500)
  int interpolated_x = (int) (1.0f * upperLeftCirclePoint.x *
                              w / videoDimension.width);
  int interpolated_y = (int) (1.0f * upperLeftCirclePoint.y *
                              h / videoDimension.height);
  int interpolated_w = (int) (1.0f * bufferCircleRadius * w /
                              videoDimension.width);
  int interpolated_h = (int) (1.0f * bufferCircleRadius * h /
                              videoDimension.height);


  float aspectRatio = 1.0f * videoDimension.width /
                             videoDimension.height;
  float surfaceRatio = 1.0f * w / h;


  //Determine black borders
  if(surfaceRatio > aspectRatio) {
     //border left/right -> change x / width
     int actualWidth = (int) (aspectRatio * h);
     int borderSize = w - actualWidth; //left and right
```

```
    //recalculate values with actual width and add half of
    //border size
    interpolated_x = (int) (1.0f * upperLeftCirclePoint.x *
                         actualWidth / videoDimension.width)
                     + borderSize/2;
    interpolated_w = (int) (1.0f * bufferCircleRadius *
                        actualWidth / videoDimension.width);
  } else {
    //border up/down -> change y / height
    int actualHeight = (int) (w / aspectRatio);
    int borderSize = h - actualHeight; //top and down
    //recalculate values with actual height and add half of
    //border size
    interpolated_y = (int) (1.0f * upperLeftCirclePoint.y *
                       actualHeight / videoDimension.height)
                     + borderSize/2;
    interpolated_h = (int) (1.0f * bufferCircleRadius *
                      actualHeight / videoDimension.height);
  }

g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
g2.setStroke(new BasicStroke(5f));
g.setColor(Color.BLACK);
g2.drawOval(interpolated_x, interpolated_y,
            interpolated_w, interpolated_h); //circle
g.setColor(Color.WHITE);
g2.fillOval(interpolated_x, interpolated_y,
            interpolated_w, interpolated_h); //circle
g.setColor(Color.BLACK);
FontMetrics metrics = g2.getFontMetrics();
float size = g2.getFont().getSize2D();

while(metrics.getHeight() < interpolated_h/3) {
  size += 2f;
  g2.setFont(g2.getFont().deriveFont(size));
  metrics = g2.getFontMetrics();
}
String bufferText = Math.round(bufferPercentage) + "%";
Rectangle2D bufferTextBounds =
                     metrics.getStringBounds(bufferText, g2);
```

```
g2.drawString(bufferText, interpolated_x + interpolated_w/2 -
                          (int) (bufferTextBounds.getWidth()/2),
               interpolated_y + interpolated_h/2 + (int)
                          (bufferTextBounds.getHeight()/4));

}
}
```

Code 19: Buffer Overlay Scaling Algorithm