2014•2015
FACULTEIT WETENSCHAPPEN
master in de informatica

Masterproef
Bandwidth management for ODV tiled streaming with MPEG-DASH

Promotor :
Prof. dr. Peter QUAX

Copromotor :
Prof. dr. Wim LAMOTTE

Geoffrey Martens
Scriptie ingediend tot het behalen van de graad van master in de informatica

Maastricht University

2014•2015
FACULTEIT WETENSCHAPPEN
master in de informatica

Masterproef

Bandwidth management for ODV tiled streaming with MPEG-DASH

Promotor :
Prof. dr. Peter QUAX

Copromotor :
Prof. dr. Wim LAMOTTE

Geoffrey Martens
Scriptie ingediend tot het behalen van de graad van master in de informatica

Abstract

Streaming video in parallel over the internet is popular these days. The challenge in this is to divide bandwidth over the streaming videos in order to improve the QoE of the users. The QoE is related to four parameters, namely the startup/inital delay, the number of times that the buffer underrun during the media stream, the amount of quality switches and the media throughput rate. While existing techniques divide the available bandwidth over the clients according to priority or the amount of streams, the first part of this thesis presents a development of bandwidth distribution logics for video multistreaming with MPEG-DASH. MPEG-DASH is developed for adaptively streaming media over HTTP. The four parameters to improve the QoE of the users are taken into account when developing the distribution logics.

Omni-directional video provides a new way to demonstrate 360 degree video of a scene. In this setup, it is the intention to show only a particular part of the 360 degree video. The user can change the visible part of the 360 degree video by interacting with it. By tiling the 360 degree video, we can save bandwidth for the hidden tiles. The second part of this thesis will relate to tiled video with MPEG-DASH. The extension of MPEG-DASH, called SRD, is used to provide spatial information about the tiles to the clients. We present six bandwidth distribution logics for tiled video. The user can interact with the 360 degree video by panning in it. Therefore the hidden tiles can be visible in a short period of time. We have made different distribution logics to divide the bandwidth over the tiles according to different approaches. For example, one distribution logic focuses on the quality of the 360 degree video and ensures that the quality of the 360 degree video is the same for all the tiles. Other approaches focus on the particular part that is visible for the user. Because we use MPEG-DASH, the quality of the video can change during the stream by requesting media segments of a higher or lower quality representation. This technique of adaptively streaming of media is used by the distribution logics to select the quality of video for the tiles.

We organized a user test with 14 particpants to subjectively and qualitatively measure the QoE of the users with MPEG-DASH tiled video streaming. We selected four distribution logics of the six and changed the bandwidth and the segment duration for every test case. We measured the interaction of the variables with the perceived values with a three-way ANOVA test. A conclusion and future work on the basis of the outcomes of the user test is given to end this thesis.

Foreword

This master thesis is the last part of my 2 master years at the UHasselt as a student. In this foreword I want to thank all the people that supported, monitored and/or trained me and believed in me to succeed in my studies.

First of all I want to thank my promotor, doctor professor Peter Quax, to give me the opportunity to do research on this subject and to support me during the period of my research. Hereby I also want to thank doctor Maarten Wijnants as my mentor who has supported, monitored, trained and corrected me when needed to do research on this topic.

Next to the people for my master thesis, I thank the professors of the UHasselt who have educated me during my study period. Without them I would not have had the knowledge to do research for this master thesis.

Aside from the people of the UHasselt, I thank my parents to give me the opportunity to study at the UHasselt and to support me financially and mentally. Hereby I also thank my girlfriend for being there in some difficult periods and to support me in my decisions.

To end this foreword, I want to thank my fellow students for helping me out when needed.

I wish you a lot of pleasure while reading this master thesis.

List of abbrevations

UHasselt	Hasselt University
RTP	Realtime Transport Protocol
RTCP	Real Time Control protocol
RTSP	Realtime Streaming Protocol
RSVP	Resource ReSerVation Protocol
MPEG-DASH	Moving Picture Experts Group Dynamic Adaptive Streaming over HTTP
MPD	Media Presentation Description
ODV	Omni-Directional Video
XML	eXtensible Markup Language
QoE	Quality of Experience
SRD	Spatial Relationship Description
URN	Uniform Resource Name
HAS	HTTP Adaptive Streaming
GoP	Group of Pictures
DOM	Document Object Model
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
OSMF	Open Source Media framework
CDN	Content Distribution Network

WiFi	Wireless Fidelity
QFF	QoE Fairness Framework
RoI	Region of Interest
MCU	Multipoint Control Unit
HEVC	High Efficiency Video Coding
RAM	Random Access Memory
SSD	Solid State Drive
LED	Light Emitting Diode
GB	Gigabyte
CDF	Cummulative Distribution Function
PTZ	Pan, Tilt and Zoom

List of Figures

Contents

Chapter 1

Introduction

1.1 Problem statement

In a world where almost every device is connected to the internet, the utilization of the *available bandwidth* in a network plays an important role. Bandwidth is the amount of data that can be transferred over a network in a time period, measured in either bits or bytes per second. The past years there has been a big revolution in network capacity and *network throughput*, but the *network bandwidth* remains scarce, especially in mobile and wireless networks. Because network bandwidth is limited, applications must use it thoughtfully.

In video streaming scenarios the problem is the same, video content is captured and stored on the server. To view the video content, the user must download the content and therefore consumes the available bandwidth in the network.

In this master thesis the focus is placed on video streaming with *MPEG-DASH*. To know what MPEG-DASH is, read Chapter 2. When streaming video to a client computer, the content is streamed in one *quality* at a certain bitrate. When the bandwidth is enough for the stream, there is no problem. But when the bandwidth drops and is below the bitrate necessary for the video stream, the delivery of the video is delayed. The user must wait until the video segment is downloaded and can be displayed. MPEG-DASH solves this problem by *dynamicly* and *adaptively streaming* content over HTTP.

Bandwidth in *dynamic networks* like wireless and mobile networks can fluctuate a lot. This is due to people continuously joining and leaving the network, or it is caused by applications consuming non-constant amounts of bandwidth. When using MPEG-DASH in these situations, there are other problems. MPEG-DASH ensures that the viewer can watch a video without experiencing the *playback to stall*, however at the cost of *reduced video quality*. When bandwidth is changing frequently, the video quality will also change frequently.

The *Quality of Experience (QoE)* is a measure of a customer's experience with a service [70]. The service in this master thesis is video streaming. Like the website of Bitmovin [23] explains, there are four parameters that influence the QoE with video streaming. These four parameters are:

1 **Initial/start-up delay:** The QoE is lower when the user has to wait a long period before the video starts playing.

2 **Buffer underruns/stalls:** The QoE is lower when the playback stalls because this will cause the video to temporarily freeze.

3 **Quality switches:** The QoE is lower when the quality of the video is changing frequently because this shows a change in quality multiple times.

4 **Media throughput:** The media througput is measured in bits per second. The lower the media throughput, the lower the quality of the video will be. This will lower the QoE.

So the QoE may be negatively impacted by one of the four parameters. In this master thesis we are focused on these four parameters. To reduce the initial/start-up delay we use the quick start method. This method ensures that an image can be displayed as soon as possible to the user. This is done by downloading a lower quality of the video as fast as possible. Because a lower quality has a lower bitrate, the data is downloaded faster than is the case with higher quality video.

By using MPEG-DASH, we can ensure that the buffer will never underrun and the playback will never stall, at the expense of visual quality. Because we are switching between qualities, there will be always video content in the buffer. The full explanation for this is done in Chapter 2.

The downstream throughput in a network is the amount of bits or bytes that the client computer receives within a time period. The throughput is measured in bits per second or bytes per second. How faster the data can be delivered to the client, the higher the quality of the video. Because how faster the data is delivered, the more data the client can receive within a time period. If the client can receive more data, the quality of the video is better because higher quality video is larger in file size than lower quality video.

When the quality is changed frequently, the QoE is lower than when the quality will not change too often. Dynamic networks play an important role in this. Dynamic networks are networks where the available bandwidth per client can change frequently, due to network link congestion, people continuously joining and leaving the network, etcetera. Because of the changing bandwidth available for the clients, the quality of the video can change frequently while streaming.

This problem is the main study of this master thesis. The aim of this master thesis is to implement an MPEG-DASH framework and to do a literature study of the problem of bandwidth distribution. The framework consists of multiple parts that together make a complete video player with bandwidth distribution logics attached to it. The implementation is explained in Appendix B.The framework has also been coupled to ODV tiled content, see Section 4.6.

In *video multistreaming* applications, the use of MPEG-DASH could improve the performance of it. Video multistreaming means that videos are streamed in parallel to the client. During the streams, the quality of the videos can vary due to changing bandwidth. It might be a good idea to give streams *priority* above others or divide the available bandwidth *fairly* across the streams.

With *ODV tiled streaming*, a 360 degree video is displayed to the viewer, the 360 degree video is referred to as *omni-directional video*. The meaning of ODV tiled streaming is that the viewer sees a spatially restricted part of the ODV and that the viewer can look arround in the ODV. An additional reason can be that the physical screen of some devices, for example low cost smartphones, has a limited resolution. As such, the omni-directional video cannot be displayed integrally on the screen. The viewer has a viewport that shows a particular part of the ODV and panning is enabled to view other content of the full video frame. The details and purpose of ODV are given in Chapter 4.

To stream such a high resolution video, the video is divided in multiple *tiles*. Every tile has its own bandwidth need and therefore the *bandwidth distribution logics* are required. Because only the tiles contained in the viewport are displayed to the user, it might be a good idea to allocate less bandwidth to the *hidden tiles* compared to the visible tiles. The problem here is when the viewer pans a lot in the video, the hidden tiles are displayed, but at a lower quality and therefore the QoE of the viewer drops. The implementation of the distribution logics are explained in Appendices H to M.

1.2 Research questions and outline of thesis

The goal of this master thesis is to implement a basic MPEG-DASH framework for video multistreaming and ODV tiled streaming, which is discussed in Appendix B. The basic implementation includes all the necessary classes to start streaming video to the video player. For video multistreaming and ODV tiled streaming the framework is extended with classes that are required for every application.

Every stream has its own bandwidth need related to the context. Because bandwidth is limited in networks, the distribution of bandwidth will have an impact on the QoE of the viewer. Since the distribution of bandwidth is highly related to the application in which it is used, we formulate the following research questions in two types of applications.

1 **Video multistreaming:** What impact will every distribution logic have on the bandwidth allocation and on the QoE of the viewer in a dynamically changing environment involving multiple media streams?

2 **ODV tiled streaming:** What impact will every distribution logic have on the QoE of the viewer and what are good distribution logics depending on contextual factors?

In this master thesis the focus is more on ODV tiled streaming. The literature study for video multistreaming will give us the best approach for this type of application. For ODV tiled streaming we have done user testing to subjectively and qualitatively measure the performance of every bandwidth distribution logic.

The rest of this chapter will give an introduction of MPEG-DASH. To answer the above research questions, we will do a case study for video multistreaming in Chapter 3. Chapter 4 will introduce ODV with SRD and will discuss related work. To evaluate the distribution logics for ODV tiled streaming, we have done user testing. The user testing scenarios and results are explained in Chapter 5. The conclusion is given in Chapter 6, to end this master thesis. We will identify the best bandwidth distribution logic for every situation we have defined for ODV tiled streaming and formulate the future work for video multistreaming and ODV tiled streaming applications.

Chapter 2

MPEG-DASH

MPEG-DASH stands for Moving Picture Experts Group *Dynamic Adaptive Streaming over HTTP*. MPEG [1] is a group consisting of people of the business world and people from academia. This group engages in the development of standards for coding and decoding video and audio content and in other topics like metadata representation with MPEG-7 [6], network communication for MPEG-DASH, etcetera.

This group of people have made a new standard for adaptive streaming, namely MPEG-DASH [76]. Streaming video over the internet is nowadays popular and bandwidth distribution plays a role in this scenario. MPEG-DASH is a standard that uses HTTP [62] to deliver media content. In Figure 2.1 the scope of MPEG-DASH is shown. We can see that Media Presentation Description, MPD parser and Segment parser fall within the scope. The control heuristics and media player are controlled by the client. MPEG-DASH delivers basic components, the implementation of the video player can be vendor specific. Therefore the client must implement the logic to switch between qualities, this is supported but not implemented in the standard.

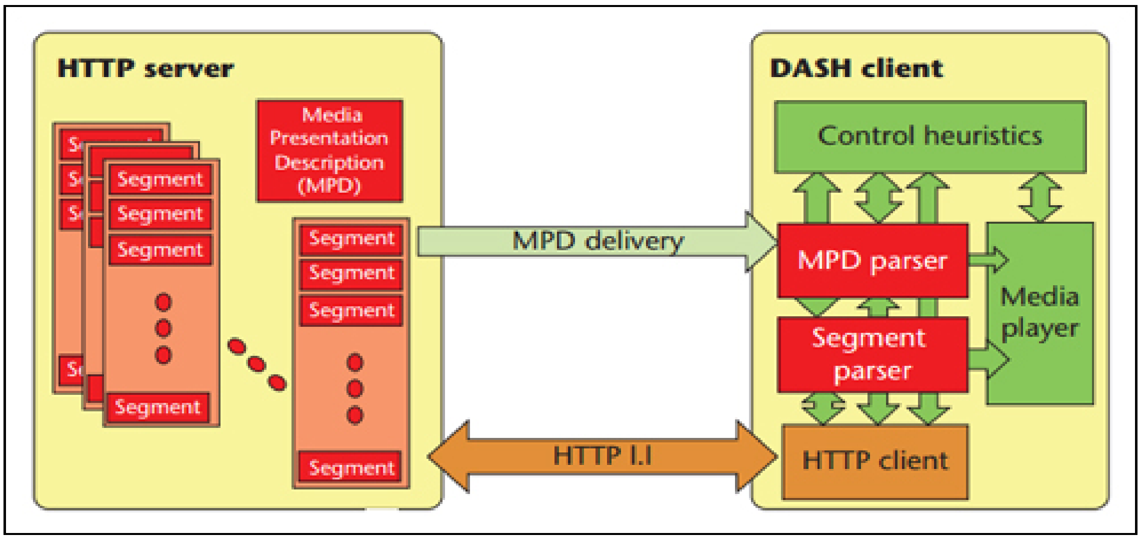


Figure 2.1: MPEG-DASH scope [10]

In a network where multiple users, named clients or viewers in this master thesis, are connected, the bandwidth is divided among all the users. The bandwidth can fluctuate a lot in scenarios like wireless networks, mobile networks, etcetera because many parameters can change the available bandwidth, like for example the number of users connected to the network.

MPEG-DASH focuses on *live* and *on-demand* streaming of media content. Live streaming is when media is recorded and immediately sent to the viewers. On-demand streaming means that the media is stored on the server and that the viewers can download the segments when they like to. With live streaming the viewer downloads the segments captured at that moment. The viewer cannot pause the media or seek in the media. With on-demand the viewer decides which segments it downloads. Here the viewer can rewind the media because the segments are stored on the server and the viewer decides which segments to download. In this master thesis we will focus on on-demand streaming of video content.

When users are receiving media content in dynamic networks, the users are obtaining one quality of the media. When the available bandwidth for the users is smaller than the needed bandwidth for the media, the playback will be jittery. This is due to the available network bandwidth. The available bandwidth is not enough for the media stream and therefore the delivery of the media is delayed. As such, it takes longer to download media parts at the currently available bandwidth than it does to consume them, which introduces playback freezes. The media parts are named as *media segments* in this master thesis.

To solve this problem, the MPEG group has developed MPEG-DASH. MPEG-DASH works with multiple qualities of the media and divides every quality in media segments of a specific duration, measured in seconds.

MPEG-DASH requires that the viewer downloads the media segments in a specific quality from the server. The client decides which quality it downloads based on characteristics like the available bandwidth, terminal capacities, mobile battery percentage, etcetera.

First the client must know which qualities at which bitrates the server provides. This information is communicated by means of an MPD file. The format and parameters of an MPD file are explained in Section 2.3 and Section 2.4, respectively.

When the client knows the available bandwidth and received the MPD file from the server, it can choose which quality to download. When the bandwidth changes, the client can change qualities on the fly. Because the media is hosted in multiple qualities and every quality is segmented in consecutive segments of a specific temporal length, the user can download segments of other qualities.

When downloading a higher or lower quality segment of the media, the viewer is likely to experience a quality change. When changing qualities, it will take some time before the change will be noticeable for the viewer. This is because the previously buffered media segments must finish their playback before the higher or lower quality media segment starts playing. The smaller the playback buffer size and/or the duration of the segments is, the faster the quality change happens.

This quality change is caused by the changing bandwidth available to the client. Because MPEG-DASH is developed for this, the client can still watch the video without experiencing any delay. A disadvantage for the user is that he/she can experience quality changes.

A big advantage of MPEG-DASH is that it is a standard that can be used by a lot of devices connected to the internet. It works over HTTP, this is because nowadays almost every device has a web browser.

For example Smart TVs, smartphones, tablets, etcetera all support HTTP and hence support media streaming over HTTP. MPEG-DASH supports multiple form factors of devices by including parameters in the MPD file. For example, a mobile device's resolution is much lower than a HD TV's resolution. MPEG-DASH is a good approach for heterogeneous consumption contexts, because different content resolutions can be defined in the MPD file.

There are not only advantages for the client devices, but also for servers. A firewall [63] is a system which contains a set of rules that allows or blocks network traffic to computers in the network and to computers outside the network. The advantage for a firewall to use MPEG-DASH, is that it does not have to define a specific rule for media streaming. When HTTP traffic is allowed to enter in the network, the media content can be received by clients in the network. If HTTP traffic is allowed to leave the network, the media content can be distributed to client computers. For HTTP, the destination port is 80, so no additional port must be assigned for the traffic exchange, which is beneficial for NAT [60] devices. A NAT device is a system which maps private internal network addresses to public external network addresses and vice versa.

2.1 Before MPEG-DASH

Before MPEG-DASH was developed, streaming protocols, like RTP [74] and RTSP [75] were used to stream media to the clients. RTP is a transport protocol which is built on UDP [71] and is designed for real-time transfers. When using UDP, it is not sure that the data will arrive. Stated differently, UDP does not offer a reliable transmission channel. RTCP was used in combination with RTP to provide feedback on the quality of the data distribution. RTCP is a control protocol, which is also discussed in RFC 3550 [74], that provides feedback, synchronisation and other features for the media stream. The protocol does not send any media content, it transports information, like the delay, jitter and other network properties experienced by the receiver in the course of the streaming session [51].

RTSP [75] uses RTP to deliver media content. RTSP is an application level protocol that is used to establish and control media sessions between clients and media servers. With this protocol, clients typically send commands to the media server, like play and pause, to control the playback of media content [50]. With MPEG-DASH, the client controls the playback of media. The client does not send commands, like start and stop, to the server to control the playback. This will be advantageous for the server because it does not need to keep the resources for the client reserved and it will also reduce network traffic. RTSP uses RSVP [56] to reserve resources accross the network path on which the media content is transported. RSVP is an IPv4 and IPv6 transport layer protocol that provides resource reservation for unicast and multicast topologies [52].

In contrast with HTTP, RTSP is a stateful protocol. A stateful protocol ensures that session information, referred as the state, is stored on the server. HTTP is stateless what means that no information about the session is stored on the server and that every request is treated independently. The state typically contains the identifier of the session to track concurrent sessions and other parameters. RTSP requires more processing power and storage space from the server than HTTP because of the stored state for every client session.

MPEG-DASH does not define information about the network properties, like RTCP does with RTP. With MPEG-DASH, the clients decide which media segments are downloaded and therefore they determine in which order the media segments are played. All the information about the media segments and the needed bandwidth are defined by means of an MPD file (see section 2.3 and 2.4). The client decides on the basis of the available bandwidth in which quality it downloads which media segments.

Therefore measuring the available bandwidth is necessary for the clients.

With RTP, separate protocols (RTCP, RTSP, RSVP) are used to provide information about network properties, to control the playback of the media content and to reserve resources across the network path. With this approach, a firewall must open all the necessary ports for these protocols in order to make them work. This approach is less secure than with MPEG-DASH because it has more open ports that can be abused. With MPEG-DASH, only the standard HTTP port (80) is used to stream media content.

With RTP streaming, UDP is used for transferring data, while TCP is used with HTTP streaming. The advantages of UDP are that it delivers the content fast and that it supports unicast and multicast transmission of media. The disadvantages of RTP are that the transmission is not reliable and adaptive streaming is not directly supported. HTTP streaming only supports unicast transmission and introduces delay because of the reliable transmission of media over TCP. With TCP, there is congestion control which is not included in UDP. With HTTP streaming, TCP ensures that the client receives the data in order and that no data is missing. Because RTP uses UDP, the data is rapidly delivered but data can be missing or is not received in sequence. When the data is not received in sequence, the client must properly order the content. This requires extra processing power and buffering for the client. With RTP, data that is not received is skipped by the video player. In Table 2.1, the comparison is made between RTP/UDP streaming and HTTP/TCP streaming that is used by MPEG-DASH.

Property	**RTP/UDP streaming**	**HTTP/TCP streaming**
Transport protocol	UDP	TCP
Supported topologies	Unicast, multicast	Unicast
Delay	Fast delivery	Delay introduced by delivering media with TCP
Reliable transmission channel?	No	Yes
Adaptive streaming supported?	Not in standard version of RTP	Yes
Separate protocol needed to control quality of media?	Yes	No
Firewall friendly?	No	Yes
Congestion control?	No	Yes

Table 2.1: Comparison of RTP with MPEG-DASH

Another technique, that is often used, is *progressive downloading* [36] of media. Progressive downloading can be used in networks where the bandwidth is not sufficient to download the entire media file. With progressive downloading, the media may begin playback before the download is complete. The video player needs information about the data it downloads. It needs to know how many data must be downloaded before playback can start. This information is communicated via metadata in the form of a header that is prepended to the data. The advantage of progressive downloading is that the client can download as many data as the network can handle. It allows the user to play the media while the file is being downloaded to the client. The disadvantage for this approach is that the metadata must be added to the data to stream the media. This will introduce extra overhead and consumes bandwidth. The setup is more complicated than with the other approaches and it continues to download data when the users are, for example, looking at other web pages and pause the video stream. The video will continue to download in the background and will slow down their connection. When downloading the video file, the playback can freeze as a result of low bandwidth. This can happen when the video files are downloaded slower than they are consumed. In constrast with MPEG-DASH, MPEG-DASH ensures that the video keeps playing when the available bandwidth reduces, but at cost of reduced video quality.

To solve the problems of previous streaming protocols and progressive downloading, MPEG developed MPEG-DASH. Like said before in this chapter, MPEG-DASH supports adaptive streaming of media over HTTP. The problems of streaming protocols, like not being firewall friendly, having no reliable transmission channel and having no support for adaptive streaming are resolved by the MPEG-DASH standard. A problem with MPEG-DASH is that it does not support multicast. A solution for this problem is given in Section 3.3.1. The problem of adaptive streaming in previously discussed protocols is resolved by MPEG-DASH by encoding multiple qualities of the media. The problem of downloading media in the background with progressive downloading is also resolved by MPEG-DASH because the video segments are not downloaded when the video pauses [42].

2.2 What is an MPD file?

An MPD file is an XML file that describes all necessary parameters for the client to be able to receive different qualities of the media. An XML [65] file is a file where parameters, attributes, values and other types of data are mentioned in. It is used to communicate information to devices that can read the XML format. This file is easy to read for both human and machine.

An MPD file is needed to start streaming because the client retrieves all the media data on basis of information included in the MPD file. The MPD file is downloaded first and the client downloads the media segments defined in the MPD file at a later point. All parameters of the particular media quality segments are described in the MPD file. In Section 2.4 a subset of these parameters are discussed.

2.3 MPD format

An example of an MPD file with one *period*, one *adaptation* set and two *representations* is shown next.

```
<?xml version="1.0"?>
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT4.000000S"
type="static" mediaPresentationDuration="PT0H0M46.97S"
profiles="urn:mpeg:dash:profile:full:2011">
        <ProgramInformation
        moreInformationURL= "http://gpac.sourceforge.net">
                <Title>template.mpd generated by GPAC</Title>
        </ProgramInformation>
        <BaseURL>
                http://ip-address-of-server/rootdirectory/subdirectory/...
        </BaseURL>
        <Period duration="PT0H0M46.97S">
                <AdaptationSet segmentAlignment="true"
                bitstreamSwitching="true" maxWidth="600" maxHeight="200"
                maxFrameRate="30" par="600:200">
```

```
                    <Representation id="1" mimeType="video/mp4"
                    codecs="avc1.640015" width="600" height="200"
                    frameRate="30" sar="1:1" startWithSAP="1" bandwidth="65536">
                            <SegmentBase>
                                    <Initialization sourceURL="template_init.mp4"/>
                            </SegmentBase>
                            <SegmentList duration="2">
                                    <SegmentURL media="segment_1_1.m4s"/>
                                    <SegmentURL media="segment_1_2.m4s"/>
                                    <SegmentURL media="segment_1_3.m4s"/>
                                    ...
                            </SegmentList>
                    </Representation>
                    <Representation id="2" mimeType="video/mp4"
                    codecs="avc1.640015" width="600" height="200"
                    frameRate="30"sar="1:1" startWithSAP="1" bandwidth="131072">
                            <SegmentBase>
                                    <Initialization sourceURL="template_init.mp4"/>
                            </SegmentBase>
                            <SegmentList duration="2">
                                    <SegmentURL media="segment_2_1.m4s"/>
                                    <SegmentURL media="segment_2_2.m4s"/>
                                    <SegmentURL media="segment_2_3.m4s"/>
                                    ...
                            </SegmentList>
                    </Representation>
            </AdaptationSet>
    </Period>
</MPD>
```

2.4 MPD parameters

In this section we describe all the important parameters of the MPD file that was shown in Section 2.3. These parameters are necessary to understand the rest of the master thesis and are referred to in other sections. We limit our discussion to MPD files that rely on *segment lists* to describe segment URLs. This is the type of MPD file that this master thesis is focused on. MPEG-DASH supports also other segment specification formats, like *template* MPD, *segment range* MPD, etcetera. This example is for on-demand media and focuses on video content. MPEG-DASH supports more than this, namely live streaming of video content, audio content streaming and many other types. Therefore this list of parameters is not an exhaustive emumeration of all the parameters of MPEG-DASH, but only the parameters that are necessary to understand this master thesis. Please see [12] for a complete overview of the MPD syntax.

XML version
The version used of the eXtensible Markup Language.

<MPD>
The start tag of the MPD file to define where the MPD data begins.

minBufferTime
This is the minimum amount of data, expressed as a `duration` data type [12], that the client-side buffer must hold before playback is allowed to commence. This is to ensure that there is enough data to play the video smoothly. This field consist of the following letters:

P: Duration designator placed at the start of the duration representation.
T: Time designator that precedes the time components of the representation.
H: Hour designator to define the amount of hours.
M: Minute designator to define the amount of minutes.
S: Seconds designator to define the amount of seconds.

mediaPresentationDuration
The duration of the full media presentation. This field consists of the same letters as the `minBufferTime` attribute.

profiles
The MPEG-DASH profile used by this MPD file. The profiles are defined to enable interoperability and to delineate the set of supported features. The profile is identified by unique uniform resource names in the MPD.

<ProgramInformation>
Descriptive information on the program is provided for a Media Presentation within the `ProgramInformation` element.

<BaseURL>
The base URL stands for the location where the content is stored. So this includes the IP-address of the server and the path to the parent directory where all media segments and initial segments are saved.

<Period>
The media is divided in one or multiple temporal parts, called *periods*. Periods temporally follow each other in the case there are multiple parts.

duration
The duration of a specific period, expressed in the same syntax that is also exploited by the `minBufferTime` attribute.

<AdaptationSet>
Set of interchangeable encoded versions of one or several media content components.

<Representation>
A *representation* is a single quality of the media in the adaptive streaming experience.

mimeType
The type of the content that is carried in the representation.

codecs
The codec that was used to encode the content.

width
The width of the video, measured in pixels.

height
The height of the video, measured in pixels.

framerate
The framerate of the video, measured in frames per second.

bandwidth
The bandwidth needed for a specific representation, measured in bits per second.

<SegmentBase>
This element is used to identify the first file needed to start viewing the media. This includes the initialization element that is specified in the `sourceURL` attribute. This `sourceURL` is the URL for the initialization file. The initialization file does not contain any media. It contains values to change the settings of the video player. To download the file, the client adds the `sourceURL` to the `BaseURL` to know the full URL of the initialization file.

<SegementList>
This element clusters all the media segments that jointly constitute a specific representation.

duration
The duration of a single segment, measured in seconds.

<SegmentURL>
For every media, segment there is a segment URL.

media
This parameter is part of the full URL of every media segment. Media segments are different from initialization files as they contain the media itself. To download each segment, the client takes the `BaseURL` and adds the `media` attribute value to it and downloads the file from the obtained URL.

2.5 Content preparation

The content used for this master thesis is video content. Video content can be captured in different ways, by a camera, by computer programs, etcetera. A video consists of multiple frames and therefore the size of video content can be large. To store and transport data in an efficient way, the data is encoded with a codec. A codec [45] is a device of computer program capable of shrinking the media data in size by encoding it. It is used to send the media data over a network, store or encrypte the data. The media data can be obtained by decoding the encoded data for playback or editing.

There are different types of codecs. In this master thesis, the x264 codec is used to encode video streams into H.264/MPEG-4 AVC format [9]. x264 [7] is a free software library, it is released under the terms of the GNU General Public License.

The videos used in this master thesis are encoded in multiple qualities with FFMPEG [13]. FFMPEG is used to convert the video content encoded with a specific codec into different qualities. High quality videos typically are larger in size than lower quality videos. Therefore FFMPEG takes a long time to convert high quality videos because of the large content size. We convert the videos using the command terminal and the FFMPEG command. To use FFMPEG after installing it, we can execute the FFMPEG command in the Linux terminal with parameters added to it. The command options we used for FFMPEG are given in Appendix A.

We recommend that the GoP size is a multiple of the frame rate of the video. The GoP size determines how many inter-frames are between consecutive I-frames. The first I-frame is counted within the GoP size. Because if the previous video segment is missing, the videoplayer searches for the first I-frame of the following video segment to play. We will now show an example to make things clear. Assume that the video segments have a duration of 1 second, the GoP size and frame rate are 5 and that we only have I- and P-frames. So we have 5 frames per second played and there are 4 P-frames between 2 consecutive I-frames. This gives the following situation:

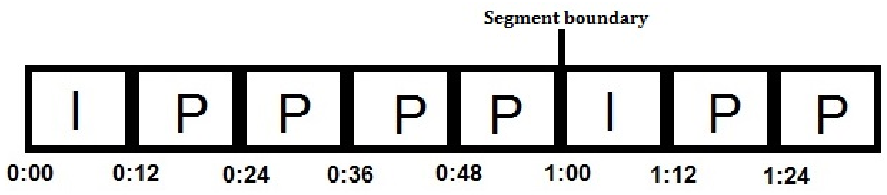


Figure 2.2: Video with GoP size of 5 and frame rate of 5

The plackback time is added under the boundaries of every frame, measured in seconds, so 1:00 presents 1 second. In this situation there is not a problem. Because the GoP size is a multiple of the frame rate, the video will not jump forward when the video segment with playback time 0:00 until 1:00 is missing, referred to as the fist video segment. When the first video segment is missing, the video player searches for the next I-frame that starts at 1:00. So the video will continue to play from 1:00. Now let's assume that the video segments have a duration of 1 second, the GoP size is 7, the frame rate is 5 and that we only have I- and P-frames. We get the following scenario:

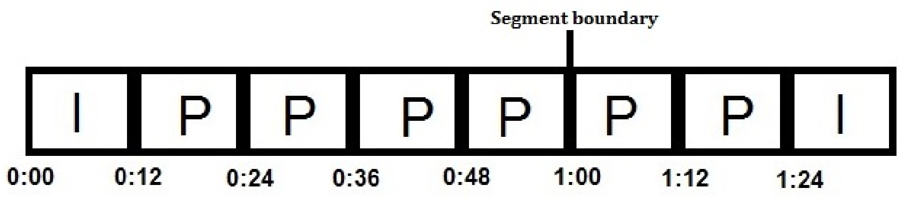


Figure 2.3: Video with GoP size of 7 and frame rate of 5

In this scenario the video player will jump forward when missing the first video segment. Because the player will search for the next I-frame, it starts at 1:24 instead of at 1:00. It skips the P-frames that are in between 1:00 and 1:24. This is the reason why we recommend that the GoP size must be a multiple of the frame rate. The fact that prior segments are missing can occur when the user is seeking in the video or with ODV tiled video streaming.

MP4Box [5] of GPAC [3] is used for preparation of HTTP Adaptive Streaming content. GPAC is an open source multimedia framework. GPAC focuses on multimedia packaging formats such as MP4 and on presentating technologies, like graphics, animation and interactivity. The core library of GPAC is libgpac. The tools that are provided by GPAC are: a multimedia player (Osmo 4/ MP4CLIENT), a multimedia packager (MP4Box) and server tools that are included in MP4Box.

We use MP4Box to generate the MPD files for every video and to split the video into multiple parts of a fixed duration, these are called *segments*. In this master thesis the videos are split into segments with a duration of 1 and 2 seconds. Like FFMPEG, MP4Box can be used by the MP4Box command with parameters added to it and the command options are given in Appendix A. FFMPEG and MP4Box are the tools used for this master thesis but other tools offering similar functionality could have been used to generate the files and content.

The video content is stored on an HTTP server. The HTTP server in this master thesis is a Linux Ubuntu 14.10 desktop computer with Apache 2.4.7 [22] installed on it. Linux Ubuntu is a Debian-based Linux operating system, with Unity as its default desktop environment. It is based on free software. Apache is the world's most widely used web server software.

The advantage of HTTP Adaptive Streaming is that it only requires an HTTP server to start streaming media. Because the media is stored on the HTTP server and the server only responds to requests from the clients, it is easy to set up and to maintain the server. The client sends requests for the content by an HTTP GET request to the full Uniform Resource Loader and receives the content afterwards. A URL [55] is a reference to a resource that specifies the location of the resource on a computer in the network. With the URL, the resource can be retrieved.

Chapter 3

Multistreaming with MPEG-DASH

3.1 Definition

A large percentage of the current internet traffic consists of media streaming data. More and more websites like YouTube, Netflix, Twitch, etcetera have on-demand and live media streaming features. Media multistreaming scenarios are common in reality, like people listening to music and in parallel watching a movie. In this master thesis we will focus on on-demand video multistreaming.

Every video player that is streaming media has its own parameters like the total duration, resolution, necessary bandwidth, etcetera. With MPEG-DASH these parameters are all defined in an MPD file, a separate MPD file is generated for every video. The MPD syntax has been discussed in Section 2.3.

A lot of applications have implemented a manner of video multistreaming. The example of this master thesis is ODV where the 360 degree video is divided into tiles. Every tile is a separate video that is streamed to the client. To perceive the tiles as a 360 degree video, the videos of the tiles are synchronised with each other. An example is shown in Figure 4.10. The full explanation of 360 degree video is given in Chapter 4.

Video conferencing also uses video multistreaming features. For example, with Skype [40] users can stream videos of multiple people that are calling. An example of this application is given in Figure 3.1.

The last past years, Google has successfully developed the Android operating system and applications for smartphones and tablets. Because smartphones are getting more and more processing power built in, they are capable of multitasking. Multitasking means that the device can execute multiple tasks at the same time. Some applications, like Stick it Pop-up Player and other alternatives are examples of multitasking applications. Stick it was designed for smartphones and tablets running the Android operating system to stream multiple videos at the same time to one client device. Stick it is an application where the user can see videos while doing some tasks with the device in the background. For example, the user could watch a movie on YouTube and Twitch while searching for something on the internet by using the webbrowser. In Figure 3.2 a screenshot is given of the Stick it application.
In the past, on-demand video streaming websites recorded the videos and stored them on a server. The server then adds the necessary metadata and sends it to the clients. This costs a lot of resources of the server, because the server must constantly stream the videos to the clients who have subscribed to receive the stream. In this normal situation the server does multiple jobs, like recording, encoding, storing, and sending the data to the clients, this is a big load for the server.

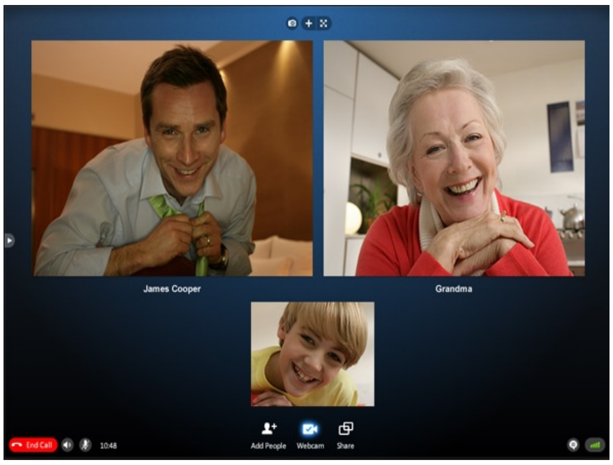


Figure 3.1: Skype video multistreaming [11]

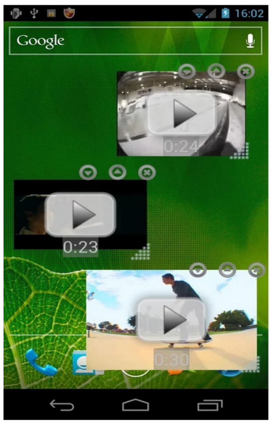


Figure 3.2: Stick it Pop-up Player[15]

To solve this problem, the involved tasks are divided over multiple servers. For our example, one server would record the video. A second server would encode and save the data that it receives from the first server and gives the resulting data to a third server who would add the necessary headers and send it to the clients. An approach to send the data to the clients is that the third server subscribes to a multicast group where the clients are subscribed to. After subscribing to the multicast group, the server sends the data to the multicast group. All the clients that subscribed to the multicast group will receive the data. An example that uses this idea is the transport layer of Akamai for live streaming in a content delivery network [69].

Because this method and other similar methods will introduce a lot of processing for the routers, this approach is not often used. The routers need to keep track of the multicast groups for the clients connected to that router and need to do the processing of the Internet Group Management Protocol. IGMP is a protocol that establishes multicast group memberships over the internet between hosts and routers on IPv4 networks [57].

A better approach for this idea is to make use of MPEG-DASH. The advantages of MPEG-DASH are that it is a standard and that there is no need for an additional protocol for streaming, like RTP [74] or RTSP [75]. MPEG-DASH is used in this master thesis to stream videos to the clients. MPEG-DASH is not only made for streaming video content, the standard implementation of MPEG-DASH supports media multistreaming. For example when a video is streamed, the audio is streamed in parallel to the client. When the clients use MPEG-DASH, the clients first request the MPD file and the initial file of every media. They decide which quality of the video they want to download based on their available bandwidth and based on the necessary bandwidth of a particular quality of the video. This parameter is mentioned in the MPD file by the `bandwidth` parameter (see section 2.4). When streaming video content, the clients can change the quality of the video on the fly by requesting segments of other qualities.

The mostly used technique to measure the available bandwidth is to send a file over the network and keep track of the download time. This is just a simple approach to measure the available bandwidth. A lot of parameters play a role to determine the available bandwidth, like jitter, router processing time, etcetera. An easy way to calculate the throughput is

$$throughput = \frac{L}{T}$$

where L is the file size and T the transmission time [67]. There are a lot of bandwidth measuring tools that do the calculations and take the necessary parameters into account, for example Cacti [2] is such a tool that is widely used by network administrators. For this master thesis, the focus is not on measuring the bandwidth. We artificially set the bandwidth limit and we can hereby determine which quality is streamed to the clients.

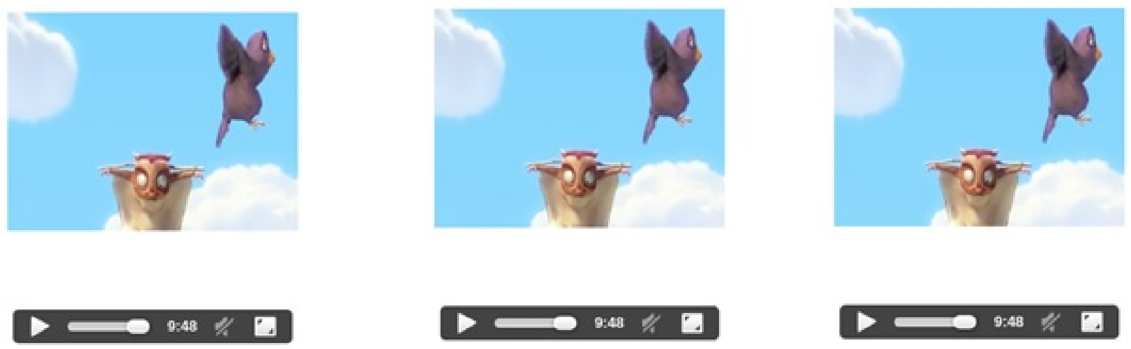


Figure 3.3: Video multistreaming example

In Figure 3.3 an example of video multistreaming involving three instantiations of the Big Buck Bunny video clip [8] is given. We see that the three videos are streamed in the same moment to the client and they are paused at the same playback time.

3.2 Implementation MPEG-DASH framework for video multistreaming

To stream video to the clients, it is the intention to make an MPEG-DASH framework. We extended our framework for video multistreaming. In this section we will generally describe the components and functionality of the framework for video multistreaming. We made the framework in a way that every component can be replaced for other needs. The framework consists of multiple classes which each of them can be replaced by other classes.

The main component of our framework is the `Downloadmanager` class, which ensures that every stream starts streaming. This implies that for every stream the MPD file and initiale video file are downloaded. All the files are downloaded with an `HTTPDownloader`. Every stream has an `HTTPDownloader` object that downloads the files for that stream through HTTP. After the stream has downloaded the initial file, the `Downloadmanager` creates a `Scheduler` object for every stream. The `Scheduler` triggers the `Downloadmanager`, on an interval basis, to download the next video segment for a stream. The interval time is the segment duration, measured in seconds. We have chosen to download one video segment per segment duration because in that time period the last played segment ends its playback. If we download one video segment when the last played segment finishes its playback, the buffer stays stable and does not underrun. The `Downloadmanager` gives the request for a video segment to the `HTTPDownloader` for the specific stream to download the next video segment. When the `Downloadmanger` receives the file from the `HTTPDownloader`, it adds the file to the `MediaSourceBuffer`. The `MediaSourceBuffer` maintains the `SourceBuffer` [41] object that stores the video segments for playback.

To manage the available bandwidth, the `Downloadmanager` has a distribution logic object. This object ensures that the available bandwidth is divided among the streams. When the available bandwidth changes, the distribution logic ensures that the allocated bandwidth per stream decreases or increases. The streams decide on the basis of their allocated bandwidth which quality of the video is downloaded. Like section 2.4 explained, the needed bandwidth for every quality of the video is communicated via the `bandwidth` attribute in the MPD file. The quality of the video is selected of which the needed bandwidth is less or equal to the allocated bandwidth for the stream. The functionality of the `QualityAdaptation` object is that it selects the quality of the video according to the allocated bandwidth for the stream. One `QualityAdaptation` object is made for the `Downloadmanager`. The `QualityAdaptation` object performs the quality selection for all the streams in the multi-streaming setup. To return which quality of the video the stream can play, the `QualityAdaptation` object reads the MPD files of the streams and selects the highest quality for the stream's allocated bandwidth. The full explanation of all the classes of the framework can be found in Appendix B.

3.3 Related work

In the past years, streaming media content over the internet has become popular. Several streaming protocols were proposed for streaming the media and to control the transmission. When using these streaming protocols, some problems were discovered, like adaptivity, reliable transmission, etcetera. In this section we discuss related work and some usefull topics for video multistreaming with MPEG-DASH.

3.3.1 Cache servers and CDNs

HTTP streaming of media does not support multicast streams. For this problem a lot of solutions have been proposed. In this section we discribe one such commonly used solution and relate it to MPEG-DASH. Figure 3.4 shows a distribution architecture for HTTP-based streaming involving mobile devices. This architecture is called a CDN [44]. A content delivery network (CDN) is a network of distributed servers, connected with each other, that delivers web content to a user. The network consists of origin servers, cache servers and clients. The origin servers provide the content while the cache servers store copies of the content. The cache servers are located nearby the users to provide quick access to the content.

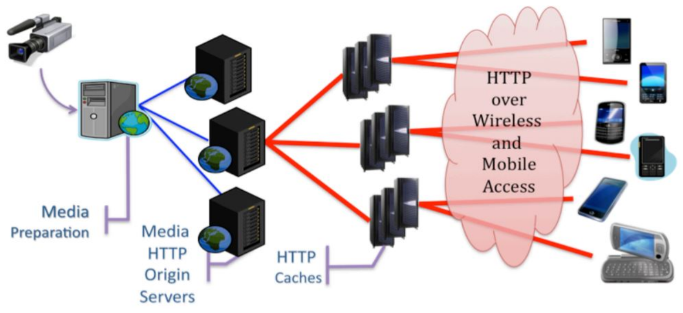

HTTP over Wireless and Mobile Access
Media Preparation
Media HTTP Origin Servers
HTTP Caches

Figure 3.4: Media distribution architecture [77]

The server does the media preparation and encodes the video into multiple quality versions. The segments and MPD files are hosted on the media HTTP origin servers. In the networks between the origin servers and the client devices, HTTP cache servers are placed. These cache servers download the video segments from the origin server once. They are constantly polling the origin servers to find out whether the data has changed. If so, they request the new data and store it. The clients typically request the data from the HTTP cache servers that are located nearby. A longer distance for delivering data will introduce more latency and intermediate links can cause the media throughput to be lower.

This approach is a solution for the multicast problem of HTTP streaming. With multiple HTTP cache servers are installed in networks, one copy of the data can be distributed to multiple client devices with one cache server. An advantage of this approach is that the HTTP cache server only needs to download one copy of the data and stores it for the clients. The media throughput is high because the clients are requesting the data from the cache server that is located nearby. When the data transfer is fast, more data can be downloaded by the clients and a better quality can be streamed. The content can be delivered faster which will decrease the initial/startup delay. Therefore the QoE of the user is higher when using a CDN to distribute the content. An example of a CDN for media streaming is made by Akamai [69].

Bandwidth can be saved with CDN architectures because the links between the origin servers and the cache servers are not congested with multiple copies of the data. Only one copy of the data is downloaded by the cache servers. When the data is modified, the data is again downloaded by the cache servers. This is bandwidth efficient because the data is not frequently sent over the link if the data is not modified on the origin server. This technique has minor overhead because the messages exchanged between the cache server and the origin server consume only small amounts of bandwidth. The origin server sends the new data back when the data is modified or answers to the cache server that the data is not modified. When the data is not modified, no content is sent to the cache server.

3.3.2 Dividing bandwidth over clients in public-shared networks

In cities where the population is large, FON routers are installed. FON routers [25] are cheap WiFi access points that connect the users to the internet. The meaning of this approach is to connect as many people as possible to the internet through the FON routers. On basis of this idea, we formulate an approach for public-shared networks. PSnet is such an approach that uses FON routers. The infrastructure of PSnet is shown in Figure 3.5.

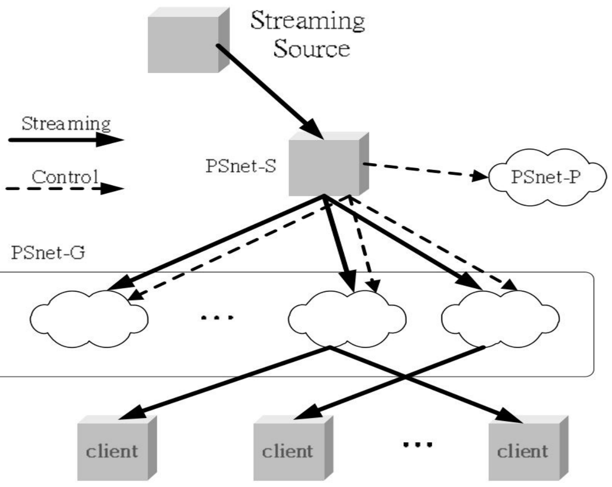


Figure 3.5: PSnet of FON routers for public-shared networks [66]

The network consists of one PSnet-S server that manages the whole system. The streaming source is typically a server that streams the media content to the interested clients. The PSnet-G architecture delivers the streams to the clients and is composed of serveral groups of organized access points, referred as PSnet-Ns, and a backup pool of access points, referred as PSnet-P. Clients wishing to receive the media stream are connected to one PSnet-G access point. As we can see on Figure 3.5, the system is represented by a tree structure.

The PSnet-S constructs the PSnet-G by organizing the PSnet-Ns to groups based on the requests from clients. These PSnet-Ns are also organized as a complete binary tree structure. In Figure 3.6, we see such a tree structure with 16 clients connected and two subtrees where each circle represents one PSnet-N access point.

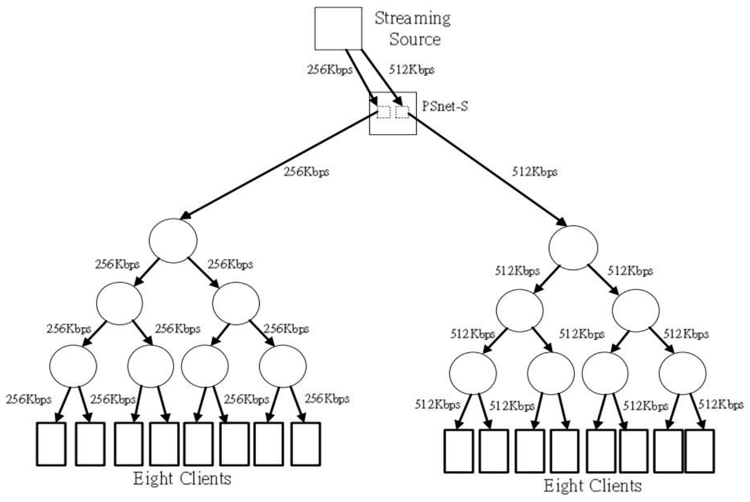


Figure 3.6: PSnet with two subtrees serving media to the clients [66]

In this setup, two clients are served by the same PSnet-N access point. In our example, the right subtree has a link speed of 512 Kbps and the left subtree has a link speed of 256 Kbps. When media content is streamed to the clients, the root node of the subtree replicates the media content to all the nodes which are located on the path to the interested client.

When the intermediate links of the subtrees are congested, the delivery of media content is delayed. An approach to solve this problem is to add more PSnet-N access points for delivering the data to the clients. In this case the client receives data from multiple PSnet-N access points. The backup pool of PSnet-N access points is used to add more subtrees to the tree structure. Multiple PSnet-N will deliver parts of the stream to the client. The setup is shown in Figure 3.7.

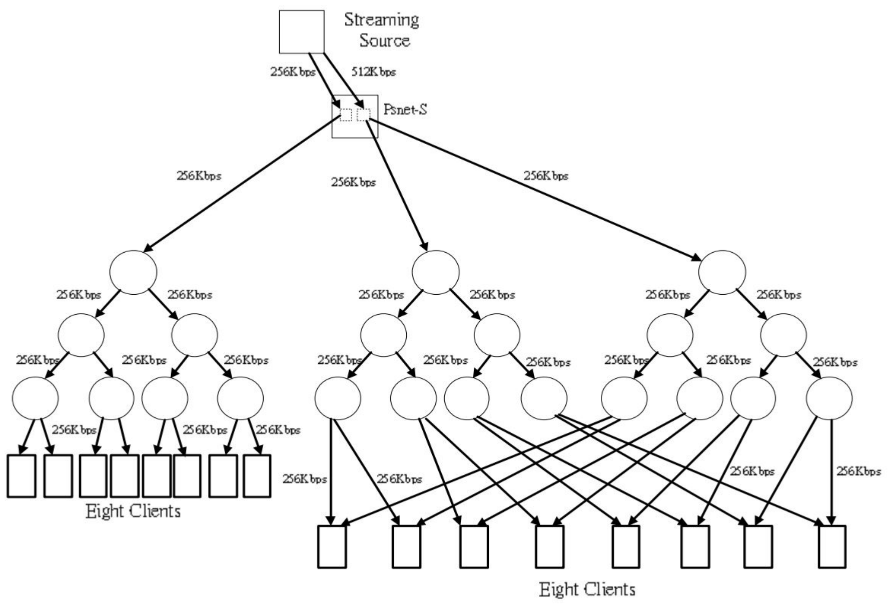


Figure 3.7: PSnet with three subtrees serving media to the clients [66]

This setup will allow the clients to receive media parts from multiple PSnet-N access points. The disadvantage of this approach is that the client must properly order the received parts. The advantage for the client is when a link is congested, the client can receive media parts from other PSnet-N access points. This will result in efficient use of the available bandwidth. Because MPEG-DASH works with media segments, every PSnet-N access point can deliver one or multiple media segments to the client. The client must properly order the media segments for playback.

This approach can be used in networks where multiple access points are installed and multiple clients are streaming media. When the system determines that links are congested, the PSnet-G can add more subtrees to the tree structure. The subtrees consist of multiple PSnet-N access points that deliver the media to the interested clients. In this case the clients are receiving media segments of multiple PSnet-N access points. This can increase the speed of media throughput for the clients when intermediate links are congested. When the media throughput is increased, more data can be delivered to the clients. The more data can be deliverd to the clients, the better the quality of the video. This will result in a better QoE for the users.

3.3.3 QoE Fairness Framework

In this section we introduce the OpenFlow-assisted QoE Fairness Framework (QFF) [64] that works with MPEG-DASH. The framework maximises the users' QoE in multimedia networks and allows vendor-agnostic functionality to be implemented for network management and active resource allocation. QFF monitors all the DASH clients in a network and dynamically allocates network resources to each device. This ensures that the QoE of the streams is optimised to achieve the maximum user-level fairness [64]. The QFF framework is typically installed on a central network device that passes traffic to the clients. The framework is shown in Figure 3.8.

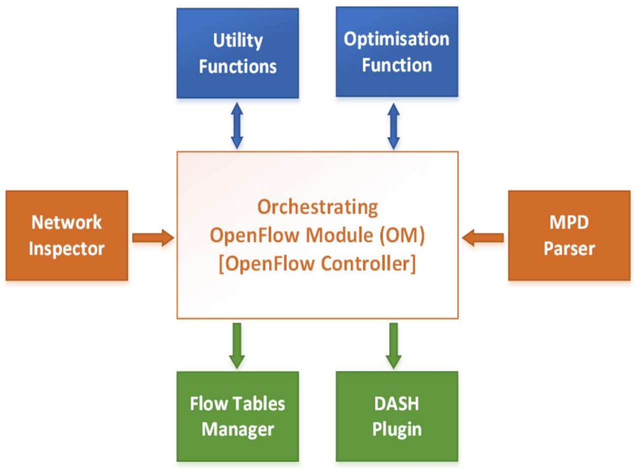


Figure 3.8: QoE Fairness Framework [64]

The Network Inspector informs the OM core of the number of clients streaming media in het network, the streaming bitrate each device is currently requesting and the available bandwidth capacity. The MPD Parser informs the parameters and attributes in the MPD files requested by the clients, like the segment duration, available encoding bitrates and the size of the media segments, to the OM core. To communicate the information to the OM core, the Network Inspector and the MPD Parser use the OpenFlow protocol [49].

The Utility Functions map the bitrate of a video to the QoE perceived by the user. Structural Similarity (SSIM) index is used as the quality metric for the Utility Functions. SSIM calculates a value for the similarity between two images. The value, referred as the SSIM index, is a decimal value between -1 and 1. A higher value results in more similarity between two images. It uses the initial uncompressed version of the image as the reference frame [53] and the image encoded at a certain bitrate to compare with each other. A database consisting of a Utility Function per video would need to be constructed to store the SSIM indexes for every version of one video encoded at a certain bitrate. The Optimisation Function finds a combination of all the bitrates for all the streaming videos in the network that result in equivalent QoE level for all the clients according to the SSIM indexes. Please read paper [53] to known how such optimum set of bitrates for all the clients is selected. The intention of this framework is to provide approximately the same QoE for every active stream in the network.

The Flow Tables Manager adds the appropriate streams to the OpenFlow switches, so that each client receives the media segments. The OpenFlow switches are responsible for delivering the media content to the clients. The DASH plugin is used to inform all the DASH clients of the bitrate that they should request to achieve network-wide QoE fairness [64]. QFF can also be used for other streaming technologies and is not restricted to DASH.

This approach can also be used in video multistreaming scenarios. Because in those scenarios the user is playing videos in parallel, the framework can be used for this. The total bandwidth is the available bandwidth for the client. The framework could divide the bandwidth among the streams to achieve single user QoE fairness between the streaming videos. A disadvantage of this approach is that for every video a database must be managed to store all the SSIM indexes and it would take more processing power than other approaches to determine which quality of video is requested for every stream. An advantage is that the quality change would occur only if the Utility functions determine that the allocated bandwidth is reduced or increased for a stream. When the user changes between streams, like in previous cases where the user streams multiple videos in parallel in different tabs, the quality of the streams that were active before the change (and that remain active after the change) will remain constant. In this case the QoE is higher for the user because the quality change less often. Because the bandwidth is divided fairly among the streams based on the QoE level of every video, the quality could be lower than with priority-based streaming. With priority-based streaming, the streams with high priority typically are streamed at higher quality. The downside of this is that streams with a lower priority can be streamed at a lower quality than in the case a uniform distribution of bandwidth is used. The QFF is similar to our approach of equal distribution of bandwidth, discussed in Section 3.4.1. Our approach will divide the bandwidth equally over the active streams. In this approach we can not assure that the QoE is approximately the same for all the active streams. Because videos can be streamed at different bitrates, equally dividing the bandwidth can not assure the qualities with the same QoE is chosen for all the active streams. With the QFF this is assured but takes more processing power and more network traffic to communicate all the necessary information.

3.3.4 Client based bandwidth management

Bandwidth management plays an important role when streaming media with MPEG-DASH. On the basis of the available bandwidth the clients decide which quality of the media is downloaded. However, so far we have discussed the bandwidth management issue in a network where multiple clients are streaming media simultaneously. We will now discuss the bandwidth management issue when the client is streaming media in parallel. To divide the allocated bandwidth for one client over the streams it is interested in, we explain the approach of the NIProxy [79]. NIProxy divides the allocated bandwidth for a client over the streams it receives.

The NIProxy works with a tree structure to allocate bandwidth to the streams the client is interested in. The tree structure consists of multiple nodes, each of a special type. The internal nodes of the tree structure have a certain bandwidth distribution strategy. The leaf nodes are the streams in which the client is interested.

The first type of internal node is the *mutex* node. The child nodes of this node compete for the available bandwidth allocated to their parent node. This ensures that at all times at most one child node is assigned bandwidth. The child node that gets the bandwidth is the node that requests less or equal bandwidth than the mutex node has available. If there are no child nodes that satisfy this constraint, none of the child nodes get any bandwidth.

Another type of internal node is the *priority* node. The child nodes of this node have priorities assigned. According to these priorities, the bandwidth from the parent node is assigned to the child nodes. The child node with the highest priority gets the requested bandwidth if there is enough bandwidth available. The residual amount of bandwidth is allocated to the child node with the second highest priority. This continues untill there is no more bandwidth.

A third type of internal node is the *weight* node. This node operates in two phases. In the first phase the bandwidth is divided over the child nodes accoring to their maximal bandwidth consumption and their weight value. Every child node receives $BW_i = w_i * MaxBW_i * f$ bandwidth. w_i stands for the weight of the node, the value is minimum 0 and maximum 1. $MaxBW_i$ corresponds to the maximum bandwidth the child node can consume and f stands for a scaling factor. The scaling factor ensures that the sum of all the allocated bandwidths for all the child nodes is not larger than the allocated bandwidth of their parent node. When the child nodes are not consuming the bandwidth they were allocated, the weight node performs a second bandwidth distribution phase. In this phase the excess bandwidth is assigned to child nodes on a one-by-one basis, in order of decreasing weight value [79]. In this phase unused bandwidth is allocated to nodes which can consume it.

The last type of internal node is the *percentage* node. Every child node of this node has a percentage assigned. This percentage determines which amount of bandwidth every child node gets. The assigned bandwidth is calculated via $p_i * BW$ with p_i representing the percentage of the child node and BW representing the assigned bandwidth for the parent node. When the parent node determines that some bandwidth remains unused, it behaves like the weight node in phase two. It allocates the excess bandwith to the child nodes in order of decreasing percentage value.

The next point to discuss is the type of the leaf nodes. A first type is the *discrete* stream hierarchy leaf node. This node sets the stream's bandwidth consumption to a discrete number of values. A simple type of this node has two consumption levels, namely 0 and the associated stream's maximum bandwidth usage. With this, the node is capable of turning the network stream on and off. The node can support a discrete number of increasing bandwidth levels according to the quality of the media that is requested.

A second type of leaf node is the *continuous* leaf node. This node is capable of setting a stream's transmission rate to a continuous range of values. These values are lying in the interval [0, maximum stream bandwidth usage]. It does this by buffering the content locally and forwarding the content at a bitrate that is reserved for this node.

We explained the basic understandings of the nodes in the tree structure for this approach. NIProxy can operate with real-time and non real-time network traffic. For real-time network traffic the discrete leaf node can be used and for non real-time network traffic the continuous leaf node can be used. In Figure 3.9 a simple example of the tree structure is shown.

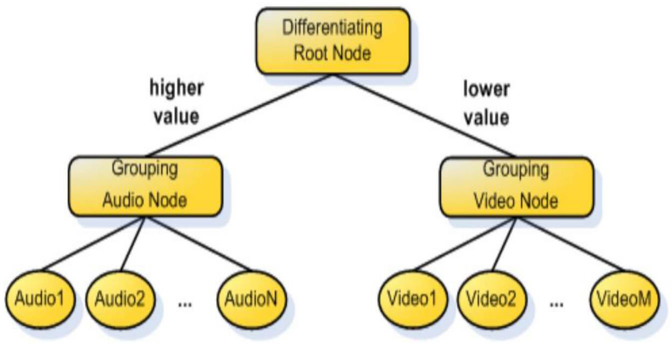


Figure 3.9: Simple example of tree structure of NIProxy [79]

To stream video with MPEG-DASH, the priority and the discrete stream hierarchy nodes are the appropriate to use. With the priority node, we can give streams more priority than others. This will result in more allocated bandwidth and better quality of video for those streams. For example, when users are watching multiple videos in parallel in different tabs of the web browser, the streams that are in the active tab can have more priority than the streams in the inactive tabs. With active, we mean that the tab is currently open and the others are closed but the videos of those tabs are still playing.

The discrete stream hierarchy node can set discrete values for every quality of the video. In the MPD file, the `bandwidth` attribute stands for the needed bandwidth to stream a particular quality of the video. The discrete stream hierarchy node can maintain all the values of the `bandwidth` attribute for all the video qualities of one stream. When the priority changes for the leaf nodes (discrete stream hierarcy nodes), the quality of the video can change according to the allocated bandwidth. With this approach, we can dynamically change the quality of the video without experiencing the screen to freeze. The disadvantage, like said before, is that the quality will change during the video playback because the playback of the previous segments must finish before the lower or higher quality video segments start playing. NIProxy was developed for streaming protocols, like RTP. They provide continuous streams of media data. DASH works with discrete downloads of segments and does not continuously stream media. Because of this difference, the NIProxy is not optimally suited for the management of bandwidth with MPEG-DASH streams.

3.3.5 User-adaptive video streaming with MPEG-DASH

When using MPEG-DASH for streaming to mobile devices, an important topic is user-adaptive video streaming. This topic is about investigating the user behavior to improve the efficiency of streaming delivery [73]. The approach relies on sensors of mobile devices to detect the presence of the user, while other types of techniques are used to detect the context. Figure 3.10 illustrates these ideas.

The visual field and viewing angle of the user is determined. The visual field is the field in which the user can immediately interact but does not interact directly with it. The viewing angle determines what the user sees. The mobile device keeps track of information about the user with sensors, like the front-facing camera, accelerometer and gyroscope to detect the presence, proximity and viewing angle of the user. All processing is done on the mobile device. The mobile device knows by means of the MPD file all the information about the videos it can receive.

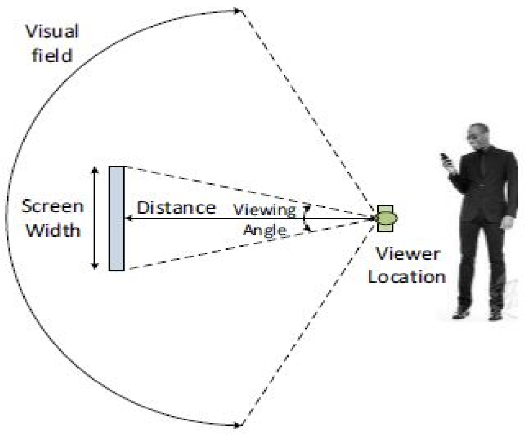


Figure 3.10: Proximity of the user to the mobile device screen

The mobile device decides on the basis of the perceived information from the sensors which quality or which video segments must be downloaded. For example, when the sun is shining on the screen and the sensors detect that the user can not see much of the video because of the reflection of the sun, the quality of the video can be reduced. Afterwards, when the sensors detect that the reflection of the sun is gone, the quality can be upgraded. Other examples are when the user is sitting far away from the mobile device or is not constantly looking at the screen. In such cases the mobile device can stream the video at a low quality or possibly even stop streaming the video. This approach could improve the usage of the mobile device. Large mobile devices have large screens that take up a lot of capacity of the battery. With this approach we could extend the battery life, reduce the usage of bandwith and improve the QoE of the users.

The sensors can detect, for example in scenarios where multiple videos are played in parallel on the screen, which video(s) the user is watching. On the basis of this information, we can detect which video(s) require(s) more bandwidth so that they might be streamed at a higher quality. The user can see only a fraction of information projected on the screen, by this we can improve the QoE for the users. For mobile devices with a small screen, this is not applicable because all the parts of the screen fall in the user's viewing angle. The fraction of the screen that the user is watching can be streamed at higher quality and the other parts can be streamed at lower quality or not streamed at all. In this way, the bandwidth consumption efficacy is improved.

We can relate this idea to our approach with a webpage or application where multiple videos are played in parallel. Mobile devices with large screen will typically have more advantages than devices with small screens. For this approach, we can use priority-based video streaming. When mobile devices detect that the quality of the video can be reduced due to the information perceived by the sensors, the priority of the stream can be reduced. This will result in less allocated bandwidth for the stream and a lower quality of video. When the mobile device sensors detect that the quality must be increased, the priority of the stream can be raised. This will cause the video quality to be better.

For mobile devices with small screens, there is a restriction, like said before, that the full screen falls in the viewing angle of the user. With larger screen we can apply the viewing angle technique. The videos, that fall in the viewing angle of the user, can have higher priority than the other ones. Our approach to divide the bandwidth among the clients according to priorities is discussed in Section

3.4.3. We divide the available bandwidth according to the priorities, expressed in percentages, to the clients. The higher the percentage, the more bandwidth is allocated and the better the quality of the video. For devices with small screens, we can apply other techniques with information perceived from the sensors. Like when the sun is shining on the screen, we can decrease the priority for the streams and when the screen is clearer, we can increase the priority of the streams. A prerequisite is that the sensors should work well because on the basis of their information the quality will be higher or lower.

3.4 Experimental evaluation

The performance of video multistreaming depends on the operation of a *scheduler* component. The scheduler is responsible for deciding when to download a video segment for a video player. The scheduler will first schedule as many video segments to download as the available bandwidth can handle. When the buffer has enough segments cached, the player can start playing. Afterwards the scheduler keeps scheduling one video segment to download per time interval corresponding with the segment duration, measured in seconds. This scheduler is called a buffering/steady scheduler [54]. So every segment duration, a video segment is downloaded. Therefore the buffer's fill level will not decrease. Aside from the scheduler, we will focus on *bandwidth distribution logics* for video multistreaming. Let's assume in our case that the available bandwidth is 250 000 bps and that we have three video qualities as specified in Table 3.1.

Quality	Needed bandwidth, measured in bits per second
1	45 652
2	89 283
3	131 087

Table 3.1: Example of three qualities of video with their bitrates

3.4.1 Equal distribution of bandwidth

A simple but good approach for a bandwidth distribution logic is to divide the available bandwidth equally among the video players streaming on that moment. Normally the equation of the bandwidth per stream is:

$$\frac{totalbandwidth}{amountofvideoplayerscurrentlystreaming}$$

The distribution logic must take the number of active video players into account. So when video streams start playing or stop playing, the distribution logic must do the calculation again by considering the new amount of active streams. This approach is also applicable for video players pausing the stream and resuming the stream. With this approach we have made a distribution logic for video multistreaming. The implementation details are explained in Appendix D. In this section we want to give an example of streaming with this distribution logic and show how the distribution logic will react when video players start streaming. In Figure 3.11, the downloaded data for the first video player is measured.

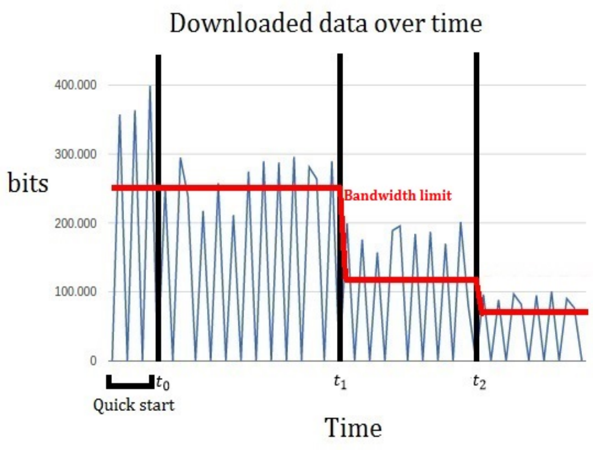


Figure 3.11: Multistreaming with equal bandwidth distribution

We will now explain the graph in detail. In the graph, we see that the peaks are higher than the bandwidth limit. This is because we did not limit the network speed on the network card. In our example, we have two second segments. We simulate the amount of downloaded bits per segment duration, in our case per two seconds. Because every peak will drop to 0 in each segment duration interval, the bitrate per two seconds will not exceed the bandwidth limit. In a real-world situation, the consumed bandwidth will stay below the bandwidth limit and the peaks will not drop to 0. Also, please note that the size of the media segments can be slightly larger or smaller than the value dictated by the `bandwidth` attribute defined in the MPD file. This is because the target bitrate is averaged over the segment boundaries. Averaged, we get the target bitrate. The minimum buffer time is expressed in video segments. Like Section 2.4 explained, this is the minimum amount of data, expressed as a duration data type [12], that the client-side buffer must hold before playback is allowed to commence. In our case the video segments have a duration of 2 seconds. The minimum buffer time is 10 seconds. To buffer 10 seconds, we must download 5 video segments. The client downloads one segment extra to make sure that the buffer keeps having 5 video segments stored when playing.

- Quick start: In the quick start period, the intent is to download video segments of a lower quality to fill the buffer as quickly as possible. Because we want to display a decent quality, we have chosen to download not quality one but quality two when we start downloading. In the graph we see that the bandwidth consumption is above 300 000 bps. Because the video segments are two seconds long, the video player receives bandwidth for two seconds. Because the video player gets bandwidth for two seconds, it must wait two seconds to request the next video segment. That is the reason why the curve goes to zero after each peak. This gives us the following equation:

  allocated bandwidth per second * video segment duration (expressed in seconds)

  So the video player gets 500 000 bps allocated (250 000 * 2). For that amount, the video player can download two video segments in quality two. The bandwidth needed for the stream is 89 283 bps and the video segments have a duration of two seconds. So to download one segment, we will need (89 283 * 2) = 178 566 bps. Because 89 283 bps is the bandwidth needed for a video segment of one second, we must multiply it by two for a two seconds video segment. We have an available bandwidth of 500 000 bits per two seconds, so we can download two video segments every 2 seconds. The consumed bandwidth is (178 566 * 2) = 357 132 bits, like shown as the first peak in the graph.

- t_0: After the quick start period, the buffer is filled with 6 video segments to start playing. Because the video player plays the video segments after each other, we can download one video segment when the currenlty playing video segment ends its playback to keep the buffer's fill state stable. After t_0, we see that the bandwidth consumption peak drops below 300 000 bps. That is because we download one segment of quality three. The needed bandwidth for this video segment is 262 174 bps. Because we have segments of two seconds, we must multiply the needed bandwidth per second (131 087 bps) by two. We see that the peaks approximately reach 262 174 bps in the graph.

- t_1: At this time a second stream starts playing and downloading video segments. We see that the allocated bandwidth for the first player drops to 250 000 bits per two seconds. 250 000 bps is calculated via:

  (available bandwidth * segment duration) / amount of video players = (250 000 * 2 / 2) = 250 000 bits per segment duration

  We can see this in the graph: the curve will not go higher than 200 000 bps after t_1. The available bandwidth is no longer sufficient for quality three and therefore video segments of quality two are downloaded.

- t_2: At t_2 a third stream starts playing. This again results in a drop in available bandwidth per stream:

  (available bandwidth * segment duration) / amount of video players = (250 000 * 2 / 3) = 166 667 bits per segment duration

  In this case we see that, after t_3, the curve will not go higher than 100 00 bps. The quality is also dropped because with 166 667 bits per segment duration video segments of quality two can not be downloaded anymore. So now the video player downloads one segment of quality one. The bandwidth needed for quality one is 45 652 bps. The video segments have a duration of two seconds, so one video segment is maximum (45652 * 2) = 91 304 bits in size.

3.4.2 Changing bandwidth during streaming

Allocated bandwidth for every player can change when more video players start streaming or video players stop streaming. The bandwidth capacity itself can change, which will also impact the amount of bandwidth that is allocated to individual video players in a multistreaming context. Let's assume that the available bandwidth is equally distributed over the streaming video players, like in Section 3.4.1, and that the bandwidth capacity is variable. We will show an example for one video player in Figure 3.12.

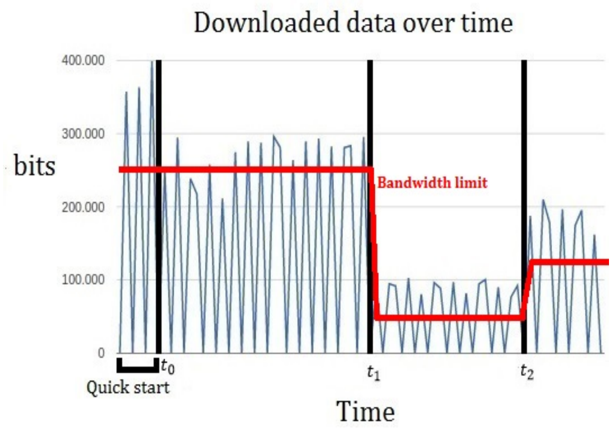


Figure 3.12: Multistreaming with changing bandwidth and equal bandwidth distribution

In this figure we also have the quick start period and the time when the player starts playing = t_0. We will not explain these matters again because we have already explained them in the previous example. We will explain the other times:

- t_0 to t_1: Steady state with quality two (see Section 3.4.1).
- t_1: At this time, the bandwidth changes from 250 000 bps to 50 000 bps. Because the segment duration is two seconds, the available bandwidth is also calculated for two seconds. The total bandwidth the player has allocated is 100 000 bits per segment duration (50 000 * 2). For this amount of bandwidth, we can download one video segment of quality one (45 652 bps). The consumed bandwidth is 91 304 bps (45 652 * 2). We see that the peaks after t_1 and before t_2 are not higher than 100 000 bps.
- t_2: The bandwidth changes from 50 000 bps to 125 000 bps at t_2. We can download 250 000 bits per segment duration (125 000 * 2). For this amount of bandwidth, one video segment of quality two is downloaded. The total consumed bandwidth for downloading one video segment of quality two is 178 566 bps (89 283 * 2). We see in the graph that the peaks after t_2 are not higher than 250 000 bps.

3.4.3 Priority-based distribution of bandwidth

The last topic for video multistreaming we will discuss, is priority-driven distribution of bandwidth. This distribution method gives priorities to streams, expressed in percentages. On the basis of the priorities, it decides which amount of the available bandwidth every stream gets. The percentage per stream determines the amount of allocated bandwidth for a stream, so 100% will give a video player all the available bandwidth and 0% will give a video player no bandwidth at all. In our example, we have given one video player 50% priority and the other two video players each 25% priority. In Figure 3.13 the downloaded data for the high priority video player is shown. In this scenario a second player will join on t_1 and a third on t_2.

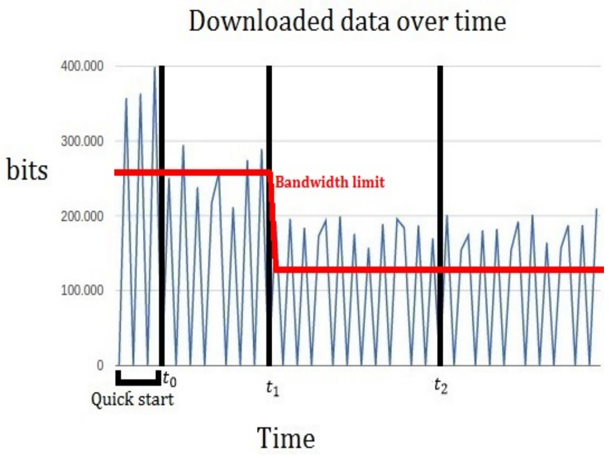


Figure 3.13: First example of multistreaming with priority-based bandwidth distribution

For the three times t_0, t_1 and t_2, this gives us the following explanation:

- t_0 to t_1: Steady state with quality two (see Section 3.4.1).

- t_1: At this time, the second video player starts streaming, so the allocated bandwidth per stream is recalculated. Because the measured video player has a priority of 50%, the video player still receives 125 000 bps. For this amount of bandwidth, the video player can download quality two (89 283 bps). Because we have two second segments, the video player can consume two seconds of it's allocated bandwidth. The allocated bandwidth is 250 000 bits per segment duration (125 000 bps * 2). The bandwidth needed for a two seconds video segment of quality two is 178 566 bps (89 283 * 2). We see after t_1 and before t_2 that the peaks are not higher than 200 000 bps.

- t_2: A third video player starts streaming, the allocated bandwidth for the measured video player does not change, because it has 50% priority. This means that the video player receives 50% of the available bandwidth. The video player hance keeps downloading quality two of the video.

In the previous example we saw that the priority is high for the measured video player and that the allocated bandwidth for that video player stays the same when the third stream becomes active at time t_2. Now we will investigate a second scenario. We will give video player one 25% priority, video player two 50% and video player three 25%. The downloaded data over time for video player one is shown in Figure 3.14.

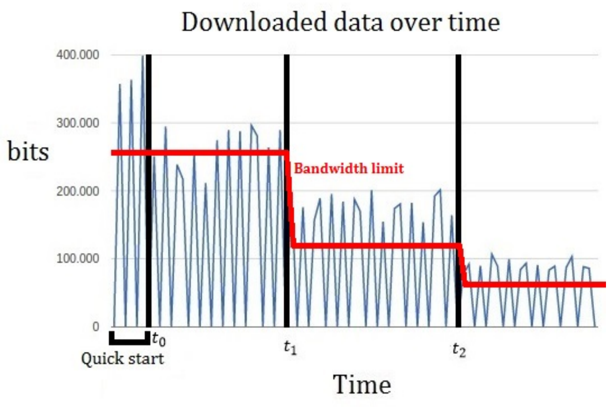


Figure 3.14: Second example of multistreaming with priority-based bandwidth distribution

We will now explain t_1 and t_2.

- t_1: The second video player starts streaming. The second video player has a priority of 50%, so that video player receives 125 000 bps of the available bandwidth. Video player one has a priority of 25%, but because there is not any other streaming video player, this video player will also receive 50% of the available bandwidth. So the allocated bandwidth for video player one is 125 000 bps. With 125 000 bps, the video player can download quality two of the video. The consumed bandwidth for one video segment of quality two is 178 566 (89 283 * 2). We see that the peaks are not higher than 200 000 bps after t_1 and before t_2.

- t_2: At this time, video player three starts streaming. Because video player two has a priority of 50%, it receives 125 000 bps of the available bandwidth. Video player one has a priority of 25%, so it receives 62 500 bps and this is also the case for video player three. With 62 500 bps allocated, video player one can download quality one of the video (45 652 bps). The consumed bandwidth is also the needed bandwidth for quality one multiplied by two, so the value is 91 304 bits per segment duration (2 * 45 652).

Our implementation differs slightly from the presented results. The difference lies in the case when video players start streaming. In the previous example, videoplayer one got 50% of the total bandwidth and when video player two started streaming, this one got also 50% of the total bandwidth, even though the latter had only 25% priority. To make this more dynamic, we have implemented a logic that will give the actively watched video players twice the priority of the inactively watched video players. Like said in previous sections, with active we mean that players are directly watched while inactive players play video on the background. So in this case when video player one is playing and video player two starts playing, video player one gets 66,66% priority and video player two gets 33,33% priority. When video player three starts playing, video player one gets 50% priority, video player two and three get each 25% priority. The implementation of this logic is discussed in Appendix E. We did not generate any graphs of the final implementation because these will be similair to the results in this section.

For videos that are played in parrallel, like is the case in ODV tiled streaming, the equal distribution of bandwidth among videos can apply in a simple case. This approach however is not used much for ODV tiled streaming because the relative importance of the tiles can differ greatly from each other. We therefore have taken other approaches for ODV tiled streaming. Please read Section 4.6 for more information.

For example, in scenarios where users have multiple videos playing in parrellel in different tabs of the web browser, the priority distribution of bandwidth can apply. Because only one set of video streams can be watched at the same time with tabs in the web browser, the other video players are not watched. So the video streams that are watched actively, get the highest priority. When the user switches between tabs, the users gets a lower quality image of the video to see because the video had a lower priority before. When the distribution logic for priority distribution of bandwidth determines the change between tabs, the active video player get more bandwidth and the quality is upgraded after some time.

3.4.4 Rate adaptation for adaptive HTTP streaming

A problem with streaming media with MPEG-DASH is that the quality of the video can change quickly due to changing bandwidth. Therefore current video players have different approaches for streaming video, they use different rate-adaptation algorithms. Rate-adaptation means that the speed of data transfer is adjusted in order to ensure the integrity of the datastream, matching the conditions of the medium where the datastream is transferred over [19]. Therefore the rate-adaptation algorithm determines the quality of the video that is played. The algorithm takes the available bandwidth into account when determining the quality.

A simple approach for the rate-adaptation algorithm ensures that the quality changes when the available bandwidth is not suffucient to stream the currently playing quality of the video. In this case when the available bandwidth is lower the bitrate for the quality of the video that is currently playing, the quality is reduced. The quality of the video that is selected, is the quality with the bitrate that is lower than or equal to the available bandwidth. This approach ensures that the client-side buffer will not underrun because the available bandwidth is enough to download the quality of the video.

The available bandwidth of a computer connected to a network can change over time, caused by a lot of things. For example, computers joining or leaving the network, applications consuming non-constant amounts of bandwidth, etcetera. In our case the available bandwidth for a client device changes when the user is watching more or less videos in parallel. Two popular video players, namely the Smooth Streaming player of Microsoft and the Nextflix player are discussed. We will investigate their rate-adaptation algorithms and describe their mode of operation. We explain how they react to positive and negative bandwidth changes. The following explanations are made with help of the evaluation of rate-adaptation algorithms paper [54]. This is a paper from 2011, so there is a chance that the results are not completely representive any more.

3.4.4.1 Smooth Streaming player

The Smooth Streaming player of Microsoft [34] is a popular player for audio and video content. Smooth Streaming is another approach of HAS like MPEG-DASH but it is vendor specific. It uses a Silverlight application on the client for dynamic bitrate switching. So the algorithm for switching between qualities of the video is vendor specific. Like MPEG-DASH, it also uses a manifest file which includes information about the segments, like the codecs used, which bitrates and resolutions are available, a list of the available chunks and their start times and duration, etcetera. The big different with MPEG-DASH lies in the manner in which chunks, in our case segments, are stored on the server.

With MPEG-DASH, the individual media segments are stored on the server and these can be requested directly by the client. Smooth streaming uses a different approach, it uses two formats, namely the wire format and the disk format [20]. The video is stored in full length on the hard drive of the server. The contiguous file is sent in a series of chunks to the clients. The wire format defines the structure of the chunks that are sent to the clients and the disk format defines the structure of the contiguous file on the hard drive. The server typically splits up the full video into fragments with one fragment per video GoP. All the fragments are stored within a single contiguous MP4 file. The client can request fragments by timecode instead of by index number. This will give the server more work because the server must be able to translate URL requests into exact byte range offsets within the MP4 file [20].

The player operates in one of two states, namely *buffering* or *steady-state*. In buffering state the player requests a new segment as soon as possible, so when the previous segment was downloaded. That is why it is called the buffering state, it tries to fill the buffer as soon as possible to start or to resume streaming. This is because the client-side buffer must store enough segments to avoid buffer underruns and therefore avoid the video player to stall. In steady-state the video player requests one segment per predefined time interval. Typically this interval equals the segment duration, measured in seconds. So in buffering state, the player tries to build up a target playback buffer as soon as possible and in steady-state it tries to maintain a constant fill level for the playback buffer.

The time when the steady-state starts, depends on the available bandwidth. The video player starts with the buffering state and when it has buffered enough data, it switches to the steady-state. So if the amount of available bandwidth is high, the buffering state is completed quickly and the steady-state can begin. When the buffer deflates as a result of decreasing bandwidth, the player switches back to buffering state until the constant fill level is reached again. The fill level of the playback buffer is measured in seconds. For this player, the playback buffer is 30 seconds, which is a long time. First of all the steady-state is reached when 30 seconds of video segments are buffered, this will delay the initial play time of the video. This is because the video player only starts playing when the steady-state is reached. When the buffer underruns, the video player stalls. When the video player stalls during the stream, the player fills the buffer until the playback buffer has reached a total duration of 30 seconds. If the buffer is totally deflated, it takes longer time to refill the buffer and the video player will stall a longer time, this will lower the QoE of the users.

The previous case assumes unlimited bandwidth availability. We now investigate the player for scenarios where the bandwidth is restricted and changes frequently. In Figure 3.15 the playback buffer size in seconds is shown under restricted bandwidth variations.

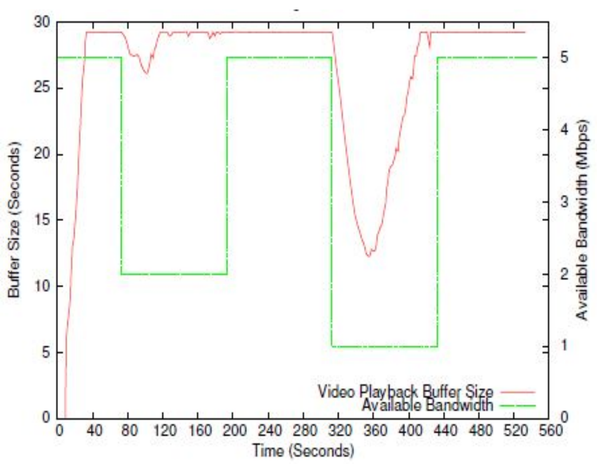


Figure 3.15: Microsoft Smooth Streaming: Playback buffer size in seconds with long bandwidth decreases [54]

To explain this graph, we assume that the video is encoded with eight bitrates between 0.35 Mbps and 2.75 Mbps. Let's assume that t stands for the time. From t = 0 to t = 40, the player is in buffering state. At t = 73 the player is in steady-state. At the same time, the available bandwidth is dropped to 2 Mbps, the player switches from the highest quality (bitrate = 2.75 Mbps) to a quality lower (bitrate = 1.52 Mpbs) after 25 seconds of delay. We see that the playback buffer decreases by 3 seconds, this is because the available bandwidth is not much lower than the bitrate necessary for the highest quality (2 Mpbs < 2.75 Mpbs). After 25 seconds the player downloads a lower quality with bitrate of 1.52 Mpbs. So at approximately t = 100, the player switches from steady-state to buffering state. When the buffer again holds 30 seconds of playback time, the player switches from buffering state to steady-state, this is at approximately t = 120. The large reaction delay of 25 seconds ensures that the player does not react to bandwidth changes based on the latest per-fragment throughput measurements. It averages the per-fragment measurements over a longer time period. By this way, it can act based on a smoother estimate of the available bandwidth variations. The QoE of the users will be high in this case because when the bandwidth increases again within 25 seconds, the quality is not reduced.

Let's assume in the next scenario that the bandwidth increases for a short period. Take the next graph as an example of a bandwidth increase for a short period.

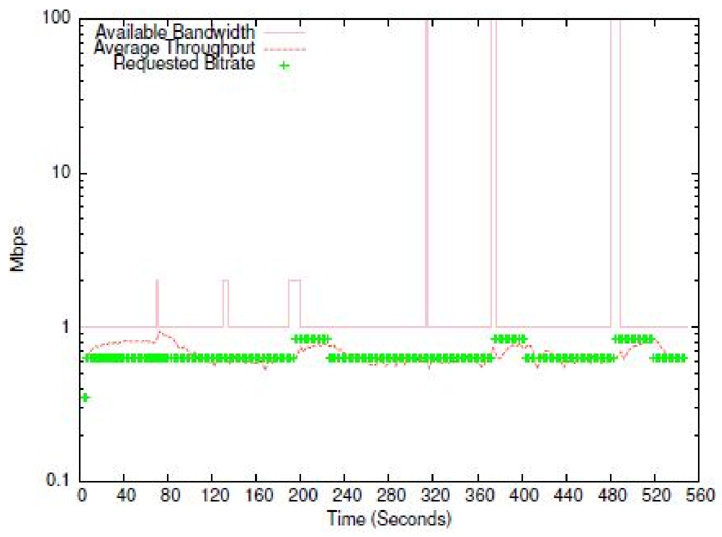


Figure 3.16: Microsoft Smooth streaming: Playback buffer size in seconds with short bandwidth increases [54]

In Figure 3.16, we see that when the bandwidth increases for a short period, the quality is not increased, like at t = 70, t = 140 and t = 320 for example. When the duration of the bandwidth change is longer, but still fairly short, like at t = 180, t = 380 and t = 500, we see that the quality is upgraded. Therefore the QoE of the users will be low because the quality can change quickly.

We can conclude that this player works well in situations where the bandwidth decreases for a long period. When the bandwidth increases for a short period, the video player will react aggressively. This aggressive behavior will cause the quality of the video to change frequently when short bandwidth changes occur. This will cause a lower QoE for the users.

3.4.4.2 Netflix player

The Netflix [35] player uses Microsoft Silverlight [33] for media representation and a different rate-adaptation logic than the Microsoft Smooth Streaming player. The player starts playing after a time period or when the buffer size reaches a certain size. When the buffer is deflated, the Netflix player stops the playback and shows a message that the player is adjusting to a slower connection. Afterwards the player resumes when the certain buffer size is reached again.

Like the Microsoft Smooth Streaming player, the Netflix player will start in a buffering state and then switches to a steady-state. In Figure 3.17, we see that the player changes the quality immediately when the bandwidth reduces. In Figure 3.18, we see the reaction of the Netflix player with short bandwidth increases.

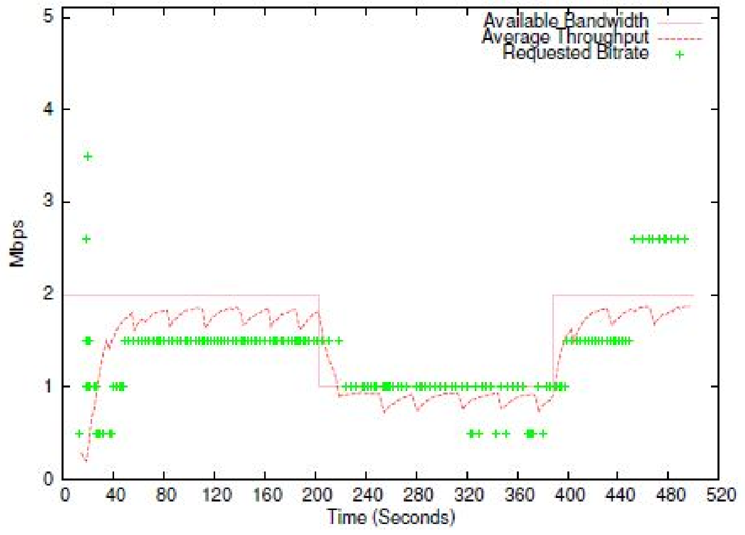


Figure 3.17: Netflix player behavior with long bandwidth decrease [54]

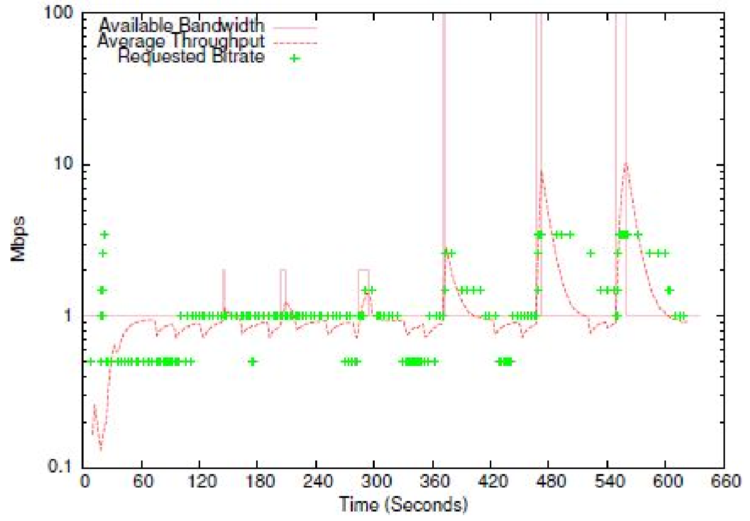


Figure 3.18: Netflix player behavior with short bandwidth increases [54]

We see in both figures that in the begin of the stream the player starts requesting bitrates above the defined bitrates. We see this in Figure 3.17 at t=0 until approximately t=40 and in Figure 3.18 at t=0 until approximately t=30. This is because the player starts measuring the capacity of the underlying path before it starts streaming. We see in both figures that the Nextflix player will react aggressively to bandwidth changes. When the bandwidth changes for a long time period, the video player will display the same quality for the whole period, this is a good thing about this video player. For bandwidth changes with a short period, the player does not perform well. This is because the video player will immediately change the quality. We also see that the larger the peaks are, the more the quality changes. This will introduce many quality changes and lower the QoE of the users.

We can conclude that the Microsoft Smooth streaming player performs well in situations where the bandwidth decreases for a longer period because it changes the quality only after 25 seconds. It reacts more aggressively when the bandwidth increases, what will cause the quality to change. For the Netflix player, we see it reacts aggressively for both negative and positive bandwidth changes. So the Netflix player will perform well in situations where the bandwidth changes for a long period, independent whether the bandwidth increases or decreases. The Netflix player however fails to cope with bandwidth fluctuations of short duration, as in these cases it tends to switch the playback quality to quickly. A better video player would wait a time period for modifying the quality in both cases where the bandwidth decreases or increases. The video player can then act better on a smoother estimate of the available bandwidth when the bandwidth increases or decreases and the quality is not changing rapidly. A disadvantage of this approach is that the buffer can underrun, so the video player must determine a good time period to wait for switching the quality whereby the buffer will not underrun and the time is long enough to avoid rapidly changing the quality.

Chapter 4

ODV Tiled Streaming with MPEG-DASH

4.1 Definition

Before explaining what omni-directional video tiled streaming is, we discuss some necessary components to fully understand the idea. ODV is like normal video but has some differences. ODV is a video where the viewer can see 360 degree content, in contrast to normal video where only a particular angle is displayed. Figure 4.1 shows a 360 degree video frame and Figure 4.2 shows only a part of that frame.

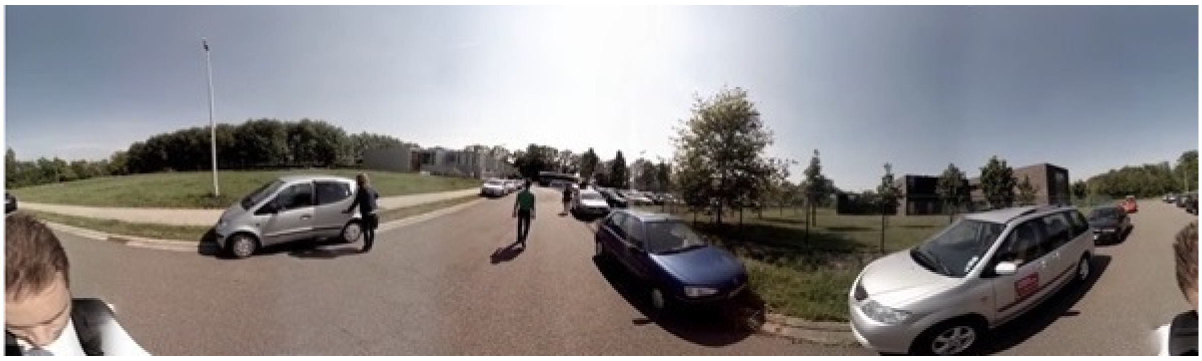


Figure 4.1: ODV full frame

If we want to relate the two video types with each other, we say that Figure 4.1 is referred to as the full video frame and that Figure 4.2 is the viewport. The resolution of the full video frame from Figure 4.1 is 2400x800 pixels. The resolution influences the quality of the image, in this case there are 2400 pixels horizontally and 800 pixels vertically. More pixels yield higher image sharpness. The reason for selecting a part of the full video frame is that we humans can only look at a certain angle what is in front of us. We can not see 360 degree around us. That is why we pick a part out of the full video frame to display, that is called the viewport. It is not the intention to show the full 360 degree video to the users, only the spatially restricted viewport.

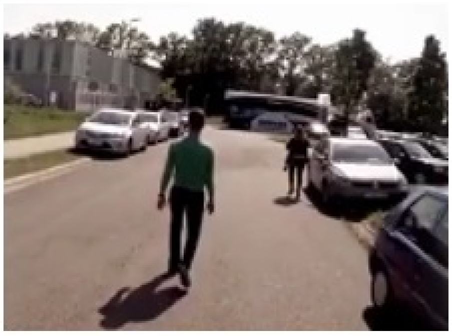


Figure 4.2: ODV viewport

The viewport can be manipulated to make other parts of the full video frame visible. The full video frame is captured with multiple cameras placed in a 360 degree position. When the frames are captured, they overlap because the cameras are standing to closely to each other. We want the full video frame with no overlapping subframes, which is solved in software during the stitching process. Figure 4.3 shows a 360 degree stand with 7 off-the-shelf GoPro [27] cameras as an example of an ODV capturing solution.

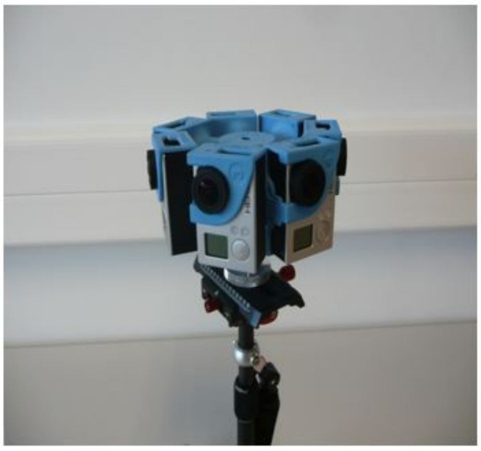


Figure 4.3: 360 degree ODV camera stand involving 7 GoPro cameras

In this masterthesis, 16 video elements are made out of the full 360 degree video, these are called *tiles*. Every part of the grid is a video and is streamed separately to the client. Every tile is a video and has therefore its own parameters, like the necessary bandwidth and quality of streaming. The full video frame is split into multiple video elements. Figure 4.4 shows the full video frame from Figure 4.1 divided into a 4x4 grid. The viewport from Figure 4.2 is hightlighted as a rectangle in the middle of the full video frame in Figure 4.4. The viewport is the only part of the full video frame the viewer can see.

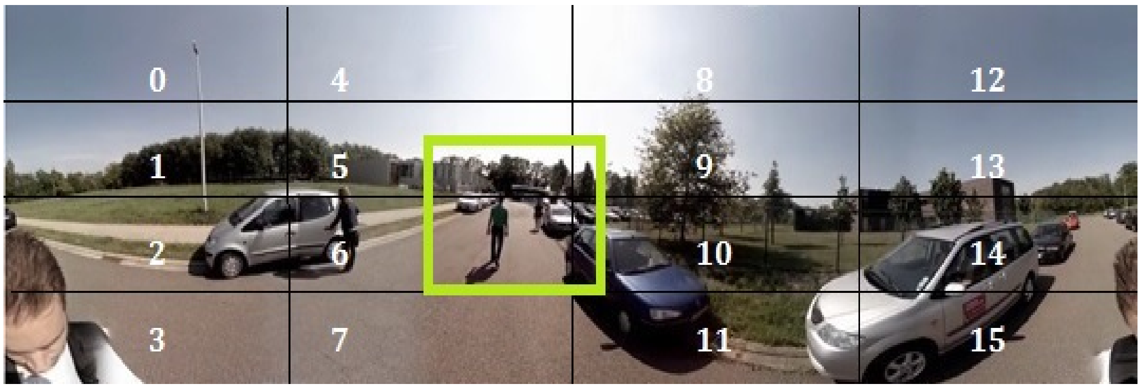


Figure 4.4: Full ODV frame tiled into a 4x4 grid with an inidication of the current viewport

ODV streaming could be used in multiple scenarios, like concerts, festivals, sport events, indoor navigation, Oculus rift [37], 3D omnidirectional games, etcetera. To give the reader an idea, we will show some examples. In Figure 4.5 an omni-directional image is shown of a music concert. In this figure we see that the cameras are placed on the stage. The singer and the band are displayed at the center of the video, they are the main parts of this video. Still, a part of the audience is included in the captured footage as well. In Figure 4.6 a 360 degree image is shown of the interior of a museum. The intention of this scenario is that the user can choose which path is selected through the museum and the user can look around. In Figure 4.5 the cameras do not move, in Figure 4.6, the cameras move.

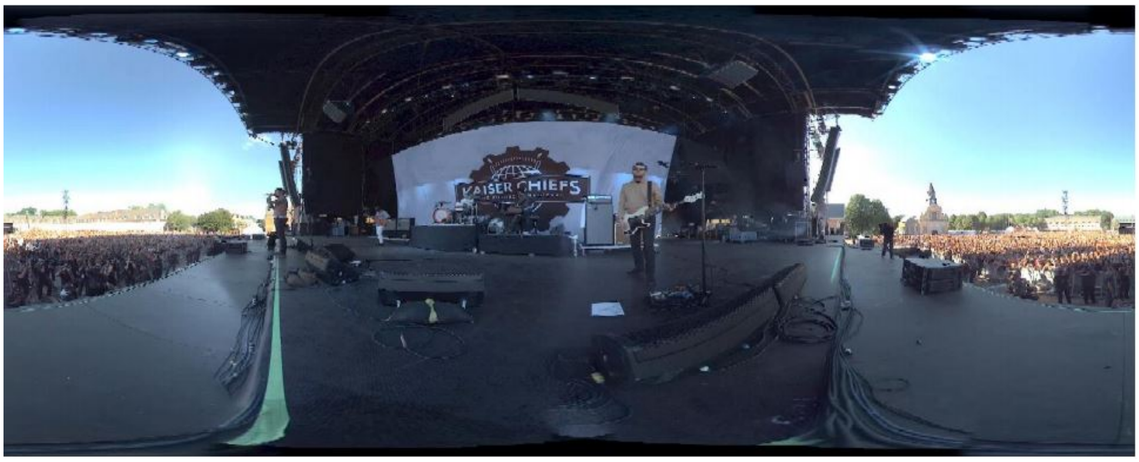


Figure 4.5: Omni-directional view at a concert

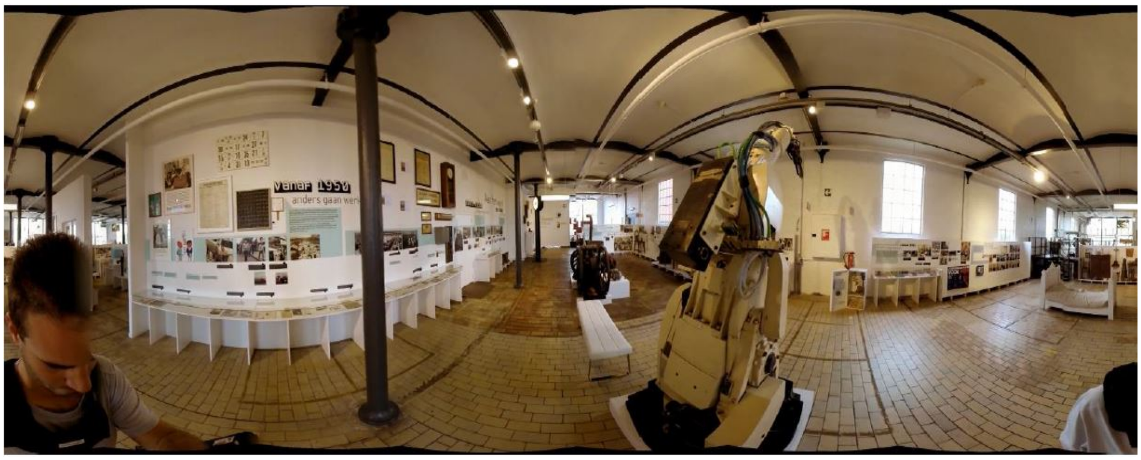


Figure 4.6: 360 degree image of a museum

4.2 ODV tiled streaming using MPEG-DASH

We have chosen to spatially divide the full 360 degree video into tiles, like shown in Figure 4.4. An alternative approach would be to stream the ODV frames integrally, without first subdividing them. This latter approach leaves little room for optimizing the bandwidth consumption behavior of the ODV stream, as it mandates that the integral ODV frame is streamed in a single, uniform quality. With tiled video streaming, we can stream the important tiles at higher quality and the other tiles at lower quality. Tiled streaming hence unlocks the opportunity to allocated bandwidth more efficiently and effectively than it is in the case the 360 degree video is streamed as one video. Every tile is a separate video and has its own parameters, like the spatial location in the full ODV frame, height, width, necessary bandwidth, quality of streaming, etcetera. For every video, an MPD file is generated, like Section 2.2 explained, this means that every tile can be streamed at different qualities.

All the content and MPD files are generated with FFMPEG and MP4Box as explained in Appendix A, after which all the resulting files are stored on an HTTP server. Segments of a particular tile can be requested by sending an HTTP GET request to the HTTP server. Like said in Section 2.4, all the parameters are listed in the MPD file that are necessary to receive the video stream.

The amount of tiles for the full video frame is chosen when the video is encoded. In this master thesis the full video frame is divided into 16 tiles with 4 rows and 4 columns. The amount of tiles determines the performance of ODV streaming. Table 4.1 shows 2 scenarios we distinguish from this idea.

	Low amount of tiles in full video frame (scenario 1)	**High amount of tiles in full video frame (scenario 2)**
Tile area	large	small
HTTP overhead	low	high
Flexibility	low	high
Granularity	low	high
Bandwidth efficiency	low	high
Processing (encoding, decoding)	low	high

Table 4.1: Comparison tiled streaming with low and high amounts of tiles in the full video frame

With a low amount of tiles in the full video frame, referred to as scenario 1 in Table 4.1, the tiles are bigger because they take up more space in the full video frame. With a large amount of tiles, referred to as scenario 2 in Table 4.1, the tiles are smaller because they take up less space in the full video frame.

When we have less tiles in the video frame, like in scenario 1, the HTTP overhead will be lower than in scenario 2. For every tile an HTTP request and response is sent for one segment. Every request and response includes an HTTP header. In scenario 1, we have less tiles, so the overhead will be smaller than in scenario 2.

There is more flexibility in scenario 2 than in scenario 1. Since in scenario 2 the tiles are smaller, we get more flexibility in assigning qualities to tiles. When we have a small amount of tiles, larger areas of the full video frame have the same quality. In scenario 2, we can assign specific qualities to more tiles than in scenario 1.

Granularity in scenario 1 is lower than in scenario 2. Granularity is the extent to which level of detail is present in the video frames [46]. When there are less tiles in the full video frame, there is less detail in the image when the quality of the tiles is low. When we have more tiles in the full video frame, we can better assign qualities to tiles because tiles are smaller. Therewith we can assign intermediate qualities to smaller tiles instead of assigning low quality to larger tiles. So the quality is better with more tiles in the full video frame and therefore the level of detail is higher.

Mapping larger tiles to the viewport can ensure that relatively large parts of the tiles fall out of the viewport. This will waste bandwidth because the part of the tile that falls out of the viewport is streamed in the same quality than the part that falls in the viewport. In Figure 4.7, we see the full video frame divided into a 4x4 grid and in Figure 4.8, we see the same video frame divided into a 2 x 2 grid. The viewport is denoted by the rectangle in the middle of the full video frame and the unvisible parts of the viewport tiles are denoted in green.

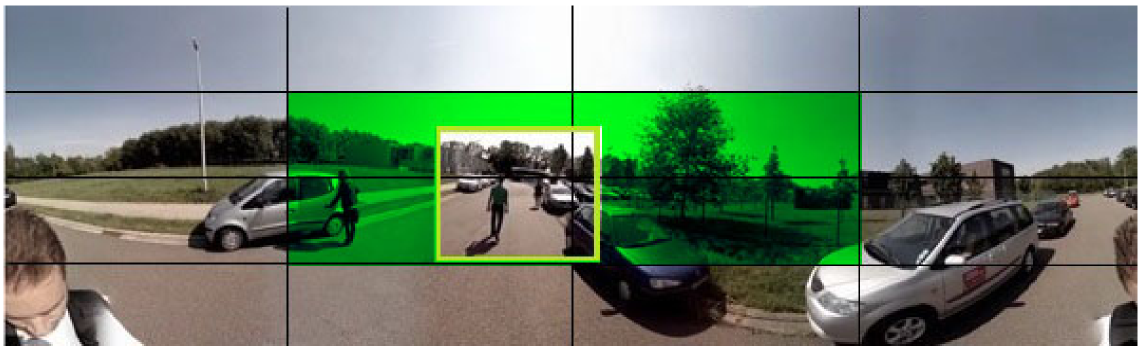


Figure 4.7: ODV full frame divided into a 4x4 grid with indication of the current viewport and unvisible parts

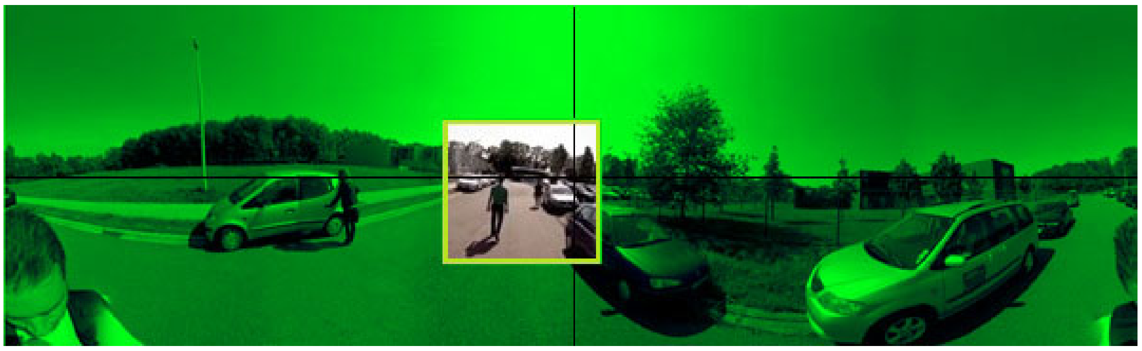


Figure 4.8: ODV full frame divided into a 2x2 grid with indication of the current viewport and unvisible parts

We see in Figure 4.7, with smaller tiles, that the unvisible parts of the viewport tiles are smaller than in Figure 4.8. Because we want the viewport tiles to stream at a higher quality, the unvisible parts will also be streamed at higher quality. If we have larger tiles, more bandwidth is needed for the whole tile to stream at high quality. This will result in wasted bandwidth if the user is not panning to the unvisible parts.

In scenario 2 there is more processing power needed than in scenario 1. Because we have more tiles, more data headers must be prepared and more data has to be sent. The client also must place the tiles on the right position. When there are more tiles, the client must determine more positions for the received video segments.

The standard maximum header size of the last version of Apache Tomcat 8 [21], which is also applicable for Apache web server, is defined by the parameter `maxHTTPHeaderSize`. The default value of this parameter for a request or response is 8 KB. For other webservers this value can vary. So the total size of HTTP header overhead in the worse case is defined with the following formula:

$$(AMOUNTREQUESTS + AMOUNTRESPONSES) * MAXHTTPHEADERSIZE$$

Typically, the HTTP overhead will be small in comparison with the media payload size.

4.3 ODV Viewport Manipulation

In our implementation, the client can change the viewport by using mouse interactions. In our ODV viewer the mouse can be used to pan by dragging it while clicking and holding the left mouse button. The mouse can be used for zooming in and out by using the scroll wheel. Within this implementation, the screen is automatically updated in response to whichever movement is performed. Figure 4.9 [59] shows the different viewport interactions that are considered in this master thesis.

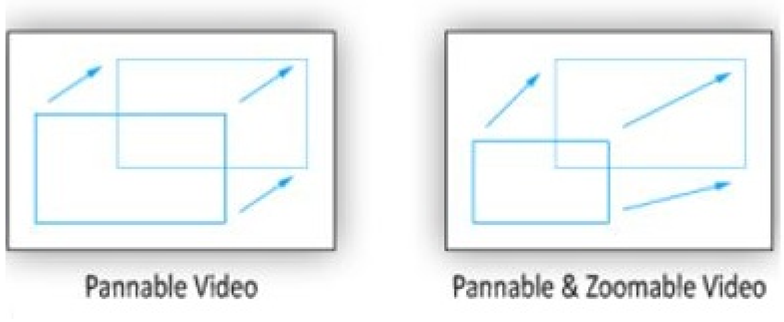


Figure 4.9: ODV viewport manipulation

Scaling down the video occurs when the zoom level is increased, namely when the user scrolls up with the mouse. The video can be scaled up by scrolling down with the mouse. It speaks for itself that zooming in can reduce the amount of tiles in the viewport and that zooming out can increase the amount of tiles in the viewport. When panning, the viewport is moved to another part of the full video frame. In this process, the visible tiles can also change.

The intent of the viewport is that the video(s) in the viewport are streamed at high quality because the user is directly interacting with these tiles of the video. But the question that arises then is what will happen when the viewport changes? To give an answer to this question, we will explain it for the two scenarios in Figure 4.9.

First we must know the coordinates of all the tiles, where they are placed in the full video frame. This is supported by an extension of MPEG-DASH, called spatial relationship description. Please read Section 4.4 to get more information about that.

With panning and scaling of the video, the visible tiles can change. Suppose in this scenario that the visible tiles will change, so that the viewport includes other tiles when panning or scaling up the viewport than before the interaction occurred. The visible tiles can be a subset of the tiles in the viewport before the interaction occurred but can also be totally new tiles. To provide a good experience to the user, the quality upgrade for the new tiles in the viewport must occur fast when the quality is intended to change. The only part of the 360 degree video that the user sees, is the viewport. So it is necessary to stream the tiles of the viewport at high quality. The other tiles are streamed at a lower quality. This is for immediately showing an image when the user pans and for saving bandwidth for the tiles in the viewport.

Every tile of the video has a buffer which stores all the video segments to play for that video element. Like said in Section 2.3, the MPD syntax includes a segment `duration` attribute. This `duration` attribute implies the duration of every segment expressed in seconds. In our experiments the duration is set to either 1 or 2 seconds, so every video segment has a playback time of 1 or 2 seconds. A video consists of multiple video segments played sequentially.

The viewport can move quickly to other tiles when the user pans or scales the video quickly. The more video segments are buffered, the longer it takes for quality changes to become apparent. If we buffer a large amount of video segments at low quality for all the other tiles around the viewport, the playback of these segments must end before other video segments of a higher quality are played. In Figure 4.10 an example with a buffer playback time of 6 seconds is shown.

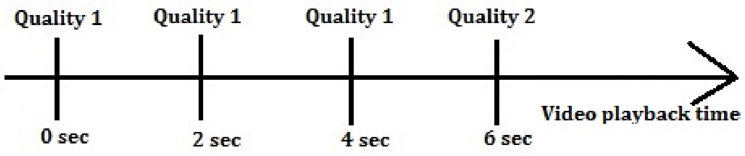


Figure 4.10: Time line which indicates the passed time to change the quality of the video

In this example, the viewport is changed in the begin of the video (playback time = 0 seconds) and the quality of the video is the lowest (quality 1). It will take 6 seconds to change the quality to quality two because 6 seconds of playback is cached in the buffer of quality one when the viewport changes.

If we choose to replace the old video segments with higher quality video segments, we must wait for the new segments to be downloaded and rebuffering can be necessary. The lower quality video segments or a part of the lower quality video segments are downloaded in vain because we replace them with the new segments of higher quality. This approach is hence bandwidth inefficient.

If we buffer high quality video segments for the tiles that are not in the viewport, the bandwidth is wasted if these tiles will not be displayed. A different approach is to predict the movement of the user from previously captured movements; this topic is however not in the scope of this master thesis. So the amount of buffered video segments will determine how fast the quality change occurs for the tiles in the viewport when the tiles in the viewport change.

This reasoning is used for pannable and scaled up video. With both scenarios the tiles in the viewport can change. When the viewport changes, we determine its new location within the ODV frame. With this information, we know which tiles are included in the new viewport, so we know for which tiles we want to download higher quality video segments. Like said before, the quality change only occurs when the playback of the lower quality video segments that are already stored in the client-side buffer for the involved tils ends.

So we have one approach for panning and scaling up the video. This approach is also applicable to a combination of pannable and scaled up video. For scaled down video, the amount of tiles in the viewport can only drop. Therefore the tiles that fall out of the viewport by scaling down the video are streamed at low quality instead of at high quality. No other changes occur when the video is scaled down.

4.4 MPEG-DASH Spatial Relationship Description (SRD)

SRD stands for spatial relationship description. A spatial relation specifies how some object is located in space in relation to some reference object. The reference object in this case is the left upper corner of the full video frame with coordinates (0,0). The first 0 stands for the start position in the horizontal direction and the second 0 stands for the start position in the vertical direction. In Figure 4.11 the used coordinate system is marked on the full video frame.

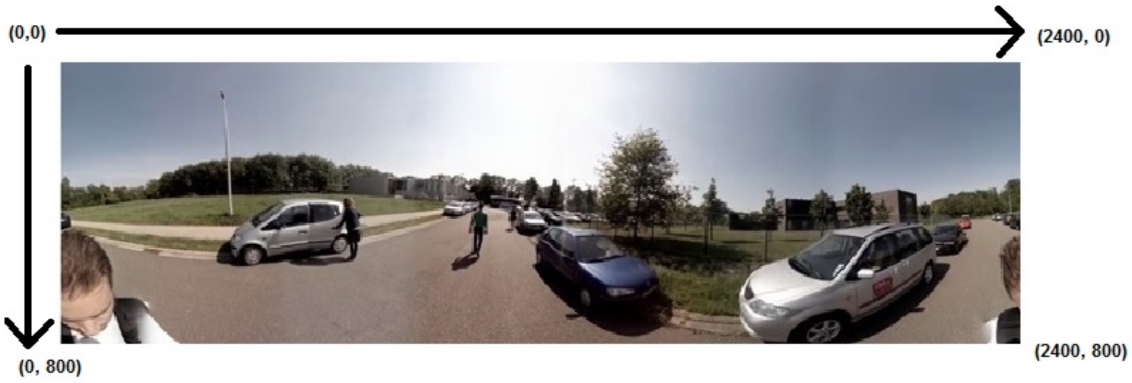


Figure 4.11: ODV coordinate system

MPEG has made an extension on their MPEG-DASH standard, named SRD [18]. Besides the tiled streaming use case, SRD also supports zoomable video. The start x, start y, width and height, measured in pixels, can be defined in the MPD file by some extra parameters. The following is an example of an MPD file with the SRD included.

```
<?xml version="1.0" encoding="UTF-8"?>
<MPD
xmlns="urn:mpeg:dash:schema:mpd:2011"
type="static"
mediaPresentationDuration="PT10S"
minBufferTime="PT1S"
profiles="urn:mpeg:dash:profile:isoff-on-demand:2011">

        <ProgramInformation>
                <Title>Example of a DASH Media Presentation Description
                using Spatial Relationships Description to indicate
                tiles of a video</Title>
        </ProgramInformation>

<Period>
        <!-- Main Video -->
        <AdaptationSet segmentAlignment="true"
        subsegmentAlignment="true" subsegmentStartsWithSAP="1">
                <Role schemeIdUri="urn:mpeg:dash:role:2011"
                value="main"/>
                <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014"
                value="0,0,0,1280,720,1280,720"/>
                <Representation mimeType="video/mp4"
                codecs="avc1.42c01e" width="1280" height="720"
                bandwidth="226597" startWithSAP="1">
                        <SegmentBase>
                                <BaseURL> full_video_small.mp4</BaseURL>
                        </SegmentBase>
                        <SegmentList>
                                ...
                        </SegmentList>
                </Representation>
        </AdaptationSet>

        <!-- Tile 1 -->
        <AdaptationSet segmentAlignment="true"
        subsegmentAlignment="true" subsegmentStartsWithSAP="1">
                <Role schemeIdUri="urn:mpeg:dash:role:2011"
                value="supplementary"/>
                <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014"
                value="0,0,0,640,360,1280,720"/>
                <Representation mimeType="video/mp4"
                codecs="avc1.42c00d" width="640" height="360"
                bandwidth="218284" startWithSAP="1">
                        <SegmentBase>
                                <BaseURL> tile1_video_small.mp4</BaseURL>
                        </SegmentBase>
                        <SegmentList>
                                ...
                        </SegmentList>
                </Representation>
        </AdaptationSet>

        <!-- Tile 2 -->
        <AdaptationSet segmentAlignment="true"
        subsegmentAlignment="true" subsegmentStartsWithSAP="1">
                <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014"
                value="0,640,0,640,360,1280,720/>
                ...
        </AdaptationSet>

        <!-- Tile 3 -->
        <AdaptationSet segmentAlignment="true"
        subsegmentAlignment="true" subsegmentStartsWithSAP="1">
                <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014"
                value="0,0,360,640,360,1280,720"/>
                ...
        </AdaptationSet>
```

```
        <!-- Tile 4 -->
        <AdaptationSet segmentAlignment="true"
        subsegmentAlignment="true" subsegmentStartsWithSAP="1">
                <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014"
                value="0,640,360,640,360,1280,720"/>
                ...
        </AdaptationSet>
</Period>
</MPD>
```

The important element in this MPD file is the `SupplementalProperty` where the `value` attribute defines the SRD. The format of `SupplementalProperty` is:

<SupplementalProperty schemeIdUri="uri" value="source_id, x, y, w, h, W, H/>

The **source_id** is a decimal number providing the identifier for the source of the content. The id is used to know which tiles belong to which reference video. The **x** is the decimal number that expresses the horizontal position of the top-left corner of the spatial object and **y** stands for the decimal number providing the vertical position of the top-left corner of the spatial object. The attributes **w** and **h** stand for the width and height of the spatial object, expressed as decimal numbers. **W** and **H** attributes are optional, they express the width and height of the reference space, in our case the full video frame. The commands in XML are embedded in the `AdaptationSet` scope. For the given MPD, this gives us the following interpretation:

- For the main video, annotated with <!– Main Video –> comment in the MPD, in our case the full video frame, the `value` attribute is **value="0,0,0,1280,720,1280,720**"
  - The first 0 stands for source_id
  - The next 2 values (0,0) stand for the horizontal and vertical positions in the reference space; in this case the main video frame starts at the top-left corner.
  - The following pair (1280,720) expresses the width and height of the video frame. The width of the video frame is 1280 pixels and the height of the video frame is 720 pixels.
  - The last pair (1280,720) stands for the width and height of the reference space. In this example, the reference space equals the full video frame. These cooridnates hence express, like the previous values, the width and height of the full video frame.

We will now explain the values for the first tile; the same reasoning is also applicable to the other tiles defined in the MPD file.

- For the first tile (<!– Tile 1 –>), the value attribute is **value="0,0,0,640,360,1280,720**"
  - The first 0 stands for source_id, this is the same as the main video.
  - The next 2 values 0 and 0 stand for the horizontal and vertical position. In this case 0 in the horizontal direction and 0 in the vertical direction, so this tile starts at the top-left corner of the reference space.

- The following pair (640, 360) expresses the width and height of the tile, so the width is 640 pixels and the height is 360 pixels.
- The last pair (1280, 720) stands for the width and the height of the full video frame, the width is 1280 pixels and the height is 720 pixels.

So for the scenario described in the MPD shown above, Figure 4.12 shows the setup.

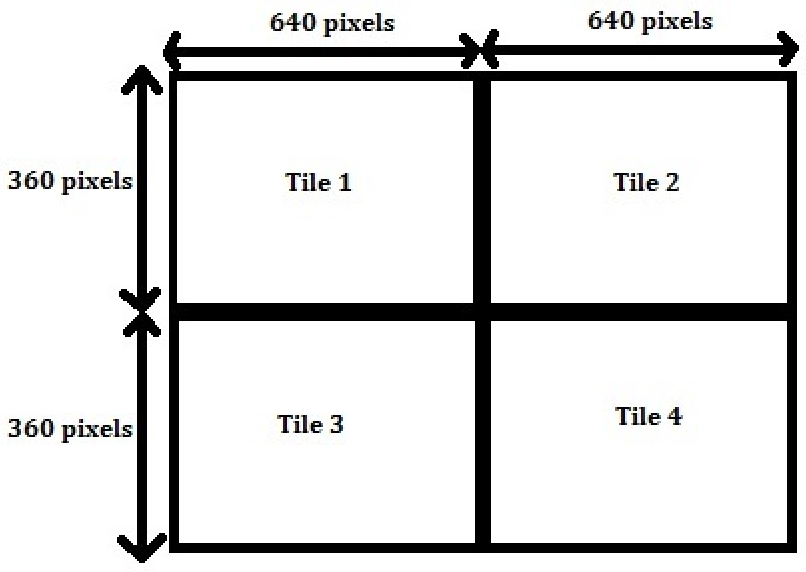


Figure 4.12: Full video frame composed of 4 tiles with a width of 640 pixels and a height of 360 pixels

The zoomable feature of SRD allows the user to see high resolution frames of the video while zooming. An example based on the previous MPD is shown next.

```
<?xml version="1.0" encoding="UTF—8"?>
<MPD
xmlns="urn:mpeg:dash:schema:mpd:2011"
type="static"
mediaPresentationDuration="PT10S"
minBufferTime="PT1S"
profiles="urn:mpeg:dash:profile:isoff—on—demand:2011">

<ProgramInformation>
        <Title>Example of a DASH Media Presentation Description using Spatial
        Relationships Description to indicate that a video is a zoomed part of
        another</Title>
</ProgramInformation>

<Period>
        <!—— Panorama Video ——>
        <AdaptationSet segmentAlignment="true" subsegmentAlignment="true"
        subsegmentStartsWithSAP="1">
                <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014"
                value="0, 0, 0, 1920, 1080, 1920, 1080"/>
                <Representation id="1" bandwidth="5000000" width="1920"
                height="1080" ...>
                        <SegmentBase>
                                <BaseURL> panorama_video.mp4</BaseURL>
                        </SegmentBase>
                        <SegmentList>
                                ...
                        </SegmentList>
                </Representation>
        </AdaptationSet>

        <!-- Zoomed Video -->
        <AdaptationSet segmentAlignment="true" sfubsegmentAlignment="true"
        subsegmentStartsWithSAP="1">
                <SupplementalProperty schemeIdUri="urn:mpeg:dash:srd:2014"
                value="0, 960, 540, 1920, 1080, 3840, 2160"/>
                <Representation id="2" bandwidth="5000000" width="1920"
                height="1080" ...>
                        <SegmentBase>
                                <BaseURL> zoomed_video.mp4</BaseURL>
                        </SegmentBase>
                        <SegmentList>
                                ...
                        </SegmentList>
                </Representation>
        </AdaptationSet>
</Period>
</MPD>
```

In this example, we discuss the first tile which is annotated by the "<!– Zoomed Video –>" XML comment. We see in the example the important element that differs from the previous example is the `value` attribute of the `SupplementalProperty` parameter. The `SupplementalProperty` of the panaorama video, annotated by the "<!— Panorama Video –>" XML comment, shows that the full video's resolution is 1920x1080 pixels.

We see that the resolution of the zoomed video is 1920x1080 pixels by the `width` and `height` attributes of the `Representation` parameter. The zoomed video, annotated by the "<!– Zoomed Video –>" XML comment has a different reference space than the panorama video. We can see this in the last two parameters of the `SupplementalProperty` of the zoomed video. We can see that the zoomed video is positioned at 960 pixels horizontal and 540 pixel vertically in the reference space. The zoomed video has a resolution of 1920x1080 pixels. The setup of this MPD file is shown in Figure 4.13.

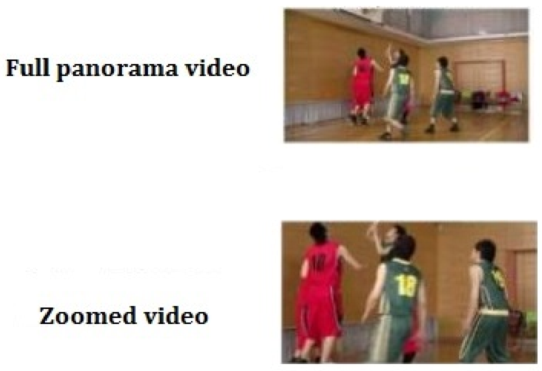


Figure 4.13: Panorama video and zoomed video [58]

4.5 Related work

In this section we will discuss related work to ODV tiled streaming. We will explain every approach and relate it to our approach.

4.5.1 Mixing tile resolutions in tiled video

In networks where multiple clients are watching the same tiled video, reducing duplicative transmissions of content is important. In such scenarios when the clients are watching the tiles of the video at different qualities, the different qualities are streamed separately to the clients. By mixing resolutions of the tiles to one resolution, we can reduce the transmissions of content. A server in the network can cache the video of the tile, which multiple users are watching, before sending it to the clients. The dupiclative transmissions are reduced with this approach because one version of the video is only transferred to the cache server. The cache server, that is placed in the same network as the clients, sends the cached version of the tile video to the clients. Figure 4.14 shows an normal situation where multiple copies of the video are sent to the clients.

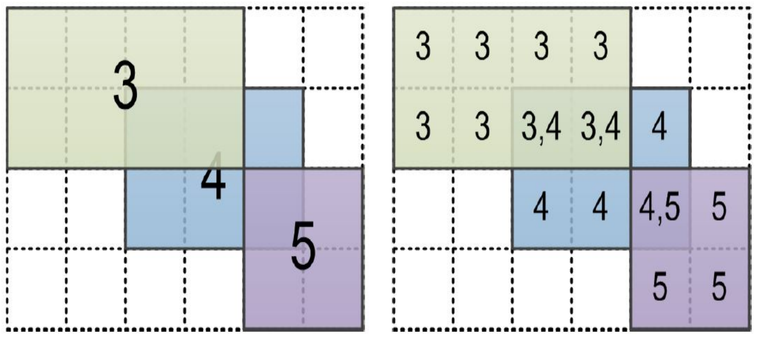


Figure 4.14: Multiple transmissions of the same video with intial qualities[78]

In Figure 4.14 we see that there are five qualities of the video. Quality three, four and five are sent to three different clients. We see that the video streamed at quality three includes 8 tiles, quality four includes 6 tiles and quality five includes 4 tiles. In the figure we see that there are overlapping tiles, referred to with the '3,4' and '4,5' marks. A solution for this is to stream the overlapping tiles at one uniform quality to the clients. In this way a cache server can cache the uniform quality of the video for the overlapping tiles. The uniform quality can be selected using a variety of divergent strategies. The first approach taken for this is to stream the highest quality among all requests. In Figure 4.15 this approach is shown.

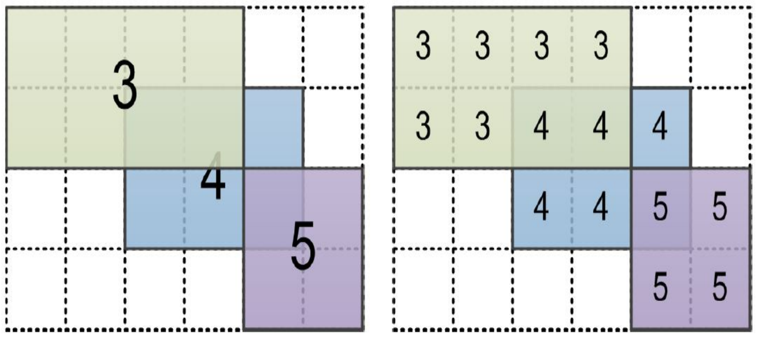


Figure 4.15: Reduced transmissions of the same video with best qualities [78]

The tiles with quality three, that overlap with tiles with quality four, are changed to quality four because this is the best quality. The same is true for the overlapping tiles with quality four and five where quality five is chosen. Another approach that reduces bandwidth, is to stream the unpopular tiles at lower quality. This approach is shown in Figure 4.16.

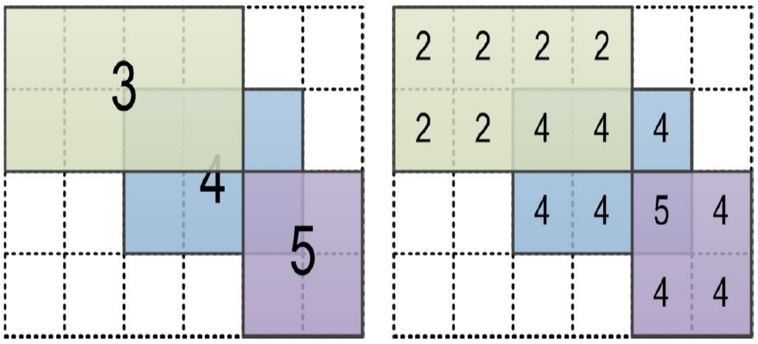


Figure 4.16: Multiple transmissions of the same video with lower quality for unpopular tiles[78]

This approach could save bandwidth because higher quality video is bigger in size than lower quality video. A disadvantage of this approach is that the QoE of the client can be lower because the transition between high and low quality tiles can be more noticeable. In our approach, we focus on the QoE of the clients. We developed our distribution algorithms to improve the QoE for the client by upgrading the quality of the viewport. With our idea of thinking, multiple copies of the overlapping tiles are sent to the clients. In the just described approach, the viewport is not taken into account which could reduce the QoE of the clients. A combination is possible of this approach and our approach where the highest possible quality for overlapping tiles in the viewport can be streamed to the clients by using a cache server. This would require that the other tiles are also streamed at a higher quality because otherwise the transition of quality is more noticeable. This approach would only work if multiple clients are watching the video and some or all tiles of the viewport overlap with viewport tiles of other clients.

The athors of [78] did some tests with participants for different videos and different resolutions. They decide to randomly mixing tiles with different resolutions with each other. Figure 4.17 shows the cummulative distribution function (CDF) of the acceptance scores of the particpants. These scores resprent a number from 0 to 1 of how acceptable the video is for the users. A higher score means that the video is more accepted. The pairs of numbers on the x asis are represented in the format (R_H, R_L). The R_H and R_L are the levels or resolutions combined with each other. The levels of resolutions are shown in Figure 4.18.

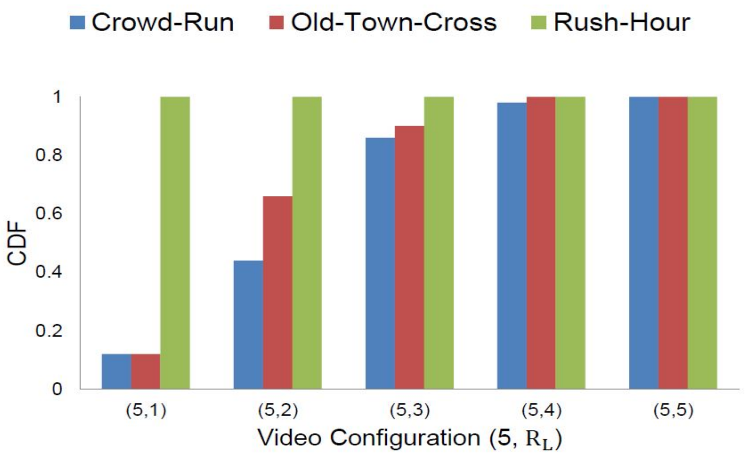
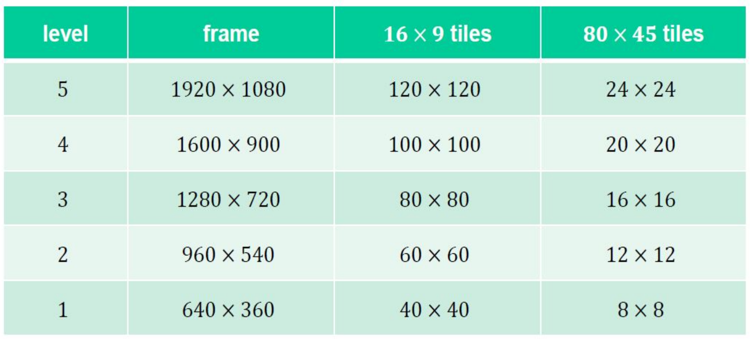


Figure 4.17: CDF distribution function of quality acceptance score when mixing multiple tile resolutions [78]



level	frame	16 × 9 tiles	80 × 45 tiles
5	1920 × 1080	120 × 120	24 × 24
4	1600 × 900	100 × 100	20 × 20
3	1280 × 720	80 × 80	16 × 16
2	960 × 540	60 × 60	12 × 12
1	640 × 360	40 × 40	8 × 8

Figure 4.18: Different levels of resolutions used in measuring the acceptance score by mixing different resolutions of tiled videos [78]

We can see in Figure 4.17 that resolution 5 is mixed by resolution 1, 2, 3, 4 and 5 respectively. In the next figure, we see the videos that are used for the tests. They included videos with low, medium and high amount of movements (motion) to do the test.

In Figure 4.17, we see that with little movement the video is always accepted. So when the resolution is low or high, the video is accepted by the user. We see that when there is more movement in the video, the video acceptance value is lower. The more movement in the video, the lower the acceptance score is when low and high resolution tile videos are mixed. In this case if the resolution is larger, the acceptance score will also be larger. This idea of thinking can be used in our approach. When there is little movement in the video, the tile videos can be streamed at low quality . When there is more movement in the video, the tile videos can be streamed at higher quality. This can be an approach to make a distribution logic for it that checks if there is little or a lot of movement in the video. On the basis of the movement, the distribution logic can assign the available bandwidth to the tiles. With more movement, the distribution logic could allocate more bandwidth to the viewport tiles and with less movement it could allocate less bandwidth to the viewport tiles. This would reduce the bandwidth consumption in the network and assure that the viewers are satisfied.

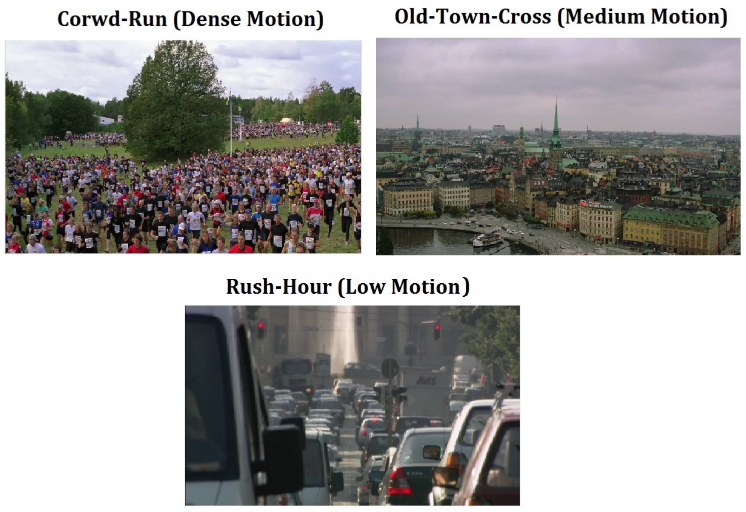


Figure 4.19: Dense motion, medium motion and little movement video used for measuring the acceptance score by mixing different resolutions of tiled videos [78]

4.5.2 Video streams based on user access patern

In scenarios where multiple clients are streaming tiled video in a fixed scene, parts of the tiles can fall out of the viewport. This would waste bandwidth for the unvisible parts of the tiles. In figure 4.20 an example is given. On the right side of the figure, the RoI or in our case the viewport is marked by the rectangle.

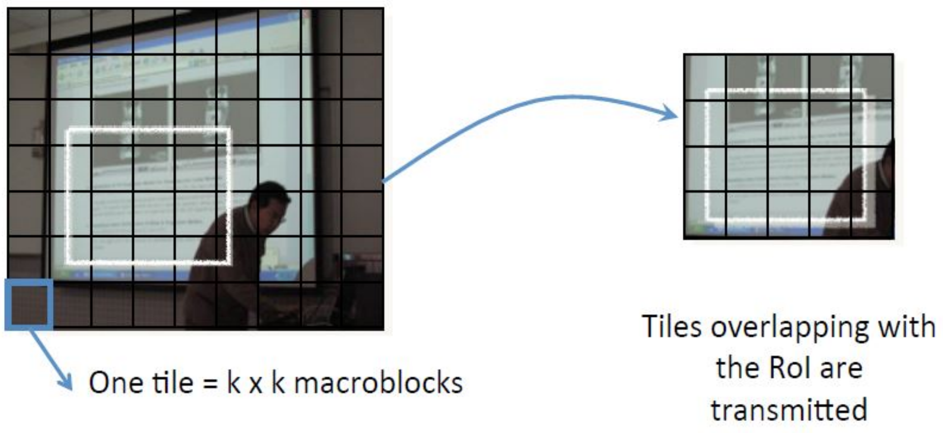

One tile = k x k macroblocks
Tiles overlapping with the RoI are transmitted

Figure 4.20: Multiple transmissions of the same video with best qualities[68]

We see on the right side of the figure that there are parts of the tiles that fall out of the viewport. These parts are streamed to the client because they are part of the viewport. To reduce this overhead, a server could store the specific coordinates of the viewport the user is watching and tile the video dynamically to match the coordinates. This would require a lot of processing power of the server because when multiple clients are requesting data, the server is heavily loaded. Another disadvantage is that the client must first send the coordinates to the server. After receiving the coordinates, the server must process the request. This will delay the delivery of the content and therefore the QoE of the user is lower. An approach to solve this is to log the user selections of the viewport. In Figure 4.21 this approach is shown.

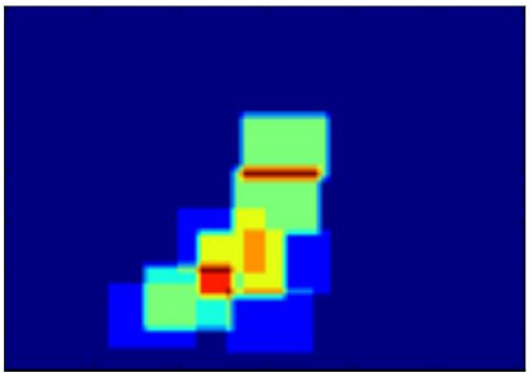


Figure 4.21: User access patterns of viewed parts of the video[68]

This approach would require the client to log the coordinates of the viewport every time the viewport changes. This consumes more network traffic and when the user starts streaming, this approach would not work optimally due to no logs are received by the server. When more logs are received the algorithms can better perform. The video is dynamically tiled according to the user's behavior with regard to viewport positioning. The tiles that are watched the most, referred as the red parts in Figure 4.21, are streamed at higher quality. The other tiles that are watched, referred to as the other colored parts, except the blue parts in Figure 4.21, are streamed at lower quality. Like said before, if multiple clients are watching, the server must process every request and also store the logs of every user. For every user a log file is kept by the server. Another disadvantage of this approach is that it only works when the scene stays the same. When the scene changes to another perspective, the user patterns can change also.

Bandwidth can be saved by not streaming parts that fall out of the viewport but it would require a lot of processing power. In our approach we use small to medium sized tiles to stream tiled video. We have chosen not to tile the video into a lot of tiles because this would also require more processing power to encode and decode the tiles. With small to mediuim sized tiles, the overhead of parts falling out of the viewport is minimal and it would not require a lot of processing power to play the videos. In our approach, we consume slightly more bandwidth than this approach, but that is a trade-off we made. An advantage of our approach is that we don not have a startup delay like this approach has.

4.5.3 Monolithic streaming of video

With monolithic streaming, the server transmits only the bits that are required for decoding of the viewport. Monolithic streaming uses information about the macroblocks, which are used for encoding and deconding of the video, to send the video parts of the viewport to the clients. Figure 4.21 shows the process of monolithic streaming. j
The process exists of two phases, namely the pre-computation and the during run-time phase. The pre-computation phase is done once before the video is streamed to the client. In the pre-computation phase, the server analyzes the dependencies of the macroblocks of the video. Like shown in Figure 4.22 the server stores the dependencies in a tree structure. The macroblocks that are dependent of

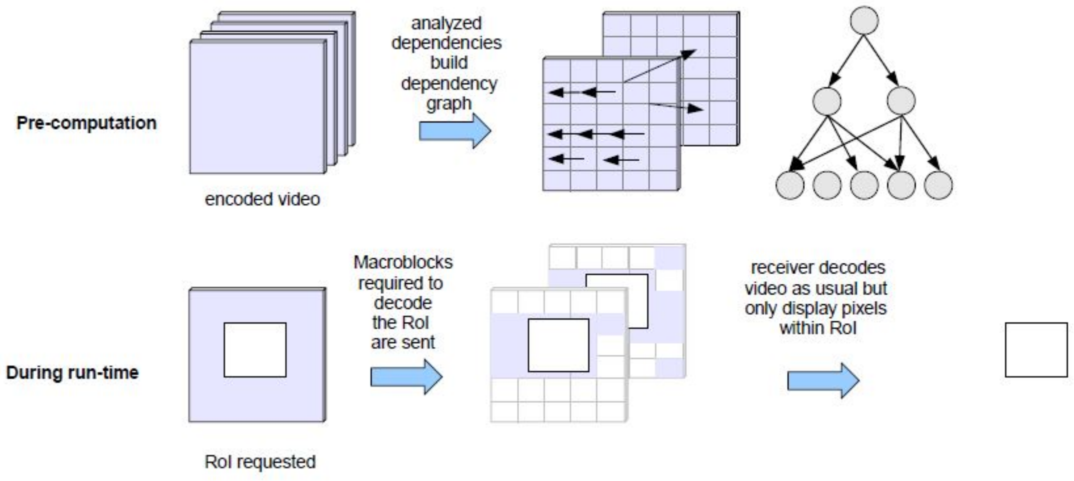

Pre-computation
encoded video
analyzed dependencies build dependency graph
During run-time
RoI requested
Macroblocks required to decode the RoI are sent
receiver decodes video as usual but only display pixels within RoI

Figure 4.22: Monolithic streaming of video process [72]

other macroblocks are the child nodes of that macroblock in the tree structure. A child macroblock is dependent on another parent macroblock if the bits of the parent macroblock are required to decode the child macroblock. I, the second phase, referred to as the during run-time phase, the sever determines which macroblocks are needed to stream the viewport. Only the macroblocks that fall in the viewport and the macroblock on which they depend are sent to the clients. In this way only the necessary bits for decoding the video of the viewport are sent by the server to the clients.

A prerequisite of this approach is that the client needs a robust video decoder because not all bits from the original video are sent or regions outside the viewport are not fully decoded because they are not needed to decode the macroblocks that fall in viewport. Bits that fall outside the viewport are transmitted to the client, but unlike tiled streaming, these contribute to the decoding of the viewport due to dependencies [72]. Nevertheless, the bits outside the viewport provide overhead because they must be transferred to decode the viewport. We can minimize this by reducing the dependencies between macroblocks. The dependencies can be reduced by encoding the video with more I-frames. I-frames are typically larger than P or B-frames, so the challenge in this is to detect a good balance between the amount of I-frames and the dependencies of the macroblocks.

We used tiled video streaming for our implementation. In our case, there are parts that fall out of the viewport. These parts are invisible and bandwidth is wasted for them. We solved this by tiling the video into an acceptable amount of tiles. The size of the tiles determines how many tiles are streamed to the client because larger tiles take more place within the full video frame. Like Section 4.2 explained, we have chosen to use little to medium sized tiles because more tiles introduce more encoding and decoding and with less tiles bandwidth is wasted because large parts of the video are invisible when the user does not pan to those parts. With monolithic streaming, the server has more processing work because during the stream it must check which macroblocks are needed to decode the viewport. This approach will also introduce some overhead because the client must send the coordinates of its viewport. In our approach the client decides which tiles are streamed and no coordinate information is transferred to the server. A counterbalance of this is that only the needed bits to decode the viewport are sent to the client with monolithic streaming.

4.5.4 Tiled video for video conferencing

Videoconferencing applications, like Skype [40], allow multiple people to particpate in a video conference. The people particpating in the video conference are not always sitting nearby the camera that captures the images. This will result in different sizes of people in the video and will not create an immerse video conferencing impression. In this section, we describe a solution that is used for video conferencing within a fixed environment. With a fixed environment we mean an environment that does not change after some time, the people stay in the same room, building, etcetera.

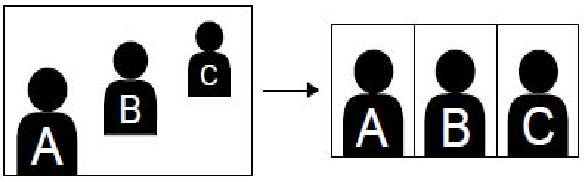


Figure 4.23: Scaled people participating in a video conference [61]

In Figure 4.23, the three people participating in the video conference are scalled to a uniform size. By this way, there is an immerse video conferencing impression created. A combination of face detection, tracking and audio analysis is used to process the input video of the camera. Each person is extracted out of the video and scaled. Afterwards the people are placed side by side, like in Figure 4.23 on the right hand side. This video is encoded in high resolution and sent to the MCU. The MCU is a device that ensures that multiple terminals and gateways can particpate in a multipoint conference [48]. An example where three streams of each three people participating in the video conference is shown in Figure 4.24.

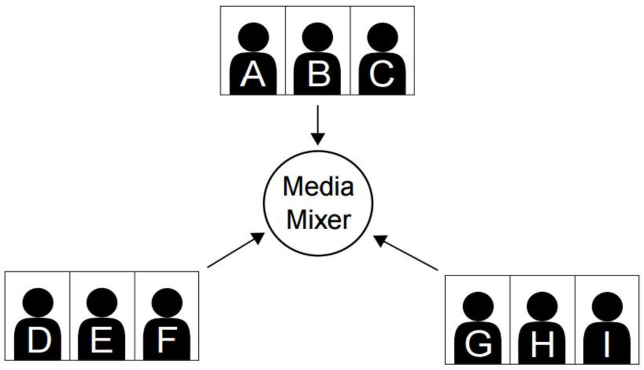


Figure 4.24: Multiple streams of scaled people participating in a video conference [61]

The three videos are encoded and sent to the MCU, referred to as the Media Mixer in Figure 4.24. The MCU decodes the received videos and decodes the videos of the last most active and relevant people. Afterwards these videos are transferred to the cliens. In Figure 4.25 this is shown for the example of Figure 4.24. The last most active people in the conference are E, B, C and I, in descending order.

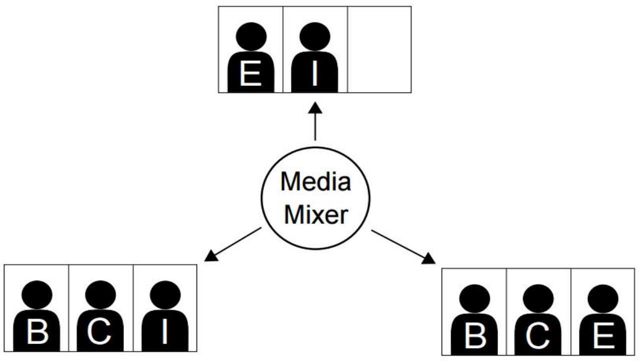


Figure 4.25: Example where most active people are streamed to clients [61]

The clients in the video conference are not receiving the videos of the people they are streaming to the MCU. For the upper client, the client is receiving video of participants E and I. Because B and C are streamed from this client, they are not received from the MCU, this is also true for the other clients. For this approach, the MCU would have a lot of processing work to do because it must decode all the videos received by the clients and then select and encode the specific videos per client. A solution exists by using HEVC [28]. HEVC is a new video compression standard that is developed by the Joint Collaborative Team on Video Coding [30]. The standard allows the video to be encoded into smaller bitrates at the same quality.

With HEVC, multiple encoded tile videos are sent in one playload to the MCU. HEVC adds headers to the payload that contain information about the tile videos, like the start byte position in the payload and the length of the tile's content expressed in bytes. By this way, HEVC allows the MCU to take parts out of the encoded video without decoding it. The MCU just needs to change the headers of the encoded parts the client is interested in. The MCU does not need to decode and encode the tile video, therefore the overhead of decoding and encoding video by the MCU is reduced. Please read [61] for the full explanation.

In our approach we tile the video into tiles with a fixed size. By using HEVC codec in place of the x264 codec, we could extend our implementation to this idea. This approach would be applicable for video conferencing but also for other video streaming applications. This would also save bandwidth because HEVC can encode the video at the same quality but at lower bitrates than the x264 [7]. If the bandwidth consumption is reduced, the quality of the video can be streamed at a higher quality and therefore the QoE of the users is also higher. HEVC is hence compatible by MPEG-DASH [16], this makes a good combination of HAS and reduced bitrates of the video. In our case, HEVC can introduce more processing from the client when multiple tiles are combined in one payload. The client needs to known where every tile video in payload is located. Therefore the payload must be iterated at least once. With our approach, this is not needed because the tile information is once transferred to the client and the client request the content for one or more specific tiles. But the data is received in separate responses and this would reduce the processing of the client. But If we want to save bandwidth, HEVC is better to use because multiple tiles can be combined in one payload. So for one request, multiple tiles can be transfered. With our approach the client needs to send for every tile video segment a separately request. This would introduce extra bandwidth consumption due to the request and response overhead.

4.6 ODV tiled streaming approach

In this section we will discuss our approach for ODV tiled streaming. Because the full video frame is divided into tiles, there are multiple video players, one per tile. We limit the number of buffered video segments to one because we want the quality to change quickly when the viewport changes. When all the video players have one video segment buffered, the current time of the video is set to 0. Afterwards the video players will be started to play at the same moment, this ensures that the video players are synchronized with each other. We have chosen four video qualities for our ODV tiled streaming experiments; the bitrates of these qualities are as follows:

- Quality 1: 50 000 bps (lowest quality)
- Quality 2: 130 000 bps
- Quality 3: 270 000 bps
- Quality 4: 400 000 bps (highest quality)

Because the user only sees the viewport, the bandwidth distribution logic must know which tiles are in the viewport and which are not. On the basis of the start position, the height and width of the viewport, the tiles that fall in the viewport can be known. We gave the tiles in the full video frame an index number to refer to them. Figure 4.26 will show the index numbers for our example of the full video frame. The pseudocode of the algorithm to detect which tiles are in the viewport is given in Appendix F. The tiles are represented in layers, according to their spatial distance to the viewport. The tiles in the viewport are referred to as layer 0, the tiles around the viewport to as layer 1, etcetera. We have opted for a column based numbering approach, but a row based approach would also be possible. To avoid confusion, we say that tiles with numbers 5, 6, 9 and 10 lie in the viewport for our example.

Every tile in the full video frame is assigned a layer number. This number represents the distance of the tile to the viewport. The tiles in the viewport are defined as layer 0, the tiles that border the viewport are defined as layer 1. The tiles that border layer 1 are defined as layer 2, etcetera. In the next sections we will discuss our bandwidth distribution algorithms for ODV tiled streaming.

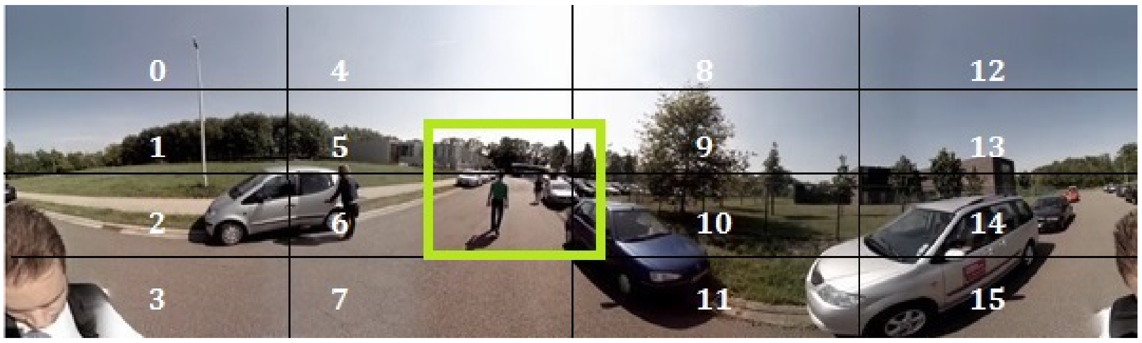


Figure 4.26: Full ODV frame with index numbers for every tile

4.6.1 Simple bandwidth distribution logic

The simple bandwidth distribution logic ensures that the quality is the same for all the tiles in the full video frame. It does not take tiles inclusion in the viewport into account when allocating bandwidth to the tiles. When the user pans in the full video frame, the same quality is shown before and after panning. Figure 4.27 shows this visually, the green tiles are streamed at the same quality. The pseudocode of this algorithm is given in Appendix H.

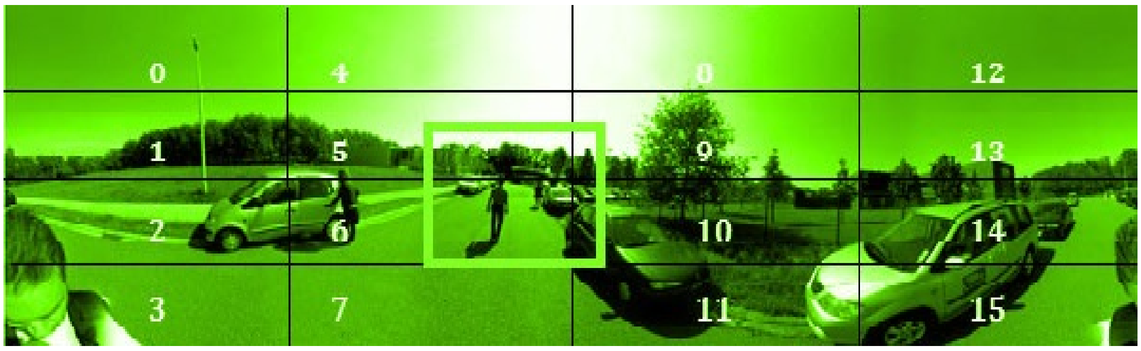


Figure 4.27: Full video frame with all the tiles with the same quality colored in green

4.6.2 Viewport only distribution logic

This distribution logic will only allocate bandwidth for the tiles that are in the viewport. The other tiles that are not in the viewport do not get any bandwidth and therefore the video players will not play. This distrubution logic ensures that the quality of video for the tiles in the viewport is the same and as high as possible. In Figure 4.28 the idea behind this logic is shown.

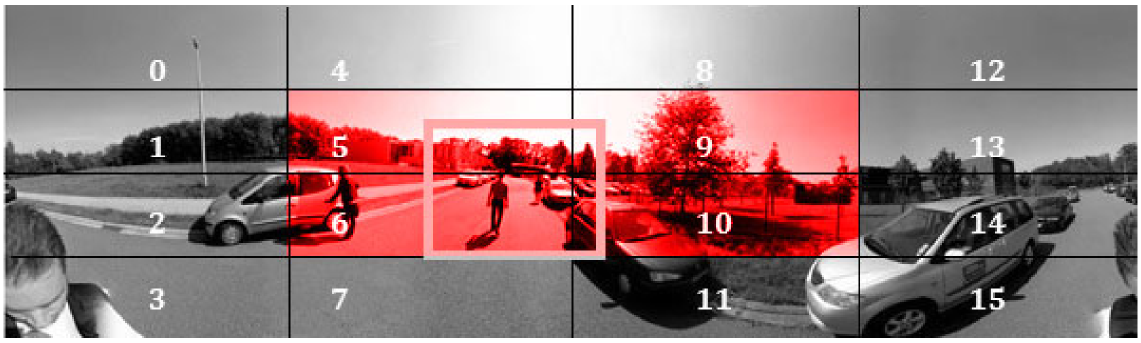


Figure 4.28: Full video frame with viewport colored in red and all other tiles colored in gray

The red tiles with numbers 5, 6, 9 and 10 are streamed at the same quality, the other tiles in gray are not streamed. When the user pans in the full video frame, the tiles that are in the new viewport are played and the other stop playing. In Appendix I, the pseudocode is given for this distribution logic.

4.6.3 Viewport at highest quality distribution logic

With this distribution logic, Bandwidth is first allocated for quality one to all the tiles of the full video frame, afterwards the tiles in the viewport are treated. The algorithm for this distribution logic iterates through all the tiles of the viewport one by one. It starts by allocating bandwidth for the lowest quality for those tiles. There is no guarantee that the tiles in the viewport are streamed at the same quality like in Section 4.6.2. In our case the algorithm starts with the tile with index number 5. The algorithm tries to allocate bandwidth for quality one. If there is remaining bandwidth, it tries to allocate bandwidth for quality one for tile with index number 6, 9 and 10. The tiles from the viewport have a red color in Figure 4.29. After the first iteration, it continues with the same tiles for quality two until the available bandwidth is totally consumed or until the highest quality for these four tiles is selected. The remaining bandwidth is used to select one uniform quality for all the other tiles. So the other tiles are not treated one by one, the algorithm selects one quality for all of them acording to the bandwidth amount that remains available after considering the tiles that are included in the viewport. If there is insufficient remaining bandwidth, these tiles are not streamed to the client. In Figure 4.29 the green tiles are the tiles for which one uniform quality is selected. The pseudocode of this distribution logic is given in Appendix J.

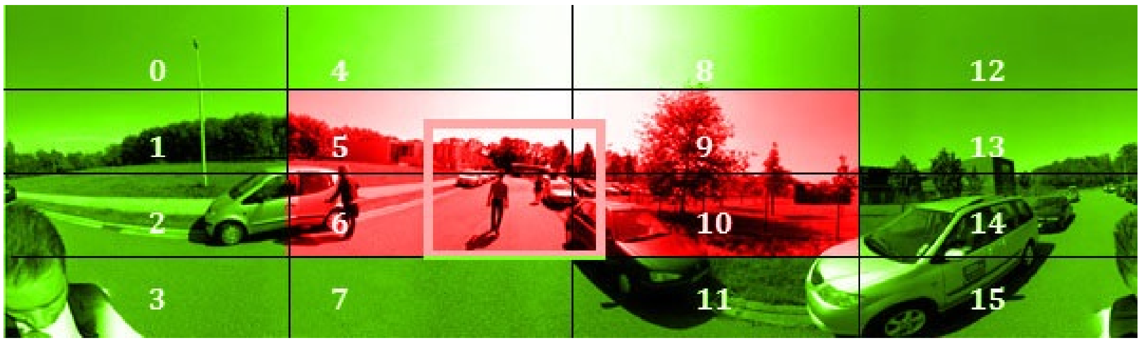


Figure 4.29: Full video frame with viewport colored in red and other tiles colored in green

4.6.4 Viewport at highest quality with lowest quality peripheral tiles

This distribution logic is similar to the previous distribution logic. It also treates the tiles in the viewport one by one and selects one quality for the other tiles. The difference is that the tiles around the viewport are streamed at the lowest quality if there is remaning bandwidth. The quality of those tiles is never upgraded to a better quality. The pseudocode of this algorithm is discussed in Appendix K.

4.6.5 Upgrade layer per layer distribution logic

Every layer of the full video frame is treated one by one with this distribution logic. The algorithm for this distribution logic starts with allocating bandwidth for the lowest quality (quality one) to all the tiles if there is enough available bandwidth. If there is not enough available bandwidth, none of the tiles are streamed. If there is bandwidth remaining after this initial step, the algorithm continues by treating layer per layer. It tries to allocate bandwidth for all the tiles in the first layer. It treats all the tiles one by one like in the viewport with highest quality distribution logic. If there is enough bandwidth after allocating bandwidth for the second lowest quality to all the tiles in the first layer, the quality for tile with index number 5 is upgraded to quality three. When there is remaining bandwidth, the other tiles from the viewport (tiles with index number 6, 9 and 10) are treated in the same way. It continues this way until all the tiles of the first layer are streamed at the highest quality or until there is no more available bandwidth. If there is available bandwidth, tiles of layer two are treated the same way. The algorithm stops when all the layers are treated or when the available bandwidth is totally consumed. In our case we only have two layers. In Figure 4.29, the red tiles are defined as layer zero and the green tiles are defined as layer one, see the last paragraph of Section 4.6.1 for the explanation about the layers. The implementation details are given in Appendix L.

4.6.6 Delta distribution logic

This last bandwidth distribution logic is the most complicated of the six logics. This logic performs well with an high amount of tiles in the full video frame. Because with an high amount of tiles, there are more layers in the full video frame. More layers in the video frame results in less quality transition. The approach of this logic is that the quality difference between layers cannot be larger than a predefined value, this value is denoted as *delta*. The algorithm for this distribution logic starts with the highest possible quality, given the available bandwidth, for the tiles in the viewport and iterates through every layer of tiles around the viewport while taking into account the delta value.

The algorithm first determines the highest possible quality for the tiles in the viewport. It starts immediately with the highest quality, so it tries to allocate bandwidth for quality four for the tiles in the viewport. The algorithm tries to keep the quality in a layer the same. When it determines that the available bandwidth is not enough to stream quality four for the tiles in the viewport, it decreases the quality. It continues with allocating bandwidth for quality three for the tiles in the viewport. If the available bandwidth is not enough to stream quality one for the tiles in the viewport, none of the tiles are streamed. Let's assume that there is enough bandwidth to stream a uniform quality for the tiles in the viewport. The algorithm starts back with allocating bandwidth for the highest quality for layer one. See the last paragraph of Section 4.6.1 for the explanation about the layers. When it has determined a quality for this layer, it checks if the difference in quality between layer zero and layer one is equal to or less than the delta value. If it can not determine a quality for layer one, the quality of layer zero (tiles in the viewport) is decreased to one quality lower than before. It continues back with allocating bandwidth for layer zero and afterwards for layer one. It does this for all the layers and the algorithm keeps checking, after bandwidth is allocated for a layer, if the quality difference with the previous layer is at most the delta value. If it is not, it decreases the quality of the viewport and continues. The algorithm stops when all the layers are treated or quality one can not be streamed for the tiles in the viewport. Like said before, the red tiles in Figure 4.29 are defined as layer zero and the green tiles are defined as layer one. The pseudocode of this algorithm is given in Appendix M.

4.6.7 Duration of quality change

On the basis of Figure 4.30 we will explain how long it takes to change the quality for tiles or how long it takes until the videos start playing. This scenario is not applicable for the simple distribution logic because the quality will not change in that case. This approach is only applicable in scenarios where the quality changes or new videos start playing when the viewport changes. Let's assume that:

S_i: The moment that video segment i is requested.
S_{i+1}: The moment that video segment i+1 is requested.
SD_i: Segment duration $= S_{i+1} - S_i$: after segment duration, measured in seconds, a new video segment is requested.
C_{vw}: The moment that the viewport is changed.
At S_{i+1} the change of the viewport is noticed.

In Figure 4.30 we can see that after $i-1$'th video segment's play time ends, the $i+1$'th segment is requested. We want one segment in the buffer to immediately show to the user when the previous segment is finished playing. The reason behind this is that we want to change the quality of the video quickly when the user pans. When we have one segment buffered, this segment must end its playback before the higher or lower quality segment starts playing. By this we can minimize the time that is passed until the quality changes.

When buffering only one segment in the buffer, the video player is more vulnerable to playback stalls. In scenarios where the bandwidth dramatically drops while downloading segments, the player can freeze due to no buffered segments. This is because the available bandwidth is not sufficient for downloading higher quality video and this delays the delivery of the segments. This is a trade-off that we made because we want to change the quality of the videos as soon as possible. When the user pans a lot, we can not wait too long to change the quality. If we wait longer and the user changes the viewport frequently, the time to change the quality will not be passed and the user will always see low quality video.

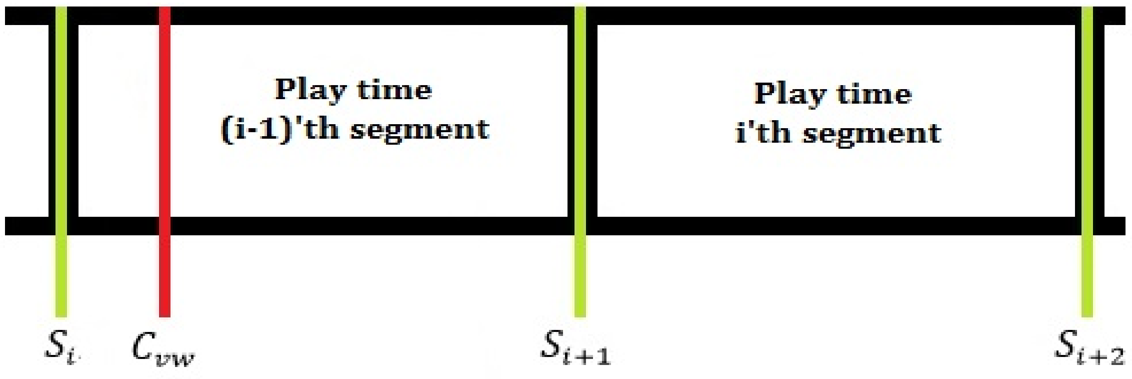


Figure 4.30: Two video segments played after each other when viewport changes

In the depicted example, the quality will change or the video player will start playing within:

$$S_{i+1} - C_{vw} + S_{i+2} - S_{i+1} = S_{i+1} - C_{vw} + SD_{i+1} seconds$$

First we must wait until the new segment is requested, this time equals $S_{i+1} - C_{vw}$ because the new video segment is requested on S_{i+1}. We must wait $S_{i+2} - S_{i+1}$ seconds because at S_i the i'th segment is requested. If we do not want to waste any bandwidth, we play the i'th segment. So the total time of waiting is $S_{i+2} - C_{vw}$ seconds.

4.7 Operations of the distribution logics

In this section, we will show how the tiled streaming distribution logics react in one specific situation. For this situation we have chosen for 1.61 Mbps as available bandwidth. For every distribution logic we will show how that logic distributes the 1.61 Mbps of available bandwidth over the different tiles of a specific ODV sequence. The qualities of video, which can be selected, are the ones discussed in Section 4.6. The viewport is indicated in each case with the rectangle in the full video frame. Figure 4.31 shows the full video frame with the qualities, expressed in numbers, of every tile with the *simple* distribution logic.

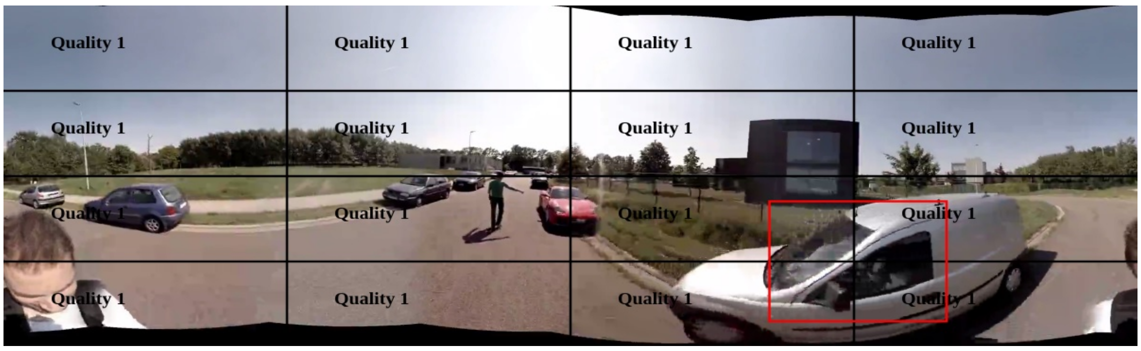


Figure 4.31: Quality per tile for the simple distribution logic

We see that quality one is selected for all the tiles. With 1.61 Mbps of available bandwidth we stream at most quality one for all the tiles. The consumed bandwidth is (16 * 50 000 bps) = 800 000 bps. We can not stream quality two because the consumed bandwidth would be more than the available bandwidth (16 * 130 000 bps = 2.08 Mbps). In the next figure the quality per tile is shown for the *viewport only* distribution logic.

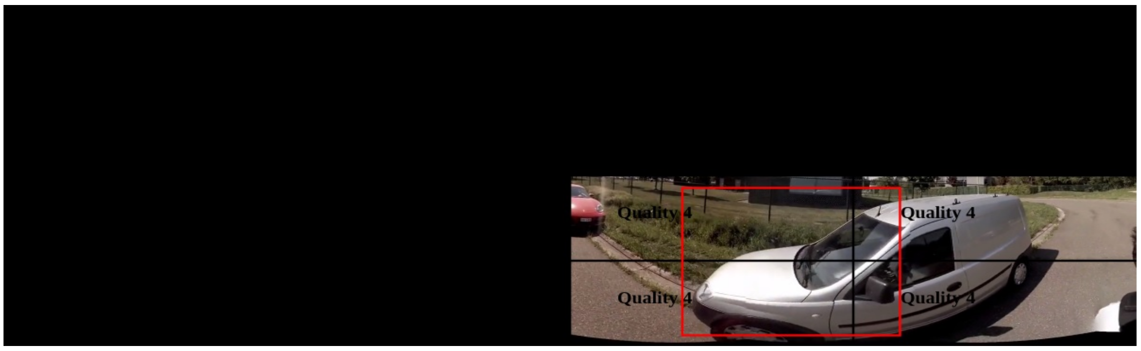


Figure 4.32: Quality per tile for the viewport only distribution logic

In this figure we see that quality four is chosen for the four tiles in the viewport. The other tiles have not bandwidth allocated and therefore they do not show any frames. The quality for the tiles in the viewport must be the same. We can allocate bandwidth for quality four, this will consume (4 * 400 000 bps) = 1.6 Mbps. Figure 4.33 shows the qualities for the *viewport at highest quality* distribution logic.

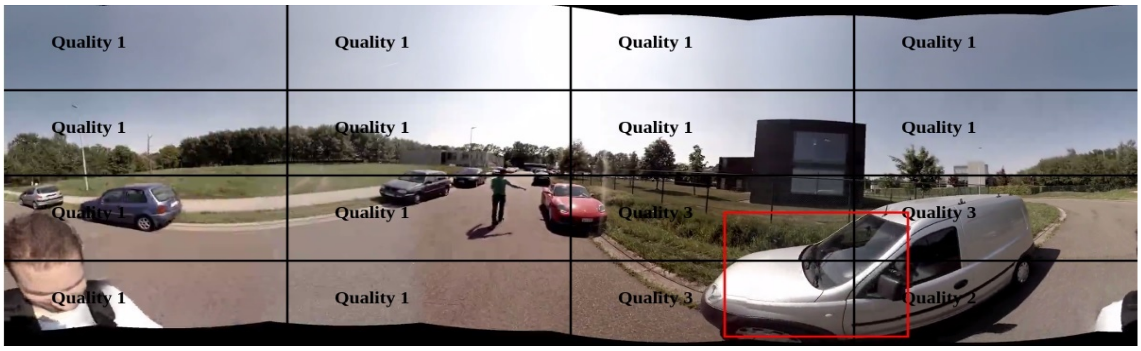


Figure 4.33: Quality per tile for the viewport at highest quality distribution logic

This distribution logic starts with allocating bandwidth for quality one to all tiles, this will consume (16 * 50 000 bps) = 800 000 bps. With the remaining bandwidth we can upgrade the quality for tiles in the viewport. First quality two is selected, this will consume (4 * 130 000 bps) = 520 000 bps for quality two. But we must reduce this value with the previously allocated bandwidth (4 * 50 000 bps) = 200 000 bps. So the total consumed bandwidth is (800 000 bps - 200 000 bps + 520 000 bps) = 1.12 Mpbs. With the remaining bandwidth (1.61 Mbps - 1.12 Mpbs) = 0.49 Mbps we can upgrade three tiles in the viewport. The consumed bandwidth is (1.12 Mbps - 3 * 130 000 bps + 3 * 270 000 bps) = 1.54 Mbps. This is the same for the *viewport at highest quality with lowest quality peripheral tiles* and the *upgrade layer per layer* distribution logics. For the *viewport at highest quality with lowest quality peripheral tiles*, the quality of the tiles around the viewport are not upgraded in comparison with this approach. Because the quality of the tiles around the viewport is the lowest one in this approach, this would not change anything for the *viewport at highest quality with lowest quality peripheral tiles* distribution logic.

For the *upgrade layer per layer* distribution logic, all the tiles have bandwidth allocated for quality one. Afterwards the quality of the tiles in the first layer is upgraded. Tile per tile is treated with this distribution logic. This will result in the same allocated bandwidth per tile than with the *viewport at highest quality* distribution logic. After upgrading the quality of the tiles in the viewport, like Figure 4.33 shows, the remaining bandwidth (1.61 Mpbs - 1.54 Mpbs = 0.06 Mpbs) is insufficient to upgrade one extra tile of layer one to quality two. The quality per tile for the *delta* distribution logic is shown in Figure 4.34. Layer zero is marked in red, layer one in green and layer two in blue.

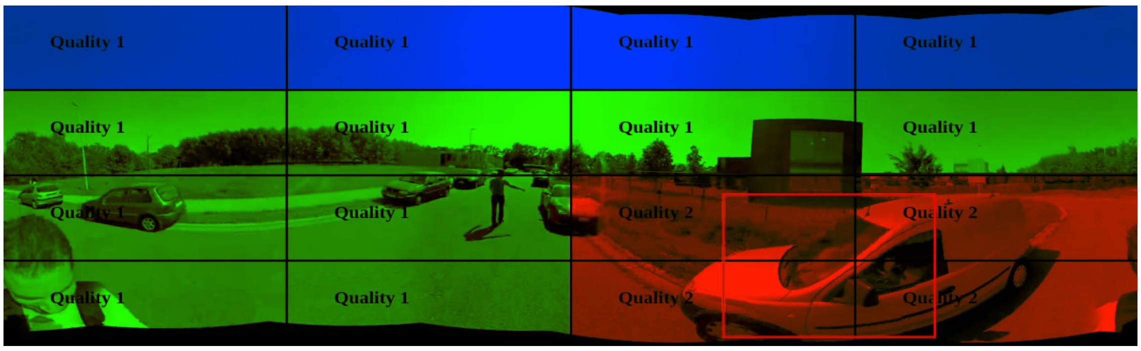


Figure 4.34: Quality per tile for the delta distribution logic

The *delta* value is set to 1. This will result in layers that differ at most one quality of each other. The tiles in the viewport have bandwidth allocated for quality two, this will consume (4 * 130 000 bps) = 520 000 bps and the tiles in the other layers have (12 * 50 000 bps) = 600 000 bps of allocated bandwidth. If we upgrade the tiles of the viewport to quality three, the tiles in layer one must also be upgraded because otherwise the delta value is not respected. This will totally consume (4 * 270 000 bps + 8 * 130 000 bps + 4 * 50 000 bps) = 2.32 Mbps. This is higher than the available bandwidth and therefore we can not upgrade the quality of layer one.

4.8 Bandwidth consumption of distribution logics

In this section we will show the bandwidth consumption for the *viewport only* and *delta* distribution logics. In every case, we will prove that the available bandwidth budget is respected. In Figure 4.35 the bandwidth consumption per tile is shown for the *viewport only* distribution logic. We did not include the other tiles because they are not streamed in this case.

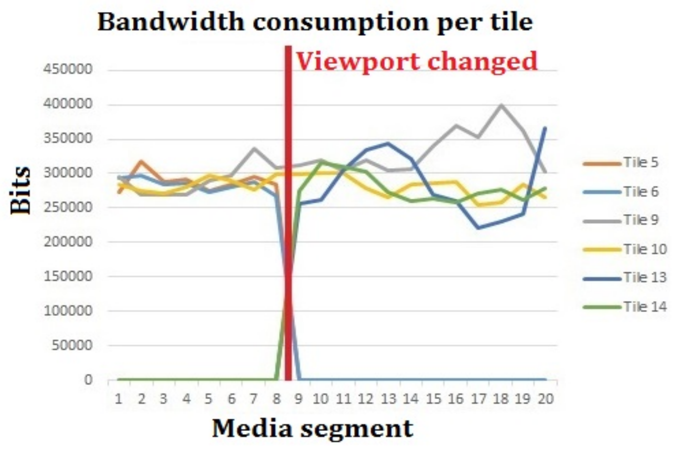


Figure 4.35: Bandwidth consumption per tile for viewport only distribution logic

In the graph, we see that the viewport is changed after video segment 8 is downloaded. Initially, the tiles with index numbers 5, 6, 9 and 10 lie in the viewport. When the viewport changes, the tiles with index numbers 9, 10, 13 and 14 are in the viewport. The available bandwidth is 1.61 Mbps. For the tiles in the viewport, quality four is selected. To stream quality four of the video, every tile needs 400 000 bps allocated of the available bandwidth. Because we have 1.61 Mbps of available bandwidth, every tile will have 402 500 bps allocated. We see, in the graph, after the viewport is changed, that the tiles with index number 5 and 6 stop downloading video segments and tiles with index number 13 and 14 start downloading video segments. We see that the bandwidth consumption per tile does not exceed 400 000 bps, so we can conclude that the available bandwidth is respected. If we do the calculation, the consumed bandwidth is at most (4 * 400 000) = 1.6 Mbps.

In the next graph, we see the downloaded portions for every tile for the *delta* distribution logic. Like in the previous example, the viewport is changed after video segment 8 is downloaded from the tiles with index numbers 5, 6, 9 and 10 to the tiles with index numbers 9, 10, 13 and 14 and the available bandwidth is 1.61 Mpbs.

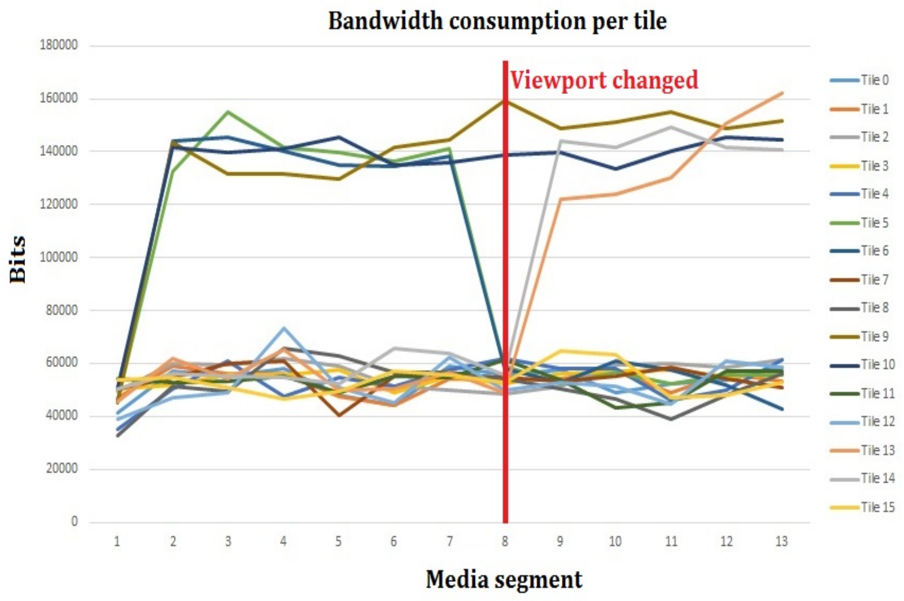


Figure 4.36: Bandwidth consumption per tile for delta distribution logic

As we can see in Figure 4.36, before the viewport is changed, the values of the tiles with index numbers 5, 6, 9 and 10 are not larger than 160 000 bps. The other values lie around 60 000 bps with a maximum of 80 000 bps. If we do the calculation, the consumed bandwidth is (4 * 160 000 bps + 12 * 80 000 bps) = 1.6 Mpbs. After the viewport is changed, we see that the values of tiles with index numbers 9, 10 and 14 are not higher than 160 000 bps and the values of the other tiles, except tile with index number 13, are not higher than 80 000 bps. For tile with index number 13, the value is slightly higher than 160 000 bps. This will not cause the available bandwidth to be exceeded, so we get approximately the same results. We can conclude that the available bandwidth is respected before and after the viewport was changed.

Chapter 5

User testing

5.1 Overview

The goal of the user testing we have done was to qualitatively measure the QoE of the users with ODV tiled video. To do an experiment for this, we have invited participants to evaluate our distribution logics. The meaning of the user testing is to see if the users notice a quality change during the video. We have done a statistical analysis of our results to detect the effects of the independent variables.

This chapter continues with the description of our experiment and the apparatus. Afterwards, the tasks that the users need to perform during the user testing are explained. Our approach is given in Section 5.5 and the methology in Section 5.6. To end this chapter, we will present the results and the three-way ANOVA analysis.

5.2 Description

In this section we will discuss the user testing we have done for this master thesis. We have chosen measure the effect of two durations of video segments, two scenarios of available bandwidth and four bandwidth distribution logics. The used durations of the video segments are:

- 1 second
- 2 seconds

The 2 scenarios of available bandwidth are:

- Low bandwidth: 1.85 Mbps
- High bandwidth: 6.2 Mbps

We have chosen the bandwidth values in such a way that the bandwidth distribution logics will react differently from each other. The four bandwidth distribution logics are:

- Simple distribution logic (Section 4.6.1)
- Viewport only distribution logic (Section 4.6.2)

- Viewport at highest quality distribution logic (Section 4.6.3)
- Delta distribution logic (Section 4.6.6)

We have chosen to include the simple bandwidth distribution logic because we want to detect if the user sees any quality change with this logic. The quality of the video with this distribution logic is not changed when the user pans the viewport, so this is a test to see if the users are seeing a quality change when there is no.

We have included the viewport only distribution logic where only the viewport is played and no other tiles. This distribution logic ensures that all the bandwidth is divided over the tiles in the viewport and thereby the quality of the video viewport is maximized. We want to determine if the user is disturbed by the black tiles around the viewport. After panning the viewport the black tiles will be visible, but they start playing after a short period. Like Section 4.6.7 explained, the duration of the segments determines how fast the quality change happens. For this we want to determine what the viewer thinks about the duration of the quality change.

The viewport with highest possible quality distribution logic ensures that the tiles in the viewport are streamed at the highest possible quality for the available bandwidth. The quality of the tiles around the viewport is determined by the remaining bandwidth. They are streamed at the highest quality that can be accommodated by the remaning bandwidth, but the quality is the same for all those tiles. With this distribution logic we want to detect if the user sees any quality change between tiles in the viewport and other tiles while panning. We also want to detect what they think about the duration of the quality change with respectively one and two seconds video segments.

The last bandwidth distribution logic we have included is the delta distribution logic. This is a more complicated distribution logic, this is also the reason why we included it. With this distribution logic we want to detect if the users see any quality change between the tiles in the viewport and the surrounding tiles.

5.3 Apparatus

To do the user testing for ODV tiled streaming, the first thing we needed was an HTTP server. Like said before, we used a virtual machine whitin Oracle's VirtualBox [43] version 4.3.20 r96996 software. We used the Linux Ubuntu version 14.04 operating system with Apache [22] version 2.4.7 to host all the necessary files. The virtual machine ran on a HP ProBook 6570b [29] with 4 GB RAM memory, 128 GB SSD internal storage and a Intel core i5-3230M 2.60 ghz processor. This laptop was also used for the user tests. The resolution of the screen is 1600x900 pixels. Because our application is developed in Javascript, we used a web browser for the user testing. The web browser we used is Google Chrome version 43.0.2357.130. Before the user testing started and after every test, we gave the user a questionnaire, which was made in Google Forms [26]. A second screen was used to show the questionnaires to the user. The screen is a Dell [24] U2913WM 29 inch LED display. This was done for the easiness, so that the user did not need to switch between tabs or instances of the browser. We have chosen the same video and the same qualities like in Section 4.6 for the user tests.

5.4 Tasks

In this section we will discuss the tasks that every user had to perform when doing the user tests. In the middle of the viewport we have drawn a red X sign for the user testing. The X sign will not move and stays in the middle of the screen. The tasks are related to this X sign; Figure 5.1 shows the X sign on the viewport.

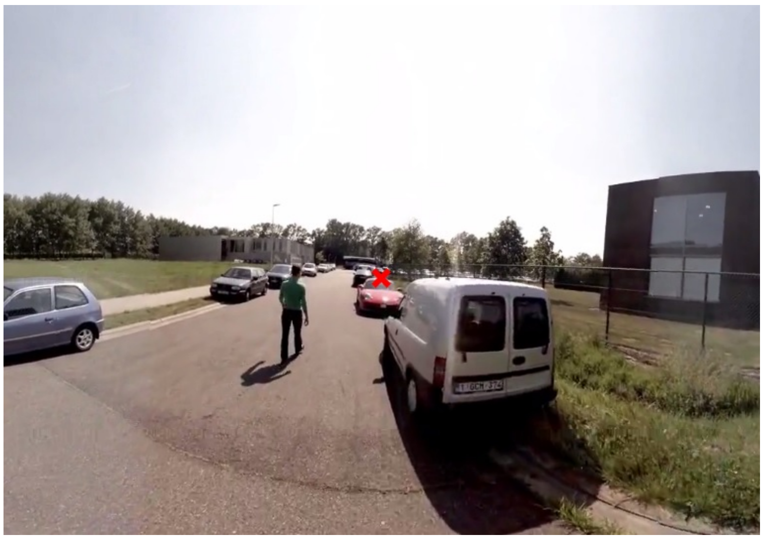


Figure 5.1: ODV viewport with X sign

To draw the X sign we placed a transparent HTML5 canvas on the viewport canvas and drew the X sign on that canvas. Because this canvas lies on top of the canvas of the viewport, the mouse interactions could not be performed on the canvas of the viewport. Therefore we pass on the mouse interactions to the canvas of the viewport to solve that problem. Cathing the mouse events on the top layer canvas and trigger them on the second layer canvas performs this action. The four tasks we have chosen for the users to perform are:

- **Task 1**

  In the video at the right side, you will see a black car. This is the third car at the right side and the brand is BMW. Try to center the X sign on that car until the end of the video. The X sign is always in the center of the screen, so it is not always possible to keep the X sign on to the car. When you cannot hold the X sign on it, just keep it above the car.

- **Task 2**

  Try to keep the viewport straight until you see the first woman in the video. She walks from right to left and she gets in a car. Try to keep the X sign on the woman untill the video ends. When she is in the car, try to keep the X sign on that car.

- **Task 3**

  In the video you will see a red Porsche car, try to keep the X sign on the front window of the Porsche. When the first car on the right side completely falls out of the screen, keep the X sign on the car that is parked opposite to the Porsche. This car is of the brand Volkswagen (see Figure 5.1). Try to keep the X sign on that car until the end of the video.

- **Task 4**

  In the video you will see a red Porsche car, try to keep the X sign on the front window of the Porsche. When the first car, on the right side in the video, completely disappaered from the screen, rotate 360 degrees to the right until the X sign is on the person who is walking in front of the camera. When the black BMW is falling out of the screen, rotate 360 degrees to the left until the X sign is back on the person who is walking in front of the camera.

These tasks were chosen because we wanted to measure the QoE with large and small panning movements, letting us measure the effect of the different qualities of the new tiles.

5.5 Approach

This section will discuss the approach we have taken to do the user tests. We have four bandwidth distribution logics with two durations of the video segments and two available bandwidth scenarios. This gives us 16 test scenarios:

$$4 * 2 * 2 = 16$$

We have excluded two test scenarios from this set, namely the test scenarios for the *viewport only* distribution logic with high available bandwidth for one and two seconds segment duration. We have excluded these tests because, with this distribution logic, it is the intention to measure how disturbed the users are with the tiles that are not playing video. After the user tests were done, we realized that it was better to exclude the tests for the *simple* distribution logic with two seconds segment durations because in those cases the quality does not change and therefore the duration of the segments does not play a role. Then we have only the one second segment duration tests for the *simple* distribution logic with high and low bandwidth. The quality with high and low bandwidth is the same with two seconds segment duration and with one second segment duration. So it was better to exclude the tests for the *simple* distribution logic with one or two seconds segment duration. In total we get 14 test scenarios.

First we have made random assignments of tasks and distribution logics to tests. We have chosen to assign the bandwidth for every test scenario ourselves because the bandwidth value can not appear more than once for the same distribution logic. This is also the same for the duration of the video segments. The letter L stands for low bandwidth with 1.85 Mbps as available bandwidth and H stands for high bandwidth with 6.2 Mbps as available bandwidth. All the random assignments for the user tests are generated by the website of RANDOM.ORG [39]. In the next table the assignements for every test scenario is shown.

Test number	**Task number**	**Bandwidth distribution logic**	**Available bandwidth**	**Duration video segments, measured in seconds**
1	2	B	L	2
2	3	B	L	1
3	3	C	L	2
4	1	D	L	2
5	2	B	H	1
6	2	D	L	1
7	3	C	H	2
8	1	A	H	2
9	2	B	H	2
10	4	C	L	1
11	4	C	H	1
12	4	A	H	1
13	1	A	L	1
14	4	A	L	2

Table 5.1: Assignments per test scenario

The letters for the distribution logics stand for:

Letter	**Bandwidth distribution logic**
A	Simple
B	Delta
C	Viewport highest quality
D	Viewport only

Table 5.2: Bandwidth distribution logics of the user tests

Before doing the user tests we have done pilot tests with two persons. These persons helped with the implementation of the MPEG-DASH framework and the bandwidth distribution logics. They were hence fimiliar with the framework and with the rationale behind the bandwidth distribution logics. We did the pilot tests to see if all the algorithms worked well and to estimate the time it takes to complete all the tests. Minimal problems were discovered such as:

- In the original version the zooming feature of the ODV player was enabled. There was no way to guarantee that different users would use the zooming functionality identically during their tests. Variable zooming behavior per users could give us results we could not compare with each other, therefore the zooming feature was disabled.

- In the original version we did not use an X sign on the canvas but a square with which the user needed to perform the tasks. The square caused confusion about the quality. Tthere was an illusion that the quality of the video for the tiles in the square was better than the surrounding tiles. In reality the quality of video for the tiles in the square was not better than the surrounding tiles. We have changed the square to the X sign in the middle of the viewport.

To reduce the learning effect while the users do the tests, we have used the Latin square [47] method to assign the tests to the users. The result is shown in the following table:

User	**Test**													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	2	3	4	5	6	7	8	9	10	11	12	13	14	1
3	3	4	5	6	7	8	9	10	11	12	13	14	1	2
4	4	5	6	7	8	9	10	11	12	13	14	1	2	3
5	5	6	7	8	9	10	11	12	13	14	1	2	3	4
6	6	7	8	9	10	11	12	13	14	1	2	3	4	5
7	7	8	9	10	11	12	13	14	1	2	3	4	5	6
8	8	9	10	11	12	13	14	1	2	3	4	5	6	7
9	9	10	11	12	13	14	1	2	3	4	5	6	7	8
10	10	11	12	13	14	1	2	3	4	5	6	7	8	9
11	11	12	13	14	1	2	3	4	5	6	7	8	9	10
12	12	13	14	1	2	3	4	5	6	7	8	9	10	11
13	13	14	1	2	3	4	5	6	7	8	9	10	11	12
14	14	1	2	3	4	5	6	7	8	9	10	11	12	13

Table 5.3: Latin square approach of the tests per participant

We see for example that test person 7 started with test 7 as his/her first test and ended with test 6 as his/her last test.

5.6 Methodology

The users tests were performed in the following order:

1 The users were welcomed and an explanation was given why the user tests were performed.

2 The users filled in a starting questionnaire, this questionnaire is shown in Appendix C.

3 The users could practice the mouse interactions with the ODV player in a demo version.

4 The users performed every test scenario and after each scenario they filled in a questionnaire, this questionnaire is shown in Appendix C.

5 The detailed purpose of the user tests were explained to the users.

6 The users were thanked for their participation with a snack and a drink.

5.7 Results

The users who did the user tests were all aged between 22 and 31. Two females and 12 males participated in the user tests. We will first show all the observed results and afterwards analyse them.

For the starting questionnaire we have collected the following values per question:

- **How often do you watch videos on the internet (YouTube, Netflix, Vimeo, ...)?**

  All people daily watch videos on the internet.

- **Do you know what 360 degree video is?**

  Just one person answered no to this question, the others all answered yes.

- **How often do you watch 360 degree video on the internet?**

  5 people answered never and 9 people answered annually.

- **How often do you watch live videos on the internet?**

  5 people answered annually, 4 people answered monthly, 3 people answered weekly and 2 people answered daily.

With these tests, we wanted to see if there is an effect on the results of the people who have more experience with 360 degree video in constrast to people who have less experience. In the following subsections, we will discuss the results for every question of the test-related questionnaire.

To measure if an independent variable has an effect on the measured scores, we did a three-way ANOVA test [17]. For all the three-way ANOVA tests we report in this master thesis we define the significance level 0.05. So when the value is less than 0.05, the result is significant. With significant we mean that the variable or interaction has an effect on the measured scores. In other words, this explains that if a variable or interaction is left out of the test, the measured scores will be changed. Before doing the three-way ANOVA test, we must check the homogeneity of the variances for the measured variable of every question. Homogeneity of variances means that the variances are the same on some level. For the significance level we define 0.05 or 5%. The homogeneity is useful for testing meaningful hypotheses. This is done by the Levene test [31] in R. When the value is larger than 0.05, we can assume the homogeneity of the variances.

5.7.1 Question 1: Rate the quality of the video on a scale from 1 to 5

For this question the aggregated results are shown in Figure 5.2. The quality of the video is measured with a score from 1 to 5, with 1 representing bad and 5 representing very good. The idea behind this is that the participants give a general score for the quality of the video. We expect that the quality scores are higher with high bandwidth than with low bandwidth. For the distribution logic D, we expect that the quality scores are low in both cases.

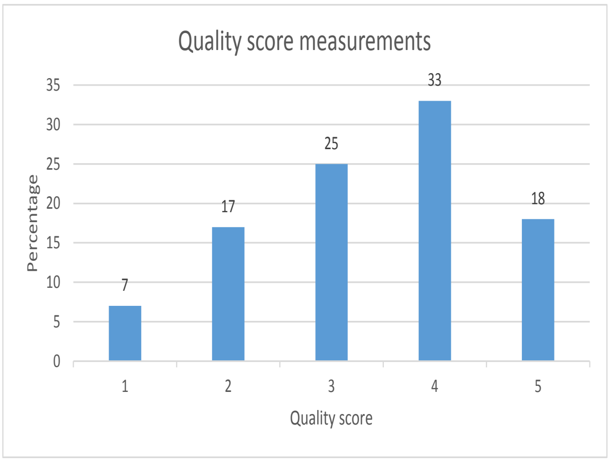


Figure 5.2: Amount of answers per quality score (1 to 5) for ODV tiled video

First we check the homogeneity of the variances. Levene's test did not show a violoation of homogeneity of variances (F(13,182) = 1.13, p = 0.33), we can continue with the three-way ANOVA test. The results of the three-way ANOVA test are:

	DF	Sum Sq	Mean Sq	F Value	Pr(>F)	
Logic	3	31.74	10.58	20.330	2.08e-11	***
Duration	1	6.61	6.61	12.706	4.66e-4	***
Bandwidth	1	110.10	110.10	211.56	< 2e-16	***
Logic:Duration	3	1.99	0.66	1.278	0.28	
Logic:Bandwidth	2	12.94	6.47	12.43	8.69e-06	***
Duration:Bandwidth	1	7.71	7.71	14.82	1.63e-4	***
Logic:Duration:Bandwidth	2	0.25	0.13	0.24	0.79	
Residuals	182	94.71	0.52			

Table 5.4: Three-way ANOVA test for quality score

With three-way ANOVA, we found significant main effects of Logic, Duration and Bandwidth on the quality score. We also found a significant interaction between Logic and Bandwidth and a significant interaction between Duration and Bandwidth. We will investigate these two interactions further. To investigate the interaction between Logic and Bandwidth we will group the logics per bandwidth. We do this by setting the bandwidth to a fixed value and execute the three-way ANOVA test again. We did not find a significant difference between the distribution logics for high bandwidth (F(2,81) = 0.413, p=0.663). We however did find a significant difference between the distribution logics for low bandwidth (F(3,108) = 10.17, p=5.88e-06). We investigate which logics differ from each other with the Tukey's [38] post-hoc test. This test compares the logics with each other on a significant level (0.05 or 5%) and shows a significant difference between the logics if $p < 0.05$. We get the following results:

Logics	diff	lwr	upr	p adj
B-A	0.60	-0.00	1.22	0.05
C-A	1.29	0.67	1.90	1.7e-6
D-A	0.50	-0.11	1.11	0.15
C-B	0.68	0.07	1.29	0.02
D-B	-0.11	-0.72	0.51	0.97
D-C	-0.76	-1.40	-0.17	6.07e-3

Table 5.5: Results of Tukey's post-hoc test for quality scores with low bandwidth for the distribution logics

In this table, in column **p adj**, we read the p value and see that there is a significant difference between logic C and A, C and B, D and C for low bandwidth. The boxlplot for the quality scores for low bandwidth per distribution logic is shown next:

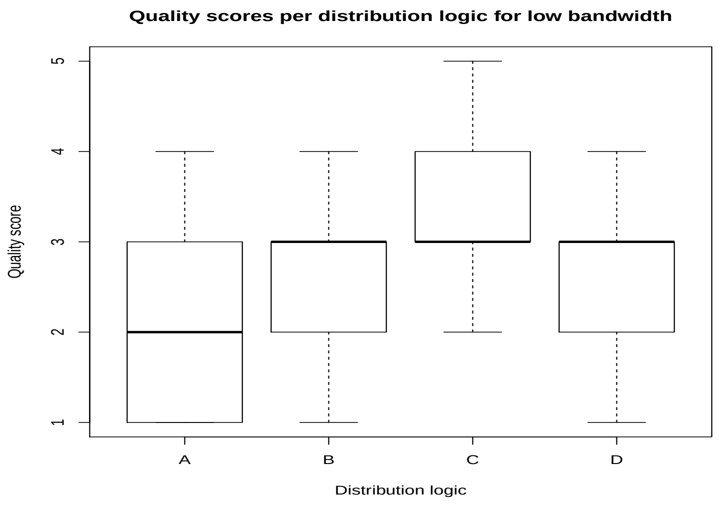


Figure 5.3: Quality scores per distribution logic for low bandwidth

The bold line in the graph represents the median, and the upper line of the box indicates the value of the third quartile and the lower line of the box indicates the value of the first quartile. The upper and lower lines that are connected with a dotted line are the maximum and minimum values. In Figure 5.3 we see that the boxplot for distribution logic C lies higher than the other ones. So we can conclude that distribution logic C has a better quality score than the other logics when the bandwidth is low. With distribution logic C, all the tiles have initially bandwidth for quality one allocated. When there is remaining bandwidth, the quality of the viewport is upgraded. With low bandwidth, the quality of the viewport is upgraded to quality two. The quality is better in the viewport and the transition of quality is hardly noticeable, that is why distribution logic C has a higher quality score with low bandwidth than other distribution logics. The mean and variance values of the quality scores per distribution logic for low bandwidth are shown in the next table.

Distribution logic	Mean	Variance
A	2.07	0.74
B	2.68	0.45
C	3.36	0.90
D	2.57	0.99

Table 5.6: Mean and variance of quality scores per distribution logic for low bandwidth

We can see in Table 5.6 that the variance of distribution logic C and D are larger than the other ones. This means that the values are more widespread with distribution logic C and D and that the values of distribution logics A and B lie closer to each other.

Further we analyzed the effect of the segment duration factor in terms of the bandwidth factor. We fixed the value for segment duration to one second. Our results are: F(1,96) = 153, p < 2e-16. We can say that there is a significant difference between high and low bandwidth for segment duration of one second. The boxplot for high and low bandwidth with one second segment duration is shown in Figure 5.4.

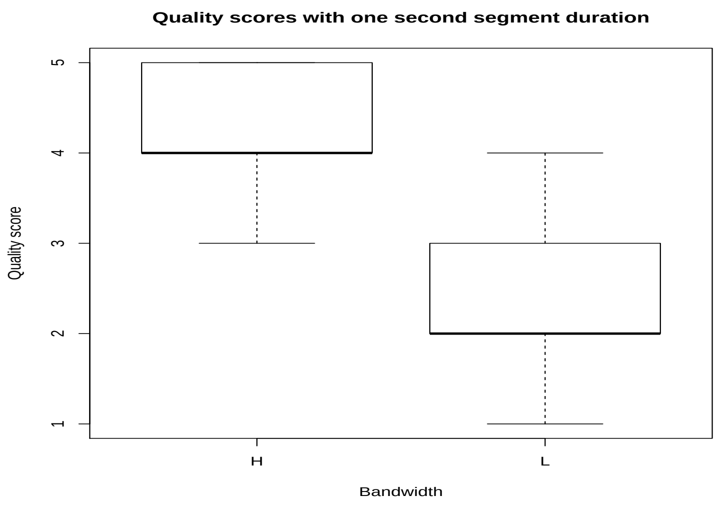


Figure 5.4: Quality scores per bandwidth logic for one second segment duration

We see in Figure 5.4 that the boxplot of high bandwidth lies higher than with low bandwidth. So we can conclude that with one second segment duration, low bandwidth ensures that the quality is lower than with high bandwidth. The mean and variance values are shown in table 5.7. We see that the quality values with low bandwidth are more widespread than the quality values with high bandwidth. Because the quality for high bandwidth is better than the quality for low bandwidth, the quality scores are greater for high bandwidth. We will now investigate the effect of bandwidth with a fixed segment duration of two seconds. We get the results: F(1,96) = 62.74, p=4.18e-12. We can say that there is a significant difference between high and low bandwidth for segment duration of two seconds. Figure 5.5 shows the boxplot for high and low bandwidth with two seconds segments.

Bandwidth	Mean	Variance
L	2.34	0.81
H	4.33	0.37

Table 5.7: Mean and variance of quality scores per bandwidth for one second segment duration

Bandwidth	Mean	Variance
L	3	0.91
H	4.31	0.32

Table 5.8: Mean and variance of quality scores per bandwidth for two seconds segment duration

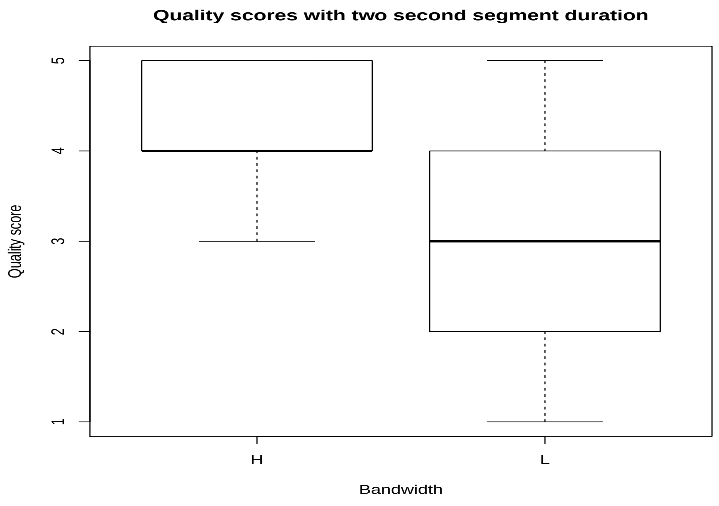


Figure 5.5: Quality scores per bandwidth amount for two seconds segment duration

We see that the boxplot for high bandwidth lies higher than the boxplot for low bandwidth. So with two seconds segments the quality score for high bandwidth is greater than for low bandwidth. The reason why the boxplot for high bandwidth lies higher is, like the previous case, with high bandwidth the quality is better than with low bandwidth. In table 5.8, the mean value and variance are shown per bandwidth for two seconds segment duration. Like in the previous case, we see that the quality values with low bandwidth are more widspread than the quality values with high bandwidth.

The participants mentioned that there were some blurry effects in the videos. They did not knew if the quality was the same for the tiles when the quality was the same and they asked themselves if the quality was the same or not. They rated the quality of the video lower when they saw the blurry effect in the tile videos. When they did not saw the blurry effect and the quality was the same, they rated the quality of the video slightly higher. The blurry effect is caused by encoder because when there is more movement in the video, the encoder tries to encode the video accoring to the target bitrate. So this can be at cost of the quality. When there is little movement in the video, the encoder can encode the video at a higher quality for the same target bitrate.

5.7.2 Question 2: Have you noticed any quality change in the video?

All the answers are shown in Figure 5.6.

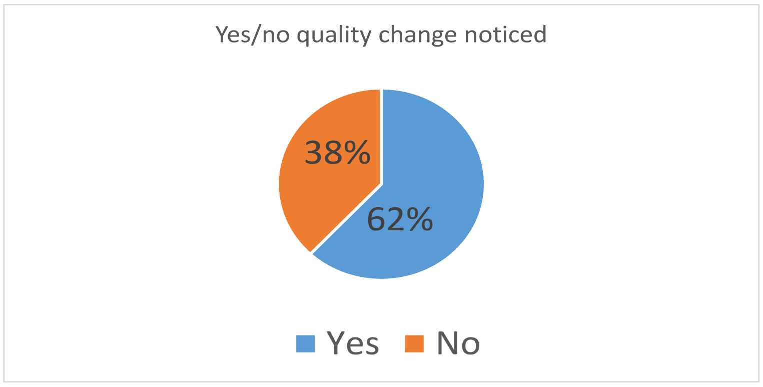


Figure 5.6: Yes/no amount of answers for question 2

People answered 121 times yes to this question and 75 times they answered no to this question. We can conclude no specific findings from this data.

5.7.3 Question 3: Give a score of how different the qualities were in the video.

For this question we have selected only the 121 samples when people actually saw a quality change (see Section 5.7.2). For this question, the difference between qualities in the video is measured with a score from 1 to 5. 1 represents very different and 5 represents no difference. We expect that the results are lower for the distribution logic D and high for the distribution logic A. The aggregated values of this question are shown in Figure 5.7.

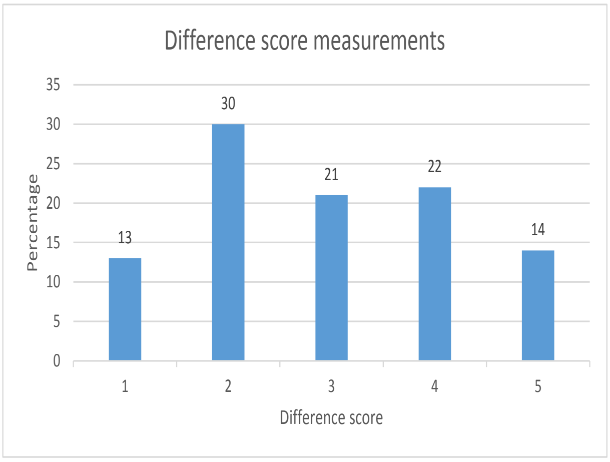


Figure 5.7: Amount of answers per difference score (1 to 5) for ODV tiled video

To do the three-way ANOVA test, we check the homogeneity of the variances. Levene's test did

	DF	Sum Sq	Mean Sq	F Value	Pr(>F)	
Logic	3	51.53	17.18	16.39	7.57e-9	***
Duration	1	0.01	0.01	0.01	0.94	
Bandwidth	1	16.61	16.62	15.85	1.24e-4	***
Logic:Duration	3	9.89	3.30	3.15	0.03	*
Logic:Bandwidth	2	0.75	0.37	0.36	0.70	
Duration:Bandwidth	1	2.00	2.00	1.91	0.17	
yLogic:Duration:Bandwidth	1	0.60	0.61	0.58	0.45	
Residuals	108	113.20	1.05			

Table 5.9: Three-way ANOVA test for difference score

not show a violation of homogeneity of variances (F(12,108)=0.8, p=0.6497). We can assume the homogeneity and continue with the three-way ANOVA test. The results are shown in the next table. With three-way ANOVA, we found significant main effects of Logic and bandwidth on the difference score. We also found a significant interaction between Logic and Duration. We analyzed the effect of the logic factor in terms of level of the duration factor. We fixed the duration to one second to check if the difference scores for the logics differ from each other with one second segment duration. We get the following results: F(3,56) = 4.865, p=0.00446, $p < 0.05$ indicates that there is a significant difference between the distribution logics for one second segments. We investigate which logics differ from each other with the Tukey's post-hoc test. We get the following table:

Logics	diff	lwr	upr	p adj
B-A	-0.63	-1.83	0.56	0.51
C-A	-0.76	-1.82	0.29	0.23
D-A	-1.64	-2.79	−0.50	2.05e-3
C-B	-0.13	-1.24	0.98	0.99
D-B	-1.01	-2.21	0.18	0.12
D-C	-0.88	-1.94	0.18	0.14

Table 5.10: Results of Tukey's post-hoc test for difference scores per distribution logic for one second segment duration

In this table, we see that there is a significant difference between Logic D and A. The next figure shows the boxplot for every logic with one second segment duration.

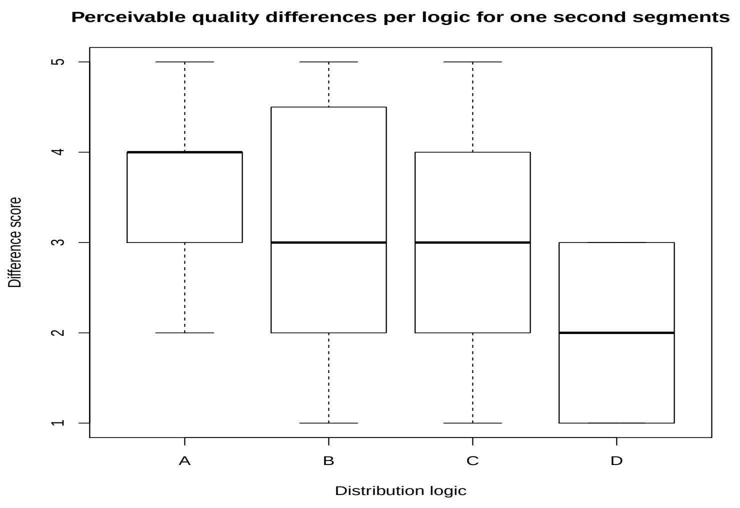


Figure 5.8: Perceivable quality differences per logic for one second segments

From the Tukey's test we see that distribution logic D and A differ from each other. In figure 5.8 we see that the boxplot for distribution logic D lies lower than the other distribution logics. So we can conclude that distribution logic D has a lower difference score than the other distribution logics. With distribution logic D, the users see more difference between qualities than with other distribution logics for one second segments. This is due to the black tiles that are displayed to the users. After the segment duration, measured in seconds, the black tiles change from black to an image of the video. In the next table, the mean and variance values of the difference score for every distribution logic with one second segment duration is shown. We see that the values for distribution logic B are very widespread in comparison with the other ones.

Logics	**Mean**	**Variance**
A	3.71	1.14
B	3.08	2.27
C	2.95	1.20
D	2.07	0.84

Table 5.11: Mean and variance values of difference score for every distribution logic with one second segment duration

For the next test we fixed the segment duration to two seconds to compare the quality difference scores for the distribution logics with two second segments. We found a significant difference between the logics for two seconds segments (F(3,57) = 12.8, p = 1.67e-06). With Tukeys post-hoc method, we can determine which logics differ from each other, the results are:

Logics	diff	lwr	upr	p adj
B-A	-0.44	-1.43	0.55	0.65
C-A	0.32	-0.69	1.33	0.83
D-A	-1.92	-2.99	−0.85	8.3e-5
C-B	0.76	-0.16	1.69	0.14
D-B	-1.48	-2.48	−0.49	1.23e-3
D-C	-2.24	-3.25	−1.24	1.20e-6

Table 5.12: Results of Tukey's post-hoc test for difference scores per distribution logic for two seconds segment duration

We see in the table that there is a significant difference between logics D and A, between D and B and between D and C. From the table we can see that distribituion logic D differs from the others and if we look to Figure 5.9 we see that the boxplot for D lies lower than the other ones. We can say that, like in the previous case, the users see more difference between qualities with distribution logic D than with other distribution logics. This is caused by the black tiles that change after the segment duration from black to an image of the video. In comparison with the previous figure, we see that the values for distribution logic D lie closer to one. We suspect that this is caused by the segment duration because with one second segment duration it takes longer to start playing the videos with the black screens. With one second segment duration it takes less time to start the videos with black screens. We see that the median of distribution logic A is smaller than the median of distribution logic C. This is contrary to our expectations but distribution logic A does have the second highest median.

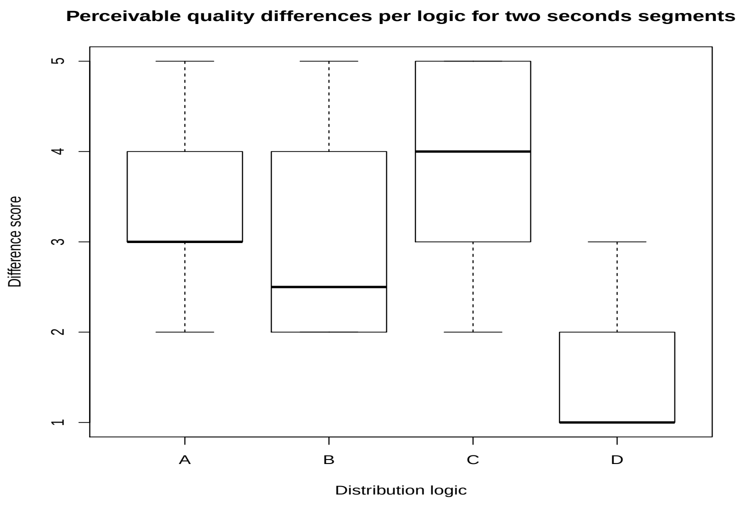


Figure 5.9: Perceivable quality differences per logic for two seconds segments

The mean and variance values of the quality difference scores per distribution logic for two seconds segments are shown in Table 5.13. We see that the scores of distribution logic B and C are more widespread than the others.

Logics	Mean	Variance
A	3.38	0.90
B	2.94	1.23
C	3.71	1.35
D	1.46	0.44

Table 5.13: Mean and variance values of difference score for every distribution logic with two seconds segment duration

Like said in Section 5.7.1, the participants noticed some artefacts in the video. When scoring the difference between the qualities, they said that when they saw a quality change between tiles because of the artefacts, they were not sure if they saw the same quality or a higher/lower quality of the video. So we suspect that the measured difference scores are, in some way, influenced by the artefacts in the videos.

5.7.4 Question 4: How disturbed were you with the quality change?

For this question we considered only the 121 samples were the users saw a quality change (see Section 5.7.2). We expect that the users are more disturbed with the distribution logic D because it is only tested with low bandwidth and tiles can stay black for a short time period. For the simple distribution logic, we expect that the users are not disturbed with the quality change. The rating distribution of this question, expressed in absolute figures, is shown in Figure 5.10.

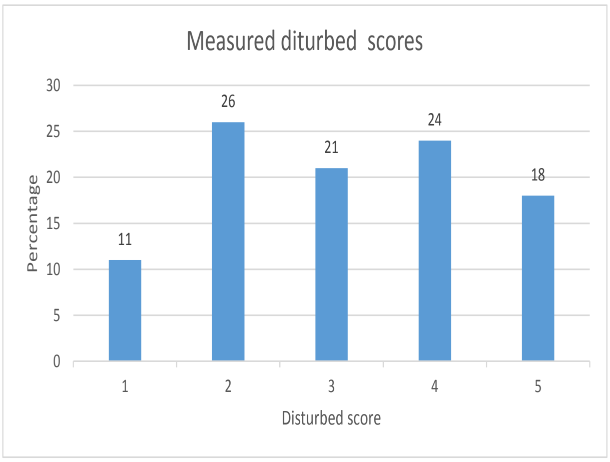


Figure 5.10: Values of how disturbed the users were with the quality changes

To execute the three-way ANOVA test, we first check the homogeneity of the variances. Levene's test did not show a violation of homogeneity of variances (F(12,108) = 1.7935, p=0.05801). We can continue with the three-way ANOVA test, the results are:

	DF	Sum Sq	Mean Sq	F Value	Pr(>F)	
Logic	3	55.01	18.34	21.72	4.40e-11	***
Duration	1	0.68	0.68	0.81	0.37	
Bandwidth	1	32.77	32.77	38.83	9.13e-09	***
Logic:Duration	3	19.06	6.35	7.52	1.27e-4	***
Logic:Bandwidth	2	2.10	1.05	1.25	0.29	
Duration:Bandwidth	1	1.04	1.04	1.23	0.27	
Logic:Duration:Bandwidth	1	0.55	0.55	0.65	0.42	
Residuals	108	91.16	0.84			

Table 5.14: Three-way ANOVA test for disturbed scores

Logics	diff	lwr	upr	p adj
B-A	0.16	-0.68	1.00	0.96
C-A	1.06	0.21	1.91	0.01
D-A	-1.77	-2.67	−0.86	1.82e-5
C-B	0.91	0.12	1.69	0.02
D-B	-1.92	-2.77	−1.09	6-e7
D-C	-2.83	-3.68	−1.98	1-e8

Table 5.15: Results of Tukey's post-hoc test for disturbed scores per distribution logic for two second segments

With three-way ANOVA, we found significant main effects of Logic and bandwidth on the disturbed score. We also found a significant interaction between Logic and Duration. We analyzed the effect of logic factor in terms of level of the duration factor. We fixed the duration factor to one second to compare the disturbed values for the distribution logics with one second segments. We get $F(3,56) = 2.276$, $p = 0.0896$, $p > 0.05$ indicates that there is no significant difference between the distribution logics for one second segments. Next we fixed the duration factor to two seconds to compare the disturbed values for the distribution logics with two seconds segments. We get $F(3,57) = 26.43$, $p = 7.65e\text{-}11$, $p < 0.05$ indicates that there is a significant difference between the distribution logics for two second segments. We investigate which logics differ from each other with Tukey's post-hoc test. The results are shown in Table 5.15.

We see that there is a significant difference between distribution logic C and A, D and A, C and B, D and B and D and C. In Figure 5.11 we see the boxplot for every distribution logic. We notice in this case that the median score of distribution logic A is smaller than distribution logic C. We expected that the value of distribution logic A would be larger than the other ones.

For this graph we compared the disturbed values for the distribution logics for two second segments. From the Tukey's test and Figure 5.11 we can see that distribution logic D is different from the others. Distribution logic D has the lowest disturbed score because the boxplot lies lower than the other ones. This is caused by the transition of the black tiles from black to an image. Because the transition is very clear when the users pan, they are more disturbed. The two second segment duration ensured that the transition was even larger because it took longer time to show an image for the black tiles. The mean and variance values are shown in the table 5.16.

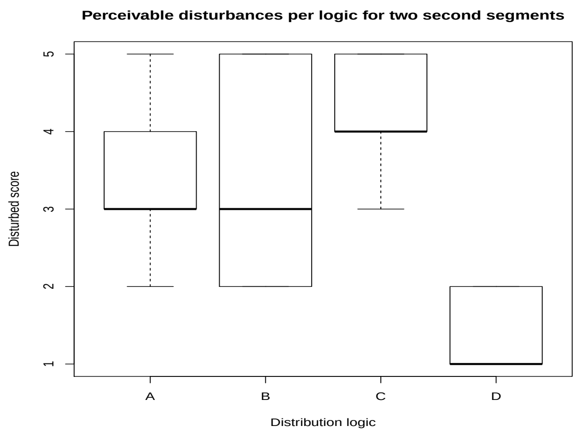


Figure 5.11: Amount of answers per disturbed score (1 to 5) for ODV tiled video

Logics	Mean	Variance
A	3.23	0.86
B	3.39	1.43
C	4.29	0.35
D	1.46	0.27

Table 5.16: Mean and variance values of disturbed score for every distribution logic with two seconds segment duration

5.7.5 Question 5: Did you find the duration of the quality change acceptable?

For this question we also took the 121 values from which the users saw a quality change. The aggregated results are given in Figure 5.12.

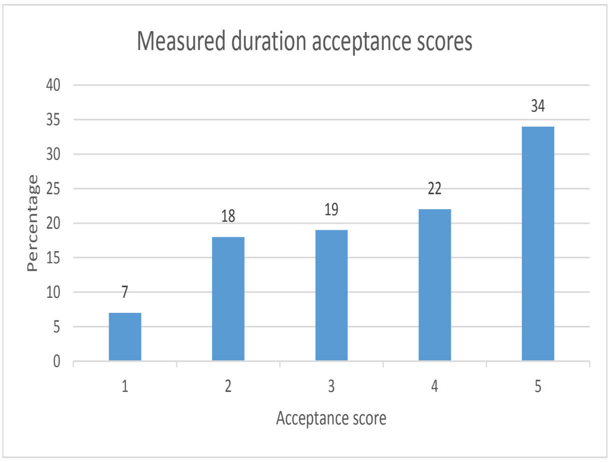


Figure 5.12: Segment duration acceptance scores

For the acceptance score of the duration of the quality change, we can not do a three-way ANOVA test because there is no assurance of the homogeneity of the variances. Levene's test did show a violation of homogeneity of variances. The results are: F(12,108) = 2.0908, p = 0.02322, $p < 0.05$ indicates that there is no homogeneity of the variances. An approach for this is to transform the data with the commonly used functions, logarithm or second power. For logarithm this gives us the results: F(12,108) = 2.6146, p = 0.004248 and for second power function: F(12,108) = 2.398, p = 0.008656. So we can conclude in both cases that $p < 0.05$ and that there is no homogeneity of the variances.

In general we can say that distribution logic C scores the best on almost every question and distribution logic D scores the lowest. We can conclude that distribution logic D is the least appreciated logic and C is the most appreciated logic.

Chapter 6

Conclusion

In this last chapter we make a conclusion to end this thesis. Before starting this thesis, we were wondering what impact the bandwidth has on the QoE of the users in the specific context of video streaming services and which improvement we could achieve on the QoE by inteligently allocating the available bandwidth budget. We implemented an MPEG-DASH streaming framework in JavaScript, hereby taking into account important lessons learned from a literature review. We designed a framework where every part can be extended or replaced by alternative implementations serving different purposes. For example, in our implementation, the data is downloaded via HTTP. We made the framework in such a way that the data could be downloaded using other protocols, simply by substituting the segment downloader module in the framework's software architecture. For video multistreaming, which is streaming video in parallel to one client, we have investigated two popular video players (Microsoft Smooth Streaming player and Netflix player) and presented two approaches (equal bandwidth division and priority-based bandwidth division) to distribute the available bandwidth over the streaming video players. We showed how the bandwidth distribution logics react to changing parameters, like the available bandwidth and the amount of streaming video players. For ODV tiled streaming, we explained what the main idea is and how this concept has been standardized in MPEG-DASH by means of the SRD extension. We discussed our approach and extended our MPEG-DASH framework with bandwidth distribution logics for ODV tiled streaming. Each of these bandwidth distribution logics implements a specific strategy to adaptively assign bandwidth (and hence visual quality) to the different spatial tiles that jointly compose the full ODV image. To subjectively and qualitatively measure the performance of every ODV tiled streaming bandwidth distribution logic in terms of QoE, we did user testing. We discussed the results of the user study and did three-way ANOVA tests to define the effects of the independent variables on the user-perceived video quality. We showed that specific distribution logics are better in certain situations and that the quality depends on the available bandwidth. In the next two subsections, we will further discuss what we have learned from the user testing and define the future work for video multistreaming and ODV tiled streaming applications.

6.1 User study insights

A first important finding of our study was that the QoE was better for the *viewport at highest quality* distribution logic (see Section 5.7 and its subsections). By this we know that the users think the quality of the video is better when the quality transition between adjacent tiles is not so obvious. When the quality transition is apparent, they rated the quality worse. Because the users see more quality difference between tiles, the quality score and QoE of the users is lower. Secondly, we learned that there can be a quality difference between tiles encoded at the same bitrates. These quality differences were more noticeable with the *simple* distribution logic. With little motion in the video, the quality of the video is better compared to that of a video which is encoded at the same bitrate, but has more motion in it. This can introduce some errors while measuring the subjective video quality. Third, We expected that the quality score for low bandwidth would be lower than for high bandwidth. This hypothesis turned out to hold, as the users gave the quality for low bandwidth a lower score in comparison with the quality for high bandwidth. For the viewport only distribution logic we expected the quality score to be low as we only tested this logic under low bandwidth conditions. Like said before, we realized after the user tests that we should not have excluded the tests for the viewport only distribution logic with high bandwidth. In our case, the influence of the black tiles is not clear because we did the user tests with low bandwidth for the viewport only distribution logic. With the viewport only distribution logic, only the tiles of the viewport are streamed, the other tiles are not streamed and stay black. After PTZ operations, it takes some time before the new tiles in the viewport start playing (see section 4.6.7). The last thing that we can conclude is that the black tiles ensure that the quality difference is more pronounced than when the tiles keep playing. This finding confirms our initial hypothesis that the presence of black tiles has a detrimental impact on the perceived video quality.

In the user tests we subjectively and qualitatively measured the quality of the video in general, the difference between the qualities and how disturbed the particpants were with the quality difference. The last thing we measured is the acceptance of the duration quality change. The duration of the quality change is the time that is elapsed after the PTZ operation to change the quality of the tiles. These variables were measured with a score from 1 to 5 (see Appendix C). For the simple distribution logic, we expected that the users would see no difference in quality because the quality is the same for all the tiles. We learned, like said before, that the video can show some artefacts caused by fluctuating quality levels between different spatial regions within the same ODV frame, and this can cause the quality score to be lower. This also applies to the quality difference score, the score can be influenced because the user is not sure if the quality is different or there are artefacts in the video. The quality difference is less perceived with the *delta* and *viewport at high quality* distribution logics. The findings from the user study confirm that quality changes are more/less noticed in those distribution logics that introduce large/small quality differences between adjacent tiles

The disturbance scores showed that the users were more disturbed with the *viewport only* distribution logic. We see that this is caused by the black tiles not displaying an image. So the users are more disturbed when they need to wait for (parts of) the video. This conclusion is also applicable to the acceptance score of the duration of quality change. With the *viewport only* distribution logic, after a PTZ operation, the tile videos are displayed to the user after some time period. The larger the segment duration, the longer it takes to start playing the black tiles. Therefore the quality change duration acceptance score of the *viewport only* distribution logic is lower. We expected the acceptance score for the *viewport only* distribution logic to be lower because the time spent to start playing video players is more noticeable than the time spent to change the quality. This is because with the quality change, images are displayed to the user and with the *viewport only* distribution logic, black parts are displayed to the users.

6.2 Future work

In this section we will discuss future work for video multistreaming and ODV tiled streaming. Bandwidth distribution in video multistreaming applications plays a crucial role for the QoE of the users.

The bandwidth distribution logics that we have presented for video multistreaming take the available bandwidth into account, but not the quality. Future work for this is to make a player that can dynamicly adjust the waiting time before switching between qualities, however without causing the client-side video buffer to underrun. When the bandwidth reduces dramatically, the buffer can underrun because the delivery of segments is delayed. The time to wait until the quality is changed is influenced by the time that the player notices the bandwidth change. This can not be too soon because then the quality is immediately changed. The player must also pay attention to other parameters like the initial/startup delay. A long initial/startup delay will introduce a lower QoE.

Bandwidth distribution for ODV tiled streaming is necessary for an application to be bandwidth friendly. For ODV tiled streaming, we have discussed several distribution logics to have a basic approach for dividing the available bandwidth over the tiles. These logics focus on the available bandwidth in a network and on the viewport of the user. The distribution logics try minimizing the time required to upgrade the quality of visible tiles when the ODV viewport has changed. These approaches only look at the available banwidth and the viewport, but there can be other approaches to take. A dynamic approach can be taken where the users or developers of a 360 degree application determine the main idea of the distribution logic. For example when the users or developers want the tiles in the viewport to be streamed at the same uniform quality, there can be a rule specifying this approach. Another approach could be that tiles around the viewport are very important in scenarios where panning is more done than holding the viewport stable. A manager could look at the rules to decide which distribution logic best fits the application scenario at hand. The manager maintains all the rules and knows which distribution logic is better in which situation. In our case with bandwidth distribution, the manager is aware of how the different bandwidth distribution logics will assign a certain bandwidth budget to the different tiles composing the ODV frame. It can then decide based on the available bandwidth and the rules which distribution logic to choose. The rules can be inserted like a formula that decribes which parts are important for the application. The different constituting factors in the formula could for example be weighed by means of percentage values. The total count of the percentages must be 100%. By this we can insert as many pecentages that as need. For example a percentage for the focus on the viewport, a percentage for the quality of the adjacent tiles and a percentage for the quality of all the other tiles. By increasing the percentage, more bandwidth is allocated to the specific rule. 100% would mean that the available bandwidth must integrally be exploited to maximize the quality factor that is associated with the involved rule. This looks like our approach of the priority-based distribution logic. But this would be combined with the rules that can be inserted. This dynamic approach could make the application more usefull to all sorts of users and developers. Developers could make their formula perfect for their application and the manager applies the formula for the distribution logics it maintains. The developers could add logics to the manager and extend the choices of distribution logics.

Appendix A

Content preparation

FFMPEG for multistreaming video:

- ffmpeg -i video.mp4 -an -vcodec libx264 -x264opts keyint=30 -x264opts min-keyint=30 -x264opts scenecut=0 -vb 400000 -r -g 30 video-400000.mp4

FFMPEG for ODV tiled streaming:

- ffmpeg -i video.mp4 -an -vcodec libx264 -x264opts keyint=30 -x264opts min-keyint=30 -x264opts scenecut=0 -vb 400000 -r -g 30 -vf crop=600:200:0:0 video-part-0-0-400000.mp4

Now we will discuss the parameters that are used with the FFMPEG command to transcode the video into multiple qualities.

- -i video.mp4

  Theoption -i we specifies the input video that is transcoded into multiple qualities.

- -an

  -an disables audio.

- -vcodec libx264

  -vcodec is used to specify the codec that is used to encode the video.

- -x264opts

  The option -x264opt can only be used when the media is encoded with the codec libx264.

- -x264opts keyint=30

  With -x264opts keyint we specify the GOP length of the video.

- -x264opts min-keyint=30

  Option -x264opts min-keyint specifies the minimum GOP length of the video.

- -x264opts scenecut=0

  The no-scenecut option is to keep x264 from generating a key frame when there is a scene cut in the video [14].

- -vb 400000

  With option -vb we specifiy the bitrate of the video. A higher bitrate results in a high quality video and a lower bitrate results in a lower quality video.

- -r 30

  Option -r specifies the frame rate of the video.

- -g 30

  -g is used to define the directive for the GOP length for the video.

- -vf crop=600:200:0:0

  With option -vf crop we can crop the video. Making use of this option we can spatially divide the full video frame into tiles.

  The parameters measured in pixels for the crop=x:y:v:w option are:

  x: Stands for the width of the video that is cropped out of the full video.

  y: Stands for the height of the video that is cropped out of the full video.

  v: Stands for the start x position starting at the top left corner of the full video.

  w: Stands for the start y position starting at the top left corner of the full video.

- video-part-0-0-400000.mp4

  The last parameter is the output file.

MP4Box command:

- MP4Box -dash 2000 -segment-name segment -out template.mpd -rap -frag 2000 video-400000.mp4#video

The command consists out of these parameters [4]:

- -dash 2000

  Enables DASH segmentation of input files with the given segment duration, expressed in ms. For onDemand profile, where each media presentation is a single segment, this option sets the duration of a subsegment .

- -segment-name segment

  Sets the file name for generated segments .

- -out template.mpd

  Specifies the output file name for the generated MPD.

- -rap

  Forces segments to begin with random access points. Segment duration may not be exactly what is asked by -dash switch since encoded video data is not modified.

- -frag 2000

  Specifies the duration of subsegments in ms.

- video-400000.mp4#video

  Name of the input file. #video option defines that the input data is video content.

Appendix B

Basic framework implementation

In Figure B.1, the Unified Modelling Language class diagram of the basic framework is shown.

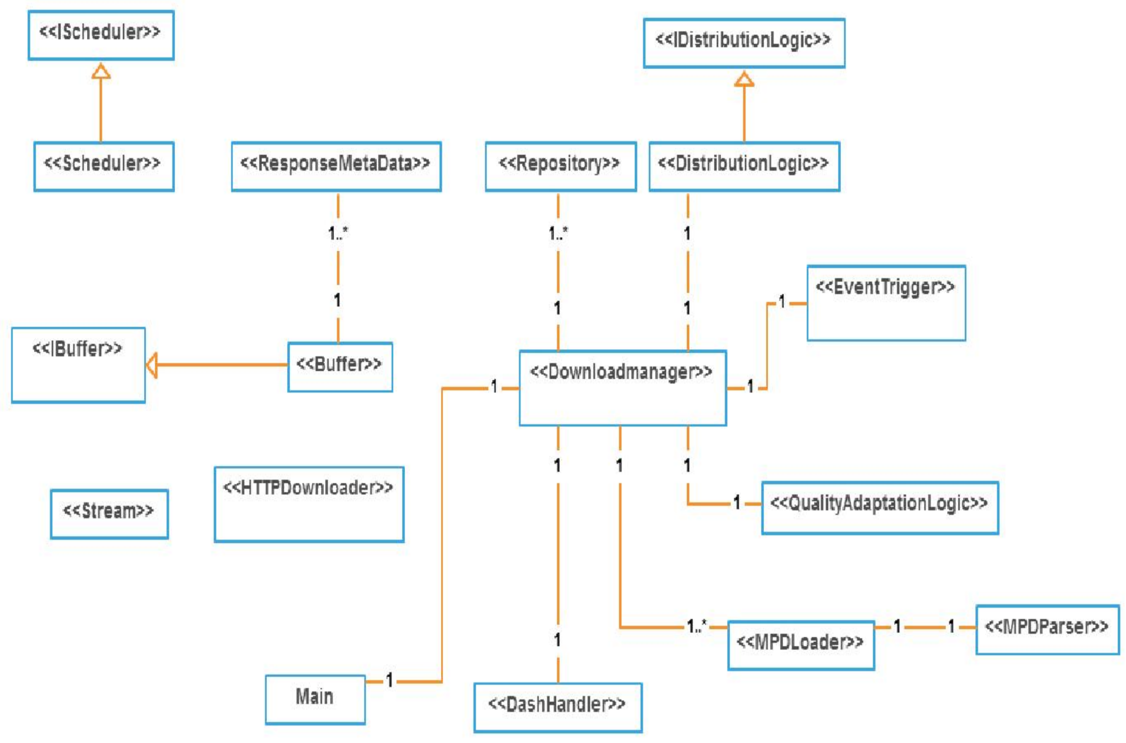


Figure B.1: UML class diagram of basic MPEG-DASH implementation

The major classes of this frame are:

- **Downloadmanager:**
  - There is only one object of the Downloadmanager class.

  - Creates the necessary objects for every stream. These objects are the MPD objects, the Scheduler objects, the Buffer objects, the HTTPDownloader objects and the Stream objects.
  - Maintains the repositories for all the necessary objects.
  - Triggers the HTTPDownloader objects to download the MPD file and the initial file.
  - Triggers the Scheduler object(s) to begin scheduling.
  - Starts playing each video player when the player is ready and has enough data.
  - Adds the downloaded video segments to the correct sourcebuffer.
  - Decides when the information of the video segments is updated.
  - Asks the QualityAdaptationLogic which quality of video segments to download on the basis of the allocated bandwidth per stream.
  - Triggers the HTTPDownloader object(s) to download video segments when the scheduler event is received for that HTTPDownloader.
  - Changes the resolution of the video players on the basis of the information perceived in the MPD file.

- **Buffer:**
  - For every stream a Buffer object is made.
  - Creates a new MediaSource object per stream and adds the URL to the right DOM video element.
  - Adds a sourcebuffer with a specific type and codecs to the MediaSource object when the Downloadmanager object calls this function.
  - Stores all the downloaded video segments in the sourcebuffer.
  - Updates and stores the information about every downloaded video segment when the Downloadmanager object calls this function. The information that is maintained, is the start time, duration and end time of every video segment, measured in seconds.

- **Scheduler:**
  - For every stream one Scheduler object is made.
  - Triggers a schedule event to the Downloadmanager object.
  - Decides how many video segments to download.

- **DistributionLogic:**
  - One DistributionLogic is made per scenario of the application.
  - Divides the available bandwidth over the streams requesting bandwidth.
  - Decides if there is enough bandwidth for the video players to start streaming.

- **QualityAdaptation:**
  - One QualityAdaptation object is made per Downloadmanager object.
  - Calculates the highest possible quality of video that can be downloaded with the allocated bandwidth per stream.

Now we will discuss the other classes of framework:

- **HTTPDownloader:**
  - For every stream one HTTPDownloader object is made.
  - Sends the Ajax [32] GET requests to the full URL of the video segment.
  - Receives the video segments from the server.
  - Keeps track of the video segment ID.

- **Repository:**
  - For every type of object per stream there is an Repository object made. These objects are DOM video elements, MPD objects, Buffer objects, Scheduler objects, HTTPDownloader objects and Stream objects.
  - Keeps track of unique objects via a dictionary.

- **EventTrigger:**
  - Per Downloadmanager object, there is one EventTrigger object made.
  - Let the Downloadmanager object know when an HTTPDownloader has finished downloading a video segment.

- **MPDLoader:**
  - Per stream an MPDLoader object is made.
  - Triggers the HTTPDownloader to download the MPD file.
  - Triggers the MPDParser to parse the downloaded MPD file.

- **MPDParser:**
  - Per MPDLoader an MPDParser object is made.
  - Parses the received MPD file to an object where the parameters and attributes are stored in like properties of the object.

- **DashHandler:**
  - Per Downloadmanager object one DashHandler object is made.
  - Returns the parameters defined in the MPD file. These parameters are the width and height of the video, the segmentlist length, all representations in the MPD file, the segment duration, the minimum buffer time, the codecs type, the bandwidth for a specific quality defined in the MPD file and the amount of qualities in the MPD file.

- **ResponseMetaData:**
  - Per downloaded video segment an ResponsMetaData object is made.
  - Stores information about the downloaded video segments in an object, like the start time, duration and end time of the video segments, measures in seconds.

- **IBuffer, IScheduler and IDistributionLogic:**
  - Interface classes where the Buffer, Scheduler and DistributionLogic classes have inherited from.

Appendix C

User testing questions

Starting questionnaire:

What is your age?

- ☐ 12 - 21
- ☐ 22 - 31
- ☐ 32 - 41
- ☐ 42 - 51
- ☐ $>$ 52

What is your gender?

- ☐ Male
- ☐ Female

How often do you watch videos on the internet (Youtube, Netflix, Vimeo, ...)?

- ☐ Never
- ☐ Annually (rarely)
- ☐ Monthly
- ☐ Weekly
- ☐ Daily

Do you know what 360 degree video is?

- ☐ Yes
- ☐ No

How often do you watch 360 degree video on the internet?

- ☐ Never
- ☐ Annually (rarely)
- ☐ Monthly
- ☐ Weekly
- ☐ Daily

How often do you watch live videos on the internet?

- ☐ Never
- ☐ Annually (rarely)
- ☐ Monthly
- ☐ Weekly
- ☐ Daily

Test scenario questionnaire:

Give a score of 1-5 how do you found the quality of the video.

	1	2	3	4	5	
Bad	☐	☐	☐	☐	☐	very good

Have you noticed any quality change in the video?

- ☐ Yes
- ☐ No (Skip next 3 questions)

Give a score of how different the qualities were in the video.

	1	2	3	4	5	
Very different	☐	☐	☐	☐	☐	No difference

How disturbed were you with the quality changes?

	1	2	3	4	5	
Very disturbed	☐	☐	☐	☐	☐	Not disturbed

Did you find the duration of the quality change acceptable?

	1	2	3	4	5	
Not acceptable	☐	☐	☐	☐	☐	Acceptable

Do you have any remarks?

Appendix D

Equally dividing distribution logic for video multistreaming

We declare the following variables for this distribution logic:

$countStreams$: Holds the total number of streaming video players.
$bandwidthAllocations$: Dictionary that stores the bandwidth allocations for every stream.
$streamBooleans$: Dictionary that stores booleans for every stream identifying whether the stream is currently playing or not.
$totalBandwidth$: Holds a number representing the total available bandwidth.
$streamBandwidth$: Holds a number allocating the bandwidth per stream.

The streams have a numerical indentifier which is used to store the allocated bandwidth in the *bandwidthAllocations* dictionary and to store the boolean for the stream in the *streamBooleans* dictionary. The first stream has identifier 0, the second stream has identifier 1, etcetera. So the id of the stream represents the key for retrieving the corresponding value. The function $addStream(streamId)$ is called when a video player starts streaming or resumes playback. The $removeStream(streamId)$ function is called when a video player stops streaming or pauses. The pseudocode for this distribution logic is shown next.

```
function ASSIGNBANDWIDTHTOACTIVESTREAMS(streamBandwidth)
    for $i \leftarrow 0, i < length(bandwidthAllocations), i++$ do
        if $streamBooleans[i] == TRUE$ then
            $bandwidthAllocations[i] \leftarrow streamBandwidth$
        end if
    end for
end function

function ADDSTREAM(streamId)
    $countStreams \leftarrow countStreams + 1$
    $streamBooleans[streamId] \leftarrow TRUE$
    $bandwidthAllocations[streamId] \leftarrow 0$
    $streamBandwidth \leftarrow (totalBandwidth/countStreams)$

    ASSIGNBANDWIDTHTOACTIVESTREAMS(streamBandwidth)
```

```
16: end function
17:
18: function REMOVESTREAM(streamId)
19:     $countStreams \leftarrow countStreams - 1$
20:     $streamBooleans[streamId] \leftarrow FALSE$
21:     $bandwidthAllocations[streamId] \leftarrow 0$
22:     $streamBandwidth \leftarrow (totalBandwidth/countStreams)$
23:
24:     ASSIGNBANDWIDTHTOACTIVESTREAMS(streamBandwidth)
25: end function
```

Appendix E

Priority-based distribution logic

Like the distribution logic in Appendix D, this distribution logic makes use of the *countStreams*, *bandwidthAllocations* and *totalBandwidth* variables. Besides those variables, we declare the following additional ones:

streamPercentages: Dictionary to store all percentages for every stream. The actively watched streams get two times the priority of the streams that are not actively watched.
streamsDict: Dictionary storing all the streams.
boolActive: Per stream a boolean is stored declaring if the stream is watched actively or not.
activeStreams: Stores the number of streams that are watched actively.
multiplyValue: The value with which the active video players' priorities are multiplied.
partPercentage: Part of the total percentage that is assigned to streams.

The pseudocode for the priority distribution logic shown on the next.

```
 1: $multiplyValue \leftarrow 2$
 2:
 3: function CALCULATEBANDWIDTHPERSTREAM(multiplyValue)
 4:     DETECTPRIORITIES(multiplyValue)
 5:     ALLOCATEBANDWIDTH
 6: end function
 7:
 8: function DETECTPRIORITIES(multiplyValue)
 9:     $partPercentage \leftarrow 100/((activeStreams * multiplyValue) + (countStreams-$
10:     $activeStreams))$
11:     for $i \leftarrow 0, i < countStreams, i++$ do
12:         if $streamsDict[i].boolActive == TRUE$ then
13:             $streamPercentages[i] \leftarrow partPercentage * multiplyValue$
14:         else
15:             $streamPercentages[i] \leftarrow partPercentage$
16:         end if
17:     end for
18: end function
19:
20: function ALLOCATEBANDWIDTH
```

```
21:     if $countStreams == 1$ then
22:         $bandwidthAllocations[streamId] \leftarrow totalBandwidth$
23:     else
24:         for $i \leftarrow 0, i < length(bandwidthAllocations), i++$ do
25:             $bandwidthAllocations[i] \leftarrow floor(totalBandwidth * streamPercentages[i])$
26:         end for
27:     end if
28: end function
```

Appendix F

Detecting layers for tiles algorithm

```
function CHECKTILES
    $repoDistances \leftarrow newRepository()$
    $cntTiles \leftarrow amountTilesInFullVideoFrame$

    for $i \leftarrow 0, i < cntTiles, i++$ do
        $repoDistances[i] \leftarrow cntTiles$
    end for

    $width \leftarrow widthofviewport$
    $height \leftarrow heightofviewport$
    $startX \leftarrow startXcoordinateofviewport$
    $startY \leftarrow startYcoordinateofviewport$
    $endX \leftarrow endXcoordinateofviewport$
    $endY \leftarrow endYcoordinateofviewport$
    $rows \leftarrow amountOfRowsInFullVideoFrame$
    $cols \leftarrow amountOfColumnsInFullVideoFrame$
    $videoWidth \leftarrow widthOfFullVideoFrame$
    $videoHeight \leftarrow heightOfFullVideoFrame$
    $xPos \leftarrow startX$
    $yPos \leftarrow startY$

    while $xPos <= endY$ do
        while $xPos <= endX$ do
            $xIndex \leftarrow floor(xPos/width) * rows$
            $yIndex \leftarrow floor(yPos/height)$
            $index \leftarrow (xIndex + yIndex)\%(cntTiles)$
            $repoDistances[index] \leftarrow 0$
            if $xPos == endX$ then
                $xPos \leftarrow xPos + with$
            else if $(xPos + width) > endX)$ then
                $xCor \leftarrow xPos + (endX - xPos)$
                if $floor(xCor/width) == floor(xPos/width)$ then
                    $xPos \leftarrow xPos + (endX - xPos) + 1$
                else
                    $xPos \leftarrow xPos + (endX - xPos)$
                **end if**
            **else**
                $xPos \leftarrow xPos + width$
            **end if**
            $xPos \leftarrow startX$
            **if** $yPos == endY$ **then**
                $yPos \leftarrow yPos + height$
            **else if** $(yPos + height) > endY$ **then**
                $yCor \leftarrow yPos + (endY - yPos)$
                **if** $floor(yCor/height) == floor(yPos/height)$ **then**
                    $yPos \leftarrow yPos + (endY - yPos) + 1$
                **else**
                    $yPos \leftarrow yPos + (endY - yPos)$
                **end if**
            **else**
                $yPos \leftarrow yPos + height$
            **end if**
        **end while**
    **end while**

    $tileLayer \leftarrow 0$
    $count \leftarrow amountOfTilesInViewport$

    **while** $count < cntTiles$ **do**
        **for** $k \leftarrow 0, i < ctnTiles, i++$ **do**
            **if** $repoDistances[i] == tileLayer$ **then**
                $tileX \leftarrow floor(k/rows) * width$
                $tileY \leftarrow floor(k\%rows) * height$
                **for** $i \leftarrow -1, i <= 1, i++$ **do**
                    **for** $j \leftarrow -1, j <= 1, j++$ **do**
                        $xPoint \leftarrow tileX + (j * width)$
                        $yPoint \leftarrow tileY + (i * height)$
                        **if** $yPoint > -1 AND yPoint < videoHeight$ **then**
                            $xIndex \leftarrow floor(xPoint/width) * rows$
                            $yIndex \leftarrow floor(yPoint/height)$
                            $index \leftarrow (xIndex + yIndex)\%cntTiles$
                            **if** $index < 0$ **then**
                                $index \leftarrow index + cntTiles$
                            **end if**
                            **if** $repoDistances[index] > (tileLayer + 1)$ **then**
                                $count \leftarrow count + 1$
                                $repoDistances[index] \leftarrow tileLayer + 1$
                            **end if**
                        **end if**
                    **end for**
                **end for**
            **end if**
        **end for**
```

```
        $tileLayer \leftarrow tileLayer + 1$
    end while
    DISTRIBUTIONLOGIC.CALCULATEBANDWIDTH
    Call checkTiles() after 1000/30 milliseconds
end function
```

Appendix G

Detecting viewport change algorithm

This algorithm determines when the viewport has changed, so when the viewer pans in the full video frame. For this function we explain and declare the following functions and variables:

$canvasDrawer.startPlaying()$: The canvasDrawer object starts playing every video.

$canvasDrawer.startDrawing()$: The canvasDrawer object starts drawing the images of every video to the canvas of the full video frame.

$viewerWrapper.checkTiles()$: Function of Appendix F.

$prevVwSnapshot$: Holds the index numbers of the tiles that were in the previous viewport.

$curVWSnapshot$: Holds the index numbers of the tiles that are in the current viewport.

$betweenSegDelay$: Value that is added or subtracted from the normal end time of the received viewport tile segments.

$startBoundary$: The end time of every tile video segment must be larger than this value. $startBoundary$ = the normal end time of the received viewport tile segments - $betweenSegDelay$.

$endBoundary$: The end time of every tile video segment must be smaller than this value. $endBoundary$ = the normal end time of the received viewport tile segments + $betweenSegDelay$.

We use $endBoundary$ and $startBoundary$ to detect which tiles are currently in the viewport. Otherwise explained, the end time of the received tile segment must lie between these boundaries. By this way, we know which tiles lie in the viewport because data for the other tiles is not transferred to the client. We use these boundaries because the end time of the segments of the viewport tiles is not always equal to each other.

$delay$: Some delay to add to the highest end time to be sure that the current time of the video players is set to a time where every video player has data to play.

```
function CHECKPERINTERVAL
    if timeCalled == 1 then
        $canvasDrawer.startPlaying()$
        $canvasDrawer.startDrawing()$
        $viewerWrapper.checkTiles()$

        for $i \leftarrow 0, i < ctntiles, i++$ do
            if $repoDistances[i] == 0$ then
                $prevVwSnapshot.push(i)$
            end if
        end for
    else if $timeCalled > 1$ then
        $endTime \leftarrow timesCaled * segmentDuration$
        $curVwSnapshot \leftarrow newArray()$
        for ( do$i \leftarrow 0, i < ctnTiles, i++$)
            $tileEndTime \leftarrow repoBuffers.getElementAt(i).getEndTimeOfCurrentBuffer()$
            $startBoundary \leftarrow (endTime - betweenSegDelay)$
            $endBoundary \leftarrow (endTime + betweenSegDelay)$
            if $tileEndTimebetweenstartBoundaryANDendBoundary$ then
                $curVwSnapshot.push(i)$
            end if
        end for

        $vwChanged \leftarrow false$

        for $i \leftarrow 0, i < length(curVwSnapshot), i++$ do
            if $!prevVwSnapShot.contains(curVwSnapShot[i])$ then
                $vwChanged \leftarrow true$
            end if
        end for

        if $!vwChanged$ then
            for $i \leftarrow 0, i < length(prevVwSnapshot), i++$ do
                if $!curVwSnapshot.contains(prevVwSnapshot[i])$ then
                    $vwChanged \leftarrow true$
                end if
            end for
        end if

        if $vwChanged == true$ then
            $lastTime \leftarrow -1$
            for $i \leftarrow 0, i < length(curVwSnapshot), i++$ do
                $tileIndex \leftarrow curVwSnapshot[i]$
                $endTimeTile \leftarrow repoBuffers.getElementAt(tileIndex).getEndTimeOfCurrentBuffer()$
                if $lastTime < endTimeTile$ then
                    $lastTime \leftarrow endTimeTile$
                end if
                $prevTileIndex \leftarrow prevSnapShot[0]$
                $endTime \leftarrow repoVideoTags.getElementAt(prevTileIndex).currentTime$
                for $i \leftarrow 1, i < length(prevVwSnapshot), i++$ do
                    $curTimeTile \leftarrow repoVideoTags.getElementAt(i).currentTime$
                    if $endTime > curTimeTile$ then
                        $endTime = curTimeTile$
                    end if
                    $prevVwSnapshot = curVwSnapshot$
                    $diff \leftarrow lastTime - endTime$
                    if $diff < 0$ then
```

```
                    $diff \leftarrow 0$
                end if
                Call changeCurTime(curVwSnapshot, diff, lastTime, delay) function after
                diff seconds
            end for
        end for
    end if
end if

$timeCalled \leftarrow timesCalled + 1$
Call checkPerInterval() function after segmentDuration seconds
end function

function CHANGECURTIME(curVwSnapShot, diff, lastTime, delay)
    for $i \leftarrow 0, i < length(curVwSnapshot), i++$ do
        $tileIndex \leftarrow curVwSnapshot[i]$
        $repoVideoTags.getElementAt(i).currentTime \leftarrow lastTime + delay$
    end for
end function
```

Appendix H

Simple distribution algorithm

To understand the distribution logics, we have made some property variables for them:

- *arrayMpd*: dictionary that stores all the MPD URLs, the key is the tile number.
- *repoMPD*: dictionary that stores all the MPD objects, the key is the MPD URL for every video player.
- *availableBandwidth*: the currently available bandwidth
- *bandwidthAllocations*: dictionary that stores all the bandwidth allocations for all the tiles, the key is the tile number.
- *cntTiles*: stores the total amount of tiles in the full video frame.
- *dashHandler*: object of the class `DashHandler`
- *segmentDuration*: duration of the video segments
- *repoBuffers*: dictionary that stores all the buffers of the video players, the key is the tile number.
- *repoDistances*: holds the distance for every tile to the viewport.

```
function CALCULATEBANDWIDTH
    $firstMpdUrl \leftarrow arrayMpd[0]$
    $amountQualities \leftarrow dashHandler.getAmountOfQualitiesInMpd$
    $(repoMpds.getElementAt(firstMpdUrl))$
    $totalBandwidths \leftarrow newArray()$

    for $\leftarrow 0, i < amountQualities, i++$ do
        $totalBandwidths[i] \leftarrow 0$
        for $j \leftarrow 0, j < cntTiles, j++$ do
            $bandwidthAllocations[j] \leftarrow 0$
            $mpdUrl \leftarrow arrayMpd[j]$
            $totalBandwidths[j] \leftarrow totalBandwidths[j] + dashHandler.getBandwidthForQuality$
            $(i, repoMpds.getElementAt(mpdUrl))$
        end for
    end for

    $streamQuality \leftarrow -1$

    for $i \leftarrow 0, i < amountQualities, i++$ do
        if $totalBandwidths[i] <= availableBandwidth$ then
            $streamQuality \leftarrow i$
        end if
    end for
    if $streamQuality != -1$ then
        for
            $i \leftarrow 0, i < cntTiles, i++$ do
            $mpdUrl \leftarrow arrayMpd[i]$
            $allocatedBandwidth \leftarrow dashHandler.getBandwidthForQuality$
            $(streamQuality, repoMpds.getElementAt(mpdUrl))$
            $bandwidthAllocations[i] \leftarrow allocatedBandwidth$
        end for
    end if
end function
```

Appendix I

Viewport only distribution algorithm

```
function CALCULATEBANDWIDTH
    $firstMpdUrl \leftarrow arrayMpd[0]$
    $amountQualities \leftarrow dashHandler.getAmountOfQualitiesInMpd$
    $(repoMpds.getElementAt(firstMpdUrl))$
    $totalBandwidths \leftarrow newArray()$

    for $i \leftarrow 0, i < amountQualities, i++$ do
        $totalBandwidths[i] \leftarrow 0$
        for $j \leftarrow 0, j < cntTiles, j++$ do
            $bandwidthAllocations[j] \leftarrow 0$
            if $repoDistances.getElementAt(j) == 0$ then
                $mpdUrl \leftarrow arrayMpd[j]$
                $totalBandwidths[j] \leftarrow totalBandwidths[j] + dashHandler.getBandwidthForQuality$
                $(i, repoMpds.getElementAt(mpdUrl))$
            end if
        end for
    end for

    $streamQuality \leftarrow -1$

    for $i \leftarrow 0, i < amountQualities, i++$ do
        if $totalBandwidths[i] <= availableBandwidth$ then
            $streamQuality \leftarrow i$
        end if
    end for

    if $streamQuality != -1$ then
        for $i \leftarrow 0, i < cntTiles, i++$ do
            if $repoDistances.getElementAt[i] == 0$ then
                $mpdUrl \leftarrow arrayMpd[i]$
                $allocatedBandwidth \leftarrow dashHandler.getBandwidthForQuality$
                $(streamQuality, repoMpds.getElementAt(mpdUrl))$
                $bandwidthAllocations[i] \leftarrow allocatedBandwidth$
            end if
        end for
    end if
end function
```

Appendix J

Viewport at highest quality distribution algorithm

```
function CALCULATEBANDWIDTH
    $tempBandwidth \leftarrow availableBandwidth$

    for $i \leftarrow 0, i < cntTiles, i++$ do
        $mpdUrl \leftarrow arrayMpd[i]$
        $bandwidth \leftarrow dashHandler.getBandwidthForQuality(0, repoMpds.getElementAt(mpdUrl))$
        $bandwidthAllocations[i] \leftarrow bandwidth$
        $tempBandwidth \leftarrow tempBandwidth - bandwidth$
    end for

    if $tempBandwidth > 0$ then
        $mpdUrl \leftarrow arrayMpd[0]$
        $amountQualities \leftarrow dashHandler.getAmountOfQualitiesInMpd$
        $(repoMpds.getElementAt(mpdUrl))$
        for $startQuality = 1, startQuality < amountQualities, startQuality++$ do
            for $i = 0, i < cntTiles, i++$ do

                if $repoDistances[i] == 0$ then
                    $tempUrl \leftarrow arrayMpd[i]$
                    $neededBandwidth \leftarrow dashHandler.getBandwidthForQuality$
                    $(startQuality, repoMpds.getElementAt(tempUrl))$

                    if $tempBandwidth + bandwidthAllocations[i] - neededBandwidth > 0$ then
                        $tempBandwidth \leftarrow tempBandwidth + bandwidthAllocations[i] - neededBandwidth$
                        $bandwidthAllocations[i] \leftarrow neededBandwidth$
                    end if
                end if
            end for
        end for

        $totalBandwidths \leftarrow newArray()$

        for $i \leftarrow 0, i < (amountQualities - 1), i++$ do
            $totalBandwidths[i] \leftarrow 0$

            for $j \leftarrow 0, j < cntTiles, j++$ do

                if $repoDistances.getElementAt(j) > 0$ then
                    $mpdUrl \leftarrow arrayMpd[j]$
                    $totalBandwidths[i] \leftarrow totalBandwidths[i] - bandwidthAllocations[j]$
                    $totalBandwidths[i] \leftarrow totalBandwidths[i] + dashHandler.getBandwidthForQuality$
                    $(i + 1, repoMpds.getElementAt(mpdUrl))$
                end if
            end for
        end for

        $streamQuality \leftarrow -1$
        for $i \leftarrow 0, i < (amountQualities - 1), i++$ do
            if $totalBandwidths[i] <= tempBandwidth$ then
                $streamQuality \leftarrow i + 1$
            end if
        end for

        if $streamQuality != -1$ then

            for $i \leftarrow 0, i < cntTiles, i++$ do

                if $repoDistances.getElementAt(i) > 0$ then
                    $mpdUrl \leftarrow arrayMpd[i]$
                    $allocatedBandwidth \leftarrow dashHandler.getBandwidthForQuality$
                    $(streamQuality, repoMpds.getElementAt(mpdUrl))$
                    $bandwidthAllocations[i] \leftarrow allocatedBandwidth$
                end if
            end for
        end if
    else
        for $i \leftarrow 0, i < cntTiles, i++$ do
            $bandwidthAllocations[i] \leftarrow 0$
        end for
    end if
end function
```

Appendix K

Viewport at highest quality with lowest quality peripheral tiles distribution algorithm

```
function CALCULATEBANDWIDTH
    $tempBandwidth \leftarrow availableBandwidth$

    for $i \leftarrow 0, i < cntTiles, i++$ do
        $mpdUrl \leftarrow arrayMpd[i]$
        $bandwidth \leftarrow dashHandler.getBandwidthForQuality(0, repoMpds.getElementAt(mpdUrl))$
        $bandwidthAllocations[i] \leftarrow bandwidth$
        $tempBandwidth \leftarrow tempBandwidth - bandwidth$
    end for

    if $tempBandwidth > 0$ then
        $mpdUrl \leftarrow arrayMpd[0]$
        $amountQualities \leftarrow dashHandler.getAmountOfQualitiesInMpd$
        $(repoMpds.getElementAt(firstMpdUrl))$
        for
            $startQuality = 1, startQuality < amountQualities, startQuality++$ do
            for
                $i = 0, i < cntTiles, i++$ do

                if $repoDistances[i] == 0$ then
                    $tempUrl \leftarrow arrayMpd[i]$
                    $neededBandwidth \leftarrow dashHandler.getBandwidthForQuality$
                    $(startQuality, repoMpds.getElementAt(tempUrl))$

                    if $tempBandwidth + bandwidthAllocations[i] - neededBandwidth > 0$ then
                        $tempBandwidth \leftarrow tempBandwidth + bandwidthAllocations[i] - neededBandwidth$
                        $bandwidthAllocations[i] \leftarrow neededBandwidth$
                    end if
                end if
            end for
        end for

    else
        for
            $i \leftarrow 0, i < cntTiles, i++$ do
            $bandwidthAllocations[i] \leftarrow 0$
        end for
    end if
end function
```

Appendix L

Upgrade layer per layer distribution algorithm

```
function CALCULATEBANDWIDTH
    $tempBandwidth \leftarrow availableBandwidth$

    for $i \leftarrow 0, i < cntTiles, i++$ do
        $mpdUrl \leftarrow arrayMpd[i]$
        $bandwidth \leftarrow dashHandler.getBandwidthForQuality(0, repoMpds.getElementAt(mpdUrl))$
        $bandwidthAllocations[i] \leftarrow bandwidth$
        $tempBandwidth \leftarrow tempBandwidth - bandwidth$
    end for

    if $tempBandwidth > 0$ then
        $tileLayer \leftarrow 0$
        $tilesHandled \leftarrow 0$
        $dictionHandled \leftarrow newDictionary()$
        $mpdUrl \leftarrow arrayMpd[0]$
        $amountQualities \leftarrow dashHandler.getAmountOfQualitiesInMpd$
        $(repoMpds.getElementAt(mpdUrl))$

        while $tilesHandled < cntTiles$ do
            $tilesHandled \leftarrow 0$
            for $startQuality \leftarrow 1, startQuality < amountQualities, startQuality++$ do

                for $i \leftarrow 0, i < cntTiles, i++$ do

                    if $repoDistances.getElementAt(i) == tileLayer$ then
                        $tempUrl \leftarrow arrayMpd[i]$
                        $neededBandwidth \leftarrow dashHandler.getBandwidthForQuality$
                        $(startQuality, repoMpds.getElementAt(tempUrl))$
                        $dictionHandled[i] \leftarrow true$

                        $tmpbndwdth \leftarrow tempBandwidth$
                        if $tmpbndwdth + bandwidthAllocations[i] - neededBandwidth > 0$ then
                            $tempBandwidth \leftarrow tempBandwidth + bandwidthAllocations[i]$
                            $bandwidthAllocations[i] \leftarrow neededBandwidth$
                            $tempBandwidth \leftarrow tempBandwidth - neededBandwidth$
                        end if
                    end if
                end for
            end for

            $tileLayer \leftarrow tileLayer + 1$

            for $i \leftarrow 0, i < length(dictionHandled), i++$ do
                if $dictionHandled[i]$ then
                    $tilesHandled \leftarrow tilesHandled + 1$
                end if
            end for
        end while

    else
        for $i \leftarrow 0, i < cntTiles, i++$ do
            $bandwidthAllocations[i] \leftarrow 0$
        end for
    end if
end function
```

Appendix M

Delta distribution logic

```
function CALCULATEBANDWIDTH
    $tempBandwidth \leftarrow availableBandwidth$
    $mpdUrl \leftarrow arrayMpd[0]$
    $amountQualities \leftarrow dashHandler.getAmountOfQualitiesInMpd(repoMpds.getElementAt(mpdUrl))$
    $highestLayer \leftarrow 0$
    $repoQualities \leftarrow newRepository()$
    $repoBandwidths \leftarrow newRepository()$

    for $i \leftarrow 0, i < cntTiles, i++$ do
        $repoQualities.insertValueWithKey(i, highestQuality)$
        $repoBandwidths.insertValueWithKEy(i, 0)$
        if $repoDistances.getElementAt(i) > highestLayer$ then
            $highestLayer = repoDistances.getElementAt(key)$
        end if
    end for

    $recursiveCalculateBandwidth(tempBandwidth, amountQualities - 1, 0, highestLayer,$
    $repoQualities, repoBandwidths$

    for $i \leftarrow 0, i < cntTiles, i++$ do
        $bandwidthAllocations[i] \leftarrow repoBandwidths.getElementAt(i)$
    end for
end function

function RECURSIVECALCULATEBANDWIDTH(bandwidthRest, quality, currentTileLayer, highestTile-
Layer, repoBandwidths, repoQualities)
    $bandwidthUsed \leftarrow 0$
    $mpdUrl \leftarrow arrayMpd[0]$
    $amountQualities \leftarrow dashHandler.getAmountOfQualitiesInMpd$
    $(repoMpds.getElementAt(firstMpdUrl)) - 1$

    for $i \leftarrow 0, i < cntTiles, i++$ do
        if $repoDistances.getElementAt(i) == currentTileLayer$ then
            $mpdUrl \leftarrow arrayMpd[i]$
            $bandwidth \leftarrow dashHandler.getBandwidthForQuality(quality,$
            $repoMpds.getElementAt(mpdUrl))$

            $bandwidthUsed \leftarrow bandwidthUsed + bandwidth$
            $bandwidthRest \leftarrow bandwidthRest - bandwidth$
            $repoBandwidths.changeValueForKey(i, bandwidth)$
            $repoQualities.changeValueForKey(i, quality)$
        end if
    end for

    if $bandwidthRest >= 0$ then
        if $currentLayer! = 0$ then
            $previousTileQuality \leftarrow 0$
            $previousTileLayer \leftarrow currentTileLayer - 1$

            for $i \leftarrow 0, i < cntTiles, i++$ do
                if $repoDistances.getElementAt(i) == previousTileLayer$ then
                    $previousTileQuality \leftarrow repoQualities.getElementAt(i)$
                    $break$
                end if
            end for

            if $|(previousTileQuality - quality)| <= delta$ then
                if $currentTileLayer! = highestTileLayer$ then
                    $recursiveCalculateBandwidth(bandwidthRest, amountQualities,$
                    $currentTileLayer + 1, highestTileLayer, repoQualities, repoBandwidths)$
                end if
            else
                if $previousTileQuality > 0$ then
                    $previousLayerBandwidth \leftarrow 0$

                    for $i \leftarrow 0, i < cntTiles, i++$ do
                        if $repoDistances.getElementAt(i) == previousTileLayer$ then
                            $previousLayerBandwidth \leftarrow previousLayerBandwidth+$
                            $repoBandwidths.getElementAt(i)$
                        end if
                    end for

                    $recursiveCalculateBandwidth(bandwidthRest + bandwidthUsed+$
                    $previousLayerBandwidth, previousTileQuality - 1, previousTileLayer,$
                    $highestTileLayer, repoQualities, repoBandwidths)$
                else
                    $recursiveCalculateBandwidth(bandwidthRest + bandwidthUsed,$
                    $quality-1, currentTileLayer, highestTileLayer, repoQualities, repoBandwidths)$
                end if
            end if
        else
            $recursiveCalculateBandwidth(bandwidthRest, amountQualities,$
            $currentTileLayer + 1, highestTileLayer, repoQualities, repoBandwidths)$
        end if
    else
        if $quality > 0$ then
            $recursiveCalculateBandwidth(bandwidthRest + bandwidthUsed, quality - 1,$
            $currentTileLayer, highestTileLayer, repoQualities, repoBandwidths)$
        else
            $previousTileQuality \leftarrow 0$
            $previousTileLayer \leftarrow currentLayer - 1$
            $previousLayerBandwidth \leftarrow 0$

            for $i \leftarrow 0, i < cntTiles, i++$ do
                if $repoDistances.getElementAt(i) == previousTileLayer$ then
                    $previousTileQuality \leftarrow repoQualities.getElementAt(i)$
                    $previousLayerBandwidth \leftarrow previousLayerBandwidth+$
                    $repoBandwidths.getElementAt(i)$
                end if
            end for

            if $previousTileQuality > 0$ then
                $recursiveCalculateBandwidth(bandwidthRest + bandwidthUsed+$
                $previousLayerBandwidth, previousTileQuality - 1, previousTileLayer,$
                $highestTileLayer, repoQualities, repoBandwidths)$
            else
                $tempLayer \leftarrow previousTileLayer$
                $tempBandwidth \leftarrow 0$

                while $tempLayer > -1$ do
                    $tempQuality \leftarrow 0$

                    for $i \leftarrow 0, i < cntTiles, i++$ do
                        if $repoDistances.getElementAt(i) == (tempLayer - 1)$ then
                            $tempQuality \leftarrow repoQualities.getElementAt(i)$
                            $tempBandwidth \leftarrow tempBandwidth+repoBandwidths.getElementAt(i)$
                        end if
                    end for

                    if $tempQuality > 0$ then
                        $tempTotalBandwidth \leftarrow bandwidthRest + bandwidthUSed+$
                        $previousLayerBandwidth + tempBandwidth$

                        $recursiveCalculateBandwidth(tempTotalBandwidth, tempQuality-1,$
                        $tempLayer, highestTileLayer, repoQualities, repoBandwidths)$
                    end if

                    $tempLayer \leftarrow tempLayer - 1$
                end while

                if $tempLayer > 0$ then
                    for $i \leftarrow 0, i < cntTiles, i++$ do
```

```
                        $repoBandwidths.changeValueForKey(i, 0)$
                        $bandwidthAllocations[i] \leftarrow 0$
                        $repoQualities.changeValueForKey(i, -1)$
                    end for
                end if
            end if
        end if
    end if
end function
```

Appendix N

Dutch summary

Video's bekijken over het internet wordt steeds meer populairder. Veel websites zoals YouTube, Twitch en Netflix bieden on-demand en live video streaming functionaliteiten aan. De QoE (Quality of Experience) van de gebruiker en de beschikbare bandbreedte spelen een belangrijke rol bij het bekijken van video's. De QoE meet de waargenomen kwaliteit van de geleverde service. In deze masterproef is de service het aanbieden van video diensten. De bandbreedte is de hoeveelheid data dat gedownload kan worden, gemeten in bits of bytes per seconde. In het algemeen is het bekend dat de QoE van gebruiker beïnvloed wordt door 4 variabelen:

1 **Start/wacht tijd van de video:** Dit is de tijd dat de gebruiker moet wachten alvorens de video start met afspelen.

2 **Aantal keren dat de videospeler stopt met spelen:** Dit is het aantal keren dat de videospeler stopt met spelen tijdens het afspelen van de video.

3 **Aantal kwaliteitsveranderingen:** Dit is het aantal keren dat de kwaliteit veranderd wordt tijdens het afspelen.

4 **Media throughput:** Dit is de effectieve snelheid waaraan de video gedownload wordt door de client.

De start/wacht tijd van de video kan de QoE van de gebruiker beïnvloeden wanneer de gebruiker langer of minder lang moet wachten. Wanneer de gebruiker langer moet wachten om de video te starten, is de QoE lager dan wanneer de gebruiker minder lang moet wachten.

Als de videospeler tijdens het afspelen van de video meerdere keren stopt met spelen wordt de gebruiker gefrustreerd door dit gedrag. De gebruiker moet zich telkens aanpassen aan de nieuwe kwaliteit.

Wanneer de media throughput lager is, kan er minder data gedownload worden door de client. De throughput geef weer, in bits of bytes, hoeveel data er effectief gedownload wordt door de client. Dit kan ervoor zorgen dat de kwaliteit van de video lager is omdat hogere kwaliteit van de video meer bandbreedte vergt dan lagere kwaliteit en zorgt er ook voor dat de QoE van de gebruikers lager is wanneer de video in lage kwaliteit getoond wordt.

Dynamische netwerken zorgen ervoor dat de beschikbare bandbreedte per client frequent kan veranderen. Dynamische netwerken zijn netwerken waar computers constant het netwerk kunnen verlaten of toe kunnen treden tot het netwerk. Bijvoorbeeld bij een toenemend aantal computers kan de beschikbare bandbreedte verminderen per computer omdat de bandbreedte evenredig verdeeld moet worden.

Door gebruik te maken van MPEG-DASH kan de performantie van geleverde video diensten verbeterd worden. De afkorting MPEG-DASH staat voor Moving Picture Experts Group Dynamic Adaptively Streaming over HTTP.

MPEG-DASH is ontwikkeld door MPEG en zorgt ervoor dat tijdens het afspelen van de video de kwaliteit veranderd kan worden. MPEG-DASH werkt via meerdere representaties van de video, namelijk meerdere kwaliteiten van de video. Voor dat de video data gestuurd wordt naar een computer die er interesse voor heeft, wordt de video gecodeerd in verschillende kwaliteiten. Elke kwaliteit wordt dan opgesplitst in aparte delen die segmenten genoemd worden. Elk segment heeft een vaste duur en behoort tot 1 representatie van de video. Door de verschillende kwaliteiten in segmenten in te delen, kan de videospeler tijdens het afspelen van de video veranderen van kwaliteit door de volgende segmenten van een andere representatie aan te vragen.

In het eerste deel van deze masterproef wordt eerst onderzoek gedaan naar video multistreaming. Video multistreaming betekent dat meerdere video's parallel afgespeeld worden door 1 client. Een voorbeeld hiervan is dat de client meerdere tabbladen in de web browser met video's kan afspelen. Het is de bedoeling de bandbreedte te verdelen over de video's die afspelen. Omdat niet alle video's gelijktijdig getoond kunnen worden wanneer meerdere tabbladen gebruikt worden, bieden we twee bandbreedte distributie logica's aan voor het verdelen van de bandbreedte over deze video's.

De eerste bandbreedte distributie logica voor video multistreaming is de distributie logica die de bandbreedte eerlijk verdeeld over alle video's. Met andere woorden, elke video krijg evenveel bandbreedte toegekend. In zulke situaties valt het voor dat de video's niet dezelfde uniforme kwaliteit tonen omdat video's meer bandbreedte kunnen vereisen om hoge kwaliteit ervan te tonen. Wanneer de beschikbare bandbreedte verandert, wordt ook de toegekende bandbreedte per videospeler veranderd.

Voor video multistreaming hebben we een buffering/steady scheduler gemaakt. Deze buffer kan in 2 fases werken, namelijk in de buffering fase en in de steady fase. De videospeler start eerst met de buffering fase en schakelt erna over naar de steady fase. De buffering fase zorgt ervoor dat er zo snel mogelijk video getoond kan worden door zo snel mogelijk lagere kwaliteit segmenten van de video te downloaden. Omdat lagere kwaliteit segmenten minder data bevatten kunnen deze sneller gedownload worden dan hogere kwaliteit segmenten. Wanneer er genoeg gebufferd is om de video af te spelen, schakelt de videospeler over naar steady fase. In deze fase wordt er om een bepaald interval 1 video segment gedownload. Dit is om ervoor te zorgen dat het bufferniveau stabiel blijft.

Naast het eerlijk verdelen van de beschikbare bandbreedte over de video's hebben we een tweede distributie logica gemaakt. Namelijk de prioriteit-gebaseerde bandbreedte distributie logica. Deze distributie logica gaat de beschikbare bandbreedte verdelen aan de hand van percentages. Elke videospeler wordt een percentage toegekend. De som van alle percentages mogen maximum 100 zijn. Het idee hierachter is dat actieve video's een hoger percentage krijgen dan niet actieve video's. Bijvoorbeeld in het geval dat er meerdere tabbladen zijn waarin video's afgespeeld worden. De video's in de gesloten tabbladen worden niet meteen bekeken, waardoor we het percentage voor deze video's kunnen verlagen en waarmee ze lagere kwaliteit van de video tonen. De video's in de actieve tabbladen krijgen een hoger percentage omdat hier rechtstreeks naar gekeken wordt. Door deze aanpak kunnen we prioriteiten stellen voor video's en kunnen bepaalde video's in hogere kwaliteit getoond worden dan andere video's.

Het tweede deel van deze masterproef gaat over ODV tiled video streaming met MPEG-DASH. ODV staat voor omni-direcitonal video en is 360 graden video waarmee de gebruiker maar een bepaald deel van het volledige 360 graden beeld te zien krijgt. Dit is de bedoeling omdat mensen in realiteit ook maar een bepaalde hoek voor zich kunnen zien, het viewport genoemd in ODV, en niet volledig 360 graden rond zich kunnen kijken. ODV tiled video is 360 graden video waarmee het volledige 360 graden beeld opgedeeld is in tegels van een vaste hoogte en breedte. De bedoeling hiervan is dat tegels die in het viewport liggen aan hoge kwaliteit getoond worden. De tegels rondom het viewport worden aan lagere kwaliteit getoond omdat de gebruiker niet rechtstreeks interageert met deze tegels. In de volgende figuur zien we een 360 graden beeld opgedeeld in tegels en het viewport als vierkant aangeduid.

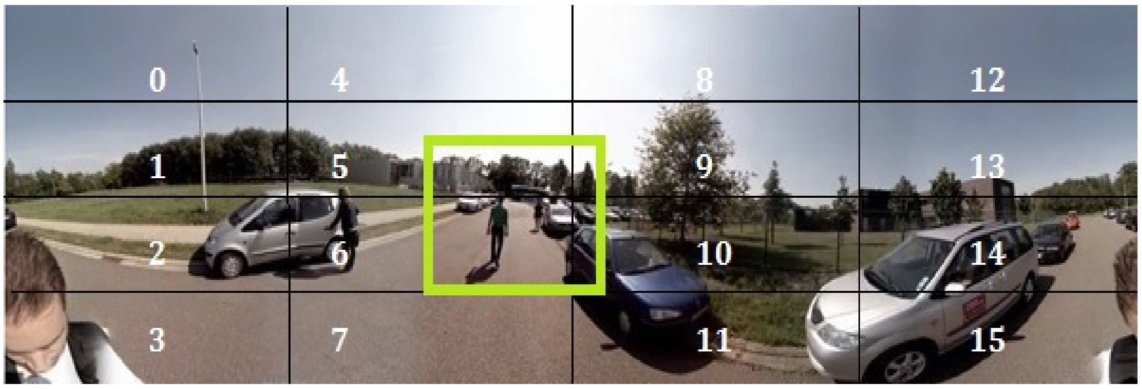


Figure N.1: Full ODV frame tiled into a 4x4 grid with an inidication of the current viewport

Om de beschikbare bandbreedte te verdelen over de tegels van het 360 graden beeld, hebben we 6 distributie logica's gemaakt. Elke distributie logica heeft een eigen focus en verdeelt de beschikbare bandbreedte naargelang die focus over alle tegels van het 360 graden beeld.

De eerste bandbreedte distributie logica die we gemaakt hebben voor ODV tiled video is de simpele distributie logica. De simpele distributie logica zorgt ervoor dat alle tegels van het 360 graden beeld dezelfde uniforme kwaliteit van de video tonen die mogelijk is met de beschikbare bandbreedte. Dus bij deze distributie logica is er geen verschil van kwaliteit tussen de tegels.

Een tweede distributie logica die we gemaakt hebben is de distributie logica die alleen de tegels van het viewport bandbreedte toekent. Dit zorgt ervoor dat alle andere tegels geen bandbreedte krijgen en ook niet afgespeeld worden. Dus de andere tegels rondom het viewport blijven zwart tijdens het afspelen van het viewport. Wanneer het viewport naar deze tegels veranderd wordt, krijgen deze tegels bandbreedte toegekend en starten deze met spelen. De kwaliteit wordt ook zoals de vorige aanpak bepaald aan de hand van de beschikbare bandbreedte en er wordt ook een uniforme kwaliteit voor de tegels in het viewport gekozen.

De volgende aanpakken werken met lagen van tegels. De tegels die in het viewport liggen zijn gemarkeerd als laag 0, de aangrenzende tegels van het viewport als laag 1, de tegels die daaraan grenzen aan laag 2, enzovoort. De derde distributie logica gaat eerst alle tegels van het 360 graden videobeeld bandbreedte toekennen om de laagste kwaliteit toe te kennen. Wanneer er bandbreedte over is worden alle tegels in het viewport per kwaliteit behandeld. Dus als er bandbreedte over is, wordt er gekeken of de kwaliteit van de tegels in het viewport verbeterd kan worden. Tegel per tegel wordt behandeld en er wordt telkens gekeken of er genoeg bandbreedte over is om de kwaliteit te verbeteren totdat de hoogste kwaliteit voor alle tegels in het viewport behaald is. Wanneer de hoogste kwaliteit behaald is en er is bandbreedte over, worden alle andere tegels rondom het viewport op dezelfde manier behandeld.

De vierde distributie logica is gebaseerd op de vorige distributie logica en werkt gelijkaardig. In deze distributie logica wordt ook eerst bandbreedte toegekend voor de laagste kwaliteit aan alle tegels. Erna worden de tegels van het viewport ook tegel per tegel per kwaliteit behandeld maar de andere tegels rondom het viewport worden niet meer behandeld. Dus deze tegels zullen altijd de laagste kwaliteit van de video tonen in vergelijking met de vorige aanpak waarmee de tegels rondom het viewport tegel per tegel per kwaliteit behandeld worden.

De volgende distributie logica gaat laag per laag behandelen. Deze distributie logica zorgt ervoor dat de kwaliteit per laag uniform is. Deze start ook met bandbreedte voor alle tegels toe te kennen. Als er bandbreedte over is, wordt het viewport behandeld. Er wordt de hoogst mogelijke uniforme kwaliteit voor het viewpor gekozen en als er bandbreedte over is voor de volgende laag, enzovoort. Dus laag per laag wordt behandeld en voor elke laag wordt de hoogst mogelijke uniforme kwaliteit gekozen.

De laatste distributie logica is de meest complexe distributie logica. Deze werkt ook met lagen en uniforme kwaliteiten per laag, maar zorgt ervoor dat het kwaliteitsverschil tussen aangrenzende lagen maximum gelijk is aan de delta waarde. De delta waarde wordt alvorens bepaald. Het algoritme start eerst me de hoogst mogelijke uniforme kwaliteit toe te kennen aan de tegels in het viewport. Daarna wordt er met de overblijvende bandbreedte, de hoogst mogelijke kwaliteit voor de volgende laag geselecteerd en gekeken of het kwaliteitsverschil maximum gelijk is aan delta. Als dit het geval is wordt de volgende laag behandeld totdat alle lage behandeld zijn. Als dit niet het geval is wordt de kwaliteit van het viewport verminderd. Er wordt dan weer gekeken of de kwaliteit van de volgende laag, geselecteerd op basis van de overblijvende bandbreedte, maximum delta waarde verschilt met de vorige laag. Er wordt telkens naar de vorige laag gekeken om het verschil tussen kwaliteiten te berekenen. Een voorbeeld hiervan is wanneer delta=1, er 3 lagen zijn in het 360 graden beeld en er 3 kwaliteiten zijn van de video. Wanneer de tegels in het viewport (laag 0) aan kwaliteit 3 (hoogste) getoond wordt, moeten de aangrezende tegels (laag 1) minimum kwaliteit 2 tonen omdat het verschil met de vorige laag maar 1 (delta) mag zijn. De andere tegels (laag 3) moeten minimum kwaliteit 1 tonen omdat het verschil met laag 2 maximum 1 (delta) mag zijn.

Om deze distributie logica's subjectief en kwalitatief te evalueren, hebben we user testing sessies georganiseerd met 14 deelnemers. Voor deze sessies hebben we 4 distributie logica's geselecteerd van de 6, namelijk de simpele distributie logica, de distributie logica waarmee het viewport alleen getoond wordt, de distributie logica waarmee het viewport in de hoogst mogelijke kwaliteit tegel per tegel per kwaliteit getoond wordt en dit ook voor alle andere tegels en de delta distributie logica. De beschikbare bandbreedte hebben we laten variëren tussen 1.85 Mbps en 6.2 Mbps. Deze hebben we zo gekozen zodat de verschillende distributie logica's verschillend konden reageren in elke situatie. Voor de segment duur hebben we ook 2 waarden gekozen, namelijk segmenten die 1 seconde en 2 seconden lang duren. De sessies bestonden uit 14 testen waarmee elke deelnemer na een test een vragenlijst kreeg. Alle deelnemers konden voor het starten van de testen een demo versie van het framework proberen. De vragenlijst vroeg de deelnemer een score van 1 tot 5 te geven voor 4 variabelen. De eerste variabele was de algemene kwaliteit die de deelnemer een score van 1 (slecht) tot 5 (heel goed) kon geven. De tweede vraag vroeg de deelnemers of ze een verschil in kwaliteiten zagen in de video. Als dit het geval was, moesten ze 3 extra vragen invullen en anders mochten deze overgeslagen worden. De 3 extra vragen bestonden uit een score te geven voor het kwaliteitsverschil (1=heel verschillend, 5=niet verschilend), hoe storend het kwaliteitsverschil was (1=heel storend, 5=niet storend) en de laatste vraag vroeg de deelnemers hoe aanvaardbaar ze de duur vonden van de kwaliteitsverschil (1=niet aanvaardbaar, 5=aanvaardbaar). Omdat het even duurde voordat de kwaliteit veranderde tijdens het draaien van het viewport werd de laatste vraag gesteld.

Enkele resultaten van de user testing waren opmerkelijk. De simpele distributie logica had de hoogste kwaliteitsverschil score voor 1 seconde segmenten. Dit is normaal omdat de kwaliteit voor de simpele distributie logica niet verandert. Echter voor 2 seconde segmenten had de simpele distributie logica de tweede hoogste score voor het kwaliteitsverschil. We denken dat dit te maken heeft met enkele artefacten die verschenen tijdens de testen van de simpele distributie logica. De video's bij de simpele distributie logica toonden namelijk enkele blurry effecten in enkele tegels terwijl de kwaliteit hetzelfde was dan andere tegels. Dit heeft te maken met de beweging in de video's. Als er veel beweging is in de video is de kwaliteit opmerkelijk slechter dan wanneer er meer beweging is in de video, ook al zijn de video's aan dezelfde bitrate gecodeerd. De kwaliteit was het beste bij de distributie logica waar het viewport tegel per tegel per kwaliteit behandeld werd en de andere tegels op gelijkaardige manier. Deze distributie logica scoorde in het algemeen ook het beste. Hieruit leerden we dat de score hoger was wanneer de overgang tussen kwaliteiten niet duidelijk is. De distributie logica waar alleen het viewport getoond werd scoorde in het algemeen het laagste omdat er zwarte tegels getoond werden wanneer het viewport gedraaid werd. Voor de testen hebben we de de simpele distributie logica met 1 en 2 seconden er niet uitgelaten. Dit hadden we beter wel gedaan omdat de duur van de segmenten toch geen rol speelde bij deze distributie logica omdat de kwaliteit niet veranderd werd. Het was beter de distributie logica die alleen de tegels in het viewport bandbreedte toekende met hoge bandbreedte en 1 en 2 seconden segment duur in de testen te zetten. Deze hadden we er namelijk uit gelaten en we zagen deze fout pas nadat de testen uitgevoerd werden.

Bibliography

[1] MPEG, Welcome to MPEG. http://mpeg.chiariglione.org/, 1988. Accessed 9 June 2015.

[2] Cacti, The complete rddtool-based graphing solution. http://www.cacti.net/, 2001. Accessed 29 May 2015.

[3] GPAC, Multimedia open source project. http://gpac.wp.mines-telecom.fr/home/, 2003. Accessed 29 May 2015.

[4] MP4Box, DASH Support in MP4Box. http://gpac.wp.mines-telecom.fr/mp4box/dash/, 2003. Accessed 29 May 2015.

[5] MP4Box, Multimedia packager. http://gpac.wp.mines-telecom.fr/mp4box/, 2003. Accessed 29 May 2015.

[6] MPEG, MPEG-7. http://mpeg.chiariglione.org/standards/mpeg-7, 2004. Accessed 29 May 2015.

[7] Videolan, x264. http://www.videolan.org/developers/x264.html, 2004. Accessed 29 May 2015.

[8] Blender, Big Buck Bunny. https://peach.blender.org/, 2008. Accessed 29 May 2015.

[9] Digitalpreservation, MPEG-4, Advanced Video Coding (Part 10) (H.264). http://www.digitalpreservation.gov/formats/fdd/fdd000081.shtml, 2011. Accessed 16 June 2015.

[10] Eltrovemo, MPEG-DASH timeline. http://blog.eltrovemo.com/dash_timeline/, 2011. Accessed 29 May 2015.

[11] TMCnet, Skype group calling launches. http://blog.tmcnet.com/blog/tom-keating/skype/skype-group-calling---meh.asp, 2011. Accessed 29 May 2015.

[12] ISO/IEC 23009-1, MPEG-DASH international standard, information technology - Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57623, 2012. Accessed 29 May 2015.

[13] FFMPEG, A complete, cross-platform solution to record, convert and stream audio and video. https://www.ffmpeg.org/, 2013. Accessed 1 June 2015.

[14] Octoshape Support, Encoding best practices using FFMPEG. https://support.octoshape.com/entries/25126002-Encoding-best-practices-using-ffmpeg, 2013. Accessed 30 May 2015.

[15] Stick it! (pop-up player). https://play.google.com/store/apps/details?id=com.myboyfriendisageek.stickit&hl=nl, 2013. Accessed 29 May 2015.

[16] Unified Streaming now offers support for H.265/HEVC delivery over MPEG-DASH. http://www.unified-streaming.com/company/news/streaming-h265-hevc-over-dash/, 2013. Accessed 26 June 2015.

[17] Analysis of Variance (ANOVA) for comparing the means. http://yatani.jp/teaching/doku.php?id=hcistats:anova, 2014. Accessed 1 June 2015.

[18] ISO/IEC 23009-1:2014/FPDAM 2, information technology, dynamic adaptive streaming over HTTP (DASH), part 1: Media presentation description and segment formats / amd 2. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=65274, 2014. Accessed 23 June 2015.

[19] Yourdictionary, dynamic rate adaptation. http://www.yourdictionary.com/dynamic-rate-adaptation, 2014. Accessed 23 June 2015.

[20] Alexzambelli, smooth streaming architecture. http://alexzambelli.com/blog/2009/02/10/smooth-streaming-architecture/, 2015. Accessed 1 June 2015.

[21] Apache, apache tomcat 8. http://tomcat.apache.org/tomcat-8.0-doc/, 2015. Accessed 30 May 2015.

[22] Apache, HTTP server project. http://httpd.apache.org/, 2015. Accessed 1 June 2015.

[23] Bitmovin Gmbh, Ultra-high-definition-quality of experience with MPEG-DASH. http://www.bitmovin.net/blog/2015/05/ultra-high-definition-quality-experience-mpeg-dash-part-1/, 2015. Accessed 29 May 2015.

[24] Dell. http://www.dell.com/, 2015. Accessed 30 May 2015.

[25] FON, The worldwide wifi network. https://corp.fon.com/nl, 2015. Accessed 21 June 2015.

[26] Google, Google Forms. https://www.google.com/forms/about/, 2015. Accessed 30 May 2015.

[27] GoPro company website. http://shop.gopro.com/EMEA/, 2015. Accessed 24 June 2015.

[28] HEVC / H.265 explained. http://x265.org/hevc-h265/, 2015. Accessed 25 June 2015.

[29] Hewlett-Packard. http://www8.hp.com/be/nl/home.html, 2015. Accessed 26 June 2015.

[30] JCT-VC - Joint Collaborative Team on Video Coding. http://www.itu.int/en/ITU-T/studygroups/2013-2016/16/Pages/video/jctvc.aspx, 2015. Accessed 25 June 2015.

[31] Levene's test. http://svitsrv25.epfl.ch/R-doc/library/car/html/levene.test.html, 2015. Accessed 1 June 2015.

[32] MDN, XMLHttpRequest. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest, 2015. Accessed 30 May 2015.

[33] Microsoft, Silverlight. https://www.microsoft.com/silverlight/, 2015. Accessed 1 June 2015.

[34] Microsoft, Smooth streaming player. https://www.microsoft.com/silverlight/smoothstreaming/, 2015. Accessed 1 June 2015.

[35] Netflix, netflix belgium. https://www.netflix.com/be/, 2015. Accessed 1 June 2015.

[36] NGINX, What is progressive download? http://nginx.com/resources/glossary/progressive-download/, 2015. Accessed 9 June 2015.

[37] Oculus Rift. https://www.oculus.com/en-us/, 2015. Accessed 23 June 2015.

[38] R, tukeyhsd test. https://stat.ethz.ch/R-manual/R-devel/library/stats/html/TukeyHSD.html, 2015. Accessed 1 June 2015.

[39] Random.org, true random number service. https://www.random.org/, 2015. Accessed 30 May 2015.

[40] Skype, Videoconferencing application. http://www.skype.com/, 2015. Accessed 19 June 2015.

[41] Sourcebuffer object. http://www.w3.org/TR/media-source/#sourcebuffer, 2015. Accessed 19 June 2015.

[42] Unique media TV, Streaming vs. progressive download. https://www.unique-media.tv/support/28/Introduction/Streaming_vs_Progressive_Download, 2015. Accessed 1 June 2015.

[43] VirtualBox. https://www.virtualbox.org/, 2015. Accessed 26 June 2015.

[44] Webopedia, content delivery network. http://www.webopedia.com/TERM/C/CDN.html, 2015. Accessed 21 June 2015.

[45] Wikipdia, Codec. http://en.wikipedia.org/wiki/Codec, 2015. Accessed 29 May 2015.

[46] Wikipedia, granularity. https://en.wikipedia.org/wiki/Granularity, 2015. Accessed 23 June 2015.

[47] Wikipedia, latin square. http://en.wikipedia.org/wiki/Latin_square, (2015). Accessed 1 June 2015.

[48] Wikipedia, Multipoint control unit. https://en.wikipedia.org/wiki/Multipoint_control_unit, 2015. Accessed 25 June 2015.

[49] Wikipedia, openflow. https://en.wikipedia.org/wiki/OpenFlow, 2015. Accessed 23 June 2015.

[50] Wikipedia, Real Time Streaming Protocol. https://en.wikipedia.org/wiki/Real_Time_Streaming_Protocol, 2015. Accessed 19 June 2015.

[51] Wikipedia, Real-time Transport Control Protocol. https://nl.wikipedia.org/wiki/Real-time_Transport_Control_Protocol, 2015. Accessed 19 June 2015.

[52] Wikipedia, Resource Reservation Protocol. https://en.wikipedia.org/wiki/Resource_Reservation_Protocol, 2015. Accessed 19 June 2015.

[53] Wikipedia, Structural similarity. `https://en.wikipedia.org/wiki/Structural_similarity`, 2015. Accessed 22 June 2015.

[54] S. Akhshabi, A. C. Begen, and C. Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the Second Annual ACM SIGMM Conference on Multimedia Systems, MMSys 2011*, pages 157–168, Santa Clara, CA, USA, February 2011.

[55] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986, Uniform Resource Identifier (URI): Generic syntax. `https://tools.ietf.org/html/rfc3986`, 2005. Accessed 30 May 2015.

[56] R. Braden, L. Zhan, S. Berson, S. Heerzog, and S. Jamin. RFC 2205, Resource ReSerVation Protocol (RSVP). `https://tools.ietf.org/html/rfc2205`, 1997. Accessed 9 June 2015.

[57] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. RFC 3376, Internet Group Management Protocol, version 3, 2002. Accessed 29 May 2015.

[58] F. Denoual, F. Maze, M. Hirabayashi, C. Concolato, and J. L. Feuvre. Input Text for SRD in MPEG-DASH Part3 Amd.1. `http://lefeuvre.wp.mines-telecom.fr/publications/`, 2014. Accessed 25 June 2015.

[59] J. Devloo, N. Lamot, J. Van Campen, E. Weymaere, S. Latré, J. Famaey, R. Van Brandenburg, and F. De Turck. Design and evaluation of tile selection algorithms for tiled HTTP adaptive streaming. In *Lecture notes in computer science*, volume 7943, pages 25–36, 2013.

[60] K. Egevang and P. Francis. RFC 1631, The IP Network Address Translator (NAT). `https://www.ietf.org/rfc/rfc1631.txt`, 1994. accessed 29 May 2015.

[61] C. Feldmann, C. Bulla, and B. Cellarius. Efficient stream-reassembling for video conferencing applications using tiles in HEVC. In *Proc. of International Conferences on Advances in Multimedia MMEDIA '13*, Venice, Italy, April 2013.

[62] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616, HTTP/1.1. `http://tools.ietf.org/html/rfc2616`, 1999. Accessed 9 June 2015.

[63] R. Finlayson. RFC 2588, IP multicast and firewalls. `https://www.ietf.org/rfc/rfc2588.txt`, 1999. Accessed 29 May 2015.

[64] P. Georgopoulos, Y. Elkhatib, M. Broadbent, M. Mu, and N. Race. Towards network-wide QoE fairness using OpenFlow-assisted adaptive video streaming. In *Proceedings of the 2013 ACM SIGCOMM Workshop on Future Human-centric Multimedia Networking*, FhMN '13, pages 15–20, New York, NY, USA, 2013.

[65] S. Hollenbeck, M. Rose, and L. Masinter. RFC 3470, Guidelines for the use of Extensible Markup Language (XML) within IETF protocols. `https://tools.ietf.org/html/rfc3470`, 2015. Accessed 29 May 2015.

[66] N.-F. Huang, H.-Y. Chang, Y.-W. Lin, and K.-S. Hsu. A novel bandwidth management scheme for video streaming service on public-shared network. In *Communications, 2008. ICC'08. IEEE International Conference on*, pages 1755–1759. IEEE, 2008.

[67] M. Jain and C. Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. In *In Proceedings of Passive and Active Measurements (PAM) Workshop*, pages 14–25, 2002.

[68] N. Q. M. Khiem, G. Ravindra, and W. T. Ooi. Adaptive encoding of zoomable video streams based on user access pattern. *Signal Processing: Image Communication*, 27(4):360–377, 2012.

[69] L. Kontothanassis, R. Sitaraman, J. Wein, D. Hong, R. Kleinberg, B. Mancuso, D. Shaw, and D. Stodolsky. A transport layer for live streaming in a content delivery network. *Proceedings of the IEEE*, (9):1408–1419, 2004.

[70] R. Mok, E. Chan, and R. Chang. Measuring the quality of experience of HTTP video streaming. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 485–492, May 2011.

[71] J. Postel. RFC 768, User Datagram Protocol. `https://www.ietf.org/rfc/rfc768.txt`, 1980. Accessed 9 June 2015.

[72] N. Quang Minh Khiem, G. Ravindra, A. Carlier, and W. T. Ooi. Supporting Zoomable Video Streams with Dynamic Region-of-interest Cropping. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, pages 259–270, New York, NY, USA, 2010.

[73] A. Y. Resnik. User-adaptive mobile video streaming using mpeg-dash. In *Applications of Digital Image Processing XXXV*, San Diego, California, United States, August 2013.

[74] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 3550, RTP: A transport protocol for real-time applications. `https://tools.ietf.org/html/rfc3550`, 2003. Accessed 3 June 2015.

[75] H. Schulzrinne, A. Rao, and R. Lanphier. RFC 2326, Real Time Streaming Protocol (RTSP). `https://www.ietf.org/rfc/rfc2326.txt`, 1998. Accessed 4 June 2015.

[76] I. Sodagar. ISO/IEC JTC1/SC29/WG11 W13533, MPEG-DASH: The standard for multimedia streaming over internet, 2012.

[77] T. Stockhammer. Dynamic adaptive streaming over http - Standards and design principles. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, pages 133–144, New York, NY, USA, 2011.

[78] H. Wang, V.-T. Nguyen, W. T. Ooi, and M. C. Chan. Mixing tile resolutions in tiled video: A perceptual quality assessment. In *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*, NOSSDAV '14, pages 25:25–25:30, New York, NY, USA, 2014.

[79] M. Wijnants and W. Lamotte. Managing Client Bandwidth in the Presence of Both Real-Time and non Real-Time Network Traffic. In *Proceedings of the 3rd IEEE International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2008)*, Bangalore, India, January 2008.