Design and Proof-of-Concept Implementation of Interactive Video Streaming with DASH.js

Benny Lam, Marcus Arnemo

Supervisor: Niklas Carlsson
Examiner: Nahid Shahmehri
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Benny Lam, Marcus Arnemo
Students in the 5 year Information Technology program complete a semester-long software development project during their sixth semester (third year). The project is completed in mid-sized groups, and the students implement a mobile application intended to be used in a multi-actor setting, currently a search and rescue scenario. In parallel they study several topics relevant to the technical and ethical considerations in the project. The project culminates by demonstrating a working product and a written report documenting the results of the practical development process including requirements elicitation. During the final stage of the semester, students create small groups and specialise in one topic, resulting in a bachelor thesis. The current report represents the results obtained during this specialisation work. Hence, the thesis should be viewed as part of a larger body of work required to pass the semester, including the conditions and requirements for a bachelor thesis.
Abstract

Traditionally, videos are played in a linear manner, which does not give the viewer many options to control their viewing experience. However, in contrast to regular video this paper presents the design and implementation of a proof-of-concept solution that supports interactive video streaming. The idea with interactive video streaming is that the viewer can personalize their experience by deciding which path a story should follow at certain branch points. The challenge is to give the user this experience while simultaneously keeping the seamless playback experience they usually have when streaming video in a traditional way. By doing careful prefetching of the alternative videos in conjunction with utilizing the concept of HTTP-based adaptive streaming, seamless playback of interactive branched video can be implemented with the dynamic adaptive streaming over a HTTP (DASH) framework. We compare our proof-of-concept implementation with a previous implementation from a different framework. The two players are tested against each other in a handful test aimed at exploring some of the aspects specific to interactive video. More specifically, we investigate the likelihood of experience "stalls", events that cause seamless playback to be interrupted, when changing the network conditions as well as when we change the nature of the so called "branch events" themselves. We find that there are distinguishable differences between the two implementations, specifically that DASH implementation has a more optimistic adaptation logic causing it to have more stalls and a higher playback rate in general.
Acknowledgments

This thesis would not see the light of day if it was not for our patient, helpful and supportive supervisor Niklas Carlsson, so we would sincerely like to thank him. We would also like to thank Vengatanathan Krishnamoorthi for providing us with information regarding his implementation of interactive branched video. We would also like to thank the DASH Industry Forum, Dan Sparacio, in particular, for insights on the buffering system on DASH.js.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract</strong></td>
<td>iv</td>
</tr>
<tr>
<td><strong>Acknowledgments</strong></td>
<td>v</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>vi</td>
</tr>
<tr>
<td><strong>List of Figures</strong></td>
<td>viii</td>
</tr>
<tr>
<td><strong>List of Tables</strong></td>
<td>ix</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Specification</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Research Questions</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Contributions and Thesis Outline</td>
<td>2</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Context</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Interactive Branched Video</td>
<td>3</td>
</tr>
<tr>
<td>2.3 Dynamic Adaptive Streaming over HTTP (DASH)</td>
<td>4</td>
</tr>
<tr>
<td>2.4 The DASH Manifest</td>
<td>6</td>
</tr>
<tr>
<td>3 Design</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Branching</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Buffer Management</td>
<td>8</td>
</tr>
<tr>
<td>4 Evaluation</td>
<td>10</td>
</tr>
<tr>
<td>4.1 Instruments</td>
<td>10</td>
</tr>
<tr>
<td>4.2 Testbed Setup</td>
<td>11</td>
</tr>
<tr>
<td>5 Results</td>
<td>13</td>
</tr>
<tr>
<td>5.1 Available Bandwidth</td>
<td>13</td>
</tr>
<tr>
<td>5.2 Round Trip Time</td>
<td>14</td>
</tr>
<tr>
<td>5.3 Number of Branch Options</td>
<td>15</td>
</tr>
<tr>
<td>5.4 Segment Length</td>
<td>15</td>
</tr>
<tr>
<td>5.5 Fraction of Time per Playback Quality</td>
<td>16</td>
</tr>
<tr>
<td>6 Discussion</td>
<td>19</td>
</tr>
<tr>
<td>6.1 Available Bandwidth</td>
<td>19</td>
</tr>
<tr>
<td>6.2 Round Trip Time</td>
<td>19</td>
</tr>
<tr>
<td>6.3 Number of Branch Options</td>
<td>20</td>
</tr>
<tr>
<td>6.4 Segment Length</td>
<td>20</td>
</tr>
<tr>
<td>6.5 Method</td>
<td>20</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Illustration of terminology</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Process behind dynamic adaptive streaming over HTTP</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Representation of the MPD structure [mani]</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Control bar</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>Buffer and scheduling logic of dash player [bufferandscheduling]</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Manifest generated by MP4Box with some modifications</td>
<td>11</td>
</tr>
<tr>
<td>4.2</td>
<td>Google Chrome Dev tools: Network Throttling Profiles</td>
<td>12</td>
</tr>
<tr>
<td>5.1</td>
<td>Stall probability (a) and average playback rate (b) under different available bandwidths</td>
<td>14</td>
</tr>
<tr>
<td>5.2</td>
<td>Cumulative distribution function of stall durations</td>
<td>14</td>
</tr>
<tr>
<td>5.3</td>
<td>Stall probability (a) and average playback rate (b) under different round trip times</td>
<td>15</td>
</tr>
<tr>
<td>5.4</td>
<td>Stall probability (a) and average playback rate (b) under different number of branch options</td>
<td>16</td>
</tr>
<tr>
<td>5.5</td>
<td>Stall probability (a) and average playback rate (b) under different segment lengths</td>
<td>16</td>
</tr>
<tr>
<td>5.6</td>
<td>Playback qualities during different bandwidth (a) and RTTs (b)</td>
<td>18</td>
</tr>
<tr>
<td>6.1</td>
<td>Video and audio buffer</td>
<td>21</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Prestudy on varying I-frame distance ........................................ 11
1 Introduction

1.1 Motivation

Today, more and more people are using the Internet for their daily news and entertainment. The Internet usage is rising rapidly and soon everyone will be using the Internet in some way, either for communicating, web browsing, gaming, file-sharing or entertainment. Video is dominating the data traffic of today and the mobile traffic is expected to grow ten-fold by 2021 for which video is forecasted to account for 40 percent of it [20][11]. An increasing level of personalization on the web (e.g., interest-based advertising, targeting ads, customizable Facebook chat etc.) has led to users being used to personalized services. When it comes to watching a video there has not been much you can customize as a user that is content related, other than playing, skipping and pausing the video. User generated content (UGC) websites such as YouTube and Vimeo has been a great asset for the creative people in the video community. Now that YouTube supports 360-degree interactive video, a user can control the content to be viewed and watch the playback from a desired perspective. However, what if video content were to be more interactive and users were able to control a whole plot or allowed to follow a certain character in a movie or TV-series? Imagine if we watched Snow White and the Seven Dwarfs as kids and were given the possibility to follow the dwarfs instead. Would that not be a fun and exciting experience? Interactive branched video (previously also called non-linear and multi-path video) lets the viewers not just watch the video but also play an active role in deciding how a story will unfold [15][8]. Another way that interactive branched video can be used is for multi-view video streaming, which allows the viewer to interactively switch between perspectives that are provided by different cameras [7]. This provides the user with a more customizable viewing experience.

1.2 Problem Specification

When browsing the web or watching a video, users want an instantaneous response to their actions. According to a study funded by Ericsson, a delay of 2 seconds when loading videos caused stress levels to increase by 23 percent and once a video begins, a single pause can cause stress levels to increase by an additional 15 percent [11]. A 2013 study found that time spent buffering was the number one predictor of user engagement [10]. Due to this, a good video on demand (VoD) service would want as few stalls as possible for the user perceived
1.3. Research Questions

Quality of Experience (QoE) to be maximized. By implementing media that permits users to interact with video content, resulting in multiple outcomes, it uncovers new challenges on how to provide instantaneous playback. Previously, an implementation of this type of branched media player has been done using the Open Source Media Framework (OSMF) \[17\][16]. A downside to this implementation is that OSMF uses the aging software Adobe Flash.

In this paper we present the design and implementation of a proof-of-concept interactive branched video player developed using the DASH.js framework. Compared to OSMF this framework uses an HTML5 standard, and thus, is not dependent on Adobe Flash. Our HTTP-based Adaptive Streaming (HAS) solution provides careful prefetching for multiple playback paths and improved buffer management of interactive branched video ensuring instantaneous playback of alternate videos. The use of HAS allows the quality of both the streaming video (currently being viewed) and the prefetched alternative videos (that are cached for potential future viewing) to be adapted such as to make the best use of the available bandwidth.

1.3 Research Questions

This project aims to compare a dash.js implementation of a branched video player to a previous OSMF implementation of branched video player\[17]. And to see if there are any distinguishable differences in how they behave under a variety of circumstances.

1. Is there any difference between how the two players perform when network conditions deteriorate?

2. Is there any difference between how the two players perform when playing videos with different properties?

3. Are there any other differences between the performance of the two players?

1.4 Contributions and Thesis Outline

The primary contribution that this paper makes is an implementation of an interactive branched video in a DASH.js framework, which is supported by a majority of platforms. This paper can serve as a starting point for further studies of interactive media within the DASH.js framework.

The thesis has the following structure: First, we have the Background which offers necessary knowledge regarding interactive branched video and dynamic adaptive streaming over HTTP. Second, the 3rd chapter, Design, describes our proof-of-concept implementation of interactive branched video. In the Evaluation section, instruments used for testing the performance of our player are described and evaluated. The results on the experiments are presented in chapter 6, Results. Then, in Discussion, we discuss our results compared to the results of the OSMF player \[17][16]. Finally, in the conclusion we return to our research questions and see whether or not we have answered them.
2 Background

Traditionally, videos are consumed in a linear manner, i.e., viewers press play and watch the video until they reach the end of it or until they decide to stop watching. In contrast to traditional linear video, we want to provide a more personal way of viewing video which can be done by adding the possibility to interact with the video order. In this chapter we define the context and describe the background and related work to interactive branched video and Dynamic Adaptive Streaming over HTTP (DASH).

2.1 Context

Forget about Flash Player and other plugins. MPEG-DASH can be embedded in HTML5-tags [13], which makes it far more accessible compared to sites that require plugins such as Silverlight [19]. Since DASH.js is an open source JavaScript player and easy to extend it is also more flexible when it comes to player development, in our case, allowing us to extend it into an interactive branched video media player. Flash is however still a powerful platform today but there are many devices, such as smart TVs, game consoles and connected devices (e.g., Chromecast), rising that do not support Flash. Since they all have separate player environments the content provider has to build and manage several players to reach its clients — Netflix is a classic example. It would be extremely cumbersome to require plugins to be installed on each new type of device that wants to make use of the application. This is why it is essential to replace Flash with a media player solution that can reach multiple platforms merely via the browser.

2.2 Interactive Branched Video

Interactive branched video can be used in many ways, we just have to be creative enough to find an area for which it could be applicable on. The main idea behind an interactive branched video is that users are allowed to interact with the video content and choose what path the video should follow (e.g., decide how a movie with several story lines should play out). At specific points in time when watching an interactive video it should be possible for the viewer to interact with the video [18]. These points are defined as branch points [15][17] [16]. By including branch points in a video at which there are multiple choices for which
2.3 Dynamic Adaptive Streaming over HTTP (DASH)

Almost all popular streaming sites (e.g., YouTube, Netflix, Vimeo, Twitch etc.) are using or are switching to HTTP adaptive streaming to deliver streaming content to their customers. Even though their rate-adaption algorithms may differ from each other, they are, at bottom, still using adaptive streaming over HTTP. The rate-adaption mechanism that we will continue to use in our work is also adaptive streaming over HTTP. The motivation for adaptive streaming is to reduce the delivery bandwidth cost as well as increasing the quality of experience of the viewers. The convenience with HAS is that it allows us to adapt the quality of a video to the available bandwidth. What makes this possible is that the video object is partitioned in chunks, typically a few seconds long. A video player can then request the different chunks at different encoding bitrates depending on the network conditions [1]. The need to reach multiple platforms and consumer electronics devices has resulted in these different HTTP
2.3. Dynamic Adaptive Streaming over HTTP (DASH)

Adaptive streaming technologies converging to Dynamic Adaptive Streaming over HTTP (DASH), also known as MPEG-DASH [14][24]. This standard is still in its infancy and is yet to be adopted by streaming sites. Moving Picture Expert Group (MPEG) developed it as an attempt at solving the complexities of delivering media to multiple types of devices through a unified common standard.

The promise with MPEG-DASH [13] is a single player development environment that can work across multiple devices to deliver video and audio. It is also capable of handling adaptive segmented playback of media. It also provides complete control over the bitrate switching logic, buffer logic and failure recovery. Furthermore, it provides the ability to use HTML5 and CSS for the user interface of the player.

Typically, you take a video file and feed it into a video element as a source object and it starts playing in your browser automatically, without you having any control of when it starts to buffer or which bitrates are being picked. However, with the general process of dynamic adaptive streaming it allows you to choose when you want to buffer and at which bitrates the video should be buffered in. When a user streams a video using a DASH service, an XML document — i.e., the media presentation description (MPD) for DASH, also known as a manifest [14] — is downloaded by the client via HTTP. The MPD is then parsed in order for the client to learn about the timing of the program, the availability of media content, the types of media, resolutions, minimum and maximum bandwidths and a plethora of other information. By utilizing the information provided by the manifest, the client can select appropriate encoded alternatives and start the streaming of the content by fetching the chunks of the video. [21]

The general process behind DASH is described visually in Figure 2.2. The content provider determines at what bitrates the video should be encoded in for the client to stream. The videos are then divided into small chunks, using softwares such as MP4Box, and hosted on the server. The client’s media player is then responsible for retrieving the file chunks from the server, with consideration to the client’s available bandwidth, and is able to playback the chunks as a video. [2]

![Figure 2.2: Process behind dynamic adaptive streaming over HTTP](image-url)
2.4 The DASH Manifest

The MPD is a hierarchical data model [21] which is an XML document, as previously mentioned. As seen in Figure 2.3, the manifest can consist of one or multiple periods. Multiple periods can be used for dividing the program content into scenes or chapters, and can also be used for separating ads from the program itself. Each period describes the start time and duration of the video or audio content.

There can be one or multiple adaptation sets in a period which contains a media stream or a set of media streams in various encoded alternatives. In the simplest case, a period could have one adaptation set containing both audio and video. However, for adaptive streaming it would be beneficial to split video and audio streams into different adaptation sets to enable the DASH clients to dynamically switch between different encoded streams according to their available bandwidth. So a movie with five audio translations would have six adaptation sets, one for video and five for audio. For a program with both audio and video there would be two adaptation sets.

Each adaptation set usually contains multiple representations which are encoded alternatives of the same media component, distinguished from other representations by, e.g., different resolutions, bitrates etc.

Each representation consists of media segments which are the actual stream chunks that the DASH client plays.

Each segment has a URI that the client uses to download the chunk over HTTP GET.

Figure 2.3: Representation of the MPD structure [23]
3 Design

3.1 Branching

In order to be able to achieve video branching at some arbitrary time during playback, a few things had to be achieved. First, the player has to be able to load information regarding certain branch events, when to branch, and where to branch to. Second, the player also has to execute the desired behavior given that it had been able to load information about the branch events. Third, the user has to be able to control which branch path was selected for branching. Finally, we cover the buffer prefetching done in order to achieve seamless playback when we are performing branching events.

Load Information

In order to cause as few problems as possible with the rest of the code, the implementation of the branch information load function was chosen to be as simple as possible. A text file was used, each line of the text file corresponds to one branch point and the data is formatted in a very simple manner, here is an example: (10,(30,40,50)). The number 10 indicates the time for the branch event, the numbers 30, 40 and 50, respectively, represent the time for different branch paths. The other method we considered using was the manifest used by MPEG-DASH video files and adding information about the different branch events there. However, doing so would have necessitated major changes to how the manifest was read by the player. For the sake of easy implementation we decided to not take this path for our proof-of-concept player. Further studies in the area of how to encode and decode this type of data could be of use to develop and industry standard.

When the user has selected which video to watch, one of the first actions the player takes is to load the manifest of that video using ManifestLoader.js and its load() function. At the very end of the execution of that function, we have amended it with a function call to our new function loadBranch() so that when the video is loaded so is the information about the branchpoints and branch paths. This is done in BranchController.js, a file we wrote, which contains most of the logic required to carry out branching.
3.2. Buffer Management

**Branch Execution**

BranchController.js contains most of the logic associated with branching. Every second it runs a check for an update on how long it is until the next branch event. It checks so that even if the user manually changed the point of playback or if there was a branching event taking place, the player is able to adapt to this and find the next branch event regardless of if the point of playback was changed. When this check returns that a branch event is taking place this second, the player checks in an array containing the different branch paths, and the first element is selected as the chosen branch path. This array is shifted around using user input so that the user’s choice of branch path is at the first element of the array, or if there was no user input then the element that was specified first in the text file is at the head of the array. After the player knows where to branch to it simply runs the seek(t) function to that time t.

**UI Adjustment**

We added some features to the user interface that would support branching videos. The default control bar, without any adjustments, contains the regular controls, i.e., a play and pause button, current time of the video, seek bar, duration of the video, mute button, volume bar and a full-screen button (Figure 3.1a). We added two buttons that make it possible to shift the elements of the array. The button that points up shifts the array to the left while the down-button shifts the array to the right. Besides from the two buttons we also added a feature that allows the user to perceive what branch path is chosen next and at what time (Figure 3.1b).

(a) Default UI

(b) Branched video UI

Figure 3.1: Default control bar (a) and control bar for branched video (b)

**3.2 Buffer Management**

The implementation used in order to achieve buffering of the different branch paths is built on top of the existing buffering solution for dash.js reference player 2.1.0. This solution is illustrated in Figure 3.2. When the buffer hit the branch point our implementation switched its behavior from building up further buffer beyond the branchpoint to instead start building buffer at the branch paths. The rate adaption was set so that a similar speed of buffer growth would be present. If a branch event had 4 branchpoints the rate would be adapted as if we only had a quarter of the throughput so we could achieve sufficient buffering prior to branching. The logic for this is found in NextFragmentRequestRule.js. By using this approach we are able to achieve buffering at our branch paths without building additional buffering systems for the player, all our buffering is done inside the original buffering system but its logic is changed when hitting branchpoints.
3.2. Buffer Management

Figure 3.2: Buffer and scheduling logic of dash player [5]
This chapter presents the tools we used and how we setup our environment for testing our interactive branched video player.

4.1 Instruments

Big Buck Bunny is the video we have used for testing our interactive branched video player. We chose this video because it is a widely used video used in testing purposes [4]. Another reason for choosing this particular video is because we want compare our results to the results of other papers [17] [16] with the smallest variance possible, and one way of doing so is by using the same video and start from there.

FFmpeg

FFmpeg is an open-source cross-platform tool for converting multimedia files between formats. It includes a command line tool (ffmpeg), a multimedia streaming server (ffserver), a media player (ffplay), and a multimedia stream analyzer (ffprobe) [12]. To encode the Big Buck Bunny video into different videos with different bitrates we used ffmpeg. We wanted to create the videos as close as possible to those used in the experiments with the OSMF player [17] [16]. Since the dash player does not support multiplexed data we had to separate audio and video from each other. The video will be used for streaming so it is essential to split our video into chunks. This can be done by setting the group-of-pictures (GOP) size of the encoder to the preferred chunk size of the content. In other words, we can create $X$ seconds long chunks with fixed I-frame positions in the videos. We had an I-frame placed every second so the distance between each I-frame is one second long. For the OSMF player, the chunk length was four seconds and the distance between the I-frames was also four seconds long [17]. Initially we placed our I-frames every four seconds just like the experiments performed with the OSMF player. However, we found that it gave us higher stall times than when we placed an I-frame every second. We also tried placing an I-frame every two seconds, while the branching performed better in this case than the case with a distance of four seconds, it still did not beat the performance with a distance of one second long, see Table 4.1.
To ensure that the I-frames were placed in the position we wanted them to, ffprobe was used to analyze the stream. We copied the output log and pasted it into an Excel sheet to examine where the I-frames were positioned. The encoding of our videos were at speeds of 250 kbps, 500 kbps, 850 kbps and 1300 kbps; the same bitrates used in the experiments in the OSMF player [17] [16]. The audio was encoded with 64 kbit/s and 96 kbit/s.

**MP4Box**

The dominant tool for generating a manifest for MPEG-DASH videos and splitting the videos into chunks is MP4Box [9] which is part of GPAC — a much larger suite of multimedia tools. MP4Box is a common tool used by the DASH Industry Forum and suggested by the community working on the DASH.js project on GitHub. It even has its own channel in the DashIF’s Slack channel which shows that it has a big role in preparing a video for adaptive streaming over HTTP. The video has been chunked into two seconds long chunks which gave the least stall time compared to chunks that were one, four, eight, and ten seconds long. Each chunk begins with a random access point which is needed to be able to switch between video representations between different chunks. The I-frames we placed every 24 frames are still there. MP4Box only specifies that each chunk will begin with an I-frame and does not modify the encoded video itself.

Figure 4.1 shows the manifest generated by MP4Box with some modifications. The manifest consists of a single period with two adaptation sets, one for audio and one for video. The adaptation set for video has four representations due to the four different video qualities that we are testing with.

<table>
<thead>
<tr>
<th>I-frame Distance</th>
<th>#1 (s)</th>
<th>#2 (s)</th>
<th>#3 (s)</th>
<th>#4 (s)</th>
<th>#5 (s)</th>
<th>Average stall time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 second</td>
<td>67.8</td>
<td>29</td>
<td>30.4</td>
<td>55.3</td>
<td>60.6</td>
<td>48.6</td>
</tr>
<tr>
<td>2 seconds</td>
<td>68.5</td>
<td>55.5</td>
<td>46.8</td>
<td>56.6</td>
<td>40</td>
<td>53.5</td>
</tr>
<tr>
<td>4 seconds</td>
<td>40.1</td>
<td>114.7</td>
<td>69.2</td>
<td>49.5</td>
<td>64.3</td>
<td>67.6</td>
</tr>
</tbody>
</table>

Table 4.1: Prestudy on varying I-frame distance

**4.2 Testbed Setup**

The experiments have been performed using a high-speed LAN (100 Mbit/s). Videos and DASH player are hosted on a server which runs on a Ubuntu 16.04 machine. The client machine runs Windows 10 and the videos are streamed in a Google Chrome version 58.0.3029.
4.2. Testbed Setup

browser with the embedded player on a web page. The cache was disabled to prevent
data from being saved so that each experiment required the data to be downloaded again
which makes each experiment homogeneous. To emulate different network conditions we
used DevTools in Google Chrome. The DevTools in Google Chrome browser enables us to
simulate various connection speeds and delays natively in the browser. It is easy to emu-
late different network conditions and it lets us add custom profiles without complications,
as seen in Figure 4.2. The experiments on branching video with the OSMF player used
Dummynet [17][16] as a tool for emulating different network conditions. We tried using
Dummynet, as well, but the adaptation mechanism in our DASH player would not recognize
that the network condition had been altered which is the reason for using Chrome’s Devtools.

The default scenario for our experiments is the same as the experiments with the OSMF
player which used an end-to-end bandwidth of 2500 kbps, RTT of 150ms with four branch
options and no competing traffic. Based on this default scenario, we perform a one-factor
analysis in which we vary one parameter at a time; in particular we vary the bandwidth,
RTT, number of branch options, and the segment length. Krisnamoorthy’s [17][16] choice of a
RTT of 150ms is the average ping latency from the university to the top sites on the web, and
although we had an average ping latency of 50ms we decided to do the experiments with
150ms to resemble experiments with the Flash player. Each experiment plays the video for
90 seconds and each experiment is repeated 20 times. Of particular interest is the probability
that a stall would occur, playback interruption time, average playback rate and average play-
back quality. If the playback is interrupted for more than 100ms then that is considered a stall.

![Figure 4.2: Google Chrome DevTools: Network Throttling Profiles](image)

The segment length in the default scenario has 10 chunks. A longer playback time than
4 seconds is required to increase the segment length. Example, if we want a segment length
of 4 chunks then we have to play the video for 8 seconds. The number of branch options can
be changed by adding a branch path to the text file with the branch options. Each branch
point is a row in a text file: \((L \cdot C, (B_1, B_2, ..., B_N))\) where \(L\) is the segment length, \(C\) is the
chunk size and \(B\) is the branch path. So for the default scenario with 4 branch points, a chunk
size of 2 seconds, and a segment length of 10 chunks we write \((20,(100,80,60,40))\) in a text file.
5 Results

5.1 Available Bandwidth

Figure 5.1a plots the stall probability against the amount of available bandwidth in kbps. A bandwidth of 1500 kbps, 2000 kbps, 2500 kbps, 3000 kbps and 3500 kbps gave a stall probability of 5%, 15%, 5%, 15%, and 0%, respectively. The stall can occur at any point during video playback. However, most of the stalls occur at and after branching.

The results of average playback quality plotted against available bandwidth can be seen in Figure 5.1b. With an end-to-end bandwidth of 1500 kbps we get an average playback rate of 537 kbps. A bandwidth of 2000 kbps gave us an average playback rate of 634 kbps. We got an average playback rate of 817 when changing the connection speed to 2500 kbps. A bandwidth of 3000 kbps and 3500 kbps resulted in an average playback rate of 1203 kbps and 1254, respectively.

The results of the stall probability when testing the player under different network conditions with varying bandwidth (Figure 5.1a) does not show a distinct pattern. The stall probability shifts up and down as network bandwidth increases. At 3500 kbps, the maximum tested bandwidth, no stalls occurred. None of the tested bandwidth had a large amount of stalls, although stalls were present for all bandwidth except 3500 kbps. When limiting the bandwidth to 3500 kbps the player was able to fetch the best quality chunk for the vast majority without stalling. With lower bandwidths it made decisions to get lower quality chunks but still did stall on occasion.

An interesting observation is that for 3000 kbps none of the stalls occurred at the point of branching but during the chunk following the branching event. This means that at the time when the branching event occurs only one chunk is stored in the buffer for the selected branch path but during the playback of that chunk the entirety of the next chunk is unable to be retrieved most likely in part due to the ongoing retrieval of a chunk for a different branch path in conjunction with an overly optimistic approach by the rate adaption logic.

To better understand the stall probability, we look closer at the stall durations. Figure 5.2 shows the cumulative distribution function of the stall durations for experiments with
different bandwidths. Note that all stall times for the available bandwidth tests are plotted and shows that a brief delay of playback always occurs. However, as previously stated, only stall times over 100ms are considered stall events. Our player is consistent for those stalls that are not considered stall events, with the vast majority of those stalls being between 16ms and 32ms. This result is consistent for other test series as well. We also observe that only 5 events of a total 100 resulted in a stall time longer than 500ms.

Figure 5.1: Stall probability (a) and average playback rate (b) under different available bandwidths

Figure 5.2: Cumulative distribution function of stall durations

5.2 Round Trip Time

The stall probability under different durations of round trip times were 5% with a delay of 50ms, 100ms and 150ms. The video was not stalled during tests with delays of 200ms. The probability of a stall with a delay of 300ms was 45%. The results are plotted in Figure 5.3a. The average playback quality is 837 kbps, 768 kbps, 817 kbps, 819 kbps, and 1028 kbps with a varying RTT of 50ms, 100ms, 150ms, 200ms and 300ms, respectively. The results are plotted in Figure 5.3b.
5.3 Number of Branch Options

The lower end of round trip times display very similar behavior to each other. RTTs of 50ms, 100ms, 150ms and 200ms each had a stall probability of 5% and below. For 300ms there was a big change, the stall probability was at 45% (Figure 5.3a). The video player acted as expected since it the probability of a stall was the highest when the RTT was highest. However, what was not expected was the drastical change between 200ms and 300ms. An explanation can be found when looking at the plot in Figure 5.6b. There we can see that for 200ms the playback quality of 250 kbps was played more than in the case when the RTT was 300ms which played the video in 1300 kbps more often. We attribute this to some mechanism in the rate adaption logic that, when the player experiences longer round trip times, causes an overly optimistic approach when selecting which quality to retrieve. This is also reflected in the average playback rate, 1028 kbps for 300ms RTT and 837 kbps for 50ms RTT which was the second highest average playback rate.

![Figure 5.3: Stall probability (a) and average playback rate (b) under different round trip times](image)

5.3 Number of Branch Options

The probability that a video would stall after a branching event with 2 branch options was 0%. Increasing the branch options to 4 and 6 would increase the stalling probability to 5%. With 8 branch options the probability was 10%. The results are plotted in Figure 5.4a. This pattern of increasing stall probability with a larger number of branch options is what you might expect as the player now has to buffer more data to ensure seamless playback. The data for average playback rate plotted against the number of branch options can be found in Figure 5.4b. The average playback rate was 818 kbps, 817 kbps, 756 kbps, and 779 kbps respectively to the number of branch options 2, 4, 6, and 8.

5.4 Segment Length

A segment length of 2 or 6 gave a stall probability of 10% while a segment length of 8 or 10 gave a stall probability of 5% and the segment length that yielded a stall probability of 0% is 4. The results are plotted in Figure 5.5a. The plot in Figure 5.5b shows the average playback rate under different segment lengths. A segment length of 2 chunks gave an average playback rate of 792 kbps. A segment length of 4 chunks gave an average playback rate of 791 kbps. The average playback rate was 782 kbps, 781 kbps, and 817 kbps when the segment length consisted of 6, 8, and 10 chunks, respectively. We observe that there is some variation in
5.5 Fraction of Time per Playback Quality

The results in this section can be seen in the plot in Figure 5.6.

Available Bandwidth

For 1500 kbps the fraction of time used taken up by each playback quality is as follows: 250 kbps took up 57.9% of the time, 500 kbps took up 16.6% of the time, 850 kbps also took up 17.1% of the time, and 1300 kbps took up 8.4% of the time.

2000 kbps as our available bandwidth yielded the following distribution of time for our different playback qualities: 250 kbps was played for 65.4% of the time, 500 kbps took up 3.5% of the time, 850 kbps took up 3.4% of the time and 1300 kbps was played 27.7% of the time.
For the default case 2500 kbps, the distribution is as follows: 250 kbps was played 41.5% of the time, 500 kbps was played 2% of the time, 850 was played kbps 12.4% of the time, and 1300 kbps was played 44.1% of the time.

3000 kbps available bandwidth had the following distribution: 250 kbps was played 5.9% of the time, 500 kbps took up 1.4% of the time, 850 kbps was played 8.6% of the time, and 1300 kbps was played 84.1% of the time.

3500 kbps available bandwidth had the following time distribution: 250 kbps quality was played 4.8% of the time, 500 kbps quality was played 1% of the time, 850 quality was played 0.1% of the time, and 1300 kbps quality was played 94.1% of the time.

For the most part results are what you would expect, as the available bandwidth increases the fraction of time used for lower quality video chunks decreases. In the case of 1500 kbps, the 1300 kbps playback rate took up 8.4% of the time. For 3500 kbps available bandwidth this Figure increased to 94.1%. There is however one thing that does not display the behavior one might have thought. Between 1500 kbps and 2000 kbps there is a increase in the fraction of time taken up by the lowest quality playback rate, 250 kbps, and a corresponding decrease of the two intermediate qualities 500 kbps and 850 kbps. We are not able to offer an explanation for this behavior but we do note that it is interesting.

**Round Trip Time**

The distribution for a round trip time of 50ms was as followed: 250 kbps was played for 31.5% of the time, 500 kbps was played for 8.9% of the time, 850 kbps had 19.6% screen time, and the highest quality was played for 40% of the time.

With 100ms round trip time we got the video to play in a 250 kbps quality 44% of the time, 500 kbps 4% of the time, 850 kbps 11% of the time, and 1300 kbps 41% of the time.

250 kbps was played 41.5% of the time with an RTT of 150ms. 500 kbps was played 2% of the time. And then we have 12.4% and 44.1% that were distributed over 850 kbps and 1300 kbps, respectively.

The allocation of the qualities, 250 kbps, 500 kbps, 850 kbps, and 1300 kbps, during the tests with 200ms RTT gave us the following: 41.8%, 1.6%, 11.4%, and 45.2%.

With an end-to-end RTT of 300ms we had the following result: 250 kbps was played 19.9% of the time, 500 kbps was played 3.7% of the time, 850 kbps was played 21.9% of the time, and 1300 was played 54.5% of the time.

For the four lowest tested round trip times, the behaviour is quite similar amongst them. This is congruent with their similarity in stall probability. For the highest round trip time 300ms there is a large portion of time occupied by high quality chunks, recall that this case also had a high stall probability.
5.5. Fraction of Time per Playback Quality

Figure 5.6: Playback qualities during different bandwidth (a) and RTTs (b)
6 Discussion

This chapter discusses the results compared to the OSMF player [17] [16].

6.1 Available Bandwidth

When comparing our results to those of the OSMF implementation [17], there are a few distinct differences. For instance, when the bandwidth limit is set to 3000 kbps and 3500 kbps, for our DASH.js implementation achieves an average playback rate of over 1200 kbps. Whereas in the case of the OSMF implementation [17], the playback rate at those bandwidth limitations is around 800 kbps in both cases. This is using the single connection prefetch policy specifically, which obtains the highest playback quality of the non-naïve policies. So we can observe that there is a clear difference in that the DASH.js implementation is able to achieve a higher average playback rate. There are also drawbacks to the DASH.js implementation, namely that it has a higher stall probability. Comparing again with the single connection policy, we have for the two implementations and in increasing available bandwidth, 5% to 25%, 15% to 0%, 5% to 5%, 15% to 0% and 0% to 0% for DASH.js and OSMF, respectively. Against the other prefetching policies (excluding naïve) DASH.js comes off even more unfavourably. We argue that both of these differences are due to bitrate adaptation and prefetching policies being overly optimistic.

6.2 Round Trip Time

Observing the stall probability depending on the round trip time for the OSMF implementation [17], we can see that the single connection prefetch policy stands out as the most likely to stall among the non-naïve policies. We can also observe that in the DASH.js implementation there is not a distinct pattern of increasing stall probability for higher round trip time. Instead there appears to be a threshold above which stalls become more common. When the round trip time is set to 300ms, the highest tested RTT, the stall probability is 45%. Compared to other results we have achieved this a very high probability of stall. Looking at the average playback rates instead, we can see that OSMF seems to have a somewhat downward trend as RTT increases. In the case of DASH.js, for the most part there seems to be no impact until we reach an RTT of 300ms, for which there a sizeable increase in average playback quality. We
see two possible explanations to this. The first possible explanation is that since the quality increased for the 300ms case, there might be some rate adaption logic that erroneously causes the increase in average playback quality. We think that it is triggered when some condition is met due to the increased RTT. The second possible explanation is that since we only use a single connection, the long duration of the RTT, and that for each chunk we have to establish a new TCP connection. This causes a lot of "delays" as the connection is inactive whilst TCP handshakes are being made, thus, the effective throughput decreases. This is especially true for the scenario around branching events where many small chunks have to be retrieved. We believe that the second explanation definitely has a hand in increasing stall probability as RTT increases. We also think that the first possible explanation is the reason behind the increased average quality in the case of 300ms for our implementation. In extension this then can also affect the stall probability in this scenario.

6.3 Number of Branch Options

Overall both implementations have similar behavior trends when looking at the number of branch options. The probability of stalls increase as the number of branch options increases. The quality declines very slightly as number of branch options increases and for both implementations the average quality level is at around 800 kbps. DASH.js had more stalling events for lower number of branch options with 0% for both players at 2 options, and 5% for DASH.js and 0% for OSMF at 4 options. For 6 options DASH.js was still at 5% and OSMF had increased to 20%. Finally for 8 options, DASH.js had 10% stall probability and OSMF 30%.

6.4 Segment Length

When looking at how the two implementations perform under different segment lengths there are differences. The DASH.js implementation performs a lot better when the segment is 2 or 4 chunks long. For 2 chunk segments the best-in-case OSMF policy, greedy bandwidth, has a stall probability of 80% compared to the stall probability of the DASH.js implementation which was 10%. However, on the other end of the spectrum the tables have turned. DASH.js has a stall probability of 10% at 10 chunks per segment and OSMF is at 0%. We notice there is also a difference in average playback quality. DASH.js stays around 800 kbps for all the segment lengths and OSMF starts at around 400 kbps for the shortest segment lengths and finishes just below 1000 kbps for 10 chunks per segment.

6.5 Method

Testbed Setup

As we described in the method chapter the hardware we used in our testbed setup consisted of two computers, one acting as a server and the other as a client. These two units were connected via Ethernet to a router. Then, we had network throttling in place over this connection. Our reason for choosing this setup was because we wanted to replicate the results of previous studies [17]. It is conceivable that we could have used one computer as both server and client and then throttled the traffic on that unit. This would have eliminated the additional network between the two computers. The network between the two computers adds a level of uncertainty to our testbed setup. However, the network conditions on the physical network are very good and as stated, our aim was to replicate the setting from the previous paper.

In our setup we used a laptop running Ubuntu 16.04 as the server when using DevTools as our throttling tool, later when we used Dummynet the tests were made on a Macbook Pro
from mid 2009. The experiments with DevTools and the experiments with Dummynet were made on different computers because we were not able to get it up running before doing the main tests. For our client we used a stationary PC which was chosen because it gave lower stall times when the content was already in the buffer. We believe that this is due to better hardware allowing it to retrieve the buffered content faster. For significant stalls (longer than 100ms) there was no noticeable difference between the stationary computer and other units we tested in the client role.

Dummynet

Dummynet is a widely used network emulator that allows the user to shape network conditions [6]. As previously stated we had problems initially with setting up Dummynet for our experiments and instead we used Google Chrome DevTools to control our network conditions. After producing our results we made further attempts at setting up Dummynet and eventually got it to work. We then conducted tests with Dummynet to validate the results we got using the DevTools. The results of these tests were vastly different than those we got during our tests with Google DevTools. The Dummynet tests were far more likely to stall during the time after the branch event up to the point where we paused playback and concluded the test run. The stall probability for the branch event itself was not that different. We then tested the default 2.1.0 DASH.js reference player with Dummynet without any branching and manually made a jump in the video playback which gave the same results as we got during the Dummynet test with branching. After the jump/branching the video stalled continuously. We then logged the buffer levels during the test and found that what had happened with Dummynet was that the two buffers, audio and video, were not balanced when the stream would stall. One would be empty and the other would contain several seconds of content ready for playback. An example true of such scenario is shown in Figure 6.1. Here, we see a big difference between the audio and video buffer after 80 seconds. The audio buffer is empty at times when the video buffer is not, which results in long stalls. To conclude, we think that these differences between our results with Chrome DevTools and those with Dummynet raises questions as to the validity of the results. However, we also noticed that the behaviour we see during the Dummynet tests is also present in the default DASH.js player which explains that the problem is likely not with our branching implementation, but rather with the DASH.js player. In a more up to date version of the DASH.js player these issues could be resolved.

Figure 6.1: Video and audio buffer
Conclusion

The research questions defined for the project has for the most part been sufficiently answered. The first research question was "Is there any difference between how the two different players perform when network conditions deteriorate?". Yes, there were clear differences, DASH.js was overly optimistic when estimating the available bandwidth and as a result of this had a higher playback quality but also more stalling events. For the case of increased round trip time there were also large differences, specifically between the OSMF prefetching policies which used multiple connections and the DASH.js implementation which used a single connection. The effective throughput was reduced when only one connection was available and multiple small chunks had to be fetched. This resulted in many stalls for the DASH.js player when the round trip time was long.

The second research question was "Is there any difference between how the two different players perform when playing videos with different properties?". In the case of number of branch points the two players displayed a similar pattern but generally there were more stalling events in the DASH.js player. In the case of the number of chunks per segment there was also differences especially when the segments were short. In the case of 2 chunks per segment the OSMF player had a stall probability of 80% and the DASH.js had a stall probability of 10%.

The third research question was "Are there any other differences between the performance of the two players?" In general the DASH.js player has some rate adaption problems where it is overly optimistic. And as a result it has more stalls and has a higher average quality. The OSMF player seems to be more "stable" in the sense that it’s behavior is more predictable and often show a distinct increasing or decreasing trend, this is not the case of the DASH.js player. Instead the DASH.js player has displayed patterns that seems to be random. With more refined prefetching policies and rate adaption mechanisms this randomness could potentially be remedied.

7.1 Future Work

The DASH.js framework is undergoing development, and since we set out on implementing a branced video player within the DASH.js framework several updates have been
released. One component that we consider to be of special interest for further exploration is the rate adaption logic in DASH.js and implementation of other prefetching policies such as BOLA [22]. Contrary to the algorithm used in our DASH.js player, BOLA does not require any prediction of available bandwidth. The BOLA algorithm uses Lyapunov optimization techniques to minimize stalls and maximize the video quality being played. It would be interesting to see how BOLA can be used with interactive branched video with multiple paths to choose from.

Another area that would be of an interest to explore further is the format used to send information about planned branch events, communicated from the server to the client. In our implementation we have designed a rudimentary mechanism of relaying this information, for branched video to be more widely adopted a standard for this needs to be developed. For example, the branching information could be stored in the MPD.
Bibliography


Bibliography


