Acknowledgements

I would like to thank my family for all the support and encouragement they have given me with my research project. It has been a tough year but thanks to them I could finish it. I would also like to thank my friends who also supported me as much as they could and gave me a hand whenever I needed.
Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed: ...................................................................... Date:

........................................
Video streaming is becoming more and more popular in society, both live and on demand video. Some big companies have designed their own standards for video streaming over HTTP. In this project, we study the standard from MPEG, which is open source, called MPEG-DASH.

One of the key features of the streaming of video over HTTP is the adaptation to the network. In this project we will go deep on this, testing how big is the impact of the adaptation and designing an algorithm that adapts the streaming to the network conditions.
# Table of Contents

ACKNOWLEDGEMENTS ............................................................................................................. 2  
DECLARATION ............................................................................................................................. 3  
ABSTRACT ................................................................................................................................... 4  

1. FINAL PAPER .......................................................................................................................... 6  

2. LITERATURE SURVEY ............................................................................................................. 6  

2.1 INTRODUCTION ..................................................................................................................... 12  
  2.1.1 Motivation ......................................................................................................................... 12  
  2.1.2 Technologies .................................................................................................................... 13  
  2.1.3 Softwares ......................................................................................................................... 14  
  2.1.4 MPEG-DASH .................................................................................................................... 15  

2.2 RELATED WORKS AND ANALYSIS .................................................................................... 18  
  2.2.1 DASH Implementations .................................................................................................. 18  
  2.2.2 Adaptation Algorithms .................................................................................................... 20  

2.3 SPECIFICATION OF THE PROBLEM ................................................................................... 27  

REFERENCES .............................................................................................................................. 28  

3. TESTING AND RESULTS ........................................................................................................ 31  

3.1 PAPER’S ADAPTATION ALGORITHM .................................................................................. 32  
  3.1.1 The algorithm ................................................................................................................... 32  
  3.1.2 Simulation results ............................................................................................................ 35  

3.2 GUILLEM’S ADAPTATION ALGORITHM ............................................................................ 40  
  3.2.1 The algorithm ................................................................................................................... 40  
  3.2.2 Simulation results ............................................................................................................ 42  

REFERENCES .............................................................................................................................. 48  

APPENDIX A.1 ............................................................................................................................ 49  
APPENDIX A.2 ............................................................................................................................ 51
1. Final Paper

Adaptation algorithm for Dynamic Adaptive Streaming over HTTP

Guillem Solivellas Mulet
Department of Networking and Computing
Dublin City University (DCU)
Dublin, Ireland
guillem.solivellas@gmail.com

Abstract—Internet video concerns a significant part of the Internet traffic and it is still growing. Adaptive video streaming over HTTP is being recently adopted because it offers advantages in terms of both QoE and resource utilization. One of the key aspects of DASH (Dynamic Adaptive Streaming over HTTP) is the adaptation algorithm. In this paper we focus on them, designing a heuristic algorithm and comparing it to an existing one. We integrated the algorithm in an open source called libdash that simulates a streaming session based on the MPEG-DASH standard. We evaluated the two algorithms in three different scenarios: (i) 4 Mbps constant link bandwidth, (ii) 4 Mbps link bandwidth with long fluctuation to 192 Kbps and (iii) 4 Mbps link bandwidth with short fluctuation to 192 Kbps.

Keywords—streaming; HTTP; network; adaptation; DASH

I. INTRODUCTION

In the last years, the demand of video in internet has increased a lot, and as Cisco forecasts [1], it will keep constantly growing. The introduction of smartphones, smart TVs and new devices with internet connection, along with the evolution of internet access technologies such as LTE and IEEE 802.11 n/ac has made possible to watch online video content everywhere at every time.

The use of the protocol HTTP for video streaming offers several advantages. It is able to cross firewalls, since almost all firewalls allow HTTP connections. Furthermore, it reuses the existing resources such as Web Servers, CDN’s and Proxy Caches. Finally, HTTP is stateless so the client manages the session and it reduces the load on the server, what brings all the intelligence to the client and makes the server a storage device.

The wireless environment is nowadays the most used by the most part of the population, because it offers flexibility in terms of mobility and plug-in devices. But the main problem of the wireless connections is the sudden decrease of bandwidth because of packet losses, and it means a significant variation of the throughput. This is one of the reasons why the adaptation algorithms are a key part of the video streaming over HTTP. The aim of these algorithms is to select the most appropriate video quality at every moment according to the network conditions and to react to throughput fluctuations.

There is a big range of researches about adaptation on video streaming over HTTP. The goal of these researches is to enhance the QoE (Quality of Experience) of the user. This goal is achieved mainly by (i) avoiding playback interruptions and (ii) maximizing the quality of the video.

In this paper, we will present a heuristic adaptation algorithm designed by us and compare it to an existing one. Our algorithm enhances the convergence to the optimal video quality, designing a Fast Start phase quite aggressive. We integrated the algorithm in the Libdash framework [2], which simulates a video streaming session using the standard MPEG-DASH [3]. Then, we created a local server installed on a Unix virtual machine with the software VMware [4] to store the video content. Finally, we could test our algorithm in different scenarios forcing the features of the network using the VMWare tool.

Some prior research about this topic is presented in Section II. We describe the algorithm design in Section III and evaluate it in some different scenarios in Section IV. Finally, we conclude our work in Section V.

II. RELATED WORK

Some researchers studied the possibility to allocate the logic adaptation at the server side, like De Cicco et al. [5], who propose to use a Quality Adaptation Controller, that feedbacks the information to the server. So, the server takes all de decisions, and the client has the only task of decoding and playing the video. It is not good in terms of load at the server, because many clients may request video content at the same time and it requires a very high level of computation.

The adaptation algorithms placed at the client side take more relevance because each client uses its own resources for its own streaming session. Miller et al. [6] propose a heuristic adaptation algorithm for adaptive streaming over HTTP that focuses on specially avoiding all the playback interruptions. It
reacts very well in front of congestion events and has a very cautious behavior. However, it lacks a bit in terms of convergence to the optimum bit rate.

Jarnikov et al. [7] introduces the use of a Markov Decision Process to calculate the adaptation strategy, which is specified off-line. It is very effective when the user knows a priori the behavior of the network, but it does not usually happen, and the throughput fluctuations may cause a non-optimal performance of the algorithm.

Mok et al. [8] designs a method called QDASH (QoE-aware DASH) that calculates the available bandwidth with a very accurate technique. The adaptation is improved, but it adds a delay for the bandwidth measurement and the integration of an architecture is also required.

III. ARCHITECTURE

The adaptation of the streaming is placed at the application level of the OSI stack. The algorithm calculates the time that is required for downloading a segment and the amount of bytes that have been downloaded. Then, dividing the number of bytes (multiplied by 8 to obtain the bits) by the time, it estimates the throughput of the network. When we talk about the throughput of the network we refer to the throughput to the bottleneck of the network, which is the link that will limit the connection speed.

For the buffer purpose, the simulator adds the length of the segment to the buffer every time a segment is downloaded. Then, to simulate the playback, it uses a decoder, that will take a determined time depending on the representation downloaded, which means the amount of bytes of the segment.

So, what happen at the lower levels of the OSI stack does not concern the algorithm. This approach reduces the complexity of the adaptation, because it just requires a couple operations at the application level. The computing time used for the algorithm is negligible.

IV. ADAPTATION ALGORITHM

The main goal of an adaptation algorithm is to maximize the QoE of the user subject to the conditions of the path that the streaming will take over the network. This goal is not something specific or a parameter that can be theoretically calculated, it depends on what do the video viewers assess for their better experience. We have decided to base our algorithm on achieving the following objectives: (i) Avoid playback interruptions due to the emptying of the buffer. (ii) Maximize the video quality allowed by the network conditions. (iii) Decrease the number of video quality switches. (iv) Start the playback as soon as possible since the user requests it. The objectives are sorted by priority, what means that to achieve the less important goals we have to assure that the most important goals are not broken.

In order to achieve the goal (i), we decide to start our algorithm selecting the lowest representation until a good approximation of the throughput can be calculated (one or two segments). This feature also assures that the goal (iv) is optimized. However, this negatively affects to goals (ii) and (iii), so we decide to introduce a phase called “Fast start”, that makes the algorithm converge quicker to the optimum video quality.

B. Definition of the parameters

We will use the capital letter R to refer to representations, and a subscript to indicate which representation is (n means the current one, n+1 the next one and n-1 the previous one). We will also use the letter B to refer to buffer related variables, like \( B_{\text{min}}, B_{\text{low}}, B_{\text{high}} \) and \( B_{\text{delay}} \), while \( B_0 \) will be the level of the buffer at the time \( t \). The variable \( \text{numReps} \) defines the number of representations between the current representation and the estimated throughput. The variable \( \text{fastStart} \) indicates whether we are in the Fast Start phase or not. Finally, the variable \( \tau \) is the length of the segments in seconds and \( \Delta t \) is the time between two consecutive representations.

C. Description of the algorithm

<table>
<thead>
<tr>
<th>Input: ( B(t) ) and ( \text{numReps} )</th>
<th>Output: ( R_0 ) and ( B_{\text{delay}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>fastStart = true; ( B_{\text{delay}} = 0 );</td>
<td></td>
</tr>
<tr>
<td>If: ( \text{fastStart} \land R_0 &lt; R_{\text{max}} )</td>
<td></td>
</tr>
<tr>
<td>If: ( \text{numReps} &gt; 2 )</td>
<td>( R_0 = R_{\text{min}} );</td>
</tr>
<tr>
<td>Else if: ( \text{numReps} == 2 )</td>
<td>( R_0 = R_{n\text{-1}} );</td>
</tr>
<tr>
<td>If: ( \text{numReps} &lt; 2 )</td>
<td>fastStart = false;</td>
</tr>
<tr>
<td>Else:</td>
<td></td>
</tr>
<tr>
<td>If: ( B(t) &lt; B_{\text{low}} )</td>
<td>( R_0 = R_{\text{min}} );</td>
</tr>
<tr>
<td>Else if: ( B(t) &lt; B_{\text{low}} )</td>
<td>( R_0 = R_{n\text{-1}} );</td>
</tr>
<tr>
<td>If: ( B(t-\Delta t) &gt; B(t) )</td>
<td>( R_0 = R_{n\text{-1}} );</td>
</tr>
<tr>
<td>Else if: ( \text{numReps} &gt; 0 )</td>
<td>( R_0 = R_{n\text{-1}} );</td>
</tr>
<tr>
<td>Else:</td>
<td></td>
</tr>
<tr>
<td>If: ( R_0 == R_{\text{max}} )</td>
<td>( B_{\text{delay}} = \tau );</td>
</tr>
<tr>
<td>Else:</td>
<td></td>
</tr>
</tbody>
</table>

Section 1: Pseudo-code of the algorithm.
The code of the algorithm written in a logic language is provided in section 1. The algorithm is computed every time a new segment is requested.

It takes two input parameters: (i) The current level of the buffer \(B(t)\) when the adaptation is set and (ii) the number of representations between the current representation and the calculated throughput \(\text{numReps}\). This last parameter is processed before the algorithm starts, and requires a simple set of operations. The buffer level is monitored during all the streaming session and the throughput is calculated every time a segment is downloaded.

The adaptation outputs two parameters: (i) The next representation to be selected \(R\) and (ii) the delay to be added before downloading the next segment. This last parameter has the aim to avoid buffer overflows.

The algorithm starts selecting the lowest representation until the throughput can be estimated with some precision. This approach also has a benefit which is that minimizes the time that the user has to wait until the playback starts. As our prior objective is to avoid playback interruptions, starting with the lowest representation also assures this goal (considering the throughput will always be higher than the lowest video bitrate possible) and let the buffer be quickly filled.

In order to achieve the goal of maximizing the video quality while trying to decrease the number of quality switches, we introduce the Fast Start phase. This part has an aggressive behavior in terms of switching up the representations. It calculates how many representations there are between the current and the estimated throughput and, depending on it, chooses the action to do. If this number is higher than 2, it increases 2 representations. If this number is 2, it increases 1 representation. Finally, if it is lower than 2, it ends the Fast Start phase meaning that we already or almost reached the desired representation.

Once the Fast Start phase is ended, we assume that the behavior has to be more conservative and capable to react to bandwidth fluctuations avoiding playback interruptions. This part considers the level of the buffer and, depending on it, decides what conditions must be checked before taking an action.

If the buffer level is under the threshold \(B_{\text{min}}\), it means that we are close to get a playback interruption. It can be due to a heavy sudden congestion or maybe because the buffer is slowly decreasing and the algorithm does not detect it.

If \(B(t)\) is between \(B_{\text{min}}\) and \(B_{\text{low}}\), it means that the buffer level is low but not critical, and another condition is checked. If the previous buffer level, \(B_{\text{high}}\), was higher than the current one, the algorithm figures that there may be a negative fluctuation of the throughput, so it decides to decrease one representation. If the previous condition is not achieved, it checks if the number of representations between the current and the estimated throughput is higher than 0. This means that the buffer level is equal or higher than before and the bandwidth can afford, at least, the next representation, so this is what the algorithm selects. If none of the previous conditions are achieved, it decides to keep the same representation.

If \(B(t)\) is between \(B_{\text{low}}\) and \(B_{\text{high}}\), it means that the buffer has a steady level, and the following condition is checked. If the previous buffer level, \(B_{\text{high}}\), was higher than the current one plus one segment, \(\tau\), the algorithm figures that there may be a congestion, so it decides to decrease one representation. If the previous condition is not achieved, it checks if the number of representations between the current and the estimated throughput is higher than 0. This means that the buffer level is steady and higher than at the previous segment, so it decides to increase one representation. If none of the previous conditions are achieved, it decides to keep the same representation.

If \(B(t)\) is over \(B_{\text{high}}\), it means that the buffer level is too high and there is a risk of buffer overflow. To avoid it, the algorithm has two options: select a higher representation or introduce a delay before downloading the next segment. So, it checks if the current representation is the maximum and decides to increase one representation if it is possible or, if it is not, to add a delay, \(B_{\text{delay}}\), equal to the segment length before downloading the next segment.

V. TEST AND RESULTS

We used an open source called libdash for the integration of our algorithm and the software VMWare to create our server that stores the content and personalize the desired environments.

We evaluated the implemented adaptation algorithm in three different scenarios that we called: (i) ideal, (ii) short heavy congestion event and (iii) long heavy congestion event. With these three scenarios, we can observe the behavior of the algorithm in the most of the possible circumstances.

In figure 2 we can see the scenario where the simulations have been placed. It consists of a server, a link which is personalized for each environment, and a client. The server stores all the video content and the client has the libdash simulator, which connects to the server, downloads the video content and simulates the playback.

![Figure 1: Scenario where the tests are made.](image)

A. Ideal

In this scenario we used a link with a 4 Mbps bandwidth and no fluctuations of this.
We can see in figure 4 how the buffer grows very fast at the beginning because the algorithm starts selecting the lowest representation. According as the increase of the representation selected, shown in figure 3, the buffer level increases slower until it reaches its maximum.

An important feature to highlight from this scenario is the convergence of the algorithm from the lowest representation (first one selected) to the optimum (the highest). In figure 5 we can see that it is achieved at the 10th segment. This is thanks to the Fast Start phase introduced, that acts quite aggressive in terms of increasing the video quality. Compared to the solution of Miller et al. [5], which needs 21 segments to reach the highest representation, this is the biggest improvement of our algorithm.

B. Short heavy congestion event

In this scenario we used a link of 4 Mbps with a heavy fluctuation of the bandwidth to 192 Kbps during a very short period of time.
We can see in figure 7 how the algorithm reacts to the congestion event decreasing one representation. As this event is very short, the algorithm detects that, for the next segment, it has ended and still has a good level of the buffer, so keeps the current representation until the buffer allows it to increase again to the highest. This behavior is very similar to the Miller’s algorithm, which is able to detect the congestion event and control it in the same way as our algorithm.

C. Long heavy congestion event

In this scenario we used a link of 4 Mbps with a heavy fluctuation of the bandwidth to 192 Kbps during a long period of time.

We can see in figure 10 how the algorithm reacts to the congestion event, trying to only decrease one representation, but finally required to select the lowest representation. As the heavy congestion event is quite long, there is a playback interruption, shown in figure 11. When the congestion ends, the algorithm converges again to the optimum representation, but slower than at the beginning because it does not use the Fast Start method. This behavior is very similar to the Miller’s algorithm, which is able to detect the congestion event and control it in the same way as our algorithm.
VI. CONCLUSIONS

We presented an adaptation algorithm for video streaming over HTTP using the standard MPEG-DASH. It is a heuristic algorithm that aims, first of all, to avoid playback interruptions and, secondly, to maximize the quality of the video as much as possible. We evaluated the algorithm in three different circumstances. In the first, with an ideal behavior of the network, it is capable to achieve the optimum representation quite fast (on the 10th segment). In the second, with a short negative peak of the bandwidth from 4 Mbps to 192 Kbps approximately, it reacts decreasing only one representation and avoids playback interruptions. In the last one, with a long negative peak of the bandwidth from 4 Mbps to 192 Kbps approximately, it tries to decrease only one representation, but when it detects that the buffer is too low, it selects the lowest representation.

The research about adaptation algorithms for streaming over HTTP is a key fact for the QoE of the video viewers on internet, who are growing every day.

REFERENCES

2. Literature Survey

2.1 Introduction

2.1.1 Motivation

In the last years the demand of video in internet has increased a lot, and as Cisco forecasts [1], it will keep increasing. The introduction of the smartphones along with the evolution of the internet access technologies such as LTE has made possible to watch a video online everywhere at every time.

Figure 1: Internet video percentage of the internet traffic [2].

The internet video traffic can be divided in different kinds of video. The figure 2 shows the impact that each kind of video has into the global video traffic. Nowadays the fastest-growing category is internet video to TV.
2.1.2 Technologies

This is the reason why new technologies have appeared. The streaming over HTTP is one of them, with the solutions developed by the main companies of the world:

- **Microsoft Smooth Streaming**.
- **Apple Inc. HTTP Live Streaming (HLS)**.
- **Adobe Systems HTTP Dynamic Streaming (HDS)**.

Later on, MPEG developed their own open source solution called **MPEG-DASH**, which was introduced as a Draft International Standard on January 2011. During this project, we will focus our study on this one.

The streaming over HTTP is a very good way to watch video over internet, mainly because of the protocol. As its name says, it uses HTTP that is able to cross Firewalls and NAT's without any problem, and leverages the existing infrastructure such as Web Servers, Proxy caches and Content Delivery Networks (CDNs). Hence, it only needs a web server where the video is stored, a client with internet access to download the content and a software in the client side able to reproduce this kind of content.

The importance of the Dynamic Adaptive Streaming over HTTP is the adaptability to the network. It means that they are able to use the properly video quality according to the network capabilities, in order to optimize the QoE (Quality of Experience) of the users. Thus, the goal of this technology is to provide the best video quality to the users supported...
The most important parameters to take into account when choosing the video quality are:

- Resolution of the client device.
- Bottleneck's throughput of the network.
- Level of the buffer.

In order to be able to offer different qualities of the video, the web server must have the video stored with different qualities, of course. Then, if we want to be able to switch the video quality during the viewing depending on our network's performance, the video must be split in small chunks called segments. So, in conclusion, the web server must have the same video with different qualities and divided in segments. Then, in order to be more adaptive, we can choose the duration of these segments, so we add another parameter. To map where the segment X with segment duration Y and quality Z is, the server uses a file called MPD (Media Presentation Descriptor). This file contains all the information the client needs to know to download each segment. So, the summary of the procedure is:

1. The client requests the MPD to the server.
2. Taking into account the network conditions and using the MPD, the client downloads the segments from the server.
3. The client plays the video with a software able to reproduce streaming over HTTP.

### 2.1.3 Softwares

As was mentioned earlier, the client needs a software to reproduce the video using this technology. The main players that can reproduce this content are:

- **VLC** with the dash plugin from ITEC [3].
- **OSMO4** from GPAC.
- **Libdash** from bitmovin Gmbh.

There are also softwares that can generate DASH content:

- **MP4Box** from GPAC.
- **DASHEncoder** from ITEC.
2.1.4 MPEG-DASH

In order to understand how MPEG-DASH works, the figure 3 illustrates a simple streaming scenario between an HTTP server and a DASH client. The server stores the video content after it is encoded in the DASH format. Thus, the server will contain the Media Presentation Descriptor (MPD) and the segments (chunks of the video). The client uses a DASH software that tracks the throughput of the network and chooses the best video quality that can be supported. The network between them is a composition of subnetworks able to transport HTTP packets.

![Figure 3: MPEG-DASH scenario.](image)

First of all, the DASH client requests the MPD. By parsing it, the client learns all the characteristics that this video has, such as the available bit rates, the offered resolutions, the encoded alternatives of multimedia, accessibility features and required Digital Right Manners (DRM), etc. Furthermore, the MPD contains the mapping of the segments and their URL to be downloaded. Once the client has used all the previous information, it can start streaming the content by requesting the segments to the server (using the mapping of segment-URL) with HTTP GET requests.

During the streaming, the DASH client keeps monitoring the network bandwidth fluctuations and some other stats of the network. Using this information, the client can adapt to the network requesting the optimized video quality according to the network's conditions. There are many different adaptation algorithms that decide what video quality to use in each situation. In the section 3.2.2, some of them are presented.

The MPD is an XML file that contains all the characteristics of the multimedia content (video, audio, text, etc.). This file consists of one or multiple periods, which are intervals of
time. Each of these periods is composed by one or multiple adaptation sets, which provides the information about one or multiple media components. Finally, each adaptation usually includes multiple representations, which are encoded alternatives of the same media component but with different characteristics. Figure 4 shows a block diagram of an example of a MPD.

The most of DASH tests are done with the “BigBuckBunny” video, created by ITEC [4]. In order to understand the XML structure of the MPD, figure 5 shows an example.

Another important aspect of MPEG-DASH is the format of the segments. Usually the first segment contains initialization information and the others are consecutive chunks of the
Each media segment is assigned a unique URL, index, start time and duration. MPEG-DASH defines segment-container formats for both ISO Base Media File Format [5] and MPEG-2 Transport Streams [6].

To conclude, it has to be clear that the MPEG-DASH only defines the MPD and segment formats. Other aspects like delivery of the MPD, playing content, etc. are outside of its scope.

As the DASH industry forum [7] says it is expected that DASH will be adopted by nearly all European broadcasters who took their survey, between the second half of 2013 and the first half of 2014. It also says that the biggest concern by far for deployment of DASH is the lack of client availability. That is the reason why many clients that support DASH are appearing nowadays.


2.2 Related Works and Analysis

2.2.1 DASH Implementations

In this section some implementations of DASH are presented. They aim to show the performance of DASH in different scenarios, compare DASH to other technologies and mainly help to understand what DASH is and how it works.

Müller et al. [8] present test-bed enabling session mobility in the context of MPEG-DASH. This means the transfer of a DASH session from one device to another keeping the current state of the streaming. For this purpose, they use the MPEG-21 standard that supports the transaction of Digital Items (DI) among users. The CMP-enabled DASH support and the session mobility are implemented with the open source player VLC.

Their implementation briefly works as follows. Before the request of the MPD, a new file called Composition of Media Presentation (CMP) is requested, that contains all the configuration parameters. Then the streaming starts as usual using DASH. To enable the session mobility, a DI called Context Digital Item (CDI) is introduced by MPEG-21. The CDI describes the current state of the streaming and, as a DI, it can be transferred between users. Once the destination device receives the CDI, it has to request again the CMP, configure it properly for the new device settings and request again the MPD using the information of the CDI to continue with the streaming at the same point.

Timmerer et al. [9] provides a tutorial that covers: an introduction to DASH; content creation, delivery and consumption; evaluation of existing MPEG-DASH implementations.

Lohmar et al. [10] studies the dynamic adaptive streaming over HTTP by MPEG focusing on the live content and analysing the end-to-end delay and its components. The most important delays are: (i) content acquisition and preparation, (ii) packetization of media segments, (iii) asynchronous fetch of media segments, (iv) time to download the segments, (v) buffering at client side and (vi) decoding and playout in the client. (i) and (vi) are not specific for DASH so they are not studied. The end-to-end delay can be defined with the following equation:

\[ T_{\text{tot}} = T_{(ii)} + T_{(iii)} + T_{(iv)} + T_{(v)} = 5d_{ms} + d_{\text{link}}. \]

\( d_{ms} \) is the duration of the segment and \( d_{\text{link}} \) is the time that the segment is travelling through
the network (it depends on the bandwidth of the link). It is clear that to minimize the delay, the duration of the segment has to be small. Comparing to RTP, HTTP introduces an extra delay of \(2d_{\text{ms}}\) approximately.

Müller et al. [11] compare their implementation of MPEG-DASH to the solutions of Microsoft (MSS), Adobe (ADS) and Apple (HLS) for vehicular environments. They evaluate three key aspects: (i) reaction to high frequency bandwidth fluctuations, (ii) smoothing of the playback under the previous conditions and (iii) maximization of the bandwidth minimizing the number of quality switches.

The experiments are made under three different network emulation settings that have been done during car drives with a HUAWEI E169 HSPDA USB Stick using a SIM-card. The results show that MSS has the best performance in terms of average bit rate. Furthermore, it switches less than ADS and MPEG-DASH, although not less than HLS. Finally, ADS does not guarantee a smooth playback while the other systems achieve it remarkably well. MPEG-DASH shows that is able to compete with these commercial systems and they propose the improvement of the adaptation process as a future research.

Sánchez et al. [12] studies the performance of Scalable Video Coding (SVC) instead of H.264/AVC for DASH environments of video on demand. The main advantage of SVC in a DASH scenario is the on-the-fly adaptation when coping with congestion events, adding or subtracting quality layers to suit with the network capabilities.

The simulations are made in two different congestion scenarios: 1) DASH clients requesting too many packets in peak hours and 2) cross-traffic due to DASH clients sharing resources with other users. The results show that in the first scenario SVC performs just a little bit better than AVC, but in the second scenario SVC is much more efficient than AVC. In a real situation, where both scenarios may be combined, SVC would perform better in front of congestion scenarios.

Lederer et al. [13] presents a dataset for DASH that allows everybody to use the same and make it easy to compare solutions. They also present an open source tool called DASHEncoder [14], which enables to create DASH content. Finally, the paper shows an evaluation of the optimal segment size in determined network conditions, using their own dataset.
Ognenoski et al. [15] proposes a segment-based teletraffic model based on the MPEG-DASH standard. They investigate the probability metrics (probabilities of buffer overflow, buffer empty and buffer active) that optimizes the performance for different segment sizes at the server and different buffer sizes at the client. It is known that small segment sizes increase the probabilities of empty buffer and active buffer, while decreasing the probability of buffer overflow, and vice versa. The performance metrics are also improved when increasing the buffer size at the client.

The results show that when a larger buffer is selected at the client, using appropriate segment sizes the overall performance is improved.

### 2.2.2 Adaptation Algorithms

The goal of all adaptation algorithms is to optimize viewing experience subject to throughput dynamics of the TCP flow on the network path from the server to the client. Viewing experience is not a concrete concept, and some features can be more important for some people than for others. Hence, the adaptation algorithms will be based on optimizing the features which affects most people.

Rate adaptation may be performed at the sender (sender-driven rate adaptation), the receiver (receiver-driven rate adaptation) or both. When the adaptation algorithm is performed at the server, it is categorized as a sender-driven rate adaptation, and when it is performed at the client, it is categorized as a receiver-driven rate adaptation.

![Classification of the adaptation algorithms.](image)

As the sender will always be, in this situation, an HTTP web server, it will not have information about the connection with the receiver. The sender could also have many different connections at the same time and to monitor all of them at the same point will
increase significantly the load. Furthermore, if the adaptation algorithm is applied at the receiver, the client will be able to personalize its streaming strategy. For these reasons, it is expected that all the adaptation algorithms will be receiver-driven in the future.

However, we will look at some sender-driven adaptation algorithms in this section.

![Figure 7: Sender-driven adaptation algorithms.](image)

Luo et al. [16] presents an end-to-end video streaming framework with the novelty of introducing packet scheduling at the server side to improve the video quality under congested network conditions. [16] is an extension of Luo et al. [17] that proposes a multi-buffer scheduling scheme for video streaming. What the packet scheduling does is prioritize some packets over others in order to enhance the QoE of the client, creating multiple buffers sorted by priority levels from the sender buffer. They also develop a rate-based congestion control scheme with the aim to use the optimal rate avoiding playback interruptions.

They use the software NS-2 for the simulations and the H.264 codec to evaluate the performance of the solution. Their results show that a high average visual quality is achieved and, when bit rate changes occur due to congestion events, the quality of the decoded frames changes smoothly. However, the goal of this framework is to maximize the receiver’s buffer while using the minimum resources and offering a good video quality level, but in any case aims to achieve the maximum.

Kuschnig et al. [18] evaluates and compare three existing adaptation algorithms, two of them are based on bandwidth estimation and the other one is based on a priority-/deadline-driven approach. The novelty is that they make use of the scalability features of H.264/SVC video codec combined with the adaptation algorithms.

The first algorithm is called Application-layer Bandwidth Estimation and calculates the throughput using the amount of bytes sent in a time interval. It will be calculated every after
a Group of Pictures (GOP) is sent, so the server will divide the number of bytes by the time required. The next bit rate representation is calculated as the average of the last 5 throughput calculations. The second algorithm is called TCP Stack-based Bandwidth Estimation (TCP-BE) and uses the same technique as the previous one. The main difference between them is that instead of measuring the TCP throughput at the application level, it uses the network stack to obtain the information. The last algorithm is called Deadline-driven Adaptive Streaming and is based on the priority streaming paradigm. It consists of giving more priority to the most important frames for the visual quality of the user and defining a schedule.

They evaluate both algorithms on an under-provisioned links scenario and on a congestion scenario. The algorithms perform well in terms of video quality and timely delivery. However, their results show that the algorithms based on bandwidth estimation have problems when the GOPs are too large, but the priority-/deadline-driven approach can cope with larger GOP sizes.

De Cicco et al. [19] proposes a Quality Adaptation Controller (QAC) for live content. This algorithm uses a feedback control, in which the controller feedbacks information to the video server for adjusting the quality levels and its sending buffer at the server side. Thus, the client has the only task of decoding and playing the stream.

The figure 8 and figure 9 show the block diagram of the control loop and the QAC server architecture.

![Figure 8: Block diagram of the control loop.](image8)

![Figure 9: QAC server architecture.](image9)

The results showed a good performance when it was deployed, but nowadays it does not work as well as any of the receiver-driven algorithms. The time required to calculate the next segment’s quality level is too high because of the complexity of the system.
There are not many sender-driven algorithms, as it has been proved that when dealing with a lot of users the performance is not the best. Now we will focus on the receiver-driven adaptation algorithms.

![Diagram of adaptation algorithm performed at: Sender, Receiver, Both.]

**Figure 10: Receiver-driven adaptation algorithms.**

**Miller** et al. [20] proposes an algorithm that starts with the lowest representation, applies a fast start mechanism and finally, based on the buffer level, will decide whether switch up/down the representation or maintain the current one.

They evaluate the algorithm in real-world scenarios, implementing it as a software library in C++ that enables to interface it with HTTP streaming clients. Based on the GStreamer framework, they create a client that works with the standard MPEG-DASH. Their results show that the algorithm performs remarkably well even in a busy wireless network with a high level of interference. It avoids playback interruptions and maintains a good video quality level, but it does not try to achieve the highest quality afforded, having a conservative behaviour.

**Liu** et al. [21] proposes a receiver-driven adaptation algorithm, with the novelty that uses a smoothed HTTP/TCP throughput measurement method to detect bandwidth changes. The advantage of this method is that it does not require information from the transport layer, it only has to measure the fetch time and use other known parameters.

The adaptation algorithm is characterized by a step-wise switch up and an aggressive switch down in terms of changing the representation. It is evaluated in a scenario with exponential and constant background traffic. The results show that the congestion is detected properly and the optimum bit rate representation can be selected quickly. However, some limitations of this algorithm are that not always the calculated optimum bit rate representation is the best and there can be too many fluctuations on selecting the bit rate representation.
Jarnikov et al. [22] proposes an adaptation algorithm based on the calculation of the bandwidth that tries to maximize the QoE of the client in the long run. The receiver is composed by the client, who sends the requests and receives the data, and a controller, who chooses the quality level of the next chunk based on the information given by the client. The strategy to follow by the controller will be calculated based on a Markov decision process (MDP) [23], that will be fed by the client with a system state every after a segment is received. The optimization criteria followed is based on a revenue function that gives penalties to playback interruptions and changes of quality level and gives rewards to proportional to the selected video quality.

The main benefit of this algorithm is that the user can personalize its QoE, giving more importance to maintain a good quality level or to try to achieve the highest quality level allowing fluctuations. The evaluation is made in a real-world scenario with the aim to compare it to the adaptive streaming client that is built in an iPhone 3G (iOS 4.2). The results show that it works at least as good as the Apple solution.

Akhshabi et al. [24] compares two commercial adaptive video players over HTTP: Smooth Streaming from Microsoft and the player used by Netflix. They also compare them with the open source player OSMF from Adobe. The study is focused on the adaptation algorithm’s performance in different situations. The simulations are made on 3 different scenarios: (i) not limiting the available bandwidth, emulating an ideal case, (ii) applying bandwidth fluctuations that can last for tens of seconds, emulating traffic from other users and (iii) applying spikes in the bandwidth that can last for a few seconds, emulating a wireless scenario.

The Smooth Streaming player has a very appropriate behaviour in the scenarios (i) and (ii), acquiring the highest sustainable bit rate quickly and acting in a conservative way in terms of bit rate switching decisions. However, in the scenario (iii) it reacts too late to the spikes causing sudden drops or wrong bit rate decreases.

The Netflix player acts very similar to the Smooth Streaming player, but the results show that the player from Microsoft is more aggressive, aiming to achieve the highest possible bit rate representation while the other one focuses more on filling the buffer and then switching to higher bit rates.

The OSMF player does not have a very effective adaptation algorithm, but as it is an open
source, users can customize the code modifying the algorithm and trying to improve it.

**Bokani** et al. [25] considers using Markov Decision Process (MDP) to determine the adaptation algorithm in mobile environments. They propose three approaches based on MPD to reduce its computational cost.

The first approach is called k-MDP and it consists of updating the Cumulative Distribution Function (CDF) each time a segment is downloaded but recalling the MDP strategy only after downloading k segments. The second approach is called s-MDP and it consists of using offline bandwidth information, thus removing the online MDP overhead. The third approach is called x-MDP and, as the second solution is offline, but it computes the MDP strategy every x meters.

The results of their simulations show that offline MDP optimizations perform better than online, because they use a lot of previous information instead of the current trip information. Furthermore, the x-MDP solution provides the best performance adapting the MDP strategy for each segment road.

**Xing** et al. [26] proposes a Multi-Link Rate Adaptation (MLRA) algorithm for DASH over multiple wireless access networks. A Markov Decision Process (MDP) is used for the streaming process, which will decide the strategy to take according to given state. To set the state of the MDP, a reward function is used, adding a reward when positive conditions occur and subtracting a reward in the other case.

To evaluate the performance of MLRA, they use a mobile phone that uses 3G and WiFi for streaming video. The MLRA is compared with the algorithm proposed by K. Evensen, called KERA (studied below). The results show that MLRA has no playback interruptions while with KERA about 20% of the segments missed the playback deadline. In terms of average playback quality, KERA always tries to select the highest possible while MLRA has a more conservative behaviour in order to avoid that playback interruptions. As KERA is greedier than MLRA, the second one performs a better playback smoothness.

**Evensen** et al. [27] proposes a client-side request scheduler to request segments of the video over multiple heterogeneous interfaces at the same time. The novelty of this proposal is the subdivision of the segments into subsegments, whose sizes are calculated on the fly based on the throughput of the links. The quality adaptation mechanism that they use is based
basically on the buffer level and using the aggregated throughput.

They evaluate two different subsegment approaches: static, which uses fixed subsegment sizes, and dynamic, which calculates the subsegment size on the fly. Both approaches are evaluated in the context of on-demand and live streaming with and without buffering. When the buffer is large enough, both approaches perform similar. In the other cases, the dynamic approach has the best behaviour.

Mok et al. [28] presents a system called QDASH (QoE-aware DASH) that uses a new available bandwidth measurement method to enhance the video quality level selection. The architecture consists of two modules: QDASH-abw and QDASH-qoe. The first module is responsible of measuring the available bandwidth by RTT variations. The second module uses the output of the first one and helps the client to choose the most suitable video quality level, also taking into account the level of the client’s buffer.

The results show that the available bandwidth measured is more accurate so the adaptation is improved but, on the other hand, an extra-time is needed for the bandwidth measurement and the integration of the architecture is required.
2.3 Specification of the problem

As we have seen in some of the related works, the MPEG-DASH system has a very good performance but it still can be improved. The key aspect that can provide this enhancement is the adaptation algorithm. This research project will try to design an adaptation algorithm for DASH that optimizes the use of the resources and increases the QoE of the users.

The proposed algorithm will be applied at the client side, as we discussed above that the receiver-driven adaptive algorithms have more potential than the sender-driven. It will lead all the intelligence to the client, while the server will only store the content and answer the requests.

The main goal of the proposed algorithm, taking into account what we learned from all the related works, will be to avoid playback interruptions. This fact is achieved nowadays in many algorithms, so it is not an improvement. The improvement of this algorithm will be the aggressiveness in terms of increasing the video quality. The algorithm will study the network and try to know as much as possible about the environment of the client (e.g. wireless with high speed movement, wireless without movement, wired, wired congested, etc.). Thus, it will apply a different level of aggressiveness to each environment.

Another goal of the proposed algorithm will be to minimize the complexity of the algorithm and optimize the use of the resources. In this project, this goal will be the last to be applied, as we consider the previous ones more important for the technology and these will help to attract the users.

Finally, the figure 11 presents a scheme of the architecture of the project. It will use the code created by the co-supervisor Lejla Rovcanin and it will be adapted to our project, modifying mainly the adaptation algorithm’s part. We can see that almost all the intelligence is coped in the client’s design, in which the adaptation algorithm is placed.
Figure 11: Architecture of the code used for the project.

References

HTTP://WWW.CISCO.COM/EN/US/SOLUTIONS/COLLATERAL/NS341/NS525/NS537/NS705/NS827/
WHITE_PAPER_C11-481360.PDF

[2] - HTTP://WWW.ABC.NET.AU/TECHNOLOGY/ARTICLES/2012/06/14/3524848.HTM


DASH-WHITEPAPER-V2.1.PDF


3. Testing and results

In this section we are going to implement the adaptation algorithm from one of the previous papers and test it in our simulation in different environments. After that, we are going to improve this algorithm for a determined environment and test whether this enhancement is achieved in all the other scenarios or only in some of them. The algorithm from the paper and the designed algorithm are both heuristic algorithms. We have decided to implement a heuristic instead of a theoretical algorithm because there is not an expression of how to calculate the QoE, it is something based on the experience and the opinion of the user.

The simulations will be done using the framework *libdash* [1], which is programmed in C++. This framework simulates a MPEG-DASH streaming session, using as a server the website of ITEC [2], where all the MPDs and all the video content required is stored. It also already has a factory of some simple adaptation algorithms. In our project, we will replace the webserver with an Apache 2.0 server installed in a virtual machine using Ubuntu 14.04. The server will just store the video content required for the simulations and the MPD. We do this approach in order to get all the control of the scenario, so we can force the uplink and downlink from the client to the server to have our desired conditions. Finally, we will add our adaptation algorithms to the implemented factory and test them in some specified environments.

We will define the different environments using the following features (extracted from the DASH dataset [3]):

- Apache server: persistent or non-persistent (keepalive on or off).
- Network emulator: delay, bandwidth limitation, packet loss.

The configuration of the Apache server is easily modified from the Ubuntu 14.04 machine and the network conditions will be applied using the Linux tool called *netem* [4]. As this last tool does not support bandwidth limitation, we will achieve it with the used virtual machine *VMware* [5].

The *figure 12* shows the environment with the configurable parameters.
For the simulations we are going to use the video sequence downloaded from http://www-itec.uniklu.ac.at/ftp/datasets/mmsys12/BigBuckBunny/bunny_2s_480p_only/. The most important features from this video sequence are:

- Segment size: 2 seconds.
- Representations: 100, 200, 350, 500, 700, 900, 1100, 1300, 1600, 1900, 2300, 2800, 3400, 4500 kbps.
- Resolution: 480x360.

3.1 Paper’s adaptation algorithm

3.1.1 The algorithm

The chosen paper is “K. Miller, E. Quacchio, G. Gennari and A. Wolisz. Adaptation Algorithm for Adaptive Streaming over HTTP. In Proc. of the IEEE International Packet Video Workshop (PV), May 2012”. The algorithm consists of two phases:

1. Fast start phase: The algorithm starts choosing the lowest representation, thus the video can start to be played quicker and it makes almost sure that the network conditions will bear it. The algorithm increases the video representation according to some conditions related to the throughput of the network. During this phase, the conditions are less severe, in order to converge faster to the appropriate bitrate. Furthermore, this phase has not restrictions of the buffer level to increase the representation. The fast start phase will end when one of the following conditions is reached:
a. The current representation achieves the maximum representation.

b. The current buffer level is lower than the previous buffer level.

c. The current representation is lower or equal to a condition related to the throughput.

2. No fast start phase: After the fast start phase, the algorithm changes its behavior and applies a more conservative mechanism to select the representation of the next segment. The buffer level is the parameter that will decide which condition related to the throughput will decide whether to increase, decrease or maintain the bitrate. The mechanism works as follows. If the buffer level is under a lower threshold, the representation is automatically decreased. If it is between the lower threshold and an optimum threshold, the algorithm checks a condition related to the throughput. If it is between the optimum threshold and the upper threshold, again another condition related to the throughput will decide. Finally, if the buffer level is above the upper threshold, the representation is automatically increased. In this case, another condition related to the throughput is added, in order to avoid the buffer overflow. If this last condition is achieved, the algorithm adds a delay before downloading the next segment.

Here is the code of the algorithm, extracted from the paper:
The code of the algorithm programmed using the framework *libdash* can be seen in the Appendix A.1.

For our simulations, we will use the following values for all the parameters: $\alpha_1=0.75$, $\alpha_2=0.33$, $\alpha_3=0.5$, $\alpha_4=0.75$, $\alpha_5=0.9$, $B_{\text{min}}=10\%$, $B_{\text{low}}=40\%$, $B_{\text{high}}=80\%$, $\tau=2$, buffer size=30s.
3.1.2 Simulation results

1. Definition of the first environment:
   - Bandwidth: 4 Mbps.
   - Long bandwidth fluctuation to 192 Kbps.
   - Server: keepalive off.

The *figure 14* shows the evolution of the monitored throughput of the link. Then, according to this approximation, we can see in *figure 15* and *figure 16* the evolution of the video bitrate (representations downloaded) and the buffer.

![Throughput in [Kbps]](image)

Figure 14: Evolution of the monitored throughput.
2. Definition of the second scenario:
   - Bandwidth: 4 Mbps.
   - Server: keepalive off.

The figure 17 shows the evolution of the monitored throughput of the link. Then, according to this approximation, we can see in figure 18 and figure 19 the evolution of the video bitrate (representations downloaded) and the buffer.
Figure 17: Evolution of the monitored throughput.

Figure 18: Evolution of the downloaded representations.
In this scenario, which is an ideal scenario with a simple limitation of the bandwidth equal to the highest video bit rate possible, we can evaluate some more statistics. The first one, showed in the figure 20, is how fast is the algorithm to achieve the maximum representation (in terms of segments). We see that the highest representation is achieved at the segment number 21. Thanks to the “fast start” phase of the algorithm, it starts with a quick increase, but when this phase ends, it becomes more conservative and waits to fill the buffer over the 80% to increase again. The second feature we can see is when does the playback start. Looking at the stored stats, we see that the first segment starts to be downloaded at the second 0.796 and the playback starts at the second 1.199.
3. Definition of the third environment:
   - Bandwidth: 4 Mbps.
   - Short bandwidth fluctuation to 192 Kbps.
   - Server: keepalive off.

The *figure 21* shows the evolution of the monitored throughput of the link. Then, according to this approximation, we can see in *figure 22* and *figure 23* the evolution of the video bitrate (representations downloaded) and the buffer.
3.2 Guillem's adaptation algorithm

3.2.1 The algorithm

The designed adaptation algorithm is a heuristic algorithm that tries to be very aggressive in terms of achieving as fast as possible the highest representation allowed by the throughput. It consists of two phases:

1. Fast start phase: this part is the one that will help us to achieve the optimum video bit rate possible according to the bandwidth conditions. It calculates how many representations there are between the current and the throughput estimated, and, depending on it, chooses the following: if this number is higher than 2, it increases 2 representations; if this number is 2, it increases 1 representation; if it is lower than 2, it ends the phase start meaning that we already or almost reached the desired representation.

2. No fast start phase: this part is the one that will deal with the bandwidth fluctuations. It considers the level of the buffer and, depending on it, decides the following:
   a. If it is under Bmin, the minimum representation is selected.
   b. If it is between Bmin and Blow, it uses another condition. If the previous buffer level was higher than the current one, one representation is decreased. If the previous condition is not achieved, and if the number of representations between the current and the throughput estimated is higher than 0, one representation is increased. If none of these are achieved, it keeps the same representation.
c. If it is between Blow and Bhigh, it uses another condition. If the previous buffer level was higher than the current one plus one segment (2 seconds in our case), one representation is decreased. If the previous condition is not achieved, and if the number of representations between the current and the estimated throughput is higher than 0, one representation is increased. If any of these are achieved, it keeps the same representation.

d. If it is over Bhigh, it uses another condition. If the current representation is the highest, it adds a delay equal to the segment length (2 seconds in our case) to avoid buffer overflow. Otherwise, one representation is increased.

Here is the pseudo-code of the algorithm:

```plaintext
Input: B(t) and numReps
Output: R_n and B_delay

fastStart = true;
B_delay = 0;

If: fastStart ^ R_n < R_max
    If: numReps > 2
        R_n = R_{n+2};
    Else if: numReps == 2
        R_n = R_{n+1};
    If: numReps < 2
        fastStart = false;
Else:
    If: B(t) < B_{min}
        R_n = R_{min};
    Else if: B(t) < B_{low}
        If: B(t-Δt) > B(t)
            R_n = R_{n-1};
        Else if: numReps > 0
            R_n = R_{n+1};
    Else if: B(t) < B_{high}
        If: B(t-Δt) > B(t) + τ
            R_n = R_{n-1};
        Else if: numReps > 0
            R_n = R_{n+1};
    Else:
        If: R_n == R_{max}
            B_{delay} = τ;
        Else:
            R_n = R_{n+1};

The code of the algorithm programmed using the framework libdash can be seen in the Appendix A.2.
```
3.2.2 Simulation results

1. Definition of the first environment:
   - Bandwidth: 4 Mbps.
   - Long bandwidth fluctuation to 192 Kbps.
   - Server: keepalive off.

The *figure 24* shows the evolution of the monitored throughput of the link. Then, according to this approximation, we can see in *figure 25* and *figure 26* the evolution of the video bitrate (representations downloaded) and the buffer.

![Throughput in [Kbps]](image1)

*Figure 24: Evolution of the monitored throughput.*

![Representation in [Kbps]](image2)
2. Definition of the second environment:
   - Bandwidth: 4 Mbps.
   - Server: keepalive off.

The figure 27 shows the evolution of the monitored throughput of the link. Then, according to this approximation, we can see in figure 28 and figure 29 the evolution of the video bitrate (representations downloaded) and the buffer.
As we did with the paper’s algorithm in the “ideal” scenario, it is significant to show how the video bitrate converges to the ideal (maximum) in terms of segments. In the figure 30 we can see that the optimum representation is achieved at the segment 10, which is quite better than with the previous adaptation algorithm.
Figure 30: Convergence of the representations from the lowest to the highest.

3. Definition of the third environment:
   - Bandwidth: 4 Mbps.
   - Short bandwidth fluctuation to 192 Kbps.
   - Server: keepalive off.

The figure 31 shows the evolution of the monitored throughput of the link. Then, according to this approximation, we can see in figure 32 and figure 33 the evolution of the video bitrate (representations downloaded) and the buffer.

Figure 31: Evolution of the monitored throughput.
3.3 Comparison of the two algorithms

The two algorithms are very similar. I introduced an enhancement in the Fast Start phase in order to increase the convergence from the lowest to the highest representation. In the second scenario, we can see that with paper’s algorithm the goal is achieved at the segment number 21, while with my algorithm it is achieved at the segment number 10. It is quite a good improvement because, with the segment length selected, the user can enjoy the video with the best possible quality 22 seconds before than using the paper’s algorithm.
In the other two scenarios, we can see how the behavior is exactly the same in terms of output parameters. In the first scenario both algorithms try to control the congestion by decreasing the representation one level for each segment, but when they detect that the congestion is too heavy, they immediately switch to the lowest representation. In the third scenario, both algorithms try the same than in the first one, and they are able to control the congestion, because it is shorter than in the first scenario.

Finally, although the reaction to fight the congestion is almost the same, the recovery from it is faster using my algorithm. It is due to the fact that my algorithm does not need to have the buffer over a certain high level to increase the representation. It first calculates if the level of the buffer is not critical. Then, if the buffer level is increasing regarding to the previous segment. And finally, if the throughput can afford a higher representation, increase it by one. So, the difference is that the paper’s algorithm needs to be sure that the conditions to increase the representation are achieved, while my algorithm can predict if the conditions will be achieved in a short period of time.
References


HTTPS://WWW-ITEC.UNI-KLU.AC.AT/FTP/DATASETS/MMSYS12/BigBuckBunny/MPDs/


Appendix A.1

Code for the paper’s adaptation algorithm using the libdash framework:

Function `getNextChunk()`:

```c
if(this->getBpsLastChunk() == 0)
    return this->getNextChunk(this->count++, 0);

int nextRep = RepIdToBitrate(atoi(BitrateToRepId(this->currentRep).c_str())) + 1;
if(this->fastStart && this->currentRep < RepIdToBitrate(13) && this->previousBuffer <= (int)(this->getBufferedMilliSec() / 1000) && this->currentRep < (this->a1 * this->getBpsLastChunk())){
    if(this->getBufferPercent() < this->bmin){
        if(nextRep <= (this->a2 * this->getBpsLastChunk())){
            this->currentRep = nextRep;
        }
    } else if(this->getBufferPercent() < this->blow){
        if(nextRep <= (this->a3 * this->getBpsLastChunk())){
            this->currentRep = nextRep;
        }
    } else{
        if(nextRep <= (this->a4 * this->getBpsLastChunk())){
            this->currentRep = nextRep;
        }
        if(this->getBufferPercent() > this->bhigh){
            this->bdelay = (int)(this->getBufferPercent() * 0.3) - ((int)(this->bhigh * 0.3) - 2);
        }
    }
}
else{this->fastStart = false;
    if(this->getBufferPercent() < this->bmin){
        this->currentRep = RepIdToBitrate(0);
    } else if(this->getBufferPercent() < this->blow){
        if(this->currentRep != RepIdToBitrate(0) && this->currentRep >= this->getBpsLastChunk()){
            this->currentRep = RepIdToBitrate(atoi(BitrateToRepId(this->currentRep).c_str()) - 1);
        }
    } else if(this->getBufferPercent() < this->bhigh){
        if(this->currentRep == RepIdToBitrate(13) || nextRep >= (this->a5 * this->getBpsLastChunk())){
            if(((int)(this->getBufferPercent() * 0.3) - 2) > (50 * 0.3))
                this->bdelay = 2;
            else
                this->bdelay = (int)(this->getBufferPercent() * 0.3) - (50 * 0.3);
        }
    } else{if(this->currentRep == RepIdToBitrate(13) || nextRep >= (this->a5 * this->getBpsLastChunk())){
        if(((int)(this->getBufferPercent() * 0.3) - 2) > (50 * 0.3))
            this->bdelay = 2;
        else
            this->bdelay = (int)(this->getBufferPercent() * 0.3) - (50 * 0.3);
    }
    else{
        this->currentRep = nextRep;
    }
}
}
```

Function `getNextChunk(int count, int64_t bitrate)`:

```c
Segment *seg = this->mpdManager->getSegment(count, this->BitrateToRepId(this->currentRep));
if(seg == NULL)
    return NULL;
```
Chunk *chunk = seg->toChunk();
this->currentRep = (int)chunk->getBitrate();

if(this->bdelay < 0)
    this->bdelay = 0;
chunk->setSleep(this->bdelay);
this->bdelay = 0;
return chunk;
Appendix A.2

Code for the Guillem’s aggressive adaptation algorithm using the *libdash* framework:

**Function `getNextChunk()`**:

```c
if(this->getBpsLastChunk() == 0)
    return this->getNextChunk(this->count++, 0);

int numReps = 0, rep = this->currentRep;
while(rep < this->getBpsLastChunk() && rep != RepIdToBitrate(13)){
    numReps = numReps + 1;
    rep = RepIdToBitrate(atoi(BitrateToRepId(rep).c_str()) + 1);
}

if(this->fastStart && this->currentRep < RepIdToBitrate(13)){
    if(numReps > 2){
        this->currentRep = RepIdToBitrate(atoi(BitrateToRepId(this->currentRep).c_str()) + 2);
    } else if(numReps == 2){
        this->currentRep = RepIdToBitrate(atoi(BitrateToRepId(this->currentRep).c_str()) + 1);
    }
}

if(this->getBufferPercent() < this->bmin){
    this->currentRep = RepIdToBitrate(0);
} else if(this->getBufferPercent() < this->bhigh){
    if(this->previousBuffer > (this->getBufferedMilliSec() / 1000))
        this->currentRep = RepIdToBitrate(this->currentRep).c_str() - 1);
    else if(numReps > 0)
        this->currentRep = RepIdToBitrate(this->currentRep).c_str() + 1);
} else if(this->getBufferPercent() < this->blow){
    if(this->previousBuffer > (this->getBufferedMilliSec() / 1000 + 2))
        this->currentRep = RepIdToBitrate(this->currentRep).c_str() - 1);
    else if(numReps > 0)
        this->currentRep = RepIdToBitrate(this->currentRep).c_str() + 1);
}
```

```c
if(this->currentRep == RepIdToBitrate(13))
    this->bdelay = 2;
else
    this->currentRep = RepIdToBitrate(this->currentRep).c_str() + 1);
```

```c
this->previousBuffer = (int)(this->getBufferedMilliSec() / 1000);
return this->getNextChunk(this->count++, 0);
```

**Function `getNextChunk(int count, int64_t bitrate)`**:

```c
Segment *seg = this->mpdManager->getSegment(count, this->BitrateToRepId(this->currentRep));
```

```c
if(seg == NULL)
    return NULL;

Chunk *chunk = seg->toChunk();
this->currentRep = (int)chunk->getBitrate();
```

```c
if(this->bdelay < 0)
    this->bdelay = 0;
chunk->setSleep(this->bdelay);
this->bdelay = 0;
return chunk;
```