He Yunfeng

Cache-friendly Rate Adaptation for Dynamic Adaptive Streaming over HTTP (DASH)

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 15.5.2015

Thesis supervisor:

Prof. Jörg Ott

Thesis advisor:

D.Sc. (Tech.) Varun Singh

Aalto-yliopisto
Sähkötekniikan korkeakoulu

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: He Yunfeng

Title: Cache-friendly Rate Adaptation for Dynamic Adaptive Streaming over HTTP (DASH)

Date: 15.5.2015 Language: English Number of pages: 8+83

Department of Networking Technology

Professorship: Networking Technology Code: S-38

Supervisor: Prof. Jörg Ott

Advisor: D.Sc. (Tech.) Varun Singh

The Internet in the recent years has seen a rapidly growing demand for multimedia content streaming. In order to deliver the streaming services to every corner of the Internet, HTTP streaming technologies have been widely adopted to replace the traditional RTSP/RTP streaming, due to the fact that HTTP streaming can avoid the issues arising from firewalls and NATs. Among the popular HTTP streaming technologies, Dynamic Adaptive Streaming over HTTP (DASH) has drawn the spotlights very recently.

In this thesis, we make comprehensive studies on the HTTP streaming technologies and specifically on DASH. By investigating various aspects of the DASH technology together with its underlying protocol and CDN infrastructures, we are able to identify a major problem posed by CDN caches, which still limits the performance of DASH. After understanding the advantages and drawbacks of the solutions proposed by other researchers, we have devised a unique client side rate adaptation algorithm, hoping to improve the performance of DASH in CDN networks, with a simple solution. Multiple experiments are designed and conducted to test our proposed algorithm. By studying the experiment results, we reveal how DASH performs under various network conditions, and at the same time make some conclusion on the design principles of a DASH client rate adaptation algorithm. Apart from the related studies and the algorithm proposal, some criticism is also made at the end of this thesis, as part of our DASH research conclusion.

Keywords: HTTP Adaptive Streaming, MPEG-DASH, CDNs, Cache, Rate Adaptation, DASH-JS, Media Source Extension

Preface

First of all, I would like to thank Prof. Jörg Ott for providing me with this interesting research topic and offering me generous help and valuable feedback in completing this thesis work. I also want to thank my thesis supervisor and my friend Varun Singh who has been supporting and helping every step of the way, with his generous help, enlightening guidance and all his patience. Due to personal reasons I have taken a long time in completing my thesis. There is zero doubt that without the help my supervisor and my professor have given me, I would never have completed this thesis work.

I would like to thank the Department of Communication and Networking at Aalto University for providing me the extra time to finish this thesis work and for offering me the precious opportunity to complete my degree study. I would also like to thank our Aalto administrative staff for providing me help whenever requested.

Thanks to all my family and friends who have been supporting and encouraging during this final stage of my student life in Aalto. Without their help and support, it wouldn't have been possible for me to complete my degree study.

Lastly, thanks to you readers. I hope this thesis can be of a little help to your research work.

Otaniemi, 15.5.2015

He Yunfeng

Contents

List of Figures

List of Tables

Abbreviations

APIs	application programming interfaces
CDNs	Content Distribution Networks, or Content Delivery Networks
CCN	content centric network
CSS	Cascading Style Sheets
CRP	Cache Replacement Policy
DNS	Domain Name System
DASH	Dynamic Adaptive Streaming over HTTP
DRM	Digital Rights Management
FMS	Flash Media Streaming Server
HTTP	Hypertext Transfer Protocol
HTML	HyperText Markup Language
HAS	HTTP adaptive streaming
HCDN	Hybrid Content Distribution Network
ITEC	Institute of Information Technology
LRU	Least-Recently Used
LFUDA	Least Frequently Used with Dynamic Aging
MPEG	Moving Picture Experts Group
MPD	Media Presentation Description
MSE	Media Source Extensions in HTML5
NAT	Network Address Translation
OWD	One-Way Delay
P2P	Peer to Peer
PLR	Packet Loss Rate
QoS	Quality of Service
RTCP	Real Time Control Protocol
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
RTMP	Real Time Messaging Protoco
RSP	Representation Switching policy
RTT	Round Trip Time
SAP	Stream Access Point
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
UDP	User Datagram Protocol
W3C	World Wide Web consortium
XML	Extensible Markup Language
3GPP	The 3rd Generation Partnership Project

1 Introduction

Over the years, transmitting multimedia streams over the Internet has been deemed demanding, as the IP network possesses a stateless nature and provides only a best-effort service. In spite of the challenges posed by the IP network, two types of multimedia applications have been developing quickly and receiving tremendous popularity. One type is real-time communication applications such as Skype. The other type is video streaming applications such as Netflix. Both applications require the timely delivery of video and audio data over the stateless and mail-system like IP network, which have led to the design of specific network architectures and QoS mechanisms. [1]

In traditional video streaming, one popular solution employs a stateful protocol called Real Time Streaming Protocol (RTSP) [2] to enable the client and the server to do real time communication. Upon establishing a session with RTSP, the client is able to give VCR-style commands, such as play and pause, to the streaming server [3]. According to the received commands, the streaming server uses Real-time Transport Protocol (RTP) together with Real-time Control protocol (RTCP) to guarantee a timely delivery of the media stream through UDP transport. In this streaming system, on the one hand the server and the client communicate through the dedicated RTSP control connection to control the playback of the video. On the other hand, the server has to perform the congestion control and decide the encoding rate of the stream, ensuring effective execution of the streaming process. This specially designed video streaming architecture has encountered obstacles, due to its complexity and firewall penetration and NAT traversal. Another widely deployed video streaming solution is the Adobe Flash Media Streaming Server (FMS), which uses proprietary media servers and protocols. In this Adobe solution, the Flash Media Server is connected by client endpoints who have installed Flash Player. The FMS talk to the Flash Player in Real Time Messaging Protocol (RTMP) and provide streaming service. Widely used and employed by the most famous service YouTube, Adobe FMS, however, possesses major drawbacks such as server deployment costs, requirement of Flash Player installation and poor performance on some platforms. [3]

Today video streaming has grown in such a rapid way that it has now claimed over half of the Internet traffic. [1] However with the help of the increasingly scaling network capacity and availability as well as the developing video compression technologies, the old complicated and costly streaming systems have given way to much simpler solutions , among which is the HTTP Adaptive Streaming (HAS).

As more and more commercially used multimedia streaming systems adopted the HAS technology, a standard called Dynamic Adaptive Streaming over HTTP (DASH) has been drawn by the Moving Picture Experts Group (MPEG) with the joint effort of The 3rd Generation Partnership Project (3GPP). [4] Based on the MPEG-DASH standard, proposals related to various aspects of DASH implementation have been made to improve the performance and functionality of the DASH system.

Based on these existing proposals and solutions, various aspects of the DASH technology are described and explained in latter chapters of this thesis. Further, based on an open source DASH library named as DASH-JS[1], an improved DASH adaptation algorithm is implemented by our research team. Under more realistic usage scenarios, multiple tests are conducted to analyze and evaluate the performance of our proposed DASH implementation.

1.1 History

In 2009, MPEG called for a specification work, aiming to provide a universal standard for the HTTP Adaptive Streaming (HAS) technology. The resulting ISO/IEC 23009-1 standard was published in April 2012 [4] and is referred to as MPEG-DASH. MPEG-DASH defined the basic format of Media Presentation Description (MPD) and the format of the DASH stream segments, without mendating the architecture of the DASH client implementations. MPEG-DASH has now laid the foundation for the interoperability of different DASH implementations.

In October 2014, the fifth version of the HyperText Markup Language (HTML) was completed and published by The World Wide Web consortium (W3C)[2]. Compared with the fourth version of HTML, significant changes have been made to standardize audio and video in HTML5. Although the new video element tag do not directly support video streaming, W3C has been drafting the Media Source Extension (MSE) to help solve the streaming support problem in HTML5. As Scott Kellicker puts [5], MSE has enabled the building of browser-independent video streaming players. By providing certain application programming interfaces (APIs) for JavaScript, MSE allows the JavaScript based client to push streaming data into the HTML media element.

Combining these standards and technologies, namely MPEG-DASH, HTML5 and MSE, a DASH player functionality can be integrated into a web browser through JavaScript. However, in order for the web based DASH player to function, the web browser has to support HTML5 and MSE as well as to provide certain video and audio codecs. Now Google-Chrome has provided the support for HTML5 and MSE as well as the codecs and containers of the ISO Base Media File Format [6]. "Firefox and Safari are also actively working on the support for these technologies." [5]

Around 2012, the Institute of Information Technology (ITEC) at Alpen-Adria University Klagenfurt released a set of DASH tools that can facilitate various experimental works on DASH. Among these tools is a JavaScript library, named as DASH-JS [7], that works with the Google Chrome browser. Together with these tools, a DASH dataset [8] that meet the standard of MPEG-DASH is also provided.

[1] http://www-itec.uni-klu.ac.at/dash/?page_id=746

[2] http://www.w3.org/TR/tr-groups-all#tr_HTML_Working_Group

1.2 Thesis Goals

This thesis studies HTTP streaming and the DASH technology in particular, aiming to fulfill the goals listed below:

- By conducting a research on HTTP streaming and the new DASH technology, the thesis provides a complete picture of how HTTP streaming, specifically DASH, works. Through the study, the thesis addresses the pros and cons of HTTP streaming and DASH
- By utilizing the open source library DASH-JS as well as other available tools, a real DASH system is built. Emulation work is done to test the performance of DASH under different network conditions.
- A new algorithm is proposed in the attempt to solve the performance problem of DASH when it is operating in a CDN network, where proxy caches are set between the origin web server and the clients. Our emulation test bed, in which there are proxy caches residing, enables us to mimic the DASH operations in CDNs and seek solutions for building a cache friendly DASH client.
- The performances of the original DASH-JS solution and our modified version are tested, based on our test bed. Multiple scenarios are emulated to help analyze and better understand DASH and the effectiveness of its congestion control.

1.3 Thesis Structure

We first conduct a background study on HTTP streaming in chapter 2. Except for the study on HTTP streaming technologies, both the HTTP protocol and CDNs as the HTTP supporting infrastructure, are introduced. With these background information, in chapter 3 the MPEG-DASH standard and the DASH system structures as well as some technical details are discussed. In chapter 4, the current research works on HTTP streaming and DASH are discussed. By learning these research results, the challenges facing DASH are identified and addressed. In chapter 5, our proposed algorithm, which aims to tackle the identified challenges, is disscussed in detail. Then the evaluation of our algorithm against the original DASH-JS algorithm is presented in chapter 6. Lastly, the conclusion of the thesis research is made in chapter 7. Somce technical details are included in the Appendix.

2 Background study

In this chapter, the background information of HTTP streaming is discussed. The main topics covered in this background study include the important aspects of the HTTP protocol and the HTTP-based CDN network infrastructure, which provides content caching services for HTTP streaming. Besides, HTTP streaming in general and its popular solutions are discussed , before we dive into the latest DASH (Dynamic Adaptive Streaming over HTTP) technology in chapter 3.

2.1 HTTP protocol and services

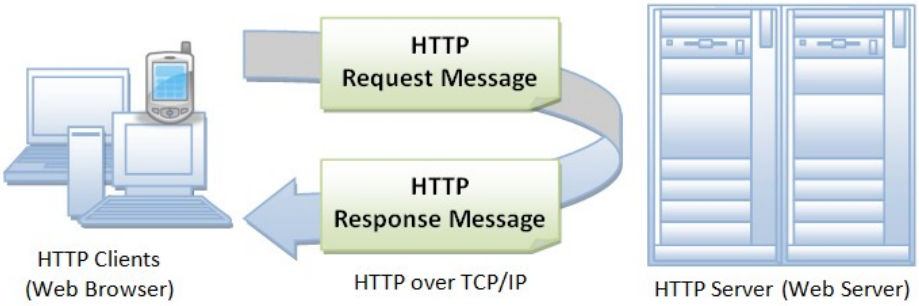
2.1.1 Basics

According to RFC 2616, "The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred" [9]. The data being transferred here refers to to the data transferred in the following-up TCP transmission. In this sense, HTTP easily supports the transfer of different types of data including multimedia streams.

The HTTP protocol uses plain text. This feature allows easier debugging as well as network analyzing. Besides, the text-based message is well supported by the UNIX system , making HTTP preferable for the globally deployed UNIX based web servers. This feature of HTTP makes it a good candidate for the streaming services, since HTTP is widely adopted and supported by web servers, as just mentioned.

Request-response model of HTTP
HTTP is a protocol that uses the request-response model in a client-server system, meaning that the basic HTTP communication involves the request message from a client and the response message from the server. This is illustrated in figure 1 quoted from [10]

Uniform Resource Locator (URL)
For the request-response system to work, the resource that is being requested and responded with must be located. This location information is provided by a Uniform Resource Locator (URL). According to RFC 1738, a URL is a unique identifier to describe the location of a piece of resource on the Internet. The generic syntax for URLs provides a framework for different schemes to describe this resource location. Based on its unique URL, a resource piece can be located and operated on. Such operation may involve access, update, replace and find attributes. Specifically the Syntax of HTTP URL is : **http ://hostname : port/filename**



[10]

Figure 1: The request and response model of HTTP

In actual operation, an HTTP client, typically a browser, sends an HTTP request message based on a Uniform Resource Locator (URL), to the origin HTTP server, to fetch the wanted resource which is generally referred to as an HTTP object. On receiving the request, the server returns an "appropriate response message, which is either the resource requested or an error message" [10].

As mentioned in RFC2616, the HTTP protocol enables the client and the server to negotiate the data representation, regardless of what type of data is being transferred. This feature made HTTP a feasible protocol for multimedia streaming, since multimedia data can be presented in many different formats. Upon receiving the requests from the client, the server can respond with the information of the media stream as well as the actual stream data through TCP transport. This process is shown in chart 2

Listing 1 shows an HTTP client issues an HTTP GET message, asking for a specific video segment whose location is indicated by a URL.

Listing 2 shows an HTTP response message sent by the HTTP server. This message tells the client that the data requested is ready to send; It also includes the media type (Content-Type) of the requested data, together with the length in bytes of the data stream. On receiving this message, the client can learn what data it is expecting to receive in the following-up TCP transmission.

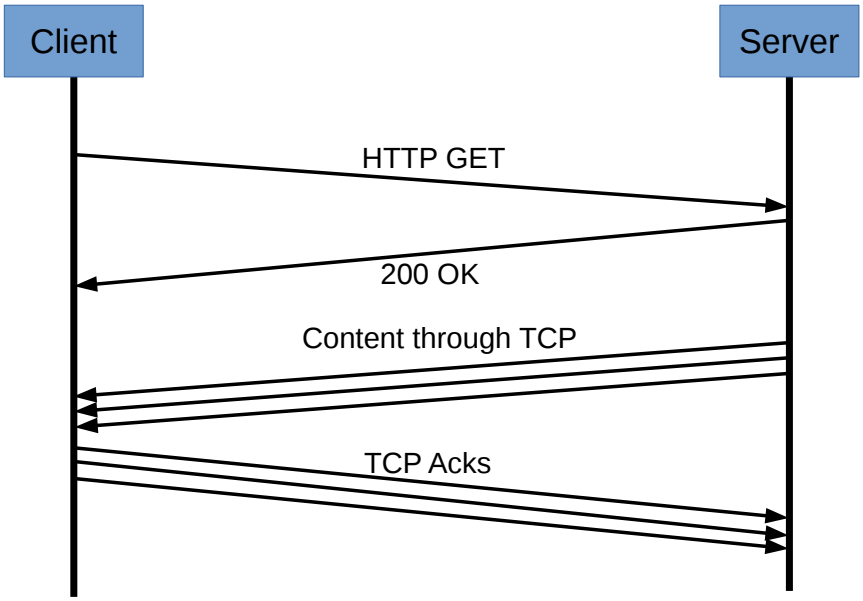


Figure 2: HTTP flow chart

Listing 1: HTTP1.1 Get message

```
GET /Dataset/OfForestAndMen/forest_1s/forest_1s_1400kbit/forest_1s8.m4s HTTP/1.1\r\n
Host: 192.168.227.168\r\n
connection: keep-alive\r\n
Cache-Control: no-cache\r\n
User-Agent: Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/35.0.1916.153 Safari/537.36\r\n
Accept: */*\r\n
Referer: http://192.168.227.168/OfForestAndMen.html?MPD=http://192.168.227.168:80/
Dataset/OfForestAndMen/MPDs/OfForest720p.mpd\r\n
Accept-Encoding: gzip,deflate,sdch\r\n
Accept-Language: en-US,en;q=0.8,zh;q=0.6,zh-TW;q=0.4,zh-CN;q=0.2\r\n
\r\n
```

Listing 2: HTTP1.1 Response message

```
HTTP/1.1 200 OK\r\n
Date: Mon, 15 Dec 2014 15:02:14 GMT\r\n
Server: Apache/2.4.4 (Unix)\r\n
Last-Modified: Mon, 14 Apr 2014 21:04:49 GMT\r\n
ETag: "3efe-4f70708afa719"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 16126\r\n
Keep-Alive: timeout=5, max=87\r\n
Connection: Keep-Alive\r\n
Content-Type: video/iso.segment\r\n
\r\n
```

2.1.2 HTTP Connection features

The legacy problem

In actual HTTP streaming, the video stream is typically divided into a large number of segments, with each segment containing a certain duration, say 1 second, of the video stream. These segments are stored in the HTTP server as HTTP objects. When the streaming starts, the request-response communications between the client and the server can happen frequently on a segment basis, meaning the client would send a large number of requests throughout the streaming process, to download the successive segments of the media stream. The detail of the streaming process can be found in section 3 of this thesis. The frequent request-response communications could cause frequent connection establishment and tear down in HTTP 1.0, since the original HTTP 1.0 specification do not allow multiple requests to use the same HTTP connection.[3] Since HTTP uses TCP as the underlying transport protocol, the frequent connection break up and establishment can greatly reduce the effective transmission rate. Besides, frequent connection tear down and establishment also create unnecessary load for the server, affecting the server performance. According to Lederer [8], a non persistent connection can lower the performance of HTTP streaming by up to 46%. For these reasons, it is helpful that the established connection is maintained for the successive requests and responses in HTTP streaming.

In the extended HTTP 1.0 implementation, an additional header, Connection: keep-alive, can be added in the HTTP request to sustain the same connection for subsequent requests and responses to use. The server receives a request with such header also attach the same keep-alive header to the response. With this header attached to all subsequent requests and responses, the connection is kept, until either the client or the server decides to tear down the connection.

Persistent connection

HTTP version 1.1 introduced the feature of persistent connection, which by default enables the same HTTP connection to maintain for multiple requests and responses. In this sense an HTTP 1.1 compatible system, which include servers, proxies and clients, can serve as a good HTTP streaming platform without causing unnecessary connection tear down and reestablishment.

Pipelining

According to krishnamurthy *et al.* [11], HTTP/1.1 supports pipelining. Although multiple requests use the same single TCP connection and the server must respond the requests in order, a client do not have to block until the response of the last request is received. Instead, the client can keep sending multiple requests over the same TCP connection without having to wait for any response. Check A.1 for more information on pipelining in HTTP streaming.

[3] http://httpd.apache.org/docs/2.2/mod/core.html#keepalive

2.1.3 Cache control in HTTP

In a typical IP network that serves HTTP, proxy servers are deployed between the servers and clients, so that the frequently requested content, which is referred to as HTTP objects, can be cached in the proxy servers and served directly by these proxies. Apart from the cache in proxies, client browsers can also cache the content and save it for future access. By allowing content caching, network transmission delay can be greatly reduced. In addition, traffic load in the network can be balanced by the proxy caches. More Details about the reason and benefit of caching are discussed in chapter 2.2.1.

HTTP/1.0 defined some simple mechanism for cache control, which is the cache control headers. A cache control header is attached in a request or response message. Clients and servers, as well as proxies in between, who join the HTTP request - response chain are supposed to perform cache operations based on the directives given by the cache control headers in the HTTP message.

Basic cache control headers include:

Expires: Used by the server response, this header includes the expiration time for caching. By including this header, the server indicates that a proxy is allowed to return the response directly from its cache for subsequent requests as long as the expiration time is not reached.

If-Modified-since: It happens when a proxy cache need to send the origin server a request to check the current validity of a response. An If-Modified-Since header is used in the request, carrying a time value provided by the Last-modified header of the previously cached response. On receiving this request, the server checks the status of the requested content. If it has been modified, the sever respond with 200 OK for the proxy cache to replace the cache entry. If the content has not been modified, the server simply respond with a 304 status code, indicating the proxy cache that the previously cached entry is still possible to use.

no-cache: Another mechanism is the no-cache header. Unlike other control headers, this header is set in the request, rather than the response. By using this header, the client indicates that the requested content shall not be cached by any proxy or by the browser cache.

These HTTP/1.0 cache mechanisms shown above, however, are heuristic and not standardized. The control headers can be misinterpreted by servers and proxies who do not support or understand them. In HTTP/1.1, these cache mechanisms are extended and clarified. New Cache-control headers are introduced to realize more functionalities. Several important headers are listed as follow:

"Etag": In HTTP/1.0, the If-Modified-Since header together with the Last-Modified

headers are employed for the revalidation of a cache entry. This mechanism however is vulnerable to clock synchronization faults. HTTP/1.1 introduces a new mechanism. The origin server generates an identity string called "entity tag" for a requested content and attach it to the Etag header of the response. As long as the content is not modified, the entity tag is preserved. Any revalidation operation between cache and the server is based on this entity tag rather than a time stamp. In this way, even when caches and the server are out of synchronization, the validation of the content is not affected.

"Cache-Control: no-cache" : This is the HTTP/1.1 replacement for the "Pragma: no-cache" header in HTTP/1.0. The difference is that "Cache - Control: no-cache" can be used both in client request and server response. There are other available directives for the "Cache-Control", for example, "Cache - Control: public" indicates that the piece of data is allowed to cache anywhere along the cache chain which the request-response communication traverse.

Some of the cache control headers mentioned above can be found in listing 1 and listing 2, which are from wireshark captures of the HTTP request and response messages during an HTTP streaming process.

2.2 HTTP supporting Infrastructure

2.2.1 Content Distribution Networks in general

One of the major motivations of using HTTP for streaming service, according to Stockhammer [2], is the availability of the globally deployed HTTP servers and the HTTP-based Content Distribution Networks (CDNs). Since HTTP streaming solutions do not require any none-standard HTTP objects to be handled by the network, the existing CDNs and its proxy servers can easily provide caching services for HTTP streaming.

In this chapter, the big picture of CDNs is drawn to help better understand the network infrastructure that serves the HTTP streaming system. More importantly this chapter provides a background for understanding the challenges that HTTP streaming is faced with in the real network environment involving CDNs. Besides, the study of CDNs also help comprehend why the experiment part of this research on HTTP streaming is conducted in the way as it is.

In a typical client server system, such as a system involving an HTTP server and clients, a client downloads data directly from the server. With the global coverage of the Internet, such simple client server system do not scale due to two major problems. The first problem is the transmission latency that is caused by geographical distances. The second problem is the latency caused by network load, as too many clients could access the same server at the same time. Needless to say, the overwhelming number of clients all over the world can easily consumes all the bandwidth and the processing

capacity of a server. Moreover, the swarm of client requests can overload the routers that connect the server, creating unacceptable queuing delay. To gain a clearer view, typical network latency can be found in the graph provided by at&t[4]. Such latency and congestion problems can impact badly especially on streaming services and real time communications.

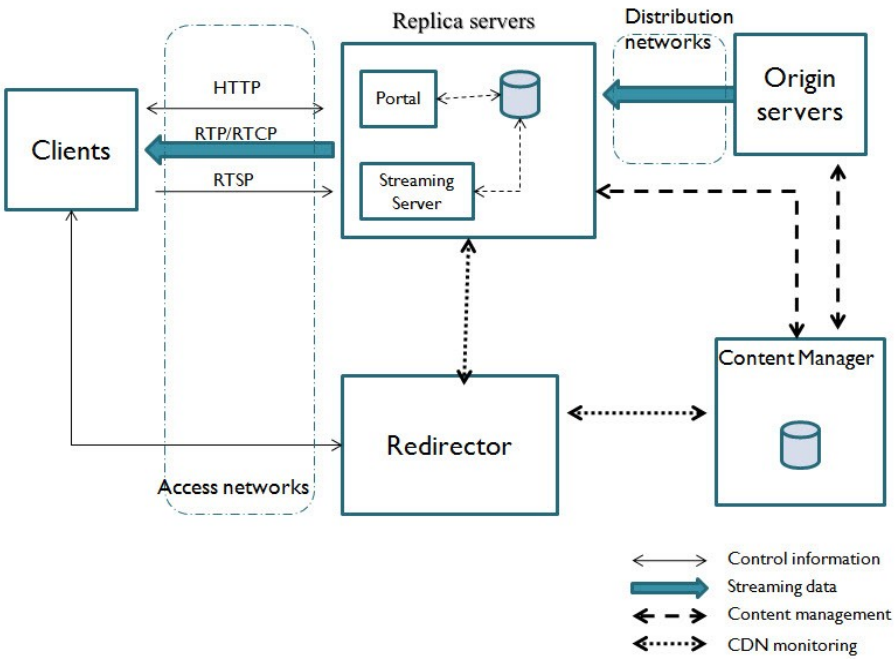
Serving as a key Internet infrastructure, Content Delivery Networks (CDNs), also known as content distribution networks, have been widely deployed to mitigate these problems. The basic idea of CDNs is to install edge servers that scatter across different geographic locations and to use these edge servers to replicate the content of the original servers. When a client sends a request and tries to access the content from the original server, the request, instead of being sent directly to the original server, is redirected to a geographically close edge server, which may have replicated or is able to replicate the requested content from the original server. In this way, the client no longer experiences the long latency to the original server. The function of CDNs, however, do not stops at resolving network latency. CDNs can also redirect client requests based on information like server load, service cost and so on. In short, CDNs helps achieve the evenly distribution of network traffic and server loads, as well as providing better Quality of Service (QoS) to the widely distributed clients.

2.2.2 CDNs topology and architecture

According to Tang *et al.* [12],the architecture of CDNs has been evolving over time, from the tree like hierarchical architecture to the distributed architecture and to the latest Hybrid Content Distribution Network (HCDN) that combines traditional CDN and Peer to Peer (P2P). Despite the fact that a CDN can be structurally complex, the basic functional components of a CDN are clear and straightforward. According to Tran *et al.* [13], a typical CDN network is composed of origin servers, edge servers (or referred to as replica servers), Clients, Access Network, Distribution Network, Content Manager and Redirector. Figure 3 illustrates the structure of a typical CDN. As explained in [13], the origin server holds the various types of data provided by content providers. In order for clients residing in different geographic locations to easily access the data, the origin server distributes the data to edge servers via the distribution network. The content manager facilitates such distribution operation and stores the information of the data distribution at the same time. The content manager also provides to the redirector this information of where and how the data is distributed in the edge servers. When a client issues a request, the redirector would intelligently redirect the request to the edge server of best choice.

Figure 3 shows a CDN case for the conventional RTP/RTSP streaming. HTTP streaming, however, can smoothly utilize the existing CDNs like the one shown in figure 3, since HTTP is well supported by todays Internet infrastructures, including CDNs. Moreover, HTTP streaming actually lessens the burden of the servers and the CDNs and scales better than traditional RTP/RTSP streaming, due to the fact

[4] http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html



[13]

Figure 3: A general architecture of CDN

that HTTP streaming is stateless and do not require the extra server resources that is demanded by RTP/RTSP streaming.

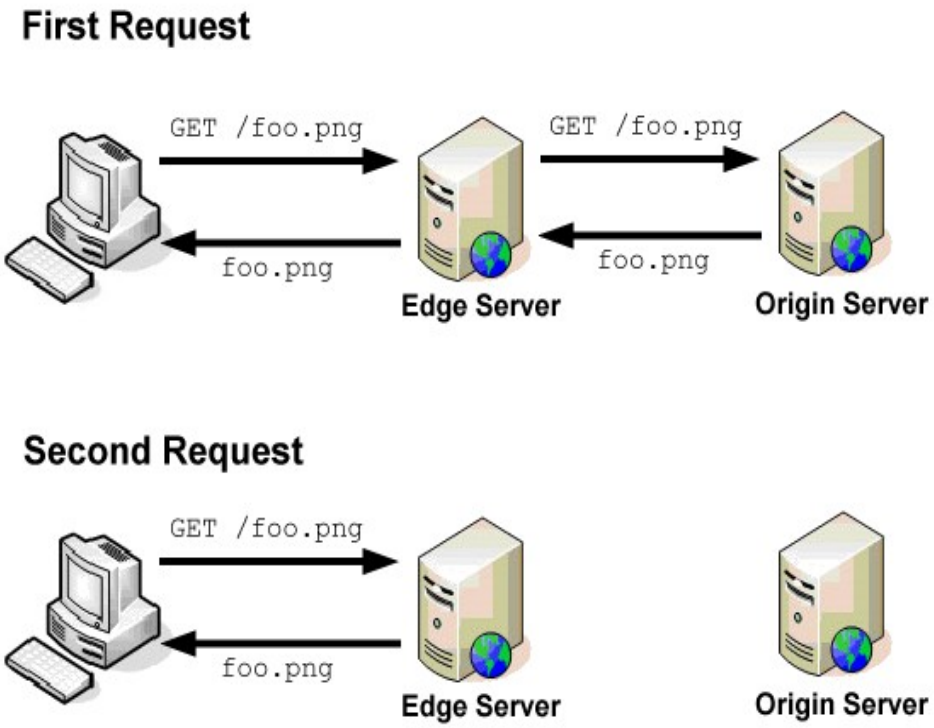
2.2.3 Cache in CDNs and HTTP streaming

Taking HTTP steaming for example, in the actual CDN operation, a client, say an HTTP streaming supportive web browser, first sends a DNS (Domain Name System) request to get the actual IP address of the server that can serve the requested content. "The CDN server handling DNS requests for this domain name looks at this request" [14] and returns the client the IP address of an edge server that is geographically close to the client. During this process, the CDN redirector responds the client with the IP of an edge server instead of the IP of the origin server. Afterwards, all subsequent communication starts between the chosen edge server and the client.

According to Zakas [14], the edge servers "are proxy caches that work in a manner similar to the browser caches". On receiving a request, the edge server first checks if the requested content is available. This is done by searching for a match for the requested "URL including query string" [14]. If the content is found to be in the cache of the edge server and the cached content has not expired, the content in the edge server is directly served to the client. In contrast, when a content is requested for the first time or the cache entry of this content has gone stale (expired), the edge server has to send a request to the origin server to fetch the latest version of this content. Based on the HTTP cache control header, the edge server then

decides if the content shall be stored in its cache. This process is illustrated in figure 4.

Based on this understanding of CDNs, the impact of a CDN on HTTP streaming can be simulated by putting up HTTP servers and proxy caches to build a network of a comparable structure with that of a CDN. By setting up the servers and proxies as well as introducing in network impairments, the behavior of an HTTP streaming system can be evaluated. Although the real CDN components can be more complex, fundamental impacts of caching and network impairments can be found out in a relatively simple testbed.



[14]

Figure 4: cache reaction on requests

Static content vs Dynamic content
In a CDN, not everything is stored in the cache of the edge servers. Edge servers (proxy cache servers) typically employ certain cache configurations to decide what types of content are allowed to be cached and how they shall be cached. It is fairly clear that caching dynamically generated contents in CDN can be a challenge and might not really benefit newly coming clients, since dynamic content is generated by the original server in real time and is dependent on the client and the server side source code. Static content in contrast, can be easily cached. Unlike dynamically generated contents, static contents such as images, audio and video, HTML files, JavaScript, Cascading Style Sheets (CSS), are not frequently changed. It is these contents that are independent of which client is requesting. As long as the content

is not altered at the server side, all clients would receive the same content. Since CDNs is used for web content caching and distribution, caching in the CDN edge servers "follow the cache rules set in the various HTTP headers" [15]. Based on the HTTP cach control headers and the caching configurations, the CDN edge servers performs the cache operation on static content and cachable dynamic content.

Cache chain
In an HTTP based CDN , the edge servers are essentially proxy servers that reside between the origin server and the clients. A CDN is structured, meaning it may employs a hierarchical topology or other topologies, as we mentioned in section 2.2.2. It is common that in a CDN there are a chain of proxy servers between the origin server and the client . When a piece of static content in the origin server is requested for the first time, the proxy servers along this chain as well as the browser cache decides whether or not to cache this requested content based on the cache control header found in the HTTP request message. The information within the cache control header is called cache response directives, and these directives applies to every cache, including the proxy and the browser, along the cache chain [15].

HTTP control header
The HTTP specification defines how the requested content can be cached and how the cached content shall be revalidated properly. More details about HTTP cache control headers is found in section 2.1.3.

Cache Policies
When HTTP streaming is operating in a Content Distribution Network, HTTP requests are first directed to the proxy cache, if the content is unavailable the request is sent to the original content server. When new or prior uncached content passes through the proxy cache, it needs to decide to cache the content based on the presence of the cache control headers in the HTTP request, and furthermore by the configured caching policy.

These configured caching policies decide what content can be cached and how it shall be cached. For example, one configuration option, known as refresh pattern, specifies whether or not and how long a certain type of files can stay in the cache. Other configurations include access control, storage space set up and so on.

Among the most important configuration options is the Cache Replacement Policy (CRP), which decides which content is purged to make space for newly items. The Least-Recently Used (LRU) is a very common CRP, and keeps recently requested items. Conversely, Least Frequently Used with Dynamic Aging (LFUDA) keeps the most popular items irrespective of their size, thus it is likely that larger items may prevent slightly less popular smaller items from being cached [16].

2.3 HTTP streaming

Having introduced the HTTP protocol and the HTTP supporting CDNs, we now move on to discuss HTTP streaming itself.

2.3.1 General discussion

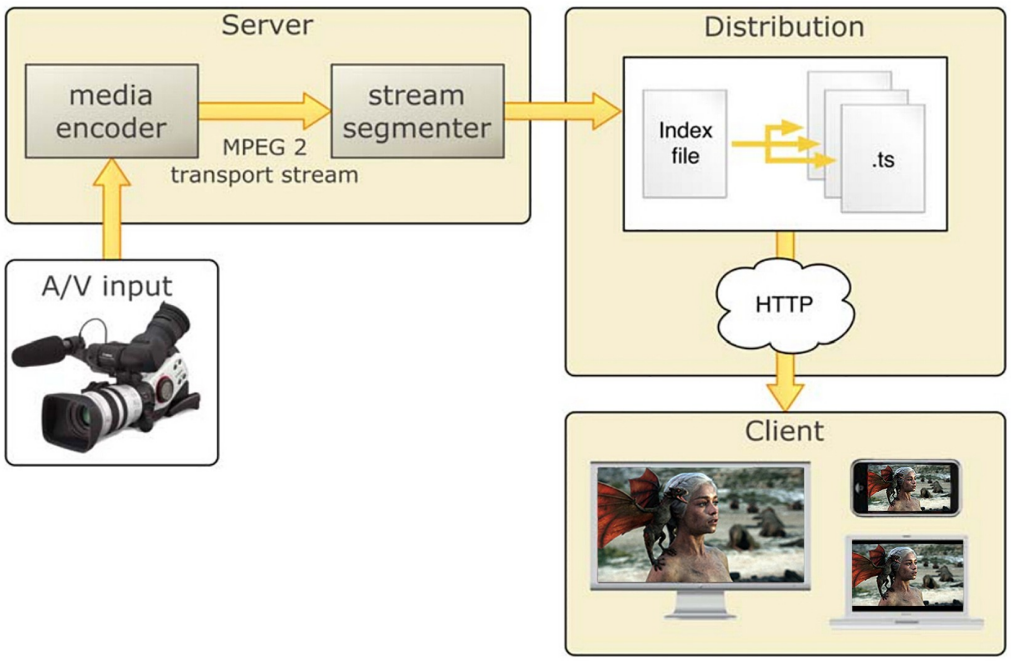
Video streaming in its simplest form can be realized by storing audio or video data file in a web server. By simply requesting the video or audio files through HTTP download, an HTTP client can pass the received file to a media player to play the stream. This type of streaming is generally referred to as HTTP streaming.
The benefits of HTTP streaming are obvious. Firstly, streaming based on HTTP can avoid issues arising from firewalls and NATs. Secondly, the widely deployed web servers and Content Distribution Networks (CDNs) can serve as the infrastructure for HTTP streaming, greatly saving the cost of system deployment [2]. In this sense HTTP streaming holds the advantage of great availability and scalability. One problem with HTTP streaming is that HTTP uses Transmission Control Protocol (TCP),which is traditionally considered not suitable for video streaming. However according to the view of Saamer Akhshabi [17], TCP itself has proven to be streaming friendly given that the video player client can perform effective rate adaptation to tackle the throughput variations caused by the congestion control of TCP. Kim and Ammar [18] also confirmed that "short-term transmission rate variation in HTTP/TCP can be smoothed out through buffering at the receiver side" [19]. Given these advantages, HTTP streaming has seen its rapid growth in streaming services on the Internet over the recent years.

2.3.2 HTTP streaming general architecture

HTTP streaming solutions in general, regardless of their exact implementations, share a similar architecture. Taking Apple's HTTP Live Streaming for example, it consists of "the server component, the distribution component and the client software" [3]. This system architecture is illustrated in figure 5.

As put by Andrew Fecheyr [3]," The server component is responsible for taking input streams of media and encoding them digitally, encapsulating them in a format suitable for delivery and preparing the encapsulated media for distribution"; "The distribution component consists of standard web servers. They are responsible for accepting client requests and delivering prepared media and associated resources to the client. For large scale distribution, edge networks or other content delivery networks can also be used"; "The client software is responsible for determining the appropriate media to request, downloading those resources, and then reassembling them so that the media can be presented to the user in a continuous stream".

Apple's HTTP Live Streaming employs the HTTP Adaptive Streaming (HAS) system design. According to Andrew Fecheyr [3], the server component encodes the audio and video input into MPEG-2 Transport Stream. The encoded stream



[3]

Figure 5: HTTP Live Streaming architecture

is then segmented into a series of short media files by a segmenter. At the same time an index file is also generated, which contains the information,such as URLs and encoding methods and encoding rates, of these media files. After the URL of the index file is published, the client software can download the index file and then request the corresponding media files of the wanted video stream.

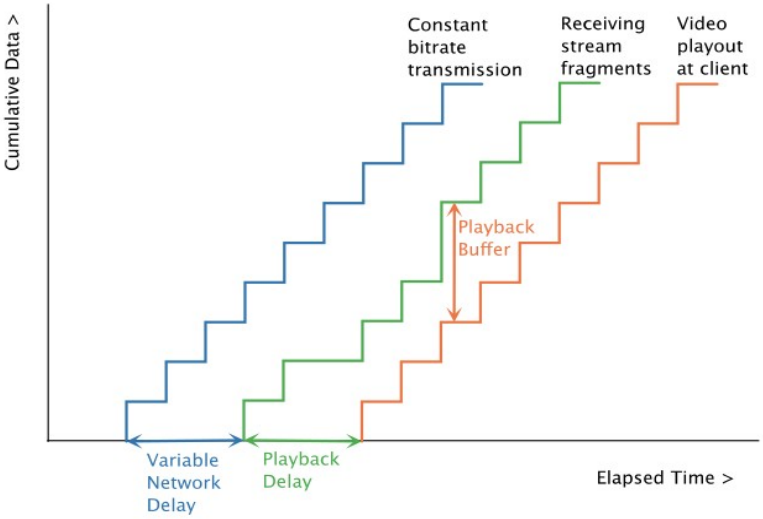
2.3.3 Live streaming vs On-demand streaming

Multimedia streaming can be put into two categories, namely, on-demand streaming and live streaming [3]. HTTP streaming, with no exception, falls into these two categories.

In HTTP streaming, on-demand streaming is usually straightforward. The entire media source is previously encoded and segmented into media files and stored in the server (the distribution component), before this resource being published online. On receiving the requests from a client, these media files are then served from the server.

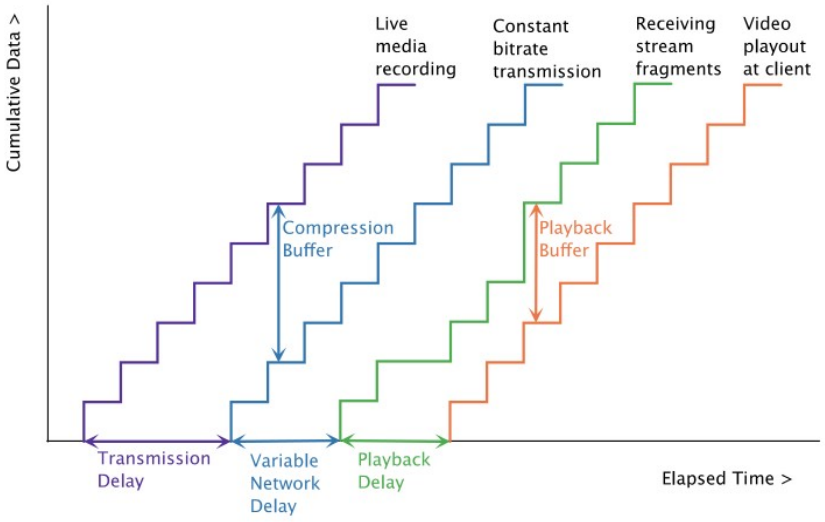
Live streaming, in contrast, requires the on-time encoding and delivery of the live feed from, say, a video camera. “The Live stream media is captured, compressed and transmitted on the fly” [3], which means a big amount of computing power is required in the live streaming server. In the case of HAS or Apple’s HTTP Live Streaming, the index file needs also be updated in real time so that the client who receives the live streaming can update the index file as every time it downloads

the current available media files (media segments). With the updated information provided by the newly acquired index file, the client is then able to request the newly generated media files from the server.



[3]

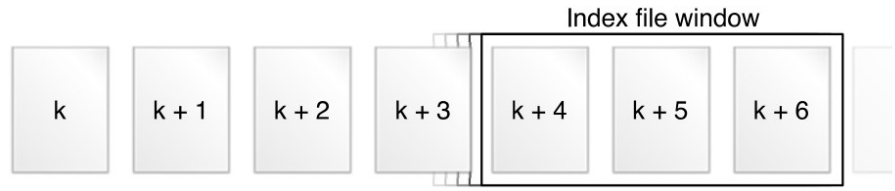
Figure 6: HTTP onDemand Streaming



[3]

Figure 7: HTTP live Streaming

Quoted from [3], figure 6 and figure 7 illustrate the difference between live streaming and on-demand streaming. Figure 6 shows the process of on-demand video streaming. The blue step line on the left represents the ideal constant rate transmission of the media files. In reality, due to network Jitter the transmitted packets would arrive at the client in an uneven fashion. To ensure the smooth playback of the video at



[3]

Figure 8: Index file of a live session, updated for every new segment

the client, a playback buffer is typically employed to delay the actual playback for a short period of time.

For the same reason, live streaming uses the same playback buffer just as on-demand streaming does. In addition, live streaming also uses a compression buffer at its distribution component. Since the dynamic encoding rate of the live stream varies with the video motion intensity, the server component that performs the capturing and encoding of the live feed can only generate the encoded media files at an uneven rate. This means with the transmission rate between the server component and the distribution component being constant, the distribution server receives the newly generated media files and the updated index file at an uneven rate. The compression buffer is used to buffer up the received data, creating a margin of delay so that the streaming process will not be interrupted at the distribution server.

Another difference between on-demand streaming and live streaming is the index file. As demonstrated by figure 8, the index file for live streaming must be updated as the live feed is encoded into new media files . As put by Fecheyr [3] "the updated index file includes the new media files and older files are typically removed. Whenever a new segment is ready, the index file is updated to include the newest segment and to remove the oldest one". In this sense, the index file provides a sliding window play-list, containing the latest available media files. In on-demand streaming, the index file is, however,only generated once and never updated. Upon its generation it contains information of all media files of the complete stream.

2.3.4 HTTP streaming evolution

Although different HTTP streaming solutions share the similar architecture, the HTTP streaming technology has been evolving over time. Popular solutions of HTTP streaming over the years are summarized as follow:

- **Progressive download**
  The early form of HTTP streaming performs video streaming by a method called progressive download, where the client simply establishes a TCP connection with the server and downloads the multimedia data progressively with best effort, until all the data is received [1]. In this streaming system, all clients requesting the same video content have to download the non-distinctive video

stream that is of the same encoding, which means neither the network condition during the downloading nor the client platform and the decoding capability is taken into consideration. In a nutshell, there is no rate adaptation in progressive download. Apart from the lack of bit rate adaptation, progressive download might also cause unnecessary network traffic, since the downloading process might greatly exceeds the playback process due to the lack of flow control. When a user decides to stop watching the content and end the streaming, data that has been fetched in advance would be wasted [2]. Being able to serve on-demand streaming, progressive download, however, is not suitable for live streaming and mobile usage. Questioned by all these drawbacks, progressive download is no longer a popular HTTP streaming solution.

– **HTTP Adaptive Streaming (HAS)**
Acknowledged as a new paradigm of HTTP streaming, Adaptive Streaming Over HTTP (HAS) has drawn the spotlight in the recent years. The basic idea of HAS involves encoding the media stream into different versions with each one processing a different resolution and bit rate. Each version of the encoded media stream is also divided into small chunks (segments), forming a unique set of media segments called a representation. As the streaming proceeds, the client can dynamically switch to a certain representation and download the needed chunk so that the bit rate of the chunk to be downloaded can match the real time network bandwidth. By providing different versions of the same video content, the HTTP server can serve the clients differently based on the connection qualities of these clients. It is worth mentioning that the server simply keeps all versions of the media stream (representations) as well as the corresponding manifest file (index file) in its storage. It is the client who would choose the wanted encoding rate of a chunk, based on its current network condition and the information given by the manifest.
This paradigm has been used in the design of several HTTP streaming services, including Adobe Flash Video [20], Netflix[5], Microsoft smooth streaming [21] and Apple's HTTP Live Streaming [3]. Their implementations and performances in detail have been discussed by Saamer Akhshabi [1].
Despite the popularity of HAS, there are however, some problems with these existing implementations. For example, with proprietary solutions mentioned above, a user is typically required to install certain client side platform and applications so that the service can be delivered. This not only creates inconveniences for the users, but also complicates the distribution of new application upgrades. Apart from Adobe Flash Video[6], none of these solutions are open source and thus can not provide enough value for the developer community.

– **MPEG-DASH**
Despite the fact that multiple streaming services, as mentioned above, are using the HAS technology, they are all standalone and proprietary solutions and

[5] http://www.netflix.com
[6] http://www.adobe.com/devnet/video/articles/osmf_overview.html

lack interoperability. Due to the promising future of HAS, the Moving Picture Experts Group (MPEG) has drawn up a standard called Dynamic Adaptive Streaming over HTTP, also known as MPEG-DASH, in the hope of building a foundation for the interoperability among different HAS solutions. With the primary specifications defined in ISO/IEC 23009-1, the MPEG-DASH standard was published in April 2012. [4]

Based on the MPEG-DASH standard, multiple implementations of DASH have been proposed. One particular implementation known as DASH-JS[7] has enabled HTML5 compatible Web browsers to acquire DASH functionalities without having to install any third party plug-ins. Providing the basic client functionalities that comply with the MPEG-DASH standard, DASH-JS serves as a reusable library that can be easily extended for desktop and mobile usages alike. Both the on-demand streaming client and the live streaming client can be implemented by extending the original DASH-JS. DASH-JS has enabled our research team to do a in depth study on DASH, allowing us to conduct various tests and performance analyses. Based on DASH-JS, a new adaptation algorithm has also been implemented by our team, aiming to tackle the challenges posed by proxy caches in DASH streaming. More details of these challenges and our solution together with the implementations are discussed in latter chapters.

2.4 Summary of background study

The HTTP protocol defines the basic request and response behavior of HTTP streaming. Important features of HTTP services such as persistent connection, pipelining and cache control are introduced in this chapter. These features, which can greatly influence the performance of HTTP streaming, shall be taken into consideration when an HTTP streaming system is designed. Due to the fact that HTTP streaming typically utilizes the widely deployed CDNs that support HTTP traffic, the comprehension of CDNs is also valuable for designing an effective HTTP streaming system.

From our study we learn that under the control of the HTTP cache-control directives, caching services are provided by the edge servers of the CDNs. It is clear that exsiting CDNs together with the widely deployed HTTP web servers, provide the infrastructures for HTTP streaming. Since the popular HTTP streaming solutions, such as Dynamic Adaptive Streaming over HTTP (DASH), do not require any non-standard HTTP content delivery, the existing infrastructures can serve HTTP streaming effectively.

With the understanding of the HTTP protocol and its features as well as the cache mechanism of CDNs, the popular HTTP streaming technologies are discussed. Among these technologies, MPEG-DASH has been acknowledged as the most promising one, which will be discussed at length in chapter 3.

[7] http://www-itec.uni-klu.ac.at/dash/?page_id=746

3 Dynamic Adaptive Streaming over HTTP

In this chapter, DASH, as the HTTP streaming technology we study, is discussed at length. First the mechanism of a DASH system is illustrated and explained. Then the MPEG DASH standard is described. In the end of the section, different solutions of the DASH client implementation are presented and discussed.

3.1 DASH system structure

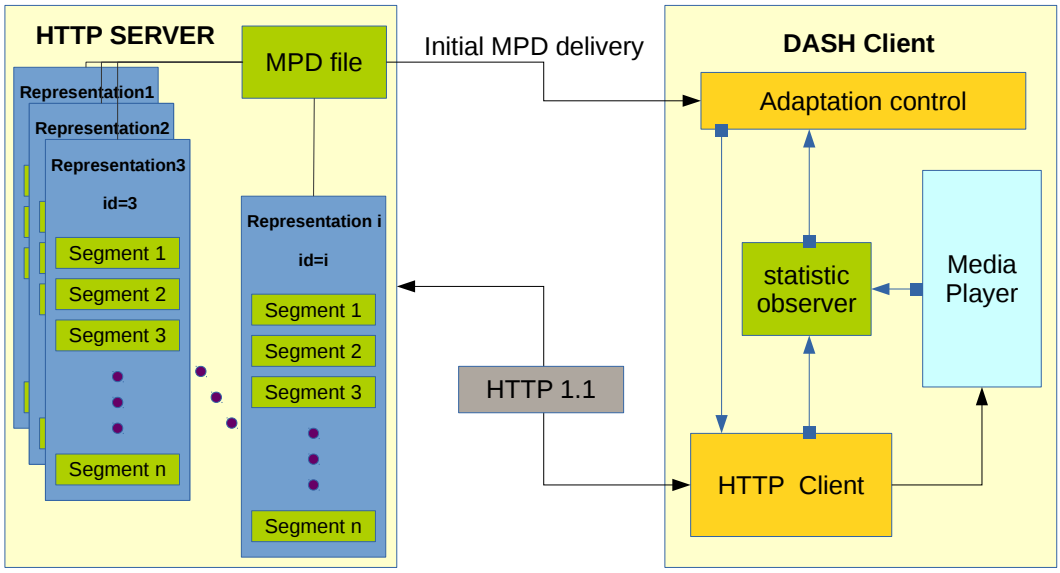


Figure 9: DASH system structure

As shown in figure 9 [4], a typical Dash system includes an HTTP server that provides the video content and a DASH client that can both perform HTTP communications and play the DASH multimedia content. During the streaming process, the DASH client accesses the DASH content by sending HTTP requests and downloading from the HTTP server. Progressively requesting and receiving the DASH content, the DASH client builds up the internal buffer of the Media Player and starts to play the content when enough data is available. As put by Stockhammer [2], during the streaming process, it is the client who takes full control of the operations, including " the on-time request and smooth playout of the sequence of segments, potentially adjusting bitrates or other attributes" [2].

At the server side an original video stream is encoded into multiple representations. Each representation is an independent stream containing the same video content as the original but possessing a different resolution or average bit rate. Each of the representations is then packed into media segments of roughly the same length in seconds. And all representations are packed into the same number of segments

so that segments within different representations are roughly aligned in playback time. “Each media segment is assigned a unique URL (possibly with byte range), an index, and explicit or implicit start time and duration. Each media segment contains at least one stream access point, which is a random access or witch-to point in the media stream where decoding can start using only data from that point forward. ” [4] This means when playing the video stream each period of the video is available in multiple segments within different representations, allowing the client to switch among different resolutions and bit rates as the streaming proceeds.

When multiple representations are generated from the original video stream, a corresponding Media Presentation Description (MPD) file, which is typically in XML format, is also generated to facilitate a dash client to “establish an adaptive dynamic streaming over HTTP” [22].The MPD file contains vital information of all representations, including the corresponding decoding methods, resolutions and bit rates, etc. It also contains the URL of each segment of a representation.

When a dash client such as a dash enabled browser has downloaded the MPD file of a video stream, it parses the MPD and then fetches video segments progressively. Each time before the client requests a segment, it goes through its rate adaptation algorithm to choose a representation that suits the user preference and the current network conditions best. When the representation is chosen for this segment, the client requests the segment based on the URL (possibly with byte range) provided by the MPD. By changing the adaptation algorithm, the rate adaptation behavior of the dash client can be altered.

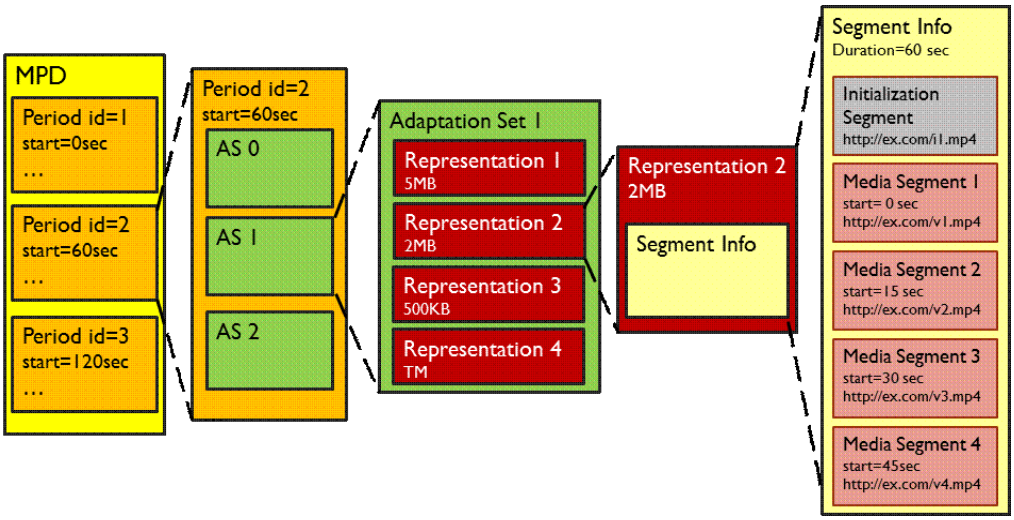
Figure 9 also illustrate the details of the streaming process of DASH. First a DASH client issues the download of the MPD file of the video stream, through the HTTP client module. The MPD is then parsed and the parsed information is fed into the adaptation control module. Based on the adaptation algorithm, the adaptation control module takes the collected data from the statistic observer and then makes a decision on representation selection for the segment to be downloaded. The decision is then passed to the HTTP client and the HTTP client starts to request the selected segment. On receiving the segment, the HTTP client feeds the segment to the media player. At the same time, the statistic observer collects information such as estimated bandwidth and player buffer level, from the HTTP client and the media player respectively. The collected data is again fed to the adaptation control module, so that the DASH client can start another round of rate adaptation (representation selection) and segment download.

During the process, the DASH client can also specify the various header directives of the HTTP requests, under the HTTP/1.1 specifications. In doing so, the DASH client can affect, say, the caching behavior of the CDN servers, and in turn influencing the cache hit ratio of incoming requests from other clients who request the same content.

3.2 MPEG DASH standard

The MPEG-DASH specification defines none of the MPD delivery method, the media-encoding formats and the client side implementation [23]. Instead, only the MPD and the segment formats are defined. By standardizing the data representation and the MPD, DASH aims to provide an HTTP streaming enabler without constraining the client design.

3.2.1 MPD format



[4]

Figure 10: The structure of MPD

As stated by Sodagar [4], "Dynamic HTTP streaming requires various bitrate alternatives of the multimedia content to be available at the server. In addition, the multimedia content may consist of several media components (e.g. audio, video, text), each of which may have different characteristics. In MPEG-DASH, these characteristics are described by MPD which is an XML document ". By generating the MPD in the XML (Extensible Markup Language) format, a hierarchical data structure can be conveniently constructed, which in turn benefits the parsing process at the client side.

Figure 10 shows the hierarchical structure of the MPD, defined by MPEG-DASH. An MPD file contains one or more periods, which forms the top level of the data hierarchy. Each period contains an interval of the multimedia content, with a starting time and playback duration. Within a period, there can be one or more adaptation sets, with each adaptation set containing a media component of the DASH content.

According to Sodagar [4], a possible composition of the adaptation sets could be that one adaptation set contains the video component of the multimedia content,

while another adaptation set contains the audio component of the multimedia content.

Each adaptation set typically contains multiple representations, with each representation being “an encoded alternative of the same media component,varying from other representations by bitrate, resolution, number of channels or other characteristics.” [4].

Each representation consists of multiple segments, which contains the actual multimedia payload. These segments are divided chunks of the original media stream in temporal sequence. Each segment is also assigned a relative URL. Together with the base URL provided in the MPD, the DASH client can construct the absolute URLs of all segments. By constructing HTTP GET messages based on the absolute URLs, these segments can be downloaded through HTTP.

With the MPD format well defined in the DASH specification, the parsing of the MPD file is left to the client side functionality. A sample MPD file is presented in the appendix A.2

3.2.2 Segment format

According to Sodagar [4], in DASH, “the multimedia content can be accessed as a collection of segments ”. As we discussed earlier, these segments are the encoded and divided pieces of the original multimedia stream.

The first segment shall serve as an initialization segment , from which the DASH client can extract the required information to initiate the media decoder. “The media stream then is divided to one or multiple consecutive media segments. Each media segment is assigned a unique URL (possibly with byte range), an index, and explicit or implicit start time and duration. Each media segment contains at least one Stream Access Point (SAP), which is a random access or “switch-to” point in the media stream where decoding can start using only data from that point forward. To enable downloading segments in multiple parts, the specification defines a method of signaling subsegments using a segment index box. This box describes subsegments and stream access points in the segment by signaling their durations and byte offsets. The DASH client can use the indexing information to request subsegments using partial HTTP GETS.” [4].

3.2.3 Live vs on demand

DASH supports both live and on-demand content streaming. In the case of on demand content, the MPD file shall provide the start time and duration of each segment. In the case of live, the MPD shall contain the segment availability information in the form of the available start time and available end time. Other information such as the approximate media start time, the fixed or variable duration of segments shall also be provided by the MPD to support live streaming. In short, for both live and on-demond streaming, the MPD file shall provide necessary information in its hierarchical structure to support the streaming process. Details and examples of

the possible implementations for Dash live streaming and on-demand streaming are presented by Stockhammer [2].

3.2.4 Copyright protection

Liu [24] addressed the necessity for digital rights protection when copyrighted content is distributed over the Internet. A Digital Rights Management (DRM) system is usually used for such protection. A consumer who uses the DRM service typically pays for a digital license, which is a digital data file. Based on the decryption key and other information provided by the digital license, the consumer is able to access the encrypted content from a content distributor who uses this DRM system.

Since DASH streams digital content over the Internet, it is important for the system to support popular DRM schemes. According to Sodagar [4], MPEG-DASH allows an adaptive set to " use a content protection descriptor to describe the supported DRM scheme."
"In conjunction with the MPEG-DASH standardization, MPEG is also developing a common encryption standard, ISO/IEC 23001-7, which defines signaling of a common encryption scheme of media content." [4] Based on the signaling messages and the content protection descriptor in the MPD, a DASH client can work with a supported DRM system and receive the encrypted stream from the server.

3.2.5 Special features

MPEG-DASH has also defined a collection of various special features to enrich the functionalities of DASH. These features include advertisement insertion, compact manifest, fragmented manifest, segment with variable durations, multiple base URLs and so on. Details can be found in [4].

3.3 DASH client solutions

As mentioned earlier, MPEG DASH defines only the MPD and segment formats of the DASH content and leaves the intelligence of DASH to the client side. After the media stream is encoded and packed into MPEG DASH standard segments, the web server merely keeps the segments and the MPD file in its storage, serving these segments through HTTP when receiving requests. The intelligent rate adaptation of the DASH system resides really in the DASH client. In this sense, how the DASH client is designed and implemented is vital to the effectiveness of a DASH system.

As shown in figure 9, a DASH client is composed of three basic functionality modules. The first is an HTTP client which enables HTTP communication. The second is an adaptation control module that performs rate adaptation. The third one is the media player that renders and plays the downloaded DASH content. To implement a DASH client, there can be multiple solutions.

The first solution is to build a standalone DASH client which possesses all three

functionalities. For example the latest VLC media player[8], which supports DASH, falls into this category. This solution is straightforward but costly. Besides, accessing a video URL through a media player is not a conventional user behavior, since people are more comfortable to browse the Internet with a web browser and find interesting videos to watch as they search around.

The second solution is a browser plug-in. This solution benefits from the browser functionality and spares the trouble of building internal HTTP functionalities. Parsing the XML based MPD file and the HTTP messages can also be left to the browser. However this solution can give rise to compatibility issues since plug-ins are usually browser dependent. Invariably, a plug-in needs be upgraded as the browser upgrades, which causes troubles for both developers and users.

The third solution is a JavaScript based DASH client. This solution is better than the previous two in several ways. Firstly, JavaScript is interpreted by the browser and operates at run time. This completely saves users the trouble of installing the client program. Secondly, due to the fact that the Javascript source code is kept in the web server, a JavaScript based client can be easily maintained and upgraded by developers, saving great software-distribution troubles. Last but not the least, a JavaScript based client needs only to focus on adaptation functionalities, thanks to the HTML 5 specification, which has pushed browsers to support multimedia content rendering. The major enablers of the JavaScript based DASH client is HTML 5 and its Media Source Extensions (MSE). More details can be found in section 1.1, in the History part.

3.4 Summary of MPEG-DASH

With the MPEG-DASH standard defining the format of MPD files and DASH segments, different implementations of DASH clients are able to share the same multimedia stream resources that comply with the MPEG-DASH standard. Likewise, with the standardized MPD and segment formats, different DASH servers, are also able to serve DASH clients of different implementations.

In this section, the general system architecture of DASH is presented and explained first. The DASH streaming process is then explained based on the understanding of the DASH architecture. After that, various aspects of the MPEG-DASH standard itself are discussed in detail. Lastly, several DASH client solutions are presented and compared, in order to demonstrate how the client side of a DASH system can be freely implemented. Based on the JavaScript library DASH-JS [7], our DASH client implementation employs the third solution presented in section 3.3. Our particular DASH implementation is presented in latter chapters.

[8] http://www.videolan.org/index.html

4 Challenges of DASH and our contribution

As DASH is still a relatively new technology, a big amount of study work has been conducted over the recent years, in order to solve some challenging problems facing DASH. In this chapter, these challenges are addressed and the important studies made on HTTP streaming and DASH are summarized as well. To study and tackle some of the major challenges, we build our DASH client solution based on DASH-JS [7]. In the latter part of this section, the contribution of our research is summarized briefly and a modeling and short recap of DASH-JS is also presented.

4.1 Challenges and related studies

4.1.1 Segment size

As HTTP streaming and DASH specifically, requires the media content to be segmented, determining the size of the media segment has been one of the challenging tasks.

Different segment sizes shall be chosen for different uses. For example, in live streaming small segments, such as 1 second segments, are favored since users are expecting very short lag between the received media and the real live event. In this situation, the use of 10s segments means a lag of at least 10 seconds since the segment must be encoded and generated before the user client is able to download it. In contrast, if the content being streamed is say, a Hollywood movie , then big segment size is preferred since no user would enjoy the quality of the picture being constantly switched. Bigger segment invariably means bigger receive buffer in the client, reducing the possibility of buffer underflow and playback interruption while the user is watching.

Segment size can also affect the adaptation behavior of DASH. For example, Liu *et al.* [25] addressed the problem of the rate adaptation effectiveness depending on the segment duration. As he claimed, a typical rate adaptation method evaluates the network capacity every time a segment is fetched, and it performs rate adaption based on the evaluated network capacity. Thus segment with long duration may result in a slow adaptation since the time required to fetch such a segment would be relatively long. Likewise, when the segment duration is short, the rate adaptation can be more agile. However, according to his claims, due to the fact that TCP congestion control causes the instantaneous rate of data transmission to vary over time and form a sawtooth shape, network capacity evaluation based on short time observation can be inaccurate. In this sense, too short a segment size can cause inaccurate estimation of the network capacity, resulting in a non-optimum rate adaptation. “In order to provide accurate and fast rate adaptation ” [25] for HTTP adaptive streaming, Liu *et al.* [25] proposed a segment-duration-based rate adaptation method.

4.1.2 Rate adaptation algorithms

Due to the request-response nature of HTTP streaming and DASH, the DASH clients, instead of the servers, have to perform rate adaption during the streaming process. Designing the client side adaptation algorithm proves to be another challenge for HTTP streaming and DASH .

There are plenty of researches made on the rate adaptation algorithms. For instance, Akhshabi [17] made a study that evaluates and compares the rate adaptation algorithms of several commercially used video streaming services, which adopt the HTTP Adaptive Streaming (HAS) technology. Microsoft Smooth streaming, Netflix and the open source Adobe OSMF were evaluated in the study. Apart from basic performance tests, a competition scenario was also devised in this study to show the behavior of adaptive streaming under more realistic situations. The study concluded that the HAS technology was at an early stage and more advanced adaptation algorithms were in demand.

In 2012, Akhshabi [26] published another study that specifically focused on the issues of HAS players under competition situations. Several factors, including the on-off period of HTTP downloading, that could affect the adaptation stability and bandwidth-sharing fairness of competing clients were examined and analyzed. Akhshabi concluded that a HAS client could fail to accurately estimate the available bandwidth when its HTTP download is in the OFF periods, during which there is neither new HTTP request issued nor data transfer. To be more specific, a competing client that downloads data presently (HTTP on-period) can seize the bandwidth possessed by another client that is currently at its OFF-period, causing the OFF-period client to over estimate its current available bandwidth. Providing the evidence of such an instability issue in rate adaptation under competition scenarios, Akhshabi however has not devise a solution yet.

The study on rate adaptation was also conducted by Liu [27], who proposed a receiver-driven rate adaptation method for HAS. In this adaptation method, a smoothed HTTP throughput measurement was suggested against the the use of instantaneous TCP transmission rate as the base for rate adaptation, to help evaluate the end-to-end network bandwidth capacity more accurately.

Compared with Liu [27], Li's Probe and Adapt [28, 29] algorithm proposed measuring media transfer capacity of the network by probing the limit of TCP transmission, meaning incrementally switching to higher media bit rate as long as the TCP throughput continues to climb. When TCP throughput drops, the algorithm backs off to choose lower media bit rates, avoiding TCP congestion. Instead of sending probe packets to the network, this algorithm merely " probes " the TCP transmission limit at the client side, effectively avoiding any network capacity waste and TCP congestion without creating any extra network traffic.

Jiang *et al.* [30] summarized the challenges of designing an adaptation algorithm for HAS. The study shows that in order to balance among the goals of clients-competition fairness, adaptation efficiency and bit rate switching stability, three deciding factors must be taken into consideration, which include chunk download scheduling, bit-rate selection and bandwidth estimation. By identifying the challenges posed by the characteristics of TCP as well as different usage scenarios, an optimized algorithm called Festive was designed, with the three deciding factors taken into account.

4.1.3 DASH specific issues

Studies specifically on MPEG-DASH have also been made recently, in the attempt to identify the important factors of DASH system design.

For instance, Stockhammer [2] made a study on the standards and design principles of DASH. General discussions were made to explain the origin and goals of DASH. The study also presented the DASH specifications defined by MPEG and 3GPP . Then it illustrated the components design and system structure of DASH. Deployment details and service examples were also demonstrated in this study.

4.1.4 Challenges of CDN Caches

In HTTP streaming and DASH, the client side rate adaptation algorithm defines the congestion control. A typical DASH client has to work with two control loops: 1) the underlying TCP congestion control that delivers the requested media segment, 2) requesting the appropriate media representation of a specific segment length, as to not cause the receiver buffer to underflow. The DASH client also needs to pre-buffer enough media content to overcome short-term connectivity issues (e.g., in wireless environments). Meanwhile the DASH client needs to also provide consistent media quality, i.e., not fluctuate between different media representations too often, because switching media quality too often or by too much affects the user experience [31]. More importantly, for a DASH congestion control algorithm to be safe to deploy, it should be able to work with the existing network infrastructure (e.g., caches, CDNs, etc). Currently, content delivery networks (CDNs) are extensively used for delivering media content for both video on demand and live streaming services. CDNs store content at various locations globally, and direct client requests to the appropriate server in an attempt to minimize delay in acquiring the media content, and also to avoid overloading the media servers (by load balancing). However, due to the availability of multiple representations of the same video, the CDNs may behave like caches, i.e., add and remove content based on various eviction policies. Consequently, the cache may contain multiple representation of the same content, but more importantly these representations may be incomplete. This creates a new challenge for the DASH congestion control, because the DASH client receives media data either from a nearby cache or from a cache further away, hence the client observes varying RTT estimates. Typically, losses and variation in RTT causes the underlying TCP to become unstable and reduce the sending rate, finally, causing the DASH congestion control algorithm to inadvertently switch representations at short time scales [22],

which would affect the user-experience.

The vital challenge posed by CDNs proxy caches on DASH is addressed by Liu *et al.* [19]. As Liu put, "Due to the fact that proxy-driven proxy cache management and the client-driven streaming solution of DASH are two independent processes, some difficulties and challenges arise in media data management at the proxy cache and rate adaptation at the DASH client". Liu, in his research, mentioned how a incompletely cached DASH content can affect the performance of a DASH client. According to him, "a DASH client may dynamically request different segments from different representations" of the media clip. As a result "segments from multiple representations may be cached by a caching proxy, and the cached segments for each cached representation are likely to form an incomplete representation". "A consequence of caching incomplete representation is variation in Segment Fetch Time (SFT) observed by the client". The variation in SFT, which is caused by discontinuous cache hits, can be interpreted by the client as congestion or throughput changes, which in turn causes the client rate adaptation algorithm to switch the representation level (media quality) frequently. Put in another way, due to the incomplete cached representations at the proxy caches, the client can inaccurate estimate the network throughput and performs premature representation switching frequently, resulting in media quality fluctuation and poor watching experience. "Furthermore, requesting a high-bitrate encoded segment upon observing a short Segment Fetch Time that is due to the fetching of cached segments may result in buffer draining".

In response to these issues, Liu proposed a DASH implementation where the DASH client co-operates with the proxy caches to optimize pre-fetching media content. However, there is explicit signaling between the caches and client, which are meant as a proposal to extend the DASH protocol. The algorithm is relatively complex and requires the caches to be DASH-aware. This means that the widely deployed commercial proxies have to implement the extension protocols and then redeployed.

Muller *et al.* [32] also studied the effects of proxy caches upon the rate adaption of DASH clients. Under the test scenario in this study, two clients request the same DASH content from a common proxy cache, which is connected to the HTTP server. Both clients have their dedicated connections to the proxy and compete for the bottleneck link from the proxy to the server. An adaptation method involving exponential back-off and network probe was proposed, in order to solve the problem of frequent quality switching. Unlike our proposal, this adaptation method uses probe detection to avoid premature bit rate switching caused by a cache hit whilst our proposal uses a buffer monitor approach to achieve the same goal.

Similarly, Lee *et al.* [33] also tried to solve the bit rate oscillations caused by proxy caches. This proposal, instead of focusing on the client side algorithm, introduced "an approach to create a video-aware cache server" in the attempt to "shape the traffic from the cache" .

Miller *et al.* [34] proposed an algorithm that is similar to ours, i.e., it is also based on measuring buffer level and average throughput. However, it varies the segment sizes depending on the use-case. It uses short segments for serving live content, and in this case the adaptation is very agile and switches often. Meanwhile uses long segment sizes for streaming stored content. Conversely, our proposed algorithm provides consistent media quality even when using small segment sizes.

Apart from the studies on CDN challenges, innovative studies on the developing Content Centric Network (CCN) are also made, in the hope of improving the network-wide performance of DASH. For example, Liu*et al.* [35] did an analysis on CCN caching and find out DASH can effectively benefit in a CCN. However, CCN is currently not widely available and stays at an experimental stage. Moreover, for DASH to utilize CCN the DASH standard has to be extended, since in CCN the content objects are addressed using different schemes than the one used by traditional HTTP objects.

4.2 Our Contribution

The main contribution of this thesis work is a DASH congestion control algorithm that is performant in content delivery networks (CDNs), i.e., in the presence of multiple levels of caches. Aiming to solve the same challenge addressed by Liu [19], our proposed algorithm, however, does not require any specific interaction with the caches, for example to indicate their presence or to exchange a map of cached content. Compared with the proposal of Liu [19], our proposal is cache-friendly and does not require any extension to the DASH protocol. Through our test we observe a much smoother playout of media, with consistent media quality.

Rainer *et al.* [7] proposed a web integration of MPEG-DASH using JavaScript and Media Source Extension[9]. This solution is open source and is named as DASH-JS [7]. Based on this work, we implement our adaptation algorithm, Gearbox. This algorithm introduced a new mechanism that bind the adaptation behavior with the receive buffer level of a DASH client. Moreover, this proposed solution provides a model that can be easily configured in order to suit different preferences and user scenarios. Details of the algorithm will be presented and explained in chapter 5.

4.3 DASH-JS and Modeling

Congestion control of a DASH client involves two decisions. One is to decide when to issue the video segment download, the other is to decide the representation level of the segment to download. DASH-JS [7] defines the default congestion control, on which our implementation builds. For convenience purpose we rephrase below the original implementation of DASH-JS and its system model. Formulas in this section are extracted from [7] and the DASH-JS source code [36].

[9] http://w3c.github.io/media-source/

For a certain DASH content, we denote the set of representations enlisted in its MPD as $\mathbf{R}$. Within the MPD file each representation in $\mathbf{R}$ is given an identity number and we denote by R_i the representation with the id of i.We denote the average bit rate of $R_i \in \mathbf{R}$ by B_i, which is a datum presented in the MPD. Each of the representations consists of m segments and we denote the segment to fetch by a DASH client by S_n, where $0 < n \leq m$.

To decide the representation level of a segment,DASH-JS takes these following operations: First, the network throughput must be measured and evaluated. Every time the DASH-JS client issues an HTTP request, the system time is recorded, denoted by T_{start}. On receiving the response, the system time is recorded again and denoted by T_{end}. Denoting the length in Byte of the response by K, the receiving rate of the last response is then measured as $B_l = \frac{K}{(T_{end} - T_{start})}$. DASH-JS estimate the current network throughput B_c by calculating an weighted average of B_l and B_c, with the initial value of B_c set to zero. With the weights set to W_1 and W_2, we have

$$B_c = \frac{W_1 B_c + W_2 B_l}{W_1 + W_2} \tag{1}$$

B_c is updated every time a response is received by the client. By adjusting the weights, the throughput estimation can be numbed or sharpened. Secondly, each time before sending a segment request, the DASH-JS client goes through its adaptation algorithm to decide the representation for the next segment. That is, to decide R_i for S_n. To do this, DASH-JS checks each $R_i \in \mathbf{R}$ and compares its corresponding B_i with B_c. R_i is then selected based on the comparison results. On deciding R_i for S_n, DASH-Js send the HTTP request based on the URL of S_n under representation R_i.

To decide when to issue the video segment download, DASH-JS takes these following operations: On starting DASH-JS client downloads the video segments (I) consecutively and in chronological order (II)in serial, meaning no subsequent request will be sent before the last response is received. The fetched segments are stored in a receive buffer, originally referred to as overlay buffer in paper [7], whose size in seconds is denoted as η. We denote the current level of this receive buffer as η_c. As long as η_c reaches a low water mark, denoted as η_{low}, DASH-Js triggers the video playback. Then the buffer is drained periodically as the video plays. At the meantime, the DASH-Js client continues to fill up the buffer by downloading subsequent segments. Once the buffer is full, DASH-JS pauses the download. When the buffer is drained to reach a high water mark denoted as η_{high}, the download will be triggered again.

In the original DASH-JS, both η_{low} and η_{high} are set to be the same which gives $0 < \eta_{low} = \eta_{high} < \eta$. The representation selection, which is decided by the adaptation algorithm, is solely based on the comparison of B_i and B_c.

There is a bug in the original implementation.After the receive buffer is filled to reach η_{low}, DASH-JS drains it by one segment at a time to feed the internal Source-

Buffer [37] of media source extension (MSE) [38]. A hidden bug in this feeding loop can possibly cause a jamming playback, which results in failures in our test of the original algorithm. This will be emphasized where needed in the evaluation part of our paper.

4.4 Summary of Challenges and our Contribution

With various chanllenging problems of DASH studied and solved by researchers, one vital problem still remaining is how the rate adaptation of DASH can effectively make use of the existing CDNs without being affected by the incomplete cached representations. Constructing a new signaling mechanism between the DASH client and CDN caches, as proposed by Liu [19], can effectively solve the problem but it is very costly in terms of deployment. Aiming to solve this incomplete-cached-representations problem addressed by Liu [19], our proposal ensures that a DASH client can provide satisfactory performance and user experience without having to exchange extra signaling messages with CDN caches. In this way, the intelligence of a DASH system is solely located at the client side and DASH can be smoothly deployed without having to alter the conventional communication protocols used by the existing network infrastructures.

5 Implementation

In this chapter details of the proposed algorithm are presented. Its design philosophies and working mechanisms are explained. The modeling of the Gearbox operation is also presented to explain and facilitate the parameter configuration of the algorithm.

5.1 Gearbox Algorithm

5.1.1 Design philosophies

We made two major changes on the original DASH-JS to improve the performance. One change concerns the receive buffer watermarks, which are set to trigger playback and buffer refilling. The other concerns the adaptation algorithm, which decides the representation selection (rate adaptation) behavior.

We lower η_{low} to make sure the playback starts quicker. We raise η_{high} to make downloading more active so that a higher buffer level can be maintained, leaving the DASH client bigger margin to act upon possible network fluctuations. The original DASH-JS set the buffer size η to be 30 s and the sole water mark is set to be 20 s. In our implementation, we set η to 40 s and we set η_{low} and η_{high} to 10 s and 35 s respectively.

To imporve the performance of the original DASH-JS, we put our main focus on altering the adaptation algorithm. Since the original algorithm performs rate adaptation solely based on the observed network bandwidth, it can result in poor performance under multiple scenarios, including the scenario of incompletely cached DASH content in a proxy cache [22] and scenarios where network bandwidth fluctuates badly. The rate adaptation challenges posed by these scenarios are detailed in section 4.1.4. Performance problems of the original algorithm under these two scenarios mainly involves premature decisions of representation switching and high possibility of buffer underflow. Both these problems, namely frequent representation switching and high buffer underflow counts, can affect user experience badly.

To solve the these problems, we have introduced a new mechanism to the adaptation algorithm so that buffer level is taken into consideration when performing representation switching. The aim is to optimize the user experience in the mentioned scenarios. There are three major goals to achieve: i) reducing representation switching frequency. This prevents users from experiencing frequent fluctuation in video quality. Moreover, less representation switching means a proxy cache is more likely to cache a relatively complete representation, which may in turn benefit cache hit ratio of subsequent DASH clients who request the same video ii) avoiding buffer underflows and the resulting pauses of the video playback. iii) When buffer level is at reasonable level, maximizing the video bit rate so that users could enjoy higher quality.

Equation 1 shows that the bandwidth estimation takes history bandwidth into consideration and calculates a weighted average. Consequently, the bandwidth estimation behavior can be adjusted through changing the weights of B_l and B_c.

Adjusting the weights can affect the rate adaptation but we keep the original set up to make a good balance between adaptation agility and video quality stability.

Another thing worth addressing is that the characteristics of TCP transmission is not particularly taken into consideration in this algorithm design. This is based on the understanding, as put in [30], that TCP and DASH client congestion control operate at different protocol levels and at significantly different time scales.

5.1.2 The mechanism of Gearbox in brief

We name our adaptation Algorithm as **Gearbox**. As shown in figure 11, Gearbox divides the receive buffer into multiple sections, with each section overlapped with the adjacent ones. Each section is called a Gear and denoted as g. The collection of all Gears is denoted as G. Each Gear $g \in \mathbf{G}$ has a lower and upper boundary denoted as β_{lg} and β_{ug} respectively, where each boundary represents a

Gear Position	Gear boundaries (BufferLevel in %)	Bitrate Threshold B_g	Threshold of BufferLevel Change
g=1	$[\beta_{l1}, \beta_{u1}] = [0\ ,\ 25]$	$B_c \times \rho^{-2}$	$\Delta_1 = 0 \times \tau$
g=2	$[\beta_{l2}, \beta_{u2}]$=[15 , 40]	$B_c \times \rho^{-1}$	$\Delta_2 = 1 \times \tau$
g=3	$[\beta_{l3}, \beta_{u3}]$=[30 , 75]	$B_c \times \rho^{0}$	$\Delta_3 = (Cn - 1) \times \tau$
g=4	$[\beta_{l4}, \beta_{u4}]$=[55, 100]	$B_c \times \rho^{1}$	$\Delta_4 = (Cn + 1) \times \tau$

Table 1: Default Gearbox configuration with four Gears(Cn=3 for all Gear)

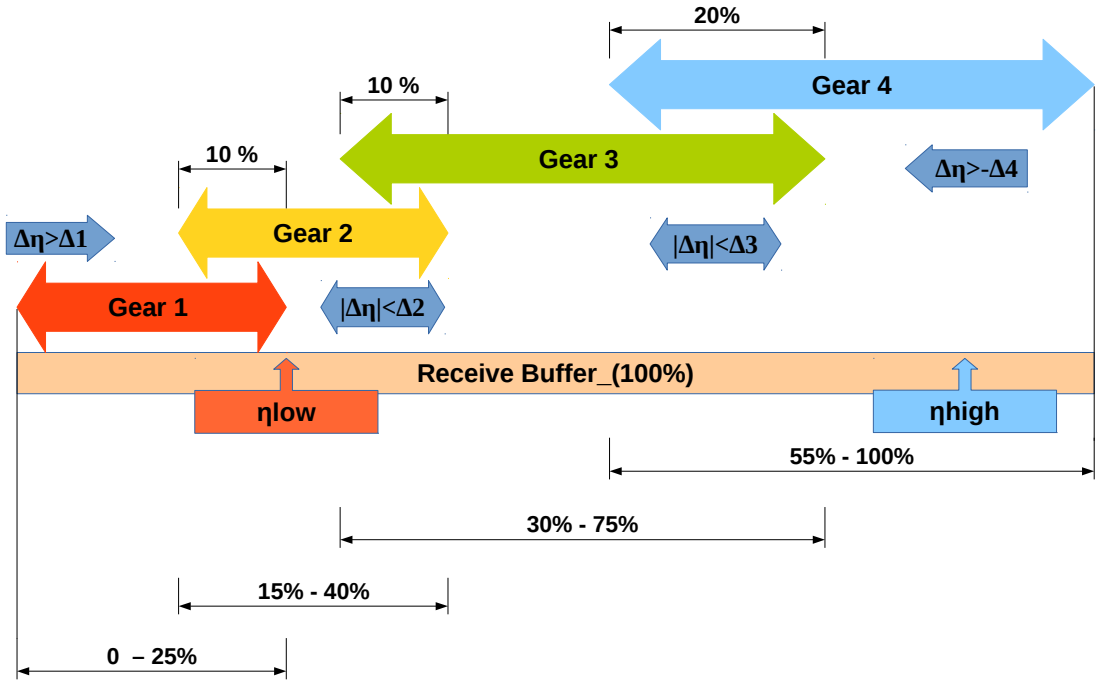


Figure 11: Diagram of Gearbox with the default four Gears

buffer level in percentage. For $g = 1$, the Gear boundaries are denoted as β_{l1} and β_{u1} respectively. When the buffer level reaches the upper or lower boundary of a certain Gear, the value of g is incremented or decremented by 1, which we refer to as a shift up or shift down. Gearbox can be configured to have **an arbitrary number of Gears** but by default we configure Gearbox to have **four Gears** in total.

Specifically in our default setting, β_{l1} and β_{u1} are set to be 0% and 25%, respectively; β_{l2} and β_{u2} are set to be 15% and 40%. Assuming the buffer is being filled from empty and the value of g is initially set to 1. When buffer level goes up to 25 percent, a shift up is performed and the value of g is then set to 2. If the buffer shrinks to less than 15 percent, a shift down is triggered and g is reset to 1.

Under each Gear, we employ a unique representation switching policy (RSP). Under a low Gear, we perform a representation selection that prioritizes buffer filling speed in order to avoid buffer underflows. Under a high Gear, the representation switching policy is set to be aggressive in order to maximize bandwidth usage and the video quality.

A gear specific RSP includes a 2-tuple set $\{B_g, \Delta_g\}$ where B_g is the threshold for the video bit rate and Δ_g the threshold for the buffer changing rate. B_g equals to B_c , the current TCP receive rate , multiplied by a coefficient; Δ_g is a multiple of the Dash segment length. Both B_g and Δ_g are configurable parameters.

To decide the coefficient required to calculate B_g Gearbox takes into data from the downloaded MPD file. By extracting each B_i of $R_i \in \mathbf{R}$, Gearbox calculates the bit rate ratios of all adjacent representations and average them to get the ratio ρ. Gearbox then takes ρ to initializes B_g for all gears. Specifically, B_g is calculated by multiplying B_c and a power of ρ. For example. At Gear 1 where $g = 1$, $B_g = B_c \times \rho^{-2}$. At Gear 2, $B_g = B_c \times \rho^{-1}$;

Δ_g is configured based on our modeling of Gearbox, which is presented in section 5.1.5.

Gearbox triggers a representation switch under two conditions. One is a gear shift and the other is the dramatic change in buffer level. Right after shifting to a new gear, representation switch is performed and a new representation is chosen based on B_g of the new gear position. When the buffer is within a Gear boundaries and no gear shift happens, a buffer level changing rate that exceeds Δ_g also triggers the representation switch, which is the same as the representation switch operation taken after a gear change.

The Gearbox mechanism is analogous to the gear change of a car. The buffer level resembles the car speed and the gear boundaries resembles the lower and upper speed limits of a certain gear. B_c resembles the driving power while the chosen B_i resembles the overall resistance. At low gear, limited by B_g the chosen B_i is small and this gives the video downloading bigger acceleration, meaning the buffer level would go up quickly. At high gear, the big value of B_g unleashes the maximum

resistance B_i. As a result, the buffer level builds up much slower or even shrinks. In a car, adjacent gears have overlapped speed ranges which serve as a margin for the driver to gear up or gear down. The bigger the margin, the lesser the driver needs to switch between the two gears. In the case of Gearbox algorithm, we also have each Gear's buffer range overlapped with adjacent ones. If there were no overlapped margin, a buffer level that constantly swings across a Gear boundary would force Gearbox to switch representation constantly.

5.1.3 Gearbox Operation

The algorithm detail is shown in algorithm 1a. It is also illustrated in flowchart 12.One thing worth reiterating is that algorithm 1a only demonstrates Gearbox with four Gears, more Gears can be easily implemented by extending the code. The details of the Gearbox operation are explained in the following text:

Each time before requesting a segment, Dash would go through Gearbox to decide the representation for that segment. First Gearbox checks the buffer level and decides if a gear shift shall be performed. If a gear boundary is reached, the gear is then shifted, which in turn triggers the representation switching. To switch to new representation, Gearbox compares every B_i of $R_i \in \mathbf{R}$ with B_g of the current Gear and chose R_i for S_n. If no gear change is performed, gearbox periodically monitors the buffer level and triggers the representation switching if the buffer level is changing too quickly.

Details of how the representation switching is triggered by a buffer-level-change are described as follow: DASH-JS downloads segments consecutively and in serial, which means there is no pipelining in the download A.1. During the download process, the change of buffer level is periodically checked when the Gear position remains unchanged for a succession of downloads. We define a macro CYCLE as the reevaluation cycle of the buffer level. A variable named as COUNTER, would increments every time Gearbox runs. During the successive downloads, if COUNTER reaches CYCLE before a gear change is performed, the latest buffer level is compared against the buffer level of the last time when COUNTER=0. COUNTER is reset to 0 either after its value reaches CYCLE or when a gear-shift is performed.

We denote the latest buffer level, **in seconds**, as η_n and the buffer level of last COUNTER=0 occurrence as η_0. Thus $\Delta = (\eta_n - \eta_0)$ can reflect the changing rate of buffer level . The threshold of buffer level change within a reevaluation cycle is denoted as Δ_g. Δ_g is set up for each $g \in \mathbf{G}$. If there is no Gear change in CYCLE consecutive downloads, Gearbox would compares Δ with Δ_g to see if the buffer is shrinking or filling up at a dramatic speed. If positive, Gearbox calls for representation switch to select the proper representation based on B_g .

In algorithm 1a, we name the video segment duration, which is in seconds, as τ. Specifically, we set CYCLE to 3. And we set the Δ_g values where $\Delta_1 = 0$,

Main-algorithm 1a SelectRepresentation

```
1: Input : β, η_n, {R_i|R_i ∈ R}, B_c, R_i(t)   Output : R_i(t)+1
2: function SelectRepresentation
3:                                   ▷ set the default Representation to be the same as the previous choice
4:     R_i(t)+1 := R_i(t);
5:                                                                          ▷ apply gear specific actions
6:     if g = 1 then
7:        if GearchangeFlag = True then
8:           Evaluates Representation();
9:           GearchangeFlag := False; COUNTER : = CYCLE;
10:       else if COUNTER=CYCLE then
11:          if (η_n − η_0) < Δ_1 then                                    ▷ Check buffer change level
12:             Ri(t) + 1 := Ri(low);                                    ▷ evaluate Representation if needed
13:          end if
14:       end if
15:       if β >= β_u1 then                                              ▷ Check buffer and change Gear
16:          g := 2; GearchangeFlag := True;
17:       end if
18:    else if g = 2 then
19:       if GearchangeFlag = True then
20:          Evaluates Representation();
21:          GearchangeFlag := False; COUNTER : = CYCLE;
22:       else if COUNTER=CYCLE then
23:          if (η_n − η_0) < −Δ_2 then
24:             Evaluates Representation();
25:          end if
26:       end if
27:       if β >= β_u2 then
28:          g := 3; GearchangeFlag := True;
29:       end if
30:       if β <= β_l2 then
31:          g := 1; GearchangeFlag := True;
32:       end if
33:    else if g = 3 then
34:       if GearchangeFlag = True then
35:          Evaluates Representation();
36:          GearchangeFlag := False; COUNTER : = CYCLE;
37:       else if COUNTER=CYCLE then
38:          if (η_n − η_0) < −Δ_3 ∨ (η_n − η_0) > Δ_3 then
39:             Evaluates Representation();
40:          end if
41:       end if
42:       if β >= β_u3 then
43:          g := 4; GearchangeFlag := True;
44:       end if
45:       if β <= β_l3 then
46:          g := 2; GearchangeFlag := True;
47:       end if
48:    else if g = 4 then
49:       if GearchangeFlag = True then
50:          Evaluates Representation();
51:          GearchangeFlag := False; COUNTER : = CYCLE;
52:       else if COUNTER=CYCLE then
53:          if (η_n − η_0) < −Δ_4 then
54:             Evaluates Representation();
55:          end if
56:       end if
57:       if β <= β_l4 then
58:          g := 3; GearchangeFlag := True;
59:       end if
60:    end if
61:                                                 ▷ reset counter and bufferlevel recorder
62:    if COUNTER=CYCLE then
63:       COUNTER:=0;
64:       η_0 := η_n;
65:    end if
66:                                                 ▷ increment COUNTER each time Gearbox runs
67:    COUNTER+=1;
68: end function
```

Sub-algorithm 1b Evaluates Representation

```
1: function EVALUATES REPRESENTATION
2:     Ri(t) + 1 := Ri(low);
3:     while Ri IN R do
4:         if B_i < B_g then                    ▷ B_g is gear specific
5:             Ri(t) + 1 := Ri;
6:         end if
7:     end while
8: end function
```

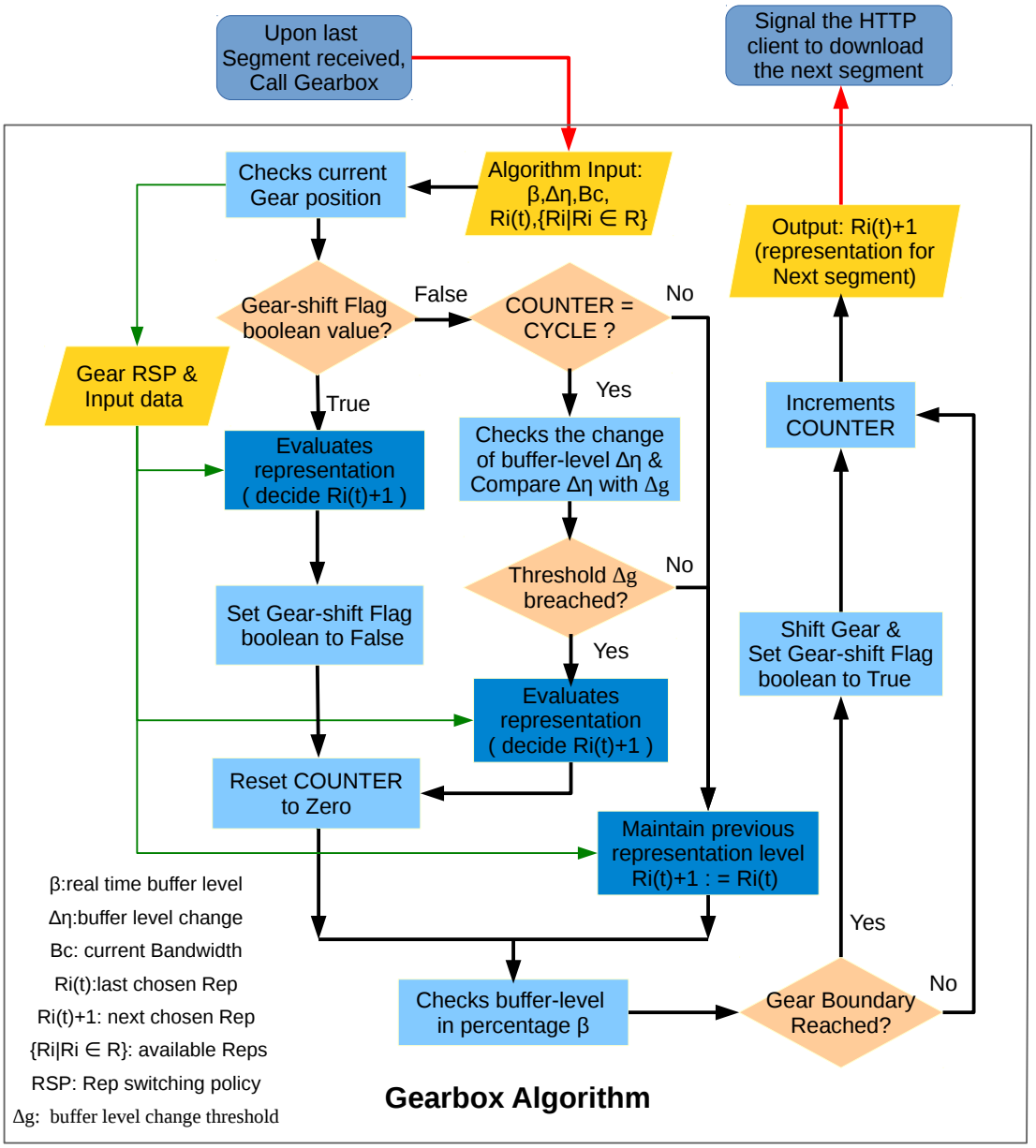


Figure 12: Flowchart of Geabox operation

$\Delta_2 = 1 \times \tau$, $\Delta_3 = (C_n - 1) \times \tau$, $\Delta_4 = (C_n + 1) \times \tau$. The CYCLE macro and Δ_g together with other adjustable parameters are configured to achieve a desirable behavior of Gearbox. These parameters are shown in Table 1. How these parameters are configured is a preference issue and the modeling and analysis of the Gearbox configuration is shown in section 5.1.5 and 6.4

The Gearbox Algorithm instance is initialized when DASH-JS is loaded. The variables initialized are shown below: g := 1; η_0 := 0; GearChangeFlag : = True; COUNTER : = CYCLE;

To operate, the algorithm takes in several parameters: i) the current buffer level, in percentage, denoted as β; ii) the current buffer level in seconds η_n; iii) the current estimated network bandwidth B_c ; iv)the average representations bit rate ratio ρ.

Assuming the algorithm is invoked at time t, right after the last segment $S_{n(t)}$is received. The R_i Gearbox will chose for the next segment to fetch $S_{n(t)+1} + 1$ is the algorithm output and is denoted by $R_{i(t)+1}$; the latest downloaded segment's R_i is denoted by $R_{i(t)}$. we denote the R_i with lowest bitrate B_i by $R_{i(low)}$. And for all $R_i \in \mathbf{R}$ and $R_{i+1} \in \mathbf{R}$, we have $B_i < B_{i+1}$. Given these settings and variables naming, the algorithm itself is presented as in algorithm 1a and is illustrated in flowchart 12.

5.1.4 Gearbox behavior under each gear

Under Gear 1, we start the segment download by choosing the representation whose bit rate is two levels lower than the one that most closely match the available bandwidth. This ensures the buffer level be filled up quickly and the playback start quickly. When under Gear 1, if the buffer level decreases, that is $(\eta_n - \eta_0) < \Delta_1 (\Delta_1 = 0)$, Gearbox would directly choose the lowest representation, which helps reduce the possibility of buffer under run. Aiming to avoid buffer starvation, the Gear 1 policy does not necessarily choose the worst video quality since the initial representation switching would not blindly choose the lowest representation.

Under Gear 2, the policy is specified based on the same principle as under Gear 1, only more tolerant. By raising the bitrate threshold and the threshold of buffer-level-change-rate, under Gear 2, the adaptation would allow the buffer to fill up slower. Slow shirking of the buffer level is also tolerated.

Under Gear 3, bit rate of the chosen representation is close to the estimated bandwidth. The buffer level changing is monitored to avoid both drastic shrinking and filling rate. If no dramatic buffer level changing is happening under Gear 3, the chosen representation would stay the same for subsequent segments downloads. The range of Gear 3 covers 45 percent of the buffer length, making it the major Gear position.

When the network bandwidth is stable and sufficient, DASH-JS would fill the buffer to reach Gear 4. Under Gear 4, the RSP is prioritized to choosing higher video quality. By setting $B_g = B_c \times \rho^1$, the bit rate of the chosen representation would be only one level higher than the representations that matches the estimated Bandwidth B_c. This ensures the buffer level will not shrink too quickly. In our case, Gearbox is configured to handle 1s and 2s segments. With the buffer size set to 40 s, the 20 percent overlapped buffer margin between Gear 3 and Gear 4 would give us 8 s of buffer. In our test, this margin gives at least over 10 s of playback, meaning even though the representation bit rate is higher than the available bandwidth, the playback can sustain for over 10 seconds before Gearbox performs a gear down operation and switch to a representation with lower bit rate.

The benefit of our design is that the buffer level changing threshold and the reevaluation cycle (in seconds) is actually the function of segment size. It is obvious that the buffer size is supposed to be proportional to the segment size to make sure an adaptation algorithm would have enough time to respond to bandwidth fluctuation. In light of this, for bigger segment size, such as 4 s and 10 s, we recommend to reconfigure the buffer size η, to make sense of the Gearbox's margin setting. Another thing worth mentioning is that when buffer level is found changing too quickly, either shrinking or filling up, Gearbox do not simply chose a lower or higher representation, instead it reevaluate the representation level by comparing B_i with B_c under the Gear-bound RSPs. This is based on the understanding that network throughput is memoryless. We prefer to reevaluate the representation based on current bandwidth estimation. Besides, since the bandwidth estimation mechanism of DASH-JS reflects historical network condition (referring to equation 1), reevaluation based on B_c does not disregard the average value of bandwidth completely.

5.1.5 Gearbox Modeling

There are two important parameters in Gearbox configuration and they are the reference bitrate ratio ρ, which decides the video bitrate threshold for each gear, and the buffer-level-change threshold Δ_g respectively. (Check Table 1 for the specifics)

Here we provide the Gearbox modeling that help optimize the two parameters. Gearbox aims to maintain the buffer level at reasonable level. We denote by η the buffer level in seconds. Given that the Dash client is downloading segments continuously and the request time is negligible, when the chosen representation bit rate is B_i Mbit/s and the current bandwidth (TCP throughput) is B_c Mbit/s it is obvious that the buffer is drained of B_i Mbit/s and filled up of B_c Mbit/s. During time Δt, the amount of data filled in the buffer is $(B_c - B_i) * \Delta t$, which is also equal to $B_i * \Delta\eta$. Then the buffer level changing rate can be expressed as

$$\frac{\Delta\eta}{\Delta t} = \frac{(B_c - B_i)}{B_i} \tag{2}$$

Since B_i is chosen based on the threshold B_g, which is gear specific, the value of B_i is no bigger than B_g. According to table 1, at Gear 2 (g = 2), for instance,

where $B_g = B_c * \rho^{-1}$, B_i is then no larger than B_c/ρ. Assuming $B_i \approx B_c/\rho$ exists in the available representations, then at gear 2 the chosen $B_i \approx B_c/\rho$ and we have $\frac{\Delta\eta}{\Delta t} = \frac{(B_c - B_i)}{B_i} \approx \rho - 1$.Under this condition, the buffer level is being filled up by $\rho - 1$ seconds per second. Similarly, we make an optimistic assumption that, at each gear, there exists a representation whose bit rate approximates the gear specific B_g. Then we can conclude the following equations:

$$\begin{aligned} \frac{\Delta\eta}{\Delta t} &\approx \rho^2 - 1; (g = 1) \\ &\approx \rho - 1; (g = 2) \\ &\approx 0; (g = 3) \\ &\approx \frac{1-\rho}{\rho}; (g = 4) \end{aligned} \tag{3}$$

In our Gearbox configuration, since ρ is determined by calculating the average ratio of adjacent representations, the buffer changing rate would depend on the representations available in the MPD file. Under our test Dash content, the average ratio ρ is 1.23. At this, $\frac{\Delta\eta}{\Delta t} \approx 0.5$ at gear 1, $\frac{\Delta\eta}{\Delta t} \approx 0.23$ at gear 2 and $\frac{\Delta\eta}{\Delta t} \approx -0.18$ at gear 4. Based on the prospective value of $\frac{\Delta\eta}{\Delta t}$ we can calculate how long the playback duration can be provided by a gear margin. For instance, gear 3 and gear 4 have an overlap margin of 20%. As the buffer size is 40 seconds in our configuration, the margin provides 8 seconds of data. At gear 4,under the most optimistic condition, the buffer shrinks at a rate of 0.19 second per second, then the 8 seconds margin can sustain $44 (= 8/0.18)$ seconds of video playback. In other words, the margin ensures that no gear shift and representation switching would happen during 44 seconds of playback, under optimum conditions. Conversely, we can decide how the margin is configured based on the prospective $\mathrm{d}\eta/\mathrm{d}t$ under each gear. The margin can be configured to be absolute value or a percentage, making it adaptable to different preferences.

The logic of calculating ρ based on the MPD is that the dynamically calculated ρ can ensure that a Gear-shift-up or shift-down would only switch up or switch down the representation level by 1. Gear 3 is configured to cover 45% percent of the buffer so that the bit rate of the chosen representation can be closest to the actual observed TCP bandwidth most of the time during playback. We refer to the bit rate of the representation chosen by Gear 3 as the "matching-bit-rate". When the buffer level is out of the Gear 3 range, Gearbox would then allow to choose representations with lower or higher bit rates, relative to the "matching-bit-rate" and the observed bandwidth.

One thing worth mentioning is that calculating ρ based on MPD could sometimes cause unwanted effects, such as the rapid buffer level shrinking at gear 4, if the bit rate differences among available representations were too big. To deal with such extreme cases, one possible solution could be to set a default value for ρ. By collecting a big number of MPD files and doing statistic analyses, the typical bit rate ratio between adjacent representations can be found. Then the algorithm can use this

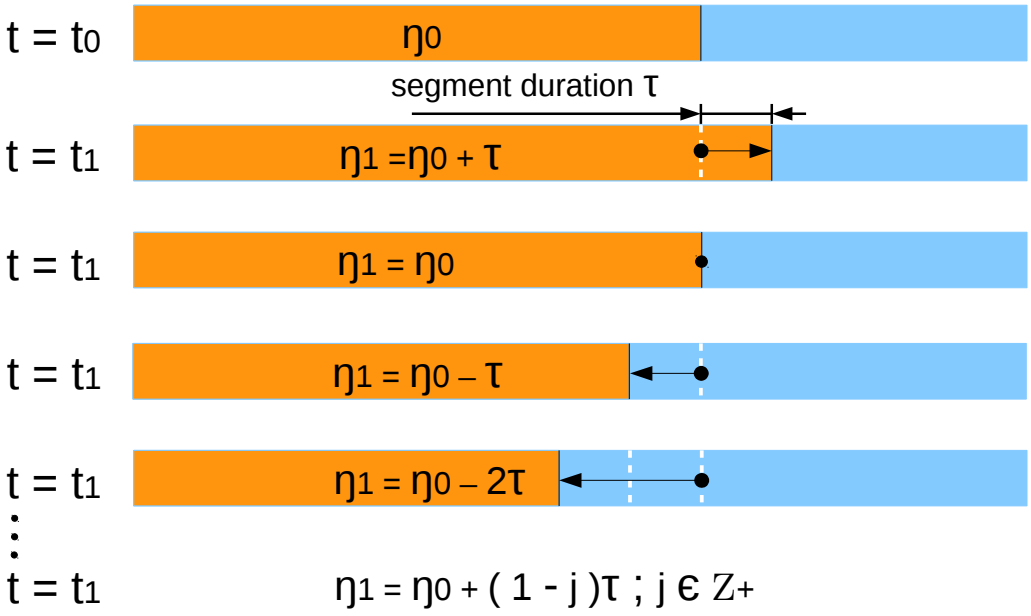


Figure 13: buffer level changes over time

typical bit rate ratio as the default value of ρ. By setting a threshold value for ρ , the algorithm can use the default value of ρ instead of the calculated one, if the calculated value of ρ based on a specific MPD breaches the threshold.

Another configurable parameter is Δ_g, the threshold for the buffer change every CYCLE. Since the receive buffer is drained or filled by 1 segment each time, the possible value of buffer level change can only be discrete, which is the multiple of segment duration τ. Figure 13 illustrates this process. Assuming at time t $= t_0$, Dash issues the request for the next segment and the buffer level in seconds is η_0. At time t $= t_1$, the segment response is received and the buffer level is then η_1. During time $[t_0, t_1]$ the dash client would drain the buffer by $j * \tau$ where $j \in \mathbb{Z}^+$, and the received segment fills the buffer by τ. We denote the buffer level of t $= t_1$ by η_1 then we have $\eta_1 - \eta_0 = (1 - j)\tau; (j \in \mathbb{Z}^+)$. Assuming from t $= t_0$ till t $= t_n$, C_n segments (the value of CYCLE) have been requested and received and j segments is drained from the buffer, then we have

$$\eta_n - \eta_0 = (Cn - j)\tau; (j \in \mathbb{Z}^+) \tag{4}$$

The threshold Δ_g for $\eta_n - \eta_0$ can be set based on equation 4, to decide the tolerance of buffer level change at each re-evaluation cycle, represented by the macro CYCLE.

Due to the fact that Dash-js issues segment requests one by one, in a non-preemptive fashion, the buffer level is only monitored after a segment download is finished. Gearbox is set to monitor the buffer level after every C_n downloads, thus the monitored buffer level change can reflect $\mathrm{d}\eta/\mathrm{d}t$ over the time period of $[t_0, t_n]$. When both

sides of equation 4 are divied by $(t_n - t_0)$, we get $\frac{(\eta_n - \eta_0)}{(t_n - t_0)} = \frac{\Delta\eta}{\Delta t} = \frac{(Cn-j)\tau}{(t_n - t_0)}$. During $(t_n - t_0)$, j segments have been drained from the buffer, and the playback time of these j segments is $j * \tau$. Then roughly we have $(t_n - t_0) \approx j * \tau$, assuming the playback has not been interrupted. Then we conclude that

$$\frac{\Delta\eta}{\Delta t} \approx \frac{(Cn - j)}{j}; (j \in \mathbb{Z}^+) \tag{5}$$

Combining equation 2 and 5 we have

$$\frac{\Delta\eta}{\Delta t} \approx \frac{(Cn - j)}{j} \approx \frac{(B_c - B_i)}{B_i}; (j \in \mathbb{Z}^+) \tag{6}$$

Given C_n is set to 3 (refer to table 1), based on equation 6 and equation 3, a desirable value of j can be chosen for each Gear so that the Gear-related threshold Δ_g can then be decided. This threshold would determine the tolerance of buffer change rate under each gear. A rate that exceeds it would trigger a reevaluation of the representation level.

5.2 Summary of Gearbox implementation

This chapter presented the design principles and the details of the Gearbox rate adaptation algorithm. For the simplicity of reading, no actual Javascript code is presented and explained. Instead, this chapter provides the algorithm logic, the flowchart and some illustrative graphs to help readers understand why and how Gearbox is designed. For more technical details and the source code, please check the attached information in the Appendix section A.5.

Gearbox is an innovative rate adaptation algorithm that aims to improve the performance of the DASH client in CDN networks, where edge servers commonly caches HTTP objects. Gearbox provides a simple mechanism that bind the rate adaptation behavior with the fill-level of the client receiving buffer. By binding different representation switching policies (RSPs) to different ranges of the receiving buffer, Gearbox is able to perform rate adaptation with carefully designed priorities. When the buffer fill-level is low, Gearbox can prioritize to avoid buffer underflow. Whereas when the buffer fill-level is high, Gearbox prioritizes to fully utilize the network capacity and to increase the video quality. Apart from this design of intelligently choose rate adaptation priorities, Gearbox also implements our idea of re-evaluation cycle, which can help effectively avoid the premature decisions of representation switching upon short term network throughput fluctuations.

Gearbox is also configurable and flexible. It can be configured to use different numbers of Gears to suit different adaptation preferences. Optimization tests on the Gearbox configuration are conducted in chapter 6, to demonstrate how Gearbox would behave under different configurations. More importantly, the performances of Gearbox under different network environments and cache conditions are also tested in chapter 6.

6 Performance Evaluation

In this chapter, the proposed algorithm is tested against the baseline rate adaptation algorithm of the original DASH-JS. These tests aim to reveal how DASH-JS performs under different network conditions and user scenarios. Based on the results, we attempt to shed some light on how the adaptation algorithm of a DASH client shall be designed in order to avoid unwanted behaviors under more realistic network environments, such as a network involving CDN caches.

Apart from the multiple tests that help evaluate the performance of both algorithms, an optimization test of the Gearbox algorithm is also conducted. Results of the optimization test not only show the effectiveness of our parameter configuration of Gearbox but also demonstrate how Gearbox can be freely configured to better serve different users and needs, such as mobile users and the need of live streaming.

6.1 Evaluation method

Both the Gearbox algorithm and the original DASH-JS algorithm will be evaluated based on multiple tests. These performance tests include basic tests without caching and more complex tests with proxy caches enabled. By testing both algorithms under multiple settings, we try to find out how the different network impairments and user scenarios can affect the performance of DASH. We also try to identify how an adaptation algorithm can better suit a DASH streaming network where proxy caches are involved.

The testing DASH content [8] is available in different segment-duration versions, with each version including multiple representations that contain video segments of the same duration in seconds. Among the available segment durations, which include 1 second, 2 seconds, 4 seconds, 6 seconds and 10 seconds, we specifically choose the segment duration of 1 second for our tests, since the shortest segment duration means the biggest number of segments, which would naturally create the worst condition of incomplete cached representation [22]. The shortest segment duration also means the client rate adaptation would operate most frequently, which can help better expose the potential problems of the tested algorithms. Tests of other segment durations may be conducted in future works.

6.2 Testbed Setup

The test bed includes a server machine and two cascaded proxy machines, together with several endpoint machines that run the clients. The server runs Apache 2.24 and serves the DASH content dataset [8] as well as the DASH-JS files (javascripts). Both cascaded proxy machines run squid3[10] and are configured to be forward proxies. The client machines use Google-chrome 35.0.1916.153 as the DASH client. Dummynet [39] is used to create the variation in link capacity, latency, and packet loss. The topology

[10] http://www.squid-cache.org/

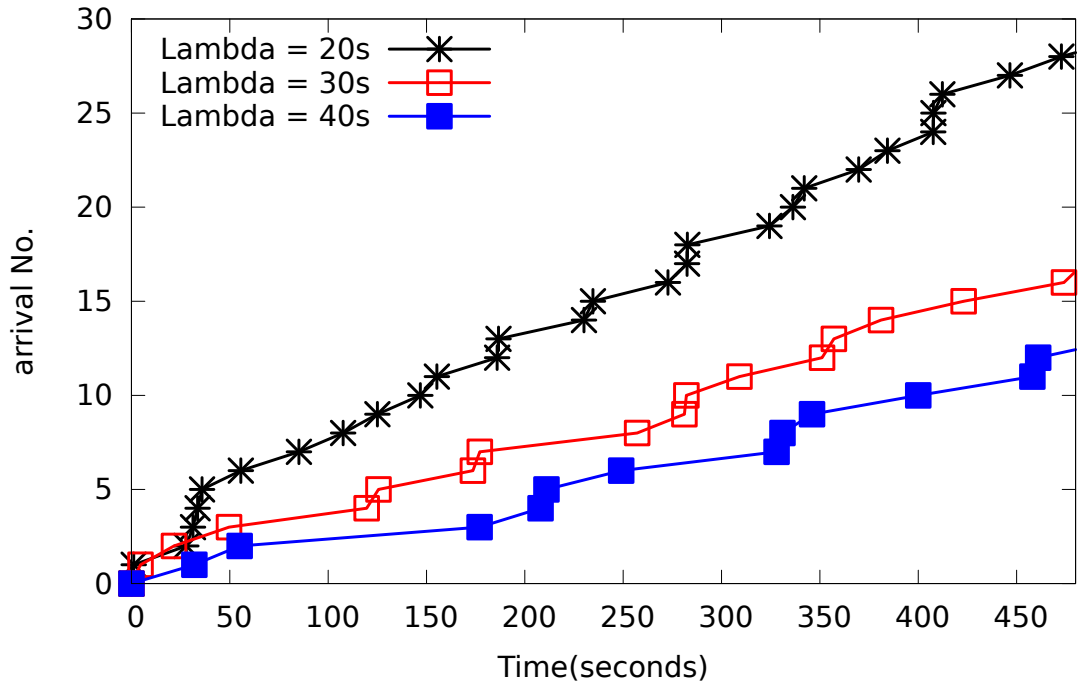


Figure 14: Poisson Arrival Patterns applied to client instances in our tests.

of the test bed is found in figure 27

The video stream "Of Forest and Men" is used in all tests [8]. The movie clip is of 7 minutes and 33 s in duration and the chosen segment duration is 1 s [25]. The tested video resolution is 1280×720. 9 different representations are available for this resolution and the average bit rate of the 9 representations are 900 Kbps, 1100 Kbps, 1400 Kbps, 1700 Kbps, 2000 Kbps, 2500 Kbps, 3000 Kbps, 4000 Kbps, and 5000 Kbps respectively. In our test, only the relatively high resolution of 1280×720 is used and the rate adaptation only choose from the 9 different represents under the same 1280×720 resolution. By doing this we eliminate the very low bit rate representations under low resolutions, in order to create more challenging conditions for the adaptation algorithm. The poor selection of low bit rate representations would increase the possibility of buffer underflow occurrences, providing more interesting results for later analyses. In our tests, the media clients either all start at the same time or request media modeled by a Poisson arrival pattern. The Poisson arrival patterns used in our tests are shown in figure 14.

All emulation results are collected through our added functions in DASH-JS. In all tests, we record the real time bandwidth estimated by the client and the chosen representation for each segment. We also record the real time buffer level of the DASH-JS "Overlay Buffer" [36], since Media Source Extension does not provide an API for accessing the real-time-level of its internal media player buffer. The real time Gear positions are also recorded when Gearbox is the active algorithm. When proxy

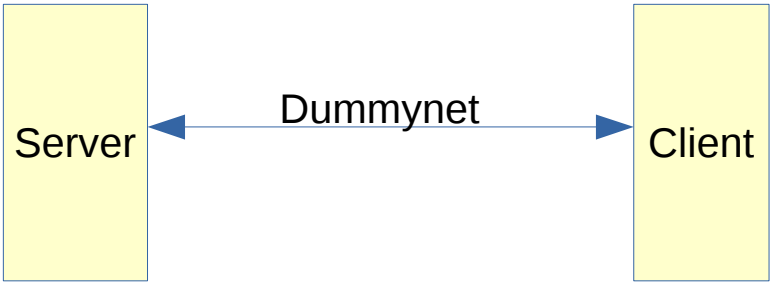


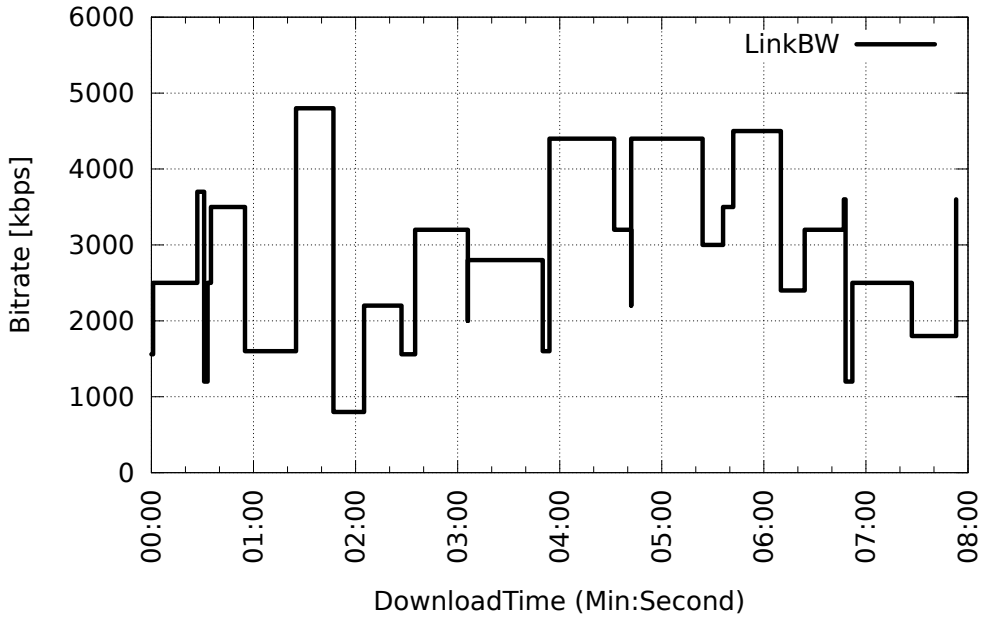
Figure 15: Testbed Setup for Microbenchmarks

caches are enabled, we also collect cache hit information from the HTTP response headers, through our DASH-JS functionalities.

6.3 Microbenchmark

In this section, we test the basic behavior and performance of DASH-JS under a varying bandwidth. By conducting this Microbenchmark we aims to find out how DASH reacts on the network impairments such as delay and packet loss. In DASH the HTTP requests are not pipelined, as a result there can hardly be any queuing problems. Thus no test on different queue sizes is conducted. Dummynet is employed to create the variations of the link bandwidth. The queue size is set to Dummynet's default, which is 50 slots. Both the original algorithm and Gearbox are put into test. As Figure 15 shows, a client is directly connected to the server and the proxy caches mentioned in section 6.2 and figure 27 are bypassed in the Microbenchmark tests. The HTTP cache control header is set to "no-cache" and this disables caching in the client browser. We configures DASH-JS at the server before each emulation round, to activate different adaptation algorithms, namely the baseline algorithm in the original DASH-JS and Gearbox in our modified version. After the client browser has acquired DASH-JS, the browser based DASH client is initialized and the emulation begins.

As Figure 16 shows, in all following Microbenchmark tests the link bandwidth is set to vary over time and the timing of bandwidth change follows a Poisson arrival pattern. This timing of bandwidth change intends to create a phantom competition situation where other "virtual clients" arriving in a Poisson pattern compete for bandwidth. However, the link bandwidth settings are random and does not necessarily mimic any realistic scenarios. The average interval of the Poisson pattern is 20 s ($\lambda = 20s$), and the bandwidth value varies from 800 Kbps to 4800 Kbps. Delay and packet loss rate are set to be constant values per test. By creating a dramatically fluctuating link bandwidth, we intend to put certain degree of pressure on the tested adaptation algorithms.



(a) The varying link bandwidth applied to all rounds of Microbenchmark tests

Figure 16: The link bandwidth applies to all microbenchmark tests. A python script is used to automate the link setting. The timing of bandwidth switching follows the Poisson arrival pattern shown in figure 14 and the average interval is 20 seconds.

6.3.1 Sample run

Figure 17 shows the sample download-time plots of both Gearbox and Baseline, under the link bandwidth shown in figure 16, with 100 ms delay. The packet loss rate is set to zero during this sample test run.

In figure 17, we can see that under the same link condition, Gearbox performs representation switching less frequently. There is no buffer underflow occurrence and the buffer level is maintained at around the 60 percent level. In contrast, the Baseline algorithm performs representation switching more often and is very sensitive to network fluctuation. The buffer encounters underflow several times, resulting in certain playback pauses. Consulting the download time line, we find that Gearbox took approximately 7 minutes, which is slightly shorter than the video length, to download all video content. The Baseline algorithm, however, took about 20 s longer to finish the download.

Under the same varying link bandwidth (figure 16), we then introduce different delay values and multiple packet loss rates to further test the performance of both algorithms.

Link delay (plr=0)	Gearbox SwitchCount	Underflow Count[Duration]	Baseline SwitchCount	Underflow Count[Duration]
10ms	15	0 [0s]	75	4 [19.3s]
50ms	19	0 [0s]	88	3 [16.7s]
100ms	14	0 [0s]	92	3 [16.1s]
150ms	8	0 [0s]	100	1 [3.20s]
250ms	2	0 [0s]	189	4 [24.4s]

Table 2: Performances of both algorithms under different delays

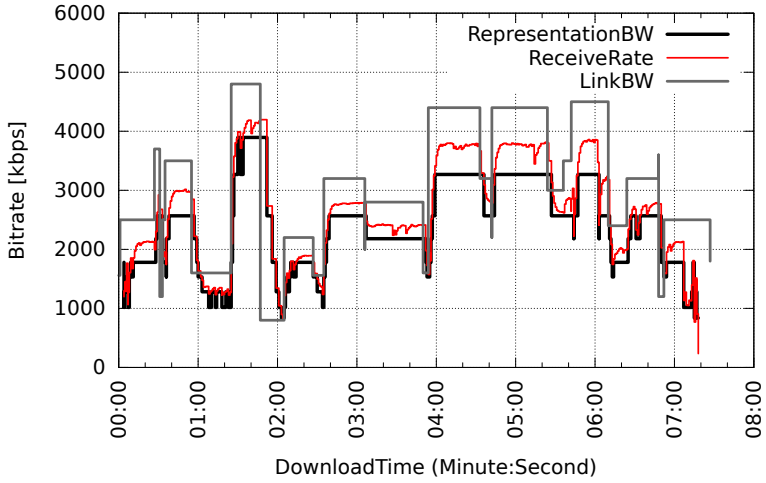
6.3.2 Delay test

We first introduce delay of 10 ms, 50 ms, 100 ms, 150 ms, and 250 ms respectively in each test round, with packet loss rate set to zero. Results of delay tests are summed up in table 2, figure 18 and figure 19.

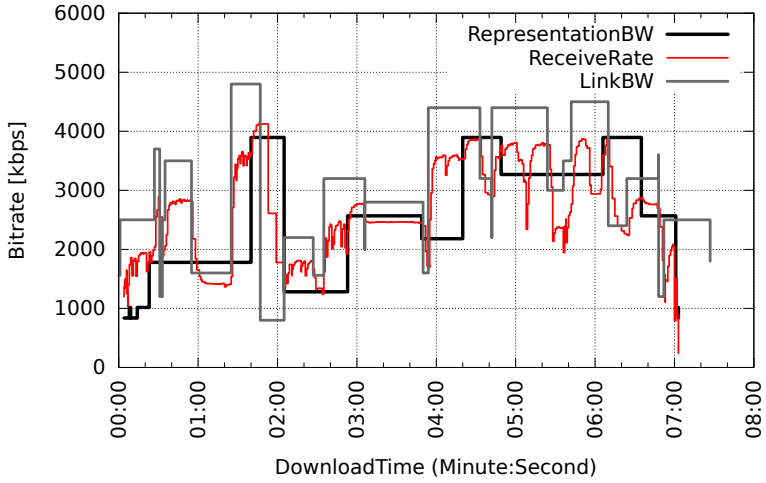
In table 2 we find that switching count of Gearbox decreases as delay increases. The baseline algorithm, on the contrary, switching representation more frequently as delay increases. Gearbox performs well in that it prevents buffer underflow under all delay settings, whereas the baseline algorithm causes buffer underflow under all settings. We can also observe that the representation switching frequency of Gearbox is significantly lower than the baseline algorithm.

Figure 18 shows the buffer level CDF under different delay. For both algorithms, bigger delay causes lower average buffer level. However, Gearbox is able to maintain the buffer level in the range of 40 percent to 70 percent during most time of the playback duration. In the case of Baseline, the buffer level is mostly under 20 percent, meaning it is not utilizing the Overlay Buffer well enough, making the client vulnerable to possible network congestion and the resulting buffer underflow.

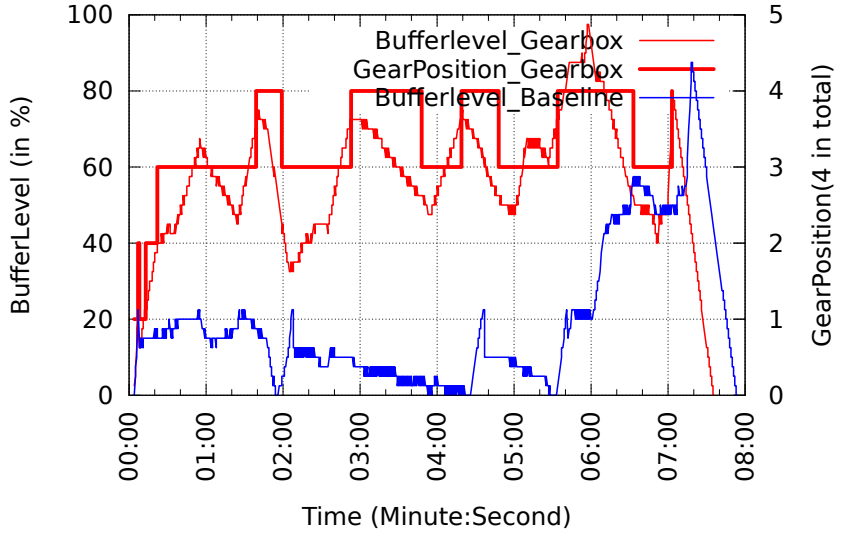
Figure 19 shows the distribution of the video bit rates, which corresponds to different representations, under different delays. Take the plots for 150 ms delay for example. The CDF lines show the baseline algorithm choose more representations while downloading. In contrast, Gearbox choose less representations and is more consistent. We can also see that Gearbox offers higher overall video bit rate.By avoiding choosing representations with lower bit rates, Gearbox is able to narrow the video bit rate range, providing better user experience. Overall, the figure shows that for both algorithms, longer delays result in lower average video bit rates. Under the longest delay of 250 ms, the average video bit rate of Gearbox goes below that of baseline, effectively preventing buffer underflow. From the results we can conclude that by assigning different adaptation policies to different buffer ranges, the Gearbox algorithm provides a way to prioritize its adaptation goals. In bad network conditions, when the receive buffer is hard to fill up, Gearbox prioritizes to ensure smooth playback of the stream. In good network conditions, when the receive buffer fills up quickly, Gearbox prioritizes to provide better video quality.



(a) realtime plot_baseline

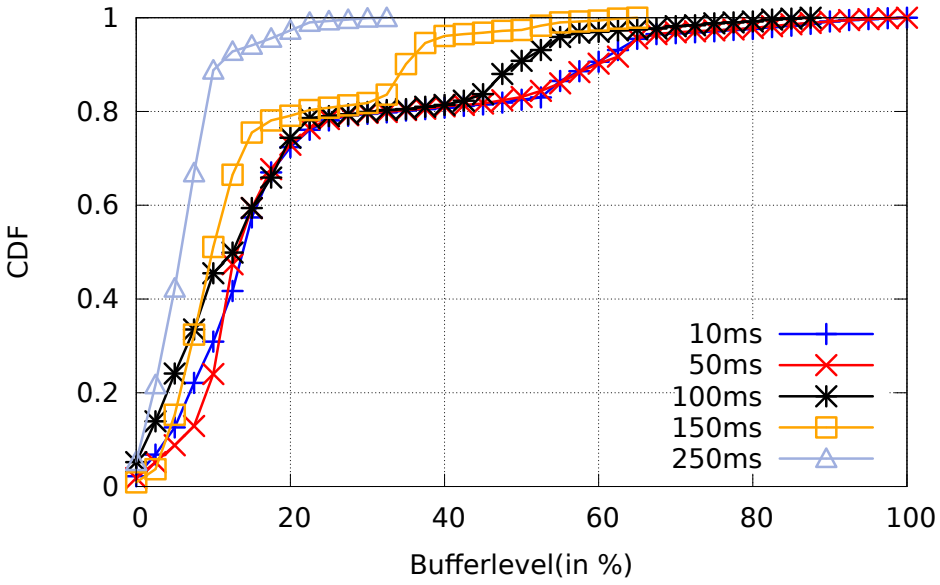


(b) realtime plot_Gearbox

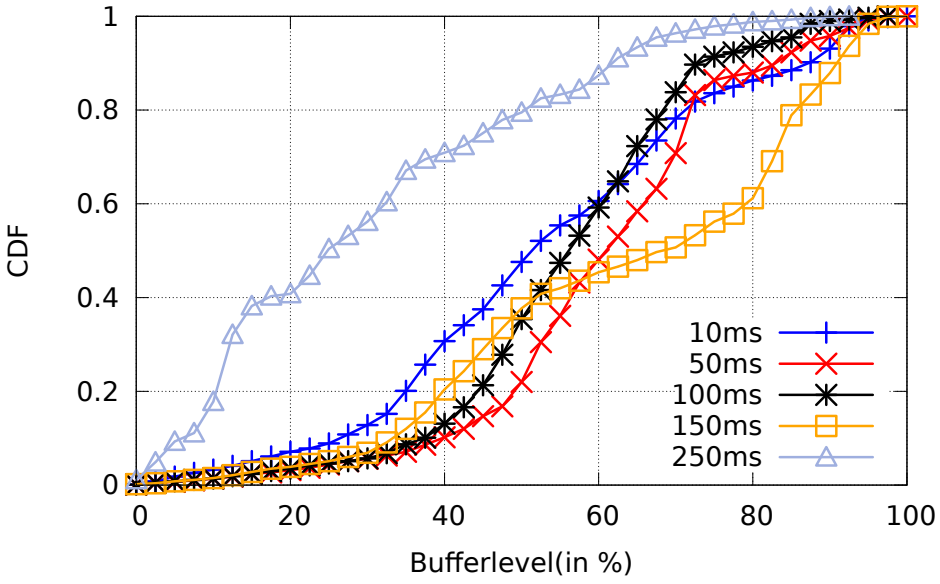


(c) realtime bufferlevel

Figure 17: MicroBenchmark: Under the same link condition, the behaviors of Baseline and Gearbox are compared. The link delay is 100ms in this sample test run

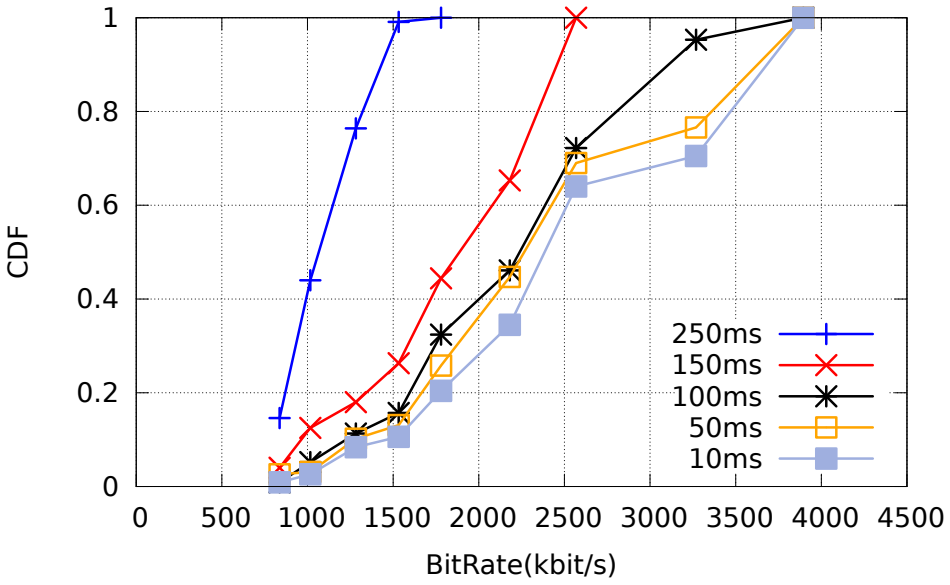


(a) Baseline

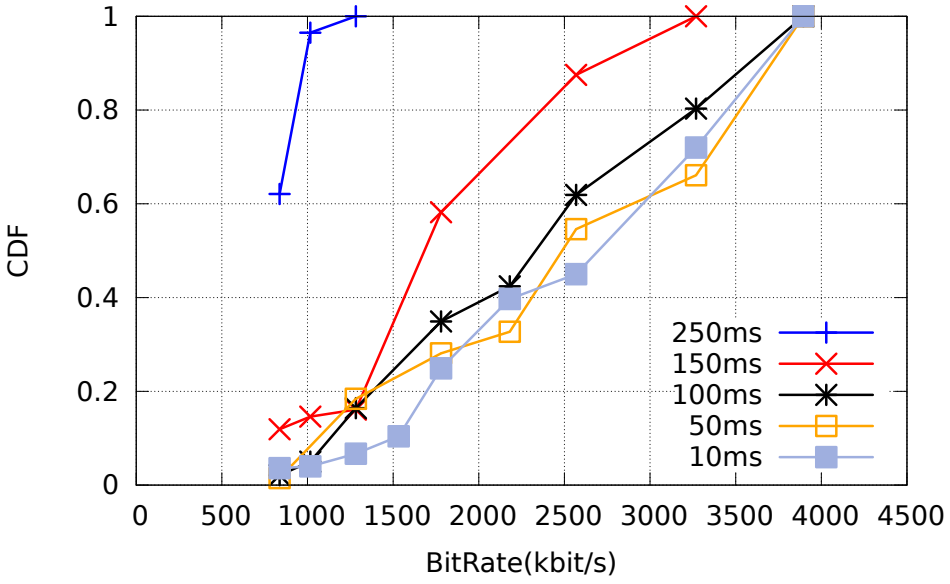


(b) Gearbox

Figure 18: MicroBenchmark: impact of link delay on both algorithms



(a) Baseline



(b) Gearbox

Figure 19: MicroBenchmark: impact of link delay on both algorithms

PLR (delay=50ms)	Gearbox SwitchCount	Underflow Count[Duration]	Baseline SwitchCount	Underflow Count[Duration]
0	19	0[0s]	88	3[16.7s]
0.01	13	0[0s]	182	2[9.40s]
0.02	4	0[0s]	137	0[0.00s]
0.05	0	13[101s]	14	14[95.8s]

Table 3: Performancec of both algorithms under different packet loss rates.

6.3.3 Packet loss rate test

We now introduce packet loss rate (plr) of 0.01, 0.02 and 0.05 all under link delay of 50 ms. Results of the plr tests are summed up in table 3, plots 20 and 21.

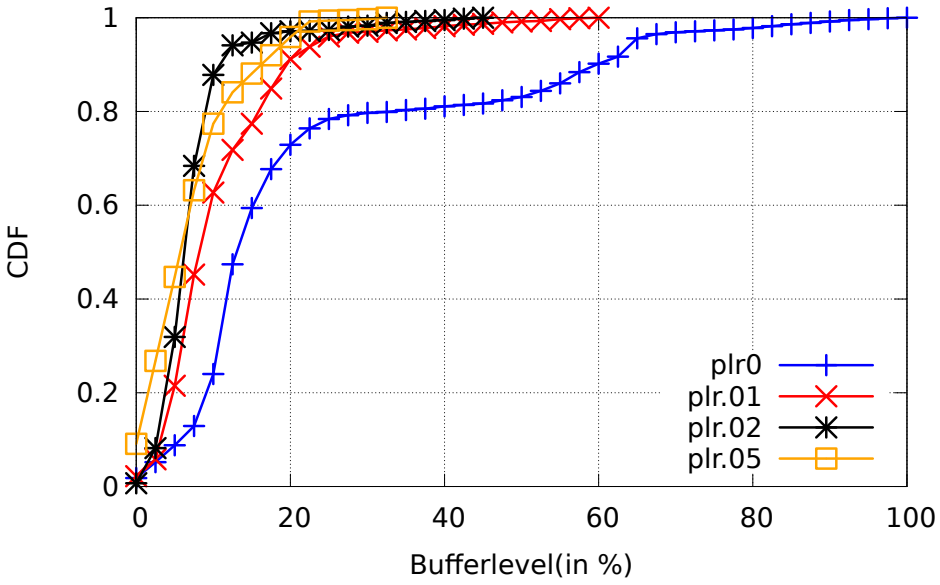
Table 3 shows Gearbox performs better than baseline under different packet loss rate settings. Gearbox switches representation less frequently and prevents buffer underflow. When plr reaches 5 percent, the TCP congestion control and retransmission greatly reduce the throughput. Figure 21, which corresponds with table 3, shows that under the condition of 5 percent packet loss, Gearbox chooses the lowest bit rate at all time, in the attempt of avoiding buffer underflow. In comparison, baseline algorithm still switches representation 14 times and resulting in one more buffer underflow count.

Table 3 also reveals that the baseline algorithm, which selects an representation solely based on the observed bit rate, does not necessarily benefit from better network conditions. This is deduced from the fact that under plr of 0.02, the actual underflow count is 0 but surprisingly, better network conditions with lower plr of 0.01 and 0 result in higher underflow counts.

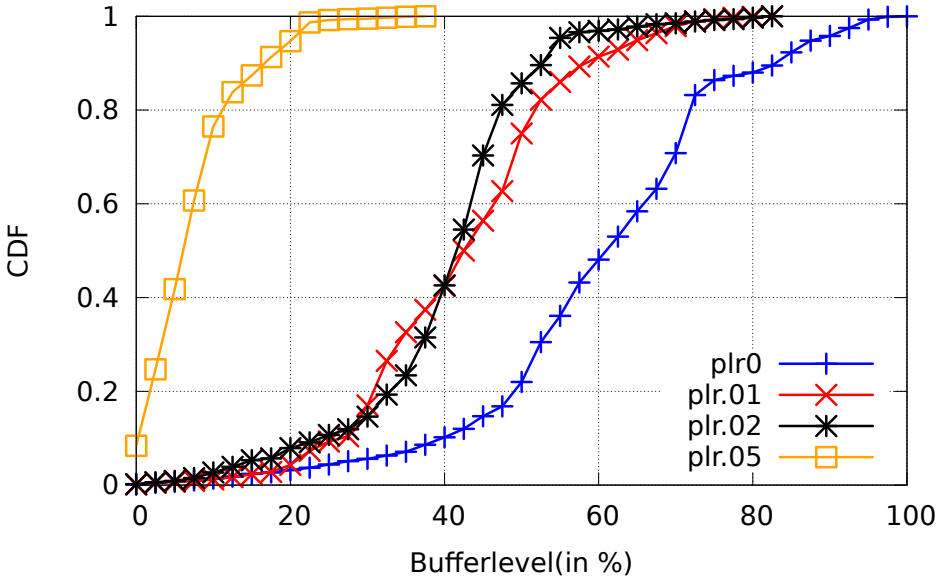
Figure 20 shows that bigger plr values cause lower buffer levels for both algorithms. Again, Gearbox is able to maintain a reasonable buffer level whereas the baseline algorithm is unable. Under plr of 5 percent, however, both algorithms perform badly since the TCP throughput is greatly reduced due to retransmission.

6.3.4 Summary of network impairments impact

From the former tests we can learn that network impairments like delay and packet loss can decrease the overall bit rate of the streamed video, which is natural since delay and packet loss can both reduce network throughput. We also learn that the Baseline algorithm, which does not consider buffer level when performing rate adaptation, can react poorly on these network impairments, leading to buffer underflows and video bit rate fluctuations that are possibly avoidable. In contrast, Gearbox takes buffer level into consideration and uses its Gear-related RSPs (representation switching policy) to perform rate adaptation, effectively avoiding buffer underflows and unnecessary video bit rate fluctuations.

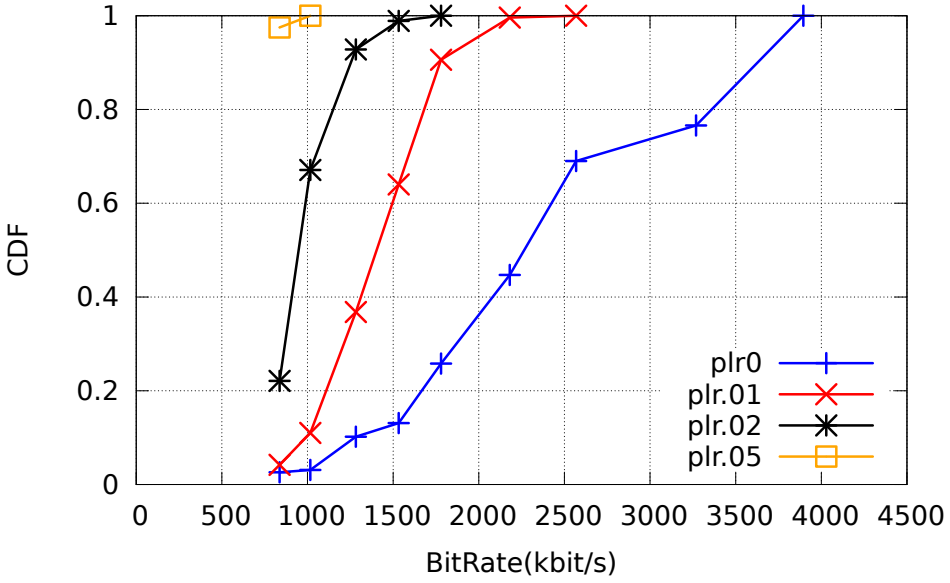


(a) Baseline

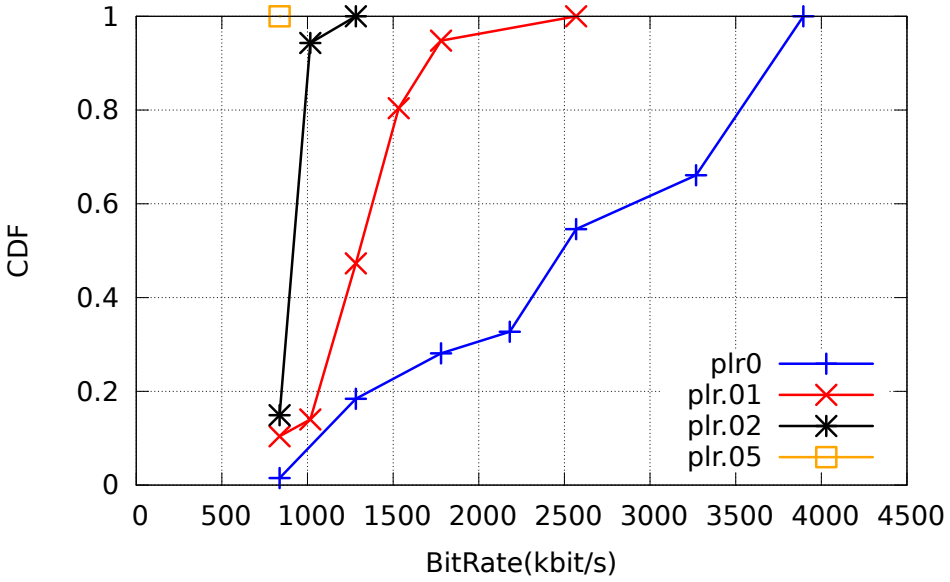


(b) Gearbox

Figure 20: MicroBenchmark: impact of Packet Loss Rate on both algorithms



(a) Baseline



(b) Gearbox

Figure 21: MicroBenchmark: impact of Packet Loss Rate to both algorithms

6.4 GearBox Optimization Tests

In this section, we reconfigure Gearbox with different settings and then test each configuration, in order to show how Gearbox can be configured freely and how the various configurations can affect the performance of Gearbox.

6.4.1 Gear Allocation

We first test the performance of Gearbox when it is configured to have different number of gears (buffer sections). When Gearbox is configured to have more than 4 gears, B_g is set to $B_c * \rho^2$ at gear 5 and $B_c * \rho^3$ at gear 6. Δ_g under newly added gears are also configured accordingly. In this test, we eliminate the margins among gears and evenly allocate the buffer range to each gear. Under this test, Gearbox with different gears are named accordingly in the figures. For instance, Gearbox with only 2 gears configured is denoted by G2, 3 gears by G3, etc. The performance of the original Gearbox under the same test link settings is also represented. The original Gearbox is denoted by G4m and it is configured according to table 1 with gear margins.

Figure 22 shows more gears would cause the average buffer level to decrease. Less gears, however, raise the average buffer level. This is due to the fact that gearbox only start to choose high bit rate representations at high gears. When there are only two gears, for example, Gearbox only employs conservative policies. Another fact is when gearbox is configured to have more gears, the range coverage of each gear shrinks since the gear coverage is configured to be a percentage of the receive buffer. Concerning the average buffer level, figure 22 shows 4 gears (buffer sections) is a balanced choice.

Figure 23 shows G4m performs well. It also shows with margins to avoid frequent gear shift, the representation choice is more evenly distributed. The figure also shows G4m provides higher quality when packet loss is zero. with packet loss, G4m also provides acceptable performance.

Figure 24 shows 6 gears would cause too many switches while 2 gears can compromise adaptation agility. We can also observe that G4m creates less pikes than G4 since the margins among buffer sections prevent frequent gear change and the resulting representation switch.

6.4.2 BufferScaling

The original Gearbox is configured as such that the gear margins are set to be a percentage of the buffersize. Under the original configuration, we reset the receive buffer size to different values, aiming to reveal how buffer scaling can affect the performance of Gearbox.

As shown in figure 25: When the receive buffer is too large, gearbox struggles to fill the buffer to reach higher gears, since the buffer level is designed to stabilize

at Gear 3 where the chosen representation most closely match the link bandwidth. Figure 25 (b) shows too big a buffer size can cause the overall video quality to drop due to the fact that gearbox is prioritized to choose relatively low representations before the buffer is filled to a safe level (in percentage). As shown in figure 26, bigger buffer also causes the download time to be greatly reduced, which is a sign of network bandwidth not fully utilized. As a solution to these problems, when the receive buffer is configured to be large, we can reduce the buffer range of lower gears to acquire better video quality and longer download time.

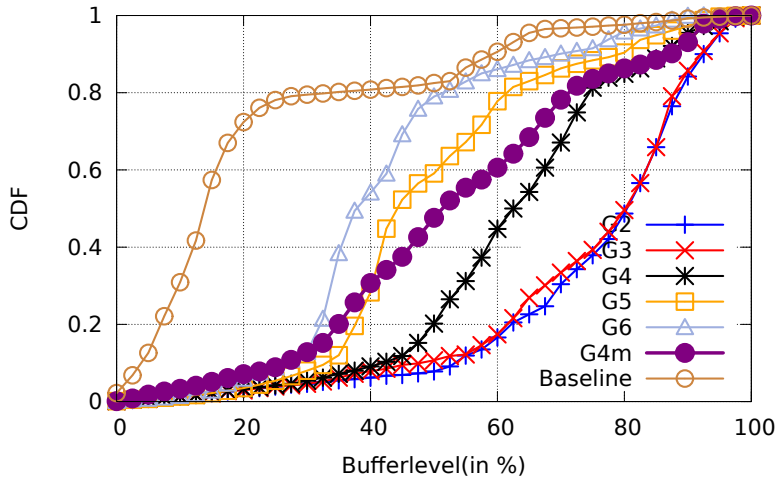
Figure 26 shows the download time of different buffer size settings. When the buffer is set to 10 seconds long, the coverage of each gear is then reduced to a mere 2.5 seconds. This results in more gear changes, leading to more unnecessary representation switches. Furthermore the buffer is too short to hold enough segments, which results in an insufficient safety margin for dealing with network fluctuations. Under this setting, the download time is about 7 minutes and 30 seconds, which is no shorter than the duration of the tested video. In contrast, when the buffer size is set to 40 seconds long, the download time is shortened. When buffersize is 120 seconds, Gearbox takes much less time to finish all segments download, which is an indication of relatively poor bandwidth utilization.

6.4.3 Summary for Gearbox optimization

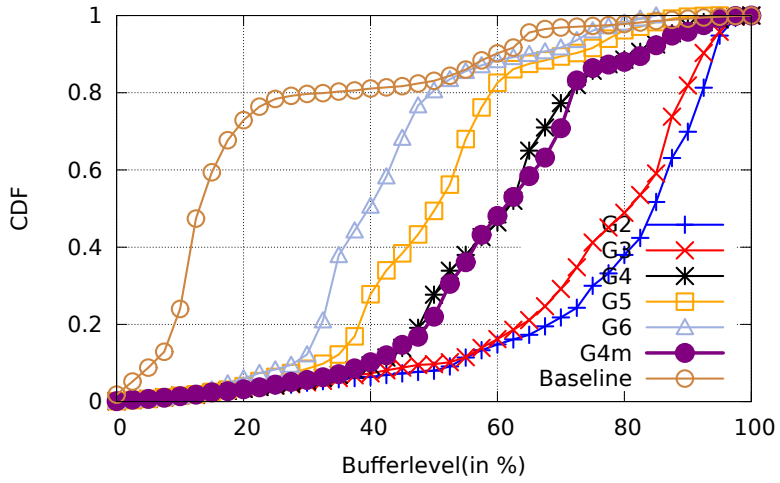
From the Gear allocation tests we can see that the behavior of Gearbox under different configurations is well predictable. Too many Gears would cause more representation switching, affecting the consistency of the video quality. Less Gears, however, would cause insufficient utilization of the available bandwidth. Even though, under all settings, Gearbox proves to perform much better than Baseline.

The results of the buffer scaling test demonstrate how buffer length can affect the rate adaptation. However, Gearbox can always be configured accordingly to fit certain length of the receive buffer. Such configuration may involve adjusting the buffer-covering-range of certain Gears.

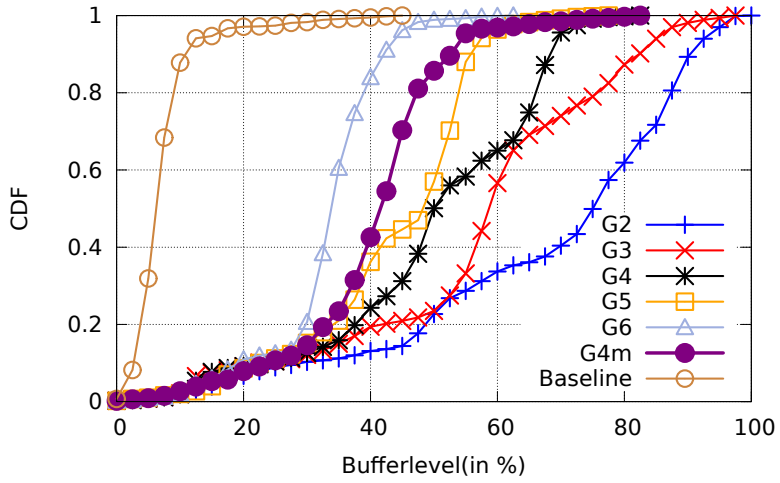
With the help of these test results, we learn how Gearbox behaves under different configurations. It is easy to acquire the desired adaptation characteristics by reconfiguring Gearbox, to fit certain usage scenarios. For example, in a mobile or wireless connection environment, it is beneficial to configure Gearbox to have bigger buffer since the network connection can be lossy and unstable. In live streaming, however, Gearbox can be configured to have smaller buffer size to acquire a more agile rate adaptation, ensuring the smoothness of the live-streaming process.



(a) 10ms_plr0

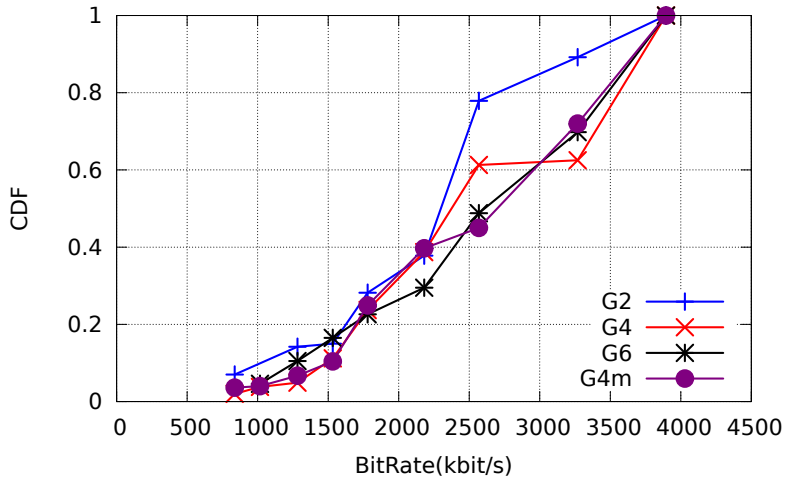


(b) 50ms_plr0

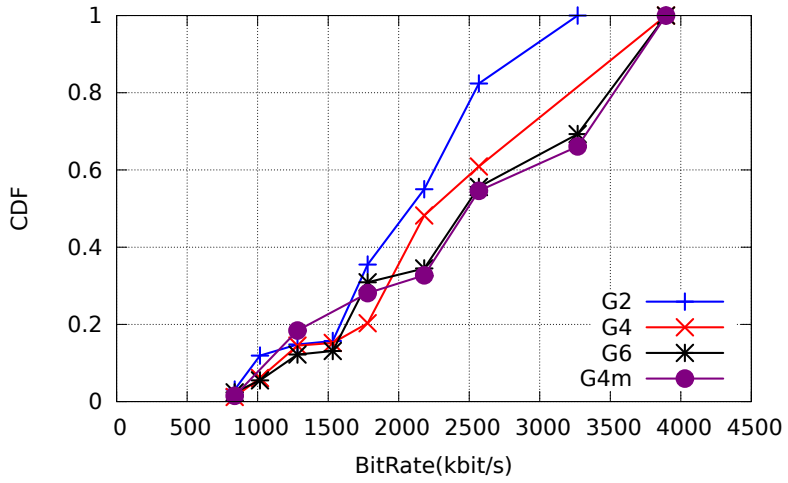


(c) 50ms_plr0.02

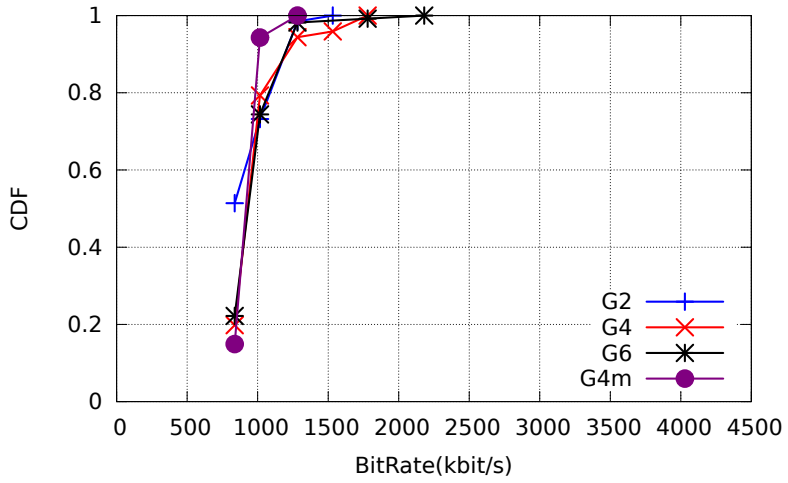
Figure 22: GearAllocationTest_Gearboxprototype_bufferlevel



(a) 10ms_plr0

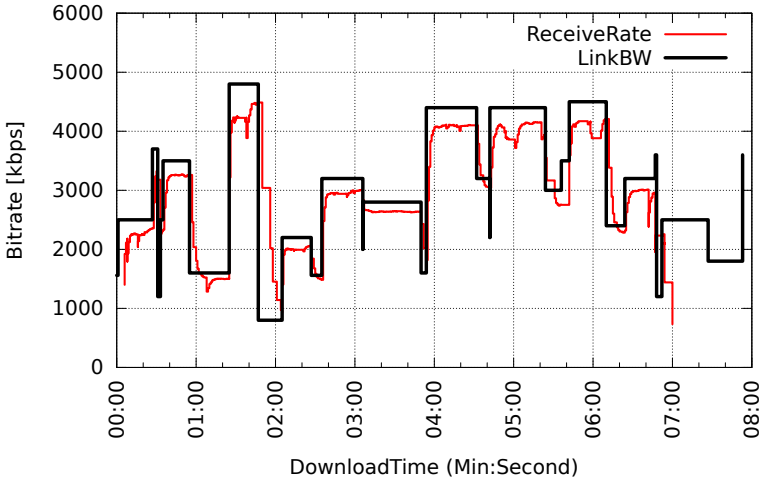


(b) 50ms_plr0

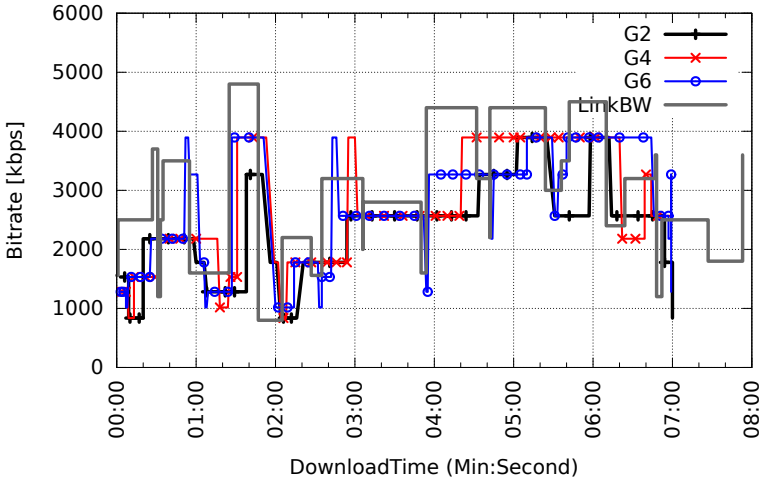


(c) 50ms_plr0.02

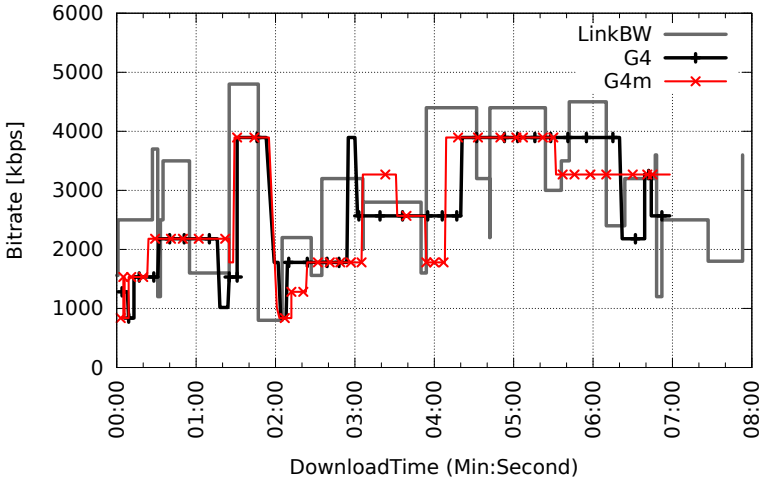
Figure 23: playback rate distribution under each gearbox configuration. For example,G2 represents Gearbox configured to have two gears



(a) LinkRate

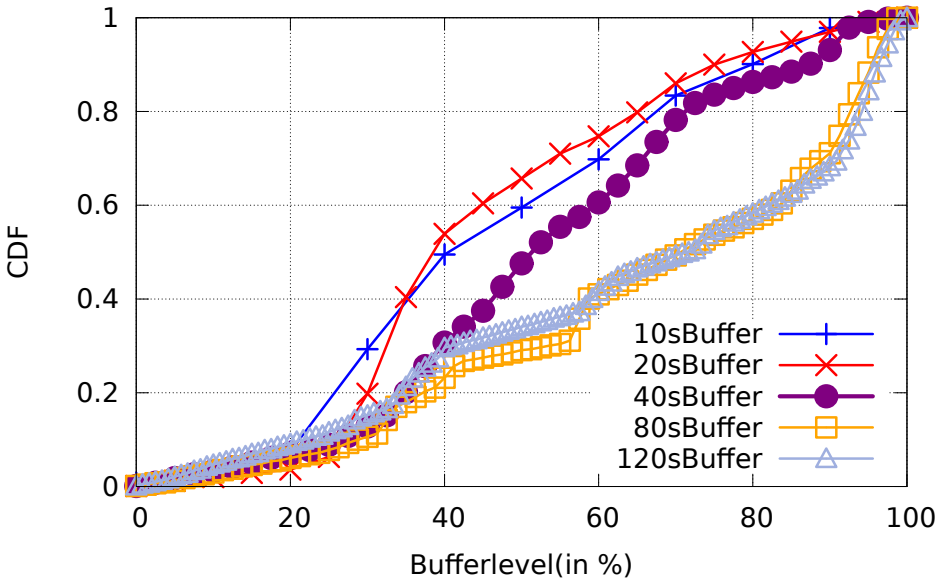


(b) PlaybackRate_GearAllocation

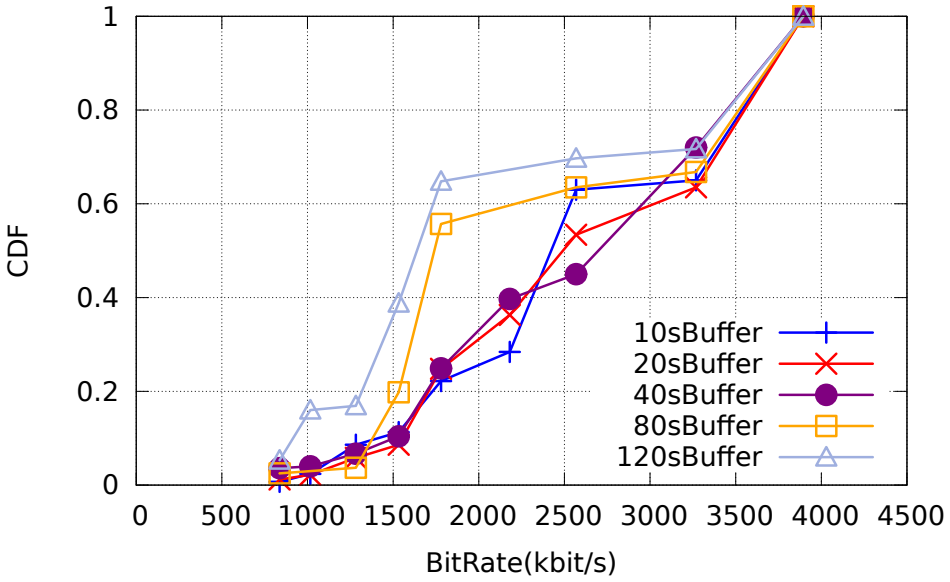


(c) PlaybackRate_G4vsG4m

Figure 24: realtime representation switching, under the link delay of 10ms.

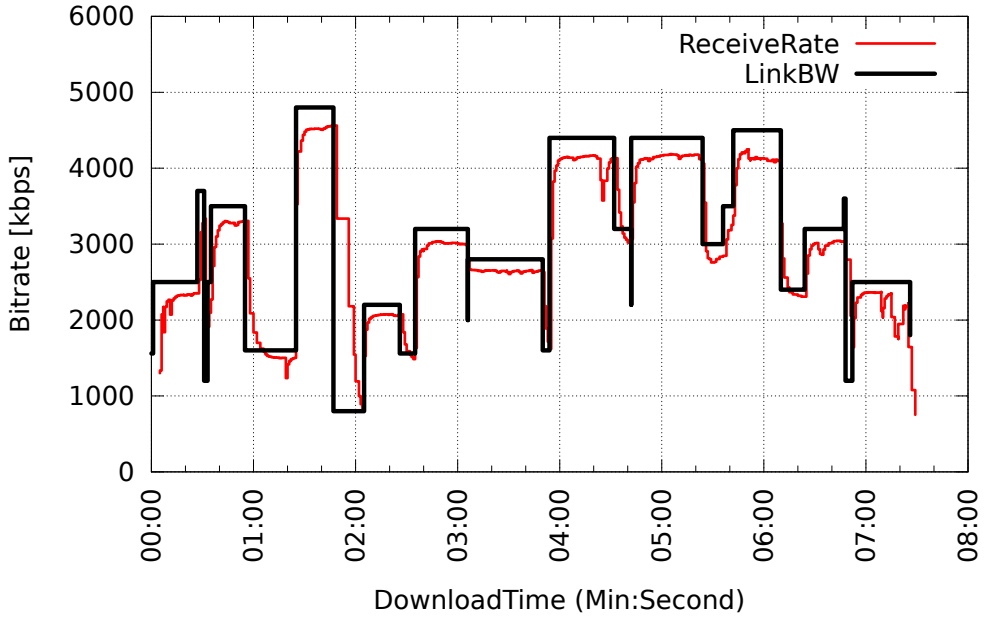


(a) BufferLevelDistribution

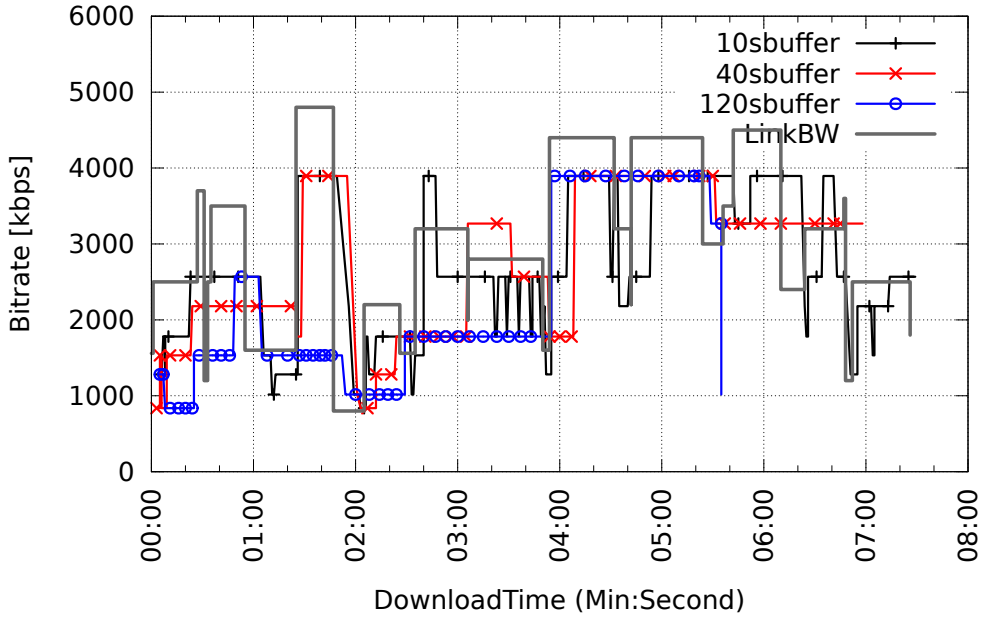


(b) PlaybackRateDistribution

Figure 25: the behavior of Gearbox G4m under different buffer size



(a) LinkRate



(b) PlaybackRate

Figure 26: realtime representation switching under different buffer size. Gearbox is configured to G4m and the link delay is 10ms, the packet loss rate is zero

6.5 Tests with proxy caches

Under this section, we connect the server and end points through two cascaded proxies to evaluate the algorithm under a more realistic network involving proxy caches. Through the following tests, we aim to find out how proxy caches could affect DASH and how our DASH client adapt to the proxy caches, which are usually found in CDNs.

6.5.1 Common settings

The topology of the test bed setting in this section is shown in figure 27; The common settings of all following tests are also listed below:

- The DASH-JS source code is configured to construct the cache control directive of the HTTP requests to be "public", so that caches along the chain are allowed to cache the requested content.
- At the two client end points, Google-chrome is used as the DASH client. Since Google-chrome has an internal cache which could not be turned off, we installed a third party application "Cache killer" [11] to facilitate our tests. Cache killer dumps any data cached by Google Chrome and makes sure no browser cache would affect our tests on the proxy caches. In this way, the browser cannot cache any content any more and multiple browser tabs (HTTP client instances) could only receive content from proxy caches rather than from the browser's internal cache.
- As shown in figure 27, link 1 is set to be a static link, with both upstream link and downstream link set to bandwidth of 20Mbit/s and delay of 75 ms. Thus the RTT is 150 ms. link2 is also set to be a static link with both upstream link and downstream link set to bandwidth of 20Mbit/s and delay of 50 ms. Thus the RTT is 100 ms; The two client end points are connected to the level 1 proxy cache. As shown in figure 27, for both link3 and link4 we set the download link to 54Mbit/s and the one way delay is 10 ms. These link settings create a bottleneck between the origin server and the edge servers, mimicking the situation of the CDN network, where the bottlenecks (latency and available bandwidth) reside in the far end of the network and end point clients must rely on the cached content from the closest edge server.
- Proxy cache level 1 is set to peer with proxy cache level 2 and cache level 1 cannot directly request content from the server. Both proxies are squid3 cache proxies and we do not enable cache digest changing in our experiment. All requests received by P1 will be directed to P2.
- We limit the disk cache space on both P1 and P2 to 400MB, and the memory cache size is set to be 256 MB on both machines. In all we have a total of

[11] https://chrome.google.com/webstore/detail/cache-killer/jpfbieopdmepaolggioebjmedmclkbap?hl=en

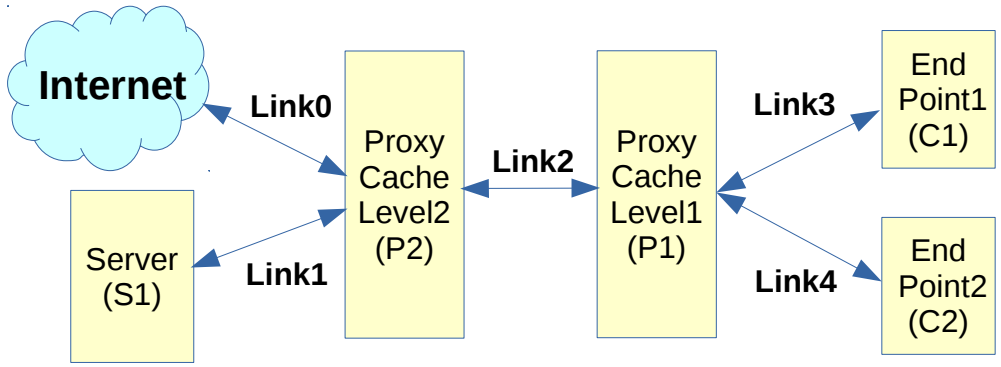


Figure 27: Testbed setup for evaluating performance of DASH congestion control in the presence of cascaded caches.

656MB cache capacity available on each machine. Limiting the cache size is necessary since we need to learn how obsolete content can be purged when cache storage is full. In our case, the video stream we test on, in highest bit rate, is about 250MB and with our competing videos requested from both End point 2 and proxy cache level 1, we can make sure that highest cache ratio of our target content cannot be 100 percent. Maximum cache object size is configured to 1024KB so that all Dash segments can be cached. (Our Dataset show that with 1s segment size, the largest segment is less than 700KB)

- The memory cache replacement policy is left as default, which is LRU. We however set disk cache replacement policy to heap LFUDA and LRU respectively,in each test round, to make comparisons on LFUDA and LRU.
- Before each round of test, we clear the proxy caches so that each test starts with empty cache.
- Figure 27 shows the topology of our test bed. This setting applies to all following tests.

To evaluate the tests we record the data collected from each client instance. The data includes the estimated bandwidth, the chosen representation bit rates, the buffer level of DASH-JS receive buffer, the Gear level when Gearbox is the active adaptation algorithm, the cache hit ratios at both P1 and P2.

6.5.2 Test Senario1: Even arrival with no cross traffic

This test scenario aims to reveal how the hit ratios of the proxy caches change when clients evenly arrive in time. In this scenario we use only End Point 1 to request content from the server, via the cascaded proxies. No cross traffic is applied to any machines. We use a shell script to open up browser tabs that all request the same content “Of forest and men” [8]. The resolution is still 1280 × 720 and the available representation sets are the same as our micro benchmark tests. Through

Client No.	Switching Count	Underflow Count	Underflow time	HitRatio P1	HitRatio P2
1	5	3	18.5s	0%	0
2	22	1	10.8s	67%	0
3	21	1	1.4s	79%	0
4	17	0	0s	79%	0
5	16	0	0s	88%	0
6	22	0	0s	87%	0
7	20	1	2.2s	80%	0
8	23	0	0s	84%	0
9	22	1	2.5s	86%	0
10	18	0	0s	65%	0

Table 4: Statistics_EvenArrival_Gearbox

script-automation, we open up one browser client instance every 30 s. 10 instances are opened up in total. Since all client instances are initiated by the same browser in the same client machine, they must compete the shared link, which has a bandwidth limit of 54Mbit/s, with a one way delay of 10 ms. DASH-JS is configured to activate the Gearbox algorithm.

In table 4 we find that when there is no competing traffic, clients that arrive later would generally get higher cache hit ratio. We can observe that the last client experiences lower hit ratio than previous clients. This is because all other 9 clients has finished there downloading roughly 30 s ago (clients are launched at a 30 s interval). With least competition, the last client is then free to use all available bandwidth and would surely increase the video bit rate by requesting higher level representations, which are not fully cached by the proxies. We can also see that clients in the middle have similar hit ratios since their observed network condition is similar throughout the entire downloading process.

After the 10 clients have finished, we launch another two individual clients, requesting the same content.

These two clients are launched one by one, meaning the 12th client is not launched until the 11th has finished its streaming. The 11th client uses the Gearbox algorithm as the previous 10 clients did while the 12th client uses the baseline algorithm. These two clients uses a reconfigured link 3, which provide 3.5Mbps of bandwidth and 10 ms one way delay. This link configuration ensures that the available bandwidth matches the middle value of the available media bit rates, enabling the tested algorithms to perform meaningful rate adaptation. In table 5 we find that the 12th client, which runs the baseline algorithm, get 1 percent of cache hit in cache level 2, instead of 0 hit. One explanation is that the baseline algorithm performs much more representation switches than Gearbox does, resulting in an increased possibility of cache hit in the far end cache.

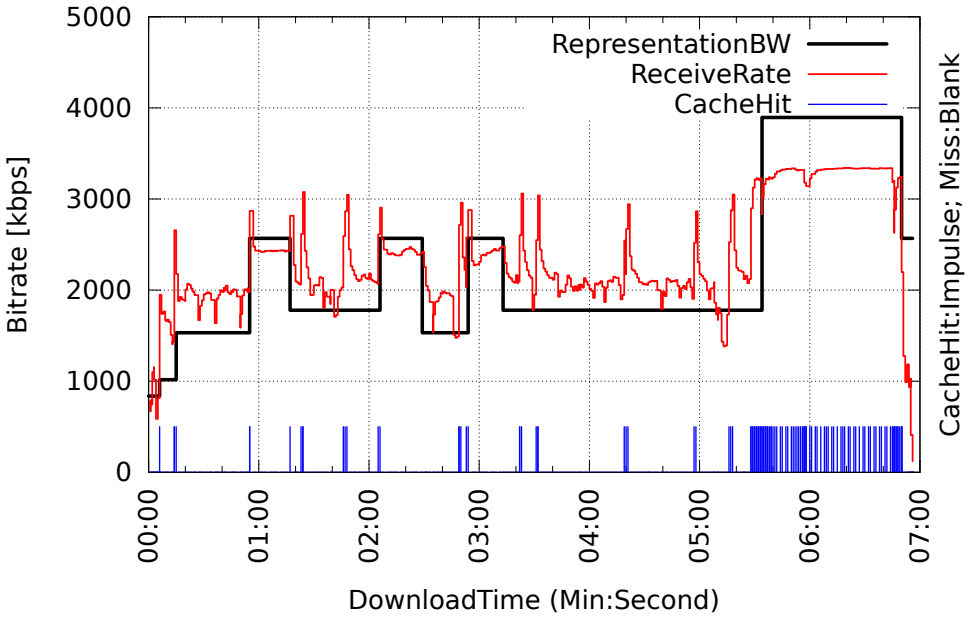
Client No.	Switching Count	Underflow Count	Underflow time	HitRatio P1	HitRatio P2
11	10	0	1.2s	27%	0
12	168	3	14.8s	20%	1%

Table 5: Statistics_SingleClients_WithNoCompetition

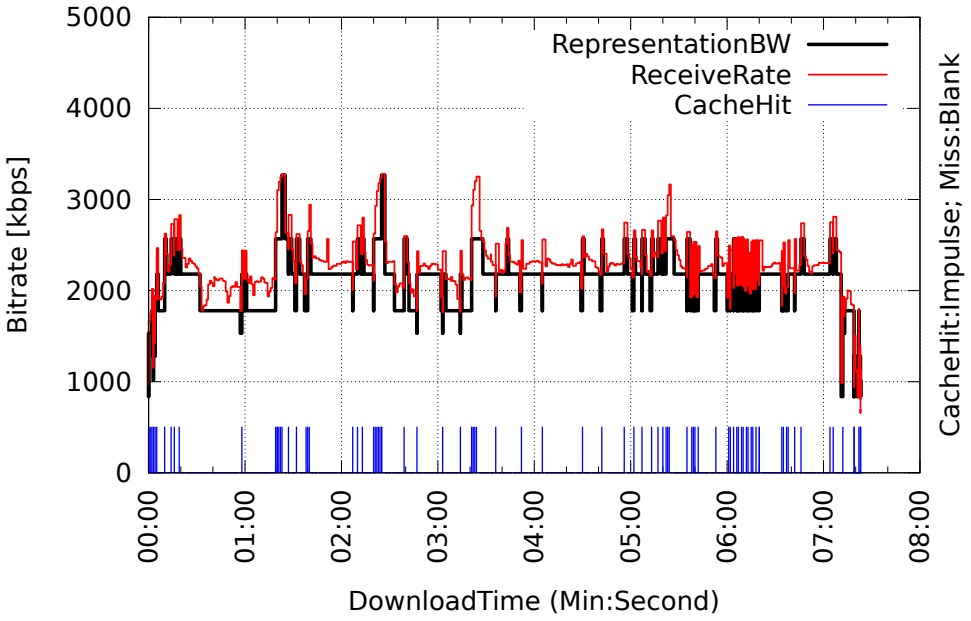
The real time download plots and CDF plots are shown in figure 28 and figure 29: Figure 28 shows that the baseline algorithm reacts poorly to cache hits. When a cache hit occurs, the client sees a dramatically increased bandwidth (Receive rate). With no buffer level taken into consideration, the baseline algorithm blindly chooses to request a high bit rate segment. If this segment to be fetched happens not be in the proxy caches, then the client buffer level would simply drop due to the fact that link throughput do not really surpass the bandwidth of the requested segment. In light of this, the more discrete the cache hit, the more likely the Baseline algorithm would create a buffer starvation. Plot (b) of figure 29 shows the bandwidth range of the chosen representations in baseline is narrower, which could imply a stable video rate. However plot 28 (b) and plot 29 (a) reveals that the constant representation switching by the baseline client causes many buffer underflows.

The buffer underflows in the baseline instance happen partly because the bit rate of a representation is not really the bit rate of a segment. To maintain the desired buffer level, the same representation must be chosen for a number of successive segments. Only in this way can the overall bit rate of the stream, within a period of time, stay at a wanted level, resulting in a relatively stable receive buffer level. In this sense,the adaptation algorithm of the baseline client is not designed effectively to maintain the representation level when necessary, in order to avoid buffer level fluctuations and the resulting buffer starvation.

Even though Gearbox also see some representation switches, most of these chosen representation levels are able to be held steady for at least 20 seconds, thanks to the design of the gear margins. Overall, gearbox performs well in that it does not easily get interrupted by occasional cache hits. The adaptation behavior of Gearbox is closely related to the fill level of its receive buffer, which can well reflect the real network throughput. In figure 28 and figure 29 we can clearly see that Gearbox adapts to the proxy caches well. It provides bigger safe margin, higher overall video quality and less representation switching and video quality fluctuations. Last but not the least, the adaptation behavior of Gearbox can benefit the proxy cache and latterly arrived clients, since it does not perform representation switching prematurely.

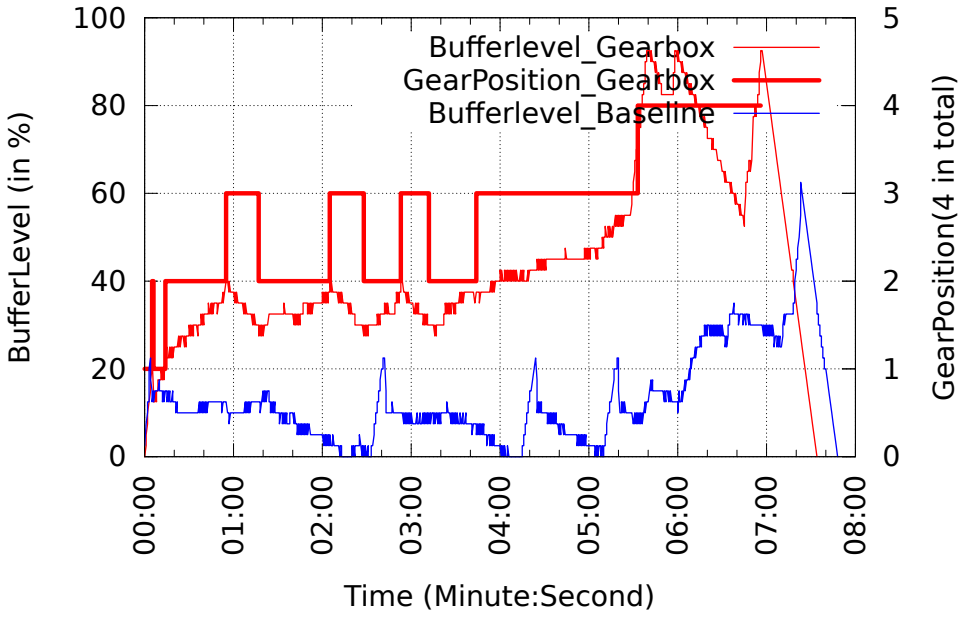


(a) throughput_3500Kbps_Gearbox

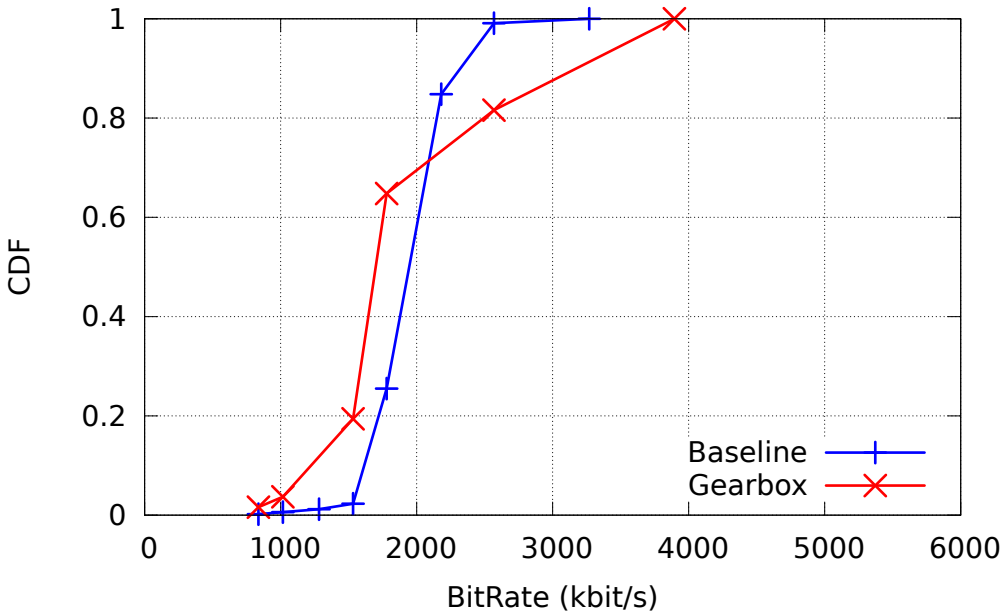


(b) throughput_3500Kbps_Baseline

Figure 28: Test Senario 1: after 10 Gearbox clients' even arrival,two more single clients are opened one at a time, under the same link condition. They are named as the 11th and the 12th client, which is a gearbox client and a baseline client respectively. under comparable hit ratio, the figure shows the different behavior of the two algorithm.



(a) bufferlevel_3500Kbps



(b) Rate

Figure 29: Test Senario 1: after 10 Gearbox clients' even arrival,two more single clients are opened one at a time, under the same link condition. They are named as the 11th and the 12th client, which is a gearbox client and a baseline client respectively. under comparable hit ratio, the figure shows the different behavior of the two algorithm.

6.5.3 Test Senario2: Poisson arrival with cross traffic

To test the performance of both algorithms under a more realistic network condition, we set up Test Scenario 2. Under this scenario, we introduce noise traffic from C2 and P2 and C1 would serve as our test client machine (please consult figure 27). All simulation results are collected at C1.
At P2, we use a linux shell script to open up browser instances that comes at a Poisson arrival pattern. The average inter arrival time is set to be **40 s** 14. Each instance opens up a distinct URL to request content ranging from a dynamic web page to a Youtube video. All these requests from the browser in P2 are redirected through the squid3 proxy server resides in P2, to create a competition at Proxy Cache Level2. Link 0 is not limited by Dummynet pipe and squid3 on P2 is allowed to directly access the Internet.
At C2 and C1, we open up browser instances that also comes at Poisson arrival pattern. The average inter arrival time is set to be **30 s**. At C2, the requested content includes web pages from the Internet as well as several video streams that served by our server S1. Instead of requesting the video clip "Of forest and Men", C2 instances create competitions at S1 by requesting different content, which includes "Big buck bunny" [8].
At C1, we open up 10 clients that arrives at Poisson arrival with 30 s average interval. The inter arrival time of the 10 clients are shown as follow: 4s, 17s, 28s, 70s, 6s, 48s, 4s, 80s, 24s. The pattern is plotted in figure 14. All 10 clients request the video clip "Of forest and men" under the resolution of 1280×720. The available representations are the same as the ones in previous tests. All common settings in section 6.5.1 also apply to Test Scenario 2.

(A) First we perform the test with the cache replacement policy for the squid3 proxies configured to heap LFUDA [12]. Again, caches along the chain are cleaned before each test round. We test the performance of the baseline algorithm in the first round and Gearbox the second round.

Client No.	Switching Count	Underflow Count	Underflow time	HitRatio P1	HitRatio P2
1	146	16	189.9s	0%	0%
2	307	22	366.8s	13%	0%
3	316	22	398.6s	17%	0.9%
4	279	21	389.7s	47%	0.2%
5	258	18	310.5s	45%	0.1%
6	249	17	300.8s	46%	0%
7	190	16	258.8s	61%	0.9%
8	179	6	64.4s	50%	1.1%

Table 6: Baseline_LFUDA_Poissonarrival

[12] http://www.squid-cache.org/Doc/config/cache_replacement_policy/

Client No.	Switching Count	Underflow Count	Underflow time	HitRatio P1	HitRatio P2
1	9	3	21.2s	13%	0
2	24	2	18.8s	65%	0
3	31	2	4.7s	59%	0
4	34	0	0s	75%	0
5	23	0	0s	88%	0
6	30	1	9.8s	87%	0
7	28	0	0s	89%	0
8	22	0	0s	80%	0
9	35	1	0.8s	60%	0
10	29	1	15.7s	66%	0.4%

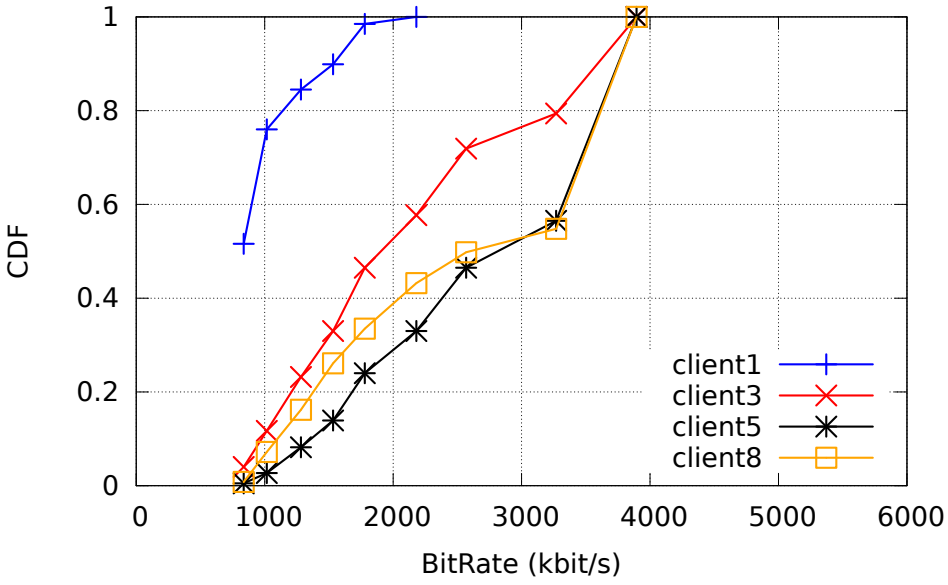
Table 7: Gearbox_LFUDA_Poissonarrival

Because of the original bug of DASH-JS [37], two baseline clients get stuck during playback and fails to finish the video playback. In table 6 we can see that latterly opened clients see an increased cache hit ratio at P1. Since the clients are all competing for the 54Mbit/s Link 3, severe buffer underflow happens for all clients. Table 6 also shows that the baseline algorithm does not benefit from cache hits in that the underflow time does not necessarily shortens when cache hit ratio rises up. The last coming client does see a much shorter underflow time, since all other clients have finished their download before the last client finishes, leaving all available bandwidth to the last client.
Table 7 shows the test result for Gearbox. One thing worth mentioning is that the second client is initiated 4s later than the first one. Since the starting time of the first two clients are very close and both encounter buffer underflow over 2 times, their actual downloading and playback time exceeds the duration of the the video. From our raw data we find the download of both clients finished at Unix time 1411336184 second since the Unix Epoch. This explains why the first client actually get some cache hit. These cache hits are contributed by the client 2 whose download occasionally get ahead of client 1. This situation can happen since client 2 benefit from the cached segments that were previously requested by client 1. The downloading process of client 2 was accelerated to the point that it surpasses the downloading process of client 1.

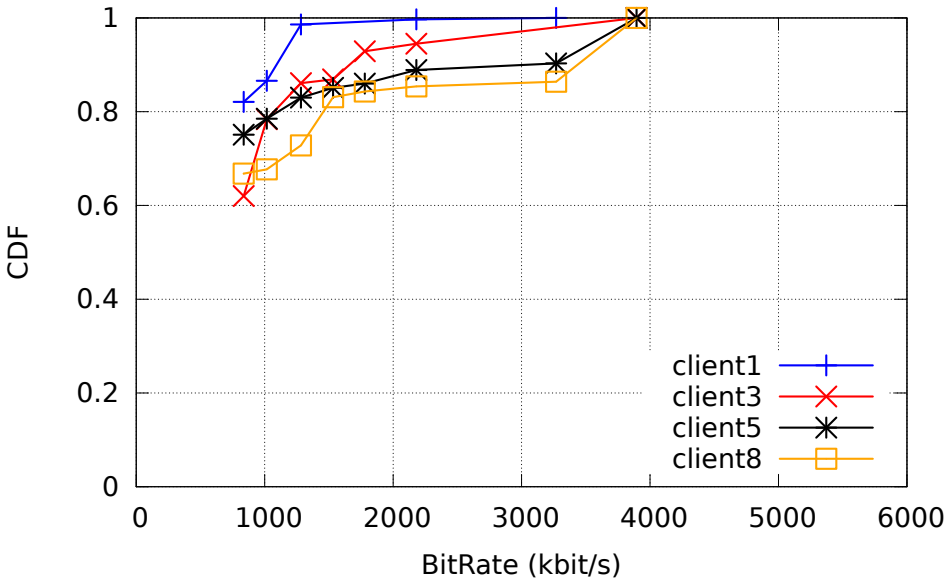
Overall we find that latterly initiated clients generally see higher cache hit ratios. The cache hit ratio tops at 89 percent, which is 28 percent higher than the highest hit ratio of the baseline test round. We can also see clearly that Gearbox does generally benefit from cache hits, since higher cache hit ratios seem to shorten the buffer underflow times.Gearbox client 6 is an exception and this is possibly due to the fact that it is initiated in the middle and experiences the worst congestion.

Figure 30 shows the CDF of the playback bit rates. Only client 1, 3, 5, 8 of the Gearbox and Baseline tests are included in the CDF. These two plots shows

that under fierce competition when bandwidth is limited, Gearbox choose lower bit rate segments to download, in order to avoid playback pauses (buffer starvations). In table 6 and table 7 we can see clearly that buffer underflow counts of Gearbox clients are significantly lower than that of the baseline clients.



(a) BaselineRate



(b) GearboxRate

Figure 30: Test Scenario 2, Case A. (a): Representation rate CDF of baseline clients following Poisson-arrival, under proxy cache policy of LFUDA. (b): Gearbox clients at the same arrival pattern under LFUDA

We also find in table 6 and table 7 that client 7 of the baseline test and client 9 of the Gearbox test have the same cache hit ratio of about 0.60. Since these two clients are also both the second-to-last-arriving client in their own test round, they serve as good samples for comparison. We refer to them as B_7 and G_9 respectively. We draw the download-time-plots and CDFs of B_7 and G_9 in figure 31, to make analysis:

In figure 31, plot (a) shows the real time buffer level of B_7 and G_9. Under the same cache hit ratio of 60 percent, plot (a) shows that under a more realistic network condition with limited bandwidth and fierce TCP traffic competition, baseline algorithm performs terrible. The video downloading time of B_7 stretches to almost 12 minutes, which is over 4 minutes longer than the video duration. G_9, however, performs well under comparable condition.

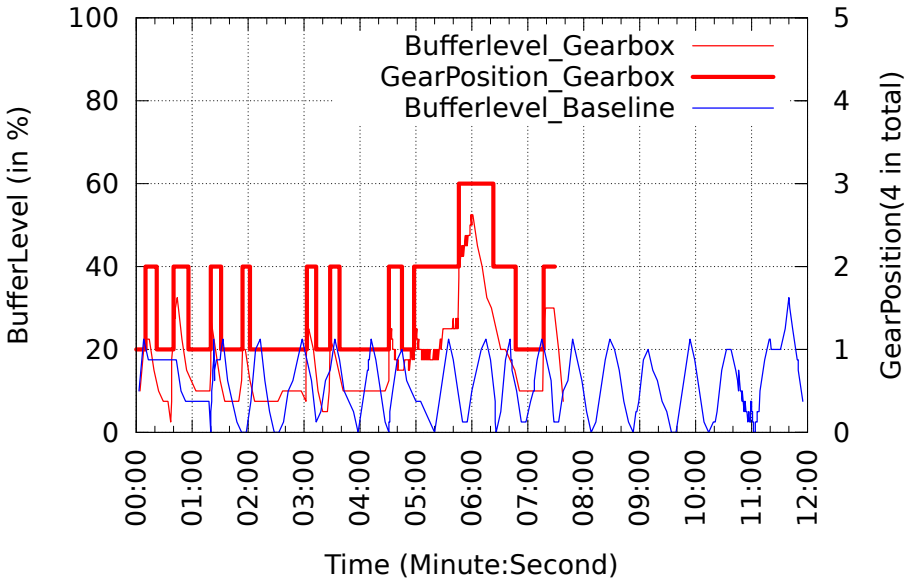
(B)Then we run the test with the cache replacement policy set to the default LRU, in order to reveal the performance of DASH under a different caching policy. The test round under LRU starts with an emptied cache chain. Except for the change in cache replacement policy, other settings are exactly the same as test (A).

Comparing table 8 with table 7, we clearly see that LRU contributes lower hit ratio, resulting in more buffer underflow occurrences. Compared with the LRU policy, heap LFUDA tries to purge small files out of the cache. Consequently LFUDA makes more space for caching bigger files, such as the DASH video segments. From this finding we can conclude that LFUDA better suits video content caching in DASH, compared with LRU.

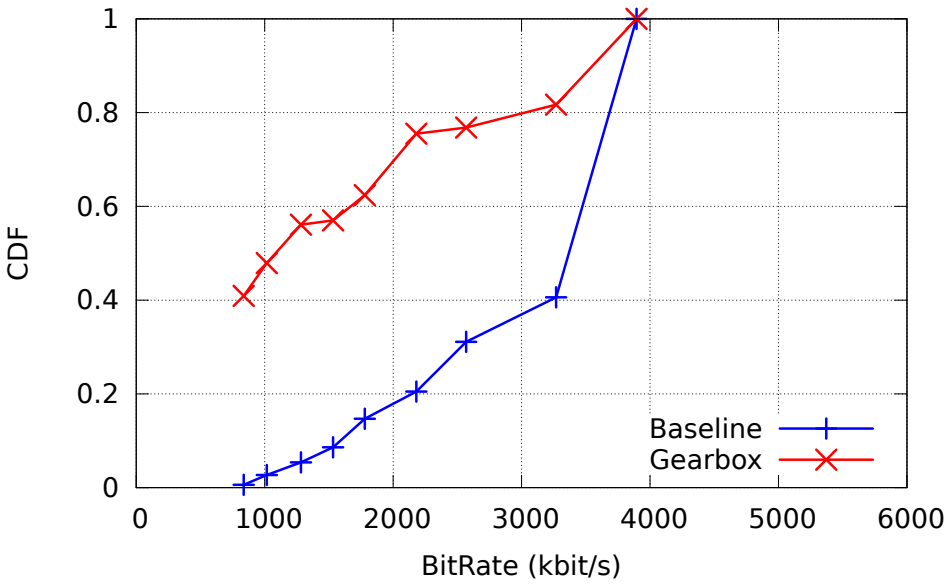
Again, 5 clients running the baseline algorithm encountered stuck-playback and failed the test due to the bug of the original DASH-JS [37]. In table 9 we can see that the first two clients, which arrive at an time interval of 4 seconds, experience the same problem as in previous tests, which is that the first client can get more cache hits. However the extra cache hits do not benefit the baseline client at all, judging from the fact that the downloading time of the first client is significantly longer than the second one. This shows that an incompletely cached representation have a fatal impact on the baseline algorithm, which performs representation selections solely based on the estimated bandwidth. Discontinuous cache hits cause the estimated bandwidth to fluctuate frequently and dramatically, causing the Baseline algorithm to make premature representation switches, which in turn lead to buffer underflow occurrences and streached download time.

Figure 32 shows the download-time-plots of client 1 and client 2 of the baseline test. Instead of offsetting the time lines to align the starting time of the two clients, we use the original recorded files to generate the plots using Unix-time. In this way we are able to observe how the first-arriving client ended up finishing the download almost two minutes later, after the second-arriving client had finished its download.

In short, figure 32 clearly reveals how a DASH client could suffer from cache hits, when its rate adaptation algorithm is designed without the awareness of the proxy caches and CDNs.



(a) bufferlevel



(b) Rate_CDF

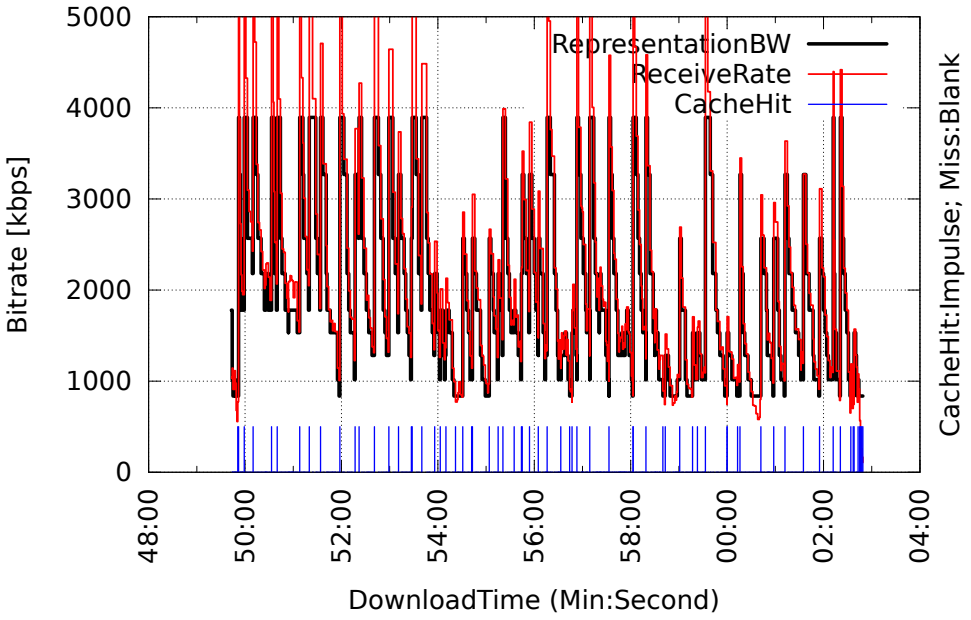
Figure 31: Test Scenario 2, Case A: the 7th Baseline client and the 9th gearbox client have comparable hit ratio of 60 percent. (a) shows how the two algorithm behave under a very stressful network condition, where multiple clients compete for the same link and content. (b) shows how Gearbox choose lower bit rate to ensure the continuity of the video playback

Client No.	Switching Count	Underflow Count	Underflow time	HitRatio P1	HitRatio P2
1	22	7	75.9s	45%	0
2	12	7	69.3s	31%	0
3	30	5	45.3s	59%	0
4	35	4	19.3s	71%	0
5	36	3	24.4s	76%	0
6	32	2	13.6s	79%	0
7	34	2	15.8s	72%	0
8	30	1	7.8s	80%	0
9	39	1	5.8s	81%	0
10	38	0	0s	60%	0

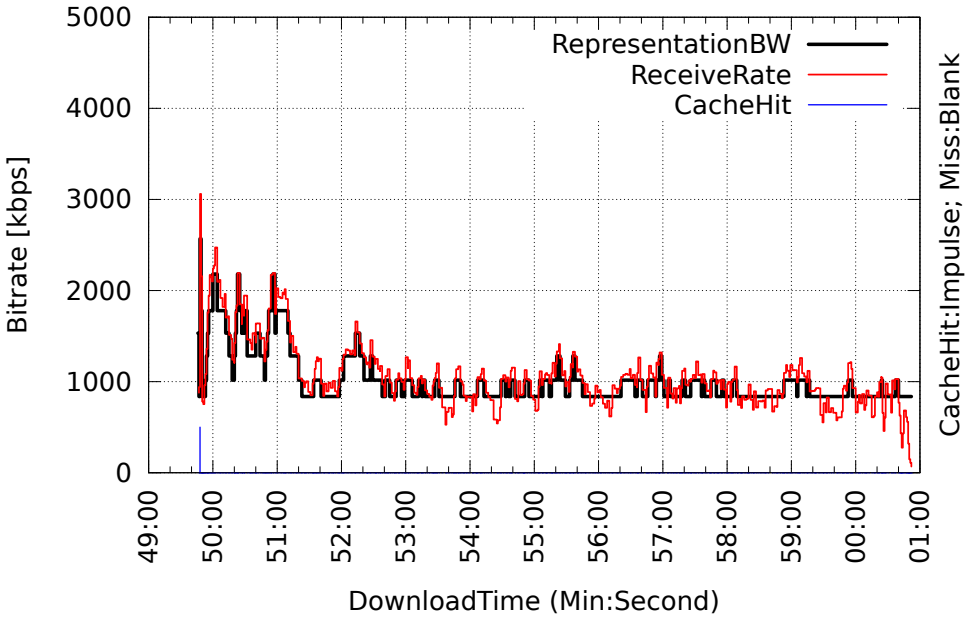
Table 8: Gearbox_LRU_Poissonarrival

Client No.	Switching Count	Underflow Count	Underflow time	HitRatio P1	HitRatio P2
1	299	21	340.6s	15%	0
2	122	16	217.6s	0.2%	0
3	311	22	92.7s	23%	0
4	291	18	310.3s	17%	0
5	176	7	414.4s	57%	0

Table 9: Baseline_LRU_Poissonarrival



(a) baselineClient1_LRU



(b) baselineClient2_LRU

Figure 32: We draw this figure to show how cache hit can worsen the experience of a baseline client. The first coming client can actually have more cache hit than the latter client. Cache hits caused the first client to make many premature representation switches, leading to buffer underflows. As a result, the first client takes two more minutes than the second client, to finish downloading

6.6 Summary of cache tests

By analyzing these test results we can draw the following conclusions on the adaptation algorithm of DASH:

- It is a bad design for a rate adaptation algorithm to make representation-switching-decisions solely based on the observed bandwidth. Such an adaptation algorithm could suffer from bandwidth fluctuations and cache hits from CDN edge servers, causing frequent video-quality-changes and unpleasant user experience. Although the agility of the rate adaptation can be adjusted through changing the bandwidth-evaluation-method, to achieve fewer representation-switches and stabler video quality, there is still no guarantee that the DASH client can be spared of buffer starvation and the resulting playback interruptions. A mechanism, such as buffer monitoring, shall be introduced in the rate adaptation algorithm, to improve the performance of DASH in CDNs.

- When the rate adaptation algorithm of a DASH client is not properly designed, the incomplete-cached-representations, instead of benefitting the client, could degrade the client performance and worsen the user experience by causing worse video quality fluctuations and more playback interruptions.

- A carefully designed DASH adaptation algorithm not only benefits the user experience but increases the cache hit rate as well, since an adaptation algorithm that maintains stabler video bit rate can naturally create a more complete cached-representation in the CDN edge servers (proxy caches).

- The proposed Gearbox algorithm provides an adaptation algorithm model where the behavior of the adaptation can be easily configured to fit different requirements. Taking the real-time buffer level into consideration when performing rate adaptation seems to be a simple and effective solution for improving the performance of DASH in complex network conditions.

- As far as the cache-replacement-policies of the squid server[13] is concerned, heap LFUDA provides higher cache-hit-ratio than the more often used LRU in the case of DASH streaming. This is because video segments in DASH streaming are typically big in size and LFUDA inclines to cache big and popular objects rather than the smaller ones.

- DASH segments can be easily cached by proxy servers due to the fact that these segments are no different than normal HTTP objects. The cache hit rate usually climbs up as more clients arrive and request the same content via the proxy cache. Judging from the overall cache hit rates, we can confidently say that CDN edge servers can efficiently cache the DASH content, without having to meet any special system requirement.

[13] http://www.squid-cache.org/Doc/config/cache_replacement_policy/

7 Conclusion

In this thesis, we have introduced the HTTP streaming technologies and have conducted a in-depth research on Dynamic Adaptive Streaming over HTTP. Various aspects of HTTP Streaming have been studied, including the basics and the special features of HTTP, the supporting CDN infrastructure, and the general architecture of typical HTTP streaming systems. Live streaming and on-demand streaming, as the two different HTTP streaming services, are also compared. With the comprehension of these mentioned subjects, we have also introduced the history of the HTTP streaming technologies, offering a big picture of how HTTP streaming has been evolving over time. With the new paradigm of HTTP Adaptive Streaming widely accepted and adopted, we have concluded that the new MPEG-DASH standard will reshape the mainstream HTTP streaming technology in the future. Serving as a standard that unifies the format of the DASH content segments and that of the MPD manifest file, MPEG-DASH is able to bridge different HTTP streaming servers and clients together, solving the compatibility issues often seen in the proprietary HTTP streaming solutions.

From our research findings we have concluded that, currently, DASH is still faced with challenges posed by CDN caches. Although there have been proposals to investigate and address these challenges, those proposals mostly involve the introduction of new protocols and new infrastructures, such as CCN, or the alternation and extension of existing protocols. More often than not, the proposed solutions would also require the cooperation of the CDN caches with the streaming clients, which inevitably generates extra signaling between the clients and the caches. To meet the need of improving the performance of DASH in a CDN network, we proposed our solution that solely optimizes the DASH client-side rate-adaptation mechanisms, in order to achieve the same goal of solving the cache challenges without making any change to the existing CDN servers and DASH standards. By simply binding the rate adaptation behavior with the receiving buffer level, we have been able to implement our effective Gearbox rate adaptation algorithm. Performance tests conducted in this thesis have shown that our algorithm design can effectively improve the performance of the original DASH-JS client, making it much more cache friendly.

7.1 Future work

For practical reasons, it is challenging for our research team to do exhaustive tests on our Gearbox algorithm. There remains a number of interesting emulation tests we would like to run and these tests and experiments may be included in our future work.

Specifically, the future work may involve testing the performance of our algorithm against other algorithm proposals. It would also be interesting to test our Gearbox algorithm with different DASH contents and different segment durations, so that the robustness of the algorithm design can be further checked. More user scenarios could also be devised, to expose possible flaws of the Gearbox algorithm design.

7.2 Criticism

DASH is an effective technology for streaming multimedia content over HTTP. It holds certain advantages over the traditional RTSP/RTP streaming. However, in building our DASH testbed and using the DASH datasets, we find that DASH could create a big waste on storage space. DASH requires the original video stream to be encoded into multiple streams with different bit rates. To fit different usage situations, these streams might be further divided into segments of different lengths. Each of the segmented version of the stream would form a unique copy of the original stream. Thus DASH would have an original video stream remade into many different versions (representations) with each version holding a unique bit rate and segment size. Each version of these segmented streams would have to take its own storage space. Taking the DASH dataset "Of Forest And Men" [8] for example, a mere 7 minutes footage, with multiple representations presented, can take up over 7 GB of storage space at the server side. All these segments from the multiple representations can flood the CDN network and create huge amount of fragments in the proxy caches. If the DASH client were not designed properly, these cached segments in the CDN network would easily grow into very discrete, useless, and fragmented garbage, taking huge amount of network storage and causing performance degradation. In my opinion, there could be a room for improving the storage efficiency of DASH content. However, storage nowadays is very cheap. In this sense, efforts made to reduce the storage requirement of DASH could be more costly than the purchase of more storage space.

References

[1] S. Akhshabi, S. Narayanaswamy, A. C. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptive video players over http," *Signal Processing: Image Communication*, vol. 27, no. 4, pp. 271–287, 2012.

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

[3] A. Fecheyr-Lippens, "A review of http live streaming," *Internet Citation*, pp. 1–37, 2010.

[4] I. Sodagar, "The mpeg-dash standard for multimedia streaming over the internet." *IEEE Multimedia*, no. 18, pp. 62–67, 2011.

[5] S. Kellicker, "Html5 and media source extensions," http://www.wowza.com/blog/html5-and-media-source-extensions, 2013, accessed: 2014-12-4.

[6] "Information technology – coding of audio-visual objects," ISO/IEC 14496-12, 2008.

[7] B. Rainer, S. Lederer, C. Muller, and C. Timmerer, "A seamless web integration of adaptive http streaming," in *Signal Processing Conference (EUSIPCO), 2012 Proceedings of the 20th European*. IEEE, 2012, pp. 1519–1523.

[8] S. Lederer, C. Müller, and C. Timmerer, "Dynamic adaptive streaming over http dataset," in *Proceedings of the 3rd Multimedia Systems Conference*. ACM, 2012, pp. 89–94.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," June 1999, rFC2616. [Online]. Available: http://tools.ietf.org/rfc/rfc2616.txt

[10] "Http(hypertext transfer protocol) basics," https://www3.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html, accessed: 2014-11-15.

[11] B. Krishnamurthy, J. C. Mogul, and D. M. Kristol, "Key differences between http/1.0 and http/1.1," *Computer Networks*, vol. 31, no. 11, pp. 1737–1751, 1999.

[12] Y. Tang, X. Li, Y. Liu, C. Liu, and Y. Xu, "Review of content distribution network architectures," in *Computer Science and Network Technology (ICCSNT), 2013 3rd International Conference on*. IEEE, 2013, pp. 777–782.

[13] H. A. Tran, A. Mellouk, S. Hoceini, and J. Perez, "User-centric content distribution network architecture," in *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2012 4th International Congress on*. IEEE, 2012, pp. 343–350.

[14] N. C. Zakas, "How content delivery networks (cdns) work," http://www.nczonline.net/blog/2011/11/29/how-content-delivery-networks-cdns-work/, 2011, accessed: 2014-12-3.

[15] K. Young, "A beginner's guide to http cache headers," http://www.mobify.com/blog/beginners-guide-to-http-cache-headers/, 2013, accessed: 2014-11-14.

[16] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 4, pp. 3–11, 2000.

[17] S. Akhshabi, A. C. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http," in *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, ser. MMSys '11. New York, NY, USA: ACM, 2011, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/1943552.1943574

[18] T. Kim and M. H. Ammar, "Receiver buffer requirement for video streaming over tcp," in *Electronic Imaging 2006*. International Society for Optics and Photonics, 2006, pp. 607 718–607 718.

[19] C. Liu, M. M. Hannuksela, and M. Gabbouj, "Client-driven joint cache management and rate adaptation for dynamic adaptive streaming over http," *International Journal of Digital Multimedia Broadcasting*, vol. 2013, 2013.

[20] Adobe, "Http dynamic streaming on the adobe flash platform," 2010.

[21] A. Zambelli, "Iis smooth streaming technical overview," 2009.

[22] C. Liu, I. Bouazizi, M. M. Hannuksela, and M. Gabbouj, "Rate adaptation for dynamic adaptive streaming over http in content distribution network," *Signal Processing: Image Communication*, vol. 27, no. 4, pp. 288–311, 2012.

[23] I. Sodagar, "The mpeg-dash standard for multimedia streaming over the internet," *MultiMedia, IEEE*, vol. 18, no. 4, pp. 62–67, 2011.

[24] Q. Liu, R. Safavi-Naini, and N. P. Sheppard, "Digital rights management for content distribution," in *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003-Volume 21*. Australian Computer Society, Inc., 2003, pp. 49–58.

[25] C. Liu, I. Bouazizi, and M. Gabbouj, "Segment duration for rate adaptation of adaptive http streaming," in *Multimedia and Expo (ICME), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1–4.

[26] S. Akhshabi, L. Anantakrishnan, A. C. Begen, and C. Dovrolis, "What happens when http adaptive streaming players compete for bandwidth?" in *Proceedings of the 22Nd International Workshop on Network and Operating System Support for Digital Audio and Video*, ser. NOSSDAV '12. New York, NY, USA: ACM, 2012, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/2229087.2229092

[27] C. Liu, I. Bouazizi, and M. Gabbouj, "Rate adaptation for adaptive http streaming," in *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, ser. MMSys '11. New York, NY, USA: ACM, 2011, pp. 169–174. [Online]. Available: http://doi.acm.org/10.1145/1943552.1943575

[28] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran, "Probe and adapt: Rate adaptation for http video streaming at scale," *Selected Areas in Communications, IEEE Journal on*, vol. 32, no. 4, pp. 719–733, 2014.

[29] Z. Li, A. C. Begen, J. Gahm, Y. Shan, B. Osler, and D. Oran, "Streaming video over http with consistent quality," in *Proceedings of the 5th ACM Multimedia Systems Conference*. ACM, 2014, pp. 248–258.

[30] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 97–108.

[31] M. Zink, O. Künzel, J. Schmitt, and R. Steinmetz, "Subjective impression of variations in layer encoded videos," in *Quality of Service—IWQoS 2003*. Springer, 2003, pp. 137–154.

[32] C. Müller, S. Lederer, and C. Timmerer, "A proxy effect analyis and fair adatpation algorithm for multiple competing dynamic adaptive streaming over http clients." in *VCIP*, 2012, pp. 1–6.

[33] D. H. Lee, C. Dovrolis, and A. C. Begen, "Caching in http adaptive streaming: Friend or foe?" in *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*. ACM, 2014, p. 31.

[34] K. Miller, E. Quacchio, G. Gennari, and A. Wolisz, "Adaptation algorithm for adaptive streaming over http," in *Packet Video Workshop (PV), 2012 19th International*. IEEE, 2012, pp. 173–178.

[35] Y. Liu, J. Geurts, J.-C. Point, S. Lederer, B. Rainer, C. Muller, C. Timmerer, and H. Hellwagner, "Dynamic adaptive streaming over ccn: a caching and overhead analysis," in *Communications (ICC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 3629–3633.

[36] B. Rainer, "Mpeg-dash implemented with javascript," https://github.com/dazedsheep/DASH-JS, 2014.

[37] DASH-JS, "Js stops feeding chunks to media source api," https://github.com/dazedsheep/DASH-JS/issues/3, 2014.

[38] A. Colwell, A. Bateman, and M. Watson, "Media source extensions," http://www.w3.org/TR/media-source/, 2014.

[39] M. Carbone and L. Rizzo, "Dummynet revisited," *ACM SIGCOMM CCR*, Jan 2010.

A Appendix

A.1 Pipelining in DASH-JS

In DASH-JS pipelining is not used in refilling. Requests are not pipelined because DASH uses the last round of request-response to collect data and perform bandwidth estimation, so that the next request can be constructed accordingly based on the representation switching algorithm of the DASH engine. This can be confirmed both by the source code and the wireshark capture. However, if the byte-range of a segment is provided by the MPD, pipelining would be used to fetch the same segment through multiple HTTP partial GETs.

A.2 MPD file example

A sample MPD is shown in listing 3. This MPD does not match the latest version of the MPEG-DASH specification but can effectively illustrate how the XML based MPD file is constructed. Listing 3 does not show the MPD file in full but instead presents the vital structure of the MPD file.

Listing 3: Media Presentation Description

```
<?xml version="1.0" encoding="UTF-8"?>
<MPD xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="urn:mpeg:DASH:schema:MPD:2011"
     xsi:schemaLocation="urn:mpeg:DASH:schema:MPD:2011"
     profiles="urn:mpeg:dash:profile:isoff-main:2011"
     type="static"
     mediaPresentationDuration="PT0H9M56.46S"
     minBufferTime="PT4.0S">
     <BaseURL>http://www-itec.uni-klu.ac.at/ftp/datasets/mmsys12/BigBuckBunny/bunny_4s/</BaseURL>
     <Period start="PT0S">
          <AdaptationSet bitstreamSwitching="true">
<Representation id="0" codecs="avc1" mimeType="video/mp4" width="320" height="240" startWithSAP="1"
 bandwidth="45226">
<SegmentBase>
    <Initialization sourceURL="bunny_4s_50kbit/bunny_50kbit_dash.mp4"/>
</SegmentBase>
<SegmentList duration="4">
    <SegmentURL media="bunny_4s_50kbit/bunny_4s1.m4s"/>
    <SegmentURL media="bunny_4s_50kbit/bunny_4s2.m4s"/>
    ...
    <SegmentURL media="bunny_4s_50kbit/bunny_4s150.m4s"/>
  </SegmentList>
</Representation>
<Representation id="1" codecs="avc1" mimeType="video/mp4" width="320" height="240" startWithSAP="1"
 bandwidth="88783">
<SegmentBase>
    <Initialization sourceURL="bunny_4s_100kbit/bunny_100kbit_dash.mp4"/>
</SegmentBase>
<SegmentList duration="4">
    <SegmentURL media="bunny_4s_100kbit/bunny_4s1.m4s"/>
    <SegmentURL media="bunny_4s_100kbit/bunny_4s2.m4s"/>
    ...
    <SegmentURL media="bunny_4s_100kbit/bunny_4s150.m4s"/>
  </SegmentList>
</Representation>
...
<Representation id="10" codecs="avc1" mimeType="video/mp4" width="1280" height="720" startWithSAP="1"
 bandwidth="782553">
<SegmentBase>
    <Initialization sourceURL="bunny_4s_900kbit/bunny_900kbit_dash.mp4"/>
</SegmentBase>
<SegmentList duration="4">
    <SegmentURL media="bunny_4s_900kbit/bunny_4s1.m4s"/>
    <SegmentURL media="bunny_4s_900kbit/bunny_4s2.m4s"/>
    ...
    <SegmentURL media="bunny_4s_900kbit/bunny_4s150.m4s"/>
  </SegmentList>
</Representation>
          </AdaptationSet>
    </Period>
</MPD>
```

A.3 Apache configuration file

Configurations of the Apache server is done by modifying the directives in the configuration file - httpd.conf. We configure the apache server to listen to multiple ports and the directives are shown in listing 4

Listing 4: Apache Server Configuration

```
# open up multiple HTTP listening ports
Listen 80
Listen 3333
Listen 3131
Listen 4444
Listen 4141
Listen 5555
Listen 5151
Listen 7777
Listen 7171
Listen 8888
```

A.4 Squid configuration file

Squid is a widely used HTTP/1.1 proxy server that provides rich services including access control and HTTP object caching.The version we use in the thesis research is Squid 3.1.19. Squid provides a configuration file, squid.conf, for the user to manually configure its functionalities. Some of the configuration directives are explained in listing 5. A sample configuration used in this thesis research is shown in listing 6. More detail of the Squid directives can be found at [14]. These configuration directives can be found in squid.conf and the entries shown in listing 6 is printed by typing in the command "squid3 -k parse" under Linux shell. There are many other directives in squid.conf, which are default directives that will not be printed out by the "squid3 -k parse" command.

Listing 5: Directives for Squid.conf

```
# configure the cache_replacement_policy and decide the purge behavior
cache_replacement_policy heap LFUDA     # e.g. set to LFUDA

# Configure the disk cache;
# 2048MB disk space, first level 16 folders, second level 256 folders
cache_dir ufs /var/spool/squid3 2048 16 256

# Set the maximum size of a single HTTP object allowed to cache
maximum_object_size 1 GB

# Memory hot-content caching
cache_mem 128 MB        #128MB of memory allocated
maximum_object_size_in_memory 512 KB    # maximum object size stored in memory cache

# thresholds for cache swap;
# Old objects shall be purged from cache when high watermark reached
# Cache swap stops when low watermark reached
cache_swap_low 90       # threshold for deactiving cache swap
cache_swap_high 95      # threshold for activing cache swap

#Refresh_pattern defines how a certain type of file shall be cached and purged
#Any file that have an extension of .m4s can stay in the cache for at least 1440 minutes
#and at most 2016(=20%*10080) minutes
refresh_pattern -i \.m4s$ 1440 20% 4320
```

[14] http://www.squid-cache.org/Doc/config/

Listing 6: Sample squid.conf

```
2015/04/04 18:50:30| Processing Configuration File: /etc/squid3/squid.conf (depth 0)
2015/04/04 18:50:30| Processing: acl manager proto cache_object
2015/04/04 18:50:30| Processing: acl localhost src 127.0.0.1/32 ::1
2015/04/04 18:50:30| Processing: acl to_localhost dst 127.0.0.0/8 0.0.0.0/32 ::1
2015/04/04 18:50:30| Processing: acl lanVM src 192.168.227.0/24
2015/04/04 18:50:30| Processing: acl lanBridge src 192.168.137.0/24
2015/04/04 18:50:30| Processing: acl dashcontent urlpath_regex \.m4s$
2015/04/04 18:50:30| Processing: acl m4sreply rep_mime_type video/iso.segment
2015/04/04 18:50:30| Processing: acl SSL_ports port 443
2015/04/04 18:50:30| Processing: acl Safe_ports port 80          # http
2015/04/04 18:50:30| Processing: acl Safe_ports port 21          # ftp
2015/04/04 18:50:30| Processing: acl Safe_ports port 443                 # https
2015/04/04 18:50:30| Processing: acl Safe_ports port 70          # gopher
2015/04/04 18:50:30| Processing: acl Safe_ports port 210                 # wais
2015/04/04 18:50:30| Processing: acl Safe_ports port 1025-65535 # unregistered ports
2015/04/04 18:50:30| Processing: acl Safe_ports port 280                 # http-mgmt
2015/04/04 18:50:30| Processing: acl Safe_ports port 488                 # gss-http
2015/04/04 18:50:30| Processing: acl Safe_ports port 591                 # filemaker
2015/04/04 18:50:30| Processing: acl Safe_ports port 777                 # multiling http
2015/04/04 18:50:30| Processing: acl CONNECT method CONNECT
2015/04/04 18:50:30| Processing: http_access allow manager localhost
2015/04/04 18:50:30| Processing: http_access deny manager
2015/04/04 18:50:30| Processing: http_access deny !Safe_ports
2015/04/04 18:50:30| Processing: http_access deny CONNECT !SSL_ports
2015/04/04 18:50:30| Processing: http_access allow localhost
2015/04/04 18:50:30| Processing: http_access allow lanVM
2015/04/04 18:50:30| Processing: http_access allow lanBridge
2015/04/04 18:50:30| Processing: http_access allow dashcontent
2015/04/04 18:50:30| Processing: http_access deny all
2015/04/04 18:50:30| Processing: http_reply_access allow m4sreply
2015/04/04 18:50:30| Processing: http_reply_access allow all
2015/04/04 18:50:30| Processing: http_port 3128
2015/04/04 18:50:30| Processing: cache_mem 256 MB
2015/04/04 18:50:30| Processing: maximum_object_size_in_memory 1024 KB
2015/04/04 18:50:30| Processing: cache_replacement_policy heap LFUDA
2015/04/04 18:50:30| Processing: cache_dir ufs /var/spool/squid3 400 16 256
2015/04/04 18:50:30| Processing: minimum_object_size 0 KB
2015/04/04 18:50:30| Processing: maximum_object_size 1 GB
2015/04/04 18:50:30| Processing: cache_swap_low 90
2015/04/04 18:50:30| Processing: cache_swap_high 95
2015/04/04 18:50:30| Processing: access_log /var/log/squid3/access.log squid
2015/04/04 18:50:30| Processing: coredump_dir /var/spool/squid3
2015/04/04 18:50:30| Processing: cache allow dashcontent
2015/04/04 18:50:30| Processing: cache allow m4sreply
2015/04/04 18:50:30| Processing: cache allow all
2015/04/04 18:50:30| Processing: refresh_pattern ^ftp:           1440    20%     10080
2015/04/04 18:50:30| Processing: refresh_pattern ^gopher:        1440    0%      1440
2015/04/04 18:50:30| Processing: refresh_pattern -i (/cgi-bin/|\?) 0    0%      0
2015/04/04 18:50:30| Processing: refresh_pattern (Release|Packages(.gz)*)$      0       20%     2880
2015/04/04 18:50:30| Processing: refresh_pattern .              0       20%     4320
2015/04/04 18:50:30| Processing: refresh_pattern -i \.m4s$ 1440 20% 4320
2015/04/04 18:50:30| Processing: refresh_pattern -i iso\.segment 1440 20% 4320
2015/04/04 18:50:30| Processing: quick_abort_min -1 KB
2015/04/04 18:50:30| Processing: visible_hostname UbuntuProxyL1_158
```

A.5 DASH-JS source code

Our implementation of the DASH client is based on the original DASH-JS [36], and can be found in our online repository on GitHub. The repository can be replicated by typing in the following command under unix shell: **git clone** https://github.com/yunfengHe/DASH-JS.git. Detailed explanations of the source code lines are written in the source code files, in the form of code commentary. To adjust the Overlay Buffer (the receive buffer) or to swap different Gearbox versions, check the source file "dash.js" and do modifications. To modify the Gearbox algorithm, check the source file "adaptationlogic.js". Other functionalities such as network monitoring and data collection can also be found and modified in the corresponding source files.