NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

INTERDEPARTMENTAL MASTERS STUDIES PROGRAMME
"DATA SCIENCE AND INFORMATION TECHNOLOGIES"

MASTERS THESIS

Peer-to-Peer video content delivery optimization service in a distributed network

Nikolaos D. Episkopos

Supervisor: **Stathes Hadjiefthymiades,** Professor

ATHENS

SEPTEMBER 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕ

ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
"ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΕΣ ΠΛΗΡΟΦΟΡΙΑΣ"

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υπηρεσία βελτιστοποίηση της διανομής περιεχομένου βίντεο μεταξύ ομότιμων σε κατανεμημένο δίκτυο

Νικόλαος Δ. Επίσκοπος

Επιβλέπων: **Ευστάθιος Χατζηευθυμιάδης,** Καθηγητής

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2022

MASTERS THESIS

Peer-to-Peer video content delivery optimization service in a distributed network

Nikolaos D. Episkopos
ID: DS1.19.0005

SUPERVISOR: **Stathes Hadjiefthymiades,** Professor

COMMITTEE:
Stathes Hadjiefthymiades, Professor
Dionysis Xenakis, Assistant Professor
Nikos Passas, Laboratory Teaching Staff

September 2022

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υπηρεσία βελτιστοποίηση της διανομής περιεχομένου βίντεο μεταξύ ομότιμων σε κατανεμημένο δίκτυο

Νικόλαος Δ. Επίσκοπος
Α.Μ.: DS1.19.0005

ΕΠΙΒΛΕΠΩΝ: **Ευστάθιος Χατζηευθυμιάδης,** Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Ευστάθιος Χατζηευθυμιάδης,** Καθηγητής
Διονύσης Ξενάκης, Επίκουρος Καθηγητής
Νίκος Πασσάς, ΕΔΙΠ

Σεπτέμβριος 2022

ABSTRACT

Dynamtic Adaptive Streaming over HTTP (DASH) has yield several improvements in the video playback Quality of Experimence (QoE) for the end users in pre-fifth generation (5G) networks. However, cloud applications that 5G networks enable, combined with cloud infrastructures at the edge of the network and in close vicinity to the end users, can offer significant improvements in both the offered Quality of Service (QoS) and QoE because of the video content caching capabilities at the edge of the network that the edge cloud can offer. Furthermore, in addition to edge caching and edge video streaming to the end users, new video infrastructures can offer Device-to-Device (D2D) video content exchange and delivery. Taking advantage of these technologies, innovative video streaming services can be developed which not only improve the video playback QoE for the end users but also reduce the video delivery costs and generated network traffic, which also means reduced end-to-end latency and reduced overhead in video content providers' Content Delivery Network (CDN). In this thesis we study the impact of using different combinations of distinct video caching techniques, video segment request and streaming algorithms and video resolution selection logics on the QoS and the QoE of end users at the network edge, which can be used in developing an innovative Peer-to-Peer (P2P) video content delivery optimization service in a distributed network.

SUBJECT AREA: 5G & MEC Networks
KEYWORDS: 5G, MEC, DASH, D2D, Caching

ΠΕΡΙΛΗΨΗ

Η δυναμικά προσαρμοζόμενη ροή βίντεο μέσω HTTP (DASH) παρέχει βελτιώσεις στην ποιότητα της εμπειρίας χρήσης (QoE) κατά την αναπαραγωγή βίντεο σε δίκτυα παλαιότερα των δικτύων 5ης γενιάς (5G). Ωστόσο, οι εφαρμογές τύπου νέφους τις οποίες μπορεί να παρέχει η αρχιτεκτονική δικτύων 5ης γενιάς, σε συνδυασμό με την υλοποίηση υπολογιστικών υποδομών νέφους στο άκρο του δικτύου και κοντά στους τελικούς χρήστες, μπορεί να βελτιώσει σημαντικά τόσο την ποιότητα της προσφερόμενης υπηρεσίας (QoS) όσο και την εμπειρία χρήσης λόγω της δυνατότητας προσωρινής αποθήκευσης περιεχομένου βίντεο στο άκρο του δικτύου, λόγω της δυνατότητας παροχής προσωρινής αποθήκευσης μέρους του βίντεο στο άκρο του δικτύου. Επιπροσθέτως, εκτός της αποθήκευσης στο και διανομής βίντεο από το άκρο του δικτύου προς τους τελικούς χρήστες, οι νέες υποδομές βίντεο θα παρέχουν τη δυνατότητα διανομής περιεχομένου βίντεο απευθείας από συσκευή σε συσκευή (D2D). Αξιοποιώντας τις τεχνολογίες αυτές, μπορούν να υλοποιηθούν καινοτόμες υπηρεσίες ροής βίντεο, οι οποίες μπορούν όχι μόνο να βελτιώσουν την εμπειρία χρήσης των τελικών χρηστών κατά την αναπαραγωγή βίντεο, αλλά και να μειώσουν το συνολικό κόστος διανομής βίντεο καθώς και την συμφόρηση των δικτύων, άρα και την καθυστέρηση από άκρο σε άκρο και τη συμφόρηση στα δίκτυα διανομής περιεχομένου (CDN) των παρόχων υπηρεσιών διανομής και ροής βίντεο. Στην παρούσα διπλωματική εργασία μελετούμε την επίπτωση που έχουν διάφοροι συνδυασμοί τεχνικών προσωρινής αποθήκευσης, διανομής, καθώς και επιλογής ανάλυσης, σε περιεχόμενο βίντεο, πάνω στην ποιότητα της προσφερόμενης υπηρεσίας και στην εμπειρία των τελικών χρηστών που βρίσκονται στο άκρο του δικτύου, οι οποίες μπορούν να αξιοποιηθούν στη δημιουργία μιας καινοτόμας υπηρεσίας που βελτιστοποιεί τη διανομή περιεχομένου βίντεο μεταξύ ομότιμων κόμβων (P2P) σε ένα κατανεμημένο δίκτυο.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Δίκτυα 5G & MEC
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: 5G, MEC, DASH, D2D, Caching

Στην Κατερίνα, που έχει σταθεί δίπλα μου όλα αυτά τα δύσκολα για εμένα χρόνια, με ανεξάντλητη υπομονή και αγάπη.

AKNOWLEDGEMENTS

I would like to express my sincere gratitude to the Assistant Professor Dionysis Xenakis for our collaboration, for the academic guidance he has been providing me with and for his supervision.

I would also like to thank my family for their love, care and their support in my decisions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This Master's thesis is part of the research conducted at Fogus Innovations and Services P.C., between the years 2021 and 2022, as part of my contribution to the EU-funded Greek national RE-CENT project (GA ID: Τ1ΕΔΚ-03524). The research presented in this thesis is part of a paper submitted to the IEEE/ACM Transactions on Networking journal. I would like to extend my gratitude to Assistant Professor Dionysis Xenakis, next to whom I was taught the research methodology and with whom I have been working on the state-of-the-art in Computer Science and Networking, for giving me the permission to use part of our research as my Master's Thesis. His contribution and supervision were the most decisive factors in every step of our joint research. I would also like to extend my gratitude to Professor Stathes Hadjiefthymiades for agreeing to participate in this thesis as the supervisor.

1. INTRODUCTION

1.1 5G applications

Fifth generation wireless systems, or 5G as they have been marketed, are being deployed all around the world in a fast pace and with them significant advancements are being introduced into modern networks, both throughout the network infrastructure and through the introduction of new and innovative cloud-based technologies, as thoroughly listed and analyzed in [1].

As far as the network infrastructure changes are concerned, the most significant advancements 5G brings to modern networks include:

- Higher frequencies in the electromagnetic radiation spectrum
- Larger bandwidth
- Higher data rates
- Lower latencies
- Lower power consumption
- Higher capacity
- Better coverage and availability

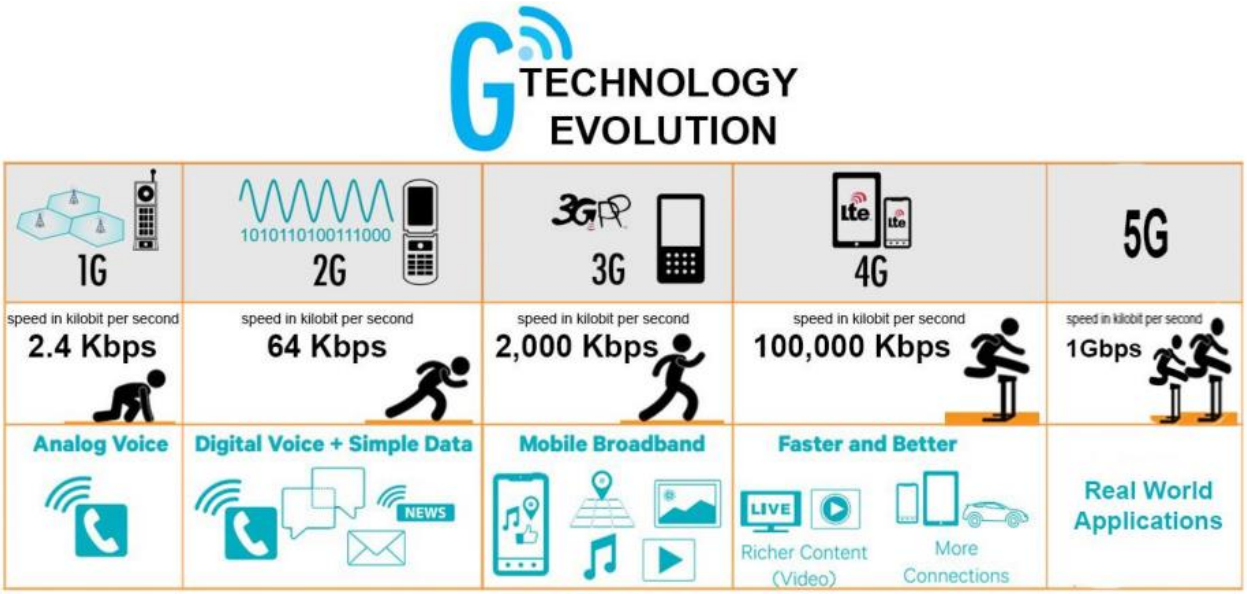

Figure 1: Evolution of Wireless Communication Technologies [2]

However, these qualitative advancements are not the only the only important differences between past and future networks. 5G also provides the infrastructure for innovative applications and services through cloud-type technologies, which include:

- Software-defined networking
- Network functions virtualization
- Radio access network as a service
- Traffic offload as a service
- Anything as a service
- Device-to-Device (D2D) communication
- Mobile Content Delivery Network as a service
- Multi-access Edge Computing (MEC)
- Distributed content delivery and caching
- Application awareness and content optimization

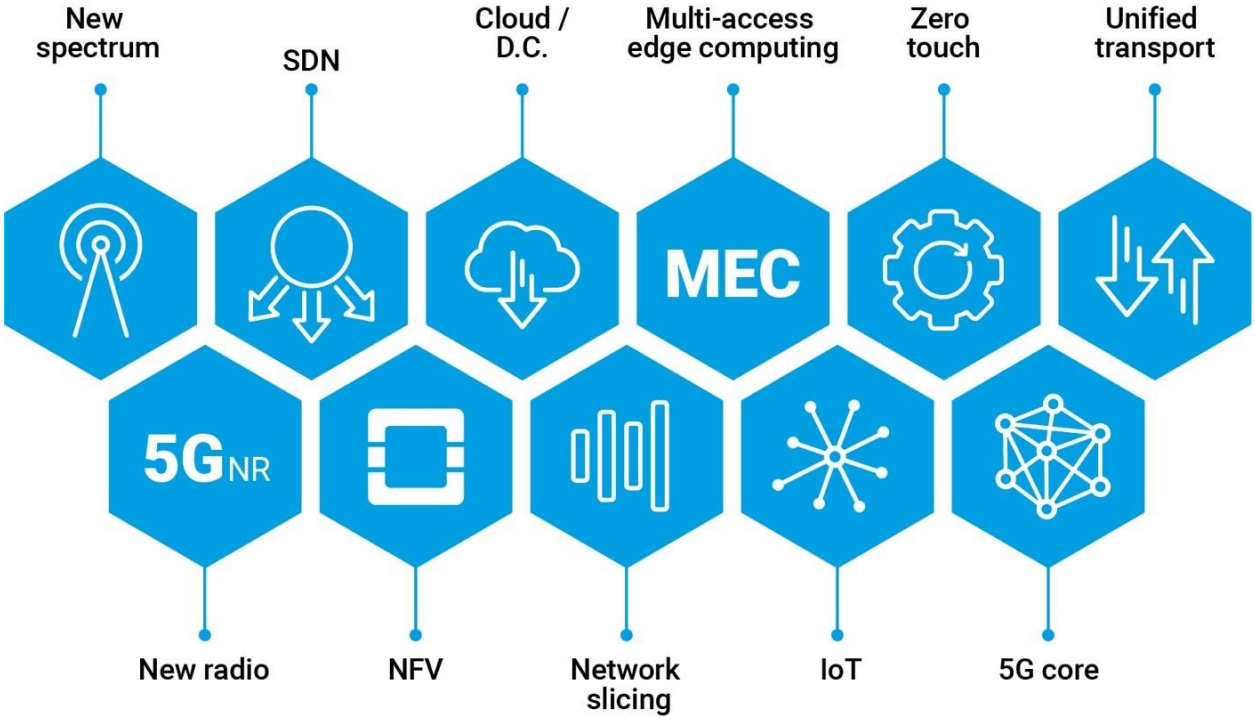

Figure 2: 5G cloud technologies [3]

This thesis focuses on the last five bullets from the list above with the cloud services, and more specifically on a combination of these five technologies, which can utilize MEC while also taking advantage of the storage and caching capabilities of mobile devices, as well as their proximity in a Local Area Network (LAN), to create an application-aware service for a mobile Content Delivery Network (CDN) with optimized D2D video content delivery.

Videos and video playback are among the most widespread applications of current networks, as well as among the most traffic-intense ones. Millions of people hours daily consuming video content from various video streaming services like YouTube, Netflix, Disney+, etc. either on their Smart TVs or on their smartphones. It is considered that in the last few years video streaming comprises more than half of the total internet traffic. Year-after-year, an increasing number of new companies which base their business on video content provision services are founded and compete with each other for a slice of the market pie, which means consumers have a variety of video content services to choose from. The latest social media like Instagram and TikTok base their whole business on the distribution and consumption of video content.

However, due to the fact that these services are provided by CDNs in servers which in most cases are locally positioned in specific sites that are chosen by their owner companies, many end users and video content consumers can be positioned in areas hundreds of kilometers away from said sites. Moreover, even more recent cable and wireless networks suffer from bandwidth limits as well as signal interference due to the increasing number of devices with wireless networking capabilities. Because of all these facts, it is not uncommon for latency problems to arise which make video loading take longer and video playback to stall, resulting in a worse Quality of Service (QoS) of the offered service, which in turn can mean a worse Quality of Experience (QoE) as well as a frustration for the end users.

To alleviate a portion of the consequences of this problem in modern networks, video content caching has been proposed as a solution, which works in a similar way like web page caching for web content does. Multiple video caching approaches have been

proposed, the most common being caching video content at the edge of the network [4], since MEC is going to be an essential part of future networks.

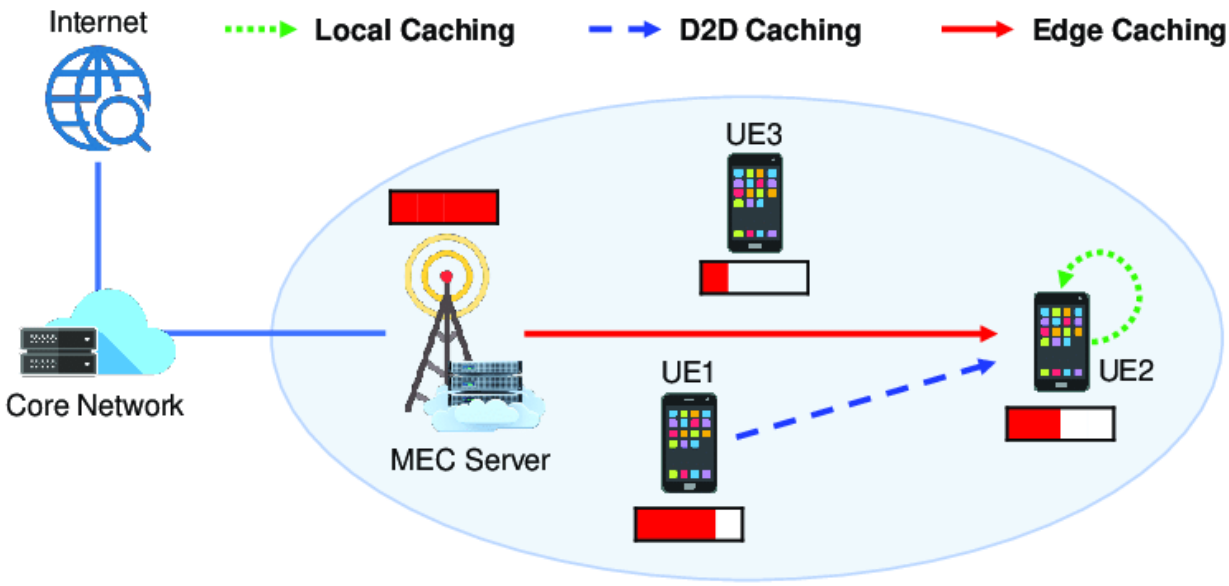

Figure 3: Device-enhanced MEC content caching via D2D communication [5]

1.2 Fundamentals

1.2.1 QoS and QoE

According to the International Telecommunications Union standards body (ITU), QoS is the "totality of characteristics of a telecommunications service that bear on its ability to satisfy stated and implied needs of the user of the service." Under this definition, QoS measures the performance of the service delivery infrastructure, including third-party or internal CDN, usually through tracking data like overall throughput, latency, error rates, and cache hit ratio. In contrast, the ITU defines QoE as "the overall acceptability of an application or service, as perceived subjectively by the end-user. It includes the complete end-to-end system effects (client, terminal, network, service infrastructure, etc.) and may be also influenced by user expectations and context". So, QoE directly measures the end user experience, including factors like playback success, playback startup time, rebuffering events and their time, as well as visual quality, both for encoding and decoding.

Fig. 4 shows how QoS and QoE interrelate. As we see, on the left is content preparation through encoding and packaging. Then the content files are handed off to the delivery infrastructure for distribution, which is what QoS measures. Once received at the viewing location, the video is decoded and played back on a video player application. As the figure shows, QoE involves the complete end-to-end experience, while QoS is the infrastructure portion until (and including) the content distribution.

Clearly, network performance is critical for achieving a good QoE. But so are other factors like source quality, encoding quality, packaging integrity, as well as the video viewing environment. It's one thing to create content that looks great on a smartphone; quite another to produce and deliver content that rocks an 85" LED in a living room. For example, we can have great QoS but poor QoE if the quality of the source material was sketchy or if the player doesn't use Adaptive Bit Rate (ABR) [7] to switch to the highest quality stream because of some missing implementation or some faulty implementation or faulty logic. We can also have a high-quality QoE and poor-quality QoS. For example,

the network capacity can be high enough to deliver the video content on time, but if the caching performance is poor both the bandwidth and distribution costs and overheads increase. This means that to get clear view of a system's performance, we have to measure both the QoS and the QoE.

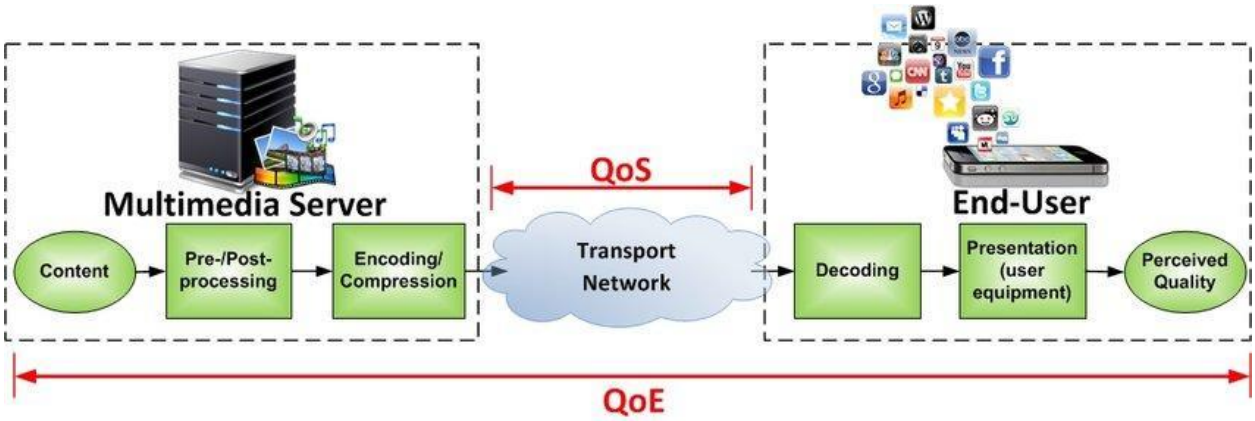

Figure 4: QoS and QoE interrelation [6]

To measure the QoS some systems install probes on the network between the systems that are to be monitored. These probes are hardware devices or software programs installed in service centers around the Internet that monitor traffic from a defined source or sources to a destination or destinations. So, if we wanted to monitor performance between the content prep headend and the core network, we would install a probe at the core network. If we wanted to monitor the streaming service between the core network and the edge, we would install a probe at the edge. These probes allow us to identify the location of the delivery issues and, therefore, the source. Other systems track QoS by installing tracking software in the video player itself, which is the most common deployment schema for QoE. The QoS metrics will vary by product and service provider, but metrics like throughput, bitrate, latency, jitter, and packet loss are commonly tracked and compared by various vendors.

QoE is usually measured by plug-ins in the video player that report performance data to the central database for analysis and presentation. The method of data extraction and data tracked varies by service provider. Two organizations have weighed in on the key data points to track. According to the Streaming Video Alliance, the key metrics are video start time, re-buffering ratio, average media bitrate, and video start failures, as presented in Table 1. The Consumer Technology Association recommends monitoring playback failures, startup time, playback stalling, bitrate, player failures, and other metrics, including advertising insertion and many other players and video playback data points like player width and video resolution.

There are several factors that may be responsible for the overall experience that a user perceives. However, the main influencing factors related to QoE are categorized into three domains: human, network, and context [8].

- Human factors: individual characteristics such as age, gender, memory, attention, satisfaction, education standards, mood, social and psychological factors as well as expectations are within the scope of the human domain.
- Network/ System: The state of the network plays a very important role in QoE. There are many metrics about the network which are divided into four layers. Each layer has the so-called Key Performance Indicators (KPIs). KPIs are values that

measure what is intended to be measured in order to offer a comparison that gauges the degree of network performance change over time. These are:

- Video Specific: we care here about the frame rate, the video content, the resolution, and the format of the video, as well as the type of the terminal device.
- Video on Demand: Here are some examples like YouTube where we care about the number and the duration of stalling events, the total duration of the video and how long it takes for a video to start playing after we press the play button.
- Transport / Network: Here are some network metrics such as round trip, jitter, packet loss ratio and congestion period.
- Physical: Here are some metrics that refer to SNR / SIR / SINR, bit/symbol rate, packet / symbol / bit error probability and energy efficiency.

- Context: In this domain we care about the situation in which the user is. Some examples are the urgency of a call, financial policy (such as if there is a high charge for a specific service or not), energy consumption issues, environmental conditions (such as the type of the weather) and customer support.

Table 1: Key QoE metrics according to the Streaming Video Alliance

Metric	Description
Video Start Time (seconds)	The amount of time from the initiation of the play event until the first frame of network delivered video is rendered. *[Note that the display of pre-loaded "splash" video streams is not relevant to this metric.]*
Re-buffering Ratio (%)	The percentage of time that a viewer experiences re-buffering issues (i.e. when video stops playing because of buffer underflow, and not due directly to user intervention such as scrubbing or pause). The ratio is calculated as total re-buffering time divided by the sum of total playing time plus total re-buffering time. *[On certain devices, eliminating scrubbing from the measured re-buffering time may not be possible. To make it comparable across all devices, a separate metric may be calculated that includes the scrubbing and pause time. Per device measurement can also be made to keep these metrics comparable.]*
Average Media Bit Rate (bits per second)	When the first chunk of video is not fully delivered within time cut-off (~10s) from the initiation of the play event.
Video Start Failure (yes or no)	When the first chunk of video is not fully delivered within time cut-off (~10s) from the initiation of the play event.

1.2.2 DASH

The Motion Picture Expert Group (MPEG) and ISO groups ratified the Dynamic Adaptive Streaming over HTTP (DASH) [9] standard, also known as MPEG-DASH, as a response to many competing yet similar (and proprietary / vendor dependent) video streaming algorithms, which belong to the family of HTTP Adaptive Streaming (HAS) [10] algorithms.

HAS algorithms were developed and used by video content provision services and platforms to stream video content to devices and applications. Examples of such algorithms are Apple HLS, Microsoft Smooth Streaming, Adobe HDS, etc. [11]. The concept of HAS is presented in Fig. 5. Despite the many HAS advantages, several inefficiencies still have to be solved to improve the user's QoE, especially in live video streaming.

DASH is an adaptive bitrate streaming technique that enables high quality streaming of media content over the Internet delivered from conventional HTTP web servers, using the TCP protocol. DASH works by encoding the video content at different bitrates / quality levels, then breaking every quality level into a sequence of small segments, covering aligned short intervals of playback time which have a typical duration of one to ten seconds, which are served over HTTP requests. Each quality level is determined by its corresponding average video bitrate. Each segment contains a short interval of playback time of content that is potentially many hours in duration, such as a movie or a live broadcast of an event, and can be decoded independently of other segments. DASH is codec-agnostic, which means it can be used with content encoded with any coding format, such as H.265, VP9, MP3, etc. HTTP-based video streams can easily traverse firewalls and reuse the already deployed HTTP infrastructure such as HTTP servers, HTTP proxies, and CDN nodes.

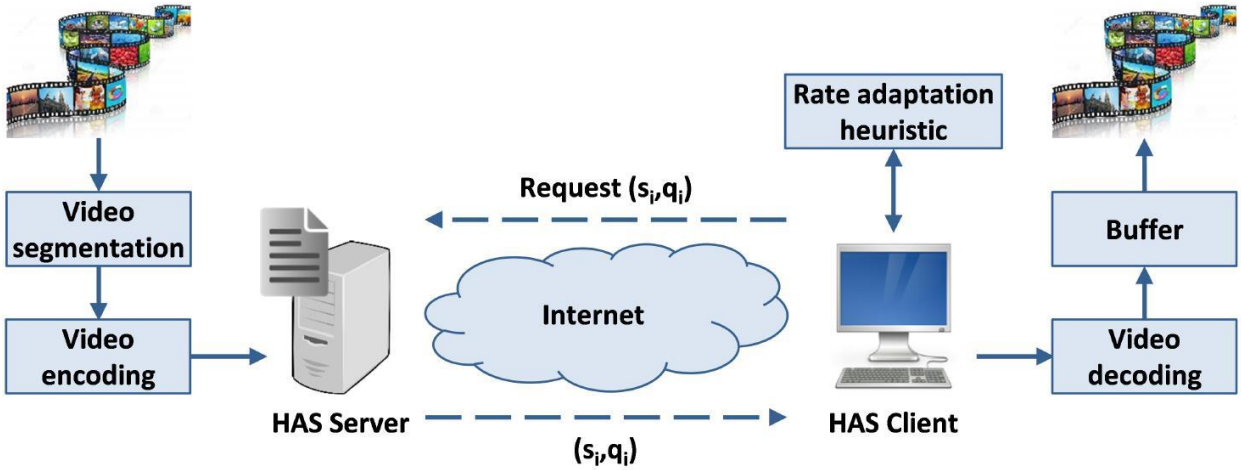

Figure 5: The concept of HTTP Adaptive Streaming (HAS) [12]

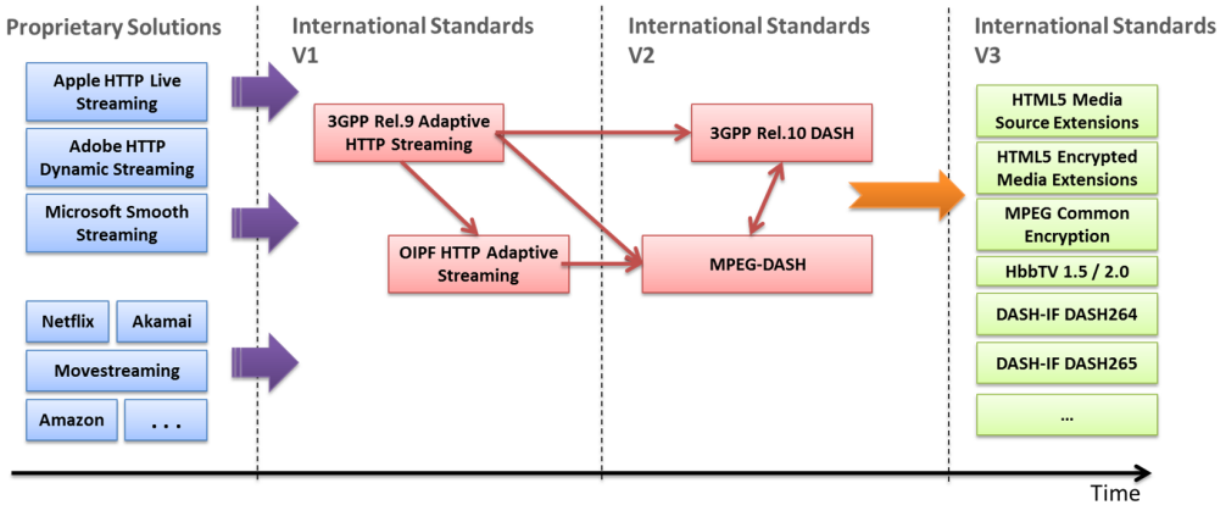

Figure 6: Evolution of HAS in time [13]

A DASH client, which plays back the video content, initiates a new session by first requesting and downloading a manifest file, which is a Media Presentation Description (MPD) file and provides a short description of the different available quality levels and segments. As a consequence, each client will first request the MPD that contains the temporal and structural information for the media content, and based on that information it will request the individual segments that fit best for its requirements. The client uses an ABR algorithm to automatically select the segment with the highest bit rate possible that can be downloaded in time without causing stalls or re-buffering events during playback. The DASH client uses the Rate Determination Algorithm (RDA) to determine the quality for the next segment to download. The objective of the RDA is to optimize the global Quality of Experience (QoE) determined by the occurrence of video freezes, the average quality level, and the frequency of quality changes. Thus, a DASH client can seamlessly adapt the video quality to changing network conditions and provide high quality playback while minimizing any freezes/stalls or re-buffering events. As a consequence, HAS facilitates video streaming over a best-effort network. The concept of DASH is presented in Fig. 7 and Fig. 8.

In summation, the Key-Targets and Benefits of MPEG-DASH are:

- reduction of startup delays and buffering/stalls during the video
- continued adaptation to the bandwidth situation of the client
- client-based streaming logic enabling the highest scalability and flexibility
- use of existing and cost-effective HTTP-based CDNs, proxies, caches
- efficient bypassing of NATs and Firewalls by the usage of HTTP
- common Encryption – signaling, delivery & utilization of multiple concurrent digital rights management (DRM) schemes from the same file
- simple splicing and (targeted) ad insertion
- support for efficient trick mode

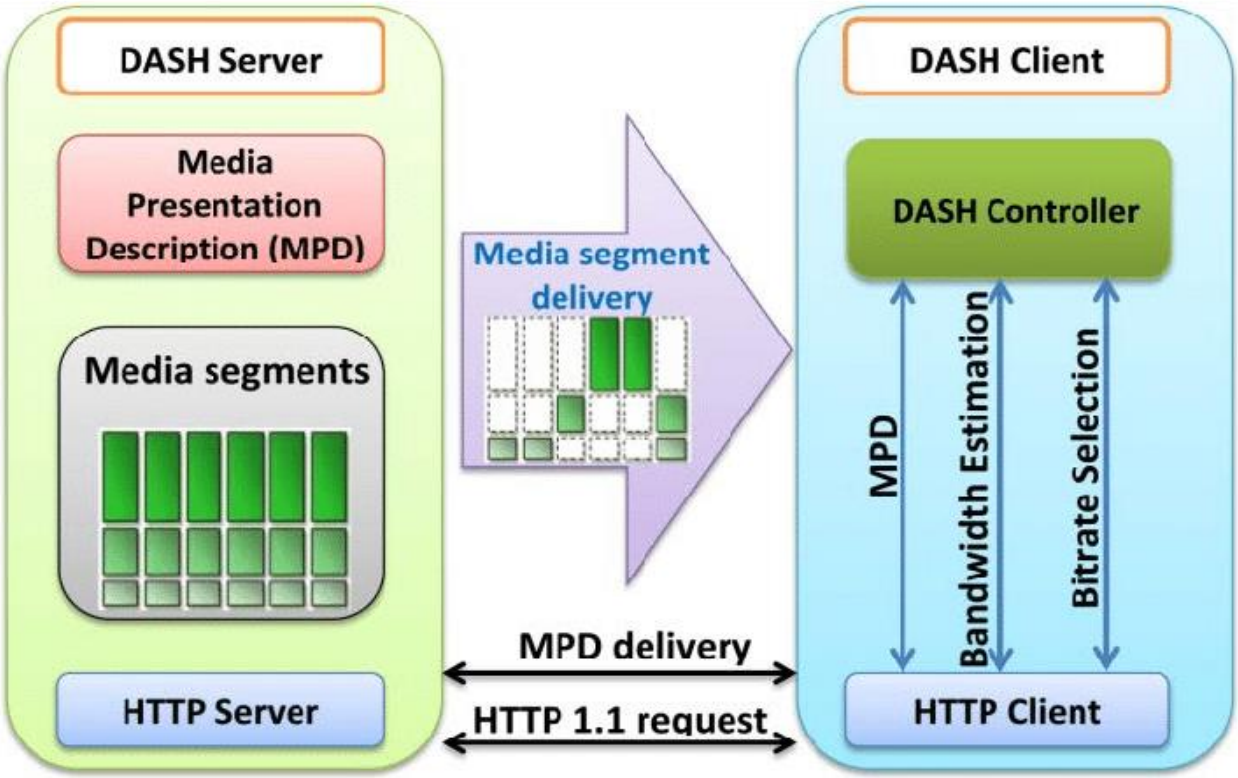

Figure 7: The concept of DASH [14]

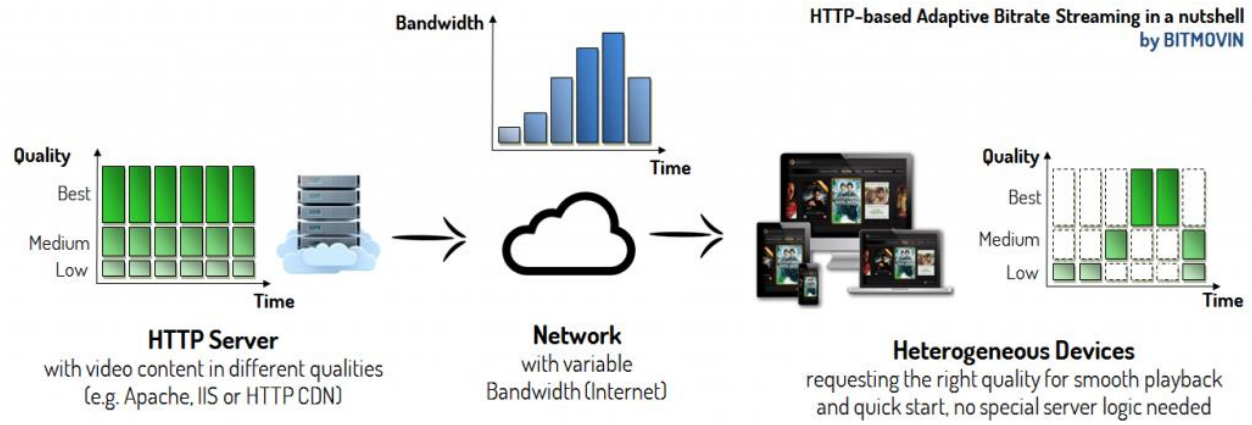

Figure 8: The concept of DASH [15]

In recent years, MPEG-DASH has been integrated into new standardization efforts, e.g., the HTML5 Media Source Extensions (MSE) enabling the DASH playback via the HTML5 video and audio tag, as well as the HTML5 Encrypted Media Extensions (EME) enabling DRM-protected playback in web browsers. Furthermore, DRM-protection with MPEG-DASH is harmonized across different systems with the MPEG-CENC (Common Encryption) and MPEG-DASH playback on different SmartTV platforms is enabled via the integration in Hybrid broadcast broadband TV (HbbTV 1.5 and HbbTV 2.0). The usage of the MPEG-DASH standard has also been simplified by industry efforts around the DASH Industry Forum and their DASH-AVC/264 recommendations, as well as forward-looking approaches such as the DASH-HEVC/265 recommendation on the usage of H.265/HEVC within MPEG-DASH. Fig. 6 shows the evolution of DASH standards in time.

Assuming co-location of Base Stations (BSs) and MEC servers, server-assisted and network-assisted DASH (SAND) has been shown to offer measurable performance gains [22].

In order to describe the temporal and structural relationships between segments, MPEG-DASH introduced the so-called MPD. The MPD is an XML file that represents the different qualities of the media content and the individual segments of each quality with HTTP Uniform Resource Locators (URLs). This structure provides the binding of the segments to the bitrate (resolution, etc.) among others (e.g., start time, duration of segments). The MPEG-DASH MPD is a hierarchical data model. Each MPD could contain one or more Periods. Each of those Periods contains media components such as video components e.g., different view angles or with different codecs, audio components for different languages or with different types of information (e.g., with director's comments, etc.), subtitle or caption components, etc. Those components have certain characteristics like the bitrate, frame rate, audio channels, etc. which do not change during one Period. Fig. 9 provides a better understanding of the structure and contents of an MPD file.

The most important details [16] of MPD's structure are:

- **Periods**, contained in the top-level MPD element, describe a part of the content with a start time and duration. Multiple Periods can be used for scenes or chapters, or to separate ads from program content.
- **Adaptation sets**, which contain a media stream or a set of media streams
- **Representations** allow an Adaptation Set to contain the same content encoded in different ways. In most cases, Representations will be provided in multiple screen Cache-Aware Adaptive Video Streaming in 5G Networks sizes and bandwidths in order to allow clients to request the highest quality content that they can play without waiting to buffer or wasting bandwidth.

- **Sub representations** contain information that only applies to one media stream in a Representation. They also provide information necessary to extract one stream from a multiplexed container, or to extract a lower quality version of a stream.
- **Media segments** are the actual media files that the DASH client plays, generally by playing them back-to-back as if they were the same file. Media Segment locations can be described using BaseURL for a single-segment Representation, a list of segments (SegmentList) or a template (SegmentTemplate).
- **Index Segments** come in two types: one Representation Index Segment for the entire Representation which is always a separate file, or a Single Index Segment per Media Segment which can be a byte range in the same file as the Media Segment.

Nevertheless, the client is able to adapt during a Period according to the available bitrates, resolutions, codecs, etc. that are available in a given Period. Furthermore, a Period could separate the content, e.g., for ad insertion, changing the camera angle in a live football game, etc. For example, if an ad should only be available in high resolution while the content is available from standard definition to high definition, you would simply introduce your own Period for the ad which contains only the ad content in high definition. After and before this Period, there are other Periods that contain the actual content (e.g., movie) in multiple bitrates and resolutions from standard to high definition.

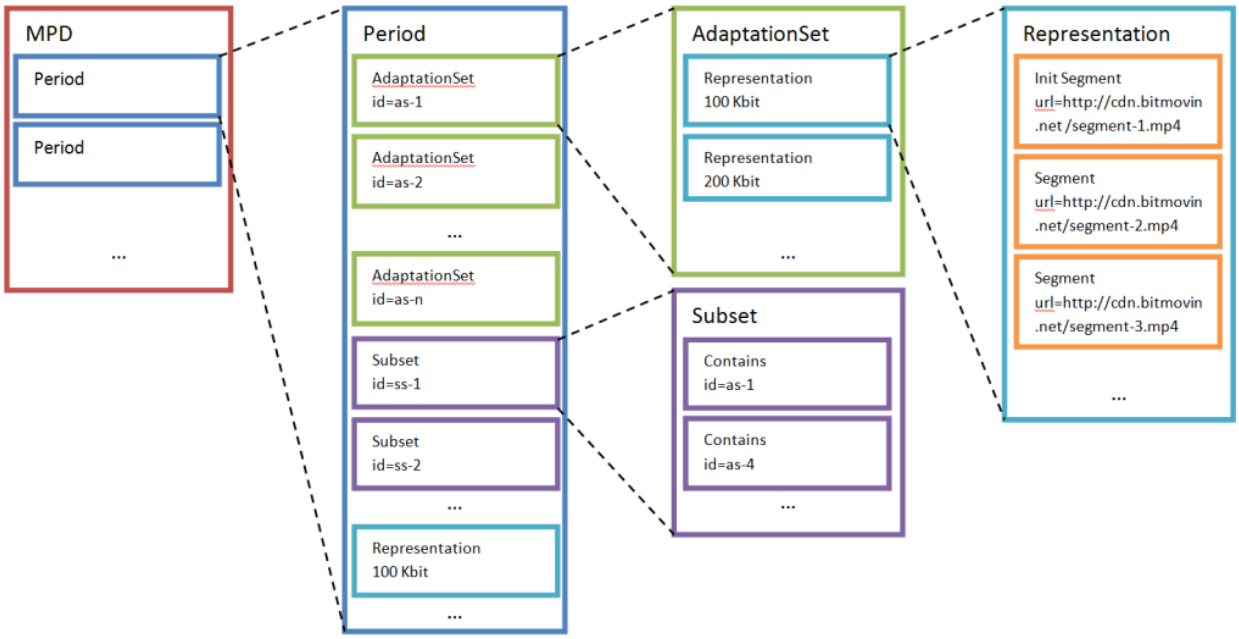

Figure 9: The hierarchical structure of an MPD file

By parsing the MPD, the DASH client knows all the information about the program timing, media-content availability, media types, resolutions, minimum and maximum bandwidths, and the existence of various encoded alternatives of multimedia components, accessibility features and required DRM, media-component locations on the network, and other content characteristics. The client capitalizes the information and selects the appropriate encoded alternative in order to start streaming the content by fetching the segments using HTTP GET requests. After appropriate buffering to allow for network throughput variations, the client continues fetching the subsequent segments but keeps on monitoring the network bandwidth fluctuations. Depending on its measurements, the client decides how to adapt to the available bandwidth by fetching segments of different bitrates to avoid buffering. The MPEG-DASH specification only defines the MPD and the segment formats. The delivery of the MPD and the media encoding formats containing

the segments, as well as the client behavior for fetching, adaptation heuristics, and playing content, are outside of MPEG-DASH's scope [9].

1.2.3 Proxy caching for Video Streaming

A Video on Demand system (VoD) system, which provides service for users, typically consists of two main components: the central video server and a set of video clients. The central video server has a (infinite) storage space to store all the available videos for clients connected via a wide area network (WAN) or a LAN. In such a framework, all the requests from clients are handled at the central server. The content exchange process starts with generating a request message from a client to the central server. In response to the client's request, the central server serves each request individually through a dedicated channel. Although this operation is simple to implement, the whole architecture is excessively expensive and without scalability due to the fact that the bandwidth bottleneck of the central server limits the number of clients it can serve in parallel, which in turn can lead to a significant drop of the end user's QoE. Furthermore, the introduction of long service latencies is another critical factor affecting the system performance, which is especially crucial when the video is transmitted over the WAN [17].

To leverage the workload of the central server and reduce the service latencies, an intermediate device called cache proxy is placed between the central server and clients. In the proxy-based architecture, a portion of video is cached in the proxy [18]. Upon receiving the content request, the proxy checks to see if it has a copy of the requested object in its cache. If so, the proxy responds by sending the cached object to the client (cache hit). Otherwise, it sends a content request of its own for said object the request to the server (cache miss). If the proxy requests the object from the central server, the received object data are cached by the proxy so that it can fulfill future content requests for the same object without retrieving it again from the central server. The result of serving a cached object is a zero server-side bandwidth usage with a huge improvement in the response and serving time for the end user. Meanwhile, the central server also delivers the non-cached portion of the video to the client indirectly through the cache proxy [19].

Existing caching mechanisms about video streaming can be mainly classified into four categories [19]:

- Sliding-interval caching which caches the playback interval between two requests.
- Prefix caching which divides the video into two parts named prefix and suffix. Prefix is the leading portion of the video, which is cached in the proxy, while the suffix is the rest of the video which is stored in the central server. Upon receiving a client's request, the proxy delivers the prefix to the client, meanwhile, it also downloads the suffix from the central server and then relays to the client.
- Segment caching generalizes the prefix caching by partitioning a video object into a number of segments. The proxy caches one or several segments based on the caching decision algorithm.
- Rate-split is a type of caching where the central server stores the video frame with the data rate. If the data rate of the video frame is higher than a threshold value which is called cutoff rate, it is partitioned into two parts where the cutoff is the boundary such that the transmission rate of the central server can keep constant.

Assuming that MEC servers are co-located with cellular BSs and that each user requests for a fixed video bitrate, collaborative video caching and transcoding is proposed to minimize the initial playback delay of end users [23].

1.2.4 CDN

Content distribution networks is an extension of the proxy caching and it is a critical component of any modern video application. In such architecture, the delivery of content will be improved by replicating commonly requested video content files across a globally distributed set of caching servers which are deployed at the edge of the network core. Unlike proxy, which only stores a portion of the video, a full copy of the video is replicated in each CDN server. Then, clients request the video from their closest CDN servers directly. This architecture significantly reduces the workload of the central server and provides a better QoS to clients.

CDNs do a lot more than just caching, such as delivering dynamic content that is unique to the requestor and not cacheable. The advantage of having a CDN deliver dynamic content is application performance and scaling. The CDN will establish and maintain secure connections closer to the requestor and, if the CDN is on the same network as the origin, as is the case for cloud-based CDNs, routing back to the origin to retrieve dynamic content is accelerated. Caches have also become much more intelligent, providing the ability to inspect information contained in the request header and vary the response based on device type, requestor information, query string, or cookie settings. CDNs can be directed to retrieve objects from multiple origins, enforce protocol policy, negotiate SSL connections, and restrict object access by location or authentication credentials [20].

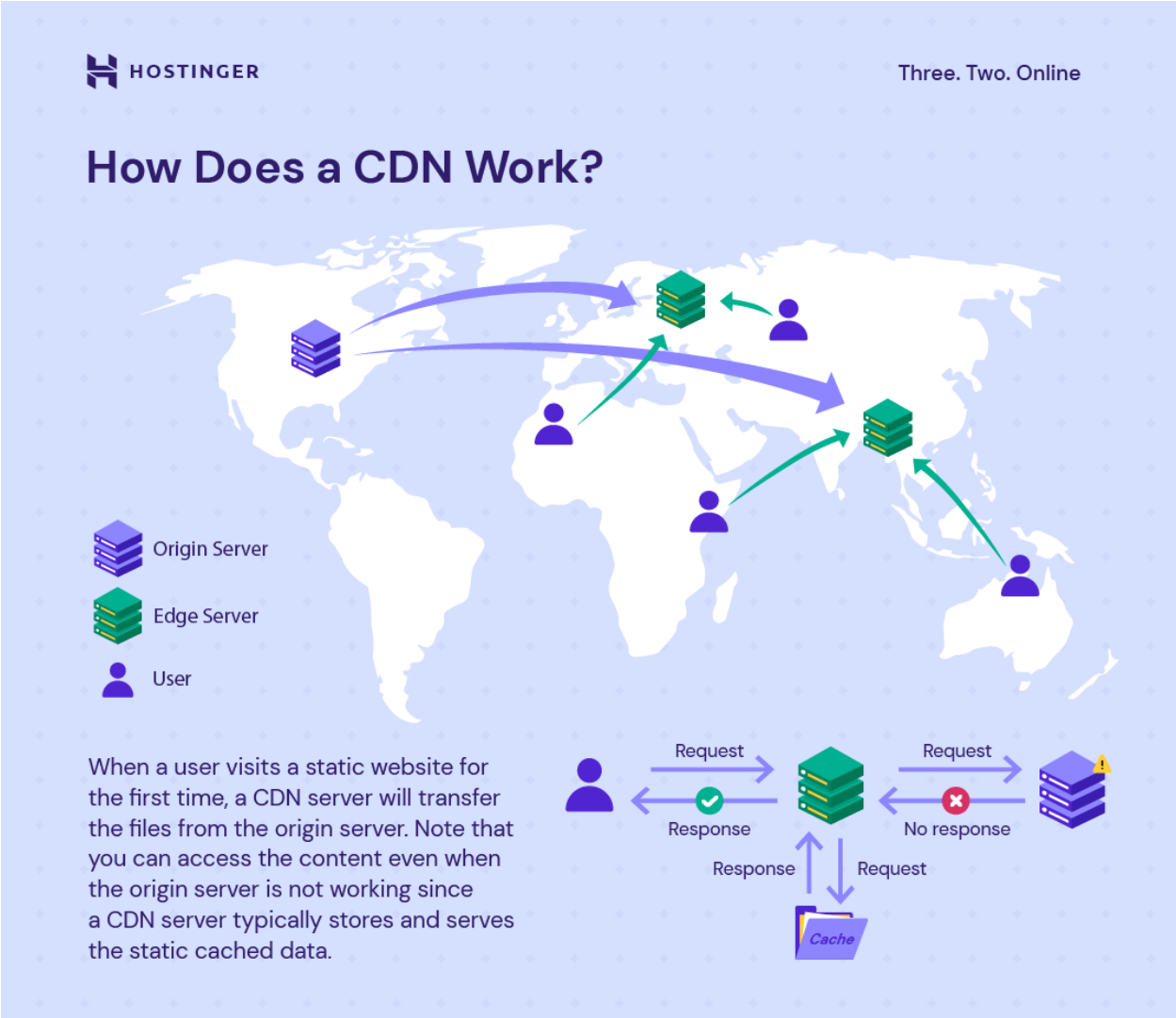

Figure 10: CDN [21]

The benefits of using a CDN are:

- Reduced hosting costs by conserving the amount of bandwidth it takes to handle your traffic through the multiplication of points of presence. A content delivery network uses optimization methods like caching, which involves placing static files into temporary storage on different computers for ease of access. These methods help reduce the overall server workload and bandwidth consumption. Static content refers to the data delivered to the end-user without any modification. These files stay the same for all users, regardless of who requests them. Examples include media files like images and videos as well as HTML, CSS, and JavaScript files. On the other hand, dynamic content is the data that cannot be cached on an edge server. This is because the content delivered may differ according to variables like user credentials or geographical locations. However, an advanced CDN's network infrastructure and request-routing algorithms can help streamline the delivery of dynamic content. In short, the combined technologies used by a CDN are useful for providing load balancing and reducing bandwidth costs.
- Increased overall speed and performance by using an effective content delivery network. Implementing numerous website optimization strategies will also help improve site speed and performance.
- Improved security since CDNs deflect web traffic away from the original server to proxies, making the primary source practically invisible, offering a better protection of sensitive data. This, in combination with DDoS filters and spreading queries over several locations help mitigate traffic explosions. Also SSL/TLS certificates secure information transfers using data authentication and encryption. These certificates ensure data security protocols are followed by enabling only the intended recipient to access and view the information.
- Optimization of content distribution and availability, by distributing traffic over multiple CDN servers, your core network infrastructure will have a lighter burden to carry. This system also ensures that in the case of some servers experiencing an outage, people can still access the website as other operational servers will handle the network traffic.

2. PROBLEM STATEMENT

2.1 System model and architecture

2.1.1 System model

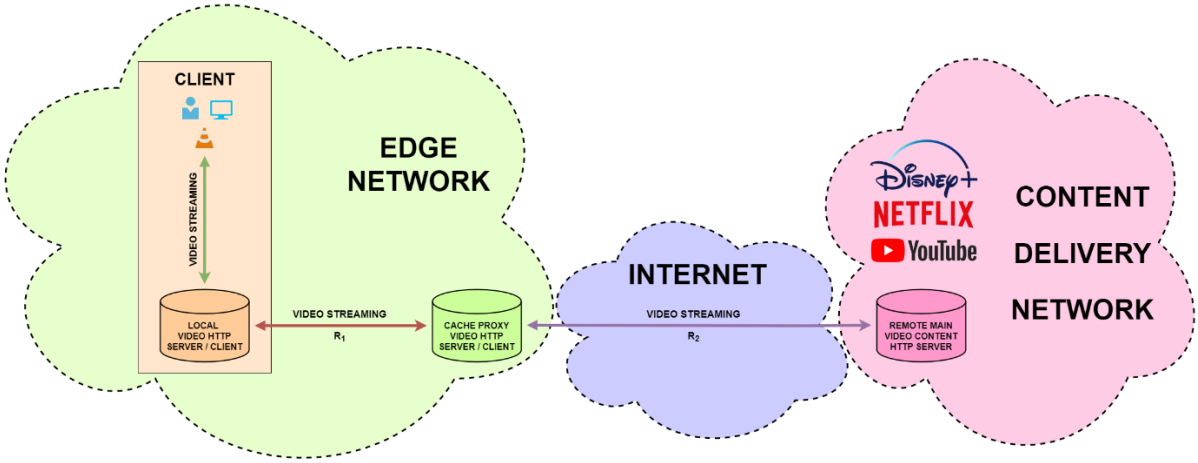

Figure 11: System model architecture

The model we propose was initially presented in [24] and consists of the three following distinct and independent network nodes (Fig. 11):

- A remote **main** video content HTTP server located in the CDN
- A **cache proxy** video HTTP server located at the MEC
- A **local** video HTTP server / client located in the end user's LAN

Each of these nodes uses DASH to either request and receive or be requested and serve (or both) video content in the form of DASH-compatible video segments. The link between the remote main video content HTTP server and the video cache proxy HTTP server (a.k.a. the proxy-to-main link) has an attainable data rate $\boldsymbol{R_2}$ (Mbps) while the link between the video cache proxy HTTP server and the local video HTTP server / client (a.k.a. the proxy-to-client link) has an attainable data rate $\boldsymbol{R_1}$ (Mbps), where $\boldsymbol{R_1} \geq \boldsymbol{R_2}$ (hence the advantage of local video content caching).

From this point and for the rest of this thesis, we are going to use the following shorter names for the three aforementioned nodes:

Table 2: Short components names

Component	Name
Remote **main** video content HTTP server	**Main**
Video cache **proxy** HTTP server	**(Cache) Proxy**
Local video HTTP server / client	**Local**

Main is located within the CDN, has virtually unlimited storage and contains every single video and audio DASH segment, which are always available and can be served at any time. Cache proxy is located at the edge of the network, e.g. at a MEC Base Station, in close proximity to the end user, has limited cache storage and can only store a subset of these segments. Local is located in the end user's LAN, acts like a client/server extension to the video request and playback software, and is responsible for forwarding the video player's content requests to the cache proxy. It can also cache a local copy of the received content.

In this architecture, whenever a client requests a segment from the cache proxy which is unavailable from the cache storage, the cache proxy forwards the segment request to main, from which it always successfully fetches the content, caches a local copy if possible and forwards it to local. In our architecture we call this limited cache space a buffer and we symbolize the buffer size with the letter $\boldsymbol{L}$ (MB).

We have a set $\mathbb{F}$ of original video files. Every original video file $\boldsymbol{f} \in \mathbb{F}$ with total duration $\boldsymbol{D_f}$ (seconds) is transcoded into every available distinct video resolution / bitrate in the video resolution set $\boldsymbol{R}$. Afterwards, every transcoded video $\boldsymbol{f_r}$ of video resolution $\boldsymbol{r} \in \boldsymbol{R}$ gets segmented into $\boldsymbol{M_f} = \left\lceil \frac{\boldsymbol{D_f}}{\boldsymbol{T_f}} \right\rceil$ evenly-timed DASH-compatible segments (and so does the audio), where T_f is the selected duration of every segment for the specific video. Each generated segment has a size $\boldsymbol{S_{f,r,i}}$ (bits) where $i = 1, 2, \dots, M_f$ is the unique identifier / ID / index of the segment. In each of these segment sizes we have included the size of the respective audio file. So, for every original video file f we end up with $7 \cdot n$ segments ($6 \cdot n$ for the video and another n for thed audio). While all these segments have the exact same duration T_f, their size differs due to the difference in their content / signals. While the durations of the video files are fixed, the value for the segment duration T_f can vary between different original videos, because it must be selected so that the generated segments do not end up being too big in size, as the time required to stream each big segment will increase a lot, eliminating the advantages of the DASH technique, while if the segments end up being too small in size, a lot of segments will be created and the network will flood with many HTTP requests in order to stream all of these small segments. So, the value for the segment duration T_f must be balanced for each video file independently, depending on the video duration, the video size and the data rate R_2.

2.1.2 Problem statement

The problem we are trying to solve consists of the following three questions – parts:

- How to request and stream DASH segments so that the total network usage time is minimized?
- How to request and stream DASH segments so that the usage of the available network resources, i.e. the data rates R_1 and R_2, are maximized?
- Which DASH segments should be cached in combination with the streaming algorithm of our choice, so that a high QoE is achieved for the end user?

2.2 Resolution selection algorithms

Video playback software which implements DASH, provides us with multiple resolution selection algorithms for each video segment, from which we distinguished the following two:

- **Predictive**: For the next segment to request, the software tries to make a “prediction”, taking into account the available resolutions and the time required to stream the current segment. Using this information, it tries to select a segment resolution which can achieve the highest video bit rate possible while minimizing the required streaming time, so that this choice results in a seamless playback and a high QoE, without causing stalls or re-buffering events. In case delays arise during streaming due to an insufficient data rate, the software identifies them and can choose a lower resolution for the next segment according to the achieved bandwidth from the current segment, so that it avoids further delays. In case the currently requested segment is streamed faster than anticipated, the software identifies this as a speedup and may choose a higher resolution for the next

segment according to the achieved bandwidth from the current segment, so that a higher quality video playback and QoE are achieved.
- **Fixed**: The software keeps requesting a fixed segment resolution constantly, regardless of streaming delays or channel capacity underutilization.

2.3 Video segment request and streaming algorithms

Since the client will request the DASH-compatible segments from the cache proxy, we distinguish the two following segment querying scenarios in the cache proxy:
- **CACHE HIT**: The requested segment is available from the cache proxy's buffer, so it gets served directly from it and delivered to the client through local
- **CACHE MISS**: The requested segment is unavailable from the cache proxy's buffer, so it gets served from main and delivered to the client through proxy first, which may cache a copy, and then through local.

Afterwards, we wanted to decide upon the cache proxy's procedure for its response. So, we came up with two simple video segment request and streaming algorithms, which can only request and serve a single segment at each time. So, all requests and their responses are performed in a serialized one-after-another way. The only difference between these two algorithms the way the cache proxy handles the case of a CACHE MISS, since this is the only case in which the segment will be streamed to the client from main through the cache proxy:
- **Send-After-Get (SAG)**: in the case of a CACHE MISS, the cache proxy **first requests and receives all bytes** of the missing segment from main and **afterwards it starts forwarding these bytes to local**. Obviously, since the segment has to be completely received from main before it gets forwarded to the client, this algorithm adds a huge latency to the total video streaming performance since it utilizes R_1 and R_2 alternately.
- **Send-While-Get (SWG):** in the case of a CACHE MISS, proxy **first requests the missing segment from main** and then it **forwards each chunk of bytes it receives from main to local as soon as it receives it**, without waiting to receive all bytes first. This exact streaming algorithm is also used by the local video HTTP server / client.

Fig. 12 displays the behavior similarity between SAG and SWG in the case of a CACHE HIT while Fig. 13 displays the behavior difference between SAG and SWG in the case of a CACHE MISS. The differences are shown through the HTTP GET requests that are being sent, alongside their respective responses, with respect to time. We can clearly see that in the case of a CACHE HIT both algorithms have the exact same behavior, since the segment had been cached before it was requested, so the cache proxy immediately starts forwarding all bytes of the segment to local, in the form of chunks of bytes.

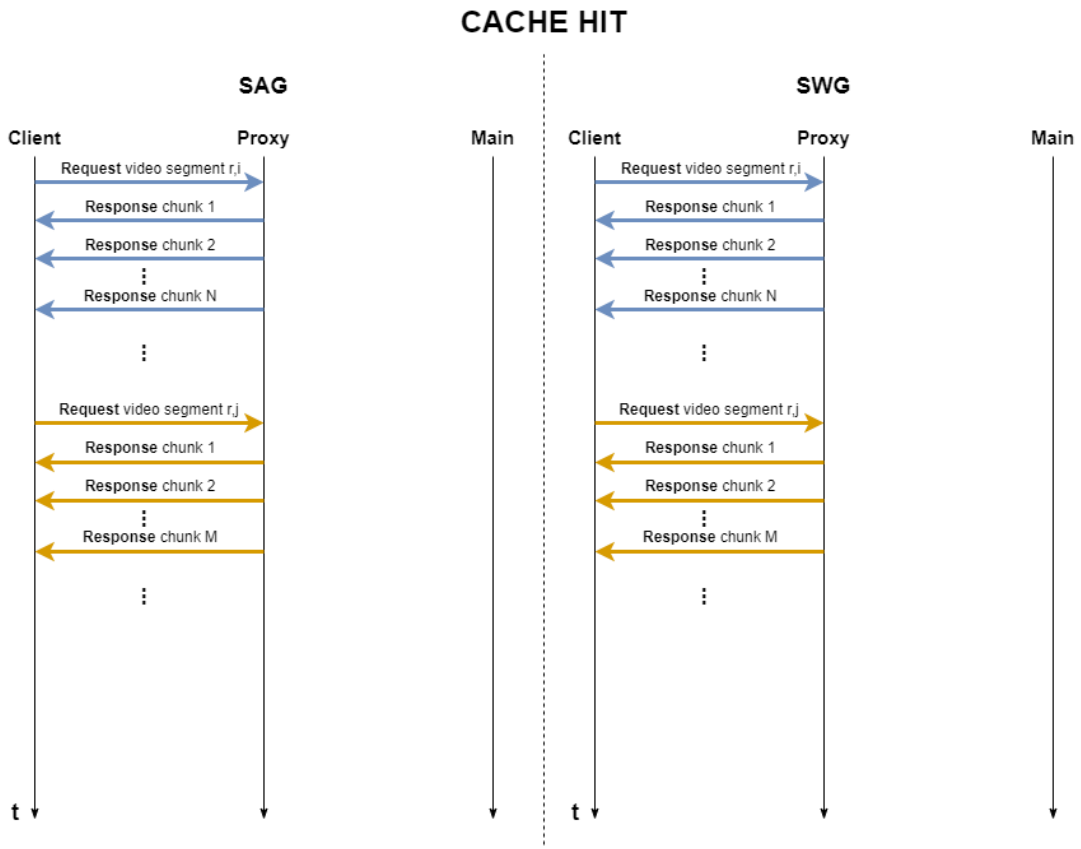

Figure 12: SAG vs. SWG - behavior difference during a CACHE HIT

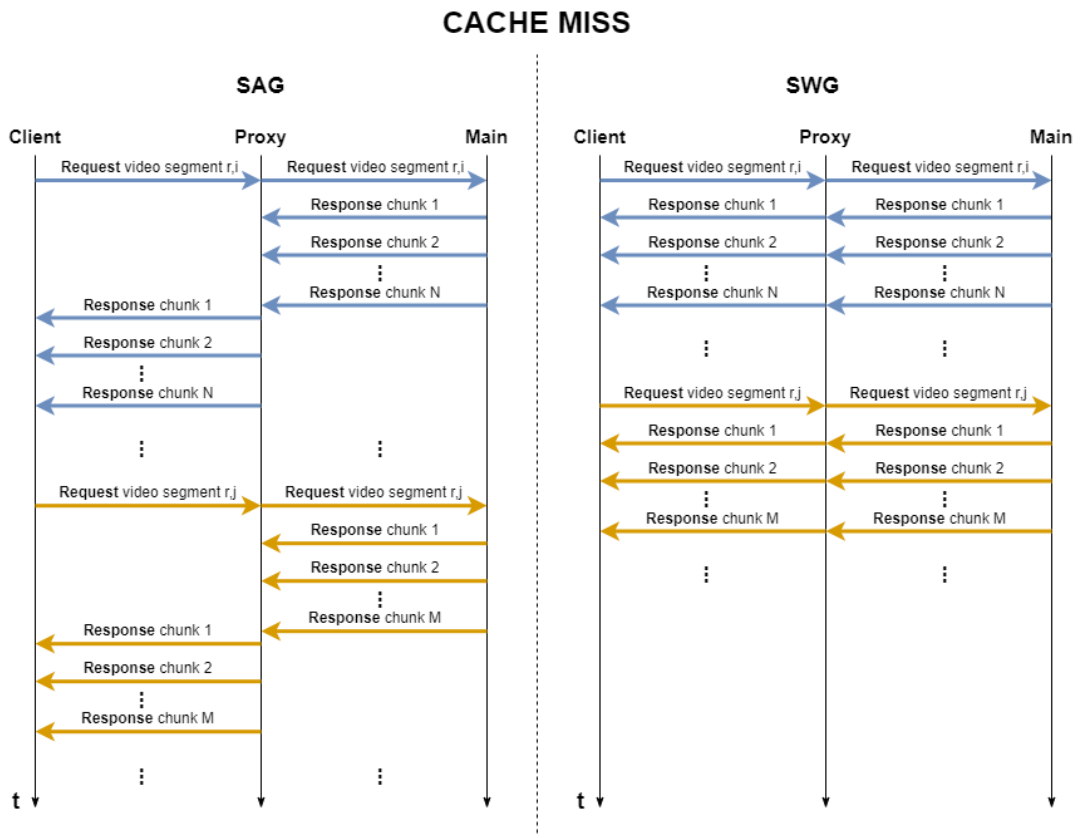

Figure 13: SAG vs. SWG behavior difference during a CACHE MISS

For a more in-depth understanding of how these proxy algorithms work, let's take a look at the following pseudocode segments:

SEND-AFTER-GET

```
INPUT:    cache[M], main_ip, http_server, http_client

1. request = http_server.get_incoming_request()
2. IF request.segment ∈ cache
   2.1. request.respond(HTTP_200, file_metadata)
   2.2. file = open(cache[request.segment], "read")
   2.3. WHILE NOT EOF(file):
     2.3.1. chunk = file.read(4096)
     2.3.2. request.response.socket.send(chunk)
   2.4. file.close()
3. ELSE IF request.segment ∉ cache:
   3.1. http_client.request(request.segment, main_ip, HTTP_GET)
   3.2. main_response = http_client.get_response()
   3.3. IF main_response.http_code < HTTP_300:
     3.3.1. request.response.socket.respond(HTTP_200, main_response.metadata)
     3.3.2. file = open(cache[request.segment], "write")
     3.3.3. counter = 0
     3.3.4. WHILE counter < main_response.metadata.content_length:
          3.3.4.1. chunk = main_response.socket.read()
          3.3.4.2. file.write(chunk)
          3.3.4.3. counter = counter + size_of(chunk)
     3.3.5. file.close()
     3.3.6. file = open(cache[request.segment], "read")
     3.3.7. WHILE NOT EOF(file):
          3.2.7.1. chunk = file.read()
          3.2.7.2. request.response.socket.send(chunk)
     3.3.8. file.close()
   3.4. ELSE IF main_response.http_code == HTTP_404
     3.4.1. request.respond(HTTP_404)
4. request.close_connection()
```

SEND-WHILE-GET

```
INPUT: cache[M], main_ip, http_server, http_client

1. request = http_server.get_request()
2. IF request.segment ∈ cache
   2.1. request.respond(HTTP_200, file_metadata)
   2.2. file = OPEN(cache[request.segment], "read")
   2.3. WHILE NOT EOF(file):
      2.3.1. chunk = file.read()
      2.3.2. request.response.socket.send(chunk)
   2.4. CLOSE(file)
3. ELSE IF request.segment ∉ cache:
   3.1. http_client.request(request.segment, main_ip, HTTP_GET)
   3.2. main_response = http_client.get_response()
   3.3. IF main_response.http_code < HTTP_300:
     3.3.1. request.respond(HTTP_200, main_response.metadata)
     3.3.2. file = OPEN(cache[request.segment], "write")
     3.3.3. counter = 0
     3.3.4. WHILE counter < main_response.metadata.content_length:
          3.3.4.1. chunk = main_response.socket.read()
          3.3.4.2. request.response.socket.send(chunk)
          3.3.4.2. file.write(chunk)
          3.3.4.3. counter = counter + SIZE_OF(chunk)
     3.3.5. CLOSE(file)
   3.4. ELSE IF main_response.http_code == HTTP_404
     3.4.1. request.respond(HTTP_404)
4. request.close_connection()
```

If we take a look at steps 2.1 to 2.4 of both algorithms, we can see that in the case of a CACHE HIT the two algorithms perform the exact same operations / steps to stream the requested and cached segment from proxy to local. However, by looking at steps 3.3.2 to 3.3.8, we can see that in the case of a CACHE MISS the SAG algorithm performs two distinct loops with the same complexity, one to stream the segment from main to proxy and one to stream the segment from proxy to local, which means it has to fetch the whole missing segment from main before it initiates the streaming procedure towards the client. On the other hand, the SWG algorithm performs both operations in a single loop by streaming every chunk of bytes of the missing segment to the client as soon as it receives it, which significantly reduces the streaming delay. In both proxy streaming algorithm variations, the size of each chunk of data being transferred should be equal to the page size of the Operating System (OS) of the node.

2.4 Theoretical performance evaluation

In section 2.3, we mentioned that the difference between SAG and SWG is their behavior in the case of a CACHE MISS. What we mostly care about is the elapsed time between the moment a segment was requested by the client and the time the client received it. We define this time as the **delivery delay** d. Assuming that a video streaming procedure consists of M_f segments and that a video segment i with video resolution r and size $S_{f,r,i}$ is being streamed, we distinguish the following two cases:

- **In the case of a CACHE MISS**, the delivery delay is given by the following formulas:
  - $d_{SAG} = \frac{S_{f,r,i}}{R_2} + \frac{S_{f,r,i}}{R_1} = S_{f,r,i}\frac{R_1 + R_2}{R_1 R_2}$ seconds
  - $d_{SWG} = \frac{S_{f,r,i}}{\min(R_1,R_2)} \overset{R_1 \geq R_2}{=} \frac{S_{f,r,i}}{R_2}$ seconds

  Since $R_1 \geq R_2$, it is very clear that $d_{SAG} \geq d_{SWG}$. So, we expect SWG to have a better overall performance because of its lower delivery delay.
- **In the case of a CACHE HIT**, although the proxy-to-client link stays active for the whole time this segment is being streamed using either SAG or SWG, the proxy-to-main link stays idle for $T_{idle}^{i} = \frac{S_{f,r,i}}{R_1}$ seconds, which means that this channel's capacity is underutilized. So, even with the use of SWG a significant portion of the bandwidth of R_2 is wasted.

In both SAG and SWG, since $R_1 \geq R_2$, R_1 is always underutilized when segments are missing from the cache storage and need to be fetched from main. In the case of SAG, the total idle time for R_1 is given from the following equation:

$T_{idle}^{R_1} = \sum_{i=1}^{n}(1 - H_{r,i})\frac{S_{r,i}}{R_2}$ where $H_{r,i} = \begin{Bmatrix} 1, & if \quad cache\ hit \\ 0, & cache\ miss \end{Bmatrix}$,

while in both SAG and SWG, the total idle time for R_2 is given from the following equation:

$T_{idle}^{R_2} = \sum_{i=1}^{n} H_{r,i}\frac{S_{r,i}}{R_1}$ where $H_{r,i} = \begin{Bmatrix} 1, & if \quad cache\ hit \\ 0, & otherwise \end{Bmatrix}$.

2.5 Simple caching algorithms

The simplest caching technique that can be used is the random segment caching, in which segments to be cached are randomly selected from the available collection of different segments. However, we can distinguish the following two random caching algorithms:

- Random caching: orders all segments of a target video content randomly (all video resolutions / bitrates) and caches files sequentially until it fully fills the available cache storage. Segments that cannot fit into the available cache storage are skipped.

- Ripple caching []: has the same behavior as Random caching, except that in this algorithm only the segments of the highest available video bitrate are considered for caching.

These random caching algorithms randomly select segments to cache one-by-one, skipping segments that cannot fit into the available cache storage, to maximize cache utilization by storing as many segments as possible, without exceeding the available cache storage size. As we can imagine, this technique does not provide any special behavior or features, because of its random nature, except for the case in which the total cache size is really big, so that a lot of segments can be cached.

```
RANDOM CACHING
Input: L, segments, M

1. Cache = Ø, Failures = 0
2. WHILE Failures < M
   2.1. s = randomly_select_one(segments)
   2.2. IF SIZE_OF(Cache ∪ {s}) ≤ L
     2.2.1. Cache = Cache ∪ {s}
   2.3. ELSE
     2.3.1. Failures = Failures + 1
```

```
RIPPLE CACHING
Input: L, segments, M, R

1. Cache = Ø, Failures = 0
2. WHILE Failures < M
   2.1. s = randomly_select_one_of_resolution(segments, R)
   2.2. IF SIZE_OF(Cache ∪ {s}) ≤ L
     2.2.1. Cache = Cache ∪ {s}
   2.3. ELSE
     2.3.1. Failures = Failures + 1
```

2.6 Experimental performance evaluation and algorithm comparisons

We ran some sets of experiments in order to compare the three video segment request and streaming algorithms, using the software we developed from scratch and the video file described in Appendix I, which has a duration $D_f = 634.6$ seconds and which was segmented using a segment duration $T_f = 10$ seconds, resulting in 64 distinct video segments per video resolution and another 64 segments for the audio. We only used a fixed video resolution instead of predictive, because the advantage of MS-SWG is diminished in the case of a predictive video resolution selection. We evaluate the performance difference using the mean data rates $\overline{\boldsymbol{R_1}}$ and $\overline{\boldsymbol{R_2}}$, the **time windows of segments streaming**, the **time windows of the stallings** that occurred, and the **Mean Opinion Score (MOS)** based on the number of stallings occurred, which is an indicator of the QoE for the end user (higher MOS value means a better QoE). MOS will be explained in detail in chapter 4.

2.6.1 SAG vs. SWG

The first set of experiments was a comparison between SAG and SWG for a scenario with $\boldsymbol{R_1 = 7}$ **Mbps**, $\boldsymbol{R_2 = 4}$ **Mbps**, $\boldsymbol{L = 500}$ **MB**, **Random caching** and with a **fixed video resolution of 1440p** which has a total file size of 418.80 MB. Fig. 14 contains the development of the whole video streaming procedure for both SAG and SWG, while Fig. 15 contains a time window of the first 200 seconds for a clearer view. At the right side of

the charts in Fig. 14 it seems as if the SAG suddenly became faster, but this is an optical illusion since the explanation for this behavior is the fact that the x-axis with the time is compressed, since there is nothing to compare any more.

From both figures, we can see that SWG completes the whole video streaming procedure about 310 seconds earlier than SAG, while also achieving higher mean data rates $\overline{\boldsymbol{R_1^{SWG}}} = \boldsymbol{4.21}$ Mbps and $\overline{\boldsymbol{R_2^{SWG}}} = \boldsymbol{3.21}$ Mbps versus $\overline{\boldsymbol{R_1^{SAG}}} = \boldsymbol{2.23}$ Mbps and $\overline{\boldsymbol{R_2^{SAG}}} = \boldsymbol{1.64}$ Mbps. Furthermore, with the use of SWG the total number of stallings is much smaller and the MOS starts dropping after about 490 seconds have passed, compared to SAG which suffers more stallings in total and its MOS drops as early as about 100 seconds. We can also see that in the case of CACHE HITs, the instantaneous data rate R_2 drops to zero with both algorithms, which proves again that the channel capacity of the proxy-to-main link is underutilized, resulting in a less than optimal total network utilization.
This behavior was expected from the theoretical performance evaluations and now it has been verified experimentally. However, we can see that even with the use of SWG the QoE for the end user is relatively poor since several stallings occur and the total time required to stream a video of total duration equal to 634.6 seconds was about 760 seconds, which is another 125.4 seconds more. So, SWG, while better is still not good enough.

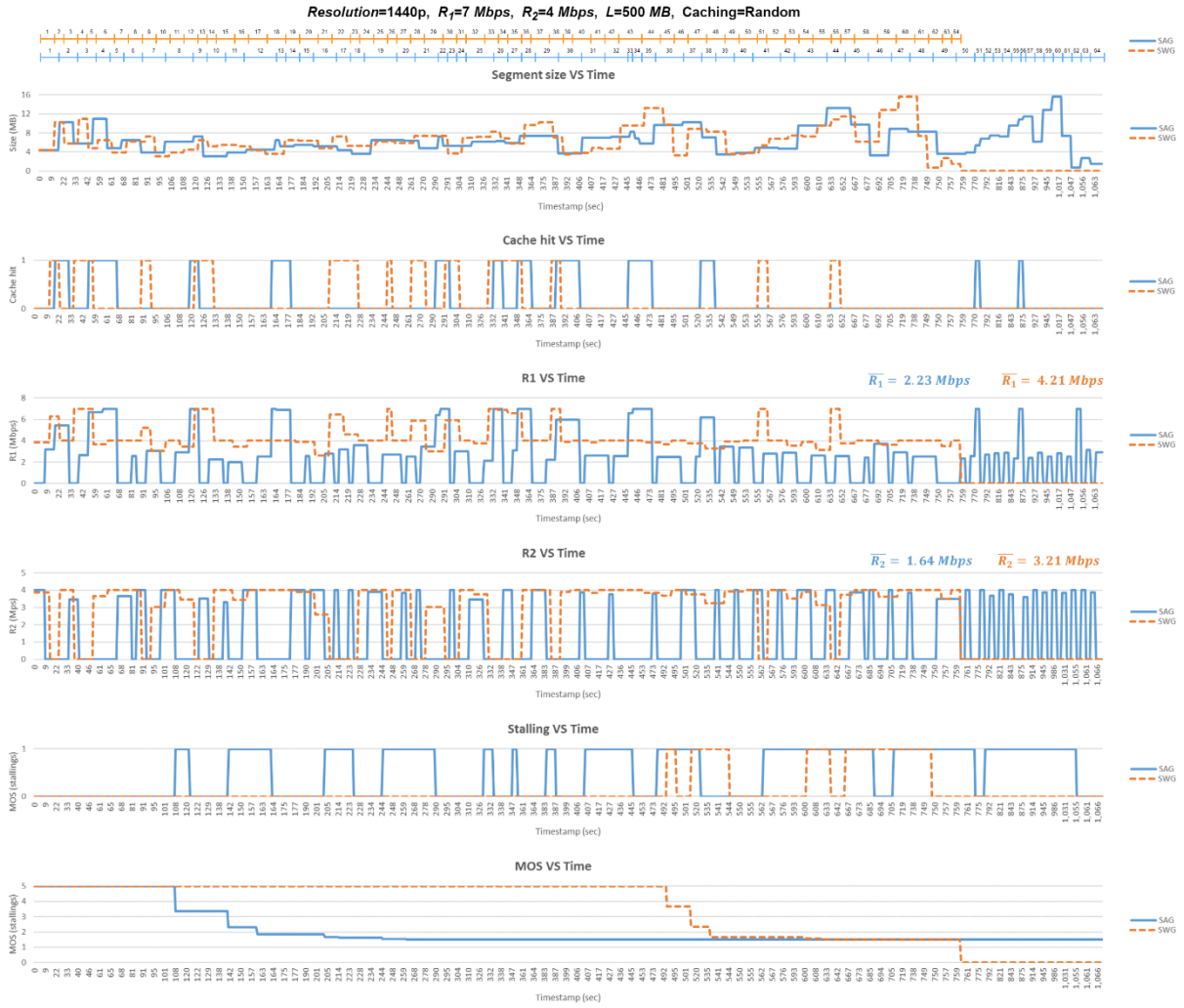

Figure 14: SAG vs. SWG for 1440p, R_1=7 Mbps, R_2=4 Mbps, L=500 MB and Random caching

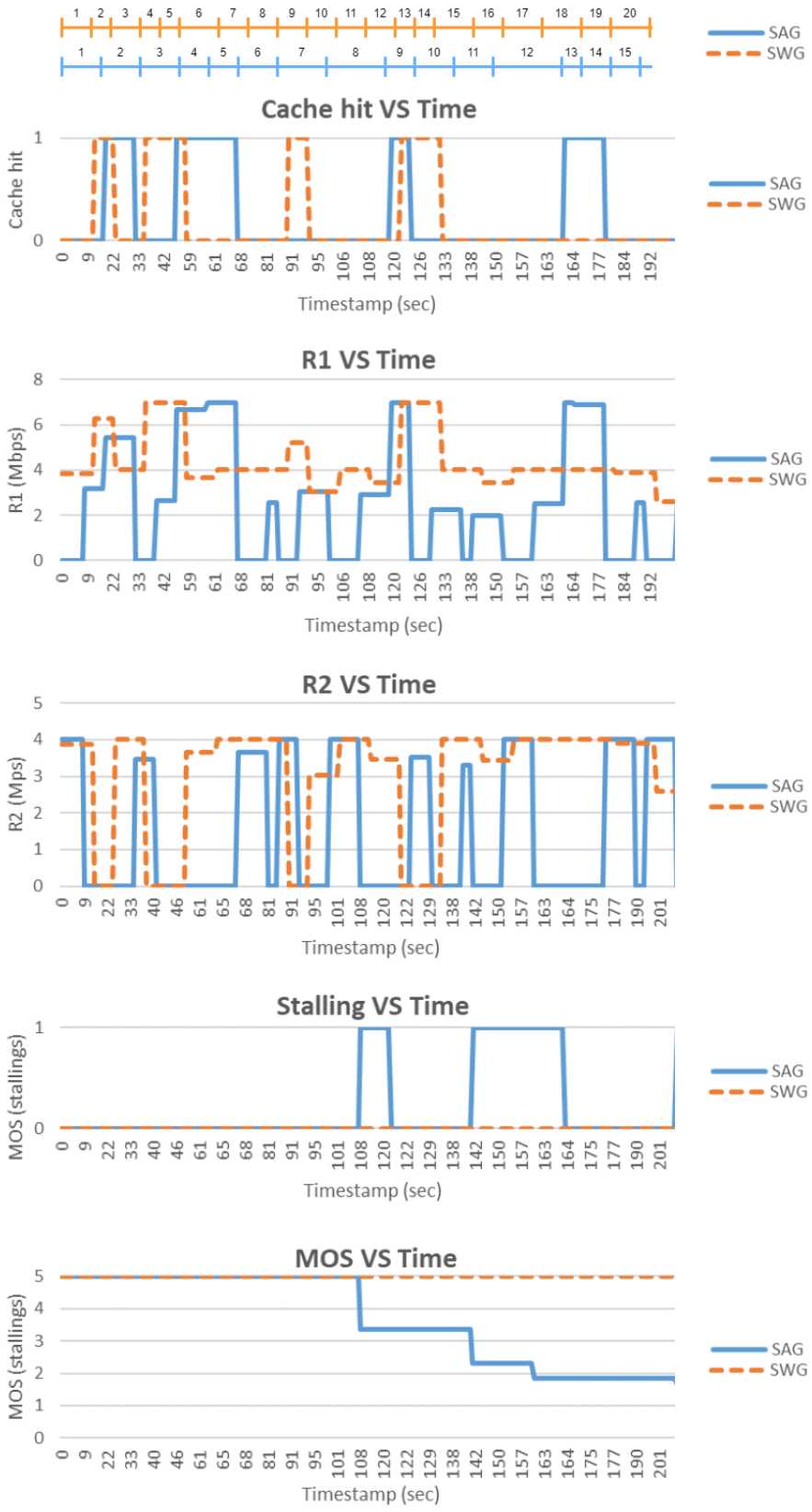

Figure 15: SAG vs. SWG for 1440p, R_1=7 Mbps, R_2=4 Mbps, $L = 500$ MB and Random caching for the first 200 seconds

3. Proposed solution

3.1 Multi-segment video streaming

From chapter 2 we deduced that both aforementioned segment request and streaming algorithms are sub-optimal because even in the case of a cache hit, which is the positive outcome of a segment request to the cache proxy, a significant portion of the available channel capacity of proxy-to-main link is wasted. The reason behind this is their serialized nature, which means that we could overcome this problem if we made sure that when a CACHE HIT occurs, some future and missing segment will be requested and streamed from main to the cache proxy simultaneously, so that R_2 is utilized when missing segments exist. Furthermore, since $R_1 \geq R_2$ we also want to make sure that while a missing segment is being streamed from main to the client through the cache proxy, parts of some future and cached segment are simultaneously streamed from the cache proxy to the client so that R_1 is fully utilized.

We wanted to maximize both utilizations as much as possible through the video streaming algorithm, so we came up with an improved version of the SWG algorithm. Taking advantage of the multi-processing and multi-threading capabilities of modern CPUs, we developed the **Multi-Segment Send-While-Get (MS-SWG)** segment request and streaming algorithm. MS-SWG can serve multiple segment requests in parrallel by offloading each incoming segment request in a new process / thread to be handled independently. For the synchronization of these parallel tasks and the proper ordering of the responses based on the IDs of the segments, so that a seamless video playback is achieved, a priority queue is employed. Since these parallel requests will be forwarded from the cache proxy to main, again in parallel, this algorithm and software structure must be implemented both in the cache proxy and in main. The client must also adopt this multiple parallel segments requests architecture. Fig. 16 gives an overview of this improved system model architecture.

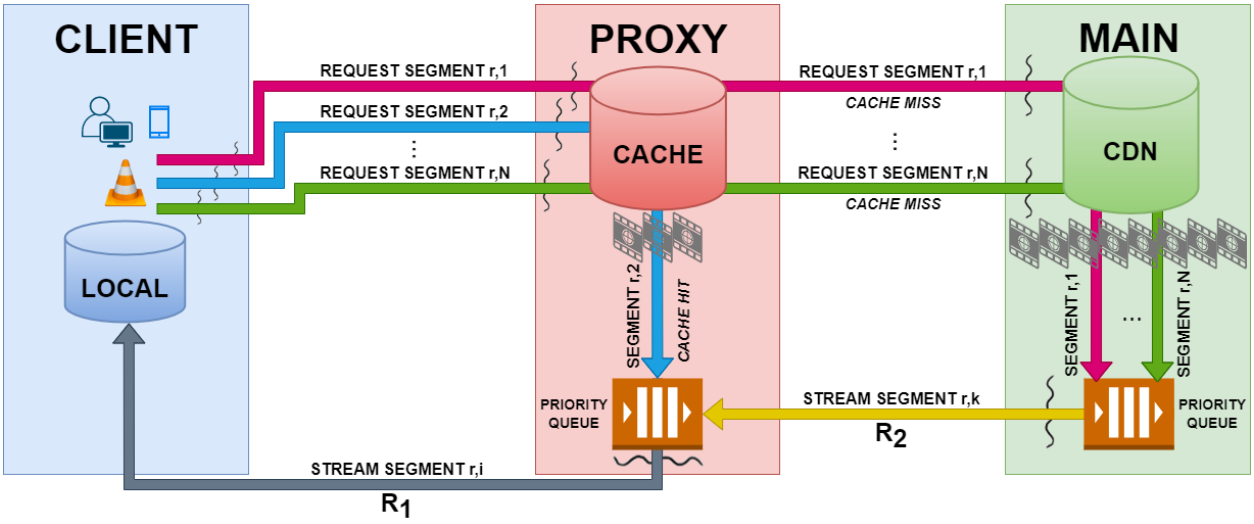

Figure 16: Video streaming platform with parallel segment requests

Given a segment i which has been requested by local, its priority p within the priority queue is given by the following equation: $p = 2i + 1$. Priority queues are based on heaps, which means that a lower value of p results to a higher segment priority. So:

1. $i < j \Rightarrow$ segment i has a higher priority than segment j, maintaining proper segment streaming order.
2. For every segment ID i both the respective video and audio segments have the same priority within the priority queue, since they are parts of the same segment.

For example, the segment with the file name **video_2160_31.m4s**, which is the 31st video segment with resolution 2160p) has a priority of **63** and so does the corresponding audio segment with the file name **video_audio_31.m4s** (which is the 31st audio segment). This choice was made so that for every segment ID, the client does not have to wait for the video segment to get completely streamed before the corresponding audio segment starts streaming. The reason behind this choice is that since the size of video segments is always bigger than the size of their corresponding audio segments, if the audio segments had a higher priority than the video ones the video playback would delay, while if the video segments had a higher priority it would take much more time for the streaming procedure to complete due to waiting for the video to get completely streamed before even begin to stream the audio. This priority assignment is used to make sure that even though chunks of different segments can be multiplexed in the queue, the actual segment delivery matches the order in which segments need to be delivered to the video player to ensure a good video playback experience.

Since multiple segments can be requested and served in parallel, we distinguish the two following segment serving scenarios in multi-segment SWG proxy streaming algorithm:

- **CACHE HIT**: Since the segment had been cached before it was requested by the client, the respective task which handles this request:
    1. Calculates the segment's priority.
    2. Splits the segment into chunks.
    3. Places each chunk in the priority queue, assigning the same priority in all chunks since they are all parts of the same segment, but with an additional automatically generated priority property to maintain proper chunk ordering.
- **CACHE MISS**: Since the segment had been missing from the cache by the time it was requested by the client, the respective task which handles this request:
    1. Requests this segment from main.
    2. Calculates the segment's priority.
    3. Places every chunk it receives from main into the priority queue as soon as it receives it, assigning the same priority in all chunks since they are all parts of the same segment, but with an additional automatically generated priority property to maintain proper chunk ordering.
    4. Caches a local copy of the segment in its buffer.

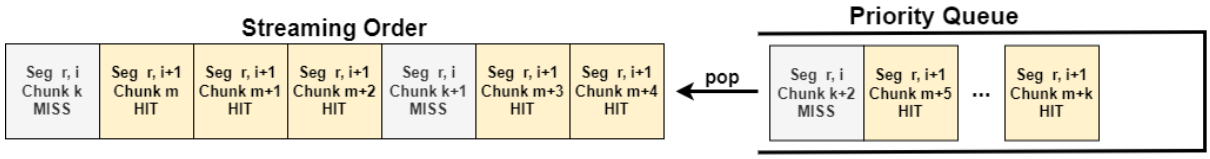

Figure 17: Priority queue's output for segments of different priority and cache status

In Fig. 17 we can see an example snapshot of proxy's priority queue state and output in the case of a missing segment i which has a higher priority than a following cached segment $i+1$. Since $R_1 \geq R_2$, their chunks will get multiplexed as they are served from the priority queue, due to the fact that although segment i (missing) has a higher priority than segment $i+1$ (cached), data rate R_2 will limit the speed at which chunks of segment i get fetched from main and pushed into the queue, which is why in the meantime chunks of segment $i+1$ are served from the queue, as all chunks of any cached segment have been pushed into the queue beforehand. However, any received chunks of segment i will be pushed to the top of the queue as soon as they get fetched from main.

For a more in-depth understanding of how both the multi-segment proxy and the multi-segment client function, let's take a look at the following pseudocode segments:

MULTI-SEGMENT SEND-WHILE-GET – INCOMING REQUEST HANDLING

```
INPUT:    tasks[N], http_server

1. request = http_server.get_request()
2. tasks[k] = create_task(request)
```

MULTI-SEGMENT SEND-WHILE-GET – ASYNC REQUEST RESPONSE AND ENQUEUE

```
INPUT:    task, cache[M], priority_queue, main_ip, http_client

1. priority = calculate_priority(task.request.segment)
2. IF task.request.segment ∈ cache
   2.1. task.request.respond(HTTP_200, file_metadata)
   2.2. file = OPEN(cache[task.request.segment], "read")
   2.3. counter = 0
   2.3. WHILE NOT EOF(file):
      2.3.1. chunk = file.read()
      2.3.2. counter = counter + SIZE_OF(chunk)
      2.3.3. is_final = FALSE
      2.3.4. IF counter == SIZE_OF(file):
         2.3.4.1. is_final = TRUE
      2.3.5. priority_queue.enqueue(task.request, chunk, is_final, priority)
   2.4. CLOSE(file)
3. ELSE IF task.request.segment ∉ cache:
   3.1. http_client.request(task.request.segment, main_ip, HTTP_GET)
   3.2. main_response = http_client.get_response()
   3.3. IF main_response.http_code < HTTP_300:
      3.3.1. task.request.respond(HTTP_200, main_response.metadata)
     3.3.2. file = OPEN(cache[task.request.segment], "write")
     3.3.3. counter = 0
     3.3.4. WHILE counter < main_response.metadata.content_length:
          3.3.4.1. chunk = main_response.socket.read()
          3.3.4.2. counter = counter + SIZE_OF(chunk)
          3.3.4.3. is_final = FALSE
          3.3.4.4. IF counter == SIZE_OF(file):
               3.3.4.4.1. is_final = TRUE
          3.3.4.5. priority_queue.enqueue(task.request, chunk, is_final, priority)
          3.3.4.6. file.write(chunk)
     3.3.5. CLOSE(file)
   3.4. ELSE IF main_response.http_code == HTTP_404
      3.4.1. task.request.respond(HTTP_404)
4. task.request.close_connection()
```

MULTI-SEGMENT SEND-WHILE-GET – ASYNC REQUEST SERVING

```
INPUT:    priority_queue

1. WHILE TRUE:
   1.1. request,  chunk, is_final = priority_queue.dequeue()
   1.2. request.response.socket.send(chunk)
   1.3. IF is_final:
      1.3.1. request.close_connection()
```

MULTI-SEGMENT CLIENT – REQUESTS INITIALIZATION

```
INPUT:    tasks[k], resolution

1. FOR i FROM 1 TO N:
   1.1. tasks[i] = create_task(resolution, i)
```

```
MULTI-SEGMENT CLIENT – ASYNC SEGMENT REQUEST
INPUT: k, resolution, proxy_ip, http_client

1. request = segment(resolution, k)
2. http_client.request(request.segment, proxy_ip, HTTP_GET)
3. proxy_response = http_client.get_response()
4. IF proxy_response.http_code < HTTP_300:
    4.1. file = OPEN(request.segment, "write")
    4.2. counter = 0
    4.3. WHILE counter < proxy_response.metadata.content_length:
      4.4.1. chunk = main_response.socket.read()
      4.4.2. file.write(chunk)
       4.4.3. counter = counter + SIZE_OF(chunk)
    4.4. CLOSE(file)
5. request.close_connection()
```

The advantage of this architecture is that in even the case of a CACHE HIT, one of the asynchronous tasks created in the cache proxy keeps fetching segment content from main. So, the channel capacity of the proxy-to-main link is never underutilized, which also means that the total network usage time is reduced. Also, since $R_1 \geq R_2$, in the case of a CACHE MISS while a non-cached segment i is being streamed to the client from main through proxy with a data rate R_2, (part of) a following cached segment j is simultaneously being streamed to the client with data rate $R_1 - R_2$ because of the priority queue, taking full advantage of the channel capacity of the proxy-to-client link so that it is fully utilized. The disadvantage of this architecture is that it can only work when all video segment resolutions are known beforehand, in order to create all the parallel requests. This means that this architecture cannot work with an adaptive video resolution, but can only work with a fixed / constant video resolution.

3.2 Experimental performance evaluation and algorithm comparisons

3.2.1 MS-SWG vs. SWG

The second set of experiments was a comparison between MS-SWG and SWG (and SAG) for a scenario with the same parameters with the first set of experiments, $\boldsymbol{R_1 = 7}$ **Mbps**, $\boldsymbol{R_2 = 4}$ **Mbps**, $\boldsymbol{L = 500}$ **MB**, **Random caching** and with a **fixed video resolution of 1440p** which has a total file size of 418.80 MB. Fig. 18 contains the development of the whole video streaming procedure for both MS-SWG and SWG, while Fig. 19 contains a time window between 470 and 650 seconds for a clearer view. At the right side of the charts in Fig. 18 it seems as if the SAG suddenly became faster, but this is an optical illusion since the explanation for this behavior is the fact that the x-axis with the time is compressed, since there is nothing to compare any more.

From both figures, we can see that MS-SWG completes the whole video streaming procedure about 160 seconds earlier than SWG, requiring about 590 seconds to completely stream a video of total duration equal to 634.6 seconds, reducing the total streaming duration and with it the total network usage time, while also achieving higher mean data rates $\overline{\boldsymbol{R_1^{MS-SWG}}} = \mathbf{5.98}$ Mbps and $\overline{\boldsymbol{R_2^{MS-SWG}}} = \mathbf{3.79}$ Mbps versus $\overline{\boldsymbol{R_1^{SWG}}} = \mathbf{4.21}$ Mbps and $\overline{\boldsymbol{R_2^{SWG}}} = \mathbf{3.21}$ Mbps. Furthermore, with the use of MS-SWG, the total number of stallings is zero and the MOS never drops, compared to SWG which suffers stallings and its MOS starts dropping after approximately 490 seconds have passed. We can also see that in the case of CACHE HITs, the instantaneous data rate R_2 never drops to zero with MS-SWG, which proves that the channel capacity of the proxy-to-main link is utilized better, resulting in a much better network resources utilization. This behavior was expected from the behavior analysis and now it has been verified experimentally. So, MS-

SWG seems like a really suitable video segment request and streaming algorithm which successfully answers the first two out of the three questions stated in the problem statement of the section 2.1.

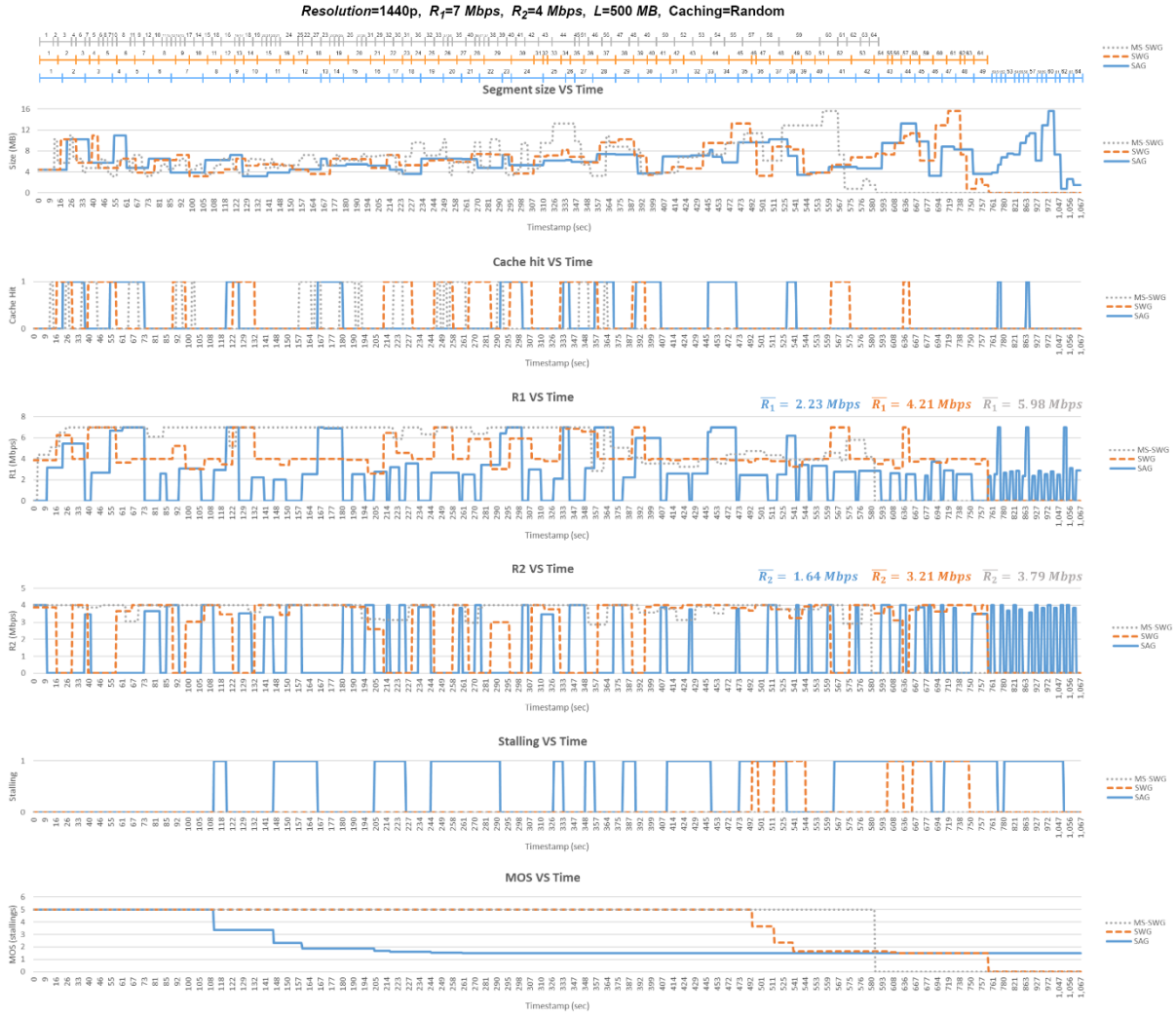

Figure 18: MS-SWG vs. SWG for 1440p, R_1=7 Mbps, R_2=4 Mbps, L=500 MB and Random caching

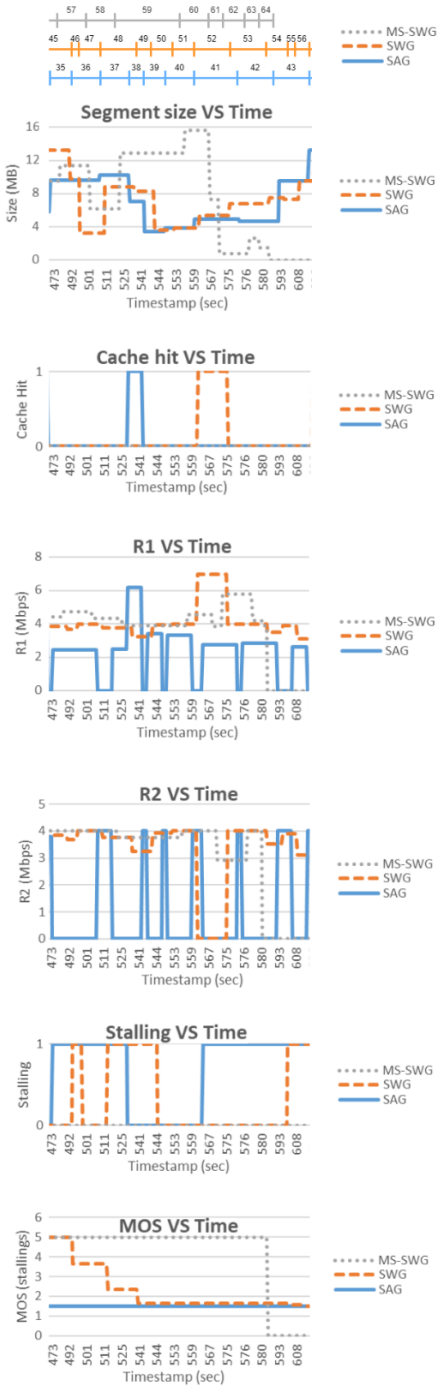

Figure 19: MS-SWG vs. SWG for 1440p, R_1=7 Mbps, R_2=4 Mbps, L=500 MB and Random caching, between 470 and 650 seconds

3.3 Epoch-based caching

3.3.1 MS-SWG and cache state

One very important detail regarding MS-SWG that was intentionally not mentioned in section 3.1, is that during a video streaming procedure MS-SWG achieves a better performance than the simple SWG only if the cache state of the cache proxy contains cached segments that are part of the specific video file and video resolution. Otherwise, (almost) all segments will be requested from main and served to the client through proxy in a serialized one-after-another order, which eliminates any advantages MS-SWG has to offer over SWG. So, for the optimal solution not only we need to have cached segments

in the cache proxy, but we also need a specific combination of cached and missing segments to achieve the best possible outcome with MS-SWG, minimizing or even eliminating any video playback latencies and re-buffering delays.

So, even though MS-SWG is an optimal segment video segment request and streaming algorithm, any type of random caching combined with MS-SWG is neither a suitable nor a sustainable caching technique. What we now need is to build a caching algorithm which can take advantage of the behavior of SWG and perform some "smart" caching selection so that the cached segments selected can help with achieving the highest utilization of both R_1 and R_2 while not exceeding the available buffer size L.

3.3.2 Proposed caching algorithm

The proposed solution to the caching problem is a novel coded caching algorithm, in which the video streaming process is divided into consecutive epochs, with each epoch enabling the User Equipment (UE) video player (client) to batch multiple requests for consecutive video segments of a target video bitrate and better utilize reserved storage/network resources. We call this algorithm **Epoch-based Caching (EBC)**. To achieve this, the cache proxy, using a recursive algorithm with dynamic programming, should:

i. infer on the maximum video bitrate it can support for the given parameter values and video file
ii. identify how segments should be allocated into streaming epochs, and
iii. decide which particular segments should be proactively cached for each streaming epoch to mitigate video stalls at the UE side.

Since edge network caching makes sense only when the channel capacity of the proxy-to-main link is lower than the channel capacity of the proxy-to-client link ($R_1 > R_2$), we readily conclude that full utilization of the channel capacity of the proxy-to-main link dictates caching of the early video segments per epoch, to prolong the time available for fetching non-cached segments through the low-end proxy-to-main link while delivering both cached and non-cached segments through the high-end proxy-to-client link.

Identifying the number of epochs as well as the number/IDs of segments that should be cached per epoch is a challenging optimization problem that has not been addressed before.

We consider that the cache proxy employs epoch-based content caching before the streaming service begins and suggests the most appropriate video bitrate to the end user upon service initiation. Given the data rates $\boldsymbol{R_1}$ and $\boldsymbol{R_2}$, the buffer size $\boldsymbol{L}$, and the video playout constraints (e.g., complete mitigation of video stalls), epoch-based network caching can have more than one feasible solutions. Accordingly, identifying the sequence of cached segments (a.k.a. caching code) that minimizes the cache usage at the cache proxy shall allow for a better utilization of the available cache, e.g., serving more users, or caching more files. Accordingly, we start our formulation with the problem of epoch-based content caching for a target video resolution $\boldsymbol{r} \in \boldsymbol{R}$ and then extend it to derive the maximum feasible video resolution $\boldsymbol{r}^*$ that the cache proxy can support.

The cache proxy should identify the most appropriate number of epochs E and infer on the particular sequence of consecutive video segments $[e_k, d_k]$ that it should cache per epoch k, where $1 \leq \mathrm{k} \leq \mathrm{E},\ e_1 = 1,\ e_k \leq d_k,\ 1 \leq e_k, d_k \leq M_f$. The caching code $[\{e_1, d_1, e_1'\}, \dots, \{e_E, d_E, e_E'\}]$, where $e_k' = e_{k+1} - 1$ is the last segment of epoch k and $e_{E+1} = M + 1$, should be carefully selected to respect the constraints $\boldsymbol{R_1}$ and $\boldsymbol{R_2}$, the buffer size $\boldsymbol{L}$**,** while completely mitigating video stalls.

Each epoch should include at least one cached and one non-cached segment, i.e. $d_k \leq e'_k$, , because if not, the respective epoch can be merged with the subsequent one. Thus, segments e_k to d_k are cached at the proxy while segments $d_{k+1} + 1$ to e'_k are fetched from main through the proxy-to-main and proxy-to-client links. We now formulate the problem of epoch-based network caching given a target video resolution $\boldsymbol{r} \in \boldsymbol{R}$, file $\boldsymbol{f} \in \mathbb{F}$ and client $\boldsymbol{u}$. The optimization problem is to identify the caching code $\boldsymbol{c}^*_{\boldsymbol{f},\boldsymbol{r}} = [\{e_1, d_1, e'_1\}, \dots, \{e_E, d_E, e'_E\}]$ with the minimum cache size requirements $\boldsymbol{C}^*_{\boldsymbol{f},\boldsymbol{r}}$ that accounts for the given parameter values and meets the video playout constraints.

This algorithm makes sure that $R_1 \geq R_{f,r,m,k} \geq R_2$, where $R_R = \frac{1}{(m-k+1)\,T_f} \sum_{i=k}^{m} S_{f,r,i}$ (bps) is the minimum channel capacity required so that segments k to m of resolution r are played back seamlessly on the client's UE. If $R_{f,r,m,k} > R_1$ stallings are sure to occur since the channel capacity of the proxy-to-client link does not suffice, while if $R_{f,r,m,k} < R_2$ it would make no sense to cache anything since the channel capacity of the proxy-to-main link suffices so that all segments could be served from main to the client through the proxy. The rest of EBC's details will not be presented in this thesis, since this system model with MS-SWG combined with EBC are part of a paper that has been submitted for review to be published at the IEEE/ACM Transactions on Networking [34] journal.

The EBC algorithm has the potential to give a much better video streaming performance while keeping the required cache storage size to a minimum. However, it has the following set of drawbacks:

- it requires an a-priopri knowledge of R_1, R_2, L, total video duration, availabale video resolutions and segment sizes in order to make its caching choice, which is not always the case in real networking and video streaming environments
- the video segments in its caching choice are segments of a specific video resolution, multiple resolutions are not supported
- it only achieves the optimal streaming performance if both the cached and the missing segments are simultaneously requested and streamed

3.4 MS-SWG with EBC vs MS-SWG with Random caching

The third set of experiments was a comparison between MS-SWG combined with EBC and MS-SWG combined with Random caching, for a scenario with the same network parameters with the previous two sets of experiments, $\boldsymbol{R_1 = 7}$ **Mbps**, $\boldsymbol{R_2 = 4}$ **Mbps** but with a different buffer size, $\boldsymbol{L = 250}$ **MB**, **Random caching** and with a **fixed video resolution of 1440p**, which has a total file size of 418.80 MB. Fig. 20 contains the development of the whole video streaming procedure for both Random and Coded caching, while Fig. 21 contains the cache hit/miss development in a time window between 0 and 350 seconds for a clearer view. At the right side of the charts in Fig. 20 it seems as if the combination of MS-SWG with Random caching suddenly became faster, but this is an optical illusion since the explanation for this behavior is the fact that the x-axis with the time is compressed, since there is nothing to compare any more.

Table 3 contains some information regarding the output of the EBC algorithm's output in this experiment. We can see that with a total cache size of only 250 MB, EBC chooses segments of the 2nd highest resolution (1440p) which requires a data rate $D \geq 5.28$ Mbps in order to avoid stallings during streaming, it only caches 29 out of the 64 available segments, which is less than half, and requires only 190.3 MB of cache storage out of the 418.8 MB which is the total video size, saving about 60 MB of cache storage.

Table 3: Coded caching algorithm output for R_1=7 Mbps, R_2=4 Mbps, L=250 MB

R_1	R_2	L	Video Resolution	Required Data Rate D	Cache Choice	Cached Segments Size	Video Size
7 Mbps	4 Mbps	250 MB	1440p	5.28 Mbps	1-2, 4, 6-10, 18-38	190.3 MB	418.8 MB

From both Fig. 20 and Fig. 21, we can see that MS-SWG combined with EBC completes the whole video streaming procedure about 260 seconds earlier than with Random caching, requiring about 490 seconds to completely stream a video of total duration equal to 634.6 seconds, reducing the total streaming duration and with it the total network usage time, while also achieving higher mean data rates $\overline{\boldsymbol{R_1^{EBC}}} = \mathbf{6.76}$ Mbps and $\overline{\boldsymbol{R_2^{EBC}}} = \mathbf{3.57}$ Mbps versus $\overline{\boldsymbol{R_1^{Random}}} = \mathbf{4.65}$ Mbps and $\overline{\boldsymbol{R_2^{Random}}} = \mathbf{3.75}$ Mbps. Furthermore, with the use EBC, the total number of stallings is zero and the MOS never drops, compared to Random caching which suffers stallings and its MOS starts dropping after approximately 310 seconds have passed. We can also see that the instantaneous data rate R_1 of EBC is always higher than the one of Random caching, which proves that channel capacity of the proxy-to-client link is utilized better, resulting in a much better network resources utilization. So, EBC seems like a really suitable video segment caching algorithm which successfully answers the third and last out of the three questions stated in the problem statement of the section 2.1.

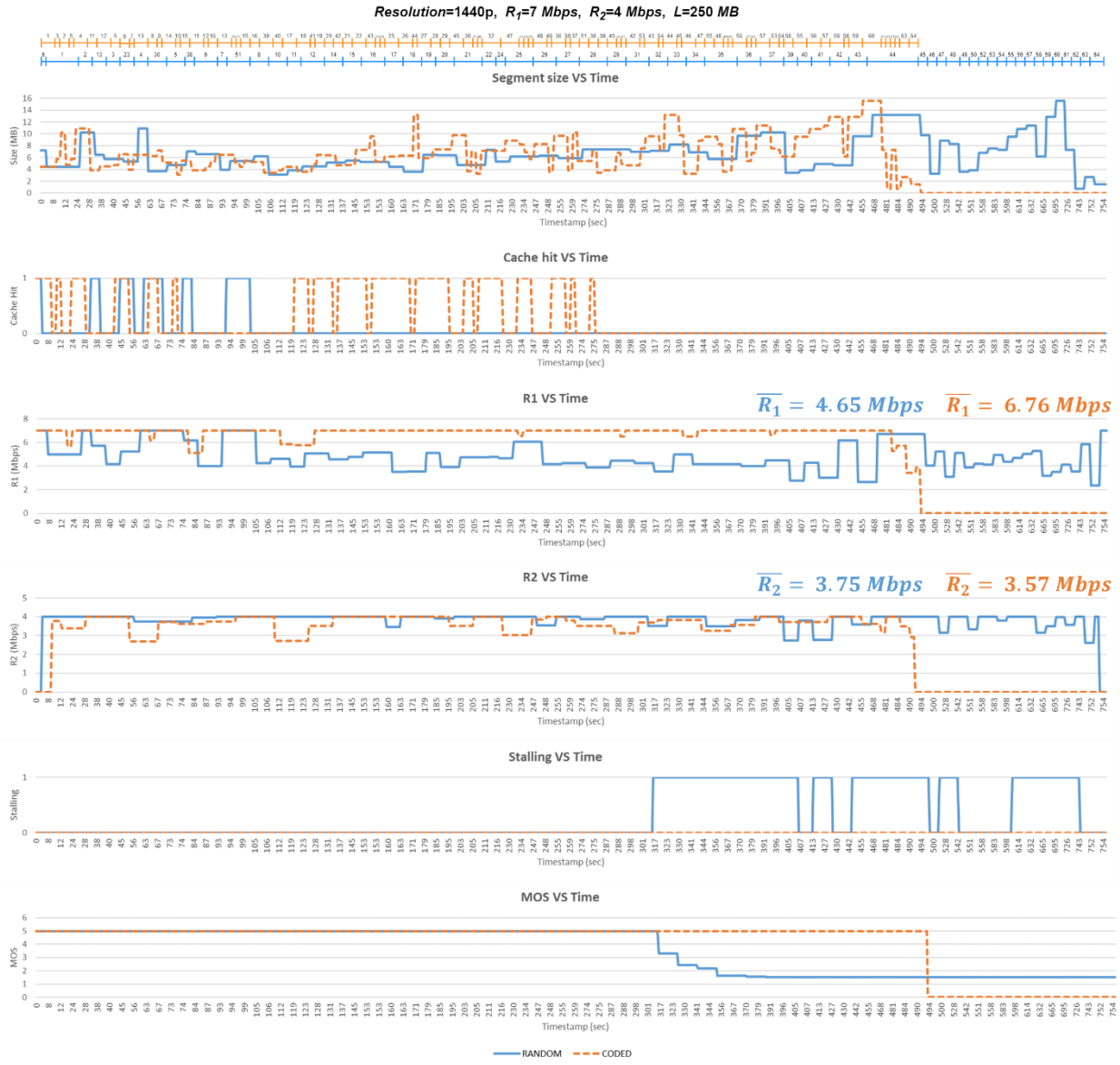

Figure 20: Performance comparison between random and EBC (coded) caching algorithms with the use of MS-SWG, for 1440p, R_1=7 Mbps, R_2=4 Mbps, L=250 MB

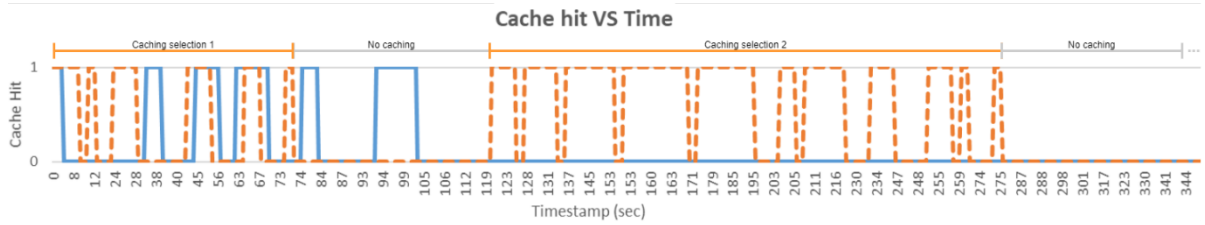

Figure 21: EBC (coded) vs. Random cache hits / misses with the use of MS-SWG, for 1440p, R_1=7 Mbps, R_2=4 Mbps, L=250 MB

4. Performance evaluation

For the performance evaluation, we used the system model presented in Chapter 2, with the addition of the proposed content caching and video segment request and streaming algorithms presented in Chapter 3. We conducted the following set of 5 distinct combinations of caching algorithm, video segment request and streaming algorithm, and resolution choice:

- EBC + MS-SWG + predictive resolution
- Random caching + MS-SWG + fixed resolution
- Random caching + SWG + fixed resolution
- Random caching + SWG + predictive resolution
- Random caching + SAG + predictive resolution

For each eperimentation combination we used the values of Table 4 for R_1, R_2 and L.

Table 4: Experimentation parameters

R_1 (Mbps)	R_2 (Mbps)	L (MB)
18	8	400
15.8	6	300
12	4	200
8.3	2	100
7	1	50
5.7		0
3		
2		
1.4		

We evaluated the performance of these combination using the following evaluation metrics:

- **Average Video Bitrate**: mean value of the bitrate required for the video streaming, based on both the segment sizes which may vary in case the resolution changes adaptively and their respective playback duration, and is computed as $AVG_{bitrate} = \frac{1}{N \cdot T_f} \sum_{i=1}^{N} S_{f,r,i}$, where $S_{f,r,i}$ is the file size of the segment i with resolution r of the original video f and T_f is the segment playback duration.
- **Average Video Resolution**: mean value of the streamed video resolution, computed as $AVG_{resolution} = \frac{1}{N} \sum_{i=1}^{N} r_i$. This value is equal to the video resolution when a fixed video resolution is used.
- **Sum of Resolution Switches**: total number of resolution changes, computed as $SS = \sum_{i=2}^{N} s_i$ where $s_i = \begin{cases} 0, & if\ r_i = r_{i-1} \\ 1, & otherwise \end{cases}$
- **Average Resolution Altitude**: mean value of altitude which is based on the difference between two different but consecutive video segment resolutions, and is computed as $AVG_{altitude} = \frac{1}{N} \sum_{i=2}^{N} a_i$ where the altitude between a streamed

video segment i and the previously streamed video segment $i-1$ is computed as $a_i = |v_i - v_{i-1}|$, where $v_i = \begin{cases} 1, \ if \ r_i = 360p \\ 2, \ if \ r_i = 480p \\ 3, \ if \ r_i = 720p \\ 4, \ if \ r_i = 1080p \\ 5, \ if \ r_i = 1440p \\ 6, \ if \ r_i = 2160p \end{cases}$.

- **Mean Opinion Score based on video resolution**: mean value of the opinion score that occurs based on the resolution of the streamed video and is computed as $MOS_{RES} = \frac{1}{N}\sum_{i=1}^{N} o_i$ where $o_i = \begin{cases} 2.07744, \ if \ r_i = 360p \\ 3.02246, \ if \ r_i = 480p \\ 3.97185, \ if \ r_i = 720p \\ 4.47112, \ if \ r_i = 1080p \\ 4.52586, \ if \ r_i = 1440p \\ 4.58036, \ if \ r_i = 2160p \end{cases}$ is the opinion on the resolution of the segment i. Obviously, $2.07 < MOS_{RES} < 4.6$ where 4.6 is the best value and 2.07 the worst.
- **Initial Playback Delay**: the time elapsed from the beginning of the video streaming procedure initiation, until the actual video playback begins. This delay can vary between different video players and even within the same video player due to the network channel conditions. For our experimentation, we consider the initial playback delay to be the time required to sucessfully stream the first segment.
- **Stallings**: total number of freezes occurred during the video playback because of segment delivery delays.
- **Total Stalling Time**: total time elapsed for all stallings that occurred during the whole video playback.
- **Mean Stalling Time**: mean value of the total stalling time, with respect to the number of stallings occurred.
- **Mean Opinion Score based on stallings**: based on both on the number of stallings ocured and the total stalling time, computed as $MOS_{STALL} = 3.5 \cdot e^{-(0.15\,T+0.19)\cdot S} + 1.5$ where T is the total stalling time and S is the total number of stallings occurred. Obviously, $1.5 \leq MOS_{STALL} \leq 5$ where 5 is the best value and 1.5 the worst.
- **Mean Network Throughput R_1**: Average channel capacity of the proxy-to-client link during the whole video streaming duration.
- **Mean Network Throughput R_2**: Average channel capacity of the proxy-to-main link during the whole video streaming duration.
- **Total Network Usage Time**: total time the network was active during the whole video streaming duration.
- **Number of cached bytes delivered**: total number of bytes of the cached segments that were delivered to the client
- **Number of non-cached bytes delivered**: total number of bytes of the non-cached (missing) segments that were delivered to the client
- **Cache Miss Ratio**
- **Cache Hit Ratio**
- **Backhaul traffic ratio**
- **Total Video Playback Duration**: total time elapsed from the video playback initiation until its completion

For the experimentation we used the video described in Appendix I. Fig. 22 shows the video bitrate development with respect to the increase in the video resolution, for the

available video resolutions and their respective file sizes. We can see that to seamlessly stream the 2160p resolution to to the client, whose total file size is equal to 728.51 MB, a data rate of at least 9.18 Mbps is required.

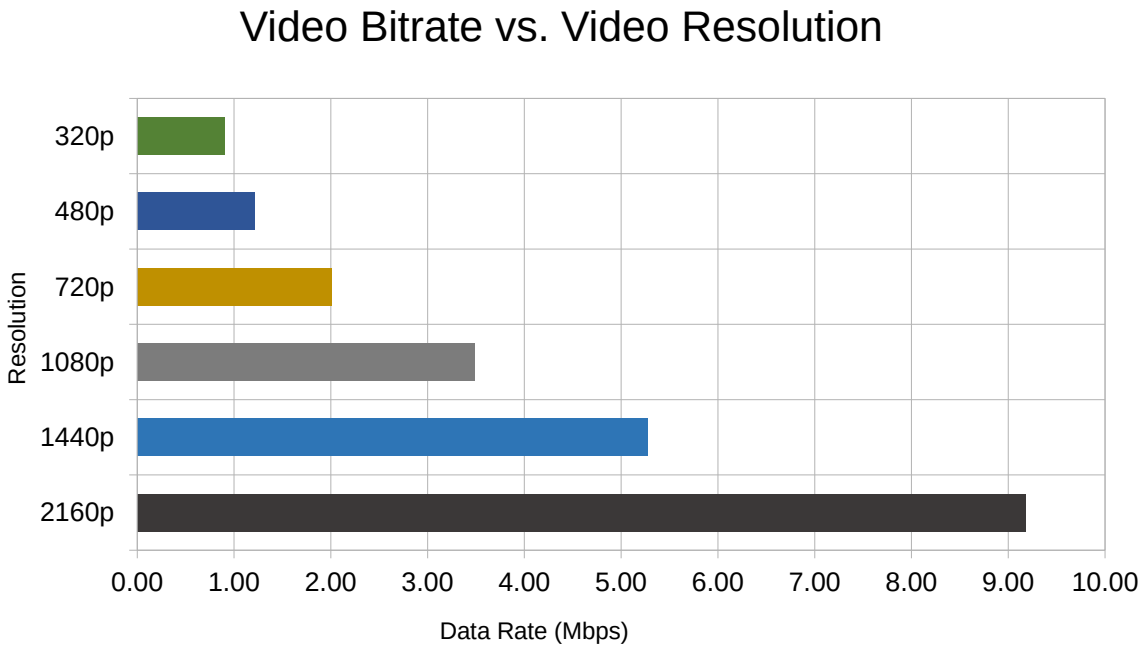

Figure 22: Video bitrate vs. Video resolution

At this point we are going to present some comparison charts between the aforementioned 5 combinations, to comment on the behavior of the system model and determine whether the proposed solution, i.e. the combination of MS-SWG and EBC, yields the expected performance results. These charts are useful in getting a greater view of each algorithm's performance in general and compared to the other ones, using different values for R_1, R_2 and L.

Obviously, in order for the EBC to make sense and the whole system model to have meaningful performance improvements, we will only present charts of the experiments in which $R_1 > R_2$ and $L > 0$. Since the highest the video resolution achieved, the highest the video bitrate also achieved, we are only going to present experiments at which all 1080p, 1440p, 2160p resolutions are achieved. Also we need to make sure that R_1 is much higher than R_2 so that our displayed example are as close to reality as possible.

So, for displaying of the system model's behavior against the development of L we choose the channel capacity of the proxy-to-client link $R_1 = 18$ Mbps and the channel capacity of the proxy-to-client $R_2 = 6$ Mbps so that $R_1 \cong 3 \cdot R_2$ while for its behavior against the development of R_1 we choose $R_2 = 6$ Mbps and $L = 300$ MB so that $\frac{1}{3}$ of the total size of the highest available video resolution file fits in the cache and the value of R_2 matches.

For every experiment with a fixed video resolution, this video resolution was based on the output of EBC, which performs the calculations described in Chapter 3 and outputs both some calculated optimal video resolution to stream alongside a cache code which consists of the segments to cache. Both the optimal resolution and the segment choice EBC calculated is based on the values of R_1, R_2 and L.

Initially we are going to present some charts to study the bitrate of the EBC's optimal resolution choice.

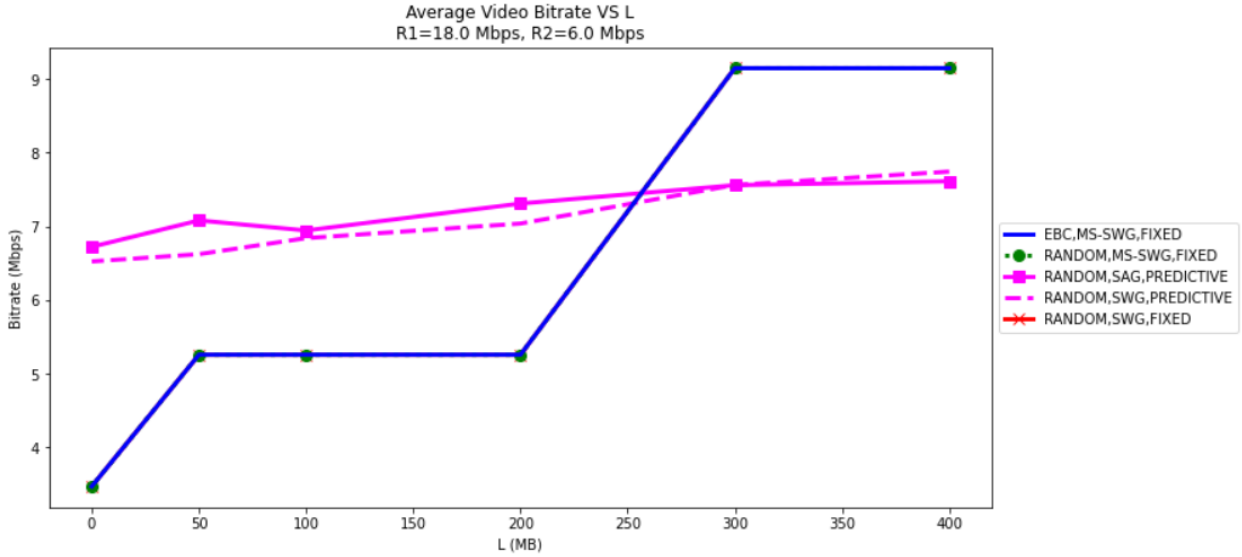

Figure 23: Average Video Bitrate vs. Buffer Size L

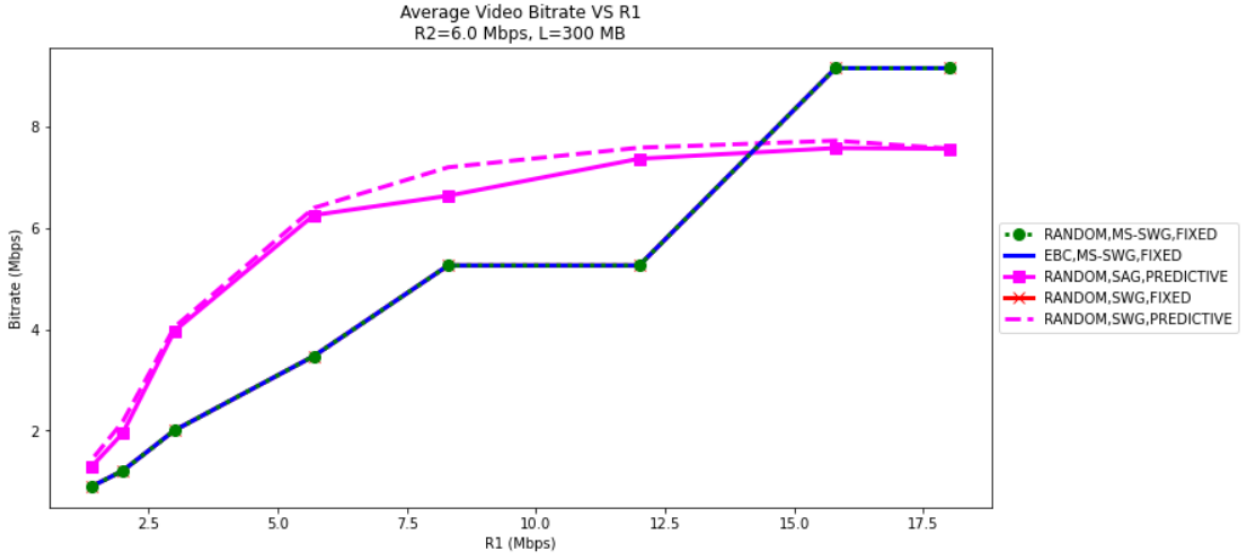

Figure 24: Average Video Bitrate vs. Channel Capacity R_1

From Fig. 23 and Fig. 24 we can see that the bitrate of EBC's optimal resolution choice is lower than the average adaptive resolution of the predictive algorithm for low values of R_1 and L. This behavior can be explained because the predictive algorithm attempts to raise the streaming resolution to the highest whenever possible. However, for high values of R_1 and L, the EBC algorithm selects a highest fixed resolution that what the predictive algorithm achieves. Now, we are going to see the evolution of MOS based on the average video resolutions of the charts above.

Now, we are going to study the effect of EBC's optimal resolution choice on the MOS.

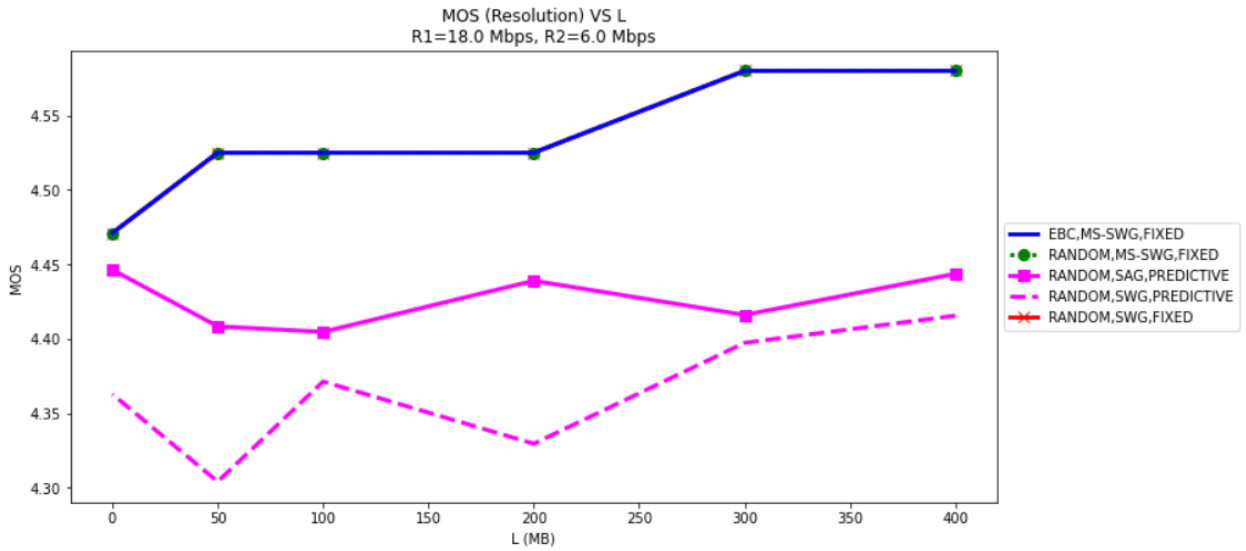

Figure 25: MOS (Resolution) vs. Buffer Size L

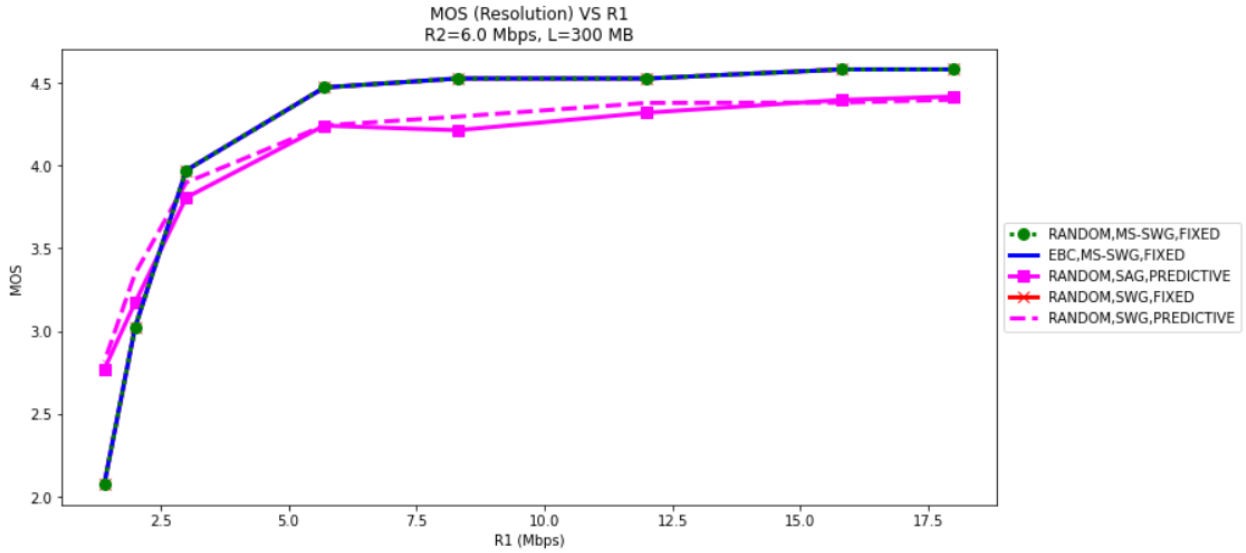

Figure 26: MOS (Resolution) vs. Channel Capacity R_1

From Fig. 25 and Fig. 26 we can see that the value of MOS based on video resolution is almost always higher with EBC's optimally selected and fixed resolution compared to the adaptive video resolution of the predictive algorithm. This happens because when adaptive video resolution drops to adapt to existing network conditions the MOS drops with it, while with the fixed video resolution MOS is constant. This means that the end user enjoys a high QoE.

Now, we will move on to the playback behavior, starting with the stallings.

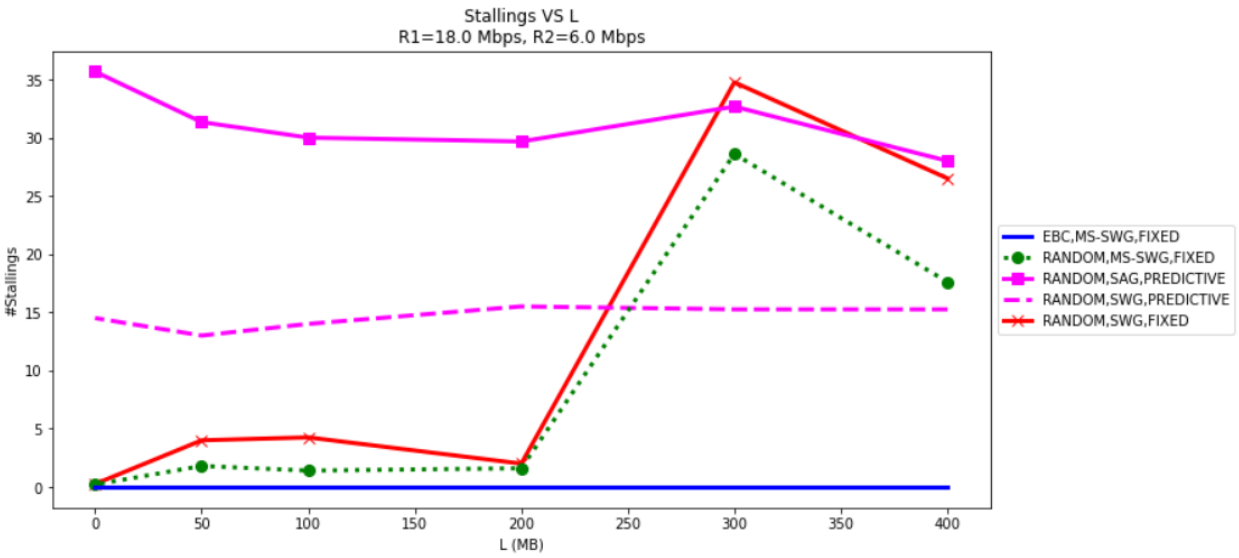

Figure 27: Number of stallings vs. Buffer Size L

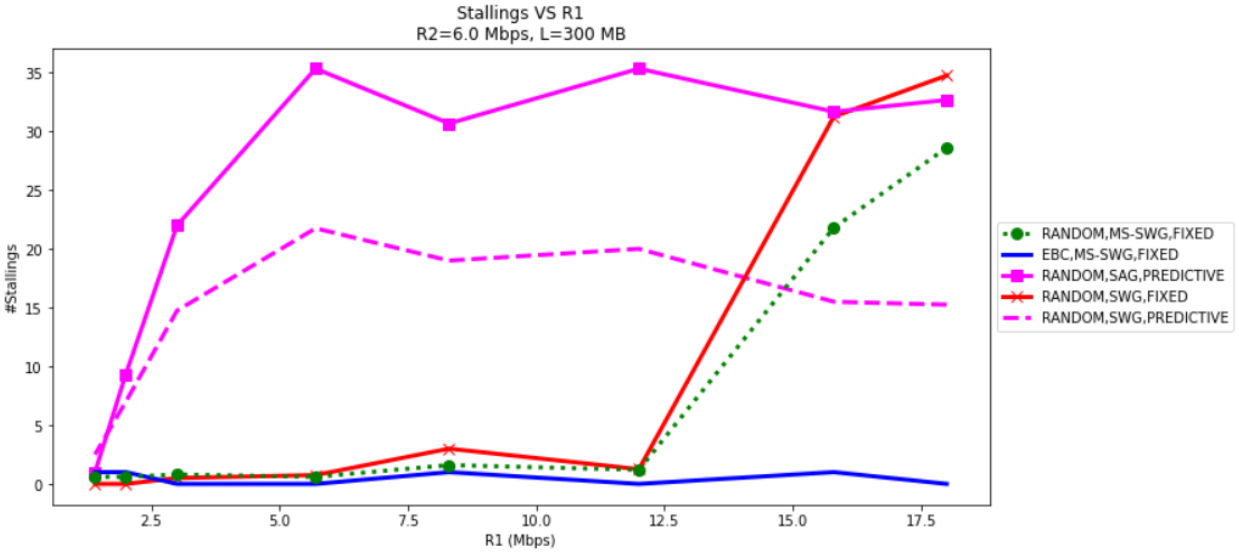

Figure 28: Number of stallings vs. Channel Capacity R_1

From Fig. 27 and Fig. 28, we can see that the number of stallings is always lower with EBC's optimally selected and fixed resolution compared to the adaptive video resolution of the predictive algorithm. Furthermore, with EBC's optimal caching selection, the number of stallings is the best among all combinations. The reason behind the small non-zero number of stallings is the fluctuation in the channel because of the randomly changing network conditions. So, EBC with MS-SWG has the best behavior among all combinations regarding stallings.

Now, we are going to study the effect of EBC's optimal caching selection on the MOS.

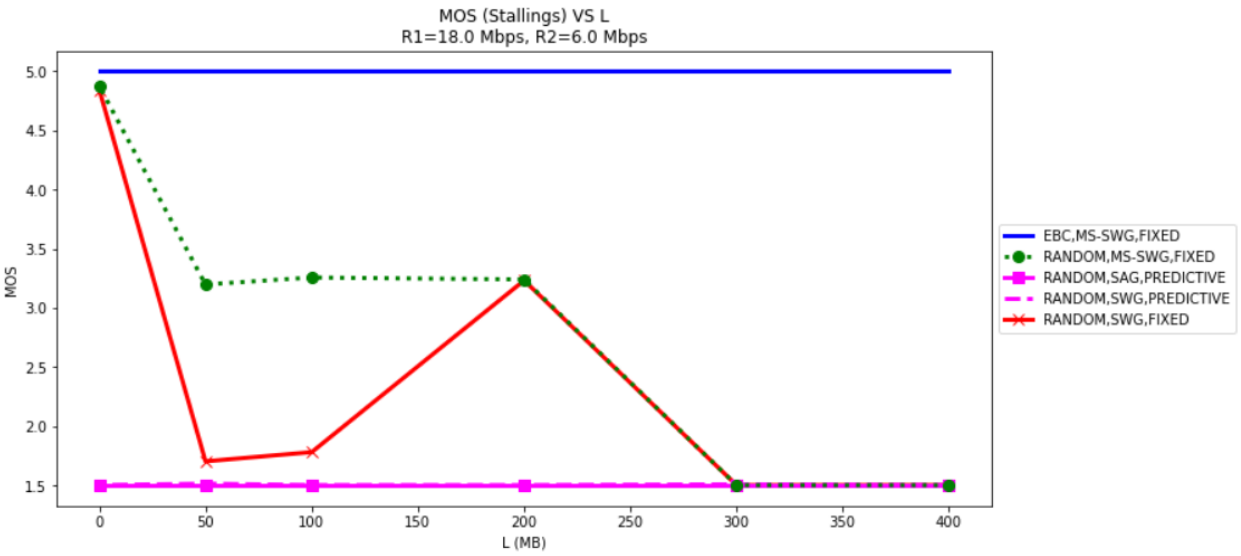

Figure 29: MOS (Stallings) vs. Buffer Size L

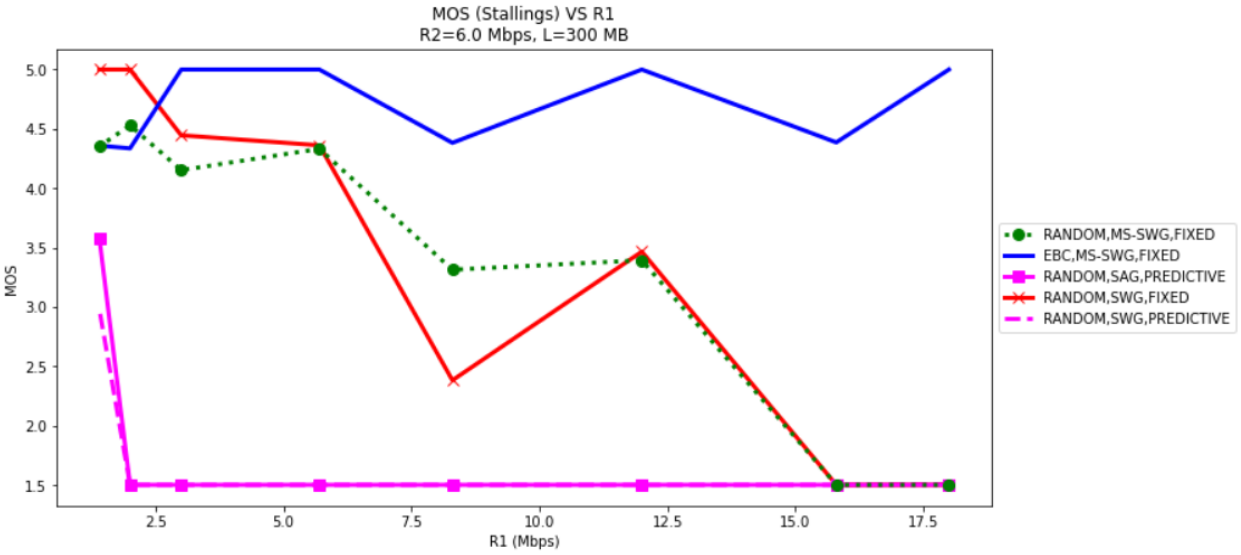

Figure 30: MOS (Stallings) vs. Channel Capacity R_1

From Fig. 29 and Fig. 30 we can clearly see that independent of the size of the buffer size L, MOS maintains the highest value possible with the combination of EBC with MS-SWG. However, with respect to the value of R_1, we can see that even though MS-SWG with EBC does not maintain the highest value possible for MOS, because of random changes in the channel condition, its value is still higher than all other combinations and close to a "perfect 5".

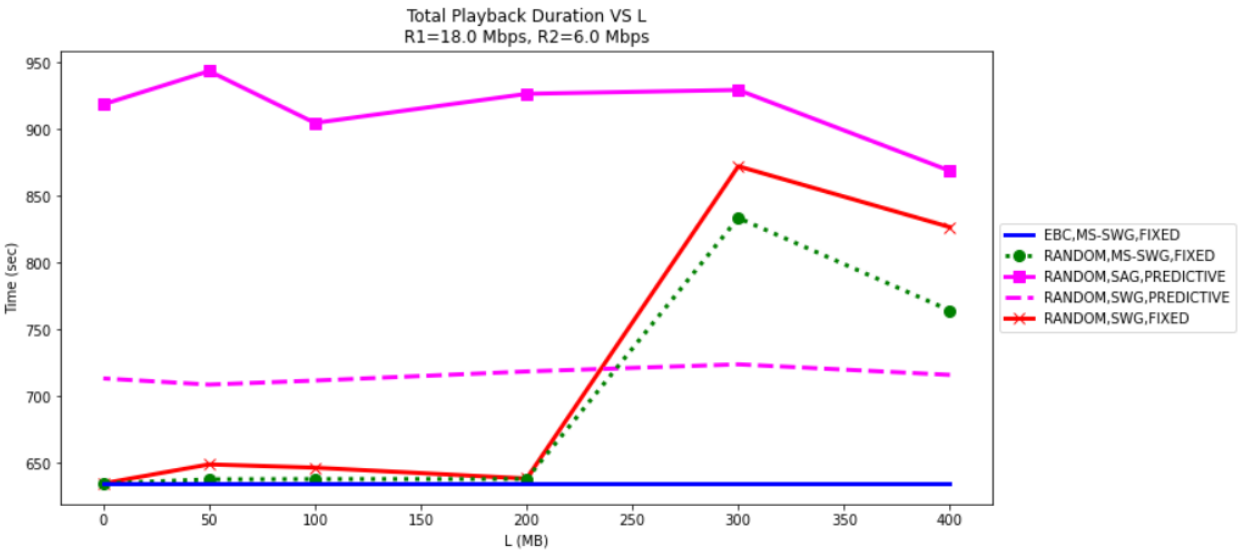

Figure 31: Total Video Playback Duration vs. Buffer Size L

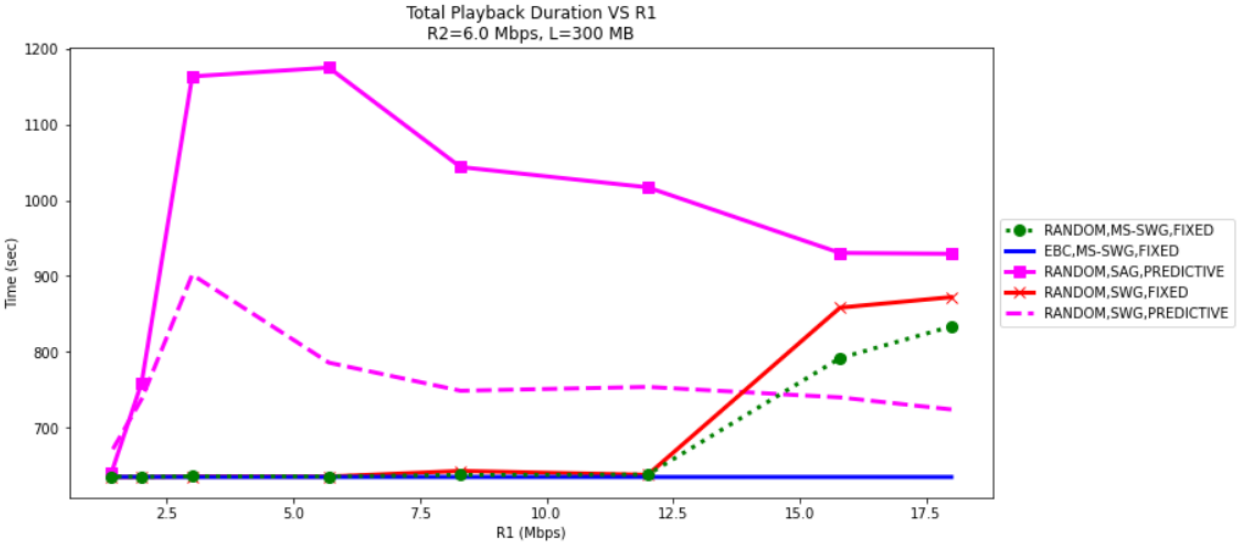

Figure 32: Total Video Playback Duration vs. Channel Capacity R_1

From Fig. 31 and Fig. 32 we can see that EBC with MS-SWG achieves the lowest total video playback duration, independently of the values for R_1, R_2 and L, because of the optimally selected video resolution and segments to cache. This means a guranteed video playback duration can be offered.

Now, we are going to study the effect of EBC's optimal caching selection on the total stalling duration.

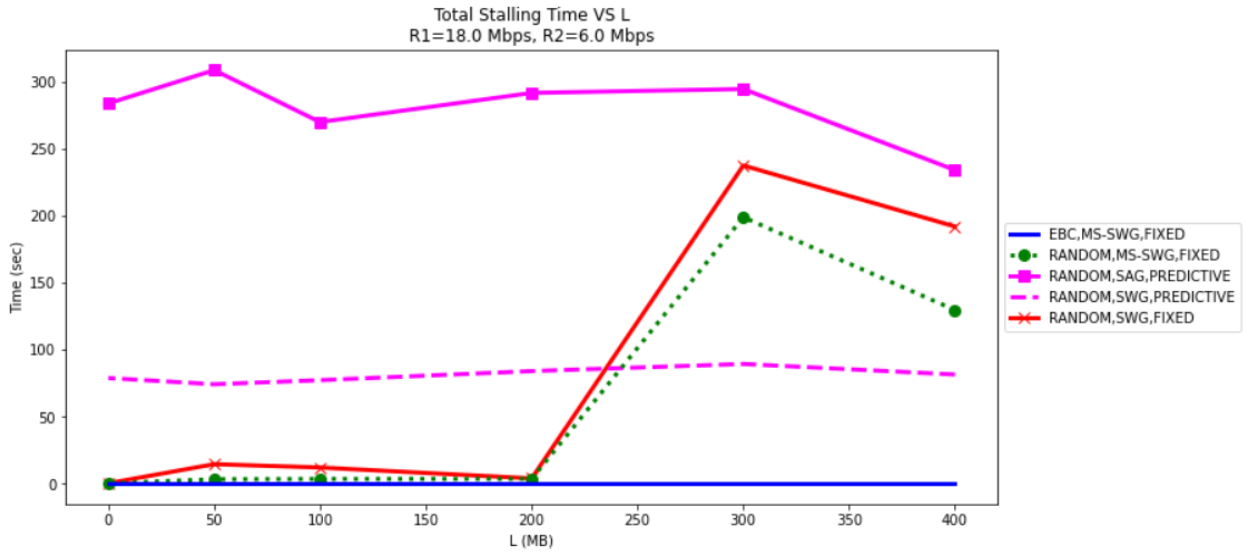

Figure 33: Total Stalling Time vs. Buffer Size L

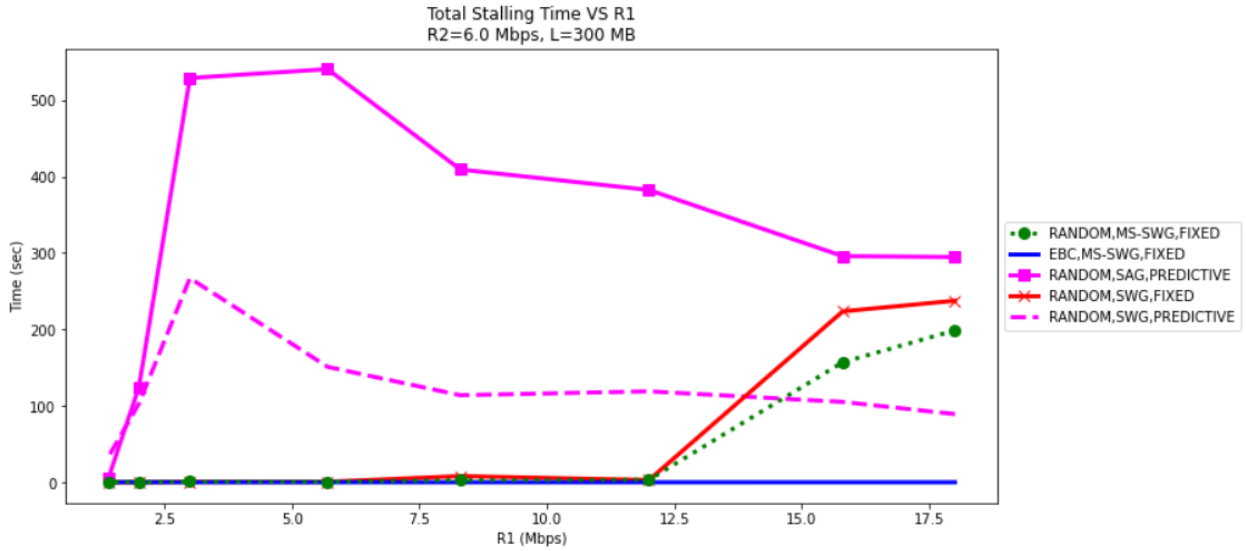

Figure 34: MOS (Stallings) vs. Channel Capacity R_1

From Fig. 33 and Fig. 34 we can clearly see that EBC with MS-SWG results in the lowest possible Total Stalling Time among all combinations. Now we have to study the behavior of the initial playback delay the end user experiences.

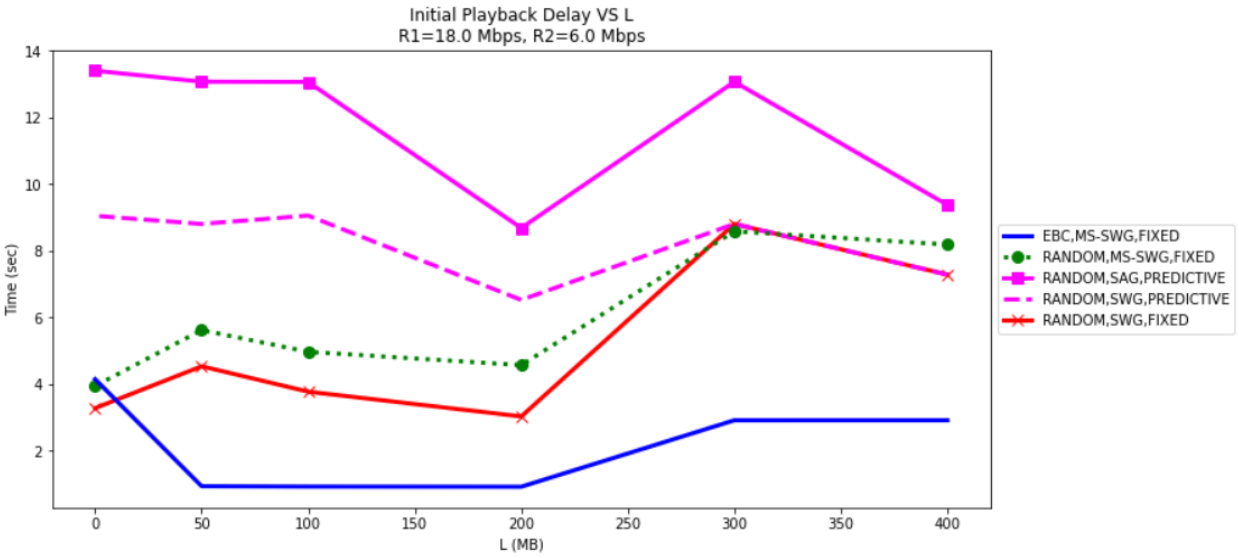

Figure 35: Initial Playback Delay vs. Buffer Size L

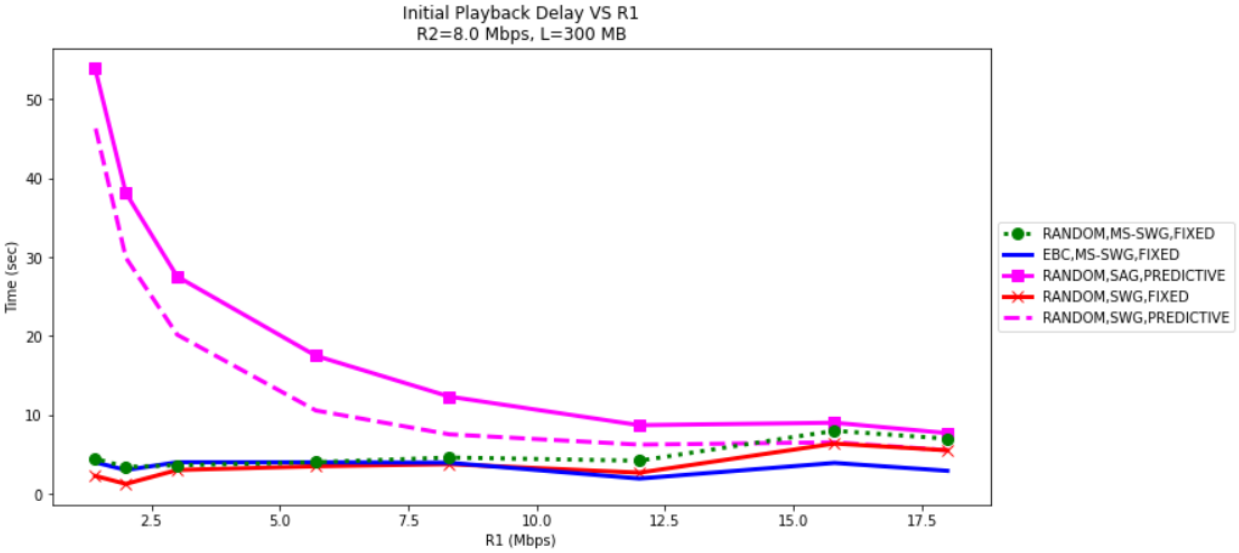

Figure 36: Initial Playback Delay vs. Channel Capacity R_1

From Fig. 35 and Fig. 36 we can see that the combination of EBC with MS-SWG not only experiences the lower initial playback delay but this delay is also as low as 5 seconds or even lower. Taking into account MOS (Stallings) and Total Stalling Time, we conclude that the end user enjoys a high QoE with MS-SWG and EBC.

Next, we are going to study the QoS of our model with respect to whether the set goals were met.

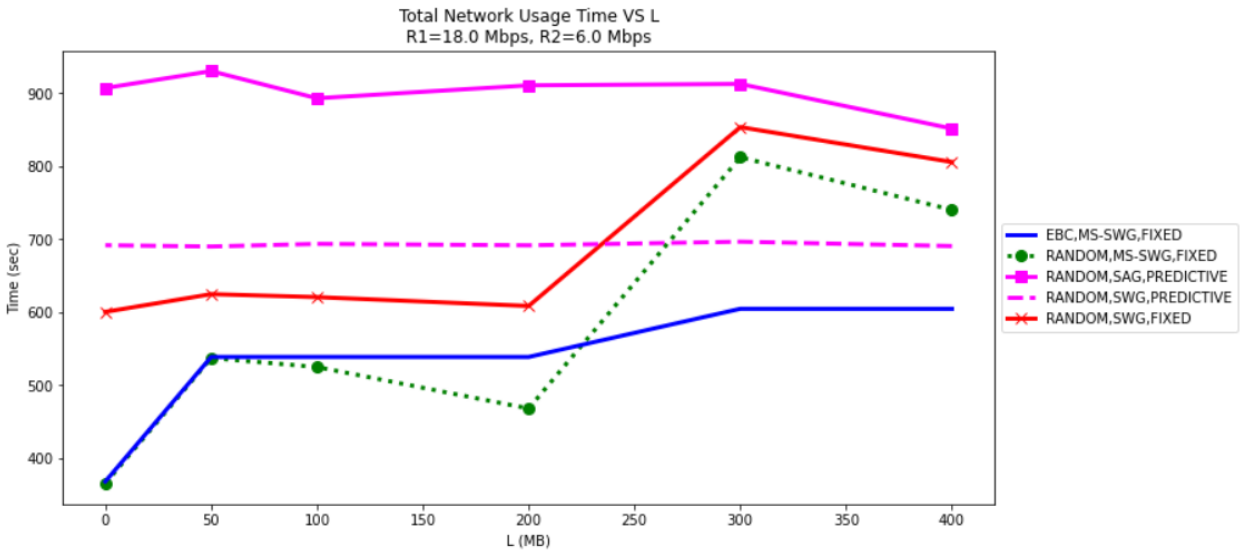

Figure 37: Total Network Usage Time vs. Buffer Size L

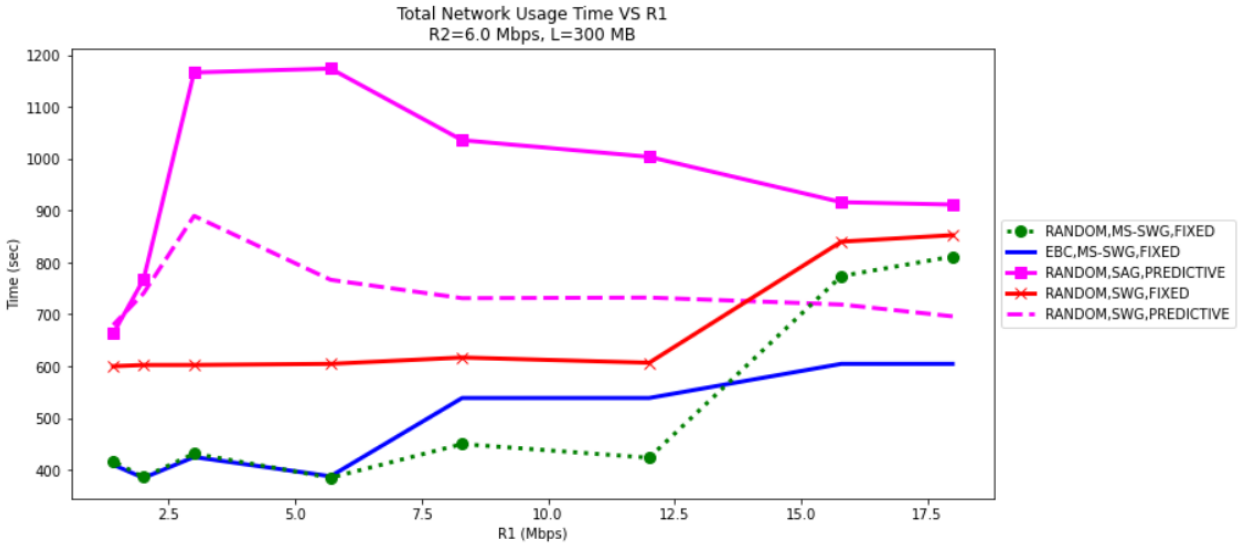

Figure 38: Total Network Usage Time vs. Channel Capacity R_1

From Fig. 37 and Fig. 38, combined with the achieved mean video bitrate development displayed in Fig. 33 and Fig. 24, we can deduce that for high values of L and R_1 where higher video bitrates are achieved, the total network usage time of EBC with MS-SWG achieves the lowest total network usage time. Moreover, EBC with MS-SWG always achieves a total network usage time that is lower than the video duration which is equal to 634.6 seconds, which means that the video streaming procedure always completes in time.

Next, we will move on with the network throughput.

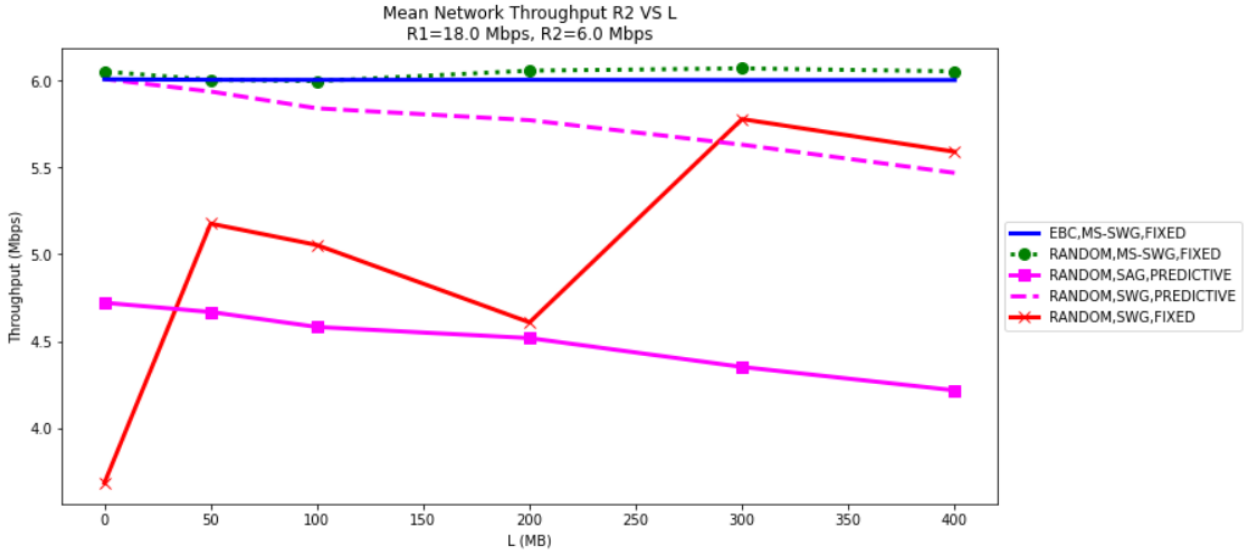

Figure 39: Mean Network Throughput $\mathbf{R_2}$ vs. Buffer Size $\mathbf{L}$

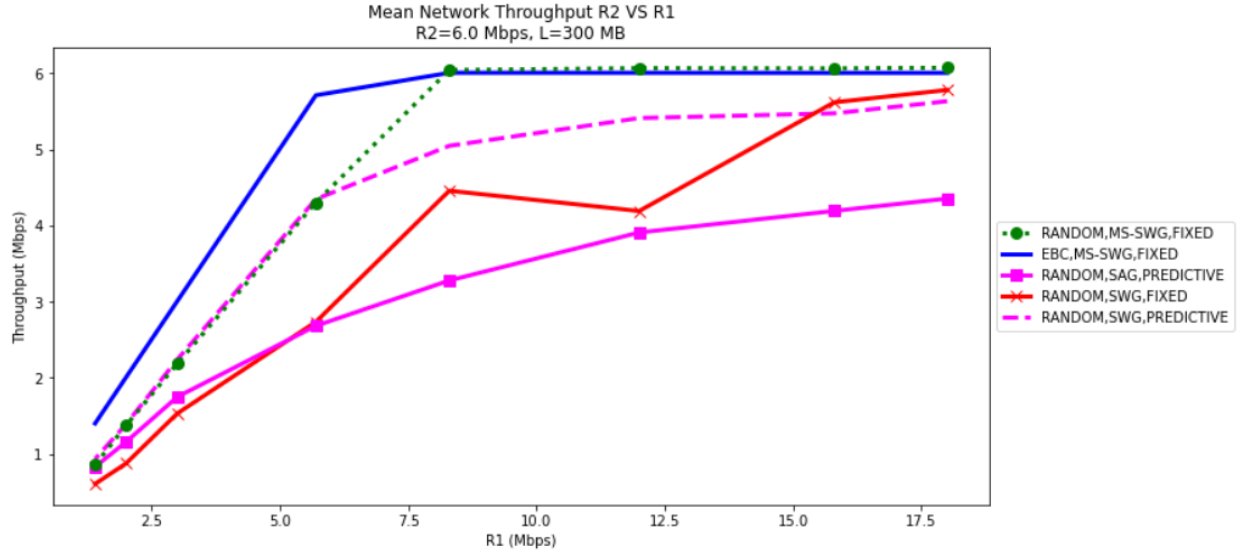

Figure 40: Mean Network Throughput R_2 vs. Channel Capacity R_1

From Fig. 39 and Fig. 40, combined with the achieved mean video bitrate development displayed in Fig. 23 and Fig. 24, we can deduce that EBC combined with MS-SWG achieves the highest mean network throughput for the channel capacity R_2 of the proxy-to-main link.

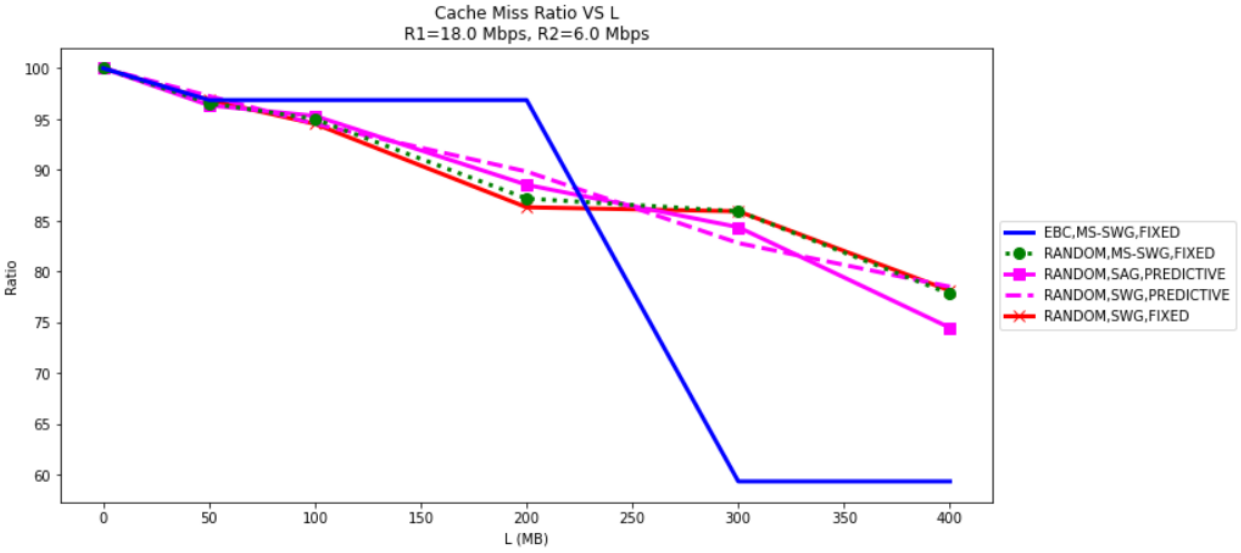

Figure 41: Cache Miss Ratio vs. Buffer Size L

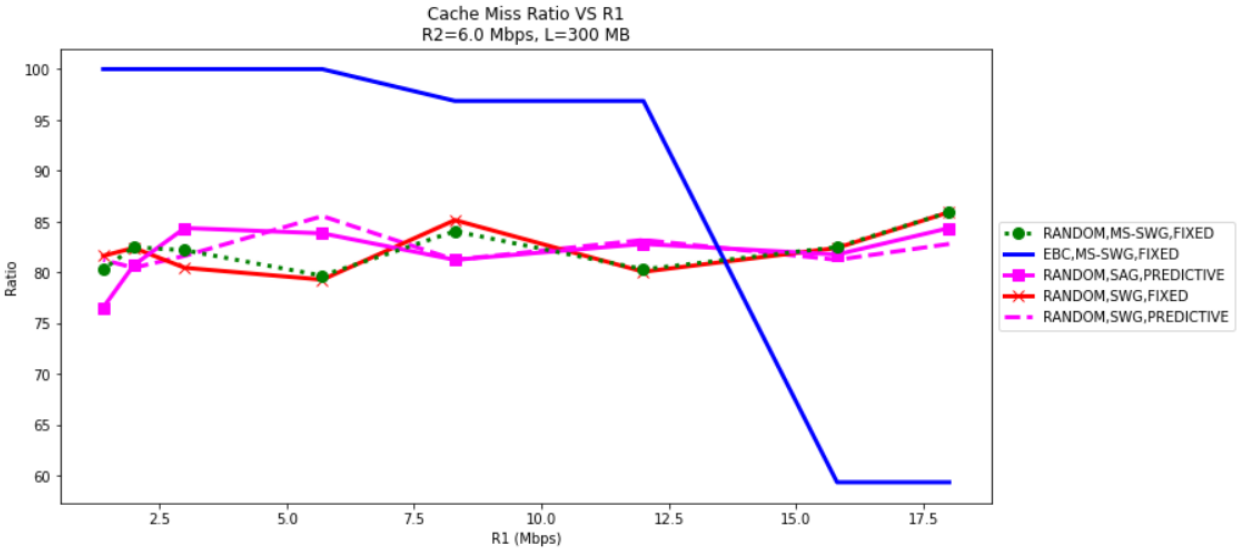

Figure 42: Cache Miss Ratio vs. Channel Capacity R_1

From Fig. 41 and Fig. 42, combined with the previous charts, we can see that EBC with MS-SWG achieves the highest performance among all combinations independently of the cache miss ratio. Furthermore, it manages to keep the cache miss ratio to a minimum for high video bitrates where the video file sizes increase because of the increase in the resolution, rendering the utilization of caching capabilities mandatory. The reason behind the low cache utilization for low values of R_1 and L is that the channel capacities of the proxy-to-client link and the proxy-to-main link are sufficient for streaming the selected video resolution of EBC while maintaining a low utilization of the caching capabilities.

Taking all these into account, we conclude that the combination of MS-SWG with EBC achieves the highest QoS for our model.

5. Conclusion and future improvements

After everything we studied in this thesis regarding DASH, video streaming, edge video caching and D2D content exchange, we can see that since both video demand and video resolutions are going to increase in the future, a series of smarter and more efficient techniques needs to be invented and deployed to meet such demands, taking advantage of the innovative technologies 5G can offer. Otherwise, both the QoS of the service offered and the QoE for the end users will remain sub-optimal. Innovative services developed in future networks must be user-centric first, i.e. their architects should first of all aim at offering a great experience to the end users, since these users are the ones that will benefit from the network improvements and that will pay for the service they get.

Our basic three node system model alongside the proposed MS-SWG segment request and streaming algorithm and the EBC caching algorithm are examples of smart techniques that can offer improvements to the QoS and the QoE, since we have already seen that they can offer significant improvements to the video streaming procedure. Moreover, the application of such smart and efficient techniques in a decentralized and distributed network can not only offer advancements in the offered video streaming procedure and service but also reduce the total network operation costs and lower the total network traffic congestion.

Furthermore, EBC was developed so that the values for the network parameters R_1, R_2 and L can be used to decide upon the optimal video resolution and segments to cache. However, a reverse technique can be applied so that using the video resolution, the segments to cache and two out of the three network parameters R_1, R_2 and L, we can deduce the optimal value for the remaining one. For example, we may be able to use the values for R_1 and R_2 to deduce the optimal value for the buffer size L, or use the values for R_2 and L to deduce the optimal value for the channel capacity R_1.

Finally, any network node that participates in a video content distribution procedure can be extended to record the events of this procedure for every procedure. This history can be used as input in Data Analysis tasks so that additional metrics and information is produced and stored. Then, Machine Learning techniques can be utilized so that nodes in the network use the recorded history in their database to make decisions on the QoS parameters for future content distribution sessions. For example, a client can use this knowledge to request specific guarantees for the streaming service, while a proxy can use this knowledge to deduce the optimal network and storage capabilities. These Machine Learning techniques can also be used for Cybersecurity purposes, either as an intrusion detection system by identifying patterns of suspicious behaviors in the network, to offer faster malicious node isolation, or as Anomaly Detectors for detecting anomaly patterns in the network.

Finally, an extension of the MS-SWG algorithm can be developed so that multiple clients are simultaneously served. This extension can also lead to a fully distributed peer-to-peer network in which every UE network device is considered as a network node which can act either as a cache proxy that distributes video content to another node or as a client which consumes video content. Furthermore, a decentralized payment system, e.g. a blockchain, can be used so that proxy nodes get paid for the content they provide, where client nodes pay for the content they receive.

ABBREVIATIONS – ACRONYMS

5G	Fifth Generation Network Systems
ABR	Adaptive Bit Rate
ACM	Association for Computing Machinery
AVC	Advanced Video Coding
AVG	Average
BS	Base Station
CDN	Content Delivery Network
CENC	Common Encryption
CPU	Central Processing Unit
CRF	Constant Rate Factor
CSS	Cascading Style Sheets
D2D	Device-to-Device
DASH	Dynamic Adaptive Streaming over HTTP
DDoS	Distributed Denial of Service
DRM	Digital Rights Management
EBC	Epoch-based Caching
EME	Encrypted Media Extensions
HAS	HTTP Adaptive Streaming
HDS	HTTP Dymanic Streaming
HEVC	High Efficiency Video Coding
HLS	HTTP Live Streaming
HTML	Hyper-Text Markup Language
HTTP	Hyper-Text Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
ISO	International Organization for Standardization
ITU	International Telecommunications Union
KPI	Key Performance Indicators
LAN	Local Area Network
LED	Light-emitting Diode
Mbps	Megabits Per Second
MB	Megabyte

MEC	Multi-access Edge Computing
MOS	Mean Opinion Score
MP4	MPEG-4 Part 14 specification
MOV	QuickTime File Format specification
MPD	Media Presentation Description
MPEG	Motion Picture Expert Group
MS-SWG	Multi-Segment Send-While-Get
MSE	Media Source Extensions
NAT	Network Address Translation
OS	Operating System
P2P	Peer-to-Peer
QoE	Quality of Experience
QoS	Quality of Service
RDA	Rate Determination Algorithm
SAG	Send-After-Get
SINR	Signal to Interference & Noise Ratio
SIR	Signal to Interference Ratio
SNR	Signal to Noise Ratio
SSL	Secure Sockets Layer
SWG	Send-While-Get
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UE	User Equipment
URL	Uniform Resource Locator
VoD	Video on Demand
WAN	Wide Area Network
XML	Extensible Markup Language

APPENDIX I

Software implementation details

For the experimentation through which we obtained the results we presented in Chapters 2, 3 and 5, we performed a full system model implementation, not just a simulation like the majority of scientific papers does. The software we developed is available in the GitHub repository [25] in the form of binary executables with fully detailed instructions for the setup and the execution. Fig. 43 contains the structure and the contents of the repository.

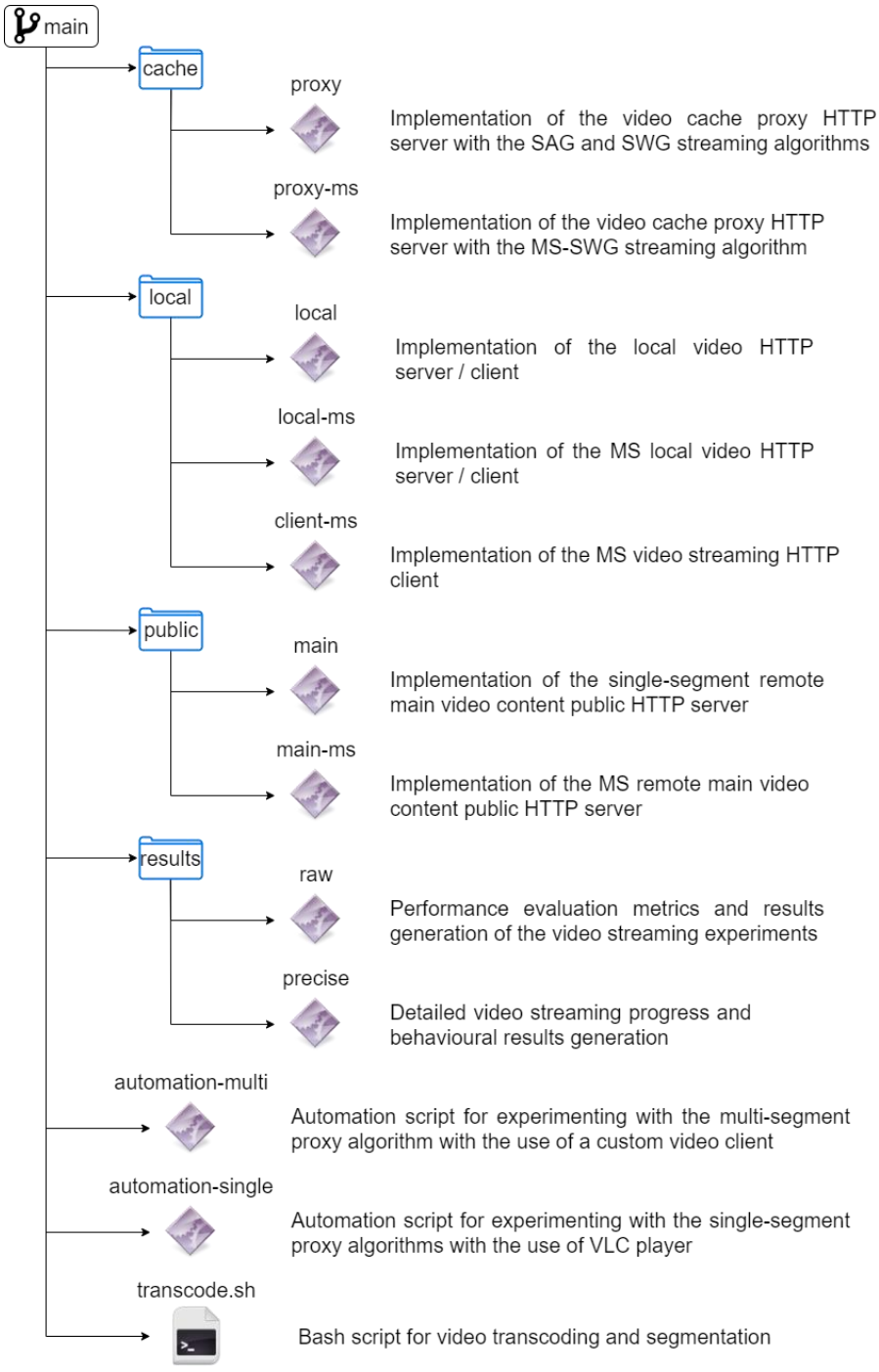

Figure 43: Software's GitHub repository structure

The implementation was done and only tested with the following software combination:

- Ubuntu [26] 20.04 Desktop
- Python [27] version 3.8
- VLC [28] media player version 3.0.9
- FFmpeg [29] version 4.2.x
- MP4Box [30] version 0.9 or newer

The experimentation software was written in the Python programming language with some additional pip packages (NumPy, Pandas, ISOdate, MPEGdash).

The data rate control was implemented in software, and in order to add some randomness to simulate an actual channel link's behavior, we introduced some randomness through the use of a Rayleigh distribution, so that the mean rate varies a little. More specifically, the mode σ of a Rayleigh random variable in our simulation is given by the following formula $\sigma = \sqrt{\frac{2}{\pi}}\,\mu(X)$ where $\mu(X)$ represents the mean data rate of the channel. This random data rate is recalculated repeatedly with a time interval of 1 second.

To be more precise, a data window of N bytes is generated which are supposed to be transmitted during the following interval of simulation time. Since this window may be exceeded before a full interval passes, because of the constant data chunk size of 4096 bytes (equal to the OS page size) that are being transmitted at any moment, when said window is exceeded the software will halt the execution for the remaining time, so that the total data size transmitted within every interval does not exceed N by more than 4095 bytes.

Video details

For our experimentation and measurements, we used the Big Buck Bunny [31] video with an original resolution of 2160p, a playback duration of 634.6 seconds and a frame rate of 30 FPS. We used the AVC/H.264 [32] video encoder of the FFmpeg software tool to transcoded the original video into the six following distinct video resolutions: 2160p, 1440p, 1080p, 720p, 480p, 360p as shown in Table 5.

Table 5: Video details

Resolution	Video File Size (MB)	Required Data Rate (Mbps)
2160p	728.51	9.18
1440p	418.80	5.28
1080p	276.43	3.48
720p	159.49	2.01
480p	96.22	1.21
320p	71.70	0.90

Figure 44: Big Buck Bunny poster

Afterwards, we used GPAC's MP4Box to partition each one of these resolutions into DASH-compatible video and audio segments suitable for individual/separate download through HTTP requests. We chose to create segments with a duration $T_f = 10$ seconds, which resulted in 64 video segments per resolution plus another 64 audio segments (448 segment files in total). In addition to that, MP4Box also created a helper manifest MPD file which can be used as an index and metadata holder for the generated segments, as well as a video and an audio initialization file. So, the CDN (main) contains 451 files in total with a total size of approximately 1.75 GB. Table 6 contains a detailed list of each segment's size in MB (including both video and audio sizes).

Table 6: Segment sizes per video resolution (including audio)

		Segment Size (MB)					
Segment ID	**Segment Duration (s)**	**2160p**	**1440p**	**1080p**	**720p**	**480p**	**320p**
1	10	8.44	4.41	3.33	1.57	0.99	0.75
2	10	19.61	10.26	6.97	3.66	2.45	1.70
3	10	9.91	5.76	3.92	2.22	1.34	0.97
4	10	19.04	10.94	7.49	4.07	2.27	1.50
5	10	8.33	4.75	3.35	2.02	1.41	1.07
6	10	11.84	6.52	4.51	2.66	1.65	1.20
7	10	6.92	3.90	2.65	1.51	0.97	0.78
8	10	11.56	6.22	4.11	2.28	1.38	1.03
9	10	13.73	7.23	4.85	2.68	1.66	1.24
10	10	5.19	3.12	2.24	1.40	0.93	0.74
11	10	7.00	3.85	2.58	1.49	0.97	0.77
12	10	8.04	4.49	3.08	1.79	1.15	0.90
13	10	12.63	6.48	4.22	2.31	1.43	1.07
14	10	9.48	5.18	3.49	1.95	1.21	0.92
15	10	10.05	5.49	3.69	2.07	1.27	0.96
16	10	9.60	5.23	3.44	1.96	1.23	0.95
17	10	8.16	4.43	2.95	1.73	1.12	0.88

18	10	6.66	3.63	2.46	1.44	0.96	0.78
19	10	11.95	6.48	4.33	2.50	1.56	1.18
20	10	11.90	6.42	4.22	2.31	1.40	1.05
21	10	8.60	4.75	3.28	1.83	1.14	0.88
22	10	12.72	7.26	5.00	2.86	1.79	1.35
23	10	10.38	5.29	3.35	1.77	1.08	0.83
24	10	10.13	5.29	3.44	1.88	1.15	0.89
25	10	11.36	6.16	4.13	2.32	1.41	1.04
26	10	11.82	6.28	4.16	2.35	1.48	1.14
27	10	10.91	5.88	3.89	2.16	1.33	1.00
28	10	13.87	7.37	4.84	2.58	1.52	1.13
29	10	13.83	7.34	4.79	2.58	1.51	1.10
30	10	6.75	3.72	2.54	1.49	1.00	0.80
31	10	12.72	6.92	5.15	2.97	1.67	1.17
32	10	13.04	7.17	4.78	2.65	1.59	1.18
33	10	15.42	8.22	5.41	2.93	1.76	1.29
34	10	12.83	6.85	4.57	2.54	1.55	1.17
35	10	10.22	5.77	3.93	2.26	1.42	1.05
36	10	17.36	9.66	6.53	3.61	2.21	1.54
37	10	18.96	10.24	6.73	3.76	2.20	1.58
38	10	13.61	7.03	4.68	2.63	1.60	1.14
39	10	6.27	3.47	2.34	1.36	0.89	0.71
40	10	6.64	3.83	2.62	1.55	1.01	0.79
41	10	8.81	4.92	3.39	1.92	1.22	0.95
42	10	8.36	4.68	3.25	1.93	1.27	0.98
43	10	16.94	9.56	6.67	3.84	2.35	1.69
44	10	23.74	13.22	8.96	4.89	2.82	1.96
45	10	16.60	9.78	7.09	3.95	2.36	1.72
46	10	5.95	3.28	2.28	1.42	1.00	0.82
47	10	17.97	8.85	5.59	2.83	1.60	1.11
48	10	15.56	8.29	5.41	2.92	1.67	1.16
49	10	6.67	3.62	2.47	1.47	0.99	0.77
50	10	6.98	3.86	2.63	1.48	0.96	0.75
51	10	10.23	5.36	3.89	1.96	1.11	0.87
52	10	13.11	6.76	4.57	2.40	1.45	1.10
53	10	14.41	7.50	5.09	2.56	1.44	1.07
54	10	14.22	7.27	4.91	2.54	1.55	1.11
55	10	9.90	9.55	5.74	4.53	2.43	1.93
56	10	11.29	10.85	6.55	5.04	2.60	2.08
57	10	16.21	11.38	7.07	3.96	2.50	1.76
58	10	12.22	6.13	4.47	2.24	1.30	0.95
59	10	19.03	12.85	7.11	4.68	2.49	1.71
60	10	17.28	15.61	7.57	5.65	2.94	2.07
61	10	7.29	7.32	4.17	3.24	1.80	1.40
62	10	0.99	0.73	0.62	0.53	0.49	0.46
63	10	4.73	2.67	1.87	1.14	0.80	0.66
64	4.6	2.57	1.48	1.02	0.64	0.43	0.36
TOTAL	634.6	728.51	418.80	276.43	159.49	96.22	71.70

Every video segment file has the file name pattern video_RESOLUTION_ID.m4s while every audio segment has the file name pattern video_audio_ID.m4s, where RESOLUTION is the video resolution and ID is the segment ID / number. All of these segments are available from main. For the actual segment requests and video playback for the end user, the VLC media player is employed.

Experimentation environment setup

In order to prepare the experimentation environment, first we must install the Ubuntu 20.04 OS on our PC. Then we should install all required the software tools (Python, VLC, FFmpeg, MP4Box, git, etc.) as described in the project's GitHub repository.

The next step is to download the binaries from the software's GitHub repository. This can either be cloned using git or downloaded as a compressed ZIP file.

To download the binaries using git, we need to use the following command:

```
git clone http://github.com/Fogus-Gr/recent-dash-proposed-caching.git
```

To download them as a compressed ZIP file we need to navigate to the software's GitHub repository with a web browser and click on Code → Download ZIP as shown in Fig. 45.

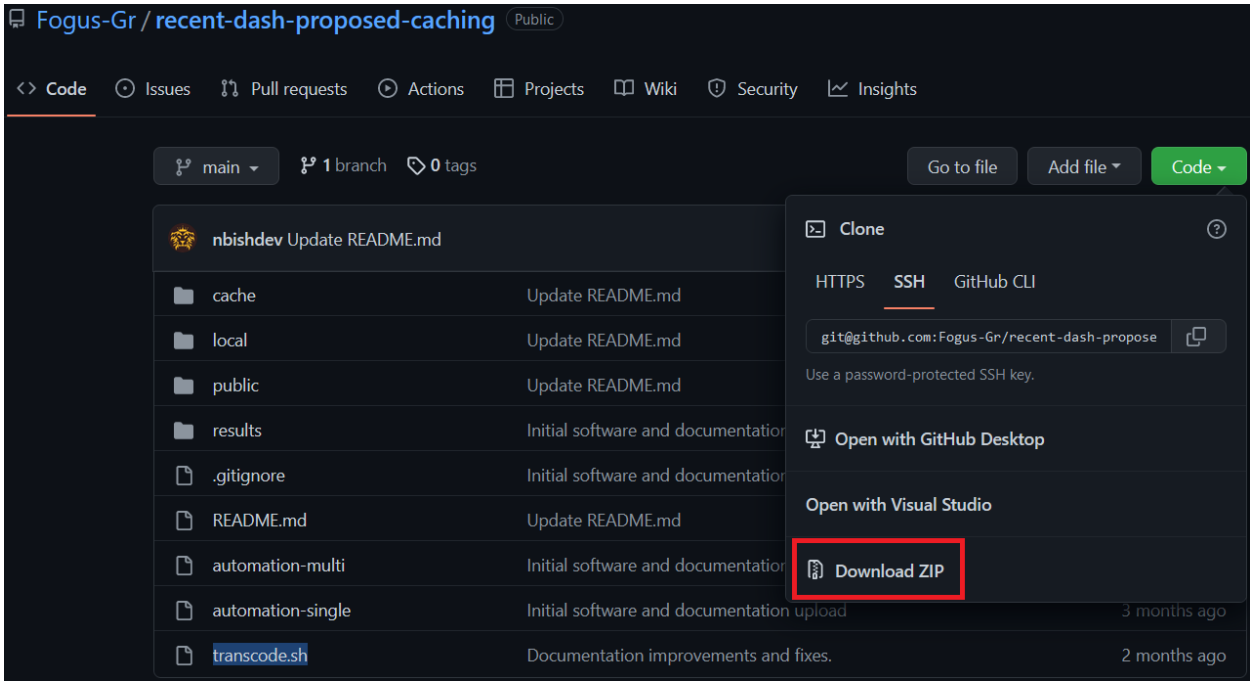

Figure 45: Download software as ZIP

Afterwards, we can either download the BBB video and manually segment it, or download a compressed ZIP file which contains the BBB video segments that were used for the experiments we performed in Chapter 5 and place them under main's serving directory. The existing segments can be downloaded from [33]. To segment the video ourselves, we can use the provided bash script file **transcode.sh**. This script locates all MP4 and MOV files in the execution directory and then it first uses FFmpeg's H.264 codec and a Constant Rate Factor (CRF) of 23, which is the default value (51 is for the worst possible video quality while 0 is for the best possible video quality), to transcode all these videos into the 6 video resolutions presented in Table 5 as MP4 files, afterwards it uses MP4Box to segment all generated MP4 files into DASH-compatible segments using the value of the SEGMENT_DURATION argument as for the generated segments' duration.

To manually transcode and segment the original video files and then place the output files under main's serving directory, we can use the transcode.sh bash script with the following

commands, replacing SEGMENT_DURATION with the desired segment duration (seconds) which in our experiments was equal to 10:

```
bash ./transcode.sh SEGMENT_DURATION
mv manifest.mpd ~/recent-d3x-video-streaming/public/
mv *init.mp4 ~/recent-d3x-video-streaming/public/
mv video*.m4s ~/recent-d3x-video-streaming/public/
```

Then, we are ready to move on to the actual software execution and experimentation.

Manual software execution instructions

As we mentioned in Chapter 2, the software we developed is split into three distinct components: the **main**, the **cache proxy** and the **local** / **client**. These components are independent from one another and can be independently executed. We will present the basic execution steps and some details on the input parameters.

First, before any software is executed, we must make sure that we have placed all the output files generated by MP4Box under main's serving directory, otherwise the software execution will be unsuccessful. If we want to have cached segments in the cache proxy's buffer, we should manually copy them to the cache proxy's serving directory before the video streaming procedure is initiated.

We will now present the software components execution instructions. The version of the components to execute should be properly selected so that they are compatible and cooperate seamlessly with each other. For example when the single-segment version of main is executed, if the multi-segment version of proxy is executed alongside it the video streaming experiment will fail because of errors due to incompatibilities.

For the IP addresses of the three software components, we use the (default) IP addresses and serving ports presented in Table 7.

Table 7: IP addresses and serving ports of system model's network components

Component	IP address	Serving port
Main	127.0.0.4	8004
Cache Proxy	127.0.0.3	8003
Local	127.0.0.2	8002
VLC	-	-

Figure 46: Experimentation network topology

Main

Second, we should execute main, since this is the basic source of video content and proxy will have to communicate with main for the missing / non-cached content.

To execute the single-segment version of main, we use the following command template:

```
usage: main [-h] [-a ADDRESS] [-p PORT] [-d DIRECTORY] -r2 RATE2
                 [-s SAMPLE] [--rayleigh]

Single-segment remote main video content public HTTP server

optional arguments:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        IP address
  -p PORT, --port PORT  serving port
  -d DIRECTORY, --directory DIRECTORY
                        serving directory
  -r2 RATE2, --rate2 RATE2
                        data rate R2 [from main to proxy] (Mbps)
  -s SAMPLE, --sample SAMPLE
                        sample number
  --rayleigh            introduce randomness in the data channel [from main to
                        proxy] through Rayleigh distribution
```

An example execution with a data rate $R_2 = 5$ Mbps and channel randomness through a Rayleigh distribution could be the following:

```
./main -a 127.0.0.4 -p 8004 -d . -r2 5.0 --rayleigh
```

To execute the multi-segment version of main, we use the following command template:

```
usage: main-ms [-h] [-a ADDRESS] [-p PORT] [-d DIRECTORY] -r2 RATE2
                    [-s SAMPLE] [--rayleigh]

Multi-Segment remote main video content public HTTP server

optional arguments:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        IP address
  -p PORT, --port PORT  serving port
  -d DIRECTORY, --directory DIRECTORY
                        serving directory
  -r2 RATE2, --rate2 RATE2
                        data rate R2 [from main to proxy] (Mbps)
  -s SAMPLE, --sample SAMPLE
                        sample number
  --rayleigh            introduce randomness in the data channel [from main to
                        proxy] through Rayleigh distribution
```

An example execution with a data rate $R_2 = 5$ Mbps and channel randomness through a Rayleigh distribution could be the following:

```
./main-ms -a 127.0.0.4 -p 8004 -d . -r2 5.0 --rayleigh
```

Cache Proxy

Now the main server should be up and running. So, we should be able to move on with the execution of the cache proxy.

To execute the single-segment version of proxy, we use the following command template:

```
usage: proxy [-h] [-a ADDRESS] [-p PORT] [-sa REMOTEADDRESS] [-sp REMOTEPORT]
                  [-d DIRECTORY] -al {sag,swg} -r1 RATE1 [-r2 RATE2]
                  [-l BUFFERSIZE] [-dl {predictive,fixed}] -c {random,ripple}
                  [-s SAMPLE] [-n NUMFILES] [--rayleigh]

Single-Segment Send-After-Get (SAG) / Send-While-Get (SWG) video cache proxy
HTTP server / client

optional arguments:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        IP address
  -p PORT, --port PORT  serving port
  -sa REMOTEADDRESS, --remoteaddress REMOTEADDRESS
                        remote server IP address
  -sp REMOTEPORT, --remoteport REMOTEPORT
                        remote server serving port
  -d DIRECTORY, --directory DIRECTORY
                        serving directory
  -al {sag,swg}, --algorithm {sag,swg}
                        proxy algorithm option
  -r1 RATE1, --rate1 RATE1
                        data rate R1 [from proxy to client] (Mbps)
  -r2 RATE2, --rate2 RATE2
                        data rate R2 [from main to proxy] (Mbps)
  -l BUFFERSIZE, --buffersize BUFFERSIZE
                        buffer size L (MB)
  -dl {predictive,fixed}, --dashlogic {predictive,fixed}
                        DASH resolution logic
  -c {random,ripple}, --caching {random,ripple}
                        caching algorithm option
  -s SAMPLE, --sample SAMPLE
                        sample number
  -n NUMFILES, --numfiles NUMFILES
                        total number of files required for a video stream
                        (automatic proxy termination when reached)
  --rayleigh            introduce randomness in the data channel [from proxy
                        to client] through Rayleigh distribution
```

An example execution with the SWG streaming algorithm, a data rate $R_1 = 15$ Mbps, a communication with main on the IP address 127.0.0.4 with a serving port of 8004, and channel randomness through a Rayleigh distribution could be the following:

```
./proxy -a 127.0.0.3 -p 8003 -sa 127.0.0.4 -sp 8004 -d . -al swg --rayleigh
```

To execute the multi-segment version of proxy, we use the following command template:

```
usage: proxy-ms [-h] [-a ADDRESS] [-p PORT] [-sa REMOTEADDRESS] [-sp REMOTEPORT]
                    [-d DIRECTORY] -r1 RATE1 [-r2 RATE2] [-l BUFFERSIZE]
                    -c {random,ebc,tier} [-s SAMPLE] [-n NUMFILES] [--rayleigh]

Multi-Segment Send-While-Get video cache proxy HTTP server / client

optional arguments:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        IP address
  -p PORT, --port PORT  serving port
  -sa REMOTEADDRESS, --remoteaddress REMOTEADDRESS
                        remote server IP address
  -sp REMOTEPORT, --remoteport REMOTEPORT
                        remote server serving port
  -d DIRECTORY, --directory DIRECTORY
                        serving directory
  -r1 RATE1, --rate1 RATE1
                        data rate R1 [from proxy to client] (Mbps)
  -r2 RATE2, --rate2 RATE2
                        data rate R2 [from main to proxy] (Mbps)
  -l BUFFERSIZE, --buffersize BUFFERSIZE
                        buffer size L (MB)
  -c {random,ebc,tier}, --caching {random,ebc,tier}
                        caching algorithm option
  -s SAMPLE, --sample SAMPLE
                        sample number
  -n NUMFILES, --numfiles NUMFILES
                        total number of files required for a video stream
                        (automatic proxy termination when reached)
  --rayleigh            introduce randomness in the data channel [from proxy to
                        client] through Rayleigh distribution
```

An example execution with a data rate $R_2 = 15$ Mbps, a communication with main on the IP address 127.0.0.4 with a serving port of 8004 and channel randomness through a Rayleigh distribution could be the following:

```
./proxy-ms -a 127.0.0.3 -p 8003 -sa 127.0.0.4 -sp 8004 -d . -r1 15.0 --rayleigh
```

Local / Client

Now that the cache proxy is also up and running, we can move on with the execution of the client. In the client's case we have 3 distinct software options:

- **Local**: this is the single-segment version of local which will be used with the VLC player for video playback
- **MS Local**: this is the multi-segment version of local which will also be used with the VLC player for video playback
- **Client**: this is a custom standalone multi-segment client which successfully streams the video content but does not have video playback capabilities

In the cases of both single-segment and multi-segment versions of local, the client consists of the combination of the local with the VLC player. This also means that the VLC player must be launched for the video streaming to begin.

To execute the single-segment version of local, we use the following command template:

```
usage: local [-h] [-a ADDRESS] [-p PORT] [-sa REMOTEADDRESS]
                  [-sp REMOTEPORT] [-d DIRECTORY]

Local video content HTTP server / client

optional arguments:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        IP address
  -p PORT, --port PORT  serving port
  -sa REMOTEADDRESS, --remoteaddress REMOTEADDRESS
                        remote server IP address
  -sp REMOTEPORT, --remoteport REMOTEPORT
                        remote server serving port
  -d DIRECTORY, --directory DIRECTORY
                        serving directory
```

An example execution with a communication with proxy on the IP address 127.0.0.3 with a serving port of 8003 could be the following:

```
./local -a 127.0.0.2 -p 8002 -sa 127.0.0.3 -sp 8003 -d .
```

To execute the multi-segment version of local, we use the following command template:

```
usage: local-ms [-h] [-a ADDRESS] [-p PORT] [-sa REMOTEADDRESS]
                     [-sp REMOTEPORT] [-d DIRECTORY]

Local video content HTTP server / client

optional arguments:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        IP address
  -p PORT, --port PORT  serving port
  -sa REMOTEADDRESS, --remoteaddress REMOTEADDRESS
                        remote server IP address
  -sp REMOTEPORT, --remoteport REMOTEPORT
                        remote server serving port
  -d DIRECTORY, --directory DIRECTORY
                        serving directory
```

An example execution with a communication with proxy on the IP address 127.0.0.3 with a serving port of 8003 could be the following:

```
./local-ms -a 127.0.0.2 -p 8002 -sa 127.0.0.3 -sp 8003 -d .
```

To execute the multi-segment standalone client, we use the following command template:

```
usage: client-ms [-h] [-sa REMOTEADDRESS] [-sp REMOTEPORT] [-d DIRECTORY]
                      -r {2160,1440,1080,720,480,360}

Multi-Segment video streaming HTTP client

optional arguments:
  -h, --help            show this help message and exit
  -sa REMOTEADDRESS, --remoteaddress REMOTEADDRESS
                        remote server IP address
  -sp REMOTEPORT, --remoteport REMOTEPORT
                        remote server serving port
  -d DIRECTORY, --directory DIRECTORY
                        serving directory
  -r {2160,1440,1080,720,480,360}, --resolution {2160,1440,1080,720,480,360}
                        video resolution to be streamed
```

An example execution with a communication with proxy on the IP address 127.0.0.3 with a serving port of 8003 could be the following:

```
./client-ms -sa 127.0.0.3 -sp 8003 -d . -r 1440
```

VLC

If the MS client has been selected, the video streaming experiment should have been initiated. Otherwise, we need to manually execute the VLC player and insert the required parameters to initiate the video streaming experiment.

To execute VLC we use the following command template:

```
vlc http://LOCAL_ADDRESS:PORT/manifest.mpd --adaptive-logic LOGIC
--adaptive-maxheight HEIGHT --stream-filter hds --ignore-config -I dummy
--no-keyboard-events --no-mouse-events vlc://quit
```

The LOGIC argument refers to the streaming resolution logic, the HEIGHT argument refers to the highest desired video height, while the rest of the command-line arguments for VLC make sure that:

- HTTP Dynamic Streaming is used as the video streaming technique
- Locally stored VLC user configuration is skipped when VLC is launched
- The simplest UI version of VLC is loaded
- VLC ignores any mouse and keyboard inputs
- VLC automatically closes when playback completes

An example execution with a communication with local on the IP address 127.0.0.2 with a serving port of 8002 and an adaptive video resolution could be the following:

```
vlc http://127.0.0.2:8002/manifest.mpd --adaptive-logic predictive
--adaptive-maxheight 2160 --stream-filter hds --ignore-config -I dummy
--no-keyboard-events --no-mouse-events vlc://quit
```

An example execution with a communication with local on the IP address 127.0.0.2 with a serving port of 8002 and a fixed video resolution of 1080p could be the following:

```
vlc http://127.0.0.2:8002/manifest.mpd --adaptive-logic highest
--adaptive-maxheight 1080 --stream-filter hds --ignore-config -I dummy
--no-keyboard-events --no-mouse-events vlc://quit
```

Automated software execution instructions

The manual software execution instruction presented above are too complicated and require too many steps. Instead, we have developed two automation binaries which can take care of the whole experimentation setup and software component execution procedures. These binaries are the following:

- **automation-single**: to be used for experimenting with the single-segment SAG and SWG video streaming algorithms with the use of the VLC player
- **automation-multi**: to be used for experimenting with the MS-SWG algorithm either with the use of the VLC player of the custom MS client

To experiment with Single-Segment video streaming, we use the following command template:

```
usage: automation-single [-h] -r1 r1 [r1 ...] -r2 r2 [r2 ...] -l l [l ...]
                             [-d TARGETIPD] -c {random,ripple} -a {sag,swg}
                             -dl {predictive,fixed} [-p PATIENCE] [-s SAMPLES]
                             [--rayleigh] [--suppress] [--proposed]

Automation script for experimenting with the single-segment proxy algorithms with
the use of VLC player

optional arguments:
  -h, --help            show this help message and exit
  -r1 r1 [r1 ...], --rate1 r1 [r1 ...]
                        value(s) for data rate R1
  -r2 r2 [r2 ...], --rate2 r2 [r2 ...]
                        value(s) for data rate R2
  -l l [l ...], --buffersize l [l ...]
                        value(s) for buffer size L
  -d TARGETIPD, --targetipd TARGETIPD
                        target initial playout delay (sec)
  -c {random,ripple}, --caching {random,ripple}
                        caching algorithm option
  -a {sag,swg}, --algorithm {sag,swg}
                        proxy algorithm option
  -dl {predictive,fixed}, --logic {predictive,fixed}
                        DASH resolution logic
  -p PATIENCE, --patience PATIENCE
                        time to wait until playback completes (expressed as a
                        fraction of the video duration)
  -s SAMPLES, --samples SAMPLES
                        number of samples for every parameters combination
  --rayleigh            introduce randomness in data channels through Rayleigh
                        distribution
  --suppress            suppress video and audio playback
  --proposed            only run experiments in which L>0, R1>R2 and
                        R1>=Required_throughput>=R2
```

Table 8 presents the values the standard arguments can take in the command above.

Table 8: Single-segment standard arguments values

Proxy Algorithm	DASH Logic	Caching Technique
sag	predictive	random
swg	fixed	ripple

For example, to reproduce our presented experiments with the combination of Random caching, SWG streaming and Fixed video resolution, we can use the following command:

```
./automation-single  -r1 1.4 2 3 5.7 8.3 12 15.8 18 -r2 1 2 4 6 8
-l 0 50 100 200 300 400 -d 10 -c random -a swg -dl fixed -p 10 -s 3
```

To experiment with Multi-Segment video streaming, we use the following command template:

```
usage: automation-multi [-h] -r1 r1 [r1 ...] -r2 r2 [r2 ...] -l l [l ...]
                            [-d TARGETIPD] -c {random,ebc,tier} -t {client,vlc}
                            [-p PATIENCE] [-s SAMPLES] [--rayleigh] [--suppress]
                            [--proposed]

Automation script for experimenting with the multi-segment proxy algorithm either
with the use of a custom video client (recommended) or with the use of VLC player

optional arguments:
  -h, --help            show this help message and exit
  -r1 r1 [r1 ...], --rate1 r1 [r1 ...]
                        value(s) for data rate R1
  -r2 r2 [r2 ...], --rate2 r2 [r2 ...]
                        value(s) for data rate R2
  -l l [l ...], --buffersize l [l ...]
                        value(s) for buffer size L
  -d TARGETIPD, --targetipd TARGETIPD
                        target initial playout delay (sec)
  -c {random,ebc,tier}, --caching {random,ebc,tier}
                        caching algorithm option
  -t {client,vlc}, --tool {client,vlc}
                        tool for performing the video streaming
  -p PATIENCE, --patience PATIENCE
                        time to wait until playback completes (expressed as a
                        fraction of the video duration)
  -s SAMPLES, --samples SAMPLES
                        number of samples for every parameters combination
  --rayleigh            introduce randomness in data channels through Rayleigh
                        distribution
  --suppress            suppress video and audio playback
  --proposed            only run experiments in which L>0, R1>R2 and
                        R1>=Required_throughput>=R2
```

Table 9 presents the values the standard arguments can take in the command above.

Table 9: Multi-segment standard arguments values

Caching Technique	Tool
random	client
ebc	vlc
tier	

In the caching algorithm selection, **tier** stands for 1-Tiered EBC (Recursion depth = 1).

To reproduce our presented experiments with the combination of EBC, MS-SWG streaming and Fixed video resolution, we can use the following command:

```
./automation-multi -r1 1.4 2 3 5.7 8.3 12 15.8 18 -r2 1 2 4 6 8
-l 0 50 100 200 300 400 -d 10 -c ebc -t client -p 10 -s 3
```

Console visual output

If the software execution was successful, three or four terminals should be launched. All three system model software components contain an HTTP server and output a message every time an HTTP response is sent. The Cache Proxy, in addition to this message, outputs the message CACHE HIT or CACHE MISS based on whether the requested segment existed in its cache when it was requested or not.

Fig. 47 presents a snapshot of the four console outputs during a single-segment video streaming experiment with a fixed video resolution of 720p. This snapshot shows the cache proxy's status message when the 33rd video segment is unavailable from its cache, so it gets requested from main, then served to local. Fig. 48 presents a snapshot of the three console outputs during a multi-segment video streaming experiment, through which we can see the multiple parallel segment requests. In Fig. 49 we can see the terminal output of proxy and the cache hit / miss notifications during a single-segment video streaming process. In Fig. 50 we can see a screenshot of the video stream playback going on in VLC.

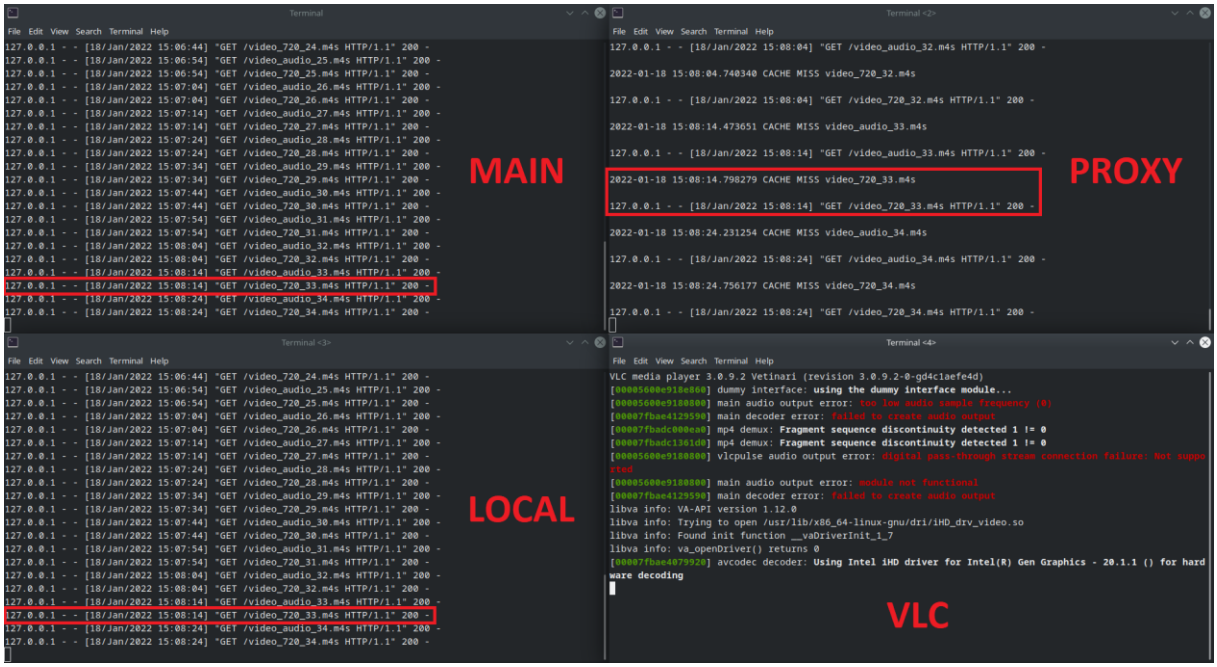

Figure 47: Single-segment video streaming console outputs snapshot

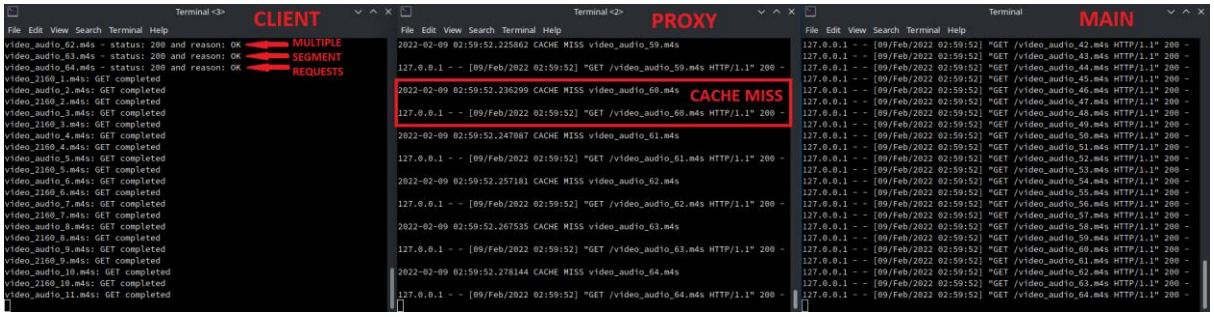

Figure 48: Multi-segment video streaming console outputs snapshot

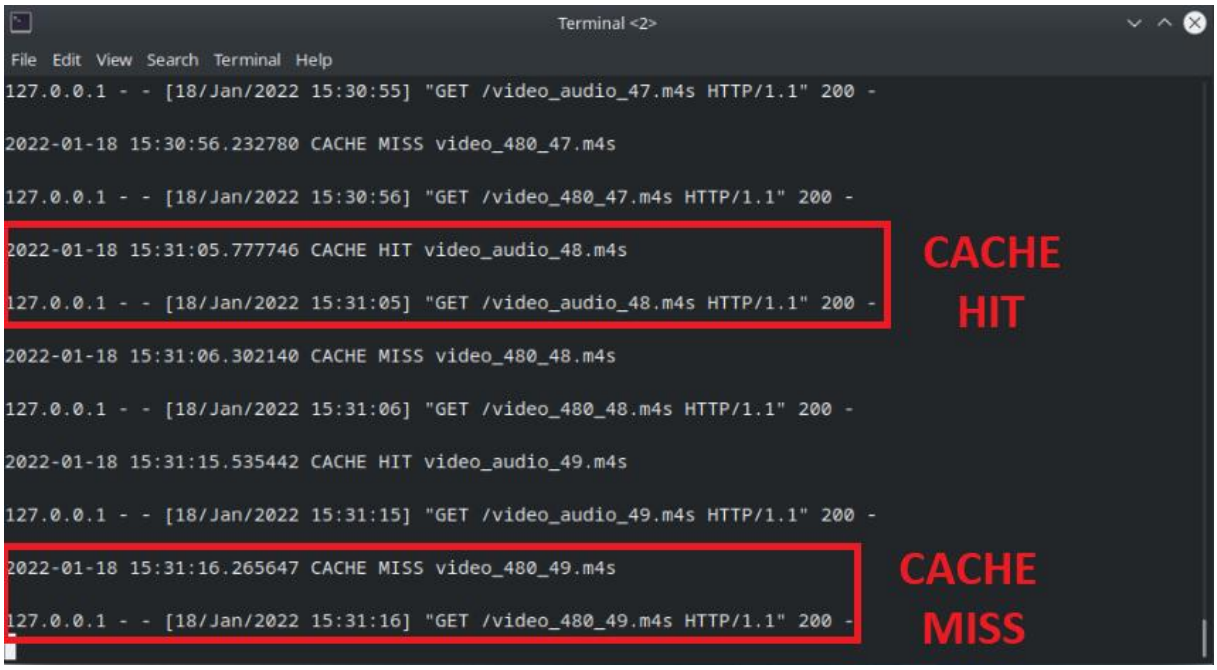

Figure 49: Cache hit /miss console output indication

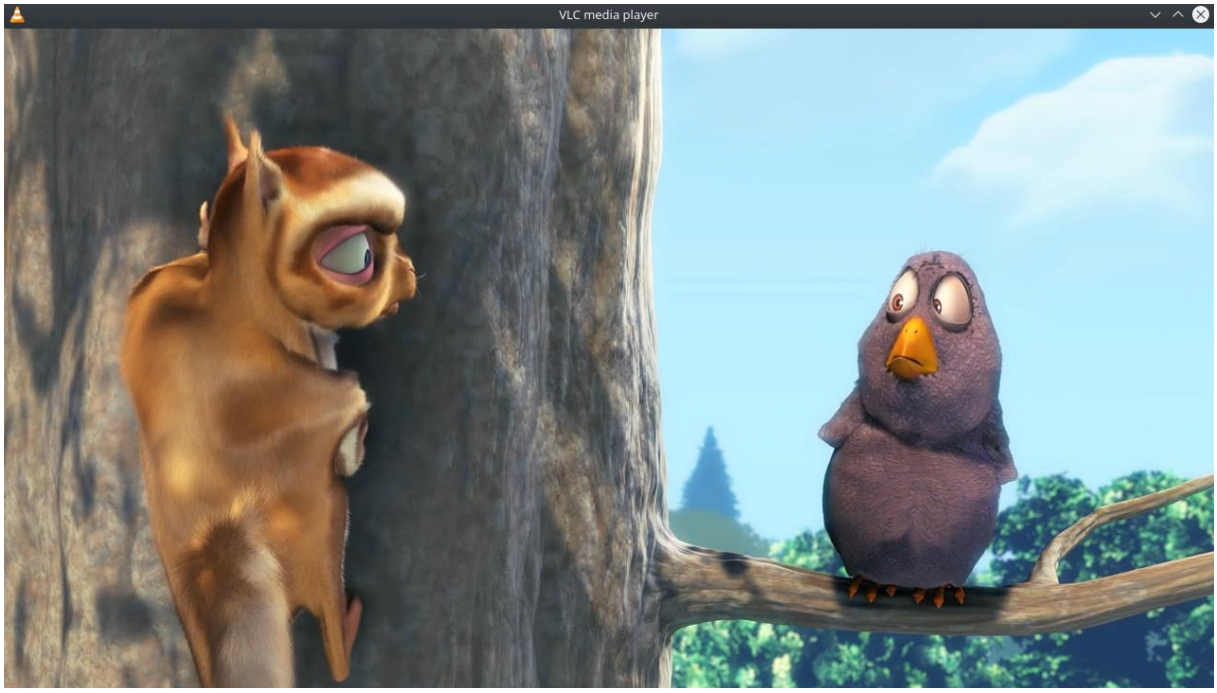

Figure 50: VLC video stream playback snapshot

Logging

The cache proxy, acting as the middle man in the video segment request and streaming process, produces a log file at the end of a video streaming procedure and places said log file under the **cache/** directory. These logs can be used for determining any anomalies or abnormal behavior of the software components during the video streaming procedure and for evaluating the performance of the whole system model. These logs will be generated when the proxy has streamed to the client all of the segments that it was expected to stream, if it was the proper command-line arguments were provided at the execution initialization.

Let's take a look at the format of the log files. The following lines are taken from a log file produced during a video streaming procedure with the SAG algorithm. Every line at the beginning contains the timestamp of the operation in nanoseconds, the name of the segment that is being processed, the operation type, the operation status and whether the processed segment was cached or not.

```
[991843954804622] manifest.mpd SERVE START HIT
[991843954937736] manifest.mpd SERVE END HIT
[991843976017796] manifest_set1_init.mp4 SERVE START HIT
[991843976037978] manifest_set1_init.mp4 SERVE END HIT
[991843983907374] video_2160_1.m4s REQUEST START MISS
[991855033970384] video_2160_1.m4s REQUEST END MISS
[991855033978958] video_2160_1.m4s SERVE START MISS
[991858041226108] video_2160_1.m4s SERVE END MISS
[991858057554038] video_audio_init.mp4 SERVE START HIT
[991858057581652] video_audio_init.mp4 SERVE END HIT
[991858065466946] video_audio_1.m4s SERVE START HIT
[991858065944120] video_audio_1.m4s SERVE END HIT
[991858083782768] video_audio_2.m4s REQUEST START MISS
[991858085089457] video_audio_2.m4s REQUEST END MISS
[991858085094141] video_audio_2.m4s SERVE START MISS
[991858085609246] video_audio_2.m4s SERVE END MISS
[991858093488287] video_360_2.m4s REQUEST START MISS
[991860090255892] video_360_2.m4s REQUEST END MISS
[991860090260423] video_360_2.m4s SERVE START MISS
[991860091880995] video_360_2.m4s SERVE END MISS
[991860128916085] video_2160_3.m4s REQUEST START MISS
[991869114619014] video_2160_3.m4s REQUEST END MISS
[991869114623690] video_2160_3.m4s SERVE START MISS
[991873127361984] video_2160_3.m4s SERVE END MISS
[991873150973445] video_audio_3.m4s SERVE START HIT
[991873151449133] video_audio_3.m4s SERVE END HIT
[991873159662632] video_2160_4.m4s REQUEST START MISS
[991894248206524] video_2160_4.m4s REQUEST END MISS
[991894248214082] video_2160_4.m4s SERVE START MISS
[991900263582304] video_2160_4.m4s SERVE END MISS
[991900273336736] video_audio_4.m4s REQUEST START MISS
[991900274739630] video_audio_4.m4s REQUEST END MISS
[991900274744767] video_audio_4.m4s SERVE START MISS
[991900275674941] video_audio_4.m4s SERVE END MISS
```

Results

The log files described above can be processed to extract the values for the metrics described in Chapter 5. These metric-value combinations can be used to perform performance evaluation for the experiment. This log processing can be performed using the ***raw*** executable located under results directory, with the following command template:

```
usage: raw [-h] [-i INIT] [-p PATH] [-t THRESHOLD]

Performance evaluation metrics and results generation of the video streaming
experiments

optional arguments:
  -h, --help            show this help message and exit
  -i INIT, --init INIT  number of video segments considered as part of the initial
                        playback delay
  -p PATH, --path PATH  path to directory with proxy outputs and video metadata
  -t THRESHOLD, --threshold THRESHOLD
                        time threshold under which delays are ignored
```

When this log processing is complete a set of TEXT files with the name prefix ***results***_ are generated (one for each conducted experiment) containing some preliminary results, alongside a CSV file with the name prefix ***all_results***_ which contains detailed evaluation metric values for all experiments in a single file.

If we want to extract more detailed behavioral results for each experiment with respect to the execution time, like the ones we presented in Chapters 2 and 3, we can use the ***precise*** executable located under results directory, with the following command template:

```
usage: precise [-h] [-i INIT] [-p PATH] [-t THRESHOLD] [-c]

Detailed video streaming progress and behavioural results generation

optional arguments:
  -h, --help            show this help message and exit
  -i INIT, --init INIT  number of video segments considered as part of the initial
                        playback delay
  -p PATH, --path PATH  path to directory with proxy outputs and video metadata
  -t THRESHOLD, --threshold THRESHOLD
                        time threshold under which delays are ignored
  -c, --combine         combine different outputs' timestamps for better comparison
```

When this log processing is complete, a set of CSV files with the name prefix ***analytical***_ are generated (one for each conducted experiment) containing start-end time ranges and the respective development of the values for some evaluation metrics during the video streaming procedure.

Charts generation

Using the CSV files generated by the previously presented two binary executables, we can create performance evaluation charts like the ones we presented in Chapters 2, 3 and 5. For the creation of said charts, the Matplotlib library was employed.

REFERENCES

[1] N. Episkopos, “On-device caching of popular video content on Android-powered devices”, Graduate (Bachelor) Thesis, Department of Informatics and Telecommunications – National and Kapodistrian University of Athens (2018) (in Greek).
[2] Fizza, Maryam and Munam Ali Shah. “5G Technology: An Overview of Applications, Prospects, Challenges and Beyond” (2015).
[3] A. Abuzaid, “Navigating 5G-era network architecture”, EXFO (2020).
[4] Ayenew, Tadege Mihretu & Xenakis, Dionysis & Passas, Nikos & Merakos, Lazaros. (2021). Cooperative Content Caching in MEC-Enabled Heterogeneous Cellular Networks. IEEE Access. PP. 1-1. 10.1109/ACCESS.2021.3095356.
[5] Mehrabi, Mahshid & You, Dongho & Latzko, Vincent & Salah, Hani & Reisslein, Martin & Fitzek, Frank. (2019). Device-Enhanced MEC: Multi-Access Edge Computing (MEC) Aided by End Device Computation and Caching: A Survey. IEEE Access. PP. 1-1. 10.1109/ACCESS.2019.2953172.
[6] Trestian, Ramona & Comsa, Ioan Sorin & Tuysuz, Mehmet. (2018). Seamless Multimedia Delivery within a Heterogeneous Wireless Networks Environment: Are we there yet?. IEEE Communications Surveys & Tutorials. PP. 1-1. 10.1109/COMST.2018.2789722.
[7] “ABR Logic”, GitHub.
[8] Laghari, Khalil & Connelly, Kay. (2012). Toward Total Quality of Experience: A QoE Model in a Communication Ecosystem. Communications Magazine, IEEE. 50. 58-65. 10.1109/MCOM.2012.6178834.
[9] DASH specification.
[10] Open IPTV Forum Solution Specification Volume 2a – HTTP Adaptive Streaming V2.1, 2011-10-09 at the Wayback Machine.
[11] HTTP/2-Based Methods to Improve the Live Experience of Adaptive Streaming.
[12] Huysegems, Rafael & van der Hooft, Jeroen & Bostoen, Tom & Rondao Alface, Patrice & Petrangeli, Stefano & Wauters, Tim & De Turck, Filip. (2015). HTTP/2-Based Methods to Improve the Live Experience of Adaptive Streaming. 10.1145/2733373.2806264.
[13] Christopher Mueller, “MPEG-DASH (Dynamic Adaptive Streaming over HTTP)”, Bitmovin (2022).
[14] Huy, Vu & Mashal, Ibrahim & Chung, Tein yaw. (2017). A novel bandwidth estimation method based on MACD for DASH. 11. 1441-1461. 10.3837/tiis.2017.03.011.
[15] ACM SIGMM Records, “Open Source Column: Dynamic Adaptive Streaming over HTTP Toolset” (2013).
[16] Brendan Long, “The structure of an MPEG-DASH MPD” (2015).
[17] SwiftCache Lite, “Different Types of Caching and Their Differences”.
[18] Symantec, “A Technical Review of Caching Technologies” (2017).
[19] (2008). Proxy-Caching for Video Streaming Applications. In: Furht, B. (eds) Encyclopedia of Multimedia. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-78414-4_590.
[20] Amazon AWS, “Content Delivery Network (CDN) Caching”.
[21] Domantas G. & Tashia T., “What Is CDN? Content Delivery Network Explained”, Hostinger Tutorials (2022).
[22] A. Mehrabi, M. Siekkinen and A. Yla-Jaaski, “Edge Computing Assisted Adaptive Mobile Video Streaming”, IEEE Trans. on Mob. Comput., vol. 18, no. 4, pp. 787-800, 1 April 2019.
[23] T. X. Tran et al., “Adaptive Bitrate Video Caching and Processing in Mobile-Edge Computing Networks”, IEEE Trans. on Mobil. Comput., vol.18, no.9, pp.1965-1978, Sept 2019.
[24] G. Kourouniotis, V. Georgara, “Cache-Aware Adaptive Video Streaming in 5G networks”, Postgraduate Diploma Thesis, Department of Informatics and Telecommunications – National and Kapodistrian University of Athens (2021).
[25] Experimentation GitHub.
[26] Ubuntu is a free and open-source GNU/Linux distribution based on Debian and developed by Canonical Ltd.
[27] Python Software Foundation. Python Language Reference, version 3.8.
[28] VideoLan VLC media player (2006).
[29] FFmpeg is a free and open-source suite of libraries and programs for handling video, audio, and other multimedia files and streams.
[30] Jean Le Feuvre. (2020). GPAC filters. 10.1145/3339825.3394929.
[31] Big Buck Bunny, short computer-animated comedy film featuring animals of the forest, made by the Blender Institute, part of the Blender Foundation, 2008, available at peach.blender.org.
[32] Richardson, Iain E. G. H.264 and MPEG-4 Video Compression: Video Coding for next Generation Multimedia. Chichester; Hoboken, NJ: Wiley, 2003.
[33] Segmentation output DASH-compatible files, produced from the original Big Buck Bunny video with video resolution 2160p.

[34] IEEE/ACM Transactions on Networking journal