NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

GRADUATE PROGRAM
COMPUTER, TELECOMMUNICATIONS AND NETWORK ENGINEERING

MSc THESIS

Cache-Aware Adaptive Video Streaming in 5G Networks

Vasiliki D. Georgara
Georgios I. Kourouniotis

Supervisor: **Lazaros Merakos,** Professor

ATHENS

APRIL 2021

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Προσαρμοστική ροή video με γνώσεις ενταμίευσης σε δίκτυα 5G

Βασιλική Δ. Γεωργάρα
Γεώργιος Ι. Κουρουνιώτης

Επιβλέπων: **Λάζαρος Μεράκος**, Καθηγητής

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2021

MSc Thesis

Cache-Aware Adaptive Video Streaming in 5G Networks

Vasiliki D. Georgara
S.N.: EN2190002

Georgios I. Kourouniotis
S.N.: EN2190003

APRIL 2021

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Προσαρμοστική ροή βίντεο με γνώσεις ενταμίευσης σε δίκτυα 5G

Βασιλική Δ. Γεωργάρα
Α.Μ.: ΕΝ2190002

Γεώργιος Ι. Κουρουνιώτης
Α.Μ.: ΕΝ2190003

ΕΠΙΒΛΕΠΩΝ: **Λάζαρος Μεράκος,** Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Λάζαρος Μεράκος,** Καθηγητής
Νικόλαος Πασσάς, Μέλος Ε.ΔΙ.Π.
Στάθης Χατζηευθυμιάδης, Καθηγητής

ΑΠΡΙΛΙΟΣ 2021

ABSTRACT

Dynamic Adaptive Streaming over HTTP (DASH) has prevailed as the dominant way of video transmission over the Internet. This technology is based on receiving small sequential video segments from a server. However, one challenge that has not been adequately examined, is the obtainment of video segments from more than one server, in a way that serves both the needs of the network and the improvement of the Quality of Experience (QoE). This thesis will investigate this problem by simulating a network with multiple video servers and a video client. It will then implement both the peer-to-many communication in the context of adaptive video streaming and the video server caching algorithm based on proposed criteria that will improve the status of the network and/or the user. All of this will be explored in the environment of Mininet, which is a network emulator, in order to simulate the DASH technology with the help of the emulator network nodes. Initially, the video was split into small segments using the ffmpeg tool, and then experiments were conducted in which a client requested the video from a cache server. If the segment could not be found in the cache server, then a request was sent from the cache server to a server that contained all segments of the video (main server). In these experiments, the added traffic was also examined, by concluded to the fact that the Mininet environment causes unavoidable limitations in the case of the traffic. What we observed was that the main server channel remained inactive throughout the requests of the cache server, resulting in unrealistic network conditions. For this reason, we have explored a new approach, eliminating the Mininet environment and working on new techniques for adding web traffic and modifying the communication of the servers, regarding the requests they receive. In this way, we were able to clearly show the limitations of the previous approach but also to conclude that the existence of caching servers is a useful tool in terms of increasing the quality of experience. The general tendency was that, as the available buffer size increased, the video playback quality increased to some extent. However, at the same time this improvement is linked to the random selection algorithm. For even better results, it is considered necessary to find an appropriate caching selection algorithm in order to take full advantage of the caching technology.

The following chapters presented in this thesis are: Chapter 1 mentions the historical background of the networks. Chapter 2 analyzes the Dynamic Adaptive Streaming over HTTP. Chapter 3 analyzes the caching techniques. Chapter 4 presents the concept of Quality of Experience and its correlation with many other factors. Chapter 5 describes in detail the process of setting up the environment and the various necessary tools for our implementation. Chapter 6 refers to the Mininet experiments, the topology, and the set-up, as well as the reasons that led us to a different approach. Chapter 7 proposes the different approach and presents the methodology and the metrics. Also, diagrams extracted from the analysis of the metrics are analyzed in Chapter 7. Finally, Chapter 8 summarizes the conclusions and issues of future research to improve the Quality of Experience even further.

SUBJECT AREA: Communication Networks

KEYWORDS: Quality of Experience, Video, Dynamic Adaptive Streaming over HTTP, Mininet, Caching

ΠΕΡΙΛΗΨΗ

Η τεχνολογία προσαρμοστικής ροής video μέσω HTTP έχει επικρατήσει ως ο κυρίαρχος τρόπος μετάδοσης video στο Internet. Η τεχνολογία αυτή βασίζεται στη λήψη μικρών διαδοχικών τμημάτων video από έναν server. Μία πρόκληση που όμως δεν έχει διερευνηθεί επαρκώς είναι η λήψη τμημάτων video από περισσότερους από έναν servers, με τρόπο που να εξυπηρετεί τόσο τις ανάγκες του δικτύου όσο και τη βελτίωση της Ποιότητας Εμπειρίας του χρήστη (Quality of Experience, QoE). Η συγκεκριμένη διπλωματική εργασία θα διερευνήσει αυτό το πρόβλημα, προσομοιώνοντας ένα δίκτυο με πολλαπλούς video servers και διάφορους video clients. Στη συνέχεια, θα υλοποιήσει τόσο την δυνατότητα επικοινωνίας peer-to-many στα πλαίσια της προσαρμοστικής ροής video όσο και τον αλγόριθμο επιλογής video server. Όλα αυτά θα διερευνηθούν στο περιβάλλον του Mininet, που είναι ένας δικτυακός εξομοιωτής, για να προσομοιωθεί η τεχνολογία DASH με τη βοήθεια των κόμβων του δικτύου του εξομοιωτή. Αρχικά, το βίντεο χωρίστηκε σε μικρά κομμάτια με τη βοήθεια του εργαλείου ffmpeg και στη συνέχεια, υλοποιήθηκαν πειράματα που ένας πελάτης ζητούσε το βίντεο από έναν server προσωρινής αποθήκευσης (cache server). Αν το συγκεκριμένο τμήμα του βίντεο δεν υπήρχε εκεί, τότε στελνόταν αίτημα από τον server προσωρινής αποθήκευσης σε έναν διακομιστή που περιείχε όλα τα τμήματα του βίντεο (main server). Στα πειράματα αυτά εξετάστηκε και η προστιθέμενη δικτυακή κίνηση, με τελικό συμπέρασμα ότι το περιβάλλον του Mininet προκαλεί αναπόφευκτους περιορισμούς στη περίπτωση της δικτυακής κίνησης, καθώς παρατηρήσαμε πως το κανάλι του server βάσης δεδομένων παρέμενε ανενεργό καθ' όλη τη διάρκεια αιτημάτων από τον server προσωρινής αποθήκευσης, με αποτέλεσμα να δημιουργούνται συνθήκες μη-ρεαλιστικού δικτύου. Γι' αυτόν τον λόγο, προβήκαμε στην υλοποίηση μιας νέας προσέγγισης, εξαλείφοντας το Mininet περιβάλλον και δουλεύοντας πάνω σε νέες τεχνικές προσθήκης δικτυακής κίνησης και τροποποιώντας την επικοινωνία των διακομιστών μεταξύ τους. Με αυτόν τον τρόπο, καταφέραμε να δείξουμε σαφέστερα τους περιορισμούς της προηγούμενης προσέγγισης αλλά και να συμπεράνουμε ότι η ύπαρξη servers προσωρινής αποθήκευσης είναι ένα χρήσιμο εργαλείο υπό όρους αύξησης της ποιότητας εμπειρίας ενός χρήστη. Η γενική τάση που παρατηρήθηκε ήταν ότι με την αύξηση του διαθέσιμου χώρου αποθήκευσης, η ποιότητα αναπαραγωγής του βίντεο ανέβαινε σε κάποιο βαθμό. Ταυτόχρονα όμως, το ποσοστό βελτίωσης αυτό, είναι άρρηκτα δεμένο με τον αλγόριθμο επιλογής κομματιών βίντεο που χρησιμοποιείται. Για ακόμα καλύτερα αποτελέσματα λοιπόν, θεωρείται αναγκαία η εύρεση της χρυσής τομής μεταξύ χωρητικότητας του χώρου προσωρινής αποθήκευσης και αλγορίθμου επιλογής κομματιών.

Στην παρούσα διπλωματική παρουσιάζονται τα εξής κεφάλαια: Στο κεφάλαιο 1 αναφέρεται η ιστορική αναδρομή της τεχνολογίας των δικτύων. Στο κεφάλαιο 2 αναλύεται η τεχνολογία προσαρμοστικής ροής βίντεο μέσω HTTP. Στο κεφάλαιο 3 αναλύονται οι διαφορετικές τεχνικές προσωρινής αποθήκευσης. Στο κεφάλαιο 4 παρουσιάζεται η έννοια της Ποιότητας Εμπειρίας του χρήστη και η συσχέτισή της με πολλούς άλλους παράγοντες. Το κεφάλαιο 5 περιγράφεται αναλυτικά η διαδικασία στησίματος του περιβάλλοντος και τα διάφορα απαραίτητα εργαλεία για την υλοποίησή μας. Το κεφάλαιο 6 αναφέρει τα πειράματα μέσω Mininet, την τοπολογία και όλο το στήσιμο, καθώς και τους λόγους που μας οδήγησαν στην πορεία μιας διαφορετικής προσέγγισης. Στο κεφάλαιο 7 προτείνεται η διαφορετική προσέγγιση και παρουσιάζεται η μεθοδολογία και οι μετρικές. Επίσης, αναλύονται διαγράμματα που εξάχθηκαν από την ανάλυση τω μετρικών. Τέλος, το κεφάλαιο 8 αφορά τα συμπεράσματα και θέματα μελλοντικής έρευνας για βελτίωση της Ποιότητας Εμπειρίας του χρήστη περαιτέρω.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Δίκτυα Επικοινωνιών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ποιότητα Εμπειρίας, Υπηρεσίες βίντεο, Dynamic Adaptive Streaming over HTTP, Mininet, Προσωρινή Αποθήκευση

Θέλουμε να ευχαριστήσουμε για την ανεκτίμητη βοήθεια, και στήριξή τους

τις οικογένειές μας καθώς και τους φίλους μας.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The current MSc thesis was conducted in the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, and more specifically under the support of the Communication Networks Laboratory of the Telecommunications and Signal Processing sector. The thesis started being conducted in August 2020 and ended in April 2021. Professor Lazaros Merakos, Dr. Eirini Liotou and Dr. Dionysis Xenakis – to whom we owe special thanks - were the supervisors of this MSc Thesis. We would also like to thank research associate Nikos Episkopos for his appreciated help and willingness.

1. INTRODUCTION: NETWORK DEVELOPMENT

In the last few decades, all kinds of communication networks have experienced a tremendous change in order to serve in a better way a huge number of users and applications. The cellular wireless Generation (G) generally refers to a change in the nature of the system, speed, technology, and frequency. Each generation has its own specification, techniques and features which differentiate it from the previous one. However, sometimes backward compatibility among the generations is possible.

The first generation(1G) mobile wireless communication network was analog used for voice calls only while the second generation (2G) was the first digital approach which supported text messaging as well. After this was 3G which provided multimedia support along with higher data transmission rates and increased capacity. The fourth generation (4G) provides access to a wide range of telecommunication services including advanced mobile services, along with a support for low to high mobility application and raises the Quality of service (QoS), increases the bandwidth and reduces the cost of resources. The 5G brings forward a real Wireless World Wide Web (WWWW) making the Internet of Things (IoT) reality while 6G is proposed to integrate 5G with satellite networks for global coverage. Finally, 7G is expected to deal with space roaming [1].

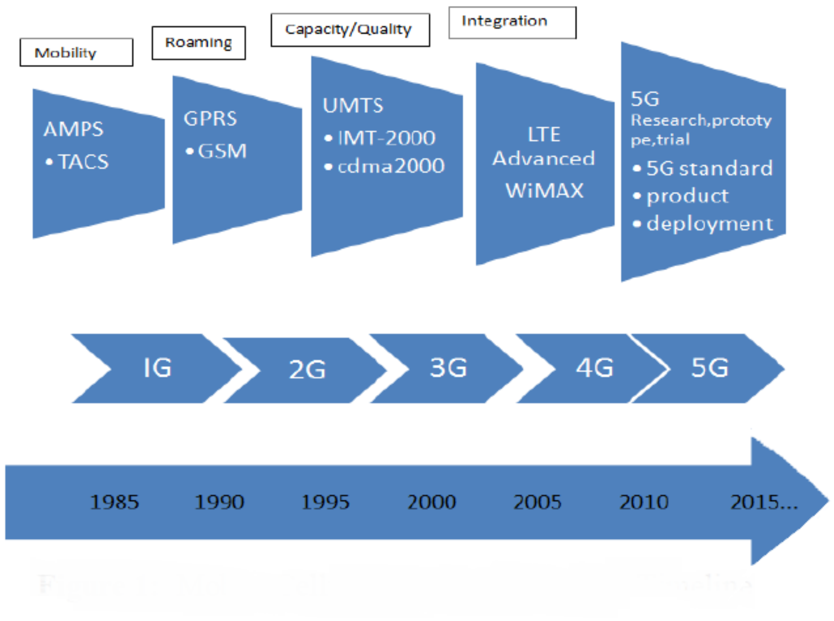


Figure 1: Mobile Cellular Network Evolution Timeline [2]

1.1 1G Mobile Communication System

The first generation (1G) of mobile networks was launched in the 1980s, and it was basically a simple kind of network which provided the users only voice call capabilities and was used by most people for a long time. The whole architecture of this model was based on the idea of cellular networks [3].

Cellular networks are high-speed, high-capacity voice and data communication networks with enhanced multimedia and seamless roaming capabilities for supporting cellular devices such as mobile phones. Their characteristics may differ depending on the size and the shape of the cells [4]. The whole geographical area is divided into cells, and every user can be served by the nearest one. Neighboring cells are not able to share the same level of frequency to avoid interference effects [5].

After the demonstration of the first handheld mobile cell phone by Motorola in 1973 and the first automated commercial network by Nippon Telegraph and Telephone (NTT) in Japan in 1979, the development of the first generation of mobile network began. This first approach consisted of 88 cells with base stations, or radio towers covering all districts of Tokyo and the handover of the call among different cells was supported. In terms of technical characteristics, the 1G systems were analog, they operated in the 150MHz band and calls were automated switching without the need for a human switchboard operator [6].

Even though 1G networks were extremely revolutionary at that time, there were a lot of drawbacks that led to the use of digital signals and the development of the next generation of mobile networks. For example, the full analog mode of communication that 1G suggested, resulted in inefficient use of spectrum and loss of a huge amount of resources. Furthermore, the coverage was extremely limited, the security was totally weak, without any type of encryption and the roaming between different operators was not supported, making the communication between people significantly difficult [6].

Finally, the voice quality was poor because in 1G, the data can be carried by only one channel from source to destination. This means that the two callers are not able to hear each other simultaneously since the number of calls was limited [7].

1.2 2G Mobile Communication System

The first generation (1G) was created to make calls and not much else. Even though it was a breakthrough, the substantial limitations kept the technology from spreading. The appearance of the 2G network was an essential stage in the revolution of the communication industry [8].

Second-generation (2G) mobile systems were introduced in the end of the 1980s. Compared to first-generation systems, second-generation (2G) systems use digital multiple access technology, such as TDMA (time division multiple access) and CDMA (code division multiple access). Consequently, compared with first-generation systems, higher spectrum efficiency, better data services, and more advanced roaming were offered by 2G systems. In the United States, there were three lines of development in second-generation digital cellular systems. The first digital system, introduced in 1991, was the IS-54 (North America TDMA Digital Cellular), of which a new version supporting additional services (IS-136) was introduced in 1996. Meanwhile, IS-95 (CDMA One) was deployed in 1993. 2G communication is generally associated with the global system for mobile (GSM) services [9].

The technology behind the Global System for Mobile communication (GSM) was originally designed for operation in the 900 MHz band, it was soon adapted for 1800MHz. The introduction of GSM into North America meant further adaptation to the 800 and 1900 MHz bands. Over the years, the versatility of GSM has resulted in the specifications being adapted to many more frequency bands.

GSM has a channel spacing of 200kHz and was designed principally for voice telephony, but a range of bearer services was defined (a subset of those available for fixed line Integrated Services Digital Networks, ISDN), allowing circuit-switched data connections at up to 9600 bits/s. During the next few years, the General Packet Radio Service (GPRS) was developed to allow aggregation of several carriers for higher speed, packet-switched applications such as always-on Internet access. The first commercial GPRS offerings were introduced in the early 2000s [10].

In order that the GSM system operates together as a complete system, the overall network architecture brings together a series of data network identities, each with several elements. The GSM network architecture is defined in the GSM specifications and it can be grouped into four main areas [11]:

- Network and Switching Subsystem (NSS)
- Base-Station Subsystem (BSS)
- Mobile Station (MS)
- Operation and Support Subsystem (OSS)

Meanwhile, investigations had been continuing with a view to increasing the intrinsic bit rate of the GSM technology via novel modulation techniques [10]. Before the 3G network came around, interim 2.5G and 2.75G standards became available. Even though they aren’t officially defined as wireless standards, they assisted with marketing efforts for new phones. 2.5G provided a new packet-switching technique, which offered better possibilities than the 2G technology. 2.75G provided an opportunity for a 300% speed increase. In the United States, AT&T became the first GSM network to use 2.75G with EDGE (Enhanced Data-rates for Global Evolution), which offers an almost three-fold data rate increase in the same bandwidth.

The combination of GPRS and EDGE brings system capabilities into the range covered by the International Telecommunication Union IMT-2000 (third generation) concept. It’s worth noting that even though 2G was surpassed by 3G, 4G, and 5G technologies, it’s still widely used around the world. For example, the European provider Vodafone is planning to support 2G until 2025. However, some companies in the USA have shut down their 2G networks [12].

Overall, the main features of 2G and 2.5G are data speed up to 64kbps, digital signals, services such as text messages, picture messages and MMS (Multimedia message), better network quality and capacity [13].

1.3 3G Mobile Communication System

The need for faster speed, global compatibility and multimedia services has led to the development of 3G systems. In an effort to consolidate existing incompatible mobile environments into a seamless global network, ITU adopted a family of radio access methods at its Radiocommunication Assembly in Istanbul in early May 2000. Known as International Mobile Telecommunications-2000 (IMT-2000), this global standard was realized after years of collaborative work between ITU and the global cellular community. At the end of May 2000, the World Radiocommunication Conference (also held in Istanbul) identified additional frequency bands for 3G (IMT-2000) use. IMT-2000 consists of five different radio access methods: W-CDMA (Wideband Code Division Multiple Access), CDMA20001X, TD-SCDMA, EDGE (Enhanced Data Rates for GSM Evolution) and DECT (Digitally Enhanced Cordless Telecommunications) [13].

The evolutionary path from 2G to 3G has been mapped out for existing networks (figure 2). Migration differs depending on the existing 2G network. In general, W-CDMA would require a brand-new network to be installed whereas CDMA2000 1X requires less investment as an upgrade from existing second-generation CDMA networks. Among the five radio access technologies approved as IMT-2000, W-CDMA and CDMA2000 1X have gained the most support from regulators, mobile network operators and equipment manufacturers [14].

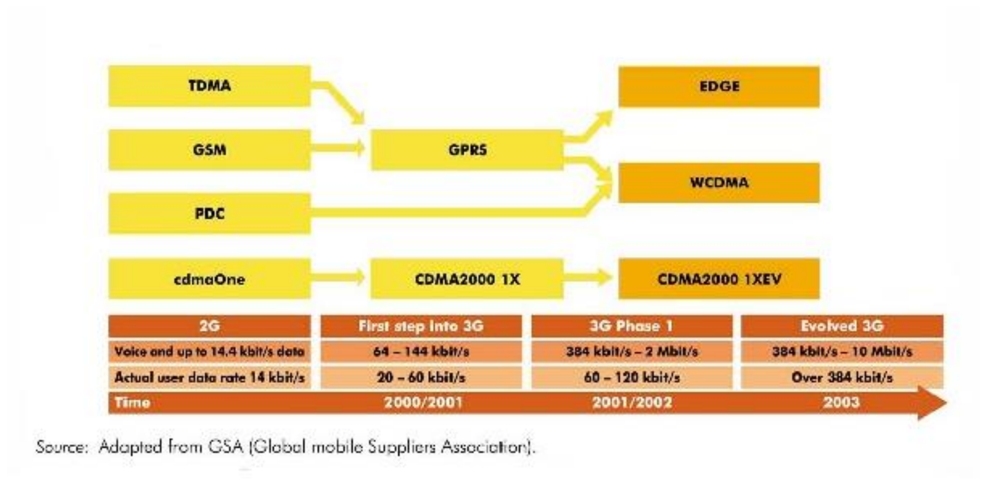


Figure 2: From 2G to 3G [14]

Among the terrestrial systems of the IMT-2000 family the most successful 3rd generation mobile cellular technology was developed by the 3rd Generation Partnership Project (3GPP) under the name Universal Mobile Telecommunications System (UMTS). There are/were also other names used for this system: Universal Terrestrial Radio Access (UTRA), Freedom of Mobile Multimedia Access (FOMA), 3GSM, etc. UMTS is the term Europe uses to refer to 3G networks [14], while CDMA2000 is the name of American 3G variant. Also, the IMT-2000 has accepted the new 3G standard from China, i.e. TD-SCDMA, is the air-interface technology for UMTS.

UMTS is offering greater spectral efficiency than GSM and has 2 modes:

1) an FDD mode (Frequency Division Duplex) where uplink (user equipment UE => UTRA network = UTRAN) and downlink (UTRAN => UE) communication are separated in the frequency domain via different frequency bands; this mode is also called Wideband CDMA (WCDMA) or Direct Spread CDMA.
2) a TDD mode (Time Division Duplex) where uplink (UE => UTRAN) and downlink (UTRAN => UE) communication are separated in the frequency domain via different time slots; note: The 3.84Mchip/s option was also called TD-CDMA or HCR (high chip rate) TDD.

Both modes use direct sequence CDMA (code division multiple access) to separate the different users, i.e. each symbol of one user is multiplied by a user specific spreading code. With this CDMA technique multiple users can transmit in the same (larger) band and the decoder, knowing the user's spreading code, can pick up the data of this user. The data of other users appears as noise in this decoding process.

Using a wide frequency band makes the system inherently resistant to many of the aspects of radio communication which plague narrow band systems, such as burst-y noise, multipath reflections, and other interfering transmissions [15].

The technology was improved to allow data up to 14 Mbps and more using packet switching. It uses Wide Band Wireless Network with which clarity is increased. It also offers data services, access to television/video, and new services like Global Roaming. It operates at a range of 2100MHz and has a bandwidth of 15-20MHz used for high-speed Internet service, video chatting. The main features of 3G are speed of 2 Mbps, introduction of smartphones, increased bandwidth and data transfer rates to accommodate web-based applications, audio and video files, faster communication, large email messages, high speed web/more security/video conferencing/3D gaming, large capacities and broadband capabilities and TV streaming/mobile TV/Phone calls [13].

Even though the 3G technology includes many upgrades in matters of security compared to 2G, the main goal for the domination of a global prototype was not achieved, apart from America where three incompatible systems were developed. The data transmission rates were not as expected, as voice cannot be transferred through the IP. Unfortunately, the 3G network did not respond to the promises of its manufacturers. The technology of CDMA and the architecture of UMTS was abandoned because there was a need for higher data rates, bigger spectral efficiency, improved packet switching system, high quality services and lower cost of infrastructure. However, this was the initial and trial version of the 3G experience.

1.4 4G Mobile Communication System

UTRA (Universal Terrestrial Radio Access) as a 3rd generation system, with the enhancements provided by High-Speed Packet Access (HSPA), for both downlink and uplink, will remain highly competitive for several years [16]. During the early 2000's, there had been several talks and proposals for 4G. By that time, it came to a mutual decision that the next generation should be more of an "evolution" instead of a revolution as in the case of the previous cellular networks [17]. That is why 4G is based on the previous 3G network, only enhanced with a new technology called Long Term Evolution or LTE. So, the industry that has developed the 3GPP technologies, launched the project in Dec. 2004 of LTE to study requirements for a new air interface called Evolved UTRA (E-UTRA). The terms LTE and E-UTRA are synonymous, however the radio specifications talk rather about E-UTRA [16].

LTE is based on the GSM/EDGE and UMTS/HSPA technologies. It increases the capacity and speed using a different radio interface together with core network improvements. It is the upgrade path for carriers with both GSM/UMTS networks and CDMA2000 networks. The different LTE frequencies and bands used in different countries mean that only multi-band phones are able to use LTE in all countries where it is supported. The standard is developed by the 3GPP and is specified in its Release 8 document series, with minor enhancements described in Release 9. LTE is sometimes known as 3.95G and has been marketed both as "4G LTE" but it does not meet the technical criteria of a 4G wireless service, as specified in the 3GPP Release 8 and 9 document series for LTE Advanced. The requirements were originally set forth by the ITU-R organization in the IMT Advanced specification. However, due to marketing pressures and the significant advancements that WiMAX, Evolved High Speed Packet Access, and LTE bring to the original 3G technologies, ITU later decided that LTE together with the aforementioned technologies can be called 4G technologies. The LTE Advanced standard formally satisfies the ITU-R requirements to be considered IMT-Advanced [18].

1.4.1 LTE Advanced

LTE Advanced is a mobile communication standard and a major enhancement of the LTE standard. It was formally submitted as a candidate 4G to ITU-T in late 2009 as meeting the requirements of the IMT-Advanced standard and was standardized by the 3GPP in March 2011 as 3GPP Release 10 [19].

LTE-A is the latest and most advanced technology for LTE. From now on all subsequent improvements will be called LTE-A. LTE-A speeds reach 1 Gbps downlink and 500 Mbps uplink and have additional LTE-like mechanisms such as carrier aggregation and relaying. According to 3GPP in the release for LTE-A specs we can have downlink speeds of 1 Gbps and uplink speeds of 500 Mbps. Additionally, less than 5ms Round Trip Time is also reachable.

1.4.2 LTE Advanced Characteristics

The target figures for data throughput in the downlink is 1 Gbps for 4G LTE Advanced. Even with the improvements in spectral efficiency it is not possible to provide the required headline data throughput rates within the maximum 20 MHz channel. The only way to achieve the higher data rates is to increase the overall bandwidth used. IMT Advanced sets the upper limit at 100 MHz, but with an expectation of 40 MHz being used for minimum performance. For the future it is possible the top limit of 100 MHz could be extended.

It is well understood that spectrum is a valuable commodity, and it takes time to reassign it from one use to another. So, the cost of forcing users to move is huge as new equipment needs to be bought. Accordingly, as sections of the spectrum fall out of use, they can be reassigned. This leads to significant levels of fragmentation.

1) Carrier Aggregation:

   To an LTE terminal, each component carrier appears as an LTE carrier, while an LTE-Advanced terminal can exploit the total aggregated bandwidth. There are a number of ways in which LTE carriers can be aggregated:

   - Intra-band: This form of carrier aggregation uses a single band. There are two main formats for this type of carrier aggregation:
     - Contiguous: The Intra-band contiguous carrier aggregation is the easiest form of LTE carrier aggregation to implement. Here the carriers are adjacent to each other.
     - Non-contiguous: Non-contiguous intra-band carrier aggregation is somewhat more complicated than the instance where adjacent carriers are used. No longer can the multi-carrier signal be treated as a single signal and therefore two transceivers are required. This adds significant complexity, particularly to the UE where space, power and cost are prime considerations.
   - Inter-band non-contiguous: This form of carrier aggregation uses different bands. It will be of particular use because of the fragmentation of bands – some of which are only 10 MHz wide. For the UE it requires the use of multiple transceivers within the single item, with the usual impact on cost, performance, and power. In addition to this there are also additional complexities resulting from the requirements to reduce intermodulation and cross modulation from the two transceivers.

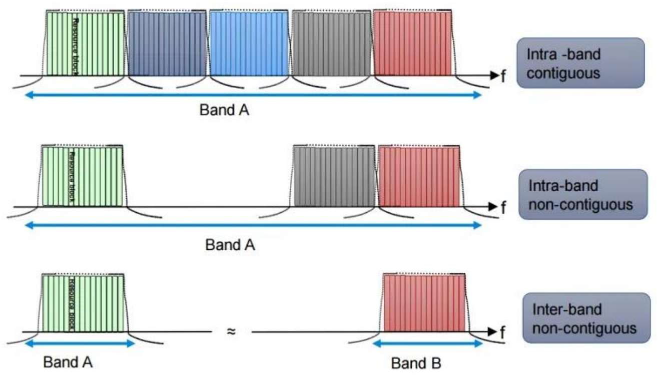


Figure 3: Carrier Components

The current standards allow for up to five 20 MHz carriers to be aggregated, although in practice two or three is likely to be the practical limit. These aggregated carriers can be transmitted in parallel to or from the same terminal, thereby enabling a much higher throughput to be obtained [20].

2) MIMO, Multiple Input Multiple Output – or spatial multiplexing

Multiple Input Multiple Output antennas, which stands for MIMO, is a basic technique in which multiple antennas are established. MIMO is used to increase the overall bitrate through transmission of two (or more) different data streams on two (or more) different antennas - using the same resources in both frequency and time, separated only through the use of different reference signals - to be received by two or more antennas [21]. In LTE-A base stations have up to 8 antennas and terminals, and up to 4 antennas. As far as how multiple antennas can be used, there are three possibilities:

- Cooperative MIMO: Let's assume that a terminal is on the border of 3 different base stations, and each of them uses 2 of its 8 antennas to transmit the signal. The terminal receives the same weak signal 6 times because it is not very close to one of all the 3 base stations. In LTE-A, the terminal is capable of exploiting all these signal portions so as to compose a high-quality signal.
- MU-MIMO (Multiple User MIMO): In this case a base station is able to send 2 different signals to 2 users in order to provide more information to both of them.
- SU-MIMO (Single User MIMO): In this case a base station can send either the same signal 4 times from its 4 antennas in order to strengthen it (so it will have fewer errors and faster speeds) or send 4 completely different signals where the one will be a continuation of the other.

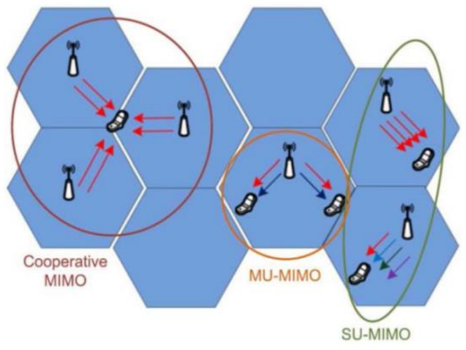


Figure 4:Types of MIMO base stations

3) Heterogeneous Networks and eICIC (enhanced Inter-Cell Interference Coordination)

Generally, there are many interference problems. In order to address this problem, eICIC (enhanced Inter-Cell Interference Coordination) is used, where base stations can communicate with each other and coordinate:

- **In Time**: It is the most common method used. In this method there are the Almost Blank Subframes (ABSFs), as they are called, which leave the macro cell deliberately empty and inform the femtocell to send to them exclusively so that there is no interference.
- **In Frequency**: In this case the stations understand the user who is having the problem and give him another frequency (in the same bandwidth) to communicate with no problem.

- **In Power**: In this case the base stations communicate after the problem and the femtocell emission power is reduced which either way has always the best signal and as a result its power loss does not cause any problems.

4) Coordinated Multi-Point transmission and reception (CoMP)

Coordinated multipoint transmission and reception refers to a wide range of techniques that enable dynamic coordination or transmission and reception with multiple geographically separated evolved base stations, called eNodeB (eNB). Its aim is to enhance the overall system performance, utilize the resources more effectively and improve the end user service quality.

4G LTE CoMP requires close coordination between several geographically separated eNBs. They dynamically coordinate to provide joint scheduling and transmissions as well as proving joint processing of the received signals. In this way a UE at the edge of a cell can be served by two or more eNBs to improve signals' reception / transmission and increase throughput particularly under cell edge conditions.

In essence, 4G LTE CoMP, Coordinated Multipoint falls into two major categories [22]:

- **Joint processing**: Joint processing occurs where there is coordination between multiple entities - base stations - that are simultaneously transmitting or receiving to or from UEs.
- **Coordinated scheduling or beamforming**: This often referred to as CS/CB (coordinated scheduling / coordinated beamforming) is a form of coordination where a UE is transmitting with a single transmission or reception point - base station. However, the communication is made with an exchange of control among several coordinated entities.

5) Relay nodes (Broadcasting)

In LTE-Advanced, the possibility for efficient heterogeneous network planning – i.e. a mix of large and small cells - is increased by introduction of Relay Nodes (RNs). The Relay Nodes are low power base stations that will provide enhanced coverage and capacity at cell edges, and hot-spot areas and it can also be used to connect to remote areas without fiber connection. The Relay Node is connected to the Donor eNB (DeNB) via a radio interface, Un, which is a modification of the E-UTRAN air interface Uu. Hence, in the Donor cell the radio resources are shared between UEs served directly by the DeNB and the Relay Nodes. When the Uu and Un use different frequencies the Relay Node is referred to as a Type 1a RN, for Type 1 RN Uu and Un utilize the same frequencies (figure 5). In the latter case there is a high risk for self-interference in the Relay Node, when receiving on Uu and transmitting on Un at the same time (or vice versa). This can be avoided through time sharing between Uu and Un or having different locations of the transmitter and receiver. The RN will to a large extent support the same functionalities as the eNB – however the DeNB will be responsible for MME selection.

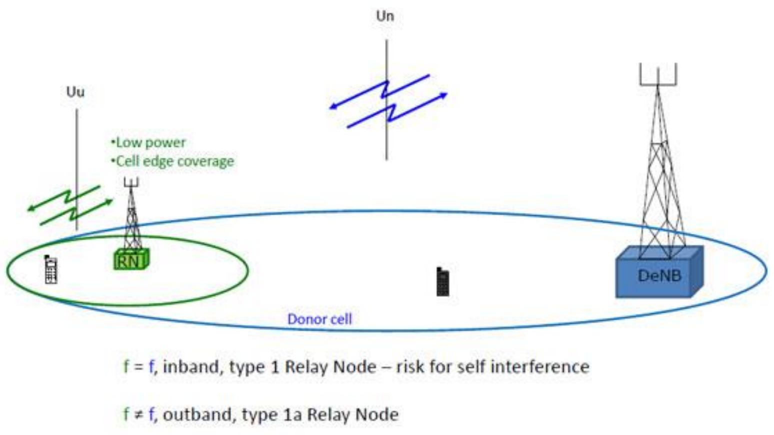

Un
Uu
•Low power
•Cell edge coverage
RN
DeNB
Donor cell
f = f, inband, type 1 Relay Node – risk for self interference
f ≠ f, outband, type 1a Relay Node

Figure 5:A Relay Node paradigm [21]

1.4.3 LTE Architecture

The architecture of LTE is comprised by 3 main components:

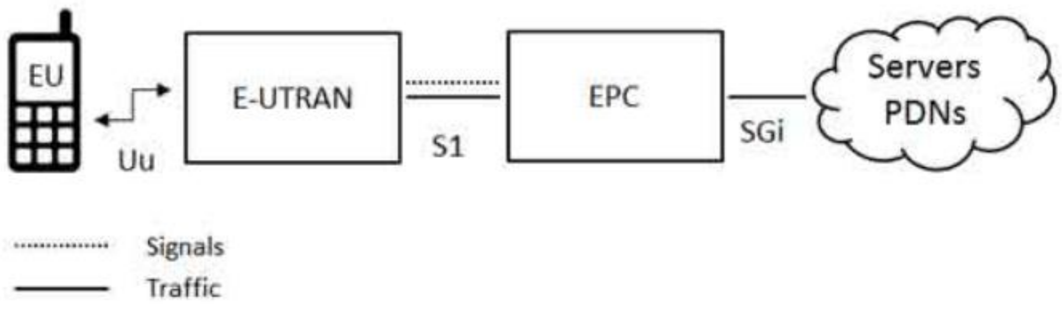


Figure 6: Basic LTE Components

1) the User Equipment- UE, which internal architecture is identical to the one used by UMTS and GSM.
2) the Evolved UMTST Terrestrial Radio Access Network (E-UTRAN), which handles the radio communications between the mobile and the evolved packet core . It only has one component, the eNB. Each eNB is a base station that controls the mobiles in one or more cells. The base station that is communicating with a mobile is known as its serving eNB. Each eNB connects with the EPC by means of the S1 interface and it can also be connected to nearby base stations by the X2 interface, which is mainly used for signaling and packet forwarding during handover.
3) the Evolved Packet Core (EPC) which belongs to the core network. The EPC is comprised by several components like the PDN Gateway which is responsible for communication with the outside world, and the Mobility Management Entity (MME) which controls the high-level operation of the mobile in terms of signaling messages. A brief presentation of all the EPC’s components is shown in the below figure.

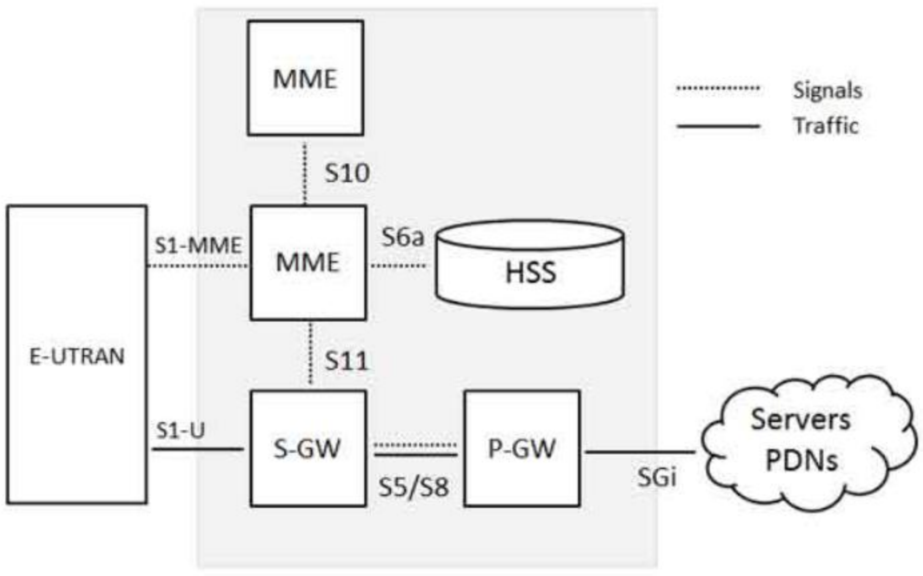


Figure 7: The Evolved Packet Core (EPC) components [23]

1.4.4 OFDM/OFDMA/SC-FDMA

In OFDM (Orthogonal Frequency Division Multiplexing) systems, the original bandwidth is subdivided into multiple subcarriers. Each of these subcarriers can then be individually modulated. Typically, in OFDM systems, we can have hundreds of subcarriers with a content spacing between them (15KHz on the LTE case). Since the multiple subcarriers in OFDM are transmitted in parallel, it’s possible for each one to transmit with a lower symbol rate. That improves robustness on the technology for mobile propagation conditions.

The chain to generate an OFDM signal starts by analyzing the symbols that need to be transmitted, after they are modulated (in LTE the modulation can be QPSK, 16QAM, 64QAM). Then they are used as input bands for an Inverse Fast Fourier Transform (IFFT) operation. This operation produces OFDM symbols, which will be transmitted. Notice that a conversion from the frequency to the time domain was made when the IFFT was used. Before the transmission, however, a cyclic prefix is included in the OFDM symbols in order to avoid inter-symbol interference. This cyclic prefix in LTE has 5.2us on the first symbol, 4.7us for the rest of them and an extended cyclic prefix for larger cells. On the receiving side of the OFDMA system we should expect an FFT operation that will convert the symbol to the frequency domain again.

The main difference between an OFDM and an OFDMA (Orthogonal Frequency Division Multiple Access) system is the fact that in the OFDM the user is allocated on the time domain only while using an OFDMA system the user would be allocated by both time and frequency. This is useful for LTE since it makes it possible to exploit frequency dependence scheduling. For instance, it would be possible to exploit the fact that user 1 might have a better radio link quality on some specific bandwidth area of the available bandwidth.

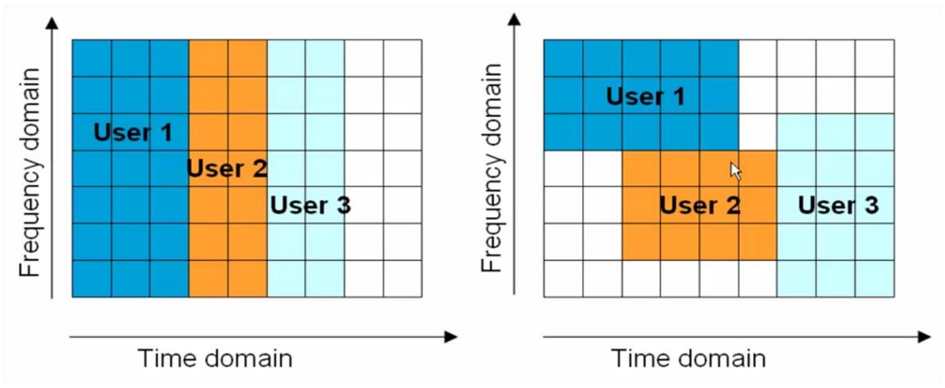


Figure 8: OFDM vs OFDMA

It is not possible to use OFDMA on the uplink since, as told before, it presents a high Peak-to-average Power Ratio. SC-FDMA (Single Carrier FDMA) presents the benefit of a single carrier multiplexing of having a lower Peak-to-average Power Ratio. On SC-FDMA before applying the IFFT the symbols are pre coded by a DFT (Discrete Fourier Transform). This way each subcarrier after the IFFT will contain part of each symbol. Looking at the figure below, it is possible to see the difference between SC-FDMA and OFDMA. Also, it is possible to notice that the inter-symbol interference will be reduced since all subcarriers on a period of time represent the same symbol.

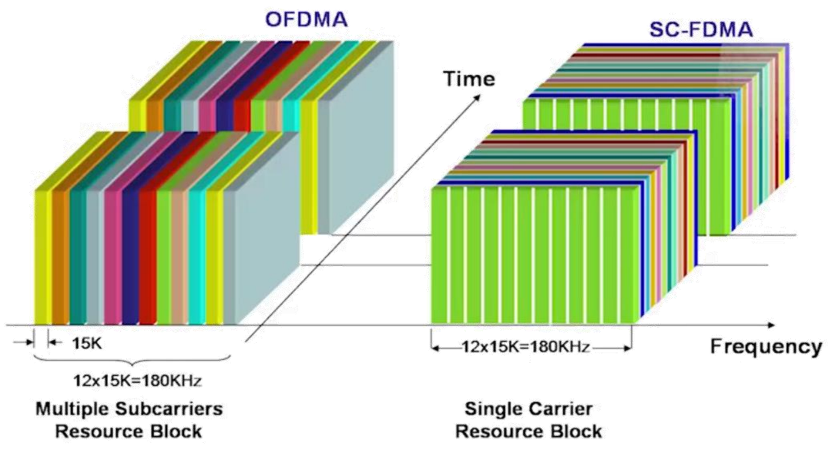

OFDMA
SC-FDMA
Time
15K
12x15K=180KHz
Multiple Subcarriers Resource Block
12x15K=180KHz
Frequency
Single Carrier Resource Block

Figure 9: Simplified LTE architecture [24]

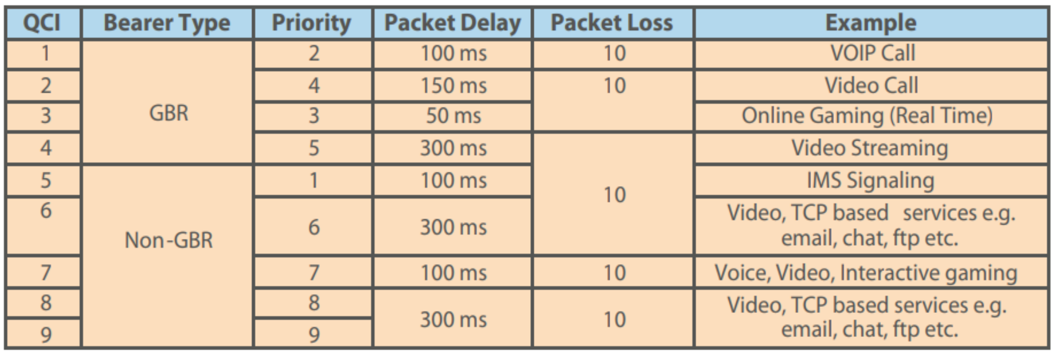
1.4.5 Quality of Service (QoS) in LTE

LTE Quality of Service (QoS) has become an important part of network planning & design when deploying 4G/LTE fixed broadband wireless for data & voice services. There are subscribers who use LTE services for critical operations (e.g. voice calls, bank transactions, hospital operations), and there are subscribers who just want to enjoy premium Internet & applications experiences. LTE was designed to meet these increased data and application demands with reliable connections and low cost of deployment. As a result, the QoS was a matter of high priority.

The QoS on LTE is provided through the bearers. “Bearer” means “Carrier” which takes and carries the information from one point to another point. EPS (Evolved Packet System) bearer, provides user plane IP connectivity between UE and the Packet Data Network Gateway (PGW), in a virtual way. An initial EPS bearer is established when the UE registers with the LTE network using an attach procedure which is called “EPS Bearer Activation”. This bearer is known as the default EPS bearer. There are also other bearers which are called dedicated. A dedicated bearer is established when there is a need to provide better QoS to a specific service like a Voice over Internet Protocol (VoIP) or a video [25].

Bearers can be characterized based on the resource type as GBRs (Guaranteed Bit Rate) or Non-GBRs. For an EPS bearer, with a GBR resource type the bandwidth of the bearer is guaranteed. Only a dedicated EPS bearer can be a GBR type bearer and no default EPS bearer can be GBR type. On the other hand, having a non-GBR resource type means that the bearer is a best effort type bearer and its bandwidth is not guaranteed. A default EPS bearer is always a Non-GBR bearer, whereas a dedicated EPS bearer can be either GBR or non-GBR.

From QoS perspective, there are 9 different classes of bearers. Each class is characterized firstly by the QCI (QoS Class Identifier) parameter which is an integer from 1 to 9 that indicates nine different QoS performance characteristics of each IP packet. For example, QCI 1 and 9 are defined as follows:



QCI	Bearer Type	Priority	Packet Delay	Packet Loss	Example
1	GBR	2	100 ms	10	VOIP Call
2		4	150 ms	10	Video Call
3		3	50 ms		Online Gaming (Real Time)
4		5	300 ms	10	Video Streaming
5	Non-GBR	1	100 ms		IMS Signaling
6		6	300 ms		Video, TCP based services e.g. email, chat, ftp etc.
7		7	100 ms	10	Voice, Video, Interactive gaming
8		8	300 ms	10	Video, TCP based services e.g. email, chat, ftp etc.
9		9			

Figure 10: The QCI identifiers definition

Each class is also characterized by the ARP (Allocation and Retention Priority) indicator as well, which defines the behavior of the network when the traffic is huge. ARP is an integer value ranging from 1 to 15, with the 1 being the highest level of priority in high congestion situations. [26]

1.4.6 LTE Basic Parameters

Table 1: LTE operational parameters

Parameters	Description
Frequency range	UMTS FDD bands and TDD bands
Duplexing	FDD, TDD, half-duplex FDD
Mobility	350 km/h
Channel Bandwidth (MHz)	1.4, 3, 5, 10, 15, 20
Modulation Schemes	QPSK, 16QAM, 64QAM
Multiple Access Schemes	UL: SC-FDMA supports 50Mbps+ (20MHz spectrum) DL: OFDM supports 100Mbps+ (20MHz spectrum)

Multi-Antenna Technology	UL: Multi-user collaborative MIMO DL: Tx AA, spatial multiplexing, CDD, max 4x4 array
Peak data rate in LTE	UL: 75 Mbps- 20MHz BW DL: 150 to 300 Mbps- 20MHz BW
Coverage	5 - 100km with slight degradation after 30km
Latency	Less than 10 ms

1.5 5G Mobile Communication System

With 4G approaching its limits, the 5G era is already upon us. Services are already available in some regions all around the world, and several rollouts have been planned globally in the years ahead. As we already know, the fifth generation of mobile communication systems isn't just a simple upgrade like its predecessors, but it changes the way devices connect to the network or to one another. As seen in Figure 11, this is a comparison of key capabilities of IMT-Advanced (4th generation) with IMT-2020 (5th generation) according to ITU-R M.2083.

The rapid rise of the mobile data traffic, mostly due to video streaming, and the growing number of mobiles and increased data forced the networks to increase the energy efficiency and made the 5G networks necessary. Furthermore, the Internet of Things (IoT), which purpose is to offer new services and applications in such a way to bridge the physical world with the virtual one, requires networks that must handle billion more devices efficiently [27].

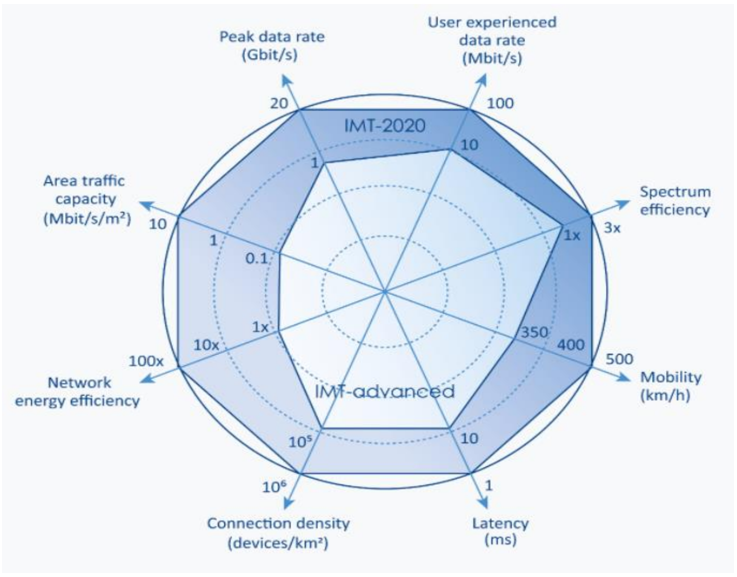


Figure 11: Comparison of key capabilities IMT-advanced and IMT-2020

The 5G communication system managed to overcome some significant challenges in terms of network features that no other type of network was able to do so:

1) Higher network capacity: Network capacity is strongly connected with the number of devices that a network is able to support. 5G networks have upped the devices that can be supported to 1.000.000, while 4G networks remain at 100.000 per km^2. That means that the need of capacity is greater when we move from 4G to 5G. This is because in IoT applications, everything will be connected even shoes. According to communication principles, the shorter the frequency, the larger the bandwidth. So, in order to deliver greater capacity, 5G will use new high bands known as millimeter waves (between 30GHz and 300GHz) as well as underutilized UHF frequencies between 300 MHz and 3 GHz. And to maintain a balance with coverage and capacity, it will leverage the mid-bands from 4G networks (1 GHz to 6 GHz).
2) Higher data transmission rate up to 1 Gbps by using 25-QAM Modulation and coding schemes. However, 5G download speed and other features may differ widely by area. For instance, average 5G speed measures done in 2019 range from 220 Mbps in Las Vegas to 950 Mbps in Minneapolis and Providence.
3) Low latency: Latency is the time it is needed for all the appropriate connections to be made (lag time). With the 5G approach, latency will be almost nothing, lower than a millisecond. This would be the key factor for a variety of breakthroughs such as autonomous vehicles or remote surgeries.
4) Sustainable costs for providers: The operators would be able to offer new services and applications, with the minimum cost.
5) Consistent Quality of Experience (QoE): The QoE (e.g., the measure of the delight or annoyance of a customer's experiences with a service) will increase by providing low latency, high bandwidth, and network traffic optimization techniques [28].

Perhaps the key ingredient enabling the full potential of 5G architecture to be realized is network slicing. This technology adds an extra dimension to the Network Function Virtualization (NFV) domain by allowing multiple logical networks to simultaneously run on top of a shared physical network infrastructure. This becomes integral to 5G architecture by creating end-to-end virtual networks that include both networking and storage functions.

As a result, the operators can effectively manage diverse 5G use cases with differing throughput, latency and availability demands by partitioning network resources to multiple users. Network slicing becomes extremely useful for applications like the IoT where the number of users may be extremely high, but the overall bandwidth demand is low. Each 5G vertical will have its own requirements, so network slicing becomes an important design consideration for 5G network architecture [29].

As seen in figure 12, this is considered the general architecture of 5G networks:

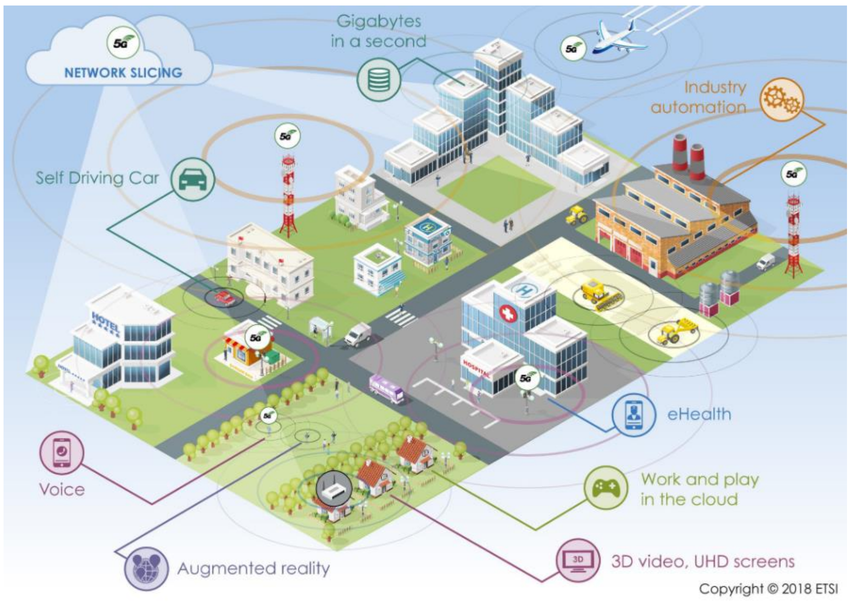


Figure 12: A 5G network architecture [27]

2. HTTP ADAPTIVE STREAMING

2.1 Introduction

Over the last few years, the delivery of videos, music or other multimedia content became more crucial than ever. Nowadays, it is considered that video streaming applications are responsible for more than half of the Internet traffic. To enable video streaming over the Internet in a best-effort way, the concept of HTTP Adaptive Streaming (HAS) was introduced. As shown in figure 13, video content is encoded at different quality levels.

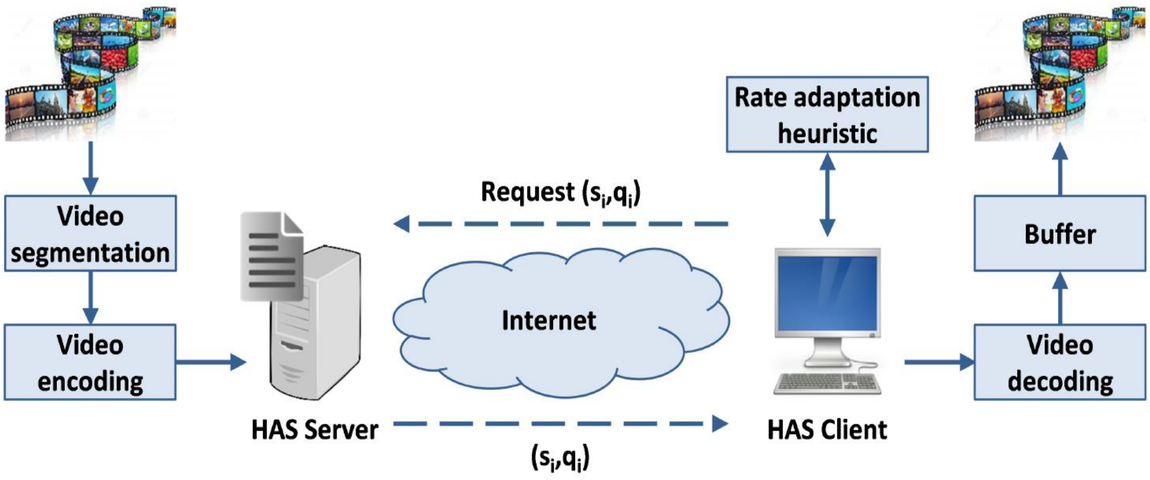


Figure 13: The concept of HTTP Adaptive Streaming

Each quality level is determined by its corresponding average bitrate. In order to adaptively stream media, we need to split the media up into chunks, so the content is divided in segments that have a typical duration (most commonly one to ten seconds). Each segment can be decoded independently of other segments. A HAS client initiates a new session by downloading a manifest file. This manifest file is used to hold the information and description of the various streams and the bandwidths they are associated with. Based on the network conditions and the current buffer-filling level, the Rate Determination Algorithm (RDA) in the HAS client determines the quality for the next segment 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.

The main advantage of HAS over progressive download and common real-time streaming is that it is able to adapt any video quality according to the available bandwidth so as to avoid video stallings. Consequently, HAS facilitates video streaming over a best-effort network. In addition, HTTP-based video streams can easily traverse firewalls and reuse the already deployed HTTP infrastructure such as HTTP servers, HTTP proxies, and Content Delivery Network (CDN) nodes. Because of these advantages, major players such as Microsoft, Apple, Adobe, and Netflix massively adopted the adaptive streaming paradigm [30].

2.2 Definition

Adaptive streaming or progressive streaming is a new streaming approach that it is designed to deliver multimedia to the user in the most efficient way possible and in the highest available quality for each specific user. The term “bitrate” is often used to describe the speed of the Internet connection, which is why adaptive streaming is also called adaptive bitrate streaming. A fast Internet connection has a higher bitrate than a slow Internet connection. Bitrate is literally the rate at which bits of data travel to the users' machine. To explain adaptive streaming as simply as possible it is best to start by explaining what adaptive streaming is not.

A progressive video stream is simply one single video file, usually an .mp4 or any other format, being streamed over the Internet. The progressive video can be stretched and squashed to fit different screen sizes, but regardless of the device playing it, the video file will not change at all. The figure 14 below shows the journey of a 720p video from a server to a web client, pointing that the file will remain the same. [31]

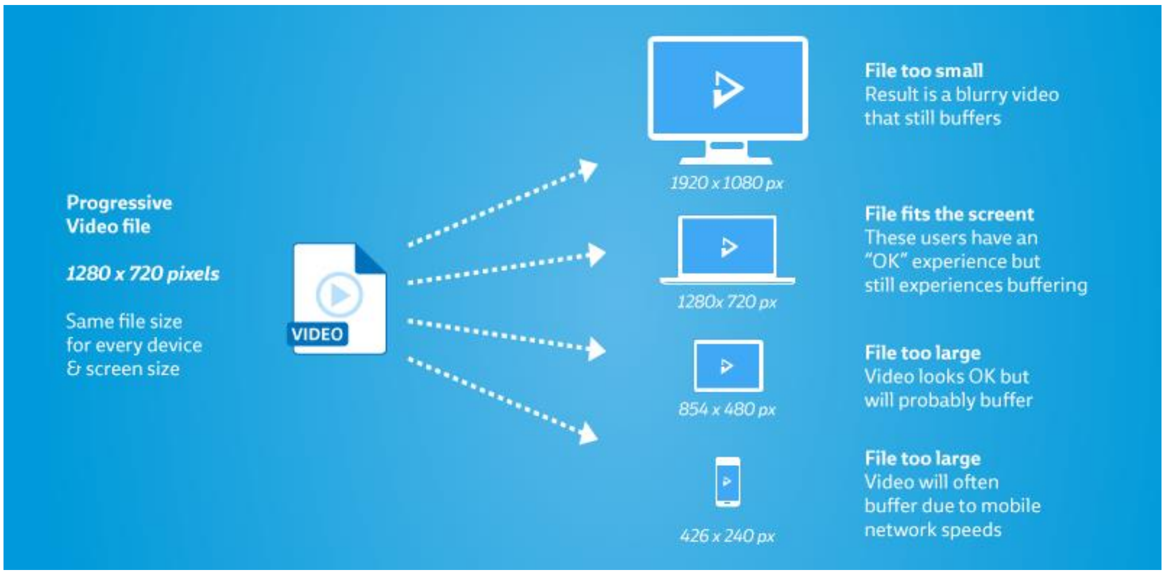

Progressive Video file
1280 x 720 pixels
Same file size for every device & screen size
VIDEO
1920 x 1080 px
File too small
Result is a blurry video that still buffers
1280x 720 px
File fits the screent
These users have an "OK" experience but still experiences buffering
854 x 480 px
File too large
Video looks OK but will probably buffer
426 x 240 px
File too large
Video will often buffer due to mobile network speeds

Figure 14: HTTP Adaptive Streaming (HAS)

2.3 How adaptive streaming solves the problems

There are two immediate problems when progressive streaming is used. The first is quality. Obviously, a video that is only 1280 x 720 will never play at correct quality levels on a screen that is 1920 x 1080px. It will be stretched and pixelated.

A solution to the quality problem is not very complex. Adaptive streaming allows the video provider to create a different video for each of the resolutions that he or she wishes to target. The figure 15 below very simply shows the way that we can stream any video file to fit specific screen sizes, ensuring that the viewer always receives a video that will not look deprecated.

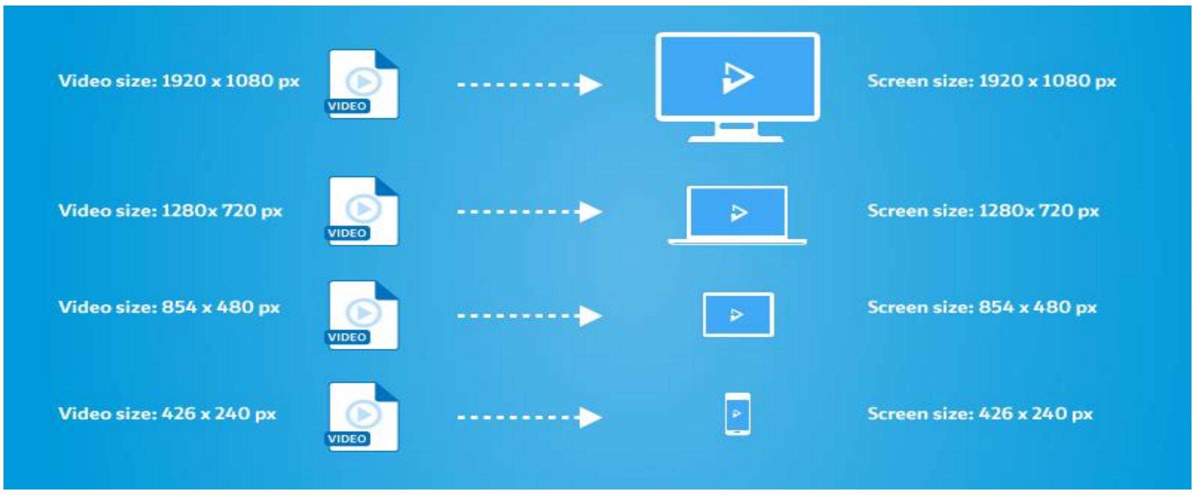

Video size: 1920 x 1080 px
VIDEO
Screen size: 1920 x 1080 px
Video size: 1280x 720 px
VIDEO
Screen size: 1280x 720 px
Video size: 854 x 480 px
VIDEO
Screen size: 854 x 480 px
Video size: 426 x 240 px
VIDEO
Screen size: 426 x 240 px

Figure 15: Video fitting to any available screen size

The second problem is buffering which is an uncomfortable situation for all the users when the video pauses sequentially. Buffering happens commonly when there is an unstable Internet connection. Consequently, the video has to pause several times because it cannot be downloaded quickly enough, in order to wait for more data and then start downloading again. Most videos play at 24 frames per second, so the Internet connection needs to download at least 24 frames every second to avoid buffering. This problem is very common, especially on mobile devices, where the connection can vary greatly depending on the user's location and can cause bad user experience.

Adaptive streaming can resolve this second buffering problem by "adapting" to the speed of the user's Internet connection. A small video can be downloaded faster than a large video, so if a user has a slow Internet connection, an adaptive video stream will switch to a smaller video file size to keep the video playing. From a user's perspective, it is preferable to watch a few minutes of lower quality video in order to avoid buffering, than to sit and watch a spinning icon until the stream catches up [31].

2.4 Dynamic Adaptive Streaming (DASH)

MPEG (Motion Picture Expert Group) issued a Call for Proposal for an HTTP streaming standard in April 2009. In the two years that followed, MPEG developed the specification with participation from many experts and with collaboration from other standard groups, such as the Third Generation Partnership Project (3GPP). More than 50 companies were involved — Microsoft, Netflix, and Adobe included — and the effort was coordinated with other industry organizations such as studio-backed digital locker initiator Digital Entertainment Content Ecosystem, LLC (DECE), OIPF, and World Wide Web Consortium (W3C). This resulted in the MPEG-DASH standard being developed [32].

MPEG-DASH (Dynamic Adaptive Streaming Over HTTP) is a flexible bitrate streaming technique. MPEG has developed quite a few extensively used multimedia standards, including MPEG-21, MPEG-7, MPEG-4 and MPEG-2. Their newest standard MPEG-DASH is an effort to resolve the intricacies of media delivery to various devices with an integrated common standard. When media content is delivered from conventional HTTP web servers, MPEG-DASH empowers high quality streaming of this media content over the Internet.

Just like Apple's HTTP Live Streaming (HLS) solution, MPEG-DASH can truly split a video file into small HTTP-based file segments before it is set to play through a stream. As a stream is playing, without causing disruption, each of these components can be combined into the stream. Especially, this is suitable for users who are working on a sluggish network because MPEG-DASH was developed to change up the bit rate delivery for these streams, giving users with a slower Internet connection a higher speed rate.

For video content producers, DASH is an amazingly attractive technology. A unique standard which allows them to encrypt once, and then distribute securely to a wide range of players, from desktop through HTML5 or plug-ins to the mobile and Over the Top (OTT) users. Without any special provisions, it also enables the deployment of the streaming services by utilizing the existing inexpensive and wide-spread Internet set-up. It supports both live streaming and on-demand. MPEG-DASH also works flawlessly with several DRM solutions, thus protecting the rights of sports and movie producers [33].

2.4.1 Implementation

The image below illustrates a media streaming scenario between a simple HTTP server and a DASH client. In figure 16, the multimedia content is captured and stored on an HTTP server and is delivered using HTTP. The content exists on the server in two parts: Media Presentation Description (MPD), which describes a manifest of the available content, its various alternatives, their URL addresses, and other characteristics; and segments, which contain the actual multimedia bitstreams in the form of chunks, in single or multiple files.

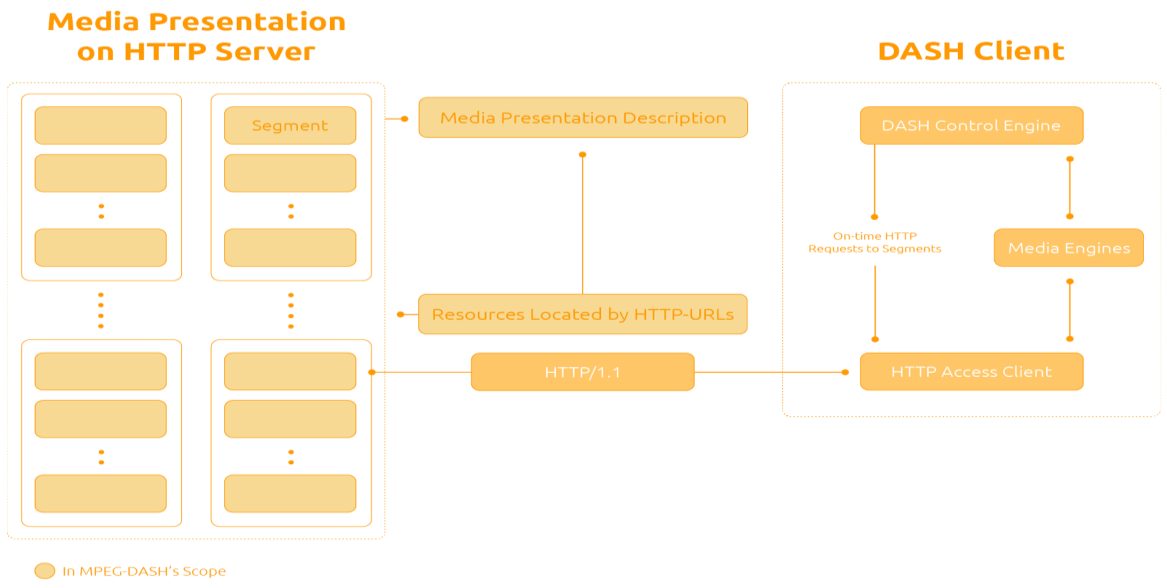

Media Presentation on HTTP Server
DASH Client
Segment
Media Presentation Description
Resources Located by HTTP-URLs
HTTP/1.1
DASH Control Engine
On-time HTTP Requests to Segments
Media Engines
HTTP Access Client
In MPEG-DASH's Scope

Figure 16: A simple DASH Implementation [32]

The MPEG-DASH Media Presentation Description (MPD) is basically an XML document containing information about media segments, their relationships and information necessary to choose between them, and other metadata that may be needed by clients. MPD's structure is the following [34]:

- **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

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.

In order to play the content, the DASH client first requests the MPD file from the server. The MPD can be delivered in various ways such as via HTTP or email. 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 digital rights management (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 [32].

2.4.2 Additional Benefits

One of the things that makes DASH streaming unique is its support for Encrypted Media Extensions (EME) and Media Source Extension (MSE) [30]. They are standards-based APIs for creating DRM schemes in the browser, generally for use with HTML5 & MPEG-DASH. Digital rights management (DRM) is a term for access control technologies that are used by hardware manufacturers, publishers, copyright holders and individuals to limit the use of digital content. The term is used to describe any technology that inhibits the use of digital content that is not desired or intended by the content provider and does not generally refer to other forms of copy protection, which can be circumvented without modifying the file or device, such as serial numbers or key files [35].

MSE is an API standard for JavaScript that generates video streams for playback over HTML5 using standard media tags. JavaScript acts as the intermediary for the streams which can be either live or on-demand. Because of this, video players can operate without the use of a plug-in and enables leading streaming standards like HLS, time shifting, and alternate audio tracks. MSE is not specific to any codec, container, or DRM scheme, so it can work with any format the browsers support, creating the opportunity for seamless cross-platform playback.

EME is a JavaScript API that is part of a larger content protection and delivery system for HTML5. EME uses Content Decryption Modules (CDM) which are built into the browser, but function independently of the browser. This was a requirement from the media and entertainment industry since the browser is viewed as insecure and untrusted. The CDM is the trusted component that content owners rely on for copyright protection. The DRM EME uses a Key System instead of what is traditionally referred to as a DRM scheme. The Key System is the component of the CDM that handles encryption and decryption of content. In the case of the web browser, each browser would have its own unique key system, which ensures cross browser compatibility [32].

3. CACHING

3.1 Introduction

With widespread penetration of the broadband Internet, multimedia service is getting increasingly popular among users. Video streaming is considered to be a major source of Internet traffic today and its usage continues to grow at a rapid rate. Being a very bandwidth-consuming kind of application, video traffic is forecasted to represent 82% of all Internet traffic by 2021 up from already observed 60% in 2016. Furthermore, Cisco predicts that we expect a huge increase in virtual reality and augmented reality traffic by 2021 [36].

To cope with this new and massive source of traffic, operators have proposed a lot of methods in order to reduce the amount of traffic traversing their networks and serve all the customers in a better way. One of the basic techniques to decrease the volume of video traffic circulating in the networks is what we call caching which allows every single user to efficiently reuse previously retrieved or computed data [37].

Caching is actually an industry-recognized technology that has been employed in various areas of the computer and networking industry for quite some time and it is widely used for improving user response time and minimizing bandwidth utilization. Although there are many different ways of implementing caching, this technology is a fairly simple concept. In fact, it is the process of storing copies of frequently used data in an easily accessible and temporary location so that time and resources are saved due to the fact that data does not have to be retrieved from the original source (e.g. from a server). Because time and resources are always a matter of great concern in the computer and networking industry, caches exist everywhere and are used in all high-performance systems.

Caches are found in both software and hardware. In fact, the CPU of every networking device such as a router or a switch takes advantage of caching in order to speed memory access. Certainly, several hundred million devices utilize cache technology in their processor, which illustrates the importance of caching, even at the hardware level [38].

3.2 Types of caching

Despite the fact that all the proposed caching mechanisms have the same purpose (e.g., reducing the loading time and minimizing the bandwidth), there are a lot of differences among them in terms of implementation. What follows are short details on each of those types of caching.

- **Page Cache** is a system that temporarily stores data such as web pages, images, and similar media content when a web page is loaded for the first time. It remembers the content and can quickly load the content each time something is requested by the user. When a user visits a page for the first time, a site cache commits selected content to memory. When that same page is visited again, the site cache can recall the same content, and then load it much quicker when compared to the first visit. Each visit to the same page is also loaded just as quickly from the cache [39].
- **Web Browser Cache** is used by the most widely used web browsers and it is founded on almost all PCs (figure 17). For example, Google Chrome or Mozilla Firefox have web caches for storing requested objects so that the same objects do not need to be retrieved multiple times from the Web server. A browser cache can save temporarily HTML pages, JavaScript scripts, images, or other types of multimedia content. In this type of caching, the end user is able to clear out its browser's cache whenever he wants [40].

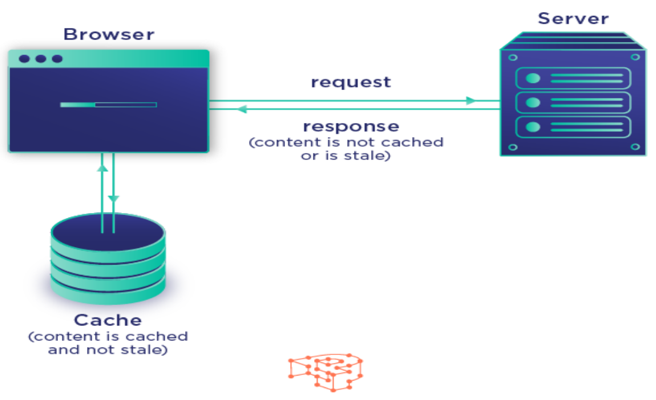

Browser
Server
request
response
(content is not cached or is stale)
Cache
(content is cached and not stale)

Figure 17: Web Browser Cache [41]

- **Server Cache** is completely different from the ones mentioned above, because, instead of just storing temporary content, it is fully handled by the server without the involvement of any end user or browser. There are a lot of types of server caching such as Object Caching (or Proxy Caching) where the client sends a request for a wanted object to a server and this request is inspected by an intermediate device that is in between the client and origin server(e.g., a proxy) [40].
- **Distributed Caching** comprises of a cluster of cache memories which may be distributed over a large geographical area. Distributed caching is primarily used by the big timers such as Google and Facebook who have a global audience and experience in high traffic volume. With the help of distributed caching, such companies can serve user requests without fail, regardless of the user's geographical location. Thanks to a wide network of caches used in this type of caching, there is virtually an infinite amount of data that can be cached and served when requested by the user [41].

3.3 Caching for Video Streaming

3.3.1 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 server and clients. The central server has a large storage space to store all the available videos for clients connected via a wide area network (WAN) or a local area network (LAN). In such a framework, all the requests from clients are handled at the central server. The request process starts with generating a request message from clients to the central server. In response to the client's request, the central server serves each request individually with 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 which will lead to significant drop of the 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 [42].

To leverage the workload of the central server and reduce the service latencies, an intermediate device called proxy is placed between the central server and clients who make requests to download their desirable content (figure 18). In the proxy-based architecture, a portion of video is cached in the proxy [40]. Upon receiving the 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 the request to the server (cache miss). If the proxy requests the object from the origin server, that object data is cached by the proxy so that it can fulfill future requests for the same object without retrieving it again from the origin server. The result of serving a cached object is that there is zero server-side bandwidth usage for the content and a huge improvement in response time for the end user. Meanwhile, the central server also delivers the uncached portion of the video to the client directly [43].

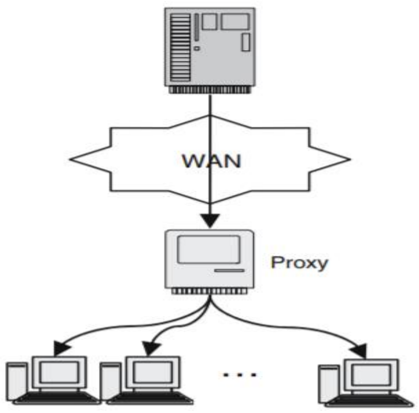


Figure 18: Client- Server connection with intermediate Proxy

Existing caching mechanisms about video streaming can be mainly classified into four categories [43]:

- **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.

3.3.2 CDN and video streaming

Content distribution network (CDN) 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 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 quality of service (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 [44].

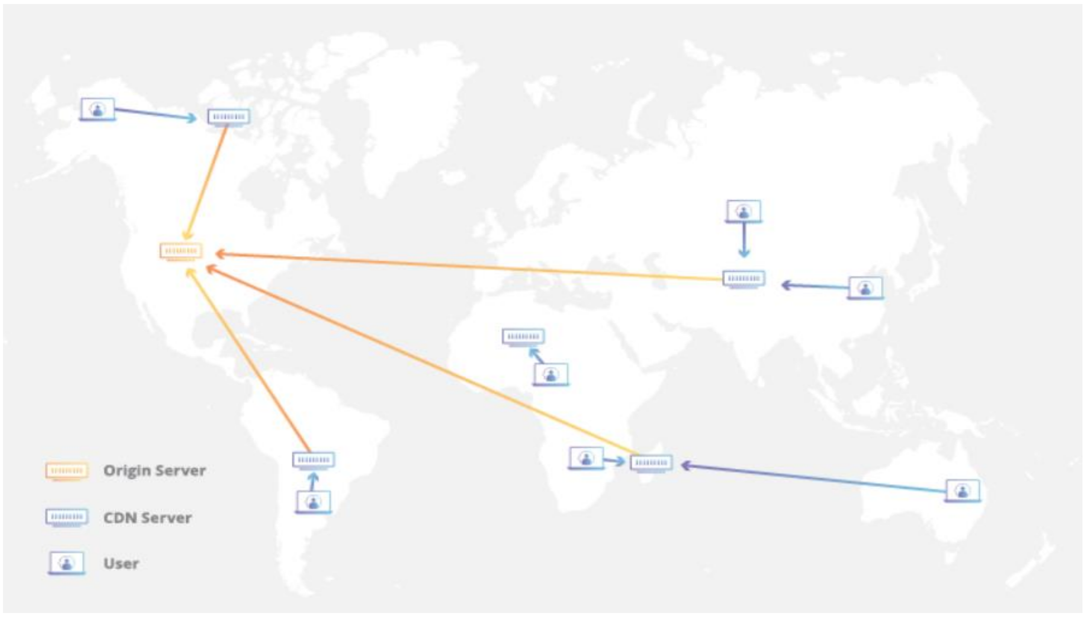


Figure 19: A Content Distribution Network [45]

3.4 Caching in HTTP Adaptive Streaming

From all the above, we can state that caching may improve the overall performance of streaming protocols like DASH. As stated earlier, in DASH the same media is encoded at different resolutions, bitrates and frame rates, called representations and each representation is virtually or physically divided into segments. The client via this protocol can download the appropriate segment according to its available bandwidth in order to have a superior enough experience. Cache servers are deployed within the access network to cache and store the available DASH segments according to the cache's buffer size. This allows clients to be served from the cache when we have a cash hit rather than retrieving every video segment from the origin server. This leads to higher speeds, lower latency, traffic avoidance and better Quality of Experience. However, the interplay between caching and HTTP Adaptive Streaming (HAS) is known to be intricate, and possibly detrimental to QoE.

Sometimes the client adaptation algorithm may overestimate the available path bandwidth when a video segment is requested directly from the cache server. This overestimation may force the client to upshift to a higher bitrate representation. If the subsequent segment for the higher bitrate representation is not already stored in the cache server, the cache will have to retrieve it from the origin server. As a result, increased delivery delay will be observed by the client as a lower throughput. The client may then downshift to a lower bitrate representation in response to the reduction in the throughput. If the subsequent segment for the lower bitrate representation is stored in the cache server, the observed throughput will increase. Consequently, an overestimation may be observed again. This undesired repeated cycle is known as bitrate oscillation and it is depicted in the figure below. This oscillation can deplete the playback buffer and cause the client algorithm to take drastic measures to refill it at the expense of video quality. A way to solve this problem is to use a HAS algorithm which is cache aware in order to benefit from the already cached video chunks. Those bitrate oscillation's problems can be easily eliminated if the client foreknew which of the chunks are closest to him (cache hit) just so that the video bitrate could be increased, and which are furthest from him (cache miss) so that the video bitrate could be decreased [37].

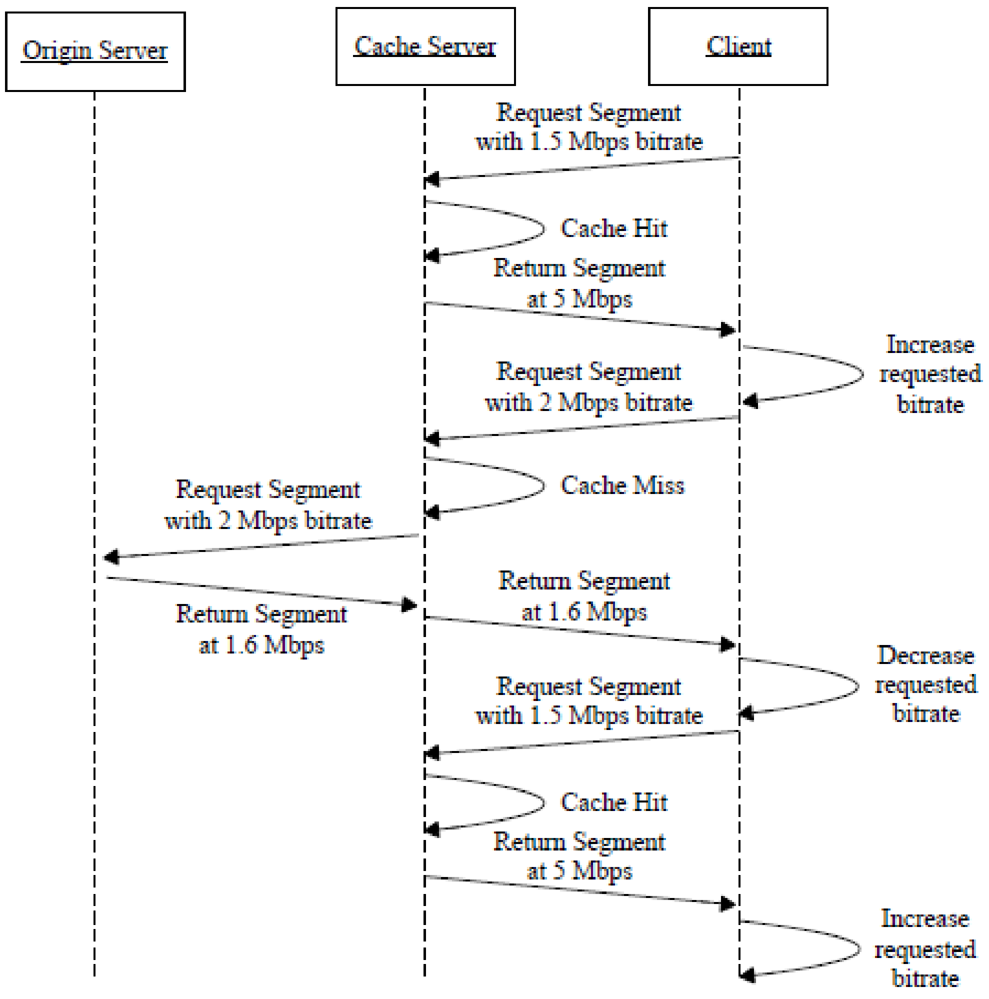

Origin Server
Cache Server
Client
Request Segment with 1.5 Mbps bitrate
Cache Hit
Return Segment at 5 Mbps
Increase requested bitrate
Request Segment with 2 Mbps bitrate
Cache Miss
Request Segment with 2 Mbps bitrate
Return Segment at 1.6 Mbps
Return Segment at 1.6 Mbps
Decrease requested bitrate
Request Segment with 1.5 Mbps bitrate
Cache Hit
Return Segment at 5 Mbps
Increase requested bitrate

Figure 20: A bitrate oscillation problem

4. QUALITY OF EXPERIENCE (QOE)

4.1 Introduction

Triggered by the exponential growth of mobile data and the proliferation of smart devices, 5G networks are expected to provide a wide range of services with the best Quality of service (QoS) guarantees. For instance, video-based services are one of the most critical services supported by 5G networks due to the fact that a very large percent of Internet traffic is video traffic. In order to improve video quality, many efforts have been made to optimize the QoS of the video transmission system, such as video compression optimization and network resource allocation. In addition to monitoring and controlling QoS, it is more crucial to assess video quality from users' perspective, which is widely known as Quality of Experience (QoE).

Currently, there is no strict definition of QoE. At the International Telecommunication Union (ITU-T), QoE is defined as "the overall acceptability of an application or service, as perceived subjectively by the end user." Recently, in the European Qualinet community, QoE was defined as "the degree of delight or annoyance of the user of an application or service". It results from the fulfillment of his or her expectations with respect to the utility and/or enjoyment of the application or service in the light of the user's personality and current state.". However, the general understanding of QoE is the same to a large extent: QoE is a new measurement approach for communication services, and it is determined by the interactions between users and services.

There is a consensus that the user QoE is becoming one of the primary focuses and will be the key to the competition in 5G wireless networks. Therefore, it is critical to investigate the needs of users, measurements of the user's subjective feelings, and methods to improve the user QoE [46].

4.2 QoE Influencing Factors

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 [47].

- **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 4 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.

4.3 QoE vs QoS

As we stated earlier, QoE is the degree of delight or annoyance of the user of an application or a service. On the other hand, QoS is “a set of technical quality requirements on the collective behaviors of one or more objects in order to define the required performance criteria”, but it does not reflect the user satisfaction. A lot of relationships have been proposed in order to relate the QoS with the QoE. For example, Stevens's power law, which is an empirical relationship in psychophysics between an increased intensity or strength in a physical stimulus and the perceived magnitude increase in the sensation created by the stimulus, has been proposed so as to relate these terms. This relationship was defined as $QoE = K * QoS^b$ [48]

However, there is an exponential relationship that is the best one for relating the QoS and the QoE, which is called “the IQX hypothesis” and is defined as $QoE = a * exp^{-b*QoS} + c$.

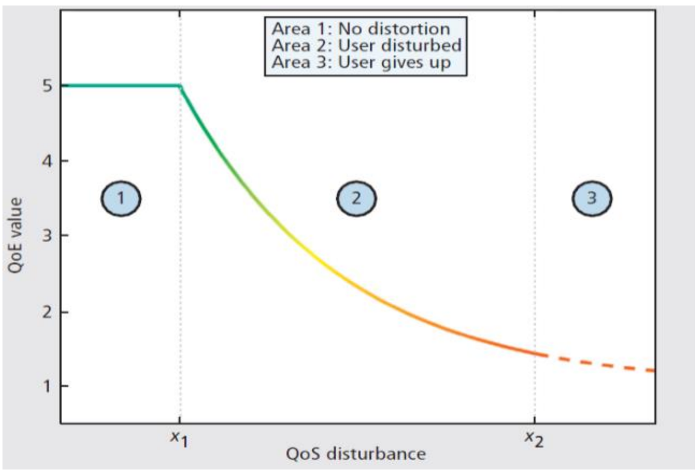


Figure 21: The IQX hypothesis diagram [49]

As we can see in the above figure, the relationship is negatively exponential, meaning that the less QoS we have available, the less QoE we expect to have. Furthermore, it is important to highlight the 3 regions that appear in the graph. In region 1, any slight drop of the QoS will not be perceived, and it will not have any impact on the user experience. This is very crucial for the operators because they are able to reduce their operating expenses while the user will not be annoyed by that, in terms of QoE. In region 2 instead, a very small QoS disturbance will strongly decrease the user’s experience. In region 3, the QoS level is very low and consequently the user will give up using this service or application.

4.4 QoE Measurements

QoE can be measured via QoE modelling. QoE assessment can be categorized into subjective assessment and objective assessment. Subjective assessment is based on psycho- acoustic/visual experiments, in which participant testers specify services in a controllable environment and give quality scores based on their own experiences. Subjective assessment is the most direct way to assess users' QoE since the results are given by humans directly. Nevertheless, the assessment methodology is excessively costly, which makes it difficult to be employed in a large- scale assessment.

In objective assessment, QoE is mathematically modeled with some objectively measurable factors as input variables, and the value of QoE is the corresponding output. The advantage of objective methods is the convenience and tractability. However, the evaluated QoE from the objective approach is an approximation, rather than the exact measurement for each user [46]. Objective assessment employs statistical models (media layer, parametric, bit-stream layer, hybrid etc.) for quality predictions.

The objective methods are algorithms and formulas that measure the quality of a video. These methods can be classified into three categories depending on the amount of information available for comparison with the original video [50]:

- Full reference: the original video and received video are available.
- No reference: only received video is available.
- Reduced reference: In addition to the received video, a description of the original video and some of its features are available.

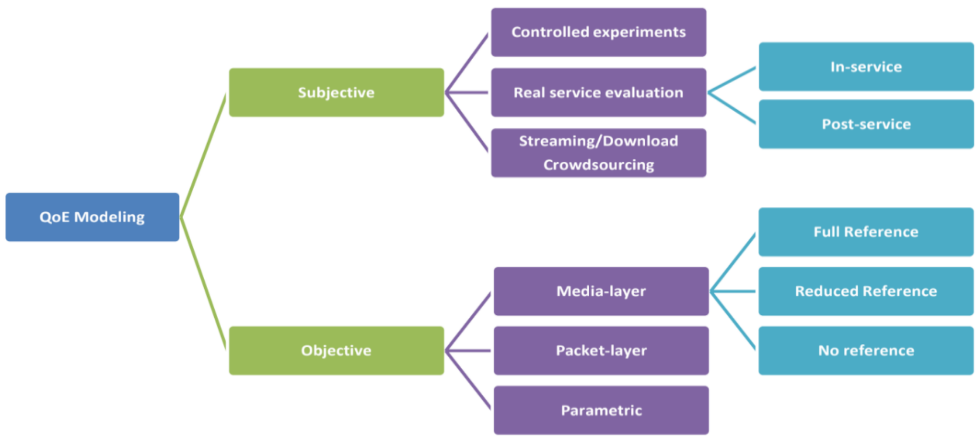


Figure 22: QoE Measurements

4.5 QoE Management

QoE management refers to the exploitation of QoE intelligence towards a more effective and efficient network management, both to the end-users' and the network operator's advantage. A high-level QoE management architecture is proposed in figure 23. In this design, an intelligent QoE entity is implemented at a central, administrative location inside the operator's network, on top of the network infrastructure. This central entity is able to obtain QoE-related input from appropriate elements and apply QoE-driven management decisions. It consists of three main building blocks: QoE-Controller, QoE-Monitor and QoE-Manager.

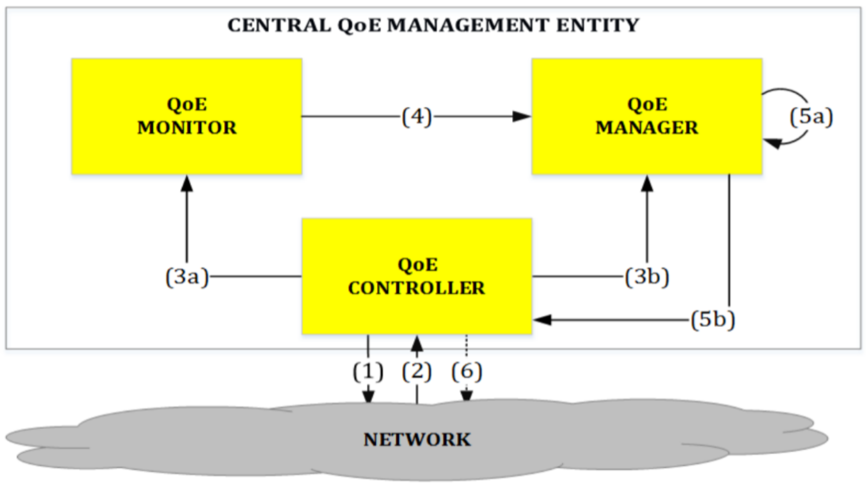


Figure 23: QoE Management Architecture

The QoE-Controller is the interface between the central QoE management entity and the network infrastructure in charge of a) synchronizing the QoE-related data acquisition process (1 and 2 in Figure 23), and b) imposing any QoE-related management decisions back to the network (6). The QoE-Monitor implements the QoE estimation functions (a.k.a. QoE models), namely it is running the appropriate QoE function per traffic flow. To operate, it uses input provided by the QoE-Controller (3a), i.e. QoE=f (input1, input2...). This QoE prediction/estimation is then provided as input to the QoE-Manager (4). The latter combines the QoE awareness with awareness about the current network topology and state (3b), in order to trigger any QoE-aware network management decisions to the network (5b) or conduct any Customer Experience Management procedures in general (5a) [51].

4.6 Big Data Analytics & QoE

With the latest advances in information technologies, the newest Traffic Monitoring and Analysis (TMA) solutions must have the ability to leverage the huge amount of information available from network elements and interfaces in mobile networks in order to ensure users' adequate QoE. With recent advances in information technologies, it is now possible to process massive volumes of information with big data analytics (BDA) platforms. “Big data” refers to data that cannot be processed by traditional means due to its volume, velocity, and variety (e.g., connection traces or packet-level traffic). With BDA, operators can better explain network performance by discovering hidden relationships between system variables (self-awareness) and take corrective actions proactively by predicting trend changes (self-adaptiveness). For this reason, specifying BDA systems has become a priority for standardization bodies

Legacy TMA tools were focused on the overall network performance. In contrast, the latest TMA solutions are able to passively monitor all the traffic crossing the network at a very fine granularity. This is achieved by adding new network elements (deep packet inspectors) that store a full or partial copy of each frame from different protocol layers. Then packet-level traffic analysis allows building service-specific performance metrics with end-to-end network visibility (service KPI, S-KPI), which can be mapped more easily into QoE figures. Such an approach is already used by the newest TMA solutions based on big networking data.

The data collected by TMA can be used offline as well for knowledge discovery by data analytics. Figure 24 shows several interrelated disciplines involved in this process, from the simplest data visualization steps in exploratory data analysis (EDA) to the most sophisticated machine learning (ML) algorithms. The aim here is to build models for classifying, characterizing, and predicting the performance of each individual session, for which the most important features must be selected.

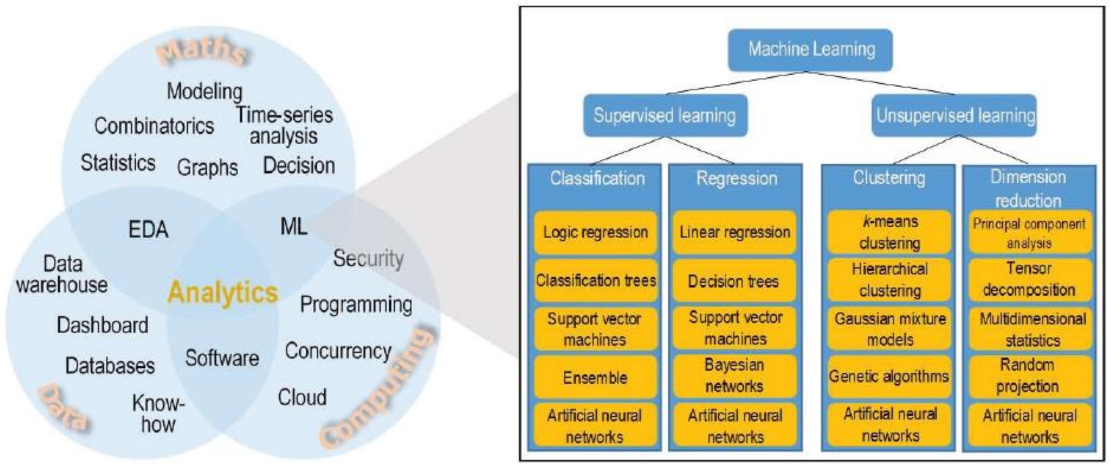


Figure 24: Taxonomy of machine learning algorithms [52]

4.7 HTTP Video Streaming influenced by QoE Factors

HTTP video streaming is a combination of download and concurrent playback. It transmits the video data to the client via HTTP where it is stored in an application buffer. After a sufficient amount of data has been downloaded (i.e., the video file download does not need to be complete yet), the client can start to play out the video from the buffer. However, network issues are possible to occur and they will decrease the throughput and introduce delays at the application layer. Such network issues are packet loss, insufficient bandwidth, jitter, and delay. Consequently, the buffer will not fill up as quickly as desired, so the video has to be interrupted until enough data has been received. These interruptions are known as stallings or rebuffering and they are one of the most important factors that influence QoE.

In general, the QoE influence factors can be categorized into technical and perceptual influence factors as depicted in figure 25. The perceptual influence factors are directly perceived by the end user of the application and are dependent but decoupled from the technical development [53].

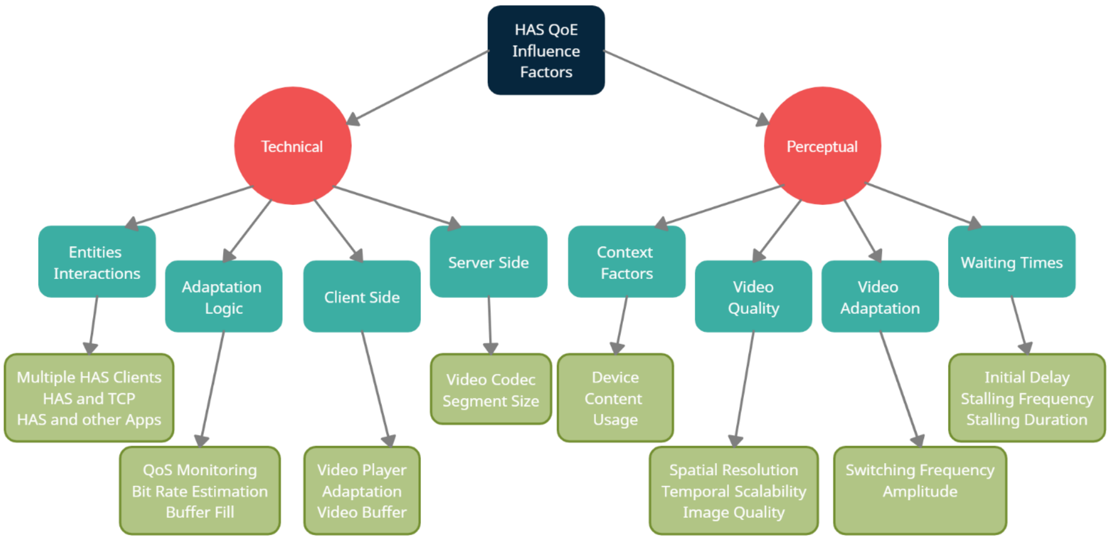

HAS QoE Influence Factors
Technical
Perceptual
Entities Interactions
Adaptation Logic
Client Side
Server Side
Context Factors
Video Quality
Video Adaptation
Waiting Times
Multiple HAS Clients HAS and TCP HAS and other Apps
QoS Monitoring Bit Rate Estimation Buffer Fill
Video Player Adaptation Video Buffer
Video Codec Segment Size
Device Content Usage
Spatial Resolution Temporal Scalability Image Quality
Switching Frequency Amplitude
Initial Delay Stalling Frequency Stalling Duration

Figure 25: HAS QoE Influence Factors

5. ENVIRONMENT SETUP

5.1 Introduction

Software-defined networking (SDN) technology is an approach to network management that enables dynamic, programmatically efficient network configuration in order to improve network performance and monitoring, making it more like cloud computing than traditional network management. SDN is meant to address the fact that the static architecture of traditional networks is decentralized and complex while current networks require more flexibility and easy troubleshooting. SDN attempts to centralize network intelligence in one network component by disassociating the forwarding process of network packets (data plane) from the routing process (control plane). The control plane consists of one or more controllers, which are considered the brain of the SDN network where the whole intelligence is incorporated. However, the intelligent centralization has its own drawbacks when it comes to security, scalability, and elasticity and this is the main issue of SDN. SDN was commonly associated with the OpenFlow protocol (for remote communication with network plane elements for the purpose of determining the path of network packets across network switches) since the latter's emergence in 2011 [54].

OpenDaylight (ODL) allows cloud engineers to programmatically deploy, configure and control virtual network services. As written on the OpenDaylight website, ODL helps Internet Service Providers, Academics and Cloud Service Providers to enable the following services: On-demand service delivery, Network Function Virtualization, Network Resource Optimization, Situational Awareness.

In other words, SDN is a technique that can be used to mimic the behavior of a traditional hardware network allowing to achieve the same behavior in a software environment. The control and data plane are separated giving you a centralized control to achieve this behavior. In this chapter we demonstrate all the steps that we followed, in order to set up the SDN environment.

5.2 System Requirements

5.2.1 Prepare Operating System (OS)

For the setup, the operating system used is the Ubuntu 18.04.1 LTS. To ensure that the server receives all of the most recent security and application packages, we run the following command:

```
$ sudo apt-get update
```

5.2.2 Java

A Java virtual machine (JVM) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. Having a specification ensures interoperability of Java programs across different implementations so that program authors using the Java Development Kit (JDK) need not worry about idiosyncrasies of the underlying hardware platform. The JVM reference implementation is developed by the OpenJDK project as open-source code [55].

OpenDaylight includes Karaf containers, OSGi (Open Service Gateway Initiative) bundles, and Java class files, which are portable and can run on any Java 8-compliant JVM.

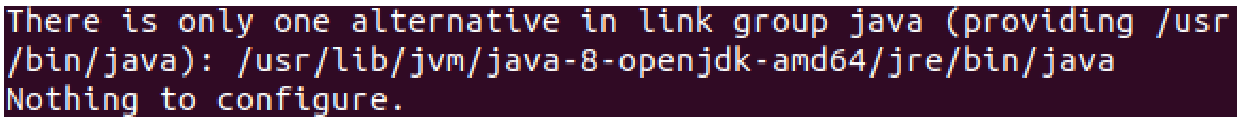
We run the commands shown below to install the Java JRE:

```
$ sudo add-apt-repository ppa:openjdk-r/ppa
$ sudo apt-get update
$ sudo apt-get install openjdk-8-jdk
```

To ensure that Ubuntu points to JAVA 8 we run the following command:

```
$ sudo update-alternatives --config java
```

The output should be the following:



```
There is only one alternative in link group java (providing /usr
/bin/java): /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java
Nothing to configure.
```

Figure 26: Correct output for JAVA 8

With the path in hand, we run the following command to update the BASHRC file and then source the file. Lastly, we check to ensure $JAVA_HOME lives in the environment.

```
$ echo 'export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre' >> ~/.bashrc
$ echo 'export TERM=xterm-color' >> ~/.bashrc
$ source ~/.bashrc
$ echo $JAVA_HOME
```

5.2.3 Mininet

Mininet is a network emulator, or perhaps more precisely a network emulation orchestration system. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. A Mininet host behaves just like a real machine; we can ssh into it (if we start up sshd and bridge the network to our host) and run arbitrary programs (including anything that is installed on the underlying Linux system.) The programs we run can send packets through what seems like a real Ethernet interface, with a given link speed and delay. Packets get processed by what looks like a real Ethernet switch, router, or middlebox, with a given amount of queueing. When two programs, like an iperf client and server, communicate through Mininet, the measured performance should match that of two (slower) native machines.

In short, Mininet's virtual hosts, switches, links, and controllers are the real thing – they are just created using software rather than hardware – and for the most part their behavior is similar to discrete hardware elements. It is usually possible to create a Mininet network that resembles a hardware network, or a hardware network that resembles a Mininet network, and to run the same binary code and applications on either platform [56].

To install Mininet natively from source, first we need to get the source code:

```
$ git clone https://github.com/mininet/mininet
```

Once we have the source tree, the command to install Mininet is:

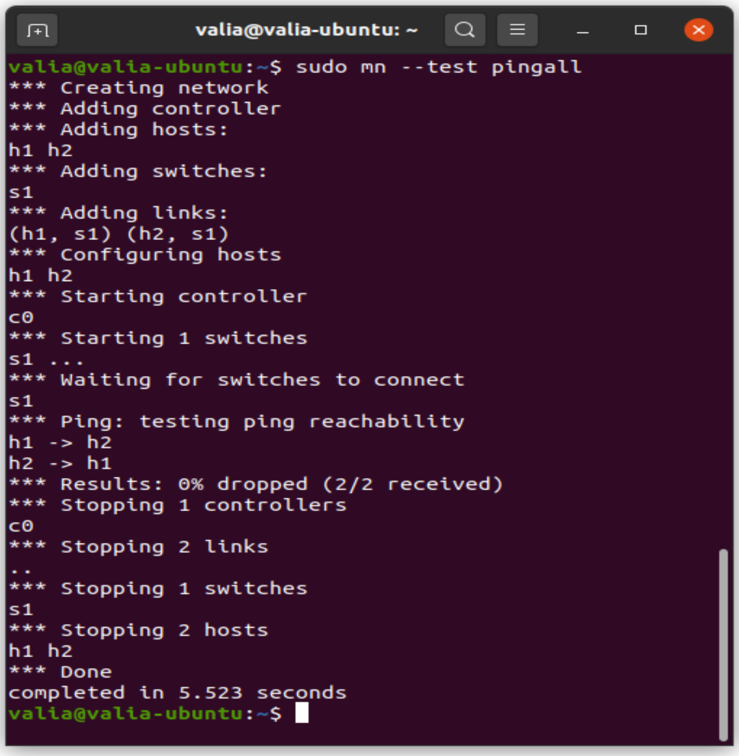
```
$ mininet/util/install.sh -a
```

The -a option installs everything that is included in the Mininet VM, including dependencies like Open vSwitch as well the additions like the OpenFlow Wireshark dissector and POX. By default, these tools will be built in directories created in our home directory.

After the installation has completed, to test the basic Mininet functionality:

```
$ sudo mn --test pingall
```

The output of the above command is shown below:



```
valia@valia-ubuntu:~$ sudo mn --test pingall
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Waiting for switches to connect
s1
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 5.523 seconds
valia@valia-ubuntu:~$
```

Figure 27: Pingall Output of Mininet’s basic topology

5.2.4 Mininet’s graphical user interface, MiniEdit

MiniEdit is an experimental tool created to demonstrate how Mininet can be extended. To show how to use MiniEdit to create and run network simulations, we will work through a tutorial that demonstrates how to use MiniEdit to build a network, configure network elements, save the topology, and run the simulation. To open it, we execute the following command in the directory where the Mininet folder is:

```
$ python ./mininet/examples/miniedit.py
```

MiniEdit has a simple user interface that presents a canvas with a row of tool icons on the left side of the window, and a menu bar along the top of the window [57].

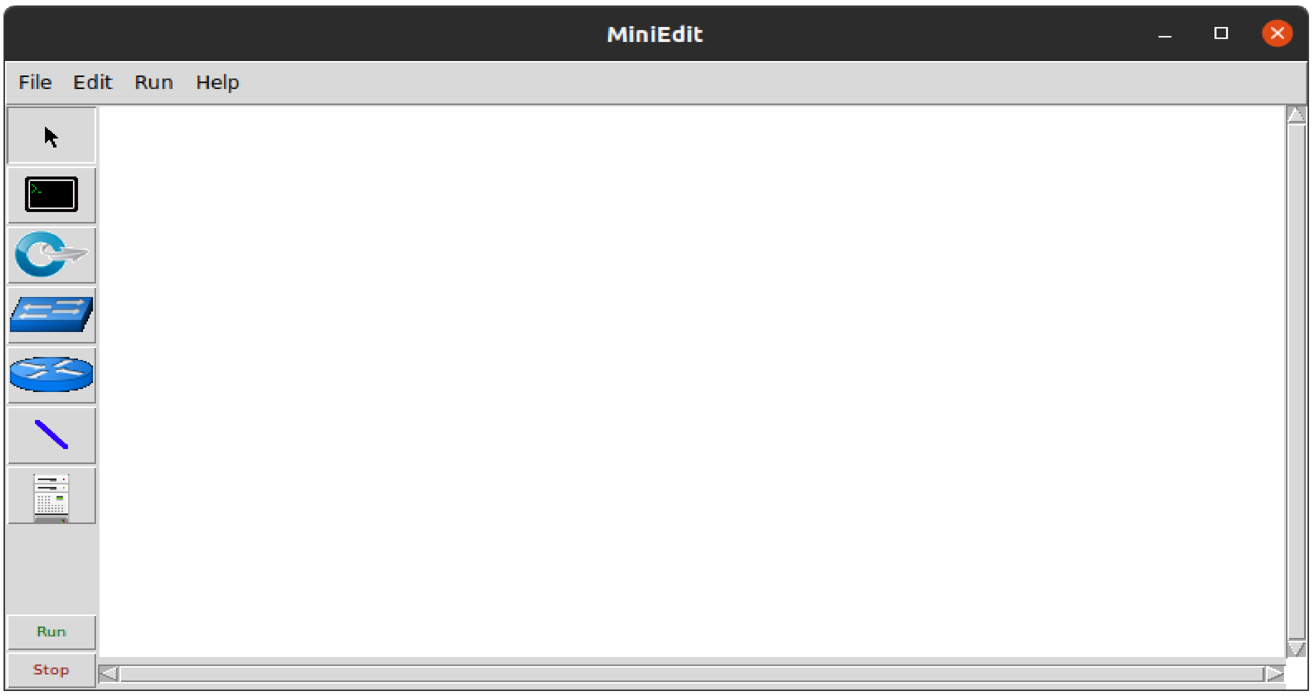


Figure 28: MiniEdit main page

A simple guide on the icons:

The Select tool is used to move nodes around on the canvas. Click and drag any existing node. Interestingly the Select tool is not needed to select a node or link on the canvas. To select an existing node or link, just hover the mouse pointer over it — this works regardless of the tool that is currently active — and then either right-click to reveal a configuration menu for the selected element or press the Delete key to remove the selected element.

The Host tool creates nodes on the canvas that will perform the function of host computers. Click on the tool, then click anywhere on the canvas you wish to place a node. As long as the tool remains selected, you can keep adding hosts by clicking anywhere on the canvas. The user may configure each host by right-clicking on it and choosing Properties from the menu.

The Switch tool creates OpenFlow-enabled switches on the canvas. These switches are expected to be connected to a controller. The tool operates the same way as the Hosts tool above. The user may configure each switch by right-clicking on it and choosing Properties from the menu.

The Legacy Switch tool creates a learning Ethernet switch with default settings. The switch will operate independently, without a controller. The legacy switch cannot be configured and is set up with Spanning Tree disabled, so we should not connect legacy switches in a loop.

The Legacy Router tool creates a basic router that will operate independently, without a controller. It is basically just a host with IP Forwarding enabled. The legacy router cannot be configured from the MiniEdit GUI.

The NetLink tool creates links between nodes on the canvas. Create links by selecting the NetLink tool, then clicking on one node and dragging the link to the target node. The user may configure the properties of each link by right-clicking on it and choosing Properties from the menu.

The Controller tool creates a controller. Multiple controllers can be added. By default, the MiniEdit creates a Mininet OpenFlow reference controller, which implements the behavior of a learning switch. Other controller types can be configured. The user may configure the properties of each controller by right clicking on it and choosing Properties from the menu.

The Run starts Mininet simulation scenario currently displayed in the MiniEdit canvas. The Stop button stops it. When MiniEdit simulation is in the “Run” state, right-clicking on network elements reveals operational functions such as opening a terminal window, viewing switch configuration, or setting the status of a link to “up” or “down”.

5.2.5 VLC Media Player

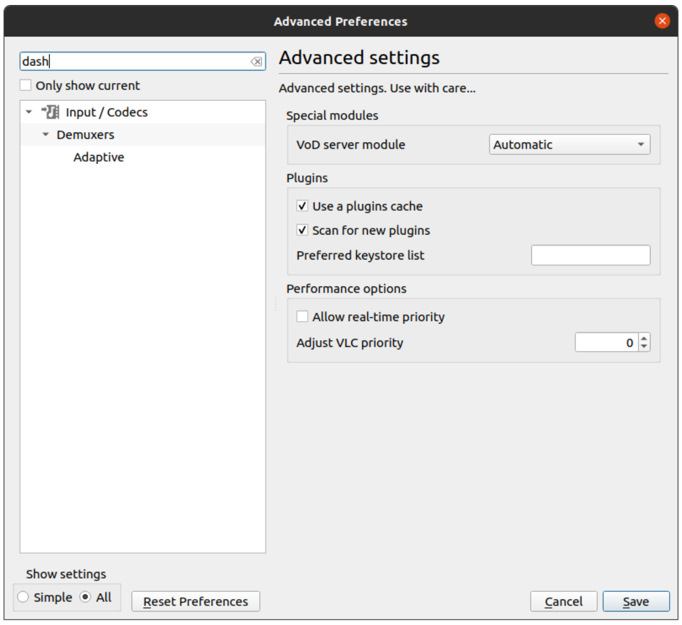
VLC media player (VLC) is a free and open-source, portable, cross-platform media player software and streaming media server developed by the VideoLAN project. VLC is available for desktop operating systems and mobile platforms, such as Android, iOS, iPadOS, Tizen, Windows 10 Mobile and Windows Phone. VLC is also available on digital distribution platforms such as Apple's App Store, Google Play, and Microsoft Store. VLC supports many audio and video compression methods and file formats, including DVD-Video, video CD and streaming protocols. It is able to stream media over computer networks and to transcode multimedia files.

The default distribution of VLC includes many free decoding and encoding libraries, avoiding the need for finding/calibrating proprietary plugins. The libavcodec library from the FFmpeg project provides many of VLC's codecs, but the player mainly uses its own muxers and demuxers. It also has its own protocol implementations. It also gained distinction as the first player to support playback of encrypted DVDs on Linux and macOS by using the libdvdcss DVD decryption library, however this library is legally controversial and is not included in many software repositories of Linux distributions as a result [58].

We execute the below command to install VLC media player:

```
$ sudo apt install vlc
```

Now, we need to make a few changes to VLC, so as to support Adaptive Streaming for DASH/HLS. To do that, we open the program, click on the Tools tab, and select Preferences. We check the All option in the Show setting section. Then, we type “dash” in the search bar on the top of the window.



Advanced settings

Advanced settings. Use with care...

Special modules

VoD server module: Automatic

Plugins

[✓] Use a plugins cache

[✓] Scan for new plugins

Preferred keystore list:

Performance options

[ ] Allow real-time priority

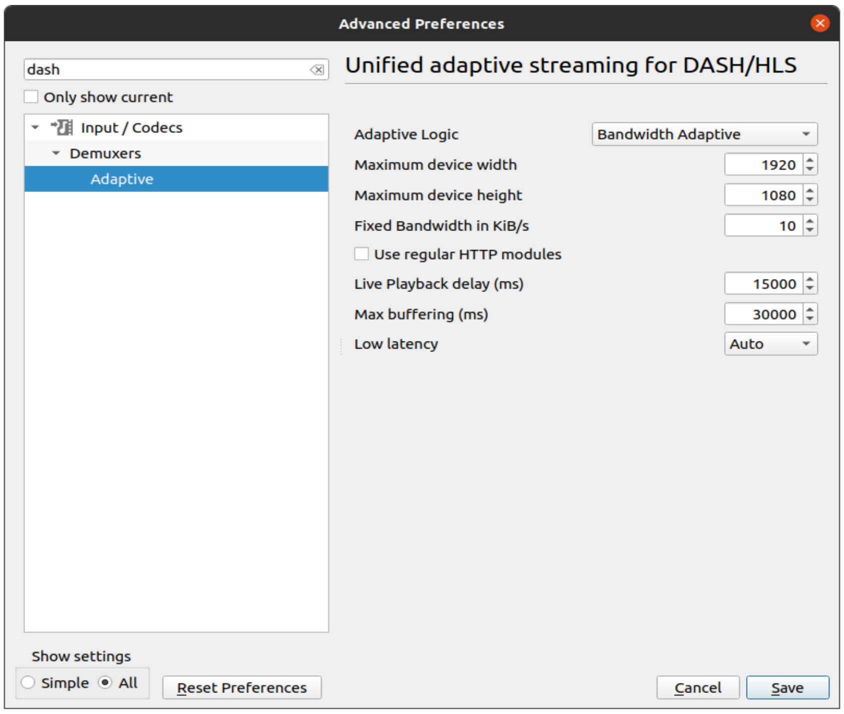
Adjust VLC priority: 0

[ ] Only show current

Show settings: Simple [✓ All]

Figure 29: VLC Preferences (1)

We navigate to the adaptive option and we can see that there are several adaptive logic algorithms: Predictive, Near Optimal, Bandwidth Adaptive, Fixed bandwidth, Lowest Bandwidth/Quality, Highest Bandwidth/Quality one. Some of them will be tested in our Mininet experiments, in the following chapters. For now, we set the parameters as follows:



Unified adaptive streaming for DASH/HLS

Adaptive Logic	Bandwidth Adaptive
Maximum device width	1920
Maximum device height	1080
Fixed Bandwidth in KiB/s	10
Use regular HTTP modules	[ ]
Live Playback delay (ms)	15000
Max buffering (ms)	30000
Low latency	Auto

[ ] Only show current

Show settings: Simple [✓ All]

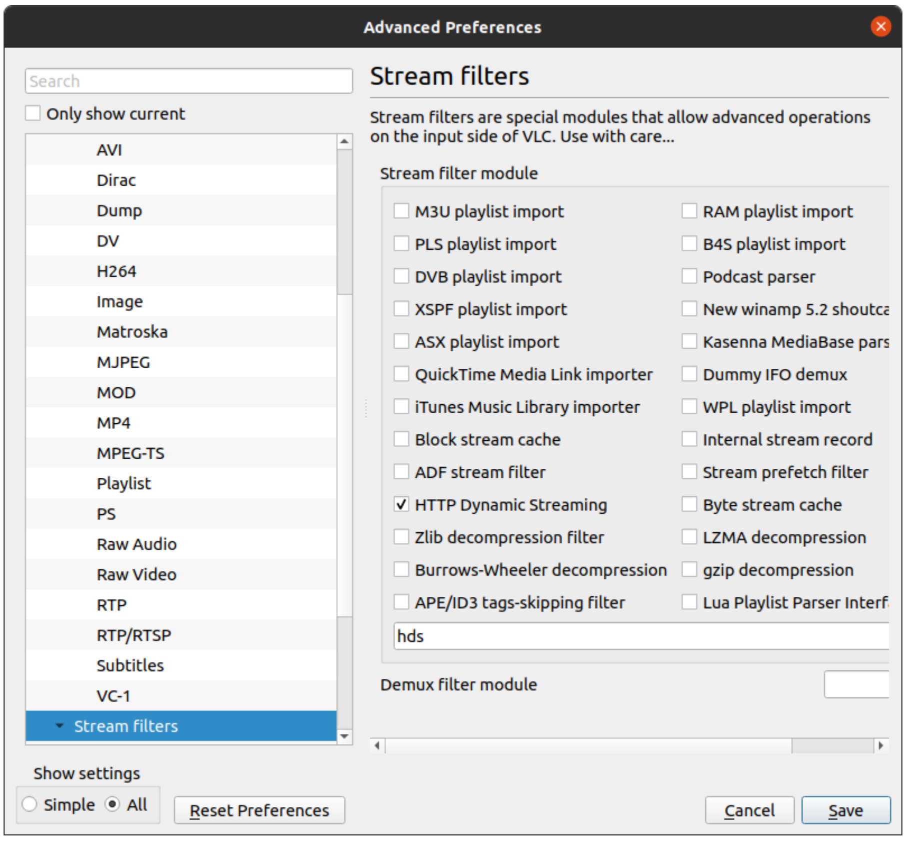
Figure 30: VLC Preferences (2)

Below we present a table for the maximum device width and height available in the VLC interface for manipulation, which are the resolutions that we will use later in our experiments.

Table 2: Display Resolutions- Pixel Dimensions

Maximum Device Width	Maximum Device Height
3840	2160 (4K)
2560	1440 (2K)
1920	1080 (FHD)
1280	720 (HD)
640	480
640	360

Then, we clear the dash filter, in order to select the Stream filters option, and finally, check the box that says HTTP Dynamic Streaming. Building VLC is a tricky procedure and is not recommended even from the developers, but it is possible by following the steps of this website https://wiki.videolan.org/Category:Building/, in case anyone wants to be more in charge of the adaptive logic by changing the code and building it again.



Stream filters

Stream filters are special modules that allow advanced operations on the input side of VLC. Use with care...

Stream filter module

[ ] M3U playlist import	[ ] RAM playlist import
[ ] PLS playlist import	[ ] B4S playlist import
[ ] DVB playlist import	[ ] Podcast parser
[ ] XSPF playlist import	[ ] New winamp 5.2 shoutca
[ ] ASX playlist import	[ ] Kasenna MediaBase pars
[ ] QuickTime Media Link importer	[ ] Dummy IFO demux
[ ] iTunes Music Library importer	[ ] WPL playlist import
[ ] Block stream cache	[ ] Internal stream record
[ ] ADF stream filter	[ ] Stream prefetch filter
[✓] HTTP Dynamic Streaming	[ ] Byte stream cache
[ ] Zlib decompression filter	[ ] LZMA decompression
[ ] Burrows-Wheeler decompression	[ ] gzip decompression
[ ] APE/ID3 tags-skipping filter	[ ] Lua Playlist Parser Interf

hds

Demux filter module

Figure 31: VLC Preferences (3)

After all that, we save our preferences.

5.2.6 FFmpeg

FFmpeg is a free and open-source software project consisting of a large suite of libraries and programs for handling video, audio, and other multimedia files and streams. At its core is the FFmpeg program itself, designed for command-line-based processing of video and audio files, and widely used for format transcoding, basic editing (trimming and concatenation), video scaling, video post-production effects, and standards compliance (SMPTE, ITU).

FFmpeg includes libavcodec, an audio/video codec library used by many commercial and free software products, libavformat (Lavf), an audio/video container mux and demux library, and the core FFmpeg command-line program for transcoding multimedia files.

FFmpeg is part of the workflow of hundreds of other software projects, and its libraries are a core part of software media players such as VLC and has been included in core processing for YouTube and iTunes. Codecs for the encoding and/or decoding of most audio and video file formats is included, making it highly useful for the transcoding of common and uncommon media files into a single common format.

The name of the project is inspired by the MPEG video standards group, together with "FF" for "fast forward". The logo uses a zigzag pattern that shows how MPEG video codecs handle entropy encoding [59].

To install the latest version of FFmpeg on Ubuntu 20.04 we start by updating the packages list [60]:

```
$ sudo apt update
```

Next, install FFmpeg by typing the following command:

```
$ sudo apt install ffmpeg
```

To validate that the package is installed properly use the ffmpeg -version command which prints the FFmpeg version:

```
$ ffmpeg -version
```

5.2.7 Python 2.7

Python is an interpreted, high-level and general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects [61].

Python was created in the late 1980s, and first released in 1991, by Guido van Rossum as a successor to the ABC programming language. Python 2.0, released in 2000, introduced new features, such as list comprehensions, and a garbage collection system with reference counting, and was discontinued with version 2.7 in 2020. Python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible, and much Python 2 code does not run unmodified on Python 3. With Python 2’s end-of-life, only Python 3.6.x and later are supported, with older versions still supporting e.g., Windows 7 (and old installers not restricted to 64-bit Windows) [62].

So, we need to install python and pip. Pip (recursive acronym for “Pip Installs Packages” or “Pip Installs Python“) is a cross-platform package manager for installing and managing

Python packages (which can be found in the Python Package Index (PyPI)) that comes with Python 2 >=2.7.9 or Python 3 >=3.4 binaries that are downloaded from python.org [63]. To install pip in Linux, we run the appropriate command for our distribution as follows:

```
$ sudo apt install python-pip
```

The command above will install Python2, Pip and all the dependencies required for building Python modules. Python usually comes preinstalled on most Linux systems. To check if python is installed in our system, we run the following command:

```
$ python -V
```

This should return a version number. In our case it’s 2.7.17.
Also, we should verify the installation by printing the pip version number:

```
$ pip --version
```

The version number may vary, but it will look something like this:

```
$ pip 9.0.1 from /usr/lib/python2.7/dist-packages (python 2.7)
```

Python’s SimpleHTTPServer module is a labor-saving tool that we can leverage for turning any directory in our system into an uncomplicated web server. It comes packaged with a simple HTTP server that delivers standard GET and HEAD request handlers. We will use that module in our implementation.

With a built-in HTTP server, we are not required to install or configure anything to have our web server up and running. To start the server, we run the below command:

```
# If Python version returned above is 3.X
$ python3 -m http.server
# If Python version returned above is 2.X
$ python -m SimpleHTTPServer
```

In our system, we have installed Python 2.7, so we will be using SimpleHTTPServer.

By default, this will run the contents of the directory on a local web server, on port 8000. We can go to this server by going to the URL localhost:8000 in our web browser. Here we’ll see the contents of the directory listed.

The last thing that we need to install is the requests module. Requests is an Apache2 Licensed HTTP library, written in Python, for human beings [62]. Requests allow us to send HTTP/1.1 requests extremely easily. There is no need to manually add query strings to our URLs, or to form-encode our POST data. Keep-alive and HTTP connection pooling are 100% automatic, thanks to urllib3 [64]. To install Requests, simply:

```
$ pip install requests
```

5.2.8 Project on advanced content (GPAC)

GPAC Project on Advanced Content (GPAC, a recursive acronym) is an implementation of the MPEG-4 Systems standard written in ANSI C. GPAC provides tools for media playback, vector graphics and 3D rendering, MPEG-4 authoring, and distribution.

GPAC provides three sets of tools based on a core library called libgpac:

- A multimedia player, cross-platform command-line based MP4Client or with a GUI Osmo4.
- A multimedia packager, MP4Box.
- Some server tools, around multiplexing and streaming (under development).

For this thesis, we will use the MP4Box, which segments the file and creates a Media Presentation Description (MPD). To install the MP4Box, we run the following command:

```
$ sudo apt-get install gpac
```

In the table that follows, we present all the tools that we used for our thesis, along with the versions:

Table 3: Used tools

TOOL	**VERSION**
openjdk	1.8.0_275
Mininet	2.3.0d6
VLC	3.0.8.0 Vetinari
FFmpeg	3.4.8-0ubuntu0.2
Python	2.7.17 and 3.6
MP4Box – GPAC	0.5.2-DEV-revVersion

6. MININET EXPERIMENTS

6.1 Preparation

For our experiments, we need a video with a resolution of 3840x2160 (4K-UHD), which we recorded with a mobile phone. The video size is 1.8 GB.

video.mp4 - VLC media player
Media Playback Audio Video Subtitle Tools View Help
02:01
05:00

Figure 32: Video playback

We create a bash.sh script, which converts every .mp4 file in the current folder and subfolder, to a multi-bitrate video in MP4 -DASH. This script will create six different quality versions of the video, an audio file and an .mpd manifest file with all the appropriate information about the given resolutions. So, we open a new terminal window inside the file, where the video is located, and we execute the bash.sh script. First, we change its permissions to make it executable.

```
$ chmod +x bash.sh
$ ./bash.sh
```

To explain the bash code, which can be found in Appendix A, first, we find the current directory name and we save it to a variable. Then, we have to check the location of the needed programs. If they are not installed correctly, an error message will inform the user in order to reinstall them [66].

Then, we save the name of every file without the file's extension, and we start the procedure of converting each file to a multi-bitrate video in MPEG-DASH.

Our first step is to convert the audio file with the help of the ffmpeg commands about the audio and video codec [66][67].

Next, we convert the video file to six different quality versions. The video codec that is used is H.264. We know that in this codec, the frames are organized in I,P, and B frames and I frames appear every 30 frames. A series of grouping frames is called GOP (Group of Pictures). The first frame in a GOP is an intra-frame called an I frame which contains most of the vital information for the following sequence of P frames which are forward predictive and B frames which are both forward and backward (bi-directional) predictive. A B frame estimates changes to the frame based on the previous and following frames. What's important to keep in mind is that I frames contain more data than P and B frames [68].

- -i is the input file
- -an: do not encode the audio
- -c:v libx264 is used because we want H:264 codec
- The keyint option determines the maximum number of frames in the GOP
- no-scenecut option is necessary in order not to add any additional IntraFrame I
- -b:v is the wanted bitrate to encode
- -bufsize tells the encoder how often to calculate the average bit rate and check to see if it conforms to the average bit rate specified
- -vsync 1 option makes the sum of the size of every chunk to be exactly the same with the complete video size so as to lose nothing in terms of MBs

After all that, we have all the video and audio files that we need and it's time to create the following with the help of the MP4Box tool.

The result of the execution of the bash file are the following files:

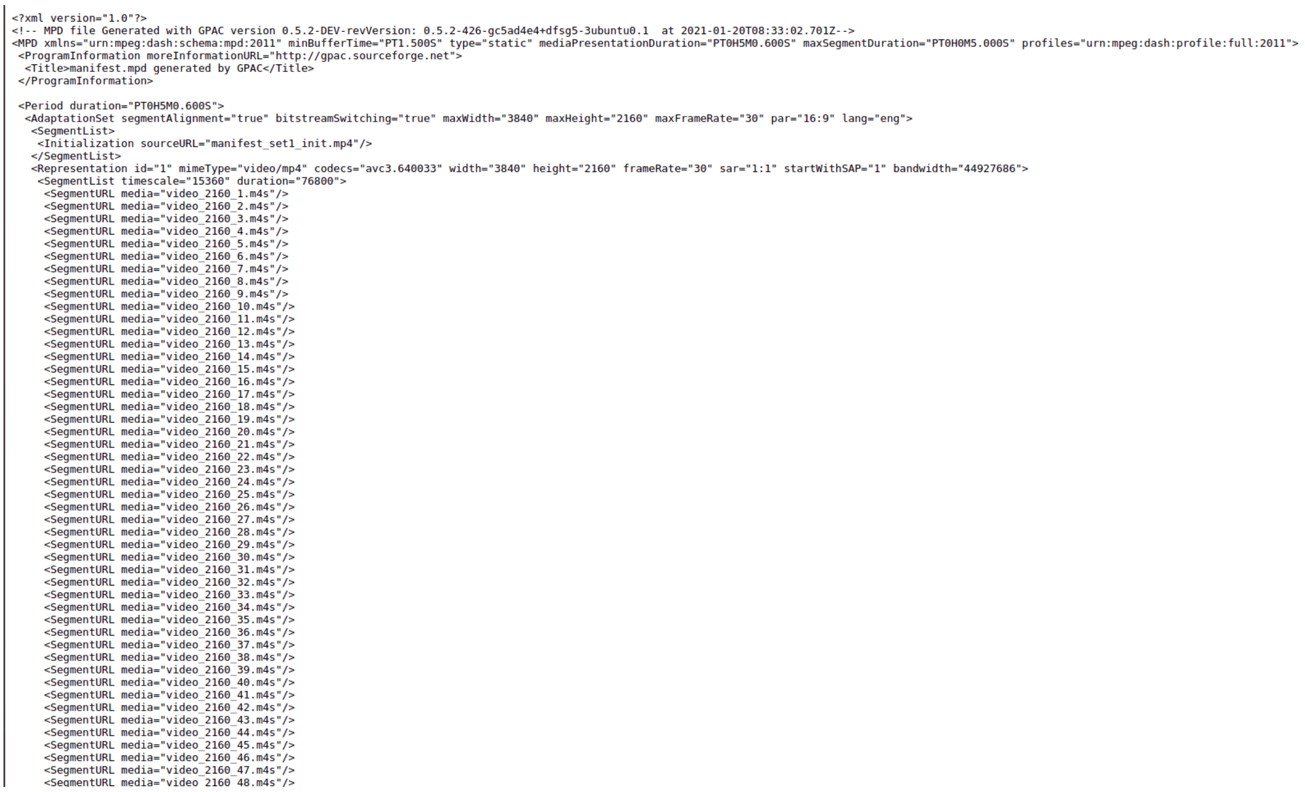
- a manifest-mpd file, which will store all the needed information about our project. This file will contain all the available video resolutions, their bitrates, and their size. In our project, a client, who wants to see a specific video, will ask about the manifest file (figures 33, 34).

Table 4: Video bitrates and sizes after the segmentation

Resolution	Video Bitrate (Mbps)	Size (MBs)
2160p	45	1700
1440p	16	597.6
1080p	8	299.1
720p	5	187.3
480p	2.5	94.1
360p	1	37.7

- all the video segments are of 5 seconds (the dash option below, defines the duration of each segment to 5000ms) for all the available resolutions and for the audio. For this reason, because our video is 5 minutes long, for each resolution the segments that are created are 61 (5 minutes = 300 seconds and 300 seconds / 5 seconds = 60 segments, and 1 more segment for initialization).

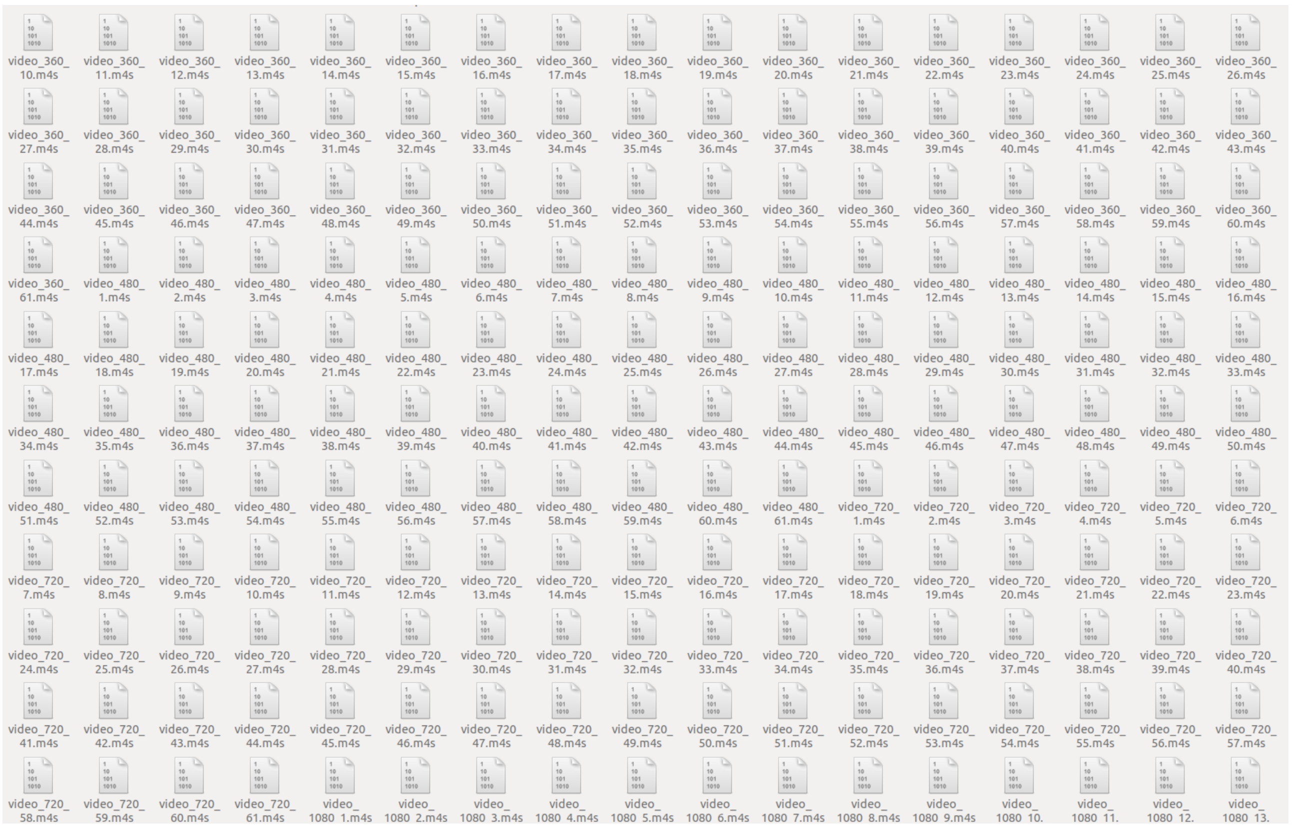
- an initialization file in order to start loading the segments (manifest_set1_init.mp4).



```
<?xml version="1.0"?>
<!-- MPD file Generated with GPAC version 0.5.2-DEV-revVersion: 0.5.2-426-gc5ad4e4+dfsg5-3ubuntu0.1  at 2021-01-20T08:33:02.701Z-->
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.500S" type="static" mediaPresentationDuration="PT0H5M0.600S" maxSegmentDuration="PT0H0M5.000S" profiles="urn:mpeg:dash:profile:full:2011">
 <ProgramInformation moreInformationURL="http://gpac.sourceforge.net">
  <Title>manifest.mpd generated by GPAC</Title>
 </ProgramInformation>

 <Period duration="PT0H5M0.600S">
  <AdaptationSet segmentAlignment="true" bitstreamSwitching="true" maxWidth="3840" maxHeight="2160" maxFrameRate="30" par="16:9" lang="eng">
   <SegmentList>
    <Initialization sourceURL="manifest_set1_init.mp4"/>
   </SegmentList>
   <Representation id="1" mimeType="video/mp4" codecs="avc3.640033" width="3840" height="2160" frameRate="30" sar="1:1" startWithSAP="1" bandwidth="44927686">
    <SegmentList timescale="15360" duration="76800">
     <SegmentURL media="video_2160_1.m4s"/>
     <SegmentURL media="video_2160_2.m4s"/>
     <SegmentURL media="video_2160_3.m4s"/>
     <SegmentURL media="video_2160_4.m4s"/>
     <SegmentURL media="video_2160_5.m4s"/>
     <SegmentURL media="video_2160_6.m4s"/>
     <SegmentURL media="video_2160_7.m4s"/>
     <SegmentURL media="video_2160_8.m4s"/>
     <SegmentURL media="video_2160_9.m4s"/>
     <SegmentURL media="video_2160_10.m4s"/>
     <SegmentURL media="video_2160_11.m4s"/>
     <SegmentURL media="video_2160_12.m4s"/>
     <SegmentURL media="video_2160_13.m4s"/>
     <SegmentURL media="video_2160_14.m4s"/>
     <SegmentURL media="video_2160_15.m4s"/>
     <SegmentURL media="video_2160_16.m4s"/>
     <SegmentURL media="video_2160_17.m4s"/>
     <SegmentURL media="video_2160_18.m4s"/>
     <SegmentURL media="video_2160_19.m4s"/>
     <SegmentURL media="video_2160_20.m4s"/>
     <SegmentURL media="video_2160_21.m4s"/>
     <SegmentURL media="video_2160_22.m4s"/>
     <SegmentURL media="video_2160_23.m4s"/>
     <SegmentURL media="video_2160_24.m4s"/>
     <SegmentURL media="video_2160_25.m4s"/>
     <SegmentURL media="video_2160_26.m4s"/>
     <SegmentURL media="video_2160_27.m4s"/>
     <SegmentURL media="video_2160_28.m4s"/>
     <SegmentURL media="video_2160_29.m4s"/>
     <SegmentURL media="video_2160_30.m4s"/>
     <SegmentURL media="video_2160_31.m4s"/>
     <SegmentURL media="video_2160_32.m4s"/>
     <SegmentURL media="video_2160_33.m4s"/>
     <SegmentURL media="video_2160_34.m4s"/>
     <SegmentURL media="video_2160_35.m4s"/>
     <SegmentURL media="video_2160_36.m4s"/>
     <SegmentURL media="video_2160_37.m4s"/>
     <SegmentURL media="video_2160_38.m4s"/>
     <SegmentURL media="video_2160_39.m4s"/>
     <SegmentURL media="video_2160_40.m4s"/>
     <SegmentURL media="video_2160_41.m4s"/>
     <SegmentURL media="video_2160_42.m4s"/>
     <SegmentURL media="video_2160_43.m4s"/>
     <SegmentURL media="video_2160_44.m4s"/>
     <SegmentURL media="video_2160_45.m4s"/>
     <SegmentURL media="video_2160_46.m4s"/>
     <SegmentURL media="video_2160_47.m4s"/>
     <SegmentURL media="video_2160_48.m4s"/>
```

Figure 33 : The Manifest File



```
video_360_10.m4s video_360_11.m4s video_360_12.m4s video_360_13.m4s video_360_14.m4s video_360_15.m4s video_360_16.m4s video_360_17.m4s video_360_18.m4s video_360_19.m4s video_360_20.m4s video_360_21.m4s video_360_22.m4s video_360_23.m4s video_360_24.m4s video_360_25.m4s video_360_26.m4s
video_360_27.m4s video_360_28.m4s video_360_29.m4s video_360_30.m4s video_360_31.m4s video_360_32.m4s video_360_33.m4s video_360_34.m4s video_360_35.m4s video_360_36.m4s video_360_37.m4s video_360_38.m4s video_360_39.m4s video_360_40.m4s video_360_41.m4s video_360_42.m4s video_360_43.m4s
video_360_44.m4s video_360_45.m4s video_360_46.m4s video_360_47.m4s video_360_48.m4s video_360_49.m4s video_360_50.m4s video_360_51.m4s video_360_52.m4s video_360_53.m4s video_360_54.m4s video_360_55.m4s video_360_56.m4s video_360_57.m4s video_360_58.m4s video_360_59.m4s video_360_60.m4s
video_360_61.m4s video_480_1.m4s video_480_2.m4s video_480_3.m4s video_480_4.m4s video_480_5.m4s video_480_6.m4s video_480_7.m4s video_480_8.m4s video_480_9.m4s video_480_10.m4s video_480_11.m4s video_480_12.m4s video_480_13.m4s video_480_14.m4s video_480_15.m4s video_480_16.m4s
video_480_17.m4s video_480_18.m4s video_480_19.m4s video_480_20.m4s video_480_21.m4s video_480_22.m4s video_480_23.m4s video_480_24.m4s video_480_25.m4s video_480_26.m4s video_480_27.m4s video_480_28.m4s video_480_29.m4s video_480_30.m4s video_480_31.m4s video_480_32.m4s video_480_33.m4s
video_480_34.m4s video_480_35.m4s video_480_36.m4s video_480_37.m4s video_480_38.m4s video_480_39.m4s video_480_40.m4s video_480_41.m4s video_480_42.m4s video_480_43.m4s video_480_44.m4s video_480_45.m4s video_480_46.m4s video_480_47.m4s video_480_48.m4s video_480_49.m4s video_480_50.m4s
video_480_51.m4s video_480_52.m4s video_480_53.m4s video_480_54.m4s video_480_55.m4s video_480_56.m4s video_480_57.m4s video_480_58.m4s video_480_59.m4s video_480_60.m4s video_480_61.m4s video_720_1.m4s video_720_2.m4s video_720_3.m4s video_720_4.m4s video_720_5.m4s video_720_6.m4s
video_720_7.m4s video_720_8.m4s video_720_9.m4s video_720_10.m4s video_720_11.m4s video_720_12.m4s video_720_13.m4s video_720_14.m4s video_720_15.m4s video_720_16.m4s video_720_17.m4s video_720_18.m4s video_720_19.m4s video_720_20.m4s video_720_21.m4s video_720_22.m4s video_720_23.m4s
video_720_24.m4s video_720_25.m4s video_720_26.m4s video_720_27.m4s video_720_28.m4s video_720_29.m4s video_720_30.m4s video_720_31.m4s video_720_32.m4s video_720_33.m4s video_720_34.m4s video_720_35.m4s video_720_36.m4s video_720_37.m4s video_720_38.m4s video_720_39.m4s video_720_40.m4s
video_720_41.m4s video_720_42.m4s video_720_43.m4s video_720_44.m4s video_720_45.m4s video_720_46.m4s video_720_47.m4s video_720_48.m4s video_720_49.m4s video_720_50.m4s video_720_51.m4s video_720_52.m4s video_720_53.m4s video_720_54.m4s video_720_55.m4s video_720_56.m4s video_720_57.m4s
video_720_58.m4s video_720_59.m4s video_720_60.m4s video_720_61.m4s video_1080_1.m4s video_1080_2.m4s video_1080_3.m4s video_1080_4.m4s video_1080_5.m4s video_1080_6.m4s video_1080_7.m4s video_1080_8.m4s video_1080_9.m4s video_1080_10. video_1080_11. video_1080_12. video_1080_13.
```

Figure 34: Segments created after the execution of bash script

6.2 Basic Client-Server Topology on Mininet

6.2.1 SimpleHTTPServer module

In order to implement our experiment, we will make use of Python's SimpleHTTPServer module. The SimpleHTTPServer module that comes with Python is a simple HTTP server that provides standard GET and HEAD request handlers. The default code that is provided will be used for the main server and a modification of it will be used for the cache server. The default code is presented in the Appendix A, extracted from the official GitHub repository.

To explain this code, the SimpleHTTPServer module defines a single class, SimpleHTTPRequestHandler, which is interface-compatible with the BaseHTTPServer.BaseHTTPRequestHandler.

```
class SimpleHTTPServer.SimpleHTTPRequestHandler(request, client_address,
server)
```

This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests. A lot of the work, such as parsing the request, is done by the base class BaseHTTPServer.BaseHTTPRequestHandler. This class implements the do_GET() and do_HEAD() functions.

The following are defined as class-level attributes of SimpleHTTPRequestHandler:

- server_version: This will be "SimpleHTTP/" + __version__, where __version__ is defined at the module level.
- extensions_map: A dictionary mapping suffixes into MIME types. The default is signified by an empty string and is considered to be application/octet-stream. The mapping is used case-insensitively, and so should contain only lower-cased keys.

The SimpleHTTPRequestHandler class defines the following methods:

- do_HEAD(): This method serves the 'HEAD' request type: it sends the headers it would send for the equivalent GET request. See the do_GET() method for a more complete explanation of the possible headers.
- do_GET(): The request is mapped to a local file by interpreting the request as a path relative to the current working directory. If the request was mapped to a directory, the directory is checked for a file named index.html or index.htm (in that order). If found, the file's contents are returned; otherwise, a directory listing is generated by calling the list_directory() method. This method uses os.listdir() to scan the directory and returns a 404-error response if the listdir() fails. If the request was mapped to a file, it is opened, and the contents are returned. Any IOError exception in opening the requested file is mapped to a 404, 'File not found' error. Otherwise, the content type is guessed by calling the guess_type() method, which in turn uses the extensions_map variable. A 'Content-type:' header with the guessed content type is output, followed by a 'Content-Length:' header with the file's size and a 'Last-Modified:' header with the file's modification time. Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with text/ the file is opened in text mode; otherwise binary mode is used.

The test() function in the SimpleHTTPServer module is an example which creates a server using the SimpleHTTPRequestHandler as the Handler [65].

To run this module, we simply type:

```
$ python -m SimpleHTTPServer 8000
```

The default port number will change, as we will see in the next chapters.

6.2.2 Implementation

A simple server-client topology in MiniEdit looks like this in the figure below. In order to rename the hosts, we click on its host icon and select properties.

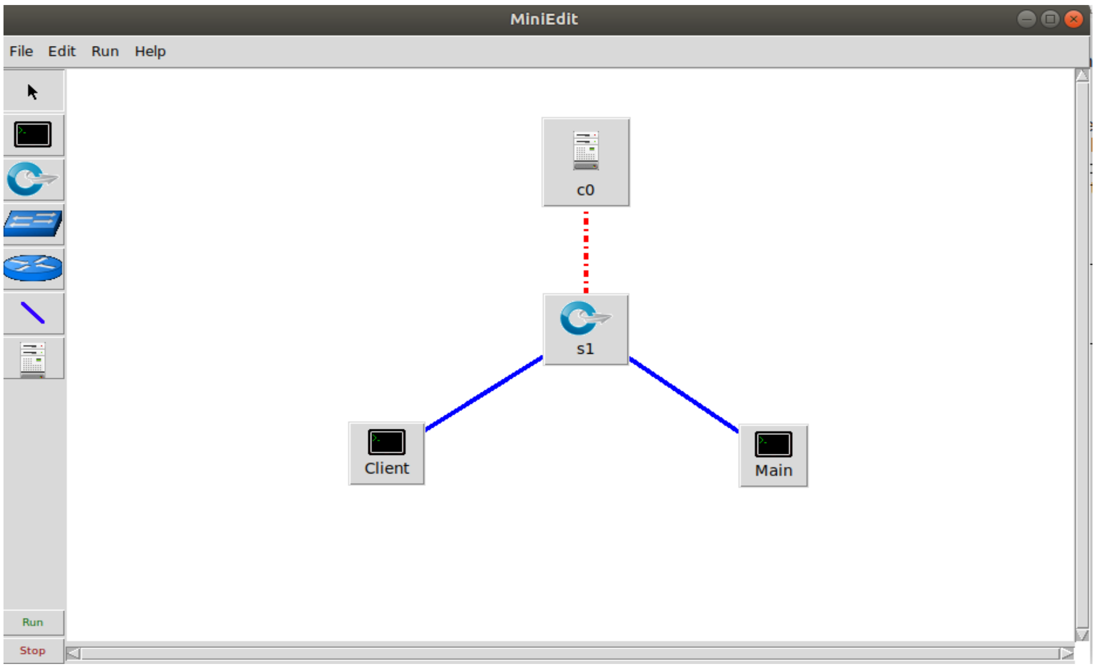
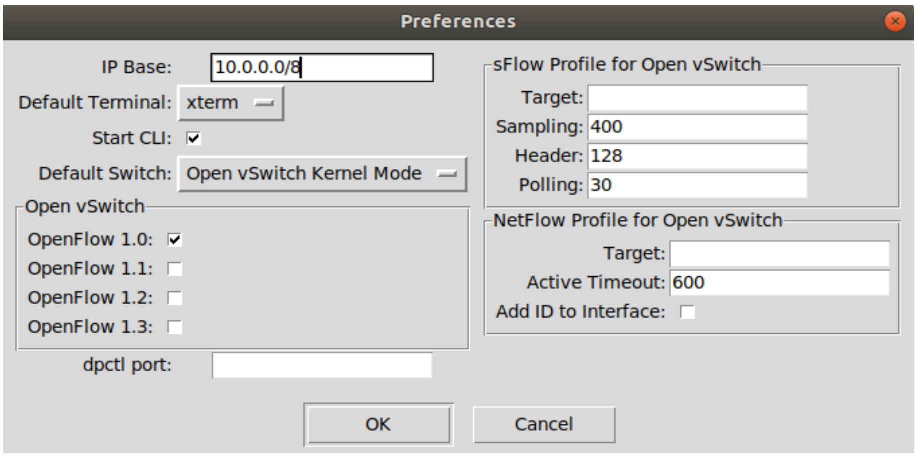


Figure 35: Basic Miniedit Topology

To set the wanted MiniEdit Preferences, we click on Edit-→ Preferences and check the Start CLI option, so as to be able to use it during our simulation. Now we can click the Run button.


Preferences

IP Base:	10.0.0.0/8
Default Terminal:	xterm
Start CLI:	✓
Default Switch:	Open vSwitch Kernel Mode

Open vSwitch

OpenFlow 1.0:	✓
OpenFlow 1.1:	
OpenFlow 1.2:	
OpenFlow 1.3:	

dpctl port:	

sFlow Profile for Open vSwitch

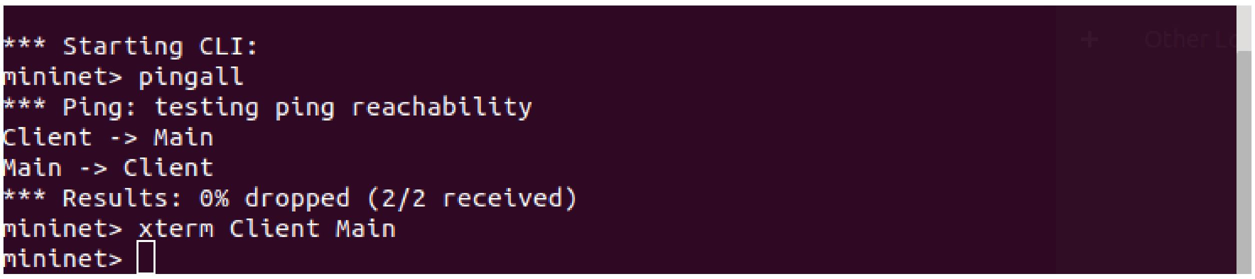
Target:	
Sampling:	400
Header:	128
Polling:	30

NetFlow Profile for Open vSwitch

Target:	
Active Timeout:	600
Add ID to Interface:	

Figure 36: MiniEdit's Preferences

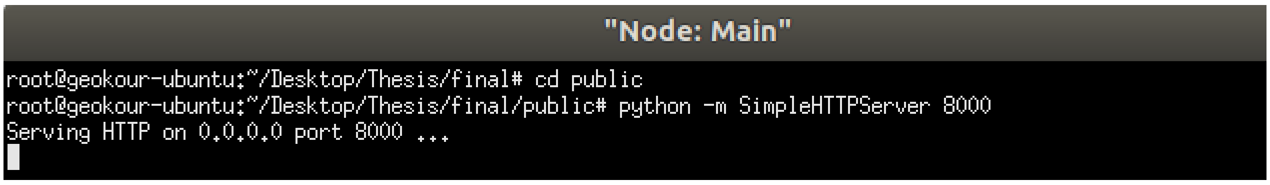
After starting the experiment, we pingall in order to test the reachability and then we execute the xterm command of each host.



```
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
Client -> Main
Main -> Client
*** Results: 0% dropped (2/2 received)
mininet> xterm Client Main
mininet> 
```

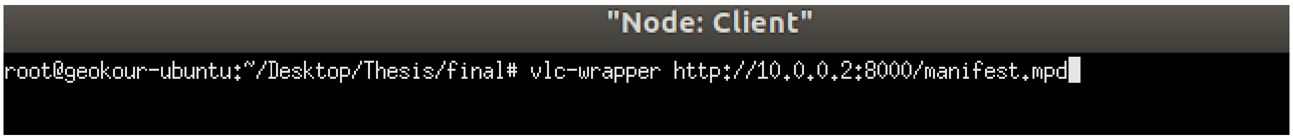
Figure 37: Miniedit commands

In the different terminals, we execute the following commands



```
"Node: Main"
root@geokour-ubuntu:~/Desktop/Thesis/final# cd public
root@geokour-ubuntu:~/Desktop/Thesis/final/public# python -m SimpleHTTPServer 8000
Serving HTTP on 0.0.0.0 port 8000 ...
```

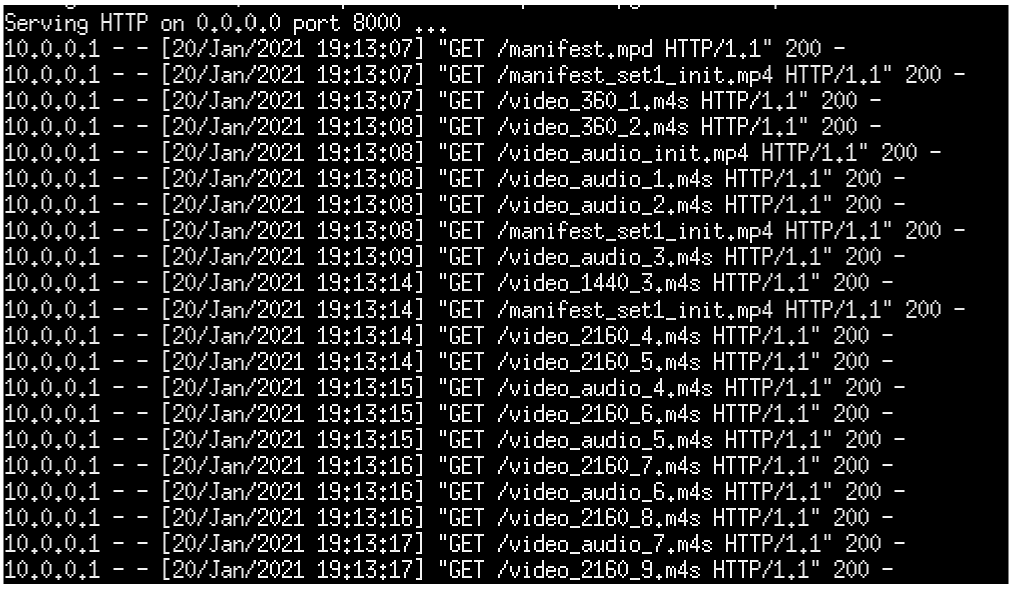
Figure 38: xterm Main Server



```
"Node: Client"
root@geokour-ubuntu:~/Desktop/Thesis/final# vlc-wrapper http://10.0.0.2:8000/manifest.mpd
```

Figure 39: xterm Client

We start the HTTP server in the Main host, using the port 8000. Secondly, we load the .mpd file in the Client, using the vlc-wrapper command. Finally, we are now ready to watch the video playing with the VLC interface. The Main terminal output will demonstrate the first request for the manifest file and all the requested chunks of the video and audio. The client's buffer starts to fill up with chunks of the lowest available resolution, in order to avoid stallings and when VLC understands that there is enough available bandwidth, it serves the client with a better-quality video.



```
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.0.1 - - [20/Jan/2021 19:13:07] "GET /manifest.mpd HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:07] "GET /manifest_set1_init.mp4 HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:07] "GET /video_360_1.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:08] "GET /video_360_2.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:08] "GET /video_audio_init.mp4 HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:08] "GET /video_audio_1.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:08] "GET /video_audio_2.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:08] "GET /manifest_set1_init.mp4 HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:09] "GET /video_audio_3.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:14] "GET /video_1440_3.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:14] "GET /manifest_set1_init.mp4 HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:14] "GET /video_2160_4.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:14] "GET /video_2160_5.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:15] "GET /video_audio_4.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:15] "GET /video_2160_6.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:15] "GET /video_audio_5.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:16] "GET /video_2160_7.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:16] "GET /video_audio_6.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:16] "GET /video_2160_8.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:17] "GET /video_audio_7.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:13:17] "GET /video_2160_9.m4s HTTP/1.1" 200 -
```

Figure 40: Output of the HTTP Server terminal

When we want to stop the simulation, we hit the stop button in MiniEdit's terminal, and type exit to quit from the Mininet's CLI Window.

6.3 Client- Cache Server- Main Server Topology on Mininet

6.3.1 Problem statement

In the previous topology, it is investigated how to deliver DASH-based content from a single server (called main) through the Mininet's environment. A custom Mininet topology is created with a client/server communication scheme through a switch, connected with the controller. The server-client implementation for video streaming had an HTTP Adaptive Video Streaming logic on top of this environment. A set of experiments were conducted under varying bandwidth conditions that indicated that the transmitted data is utterly connected with the existing traffic in the network. More precisely, the higher the bandwidth the higher the resolution selected by the media player. Furthermore, in some cases, the resolution of the broadcasted video would not change instantly. Six different resolutions were offered in the channel: 360p, 480p, 720p,1080p, 1440p and 2160p, and they were the only factors that were influenced by the change in the bandwidth.

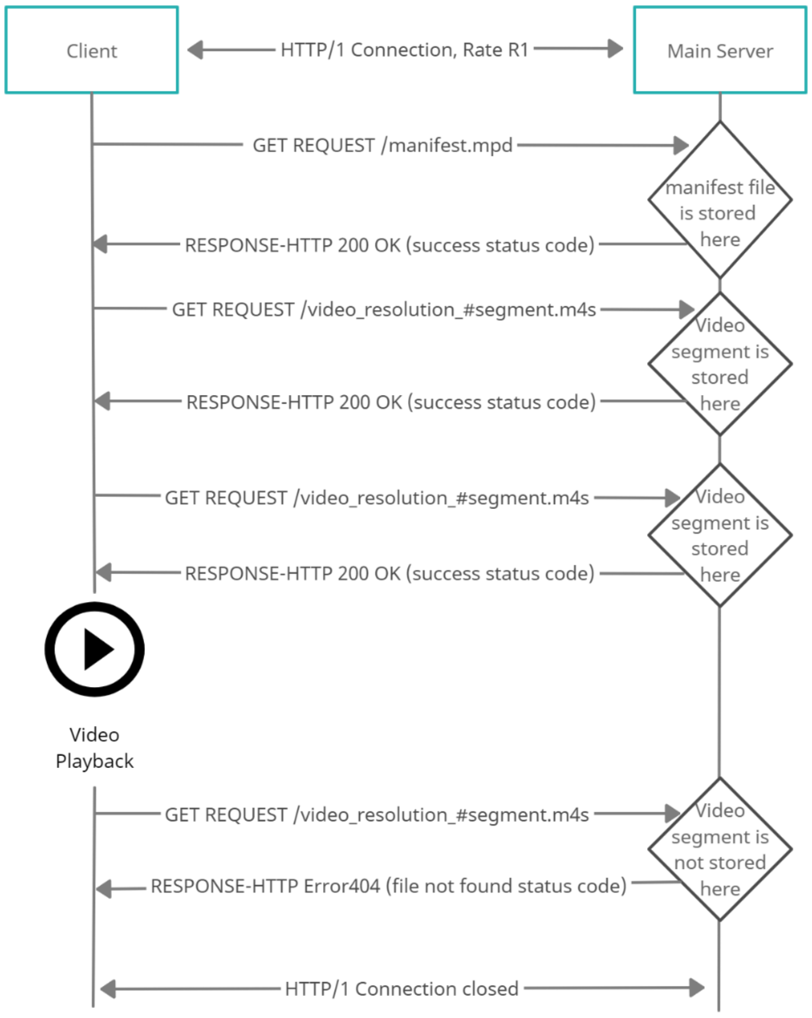

Client
HTTP/1 Connection, Rate R1
Main Server
GET REQUEST /manifest.mpd
manifest file is stored here
RESPONSE-HTTP 200 OK (success status code)
GET REQUEST /video_resolution_#segment.m4s
Video segment is stored here
RESPONSE-HTTP 200 OK (success status code)
GET REQUEST /video_resolution_#segment.m4s
Video segment is stored here
RESPONSE-HTTP 200 OK (success status code)
Video Playback
GET REQUEST /video_resolution_#segment.m4s
Video segment is not stored here
RESPONSE-HTTP Error404 (file not found status code)
HTTP/1 Connection closed

Figure 41: Flow of Information without cache server

Now we are called to implement a more enhanced network topology, regarding the basic topology. The main actors in this implementation are three:

1) The client who wants to see a video at the best available quality.

2) The cache server, which has in store some of the video segment according to a caching algorithm and limited by its buffer size, as well as the manifest file.
3) The main server, which is equipped with all the video resolution chunks.

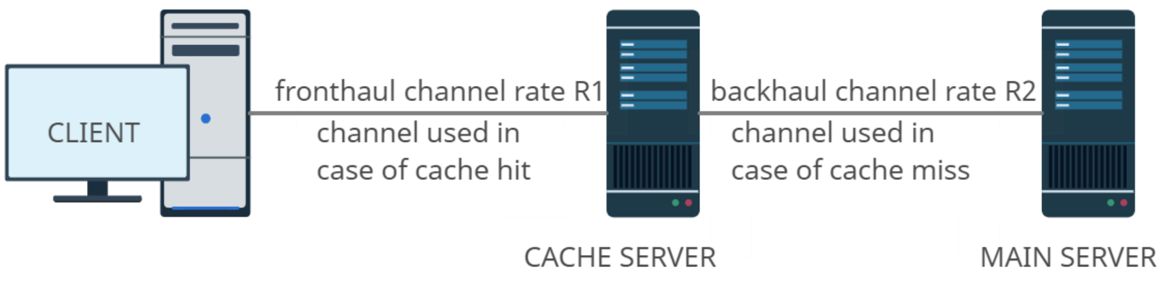

CLIENT
fronthaul channel rate R1
channel used in case of cache hit
backhaul channel rate R2
channel used in case of cache miss
CACHE SERVER
MAIN SERVER

Figure 42: Implementation with an intermediate cache server

The cache server is designed to be a simple repository for our video content, providing local clients with accelerated access to cached files. The main server is intended to provide a content management service which runs entirely behind the user interface of our custom application, creating the backhaul connection with the cache server, and containing all the given resolutions and the coding rates. In fact, the main server acts the same as the one of the first setup, but it doesn’t interact directly with the client.

Specifically, the cache server will contain a few segments from some of the video resolutions, meaning that it will have some caching capabilities. So, the basic connection will be between the client and the cache server. The backhaul connection, between the cache and the main server, will be utilized so the cache server can be able to receive chunks with a fixed rate from the main server, that contains all the segments. The number of the chunks that will be stored in the cache server depends on its available buffer size L and the implemented caching algorithm. The link rate between client and cache will be referred as R1 and the link rate between cache and main server as R2.

The roles of the actors of our problem statement are the following:

- **Client**: The client requests a video. The video has to be displayed in the best resolution, based on the available bandwidth. Practically, the client requests the manifest file from the cache server which contains all the available video details. For this reason, the cache must have available information about the manifest.
- **Cache Server**: The cache server has some chunks of the video in specific resolutions (according to the caching policies that we will introduce later). It interacts both with the client at one end and with the main server at the other end. The cache server as we have already stated, needs to set up a session with the main server that has all the video chunks in order to cache as many chunks as it can. As long as the segments are stored in the cache, the client requests the video and a new fronthaul connection with the cache server is set up. Each time a request is received, the cache checks the cache buffer and if there is a cache hit, the cache server sends it directly to the client and the channel cache-main is not utilized at all. Otherwise, in the case of a cache failure, the cache asks from the main, the wanted chunk in order to serve the client’s request.
- **Main Server**: The main server has all the possible resolutions of the video (360p, 480p, 720p, 1080p,1440p, 2160p) that are defined by the manifest file. This server interacts only with the cache server providing specific video chunks. In setup 1, its role is to give the best quality of the video to the client according to the bandwidth. In

setup 2, its role is only to give the manifest file to the cache (only in cases that the cache isn't already equipped with it) and serve the client when there is a cache miss. In case that the cache has all the available chunks of the video, meaning that it can fully serve the client, the backhaul link is not used at all.

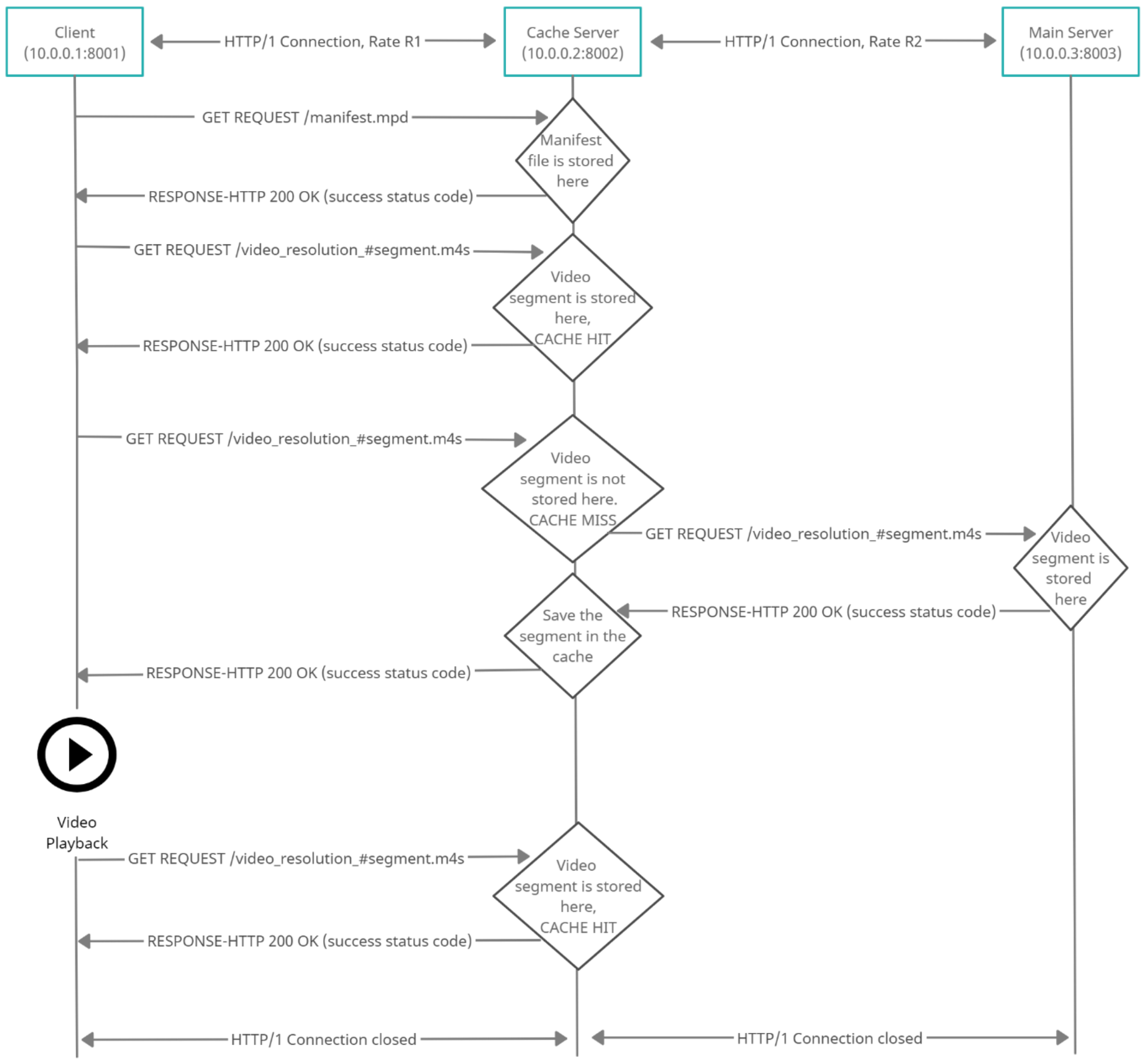

Client (10.0.0.1:8001)
HTTP/1 Connection, Rate R1
Cache Server (10.0.0.2:8002)
HTTP/1 Connection, Rate R2
Main Server (10.0.0.3:8003)
GET REQUEST /manifest.mpd
Manifest file is stored here
RESPONSE-HTTP 200 OK (success status code)
GET REQUEST /video_resolution_#segment.m4s
Video segment is stored here, CACHE HIT
RESPONSE-HTTP 200 OK (success status code)
GET REQUEST /video_resolution_#segment.m4s
Video segment is not stored here. CACHE MISS
GET REQUEST /video_resolution_#segment.m4s
Video segment is stored here
RESPONSE-HTTP 200 OK (success status code)
Save the segment in the cache
RESPONSE-HTTP 200 OK (success status code)
Video Playback
GET REQUEST /video_resolution_#segment.m4s
Video segment is stored here, CACHE HIT
RESPONSE-HTTP 200 OK (success status code)
HTTP/1 Connection closed
HTTP/1 Connection closed

Figure 43: Flow of information with an intermediate cache server

6.3.2 CacheHTTPServer module

Based on our knowledge gained by the SimpleHTTPServer in the previous section for the basic implementation, we use the above code for the cache server, with an objective of adding caching capabilities. So, we have modified the module of the SimpleHTTPServer, which is given as an open-source by Python 2.7, to be able to serve as a cache-enabled device. We create a file as "CacheHTTPServer.py" and copy the code of the SimpleHTTPServer in it, in order to change it. The highlighted parts are the personal additions to the default code. They are included in a tag named "#THESIS DI" for further understanding.

First of all, we have imported the requests module, which allows us to send HTTP requests using Python. The HTTP request returns a Response Object with all the response data (content, encoding, status, etc.). We have analyzed the requests module further in previous chapters. The next part of the code is unchanged.

In the try-except part, we add the appropriate code block, included in the “#THESIS DI” comment. In general, what we have tried to do is, when a file is not found (404 error) in the cache, it will not return None as before, but instead, it will print that we have come up with a “CACHE MISS” case. So, when the file is not in the cache, we search the path, in order to extract the filename. In the variable “splitted”, we save the last part from the full path, which is the name of the segment. Then, we will request this file from the main server, by using the IP and the port of the main server (10.0.0.3:8003), followed by the name of the segment that was not found in the cache (splitted). With the open command, the segment is copied locally in the cache, so it downloads the specific file that was not found before. Lastly, the 3 last lines are what the code used to do before our additions, so they are commented out.

```
try:
            # Always read in binary mode. Opening files in text mode may cause
            # newline translations, making the actual size of the content
            # transmitted *less* than the content-length!
            print("Correct Directory Path:")
            print(path + "\n")
            f = open(path, 'rb')
        except IOError:
            # THESIS DI ...
            print('CACHE MISS')
            print("Path:")
            print(path + "\n")
            splitted = path.rsplit('/', 1)[1]
            print("Splitted Path:")
            print(splitted + "\n")

            url_IP = "http://10.0.0.3:8003/"
            url = url_IP + splitted;
            r = requests.get(url)
            open(splitted, 'wb').write(r.content)
            # print len(r.content)
            # self.send_error(404, "File not found")
            # return None
```

The next highlighted code block, starting from the “ if os.path.isdir(path):” and ending in the tag “# THESIS DI” is copied from above so that in the case of a 404 error, instead of returning None, it should return normally the new path for the segment not found in the cache.

```
    if os.path.isdir(path):
                parts = urlparse.urlsplit(self.path)
```

```
            if not parts.path.endswith('/'):

                # redirect browser - doing basically what apache does
                self.send_response(301)
                new_parts = (parts[0], parts[1], parts[2] + '/',
                             parts[3], parts[4])

                new_url = urlparse.urlunsplit(new_parts)

                self.send_header("Location", new_url)
                self.end_headers()
                return None
            for index in "index.html", "index.htm":
                index = os.path.join(path, index)

                if os.path.exists(index):
                    path = index
                    break
            else:

                return self.list_directory(path)
        ctype = self.guess_type(path)
        f = open(path, 'rb')
        # ... THESIS DI
    try:
        self.send_response(200)
        self.send_header("Content-type", ctype)
        fs = os.fstat(f.fileno())
        self.send_header("Content-Length", str(fs[6]))
        self.send_header("Last-Modified", self.date_time_string(fs.st_mtime))
        self.end_headers()
        return f
    except:
        f.close()
        raise
```

The rest of the code is identical.

6.3.3 Caching Algorithms

As we have described above, another factor in conducting the experiments to extract the wanted metrics is the caching algorithms. For this Thesis, we have a simple random caching algorithm. It downloads segments and saves them in the cache, as we will explain below. The buffer size can be changed depending on the capacity of the cache.

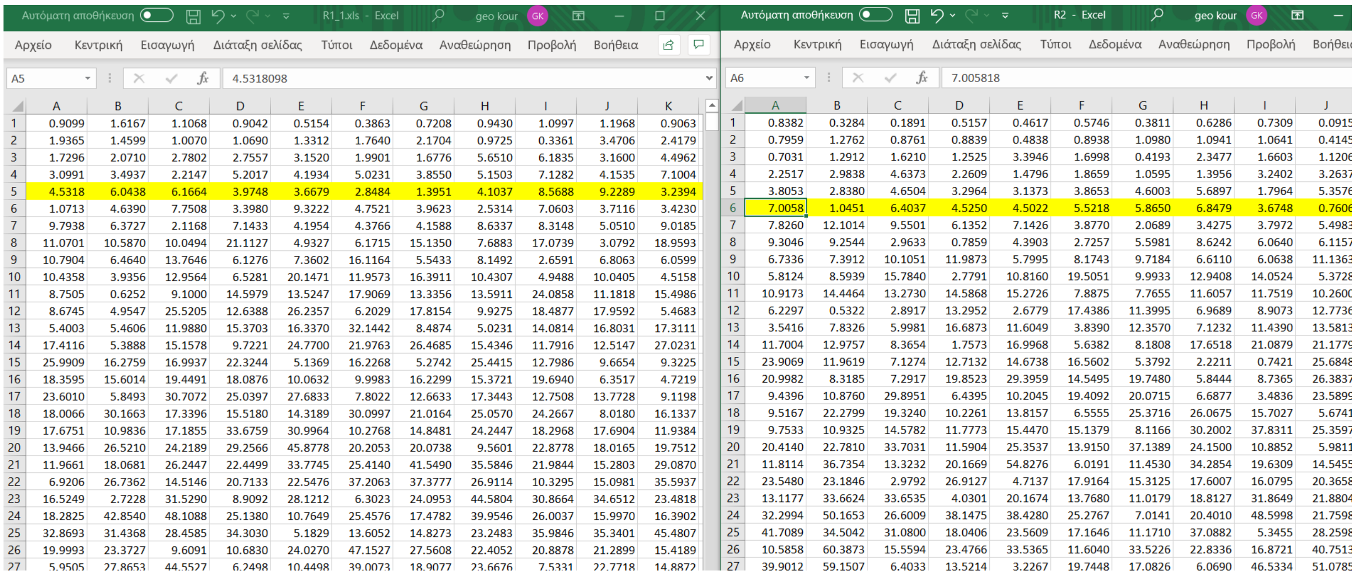
The algorithm generates two random numbers, one for the resolution and one for the segment. So, the range to choose for the resolution is 360, 480, 720, 1080, 1440 or 2160, and the range to choose for the segment is [1,61], because each resolution has 61 segments. After, randomly choosing the file, it checks if it fits into the buffer. If it fits, it copies in from the main server's public folder. If it doesn't it moves to the next one only if the available buffer size is less than the size of the smallest available non-cached segment.

In the code that we provide below, the buffer size is set to 1000MBs and that can change manually. The file is named random_file.sh and it is presented in the Appendix A as well.

6.3.4 Wireless Channel Emulation

If we type the command "links" in the Mininet's CLI, we will find out the interfaces that we will apply our rate rules in order to change the traffic conditions. These are the interfaces S2-eth2 for R1 and S3-eth2 for R2. Due to the fact that we want to simulate a wireless channel, the rates R1 and R2 will not be constant at all, but they will be distributed in a specific way. We will use two different Rayleigh distributions for R1, and one for R2 (figure 43,44). The files for the rates were given to us offline and they are extracted from a MATLAB simulation to touch real time values, and we use them in order to run our rate scripts.

To be more specific, every line of the of R1.xls files matches with a rate distribution. The first line is for 1 Mbps, the second for 2 Mbps, etc. For R2, the first line is for 0.5 Mbps, the second for 1 Mbps, the third for 2 Mbps, etc. Every cell is the current rate's value for 100ms and each next cell is for the next 100ms etc. The R1_1.xls presents a uniform Rayleigh distribution while the R1_2.xls simulates the existence of an obstacle in the middle of the distribution or a handover process. However, distributions have the same mean value.



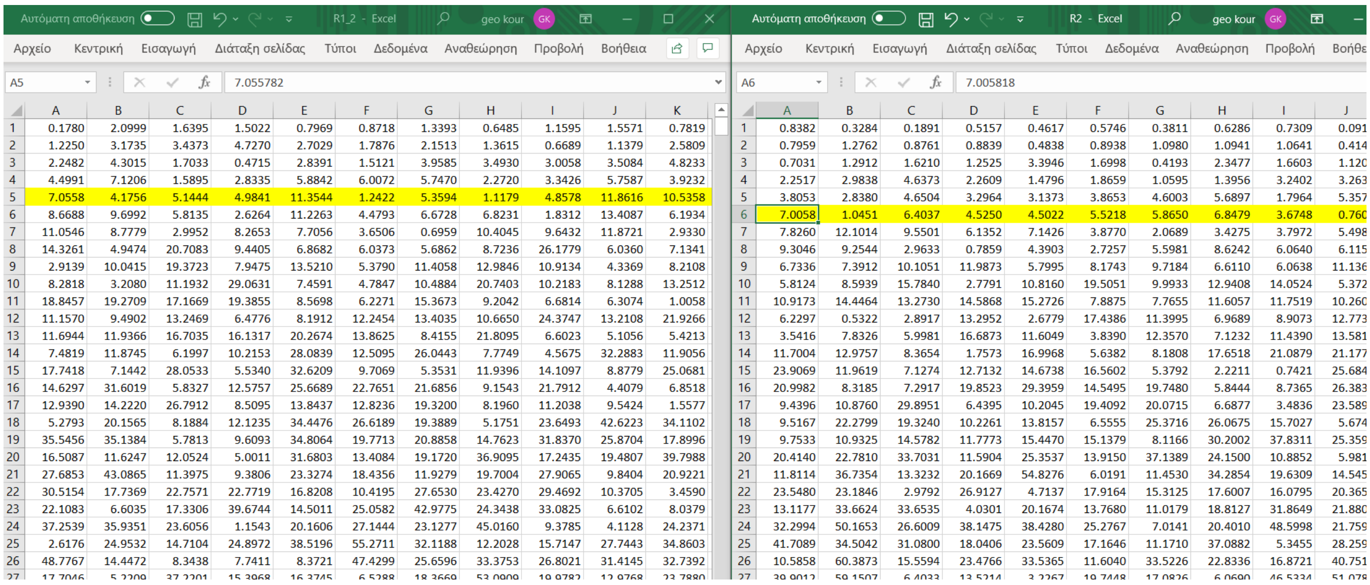
A5 4.5318098

	A	B	C	D	E	F	G	H	I	J	K
1	0.9099	1.6167	1.1068	0.9042	0.5154	0.3863	0.7208	0.9430	1.0997	1.1968	0.9063
2	1.9365	1.4599	1.0070	1.0690	1.3312	1.7640	2.1704	0.9725	0.3361	3.4706	2.4179
3	1.7296	2.0710	2.7802	2.7557	3.1520	1.9901	1.6776	5.6510	6.1835	3.1600	4.4962
4	3.0991	3.4937	2.2147	5.2017	4.1934	5.0231	3.8550	5.1503	7.1282	4.1535	7.1004
5	4.5318	6.0438	6.1664	3.9748	3.6679	2.8484	1.3951	4.1037	8.5688	9.2289	3.2394
6	1.0713	4.6390	7.7508	3.3980	9.3222	4.7521	3.9623	2.5314	7.0603	3.7116	3.4230
7	9.7938	6.3727	2.1168	7.1433	4.1954	4.3766	4.1588	8.6337	8.3148	5.0510	9.0185
8	11.0701	10.5870	10.0494	21.1127	4.9327	6.1715	15.1350	7.6883	17.0739	3.0792	18.9593
9	10.7904	6.4640	13.7646	6.1276	7.3602	16.1164	5.5433	8.1492	2.6591	6.8063	6.0599
10	10.4358	3.9356	12.9564	6.5281	20.1471	11.9573	16.3911	10.4307	4.9488	10.0405	4.5158
11	8.7505	0.6252	9.1000	14.5979	13.5247	17.9069	13.3356	13.5911	24.0858	11.1818	15.4986
12	8.6745	4.9547	25.5205	12.6388	26.2357	6.2029	17.8154	9.9275	18.4877	17.9592	5.4683
13	5.4003	5.4606	11.9880	15.3703	16.3370	32.1442	8.4874	5.0231	14.0814	16.8031	17.3111
14	17.4116	5.3888	15.1578	9.7221	24.7700	21.9763	26.4685	15.4346	11.7916	12.5147	27.0231
15	25.9909	16.2759	16.9937	22.3244	5.1369	16.2268	5.2742	25.4415	12.7986	9.6654	9.3225
16	18.3595	15.6014	19.4491	18.0876	10.0632	9.9983	16.2299	15.3721	19.6940	6.3517	4.7219
17	23.6010	5.8493	30.7072	25.0397	27.6833	7.8022	12.6633	17.3443	12.7508	13.7728	9.1198
18	18.0066	30.1663	17.3396	15.5180	14.3189	30.0997	21.0164	25.0570	24.2667	8.0180	16.1337
19	17.6751	10.9836	17.1855	33.6759	30.9964	10.2768	14.8481	24.2447	18.2968	17.6904	11.9384
20	13.9466	26.5210	24.2189	29.2566	45.8778	20.2053	20.0738	9.5601	22.8778	18.0165	19.7512
21	11.9661	18.0681	26.2447	22.4499	33.7745	25.4140	41.5490	35.5846	21.9844	15.2803	29.0870
22	6.9206	26.7362	14.5146	20.7133	22.5476	37.2063	37.3777	26.9114	10.3295	15.0981	35.5937
23	16.5249	2.7228	31.5290	8.9092	28.1212	6.3023	24.0953	44.5804	30.8664	34.6512	23.4818
24	18.2825	42.8540	48.1088	25.1380	10.7649	25.4576	17.4782	39.9546	26.0037	15.9970	16.3902
25	32.8693	31.4368	28.4585	34.3030	5.1829	13.6052	14.8273	23.2483	35.9846	35.3401	45.4807
26	19.9993	23.3727	9.6091	10.6830	24.0270	47.1527	27.5608	22.4052	20.8878	21.2899	15.4189
27	5.9505	27.8653	44.5527	6.2498	10.4498	39.0073	18.9077	23.6676	7.5331	22.7718	14.8872

A6 7.005818

	A	B	C	D	E	F	G	H	I	J
1	0.8382	0.3284	0.1891	0.5157	0.4617	0.5746	0.3811	0.6286	0.7309	0.0915
2	0.7959	1.2762	0.8761	0.8839	0.4838	0.8938	1.0980	1.0941	1.0641	0.4145
3	0.7031	1.2912	1.6210	1.2525	3.3946	1.6998	0.4193	2.3477	1.6603	1.1206
4	2.2517	2.9838	4.6373	2.2609	1.4796	1.8659	1.0595	1.3956	3.2402	3.2637
5	3.8053	2.8380	4.6504	3.2964	3.1373	3.8653	4.6003	5.6897	1.7964	5.3576
6	7.0058	1.0451	6.4037	4.5250	4.5022	5.5218	5.8650	6.8479	3.6748	0.7606
7	7.8260	12.1014	9.5501	6.1352	7.1426	3.8770	2.0689	3.4275	3.7972	5.4983
8	9.3046	9.2544	2.9633	0.7859	4.3903	2.7257	5.5981	8.6242	6.0640	6.1157
9	6.7336	7.3912	10.1051	11.9873	5.7995	8.1743	9.7184	6.6110	6.0638	11.1363
10	5.8124	8.5939	15.7840	2.7791	10.8160	19.5051	9.9933	12.9408	14.0524	5.3728
11	10.9173	14.4464	13.2730	14.5868	15.2726	7.8875	7.7655	11.6057	11.7519	10.2600
12	6.2297	0.5322	2.8917	13.2952	2.6779	17.4386	11.3995	6.9689	8.9073	12.7736
13	3.5416	7.8326	5.9981	16.6873	11.6049	3.8390	12.3570	7.1232	11.4390	13.5813
14	11.7004	12.9757	8.3654	1.7573	16.9968	5.6382	8.1808	17.6518	21.0879	21.1779
15	23.9069	11.9619	7.1274	12.7132	14.6738	16.5602	5.3792	2.2211	0.7421	25.6848
16	20.9982	8.3185	7.2917	19.8523	29.3959	14.5495	19.7480	5.8444	8.7365	26.3837
17	9.4396	10.8760	29.8951	6.4395	10.2045	19.4092	20.0715	6.6877	3.4836	23.5899
18	9.5167	22.2799	19.3240	10.2261	13.8157	6.5555	25.3716	26.0675	15.7027	5.6741
19	9.7533	10.9325	14.5782	11.7773	15.4470	15.1379	8.1166	30.2002	37.8311	25.3597
20	20.4140	22.7810	33.7031	11.5904	25.3537	13.9150	37.1389	24.1500	10.8852	5.9811
21	11.8114	36.7354	13.3232	20.1669	54.8276	6.0191	11.4530	34.2854	19.6309	14.5455
22	23.5480	23.1846	2.9792	26.9127	4.7137	17.9164	15.3125	17.6007	16.0795	20.3658
23	13.1177	33.6624	33.6535	4.0301	20.1674	13.7680	11.0179	18.8127	31.8649	21.8804
24	32.2994	50.1653	26.6009	38.1475	38.4280	25.2767	7.0141	20.4010	48.5998	21.7598
25	41.7089	34.5042	31.0800	18.0406	23.5609	17.1646	11.1710	37.0882	5.3455	28.2598
26	10.5858	60.3873	15.5594	23.4766	33.5365	11.6040	33.5226	22.8336	16.8721	40.7513
27	39.9012	59.1507	6.4033	13.5214	3.2267	19.7448	17.0826	6.0690	46.5334	51.0785

Figure 44: Rate Traces for R1 = R2 = 5 Mbps (from R1_1.xls and R2.xls)



R1_2 - Excel (A5: 7.055782)

	A	B	C	D	E	F	G	H	I	J	K
1	0.1780	2.0999	1.6395	1.5022	0.7969	0.8718	1.3393	0.6485	1.1595	1.5571	0.7819
2	1.2250	3.1735	3.4373	4.7270	2.7029	1.7876	2.1513	1.3615	0.6689	1.1379	2.5809
3	2.2482	4.3015	1.7033	0.4715	2.8391	1.5121	3.9585	3.4930	3.0058	3.5084	4.8233
4	4.4991	7.1206	1.5895	2.8335	5.8842	6.0072	5.7470	2.2720	3.3426	5.7587	3.9232
5	7.0558	4.1756	5.1444	4.9841	11.3544	1.2422	5.3594	1.1179	4.8578	11.8616	10.5358
6	8.6688	9.6992	5.8135	2.6264	11.2263	4.4793	6.6728	6.8231	1.8312	13.4087	6.1934
7	11.0546	8.7779	2.9952	8.2653	7.7056	3.6506	0.6959	10.4045	9.6432	11.8721	2.9330
8	14.3261	4.9474	20.7083	9.4405	6.8682	6.0373	5.6862	8.7236	26.1779	6.0360	7.1341
9	2.9139	10.0415	19.3723	7.9475	13.5210	5.3790	11.4058	12.9846	10.9134	4.3369	8.2108
10	8.2818	3.2080	11.1932	29.0631	7.4591	4.7847	10.4884	20.7403	10.2183	8.1288	13.2512
11	18.8457	19.2709	17.1669	19.3855	8.5698	6.2271	15.3673	9.2042	6.6814	6.3074	1.0058
12	11.1570	9.4902	13.2469	6.4776	8.1912	12.2454	13.4035	10.6650	24.3747	13.2108	21.9266
13	11.6944	11.9366	16.7035	16.1317	20.2674	13.8625	8.4155	21.8095	6.6023	5.1056	5.4213
14	7.4819	11.8745	6.1997	10.2153	28.0839	12.5095	26.0443	7.7749	4.5675	32.2883	11.9056
15	17.7418	7.1442	28.0533	5.5340	32.6209	9.7069	5.3531	11.9396	14.1097	8.8779	25.0681
16	14.6297	31.6019	5.8327	12.5757	25.6689	22.7651	21.6856	9.1543	21.7912	4.4079	6.8518
17	12.9390	14.2220	26.7912	8.5095	13.8437	12.8236	19.3200	8.1960	11.2038	9.5424	1.5577
18	5.2793	20.1565	8.1884	12.1235	34.4476	26.6189	19.3889	5.1751	23.6493	42.6223	34.1102
19	35.5456	35.1384	5.7813	9.6093	34.8064	19.7713	20.8858	14.7623	31.8370	25.8704	17.8996
20	16.5087	11.6247	12.0524	5.0011	31.6803	13.4084	19.1720	36.9095	17.2435	19.4807	39.7988
21	27.6853	43.0865	11.3975	9.3806	23.3274	18.4356	11.9279	19.7004	27.9065	9.8404	20.9221
22	30.5154	17.7369	22.7571	22.7719	16.8208	10.4195	27.6530	23.4270	29.4692	10.3705	3.4590
23	22.1083	6.6035	17.3306	39.6744	14.5011	25.0582	42.9775	24.3438	33.0825	6.6102	8.0379
24	37.2539	35.9351	23.6056	1.1543	20.1606	27.1444	23.1277	45.0160	9.3785	4.1128	24.2371
25	2.6176	24.9532	14.7104	24.8972	38.5196	55.2711	32.1188	12.2028	15.7147	27.7443	34.8603
26	48.7767	14.4472	8.3438	7.7411	8.3721	47.4299	25.6596	33.3753	26.8021	31.4145	32.7392
27	17.7046	5.2209	37.2201	15.3968	16.3745	6.5288	18.3669	53.0909	19.9782	12.9768	23.7880

R2 - Excel (A6: 7.005818)

	A	B	C	D	E	F	G	H	I	J
1	0.8382	0.3284	0.1891	0.5157	0.4617	0.5746	0.3811	0.6286	0.7309	0.091
2	0.7959	1.2762	0.8761	0.8839	0.4838	0.8938	1.0980	1.0941	1.0641	0.414
3	0.7031	1.2912	1.6210	1.2525	3.3946	1.6998	0.4193	2.3477	1.6603	1.120
4	2.2517	2.9838	4.6373	2.2609	1.4796	1.8659	1.0595	1.3956	3.2402	3.263
5	3.8053	2.8380	4.6504	3.2964	3.1373	3.8653	4.6003	5.6897	1.7964	5.357
6	7.0058	1.0451	6.4037	4.5250	4.5022	5.5218	5.8650	6.8479	3.6748	0.760
7	7.8260	12.1014	9.5501	6.1352	7.1426	3.8770	2.0689	3.4275	3.7972	5.498
8	9.3046	9.2544	2.9633	0.7859	4.3903	2.7257	5.5981	8.6242	6.0640	6.115
9	6.7336	7.3912	10.1051	11.9873	5.7995	8.1743	9.7184	6.6110	6.0638	11.136
10	5.8124	8.5939	15.7840	2.7791	10.8160	19.5051	9.9933	12.9408	14.0524	5.372
11	10.9173	14.4464	13.2730	14.5868	15.2726	7.8875	7.7655	11.6057	11.7519	10.260
12	6.2297	0.5322	2.8917	13.2952	2.6779	17.4386	11.3995	6.9689	8.9073	12.773
13	3.5416	7.8326	5.9981	16.6873	11.6049	3.8390	12.3570	7.1232	11.4390	13.581
14	11.7004	12.9757	8.3654	1.7573	16.9968	5.6382	8.1808	17.6518	21.0879	21.177
15	23.9069	11.9619	7.1274	12.7132	14.6738	16.5602	5.3792	2.2211	0.7421	25.684
16	20.9982	8.3185	7.2917	19.8523	29.3959	14.5495	19.7480	5.8444	8.7365	26.383
17	9.4396	10.8760	29.8951	6.4395	10.2045	19.4092	20.0715	6.6877	3.4836	23.589
18	9.5167	22.2799	19.3240	10.2261	13.8157	6.5555	25.3716	26.0675	15.7027	5.674
19	9.7533	10.9325	14.5782	11.7773	15.4470	15.1379	8.1166	30.2002	37.8311	25.359
20	20.4140	22.7810	33.7031	11.5904	25.3537	13.9150	37.1389	24.1500	10.8852	5.981
21	11.8114	36.7354	13.3232	20.1669	54.8276	6.0191	11.4530	34.2854	19.6309	14.545
22	23.5480	23.1846	2.9792	26.9127	4.7137	17.9164	15.3125	17.6007	16.0795	20.365
23	13.1177	33.6624	33.6535	4.0301	20.1674	13.7680	11.0179	18.8127	31.8649	21.880
24	32.2994	50.1653	26.6009	38.1475	38.4280	25.2767	7.0141	20.4010	48.5998	21.759
25	41.7089	34.5042	31.0800	18.0406	23.5609	17.1646	11.1710	37.0882	5.3455	28.259
26	10.5858	60.3873	15.5594	23.4766	33.5365	11.6040	33.5226	22.8336	16.8721	40.751
27	39.9012	59.1507	6.4033	13.5214	3.2267	19.7448	17.0826	6.0690	46.5334	51.078

Figure 45: Rate Traces for R1 = R2 = 5 Mbps (from R1_2.xls and R2.xls)

In order to implement the rate scripts, we present in the Appendix A the core body of the code for the rate R1. The code will be the same for the R2 with the difference that we have to apply the correct rate distribution in the corresponding interface. Inside the parenthesis of the rates variable in the 3rd line of the script we copy and paste the corresponding line of excel sheet (like the one that it is presented in Figure 44 if we want R1=5Mbps).

6.3.5 Metrics

In order to exploit the data of the experiments that we conducted, we have collected some metrics that are relative to the QoE and we present them below.

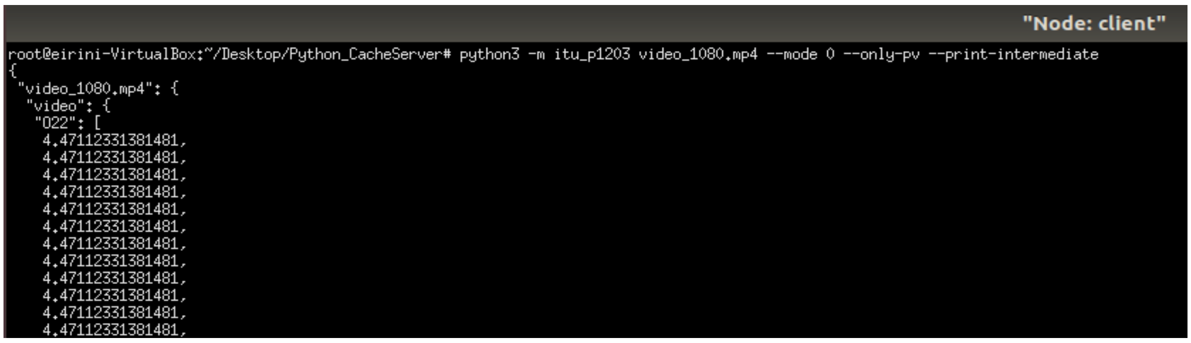
- The **Average Resolution** refers to the average over all segments that have been selected by the media player, where each segment can take resolution values from this set {2160, 1440, 1080, 720, 480, 360}.
- Sum of Switches refers to the sum of changes of resolutions that has occurred during the video playback.
- The **Average Altitude** refers to the average value of the jumps between the previous and the current segment over all segments that have been selected by the VLC, where each segment can take altitude values from the set [0,6]. For example, if the first segment is 480p and the second is 1440p, then the altitude, or the jumps, are 3.
- The **Average Video Bitrate** (Mbps) refers to the average bitrate over all segments that have been selected by the media player, where each segment size is divided by 5 seconds, which is its duration.
- The **Network Usage Time** in seconds refers to the duration of the video playback, from the time of delivery of the first segment until the time of delivery of the last segment.
- The **Mean Network Throughput R1** (Mbps) refers to the size of all the segments (cached and non-cached) that have been selected by the media player over the network usage time.
- The **Mean Network Throughput R2** (Mbps) refers to the size of all the segments not stored in the cache, over the network usage time.
- The **Mean Opinion Score (MOS) for stallings** refers to the numerical measure of the overall quality of an event, system or experience in networks and telecommunications. It is a crucial parameter for the domain of QoE and is expressed as a single rational number, typically in the range 1–5, where 1 is lowest perceived

quality, and 5 is the highest perceived one. This MOS value, based on stallings, is computed with the formula : $MOS_{stallings} = 3.5 * e^{-(0.15M+0.19)N} + 1.5$, where M = mean stalling duration and N = number of stallings [69].

- The **Mean Opinion Score (MOS) for resolutions** has been defined by ITU-T in several ways. We will use the ITU-T p.1203 Standalone Implementation for the MOS based on resolutions. In order to use this tool, we install it from the official repository from GitHub: https://github.com/itu-p1203/itu-p1203.The ITU-T Rec. P.1203.1 model is restricted to information provided to it by an appropriate bit stream analysis module or set of modules. The model is applicable to progressive download and adaptive streaming, where the quality experienced by the end user is affected by audio-coding and/or video degradations due to coding, spatial re-scaling or variations in video frame rates, as well as delivery degradations due to initial loading delay, stalling, and media adaptations [70].
  The model's input consists of information that are obtained by a prior stream inspection. There are four different modes of inspection which result in models of different complexity. We will use the Mode 0 where the information is obtained from available metadata during the adaptive streaming (for example from manifest files used in DASH, about codec and bitrate, and initial loading delay and stalling). Due to our mode, we will not have score fluctuations among the same resolution.
  An example of how we ran offline QoE evaluation is shown in the next command:

```
$ python3 -m itu_p1203 input --mode 0 --only-pv --print-intermediate
```

  The output of the command for one of our video resolutions is shown in the next figure. The tool will calculate per second video scores. We will execute this command six times in order to extract the MOS of all the available resolutions. As a result, each resolution is matched with a MOS score.



```
"Node: client"
root@eirini-VirtualBox:~/Desktop/Python_CacheServer# python3 -m itu_p1203 video_1080.mp4 --mode 0 --only-pv --print-intermediate
{
 "video_1080.mp4": {
  "video": {
   "O22": [
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
    4.47112331381481,
```

Figure 46: MOS of a 1080p video

The MOS values that we have extracted for the "mode 0" of the tool are the following:

Table 5: MOS of the available resolutions

Resolution	MOS (Mean Opinion Score)
2160p	4.58036
1440p	4.52586
1080p	4.47112
720p	3.97185
480p	3.02246
360p	2.07744

Compared to the MOS based on stallings, we will not use the MOS based on resolutions because the QoE depends mostly on how much the video freezes and not on the resolution changes. For example, it is preferred to watch a video on 720p without it freezing all the time, than on 1440p with many stallings.

- The **Number of Stallings** refers to the sum of stalling events. A stalling occurs when a segment has been delivered in more than 5 seconds, which is its duration.
- The **Total Stalling Time** refers to the sum of stalling durations of all the segments.
- The **Mean Stalling Time** refers to the total stalling time over all stallings occurred in a video playback.
- The **Initial Playback Delay** in seconds, refers to the time of delivery of the first segment from the time of the arrival of the manifest file, which is the start of the experiment.
- The **Number of Cached Bits** delivered refers to the sum of size in bits of the segments that were already stored in the cache.
- Number of Non-Cached bits delivered refers to the sum of size in bits of the segments that were not stored in the cache but were served by the main server.
- Last, the **Cache Hit Ratio** refers to the sum of segments that were already in the cache over all the segments cached and non-cached, in percentage.

6.3.6 Raw file

In order to extract metrics from the output of experiment, we have made a bash shell script called “raw.sh” that takes as an input the output of the cache server and measures all the metrics that we need. To collect all the metrics that we want, we export the output of the cache server into a text file, named metrics.txt.

All those metrics are written in csv files, for a better manipulation to make the diagrams. The code of the raw.sh is presented in the Appendix A.

6.3.7 Implementation

Everything that was explained and described in the previous sections, will now be implemented. The process of producing the segments of the video for each quality version and the manifest file that describes it all, has been completed already.

So, the custom topology for the client, cache, and main server, demonstrated with the use of MiniEdit, is shown below.

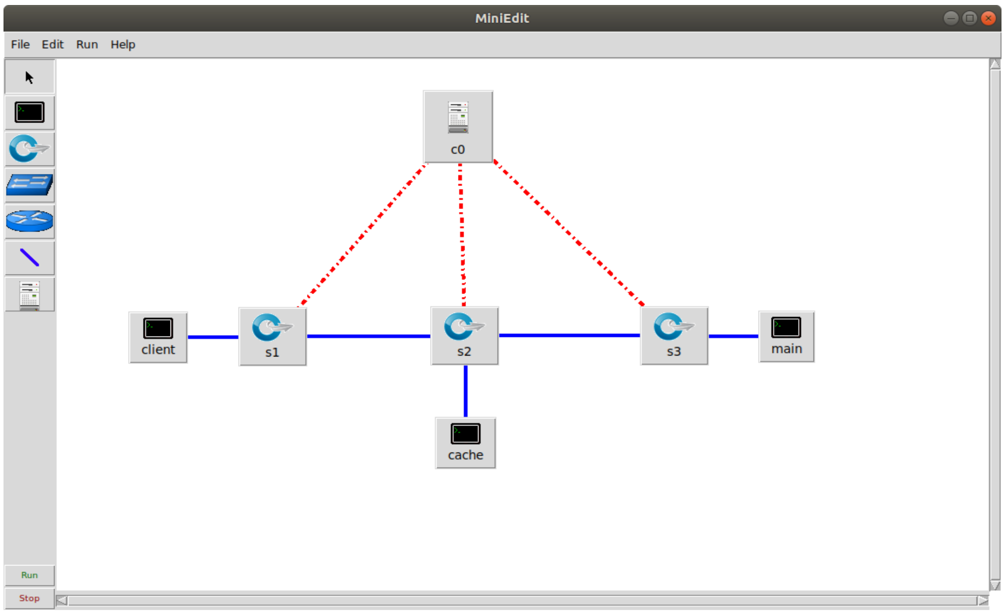
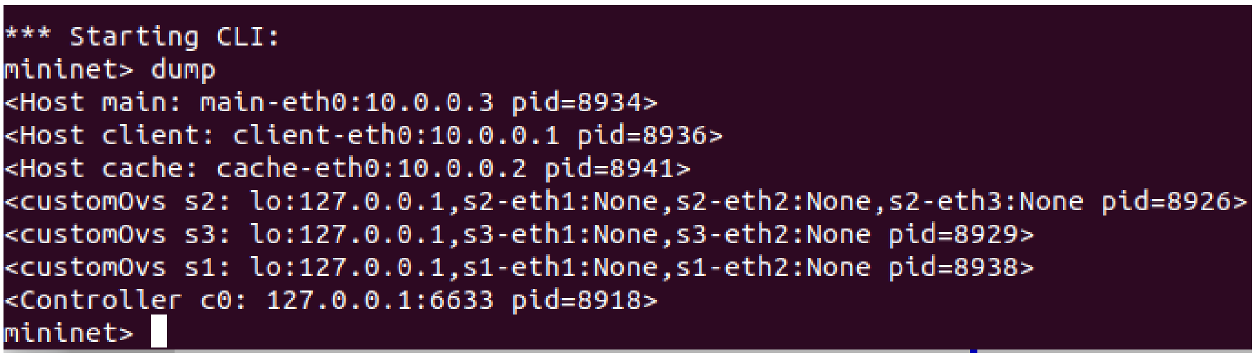


Figure 47: Enhanced topology with cache server

We have renamed the hosts h1, h2 and h3 to client, cache and main, accordingly, for our convenience. Having enabled the CLI option in Preferences, we hit the Run button in the lower left corner. In the Mininet CLI, we type the command "dump", in order to be presented with the IP's of each node. As we can see, the IP of the client is 10.0.0.1, the IP of the cache is 10.0.0.2, and the IP of the main server is 10.0.0.3.



```
*** Starting CLI:
mininet> dump
<Host main: main-eth0:10.0.0.3 pid=8934>
<Host client: client-eth0:10.0.0.1 pid=8936>
<Host cache: cache-eth0:10.0.0.2 pid=8941>
<customOvs s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None pid=8926>
<customOvs s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None pid=8929>
<customOvs s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=8938>
<Controller c0: 127.0.0.1:6633 pid=8918>
mininet> 
```

Figure 48: Starting the CLI

So, for our simulation, we have created two separate folders, one for the cache and one for the main server, the R1.sh script and the R2.sh script.

The main server's public folder contains the manifest file as well as every segment for each quality. So, the public folder, as it is its role, contains everything. On the other hand, the cache folder contains the "CacheHTTPServer.py" file in Python for the cache server, the manifest file, some video segments according to the appropriate caching policy and the raw.sh script.

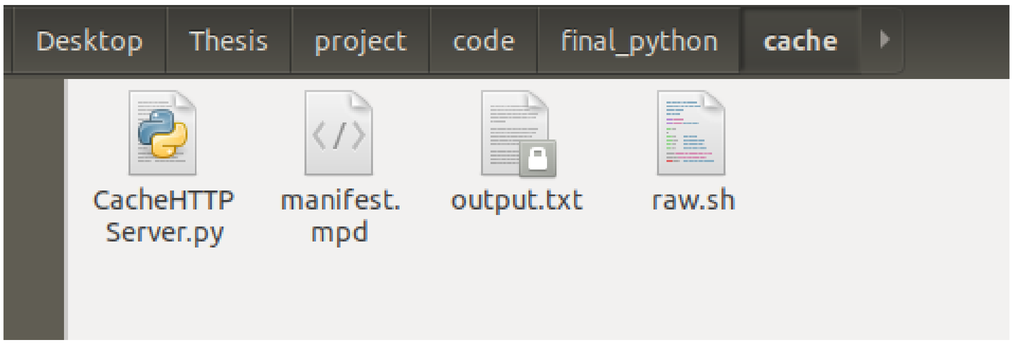


Figure 49: The cache folder (with no cache policy)

We must be sure that before the start of any experiment, we should clear the cache folder. If we wanted to use a caching policy, we would run the script random_file.sh to fill the cache , before the start of the experiment.

Next, we type the following command in the Mininet CLI, and the xterms for all the hosts will appear:

```
$ xterm Client Cache Main s2 s3
```

In each xterm, we execute the following commands that are shown in the figure below. First, in the main window, we navigate into the public folder and start running the main server in port 8003 with the command:

```
$ python -m SimpleHTTPServer 8003
```

Next, in the same way, we navigate into the cache folder and start the cache server to listen in port 8002 by typing the following command in the cache xterm:

```
$ python -m CacheHTTPServer 8002 &> output.txt
```

After we execute this command, the output.txt file is created in the cache, and all the metrics will be saved there. Now it's time to execute the scripts for the rate control by typing in s2

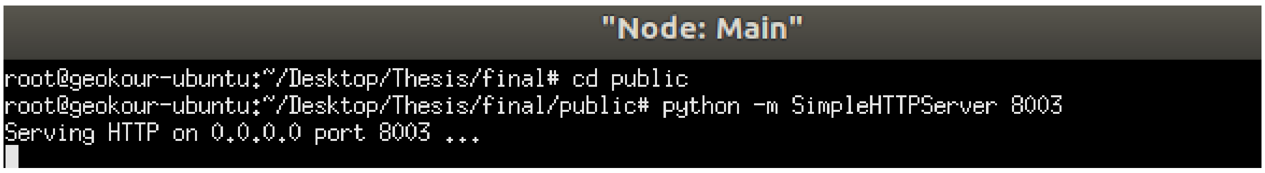
```
$ ./r1.sh
```

and in s3

```
$ ./r2.sh
```

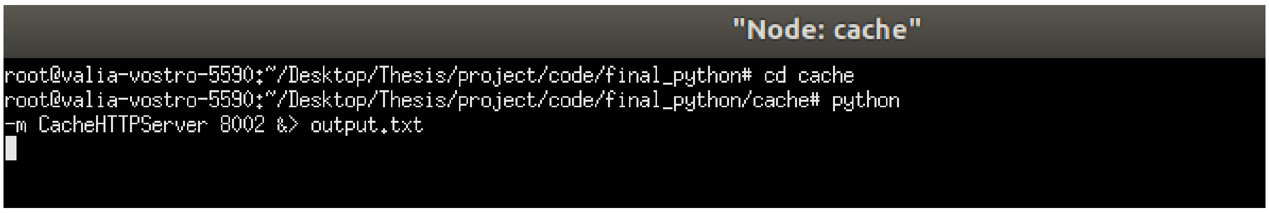
And, last but not least, we start the video player in the client xterm to request the manifest file from the IP and port of the cache server:

```
$ vlc-wrapper http://10.0.0.2:8002/manifest.mpd
```



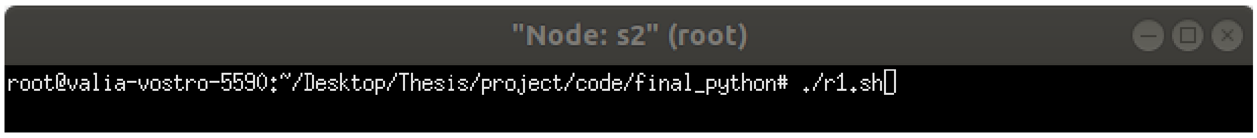
```
root@geokour-ubuntu:~/Desktop/Thesis/final# cd public
root@geokour-ubuntu:~/Desktop/Thesis/final/public# python -m SimpleHTTPServer 8003
Serving HTTP on 0.0.0.0 port 8003 ...
```

Figure 50: xterm Main Server



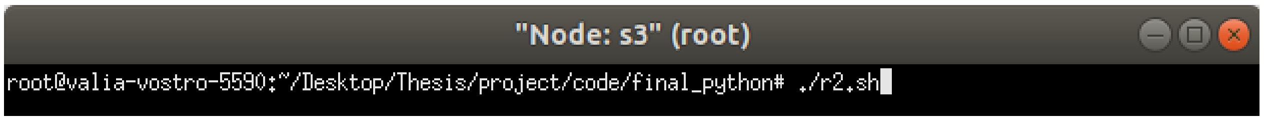
```
root@valia-vostro-5590:~/Desktop/Thesis/project/code/final_python# cd cache
root@valia-vostro-5590:~/Desktop/Thesis/project/code/final_python/cache# python
-m CacheHTTPServer 8002 &> output.txt
```

Figure 51: xterm Cache Server



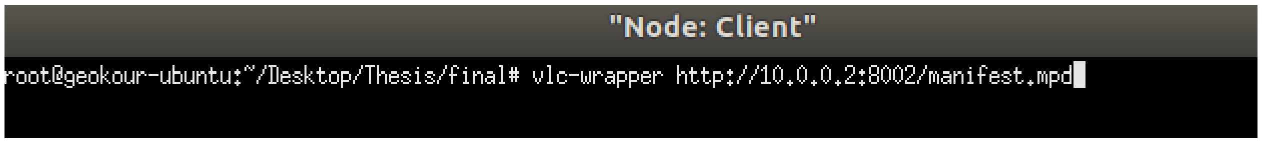
```
root@valia-vostro-5590:~/Desktop/Thesis/project/code/final_python# ./r1.sh
```

Figure 52 : xterm s2 switch



```
root@valia-vostro-5590:~/Desktop/Thesis/project/code/final_python# ./r2.sh
```

Figure 53: xterm s3 switch



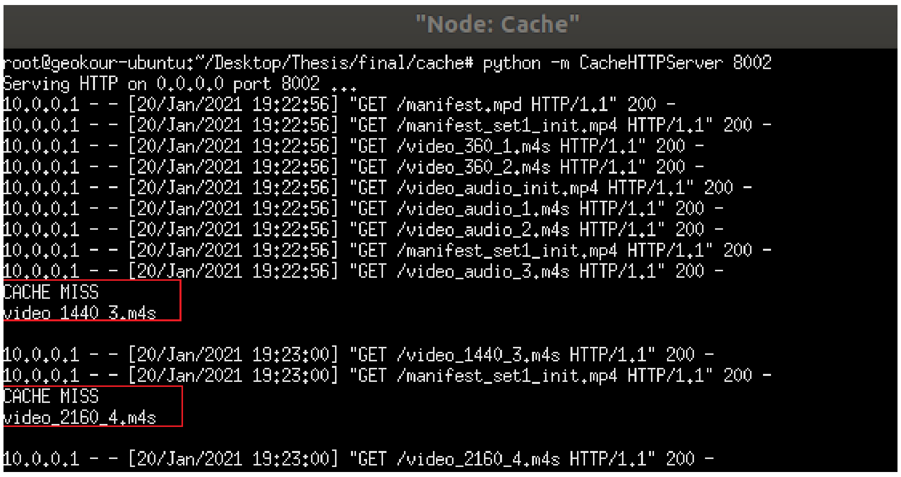
```
root@geokour-ubuntu:~/Desktop/Thesis/final# vlc-wrapper http://10.0.0.2:8002/manifest.mpd
```

Figure 54: xterm Client

After executing the command in the client window, we are able to watch the video.

Figure 55: Video playback

To follow the flow of information, we present the below two screenshots to show what is happening between the two servers. So, the video starts playing, by requesting the manifest from the cache server. Then, sequential requests for each segment are starting towards the cache server. The start is always with the lowest quality, so the initialization file is asked first, and the first segments (.m4s files) are asked second. Those files are all in the cache, so, we see the messages printed in the cache console. In the case where a file is requested by the client from the cache and the cache does not have it, the main server is enabled, and the cache requests it from it. In the red rectangle we see that after some segments that are found in the cache, there is a cache miss for the video_1440_3.m4s segment (same for video_2160_4.m4s), so it is requested from the main server. The main server responses with the correct file, and then the video is continued.

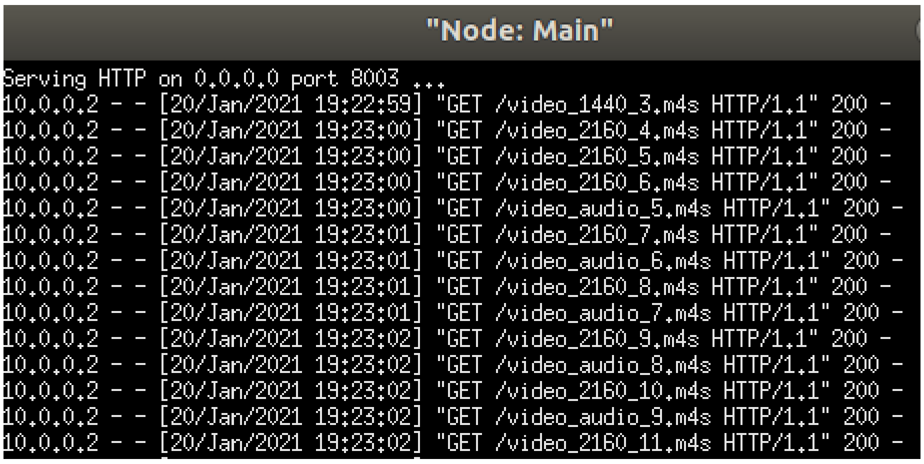


```
"Node: Cache"
root@geokour-ubuntu:~/Desktop/Thesis/final/cache# python -m CacheHTTPServer 8002
Serving HTTP on 0.0.0.0 port 8002 ...
10.0.0.1 - - [20/Jan/2021 19:22:56] "GET /manifest.mpd HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:22:56] "GET /manifest_set1_init.mp4 HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:22:56] "GET /video_360_1.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:22:56] "GET /video_360_2.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:22:56] "GET /video_audio_init.mp4 HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:22:56] "GET /video_audio_1.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:22:56] "GET /video_audio_2.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:22:56] "GET /manifest_set1_init.mp4 HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:22:56] "GET /video_audio_3.m4s HTTP/1.1" 200 -
CACHE MISS
video_1440_3.m4s

10.0.0.1 - - [20/Jan/2021 19:23:00] "GET /video_1440_3.m4s HTTP/1.1" 200 -
10.0.0.1 - - [20/Jan/2021 19:23:00] "GET /manifest_set1_init.mp4 HTTP/1.1" 200 -
CACHE MISS
video_2160_4.m4s

10.0.0.1 - - [20/Jan/2021 19:23:00] "GET /video_2160_4.m4s HTTP/1.1" 200 -
```

Figure 56: Output of the cache server terminal



```
"Node: Main"
Serving HTTP on 0.0.0.0 port 8003 ...
10.0.0.2 - - [20/Jan/2021 19:22:59] "GET /video_1440_3.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:00] "GET /video_2160_4.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:00] "GET /video_2160_5.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:00] "GET /video_2160_6.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:00] "GET /video_audio_5.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:01] "GET /video_2160_7.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:01] "GET /video_audio_6.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:01] "GET /video_2160_8.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:01] "GET /video_audio_7.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:02] "GET /video_2160_9.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:02] "GET /video_audio_8.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:02] "GET /video_2160_10.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:02] "GET /video_audio_9.m4s HTTP/1.1" 200 -
10.0.0.2 - - [20/Jan/2021 19:23:02] "GET /video_2160_11.m4s HTTP/1.1" 200 -
```

Figure 57: Output of the main server terminal

And as we can see in the next figure, the missing segments from the cache have now been downloaded in the cache folder.

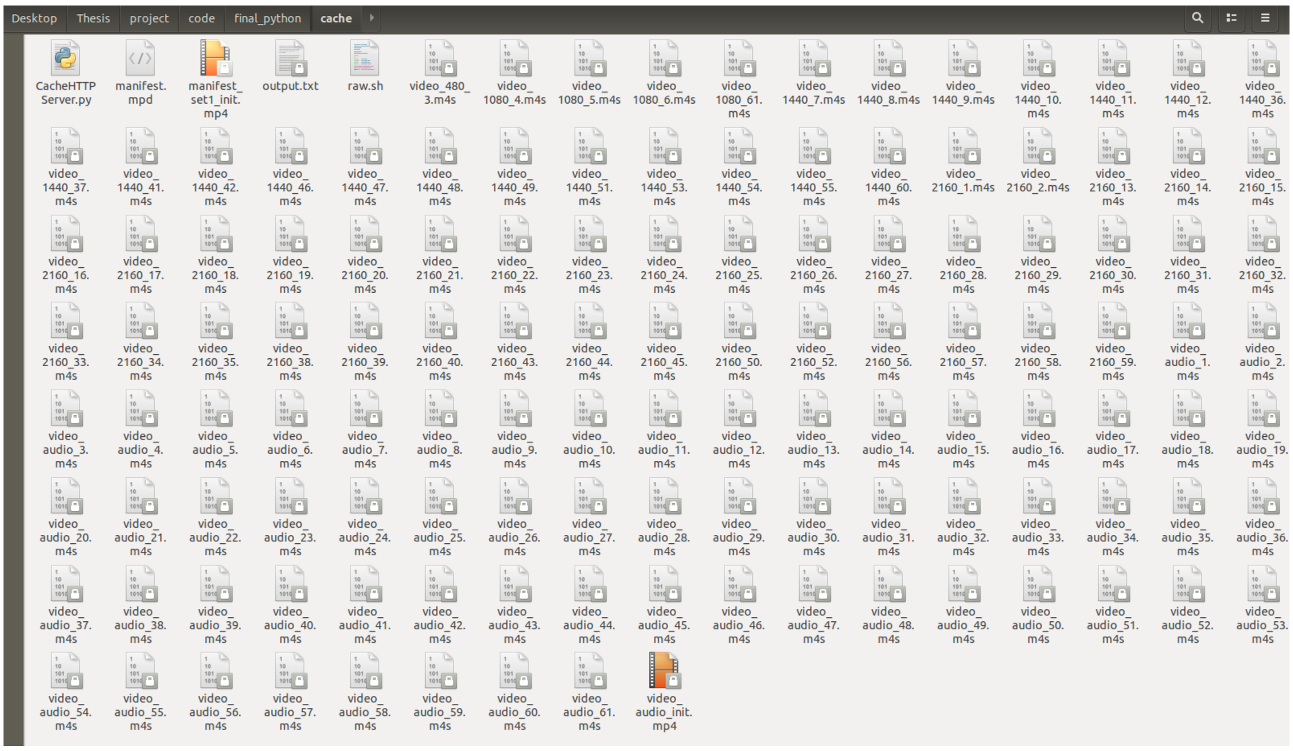
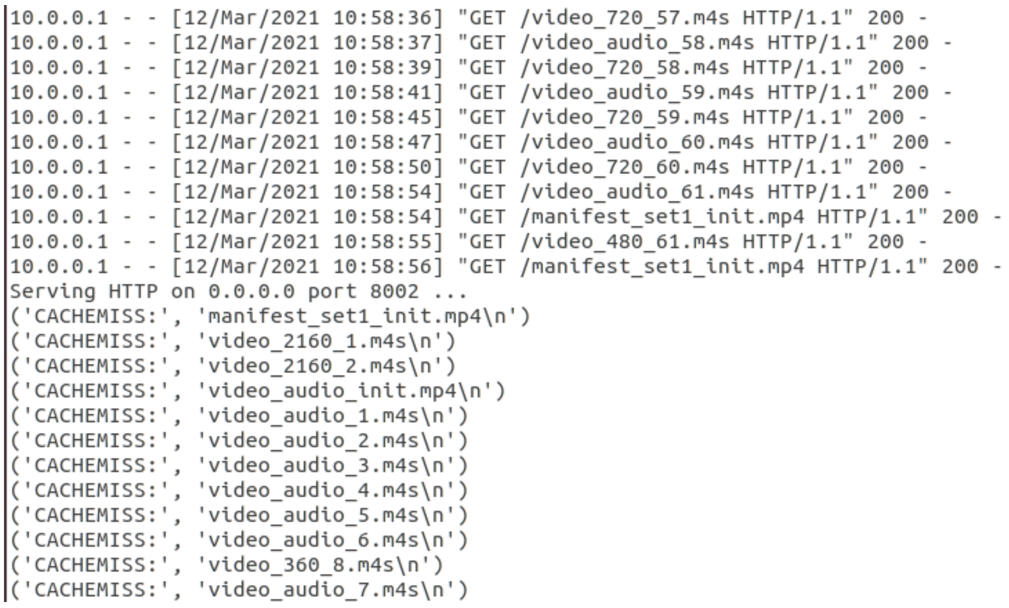


Figure 58: Contents of the cache folder after the implementation

After the completion of an experiment, we will find in cache folder the .txt with the information about the segments that played in this experiment. The format of this output follows:

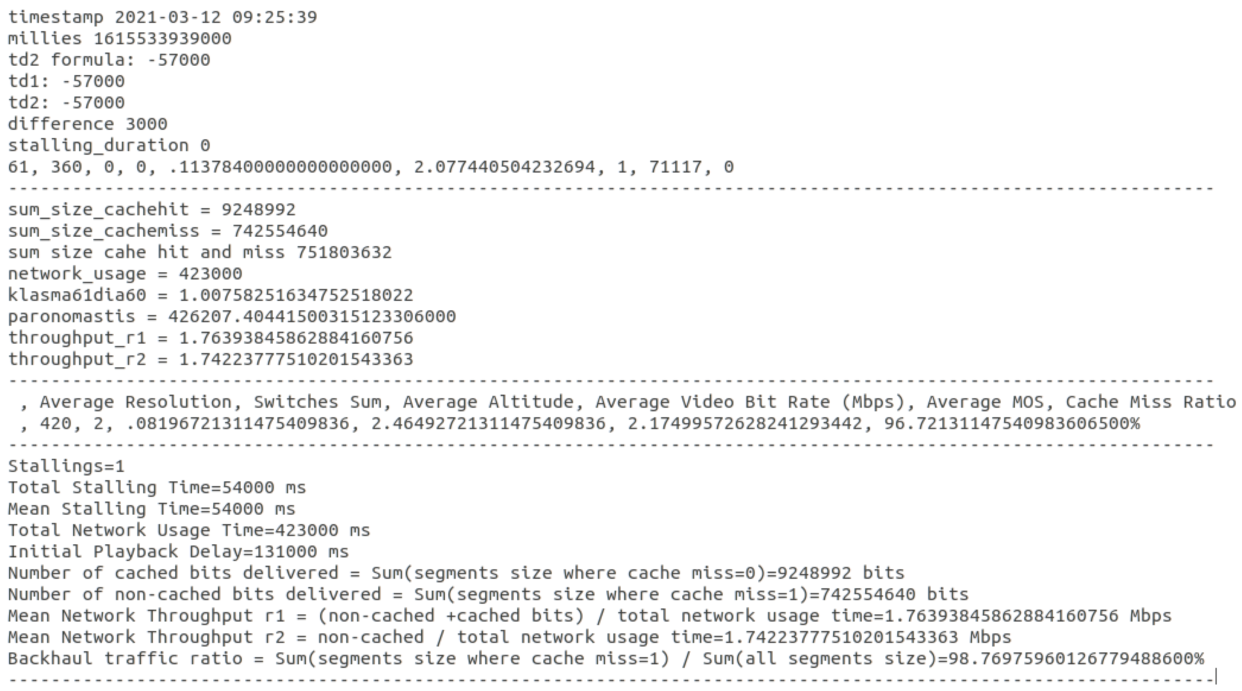


```
10.0.0.1 - - [12/Mar/2021 10:58:36] "GET /video_720_57.m4s HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:37] "GET /video_audio_58.m4s HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:39] "GET /video_720_58.m4s HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:41] "GET /video_audio_59.m4s HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:45] "GET /video_720_59.m4s HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:47] "GET /video_audio_60.m4s HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:50] "GET /video_720_60.m4s HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:54] "GET /video_audio_61.m4s HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:54] "GET /manifest_set1_init.mp4 HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:55] "GET /video_480_61.m4s HTTP/1.1" 200 -
10.0.0.1 - - [12/Mar/2021 10:58:56] "GET /manifest_set1_init.mp4 HTTP/1.1" 200 -
Serving HTTP on 0.0.0.0 port 8002 ...
('CACHEMISS:', 'manifest_set1_init.mp4\n')
('CACHEMISS:', 'video_2160_1.m4s\n')
('CACHEMISS:', 'video_2160_2.m4s\n')
('CACHEMISS:', 'video_audio_init.mp4\n')
('CACHEMISS:', 'video_audio_1.m4s\n')
('CACHEMISS:', 'video_audio_2.m4s\n')
('CACHEMISS:', 'video_audio_3.m4s\n')
('CACHEMISS:', 'video_audio_4.m4s\n')
('CACHEMISS:', 'video_audio_5.m4s\n')
('CACHEMISS:', 'video_audio_6.m4s\n')
('CACHEMISS:', 'video_360_8.m4s\n')
('CACHEMISS:', 'video_audio_7.m4s\n')
```

Figure 59: Output of an experiment

This output will be used from the raw.sh script, in order to export all the wanted experiment metrics in txt file, named metrics.txt

```
$ raw.sh &> metrics.txt
```



```
timestamp 2021-03-12 09:25:39
millies 1615533939000
td2 formula: -57000
td1: -57000
td2: -57000
difference 3000
stalling_duration 0
61, 360, 0, 0, .11378400000000000000, 2.077440504232694, 1, 71117, 0
---------------------------------------------------------------------------------------------------------
sum_size_cachehit = 9248992
sum_size_cachemiss = 742554640
sum size cahe hit and miss 751803632
network_usage = 423000
klasma61dia60 = 1.00758251634752518022
paronomastis = 426207.40441500315123306000
throughput_r1 = 1.76393845862884160756
throughput_r2 = 1.74223777510201543363
---------------------------------------------------------------------------------------------------------
 , Average Resolution, Switches Sum, Average Altitude, Average Video Bit Rate (Mbps), Average MOS, Cache Miss Ratio
 , 420, 2, .08196721311475409836, 2.46492721311475409836, 2.17499572628241293442, 96.72131147540983606500%
---------------------------------------------------------------------------------------------------------
Stallings=1
Total Stalling Time=54000 ms
Mean Stalling Time=54000 ms
Total Network Usage Time=423000 ms
Initial Playback Delay=131000 ms
Number of cached bits delivered = Sum(segments size where cache miss=0)=9248992 bits
Number of non-cached bits delivered = Sum(segments size where cache miss=1)=742554640 bits
Mean Network Throughput r1 = (non-cached +cached bits) / total network usage time=1.76393845862884160756 Mbps
Mean Network Throughput r2 = non-cached / total network usage time=1.74223777510201543363 Mbps
Backhaul traffic ratio = Sum(segments size where cache miss=1) / Sum(all segments size)=98.76975960126779488600%
---------------------------------------------------------------------------------------------------------
```

Figure 60: The metric.txt file

6.4 Limitations of the Mininet approach

After conducting many experiments with different combinations of [R1, R2, L], we have concluded to the fact that this approach had many drawbacks that led to a result that did not meet realistic scenarios. The restrictions that we met was the following:

- The channels were not utilized to a full extent despite the fact that we had enough resources available. For example, if we had a backhaul connection channel of 5 Mbps available, only 3 Mbps were used. This problem persists mostly due to the limitations of the Mininet interface but also due to the Get Before Send technique.
- The rate control was not accurate at all. In terms of explaining, we noticed that the Mininet interface had a default bandwidth value of 50 Gbps. What we observed was that while the rate was changing in our scripts, by applying one rule after the other, there was a gap between these rules. During this gap, the default value of the Mininet interface was applied, as it was the dominant one. This resulted to a higher data rate than the one anticipated by the experiment setup.
- The fact that the Mininet interface was restricting us from automating the procedure, made us conduct every experiment separately and that was exhausting and risky.

7. PROPOSED SET OF EXPERIMENTS

For all the reasons that we mentioned in the previous section, we have given up on the Mininet approach and we have continued our Thesis with a new perspective. Now that we do not depend on Mininet and its limitations, our new approach is implemented in regular gnome terminals with the IPs of the localhost 127.0.0.1. We will see how this approach is implemented in the next chapters.

7.1 Introduction

All things considered, the changes we have made for this approach are several. In the first place, along with Mininet, we have also eliminated the trace files that we used to add traffic to the fronthaul and the backhaul channel. Now the rate values change every second that passes with the help of the Rayleigh distribution formula within the cache code [71].

Furthermore, the caching policy remains the same, we fill the cache with random files until the buffer is full and the metrics that we collect are the same as well. We have automated the procedure with a single file that when is executed, all the experiments for all the different R1, R2, and L, which is the buffer size, run back to back without us needing to run each experiment separately. Also, for each set of [R1, R2, L] we conduct the experiment three times and we name it as a different sample, in order to see the differences and extract a mean value of the three experiments. These will become clearer below, where we give examples of the implementation and the code itself.

One thing to notice is that we have conducted two different sets of experiments. The first set, given the name “Get Before Send”, or GBS for short, is an approach where the data from the server is fetched serially, meaning that the second segment is sent to the client only after the first segment has been fetched. This approach is expecting to show that the traffic R1 that we applied in the fronthaul channel and the traffic R2 that we applied in the backhaul channel were under-utilized. This was one of the main problems that we noticed in the previous GBS implementation and it’s something that we expect to be more noticeable if we compare the GBS with the second set of experiments.

This under-utilization can be easily seen in the figures below, if we execute a simple GBS experiment, with R1=20 Mbps, R2=10 Mbps, and L=1000 MB and export some specific metrics as time passes. As one can easily observe, by comparing the 4 diagrams below, despite the fact that the buffer is big enough, the client is not able to exploit the bandwidth at all. At the same time, the number of stallings that cause the QoE to drop, is a matter of concern. As a result, a lot of bitrate oscillations, that lead to resolution changes, can be depicted. Last but not least, at the last diagram, it is clear that the backhaul channel is not fully utilized, as it is able to transmit data at a rate of 10 Mbps, and due to GBS technique it is almost at the 2/3 of that. The same experiment that we executed, will also be executed for the second set of experiments , in order to verify what we have claimed for the GBS restrictions.

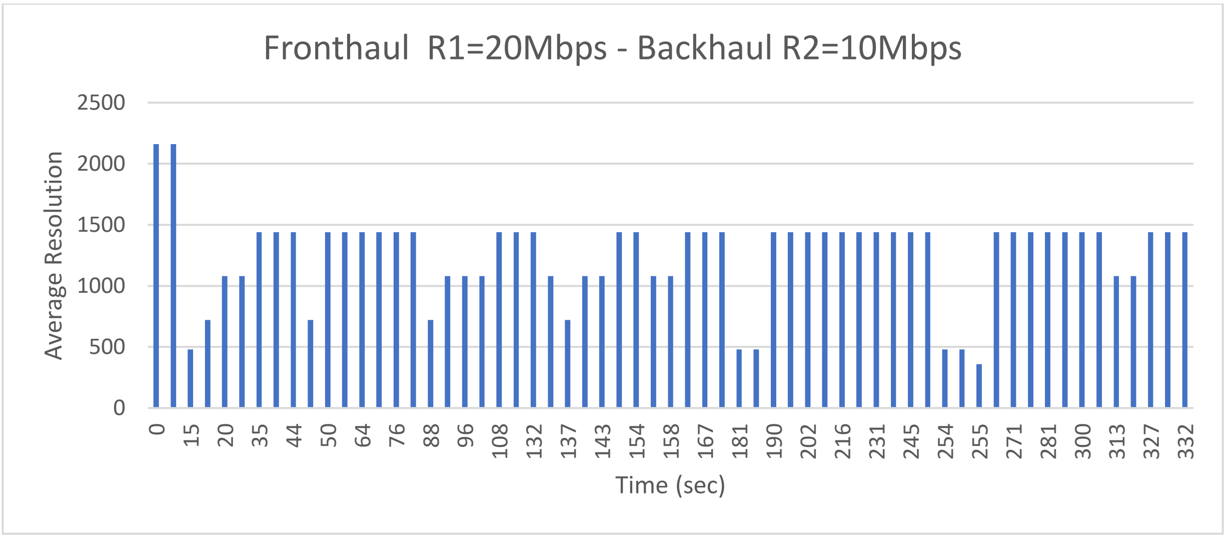


Figure 61: Average resolution vs time

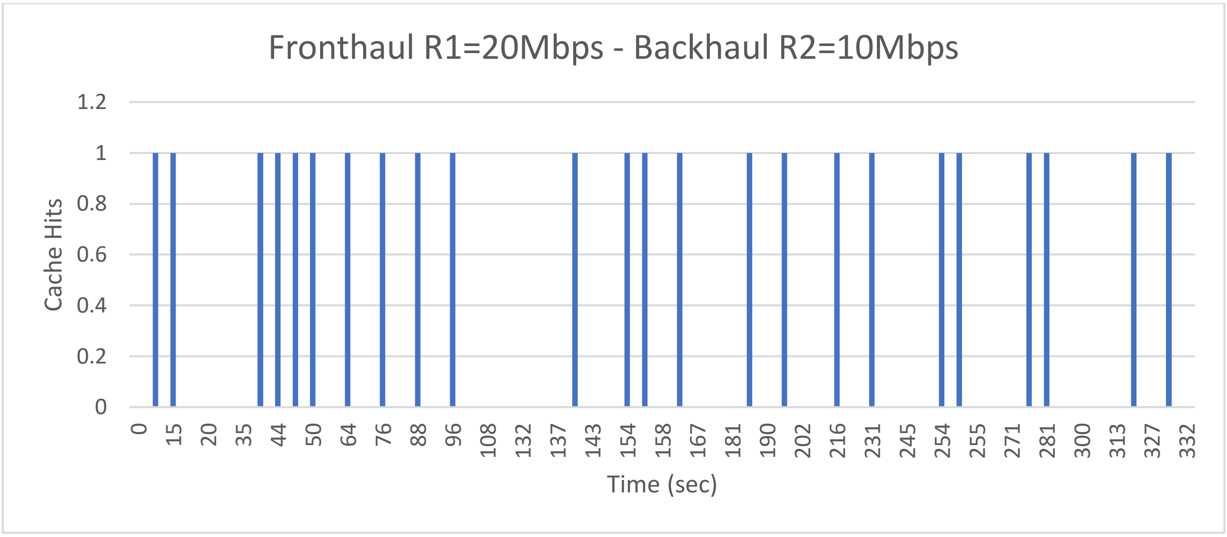


Figure 62: Cache hits vs time

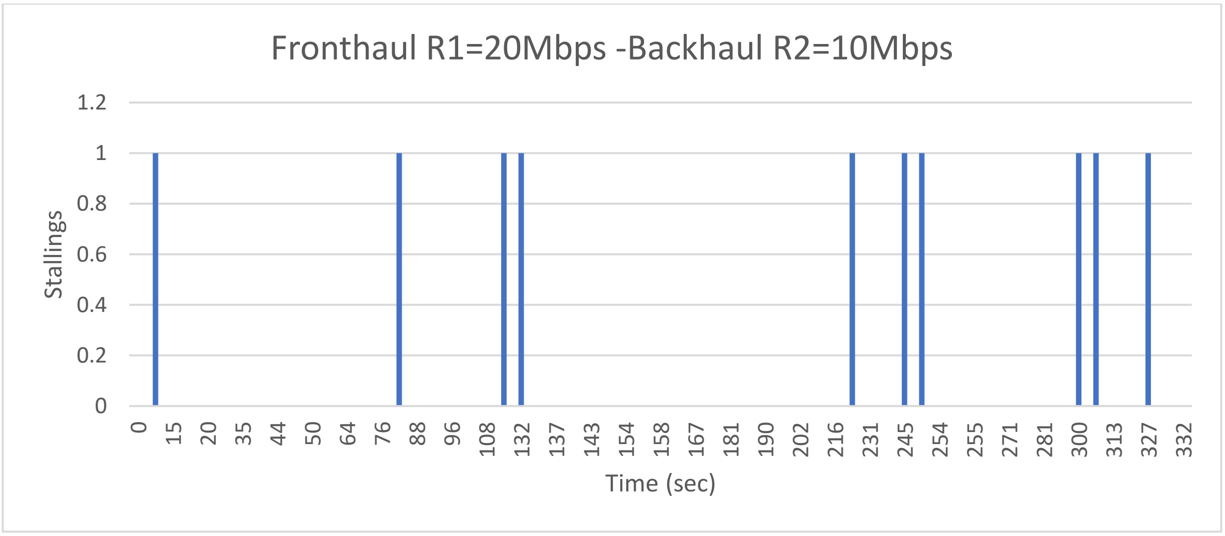


Figure 63: Number of stallings vs time

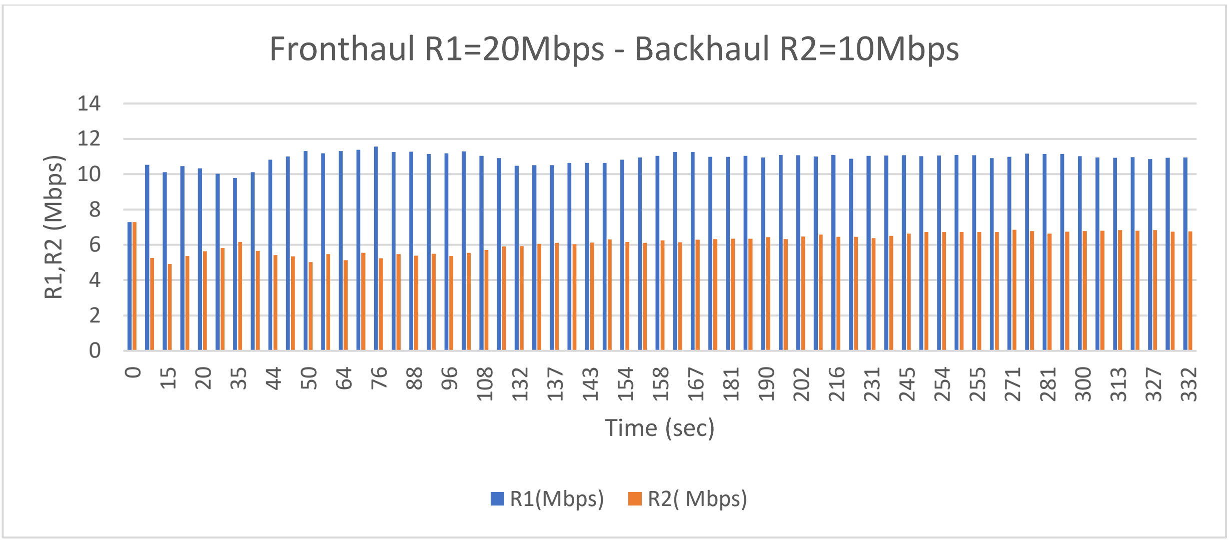


Figure 64: R1 and R2 vs time

The second set of experiments named "Send While Get", or SWG, as the name suggests, is more efficient than the first and the segments are fetched from the server simultaneously and are sent to the client without the need of any queue. For example, in the GBS approach, a segment was sent to the client chunk-by-chunk, and the second segment was not sent until the whole first segment was fetched. In the SWG approach, when a chunk of the first segment has been sent, then the request of the second segment is starting to be served, regardless of how many chunks of the first segment have been sent by that time. In this approach, given the fact that it is smarter than the first, is expected to show that the channels are utilized in a better way, leading us to better results in all the QoE metrics that we will export later after conducting a variety of experiments.

The new topology that we have come up with is the following.

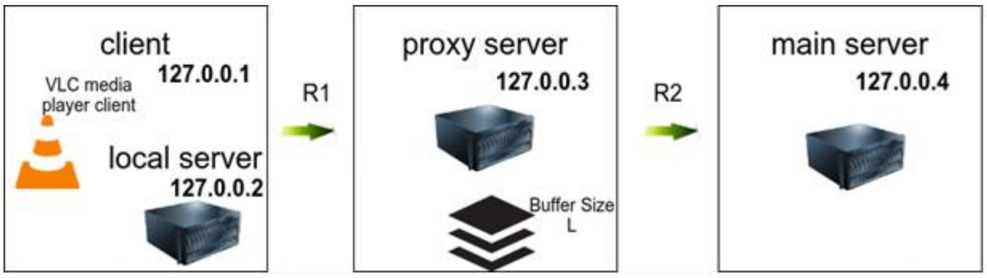


Figure 65: The new topology without Mininet

We have also added an extra server to the topology, called Local Server, where its objective is, in case of a smart caching policy aware of the contents in the cache, to download proactively more segments, directly from the Main Server, which are more likely to be selected by the media player. However, in our case, the existence of the Local Server is not of this use and acts only as a regular server that if the request can be served then it responds, and not in the opposite case.

So, the client part of our topology is considered both the VLC client and the Local Server and that's why in the figure above they exist in the same node. Moving on, the Proxy Server is the cache to our topology with the available buffer sizes 0, 100, 1000, 3000 MBs and the Main Server is the database that contains all the segments of all the different resolutions, 3000 MBs in total. We have created 3 folders:

- the local folder: starts out empty before each experiment and only contains the local.py code file.
- the cache folder: filled with random segments based on the buffer size and contains the 2 proxy code files (GBS and SWG approaches). These code files were given to us by the group of researchers that we worked together.
- the public folder: contains every file that the other servers may request along with the main.py code file.

We continue with the explanation of each set of experiments.

7.2 GBS Implementation

For the first set of experiments is the GBS implementation, as we have already explained, if a cache hit takes place, the proxy server forwards the video segment (locally cached) directly. If a cache failure takes place, the proxy downloads the full video segment before forwarding it to the client. It must become clear that all servers are HTTP servers written in python 3.6. We will explain the proxy-get-before-send.py code below.

The code starts with all the necessary imports and with the BaseHTTPRequestHandler class. This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). BaseHTTPRequestHandler provides a number of class and instance variables, and methods for use by subclasses. The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request [72].

The do_HEAD() method serves the 'HEAD' request type: it sends the headers it would send for the equivalent GET request. In the do_GET() method the request is mapped to a local file by interpreting the request as a path relative to the current working directory [72].

Then, the send_head() method is responsible for the GET and HEAD commands and it directs to the right method if the file exists in the cache (cache hit), or not (cache miss). Also, the name of the output file is constructed here, based on the arguments as: output_r1_r2_L_sample.txt and the moment of the GET request is written in it. If all the files have been written in the txt file, then a Keyboard Interrupt is raised, in order to stop the experiment.

The respond() and the request_and_respond() method are called in case of a cache hit and cache miss accordingly. The procedure is similar. The file that has been requested is given whole, byte by byte, and with the help of a counter to count each second that has passed, we change the rate that is applied to the fronthaul channel with the Rayleigh distribution. If the file is not in the cache, we must retrieve it from the main server, so we establish a connection with it and request the file from there, doing the exact procedure for the traffic in the backhaul channel now.

The rest of the code for the proxy is pretty straightforward. We need it to search the cache for types of files like .mp4, .mp4 and .mpd. There are options for default arguments, but we choose to pass them through the terminal for convenience in the understanding. The proxy server’s IP is 127.0.0.3 and its port is 8003.

The code for the local.py and the main.py is the same with the only differences being that we change the IPs and the ports when we execute them in the terminals and the default rate for the local server is a large number (1000000) because we consider the local and the client as one entity. The IP and port of the local server is 127.0.0.2 and 8002 accordingly and for the main server 127.0.0.4 and 8004. The code for the local.py and main.py is presented below:

The commands to run an example experiment [R1, R2, L, sample] = [10, 5, 100, 1] for each node are the following:

In the public folder:

```
$ python3 main.py -a 127.0.0.4 -p 8004 -s1 127.0.0.4 -p1 8004 -s2 127.0.0.4 -p2 8004
-r 5
```

- -a: is the address of the server
- -p: is the serving port
- -s1: is the remote server No. 1, in this case it is the same address as the main server has no other connections
- -p1: serving port of remote server No. 1, same as itself
- -s2: is the remote server No. 2, in this case it is the same address as the main server has no other connections
- -p2: serving port of remote server No. 2, same as itself
- -r: the data rate of the backhaul channel

In the cache folder:

```
$ python3 proxy-get-before-send.py -a 127.0.0.3 -p 8003 -sa 127.0.0.4 -sp 8004 -r1 10
-r2 5 -l 100 -sample 1
```

- -a: is the address of the proxy server
- -p: is the serving port
- -sa: is the remote server address of the main server
- -sp: is the remote server port of the main server
- -r1: the data rate r1 of the fronthaul channel
- -r2: the data rate r2 of the backhaul channel
- -l: is the buffer size
- -sample: is the number of the sample

In the local folder:

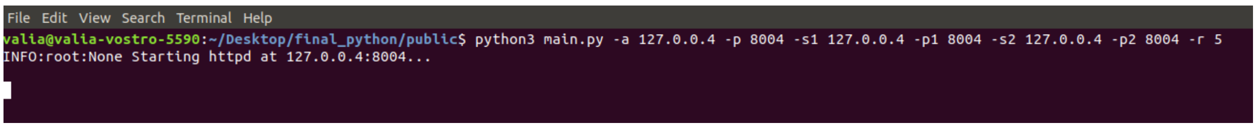
```
$ python3 local.py -a 127.0.0.2 -p 8002 -s1 127.0.0.3 -p1 8003 -s2 127.0.0.3 -p2 8003
```

The arguments of the local server are the same as the main server, but the remote server No.1 and 2 is the address of the cache server as there is no other connection for it.

In the directory where the above folders are, we execute the command for the VLC to start:

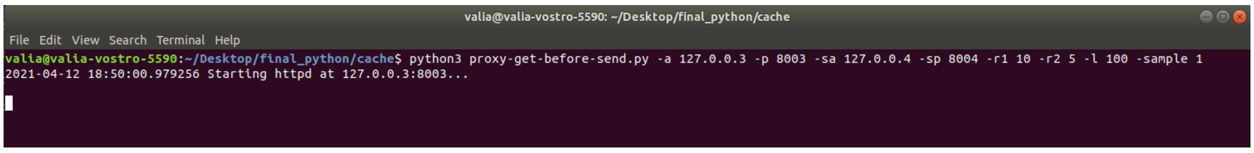
```
$ vlc-wrapper http://127.0.0.2:8002/manifest.mpd
```

A visualization of all the above commands is shown below in the figures:



```
valia@valia-vostro-5590:~/Desktop/final_python/public$ python3 main.py -a 127.0.0.4 -p 8004 -s1 127.0.0.4 -p1 8004 -s2 127.0.0.4 -p2 8004 -r 5
INFO:root:None Starting httpd at 127.0.0.4:8004...
```

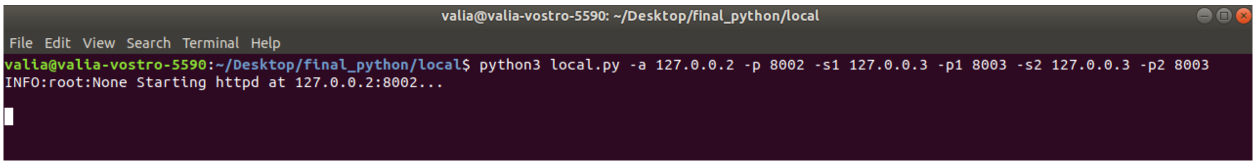
Figure 66: Main Server command



```
valia@valia-vostro-5590:~/Desktop/final_python/cache$ python3 proxy-get-before-send.py -a 127.0.0.3 -p 8003 -sa 127.0.0.4 -sp 8004 -r1 10 -r2 5 -l 100 -sample 1
2021-04-12 18:50:00.979256 Starting httpd at 127.0.0.3:8003...
```

valia@valia-vostro-5590: ~/Desktop/final_python/cache
File Edit View Search Terminal Help

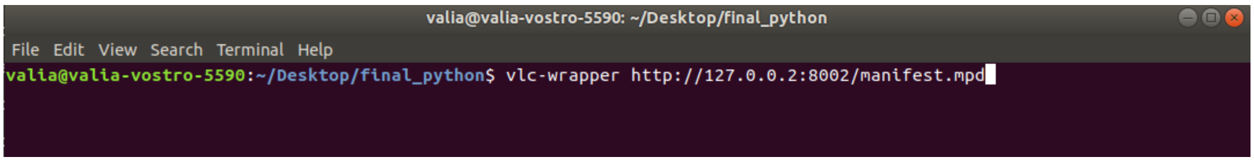
Figure 67: Proxy Server command



```
valia@valia-vostro-5590:~/Desktop/final_python/local$ python3 local.py -a 127.0.0.2 -p 8002 -s1 127.0.0.3 -p1 8003 -s2 127.0.0.3 -p2 8003
INFO:root:None Starting httpd at 127.0.0.2:8002...
```

valia@valia-vostro-5590: ~/Desktop/final_python/local
File Edit View Search Terminal Help

Figure 68: Local Server command

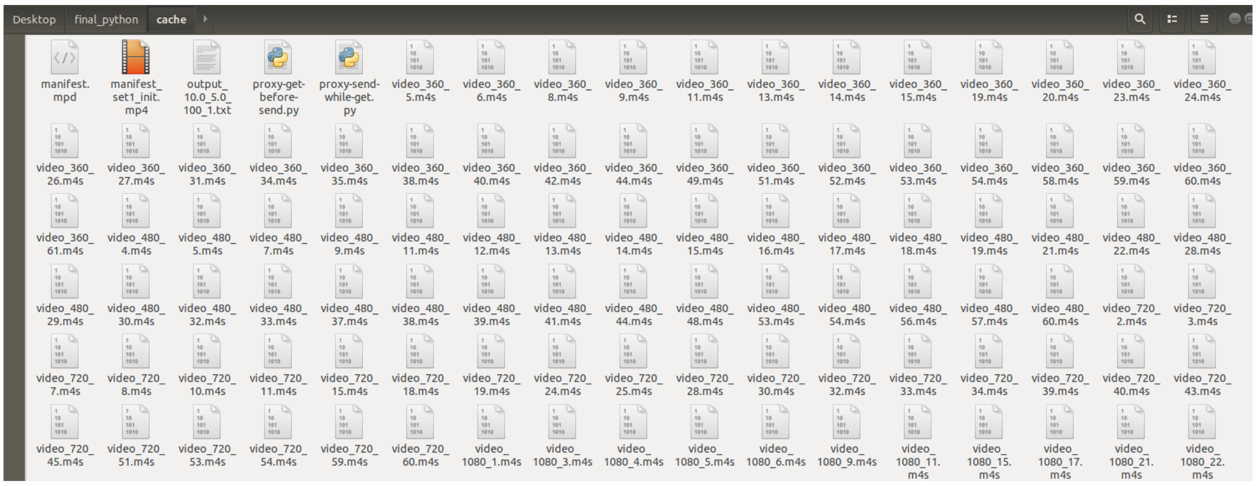


```
valia@valia-vostro-5590:~/Desktop/final_python$ vlc-wrapper http://127.0.0.2:8002/manifest.mpd
```

valia@valia-vostro-5590: ~/Desktop/final_python
File Edit View Search Terminal Help

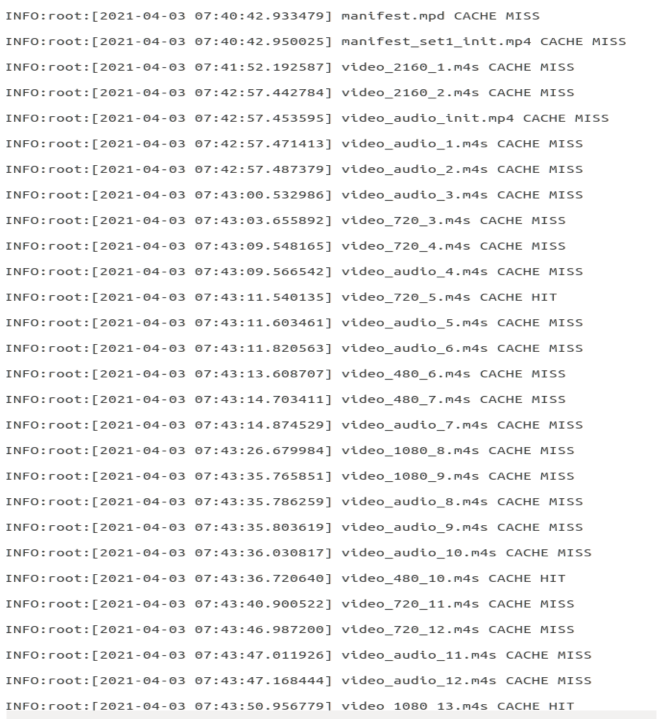
Figure 69: Client command

When this experiment is finished, in the cache folder, along with all the segments that already existed in the cache or have been requested, we can find the output txt. Its format can be shown below.



```
manifest.mpd  manifest_set1_init.mp4  output_10.0_5.0_100_1.txt  proxy-get-before-send.py  proxy-send-while-get.py  video_360_5.m4s  video_360_6.m4s  video_360_8.m4s  video_360_9.m4s  video_360_11.m4s  video_360_13.m4s  video_360_14.m4s  video_360_15.m4s  video_360_19.m4s  video_360_20.m4s  video_360_23.m4s  video_360_24.m4s
video_360_26.m4s  video_360_27.m4s  video_360_31.m4s  video_360_34.m4s  video_360_35.m4s  video_360_38.m4s  video_360_40.m4s  video_360_42.m4s  video_360_44.m4s  video_360_49.m4s  video_360_51.m4s  video_360_52.m4s  video_360_53.m4s  video_360_54.m4s  video_360_58.m4s  video_360_59.m4s  video_360_60.m4s
video_360_61.m4s  video_480_4.m4s  video_480_5.m4s  video_480_7.m4s  video_480_9.m4s  video_480_11.m4s  video_480_12.m4s  video_480_13.m4s  video_480_14.m4s  video_480_15.m4s  video_480_16.m4s  video_480_17.m4s  video_480_18.m4s  video_480_19.m4s  video_480_21.m4s  video_480_22.m4s  video_480_28.m4s
video_480_29.m4s  video_480_30.m4s  video_480_32.m4s  video_480_33.m4s  video_480_37.m4s  video_480_38.m4s  video_480_39.m4s  video_480_41.m4s  video_480_44.m4s  video_480_48.m4s  video_480_53.m4s  video_480_54.m4s  video_480_56.m4s  video_480_57.m4s  video_480_60.m4s  video_720_2.m4s  video_720_3.m4s
video_720_7.m4s  video_720_8.m4s  video_720_10.m4s  video_720_11.m4s  video_720_15.m4s  video_720_18.m4s  video_720_19.m4s  video_720_24.m4s  video_720_25.m4s  video_720_28.m4s  video_720_30.m4s  video_720_32.m4s  video_720_33.m4s  video_720_34.m4s  video_720_39.m4s  video_720_40.m4s  video_720_43.m4s
video_720_45.m4s  video_720_51.m4s  video_720_53.m4s  video_720_54.m4s  video_720_59.m4s  video_720_60.m4s  video_1080_1.m4s  video_1080_3.m4s  video_1080_4.m4s  video_1080_5.m4s  video_1080_6.m4s  video_1080_9.m4s  video_1080_11.m4s  video_1080_15.m4s  video_1080_17.m4s  video_1080_21.m4s  video_1080_22.m4s
```

Figure 70: Contents of cache folder after experiment and the output file



```
INFO:root:[2021-04-03 07:40:42.933479] manifest.mpd CACHE MISS
INFO:root:[2021-04-03 07:40:42.950025] manifest_set1_init.mp4 CACHE MISS
INFO:root:[2021-04-03 07:41:52.192587] video_2160_1.m4s CACHE MISS
INFO:root:[2021-04-03 07:42:57.442784] video_2160_2.m4s CACHE MISS
INFO:root:[2021-04-03 07:42:57.453595] video_audio_init.mp4 CACHE MISS
INFO:root:[2021-04-03 07:42:57.471413] video_audio_1.m4s CACHE MISS
INFO:root:[2021-04-03 07:42:57.487379] video_audio_2.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:00.532986] video_audio_3.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:03.655892] video_720_3.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:09.548165] video_720_4.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:09.566542] video_audio_4.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:11.540135] video_720_5.m4s CACHE HIT
INFO:root:[2021-04-03 07:43:11.603461] video_audio_5.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:11.820563] video_audio_6.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:13.608707] video_480_6.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:14.703411] video_480_7.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:14.874529] video_audio_7.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:26.679984] video_1080_8.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:35.765851] video_1080_9.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:35.786259] video_audio_8.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:35.803619] video_audio_9.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:36.030817] video_audio_10.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:36.720640] video_480_10.m4s CACHE HIT
INFO:root:[2021-04-03 07:43:40.900522] video_720_11.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:46.987200] video_720_12.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:47.011926] video_audio_11.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:47.168444] video_audio_12.m4s CACHE MISS
INFO:root:[2021-04-03 07:43:50.956779] video_1080_13.m4s CACHE HIT
```

Figure 71: Output file format

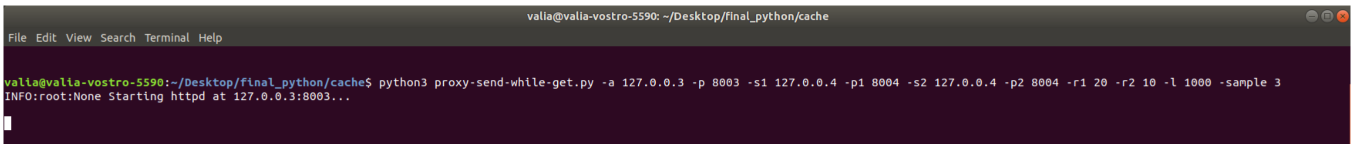
7.3 SWG Implementation

In this second set of experiments, we use the Send While Get technique, which means that the Proxy Server does not wait for a single segment to be delivered to the Local Server, but while a segment is sent, the next one is already beginning the process of being requested and delivered. If a cache hit takes place, the proxy server forwards the video segment (locally cached) directly. If a cache failure takes place, we have configured the proxy server to read the proxy-to-main socket packet per packet (8192) and forward directly the packets upon reception.

The commands for an example experiment [r1, r2, L, sample] = [20, 10, 1000, 3] are shown below for the new proxy, the rest remain the same:

```
$ python3 proxy-send-while-get.py -a 127.0.0.3 -p 8003 -s1 127.0.0.4 -p1 8004 -s2 127.0.0.4 -p2 8004 -r1 20 -r2 10 -l 1000 -sample 3
```

The arguments follow the same logic as with the GBS technique.



valia@valia-vostro-5590: ~/Desktop/final_python/cache
File Edit View Search Terminal Help

```
valia@valia-vostro-5590:~/Desktop/final_python/cache$ python3 proxy-send-while-get.py -a 127.0.0.3 -p 8003 -s1 127.0.0.4 -p1 8004 -s2 127.0.0.4 -p2 8004 -r1 20 -r2 10 -l 1000 -sample 3
INFO:root:None Starting httpd at 127.0.0.3:8003...
```

Figure 72: Execution of the SWG proxy

After the demonstration of how to run this technique, we will conduct a simple experiment with the data of R1=20 Mbps, R2=10 Mbps, L=1000 MB like the one that we did before for the GBS approach. The manipulation of the raw data gave us the below figures.

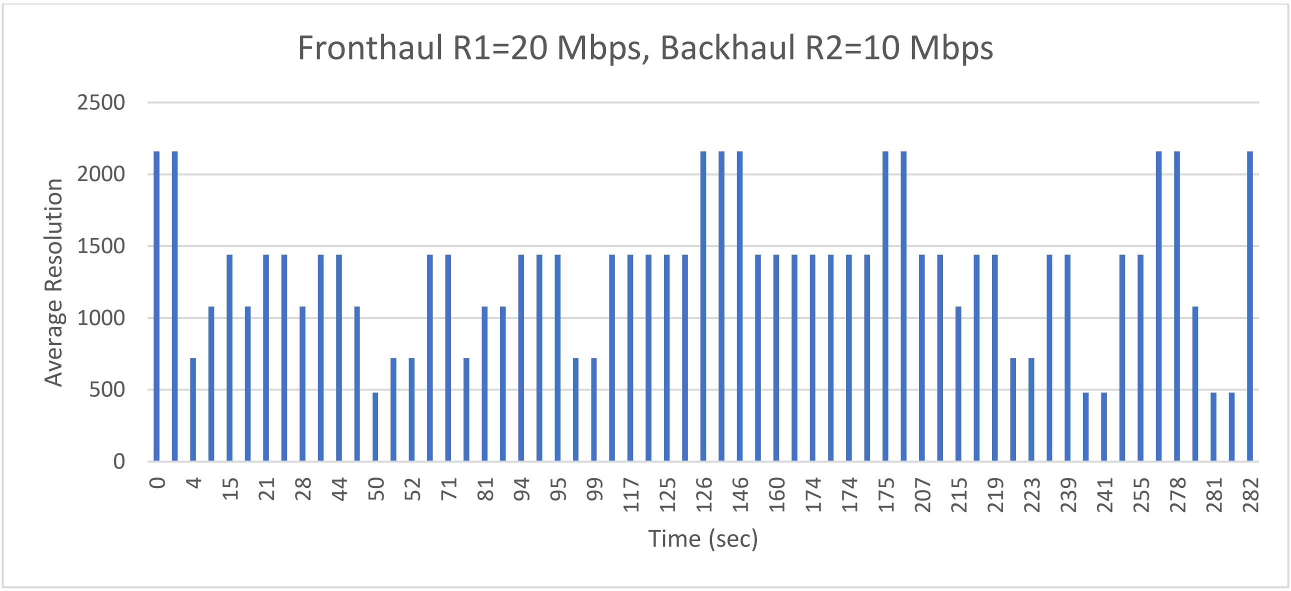


Figure 73: Average resolution vs time

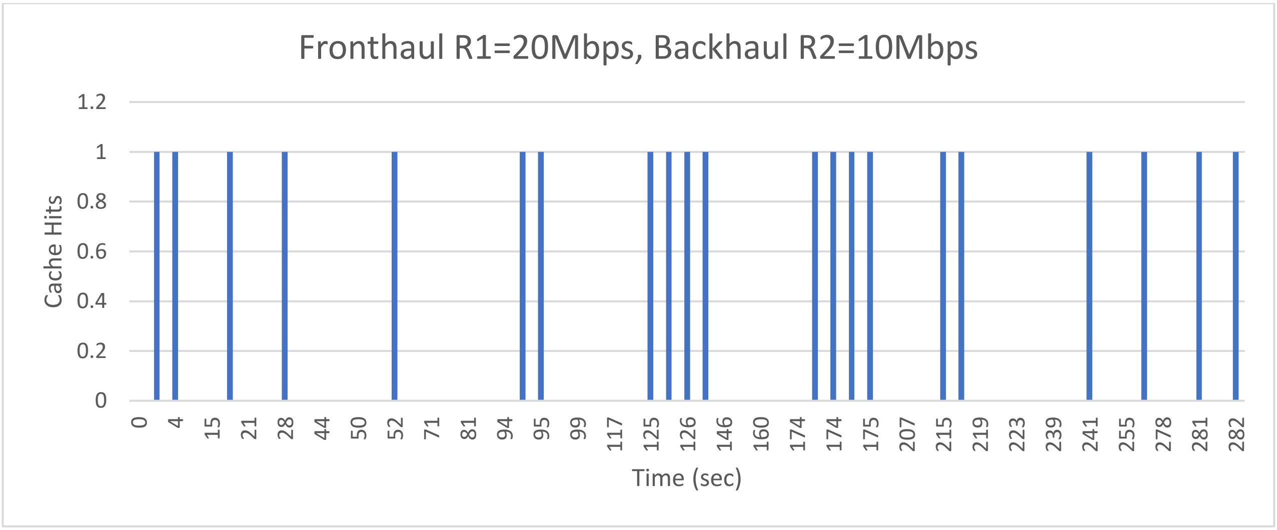


Figure 74: Cache hit vs time

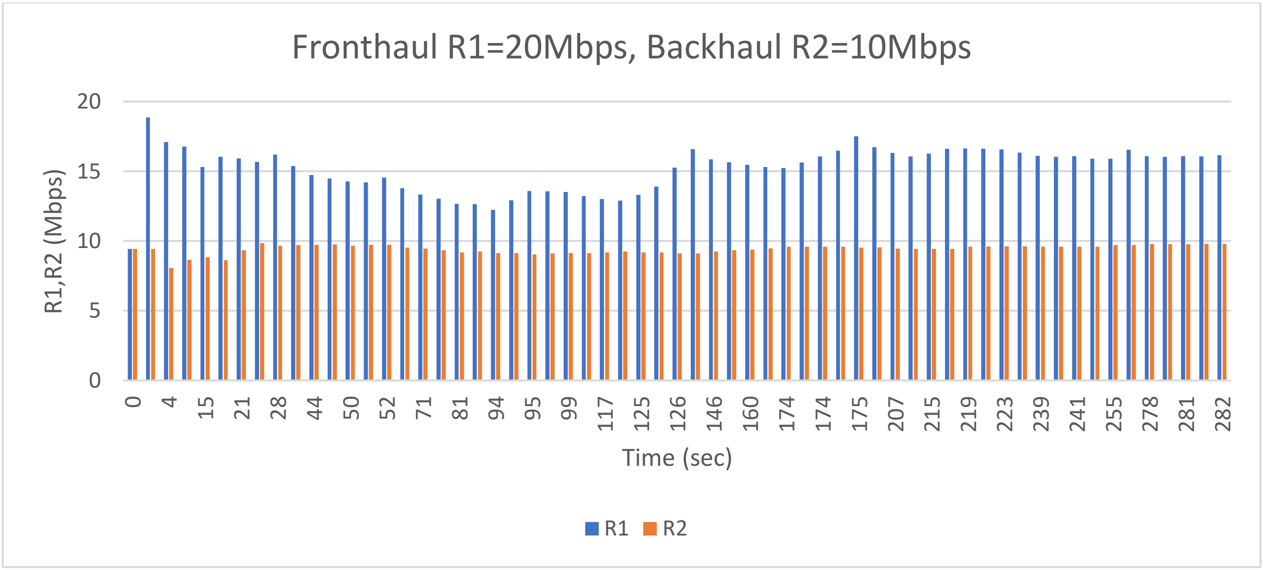


Figure 75: Rate R1, R2 vs time

By studying the above diagrams , what we can say with certainty is that there are no stallings in this approach, which means that the QoE is increased for the user. Also, the bitrate oscillations are fewer, resulting in the client exploiting better the cache hits. However, the most significant observation is that both the fronthaul and the backhaul channel, are utilized to a better extent , due to the fact that no channel stays inactive for a long period of time.

7.4 Automation

In order to automate the procedure and run many experiments back to back, we have implemented an automation.py script that does that for us.

At first, before each experiment, the contents of the folder of the Local and the Proxy Server must be cleared from all the .mpd, .mp4 and .m4s files:

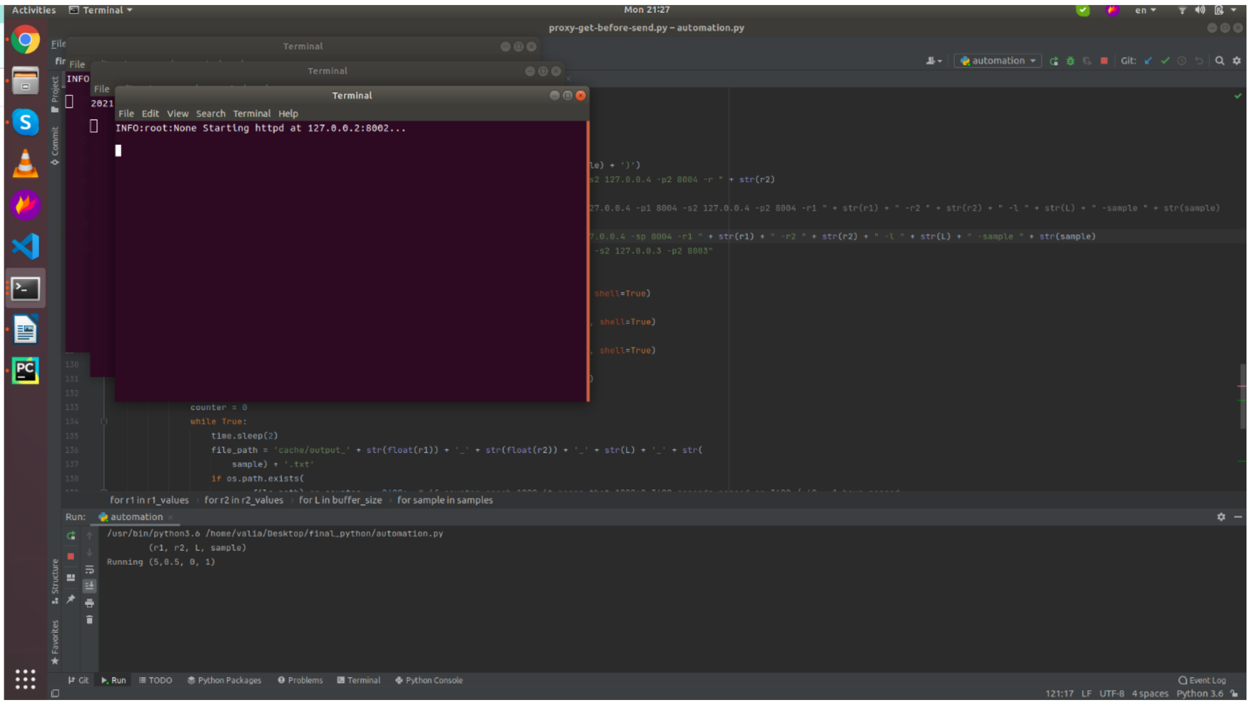
Next, depending on the type of cache (where it has a buffer capacity of 0, 100 or 1000MB), it fills it as close to that number as possible. It tries to find randomly files from the public folder and checks if they fit in the cache. If the segment fits, it copies it, if it doesn't then it tries up to 20 times to find another to fit. This happens because there are segments that are of 30MB and others of some kilobytes.

If the buffer size of the cache is 3000MB that means that all content from the public folder can be copied in the cache.

Finally, when an experiment is finished, we explained above that the output file is written in the cache folder and the proxy terminal is accepting a keyboard interrupt, so it closes. When the output file has been created, there is no need to continue the experiment even if the video has not finished because we have all the information we need. So, we kill all the other terminals in order to open the new ones for the next experiment. This can be achieved by finding the process ID of the terminal commands of the local, main and client, along with the window of the VLC still playing, and killing them all.

After finishing with all the methods, the main program consists of the matrices of the data for the experiments (R1, R2, L, sample) and 4 consecutive for loops. Then, the commands for each terminal are stored in some variables and we use the subprocess.Popen function to open the four different terminals. The subprocess module allows us to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module takes several input arguments. We use the cwd to change the working directory, as the python files exist in different folders. Also, we use the option shell=True to cause the subprocess to spawn an intermediate shell process and tell it to run the command. We add the sleep function between opening the terminals to avoid over-congesting the hard drive of our PC. When the terminals have opened and the commands have been executed, there is no way to find out when the experiment has finished, except when the output file has been created in the cache folder. So, we start an endless loop for searching for that output and when we find it, we kill all the open processes in order for the new experiment to start. The code can be seen in the Appendix B.

The IDE we use for the automation.py is the PyCharm Community Edition 2021.1. We simply press the play button and let the editor run the experiments back to back. The combination of the set of data in the code ran for 3 days. We present an output of the interface.



```
INFO:root:None Starting httpd at 127.0.0.2:8002...
```

```
        counter = 0
        while True:
            time.sleep(2)
            file_path = 'cache/output_' + str(float(r1)) + '_' + str(float(r2)) + '_' + str(L) + '_' + str(
                sample) + '.txt'
            if os.path.exists(
```

```
/usr/bin/python3.6 /home/valia/Desktop/final_python/automation.py
        (r1, r2, L, sample)
Running (5,0.5, 0, 1)
```

Figure 76: Example of the automation process

7.5 New Raw File and Metrics

Now that the format of the output file has changed, we need a new file that calculates the metrics for the new experiments. The metrics that we need are the same that we explained in chapter 6.3.5. After the completion of all the experiments we have many output files. We transfer those files in a folder within the public folder called “outputs”. We also create a folder named “metrics” in the public folder. This is because the raw.py file needs to be with all the segments because it scans each segment written in the output file and finds its size. The raw file is run by executing:

```
$ python3 raw.py
```

The result of this run is the creation of a metrics txt file, corresponding to the output file, created in the metrics folder along with a csv where all the average values are passed in cells for each experiment for our convenience. Each set of experiments is executed three times, and the average value of the three samples is calculated, in order to eliminate the bias. These average values are used in the diagrams below.

The code of the raw.py is found in the Appendix B.

7.6 Diagrams analysis

In chapter 7.6.1, we will analyze the QoE metrics and their evolvement over the fronthaul traffic R1, for 3 different buffer sizes. They will be compared with each other, in contemplation of the optimal combination of caching and QoE, so that the available network resources will be fully utilized.

In chapter 7.6.2, we will examine some metrics and their evolvement over the increase of the buffer size for four pairs of fronthaul R1 and backhaul R2 values.

7.6.1 QoE metrics VS Fronthaul channel rate R1

In the 3 next diagrams we can see how the average resolution improves with the increase of the buffer size L, for different values of the fronthaul channel with rate R1.

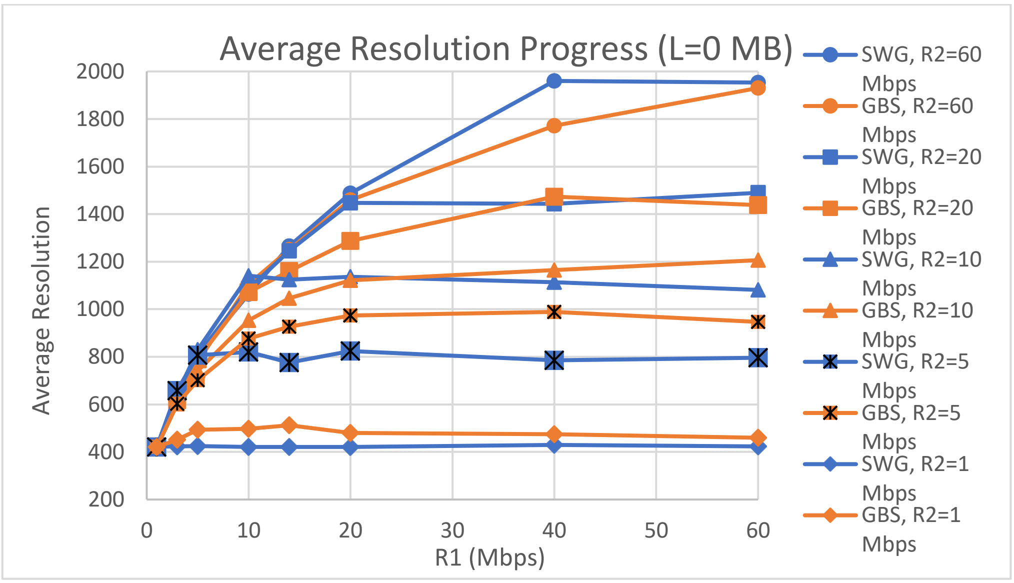


Figure 77: Average Resolution vs R1 (L=0)

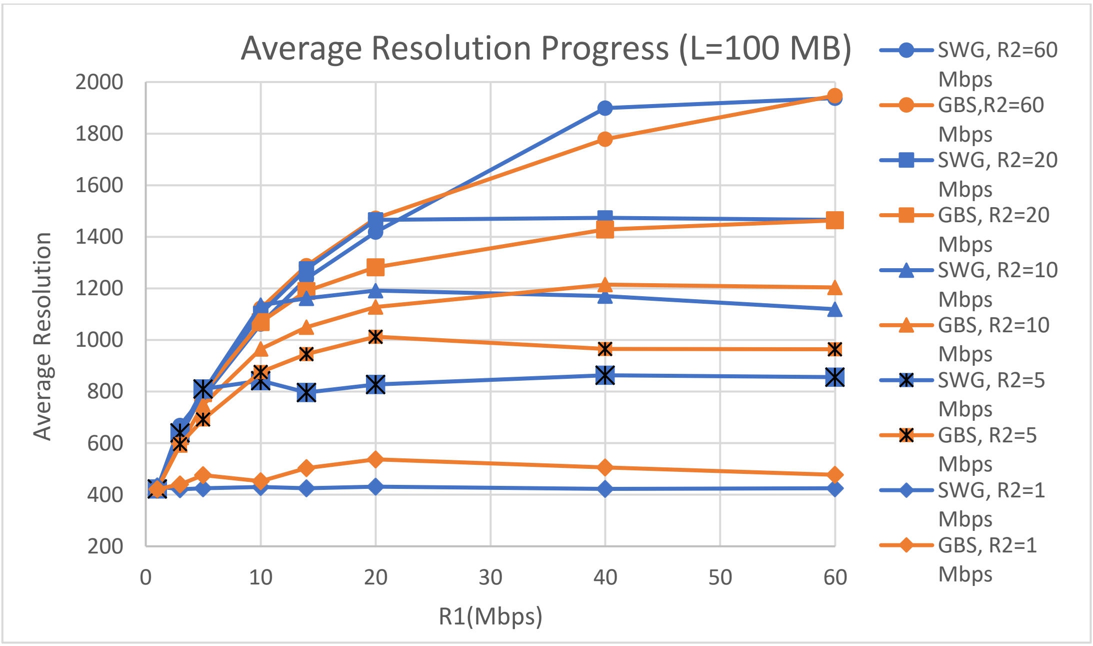


Figure 78: Average Resolution vs R1 (L=100 MB)

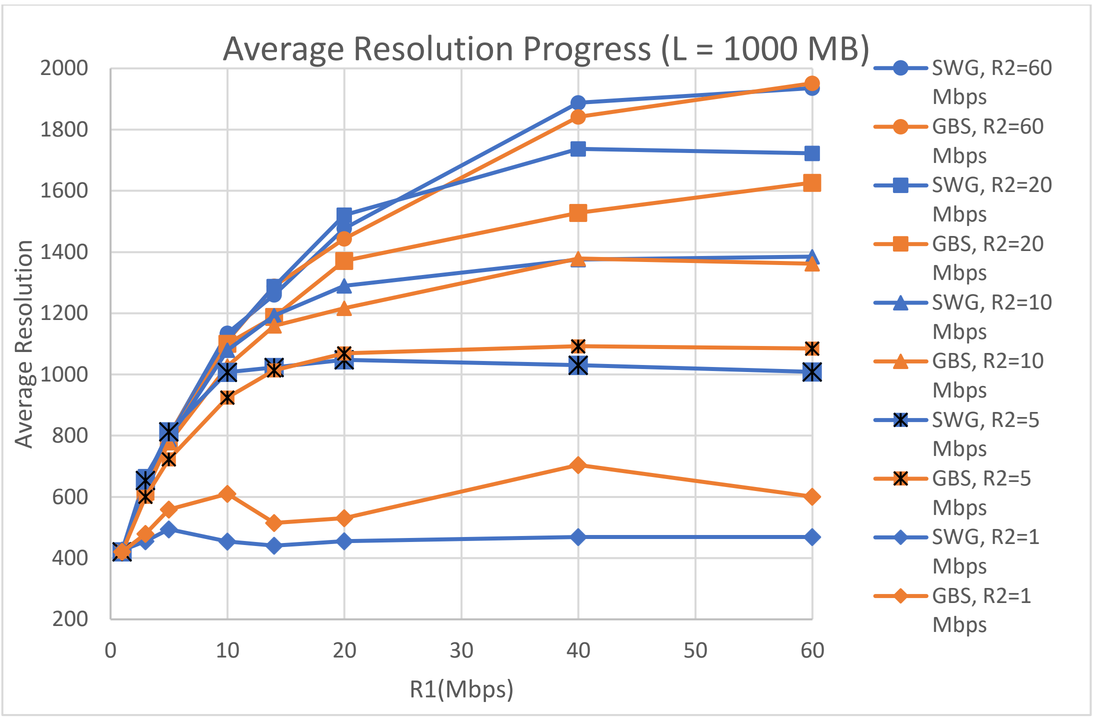


Figure 79: Average Resolution vs R1 (L=1000 MB)

After plotting the graphs, we can come to the conclusion that:

- As it is reasonable, the higher the backhaul R2 (for the same R1), the higher the average resolution playing for the client.
- We notice an increase of the average resolution at every curve of the figure, while R2<R1. When R2>=R1 is valid, the value of the quality of the video stabilizes to some extent and can be further improved only if we have big enough buffer available.
- For R2 = 1, 5 or 10 Mbps, a false image is given that the GBS approach gives better results. The fact in this case is that the HAS algorithm is over-optimistic, making the media player increase the quality. However, this increase is not justified in terms of bandwidth, and this is the reason of the continuous occurrences of bitrate oscillations and as a result, of stallings. On the other hand, the SWG approach has less stallings than the GBS one, so it is the one that gives better results in terms of QoE.
- The right approach that shows the dominance of the SWG technique is mostly shown at the curves of the figures for R2 = 20 and 60 Mbps, regardless of buffer size.
- While the buffer size L is increased, we can notice an increase of the average resolution for the same R1 and R2 values. The positive impact of caching is clearly noticeable. To make the example more understandable, in case of an R1 = 20 Mbps and R2 = 10 Mbps, if the buffer size L = 0 MB, then the average resolution ranges between 1100 and 1200, but if the L = 1000MB, then the average resolution is almost increased at about 100 units, reaching a value close to 1300.

On the next 3 figures, we can see the average altitude for 2 distinct values of R1, in the 3 different buffer sizes that we have available in our experiment.

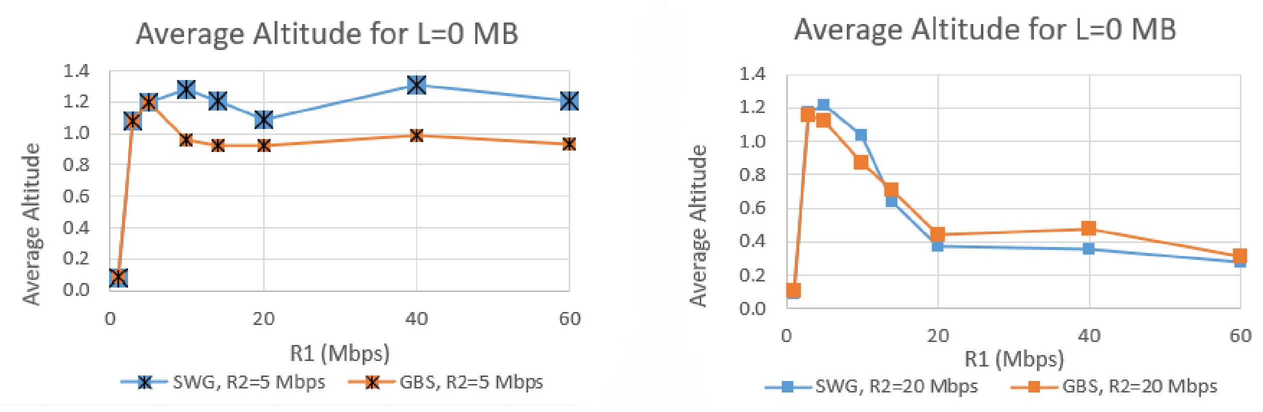


Figure 80: Average Altitude VS R1 for R2=5,20 Mbps (L=0 MB)

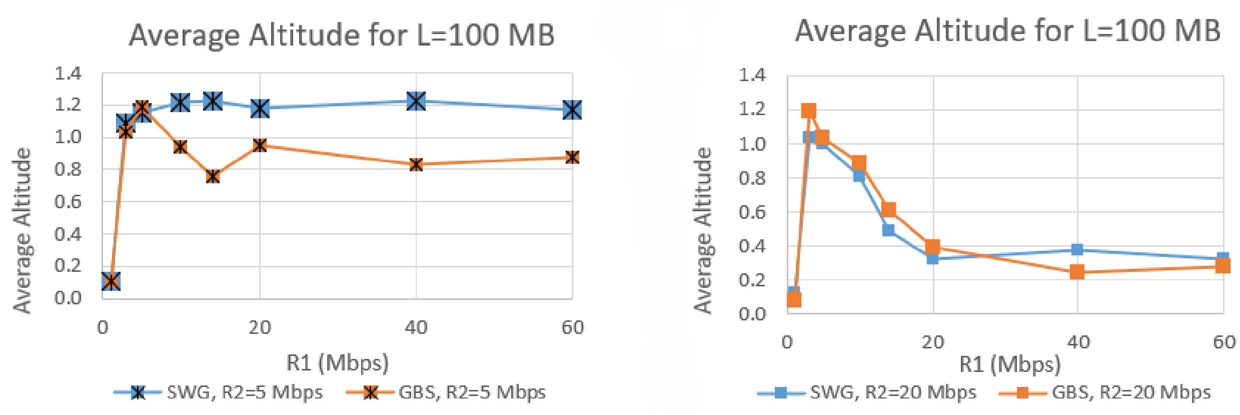


Figure 81: Average Altitude VS R1 for R2=5,20 Mbps (L=100 MB)

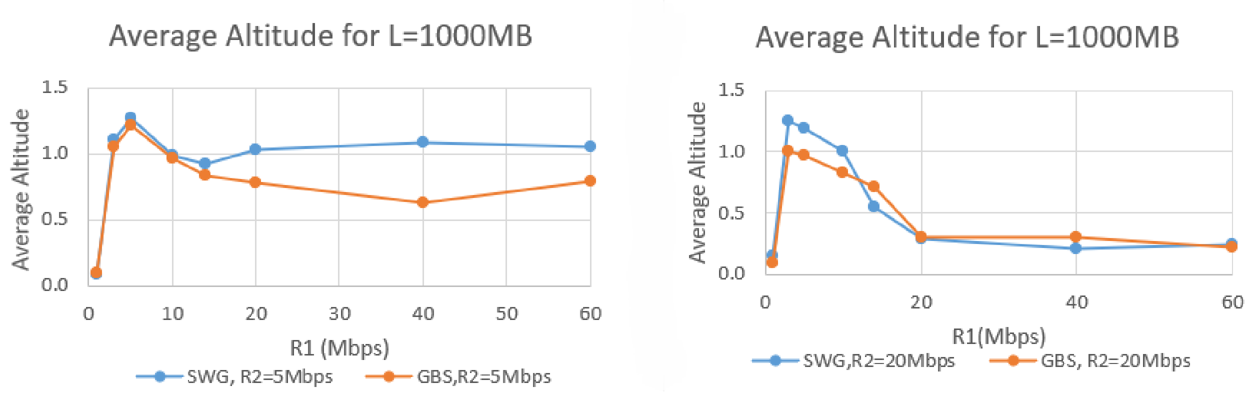


Figure 82: Average Altitude VS R1 for R2=5,20 Mbps (L=1000 MB)

- The peak of the curve in the figure for R1 = 5 Mbps, is a good example for someone to notice the over-optimism of the HAS algorithm, which causes many bitrate oscillations, leading to the increase of the altitude value.
- If the channel is of a higher rate, this impact is eliminated, and the average altitude reaches to very low values.
- In order to fully extinguish the impact and cause it to reach lower values, a higher R2 is needed, as shown by comparing the figures above.
- The higher the R2, the SWG approach remains the best option.

- The increase of the buffer size does not demonstrate any big difference in the behavior of the curve in the figure. A quite general conclusion is that the higher the buffer size L, the lower the average altitude.

Concerning the mean network throughput R1, we are able to observe the followings in terms of buffer sizes 0, 100 and 1000MB:

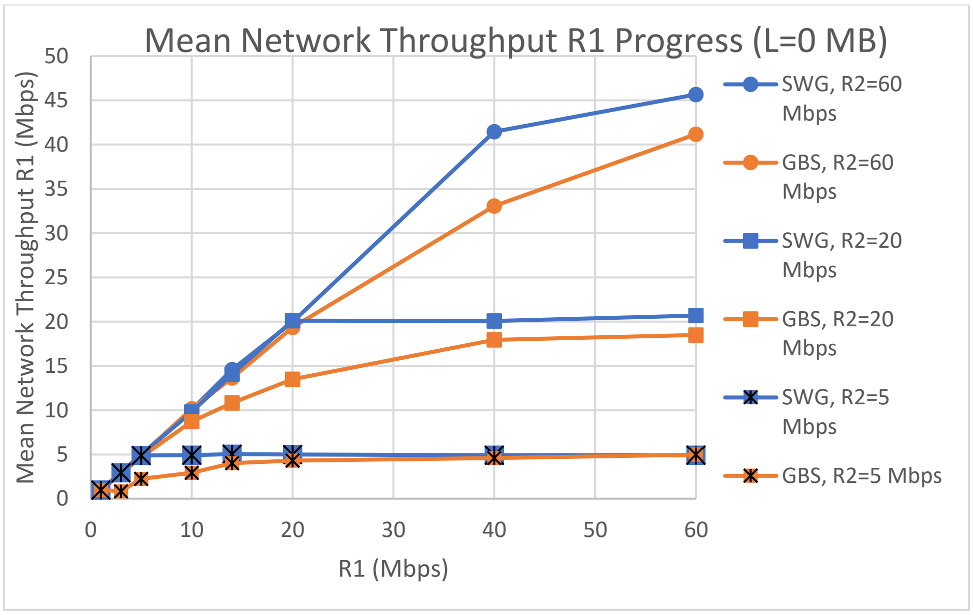


Figure 83: Mean Network Throughput R1 vs R1 for L= 0 MB

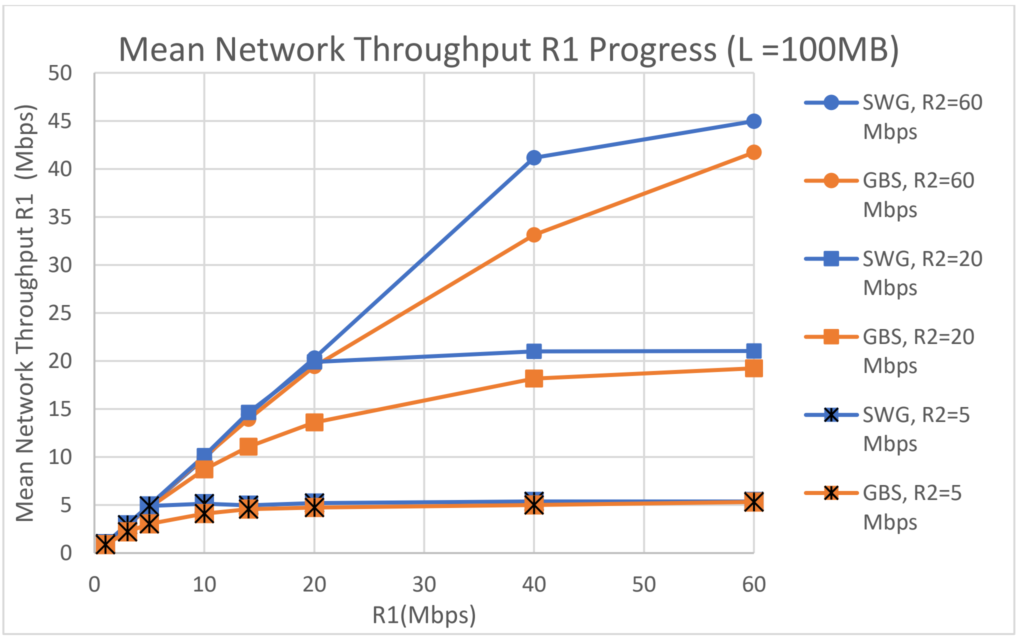


Figure 84: Mean Network Throughput R1 vs R1 for L= 100MB

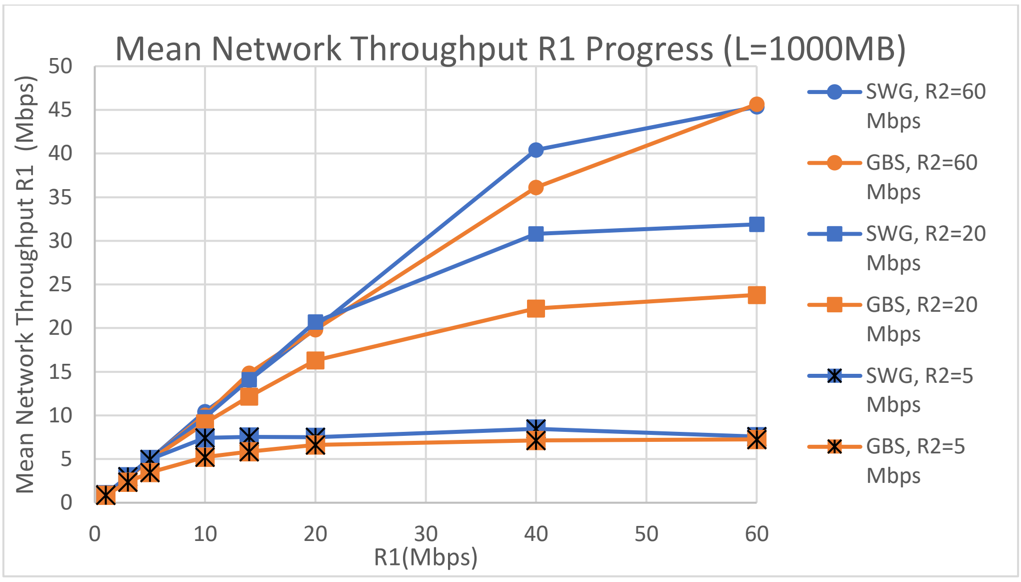


Figure 85: Mean Network Throughput R1 vs R1 for L= 1000MB

- The results are quite expected, as the mean throughput is increased linearly until the point where R1 = R2, where it reaches the highest value (for L=0).
- For R2 = 60 Mbps, the throughput is not 60 as expected, but it is lower because the highest available quality (2160p) has an average bitrate of almost 45Mbps.
- The higher the buffer size, meaning that the fronthaul channel R1 is used more often than the backhaul, the higher the mean throughput R1.
- There is better utilization of the channels in the SWG approach, especially when then buffer size is higher.
- In the GBS approach, the network resources remain unutilized, something prohibited for any vendor.
- The gap shown in the diagram between the 2 curves of SWG and GBS (for the same R2 value), moves to higher values of the R1 throughput. In other words, the R2 increase is responsible for the change of place of the gap in the right direction upwards.

In the next 3 figures, we validate the behavior of the backhaul R2 for the buffer sizes of 0 MB, 100 MB and 1000 MB .

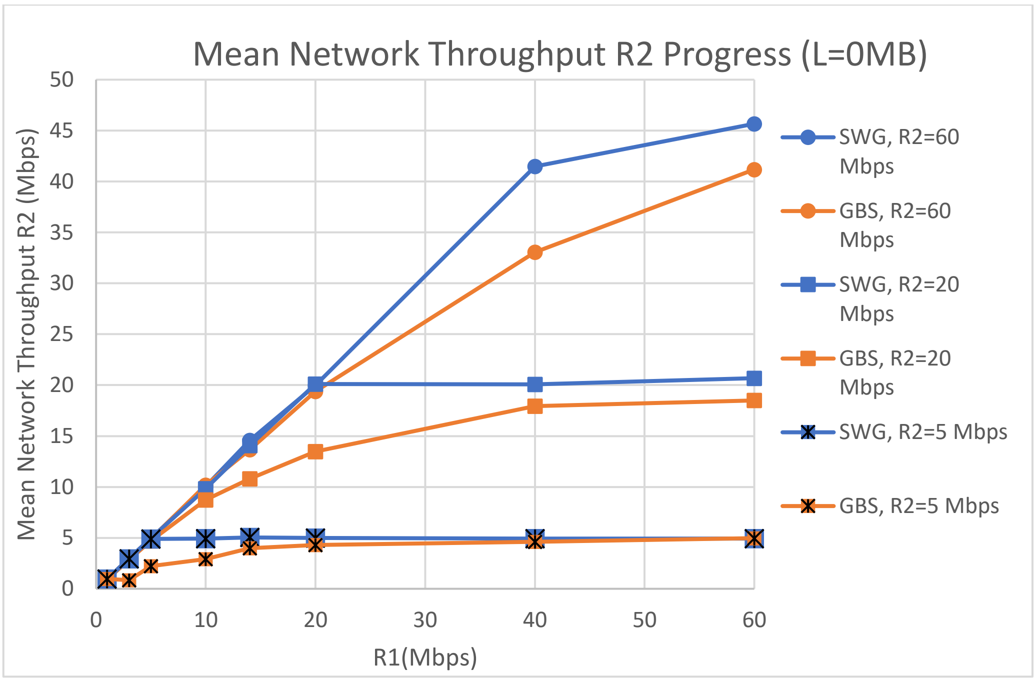


Figure 86 : Mean Network Throughput R2 vs R1 for L= 0 MB

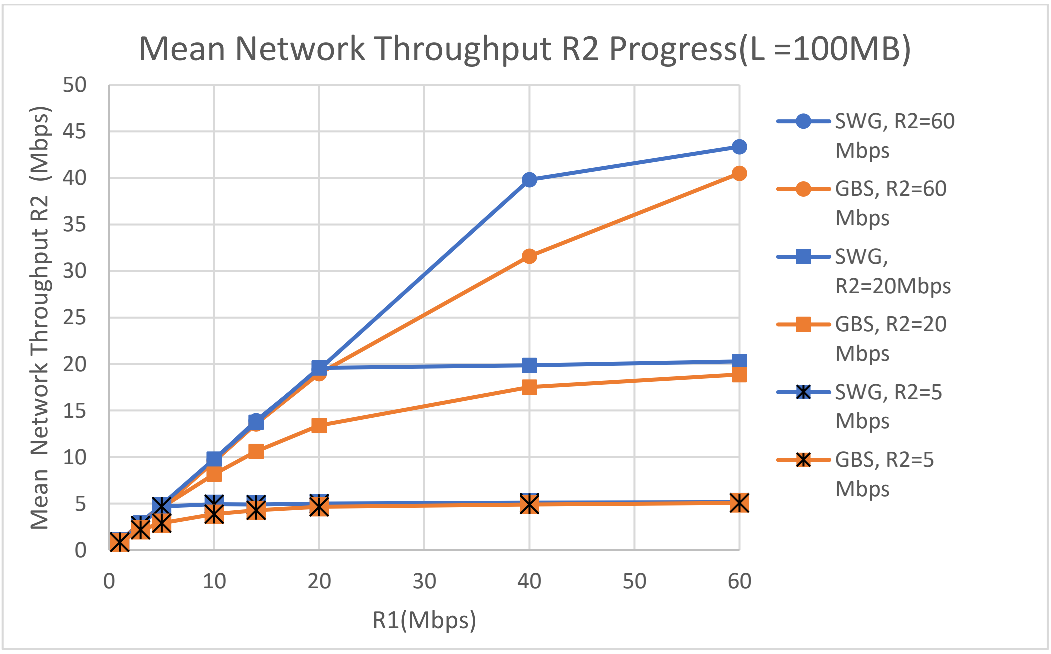


Figure 87: Mean Network Throughput R2 vs R1 for L=100 MB

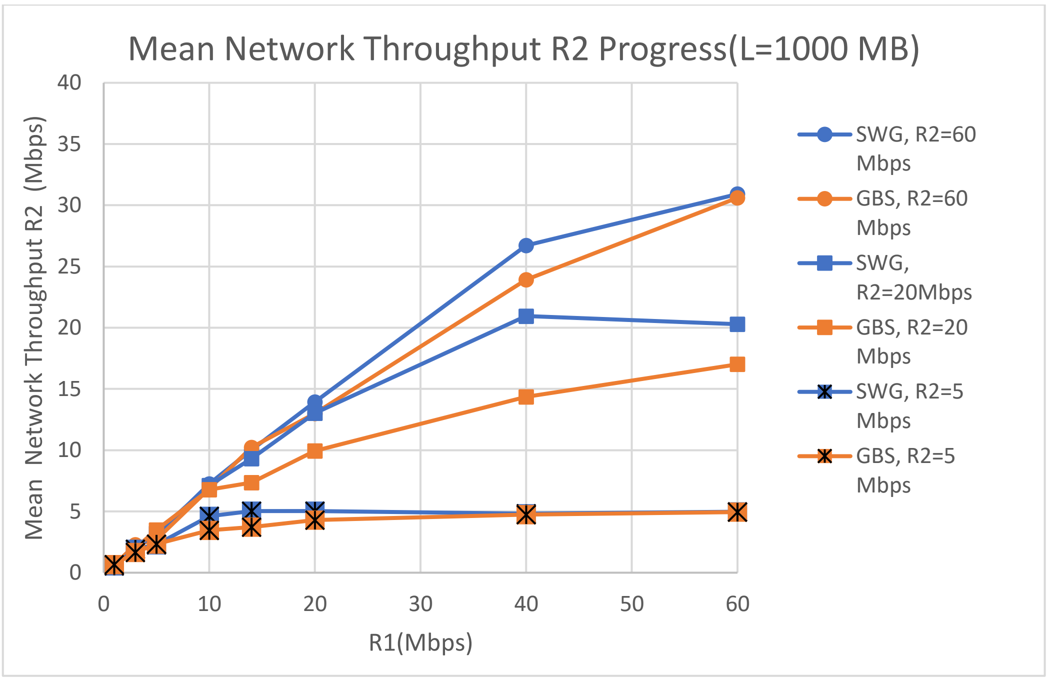


Figure 88: Mean Network Throughput R2 vs R1 for L=1000 MB

We can easily notice that:

- The curve is linear until it reaches an upper limit.
- Again, better utilization of the channel is shown in the SWG approach, compared to the GBS approach that leaves resources even more unutilized.
- As the buffer size increases, the mean network R2 throughput is reduced for the same R1 rate. This happens because the fronthaul link is used more often due to the fact that we have more cache hits.
- However, the use of a non-smart algorithm and the non-existence of a multithread approach makes the backhaul channel idle for a long time, in case of a cache hit.
- Ideally, in case of a cache hit where the fronthaul channel would serve the client, the backhaul channel should download new segments to the cache, in order to benefit later, in terms of QoE and network resources.
- The SWG approach is generally more beneficial, but the biggest benefit comes in the points from R1 = 0 to 10 Mbps, where there is a range of available resolutions. The higher the R1 than those values, there is no more benefit, due to the absence of another higher resolution.

The Mean Opinion Score, that a client may extract if he watches a video, is depicted below for 3 different kind of cache servers:

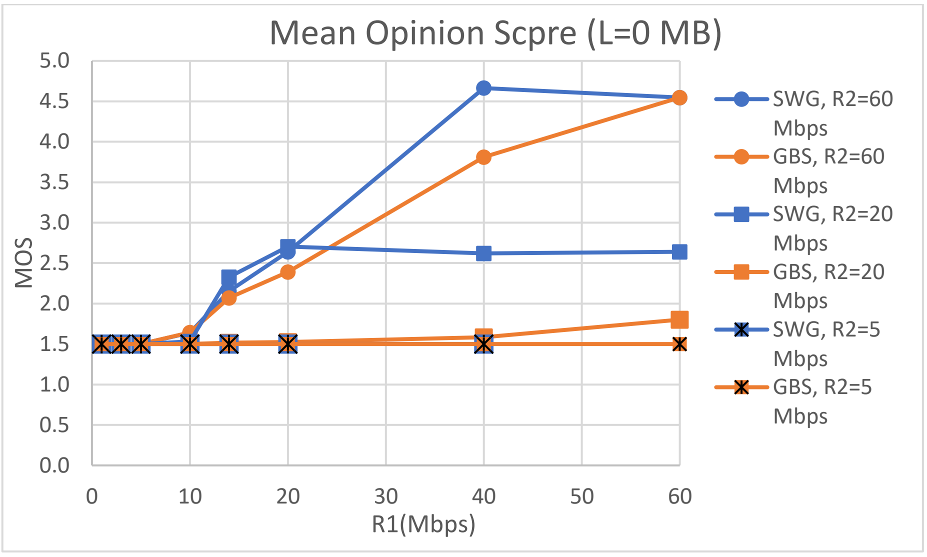


Figure 89: MOS vs R1 for L=0 MB

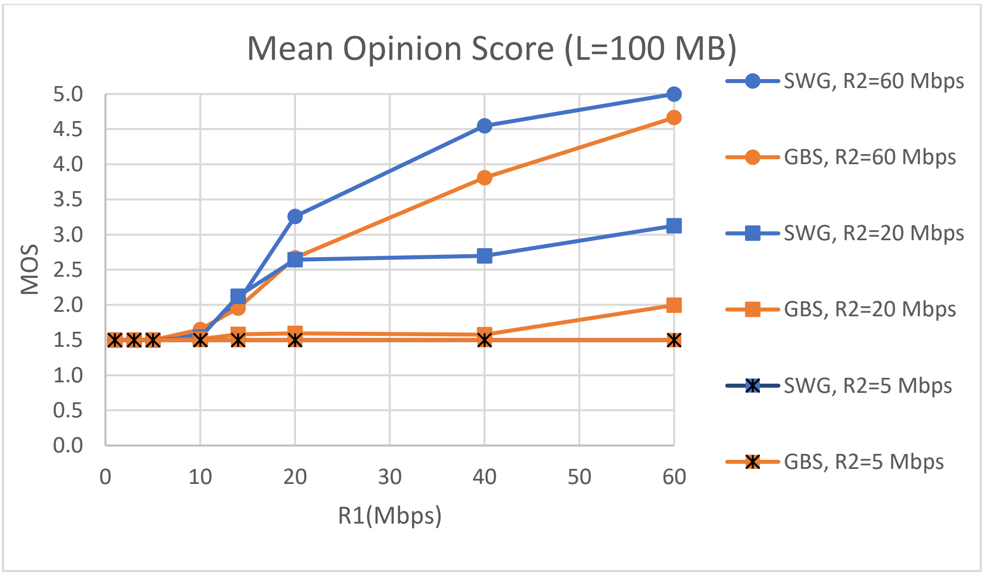


Figure 90: MOS vs R1 for L=100 MB

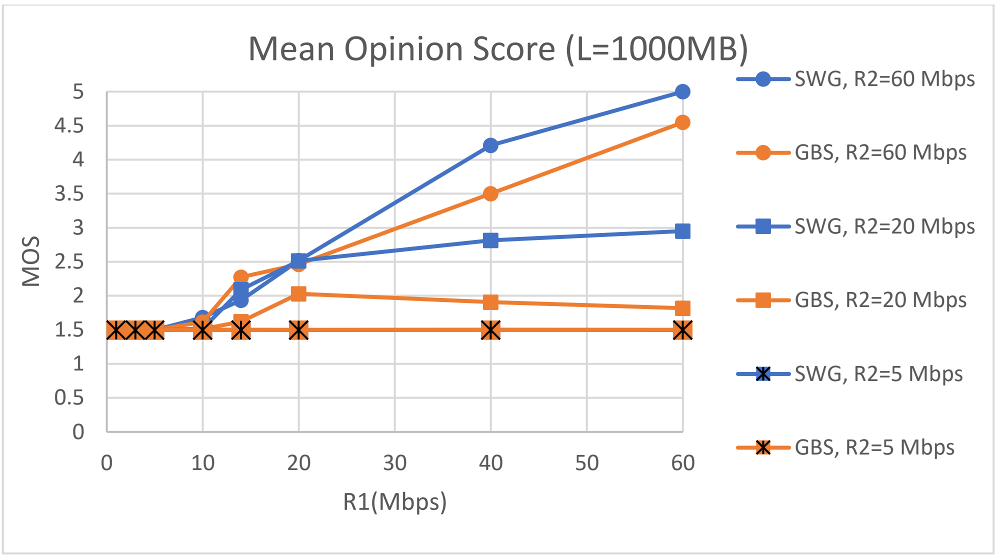


Figure 91: MOS vs R1 for L=1000 MB

We can conclude that :

- The MOS metric demonstrates the client's preference towards the SWG approach.
- The HAS algorithm, due to the fact that acts as over-optimistic for the available bandwidth, leads to the video freezing many times (stalling events), and as a result the client is not satisfied.
- The bottom line is that the caching algorithms and the HAS algorithms are always interdependent because if the HAS algorithm is not cache-aware then the caching does not help very much. In other words, we are wasting resources for storage, without it being beneficial.
- The higher the R2 value, the less the bottleneck, so there is a benefit in the MOS value.
- For R2 = 5 Mbps, the stallings that occur are so many that they lead to the worst value of the MOS. As a result, both approaches do not differ that much in terms of what the client feels while watching the video.

Lastly, concerning the number of stallings of our experiments, we conclude to the following results :

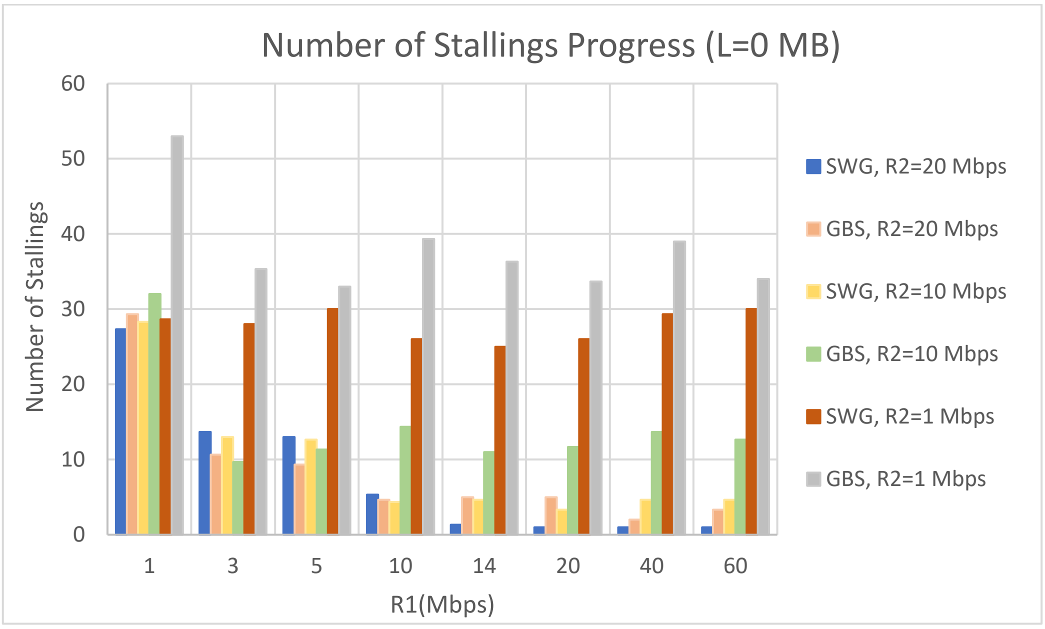


Figure 92: Number of stallings vs R1 for L=0 MB

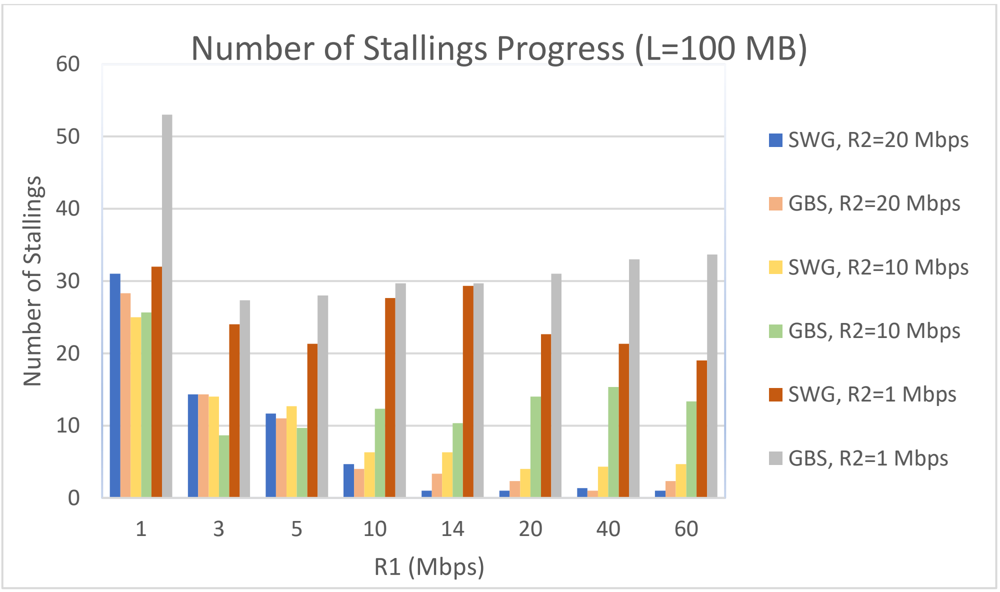


Figure 93: Number of stallings vs R1 for L=100 MB

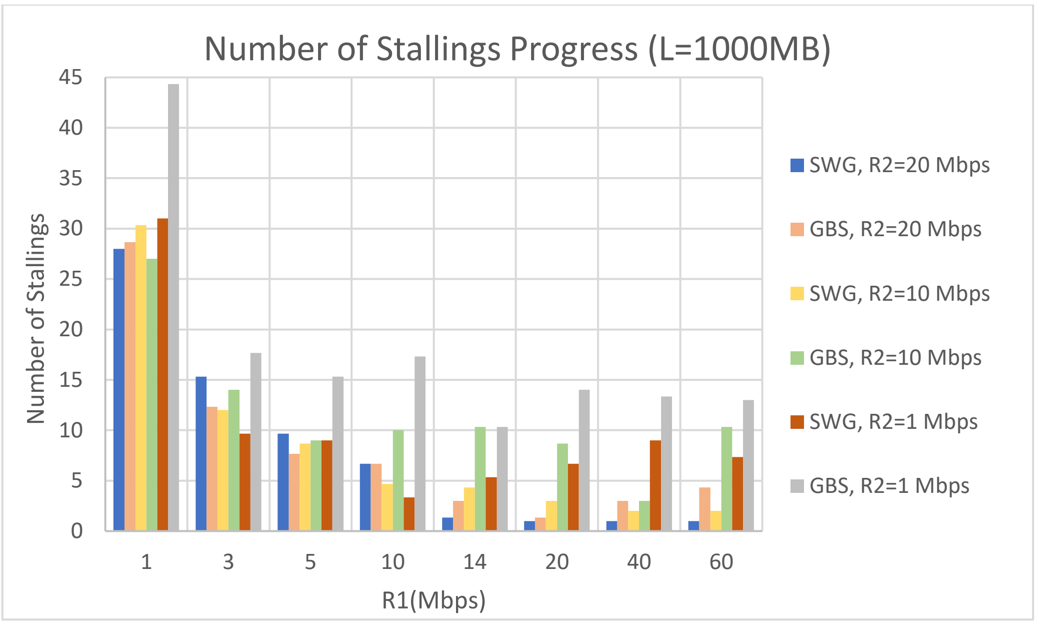


Figure 94: Number of stallings vs R1 for L=1000 MB

- To highlight a clear conclusion, it is possible that more samples were needed.
- Generally, there is a tendency for stallings to decrease while the R1 increases and while the buffer increases, since more segments are found cached and the media player uses them in his favor to avoid freezing.
- For a stable value of R1, there are more stallings for lower values of R2.
- The GBS approach compared to the SWG presents more stallings in the majority of cases. If we had more samples, the GBS approach would always have more stallings.
- The existence of a larger buffer benefits in terms of stallings, mostly in cases of higher R1 values. In cases of lower R1 values, there is also an improve in terms of stallings, but it is a smaller improve compared to higher R1 values.

7.6.2 QoE metrics VS Buffer Size L

The average resolution progress, as the buffer size is increased is shown in the next figure.

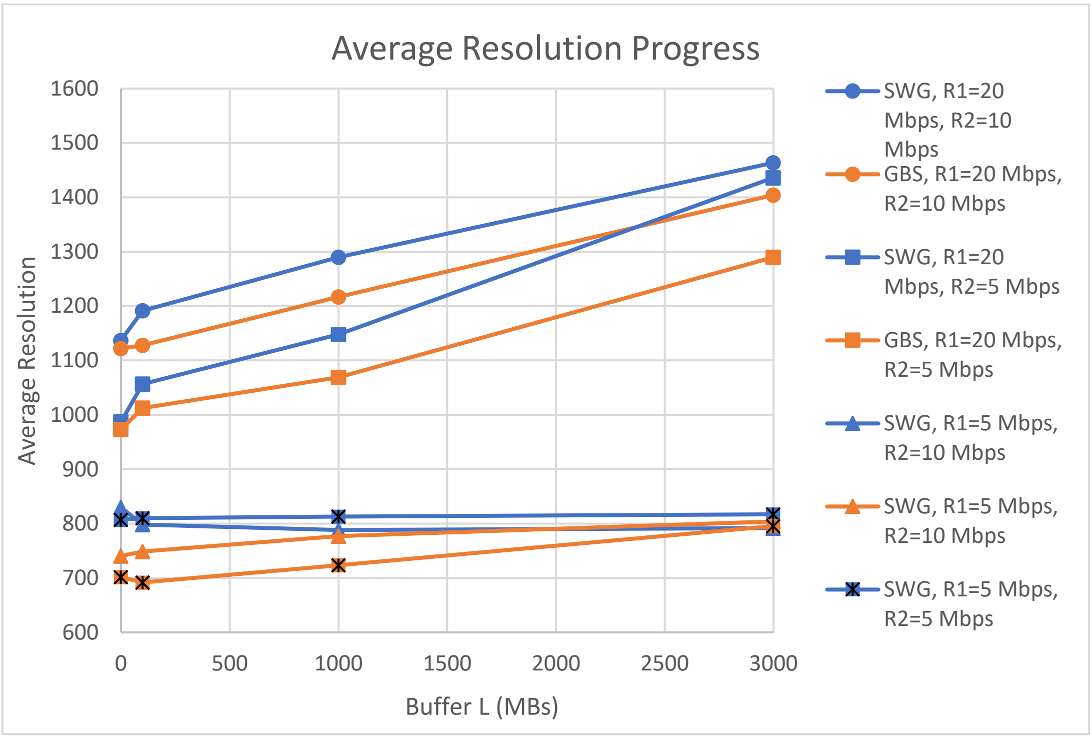


Figure 95: Average resolution vs buffer size L

We can observe that:

- The increase of the buffer size creates a satisfying profit over resolution, in the case of larger R1 values.
- If the R1 is low, there is a slightly smaller increase, or no increase at all, which means that caching is not always beneficial, but should be accompanied with high-speed channels.
- The SWG approach in this diagram is much better than the GBS.
- For R1= 20Mbps and R2 = 10Mbps in the SWG approach, we notice that the average resolution for L= 0 is 1100. If we decide to up the buffer size at L = 100MB, the average resolution only increases by 10%, reaching a value of 1200. If we increase the buffer size from 100MB to 1000MB, the expected increase of the average resolution should be more than 10%, but in our case it’s just a 10% increase. The question is if any vendor prefers to waste resources in terms of storage for just a small improvement in the resolution.

As for the backhaul throughput progress over the buffer size (Figure 96), we can conclude that:

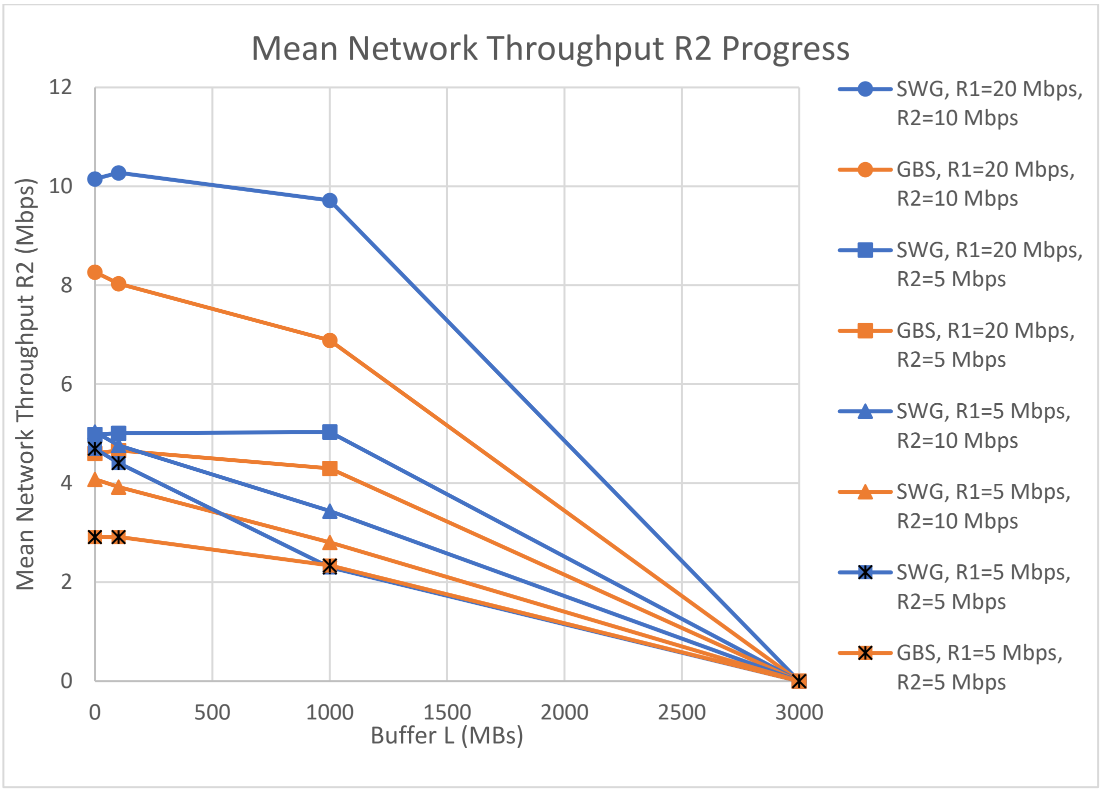


Figure 96: Mean network throughput progress vs buffer size L

- The underutilization of the channels is clear, especially in the GBS approach.
- This is demonstrated more clearly when the buffer size is big enough and a large portion of segments are cached. In this case there is a big waste of resources, especially for larger R2 values.
- If we dealt with a multithread approach or a cache aware HAS algorithm, then the curve would not fall progressively to zero (until all the segments of main will be cached), but the available R2 would be used to download even more segments proactively, optimizing the QoE of the client.

Now, if we compare the number of stallings that have occurred for two different fronthaul rate values, we can say that :

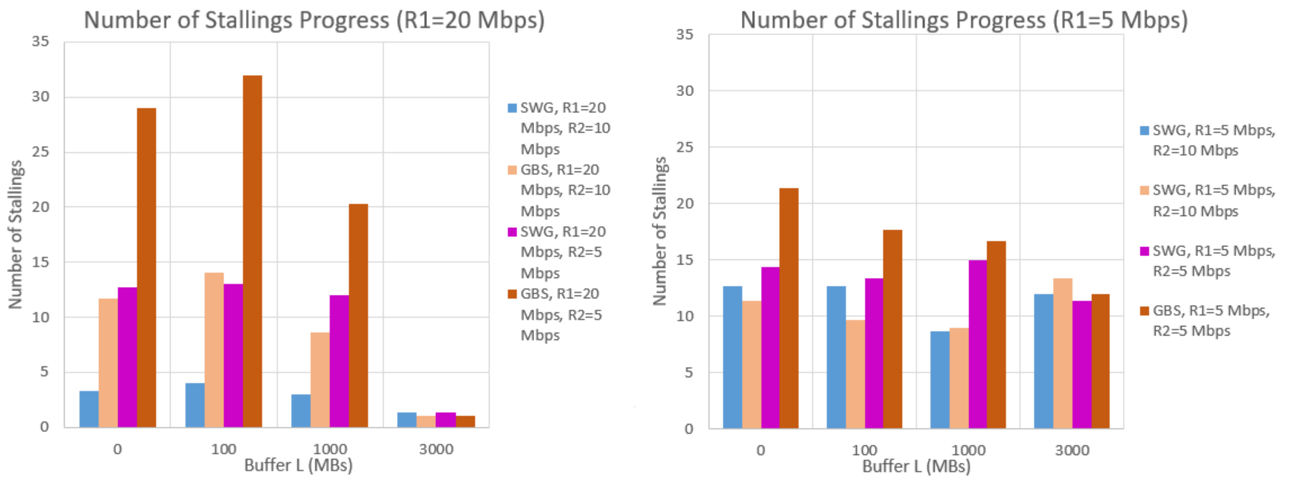


Figure 97: Number of stallings vs buffer size for R1= 5 Mbps and R1= 20Mbps

- In both cases, the GBS approach has more stallings than the SWG approach. This is clear for R1 = 20Mbps.
- The higher R1 values can benefit to a larger extent from the increase of the buffer size and decrease the stallings in those cases.
- If the value of the R1 is low, there is no reason for any vendor to up the available resources and provide extra buffer size, since, at the end of the day, the QoE perceived by the client will not show many differences on the stallings perspective. Stallings will still occur, and they will annoy the client.

8. CONCLUSION AND FUTURE WORK

In this thesis, we examined how to deliver DASH-based content with the help of an intermediate server, apart from the main server, in order to demonstrate possible caching benefits for different sizes of intermediate storage servers. The goal was to exploit and take full advantage of the available bandwidth, along with the advantages of the caching approach. For this reason, we created a simple client-cache-main topology in the Mininet environment to implement the HTTP Adaptive logic, where if the cache server contained some segments, it served them immediately. Otherwise, the main server was delivering the missing segments. However, the limitations imposed on us by Mininet, the difficulty of the automation process and the under-utilization of the channels concerning the available bandwidth, pushed us to try another approach outside of Mininet.

With this new approach, we changed the testing environment and tried to compare the basic Get Before Send logic of exchanging requests (a similar logic implemented in Mininet), with a slightly smarter Send While Get approach, so that it can serve some requests at the same time of requesting the next segment. Our aim was to increase the performance of our experiment to some extent and, consequently, the Quality of Experience of the client while watching the video.

The two main findings of our research are that the SWG logic enhances the performance of almost all QoE-wised metrics, especially when accompanied by a satisfactory caching policy, and reduces the under-utilization that we observed in our initial approach. We also noticed that the chosen HAS algorithm plays an important role in our experiments due to the dangers hidden by its over-optimism of which segment (in terms of resolution) will be the next to be played. So, if the media player is not aware of what is available on the intermediate server, this may lead to many stallings happening in the experiment, resulting in the loss of any benefit we had due to caching.

So, moving in this direction, and knowing that it is difficult to manipulate with the VLC code to change the HAS algorithm, we tried to give an idea for a future work on this issue. It would be quite interesting if the local server that we have placed close to the client, was configured to receive proactively the remaining segments that are very likely to be chosen by the media player, because in our case the local server does not have a specific responsibility. This will give even better results in the user experience.

Finally, the caching algorithm that we used in all the implementations, responsible for filling the buffer of the intermediate server, is random and does not contain any particular intelligence in how to fill the cache with segments. An implementation of a smarter algorithm for future work is proposed, not based on randomness, but on the possibility of playing certain segments instead of others. For example, if the channels had a traffic of 5Mbps, then there is no reason to cache the segments of the 4K resolution, but certainly smaller than that. This would point out the assistance provided by the caching policy to all video streaming services.

APPENDIX A

Chapter 6 code files

1. bash.sh

```
#!/bin/bash

# THIS SCRIPT CONVERTS EVERY MP4 (IN THE CURRENT FOLDER AND SUBFOLDER) TO A
MULTI-BITRATE VIDEO IN MP4-DASH
# For each file "videoname.mp4" it creates a folder "dash_videoname"
containing a dash manifest file "stream.mpd" and subfolders containing video
segments.



MYDIR=$(dirname $(readlink -f ${BASH_SOURCE[0]}))
SAVEDIR=$(pwd)

# Check programs
if [ -z "$(which ffmpeg)" ]; then
   echo "Error: ffmpeg is not installed"
   exit 1
fi

if [ -z "$(which MP4Box)" ]; then
   echo "Error: MP4Box is not installed"
   exit 1
fi

cd "$MYDIR"

TARGET_FILES=$(find ./ -maxdepth 1 -type f \( -name "*.mov" -or -name "*.mp4"
\))
for f in $TARGET_FILES
do
 fe=$(basename "$f") # fullname of the file
 f="${fe%.*}" # name without extension

 if [ ! -d "${f}" ]; then #if directory does not exist, convert
   echo "Converting \"$f\" to multi-bitrate video in MPEG-DASH"

   ffmpeg -i "${fe}" -c:a copy -vn "${f}_audio.mp4"
   ffmpeg -i "${fe}" -an -c:v libx264 -x264opts 'keyint=30:min-keyint=30:no-
```

```
scenecut' -b:v 45000k -maxrate 45000k -bufsize 90000k -vf 'scale=-1:2160'
"${f}_2160.mp4" -async 1 -vsync 1

   ffmpeg -i "${fe}" -an -c:v libx264 -x264opts 'keyint=30:min-keyint=30:no-
scenecut' -b:v 16000k -maxrate 16000k -bufsize 32000k -vf 'scale=-1:1440'
"${f}_1440.mp4" -async 1 -vsync 1

   ffmpeg -i "${fe}" -an -c:v libx264 -x264opts 'keyint=30:min-keyint=30:no-
scenecut' -b:v 8000k -maxrate 8000k -bufsize 16000k -vf 'scale=-1:1080'
"${f}_1080.mp4" -async 1 -vsync 1

   ffmpeg -i "${fe}" -an -c:v libx264 -x264opts 'keyint=30:min-keyint=30:no-
scenecut' -b:v 5000k -maxrate 5000k -bufsize 10000k -vf 'scale=-1:720'
"${f}_720.mp4" -async 1 -vsync 1

   ffmpeg -i "${fe}" -an -c:v libx264 -x264opts 'keyint=30:min-keyint=30:no-
scenecut' -b:v 2500k -maxrate 2500k -bufsize 5000k -vf 'scale=-1:478'
"${f}_480.mp4" -async 1 -vsync 1

   ffmpeg -i "${fe}" -an -c:v libx264 -x264opts 'keyint=30:min-keyint=30:no-
scenecut' -b:v 1000k -maxrate 1000k -bufsize 2000k -vf 'scale=-1:360'
"${f}_360.mp4" -async 1 -vsync 1


   rm -f ffmpeg*log*

   # if audio stream does not exist, ignore it

   if [ -e "${f}_audio.mp4" ]; then


      #MP4Box -dash 1000 -rap -frag-rap -bs-switching no -profile
"dashavc264:live" "${f}_1080.mp4" "${f}_720.mp4" "${f}_480.mp4"
"${f}_360.mp4" "${f}_242.mp4" "${f}_audio.mp4" -out "manifest.mpd"

      MP4Box -dash 5000 -rap -frag-rap -segment-name %s_ "video_2160.mp4"
"video_1440.mp4" "video_1080.mp4" "video_720.mp4" "video_480.mp4"
"video_360.mp4" "video_audio.mp4" -out "manifest.mpd"


   fi

   # create a jpg for poster. Use imagemagick or just save the frame directly
from ffmpeg is you don't have cjpeg installed.

   ffmpeg -i "${fe}" -ss 00:00:00 -vframes 1  -qscale:v 10 -n -f image2 - |
cjpeg -progressive -quality 75 -outfile "${f}"/"${f}".jpg


   fi


done


cd "$SAVEDIR"
```

2. SimpleHTTPServer.py

```
 """Simple HTTP Server.

This module builds on BaseHTTPServer by implementing the standard GET
```

```
and HEAD requests in a fairly straightforward manner.
"""
__version__ = "0.6"

__all__ = ["SimpleHTTPRequestHandler"]
import os
import posixpath
import BaseHTTPServer
import urllib
import cgi
import sys
import shutil
import mimetypes
try:
   from cStringIO import StringIO
except ImportError:
   from StringIO import StringIO



class SimpleHTTPRequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):

   """Simple HTTP request handler with GET and HEAD commands.
   This serves files from the current directory and any of its
   subdirectories.  The MIME type for files is determined by
   calling the .guess_type() method.
   The GET and HEAD requests are identical except that the HEAD
   request omits the actual contents of the file.
   """

   server_version = "SimpleHTTP/" + __version__

   def do_GET(self):
       """Serve a GET request."""
       f = self.send_head()
       if f:
           self.copyfile(f, self.wfile)
           f.close()

   def do_HEAD(self):
       """Serve a HEAD request."""
```

```
        f = self.send_head()
        if f:
            f.close()

    def send_head(self):
        """Common code for GET and HEAD commands.
        This sends the response code and MIME headers.
        Return value is either a file object (which has to be copied
        to the output file by the caller unless the command was HEAD,
        and must be closed by the caller under all circumstances), or
        None, in which case the caller has nothing further to do.
        """
        path = self.translate_path(self.path)
        f = None
        if os.path.isdir(path):
            if not self.path.endswith('/'):
                # redirect browser - doing basically what apache does
                self.send_response(301)
                self.send_header("Location", self.path + "/")
                self.end_headers()
                return None
            for index in "index.html", "index.htm":
                index = os.path.join(path, index)
                if os.path.exists(index):
                    path = index
                    break
            else:
                return self.list_directory(path)
        ctype = self.guess_type(path)
        try:
            # Always read in binary mode. Opening files in text mode may cause
            # newline translations, making the actual size of the content
            # transmitted *less* than the content-length!
            f = open(path, 'rb')
        except IOError:
            self.send_error(404, "File not found")
            return None
        self.send_response(200)
        self.send_header("Content-type", ctype)
        fs = os.fstat(f.fileno())
        self.send_header("Content-Length", str(fs[6]))
        self.send_header("Last-Modified", self.date_time_string(fs.st_mtime))
        self.end_headers()
        return f

    def list_directory(self, path):
        """Helper to produce a directory listing (absent index.html).
        Return value is either a file object, or None (indicating an
        error).  In either case, the headers are sent, making the
        interface the same as for send_head().
        """
        try:
            list = os.listdir(path)
        except os.error:
            self.send_error(404, "No permission to list directory")
            return None
        list.sort(key=lambda a: a.lower())
        f = StringIO()
        displaypath = cgi.escape(urllib.unquote(self.path))
        f.write('<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">')
        f.write("<html>\n<title>Directory   listing   for   %s</title>\n"   %
displaypath)
        f.write("<body>\n<h2>Directory listing for %s</h2>\n" % displaypath)
        f.write("<hr>\n<ul>\n")
        for name in list:
            fullname = os.path.join(path, name)
            displayname = linkname = name
            # Append / for directories or @ for symbolic links
            if os.path.isdir(fullname):
                displayname = name + "/"
                linkname = name + "/"
            if os.path.islink(fullname):
                displayname = name + "@"
                # Note: a link to a directory displays with @ and links with /
            f.write('<li><a href="%s">%s</a>\n'
                    % (urllib.quote(linkname), cgi.escape(displayname)))
        f.write("</ul>\n<hr>\n</body>\n</html>\n")
        length = f.tell()
        f.seek(0)
        self.send_response(200)
        encoding = sys.getfilesystemencoding()
        self.send_header("Content-type", "text/html; charset=%s" % encoding)
        self.send_header("Content-Length", str(length))
        self.end_headers()
        return f

    def translate_path(self, path):
        """Translate a /-separated PATH to the local filename syntax.
        Components that mean special things to the local file system
        (e.g. drive or directory names) are ignored.  (XXX They should
        probably be diagnosed.)
        """
        # abandon query parameters
        path = path.split('?',1)[0]
        path = path.split('#',1)[0]
        path = posixpath.normpath(urllib.unquote(path))
        words = path.split('/')
        words = filter(None, words)
        path = os.getcwd()
        for word in words:
            drive, word = os.path.splitdrive(word)
            head, word = os.path.split(word)
            if word in (os.curdir, os.pardir): continue
            path = os.path.join(path, word)
        return path

    def copyfile(self, source, outputfile):
        """Copy all data between two file objects.
        The SOURCE argument is a file object open for reading
        (or anything with a read() method) and the DESTINATION
        argument is a file object open for writing (or
        anything with a write() method).
        The only reason for overriding this would be to change
        the block size or perhaps to replace newlines by CRLF
        -- note however that this the default server uses this
        to copy binary data as well.
        """
        shutil.copyfileobj(source, outputfile)

    def guess_type(self, path):
        """Guess the type of a file.
        Argument is a PATH (a filename).
        Return value is a string of the form type/subtype,
        usable for a MIME Content-type header.
        The default implementation looks the file's extension
        up in the table self.extensions_map, using application/octet-stream
        as a default; however it would be permissible (if
        slow) to look inside the data to make a better guess.
        """

        base, ext = posixpath.splitext(path)
        if ext in self.extensions_map:
            return self.extensions_map[ext]
        ext = ext.lower()
        if ext in self.extensions_map:
            return self.extensions_map[ext]
        else:
            return self.extensions_map['']

    if not mimetypes.inited:
        mimetypes.init() # try to read system mime.types
    extensions_map = mimetypes.types_map.copy()
    extensions_map.update({
        '': 'application/octet-stream', # Default
        '.py': 'text/plain',
        '.c': 'text/plain',
        '.h': 'text/plain',
        })


def test(HandlerClass = SimpleHTTPRequestHandler,
         ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)


if __name__ == '__main__':
    test()
```

3. random_cache.sh

```
#!/bin/bash

# THIS SCRIPT GENERATES A RANDOM NUMBER BETWEEN 1 and 61 TO CHOOSE A RANDOM
SEGMENT
# IT ALSO CHOOSES A RANDOM RESOLUTION FOR THE RANDOM SEGMENT
# THIS RANDOM FILE IS COPIED IN THE CACHE FOLDER IF THE AVAILABLE BUFFER
ALLOWS IT
# AND THEN IT IS DELETED FROM THE NONCACHED BUFFER FOLDER SO THAT IT IS NO
LONGER COPIED

rm -r noncached/
cp -R public/ noncached/

folder_size=0

buffersize=$((539 * 1024))
echo “Buffer ${buffersize}”
search=1

for I in {1..366} :
do
   while [ $search -eq 1 ]
   do
        segment=$((($RANDOM%61)+ 1))
        echo “random segment generated: $segment”

        resolution=(“360” “480” “720” “1080” “1440” “2160”)
        rand_res=$(($RANDOM%6))
        echo “random resolution generated: ${resolution[$rand_res]}”

        random_file=”video_${resolution[$rand_res]}_${segment}.m4s”
        echo “random file: ${random_file}”

        #disk_usage=$(du -hs -m)

        ls_output=$(ls -l —block-size=KB “noncached/${random_file}”)
        echo “${ls_output}”
        #    OUTPUT:    -rw-r—r—1    valia    valia    3MB    Jan    14    08:36
noncached/video_1080_2.m4s
```

```
        if [[ $ls_output != *”cannot access”* ]]; then
            search=0

            segment_size=${ls_output%kB*}  # retain the part before the kB
            segment_size=${segment_size##*valia}  # retain the part after the
last valia
            echo “segment size: $segment_size”

        fi
    done



    total=”$((folder_size + segment_size))”

    if [[ $total -lt $buffersize && search -eq 0 ]]; then
        echo “Segment will be copied in the cache”

        #cp file
        cp -n noncached/${random_file} 2cache/
        echo “File is copied”
        echo “${random_file}” >> files_in_cache.txt

        folder_size=”$(($segment_size + folder_size))”
        echo “FOLDER SIZE--------- ${folder_size}”

        #delete file after it has been put in cache
        delete=$(rm -rf “noncached/${random_file}”)
    else
        echo “Cache is full”
    fi

    search=1
done
echo “done”
```

4. r1.sh

```
#!/bin/bash
```

```
rate=()


i=0


while :
do


   tc qdisc replace dev s3-eth1 root handle 1: tbf rate 100mbit burst 250000
limit 500000
   tc qdisc replace dev s2-eth2 root handle 1: tbf rate 100mbit burst 250000
limit 500000


   tc -r qdisc show dev s3-eth1


   i=$((i+1))
   if [ $i == 3001 ]; then    # 3000 einai h teleytaia thesi tou pinaka, kai
meta tha ksekinaei ap oth  arxh
    echo "from scratch"
       i=0
   fi


   sleep 1




done
```

APPENDIX B

Chapter 7 code files

1. automation.py

```
#!/usr/bin/env python
import os
import random
import re
import shutil
import subprocess
import time


def clear_folder(path):
   directory = path
   files_in_directory = os.listdir(directory)
   mpd_files = [file for file in files_in_directory if file.endswith(".mpd")]
   mp4_files = [file for file in files_in_directory if file.endswith(".mp4")]
   m4s_files = [file for file in files_in_directory if file.endswith(".m4s")]

   for file in mpd_files:
       path_to_file = os.path.join(directory, file)
       os.remove(path_to_file)
   for file in mp4_files:
       path_to_file = os.path.join(directory, file)
       os.remove(path_to_file)
   for file in m4s_files:
       path_to_file = os.path.join(directory, file)
       os.remove(path_to_file)


def fill_cache(buffers_size):
   files_in_directory = os.listdir('./public')
   segments_size = {}
   sum_size = 0.0
   attempts = 0
   transfer_segments = []
   for segment in files_in_directory:
       if segment.endswith('.m4s') and 'audio' not in segment:
           segments_size[segment]    =    int(os.path.getsize('./public/'    +
segment)) / 1000000
   segments = list(segments_size.keys())
   while True:
       print('---------------------------------------------')
       rd = random.randint(0, len(segments_size.keys()) - 1)
       current_segment = segments[rd]
       sum_size = sum_size + segments_size[current_segment]
       print('Choose  segment  -->  '  +  current_segment  +  '  with  size  '  +
str(segments_size[current_segment]))
       if sum_size <= buffers_size:
           if current_segment not in transfer_segments:
               print("copy  to  cache  -->  "  +  current_segment  +  "  with  total
size " + str(sum_size))
               shutil.copy('./public/' + current_segment, './cache')
               transfer_segments.append(current_segment)
           else:
               print('I have already copy file ' + current_segment + ', so I
will remove the size')
               sum_size = sum_size - segments_size[current_segment]
       else:
           if attempts == 20:
               print("reached maximum capacity")
               break
           else:
               print('Trying one more time...')
               print()
               sum_size = sum_size - segments_size[current_segment]
               attempts += 1



def copy_all_to_public():
   files_in_directory = os.listdir('./public')
   for file in files_in_directory:
       if      file.endswith('.m4s')      or      file.endswith('.mpd')      or
file.endswith('mp4'):
           shutil.copy('./public/' + file, './cache')



def kill_all(name, vlc=False):
   if vlc:
       child    =    subprocess.Popen(["ps    aux    |    grep    -i    'vlc'"],
stdout=subprocess.PIPE, shell=True)
   else:
       child  =  subprocess.Popen(["ps  aux  |  grep  -i  '"  +  name  +  "'"],
stdout=subprocess.PIPE, shell=True)
   result = child.communicate()[0].decode('utf-8').split('\n')
   for i in result:
       i = re.sub(' +', ' ', i)
       parts = i.split(' ')
       this_name = ''
       if len(parts) < 1:
           continue
       if vlc:
           if len(parts) > 10:
               if 'vlc' in parts[10]:
                   os.system('kill ' + parts[1])
       else:
           for j in parts[10:]:
               this_name = this_name + j + ' '
           if this_name == name + ' ':
               os.system('kill ' + parts[1])



# -------------------------Open terminals and start running experiments-----
------------------------

r1_values = [5, 10, 20, 50, 100]
r2_values = [0.5, 2.5, 5, 10, 20, 25, 50, 100, 200, 500, 1000]
buffer_size = [0, 100, 500, 1000, 3000]
samples = [1, 2, 3]

print('        (r1, r2, L, sample)')
for r1 in r1_values:
   for r2 in r2_values:
       for L in buffer_size:
           for sample in samples:
               # ----------Clear Local Folder-----------------
               clear_folder("./local")
               # ---------------Clear Cache Folder------------
               clear_folder("./cache")
               # ---------------------------------------------
               if L == 0:
                   pass
```

```
            elif L == 3000:
                copy_all_to_public()
            else:
                fill_cache(L)
            time.sleep(5)
            print('Running (' + str(r1) + ',' + str(r2) + ', ' + str(L) +
', ' + str(sample) + ')')
            main_command = "python3 main.py -a 127.0.0.4 -p 8004 -s1
127.0.0.4 -p1 8004 -s2 127.0.0.4 -p2 8004 -r " + str(r2)
            # smart proxy
            proxy_command = "python3 proxy-send-while-get.py -a 127.0.0.3
-p 8003 -s1 127.0.0.4 -p1 8004 -s2 127.0.0.4 -p2 8004 -r1 " + str(r1) + " -
r2 " + str(r2) + " -l " + str(L) + " -sample " + str(sample)
            # dummy proxy
            # proxy_command = "python3 proxy-get-before-send.py -a
127.0.0.3 -p 8003 -sa 127.0.0.4 -sp 8004 -r1 " + str(r1) + " -r2 " + str(r2)
+ " -l " + str(L) + " -sample " + str(sample)
            local_command = "python3 local.py -a 127.0.0.2 -p 8002 -s1
127.0.0.3 -p1 8003 -s2 127.0.0.3 -p2 8003"
            client_command = "vlc-wrapper
http://127.0.0.2:8002/manifest.mpd"

            main = subprocess.Popen(["gnome-terminal -- " + main_command],
cwd='public/', shell=True)
            time.sleep(1)
            proxy = subprocess.Popen(["gnome-terminal -- " +
proxy_command], cwd='cache/', shell=True)
            time.sleep(1)
            local = subprocess.Popen(["gnome-terminal -- " +
local_command], cwd='local/', shell=True)
            time.sleep(1)
            client = subprocess.Popen(["gnome-terminal -- " +
client_command], shell=True)

            counter = 0
            while True:
                time.sleep(2)
                file_path = 'cache/output_' + str(float(r1)) + '_' +
str(float(r2)) + '_' + str(L) + '_' + str(
                    sample) + '.txt'
                if os.path.exists(
                        file_path) or counter == 2400:  # if counter reach
1800 it means that 1800*2=3600 seconds passed or 3600 / 60 = 1 hour passed
                    if counter == 2400:
                        print(
                            'Tuple (r1,r2) has errors --> (' + str(r1) +
', ' + str(r2) + ',' + str(L) + ', ' + str(
                                    sample) + ')')
                    elif counter == 0:
                        print('Experiment (' + str(r1) + ', ' + str(r2) +
') already exists in cache')
                        # Proxy output has been completed
                    else:
                        print('Experiment completed')
                    print()
                    time.sleep(1)
                    kill_all(main_command)
                    if counter == 0:
                        kill_all(proxy_command)
                    kill_all(local_command)
                    # kill_all(client_name)
                    kill_all(client_command, vlc=True)
                    time.sleep(2)
                    break
                counter += 1
```

2. raw.py

```
  import re
import time
import os
import math
import glob
import sys

resolution_to_altitude = {'360': 1, '480': 2, '720': 3, '1080': 4, '1440': 5,
'2160': 6}
outputs = []

if len(sys.argv) > 1:
   outputs = [str(sys.argv[1])]
else:
   for q in glob.glob('outputs/*.txt'):
       outputs.append(q[8:])
output_file_number = None
file = open('metrics/all_metrics.csv', 'w')
file.writelines(['Name,r1,r2,L(buffer  size),#sample,MOS(resolution),Average
Resolution,Sum of Switches,Average Altitude,Average Video Bitrate,'
                'Mean    Network    Throughput    r1,Mean    Network    Throughput
r2,MOS(stallings),Stallings,'
                'Total  Stalling  Time,Mean  Stalling  Time,Total  Network  Usage
Time,Initial Playback Delay,'
                'Number  of  cached  bits  delivered,Number  of  non-cached  bits
delivered,Cache Miss Ratio,'
                'Cache Hit Ratio,Backhaul traffic ratio\n'])
file.close()
for output_file in outputs:
   if len(sys.argv) > 1:
       output_file_number = re.search('_.*\.txt', output_file).group()
       print(output_file_number)

   file = open('outputs/' + output_file, 'r')
   Lines = file.readlines()

   data_to_csv = []

   pattern = '%Y-%m-%d %H:%M:%S'
   resolution_before = None
   seconds_before = None
   my_lines = []

   manifest_time = 0
   td_1 = 0
   number_of_stallings = 0
   sum_mos_res = 0
   sum_stalling_duration = 0
   sum_size = 0
   sum_bitrate = 0
   sum_resolution = 0
   sum_switches = 0
   sum_cache_hit = 0
   sum_size_cache_hit = 0
   sum_altitude = 0
   time_1st = 0
   time_61st = 0

   for line in Lines:
       if line.startswith('INFO:root:[') and 'audio' not in line:
           if 'manifest.mpd' in line:
               date_time_m = re.search('\[([^.]+)', line).group()[1:]
               manifest_time   =   int(time.mktime(time.strptime(date_time_m,
pattern)))

           if 'video_' in line:
               video_name = line.split(' ')[2]
               date_time = re.search('\[([^.]+)', line).group()[1:]
               resolution = re.search('_.*_', line).group()[1:-1]
               segment = re.search('\_[0-9]*\.', line).group()[1:-1]

               # timestamp in epoch time
               seconds = int(time.mktime(time.strptime(date_time, pattern)))

               if resolution == '360':
                   mos_res = 2.07744
               elif resolution == '480':
                   mos_res = 3.02246
               elif resolution == '720':
                   mos_res = 3.97185
               elif resolution == '1080':
                   mos_res = 4.47112
               elif resolution == '1440':
                   mos_res = 4.52586
               elif resolution == '2160':
                   mos_res = 4.58036

               sum_mos_res += mos_res

               if resolution_before is None:
                   resolution_before = resolution

               if seconds_before is None:
                   seconds_before = seconds
               seconds_diff = seconds - seconds_before
               sum_resolution += int(resolution)
               td_2 = td_1 + seconds_diff - 5

               if segment == '1':
                   time_1st = seconds
                   td_1 = 0
                   td_2 = 0
               if segment == '61':
                   time_61st = seconds

               if td_2 > 0:
                   number_of_stallings += 1
                   stalling = 1
                   stalling_duration = td_2
                   td_1 = 0
               else:
                   stalling = 0
                   stalling_duration = 0
                   td_1 = td_2

               sum_stalling_duration += stalling_duration

               # Size calculation
               size = int(os.path.getsize(video_name))
               sum_size += size

               # Bitrate calculation to bps
               bitrate = (size / 5) * 8
               # to mbps
               bitrate = bitrate / 1000000
               sum_bitrate += bitrate

               # Altitude
               altitude   =   abs(resolution_to_altitude.get(resolution)   -
resolution_to_altitude.get(resolution_before))
               sum_altitude += altitude

               switch = 0
               if resolution != resolution_before:
                   switch = 1
                   sum_switches += switch

               #     ip     =     re.search('[0-9]*\.[0-9]*\.[0-9]*\.[0-9]',
line).group()
               ip = line.split(' ')[-1].rstrip()
               cache_hit = 0
               if ip == 'HIT':
                   cache_hit = 1
```

```
                    sum_cache_hit += 1
                    sum_size_cache_hit += size

                momentary_network_usage = seconds - manifest_time
                momentary_throughput_r1 = ((sum_size * 8) /
momentary_network_usage) / 1000000  # Mbps
                momentary_throughput_r2 = (((sum_size - sum_size_cache_hit) *
8) / momentary_network_usage) / 1000000  # Mbps
                if momentary_throughput_r2 == 0.0:
                    r1_divided_by_r2 = 0
                else:
                    r1_divided_by_r2 = momentary_throughput_r1 /
momentary_throughput_r2

                print('----------')
                print(line)
                print('video_name', video_name)
                print('CACHE ' + str(ip))
                print('date_time', date_time)
                print('resolution_before', resolution_before)
                print('resolution', resolution)
                print('size', size)
                print('bitrate', bitrate)
                print('segment', segment)
                print('seconds', seconds)
                print('seconds_before', seconds_before)
                print('seconds_diff', seconds_diff)
                print('altitude', altitude)
                print('sum_size', sum_size)
                data_to_csv.append(
                    str(segment) + ', ' + str(resolution) + ', ' + str(switch)
+ ', ' + str(altitude) + ', ' + str(bitrate) + ', ' + str(cache_hit) + ', ' +
str(size) + ', ' + str(stalling) + ', ' + str(mos_res) + ', ' +
str(seconds_diff) + ', ' + str(stalling_duration) + ', ' +
str(momentary_throughput_r1) + ', ' + str(momentary_throughput_r2) + '\n')
                print('----------')

                resolution_before = resolution
                seconds_before = seconds

            my_lines.append(line)

    '''
```

```
        METRICS
    '''

    if time_1st == 0 or time_61st == 0 or manifest_time == 0:
        raise Exception('cant find 1st or 61st segment or manifest.mpd')

    avg_bitrate = sum_bitrate / 61
    avg_altitude = sum_altitude / 61
    avg_resolution = sum_resolution / 61
    avg_mos_res = sum_mos_res / 61
    percentage_cache_hit = (sum_cache_hit / 61) * 100

    percentage_cache_miss = ((61 - sum_cache_hit) / 61) * 100

    sum_size_cache_miss = sum_size - sum_size_cache_hit
    traffic_ratio_percentage = (sum_size_cache_miss / sum_size) * 100
    if number_of_stallings != 0:
        mean_stalling_time = sum_stalling_duration / number_of_stallings
    else:
        mean_stalling_time = 0
    mos_stallings  =  (3.5  *  math.exp((-(0.15  *  mean_stalling_time  +  0.19))  *
number_of_stallings)) + 1.5
    network_usage = time_61st - manifest_time

    throughput_r1 = ((sum_size * 8) / network_usage) / 1000000  # Mbps
    throughput_r2  =  ((sum_size_cache_miss  *  8)  /  network_usage)  /  1000000   #
Mbps

    initial_playback_delay = time_1st - manifest_time


    '''
    write to csv
    '''

    all_metrics = [

'=============================================================================
==================================' +
        '\nAverage                                                       Resolution
= ' + str(avg_resolution) +
        '\n---------------------------------------------------------------------
------------------------------------------' +
        '\nSum                          of                              Switches
```

```
= ' + str(sum_switches) +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nAverage                                                        Altitude
= ' + str(avg_altitude) +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nAverage                         Video                           Bitrate
= ' + str(avg_bitrate) + ' Mbps' +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nMean  Network  Throughput  r1  =  (non-cached  +cached  bits)  /  total
network usage time  = ' + str(throughput_r1) + ' Mbps' +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nMean Network Throughput r2 = non-cached / total network usage time
= ' + str(throughput_r2) + ' Mbps' +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nMOS                         (for                           stallings)
= ' + str(mos_stallings) +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nMOS                        (for                        resolutions)
= ' + str(avg_mos_res) +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nStallings                                                            =
' + str(number_of_stallings) +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nTotal                         Stalling                             Time
= ' + str(sum_stalling_duration) + ' seconds' +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nMean                          Stalling                             Time
= ' + str(mean_stalling_time) + ' seconds' +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nTotal               Network               Usage                    Time
= ' + str(network_usage) + ' seconds' +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nInitial                         Playback                           Delay
= ' + str(initial_playback_delay) + ' seconds' +
       '\n-----------------------------------------------------------------------------
------------------------------------------' +
       '\nNumber         of          cached          bits          delivered
= ' + str(sum_size_cache_hit) + ' bits' +
       '\n-------------------------------------------------------------------------
-------------------------------------------' +
       '\nNumber         of         non-cached         bits         delivered
= ' + str(sum_size_cache_miss) + ' bits' +
       '\n-------------------------------------------------------------------------
-------------------------------------------' +
       '\nCache                          Miss                          Ratio
= ' + str(percentage_cache_miss) + '%' +
       '\n-------------------------------------------------------------------------
-------------------------------------------' +
       '\nCache                          Hit                           Ratio
= ' + str(percentage_cache_hit) + '%' +
       '\n-------------------------------------------------------------------------
-------------------------------------------' +
       '\nBackhaul  traffic  ratio  =  non  cached  bits  /  total  size
= ' + str(traffic_ratio_percentage) + '%' +

'\n===========================================================================
====================================']
   file.close()
   if len(sys.argv) > 1:
       file1 = open('metrics/metrics' + output_file_number, "w")
   else:
       file1 = open("metrics/metrics" + str(output_file[6:]), "w")

   file1.writelines(['Segment  Number,  Resolution,  Switch,  Altitude,  Video
Bitrate (Mbps), Cache hit ,Size (Bytes) ,'
                     'Stallings, MOS, Segment Duration, Stalling Duration,
R1, R2 \n'] + data_to_csv + all_metrics)
   file1.close()
   with open('metrics/all_metrics.csv', 'a') as fd:
       parts = output_file.split('_')
       fd.write(

str([output_file,float(parts[1]),float(parts[2]),int(parts[3]),int(parts[4][
:-4]), avg_mos_res, avg_resolution,
                sum_switches,    avg_altitude,avg_bitrate,    throughput_r1,
throughput_r2, mos_stallings,
                number_of_stallings,                    sum_stalling_duration,
mean_stalling_time, network_usage, initial_playback_delay,
                sum_size_cache_hit,                         sum_size_cache_miss,
percentage_cache_miss, percentage_cache_hit,
                traffic_ratio_percentage])[1:-1] + '\n')
```

ABBREVIATIONS - ACRONYMS

1G	First-Generation Cellular Network
2G	Second-Generation Cellular Network
3G	Third-Generation Cellular Network
3GPP	3rd Generation Partnership Project
4G	Fourth-Generation Cellular Network
5G	Fifth-Generation Cellular Network
6G	Sixth- Generation Cellular Network
7G	Seventh- Generation Cellural Network
ABR	Adaptive Bit Rate
ABS	Adaptive Bitrate Streaming
ABSF	Almost Blank Subframes
API	Application Programming Interface
ARP	Allocation and Retention Priority
AT&T	American Telephone and Telegraph Company
BDA	Big Data Analytics
BSS	Base Station Subsystem
BW	Bandwidth
CDD	Code Division Duplex
CDM	Content Decryption Modules
CDMA	Code Division Multiple Access
CDN	Content Delivery Networks

CoMP	Coordinated Multi-Point transmission and reception
DASH	Dynamic Adaptive Streaming
DECE	Digital Entertainment Content Ecosystem
DECT	Digitally Enhanced Cordless Telecommunications
DeNB	Donor-eNB
DFT	Discrete Fourier Transform
DL	Downlink
DRM	Digital Right Management
EDA	Exploratory Data Analysis
EDGE	Enhanced Data-rates for Global Evolution
elCIC	enhanced Inter-Cell Interference Coordination
EME	Encrypted Media Extensions
eNB	evolved Base Stations
EPC	Evolved Packet Core
EPS	Evolved Packet System
E-UTRA	Evolves Terrestrial Radio Access Network
FDD	Frequency Division Duplex
FHD	Full High Definition
FOMA	Freedom of Mobile Multimedia Access
GBS	Get Before Send
GOP	Group of Pictures
GPRS	General Packet Radio Service

GSM	Global System for Mobile
GUI	Graphical User Interface
HAS	HTTP Adaptive Streaming
HCR	High Chip Rate
HD	High Definition
HLS	HTTP Live Streaming
HSPA	High Speed Packet Access
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IFFT	Inverse Fast Fourier Transform
IoT	Internet Of Things
IP address	Internet Protocol address
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ISDN	Integrated Services Digital Networks
ITU	International Telecommunication Union
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KPI	Key Performance Indicator
LAN	Local Area Network
LTE	Long Term Evolution

LTE-A	LTE Advanced
MIMO	Multiple Input Multiple Output
ML	Machine Learning
MME	Mobility Management Entity
MMS	Multimedia Message
MOS	Mean Opinion Score
MPD	Media Presentation Description
MPEG	Motion Picture Expert Group
MS	Mobile Station
MSE	Media Source Extension
MU-MIMO	Multiple User- Multiple Input Multiple Output
NFV	Network Function Virtualization
NSS	Network and Switching Subsystem
NTT	Nippon Telegraph and Telephone
ODL	OpenDaylight
OFDM	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
OS	Operating System
OSGi	Open Service Gateway initiative
OSS	Operation and Support Subsystem
OTT	Over The Top
PGW	Packet Data Network

PyPI	Python Package Index
QAM	Quadrature Amplitude Modulation
QCI	QoS Class Identifier
QoE	Quality of Experience
QoS	Quality of Service
QPSK	Quadrature Phase Shift Keying
RDA	Rate Determination Algorithm
RN	Relay Node
SC-FDMA	Single Carrier FDMA
SDN	Software Defined Networking
SINR	Signal to Interference plus Noise Ratio
SIR	Signal to Interference Ratio
S-KPIs	Service Key Performance Indicators
SNR	Signal to Noise Ratio
SSL	Secure Socket Layer
SU-MIMO	Single User- Multiple Input Multiple Output
SWG	Send While Get
TDD	Time Division Duplex
TDMA	Time Division Multiple Access
TD-SCDMA	Time Division Synchronous Code Division Multiple Access
TMA	Traffic Monitoring and Analysis
UE	User Equipment

UE-AMBR	User Equipment - Aggregate Maximum Bit Rate
UHD	Ultra-High Definition
UL	Uplink
UMTS	Universal Mobile Telecommunications System
URL	Uniform Resource Locator
UTRAN	UMTS Terrestrial Radio Access Network
VoD	Video on Demand
VoIP	Voice over Internet Protocol
WAN	Wide Area Network
WCDMA	Wideband Code Division Multiple Access
W-CDMA	Wideband Code Division Multiple Access
WiMAX	Worldwide Interoperability for Microwave Access
WWWC (W3C)	World Wide Web Consortium
WWWW	Wireless World Wide Web
XML	Extensible Markup Language

REFERENCES

[1] Anju Uttam Gawas, “An Overview on Evolution of Mobile Wireless Communication Networks: 1G-6G”, International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169 Vol 3, Issue 5, 2015.

[2] Garg, Atul. (2014). Digital Society from 1G to 5G: A Comparative Study. International Journal of Application or Innovation in Engineering & Management. Volume 3. 186 to 193.

[3] https://www.linkedin.com/pulse/mobile-wireless-communication-technology-journey-0g-mutabazi/

[4] INTRODUCTION TO DIGITAL COMMUNICATION AND COMMUNICATION NETWORKS Vijay K. Garg, Yih-Chen Wang, in The Electrical Engineering Handbook, 2005

[5] https://en.wikipedia.org/wiki/Cellular_network

[6] https://blog.xoxzo.com/en/2018/07/24/history-of-1g/

[7] https://www.ijemr.net/DOC/FeaturesAndLimitationsOfMobileGenerations(57-61).pdf

[8]https://www.intraway.com/blog/the-history-of-2g/?utm_source=rss&utm_medium=rss&utm_campaign=the-history-of-2g

[9] Sharma, Purnima & Sharma, D. & Singh, R K. (2015). Evolution of mobile wireless communication networks (0G-8G). International Journal of Applied Engineering Research. 10. 14765-14778.

[10] https://www.etsi.org/technologies/mobile/2g

[11] https://www.electronics-notes.com/articles/connectivity/2g-gsm/network-architecture.php

[12]https://www.intraway.com/blog/the-historyof2g/?utm_source=rss&utm_medium=rss&utm_campaign=the-history-of-2g

[13] Salih, Azar & Zeebaree, Subhi & Abdulraheem, Ahmed & Zebari, Rizgar & Mohammed Sadeeq, Mohammed & Ahmed, Omar. (2020). Evolution of Mobile Wireless Communication to 5G Revolution. Technology Reports of Kansai University. 62. 2139-2151.

[14] https://www.itu.int/itunews/issue/2003/06/thirdgeneration.html

[15] https://www.etsi.org/technologies/mobile/3g

[16] https://www.etsi.org/technologies/mobile/4g

[17] https://whatsag.com/mobile-technology/the-history-of-4g.php

[18] https://en.wikipedia.org/wiki/LTE_(telecommunication)

[19] https://en.wikipedia.org/wiki/LTE_Advanced

[20] https://www.cablefree.net/wirelesstechnology/4glte/lte-carrier-aggregation/

[21] https://www.3gpp.org/technologies/keywords-acronyms/97-lte-advanced

[22] https://www.electronics-notes.com/articles/connectivity/4g-lte-long-term-evolution/coordinated-multipoint-comp.php

[23] https://www.tutorialspoint.com/lte/lte_network_architecture.html

[24]https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2014_2/rafaelreis /ofdma_scfdma.html

[25] https://www.netmanias.com/en/post/blog/5933/lte-qos/lte-qos-part-2-lte-qos-parameters-qci-arp-gbr-mbr-and-ambr

[26] https://bectechnologies.net/wordpress/wp-content/uploads/2015/08/QoS.pdf

[27] https://www.etsi.org/technologies/mobile/5g

[28] https://www.thalesgroup.com/en/markets/digital-identity-and-security/mobile/inspired/5G

[29] https://www.viavisolutions.com/en-us/5g-architecture

[30] HTTP/2-Based Methods to Improve the Live Experience of Adaptive Streaming

[31] https://bitmovin.com/adaptive-streaming/

[32] https://www.encoding.com/mpeg-dash/

[33] https://www.muvi.com/wiki/mpeg-dash-dynamic-adaptive-streaming-http.html

[34] https://www.brendanlong.com/the-structure-of-an-mpeg-dash-mpd.html

[35] https://www.encoding.com/digital-rights-management-drm/

[36] Cisco (2017). Vni global IP traffic forecast, 2016 - 2021

[37] Lee, Danny & Dovrolis, Constantine & Begen, Ali. (2014). Caching in HTTP adaptive streaming: Friend or Foe? Proceedings of the 24th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV 2014. 10.1145/2578260.2578270.

[38] https://docs.broadcom.com/doc/caching-technologies-en

[39] https://www.ironistic.com/four-major-caching-types-and-their-differences/

[40] https://docs.broadcom.com/doc/caching-technologies-en

[41] https://pressidium.com/blog/2017/browser-cache-work/

[42] https://swiftcachelite.swiftserve.com/different-types-of-caching-and-their-differences/

[43] (2008) Proxy-Caching for Video Streaming Applications. In: Furht B. (eds) Encyclopedia of Multimedia. Springer, Boston, MA

[44] https://aws.amazon.com/caching/cdn

[45] https://www.cloudflare.com/learning/cdn/what-is-caching/

[46] Wang, Ying & Li, Peilong & Jiao, Lei & Su, Zhou & Cheng, Nan & Shen, Xuemin & Zhang, Ping. (2017). A Data-Driven Architecture for Personalized QoE Management in 5G Wireless Networks. IEEE Wireless Communications. 24. 102-110. 10.1109/MWC.2016.1500184WC.

[47] 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.

[48] https://en.wikipedia.org/wiki/Stevens%27s_power_law

[49] https://docplayer.gr/47701296-Quality-of-experience.html

[50] T. Abar, A. Ben Letaifa and S. El Asmi, "Objective and subjective measurement QoE in SDN networks," 2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC), Valencia, 2017, pp. 1401-1406, doi: 10.1109/IWCMC.2017.7986489.

[51] Liotou, Eirini & Tsolkas, Dimitris & Samdanis, K. & Passas, Nikos & Merakos, Lazaros. (2016). Towards Quality of Experience Management in the Next Generation of Mobile Networks.

[52] García, Antonio & Toril, Matias & Oliver Balsalobre, Pablo & Luna-Ramírez, Salvador & Garcia, Rafael. (2019). Big Data Analytics for Automated QoE Management in Mobile Networks. IEEE Communications Magazine. 57. 91-97. 10.1109/MCOM.2019.1800374.

[53] Seufert, Michael & Egger-Lampl, Sebastian & Slanina, Martin & Zinner, Thomas & Hossfeld, Tobias & Tran-Gia, Phuoc. (2014). A Survey on Quality of Experience of HTTP Adaptive Streaming. Communications Surveys & Tutorials, IEEE. 17. 10.1109/COMST.2014.2360940.

[54] https://en.wikipedia.org/wiki/Software-defined_networking

[55] https://en.wikipedia.org/wiki/Java_virtual_machine

[56] https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

[57] http://www.brianlinkletter.com/how-to-use-miniedit-mininets-graphical-user-interface/

[58] https://en.wikipedia.org/wiki/VLC_media_player

[59] https://en.wikipedia.org/wiki/FFmpeg

[60] https://linuxize.com/post/how-to-install-ffmpeg-on-ubuntu-18-04/

[61] https://en.wikipedia.org/wiki/Python_(programming_language)

[62] https://www.tecmint.com/install-pip-in-linux/

[63] https://pypi.org/project/requests/2.7.0/

[64] https://requests.readthedocs.io/en/master/

[65] https://docs.python.org/2/library/simplehttpserver.html#module-SimpleHTTPServer

[66] https://rybakov.com/blog/mpeg-dash/

[67] https://blog.desgrange.net/post/2017/04/17/encode-videos-dynamic-adaptive-streaming-http.html

[68] https://www.haivision.com/resources/streaming-video-definitions/group-of-pictures/

[69] Hossfeld, Tobias & Schatz, Raimund & Biersack, Ernst & Plissonneau, Louis. (2013). Internet Video Delivery in YouTube: From Traffic Measurements to Quality of Experience. 10.1007/978-3-642-36784-7_11.

[70] http://gain.di.uoa.gr/casper/files/Deliverables/d6.1.pdf

[71] https://numpy.org/doc/stable/reference/random/generated/numpy.random.rayleigh.html

[72]https://docs.python.org/3/library/http.server.html