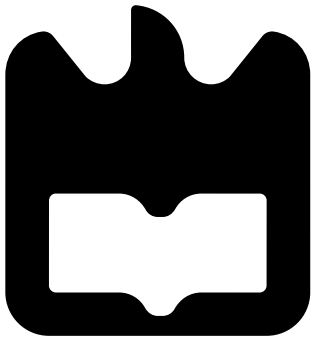
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2017

Rui Pedro Matias Dias

Distribuição de Conteúdos Over-The-Top Multimédia em Redes sem Fios
Multimedia Over-the-Top Content Distribution in Wireless Networks



Rui Pedro Matias Dias

Distribuição de Conteúdos Over-The-Top Multimédia em Redes sem Fios
Multimedia Over-the-Top Content Distribution in Wireless Networks

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e de Telecomunicações, realizada sob a orientação científica da Professora Doutora Susana Sargento, Professora Associada com Agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e co-orientação científica do Doutor Lucas Guardalben, Invenstigador do Instituto de Telecomunicações de Aveiro.

o júri / the jury

presidente / president

Professor Doutor António Guilherme Rocha Campos
Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Professor Doutor Pedro Nuno Miranda de Sousa
Professor Auxiliar da Universidade do Minho - Escola de Engenharia (arguente)

Doutor Lucas Guardalben
Investigador do Instituto de Telecomunicações da Universidade Aveiro (co-orientador)

agradecimentos / acknowledgements

Queria aproveitar esta oportunidade para agradecer aos meus pais, à minha irmã e avós por me possibilitarem o estudo na universidade e o curso que escolhi, bem como, todo o apoio dado durante todo o percurso do curso.

Quero agradecer ao Rodrigo Almeida e ao João Arantes por me terem acompanhado e apoiado durante o percurso académico.

Um agradecimento especial ao Adriano Fiorese pela a paciência e constante apoio e partilha de conhecimento fundamental na realização desta dissertação. Também agradeço ao José, André, Tiago e Pedro, por toda a ajuda e motivação dada no decorrer da realização da dissertação.

Agradeço á Professora Doutora Susana Sargento e Doutor Lucas Guardalben por me terem sugerido este tema e por toda a orientação e apoio dado nesta dissertação.

Obrigado a todos!...

Resumo

Hoje em dia a Internet é considerada um bem essencial devido ao facto de haver uma constante necessidade de comunicar, mas também de aceder e partilhar conteúdos. Com a crescente utilização da Internet, aliada ao aumento da largura de banda fornecida pelos operadores de telecomunicações, criaram-se assim excelentes condições para o aumento dos serviços multimédia *Over-The-Top* (OTT), demonstrado pelo o sucesso apresentado pelos os serviços Netflix e Youtube.

O serviço OTT engloba a entrega de video e áudio através da Internet sem um controlo direto dos operadores de telecomunicações, apresentando uma proposta atractiva de baixo custo e lucrativa.

Embora a entrega OTT seja cativante, esta padece de algumas limitações. Para que a proposta se mantenha em crescimento e com elevados padrões de Qualidade-de-Experiência (QoE) para os consumidores, é necessário investir na arquitetura da rede de distribuição de conteúdos, para que esta seja capaz de se adaptar aos diversos tipos de conteúdo e obter um modelo otimizado com um uso cauteloso dos recursos, tendo como objectivo fornecer serviços OTT com uma boa qualidade para o utilizador, de uma forma eficiente e escalável indo de encontro aos requisitos impostos pelas redes móveis atuais e futuras.

Esta dissertação foca-se na distribuição de conteúdos em redes sem fios, através de um modelo de cache distribuída entre os diferentes pontos de acesso, aumentando assim o tamanho da cache e diminuindo o tráfego necessário para os servidores ou caches da camada de agregação acima. Assim, permite-se uma maior escalabilidade e aumento da largura de banda disponível para os servidores de camada de agregação acima. Testou-se o modelo de cache distribuída em três cenários: o consumidor está em casa em que se considera que tem um acesso fixo, o consumidor tem um comportamento móvel entre vários pontos de acesso na rua, e o consumidor está dentro de um comboio em alta velocidade.

Testaram-se várias soluções como Redis2, Cachelot e Memcached para servir de cache, bem como se avaliaram vários proxies para ir de encontro ás características necessárias. Mais ainda, na distribuição de conteúdos testaram-se dois algoritmos, nomeadamente o *Consistent* e o *Rendezvouz Hashing*.

Ainda nesta dissertação utilizou-se uma proposta já existente baseada na previsão de conteúdos (*prefetching*), que consiste em colocar o conteúdo nas caches antes de este ser requerido pelos consumidores.

No final, verificou-se que o modelo distribuído com a integração com prefecthing melhorou a qualidade de experiência dos consumidores, bem como reduziu a carga nos servidores de camada de agregação acima.

Abstract

Nowadays, the Internet is considered an essential good, due to the fact that there is a need to communicate, but also to access and share information. With the increasing use of the Internet, allied with the increased bandwidth provided by telecommunication operators, it has created conditions for the increase of Over-the-Top (OTT) Multimedia Services, demonstrated by the huge success of Netflix and Youtube.

The OTT service encompasses the delivery of video and audio through the Internet without direct control of telecommunication operators, presenting an attractive low-cost and profitable proposal.

Although the OTT delivery is captivating, it has some limitations. In order to increase the number of clients and keep the high Quality of Experience (QoE) standards, an enhanced architecture for content distribution network is needed. Thus, the enhanced architecture needs to provide a good quality for the user, in an efficient and scalable way, supporting the requirements imposed by future mobile networks.

This dissertation aims to approach the content distribution in wireless networks, through a distributed cache model among the several access points, thus increasing the cache size and decreasing the load on the upstream servers. The proposed architecture was tested in three different scenarios: the consumer is at home and it is considered that it has a fixed access, the consumer is mobile between several access points in the street, the consumer is in a high speed train.

Several solutions were evaluated, such as *Redis2*, *Cachelot* and *Memcached* to serve as caches, along with the evaluation of several proxies server in order to fulfill the required features. Also, it was tested two distributed algorithms, namely the *Consistent* and *Rendezvous Hashing*.

Moreover, in this dissertation it was integrated a *prefetching* mechanism, which consists of inserting the content in caches before being requested by the consumers.

At the end, it was verified that the distributed model with *prefetching* improved the consumers QoE as well as it reduced the load on the upstream servers.

Contents

List of Figures

List of Tables

Acronyms

AP	Access Point
API	Application Programming Interface
CDN	Content Delivery Network
CPU	Central Processing Unit
DSMC	Distributed Smart Management Cache
FIFO	First In, First Out
FPS	Frames per Second
GB	GigaByte
GHz	GigaHertz
GPS	Global Positioning System
HDS	Adobe HTTP Dynamic Streaming
HLS	Apple HTTP Live Streaming
HTTP	Hypertext Transfer Protocol
ID	IDentifier
IIS	Internet Information Services
IP	Internet Protocol
IPTV	Internet Protocol Television
ISP	Internet Service Provider
IT	Instituto de Telecomunicações
ITU-T	International Telegraph Union Telecommunication Standardization Sector
LRU	Least Recently Used
MB	MegaBytes
Mbps	Megabits Per Second

MD5 Message-Digest algorithm 5

MOS Mean Opinion Score

MPEG-DASH Moving Pictures Expert Group - Dynamic Adaptive Streaming over HTTP

MPU Most Popularly Used

NAP Networks Architectures and Protocols

OTT Over-the-Top

PC Personal Computer

P2P Peer-to-Peer

QoE Quality of Experience

QoS Quality of Service

RAM Random Access Memory

RTCP Real-Time Transport Control Protocol

RTP Real-Time Transport Protocol

RTSP Real Time Streaming Protocol

SHA-1 Secure Hash Algorithm 1

SHA-256 Secure Hash Algorithm 256

SHA-512 Secure Hash Algorithm 512

TCP Transmission Control Protocol

TTL Time to Live

TV Television

UDP User Datagram Protocol

URL Uniform Resource Locator

VoD Video on Demand

XML Extensible Markup Language

Chapter 1

Introduction

This chapter presents the motivations, objectives and a brief description of this document.

1.1 Motivation

In recent years, due to the increasing bandwidth and reduction of access times provided by telecommunication operators, consumers habits regarding content consumption have been changing [8][9], which can be demonstrated by the clear increase in the trend of non-linear Television (TV) video watching versus broadcast TV services [10]. Also, with the huge growth of mobile market [8], it is possible to watch the content anywhere, anytime and on any device. However, due to the fact that the OTT content delivery proposal is not directly controlled by the telecommunication operators, Quality of Service (QoS) is not guaranteed since all the content, including video, that traverses through the network receives the same treatment, therefore compromising final users' QoE[11][12].

Among other parameters, the QoE metric is mainly influenced by the video resolution that is displayed to the client versus the resolution that the client's device can support, as well as if the video freezes or not. The same happens in Live scenarios, where video is transmitted over the network and has to have short buffer size, which means that it has some probability to freeze. However, it is possible to increase buffer size, but in this case, the live video will present an increasing additional delay. In order to maintain high QoE standards, it is necessary to invest in the multimedia delivery infrastructure optimizing delays and avoiding video effects that result on final users experience frustrations.

Moreover, with a trend to change to a higher resolution, the Content Delivery Network (CDN)s will have an additional effort, since a higher resolution requires a higher bitrate, e.g., 4k, generating a greater utilization of bandwidth and computational resources. Therefore, it is necessary to improve the content delivery.

In this sense, it is necessary to store video content close to consumers, avoiding overload the network on the video origin side. One way to avoid the overhead of video sources is to have a cache model that is shared among the several edge caches available on the network. Thus, it increases the cache size, since there is no content redundancy between them. Also, with the increase in available cache size, it is possible to store more content, emphasizing the fact that video with higher qualities represent lower duration of video that can be stored for the same cache size, since content with high qualities needs higher storage size. Another way to improve QoE is through the use of the prefetching[12] mechanism that inserts the content

in the caches before being requested by the consumers. By using these two mechanisms, it is possible to improve the consumers QoE.

Moreover, in the mobile market there is a strong interference and a significant packet loss due to the wireless networks which cause an impact on QoE. Moreover, in this market, due to the harsh network conditions, consumers will request several different video qualities at the same time generating high impact on the edge-cache performances and additional efforts on the origin server(s).

Summing up, the edge caches must work together, thus forming a network of content distribution promoting the reduction of the load in the origins, even though the content is closer to the users, allowing a better QoE to the clients.

1.2 Objectives and Contributions

The objective of this dissertation is to propose a distributed cache architecture for content distribution, based on distributed caches on the network that work in a distributed manner. This architecture is an edge-based cache approach to deal with the OTT content distribution challenges on mobile scenarios.

Therefore, this work specific objectives are as follows:

- To evaluate different technologies: a study of requirements and technologies assessment for In-Memory cache solutions, distributed hashing strategies and proxy cache solution for distributed caching.
- To propose, implement and test the proposed distributed caching architecture: a distributed cache prototype is implemented in order to cache efficiently the content, and reducing the load on the upstream servers. Beyond the implementation, the architecture assessment is performed on three main scenarios (fixed, mobile and highly mobile).
- To integrate the proposed cache architecture with the prefetching mechanism: it allows to improve the client quality of experience, since the content is inserted in the caches before the clients request it.
- To improve consumers' QoE: due to the fact that edge-caches share the available local cache storage among them, it reduces the probability of fetching video content from the video origin.
- To build a demonstrator: It allows to test the different models for the different scenarios.
- To measure consumer QoE in a real client: it allows to have reliable results considering the adaptation algorithm of the player used.

An integrated distributed cache demonstrator has been deployed in Altice Labs and Networks Architectures and Protocols (NAP)/Instituto de Telecomunicações (IT).

An internal research project report for the P2020 UltraTV project (Altice Labs and NAP/IT) on Content distribution techniques specification has been delivered.

The work proposed, implemented and assessed using the DSMC distributed caching architecture resulted in a published paper at INFORUM Symposium on Computer Networks,

October 12 2017, Aveiro, as well as in a submitted paper to IEEE/IFIP WONS 2018 14th Wireless On-demand Network systems and Services Conference 6-8 February 2018, Isola 2000, France (under review).

1.3 Document Organization

This document is structured as follows:

- Chapter 1: it presents the motivation, objectives and contribution of the work.
- Chapter 2: it presents the state of the art about the OTT Multimedia Networks, Multimedia Streaming Technologies, Prefetching, QoE and Caching Algorithms. It also presents the technologies assessment.
- Chapter 3: it presents the proposed architecture, modules and the integration with the overall architecture.
- Chapter 4: it presents the implementation of the proposed architecture in a practical point of view.
- Chapter 5: it presents the evaluation of the implemented solution, in different scenarios.
- Chapter 6: it presents the dissertation's conclusions and the future work.

Chapter 2

State of the art

2.1 Introduction

To understand the fundamental concepts of the OTT delivery networks and to support the developed work, this chapter introduces and provides a deep insight on the several areas of the topics being studied.

The organization of this chapter is described as follows:

- Section 2.2: introduces the concepts of the OTT Multimedia Networks and of the video content services.
- Section 2.3: describes the evolution of the several streaming technologies, giving special attention to the streaming technology used in this work, the adaptive segmented HTTP-based delivery.
- Section 2.4: shows the concept of prefetching mechanism and the importance for OTT Multimedia Networks.
- Section 2.5: presents the concept of QoE and its relevance in OTT Multimedia Networks.
- Section 2.6: gives an overview of some popular caching algorithms, presenting also caching algorithms for video content.
- Section 2.7: describes the several content distribution systems.
- Section 2.8: presents distributed hashing algorithms.
- Section 2.9: describes caches solutions.
- Section 2.10: presents initial assessments for strategies, algorithms and technologies that will be used in this work.
- Section 2.11: presents the chapter considerations.

2.2 Over-The-Top (OTT) Multimedia Networks

In recent years, the OTT multimedia networks have grown, since the telecommunication operators were able to provide an increased bandwidth and the reduction of access times, allowing a suitable delivery of the content[11][12].

The networks can be categorized in closed or open [13], depending if the network is controlled or not by the telecommunication operator.[11][12].

In a closed network, the Internet Service Provider (ISP) controls the content delivery and it ensures a high QoS to the clients. This service is provided in Internet Protocol Television (IPTV) services, which is typically paid[11][12].

In an open network, the ISP does not controls the content delivery, then it does not ensure the QoS in the content delivery. The delivery of the content uses the ISP network infrastructure for free then all the content that traverses the network has the same treatment, which means that it is an unmanaged network[11][12].

The OTT multimedia networks is characterized as an unmanaged network, since there is not an entity to control the network. Some examples of the OTT service providers are Netflix [14] and Youtube [15][11][12].

Due to the fact that there is no network control the QoE is compromised [16]. To have a good QoE it is necessary to offer a video transmission with a low-buffering time, a resolution adequate to the devices screen, and the video must not freeze during the playback time[11][12]. To have high QoE standards, it is necessary to understand multimedia delivery infrastructure to provide a good experience to the consumers.

OTT Multimedia Services in Telecommunication Operators

At the beginning, the telecommunication operators have been delivering the TV content in managed networks. But, the recent consumption trend demonstrates a clear increasing trend of non-linear TV video watching versus broadcast TV services [10][11][12].

In addition, there is a huge success of *pure-OTT* content distribution, demonstrated by YouTube and Netflix, which are competing with these telecommunication operators providers. Thus, it has allowed changing the consumers habits [8] [9]. To win the struggle of acquiring new clients, the telecommunication operators are moving to the *OTT-based* content distribution services. Then the operators want to provide multi-screen services to it's consumers, so they can have the choice to access the content anywhere and anytime rather than being restricted to a single device, for instance the TV[11][12].

Thereafter, it is necessary to understand the role of the streaming protocols, mainly the Adaptive Segmented HTTP-based delivery, and the content distribution systems, focusing in distributed cache, that are described in the following sections.

The telecommunication operators can provide several services to deliver the video content, such as, **Linear TV**, **Time-shift TV** and **Video on Demand (VoD)** [17].

Linear TV is the regular TV broadcast that delivers a predetermined program lineup to the users. This service was considered for decades the traditional way to watch TV content [17].

In the case of **Time-shift TV**, the users are allowed to watch the linear-TV content later, as it can be recorded, using some services like *PauseTV*, *Start over TV*, *Personal Video Recorder* and *Catch up TV* [17].

Regarding the *Pause TV*, the users are allowed to pause the television program they are currently watching, and then they can resume the TV broadcast when they want, continuing where they left off or skip a particular segment or even skip to linear TV broadcast [17].

In *Start over TV*, users are able to restart ongoing programs or programs that already finished (typically it is possible to rewind some minutes up to 24 hours, it depends on the content provider operator)[17].

In *Personal Video Recorder*, users can schedule a recording of a TV show. Then, they can watch a recording whenever they want [17].

In *Catch up TV*, users are able to watch TV programs that have finished in the previous hours up to 30 days, depending on the content provider operator. The content providers are the ones that record the content[17].

In case of the **VoD**, the users can watch the content they pay for, in many ways, such as, *Transaction VoD*, *Electronic Sell Trough VoD* and *Subscription VoD* [17].

Regarding *Transaction VoD*, the users rent what they want to watch. They can watch the content purchased several times during the rental time, which is usually from 24h or 48h, depending on the content provider operator [17].

In *Electronic Sell Trough VoD*, the user pays a fee enabling him to access the purchased content in a specific platform [17].

In *Subscription VoD*, the users pay a monthly fee enabling them to watch anything they want from a catalog. This type of VoD service is the one adopted by the OTT content providers such as Netflix [17].

2.3 Multimedia Streaming Technologies

Streaming is defined by the process of transmitting data from a transmitter to a receiver capable of consuming the content. The difference between streaming and non-streaming is that, in the case of streaming, the receiver can reproduce the contents without having received all the data file; in the case of non-streaming, the receiver must receive all the data before being reproduced [18].

Initially, streaming gained popularity with the appearance of the Real Time Streaming Protocol (RTSP) [19]. Due to the fact that there was an increased bandwidth provided by telecommunication operators, video streaming gained easier accessibility, which it was demonstrated by the huge success of the IPTV and OTT services platforms[11][12].

Nowadays video streams use different types of protocols, which differ on their implementation details. However, these streaming protocols can be classified into two main groups[11][12]:

1. *Push-Based*: the server is responsible to stream the video data to the client. Thus, the server is responsible to control the connection, having to be attentive to the changes of the state of the session [20].

2. *Pull-based*: the client is responsible to request the video content from the server [20]. A common protocol for this category is the HTTP.

Traditional Streaming

The traditional streaming was the first to gain popularity in multimedia streaming. An example of a traditional streaming protocol is RTSP. This protocol is used to establish and control multimedia streaming. The delivery is done from the server to the client using a Real-Time Transport Protocol (RTP) channel that it can use Transmission Control Protocol (TCP)

or User Datagram Protocol (UDP) [21]. In order to have a stable session, the RTSP uses Real-Time Transport Control Protocol (RTCP) packets to collect information about the network conditions [21][11][12].

Figure 2.1 depicts an example of RTSP streaming.

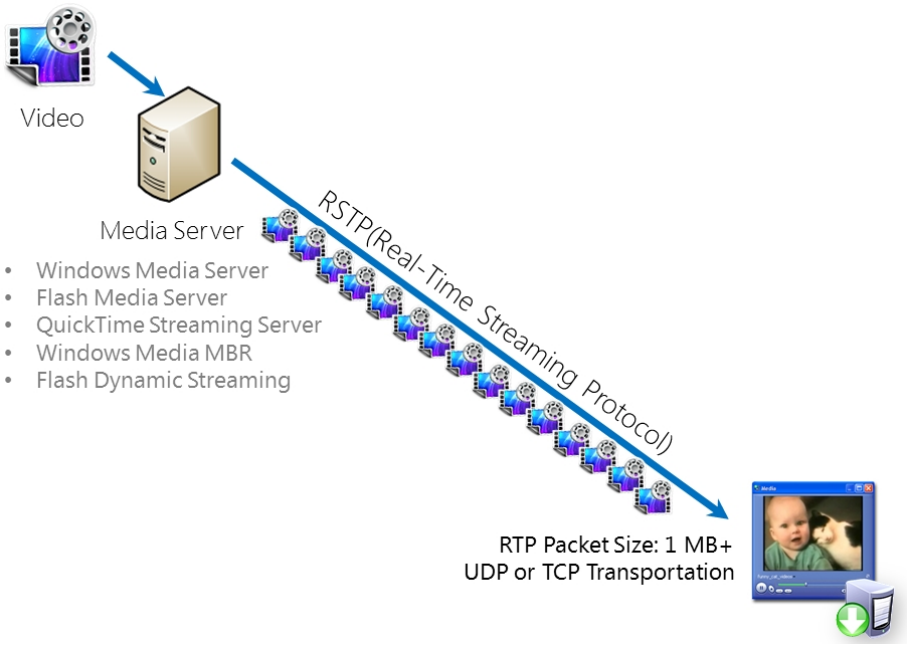


Figure 2.1: Traditional Streaming[1]

However RTSP has several limitations, which includes the scalability and complexity, since to support several clients the server has to manage several sessions [19][11][12].

Due to the presented limitations, the RTSP has become obsolete, but it can still be used in video-conferencing[11][12].

Progressive Download

Progressive Download is a pull-based approach, where the clients download the content via HTTP, as depicted in Figure 2.2. The term progressive is because an user can start to watch the content, when the player receives only part of video content. Typically the players have a buffer capable of holding a few seconds of content; when this buffer is fulfilled, the video starts to play, as depicted the Figure 2.3.

Due to the fact that in this approach the server do not need to reserve resources for each client, it can support millions of users by using proxies servers, caches and (CDNs) to optimize the scalability and reduce the load on the origin server side[11][12].

However, there are some limitations and disadvantages which includes no support for live streaming, no adjustable streaming based on network limitations, causing the stop of the playback and resulting on a waiting period for a buffering[11][12].

Adaptive Streaming Technologies

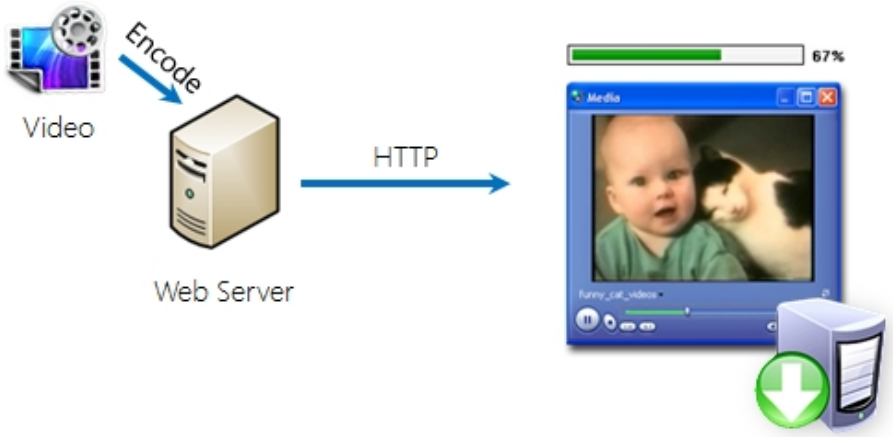


Figure 2.2: Progressive Download Architecture [2]

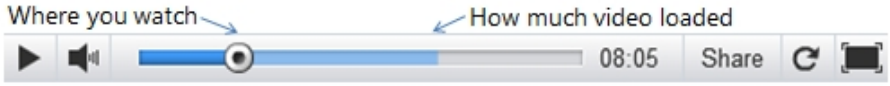


Figure 2.3: Progressive Download player[3]

The Adaptive Streaming is an enhanced version of the progressive download and traditional streaming, since it takes the advantages of both technologies. The advantages consist in the support of some streaming adaptation based on network limitations, similarly to the RTSP protocol, and it is high scalability, similarly to the progressive download[11][12].

This approach is the one considered in this work and presents features that are essential since, in scenarios where the network conditions have some kind of degradation, an adaptation is necessary in order to provide a smooth visualization of the video. This multimedia streaming technology relies on the adaptation of the bandwidth required by the delivery of video content. It consists in varying the quality of the consumed video, which varies the bit rate required to delivery the video [22][11][12].

Typically, the most used adaptive streaming technology is HTTP-based delivery, which will be presented below.

Adaptive Segmented HTTP-based delivery

The adaptive Segmented HTTP-based delivery consists in encoding the original video into different qualities and break them into short chunks that have a duration of 2 to 10 seconds.

The client device is the active entity that is in charge to do the adaptation of the quality to be downloaded during the execution of the video. This adaption is based in various parameters, namely the network quality, computational resources and battery status related with the client device[11][12].

In order to estimate the network conditions, the adaptive streaming systems rely on the previous downloaded chunks to estimate the network conditions, as well as the buffer status [22].

Since the adaptation results in varying the quality of the video, the user device can

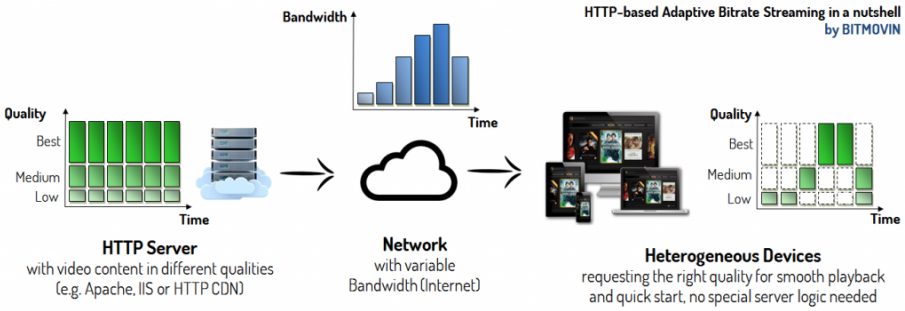


Figure 2.4: Segmented HTTP Adaptive Streaming[4]

consume different chunks with different video resolutions.

There are multiple implementations of adaptive segmented HTTP-based delivery, such as Apple HTTP Live Streaming (HLS) [23], Adobe HTTP Dynamic Streaming (HDS) [24], Moving Pictures Expert Group - Dynamic Adaptive Streaming over HTTP (MPEG-DASH) [25] and Microsoft Smooth Streaming [26].

This work used Microsoft Smooth Streaming implementation, since some of the previous works performed in our research lab rely on this implementation. The proposed distributed caching architectural model will be integrated with those works.

Microsoft Smooth Streaming

Microsoft Smooth Streaming is an adaptive segmented HTTP-based delivery implementation that was developed by the Microsoft.

This implementation is composed by three main components, such as, the encoder, Smooth Streaming-enabled Internet Information Services and the Smooth Streaming Client. Figure 2.5 depicts the three main components.

Figure 2.5: Smooth Streaming Architecture [5]

The encoder is the one that is in charge to encode the video content in several qualities, generating two different manifests in Extensible Markup Language (XML) format, as explained below. The Smooth Streaming-enabled Internet Information Services is the origin server that provides the streaming services to the clients. The Smooth Streaming Client is

responsible for requesting the content to an origin server, and for the displaying of the video content to the user. Microsoft provides a Smooth Streaming Client that uses the Silverlight plug-in[27].

The manifests generated by the encoder are the client manifest and the server manifest. The server manifest file is used by the Smooth Streaming-enabled Internet Information Services, the origin server, that provides some characteristics about the content, including the number of encoded content, its type that can be text, audio or video and the information about the content, for instance, the bit rate, resolution, codec type and its location[11][12].

On the other hand, the client manifest provides the information of the content, the number of available resolutions, duration and how they are fragmented, which means, the number and the duration of the chunks[11][12].

Figure 2.6 depicts an example of a client manifest file.

```
<?xml version="1.0" encoding="utf-16"?>
<!--Created with Expression Encoder version 4.0.3158.0-->
<SmoothStreamingMedia MajorVersion="2" MinorVersion="1" Duration="1303410000">

  <StreamIndex Type="video" Name="video" Chunks="65" QualityLevels="8" MaxWidth="1280" MaxHeight="720"
DisplayWidth="1280" DisplayHeight="720" Url="QualityLevels({bitrate})/Fragments(video={start time})">

  <QualityLevel Index="0" Bitrate="2962000" FourCC="WVC1" MaxWidth="1280" MaxHeight="720"
CodecPrivateData="250000010FD37E27F1678A27F859F180490825A645A6440000010E5A67F840" />

  <QualityLevel Index="1" Bitrate="2056000" FourCC="WVC1" MaxWidth="992" MaxHeight="560"
CodecPrivateData="250000010FD37E1EF1178A1EF845FF8A8B8049081BEBE7D7CC0000010E5A67F840" />

  <QualityLevel Index="2" Bitrate="1427000" FourCC="WVC1" MaxWidth="768" MaxHeight="432"
CodecPrivateData="250000010FCB6C17F0D78A17F835F18049081AB8BD71840000010E5A67F840" />
   <.../>

   <c d="20000000" />

 </StreamIndex>

 <StreamIndex Type="audio" Index="0" Name="audio" Chunks="65" QualityLevels="1"
Url="QualityLevels({bitrate})/Fragments(audio={start time})">

  <QualityLevel FourCC="WMAP" Bitrate="128000" SamplingRate="44100" Channels="2"
BitsPerSample="16" PacketSize="5945" AudioTag="354"
 CodecPrivateData="1000030000000000000000000000E0000000" />

   <c d="20897959" />

 </StreamIndex>
</SmoothStreamingMedia>
```

Figure 2.6: Smooth Streaming client Manifest

Initially, the client requests to the origin server a manifest file of a particular content, and then the origin responds to the client with all the essential information to start requesting the video chunks to play the video. Moreover, the chunks can be stored in several caches, in order to reduce the load on origin and reduce the clients perceived latency. Also, the chunks can be pre-fetched, in order to reduce the latency, since the content is inserted in cache before the clients request it.

2.4 Prefetching

Prefetching is a mechanism that allows to pre-load data, storing it into memory, before being requested by a user. Prefetching pre-populates a memory system, allowing to reduce a user perceived latency [28][29]. The prefetching is a good method to increase the cache performance, hit ratios, since it loads the data before it is request by a consumer [28]. However, in order to introduce small overhead in the network and to not waste bandwidth, it is necessary to select objects to prefetch [30]. There are several prefetching algorithms that prefetch the content by popularity [31] (prefetch only the most popular objects), lifetime [30] (prefetch the objects with the longest lifetime, which means, the objects that are less frequently updated) and good fetch [32] (prefetch the objects that have a probability of being accessed before being updated passes a particular limit)[12].

The work in [33] proposes a prefetching mechanism for adaptive segmented HTTP-based delivery, Microsoft Smooth Streaming. This prefetching approach is the one integrated with distributed caching architecture proposed in this work. This prefetching mechanism predicts the future chunks of a given video in a certain quality and pre-populate the cache. Two prefetching algorithms were assessed: the first algorithm takes into account the quality of the previous chunk to request the next chunk, and the second algorithm takes into account the quality of the four previous chunks to request the next chunk, in a weighted mean. The results show that both algorithms reduce the clients requests access time and increase the cache performance, but the second algorithm presents a better performance compared to the first algorithm, also with a better perceived QoE.

The work proposed in [34] uses a prefetching mechanism that overcomes periods with low bandwidth or lack of connectivity within train tunnels. The prefetching mechanism previously downloads the requests in the good signal zones, and than in the future the clients will be able to continue watching videos in the tunnel or in bad connectivity zones. It is noticed that, without using the prefetching buffer, underruns occur during the bad reception while travelling within the tunnel. Finally, this approach overcomes all periods of bad quality or lacking reception by pre-buffering data.

2.5 QoE

QoE is a measure of the system quality provided by the clients satisfaction with a particular service. So, it takes into consideration the user's perspective on a particular service. QoE is highly dependent on the conditions and limitations imposed by the entire system. It is an important metric to take into consideration since it presents an evaluation of a particular system. Therefore, there is a relevant research in QoE topic [35]. Moreover, QoE is characterized by the International Telegraph Union Telecommunication Standardization Sector (ITU-T) [36] as: *"The overall acceptability of an application or service, as perceived subjectively by the end-user."*

In order to quantify the QoE it is used the MOS [7], which is comprised in ranges from 1 to 5, poor to excellent, respectively. The table 2.1 shows the MOS levels.

QoE in Adaptive Streaming solutions

The adaptive streaming technologies are capable to adapt to the network limitations of the system, reducing the playback interruptions (re-buffering), and therefore improving the clients perceived QoE. A QoE probe [37] is proposed and implemented to determine the

Table 2.1: Mean Opinion Score - MOS[7]

MOS	Quality
5	Excellent
4	Good
3	Fair
2	Poor
1	Bad

QoE estimation in the Adaptive Streaming. This QoE probe is the one used in this work to determine the QoE estimation in a HTTP adaptive video streaming with several metrics extracted throughout the video playback session, such as bit-rate, frames per second of a specific chunk, re-buffering and screen resolution ratio. At the end, the probe outputs the MOS for each consumed chunks. Considering these metrics, depending on the conditions and limitations imposed by the network and the users device, the perceived QoE is highly influenced.

Also, another factor that has influence in QoE is the caching eviction policy used in the proxy cache servers. Because, a poor caching eviction policy results in less hit-ratios, worse cache performance, and then the content might need to be requested to the origin, increasing the backend traffic and delays.

2.6 Cache Replacement Algorithms

A cache algorithm is a method that allows the decision to evict elements when a cache is full, in order to make room for new elements. There are several caching algorithms in the literature, and 3 of them will be explained in the following (FIFO, LRU, MPU). The FIFO and LRU algorithms are widely used in traditional caches; the MPU is an algorithm proposed in [38], which is proposed for Catch-up TV video services.

To exemplify the eviction procedure of these algorithms, it is considered the following string [a, b, c, d, a, b, d, c, d, a, b, d, c], where each letter corresponds to an item and the string represents the order of each request.

FIFO

Regarding the FIFO eviction algorithm, it keeps a list based on the arrival timestamp of each item, which means that it has a list from the oldest item to the newest item. Then, it always removes the oldest item. Table 2.2 depicts an example with the reference string, where it is observed that using the reference string this algorithm presents 2 hits.

Table 2.2: FIFO Cache Replacement Policy

String Sequence	a	b	c	d	a	b	d	c	d	a	b	d	c
block 1	a	a	a	d	d	d	d	c	c	c	b	b	b
block 2		b	b	b	a	a	a	a	d	d	d	d	c
block 3			c	c	c	b	b	b	b	a	a	a	a
Result	miss	miss	miss	miss	miss	miss	**hit**	miss	miss	miss	miss	**hit**	miss

LRU

The LRU eviction policy maintains a list of the items recently used and removes the least recently requested elements. Table 2.3 depicts an example with the reference string, where it is observed that, using the reference string, this algorithm presents 3 hits.

Table 2.3: LRU Cache Replacement Policy

String Sequence	a	b	c	d	a	b	d	c	d	a	b	d	c
block 1	a	a	a	d	d	d	d	d	d	d	d	d	d
block 2		b	b	b	a	a	a	c	c	c	b	b	b
block 3			c	c	c	b	b	b	b	a	a	a	c
Result	miss	miss	miss	miss	miss	miss	**hit**	miss	**hit**	miss	miss	**hit**	miss

MPU

The MPU cache eviction policy takes into consideration the content demand to do the decisions to evict the items. Through the content demand it is generated the *"priority maps"* that identify which items to keep in cache of the Catch-up TV content.

Thus, the evictions are based on the priority of each item, which means the items with higher priority tend to keep in cache, while the items with lower priorities tend to be more evicted. This algorithm presents a higher number of cache hits, providing a better cache performance for Catch-up TV services.

Table 2.4 depicts an example using the reference string with a priority map defined by {[a:1],[b:2],[c:5],[d:5]}, where the letter represents the items and the numbers represents their priority: the highest priority is the highest number. It is observed that this algorithm presents 5 hits.

Table 2.4: MPU Cache Replacement Policy

String Sequence	a	b	c	d	a	b	d	c	d	a	b	d	c
block 1	a	a	a	d	d	d	d	d	d	d	d	d	d
block 2		b	b	b	a	b	b	b	b	a	b	b	b
block 3			c	c	c	c	c	c	c	c	c	c	c
Result	miss	miss	miss	miss	miss	miss	**hit**	**hit**	**hit**	miss	miss	**hit**	**hit**

A cache performance depends on the cache algorithm and mainly when it is requested. Also, to present a better cache performance, the cache algorithms needs to know which items have to keep in the cache[11][12].

2.7 Content Distribution

Content distribution is a subject that has been studied by both the academic and the business side. There are several solutions in this area that have been recurrently explored, to offer customers the best quality without compromising the quality of user experience. The most usual approaches are based on proxy caching [39][40], Peer-to-Peer (P2P) [41][42][43] and Hybrid Delivery [44][45][46][47][48].

A proxy caching approach inserts caches that serve as intermediates between the users and the origin server. Then, the users make the requests to the caches, and if the caches do not have the content, they will request in the origin. Otherwise, if the caches have the content, they return the content to the user, not needing to fetch from the origin. This solution has

the disadvantage that in case a miss cache occurs, it represents an additional delay. However, it presents fast responses to the clients requests.

In the P2P approach the content is spread by the users, in which the customers also provide content, communicating with each other; this approach borrows the uplink capacity of the clients. It reduces the cost of maintaining the Proxy servers, since mostly the content is provided by the clients. An example of P2P video streaming is the Popcorn Time platform [41]. However, this approach has some limitations, like the lag occurred when starting a new streaming session[49] because locating and acquiring content from clients takes longer than acquire from a proxy server[11][12]. Also, the playback delay in live streaming is more severe since the P2P systems boasts a higher number of delays[50]. Furthermore, keeping the P2P infrastructure is complex which can interfere with the degradation of the QoE [51]. Also, the P2P systems in Adaptive Segmented HTTP-based delivery are very complex to implement, since the adaptation of the qualities is performed according to the variation of the network conditions, since the quality of the video also depends on the quality of the network. If a device has a poor quality consumed, due to the network limitation, then the network may not support the delivery of content to other customers. In case a client has good network conditions, it has higher qualities consumed; however, it cannot provide it to devices with lower quality since the content is different.

Hybrid Delivery uses the P2P system with the proxy caching system. This approach has the advantages of the proxy caches, which can reduce the delay of the P2P systems. Also, it can efficiently save bandwidth and load on proxy caches. For a live streaming, the low latency is mandatory which should not be used in the P2P systems [50]; however, it can be used for VoD services [46][47][11]. The work in Sanguankotchakorn and Krueakampliw proposed a Hybrid Delivery to reduce the delay in the P2P systems. It provides superior performance than P2P systems, in terms of delivery delay and start up delay. The use of this type of Hybrid Solutions in Adaptive Segmented HTTP-based delivery systems makes a tradeoff in the clients QoE, since it provides a large end-to-end delay.

Summing up, the P2P presents some limitations due to its additional latency and high complexity. Also, with the imposed limitations in mobile devices, where the link conditions are always varying, inducing additional latency, which is not a good approach. Finally, overloading peers interferes with the available bandwidth, which can compromise the QoE of the entire population.

Shared web caching:

Fan et al. it is presented a strategy to perform a shared web caching using Bloom filters, in which the proxies share the content among them[52]. So, when a request arrives, the proxy verifies if the content is in its local cache; if not, it checks if another proxy cache has the requested content, using the Bloom filter of that respective proxy[52]. If it has the content, the request is done for that respective proxy to obtain the content[52]. The Bloom filters allows to the proxies to know if the content is in the neighboring caches, but the proxies need to send periodically the Bloom Filters between them. These Bloom Filters reduce the overhead in the network, since the proxies caches do not send a list of keys of their caches, since the proxies send the Bloom filter, which is a compact way to represent the contents of their caches[52].

However, the Bloom filters present false positives and if the Bloom filters are not updated, the whole system is not synchronized. Thereafter, in video live scenarios the caches will not share the caches among them, since the requests for live scenarios are done at the same time, and none of the caches have the updated Bloom filter neither the content, so the requests will

be served all by the origin.

In Karger et al., it is presented an algorithm, Consistent Hashing[53], that allows to distribute the content among the several available caches. This algorithm allows to the caches behave together in one coherent system. However, it presents some limitations, such as, unbalanced distribution but it can be minimized, as described in section 2.8.1. This algorithm allows a proxy to know which cache is responsible for a particular content. It has to hash the request identifier and the algorithm provides the cache that is responsible for a particular content. Also, in case of a cache failure, only the content of that particular cache is remapped, maintaining the consistency for the other caches. In Thaler and V.Ravishankar, it is presented the rendezvous algorithm[54] that has the same goals as the consistent hashing method. However, for each request the request identifier needs to be hashed with all available caches, which results to an additional delay. These two algorithms will be better presented in the ensuing subsection, and they were tested and compared in section 2.10.3.

2.8 Distributed Hashing Algorithms

This section presents a theoretical analysis of the two hashing algorithms, Consistent Hashing ans Rendezvous Hashing.

2.8.1 Consistent Hashing

Consistent Hashing [53] is a method that allows to do the decision on the distribution of content among the caches. This method is fast and does a concise distribution of the load across multiple servers. This is due to the fact that consistent hashing uses hash functions that allow fast calculation to figure out where the content is going to be stored. Also, for this method to work correctly, it is necessary to use a hash function that avoids the storage of all the content in just a few cache servers. Another advantage of this method is that there is no content replication between the nodes/caches, which allows to take full advantage of the total size of the global cache.

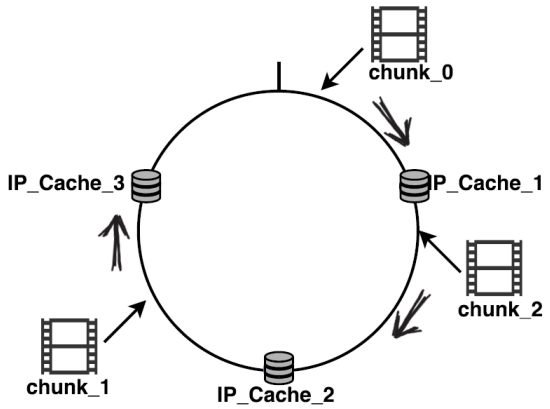


Figure 2.7: Theoretical Ring Consistent Hashing

Figure 2.7 shows the theoretical base of the consistent hashing. It is based on the use of a logical ring connecting all the cache servers where the content can be mapped in any part

of the circle, that is, in any cache server belonging to the ring.

Knowing the positions of the caches/servers in the ring, it is possible to distribute the content based on the following steps: when it is necessary to know where the chunk/content needs to be stored, it is necessary to make the hash of the chunk IDentifier (ID), which results in an integer, that is the content's cache server position in the ring. Therefore, it is necessary to move clockwise along the ring to find the server/cache where this content will be stored. If the exact position does not exist, the content is stored in the closest (in a clockwise manner) position. Taking the example of the Figure 2.7, the chunk_0 is stored in cache_1, as well as chunk_2 is stored in cache_2 and chunk_1 is stored in cache_3.

2.8.1.1 Addition and Removal of caches/nodes

Consistent hashing also allows to do the addition and removal of cache servers. When such events occur, only part of the content in the ring of cache servers need to be remapped among the cache servers. Taking as example the removal of cache 2 shown on Figure 2.8, it is possible to note that the content distribution (load) after the removal is not equal among servers. This happens because new content needs to be stored in the remaining cache servers. Moreover, when the cache server is removed from the consistent hashing ring of cache servers, content requested by users that previously was cached on cache server 2 is missed now, and it is remapped to the next cache server position in the ring in a clockwise way.

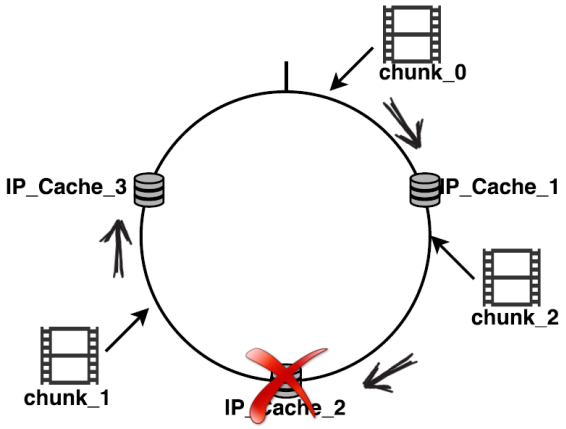


Figure 2.8: Server Removal in Consistent Caching

Thus, based on Figure 2.8, it is possible to observe that when server cache_2 is removed, chunk_2 which was stored in cache_2 server, is mapped to cache_3. However, as already mentioned, all the content is not equally distributed, due to the fact that cache_3 has a longer interval in the ring compared to the interval of cache_1, meaning that all the content hashing between cache_1 and cache_3 will be mapped to cache_3. To surpass this problem of unequal distribution, virtual nodes are inserted in the ring. Nevertheless, those virtual nodes are pointed to each real server. Thus, even if they have different positions in the ring, in reality it is only one server/cache.

Figure 3.3 shows that the insertion of virtual nodes allows the address space to be tendentiously equally distributed between the caches. This example added only two virtual nodes

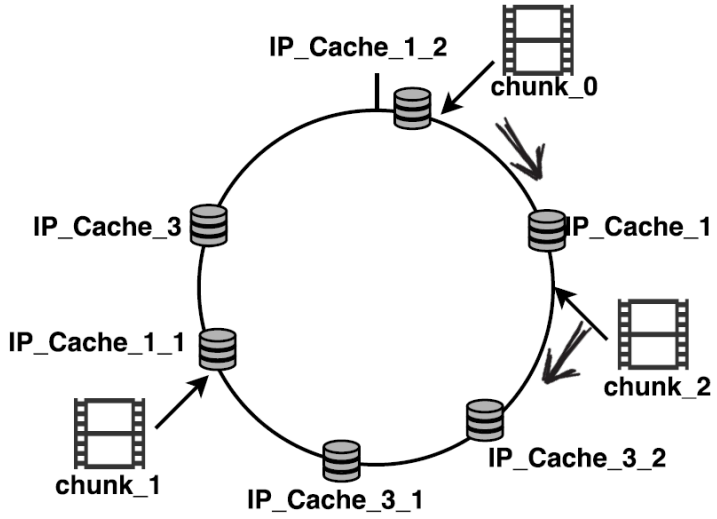


Figure 2.9: Virtual nodes insertion and content remapping

for each cache. However, it is not mandatory to be the exact number. In fact, as we increase the number of virtual nodes, a better distribution is observed.

2.8.1.2 Weighted distribution

The number of virtual nodes can be efficiently calculated by weighting cache servers. Therefore, it is allowed to have several virtual nodes comprising a particular cache server according to different weights for each cache. This is useful in the case of cache servers presenting different cache sizes. Thus, it is possible to have a weighted distribution of virtual nodes and consequently the insertion of more virtual nodes/servers comprising the servers with more weight. For example, if cache_1 has the triple size of cache_2, then cache_1 will have three nodes in the ring and cache_2 only one. Therefore, cache_1 has three times the address space of cache_2, as shown in Figure 2.10.

Based on Figure 2.10, it can be observed that the address space of cache 1 is three times larger than the address space of cache_2. So it is possible to have a weighted content distribution. This solution is similar to the solution for the problem of the inequality in the distribution of content, i.e., the introduction of virtual nodes but now in a weighted manner.

2.8.2 Rendezvous Hashing

Rendezvous Hashing [54] is a method that allows to distribute the content among caches, like consistent hashing. This method uses hash functions, where it is possible to find where the content is stored. Due to the fact that Rendezvous Hashing uses hash functions, it is a relatively fast method in deciding where to store or fetch the content. Thus, it ensures that the content is not repeated among the several caches, allowing to take advantage of the overall size of the global cache. Rendezvous Hashing, unlike the consistent hashing method, needs only to keep an updated list of the number of caches, not requiring the insertion of virtual nodes in a ring or a ring, saving memory usage. This is due to the fact this method has a different analogy to work. So, in this case, to distribute the content it is necessary to have the

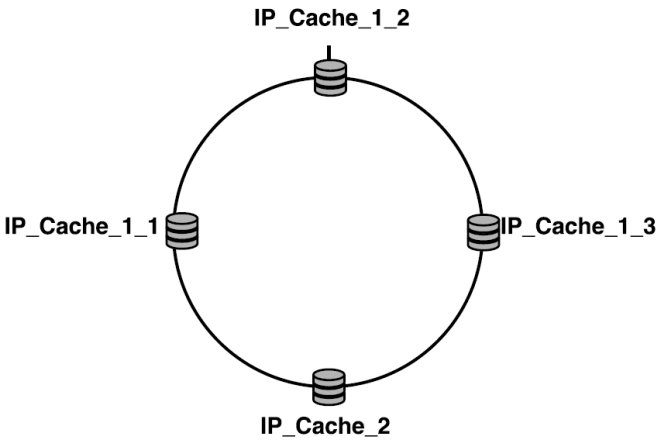


Figure 2.10: Space address for cache servers and weighted cache servers

hashing of the key for all the options (caches), always using the same hash function. In other words, for each request it is necessary to calculate a hash value for the set of cache/server with the key. That is, for each request or storage the content, it is necessary to hash the request name with all caches available, so it needs to compute n times, where n is the number of caches available, the result of the hash function of the set pair key server_name/ip. By the result of the hash function, resulting in an integer, it is possible to calculate the largest number among the various pairs of set key_server name/ip. Thus, it is possible to distribute the content among the various caches, choosing the cache that presents the pair with the highest value. However, this method is not as scalable as the consistent hashing method because, for each request, it needs to hash the set of key-servers. Thus, at increasing the number of caches the response of the algorithm becomes slower.

2.9 Approaches for Caches Implementation

There are several approaches for caches implementation, but some of the most used are Redis [55] , Memcached [56] and Cachelot [57]. They are used by companies like Facebook, Twitter, Youtube, Reddit, Orange, etc. These solutions use the concept of in-memory key-value data model so, for each key there is a single value, and the value is the content being accessed through the search of the respective key.

Figure 2.11 shows that, for each key, there is an associated value/content, and the content can be video fragments. But there are limitations in the size of the chunks to be placed in cache in order to take full advantage of cache memory. The most likely solution to surpass this limitation is to define a maximum value size to be stored in the memory cache, especially in the case of video chunks.

After having introduced the key-value data concepts, it will be explained each solution below.

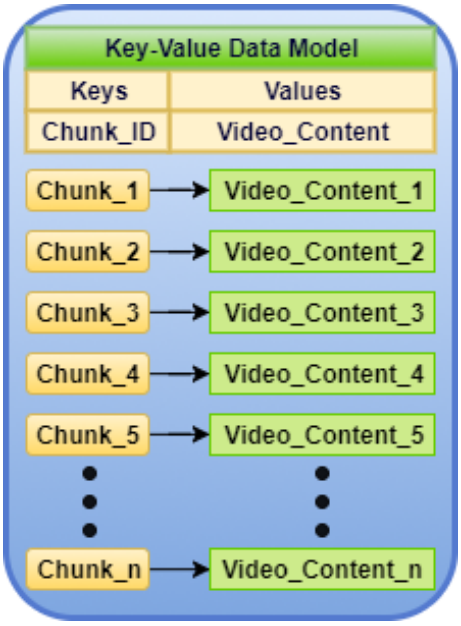


Figure 2.11: Key-Value Data Model

2.9.1 Redis

Redis [55] is an open source in-memory data storage structure that is used as a database and memory cache. It is allowed to store eight different data structures, such as string, hashes, lists, sets, sorted sets, bitmaps, hyperlogs and geospatial indexes. It is written in C, allowing to store data with optional durability. In order to achieve good performance, Redis typically keeps the whole data in memory, but it can be configured to synchronize data and write it to a file system, ensuring persistence. Redis supports replication master to slave only, therefore, a master can replicate the content to any number of slaves. A slave server is exact a copy of the master server. The replication process is non-blocking at the master side, and any slave server can be a master to another slave allowing single-rooted replication tree, as shown in the Figure 2.12.

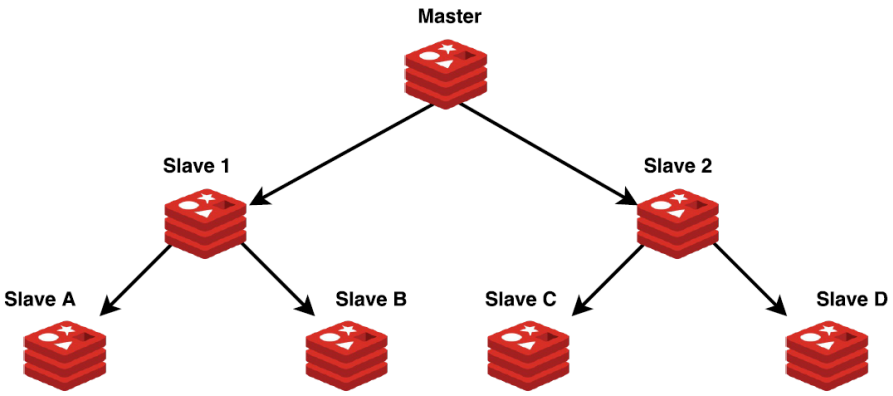


Figure 2.12: Redis Architecture example.

Redis could be used as a cache having the following eviction policies:

- allkeys-lru: evicts the less recently used keys first.
- volatile-lru: evicts the less recently used keys first, but only among keys that have an expire set.
- allkeys-random: evicts random keys.
- volatile-random: evicts random keys, but only evicts keys with an expire set.
- volatile-ttl: evicts only keys with an expire set, preferring to first evict those which have the shorter Time to Live (TTL).

2.9.2 Memcached

Memcached [56] is an open source in-memory high performance cache server. Memcached is often used to speed up websites by using the content stored in memory. At a first glance, Memcached stores simple key-value pairs, only strings and integers, and it is more efficient than Redis. Therefore, to store more complex data such as the case of arrays or objects, it needs to be serialized first, and therefore un-serialized. Similar to Redis, Memcached is written in C and the data can be stored with optional durability. In consequence of holding data in RAM memory, if the system restarts, then data is lost. Another limitation is on the key length that must be at most 250 bytes. With regard to the clients, Memcached may communicate with servers via TCP, but it does not support replication. Memcached has Application Programming Interface (API)s to provide a very large distributed hash table across multiple servers. Therefore, when the Memcached memory is full, data has to be purged through the LRU policy as well.

Figure 2.13 illustrates two deployment scenarios. In the first scenario each server is completely independent, wasting memory and resources and requiring additional effort to keep the cache consistent between nodes. The second scenario stresses the advantages to share the same pool of memory, increasing the memory cache as unique logical structure, thus both servers share the same memory allocated (128MegaBytes (MB)) across the entire system.

2.9.3 Cachelot

Cachelot [57] is an open source in-memory LRU Memcached-compatible solution; it presents a memory management more efficiently than Memcached solution, because it can store more content in the same size of memory used. The Cachelot allows to read and store information in cache with no side effects. Moreover, Cachelot API works within a fixed amount of memory, no garbage collector, and its memory utilization overhead is low (5-7%)[57].

Some Cachelot characteristics:

- Single-threaded.
- Communication with the servers via TCP, UDP, and Unix sockets.
- Does not support content replication.
- Memcache-compatible.
- Can work as consistent cache.

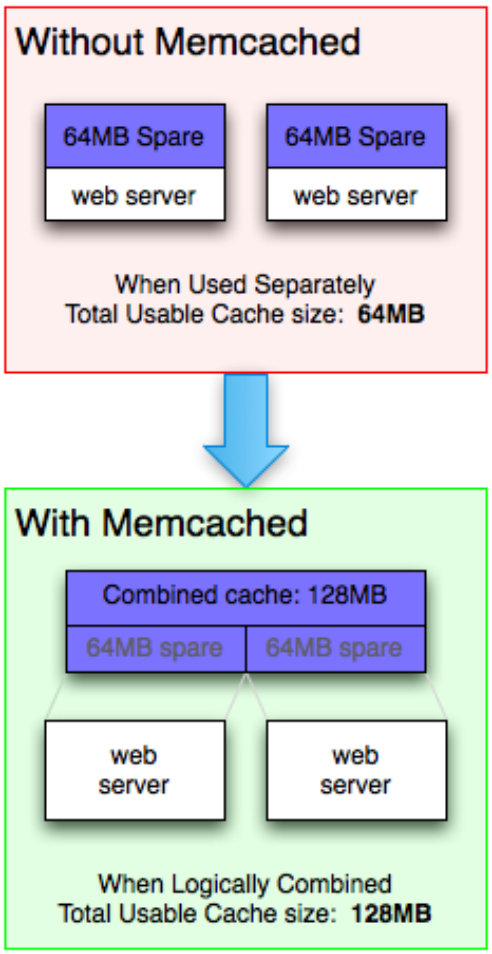


Figure 2.13: Memcached Architecture [6].

2.10 Technologies Overview and Assessments

This section performs initial assessments in order to choose which strategies, algorithms and technologies will be considered along this dissertation. This performance will be assessed through exhaustive performance tests and tables of functionalities comparison, and the results will allow to select the technologies to be integrated in the DSMC proposed architecture, which will be presented with more details in Chapter 3. In-memory cache solutions (Cachelot, Memcached and Redis2), Proxy cache solutions (Nginx, Apache ATS, Varnish and Squid) and two hashing distributed algorithms (Consistent and Rendezvous Hashing) are in-depth evaluated in this Thesis. More details of the technologies assessments will be given in the ensuing subsections.

2.10.1 Proxy Servers and Content Caching Technologies

In order for a service to support several client requests, there was the need to develop various proxy server solutions. These proxy servers reduce the load on origin server by

caching the content in its memory. Thus, whenever a content is requested, it can be delivered quickly, with low latency. Therefore, the proxy server acts as an intermediary between the clients and the origin server. The proxy server can cache various content types, for instance, video, static content and web pages.

There are several open source proxy server solutions, such as Apache [58], Nginx [59], Varnish [60] and Squid [61]. These proxy cache servers are based in a modular approach which supports multiple plugins and therefore be able to implement multiple functions allowing the proxy caches to be more robust.

Some of the features of the presented proxy servers are illustrated in Table 2.5. This comparison is necessary and based on the need to use a proxy server in the proposed architecture, having to necessarily present the characteristics described below.

Table 2.5 provides a comparison of the various proxy server solutions.

Supported Features	**ATS**	**Nginx**	**Varnish**	**Squid**
Reverse Proxy	Yes	Yes	Yes	Yes
Plugin APIs	Yes	Yes	Yes	Yes
Cache	Yes	Yes	Yes	Yes
Memcached Support	Yes	Yes	Yes	No
Transparent Cache and Memcached Support	No	Yes	No	No

Table 2.5: Comparison of Proxy Server Solutions.

Considering previous studies in our group [33] that revealed the superior performance of Nginx when compared to Squid. Nginx and Squid present the features to implement the reverse proxy and cache the content, in the integrated cache, which means that the caching of the content is done in the cache provided by the proxy, Nginx will be the one chosen in the proxy with a local cache. Due to the Nginx is the only that supports the use of external caching with the transparent caching feature, it will be used in the proposed architecture.

2.10.2 In-Memory Content Cache Solutions

To fulfill the proposed DSMC architecture requirements, it is necessary an in-memory caching solution. Three open-source solutions were evaluated: Redis2 [55] , Memcached [56] and Cachelot [57]. After individually evaluating all solutions under the same conditions, the one with the best performance will be chosen to proceed with more exhaustive tests. The tests will be performed through this script [62], which is changed to include also Redis2 solution, using as based the benchmark provided in [57]. In this test, it was chosen the solution that presents the greatest effective memory, in other words, the greatest total size of all keys and values that have remained in the cache.

Cachelot, Memcached and Redis2 use LRU as eviction policy i.e., to discard the least used items (objects) first. The performance tests consist of saturating the cache of each solution in order to confirm which of the solutions will have the smallest number of content evictions. For these assessments a Personal Computer (PC) with 512 MB Random Access Memory (RAM) memory (cache size) is considered for each solution. Regarding the size of the value to be stored, four tests are carried out: i) content of size 10 up to 1024 bytes, ii) 1024 up to 4096 bytes, iii) 4096 bytes up to 1MB and iv) 10 bytes up to 1MB, each one executed 10 times. The communication is performed through TCP connections.

Figure 2.14 presents the first analysis with 10 up to 1024 bytes. As can be seen the Cachelot solution performs better than the Memcached and Redis2 solutions, due to the fact

that it can store more information, i.e , approximately 14.5% more information for the same amount of reserved RAM.

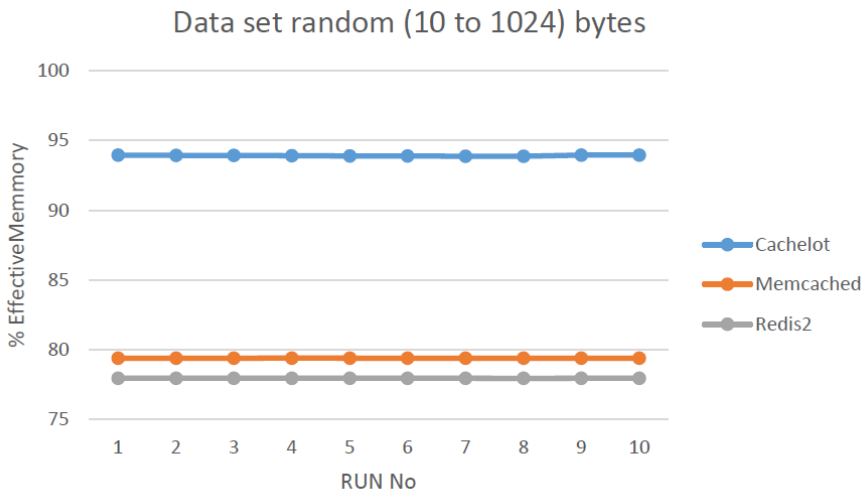


Figure 2.14: Data set random (10 to 1024) bytes.

Figure 2.15 shows the second analysis of 1024 up to 4096 bytes. Once more, it is observed that Cachelot solution performs better than both Memcached and Redis2. It is approximately 12% more efficient than the Memcached and Redis2 solutions.

Figure 2.15: Data set random (1024 to 4096) bytes.

Figure 2.16 shows the third analysis of 4096 bytes up to 1MB. As observed, with the increase on the data size, the difference is to become less evident, and once more Cachelot presents better performance in average, 10% more efficient compared to the other solutions.

Figure 2.17 presents the last analysis of 10 bytes up to 1MB. Similarly to the third analysis, Cachelot presents better performance (10% more efficient) than the other solutions. Cachelot

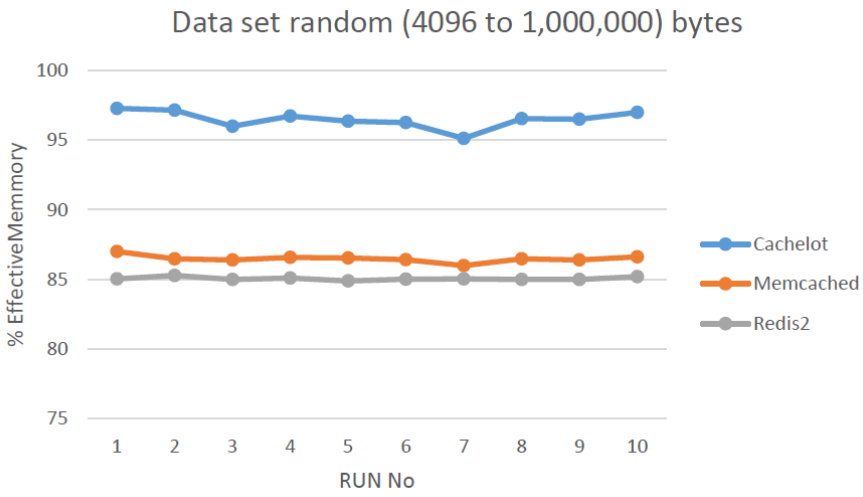


Figure 2.16: Data set random (4096 to 1,000,000).

keeps more items in the same amount of RAM, saving more RAM memory, depending on the data fragments and store patterns.

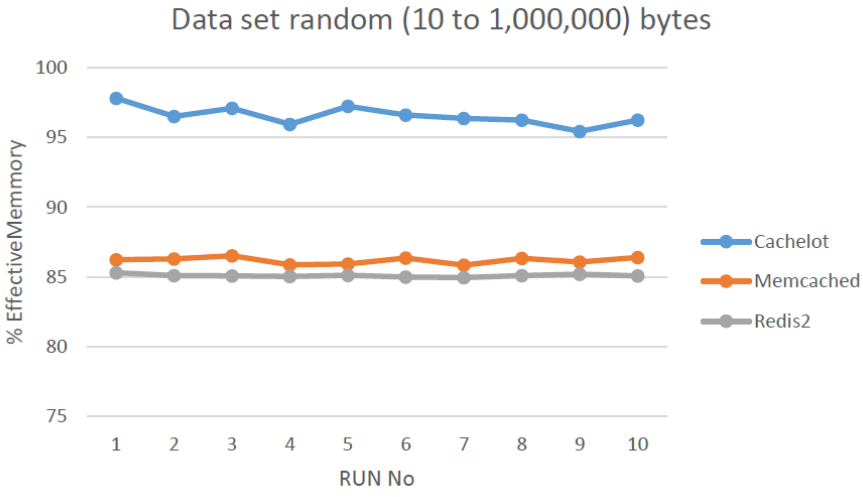


Figure 2.17: Data set random (10 to 1,000,000).

2.10.3 Consistent and Rendezvous Hashing

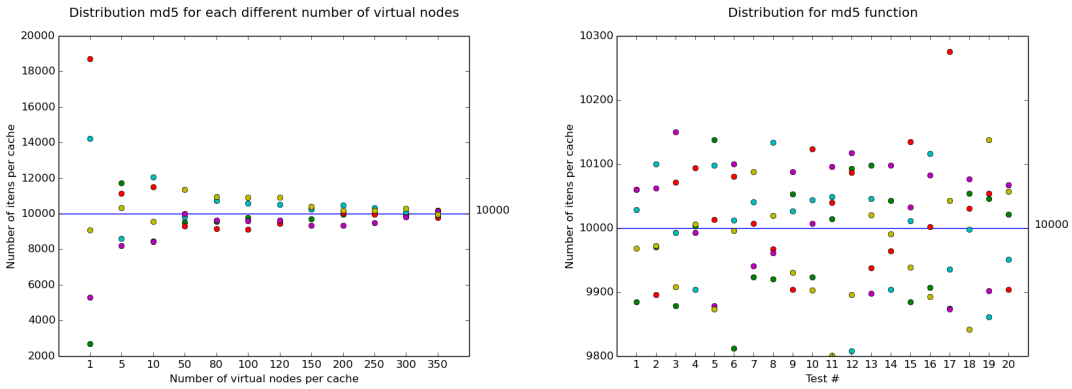
To achieve the proposed DSMC architecture requirements, it is required a distributed hashing algorithm. Many distributed hashing algorithms can be found in the literature[63], but the focus in this study will be on Consistent and Rendezvous hashing approaches which are common approaches used for distributed caching [53]. Therefore, a content distribution study using these two algorithms is performed with the purpose of selecting the one that presents the best performance under several increasing cache nodes. This evaluation is performed in

two phases:

- First Phase: - Evaluate the performance of consistent hashing vs rendezvous hashing under the same conditions;
- Second Phase: - Choose the best technology evaluated in the first phase.

For the first phase, a set of tests are performed to obtain an equal distribution among caches, considering the same conditions for both distributed hashing algorithms. The tests consist on randomly generating a string with size between 100 and 120 alphanumeric characters for each iteration. The number of items placed in each cache is counted, knowing that under these conditions the theoretical distribution should be the same among caches; in other words, it shall have the same number of items per cache. Also, in these tests, several hash functions are considered, using only 32 bits for the ring addressing space. The hash functions used are MD5, SHA-1, SHA-256 and SHA-512. The considered metrics are elapsed time and distribution relative error, taking into consideration the different virtual nodes, only in consistent hashing algorithm, used to rectify and minimize the unequal distribution. There are 50 thousand different keys used, so theoretically and uniformly, each cache node shall have about 10 thousand items.

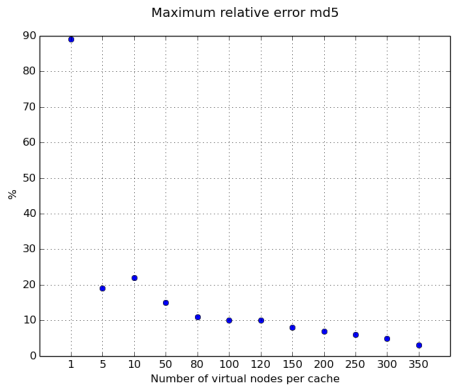
2.10.3.1 MD5: Tests



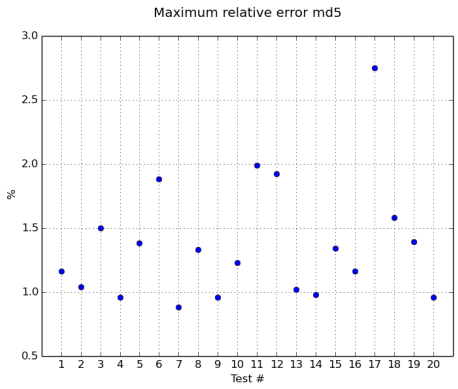
(a) Consistent Hashing (b) Rendezvous Hashing

Figure 2.18: Distribution - MD5

In Figure 2.18 each different color represents a cache. By varying the number of virtual nodes, the caches will have a different number of items. It is verified that the consistent hashing method tends to decrease the error in content distribution with the increase of virtual nodes. Also, observing the result of the consistent hashing method using the hash function MD5, it is demonstrated that, from the number of 150 virtual nodes per cache, the relative error is less than 10%. Also in this example, the standard deviation from 150 virtual nodes is less than 500 items among caches, and the error decreases linearly from 150 virtual cache nodes.

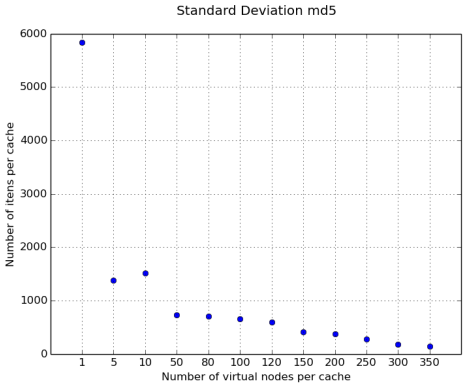


(a) Consistent Hashing

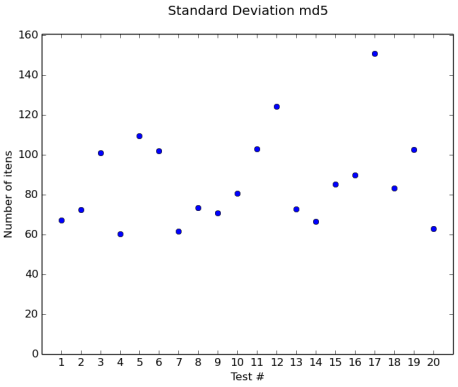


(b) Rendezvous Hashing

Figure 2.19: Maximum relative error - MD5



(a) Consistent Hashing



(b) Rendezvous Hashing

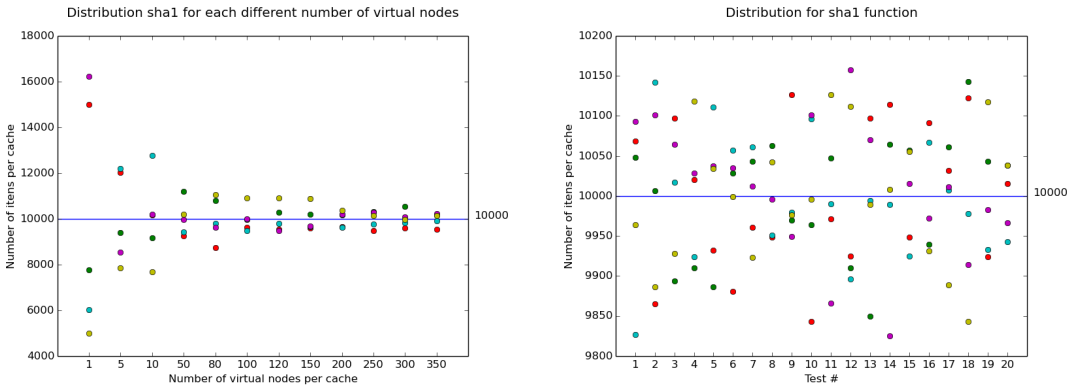
Figure 2.20: Standard Deviation - MD5

One the other hand, the rendezvous method in Figure 2.19b contains a low distribution error of the content, compared to the consistent hashing (Figure 2.19a), not exceeding 3% of the relative error, using the hash function MD5. The standard deviation (Figure 2.20b) is less than 160 items among the caches.

2.10.3.2 SHA-1: Tests

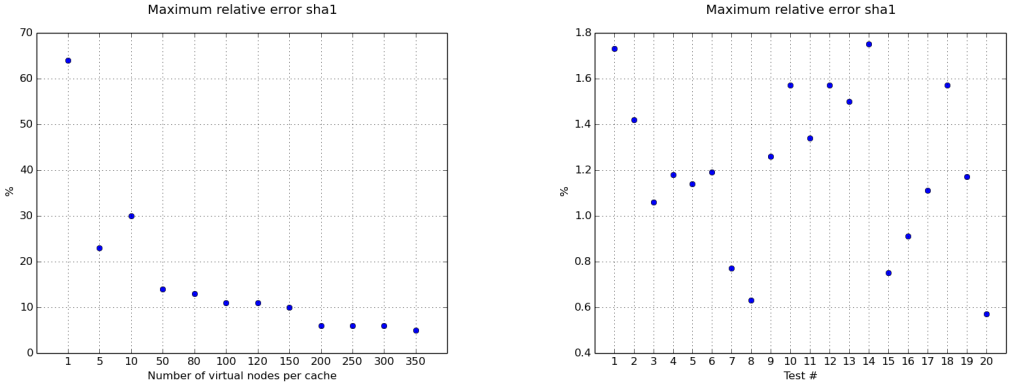
By using the hash function SHA-1, it is possible to observe that the consistent hashing method tends to decrease the distribution error of the content as it increases the number of virtual nodes (Figure 2.22a). Also, it is demonstrated that, from the number of 150 virtual nodes per cache, the relative error is less than 10%. The standard deviation (Figure 2.23a) from 150 virtual nodes is less than 500 items among caches. This also presents a more linear behaviour, since from 150 the error decreases linearly (Figure 2.23a).

By analysing Figure 2.21b, rendezvous method contains a low distribution error of the



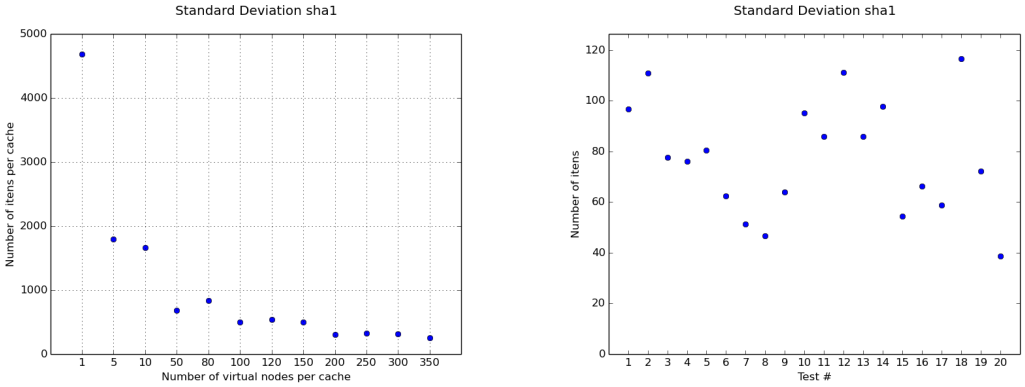
(a) Consistent Hashing (b) Rendezvous Hashing

Figure 2.21: Distribution - SHA-1



(a) Consistent Hashing (b) Rendezvous Hashing

Figure 2.22: Maximum relative error - SHA-1

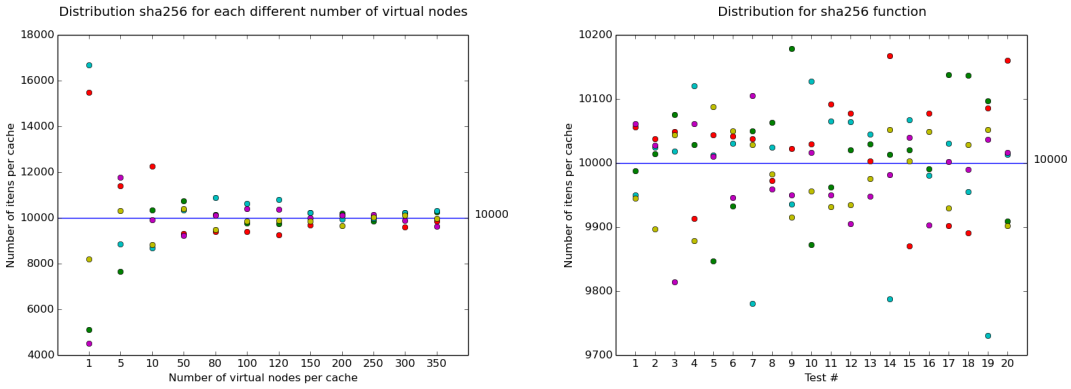


(a) Consistent Hashing (b) Rendezvous Hashing

Figure 2.23: Standard Deviation - SHA-1

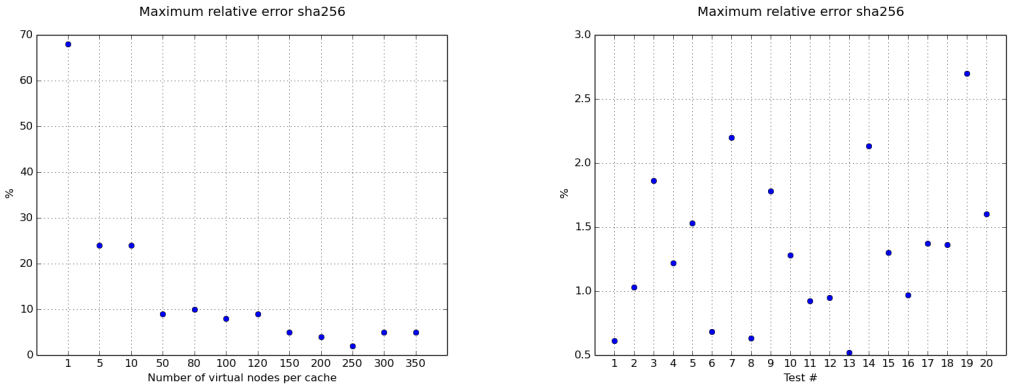
content, not exceeding 1.8% of the relative error (Figure 2.22b), using the hash function SHA-1. Also in this example, the standard deviation is less than 120 items among caches (Figure 2.23b).

2.10.3.3 SHA-256: Tests



(a) Consistent Hashing (b) Rendezvous Hashing

Figure 2.24: Distribution - SHA-256

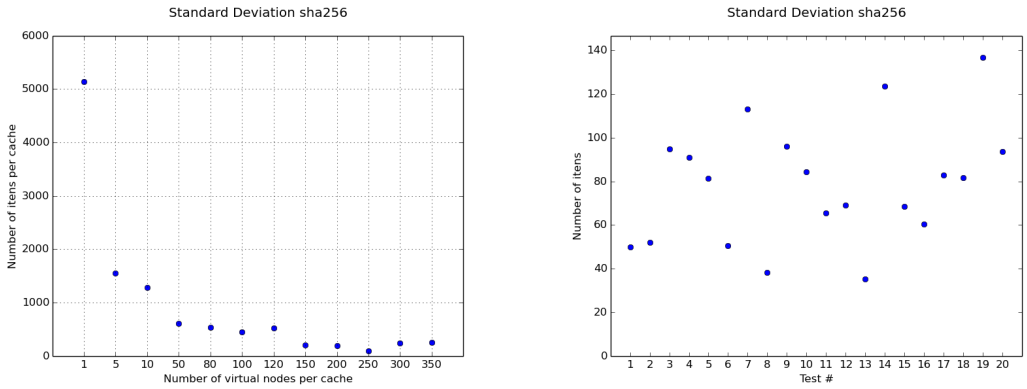


(a) Consistent Hashing (b) Rendezvous Hashing

Figure 2.25: Maximum relative error - SHA-256

By using the hash function SHA-256, it is possible to observe that the consistent hashing method, once more, tends to decrease the distribution error of the content (Figure 2.24a). Also, it is demonstrated that, from the number of 120 virtual nodes per cache, the relative error is less than 10% (Figure 2.25a). The standard deviation from 120 virtual nodes is less than 500 items among caches; however, it has a non-linear behaviour (Figure 2.26a), since for 300 and 350, it has an increase of the error compared to the previous ones, 150, 200 and 250. However, the error is less than 10% in both cases.

For Rendezvous hashing (Figures 2.24b,2.25b and 2.26b), the error is still low compared to the consistent hashing, not exceeding 3% of the relative error using the hash function

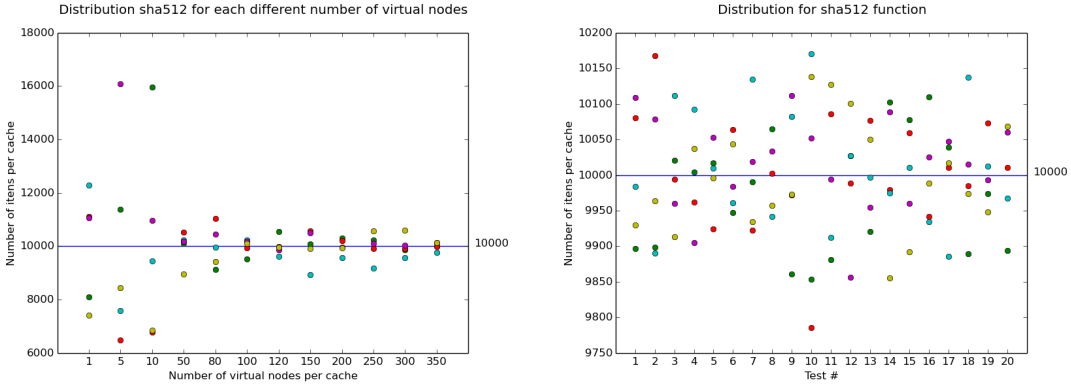


(a) Consistent Hashing (b) Rendezvous Hashing

Figure 2.26: Standard Deviation - SHA-256

SHA-256, and the standard deviation is less than 140 items, among the caches.

2.10.3.4 SHA-512: Tests



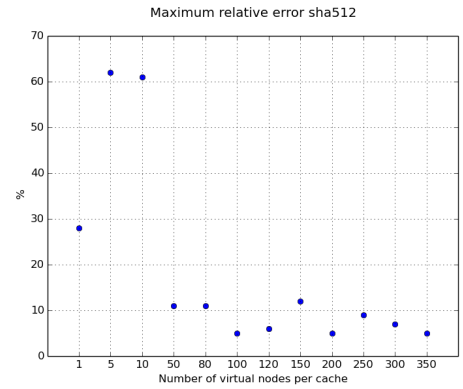
(a) Consistent Hashing (b) Rendezvous Hashing

Figure 2.27: Distribution - SHA-512

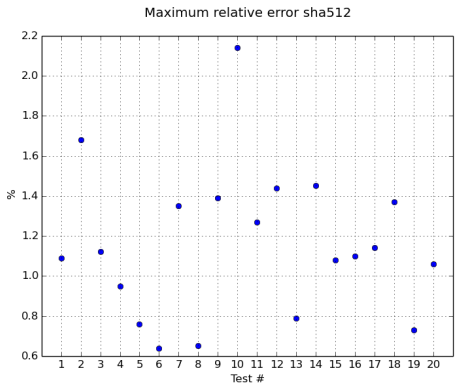
Finally, and using hash function SHA-512 with consistent hashing algorithm (Figures 2.27a, 2.28a and 2.29a), it is shown the relative error for 100 and 120 virtual nodes per cache, and once more it is less than 10%; however, with 150 virtual nodes the relative error is larger than 10%, and with 200 virtual nodes it is less than 10%. On the other hand, the distribution error of rendezvous hashing is low (Figures 2.27b, 2.28b and 2.28b), compared to the consistent hashing, not exceeding 2.2% of the relative error. The standard deviation is less than 160 items among the caches.

2.10.3.5 Elapsed Time

After analyzing both consistent and rendezvous hashing methods with several available caches, it is decided to use the hash function MD5 for both methods for the elapsed time

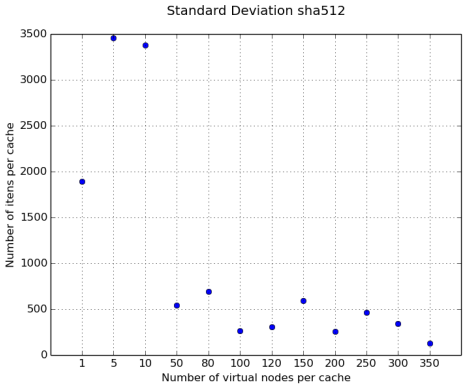


(a) Consistent Hashing

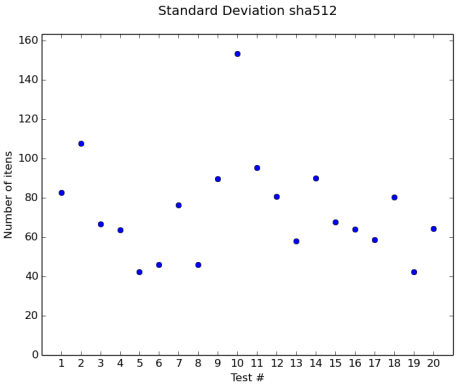


(b) Rendezvous Hashing

Figure 2.28: Maximum relative error - SHA-512



(a) Consistent Hashing



(b) Rendezvous Hashing

Figure 2.29: Standard Deviation - SHA-512

evaluations. The MD5 presents a linearity on relative error with the increase of virtual nodes, uses 150 virtual nodes per cache, and the relative error is less than 10% for both algorithms analysed. The elapsed time represents how fast each algorithm is, and is evaluated as following:

- measuring the elapsed time by distributing 50000 different keys over several caches and by each method to verify which algorithm is faster;
- measuring the distribution difference for different numbers of caches available.

Figure 2.30 presents time elapsed for each test under the same conditions, it demonstrates that the consistent hashing method, although more complex, is faster than the rendezvous method, which consistent hashing spent approximately 3.3 seconds to distribute 50,000 items, while rendezvous hashing spent 4.9 seconds.

Figure 2.31 shows that the different numbers of available caches, particularly 2,3,4,5, and reinforced by the number of caches available to distribute the content. The rendezvous hashing presents a more balanced distribution compared to the consistent hashing algorithm.

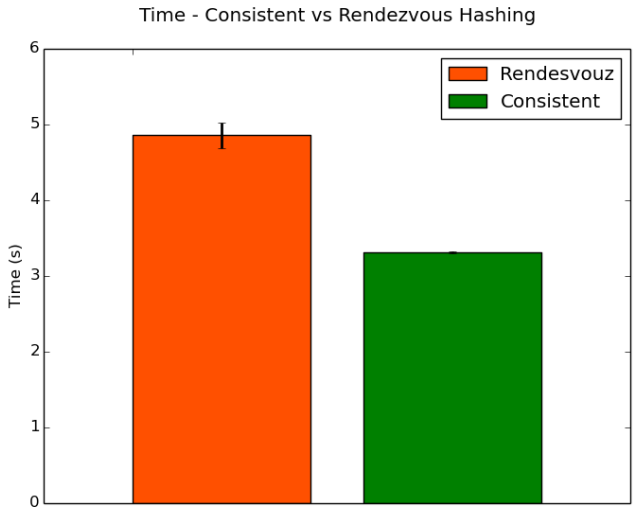


Figure 2.30: Time elapsed for the two methods

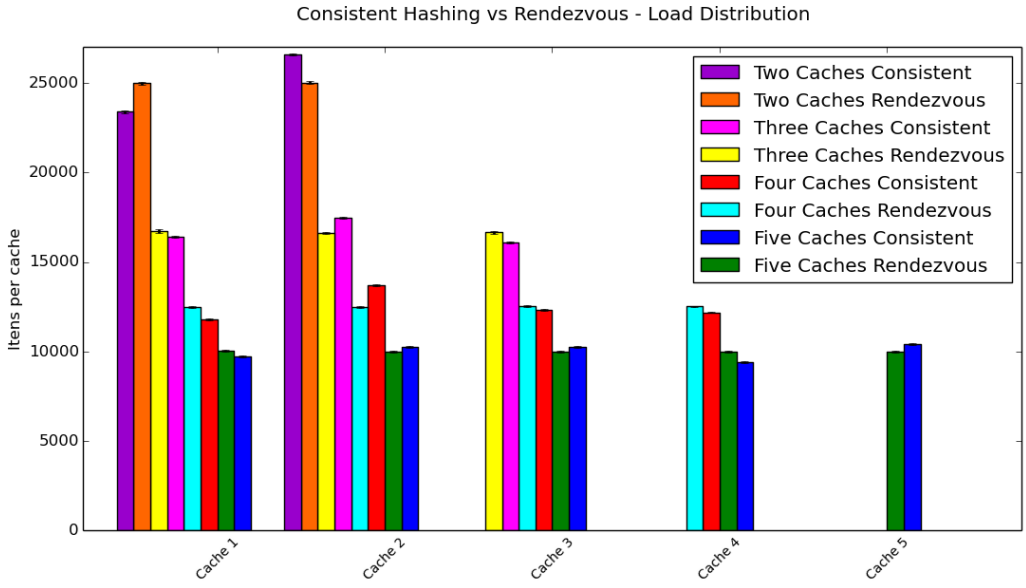


Figure 2.31: Consistent and Rendezvous Hashing Distribution several number of caches

---

However, the consistent hashing algorithm always presents an error less than 10% if 150 virtual nodes per cache are considered.

2.11 Chapter Considerations

This chapter presented the state of the art work in the area of distributed content disseminations and related aspects, as well as it assessed several technologies to be chosen and used in the scope of the Thesis. Considering the technologies assessment, the tests performed

allowed to conclude that cachelot solution is more efficient to manage and control the items in-memory cache, and that consistent hashing is faster than rendezvous hashing to decide which cache will be responsible to store or provide the content. Considering the functionalities of the proxy solutions, it is shown that Nginx can handle more concurrent connections and brings together all needed features of the proposed DSMC architecture which will be better detailed in the next Chapter.

Chapter 3

Proposed Distributed Caching Model and Architecture

After describing the concepts and solutions pointing out in the state of the art, this chapter presents the proposed OTT architecture model and its application on several scenarios (consumers at home, street and within a train), each one with its requirements and limitations. Therefore, this chapter also describes the technologies, algorithms and components of the proposed architecture, including the messages exchanging between the different components.

This chapter is organized as follows:

- Section 3.1 presents an overview of the distributed cache model with the proposed scenarios and use cases;
- Section 3.2 introduces the Distributed Smart Management Cache (DSMC) proposed architecture;
- Section 3.3 presents the final considerations.

3.1 Distributed Cache Model

In traditional CDN's each cache node typically has a built-in local cache that does not horizontally share content with other cache nodes. In our proposed architecture, a N-Tier approach is used, where the first cache tier and the one closer to the consumers comprises edge cache nodes responsible to delivery the content to consumers or groups of consumers. This approach pushes applications and services logic to edge caches, offloading origins/aggregators load requirements towards a distributed intelligent network. However, with the huge explosion of the mobile market and with the demand for higher quality resolution, such as 4k, especially for adaptive video, caches may have an additional effort.

This dissertation aims to propose a distributed cache network for content distribution, based on distributed caches on the network that work in a distributed manner. This approach makes essential to efficiently distribute content close to the consumers demands, lowering the latency from the consumers side. By using a distributed cache approach, it is possible to share the same pool of memory, increasing the memory as a unique logical storage structure within the whole distributed cache system. Each node might use other nodes' shared memory, taking full advantage of the distributed cache system with no content replication, avoiding

content redundancy, among the distributed caches, providing a global cache. A global cache is a pool of edge-cache nodes that work in a cooperative way, sharing the content load between edge group members. The group formation relies on the consistent hashing ring which are responsible to control and manage the content being cached at edge cache members.

At a first glance, the content will be closer to the consumer, and using the distributed caching model, the cache storage space is higher, thus increasing probabilities of content being request by the consumers to be cached, enabling to deliver the content faster, and saving the load in the aggregators and origin servers side, thereby improving QoE and service of the network. Moreover, due to the fact that the distributed cache model reduces the load of aggregators and origins servers, it also might handle more consumers, thus maintaining the same perceived QoE standards.

In addition, to increase data availability, this dissertation presents an integration of distributed cache model with prefetching mechanism. The prefetching mechanism is based on a forecasting mechanism which tries to cache the content before being required by consumers, enhancing access times and QoE, since most of the edge caches are physically closer to each others.

Figure 3.1 shows the distributed cache CDN model, comprising the global architecture along with its main components and scenarios. It presents an overview of the distributed content architecture starting from the scenarios point of view (consumers at home, street and within a train) with all modules involved in the communication process. A N-Tier caching approach is considered, where it is possible to add supplementary caching layers, denoted as 2-Tier caching, improving the distribution of the load of caching layers, as well as the available storage space. At a first glance, the content consumption pipeline starts from consumers which request chunks of a particular video to an edge cache. Then, the edge cache has to know if the content is in the global cache structure formed by its edge caches' group. If it has the content, then it delivers immediately to the client/consumer. Otherwise, the edge cache will request video chunks to the aggregator. If the aggregators do not have stored that content, then the request is forwarded to the origin server (the content Origin is responsible for holding the complete set of VoD and Live content). On the return, those chunks of content are stored in the requesting aggregator and forwarded to the requested edge cache that delivers it to the consumer, and also decides where to keep it stored in the global cache provided by the whole edge cache group members. To accomplish the distributed edge global cache, a strategy for adding, removing and updating edge nodes is proposed, which minimizes the content remapping among the edge nodes group, i.e., minimizes the need of moving data among nodes to keep a consistent cache state.

The distributed process is placed at the edge caches, along with the prefetching mechanism. The content being consumed by the consumer gives the overall service QoE estimation, which is measured through the MOS.

The key components of the proposed distributed model from the edge cache nodes will be individually described on the next subsections.

3.1.1 Origin

This component has the responsibility to provide the content required by the consumers, acting as conventional HTTP server and delivering the content to the consumers. The origin works under the operating system Microsoft Windows Server 2012, since it deliverers Microsoft Smooth Streaming adaptive HTTP-based content. The origin has 10 different videos available,

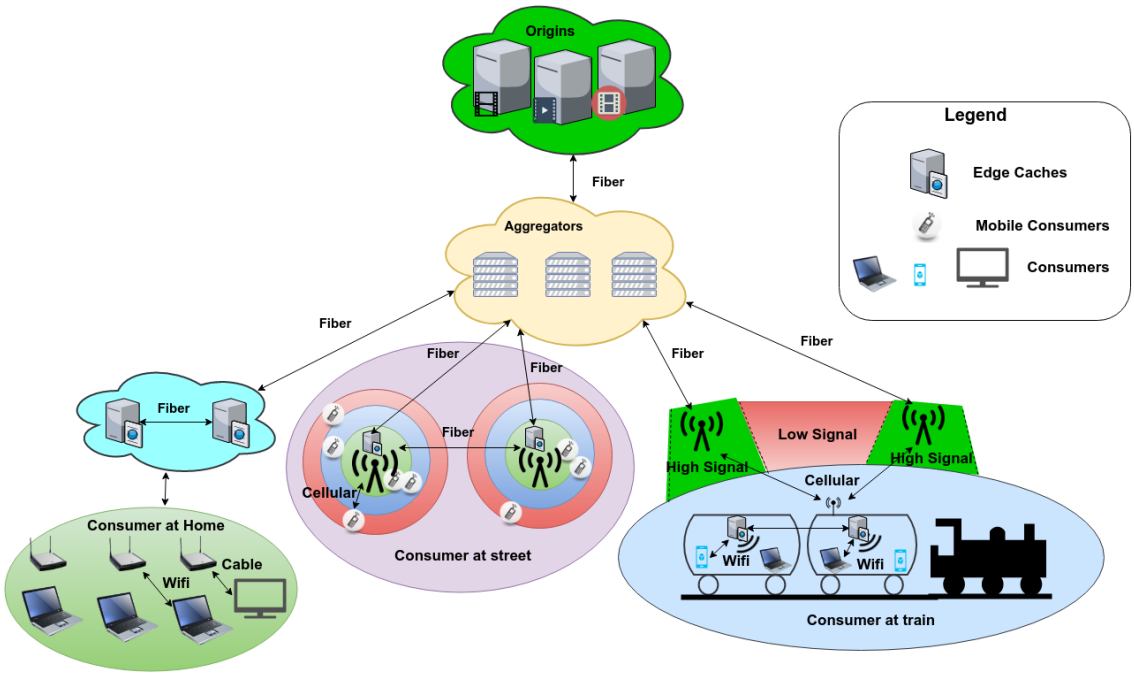


Figure 3.1: Distributed Cache Model

each video has 8 available qualities. The video used is Elephants Dream H.264 720p [1] with different bitrates as presented in Table 3.1, which shows the available qualities of the video and the average size of each video chunk per video quality. Therefore, Table 3.2 presents the available quality of the audio and its average chunk size.

Table 3.1: Average size of video chunks and bit rate of the several qualities

Qualities (Bitrate)	Average size (Bytes)
230,000	57,500
331,000	82,750
477,000	119,250
688,000	172,000
991,000	247,750
1,427,000	356,750
2,056,000	514,000
2,962,000	740,500

Table 3.2: Average size of each audio chunk and bit rate

Quality (Bitrate)	Average size (Bytes)
160,000	40,000

[1] `https://www.iis.net/downloads/microsoft/smoothstreaming`

3.1.2 N-Tier Caching

A N-Tier model is considered, which uses 2-tier caching since there is benefits of having caches between the origin and the edge caches, the aggregators. In a mobile scenario the consumer is moving from one access point to another. If the access points are not in the same edge cache group, the content does not need to be requested from the origin, since it is served from the aggregator. Also, when an edge cache fails, the aggregator provides the content, then there is no need to be requested from the origin. The aggregation cache is widely used in a traditional CDN when there are larger delays between the the clients and the origin. With the introduction of an additional tier caching, the end-to-end delay will be larger for the content that needs to be fetched from the origin, if the content is not in any of the tier caching. This extra delay might be compensated when the required content is available close to the consumer[11].

Aggregator

The Aggregator nodes are intermediate caches between edge caches and the origin servers, which have the main functionality of reducing excessive backend traffic to the origin servers. To accomplish this task, the aggregator works as a proxy caching solution that implements the reverse proxy and to cache the content features. In other words, the edge caches do the request to the aggregators and the aggregators have to verify if it is in its cache. In case the content is in cache, a hit occurs, then the aggregator sends to the edge cache the required content, not needing to fetch from the origin, saving bandwidth and load on the origin side. Otherwise, if the aggregator does not have the content requested by the edge caches, it will request from the origin server. Thereafter, when the origin sends the content being requested, the aggregator delivers it to the edge cache and, at the same time, it keeps a copy of the content in cache. To perform this functionality, Nginx is chosen which presents the reverse proxy features with integrated caching.

Edge Caches

The edge caches are the first contact with the consumer requests. As can be seen in Figure 3.1, these edge caches work in groups, in a distributed manner, sharing the content between them, avoiding content redundancy and increasing their cache performance and hit ratio, which also reduces load for aggregators, saving bandwidth and load on the aggregators. Also, since they share the same storage space, it is possible to increase the size of the caches in the group, allowing to have more content close to the consumer demands, which is an important requirement considering that the trend is to have videos with more quality resolution, implying a larger and distributed global storage space.

3.1.3 Consumers

Consumers are end-user entities (e.g., cell phones, tablets, PCs, Set-top-boxes, etc.) which consume mostly chunks of a particular video from the edge caches. In this dissertation, real consumers are provided by smooth streaming players. Therefore, smooth streaming technologies are adaptive, which means that is also possible to measure the conditions imposed by the network and by consumers' habits, and which qualities are currently being consumed. In case that the network does not present the best conditions (e.g., high jitter, and latency, poor bandwidth, etc.), it is also possible to measure the impact on the whole system, of not having the homogeneity between the qualities of the different consumers which may request different bitrates. The conditions imposed by the network can be defined by the type of

scenario (fixed, mobile, or highly mobile) described in the next subsection.

3.1.4 Proposed Scenarios and Requirements

This subsection describes the content delivery scenarios and their main requirements considered and evaluated in this dissertation. They consider the most relevant and challenging network environments to ensure the best possible experience of consuming content in different situations (fixed, mobile and highly mobile), taking into account the model presented in Figure 3.1.

3.1.4.1 Consumer at Home

The consumer at home is characterized by the customers watching videos in a more stable network, which do not suffer from high impairments regarding latency or low bandwidth, and supports all kind of network traffic over cooper/fiber links, which are typical conditions suitable for consumers at home. The network is capable of supporting the consumption of various clients at home. The introduction of the edge caches with distributed caching is possible to improve aspects that allow saving the load and the bandwidth consumed in the upstream servers, aggregators, since they present a larger cache size and sharing the content between them. The aggregators tier might support more clients, maintaining high QoE levels.

3.1.4.2 Consumer Mobility at Street

Mobility at streets scenario considers several mobile clients that are scattered over 3 different wireless access regions, where each region presents different impairments mostly caused by the backend link latencies. Each edge cache node plays the role of Access Point (AP)'s with caching functionality for the content mobile consumers. The link between edge cache nodes and the consumer suffers from limitations, such as, latencies and higher jitter due to physical obstacles and the distance of clients from the AP's, which are characteristics of mobile networks over urban or rural environments. In this type of scenarios, due to the fact that there are different network conditions for the various clients, there is no homogeneity of the qualities consumed, compared to the consumer at home scenario (stable scenarios the quality is pushed to the maximum bitrates), generating an impact on caches performance, bandwidth utilization, etc. Therefore, adding edge caches with distributed caching functionalities will make possible to reduce the load and bandwidth consumed in the upstream servers, mostly on aggregators, as the edge caches are able to increase the cache size horizontally, and the content is shared between the various edge caches. As a result, the aggregators might support more clients, maintaining high QoE levels even when the scenario enables mobile consumers.

3.1.4.3 Consumer at high speed mobility on a train

Consumers at the train scenario considers that the consumers are within a train wagon, where they can connect to on-board AP with caching functionality via Wi-fi on each wagon. Each AP can communicate to the aggregators via a cellular network. The limitations in this type of scenario are in the link between the aggregators and the edge caches, since the limitations are characterized by the fact that they are in zones with high signal and others with low signal. Consumers have more limitations compared to the previous scenarios, since there are areas where video can be downloaded, since it has good bandwidth, and others where the

network does not support consumer demands, since it is in a rural area or even within a tunnel that does not have connectivity to the mobile network, causing video freezing resulting on unsatisfactory perceived QoE from consumers. The insertion of edge caches with distributed cache support in each AP is possible to reduce the bandwidth consumed from the link between the aggregators and edge caches, since they are able to share content among the various edge caches. The distributed caching approach enables the cache to share content between them, increasing the cache size, and therefore allowing the prefetching approach to place the content in advance. For example, in areas with good connection, the prefetching approach might request content chunks that will be consumed in areas with low signal, allowing the client videos to not freeze during the journey, improving consumers perceived QoE.

3.2 Distributed Caching Architecture

In this section the proposed DSMC architecture is described, jointly with its building blocks and communication between edge caches (locally at each edge cache) as well horizontally with its neighbours.

Figure 3.2 shows the block diagram and interactions of an edge cache. The proposed architecture for one edge cache is composed mainly of four modules: a proxy cache solution, the DSMC, a In-Memory cache solution, and a prefetching mechanism. The following subsections will detail each module and its interactions.

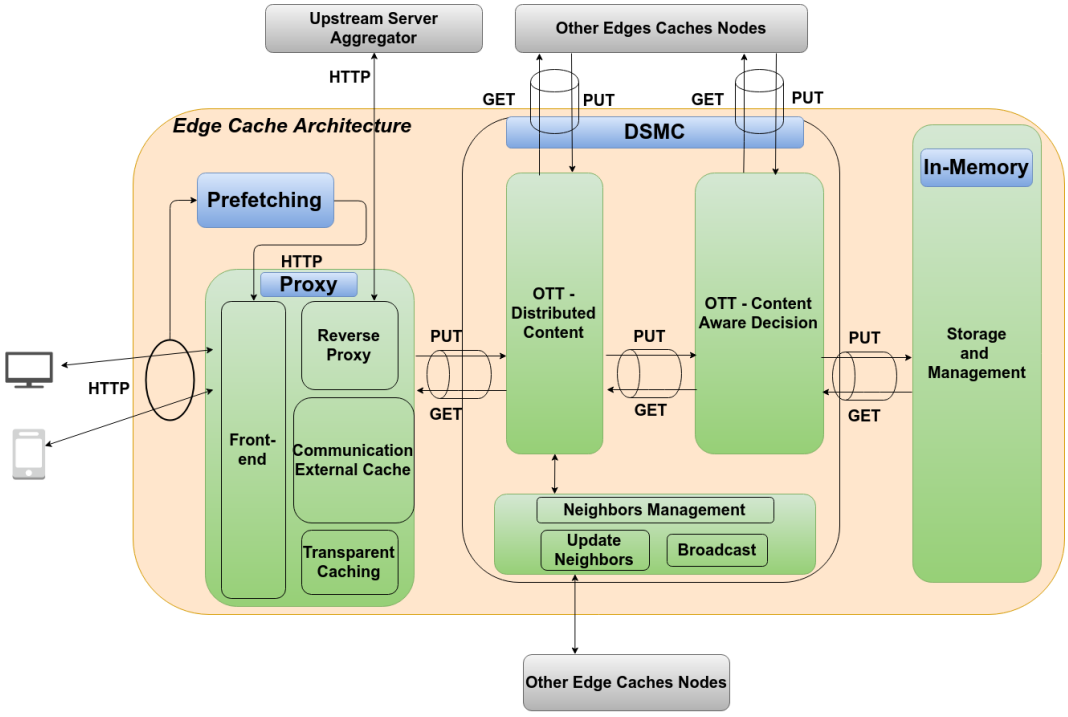


Figure 3.2: Architecture from the Edge Cache

3.2.1 Proxy Cache Solution

This module is responsible to handle HTTP consumers and prefetching requests, and it works under the Nginx proxy cache solution. As can be seen in the Figure 3.2, the proxy cache solution interacts with several other features (Frontend, Reverse Proxy, Communication External Cache and Transparent Caching), and communicates between other modules through PUT and GET messages, and to the upstream server aggregators through HTTP requests:

- The communication to the external cache uses the Memcached messages protocol, to retrieve and insert the content from or to an external cache;
- The transparent caching, if a certain content is not in the cache; then, it needs to be stored.
- The reverse proxy is used as follows: if the content is not in the cache, it is necessary to send the request to the upstream servers, the aggregator; and when the aggregators answer with the content, it needs to be delivered to the client.
- From the proxy caching solutions presented in chapter 2, the Nginx and Squid were compared in previous works and present the features to implement the reverse proxy and cache the content, in the integrated cache, which means that the caching of the content is done in the cache provided by the proxy. These two solutions can be used in the aggregator, since these only have a local cache. The Nginx solution was used in the aggregator instead of Squid, due to the fact that in the edge caches Nginx is also used with different functionalities. Thus, the Nginx is used for the integration with the whole presented system.
- In the edge caches, the Nginx solution was used, since it was the only one that presented to have the 3 features necessary for the integration with the whole presented system.
- For each user HTTP request, Nginx checks if the required content is in the distributed cache, via the DSMC module. If the content is in the global cache, a hit occurs, then DSMC returns the contents to the Nginx and it delivers immediately to the client. Otherwise, If the content is not in a global cache, then it is fetched from the aggregator; when the aggregator responds, the content is sent to the client, and at the same time a copy of the content is sent to the DSMC module.

3.2.2 Prefetching

The prefetching [33] has the aim to predict the future consumption of clients and populate the caches with the content before the consumer requests it. In other words, when a consumer requests the content to the proxy via HTTP, the prefecthing mechanism calculates the next chunks and predicts their qualities, which customers will later request them. Once the prefetching predicts the future chunks, it requests those chunks to the proxy via HTTP, which will eventually be requested by the clients. Thus, the prefetching pre-populates the cache.

The normal requests and prefetching requests are differentiated and handled differently by the system. In chapter 4 this differentiation and different handling will be described.

3.2.3 Smart Management Distributed Cache (DSMC)

The DSMC is composed of 3 main modules: OTT-Distributed Content, OTT-Content Aware Decision and Neighbors Management, each one with its specific functionalities within the DSMC distributed edge caches.

3.2.3.1 OTT-Distributed Content

This module has the main function to distribute the content among the several edge caches. Therefore, when it receives the Nginx requests messages via the Memcached message protocol, the content ID, key, is obtained from the message. With the distributed hashing algorithms, it is possible to choose the edge cache that is in charge of that particular content.

In this module we used Consistent hashing[53] algorithm to find out which edge cache is responsible for that content. This module sends the memcached message to the OTT-Content Aware Decision of the edge cache responsible for that given content. Also, this module has to distinguish the requests of the prefetching, so that it is possible to reduce the overhead created by the prefetching as well as to maintain the consistency of the prefetching and clients requests.

Consistent hashing is used to decide on how to distribute the received content chunks by edge cache neighbours in a based ring topology and by local cache, through the use of a based distributed consistent hashing algorithm. As soon as the content needs to be stored/fetched, the consistent hashing algorithm will automatically choose an edge cache where it can store and/or fetch the data. When the decision relies in only one edge cache, the decision is easy; otherwise, as soon as multiple edge caches make part of the ring, which leads to our case, the consistent hashing shall find a way to store the content across multiple edge caches.

We have proposed a simple version of consistent hashing [53] distributed algorithm, which allows addition and removal of edge cache nodes from the ring, and remapping of only one part of the content. The key fragments of the video are placed on each edge cache according to a weighted distribution, which prioritizes edge caches with more resources available (storage and processing capacity). This approach allows to balance the content distribution among edge caches, minimizing access overhead and bandwidth consumption on a few set of popular nodes. Therefore, the DSMC has to update the neighbours, as well as to receive the messages from Nginx and, through the consistent hashing, send the messages to the neighbour caches. Each video fragment has an associated value/content key, whose hashing is used for video content fragments management and control across the DSMC architecture. Virtual nodes pointing out to physical edge cache servers are adopted in order to allow a better load balance between the physical nodes. The virtual nodes' structure is managed and controlled by the consistent hashing module.

Fig. 3.3 shows the ring topology of consistent hashing. The content can be mapped in any part of the ring, that is, in any cache server belonging to the ring. So, the video chunk content distribution is based on the content chunk ID, extracted from Uniform Resource Locator (URL), hashing. This results in an integer, which should correspond to the content's cache server position in the ring. Therefore, it is necessary to move clockwise along the ring to find the edge cache server where this content will be stored. If the exact position does not exist, the content is stored in the cache server in the closest (in a clockwise manner) position.

The consistent hashing ring is updated by the sub-module Update Neighbours 3.2.3.3. This sub-module is crucial for updating the available edge caches to store and serve content.

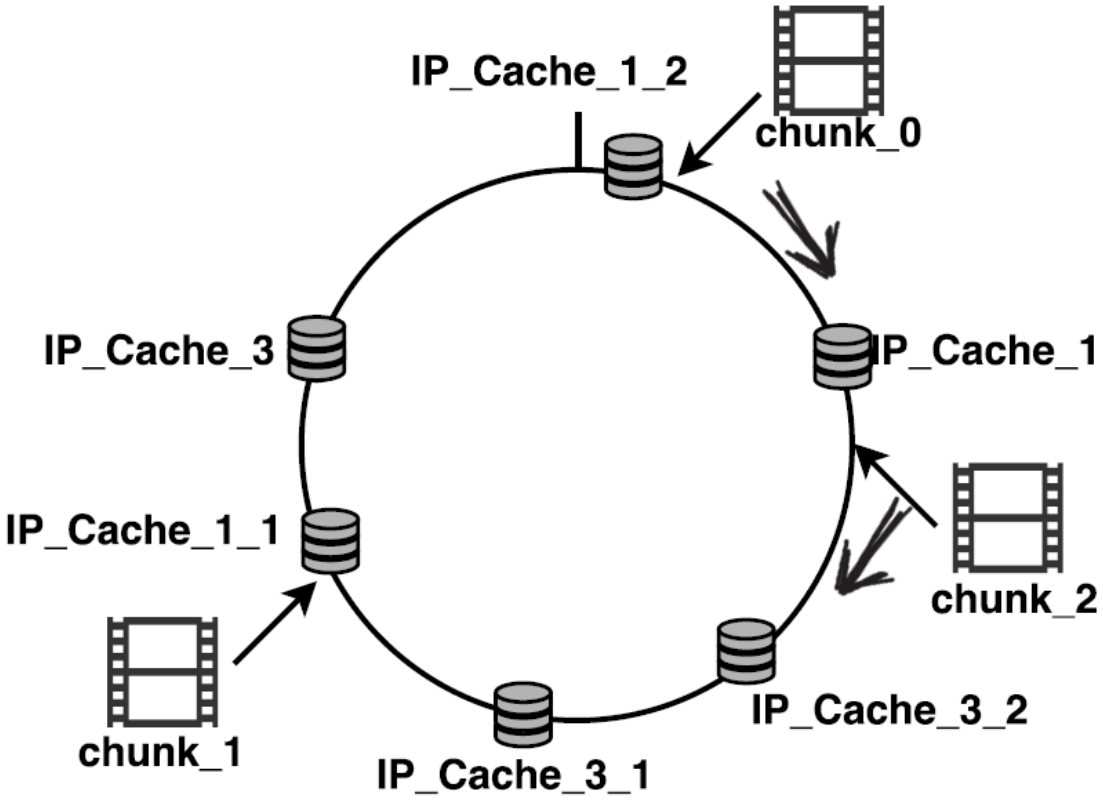


Figure 3.3: Consistent hashing ring topology with virtual nodes

As the number of available edge caches increases, a larger global cache size is available. Also, note that with the use of virtual nodes it is possible to have a weighted distribution, allowing caches with higher memory size to receive/serve more content, proportional to their cache size.

3.2.3.2 OTT-Content Aware Decision

This module receives the Memcached protocol messages from the OTT-Distributed Content module of the various edge caches. The aim is to implement a management of the received requests: when receiving multiple requests from the same content of the various edge caches and if this content is not cached, the miss occurs, and subsequently it unnecessarily overloads the aggregators, so the requests have to wait for the contents to be stored.

A mechanism to control the requests through a lock and release based approach is proposed in order to allow the content to be received and stored, and then delivered to the consumers. The edge cache responsible for this lock is the one that is in charge of the given chunk/content. The edge cache that is in charge to fetch the content from the origin is the first to verify that the chunk is not in cache, while the others have to wait for this edge cache to SET (store) the content.

For such purpose, this module has a list of requests that were recently requested and that were a miss. Thereafter, the requests that are in the list, have to wait for the content to

be stored in the cache. The edge cache responsible for this lock is the one that is in charge of the given chunk/content. The egde cache that is responsible to fetch the content to the aggregator is the first one to verify that the chunk is not in cache, while the others have to wait for it to put the content. This effect happens with a huge impact on the Live scenarios, but it also happens with some probability in VoD. To surpass these cases, after verifying that a certain content is not found in the list, it is then sent to the in memory cache via the Memcached messages protocol, to get the content. If a message to put the content is sent, then the set to insert the content is sent to the in memory cache, and the ID/key of the chunk is removed from the list, so that the waiting queues can fetch the contents.

3.2.3.3 Edge Cache Neighbours Management

This module consists of two sub-modules, Update Neighbours and Broadcast, which aims to know how many available caches exist in the network.

Update Neighbors

This module has to find and be aware on how many neighbours exist. This is accomplished by sending broadcast messages using UDP, informing about the cache storage size and broadcaster Internet Protocol (IP) address. Through these messages exchange, it is possible to have an updated list of neighbour edge caches, which it is used to update the consistent hashing ring. So, this module updates the consistent hashing ring, that is used in the OTT-Distributed Content to distribute the content by the available caches updated by this module.

Broadcast

This module aims to broadcast periodically the information about the edge cache. In the broadcast packet, the size of the cache for each edge cache is sent, allowing the other edge caches to know what is the cache size the edge cache has available, as well as which edge cache is available to cache and serve the content. This process will be detailed in Chapter 4.

3.2.4 In-Memory Cache

This module receives Memcached messages from the OTT-Content Aware Decision module. Depending on the type of message, a different operation is performed. If the message is a PUT message, which means a SET, the contents are stored in the cache. Otherwise, if it is a GET message, the content is verified if it exists in the cache. If the content required is in cache, then a hit occurs and it is sent to the OTT-Aware Decision module. Otherwise, it the content is not in cache, then a miss occurs and it is sent to the OTT-Aware Decision module that the content is not cached. Then, the locally in-memory content storage of the edge cache following rules of a content eviction policy is performed.

Although there are several eviction policies, in this work it is used the LRU eviction policy from Cachelot solution. LRU deletes the objects (content) that are not used for the longest period of time, i.e., not necessarily the largest object and even not the first object stored in the edge cache. Cachelot uses an in-memory key-value data model that contains a single value for each key, and that value is the content being accessed through the search of the respective key. As soon as Cachelot receives a message SET from the OTT-Content Aware Decision module, it will store the content locally, and then Cachelot responds if the request is stored successfully. In case of GET messages, Cachelot checks if the content is in local cache,

which returns the required content (a hit) in a positive case; otherwise a miss is returned and then the video content is locally stored in memory.

3.3 Chapter Considerations

This chapter described the overall proposed OTT architecture model and its application on several scenarios (consumers at home, street and within a train), each one with its requirements and limitations. This chapter also described the edge cache architecture, its modules and interactions. The next chapter will describe in more detail the several modules and the interactions between them, and how they will be implemented in an overall integrated system.

Chapter 4

Implementation

This chapter focuses on the implementation of the distributed content caching framework from its practical point of view.

The implementation is divided into two main blocks: consumers which consume the video and the Edge Caches which implement the distributed functionalities of the proposed architecture.

The implementation of the blocks is developed using python [64] language, due to its easy prototyping with good documentation support.

The chapter is organized as follows.

- Section 4.1: presents information about consumers.
- Section 4.2: presents the implementation of the edge nodes and all the associated functionalities.
- Section 4.3: presents the chapter considerations.

4.1 Consumers

The role of the consumers is to consume the content video, generating traffic on the network. In this work, the consumers are implemented through a probe, which has been implemented in our group NAP in IT Aveiro [37]. This probe contains a model to determine the QoE estimation using HTTP adaptive video streaming. To estimate the QoE, several metrics are extracted throughout the video playback session. This probe supports live content and on-demand Microsoft Smooth Streaming streams. To estimate the QoE it uses several metrics, such as bit-rate, frames per second of a specific chunk, re-buffering and screen resolution ratio.

These metrics are described in the following:

- Bitrate: The user QoE is highly influenced by the bitrate. In the adaptive video streaming, the video is encoded with several qualities. Each video quality chunk is encoded with a constant bit rate, then the chunks of the same quality have an roughly equal size in bytes. However, in highly dynamic scenes there are lower compression gains, which it compromises the constant bit rate per chunk quality[37].

- Frames per Second (FPS): The number of FPS is a limitation of the device. The device can offer a higher resolution in exchange for the decrease of the FPS. Although, a video that presents lower number of FPS or a drop is evident to the consumers, which compromises the QoE[37].

- Re-buffering: The Re-buffering is the elapsed time that a player waits to download a new chunk after the video freezes. If the video freezes, the user has a poor QoE, since the requested video stops playing[37].

- Screen resolution ratio: The ratio between the video quality offered with the available screen resolution size is important to know if the best quality video is offered to the consumer. If this ratio decreases, which means, the quality of the video is lower than the one it is possible to show, the user has a low QoE. Also, with this approach it is possible to distinguish the size of a device with a smaller and larger screen[37].

Considering these metrics, depending on the conditions and limitations imposed by the network and the whole system, the QoE is highly influenced.

For each probe, the client, an output is obtained with several metrics of the execution of the entire video. Figure 4.1 shows the output log of the probe.


```
{"manifest":{"live":false,
"duration":653791,"video":{"codec":"H264","maxWidth":1280,
"maxheight":720,"tracks":
[{"b":230000,"w":224,"h":128},
{"b":331000,"w":284,"h":160},
{"b":477000,"w":368,"h":208},
{"b":688000,"w":448,"h":252},
{"b":991000,"w":592,"h":332},
{"b":1427000,"w":768,"h":432},
{"b":2056000,"w":992,"h":560},
{"b":2962000,"w":1280,"h":720}]},
"audio":{"codec":"AACL","tag":255,"bitsPerSample":16,
"channels":2,"packetSize":4,"samplingRate":44100,
"bitrate":160000}},"configuration":
{"url":"http://192.168.122.116/development/smoothstreaming
/ElephantsDream_1.ism/Manifest","screenWidth":1280,
"screenHeight":720,"fps":24,"output":"out0.json"},
"qoe":[{"t":1819,"v":1.2575,"b":230000},
{"t":3862,"v":1.2725,"b":2962000},
{"t":5864,"v":2.70225551763106,"b":2962000},
{"t":7866,"v":3.7094222701352,"b":2962000},
{"t":9864,"v":4.15311319081205,"b":2962000},
{"t":11866,"v":4.37520811226299,"b":2962000},
{"t":13864,"v":4.50970219185816,"b":2962000},
.....
.....
.....
{"t":651929,"v":4.76474227231706,"b":2962000},
{"t":653930,"v":4.77306434483021,"b":2962000}]
"buffering":[{"t":1856,"v":114}]}
```


Figure 4.1: Probe output log

By the analysis of the image it is possible to observe that the probe has 3 different fields:

- Header field: The header field presents the information about the characteristics of the video and the resolutions available. Each set defined by {b, w, h} represents the different qualities of the content, where "b" is the bitrate, "w" and "h" are the screen size resolution, width and height, respectively[37].

- QoE field: The QoE field presents the logs of the consumed chunks, each chunk has approximately 2 seconds of the video. Each set in this field is defined by three metrics. An example taken from the figure is described as {"t": 13864, "v": 4.50970219185816, "b": 2962000}, where "t" is the time at which each particular chunk is requested in msec, "v" is the estimated MOS and "b" is the bitrate required by that given chunk[37].

- Buffering Field: The Buffering field presents the time when a video freezes and the duration of this gap. The set in this field is defined by 2 metrics, as presented in the buffering image: {"t": 1501, "v": 135}, where "t" is the time, in msec, when a video freezes, and "v" is the duration, in msec, when the video was not playing[37].

The probe consumes the content as a real smooth streaming consumer, which means that, at the beginning of the video, it requests the manifest file that presents the qualities of the video, audio and its heuristics. After consuming the Manifest file, the probe starts to fill the buffer, since it is empty. The size of the buffer is about 30 seconds. Due to the fact that it fills the buffer at the beginning of the video, a consumption peak in the network is generated, in order to download all the chunks until filling the buffer. For each chunk it is calculated the QoE, the quality of the chunk, it is analyzed if the video was buffered and the time it was doing buffering.

Consumers consume different videos with different popularities. This work considered 10 available videos. The Zipf's law [65] function, an empirical law that allows predicting the popularity of the videos, is used to determine the video popularity. This work is tested for the different curves of popularity as shown in Figure 4.2 which presents the cumulative distribution function for three different probability curves. The curves are given by the calculation of the Equation 4.1:

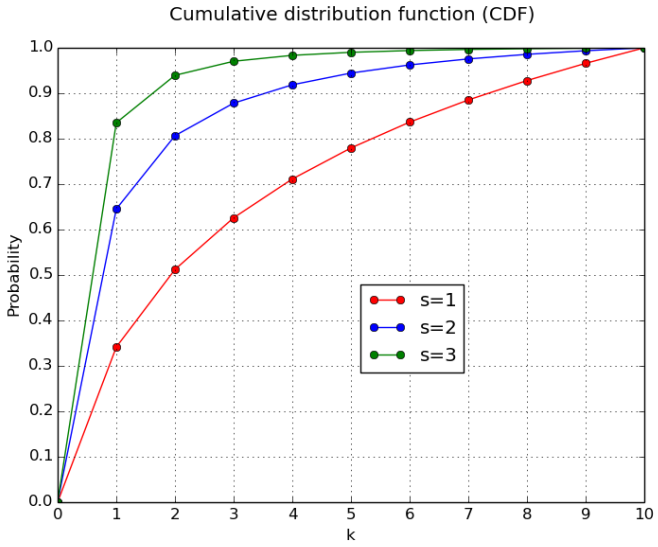


Figure 4.2: Cumulative distribution function - Zipf's law

$$CDF(k; s; N) = \frac{\sum_{x=1}^{k} \frac{1}{x^s}}{\sum_{x=1}^{N} \frac{1}{x^s}} \tag{4.1}$$

1. N is the number of videos available, in this case 10.

2. k is the ID of video.

3. s is the value of the exponent characterizing the distribution, in this case 1, 2 and 3.

Knowing that the x-axis is the identifier of the video number, which means that each point represents a video in the universe of 10 different videos, for each video there is a probability of popularity: the video is more popular than the others, based on the probability given by the Zipf' law function.

4.2 Edge Caches

The edge caches architecture has several modules, such as Prefetching, Nginx, DSMC and Cachelot. Figure 3.2 presents the architecture of a single edge cache with the interaction of the modules. As can be observed, DSMC interconnects the Nginx and Cachelot modules. The DSMC receives the Memcached messages from Nginx and uses consistent hashing to find the edge cache that is in charge of that content, sending the Memcached messages to this edge cache. It also has to differentiate the Prefetching messages sent by the Nginx, in order to reduce the overhead created by the Prefetching mechanism, if the Prefetching request is a hit. Also, this module has to update the edge cache neighbours to have an updated edge caches nodes to serve and cache the content. Moreover, the DSMC module has to implement a mechanism to lock a given request chunk that was recently requested and was a miss. Thereafter, the request has to wait until that content is stored in cache. Thus, it must have a list of the requests recently made that were a miss; when there is, simultaneously, the same request by the several clients in different edge caches, all of them will be a miss, and subsequently unnecessarily overload the aggregator, so the requests have to wait for the contents to be stored. The following subsections will present more details on the implementation of the several modules and the communication between them.

4.2.1 Nginx

Nginx is chosen because it is possible to add all the features described in subsection 3.2.1. Nevertheless, as described in section 5.2.1.1, the Nginx base platform does not have the communication with an external cache, as well as the implementation of the several features required with an external cache. However, Nginx allows to make external modules to work with its source code. Then, there was an exhaustive search to find external modules that allow to build the required functionalities. The modules found are made by several teams from the open source community, as can be seen in section 5.2.1.1. In order for the Nginx to communicate with the DSMC module, the Nginx was built with the modules presented in section 5.2.1.1. Thus, the Nginx has the features and the interactions represented in its flowchart. Figure 4.3 presents the Nginx flowchart.

The configurations and all interactions have been provided in the framework of this thesis.

Nginx is used to handle the HTTP requests from the consumers and the Prefecthing mechanism. For each HTTP request from the consumers or Prefetching mechanism, Nginx has to read the HTTP URL and create the TCP Memcached protocol message "GET". This message needs to contain the key of the path and the arguments from the HTTP request, and send it via TCP to the DSMC. When the DSMC responds, the data is read by the Nginx.

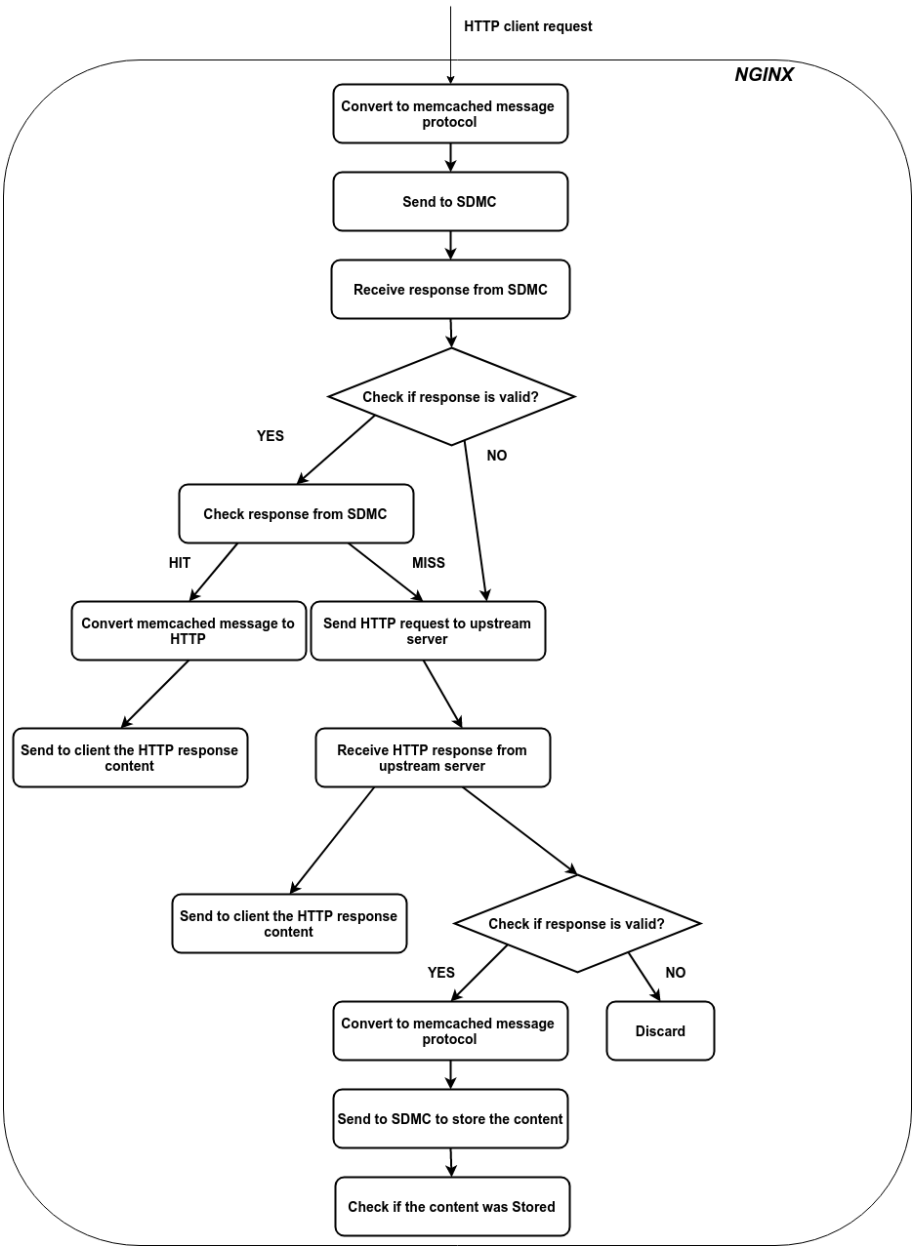


Figure 4.3: Nginx flowchart

The DSMC is located in the localhost, listening in port "11212". After Nginx received the data, it has to check if the content received is valid, which means that it has to check if the key retrieved from DSMC matches with the key that was requested. Moreover, it has to check if the field message length in a Memcached message is the length of the Memcached body.

After verifying if the message is valid and the data received is a hit, it considers that the content is in a global cache. Then Nginx has to create an HTTP response with the data from the Memcached message received and send it immediately to the consumer. On the other hand, if the message received is a miss, the content is not in the global cache, or if the data of the message is not valid, then the Nginx is in charge to request the content by HTTP

in an upstream server, the aggregator. When the upstream server responds to the request, the content is sent to the consumer which requested the content and, at the same time, if the response is valid, it is stored in a cache, otherwise it is discarded. To store the content in an edge cache, it has to create the TCP Memcached protocol message "SET" with the key defined by the path and arguments from the HTTP request, and send it to the DSMC module. Then, the DSMC has to send back a message if the content was successfully stored.

4.2.2 Prefetching

The Prefetching is a mechanism that pre-populates the cache before consumers request the content. This mechanism enables the improvement of the QoE of the consumers, since the content is closer to the user before it is required. The base prefecthing has been developed in the NAP group of IT Aveiro [33][12]. However, it was developed for a different purpose and scenarios, and without integration with the DSMC module. Therefore, in the framework of this thesis prefetching has been integrated with the DSMC module, and it has been modified to improve the QoE of the consumers in the train scenario.

The prefetching flowchart is shown in Figure 4.4.

The prefetching module listens to the HTTP packets on port 80, which is the port where the consumers request the content in each edge cache. When a user does a HTTP request to an edge cache, the HTTP request is observed to analyze which content is being requested by the clients. Since a device from a consumer can have several applications, more applications can communicate with the edge cache via HTTP, rather than video; then, the prefetching mechanism needs to validate the packages that were analyzed in order to select the video packets.

The video client chunks in this work have the following format:

"/ElephantsDreamvideoNumber.ism/QualityLevels(qualityLevels)/Fragments(video=chunk"

"Number)"

All requests that do not follow this pattern will be discarded as they are not valid for OTT content.

After verifying if this pattern is followed, then the quality of the video is extracted, as well as the ID, videoNumber and the chunk number. With this information it is possible to estimate the next chunk video that will be requested by the consumers. However, in a real scenario the quality of the next chunk may vary due to the network limitations of the clients. Therefore, it is a great challenge to predict the quality of the next chunk. After extracting the parameters of the customers' requests, the next chunks can be calculated, by the increment of the chunk number by 2, which is the duration of each chunk, 2 seconds of video. Moreover, to calculate the next chunks' quality is difficult; however, in this work it is used the quality that has been requested in the previous chunks. The available qualities are eight and were described in the previous chapter.

After calculating the next chunks, the Prefetching has to make requests to the Nginx to populate the cache. The requests are done via HTTP and have the following format:

"http://127.0.0.1/ElephantsDreamvideoNumber.ism/QualityLevels(qualityLevels)/Frag-

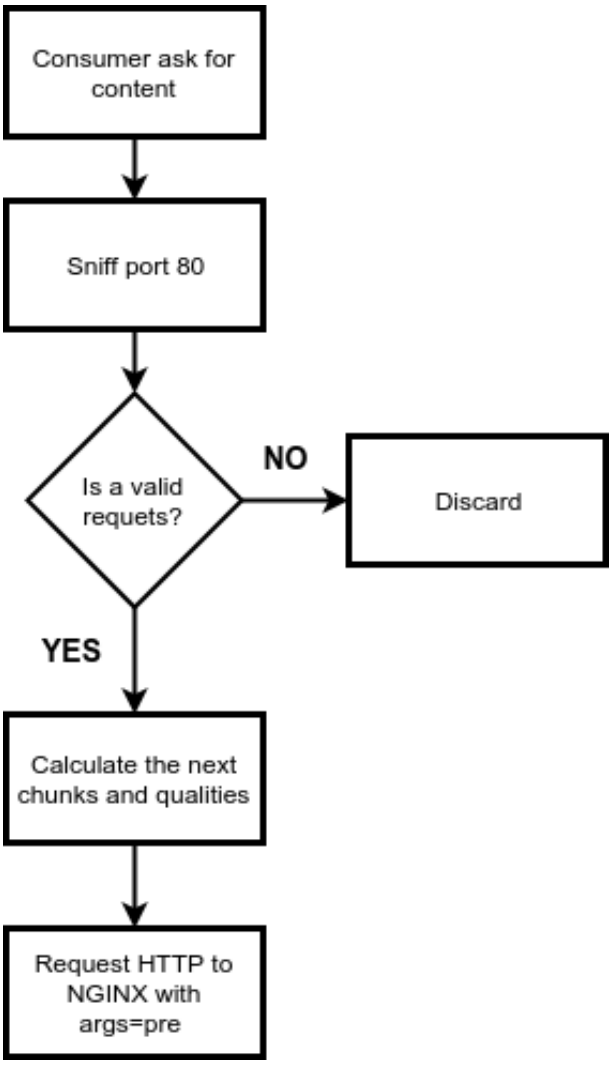


Figure 4.4: Prefetching flowchart

”ments(videochunkNumber)?pre ”

The difference between the prefetching requests and the client requests is the ”pre” argument. This is used with the distributed cache to reduce the overhead in the network: if the Prefetching is a hit, it will be addressed in the DSMC module. Therefore, the prefetching module is then responsible for predicting the next quality and pre-populating the cache with the content, before the consumers requested that content.

4.2.3 DSMC

The DSMC is composed of 4 blocks: Broadcast, Update, OTT-Distribute Content and OTT- Aware Mechanism. The Broadcast and Update modules are implemented using UDP packets. The UDP was chosen because it is simple to use and mainly allows to broadcast the information for all nodes with only one transmission. However, the transmission may be unreliable or un-ordered due to network conditions. Then, the Broadcast module is responsible

to send the information of each edge cache about its existence, by the source IP address, and the size of its cache.

The OTT-Distribute Content and OTT-Aware Mechanism modules are implemented using TCP packets with Memcached protocol. The OTT-Distribute Content receives the TCP Memcached message from Nginx. These messages are of two types: "GET" to retrieve the content and "SET" to cache the content. Then, the OTT-Distribute Content has to distribute the content by neighbours and by itself, through the use of consistent hashing algorithm. Moreover, this module has to differentiate the content that is from prefetching or from a client. It is possible to differentiate these requests, since prefetching presents a different key with the "pre" at the end of the key.

The OTT-Aware Mechanism receives the TCP Memcached message from the OTT-Distribute Content. Its aim is to implement a mechanism to lock a given request chunk that was recently requested and was a miss. Thereafter, the request has to wait until that content is stored in cache. To accomplish that, it must have a list of the requests recently made that were a miss: when there are the same requests by the several clients in different edge caches, all of them will be miss, and subsequently unnecessarily overload the origin, so the requests have to wait for the contents to be stored. The edge cache responsible for this lock is the one that is in charge of the given chunk/content. The edge cache that is in charge of fetching content from the origin is the first to verify if the chunk is not in cache, while the others have to wait for this cache to include the content. This effect happens with a huge impact in the Live scenarios, and with some probability in VoD. Therefore, it is possible to eliminate these cases. After verifying that the chunk ID is not in the list and the request is a "GET", the edge cache sends the Memcached message to the Cachelot. If it is a "SET", which means to cache the content, the chunk ID is removed from the list, since the content is already stored by the "SET" message. Moreover, this module has to verify if a "GET" message is from prefetching, since if a "GET" message is a hit, only a few bytes are sent to the OTT-Distributed Content specifying that it was a hit, instead of sending the content from that chunk; then, it is possible to reduce the overhead in the network generated by prefetching.

4.2.3.1 Broadcast

The Broadcast block needs to send information about the cache size. Figure 4.5 presents the Broadcast flowchart.

This module has to create an UDP packet, with data of memory size reserved for the cache of each edge cache. Thereafter, it is created an UDP socket in port '54545' to send this information to the neighbours' edge caches. Every 200ms the packet is broadcast; in this way it is possible to warn that a certain edge cache is available to store and serve the content. If the edge cache is shutdown, this will not send the broadcast packets and will no longer be used to cache the content, as described in the following module, Update module.

4.2.3.2 Update

The Update module flowchart is presented in figure 4.6.

In this block a UDP server listening in port "54545" is created, that receives the broadcast packets from the several edge caches. After receiving the broadcast packets, it reads them and obtains the IP address source of the package and the size of the cache of a particular edge cache. If the source IP address is not in the temporary list, it is inserted in the temporary

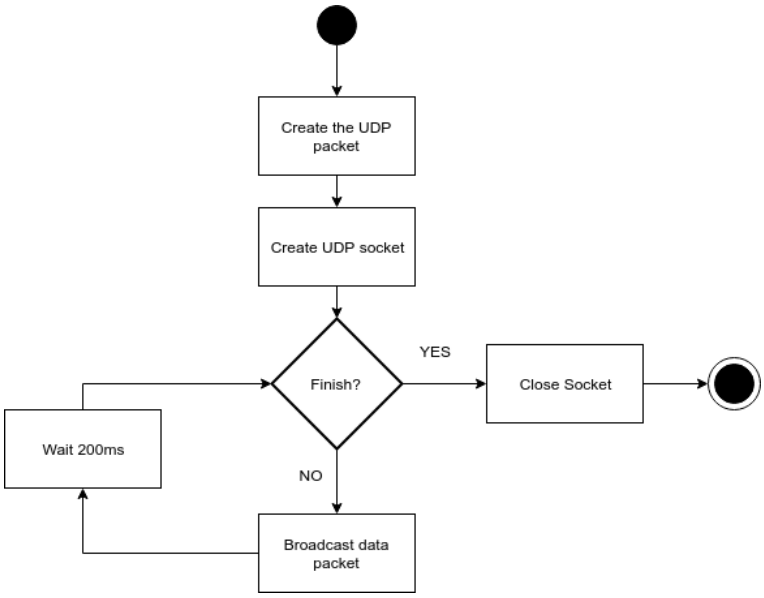


Figure 4.5: Broadcast flowchart

list, updating it. At the end of 5 seconds, the refresh of the listOfNodes is done if the listOfNodes contains different IPs from the temporary list. If the listOfNOdes is different from the temporary list, a copy of the list is done to the listOfNodes. The listOfNodes are the edge caches that are in the ring of the Consistent Hashing algorithm. Then, the ring is updated. Thereafter, the temporary list is emptied to receive new updates. Otherwise, if the listOfNodes contains the same IPs of the edge caches of the temporary list, it is emptied to receive new updates and the program will listen new messages on the network again.

The ring update is done by adding or removing the virtual nodes from each edge cache. The number of virtual nodes for an edge cache varies depending on the cache size. For instance, if the cache size is 512 MB, then in the ring there are 150 virtual nodes for that edge cache. However, if an edge cache has a higher cache size, the number of virtual nodes is higher. For example, if an edge cache is twice the default size, which means, 1024 MB, the number of virtual nodes for this edge cache is 300. Thus, it must have a weighted distribution, taking advantage of the resources available of each edge cache.

4.2.3.3 OTT-Distribute Content

The OTT-Distribute Content aims to distribute the content among the several edge caches. The OTT-Distribute Content flowchart is presented in Figure 4.7.

This module receives the TCP Memcached messages from Nginx. To receive the TCP Memcached messages, a TCP server is created, in port 11212 in each edge cache. Since this TCP server has to deal with a large amount of data and handle many requests, it is necessary to create threads to handle each request (a single thread is not suitable if each request takes a long time to complete, generating latencies in requests to process). Thus, the socket must be bound to an address and listen for new connections. When the server accepts new incoming requests, it returns a tuple with a new socket object and an address. The new socket object is usable to send and receive data, and the address is the address bound to the socket on the

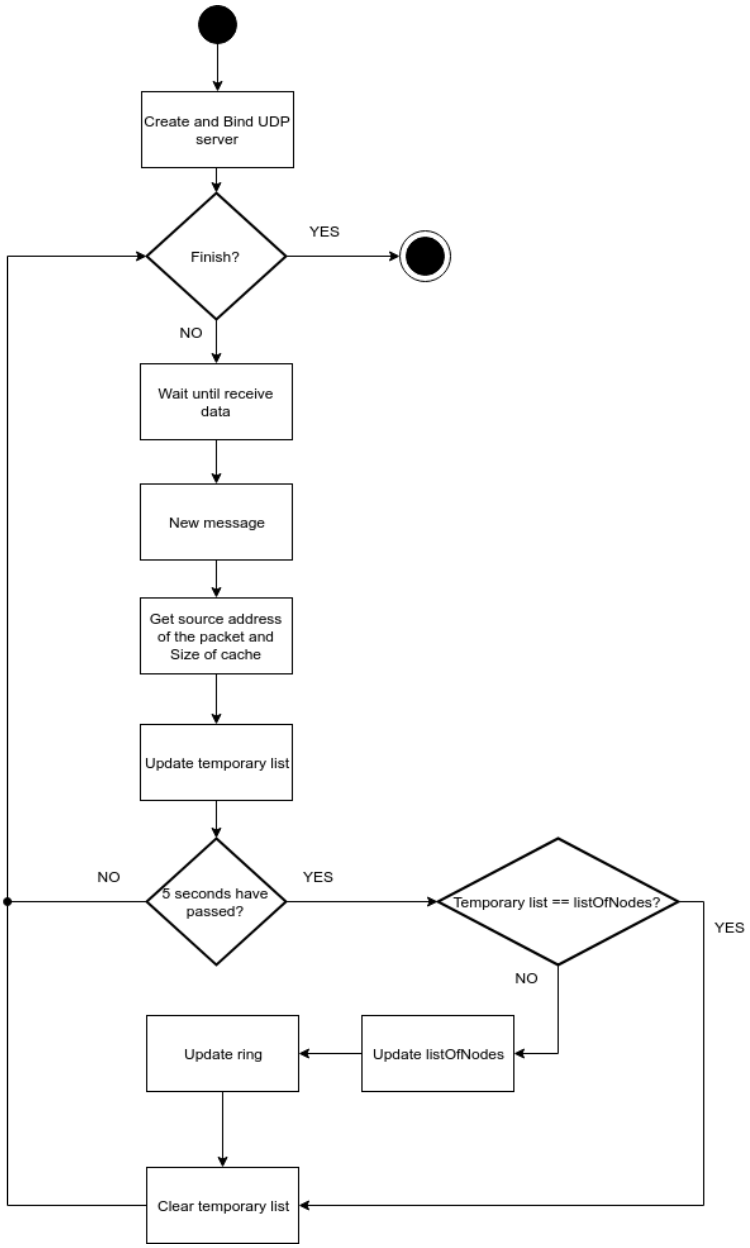


Figure 4.6: Update flowchart

other end of the connection. In order for a server to be able to process several requests at the same time, when it accepts new incoming requests, it needs to pass the new socket object and the address for a thread to handle each request. Thus, it is possible to handle several requests.

When the OTT-Distribute Content receives TCP Memcached messages from Nginx, it has to verify the message type: if it is a "SET", to store the content, or a "GET", to retrieve the content. Moreover, if the message is a "SET", it has to extract the key from that message and check if the key is a prefetching request. This analysis is done through the key field of the Memcached message, since the key from Prefecthing ends with "pre", as illustrated below.

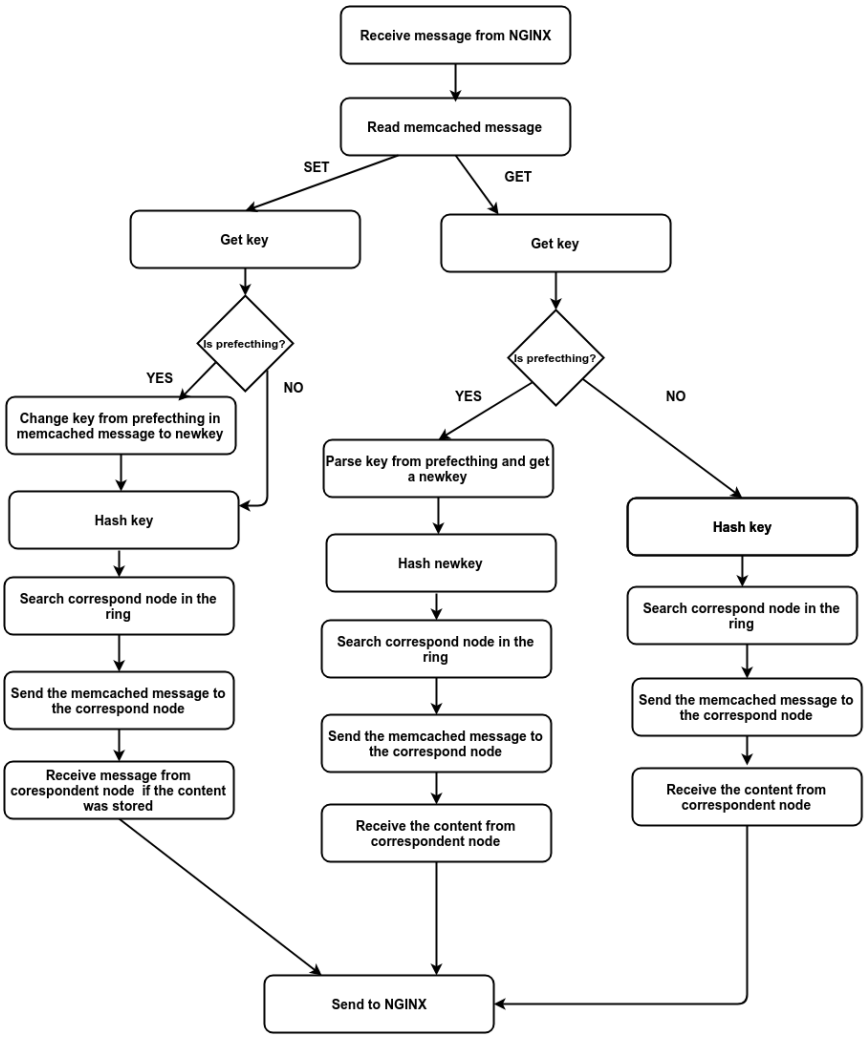


Figure 4.7: OTT-Distribute Content flowchart

An example of a client key:

"/ElephantsDream_videoNumber.ism/QualityLevels(qualityLevels)/Fragments(video=chunk"

"Number)"

An example of a Prefecthing key:

"/ElephantsDream_videoNumber.ism/QualityLevels(qualityLevels)/Fragments(video=chunk"

"Number)pre"

If the message has a key from prefetching, this module has to remove the part of the "pre" at the end of the key. Thereafter, it needs to change the key to a new key in the Memcached message to be consistent with the key of the consumers. Thus, the decision to a particular content chunk will be different between the prefecthing and the clients. Then, the new key is hashed and is searched in the consistent hashing ring, which returns the IP of the edge cache that is in charge to cache the content identified by that key. The next step is to send the "SET" message to the OTT-Aware Mechanism module of the correspondent edge cache. Then, the OTT-Aware Mechanism module responds if the request was successfully stored and the OTT-Distribute Content sends this message to Nginx, in order to inform that the chunk is successfully stored.

If the key is from a client, then the key is simply hashed and searched in the ring. The consistent hashing returns the IP of the edge cache that is responsible to cache the content identified with this key. The next step is to send the "SET" message to the OTT-Aware Mechanism module of the correspondent edge cache. Then, the OTT-Aware Mechanism module responds if the request was successfully stored and the OTT-Distribute Content sends this message to Nginx, in order to inform that the chunk is successfully stored.

On the other hand, if the message from Nginx is a "GET" message, it is necessary to verify if the key is from prefetching. If it is, the previous steps apply. The next step is to send the "GET" message to the OTT-Aware Mechanism module of the correspondent edge cache. The OTT-Aware Mechanism then returns only a small message, which indicates that it was a hit when the content is in global cache, or returns only a small message indicating it was Miss if the content is not in the global cache. Thereafter, it is necessary to send this message to Nginx. If the "GET" message is from a client, the key is hashed and it is searched in the consistent hashing ring, which returns the IP of the edge cache responsible for that content chunk. The next step is to send to the corresponding edge cache that message to the OTT-Aware Mechanism module. Then, the OTT-Aware Mechanism returns the content, if it is a hit, or returns only a small message containing it was Miss in the case of miss and the content is not in the cache. At the end, the messages received from the OTT-Aware mechanism will be sent to the Nginx, by the OTT-Distribute Content.

4.2.3.4 OTT-Aware Mechanism

The OTT-Aware Mechanism aims to do the management of the incoming requests from the several OTT-Distribute Content modules in the several edge caches. This module has a list of the requests recently received that were a miss.

This module implements a lock mechanism, because if the same content request is done by the several clients, all of them will be a miss, and subsequently unnecessarily overload the aggregator. Thus, the requests have to wait for the content to be stored. The edge cache responsible for this lock is the one that is in charge of the particular content chunk.

The edge cache that is in charge of fetching the content from the aggregator is the first to verify that the chunk is not in cache, while the others have to wait for it to put the content. This effect happens with a huge impact on the Live scenarios, but it also happens with some probability in VoD. In this way, it is possible to surpass these problems.

The OTT-Aware Mechanism flowchart is presented in Figure 4.8.

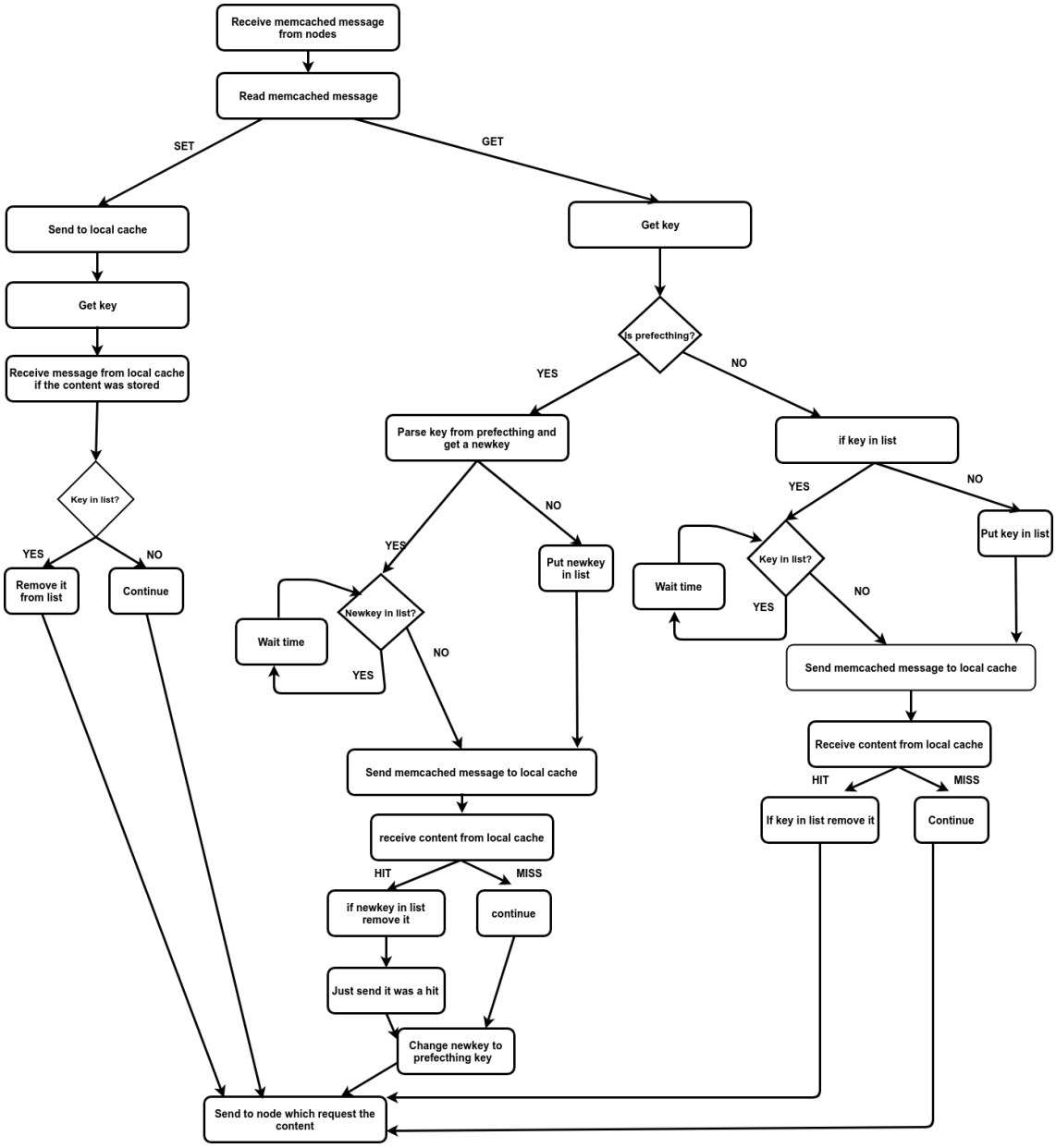


Figure 4.8: OTT-Aware Mechanism

The OTT-Aware Mechanism receives the TCP Memcached messages from the several OTT-Distribute Content edge caches. To accomplish this purpose, a TCP server is created on port '11213' in each edge cache. Since this TCP server has to deal with a large amount of data and handle many requests, it is necessary to create threads to handle each request.

The OTT-Aware Mechanism receives the TCP Memcached messages from the OTT-Distribute Content module and it has to verify the message type, which means, a "SET", to store the content, or a "GET", to retrieve the content. If the message is a "SET", this module sends the Memcached message "SET" to the local cache, Cachelot, and extracts the key from the message. Then, the local cache responds if the request was stored successfully. Thus, this module has to check if the key is in a list. If it is in a list, it is removed from there; if not, it continues execution. At the end, it sends to the OTT-Distribute Content module from the edge cache that sent the content to store, the information that the content was successfully stored.

On the other hand, if the message is a "GET", this module has to verify if the request is from the prefetching. Then, if the new key is in the list, the request has to wait until the key is no longer there. If the new key is not in the list, it is inserted in it. Then, the "GET" message is sent to Cachelot, which answers with the content chunk if it is a hit, and the new key is removed from the list. Thereafter, only a message is sent to the OTT-Distribute Content module from the edge cache that requested the particular content chunk, indicating that it was a hit with the key from prefetching. It is then possible to reduce the overhead created by the prefetching. Otherwise, it is a miss, the key is changed back to the key from the prefetching and it is sent to the OTT-Distribute Content that is in the edge cache that request a particular content chunk. Also, the key is inserted in the list.

If the message is a "GET" and is not from the prefetching, but from a client, it is checked if the key of this request is in the list. In case the key is in the list, then the request has to wait. The "GET" message is sent to Cachelot, which answers if it was a hit or miss: if it is a hit and if the key is in the list, then it is removed from the list and the content is sent to the OTT-Distribute Content from the edge cache that requested a particular chunk; otherwise, it is a miss, the same response from Cachelot is sent to the OTT-Distribute Content module from the edge cache that requested a particular chunk.

4.3 Chapter Considerations

This chapter presented the implementation of the solution proposed in Chapter 3.

First, it described the clients that are watching videos like real customers smooth streams with different popularity. Also, it is presented the edge caches in a technical and practical way, their functionalities and implementation.

The next chapter describes how the software is configured to test the several testbeds for the different scenarios used to obtain results in a controlled environment.

Chapter 5

Integration and Evaluation

This chapter presents the integration of the solutions adopted and used in order to evaluate the proposed architecture, in a real testbed using Virtual Machines with Ubuntu 14 installed. It presents the several tests comprising all scenarios developed in this assessment process. Moreover, the obtained results are presented and discussed. Thus, this section is organized as follows:

- section 5.1: presents a brief description of the hardware and Operating System of each virtual machine used.
- section 5.2: presents the main configurations and the required scripts used to measure and calculate the performance metrics.
- section 5.3: presents the evaluation of each scenario and model, as well as the discussion of the results.
- sections 5.4: presents the chapter conclusions.

5.1 Hardware and Operating System

This section describes the hardware and operating system of consumers, origin, edge caches and aggregator used to implement the several scenarios and models, with the architecture proposed in chapter 3 and its implementation detailed in chapter 4. Thus, it is important to note that these entities run in virtual machines. These virtual machines are running on two physical machines with 32 logical cores and 32GigaByte (GB) of RAM each one. However, one physical machine is used with all available resources, and another physical machine only uses 12 logical cores and 16GB of RAM, since there are more virtual machines running on the first physical machine.

It is used the VMware[66] hypervisor to deploy virtual machines.

Table 5.1 illustrates the hardware specifications of virtual machines of consumers, origin, edge caches, and aggregator.

Table 5.2 shows the software specifications in the different virtual machines.

The used origin server is the Microsoft Windows Server 2012 [67], with Microsoft Smooth Streaming [68] installed.

The aggregator, edge caches and consumers use Ubuntu 14.04.5 LTS [69].

Table 5.1: Hardware specifications

	CPU	Memory
Origin Server	2 cores, 5GigaHertz (GHz)	6GB
Aggregator	2 cores, 5GHz	2GB
Edge caches	2 cores, 5GHz	2GB
Consumer_A_B	6 cores, 15GHz	6GB
Consumer_C	12 cores, 30GHz	16GB
Location: Instituto de Telecomunicações of Aveiro		

Table 5.2: Operating System

	Operating System
Origin Server	Microsoft Windows Server 2012
Aggregator	Ubuntu 14.04.5 LTS
Edge caches	Ubuntu 14.04.5 LTS
Consumers	Ubuntu 14.04.5 LTS
Location: Instituto de Telecomunicações of Aveiro	

5.2 Configurations and Scripts required

This section presents the main configurations and the scripts developed to run the different software, as well as to measure and calculate the different performance metrics of the scenarios and models used in this work.

5.2.1 Configuration of the several software

In this subsection it is presented and explained the different configurations of the different software used, such as Nginx [70], Cachelot, Probes and Netem [71]. The most relevant settings are presented in the following sub-subsections.

5.2.1.1 Nginx Configuration for edge caches

A messaging protocol is used in order to establish communication between Nginx, DSMC and Cachelot. Thus, the Memcached protocol is used since it is similar for both modules. In this case, it is used the following modules that are not distributed with the Nginx source code:

- ngx_devel_kit[72]: this module is used to extend the functionality of Nginx, it is working as the basis for other Nginx modules.
- set_misc_nginx_module[73]: this module defines internal variables used in the protocol Memcached, useful to define some parameters, like key, expiration time, etc.
- memc_nginx_module[74]: this module implements the memcached TCP protocol that communicates with the DSMC, supporting several commands but mainly to store and get content.

- srcache_nginx_module[75]: this module serves to make the transparent cache, which means, if a given content is in the global cache, verified by the DSMC, then it returns to the client. Otherwise, if is not in cache, it is fetched on the upstream servers/aggregator. When the upstream server/aggregator answers with the content, the Nginx has the job to deliver the content to the client and, at the same time, to make the content storage in the global cache, with the help of the DSMC.

Since these modules are not distributed with the Nginx source code, they have been installed and added to the installation. The version nginx-1.9.15 is used since it is the stable version for all the modules used.

The Nginx configuration file for the edge caches is shown in figure 5.1. Some comments are made to understand the purpose of each line and directive.

Overall, the main configuration directives in the Nginx configuration file for the edge caches are:

- set: define a variable with its value;
- srcache_fecth: fetch the content in the argument;
- srcache_store: store the content in the argument;
- srcache_store_max_size: is used to define the maximum number of bytes to store per each request in srcache module, 0 means no maximum;
- proxy_pass: defines the upstream server or aggregator;
- server: indicates the IP and port of the DSMC module;
- client_max_body_size: sets the maximum number of bytes to store per each request in memc module;
- memc_pass: defines the IP and port of a server;
- memc_connect_timeout: sets the connections timeout;
- memc_send_timeout: defines the timeout of a send message to DSMC;
- memc_read_timeout: defines the timeout of a read message from DSMC;
- memc_ignore_client_abort: ignores the closed connections of the DSMC;
- log_format: writes the cache logs, Hits and Misses of all requests.

For more information, the documentation is accessible at the following sites[70][72][73][74][75].

5.2.1.2 Nginx Configuration for aggregator

In the aggregator's case, it presents an exclusive local cache, so the built-in Nginx cache is used to. Therefore, Nginx is configured as a reverse proxy with cache capability. Figure 5.2 presents the configuration of Nginx to the aggregator.

In this case, the main Nginx configuration directives are:

```
http {

        log_format rt_cache '$remote_addr - $upstream_cache_status [$time_local]  '
        '"$request" $srcache_fetch_status $body_bytes_sent '
        '"$http_referer" "$http_user_agent"';

        access_log   /home/user/FINAL/example.com.access.log rt_cache;


    upstream sdmc {
        server 127.0.0.1:11212;
    }


    server{

        location = /memc {
            internal;
            client_max_body_size 2M;
            memc_connect_timeout 10000ms;
            memc_send_timeout 10000ms;
            memc_read_timeout 10000ms;
            memc_ignore_client_abort on;

            set $memc_key $query_string;
            set $memc_exptime 0;

            memc_pass sdmc;
        }

        location / {
                set $key $uri$args;
                srcache_fetch GET /memc $key;
                srcache_store PUT /memc $key;
                srcache_store_statuses 200 301 302;
                srcache_store_max_size 0;
                proxy_pass http://192.168.122.121:80$request_uri;
            }
    }

}
```

Figure 5.1: Nginx configuration, edge caches

- proxy_cache_path: indicates the cache path to store the content;
- proxy_pass: indicates the upstream server, in this case origin server;
- log_format: writes the cache logs, Hits and Misses of all requests.

For more information, the documentation is accessible at the following site [70].

```
user nobody nogroup;
worker_processes 1;
events{
worker_connections 1024;
}

http{
    proxy_cache_path /var/cache/nginx levels=1:2   keys_zone=my-cache:1024m
    inactive=24h  max_size=1g;

    log_format rt_cache '$remote_addr - $upstream_cache_status [$time_local] '
            '"$request" $status $body_bytes_sent '
            '"$http_referer" "$http_user_agent"';

     access_log   /home/user/FINAL/example.com.access.log rt_cache;

    server
    {
        listen 80;
        location /
        {
            proxy_pass http://192.168.122.122;
            proxy_cache my-cache;
            proxy_cache_valid 200 302 60m;
        }
    }

}
```

Figure 5.2: Nginx configuration, aggregator

5.2.1.3 Cachelot Configuration

The cachelot is used as a local cache only for the edge caches. It has to be configured to have the best cache management, taking into account the content it has to store. To accomplish that it is executed with the following command:

./cachelotd -m512 -p11214 -P2

The used arguments are:

- -p: indicates the TCP port where the Cachelot is open;
- -P: is the size of the page in MB; it is used to have a better cache performance, since smaller size pages leads to more accurate evictions, although the page size limits the storage of the item to store, since the largest item to store must be less than or equal to the maximum page size;

- -m: is the size of the local cache in MB.

For more information, the documentation is accessible at the following site[57].

5.2.1.4 Probes

Probe simulates a real video content consumer that, in addition to consuming the content, it also provides metrics regarding the consuming process. The main metric is the QoE. In order for the probe to be used, it is necessary to pass arguments regarding the content to be consumed and the file where to store the measured metrics. To this end, as an example, the following command is executed:

mono SmoothStreamingProbe.exe -u = http://edge_cache_ip/development/

smoothstreaming/ElephantsDream_idVideo.ism/Manifest -o = out_id_client.json

The used arguments are:

- -u: indicates the URL of manifest that indicates which server provides the required video;
- edge_cache_ip: indicates the edge cache that serves this client;
- idVideo: indicates the identifier of the video;
- -o: is used to output the QoE metric in a file with name given by the argument out_id_client.json;
- out_id_client: is used to indicate the id of the client.

5.2.1.5 Netem

Netem provides a network emulation functionality to introduce latency, delay, packet loss, bandwidth limit, etc. It is used on the assessment of scenarios comprising network limitations. In this work the following commands were used.

sudo tc qdisc add dev eth1 root handle 1: tbf rate bandwithMbit burst 1600 limit 3000

sudo tc qdisc add dev eth1 parent 1:1 handle 10: netem delay mean jitter distribution normal loss value%

The used arguments are:

- bandwidthMbit: limits the available bandwidth;
- eth1: is the network interface to add the limitation;
- delay mean jitter: adds latency with a predefined mean value and a jitter value;
- distribution normal: is used to set a variation in network delay based on a normal distribution;
- loss value: adds the packet loss in percentage.

For more information, the documentation is accessible at the following site[71].

5.2.2 Metrics

To evaluate the results of the tests and to obtain reliable comparisons, several metrics are measured during the several exhaustive tests. These metrics are important to monitor the behavior of the several tests and are described in the following topics. Moreover, these metrics are important to evaluate the performance of the several architectural models in the different scenarios.

CPU Usage

The metric CPU usage is the capacity by which the processor is currently working. In other words, it is the CPU time as a percentage of CPU capacity. A computer can vary this metric depending on how many tasks it has to handle. However, when the CPU usage reaches the 100% an overload can occur, since there is no more capacity to handle all the tasks. Thus, regarding this metric, in order to have reliable results, it is necessary that the virtual machines do not reach this value. The CPU usage information can be fetched in many ways. In this case, the psutil is used, which is a cross-platform library for system information retrieval. The CPU usage can be calculated by subtracting the idle CPU time from the total CPU time, divided by the total CPU time, as can be see in Equation 5.1. The total and idle times are measured with a sampling period of 1 second.

$$\text{CPU}usage = \frac{TOTALtime - IDLEtime}{TOTALtime} * 100 \tag{5.1}$$

Cache Metrics

The main metrics that evaluate the cache performance is the hit and miss ratios. In order to calculate the hit and miss ratios, Equations 5.2 and 5.3 are provided.

$$Hitratio(\%) = \frac{TotalHits}{TotalHits + TotalMisses} * 100 \tag{5.2}$$

$$Missratio(\%) = \frac{TotalMisses}{TotalHits + TotalMisses} * 100 \tag{5.3}$$

The hits and misses are provided by the Nginx logs explained in subsections 5.2.1.2 5.2.1.1

Consumed Bandwidth

The consumed bandwidth is the amount of data transferred on a given interface. This can be used on two metrics, download and upload consumed bandwidth. Download is the amount of data transferred from the outside to the computer being measured. Upload is the amount of data transferred from the computer measure to the outside. The download and upload consumed bandwidth information can be retrieved in many ways. In this work, the psutil tool is also used. The download and upload consumed bandwidth can be calculated by subtracting the number of bytes received or sent in a previous time instant from the current time instant, divided by the elapsed time between the sampling time period, according to Equations 5.4 and 5.5, respectively. The measured parameters, bytes sent and bytes received, are measured with a sampling period of 1 second.

$$Download\_ratio = \frac{(receivedbytes(t) - receivedbytes(t - samplingtime)) * 8}{sampling\_time} \tag{5.4}$$

$$Upload\_ratio = \frac{(sentbytes(t) - sentbytes(t - samplingtime)) * 8}{sampling\_time} \tag{5.5}$$

QoE

The quality of experience is obtained through the probes as discussed in section 4.1.

5.2.3 Support scripts

To support the evaluation of the several scenarios, model scripts are developed. The performed scripts have several purposes, as shown in the following points:

- scripts CPU usage and consumed bandwidth: These scripts are used to measure the consumed bandwidth and CPU usage, in order to verify if the obtained results are valid;
- scripts to compose the test scenario: They are used to launch the several programs in order to have a full system;
- scripts to generate final results: These scripts are used to show the data in a clear way.

More detailed information about the scripts is presented in the following subsections.

5.2.3.1 Script CPU usage and Consumed Bandwidth

The flowchart in Figure 5.3 illustrates the process for the measurement of the CPU usage and the consumed bandwidth, as well as the process to store the information in a log file. The sampling time is 1 second, to each measurement. This script is in charge to fetch the CPU

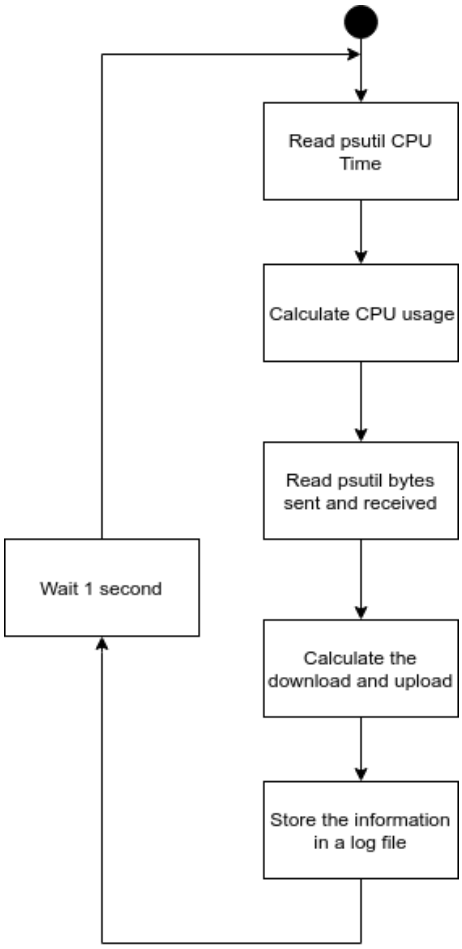


Figure 5.3: CPU usage and Consumed Bandwidth Flow Chart

information using the psutil library. Then, it calculates the CPU usage by the Equation 5.1. Thereafter, it is used again the psutil library to retrieve the information about the bytes sent and received in a particular network interface. The download an upload ratio is calculated by the Equations 5.4 5.5. The calculated metrics are stored in a file and the script waits 1 second, which is the sampling time. Then, it repeats the whole process again.

5.2.3.2 Scripts to compose the test scenario

To perform the evaluation of the several scenarios and to test the different models, these scripts are developed to launch the several scripts as well as the software and programs developed. These scripts can be divided in three and are described in the following topics.

Client side

The flowchart illustrated in Figure 5.4 represents the flowchart of the clients. This script runs on clients' virtual machines. So, it is in charge of launching a given number of probes, which is given by argument. This number of probes is the number of customers, and a customer is a probe. This script, before launching the probes, it has to create a consumer profile, which means, it has to define which video to watch, given by the zipf's law function (defining the video popularity among the clients), as well as, which edge cache is in charge of providing the content for the clients.

The clients in the test runs are evenly distributed across the several edge caches. As can be seen in the flowchart, after the consumer profile is created, the probe that simulates this client is launched; this script continues iteratively until it reaches the desired number of clients. Then, the script has to wait for the clients to finish to consume all the content. After the clients consume the content, several probe logs are generated, which are used to analyze the QoE of the several clients.

Edge caches side

The flowchart illustrated in figure 5.5 represents the flow chart of the edge caches. Before starting the script, it is necessary that the Nginx configuration file is configured. Also, before the script is executed, the previous test data must be removed before proceeding with the script. This script is in charge of launching the Nginx, DSMC and Cachelot, the essential applications for the operation of the architecture. However, it also launches the script to measure the bandwidth consumed and the CPU usage script. At the end of the test, after the clients consume all the content, they have to kill all the processes launched. Moreover, after the tests, it is necessary to transfer all the created log files to another folder, since each iteration is erased.

Aggregator side

The flowchart illustrated in figure 5.6 represents the flow chart of the aggregator. Before starting the script, it is necessary that the Nginx configuration file is configured. Also, before the script is executed, the previous test data must be removed before proceeding with the script. In addition to the log data that has to be removed before each iteration, the Nginx cache has to be emptied. This script is in charge of launching Nginx and the scripts to measure the bandwidth consumed and the CPU usage script. At the end of the test, after the clients consume all the content, the script has to kill all the processes launched. After the tests, it is necessary to transfer all the created log files to another folder, since in each iteration it is erased.

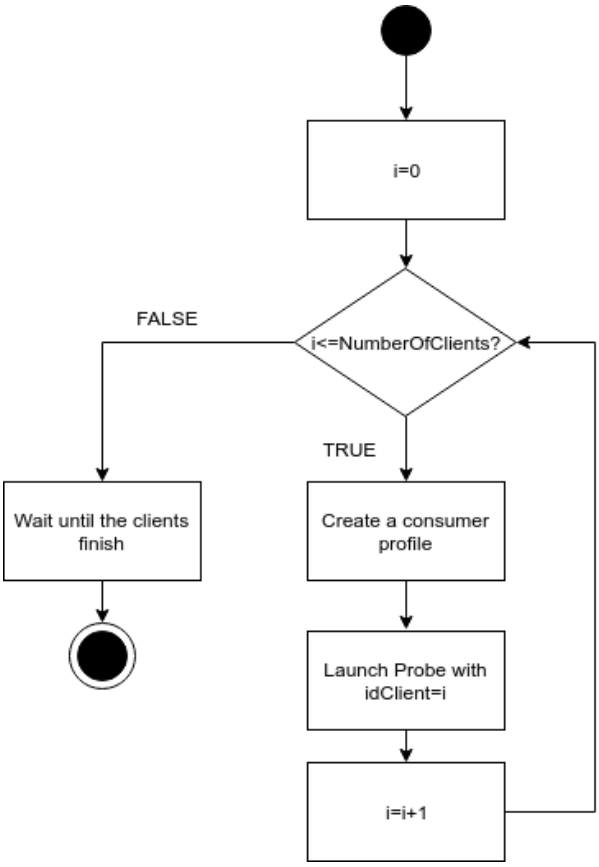


Figure 5.4: Clients Flow Chart

5.2.3.3 Scripts to generate final results

These scripts are executed to gather and analyse the results. They are responsible to read the generated logs for each scenario and calculate the presented evaluation metrics, such as, consumed bandwidth, performance metrics, CPU usage and QoE. Then, at the end, this data is presented in graphics created using python, showing the results in graph plots with confidence intervals.

5.3 Evaluation

In order to evaluate the proposed architecture, it is necessary to implement a testbed composed of the several proposed modules, and assess its behaviour comprising different caching models on several video content distribution scenarios, using caching metrics such as hit and miss ratio and consumed bandwidth, CPU usage and QoE.

The content originator server is a Windows Server running Internet Information Services (IIS) Smooth Streaming. It has all the content of all the videos. The aggregator is a server that has a local cache and the ability to reverse proxy. The edge caches have different configurations depending on the model under evaluation, that is, the decentralized model, the distributed model and the centralized model.

The decentralized caching model is characterized by the traditional CDNs, where each edge

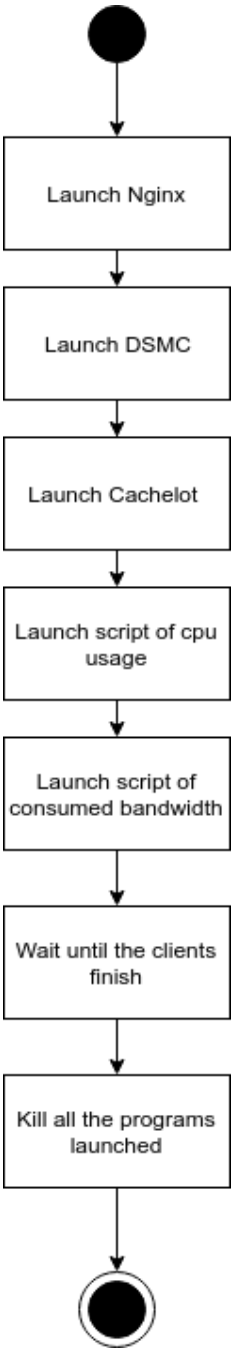


Figure 5.5: Edge Caches Flow Chart

cache has a local cache that is not shared by other edge cache nodes. On other hand, on the distributed model, which comprises the proposed architecture, the edge caches communicate between each other, therefore sharing and providing the larger cache among them.

Therefore, on the point of view of the requests made by the content consumers, and the edge caches, the aforementioned architectural models can be explained as follows:

- Centralized model: where all consumer requests are made to the proxy which forwards

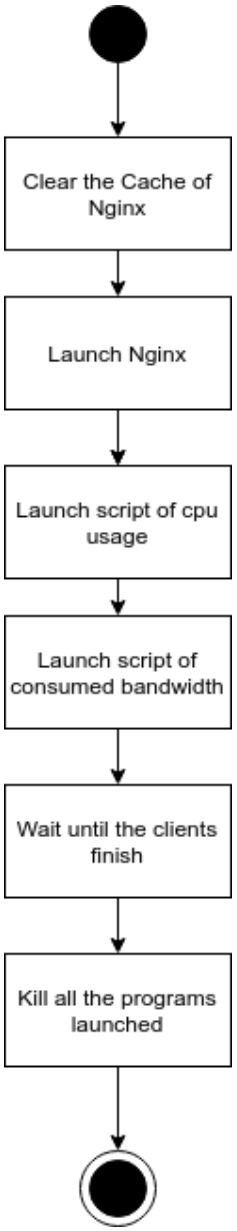


Figure 5.6: Aggregator Flow Chart

to the aggregator that has a local cache. If the content is not in the aggregator, it has the function to make the requests to the origin, which has all the content required by the consumers, as depicted in Figure 5.7. In this model there are no edge caches involved and it is used as a baseline to comparison.

- Decentralized model: In the decentralized model the consumers perform the content requests to the edge caches that have a local cache. If the content is not in the edge local cache, this one has to fetch the content in the aggregator that presents a local cache; if it does not have the content, it searches the content in the origin, as depicted in Figure 5.8.

- Distributed model: This is the proposed architecture model. The consumers request the content to the edges caches. However, the cache is shared among them, thus providing and using a global cache. This allows content not to be repeated among the edge caches. If the content is not found in the shared cache of the edge caches, one of the edge cache fetches in the aggregator. Then, if it does not have the content, it fetches in the origin, as depicted in Figure 5.9.

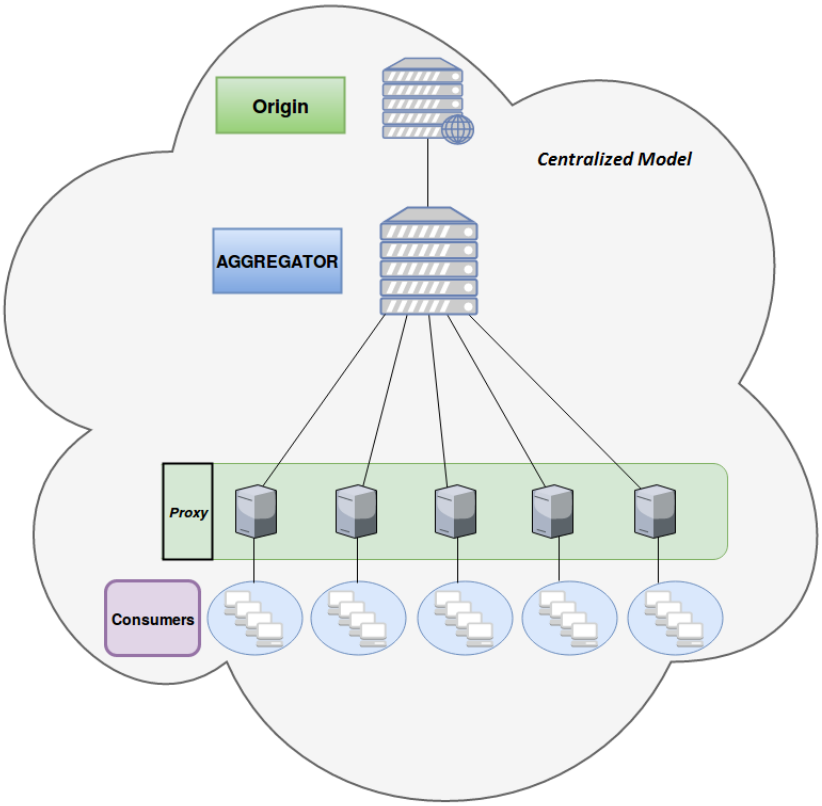


Figure 5.7: Centralized Model

Specific aspects of the different models

On the centralized model, all the traffic originated by the consumers is handled by the aggregator, because the proxy only performs the forwarding of the clients requests. This type of scenario causes a higher use of resources, such as CPU and bandwidth in the aggregator, since it has to handle all clients' requests. In the decentralized model, with the addition of an extra tier of caches, the edge caches, that provide the content to the consumers, CPU and bandwidth resources are saved in the aggregator, since most of the requests are handled by the edge cache, which avoids to request the aggregator so often. In the distributed model, which provides a distributed cache model, i.e, the edge caches in addition to having a local cache, they are shared among them, providing a global cache, allowing a higher cache size. This approach allows to save bandwidth and CPU in the aggregator side, since the caches of the

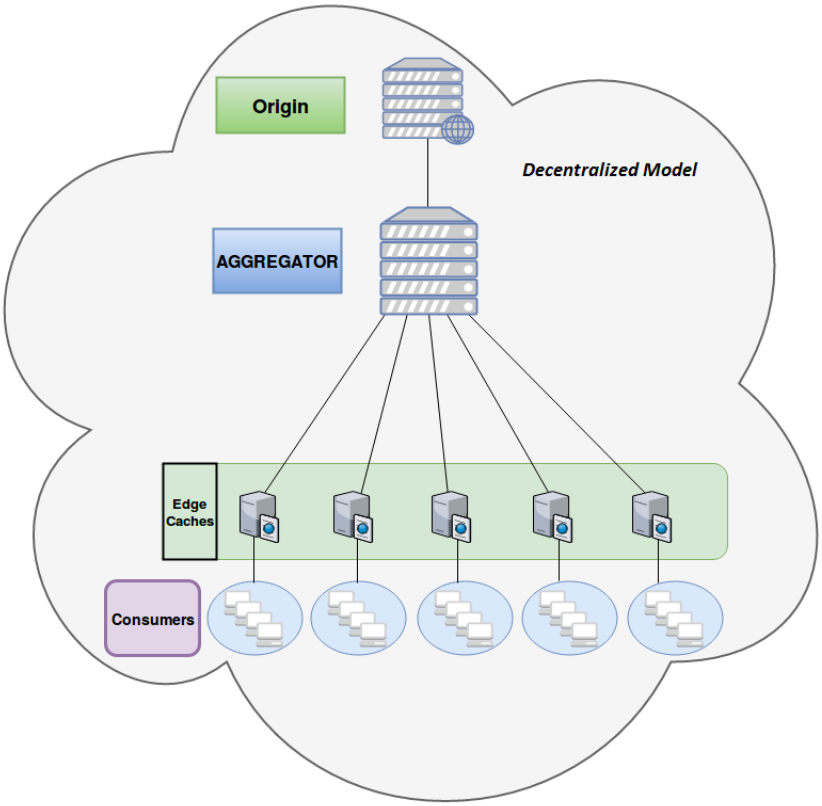


Figure 5.8: Decentralized Model

edges are shared. Also, due to the fact that they are shared, there is no content redundancy, allowing a higher storage of content and reducing consumption in the aggregator.

Scenarios

Three scenarios are evaluated for each different architectural model, which are described in the following:

- Consumer at Home: in this type of scenario the network presents good conditions, which means that the client and the whole system presents low latency, low jitter and enough bandwidth to support the consumption demand of the various clients.
- Mobile consumer in the street: in this type of scenario the network presents some limitations compared to the consumer at home, which means that the network presents latencies between clients and the edge caches, introduced by netem/tc. The clients are in different regions, presenting different average latencies with different jitter values.
- Consumer on a high speed train: in this type of scenario the network has more limitations compared to other scenarios. The limitations are between the edge caches and the aggregator, which are characterized by the mobility of the train and the existence of zones with low signal, which includes limitations mainly in the available bandwidth.

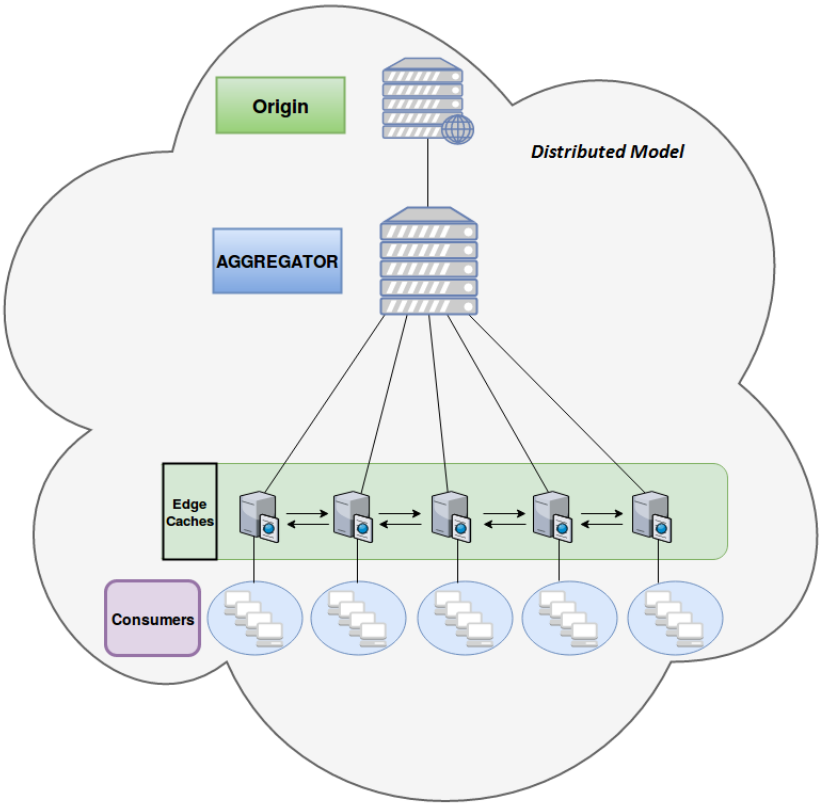


Figure 5.9: Distributed Model

To perform the evaluation of the different models in different scenarios, each edge cache has a 512 MB of in memory LRU cache provided by Cachelot and with a page size predefined with 2MB. The aggregator has 1GB of memory LRU cache provided by the Nginx. Moreover, in all the tests performed the video popularity is given by the zipf's law with a popularity curve defined by s=3, except for the tests with a particular different popularity. For the popularity defined by the curve s=3, the tests it consumes only 3 videos, since the others have a low probability. Also, for s=2 it consumes only 4 videos and for the s=1 it consumes 7 videos for the same reasons as explained to the s=3.

5.3.1 Consumer at home

In the consumer at home scenario, results are presented for the different cases of vertical scalability, horizontal scalability, consumers with consumption of different video popularities and weighted distribution.

The vertical scalability is used to analyze the behavior of the different models with different number of clients. It is analyzed the increase in the number of clients in the level of CPU usage, consumed bandwidth and the performance of the caches, always with the same number of edge caches/proxies.

The horizontal scalability is used to analyze the behavior of the different models with different number of edge caches/proxies available. It is analyzed the increase in the number of the edge caches/proxies, in the level of CPU usage, consumed bandwidth and performance of the caches, always with the same number of customers.

It is also tested the decentralized and distributed model with different video popularities. This test is used to analyze the impact on the caches performance, consumed bandwidth and CPU usage regarding consumer habits, for example, considering a video more or less viral than others.

Finally, in the distributed model it is tested the non-equal distribution between caches, that is, one cache receives more requests than the others. This allows more resourceful caches, both computationally as well as network prone, to serve more traffic than others that do not have as many resources, taking a strong advantage of existing resources.

5.3.1.1 Vertical scalability

Vertical scalability is tested to verify how the different models behave with different numbers of clients by edge cache. Exhaustive tests are performed with the same conditions: the same number of clients for each test, as well as the same physical resources.

Hit and Miss ratio

Figure 5.10 depicts the hit and miss ratio in the centralized model on the aggregator side. It is important to note that there is only one cache in the centralized model that is in the aggregator. It is observed that, with the same popularity, that is, the same percentage of users watching the same videos, the performance of the cache changes, since with the increase on the number of consumers the hit ratio increases and consequently the miss ratio decreases.

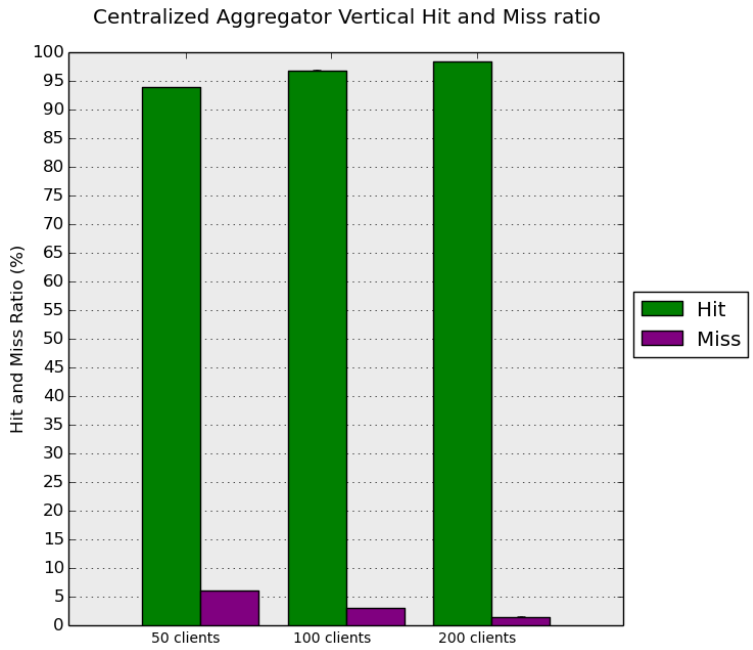


Figure 5.10: Centralized model for different number of clients- Cache Hits and Misses

Figure 5.11 depicts the hit and miss ratio in the decentralized model in the aggregator

and in the edge caches. It is observed that with the same popularity, the performance of the edge caches changes, since with the increase on the number of users the hit ratio increases, and consequently the ratio miss decreases. It is also shown that, with the introduction of an extra cache level, the aggregator presents a lower hit ratio, since the requests are answered mostly by the first layer of caches, which are the edge caches.

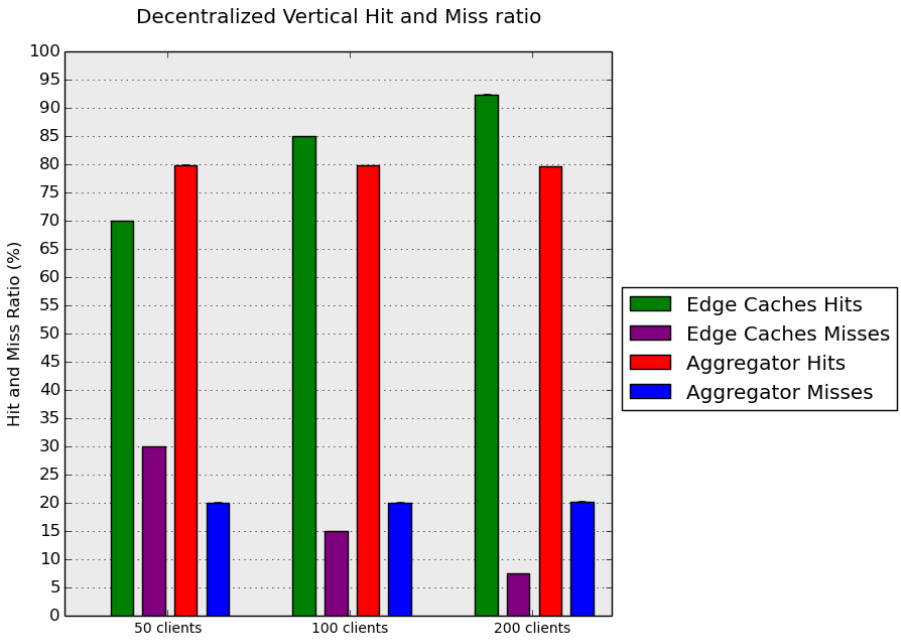


Figure 5.11: Decentralized model for different number of clients - Cache Hits and Misses

Figure 5.12 depicts the hit and miss ratio of the distributed model in the aggregator, as well as, in the edge caches. It is observed that, with the increasing number of clients, maintaining the same popularity, there is a higher hit ratio, and consequently a lower miss ratio on the edge caches. It is also observed that, with the distributed model compared to the decentralized model, the hit ratio is higher and the miss ratio is lower. This is due to the fact that the edge caches share the local cache among them and coordinate to get the content in the aggregator in case none of them have the particular content. As there is no repetition of requests in the aggregator, it presents a low hit ratio and a high miss ratio. Thus, all the content served by the aggregator to the edge caches is only required once, so the aggregator has always to request the content to the origin, since it does not have the content in its local cache.

It is possible to conclude that the distributed model has a higher hit ratio than the decentralized one in the edge caches side, allowing the aggregator to handle a larger number of clients.

Consumed Bandwidth

Figure 5.13 depicts the consumed bandwidth of the centralized model in the aggregator and proxy side. It is observed that, with the increasing number of clients, maintaining the same popularity, the aggregator upload rate increases, since it serves all clients. This happens because the clients request the content to the proxies and these do the same number of requests to the aggregator. So, the aggregator needs to answer to all the clients connected to it. Also,

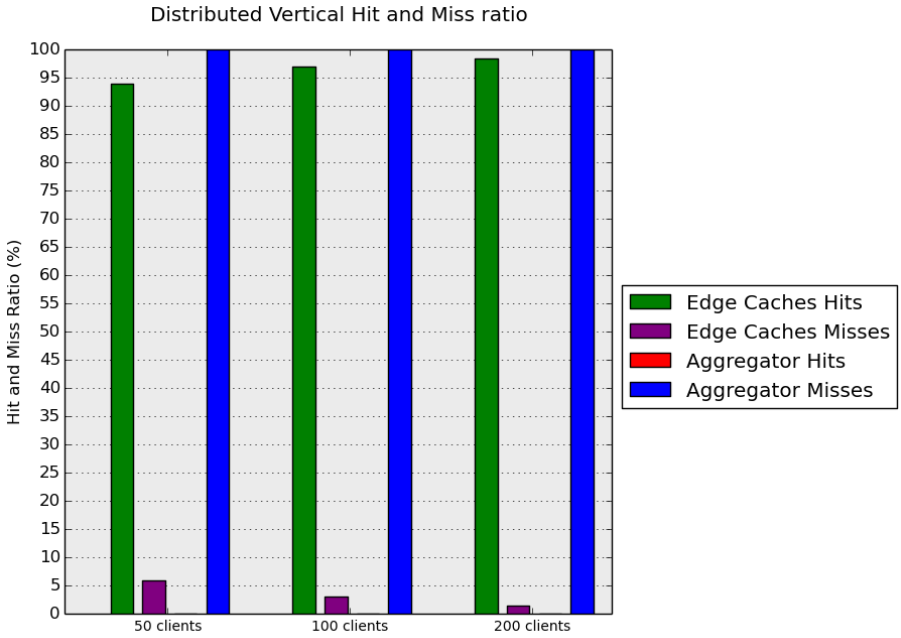


Figure 5.12: Distributed model for different number of clients - Cache Hits and Misses

in the proxy's side it is verified that the upload rate is equal to the download rate due to the fact that all the content downloaded from the aggregator is uploaded to clients. The download rate of the aggregator is always the same due to the same popularity among the several tests for the different number of clients.

Figure 5.14 depicts the consumed bandwidth by the decentralized model in the aggregator and edge caches side. It is observed that, increasing the number of clients and maintaining the same popularity, only the edge caches increase the upload rate, since there are more consumers to serve. Due to the fact that the popularity is equal in all the tests, the download rate, i.e, the content fecthing on the aggregator, is the same for the several tests with different numbers of clients. Also, it is shown that the download rate of the aggregator is the same, as well as the upload rate for the different number of clients. The download rate in the aggregator is lower than the upload rate, since in this model the 5 edge caches do not share the caches, so it is necessary that all edge caches do the same requests to the aggregator.

Figure 5.15 depicts the consumed bandwidth by the distributed model in the aggregator and edge caches side. It is observed that, increasing the number of clients and maintaining the same popularity of visualizations, only the edge caches increase the upload and download rate, since there are more consumers requesting the content and the edge caches have to provide content for them. The download rate varies according to the number of customers, since it has to download the content in the neighboring caches, to provide it to the consumers. It is also possible to observe that the upload rate also increases since the edge caches have to provide the content not only to the clients, but among them as well. Also, in the aggregator side the download and upload rate are the same for the different number of clients. This happens because in this model the 5 edge caches share the same global cache.

It is possible to conclude that the distributed model compared to the decentralized saves bandwidth on the aggregator side, since the edge caches have to coordinate themselves to just

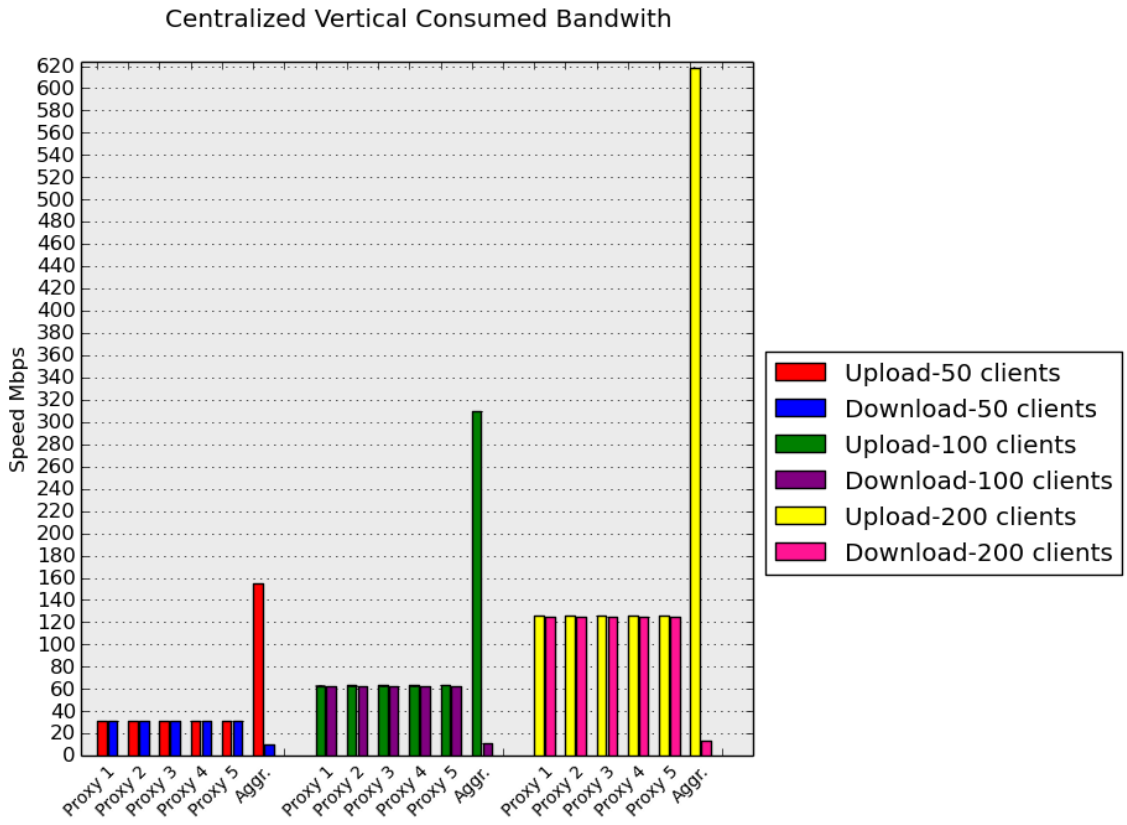


Figure 5.13: Centralized model for different number of clients, consumed bandwidth

one of them to get the content from the aggregator in case that none of them have it.

CPU usage

Figure 5.16 depicts the percentage of CPU usage for the centralized model in the aggregator and proxies. It is observed that, increasing the number of clients, the proxies present a higher rate of CPU usage, since they have to handle more requests from clients. It is also possible to verify that, with the increasing number of clients, it is verified that the aggregator presents a higher rate of CPU usage, since it has to handle all the requests done by the clients.

Figure 5.17 depicts the percentage of CPU usage for the decentralized model in the aggregator and edge cache. It is observed that, increasing the number of clients, the edge caches have a higher rate of CPU usage. It is also possible to verify that, with the increasing number of clients, the aggregator presents the same rate of CPU usage since it has to handler the requests done by the edge caches, which are the same number of requests independently of the different number of clients tested. The number of requests, in the aggregator, is the same in the different number of clients because the edge caches serve the content that has the same homogeneity.

Figure 5.18 depicts the percentage of CPU usage for the distributed model in the aggregator and edge caches. It is observed that, increasing of the number of clients, the edge caches present a higher rate of CPU usage. It is also possible to verify that, with the increasing number of clients, the aggregator presents the same rate of CPU usage since it has to handler

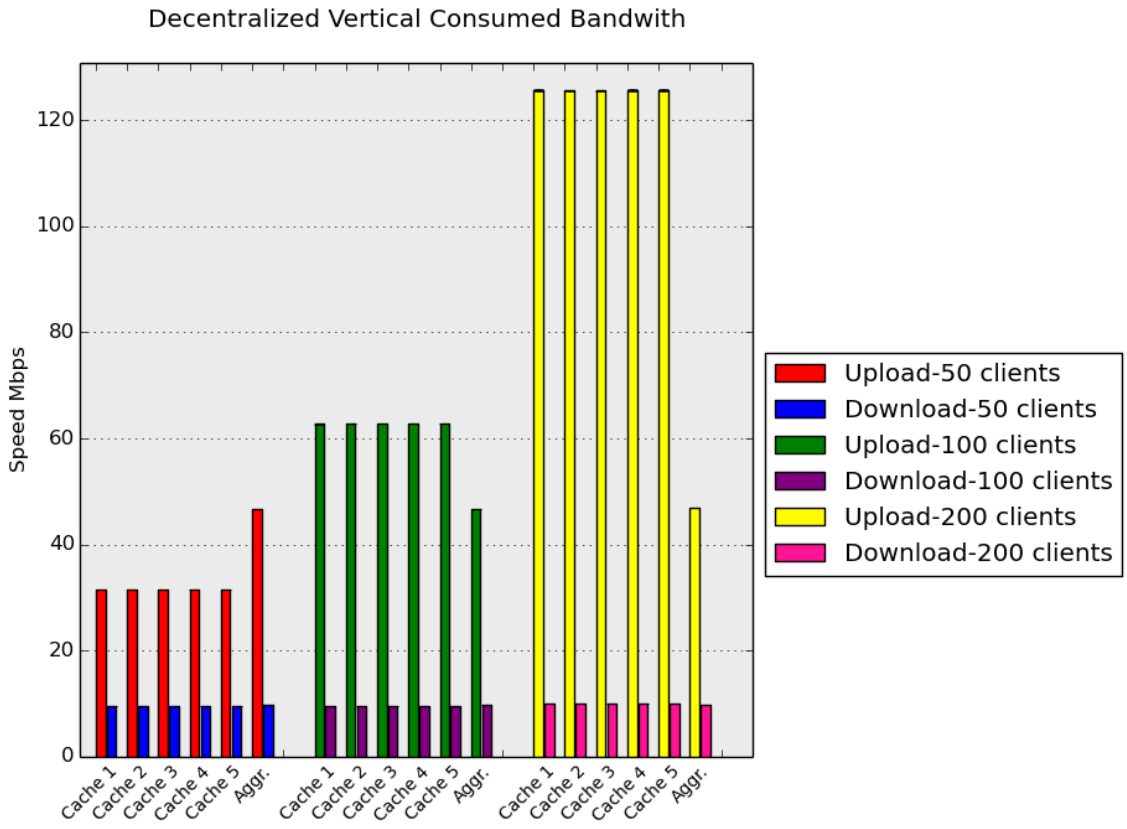


Figure 5.14: Decentralized model for different number of clients, consumed bandwidth

---

the requests done by the edge caches, which are the same number of requests independently the different number of clients tested.

It is possible to conclude that distributed model compared to the decentralized one saves CPU usage on the aggregator side, since the edge caches coordinate themselves in order that just one of them fetches the content from the aggregator in case that none of them have it. Therefore, the number of requests reaching the aggregator is lower on the distributed model. Thus, the lower requests to process, the lower CPU is demanded. Thus, independently of the increasing number of clients and consequently the increasing number of requests, the aggregator receives the same load of requests to process. This happens since the video popularity is the same, independently of the number of clients. So, when a particular content is already stored in the edge caches, it does not need to be fetched in the agregator, which avoids aggregator resources, including CPU consumption.

Considerations

The proposed distributed model presents a higher hit ratio on the edge caches, thus maintaining the aggregator with less load. Moreover, with the existence of groups of edge caches on the distributed model, it is possible to increase the number of clients in the system. However, the edge caches require more bandwidth compared to the decentralized model.

According to the results of the three different models comprising different number of clients, it was observed that the cache performance, hit and miss ratio changes. It was also verified that the centralized model presents a higher CPU usage rate and a higher usage

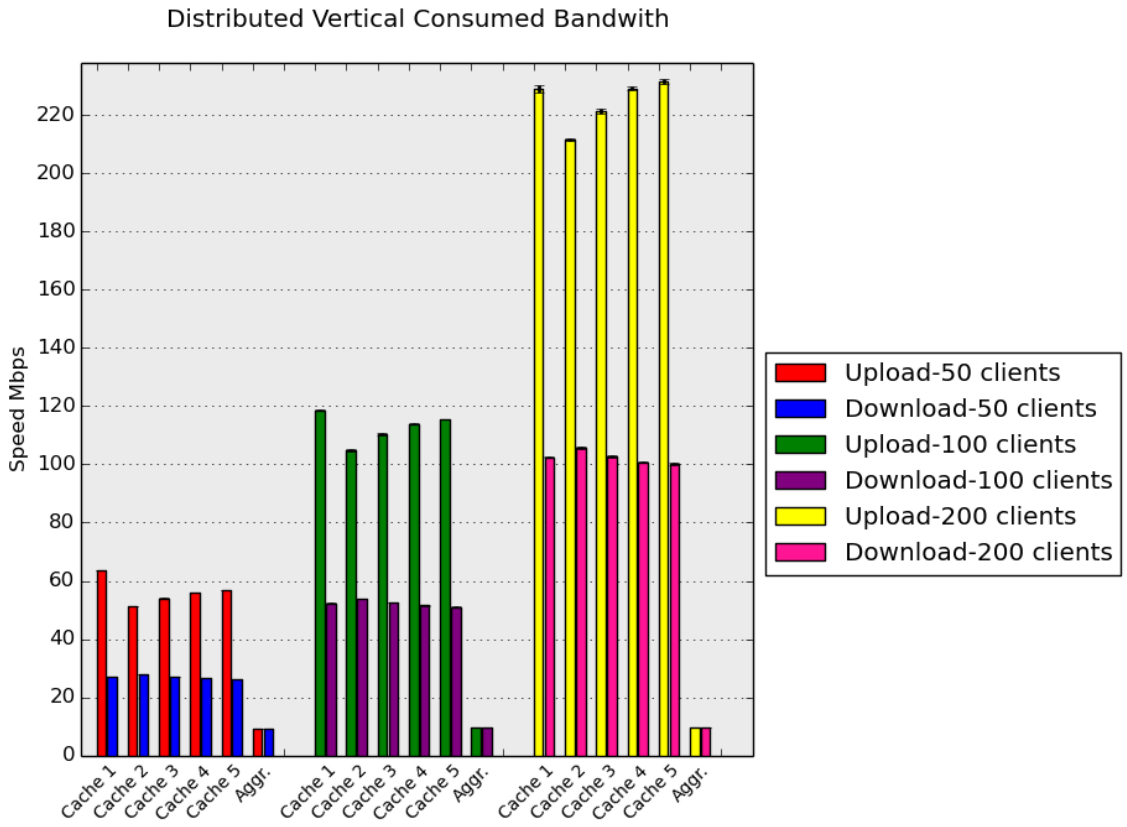


Figure 5.15: Distributed model for different number of clients, consumed bandwidth

of bandwidth on the aggregator side, since it has to handle all the client requests. In the decentralized model, the CPU usage as well as the consumed bandwidth on the aggregator side were reduced when compared with the centralized model; however, there is a higher CPU usage in the edge caches side. Finally, it was also observed that the distributed model presents a higher hit ratio than the decentralized model, as well as it presents a lower consumed bandwidth and lower CPU usage on the aggregator side. However, it presents a higher CPU usage rate and a higher consumed bandwidth in the edge caches side compared to the decentralized model.

5.3.1.2 Horizontal scalability

The horizontal scalability is also evaluated, i.e., it is assessed how the different models with different numbers of edge caches/proxies behave under the same stress, i.e., when the number of consumers is kept steady about 200 clients. To assess the horizontal scalability, exhaustive tests are performed for the different topologies with the same number of clients in the different models as well as the same physical resources. In these tests the CPU usage percentage, hit and miss ratio and consumed bandwidth are measured.

Hit and Miss ratio

Figure 5.19 depicts the hit and miss ratio of the centralized model on the aggregator side. It is observed that, keeping the same number of clients and the same popularity, i.e., the

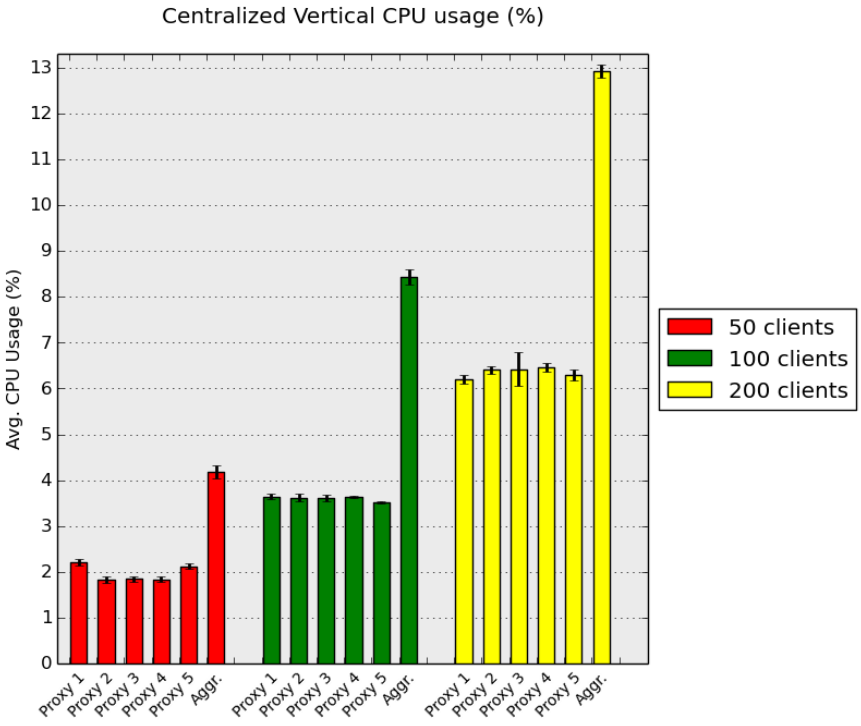


Figure 5.16: Centralized model for different number of clients, (%) CPU usage

percentage of users consuming the same videos, the hit and miss ratio on the side of the aggregator remains equal, since all requests are handled by the aggregator. This happens because the proxies only forward the received requests to the aggregator.

Figure 5.20 depicts the hit and miss ratio of the decentralized model in the aggregator and edge caches side. It is observed that, keeping the same number of clients and the same popularity and varying the number of edge caches, the hit and miss ratio changes in the aggregator side as well as in the edge caches side. Considering the aggregator, this variation is due to the fact that, with more edge caches, the number of clients is distributed by these extra edge caches. This implies that the aggregator has to handle more requests, since with fewer clients per edge cache, requests to the same content are scarce and requests to content that is not in the cache increases, leading to more requests reaching the aggregator, generating misses in the cache and in the aggregator. In the case of 5 caches in the edge caches side there is a higher hit ratio than with 10 edge caches. Also, in the case of the 5 caches in the aggregator side there is a lower hit ratio than with 10 edge caches.

Figure 5.21 depicts the hit and miss ratio on the distributed model in the aggregator and edge caches side. The results show that hit and miss ratios do not change, regardless on the edge caches or on the aggregator. This is due to the fact that edge caches in both cases share the caches providing a global cache among them, coordinating to just one of them to fetch the content in the aggregator in case none of them have it. As there is no repetition of requests in the aggregator, consequently it presents a low hit ratio and a high miss ratio. Thus, all the content served by the aggregator to the edge caches is only required once generating an aggregator miss. So, the aggregator has always to request the content to the origin, since the aggregator does not have the content in cache. Thus, on the side of the aggregator it presents

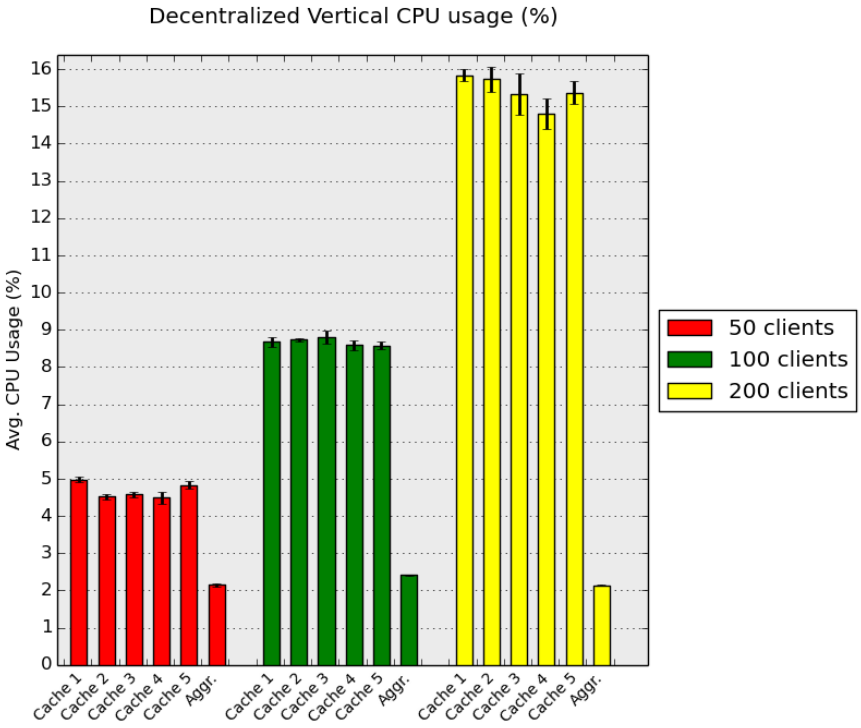


Figure 5.17: Decentralized model for different number of clients, (%) CPU usage

---

100% miss ratio in both the different number of edge caches.

Consumed Bandwidth

Figure 5.22 depicts the consumed bandwidth of the centralized model in the aggregator and proxies side. It is observed that, keeping the same number of clients and the same popularity and varying the number of edge caches, the consumed bandwidth in the aggregator side is the same for both numbers of tested edge caches. The explanation is the same for the observed results regarding Hits and Misses in this scenario. In this case, all requests from clients are handled by the aggregator, since proxies only forward to the aggregator the same amount of requests they receive from clients. The main difference illustrated is that, with the increasing number of edge caches, the download and upload rate is split for the several proxies.

Figure 5.23 depicts the consumed bandwidth of the decentralized model in the aggregator and edge caches side. In this case, it is observable that the consumed bandwidth in the aggregator side is higher when there are 10 edge caches than with 5 edge caches. This is due to the fact that, having more edge caches, clients are distributed by all edge caches, implying that the aggregator has to handle more requests, since the misses in the edge caches is handled by the aggregator. So, with 5 edge caches in the edge caches side there is a higher use of the upload since they have to serve more clients per edge cache, maintaining the global number of clients compared to the case of the 10 edge caches.

Figure 5.24 depicts the consumed bandwidth of the distributed model in the aggregator and edge caches side. In this case, the consumed bandwidth in the aggregator side does not change. This is due to the fact that caches, in both numbers of edge caches available, share the caches among them providing a global cache. The distributed model does not allow edge caches to request the same request twice to the aggregator. So, the aggregator presents 100%

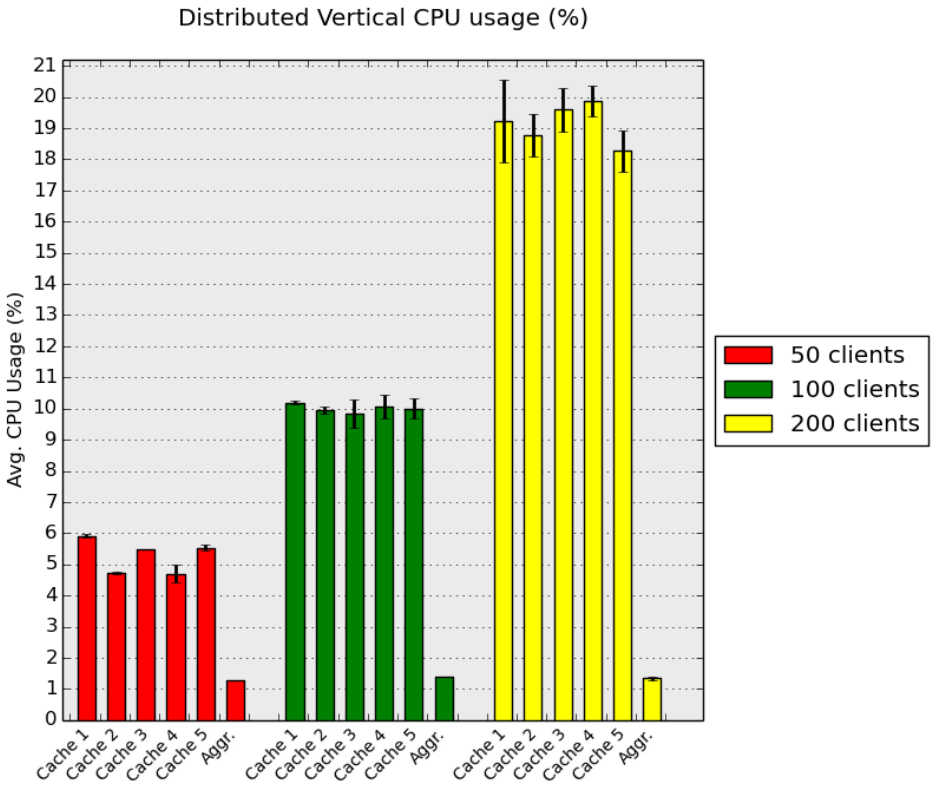


Figure 5.18: Distributed model for different number of clients, (%) CPU usage

miss ratio independently of the number of edge caches used. It is also observed that there is a higher bandwidth consumption rate per client, download and upload, in the edge caches side, with the higher number of caches, because they need to fetch more often the content externally than locally. This is due to the fact that the content is more distributed with more edge caches, then they need to fetch more often externally the content.

CPU Usage

Figure 5.25 depicts the CPU usage for the centralized model in the aggregator and proxies side. It is observed that, keeping the same number of clients and the same popularity and varying the number of proxies, the CPU usage in the aggregator side maintains since all requests are handled by the aggregator in both cases. It is also observed that, with 10 proxies the CPU usage is lower than with 5 proxies, due to the fact that clients are more distributed by the proxies.

Figure 5.26 depicts the CPU usage of the decentralized model in the aggregator and edge caches side. It is observed that the CPU usage in the aggregator side is higher with 10 edge caches compared with 5 edge caches. This is due to the fact that having more edge caches the clients are more distributed, which implies that the aggregator has to handle more requests, since the misses that present in the edge caches is handled by the aggregator. Therefore, with 5 edge caches there is a higher CPU usage, since CPUs have to serve more clients per edge cache than with 10 edge caches.

Figure 5.27 depicts the CPU usage of the distributed model in the aggregator and edge caches side. In this case, it is possible to observe that the CPU usage in the aggregator side

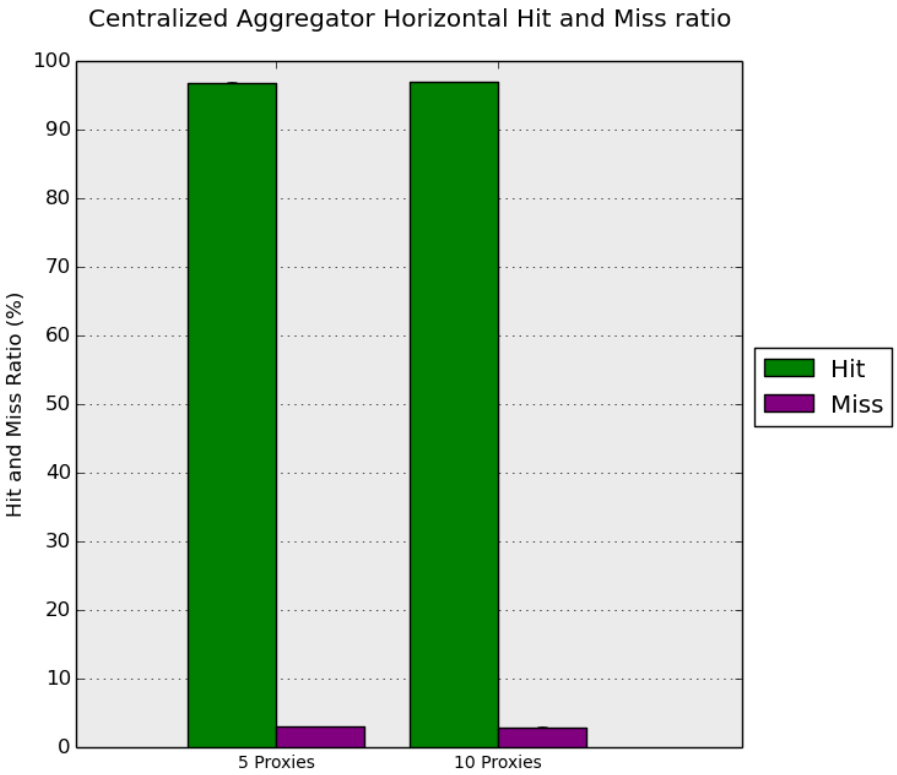


Figure 5.19: Centralized model for different number of proxies

does not change. This is due to the fact that caches in both cases share the caches providing a global cache among them. This model does not allow to request the same request in the aggregator. It is also verified that there is a higher CPU usage per client in the edge caches side, since with the higher number of caches, the content is distributed among them, needing to fetch content more externally than locally.

Considerations:

Analyzing the results of the three models, considering the increasing number of edge caches and keeping the same number of clients and video popularities, it is possible to conclude that the cache performance regarding the aggregator on the centralized model does not change, since the number of handled requests are the same. It was verified that the centralized model presents a higher CPU usage and consumed bandwidth in the aggregator side, since this one has to handle all the requests.

In the decentralized model, it is possible to conclude that, compared with the centralized one, the CPU usage is reduced, as well as the bandwidth consumed in the aggregator side. However, there is a higher CPU usage in the edge caches side compared to the centralized one. It is also verified that, with a higher number of edge caches, the CPU usage increases in the aggregator side, since it has to handle more requests.

Finally, it is also possible to conclude that the distributed model presents a higher hit ratio in the edge caches side compared with the decentralized model. Also, it presents a lower consumed bandwidth and CPU usage in the aggregator side. However, it has a higher CPU usage and a higher consumed bandwidth in the edge caches since they have to provide the content among them.

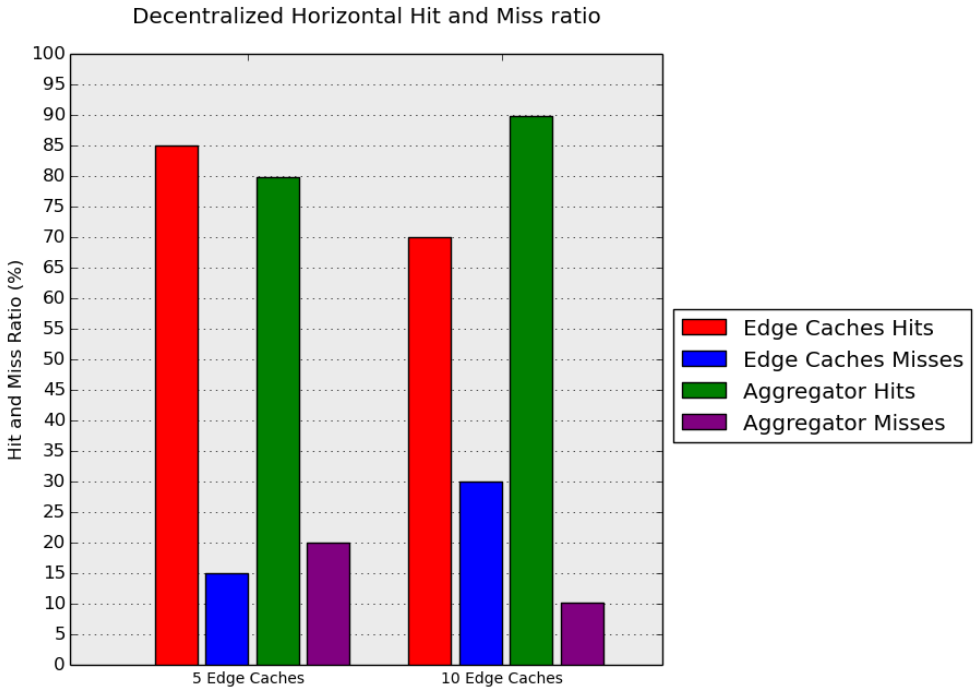


Figure 5.20: Decentralized model for different number of edge caches

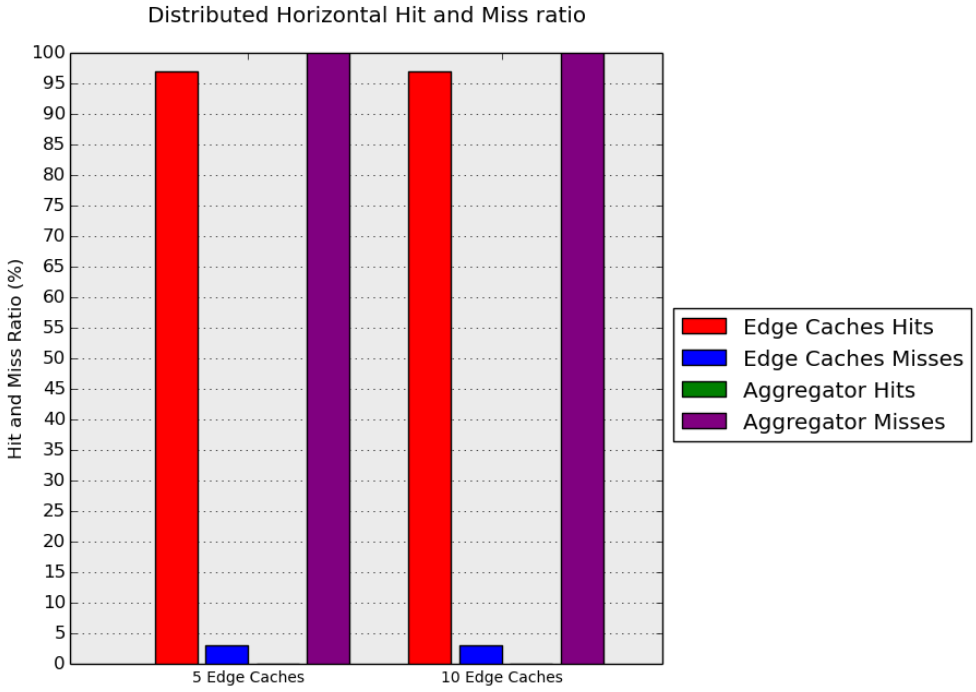


Figure 5.21: Distributed model for different number of edge caches

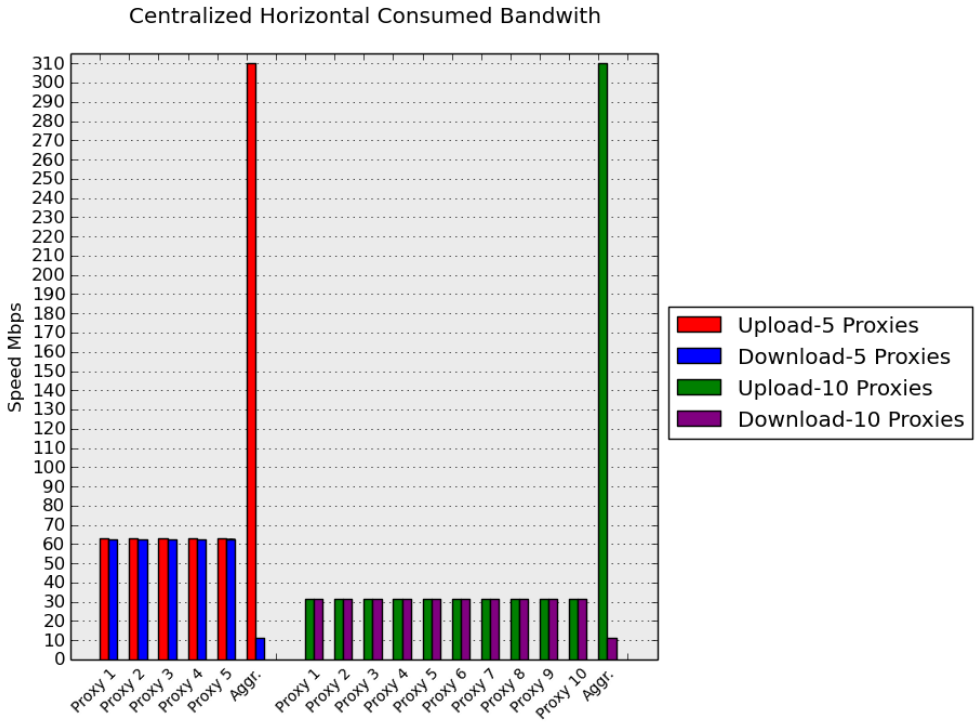


Figure 5.22: Centralized model for different number of proxies, consumed bandwidth

5.3.1.3 Approach by different popularity

This subsection presents and discusses results comprising exhaustive tests performed in order to evaluate the proposed architecture coping with the distributed and decentralized models with three different levels of video consumption popularity, defined by the Zipf' law popularity curves with its parameters s = 1, s = 2 and s = 3. In these experiments, the number of edge caches is 5.

For each popularity curve, tests are performed with the same number of 200 clients. Also, the hit and miss ratios are measured, as well as the percentage of CPU usage, and the bandwidth consumed in the edges caches and aggregator.

Hit and Miss ratio

Figures 5.28a, 5.28b and 5.28c show results regarding the three different video popularities tested, comprising the metrics Hit and Miss ratio on both the decentralized and distributed architectural models.

With the popularity defined by the parameter s = 1 of the Zipf's law function, it is observed that the distributed model presents a higher hit ratio compared to the decentralized model in the edge caches; however, the edge caches present a higher consumed bandwidth since they also provide content.

It is also shown that the aggregator side does not present hit ratio, since the content is not requested to it twice; in other words, the content is only requested to the aggregator once, since when the request arrives to the edge caches, content is searched between the edge caches, and if it is not present, then edge caches coordinate themselves to forward the request to the aggregator in order to avoid repeated requests on the aggregator side.

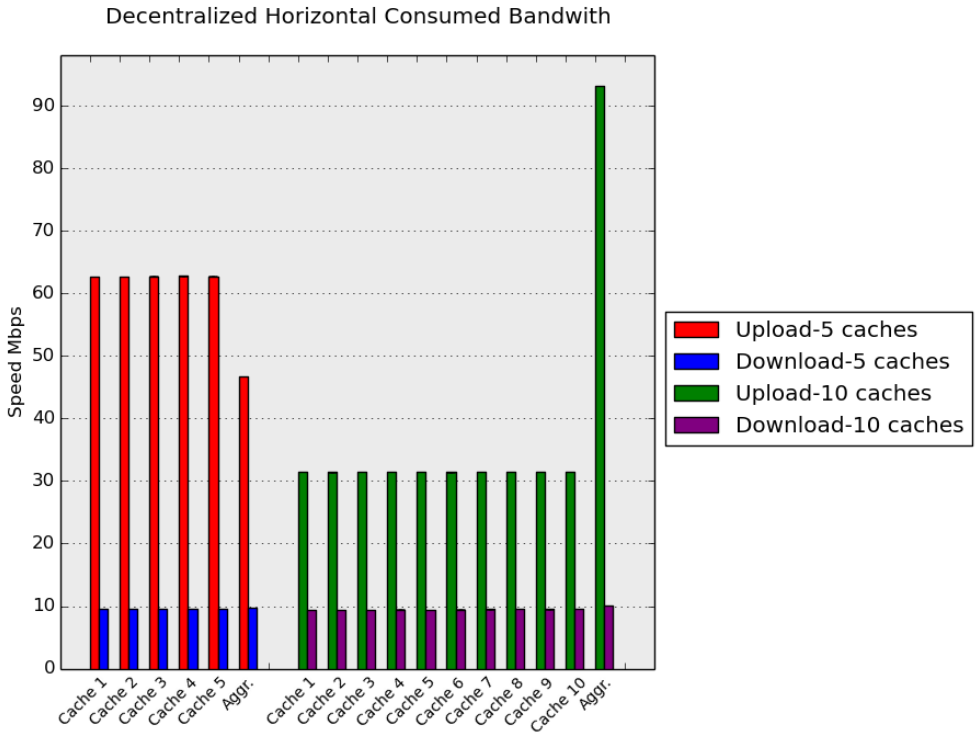


Figure 5.23: Decentralized model for different number of edge caches, consumed bandwidth

---

With the popularity defined by the s = 2 parameter of the Zipf's law function, compared to the s = 1 parameter, there is a higher frequency in the users views at the same video. By analyzing the results, it is observed that the distributed scenario has a higher hit ratio compared to the decentralized scenario in the edges caches, but these present a greater use of bandwidth since edge caches, beyond consuming content from the aggregator, they also provide it among them and to the consumers.

When results comprising the popularity defined by the parameter s = 3 of the zip's law function is compared to the results of the popularity defined by parameter s = 2, it is possible to observe that there is a higher frequency in the users views at the same video. By analyzing the results, it is observed that the distributed scenario has a higher hit ratio compared to the decentralized scenario in the edges caches, but the edge caches present a greater use of bandwidth since they provide the content.

Consumed Bandwidth and CPU Usage

Figures 5.29, 5.30 and 5.31 show results regarding the three different video popularities tested, comprising the metric Consumed Bandwidth on both the decentralized and distributed architectural models. On the other hand, Figures 5.32, 5.33 and 5.34 depicts results regarding metric CPU Usage on the same models.

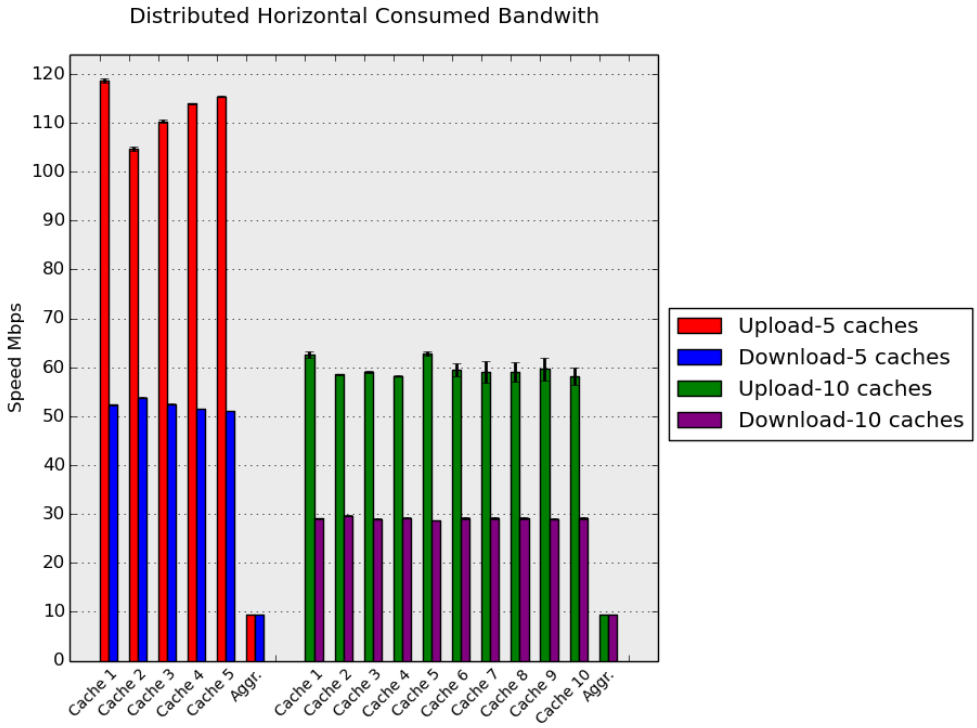


Figure 5.24: Distributed model for different number of edge caches, consumed bandwidth

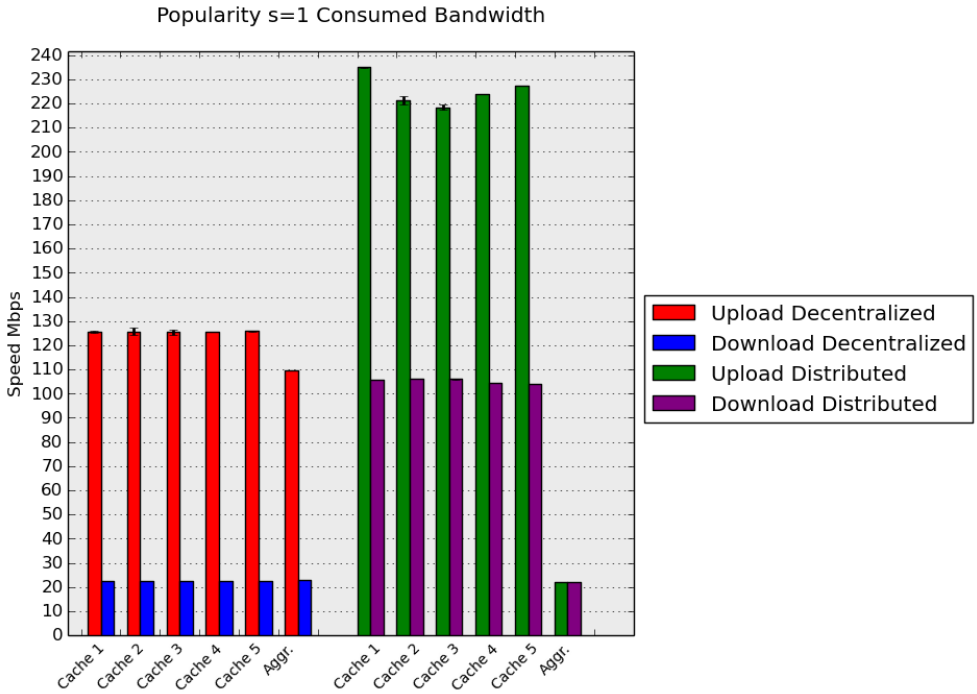


Figure 5.29: User at Home Scenario - Consumed Bandwidth, video popularity s=1

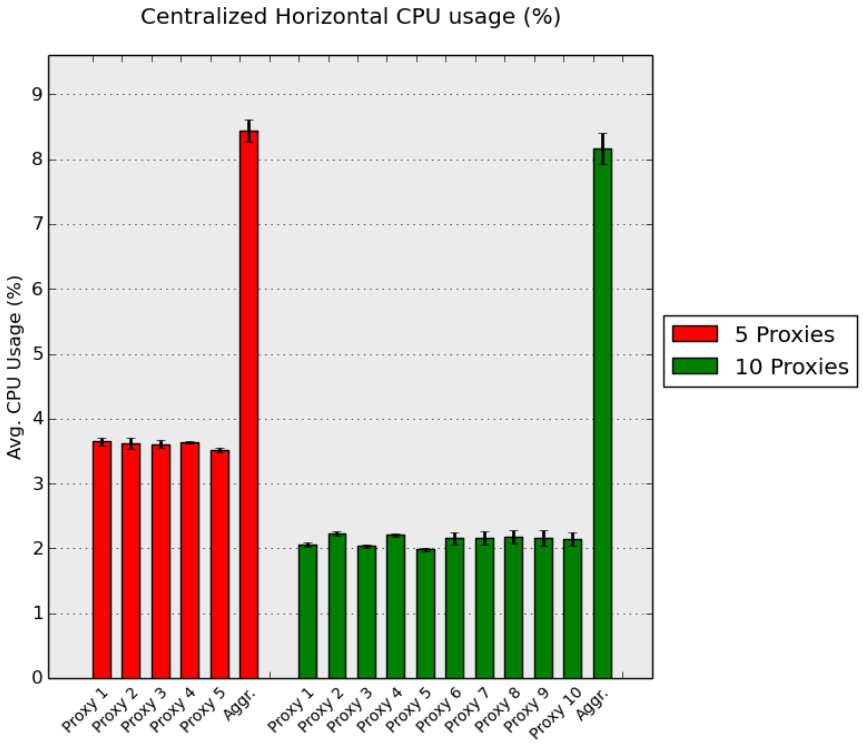


Figure 5.25: Centralized model for different number of proxies, CPU usage

---

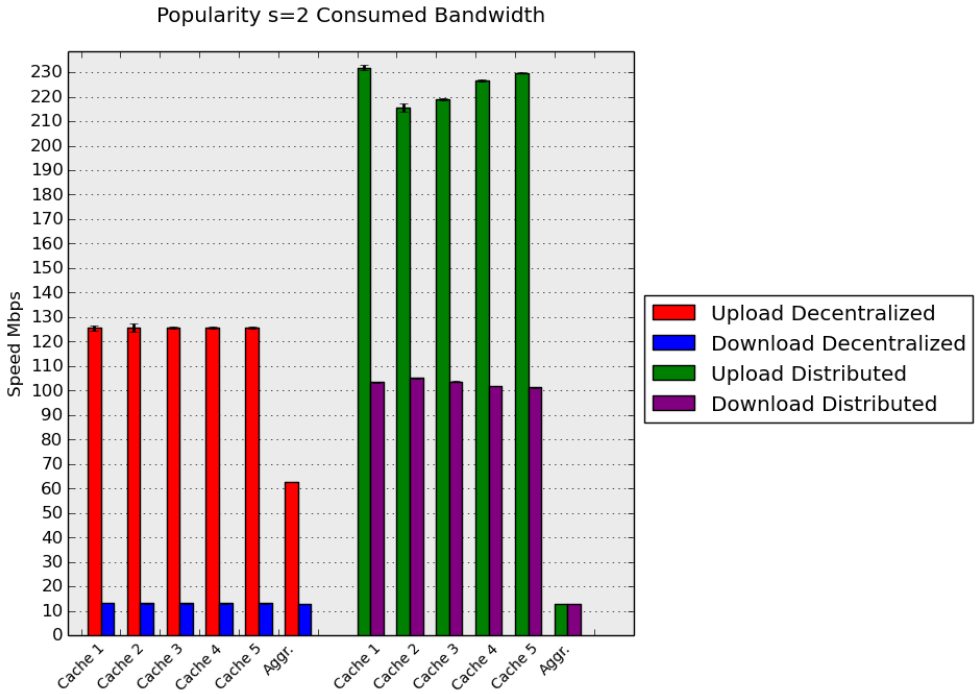


Figure 5.30: User at Home Scenario - Consumed Bandwidth, video popularity s=2

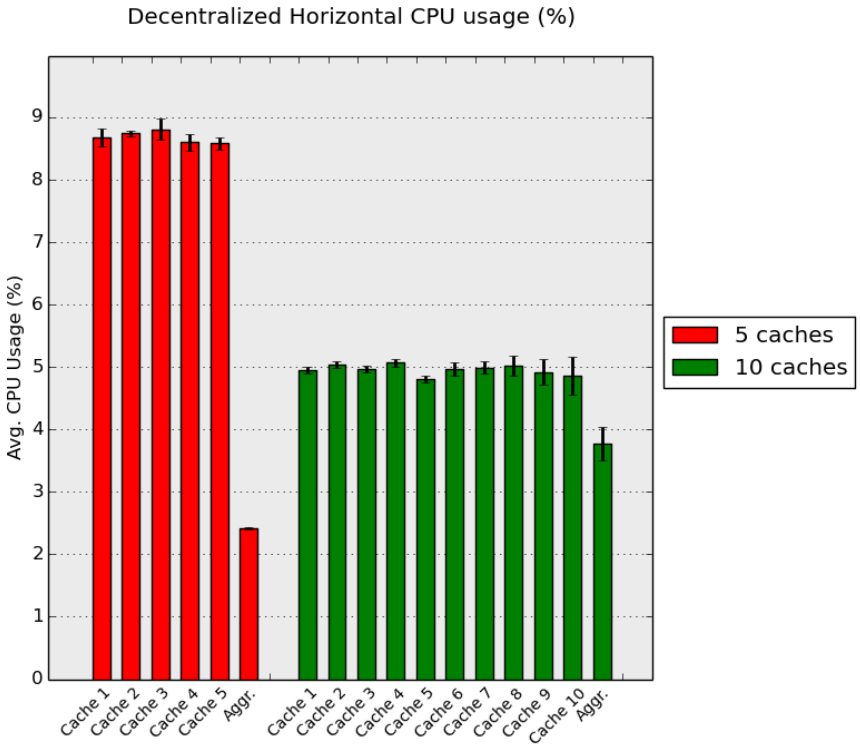


Figure 5.26: Decentralized model for different number of edge caches, CPU usage

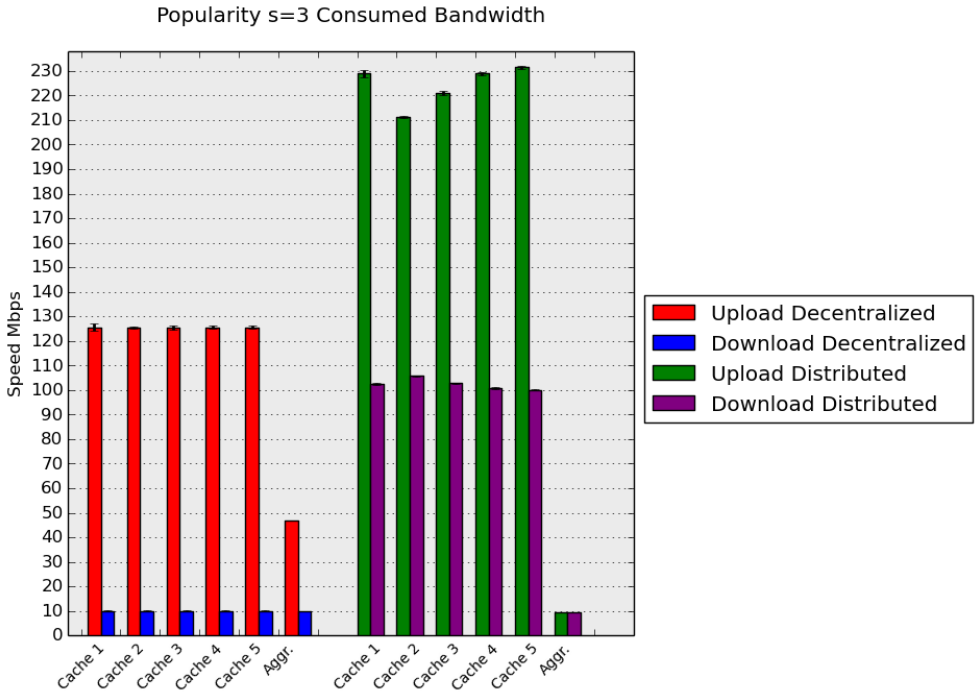


Figure 5.31: User at Home Scenario - Consumed Bandwidth, video popularity s=3

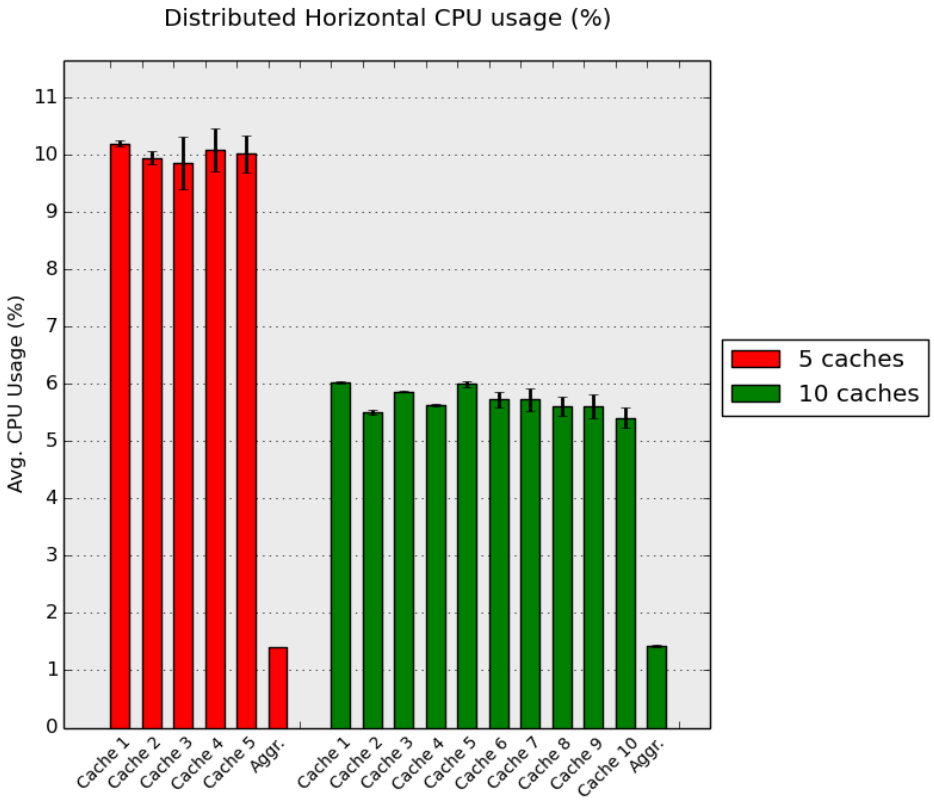


Figure 5.27: Distributed model for different number of edge caches, CPU usage

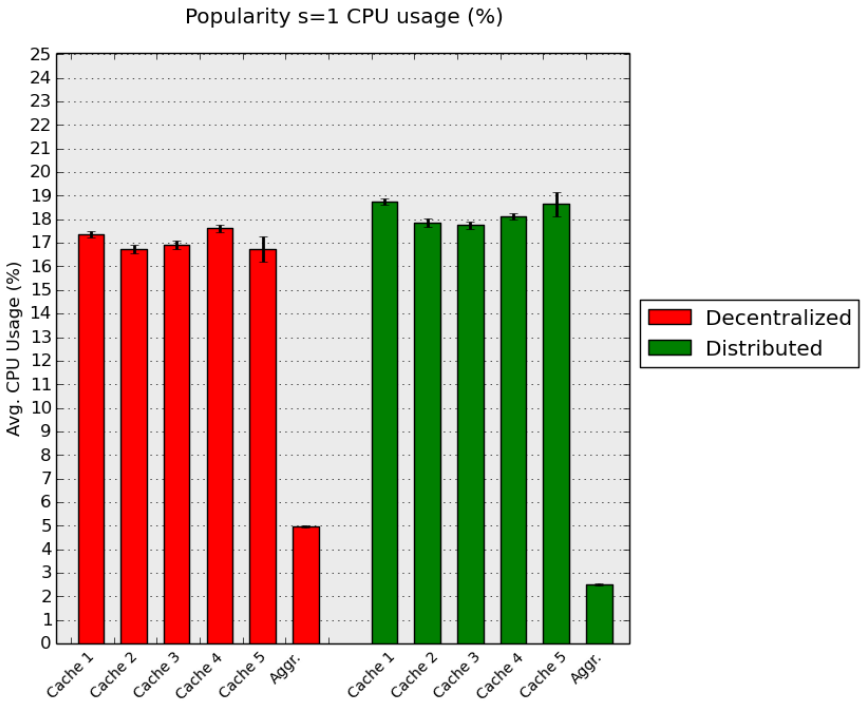
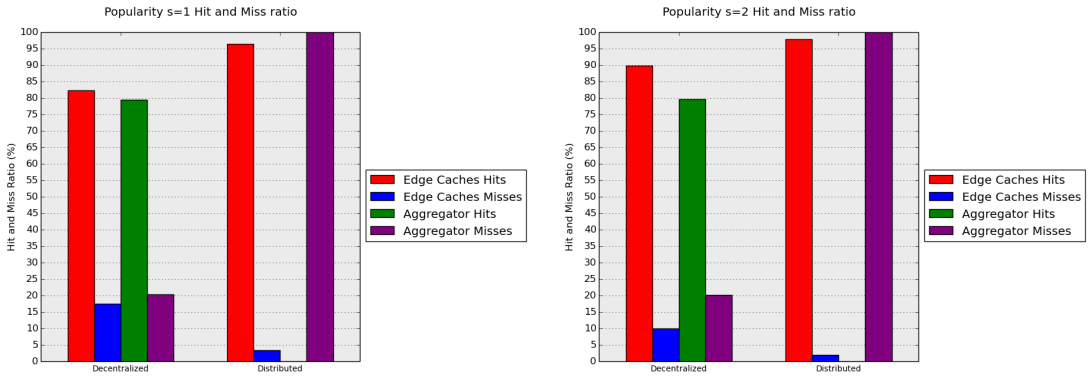
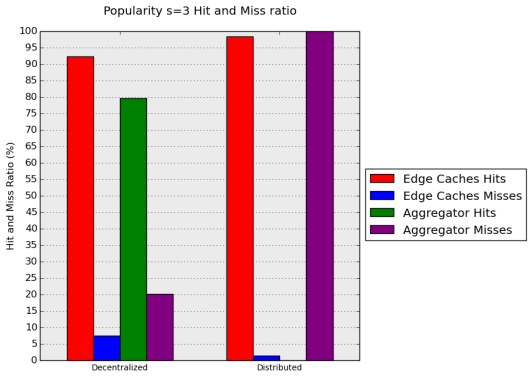


Figure 5.32: User at Home Scenario - CPU usage, video popularity s=1



(a) Popularity s=1

(b) Popularity s=2



(c) Popularity s=3

Figure 5.28: Hit and Miss ratio for different video popularities

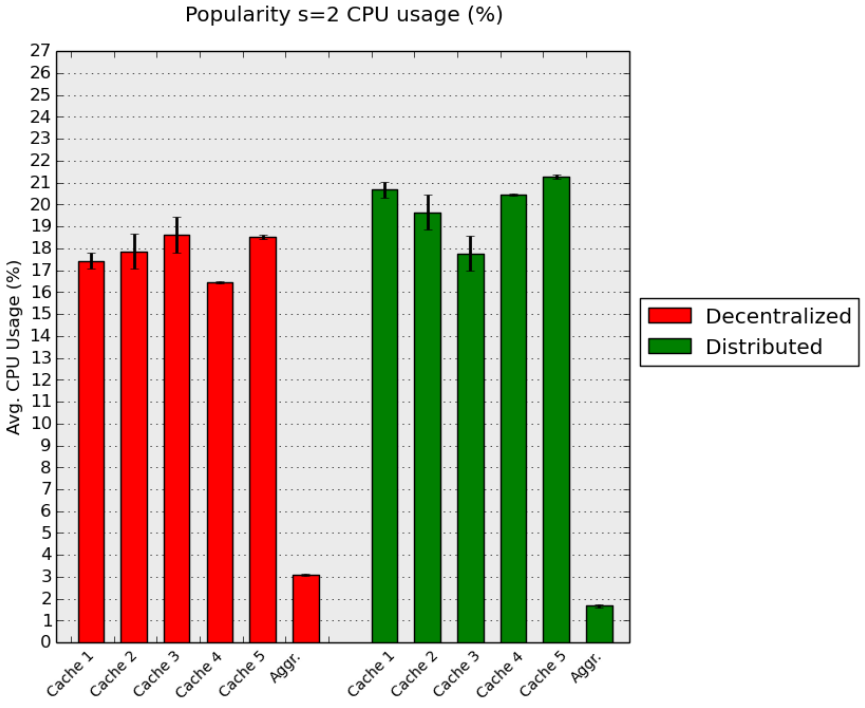


Figure 5.33: User at Home Scenario - CPU usage, video popularity s=2

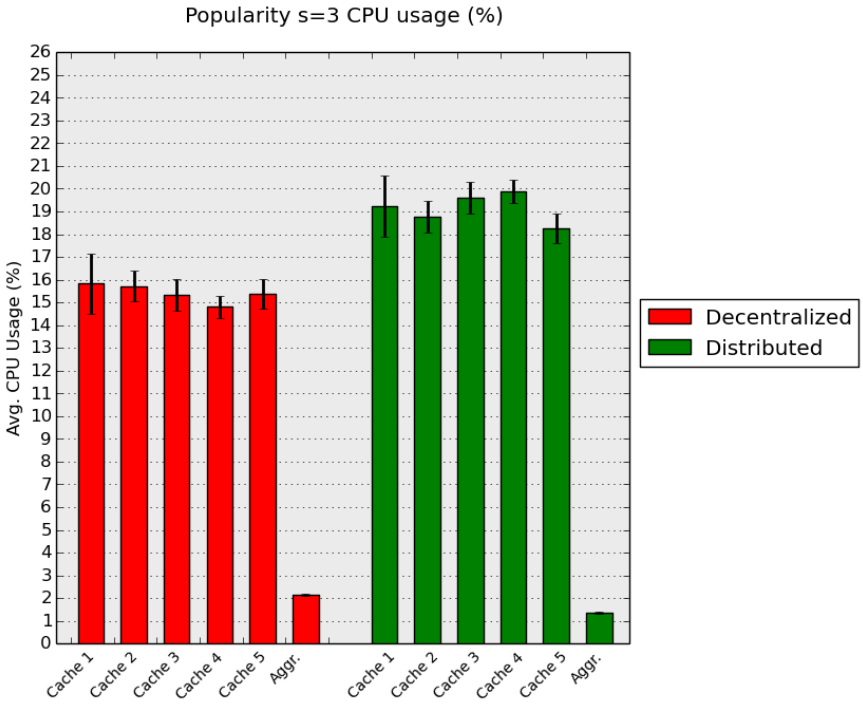


Figure 5.34: User at Home Scenario - CPU usage, video popularity s=3

By analyzing the results obtained, it is possible to observe that, depending on the habits of the clients, i.e., depending on the popularity of the videos, the bandwidth consumption as well as the load on the aggregator is changed. It is also observed that the distributed models present a higher usage of the bandwidth in the edge caches side. However, the aggregator presents/shows a lower load and also a reduction of the consumed bandwidth compared to the decentralized model, allowing the aggregator to be able to respond to a higher number of clients/requests, thus improving QoE. Also, depending on the popularity of the videos, it is possible to observe that the hit and miss ratio parameters also vary.

In the results, the more popular a video (s = 3) is when compared to the others (s = 1 and s = 2), the lower is the bandwidth utilization and load on the side of the aggregator, as well as the larger is the hit ratio on the edge caches side. Thus, it is concluded that the distributed model presents a better performance, in terms of reducing computational and network load in the upstream server, in this case the aggregator. However, on the edge caches side, the distributed model presents a higher utilization of the network, as well as a higher computational use compared to decentralized model.

5.3.1.4 Weighted Distribution

This section shows and discusses the results regarding several tests performed on the distributed model comprising different weights in the distribution of content between the edge caches. In this case there is a preference on where to put the content in one cache against the others. This is useful in the case where edge caches do not have the same capabilities. Thus, if an edge cache has a higher local cache size or even higher bandwidth, then it can store and provide more content than the others, therefore making better use of resources. Thus, in

these tests the middle edge cache receives, stores and provides more content face to the other edge caches. Moreover, the percentage of the content which an edge cache has compared to the other is calculated by the Equation 5.6:

$$PercentContent_i = \frac{Weigh_i}{\sum_{j=1}^{N} Weigth_j} * 100 \tag{5.6}$$

Where:

- $PercentContent_i$ is the content percentage of an edge cache i;
- $Weight_x$ is the weight of a given edge cache x;
- N is the total number of edge caches.

Also, the formula to validate the the upload rate is given by 5.7:

$$Upload_i = \frac{Weight_i}{\sum_{j=1}^{N} Weigth_j} * br * (NGC - NCC) + br * NCC \tag{5.7}$$

Where:

1. $Upload_i$ is the upload rate of an edge cache i;
2. $Weight_x$ is the weight of a given edge cache x;
3. N is the total number of edge caches;
4. br is the average bit rate for each client. In this case it is 3 Megabits Per Second (Mbps), since all the clients consume the higher resolution.
5. NGC is the the number of global clients. In this case it is 200.
6. NCC is the number of clients in a given edge cache. In this case, it is 40 since clients are uniformly distributed.

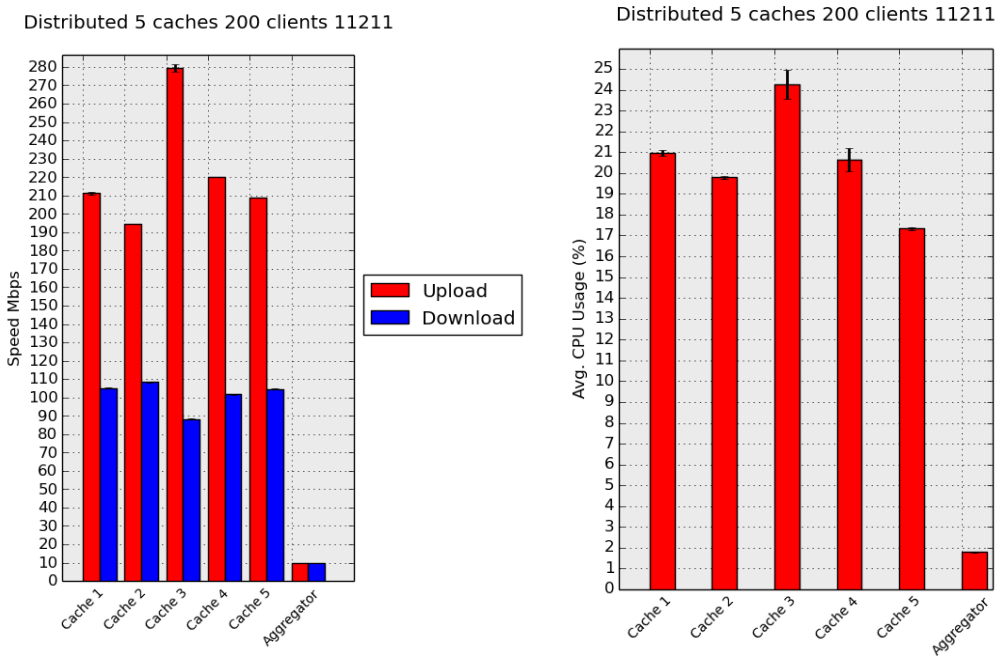
This formula is only valid when users are seeing the same video quality, verified by the measured QoE. The qualities required are always the same, which in this case is the maximum video resolution, and the same video popularity is used in these tests. Thus, tests were performed to analyze the impact of different weights on bandwidth and computational resources load in the system. For this purpose, 200 clients are distributed equally by each edge cache, that is, 40 clients per cache. Five edge caches are used as well as one aggregator and one origin server.

Weighted Distribution for 11211

In this test, edge caches 1, 2, 4 and 5 have weight values equal to 1. On the other hand, edge cache 3, the middle cache, has a weight of 2. The middle edge cache has 33.3 % of the stored content and the other edge caches have approximately 16.6% each one, according to Equation 5.6.

Figure 5.35a shows results comprising consumed bandwidth and Figure 5.35b shows results regarding CPU usage, both with these weight values.

The results show that the edge cache 3 (middle edge cache) presents a higher CPU load compared to the others since it has to handle more requests and content. In terms of bandwidth, it has a higher use of bandwidth compared to others, due to the fact that it has a



(a) Consumed Bandwidth

(b) CPU usage(%)

Figure 5.35: User at Home - Weighted Distribution 11211

higher upload rate. It has a lower download rate compared to others, since this edge cache is the one that provides more content, but it does not need to fetch content from the others and thus allowing less downlink usage. According to Equation 5.7, edge cache 3 must have an upload rate of 280 Mbps, which is observed in Figure 5.35a.

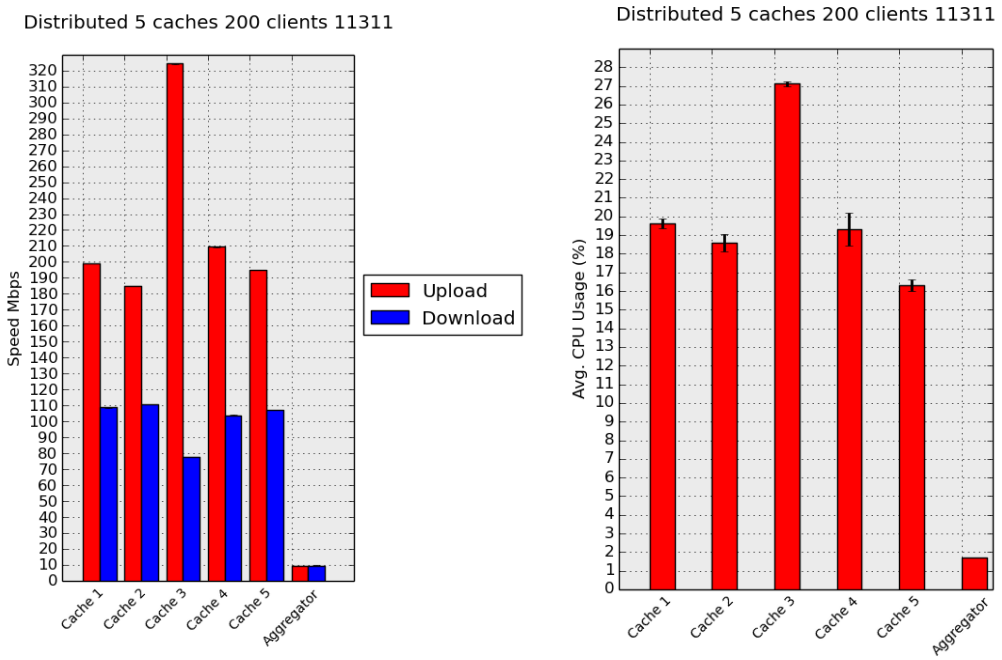
Weighted Distribution for 11311

In this test, edge cache 3 has a weight of 3 and the others remain with a weight of 1. The middle edge cache has 42.8% of the content and the other edge caches approximately 14.3%, according to Equation 5.6. Figure 5.36a shows results comprising consumed bandwidth and Figure 5.36b shows results regarding CPU usage.

The results in Figures 5.36a and 5.36b show that the edge cache 3 presents a higher CPU load compared to the others, since it has to handle more requests. Moreover, the edge cache 3 presents a larger CPU load compared to the weighted distribution 11211 as well as a higher uplink utilization. However, the other edge caches present a higher usage of the downlink, and a reduction of uplink, since the content is mostly found in edge cache 3. The edge cache 3 presents a reduction of downlink since it has 42.8% of the content, which means that it does not need to get content in the neighbors edge cache, and consequently it does not need to use often its downlink communication channel to do so. Also, in terms of consumed bandwidth, it has a higher usage of bandwidth compared to others, in this case, it has a higher upload rate since it has to provide more content than the others. According to Equation 5.7, edge cache 3 must have an upload rate of 320 Mbps, which is observed in Figure 5.36a.

Weighted Distribution for 11411

In this test, edge cache 3 has a weight of 4 and the others remain with a weight of 1. The



(a) Consumed Bandwidth

(b) CPU usage(%)

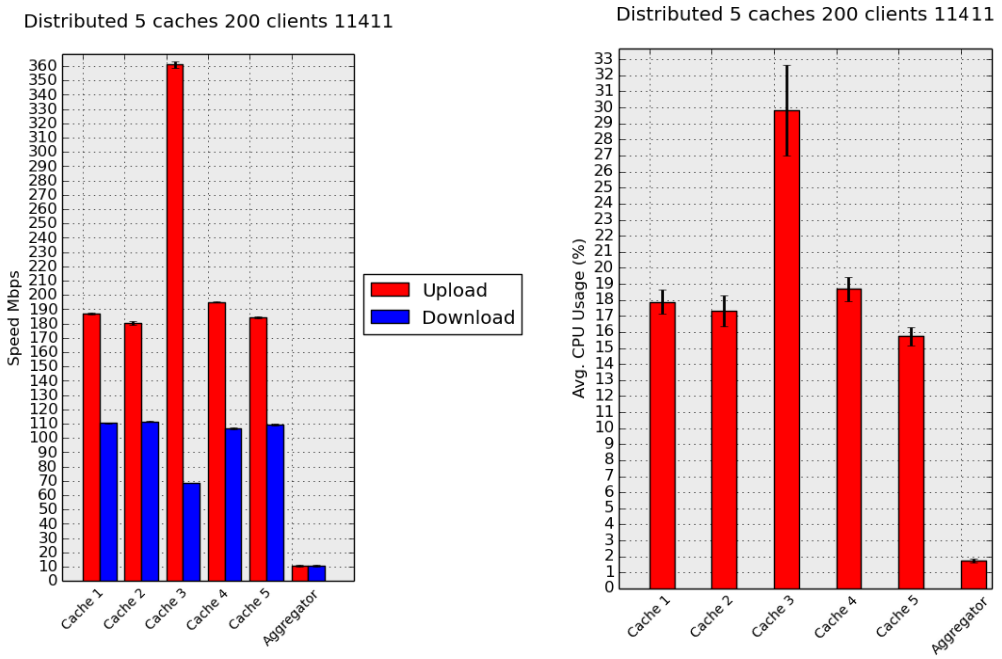
Figure 5.36: User at Home - Weighted Distribution 11311

middle edge cache has 50% of the stored content and the other edge caches have approximately 12.5% each one, according to Equation 5.6. Figure 5.37a shows results comprising consumed bandwidth and Figure 5.37b shows results regarding CPU usage.

The results in Figures 5.37a and 5.37b show that the edge cache 3 (middle edge cache) presents a higher CPU load compared to the others, and to the same edge cache with the weights of 3 and 2. However, the other edge caches present a higher download rate, and a reduction in the uplink rate, since the content is mostly found in edge cache 3. The edge cache 3 presents a reduction in the downlink since it has 50% of the content, which means that it does not need to get content in the neighbors edge cache. As in previous cases, it has also a higher use of bandwidth compared to others. According to Equation 5.7, edge cache 3 must have an upload rate of 360 Mbps, which is observed in Figure 5.37a.

Considerations

From the previous results, we observed that it is possible to perform a weighted distribution, allowing, for example, a better utilization of the resources, since a machine that has more bandwidth or more computational resources can store and serve more content than the other edges caches. The more weight a edge cache has, the more content it will provide, so its uplink rate is higher than the others and there will be a higher computational effort. In addition to the weighted edge cache having a higher uplink rate compared to others, it has also a higher CPU load since it will serve more content to the others. Contrary to the edge cache that has more weight, the other edge caches have less load in the CPU as well as a reduction of the uplink rate, since these do not provide as much content. However, the download rate increases since the content has to be fetched from the neighbors and, in particular, the one that has the least weight, since there is a higher probability that the content is stored



(a) Consumed Bandwidth

(b) CPU usage(%)

Figure 5.37: User at Home - Weighted Distribution 11411

in another edge cache.

5.3.2 Mobile consumer in the street

This section shows and discusses results comprising the tests performed on the distributed and decentralized models, considering that clients are mobile, i.e., the link between the edge caches and the consumer suffers from limitations, inducing some latency and a higher jitter due to obstacles and the distance of clients from the base stations and access points, which are characteristic of mobile networks. In this scenario there are several clients that are spread over 3 different regions, with different latencies for each region. In this specific case, clients from region A have an average latency of 80ms with a jitter of 40ms. The latency is defined by a normal distribution in all cases. In region B, clients have variable latency with an average of 350 ms and jitter of 200 ms. Finally, in region C, clients present latency with an average of 800 ms and jitter of 250ms.

QoE is measured for each customer in each region in order to analyze its variation comprising the limitations imposed on the three groups. It is also measured the performance of the edge caches in the different architectural models comparing with a scenario with perfect conditions, where the network has low latency and enough throughput to serve the consumer requests. In addition to the performance, it is measured the impact of bandwidth comparing the different models comprising the aforementioned limited network conditions, as well as with perfect conditions. The tests are performed with a global number of 150 clients. Thus, each client/customer group has a global number of 50 clients. Moreover, each edge cache has 10 clients in region A, 10 clients in region B and 10 clients in region C. Bearing this in mind, five edge caches and one aggregator, as well as one origin are used, as illustrated in

Figure 5.38, which depicts the mobile consumer in the street.

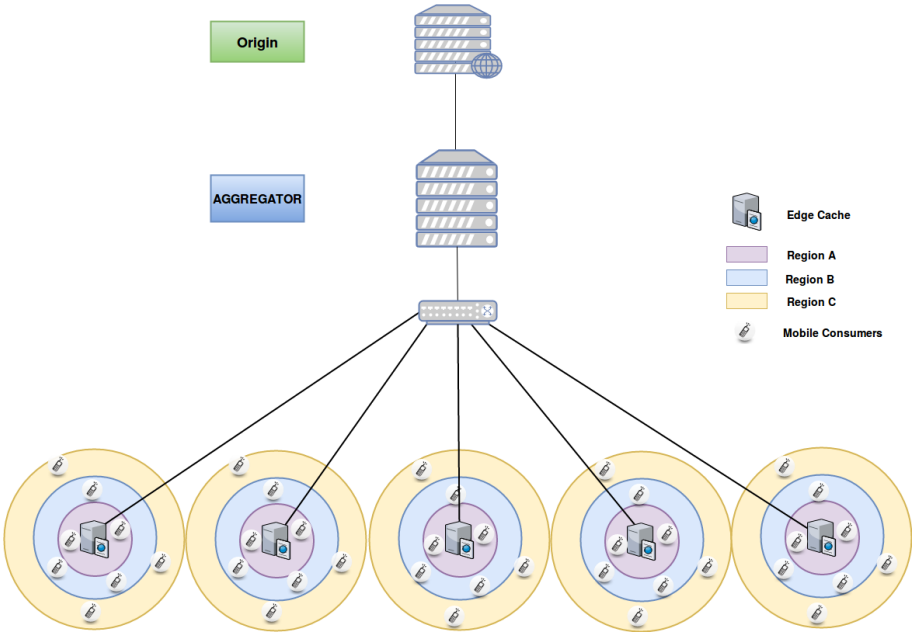


Figure 5.38: Street Mobile Scenario

QoE

Figures 5.39a and 5.39b depict the results comprising the QoE on the decentralized and distributed models, respectively. The results show the deterioration of the network conditions: with the increase of the latency, the QoE is lower, since the network is not propitious to provide good data transmission conditions. In region A the latency has no significant impact in the QoE, always larger than 4. In region B a degradation of the QoE occurs, though it continues above 4, which is a good QoE. Due to jitter there is a higher discrepancy among the QoE of the different customers, since the confidence interval is larger. In region C there is an average QoE below 4, with a significant degradation compared to the other regions. Moreover, the confidence interval is also larger due to jitter. The distributed model, compared to the decentralized model, has significant improvements in region C, with the same limitations imposed by the latency generated by the tc/netem.

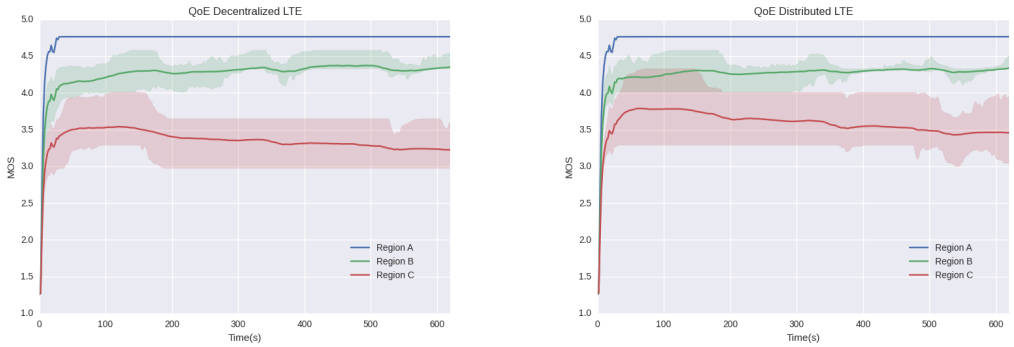
Hit and Miss ratio

In addition to the QoE, the hit and miss ratios were also calculated in both scenarios (with and without network limitations), for both distributed and decentralized models.

In Figure 5.40 it is shown that, in the decentralized model as well as in the distributed model, with the same percentage of clients watching the same videos, but with different network latencies, the performance in the edge caches side decreases, with a lower hit ratio and a higher miss ratio. It also turns out that, in the aggregator side, the performance is also decreased. In both cases, the distributed scenario has always a better result in terms of edge caches performance.

Consumed Bandwidth

The consumed bandwidth is measured on the scenarios for both the decentralized and distributed models. The decentralized model has reduced the upload in edge caches side, since there is a consumption of a lower video resolution, implying a smaller bit rate per client.



(a) Decentralized model (b) Distributed model

Figure 5.39: QoE mobile scenario

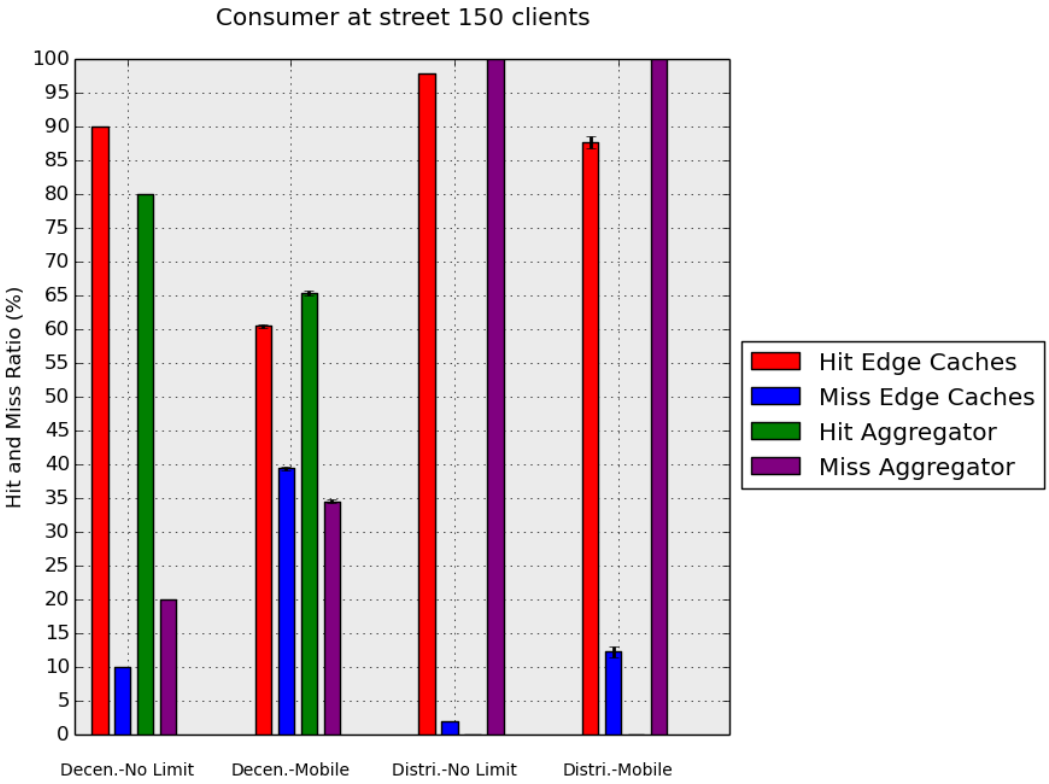


Figure 5.40: Street mobile scenario hit and miss ratio

Also, the download rate on the edge caches increased, even with the same video popularity, since customers are watching different resolution qualities, also increasing the upload and download of the aggregator. In the distributed model there is a reduction of the bandwidth consumed in the upload and download; in the edge caches side, clients consume inferior qualities, with lower bit rate. However, the consumption in the aggregator has increased, since there is a higher difference of qualities, maintaining the same popularity among the videos.

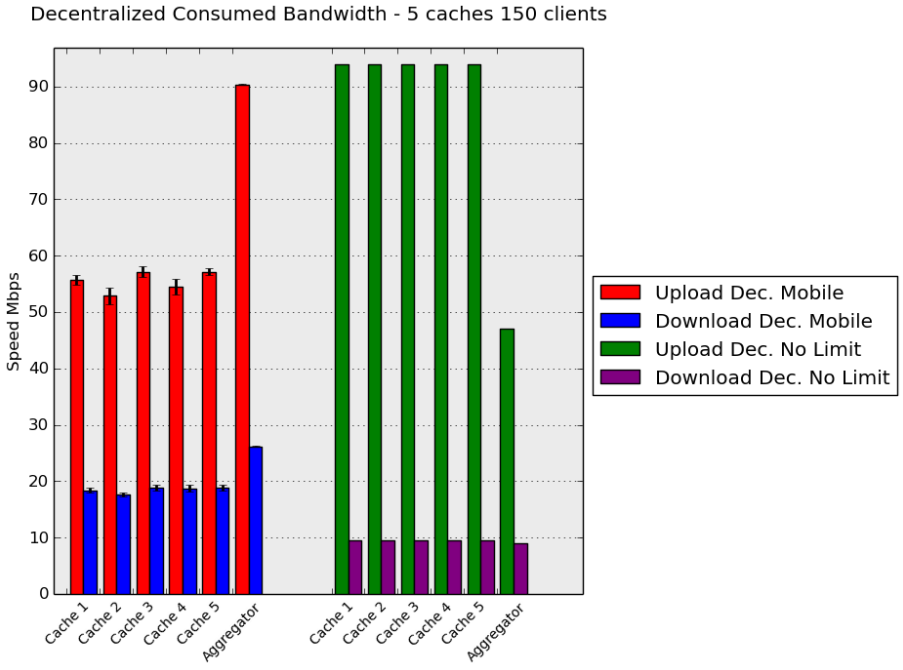


Figure 5.41: Street mobile scenario, decentralized model, consumed bandwidth

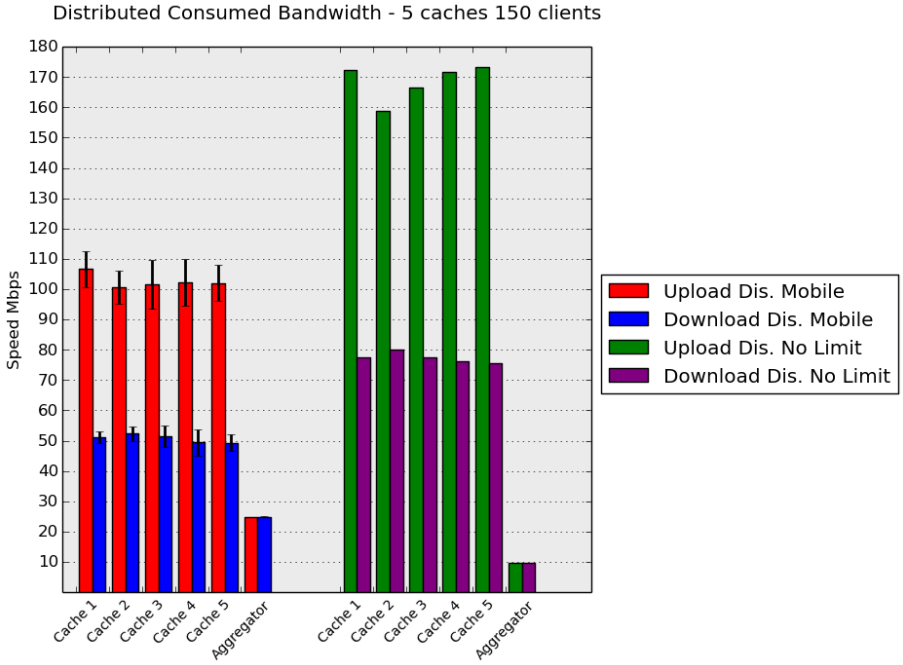


Figure 5.42: Street mobile scenario, distributed model, consumed bandwidth

---

Analyzing the results of the two models, distributed and decentralized, it is possible to conclude that in the distributed model, the consumed bandwidth in the edge caches side is higher than in the decentralized model. Moreover, the distributed model presents a lower

consumed bandwidth in the aggregator side compared to the decentralized model.

CPU usage

CPU usage is measured in both models and scenarios. Figures 5.43 and 5.44 depict results of the performed experiments. The decentralized model, with the clients watching the same videos, but with different latencies, presents a lower CPU usage in the edge caches side, since clients consume video resolution with lower bit rate. Also, in the aggregator side, the CPU usage increased, since there is a demand for different qualities.

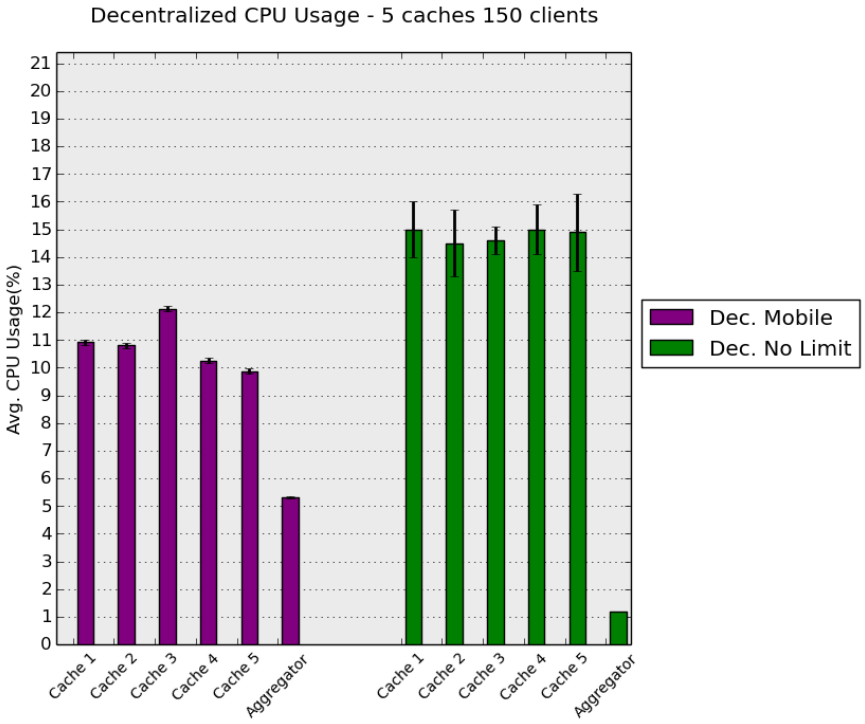


Figure 5.43: Street mobile scenario, decentralized model, Average CPU Usage (%)

It is possible to conclude that the distributed model always presents a higher CPU usage in the edge caches side, but presents a smaller CPU usage in the agregator side compared to the decentralized model, allowing the aggregator to handle a higher number of clients.

5.3.3 Consumer on a high speed train

This section shows and discusses results comprising exhaustive tests performed on the distributed model, considering that clients are inside the train. Due to the mobility of the train, there are zones with good and others with bad signal quality/reception and transmission. Figure 5.45 depicts the consumer on a high speed train.

In this scenario, it is assumed that there is no network connection between the aggregator and edge caches inside tunnels. Particularly, it is considered a hypothetical itinerary composed by a regular train line composed by a railroad track with 3 different tunnels size.

The edge caches are access points inside the train, responsible to offer network connectivity to the users, as well as to behave as the local video content providers. Regarding the scenario tested in this work, there is an edge cache per wagon. Therefore, concerning the performed

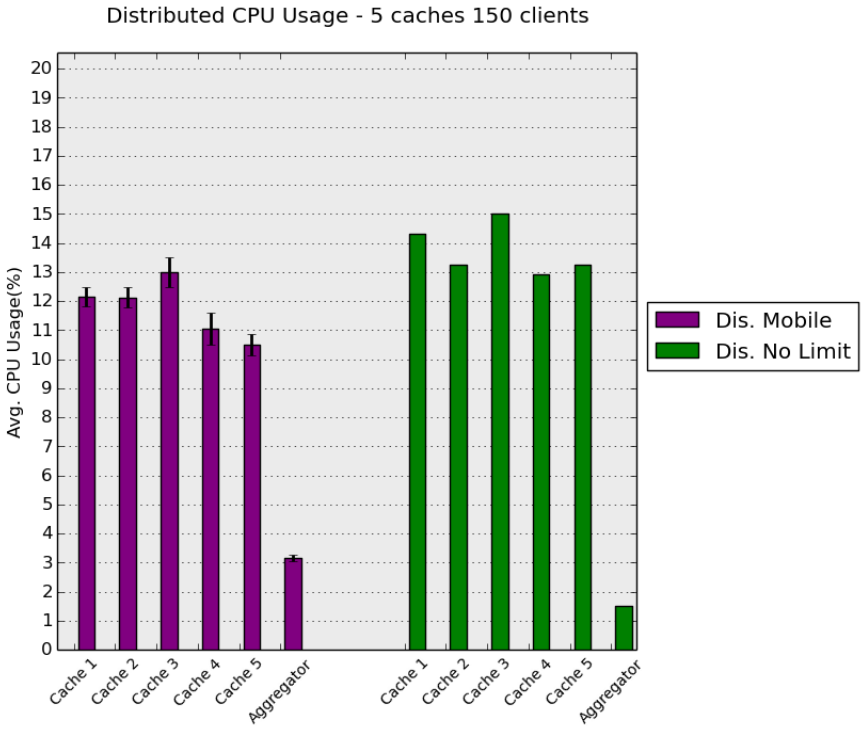


Figure 5.44: Street mobile scenario, distributed model, Average CPU Usage (%)

experiments in this scenario, the limitations are at the bandwidth level.

Figure 5.46 presents the available bandwidth on the train scenario with three tunnels. As can be seen in Figure 5.46, this scenario assumes the available bandwidth of 30Mbps between wagon and the infrastructure network in order to reach the origin server. Inside the wagon, i.e., the connection between the edge-cache and consumer, it is used a Wi-Fi 802.11c bandwidth.

The tunnels size is illustrated by the time without available bandwidth: 40 seconds near the 100s, 60 seconds near the 200s and 120 seconds between 400 and 500s. The tunnels are emulated by the tc/netem command, with a packet loss inside the tunnel of 100 %,i.e., emulating no connection between the aggregator and edge caches.

When the train and the consumers are in the tunnel, the use of a prefetching mechanism enables to download the requests in the good signal zones, that in the future, when the train will be in the tunnel, will be requested by the clients. Thus, it is possible to improve the consumers perspective about the visualization of the video.

In this scenario, experiments were performed with 2 clients distributed by two edge caches. QoE is measured for the two clients in order to analyze its variation comprising the limitations imposed on the train scenario. For the sake of the performed experiment, it is assumed that the train velocity allows the tunnel arrival to be informed in advance with enough time to be performed the content prefetching regarding the tunnel duration.

QoE

Figures 5.47a and 5.47b depict the results comprising the user at train scenario regarding

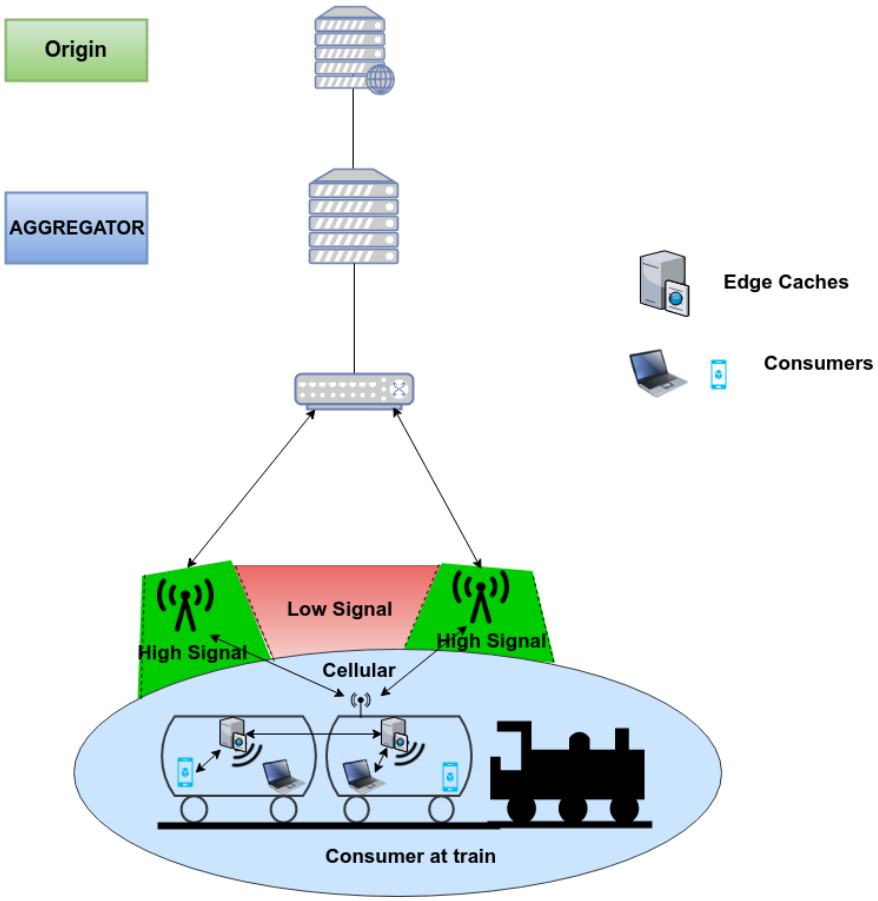


Figure 5.45: Train Scenario

QoE on the distributed model without and with prefetching mechanism, respectively.

It is observed that, without prefetching, probes freeze inside the tunnels, which is shown by the sharp fall of QoE. Also, it is observed that inside the tunnel the video does not freeze as soon as it enters the tunnel, but 30 seconds later, since the probe buffer size is about 30 seconds.

Moreover, in the Figure 5.47a, the QoE is good in the areas with good network signal quality, i.e., when the network supports the clients demand. But, when the network is not propitious to provide good data transmission conditions, the video freezes.

However, by the analysis of the Figure 5.47b, with the introduction of the prefetching mechanism, it is possible to surpass these limitations imposed by the tunnels. This happens because the video content is pre-cached while train passes by the areas with good signal quality. Thus, it is shown that, with the prefetching mechanism, the quality of experience does not decrease and the video does not freeze, improving substantially the clients quality of experience.

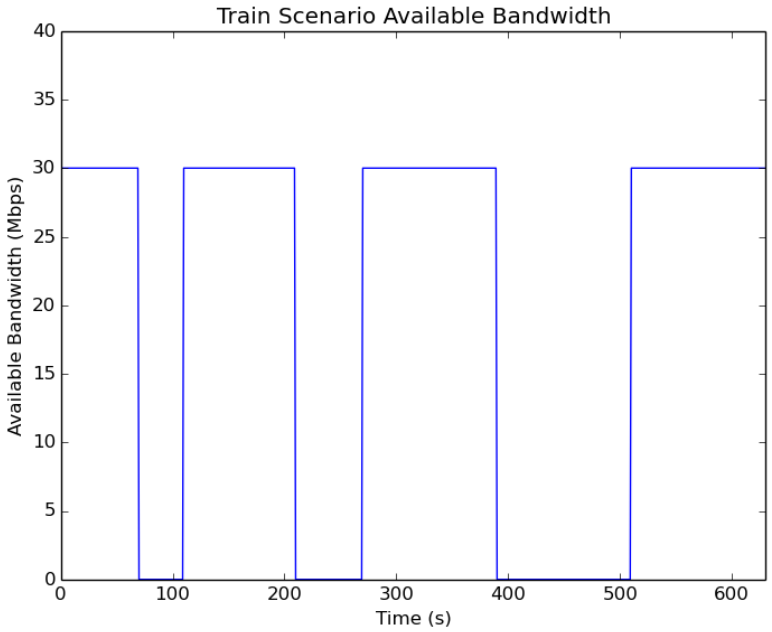
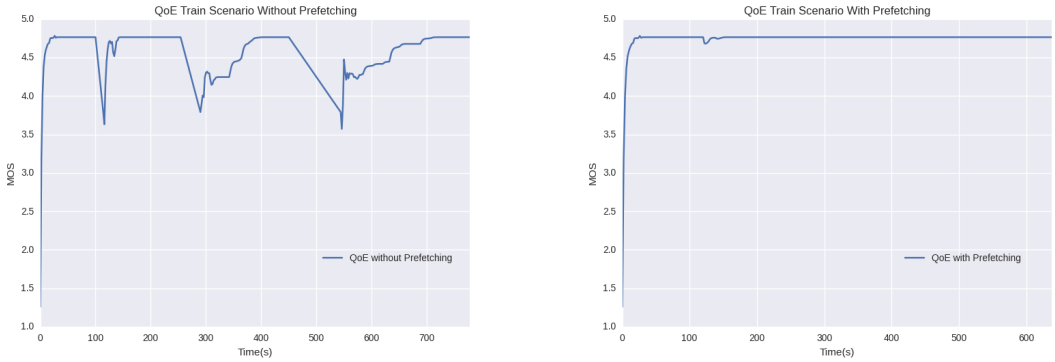


Figure 5.46: Train Scenario, Available Bandwidth



(a) Without prefecthing

(b) With prefecthing

Figure 5.47: QoE train scenario

5.4 Chapter Considerations

This Chapter focused on explaining how the whole approach presented in Chapter 4 is integrated, evaluating the several architectural models with the scenarios presented in Chapter 3.

With regard to the three architectural models, the proposed distributed model presented the best performance in load and consumed bandwidth in the upstream servers, as well as it presented best cache performance. However, the decentralized model presented a lower consumed bandwidth and load on the egde cache side.

Regarding the different scenarios, the consumer at home presents a lower consumed band-

width and load in the upstream servers, as well as it presents a better cache performance, since there is a content homogeneity. However, in the mobile consumer in the street, it is shown a lower consumed bandwidth and load on the egde cache side, since it provides a lower quality (bit rate) to the clients.

Also, regarding the consumer on a high speed train, the distributed model with the prefetching mechanism improved the clients QoE, since the video did not freeze inside the tunnels.

Chapter 6

Conclusions and Future Work

In recent years, the consumption of OTT multimedia services has been increasing and it is expected to continue. Thus, to support this trend it is important to surpass various limitations, namely the scalability and QoE, especially in wireless delivery content.

This dissertation proposed a distributed caching architecture for OTT content distribution, with prefetching mechanism integrated. This architecture relies on a content distribution model where caches are split in two intermediary tiers between consumers and content origin server. The proposed architecture is composed of several modules that deal with the content distribution among the edge caches, allowing to store content and thus increase the available size for caching, closer to the consumers. These modules are responsible to communicate among the edge caches in order to control the global cache formed by them, to distribute the content at this global cache, as well as to serve consumers.

The proposed architecture evaluation was performed taking into account its implementation on three architectural models:

1. Centralized, where there are no edge caches, just a proxy which forwards the requests;
2. Decentralized, where edge caches are independent from one and another;
3. Distributed, where edge caches interconnect themselves forming a group global cache;

Also, several experiments were performed comprising three scenarios:

1. A wired stable network suitable to consumer at home scenario;
2. A mobile scenario, comprising some conditions of mobile networks suitable for mobile consumers at the street, for instance;
3. Other mobile scenario, comprising some conditions of mobile networks suitable for consumers at high speed trains.

For the consumer at home scenario, the vertical and horizontal scalability was tested. In this scenario it was also tested the approach of different video popularities for the decentralized and distributed models and the edge cache weighted distribution for the distributed model. Moreover, for the mobile consumer at the street, tests were performed with the distributed and decentralized models and, for the consumer at the train, it was performed several tests with the distributed model, with prefetching mechanism.

Results regarding edge caches performance and bandwidth consumption show that the proposed architecture is well suited to the several scenarios, and outperforms all other approaches. Particularly in the mobile consumer at street scenario, QoE on far from base station consumers has been improved. Moreover, in mobile consumer at the train scenario with prefetching, QoE was significantly improved.

Finally, to support the growing demand of OTT services, an inevitably improvement in content delivery is needed. The cooperation among the several edge caches and predicting the users content can effectively provide a better QoE and reduce the bandwidth required to serve all the requests, saving aggregators or origin servers from content providers.

Future Work

Throughout this work, there are some aspects that need to be improved or developed. Noteworthy:

- *Real World Evaluation*: The various tests performed are implemented in a controlled network. However, in the future the tests could be performed in real equipment on a real life network, which is expected to produce more accurate results.

- *Test other types of adaptive streaming technologies*: It is a plan to test the proposed architecture with several types of streaming technologies, beyond Microsoft Smooth Streaming, such as Apple HTTP Live Streaming, Adobe HTTP Dynamic Streaming, MPEG-Dash.

- *Predicting areas with high and low signal*: The train scenario can be enforced in a real world predicting the areas with high and low network signal, through Global Positioning System (GPS), and with the prefetching mechanism. Moreover, learning approaches can be used to predict the expected QoE and the required prefetching and distribution to finally improve the QoE in a real-time approach.

Bibliography

[1] Mingfeiy, "How does video streaming work? ," 2017, Accessed: 04-2017. [Online]. Available: http://mingfeiy.com/wp-content/uploads/2012/06/ScreenShot145.jpg

[2] ——, "How does video streaming work? ," 2017, Accessed: 04-2017. [Online]. Available: http://mingfeiy.com/wp-content/uploads/2012/06/ScreenShot141.jpg

[3] ——, "How does video streaming work? ," 2017, Accessed: 04-2017. [Online]. Available: http://mingfeiy.com/wp-content/uploads/2012/06/ScreenShot143.jpg

[4] Bitmovin, "MPEG-DASH in a Nutshell ," 2017, Accessed: 04-2017. [Online]. Available: https://ox4zindgwb3p1qdp2lznn7zb-wpengine.netdna-ssl.com/wp-content/uploads/2016/04/adaptive-streaming-1024x350.png

[5] Microsoft, "Smooth Streaming Architecture," 2017. [Online]. Available: https://www.aspfree.com/c/a/silverlight/microsoft-silverlight-and-smooth-streaming-join-forces/

[6] Memcached, "Memcached Image," 2017, Accessed: 05-2017. [Online]. Available: https://memcached.org/about

[7] T. R. ITU-T, "ITU-T Recommendation P.910: Subjective video quality assessment methods for multimedia applications," Tech. Rep., 1999, 2000, Accessed: 03-2017. [Online]. Available: https://www.itu.int/rec/T-REC-P.910-199909-S/en

[8] R. M. H. R. Rajat Banerji, Abhishek, "Digital Media: Rise of On-demand Content," Deloitte, Tech. Rep., 2010.

[9] G. Petersen, "The Rise of OTT Video and Average Profit Per User ," Calix, Tech. Rep., 08 2016.

[10] KantarMedia, "Linear vs non-linear viewing: A qualitative investigation exploring viewersbehaviour and attitudes towards using different TV platforms and services providers," KantarMedia, Tech. Rep., 05 2016.

[11] J. Nogueira, "Over-the-top multimedia delivery: A catch-up tv case study," PhD thesis, University of Aveiro, 2017.

[12] J. Silva, "Content Distribution in OTT Wireless Networks," Master's dissertation, University of Aveiro, 2016.

[13] O. I. Forum, "Open IPTV Forum Release 2 Specification Functional Architecture," Open IPTV Forum, Tech. Rep., 2013.

[14] Netflix, "Netflix," 2017, Accessed: 05-2017. [Online]. Available: https://www.netflix.com

[15] Youtube, "Youtube," 2017, Accessed: 05-2017. [Online]. Available: https://www.youtube.com/

[16] N. Bouten, S. Latr, W. Van de Meerssche, B. De Vleeschauwer, K. Schepper, W. Van Leekwijck, and F. De Turck, "A multicast-enabled delivery framework for qoe assurance of over-the-top services in multimedia access networks," vol. 21, 12 2013.

[17] J. Abreu, J. a. Nogueira, V. Becker, and B. Cardoso, "Survey of catch-up tv and other time-shift services: A comprehensive analysis and taxonomy of linear and nonlinear television," *Telecommun. Syst.*, vol. 64, no. 1, pp. 57–74, Jan. 2017. [Online]. Available: https://doi.org/10.1007/s11235-016-0157-3

[18] L. A. Adamic and B. Huberman, "Rtsp audio and video streaming for qos in wireless mobile devices," in *IJCSNS International Journal of Computer Science and Network Security*, vol. 8, no. 1, 1 2008.

[19] R. L. H. Schulzrinne, A. Rao, "Real Time Streaming Protocol," RFC2326, 1998, Accessed: 04-2017. [Online]. Available: https://tools.ietf.org/html/rfc2326

[20] A. Begen, T. Akgul, and M. Baugher, "Watching video over the web: Part 1: Streaming protocols," *IEEE Internet Computing*, vol. 15, no. 2, pp. 54–63, March 2011.

[21] R. F. V. J. H. Schulzrinne, S. Casner, "RTP: A Transport Protocol for Real-Time Applications," RFC3550, 2003, Accessed: 04-2017. [Online]. Available: https://tools.ietf.org/html/rfc3550

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

[23] Apple, "HTTP Live Streaming," 2017. [Online]. Available: https://developer.apple.com/streaming/

[24] Adobe, "AdobeHTTP Dynamic Streaming," 2017. [Online]. Available: http://www.adobe.com/pt/products/hds-dynamic-streaming.html

[25] MPEG, "MPEG-DASH," 2017. [Online]. Available: https://mpeg.chiariglione.org/standards/mpeg-dash

[26] Microsoft, "Microsoft Smooth Streaming," 2017. [Online]. Available: https://www.iis.net/downloads/microsoft/smooth-streaming

[27] ——, "Silverlight," 2017. [Online]. Available: https://www.microsoft.com/silverlight/

[28] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, Jun. 2000. [Online]. Available: http://doi.acm.org/10.1145/358923.358939

[29] H. G. Sandhaya Gawade, "Review of algorithms for web pre-fetching and caching," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 1, no. 2, april 2012.

[30] B. Wu and A. D. Kshemkalyani, "Objective-optimal algorithms for long-term web prefetching," *IEEE Transactions on Computers*, vol. 55, no. 1, pp. 2–17, Jan 2006.

[31] E. P. Markatos and C. E. Chronaki, "A top-10 approach to prefetching on the web," *In Proceedings of the INET98 Internet Global Summit*, 1996.

[32] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin, "The potential costs and benefits of long-term prefetching for content distribution," *Computer Communications*, vol. 25, no. 4, pp. 367 – 375, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S014036640100408X

[33] J. Silva, A. Dias, J. Nogueira, L. Guardalben, and S. Sargento, "Content-aware prefetching in over-the-top wireless networks," in *2017 IEEE Symposium on Computers and Communications (ISCC)*, July 2017, pp. 515–522.

[34] M. Borkowski, O. Skarlat, S. Schulte, and S. Dustdar, "Prediction-based prefetch scheduling in mobile service applications," in *2016 IEEE International Conference on Mobile Services (MS)*, June 2016, pp. 41–48.

[35] J. Forlizzi and S. Ford, "The building blocks of experience: An early framework for interaction designers," in *Proceedings of the 3rd Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, ser. DIS '00. New York, NY, USA: ACM, 2000, pp. 419–423. [Online]. Available: http://doi.acm.org/10.1145/347642.347800

[36] T. R. ITU-T, "Recommendation P.10/G.100: Amendment 2: New Definitions for Inclusion in Recommendation ITU-T P.10/G.100," Tech. Rep., 2006, 2008, Accessed: 03-2017. [Online]. Available: https://www.itu.int/rec/T-REC-P.10-200807-S!Amd2/en

[37] A. Salvador, J. Nogueira, and S. Sargento, *QoE Assessment of HTTP Adaptive Video Streaming*. Cham: Springer International Publishing, 2015, pp. 235–242. [Online]. Available: https://doi.org/10.1007/978-3-319-18802-7_32

[38] J. Nogueira, D. Gonzalez, L. Guardalben, and S. Sargento, "Over-the-top catch-up tv content-aware caching," in *2016 IEEE Symposium on Computers and Communication (ISCC)*, vol. 00, June 2016, pp. 1012–1017. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ISCC.2016.7543869

[39] J. Liu and J. Xu, "Proxy caching for media streaming over the internet," *IEEE Communications Magazine*, vol. 42, no. 8, pp. 88–94, Aug 2004.

[40] L. B. Claudiu Cobarzan, "Further Developments of a Dynamic Distributed Video Proxy-Cache System," in *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on*, 03 2007, pp. 349–357.

[41] Community, "Popcorn Time," 2017, Accessed: 05-2017. [Online]. Available: https://popcorntime.sh/

[42] J. Li, "On peer-to-peer (P2P) content delivery," *Peer-to-Peer Networking and Applications*, vol. 1, no. 1, pp. 45–63, 11 2007.

[43] D. Ghosh, P. Rajan, and M. Pandey, "P2P-VoD Streaming:," in *Advanced Computing, Networking and Informatics- Volume 2*, ser. Smart Innovation, Systems and Technologies. Springer, Cham, 2014, pp. 169–180.

[44] Community, "Peer5," 2017, Accessed: 05-2017. [Online]. Available: https://www.peer5.com

[45] T. Sanguankotchakorn and N. Krueakampliw, "A hybrid pull-push protocol in hybrid cdn-p2p mesh-based architecture for live video streaming," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Sept 2017, pp. 187–192.

[46] G. K. Mikael Goldmann, "Measurements on the spotify peer-assisted music-on-demand streaming system," in *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, 10 2011, pp. 206–211.

[47] M. Ellis, S. D. Strowes, and C. Perkins, "An experimental study of client-side spotify peering behaviour," in *2011 IEEE 36th Conference on Local Computer Networks*, Oct 2011, pp. 267–270.

[48] Community, "Teleport," 2017, Accessed: 05-2017. [Online]. Available: http://teleport.media/

[49] B. Li, S. Xie, Y. Qu, G. Y. Keung, C. Lin, J. Liu, and X. Zhang, "Inside the new coolstreaming: Principles, measurements and performance implications," in *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, April 2008.

[50] G. Kreitz and F. Niemela, "Spotify – large scale, low latency, p2p music-on-demand streaming," in *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, Aug 2010, pp. 1–10.

[51] K. W. Hwang, V. Gopalakrishnan, R. Jana, S. Lee, V. Misra, and K. K. Ramakrishnan, "Abandonment and its impact on p2p vod streaming," in *IEEE P2P 2013 Proceedings*, Sept 2013, pp. 1–10.

[52] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000. [Online]. Available: http://dx.doi.org/10.1109/90.851975

[53] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/258533.258660

[54] D. G. Thaler and C. V.Ravishankar, "A name-based mapping scheme for rendezvous," *University of Michigan Technical Report CSE-TR-316-96*, 11 1996.

[55] S. Sanfilippo, "Redis," 2017, Accessed: 02-2017. [Online]. Available: https://redis.io/

[56] B. Fitzpatrick, "Memcached," 2017, Accessed: 09-2017. [Online]. Available: https://memcached.org/

[57] I. Krasnoshchok, "Cachelot," 2017, Accessed: 09-2017. [Online]. Available: https://cachelot.io/

[58] The Apache Software Foundation, "Apache Traffic Server," 2017, Accessed: 12-2017. [Online]. Available: http://trafficserver.apache.org

[59] NGINX Inc., "Nginx," 2017, Accessed: 12-2017. [Online]. Available: http://nginx.com

[60] Varnish Software, "Varnish Cache," 2017, Accessed: 05-2017. [Online]. Available: https://www.varnish-software.com/

[61] D. Wessels, "Squid," 1996, Accessed: 08-2017. [Online]. Available: http://www.squid-cache.org/

[62] I. Krasnoshchok, "CachelotScript," 2017, Accessed: 09-2017. [Online]. Available: https://github.com/cachelot/cachelot/blob/master/test/memory_consumption.py

[63] P. Felber, P. Kropf, E. Schiller, and S. Serbu, "Survey on load balancing in peer-to-peer distributed hash tables," *IEEE Communications Surveys Tutorials*, vol. 16, no. 1, pp. 473–492, First 2014.

[64] Python, "Python," 2017, Accessed: 04-2017. [Online]. Available: https://www.python.org/

[65] L. A. Adamic and B. Huberman, "Zipf's law and the internet," in *Glottometrics,To honor G.K. Zipf*, vol. 3, 11 2002, pp. 143–150.

[66] VMware, "VMware," 2017, Accessed: 09-2017. [Online]. Available: https://www.vmware.com/

[67] Microsof, "Windows Server 2012," 2017, Accessed: 09-2017. [Online]. Available: https://imagine.microsoft.com/en-us/Catalog/Product/77

[68] Microsoft, "Microsoft Smooth Streaming," 2017, Accessed: 09-2017. [Online]. Available: https://www.iis.net/downloads/microsoft/smooth-streaming

[69] Ubunto, "Ubuntu 14.04.5 LTS," 2017, Accessed: 09-2017. [Online]. Available: http://releases.ubuntu.com/14.04/

[70] I. Sysoev, "Nginx," 2017, Accessed: 09-2017. [Online]. Available: https://www.nginx.com/

[71] Community, "Netem," 2017, Accessed: 09-2017. [Online]. Available: https://wiki.linuxfoundation.org/networking/netem

[72] M. Clyne, "Nginx Development Kit," 2017, Accessed: 09-2017. [Online]. Available: https://github.com/simpl/ngx_devel_kit.git

[73] O. I. Yichun Zhang, "ngx_set_misc," 2017, Accessed: 09-2017. [Online]. Available: https://github.com/openresty/set-misc-nginx-module

[74] ——, "ngx_memc," 2017, Accessed: 09-2017. [Online]. Available: https://github.com/openresty/memc-nginx-module

[75] C. I. Yichun Zhang, "ngx_srcache," 2017, Accessed: 09-2017. [Online]. Available: https://github.com/openresty/srcache-nginx-module.git