Advanced contributions in HTTP adaptive streaming

Nassima Bouzakaria

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2017-ENST-0037

EDITE – ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

Télécom ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Nassima BOUZAKARIA

le 25 Juillet 2017

Contributions au streaming HTTP adaptatif avancé

—

Advanced contributions in HTTP adaptive streaming

Directeur de thèse : **Cyril CONCOLATO**
Encadrant : **Jean Le FEUVRE**

Jury

M. Simon GWENDAL, Maître de Conférences HDR, IMT Atlantique, Cesson-Sévigné	Rapporteur
M. Vincent CHARVILLAT, Professeur, Université de Toulouse, Toulouse	Rapporteur
M. Dario ROSSI, Professeur, Télécom ParisTech, Paris	Examinateur
Mme. Anissa MOKRAOUI, Professeur, L2TI Institut Galilée, Université Paris 13, Villetaneuse	Examinateur
M. Cyril CONCOLATO, Maître de Conférences HDR, Télécom ParisTech, Paris	Directeur de thèse
M. Jean LE FEUVRE, Ingénieur d'Étude, Télécom ParisTech, Paris	Encadrant

THÈSE

Télécom ParisTech
Grande École de l'Institut Télécom - membre fondateur de ParisTech
46 rue Barrault — 75634 Paris Cedex 13 — Tél. +33 (0)1 45 81 77 77 — www.telecom-paristech.fr

Acknowledgments

This work would not have been possible without the help and support of many people to whom I am really grateful.

First and foremost, I am deeply grateful to my advisors, Cyril Concolato and Jean Le Feuvre, for their time, support, guidance and encouragement. I would like to thank all members of the Image, Data, Signal department at Telecom ParisTech for the excellent scientific environment they provide. To my colleagues and friends, thank you for listening and supporting me especially during the stressful moments.

Last, but not the least, I owe my tremendous gratitude to my parents, my two sisters and my brother for their love, moral support and constant encouragement.

Résumé

Le streaming adaptatif HTTP est une technologie récente dans les communications multimédia, utilisant notamment le standard MPEG-DASH. L'un des principaux problèmes dans le déploiement des services de streaming en direct est la réduction de plusieurs types de latence tel que le délai de démarrage et la latence de bout en bout. Dans cette thèse, nous abordons le problème de ces latences dans les services de streaming en direct utilisant MPEG-DASH.

Tout d'abord, nous examinons les causes du délai de démarrage dans les systèmes MPEG-DASH et les stratégies communes pour réduire ce délai. Nous proposons une nouvelle méthode basée sur HTTP / 1.1 qui est compatible avec les infrastructures Web existantes.

Deuxièmement, nous étudions les principaux composants qui sont à l'origine de la latence totale, nous proposons un système de streaming en direct à faible latence.

Troisièmement, nous montrons comment un système de streaming en direct utilisant MPEG-DASH et à faible latence peut être combiné avec un système utilisant un réseau Broadcast. Notre approche proposée garantit la synchronisation des deux contenus transmis via deux réseaux de distribution DASH et broadcast.

Abstract

HTTP adaptive streaming is a recent topic in multimedia communications with on-going standardization activities, especially with the MPEG-DASH standard which covers on demand and live services. One of the main issues in live services deployment is the reduction of various latencies, the initial delay before the playback and the overall end-to-end latency. In this thesis, we address the problem of these latencies in live DASH streaming.

First, we review the causes of startup delay in DASH and common strategies used to reduce this delay. We propose a new method based on HTTP/1.1 and compatible with existing caching and delivery infrastructures for reducing the initial setup of an MPEG-DASH session.

Second, we investigate the major contributor components to the end-to-end latency. We propose a complete novel low latency live DASH streaming system.

Third, we show how such a low latency live DASH system can be used to enable combined broadcast and broadband services while keeping the client buffering requirements on the broadcast link low. Our proposed approach insures two functionalities: synchronization of both contents delivered through different distribution networks and keeping the client buffering requirements on the broadcast link low.

Table of Contents

List of Figures

List of Tables

List of Acronyms

IP	Internet Protocol
QoS	Quality of Service
QoE	Quality of Experience
VoD	Video on Demand
NALU	Network Abstraction Layer Unit
AU	Access Unit
RTP	Real-time Transport Protocol
TS	Transport Stream
ISOBMFF	ISO Base Media File Format
UDP	User Datagram Protocol
PES	Packetized Elementary Stream
RAP	Random Access Point
IDR	Instantaneous Decoding Refresh
B frame	Bidirectional frame
GOP	Group Of Pictures
MTU	Maximum Transmission Unit
TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol

Chapter 1

Introduction

Contents

1.1 Context

The popularization of convenient and advanced video capture and production technologies, the evolution of video delivery systems and the emergence of different video display devices have created a larger number of multimedia applications including video streaming that deeply stepped into people's daily life. The rise of popularity of video streaming services resulted in increased volumes of video contents. Today, video delivery accounts for 64% of the majority of Internet traffic [3]. It is expected to grow to 80% by 2019. This increasing demand for video services has changed viewer expectations of quality.

Over-The-Top (OTT) video streaming has become a cost-effective means for video delivery nowadays since it relies on the open unmanaged Internet. In contrast to managed newtork where the operator ensures a high level of service quality. OTT delivery offers a viewing freedom since it is not limited to the PC screen, but extends to any connected device, e.g. connected TV, gaming consoles, smartphones, connected tablets, etc. Viewers can benefit from Video on Demand (VoD) and live streaming services provided by OTT. Compared to VoD, live streaming is gaining popularity, especially for watching live sports in the case of globally popular events.

The experienced quality of live streaming over OTT is commonly compared to live broadcasting experience when delivered over traditional distribution systems, such as

managed digital cable, terrestrial and satellite broadcast networks as well as managed IPTV networks. The challenges raised when delivering a TV-like experience for live OTT system have inspired many contributions and yet many issues remain open. In current deployments, live OTT suffers from much higher latencies, typically from a few seconds up to half a minute compared to the managed broadcasting services. This latency is defined as the time difference between the instants when the live event occurs and when it is played back to the viewer. This problem is not convenient when someone is watching for example a football match, and may surprisingly hear his neighbors cheering over a scored goal, which he will only see on his screen after several seconds, due to the delay introduced by OTT streaming. As a consequence, the live latency is thus becoming an important factor impacting the overall quality experienced by viewers. Another type of latency issue in live OTT is the startup delay, which is the time difference between the moment when a viewer clicks on the "Play" button and the moment the video starts playing. In other words, it is the time needed to download and to buffer all necessary information for the initial playback. Viewers are highly impatient and less tolerant of slow starts in live streams compared to VoD [4]. Larger startup delay increases the risk that viewers churn or abandon the video stream.

Moreover, very low latency steaming is required for interactive or bidirectional applications such as video conferencing, live streaming with voting or telemedicine. Such applications are characterized by very strict delay constraints. Another use case where low latency is undeniably important is the hybrid delivery scenario where the OTT (broadband) and broadcast networks are combined to enhance the broadcast service with premium services, e.g. subtitles, gestures, languages, via broadband. The latency of the OTT system should be lower than the broadcast for live capture/generation. Otherwise, additional buffers are required to synchronize the broadcast content with the broadband content.

In recent years, HTTP adaptive streaming has emerged as the technology of choice for the delivery of OTT services. It enables dynamic adaptation of video quality to varying network bandwidth and client's device capabilities, by choosing among several profiles (versions of a stream encoded with certain bitrate and quality level) available on the server. In this trend, a new standard called MPEG Dynamic Adaptive Streaming over HTTP (DASH) was developed and is used worldwide [1]. Some research works have been proposed for reducing the latencies listed above, in HTTP adaptive streaming. However, the latency is still in the order of seconds.

The goal of this thesis is to propose new approaches to reduce the startup delay as well as the end-to-end latency observed by viewers when using live DASH streaming. We target a very low latency, i.e. latency in the order of frames (e.g. less than 200 ms), to deliver a successful live OTT experience.

1.2 Summary of Contributions

This thesis presents contributions related to live video streaming services using DASH. We organized them into three themes: startup, delivery and applications.

- Live DASH Streaming Startup

  Two contributions in this field were proposed. The first one consists in an analytical evaluation of the different existing strategies that a DASH client can use to start a video streaming session. The second one involves three methods to reduce the startup delay in live DASH streaming. All methods are based on the idea that the starting phase of a DASH session should not require multiple round-trips between the client and the server. These approaches make a client capable of retrieving the necessary information to start the initial playback using only one HTTP request and HTTP response. The proposed methods have been designed to have no negative impact on the existing caching and delivery infrastructures.

- Live DASH Streaming Content Delivery

  Regarding this category, our main contribution consists in proposing a complete novel low latency live DASH streaming system. It aims at reducing the end-to-end latency in live DASH. The three major functions, including content preparation, content distribution and content display of the live DASH streaming system are modified as follows. First, we make the content preparation process (i.e. content segmentation process) that is in charge of segmenting the content into several segments progressive without changing the encoding process. For that, we divide each segment into multiple small parts that we push immediately to the web server. Second, we make the client capable of sending out a request once some parts of the segment content are available. Third, we develop an intelligent web server that can send available data parts to client before the media segment is fully ready and published. Finally, we modify the client to be able of receiving and processing incomplete segments, i.e. those data parts.

- Live DASH Streaming Applications

  Hybrid delivery is one of the multimedia applications that requires a low latency delivery especially the broadband side. Our contribution consists in combining broadcast and broadband delivery services for very low latency, for quality enhancement for example through scalable codecs. We investigate a scenario where we consider a basic content delivered over traditional broadcast channels enhanced with an additional content delivered over broadband networks using DASH. Our proposed approach insures two functionalities: synchronization of both contents delivered through different

distribution networks and keeping the client buffering requirements on the broadcast link low. For that, we use the low latency live DASH streaming system that we have described in the previous contributions.

1.3 Thesis Organization

This manuscript is organized into three distinct parts. The first part presents in detail the fundamentals of video streaming over IP-based networks. Chapter 2 describes the factors impacting the perceived quality and presents the video streaming chain with a detailed overview of its components. It ends with an overview of the basic concepts of HTTP adaptive streaming system.

The second part describes our contributions for the delivery of live streaming video content using MPEG-DASH. Chapter 3 introduces our approaches related to the improvement of the starting of live DASH streaming. It begins with an analytical evaluation of the different DASH client strategies for starting a streaming session. Then, we propose three methods to reduce the startup delay. The methods were deployed and evaluated over different networks and using different versions of the HTTP protocol. The obtained results are reported and analyzed. Chapter 4 presents our contributions for the low latency live DASH streaming service. We describe our proposed system to reduce the end-to-end latency. Finally, we validate this low latency system by some experimentations using live and interactive streaming services.

The third part exposes our contributions on the hybrid delivery application. Chapter 5 starts with a summary of the main issues and challenges that we have identified in the delivery of media content over hybrid broadcast/broadband networks. Next, we experiment and evaluate our proposed hybrid delivery system when using a multi-resolution content use case. Finally, we conclude the chapter after reporting the obtained results.

We end this manuscript with a summary of the proposed methods and their associated results, as well as some future works of this thesis in Chapter 6.

1.4 List of Publications

Conference papers

- N. Bouzakaria, C. Concolato and J.L. Feuvre, “Overhead and performance of low latency live streaming using MPEG-DASH”, Proceeding of *The 5th International Conference on Information, Intelligence, Systems and Applications (IISA)* , Crete, Greece, July 2014.

- N. Bouzakaria, C. Concolato and J.L. Feuvre, "Fast dash bootstrap", Proceeding of *IEEE 17th International Workshop on Multimedia Signal Processing (MMSP)* , Xiamen, China, October 2015.

- J.L. Feuvre, C. Concolato, N. Bouzakaria and V. T. Nguyen, "MPEG-DASH for Low Latency and Hybrid Streaming Services", Proceeding of *The 23rd ACM International Conference on Multimedia (ACM MM)* , Brisbane, Australia, October 2015.

Contributions to Standardization

- C. Concolato, J. Le Feuvre and N. Bouzakaria, "Data URLs in MPD", *Moving Picture Experts Group (MPEG)*, Geneva, Switzerland, May 2016, n° M38649

- C. Concolato, J. Le Feuvre and N. Bouzakaria, "Guidelines for DASH Fast Start", *Moving Picture Experts Group (MPEG)*, Geneva, Switzerland, October 2015, n° M37254.

- C. Concolato, J. Le Feuvre and N. Bouzakaria, "Use of HTTP/2 Push for DASH Bootstrap", *Moving Picture Experts Group (MPEG)*, Geneva, Switzerland, October 2015, n° M37255.

Chapter 2

Video Streaming Over IP: Quality of Experience and HTTP Adaptive Streaming

Contents

2.1 Introduction

The work done in this thesis is aimed at proposing new video delivery approaches for MPEG Dynamic Adaptive Streaming over HTTP (MPEG-DASH). We begin this thesis

manuscript by a state of-the-art chapter that covers the fundamentals of video streaming over IP-based networks that are useful in our research.

In this chapter, we first give an overview of the different expectations of viewers in terms of Quality of Experience (QoE) when using a video streaming service in Section 2.2. We then present a typical video streaming chain in Section 2.3. On the first hand, basic concepts of the chain's components are reviewed. On the second hand, the technical features and functions of each component are analysed and discussed from the QoE point of view. Finally, we review the HTTP adaptive streaming system in Section 2.4.

2.2 Quality of Experience In Video Streaming

Video streaming over IP-based networks is exposed to various types of network impairments that can occur along the communication paths. IP does not guarantee any particular timeliness of delivery, or that a packet will be delivered at all. IP packets may be lost, reordered, delayed, duplicated, or corrupted. IP itself does not attempt to correct these problems that can severely deteriorate the video quality as perceived by viewers. Usually, service providers who use such networks for video streaming deploy systems to remove those problems and insure that their viewers receive adequate quality at all times. They assess the Quality of Service (QoS), i.e. the quality of the network delivery, through network performance analysis mechanisms.

QoS assessment addresses the different challenges associated with the accurate measurement or estimation of network level parameters such as bandwidth, one way delay [5], packet loss [6], or jitter [7]. However, these QoS parameters are typically used to indicate the impact on the video quality from the network performance and data transmission point of views, but do not reflect the viewer's perception. They are not sufficient for measuring the quality experienced by viewers.

Thus, Quality of Experience (QoE) [8] was introduced to overcome the limitations of QoS parameters. It concerns the aspects that are related to the human perception. It is based on human auditory and visual systems and relates to the perceived auditory and visual experience of the viewer with the contents. This manuscript focuses on the visual component of the QoE. The methodologies used for video QoE assessment relies on set of metrics, which must be able to assess the viewer satisfaction with the contents played on its device. We can classify QoE metrics into two groups: metrics to assess the quality of the perceived video content and metrics to evaluate the quality of the provided service. QoE metrics have different relative importances depending on the delivered service type. For a given video streaming session, QoE assessment indicates if the viewers are satisfied or disappointed.

In this section, we want to review the different expectations of viewers in terms of QoE in video streaming using IP networks. For that, we first review the different types of video streaming services. Then, we describe the different phases of a video streaming session that may impact the viewer's perception. Finally, we highlight the most relevant QoE metrics along with the different QoE assessment methods.

2.2.1 Streaming Service Types

Video streaming offers different types of services, namely, Video on Demand (VoD), live streaming, and real-time interactive streaming. We discuss each service type separately.

- VoD is a type of service that allows viewers to request a pre-recorded video content from a server at any time, rather than having to watch it at a specific live time. There is a broad diversity of VoD service models that are detailed in [9]. Interactive VoD (IVoD) is getting more important because it offers the same Personal Video Recorder (PVR) functions of modern TV systems such as: Play/Resume (i.e. start a playback from the beginning or resume it after a temporary stop), Pause (i.e. pause a playback from a few seconds up to several hours), Stop (i.e. stop the playback of a video), Fast/Slow Forward (i.e. browse through a video in the forward direction at a faster or a lower speed than standard forward), Fast/Slow Rewind (i.e. browse a video backwards at a faster or a lower rate than the standard rewind), Jump Forward/Backward (i.e. jump to a particular time in a video in a forward or a backward direction), etc. These functionalities allow viewers to interact and control the content being watched.

- Live streaming is a type of service that delivers content over the network as it is captured to viewers, for instance live sports or live news events. In live streaming, viewers are limited to the interactive functions that perform on past content. The capacity to watch live content from minutes in the past, up to the beginning of the program, is called time-shifting. Catch-up is one of the most advanced time-shifting services that offers a way to watch live programs through VoD service for a period of days after the original event. It is worth noticing that catch-up services are widely used for live TV contents where the viewers are allowed to catch up on TV programs (e.g. a live football game) that have been missed. For more detail, an exhaustive overview of time-shifting services is provided in [10].

- Real-time interactive streaming is a type of service in which the viewer interactions can impact the production of the live content at the source. The form of the interaction can vary. Video conferencing and cloud gaming are the most common applications of

real-time interactive service. Video conferencing is simply a telephone call with added video so that participants can see each other as well as hear each other. Cloud gaming renders an interactive gaming application remotely in the cloud and streams the scenes as a video sequence back to the player over the Internet. A cloud gaming player interacts with the application through a client, which is responsible for displaying the video from the cloud rendering server as well as collecting the player's commands and sending the interactions back to the cloud. Other interactive applications include telemedicine and remote surgery, where a surgeon in one location is able to perform surgery in another location over the network using remote-control robots and a video feed.

Such applications are characterized by very strict delay constraints compared to live streaming. The term "interactive" in the real-time interactive streaming service does not refer to the same notion as interactive in VoD and live streaming where the viewers interact with the content. It rather means interactivity with the application or between participants.

The contributions of this thesis are mainly targeting live streaming and real-time interactive streaming services.

2.2.2 Streaming Session States

In this section, we propose a description of the multiple states of a video streaming session that are perceived by a viewer. We identify as well, the states that may impact the viewer's QoE.

Viewer actions and the underlying network conditions move the video streaming session from one state to another. As shown in Figure 2.1, a video player is initially in the idle state. It transits to the *Starting* state when it starts displaying the first loaded video data on the viewer's screen. This can be initiated by a viewer, for example, pressing the "Play" button of a video player. The *Starting* state may affect the viewer's perception, especially if there is a long time between the viewer's action and the start of the initial playback.

When the playback can start, i.e. when the buffer is filled sufficiently, a video player enters into the *Playing* state to continue the video playback. If packet losses, delay fluctuations, or bandwidth drops occur in the network transmission the client's reception rate may drop below the client's consumption rate. Hence, the client buffer starts draining which may result in a buffer underflow. In this case, a video player enters into the *Paused* state where the viewer experiences an interruption. Once the buffer is replenished sufficiently, a video player moves in the *Playing* state to resume the video playing. The duration of the *Paused* state and the number of transitions beween the *Paused* and *Playing* states may impact the viewer QoE.

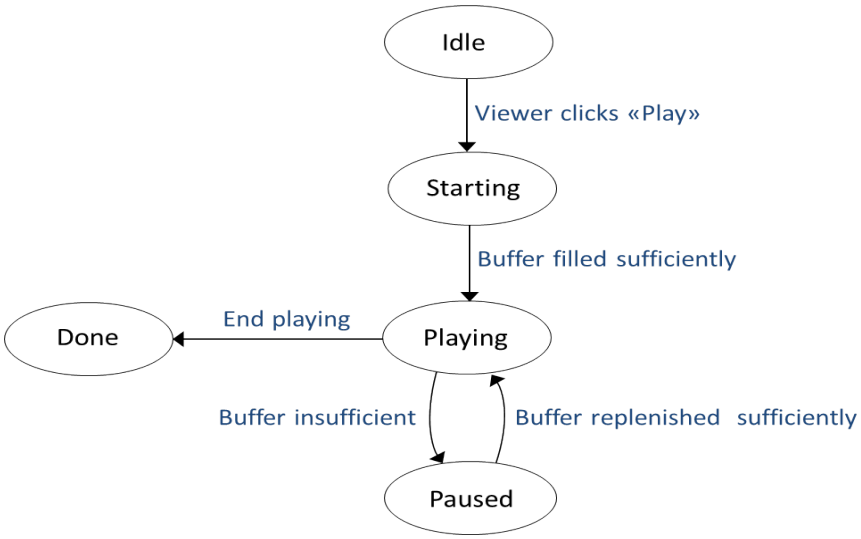


Figure 2.1 – Video streaming session states.

The *Paused* state can be avoided or at least reduced by a video player if a video streaming system is adaptive. In an adaptive video streaming system, several bitrates/qualities encodings can be available for the same video content on a server or potentially encoded on-the-fly. The decision to switch between bitrates and qualities can be made either by a client or a server.
To prevent the buffer underflow event when network conditions become bad, a client or a server adapts the bitrate to the underlying network bandwidth by choosing a lower video bitrate encoding compared to the current one. The buffer is then filled with the new low video bitrate encoding and the player continues providing an uninterrupted playback in the *Playing* state.

During the video playback process, a viewer could initiate actions such as pause, forward, rewind, etc. to control and interact with the content. When the playback of a video data ends, a video player goes to the *Done* state. It is worth noticing that a viewer can end voluntarily a video playback either before the video startup or after watching some portion of it, for example, clicking the ”Stop” button of a video player, in particular when he is not satisfied with the content or with the QoE.

2.2.3 Quality of Experience Assessment

In this section, we give an overview of the different QoE assessment metrics that can evaluate the quality of a video streaming session from the viewer's point of view. A complete QoE assessment considers not only the factors that affect directly the quality of the perceived video content by viewers, but it also takes into account how viewers perceive the overall quality of the delivered service. QoE assessment relies on a set of metrics that we classify

into two categories: video content quality metrics and service performance metrics, that we explore below.

Video Content Quality Metrics

Video content quality metrics consider factors that directly affect the quality of the received video content. Opposed to the classical QoS metrics, which are mostly network centered, video content quality metrics are independent from how the video was delivered. They depend on several video parameters like encoding bitrate, video resolution, framerate, dropped frames, and used video codec. They are based on two main video quality assessment methodologies, namely subjective and objective.

Subjective assessment [11] relies on an accurate and repeatable approach to estimate how video streams are perceived by viewers, i.e. what is their opinion on the quality of a particular video. The most common used subjective metric is Mean Opinion Score (MOS) that a viewer utilizes for the evaluation of each video by selecting a score from a quality scale that ranges from 1 to 5 (i.e. bad to excellent) [12]. The minimum threshold for an acceptable video content quality corresponds to a MOS of 3.5 [13].

Objective video quality assessment is based on mathematical models (algorithms) to measure and estimate the quality of a video. These algorithms are classified into three approaches depending on the amount of reference information they require during the video quality assessment: Full-Reference (FR), No-Reference (NR), and Reduced-Reference (RR). They are explained in detail in [11]. There is a broad diversity of objective metrics referred in [14], that can be used to generate a quantitative measure of the video quality. Peak Signal to Noise Ratio (PSNR) is the most used objective quality metric among the FR assessment approach. It makes pixel-by-pixel comparison between the reference (original) and decoded content to detect content distortions. Content with higher similarity will result in higher PSNR values (above than 37 dB reflects an excellent video content quality) [14].

Service Performance Metrics

Service performance metrics consider the factors that influence the performance and the usability of a delivered service. We describe in the following the most relevant metrics for the assessment of the performance of video streaming services.

- Startup delay is defined as the time span since the viewer queries the system about a specific video content, for instance, by clicking the ”Play” button of a video player, until the video is rendered on the viewer’s screen (i.e. until the video player transits to the *Starting* state to begin the first playback).

- Rebuffering duration and frequency are two metrics that can be used when rebuffering events occur. The rebuffering duration measures the total time spent in the filling of the client's buffer during the total video streaming session. It corresponds to the duration of the *Paused* state. The rebuffering frequency measures how frequent the viewer experienced a rebuffering event during the total video streaming session. It corresponds to the number of video player's transitions beween the *Paused* and *Playing* states. There is a trade-off between the rebuffering duration and the risk of shortly recurring interruption events.
- End-to-end latency is defined as the time span from when the image is captured (image acquisition) until the image is rendered on the viewer's screen (image playout).
- Response delay is the time elapsed between when a viewer performs an action impacting the content generation at the source and when he/she views the result.

The goal of our contributions is the optimization of the quality of the delivered service, specifically live streaming and real-time interactive streaming. For that, we target in our work two essential service performance parameters, the startup delay and the end-to-end latency. Our main contributions consist in reducing these delays to achieve a fast startup and a low latency video delivery, which can enhance the viewer QoE.

2.2.4 Viewer Behaviors and Expectations

Viewers may behave differently during a video streaming session depending on the network QoS, the quality of the video content, the type of service, the real-time requirements of a service. In this respect, three categories of viewer behaviors can be distinguished [15].
The first category is viewer abandonment where a viewer abandons the video stream after watching some portion of it, or even before it starts playing. This way of behaving may reflect the frustration of a viewer and the failure of a session. The video abandonment can be either forced when errors and problems occur in a system, or voluntarily when the offered QoE of a streaming session does not fit the viewer expectations.
The second category is viewer engagement which relates to the amount of time a viewer watches a video. If a viewer plays a video completely, it may reflect its satisfaction and the success of a session.
The third category is repeated viewing that refers to the behavior of viewers over longer periods of time. It consists in the viewers, who after watching videos on a content provider's site return after some period of time to watch more. For instance, a viewer who experienced a failed viewing is less likely to revisit the site to view more videos within a specified time period than a similar viewer who did not experience a failed viewing.

Based on existing research works, we review the viewer expectations in terms of QoE for video streaming by identifying QoE metrics impacting the most the viewer engagement, across different content genres and service types:

- In the case of VoD services, videos can be divided into short videos that have a duration of less than 15 minutes and long videos that have a duration of more than 15 minutes [4]. Examples of short video consist of news clips, trailers, and short interviews. In contrast, long video includes movies, episodes, and programs. The viewer requirements for VoD service are not very tight. For instance, the end-to-end latency has no interest because the viewer is watching a video which is pre-recorded and stored on the server.
  In VoD services, the response delay is not applicable because the viewer can interact and control the content being watched but it cannot modify its production at the source.
  Regarding the startup delay, high values up to several seconds in this metric can be frustrating for the viewers but they usually tolerate it, especially if they intend to watch a long video. For long and short VoD, a wait time of 2 s or less before beginning the playback does not have a large effect on viewers. However, [4] reports that 4.20% of viewers viewing a long VoD content abandon between 2 s and 3 s of waiting whereas 5.70% of viewers abandon requesting a short content. On the other side, short startup delay might be desirable for user-generated content where the viewers start many videos but watch only the first seconds, in order to search for some contents they are interested in [16] [17].
  As expected, rebuffering has a relevant impact on viewer behavior compared to the startup delay. Approximately 90% of viewers prefer waiting longer before the video consumption starts than experiencing unexpected stalling within the service [18]. For long and short VoD, rebuffering duration is the most important service performance metric [19]. In the experiments made in [4], 35% of long VoD viewers experience buffering compared with 30% for short VoD. The time spent in the rebuffering for short and long VoD is 6.5 min and 6.8 min respectively for 90 minutes of viewing sessions [4]. For the rebuffering frequency, [19] indicates that one rebuffering event is prefered over frequent rebuffering that may be annoying to viewers.
  A viewer is interested in the quality of the rendered video content. It depends mostly on the used bitrate and resolution in the video encoding. Video resolution has an impact on viewing. In long and short VoD, a high resolution is viewed approximately 26.5% longer than the low resolution [4].

- Live streaming has tighter constraints than VoD. Regarding the startup delay, viewers are less tolerant to slow starts in live streams. According to [4], more than 18% of

viewers requesting a live stream abandoned between 2 s and 3 s of waiting before the video starts, which is more than 4 times higher than long VoD, but a wait time of 2 s or less does not have a large impact on viewers.
The end-to-end latency has a significant effect on viewers if they compare their live service relying on IP networks to the existing live TV broadcast systems that are able to deliver the same content with a constant latency in the order of 6 s.
In live streaming, the buffer should be shorter than in VoD to ensure that the live stream is received by viewers with a small latency. However, the use of small buffers increases buffering events because there is little time to recover when the bandwidth fluctuates. According to [4], 48% of live streams experience buffering. The time spent in the buffering for live streams is 10.8 min for a 90 minutes of viewing session [4]. An increase of the buffering of only 1% can lead to more than 3 minutes of reduced viewer engagement [19].
In contrast to VoD, live viewers do not require but only prefer high video content quality if it is available.
Response delay is not applicable to live streaming scenarios where interactive actions such as catchup and rewinding video do not change the content at the source.

- In real-time interactive streaming service, viewers are more sensitive to delay than in the other services. Limiting end-to-end latency is very important. In video conferencing for instance, long delays make the interaction between participants difficult, because each participant has to wait for the signals to reach him to see if he can begin speaking or not. A widely recognized maximum delay limit for each direction in a two-way videoconference is 150 milliseconds. After display start, the playout must be continuous without any interruptions. Rebuffering is not acceptable at all. Otherwise, any late media data will be discarded.

Based on the previous analysis, we summarize in Table 2.1 the viewer expectations depending on the offered service type. It can be observed that viewers impose different constraints for each service type. We can note that rebuffering constantly has the highest impact on viewer engagement across all types of service. The time spent before the playback starts and the end-to-end latency have an important impact on the viewer experience for live streaming and real-time interactive streaming services. We also see that almost all metrics play a significantly more important role in the case of real-time interactive than live and VoD services.

Viewer expectations	VoD	Live streaming	Real-time interactive streaming
Short startup delay	Medium	High	High
Free / infrequent rebuffering	High	Highest	Highest
Low end-to-end latency	Low	Medium	High
Low response delay	Not applicable	Not applicable	High
High video content quality	High	Medium	Medium

Table 2.1 – Viewer expectations depending on the delivered service type.

2.3 Selected Features of a Video Streaming Chain

In this section, we want to identify and explain the causes of delay experienced by viewers in video streaming services. For that, we first present the architecture of a typical video streaming chain. Then, we review its components and their features. For each component, we identify the major features which may contribute to the end-to-end latency. Finally, we show how the viewer requirements (i.e. low end-to-end latency) in live streaming and real-time interactive streaming services may impact and drive the setting of these components.

2.3.1 Overview of a Video Streaming Chain

Figure 2.2 depicts the different key components with which a typical architecture for video streaming can be built. Audiovisual content is captured by a camera device, encoded and then passed to a packager (sometimes called a segmenter) in order to generate files or packets suitable for delivery protocols and networks. The packaged streams are directly delivered over the network or forwarded to a server where they can be accessed. A client can request and receive the content through the IP network, which is then decoded and displayed on the viewer's device. The content can be buffered before invoking the decoding process. Each component in the chain introduces a delay as shown in Figure 2.2. An analysis

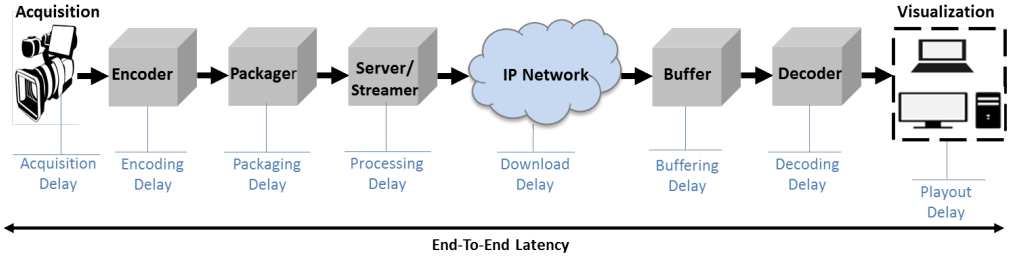


Figure 2.2 – Architecture of a typical video streaming chain.

of the different delays contributing to end-to-end latency experienced by a viewer in live streaming was explored in [20]. However, the authors of [20] do not investigate the origin sources of these delays, i.e. which used features or functions add a delay. We note that the acquisition delay is not further discussed as we did not consider it in this thesis.

2.3.2 Components and Features

In this section, we represent the relationships among the different components of a video streaming chain, i.e. how these pieces work together, by defining and examining their functions. Essential features of each component are analyzed and discussed from the end-to-end latency point of view.

2.3.2.1 Encoder

Video content can be either generated by users (i.e. viewers) producing so-called user-generated content which can go directly from a camera device to the encoder, or gathered from a variety of sources into a central production where it may incur many steps of processing such as editing, color correction, encoding, decoding, logo insertion, etc. before invoking the final encoder. In our work, we start evaluating the end-to-end latency either from the frame capture time or from the final encoder input, depending on the encoding methods used (live/on-the-fly or offline).
The captured video content may be encoded either on-the-fly or off-line. Off-line encoding is used for VoD service in order to maximize the video quality. It has the advantage that it does not require real-time encoding constraints. This enables more efficient encoding such as two-pass encoding. In the first pass of two-pass encoding, the encoder analyzes the video from the beginning to the end to determine the best possible way to fit the video within the bitrate limits. It determines for example the best possible positions for intra-frames to constitute Groups Of Pictures (GOPs). In the second pass, the collected data from the first pass is used to optimize the bitrate and to achieve the maximum encoding quality. On-the-fly encoding is used for live streaming and real-time interactive streaming services that require a real-time encoding, and potentially a very short encoding delay for real-time interactive streaming service. In our experiments, we use on-the-fly encoding as we target a low latency delivery in both services.

Video Coding Reminder

Since uncompressed video images consume a large amount of bandwidth, compression is usually used for storage, archival and transmission. The encoder takes as input a series

of raw video images, encodes it, and produces compressed video bitstream suitable for further processing by the packager. As Advanced Video Coding (H.264/AVC) [21] and High Efficiency Video Coding (H.265/HEVC) [22] are the most common video coding standards for OTT, we limited ourselves to these two video codecs in our work.
As H.264/AVC, an HEVC bitstream consists of a number of encoded frames called access units (AUs), each including coded data to which timing information such as Decoding/Presentation timestamps (DTS/PTS) can be attributed. Each AU is divided into NAL units (NALUs). AVC and HEVC encoders output each NALU once it is produced. In our work, we do not consider the generated NALUs until a complete frame is constructed.

Encoder Features

We present in this section the coding features and how their use for video streaming services impact the end-to-end latency.

- B frames

  B frames are bidirectional predicted frames, i.e. they are predicted from both previous and future frames, either I, P or B frames. By using bidirectional prediction, compression performance improves because the temporal correlation among several neighboring frames is better exploited [23]. Reference frames need to be encoded and transmitted before the B frame itself. This re-ordering of frames by the encoder introduces a delay. For example, the additional delay for a typical IBP-coding structure is one frame duration. It is two frames durations for IBBP-coding structure. The introduced delay depends on the GOP structure, i.e. depends on the position of the reference frames in the GOP. The greater the distance between the reference frames, the larger the delay. This coding type is most common in VoD and live streaming services but it is not suitable for real-time interactive streaming service where latency is of importance and the delay is to be kept at a minimum. In our live streaming experiments, we do not use B frames.

- GOP type

  There are two primary GOP types known as closed and open GOP. Both GOPs types use a Random Access Point (RAP) frame which is a location in a bitstream at which a decoder can begin successfully decoding frames following the RAP in display order without needing to decode any earlier frames in the bitstream. A RAP is provided by a frame which can be independently decoded of all other frames. RAP frames are used in video streaming to provide functionalities such as seeking and tune-in. It is interesting to note that the GOP type affects the switching in adaptive streaming systems as we explain below.

A) Closed GOP

Closed GOPs cannot contain any frames that reference a frame in the previous or next GOP. As shown in Figure 2.3, a closed GOP must start with a specific RAP frame called an Instantaneous Decoding Refresh (IDR) frame, followed by non-IDR frames that can be decoded in decoding order without inter prediction from any frame decoded before the IDR frame. The presence of an IDR frame causes a reset of the decoding process, i.e. mark all reference frames as "unused for reference" immediately after decoding the IDR frame, while an I frame that is not an IDR frame does not. We show in Figure 2.3 the display order as well as the decoding order of the frames in the GOP.

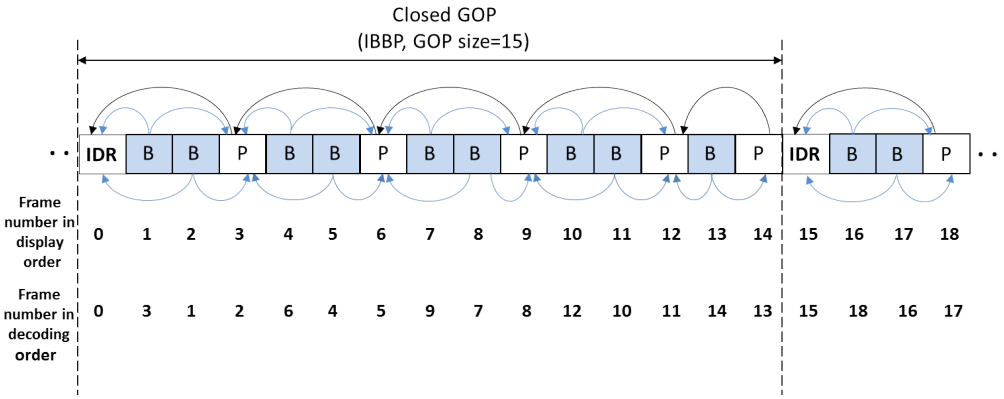


Figure 2.3 – Closed GOP.

B) Open GOP

A GOP is open when the reference frames from the previous GOP at the current GOP boundary can be exploited. As shown in Figure 2.4, open GOPs may begin in display order with one or more B frames that reference the last P frame of the previous GOP as well as the first I frame of its own GOP. The frames that precede the I frame in display order appear in the stream afterward in decoding order, e.g. the first two B frames of the current GOP in Figure 2.4. These frames are called "leading frames". They do not only refer to the I frame, but also to some earlier reference frames, e.g. the last P frame of the previous GOP in Figure 2.4, that is not available and known to a decoder which has just started decoding the stream. Consequently, these frames cannot be correctly decoded. They must therefore be discarded by a decoder that starts its decoding process at I frame.

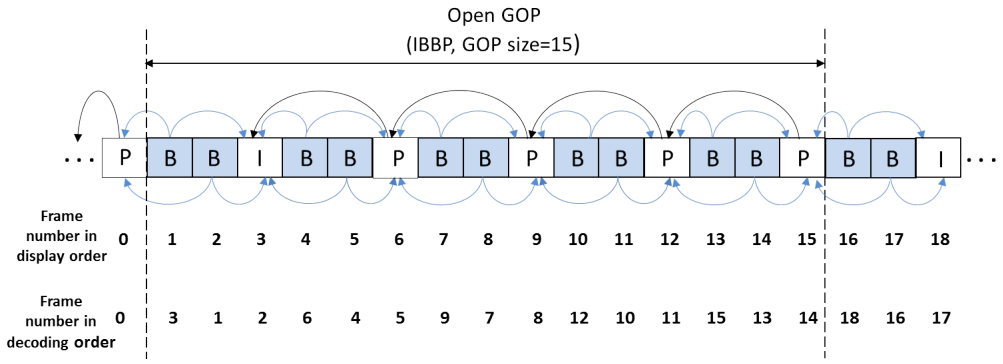


Figure 2.4 – Open GOP.

Open GOPs are more efficient than closed GOPs because they reuse data from previous GOPs. This can reduce the bitrate and provide an efficient compression. Additionally, they allow a better smoothing of the used bitrate, i.e. they have a smoother data distribution in the GOP. When using a closed GOP, the bitrate is not smooth because the first P frame in the GOP is transmitted right after the IDR frame as shown in Figure 2.3. Note that P frame requires an important bitrate but much less than I frame does. However, in an open GOP as depicted in Figure 2.4, two B frames that do not need a high bitrate are transmitted between the I and P frames, which can smooth the bitrate peaks of I and P frames.

Open GOPs are often used in broadcast TV systems due to their good compression performance. In case of channel change, the inability to decode some frames as explained above has no impact on viewer experience because it concerns "past frames", before the IDR and therefore before the first frame presented after tune-in. In OTT however, since the network bandwidth varies over the time, the client is likely to switch regularly between streams encoded with different bitrates. If open GOPs are used in OTT systems, the non-decodable frames would be discarded at each switching which may be noticeable in the playback and result in a degradation in the quality experienced by viewers. That is why the use of open GOPs in OTT is still limited. Currently, most of OTT providers use closed GOPs.

- GOP length

The GOP length represents a trade-off between the needs of random access versus bandwidth. It also enables the error recovery in lossy environments. By increasing the GoP length, there will be less I frames and more inter-frames which can optimize the bandwidth and ensure a high compression. On the other hand, streams with smaller GOP length contain frequent RAPs which can be desirable for fine grain random

access (seeking and tune-in). The seek time as well as the tune-in time will be lower at the expense of high bitrate or low quality.

- Multiple version encoding

  For adaptive video streaming systems, the encoder can encode the same video content into different versions (e.g. different bitrates and resolutions). The use of this feature enables dynamic adaptation of video quality to varying network bandwidth and client's device capabilities so as to provide uninterrupted video streaming service.
  Figure 2.5 illustrates an example of a bitstream switching between two streams encoded with two different bitrates but that are not RAP aligned. When a client decides to switch to a new rate (e.g. rate 2 in Figure 2.5), if it is not aware of the RAP positions in the new stream, it has to retrieve the new stream from the beginning, find the most recent I frame, decode it, and then decode all necessary inter-frames up to the desired frame. This results in a double download of "δd" frames and in a double decoding of "δg" frames, which can increase the downloading and decoding delays and consequently the end-to-end latency. On the other hand, the server knows in advance the RAP positions in the new stream. If it drives the adaptation process, it will switch to rate 2 exactly at the RAP frame as shown in Figure 2.5. Hence, whether the rate adaptation is server-driven or the encoded streams are RAP aligned the double downloading and decoding is avoided.

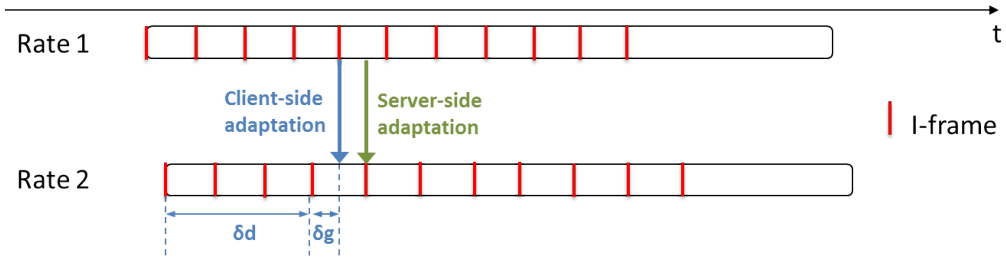


Figure 2.5 – Bitstream switching between two versions that are not RAP frame-aligned.

2.3.2.2 Packager

A packager receives the encoded frames and packages them in a specific format suitable for storage or transmission for a given delivery protocol and network. The basic principle of packaging consists of creating packets or files by adding a certain amount of information in a header of a packet or a file to the media data. In our work, we have identified a packager as an important contributor to end-to-end latency in live video streaming. Hence, reducing the packaging delay was one of the challenges of this thesis.
A packager provides a set of functions that we define below. Following that, we review the common packaging formats.

Packager Functions

- Aggregation

  The aggregation process is introduced when the encoded media data to be delivered over the network is smaller than the Maximum Transmission Unit (MTU) size of the underlying network, i.e. the largest packet size that can be transmitted over the network without being fragmented on the network layer. The packager can aggregate either complete encoded frames or parts of them in a specific format, depending on the delivery protocol and network, to constitute packets or files. The aggregation at the packager level avoids undesirable packetization overhead because only one header is used for several frames or parts of them. The process of aggregating many encoded frames can increase the packaging delay if the frames are transmitted sequentially to the packager (usual situation), so the packet/file can only be delivered once the last frame is produced.

- Fragmentation

  Large encoded media data that exceed the network MTU must be fragmented into several transport packets before transmission. This process is typically needed for video data as it is considerably larger than audio. It creates an overhead because many packets headers are used per frame. However, the packaging delay when fragmentation is used is low because the packet can be sent before the entire frame is produced.
  In modern bitstream formats, the encoder typically defines some marker points in the encoded frames called synchronization parsing points that the packager relies on to split the frame in appropriate places to help the client use the data in the event of some parts being lost. Making the packager aware of the boundaries to perform the fragmentation refers to the media-aware fragmentation mechanism.

- Multiplexing

  The multiplexing functionality consists in mixing different media types into a single stream and transmiting it over the network. It mainly has two functions:

  – Synchronization: It allows the packager to synchronize the different media streams (e.g. video and audio) and avoid possible synchronization issues at the client due to separate stream delivery.

  – Media interleaving: Groups of encoded frames of different streams are stored alternatively in the file/packet, e.g. N milliseconds of video frames, followed by N milliseconds of audio frames, followed by N milliseconds of video frames, etc. The groups are stored/transmitted consecutive in the file/packet. Typically, interleaved frames are grouped within an interleaving window. When the inter-

leaving window is small the overhead increases and the frames can be buffered and reconstructed back to their original order quickly. The obvious disadvantage of a large interleaving window is that it increases latency. As video frames take longer time to encode compared to audio frames, the file/packet can only be delivered once the last frame of the video group is produced which can increase latency.

In this thesis, we only dealt with frame aggregation.

Common Packager Formats

There are several types of packaging formats, each targeting different areas of applications. Standard ones are Real-time Transport Protocol (RTP), MPEG-2 Transport Stream (MPEG-2 TS), and ISO Base Media File Format (ISOBMFF). Our system focuses on the latter two formats. In the following, we review how the previous packager's functions are used in these formats. We also evaluate the packaging delay introduced by each format.

1. RTP

   RTP is designed for real-time packet streaming. When using H.264/AVC and H.265/HEVC, NALUs are the data source for RTP packets. In the simple packetization model, one NALU is put in one RTP packet. This results in a negligible packaging delay.
   In some cases, NALUs may be bigger than the RTP packet size. For example, when the content is pre-encoded without knowing the MTU size of the underlying network. Hence, large NALUs are broken into several parts called Fragment Units (FUs). Each FU is then put into the payload of an RTP packet, but this keeps the packaging delay low.
   In RTP systems, some source data are typically very small- a few bytes at most (e.g. the parameter set NALUs). To respect the RTP packet size, it is better for the packager to aggregate them with other NALUs into a single packet. The aggregation process can reduce the introduced overhead. The NALUs may be produced after a non-negligeable time. However in such cases, these NALUs would typically not be aggregated on a single packet, but sent as dedicated packets.

2. MPEG-2 TS

   MPEG-2 TS is typically used in digital TV broadcast networks and in some video streaming technologies such as Apple's HTTP Live Streaming. It is one of the multiplex stream formats specified by the MPEG-2 systems standard. It consists of relatively short fixed-length packets of 188 bytes, each with a 4 byte header. The packetization for this model is illustrated in Figure 2.6. It is as follow: an AU, or

one or more NALUs are packaged at first into a long variable-length packet called Packetized Elementary Stream (PES) packet. A PES packet is then partitioned up into 184 bytes parts to fit in the TS packet payload. A TS header is added to constitute a TS packet. Thus, the packaging delay is negligible.

As with the RTP packaging format, a TS packager may aggregate many AUs in a single PES packet, particularly for audio, which increases potentially the delay but reduces padding. It may also fragment an AU across different PES packets, with no impact on the delay.

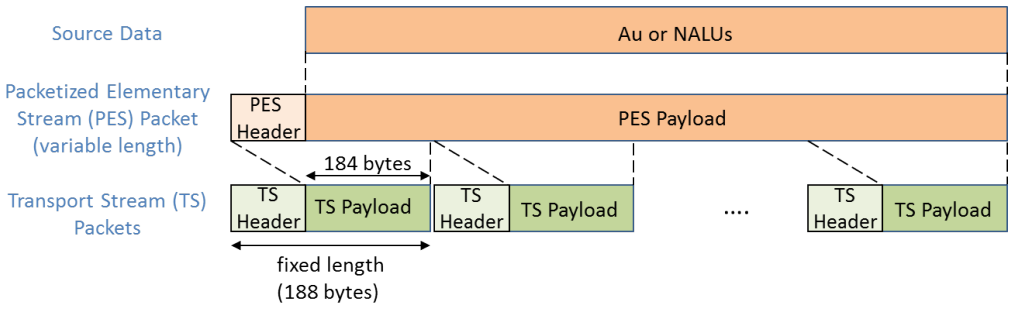


Figure 2.6 – Generation of MPEG-2 TS packets.

3. ISOBMFF

ISO Base Media File Format (ISOBMFF) is used for storage, file exchange, editing and streaming purposes over IP. ISOBMFF organizes the media data and its metadata in data structures called boxes. A box typically consists of four bytes for the box type, four bytes for the box length, and the remaining bytes for the payload. Files are structured as a series of boxes that can be organized sequentially and hierarchically. The metadata for each media type present in the file is stored in a track box (”trak”), which are subsequently grouped with the others in a movie box (”moov”). The media data of each track could be either enclosed in the same file in a media data box (”mdat”) or in a separate file. The media data of each track consists of media samples which are the access units of a media type, i.e. the output of the encoder.

The internal file organization differs significantly when using ISOBMFF for storage only or when using it for streaming purposes. The above described file organization method is not suitable for incremental generation/consumption of the media content at the server/client for two reasons. First, the amount of metadata of the ”moov” box is constantly growing as data is encoded. The ISOBMFF file cannot be written until the movie box is completely constructed. Second, if the video is long (e.g. a movie) the tables describing the samples in the ”moov” box are large values and could take a long time to be downloaded.

For that, progressive generation of the metadata and the media data was introduced. The file still contains the ”moov” box that holds decoder initialization information and metadata that is valid for the whole file. The rest of the file is a sequence of movie fragments that contains an alternating series of movie fragments boxes (”moof”) and media data boxes (”mdat”). The movie fragment box contains the metadata for a single fragment while the media data box stores the associated media samples (e.g. audio or video encoded frames). A movie fragment (i.e. ”moof” and ”mdat” boxes) is the smallest entity that can be independently parsable. For advanced streaming purposes such as HTTP adaptive streaming, an additional layer for the organization method, called segmentation, was proposed by 3GPP [24]. It consists in dividing the ISOBMFF file into self-contained independently decodable files called segments. Each segment consists of either a movie box, with its associated media data and other associated boxes or one or more movie fragments boxes, with their associated media data, and other associated boxes.

So, in ISOBMFF, the packager aggregates either all access units into a single ISOBMFF file, or it can select an integral number of complete closed GOPs to constitute one segment in HTTP adaptive streaming, that we will explore in 2.4. This results in a packaging delay equal to either the file duration or the media segment duration. Long segments ensure a low overhead but they introduce a long packaging delay. This configuration is not suitable for live streaming and real-time interactive services. One of our contributions consists in using the movie fragments as new delivery entities to clients instead of media segments, thereby reducing the packaging delay to the duration of a fragment.

2.3.2.3 Networks and Servers

Servers are responsible for receiving the packaged stream to form IP packets. RTP packets are typically delivered using User Datagram Protocol (UDP) over IP networks. Two methods can be used for the carriage of MPEG-2 TS packets over IP: the direct UDP method where the TS packets are carried in the UDP payload, or the RTP method where RTP payload is used to transport the TS packets. ISOBMFF files are generally delivered using the HTTP protocol, transported over Transmission Control Protocol (TCP), and delivered over IP.

The server may become overloaded when the demand for video contents increases. This phenomenon causes degradations in network performance, resulting in a decrease in useful bandwidth, an increase in queuing delay, packet loss rate and jitter. Server scalability (i.e. the capacity to accept a very large number of clients) is achieved by two different solutions that we explore below.

1. RTP-based services

   Multicast or one-to-many video streaming over IP is the first solution in RTP-based services. The server streams a single packet of data at the same time to many clients. This can reduce the network resource usage and increase the bandwidth efficiently. However, muticast provides limited flexibility as the clients cannot request different video contents at different instants. It is useful only when multiple clients would watch the same video at the same live time. Additionally, the system deployment is not easy because routers and intermediaries are not configured to route multicast at all, or only configured and allowed in local networks. Moreover, multicast requires UDP which may also be filtered.

2. HTTP-based services

   In HTTP-based services, large distributed systems called Content Delivery Networks (CDNs) were developed to overcome the scalability problem. They consist in hundreds or thousands of servers placed in the path between the source and the clients. The server that is located close to the source is the origin server while the ones which are close to the clients are the edge servers. The primary technique that a CDN uses to speed the delivery of contents to end users is caching, which entails storing replicas of content in multiple edge servers, so that user requests can be served by a nearby edge server rather than by a far-off origin server. This fact implies that CDN can reduce the request/response time, the network bandwidth consumption, the probability of packet loss and the total network resource usage. The server thus has sufficient bandwidth to handle additional network requests.

Network Constraints

In contrast to the previous components that present some technical features and functions, the network provides a number of constraints and challenging issues that are detailed below.

- Time-varying network bandwidth

  The bandwidth available between two points in the open Internet network is generally time-varying and requires constant estimation. The server is limited to send data to clients at the available bandwidth. If the server transmits faster than the available bandwidth then congestion occurs and packets are lost, causing degradation in video quality. If the server transmits slower than the available bandwidth, the client does not use effectively the bandwidth and receives sub-optimal video quality. The main challenge in particular in HTTP adaptive streaming is to estimate the available

bandwidth and then adapt the encoded video bitrate to be transmitted to the available bandwidth.

2.3.2.4 Client

A client requests data, receives the network packets from the server, unpackages them to retrieve the encoded frames, decodes and displays the frames on the viewer's screen. Buffering in the client side may be needed to deal with transport jitter and provide a smooth playback. A client is also required to synchronize the different frames of different media arriving at different times. Additionally, it may be in charge of applying the viewer preferences if requested (e.g. language and subtitle).
The playout delay is the time between when the frame is fully decoded and ready to be played out and when it is actually displayed.

In the client, a buffer is used to store a few seconds of content before their decode and display to minimize sporadic failures or delay fluctuations in the network transmission. The size of the buffer must be tuned carefully according to the current network conditions and the application's needs. If it is too big, it adds unnecessary delay, if it is too small, the playback may freeze. There are two kinds of buffers at the client side, de-jittering and decoding. A de-jittering buffer is designed to remove the delay variation (i.e. jitter) in packet arrival times. For transmission of video packets over IP networks, it has been shown that the typical value of delay jitter can be up to 2 s [25]. A decoding buffer is used to store data in case of the decoder cannot consume it as soon as it arrives. Usually, they are the same buffer.

2.3.3 Summary

The analysis that we have conducted in this section is aimed at identifying and explaining the main contributing components to the end-to-end latency. Content encoding, packetization, downloading, buffering at the client side, decoding and playout add delays which may increase the end-to-end latency.

Three factors may impact the introduced delay by encoding and decoding processes. The use of B frames requires a re-ordering of frames in the encoder/decoder which can introduce an additional delay. In case of client-driven rate adaptation, the non-temporal RAP alignment increases the downloading delay as well as the decoding delay. The GOP type, i.e. open or closed, does not impact the latency but it affects the switching in adaptive streaming systems. The RAP frequency impacts the bitrate and operations such as seeking and tune-in, but not the delay.
For all packaging formats, the number of encoded frames aggregated in a single packet or

file presents a trade-off between overhead and latency. The ability to package less than a frame determines if latency can be smaller than a frame duration.
The download delay depends on the content size, available network bandwidth and the roundtrip time between the client and the server. The buffering delay depends on the buffer size which is usually fixed by the application type. Finally, the playout delay is bounded between 0 ms and the frame duration.
The main challenge for content and service providers is to design a video streaming chain to reliably deliver high-quality video experience to viewers over IP network when dealing with the above components' features and constraints.

For viewers that require a low end-to-end latency, the streaming chain should be configured as follows. When using B frames, the distance between the reference frames should not be too long. If latency is to be reduced to its minimum, B frames should not be used. Regarding VoD, we can use many B frames and no real-time encoding constraints. From the adaptivity point of view, the GOPs should not be very long. A long GOP could be too slow to adapt to sudden bandwidth drops, and could increase the rebuffering events which may impact the viewer QoE as shown in Section 2.2. Live streaming and real-time interactive streaming services require the use of short buffers.

2.4 HTTP Adaptive Streaming

Diverse IP-based video streaming systems have been presented and used in the academic studies and industrial implementations [26] [27]. HTTP Adaptive Streaming (HAS) has emerged as the technology of choice for the delivery of audiovisual content over the Internet. In this section, we review the basic concept of HAS with its various features.

In an HAS system, it is necessary for the server to maintain multiple versions of the same video content, encoded in different bitrates and quality levels. Each encoded version goes through a segmentation process where it is divided into short-duration segments, typically a few seconds. The segmentation process is one the main challenges of this thesis related to low latency live video streaming. We address this problem in Chapter 4.
Once the client chooses the appropriate video version to download, it sends an HTTP request to fetch a particular segment from an HTTP web server and then renders the media while the next segment content is being downloaded. The client drives the video quality adaptation by requesting different segments at different encoding versions. If during the encoding process, all versions of the same content are perfectly RAP aligned and no open GOPs are used, switching between them can be completely seamless, even with one decoder.
A rate adaptation algorithm at the client-side will decide which quality version is requested from the server. Several algorithms have been proposed recently, which can be classified

into three main categories with respect to the required input information, ranging from network characteristics to application-layer parameters such as the playback buffer. Firstly, throughput-based algorithms, such as PANDA [28] or Festive [29], rely their decision on the observed TCP throughout. Secondly, time-based algorithms such as ABMA [30] rely on the same principle of probing, but this time to estimate the download time of each segment. Lastly, buffer-based algorithms, such as BBA [31] and BOLA [32], observe and react to the level of the client's playback buffer. [33] provides a comprehensive comparative study of the main existing HAS algorithms by evaluating their performance per class under controlled experimental conditions.

HAS has an issue which relates to the undesirable behaviors of HAS players that affect viewer QoE [34], [29], [35]. It is common that two or more players share a network bottleneck and compete for available bandwidth. This competition can lead to three performance problems: instability (i.e. the video quality often changes so the client switching is too frequent), unfairness (i.e. allocate throughput unfairly to multiple competing players sharing a bottleneck link) and inefficiency (i.e. bandwidth under-utilization). Most of these problems happen because the clients repeatedly go between downloading and pause phases (called ON and OFF periods), which confuses other competing clients about how to estimate their fair share bandwidth correctly during the OFF periods [34]. To solve these problems, some researchers try removing the ON/OFF periods with the help of the server [36], [37], while others develop more accurate bandwidth estimation algorithms [29], [38].

Numerous streaming service providers have adopted HAS technology and have proposed new solutions that are widely deployed such as Microsoft's Smooth Streaming (MSS), Apple's HTTP Live Streaming (HLS), Adobe's HTTP Dynamic Streaming (HDS). In this trend, a new standard called MPEG Dynamic Adaptive Streaming over HTTP (DASH) was developed and used worldwide. We present in the following MPEG-DASH as our work focuses on this standard. We then describe and compare the main features of HAS systems, especially DASH-based ones.

2.4.1 MPEG-DASH

Adaptive streaming solutions developped by different vendors (like Microsoft, Apple, and Adobe) use different file formats. In other words, no interoperability exists between devices and servers of various vendors. To receive a content from each server, a device must support its corresponding proprietary client. Therefore, MPEG-DASH, a new common standard in adaptive streaming, has been developed by MPEG and 3GPP to enable the interoperability in the industry [1]. In this section, we mainly review the DASH principles regarding how the available media on HTTP servers is presented and described, and the DASH client behavior.

Media Presentation Description

In MPEG-DASH, a description of the available media content at the server and its various alternatives is provided by a Media Presentation Description (MPD) file, which is an XML-based document. Figure 2.7 demonstrates the MPD hierarchical data model. Each media content may be composed of one or more media components (e.g. video, audio, subtitle), each of which might have different characteristics. A long media content could be divided into one or more temporal chapters called periods. Each period has a starting time and duration and consists of one or multiple adaptation sets. An adaptation set includes information about one or more media components and its various encoded versions. Each adaptation set consists of multiple representations. A representation is an encoded version of the content components. Each representation varies from other representations by bitrate, resolution, number of channels, or other characteristics. The media stream in a representation is chopped into a sequence of portions that can be independently decoded, called media segments. Media segments can be in separate files (common for live streaming service), or they can be byte ranges within a single file (common for VoD service). Each media segment has a unique URL that indicate an addressable location on a web server that can be requested by a client using HTTP GET, as complete ressource or with byte ranges. It exists several addressing schemes that define how media segment URLs are identified in the MPD file:

- Segment List is a complete list of segment URLs provided for all available segments.
- Segment Template (Segment Time-Based): A URL template is provided from which clients build a segment list where the segment URLs include segment start times.
- Segment Template (Segment Number-Based): A URL template is provided from

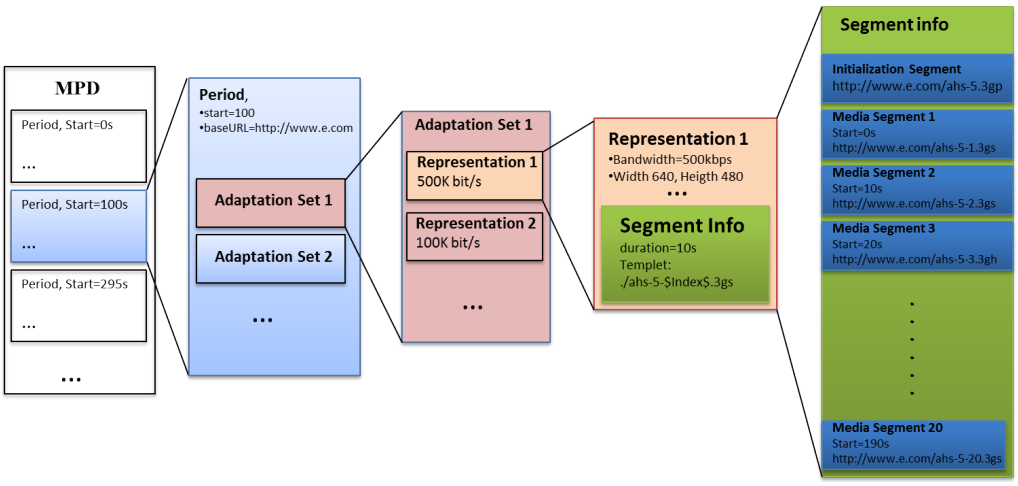


Figure 2.7 – MPD hierarchical data model [1].

which clients build a segment list where the segment URLs include segment numbers (like index numbers).

- Segment Base (BaseURL): A non-segmented scheme where a single segment is identified with a single URL (BaseURL), with the intent that the content will be retrieved through byte-range requests given within the content.

DASH Client Behavior

DASH relies on a client-driven streaming approach. In other words, it is the client's responsibility to make its choices on which segments to download. A simplified DASH client behavior can be divided into three phases as shown in Figure 2.8. We note that the other HAS solutions follow similar client behaviors.

1. Bootstrap Phase

   In a first phase which we call bootstrap, the client retrieves all necessary information to start the streaming session. It begins by fetching the MPD file from the web server. Once the MPD is received, the client parses it and learns about the program timing, media-content availability, media types, resolutions, and the existence of

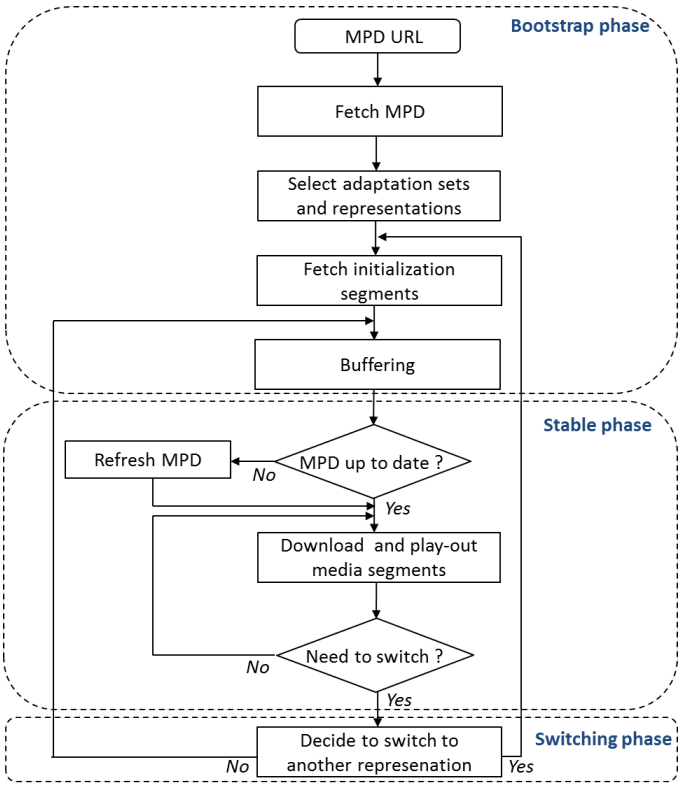


Figure 2.8 – Simplified view of the phases of DASH client.

various encoded alternatives of multimedia components, media-component locations on the network, and other content characteristics. Using this information, the client selects a set of adaptation sets compatible with its capabilities in terms of codecs, content media types (audio, video, subtitle), languages. Within each adaptation set, it chooses a representation that best satisfies its needs regarding bitrate, resolution. It then fetches the initialization segment (IS) that contains decoder configuration (e.g ISOBMFF "moov" box) of each selected representation using HTTP GET requests. After some buffering, the client enters a stable phase.

2. Stable Phase

   In the stable phase, the client continuously downloads media segments and potentially updates the MPD. The client plays media segments and monitors the network bandwidth fluctuations.

3. Switching Phase

   In some cases, the client may need to switch dynamically between different representations (with lower or higher bitrates) to adapt to fluctuating network conditions and to maintain an adequate buffer. It enters then a switching phase where it may have to initialize newly selected representations. Following that, it can fetch and play segments of the new quality, entering again the stable phase.
   There are cases where the bootstrap and switching phases interact: when a DASH client switches between 2 representations not sharing the same IS, the DASH client has to download the IS corresponding to the new representation, prior to the new media segment.

2.4.2 HTTP Adaptive Streaming Features

We describe the main features of HAS systems that adaptive streaming solutions have picked. All solutions are implemented differently but they share a similar operational model.

- File organization unit: HAS relies on two different ways to organize the different encoding bitrate versions of the same video content at the server side: one-file-per-segment-and-bitrate or one-single-file-per-bitrate. If encoded bitrate version is partitioned into many segments as described above and each segment is self-contained file, this refers to the one-file-per-segment-and-bitrate approach. Otherwise, each encoding bitrate version is stored in a single file, which refers to one-single-file-per-bitrate approach. This approach is efficient to decrease the web servers and CDNs load, and management of millions of tiny files. It is used typically for VoD service. All HAS solutions may use both approaches.

- Segment duration (i.e. how much media time each segment should carry) is generally fixed over the streaming period. In practice, default segment duration is 2 s in MSS [39], 10 s in HLS [40] and it ranges from 2 to 10 s in DASH. In [41], the authors observe that the choice of segment duration has a great impact on the accuracy of bandwidth estimation and video quality adaptation. A short duration could lead to suboptimal bandwidth estimation and incur more overhead from frequent HTTP requests/responses. However, a long duration allows more accurate bandwidth estimation but could be too slow to adapt to sudden bandwidth drop, and could increase the rebuffering events. Note that timely response to fast-changing bandwidth is the key to minimizing the frequency of rebuffering events that significantly undermine viewer QoE.

- Manifest file and update: In HAS system, a client fetches first from a web server a description of a streaming session in a manifest file. This tells a client which content media types are accessible, which codecs were used to encode the content, which bitrate, resolution, and language are available, and a list of the available segments with either their start times or durations.
  This manifest in MSS is called a client manifest with a specific extension (*.ismc). In addition to this, the IIS Smooth server stores and uses another type of manifest (*.ism) for the mapping between requested bitrates and the MP4 files stored on disk. In Live MSS, a client doesn't need to repeatedly download a manifest. It can continue fetching segments without having to request a new update manifest because the next URL segment is inband. The current segment holds the timestamp of the next segment or two in a box inside the file [42].
  HLS defines two types of manifest of the same extension (*.m3u8): Normal and variant playlist files. The normal playlist is defined per video encoding version and lists URLs of segments that should be played in the chronological order. Variant playlist file lists a collection of normal playlist files with their metadata (e.g. bitrate, resolution, codec, an ID). The variant playlists support delivery of multiple streams of the same content with varying quality levels for different bandwidths or devices.
  In case of live streaming, whenever a new media segment is ready, the normal playlist file is updated to include the newest segment and to remove the oldest one. HLS recommends that the playlist contains at least the 3 latest segments so that the client can pause, resume, and rewind for at least the duration of 3 segments (30 s by default) [39]. The client loads a new version of the normal playlist file periodically to get the URLs of the new media segments. When the client decides to switch between the alternates streams, it has to download the appropriate normal playlist that satisfies its needs regarding the quality level.
  DASH defines a client MPD manifest. In live streaming, the client has to update the MPD file if the URLs of the available segments are listed explicitly or an MPD

update period is given.

- Types of data: Three types of data are used in HAS systems including media, initialization, and index data.
  First, media data is the actual media files or segments that a client plays. All solutions define a specific media data type for the video playback. Second, initialization data consists of metadata required for decoder initialization to start the video playback. It is inband in MPEG-2 TS format (i.e. stored in the same file as media data) or outband in ISOBMFF (i.e. stored in a separate segment file). All solutions use the initialization data. Finally, index data provides mainly a mapping between the time and the location of media in a segment or in a file. MSS uses an index data and it is stored in a movie fragment random access "mfra" [39]. DASH may use an index segment and it consists in one or more segment index box ("sidx") which is placed before the movie fragment box ("moof").

- HTTP requests/responses transactions: As we have already mentioned before, a client is required to download a manifest file from a web server either in MSS or DASH using one HTTP request. HLS needs two round trips to fetch and receive the two types of manifest files as mentioned above, which increases the startup delay. When using initialization segments for ISOBMFF format, a client has to fetch the initialization segment before starting issuing media segments using additional HTTP requests/responses.

2.5 Conclusion

This chapter presented the basic aspects of QoE of video streaming services using IP networks. We have presented a qualitative description of viewers' behaviors and expectations in terms of QoE for different services types. We focused in particular on the quality of the provided service targeting the latency service performance metric. In the litterature, many QoE metrics were proposed to accurately assess viewer QoE for video streaming services. This chapter can be extended to include other evaluation metrics such as a switch frequency. This metric is used to report the number of switch events that look place during the video streaming session.

This chapter explains as well all the aspects that are related to the latency in a video streaming chain. It presented the architecture of a video streaming chain with its main components. We analyzed the features of each component of the chain from the latency point of view. We also presented how a video streaming chain should be configured when low latency is needed. The choice of components settings may differ if latency is not the first viewer requirement. Additionally, the analysis of the video streaming chain can be

extended regarding the multicast and CDN distribution networks.

Finally, this chapter presented the specific aspects that are related to the HTTP adaptive video streaming that are related to latency (e.g. number of manifests to download, manifest update, download of initialization segments, segment duration, etc). Our contributions target the optimization of the bootstrap as well as the steady phases in terms of latency.

Chapter 3

Improving the Starting of Live DASH Streaming Sessions

Contents

3.1 Introduction

Today's viewers are highly impatient and less tolerant of slow starts in live streams compared to VoD streaming. Some studies [4] indicate that if the startup delay exceeds 2 seconds, the number of people that abandon viewing dramatically increases. To retain viewers, it is therefore important to provide them with an "instantaneous" video playback.

One of the solutions to address the slow start of a live streaming session issue is to reduce the startup delay that we have defined in Chapter 2 as the delay between the time when a viewer issuing "Play" to the start of video display.

The startup delay components in DASH live streaming are shown in a simplified manner in Figure 3.1.

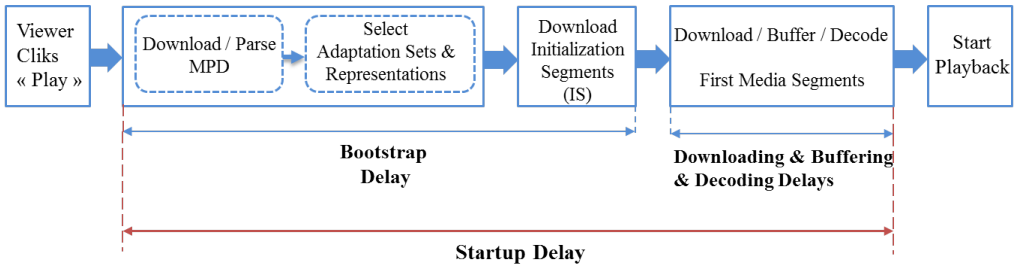


Figure 3.1 – Startup delay components in DASH live streaming.

The bootstrap delay corresponds to the duration of the bootstrap phase that is presented in Chapter 2, in which a client retrieves all necessary information (i.e. MPD and IS files) to start the DASH streaming session. A client starts by issuing an HTTP request to retrieve the MPD from a web server. Once the MPD is received, the client parses it and selects a set of adaptation sets compatible with its capabilities in terms of codecs, content media types (audio, video, subtitle), and languages. Within each adaptation set, it chooses a representation that best satisfies its needs regarding bitrate and resolution. A client issues then additional HTTP requests to fetch the IS of each chosen representation for the initial playback. This separate download of MPD and IS increases the bootstrap delay which increases the startup delay; in this chapter, we focus on the bootstrap delay.

Following that, a client proceeds to download, buffer and decode the first media segments to be ready for the initial playback as depicted in Figure 3.1. For that, a client can use different strategies depending on its configuration and implementation. Figure 3.2 shows a regular strategy which is the most deployed in the existing HTTP live streaming technologies and widely used in the current web browsers. The principle of this strategy consists in buffering one media segment or more before invoking the playback. This means that a client cannot start playing a media segment before some media segments are completely downloaded and buffered. In Figure 3.2, we consider the case where only one entire segment is buffered before starting playback.

According to Figure 3.2, media segments are generated and then stored on a web server. A client sends an HTTP request to download the first media segment (S_i). A media segment is defined as a consecutive series of frames, which are a sequence of bytes. A client receives the requested media segment byte by byte from a web server. It starts receiving the first byte of the first frame (FB_{f1}). After a certain amount of received bytes, it receives the last byte of the first frame (LB_{f1}). Thus, the first complete frame (f_1) is constituted. $D_{dow\ f1}$ is the time needed to download this f_1. A client continues receiving the bytes and forming the remaining frames until it receives the last frame (f_m) of a media segment. The time that

is required to download all frames constituting a media segment (S_i) is the downloading delay (D_{Si}). Once a media segment is entirely downloaded, it is sent immediately to the decoding buffer. The buffering delay is defined as the delay between the time when the first frame enters into the buffer and when it comes out of the buffer for decoding. Filling the decoding buffer is instantaneous and the for the first frame decoding can start immediately. Therefore, the buffering delay is considered to be zero on startup. The decoding of the first frame takes a certain time ($D_{decod\ f1}$) depending on the frame type (i.e. I, P, or B frames). In Figure 3.2, we consider only I and P frames. If B frames were involved, we would have an additional delay in the decoding due to frames reordering. Finally, the frame f_1 goes out the decoder to be displayed for $D_{play\ f1}$ time. The decoder usually starts decoding the second frame (f_2) while the first one is displaying. The frames decoding/displaying process continues until all frames of a media segment are completely displayed.
The client continues to request, download, buffer, decode and playout the following media segments in the same manner as the first one.

The downloading delay could be negligible if the beginning of the video that the viewer is likely to watch is prefetched by the video player. The main principle of prefetching is to retrieve content from the server before it is requested by a client and store it in a location

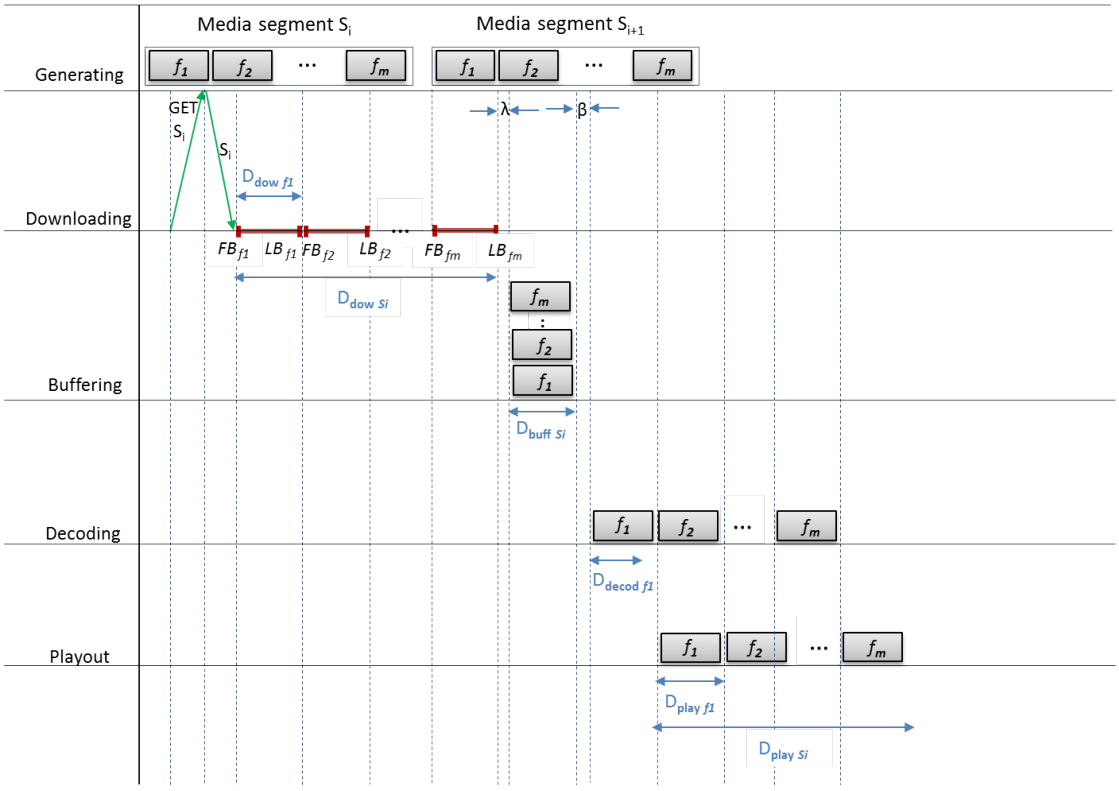


Figure 3.2 – Typical strategy to download, buffer and display media segments.

that can be accessed by a client conveniently and fast in periods of low link usage [43]. Prefetching is interesting and efficient to reduce the startup delay but it cannot be used for live streaming as live content is provided on-the-fly. Therefore, it is out-of-scope of our work because we target a low latency live DASH streaming case.

Initialization segments are not only downloaded in the bootstrap phase for the initial playback, but may be also retrieved in the switching phase of a streaming session. In particular, when a DASH client switches between 2 representations not sharing the same IS, the DASH client has to download the IS corresponding to the new representation, prior to the new media segment. If the available network bandwidth is low or if the time until the playback of the new segment is too short, this might be problematic. To anticipate such problem, a DASH client may request more than one IS in the bootstrap phase. For instance, it may fetch all IS for all video qualities to be ready to switch. This might increase the duration of the bootstrap phase but can improve the switching.

The main contribution of this chapter consists in reducing the bootstrap delay which can reduce the startup delay in DASH live streaming. This chapter proposes and evaluates several methods to reduce the bootstrap delay. All methods are based on the idea that the bootstrap phase should not require multiple round-trips between the client and the server. The methods exploit the counter-intuitive fact that in some situations downloading the MPD and all IS in one download can be achieved faster than downloading the MPD and then the only needed IS although smaller in size. The proposed methods have been designed to have no negative impact on the existing caching and delivery infrastructure.

The remainder of this chapter is organized as follows. Section 3.2 reviews typical DASH client strategies for bootstrap. Section 3.3 presents in detail our proposal. Finally, section 3.4 describes the test-bed of our experiments, details the experimentations and the obtained results before section 3.5 which concludes the chapter.

3.2 DASH Client Bootstrap Strategies

For the startup of a streaming session, during the bootstrap phase, the DASH client is required to download MPD and IS files [1]. For that, the client can use several strategies depending on the version of the HTTP protocol and on the number of desired TCP connections. Because MPD and IS files are small size resources (as will be shown in Section 3.4) and are delivered after the establishment of a TCP connection, it is essential to begin this section with an overview of the TCP startup mechanisms, in particular the TCP three-way handshake and TCP slow start phase. We then present the system parameters that we

[1] IS is necessary only when media segments are based on ISO/IEC 14496-12 (ISOBMFF), that we consider in our work, as it is used by most existing DASH deployments.

use throughout the chapter. Finally, we describe our first contribution which consists of an analytical evaluation of the different DASH client bootstrap strategies in terms of number of TCP connections, number of HTTP requests/responses, and the associated bootstrap delay.

3.2.1 TCP Startup Mechanisms

Every TCP connection must go through the TCP three-way handshake and then the TCP slow start phase. In the following, we illustrate how these mechanisms operate for a simple HTTP transfer.

TCP Three-Way Handshake

The TCP three-way handshake enables a server and a client to negotiate the parameters of the network connection (e.g. Initial Sequence Number (ISN) and Maximum Segment Size (MSS)) before beginning the data transfer. ISN is the starting segment sequence number that both sides must generate randomly for security issues. It is used for the numbering of the transmitted bytes in a TCP segment. MSS defines the largest amount of data, specified in bytes, that a client or a server can receive in a single TCP segment.
This mechanism is known as three-way handshake because three TCP segments (SYN, SYN ACK and ACK) are exchanged between the client and the server as depicted in Figure 3.3:

- SYN: The client begins by picking an ISN that will be indicated in the Sequence Number field of a TCP segment header. It also indicates its MSS value in the options field. It then sends a TCP segment with an activated SYN (Synchronization) flag. This SYN segment will inform the server what sequence number the client will start its TCP segments with. It may also include additional TCP flags and options.

- SYN ACK: The server acknowledges the client's SYN segment by adding the next sequence number that it is expecting to receive in an Acknowledgment Number field of the TCP segment header, which consists of the client's ISN incremented by one (ISN_{client} +1). It then picks its own random ISN, appends its own set of flags and options, activates the ACK (Acknowledgment) and SYN flags, and finally transmits the ACK SYN response.

- ACK: Finally, the client acknowledges the server's SYN ACK segment by incrementing the server's ISN by one and adding this value in an Acknowledgment Number field of the TCP segment header. The received acknowledgment number in the previous SYN ACK segment indicates which value of sequence number the server is expecting to receive from a client in the ACK segment. So, the client appends this value in the

Sequence Number field of the TCP segment header. The client can now complete the handshake by dispatching the last ACK segment. The connection is therefore established and the data transfer can start between the client and the server.

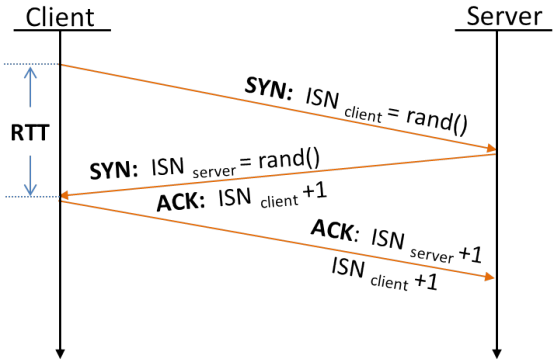


Figure 3.3 – TCP three-way handshake.

As a consequence, each new TCP connection will have a full roundtrip of latency before any data can be transferred which can penalize the startup delay. [44] has identified the TCP three-way handshake as an expensive component to create for each new TCP connection. The authors proposed a new mechanism called TCP Fast Open (TFO) to minimize the roundtrip penalty of the three-way handshake imposed on new TCP connections, by enabling a safe data transfer during TCP's initial handshake. This means that data will be transferred within a TCP SYN and SYN ACK segments. The analysis and testbed results have shown that TFO can improve single HTTP request latency by over 10% and the whole page load time from 4% to 40%. TFO eliminates one full RTT of latency but it works only in certain cases: there are limits on the maximum data size to be transferred during the handshake and only certain types of HTTP requests can be sent. A detailed discussion on the capabilities and limitations of TFO is provided in [45]. This proposal requires modifications in TCP stack and in existing web servers that we did not use.

TCP Slow Start

Once a connection is established, a client and a server cannot use immediately the full capacity of the link for the data transfer. TCP must enter the slow start phase where a client and a server try to quickly converge on the available bandwidth on the network path between them. The slow start algorithm is based on a congestion window (cwnd) which identifies how many TCP data segments a server may transmit without receiving an acknowledgment (ACK) from the client. Slow start mechanism starts with a small initial congestion window (init_cwnd) and increases it by the number of data segments acknowledged at each roundtrip as shown in Figure 3.4. TCP leaves the slow start phase either if packet loss occurs in the network or if cwnd reaches the size of the available client

buffer space.

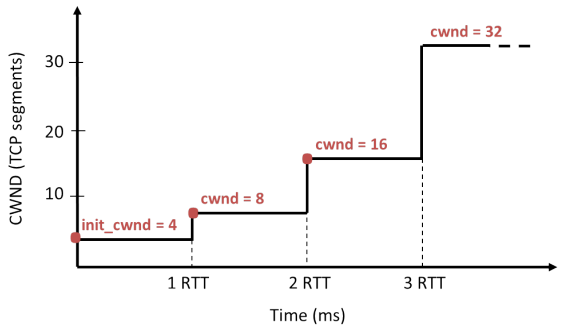


Figure 3.4 – Congestion window size growth [2].

The init_cwnd is at most 4 TCP segments, but more typically is 3 TCP segments (approximately 4KB) [46] in an Ethernet network. However, recently [47] involves increasing the init_cwnd to 10 TCP segments (about 15 KB) in order to minimize the latency caused by the slow start phase. This proposal can reduce the startup delay for short web transfers, such as those required for the delivery of DASH MPD and IS. The authors have shown that 90% of HTTP web responses of top sites and Google applications fit in these segments. The authors have show that the average latency of HTTP responses improved by approximately 10% with the largest benefits being demonstrated in high RTT and bandwidth delay product networks. In addition to the regular 3 TCP segments, we will use this init_cwnd of 10 TCP segments in our evaluation.

3.2.2 Evaluation Parameters

In the following evaluation of the different DASH client bootstrap strategies, we consider that the server can send MPD and IS in the initial slow start phase of a TCP connection as they are short file transfers. For that, we assume that there is no packet loss, no delayed acknowledgment and no congestion. Table 3.1 reports the employed parameters throughout the chapter.

Notations	
N	Number of adaptation sets in an MPD
M_j	Number of representations within an adaptation set j
M	Number of representations in an MPD
N_{IS}	Number of IS in an MPD
N_{IS}^{C}	Number of IS chosen by the client at the startup
D_{ss} (A)	Download time of resource A in the slow start phase of a TCP connection using HTTP/1.x (see formula 3.4)
$\acute{D}_{ss}$ (A)	Download time of A in the slow start phase of a TCP connection using HTTP/2 (see formula 3.4)

Table 3.1 – System parameters.

We note N the number of adaptation sets in an MPD. Each adaptation set is indexed by "j". The number of representations within an adaptation set "j" is noted by M_j and can vary based on media content type. The number (M) of representations in an MPD can be expressed in Equation 3.1 as the sum of the number of representations in all adaptation sets.

$$M = \sum_{j=1}^{N} M_j \tag{3.1}$$

Additionally, each representation may have an IS or all representations within an adaptation set may share the same IS. Therefore, the number (N_{IS}) of IS in an MPD is bounded between N and M as expressed in Equation 3.2:

$$N \leq N_{IS} \leq M \tag{3.2}$$

In the bootstrap phase, a client may choose to request a certain number (N_{IS}^{C}) of IS which is bounded between N and N_{IS} as shown in Equation 3.3:

$$N \leq N_{IS}^{C} \leq N_{IS} \tag{3.3}$$

The download time of resource A in the slow start phase of a TCP connection is given below [48]:

$$D_{ss}(A) = \lceil \log(\frac{S \times (\gamma - 1)}{init\_cwnd \times MSS} + 1) \rceil \times RTT + \frac{S}{C} \tag{3.4}$$

where S is the download resource size, γ is set to 2 because we assumed no delayed acknowledgments and C is the network capacity. D_{ss} is composed of the number of RTT required to transfer data in the slow start phase plus the transmission delay.

3.2.3 Evaluating DASH Client Bootstrap Strategies

In this section, we present our first contribution that consists in evaluating five strategies of DASH clients in the bootstrap phase, in terms of number of TCP connections, number of HTTP requests/responses and the associated bootstrap delay. Furthermore, we highlight their advantages and drawbacks. Table 3.2 will summarize the analytical evaluation of the different strategies.

1. Non-Persistent TCP Connection

   Using HTTP/1.0, connections are non-persistent. As depicted in Figure 3.5, this means that a client has to open a new TCP connection to send each HTTP request and receive the MPD file and the chosen IS (N_{IS}^{C}). Hence, the number of open TCP connections is identical to the number of HTTP requests/responses and it equals ($1 + N_{IS}^{C}$) as expressed in Table 3.2. Each connection begins with a TCP three-way handshake which takes a full RTT between the client and the server. Following that,

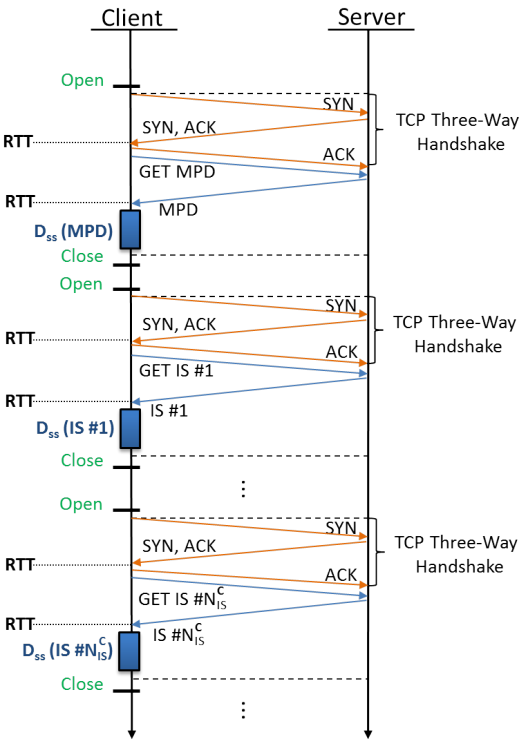


Figure 3.5 – Non-Persistent TCP connection.

the client will incur another RTT to retrieve each resource (MPD or IS) due to the request-response cycle. Finally, we have to add the download time of the resource in the slow start phase to get the total time for every sent HTTP request.

The required bootstrap delay (D_{boot_1}) to fetch the MPD file and the chosen IS when using a non-persistent TCP connection is the sum of the total times for all sent HTTP requests. It is indicated in Equation 3.5.

$$D_{boot_1} = 2 \times (1 + N_{IS}^{C}) \times RTT + D_{ss}(MPD) + \sum_{k=1}^{N_{IS}^{C}} D_{ss}(IS_k) \tag{3.5}$$

2. Persistent TCP Connection Without Pipelining

As depicted in Figure 3.6, TCP connections can be maintained to send and receive multiple HTTP requests/responses when using the keepalive, or the persistent HTTP/1.1 feature. Both names refer to the same mechanism. The server can deliver the associated resources (MPD and IS) over a single TCP connection. This feature allows avoiding connection setup for each IS which eliminates many TCP three-way handshakes and slow start phases. The client incurs only one handshake, plus one slow start phase in the beginning. Using this approach, the server is idle most of the time because it has to wait for its response to reach the client and for the next request to arrive. In some cases, this can trigger a Slow Start Restart (SSR) TCP

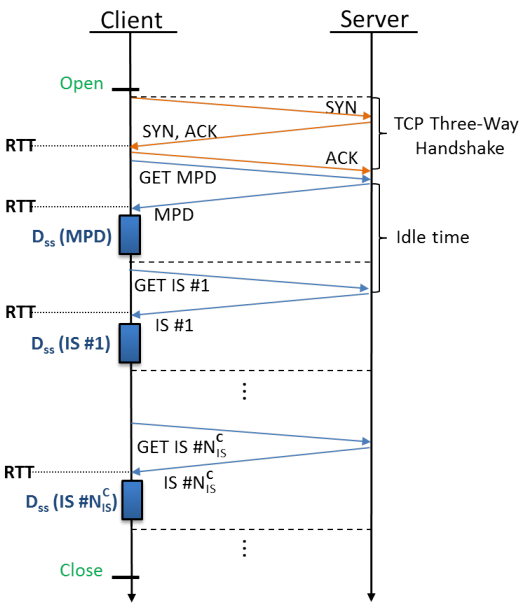


Figure 3.6 – Persistent TCP connection without pipelining.

behavior [49]. The SSR mechanism resets the cwnd to the initial default value after a connection has been idle for a server-defined period of time. It aims at avoiding network congestion, especially since the network conditions may have changed while the connection was idle. The SSR can have a significant impact on performance of long-lived TCP connection that may be idle for bursts of time, e.g. HTTP/1.1 persistent connections. As a result, in our experiment, we disabled SSR on the server.

With keepalive, the first request for the MPD incurs two RTT: one RTT for the TCP handshake and one RTT for the request/response cycle. The following requests for the N_{IS}^{C} files incur only one RTT. Each resource suffers from the download time in the slow start phase. The bootstrap delay for $(1 + N_{IS}^{C})$ HTTP requests/responses delivered via a single TCP connection is indicated in Equation 3.6.

$$D_{boot_2} = (2 + N_{IS}^{C}) \times RTT + D_{ss}(MPD) + \sum_{k=1}^{N_{IS}^{C}} D_{ss}(IS_k) \tag{3.6}$$

3. Persistent TCP Connection With Pipelining

Pipelining is a little improvement of the persistent technique where HTTP requests and responses can be pipelined on a connection, so that the full roundtrip imposed by response and request propagation latencies during which a server is idle is eliminated. A DASH client sends an HTTP request to fetch the MPD and waits for the full HTTP response. Following that, a client makes multiple requests for the IS files without waiting for each individual HTTP response. We note that typically a client

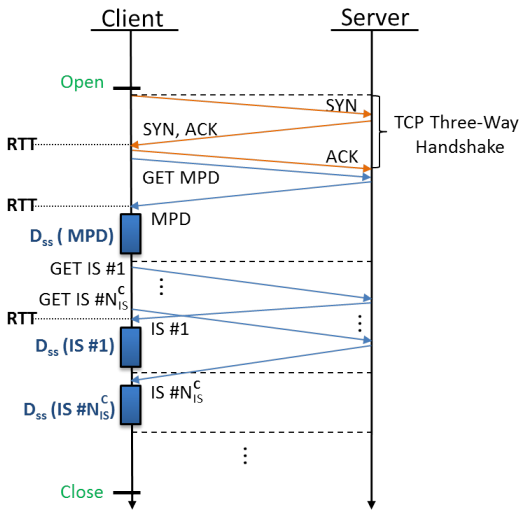


Figure 3.7 – Persistent TCP connection with pipelining.

cannot use the pipelining to get the IS files while it requests the MPD because it has first to receive and parse it for selecting the IS to request.
The server still processes the HTTP requests in sequence, but can respond to a request as soon as the previous one is done. The server sends the responses in the same order as the requests were received which may imply a head-of-line blocking problem [2]. A large or slow response can still block others behind it. A client can incur an unpredictable delay. For a fast startup, a DASH client that will choose to request multiple IS for a given adaptation set has to avoid this problem. For that, it has to dispatch first the requests for the IS required for the initial playback, following by the requests for the first media segments and finally the requests for those extra IS. The head-of-line blocking problem is solved by multiplexing in HTTP/2, but this is not possible in HTTP/1.1. We note that in practice, not all web servers and intermediaries support pipelining.

The number of HTTP requests/responses is $(1 + \mathrm{N}_{\mathrm{IS}}^{C})$ as it is reported in Table 3.2. With the pipelining technique, the client incurs only three RTT: one for the handshake, one for the MPD request/response cycle, and one for the first IS request/response cycle, plus the download time of all retrieved resources to get the bootstrap delay that is expressed in Equation 3.7.

$$D_{boot_3} = 3 \times RTT + D_{ss}(MPD) + \sum_{k=1}^{N_{IS}^{C}} D_{ss}(IS_k) \tag{3.7}$$

4. Parallel TCP connections

In absence of multiplexing in HTTP/1.x, the DASH client is left with no other choice than to open multiple TCP connections in parallel to fetch the chosen IS files. We note that the client has already requestd, received and parsed the MPD for selecting the IS to request before opening the multiple TCP connections that are shown in Figure 3.8. In practice, most modern browsers, both desktop and mobile, open up to six connections per server [2].
The use of parallel TCP connections eliminates the response queue on the server side compared to the persistent TCP connection with pipelining. However, opening multiple connections and performing multiple HTTP transactions in parallel have several disadvantages. It is not always supported by servers. It creates a competition for shared bandwidth between the parallel TCP streams. TCP's mechanisms for starting up the connection and then probing the available bandwidth have to be repeated for each new connection which introduce latency. Finally, the implementation complexity is raised on the client as it has to handle collections of connections.

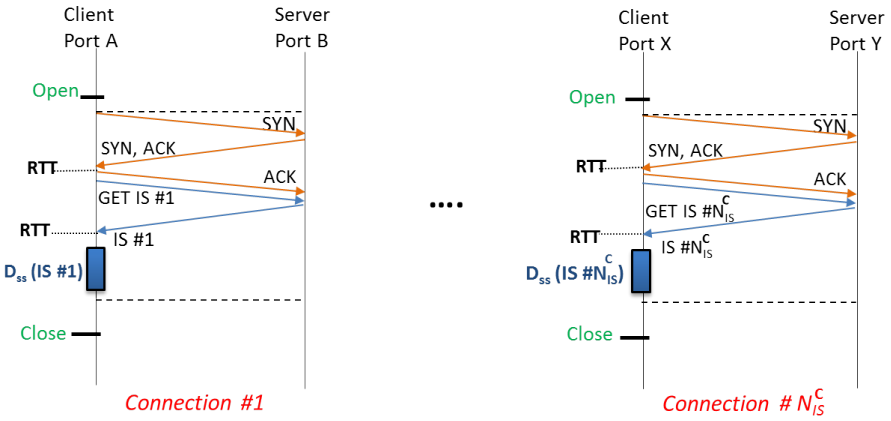


Figure 3.8 – Paralle TCP connections without pipelining to fetch N_{IS}^{C} resources.

In the absence of pipelining, the number of HTTP requests/responses is the same as the number of connections and equals to $(1 + N_{IS}^{C})$. The total delay for every HTTP request is two RTT (one RTT for the handshake and one RTT for the resource request/response cycle) plus the download delay of each resource in the slow start phase. Hence, the bootstrap delay for $(1 + N_{IS}^{C})$ resources is obtained by the sum of the total delay for the MPD request and the maximum of the total delays for the N_{IS}^{C} requests delivered over N_{IS}^{C} parallel connections. It is indicated in Equation 3.8.

$$D_{boot_4} = 2 \times RTT + D_{ss}(MPD) + \max_{k=1,..N_{IS}^{C}} (2 \times RTT + D_{ss}(IS_k)) \tag{3.8}$$

5. HTTP/2 Connection

DASH clients can overcome the limitations of the previous strategies based on HTTP/1.x by using HTTP/2. All HTTP/2 communication is performed within a single connection that can carry any number of bidirectional streams. A stream is a flow of bytes that has a unique integer identifier. It consists in a sequence of one or multiple frames. A frame in turn is the smallest unit in HTTP/2, each containing a frame header, which at minimum identifies the stream to which the frame belongs. The frames may be interleaved and then reassembled via the embedded stream identifier. There are a number of frame types that serve a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses; other frame types like SETTINGS, WINDOW_UPDATE, and PUSH_PROMISE are used in support of other HTTP/2 features. The server push and request-response multiplexing are the most promising features in HTTP/2. The server push is explored in the following examples. HTTP/2 uses true multiplexing that allows many streams (i.e. MPD and IS) to be interleaved together on a connection at the same time, so

that the head-of-line blocking problem is eliminated.

Figure 3.9 illustrates an HTTP/2 client/server communication to download the MPD and IS resources at the frame level with enabled server push. A client sends an HTTP request for the first stream (i.e. MPD) using a HEADERS frame. A web server starts sending the MPD response that consists in a HEADERS frame and one or more DATA frames. With server push, instead of triggering one request for each IS, the web server can actively push all IS (N_{IS}) resources after receiving the first request for the MPD. This is achieved by the server sending PUSH_PROMISE frames to the client to signal its intention to push all IS. According to [50], the PUSH_PROMISE frames must be sent by the server before the end of stream of the requested MPD, i.e. before sending the last DATA frame of the MPD stream, as shown in Figure 3.9. Each PUSH_PROMISE frame includes the identifier of the stream the server plans to create (e.g. the first IS is identified as stream 2 and the last one is identified as stream $(2 + N_{IS})$ in Figure 3.9). We note that the web server sends all present IS in the MPD because a client cannot select the desired IS when requesting the MPD since it ignores the DASH session structure. Therefore, IS and MPD delivery in HTTP/2 are independent.
After transmitting the last DATA frame of MPD, the web server starts pushing a HEADERS frame and one or more DATA frames for each IS response. The client and the web server can exchange other types of frame (e.g. SETTINGS, UPDATE_WINDOW, etc).

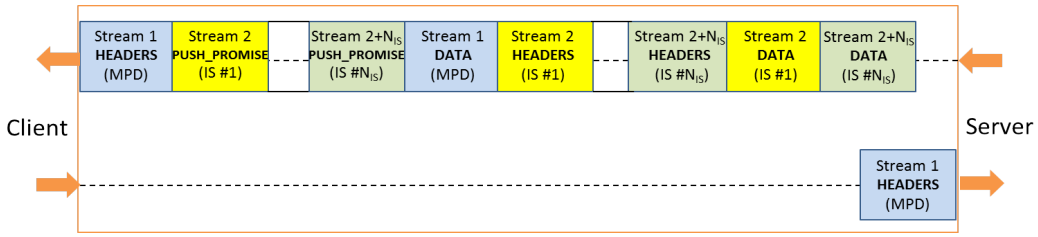


Figure 3.9 – Client request and server responses within an HTTP/2 connection using a server push.

As depicted in Figure 3.10, an HTTP/2 connection starts with a TCP three-way handshake which takes one full RTT as the TCP transport layer does not change. Most client implementations (Firefox, Chrome) support HTTP/2 only over an encrypted connection using Transport Layer Security (TLS) protocol. Unfortunately, establishing a TLS secure channel between the client and the server requires a TLS handshake which takes two RTT or one RTT for abbreviated TLS handshake [2].

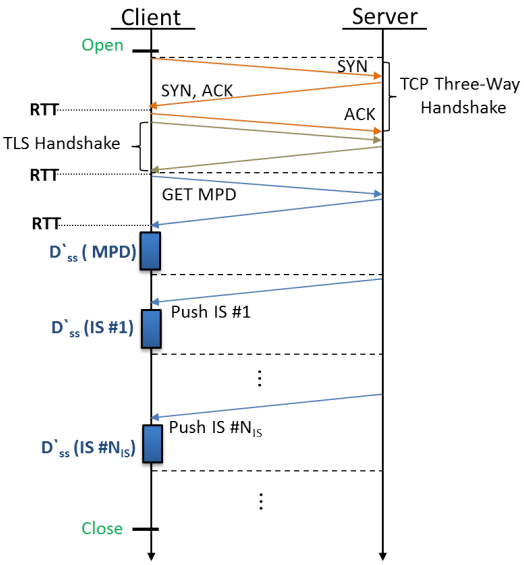


Figure 3.10 – HTTP/2 connection with an abbreviated TLS handshake using a server push.

With server push, the number of HTTP requests/responses is 1. Otherwise, it is (1 + N_{IS}). If we consider the case of an abbreviated TLS handshake, the client incurs only three RTT: one for the handshake, one for the TLS, and one for the MPD request/response cycle, plus the download time of all retrieved resources to get the bootstrap delay, that is indicated in Equation 3.9.

$$D_{boot_5} = 3 \times RTT + \acute{D}_{\text{ss}}(MPD) + \sum_{k=1}^{N_{IS}} \acute{D}_{ss}(IS_k) \tag{3.9}$$

6. Summary

The bootstrap delay formulas of all strategies are summarized in Table 3.2. They are dominated by an RTT component influenced by the number of TCP connections and the number of requests, and by the slow start phase. Based on the analytical evaluation of the strategies presented in Table 3.2, the minimum bootstrap delay using HTTP/1.x is obtained using a persistent TCP connection with pipelining. However, this strategy is not widely supported and still suffers from a big number of RTT mainly due to the number of HTTP requests/responses transfers. In the rest of this chapter, for HTTP/1.x we will experiment only with the persistent TCP connection without pipelining strategy because it is the most used and supported strategy by web servers. Note however that the benefits of our proposed approach would be the same compared to the pipelining approach.

:

Request Strategies	Nb of TCP Connections	Nb of Requests /Responses	Bootstrap Delay
Non-persistent	$1 + \mathrm{N}_{\mathrm{IS}}^{C}$	$1 + \mathrm{N}_{\mathrm{IS}}^{C}$	$D_{boot_1} = 2 \times (1 + N_{IS}^{C}) \times RTT + D_{ss}(MPD) + \sum_{k=1}^{N_{IS}^{C}} D_{ss}(IS_k)$
Persistent	1	$1 + \mathrm{N}_{\mathrm{IS}}^{C}$	$D_{boot_2} = (2 + N_{IS}^{C}) \times RTT + D_{ss}(MPD) + \sum_{k=1}^{N_{IS}^{C}} D_{ss}(IS_k)$
Pipelined	1	$1 + \mathrm{N}_{\mathrm{IS}}^{C}$	$D_{boot_3} = 3 \times RTT + D_{ss}(MPD) + \sum_{k=1}^{N_{IS}^{C}} D_{ss}(IS_k)$
Parallel	$1 + \mathrm{N}_{\mathrm{IS}}^{C}$	$1 + \mathrm{N}_{\mathrm{IS}}^{C}$	$D_{boot_4} = 2 \times RTT + D_{ss}(MPD) + \max_{k=1,..N_{IS}^{C}}(2 \times RTT + D_{ss}(IS_k))$
HTTP/2	1	1 / $1 + \mathrm{N}_{\mathrm{IS}}$	$D_{boot_5} = 3 \times RTT + \acute{D}_{ss}(MPD) + \sum_{k=1}^{N_{IS}} \acute{D}_{ss}(IS_k)$

Table 3.2 – Analytical evaluation of the different DASH client bootstrap strategies.

3.3 Improved DASH Bootstrap

In this section, we present our new approach to reduce the bootstrap delay. It consists in using a single HTTP request and HTTP response to retrieve the necessary information to start the playback. The first HTTP request made by the DASH client to retrieve the MPD is not modified, but the response sent by the origin server is. The principle of creating this HTTP response is to rely on the MPD to carry the additional IS resources. This can be done in three ways which are detailed below.

1. Base64 IS Embedding

   A simple option is to encode the binary IS into an ASCII string using the Base64 encoding for binary-to-text encoding. The encoding process consists in representing groups of 3 bytes (24 bits) of input bits as output strings of 4 encoded ASCII characters. The encoded IS is then put in the MPD file, in the initialization attribute, using the data URI scheme[2]. The data URI scheme allows embedding an arbitrary resource directly in files in any attribute that can use URL. It has the following syntax: data:[<mediatype>][;base64],<data>. Figure A.1 shows an MPD file example with Base64 IS embedding. When a client receives the MPD, it will need to decode the Base64 string to recover the original binary IS. The advantage of this naïve

[2]http://tools.ietf.org/rfc/rfc2397.txt

method is its compatibility with the current DASH standard. The drawback is that the base64 encoding expands encoded stream by a factor of 4/3, incurring a 33% byte overhead. It may therefore not be acceptable but is a good anchor point.

```
<MPD ...>
 <Period ...>
  <AdaptationSet ...>
   <Representation mimeType="video/mp4" ... >
                   codecs="avc1.4d401f" ...>
    <SegmentTemplate
       initialization="data:video/mp4;base64,
       AAAAGGZOeXB..."/>
   </Representation>
 <AdaptationSet ...>
   <Representation mimeType="audio/mp4" ... >
                   codecs="mp4a.40.5" ...>
    <SegmentTemplate
       initialization="data:audio/mp4;base64,
       AAAAFGZ0eXB..."/>
   </Representation>
   ...
</MPD>
```

Figure 3.11 – Base64 IS embedding in MPD.

2. Multipart Content Embedding

In DASH, each HTTP response contains only one entity in its body. Our second proposed option consists of the combination of different entities of independent data types (MPD and IS) in the single body of the HTTP response, using HTTP/1.1 "multipart" media type. To respond to the HTTP request made by the DASH client to retrieve the MPD, the server is expected to send a single HTTP response body carrying the MPD part and the IS parts as shown in Figure A.2.

```
--?--?--
Content-Type:application/dash+xml
Content-Disposition:attachement;filename=MPD.mpd
Content-Transfer-Encoding:base64
Content-Length: 3343

<MPD ...>
 <Period ...>
  <AdaptationSet ...>
   <Representation mimeType="video/mp4" ... >
    codecs="avc1.4d401f" ...>
   </Representation>
<MPD ...>
--?--?--
Content-Type:video/mp4;codecs=avc1
Content-Disposition:attachement;filename=mysegs_mp4-onDemand-
h264bl_fullinit.mp4
Content-Transfer-Encoding:base64
Content-Length: 838

tdmhkAQAAAAAAADM9P...
--?--?--
```

Figure 3.12 – "Multipart/mixed" content-type of MPD and IS entities.

In the case of multipart entities, a ”multipart” content type field must appear in the entity's header of the HTTP response. The multipart body can contain one or more body parts, each preceded by a boundary delimiter, and the last one followed by a closing boundary delimiter. Each body part consists of a header area, a blank line, and a body area.

In our approach, we used the ”multipart/mixed” subtype because MPD and IS body parts are independent. We added HTTP headers in each body start including content-type header for giving the media type of this content, content-length for specifying the length in bytes of each body part, content-disposition for naming each part with a corresponding name in the MPD, and content-transfer-encoding for indicating what type of transformation has been applied to the body part.

In the Base64 IS embedding method, only Base64 IS are embedded in the MPD file to constitute the HTTP response body. In the Multipart content embedding method, the HTTP response body is a set of parts (MPD and Base64 IS), and for each part several HTTP headers are added. Consequently, this latter method introduces more overhead than the former. Note that multipart method were ruled out in our evaluations in Section 3.4

3. ISOBMFFMoov Embedding

   Our third option consists in adding to the MPD the information required to reconstruct the IS at the client side from that MPD only. For that, we first looked more closely to the IS content and structure, i.e. which information an IS contains and how it is organized. Following that, we compared the MPD and IS of different contents to identify the potential common and missing pieces of information. Finally, we created a new element in the MPD to carry the missing information. Each step is described and detailed below.

IS Structure

Conforming to the ISOBMFF, an IS is structured as a series of boxes that can be organized sequentially and hierarchically. An IS contains the metadata for the whole presentation, which is stored in a single movie box (”moov”). It describes the encoding of the media content (elementary stream), specifically of the representation. It stores the metadata of each media content in a track box (“trak”), which is subsequently grouped with the others if any in the movie box (”moov”). In contrast to a media segment, IS shall not contain any media data. Figure 3.13 provides an overall view of the IS structure for unencrypted media contents and shows the possible information containment.

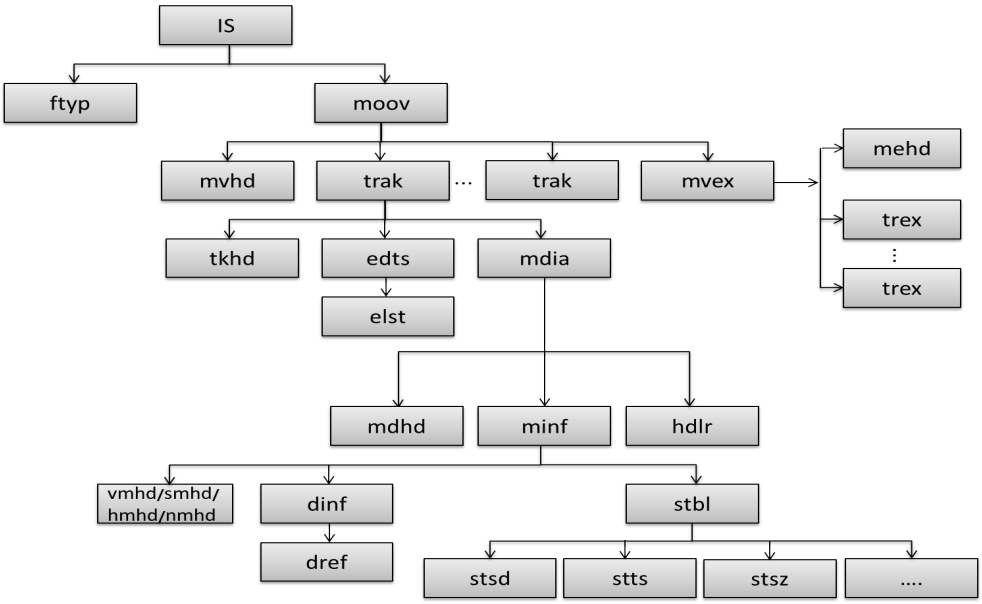


Figure 3.13 – IS structure.

A brief description of the main boxes constituing an IS file is provided in Table 3.3.

Table 3.3 – Description of the main IS boxes.

Box Type	Box Name	Description
ftyp	file type	identifies the specifications to which this file is conformant.
moov	movie	contains all the metadata for the presentation.
mvhd	movie header	defines generic information about the presentation (e.g. presentation duration, presentation timescale).
trak	track	contains all the metadata about an individual media content type or stream.
tkhd	track header	defines the caracteristics of a single track (e.g. track duration, track identifier).
edts	edit	contains edit lists, typically used to adjust synchronization.
elst	edit list	maps the presentation timeline including the edit lists to the media timeline as it is stored in the file.
mdia	media	contains information about the media data within a track.

Continued on next page

Table 3.3 – *Continued from previous page*

Box type	Box name	Description
mdhd	media header	contains the caracteristics of the media in a track (e.g. media duration, media timescale, media language).
hdlr	handler reference	indicates the type of the track or the type of metadata.
minf	media information	container for all media information in the track.
vmhd/smhd/hmhd/nmhd	video/sound/hint/null media header	each track has a specific media information header that contains the overall information.
dinf	data information	contains information that declares the location of the media information in a track
dref	data reference box	declares the location of the media data used within the presentation.
stbl	sample table	contains sample signaling information.
stsd	sample description	signals elementary stream decoder configuration.
stts	Sample To Time	indicates the time at which a sample should be decoded (Decoding Time Stamp (DTS)). It is empty.
stsz	Sample To Size	signals the size of each sample. It is empty.
mvex	movie extend	indicates that the client has to expect movie fragments.
mehd	movie extends header	gives the duration of the movie with all fragments (not in live).
trex	track extends header	assigns for each track the default sample properties that are used by the movie fragments.

IS and MPD Information Analysis

We analyzed the MPD file and the IS of different content. We gathered the common information and we identified the information which is only present in the IS. Table 3.4 summarizes the information comparison of the MPD and IS.

Information	MPD	IS
Media presentation duration	×	×
Samples configuration		×
Track ID		×
Content type	×	×
Video resolution	×	×
Media language	×	×
Decoder configuration		×
Edit list		×

Table 3.4 – MPD and IS information comparison.

According to Table 3.4, it appears that four useful information present in the IS are also present in the MPD. In the following, we give an overview of that common information and how it is designed in the MPD and IS. We also show the containment of each information, i.e. in which box type or element is stored.

(a) Media presentation duration corresponds to the length of the media display. In the MPD, this information is specified in a *media presentation duration* attribute, which is found in an MPD element. In the IS, it is indicated in a *fragment_duration* field of a movie extends header box (”mehd”) in case of a fragmented content.

(b) Video resolution is one of the most important visual characteristic of any video. Width and height specify the horizontal and the vertical visual presentation size of the video media type. They are designed in *width* and *height* attributes, which are placed in a representation element of the MPD. In the IS, they are identified by *width* and *height* fields in a track header box (”tkhd”), which is found in a track box (”trak”). This means that they are specific for each track.

(c) Content type specifies the nature of media content for the adaptation set such as audio, video, and subtitle. It may be defined for each media component when multiple media contents are mutiplexed in an adaptation set through a *content type* attribute. In the IS, it is specified by a handler reference box (”hdlr”), which is contained in a media box (”mdia”), which is in turn found in a track box (”trak”).

(d) Media language declares the language code for the media. It is defined by a *lang* attribute, which is placed in an adaptation set element if media representations of an adaptation set share the same language. Otherwise, a *lang* attribute is found for each media content in a media content component element if media

contents of an adaptation set are multiplexed. In IS, it is defined by a *language* field in a media header box ("mdhd").

From Table 3.4, we identified four potential missing pieces of information from the MPD:

(a) Decoder configuration in ISOBMFF is stored in the sample description box ("stsd"). As shown in Figure 3.13, "stsd" box is contained in a sample table box ("stbl"). For some video packaging types (identified by "avc3" and "hev1"), this box is mostly empty, can be reconstructed from the MPD information, and can therefore be omitted. For other packaging types such as audio, subtitle or some video (identified by "avc1", "hvc1" or others), the box does contain information required by the client and has to be embedded in the MPD.

(b) Samples configuration includes the default sample properties which are stored in the track extends box ("trex"), and the sample group configurations box ("sgbd") which is contained in a sample table box ("stbl"). Default sample properties are the default values for size, duration, description index, flags that are may be required to parse the movie fragments. Sample group configurations give information about the characteristics of sample groups such as random access or pre-roll.
Default sample properties and sample group configurations can be configured at the IS file level or for each media segment. In some profiles such as Common Media Application Format (CMAF) [51], the properties are defined only at the media segment level. In our approach, we follow this profile and we assume that this information is not needed to be exposed in the MPD.

(c) A track may have an edit list, which shall be sent to the client to ensure proper synchronization. In all DASH cases however, the edit list only consists in a single time offset used to adjust synchronization between tracks, either advancing or delaying the playback position of one track compared to other tracks. This offset has to be embedded in the MPD.

(d) Track ID is a unique value that identifies the track over the entire life-time of the media presentation. In the case of multiplexed media contents such as audio and video, track ID is required in the MPD.

ISOBMFFMoov Element

From the MPD and IS analysis and comparison, we introduced a new <ISOBMFF-Moov> element for each representation in the MPD. Because MPD is an XML file supporting only text, each binary IS information such as the sample description box ("stsd") must be encoded into text using the Base64 encoding before embedding it in the MPD. Each <ISOBMFFMoov> element carries for each track, through an ISOBMFTrack element, its ID, the base64 stsd box and the edit list as a media offset to the MPD timeline as shown in Figure A.3. Note that the proposed <ISOBMFF-Moov> element could be extended to handle other file or track level boxes, such as static meta and encryption boxes.

```
<MPD ...>
 <Period ...>
  <AdaptationSet ...>
   <Representation mimeType="video/mp4" ... >
                   codecs="avc1.4d401f" ...>
    <ISOBMFFMoov>
      <ISOBMFTrack id="1"
                   stsd="AAAlHN0c2QA..."
                   editDelay="0.04s"/>
    </ISOBMFFMoov>
    <SegmentTemplate ... />
   </Representation>
   ...
```

Figure 3.14 – ISOBMFFMoov Embedding in MPD.

Our proposed approach is slightly similar to the IIS Microsoft Smooth Streaming existing approach[3]. However, this latter does not use at all IS files for decoder initialization. Manifests carry neither edit lists nor track ID. It carries only the decoder configuration information. This approach is not suitable for generic ISOBMFF content.

With our approach, we need to decide which IS to embed in the MPD. On the one hand, we can embed all IS of the adaptation sets in the MPD. Regarding caching, the MPD is always present in the caches as it is the first resource that each client has to fetch from the edge server. The caching of IS files depends on the clients. The non-popular IS, i.e. those that are not yet requested by any client, are not present in the caches. Embedding all IS in the MPD in our approach has a benefit of avoiding cache miss on non-popular IS and ensures a 100% cache hit ratio.
On the other hand, a client can select the desired IS files when requesting the MPD although it ignores what a server has as resources. For that, we propose to send in the single HTTP request for the MPD additional information that express the client needs for

[3]http://www.iis.net/downloads/microsoft/smooth-streaming

IS files regarding its capabilities and preferences. Based on that request information, the server may take a decision on which IS to send to client. The first idea is to use HTTP request headers to pass additional information about the client to the server. We use for instance an "Accept" request header to tell the server what desired MIME-media types a client is looking for (e.g. audio, video, text, application, etc). Additionally, we can specify the codecs that a client supports. We can also indicate the languages that are acceptable for the response through "Accept-language" request header. We could add another header to indicate the quantity of IS that a client would like to receive. This would however be less cache efficient, so we did not use this approach.

3.4 Evaluation

3.4.1 Settings

Our proposed approach has been evaluated under two different network types including: a DSL network and a mobile network with 3G technology. In the following, we present the test-bed architecture as well as the settings of each network.

3.4.1.1 DSL Netwrork

Figure 3.15 depicts the architecture of the experimental system that we used for emulating a DSL network. It consists of four components: a web client, a bandwidth shaper, a network emulator, and a web server, connected via Ethernet in a local area network. The network emulator component was used to add a delay to obtain an RTT value of 50 ms using the Linux Emulator Network (Netem) [52]. Based on the bandwidth shaper component, we limited the maximum outgoing bandwidth to 2 Mbps from the server to the client using Linux Traffic Control (TC) [53] command line tool and the Hierarchical Token Bucket (HTB) [53]. The Network emulator and the bandwidth shaper were running on the web server. These settings were set following Google Chrome's network throttling settings.

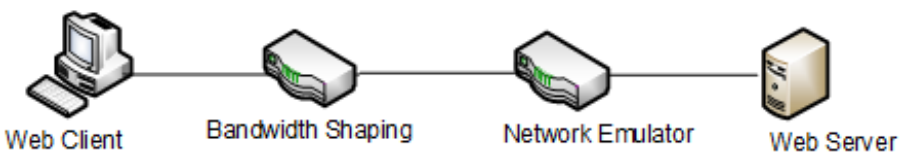


Figure 3.15 – Experimental setup for emulating a DSL network.

We implemented a web server that supports HTTP/1.1 and HTTP/2 on top of NodeJS[4]. In the case of HTTP/2, we used the server push mechanism to start pushing all the IS resources as soon as the server receives the client request for the MPD. If the client does not want the pushed IS files, it can reject it. We used the Chrome Canary browser and the Dash-JS video player which is based on XMLHttpRequest (XHR).

Because init_cwnd is a critical parameter in determining how quickly the DASH streaming session can start, our experiments were evaluated under two different values of init_cwnd: 3 and 10 TCP segments. The init_cwnd was configured on the server side that runs Ubuntu with the default congestion control algorithm "TCP Cubic", using the "initcwnd" option of the ip route command. The Maximum Transmission Unit (MTU) allowed by Ethernet is set to 1500 bytes. When we exclude IP and TCP headers from the MTU, it remains a Maximum Segment Size (MSS) with 1460 bytes. Hence, the init_cwnd of 3 TCP segments is approximately 4380 bytes (i.e. init_cwnd = 3*MSS) and the init_cwnd of 10 TCP segments is about 14 600 bytes (i.e. init_cwnd = 10*MSS).

3.4.1.2 Mobile Network

Figure 3.16 shows a simple mobile test-bed infrastructure. It comprises a web server supporting HTTP/1.1 on top of NodeJS, a web client running Dash-JS video player in the Google Chrome browser, and a smartphone device equipped with a 3G interface. We used the hotspot feature on the smartphone to share the mobile data connection with the host running a web client to create a real-world mobile environment. We used Wi-Fi to tether the smartphone to the host.

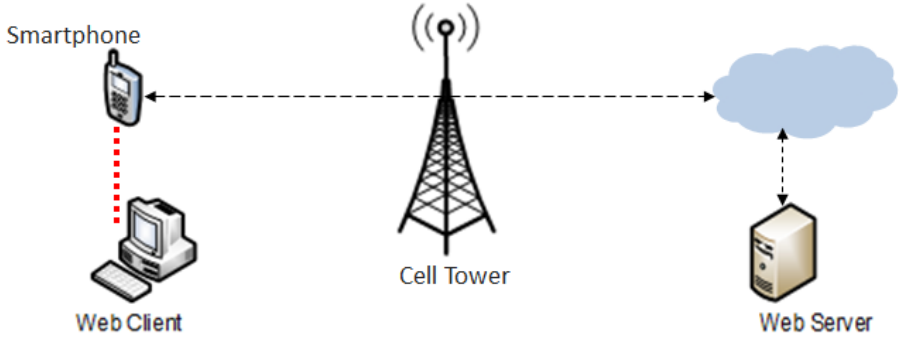


Figure 3.16 – Experimental setup for a simple mobile network.

[4]https://nodejs.org/

3.4.2 Dataset

We used ISOBMFF live profile DASH content from the DASH Industry Forum (DASHIF)[5]. We have selected 33 sequences (MPD files and associated IS), for which multiple bitrates, resolutions, frame rates, languages are available, as it is summarized in Table 3.5. Common encryption test sequences were not selected due to some authoring issues in the source IS (many/a lot of padding data). Some sequences could not be downloaded from the DASHIF server as they were moved or deleted. Some MPDs include multiple periods. We considered only IS files that belong to the first period because those ones have to be downloaded in the bootstrap phase for the initial playback.

Video			Audio		
Bitrate (Kbps)	Resolution	Frame Rate	Bitrate (Kbps)	Frame Rate	Language
500	320x240	24	33	24	english
900	512x288	25	64	48	french
1000	640x360	29.97	96		
1500	720x480		128		
2000	768x432				
2500	1280x720				
3000	1920x1080				
4000					
8000					

Table 3.5 – Characteristics of the selected 33 sequences from DASHIF.

According to Table 3.1, the number of adaptation sets (N) in each MPD equals to 2: video and audio. The number of representations within a video adaptation set (M_1) varies and is bounded between 2 and 4. Video representations differ in bitrates and resolutions. Videos identified by "avc1" and "avc3" are present. Additionally, audio adaptation set (M_2) contains only 1 representation, using the "aac" codec. Hence, the number of representations (M) in MPDs varies between 3 and 5. None of these sequences uses shared IS among representations. Each representation consists of one IS file. Therefore, the number of IS in MPDs (N_{IS}) varies between 3 and 5.

3.4.3 Experiments And Results

In order to validate our proposal presented in Section 3.3, we have conducted two types of experiments. First, experiments to measure the total download size of the MPD and IS files, including the HTTP response headers, for three methods: the persistent TCP connection without pipelining method, and our two proposed methods (Base64 IS Embedding and

[5]http://dashif.org/testvectors/

ISOBMFFMoov Embedding). Depending on the amount of IS to download in the bootstrap phase, two strategies were evaluated for the regular persistent method. One strategy downloads all IS of the adaptation sets in the bootstrap phase preparing for futur switching. The other one downloads only the IS needed to start the video playback.
The second serie of experiments consists in measuring and comparing the bootstrap delay in a DSL network between the persistent TCP connection without pipelining strategy when using Dash-JS player, our ISOBMFFMoov Embedding proposal using HTTP/1.1 and HTTP/2, and the HTTP/2 push-based approach. In a 3G mobile network, we also measured the bootstrap delay for our ISOBMFFMoov Embedding approach compared with the persistent TCP connection without pipelining strategy. Finally, we computed the startup delay and the percentage of bootstrap delay.

3.4.3.1 Total Download Size

This section presents, step by step, the process used to measure the total download size of our test sequences when using four different methods.

1. We first downloaded the 33 test sequences from the DASHIF server. Each sequence consists of an MPD file and associated IS for all representations of the video and audio adaptation sets. We measured the size of each downloaded resource as well as the size of its HTTP response header. Table A.1 represents for each sequence the MPD size, the IS size for the only representation in the audio adaptation set, and finally the IS size for all representations in the video adaptation set.

Table 3.6 – MPD and IS sizes of 33 sequences.

Sequence Number	MPD Size (Byte)	IS Audio Size (Byte)	IS Video Size (Byte)
0	1597	615	687, 687
1	1482	656	720, 720
2	2251	776	841, 841, 841
3	2358	676	841, 841, 841
4	1766	615	687, 687
5	2332	776	841, 841, 841
6	2333	776	841, 841, 841
7	1773	656	720, 724, 721, 719
8	1773	656	720, 724, 721, 719
9	2371	676	839, 839, 841
10	2326	776	839, 839, 841

Continued on next page

Table 3.6 – *Continued from previous page*

Sequence Number	**MPD Size (Byte)**	**IS Audio Size (Byte)**	**IS Video Size (Byte)**
11	2384	776	840, 840, 841
12	2388	776	840, 840, 841
13	3939	776	841, 841, 841
14	3181	776	841, 841
15	3172	776	841, 841
16	4125	776	841, 841, 841
17	5768	776	841, 841, 841
18	5010	776	841, 841
19	5271	776	841, 841, 841
20	4098	776	841, 841, 841
21	3340	776	841, 841
22	4095	776	841, 841, 841
23	3337	776	841, 841
24	2953	776	841, 841, 841
25	2945	776	841, 841, 841
26	3383	776	841, 841, 841
27	2704	776	848, 848, 848, 848
28	2696	776	848, 848, 848, 848
29	3114	776	848, 848, 848, 848
30	2768	776	846, 846, 847, 848
31	2719	776	846, 846, 847, 848
32	3178	776	846, 846, 847, 848
Maximum	**5768**	**776**	**848**
Average	**2998**	**755**	**824**
Minimum	**1482**	**615**	**687**

At the end of the Table A.1, we report the maximum, average, and minimum sizes of the MPD and IS sizes. We can see first that they are small size resources. We can note that the MPD size varies between arround 1 KB and 6 KB. The IS video size is within the range 600 bytes to 900 bytes while the IS audio size varies between arround 600 bytes and 800 bytes.

The average HTTP response header size is 487 bytes for the response carrying an IS video or audio content. It is 499 bytes for the response involving an MPD data. We also measured the average of HTTP request header size which is 191 bytes when

requesting the MPD and it is 204 bytes when fetching the IS file. The large size of those HTTP response headers is due to long strings used for cache information (Etag, modified dates) and to CORS (Cross-Origin Resource Sharing) headers that DASHIF server sends back for access control requests as defined by the CORS specification [6]. Figure 3.17 shows what headers the DASHIF server can send to the client making a simple GET request for the MPD resource (manifest.mpd) of the sequence number 0. The server sends Access-Control-Allow-Headers with a value of "origin, range, accept-encoding, referer", confirming that these are permitted headers to be used with the actual request. It also responds with Access-Control-Allow-Methods indicating that GET, HEAD and OPTIONS are acceptable methods to query the resource in question. Access-Control-Allow-Origin header allows the server to describe the set of origins that are permitted to read that information using a web browser.

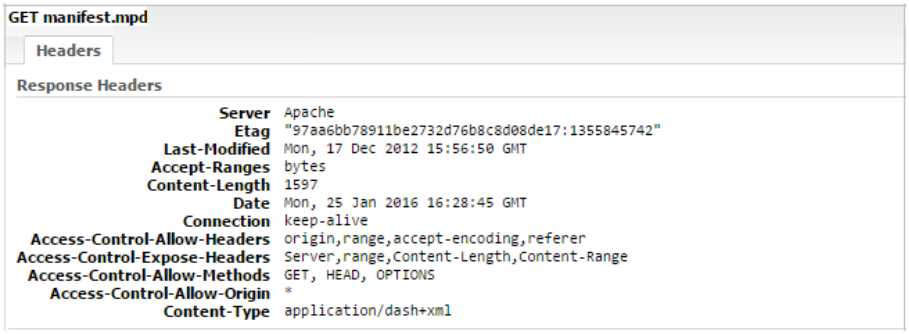


Figure 3.17 – Example of an exchange of HTTP headers between client and DASHIF server.

2. Based on our 33 test sequences, we generated new MPD files (i.e. MPD with Base64 IS embedding and ISOBMFFMoov-based MPD) in conformance to our two proposed methods. We then measured the size of those MPDs. The results are provided in Table 3.7.

Table 3.7 – Generated MPD size in bytes for Base64 IS embedding and ISOBMFFMoov embedding methods of 33 sequences.

Sequence Number	MPD Base64 IS Size (Byte)	MPD ISOBMFFMoov Size (Byte)
0	4502	2429
1	4475	2232
2	6604	3036
3	6675	3107
4	4724	2651
5	6614	3046

Continued on next page

[6] https://www.w3.org/TR/cors/

Table 3.7 – *Continued from previous page*

Sequence Number	MPD Base64 IS Size (Byte)	MPD ISOBMFFMoov Size (Byte)
6	6615	3047
7	6902	3149
8	6902	3149
9	6674	3114
10	6651	3091
11	6630	3070
12	6656	3096
13	8275	4707
14	6408	3735
15	6435	3762
16	8451	4883
17	10116	6548
18	8249	5576
19	9608	6040
20	8371	4803
21	6504	3831
22	8388	4820
23	6521	3848
24	8380	3921
25	8375	3916
26	8665	4206
27	8187	3740
28	8182	3735
29	8472	4025
30	8216	3765
31	8191	3740
32	8501	4050
Maximum	**10116**	**6548**
Average	**7367**	**3814**
Minimum	**4475**	**2232**

At the end of the Table 3.7, we report the maximum, average, and minimum sizes of the generated MPDs for both approaches for all sequences. We can see that the generated MPD with embedded Base64 IS size varies between around 4KB and 10KB.

We can note that the generated MPD with embedded ISOBMFFMoov varies between arround 2KB and 7KB.

3. We calculated the total download size of each sequence when using our two methods (Base64 IS embedding, ISOBMFFMoov embedding), i.e. the size of the generated MPD plus the size of the HTTP header of the MPD response.
   We compared those results with the total download size of the persistent TCP connection without pipelining when :

   - N_{IS}^{C} is maximal (i.e. equal to M) as implemented by GPAC player .
   - N_{IS}^{C} is minimal (i.e. equal to N) as implemented by Dash-JS player where the average IS size of video adaptation set is used for each sequence.

   The total download size of these two last methods is obtained from the sum of the MPD size, the size of N_{IS}^{C} IS for the representations of the video and audio adaptation sets, and their HTTP headers. For that, we used the previous size measurements (i.e. MPD size, IS size, and their HTTP response headers sizes) summarized in Table A.1.

   Table A.2 represents for each method the total download size of 33 sequences. We give below a detailed example showing how a total download size for each method was calculated for the first sequence of number 0.

   - When the GPAC player downloads all IS for the representations of the audio and video adaptation sets for sequence 0 ($N_{IS}^{C} = M = 3$ as shown in Table A.1), the total download size is defined as the sum of MPD size, all IS video and audio sizes, MPD HTTP response header size, HTTP response header size of each IS. It equals 5546 bytes ($5546 = 1597 + 615 + 687 + 687 + 499 + 487 + 487 + 487$) as presented in Table A.2.

   - For the Dash-JS player that downloads the minimal amount of IS ($N_{IS}^{C} = N = 2$), the total download size is the sum of the MPD size, IS audio size, average IS size of video adaptation set, and their HTTP response headers. We obtained 4372 of downloaded bytes ($4372 = 1597 + 615 + avg(687 + 687) + 499 + 487 + 487$) as shown in Table A.2.

   - In ISOBMFFMoov embedding approach, the total download size is the sum of the generated MPD with embedded ISOBMFFMoov size, and the HTTP header size of the MPD. As presented in Table A.2, it is 2928 bytes ($2928 = 2429 + 499$).

- The total download size when using the Base64 IS embedding approach is the sum of the genrated MPD with embedded Base64 IS size, and the header size of the MPD. It equals 5001 bytes (5001 = 4502+499) as shown in Table A.2.

Table 3.8 – Total download size (MPD, IS video, IS audio, and HTTP response headers) of 33 sequences for each method over HTTP/1.1 using a DASHIF server.

Sequence Number	N_{IS}^{C}=M (GPAC)	N_{IS}^{C}=N (Dash-JS)	MPD Base64 IS	MPD ISOBMFFMoov
0	5546	4372	5001	2928
1	5538	4331	4974	2731
2	7997	5341	7103	3535
3	8104	5448	7174	3606
4	5715	4541	5223	3150
5	8078	5422	7113	3545
6	8079	5423	7114	3546
7	8247	4623	7401	3648
8	8247	4623	7401	3648
9	8113	5459	7173	3613
10	8068	5414	7150	3590
11	8128	5473	7129	3569
12	8132	5477	7155	3595
13	9685	7029	8774	5206
14	7599	6271	6907	4234
15	7590	6262	6934	4261
16	9871	7215	8950	5382
17	11514	8858	10615	7047
18	9428	8100	8748	6075
19	11017	8361	10107	6539
20	9844	7188	8870	5302
21	7758	6430	7003	4330
22	9841	7185	8887	5319
23	7755	6427	7020	4347
24	10026	7370	8879	4420
25	10018	7362	8874	4415
26	10456	7800	9164	4705
27	9806	5801	8686	4239

Continued on next page

Table 3.8 – *Continued from previous page*

Sequence Number	N_{IS}^{C}=M (GPAC)	N_{IS}^{C}=N (Dash-JS)	MPD Base64 IS	MPD ISOBMFFMoov
28	9798	5793	8681	4234
29	10216	6211	8971	4524
30	9865	5863	8715	4264
31	9816	5814	8690	4239
32	10275	6273	9000	4549
Maximum	**11514**	**8858**	**10615**	**7047**
Average	**8793**	**6168**	**7866**	**4313**
Minimum	**5538**	**4331**	**4974**	**2731**

At the end of the Table A.2, we report the maximum, average, and minimum sizes of the total download size for the 33 DASHIF sequences. We can see first that data size remains small ($\leq$ 12 KB) in all approaches. We can notice that our ISOBMFFMoov-based embedding method reduces the downloaded data size by an average of 36% compared to the approach that downloads the minimal amount of IS and the MPD separately (as implemented in Dash-JS). As we can also see, the strategy used by GPAC, which downloads all IS to prepare for future switches, always leads to more bytes downloaded than Dash-JS. Interestingly also, we can see that downloading all IS in one single HTTP response using Base64 encoding may lead to a smaller amount of data being downloaded compared to the GPAC method.

4. Finally, we used a NodeJS-based web server to serve our ISOBMFFMoov-based MPDs using HTTP/1.1 and HTTP/2, and to serve separatley the MPD and the minimal amount of IS (i.e. one IS video and one IS audio) using HTTP/1.1 and HTTP/2 with and without enabled server push. For each method, we measured the total download size of those resources from the transfer size field in the Network Panel of Google Chrome. Table 3.9 reports for each method the total download size of the 33 sequences.

Table 3.9 – Total download size (MPD, IS video, IS audio, and HTTP response headers) of 33 sequences for each method over HTTP/1.1 and HTTP/2 using a nodeJS-based server.

Sequence Number	N_{IS}^{C}=N HTTP/1.1 Dash-JS	HTTP/1.1 MPD ISOBMFFMoov	N_{IS}^{C}=N HTTP/2 With Push (Dash-JS)	N_{IS}^{C}=N HTTP/2 Without Push (Dash-JS)	HTTP/2 MPD ISOBMFFMoov
0	3471	2617	3234	3172	2629
1	3400	2420	3201	3154	2432
2	4410	3224	4210	4141	3235
3	4517	3295	4325	4271	3306
4	3640	2839	3403	3364	2850
5	4491	3234	4291	4222	3245
6	4492	3235	4292	4223	3246
7	3690	3337	3491	3421	3348
8	3690	3337	3491	3421	3348
9	4528	3302	4337	4259	3313
10	4483	3279	4285	4214	3291
11	4542	3258	4352	4273	3269
12	4546	3284	4348	4300	3296
13	6098	4896	5899	5852	4915
14	5340	3923	5141	5071	3934
15	5331	3950	5132	5085	3961
16	6285	5072	6094	6024	5091
17	7928	6737	7737	7667	6756
18	7170	5765	6979	6932	5785
19	7431	6229	7240	7170	6248
20	6258	4992	6066	5997	5012
21	5499	4019	5299	5230	4030
22	6254	5009	6054	5985	5028
23	5496	4036	5297	5227	4047
24	5112	4109	4913	4843	4120
25	5104	4104	4903	4835	4115
26	5542	4395	5343	5273	4414
27	4870	3928	4674	4601	3939
28	4862	3923	4664	4593	3934

Continued on next page

Table 3.9 – *Continued from previous page*

Sequence Number	N_{IS}^{C}=N HTTP/1.1 Dash-JS	HTTP/1.1 MPD ISOBMFFMoov	N_{IS}^{C}=N HTTP/2 W Push (Dash-JS)	N_{IS}^{C}=N HTTP/2 WO Push (Dash-JS)	HTTP/2 MPD ISOBMFFMoov
29	5280	4213	5084	5011	4224
30	4932	3953	4742	4663	3964
31	4883	3928	4688	4614	3939
32	5342	4238	5152	5073	4250
Maximum	**7928**	**6737**	**7737**	**7667**	**6756**
Average	**5119**	**4002**	**4920**	**4854**	**4016**
Minimum	**3400**	**2420**	**3201**	**3154**	**2432**

At the end of Table 3.9, we represent the maximum, average, and minimum of the total download size for the mentioned methods using a NodeJS web server. For ISOBMFFMoov embedding and Dash-JS based HTTP/1.1 methods, we can note that the total download size of resources is less than when fetching them from the DASHIF server (e.g. Apache server) as shown previously in Table A.2. This is due to the decrease in the amount of HTTP response headers. Figure 3.18 shows that no cache headers are present. Only Access-Control-Allow-Origin from the CORS headers is available.

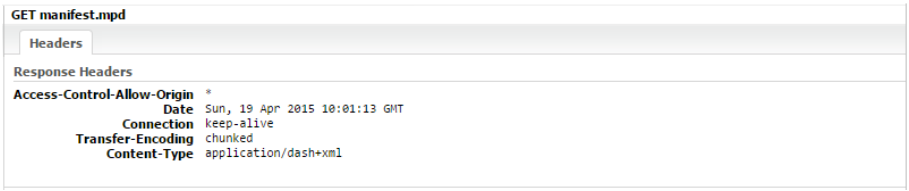


Figure 3.18 – HTTP response header of MPD when using a nodeJS-based web server.

The total download size is reduced by approximately by 25% when using our method over HTTP/1.1 compared to the amount of data being downloaded for the method that uses Dash-JS player over HTTP/1.1.

We can see that the total download size of the Dash-JS method without push over HTTP/2 is slightly less than over HTTP/1.1 by an avearage of 6%. This is possibly due to the HTTP/2 header compression that is applied by default.

Additionally, we can see that when using our method over HTTP/2, the total download size is reduced by an average of 21% compared to the HTTP/2 push method. To investigate each HTTP/2 session, we used the Net Internals console of Google Chrome that gives the raw output of the HTTP/2 streams and frames. Figure

3.19 and 3.20 present the logs of the HTTP/2 push method and our ISOBMFFMoov embedding approach over HTTP/2, in which we index each frame type used in HTTP request or response messages by a color bloc: green for HEADERS frame type, purple for DATA frame type, and red for PUSH_PROMISE frame type.
Based on Figure 3.19, we notice that the extra size in the HTTP/2 push method is firstly due to the two PUSH_PROMISE frames that the server needs to notify the client in advance of the two IS video and audio that it intends to send. Each PUSH_PROMISE frame contains the HTTP headers and the stream identifier of the promised resource. For example, video and audio IS are identified respectively as stream 2 and stream 4.
Secondly, the HTTP/2 push method requires two response messages to send the video and audio IS. Each response contains three frames, i.e. one frame is the HEADERS frame carrying the HTTP response header and two frames are DATA frames containing the IS data. In our approach however, only one response message is needed to send the ISOBMFFMoov-based MPD. Only 3 frames are required as shown in Figure 3.20: one frame for the HEADERS frame and two for the DATA frames.

As we can also see, the HTTP/2 push method leads to more bytes downloaded than when not using the HTTP/2 server push mechanism. This is due to the additional PUSH_PROMISE frames. Both methods download IS video and audio files. Hence, they use the same number as well as the same frame types (i.e. HEADERS and DATA) for each request/response message. The other frame types such as SETTINGS and WINDOW_UPDATE are also used by both methods.

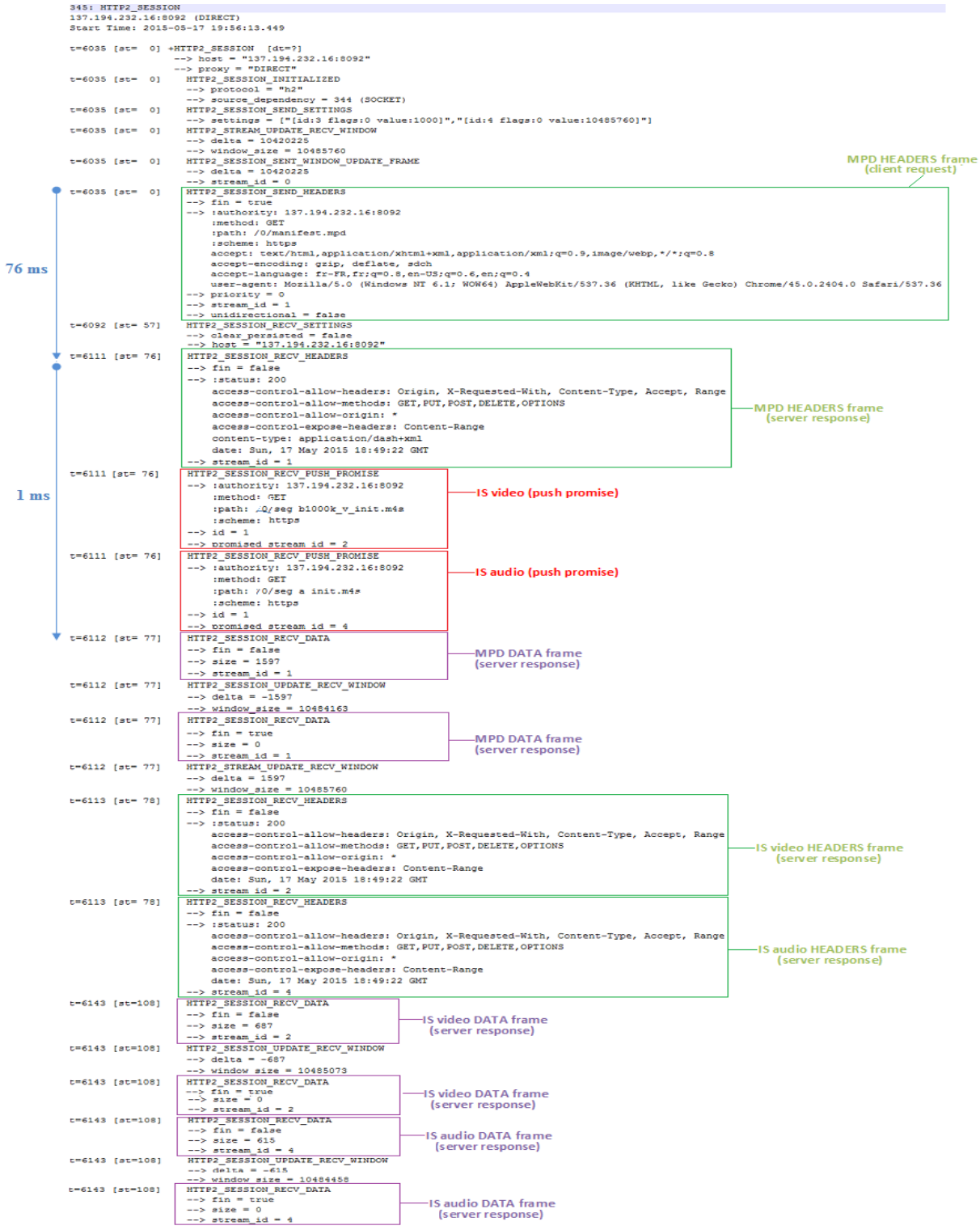


Figure 3.19 – Chrome Net Internals log for the persistent TCP connection without pipelining method when using the HTTP/2 server push mechanism to download IS video and audio.

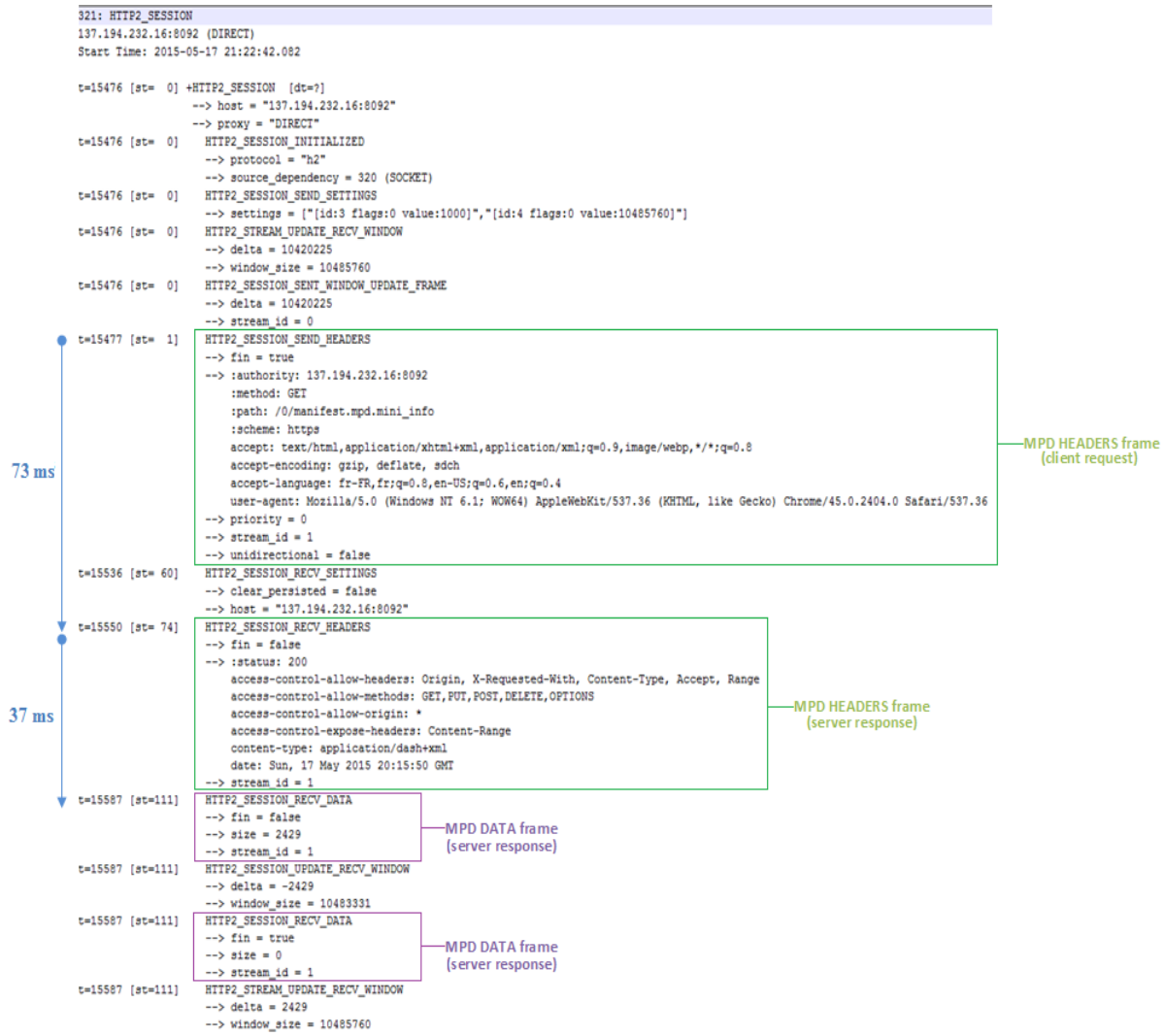


Figure 3.20 – Chrome Net Internals log for the ISOBMFFMoov embedding method over HTTP/2.

We present the results of Table 3.9 in Figure 3.21. The DASHIF sequences are sorted according to the total download size measured when the IS and MPD are delivered separately over a single persistent TCP connection.

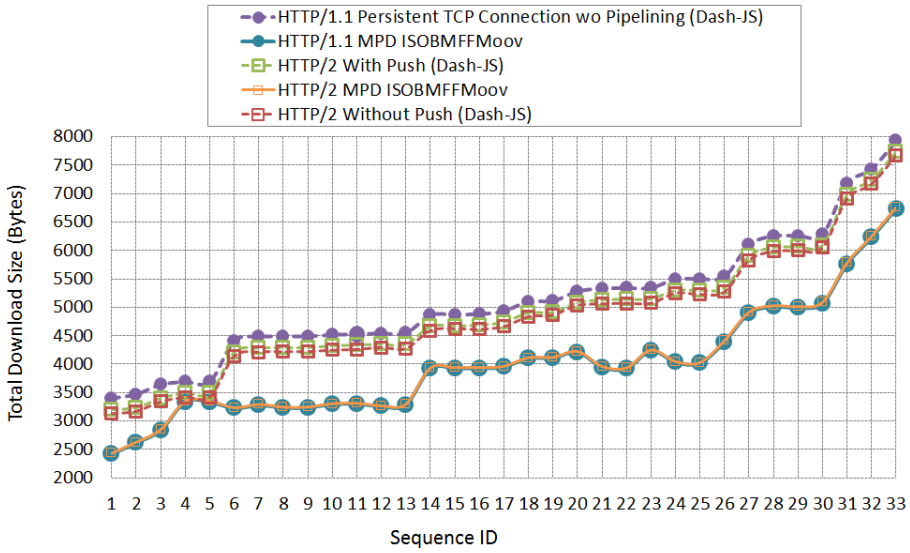


Figure 3.21 – Total download size of 33 sequences for each method over HTTP/1.1 and HTTP/2 using a nodeJS-based server.

3.4.3.2 Bootstrap Delay

In this section, we measured the bootstrap delay when using four different methods under two different network types.

DSL Network

We compared, in terms of bootstrap delay, our ISOBMFFMoov-based approach first to the persistent TCP connection without pipelining approach over HTTP/1.1, and then to the HTTP/2 server push approach.
For the persistent strategy, we measured using Google Chrome Network Panel the elapsed time between when the Dash-JS player establishes a TCP connection to request the MPD from the web server and when it receives the last byte of the last IS. The MPD processing time by Dash-JS as reported by Chrome is deduced in this measurement. Additionally, we also measured the time to download the ISOBMFFMoov-based MPD.

Figure 3.22 displays a visual waterfall of all network requests made by the Dash-JS player over HTTP/1.1 for a given sequence using the two approaches: persistent TCP connection without pipelining and ISOBMFFMoov-based approach. The former approach requires three requests to download separately the MPD, IS video, and IS audio. Only one request is required to fetch the ISOBMFFMoov-based MPD. For each resource, it shows the body size which is the compressed content size sent. It shows also the time it took to load each resource. In the single persistent TCP connection as depicted in Figure 3.22(a), it took 114 ms for MPD, 52.9 ms for IS video, 52.4 for IS audio, and all adding up to 219.4 ms in total which

represents the bootstrap delay. However, Figure 3.22(b) shows that MPD ISOBMFFMoov embedding spent only 113.7 ms. There is a difference of two RTTs between both approaches.

http://127.0.0.1/dash.js-1.3.0/samples/dash-if-reference-player/		
GET manifest.mpd	1.6 KB	114.04 ms
GET seg_b1000k_v_init.m4s	699 B	52.9 ms
GET seg_a_init.m4s	627 B	52.4 ms
3 Requests	**2.9 KB**	**219.34 ms**

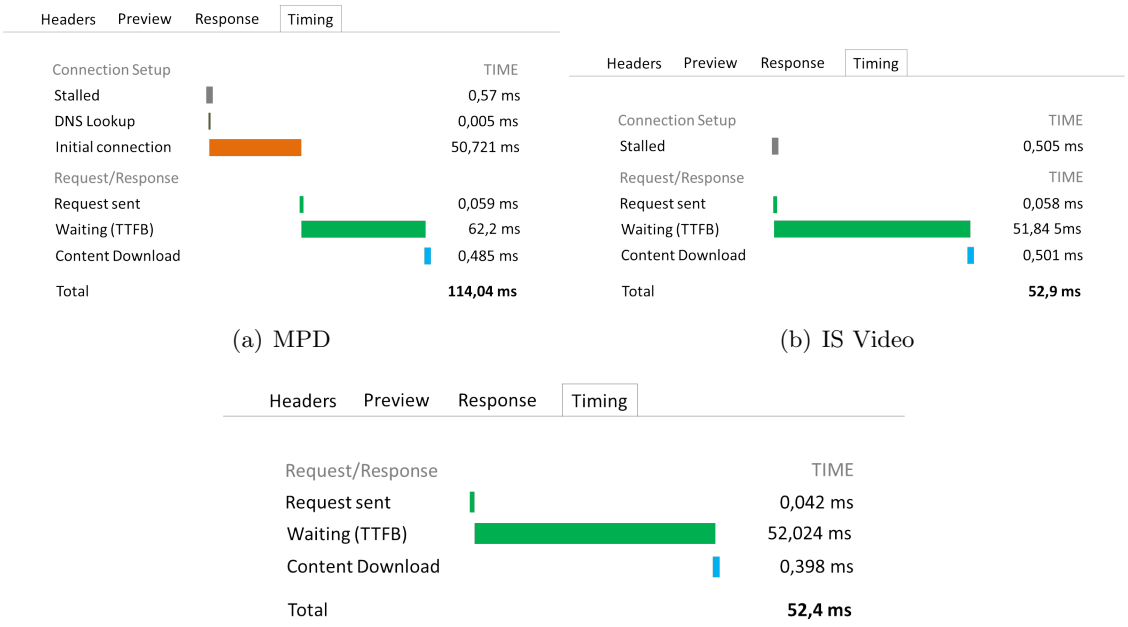
(a) Persistent TCP connection without pipelining strategy.

http://127.0.0.1/dash.js-1.3.0/samples/dash-if-reference-player/		
GET manifest.mpd.mini_info	2.4 KB	113.7 ms
1 Requests	**2.4 KB**	**113.7 ms**

(b) ISOBMFFMoov strategy.

Figure 3.22 – Waterfall based on Dash-JS player over HTTP/1.1.

As illustrated in Figures 3.23 and 3.24, each resource loading time is represented as a set of bars that specify the time spent on the various network stages: stalling (time the request spent waiting before it could be sent), DNS lookup (time spent performing the DNS lookup), initial connection (time it took to establish a TCP connection), request sent (time spent issuing the network request), waiting (known as the Time To First Byte (TTFB) which is the time spent waiting for the initial response), content download (time spent receiving the response data), and SSL (if required time spent completing a SSL handshake) to download the resource. Each bar color indicates its specific phase.
Close analysis of Figure 3.23(a) shows that the download time of the MPD, indicated in blue, is a small fraction of the total latency of each connection. In addition to the TCP connection handshake delay indicated by the orange bar, there is a lot of network latency while waiting to receive the first byte of each response as marked by the green bar. Same analysis concerning Figure 3.23(b) and 3.23(c) except that no TCP handshake time is needed as they are delivered over a persistent TCP connection.



(a) MPD

(b) IS Video

(c) IS Audio

Figure 3.23 – Network Timing of MPD, IS video, and IS audio using a persistent TCP connection without pipelining over HTTP/1.1.

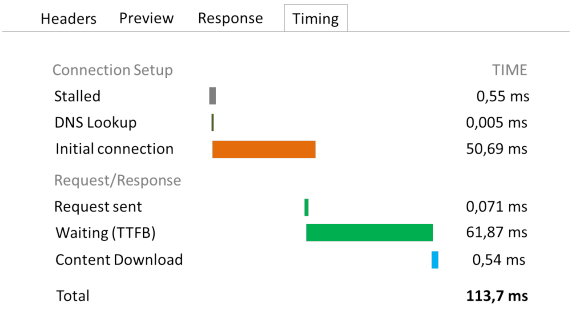


Figure 3.24 – Network Timing of ISOBMFFMoov-based MPD over HTTP/1.1.

Figure A.4 shows these measurements when the downloads were made over an Ethernet network, using HTTP/1.1, with varying TCP's init_cwnd (3 and 10 TCP segments). The DASHIF sequences are sorted according to the total download size measured in the previous experiment when the IS and MPD are delivered separately over a single persistent TCP connection.

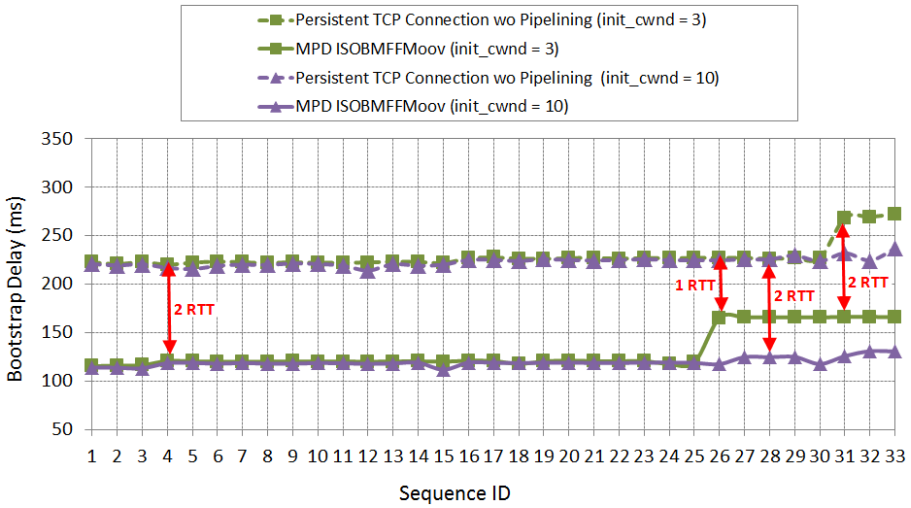


Figure 3.25 – Bootstrap delay measured for the ISOBMFFMoov-based approach and persistent TCP connection without pipelining approach over HTTP/1.1.

These results show first that the bootstrap delay using our approach is decreased by around 2 RTT (100 ms) compared to the persistent approach used by Dash-JS player. These 2 RTT are due to the two request-response cycles that the player needs to retrieve the video and audio IS to start the initial playback. We notice also that the bootstrap delay when init_cwnd is set to 3 TCP segments seems stable for almost all sequences using our approach. From the 26th to the 33th sequence, the delay is increased by 1 RTT (50 ms). This is explained by the fact that the number of TCP segments allowed in the init_cwnd (3 TCP segments about 4380 bytes) is not sufficient to fit the entire MPD with embedded IS information. In this case, the TCP slow start algorithm requires waiting for acknowledgements to arrive before new data is sent which induces an additional RTT. The size of these eight sequences is increased because they are packaged using the "avc1" mode and therefore, the embedded base64 encoded "stsd" box in the MPD is larger. We observe the same behavior for the persistent approach except that the bootstrap delay is increased by 1 RTT (50 ms) from the 30th to the 33th sequence. This is due to the MPD size that exceeds the init_cwnd size for these sequences.

When increasing the init_cwnd to 10 TCP segments (approximately 14 600 bytes), the bootstrap time is stable for all sequences for both approaches. This is because the size of the downloaded resources is less than the init_cwnd size.

Finally, it should be noted that our approach is more efficient when the number of IS chosen by the client (N_{IS}^{C}) is closer to N_{IS}, i.e. when the MPD contains several adaptation sets with different content types. We presented here a worst case, with only 2 adaptation sets.

Beside the experiment measurements, we also computed the theoretical download time for those resources for both approaches according to the formulas shown in Table 3.2. In Figure 3.26, we compared the theoretical bootstrap delay with the real one for both approaches and we confirmed that they are approximately identical which proved the reliability of our experiments.

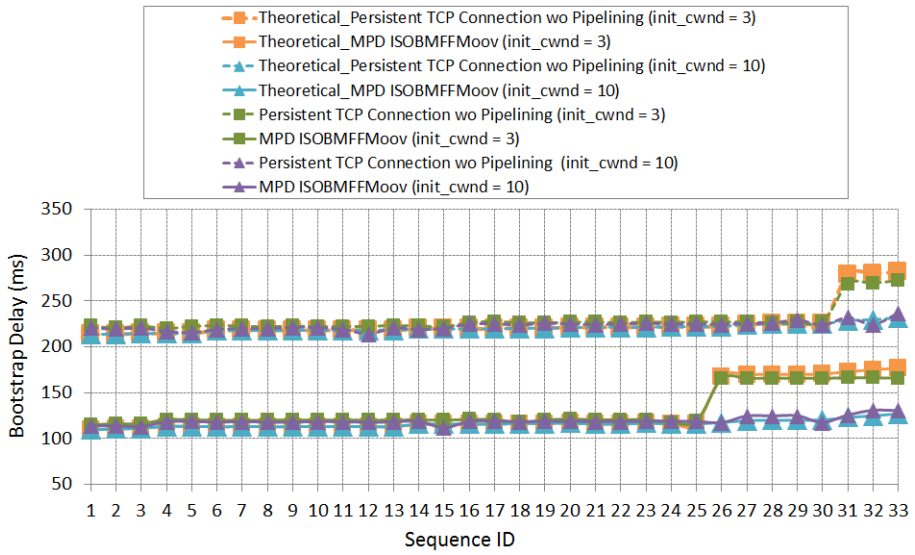


Figure 3.26 – Theoretical bootstrap delay Vs real bootstrap delay for the ISOBMFFMoov-based approach and persistent TCP connection without pipelining approach over HTTP/1.1.

Finally, we measured the bootstrap delay using our ISOBMFFMoov-based approach and the server push method over HTTP/2. The server push is enabled on the web server and on the client. The server is aware of all IS (i.e. IS video and IS audio) to push for a given MPD. The results in Figure A.5 show that both methods take 3 RTT: one RTT for the TCP handshake, one RTT for the TLS handshake, and one RTT for the only MPD request.

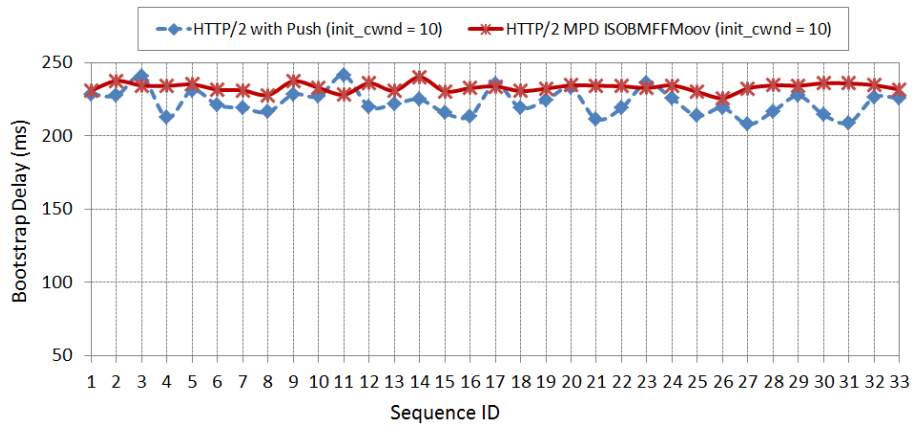


Figure 3.27 – Bootstrap delay measured for the ISOBMFFMoov-based approach and persistent TCP connection without pipelining approach using a server push over HTTP/2.

Both methods provide similar results with a slight advantage for the HTTP/2 push method despite the fact that the download size of our approach is smaller (see Table 3.9). After deep inspection with Net Internals logs, we suspect a problem with the NodeJS web server that sends the ISOBMFFMoov-based MPD late compared to the push method.

When looking more closely to Figure 3.20, we note that the client sends a HEADERS frame for the HTTP request to download the ISOBMFFMoov-based MPD at $t = 1ms$ and after 73 ms of delay it receives the HTTP response header in the HEADERS frame, which is more than one RTT. Following that, the web server sends the first DATA frame carrying MPD data after 37 ms. However, in the push method in Figure 3.19, we can see that after 76 ms the HEADERS frame of the MPD response is sent to the client which is approximately close to our approach. Interestingly, the first DATA frame for the MPD response is not delayed and it is sent immediately as shown in Figure 3.19. This can explain the fact that the bootstrap delay in the push method is slightly smaller than when using our method. This needs to be confirmed with another implementation.

Mobile Network

We now compare, in terms of bootstrap delay, our ISOBMFFMoov-based approach to the persistent TCP connection without pipelining strategy in a mobile 3G-based network. As in a DSL network, we used Google Chrome Network Panel to measure the bootstrap delay for both approaches. For each sequence, we repeated the measurement 10 times and then we calculated the average value. Figure A.6 shows the average of these measurements. Note that the sequences are sorted according to the total download size measured in the previous experiment when the IS and MPD are delivered separately over a single persistent

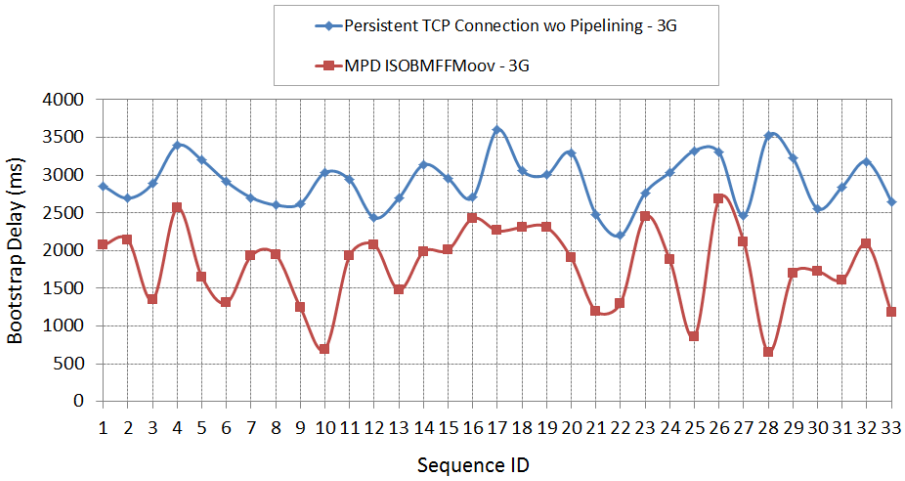


Figure 3.28 – Average bootstrap delay measured for the ISOBMFFMoov-based approach and persistent TCP connection without pipelining approach using a 3G mobile network.

TCP connection. These results show many variations in the download delay that varies between 3 s and 600 ms. During the measurements test, we notice that the delay varies even between the 10 measurements for a given sequence. Interestingly, we can see that the bootstrap delay using our approach is decreased by an average of more than 1 s compared to the persistent approach.

3.4.3.3 Startup Delay

We now evaluate the bootstrap delay measured over HTTP/1.1 compared to the startup delay when using our ISOBMFFMoov-based approach and the persistent TCP connection without pipelining. For that, we choose two sequences which include representations with high and low bandwidth (e.g. 500 Kbps and 8 Mbps) in a video adaptation set.

We changed the buffering delay (D_{buff}) from 40 ms (one frame duration) to 30 s. For each buffering delay, we calculated the download delay (D_{down}) as expressed in Equation 3.10, using a network capacity (C) of 2 Mbps and the bitrate (B) of the video representation.

$$D_{down} = D_{buff} \times B/C. \tag{3.10}$$

We then calculated the startup delay (D_{stup}) that consists of the sum of the download delay and the bootstrap delay. We used the previous measured bootstrap delays for both approaches of the two sequences.

$$D_{stup} = D_{boot} + D_{down} \tag{3.11}$$

Finally, we derived the percentage of bootstrap delay compared to the startup delay as expressed in Equation 3.12 and we presented it in terms of buffering delay in Figure 3.29.

$$D_{boot}(\%) = D_{boot}/D_{stup} \tag{3.12}$$

According to Figure 3.29, we can note that the percentage of the bootstrap delay becomes negligible when the buffering delay increases. The optimization of the bootstrap delay is relevant when the buffering is small, i.e. when using low latency (see Chapter 4). We can see also that the percentage of bootstrap delay is less important for video representations with a high bandwidth than for video representations with a low bandwidth.

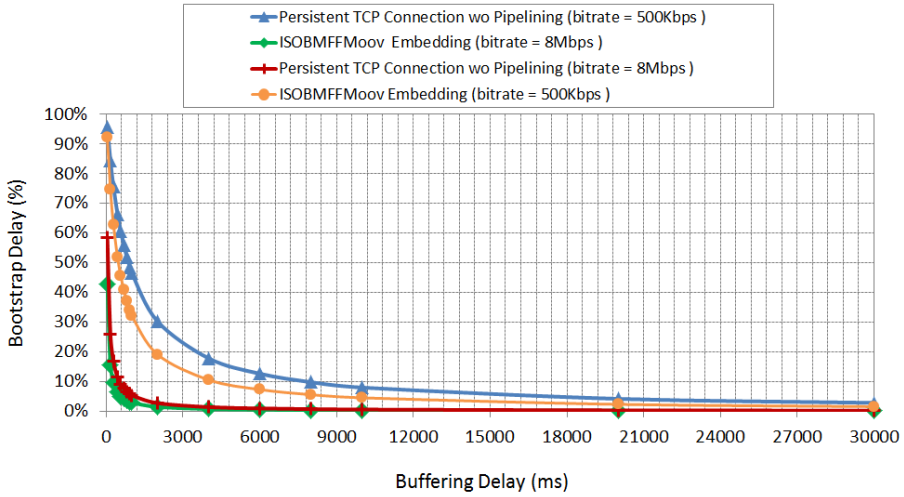


Figure 3.29 – Evaluation of bootstrap delay in terms of buffering delay.

In Figure 3.30, we present the percentage of the bootstrap delay compared to the startup delay for the two representations. We can see that under 600 ms of startup delay it becomes interesting to optimize the bootstrap delay because it represents between 20% to 40% of the startup. We can note that the percentage of bootstrap delay is reduced when we embedded MPD and IS in a single download compared to the approach that downloads the IS and MPD separately.

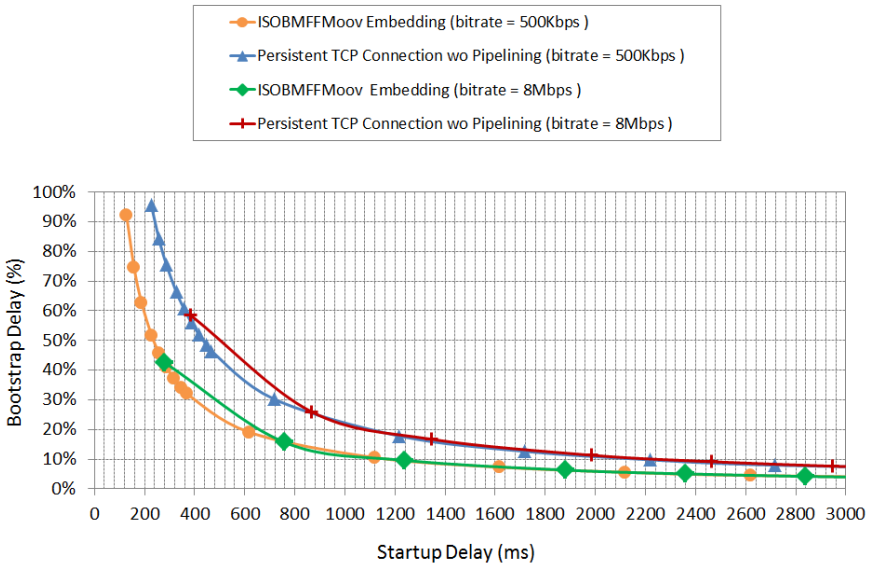


Figure 3.30 – Evaluation of bootstrap delay in terms of startup delay.

To view differently the results of Figure 3.30, we can compare our approach with the persistent method when using the longest startup delay as it is observed in the persistent method. Figure 3.31 presents the percentage of the bootstrap delay compared to the startup delay of the persistent method.

As shown in Chapter 2, [4] reports that a startup delay of 2 s or less does not have a large effect. According to our results, it is worth highlighting that when the startup delay is 2 s or less, the buffering delay is small and the percentage of bootstrap delay compared to the startup delay becomes more important. This case is needed and used in low latency live streaming where a client should use a short buffer length. It is therefore important to reduce the bootstrap delay in low latency live streaming case which is the scope of Chapter 4.

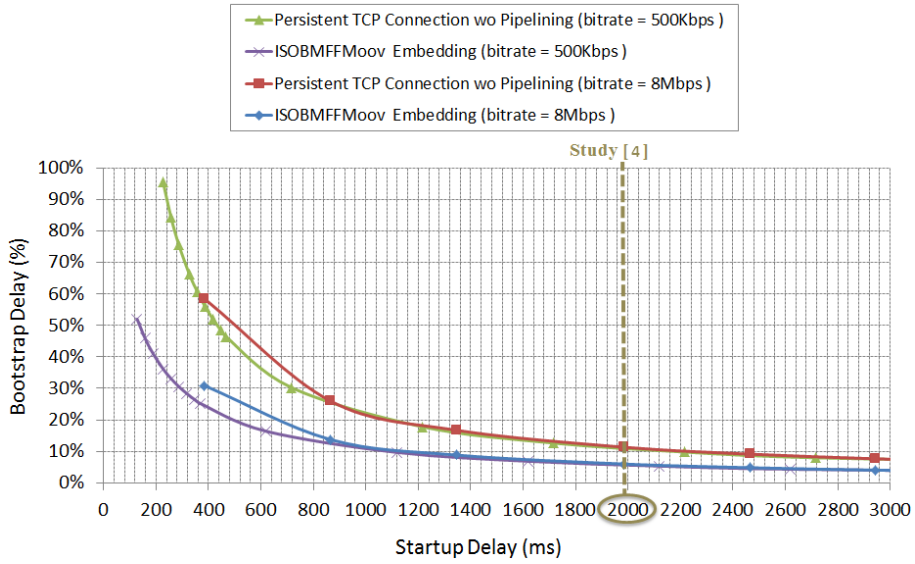


Figure 3.31 – Evaluation of bootstrap delay in terms of startup delay of the persistent method.

3.5 Conclusion

In this chapter, we have reviewed the possible causes of latency in the bootstrap phase of a DASH session. Using the different options offered by the HTTP and TCP layers, we have provided an analytical evaluation of the different DASH client bootstrap strategies in terms of number of TCP connections, number of HTTP requests/responses and the associated bootstrap delay. Then, we have proposed three methods (i.e. Base64 IS Embedding, Multipart Content Embedding, ISOBMFFMoov Embedding) to reduce the bootstrap delay. They are based on a single HTTP request and HTTP response to retrieve the MPD and IS resources.

Base64 IS Embedding and ISOBMFFMoov Embedding were the two approaches that we

have evaluated in our experiments. We have measured and compared the total download size and the bootstrap delay of our proposal to several existing approaches. These measurements were made over two different networks (ADSL and 3G-mobile) and using two versions of the HTTP protocol (HTTP/1.x and HTTP/2 with and without a server push mechanism). We have shown that the total bootstrap download size is reduced by 36% when using our ISOBMFFMoov embedding approach compared to the amount of data being downloaded for the persistent TCP connection without pipelining strategy over HTTP/1.1. It is decreased by an average of 21% in our method over HTTP/2 versus HTTP/2 push method. Regarding the bootstrap delay, we have shown a gain of 2 RTTs in HTTP/1.x and almost no penalty when using HTTP/2 in a DSL network with push.
In a mobile network, we have shown a gain of 1 s or more when using our ISOBMFFMoov embedding approach compared to the persistent TCP connection without pipelining.
Finally, we have presented that reducing the bootstrap delay is critical to reduce the startup delay when the buffering delay is small. We have shown that our method can give up to 50% time saving.

The work presented in this chapter has been resulted in one paper published in the IEEE 17th international workshop of the Multimedia Signal Processing (MMSP) in 2015 [54], in three contributions to standardization in Moving Picture Experts Group (MPEG) standard in 2015, and is included in the Broadcast TV Profile of MPEG-DASH.

Chapter 4

Contributions to Reducing Live DASH Latency

Contents

4.1 Introduction

Live DASH streaming may suffer from significant end-to-end latency which can be in the order of tens of seconds [20]. This latency is defined as the difference between the time when a live event occurs (e.g. when an image is captured) and when it is played to the viewer (e.g. when an image is rendered on the viewer's screen). In VoD service, such latency has no interest because the viewer is watching a video which is pre-recorded and stored on the web server. But it may not be acceptable for live events such as live sports games.

The end-to-end latency of HTTP adaptive live streaming becomes more problematic when comparing it to other live delivery channels. For instance, existing live TV broadcast

systems feature a constant latency (e.g. six seconds). This issue of latency is even more important when someone is watching for example a football match, and may hear his neighbors cheering over a scored goal before he actually sees the goal scored on the screen. In this case, the viewer is most likely to switch to other delivery systems to be close to the live event. As a consequence, the end-to-end latency has a relevant impact on the overall viewer's QoE and engagement as shown in Chapter 2.

While packet-based streaming solutions, e.g. using RTP, can achieve latency under one frame, HTTP adaptive streaming solutions such as DASH are not used today for very low latency streaming. The major reason for that is that HTTP adaptive streaming relies on a segmentation process, whereby encoded media frames are aggregated into segments with pre-defined (often fixed and long) durations. Each video segment is used as a download and a switching unit in HTTP requests and responses. In typical scenarios, the video content cannot be delivered until the video segment it belongs to is fully generated. Therefore, in the live streaming case, the live latency is at least one segment duration even without considering any encoding/decoding, buffering and network delays that are introduced in Chapter 2. In a typical configuration, the segment duration is in the order of ten seconds, leading to a minimal latency of tens of seconds, which is not acceptable in many live streaming scenarios.

Despite the benefits of HTTP adaptive streaming explained in Chapter 2, DASH is not initially adapted for low latency. In this chapter, we propose and demonstrate how to achieve very low latency, i.e. latency similar to the one achievable with RTP systems, but using DASH. A straightforward solution to lower the latency is to reduce the duration of the segments. However, the reduced segment duration may greatly impact the performance of HTTP servers and caches. For example, it would cause an explosion in the number of HTTP requests and responses since each segment requires an HTTP request/response. Additionally, it would increase the number of GOPs as each segment in HTTP adaptive streaming solutions shall start with a RAP (i.e. IDR in AVC and HEVC); reducing the segment duration implies reducing the GOP size, therefore increasing the required bitrate to achieve the same video quality. To solve the latency issue, we introduce a separation between the delivery unit (i.e. what can be sent) and the download unit (i.e. what can be requested). We rely in our approach on specific parts of a media segment called movie fragments as new delivery units in HTTP responses while we keep the media segment as the basic download and switching unit in HTTP requests.

This chapter is organized as follow. Section 4.2 focuses on the basic concepts of live DASH streaming. Section 4.3 describes our proposed approach to reduce the end-to-end latency. Section 4.4 describes some experiments made to validate the approach and Section 4.5 concludes the chapter.

4.2 Basic Live DASH Latency

A live DASH streaming service provides a live stream as a sequence of media segments, which are continuously downloaded by the client. In addition to the basic delays of a streaming chain already introduced in Chapter 2, the authors of [20] have identified some new delays specific to DASH, i.e. segmentation delay and asynchronous fetch delay, that we explain below:

- **Segmentation delay** is the time needed by a segmenter (also described as a packager) to create a segment in ISOBMFF format. It depends directly on the media segment duration. The segmentation process will be described later in detail.
- **Asynchronous fetch delay** is due to the fact that a media segment is not requested immediately once it is completely available on a web server. There is an uncertainty about when a new media segment is available and when it should be requested. This is due to the possible mismatch between the clock times used by the client and the server.
  Additionally, to be able to make a request, a client needs to know the media segment URL and when it is available. In some systems, each time the segmenter completes a new media segment, the MPD is updated. In this case, a client is required to regularly fetch the MPD to get the newest media segment and then compute its availability, which can introduce an additional delay. In other systems, the media segment URL is known in advance and its availability can be computed, thereby avoiding the additional MPD update delay.

In order to minimize the end-to-end latency, it is important to understand how media segments are generated, how a client can determine the latest available media segment, and when it is available in a live DASH streaming session. In the following, we respond to all these issues.

4.2.1 Segmenting Live Content

In live DASH streaming, a sequence of media segments are created on-the-fly from a continuous live stream. The live video segmenter aggregates and packages a certain amount of incoming encoded frames to produce a compliant ISOBMFF segment in such a way that the time interval between the first and the last frames in the segment equals segment duration (d_s).
Each generated ISOBMFF media segment typically consists of a segment type box (*"styp"*) which acts as a file identity, an optional segment index box (*"sidx"*) that provides the list of RAPs in that segment (time and position), and a movie fragment.
A movie fragment consists of a movie fragment box (*"moof"*) that holds the fragment's

metadata (e.g. timestamp, duration, size, etc) and a media data box (*"mdat"*) that contains all the media samples. HTTP adaptive streaming solutions use a self-contained fragment, i.e. the *"moof"* box references only data contained in the *"mdat"* box. A *"moof"* box contains a movie fragment header box (*"mfhd"*) that holds the fragment sequence number, and then at most one track fragment box (*"traf"*) for each media type. When the presentation contains a single video track and a single audio track, each *"moof"* may contain two *"traf"*, one for the video and one for the audio. The *"traf"* contains in turn one or more track run boxes (*"trun"*) that documents a contiguous set of samples of that traf (e.g. sample size, duration and RAP). The track fragment header box (*"tfhd"*) sets up information and defaults used for those runs of samples. The media samples of each *"traf"*, i.e. encoded media frames, are enclosed in the *"mdat"* box. Figure 4.1 presents the structure of a media segment with only one media type.

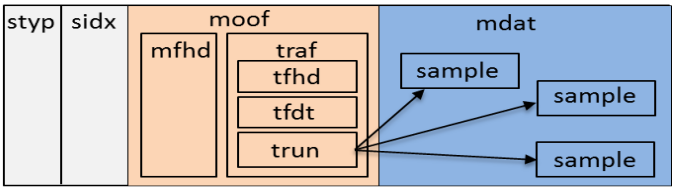


Figure 4.1 – Structure of an ISOBMFF media segment with one media type.

Figure 4.2 depicts the regular segmentation process that all HTTP adaptive streaming solutions use. The segmenter receives the live video stream from the encoder at t_0. It then constructs a sequence of media segments following the ISOBMFF media segment structure of Figure 4.1. All HTTP adaptive streaming solutions force storing the *"moof"* box before the *"mdat"* box in a media segment. This implies that the *"moof"* box should be sent to the web server before the *"mdat"* box. However, the segmenter cannot construct the *"moof"* box until all the media samples are received, i.e. until the "mdat" box is completely constructed. Hence, the segmenter cannot output and deliver a media segment to the web server before the entire segment (i.e. "moof" and "mdat" boxes) is ready. Thus, the segmentation process

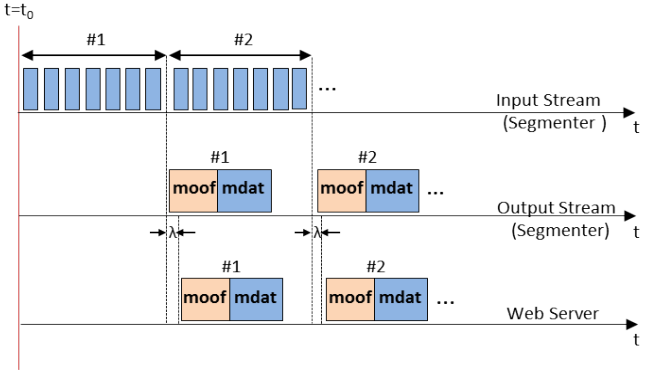


Figure 4.2 – Regular segmentation of a live stream into a sequence of media segments following the basic media segment structure of Figure 4.1 (i.e. "moof" box followed by "mdat" box).

introduces a delay of at least one segment duration to the end-to-end latency.

An alternative approach to reduce this latency even further could be to reverse the order of ”moof” and ”mdat” boxes, but it opens compatibility issues with current systems that we did not tackle, but that could be investigated in future work.

4.2.2 Fetching Live Edge

In case of live session, a DASH client is required to fetch a media segment, possibly the latest, at the right time based on the MPD information. The MPD can describe all available and not-yet available media segments. It can provide in advance all media segments URLs (e.g. using a segment template or list addressing schemes). In this case, the client can determine the latest media segment URL without updating the MPD. In Apple's HLS, whenever a new media segment is ready, the manifest is updated to include the newest segment URL and to remove the oldest one [40].
As opposed to other HTTP adaptive streaming solutions, DASH usually expects that both servers and clients are synchronized on a common clock (the UTC clock). This approach enables clients to find the current live point and to make only the requests for segments at precise times.

As illustrated in Figure 4.3, we present the different steps that a DASH client has to execute to determine the most recent media segment and when it is available on the web server.

1. When the client connects to the live stream, it first fetches the MPD from the web

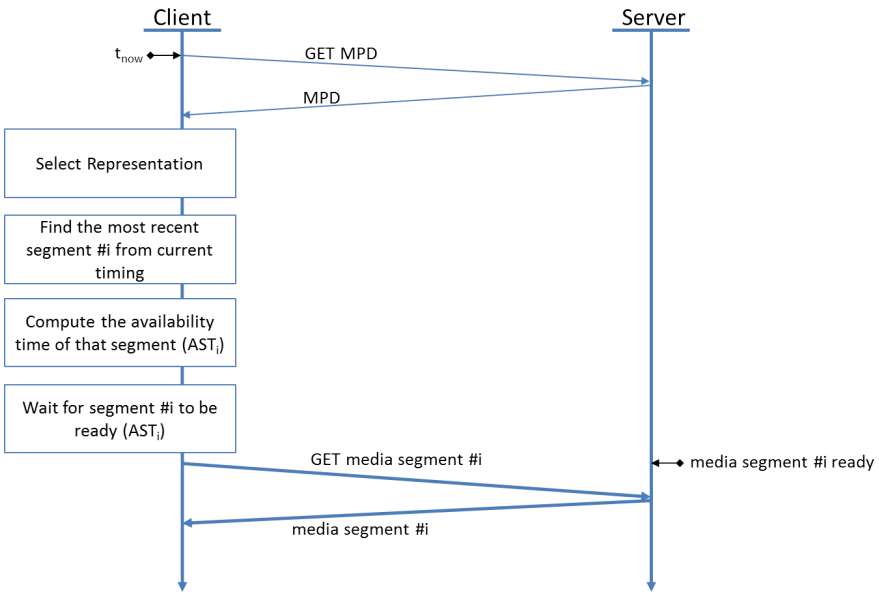


Figure 4.3 – Fetching the live edge.

server. The MPD may be generated on-the-fly, or may have been generated before.

2. In addition to the description of media representations and segments' URLs information, the MPD for live sessions should contain additional key fields such as the *type* field set to "dynamic" as well as the *availabilityStartTime* field. The "dynamic" type indicates that the MPD may be updated. The *availabilityStartTime* field is a reference UTC time used for the computation of the availability of media segments. It usually specifies the start time of a live streaming session.

3. The client then selects the suitable representation based on the described quality/bandwidth to start requesting the associated media segments. When the MPD is dynamic, the client must carefully determine the latest available media segment. For that, it uses the @*availabilityStartTime* attribute value (AST_{MPD}) indicated in the MPD and the current time ($t = t_{now}$), i.e. the current UTC wallclock of client. The number of the media segment being produced at $t = t_{now}$ in the period can be computed as:

$$i = \left\lfloor \frac{t_{now} - (AST_{MPD} + periodStart)}{d_s} \right\rfloor + startNumber \tag{4.1}$$

where periodStart is the start time of the period to play and is equal to the value of @*start* attribute specified in the MPD, d_s is the media segment duration, and startNumber is the number of the first media segment in the period. All these parameters values are present in the MPD.

4. The client should determine precisely the availability start time of the computed media segment "i", i.e. when segment "i" is fully ready on the web server, to make the necessary request at the right time. The availability start time (AST_i) of the media segment "i" is expressed as:

$$AST_i = AST_{MPD} + periodStart + d_s \times (1 + i - startNumber) \tag{4.2}$$

5. Once the client determines the latest media segment number ("i") to request and its availability time (AST_i), it waits for the segment to be ready and makes the request. The server responds with the entire requested segment.

4.2.3 Progressive File Delivery over HTTP

HTTP adaptive streaming solutions using the regular HTTP/1.1 transfer can only deliver files once completed. This means that a web server cannot deliver the data content until the file it belongs to is fully produced which can add a delay of at least one file duration.

HTTP/1.1 "Chunked-Transfer Encoding" mechanism [55] enables the web server to start sending available data parts of a file before the file is completely generated. It modifies the body of the HTTP response in order to transfer it as a series of chunks, followed by a final chunk of length zero, followed by an optional trailer (that should be treated like headers, as if they were at the top of the response), and a blank line. Each chunk consists in two parts:

- a line with the size of the chunk data, in hexadecimal, possibly followed by extra parameters, and ending with CRLF.
- the data itself, followed by CRLF.

This allows dynamically produced content to be transferred along with the information necessary for the client to verify that it has received the full response as well as to be cached. Figure 4.4 depicts an example of HTTP/1.1 chunked response. The Chunked Transfer-Encoding header must appear in the HTTP response headers at the place of Content-Length header. All HTTP/1.1 clients must be able to receive and treat chunked responses.

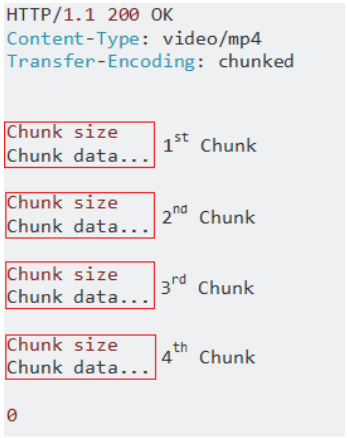


Figure 4.4 – Example of an HTTP/1.1 chunked response.

Obviously, if a single movie fragment (i.e. "moof" and "mdat" boxes regardeless of their order) is used in one media segment, the Chunked-Transfer Encoding mechanism will not be beneficial at all. To benefit from this feature, we propose in the next section our new approach based on making the segmentation process progressive without changing the encoding process.

4.3 Low Latency Live DASH Proposal

In this section, we present our new approach to reduce the end-to-end latency, specifically the segmentation delay in live DASH streaming. It consists in using a finer granularity

unit for HTTP delivery than the media segment while maintaining the media segment as a download unit in HTTP requests. For that, we use a new internal media segment organization as it is depicted in Figure A.7.

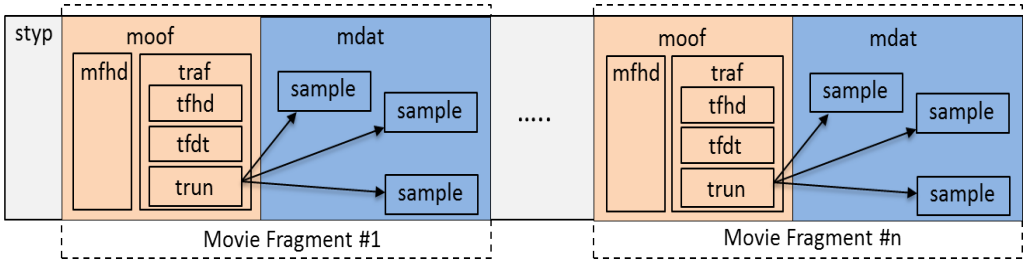


Figure 4.5 – Structure of an ISOBMFF media segment with multiple movie fragments.

Each media segment is divided into multiple small parts in such a way that they are independently parsable. These parts are called movie fragments. Based on the Chunked-Transfer Encoding mechanism, the web server can push available data parts, i.e. those movie fragments, to the client before the media segment is fully ready. But the request has to arrive before a typical DASH request, before the end of the segment.

To address this problem, we modify manually in our first experiments the availability start time of a media segment through the *availabilityStartTime* attribute in the MPD. We set its value to the time at which one or more fragments are available to force a client to send a request earlier. This strategy works very well and allows reducing the segmentation delay to the duration of fragments. However, the main problem of this strategy is that the shifted AST value is global in the MPD, i.e. it is applied to all media contents and distribution networks described in the MPD. Additionally, a client may not be optimized for chunk-transfer processing of DASH segments, and using chunk transfer may alter its bandwidth estimation process because the download time is close to the segment duration whatever the bandwidth.

To overcome this problem, we proposed the introduction of a new attribute named *availabilityTimeOffset* (ATO) at the representation level in the MPD. The ATO attribute indicates the difference between the availability start time of the segment and the UTC time at which the server can start delivering data for this segment, e.g. using HTTP/1.1 chunks. Typically, this latter time corresponds to the time at which one or more fragments are available. This is a fundamental change: with the presence of this attribute, a client is now aware that a part of the segment is available earlier than the segment. The client is also capable of making the necessary request for the current segment at a time that will not imply waiting or that will not trigger a HTTP 404 response, although the segment is not fully produced. If the ATO is chosen to match the time at which the first fragment is fully produced, the packaging latency can be reduced to the duration of a fragment.

The availabilityTimeOffset (ATO) of the media segment "i" is expressed in Equation A.1:

$$ATO_i = AST_i - (AST_i - d_s + d_c) = d_s - d_c \tag{4.3}$$

Figure A.8 shows the relationship between the availabilityStartTime of segment "i" (AST_i), availabilityTimeOffset (ATO_i), segment duration (d_s) and fragment duration (d_c).

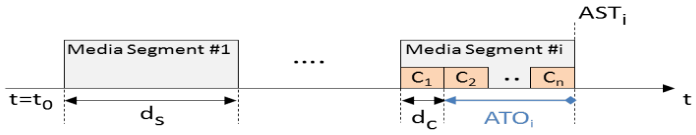


Figure 4.6 – Determination of the availability time of a media fragment in DASH.

In the following, we present the updated client procedure enabling issuing an HTTP request for the most recent media segment once one fragment or more are available on the web server. The web server pushes the newly available fragments as HTTP/1.1 chunks without waiting for the completion of the segment.

1. Step 1, 2 and 3 are similar to those presented in Section 4.2.2. The only difference is that we indicate in the MPD when one fragment or more are available on the web server through the *availabilityTimeOffset* (ATO) attribute.

2. After computing the latest media segment number "i" and using the ATO value, the client waits for the movie fragments of that segment to be ready and makes the request. This waiting time is at most one fragment duration because the live media is encoded in real-time.

3. Each media segment contains $\#n$ movie fragments. The server is able to send out the fragments earlier, at best as soon as it has been completely generated, using HTTP/1.1 chunks. It keeps sending the remaining chunks of segment once they are ready until the end of segment is detected.

4. The client is able to consume any received partial response (i.e chunks) before the reception of all parts. Especially, each chunk (i.e. "moof" and "mdat" boxes) can be parsed and queued for playback even if the complete segment is not yet received.

4.4 Evaluation

4.4.1 Design and Implementation

To experiment with the proposed approach, we have designed and implemented a complete DASH streaming system based on three main functions: content preparation, content distribution, and content decoding and display, as depicted in Figure 4.7 and detailed below:

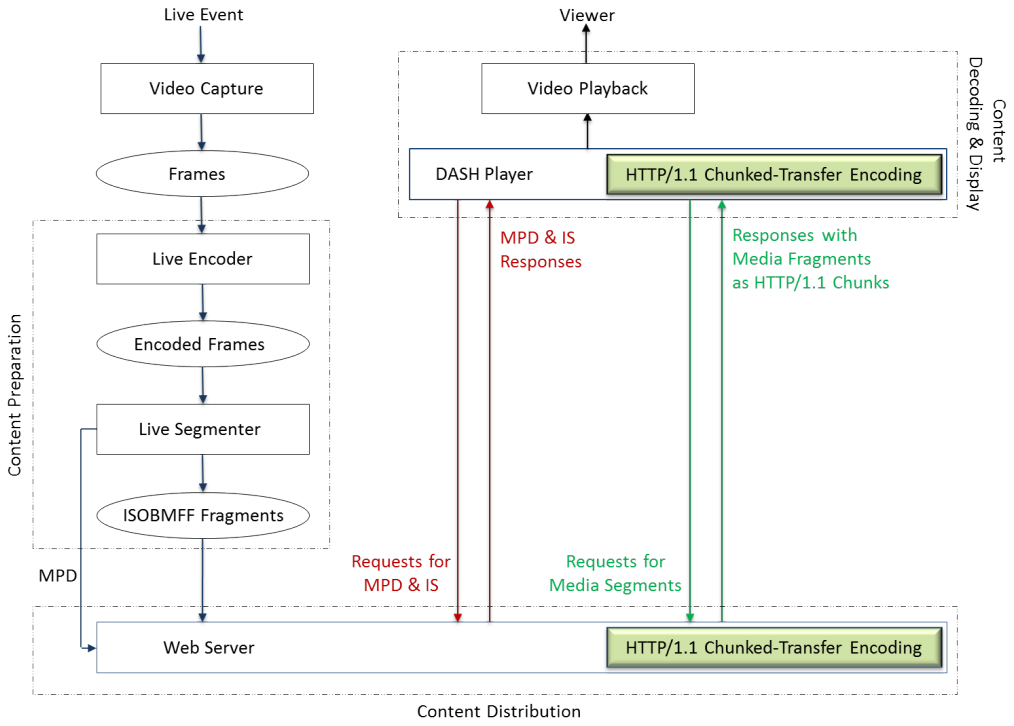


Figure 4.7 – Architecture of a low latency live DASH streaming system.

- Content Preparation

  The content preparation part of the system is in charge of three tasks: encoding the live video stream in real-time, fragmenting and segmenting the video into small fragments and segments according to the ISOBMFF format as depicted in Figure A.7, and generating the DASH MPD. The live video encoder encodes the input live video stream into multiple versions (i.e. DASH representations), with different resolutions and bitrates. Each fragment (i.e. *"moof"* and *"mdat"* boxes) is flushed to disk/file in an atomic way, i.e. the *"moof"* and *"mdat"* boxes are stored temporarily in the memory and are then written at once in the file as soon as they are completely constructed. We set the *availabilityTimeOffset* (ATO) attribute value in the MPD to indicate the availability time of the first fragment of each segment.

In our experiments, we used the DashCast live encoder and segmenter tool from the GPAC[1] project. We have used different number of fragments per segments, ranging from 1 fragment carrying one video frame to 1 fragment carrying the whole segment.

- Content Distribution

The ISOBMFF fragments and associated MPD are deployed on a web server. The web server supports HTTP/1.1. It is connected to the client via Ethernet in a Local Area Network (LAN).
In our experiments, we have implemented an ISOBMFF-aware web server based on the NodeJS[2] framework. This web server is responsible for monitoring content changes in media segments during the generation. An event is triggered each time new data is written in the segment file. The web server distributes media segments as it is illustrated in Figure 4.8. The web server starts monitoring a media segment file when that file is requested by a client. When the media segment file is modified, the server starts the parsing to detect new ISOBMFF fragments. If there is any *"moof"* box followed by *"mdat"* box, this denotes the availability of a new fragment whereby the server forms up an HTTP/1.1 chunk using the Chunked-Transfer Encoding mechanism and sends the response to the client. Otherwise, it keeps listening and

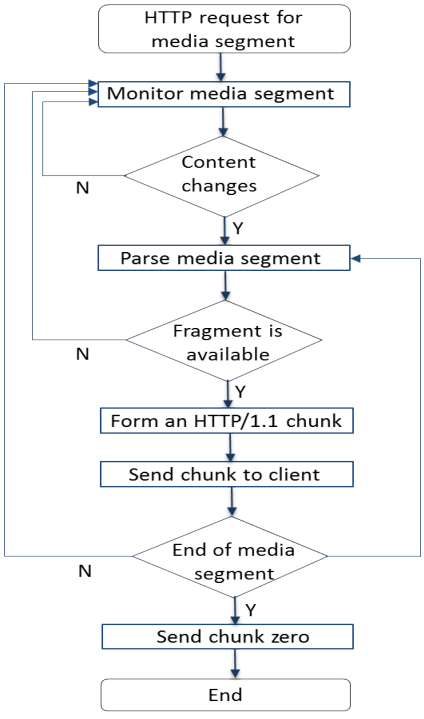


Figure 4.8 – Flowchart of our proposed web server.

[1] https://gpac.io/
[2] http://nodejs.org/

waiting for new fragments. In our system, the server detects the end of a media segment by the presence of a new end of dash segment box (*"eods"*), as shown in Figure 4.9. This enables a server to terminate the data chunk transfer by sending a final chunk of length zero.

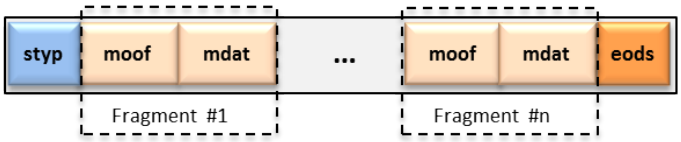


Figure 4.9 – Structure of a media segment with "eods" box.

If a client joins the live stream and requests the current media segment that is partially generated (i.e. a client makes a request late regarding the availability time of the first fragment), our web server is configured in such a way that all produced fragments since the beginning of a media segment are sent to the client in one chunk.

- Content Display

  In our approach, we use the MP4Client player from the GPAC project. It is a compliant DASH client with HTTP/1.1 capability, i.e. it is able to receive chunked responses; and is capable of processing progressively incomplete segments.

4.4.2 Experiments and Results

In order to validate our approach, we have conducted two types of experiments: experiments to measure the overhead introduced by the packaging and transport tools, and experiments to measure the latency of the system. This section details these two parts.

4.4.2.1 Overhead Measurements

For our overhead measurements, we used two video sequences (sports and cartoon), initially compressed with the AVC format, with the characteristics reported in Table 4.1.

Sequence	Bite rate (Kbps)	Frame rate (fps)	Resolution	GOP size	Duration
RedBull	6000	24	1920x1080	15 s	1 min
Big Buck Bunny	9000	29.97	1920x1080	3 s	10 min

Table 4.1 – Input video sequence characteristics.

We used the open source x264[3] encoder with the Constant Rate Factor (CRF) encoding mode to encode these sequences at different resolutions (ranging from QCIF 176x144 to full HD 1920x1080). We targeted different quality levels for each resolution encoding by setting CRF to the following different values: 16, 18, 19, 20, 22, 32, 24, 26, 28, and 30. We started with a low CRF value (e.g. 16) and then we raised it until 30 where the quality becomes really bad.
Additionally, we set the GOP size to be 1 second, corresponding to a typical DASH segment starting with a RAP, i.e. an IDR frame in the AVC coding format. No B frames were used and only 1 reference frame was used for prediction. We used a frame rate of 24 for the Big Buck Bunny sequence and 30 for the RedBull sequence. We kept only the sequences which resulted in a bitrate lower than the initial one. The exact command line for this encoding is provided below:

x264 –sar 1:1 –o output_video.h264 input_video.h264 –ref 1 –crf <x> –bframes 0 –keyint <y> –vf resize:<w>,<h> –b-pyramid none –fps <f> –preset veryslow –tune psnr –psnr

Each quality encoding with a constant CRF value results in a couple of values: Y-PSNR value which reflects the output video quality level and a bitrate value. Table 4.2 represents the output results of the encoding of Big Buck Bunny sequence.

Table 4.2 – Encoding of the Big Buck Bunny sequence at different CRF values for all resolutions.

Resolution	CRF	Bitrate (Kbps)	Y-PSNR (dB)
176x144	16	412.12	47.081
	18	343.08	45.544
	19	312.69	44.784
	20	284.26	44.021
	22	234.02	42.508
	23	211.75	41.761
	24	191.58	41.03
	26	156.07	39.557
	28	126.37	38.113
	30	101.74	36.694
352x288	16	1279.6	47.437
	18	1052.14	46.025
	19	952.97	45.322
	20	861.55	44.622
	22	701.71	43.231
	23	631.95	42.546
	24	569.12	41.871

Continued on next page

[3]https://www.videolan.org/developers/x264.html

Table 4.2 – *Continued from previous page*

Resolution	CRF	Bitrate (Kbps)	Y-PSNR (dB)
	26	459.47	40.505
	28	368.08	39.145
	30	292.26	37.789
640x360	16	2522.37	47.746
	18	2060.22	46.4
	19	1859.82	45.728
	20	1676.42	45.061
	22	1357.09	43.739
	23	1218.52	43.089
	24	1093.59	42.447
	26	878.84	41.157
	28	703.84	39.869
	30	562.1	38.589
960x540	16	4559.88	48.364
	18	3695.03	47.017
	19	3326.55	46.354
	20	2991.21	45.696
	22	2409.96	44.413
	23	2160.59	43.789
	24	1938.27	43.179
	26	1560.4	41.964
	28	1252.64	40.744
	30	1001.52	39.524
704x576	16	3791.54	47.953
	18	3063.98	46.672
	19	2754.33	46.038
	20	2474.08	45.41
	22	1991.09	44.159
	23	1784.99	43.544
	24	1600.35	42.936
	26	1284.27	41.712
	28	1027.05	40.484
	30	818.83	39.257
1280x720	16	6857.14	48.428
	18	5455.84	47.177
	19	4869.94	46.562
	20	4346.87	45.961

Continued on next page

Table 4.2 – *Continued from previous page*

Resolution	CRF	Bitrate (Kbps)	Y-PSNR (dB)
	22	3463.92	44.78
	23	3095.83	44.201
	24	2771.11	43.634
	26	2223.47	42.489
	28	1784.99	41.342
	30	1431.98	40.188
1920x1080	20	7604.31	47.802
	22	6172.14	46.371
	23	5546.75	45.691
	24	4986.47	45.053
	26	4020.28	43.796
	28	3216.31	42.55
	30	2559.99	41.336

Based on those results, we represent in Figure 4.10 the output video quality (Y-PSNR) in terms of the required bitrate for all CRF values and resolutions.

The encoding results of the Red Bull sequence exhibits the same pattern. For all resolutions, it can be seen that lower CRF values would result in better quality (high PSNR value) at the expense of higher bitrates.

The total overhead of our approach can be decomposed into: the overhead introduced by the packaging of encoded frames into ISOBMFF fragments, and the overhead introduced by the download of those fragments as HTTP/1.1 chunks.

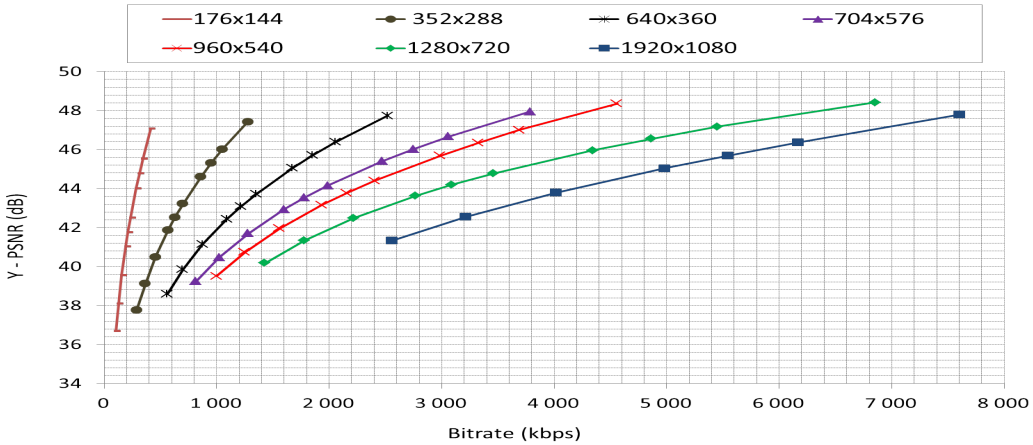


Figure 4.10 – Encoding of the Big Buck Bunny sequence at different CRF values for all resolutions.

- ISOBMFF overhead

  The packaging in ISOBMFF used in our DASH streaming system introduces an overhead which can be decomposed in: an initial overhead due to the packaging of encoded media frames into the structured ISO format; and the additional overhead due to the fragmentation required in DASH.

  The initial overhead does not depend on the bitrate but does depend on the number of samples, i.e. the number of encoded frames per seconds. Our experiments on the Big Buck Bunny and Red Bull sequences show that the additional overhead introduced by the simple storage of encoded sequences in MP4 files compared to the raw AVC sequences is negligible. Typically, across video sequences, resolutions and bitrates, we have an overhead that ranges between 0.0028% for higher resolution with higher bitrate (e.g. 7604 kbps for Full HD sequence encoded at a CRF value of 20) and 0.26% for lower resolution with lowest bitrate (e.g. 100 kbps for QCIF sequence encoded at a CRF value of 30).

  When it comes to fragmented ISOBMFF, we only consider the encoded videos with the default CRF value of 23 for the x264 encoder. We consider a constant segment duration of 1 s containing 24 frames for the Big Buck Bunny sequence and 30 frames for the RedBull sequence. The overhead depends on the number of fragments per segment, i.e. on the number of frames per fragment. Figure A.9 shows the results for the Big Buck Bunny sequence, with the overhead computed with respect to the non-fragmented sequence.

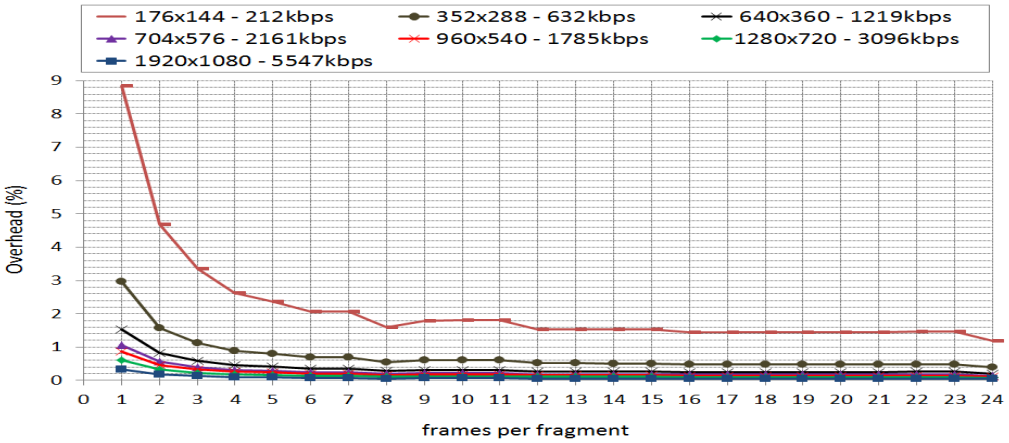


Figure 4.11 – Overhead introduced by the ISOBMFF fragmentation.

We can see that the fragmentation introduces an overhead, which decreases as the number of frames per fragment increases. We can also see that, for a given number of frames per fragment, the fragmentation overhead decreases as the video size increases to higher resolutions (hence higher bitrates). For classical resolutions (SD and more),

we can note that the maximum introduced overhead when fragments carry only one frame is 1% which is less than 2% of overhead when using RTP [56]. For 3 frames per fragment, the overhead is fewer than 4% for all resolutions and bitrates. A high overhead of 9% can be reached up when using one frame per fragment for a QCIF resolution encoded at a bitrate of 212 kbps. This corresponds to what we can observe for audio. For audio, we can have an overhead that ranges approximately between 9% for a bitrate of 200 Kbps and 20% for a bitrate of 64 kbps. This is not a problem for audio because the total fragment size is still very small and can fit in one IP packet.

- HTTP/1.1 overhead

  Finally, the last overhead introduced by the system is in the delivery of content over HTTP/1.1. In typical DASH scenarios, an HTTP request is made for every media segment. The size of the request is highly dependent on the information present in the header (descriptions of the user agent, of the server, list of accept headers, use of byte range, CORS as seen in Chapter 3, etc). In [20], the authors report a typical size of 140 bytes. In [57], the authors report a size of 280 bytes. We can assume an average size of 200 bytes per request. However, in our approach what matters more is the overhead introduced by the Chunked-Transfer Encoding mechanism.

  The chunk size is expressed with 4 bytes. The CRLF is represented with 2 bytes. Hence, each chunk with no extra parameters (extensions) requires 8 bytes. The maximum overhead of an HTTP/1.1 chunked response body containing $\#n + 1$ chunks (i.e. $\#n$ chunks correspond to $\#n$ ISOBMFF fragments and one for the last chunk of length zero) and no trailers can be computed as in Equation 4.4. Each chunk or ISOBMFF fragment contains between 1 frame up to the full segment. Note that the content length in Equation 4.4 represents the length of the chunked response body which is the sum of all chunks sizes.

$$Overhead_{max} = \frac{8 \times (n+1)}{\text{content length}} \tag{4.4}$$

  We supposed a sequence encoded at 8 Mbps, at 24 fps, and fragmented with one frame per fragment. Following Equation 4.4, the maximum additional overhead represents 0.02% which can be considered negligible.

As a consequence, the total overhead for the delivery of an AVC encoded video sequence, stored in fragmented ISOBMFF, delivered with HTTP/1.1, using 1 chunk per fragment, is mostly the overhead of the fragmentation.

4.4.2.2 Latency measurements

At first, we measured the latency from the encoder output to the decoder input, i.e. the time needed to produce and to send ISOBMFF fragments. Following that, we measured it from the capture to the display.

For our measurements, we have implemented a complete DASH streaming system in accordance to our proposed low latency live DASH streaming architecture shown in Figure 4.7. Our web server is located on the same physical machine as DashCast. DashCast and MP4Client were run on different machines, whose UTC system times where configured to use the same Network Time Protocol (NTP) server (e.g. "ntp.enst.fr") to be synchronized. However, we noticed an important mismatch (e.g. up to one minute) between the times used by both machines. We also used the local NTP server of windows 7 system but we were still having the drift time between client and server. Hence, we had to adjust the system time of both machines regularly in our experiments.

Therefore, we decided not to rely on NTP and used a dedicated NTP-inspired mechanism to synchronize the machines, as follows. Upon sending the MPD to the client, the web server adds an extra HTTP header indicating its UTC system time. When the client receives the MPD from the server, it fetches its own UTC system time, substracts the time read from the HTTP headers and obtains an estimated UTC time difference between the two machines. This difference is assigned to λ. This mechanism is very simple, and omits the delivery time of the MPD. Additionally, it assumes no drift between the servers, so is calculated just once and applied every time the availability start time of a segment is compared to the client system time.

1. Inner-chain latency

   As shown in Figure A.10, we have instrumented DashCast (respectively MP4Client) to log the different UTC times at which:

   - a frame was completely encoded (resp. starting to be decoded).
   - a fragment was fully produced (resp. a chunk was fully received).

   MP4Client was configured to remove all buffering. We set the segment duration (d_s) to 2 s and the fragment duration (d_c) (equal to the chunk duration) to 200 ms to obtain 10 fragments per segment. The following DashCast command was run, grabbing the screen of the computer at a resolution of 800x600 pixels, at 25 frames per second, encoding it according to the configuration given in the dashcast.conf file,

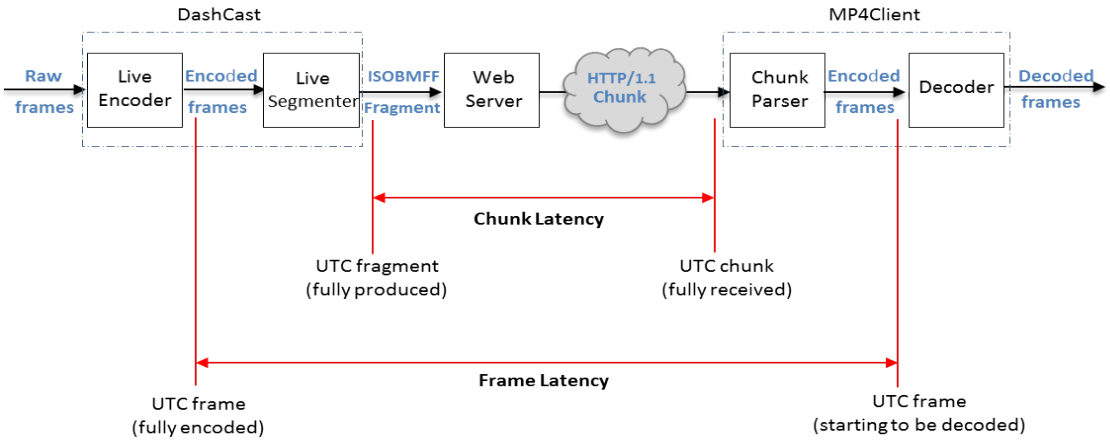


Figure 4.12 – Inner-chain latency measurements for live streaming service.

using "eods" marker and with ATO equal to $d_s - d_c$ (1800 ms). The input video is encoded at 2 Mbps and at a resolution of 1280x720 pixels.

DashCast -vf x11grab -v :0 -seg-dur 2000 -frag-dur 200 –live -conf dashcast.conf -ast-offset -1800 -vres 800x600 -vfr 25 – seg-marker eods

We run the system for 894 seconds during which MP4Client retrieved 447 segments that include 4470 fragments or chunks.

Figure 4.13 shows the frame latency, i.e. the difference between the time at which a frame was encoded and at which it was starting to be decoded as shown in Figure A.10. With 25 frames per second, a fragment of 200 ms contains 5 frames and a segment of 2 s includes 50 frames. On Figure 4.13, we represent the latency of 200 frames of 4 segments (i.e. 40 fragments). In this configuration (5 frames per fragment), the fragmentation overhead is 0.2% as shown previously in Figure A.9.

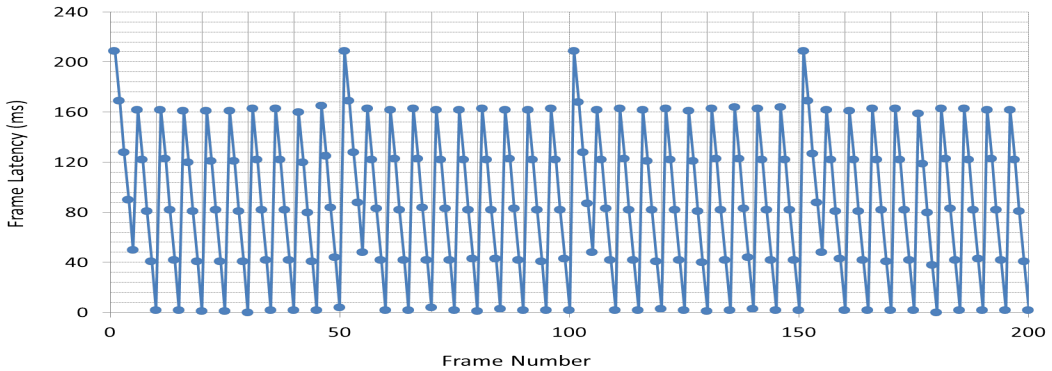


Figure 4.13 – Frame latency of 200 frames of 4 segments (i.e. 40 fragments) (d_s=2s, d_c=200ms, ATO=1800ms).

We can see that the latency of most fragments is around 160 ms. The first frame of each fragment experiences a high latency (160 ms) because it waits for the four remaining frames to construct the fragment. The second output encoded frame is delayed by 120 ms because it waits for the remaining 3 frames of the fragment. The third one incurs a latency of two frames duration (80 ms). The fourth frame waits only 40 ms. The last frame suffers almost no latency.

The waiting time of each encoded frame at the segmenter depends on the availability time of the remaining frames needed to construct the fragment for the encoding. For example, the first frame (f_1) is ready immediately for the encoding. It is then transmitted to the segmenter where it is hold until the last required frame (f_5) for the fragment is generated and available at the segmenter, at most after four frames (160 ms). Hence, the fragment is fully produced and ready to be sent to the web server after 160 ms of delay.

For the first frame of each segment, the maximum latency is around 210 ms instead of 160 ms. This latency is however reduced for the next fragment as it is pushed by the server and no request is made by the client. We decompose this additional delay of 50 ms into two values as follows. The first delay of 10 ms can be explained by the client making the request for the segment too late. This can be due to slight variations in the estimated UTC drift time between the client and server machines, very likely due to the MPD request download time which is not estimated. The second delay of 40 ms is due to the fact that we have indicated through ATO that the first fragment of each segment will be available at 200 ms, whereas the 5 frames constituting the chunk are encoded and packaged in 160 ms. This is due to our proposed model that considers by default the availability time of each frame as the sum of its acquisition time, its duration, and its encoding time (acquisition and encoding delays are negligible in our case). This principle is applied to all media types (e.g. video, audio, and subtitle). For instance in audio, the encoder has to wait for an integral number of samples needed to construct a coding frame before invoking the compression. If the number of samples per coded frame is 1024 for example, the encoding process does not start before receiving 1024 samples (i.e. before $\frac{1024}{\text{sampling rate}}$ seconds). However, for video, a frame is ready for compression as soon as captured, hence is processed by the segmenter roughly at its capture time (assuming encoding time is negligeable).

Hence, the ATO in our proposed model represents the availability time of the fragment, i.e. the availability time of the last frame of the fragment, including its duration. For the video case however, the end of production of the fragment is one frame duration less which explains the delay of 40 ms.

Figure 4.14 shows the histogram of the number of frames per latency value for the total streaming session. As we can see on the histogram there are 5 high bursts corresponding to the latency of each frame in the fragment. The low burst at the right side represents the number of first frames of each segment that experiences 200 ms of latency. The second burst represents the number of first frames of each fragment being delayed most (160 ms), and the last burst at the left side shows the number of fifth frames suffering almost no latency.

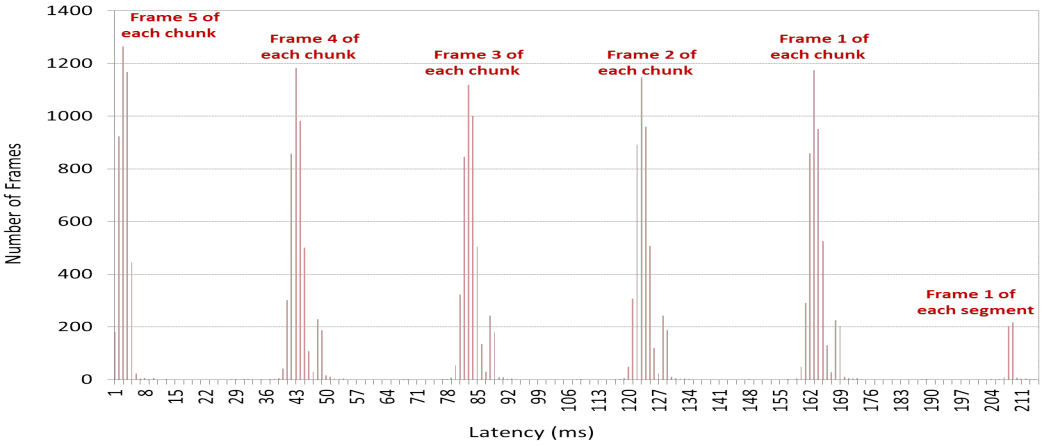


Figure 4.14 – Frame latency histogram.

Figure 4.15 shows the chunk latency, i.e. the difference between the time at which a fragment was fully produced by DashCast and at which it was received as an HTTP chunk by MP4Client (i.e. specifically by the chunk parser as shown in Figure A.10). The figure represents 40 chunks which is equivalent to 4 segments. The other segments exhibit the same pattern. As can be seen on the figure, the chunk latency is of the order of 2-4 ms for every chunk, except for the first chunk of each segment, where it is in the order of 50 ms. As explained previously, this is due to the client

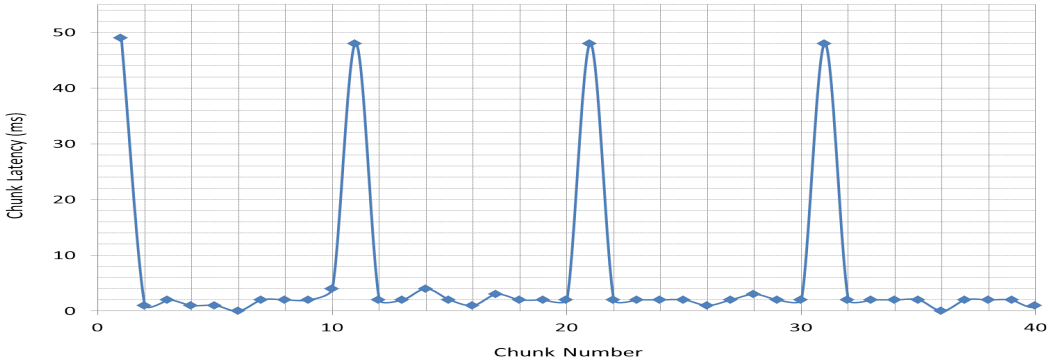


Figure 4.15 – Chunk latency of 40 chunks of 4 segments (d_s=2s, d_c=200ms, ATO=1800ms).

that requests the segment (i.e. the first chunk) approximately 40 ms after the last frame encoding and to the variation in the UTC drift estimation (around 10 ms).

We construct an histogram to provide a quick summary of the number of chunks per latency value for the total streaming session in Figure 4.16. As can be seen on the histogram, the smallest chunk latency is about 1 ms and the highest is about 54 ms. Most of the chunks (i.e. 3997 chunks) are between 1 ms and 5 ms of latency. For instance, the biggest number of chunks (i.e. 1278 chunks) have a latency of 3 ms. A handful of chunks (i.e. 26 chunks) are between 6-28 ms of latency. We can see on the right side there are a few chunks (i.e. 430 chunks) whose latencies are between 45–49 ms, and some chunks (i.e. 17 chunks) were between 50-54 ms. These last bars whose latencies are higher than the rest are the first chunks of segments. If we sum the chunks whose latencies that lies in 45–49 ms and 50-54 ms intervals, we obtain 447 chunks which is equivalent to the number of received segments during the streaming session.

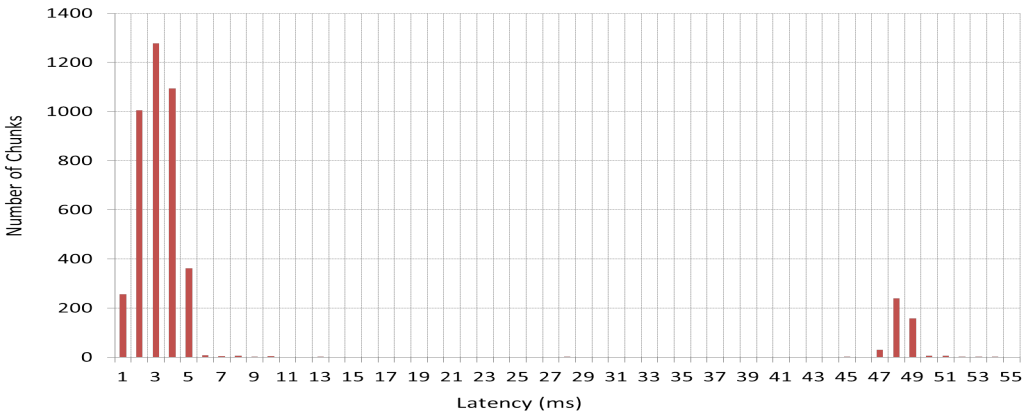


Figure 4.16 – Chunk latency histogram.

These results show that the client receives all HTTP/1.1 chunks that our web server sends, i.e. all fragments that the segmenter provides. This means that our web server identifies correctly the fragments when monitoring the segment file and then delivers them as chunks.
Additionally, when the client joins the live stream it makes requests for segments at the precise time, i.e. it is accurate to the first fragment availability time of a segment, because we do not have a case where the first received chunk contains $\#n$ fragments. Moreover, the latency of the total frames of the streaming session is constant as shown in Figure 4.17. It does not exceed the higher latency of 210 ms which means that no drift between the web server and the client during the total streaming session.

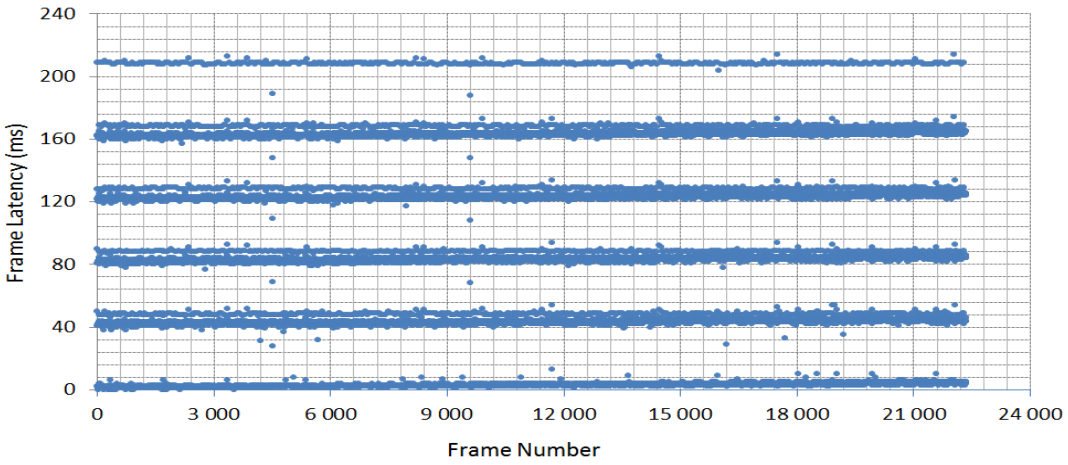


Figure 4.17 – Frame latency of the total streaming session (i.e. 22350 frames of 447 segments and 4470 fragments).

2. End-to-end latency

For latency measurements, we added a new box called producer reference time box ("prft") which is located before any "moof" box in the movie fragment as shown in Figure 4.18.

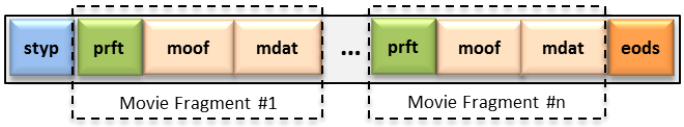


Figure 4.18 – Structure of a media segment with "prft" boxes.

We added the NTP time at which the first frame of each fragment was captured in the "prft" box of each fragment. When the MP4Client receives the fragments from the server specifically the first frames of those fragments, it fetches its own NTP system time, substracts the time read from the "prft" box and obtains an estimated latency. Compared with previously, we measured the latency between the capture and the display. We set d_s to 1 s. We used one frame per fragment and then 5 frames per fragment.

(a) One frame per fragment

We captured from a video webcam device the content at a resolution of 640x480 pixels, at 30 frames per second using the following command:

DashCast -vf dshow -vres 640x480 -vfr 30 -v video="HD Webcam C615" -pixf yuv420p -live -low-delay -seg-marker eods -insert-utc -min-buffer 0.05 -frag 33 -ast-offset -1000

We set d_c to 33 ms (approximately one frame duration). We set ATO to 1000 ms which means that a client makes requests for segments at the beginning of the first frame, i.e. when the first frame is completely encoded. In this configuration (one frames per fragment), the fragmentation overhead is 0.6% as shown previously in Figure A.9.

As shown in Figure A.11, MP4Client was configured to provide two buffers, decoding and composition. We set the length of the decoding buffer to 10 ms. So, when the buffer receives a frame its level exeeds 10 ms which means that the frame should be played out immediately. As shown in Chapter 2, the decoding time of I frames is higher than P frames and even higher than B frames. To smooth the playback, we used a composition buffer after the decoder. We set its length to 2 frames (approximately 66 ms).

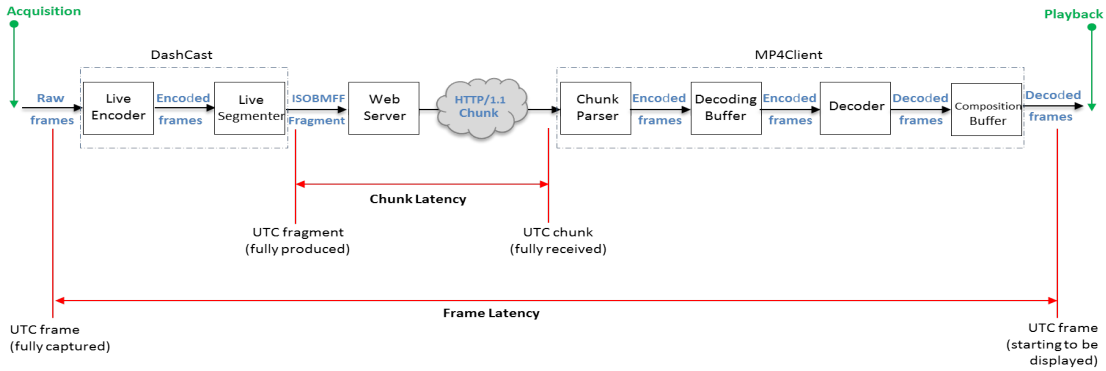


Figure 4.19 – End-to-end latency measurements for live streaming service.

Figure 4.20 shows the latency of the first frame of each segment, i.e. the difference between the time at which the frame was captured and at which it was starting to be displayed as shown in Figure A.11. It represents the latency for 10 segments. As we can see the latency is almost 100 ms which consists in the sum of the encoding delay, the segmentation delay, the network delay, the decoding buffer

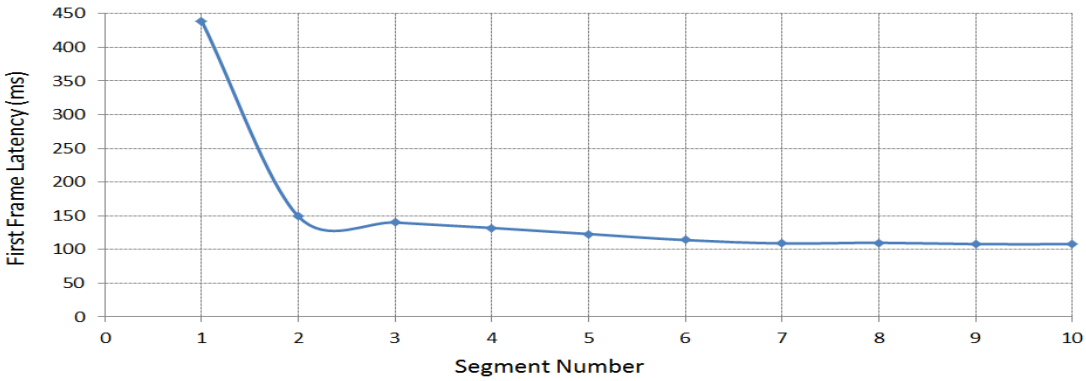
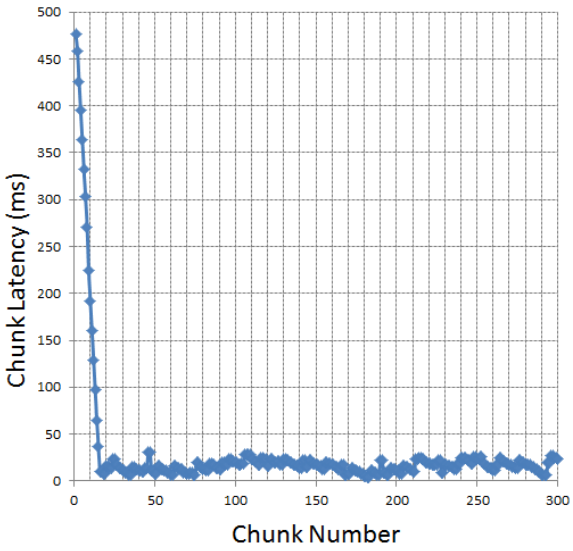


Figure 4.20 – Latency of the first frame of 10 segments (d_s=1s, d_c=33ms, ATO=1000ms).

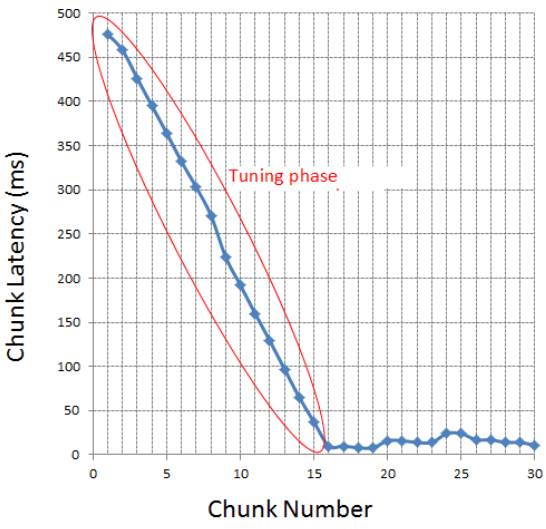
delay, the decoding delay, and the composition buffer delay.

The buffering delays are in the order of 99 ms (approximately 3 frames). The encoding, network, decoding delays are negligible (i.e. in the order of microseconds). As we use one frame per fragment, the segmentation delay is instantaneous (less than 1 ms).
The first frame of the first segment of the streaming session has a high latency of 439 ms. This is due to the client that joins the live stream a certain time after the beginning of the current segment, and plays in fast-forward mode that segment up to the current time. This latency is however reduced for the next segments once the client has played the first segment and makes requests for next segments at their precise availability time, i.e. once the first frame of the segment is available.

We also measured the chunk latency and the results are shown in Figure 4.21(a). With 30 frames per second, and approximately 33 ms per chunk, a segment contains 30 chunks. We represent the latency of 300 chunks which is equivalent to 10 segments. As can be seen on the figure, the chunk latency is in the order of 20-25 ms for each chunk which is below the chunk duration.



(a) 300 chunks of 10 segments.



(b) Zoom on the tuning phase of Figure 4.21(a) (30 chunks of 2 segments).

Figure 4.21 – Chunk latency (d_s=1s, d_c=33ms, ATO=1000ms).

In traditional streaming systems (broadcast or multicast IP), when a client connects to a session it has to wait for the next RAP to start decoding and playing, which introduces a delay. In RTSP system however, a server usually adjusts the time requested by the client to the nearest RAP in the stream in

order to avoid such waiting delay. In HAS solutions, when the client connects to the live stream after a certain time of the beginning of the current segment, it has the choice to either wait for the next segment and makes the request at the right time or request the current video segment, decode all frames very fast up to the current time (i.e. live time) and then starts playing. To reduce the startup delay (seen in Chapter 3), we used the second strategy in our experiments.

We provide a zoom of the tuning phase of Figure 4.21(a) in Figure 4.21(b). We can note that the client joins the live stream approximately at the middle of the current segment. The tuning phase shows 15 HTTP/1.1 chunks containing 30 fragments. The first chunk incurring a high latency of 439 ms includes all produced fragments since the beginning (i.e. 16 fragments) of the media segment. The remaining 14 chunks contain one fragment and their latency decrease progressively.

(b) Five frames per fragment

We set now d_c to 166 ms where each fragment contains 5 frames. Thus, a segment of 1 s includes 5 chunks and 30 frames. ATO equals to 867 ms which corresponds to the availability time of the last frame of the fragment. In this configuration (5 frames per fragment), the fragmentation overhead is 0.2% as shown previously in Figure A.9.

Figure 4.22 shows the latency of the first frame of each segment from the capture to the playback. The figure represents 10 segments. As it can be seen, the first frame experiences a high latency in the order of 220 ms. The main contributor to this latency is the segmentation delay (133 ms) where the first frame would wait for the four remaining frames to construct the chunk, as explained previously. Additionally, the latency of the two decoded frames (66 ms) that are stored in the composition buffer. We can see a high latency (around 586 ms) in the initialization phase that decreases progressively. This is due to the client that joins the live stream after a certain time of the beginning of the current segment and choose to request that segment that is partially generated as explained previously.

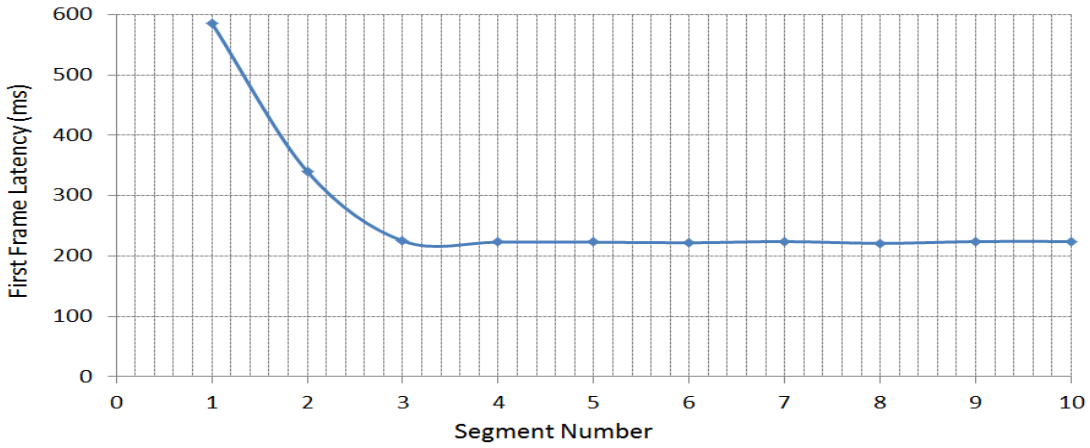


Figure 4.22 – Latency of the first frame of 10 segments (d_s=1s, d_c=166ms, ATO=867ms).

Figure 4.23 shows the chunk latency for 50 chunks. We can see that the latency is around 150 ms which consists in the sum of the segmentation delay (133 ms) and the transmission delay (approximately 25 ms).

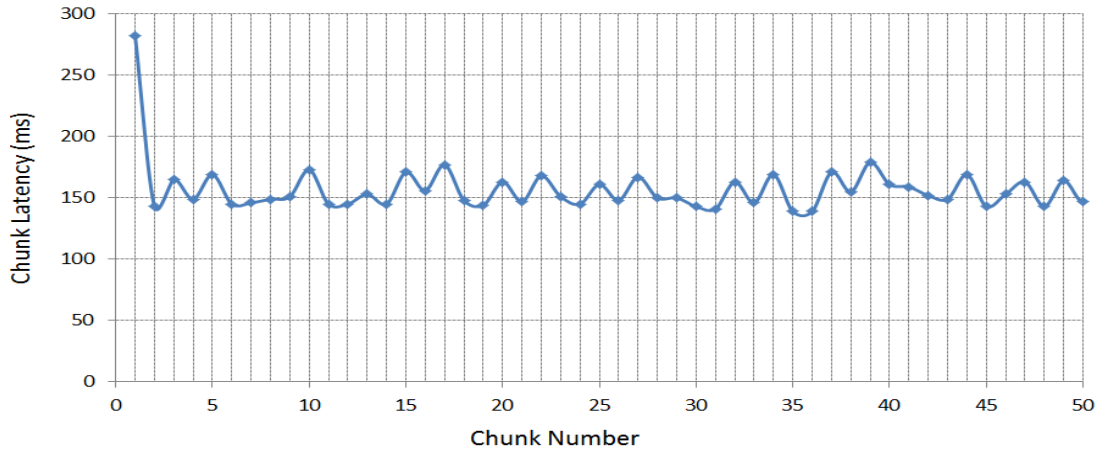


Figure 4.23 – Chunk latency of 50 chunks of 10 segments (d_s=1s, d_c=166ms, ATO=867ms).

4.5 Conclusion

In this chapter, an analysis of the latency of live streaming services using DASH was presented, indicating that the major component inducing extra latency compared to other delivery systems was the segmentation process. We have proposed a new method to reduce the end-to-end latency, specifically the segmentation delay in live DASH streaming without requiring changes in the infrastructure. It is based on specific packaging format, i.e. ISOBMFF movie fragments, and on HTTP/1.1 Chunked-Transfer Encoding mechanism. This approach enables a client to send out a request as soon as some parts (fragments)

of a segment are ready. With this approach, the fragments are pushed to the web server early and the download of the segment can start before it is completely ready. In order to validate our approach, we have conducted two types of measurements.

First, we measured the overhead introduced by the ISOBMFF packaging, the associated fragmentation and the transport tools such as chunked encoding. We have shown that the fragmentation process of DASH is the main contributor to the overall overhead. The overhead introduced by the simple storage of encoded media frames into the structured ISO format as well as by the HTTP/1.1 chunked encoding delivery is negligible. As a result, we have shown that for 3 frames per fragment the overhead is fewer than 4% for all resolutions and bitrates, but more interestingly, that the maximum overhead that can be reached when using one frame per fragment for classical resolutions (SD and more) is 1%, which is less than when using RTP.

Second, we have measured the latency at first from the encoder output to the decoder input. A very low latency in the order of 160 ms can be achieved when using 5 frames per fragment. The measured latency from the capture to the display is in the order of 100 ms for a fragment being only one frame. It is approximately 225 ms for a fragment containing 5 frames. Through these results, we have validated that our method is capable of achieving a very low latency.

In future work, we plan to examine how such low latency system will behave in real content delivery networks, and to further evaluate our proposal with multiple players and switching phases. Additionally, we plan to measure the latency in real-time interactive service. Finally, we could investigate the method that reverses the ”moof” and ”mdat” boxes in the media segment structure and check if the latency is even further reduced than the typical method.

The work presented in this chapter has been published in the the 5th international conference on Information, Intelligence, Systems and Applications (IISA) in 2014. It has been also standardized in the first DASH amendment and is being deployed by several companies (Harmonic, Ericsson, etc).

Chapter 5

Hybrid Streaming Services

Contents

5.1 Introduction

Recent years have seen the increasing presence of connected devices in the home network like mobile phones, tablets, set-top-boxes and TV sets as well. Some devices are capable of connecting to several networks concurrently. For instance, connected TVs can receive content from broadcast channels (e.g. terrestrial and satellite) as well as from broadband networks. On broadcast channels, identical content can be efficiently delivered to a lot of viewers, but delivering personalized content or high content quality (e.g. UHD, HDR, etc) that can only be decoded by a small subset of TVs is impractical due to bandwidth limitation. On broadband networks, personalized content can be delivered to individual viewers, but stable delivery to a large audience is costly. [58] With the growing adoption of connected TVs, it is interesting to enhance broadcast services with premium services of various kinds (e.g. alternate views, alternate audio, subtitles, sign language, etc) via

broadband. Moreover, it is interesting to deliver a basic content quality (e.g. HD) to all viewers via broadcast and an additional content quality to only viewers that are eligible for a high quality (e.g. UHD) via broadband. This is called "transition phases" service in DTV industry. Combining broadcast channels and broadband networks is sometimes named hybrid delivery systems.

HTTP adaptive streaming solutions, including DASH, are the most popular systems for content delivery on unmanaged broadband networks. In this chapter, we investigate scenarios where we consider basic contents delivered over traditional broadcast channels enhanced with additional contents delivered over unicast IP using DASH. In this case, the latency of the DASH system should be lower than the broadcast. Otherwise, it may be necessary to delay the broadcast content by introducing additional buffers either at the client side or at the encoder side.

In this chapter, we use the approach described in the previous chapter to reduce the latency in live DASH, and we show how such a system can be used to enable combined broadcast and broadband services while keeping the buffering requirements on the broadcast link low. Additionally, we show how data from both channels can be accurately synchronized.

This chapter is organized as follows. Sections 5.2 present the challenges of a hybrid delivery system. Section 5.3 demonstrates our proposed hybrid delivery system. Section 5.4 describes an experiment made to validate the approach and Section 5.5 concludes the chapter.

5.2 Hybrid Delivery Challenges

This section summarizes the main issues that we have identified in the delivery of media content over hybrid broadcast/broadband networks.

- Stream location

  In hybrid delivery, when a client connects to the broadcast stream, it needs to locate the upcoming external media stream delivered over a broadband channel [59]. For instance, if DASH is used, it needs a link to download the MPD.

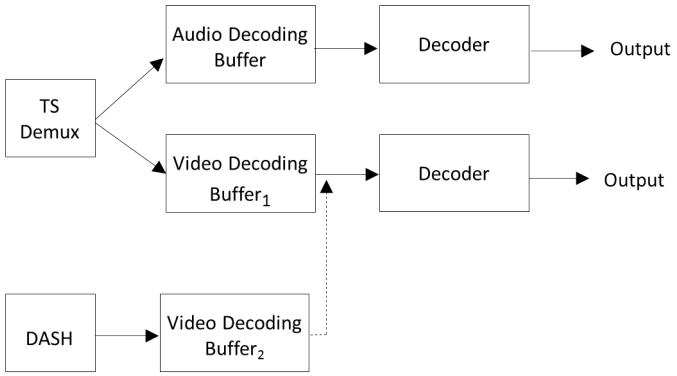
- Stream synchronization

  In hybrid delivery, we are interested in inter-stream multi-networks synchronization [60] [61]. The different streams, i.e. TS and DASH streams, are not signalled as being synchronized because they are generated based on different clocks and delivered through different distribution networks. The clock of MPEG-2 TS is the Program Clock Reference (PCR). All media timestamp information (DTS/PTS) of each frame refers to PCR. In DASH, the presentation time of each frame maps to the MPD

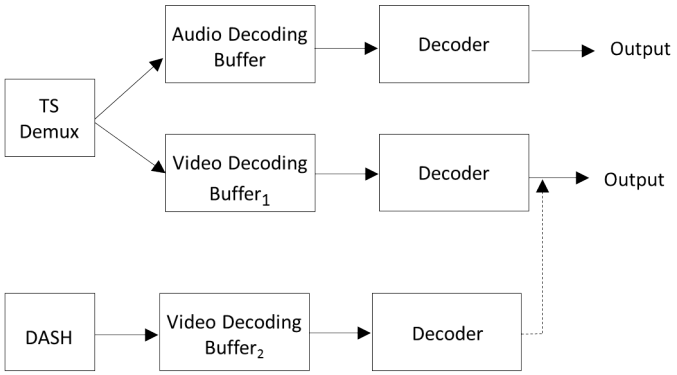
media presentation timeline which is common to all DASH representations in the Period. The presentation time offset of the representation relative to the start of the Period is given through the *presentationTimeOffset* attribute in the MPD to map actual timestamp to the MPD media presentation timeline.

There is therefore a need to synchronize those streams. However, different services may require different levels of synchronization. Some use cases of real-time interactive streaming service (e.g. scalable/multiviewing video coding) require frame-accurate synchronization of the different streams delivered through different distribution networks on the client device. This frame-accurate synchronization may be required for synchronized decoding or synchronized rendering.

If N streams delivered through N distribution networks are decoded by a single decoder that outputs a single stream, the synchronization at the frame level must be done before the decoding process. In other words, the frames coming from N streams must be processed by the decoder at the same time. Figure 5.1(a) illustrates this use



(a) Single video decoder for TS and DASH streams and frame-accurate content synchronization before the decoding process.



(b) Video decoder for each TS and DASH stream and frame-accurate content synchronization after the decoding process but before the presentation.

Figure 5.1 – Use cases with a frame-accurate content synchronization.

case with one TS and DASH stream.
If the N streams are decoded by N decoders but only one stream is recomposed at the end as depicted in Figure 5.1(b), the frame-accurate synchronization is done after the decoding process but before the presentation. The only requirement in this case is that the frames of the N streams must be displayed at the same time regardless their decoding times.
Any delay between the frames coming from N streams may be a problem in the decoding or in the rendering processes which can be noticeable in the playback and result in a degradation in the quality experienced by viewers.

In live streaming, use cases that have less constraints in synchronization (e.g. Picture/Picture and audio descriptions) do not require a frame-accurate content synchronization. The introduced delay between the frames of the N streams may not deteriorate the viewer QoE if it is not too important.

- End-to-end latency

  The final challenge in such hybrid scenario is that typical broadcast channels feature a constant latency, usually lower than HTTP adaptive streaming solutions, including DASH. Therefore, we aim at providing a novel DASH system with latency close or lower to broadcast channels without introducing any additional buffers at the broadcast chain.

5.3 Hybrid Delivery Proposed System

In this section, we present our proposed system that combines DASH streaming over broadband with broadcast and uses two new functionalities, i.e. TEMI and the low end-to-end latency live DASH system.

5.3.1 Timeline and External Media Information (TEMI)

TEMI [62] enables signaling and synchronization of external enhancements of contents carried over MPEG-2 TS. Specifically, it enables transport of a media timeline in an MPEG-2 TS content and signaling of the location of current and potentially upcoming external media enhancements carried over a broadband channel.
In order to provide frame-accurate timeline alignments despite the potential PCR discontinuities that typically occur in an MPEG-2 TS network, different types of time codes can be inserted into the TEMI such as times relative to global clocks (e.g. NTP) or to an external media clock (e.g. DASH). The TEMI information can be sent in a dedicated PES stream identified in the Program Map Table (PMT), for cases where bandwidth

requirements are not too constrained, or can be inserted in the adaptation field of TS packets of a media elementary stream when the overhead of sending one TS packet per time information would be too high.
The payload of a TEMI PES packet contains a single complete access unit composed of one or several AF descriptors. AF descriptors are structures used to carry various features of the timeline or other information; they all have a format which begins with an 8-bit tag value that identifies the descriptor type (e.g. timeline and location descriptors). The tag value is followed by an 8-bit AF descriptor length and data fields. Location descriptor is used to signal the location of external data that can be synchronized with the TS content. The timeline descriptor is used to carry timing information that can be used to synchronize external data. AF descriptors of different types may be sent in different access units and at different rates, and are independently decodable (all TEMI access units are therefore random access points).

5.3.2 TEMI and Low End-To-End Latency Live DASH System

A low-latency DASH broadband channel is produced with the system described in Chapter 4. Additionally, an MPEG-2 system generates a multiplexed TS with the TEMI stream. TEMI carries the HTTP URL of the MPD and the corresponding media time in the DASH session.

When the broadband content to be synchronized with the broadcast is live content, i.e. not entirely available at the beginning of the session, DASH requires the use of 'dynamic' MPD as seen in Chapter 4, which implies accurate UTC clock at both server and client sides in order to locate the live edge. As seen in Chapter 4, the live edge, i.e. the media segment number "i" in the period being produced at time UTC, can be computed as:

$$i = \left\lfloor \frac{UTC - (AST_{MPD} + periodStart)}{d_s} \right\rfloor + startNumber \tag{5.1}$$

In hybrid delivery systems, the UTC timing can be obtained from the broadcast or broadband environment.
On the first hand, most broadcast systems are designed to work on non-UTC synchronized devices, and consequently most DTV receivers do not maintain a precise UTC clock. On the second hand, in broadband systems, the accurate UTC timing configuration of both server and clients may not be always satisfied: different, not accurately synchronized NTP servers may be used by client and servers; or no NTP may be available at the client side. The latest DASH standard allows embedding either an URL to fetch the specific time server or directly the time value in the *UTC timing* element in the MPD. The client is required to retrieve the time information before processing segments if an URL is indicated,

therefore introducing an additional delay. In order to avoid this, we propose to inject NTP time used by the DASH origin server in the TS broadcast as shown in Figure 5.2, thereby helping the client find the live edge unambiguously.

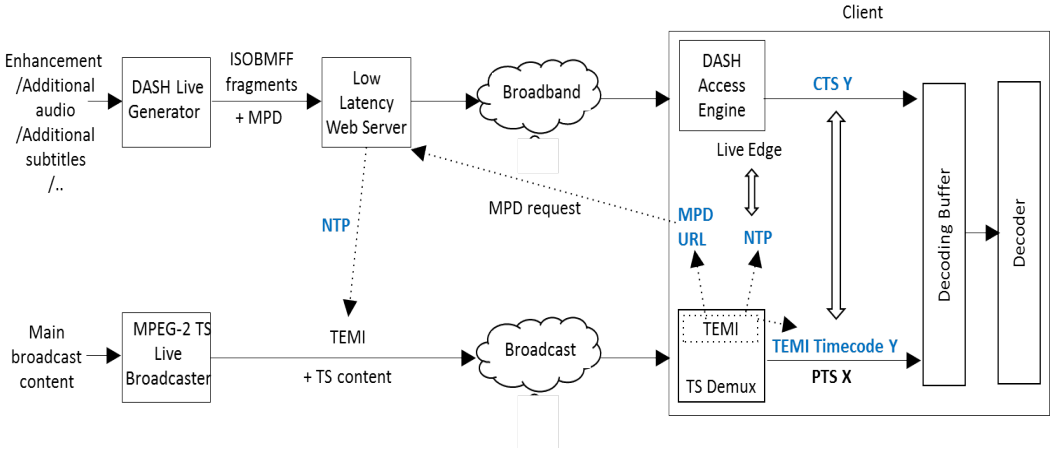


Figure 5.2 – Hybrid broadcast/broadband delivery proposed system.

The client is capable of receiving both TS and DASH streams through different delivery networks. The MPEG-TS demultiplexer is in charge of demultiplexing the TEMI from the TS stream as shown in Figure 5.2. Following that, it parses timeline and location descriptors and notifies the client about available additional content and the timecode corresponding to the current time in the MPEG-TS program (PCR). Based on the MPD URL carried in the location descriptor, the client sends an HTTP request to fetch the MPD file from the web server.

Figure 5.3 shows an example of the client reception of TS and DASH frames. A DASH fragment is made of 5 frames and the availabilityTimeOffset (ATO) indicates the time at which one fragment is available.
As shown in Figure 5.3, the client starts receiving the TS frames one by one. Based on the NTP injected in some frames, the segment in which the desired frame (e.g. f_1) is packaged is computed using the Equation 5.1. Using the availabilityTimeOffset (ATO), the client waits for the movie fragment of that computed segment to be ready and makes the request. The DASH frames are received at once as they are packaged in one fragment. As shown in Figure 5.3, the DASH frames are received after the TS frames. Hence, the reception time of a DASH frame minus the reception time of the same frame from the TS is positive. However, some TS frames could arrive later than the DASH frames if the TS propagation time would be longer than the request/response transmission delay on the broadband network. Note that this behavior assumes that the DASH and TS encoders are perfectly synchronized.

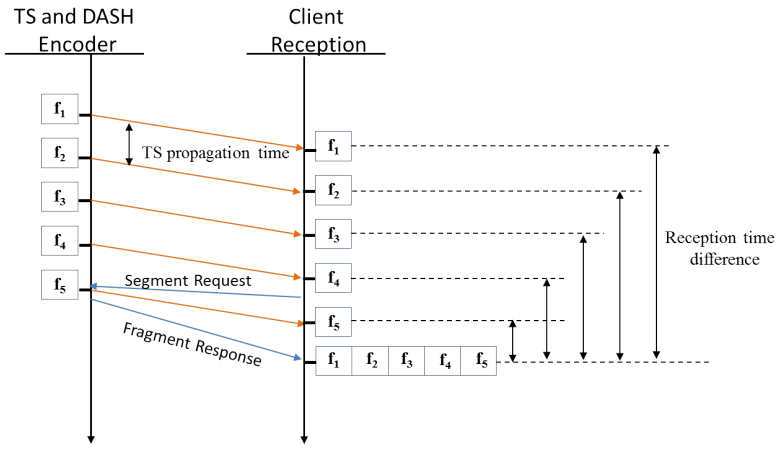


Figure 5.3 – Client reception of TS and DASH frames.

5.4 Evaluation

5.4.1 System Implementation

For the evaluation of our proposed hybrid delivery system, we consider the use case of multi-resolution content (scalable content). It consists in an HEVC encoded video at HD resolution broadcasted as MPEG-2 TS to be synchronized with an SHVC video enhancement layer at 4K resolution encapsulated in a DASH presentation.

In the following, we present the scalable content generation procedure, the test-bed architecture as well as the settings of each network.

5.4.1.1 Content Generation

We used an HEVC stream with one enhancement layer as input encoded bitstream to generate the contents to be delivered over broadband and broadcast networks:

1. import the video stream in an ISOBMFF file, with one track per layer:
   MP4Box -add input_video_3840x1600.hevc:fps=24:svcmode=splitnox:noedit -new video_3840x1600.mp4
2. Create an ISOBMFF file containing only the base layer at HD resolution (HD HEVC):
   MP4Box -add video_3840x1600.mp4 -new video_1920x800.mp4
3. Create an ISOBMFF file containing only the enhancement layer at 4K resolution (UHD SHVC):
   MP4Box -rem 1 video_3840x1600.mp4

5.4.1.2 DASH over Broadband Network

A low latency DASH system is in charge of delivering the SHVC enhancement video layer over the broadband network with the system described below and composed of the following open-source tools, developed within GPAC[1]:

- Live DASH encoding simulator and packager

  It uses as input an already encoded video, i.e. an SHVC enhancement layer stored in an MP4 file. It packages and divides the stream into ISOBMFF compliant movie fragments. To simulate a real-time live, MP4Box is configured to flush each fragment to disk at the end of its duration using the "-frag-rt" operation. This ensures that the client cannot get any future fragment ahead of time. It also produces the MPD describing the service.
  The exact command line for creating the DASH session for the scalable stream is provided below:

  MP4Box -dash-live 1333 -frag 208 -frag-rt -min-buffer 208 -insert-utc -segment-marker eods -profile live -segment-name segment -out manifest.mpd -mpd-refresh 20 -ast-offset -1125 video_3840x1600.mp4

  The input scalable video has the following characteristics: a duration of 12min13s, a GoP length of 32 frames, and a frame rate of 24 fps. We have configured MP4Box to produce segments of one GOP (1333 ms). We obtained 550 segments for the whole video duration. Each media segment is divided into multiple movie fragments of 5 frames each (208 ms) except the last one which is made of two frames. We obtained 7 fragments per segment.
  We set the *availabilityTimeOffset* (ATO) value in the MPD to 1125 ms which corresponds to the time at which one fragment is available. Thus, a client is aware that a fragment of the segment is available earlier than the segment to make a request.

- HTTP/1.1 compliant server [2]

  The HTTP server is the same ISOBMFF-aware web server used in Chapter 4. It is used to deliver both the MPD and ISOBMFF media segments. To achieve low latency, the server detects when new movie fragments are flushed on disk and pushes them immediately to client using HTTP/1.1 Chunked-Transfer encoding mechanism.

[1] https://gpac.io/
[2] https://github.com/gpac/node-gpac-dash

5.4.1.3 Broadcast Channel

A broadcast channel is simulated by the use of an IP multicast delivery of an MPEG-2 Transport Stream (TS), generated from the base HEVC layer. The MPEG-2 TS PCR information is randomly initialized at startup to demonstrate synchronization aspects. The TS stream is enriched with location (i.e. the HTTP URL of the DASH session) and timing information (i.e. UTC time and PCR-to-DASH-time mapping) that are signaled using TEMI.
We start the multicast stream using the following command line:

MP42TS -src video_1920x800.mp4 -single-au -dst-udp 239.255.0.1:1234 -temi-noloop -temi <MPD URL> -ifce <IP address server> -insert-ntp -rate 10000

5.4.2 Experiments and Results

In order to validate our hybrid approach, we have conducted an experiment to measure the buffer length and the arrival time of each frame from both MPEG-2 TS and DASH streams.

Figure 5.4 represents the buffer length of both streams at the client side. The buffer length is measured in both cases as the media time of the frame received last minus the media time of the frame currently being played. We can see that at t=0 s the client starts receiving the TS frames and the buffer is filling up progressively until it reaches about 680 ms per configuration. After that, the buffer length remains constant because after each played frame a new frame is received.

At t=1.8 s approximately, the client starts receiving and buffering the DASH movie

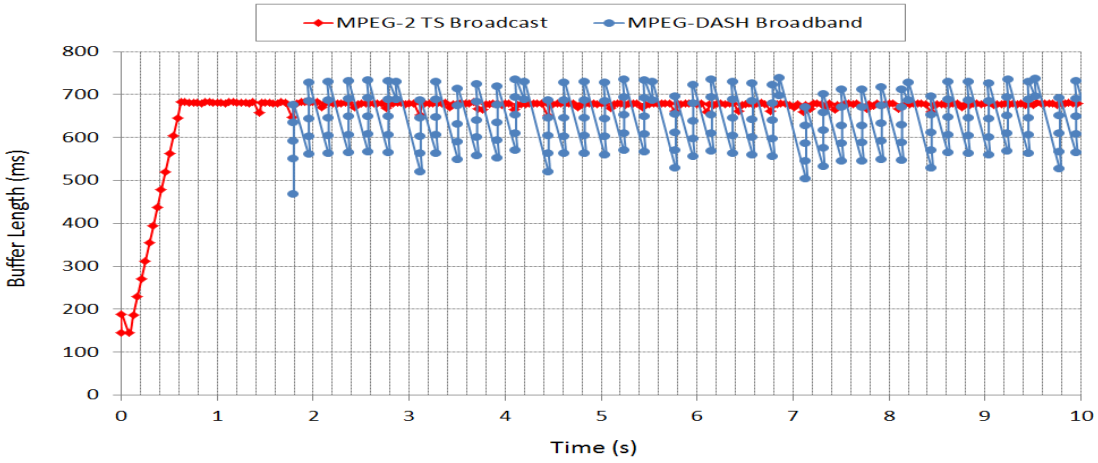


Figure 5.4 – Buffer length of MPEG-2 TS and DASH streams.

fragments. First, we can see that the DASH buffer length starts at a value of 468 ms, meaning that the received frame is meant to be played after 468 ms. This means that the DASH client is not fetching the live edge, but it is fetching DASH frames that have media times compatible with what the TS buffer contains. This guarantees synchronization. Then, we can see the 5 frames of a fragment are received at the same time (vertical lines), and the last frame of the fragment is meant to played after 675 ms.

We can see on the figure that sometimes the DASH buffer length graph is above the TS buffer length graph. This means that some DASH frames arrive before the TS frames. As explained in Section 5.3.2, in theory, DASH frames arrive after TS frames excpet if the TS propagation time is longer than the HTTP request/response transmission delay. In our experiment, we started the TS and DASH sessions, i.e. MP4Box, MP42TS and the web server, at the same time. Based on the obtained results, we realize that the DASH generator is started a few milliseconds before the MP42TS. This can explain the DASH frames that are received before the TS frames.

Based on the results, we realize that we could set the broadcast buffer length to 212 ms (i.e. 680 - 468 = 212 ms) instead of 680 ms since the minimum value of the DASH buffer is 468 ms. However, in OTT networks, these 468 ms of supplemental buffer can be used to deal with the jitter.

Additionally, in our experiments on a LAN, the TS packet delivery time and the HTTP request/response times are both negligible. In practice, if the TS packet delivery time (e.g. via satellite) is larger than the request/response times, this additional TS buffer could even be removed but this remain to be investigated.

Figure 5.5 represents a screenshot of MP4Client receiving and playing the TS and DASH streams. With the GUI mode, we can see the client statistics such as the bandwidth, the buffer level, the framerate, and the CPU.

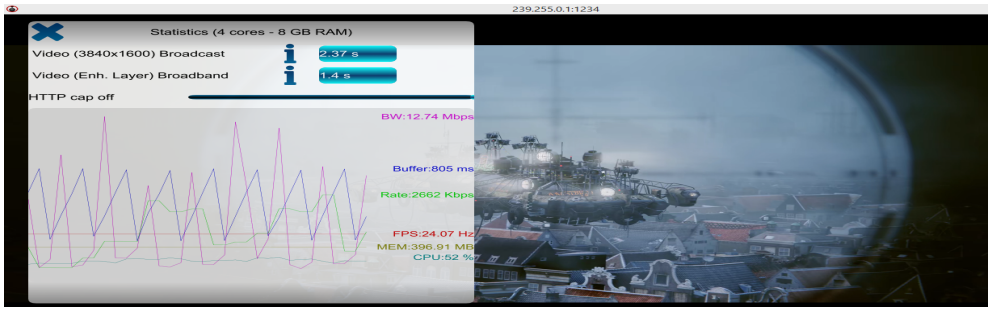


Figure 5.5 – Screenshot of MP4Client in the GUI mode.

5.5 Conclusion

Hybrid delivery of media content allows providing additional and customized content on the broadband network, synchronized to the content delivered over the broadcast channel. Extending broadcast services with HTTP streaming solutions requires accurately synchronizing data from both channels, possibly at the frame level (e.g. for scalable enhancement) introducing no or small additional buffers and latency. In this chapter, we have proposed to use TEMI to locate and synchronize external DASH content with MPEG-2 TS content. We have demonstrated a low latency DASH system for both broadband and hybrid broadcast/broadband delivery chains, based on open source tools and standards for a multi-resolution content usage scenario. Our proposed hybrid delivery system has been implemented and validated in local networks. In future work, we plan to examine how such system will behave in real content delivery networks, and if the network bandwidth variations on the broadband system can delay the DASH stream compared to TS and hence impact the synchronization of both streams and the buffering requirements on the broadcast chain.

The work presented in this chapter has been resulted in one paper published and demonstrated in the ACM 23rd International Conference on Multimedia (ACM MM) in 2015 [63].

Chapter 6

Conclusion & future work

Contents

6.1 Thesis objectives

The purpose of this thesis was to introduce and develop new solutions to achieve fast live DASH streaming startup, low latency live DASH content delivery, and hybrid delivery of broadcast and broadband contents (i.e. DASH and MPEG-2 TS contents). Three main contributions have structured this thesis:

- The proposal and the evaluation of different methods for reducing the startup delay, specifically the bootstrap delay in live DASH.
- The development and the evaluation of a complete novel low latency live DASH system.
- The synchronized combination of the broadband and broadcast contents.

6.2 Summary

In this section, we summarize our major contributions and the key results of this thesis while citing the limitations.

Fast DASH Bootstrap

Our first contribution in this thesis is related to the improvement of the bootstrap phase of DASH. It aims at reducing the bootstrap delay in live DASH. The first step was to analyze existing DASH client bootstrap strategies in terms of number of TCP connections, number of HTTP requests/responses and the associated bootstrap delay. The results of this analytical evaluation reveal a need for exploring and designing new methods in which the bootstrap phase should not require multiple round-trips between the client and the server. Our proposed methods use a single HTTP response and HTTP request to retrieve the necessary information (i.e. MPD and initialization data) to start the initial playback. The idea of our main method "ISOBMFFMoov Embedding" is to rely on the MPD to carry the additional IS when creating the HTTP response for the MPD request.

The results show that the total download size when using our method over HTTP/1.1 and HTTP/2 respectively is reduced by an average of 25% and 21% compared to the HTTP/1.1 persistent approach without pipelining and the HTTP/2 push method. Furthermore, we show a gain of 2 RTTs in HTTP/1.1 and no penalty when using HTTP/2. This suggests that our method can be valuable even during the transition phase to HTTP/2. More interestingly, our method over HTTP/2 is more efficient than HTTP/2 server push in terms of the amount of data being downloaded. We show that reducing the bootstrap delay is critical to reduce the startup delay when the buffering delay is small which is the scope of our second contribution. Our method is generic and can be applied to HTTP adaptive streaming solutions. Additionally, it is compatible with the existing caching and delivery infrastructure.

Low latency live DASH system

This second contribution is designed for interactive or bidirectional applications such as video conferencing and live streaming, or for hybrid delivery scenarios. Our contribution consists in proposing a complete novel low latency live DASH system. It aims at reducing the end-to-end latency, specifically the segmentation delay that we have identified as the greatest latency contributor in live DASH compared to traditional delivery systems. The proposed model is able to achieve low latency based on the HTTP/1.1 Chunked-Transfer Encoding mechanism and on a specific packaging using the ISOBMFF movie fragments.

With this system, we validated our approach for very low end-to-end live latency streaming in local networks, with latency in the order of 100 ms and an overhead of 0.6% approximately for a fragment being only one frame. We manipulate here the live video streams at the frame scale. Our system does not consider any structure within the frame because it relies on the ISOBMFF packaging format that does not allow the fragmentation at the frame level.

This system is quite successful already because it is being deployed by several companies.

Hybrid broadcast/broadband delivery system

Hybrid delivery is one of the multimedia applications that require a low latency delivery. Our contribution consists in using broadcast and broadband delivery services and synchronizing streams with frame-level accuracy. In other words, a basic content is delivered over traditional broadcast channels and is enhanced with an additional content which is delivered over unicast IP using DASH. Synchronization of both contents as they are delivered through different distribution networks and keeping the client buffering requirements on the broadcast link low were our two challenges. To overcome these problems, we proposed a model that uses the above low latency DASH system and TEMI to locate and synchronize external DASH content with MPEG-2 TS content. Through experiments, we demonstrated that this model provides synchronization with or no small additional buffers.

6.3 Perspectives

At the time of concluding this manuscript, several interesting perspectives can be proposed to continue the work done in this thesis. Interesting extensions to improve the performance and other research directions in the context of video streaming adaptation are listed below.

- We manipulate the live video streams at the frame scale, i.e. the ISOBMFF packager is limited to produce fragments from the frame level. Enabling the packaging of small parts of a frame could be needed in low latency services.
- Another interesting research topic could be the examination of how such low latency system will behave in real content delivery networks. Until now, the DASH client was directly connected to the web server in a local network. A CDN has clever rerouting, redirection and caching mechanisms which can influence the delivery of a fragment to the client. Early experimentations show that our system behaves well, but further investigations are needed.
- In the current experiments, our fast DASH bootstrap proposal is only compared to the HTTP/2 server push method. It could be useful to compare it to other HTTP/2 mechanisms as such as the full request and response multiplexing.
- Switching has not been part of our research in this thesis but it is the next phase of a DASH client after the bootstrap and stable phases. We could investigate a fast switching process to our low latency live DASH system. In HTTP adaptive streaming, a client can only switch at the end of segment because switching in the middle of

segment requires at worse double download and decoding (as explained previously in Chapter 2). A client is constrained to wait the end of segment to switch. This waiting time for switching becomes even more significant when long media segments are used. A solution could be to use the RAPs contained in a segment as switching points. Thus, the client is able to switch within a segment at each GoP.

- Studying the impact of our low latency system on other recent use cases such as Virtual Reality (VR) applications, especially 360 VR videos is interesting. Adaptive 360 VR live video streaming presents many technical challenges including the end-to-end latency delivery and switching latency. Combining our low latency system with a new fast switching process may improve the viewer experience.

Annex A

Résumé en Français

A.1 Introduction

Le développement de l'industrie multimédia, l'évolution des systèmes de diffusion vidéo et l'émergence de différents périphériques d'affichage vidéo ont créé un grand nombre d'applications multimédia, y compris le streaming vidéo. Le streaming vidéo qui ne cesse de gagner en popularité a entraîné une augmentation des volumes de contenu vidéo. Aujourd'hui, le streaming vidéo correspond à 64% de la majorité du trafic Internet. On s'attend à ce qu'il atteigne 80% d'ici 2019. Cette demande croissante pour les services vidéo a changé les attentes des utilisateurs par rapport à la qualité.

Le streaming vidéo Over-The-Top (OTT) est devenu le moyen le plus rentable pour la diffusion de vidéos car il s'appuie sur le réseau Internet non managé. La diffusion OTT offre une liberté de visualisation car elle n'est pas limitée uniquement au PC, mais elle s'étend à tout périphérique connecté, par exemple TV, consoles de jeux, smartphones, tablettes, etc. Les utilisateurs peuvent bénéficier de la vidéo à la demande (VoD) et des services de streaming en direct (live) fournis par OTT. Le streaming live est plus populaire que la VoD, en particulier pour regarder des émissions de sport en direct dans le cas d'événements mondialement populaires.

La qualité d'expérience des services de streaming OTT en direct est généralement comparée à la qualité des systèmes de broadcast traditionnels tels que les systèmes numériques par câble, terrestre, satellite, et les réseaux IPTV. Dans les déploiements actuels, le streaming OTT en direct souffre de latences beaucoup plus élevées, généralement de quelques secondes jusqu'à une demi-minute par rapport aux services de broadcast. Cette latence est définie comme étant le délai entre le moment où l'événement survient et le moment où il est joué. Elle devient une problématique quand un utilisateur regarde un évènement en direct (e.g. un match de foot) sur son ordinateur et il entend ses voisins qui regardent le

même évènement à la télévision applaudissant le but marqué avant qu'il le voit sur son écran. Dans ce cas, cet utilisateur peut rapidement savoir qu'il y a un problème de latence. Il est donc susceptible de passer à d'autres systèmes de streaming pour qu'il soit proche du live. En conséquence, la latence devient ainsi un facteur important qui affecte la qualité globale vécu par les utilisateurs.

Un autre type de problème de latence dans le streaming OTT en direct est le délai de démarrage. Ce dernier est la différence de temps entre le moment où un utilisateur clique sur le bouton "Play" et le moment où la vidéo commence à jouer. En d'autres termes, le délai de démarrage est le temps nécessaire pour télécharger et bufférisér toutes les informations et les données média nécessaires pour démarrer la session de streaming vidéo. Les utilisateurs sont très impatients et moins tolérants aux démarrages lents dans les services de streaming en direct par rapport à la VoD [4].

En outre, le streaming à très faible latence est nécessaire pour les applications interactives ou bidirectionnelles telles que la vidéoconférence, le vidéo gaming, ou la télémédecine. Telles applications sont caractérisées par des contraintes de délais très strictes. Un autre cas d'utilisation où la faible latence est importante est le scénario de streaming hybride où les réseaux broadband et broadcast sont combinés pour améliorer et enrichir le service de broadcast avec des services variés par exemple rajouter et envoyer des sous-titres, langage des signes, audio avec plusieurs langues, via le réseau broadband. La latence du système broadband devrait être inférieure à la latence des systèmes broadcast pour la capture/génération en direct. Dans le cas contraire, des buffers supplémentaires sont nécessaires pour synchroniser le contenu broadcast avec le contenu broadband.

Ces dernières années, le streaming adaptatif sur HTTP est apparu comme la technologie de choix pour la diffusion des services OTT. Il permet une adaptation dynamique de la qualité de la vidéo à la bande passante variée du réseau et aux capacités des périphériques du client. Une nouvelle norme appelée MPEG Dynamic Adaptive Streaming over HTTP (DASH) a été développée et est utilisée dans le monde entier [1]. Certains travaux de recherche ont été proposés pour réduire les latences énumérées ci-dessus, dans le streaming adaptatif sur HTTP. Cependant, la latence est toujours dans l'ordre des secondes.

L'objectif de cette thèse est de proposer de nouvelles approches pour réduire le délai de démarrage ainsi que la latence de bout en bout observée par les utilisateurs lors de l'utilisation du streaming DASH live. Nous ciblons une latence très faible, c'est-à-dire une latence dans l'ordre des trames (par exemple moins de 200 ms).

Dans ce résumé, nous présentons nos contributions pour le streaming vidéo en direct utilisant DASH. Nous décrivons premièrement nos approches liées à l'amélioration du démarrage de la session du streaming DASH en direct. Ensuite, nous présentons nos solutions pour réduire la latence de bout en bout d'un système DASH en direct. Après,

nous exposons nos contributions sur l'application de distribution hybride. Enfin, nous conclurons ce résumé.

A.2 Contributions

Cette thèse présente des contributions liées aux services de streaming vidéo en direct utilisant DASH. Nous les avons organisés en trois thèmes: démarrage, distribution et applications.

A.2.1 Réduction du Délai de Démarrage en DASH Live

Dans ce domaine, nous avons examiné les causes possibles de délai dans la phase de démarrage d'une session DASH et nous avons proposés deux contributions.

A.2.1.1 Évaluation des Stratégies de Bootstrap du Client DASH

La première contribution consiste à évaluer différentes stratégies qu'un client DASH peut utiliser pour démarrer une session de streaming vidéo. L'évaluation analytique est faîte en termes de nombre de connexions TCP, nombre de requêtes/réponses HTTP et de délai de bootstrap associé. Nous avons trouvé que le délai de bootstrap dans toutes les stratégies est dominé par la composante RTT influencée par le nombre de connexions TCP et le nombre de requêtes, et par le temps de téléchargement des ressources dans la phase slow start. En se basant sur cette évaluation analytique, le délai de bootstrap minimal utilisant HTTP/1.x est obtenu à l'aide d'une connexion TCP persistante avec pipelining. Cependant, cette stratégie n'est pas largement supportée par les serveurs web et souffre encore d'un grand nombre de RTT dû principalement au nombre de requêtes/réponses HTTP.

Pour nos expérimentations sur HTTP/1.x, nous utilisons uniquement la connexion TCP persistante sans pipelining car elle est la stratégie la plus utilisée et supportée par les serveurs web. Les avantages de notre approche proposée seraient les mêmes par rapport à l'approche de pipelining.

A.2.1.2 Amélioration de la Phase Bootstrap en DASH Live

La seconde contribution comporte trois méthodes (i.e. Base64 IS Embedding, Multipart Content Embedding, ISOBMFFMoov Embedding) pour réduire le délai de démarrage, plus spécifiquement le délai de bootstrap en DASH live. Les méthodes proposées ont

été conçues pour ne pas avoir d'impact négatif sur les infrastructures existantes (caches et serveurs web). Elles sont basées sur l'idée que la phase de démarrage d'une session DASH ne devrait pas nécessiter de multiples RTTs entre le client et le serveur. Elles consistent à utiliser une seule requête HTTP et une réponse HTTP pour récupérer les informations (MPD et IS) nécessaires pour démarrer la session vidéo. La première requête HTTP effectuée par le client DASH pour récupérer le MPD n'est pas modifiée, mais la réponse envoyée par le serveur d'origine est modifiée. La création de cette réponse HTTP est de s'appuyer sur le MPD pour transporter les ressources IS. Cela peut se faire de trois façons.

1. Base64 IS Embedding

   Le principe de cette méthode est d'encoder le fichier binaire IS en ASCCI en utilisant l'encodage Base64. On met ensuite l'IS encodé dans l'attribut "initialization" du MPD en utilisant "data URI shceme" [1]. Figure A.1 montres un exemple du fichier MPD intégrant des IS encodés en Base64. L'avantage de cette méthode est sa compatibilité avec le standard DASH. L'inconvénient est qu'elle rajoute un overhead de 33%.

```
<MPD ...>
 <Period ...>
  <AdaptationSet ...>
   <Representation mimeType="video/mp4" ... >
                   codecs="avc1.4d401f" ...>
    <SegmentTemplate
       initialization="data:video/mp4;base64,
       AAAAGGZOeXB..."/>
   </Representation>
 <AdaptationSet ...>
   <Representation mimeType="audio/mp4" ... >
                   codecs="mp4a.40.5" ...>
    <SegmentTemplate
       initialization="data:audio/mp4;base64,
       AAAAFGZ0eXB..."/>
   </Representation>
   ...
</MPD>
```

Figure A.1 – Base64 IS embedding in MPD.

2. Multipart Content Embedding

   Dans DASH, chaque réponse HTTP contient une seule entité dans son corps. Notre deuxième méthode proposée consiste à la combinaison de différentes entités de types de données indépendants (MPD et IS) dans le corps unique de la réponse HTTP, en utilisant le type de média "HTTP/1.1 multipart". Pour répondre à la requête HTTP

[1] http://tools.ietf.org/rfc/rfc2397.txt

faite par le client DASH pour récupérer le MPD, le serveur devrait envoyer plusieurs parties (MPD et IS) dans le corps de la réponse HTTP. Dans notre approche, nous avons utilisé le sous-type "multipartite/mixte" vu que les différentes parties (MPD et IS) sont indépendantes. Comme le montre la Figure A.2, nous avons ajouté des en-têtes HTTP dans chaque début de partie, y compris l'en-tête "content-type" pour donner le type de média de ce contenu, "content-length" pour spécifier la longueur en octets de chaque partie, "content-disposition" pour nommer chaque partie avec un nom correspondant dans le MPD, et "content-transfer-encoding" pour indiquer quel type de transformation a été appliqué à la partie.

```
--?--?--
Content-Type:application/dash+xml
Content-Disposition:attachement;filename=MPD.mpd
Content-Transfer-Encoding:base64
Content-Length: 3343

<MPD ...>
 <Period ...>
  <AdaptationSet ...>
   <Representation mimeType="video/mp4" ... >
    codecs="avc1.4d401f" ...>
   </Representation>
<MPD ...>
--?--?--
Content-Type:video/mp4;codecs=avc1
Content-Disposition:attachement;filename=mysegs_mp4-onDemand-
h264bl_fullinit.mp4
Content-Transfer-Encoding:base64
Content-Length: 838

tdmhkAQAAAAAAADM9P...
--?--?--
```

Figure A.2 – "Multipart/mixed" content-type of MPD and IS entities.

Dans la méthode Base64 IS embedding, seuls les IS en Base64 sont intégrés dans le fichier MPD pour constituer le corps de la réponse HTTP. Dans la méthode Multipart content embedding, le corps de la réponse HTTP est un ensemble de parties (MPD et Base64 IS), et pour chaque partie, plusieurs en-têtes HTTP sont ajoutés. Par conséquent, cette dernière méthode introduit plus d'overhead que la première. Dans toutes nos évaluations, la méthode multipart est exclue.

3. ISOBMFFMoov Embedding

En examinant de plus près le problème, il semble que la plupart des informations utiles présentes dans l'IS sont aussi présentes dans le MPD. Le principe de notre troisième méthode consiste à ajouter au MPD les informations nécessaires pour reconstruire l'IS du côté client à partir de ce MPD uniquement. Pour cela, nous avons analysé le MPD et l'IS de différents contenus et nous avons identifié les informations

manquantes.

A partir de cette analyse, nous avons introduit un nouveau élément "ISOBMFFMoov" dans le MPD comme le montre la Figure A.3. Cet élément contient les informations suivantes:

- Configuration de décodeur (Stsd) en Base64
- Information de synchronisation entre pistes (Edit list)
- Identifant de piste si fichiers multiplexés (Track ID)

```
<MPD ...>
 <Period ...>
  <AdaptationSet ...>
   <Representation mimeType="video/mp4" ... >
                   codecs="avc1.4d401f" ...>
    <ISOBMFFMoov>
      <ISOBMFTrack id="1"
                   stsd="AAAlHN0c2QA..."
                   editDelay="0.04s"/>
    </ISOBMFFMoov>
    <SegmentTemplate ... />
   </Representation>
   ...
```

Figure A.3 – ISOBMFFMoov Embedding in MPD.

A.2.1.3 Expérimentations et Résultas

Base64 IS Embedding et ISOBMFFMoov Embedding ont été les deux approches que nous avons évaluées dans nos expériences. Nous avons mesuré et comparé la taille totale du téléchargement et le délai de bootstrap de notre proposition à plusieurs approches existantes. Ces mesures ont été réalisées sur deux réseaux différents (ADSL et mobile 3G) et en utilisant deux versions du protocole HTTP (HTTP/1.x et HTTP/2).

Nous avons d'abord mesuré la taille du MPD et de l'IS audio et vidéo pour chaque séquence. Tableau A.1 rapporte les tailles maximales, moyennes et minimales des tailles MPD et IS. Nous pouvons voir d'abord que ce sont des ressources de petite taille. Nous pouvons noter que la taille du MPD varie entre 1 kilooctet et 6 kilooctets. La taille de l'IS vidéo est comprise entre 600 et 900 octets, tandis que la taille de l'IS audio varie entre 600 et 800 octets.

Sequence Number	MPD Size (Byte)	IS Audio Size (Byte)	IS Video Size (Byte)
Maximum	**5768**	**776**	**848**
Average	**2998**	**755**	**824**
Minimum	**1482**	**615**	**687**

Table A.1 – MPD and IS sizes of 33 sequences.

Nous avons calculé ensuite la taille totale du téléchargement de chaque séquence lorsque nous utilisons nos deux méthodes (Base64 IS embedding, ISOBMFFMoov embedding), i.e. la taille du MPD généré plus la taille de l'en-tête HTTP de la réponse MPD. Nous avons comparé ces résultats avec la taille totale du téléchargement en utilisant une connexion TCP persistante sans pipelining lorsque:

- N_{IS}^{C} est maximal (i.e. égal à M) tel qui est implémenté par le player GPAC.
- N_{IS}^{C} est minimal (i.e. égal à N) tel qui est implémenté par le player Dash-JS.

La Table A.2 représente pour chaque méthode la taille totale du téléchargement de 33 séquences.

Sequence Number	N_{IS}^{C}=M (GPAC)	N_{IS}^{C}=N (Dash-JS)	MPD Base64 IS	MPD ISOBMFFMoov
Maximum	**11514**	**8858**	**10615**	**7047**
Average	**8793**	**6168**	**7866**	**4313**
Minimum	**5538**	**4331**	**4974**	**2731**

Table A.2 – Total download size (MPD, IS video, IS audio, and HTTP response headers) of 33 sequences for each method over HTTP/1.1 using a DASHIF server.

Nous avons remarqué que la taille totale de téléchargement est faible dans toutes les approches (inférieur ou égale à 12 kilooctets).
Nous avons noté aussi que la taille totale est supérieure quand on télécharge tous les IS séparément (tel que est implémenté dans GPAC player) par rapport à un téléchargement minimum (tel que est implémenté dans Dash-JS).
Nous avons montré aussi que la taille totale est inférieure quand on télécharge les IS en base64 par rapport au téléchargement séparé des IS.
Nous avons montré aussi que la taille totale est réduite dans notre méthode "ISOBMFFMoov embedding" de 36% par rapport à la méthode qui télécharge un minimum d'IS séparément sur HTTP/1.1.
En outre, nous avons montré que la taille totale de téléchargement est diminuée d'une moyenne de 21% dans notre méthode "ISOBMFFMoov embedding" sur HTTP/2 par rapport à la méthode qui utilise le serveur push sur HTTP/2.

Nous avons comparé, en termes de délai de bootstrap, notre approche basée sur "ISOBMFFMoov" d'abord à la méthode utilisant la connexion TCP persistante sans pipelining sur HTTP/1.1, puis à l'approche utilisant le serveur push sur HTTP/2.
Pour la stratégie persistante, nous avons mesuré en utilisant "Google Chrome Network Panel" le temps écoulé entre le moment où le player Dash-JS établit une connexion TCP pour demander le MPD du serveur web et le moment où il reçoit le dernier octet du dernier IS. Le temps de traitement du MPD par Dash-JS tel que rapporté par Chrome est déduit dans cette mesure. De plus, nous avons également mesuré le temps de téléchargement du MPD lors de l'utilisation de notre approche "ISOBMFFMoov".

Les Figures A.4 et A.5 montrent ces mesures lorsque les téléchargements ont été faits sur un réseau Ethernet, en utilisant HTTP/1.1 et HTTP/2, avec des init_cwnd variables (3 et 10 segments TCP). Elles montrent un gain de 2 RTT dans HTTP/1.1 et presque aucune pénalité lors de l'utilisation de HTTP/2 dans un réseau DSL avec le serveur push.

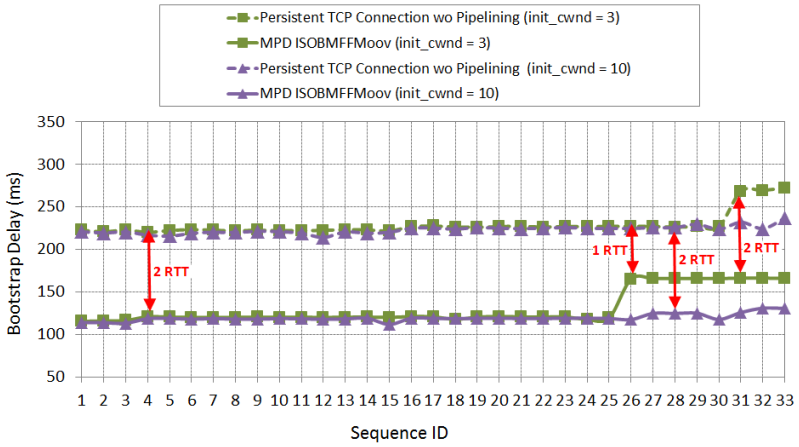


Figure A.4 – Bootstrap delay measured for the ISOBMFFMoov-based approach and persistent TCP connection without pipelining approach over HTTP/1.1.

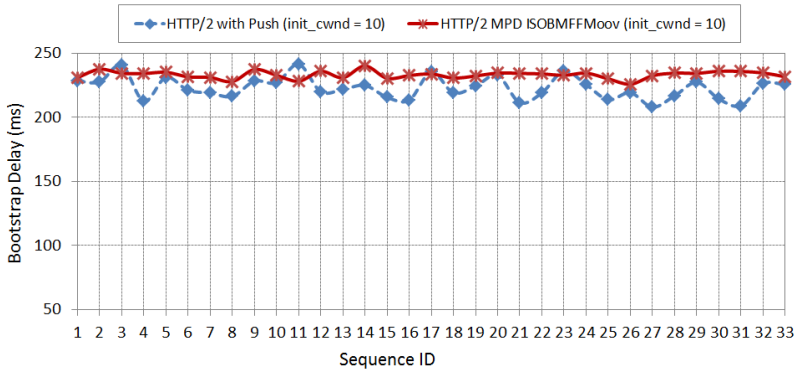


Figure A.5 – Bootstrap delay measured for the ISOBMFFMoov-based approach and persistent TCP connection without pipelining approach using a server push over HTTP/2.

Dans un réseau mobile, nous avons montré un gain de 1 seconde ou plus lors de l'utilisation de notre approche ISOBMFFMoov par rapport à la stratégie de connexion TCP persistante sans pipelining.

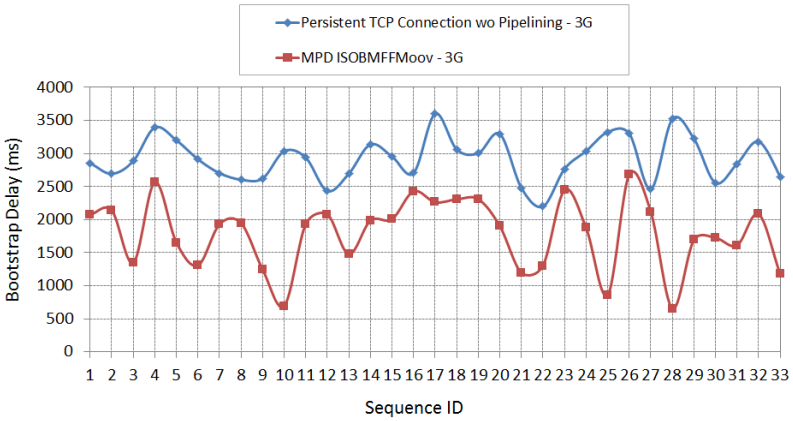


Figure A.6 – Average bootstrap delay measured for the ISOBMFFMoov-based approach and persistent TCP connection without pipelining approach using a 3G mobile network.

A.2.2 Réduction de la Latence en DASH Live

En ce qui concerne cette catégorie, une analyse de la latence des services de streaming en direct utilisant DASH a été présentée. Cette analyse indique que le composant principal induisant une latence supplémentaire par rapport aux autres systèmes de streaming était le processus de segmentation. Notre contribution principale consiste à proposer un nouveau système de streaming en direct à faible latence utilisant DASH.

A.2.2.1 Proposition du Système DASH Live à Faible Latence

Nous avons proposé une nouvelle méthode pour réduire la latence de bout en bout, en particulier le délai de segmentation en DASH live. Le principe de cette solution consiste à utiliser une unité de transfert différente et plus petite que le segment tout en maintenant le segment comme un élément de téléchargement dans les requêtes HTTP, notamment pour le cache.

Pour cela, nous avons utilisé une nouvelle organisation interne de segment média, telle qu'elle est décrite dans la Figure A.7. Chaque segment média est divisé en plusieurs petites parties appelées "movie fragments" de manière à pouvoir être analysé indépendamment. En utilisant le mécanisme "Chunked-Transfer Encoding" de HTTP/1.1, le serveur web peut commencer à pousser ces fragments au client sous la forme de chunks avant la fin de la production du segment. Avec cette solution, l'overhead provient donc de la fragmentation

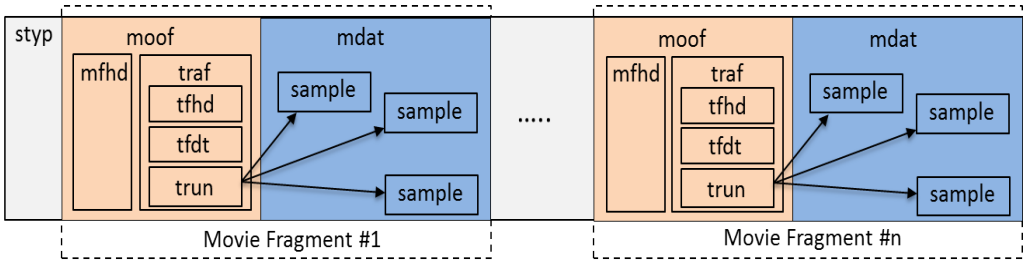


Figure A.7 – Structure of an ISOBMFF media segment with multiple movie fragments.

supplémentaire et du chunking. En plus, la latence peut être réduite à la durée d'un fragment.

Cependant, le client doit faire sa requête pour le segment avant la fin de sa production. Pour résoudre ce problème, on a proposé d'introduire un nouvel attribut "Availability Time Offset" au niveau de la représentation dans le MPD. L'ATO indique le moment où un ou plusieurs fragments sont disponibles au niveau du serveur ce qui permet au client d'envoyer ses requêtes plus tôt. Si l'ATO est choisi pour correspondre au moment où le premier fragment est entièrement produit, la latence de packaging peut être réduite à la durée d'un fragment.

L'availability time offset (ATO) du segment média "i" est exprimé dans l'Équation A.1:

$$ATO_i = AST_i - (AST_i - d_s + d_c) = d_s - d_c \tag{A.1}$$

La Figure A.8 montre la relation entre "AvailabilityStartTime" du segment "i" (AST_i), "availabilityTimeOffset" (ATO_i), la durée du segment (d_s) et la durée du fragment (d_c).

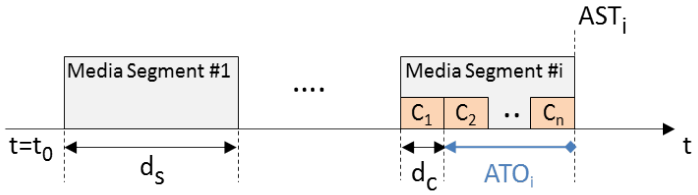


Figure A.8 – Determination of the availability time of a media fragment in DASH.

La procédure que le client utilise pour émettre une requête HTTP pour le segment média le plus récent une fois qu'un fragment ou plus est disponible sur le serveur web est comme suit.

1. Lorsque le client se connecte au flux live, il demande d'abord le MPD du serveur Web. Le MPD peut être généré à la volée ou peut avoir été généré auparavant.

2. Outre la description des représentations et des informations sur les URL des segments, le MPD pour les sessions en direct doit contenir des champs clés supplémentaires tels que le champ *type* défini sur "dynamic" ainsi que le champ *availabilityStartTime*. Nous indiquons aussi dans le MPD quand un fragment ou plus est disponible sur le serveur web à travers l'attribut *availabilityTimeOffset* (ATO).

3. Le client sélectionne ensuite la représentation appropriée en fonction de la qualité/bande passante décrite pour commencer à demander les segments média associés. Lorsque le MPD est dynamique, le client doit déterminer précisément le dernier segment média disponible.

4. Après avoir calculé le numéro du dernier segment média "i" et utilisé la valeur ATO, le client attend que les movie fragments de ce segment soient prêts et effectue la requête.

5. Chaque segment média contient $\#n$ movie fragments. Le serveur est capable d'envoyer les fragments plus tôt, au mieux dès qu'il a été complètement généré, en utilisant des chunks HTTP/1.1.

6. Le client est capable de consommer toute réponse partielle reçue (i.e chunks) avant la réception de toutes les parties. En particulier, chaque chunk (i.e. les boîtes "moof" et "mdat") peut être analysé et mis en file d'attente pour la lecture même si le segment complet n'est pas encore reçu.

A.2.2.2 Expérimentations et Résultats

Afin de valider notre approche, nous avons effectué deux types de mesures.

1. Mesures d'overhead

   L'overhead totale de notre approche peut être décomposé en: overhead introduit par le packaging des trames encodées dans des fragments ISOBMFF et overhead introduit par le téléchargement de ces fragments sur HTTP/1.1 en tant que chunks.

   Le packaging ISOBMFF utilisé dans notre système de streaming DASH introduit un overhead qui peut être aussi décomposé en: un overhead initial dû au packaging des trames média encodés dans le format structuré d'ISO; et un overhead supplémentaire dû à la fragmentation requise dans DASH. Nous avons montré que le processus de fragmentation de DASH est le principal contributeur à l'overhead totale. L'overhead introduit par le simple stockage des trames encodées dans le format d'ISO ainsi que par la distribution de HTTP/1.1 Chunked Encoding sont négligeables.

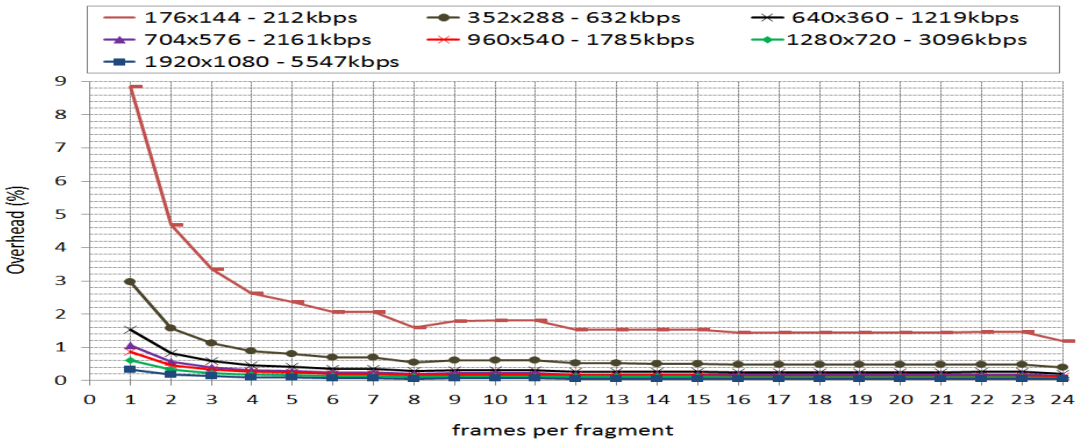


Figure A.9 – Overhead introduced by the ISOBMFF fragmentation.

La Figure A.9 montre les résultats pour la séquence Big Buck Bunny, avec l'overhead calculé par rapport à la séquence non fragmentée. Elle montre que pour 3 trames par fragment, l'overhead est inférieur à 4% pour toutes les résolutions et débits. Plus intéressant, l'overhead maximal qui peut être atteint lors de l'utilisation d'une trame par fragment pour les résolutions classiques (SD Et plus) est 1%, ce qui est inférieur quand on utilise RTP.

2. Mesures de latence

Pour expérimenter l'approche proposée, nous avons conçu et implémenté un système de streaming DASH complet basé sur trois fonctions principales: la préparation du contenu, la distribution du contenu, le décodage et l'affichage du contenu.

Premièrement, nous avons mesuré la latence de la sortie d'encodeur à l'entrée du décodeur comme le montre la Figure A.10.

Avec 25 trames par seconde, le fragment de 200 ms contient 5 trames et le segment de 2 s comprend 50 trames. Avec cette configuration (5 trames par fragment), l'overhead de la fragmentation est de 0,2% comme montré précédemment dans la figure A.9. La latence mesurée est de l'ordre de 160 ms.

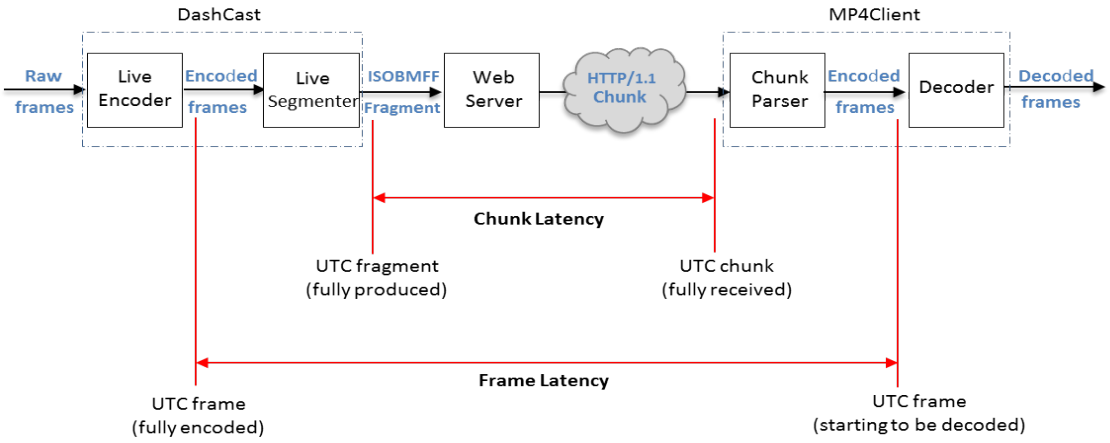


Figure A.10 – Inner-chain latency measurements for live streaming service.

Deuxièmement, nous avons mesuré la latence de la capture à l'affichage comme le montre la Figure A.11.

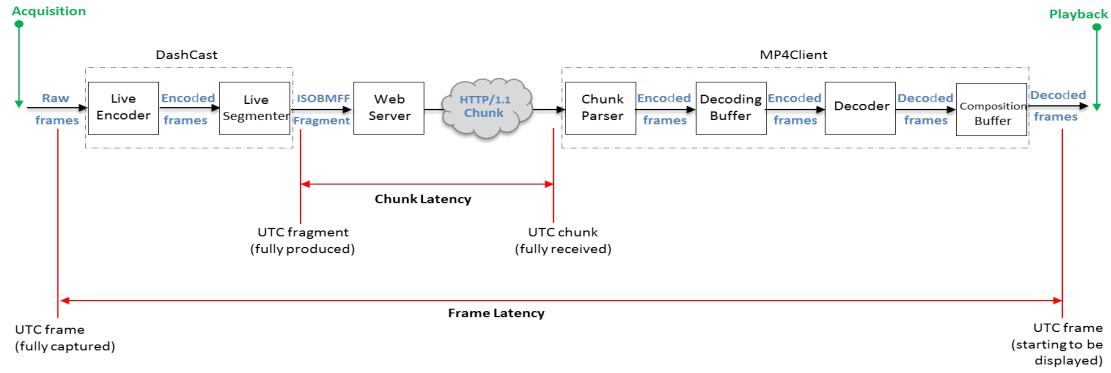


Figure A.11 – End-to-end latency measurements for live streaming service.

La latence totale est de l'ordre de 100 ms pour un fragment contenant une seule trame. Elle est approximativement 225 ms pour un fragment contenant 5 trames.

Grâce à ces résultats, nous avons validé que notre méthode est capable d'obtenir une latence très faible. Dans le cadre de travaux futurs, nous prévoyons d'examiner comment ce système à faible latence se comportera dans des CDN, et d'évaluer davantage notre proposition avec plusieurs players et des phases de switching. En outre, nous prévoyons de mesurer la latence dans les services de streaming interactif en temps réel. Enfin, nous pourrions étudier la méthode qui inverse les boîtes "moof" et "mdat" dans la structure du segment média et vérifier si la latence est encore plus réduite que la méthode typique.

A.2.3 Les applications en DASH Live

La distribution hybride est l'une des applications multimédia qui nécessitent une distribution à faible latence, en particulier sur le réseau broadband. Elle permet de fournir un contenu supplémentaire et personnalisé sur le réseau broadband, synchronisé avec le contenu distribué sur les réseaux broadcast.

La diffusion d'un contenu vidéo sur des réseaux hybrides présentent quelques problèmes tels que :

- L'emplacement du flux broadband

  En diffusion hybride, lorsqu'un client se connecte au flux broadcast, il doit localiser le flux vidéo externe sur le réseau broadband. Par exemple, si DASH est utilisé, on a besoin de l'URL du MPD.

- La corrélation de temps broadcast et broadband

  Vu que les deux réseaux sont indépendants il peut y avoir des manipulations sur les timestamps dans un réseau, et en particulier dans le réseau broadcast.

- La latence totale

  Le réseau broadcast a une latence constante, et relativement plus faible que la latence des solutions de streaming adaptatif sur HTTP dont DASH. Notre but est de fournir un système DASH avec une latence proche ou inférieure à celle de broadcast.

Notre approche proposée assure deux fonctionnalités: la synchronisation des deux contenus fournis par les différents réseaux de distribution et un buffer du client faible sur le réseau broadcast. Notre système s'appuie sur deux outils:

- TEMI qui est un outil de la norme MPEG-2 TS qui a été développé par notre équipe. Cet outil permet de localiser, synchroniser le contenu DASH externe avec le contenu MPEG-2 TS et aussi synchroniser le client et le serveur web à travers une horloge NTP.

- Le système de streaming en direct à faible latence utilisant DASH que nous avons décrit dans les contributions précédentes.

Nous avons étudié un scénario d'utilisation de contenu multi-résolution où nous considérons un contenu de base envoyé sur les réseaux traditionnels de broadcast améliorés avec un contenu supplémentaire envoyé sur les réseaux broadband en utilisant DASH. Notre système de distribution hybride proposé a été implémenté et validé dans les réseaux locaux. Dans le cadre de travaux futurs, nous prévoyons d'examiner comment ce système se comportera

dans des réseaux de distribution de contenu réels, et si les variations de la bande passante du réseau dans le système broadband peuvent retarder le flux DASH par rapport à TS et par conséquence influencer la synchronisation des deux flux et les exigences de buffering sur la chaîne de broadcast.

A.3 Conclusion et Perspectives

A.3.1 Conclusion

Dans cette thèse, nous avons exploré des solutions pour améliorer les services de streaming vidéo en direct. L'objectif de cette thèse était de proposer et de développer de nouvelles solutions pour réaliser un démarrage rapide de la session du streaming en direct utilisant DASH, une distribution de contenu DASH en direct à faible latence, et une diffusion hybride de contenus broadcast et broadband (i.e. contenus DASH et MPEG-2 TS).

Trois principales contributions ont structuré cette thèse:

- La proposition et l'évaluation de différentes méthodes pour réduire le délai de démarrage, en particulier le délai de bootstrap en DASH live.
- Le développement et l'évaluation d'un nouveau système DASH live à faible latence.
- La combinaison synchronisée du contenu broadband et broadcast.

A.3.2 Perspectives

Au moment de la conclusion de ce manuscrit, plusieurs perspectives intéressantes peuvent être proposées pour poursuivre le travail effectué dans cette thèse. Des extensions intéressantes pour améliorer les performances et d'autres axes de recherche dans le contexte de l'adaptation de streaming vidéo sont listées ci-dessous.

- Nous manipulons les flux vidéo en direct à l'échelle de la trame, c'est-à-dire que le packager ISOBMFF est limité à produire des fragments à partir de la trame. Le packaging de petites parties d'une trame peut être nécessaire dans les services de streaming à faible latence.
- Un autre sujet de recherche intéressant pourrait être l'analyse de la façon dont un tel système à faible latence se comportera dans de vrais réseaux de diffusion de contenu. Jusqu'à présent, le client DASH était directement connecté au serveur Web dans un réseau local. Un CDN dispose de mécanismes intelligents de réacheminement, de redirection et de mise en cache qui peuvent influencer la distribution d'un fragment

au client. Les premières expériences montrent que notre système se comporte bien, mais d'autres investigations sont nécessaires.

- Dans les expérimentations actuelles, notre proposition relative à l'amélioration de la phase Bootstrap en DASH live est seulement comparée à la méthode push du serveur HTTP/2. Il pourrait être utile de la comparer à d'autres mécanismes HTTP/2 tels que le multiplexage de requête et de réponse.

- La switching n'a pas fait partie de nos recherches dans cette thèse. Nous pourrions étudier un processus de switching rapide pour notre système DASH Live à faible latence. Dans le streaming adaptatif sur HTTP, un client ne peut switcher qu'à la fin du segment car le switching au milieu du segment nécessite au pire un double téléchargement et un décodage. Un client est contraint d'attendre la fin du segment pour switcher. Ce temps d'attente pour la switching devient encore plus important lorsque des segments média longs sont utilisés. Une solution pourrait consister à utiliser les RAP contenus dans un segment comme points de switching. Ainsi, le client est capable de switcher dans un segment à chaque GoP.

- Étudier l'impact de notre système à faible latence sur d'autres cas d'utilisation récents tels que les applications de réalité virtuelle (VR), en particulier les vidéos 360 VR est intéressant. Le streaming 360 VR adaptatif en direct présente de nombreux défis techniques, notamment la latence de distribution de bout en bout et la latence de switching. La combinaison de notre système à faible latence avec un nouveau processus de switching rapide peut améliorer l'expérience de l'utilisateur.

A.4 Liste des Publications

Articles de Conférence

- N. Bouzakaria, C. Concolato and J.L. Feuvre, "Overhead and performance of low latency live streaming using MPEG-DASH", Proceeding of *The 5th International Conference on Information, Intelligence, Systems and Applications (IISA)* , Crete, Greece, July 2014.

- N. Bouzakaria, C. Concolato and J.L. Feuvre, "Fast dash bootstrap", Proceeding of *IEEE 17th International Workshop on Multimedia Signal Processing (MMSP)* , Xiamen, China, October 2015.

- J.L. Feuvre, C. Concolato, N. Bouzakaria and V. T. Nguyen, "MPEG-DASH for Low Latency and Hybrid Streaming Services", Proceeding of *The 23rd ACM International Conference on Multimedia (ACM MM)* , Brisbane, Australia, October 2015.

Contributions à la Normalisation

- C. Concolato, J. Le Feuvre and N. Bouzakaria, "Data URLs in MPD", *Moving Picture Experts Group (MPEG)*, Geneva, Switzerland, May 2016, n° M38649

- C. Concolato, J. Le Feuvre and N. Bouzakaria, "Guidelines for DASH Fast Start", *Moving Picture Experts Group (MPEG)*, Geneva, Switzerland, October 2015, n° M37254.

- C. Concolato, J. Le Feuvre and N. Bouzakaria, "Use of HTTP/2 Push for DASH Bootstrap", *Moving Picture Experts Group (MPEG)*, Geneva, Switzerland, October 2015, n° M37255.

Bibliography

[1] I. Sodagar, "The mpeg-dash standard for multimedia streaming over the internet," *IEEE MultiMedia*, vol. 18, no. 4, pp. 62–67, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1109/MMUL.2011.71 *Cited in Sec.* (document), 1.1, 2.4.1, 2.7, A.1

[2] *High performance browser networking.* I. Grigorik, May 2013. *Cited in Sec.* (document), 3.4, 3, 4, 5

[3] Cisco, "Cisco Visual Networking Index: Forecast and Methodology, 2014–2019," Cisco, Tech. Rep., 2015. [Online]. Available: http://www.cisco.com/ *Cited in Sec.* 1.1

[4] Conviva, "Viewer experience report," Tech. Rep., 2013. *Cited in Sec.* 1.1, 2.2.4, 3.1, 3.4.3.3, A.1

[5] G. Almes, S. Kalidindi, and M. J. Zekauskas, "A one-way delay metric for IPPM," Internet Requests for Comment, RFC Editor, Fremont, CA, USA, RFC 2679, Sep. 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2679.txt *Cited in Sec.* 2.2

[6] ——, "A one-way packet loss metric for IPPM," Internet Requests for Comment, RFC Editor, Fremont, CA, USA, RFC 2680, Sep. 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2680.txt *Cited in Sec.* 2.2

[7] C. Demichelis and P. Chimento, "IP packet delay variation metric for IP performance metrics (IPPM)," Internet Requests for Comments, RFC 3393, Fremont, CA, USA, RFC 3393, Nov. 2002. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3393.txt *Cited in Sec.* 2.2

[8] S. M. Patrick Le Callet and e. Andrew Perkis, ""qualinet white paper on definitions of quality of experience (2012)," European Network on Quality of Experience in Multimedia Systems and Services (COST Action IC 1003), Lausanne, Switzerland, Version 1.1, Tech. Rep., June 3 2012. *Cited in Sec.* 2.2

[9] D. V. E. DVEO, "Introduction to cdn and vod principles overview (rev 1.7)," Tech. Rep., November 2015. *Cited in Sec.* 2.2.1

[10] J. Abreu, J. 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," *Telecommunication Systems*, pp. 1–18, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11235-016-0157-3 *Cited in Sec.* 2.2.1

[11] O. B. Maia, H. C. Yehia, and L. de Errico, "A concise review of the quality of experience assessment for video streaming," *Computer Communications*, vol. 57, pp. 1–12, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2014.11.005 *Cited in Sec.* 2.2.3

[12] International Telecommunication Union, "ITU-T Recommendation P.800.2: Mean opinion score interpretation and reporting," Tech. Rep., 2013. *Cited in Sec.* 2.2.3

[13] F. Kuipers, R. Kooij, D. De Vleeschauwer, and K. Brunnström, "Techniques for measuring quality of experience," in *Proceedings of the 8th International Conference on Wired/Wireless Internet Communications*, ser. WWIC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 216–227. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13315-2_18 *Cited in Sec.* 2.2.3

[14] M. Mu, P. Romaniak, A. Mauthe, M. Leszczuk, L. Janowski, and E. Cerqueira, "Framework for the integrated video quality assessment," *Multimedia Tools and Applications*, vol. 61, no. 3, pp. 787–817, 2012. [Online]. Available: http://dx.doi.org/10.1007/s11042-011-0946-3 *Cited in Sec.* 2.2.3

[15] S. Krishnan and R. Sitaraman, "Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs," *Networking, IEEE/ACM Transactions on*, vol. 21, no. 6, pp. 2001–2014, Dec 2013. *Cited in Sec.* 2.2.4

[16] A. Finamore, M. Mellia, M. M. Munafò, R. Torres, and S. G. Rao, "Youtube everywhere: Impact of device and infrastructure synergies on user experience," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 345–360. [Online]. Available: http://doi.acm.org/10.1145/2068816.2068849 *Cited in Sec.* 2.2.4

[17] L. Chen, Y. Zhou, and D. M. Chiu, "Video browsing - a study of user behavior in online vod services," in *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, July 2013, pp. 1–7. *Cited in Sec.* 2.2.4

[18] T. Hossfeld, S. Egger, R. Schatz, M. Fiedler, K. Masuch, and C. Lorentzen, "Initial Delay Vs. Interruptions: Between The Devil And The Deep Blue Sea," in *Proc. QoMEX (Quality of the Multimedia Experience) 2012, Yarra Valley, Australia*, Jul. 2012. *Cited in Sec.* 2.2.4

[19] F. Dobrian, A. Awan, D. A. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang, "Understanding the impact of video quality on user engagement," *Commun. ACM*, vol. 56, no. 3, pp. 91–99, 2013. [Online]. Available: http://doi.acm.org/10.1145/2428556.2428577 *Cited in Sec.* 2.2.4

[20] T. Lohmar, T. Einarsson, P. Frojdh, F. Gabin, and M. Kampmann, "Dynamic adaptive http streaming of live content," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a*, June 2011, pp. 1–8. *Cited in Sec.* 2.3.1, 4.1, 4.2, 4.4.2.1

[21] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the h.264/avc video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 560–576, July 2003. *Cited in Sec.* 2.3.2.1

[22] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the high efficiency video coding (hevc) standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1668, Dec 2012. *Cited in Sec.* 2.3.2.1

[23] A. Leontaris and P. C. Cosman, "End-to-end delay for hierarchical b-pictures and pulsed quality dual frame video coders," in *Proceedings of the International Conference on Image Processing, ICIP 2006, October 8-11, Atlanta, Georgia, USA*, 2006, pp. 3133–3136. [Online]. Available: http://dx.doi.org/10.1109/ICIP.2006.312937 *Cited in Sec.* 2.3.2.1

[24] G. T. 26.244, ""transparent end-to-end packet switched streaming service (pss), 3gpp file format (3gp)"." *Cited in Sec.* 3

[25] L. Zhang, L. Zheng, and K. S. Ngee, "Effect of delay and delay jitter on voice/video over IP," *Computer Communications*, vol. 25, no. 9, pp. 863–873, 2002. [Online]. Available: http://dx.doi.org/10.1016/S0140-3664(01)00418-2 *Cited in Sec.* 2.3.2.4

[26] A. C. 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, 2011. *Cited in Sec.* 2.4

[27] ——, "Watching video over the web: Part 2: Applications, standardization, and open issues," *IEEE Internet Computing*, vol. 15, no. 3, pp. 59–63, 2011. *Cited in Sec.* 2.4

[28] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran, "Probe and adapt: Rate adaptation for HTTP video streaming at scale," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 4, pp. 719–733, 2014. [Online]. Available: http://dx.doi.org/10.1109/JSAC.2014.140405 *Cited in Sec.* 2.4

[29] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 97–108. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413189 *Cited in Sec.* 2.4

[30] A. Beben, P. Wisniewski, J. M. Batalla, and P. Krawiec, "ABMA+: lightweight and efficient algorithm for HTTP adaptive streaming," in *Proceedings of the 7th International Conference on Multimedia Systems, MMSys 2016, Klagenfurt, Austria, May 10-13, 2016*, 2016, pp. 2:1–2:11. [Online]. Available: http://doi.acm.org/10.1145/2910017.2910596 *Cited in Sec.* 2.4

[31] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, "A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service," in *Proc. ACM SIGCOMM*, Aug. 2014. *Cited in Sec.* 2.4

[32] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, "BOLA: Near-optimal bitrate adaptation for online videos," in *IEEE INFOCOM.*, Apr. 2016. *Cited in Sec.* 2.4

[33] T. Karagkioules, C. Concolato, D. Tsilimantos, and S. Valentin, "A comparative case study of http adaptive streaming algorithms in mobile networks," in *2017 NOSSDAV*, June 2017, pp. 1–6. *Cited in Sec.* 2.4

[34] S. Akhshabi, L. Anantakrishnan, A. C. Begen, and C. Dovrolis, "What happens when http adaptive streaming players compete for bandwidth?" in *Proceedings of the 22Nd International Workshop on Network and Operating System Support for Digital Audio and Video*, ser. NOSSDAV '12. New York, NY, USA: ACM, 2012, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/2229087.2229092 *Cited in Sec.* 2.4

[35] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari, "Confused, timid, and unstable: Picking a video streaming rate is hard," in *Proceedings of the 2012 Internet Measurement Conference*, ser. IMC '12. New York, NY, USA: ACM, 2012, pp. 225–238. [Online]. Available: http://doi.acm.org/10.1145/2398776.2398800 *Cited in Sec.* 2.4

[36] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen, "Server-based traffic shaping for stabilizing oscillating adaptive streaming players," in *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, ser. NOSSDAV '13. New York, NY, USA: ACM, 2013, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/2460782.2460786 *Cited in Sec.* 2.4

[37] R. Houdaille and S. Gouache, "Shaping http adaptive streams for a better user experience," in *Proceedings of the 3rd Multimedia Systems Conference*, ser. MMSys '12. New York, NY, USA: ACM, 2012, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/2155555.2155557 *Cited in Sec.* 2.4

[38] G. Tian and Y. Liu, "Towards agile and smooth video adaptation in dynamic http streaming," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413190 *Cited in Sec.* 2.4

[39] Microsoft. (2010) Iis smooth streaming technical overview. [Online]. Available: http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=17678 *Cited in Sec.* 2.4.2

[40] A. Fecheyr-Lippens, "A review of http live streaming," Tech. Rep., Jan. 2010. [Online]. Available: http://issuu.com/andruby/docs/http_live_streaming?viewMode=magazine&mode=embed *Cited in Sec.* 2.4.2, 4.2.2

[41] S. Bae, D. Jang, and K. Park, "Why is HTTP adaptive streaming so hard?" in *Proceedings of the 6th Asia-Pacific Workshop on Systems, APSys 2015, Tokyo, Japan, July 27-28, 2015*, 2015, pp. 12:1–12:8. [Online]. Available: http://doi.acm.org/10.1145/2797022.2797031 *Cited in Sec.* 2.4.2

[42] "Comparing adaptive http streaming technologies," rgb NETWORKS, Tech. Rep., 2011. *Cited in Sec.* 2.4.2

[43] C.-M. Huang and T.-H. Hsu, "A user-aware prefetching mechanism for video streaming," *World Wide Web*, vol. 6, no. 4, pp. 353–374, Dec. 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1025661921237 *Cited in Sec.* 3.1

[44] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, "Tcp fast open," in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '11. New York, NY, USA: ACM, 2011, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/2079296.2079317 *Cited in Sec.* 3.2.1

[45] Y. Cheng, J. Chu, A. Jain, and S. Radhakrishnan, "TCP Fast Open," RFC 7413, Dec. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7413.txt *Cited in Sec.* 3.2.1

[46] M. Allman, S. Floyd, and C. Partridge, "Increasing TCP's Initial Window," Oct. 2002, rFC 3390. *Cited in Sec.* 3.2.1

[47] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing tcp's initial congestion window," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 3, pp. 26–33, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1823844.1823848 *Cited in Sec.* 3.2.1

[48] N. Cardwell, S. Savage, and T. Anderson, "Modeling tcp latency," in *in IEEE INFOCOM*, 2000, pp. 1724–1751. *Cited in Sec.* 3.2.2

[49] M. Allman, V. Paxson, and W. Stevens, "RFC 2581 (rfc2581) - TCP Congestion Control," Tech. Rep. 2581, 1999. [Online]. Available: http://www.faqs.org/rfcs/rfc2581.html *Cited in Sec.* 2

[50] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540, May 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7540.txt *Cited in Sec.* 5

[51] *ISO/IEC FDIS 23000-19, Common media application format (CMAF)*, Std. *Cited in Sec.* 3b

[52] S. Hemminger *et al.*, "Network emulation with netem," in *Linux conf au*, 2005, pp. 18–23. *Cited in Sec.* 3.4.1.1

[53] B. Hubert, T. Graf, G. Maxwell, R. Van Mook, M. Van Oosterhout, P. B. Schroeder, J. Spaans, and P. Larroy, *Linux Advanced Routing & Traffic Control HOWTO*, Linux Advanced Routing & Traffic Control, Apr. 2004. *Cited in Sec.* 3.4.1.1

[54] N. Bouzakaria, C. Concolato, and J. L. Feuvre, "Fast dash bootstrap," in *2015 IEEE 17th International Workshop on Multimedia Signal Processing (MMSP)*, Oct 2015, pp. 1–6. *Cited in Sec.* 3.5

[55] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616, hypertext transfer protocol – http/1.1," 1999. [Online]. Available: http://www.rfc.net/rfc2616.html *Cited in Sec.* 4.2.3

[56] Y. F. Alex MacAulay, Boris Felts, "Ip streaming of mpeg-4:native rtp vs mpeg-2 transport stream," Envivio, Tech. Rep., October 2005. *Cited in Sec.* 4.4.2.1

[57] I. Kofler, R. Kuschnig, and H. Hellwagner, "Implications of the ISO base media file format on adaptive HTTP streaming of H.264/SVC," in *2012 IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, NV, USA, January 14-17, 2012*, 2012, pp. 549–553. [Online]. Available: http://dx.doi.org/10.1109/CCNC.2012.6180986 *Cited in Sec.* 4.4.2.1

[58] S. Aoki, K. Aoki, H. Hamada, Y. Kanatsugu, M. Yamamoto, and K. Aizawa, "A new transport scheme for hybrid delivery of content over broadcast and broadband," in *2011 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, June 2011, pp. 1–6. *Cited in Sec.* 5.1

[59] C. Concolato, S. Thomas, R. Bouqueau, and J. L. Feuvre, "Synchronized delivery of multimedia content over uncoordinated broadcast broadband networks," in *Proceedings of the Third Annual ACM SIGMM Conference on Multimedia Systems, MMSys 2012, Chapel Hill, NC, USA, February 22-24, 2012*, 2012, pp. 227–232. [Online]. Available: http://doi.acm.org/10.1145/2155555.2155590 *Cited in Sec.* 5.2

[60] A. Baba, K. Matsumura, S. Mitsuya, M. Takechi, H. Fujisawa, H. Hamada, S. Sunasaki, and H. Katoh, "Seamless, synchronous, and supportive: Welcome to hybridcast: An advanced hybrid broadcast and broadband system," *IEEE Consumer Electronics Magazine*, vol. 1, no. 2, pp. 43–52, April 2012. *Cited in Sec.* 5.2

[61] L. B. Yuste, F. Boronat, M. Montagud, and H. Melvin, "Understanding Timelines Within MPEG Standards," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 368–400, 2016. *Cited in Sec.* 5.2

[62] "Iso/iec 13818-1, temi,." [Online]. Available: http://mpeg.chiariglione.org/standards/mpeg-2/systems/text-isoiec-13818-12013dam-6-delivery-timeline-external-data. *Cited in Sec.* 5.3.1

[63] J. Le Feuvre, C. Concolato, N. Bouzakaria, and V.-T.-T. Nguyen, "Mpeg-dash for low latency and hybrid streaming services," in *Proceedings of the 23rd ACM International Conference on Multimedia*, ser. MM '15. New York, NY, USA: ACM, 2015, pp. 751–752. [Online]. Available: http://doi.acm.org/10.1145/2733373.2807977 *Cited in Sec.* 5.5

[64] M. M. Hannuksela, Y.-K. Wang, and M. Gabbouj, "Random access using isolated regions," in *Proceedings 2003 International Conference on Image Processing (Cat. No.03CH37429)*, vol. 3, Sept 2003, pp. III–841–4 vol.2. *Cited in Sec.*

[65] G. BJONTEGAARD, "Calculation of average psnr differences between rd-curves," *ITU SG16 Doc. VCEG-M33*, 2001. [Online]. Available: http://ci.nii.ac.jp/naid/10029505309/en/ *Cited in Sec.*