Chapter 1

Dynamic Adaptive Streaming over HTTP

We refer to Dynamic Adaptive Streaming over HTTP (DASH) as a video streaming protocol which is capable of dynamically adapt to the conditions of the system in order to provide seamless video playback. Several streaming protocols, both public and proprietary, are being developed providing an infrastructure that allows the server to make media available with different resolutions and the client to select the one that most fits the channel and device capabilities while the stream is in playback. In this chapter we illustrate the core of the DASH protocols and the main design principles. The main adaptivity algorithms are also presented.

1.1 Introduction

According to Cisco Visual Networking Index, in 2016 video accounted for 60% of the mobile data traffic and it is forecasted to account for 78% by 2021 (more than 38 Exabytes per month), as shown in Figure 1.1 [1]. Alongside the growth in popularity, the

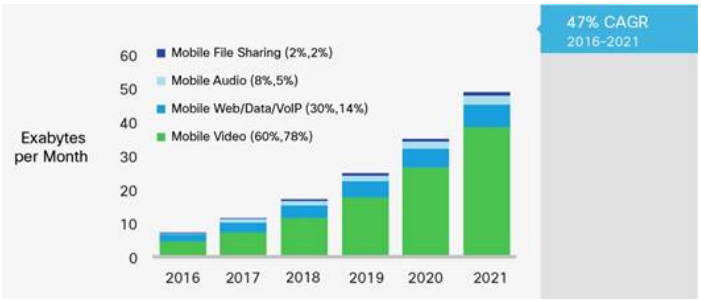

Figure 1.1: Mobile data forecast for 2021. Source: Cisco VNI Mobile, 2017

user expectations on the quality of service are increasing [2]. To fulfill both the growth in servers load and user quality expectation, advanced technology is needed. One of these technologies is Dynamic Adaptive Streaming over HTTP (DASH), which is a protocol that has been standardized over the past eight years by MPEG, 3GPP, Open IPTV Forum, Apple, Adobe and Microsoft. [3–7] The basic idea is to adapt dynamically the quality of the stream according to network and device conditions in order to have a smooth and seamless playback [2, 7, 8].

1.2 Different Standards

DASH is a MPEG protocol. In fact it is also referred to as MPEG-DASH. It is a ISO/IEC standard from 2012 (ISO/IEC 23009-1) [7]. Latest revision is of year 2014 [4]. The idea is based on 3GPP Adaptive HTTP Streaming (AHS) release 9 and Open IPTV Forum release 2 [7]. Latest 3GPP release (version 14, 2017) is maintained by the Technical Specification Group (TSG) SA4 (Services and System Aspects) and their streaming protocol is within the general name of Transparent end-to-end Packet-switched Streaming Service (PSS) (TS 26.233, TS 26.234, TS 26.244, TS 26. 247) [5, 9–11]. In particular the last one includes specifications about DASH. Apple's HTTP Live Streaming protocol is still in draft. The first version was published in 2009 [3] and as today it has arrived at version 23 [3]. It doesn't refer to the term DASH but the basic idea is still to let the client adapt the bitrate to the current network condition in order to provide a seamless playback. Another alternative of the same idea is provided by the Microsoft Smooth Streaming Protocol which first revision is of 2010 [6] and the latest is of January 2017 [6]. It is based on the Real Time Streaming Protocol (RFC 2326), which is a stateful protocol, although it makes a better use of HTTP caches and provides adaptable streaming quality through the use of MPEG-4 -based data structure [6].

1.3 The MPEG-DASH infrastructure

All these different protocols use different names but the core structure is the same. First of all they all use HTTP as the underlying content delivery system, HTTP/1.1 in particular. We'll talk about the advantages of using HTTP later. They all use a Media Presentation Description (which e.g. corresponds to to a Playlist in Apple's HLS [3]) and Segments (which e.g. are called Fragments in Microsoft's SSP [6]). We present here the MPEG protocol basics.

1.3.1 The data model

MPEG-DASH [4] defines two basic elements and their format. The first is a Media Presentation Description (MPD) which is a XML formatted file which describes the Media Presentation, which is a bounded or unbounded list of HTTP URLs to the Segments which compose the video to be streamed. The latter is the Segment, which is the content response of an HTTP GET or partial HTTP GET (as specified in RFC 2616: HTTP/1.1 [12]). A Segment can contain both media data and metadata useful for segments alignment, synchronization, media navigation an other features. The client first requests the MPD and then progressively requests the Segments to be played back, until the broadcast ends or the user interrupts the process. This precess is illustrated in Figure 1.2. The MPD is periodically refreshed to update the list of URLs.

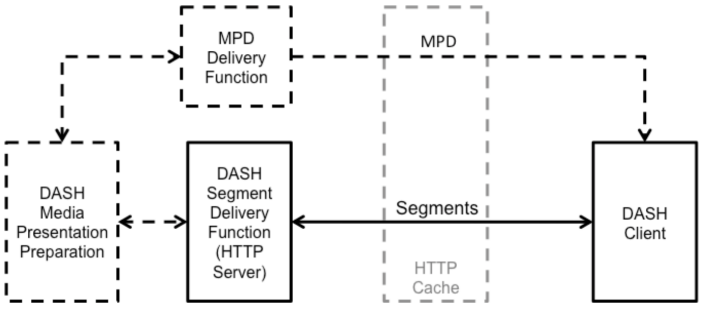

Figure 1.2: Streaming schema for MPEG-DASH protocol. Source: [4]

The MPD structure is illustrated in Figure 1.3. In particular the MPD describes the sequence of Periods, which are media content

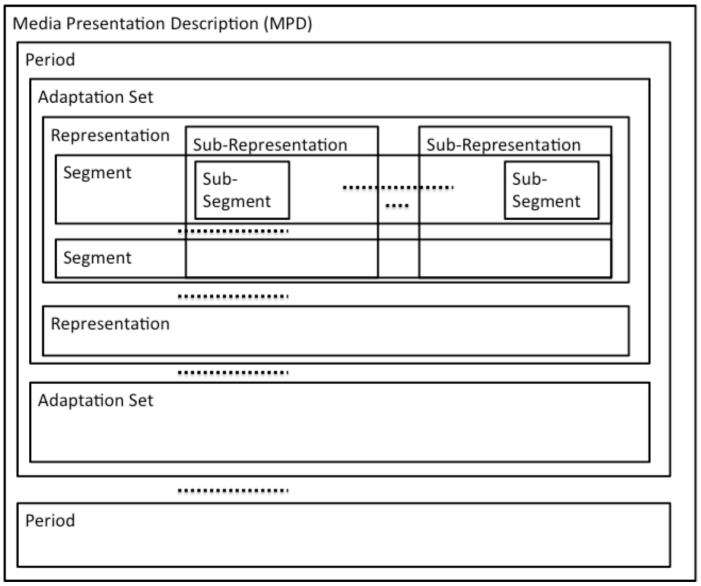

Figure 1.3: The structure of a Media Presentation Description (MPD). Source: [4]

periods during which the encoded versions of the media are consistent. The content within a period is arranged in an Adaptation Set, which contains different Representations of the media. Representations are deliverable encoded versions of one or several media content components, such as different resolutions, subtitles, languages and captions. Representations are then divided into Segments, which are the largest media units downloadable with an HTTP request, as mentioned before. Segments have more or less the same duration (which is not specified in the protocol, in contrast with Microsoft and Apple's solutions). Periods can vary for different representations, which can be a parameter for network adaptation [4], although the adaptivity is declared to be only at

client side [4]. In particular Segment duration is a lower bound for end-to-end latency, so it is convenient to set a short Segment duration in case of low latency requirement, such us live streams [4]. An important feature is the possibility to divide Segments into Subsegments. This allows for Segment Indexes, which are pointers (or better byte ranges) to other Subsegments [2, 4] (Figure 1.4). This allows the client to perform a byte range request through a partial HTTP GET request [12] to first download the Index, than to selectively request for specific subsegments [2, 4]. This allows for navigation through the media [2]. A few examples of Segment Indexing can be found in [2]. Clients may decide to jump from one representation to another. This is made easier

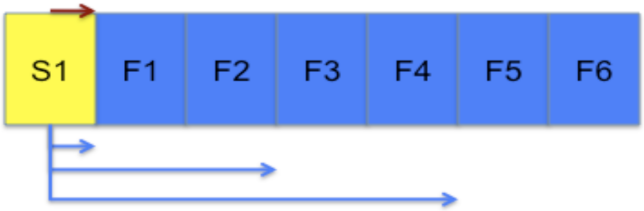

Figure 1.4: Index and media Subsegments. An Index Subsegment points to the first media Subsegment and to other Subsegments. Source: [2]

through Stream Access Points, usually located at the beginning of Segments or Subsegments, where the different Representations are aligned, in order to play back subsequent media segments with continuous timestamps, without audio or video glitches and without downloading overlapping media content [2, 4].

1.3.2 The DASH client model

MPEG-DASH [4] considers a DASH client model logically composed of three parts, as shown in Figure 1.5. The first part is the DASH access engine, which can request and receive the Media Presentation Description and the Segments (or part of them) from an HTTP server and handle them as described in the protocol. It then passes to the Media engine module the media in MPEG format plus timing information that maps the media internal timestamps to the presentation timestamps. The access engine also receives and extracts Events which can be handled by the access engine itself or handed off to a different Application module.

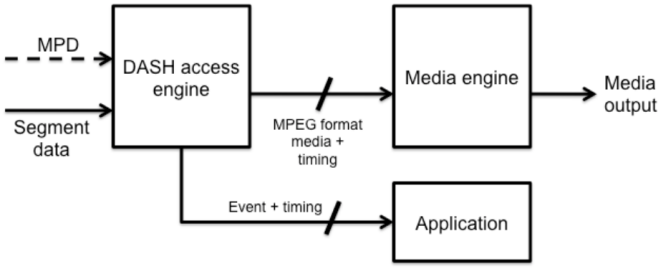

Figure 1.5: MPEG-DASH Client Model: [4]

1.4 Why HTTP is a good idea

The main solution for streaming before DASH was widely adopted was the Real Time Streaming Protocol. It was a stateful protocol in which the client established a connection with a server and the latter kept the state of the client, sending it a continuous stream of packets through UDP or TCP until it disconnected [2]. HTTP is a stateless protocol in which requests and responses of some data are considered as a complete stand-alone transaction, as shown in Figure 1.6. Through HTTP the streaming session is completely handled by the client, avoiding an overhead from the frequent communication between the client and the server to maintain a stateful connection (only one or more simple TCP connection are held) [2, 12]. Another alternative is HTTP progressive download through byte ranges requests [2]. The disadvantages of this are that the whole content must be ready in advance, therefore it isn't suitable for live streaming. Moreover bandwidth might be wasted if the user decides to stop watching the stream. Last and most important, it isn't bitrate adaptive [2]. For these reasons HTTP progressive download wasn't enough for the new streaming technologies requirements. However the preexisting Content Delivery Networks based on HTTP were widely developed and efficient and could be exploited. An example of a general media distribution architecture is in Figure 1.7) , where a video is taken from a source and a Media Preparation server encodes it with different bitrates, than splits the different versions into segments according to the protocol. Than the segments are sent with the Media Presentation Description to several HTTP Servers. The client

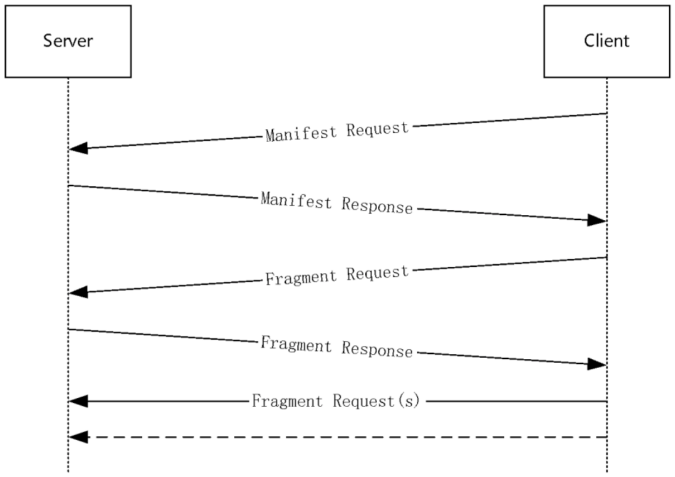

Figure 1.6: Communication between client and server in the DASH protocol through HTTP. Source: [6]

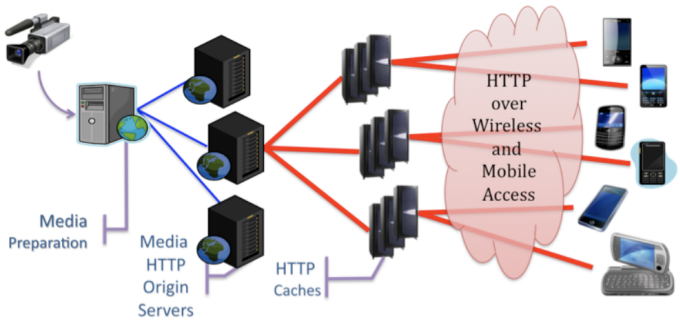

Figure 1.7: HTTP media distribution architecture with HTTP caches. Source: [2]

requests the segments according to some adaptation algorithm, taking in consideration various elements such as network condition, available bandwidth, device conditions or user preferences. The HTTP response can be speeded-up thanks to HTTP cache servers, as illustrated in Figure 1.8 [2, 6, 8]. Not only the HTTP transaction is quicker with CDNs but the massive amount of connections to be handled are offloaded from the central servers and distributed to the peripheral servers. For these advantages provided by HTTP CDNs, they were chosen as a base for Dynamic Adaptive Streaming [2]. Another reason for choosing HTTP is that it is the most used protocol for content delivery and communication over World Wide Web and so the user doesn't need to have particular skills to set up the firewall properly in order to receive the stream, since most firewalls let the traffic pass through the standard HTTP port [2, 8]. Other advantages of using HTTP are that the client can decide the initial streaming quality based on the currant bandwidth without having to negotiate with the server [2] (stateless connection), as well as the ability to decide to switch stream quality independently from the server [2].

1.5 Adaptive Bit Rate algorithms

In this section we describe some algorithms that can be used to adapt the bit rate in order to better exploit the available bandwidth. In general, these algorithms are called Adaptive Bit Rate (ABR) algorithms, and they are designed with three goals in mind [13]:

1. maximize efficiency: stream at the highest possible bit rate;
2. minimize re-buffering: avoid playback freezing;
3. stability: switch bit rate only when necessary.

The first goal is meant to serve the user with the best video quality possible: to meet this requirement, the client will choose the highest bit rate that the bandwidth, in conjunction with the buffer, can sustain. However the channel is not the only criterion by which the client will choose the bit rate: also the screen size of the device, the size of the player video and the CPU come into play in order to avoid waste of resources. The second and the third goal are meant to offer the user the best experience possible: fast

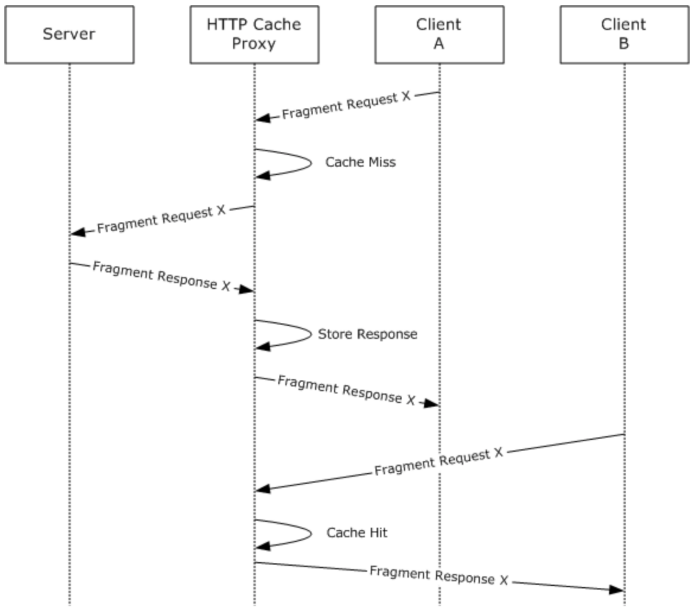

Figure 1.8: Efficient content delivery with the use of HTTP caches. Source: [6]

start-up time, avoid playback freezing events and refrain to switch bit rate if it is not strictly required in order to provide smooth streaming [14].

In literature we can find many algorithms and implementations that work either on the client side or on the server side. We here focus our attention on the former and we see two of them.

1.5.1 Segment Aware Rate Adaptation

Segment Aware Rate Adaptation (SARA) is an ABR algorithm that decides the bit rate to be selected for each segment of the video depending not only on the current network condition and the buffer occupancy but also on the size of the segment to be downloaded.

As described in section 1.3, each video is encoded with different selection of resolutions and bit rates and then it is divided in multiple segments. The server collects all this information in the MPD file. Since each segment is usually of a fixed playback duration but differs greatly on size (see figure 1.9), some algorithms proposed in literature may fail to predict the time to download the next segment [15]. SARA instead proposes to enhance the MPD file with the size of each segment and then estimates the throughput based also on this information.

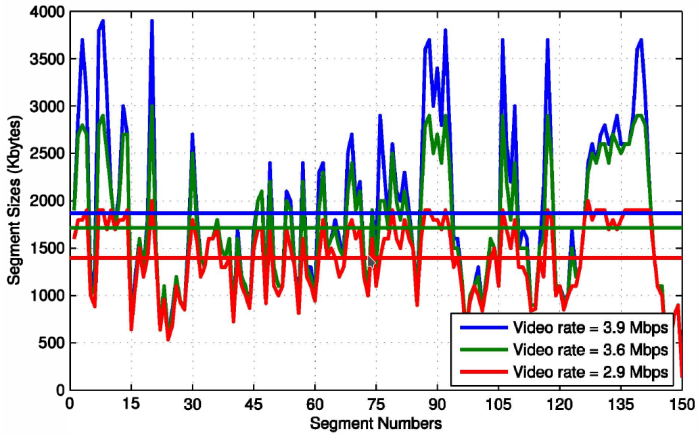

Figure 1.9: Variation in segment sizes for *The Big Buck Bunny Movie* [16]. Each segment lasts for 4 seconds and it is encoded with three different bit rates. Source: [15]

In order for the rate adaptation algorithm to pick the appropriate bit rate, it is necessary to measure the throughput during the entire video session: to avoid the effect of instantaneous throughput variation we use harmonic mean [15]. For a generic segment i

we assign a weight w_i that is proportional to the segment size. Then, we take into account the download rates of the segments, call them $d_1, d_2, \cdots$. The weighted harmonic mean download rate for n downloaded segments is then

$$H_n = \frac{\sum_{i=1}^{n} w_i}{\sum_{i=1}^{n} \frac{w_i}{d_i}}$$

Based on this value, we can predict the time to download the next segment as w_{n+1}/H_n.

So, from the set of all representations, SARA selects to download the most suitable one. Every segment that is downloaded is stored on a buffer B. This buffer have a maximum capacity B_{max}, a current capacity B_{curr} and three other thresholds I, B_α and B_β. The buffer is pictured in 1.10 and all of the thresholds are defined in terms of the number of segments.

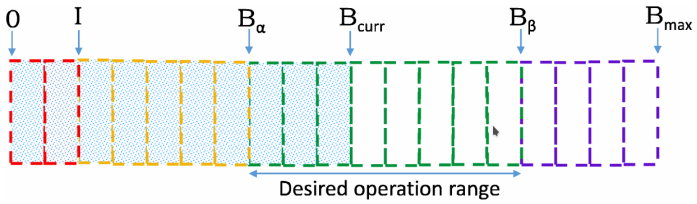

Figure 1.10: Buffer and the thresholds. Source: [15]

The algorithm works based on the current buffer capacity and the predicted next segment download time. It works in four stages:

- **Fast Start** ($B_{curr} \leq I$): the algorithm selects the lowest bit rate. This ensure that the start up time of the playback is kept as small as possible. This is important to prevent the user to abandon the video session [14].

- **Additive Increase** ($I < B_{curr} \leq B_\alpha$): in this stage we start to increase the bit rate. This is done in small steps to avoid the buffer to go back under I.

- **Aggressive Switching** ($B_\alpha < B_{curr} \leq B_\beta$): now, based on the network condition and the buffer occupancy, we start to select the most suitable bit rate that is greater than or equal to the current one.

- **Delayed Download** ($B_\beta < B_{curr} \leq B_{max}$): in this stage, the most suitable bit rate for the current network condition is selected, however the request is delayed until the buffer occupancy goes under B_β. This ensures two things: it prevents buffer overflows and it avoids unnecessary downloads of segments in case the user stops the playback.

The algorithm is listed in 1. The algorithm is called after every segment downloaded to determine the representation to download for the next segment.

At the start of the video session or after a re-buffering event, the number of segments in the buffer is below I and so the algorithm works in the *Fast Start* phase: the lowest bit rate is selected. When the capacity of the buffer is beyond I, the algorithm switches to the *Additive Increase* phase.

The time required to download the next segment is given by w_{n+1}^{curr}/H_n. In general, any time during the playback of the video, if the download time of the next segment is greater than $B_{curr} - I$, it is not feasible to download the next segment of the current bit rate before the buffer goes below I: the best possible bit rate is chosen among the ones that can be downloaded in the duration $B_{curr} - I$, given the next segment size and the current weighted Harmonic mean download rate. SARA uses a conservative approach and tries to keep the buffer above the I threshold; if it is not possible, it switches to the lowest bit rate possible.

In the *Additive Increase* phase the algorithm starts increasing the bit rate of the video, if possible. The increase in quality is done step by step: if the current representation has a bit rate r^{curr}, the next one will have a bit rate that is the minimum bit rate, among the ones available, that is greater than r^{curr}. This approach ensures that the quality is increased gradually.

After the buffer capacity goes above B_α the increase in quality is more aggressive and the algorithm selects the representation with the highest bit rate that can be downloaded before the buffer capacity goes under I. During the *Aggressive Switching* phase, the request is sent immediately, but when it enter the *Delayed Download* phase ($B_{curr} > B_\beta$), the request is sent only when the buffer goes below B_β, thus avoiding buffer overflows or unnecessary data fetch in case the user stops the video session prematurely.

Algorithm 1 Segment Aware Rate Adaptation algorithm

Input: $\mathfrak{R}$: set of the available bit rates $\{r^{min}, \cdots, r^i, \cdots, r^{max}\}$
I, B_α, B_β: buffer thresholds (constants)
n: segment number of the most recent download
r^{curr}: bit rate of the most recent downloaded segment
B_{curr}: current buffer occupancy in seconds
$W_{n+1} = \{w^{min}_{n+1}, \cdots, w^i_{n+1}, \cdots, w^{max}_{n+1}\}$ the sizes of the segments of bit rate $\{r^{min}, \cdots, r^i, \cdots, r^{max}\}$
H_n: weighted Harmonic mean download rate for the first n segments
Output: l_{n+1}: the bit rate of the next segment to be downloaded
δ: the wait time before download the next segment

if $B_{curr} \leq I$ **then**
  $l_{n+1} = r^{min}$;
else
  if $\frac{w^{curr}_{n+1}}{H_n} > B_{curr} - I$ **then**
    $l_{n+1} = \max\{r^i \mid r^i \in \mathfrak{R},\ \frac{w^i_{n+1}}{H_n} \leq B_{curr} - I,\ i \leq curr\}$;
    $\delta = 0$;
  else if $B_{curr} \leq B_\alpha$ **then**
    if $\frac{w^{curr+1}_{n+1}}{H_n} < B_{curr} - I$ **then**
      $l_{n+1} = \min\{r^i \mid r^i \in \mathfrak{R},\ r^i > r^{curr},\ i \geq curr\}$;
    else
      $l_{n+1} = r^{curr}$;
    end if
    $\delta = 0$;
  else if $B_{curr} \leq B_\beta$ **then**
    $l_{n+1} = \max\{r^i \mid r^i \in \mathfrak{R},\ \frac{w^i_{n+1}}{H_n} \leq B_{curr} - I,\ i \geq curr\}$;
    $\delta = 0$;
  else if $B_{curr} > B_\beta$ **then**
    $l_{n+1} = \max\{r^i \mid r^i \in \mathfrak{R},\ \frac{w^i_{n+1}}{H_n} \leq B_{curr} - B_\alpha,\ i \geq curr\}$;
    $\delta = B_{curr} - B_\beta$;
  else
    $l_{n+1} = r^{curr}$;
    $\delta = 0$;
  end if
end if

Performance How well does SARA perform? In [15] the proposed algorithm is compared with a very basic adaptive algorithm: it uses the average segment download rate and the current buffer occupancy to select the next segment. It starts downloading the video at the lowest possible bit rate and it measures the average download rate. When the buffer capacity is beyond I (same as in SARA), the algorithm, based on the average download rate and the buffer capacity, chooses the next segment as the one with the bit rate closest to the available bandwidth. The switching-up and switching-down is optimistic.

The basic algorithms is set with $I = 2$ and $B_{max} = 10$. The SARA algorithm is set with $I = 2$, $B_\alpha = 5$, $B_\beta = 10$ and $B_{max} = 12$.

In figure 1.11 is depicted the variation of the bit rate chosen by both algorithms. Even with a steady bandwidth, SARA performs better than the basic algorithm: the great variation of segment sizes affected both algorithm but since SARA knew before hand the segment sizes it could anticipate their decreasing, thus choosing a better bit rate.

In figures 1.12 and 1.13, we see that SARA keeps performing better than the basic algorithm even when the available bandwidth is greater. Overall SARA chooses a better bit rate and the buffer avoids SARA to switch-down the bit rate in response to a sudden decrease of the segment sizes.

1.5.2 Buffer-based Bit Rate Adaptation

In this section we illustrate a buffer based adaptation algorithm, that is an algorithm that decides the bit rate mainly based on the buffer characteristics.

If the throughput drops suddenly, requesting video segments with an increase in quality may lead to a buffer underflow, thus interrupting the playback. With the algorithm proposed in [17], we try to estimate the buffer level in the near future to avoid buffer

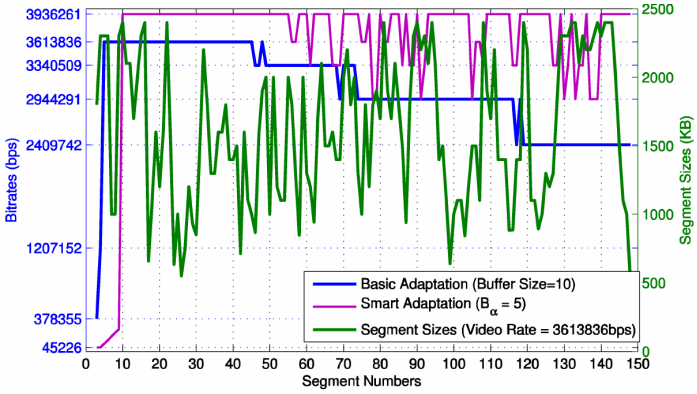

Figure 1.11: Effect of segment sizes on Basic and SARA with a bandwidth of 1 Mbps. Source: [15]

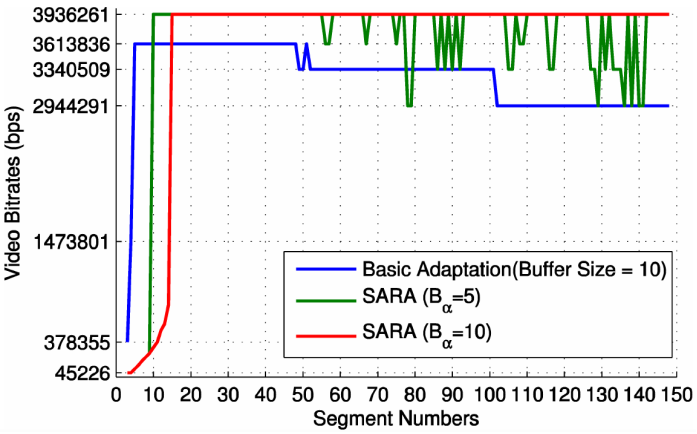

Figure 1.12: Bit rate variation with bandwidth of 4 Mbps. Source: [15]

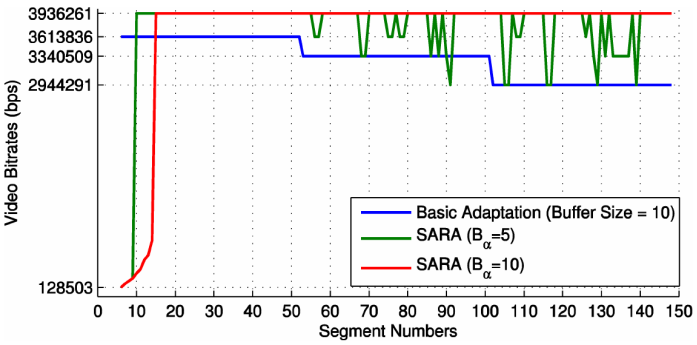

Figure 1.13: Bit rate variation with bandwidth of 8 Mbps. Source: [15]

underflows.

First, we provide two definitions:

- **Arrival curve**: represents the accumulated data size received by the client at a given time t;
- **Playback curve**: represents the accumulated data size consumed by the client at a given time t.

An example of these curves is plotted in picture 1.14. Suppose that the client starts receiving data at time t_0^r and the playback starts at time t_0^p: then, the initial buffering data is $(t_0^r - t_0^p)$ seconds. If the average arrival rate is equal to the playback rate, the buffer will be stable.

In the following, suppose that the server has a set of K representations for each segment: the set of possible bit rates is $\mathfrak{R} = \{R_0, R_1, \cdots, R_{K-1}\}$. With B_l we denote the request bit rate for segment l. At time t_i, after downloading the current segment i with bit rate B_i, the client decides a sequence of bit rates for the next N segments $\{b_{(i,1)}, b_{(i,2)}, \cdots, b_{(i,N)}\}$, where (i, j) means the segment $i + j$ and $b_{(i,j)}$ means the bit rate decided for the segment $i + j$. This sequence of bit rates is called a path P.

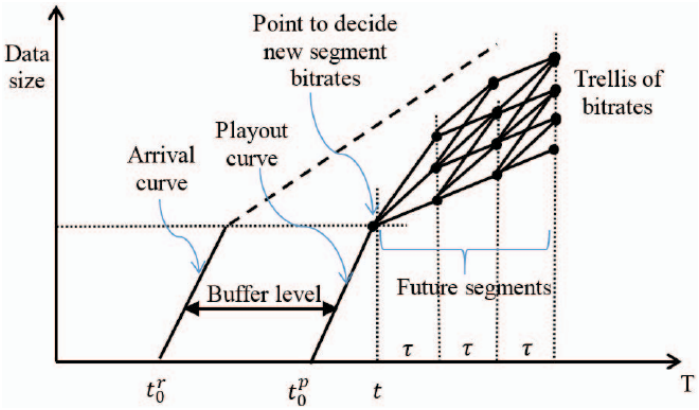

Figure 1.14: Arrival curve and playback curve. Source: [17]

In picture 1.14 we see trellis of bit rates: those are all the possible sequences of bit rates that the client could decide to request at time t. Thus, the decision problem is represented by a trellis in which each sequence of bit rates is shown as a path.

We observe that the video bit rate $b_{(i,0)} = B_i$ is the video bit rate requested for the current segment and it is considered as the starting point of our path P. Moreover, the final bit rate $b_{(i,N)}$ in path P should be the highest value but lower than the current throughput T_{curr}, that is $b_{(i,N)} = \sup\{R_k \in \mathfrak{R} \ : \ R_k < T_{curr}\}$, because the client has to preserve its buffer.

At each step, we denote with $\beta_{(i,j)}$, $0 \leq \beta_{(i,j)} \leq \beta_{max)}$ and β_{max} the maximum buffer size, the buffer level right after downloading the segment (i, j). An estimation of the buffer level could be the following:

$$\beta_{(i,j)} = \begin{cases} \beta_{(i,j-1)} + \tau \cdot \left(1 - \frac{b_{(i,j)}}{T_{(i,j}}\right) & \forall j > 0 \\ \beta(i,0) & j = 0 \end{cases} \tag{1.1}$$

where τ is the duration of the segment (i, j), $T_{(i,j)}$ is the estimated throughput for segment (i, j), thus the interval to download segment (i, j) is $\frac{\tau \cdot b_{(i,j)}}{T_{(i,j)}}$. To avoid buffer underflows, all estimations of the buffer level need to be higher than a predefined minimum β_{min}, that is

$$\beta_{(i,j)} \geq \beta_{min}, \quad 1 \leq j \leq N \tag{1.2}$$

needs to hold.

Now, we assume the in the near future the throughput remains close to the estimated throughput $T_{est} \cong T_{(i,N)} \cong T_{(i,N-1)} \cong \cdots \cong T_{(i,0)}$. Then, equation 1.1 becomes

$$\beta_{(i,j)} = \begin{cases} \beta_{(i,j-1)} + \tau \cdot \left(1 - \frac{b_{(i,j)}}{T_{est}}\right) & \forall j > 0 \\ \beta(i,0) & j = 0 \end{cases} \tag{1.3}$$

Using the throughput estimation of [18], where the throughput estimate is stable under small fluctuations but highly responsive on sudden changes of bandwidth, after the client has decided a path P at time t_i, it builds a new path P' at time $t_{i'}$ only if the throughput has changed by an arbitrary factor.

Algorithm 2 Bit rate selection for segment (i, j)

Input: $j, b_{(i,j-1)}, \beta_{(i,j-1)}$
Output: $b_{(i,j)}, \beta_{(i,j)}$

for $b_{(i,j)} := (b_{(i,j-1)} + \Delta b) \rightarrow b_{(i,N)}$ **do**
  $\beta_{(i,j-1)} + \tau \cdot \left(1 - \frac{b_{(i,j)}}{T_{est}}\right)$
  if $\beta_{min} < \beta_{(i,j)}$ **then**
    if $b_{(i,j)} = b_{(i,N)}$ **then**
      $P := \{b_{(i,0)}, b_{(i,1)}, \cdots, b_{(i,j)}\}$;
      ** store path P and stop **;
    else if $j < N$ **then**
      ** select bit rate for segment $(i, j+1)$ with input parameters $(j+1, b_{(i,j)}, \beta_{(i,j)})$ **
    end if
  end if
end for

The algorithm 2 describes how the path is built. Since the client will request the final segment after N segments, the simplest way to build the path is to request video bit rates with a small change per step. This amount should be

$$\Delta b = \frac{b_{(i,N)} - b_{(i,0)}}{N};$$

in this way we keep the standard deviation of the decided bit rates in path P small [17]. From the user perspective this means that the video quality changes gradually, thus providing a smooth playback.

The buffer estimation is done by applying equation 1.3 and, to avoid buffer underflows when there is a drastically degradation of the throughput, constraint 1.2 is enforced. Note that this algorithm works also in the case the throughput increases: this time, $\Delta b > 0$, 1.2 keeps holding and the path is decided with small increasing steps.

The last step is to decide N. If the current buffer level is high, then N can be set to a high value: the client decides the video bit rates with smaller steps and need a larger number of segments to reach the final bit rate. On the contrary, if the current level of the buffer is small, it is better to set N at a low value in order to react quickly to the throughput change. Then N could be set as follows

$$N = [R_{buffer} \cdot \Omega + 1]$$

where R_{buffer} is the buffer ratio function, defined as

$$R_{buffer} = \begin{cases} \frac{\beta_{curr} - \beta_{min}}{\beta_{max} - \beta_{min}} & \beta_{curr} > \beta_{min} \\ 0 & \beta_{curr} \leq \beta_{min} \end{cases},$$

and Ω is a constant that represent the aggressiveness of path P. Note that in the worst case ($\beta_{curr} \leq \beta_{min}$), the algorithm behaves exactly as the instant throughput based method.

Performance In the following we show some experimental results of the proposed algorithm. The various constants described in the previous paragraph are set as follows: $\beta_{min} = \beta_{max}/2 = 10$ seconds, $\Omega = 6$ and the path is built every time the throughput estimation changes by at least 10%.

The proposed algorithm is compared with two other algorithms. The first one, here called "Aggressive method", is an instant throughput based algorithm, that is, the client selects the representation of the next segment to download only based on the instant throughput of the last segment downloaded [19]. The second one, here called "Threshold-based method", is a different buffer based method: it defines three thresholds on the buffer capacity, $0 \leq B_{min} < B_{low} < B_{max}$, and it chooses the next segment to download based on how much data are stored in the buffer. If the buffer capacity is in the interval $[0, B_{min})$, it starts downloading the next segment with the lowest bit rate in order to ensure that the playback of the video starts as soon as possible and subsequently, if the throughput is high enough, it switches to a higher representation. When the buffer capacity is greater or equal to B_{min}, the aggressiveness of the algorithm increases. The goal is to keep the buffer capacity in the interval $[B_{low}, B_{max}]$ [19]. The thresholds are set as follows: $B_{min} = 5s$, $B_{low} = 10s$, $B_{max} = 20s$ [17].

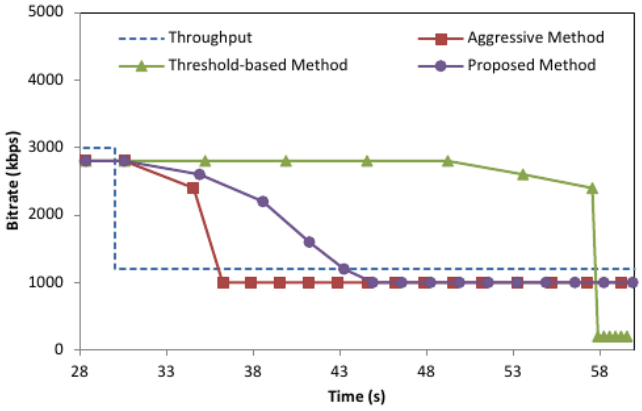

Figure 1.15: Adapted bit rate on a simple scenario. Source: [17]

In figures 1.15 and 1.16 we can see the adapted bit rate and the buffer capacity on a simple scenario. The "Aggressive method" reacts quickly to the change in throughput, however the bit rate variations could impact negatively to the user experience since it cannot select representation above the throughput even when the buffer is full. The "Threshold-based method" instead tries to keep the bit rate high but when the buffer level goes below the minimum it switches to the lowest representation in order to keep the buffer in the target interval. The algorithm proposed here selects the representation with a gradually decreasing bit rate (small steps of 400 kbps [17]) and the buffer level is maintained and rarely goes below the minimum.

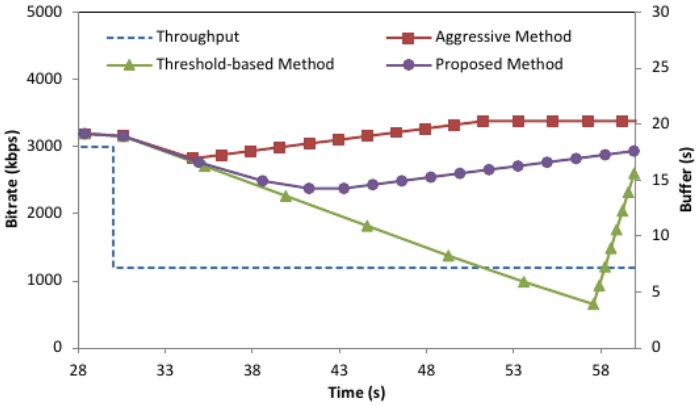

Figure 1.16: Buffer capacity on a simple scenario. Source: [17]

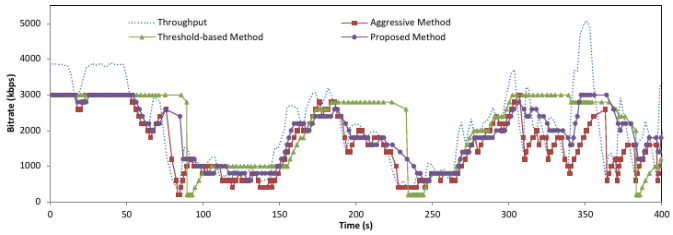

Figure 1.17: Adapted bit rate on a complex scenario. Source: [17]

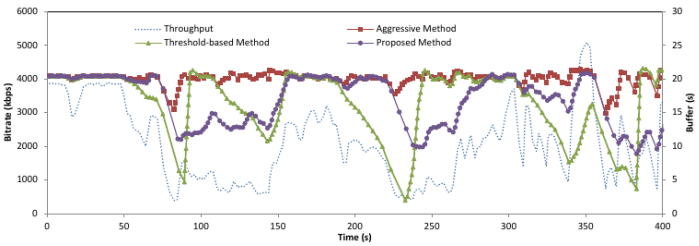

Figure 1.18: Buffer capacity on a complex scenario. Source: [17]

In figures 1.17 and 1.18 we can see the adapted bit rate and the buffer level in a complex scenario where the throughput varies a lot. In these plots we can see that the "Aggressive method" is the worst with the respect to the other two. The "Threshold-based method" tries to keep the bit rate high but reacts negatively when the throughput decreases for a long time. The proposed method instead, even though it does not provide in some cases a higher bit rate of the "Threshold-base method", the increasing/decreasing in small steps of the representation copes better with large throughput fluctuations.

Bibliography

[1] Cisco, "Cisco visual networking index: Global mobile data traffic forecast update, 2016–2021 white paper," tech. rep., Cisco, March 2017.

[2] T. Stockhammer, "Dynamic adaptive streaming over http - standards and design principles," in *MMSys '11 Proceedings of the second annual ACM conference on Multimedia systems*, Qualcomm Incorporated c/o Normon Research, 2011.

[3] E. R. Pantos and W. May, *HTTP Live Streaming*. Apple Inc. and Major League Baseball Advanced Media, 23 ed., May 2017.

[4] *ISO/IEC 23009-1: "Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats"*.

[5] 3rd Generation Partnership Project, *3GPP TS 26.233: "Transparent end-to-end packet-switched streaming service (PSS); General description"*, June 2017.

[6] *MS-SSTR: Smooth Streaming Protocol*.

[7] "Dynamic adaptive streaming over http." `https://en.wikipedia.org/wiki/Dynamic_Adaptive_Streaming_over_HTTP`.

[8] M. G. Michalos, S. P. Kessanidis, and S. L. Nalmpantis, "Dynamic adaptive streaming over http," *Journal of Engineering Science and Technology*, vol. 5, no. 2, pp. 30–34, 2012.

[9] 3rd Generation Partnership Project, *3GPP TS 26.234: "Transparent end-to-end Packet-switched Streaming Service (PSS); Protocols and codecs"*, March 2017.

[10] 3rd Generation Partnership Project, *3GPP TS 26.244: "Transparent end-to-end packet switched streaming service (PSS); 3GPP file format (3GP)"*, March 2017.

[11] 3rd Generation Partnership Project, *3GPP TS 26.247: "Transparent end-to-end Packet-switched Streaming Service (PSS); Progressive Download and Dynamic Adaptive Streaming over HTTP (3GP-DASH)"*, June 2017.

[12] E. R. Fielding and E. J. Reschke, *IETF RFC 7230: "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing"*, June 2014.

[13] "Adaptive Bitrate Algorithm: How They Work and How to Optimize Your Stack." `https://www.slideshare.net/EricaBeavers/abr-algorithms-explained-from-streaming-media-east-2016`.

[14] L. Yitong, S. Yun, M. Yinian, L. Jing, L. Qi, and Y. Dacheng, "A study on quality of experience for adaptive streaming service," in *2013 IEEE International Conference on Communications Workshops (ICC)*, pp. 682–686, June 2013.

[15] D. M. Parikshit Juluri, Venkatesh Tamarapalli, "SARA: Segment aware rate adaptation algorithm for dynamic adaptive streaming over HTTP," in *Communication Workshop (ICCW), 2015 IEEE International Conference on*, IEEE, 2015.

[16] "The Big Buck Bunny Movie." `http://www.bigbuckbunny.org`.

[17] H. T. Le, D. V. Nguyen, N. P. Ngoc, A. T. Pham, and T. C. Thang, "Buffer-based bitrate adaptation for adaptive http streaming," in *2013 International Conference on Advanced Technologies for Communications (ATC 2013)*, pp. 33–38, Oct 2013.

[18] T. C. Thang, Q. D. Ho, J. W. Kang, and A. T. Pham, "Adaptive streaming of audiovisual content using MPEG DASH," *IEEE Transactions on Consumer Electronics*, vol. 58, pp. 78–85, February 2012.

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