University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Doctoral Dissertations | Dissertations and Theses

April 2021

Video Adaptation for High-Quality Content Delivery

Kevin Spiteri
University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

 Part of the Computer Sciences Commons

Recommended Citation

Spiteri, Kevin, "Video Adaptation for High-Quality Content Delivery" (2021). *Doctoral Dissertations*. 2141.
https://doi.org/10.7275/20604181 https://scholarworks.umass.edu/dissertations_2/2141

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

VIDEO ADAPTATION FOR HIGH-QUALITY CONTENT DELIVERY

A Dissertation Presented

by

KEVIN SPITERI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2021

College of Information and Computer Sciences

© Copyright by Kevin Spiteri 2021

All Rights Reserved

VIDEO ADAPTATION FOR HIGH-QUALITY CONTENT DELIVERY

A Dissertation Presented

by

KEVIN SPITERI

Approved as to style and content by:

Ramesh Sitaraman, Chair

Prashant Shenoy, Member

Don Towsley, Member

Michael Zink, Member

James Allan, Chair of the Faculty
College of Information and Computer Sciences

ABSTRACT

VIDEO ADAPTATION FOR HIGH-QUALITY CONTENT DELIVERY

FEBRUARY 2021

KEVIN SPITERI

B.Eng., UNIVERSITY OF MALTA

M.S., OAKLAND UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Ramesh Sitaraman

Modern video players employ complex algorithms to adapt the bitrate of the video that is shown to the user. Bitrate adaptation requires a tradeoff between reducing the probability that the video freezes (rebuffers) and enhancing the quality of the video. A bitrate that is too high leads to frequent rebuffering, while a bitrate that is too low leads to poor video quality. In this dissertation we propose video-adaptation algorithms to deliver content and maximize the viewer's quality of experience (QoE).

Video providers partition videos into short segments and encode each segment at multiple bitrates. The video player adaptively chooses the bitrate of each segment to download, possibly choosing different bitrates for successive segments. We formulate bitrate adaptation as a utility-maximization problem, and design algorithms to provide provably near-optimal time-average utility.

Real-world systems are generally too complex to be fully represented in a theoretical model and thus present a new set of challenges. We design algorithms that deliver

video on production systems, maintaining the strengths of the theoretical algorithms while also tackling challenges faced in production. Our algorithms are now part of the official DASH reference player dash.js and are being used by video providers in production environments.

Most online video is streamed via HTTP over TCP. TCP provides reliable delivery at the expense of additional latency incurred when retransmitting lost packets and head-of-line blocking. Using QUIC allows the video player to tolerate some packet loss without incurring the performance penalties. We design and implement algorithms that exploit this added flexibility to provide higher overall QoE by reducing latency and rebuffering while allowing some packet loss.

Recently virtual reality content is increasing in popularity, and delivering 360° video comes with new challenges and opportunities. The viewing space is often partitioned in tiles, and a viewer using a head-mounted display only sees a subset of the tiles at any time. We develop an open source simulation environment for fast and reproducible testing of 360° algorithms. We develop adaptation algorithms that provide high QoE by allocating more bandwidth resources to deliver the tiles that the viewer is more likely to see, while ensuring that the video player reacts in a timely manner when the viewer changes their head pose.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Online videos are the "killer" application of the Internet with videos currently accounting for more than half of the Internet traffic. Video viewership is growing at a torrid pace and videos are expected to account for more than 85% of all Internet traffic within a few years [10]. As all forms of traditional media migrate to the Internet, video providers face the daunting challenge of providing a good quality of experience (QoE) for users watching their videos. Video providers are diverse and include major media companies (e.g., NBC, CBS), news outlets (e.g., CNN), sports organizations (e.g., NFL, MLB), and video subscription services (e.g., Netflix, Hulu). Recent research has shown that low-performing videos that start slowly, play at lower bitrates, and freeze frequently can cause viewers to abandon the videos or watch fewer minutes of the videos, significantly decreasing the opportunity for generating revenue for the video providers [16, 27, 47], underscoring the need for a high-quality user experience.

1.1 Adaptive video streaming

Providing a high-quality experience for video users requires balancing two contrasting requirements. The user would like to watch the highest-quality version of the video possible, where video quality can be quantified by the bitrate at which the video is encoded. For instance, watching a movie in high definition (HD) encoded at 10 Mbps arguably provides a better user experience than watching the same movie in standard definition (SD) encoded at a bitrate of 800 kbps. In fact, there is empirical evidence that the user is more engaged and watches longer when the video is presented

at a higher bitrate [16]. However, it is not always possible for users to watch videos at the highest encoded bitrate, since the bandwidth available on the network connection between the video player on the user's device and the video server constrains what bitrates can be watched. In fact, choosing a bitrate that is higher than the available network bandwidth[1] will lead to video freezes in the middle of the playback, since the rate at which the video is being played exceeds the rate at which the video can be downloaded. Such video freezes are called *rebuffers* and playing the video continuously *without rebuffers* is a key factor in the QoE perceived by the user [27]. Thus, balancing the contrasting requirements of playing videos at a high bitrate while at the same time avoiding rebuffers is central to providing a high-quality video watching experience.

The need to adjust the video playback to the characteristics of the device and the network has led to the evolution of Adaptive Bitrate (ABR) streaming and HTTP Adaptive Streaming (HAS) which is now the de facto standard for delivering videos on the Internet [46]. ABR streaming requires that each video is partitioned into *segments,* where each segment corresponds to a few seconds of play. Each segment is then encoded in a number of different bitrates to accommodate a range of device types and network connectivities.

Several popular implementations of HAS streaming systems exist, including Apple's HTTP Live Streaming (HLS) [3], Microsoft's Live Smooth Streaming (Smooth) [62] and Adobe's Adaptive Streaming (HDS) [1]. Each has its own proprietary implementation and slight modifications to the basic ABR technique described above. A key recent development is a unifying open-source standard for ABR streaming called MPEG-DASH [52]. DASH is broadly similar to the other ABR protocols and is a

[1]Throughout this dissertation, we say **bandwidth** when talking about network throughput and **bitrate** when talking about encoding quality.

particular focus in our empirical evaluation. The DASH Industry Forum also provides an open-source reference video player, dash.js [13].

Recently, virtual reality and augmented reality has been gaining popularity [63]. The 360° video content can be watched by a viewer using a head-mounted display (HMD) and responds to the viewer's motion. Providing high-quality experience for 360° videos is more complex than for 2-D videos. The video covers a larger viewing area and thus requires a higher bitrate. In order to add depth perception, the video needs to be transmitted in stereo, further increasing the overall bitrate. Video providers can reduce the bandwidth requirements by exploiting the fact that the viewport, i.e., the area of video visible to the user, is only a subset of the 360° area encoded in the video. Providers may partition the video area in tiles and encode tiles separately. The 360° video player only needs to download the tiles that are present in the viewport. The video player needs a view prediction algorithm to predict which tiles to download. DASH has a spatial relationship description (SRD) extension that supports tiles for 360° video [36].

1.1.1 ABR metrics for high QoE

We started our ABR research by getting extensive feedback from video providers and users across a spectrum of the media industry. Particularly, we utilized a biweekly conference call by the dash.js community for feedback [19]. There was near consensus on the requirements for a good ABR algorithm that we state below.

1. *High Bitrate.* Should play the video at the highest sustainable quality (i.e., bitrate).

2. *Low Rebuffering.* Should avoid rebuffering events (i.e. freezes) that occur due to the client buffer being empty.

3. *Low Oscillations.* Should avoid excessive bitrate oscillations where the video quality is frequently modified during the playback.

4. *Responsiveness to Network Events.* Should react quickly to network events. For instance, if the network throughput suddenly drops (resp., increases), the ABR algorithm should decrease (resp., increase) the video bitrate to adjust to the new network state.

5. *Responsiveness to User Events.* Should react quickly to user events. For instance, if a user starts up a new video, or seeks to a new spot within the same video, the playback should start to play quickly at the highest sustainable bitrate.

6. *Low-Latency Live Streaming.* Should perform well when streaming live videos that requires low *latency*, where *latency* is the maximum time between when the video is captured and when the user sees it. A key challenge is that since latency must be low, the client buffer is necessarily small and can hold no more than a few segments. Thus, video segments cannot be fetched by the client well in advance of when they are played out. A small buffer leaves little room for error as a single suboptimal ABR decision could result in draining the buffer, resulting in rebuffering. The precise definition of low latency is subjective and depends on the use case [54, 55]. In this dissertation, by low latency we mean latencies under 10s.

1.2 Adaptive video streaming challenges and proposed solutions

Achieving a high QoE for video streaming is a major challenge due to the sheer diversity of video-capable devices that include smartphones, tablets, desktops, and televisions (Figure 1.1). Further, the devices themselves can be connected to the Internet in a multitude of ways, including cable, fiber, DSL, WiFi and mobile wireless,

Figure 1.1. Video streaming on different devices via diverse network conditions.

each providing different bandwidth characteristics. We list four adaptive streaming challenges that we address in this dissertation and outline the proposed solutions.

1.2.1 Principled aproach to adaptive bitrate streaming

Challenge: There is a wide variety of ABR algorithms that commonly fall in one of two categories, throughput based and buffer based. Throughput based algorithms estimate the network throughput and select a video bitrate that does not exceed the estimate throughput. Buffer based algorithms choose the bitrate based on how much downloaded video is already available in the video buffer. However, algorithms in both classes are traditionally heuristics without principled justification.

Proposed solution: We formulate video streaming as an optimization problem using Lyapunov optimization with two main objectives, high bitrate and low rebuffering. We derive BOLA, a buffer based ABR algorithm. We did not specifically design BOLA to be buffer based, but the optimization framework yielded a buffer based algorithm and a novel function mapping the buffer level to the desired bitrate. We also prove that the algorithm provides near optimal utility. This work has been published in IEEE/ACM Transactions on Networking [51].

1.2.2 Taking the theoretical algorithm to production

Challenge: Designing a principled ABR algorithm requires accurate modelling of multiple system components such as the network and video player. However, theoretical models generally cannot capture all the complexity of production systems. Thus, deploying the theoretical work in production comes with a new set of challenges [31]. Practical considerations for video streaming include finite buffer capacity, user behavior when controlling the video player, and sudden changes in network conditions.

Proposed solution: We design two novel ABR algorithms that augment the theoretically-derived BOLA algorithm to work in production environments. The first algorithm called BOLA-E introduces the concept of a virtual placeholder buffer that helps the algorithm work around the practical limitations of the video buffer. The second algorithm called DYNAMIC detects when practical limitations affect BOLA and dynamically switches between BOLA and a basic throughput based algorithm accordingly. Our implementation of the algorithms is now part of the official DASH reference player dash.js [13] and is being used by video providers in production environments. This work has been published in ACM Transactions on Multimedia Computing, Communications, and Applications [50].

1.2.3 Transport protocols for video delivery

Challenge: Early video streaming systems used the Internet as a best-effort system without reliable transport [57]. On the other hand, modern video providers use HAS as the default video delivery method [46], delivering video via HTTP over TCP or QUIC. While this exploits the vast resources that content delivery networks (CDNs) have to deliver HTTP content, TCP or QUIC packet loss recovery can increase latency which in turn can increases rebuffering.

Proposed solution: We propose a modification to the QUIC protocol to allow a mix of reliable and unreliable delivery, delivering headers and key frames reliably and

delivering the non-header parts of non-key frames unreliably. Our approach called VOXEL brings some of the strengths of early video streaming systems while allowing easy integration with existing HTTP infrastructure. In fact, VOXEL can be deployed incrementally and is interoprable with traditional HTTP video streaming. Our main contribution to VOXEL in this dissertation involves augmenting BOLA-E to exploit the capabilities provided by VOXEL to reduce rebuffering and improve QoE. This work is under review for publication [37].

1.2.4 Simulation framework for 360° video adaptation algorithms

Challenge: Delivering tiled 360° video cannot rely on traditional 2-D algorithms. First, the video player needs to use a view prediction algorithm to predict which part of the video falls within the viewport. Second, a 360° ABR algorithm needs to use information from the view prediction algorithm to allocate more bandwidth to the more important tiles. This makes algorithm design more challenging. Also, the need for HMDs makes live testing of new algorithms more difficult.

Proposed solution: We develop Sabre360, a simulation testbed for 360° video that allows rapid development of 360° algorithms. Sabre360 simulates a 360° video session using three data elements as inputs: the video metadata, a headset motion trace recorded for the video, and a prerecorded bandwidth trace. Sabre360 also takes a view prediction algorithm and an ABR algorithm as input. After the session simulation, Sabre360 provides a detailed session log that can be parsed to provide QoE metrics to evaluate the algorithms. Sabre360 allows efficient testing, making it possible to tune the algorithms before the live testing stage.

1.3 Dissertation outline

We formulate a model for online video delivery and develop the BOLA algorithm that provably provides near-optimal time-average utility in Chapter 2. Chapter 3

describes how we augmented BOLA for production settings, and implemented it as the default ABR algorithm in dash.js. Our algorithms are now being used by video providers in production environments. We describe a new transport protocol for video streaming in Chapter 4 and further augment BOLA to exploit the protocol. We also develop a simulation framework for developing 360° video algorithms in Chapter 5. In Chapter 6 we summarize our work and propose some ideas for future work.

CHAPTER 2

BOLA: NEAR-OPTIMAL BITRATE ADAPTATION FOR ONLINE VIDEOS

Our first contribution is a principled approach to the design of bitrate adaptation algorithms for ABR streaming. In particular, we formulate bitrate adaptation as a utility maximization problem that incorporates the two key components of QoE: the average bitrate of the video experienced by the user and the duration of the rebuffer events. An increase in the average bitrate increases utility, whereas rebuffering decreases it. A strength of our framework is that utility can be defined in a very general manner, say, depending on the content, video provider, or user device.

Using Lyapunov optimization, we derive an online bitrate adaptation algorithm called BOLA (Buffer Occupancy based Lyapunov Algorithm) that provably achieves utility that is within an additive factor of the maximum possible utility in the large video regime. While numerous bitrate adaptation algorithms have been proposed [15, 23, 26, 30, 32, 61] and implemented within video players, our algorithm is the first to provide a *theoretical guarantee* on the achieved utility. Further, BOLA provides an explicit knob for video providers to set the relative importance of a high video quality in relation to the probability of rebuffering.

While not an explicit part of the Lyapunov optimization framework, we also show how BOLA can be adapted to avoid frequent bitrate switches during video playback. Bitrate switches are arguably less annoying than rebuffering, but it is still of some concern to video providers and users alike if such switches occur too frequently.

Most algorithms implemented in practice use a *throughput-based* approach where the available bandwidth between the server and the video player is predicted and the

predicted value is used to determine the bitrate of the next segment that is to be downloaded. A complementary approach is a *buffer-based* approach that does not predict the bandwidth, but only uses the amount of data that is currently stored in the buffer of the video player. Recently, there has been empirical evidence that a buffer-based approach has desirable properties that bandwidth-based approaches lack and has been adopted by Netflix [23]. An intriguing outcome of our work is that the optimal algorithm within our utility maximization framework requires only knowledge of the amount of data in the buffer and no estimate of the available bandwidth. Thus, our work provides the first theoretical justification for why buffer-based algorithms perform well in practice and adds new insights to the ongoing debate [61] within the video streaming and DASH standards communities of relative efficacy of the two approaches. Further, since our algorithm BOLA is buffer-based, it avoids the overheads of more complex bandwidth prediction present in current video player implementations and is more stable under bandwidth fluctuations. Note that our results imply that the buffer level is a *sufficient statistic* that indirectly provides all information about past bandwidth variations required for choosing the next bitrate.

2.1 System Model

Our system model closely captures how ABR streaming works on the Internet today. We consider a video player that downloads a video file from a server over the Internet and plays it back to the user. The video file is segmented into segments that are downloaded in succession. The available bandwidth between the server and the player varies over time. This can be due to reasons such as network congestion and wireless fading among others. The viewing experience of the user is determined by both the video quality as quantified by the bitrates of the segments that are played back and the playback characteristics such as rebuffering. The objective of the player

is to maximize a utility associated with the user's viewing experience while adapting to time-varying (and possibly unpredictable) changes in the available bandwidth.

Video Model: The video file is segmented into N segments indexed as $\{1, 2, \ldots, N\}$ where each segment represents p seconds of the video. On the server, each segment is available in M different bitrates where a segment encoded at a higher bitrate has a larger size in bits and its playback provides a better user experience and higher utility. Suppose the size (in bits) of any[1] segment encoded at bitrate index m is S_m bits and suppose the utility derived by the user from viewing it is given by υ_m where $m \in \{1, 2, \ldots, M\}$. WLOG, let the segment bitrates be non-decreasing in index m. Then, the following holds.

$$\upsilon_1 \leq \upsilon_2 \leq \ldots \leq \upsilon_M \iff S_1 \leq S_2 \leq \ldots \leq S_M. \tag{2.1}$$

Note that the actual encoding bitrate for bitrate index m is given by S_m/p bits/second.

Video Player: The video player downloads successive segments of the video file from the server and plays back the downloaded segments to the user. Each segment must be downloaded in its entirety before it can be played back. We assume that the player sends requests to the server to download one segment at a time. Also, the segments are downloaded in the same order as they are played back. The video player has a finite buffer of size $Q_{\max}$ segments[2] to store the downloaded but yet-to-be-played-back segments. Measuring the buffer in segments is equivalent to measuring it in seconds since the segment duration p is fixed. If the buffer is full the player cannot download any new segments and waits for a fixed period of time given by Δ seconds before attempting to download a new segment. The segments that are

[1]For simplicity, we assume that the segment size (in bits) is S_m for all segments of a given bitrate index m. However, our framework can be easily extended to the case where the segment size for the same bitrate can vary across segments.

[2]It is common practice for video players to measure the buffer in seconds of playback time rather than in bits.

fully downloaded are played back at a fixed rate of $1/p$ segments/second without any idling.

When sending a download request for a new segment, the player also specifies the desired bitrate for that segment. This enables the player to tradeoff the overall video quality with the likelihood of rebuffering that occurs when there are no segments in the buffer for playback. Note that while each segment has a fixed playback time of p seconds, the size of the segment (in bits) can be different depending on its bitrate. Thus, the choice of bitrate for a segment impacts its download time.

Network Model: The available bandwidth (in bits/second) between the server and player is assumed to vary continuously in time according to a stationary random process $\omega(t)$. We do not make any assumptions about knowing the statistical properties or probability distribution of $\omega(t)$ except that it has finite first and second moments as well as a finite inverse second moment. Suppose the player starts to download a segment of bitrate index m at time t. Then the time t' when the download finishes satisfies the following:

$$S_m = \int_t^{t'} \omega(\tau) d\tau \tag{2.2}$$

Let $\mathbb{E}\{\omega(t)\} = \omega_{\text{avg}}$. Then, $\mathbb{E}\{t' - t\} = S_m/\omega_{\text{avg}}$.

2.2 Problem Formulation

We consider two primary performance metrics[3] that affect the overall QoE of the user: (1) time-average playback quality which is a function of the bitrates of the segments viewed by the user and (2) fraction of time spent not rebuffering. To

[3]We do not include the secondary objective of avoiding frequent bitrate switches in our formulation, but we deal with it empirically in Section 2.4.5.

formalize these metrics, we consider a time-slotted representation of our system model. The timeline is divided into non-overlapping consecutive slots of variable length and indexed by $k \in \{1, 2, \ldots\}$. Slot k starts at time t_k and is $T_k = t_{k+1} - t_k$ seconds long. We assume that $t_1 = 0$. At the beginning of each slot, the video player makes a control decision on whether it should start downloading a new segment, and if yes, its bitrate. If a download decision is made, then a request is sent to the server and the download starts immediately[4]. This download takes T_k seconds and is completed at the end of slot k. Note that T_k is a random variable whose actual value depends on the realization of the $\omega(t)$ process as well as the choice of segment bitrate. If the player decides not to download a new segment in slot k (for example, when the buffer is full), then this slot lasts for a fixed duration of Δ seconds.

We define the following indicator variable for each slot k:

$$a_m(t_k) = \begin{cases} 1 & \text{if the player downloads a segment} \\ & \text{of bitrate index } m \text{ in slot } k, \text{ and} \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

Then, for all k, we must have $\sum_{m=1}^{M} a_m(t_k) \leq 1$. Moreover, when $\sum_{m=1}^{M} a_m(t_k) = 0$, then no segments are downloaded.

Denote the buffer level (measured in number of segments) at the start of slot k by $Q(t_k)$. The dynamics of this queue can be expressed using the following equation:

$$Q(t_{k+1}) = \max[Q(t_k) - \frac{T_k}{p}, 0] + \sum_{m=1}^{M} a_m(t_k) \tag{2.4}$$

Here, the arrival value into this queue in slot k is given by $\sum_{m=1}^{M} a_m(t_k)$ which is 1 if a download decision is made in slot k and 0 otherwise. The departure value is T_k/p

[4]Any delays associated with sending the request can be added to the overall download time.

which represents the total number of segments (including fractional segments) that could have departed the buffer in slot k. Note that the actual value of T_k is revealed at the end of slot k. Also note that a segment that is downloaded in slot k becomes available for playback only from the next slot. We assume that the buffer level is initialized to 0, i.e., $Q(t_1) = 0$.

Let K_N denote the index of the slot in which the N^{th} (i.e., last) segment is downloaded. Also, denote the time at which the player finishes playing back the last segment by T_{end}. Then the first performance metric of interest is the time-average expected *playback utility* $\overline{v}_N$ which is defined as

$$\overline{v}_N \triangleq \frac{\mathbb{E}\left\{\sum_{k=1}^{K_N}\sum_{m=1}^{M} a_m(t_k)v_m\right\}}{\mathbb{E}\left\{T_{\text{end}}\right\}} \tag{2.5}$$

where the numerator denotes the expected total utility across all N segments. Note that a segment can only be played back after it has been downloaded entirely. Thus, T_{end} is greater than the last segment's download finish time, i.e., $T_{\text{end}} > t_{K_N} + T_{K_N}$.

The second performance metric of interest is the expected fraction of time $\overline{s}_N$ that is spent not rebuffering and can be interpreted as a measure of the average playback "smoothness". This can be calculated by observing that the actual playback time for all N segments is Np seconds. Thus, the expected *playback smoothness* $\overline{s}_N$ is given by

$$\overline{s}_N \triangleq \frac{Np}{\mathbb{E}\left\{T_{\text{end}}\right\}} = \frac{\mathbb{E}\left\{\sum_{k=1}^{K_N}\sum_{m=1}^{M} a_m(t_k)p\right\}}{\mathbb{E}\left\{T_{\text{end}}\right\}} \tag{2.6}$$

where in the last step we use the relation that $Np = \sum_{k=1}^{K_N}\sum_{m=1}^{M} a_m(t_k)p$. Note that $T_{\text{end}} \geq Np$ (since at most one segment can be played back at any time), so that $\overline{s}_N \leq 1$.

2.2.1 Design Objective

We want to design a control algorithm that maximizes the joint utility $\overline{v}_N + \gamma \overline{s}_N$ subject to the constraint that $Q(t_k) \leq Q_{\max}$ for all k. Here, $\gamma > 0$ is an input weight parameter for prioritizing playback utility versus the playback smoothness.

This problem can be formulated as a stochastic optimization problem with a time-average objective over a finite horizon and dynamic programming (DP) based approaches can be used to solve it [6]. However, traditional DP based methods have two major disadvantages. First, they require knowledge of the distribution of the $\omega(t)$ process which may be hard to obtain. Second, even when such knowledge is available, the resulting DP can have a very large state space. This is because the state space for this problem under a DP formulation would consist of not only the timeslot index k and value t_k, but also the buffer size $Q(t_k)$. Further, an appropriate discretization of the $\omega(t)$ process would be required to obtain a tractable solution.

In order to overcome the above challenges associated with traditional DP based methods, we take an alternate approach in this chapter. First, we consider the bitrate adaptation problem in the limiting regime when the video size becomes large, i.e., $N \to \infty$. Second, we replace the finite buffer constraint with a *rate stability* constraint (made precise in the next section). The reason for making these assumptions is that it results in simplifications to the original problem as discussed in the next section. This allows us to develop a bitrate adaptation algorithm that does not require *any* knowledge of the distribution of $\omega(t)$, yet offers provable theoretical performance guarantees in the large video size regime while satisfying the finite buffer constraint. As shown later in Section 2.4.4, with slight modifications, this algorithm can be used for finite sized videos as well and offers close to optimal performance in our experiments.

2.2.2 Problem Relaxation

Consider the bitrate adaptation problem in the limiting regime when the video size becomes large, i.e., $N \to \infty$. Then, the metrics $\overline{v}_N$ and $\overline{s}_N$ can be expressed as

$$\begin{aligned}\overline{v} \triangleq \lim_{N\to\infty} \overline{v}_N &= \lim_{N\to\infty} \frac{\mathbb{E}\left\{\sum_{k=1}^{K_N}\sum_{m=1}^{M} a_m(t_k)\upsilon_m\right\}}{\mathbb{E}\left\{T_{\text{end}}\right\}} \\ &= \frac{\lim_{K_N\to\infty}\frac{1}{K_N}\mathbb{E}\left\{\sum_{k=1}^{K_N}\sum_{m=1}^{M} a_m(t_k)\upsilon_m\right\}}{\lim_{K_N\to\infty}\frac{1}{K_N}\mathbb{E}\left\{\sum_{k=1}^{K_N} T_k\right\}} \end{aligned} \tag{2.7}$$

$$\begin{aligned}\overline{s} \triangleq \lim_{N\to\infty} \overline{s}_N &= \lim_{N\to\infty} \frac{\mathbb{E}\left\{\sum_{k=1}^{K_N}\sum_{m=1}^{M} a_m(t_k)p\right\}}{\mathbb{E}\left\{T_{\text{end}}\right\}} \\ &= \frac{\lim_{K_N\to\infty}\frac{1}{K_N}\mathbb{E}\left\{\sum_{k=1}^{K_N}\sum_{m=1}^{M} a_m(t_k)p\right\}}{\lim_{K_N\to\infty}\frac{1}{K_N}\mathbb{E}\left\{\sum_{k=1}^{K_N} T_k\right\}} \end{aligned} \tag{2.8}$$

This follows by noting that the difference between the expected total playback finish time $\mathbb{E}\left\{T_{\text{end}}\right\}$ and the expected total download finish time $\mathbb{E}\left\{\sum_{k=1}^{K_N} T_k\right\}$ is upper bounded by a finite value due to the finite $Q_{\max}$. Specifically, this upper bound is given by $Q_{\max}p$. Therefore, instead of considering the total playback finish time, we can consider the total download finish time in the objective when the video size becomes large.

Next, replace the finite buffer constraint with a rate stability constraint [33]. This constraint only requires that the time-average arrival rate into the buffer cannot exceed the time-average playback rate. This is equivalent to requiring that

$$\lim_{K_N\to\infty}\frac{1}{K_N}\mathbb{E}\left\{\sum_{k=1}^{K_N}\sum_{m=1}^{M} a_m(t_k)p\right\} \leq \lim_{K_N\to\infty}\frac{1}{K_N}\mathbb{E}\left\{\sum_{k=1}^{K_N} T_k\right\} \tag{2.9}$$

The rate stability constraint is a relaxation of the finite buffer constraint since any policy that ensures finite buffers is always rate stable but not vice versa. Therefore,

under this relaxation, the optimal time-average utility cannot be smaller than the optimal time-average utility with the finite buffer constraint.

With these relaxations, our performance objective for the bitrate adaptation problem is to maximize the joint utility $\overline{v} + \gamma\overline{s}$ subject to the rate stability constraint (2.9).Let us denote the optimal time-average utility for this problem by $v^* + \gamma s^*$. This problem fits in the framework of Lyapunov optimization for renewal systems [34].Specifically, this framework extends the original Lyapunov optimization technique [33] to systems with variable length renewal frames and shows that minimizing a "drift-plus-penalty" ratio over every frame yields an optimal control algorithm. We refer to [34] for details on this method. In the context of our bitrate adaptation problem, the variable length slots represent the renewal frames.

The following characterization can be made about the optimality of *i.i.d. algorithms*.

Lemma 1 *For the bitrate adaptation problem in the limiting regime when the video size becomes large, i.e., $N \to \infty$, there exists a* buffer-state-independent *stationary algorithm that makes i.i.d. control decisions in every slot and satisfies the rate stability constraint while achieving time-average utility no smaller than $v^* + \gamma s^*$.*

Proof: This follows from Lemma 1 in [34] and uses the fact that the conditional expectations and conditional second moments of the frame length and utility are bounded under any algorithm. The full proof is omitted for brevity. □

Note that such a buffer-state-independent stationary algorithm is not necessarily feasible for our finite buffer system. Further, calculating it explicitly would require knowledge of the distribution of $\omega(t)$. However, instead of calculating this policy explicitly, we will use its existence and characterization per Lemma 1 to design an *online* control algorithm using the technique of Lyapunov optimization over renewal frames.

In the next section, we will present this algorithm and show that it meets the finite buffer constraint while achieving a time-average utility that is within $O(1/Q_{\max})$ of $\upsilon^* + \gamma s^*$ without requiring any knowledge of the distribution of $\omega(t)$.

2.3 BOLA: An Online Control Algorithm

We first give a high-level intuition of the Lyapunov optimization over renewals technique. This technique converts the problem of optimizing the time-average metrics in (2.7)–(2.8) subject to the time-average constraint in (2.9) into a series of per slot optimization problems. The problem to be solved in each slot involves minimizing a ratio of the expected drift-plus-penalty value in that slot to the expected length of the slot. As shown in the Appendix, this can be done without requiring any knowledge of the distribution of $\omega(t)$. The drift term consists of $\mathbb{E}\left\{(Q(t_{k+1})^2 - Q(t_k)^2)/2 \mid Q(t_k)\right\}$ and serves to meet the rate stability constraint (2.9). The penalty term consists of the playback utility and playback smoothness received in that slot. We keep the utility and smoothness as separate terms even though they can be folded into one metric. This allows us to tune the relative importance of increasing video bitrate and reducing rebuffering without changing the algorithm. The algorithm uses a control parameter $V > 0$ to allow a tradeoff between the buffer size and the performance objectives.

We now present the algorithm. In every slot k, given the buffer level $Q(t_k)$ at the start of the slot, our algorithm makes a control decision by solving the following deterministic optimization problem. Let

$$\rho(t_k, \mathbf{a}(t_k)) = \begin{cases} 0 & \text{if } \sum_{m=1}^{M} a_m(t_k) = 0, \\ \dfrac{\sum_{m=1}^{M} a_m(t_k)\big(V\upsilon_m + V\gamma p - Q(t_k)\big)}{\sum_{m=1}^{M} a_m(t_k) S_m} & \\ & \text{otherwise.} \end{cases} \tag{2.10}$$

Then determine $\mathbf{a}(t_k)$ by solving the optimization problem:

$$\begin{array}{ll} \text{Maximize:} & \rho(t_k, \mathbf{a}(t_k)) \\ \text{Subject to:} & \sum_{m=1}^{M} a_m(t_k) \leq 1, a_m(t_k) \in \{0,1\} \end{array} \tag{2.11}$$

The constraints of this problem result in a very simple solution structure. Specifically, the optimal solution is given by:

1. If $Q(t_k) > V(\upsilon_m + \gamma p)$ for all $m \in \{1, 2, \ldots, M\}$, then the no-download option is chosen, i.e., $a_m(t_k) = 0$ for all m. Note that in this case $T_k = \Delta$.

2. Else, the optimal solution is to download the next segment at bitrate index m^* where m^* is the index that maximizes the ratio $\big(V\upsilon_m + V\gamma p - Q(t_k)\big)/S_m$ among all m for which this ratio is positive.

Notice that solving this problem does not require any knowledge of the $\omega(t)$ process. Further, the optimal solution depends only on the buffer level $Q(t_k)$. That's why we call our algorithm *BOLA: Buffer Occupancy based Lyapunov Algorithm.* These properties of BOLA should be contrasted with the bandwidth prediction based strategies that have been recently proposed for this problem that require explicit prediction of the available bandwidth for control decisions.

The following theorem characterizes the theoretical performance guarantees provided by BOLA.

Theorem 2 *Suppose BOLA as defined by (2.11) is implemented in every slot using a control parameter* $0 < V \leq \frac{Q_{\max}-1}{\upsilon_M + \gamma p}$. *Assume* $Q(0) = 0$. *Then, the following hold.*

1. *The queue backlog satisfies* $Q(t_k) \leq V(\upsilon_M + \gamma p) + 1$ *for all slots* k. *Further, the buffer occupancy in segments never exceeds* $Q_{\max}$.

2. The time-average utility achieved by BOLA satisfies

$$\overline{v}^{\text{BOLA}} + \gamma \overline{s}^{\text{BOLA}} \geq v^* + \gamma s^* - \frac{p^2 + \Psi}{2p^2 V} \tag{2.12}$$

where Ψ is an upper bound on $\mathbb{E}\{T_k^2\}$ under any control algorithm and is assumed to be finite.

Proof: We first show part 1 using induction. Note that the bound $Q(t_k) \leq V(v_M + \gamma p) + 1$ holds for $k = 1$ since $Q(t_1) = Q(0) = 0$. Now suppose it holds for some k. We will show that it will also hold for $k + 1$. We have two cases.

Case 1: $Q(t_k) \leq V(v_M + \gamma p)$

From the queueing equation (2.4), it follows that the maximum that $Q(t_k)$ can increase in slot k is by 1. This implies that $Q(t_{k+1}) \leq V(v_M + \gamma p) + 1$.

Case 2: $V(v_M + \gamma p) < Q(t_k) \leq V(v_M + \gamma p) + 1$

We have $Q(t_k) > V(v_m + \gamma p)$ for all $m \in \{1, 2, \ldots, M\}$ (using (2.1)). It follows from the structure of optimal solution to (2.11) that BOLA will choose the no-download option in this case. As a result, $Q(t_k)$ cannot increase and we have that $Q(t_{k+1}) \leq V(v_M + \gamma p) + 1$.

$Q(t_k)$ denotes the total number of segments in the buffer. This can be at most $Q_{\max}$ using the relation

$$V \leq \frac{Q_{\max} - 1}{v_M + \gamma p}.$$

In part 2, we show the bound in (2.12) using the technique of Lyapounov optimization over variable size frames [33]. We first define a Lyapunov function $L(Q(t_k))$ as

$$L(Q(t_k)) = \frac{1}{2} Q^2(t_k)$$

and define the per-slot conditional Lyapunov drift $D(t_k)$ as

$$D(t_k) \triangleq \mathbb{E}\left\{L(Q(t_{k+1})) - L(Q(t_k)) | Q(t_k)\right\}.$$

We use the queueing equation (2.4), to bound $D(t_k)$. We consider two cases for (2.4): $Q(t_k) \leq T_k/p$ and $Q(t_k) > T_k/p$. In the first case we have

$$D(t_k) = \mathbb{E}\left\{\frac{1}{2}\left(\sum_{m=1}^{M} a_m(t_k)\right)^2 - \frac{1}{2}Q^2(t_k)|Q(t_k)\right\}. \tag{2.13}$$

In the second case we have

$$D(t_k) = \mathbb{E}\left\{\frac{1}{2}\left(\frac{T_k}{p} - \sum_{m=1}^{M} a_m(t_k)\right)^2 - Q(t_k)\left(\frac{T_k}{p} - \sum_{m=1}^{M} a_m(t_k)\right)|Q(t_k)\right\} \tag{2.14}$$

In both cases, $D(t_k)$ is bounded by

$$D(t_k) \leq \frac{p^2+\Psi}{2p^2} - Q(t_k)\mathbb{E}\left\{\frac{T_k}{p} - \sum_{m=1}^{M} a_m(t_k)|Q(t_k)\right\} \tag{2.15}$$

where Ψ is an upper bound on $\mathbb{E}\{T_k^2\}$ under any control algorithm and is assumed to be finite.

Following the methodology of the Lyapunov optimization technique, we subtract $V \times$ reward term from both sides of the above to get

$$\begin{aligned} D(t_k) - V\mathbb{E}\left\{\sum_{m=1}^{M} a_m(t_k)(\upsilon_m + \gamma p)|Q(t_k)\right\} \\ \leq \frac{p^2+\Psi}{2p^2} - Q(t_k)\mathbb{E}\left\{\frac{T_k}{p} - \sum_{m=1}^{M} a_m(t_k)|Q(t_k)\right\} \\ - V\mathbb{E}\left\{\sum_{m=1}^{M} a_m(t_k)(\upsilon_m + \gamma p)|Q(t_k)\right\} \end{aligned} \tag{2.16}$$

Let us denote the control decisions (and resulting slot lengths) under our control algorithm by the superscript BOLA while those under the stationary policy of Lemma 1 by STAT. Since BOLA greedily maximizes over a frame, it ensures that

$$\mathbb{E}\left\{\sum_{m=1}^{M} a_m^{\text{BOLA}}(t_k)(Q(t_k) - V(\upsilon_m + \gamma p))|Q(t_k)\right\}$$
$$\leq \eta \times \mathbb{E}\left\{\sum_{m=1}^{M} a_m^{\text{STAT}}(t_k)(Q(t_k) - V(\upsilon_m + \gamma p))|Q(t_k)\right\} \quad (2.17)$$

where $\eta = \frac{\mathbb{E}\left\{T_k^{\text{BOLA}}|Q(t_k)\right\}}{\mathbb{E}\left\{T_k^{\text{STAT}}|Q(t_k)\right\}}$. To see this, compare the ratio on the left hand side above with the objective in (2.11) while noting that we can express the denominator as $\mathbb{E}\left\{T_k^{\text{BOLA}}|Q(t_k)\right\} = (\sum_{m=1}^{M} a_m^{\text{BOLA}}(t_k)S_m)/\omega_{\text{avg}}$. It should be noted that this ratio can be minimized without requiring knowledge of ω_{avg}. Then we use (2.17) to express (2.16) as

$$D^{\text{BOLA}}(t_k) - V\mathbb{E}\left\{\sum_{m=1}^{M} a_m^{\text{BOLA}}(t_k)(\upsilon_m + \gamma p)|Q(t_k)\right\}$$
$$\leq \frac{p^2 + \Psi}{2p^2} - Q(t_k)\mathbb{E}\left\{\frac{T_k^{\text{BOLA}}}{p} - \eta\sum_{m=1}^{M} a_m^{\text{STAT}}(t_k)|Q(t_k)\right\}$$
$$- V\eta\mathbb{E}\left\{\sum_{m=1}^{M} a_m^{\text{STAT}}(t_k)(\upsilon_m + \gamma p)|Q(t_k)\right\}$$

Substituting the time-average values for the stationary policy we get

$$D^{\text{BOLA}}(t_k) - V\mathbb{E}\left\{\sum_{m=1}^{M} a_m^{\text{BOLA}}(t_k)(\upsilon_m + \gamma p)|Q(t_k)\right\}$$
$$\leq \frac{p^2 + \Psi}{2p^2} - Q(t_k)\Big(\frac{1}{p} - r^{\text{STAT}}\Big)\mathbb{E}\left\{T_k^{\text{BOLA}}|Q(t_k)\right\}$$
$$- V(\upsilon^* + \gamma s^*)\mathbb{E}\left\{T_k^{\text{BOLA}}|Q(t_k)\right\} \quad (2.18)$$

where r^{STAT} denotes the expected arrival rate under the stationary policy and cannot exceed $1/p$ since it is rate stable. Thus we have

$$D^{\text{BOLA}}(t_k) - V\mathbb{E}\left\{\sum_{m=1}^{M} a_m^{\text{BOLA}}(t_k)(\upsilon_m + \gamma p)|Q(t_k)\right\}$$
$$\leq \frac{p^2 + \Psi}{2p^2} - V(\upsilon^* + \gamma s^*)\mathbb{E}\left\{T_k^{\text{BOLA}}|Q(t_k)\right\} \quad (2.19)$$

Taking conditional expectation of both sides and summing over $k \in \{1, 2, \ldots, K_N\}$, we get

$$\mathbb{E}\left\{L(Q(t_{K_{N+1}}))\right\} - V\mathbb{E}\left\{\sum_{k=1}^{K_N}\sum_{m=1}^{M} a_m^{\text{BOLA}}(t_k)(\upsilon_m + \gamma p)\right\}$$
$$\leq \frac{(p^2+\Psi)K_N}{2p^2} - V(\upsilon^* + \gamma s^*)\mathbb{E}\left\{\sum_{k=1}^{K_N} T_k^{\text{BOLA}}\right\} \tag{2.20}$$

Dividing both sides by $V\mathbb{E}\left\{\sum_{k=1}^{K_N} T_k^{\text{BOLA}}\right\}$ and taking the limit as $N \to \infty$ yields the bound in (2.12). □

Remarks: The performance bounds in Theorem 2 show a $[O(1/V), O(V)]$ utility and backlog tradeoff that is typical of Lyapunov based control algorithms for similar utility maximization problems. Specifically, the time-average utility of BOLA is within an $O(1/V)$ additive term of the optimal utility and this gap may be made smaller by choosing a larger value of V. However, the largest feasible value of V is constrained by the buffer size and there is a linear relation between them.

2.3.1 Understanding BOLA With an Example

We now present a sample run to illustrate how BOLA works. We slice a 99-second video using 3-second segments and encode it at five different bitrates. While BOLA only requires the utilities to be a non-decreasing function of the segment bitrate, it is natural to consider concave utility functions with diminishing returns, e.g., a 1 Mbps increase in segment bitrate likely provides a larger utility gain for the user when that increase is from 0.5 Mbps to 1.5 Mbps than when it is from 5 Mbps to 6 Mbps. A natural choice for our example is the logarithmic utility function: let $\upsilon_m = \ln(S_m/S_1)$. Pick $\gamma = 5.0/p$ and $V = 0.93$. The bitrates and utilities are below.

bitrate (Mbps)	0.331	0.688	1.427	2.962	6.000
S (Mb)	0.993	2.064	4.281	8.886	18.00
υ	0.000	0.732	1.461	2.192	2.897

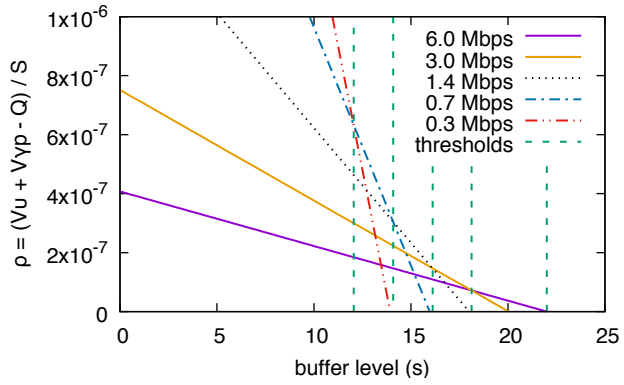


Figure 2.1. The value of $(V\upsilon_m + V\gamma p - Q)/S_m$ for different bitrates depends on the buffer level. ($\gamma p = 5$ and $V = 0.93$.) Note that the buffer level is Qp seconds.

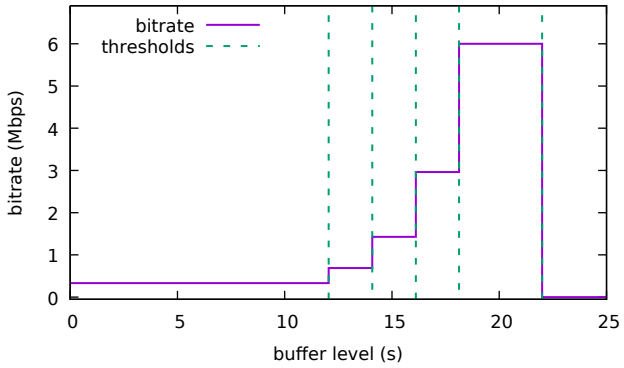


Figure 2.2. BOLA's bitrate choice as function of buffer level. ($\gamma p = 5$,$V = 0.93$.) Note that the buffer level is Qp seconds.

For any slot we choose the segment bitrate to maximize $(V\upsilon_m + V\gamma p - Q)/S_m$ for $1 \leq m \leq M$. Fig. 2.1 shows the relationship between the expression and the buffer level Q for different m. The line intersections mark the buffer levels that correspond to decision thresholds. Fig. 2.2 summarizes BOLA's bitrate choices as a function of the buffer level.

Fig. 2.3 shows how BOLA works. We use a synthetic network bandwidth profile as shown in Fig. 2.3(a). We can see the feedback loop involving the bitrate in (a) and the buffer level in (b). BOLA chooses the bitrate based directly on the buffer level using Fig. 2.2. The bitrate affects the download time, thus it indirectly affects the

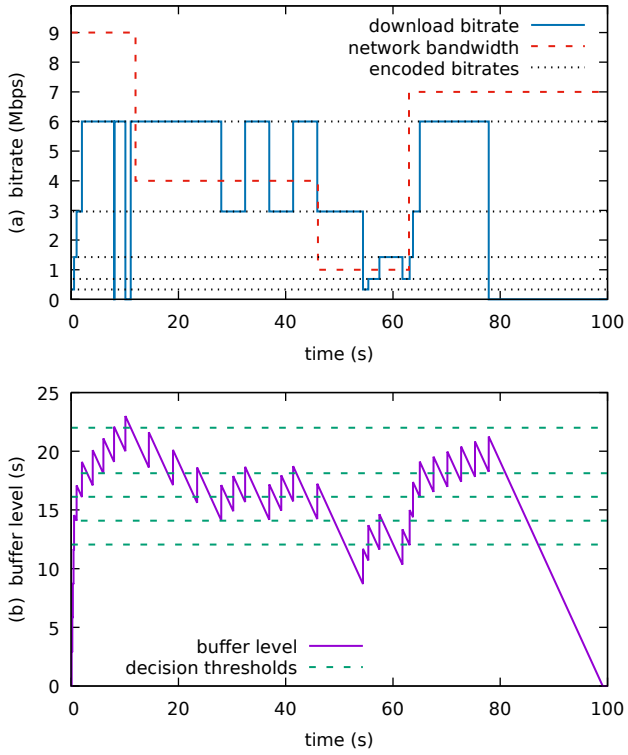


Figure 2.3. Sample video download and playback using BOLA. (a) The video is encoded at 5 different bitrates. The network bandwidth varies from high to low and back to high. The downloaded segment bitrate adapts to the network bandwidth. (b) The buffer level variation triggers bitrate changes when it crosses the thresholds.

buffer level at the beginning of the following slot. Finally, when all the segments are downloaded, the video player plays out the segments remaining in the buffer.

2.3.2 Choosing Utility and Parameters γ and V

While we chose a logarithmic utility function for the example, a video provider can use any utility function satisfying (2.1). The utility function might also take into account system characteristics such as the type of device a viewer is using.

γ corresponds to how strongly we want to avoid rebuffering. Increasing γ translates the graphs in Figs. 2.1 and 2.2 to the right, effectively shifting the thresholds higher without changing their relative distance. BOLA will thus download more low-bitrate segments to maintain a larger (and safer) buffer level.

Increasing V expands the graphs in Figs. 2.1 and 2.2 horizontally about the origin. If we have a maximum buffer level $Q_{\max}$ we want to avoid downloading unless there is enough space for one full segment on the buffer, that is unless $Q \leq Q_{\max} - 1$. For a given $Q_{\max}$ we can set $V = (Q_{\max} - 1)/(\upsilon_M + \gamma p)$.

While we showed how to choose reasonable values for γ and V, video providers are more familiar with choosing buffer level targets. A method to derive the parameters from buffer level targets is included in Section 2.5.2. Alternatively, video providers might choose γ and V by employing an approach such as Oboe [2] to auto-tune the BOLA parameters.

2.4 Implementation and Empirical Evaluation

We first implemented a basic version of BOLA, named BOLA-BASIC, directly from (2.11). Recall that when the buffer level is full BOLA does not download a segment but waits for Δ seconds. Rather than picking an arbitrary value for Δ, we use a dynamic wait until $Q(t_k) \leq V(\upsilon_M + \gamma p)$. This has the same effect as picking a fixed but very small Δ, so the theoretical analysis still holds. We also implemented

Table 2.1. Bitrates used for Big Buck Bunny Test Video

Bitrate Index m	Bitrate (Mbps) Mean	Bitrate (Mbps) Standard Deviation	Segment Size S (Mb) Mean	Segment Size S (Mb) Standard Deviation	Utility $v = \ln(S/S_1)$
1	0.230	0.038	0.690	0.113	0.000
2	0.331	0.054	0.993	0.162	0.364
3	0.477	0.096	1.431	0.287	0.729
4	0.688	0.120	2.064	0.360	1.096
5	0.991	0.182	2.973	0.545	1.461
6	1.427	0.275	4.281	0.825	1.825
7	2.056	0.394	6.168	1.182	2.190
8	2.962	0.564	8.886	1.691	2.556
9	5.027	0.891	15.08	2.673	3.084
$M = 10$	6.000	1.078	18.00	3.232	3.261

other versions of BOLA, namely BOLA-FINITE, BOLA-O, and BOLA-U, that we describe later in this section.

2.4.1 Test Methodology

We simulated all versions of BOLA using the Big Buck Bunny movie [8]. The 10-minute movie was encoded at 10 different bitrates and sliced in 3-second segments. Although each quality index has a specified average bitrate, segments may have variable bitrate (VBR) because of the varying nature of the movie. We simulate playback times longer than 10 minutes by repeating the movie. Again we choose a logarithmic utility function:

$$v_m = \ln(S_m/S_1). \tag{2.21}$$

Table 2.1 shows the mean and standard deviation of the bitrate and segment size for each quality index and the respective utility values.

The DASH Industry Forum provides benchmarks for various aspects of the DASH standard [14]. The benchmarks include twelve different network profiles. Profiles 1–6 have network bandwidths ranging from 1.5 to 5 Mbps while profiles 7–12 have bandwidths ranging from 1 to 9 Mbps. Different latencies are provided for each bandwidth, where the latency is half the round-trip time (RTT). Table 2.2 shows the

Table 2.2. Network Profiles for the Dash Benchmarks

1 Mbps (ms)	3 Mbps (ms)	5 Mbps (ms)	7 Mbps (ms)	9 Mbps (ms)	11 Mbps (ms)
5.0 (38)	5.0 (13)	5.0 (11)			
4.0 (50)	4.0 (18)	4.0 (13)	9.0 (25)	9.0 (10)	9.0 (6)
3.0 (75)	3.0 (28)	3.0 (15)	4.0 (50)	4.0 (50)	4.0 (13)
2.0 (88)	2.0 (58)	2.0 (20)	2.0 (75)	2.0 (150)	2.0 (20)
1.5 (100)	1.5 (200)	1.5 (25)	1.0 (100)	1.0 (200)	1.0 (25)
2.0 (88)	2.0 (58)	2.0 (20)	2.0 (75)	2.0 (150)	2.0 (20)
3.0 (75)	3.0 (28)	3.0 (15)	4.0 (50)	4.0 (50)	4.0 (13)
4.0 (50)	4.0 (18)	4.0 (13)			

odd-numbered bandwidth characteristics. Profile 1 spends 30s at each of 5, 4, 3, 2, 1.5, 2, 3 and 4 Mbps respectively, then starts back at the top. Even-numbered profiles are similar to the preceding odd-numbered profiles but start at the low bandwidth stage. For example, profile 2 starts at 1.5 Mbps.

In addition, we also tested our algorithms using a set of 86 3G mobile bandwidth traces that are publicly available [43]. One trace was excluded because it had an average bandwidth of 80 kbps; our lowest video bitrate is 230 kbps. Since the traces do not include latency measurements, we used 50 ms latency giving a RTT of 100 ms throughout. This is the median RTT measured empirically in [45].

2.4.2 Computing an Upper Bound on the Maximum Utility

In order to evaluate how well BOLA performs on the traces, it is important to derive an upper bound on the maximum utility that is obtainable by *any* algorithm on a given trace. We derive an *offline* optimal algorithm that provides the maximum achievable utility using dynamic programming. We define a table $r(n, t, b)$ that contains the maximum utility possible when we download the n^{th} segment and finish at time t with buffer level b. We initialize the table with $r(0, 0, 0) = 0$. Let $x(n, t, m)$ be the time to download the n^{th} segment at bitrate index m starting at time t. Note that the dependency of x on n is due to VBR. We quantize the time with granularity

δ. While some accuracy is lost, we ensure the final result will still be an upper bound by rounding the download time down.

$$x_\delta(n,t,m) = \lfloor x(n,t,m)/\delta \rfloor \cdot \delta$$

We cap the buffer level at $b_{\max}$.

$$x'_\delta(n,t,b,m) = \max[x_\delta(n,t,m), b + p - b_{\max}]$$

Let $y(n,t,b,m)$ be the rebuffering time.

$$y(n,t,b,m) = \max[x'_\delta(n,t,b,m) - b, 0]$$

We generate entries for $r(n,\cdot,\cdot)$ from $r(n-1,\cdot,\cdot)$ using

$$r(n,t,b) = \max_{m,t',b'} \Big(r(n-1,t',b') + \upsilon_m - \gamma y(n,t',b',m) \Big)$$

such that $t = t' + x'_\delta(n,t',b',m)$ and
$b = b' - x'_\delta(n,t',b',m) + y(n,t',b',m) + p$.

The dynamic programming algorithm is shown in Fig. 2.4.

2.4.3 Evaluating BOLA-BASIC

Fig. 2.5 shows the time-average utility of BOLA-BASIC when the video length is 10, 30 and 120 minutes. We set $\gamma p = 5$ and varied V for different buffer sizes. We compared the utility of BOLA-BASIC with the offline optimal bound described in Section 2.4.2. The offline optimal gave nearly the same utility for the different video lengths. BOLA-BASIC only obtains about 80% of the offline optimal bound. Also, the utility of BOLA-BASIC decreases slightly when the buffer size is increased because

1: $r(0,t,b) \leftarrow \{0 \text{ for } t=b=0,\ -\infty \text{ otherwise}\}$
2: **for** n in $[1,N]$ **do**
3: initialize $r(n,t,b) \leftarrow -\infty$ for all t,b
4: **for** all (t',b') such that $r(n-1,t',b') > -\infty$ **do**
5: **for** m in $[1,M]$ **do**
6: $x \leftarrow$ download time(n,t',m)
7: $x_\delta \leftarrow \lfloor x/\delta \rfloor \cdot \delta$
8: $x'_\delta \leftarrow \max[x_\delta, b' + p - b_{\max}]$
9: $y \leftarrow \max[x'_\delta - b', 0]$
10: $t \leftarrow t' + x'_\delta$
11: $b \leftarrow b' - x'_\delta + y + p$
12: $r' \leftarrow r(n-1,t',b') + \upsilon_m - \gamma y$
13: $r(n,t,b) \leftarrow \max[r(n,t,b), r']$
14: **end for**
15: **end for**
16: **end for**
17: $r^* \leftarrow \max_{(t,b)} \dfrac{r(N,t,b)}{(t+b)}$

Figure 2.4. Calculating the Offline Optimal Utility Upper Bound

it must download more lower-bitrate segments during startup before it can reach the buffer levels required to switch to higher-bitrate segments. Our results suggests that there is room to improve BOLA-BASIC that motivates our next version.

2.4.4 Adapting BOLA to Finite-Sized Videos

BOLA-BASIC was derived under the assumption that the videos are infinite. Thus, some adaptations are needed for BOLA to work effectively with smaller videos. Motivated by our initial experiments, we implemented two adaptations to BOLA-BASIC to derive a version we call BOLA-FINITE.

1) Dynamic V value for startup and wind down: A large buffer allows BOLA-BASIC to perform better but it has two drawbacks. First, it takes longer to prime a large buffer during startup. Lower bitrate segments are preferred until the buffer level reaches steady state. Second, at some late stage all downloads are complete and any remaining buffered video is played out. Any available bandwidth during this period is not utilized. Shortening this period would result in less unutilized available band-

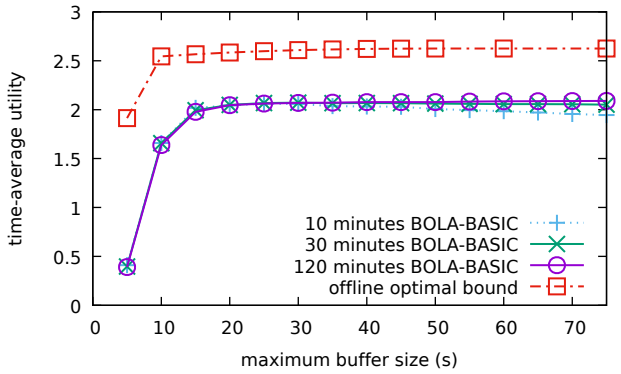


Figure 2.5. Time-average utility for $\gamma p = 5$ using profile 1 for BOLA-BASIC.

for n in $[1, N]$ **do**
  $t \leftarrow \min[\text{playtime from begin}, \text{playtime to end}]$
  $t' \leftarrow \max[t/2, 3p]$
  $Q^{\text{D}}_{\max} \leftarrow \min[Q_{\max}, t'/p]$
  $V^{\text{D}} \leftarrow (Q^{\text{D}}_{\max} - 1)/(v_M + \gamma p)$
  $m^*[n] \leftarrow \arg\max_m (V^{\text{D}} v_m + V^{\text{D}} \gamma p - Q)/S_m$
  if $m^*[n] > m^*[n-1]$ **then**
    $r \leftarrow$ bandwidth measured when downloading segment $(n-1)$
    $m' \leftarrow \max m$ such that $S_m/p \le \max[r, S_1/p]$
    if $m' \ge m^*[n]$ **then**
      $m' \leftarrow m^*[n]$
    else if $m' < m^*[n-1]$ **then**
      $m' \leftarrow m^*[n-1]$
    else if some utility sacrificed for fewer oscillations **then**
      pause until $(V^{\text{D}} v_{m'} + V^{\text{D}} \gamma p - Q)/S_{m'} \ge (V^{\text{D}} v_{m'+1} + V^{\text{D}} \gamma p - Q)/S_{m'+1}$ ▷ BOLA-O
    else
      $m' \leftarrow m' + 1$ ▷ BOLA-U
    end if
    $m^*[n] \leftarrow m'$
  end if
  pause for $\max[p \cdot (Q - Q^{\text{D}}_{\max} + 1), 0]$
  download segment n at bitrate index $m^*[n]$, possibly abandoning
end for

Figure 2.6. The BOLA Algorithm.

1: **function** SHALLABANDON(m, S_m^{R})
2: $r_m \leftarrow (V^{\mathrm{D}}\upsilon_m + V^{\mathrm{D}}\gamma p - Q)/S_m^{\mathrm{R}}$
3: $r_{m'} \leftarrow (V^{\mathrm{D}}\upsilon_{m'} + V^{\mathrm{D}}\gamma p - Q)/S_{m'}$
4: **return** *true* if $r_{m'} > r_m$ for some m' subject to $1 \leq m' < m$
5: **end function**

Figure 2.7. BOLA-FINITE's download abandonment heuristic: m is the current segment bitrate and S_m^{R} is the number of bits remaining to download in the current segment.

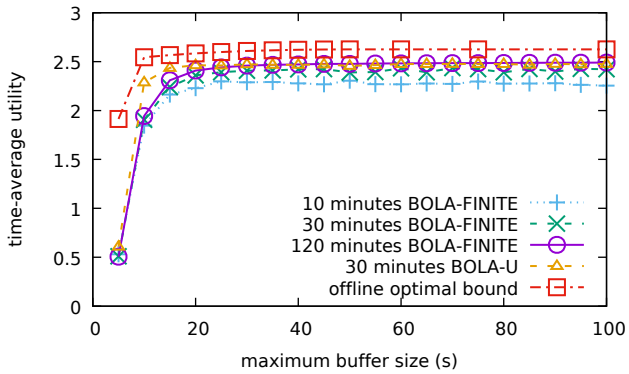


Figure 2.8. Time-average utility for $\gamma p = 5$ using profile 1 for BOLA-FINITE and BOLA-U.

width. We mitigate these effects by introducing a dynamic V^{D} which corresponds to a dynamic buffer size $Q_{\max}^{\mathrm{D}}$, shown in lines 2–5 in Fig. 2.6. BOLA-FINITE does not try to fill the whole buffer too soon and does not try to maintain a full buffer too long. We still need a minimum buffer size $3p$ for the algorithm to work effectively.

2) Download abandonment: BOLA-BASIC takes control decisions just before the download of each segment. Consider a scenario where the player is downloading high-bitrate 6 Mbps segments in good network conditions. The network bandwidth suddenly drops to 1 Mbps as the player has just started a new segment download. The segment will take $6p$ seconds to download, depleting the buffer and possibly causing rebuffering. BOLA-FINITE mitigates this problem by monitoring download progress and possibly abandoning a download. Fig. 2.7 shows how BOLA-FINITE decides whether or not to abandon the download. If a segment at bitrate index m

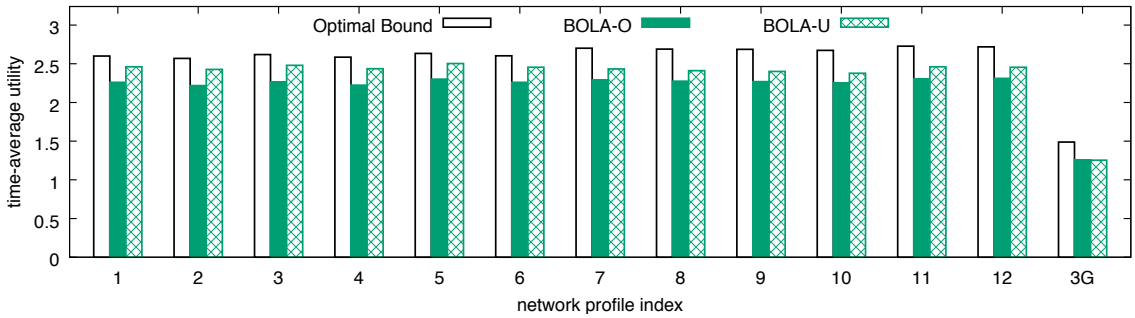


Figure 2.9. The time-average utility of BOLA-O and BOLA-U with $\gamma p = 5$ and a 25-second buffer playing a 30-minute video for the DASH test network profiles 1–12 and mobile traces (3G). BOLA utility is within 84–95% of offline optimal utility.

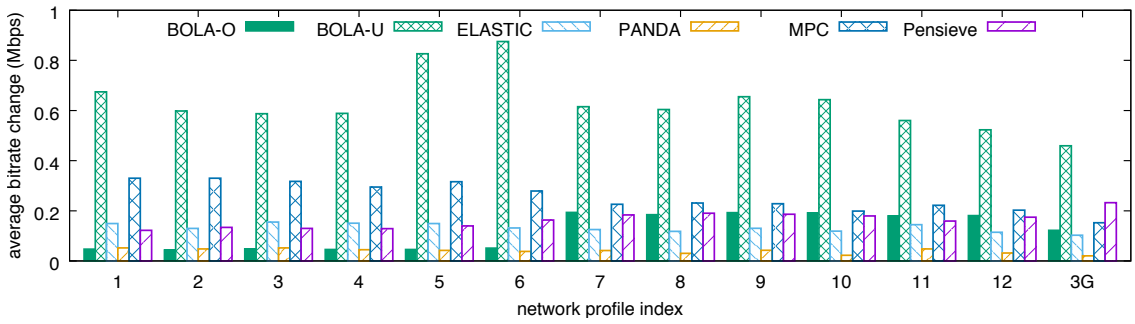


Figure 2.10. The average bitrate change between adjacent segments was smaller for BOLA-O than for BOLA-U, but some bitrate change is needed to accurately track the network bandwidth. In our experiments, as an average across network profiles, ELASTIC and PANDA tracked the bandwidth with similar accuracy to BOLA-O, while MPC and Pensieve had more oscillations.

is being downloaded, the remaining size S_m^{R} is less than S_m. The segment can be abandoned and downloaded at some bitrate index m' subject to $1 \leq m' < m$ when $(V^{\mathrm{D}}\upsilon_m + V^{\mathrm{D}}\gamma p - Q)/S_m^{\mathrm{R}} < (V^{\mathrm{D}}\upsilon_{m'} + V^{\mathrm{D}}\gamma p - Q)/S_{m'}$. The control idea remains the same, but the current bitrate m has a smaller corresponding size S_m^{R} because part of the segment has already been downloaded. Fig. 2.3 illustrates a scenario where abandonment might help. At 46s a 3 Mbps segment download starts. Since there is a bandwidth drop at the time, the segment takes almost 9s to download. The buffer is depleted and BOLA-BASIC switches to downloading at a bitrate of 0.3 Mbps. BOLA-FINITE with abandonment logic would have detected the rapidly depleting buffer and stopped the long download, with the system only dropping to the 1.4 and 0.7 Mbps download bitrates in the low-bandwidth period.

Fig. 2.8 shows the time-average utility of BOLA-FINITE for 10, 30 and 120 minutes of playback time with $\gamma p = 5$. Comparing with BOLA-BASIC in Fig. 2.5, we see that the time-average utility is much closer to the offline optimal bound. The benefit of the adjustments is also evident as the buffer grows larger, as there is no significant decrease in utility caused by filling the buffer with low-bitrate segments in the earlier stages of the video.

2.4.5 Avoiding Bitrate Oscillations

While our performance objective optimizes playback utility and playback smoothness, users are also sensitive to excessive bitrate switching. We discuss three causes of bitrate switches.

1) Bandwidth variation: As the network conditions change, the player varies the bitrate, tracking the network bandwidth. Such switches are acceptable; the player has no control on the bandwidth and should adapt to different network conditions.

2) Dense buffer thresholds: Either a larger number of bitrate levels and/or a smaller buffer size may push the threshold levels closer. If the differences between

threshold levels are less than the segment duration p, adding one downloaded segment to the buffer may push the buffer level over several threshold levels at once. This might cause BOLA-FINITE to overshoot and choose a bitrate that is too high for the available bandwidth. Consequently, the segment download would take much more than p seconds, leading to excessive buffer depletion, causing BOLA-FINITE to switch down its bitrate by more than one level. In such a scenario BOLA-FINITE can oscillate between bitrates, even when the available bandwidth is stable.

3) Bitrate quantization: Having a stable network bandwidth and widely-spaced thresholds still does not avoid all bitrate switching. Suppose the bandwidth is 2.0 Mbps and it lies between two encoded bitrates of 1.5 and 3.0 Mbps. While the player downloads 1.5 Mbps segments, the buffer keeps growing. When the buffer crosses the threshold the player switches to 3.0 Mbps, depleting the buffer. After the buffer gets sufficiently depleted, the player switches back to 1.5 Mbps, and the cycle repeats. In this example, a viewer might prefer the video player to stick to the 1.5 Mbps bitrate, sacrificing some utility in order to have fewer oscillations. Or, a viewer might want to maximize utility and play a part of the video in the higher bitrate of 3.0 Mbps at the cost of more oscillations. We describe two variants of BOLA below to suit either viewer.

The first variant that we call BOLA-O mitigates **o**scillations by introducing bitrate capping (lines 7–20 in Fig. 2.6) when switching to a higher bitrate. BOLA-O verifies that the higher bitrate is sustainable by comparing it to the bandwidth as measured when downloading the previous segment (lines 8–11). Since the motive is to limit oscillations rather than to predict future bandwidth, this adaptation does not drop the bitrate to a lower level than in the previous download (lines 12–13). Continuous downloading at a bitrate lower than the bandwidth would cause the buffer to keep growing. BOLA-O avoids this by allowing the buffer to slip to the appropriate threshold before starting the download (line 15).

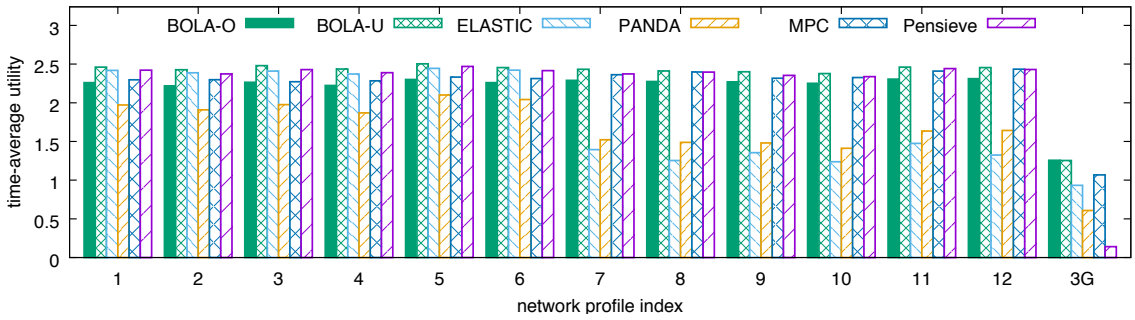


Figure 2.11. The time-average utility of BOLA-O, BOLA-U, ELASTIC, PANDA, MPC and Pensieve with $\gamma p = 5$ playing a 30-minute video for the DASH test network profiles 1–12 and mobile traces (3G). Compared with ELASTIC and PANDA, BOLA-U has about 1.75 times the utility of the other algorithms in roughly half the cases. MPC has a utility between BOLA-O and BOLA-U. Pensieve has a utility between BOLA-O and BOLA-U for profiles 1–12 but performs worse for the mobile (3G) traces.

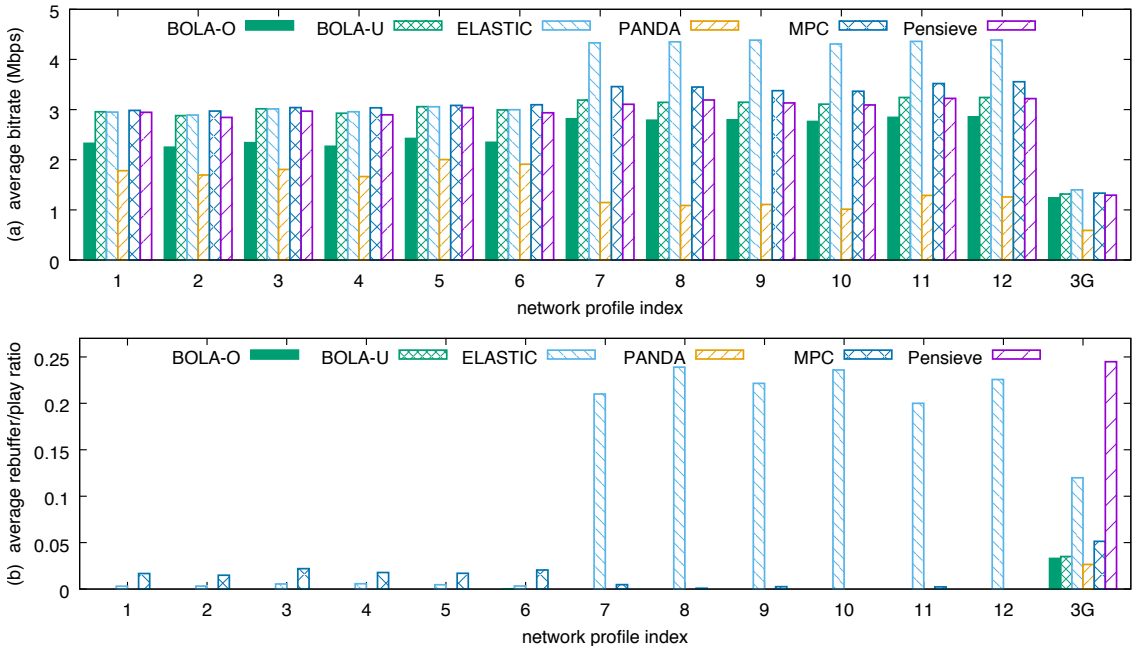


Figure 2.12. Comparing BOLA with ELASTIC, PANDA, MPC and Pensieve using raw metrics: average bitrate and rebuffer-to-play ratio. BOLA, PANDA and Pensieve do not rebuffer for profiles 1–12. ELASTIC has almost no rebuffering for profiles 1–6, but it has a rebuffer-to-play ratio greater than 20% for profiles 7–12. MPC has some rebuffering for almost all profiles. Pensieve has no rebuffering for profiles 1–12. But, Penseive has a 24% rebuffer-to-play ratio for the mobile (3G) traces, as it is unable to perform well for bandwidth conditions that are significantly different from its training set.

The second variant that we call BOLA-U does not sacrifice **u**tility. Excessive buffer growth is avoided by allowing the bitrate to be one level higher than the sustainable bandwidth (line 17). This allows the player to choose 3 Mbps in the example. While BOLA-U does not handle the third type of oscillations, it handles the more severe second type.

Looking back at Fig. 2.8, we see that the added stability of BOLA-U pays off when using a small buffer size and BOLA-U achieves a larger utility than BOLA-FINITE. Fig. 2.9 shows the time-average utility of BOLA-O and BOLA-U with $\gamma p = 5$ and $Q_{\max} p = 25$s playing a 30-minute video. The utility lost by BOLA-O to avoid oscillations is clearly evident. In practice the lost utility is limited by the distance between encoded bitrates; if the next lower bitrate level is not far from the network bandwidth, then little utility will be lost.

We measure oscillations by comparing consecutive segments. The change in bitrate between a segment and the next is the absolute difference between bitrates (in Mbps) of the two segments. Fig. 2.10 shows the bitrate change averaged across all the segments. While BOLA-U has a high average bitrate change because of the quantization, BOLA-O only switches bitrate because of network bandwidth variations.

2.4.6 Comparison With State-of-the-Art Algorithms

We now compare BOLA with four state-of-the art algorithms, ELASTIC [15], PANDA [30], MPC [61] and Pensieve [32]. We use the default design parameters in [15, 30, 32, 61]. We test both BOLA-O and BOLA-U. Although BOLA performs better with larger buffers, we limited the buffer size to 25s for the tests to ensure fairness. ELASTIC targets a buffer level of 15s but the buffer level varies higher. PANDA targets a minimum buffer level of 26s. We use the RobustMPC variant of MPC with a buffer size of 25s. MPC relies on bandwidth estimation; we use the harmonic mean over the last five segment downloads to be consistent with the

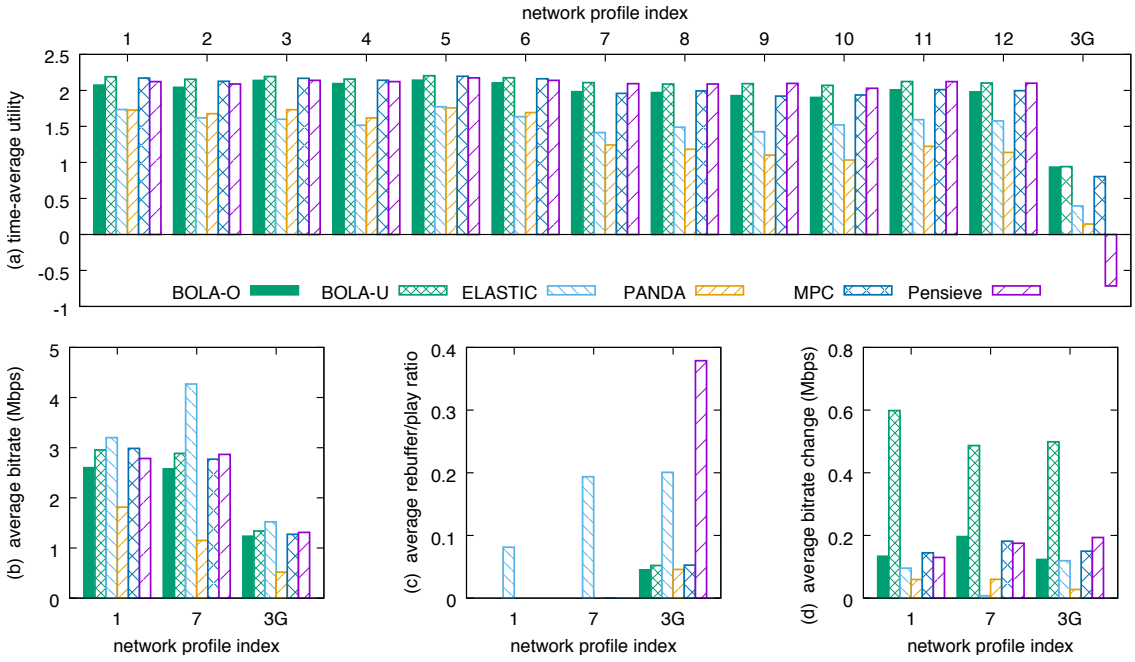


Figure 2.13. The time-average utility of BOLA-O, BOLA-U, ELASTIC, PANDA, MPC and Pensieve with $\gamma p = 5$ playing a different video for 30 minutes, using the DASH test network profiles 1–12 and the mobile traces (3G). Note that Pensieve has negative 3G utility because of excessive rebuffering (the average rebuffer-to-play ratio is 38%). The raw metrics are also provided in the plots above.

empirical evaluation method in [61]. We trained a Pensieve neural network model for the video with a buffer size of 25s. For training Pensieve, we used bandwidth traces generated using the tool provided in the Pensieve repository as recommended.

Fig. 2.11 compares the algorithms using each of the 12 network profiles and the mobile traces. BOLA-U consistently performs significantly better than PANDA. While BOLA-U and ELASTIC perform similarly for profiles 1–6, BOLA-U performs significantly better for the other profiles that have larger bandwidth variations. MPC and Pensieve consistently obtains a utility between BOLA-O and BOLA-U for profiles 1–12, but perform worse for the mobile traces. We repeat the comparison using the average bitrate and rebuffering metrics in Fig. 2.12. This gives an insight into the strengths and weaknesses of the different algorithms.

Comparing BOLA-U with ELASTIC: For profiles 1–6, BOLA-U has approximately the same bitrate as ELASTIC. ELASTIC has a higher bitrate for profiles 7–12, but that comes at a significant cost in terms of rebuffering. For these profiles, the ratio of the rebuffering time to the play time is more than 20% for ELASTIC, while BOLA-U has no rebuffering. For the mobile traces, ELASTIC has marginally higher bitrate than BOLA-U but has a 12.0% rebuffer-to-play ratio compared with BOLA-U's 3.5%. ELASTIC rebuffers significantly more because it does not react in time when the bandwidth drops.

Comparing BOLA-U with PANDA: Both algorithms do not rebuffer for profiles 1–12. For the mobile traces, BOLA-U and PANDA have a rebuffer-to-play ratio of 3.5% and 2.6% respectively. However, PANDA has significantly lower bitrate than BOLA-U. The reason is that PANDA is more conservative and in some cases does not change to a higher bitrate even if it is sustainable.

Comparing BOLA-U with MPC: Both algorithms have similar average bitrates but MPC has slightly higher bitrate for some of the profiles. However, while BOLA-U does not rebuffer, MPC has some rebuffering for most of the profiles. While

it is possible to tune the MPC parameters to avoid that rebuffering, it is not clear how to choose parameters that consistently work for different network conditions. Another factor that might contribute to MPC rebuffering is the bandwidth estimation. When there is a large drop in bandwidth, the recommended harmonic mean bandwidth estimator takes a while to react. Even though RobustMPC factors in network estimation error, rebuffering is not totally eliminated.

Comparing BOLA-U with Pensieve: For profiles 1–12, Pensieve obtains utility between BOLA-O and BOLA-U, but consistently closer to BOLA-U. However, Pensieve has too much rebuffering in the mobile traces, resulting in much worse utility for these traces. While the network traces used to train Pensieve included periods with low bandwidth similar to the mobile traces, Pensieve did not learn a model that would perform well in relatively low bandwidth situations in a mobile setting. This points to a weakness in Pensieve as it is unable to adapt to bandwidth conditions that are significantly different from the training set.

In Fig. 2.10 we show our results for our secondary metric of bitrate oscillations. BOLA-U does not perform well in this metric, since it attempts to maximize utility at the cost of increased oscillations. Comparing BOLA-O with ELASTIC, PANDA, MPC, and Pensieve, ELASTIC has a lower average change than BOLA-O only in the cases where it has a slow reaction and excessive rebuffering. PANDA has a lower average change because it is more conservative and in some cases does not change to a higher bitrate even if that bitrate is sustainable. MPC has higher average change than BOLA-O for profiles 1–12. Pensieve has similar average change to BOLA-O for profiles 1–12.

We also tested the algorithms with more videos to investigate performance when changing characteristics such as content type, segment duration, and available bitrates. The tests showed similar results. One example is the video provided with Pensieve. The video has 49 segments with a segment duration of 4s. It is encoded at

six bitrates: 0.3, 0.75, 1.2, 1.85, 2.85 and 4.3 Mbps. Fig. 2.13 shows the utility and metrics for the same six algorithms with similar conclusions to Figs. 2.10–2.12. Note that Pensieve fails to perform well on mobile traces again, since it is significantly different from its training set.

Thus, from our empirical analysis, we can conclude that BOLA achieves higher utility, and performs more consistently across different scenarios in comparison with ELASTIC, PANDA, MPC and Pensieve. One reason for the consistency of BOLA is that it does not have a large number of parameters. BOLA has two design parameters γ and V, which have an intuitive significance as discussed in Section 2.3.2, and an option of whether or not to trade off some utility to reduce oscillations. Other algorithms have a number of different parameters and tuning the parameters for a particular scenario might make the system less suited for other scenarios. Also, BOLA's ability to abandon a segment during a download and start the download at a lower bitrate allows BOLA to achieve significantly less rebuffering than the other algorithms.

2.5 Deployment

2.5.1 The DASH Reference Player

After developing a theoretical foundation for BOLA and testing it by simulation, we deployed BOLA in a production setting. Particularly, we implemented BOLA in dash.js, the open-source standard DASH reference player [13]. Through dash.js, BOLA is now being used in production by several major video providers and delivery networks such as Akamai, BBC, CBS and Orange. Deployment in production presented a number of new challenges such as operating with even smaller buffer capacities, correctly handling events such as a user seek to a different point in the video, and tolerating delays caused by the video player unrelated to the network con-

ditions. The techniques we implemented to handle these new challenges are described in Chapter 3.

2.5.2 BOLA Parameters

One deployment challenge involves choosing the BOLA parameters γ and V. We gave an intuition to pick the parameters in Section 2.3.2, but video providers are more familiar with choosing buffer level targets. For this purpose, we now discuss how to derive γ and V from intuitive requirements. Consider the following requirements:

1. We want a maximum buffer level $Q_{\max}$.

2. We want to download at the highest bitrate when the buffer level is $Q_{\max}$.

3. We want to download at the lowest bitrate when the buffer level is less than a threshold Q_{low}, and we want to download at a higher bitrate when the buffer level goes above the threshold.

These requirements are easy to understand for video providers who might not be familiar with BOLA. In fact, video providers usually have some preferred maximum buffer level $Q_{\max}$. Further, they might have preference for Q_{low} such as 10s as described in Chapter 3.

To satisfy requirements 1–2, we want (2.11) to switch from choosing $a_M = 1$ to choosing $\sum a_m = 0$ at the threshold when the buffer level is $Q_{\max}$. This happens if

$$\begin{aligned} \rho_{a_M=1} &= \rho_{\mathbf{a}=\mathbf{0}} \\ \frac{V(\upsilon_M + \gamma p) - Q_{\max}}{S_M} &= 0 \end{aligned} \tag{2.22}$$

Note that BOLA satisfies requirement 2 and downloads at the highest bitrate just before the $Q_{\max}$ threshold because at that buffer level we get $\rho_{a_m=1} < \rho_{a_M=1}$ for $m < M$. This is illustrated in Fig. 2.1.

To satisfy requirement 3, we want (2.11) to switch from choosing $a_1 = 1$ to choosing $a_2 = 1$ at the threshold when the buffer level is Q_{low}. This happens if

$$\begin{aligned} \rho_{a_1=1} &= \rho_{a_2=1} \\ \frac{V(\upsilon_1 + \gamma p) - Q_{\text{low}}}{S_1} &= \frac{V(\upsilon_2 + \gamma p) - Q_{\text{low}}}{S_2} \end{aligned} \tag{2.23}$$

Solving (2.22)–(2.23), we obtain

$$V = \frac{Q_{\max} - Q_{\text{low}}}{\upsilon_M - \alpha}, \quad \gamma p = \frac{\upsilon_M Q_{\text{low}} - \alpha Q_{\max}}{Q_{\max} - Q_{\text{low}}}$$

where

$$\alpha = \frac{S_2\upsilon_1 - S_1\upsilon_2}{S_2 - S_1}.$$

When calculating the BOLA parameters from Q_{low} and $Q_{\max}$, the previous intuition about γ and V still hold. If a video provider chooses a larger Q_{low}, γ will be larger and BOLA will give more weight to rebuffering. If a video provider chooses a larger $Q_{\max}$, V will be larger.

2.6 Related Work

There has been a lot of recent work on bitrate adaptation algorithms, much of which is based on estimating the bandwidth of the network connection. FESTIVE [26] uses a harmonic bandwidth estimator to predict future bandwidth from past downloads, limiting bitrate change to one level between successive segments for stability. Notably, FESTIVE attempts to find a tradeoff between efficiency and fairness with competing downloads. BBA [23] is a buffer-based algorithm. BOLA has a few similarities to BBA but the mapping function from buffer level to video bitrate is different. Also, BBA assumes that the buffer size is large (in the order of minutes), thereby making it not suitable for short videos. Further, it does not provide any theoretical

guarantees for its buffer-based approach. A notable algorithm is ELASTIC [15] that uses control theory to adjust the bitrate so as to keep the buffer occupancy at a constant level. Another notable algorithm is PANDA [30] which also estimates the network bandwidth. PANDA drops the download bitrate as soon as low bandwidth is detected but only increases the bitrate slowly to probe the real capacity when a higher bandwidth is detected. Like FESTIVE, PANDA trades efficiency for fairness. In [61], an algorithm using model predictive control (MPC) is proposed to optimize a comprehensive set of metrics. In this approach, the bitrate for the current segment is chosen based on a network bandwidth prediction for the next few segments. But, its performance depends on the accuracy of such a prediction. The approach also requires significant offline optimization to be performed outside of the client for an exhaustive set of scenarios. [32] presents Pensieve, a reinforcement-learning approach to ABR. A neural network model can be trained for a video using a particular buffer size, using a QoE function for reward. A set of bandwidth traces is used as training data. Unfortunately, a trained model does not transfer easily to a different video or, more importantly, to bandwidth conditions not represented in the training data. Unlike prior work, we derive a buffer-based algorithm with theoretical guarantees that is simple to implement within the client and we empirically show its efficacy on extensive network traces. In recent work [2], a method called Oboe for auto-tuning the parameters of BOLA and MPC was presented and shown to improve both algorithms. Further, the work showed that Oboe used in conjunction with traditional ABR algorithms performs better than reinforcement-learning based ABR such as Pensieve.

2.7 Conclusion

We formulated video bitrate adaptation for ABR streaming as a utility maximization problem and derived BOLA, an online control algorithm that is provably near-optimal. Further, we empirically demonstrated the efficacy of BOLA using ex-

tensive traces. In particular, we showed that our online algorithm achieves utility close to the optimal offline algorithm. We showed that our algorithm performs better than state-of-the-art algorithms in a number of different test scenarios.

CHAPTER 3

IMPROVING BITRATE ADAPTATION IN THE DASH REFERENCE PLAYER

To our knowledge, BOLA is the only known online ABR algorithm with provable optimality guarantees within a utility framework. While BOLA achieves near-optimal utility in steady-state conditions, the theory does not apply to transient conditions. Theoretical models generally cannot capture all the complexity of production systems, but our approach is to start with the sound theoretical foundations provided by BOLA and then adapt it to practical implementations that model the intricacies of production settings [31].

While BOLA provided a high bitrate without significant buffering or oscillations, it fell short on some requirements that are important for a real-world production implementation. In particular, since BOLA predominantly used the buffer levels for decision making, it does not respond quickly to user events such as startup and seeking when the buffer starts out empty. It also does not respond quickly enough to rapid changes in the network throughput profile. Further, it did not perform sufficiently well in the live streaming context where the low-latency requirement mandates small buffers. Such deficiencies are not specific to BOLA and are common in other known state-of-the-art algorithms such as BBA [24]. To improve BOLA, we took different approaches shown in Fig. 3.1 and described below.

Algorithm BOLA-E We introduced the notion of a *virtual segment* that contains no video data. We developed a new *placeholder algorithm* that judiciously adds and removes virtual segments to change the buffer levels used by BOLA for bitrate

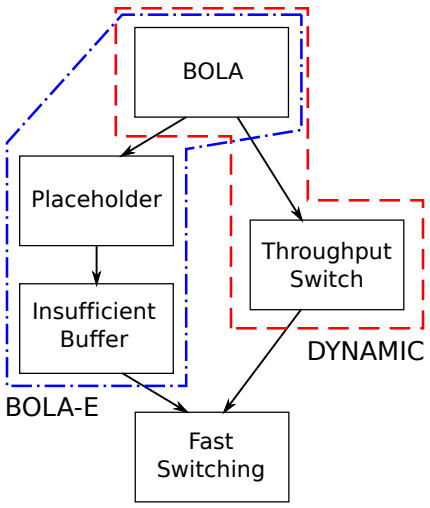


Figure 3.1. Overview of our design and production implementation of ABR algorithms for dash.js.

switching decisions. The placeholder algorithm significantly improves the responsiveness of BOLA to network and user events. Further, we devised the insufficient buffer rule that helps avoid rebuffering when buffer levels are low, especially in live streaming situations when buffers are small. BOLA with the placeholder algorithm and the insufficient buffer rule constitutes an enhanced version of BOLA that we call BOLA-E.

BOLA-E was first released as an experimental version in dash.js version 2.0.0 on Feb 12th 2016. A stable version was released in version 2.6.0 on Sept 1st 2017 and has been in use by video providers since. BOLA-E is not turned on in the DASH reference player by default, but it is one of two optional ABR algorithms available for video providers. We present BOLA-E and evaluate it in Section 3.2.

Algorithm DYNAMIC Another approach to improving BOLA is to use a throughput-based ABR algorithm when the buffer level is low and then dynamically switch to BOLA when the buffer level is high. The rationale for this approach is that throughput-based ABR performs better in situations such as startup and seek when the buffer is low or empty. And BOLA performs better when the buffer levels are

sufficient large. DYNAMIC was also first released as part of dash.js version 2.6.0 on Sept 1st 2017 and has been in use by video providers since. DYNAMIC is currently the primary ABR algorithm in the DASH reference player and is turned on by default for video providers. We present DYNAMIC and evaluate it in Section 3.3.

Algorithm FAST SWITCHING We developed a technique called FAST SWITCHING that can be used with any ABR algorithm to improve video quality by *replacing* lower-bitrate segments in the client buffer with higher-bitrate segments. Consider a situation where a wireless client has downloaded a sequence of low-bitrate video segments when the connectivity was poor. Suppose now that the client's connectivity improves. FAST SWITCHING allows the client to replace the low-bitrate segments in the buffer by higher-bitrate segments that can now be downloaded with the improved connectivity. Thus, FAST SWITCHING allows the user to switch to higher-quality viewing sooner than it would have been otherwise possible. We implemented FAST SWITCHING in dash.js version 2.2.0 on July 6th 2016. FAST SWITCHING can be turned on by video providers in conjunction with any ABR algorithm, including the default DYNAMIC or the optional BOLA-E. We present FAST SWITCHING and evaluate it in Section 3.4.

We implemented and evaluated BOLA in the DASH reference player dash.js [13]. Our work has significantly improved dash.js, as noted by the dash.js community [18]. And, the algorithms described in this chapter are actively used by video providers (including Akamai, BBC, CBS, and Orange) in production, as they build their own video players based on the standard reference player.

3.1 Sabre: An Open-Source Tool for Simulating ABR Environments

An accurate simulation tool for ABR is critical for algorithm development. However, simulation results are not useful if the simulation tool does not reflect the

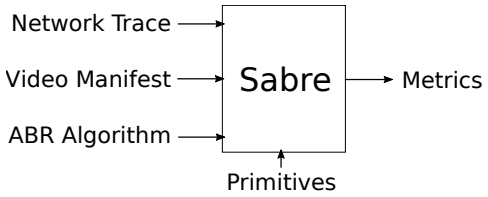


Figure 3.2. Sabre: Inputs, Outputs, and Primitives.

conditions of a practical player. We developed Sabre, an accurate tool for simulating ABR environments that can be used for designing and evaluating new ABR algorithms. For simulation accuracy, we based the design of Sabre on the architecture of the DASH reference player dash.js. However, other video players, such as Google's Shaka Player and the HLS player hls.js, are functionally similar to dash.js, allowing Sabre to be used as an effective tool for simulating other players as well. *We made Sabre open source and publicly available to the community on GitHub [48], so that others can use and continue to develop the tool. We also used Sabre to empirically evaluate algorithms presented in this chapter prior to their production implementation in dash.js.*

Using Sabre offers several major benefits. Playing a long video can be simulated in a fraction of the time, e.g., a one-hour video can be simulated in less than one second. Further, it is easy to simulate very specific network conditions in a reliable and reproducible way. In addition, it is possible to perform simulations at a large scale using several videos and thousands of network traces, as we do in our work.

To simulate video streaming, Sabre invokes the ABR algorithm before downloading a segment. The ABR algorithm provides the bitrate of the segment to be downloaded. If segment replacement is enabled, it also provides information on whether the segment to be downloaded is new or a replacement for an existing segment. As the segment is being downloaded, Sabre collects and periodically reports metrics to the ABR algorithm for use in its decision making. Similar to dash.js, Sabre allows abandonment of a segment download in progress. Further, dash.js uses the

XMLHTTPRequest progress events provided by the browser. Sabre simulates the progress events to allow simulation of segment abandonment strategies.

3.1.1 Inputs

The inputs to Sabre are described below.

(1) Network Trace. Sabre requires a network trace to simulate a video session. A trace should have a sequence of records where each record contains the time duration, and network throughput and latency for that duration. The traces allows reproducible simulation of real-world network conditions, facilitating comparison between different algorithms or between different settings for tuning a particular algorithm. The network traces can be measured from an actual system or they can be synthetic.

(2) Video Description. Sabre also requires a video description that is analogous to the DASH manifest. The video description includes the segment length (in seconds), the encoded bitrates, and a segment size matrix $C[i, j]$. $1 \leq i \leq N$, $1 \leq j \leq M$, where N is the total number of segments in the video and M is the number of encoded bitrates. The value of $C[i, j]$ represents the size (in bits) of the i^{th} segment of the video encoded at the j^{th} bitrate. By allowing the segment size matrix to be specified we enable Sabre to accurately simulate variable bitrate (VBR) videos. Note that the video description could represent an actual video or could be generated synthetically.

(3) ABR Algorithm. The ABR algorithm is invoked before downloading a new segment. The algorithms in this chapter such as BOLA-E and DYNAMIC are available with the Sabre software. However, the user may also develop their own ABR algorithms as Python modules and test them with Sabre.

3.1.2 Outputs

Sabre continuously collects and reports a detailed list of events and metrics such as bitrate, download time, and size of each downloaded segment, the duration of each

rebuffer event, each change in bitrate as the segments are played out, and all segment abandonments and replacements.

The Sabre output includes three important metrics that we use throughout the chapter. The *rebuffer ratio* is the fraction of time a video session spends in the rebuffer state. The rebuffer ratio equals the total rebuffer time divided by the sum of the total rebuffer time and the total play time. The *average bitrate* is the average of the encoded segment bitrate over all rendered segments. The *average bitrate oscillation* is the average difference in the bitrates of consecutively rendered segments. That is, the average oscillation equals

$$\text{oscillations} = \frac{1}{N-1} \sum_{i=1}^{N-1} \left| \text{bitrate}(i) - \text{bitrate}(i+1) \right| \tag{3.1}$$

where bitrate(i) is the encoded bitrate of the i^{th} rendered segment and N is the number of segments.

Note that the bitrate itself may not be directly proportional to QoE. For example, the QoE improvement obtained by upgrading a 1 Mbps video to a 2 Mbps video is much larger than the QoE improvement obtained by upgrading a 10 Mbps video to an 11 Mbps video, even though the bitrate increase is the same. However, the bitrate-to-QoE relationship is generally monotonic and increasing one increases the other. Thus, we use bitrate as a measure of QoE thoughout this chapter.

3.1.3 Primitives

Sabre also provides primitives that capture common functions that an ABR algorithm developer can use. Currently, we only offer three throughput estimation primitives. These primitives produce a network throughput estimate based on the history of past segment download times. The *sliding-window* throughput primitive produces an estimate by averaging the achieved throughput for the past k successful segment downloads, where k is the window size specified by the user. The *exponential-*

window throughput primitive produces an estimate by exponentially averaging the past downloads with a half-life of λ, where λ can be specified by the user. We also support the *dual-exponential* throughput primitive that uses the exponential-window throughput primitive with half-lives of λ_1 and λ_2 and takes the smaller of the two estimates. We support this primitive since it is used in Google's Shaka Player [21] and in the open source HLS player hls.js [12]. The implementation of Sabre is modular enough for the user to provide additional throughput or other primitives.

3.1.4 Caveats

Sabre does not simulate low-level protocols such as TCP, and relies on download traces collected by the player during real-world testing. Also, Sabre does not simulate low-level implementation details such as the exact behavior of the browser's Media Source Extensions buffer. However, omitting that level of detail does not significantly affect ABR algorithm performance.

Sabre does not simulate audio. The DASH standard requires that video and audio are delivered separately, and the audio download usually happens on a TCP session which runs parallel to the video download. Again, simulating the interaction accurately requires simulation of lower-level protocols. On the other hand, the size of the audio stream is usually only a small fraction of the size of the video stream, allowing a simple workaround. Consider an example video that is accompanied by a 160 kbps audio. We can reduce the network bandwidth available by 160 kbps throughout the network trace to simulate the video in Sabre.

3.1.5 Network traces used with Sabre in our work

1. *3G traces.* We use 3G traces from [43], a collection of 86 traces gathered in Norway using a 3G/HSDPA connection on trips by bus, metro, tram, ferry, car and train. The traces have a 1s granularity.

Table 3.1. Segment Bitrates for the Big Buck Bunny Movie

SD	Mean	6.00	5.03	2.96	2.06	1.43	0.99	0.69	0.48	0.33	0.23
Bitrate(Mbps)	Std. dev.	1.08	0.89	0.56	0.39	0.28	0.18	0.12	0.10	0.05	0.04
HD	Mean	35.0	16.0	8.0	5.0	2.5	1.0				
Bitrate(Mbps)	Std. dev.	6.3	2.8	1.5	1.0	0.5	0.2				

2. *4G traces.* We use 4G traces from [56], a similar collection of 40 traces gathered in Belgium using a 4G/LTE connection on trips by bicycle, bus, car, train, tram and on foot, with a 1s granularity.

3. *FCC traces.* The FCC provides a public set of broadband traces [11]. We obtain throughput traces from measurements in the web browsing category[1], with each data point representing the throughput for 5s. The traces have a 5s granularity.

3.1.6 Video descriptions for Sabre in our work

We used the Big Buck Bunny Movie [20], a ten minute movie, for our simulations. Table 3.1 shows the bitrates for both the SD and HD video descriptions, with the standard deviation caused by VBR. We use a standard definition (SD) encoding that is the same encoding used in Chapter 2 with ten bitrates ranging from 230 kbps to 6 Mbps, with a segment length of 3s. The input to Sabre contains the size in bits for each segment $C[i, j]$. We also generated a high definition (HD) video description with six bitrates[2] ranging from 1 Mbps to 35 Mbps by scaling the sizes of the SD video segments drawn from the highest six SD bitrates. Using this scaling, we obtained HD bitrates while still maintaining the VBR variability.

[1]We parse and use the traces in a manner similar to [32] (`https://github.com/hongzimao/pensieve/tree/master/traces/fcc`).

[2]We use the set of bitrates recommended for YouTube (`https://support.google.com/youtube/answer/1722171`)

3.1.7 Sabre Validation

For Sabre to be useful during development of ABR algorithms, we need to ensure that its results accurately predict the results that would be obtained by an actual real-world video player. In this section we evaluate how accurately Sabre emulates real-world video players such as dash.js.

We first ran 20 video session in dash.js. We used a step function to modulate the network throughput, spending two minutes each at 5, 10, 20, and 10 Mbps, then repeating from the start. By frequently modulating the throughput, we can test the accuracy of Sabre under a circumstance where the ABR algorithm is frequently switching bitrates. We used the Big Buck Bunny Movie that can be loaded in the reference dash.js player [13], a 10-minute, 34-second video with an encoding that uses ten bitrates ranging from 250 kbps to 15 Mbps. We selected the BOLA-E algorithm and set the buffer capacity to 25 seconds. For each video session, we recorded the throughput as seen by the player and three QoE metrics: rebuffer ratio, average bitrate and average bitrate oscillations.

To compare our results from the dash.js video player with Sabre, we simulated the same 20 video sessions in Sabre using BOLA-E and a 25-second buffer. To give Sabre a matching video description input, we measured the size of each video segment. Then, we generated the network trace inputs from the throughput measurements made during the corresponding dash.js sessions. After simulating each session, we compared the QoE metrics given by the Sabre simulation to the QoE metrics recorded in the corresponding dash.js session.

Fig. 3.3 shows the bitrates selected by the ABR algorithm for a typical session as measured on dash.js and Sabre. The actual player (dash.js) and the simulated player (Sabre) show similar bitrate switching behavior, as network conditions change in accordance with the step-modulated network throughput. Table 3.2 show the error between QoE measurements derived from dash.js and the corresponding Sabre session.

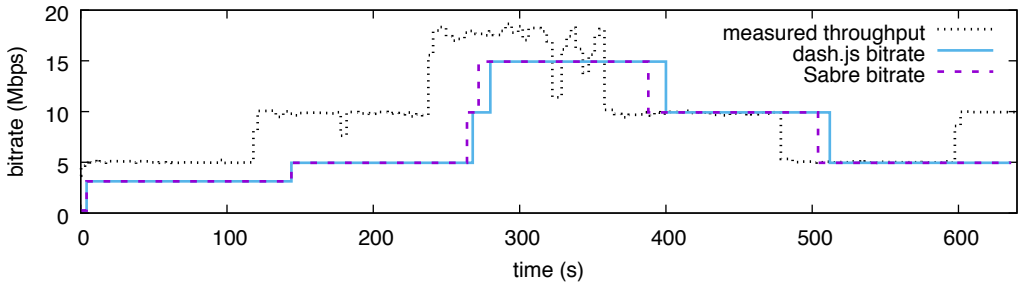


Figure 3.3. The bitrate of the segments downloaded and played by the dash.js player and by the Sabre simulator for a typical session. The throughput shown is what we measured in dash.js to be replayed by Sabre.

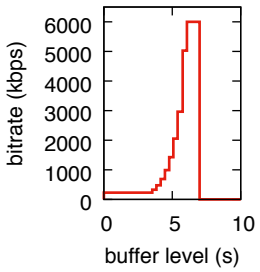
Table 3.2. The error in QoE metrics between dash.js and the corresponding Sabre measurements

Network condition	Rebuffer Ratio Error		Average Bitrate % Error		Average Oscillation % Error	
	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
Step	88×10^{-6}	307×10^{-6}	2.32	2.83	5.03	11.39
3G	14×10^{-3}	22×10^{-3}	1.88	1.68	5.25	7.01
4G	0	0	1.41	1.72	5.91	11.49

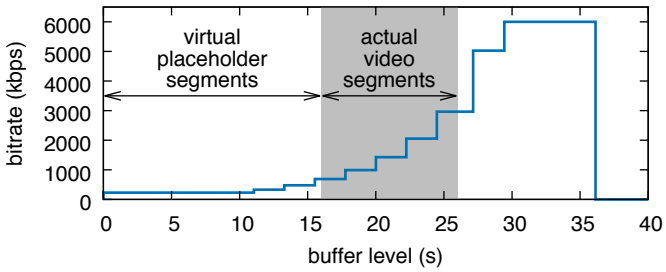
The average bitrate reported by Sabre has an average error of 2.3%, while the average bitrate oscillation has an average error of 5.0%. We also ran test sessions with network conditions corresponding to the 3G and 4G traces described in Section 3.1.5. For these traces as well, the QoE metrics given by Sabre closely match the metrics given by dash.js. Thus, Sabre produces QoE metrics that accurately reflect measurements from the real-world dash.js player.

3.2 BOLA-E: Enhancements to BOLA

Buffer-based ABR algorithms such as BOLA work best during steady-state conditions, but are not very responsive to *user events* such as startup and seeking. The buffer is usually empty at these events, and a naive buffer-based ABR algorithm might download many lower-bitrate segments before reaching a sufficient buffer level to download at the highest sustainable bitrate. A number of heuristics have been



(a) BOLA bitrate selection function for a 10s buffer capacity. Note that a segment length of 3s means that a download is not allowed if the buffer level is above 7s because it could lead to a buffer overflow.



(b) BOLA-E bitrate decision function for a 10s buffer capacity after buffer expansion. While the video buffer is still limited to 10s of actual segments, the placeholder segments can slide the 10s window, allowing more separation between thresholds.

Figure 3.4. Buffer expansion allows BOLA-E to have a larger separation between thresholds, reducing oscillations.

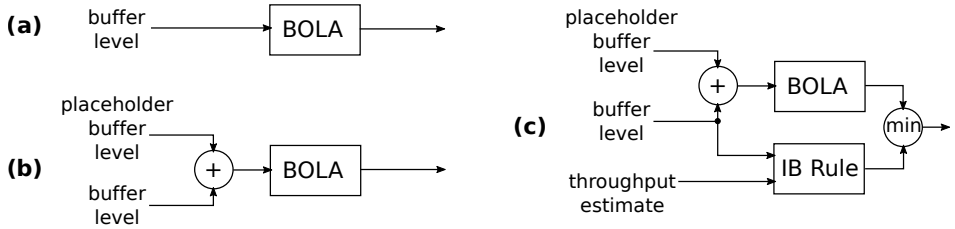


Figure 3.5. The evolution of BOLA-E. (a) The original BOLA. (b) Adding the placeholder algorithm for BOLA-PL. (c) Adding the insufficient buffer rule for BOLA-E.

proposed to mitigate slow startup in buffer-based algorithms [24, 51], but the heuristics still fall short of the performance achieved by throughput-based algorithms in the transient period. In Section 3.2.1 we design and implement the placeholder algorithm as an improvement to BOLA to overcome this issue.

Further, buffer-based algorithms require a sufficient buffer capacity for stable operation. However, this is not possible for live streams, such as live sporting events, that require low latency. In this case, the buffer capacity must be smaller than the latency bound that we are trying to achieve. If the buffer capacity is small, the thresholds between different bitrate choices get too close. Consider a typical video encoded at ten bitrates with a segment length of 3s being streamed to a video player with a 10s buffer capacity. This buffer capacity allows less than 1s separation between many consecutive thresholds as seen in Fig. 3.4(a). Even small segment size variability due to VBR could cause variability in the buffer level. With a small separation between thresholds, this buffer level variability would then be enough to make the ABR algorithm frequently switch between bitrates, causing excessive oscillations. In Section 3.2.2 we design and implement the insufficient buffer rule and use it, together with buffer expansion, to overcome this issue. Fig. 3.5 graphically shows how we put together the new algorithm BOLA-E from the original BOLA using the placeholder algorithm, and the insufficient buffer rule.

3.2.1 The Placeholder Algorithm

A fundamental problem with buffer-based algorithms is that the buffer level is not a good proxy for the available network throughput in certain situations. In particular, the buffer level underestimates or provides no information about the current throughput when the user starts up or seeks a video. In fact, in the case of a startup or a seek the buffer starts out empty. The main idea of the placeholder algorithm is that the buffer levels could be made to appear larger by judiciously inserting and

removing virtual *placeholder segments* in the buffer, as and when needed. The buffer level used for ABR decisions includes *both* placeholder and actual video segments. Note that placeholder segments have no video content and cannot be played out. They are used purely to manipulate the buffer level that is used for decision making by the ABR algorithm.

The placeholder algorithm improves responsiveness to startup and seek events by inserting placeholder segments using the following steps.

1. Obtain a throughput estimate.

2. Choose the appropriate bitrate corresponding to the throughput estimate derived in step (1).

3. Calculate the buffer level that would allow BOLA to pick the chosen bitrate. To do that, it uses the bitrate selection function used by BOLA, such as the one shown in Fig. 3.4(a). We can pick the buffer level (x-axis) that corresponds to the bitrate (y-axis) chosen in step (2).

4. Insert enough virtual placeholder segments in the buffer to obtain the desired buffer level. That is, the number of placeholder segments that are inserted equals the desired buffer level from step (3) minus the total size of the actual segments in the buffer.

Note that the algorithm needs to download one low-bitrate segment at startup to obtain a throughput estimate in step (1) above. However, in the case of seek, it will already have a good estimate available from prior segment downloads.

The placeholder algorithm also *removes* placeholder segments when a situation demands that the bitrate must be held steady and not stepped up. One such situation is when BOLA disallows switching up to a bitrate when such a switch is likely to be followed by a switch to a lower bitrate within a short time. In this situation, the

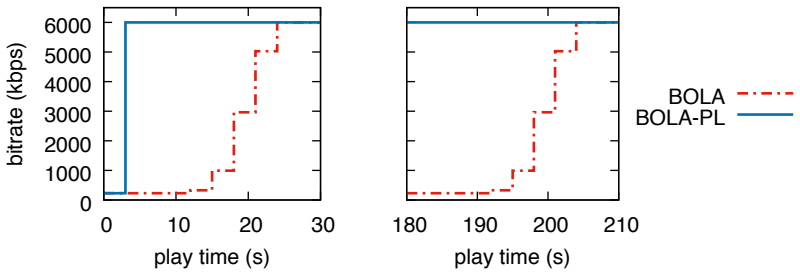
placeholder algorithm attempts to reduce the buffer level to the appropriate value by removing placeholder segments.

3.2.1.1 Evaluation

We now evaluate the placeholder algorithm for responsiveness to user events such as startups and seeks. First, we use a synthetic network trace that keeps the throughput relatively steady at 8 Mbps. We use the SD video described in Section 3.1.6. We then use Sabre to evaluate BOLA without the placeholder algorithm and BOLA-PL which is BOLA with the placeholder algorithm. Both algorithms use a buffer capacity of 25s in the evaluation. Ideally, the video should start playing as quickly as possible after the startup/seek event at a bitrate of 6 Mbps, which is the highest encoded bitrate of the SD video.

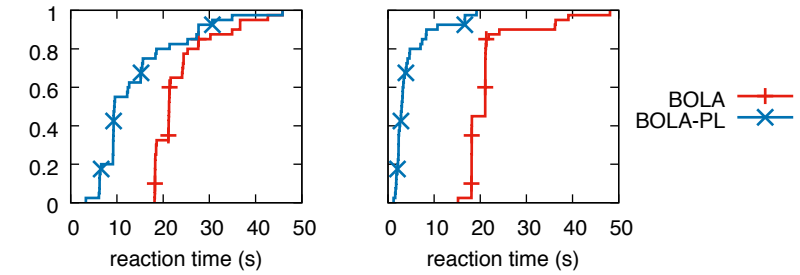
Fig. 3.6 evaluates BOLA-PL and BOLA for a startup event when the user clicks the start button and for a seek event where the user seeks to the 3-minute point in the video. BOLA-PL starts playing at the highest bitrate at 3.1s for the startup scenario. Specifically, it switched to high quality from the second segment onwards. That is because the placeholder algorithm needed the first segment download to obtain an initial throughput estimate. However, BOLA-PL started to play at the highest bitrate starting from the first segment after a seek, i.e., the high bitrate playback started after the 2.4s it took to complete downloading the first segment. On the other hand, BOLA is much less responsive for both startup and seek scenarios as it has to wait for the buffer level to rise before switching to the highest bitrate. In particular, it took BOLA 24.1s to switch the highest bitrate for both the startup and seek scenarios.

Fig. 3.7 compares the startup and seek performance of BOLA and BOLA-PL for the 4G traces described in Section 3.1.5. We repeated the startup and seek experiments for the HD video for each of the 40 traces and computed the CDF of the response time, where the response time is the time that it takes for the video to play



(a) Startup bitrate. (b) Seek bitrate.

Figure 3.6. Bitrate of the video playout as a function of the video play time. BOLA with the placeholder algorithm (BOLA-PL) reacts more quickly by reaching the highest sustainable bitrate within a much shorter period of time after a startup or a seek than BOLA alone.



(a) Startup reaction time. (b) Seek reaction time.

Figure 3.7. CDFs of the reaction time for BOLA versus BOLA-PL during startup and seek for 40 4G network traces. BOLA-PL reacts much more quickly and streams at the highest sustainable bitrate sooner than BOLA.

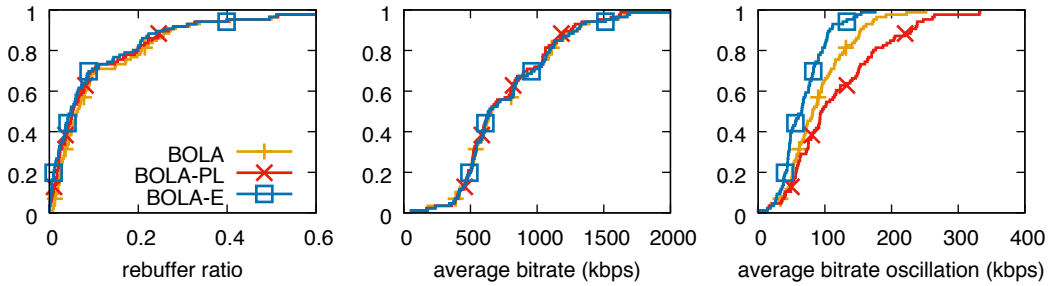


Figure 3.8. CDFs of the QoE metrics for BOLA, BOLA-PL and BOLA-E when streaming the SD video with a buffer capacity of 10s for 86 3G traces. BOLA-E significantly reduces oscillations.

at the highest sustainable bitrate. The median startup response time for BOLA-PL is 9.3s, whereas BOLA took much longer to respond at 21.3s. The median seek response time for BOLA-PL is 3.1s, BOLA again took much longer to respond at 21.1s.

3.2.2 Insufficient Buffer Rule

We now propose a solution to the problem of avoiding oscillations in low-latency live streaming. The low latency requirement implies that the buffer capacity must be small. For any buffer-based algorithm such as BOLA, this means that thresholds where the bitrate changes are made are close together, and even a small variance in segment size or network throughput can cause oscillations.

Using placeholder segments allows a novel approach to this problem by allowing the buffer capacity to be large, but still restricting the total size of the actual segments in the buffer to be no more than a small value. That is, we allow a large buffer with significant separation between thresholds for bitrate switching. However, we let only a small number of actual segments to be stored in the buffer, the remainder being placeholder segments. With this approach, the latency is kept small, since only the actual segments contribute to latency. Note that since only a few actual segments can be stored in a buffer of much larger size, there will be instances when there is enough space in the buffer and the network throughput is high enough for a segment to be downloaded. But, the algorithm must pause as a new segment is not yet available as it falls outside the latency window. In these instances, a placeholder segment is placed in buffer to indicate that an actual segment *could* have been downloaded if that segment was available.

We now illustrate the *buffer expansion* described above with an example. Fig. 3.4(a) shows an example of BOLA's bitrate switching thresholds for a low latency live stream with a buffer capacity of 10s. Fig. 3.4(b) shows how it can be "stretched" to a larger buffer by modifying BOLA's parameters V and γ. The thresholds in Fig. 3.4(b) are

at least 2s apart, reducing the potential for oscillations. However, the total size of the actual segments that can be stored in the buffer is still at most the original buffer capacity of 10s.

Unfortunately, buffer expansion results in a large buffer with many placeholder segments but with few actual segments. This can increase rebuffering, even though it cuts down on the oscillations. Placeholder segments induce BOLA-PL to download at a higher bitrate as shown in Fig. 3.4(b), but it can cause rebuffering when the video segments run out, as the placeholder segments are virtual and cannot be played out. We propose the insufficient buffer rule to solve this rebuffering issue. The rule verifies each ABR choice by BOLA-PL to make sure the download is unlikely to cause a rebuffering event using the following steps.

1. Multiply the current throughput estimate by 50% to obtain a *safe throughput.*

2. Multiply the safe throughput by the video buffer level (not counting the placeholder segments) to obtain a *safe download size.*

3. Limit the ABR choice to segments with size not larger than the safe download size, always allowing the lowest bitrate.

Combining a buffer-based algorithm with the placeholder algorithm and the insufficient buffer produces a hybrid algorithm which has the benefits of buffer-based algorithms while avoiding their usual drawbacks.

3.2.2.1 Evaluation

We now compare BOLA-E that includes the buffer expansion and the insufficient buffer rule with BOLA-PL that does not include either. First, we note that we empirically confirmed that the responsiveness of BOLA-E to startup and seek events are identically to that of BOLA-PL shown in Figs. 3.6 and 3.7. Next, we evaluated these algorithms on both 3G and 4G traces described in Section 3.1.5 with a 10s buffer

capacity to evaluate their potential for rebuffering and oscillations. Fig. 3.8 shows that BOLA-PL and BOLA-E have nearly identical rebuffering and average bitrate behavior for the 3G traces. However, BOLA-E that uses the buffer expansion and the insufficient buffer rule has much fewer oscillations than BOLA-PL. In particular, BOLA-E had a median bitrate oscillation of 65 kbps versus 95 kbps for BOLA-PL. We also included metrics for BOLA in Fig. 3.8 to show that, while BOLA-E improves reaction time, it does not degrade the steady-state QoE metrics. In fact, it reduces bitrate oscillations. Note that BOLA-PL without buffer expansion and without the insufficient buffer rule increases bitrate oscillations when compared to BOLA. The empirical results for 4G traces were similar to that of the 3G traces and we do not include them here for space limitations.

3.3 DYNAMIC: BOLA with THROUGHPUT

Section 3.2 introduced enhancements to the buffer-based algorithm BOLA to mitigate issues with startup, seek and low-latency streaming and created a new algorithm BOLA-E. In this section, we describe a different approach to mitigate the same issues, leading to the creation of DYNAMIC that is currently the default ABR algorithm in the DASH reference player dash.js (see Fig. 3.1).

We observed that throughput-based algorithms perform well in low-buffer-level situations, whereas buffer-based algorithms such as BOLA perform better at larger buffer levels. Thus, we propose the DYNAMIC algorithm that uses a simple throughput-based algorithm called THROUGHPUT when the buffer levels are low (such as during startup and seek events), and uses BOLA when the buffer levels are high as shown in Fig. 3.9.

THROUGHPUT is a simple heuristic that first estimates the network throughput by using the sliding-window primitive described in Section 3.1.3 and then picks the

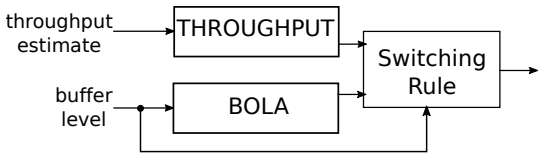


Figure 3.9. The DYNAMIC algorithms combines BOLA and THROUGHPUT.

highest encoded bitrate that is lower than a safety factor of 90% of the estimated throughput.

Algorithm DYNAMIC works as follows. At startup, DYNAMIC starts by invoking THROUGHPUT. At this stage, BOLA still prefers a bitrate that is too low. When the buffer level reaches 10s or more[3] and BOLA chooses a bitrate at least as high as the bitrate chosen by THROUGHPUT, DYNAMIC switches to BOLA. DYNAMIC switches back to THROUGHPUT when the buffer level falls below 10s and BOLA chooses a bitrate lower than THROUGHPUT.

3.3.1 Evaluation

First, we study the response time of DYNAMIC with respect to BOLA and THROUGHPUT for startup and seek events using a 25s buffer. Fig. 3.10 plots the CDF of the reaction time when the ABR algorithm reaches the highest sustainable bitrate. As expected, both THROUGHPUT and DYNAMIC provide fast response times, while BOLA responds slower since it needs to build up its buffer to a sufficient level to switch up to the highest sustainable bitrate. Note that the improvement in reaction time does not incur degradation in other QoE metrics, as shown in Fig. 3.11.

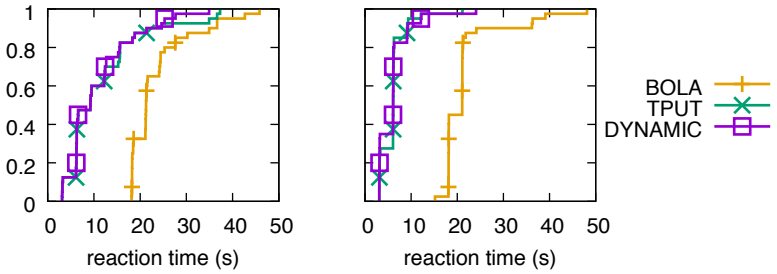
Fig. 3.11 compares DYNAMIC, BOLA and THROUGHPUT individually for two scenarios on 40 4G traces and plots the CDFs for the rebuffer ratio, average bitrate, and average bitrate oscillations. The first scenario, shown in Fig. 3.11(a), is

[3]We choose 10s because BOLA can have issues with lower buffer capacities (`https://github.com/Dash-Industry-Forum/dash.js/issues/1204`).

derived by simulating our HD video over 4G traces with a buffer capacity of 25s to emulate a typical VOD viewing experience. In this scenario, all three algorithms achieve similar rebuffer ratios. However, both BOLA and DYNAMIC achieve a greater throughput than THROUGHPUT. In particular, both BOLA and DYNAMIC achieve 19% and 22% more median throughput respectively than THROUGHPUT. Further, THROUGHPUT has more oscillations than either BOLA or DYNAMIC. In particular, at the 90^{th} percentile, both BOLA and DYNAMIC have less oscillations of 1301 kbps and 1421 kbps respectively, while THROUGHPUT has higher oscillations at 1929 kbps. In summary, for a typical VOD setting, both BOLA and DYNAMIC perform consistently better than THROUGHPUT.

The second scenario, shown in Fig. 3.11(b), evaluates the three algorithms over the 4G traces with a small 10s buffer to simulate a low-latency live streaming scenario. Note that with this low buffer capacity DYNAMIC can never cross the 10s buffer level threshold to select BOLA. In this scenario, all three algorithms achieve similar rebuffer ratios. BOLA achieves a greater throughput than THROUGHPUT and DYNAMIC. In particular, THROUGHPUT and DYNAMIC achieve 11% (i.e., 2049 kbps) less median throughput than BOLA. However, BOLA's high throughput comes at the cost of excessive oscillations. BOLA has more oscillations than THROUGHPUT and DYNAMIC. In particular, at the median value, THROUGHPUT and DYNAMIC have 1089 kbps bitrate oscillations, while BOLA has higher oscillations at 2465 kbps. In summary, for a typical live setting, THROUGHPUT and DYNAMIC perform consistently better than BOLA.

The main conclusion we can draw from our experiments is that, while BOLA works better in a VOD scenario with larger buffers, and THROUGHPUT works better for smaller buffer scenarios like low-latency live streaming, DYNAMIC combines the advantage of both and works well in both situations. DYNAMIC also provides a fast



(a) Startup reaction time. (b) Seek reaction time.

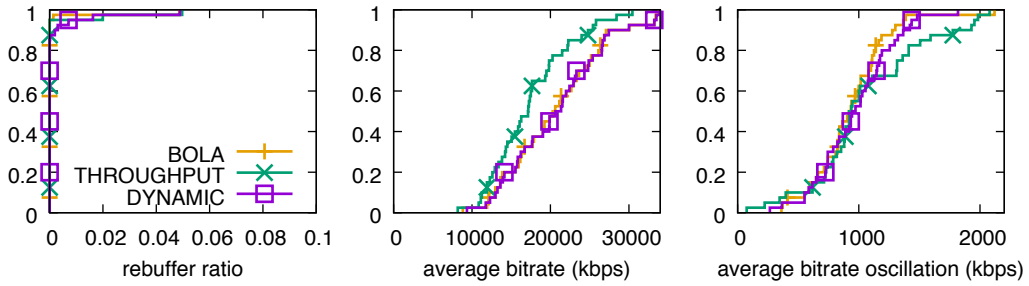
Figure 3.10. CDF of the reaction time for BOLA versus THROUGHPUT versus DYNAMIC during startup and seek for 40 4G network traces. THROUGHPUT and DYNAMIC react much more quickly and stream at the highest sustainable bitrate sooner than BOLA.

response in startup and seek scenarios, making it a good choice overall as an ABR algorithm.

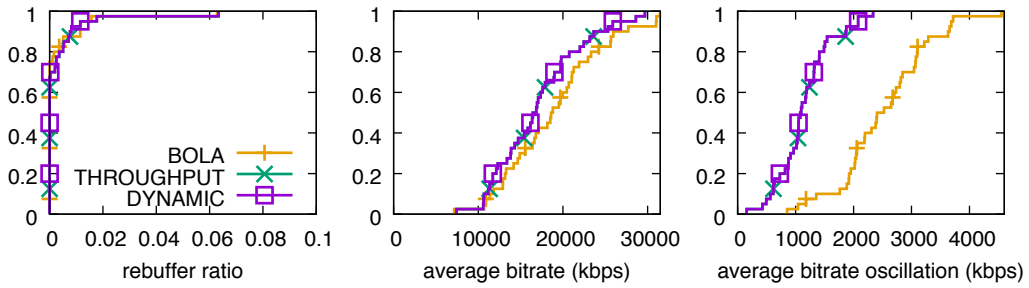
3.4 FAST SWITCHING: A Segment Replacement Algorithm

A large buffer can improve stability in ABR performance because it can absorb minor variations in network conditions, but it may deteriorate the video player responsiveness to network events. If the network throughput suddenly increases significantly, the ABR algorithm may download segments at a higher bitrate. However, the video player must first play out the low-bitrate segments that are *already* in the buffer before it can render the newly-fetched high-bitrate segments. The bigger the buffer capacity, the more low-bitrate segments it might hold, and the longer the wait before the user can switch to a higher quality.

We propose an algorithm called FAST SWITCHING that improves the video player responsiveness to higher network throughput by replacing segments already in the buffer. In particular, FAST SWITCHING allows video providers to have larger buffers for reasons of ABR stability, but yet have quicker response times to network events. We have found that FAST SWITCHING is particularly useful for



(a) VOD streaming with a 25s buffer.



(b) Live streaming with a 10s buffer.

Figure 3.11. DYNAMIC combines strengths of BOLA and THROUGHPUT: It has the higher bitrate of BOLA for VOD with a large buffer capacity and the low oscillations of THROUGHPUT for live streaming with a small buffer capacity.

video providers with longer VOD content, such as TV episodes and movies, where larger buffers are desirable and there is no low latency requirement. Note that FAST SWITCHING can be used with any bitrate selection strategy, including BOLA-E, THROUGHPUT, and DYNAMIC. In fact, the current implementation of dash.js allows FAST SWITCHING to be added to all three options.

FAST SWITCHING works using the following steps.

1. *Decide whether to download a new segment or a replacement segment.* Before downloading a segment, the algorithm invokes a bitrate selection algorithm (e.g., BOLA-E) to determine the bitrate b that can be used at the current time. If there is segment in the buffer with a bitrate lower than b and if such a segment can be *safely replaced,* then FAST SWITCHING decides that the next segment downloaded will be a replacement. Otherwise, the next segment downloaded will be a new segment that is appended to the end of the buffer. Intuitively, a segment *cannot* be safely replaced if it is too close to the play head and it will likely start to be played out before the replacement can be downloaded. That would result in a wasted download. FAST SWITCHING considers any segment that is scheduled to start rendering within the next $1.5 \times$ (the segment length) seconds to be *not* safely replaceable.

2. *Determine which segment to replace.* If it is determined in step (1) that a segment needs to be replaced, FAST SWITCHING downloads a replacement for the *earliest* segment in the buffer that is both safely replaceable and has a lower bitrate than the current bitrate b.

The choice of $1.5 \times$ (the segment length) in defining a safe replacement gives a 50% safety factor to account for possible variations in the download time due to network and/or segment size variability. Note also that FAST SWITCHING replaces segments in the earliest-deadline-first (EDF) order, starting from the segment that has the earliest deadline to be played out (i.e., closest to the play head). This ordering

may make it possible to replace more segments, since segments further down in the ordering have more time for replacement.

The FAST SWITCHING algorithm works with both throughput-based and buffer-based ABR algorithms. However, buffer-based algorithms might need adjustments. When FAST SWITCHING chooses to replace an existing segment, the buffer level is depleted since segments are being played out, but the buffer level is not increased by the downloaded segment. This lower buffer level might induce buffer-based ABR algorithms to choose a lower bitrate and thus increasing oscillations.

We handle this problem using two different approaches when integrating FAST SWITCHING with BOLA-E and DYNAMIC. BOLA-E inserts one placeholder segment in the buffer after every successful segment replacement. This solution does not work for DYNAMIC because it does not use the placeholder algorithm. Instead, DYNAMIC switches to THROUGHPUT whenever there is segment replacement, till the buffer level stabilizes, after which it can switch back to BOLA.

When using FAST SWITCHING, the player discards some lower-bitrate segments by replacing them with higher-bitrate segments, increasing the total bits downloaded by the client. In the SD example above, when the client experiences a 48% improvement in median average bitrate, 10% of the bits were downloaded and discarded by the client. However, bits are downloaded and discarded only for the short period of time when network throughput changes drastically and segment replacement is necessary. So, the overall impact of FAST SWITCHING on server-client traffic is less significant.

3.4.1 Evaluation

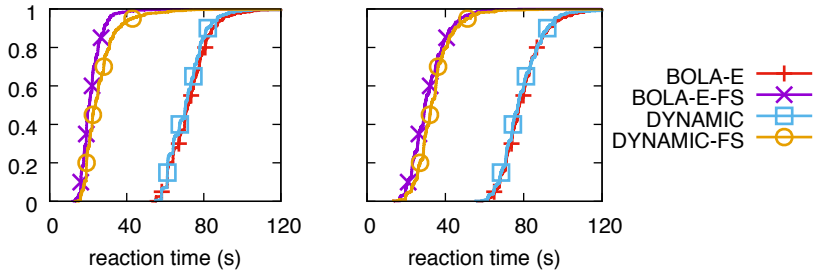
We now evaluate FAST SWITCHING by integrating it with both BOLA-E and DYNAMIC. To effectively simulate FAST SWITCHING, we need to generate scenarios where the network throughput increases. We use two videos for the simulation,

the SD video and the HD video described in Section 3.1.6. For the SD video, we use the FCC traces described in Section 3.1.5 to generate 1000 network traces, where each trace consists of 60s at a low average throughput less than 1 Mbps and 120s at a high average throughput between 6 and 12 Mbps. For the HD video, we use the FCC traces to generate 1000 network traces, where each trace consists of 60s at a low average throughput less than 2.5 Mbps and 120s at a high average throughput above 16 Mbps. For each video, the 1000 network traces were generated by randomly picking two traces with the desired properties from the FCC traces, one trace that is randomly picked for the low throughput period that is concatenated with another that is randomly picked for the high throughput period. The traces are picked randomly without replacement, so that we do not have the same trace picked twice and all the 1000 traces that are picked for a video are unique.

For each trace, some time after the network throughput increases, the viewer starts to see the video at a higher bitrate that can be sustained by the higher throughput. We measure the reaction time as the elapsed time from when the network throughput increased to when the user started viewing the video at the highest sustainable bitrate. Specifically, for the SD (resp., HD) video, we measure the time until the video starts rendering at 6 Mbps (resp., 16 Mbps). In both cases, we simulate a video player with a 25s buffer.

Fig. 3.12 shows the reaction times for BOLA-E and DYNAMIC with FAST SWITCHING, denoted by "BOLA-E-FS" and "DYNAMIC-FS" respectively, in comparison with BOLA-E and DYNAMIC by themselves. We can see that FAST SWITCHING improves the median reaction time by about 50s for both ABR algorithms and for both the SD and the HD videos. This means that the user will see a higher quality video about 50s earlier with FAST SWITCHING than without.

Note that the improvement of 50s is more than the buffer capacity of 25s. This is possible because there are two components to the reaction time for an ABR algorithm



(a) SD video. (b) HD video.

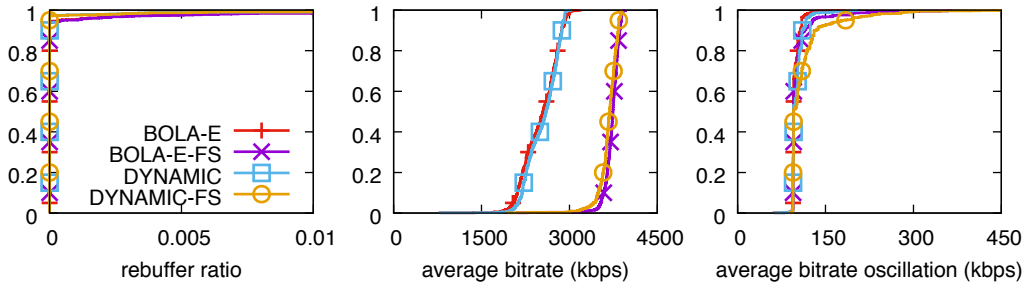
Figure 3.12. CDFs of reaction time when increasing the network throughput using BOLA-E and DYNAMIC with and without FAST SWITCHING using a 25s buffer. The reaction time shows how long it takes to start rendering at the highest sustainable bitrate after the network throughput increases.

without FAST SWITCHING. First, the ABR algorithm needs to determine that the throughput has increased and choose the corresponding higher bitrate. Second, the low-bitrate segments already in the buffer need to be played out before the new higher-bitrate segments can be played. FAST SWITCHING mitigates both components of the reaction time, as it can replace low-bitrate segments downloaded in both phases.

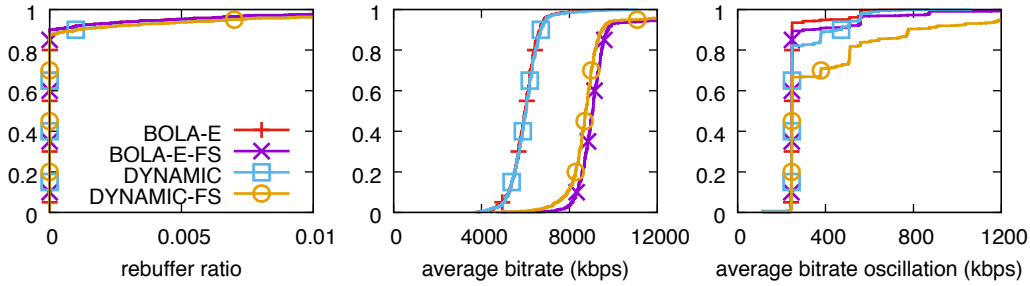
Since FAST SWITCHING switches to a higher bitrate sooner, we expect it to also improve the average bitrate for the above experiments. In fact, the middle column in Fig. 3.13 shows it gives a significant improvement. It improves the median bitrate by about 45% for both ABR algorithms and for both the SD and HD videos.

Fig. 3.13 also shows that for the experiments above, for all cases FAST SWITCHING does not noticeably increase rebuffering and for most cases it does not significantly increase bitrate oscillations. It only increases the bitrate oscillations significantly for some of the DYNAMIC tests for the HD video. In particular, at the 90^{th} percentile, it increases bitrate oscillations by 306 kbps.

Note that while the QoE metrics for BOLA-E and DYNAMIC are similar, there is a noticable differene in bitrate oscillations in Fig. 3.13 (b). FAST SWITCHING causes the buffer level to drop, leading DYNAMIC to switch to the THROUGHPUT algo-



(a) Streaming the SD video.



(b) Streaming the HD video.

Figure 3.13. CDFs of QoE metrics with and without FAST SWITCHING using a 25s buffer.

rithm. Since the network traces have high bandwidth variability, the THROUGHPUT algorithm gives higher bitrate oscillations. This problem does not affect BOLA-E.

A drawback of FAST SWITCHING is download overhead. When a replacement segment is downloaded, the previously-downloaded segment is discarded and never played back to the viewer. Thus such an overhead induces additional bandwidth costs. To measure the overhead, we tested FAST SWITCHING with the 4G traces; the 4G traces have throughput variations occurring at frequencies that are typical in real-world scenarios. Fig. 3.14 shows the discarded fraction; the discarded fraction is the number of discarded bits as a fraction of the total downloaded bits for a session. The median discarded fraction is 6%. The benefits obtained from this overhead are faster reactions times and an improvement in the average bitrate, with a 5% improvement in bitrate in the median case.

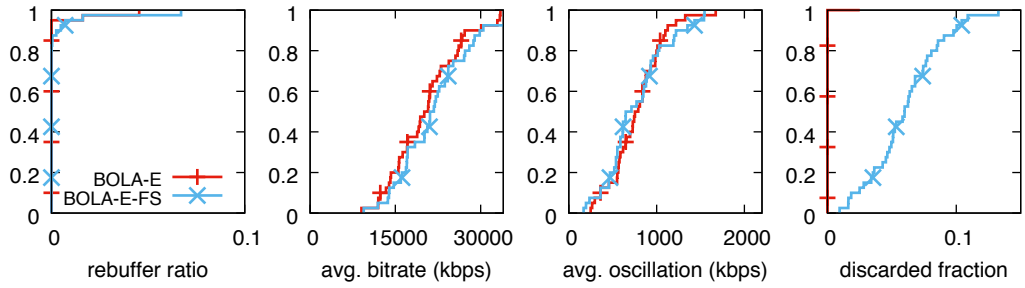


Figure 3.14. CDFs of QoE metrics with FAST SWITCHING using 4G traces. The discarded fraction is the fraction of total downloaded bits that were replaced and hence never played back to the viewer.

3.4.2 Rationale for the design choices made in FAST SWITCHING

Figs. 3.12–3.13 show that using FAST SWITCHING improves the reaction time to network events and consequently also improves the QoE metrics. In this section, we outline and evaluate the different design choices that are possible within the context of segment replacement and justify the choices made in FAST SWITCHING.

1) Replacement offset. FAST SWITCHING replaces segments in an earliest-deadline-first (EDF) fashion, starting with segments that are closest to the play head. However, starting with segments that are "too close" to the play head entails the risk of the segment being played out before the download of the replacement is complete. Therefore, we use a time offset to determine the distance (in seconds) from the play head that a segment must have to be considered to be safely replaceable. We experimented with different values for the offset including 0×, 1.5×, and 3 × (the segment length) using the FCC traces used in Section 3.4.1 that step up from low throughput to high throughput. A 0× offset means that the first segment after the currently playing segment is replaced. Fig. 3.15 shows the QoE metrics for the different factors when using BOLA-E with FAST SWITCHING for the SD video. A 0× offset introduces excessive oscillations and causes a significant drop in average bitrate for half the traces. On the other hand, a 3× offset is slow to react and does not get the full average bitrate improvement from replacement. We also ran tests for other videos

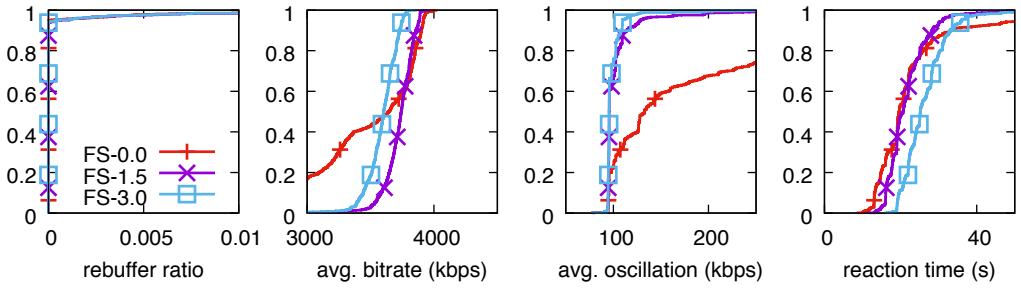


Figure 3.15. CDF of QoE metrics of BOLA-E with FAST SWITCHING with different replacement offsets.

such a HD video, and other ABR algorithms such as DYNAMIC and other replacement offsets between 0× and 3×. We found that a 1.5× offset is the sweet spot. It is the lowest factor that consistently has oscillations as low as the 3× offset, but has a faster reaction time and higher average bitrate. Thus, we choose a replacement offset of 1.5× in our production implementation of FAST SWITCHING.

2) Replacement order. Besides EDF, we explored other ways of ordering the segments that need to be replaced. In particular, we evaluated latest-deadline-first (LDF) order where replacement starts from the segments that are farthest from the play head. We also evaluated lowest-quality-first (LQF) where the lowest quality segments in the buffer are replaced first, though no segment that is within a replacement offset of 1.5 × (the segment length) can be replaced. When there are several LQF candidates with the same low bitrate, we replaced the segment that would give the lowest oscillation metric after replacement. We evaluated and compared EDF, LDF, and LQF empirically, again with the FCC network traces used in Section 3.4.1. Fig. 3.16 shows the QoE metrics for the different replacement orders when using BOLA-E with FAST SWITCHING for the SD video. EDF reacts faster than LDF and LQF. For example, the median reaction time for EDF is 21s (resp., 33s), whereas LDF took 24s (resp., 42s) and LQF took 22s(resp., 36s) when playing the SD (resp., HD) video. Consequently, we chose EDF for our production implementation of dash.js.

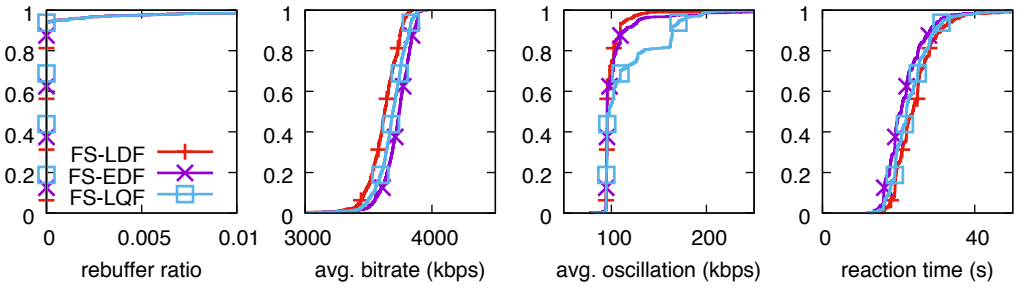


Figure 3.16. CDF of QoE metrics of BOLA-E with FAST SWITCHING for different replacement orders.

3.5 Related Work

We already explored related work in ABR algorithms in Section 2.6. Regarding evaluation tools, MACI [53] provides an emulation environment that allows automatic loading and evaluation of complete players such as dash.js. MACI and Sabre serve complementary roles in ABR algorithm development. MACI is a complete environment that plays the videos, while Sabre is a simulation environment that can more quickly test ABR algorithm for wide range of videos and network traces without actually playing the videos.

3.6 Conclusion

We designed and implemented Sabre, an open-source publicly-available tool that can be used by researchers to run accurate simulations of ABR algorithms using a player architecture similar to dash.js. We used Sabre to design and test BOLA-E and DYNAMIC, two algorithms that enhance the buffer-based ABR algorithm BOLA. We also developed a FAST SWITCHING algorithm that can replace segments that have already been downloaded with higher-bitrate (thus higher-quality) segments. The new algorithms provide higher QoE to the user in terms of higher bitrate, fewer rebuffers, and lesser bitrate oscillations. In addition, these algorithms react faster to user events such as startup and seek, and respond more quickly to network events

such as improvements in throughput. Further, they perform well for live streams that require low latency, a challenging problem for ABR algorithms, since the client buffer needs to be kept very small. Overall, the algorithms presented in our chapter offers superior video QoE and responsiveness for real-life adaptive video streaming. All three algorithms presented in this chapter are now part of the official DASH reference player dash.js and are being used by video providers in production environments. Through dash.js, BOLA is now being used in production by several major video providers and delivery networks such as Akamai, BBC, CBS and Orange. For video providers wanting to choose BOLA-E versus DYNAMIC in the production player, we also compared them head-to-head. BOLA-E is slightly better than DYNAMIC when the network bandwidth has higher variability, a situation that is also very challenging for the THROUGHPUT algorithm. On the other hand, DYNAMIC is slightly better for small buffer capacities, a situation that can be challenging for all buffer-based algorithms including BOLA-E. While DYNAMIC is currently the default choice, we found both algorithms performed similarly well in the QoE and responsiveness metrics.

CHAPTER 4

ADAPTIVE VIDEO STREAMING OVER LOSSY TRANSPORT

While early video streaming systems used the Internet as a best-effort system without reliable transport [57], modern video provides use HTTP adaptive streaming (HAS) as the default video delivery method [46]. A benefit of HTTP and the underlying TCP protocol is that HAS systems have reliable transport. However, TCP's reliability comes at a price. Lost packet retransmissions introduce delays and can cause head-of-line blocking.

A middle ground between best-effort transport and reliable transport is selective reliability such as the technique proposed by Feamster et al. [17], where some frames are delivered over reliable transport and other frames are delivered over unreliable transport. However, TCP does not support such a method.

QUIC is an alternative to TCP for HAS. Streaming video using QUIC can have several performance benefits [28]. QUIC does not have the three-way SYN handshake and needs less round trips to establish a secure connection, giving a faster video startup time. However, QUIC's reliable transport still has the same problems as TCP.

We propose VOXEL [37], a new video streaming system that merges partially reliable transport with HAS. While we build on previous ideas, to our knowledge VOXEL is the first system that combines reliable and unreliable transport with adaptive bitrate (ABR) streaming over HTTP. VOXEL can tolerate two types of loss. First, it allows some packet loss in non-key frames, similar to [17]. Second, it can completely

drop some frames within a segment. Both types of loss can be allowed to improve the overall experience, mostly by avoiding rebuffering events. VOXEL also allows ABR algorithms to exploit the selective reliability. Our primary contribution is an extension to BOLA-E that exploits the new transport capabilities to improve QoE.

4.1 VOXEL: system considerations

Several video streaming solutions exist that either tune the transport protocol [17, 41] or optimize the application layer, i.e., the ABR algorithm [32, 51, 61]. With VOXEL, we propose a cross-layer optimization approach. We now describe some key ideas behind the development of VOXEL.

Lossy video delivery can still achieve high QoE

Video content can allow the loss of some frames without a significant impact on QoE. Further, the QoE penalty of dropping a frame depends on the relative importance of the frame within the semgent rather than on the frame size [60]. When a frame is dropped, the overall QoE drop depends on how many other frames in the segment have references to the dropped frame because errors propagate through such references. Note that the key I-Frame in each segment does not depend on any other frame, and most other frames depend directly or indirectly on the I-Frame unless there is a scene change in the segment.

Video content can also tolerate packet loss within frames, where only a fraction of a frame is lost [17]. While header information can render the whole frame unusable, non-header information loss can be tolerated.

Complex relationship between bitrate and QoE

Modern ABR algorithms strive to optimize QoE indirectly by optimizing metrics such as bitrate [32, 51, 61]. The algorithms generally use some function of bitrate such as the logarithmic function in (2.21) to give diminishing returns when increasing

bitrate. However, the relationship between metrics such as bitrate and the QoE is complex and any such function only approximates the QoE. When exploring lossy video delivery, the metrics such as bitrate cannot capture the relative importance of different frames within a segment because the frame size in bits does not capture information such as inter-frame dependency.

Three common QoE metrics are peak signal to noise ratio (PSNR) [25], structural similarity (SSIM) [58] and video multimethod assesment fusion (VMAF) [35]. We use SSIM for our discussion because it has better correlation with the mean opinion score (MOS) than PSNR [58]. VMAF has even better correlation than SSIM with regards to compression and scaling artifacts, but VMAF does not support packet loss [35]. Romaniak et al. [44] show that SSIM can capture artifacts caused by packet loss or frame drops.

Dropping frames can bridge gap between bitrate levels

ABR algorithms support switching between different bitrates at segment boundaries. However, there are some drawbacks to having too many bitrate levels. First, having more bitrate levels requires more transcoding at the content preparation phase and, more importantly, uses more storage. Second, having more bitrate levels lowers the probability of cache hits because it is more likely that different clients are streaming at different bitrates.

One way to bridge the gap between two bitrates is to drop some less-important frames from segments at the higher bitrate. While ABR algorithms do not typically download segments at a bitrate higher than the estimate network throughput, allowing frame drops gives the client more flexibility to utilize more of the available bandwidth. Also, a client can decide to drop some frames from a segment while the segment is in the process of being downloaded. Thus, the client can download at

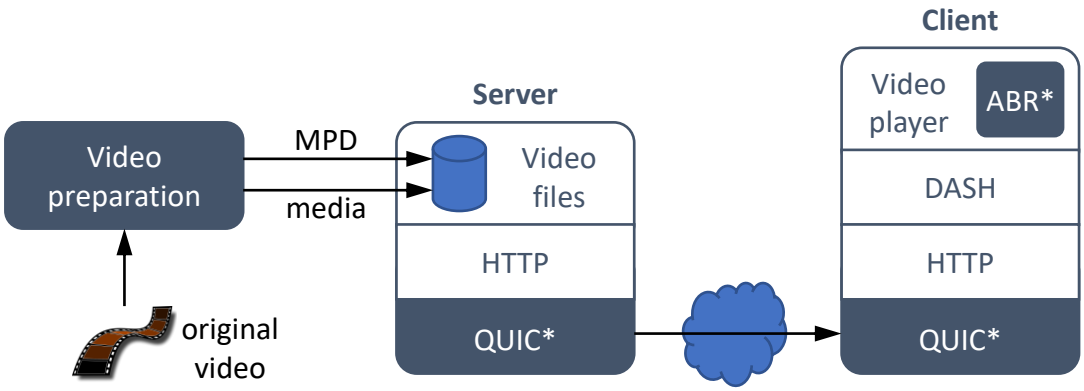


Figure 4.1. Video streaming using VOXEL. VOXEL differs from traditional streaming in three areas: video preparation, the QUIC* transport layer, and the ABR* adaptation algorithm.

a higher bitrate knowing that if network conditions deteriorate, rebuffering can be avoided by dropping some frames.

4.2 VOXEL: system architecture

We now describe VOXEL, a video streaming system we develop around the ideas in Section 4.1. Figure 4.1 shows the main VOXEL components. While similar to traditional systems, VOXEL has three important differences.

1. We prepare video content for streaming and identify the relative importantance of frames in each video segment. We rearrange the frames in each segment in order of importance.

2. At the transport layer, we build on the QUIC extensions from [38] to offer a partially reliable transport, allowing us to exploit the observation that not all frames require reliable delivery.

3. At the application layer, we design an ABR algorithm that uses the information from the video preparation stage and interects closely with the transport layer to achieve higher QoE.

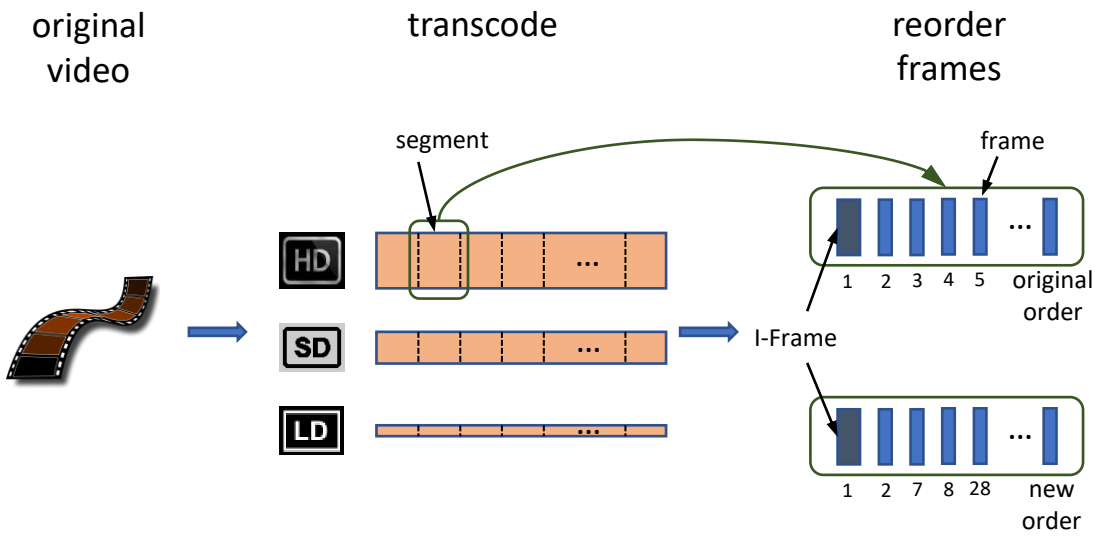


Figure 4.2. Preparing video content for VOXEL.

We describe each module in greater detail below.

VOXEL is backwards compatible with existing clients. A VOXEL-unaware client can still stream video from a VOXEL server. Such a client will request the stream over traditional reliable transport. Also, a VOXEL-aware client can stream video using a VOXEL-unaware ABR algorithm. Such a client will receive some but not all of VOXEL's benefits. The support for different degrees of VOXEL awareness makes it easy to incremental deploy VOXEL in the Internet.

4.2.1 Preparing the Video Content

Figure 4.2 shows how VOXEL prepares video content. The first step is transcoding the video, similar to most common HAS solutions. We transcode the original video into a number of different bitrates and partition each transcoded video into segments which can be presented using a typical DASH manifest. We then add an extra step to prioritize frames within each segment. This allows clients to download frames in order of importance and improves the QoE obtained by a partially downloaded segment when a segment download is interrupted.

For each segment, we try three different prioritization orders. The I-Frame has the highest importance in all orders and is always downloaded first.

1. ***Original order*** in which we deliver the segments in regular (MPEG) decoding order.

2. ***Order by inbound references*** where we rank frames using the number of inbound references from other frames. We then reorder the frames, and the frames with fewer inbound references are moved to the end of the segment. We count both direct and indirect inbound references because frame dependency can propagate through indirect references.

3. ***Order by grouping unreferenced frames*** where we move frames that have no inbound references from other frames to the end of the segment. We leave frames with inbound references in the original order, and we also preserve the relative order of frames with no inbound references. This order is an intermediate step between the first two orders.

We select an order from the three for each individual segment independently. To select an order, we analyze the drop in QoE when partially downloading a segment for each of the three orders. For each order, we drop the frames in the segment starting from the end, and we calculate the SSIM score for each iteration. This process gives us a table mapping the number of bits downloaded to the SSIM score for each order for each segment. When analyzing a segment at bitrate level m, we use the SSIM score for the complete segment at the next lower bitrate level $m-1$ as a reference SSIM. For each order, we look up the partial download size at quality m that obtains the reference SSIM. We select the order that has the smallest size.

We include the frame order along with the table of sizes and corresponding SSIM scores as metadata in the DASH manifest. For the experiments in Section 4.3, the additional metadata increases the manifest file size by a factor of approximately 8.

However, the manifest file size is still much smaller than the size of the media segments. VOXEL-aware clients can exploit this metadata and download the required frames in order of importance. VOXEL-unaware clients can simply ignore this metadata and download the segments in regular decoding order.

4.2.2 QUIC*: Enriching the Transport Layer

VOXEL's transport layer builds on the QUIC extensions proposed in [38]. While a separate control stream was used in [38], we use HTTP headers to control the media stream. We designed a modified version of QUIC, which we call QUIC*. QUIC* supports the reliable streams in vanilla QUIC and also supports unreliable streams without retransmissions. Unlike raw UDP, the unreliable streams in QUIC* still use the congestion and flow control mechanisms of the QUIC connection. A client can open a reliable stream by simply sending a HTTP GET request. A client that wants to open an unreliable stream sends a HTTP GET request but includes the custom HTTP header *x-voxel-unreliable*. A VOXEL-unaware client will not use the custom header and will only use reliable streams. This backward compatibility allows incremental deployment.

During a video session, a VOXEL client uses two HTTP requests per segment. The first request obtains the I-Frame and header information for all frames over a reliable stream. The second request obtains the video data for the remaining frames over an unreliable stream. Packets lost on the unreliable stream are not retransmitted, leading to some QoE deterioration. Note that frames normally span multiple packets, and a packet loss does not translate directly to a dropped frame but a partially corrupted frame. While unreliable transport can lead to packet loss, it also has the benefit of avoiding head-of-line blocking and stream interruption. Since the headers are delivered reliably, VOXEL can preserve the data structure integrity by filling holes with zero padding. This leads to artifacts related to data loss, but the video player

can still use the rest of the frame data. Our experiments in Section 4.3 show that VOXEL significantly reduces rebuffering, particularly in scenarios with smaller buffer sizes that are typical in real time live-streaming like scenarios.

VOXEL also exploits typical video-client behavior to selectively retransmit lost data on the unreliable streams. Video clients have periods with pauses in the download process. Such pauses can happen when the buffer level is at maximum capacity. VOXEL uses any such periods to retransmit packets lost on the unreliable stream.

4.2.3 ABR*: Enhancing the ABR Algorithm

VOXEL provides a framework for a new class of ABR algorithms with the following key features.

Optimize for QoE. ABR algorithms such as BOLA optimize a utility function typically based on bitrate. Such algorithms normally allow different utility functions. VOXEL uses a QoE-metric-based utility. We use SSIM in our evaluation, but *VOXEL is QoE-metric agnostic.* Experimental results generalize to other metrics such as VMAF and PSNR [37].

Support partial-segment downloads. Traditional ABR algorithms choose a bitrate *at segment boundaries.* They can choose from a limited set of bitrates. VOXEL allows partial segment downloads and presents the respective QoE metrics for different download subsets, thereby significantly increasing the available decision space.

Segment abandonment options. Traditional ABR algorithms might abandon a segment *while downloading the segment.* They abandon a high-bitrate segment download to start a low-bitrate download if they detect a high risk of rebuffering. VOXEL introduces another option: stop the download, but keep the partial segment and move on to the next. The new option downloads fewer bits than traditional ABR

algorithms during periods with poor network conditions. Also, a partial high-bitrate segment might give better QoE than a complete low-bitrate segment.

To complete our VOXEL implementation, we develop ABR*, a novel ABR algorithm based on BOLA-E. BOLA-E already supports low-buffer scenarios, and preliminary tests with unmodified BOLA-E over QUIC* showed that BOLA-E performed slightly better over QUIC* when compared with BOLA-E over QUIC [37]. This makes BOLA-E a good base algorithm on which to build ABR*.

4.3 ABR*: Extending BOLA-E for VOXEL

In Chapter 3, we developed BOLA-E that supports low buffer scenarios. However, as with any ABR algorithm, network condition deterioration will cause BOLA-E to drop the bitrate and eventually rebuffer. By changing the transport layer from QUIC to QUIC*, we can avoid some packet retransmissions. This reduces stream interruptions at the cost of some limited video corruption, with an overall improvement in QoE [37]. However, cross-layer optimization can allow an even better improvment. We now present a sequence of enhancements to BOLA-E leading to ABR*-O, an ABR algorithm that exploits the rest of the VOXEL system to farther improve QoE.

Experimental methodology

To evaluate the algorithms, we use a set of 86 3G traces [43]. We use the 3G traces because we want to explore VOXEL with poor network conditions. While we expect most ABR algorithms to perform similarly well with good network conditions, we want to design ABR algorithms that take longer to incur QoE degradation as network conditions deteriorate.

Since Sabre does not support transport layer simulation, we test our algorithms on physical machines. Each video simulation session is run on three bare-metal machines running Linux: one machine emulates the server, one machine emulates an

intermediate router, and one machine emulates the client machine. We shape the traffic flow through the router using the Linux traffic control utility tc, changing the bandwidth every second to match the network trace in use. We test the ABR algorithms using four different videos: Big Buck Bunny (BBB), Elephants Dream (ED), Sintel, and Tears of Steel (ToS). We transcoded each video at 14 bitrates, with resolutions ranging from 144p to 2160p and bitrates ranging from 160 kbps to 10 Mbps. We partitioned the videos in 4 second segments, and we selected a 5 minute excerpt from each video.

4.3.1 BOLA-SSIM: Incorporating SSIM utility and partial downloads

The VOXEL content preparation phase provides each segment at several bitrates similar to traditional HAS systems. VOXEL also gives a table of cut-off points in each segment with the SSIM score for each cut-off point. However, BOLA-E only considers downloading the segment at each bitrate. We first updated BOLA-E to expand the decision space. Each bitrate representation can be either downloaded whole or downloaded until any one of the cut-off points. Since we are trying to exploit the different relative importance of frames, the size-based utility function no longer works. Thus, we updated BOLA-E to set the utility function to the SSIM score. We call the algorithm after the two updates BOLA-SSIM.

4.3.1.1 Evaluation

We ran video sessions with BOLA-E and BOLA-SSIM, both with a buffer capacity of 8s, over the 86 3G traces using each of the four videos. Note that with a buffer capacity of 8s, the player does not start downloading a new 4s segment if the buffer level is above 4s. Figure 4.3 shows the average rebuffer ratio (time spent rebuffering divided by total time) and SSIM score for the sessions. We will describe ABR*, the third algorithm shown in the Figure, in Section 4.3.2. Figure 4.3 also shows the 90^{th} percentile rebuffer ratio and the 10^{th} percentile SSIM ratio to compare the algorithms

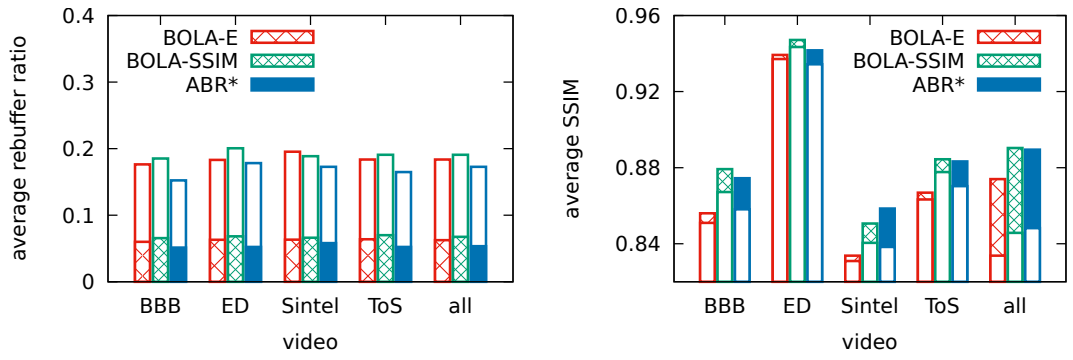


Figure 4.3. The average rebuffer ratio and average SSIM over 86 3G traces for BOLA-E, BOLA-SSIM and ABR* with a buffer capacity of 8s for four videos individually and together (all). The rebuffer ratio plot displays the mean (pattern) and 90^{th} percentile (plain). The SSIM plot displays the mean (pattern) and 10^{th} percentile (plain).

at poor network conditions. BOLA-SSIM obtains a higher SSIM score across all videos when compared with BOLA-E. However, this comes at the cost of a small increase in rebuffering. BOLA-SSIM has an advantage of optimizing directly for SSIM. Also, the increased download options allow BOLA-SSIM to be more aggressive. If the estimate sustainable throughput lies between two bitrates, BOLA-E might download at the lower bitrate of the two. On the other hand, BOLA-SSIM can choose the higher bitrate with one or more dropped frames. While BOLA-SSIM aggressively utilize more of the bandwidth, it runs a higher risk of rebuffering.

We repeated the experiments with a buffer capacity of 32s, plotting the results in Figure 4.4. Again, BOLA-SSIM obtains a higher SSIM score at the cost of a small increase in rebuffering. Note that the larger buffer gives lower rebuffering and higher SSIM across all videos and all algorithms as expected.

4.3.2 ABR*: Enhanced download abandonment

BOLA-SSIM uses the partial segment downloads provided by VOXEL to select a new download. We can also use the partial segment downloads while abandoning a segment during a download. Recall that download abandonment happens when

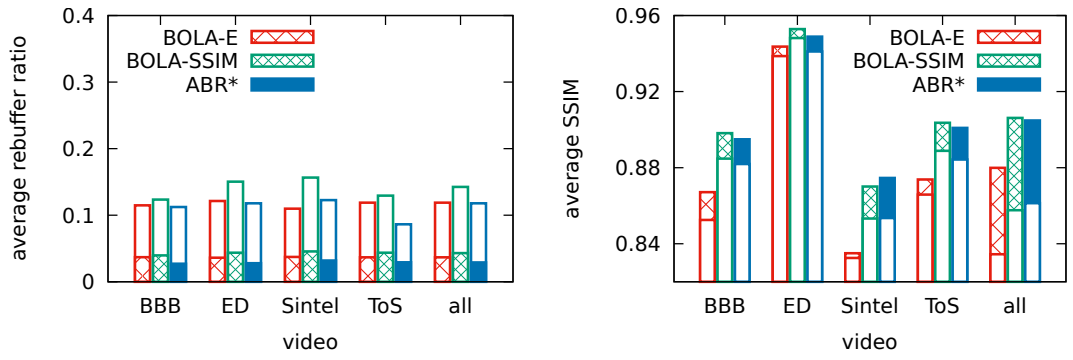


Figure 4.4. The average rebuffer ratio and average SSIM over 86 3G traces for BOLA-E, BOLA-SSIM and ABR* with a buffer capacity of 32s for four videos individually and together (all). The rebuffer ratio plot displays the mean (pattern) and 90^{th} percentile (plain). The SSIM plot displays the mean (pattern) and 10^{th} percentile (plain).

a download is taking too long and has a high risk of rebuffering before completion. While BOLA-E and BOLA-SSIM can abandon a download to download a lower bitrate, VOXEL provides another option. The part of the download already complete can be used as a partial segment. Thus, there is no need to restart the download at a lower bitrate. We call the algorithm after this update ABR*.

4.3.2.1 Evaluation

We repeated the experiments from Section 4.3.1 for ABR*. Figures 4.3–4.4 show the results for ABR*. While ABR* still obtains a higher SSIM score than BOLA-E, it obtains a slightly lower SSIM score than BOLA-SSIM. This was expected because ABR* might decide to interrupt a download and obtain a lower SSIM rather than finishing the download and causing a rebuffering event. On the other hand, ABR* has lower rebuffering than both BOLA-E and BOLA-SSIM.

Figure 4.5 shows the CDFs for the rebuffer ratio and SSIM corresponding to the BBB bars in Figure 4.3. Note the behavior at the most challenging traces. While the SSIM score for BOLA-E does not go below 0.850, the SSIM score for ABR* is 0.792 for the worst case. However, while the rebuffer ratio for ABR* goes up to a maximum

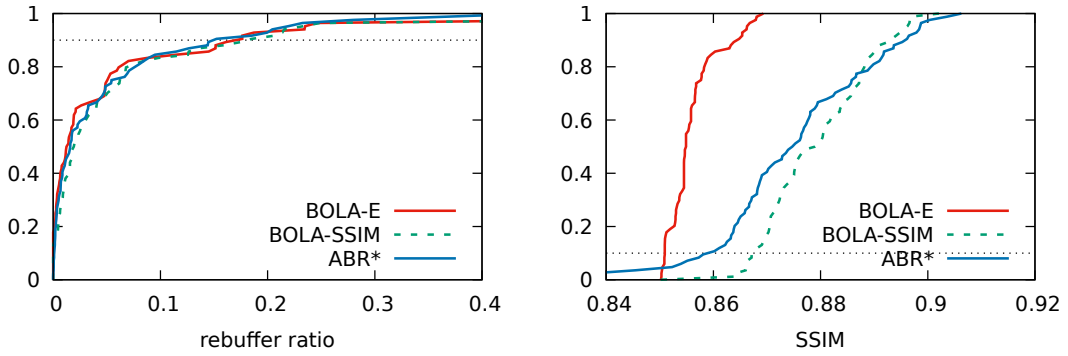


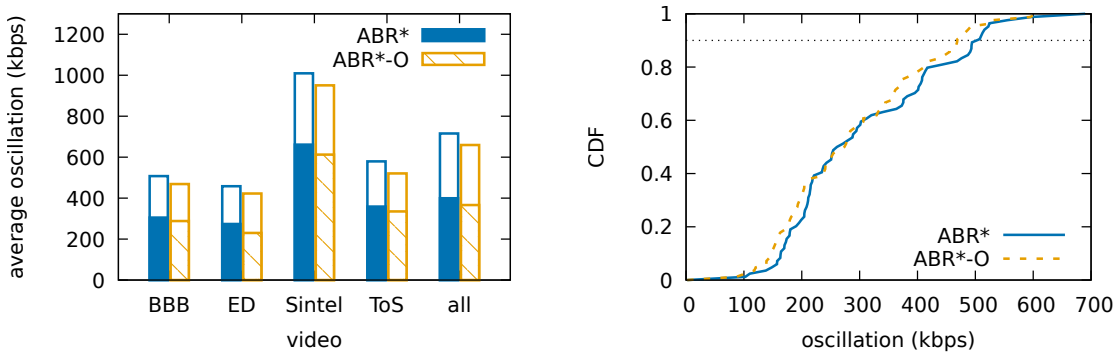
Figure 4.5. The rebuffer ratio and SSIM CDFs over 86 3G traces for BOLA-E, BOLA-SSIM and ABR* with a buffer capacity of 8s for the BBB video. The CDFs correspond to the BBB bars in Figure 4.3.

of 45.4%, it goes up to 76.0% for BOLA-E. This happens because during extreme network conditions, the number of cases when ABR* can abandon a download and keep the partially complete segment increases. Note that while a session with that much rebuffering is unwatchable, the worst network traces can still represent shorter dips in otherwise acceptable network conditions.

4.3.3 ABR*-O: Decreasing bitrate oscillations

BOLA-E avoids bitrate oscillations by extending BOLA-O. Recall that BOLA-O avoids bitrate oscillations by not choosing a higher bitrate than the estimate sustainable bandwidth. However, ABR* has many download options and a download action consists of an SSIM score from the table of available scores for each segment rather than a bitrate. Thus, ABR* switches between bitrates more frequently. Note that for the purpose of measuring bitrate switches, we do not distinguish between a particular segment being downloaded completely or partially.

To reduce oscillations, we penalize any bitrate switch. When ABR* is choosing the next bitrate, we give a penalty to any bitrate different from the last downloaded bitrate. If there are M bitrate levels, v_1 is the SSIM score for the lowest bitrate, and v_M is the SSIM score for the highest bitrate, then we set the bitrate change penalty



(a) average oscillations

(b) oscillation CDF

Figure 4.6. The bitrate oscillations over 86 3G traces for ABR* and ABR*-O with a buffer capacity of 8s. (a) The average bitrate oscillations for four videos individually and together (all). The plot displays the mean (pattern) and 90$^{\text{th}}$ percentile (plain). (b) The bitrate oscillation CDFs for the BBB video.

as $(\upsilon_M - \upsilon_1)/(M-1)$. This corresponds to the average difference in utility between adjacent bitrate levels. The penalty makes the ABR* algorithm more reluctant to change bitrates, but it would still change bitrates if the difference in utility is higher than the penalty. We call this updated algorithm ABR*-O.

4.3.3.1 Evaluation

We repeated the experiments in Section 4.3.1 for ABR*-O using an 8s buffer capacity to compare ABR* with ABR*-O. Figure 4.6 shows the average bitrate oscillations for all videos and the bitrate oscillations CDF for the BBB video. ABR*-O achieves 5.4% less average oscillations, and 7.6% less oscillations at the 90$^{\text{th}}$ percentile. We also compared the average rebuffer ratio and average SSIM in Figure 4.7 and found no significant difference. Thus, ABR*-O reduces bitrate oscillations without any degradation in rebuffering or SSIM.

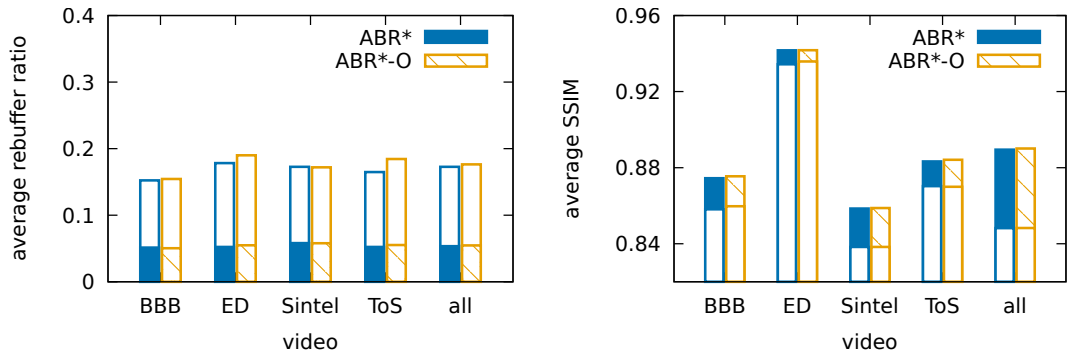


Figure 4.7. The average rebuffer ratio and average SSIM over 86 3G traces for ABR* and ABR*-O with a buffer capacity of 8s for four videos individually and together (all). The rebuffer ratio plot displays the mean (pattern) and 90$^{\text{th}}$ percentile (plain). The SSIM plot displays the mean (pattern) and 10$^{\text{th}}$ percentile (plain).

4.4 Related work

Early video streaming systems used the Internet as a best-effort system without reliable transport [57]. RTSP and RTP were commonly used for signalling and delivering media respectively. While RTSP signalling can be delivered on either TCP or UDP, it is generally delivered over TCP. RTP can also be delivered on either TCP or UDP. However, RTP is mostly delivered over UDP [22]. A selective retransmission extention to RTP called SR-RTP [17] explores a middle ground between using a reliable or an unreliable protocol. SR-RTP retransmits packets dropped in the key I-Frames but does not retransmit packets droppen in other frames.

Early applications for video streaming included several commercial options [57]. RealNetworks' RealVideo solution used Robust UDP to reduce packet loss, tuned the video codecs to minimize damage caused by packet loss, and also used forward error correction [41]. RealNetworks also used SureStream, creating different streams for different connection rates, prioritizing key frames, and dropping some frames in a process called *stream thinning* [42]. Vosaic (video mosaic) was another system that streamed video over an unreliable stream [9]. The video server transmits video at the recorded frame rate and the video client gives feedback regarding frame drop rate

and packet drop rate. The server then adjusts the frame rate based on the client feedback.

Modern video provides use HTTP adaptive streaming (HAS) as the default video delivery method [46]. A benefit of HTTP and the underlying TCP protocol is that HAS systems have reliable transport. However, TCP's reliability comes at a price. Lost packet retransmissions introduce delays and can cause head-of-line blocking.

HTTP video delivery can be delivered via both TCP and QUIC. Bhat et al. [7] show that QUIC does not provide QoE performance improvements when compared with TCP. One factor might have been that the work used an experimental QUIC implementation provided by the Caddy web server. Langley et al. [28] show a constrasting picture, with significant QoE improvement with QUIC when compared with TCP. However, neither of the two studies investigate unreliable transport.

Another approach to lossy video delivery is to use reliable a reliable protocol but drop frames to save bandwidth. Yahia et al. [60] propose a scheme where each frame in a segment is delivered as in independent HTTP/2 object, and the video player can drop the less important frames in a segment. However, the approach does not support multiple bitrates so it cannot work effectively with high network variations.

4.5 Conclusion

Prior work involves either tuning the transport protocol or optimizing the application layer, i.e., the ABR algorithm. We propose VOXEL, a novel video streaming system with cross-layer optimization. Empirical results show that VOXEL outperforms state-of-the-art video streaming by exploiting cross-layer synergies. VOXEL's design uses the insights that videos can tolerate some loss and not all frames have the same impact on QoE when dropped. We design ABR*, an ABR algorithm within the VOXEL system that uses selective transport reliability to avoid rebuffering without a significant impact on QoE.

CHAPTER 5

SIMULATION OF ADAPTIVE 360° VIDEO

Recently 360° video content is increasing in popularity [63]. ABR algorithms designed for regular videos do not necessarily work out of the box for 360° videos. The viewer only sees a fraction of the whole video, i.e., the *viewport,* at any one time. Thus, delivering the whole 360° video at a high bitrate can be inefficient. This inefficiency is generally mitigated by partitioning the viewing area into *tiles* and delivering only a subset of the tiles that are present in the viewer's viewport. However, this approach can cause problems if not implemented correctly. A viewer expects a seamless viewing experience when changing their pose. A pose change, however, might change the subset of tiles visible in the viewport. If the video content for the tiles in the new viewport is not present in the player buffer, the content will not be available to the viewer and the viewer may see a blank screen. The challenge is exacerbated by the fact that a delay from a time the viewer moves their head to the time the display reacts can cause motion sickness.

In order to deliver 360° video efficiently with a high QoE, a 360° video player uses a *view prediction* scheme to predict which tiles are more likely to fall within the viewport. The player then allocates bandwidth resources based on the prediction, using a 360° *ABR algorithm* to download different tiles at the appropriate bitrates. Note that while traditional 2-D ABR algorithms only need to address unpredictability in the network conditions, 360° ABR algorithms need to address unpredictability both in the network conditions and in the viewport.

Testing is an important step in the development of 360° video adaptation algorithms. Testing 360° algorithms on live systems is even more difficult than testing 2-D videos. The viewers need to use a head mounted display (HMD), and algorithm performance is highly dependent on the viewer behavior during each session. This makes a fair comparison difficult.

We propose and develop Sabre360, a simulation testbed to test 360° algorithms using pre-recorded head movement traces along with pre-recorded bandwidth traces. Sabre360 provides a fair comparison between 360° algorithms by using the same traces for each test session. We also develop two novel 360° ABR algorithms, a baseline throughput-based algorithm and a BOLA-based algorithm.

5.1 Sabre360: simulation testbed for 360° videos

We introduced Sabre, a Python based simulation environment for 2-D ABR algorithms, in Chapter 3. However, Sabre cannot be used to simulate 360° video streams that have multiple tiles and different viewports. We now describe Sabre360, an extension of the simulation environment for 360° video streaming. We made Sabre360 open source and publicly available to the community on GitHub [49].

Sabre360 facilitates the design of new 360° ABR algorithms. Figure 5.1 shows the system architechure. A view prediction algorithm module and an adaptive bitrate algorithm module can be tested with a set of videos. Each video needs one or more head movement traces for simulation. Each video and head movement trace combination can be simulated using a set of bandwidth traces. One simulation session involves simulating one video with one head movement trace and one bandwidth trace. A simulation session yields a detailed event log file which can be used to calculate QoE metrics.

The simulation process is reproducible and repeating a simulation session with the same video, head movement trace and bandwidth trace yields identical results.

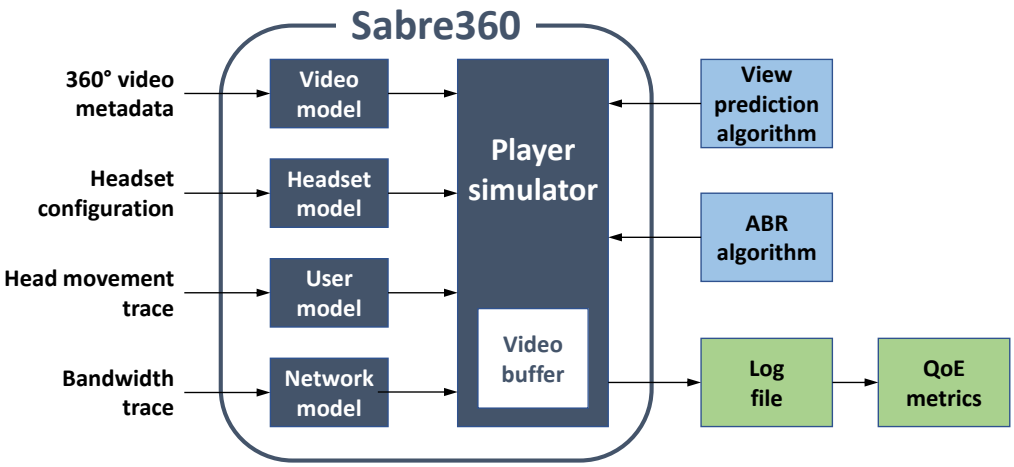


Figure 5.1. The Sabre360 architecure. Different systems are modelled independently and the core player orchestrates a video session. The view prediction algorithm and adaptive bitrate algorithm can be provided by the user.

The repeatable nature of Sabre360 allows a fair comparison when simulating multiple ABR algorithms. A video session simulation involves multiple components working in tandem orchestrated by the central component, the player simulator. We will call the player simulator component simply as the player in the rest of this chapter.

5.1.1 Video Model

The video model encapsulates the information relating to the 360° video files. While Sabre360 does not need the encoded 360° video media files to simulate a streaming session, it needs the metadata describing the video and media file sizes. This corresponds to the information presented in the DASH manifest. Note that tiled 360° videos manifests use the spatial relationship description (SRD) [36] extension to DASH.

The video model reads video information from a JSON file containing the following:

Segment duration: the segment duration in ms.

Tile configuration: the number of horizontal and vertical tiles in the video.

Bitrates: the average video bitrates in kbit/s for each bitrate level. The average bitrate for a quality level is the total size in bits of all the tiles for all the segments at the quality level divided by the video duration.

Segment sizes: the size in bits at each bitrate level for each tile in every segment.

5.1.2 Network Model

The network model provides a mechanism for simulating network conditions based on an input network trace obtained from a JSON file. The trace consists of a number of periods, with each period having some fixed bandwidth and round-trip latency. Once the whole trace is consumed, the network model repeats it from the top.

The network model simulates sequential segment downloads over a network analogous to a HTTP session. The player requests a download for a size corresponding to the required tile, and the network model returns the time taken for the download. The player can also indicate a delay, in which case the network model seeks ahead in the bandwidth trace.

There are cases ABR algorithms might want to abandon a segment download, for example if network conditions deteriorate suddenly right after requesting a high-bitrate segment. Abandoning the segment to request a low-bitrate segment is a useful mechanism to avoid rebuffering. To enable segment abandonment, the network model sends regular progress updates to the player, and the player can abandon the segment at that time. A progress update is sent when at least 50 ms have passed since the last progress update (or since the initial request) and at least 1500 bytes have been downloaded since the last progress update (or since the initial request). This behavior is similar to practical players. For example, dash.js [13] uses XMLHttpRequest onprogress events to trigger checking whether to abandon a segment.

Traditional video players such as dash.js process segments sequentially, only requesting a segment after the previous segment is fully downloaded. This leads to underutilization of network bandwidth, specifically a round trip time for each request. When streaming tiled 360° videos, this underutilization is much more significant. For example, if the segment duration is 1s and the player needs to download ten tiles with a round trip delay of 50 ms, then the player loses 500 ms or 50% of the download time.

Sabre360 allows two solutions. First, requests for multiple tiles can be grouped and sent together. One way to implement this in practice is to use a custom HTTP/2 server that pushes a number of tiles in response to one suitably crafted GET request [40]. Second, the network model allows request pipelining, where a request can be sent while the previous requests have not yet been fully downloaded. This behavior is supported by regular HTTP/2 servers.

5.1.3 Headset Model

Traditional 2-D videos are rendered on a variety of devices that differ on characteristics such as screen size, resolution and aspect ratio. Head mounted displays (HMDs) used for rendering 360° videos are even more diverse [63]. 360° video tiling further complicates video streaming simulation. Simulating video delivery over the network is not enough, and Sabre360 also needs to simulate the video presentation to the viewer. While Sabre360 does not explore video decoding and rendering, it does keep track of which tiles are visible to the viewer and whether the tiles are fully visible or fractionally visible.

The headset model assumes that the headset supports the tiling scheme described for the video model. However, the viewport does not display all the tiles at the same time. Instead, the number of visible tiles is determined using a configurable field of view (FoV).

The Sabre360 headset model supports equirectangular projection (ERP), the most common 360° video projection [63]. ERP projects the 3-D sphere onto a rectangle with significant stretching at the north and south poles. While Sabre360 supports different projections through its modular video and headset models, we only consider ERP.

We use unit quaternions [59] to represent the headset direction. Quaternions represent orientations and rotations in a 3-D space. A unit vector $\langle x, y, z \rangle$ in the direction of a unit quaternion (qx, qy, qz, qw) can be calculated as follows [59]:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \times qx \times qz + 2 \times qy \times qw \\ 2 \times qy \times qz - 2 \times qx \times qw \\ 1 - 2 \times qx^2 - 2 \times qy^2 \end{bmatrix} \tag{5.1}$$

The unit vector can be projected onto a sphere.

We project rectangular video onto a sphere using the same reference frame as [59]. The vector $\langle 0, 1, 0 \rangle$ (respectively, $\langle 0, -1, 0 \rangle$) represents the viewer looking directly north (respectively, south). With no vertical tilt, the vector $\langle 0, 0, 1 \rangle$ represents the viewer looking forward and the vector $\langle 1, 0, 0 \rangle$ represents the viewer looking directly to their right. Figure 5.2 shows how we project a rectangular video on the inside of a sphere. We represent each point on the rectangle by (X, Y) where $X = 0$ (respectively, 1) represents the left (respectively, right) edge, $Y = 0$ (respectively, 1) represents the top (respectively, bottom) edge. The point $(0, 0.5)$ is projected onto the vector $\langle 1, 0, 0 \rangle$ and the point $(0.25, 0.5)$ is projected onto the vector $\langle 0, 0, -1 \rangle$.

In order to determine the tiles visible in the viewport from a quaternion, we first use (5.1) to determine the unit vector. We then calculate the point (X, Y) from the vector $\langle x, y, z \rangle$ as follows:

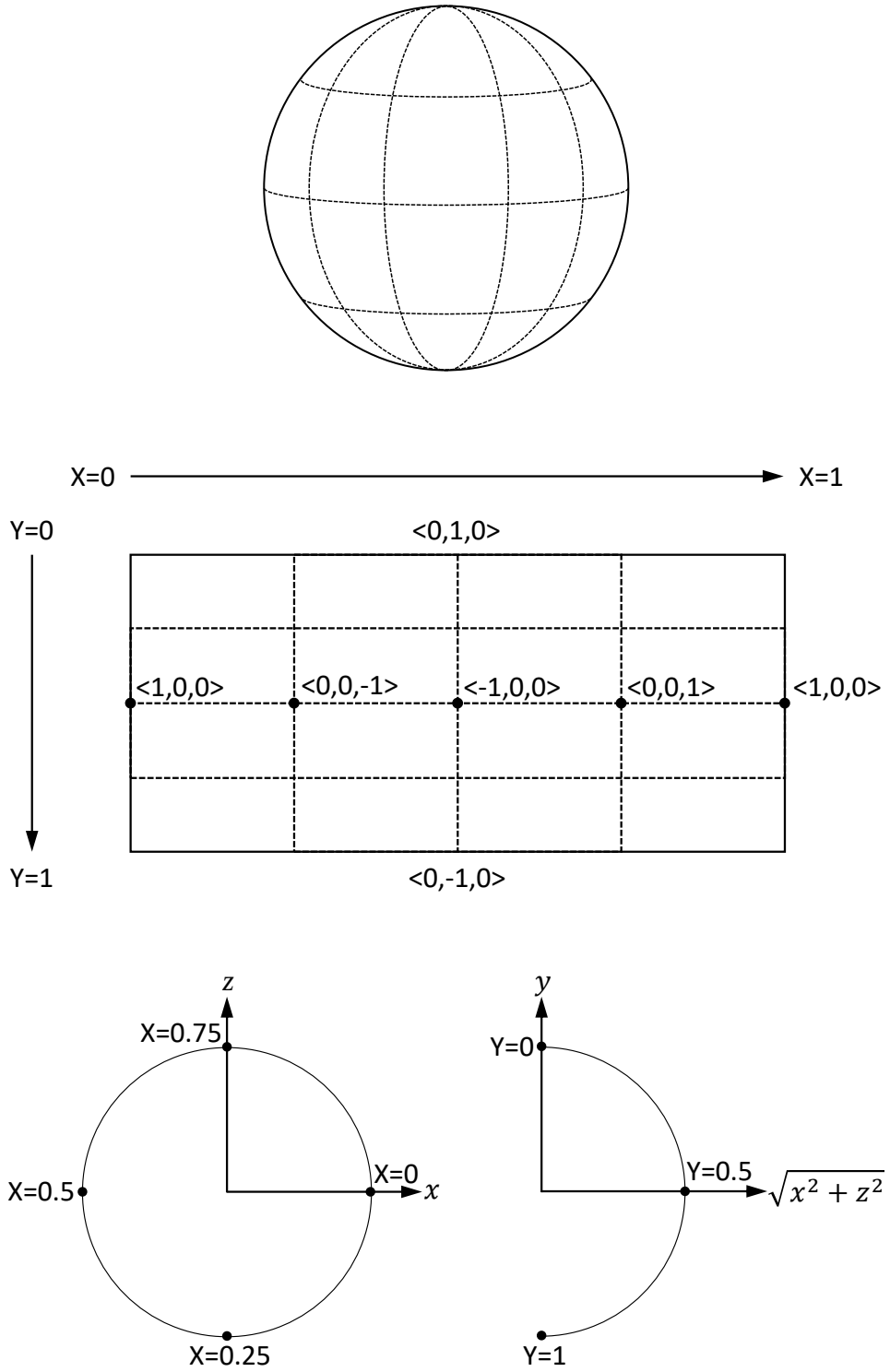


Figure 5.2. Equirectangular projection coordinates. The inside of a sphere is projected onto a rectangle with coordinates (X, Y). The unit vector $\langle x, y, z \rangle$ is mapped to the coordinates (X, Y) as shown.

$$X = \frac{\pi - \text{atan2}(-z, -x)}{2\pi} \tag{5.2}$$

$$Y = \frac{\text{acos}(y)}{\pi} \tag{5.3}$$

We finally calculate the left, right, top and bottom edges using the FoV angles:

$$X_l = X - FoV_X/(4\pi) \tag{5.4}$$

$$X_r = X + FoV_X/(4\pi) \tag{5.5}$$

$$Y_t = Y - FoV_Y/(2\pi) \tag{5.6}$$

$$Y_b = Y + FoV_Y/(2\pi) \tag{5.7}$$

In the horizontal (X) direction, the headset model is able to wrap edges that go past the video boundaries in (5.4)–(5.5) because the left and right edges are circularly connected. In the vertical (Y) direction, the headset model clips edges that go past the video boundaries in (5.6)–(5.7) because the north and south are not connected. The headset model finally determines which tiles fall within the calculated edges, and what fraction of the tiles is visible.

Note that the rectangle (X_l, Y_t)–(X_r, Y_b) is not an accurate boundary of the visible area because of the distortion introduced by ERP projection. However, the Sabre360 architecture allows for accurate boundary calculation through an updated headset model.

Another task for the headset model is to determine how to handle video buffer outages. If a tile is missing when a segment starts rendering, the headset model keeps playing without interruption. This corresponds to a viewer seeing the tile frozen while the rest of the tiles in the viewport continue playback.

5.1.4 User Model

While a traditional 2-D video can be rendered entirely without any user input, 360° video rendering depends on the viewer head pose. The Sabre360 user model reads a

headset trace from a JSON file and updates the player with each pose change. While the user model included with Sabre360 encodes headset poses using quaternions, it is trivial to update the user model to use other representations. Note that any headset trace is only useful when used with same video being displayed while it was captured.

5.1.5 Player Simulator

The player simulator brings all the Sabre360 components together. A Sabre360 session needs a configuration with the following entries:

Buffer size: the buffer capacity in seconds. The player will not download any tile that goes farther into the future than the buffer capacity from the play head.

Video manifest: a JSON file for the video model.

Bandwidth trace: a JSON file for the network model.

Headset trace: a JSON file for the user model.

View prediction algorithm: a Python module with a view prediction algorithm. See Section 5.2.1.

View prediction algorithm configuration: any configuration required by the selected view prediction algorithm.

ABR algorithm: a Python module with an ABR algorithm. See Section 5.2.2.

ABR algorithm configuration: any configuration required by the selected ABR algorithm.

Log file path: a path where to store the output log file.

During the initialization phase, the player initializes the video model with a manifest JSON, the network model with a bandwidth trace JSON, and the user model

with a headset trace JSON. The player also initializes an empty buffer for each tile. Then the player repeats the following loop to until the whole video is played:

1. Estimate the throughput using the throughput estimator. Note that only the network model is allowed to access the bandwidth trace, all the other components can only use throughput estimates.

2. Query the ABR algorithm for an action. The ABR algorithm may query the view prediction algorithm in order to pick an action. The action will be a segment index, tile index, and bitrate level. The ABR algorithm is allowed to either download a new tile or upgrade a previously-downloaded tile at a higher bitrate.

3. Call the network model with the download size. The network model uses a callback to send progress updates every 50 ms until the whole download is complete.

4. On each progress update and on download completion, check the user model for pose changes. Notify the view prediction algorithm with each pose change. Maintain the play head and buffer level at the correct level throughout the pose changes.

5. On each progress update, after completing Step 4, query the ABR algorithm to determine whether to abandon the download. If the ABR algorithm decides to abandon, then instruct the network model to cancel the download and repeat from Step 1.

6. When a download is completed successfully, after completing Step 4, update the buffer state. Then repeat from Step 1.

The above steps are simplified and omit request pipelining as described in Section 5.1.2. To handle pipelining, the player uses the throughput estimator to estimate

when a download will complete. From the estimate download completion time, we subtract either 50 ms or twice the estimate round trip time, whichever is larger. At this time the player queries the ABR algorithm using the predicted system state, and sends the query to the network model in advance. The player can have multiple requests pipelined in this manner.

5.2 Adaptive algorithms

Sabre360 helps fast development of 360° adaptive algorithms. Figure 5.1 shows that Sabre360 needs two input algorithms, a view prediction algorithm and an ABR algorithm. Sabre360 also includes example implementations for both. Thus, a user can simulate a video session using Sabre360 out of the box. A user can then replace either algorithm or both.

5.2.1 View prediction algorithm

The efficiency of a 360° streaming system depends on accurate view prediction [39, 63]. Sabre360 takes a Python module implementation of a view prediction algorithm as input. The algorithm receives pose change events in the past, and can be queried about the future. A view prediction algorithm has two methods to interact with Sabre360. First, it has a method allowing Sabre360 to notify the view prediction algorithm whenever the user's pose changes. Second, it has a method to query the view prediction for a particular segment in the future. The view prediction algorithm returns a vector of probabilities, indicating the probability of each tile of being in the visible viewport for the requested segment. Note that the prediction for a segment in the future might change as the user model advances through the headset trace.

Navigation graph based view prediction

Sabre360 includes an example view prediction algorithm based on navigation graphs [39]. A navigation graph is a data structure than encodes previous view-

port changes, allowing for efficient estimation of future viewports. Navigation graphs work in two modes, cross-user and single-user.

In cross-user mode, each node (called a view) in the graph is a tuple consisting of a segment index and the set of all tiles visible at any point during the whole duration of the segment. An edge from u to v is weighted by the probability that a viewer in view u goes to view v. Note that the segment index in v is one plus the segment index in u. A cross-user navigation graph is populated by aggregating data from several viewers watching a particular video. During a video streaming session, the navigation graph can be used with a current view to give a list of potential next views, along with probabilities. The process can be repeated to predict views several segments into the future. The probability that a particular tile will be visible can be calculated by summing the probabilities of all views that include the tile. Note that the navigation graph can be sparse and the current view is not guaranteed to be in the navigation graph. In such a case, the view with the largest number of tiles intersecting with the current view is used.

In single-user mode, each node (view) in the graph is the set of all tiles visible at any point during the whole duration of a segment. Each streaming session starts with an empty graph, and the graph is updated after every segment. Note that the view does not include the segment index. Thus, the single-user navigation graph captures head movement patterns independent from content.

In the view prediction algorithm proposed in [39], a cross-user navigation graph is used together with a single-user navigation graph. During the first segments, the single-user graph is still unpopulated so only the cross-user graph is used for prediction. After every segment is played, the algorithm checks the precision of prediction using both graphs individually. The graph that yields the higher precision is then used for the next segment prediction. When a viewer's watching pattern matches a common trend, the cross-user graph is more likely to take over. However, when the

viewer does not follow common trends the single-user graph is more likely to take over.

The Sabre360 implementation of navigation-graph based view prediction can input the cross-user navigation graph from a JSON file passed indicated by the view prediction algorithm configuration.

5.2.2 ABR algorithm

The ABR algorithm controls all video download. Sabre360 takes a Python module implementation of an ABR algorithm as input. Sabre360 queries the ABR algorithm when it is ready for the next download. The ABR algorithm can obtain the following information from the player:

Time: both the wall time and the play head time.

Buffer contents: the bitrate for each tile within the playback buffer.

Pose: the current headset pose obtained from the user model.

View prediction: the ABR algorithm can query the view prediction algorithm through the player.

The ABR algorithm returns an action, indicating the segment, tile and bitrate to download. During the course of the download, the player obtains progress events from the network model and queries the ABR algorithm whether to continue the download or abandon. The player finally informs the ABR algorithm about any successful download, indicating the download size, download time and time to first byte.

The ABR algorithm can perform throughput estimation based on the successful download reports it receives from the player. Sabre360 also includes a throughput estimation primitive using an exponentially weighted moving average. The ABR algorithm can either use the Sabre360 primitive or perform its own throughput estimation.

```
1: function GetAction
2:     tp ← throughput estimate
3:     b ← tp × segment duration
4:     s ← first segment after play head with at least one empty tile
5:     b ← b − (bits previously used to download any tiles in s)
6:     p_s ← ViewportPredictor.GetTileProbabilities(s)
7:     q ← AllocateQuality(s, b, p_s)
8:     t ← arg max_t p_s[t] subject to tile t in s being empty
9:     return (s, t, q[t])
10: end function
```

Figure 5.3. Baseline 360° ABR algorithm.

Note that the ABR algorithm does not have access to the network trace used by the network model because deployed algorithms can only estimate throughput from the download history.

Sabre360 includes two example ABR algorithms, a baseline throughput-based algorithm and a BOLA-base algorithm. The user can either use one of the example algorithms or implement their own.

Baseline ABR algorithm

When streaming traditional 2-D video, the THROUGHPUT algorithm can be used as a baseline algorithm because it is simple but performs reasonably well. We now design a baseline throughput-based algorithm for 360° video.

Figures 5.3–5.4 describes the algorithm. The main idea is to use throughput estimation to estimate the number of bits available to download a particular segment, and then use tile probabilities from the view prediction algorithm to allocate more bits to the more important tiles. The algorithm is designed to download one tile at a time without saving any state information.

To decide the next action, the Baseline algorithm first queries the primitive throughput estimator and multiplies the throughput estimate by the segment du-

1: **function** AllocateQuality($s, b, \mathbf{p}_s$)
2: $\quad \mathbf{z}_m \leftarrow$ segment sizes for s ▷ for $1 \leq m \leq M$
3: $\quad \mathbf{q} \leftarrow \langle 1, 1, \ldots, 1 \rangle$ ▷ start with lowest quality $m = 1$
4: $\quad \mathcal{T} \leftarrow$ set of empty tiles in s
5: $\quad b \leftarrow b - \sum_{t \in \mathcal{T}} \mathbf{z}_1[t]$
6: $\quad t \leftarrow \arg\min_t(\mathbf{z}_{\mathbf{q}[t]+1}/\mathbf{p}_s[t])$ subject to $(\mathbf{q}[t] + 1 \leq M)$, $(\mathbf{z}_{\mathbf{q}[t]+1}[t] - \mathbf{z}_{\mathbf{q}[t]}[t] \leq b)$
7: $\quad$ **if** we found some t in line 6 **then**
8: $\quad\quad b \leftarrow b - (\mathbf{z}_{\mathbf{q}[t]+1}[t] - \mathbf{z}_{\mathbf{q}[t]}[t])$
9: $\quad\quad \mathbf{q}[t] \leftarrow \mathbf{q}[t] + 1$
10: $\quad\quad$ **repeat** line 6
11: $\quad$ **end if**
12: $\quad$ **return q**
13: **end function**

Figure 5.4. Baseline 360° ABR algorithm helper.

ration to obtain a segment bit budget. The algorithm then searches for the first segment s in the buffer with at least one tile missing. The size in bits of any tile from segment s that is already in the buffer is then subtracted from the bit budget. The bit budget is then distributed across all the remaining tiles from segment s as follows. Let b be the bit budget and p_t be the probability that tile t is in the viewport. Then the Baseline algorithm bit allocation for tile t is $bp_t / \sum_{t'} p_{t'}$. Figure 5.4 shows how the algorithm handles quantization.

BOLA-based ABR algorithm

Sabre360 also includes a 360° ABR algorithm based on BOLA, which we call BOLA-TS. BOLA-TS is similar to BOLA-E but with the following differences:

View prediction: BOLA-TS queries the view prediction algorithm at the start of each call.

One buffer per tile: BOLA-TS calculates the buffer level for each tile separately. This allows BOLA-TS to give higher priority to more important tiles. For example, it might download the tile in forward direction for five segments in

buffer while only downloading the tile in the opposite direction for two segments in the buffer.

Objective function weighting: BOLA-TS calculates the objective function ρ for each potential tile download. Then each value is multiplied by the probability that the particular tile will be viewed.

Tile replacement: Consider a tile downloaded at a low bitrate because of its low viewing probability. However, the viewer might change their pose and significantly increase the tile's viewing probability. In such a case BOLA-TS has the option of upgrading the tile in the buffer. Note that the FAST-SWITCHING algorithm does not work directly because we only want to replace tiles based on the most recent view prediction information.

Low bitrate safety baseline: The insufficient buffer rule was designed for 2-D videos. We extended the idea to download a low-bitrate safety baseline for 360° videos to avoid rendering empty tiles. With the new safety algorithm, we first determine the earliest segment with at least one missing tile. We also calculate the number of bits required to download the missing tiles at the lowest bitrate. We then use the throughput estimate to determine how many bits can be safely expected to be downloaded until the segment starts to render. BOLA-TS is not allowed to start a download that might prevent the tiles from being downloaded safely.

While most of the above changes were relatively simple to implement, we need some analysis to fit tile replacement within the BOLA utility framework. We want to determine a fair utility to compare replacement downloads with new downloads.

Consider two quality levels m_1 and m_2 for a particular segment having utilities υ_1 and υ_2 and sizes S_1 and S_2 bits respectively, where $\upsilon_1 < \upsilon_2$ and $S_1 < S_2$. Then there is some buffer level Q' such that $\rho_{a_1=1} = \rho_{a_2=1}$. Thus, at Q', BOLA has no

preference between the two. The key insight is that if we could upgrade a segment from quality m_1 to quality m_2 by downloading $(S_2 - S_1)$ bits, then we expect BOLA to have no preference between the upgrade and either of the two segments at buffer level Q', that is:

$$\rho_{a_1=1} = \rho_{a_2=1} = \rho_{1\to 2} \tag{5.8}$$

We can use this insight to calculate a fair utility value for a replacement operation.

Starting with m_1 and m_2, we have:

$$\frac{V(\upsilon_1 + \gamma p) - Q'}{S_1} = \frac{V(\upsilon_2 + \gamma p) - Q'}{S_2}. \tag{5.9}$$

Solving, we get:

$$Q' = V(\upsilon_1 + \gamma p) - \frac{VS_1(\upsilon_2 - \upsilon_1)}{S_2 - S_1} \tag{5.10}$$

Assume for now that there is an efficient upgrade option. That is, we can upgrade a segment from quality level m_1 to level m_2 by downloading exactly $(S_2 - S_1)$ bits. Let the upgrade have some utility $\upsilon_{1\to 2}$. We can now write an expression for the objective function for the upgrade as

$$\rho_{1\to 2} = \frac{V\upsilon_{1\to 2} - Q}{S_2 - S_1}. \tag{5.11}$$

Note that we do not include a γp term in (5.11) because $\upsilon_{1\to 2}$ is still unknown and can potentially include the term.

We know from (5.8) that $\rho_{a_2=1} = \rho_{1\to 2}$.

$$\frac{V(\upsilon_2 + \gamma p) - Q'}{S_2} = \frac{V\upsilon_{1\to 2} - Q'}{S_2 - S_1}. \tag{5.12}$$

Solving (5.10) and (5.12) we get:

$$\upsilon_{1\to 2} = \upsilon_2 - \frac{S_1(\upsilon_2 - \upsilon_1)}{S_2 - S_1} + \gamma p \tag{5.13}$$

and

$$\rho_{1\to 2} = \frac{V(\upsilon_2 - (\upsilon_2 - \upsilon_1)S_1/(S_2 - S_1) + \gamma p) - Q}{S_2 - S_1} \tag{5.14}$$

Note that (5.13) includes the γp term.

However, an upgrade from quality level m_1 to level m_2 only has a size of $(S_2 - S_1)$ bits when using scalable coding. Otherwise, the upgrade has a size of S_2 bits. If scalable video is not in use, we get

$$\rho'_{1\to 2} = \frac{V(\upsilon_2 - (\upsilon_2 - \upsilon_1)S_1/(S_2 - S_1) + \gamma p) - Q}{S_2} \tag{5.15}$$

BOLA-TS uses (5.13) to fairly calculate the utility for replacement segments.

5.3 Evaluating 360° adaptive algorithms using Sabre360

We now show how Sabre360 can be used to evaluate adaptive algorithms by comparing the Baseline ABR algorithm with BOLA-TS using the navigation-graph based view prediction algorithm. We use the 48 headset traces provided by [59] for the *Google Spotlight-HELP* video. We used the first 42 traces to populate the cross-user navigation graph and the remaining 6 traces for evaluation. The video is encoded at five bitrates between 2 and 23 Mbps, tiled into 4×4 tiles and segmented with 1s segments. In our evaluations, we used the 40 4G traces provided by [56]. Sabre360 gives a log file for each session, and we process the log files to generate metrics information. During our experiments, we focused on three metrics described below.

Average utility: We calculate the BOLA utility as $\ln(S_m/S_1)$ in (2.21). The headset model does not rebuffer when missing a tile, but we still give a penalty of $-\upsilon_M$ for each missing tile. To calculate the average utility, we first calculate the average utility for each visible time at any moment in time. Then we calculate the average of the instantaneous utilities across the whole video session. Note that we the average utility calculation is not conditioned by segment boundaries.

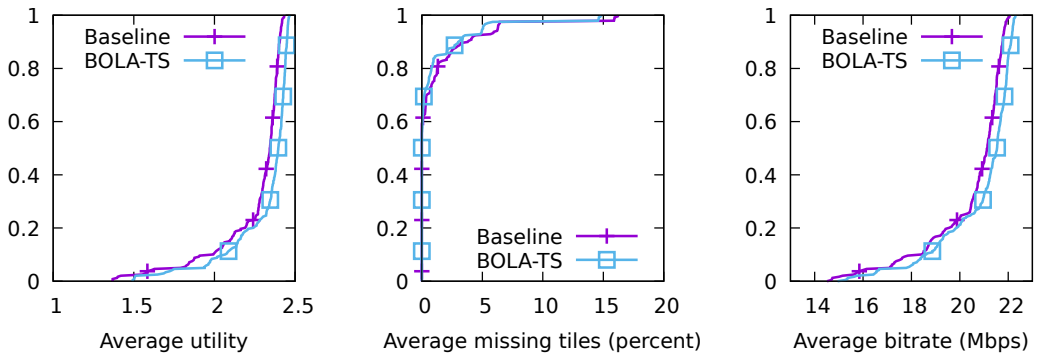


Figure 5.5. CDF for 6 viewers for 40 4G traces with a 5s buffer capacity. BOLA-TS has 2.31 average utility, 0.9% average missing tiles and 20.9 Mbps average bitrate. Baseline has 2.25 average utility, 1.2% average missing tiles and 20.5 Mbps average bitrate.

Table 5.1. QoE metrics for the Baseline algorithm and BOLA-TS for 6 viewers for 40 4G traces.

	Baseline			BOLA-TS		
Buffer Capacity (seconds)	Utility	Missing tiles (percent)	Bitrate (Mbps)	Utility	Missing tiles (percent)	Bitrate (Mbps)
3	2.21	1.93	20.3	2.27	1.17	20.4
5	2.25	1.16	20.5	2.31	0.94	20.9
10	2.28	0.64	20.6	2.30	0.44	20.6

Average missing tiles: When playing video, the user can see multiple tiles. Any tile in the viewport that is missing the video data is considered a missing tile. Similar to utility, we first calculate the average at any moment in time then we average the result across the whole video.

Average bitrate: Again, we first calculate the average across all visible tiles, with missing tiles having a 0 bitrate, and then average the result across the whole video. Note that the average bitrate might be higher than the network bandwidth. For example, if a video player can successfully stream a 20 Mbps video by only downloading half the tiles, then the average bitrate would still be 20 Mbps even though it only uses 10 Mbps network capacity.

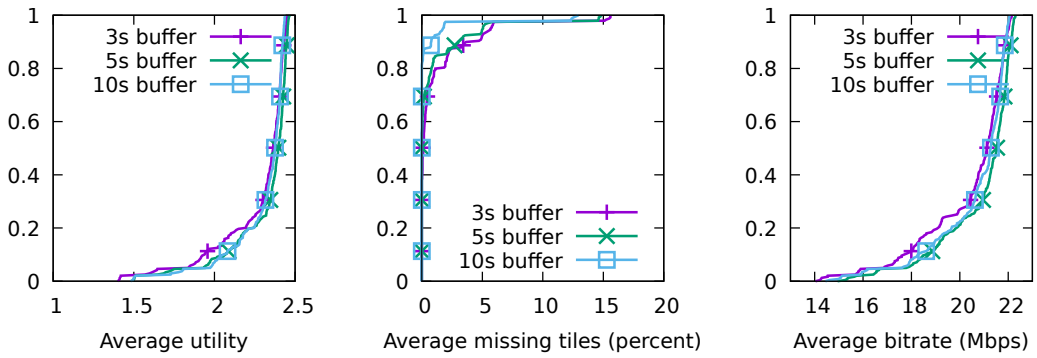


Figure 5.6. CDFs for 6 viewers for 40 4G traces using BOLA-TS.

We first used Sabre360 to simulate the Baseline algorithm and BOLA-TS with a buffer capacity of 5s for 6 viewers and 40 4G traces, testing each viewer headset trace with each of the network traces. Figure 5.5 shows the average utility, missing tiles and bitrate. The mean values are shown in Table 5.1. BOLA-TS has a 3% higher utility than the Baseline, with 19% less missing tiles and 2% higher bitrate.

We repeated the experiments with buffer capacities of 3s and 10s. Figure 5.6 compares BOLA-TS for different buffer capacities. The mean values are shown in Table 5.1. Note that while increasing the buffer from 3s to 5s gives better QoE across all metrics, increasing the buffer from 5s to 10s causes BOLA-TS to lose some bitrate and utility. With traditional 2-D video, we expect to obtain better QoE for a larger buffer. It is reasonable to expect a similar trend for 360° video. However, view prediction loses accuracy in the future. This might cause the ABR algorithm to waste bandwidth resources on less useful downloads. With a smaller buffer, BOLA-TS has more accurate view prediction and uses resources more effectively. On the other hand, the larger buffer still reduces the missing tiles.

Sabre360 also records download size information. Unused downloads are practically inevitable when streaming tiled 360° videos. Since view prediction cannot be 100% accurate, we expect to download some tiles which are never rendered. Figure 5.7 shows a CDF for the total number of bits downloaded during the previous

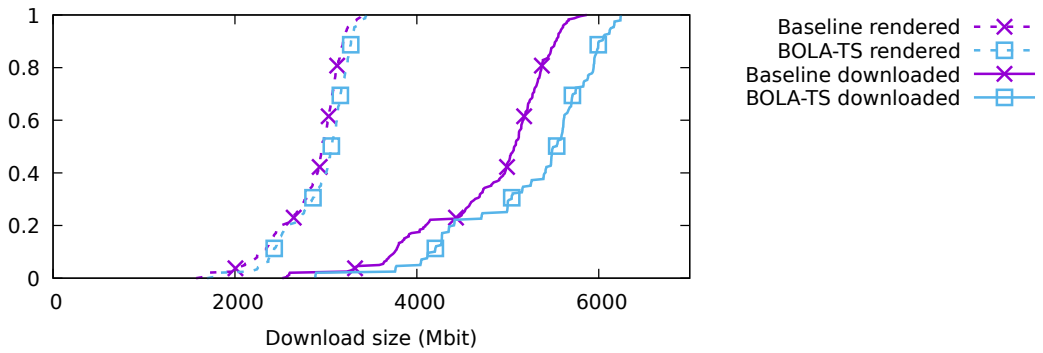


Figure 5.7. The download size CDF for 6 viewers for 40 4G traces with a buffer capacity of 5s. The download size is the total number of bits downloaded in one session. The rendered size is the subset of the total download including only the downloaded tiles that were in the viewport at some point. The Baseline algorithm used 59% of the downloaded bits while BOLA-TS used 56% of the downloaded bits.

experiments with a buffer capacity of 5s. We also measured the total number of bits used to download the useful tiles, i.e., the tiles that were in the viewport at some point. The Baseline algorithm used 59% of the downloaded bits while BOLA-TS used 56% of the downloaded bits.

Sabre360 can also be used with synthetic traces to test specific scenarios. We tested BOLA-TS with and without tile replacement. Since we did not see any significant difference when using the 4G traces, we created 100 synthetic traces. In each synthetic trace, we have 5s of low bandwidth (below 2 Mbps) followed by 10s of high bandwidth (above 20 Mbps), followed again by 5s of low bandwidth and so on. Figure 5.8 shows that replacement improves bitrate and utility. Disabling tile replacement loses 4.9% utility, 1.4% bitrate and has 0.6% more missing tiles. While we do not expect replacement to improve the missing-tile metric, the difference is not significant. Measuring the download sizes, BOLA-TS with replacement uses 60.9% of the total downloaded bits while BOLA-TS without replacement uses 61.3%. The Baseline algorithm has 33.8% less utility, 0.9% more bitrate and 59% more missing tiles when compared with BOLA-TS. Note that the missing tiles have a higher effect

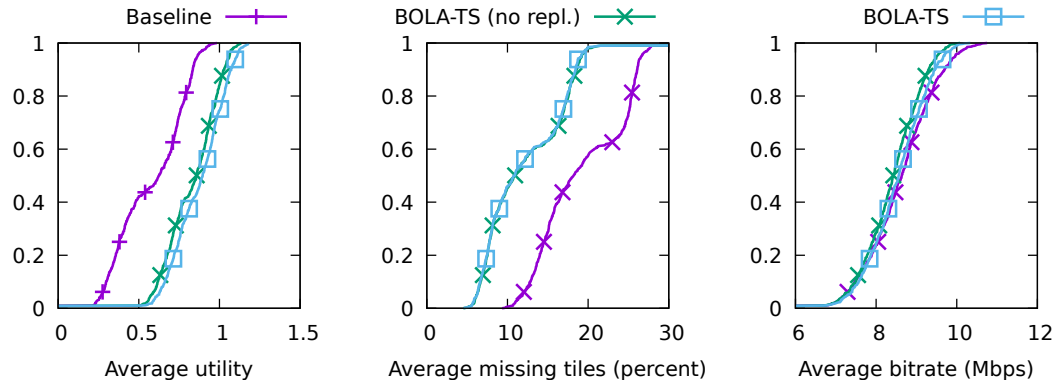


Figure 5.8. CDFs for 6 viewers for 100 synthetic traces with high bandwidth variations using buffer capacity of 3s. We use BOLA-TS with replacement enabled (default) and disabled.

on QoE than the higher bitrate. We repeated the experiments with buffer capacities of 5s and 10s. There is no significant difference between BOLA-TS with replacement and BOLA-TS without replacement for the larger buffer capacities.

5.4 Conclusion

We designed and implemented Sabre360, a simulation testbed for 360° video adaptation algorithms. We demonstrated how Sabre360 can be a useful tool to design and compare new algorithms. We also developed two 360° ABR algorithms and compared their performance using Sabre360.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this dissertation we propose algorithms for adaptive video delivery that improve the viewer QoE. The algorithms are easy to implement and we implemented them in dash.js, the DASH reference player. Video providers are using our algorithms in a production setting. We also develop two open source testbeds for simulating adaptive video algorithms, one for traditional 2-D videos and one for 360° videos. We summarize our contributions below.

Adaptive bitrate selection: We developed BOLA, a bitrate selection algorithm with a near-optimal theoretical guarantee. We prove that BOLA gives a time-average utility within an additive constant of the optimal. Further, the additive constant can be made smaller arbitrarily by increasing the buffer capacity. We extended BOLA to handle practical considerations such as limited buffer capacity, user actions such as startup and seeking, and abrubt network condition changes. We develop a variant of BOLA to perform better when lossy video delivery is allowed and another variant of BOLA to work with tiled 360° videos.

Simulation testbed: We develop Sabre, an open source simulation framework for designing ABR algorithms. We show that ABR algorithms implemented in Sabre give similar QoE metrics to their counterpart algorithms implemented in dash.js when simulated using the metadata and network traces for the dash.js video sessions. We extended the simulation testbed for 360° videos in Sabre360. Sabre360 can help the development of view prediction algorithms and 360° ABR algorithms. Both projects are available on GitHub under a free BSD license.

Algorithms in production: We took our ABR algorithms from theory to practice by implementing them in dash.js, the DASH reference player. Our algorithms are the default ABR algorithms in dash.js and are being used in production by several CDNs and video providers including Akamai, BBC, CBS and Orange.

6.1 Future work

The algorithms presented in this dissertation advance the state-of-the-art in ABR algorithms, but there are several directions in which the perfomance can be improved. We list a few possible directions for future work.

6.1.1 Ultra low-latency live streaming

We show that our algorithms perform well with buffer capacities as low as 10s in Chapter 3. In Chapter 4, we extend our algorithms to work well even when we can only have one full segment in the buffer before starting to download the next segment. However, if the player has one segment in the buffer and starts to download the next segment that is available at the server, the glass-to-glass latency is at least twice the duration of a segment. This limitation can be overcome using chunked encoding, where a segment is partitioned into chunks [29]. The video server can start delivering chunks from a segment before the full segment has even been captured by the camera. The video client can start downloading each chunk as soon as it is available. State-of-the-art ABR algorithms for chunked video are currently limited to overcoming throughput estimation challenges for chunked video delivery [4, 5]. As future work, principled ABR algorithms can be designed to address the challenges of chunked delivery. Such algorithms also need to determine how aggresively the video player can track the live edge. For example, if the network conditions are not very reliable, then the player might prefer to add a few seconds of delay to build a buffer that can absorb some network fluctuations.

6.1.2 QoE analysis

ABR algorithms generally use simple metrics such as bitrate as a proxy for QoE metrics. In Chapter 4 we show that this is not always sufficient, and current QoE metrics such as PSNR and SSIM are not as accurate as VMAF regarding correlation to MOS, but VMAF does not support artifacts related to lossy video transmission. Future work can include developing QoE metrics that have high correlation to MOS and support artifacts such as lossy transmission.

QoE analysis for 360° videos is another possible future work direction. In addition to saving bandwidth by not downloading tiles outside of the viewport, 360° ABR algorithms can also save bandwidth by downloading tiles in the peripheral vision area at a lower bitrate. To do this while still achieving high QoE, we need user studies showing how it affects viewers' experience. Another area that warrants attention is bitrate oscillation. In the 2-D video case, good QoE can be achieved by reducing bitrate oscillations between successive segments. In the 360° video case, more work is needed to study how users perceive bitrate oscillations in the peripheral vision. Also, the effect of having different bitrates for different tiles within the viewport needs further study.

6.1.3 ABR algorithms and caching

State-of-the-art ABR algorithms assume that the video client downloads segments from a server that has all the video content ready for download. Content delivery in practice is more nuanced. A video client generally downloads video from a CDN edge server, and segment download performance is impacted by whether the segment requested is available in the edge server cache. As future work, an ABR algorithm can distinguish between delays caused by last-mile network conditions and delays caused by cache misses when estimating the network conditions.

On the server side, prefetching segments at the edge before the client requests the segments improves delivery performance. A simple prefetching technique is to prefetch the next segment at the same bitrate as the last segment download. As future work, more advanced prefetching techniques can be developed. Such techniques may also include signalling between the client and server to improve both cache performance and ABR algorithm design.

6.1.4 Machine learning as an ABR algorithm development tool

Machine learning (ML) has been used for developing ABR algorithms such as Pensieve [32]. In Chapter 2 we show that while such algorithms can perform well, the training does not transfer well if the network conditions are different from the training network conditions. Further, it is difficult to explain why the algorithms work. The difficulty in explaining the ML algorithms presents an obstacle to deploying the algorithms in production because it is difficult to fix problems that cannot be understood. As future work, ML can be used as a tool to design ABR algorithms, but the resulting algorithms should be simple to understand. There are several directions for this. One direction can be to use an algorithm such as Pensieve but restricting the input signals such as throughput history. By comparing the performance of algorithm instances with different input signals, we could get an insight into which input signals are more important and less important for designing a simple ABR algorithm.

BIBLIOGRAPHY

[1] Adobe. HTTP Dynamic Streaming. http://www.adobe.com/products/hds-dynamic-streaming.html. Accessed: September, 25, 2014.

[2] Akhtar, Zahaib, Nam, Yun Seong, Govindan, Ramesh, Rao, Sanjay, Chen, Jessica, Katz-Bassett, Ethan, Ribeiro, Bruno, Zhan, Jibin, and Zhang, Hui. Oboe: Auto-tuning video abr algorithms to network conditions. In *Proc. ACM SIGCOMM* (2018).

[3] Apple. HTTP Live Streaming. https://developer.apple.com/streaming/. Accessed: September, 25, 2019.

[4] Bentaleb, Abdelhak, Timmerer, Christian, Begen, Ali C, and Zimmermann, Roger. Bandwidth prediction in low-latency chunked streaming. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video* (2019), pp. 7–13.

[5] Bentaleb, Abdelhak, Timmerer, Christian, Begen, Ali C, and Zimmermann, Roger. Performance analysis of acte: A bandwidth prediction method for low-latency chunked streaming. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM) 16*, 2s (2020), 1–24.

[6] Bertsekas, Dimitri. *Dynamic programming and optimal control*, vol. 1. Athena Scientific Belmont, MA, 1995.

[7] Bhat, Divyashri, Rizk, Amr, and Zink, Michael. Not so QUIC: A performance study of DASH over QUIC. In *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video* (2017), ACM, pp. 13–18.

[8] Blender Foundation. Big Buck Bunny Movie. https://peach.blender.org/. Accessed: July 31, 2015.

[9] Chen, Zhigang, Tan, See-Mong, Campbell, Roy H, and Li, Yongcheng. Real time video and audio in the world wide web. In *Fourth International World Wide Web Conference* (1995), Citeseer, pp. 333–348.

[10] Cisco. Visual Networking Index. `http://bit.ly/KXDUaX`.

[11] Commision, Federal Communications. Raw data — measuring broadband america 2016. `https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-2016`, 2016.

[12] Dailymotion. Introducing hls.js. `http://engineering.dailymotion.com/introducing-hls-js/`, 2015.

[13] DASH Industry Forum. dash.js javascript reference client. `https://reference.dashif.org/dash.js/`.

[14] DASH Industry Forum. Guidelines for Implementation: DASH-AVC/264 Test cases and Vectors. http://dashif.org/guidelines/, January 2014.

[15] De Cicco, Luca, Caldaralo, Vito, Palmisano, Vittorio, and Mascolo, Saverio. Elastic: a client-side controller for dynamic adaptive streaming over HTTP (DASH). In *Packet Video Workshop (PV), 2013 20th International* (2013), IEEE, pp. 1–8.

[16] Dobrian, Florin, Sekar, Vyas, Awan, Asad, Stoica, Ion, Joseph, Dilip, Ganjam, Aditya, Zhan, Jibin, and Zhang, Hui. Understanding the impact of video quality on user engagement. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 362–373.

[17] Feamster, Nick, and Balakrishnan, Hari. Packet Loss Recovery for Streaming Video. In *12th International Packet Video Workshop* (April 2002).

[18] Forum, DASH Industry. September 13, 2016 dash.js meeting minutes. `https://github.com/Dash-Industry-Forum/dash.js/wiki/Meeting-Minutes`, 2016.

[19] Forum, DASH Industry. dash.js meeting minutes. `https://github.com/Dash-Industry-Forum/dash.js/wiki/Meeting-Minutes`, accessed 2019.

[20] Foundation, Blender. Big buck bunny movie. `https://peach.blender.org/`, 2008.

[21] Google. Shaka player. `https://opensource.google.com/projects/shaka-player`, 2015.

[22] Handley, Mark, and Perkins, Colin. Rfc2736: Guidelines for writers of rtp payload format specifications, 1999.

[23] Huang, Te-Yuan, Johari, Ramesh, McKeown, Nick, Trunnell, Matthew, and Watson, Mark. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proc. ACM SIGCOMM* (2014).

[24] Huang, Te-Yuan, Johari, Ramesh, McKeown, Nick, Trunnell, Matthew, and Watson, Mark. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review 44*, 4 (2015), 187–198.

[25] Huynh-Thu, Quan, and Ghanbari, Mohammed. Scope of validity of PSNR in image/video quality assessment. *Electronics letters 44*, 13 (2008), 800–801.

[26] Jiang, Junchen, Sekar, Vyas, and Zhang, Hui. Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with festive. *IEEE/ACM Transactions on Networking (ToN) 22*, 1 (2014), 326–340.

[27] Krishnan, S Shunmuga, and Sitaraman, Ramesh K. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. In *Proc. ACM IMC* (2012).

[28] Langley, Adam, Riddoch, Alistair, Wilk, Alyssa, Vicente, Antonio, Krasic, Charles, Zhang, Dan, Yang, Fan, Kouranov, Fedor, Swett, Ian, Iyengar, Janardhan, et al. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 183–196.

[29] Law, Will. Ultra-low-latency streaming using chunked-encoded and chunked-transferred cmaf. Tech. rep., Technical report, Akamai, 2018.

[30] Li, Zhi, Zhu, Xiaoqing, Gahm, Joshua, Pan, Rong, Hu, Hao, Begen, Ali, and Oran, David. Probe and adapt: Rate adaptation for HTTP video streaming at scale. *IEEE JSAC 32*, 4 (2014), 719–733.

[31] Maggs, Bruce M, and Sitaraman, Ramesh K. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review 45*, 3 (2015), 52–66.

[32] Mao, Hongzi, Netravali, Ravi, and Alizadeh, Mohammad. Neural adaptive video streaming with pensieve. In *Proceedings of the 2017 ACM SIGCOMM Conference* (2017), ACM.

[33] Neely, Michael J. Stochastic network optimization with application to communication and queueing systems. *Synthesis Lectures on Communication Networks 3*, 1 (2010), 1–211.

[34] Neely, Michael J. Dynamic optimization and learning for renewal systems. *IEEE Transactions on Automatic Control 58*, 1 (2012), 32–46.

[35] Netflix. Toward A Practical Perceptual Video Quality Metric. https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652, June 2016.

[36] Niamut, Omar A, Thomas, Emmanuel, D'Acunto, Lucia, Concolato, Cyril, Denoual, Franck, and Lim, Seong Yong. MPEG DASH SRD: spatial relationship description. In *Proceedings of the 7th International Conference on Multimedia Systems* (2016), ACM.

[37] Palmer, Mirko, Appel, Malte, Spiteri, Kevin, Chandrasekaran, Balakrishnan, Feldmann, Anja, and Sitaraman, Ramesh. Video streaming optimization across enriched layers, 2020. Unpublished.

[38] Palmer, Mirko, Krüger, Thorben, Chandrasekaran, Balakrishnan, and Feldmann, Anja. The quic fix for optimal video streaming. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (New York, NY, USA, 2018), EPIQ'18, ACM, pp. 43–49.

[39] Park, Jounsup, and Nahrstedt, Klara. Navigation graph for tiled media streaming. In *Proceedings of the 27th ACM International Conference on Multimedia* (New York, NY, USA, 2019), MM '19, ACM, pp. 447–455.

[40] Petrangeli, Stefano, Swaminathan, Viswanathan, Hosseini, Mohammad, and De Turck, Filip. An HTTP/2-based adaptive streaming framework for 360 virtual reality videos. In *Proceedings of the 25th ACM international conference on Multimedia* (2017), pp. 306–314.

[41] RealNetworks. RealVideo technical white paper. http://www.realaudio.com/products/realvideo/overview/index.html. Internet Archive https://web.archive.org/web/19970711223650/http://www.realaudio.com/products/realvideo/overview/index.html.

[42] RealNetworks. SureStream™ — delivering superior quality and reliability. http://www.real.com/devzone/library/whitepapers/surestrm.html. Internet Archive https://web.archive.org/web/19990427160012/http://www.real.com/devzone/library/whitepapers/surestrm.html.

[43] Riiser, Haakon, Vigmostad, Paul, Griwodz, Carsten, and Halvorsen, Pål. Commute path bandwidth traces from 3G networks: analysis and applications. In *Proceedings of the 4th ACM Multimedia Systems Conference* (2013), ACM, pp. 114–118.

[44] Romaniak, Piotr, and Janowski, Lucjan. How to build an objective model for packet loss effect on high definition content based on ssim and subjective experiments. In *Future Multimedia Networking* (Berlin, Heidelberg, 2010), Sherali Zeadally, Eduardo Cerqueira, Marília Curado, and Mikołaj Leszczuk, Eds., Springer Berlin Heidelberg, pp. 46–56.

[45] Romirer-Maierhofer, Peter, Ricciato, Fabio, D'Alconzo, Alessandro, Franzan, Robert, and Karner, Wolfgang. Network-wide measurements of TCP RTT in 3G. In *Traffic Monitoring and Analysis*. Springer Berlin Heidelberg, 2009, pp. 17–25.

[46] Seufert, Michael, Egger, Sebastian, Slanina, Martin, Zinner, Thomas, Hoßfeld, Tobias, and Tran-Gia, Phuoc. A survey on quality of experience of http adaptive streaming. *IEEE Communications Surveys & Tutorials 17*, 1 (2014), 469–492.

[47] Sitaraman, Ramesh K. Network performance: Does it really matter to users and by how much? In *Proc. COMSNETS* (2013).

[48] Spiteri, Kevin. Sabre: Simulating ABR environments. https://github.com/UMass-LIDS/sabre.

[49] Spiteri, Kevin. Sabre360: Simulation testbed for 360° videos. https://github.com/UMass-LIDS/sabre360.

[50] Spiteri, Kevin, Sitaraman, Ramesh, and Sparacio, Daniel. From theory to practice: Improving bitrate adaptation in the DASH reference player. *ACM Trans. Multimedia Comput. Commun. Appl. 15*, 2s (July 2019).

[51] Spiteri, Kevin, Urgaonkar, Rahul, and Sitaraman, Ramesh K. BOLA: near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking 28*, 4 (2020), 1698–1711.

[52] Stockhammer, Thomas. Dynamic Adaptive Streaming over HTTP – Standards and Design Principles. In *Proc. ACM MMSys* (2011).

[53] Stohr, Denny, Frömmgen, Alexander, Rizk, Amr, Zink, Michael, Steinmetz, Ralf, and Effelsberg, Wolfgang. Where are the sweet spots? a systematic approach to reproducible dash player comparisons. In *Proceedings of the 2017 ACM on Multimedia Conference* (2017), ACM.

[54] Systems, Wowza Media. Low latency streaming. `https://www.wowza.com/low-latency`, accessed 2019.

[55] Systems, Wowza Media. What is low latency, and who needs it? `https://www.wowza.com/blog/what-is-low-latency-and-who-needs-it`, accessed 2019.

[56] van der Hooft, J., Petrangeli, S., Wauters, T., Huysegems, R., Alface, P. R., Bostoen, T., and De Turck, F. HTTP/2-based adaptive streaming of HEVC video over 4G/LTE networks. *IEEE Communications Letters 20*, 11 (2016), 2177–2180.

[57] Vandalore, Bobby, Feng, Wu-chi, Jain, Raj, and Fahmy, Sonia. A survey of application layer techniques for adaptive streaming of multimedia. *Real-Time Imaging 7*, 3 (2001), 221–235.

[58] Wang, Zhou, Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. Image Quality Assessment: From Error Visibility to Structural Similarity. *Trans. Img. Proc. 13*, 4 (Apr. 2004).

[59] Wu, Chenglei, Tan, Zhihao, Wang, Zhi, and Yang, Shiqiang. A dataset for exploring user behaviors in VR spherical video streaming. In *Proceedings of the 8th ACM on Multimedia Systems Conference* (2017), pp. 193–198.

[60] Yahia, Mariem Ben, Louedec, Yannick Le, Simon, Gwendal, Nuaymi, Loutfi, and Corbillon, Xavier. HTTP/2-based frame discarding for low-latency adaptive video streaming. *ACM Trans. Multimedia Comput. Commun. Appl. 15*, 1 (Feb. 2019), 18:1–18:23.

[61] Yin, Xiaoqi, Jindal, Abhishek, Sekar, Vyas, and Sinopoli, Bruno. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proc. ACM SIGCOMM* (2015).

[62] Zambelli, Alex. IIS smooth streaming technical overview. *Microsoft Corporation* (2009).

[63] Zink, Michael, Sitaraman, Ramesh, and Nahrstedt, Klara. Scalable 360° video stream delivery: Challenges, solutions, and opportunities. *Proceedings of the IEEE 107*, 4 (2019), 639–650.