From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player

Kevin Spiteri
UMass, Amherst
kspiteri@cs.umass.edu

Ramesh Sitaraman
UMass, Amherst & Akamai Tech.
ramesh@cs.umass.edu

Daniel Sparacio*
CBS Interactive
daniel.sparacio@cbsinteractive.com

ABSTRACT

Modern video streaming uses adaptive bitrate (ABR) algorithms than run inside video players and continually adjust the quality (i.e., bitrate) of the video segments that are downloaded and rendered to the user. To maximize the quality-of-experience of the user, ABR algorithms must stream at a high bitrate with low rebuffering and low bitrate oscillations. Further, a good ABR algorithm is responsive to user and network events and can be used in demanding scenarios such as low-latency live streaming. Recent research papers provide an abundance of ABR algorithms, but fall short on many of the above real-world requirements.

We develop `Sabre`, an open-source publicly-available simulation tool that enables fast and accurate simulation of adaptive streaming environments. We used `Sabre` to design and evaluate `BOLA-E` and `DYNAMIC`, two novel ABR algorithms. 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 very well for live streams that require low latency, a challenging scenario for ABR algorithms. Overall, our algorithms offer superior video QoE and responsiveness for real-life adaptive video streaming, in comparison to the state-of-the-art. *Importantly all three algorithms presented in this paper are now part of the official DASH reference player* `dash.js` *and are being used by video providers in production environments.* While our evaluation and implementation are focused on the DASH environment, our algorithms are equally applicable to other adaptive streaming formats such as Apple HLS.

CCS CONCEPTS

• **Information systems → Multimedia streaming**;

KEYWORDS

video streaming, video QoE, bitrate adaptation

*This work was performed when Daniel Sparacio was at Akamai Technologies.

ACM Reference Format:
Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. 2018. From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player. In *MMSys'18: 9th ACM Multimedia Systems Conference, June 12–15, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3204949.3204953

1 INTRODUCTION

Online video viewership is growing at a rapid pace and video traffic now dominates the Internet. Videos accounted for 76% of the consumer Internet traffic in 2016 and that share is predicted to increase to 82% by 2021 [3]. The rapid increase in online video viewership has resulted in a major rise in the diversity of users who watch videos. Users use a wide range of devices to watch videos, including Internet-enabled televisions, desktops, tablets, and cell phones. Further, they are connected to the Internet with widely different data rates. Such connections include multiple generations of cellular technology, WiFi, cable, DSL, and fiber-to-the-home.

Video providers are equally diverse and include movie sites (such as Hulu and Netflix), news portals (such as CNN and BBC), social networks (such as Facebook), and live sports channels (such as ESPN and MLB). The quality-of-experience (QoE) of users who watch videos is a central concern for video providers. Recent research [7, 13] has underscored the impact of poor video QoE on users. It is known that a video that starts up slowly, or rebuffers (i.e., freezes) in the middle, or plays at a low quality (i.e., bitrate) can lead users to abandoning the video or watching less of it. Consequently, video providers who rely on engaging their audience for their business goals place strong emphasis on providing a high QoE for their users.

Maximizing the QoE of the user involves factors that are often in conflict. On the one hand, it is desirable to play the video at the best quality at the highest encoded bitrate. On the other hand, it is also desirable to play the video continuously without the freezes caused by rebuffering. However, these two factors can conflict. For instance, playing the video at HD (high-definition) quality at a few Mbps provides a richer experience than playing it at standard definition (SD) of a few hundred kbps. However, playing a video at a high bitrate that is more than the sustainable network throughput between the server and the client (i.e., video player) will cause the client to rebuffer, since the client is unable to download the video at the rate at which it is being played. Thus, providing good QoE requires dynamically adapting the video bitrate to the network throughput, providing the highest video quality without rebuffering.

1.1 Adaptive Video Streaming

Adaptive video streaming is a popular approach for adapting the video presentation to the user's device and connectivity. In this approach, each video is partitioned into *segments* where each segment corresponds to a few seconds of play. Each segment is encoded in a number of different bitrates to accommodate the vastly different devices and network connectivities. Video bitrates range widely from 4K quality (~ 25 Mbps) to SD quality (~ few hundred kbps). When a client plays a video, it fetches the segments in sequence over HTTP. An *adaptive bit rate (ABR) algorithm* that is executed within the client can vary the bitrate of each downloaded segment in accordance with the current network throughput. Specifically, during the course of the video playback, the client steps down the bitrate when the network connection degrades and steps up the bitrate to provide a richer experience when the network improves.

To prevent video freezes, the client has a buffer that can store a number of segments. The client prefetches segments from the server *ahead* of when they need to be played out and stores them in the buffer. The segments are removed from the buffer in FIFO order and played out for the user. If the buffer is empty and there are no more segments to play out, the video playback must freeze until the next segment is received, resulting in a rebuffering event.

1.1.1 ABR Algorithms. The design and implementation of ABR algorithms is the focus of our work. A key goal of the ABR algorithm is to play the video at the highest bitrates without rebuffering. To achieve that end, the ABR algorithm carefully orchestrates what segments are downloaded, when, and at what bitrates. Additionally, ABR algorithms also attempt to minimize bitrate oscillations where the bitrates are switched frequently causing the user to perceive frequent changes in video quality. ABR algorithms have received significant attention in recent years and are classified into three broad categories: *throughput-based, buffer-based,* and *hybrid* schemes.
(1) Throughput-based algorithms work by estimating the network throughput available between the client and server and using that estimate to decide on bitrate of next segment that is to be downloaded. Such algorithms include `Festive` [12], `PANDA` [14], and `Squad` [23].
(2) Buffer-based algorithms predominantly use the level to which the buffer is full to decide on the bitrate of the next segment. Note that the buffer level is an (indirect) indicator of network throughput, as a lower (resp., higher) buffer level would indicate that the network throughput has recently been less (resp., more) than anticipated. Thus, a buffer-based algorithm would choose a higher bitrate when the buffer level is higher and lower if buffer level is lower. Such algorithms include `BBA` [11] and `BOLA` [18].
(3) Hybrid algorithms use both throughput prediction and buffer levels in an attempt to exploit the advantages of both. Such algorithms include `ELASTIC` [6], `MPC` [24], and `ABMA+` [2].

1.1.2 Adaptive Video Streaming Formats. A significant fraction of the world's online videos uses adaptive video streaming. The popular proprietary formats include Apple HTTP Live Streaming (HLS) [16], Microsoft Smooth Streaming [25], and Adobe HTTP Dynamic Streaming [1]. In 2012, the International Organization for Standardization (ISO) ratified a new open standard for adaptive video streaming called MPEG-DASH (or, simply, DASH) [19]. While both the proprietary formats and DASH are similar in technology, DASH has enormous potential for being the single open standard that could replace multiple proprietary standards in the future.

The significant potential of DASH has brought together most of the major players in the video industry together to form the DASH Industry Forum (DASH-IF) that maintains and promotes the DASH standard. In particular, it maintains a DASH reference client (i.e., video player) called `dash.js` [8] that encapsulates the standard and its best practices. Video providers wishing to use DASH often use the reference client `dash.js` to build their own video players. While our work on ABR algorithms is applicable to any adaptive video streaming protocol, we implement and empirically evaluate our approach within the DASH framework.

1.2 Our Contributions

The primary contributions of our work follow.
(1) We design and implement two novel ABR algorithms: `BOLA-E` and `DYNAMIC`. Both algorithms effectively minimize rebuffering and bitrate oscillations, while maximizing the average bitrate of the video stream viewed by the user. Further, both algorithms respond quickly to user events such as when the user starts or seeks in a video. Finally, we show that both `BOLA-E` and `DYNAMIC` provide a high QoE even for videos with the most stringent requirements, such as low-latency live streaming.
(2) We design and implement a segment replacement algorithm called `FAST SWITCHING` that judiciously replaces low-bitrate segments in the client's buffer with high-bitrate ones, whenever possible. If the network throughput increases during playback, `FAST SWITCHING` responds quickly to the network event and allows a user to experience the positive effects of that increase more quickly, since the low-bitrate segments in the client's buffer can be replaced with high-bitrate ones when the throughput is high. In our experiments, we showed that when the network throughput increases during playback, the client would see a higher-quality video 50s sooner with `FAST SWITCHING` than without.
(3) We have created a tool called `Sabre` that is valuable for simulating ABR environments. Its architecture is similar to that of the DASH reference player `dash.js` and it can be used to develop and evaluate ABR algorithms. `Sabre` simulates the player environment accurately for any specified video description and network traces. It outputs QoE metrics of interest such as segment bitrates and rebuffer events. *We published the tool as an open-source project that is available publicly at https://github.com/UMass-LIDS/sabre so that others can use it for ABR algorithm development.* While the tool can be used with any adaptive streaming format, we expect it to be particular useful to the DASH community.
(4) Finally, the foremost contribution of our work is that we have implemented `BOLA-E`, `DYNAMIC` and `FAST SWITCHING` in the standard DASH reference player `dash.js`. As of the current version (`dash.js` version 2.6.7) [8], `DYNAMIC` is the default ABR algorithm provided by the standard player. Both `BOLA-E` (referred to as just `BOLA` in the standard implementation) and `FAST SWITCHING` are options that users can select[1]. Consequently, our work has significantly improved the standard DASH reference player. *And, the algorithms described in this paper are actively used by video providers in production, as they build their own video players based on the standard reference player.* In Appendix A, we describe the `dash.js` architecture that is of independent interest, and we describe how we implemented our algorithms within that architecture. While the ABR algorithms are implemented in the DASH framework, the main

[1]The ABR algorithms can be selected by clicking the "Show Options" button in [8].

ideas for improving QoE are generally applicable to any adaptive video streaming system.

Roadmap. The rest of the paper is organized as follows. First, we provide some background on adaptive bitrate streaming and describe our approach to designing ABR algorithms (Section 2). Next we describe `Sabre`, a tool that we developed for simulating ABR environments and that will be publicly-available as open source (Section 3). `Sabre` is used for evaluating our algorithms in the rest of the paper. We then develop and evaluate `BOLA-E` (Section 4), `DYNAMIC` (Section 5) and `FAST SWITCHING` (Section 6). We then describe related work (Section 7) and conclude (Section 8). Finally, we describe how we implemented our algorithms in `dash.js` (Appendix A) and give implementation details about `Sabre` (Appendix B).

2 OUR APPROACH TO IMPROVING ABR ALGORITHMS

Our goal is to improve ABR algorithms that are currently used in practice. While our algorithmic techniques apply equally to both proprietary and open adaptive video streaming formats, we focus our implementation and evaluation on DASH. Our focus on DASH is for two reasons. First, we believe that the open standard allows for greater flexibility for trying out new ideas and innovations. In fact, the ABR algorithms described in this paper (`BOLA-E`, `DYNAMIC`, and `FAST SWITCHING`) are currently part of the standard reference player `dash.js`. Second, we believe that the DASH open standard will become increasingly important as more video producers migrate to the standard, amplifying the real-world impact of our work.

2.1 Requirements for ABR algorithms

We started our ABR research by getting extensive feedback from video providers and users across a spectrum of the media industry. 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.

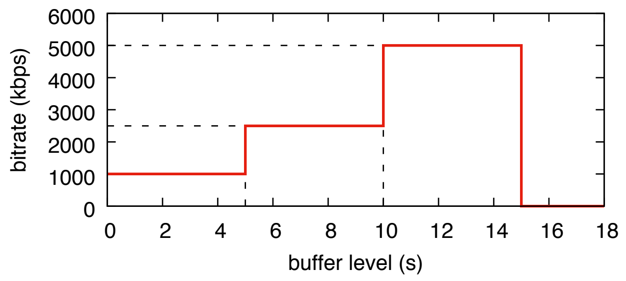


Figure 1: Example of a bitrate selection function with buffer capacity is 18s with thresholds at 5s, 10s, and 15s.

State-of-the-art ABR algorithms known in the literature often fall short on some of these requirements. These requirements formed the guiding principle for the ABR algorithm development and implementation we describe in the rest of the paper.

2.2 Overview of our design and implementation of ABR algorithms

We started out by implementing a buffer-based scheme called `BOLA` [18] that is to our knowledge the only known online ABR algorithm with provable optimality guarantees within a utility framework. `BOLA` utilizes Lyapunov optimal control to make ABR decisions based on buffer levels to maximize an arbitrary utility function that combines the two key QoE metrics of video bitrate and rebuffering. In particular, it is shown in [18] that `BOLA` asymptotically achieves utility that is within an additive factor of optimal.

Like other buffer-based ABR algorithms, `BOLA` uses a *bitrate selection function* that maps the current *buffer level* to the bitrate (in kbps) of the next segment to be downloaded, where buffer level is the total number of seconds of video segments stored in the buffer. The buffer may not exceed the *buffer capacity,* which is the total number of seconds of video that the buffer can store. As noted earlier, buffer capacity is a lower bound on live stream latency, and so a live stream that uses low latency can only use a small buffer. Figure 1 shows an example of a bitrate selection function for a video that is encoded in three bitrates (1000 kbps, 2500 kbps, and 5000 kbps) and has a buffer capacity of 18s.

Video segments typically enter and exit the buffer in FIFO fashion. As segments get played out from the front of the buffer, new segments get added to the tail of the buffer. *Segment length* is the number of seconds of video in the segment, and *segment size* is the number of bits in the segment. Note that segment lengths are usually fixed (say, 3s), while segment sizes can vary, especially for VBR videos. As the buffer level increases, the ABR algorithm views that as a sign of good network throughput, and it increases the video bitrate by consulting its bitrate selection function. The buffer levels where the bitrate changes are called *thresholds.* In Figure 1, the thresholds are 5s, 10s, and 15s.

We started out by implementing and evaluating `BOLA` in the DASH reference player `dash.js`. While `BOLA` provided a high bitrate without significant buffering or oscillations, it fell short on the other requirements of Section 2.1 that are important for a real-world production implementation. In particular, since `BOLA` predominantly used the buffer levels for decision making, it did not respond quickly to user events such as startup and seeking when

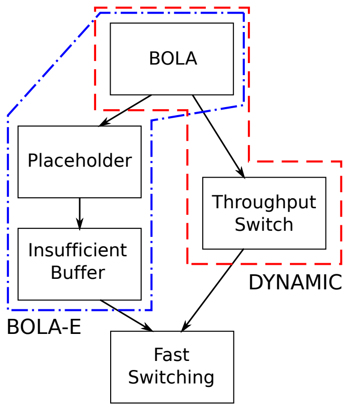


Figure 2: Overview of our design and production implementation of ABR algorithms for `dash.js`.

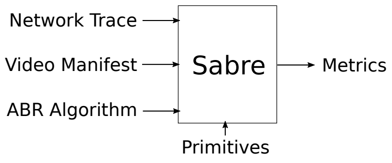


Figure 3: `Sabre`: Inputs, Outputs, and Primitives.

the buffer starts out empty. It also did 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` [11]. To improve `BOLA`, we took different approaches shown in Figure 2 and described below.

2.2.1 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 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 4.

2.2.2 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 producers. We present `DYNAMIC` and evaluate it in Section 5.

2.2.3 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 producers in conjunction with any ABR algorithm, including the default `DYNAMIC` or the optional `BOLA-E`. We present `FAST SWITCHING` and evaluate it in Section 6.

3 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 conditions of a practical player. We developed `Sabre`, an accurate tool for **s**imulating **ABR** **e**nvironments 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. Further details about `Sabre` are provided in Appendix B.

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.

The architecture of `Sabre` is shown in Figure 3. `Sabre` inputs a *network trace* that describes the time sequence of network conditions between the client and server, a *video description* that describes the video that the client downloads from the server, and the *ABR algorithm* that is used by the client to perform bitrate adaptation. `Sabre` outputs metrics that are key to video QoE such as bitrate of rendered segments, rebuffer events, and bitrate oscillations. `Sabre` also provides a set of primitives that the algorithm developer can use to create the ABR algorithm.

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.

Using `Sabre` the user can rapidly design, prototype, and test ABR algorithms. We used `Sabre` to empirically evaluate algorithms presented in this paper prior to their production implementation in `dash.js`. *We made* `Sabre` *open source and publicly available to the community on GitHub, so that others can use and continue to develop the tool.*

3.1 Inputs

The inputs to `Sabre` are described below, and are described in more detail in Appendix B.

(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 $C[i, j]$. $1 \le i \le N$, $1 \le j \le M$, 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 can be provided to `Sabre` as a Python module. The ABR algorithm is invoked before downloading a new segment. Python modules for the algorithms in this paper such as `BOLA-E` and `DYNAMIC` are available with the `Sabre` software. However, the user may also develop their own ABR algorithms to simulate with `Sabre`.

3.2 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 paper. 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

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

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

3.3 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` [10] and in the open source HLS player `hls.js` [5]. The implementation of `Sabre` is modular enough for the user to provide additional throughput or other primitives.

3.4 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. See Appendix B for more details.

`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. By playing the same video with and without audio for several actual videos, we have seen that the effect of the audio stream can be represented adequately by reducing the network throughput available for the video stream in the `Sabre` simulation.

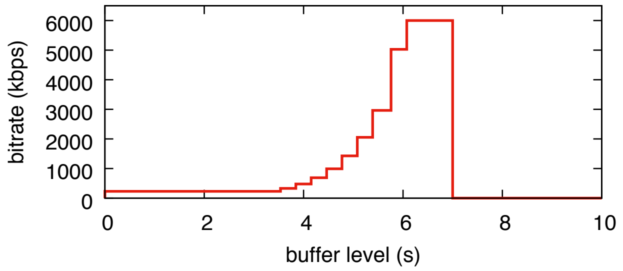
3.5 Network traces used with `Sabre` in our work

(1) *3G traces.* We use 3G traces from [17], 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.
(2) *4G traces.* We use 4G traces from [22], 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 [4]. We obtain throughput traces from measurements in the web browsing category[2], with each data point representing the throughput for 5s. The traces have a 5s granularity.

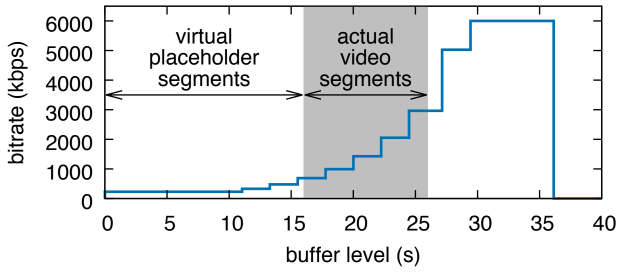
[2]We parse and use the traces in a manner similar to [15] (https://github.com/hongzimao/pensieve/tree/master/traces/fcc).

Table 1: Segment Bitrates for the Big Buck Bunny Movie

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



(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 4: Buffer expansion allows BOLA-E to have a larger separation between thresholds, reducing oscillations.

3.6 Video descriptions for Sabre in our work

We used the Big Buck Bunny Movie [9], a ten minute movie, for our simulations. Table 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[3] 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[4] 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.

[3]We use the same encoding used by [18].
[4]We use the set of bitrates recommended for YouTube (https://support.google.com/youtube/answer/1722171)

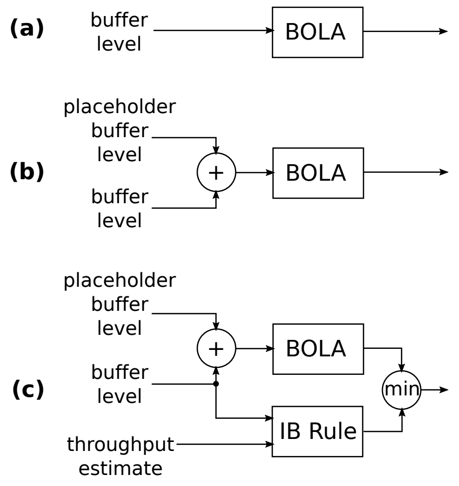


Figure 5: The evolution of BOLA-E. (a) The original BOLA. (b) Adding the placeholder algorithm for BOLA-PL. (c) Adding the insufficient buffer rule for BOLA-E.

4 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 proposed to mitigate slow startup in buffer-based algorithms [11, 18], but the heuristics still fall short of the performance achieved by throughput-based algorithms in the transient period. In Section 4.1 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 Figure 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 4.2 we design and implement the insufficient buffer rule and use it, together with buffer expansion, to overcome this issue. Figure 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.

4.1 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 virtual *placeholder segments* in the buffer, as and when needed. The placeholder algorithm can also remove the placeholder segments when they are not 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 Figure 4(a). We can pick the buffer level (x-axis) that corresponds to the bitrate (y-axis) chosen in step (2).

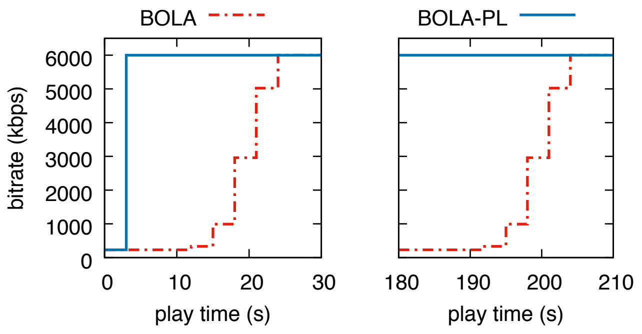
(4) Insert enough virtual placeholder segments in the buffer to obtain the desired buffer level. That is, the number of placeholder segments that are inserted equals the desired buffer level from step (3) minus the total size of the actual segments in the buffer.

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 that it deems is not sustainable, even though the buffer level is high enough to warrant that switch. In this situation, the placeholder algorithm attempts to reduce the buffer level to the appropriate value by removing placeholder segments.

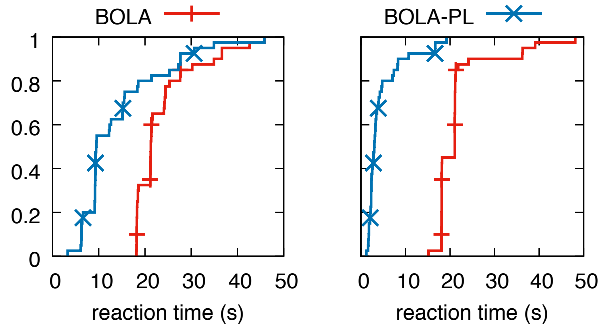
4.1.1 Evaluation. We now evaluate the placeholder algorithm for responsiveness to user events such as startups and seeks. First, we use a synthetic network trace that keeps the throughput relatively steady at 8 Mbps. We use the SD video described in Section 3.6. We then use Sabre to evaluate BOLA without 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.

Figure 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



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

Figure 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 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.

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.

Figure 7 compares the startup and seek performance of BOLA and BOLA-PL for the 4G traces described in Section 3.5. 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 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.

4.2 Insufficient Buffer Rule

We now propose a solution to the problem of avoiding oscillations in low-latency live streaming. The low latency requirement implies

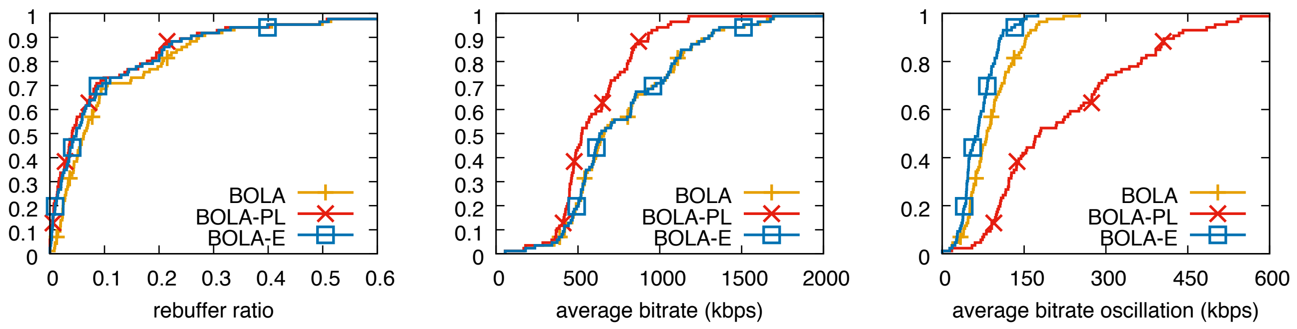


Figure 8: CDFs of the QoE metrics for BOLA, BOLA-PL and BOLA-E when streaming the SD video with a buffer capacity of 10s for 86 3G traces. BOLA-E significantly reduces oscillations.

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. Figure 4(a) shows an example of BOLA's bitrate switching thresholds for a low latency live stream with a buffer capacity of 10s. Figure 4(b) shows how it can be "stretched" to a larger buffer by modifying BOLA's parameters V and γ. The thresholds in Figure 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 Figure 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.

4.2.1 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 Figures 6 and 7. Next, we evaluated these algorithms on both 3G and 4G traces described in Section 3.5 with a 10s buffer capacity to evaluate their potential for rebuffering and oscillations. Figure 8 shows that BOLA-PL and BOLA-E have nearly identical rebuffering behavior for the 3G traces. However, BOLA-E that uses the buffer expansion and the insufficient buffer rule has a higher average bitrate and much fewer oscillations than BOLA-PL. In particular, BOLA-E had a median average bitrate of 638 kbps versus 524 kbps for BOLA-PL. Also, BOLA-E had a median bitrate oscillation of 65 kbps versus 179 kbps for BOLA-PL. We also included metrics for BOLA in Figure 8 to show that, while BOLA-E improves reaction time, it does not degrade the steady-state QoE metrics. In fact, it reduces rebuffering and bitrate oscillations. The empirical results for 4G traces were similar to that of the 3G traces and we do not include them here for space limitations.

5 DYNAMIC: BOLA WITH THROUGHPUT

Section 4 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 Figure 2).

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 Figure 9.

THROUGHPUT is a simple heuristic that first estimates the network throughput by using the sliding-window primitive described in

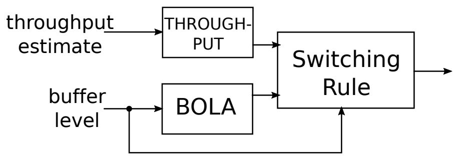


Figure 9: The DYNAMIC algorithms combines BOLA and THROUGHPUT.

Section 3.3 and then picks the 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[5] 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.

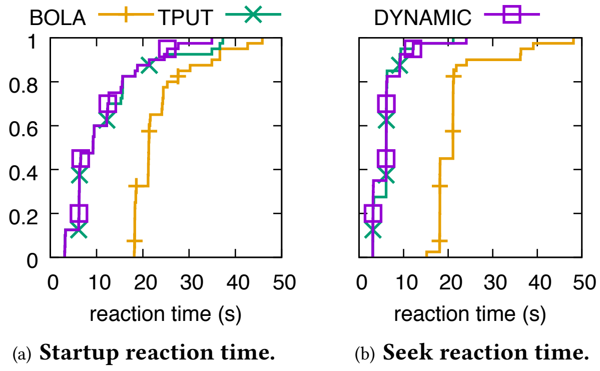
5.1 Evaluation

First, we study the response time of DYNAMIC with respect to BOLA and THROUGHPUT for startup and seek events using a 25s buffer. Figure 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 Figure 11.

Figure 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 Figure 11(a), is 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 90th 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 Figure 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

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



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

Figure 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.

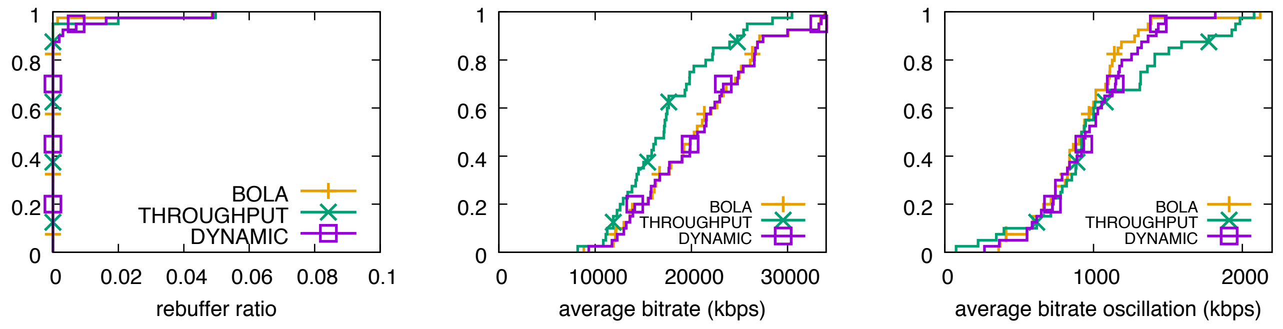
high throughput comes at the cost of excessive oscillations. BOLA has more oscillations than THROUGHPUT and DYNAMIC. In particular, at the median value, THROUGHPUT and DYNAMIC have 1089 kbps bitrate oscillations, while BOLA has higher oscillations at 2465 kbps. In summary, for a typical live setting, 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 response in startup and seek scenarios, making it a good choice overall as an ABR algorithm.

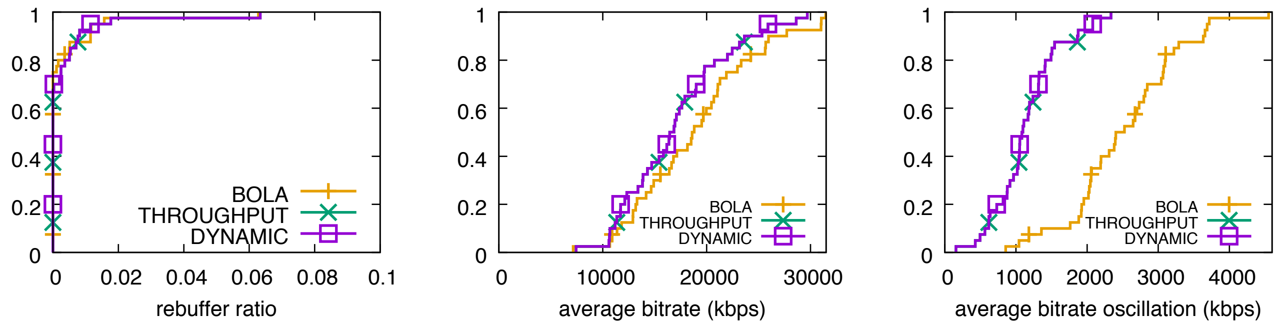
6 FAST SWITCHING: A SEGMENT REPLACEMENT ALGORITHM

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 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



(a) **VOD streaming with a 25s buffer.**



(b) **Live streaming with a 10s buffer.**

Figure 11: `DYNAMIC` combines strengths of `BOLA` and `THROUGHPUT`: It gets the higher bitrate of `BOLA` for VOD with a large buffer capacity and also the low oscillations of `THROUGHPUT` for live streaming with a small buffer capacity.

`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 within 1.5 × (the segment length) of the play head 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×(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.

6.1 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 3.6. For the SD video, we use the FCC traces described in Section 3.5 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.

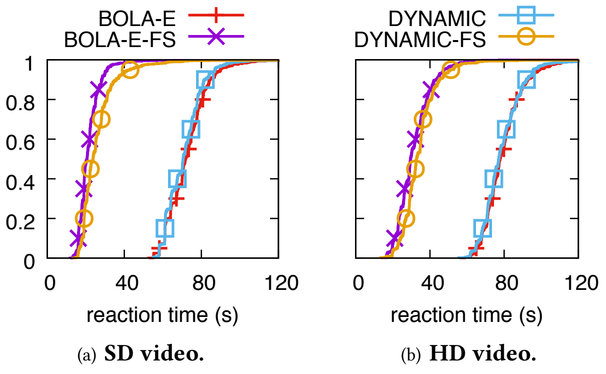
For each trace, some time after the network throughput increases, the viewer starts to see the video at a higher bitrate that can be sustained by the higher throughput. We measure the reaction time as the elapsed time from when the network throughput increased to when the user started viewing the video 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.

Figure 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 that does not use 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 can mitigate 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 Figure 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.

Figure 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 90th percentile, it increases bitrate oscillations by 306 kbps.



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

Figure 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.

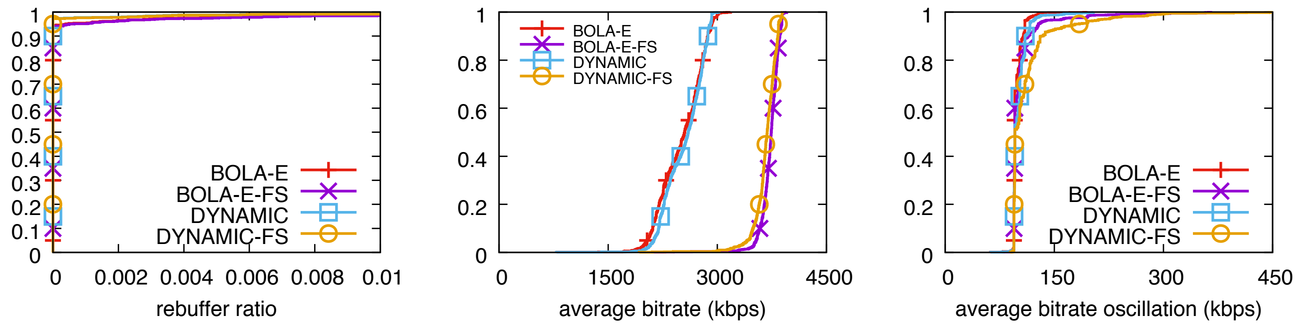
7 RELATED WORK

Throughput-based algorithms use past download history to predict the future. Festive [12] is one such algorithm that also aims to aid fairness by delaying downloads. PANDA [14] is another throughput-based approach that aims to improve fairness by probing the network capacity. By accounting for fairness, Festive and PANDA choose bitrates conservatively. Squad [23] performs bitrate switching while minimizing bitrate variability using a novel metric called spectrum.

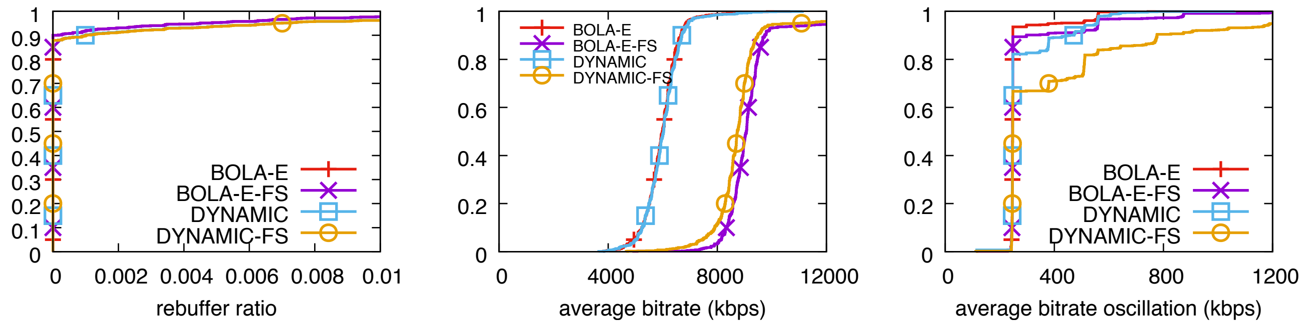
Alternatively, buffer-based algorithms use a bitrate selection function to choose the bitrate based on the buffer level. One such algorithm is BBA [11]. It needs a large buffer capacity (a four-minute buffer is used for evaluation) for stable operation. BOLA [18] derives a bitrate selection function, presenting a utility framework and using Lyapunov optimization techniques to prove the approach is asymptotically near-optimal in steady-state. It also uses some throughput information to prevent oscillations, enabling stable streaming at lower buffer capacities.

Other ABR algorithms are hybrid and use both throughput and buffer level information. ELASTIC [6] designs a feedback control system that attempts to keep the buffer level close to some predefined set-point. However, it can be slow to react to some throughput changes. MPC [24] presents another utility framework and uses a model predictive control algorithm to optimize utility within a finite horizon. MPC uses the buffer level by allowing the algorithm to be more or less aggressive. One drawback of MPC is that the optimization process requires significant computation which has to be precomputed offline and delivered to the video player. ABMA+ [2] starts with the goal of minimizing rebuffer probability. Similar to MPC, it relies on complex estimation calculations which are precomputed offline. The offline computations make the implementation more complex for both MPC and ABMA+.

There is also recent work on using machine learning for ABR algorithms. C2SP [21] involves learning a Hidden Markov Model for throughput prediction. It performs ABR in three stages: an offline prediction engine that does model training and clustering model learning, an online throughput prediction stage, and a bitrate



(a) **Streaming the SD video.**



(b) **Streaming the HD video.**

Figure 13: CDF of QoE metrics with and without `FAST SWITCHING` using a 25s buffer.

selection stage. `Pensieve` [15] trains a neural-network to choose the bitrate based on the video description and recent download metrics. Note that [2, 15, 21, 24] require precomputed tables, introducing additional complexity for deployment in production systems.

Regarding evaluation tools, `MACI` [20] 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.

8 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 paper offers superior video QoE and responsiveness for real-life adaptive video streaming. All three algorithms presented in this paper are now part of the official DASH reference player `dash.js` and are being used by video providers in production environments. For video providers wanting to choose `BOLA-E` versus `DYNAMIC` in the production player, we also compared them head-to-head. While `DYNAMIC` is currently the default choice, we found both algorithms performed similarly well in the QoE and responsiveness metrics.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported in part by the NSF under Grant No.: CNS-1413998.

REFERENCES

[1] Adobe. 2013. HTTP Dynamic Streaming Specification Version 3.0 FINAL. (2013).
[2] Andrzej Beben, P Wiśniewski, J Mongay Batalla, and Piotr Krawiec. 2016. ABMA+: lightweight and efficient algorithm for HTTP adaptive streaming. In *Proceedings of the 7th International Conference on Multimedia Systems*. ACM, 2.
[3] Cisco. 2017. Cisco Visual Networking Index: Forecast and Methodology, 2016–2021. (2017).
[4] Federal Communications Commision. 2016. Raw Data — Measuring Broadband America 2016. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-2016. (2016).
[5] Dailymotion. 2015. Introducing hls.js. http://engineering.dailymotion.com/introducing-hls-js/. (2015).
[6] Luca De Cicco, Vito Caldaralo, Vittorio Palmisano, and Saverio Mascolo. 2013. Elastic: a client-side controller for dynamic adaptive streaming over http (dash). In *Packet Video Workshop (PV), 2013 20th International*. IEEE, 1–8.
[7] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. 2011. Understanding the impact of video quality on user engagement. In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 362–373.

[8] DASH Industry Forum. 2018. DASH Reference Client 2.6.7. https://reference.dashif.org/dash.js/v2.6.7/samples/dash-if-reference-player/index.html. (2018).
[9] Blender Foundation. 2008. Big Buck Bunny Movie. https://peach.blender.org/. (2008).
[10] Google. 2015. Shaka Player. https://opensource.google.com/projects/shaka-player. (2015).
[11] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. 2015. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 187–198.
[12] Junchen Jiang, Vyas Sekar, and Hui Zhang. 2012. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies.* ACM, 97–108.
[13] S Shunmuga Krishnan and Ramesh K Sitaraman. 2013. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking* 21, 6 (2013), 2001–2014.
[14] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C Begen, and David Oran. 2014. Probe and adapt: Rate adaptation for HTTP video streaming at scale. *IEEE Journal on Selected Areas in Communications* 32, 4 (2014), 719–733.
[15] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. *Proceedings of the 2017 ACM SIGCOMM Conference.*
[16] R. Pantos and W. May. 2017. *HTTP Live Streaming.* RFC 8216. RFC Editor.
[17] Haakon Riiser, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. 2013. Commute path bandwidth traces from 3G networks: analysis and applications. In *Proceedings of the 4th ACM Multimedia Systems Conference.* ACM, 114–118.
[18] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. 2016. BOLA: near-optimal bitrate adaptation for online videos. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on.* IEEE, 1–9.
[19] Thomas Stockhammer. 2011. Dynamic adaptive streaming over HTTP — standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems.* ACM, 133–144.
[20] Denny Stohr, Alexander Frömmgen, Amr Rizk, Michael Zink, Ralf Steinmetz, and Wolfgang Effelsberg. 2017. Where are the sweet spots? A systematic approach to reproducible DASH Player comparisons. In *Proceedings of the 2017 ACM on Multimedia Conference.* ACM.
[21] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. 2016. CS2P: Improving Video Bitrate Selection and Adaptation with Data-Driven Throughput Prediction. In *SIGCOMM.*
[22] J. van der Hooft, S. Petrangeli, T. Wauters, R. Huysegems, P. R. Alface, T. Bostoen, and F. De Turck. 2016. HTTP/2-Based Adaptive Streaming of HEVC Video Over 4G/LTE Networks. *IEEE Communications Letters* 20, 11 (2016), 2177–2180.
[23] Cong Wang, Amr Rizk, and Michael Zink. 2016. Squad: A spectrum-based quality adaptation for dynamic adaptive streaming over HTTP. In *Proceedings of the 7th International Conference on Multimedia Systems.* ACM.
[24] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. 2015. A control-theoretic approach for dynamic adaptive video streaming over HTTP. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 325–338.
[25] Alex Zambelli. 2009. IIS smooth streaming technical overview. *Microsoft Corporation* (2009).

A ABR ALGORITHMS IN `DASH.JS`

The algorithms described in this paper are implemented in `dash.js` that is the reference player of the MPEG-DASH standard that is maintained by DASH Industry Forum (DASH-IF). DASH-IF is a consortium that includes most major participants in the video streaming industry and currently has 60+ members. Our code is part of the `dash.js` project repository that can be found at https://github.com/Dash-Industry-Forum/dash.js. Extensive documentation is available at https://github.com/Dash-Industry-Forum/dash.js/wiki, and API documentation is also available. The current version is 2.6.7 released on Mar 14th 2018.

The `dash.js` player is written in JavaScript and runs inside a web browser. Video rendering is delegated to the Media Source Extensions (MSE) as implemented by the browser. The player functionality[6] is handled by a number of modules internally known as *controllers.* We describe these controllers as they interact with our algorithms. Figure 14 shows a simplified overview of the player architecture. A streaming session starts when the player loads a manifest file which describes the video, known as the Media Presentation Description (MPD) . The `DashAdapter` parses the MPD to obtain information such as video length, segment length, encoded bitrates, and possible live-streaming details.

Our algorithms are incorporated in the `ScheduleController` and `abrController` as described below. The `ScheduleController` manages the high-level task of segment download. To download a segment, the `ScheduleController` queries the `abrController` to choose a bitrate, then instructs the `FragmentLoader`[7] to download the video segment at that bitrate. Once the segment is downloaded, the `BufferController` receives it and sends the bytes to the MSE buffer. After the MSE buffer has been updated, the `ScheduleController` can restart the process for the next segment. This process might need to be scheduled in the future for a number reasons. For example, the next segment might not yet be available in a live stream, or the buffer level might have reached the buffer capacity. The `ScheduleController` is the module where we implemented the `FAST SWITCHING` algorithm described in Section 6. It can be found in the source tree at `src/streaming/controllers/ScheduleController.js`.

The `abrController` manages a collection of ABR rules which can be found in the source tree at `src/streaming/rules/abr`. Figure 15 shows how the controller uses the rules to choose a bitrate. The controller maintains a collection of ABR rules, the first three of which were designed and implemented by us as described in the paper.

(1) **`ThroughputRule`:** Select bitrate based on the `THROUGHPUT` algorithms described in Section 5.

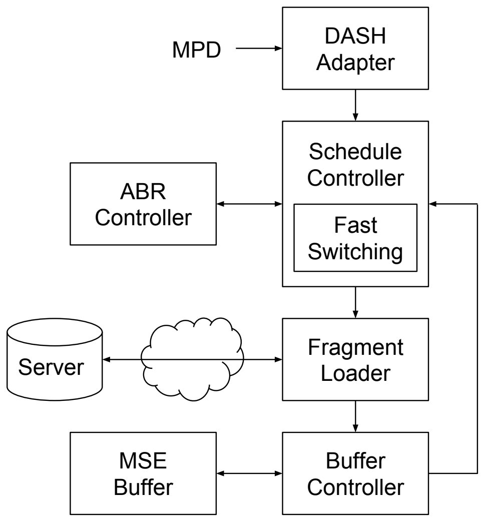


Figure 14: The simplified `dash.js` player architecture. Our work is incorporated in the Schedule and ABR controllers.

[6]We will not describe a number of `dash.js` features such as DRM support and subtitles; while these features are crucial for commercial streaming, their details are beyond the scope of this paper.

[7]In `dash.js` segments are sometimes referred to as fragments.

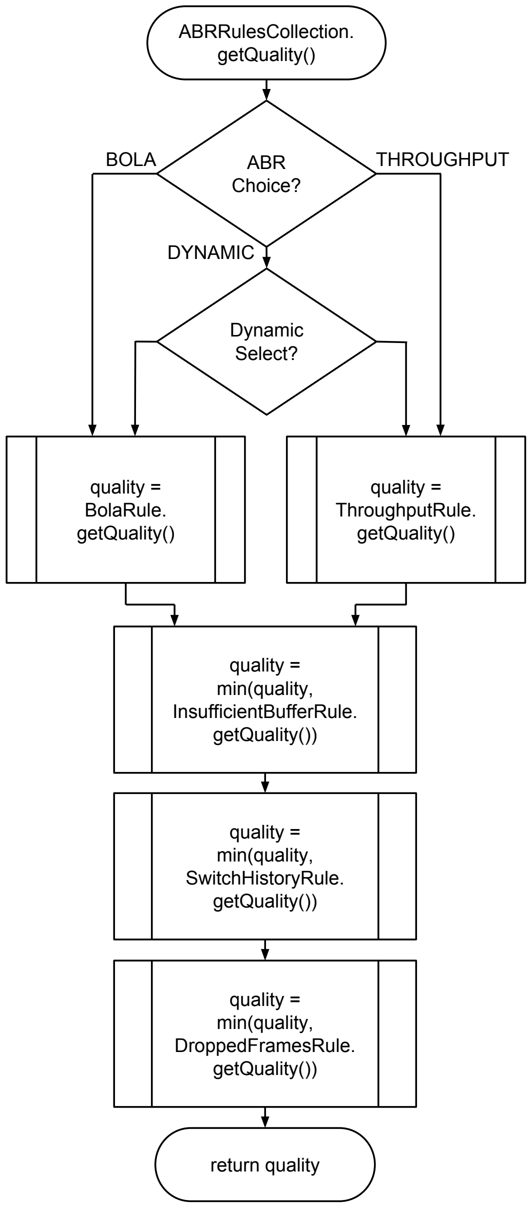


Figure 15: The `abrController` in the current `dash.js` player provides the video provider a choice of the three ABR algorithms described in this paper.

(2) **`BolaRule`:** Select bitrate based on the `BOLA-E` algorithm described in Section 4.
(3) **`InsufficientBufferRule`:** Limit bitrate based on the insufficient buffer rule described in Section 4.
(4) **`SwitchHistoryRule`:** Additional heuristic to detect and avoid any extreme bitrate oscillations allowed by the ABR algorithms.
(5) **`DroppedFramesRule`:** Additional heuristic to avoid using bitrates that exceed the device computational resources.

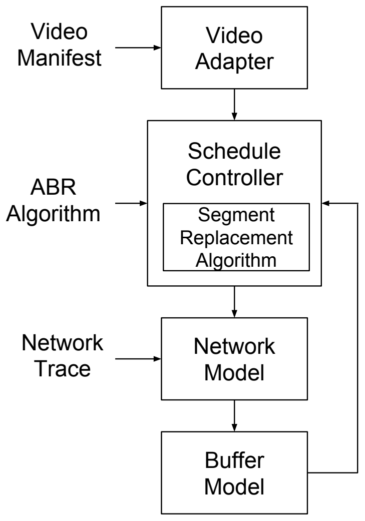


Figure 16: The `Sabre` architecture.

We modified the `abrController` to offer the video provider the choice of using `DYNAMIC` which is turned on by default, `BOLA`[8], or `THROUGHPUT`. As shown in Figure 15, when `DYNAMIC` is chosen, either `THROUGHPUT` or `BOLA` is selected according to the criteria described in Section 5. When the video provider optionally chooses either `THROUGHPUT` or `BOLA`, only the chosen algorithm is executed as shown in the figure.

Using our ABR algorithms in `dash.js`. An example player implementation based on `dash.js` can be found at http://reference.dashif.org/dash.js/. The implementation provides options to choose between the different ABR algorithms described in this paper. Developers can build their own video players based on `dash.js` as described on the project repository front page. The documentation indicates the development dependencies required and also provides a quick-start guide to setting up a basic player. The API documentation also shows how the ABR algorithm can be changed using a simple JavaScript command.

B SIMULATING ABR ALGORITHMS IN `SABRE`

`Sabre` is an open-source simulation environment for ABR algorithms licensed under the Simplified BSD License. It is a Python tool that facilitates initial development and quick evaluation of algorithms in an environment similar to real production players, without requiring the algorithm researchers to learn the often complex implementation details of a production player. `Sabre` takes a video description, network trace, and an ABR algorithm as inputs and gives a collection of QoE metrics as output. The software and documentation are available at https://github.com/UMass-LIDS/sabre.

Figure 16 shows an overview of the `Sabre` architecture. The video adapter can read a simplified manifest for the video that is being simulated. We simulate the download of video segments using a network model that can emulate network conditions from

[8]The algorithm described as `BOLA-E` in Section 4 is labeled as simply `BOLA` in the actual production implementation of `dash.js`.

a trace file. The segment replacement algorithm inside the schedule controller and the ABR algorithm can be linked in as inputs to Sabre, allowing new algorithms to be evaluated.

Sabre's architecture is similar to that of dash.js, as can be seen from the similarities between Figures 14 and 16. This helps algorithms developed within the Sabre environment to be easily implementable in 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.

B.1 Using Sabre for video player simulations

Sabre can be used either through the command line or programmatically. The inputs and outputs of Sabre are described below.

B.1.1 Video Manifest. The input video specification is given as a file in the JavaScript Object Notation (JSON) format. An example is shown below.

```
{
    "segment_duration_ms": 3000,
    "bitrates_kbps": [ 1000, 2000, 4000 ],
    "segment_sizes_bits": [
        [  3000000,  6000000, 12000000 ],
        [  3177800,  6311760, 12310936 ],
        [  2932704,  5854096, 11732072 ],
        [  2667248,  5652216, 11217520 ],
        [  3222248,  6181928, 12739472 ]
    ]
}
```

The file contains three fields: the segment duration, the list of bitrates, and the size of each video segment available. In the example, each video segment is 3s long, and the video is encoded at three bitrates: 1000, 2000 and 4000 kbps. The video is 15s long, and there are five segments available at each bitrate. Note that the segments sizes for a particular bitrate might be different because of VBR. The video manifests used in this paper are provided in the source tree at mmsys18/{bbb,bbb4k}.json.

B.1.2 ABR Algorithm. The ABR algorithms that are described in this paper are provided as Python classes in the main module src/sabre.py. The user can provide a new ABR algorithm as input by creating a new Python class. Detailed documentation of the interface is available at https://github.com/UMass-LIDS/sabre/wiki. An example of a simple user-defined ABR class is provided below.

```
class NewAbr(Abr):
    def get_quality_delay(self, segment_index):
        manifest = self.session.manifest
        bitrates = manifest.bitrates
        throughput = self.session.get_throughput()
        quality = 0
        while (quality + 1 < len(bitrates) and
               bitrates[quality + 1] <= throughput):
            quality += 1
        return quality
```

B.1.3 Segment Replacement Algorithm. The FAST SWITCHING algorithm described in this paper is provided as a Python class in the main module src/sabre.py. A new replacement algorithm can be provided by the user as an input by creating a new Python class. Detailed documentation of the interface is available at https://github.com/UMass-LIDS/sabre/wiki. An example of a simple user-defined replacment class is provided below.

```
class NewReplacement(Replacement):
    def check_replace(self, quality):
        buffer = self.session.buffer_contents
        for i in range(2, len(buffer)):
            if buffer[i] < quality:
                # return -ve index from end of buffer
                return i - len(buffer_contents)
        # if we arrive here, no switching occurs
        return None
```

B.1.4 Network Trace. The input network trace is given as a file in the JSON format. An example is shown below.

```
[
    {"duration_ms": 30000, "bandwidth_kbps": 5000,
     "latency_ms":  75},
    {"duration_ms": 30000, "bandwidth_kbps": 3000,
     "latency_ms": 150},
    {"duration_ms": 30000, "bandwidth_kbps": 1500,
     "latency_ms": 200},
    {"duration_ms": 30000, "bandwidth_kbps": 3000,
     "latency_ms": 150},
]
```

The file contains a list of four records representing different time periods and the network state in each of those periods. The first period lasts 30s, and while in this state the network allows a throughput of 5000 kbps, with a round-trip latency of 75 ms. When the four periods come to an end after two minutes, Sabre restarts at the top. The network traces used in this paper are provided in the source tree at mmsys18/{3Glogs,4Glogs,sd_fs,hd_fs}/.

B.1.5 Outputs. While Sabre runs a session for the provided inputs, it collects a set of QoE metrics. An example output is below.

```
total played bitrate: 1765337
time average played bitrate: 2956
total play time: 1791
total rebuffer: 0
rebuffer ratio: 0
time average rebuffer: 0
total rebuffer events: 0
time average rebuffer events: 0
total bitrate change: 341760
time average bitrate change: 572
reaction time: 3.252272
```

Sabre can also provide a detailed history of all downloads (successful or abandoned) and which segments were played back.

B.2 A Sabre script example

To generate the plots for this paper, we used an automated script that can be found in the source tree at mmsys18/generate.py. Documentation about the script can be found at https://github.com/UMass-LIDS/sabre/wiki/MMSys18Plots.