Catching the Moment With LoL$^+$ in Twitch-Like Low-Latency Live Streaming Platforms

Abdelhak Bentaleb, *Member, IEEE*, Mehmet N. Akcay, May Lim, Ali C. Begen, *Senior Member, IEEE*, and Roger Zimmermann, *Senior Member, IEEE*

***Abstract*—Our earlier Low-on-Latency (dubbed as LoL) solution offered an accurate bandwidth prediction and rate adaptation algorithm tailored for live streaming applications that targeted an end-to-end latency of up to two seconds. While LoL was a significant step forward in multi-bitrate low-latency live streaming, further experimentation and testing showed that there was room for improvement in three areas. First, LoL used hard-coded parameters computed from an offline training process in the rate adaptation algorithm and this was seen as a significant barrier in LoL's wide deployment. Second, LoL's objective was to maximize a collective QoE function. Yet, certain use cases have specific objectives besides the singular QoE and this had to be accommodated. Third, the adaptive playback speed control failed to produce satisfying results in some scenarios. Our goal in this paper is to address these areas and make LoL sufficiently robust to deploy. We refer to the enhanced solution as LoL$^+$, which has been integrated to the official dash.js player in v3.2.0.**

***Index Terms*—HAS, ABR, DASH, CMAF, low latency, chunked transfer encoding, adaptive playback speed, SOM.**

I. Introduction

With the rise of low-latency live (LLL) streaming applications such as Twitter's Periscope, Amazon's Twitch and Facebook's Live, and users' growing interest in eSports and streaming of live sports, the demand for low-latency services is higher than ever. As proprietary solutions such as Adobe's Real-time Messaging Protocol (RTMP) [3] fade away, HTTP adaptive streaming (HAS), using primarily the open Dynamic Adaptive Streaming over HTTP (DASH) standard and Apple's HTTP Live Streaming (HLS) protocol, dominates the market today. HAS has been doing a tremendously good job in delivering billions of live and on-demand streams every day in a cost-effective manner. By using multiple switchable versions of the same content, players trade off the video quality against the likelihood of rebuffering events when the available bandwidth drops. In live streaming, the latency (measured from the moment of capturing to the moment of rendering) also becomes part of this trade-off. At lower values, the latency plays a more critical role in rate adaptation as the player's breathing room for absorbing bandwidth drops narrows significantly. As opposed to the accustomed 30–60 seconds of latency for the traditional HAS, the target for LLL streaming applications is mostly five seconds or less [33].

Manuscript received October 17, 2020; revised February 7, 2021; accepted May 4, 2021. Date of publication May 12, 2021; date of current version May 11, 2022. This work was supported in part by National Research Foundation, Singapore under its AI Singapore Programme AISG under Grant AISG-100E-2019-047, in part by the Singapore Ministry of Education Academic Research Fund Tier 2 under MOE's official under Grant MOE2018-T2-1-103, in part by under Grant 31T102-UPAR-1-2017 from UAE University. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Chonggang Wang. *(Corresponding author: Abdelhak Bentaleb.)*

Abdelhak Bentaleb, May Lim, and Roger Zimmermann are with the School of Computing, National University of Singapore, Singapore 119077, Singapore (e-mail: bentaleb@comp.nus.edu.sg; maylim@comp.nus.edu.sg; rogerz@comp.nus.edu.sg).

Mehmet N. Akcay and Ali C. Begen are with the Ozyegin University, Istanbul 34794, Turkey (e-mail: necmettin.akcay@ozu.edu.tr; ali.begen@ozyegin.edu.tr).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TMM.2021.3079288.

Digital Object Identifier 10.1109/TMM.2021.3079288

A. Motivation

End-to-end latency is impacted by several serialized processes in the delivery workflow including capturing, encoding, packaging, publishing to the origin, delivering through a content delivery network (CDN), buffering, decoding and rendering. One way to reduce latency is to use short segment durations (1–2 seconds as opposed to 6–10 seconds), however, shorter segments naturally reduce encoding efficiency. Instead, the processes should be streamlined. This can be achieved using an appropriate packaging format such as the Common Media Application Format (CMAF) standard [17] and an appropriate transfer mechanism such as HTTP/1.1 chunked transfer encoding (CTE) (RFC 7230).

In CMAF, the duration of the media segment is decoupled from latency since a segment is generated and delivered in multiple small non-overlapping pieces, called chunks [6]. For example, a six-second and 30-f/s segment can be chunked at frame level (*i.e.*, 180 chunks, one every 33.3 ms), which results in a significant reduction in serial delays from capturing to rendering, and therefore, lowers latency. For media delivery, CTE is a data streaming mechanism that was introduced in HTTP/1.1, where the chunks of a segment are sent out and received by the player independently of one another, enabling chunks to be delivered without waiting for the segment to be fully encoded and packaged. For packaging, CMAF is a media container standard enabling the use of the chunk concept with CTE, while also unifying the media format for both DASH and HLS.

Despite the benefits of CMAF and CTE in satisfying LLL streaming requirements, they create new challenges for the player and tasks like throughput measurements, bitrate selection, buffer management and playback speed adaptation become non-trivial [6], [7], [25]. For example, when the player receives all the chunks of a segment, it would erroneously calculate the throughput to be almost equal to the encoding bitrate of that segment if it uses the basic formula of dividing the segment size by the total download duration. Consequently, the player would make less-than-ideal bitrate selections. On the other hand, the player must fetch the chunks of the live-edge segment that is still being encoded and packaged at the time of the request, limiting the playback buffer to a few seconds or less. Under such circumstances, a good buffer management mechanism is essential to avoid rebufferings and this requires a good playback speed control.

In this paper, we demonstrate player enhancements that are effective for media delivery in LLL streaming. To this end, we address the challenges mentioned above by proposing a bitrate selection module and a playback speed control module that improves viewer experience for LLL streaming as well as an accurate throughput measurement module for CTE. These modules are collectively called Low-on-Latency-plus (LoL$^+$). Our solution is an extension and enhancement to our earlier LoL solution [25]. LoL$^+$ addresses the following three shortcomings that were left as future work for LoL:

- LoL implements a learning-based adaptive bitrate (ABR) algorithm, which uses a self-organizing map (SOM) [23] for bitrate selection. This algorithm shows good performance under certain network conditions. However, it occasionally suffers from low performance due to initially tuned but then static weight values for the set of considered SOM features,[1]. In contrast, LoL$^+$ employs a two-step technique for weight selection. At the first stage (the beginning of a live session), it implements the k-means++ [11] approach to find suitable initial weights for every SOM feature. At the second stage (the steady state of a live session), it employs a weight assignment algorithm that dynamically adjusts the weights at every segment download boundary.
- LoL performance relies on a well-defined linear QoE function where the SOM model uses the QoE as one of the features to make bitrate selections. However, realistically the QoE function should differ depending on the streaming scenario, and thus, employing a fixed function limits LoL's deployment where it fails to work well. Conversely, LoL$^+$ uses each metric that influences the QoE individually as a feature in the SOM model. Hence, it allows more flexibility and deployability under different network conditions and streaming scenarios.
- LoL uses a heuristic approach to control the playback speed, which is largely based on the current latency. This technique is not sensitive enough to buffer declines, which results in frequent rebufferings. LoL$^+$ adopts a better-balanced hybrid approach to control the playback speed, jointly considering both latency and buffer level. This approach is better able to reduce rebufferings and latency deviations from its target.

[1]As shown in [11] the SOM model is sensitive to the initial weight values.

B. Key Contributions

Our contributions are four-fold:

- First, we propose a series of sophisticated and robust player improvements for LLL streaming. LoL$^+$ consists of five essential modules: (i) The *bitrate selection module* implements a learning-based ABR algorithm to choose a suitable bitrate at each segment download boundary. The ABR algorithm is based on a SOM model that considers multiple QoE metrics as well as variability in network conditions in the ABR formulation. (ii) The *playback speed control module* implements a hybrid playback speed algorithm that combines the current latency and buffer level to control the playback speed (*i.e.*, speed up, slow down or normal). (iii) The *throughput measurement module* accurately calculates the throughput by removing the idle times between the chunks of a segment through a three-step algorithm (chunk boundary identification, chunk filtering, and throughput calculation and smoothing). (iv) The *QoE evaluation module* computes the QoE considering five key metrics: selected bitrate, number of bitrate switches, rebuffering duration, latency and playback speed. Lastly, (v) the *weight selection module* implements a two-step dynamic weight assignment for the SOM model features.
- Second, we formulate the weight selection problem of the SOM model as an assignment problem [9] with convex optimization and linear inequality constraints. We use a mixed integer linear programming (MILP) framework [8] to model the optimization function and then solve it using a heuristic-based dynamic weight adjustment algorithm.
- Third, we design a learning-based ABR algorithm that adapts a SOM model. Our model considers the player context (*i.e.*, QoE metrics and network conditions such as the available bandwidth) to make bitrate selections.
- Fourth, we provide an implementation of LoL$^+$, which has been integrated into the official dash.js (v3.2.0) player [1]. We validate our solution through trace-driven experiments. The results show the superiority of LoL$^+$ against four state-of-the-art ABR algorithms for LLL streaming.

The rest of the paper continues with the related work specific to LLL streaming in Section II. Section III provides a detailed description of the LoL$^+$ solution. Implementation and experimental evaluation are presented in Section IV, followed by the conclusions in Section V.

II. Related Work

Over the past decade, many ABR algorithms have been developed, most of which have been designed for video-on-demand or unconstrained-latency live scenarios. In this section, we present the solutions for LLL streaming and start with the most recent ones. For a general overview of ABR algorithms, refer to [5].

Bentaleb *et al.* [6], [7] proposed the first ABR for CTE and CMAF, termed ACTE. ACTE comprises three entities: (i) a per-chunk sliding window-based bandwidth measurement, (ii) a recursive least squares algorithm for the bandwidth prediction for the next few seconds, and (iii) an ABR algorithm that takes into consideration both the predicted bandwidth and playback buffer occupancy. Similarly, Gutterman *et al.* [15]

designed STALLION, a sliding window-based algorithm to measure the mean and standard deviation of the bandwidth and latency for better bitrate selection. In [27], Peng *et al.* implemented a hybrid-based control ABR algorithm for LLL streaming. It consists of a heuristic for buffer-based playback rate control mechanism, a frame dropping technique for better QoE and low latency, and an ABR rule that uses Kaufman's moving average to select the bitrate. Swaminathan *et al.* [32] proposed a low-latency CTE approach that decoupled the live latency from the segment duration. The solution analytically evaluated the latency in three LLL scenarios: (1) a segment-based method where the player requests the segment only if it is fully available, (2) a server-wait method where the player requests a segment before it is ready at the server, and (3) a chunked encoding method where the player receives the chunks before the full segment is available. However, this work did not include ABR and rate adaptation. Combining multi-path capabilities and CTE, Houze *et al.* [16] designed a low-latency solution that splits and downloads multiple video frames from different paths while taking the frame sizes into account. Shuai *et al.* [29] analyzed the impact of the playback buffer occupancy on live streaming and proposed an ABR rule based on buffer stabilization for LLL streaming with two-second segments.

Leveraging HTTP/2 capabilities, van der Hooft *et al.* [35] described an HTTP/2 push-based approach for LLL streaming with very short segments. Similarly, Benyahia *et al.* [36] designed a frame discarding technique at the segment level by using the HTTP/2 stream resetting feature to meet a one-second latency constraint. Regarding the implementation of CMAF low latency, DASH-IF and Akamai provided a proof-of-concept in the dash.js player [1]. Essaili *et al.* [13] developed an HTML5 player with CMAF low-latency support.

Rather than relying on fixed heuristics, learning-based ABR algorithms learn from the environment and create suitable policies based on past data, and thus, they adapt to system dynamics. Several learning-based attempts have been proposed to efficiently tackle problems in video delivery. In [25], the LoL (Low-on-Latency) solution was designed for LLL streaming. It includes three modules: bitrate adaptation (both heuristic and learning-based), heuristic-based playback speed control and throughput measurement. With the same goal, the authors of [21] developed the L2A (Learn2Adapt) solution for LLL streaming and formulated the bitrate selection problem under a latency constraint as online convex optimization. Using a data-driven approach, Sun *et al.* [31] proposed CS2P, a bandwidth estimator that runs over two stages. It first learns the sessions with similar vital features (*e.g.*, ISP, geographical region, IP) and then groups similar sessions into clusters. Thereafter, for each cluster, it trains a hidden Markov model (HMM) to estimate the corresponding bandwidth. A number of other data-driven machine learning solutions have also emerged: AMP [4], CFA [18], PREM [39], LiveNAS [22], Pytheas [20] and DDS [12].

III. The LoL$^+$ Solution

We first present an overview of the dash.js player with the newly added LoL$^+$ modules, and then discuss each module in detail. A list of notations is presented in Table I.

TABLE I
List of the Key Symbols and Notations

Notation	Definitions
s_i	Segment i ($i \in [1, \ldots, K]$)
K	Cumulative number of segments ($= \lvert S \rvert$)
R_{s_i}	Bitrate selected (Kbps) at s_i
E_{s_i}	Rebuffering time (in seconds) at s_i
L_{s_i}	Latency (in seconds) at s_i
P_{s_i}	Playback speed (*e.g.*, 1, 0.95, 1.05) at s_i
H_{s_i}	Number of bitrate switches at s_i
S	List of downloaded segments ($S = \{s_1, \ldots, s_K\}$)
α	Bitrate reward factor
β	Rebuffering penalty factor
γ	Latency penalty factor
σ	Playback speed penalty factor
μ	Bitrate switching penalty factor
X_{s_i}	Measured throughput of segment s_i
Q_{s_i}	Size of segment s_i
T_{s_i}	Download time of segment s_i
$\tilde{T}_{s_i}$	Download time of segment s_i without the idle times
s_i^c	Chunk $c = \{1, \ldots, z_{s_i}\}$ of segment s_i
$q_{s_i^c}$	Size of chunk c in segment s_i
$t_{s_i^c}$	Download time of chunk c in segment s_i
$x_{s_i^c}$	Measured throughput of chunk c in segment s_i
z_{s_i}	Last chunk in segment s_i
$\lvert z_{s_i} \rvert$	Total number of chunks in segment s_i
$\tilde{z}_{s_i}$	Last of the remaining chunks after filtering in segment s_i
$\lvert \tilde{z}_{s_i} \rvert$	Total number of the remaining chunks after filtering in segment s_i
M_{s_i}	Set of SOM model neurons at segment s_i
j	Total number of neurons
$m_{s_i}^{\hat{j}}$	Neuron of SOM model ($m_{s_i}^{\hat{j}} \in M_{s_i}$)
$\mathbf{w}_{s_i}^{\star}$	Best weight vector for a neuron at segment s_i
W_{s_i}	Set of possible weight vectors at segment s_i
$w_{s_i}^{\hat{v}}$	Weight vector at s_i ($w_{s_i}^{\hat{v}} \in W_{s_i}$)
U_S^K	Total utility ($= \text{QoE}_S^K$)
u_{s_i}	Step utility at s_i
$\mathcal{M}_{s_i}^{(M,W)}$	Matrix of the achieved utilities for W_{s_i} at s_i
$u_{s_i}^{(m_{s_i}^{\hat{j}}, w_{s_i}^{\hat{v}})}$	Achieved step utility of assigning $w_{s_i}^{\hat{v}}$ to $m_{s_i}^{\hat{j}}$
V	Value vector for weight vector
$\mathcal{F}$	Objective function
L_{target}	Target latency
L_{ideal}	Ideal latency
P_{ideal}	Ideal playback speed
B_{ideal}	Ideal buffer occupancy
B_{s_i}	Playback buffer occupancy (in seconds) at s_i
B^{low}	Playback buffer occupancy level safe threshold
λ	Learning rate of the SOM model
$\mathcal{D}_{s_i}$	Euclidean distance between the SOM features at s_i
$\mathcal{D}_{s_i}^{\star}$	Min. euclidean distance between the SOM features at s_i
$R_{s_i}^{\star}$	Best bitrate level at s_i
τ	Segment duration

A. LoL$^+$ Overview

LoL$^+$ is a set of player enhancement modules for LLL streaming. The ultimate goal of LoL$^+$ is to keep the latency low while maintaining high QoE, and also to work well in most common network conditions. It attempts to find the best bitrate at any given time by using a modern unsupervised learning mechanism [23], particularly a self-organizing map (SOM), to make bitrate selection. It also uses a weight assignment algorithm that combines a k-means++ [11] method and greedy search optimization rule to select the best weights for the SOM features. For accurate throughput measurements, LoL$^+$ develops a robust measurement algorithm with high accuracy, and adjusts the playback speed via a hybrid mechanism that relies on both latency and buffer occupancy. Fig. 1 describes LoL$^+$ integration within the dash.js player [1], and Fig. 2 shows the LoL$^+$ modules. As shown in Fig. 1, there are four essential components in total. One of these components was modified to integrate the LoL$^+$ and one was added to log information regarding player's status.

1) Logger: It records the player status at each segment download and keeps track of various QoE metrics.

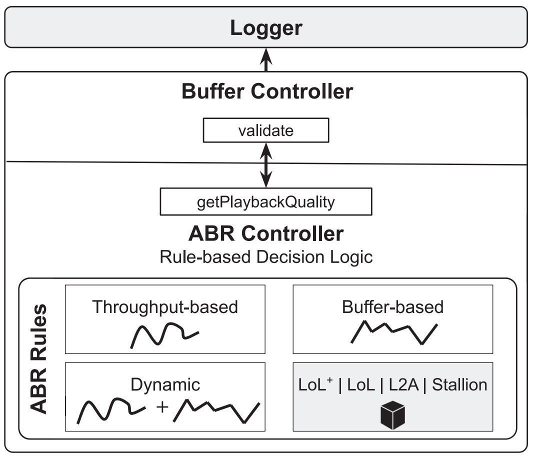


Fig. 1. The LoL$^+$ solution within the dash.js player [1]. The newly added components are highlighted in gray.

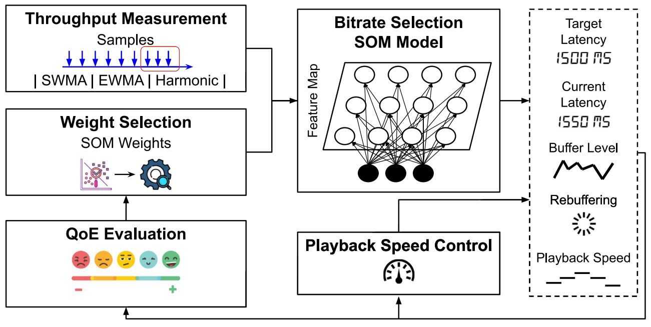


Fig. 2. The LoL$^+$ modules.

2) Buffer Controller: It monitors the level of the playback buffer occupancy to alleviate rebuffering. This is achieved by periodically calling the `validate` function, which checks whether the selected bitrate level may introduce a buffer underflow or overflow. If the buffer occupancy reaches the low or high threshold, this component triggers an event to the ABR controller asking it to select a new bitrate level to maintain the buffer occupancy within a safe region.

3) ABR Controller: It uses `getPlaybackQuality` function to obtain the bitrate selected by the ABR algorithm. It forwards the selected bitrate to the scheduler, which is responsible to issue an HTTP GET request to download the requested segment. It also manages the ABR rules derived from the implemented ABR algorithms.

4) ABR Rules: It implements various ABR algorithms: throughput-based, buffer-based (BOLA [30]) and dynamic. To compare with LoL$^+$, we integrated three additional ABR algorithms: LoL [25], L2A [21] and STALLION [15]. As depicted in Fig. 2, LoL$^+$ implements five modules:

a) Throughput Measurement: It accurately measures the per-chunk throughput (Section III-B).

b) QoE Evaluation: It collects five key metrics (selected bitrate, rebuffering duration, latency, number of bitrate switches and playback speed) and implements a flexible linear-based QoE model (4) that combines these metrics into one QoE score (Section III-B).

c) Weight Selection: It implements the dynamic weight assignment algorithm (Section III-B).

d) Playback Speed Control: It enables the adjustment of the playback speed to avoid rebuffering or deviation from the target latency (Section III-B).

e) Bitrate Selection: It implements the SOM-based ABR algorithm. It takes different inputs from various modules, such as the weight vector of the SOM model, QoE metrics, playback speed as well as throughput measurements and outputs the selected bitrate for the next segment to be requested (Section III-B).

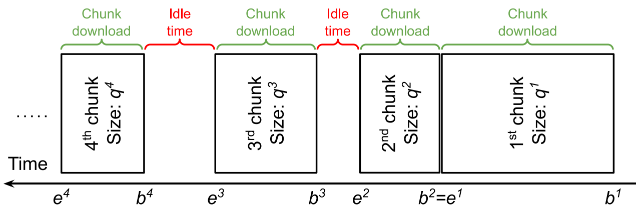


Fig. 3. Throughput measurement in CTE.

B. LoL$^+$ Design

We present the core functional design and implementation details for the LoL$^+$ solution below.

a) Throughput Measurement Module: In LLL streaming with CTE, the available chunks of a segment are sent by the origin server in a burst at full network speed (transmission is network-limited), whereas the chunks that are yet to be prepared will be sent as they become available (transmission will be source-limited), leaving idle times between the chunks of a particular segment [6], [7]. Therefore, the basic equation of throughput calculation (1) will produce a throughput value that is close to the encoding bitrate, which is incorrect and prevents the ABR algorithm from switching to a higher bitrate level. After the segment s_i is fully downloaded, the following formula is used to calculate the throughput (denoted by X_{s_i}):

$$X_{s_i} = \frac{Q_{s_i}}{T_{s_i}}, \text{ where } Q_{s_i} = \sum_{c=1}^{z_{s_i}} q_{s_i^c} \text{ and } T_{s_i} = e_{s_i^z} - b_{s_i^1}, \quad (1)$$

where Q_{s_i} is the segment size, T_{s_i} is the segment download time (*i.e.*, the difference between receiving the first byte of the first chunk until the last byte of the last chunk), $q_{s_i^c}$ is the chunk size, z_{s_i} is the last chunk, $|z_{s_i}|$ total number of chunks of s_i, b and e are the beginning and end times of the chunk download, respectively (see Fig. 3). When calculating T_{s_i}, the idle times are inadvertently included, which results in inaccurate throughput measurements. ACTE [6] was the first attempt to address this issue. Although ACTE generally achieved a good performance, it occasionally introduced overestimations in case of increased inter-chunk idle times.

Learning from the shortcomings of ACTE, we design a new throughput measurement method (Algorithm 1) that can work efficiently under different transmission scenarios (network-limited or source-limited) and deal with different inter-chunk idle times (short or long). At each segment download, the algorithm sequentially runs the following three steps:

1) Chunk Boundary Identification (lines 4–22): The algorithm considers only the chunks where a 'moof' box is presented in a chunk (Flag$_1$ $==1$). It identifies the exact timing for the beginning $b_{s_i^c}$ and end $e_{s_i^c}$ times of chunk c of segment s_i by capturing the 'moof' box in the fragmented MP4 data, $\forall c \in s_i$ where $c = \{1, \ldots, z_{s_i}\}$. A CMAF chunk may be transmitted in multiple HTTP chunks, but this is hidden from the JavaScript APIs used for downloading. An HTTP chunk may also carry multiple CMAF chunks. In either case, as chunks conform to the CMAF restrictions, our algorithm leverages the `Stream Response Body` of the Fetch API[2], which allows tracking the progress of the chunk downloads and parsing them in real time. When a 'moof' box of a chunk is captured, the algorithm stores the time as the beginning time of the chunk download using `performance.now()`. Then, it stores the end time using `performance.now()` of the chunk and its size (in bytes) when the chunk is fully downloaded (Flag$_2$ $==1$). With accurate beginning and end times for each chunk downloaded, our algorithm is able to successfully remove the idle periods from the segment download time.
2) Chunk Filtering (lines 13–16): The algorithm removes the first and the last chunk of a segment to avoid transient outliers.
3) Segment Throughput Calculation and Smoothing (line 23): After downloading all the chunks of a segment, the algorithm computes the segment throughput based on the chunks that passed the filtering process and uses a default Sliding Window Moving Average (SWMA)-based smoothing function as follows:

$$X_{s_i} = \frac{1}{|\tilde{z}_{s_i}|} \sum_{c=1}^{\tilde{z}_{s_i}} x_{s_i^c}, \text{ where } x_{s_i^c} = \frac{q_{s_i^c}}{e_{s_i^c} - b_{s_i^c}}, \qquad (2)$$

and X_{s_i} is the segment throughput, $\tilde{z}_{s_i}$ is the last chunk in segment s_i, $|\tilde{z}_{s_i}|$ is the total number of chunks of segment s_i after the filtering process, and $x_{s_i^c}$ is the chunk throughput. This module also implements two more smoothing functions: Exponentially Weighted Moving Average (EWMA) and Harmonic [19].

b) Weight Selection Module: As shown in previous studies [11], [26], the SOM model used in bitrate selection is sensitive to the initial weight values assigned to various SOM features. Assigning these weights randomly or tuning them manually may reduce the bitrate selection performance. We solve this problem by proposing a two-stage technique.

We formulate the weight selection problem as an *assignment problem* [9] using a convex optimization [8] with linear inequality constraints where the objective function (3) is strictly concave with non-negativity constraints. Specifically, we use the MILP [8] framework to model the objective function. Our formulation is explained as follows.

[2]https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Algorithm 1: Throughput Measurement.

```
function CalculateThroughput$s_i$
  $|\tilde{z}_{s_i}| \leftarrow 0$, Sum $\leftarrow 0$, $X_{s_i} \leftarrow 0$
  for Each segment downloading step $s_i \in S$, $i > 0$ do
    for Each chunk downloading step $c > 0, \forall c \in s_i$ do
      Flag$_1$ $\leftarrow$ ParsePayload('moof', $c$)
      if (Flag$_1$ $== 1$) then        ▷'moof' is present in chunk $c$
        $b_{s_i^c} \leftarrow$ performance.now()
        Flag$_2$ $\leftarrow$ ParsePayloadCompleted('mdat', $c$, end)
        if (Flag$_2$ $== 1$) then        ▷'mdat' end of chunk $c$
          $e_{s_i^c} \leftarrow$ performance.now()
          $q_{s_i^c} \leftarrow$ ChunkSize($c$)
        end if
        if ($c = 1$ || $c = z_{s_i}$) then
          ChunkFilter('noise')
          $x_{s_i^c} \leftarrow 0$
        else
          $x_{s_i^c} \leftarrow$ ChunkThroughput($q_{s_i^c}, e_{s_i^c}, b_{s_i^c}$)        ▷(2)
        end if
        Sum $\leftarrow$ Sum $+ x_{s_i}^c$
        $|\tilde{z}_{s_i}| \leftarrow |\tilde{z}_{s_i}| + 1$
      end if
    end for
    $X_{s_i} \leftarrow$ SegThroughput($|\tilde{z}_{s_i}|$, Sum, SWMA)        ▷(2)
    Return($X_{s_i}$)
  end for
end Function
```

First, the SOM model consists of a set of neurons (denoted by M) with a fixed total number j, where each neuron (denoted by $m_{s_i}^{\hat{j}} \in M_{s_i}$) takes the SOM features as an input. At each s_i, we define the set M as follows:

$$M_{s_i} = \left\{m_{s_i}^1, \ldots, m_{s_i}^{\hat{j}}, \ldots, m_{s_i}^{j}\right\}, \forall \hat{j} \in [1, \ldots, j], \forall s_i \in S$$

For each neuron $m_{s_i}^{\hat{j}} \in M_{s_i}$, the objective is to determine the best weight vector (denoted by $\mathbf{w}_{s_i}^{\star} \in W_{s_i}$) for the set of SOM features, namely, throughput (X), latency (L), rebuffering (E) and number of bitrate switches (H) that maximize the player step utility (denoted by u_{s_i}), and thus, the player total utility (denoted by $U_S^K = \sum_{i=0}^{K} u_{s_i}$). We define the utility function as the QoE function (4) and assume that our utility function is a strictly increasing concave function following a network utility maximization framework [10]. The utility value is computed using the QoE evaluation module shown in Fig. 2. Then, each possible weight vector (denoted $w_{s_i}^{\hat{v}}$) at s_i is defined as:

$$w_{s_i}^{\hat{v}} = \left\{w_{s_i}^{X_{\hat{v}}}, w_{s_i}^{L_{\hat{v}}}, w_{s_i}^{E_{\hat{v}}}, w_{s_i}^{H_{\hat{v}}}\right\}, \forall s_i \in S, \quad \forall w_{s_i}^{\hat{v}} \in W_{s_i},$$

where $i \in [1, \ldots, K]$ is the number of a segment s, $S = \{s_1, \ldots, s_i, \ldots, s_K\}$ is the list of the downloaded segments, K ($= |S|$) is the cumulative number of segments, $w_{s_i}^{n}$ is the weight

Algorithm 2: Heuristic-based Dynamic Weight Assignment.

```
 1: function WeightVectorSelection$s_i$
 2:   $\mathbf{w}^{\star}_{s_{i+1}} \leftarrow \{\}$, $W_{s_i} \leftarrow \{\}$, $u^{\star}_{s_i} \leftarrow 0$, $V \leftarrow \{0, 0.2, 0.4, 0.6, 0.8, 1\}$
 3:   $W_{s_i} \leftarrow$ WeightVectorGenerator($V$)
 4:   for Each segment downloading step $s_i \in S$, $i > 0$ do
 5:     $M_{s_i} \leftarrow$ GetNeurons(SOM)
 6:     if Initial Stage ($s_1$) then
 7:       $\mathbf{w}^{\star}_{s_1} \leftarrow$ k-means++($\{w^X_{s_1}, w^L_{s_1}, w^E_{s_1}, w^P_{s_1}, w^H_{s_1}\}$)
 8:     else
 9:       for Each SOM Neuron $m^{\hat{j}}_{s_i} \in M_{s_i}$, $1 \le \hat{j} \le j$ do
10:         for Each weight vector $w^{\hat{v}}_{s_i} \in W_{s_i}$, $1 \le \hat{v} \le v$ do
11: $u^{(m^{\hat{j}}_{s_i}, w^{\hat{v}}_{s_i})}_{s_i} \leftarrow$ ComputeQoE($R_{s_i}, E_{s_i}, L_{s_i}, P_{s_i}$)
12: $\mathcal{M}^{(M,W)}_{s_i}$ = Push($u^{(m^{\hat{j}}_{s_i}, w^{\hat{v}}_{s_i})}_{s_i}$)
13: if A1, A2, A3 in (3) are satisfied then
14: if $u^{(m^{\hat{j}}_{s_i}, z^{\hat{v}}_{s_i})}_{s_i} > u^{\star}_{s_i}$ then
15: $u^{\star}_{s_i} \leftarrow u^{(m^{\hat{j}}_{s_i}, w^{\hat{v}}_{s_i})}_{s_i}$
16: $\mathbf{w}^{\star}_{s_{i+1}} \leftarrow w^{\hat{v}}_{s_i}$
17: else
18: GoTo(10)
19: end if
20: else
21: if $\hat{v} = v$ then
22: $\mathbf{w}^{\star}_{s_{i+1}} \leftarrow \mathbf{w}^{\star}_{s_i}$
23: end if
24: end if
25:         end for
26:       end for
27:     end if
28:     Return($\mathbf{w}^{\star}_{s_{i+1}}$)
29:   end for
30: end Function
```

of a feature n, $W_{s_i} = \{w^1_{s_i}, \ldots, w^{\hat{v}}_{s_i}, \ldots, w^v_{s_i}\}$ and v are the set and total number of possible weight vectors at s_i, respectively. Further, each segment s_i has a fixed duration (denoted by τ) and comprises a set of z_{s_i} chunks such that $\forall c \in s_i$, and $c = \{c^1, \ldots, c^z\}$. To simplify the problem formulation and represent the SOM features in the same space, we use normalized features in the range between 0 and 1. Therefore, the weight of each feature should be also in the same range, *i.e.*, each weight takes one value from the set $V = \{0.2, 0.4, 0.6, 0.8, 1\}$. For example, $w^1_{s_i} = \{0.2, 0.2, 0.2, 0.2\}$ represents the first weight vector in W_{s_i}. To reduce the complexity in solving the optimization problem, we consider these possible values in V.

Second, the level of satisfaction for a given neuron depends on the weight vectors that are able to maximize the utility in the next segment to be downloaded (s_{i+1}). Let $u^{(m^{\hat{j}}_{s_i}, w^{\hat{v}}_{s_i})}_{s_i}$ denote the achieved utility of assigning a weight vector $w^{\hat{v}}_{s_i}$ ($1 \le \hat{v} \le v$) to

Algorithm 3: Generate Possible Weight Vectors.

```
 1: function WeightVectorGenerator$V$
 2:   $W_{s_i} \leftarrow \{\}$, $w_{s_i} \leftarrow \{\}$, $w^{\hat{v}}_{s_i} \leftarrow \{\}$
 3:   for Each possible weight vector $w^{\hat{v}}_{s_i}$ permutation from $V$ do
 4:     $w^{\hat{v}}_{s_i} \leftarrow$ Generate($\{w^{X_{\hat{v}}}_{s_i}, w^{L_{\hat{v}}}_{s_i}, w^{E_{\hat{v}}}_{s_i}, w^{P_{\hat{v}}}_{s_i}, w^{H_{\hat{v}}}_{s_i}\}$, $V$)
 5:     if $w^{\hat{v}}_{s_i} \ne w_{s_i}$ then
 6:       $W_{s_i}$ = Push($w^{\hat{v}}_{s_i}$)
 7:       $w_{s_i} \leftarrow w^{\hat{v}}_{s_i}$
 8:     end if
 9:   end for
10:   Return($W_{s_i}$)
11: end Function
```

a neuron $m^{\hat{j}}_{s_i}$ ($1 \le \hat{j} \le j$) of the SOM model. Hence, we define the possible achieved utility matrix (denoted by $\mathcal{M}^{(M,W)}_{s_i}$) for assigning the set of possible weight vectors to the set of existing neurons as follows:

$$\mathcal{M}^{(M,W)}_{s_i} = \begin{Bmatrix} u^{(m^1_{s_i}, w^1_{s_i})}_{s_i} & u^{(m^1_{s_i}, w^2_{s_i})}_{s_i} & \cdots & u^{(m^1_{s_i}, w^v_{s_i})}_{s_i} \\ u^{(m^2_{s_i}, w^1_{s_i})}_{s_i} & u^{(m^2_{s_i}, w^2_{s_i})}_{s_i} & \cdots & u^{(m^2_{s_i}, w^v_{s_i})}_{s_i} \\ \vdots & \vdots & \ddots & \vdots \\ u^{(m^j_{s_i}, w^1_{s_i})}_{s_i} & u^{(m^j_{s_i}, w^2_{s_i})}_{s_i} & \cdots & u^{(m^j_{s_i}, w^v_{s_i})}_{s_i} \end{Bmatrix}$$

Third, we express the objective ($\mathcal{F}$) as an MILP framework:

$$\mathbf{Find}(\mathbf{w}^{\star}_{s_{i+1}}) \text{ such that } U(\mathbf{w}^{\star}_{s_{i+1}}) = \arg\max \sum_{i=0}^{K} u_{s_i}(\mathbf{w}^{\star}_{s_{i+1}})$$
$$s.t. \begin{cases} \mathbf{A1}: L_{s_{i+1}} \le L_{target} + |L_{s_i} - L_{s_{i-1}}| \\ \mathbf{A2}: B_{s_{i+1}} \ge B^{low} \\ \mathbf{A3}: V^{min} \le w^{\hat{v}}_{s_{i+1}}(.) \le V^{max}, \quad \forall w^{\hat{v}}_{s_{i+1}} \in W_{s_{i+1}} \end{cases} \tag{3}$$

where $\mathbf{w}^{\star}_{s_{i+1}}$ is the best weight vector that should be selected and assigned for a neuron for the next segment to be downloaded (s_{i+1}). The aims are defined as follows: **A1** ensures that the latency is under control (*i.e.*, close to the target latency). **A2** ensures that the playback buffer occupancy stays above the safe threshold. **A3** ensures that values of the weights assigned for features in each weight vector are between the minimum and maximum bounds.

Fourth, we propose a two-stage heuristic-based technique that is based on a k-means++ approach in the first stage and a greedy-search approach in the second stage to solve the MILP optimization problem (3). Our dynamic weight assignment (Algorithm 2) achieves a good performance in polynomial time under real conditions as empirically shown in Section IV-C1. Its complexity depends on the number of neurons j and possible weight vectors v for every segment downloading step. Although an exhaustive approach like brute-force search finds the optimal solution, its complexity grows dramatically with the increase in the numbers of j and v. In our heuristic-based technique, most of the computation happens a total of K times during the entire live session. According to the objective function, the weight vector assignment for each neuron has the time complexity $\mathcal{O}(\max(j+v, j.v+j)) = \mathcal{O}(j.v+j)$ in the worst case. Therefore, the overall worst case complexity for the live streaming session is $\mathcal{O}(k.(j.v+j))$, which is significantly less than the upper bound of the exhaustive brute-force search ($\mathcal{O}(k.j^v)$). It is worth mentioning that the weight selection module depends on the adopted QoE function, which might impact the selection process if it is not appropriately picked.

c) QoE Evaluation Module: In recent years, there have been many proposed QoE models in the literature [28]. However, most of the existing models do not capture the latency and playback speed's impact on the QoE, limiting their effectiveness in LLL scenarios. To that effect, we need a flexible QoE model that considers the most important metrics for LLL streaming. Hence, we used our prior QoE model [25] that conforms with requirements of LLL streaming. At each segment s_i download step, the QoE model considers five essential metrics: bitrate selected reward (R_{s_i}), bitrate switches penalty ($H_{s_i} = |R_{s_{i+1}} - R_{s_i}|$), rebuffering time penalty (E_{s_i}), latency penalty (L_{s_i}) and playback speed penalty (P_{s_i}). The QoE model is given as:

$$\text{QoE}_S^K = \sum_{i=1}^{K} (\alpha R_{s_i} - \beta E_{s_i} - \gamma L_{s_i} - \sigma |1 - P_{s_i}|) - \sum_{i=1}^{K-1} \mu H_{s_i}, \tag{4}$$

where the notations are given in Table I. Based on the subjective tests in [37], [38] and our measurements, we fixed the weights of each metric in (4) as follows: α = segment duration; β = maximum encoding bitrate [in Kbps]; $\gamma = \{$if $L \leq 1.6$ seconds, then $= 0.05 \times$ minimum encoding bitrate, otherwise $= 0.1 \times$ maximum encoding bitrate$\}$; σ = minimum encoding bitrate; and $\mu = 1$. The playback speed is typically [1]: *normal:* $1\times$, *faster: e.g.*, $1.3\times$ or *slower: e.g.*, $0.7\times$. Thus, if the playback speed is normal, there is no playback speed penalty.

d) Playback Speed Control Module: The main goal of this module is to determine a playback rendering speed that ensures low latency while taking a calculated risk of rebuffering. It is an event-based module that is triggered as per the cases defined in Algorithm 4 and operates independently from the bitrate selection module. Algorithm 4 is a hybrid playback speed control technique that considers two variables: (i) the difference between the current and target latency, and (ii) the difference between the current playback buffer level and the safe threshold (B^{low}). Algorithm 4 reacts quickly to changes in these variables making the playback more robust. Note that all the media is still played out (no skips) despite the playback rate changes.

Case 1: The current buffer level is below the safe threshold. In this case, LoL$^+$ selects a slower playback speed.

Case 2: The current buffer level is equal to or above the safe threshold and further:

- 2a: The current latency is close enough to the target latency ($\epsilon = \pm 2\%$), then LoL$^+$ selects the normal playback speed.
- 2b: The current latency is lower than the target latency, then LoL$^+$ selects a slower playback speed.
- 2c: The current latency is higher than the target latency, then LoL$^+$ selects a faster playback speed.

Algorithm 4: Playback Speed Control (Pseudocode).

```
1:  function PlaybackSpeedSelection$L_{s_i}, L_{target}, B_{s_i}, B^{low}$
2:    if ($B_{s_i} < B^{low}$) then
3:      $B_{delta} \leftarrow |B^{low} - B_{s_i}|$
4:      $P_{s_i} \leftarrow$ CalculateSpeed($B_{delta}$, Slower, Limit=0.7)
5:    end if
6:    if ($B_{s_i} \geq B^{low}$) then
7:      if ($L_{s_i} \approx L_{target}$) then
8:        $P_{s_i} \leftarrow$ CalculateSpeed(Normal, Speed=1)
9:      end if
10:     if ($L_{s_i} < L_{target}$) then
11:       $L_{delta} \leftarrow |L_{target} - L_{s_i}|$
12:       $P_{s_i} \leftarrow$ CalculateSpeed($L_{delta}$, Slower, Limit=0.7)
13:     end if
14:     if ($L_{s_i} > L_{target}$) then
15:       $L_{delta} \leftarrow |L_{target} - L_{s_i}|$
16:       $P_{s_i} \leftarrow$ CalculateSpeed($L_{delta}$, Faster, Limit=1.3)
17:     end if
18:   end if
19:   Return($P_{s_i}$)
20: end Function
```

LoL$^+$ can be configured for the range of the playback speed (*e.g.*, $0.7 - 1.3\times$) [1], [27], buffer level safe threshold and target latency. Note that the playback speed is calculated based on how large the deviation is between the current and target latency, and the current buffer level and safe threshold. The larger the deviation, the faster/slower the playback speed.

e) Bitrate Selection Module: This module is invoked at the end of each segment download and its goal is to select the best bitrate level for the next segment that maximizes the QoE while maintaining the latency close to the target. This module employs a learning-based rule (Algorithm 5) that is based on SOM [24], which was first proposed by Kohonen [24] and is now one of the widely used techniques for unsupervised classification problems. A SOM is a type of artificial neural network (ANN) that is trained using unsupervised learning and consists of a single-layer linear 2D grid of neurons. All the neurons in the grid are connected directly to the input vector, but not to one another. Thus, a neuron does not know the values of its neighbours, and only updates the weight of its connections as a function of the given inputs. The grid is referred to as a map that organizes itself at each iteration as a function of the input data with the application of competitive learning as opposed to error-correction learning in other ANNs. There are two main reasons for using a SOM for bitrate selection in LLL streaming: (i) a SOM uses online learning without requiring supervised model training, allows bitrate decision making in real time and evolves its model quickly over time, and (ii) the data points (variability in the network conditions) are unknown a priori.

For each representation in the manifest, a corresponding SOM neuron is created and initialized with the value of the bitrate level. The algorithm evaluates the current state and considers the set of data points to be classified. At each segment download, it gathers the current state and classifies it to one of the neurons to find the best matching unit (BMU) in the SOM model.

Our SOM model considers a quadruplet of features for the data points that combines player context (*i.e.*, multiple individual QoE metrics) and network context (*i.e.*, measured throughput). At the beginning of a live streaming session, the value of the throughput feature is initialized with the minimum encoding bitrate level. For more accurate bitrate selection and model simplification, we normalize all the data points to between zero and one. Every state (white circles in Fig. 2) comprises a quadruplet set of features that are calculated after each segment download. To enhance the SOM model further, we also include the latency (set to the target value), rebuffering duration (set to zero) and number of bitrate switches (set to zero) as a target. Then, we use a distance function $\mathcal{D}()$ to find the best matching (5) that represents the euclidean distance of the four features used in each neuron. The distance function $\mathcal{D}()$ at s_i is defined as follows:

$$\mathcal{D}_{s_i}(a,b) = \sqrt{\sum_{n=1}^{4} w_{s_i}^{n} \times (a[n]-b[n])^2}, \qquad (5)$$

where n is a specific feature (X, L, E, or H) that is associated with a weight value ($w_{s_i}^{n}$) selected by Algorithm 2.

Usually, after finding the best BMU, the corresponding neuron and its neighbours should be updated. In our context, the update function is slightly different since we only have two known SOM neurons. The first one is the current neuron (that represents the current selected bitrate level) and the second neuron is the next best one to transit to. That is, we update the current SOM neuron with the newly reported values of the features (*i.e.*, the player reports its status at every segment download) and then we update its neighbours as well using the same values. Next, the best BMU neuron is calculated using (5) with latency, rebuffering duration and number of bitrate switches all equal to their target values. The best neuron that achieves the target values (the closest one to the BMU neuron) wins. After finding the winning neuron, it becomes the current one and will be updated with the reported values of the features, and so on. The process is repeated for each segment download until the streaming session ends.

Algorithm 5 shows the pseudocode of the bitrate selection, which performs two main updates. The first update moves the current active neuron towards the freshly measured features. Later, the BMU is calculated as defined in the algorithm. Then, the second update moves the BMU towards an ideal (but in practice unreachable) point that is predefined. For our runs, we used ideal latency L_{ideal} (zero seconds), ideal rebuffering E_{ideal} (zero seconds) and ideal number of bitrate switches H_{ideal} (zero switches). So, there is a balance between those two updates where one of them moves away from the ideal point and the other is trying to move back to the ideal point. At each iteration, the algorithm selects the one closer to the ideal point and updates all neighbours accordingly. The neighbourhood function used in the algorithm is a Gaussian distribution function. The learning rate (denoted by λ) is fixed to 0.01 as in the original paper [24].

Algorithm 5: SOM Bitrate Selection (Pseudocode).

```
1:  function NextMaxRate()
2:    $\mathcal{D}^{\star}_{s_i} \leftarrow 0$, BMU $\leftarrow \emptyset$, $R^{\star}_{s_{i+1}} \leftarrow 0$, $\lambda \leftarrow 0.01$
3:    for Each segment downloading step $s_i \in S, i > 0$ do
4:      $X_{s_i} \leftarrow$ CalculateThroughput($s_i$)
5:      $B_{s_i} \leftarrow$ GetBufferLevel($s_i$)
6:      $L_{s_i} \leftarrow$ GetLatency($s_i$)
7:      $E_{s_i} \leftarrow$ GetRebuffering($s_i$)
8:      $H_{s_i} \leftarrow$ GetSwitches($s_i$)
9:      $w_{s_i} \leftarrow$ WeightVectorSelection($s_i$)
10:     Normalize($\{X_{s_i}, L_{s_i}, E_{s_i}, H_{s_i}\}$, [0,1])
11:     Update($m^{\hat{j}}_{s_i}$, $\{X_{s_i}, L_{s_i}, E_{s_i}, H_{s_i}\}$)
12:     for all neurons $\forall\, m^{\hat{j}}_{s_i} \in M_{s_i}, 1 \le \hat{j} \le j$ do
13:     $R_{s_i} \leftarrow R^{\star}_{s_i}$
14:     $\mathcal{D}_{s_i} \leftarrow$ GetDistance($w_{s_i} \times \{X_{s_i}, L_{s_i}, E_{s_i}, H_{ideal}\}$)
15:     if ($\mathcal{D}_{s_i} < \mathcal{D}^{\star}_{s_i}$) then
16:       if ($L_{s_i} \le L_{target}$) && ($B_{s_i} \ge B^{low}$) then
17:         $\mathcal{D}^{\star}_{s_i} \leftarrow \mathcal{D}_{s_i}$
18:         BMU $\leftarrow m^{\hat{j}}_{s_i}$
19:         $R^{\star}_{s_{i+1}} \leftarrow$ BMU.bitrate
20:         $R_{s_{i+1}} \leftarrow R^{\star}_{s_{i+1}}$
21:       else
22:         BMU $\leftarrow$ SelectNeuron($X_{s_i}$)
23:         $R^{\star}_{s_{i+1}} \leftarrow$ BMU.bitrate
24:         $R_{s_{i+1}} \leftarrow R^{\star}_{s_{i+1}}$
25:       end if
26:     end if
27:   end for
28:   Update($m^{\hat{j}}_{s_i}$, $\{X_{s_i}, L_{ideal}, E_{ideal}, H_{s_i}\}$)
29:   Return($R_{s_{i+1}}$)
30:   NextMaxRate()
31: end for
32: end Function
```

IV. Experimental Evaluation

In this section, we present the results from the trace-driven experimental evaluation of LoL$^+$. We first describe the implementation of LoL$^+$ in dash.js (v3.0.1) [1]. We then present the methodology, results and detailed analysis. Our goal is to answer the following questions:

- Can LoL$^+$ deliver high quality video with no rebuffering at a low latency? How does it compare against the other ABR algorithms (LoL [25], L2A [21], STALLION [15] and Dynamic [1])?
- How effective are the LoL$^+$ modules in different scenarios?
- How sensitive is LoL$^+$ to the QoE model and player parameters (*e.g.*, target latency)?

TABLE II
LoL$^+$ Implementation in the Dash.js Player

Module	State	Code Lines	Changed
LoLp_BitrateSelection.js	New	472	-
LoLp_QoEEvaluation.js	New	179	-
LoLp_WeightSelection.js	New	197	-
FetchLoader.js	Modified	258	36% (92 lines)
BoxParser.js	Modified	275	29% (80 lines)
PlaybackController.js	Modified	1077	17% (183 lines)

- What is the impact of the weight selection of the learning model on the selected bitrates, latency and rebuffering?

A. Implementation

LoL$^+$ has been implemented on top of the JavaScript-based dash.js player (v3.0.1) [1] and our source code is available online [2]. LoL$^+$ consists of $\sim$1500 lines of new or modified code and the implementation details for each module are highlighted in Table II. Our throughput measurement module is included in `FetchLoader.js` and `BoxParser.js`. Further, our playback speed control module is included in `Playback-Controller.js`. At the server side, we used FFmpeg for encoding and packaging, and a Python-based framework (347 lines of code) for the chunk ingest and origin. We note that LoL$^+$ has been added as one of the ABR rules for LLL streaming in the official dash.js (v3.2.0) player.

B. Methodology and Evaluation Setup

1) Video and Encoding Parameters: Consistent with the requirements of the Twitch grand challenge [33], we used the animation video *Big Buck Bunny*[3] with a segment duration (τ) of 500 and a chunk duration of 33 milliseconds (equal to one frame duration at 30 f/s). We encoded the video using the H.264 codec of FFmpeg into three representations {360p@200 Kbps, 480p@600 Kbps, 720p@1000 Kbps}. The live session duration and number of segments (K) were set according to the duration of the bandwidth profile.

2) Bandwidth Profiles: To emulate different real-world network conditions, we used a set of 11 bandwidth profiles to throttle the bandwidth between the server and player. These profiles were extracted from datasets and can be found in [2].

- We used five bandwidth profiles that were provided by Twitch [33], namely: CASCADE, INTRA-CASCADE, SPIKE, SLOW-JITTERS and FAST-JITTERS. The average bitrate (Mbps) and duration (seconds) of each profile is {(0.8150), (0.66 135), (0.77,30), (0.85,30), (1.16,11.6)}, respectively.
- We randomly selected two types of live channels from Twitch based on popularity: {low and medium}. Then, we measured the bandwidth between the Twitch player and server every five seconds. We collected two bandwidth profiles, namely: TW-LOW and TW-MED. The duration of these profiles is 360 and 190 seconds with an average bitrate of 0.67 Mbps and 1.31 Mbps, respectively.
- We extracted four bandwidth profiles with different mobility types from the Belgium LTE dataset [34]. Each bandwidth value in the profile was logged every second and the profiles captured were: BICYCLE, TRAIN, TRAIN-MODIFIED and TRAM. The duration of each profile is 600 seconds and the average bitrates are 3.9, 2.96, 2.97 and 2.72 Mbps, respectively. We note that the TRAIN profile has many values lower than the lowest encoding bitrate ($<$ 200 Kbps) and in the modified profile these values were set to the lowest encoding bitrate.

3) ABR Algorithms and Metrics: We compared LoL$^+$ against four LLL ABR algorithms, including LoL [25], L2A [21], STALLION (STA) [15] and Dynamic (DYN) of dash.js [1] (with low-latency mode enabled). To evaluate the performance of LoL$^+$, we used the following metrics: (i) latency, (ii) QoE and its metrics, and (iii) mean absolute error (MAE), which measures the absolute difference between the actual and measured throughput values.

4) Setup: To build an end-to-end live streaming system, we used a MacBook Pro (macOS Catalina, 6-Core Intel Core i7 processor, 16 GB memory and Intel UHD Graphics 630) with two virtual machines (VMs). The first VM executed the dash.js player in the Google Chrome browser (v86). The second VM was used to run the FFmpeg encoder with CMAF packaging enabled to feed into the origin server. To emulate the network, we used Chrome-devtools at the player to throttle the bandwidth between the origin server and player according to the described bandwidth profiles. We set the minimum IDR-frame interval to 15 frames at the encoder (*i.e.*, an I-frame at the beginning of each segment) and B^{low} to 0.5 seconds.

C. Results and Analysis

We executed five runs for each bandwidth profile and took the averages in each of the following experiments:

1) Impact of the Weight Selection: To investigate the impact of the weight selection on the bitrate selection module, we implemented two more techniques (MAN: manual selection, RAN: random selection) to compare with the dynamic weight assignment (DWA).

- MAN: We manually tune the weights for the SOM features by experimenting with different possibilities and selecting the best performing set.
- RAN: The well-known Xavier activation function [14] is used to set the weights to the values chosen from a random uniform distribution that is bounded between $\pm\sqrt{6}/\sqrt{n_j + n_{j+1}}$, where n_j is the number of incoming network connections to the layer and n_{j+1} is the number of outgoing network connections from that layer.
- DWA: This uses Algorithm 2.

The results of this experiment for different bandwidth profiles are given in Table III. In contrast to MAN, which selects the weights only once at the session initialization, DWA enables LoL$^+$ to perform better in most profiles. However, in a few cases, LoL$^+$ performs similar to MAN and RAN. This outcome is achieved because of Algorithm 2, which strives to select the best performing weights set at each segment download. DWA aims to

[3]https://peach.blender.org/download/

TABLE III
IMPACT OF THE WEIGHT SELECTION ON THE LEARNING MODEL (MAN: MANUAL, RAN: RANDOM XAVIER [14], DWA: DYNAMIC WEIGHT ASSIGNMENT (ALGORITHM 2))

Weights Selection	Avg. Bitrate (Kbps)	Avg. Rebuffering (s)	Avg. Latency (s)	Avg. Switches (#)
CASCADE (150 seconds)				
MAN	494.03	0.05	1.52	49.0
RAN	444.09	0.10	1.52	55.0
DWA	469.91	0.15	1.52	55.0
INTRA-CASCADE (135 seconds)				
MAN	267.14	0.56	1.53	33.2
RAN	280.88	0.56	1.53	44.2
DWA	281.98	0.35	1.53	44.4
SPIKE (30 seconds)				
MAN	521.96	1.76	1.69	5.00
RAN	526.72	1.56	1.65	5.60
DWA	555.02	1.30	1.61	9.00
SLOW-JITTERS (30 seconds)				
MAN	368.47	1.00	1.59	12.0
RAN	346.60	0.60	1.54	15.8
DWA	354.04	0.61	1.54	14.2
BICYCLE (600 seconds)				
MAN	884.25	0.5	1.50	41.6
RAN	891.38	1.15	1.50	147.6
DWA	797.59	0.55	1.50	28.0

TABLE IV
IMPACT OF THE QOE FUNCTION ON THE LEARNING-BASED SOM MODEL. $\mathcal{A}$: SOM USES INDIVIDUAL QOE METRICS, $\mathcal{B}$: SOM USES QOE (4) AS A FEATURE. DWA IS USED FOR WEIGHT SELECTION

Learning Model	Avg. Bitrate (Kbps)	Avg. Rebuffering (s)	Avg. Latency (s)	Avg. Switches (#)
CASCADE (150 seconds)				
$\mathcal{A}$	435.29	0.31	1.52	36.20
$\mathcal{B}$	455.44	0.25	1.52	53.40
INTRA-CASCADE (135 seconds)				
$\mathcal{A}$	279.96	0.66	1.53	41.80
$\mathcal{B}$	304.81	0.55	1.53	44.40
SPIKE (30 seconds)				
$\mathcal{A}$	530.47	1.77	1.62	7.00
$\mathcal{B}$	501.34	2.06	1.71	7.60
SLOW-JITTERS (30 seconds)				
$\mathcal{A}$	326.74	1.51	1.61	10.6
$\mathcal{B}$	362.44	1.00	1.57	12.0
FAST-JITTERS (11.6 seconds)				
$\mathcal{A}$	757.85	0	1.45	5.40
$\mathcal{B}$	668.88	0	1.47	5.00

TABLE V
IMPACT OF THE TARGET LATENCY

L_{target} (s)	Avg. Bitrate (Kbps)	Avg. Rebuffering (s)	Avg. Latency (s)	Avg. Switches (#)
CASCADE (150 seconds)				
1.5	569.91	0.15	1.52	55.0
2.0	664.53	0.10	2.00	5.80
3.0	676.10	0.05	2.96	5.00
INTRA-CASCADE (135 seconds)				
1.5	381.98	0.35	1.53	44.40
2.0	404.31	0.00	1.99	6.20
3.0	390.12	0.00	2.95	4.00
SPIKE (30 seconds)				
1.5	555.02	1.30	1.61	9.00
2.0	607.30	1.16	2.07	3.20
3.0	638.42	0.35	3.01	4.00
SLOW-JITTERS (30 seconds)				
1.5	554.04	0.61	1.54	14.2
2.0	549.95	0.40	2.02	12.0
3.0	571.83	0.00	2.95	11.0

balance different QoE metrics and its performance will converge to near-optimal, especially, in long sessions (>200 seconds). For instance, LoL$^+$ with DWA in the BICYCLE profile (duration of 600 seconds) performs a suitable bitrate selection with small rebuffering duration and low number of bitrate switches. Conversely, MAN and RAN suffer from instability with frequent bitrate switches.

2) Impact of the QoE Function on the Learning Model: To investigate the sensitivity of LoL$^+$ regarding the QoE function (4), we implemented two learning-based SOM models to make bitrate selections. The first model (denoted by $\mathcal{A}$) is decoupled from the QoE function and uses the individual QoE metrics separately as features, whereas the second model (denoted by $\mathcal{B}$) largely depends on the aggregate QoE function in (4) and uses its value as a feature in addition to the throughput. As shown in Table IV, LoL$^+$ using $\mathcal{A}$ achieves comparable results with a slightly higher average bitrate and fewer switches than $\mathcal{B}$. With $\mathcal{B}$, the bitrate selection module may pick a miscalculated bitrate when the QoE model (4) is replaced with another function that leads to a low QoE. This problem is exacerbated considering the diversity of real-world network environments, where it is not possible to have a QoE function that can perfectly fit all kinds of network conditions. However, $\mathcal{A}$ optimizes each QoE metric individually and this allows LoL$^+$ to work efficiently.

3) Impact of the Target Latency: To investigate the impact of setting the target latency on LoL$^+$, we tested different values of $L_{target} = \{1.5, 2.0, 3.0\}$ seconds as highlighted in Table V. We observe that LoL$^+$ performs adequately when the target latency is 1.5 seconds. As the target latency is further relaxed, LoL$^+$ experiences shorter rebuffering and fewer bitrate switches, and achieves higher bitrate as it is given more space in the playback buffer to absorb bandwidth drops.

4) Impact of the Playback Control Module: We compared our hybrid playback control module against the solely latency-based and buffer-based counterparts:

- LAT: Given a user-specified playback speed range (*e.g.*, $0.7\times$ to $1.3\times$ in our case), user-specified target latency (*e.g.*, $L_{target} = 1.5$ seconds in our case) and current latency, this approach calculates a playback speed proportionate to the difference between the current and target latency. If the current latency is much higher (lower) than the target latency, the playback speed would be set closer to the upper (lower) bound specified. Note that this implementation is the default playback speed control mechanism in dash.js.
- BUF: The buffer-based approach works equivalent to the latency-based one above, except we replace the target and current latency with the target and current buffer level, respectively.
- HYB: The LoL$^+$ hybrid playback speed control implementation is given in Algorithm 4, which combines both the latency and buffer-based approaches. HYB first checks if the buffer level is critically low and if so, it adopts the buffer-based approach. Otherwise, it uses the latency-based approach. This is to ensure that the buffer is healthy before the playback speed is adjusted based on latency as the converse may cause more rebufferings, which would increase the latency further.

Note that playback speed here is defined and recorded as per the value provided by dash.js (using `player.getPlaybackRate()`). It is a configuration value provided to the player and does not consider the case of rebuffering. For example, if the playback speed is currently configured at $1.1\times$ (using `player.setPlaybackRate()`),

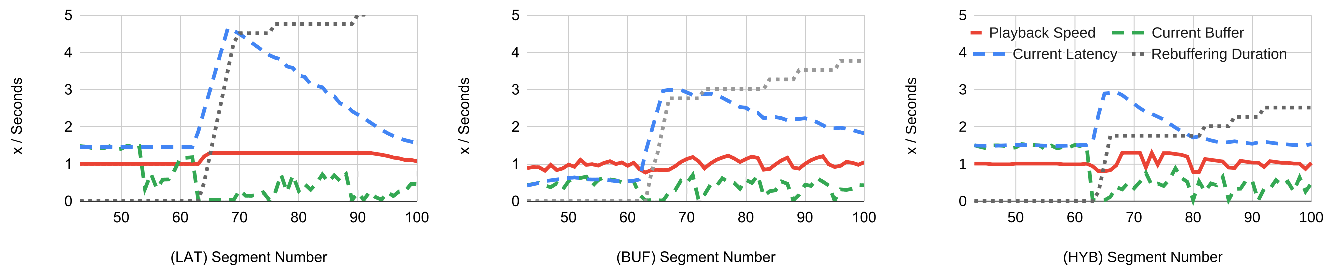


Fig. 4. Sample runs for latency-based, buffer-based and hybrid playback speed control (SLOW-JITTERS).

TABLE VI
IMPACT OF THE PLAYBACK SPEED CONTROL MODULE

Playback Module	Avg. Bitrate (Kbps)	Avg. Rebuf. (s)	Avg. Latency (s)	Avg. Switches (#)	Avg. Playback Speed
CASCADE (150 seconds)					
LAT	528.78	0.20	1.47	62.0	1.00
BUF	211.44	1.00	0.85	3.60	0.93
HYB	469.91	0.15	1.52	55.0	0.98
INTRA-CASCADE (135 seconds)					
LAT	288.86	0.81	1.51	42.8	1.00
BUF	208.95	1.86	0.91	3.20	0.93
HYB	281.98	0.35	1.53	44.4	0.98
SPIKE (30 seconds)					
LAT	529.92	2.26	1.60	6.00	1.04
BUF	276.17	1.01	1.00	4.40	0.94
HYB	555.02	1.30	1.61	9.00	1.00
SLOW-JITTERS (30 seconds)					
LAT	370.31	1.66	1.61	11.0	1.04
BUF	232.77	1.15	0.94	2.00	0.95
HYB	354.04	0.61	1.54	14.2	0.99

then the player would attempt to play the video at $1.1\times$ and even if the buffer is depleted and rebuffering occurs, this playback speed value would not change.

The results with these different playback controls for different bandwidth profiles are given in Table VI. In contrast to BUF (average playback speed of $0.94\times$), LAT and HYB generally achieve an average playback speed that is close to $1\times$ ($0.98\times$ to $1.04\times$), which is a good sign that they do not adapt the playback speed too aggressively, and hence, the speed changes are likely imperceptible to viewers. Also, HYB generally achieves the lowest average rebuffering and an average latency similar to LAT. The reason for this is that HYB considers the current buffer level and latency, allowing it to react quickly and in anticipation of any change in player status and network conditions. Because of the low average bitrate, BUF suffers from low video quality.

To better observe these behavioral differences, Fig. 4 provides three graphs that portray three sample runs, with each using a different playback control module (LAT, BUF and HYB) on profile SLOW-JITTERS. In the first graph (LAT), the playback speed closely follows the current latency. For example, as the throughput drops and latency increases sharply at segment 62, the playback speed also increases quickly to the maximum bound of $1.3\times$. However, this depletes the buffer and causes a sharp increase in rebuffering duration from zero to about 4.5 seconds. In the second graph (BUF), the playback speed closely follows the current buffer level instead. This maintains the buffer at a healthier level, which allows the rebuffering duration to plateau at an improved 2.9 seconds (compared to 4.5 seconds in LAT). Consequently, the latency also peaks at an improved 3 seconds (instead of 4.6 seconds in LAT), even though BUF does not consider the latency. This is because, in LAT, the high playback speed and the consequent buffer starvation at segment 62 further causes latency to increase, which is undesirable and fortunately avoided in BUF. Finally, in the third graph (HYB), the playback speed generally follows the current buffer level to ensure the buffer is kept healthy (similar to BUF). On top of that, due to the additional latency considerations, HYB is able to bring the latency down faster when the buffer is healthy. This can be seen in both graphs between segments 65 and 90, where BUF reduces the latency from about 3 seconds to 2.1 seconds, while HYB reduces the latency from about 3 seconds to 1.6 seconds within the same period.

5) Impact of the Throughput Measurement Module: We conducted a set of experiments to evaluate the effectiveness of Algorithm 1 by calculating the mean absolute error (MAE) for each of the bandwidth profiles and then averaging them to obtain an overall measurement error. We compare this accuracy against the default throughput measurement provided by dash.js. Results show that our module is able to reduce the overall measurement error from 0.67 to 0.30, compared to the algorithm in dash.js, across all bandwidth profiles. Fig. 5 shows how the true and measured values vary over time using one sample run for three bandwidth profiles. We observe that our throughput measurement can accurately track the actual available bandwidth while the dash.js throughput measurement algorithm suffers especially when the transmission is source-limited and the inter-chunk idle periods become more significant. As noted earlier, when these periods are not excluded in the calculation of segment download time, throughput measurement errors are inevitable. While the earlier versions of dash.js (*i.e.*, v2.9.x) did not perform any chunk filtering and consequently often underestimated the throughput, the more recent versions (*i.e.*, v3.x) include chunk filtering, which addresses the underestimation problem. However, the bandwidth measurements now seem to err in the opposite direction and overestimate the measured throughput.

6) Impact of the ABR Algorithm: The main aim of these experiments is to show how responsive LoL$^+$ is to the QoE metrics considered in (4). We compare the performance of LoL$^+$ against the other submissions in the Twitch grand challenge, L2A and STALLION, as well as against the Dynamic algorithm provided by dash.js. These ABR algorithms are specifically designed for near-second latency [33] and would provide us with a good benchmark of how LoL$^+$ performs.

The results of these experiments are shown in Table VII and Fig. 6. One key finding is that LoL$^+$ reduces the rebuffering duration for all profiles compared to LoL, with an average reduction of 62.6%. In particular, significant improvements are achieved in profile TW-LOW (rebuffering reduced by 98.4%),

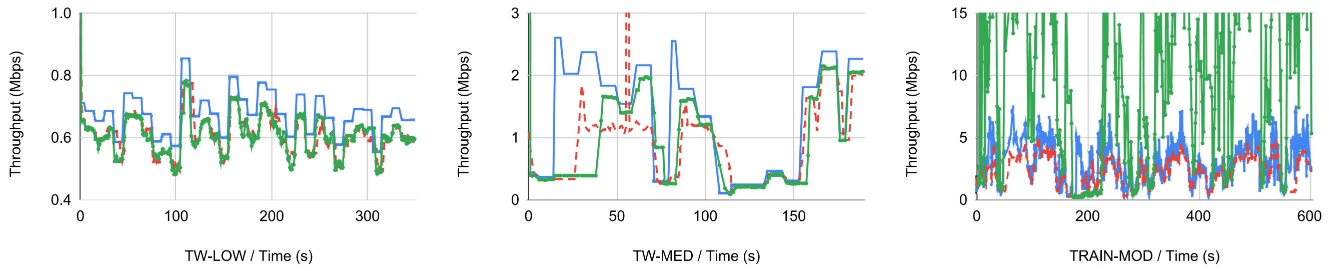
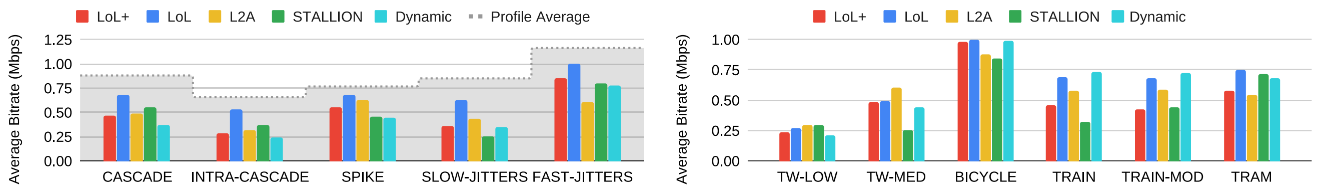
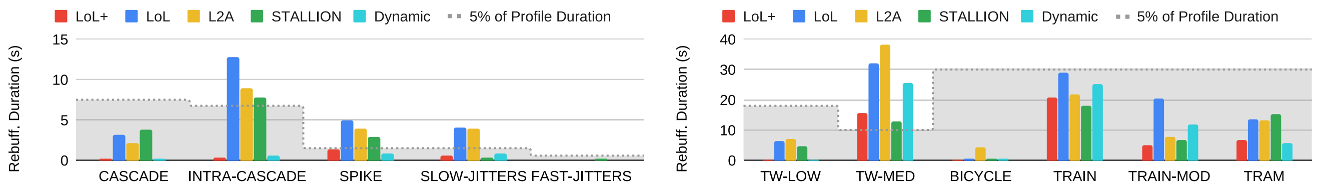


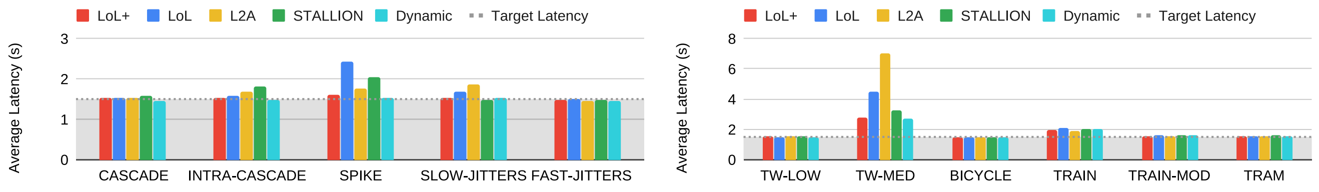
Fig. 5. Throughput measurements made by dash.js (dotted green line) and our implementation (dashed red line) compared to the true values (solid blue line) for three bandwidth profiles.



(a) Average bitrate (Mbps).



(b) Average rebuffering duration (s).



(c) Average latency (s).

Fig. 6. Average results of the ABR algorithms across different bandwidth profiles. (a) Average bitrate (Mbps). (b) Average rebuffering duration (s). (c) Average latency (s).

INTRA-CASCADE (rebuffering reduced by 97.3%) and CASCADE (rebuffering reduced by 95.2%). Latency has also improved with an average reduction of 8.5% across all profiles. Although the improvements in rebuffering are accompanied by some reduction in the average bitrate across all profiles, this trade-off seems reasonable based on the guidance from the industry [33].

Similarly, LoL$^+$ achieves significant improvements in rebuffering over L2A, with an average reduction of 61.9% in rebuffering duration across all profiles. Latency is also improved with an average reduction of 8.1% across all profiles. The impact on the average bitrate is far less significant with an average reduction of 6.6% across all profiles.

When comparing LoL$^+$ and STALLION, the improvements are consistent across all three key metrics. LoL$^+$ achieves an average reduction of 42.7% and 5.2% in rebuffering and latency, respectively. The average bitrate also improves by an average increase of 12.5% across all profiles.

The results of Dynamic are most similar to those of LoL$^+$. Nonetheless, LoL$^+$ still improves on rebuffering with an average reduction of 12.3% across all profiles. The difference is insignificant for their bitrate and latency performances (0.9% and 1.6%, respectively). That is, both have a similar bitrate and latency performance.

D. Summary of the Results

As seen in the earlier experiments, different ABR algorithms may perform differently based on which metrics they tend to prioritize or sacrifice, which makes comparisons between the algorithms tricky. One way to compare them is to identify the worst and best performing algorithm in each metric and quantify

TABLE VII
IMPACT OF THE ABR ALGORITHM

ABR	Avg. Bitrate (Kbps)	Avg. Rebuffering (s)	Avg. Latency (s)	Avg. Switches (#)
CASCADE (150 seconds)				
LoL$^+$	469.91	0.15	1.52	55.0
LoL	675.08	3.11	1.52	5.00
L2A	491.77	2.11	1.52	10.4
STA	546.49	3.82	1.59	8.00
DYN	371.87	0.15	1.46	34.40
INTRA-CASCADE (135 seconds)				
LoL$^+$	281.98	0.35	1.53	44.4
LoL	533.10	12.77	1.57	5.00
L2A	316.06	8.94	1.69	8.20
STA	367.18	7.82	1.8	4.00
DYN	239.38	0.56	1.49	22.4
SPIKE (30 seconds)				
LoL$^+$	555.02	1.30	1.61	9.00
LoL	675.20	4.92	2.44	4.00
L2A	626.18	3.93	1.75	3.00
STA	458.87	2.87	2.04	3.40
DYN	447.96	0.80	1.53	7.00
SLOW-JITTERS (30 seconds)				
LoL$^+$	354.04	0.61	1.54	14.2
LoL	630.72	4.02	1.68	11.6
L2A	429.00	3.97	1.86	5.2
STA	248.06	0.35	1.48	2.0
DYN	348.92	0.81	1.54	7.4
FAST-JITTERS (11.6 seconds)				
LoL$^+$	852.29	0	1.48	3.4
LoL	1000	0	1.50	1.0
L2A	602.14	0	1.46	1.6
STA	798.33	0.15	1.47	3.6
DYN	774.76	0	1.46	3.2
TW-LOW (350 seconds)				
LoL$^+$	238.03	0.10	1.54	53.8
LoL	271.73	6.26	1.51	9.8
L2A	295.71	6.9	1.54	45.0
STA	299.47	4.7	1.56	23.4
DYN	207.45	0.10	1.46	12.80
TW-MED (190 seconds)				
LoL$^+$	485.85	15.55	2.80	20.6
LoL	495.93	32.03	4.50	12.2
L2A	606.16	38.31	7.03	10.8
STA	255.17	12.85	3.26	5.20
DYN	437.27	25.57	2.74	22.6
BICYCLE (600 seconds)				
LoL$^+$	979.10	0.30	1.49	20.8
LoL	998.35	0.45	1.5	4.60
L2A	876.96	4.44	1.5	11.8
STA	842.56	0.65	1.48	6.00
DYN	987.24	0.51	1.48	8.80
TRAIN (600 seconds)				
LoL$^+$	457.45	20.75	1.98	81.8
LoL	693.69	29.04	2.11	33.4
L2A	581.15	21.64	1.91	41.8
STA	321.73	18.01	2.01	12.8
DYN	728.37	25.31	2.04	76.00
TRAIN-MODIFIED (600 seconds)				
LoL$^+$	427.71	4.93	1.54	71.2
LoL	682.58	20.27	1.63	41.2
L2A	587.98	7.72	1.53	44.2
STA	445.17	6.68	1.58	19.2
DYN	727.05	11.84	1.58	75.0
TRAM (600 seconds)				
LoL$^+$	576.39	6.81	1.55	114
LoL	751.18	13.65	1.53	47.8
L2A	540.46	13.25	1.54	43.8
STA	718.80	15.4	1.6	18.6
DYN	684.92	5.64	1.53	70.6

the relative performance of the other algorithms against these boundaries so that users may choose an algorithm based on their own priorities and preferences. Hence, we normalize the QoE metrics into performance scores between 1 (worst) and 5 (best). The normalized scores are shown in Table VIII. For each metric, an ABR algorithm's performance score is calculated based on its achieved result in proportion to the worst and best performing results across all algorithms in that profile. We then calculate the

TABLE VIII
AVERAGE SCORES ACHIEVED BY EACH ALGORITHM ACROSS ALL BANDWIDTH PROFILES. THE BLUE/RED SCORES REPRESENT THE BEST/WORST SCORES FOR EACH QOE METRIC. HIGHER IS BETTER

ABR	Avg. Bitrate Score	Avg. Rebuffering Score	Avg. Latency Score	Avg. Switches Score
LoL$^+$	2.53	**4.76**	3.77	*1.09*
LoL	**4.69**	*1.97*	*2.40*	4.13
L2A	2.92	2.36	3.08	3.76
STA	*2.26*	3.19	2.84	**4.45**
DYN	2.76	4.34	**4.52**	2.53

average performance scores across all profiles to observe their overall performance.

We see that LoL achieves the best score in the average bitrate and lowest score in rebuffering and latency. STALLION achieves the best score in the average number of bitrate switches and the lowest score in the average bitrate. Dynamic has the best latency score while L2A has no best or worst score in any of the metrics. LoL$^+$ has the best score in rebuffering but this comes at the expense of the worst score in the average number of bitrate switches. We observe that LoL$^+$ outperforms LoL in rebuffering and latency but not in number of bitrate switches. This is because LoL$^+$ is more responsive to the varying conditions in the network.

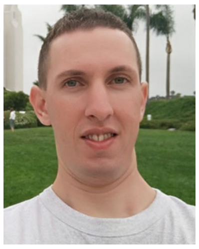
V. CONCLUSION

Existing player implementations struggle when operating with a tiny amount of buffered media. Thus, we designed LoL$^+$– a player specifically designed for LLL streaming. LoL$^+$ delivers good QoE for any given target latency. Its source code is available online [2] and has already been merged into the main dash.js branch. In the future, we plan on contributing LoL$^+$ to other open-source player implementations including Google's Shaka player and hls.js.

REFERENCES

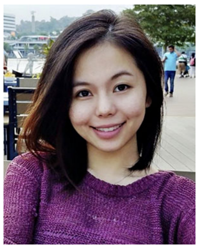
[1] DASH Reference Player, Accessed: Aug. 18, 2020. [Online]. Available: https://reference.dashif.org/dash.js/
[2] Low-on-Latency-plus (LoL$^+$), Accessed: Sep. 5, 2020. [Online]. Available: https://github.com/NUStreaming/LoL-plus
[3] Adobe, *Real-Time Messaging Protocol (RTMP)*. [Online]. Available: https://www.adobe.com/devnet/rtmp.html
[4] A. Bentaleb, A. C. Begen, S. Harous, and R. Zimmermann, "Data-driven bandwidth prediction models and automated model selection for low latency, *IEEE Trans. Multimedia*," to be published, doi: 10.1109/TMM.2020.3013387.
[5] A. Bentaleb, B. Taani, A. C. Begen, C. Timmerer, and R. Zimmermann, "A survey on bitrate adaptation schemes for streaming media over HTTP," *IEEE Commun. Surv. Tut.*, vol. 21, no. 1, pp. 562–585, Jan.–Mar. 2019.
[6] A. Bentaleb, C. Timmerer, A. C. Begen, and R. Zimmermann, "Bandwidth prediction in low-latency chunked streaming," in *Proc. 29th ACM Workshop Netw. Operating Syst. Support Digit. Audio Video*, 2019, pp. 7–13.
[7] A. Bentaleb, C. Timmerer, A. C. Begen, and R. Zimmermann, "Performance analysis of ACTE: A bandwidth prediction method for low-latency chunked streaming," *ACM Trans. Multimedia Comput., Commun. Appl.*, vol. 16, no. 2s, pp. 1–24, 2020.
[8] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex Optimization*. Cambridge, USA: Cambridge Univ. Press, 2004.
[9] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems: Revised Reprint*. Philadelphia, PA, USA: SIAM, 2012.
[10] S. D'Aronco, L. Toni, and P. Frossard, "Price-based controller for utility-aware HTTP adaptive streaming," *IEEE MultiMedia*, vol. 24, no. 2, pp. 20–29, Apr.–Jun. 2017.

[11] Y. Dogan, D. Birant, and A. Kut, “SOM : Integration of self organizing map and k-means algorithms,” in *Proc. Int. Workshop Mach. Learn. Data Mining Pattern Recognit.*, 2013, pp. 246–259.
[12] K. Du *et al.*, “Server-driven video streaming for deep learning inference,” in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protoc. Comput. Commun.*, 2020, pp. 557–570.
[13] A. El Essaili, T. Lohmar, and M. Ibrahim, “Realization and evaluation of an end-to-end low latency live DASH system,” in *Proc. IEEE Int. Symp. Broadband Multimedia Syst. Broadcasting*, 2018, pp. 1–5.
[14] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proc. 13th Int. Conf. Artif. Intell. Stat.*, 2010, pp. 249–256.
[15] C. Gutterman, B. Fridman, T. Gilliland, Y. Hu, and G. Zussman, “STALLION: Video adaptation algorithm for low-latency video streaming,” in *Proc. 11th ACM Multimedia Syst. Conf.*, 2020, pp. 327–332.
[16] P. Houze, E. Mory, G. Texier, and G. Simon, “Applicative-layer multipath for low-latency adaptive live streaming,” in *Proc. IEEE Int. Conf. Commun.*, 2016, pp. 1–7.
[17] *Information Technology - Multimedia Application Format (MPEG-A) - Part 19: Common Media Application Format (CMAF) for Segmented Media*, ISO/IEC Standard 23000-19:2020. [Online]. Available: https://www.iso.org/standard/79106.html
[18] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang, “CFA: A practical prediction system for video QoE optimization,” in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 137–150.
[19] J. Jiang, V. Sekar, and H. Zhang, “Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE,” in *Proc. 8th Int. Conf. Emerg. Netw. Exp. Technol.*, 2012, pp. 97–108.
[20] J. Jiang, S. Sun, V. Sekar, and H. Zhang, “Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation,” in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 393–406.
[21] T. Karagkioules, R. Mekuria, D. Griffioen, and A. Wagenaar, “Online learning for low-latency adaptive streaming,” in *Proc. 11th ACM Multimedia Syst. Conf.*, 2020, pp. 315–320.
[22] J. Kim, Y. Jung, H. Yeo, J. Ye, and D. Han, “Neural-enhanced live streaming: Improving live video ingest via online learning,” in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protoc. Comput. Commun.*, 2020, pp. 107–125.
[23] T. Kohonen, “The self-organizing map,” *Proc. IEEE*, vol. 78, no. 9, pp. 1464–1480, Sep. 1990.
[24] T. Kohonen, *Self-Organizing Maps*, vol. 30. Berlin, Germany: Springer Science & Business Media, 2012.
[25] M. Lim, M. N. Akcay, A. Bentaleb, A. C. Begen, and R. Zimmermann, “When they go high, we go low: Low-latency live streaming in dash.js with LoL,” in *Proc. 11th ACM Multimedia Syst. Conf.*, 2020, pp. 321–326.
[26] S. Park, S. Seo, C. Jeong, and J. Kim, “The weights initialization methodology of unsupervised neural networks to improve clustering stability,” *J. Supercomput.*, vol. 76, pp. 1–17, 2019.
[27] H. Peng, Y. Zhang, Y. Yang, and J. Yan, “A hybrid control scheme for adaptive live streaming,” in *Proc. 27th ACM Int. Conf. Multimedia*, 2019, pp. 2627–2631.
[28] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia, “A survey on quality of experience of HTTP adaptive streaming,” *IEEE Commun. Surv. Tut.*, vol. 17, no. 1, pp. 469–492, Jan.–Mar. 2015.
[29] Y. Shuai and T. Herfet, “Towards reduced latency in adaptive live streaming,” in *Proc. 15th IEEE Annu. Consum. Commun. Netw. Conf.*, 2018, pp. 1–4.
[30] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, “BOLA: Near-optimal bitrate adaptation for online videos,” in *Proc. IEEE 35th Annu. Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[31] Y. Sun *et al.*, “CS2P: Improving video bitrate selection and adaptation with data-driven throughput prediction,” in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 272–285.
[32] V. Swaminathan and S. Wei, “Low latency live video streaming using HTTP chunked encoding,” in *Proc. IEEE 13th Int. Workshop Multimedia Signal Process.*, 2011, pp. 1–6.
[33] Twitch, “Grand challenge on adaptation algorithms for near-second latency,” in *ACM Multimedia Syst. Conf.*, 2020.
[34] J. Van Der *et al.*, “HTTP/2-based adaptive streaming of HEVC video over 4 G/LTE networks,” *IEEE Commun. Lett.*, vol. 20, no. 11, pp. 2177–2180, Nov. 2016.
[35] J. Van Der *et al.*, “An HTTP/2 push-based approach for low-latency live streaming with super-short segments,” *J. Netw. Syst. Manage.*, vol. 26, no. 1, pp. 51–78, 2018.
[36] M. B. Yahia, Y. L. Louedec, G. Simon, L. Nuaymi, and X. Corbillon, “HTTP/2-based frame discarding for low-latency adaptive video streaming,” *ACM Trans. Multimedia Comput., Commun. Appl.*, vol. 15, no. 1, pp. 1–23, 2019.
[37] G. Yi *et al.*, “The ACM multimedia 2019 live video streaming grand challenge,” in *Proc. 27th ACM Int. Conf. Multimedia*, 2019, pp. 2622–2626.
[38] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, “A control theoretic approach for dynamic adaptive video streaming over HTTP,” in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 325–338.
[39] G. Yushuo, Z. Yuanxing, W. Bingxuan, B. Kaigui, X. Xiaoliang, and S. Lingyang, “PERM: Neural adaptive video streaming with multi-path transmission,” in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2020, pp. 1103–1112.



Abdelhak Bentaleb (Member, IEEE) received the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2019. He is currently a Research Fellow with same department. He is the Co-Founder of Atlastream Inc., Singapore. His research interests include multimedia systems and communication, video streaming architectures, content delivery, distributed computing, computer networks and protocols, wireless communications, and mobile networks. He was the recipient of many prestigious awards, including the SIGMM Award for Outstanding Ph.D. Thesis Award, the DASH-IF Best Ph.D. Dissertation Award, and the Dean’s Graduate Research Excellence Award AY2018/2019.

Mehmet N. Akcay received the B.Sc. degree in the field of computer engineering from Istanbul Technical University, Istanbul, Turkey, in 2005 and the M.Sc. degree in the field of computer engineering from Bogazici University, Istanbul, Turkey, in 2008. He is currently working toward the Ph.D. degree in computer science with Ozyegin University, Istanbul, Turkey. He was with industry for more than 10 years. His research interests include HTTP adaptive streaming, low-latency live streaming, and software verification using formal methods.



May Lim received the B.E.Sc. and M.Sc. degrees from Nanyang Technological University, Singapore, in 2015. She is currently working toward the Ph.D. degree in computer science with the National University of Singapore, Singapore. Her current research focuses on multimedia streaming systems and she worked on relating to low-latency streaming for live 2D and 6DoF videos.

Ali C. Begen (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Georgia Tech., Atlanta, GA, USA. Since 2001, he has been a Research and Development Engineer and has broad experience in mathematical modeling, performance analysis, standards development, intellectual property and innovation. Between 2007 and 2015, he was with the Video and Content Platforms Research and Advanced Development Group with Cisco, where he designed and developed algorithms, protocols, products and solutions in the service provider and enterprise video domains. He is currently affiliated with Ozyegin University, Istanbul, Turkey, where he teaches and advises students with Computer Science Department. He was the recipient of a number of academic and industry awards, and was granted more than 30 U.S. patents. He is a Senior Member of the ACM. In 2016, he was elected Distinguished Lecturer by the IEEE Communications Society, and in 2018, he was re-elected for another two-year term. In 2017, he initiated and since then has been chairing the Turkish delegation for JPEG and MPEG under ISO/IEC JTC1/SC29. He was also listed among the world's most influential scientists in the subfield of networking and telecommunications in 2020.

Roger Zimmermann (Senior Member, IEEE) received the M.S. and Ph.D. degrees from the University of Southern California, Los Angeles, CA, USA, respectively. He is currently a Professor with the Department of Computer Science, National University of Singapore (NUS), Singapore. He is also a Lead Investigator with the Grab-NUS AI Lab and from 2011 to 2021, he was the Deputy Director with the Smart Systems Institute, NUS. He has coauthored a book, seven patents, and more than 330 conference publications, journal articles, and book chapters in the areas of multimedia processing, networking and data analytics. He is a Distinguished Member of the ACM. He recently was the Secretary of ACM SIGSPATIAL from 2014 to 2017, the Director of the IEEE Multimedia Communications Technical Committee Review Board, and an Editorial Board Member of the Springer MTAP journal. He is also an Associate Editor for the IEEE MultiMedia, *ACM Transactions on Multimedia Computing, Communications, and Applications,* andIEEE Open Journal of the Communications Society.