Rldish: Edge-Assisted QoE Optimization of HTTP Live Streaming with Reinforcement Learning

Huan Wang[1], Kui Wu[1], Jianping Wang[2], and Guoming Tang[3]
[1]Department of Computer Science, University of Victoria, B.C., Canada
[2]Department of Computer Science, City University of Hong Kong, Hong Kong
[3]Peng Cheng Laboratory, Shenzhen, China

***Abstract*—Recent years have seen a rapidly increasing traffic demand for HTTP-based high-quality live video streaming. The surging traffic demand, as well as the real-time property of live videos, make it challenging for content delivery networks (CDNs) to guarantee the Quality-of-Experiences (QoE) of viewers. The initial video segment (*IVS*) of live streaming plays an important role in the QoE of live viewers, particularly when users require fast join time and smooth view experience. State-of-the-art research on this regard estimates network throughput for each viewer and thus may incur a large overhead that offsets the benefit. To tackle the problem, we propose *Rldish*, a scheme deployed at the edge CDN server, to dynamically select a suitable *IVS* for new live viewers based on Reinforcement Learning (RL). *Rldish* is transparent to both the client and the streaming server. It collects the real-time QoE observations from the edge without any client-side assistance, then uses these QoE observations as real-time rewards in RL. We deploy *Rldish* as a virtualized network function (VNF) in a real HTTP cache server, and evaluate its performance using streaming servers distributed over the world. Our experiments show that *Rldish* improves the state-of-the-art *IVS* selection scheme w.r.t. the average QoE of live viewers by up to 22%.**

I. INTRODUCTION

Background: Live large-scale events drive nearly 10 times more viewer engagement than on-demand videos, and the HTTP-based live streaming has been gaining increasing popularity in recent years [1]. Due to the stringent real-time requirement and high traffic spikes of live videos, content delivery networks (CDNs) continue to struggle with delivering high-quality live videos to viewers while guaranteeing their Quality-of-Experiences (QoE) [2], [3]. Since live viewers are normally very sensitive to QoE deterioration such as slow startup time and the dreaded spinning pinwheel (i.e., video buffering), lower QoE generally means a higher abandonment rate of users.

To address such an issue, CDN operators rely on the widely distributed edge servers (e.g., the edge data centers [4]) to handle the increasing demand of live videos. Using edge servers as the cache, most viewers can fetch the requested contents directly from the cache rather than the original video sources. Nevertheless, the first batch of requests would still miss the edge cache due to the real-time property of live streaming [5]. Even worse, the performance of TCP-based content transmission may decline due to the long-latency backhaul in the content delivery path [6].

The existing research mainly uses adaptive bitrate algorithms to improve the QoE of viewers via dynamically choosing a bitrate for each video segment [7], [8], [9]. While these efforts have shown considerable QoE improvement, they may lead to decreased video quality during the streaming process. Many studies have pointed out that users may quickly abandon a video session if the quality is not sufficient [10], [11]. An alternative method to improve the QoE is to conduct transient holding of a minimum number of video segments at the edge servers such that the clients can select a suitable initial video segment (*IVS*) that best matches their network throughput [12] to start the playback. Our experience in real-world live video streaming suggests that this is a much more effective way for QoE improvement.

Existing *IVS* selection strategies either use a fixed value [13] or use the "optimal" value [12]. In the former, the RFC standard of HTTP Live Streaming (HLS) suggests that "client should not choose a segment that starts within *three* segment durations (the maximum playback duration of video segments in the playlist) from the end of the playlist file" in order to avoid playback stalls [13]. In the latter, the "optimal" *IVS* value is derived to match the current network conditions (e.g., throughput) [12].

Clearly, the former will not work well for high-quality live video streaming due to the dynamic network conditions. The latter is promising but has two main pitfalls. First, it relies on the *per-user* based network throughput estimation. The network throughput is related to multiple complex factors (e.g., RTT and router buffer size), which frequently changes over time [14]. This can incur high computational overhead, especially for live videos where the number of live viewers is large [5]. Second, when a user joins a live channel, the server can only infer the user's network throughput through the signal strength (e.g., RSRP, RSRQ and RSSI in LTE) and mobility pattern (e.g., fast, slow, static). The network throughput estimation in this case may not be accurate, leading to a suboptimal choice of *IVS*. In practice, the overhead of searching for the "optimal" value may offset the benefit. It is also hard to quickly react to the network condition change.

Our ideas: To overcome the above problem, we propose a reinforcement learning-based dynamic *IVS* selection scheme (*Rldish*) deployed on edge CDN server to maintain a balance between exploring suboptimal decisions and exploiting currently optimal decisions. *Rldish* uses a real-time *exploration and exploitation (E2) model [15] to learn the IVS selection automatically, and is deployed as a virtualized network function*

(VNF) on the CDN edge server by the CDN operator. It can work seamlessly with existing edge CDN proxy (cache) server (e.g., Nginx) [5], [16], and can also react to the network condition (throughput) change via real-time exploration.

Rldish makes the *IVS* decisions on a *per-stream basis* to avoid high overhead in per-user based throughput estimation. Since an edge CDN server generally serves its proximal end users, viewers accessing the same live video usually share the common video delivery path from the origin server to the edge and generally experience the similar network conditions when fetching the same video from the edge. Based on this observation, *Rldish* continuously updates the currently optimal decisions on *IVS* selection for the live viewers on a *per-stream basis*, based on the real-time QoE measurements and feedback. The decisions will then be updated into the media playlist files of each stream for the subsequent live viewers.

Challenges: Our goal is to implement *Rldish* only at the edge servers and make no changes on either the client or the origin server. We are faced with the following challenges:

First, how to mitigate the negative impact of caching on the learning performance? It is uncertain that whether an exploration action will lead to a cache hit at the edge server. In other words, the same *IVS* may miss the edge cache sometimes but hits the cache at other times. Since cache hit or miss would greatly impact the QoE of viewers, it may "confuse" RL due to significant different rewards fed by the same choice.

Second, how to conduct real-time QoE measurements at the edge? RL requires feedback of viewers' QoE to compute the reward of previous explorations. However, the viewers' QoE metrics (e.g., startup latency, video buffering ratio/time) are generally collected at the client side. Modifying the client side to collaborate with the edge is not desirable.

Third, how to support different types of QoE objectives? Live videos consist of different types of traffic (e.g., live event streaming, and user-generated live videos). Viewers may have different QoE metrics for different contents. For example, live event streaming might value the lower playback latency more.

Contributions: This paper makes the following contributions:

- *Rldish* is the first research work to apply RL to *IVS* selection of HTTP-based live streaming to optimize the QoE of live viewers. It collects the real-time QoE observations from the edge as the rewards of explorations without client-side assistance, and is completely transparent to both the clients and the streaming servers.
- We systematically tackled the above technical challenges. For the first challenge, we define a new coordinate system for RL choices by considering the real-time caching status of each live stream in the edge server. Each RL exploration is then calibrated based on its relative position in the coordinate system such that caching status rather than a static *IVS* is used in the exploration. For the second challenge, we dive into Nginx kernel and make changes to collect the TCP and HTTP performance data (e.g., HTTP response time and RTT) of live viewers to further generate the QoE data. For the third challenge, we define a reward function by elaborately considering three different types of QoE metrics of live video and offering options for service providers to tune their QoE preferences.
- We cast the RL problem in the core of *Rldish* as a non-stationary multi-armed bandit (MAB) problem and enhance the Discounted-UCB (D-UCB) [17], a variant of Upper Confidence Bound (UCB) algorithm, for the special needs of *Rldish*.
- We deploy *Rldish* as a virtualized network function (VNF) in a real HTTP cache server, and perform extensive real-world experiments to evaluate its performance using streaming servers distributed over the world. Evaluation results show that *Rldish* improves the state-of-the-art *IVS* selection scheme w.r.t. the average QoE of live viewers by up to 22%.

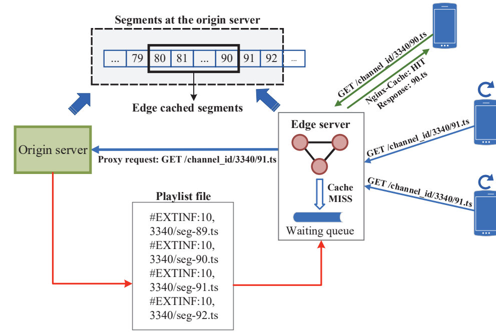


Fig. 1. Architecture of live video delivery over edge servers.

II. Background and Overview of Rldish

A. HTTP-based Live Video Delivery

HTTP-based live video streaming has gained increasing popularity in recent years owing to that HTTP is compatible with large numbers of client-side applications (e.g., web browsers and mobile applications) [3]. As illustrated in Fig. 1, once a raw live video is generated from the source (i.e., the broadcaster), it is first uploaded to the origin server, where it is encoded into multiple streams with different pre-determined bitrates. The server then splits each stream into a sequence of small video segments. To watch videos, the clients download the segments sequentially with HTTP GET [13], [2].

Every time when a client joins a live channel, she first needs to request the playlist file (shown in Fig. 1) from the origin server which contains a list of up-to-date (i.e., the segments that can be readily fetched) video segments of the requested live channel. Based on this playlist, the client then chooses an initial video segment to start the playback [13]. Afterwards, the client generally plays the video segments in the order that they appear in the playlist.

For live streaming, this playlist file is continuously updated by the streaming server once new segments have been generated. The client continuously accesses the newest playlist file to know the up-to-date video segment information (i.e., URI). Since caching video segments at the edge server is generally triggered by viewer requests and the video segments need time

to be delivered from the streaming server to the edge server, it is possible that the newest several video segments may not be fully cached in the edge server when their information is already shown in the playlist file (e.g., segment 91th and 92th in Fig. 1).

B. The Impact of IVS on QoE of Live Viewers

The selection of *IVS* can impact the QoE of live viewers greatly. We use the example in Fig. 1 to explain. The figure shows a scenario where segments of a live stream (with bitrate 3340 kbps) from sequence number 0th to 92th have all been generated at the origin server, while the newest two segments 91th and 92th have not yet been cached at the edge server. At the same time, three new live viewers near the edge server try to join this live channel based on the information of the fetched playlist.

Although playing the newest segment (segment 92th) would provide the users with the smallest streaming latency, it may generate *playback stalls* for the watching process later on, since there are no pre-buffered segments at the edge that can be used for the subsequent segment requests of the client. Since the edge server typically issues a new HTTP request to the origin server to fetch the segment once a cache miss happens, this also induces a high *startup latency* for live viewers. A naïve idea to reduce the high startup latency and remove the buffering events is to join the live channel with a relatively conservative *IVS* (e.g., segment 89th in Fig. 1). Nevertheless, a too conservative *IVS* may lead to unnecessary latency (i.e., a receiver's playback is far behind the live streaming source).

Therefore, it is critical to develop a dynamic *IVS* selection policy to optimize QoE. The policy needs to consider both the complex caching status and real-time network conditions. Note that QoE of live viewers is mainly determined by backhaul throughput under the cloud-edge streaming infrastructure because viewers generally have sufficient download speeds from their local edge CDN servers. One might consider that when the backhaul throughput is under a poor condition, QoE of live viewers will get worse again after a certain period of video watching even if it started with the right *IVS*. Actually, the proxy (edge) server normally maintains multiple keep-alive TCP connections with the streaming server. Thus, the download processes of multiple video segments will be conducted simultaneously if the backhaul throughput is less than the average bitrates of live stream. Therefore, the edge server can ensure the client, once it selects the right *IVS*, to have local access to the subsequent video segments [12].

C. Overview of Rldish

Rldish realizes the dynamic *IVS* selection policy within the edge server. We present the core design and the implementation of *Rldish* in Fig. 2. It consists of three key components:

1) *QoE Collector*: it communicates with the (HTTP) proxy server in real time to collect the QoE data (e.g., startup latency, video buffering time) of live viewers (refer to § III-C for more details).

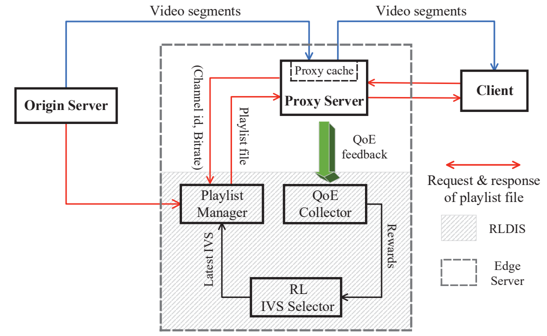


Fig. 2. System design of *Rldish*.

2) *RL based IVS Selector*: it accepts the fresh data of the rewards feedback from the *QoE Collector* to continuously update the latest *IVS* selection for each live stream by running a reinforcement learning algorithm.
3) *Playlist Manager*: it periodically sends requests to the origin server for the most up-to-date playlist files of live streams currently accessed by local users. It maintains the playlist files of all live streams in the local cache by using the new *IVS* (from the RL) to update the original playlist files.

When a user joins a live channel, the proxy server will locate the first request for the user's playlist file to the local edge cache. By identifying the requested channel ID and bitrate information from the request URI, the corresponding playlist file for the requested stream can be obtained from the *Playlist Manager*. Once the first requested playlist file has been successfully delivered to the client, the subsequent requests for playlist files are handled by the proxy server independently: it returns the up-to-date unmodified playlist file to the client. In practice, there are many ways for the HTTP proxy server to distinguish whether a playlist request is from a new client or not so as to take different response decisions. One of the easiest way is to use an HTTP Cookie to indicate the state information of a connection.

After the user has played the live stream for a certain time (e.g., 3 minutes after the first playlist request), *QoE Collector* collects data (e.g., startup latency, total buffering duration) from the proxy server by analyzing the transmission finish time of each individual segment and their expected playback time at the client end. The collected QoE data, as the feedback of the previous trial, are reported to the RL module, which provides the *Playlist Manager* with the newest *IVS* information to update the playlist files for new coming live viewers. In this way, *Rldish* could control the playback behaviour of all viewers served at the edge. *Note that clients can still adjust their video bitrates during the watching process to obtain better QoE*, since *Rldish* only manipulates the *IVS* selection step.

In the following, we introduce the core RL algorithms used by the *IVS Selector*.

III. CORE ALGORITHMS

A. Discounted-UCB (D-UCB) for Non-stationary MAB

The problem of dynamically learning and choosing *IVS* in real time according to time-varying network conditions can be modelled as the non-stationary multi-armed bandit (MAB) problem [17], where the bandit facing K arms needs to decide which arm to play when the environment may change (i.e., the distribution of rewards may change) over time. This model is consistent with our situation where the network conditions may change over time (for both paths from the origin to the edge and from the edge to the end users). We modify Discounted-UCB (D-UCB) [17], a variant of UCB (Upper Confidence Bound) algorithm, to solve this problem. For the paper to be self-contained, we introduce the details of D-UCB.

The D-UCB algorithm can adapt to the QoE drift of the live streaming, since it automatically gives higher weight to more recent measurements by exponentially discounting historical measurements with a discount factor [15], [17]. The details of D-UCB are shown in Algorithm 1, where $T \to \infty$. At each time t, the bandit chooses an arm $I_t \in \{1, \cdots, K\}$ with the highest instantaneous expected reward $\bar{X}_t(\gamma, i) + c_t(\gamma, i)$, where $\bar{X}_t(\gamma, i)$ (shown in Equation (1a)) is the discounted empirical average of the observed rewards, and $\gamma \in (0, 1)$ is a discount factor to control the algorithm's preference degree for the recent measurements. The smaller the γ, the higher discount rate on the historical measurements, therefore the higher weight on recent measurements. $X_s(i)$ denotes the instantaneous reward of arm i at time s.

$$\bar{X}_t(\gamma, i) = \frac{1}{N_t(\gamma, i)} \sum_{s=1}^{t} \gamma^{t-s} X_s(i) \mathbb{1}_{\{I_s = i\}} \tag{1a}$$

$$N_t(\gamma, i) = \sum_{s=1}^{t} \gamma^{t-s} \mathbb{1}_{\{I_s = i\}}, \tag{1b}$$

$$\mathbb{1}_{\{I_s = i\}} = \begin{cases} 1 \text{ , if } I_s = i, \\ 0 \text{ , otherwise.} \end{cases} \tag{1c}$$

$c_t(\gamma, i)$ is the discounted padding function, which is defined as follows:

$$c_t(\gamma, i) = 2B \sqrt{\frac{\xi \log n_t(\gamma)}{N_t(\gamma, i)}}, \ n_t(\gamma) = \sum_{i=1}^{K} N_t(\gamma, i), \tag{2}$$

where B is the upper bound on the rewards and $\xi > 0$ is a parameter to control the probability of exploration. When an arm is frequently used in the past, its padding function gets smaller than the other arms, so that the other arms get a chance of being explored [14].

Algorithm 1: Discounted UCB

```
1 for t from 1 to K, play arm I_t = t;
2 for t from K+1 to T do
3     play arm I_t = argmax_{1≤i≤K} X̄_t(γ,i) + c_t(γ,i).
```

B. Tailored D-UCB Algorithm

The D-UCB shown in Algorithm 1 needs to keep all the historical rewards (i.e., $X_s(i), \forall s \in \{1, \cdots, t\}$, $i \in \{1, \cdots, K\}$) to calculate $\bar{X}_t(\gamma, i)$ and $c_t(\gamma, i)$ for each time step, which may result in high computational overhead. Considering the limited resource of edge server and high traffic volume of live streaming, we need to tailor the D-UCB algorithm to solve our problem more efficiently.

Let $\hat{X}_t(\gamma, i) := \sum_{s=1}^{t} \gamma^{t-s} X_s(i) \mathbb{1}_{I_s = i}$. We have $\bar{X}_t(\gamma, i) = \frac{\hat{X}_t(\gamma, i)}{N_t(\gamma, i)}$. Instead of recalculating $\hat{X}_t(\gamma, i)$ and $N_t(\gamma, i)$ with the historical rewards each time, the calculations could be completed easily using their previous states, i.e.,

$$N_t(\gamma, i) = \gamma^{\Delta_i(t)} \times N_r(\gamma, i) + \mathbb{1}_{\{I_t = i\}}, \tag{3a}$$

$$\hat{X}_t(\gamma, i) = \gamma^{\Delta_i(t)} \times \hat{X}_r(\gamma, i) + X_t(i) \mathbb{1}_{\{I_t = i\}}, \tag{3b}$$

where r is the last time (before t) that arm i was selected and $\Delta_i(t) = t - r$. For a given γ and time t, we represent the $N_t(\gamma, i)$ and $\hat{X}_t(\gamma, i)$ of all arms in the format of vector, i.e., $\boldsymbol{N} = [N_t(\gamma, 1), \cdots, N_t(\gamma, K)]$ and $\boldsymbol{X} = [\hat{X}_t(\gamma, 1), \cdots, \hat{X}_t(\gamma, K)]$, to allow easy updates on $\bar{X}_t(\gamma, i)$ and $c_t(\gamma, i)$. Then the instantaneous expected rewards for all the arms at time t can be computed as:

$$\mathbf{R} := \boldsymbol{X} \oslash \boldsymbol{N} + 2B((\xi \log \|\boldsymbol{N}\|_1) \boldsymbol{N}^{\circ -1})^{\circ \frac{1}{2}} \tag{4}$$

where $\oslash$ and $\circ$ denote the entrywise division and power operation, respectively. The details of the tailored D-UCB is shown in Algorithm 2, where $\boldsymbol{e}_i$ denotes $(0, \cdots, 0, 1, 0, \cdots, 0)$ where the ith element is 1 and all other elements are 0.

Algorithm 2: Tailored D-UCB

```
1 X = N = R = 0 ∈ ℝ^K
2 for t from 1 to T do
3     play arm I_t = t (If t ≤ K)
4     play arm I_t = argmax_{1≤i≤K} R (Otherwise)
5     X_t(i) ← Get the reward of I_t, i ← I_t
6     X = γX + X_t(i)e_i, N = γN + e_i
7     calculate R (refer to Equation (4))
```

C. Definition of Reward Function

In order to use RL to dynamically select the best *IVS* to optimize QoE, we need to define the metrics that measure the QoE of live viewers. While QoE may be measured in many different angles and there is no consensus on the QoE metrics, we use the most-adopted ones for QoE evaluation in live streaming [18], [19], including:

- *General latency (gl)*: the delay time that the viewer's playback is behind the video source's live production progress.
- *Startup latency (sl)*: the delay between the time when the user sends the first request and the time when the playback starts.
- *Buffering time (bt)*: the total time of playback stalls experienced by the live viewer.

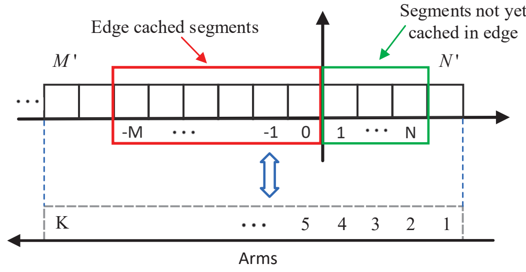


Fig. 3. Illustration of arm definition for RL.

The above three metrics together are sufficient to describe the QoE of live viewers. In addition, these metrics are closely related to the *IVS* selection: A high startup latency is normally caused by the cache miss of the first requested segment (*IVS*) in the edge server; a high general latency is mainly caused by a too conservative *IVS* (i.e., segment far away from the end of original playlist); a long buffering duration is mainly due to a too aggressive *IVS* (i.e., segment too close to the end of original playlist). Combining all the above three metrics, our reward function is given in Equation (5), where $\alpha+\beta+\delta=1$ and $0 \leq X_t(i) \leq 1$.

$$X_t(i) = 1 - (\alpha * \frac{sl_t(i)}{sl_{max}} + \beta * \frac{gl_t(i)}{gl_{max}} + \delta * \frac{bt_t(i)}{bt_{max}}). \quad (5)$$

In reward function (5), $sl_t(i)$, $gl_t(i)$, and $bt_t(i)$ denote the startup latency, general latency, and buffering time of arm i at time t, respectively. Accordingly, sl_{max}, gl_{max} and bt_{max} denotes the maximum startup latency, the maximum general latency, and the maximum buffering time observed in the history, respectively. α, β and δ are the weight factors of the three different QoE metrics. In practice, since different live video providers may emphasize different QoE metrics (e.g., streaming of live events may favor lower general latency), they could customize this reward function by using different values of scale factors (i.e., α, β and δ).

D. Definition of Arms

The D-UCB algorithm requires a decision arm space with discrete values, which is in accord with our problem with discrete video segment choices. An intuitive idea is to define an arm as the gap between a certain segment and the last segment (in the current playlist). However, this would lead to the problem raised as Challenge 1 in § I.

To solve this problem, we define the arms by *considering the real-time position of video segments in the edge cache instead of their position in the playlist*. Fig. 3 shows the general caching status of a live stream at an arbitrary time instant, where each video segment is denoted by a square (the closer to the right, the newer the segment). Video segments that have already been cached in the edge are marked by the red box, and the segments that have been generated (shown in the playlist) but have not been cached in the edge server are marked by the green box. To ease illustration, we label the segments from $-M$ to N, where $-M$ and N correspond to the oldest segment cached in the edge and the newest segment shown in the playlist file, respectively, and 0 refers to the newest segment currently in the edge cache. Note that since edge caching is a dynamic process triggered by user requests, M and N are dynamic (non-negative) values during the whole streaming process.

We then select a fixed, sufficiently large segment interval $[M', N']$ such that $[-M, N] \in [-M', N']$ holds all the time. Based on the new interval, we could setup a *one-to-one* mapping between the arm set $\{1, \cdots, K\}$ and $\{-M', \cdots, N'\}$, as shown in Fig. 3, where $K = M' + N' + 1$. Therefore, whenever an *IVS* choice ($I_t \in \{1, \cdots, K\}$) is given by RL, *Playlist Manager* first uses the above mapping to find the corresponding video segment in the playlist, then modifies the playlist file accordingly.

TABLE I
HTTP REQUEST & RESPONSE DATA

Metrics	Description
Request Processing Time (rpt)	Time elapsed from when the edge receives the first byte of request until the last byte of response is acknowledged by the client
Upstream Response Time (urt)	Time elapsed from when the edge establishes a connection to an upstream server until it receives the last byte of the response body
Segment Size (ss)	Size of the requested segment
Round Trip Time (rtt)	Smoothed round-trip time (srtt) between the edge and the client
Response Finish Time (rft)	Local time when the last byte of response is acknowledged by the client
Caching Status	Segment request HIT or MISS at the edge cache

IV. Implementation of Key Components

A. QoE Collector

To measure the QoE metrics from the edge side, *QoE Collector* collects the performance data of HTTP interactions of each live session from Nginx, as shown in Table I, all of which can be tracked during the interaction process for each HTTP request. It then calculates QoE metrics, the startup latency, buffering time, and general latency, using the collected information.

1) Startup Latency (sl): Fig. 4 shows the basic HTTP interactions between the edge and the client. Let l and s denote the request for playlist file and *IVS*, respectively. Based on the Caching Status of each HTTP requests (shown in Table I), *Rldish* measures the startup latency of each live video session as follows.

When an *IVS* request misses the edge cache,

$$sl := \frac{t_4 - t_3}{ss} * \bar{ss} + (t_3 - t_1), \quad (6)$$

where ss and $\bar{ss}$ are the size of *IVS* and the average video segment size of this live stream, respectively, and t_1, t_3, t_4 are calculated with the performance data collected (refer to Table I and Fig. 4) by $t_1 = rft_l - rtt - rpt_l$, $t_3 = t_2 + urt_s (t_2 = rft_s - rtt - rpt_s)$, and $t_4 = rft_s - rtt$, respectively. Note that $(t_4 - t_3)$ is the *IVS* transmission time, and $(t_3 - t_1)$ is the

sum of the other startup time excluding the *IVS* transmission time.

It is worth mentioning that even for the same live stream (with the same bitrate), the sizes of generated video segments (with the same playback duration) are not exactly the same. Different sizes of *IVS* generally bring different segment transmission time, which implies different startup latency and different rewards. Our reward function implicitly considers the impact of different *IVS* sizes on RL by normalizing the video transmission time in (6).

When an *IVS* request hits the edge cache, urt_s becomes (nearly) 0 and the startup latency could be calculated by $\frac{(t_4-t_2)}{ss} * \bar{ss} + (t_2 - t_1)$ instead. Note that we use t_1 as the request starting time to calculate startup latency without including the time for (local) TCP connection establishment between the edge and the client. This is because: i) the connection establishment time is irrelevant to the arm selection, and ii) the time is much shorter than the *IVS* transmission time.

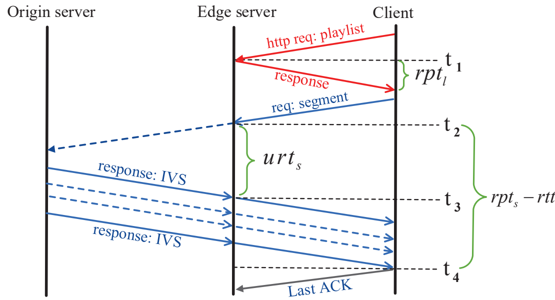


Fig. 4. Illustration of HTTP interactions of live streaming between the client and the edge server.

2) Buffering Time (bt): To calculate the buffering time of a live viewer, we collect the *rft* of all video segment requests during a certain time period from the beginning of the live session. The total buffering time could be derived as follows:

$$bt = \sum_{i=2}^{N} max\{rft_i - rpt_i, 0\}, \tag{7}$$

where rft_i is the response finish time of ith video segment and rpt_i denotes the playback start time of ith segment (the time when the playback of $(i-1)^{th}$ segment is finished). If we use pt_{i-1} to denote the real playback start time of $(i-1)^{th}$ segment, then we have $rpt_i = pt_{i-1} + sd$, where sd is the segment (playback) duration. We can calculate pt_{i-1}, recursively, with $pt_{i-1} = max\{rpt_{i-1}, rft_{i-1}\}$. Note that when $i = 2$, pt_{i-1} is the startup latency sl, and the corresponding rpt_{i-1} is 0 (no startup latency). In this recursive way, we can obtain the total buffering time.

3) General Latency (gl): The general latency of a live session is defined as the segment duration time multiplied by the number of segments between *IVS* and the newest segment in the playlist when a user sends her first request. This metric can be easily calculated since all the related values are readily available. Note that in a given live stream, while the segments may have different sizes, their duration time is all the same (e.g., each segment lasts for 5 seconds).

B. Playlist Manager

The *Playlist Manager* generates the final playlist files that will be accessed by clients, and it needs to communicate with multiple units to accomplish that function. To reduce the overhead, the playlist file of a live stream is not generated individually for each new request. Instead, there are only *three types of events* in our system that can trigger the update of playlist file of a live stream: i) RL module provides a new choice (different from previous one) of *IVS* of the live stream, ii) a fresh new playlist file with the description of new segments is obtained from the origin server, and iii) new segments of the live stream are cached in the edge server.

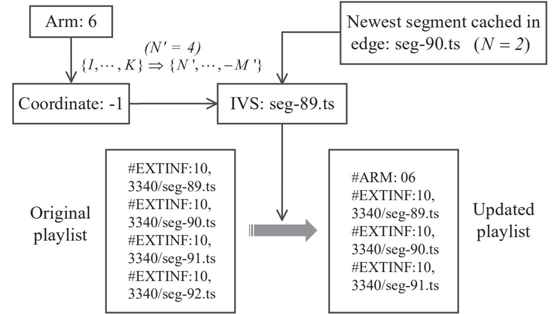


Fig. 5. Example of playlist file update procedure.

The first two types of events are controlled by *Rldish* itself, while the last type of events (i.e., edge caching) is managed by the Nginx server. To know exactly the edge caching status, the *Playlist Manager* monitors the edge cache in real time to find if new cache files have been generated and which video segments are in the cache files.

Using HLS protocol as the example, Fig. 5 shows an example on how to update the playlist file, where arm 6 given by RL will first be directed to video segment 89th. Then we remove the information of segment 92th from the original playlist so that the HLS client will play segment 89th as the *IVS* to start the live streaming. Note that the information of 90th and 91th could also be eliminated from the updated playlist in this example. Also note that the HLS client by default starts the playback from fragment $N-2$, N being the last segment of the live playlist [13]. Thus, segment 90th would be played as the *IVS* if the playlist is not updated by the *Playlist Manager*

We add a new tag `#ARM` in the updated playlist to mark the arm selection of each playlist file during the whole streaming process. The *QoE Collector* module will then obtain the value of this tag from the HTTP response to know the arm selection of each live session (persistent HTTP connection used), i.e., the mapping between arm selections and their corresponding QoE rewards, as introduced in § III-C and § III-D.

V. Performance Evaluation

A. Experimental Setup

Clients & servers setup: We use HLS as the streaming protocol in our experiment. Based on the well-known hls.js (version 0.12.4) library [20], we implement the HTTP live streaming client relying on HTML5 video. Our client adopts the default HLS configuration which proactively buffers the future video segments when the duration of total buffered segments is less than 30 seconds or the total buffered segment size is less than 60 MB. The client runs on a Google Chrome browser (version 75) on top of Windows PCs.

The edge cache server is deployed on a local data center in Hong Kong with an average 8 ms RTT to our client machines. The bandwidth for a single segment download is set to 64 Mbps, which is in accord with general download speed from a local CDN edge server. It runs Nginx (version 1.9.9) as the HTTP reverse proxy. *Rldish* is implemented and deployed as a Docker [21] container on this edge server. Using the RL algorithm introduced in § III, it accesses the QoE observations from Nginx and provides a revised playlist file to the Nginx through shared volume mounted from the host machine.

Since in practice the origin server managed by the content providers could be in different locations, we create 3 VM instances, located at East Coast North America, West Coast North America, and Japan, respectively, to serve as the origin streaming servers. The first two instances are launched on East Cloud and Arbutus Cloud of Compute Canada, respectively, and the third VM in Tokyo is launched via Google Cloud. The average RTTs from our edge server to the 3 VMs are 234 ms, 156 ms, and 52 ms, respectively. Each VM runs the system of Ubuntu 18.04 with kernel 4.15.0-34 as well as an Apache HTTP web server.

Live video setup: To generate the source of live video streaming, we use FFmpeg [22] to convert a local 2K High Frame Rate (HFR) video file (MP4 format) into 3 HLS live source with average bitrates 24 Mbps (2K HFR), 16 Mbps (2K standard), and 8 Mbps (1080P), respectively. Each live source is further generated into two HLS streams with a video segment length of 5 seconds and 10 seconds, respectively. We deploy the video segments of the 6 HLS streams (i.e., stream with quality 2K HFR, 2K standard and 1080p with segment length 5s and 10s respectively) onto each of the above-mentioned 3 origin servers. Then for each stream, we create a background process (in each of the VM), which periodically (every 5 or 10 seconds) generates a new HLS playlist file according to the normal segment playback sequence (i.e., updating the playlist file by adding new segment information into it). The streaming process at the origin server is repeated, once it reaches the last segment of a video.

Test scenarios: To evaluate the performance of *Rldish* and other schemes, we set up multiple HLS clients. Each client joins a live stream by first sending the HTTP request to HTTP proxy server in the edge (in practice, DNS CNAME redirection is commonly used to accomplish this procedure). It then plays the live video for 2 minutes. The QoE data of the live viewers are collected during this period.

To evaluate the performance of *Rldish* under changing network conditions, we manually set up the streaming servers to limit the segment download throughput for a single request. When conducting our experiment on a given live stream, we setup the backhaul network throughput range from $1/3$ the stream bitrate to the stream bitrate. Refer to Fig. 8 for more details on the throughput setting. For a given throughput, each of the clients repeatedly joins the live streaming and plays the video 30 times based on the real-time *IVS* decisions made by *Rldish*. The network throughput then changes to another value within the range, to evaluate if *Rldish* could quickly react to the network condition change. In order to collect sufficient QoE observations to evaluate the performance of *Rldish*, our HLS clients join each of the live streams broadcast by each of our streaming servers (18 live streams in total).

B. Evaluation Methodology

We evaluate the performance of *Rldish* and compare it with that of two other schemes:

ETHLE: this scheme was proposed in [12], which calculates the *IVS* choice (x) based on the following inequality:

$$\arg\min_x x \geq \frac{1}{l_{seg}}(d_{startup} + \frac{s_{seg} - th_{startup}}{bw_{max}}), \quad (8)$$

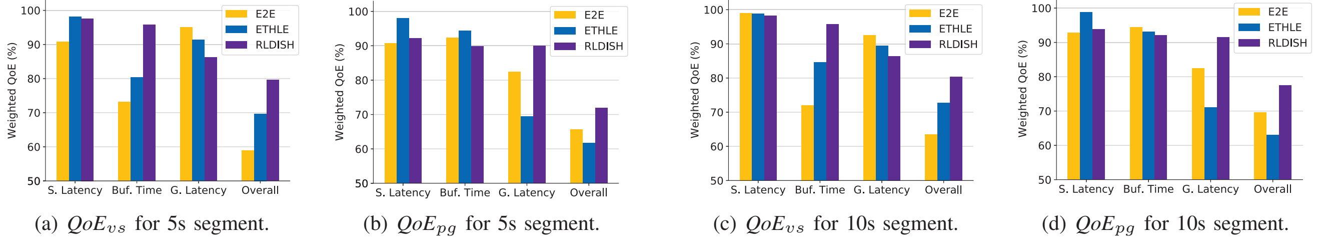
where l_{seg} and s_{seg} are the duration and size of a single video segment, respectively, $d_{startup}$ and $th_{startup}$ are the duration and total number of bytes transferred in the slow start period of TCP, respectively. bw_{max} is the bottleneck bandwidth of the backhaul link. We assume this scheme has the complete network throughput information (i.e., our backhual throughput setting) without using the estimation method. Refer to [12] for more details of the algorithm.

E2E: this scheme serves the live viewers via the CDN edge server directly. It does not adopt dynamic *IVS* selection at the edge, and the client adopts the default HLS setting which starts the playback from segment $N-2$, where N is the last fragment of the live playlist.

For comparison, we also provide the results for the *offline optimal* scheme, which makes an *offline* selection on an *IVS* with the highest average QoE (based on our former experiments) under the current network condition. It has the complete network throughput information and thus represents the best performance that *Rldish* and other methods can possibly achieve. To ease analysis, the performance data (i.e., the QoE of viewers) are collected at the edge directly using the methods introduced in § IV. It is worth mentioning that with Chrome DevTools, we validated that the QoE data collected at the browser side are pretty close to those collected at the edge.

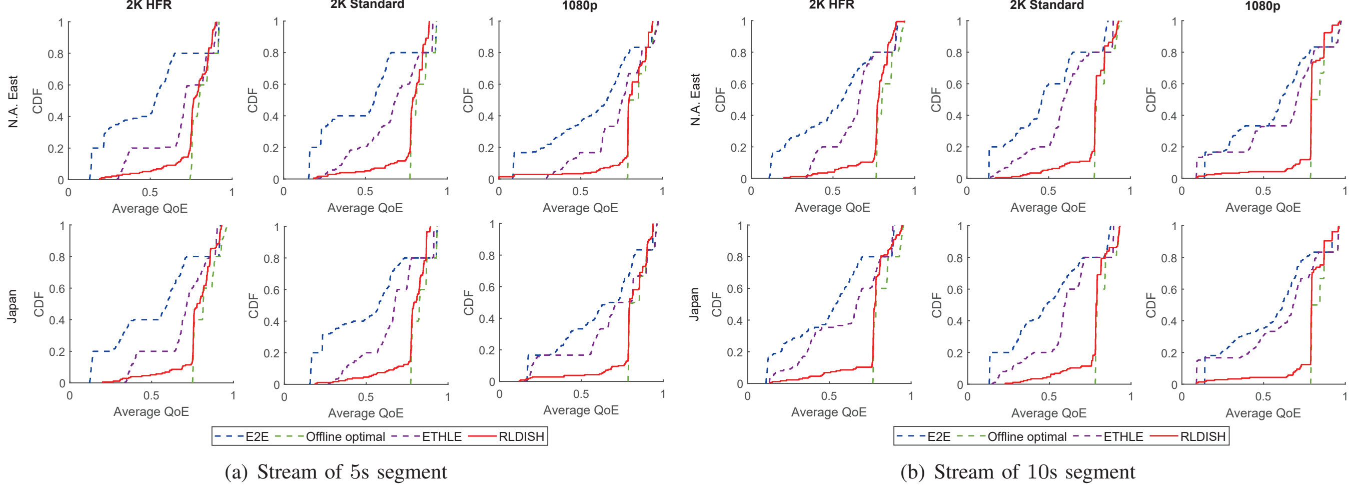
C. QoE Criteria

There are well-defined metrics for the QoE of live viewers, which have been studied in [18], [19]. The QoE metrics considered in these work generally include startup latency, buffering ratio/time, playback delay (general latency), and



(a) QoE_{vs} for 5s segment. (b) QoE_{pg} for 5s segment. (c) QoE_{vs} for 10s segment. (d) QoE_{pg} for 10s segment.

Fig. 6. The average performance on QoE of *Rldish* and other schemes for all live streams. The results are normalized and weighted based on the QoE criteria used. Refer to the error bars of Fig. 8 for the QoE distributions of *Rldish*.



(a) Stream of 5s segment (b) Stream of 10s segment

Fig. 7. CDF results of QoE_{vs} of live viewers for *Rldish* and other schemes with streaming severs located in North America and Japan respectively.

rate of bitrate fluctuation. Since the bitrate decrease generally means QoE deterioration [23], this QoE deterioration can be roughly reflected by the total buffering time. Therefore, we consider startup latency (*sl*), total buffering time (*bt*), and general latency (*gl*) as defined in Section III-C.

(1) QoE_{vs}: this QoE criterion favors low video buffering time by assigning a high weight to the live streaming with a low buffering time. It focuses on visual comfort of viewers. In our experiments, the weights of *sl*, *gl*, and *bt* in this QoE criterion are set to $\alpha = 10\%$, $\beta = 30\%$ and $\delta = 60\%$, respectively.

(2) QoE_{pg}: this QoE criterion favors low general latency (i.e., the progress of playback). It gives a high weight to the streaming with low general latency and thus focuses more on the timeliness of live streaming. Note that while QoE_{pg} values more on general latency, it does not necessarily mean that live viewers have to experience playback stalls during the watching process. Actually, the playback stalls can be avoided by adapting to a lower bitrate. In our experiments, the weights of *sl*, *gl*, and *bt* in this QoE criterion are set to $\alpha = 10\%$, $\beta = 60\%$ and $\delta = 30\%$, respectively.

Remark 1. *It is more illustrative to use a higher QoE score to mean better QoE. Since the lower values in startup latency, general latency, and total buffering time mean better QoE, we use the following equation to transform a metric to a (normalized and weighted) QoE score:*

$$QoE(m) = 1 - w * \frac{m_{ob}}{m_{max}}, \tag{9}$$

where m denotes a QoE metric (i.e., startup latency, general latency, or total buffering time), m_{ob} is the observed value of m and m_{max} is the historical worst value of m, w denotes the weight of metric m. With this transform, a ***higher QoE score means a better performance****. All experimental results in the latter figures use the above transform.*

D. Performance Evaluation

1) Overall Performance: The average QoE for a certain segment length of each scheme (180 runs over 9 live streams, 1620 runs in total) is given in Fig. 6, where two QoE criteria QoE_{vs} and QoE_{pg} are considered. The figure shows the detailed QoE scores of all the schemes with the weighted score on each QoE metric. In the case of live stream with segments of 5 seconds, *Rldish* improves ETHLE by 14.2% and 16.5% w.r.t. QoE_{vs} and QoE_{pg}, respectively, and improves E2E by 35.9% and 9.8% w.r.t. QoE_{vs} and QoE_{pg}, respectively. In the case of live stream with segments of 10 seconds, *Rldish* improves ETHLE by 10.3% and 22.8% w.r.t. QoE_{vs} and QoE_{pg}, respectively, and improves E2E by 26.5% and 11.2% w.r.t. QoE_{vs} and QoE_{pg}, respectively.

Recall that QoE_{vs} favors low buffering time. *Rldish* under this QoE criterion sacrifices slightly on the general latency (by using a relative conservative *IVS*) in order to guarantee lower buffering time for the viewer. With QoE_{pg}, *Rldish* improves its performance on general latency while the performance on buffering time is degraded. The service provider can adjust

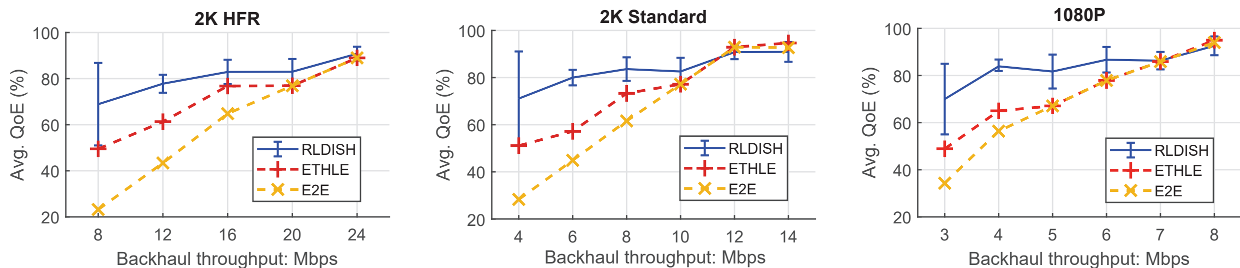


Fig. 8. The average QoE_{vs} of different schemes under different network throughput using the dataset N.A. West VM as the streaming server

the weights of different QoE metrics in the reward function to satisfy different QoE objectives.

Since ETHLE and E2E schemes could not adapt to different QoE objectives, their detailed QoE metrics (e.g., buffering time) should remain the same for the two QoE criteria cases. However, Fig. 6 shows a performance deterioration on QoE scores for the two schemes from QoE_{vs} to QoE_{pg}. This is because the QoE scores could get worse when the same latency value is multiplied by a higher weight (refer to (9)). Fig. 7 provides more detailed QoE_{vs} results in the form of CDFs for each live stream. Since the QoE results do not show too much difference between the streaming servers located in North America (N.A.) West and N.A. East, only N.A. East results are shown in this figure.

There are two key takeaways from Fig. 7. First, *Rldish* provides the majority of live viewers with the QoE score at around $75\% - 80\%$ (We can see a sharp increase during that range in the CDF). By conducting deep analysis on the QoE data from our experiments, we find that these QoE values are mainly contributed by the *IVS* selections at a critical point: these are the *IVS* choices which reach a good balance in buffering time and general latency for the viewer. At the critical point, *Rldish* should not use a newer segment as the *IVS* since that will lead to the punishments on playback stalls. These *IVS* selections normally lead to small buffering time (close to zero) and similar general latency. Since they are usually the newest video segment in the edge server (*IVS* just hits the edge cache), their startup latency is also pretty small.

Second, *Rldish* may generate a small fraction of poor decisions. It has overall around 5% of total viewer's QoE in the range of 20% to 50% according to CDF. These poor decisions are mainly caused by the exploration choices of RL including the startup explorations. This problem could be alleviated, since startup explorations of RL could be accomplished by pre-testing clients (machines) located near the edge server. Fig. 8 breaks down the QoE scores of all schemes under different throughput of backhaul network. Despite a relatively high standard deviation on QoE scores during the startup phase of RL (shown with error bars), *Rldish* outperforms the other schemes when backhaul network throughput is low.

VI. Related Work

Related work can be primarily grouped into two categories: QoE enhancement via bitrate adaption and network throughput prediction, and optimization of *IVS* selections. In the first category, research efforts have been devoted to adaptive live streaming algorithms. In [24], Bruneau-Queyreix et al. proposed a prototype of a hybrid P2P/multi-server video quality-adaptive streaming solution, which simultaneously uses multiple servers and peers to enhance the QoE of live viewers with expanded bandwidth and link diversity. Aiming at satisfying live viewers' personalized QoE, Wang et al. [25] proposed a reinforcement learning-based solution to automatically learn the transcoding selections so that the backhaul bandwidth could be saved. To reduce the general latency of live videos, the authors in [26] proposed an adaptation bitrate algorithm for HTTP-based live streaming by conducting TCP throughput prediction. The research of QoE optimization of live videos has also been conducted in [27], [7] and [28].

In the second category, Ge et al. [12] improved the QoE of live viewers by holding a minimum number of video segments at the edge server such that the clients can select the optimal *IVS* to best match their network throughput. While this work has shown decent QoE improvement, it may have two pitfalls. First, it relies on the *per-user* based network throughput estimation to derive the optimal *IVS* values, which may lead to high computational overhead. Second, this scheme may not quickly react to network throughput changes.

VII. Conclusion

By designing, implementing, and evaluating *Rldish*, this paper tackled the technical challenges in applying RL to *IVS* selection of HTTP-based live streaming to improve the QoE of live viewers. Compared with the state-of-the-art solutions, *Rldish* is lightweight and completely transparent to both the clients and the streaming server. More importantly, *Rldish* solved three critical difficulties of using RL in this specific application context, and enhanced the existing D-UCB RL algorithm for the special needs of *Rldish*. We deployed *Rldish* as a virtualized network function (VNF) in a real HTTP cache server, and performed extensive real-world experiments to evaluate its performance. The evaluation results show that *Rldish* improves the state-of-the-art *IVS* selection scheme w.r.t. the average QoE of live viewers by 10% to 22%.

Acknowledgment

This work was supported in part by Mitacs (No. FR36061), Natural Sciences and Engineering Research Council of Canada (NSERC) (No. RGPIN-2018-03896), Hong Kong General Research Fund under project 11216618, National Natural Science Foundation of China (No. 61802421, 61828202), and China Postdoctoral Science Foundation (No. 2019M663017).

REFERENCES

[1] Cisco, "Global mobile data traffic forecast update, 2015–2020," *White Paper, February*, vol. 3, 2016.

[2] M. K. Mukerjee, D. Naylor, J. Jiang, D. Han, S. Seshan, and H. Zhang, "Practical, real-time centralized control for cdn-based live video delivery," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 311–324, 2015.

[3] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017, pp. 197–210.

[4] D. Puthal, M. S. Obaidat, P. Nanda, M. Prasad, S. P. Mohanty, and A. Y. Zomaya, "Secure and sustainable load balancing of edge data centers in fog computing," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 60–65, 2018.

[5] F. Larumbe and A. Mathur, "Under the hood: Broadcasting live video to millions," https://code.fb.com/ios/under-the-hood-broadcasting-live-video-to-millions/.

[6] C. Ge, N. Wang, G. Foster, and M. Wilson, "Toward qoe-assured 4k video-on-demand delivery through mobile edge virtualization with adaptive prefetching," *IEEE Transactions on Multimedia (TMM)*, vol. 19, no. 10, pp. 2222–2237, 2017.

[7] A. Detti, B. Ricci, and N. Blefari-Melazzi, "Tracker-assisted rate adaptation for mpeg dash live streaming," in *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2016, pp. 1–9.

[8] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over http," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 325–338.

[9] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli, "Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 272–285.

[10] F. Dobrian, A. Awan, D. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang, "Understanding the impact of video quality on user engagement," *Communications of the ACM*, vol. 56, no. 3, pp. 91–99, 2013.

[11] S. S. Krishnan and R. K. Sitaraman, "Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 6, pp. 2001–2014, 2013.

[12] C. Ge, N. Wang, W. K. Chai, and H. Hellwagner, "Qoe-assured 4k http live streaming via transient segment holding at mobile edge," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 36, no. 8, pp. 1816–1830, 2018.

[13] J. Roger Pantos, William May, "RFC 8216: HTTP Live Streaming," https://tools.ietf.org/html/rfc8216.

[14] X. Nie, Y. Zhao, D. Pei, G. Chen, K. Sui, and J. Zhang, "Reducing web latency through dynamically setting tcp initial window with reinforcement learning," in *Quality of Service (IWQoS), 2018 IEEE/ACM 26th International Symposium on*. IEEE, 2018.

[15] J. Jiang, S. Sun, V. Sekar, and H. Zhang, "Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 393–406.

[16] T. Mauro, "Why Netflix Chose NGINX as the Heart of Its CDN," https://www.nginx.com/blog/why-netflix-chose-nginx-as-the-heart-of-its-cdn/.

[17] A. Garivier and E. Moulines, "On upper-confidence bound policies for switching bandit problems," in *International Conference on Algorithmic Learning Theory (ALT)*. Springer, 2011, pp. 174–188.

[18] A. Ahmed, Z. Shafiq, and A. Khakpour, "Qoe analysis of a large-scale live video streaming event," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 395–396, 2016.

[19] 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 Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 469–492, 2014.

[20] "hls.js," https://github.com/video-dev/hls.js/.

[21] "Docker," https://www.docker.com/.

[22] "FFmpeg," https://ffmpeg.org/.

[23] S. Tavakoli, S. Egger, M. Seufert, R. Schatz, K. Brunnström, and N. Garcia, "Perceptual quality of http adaptive streaming strategies: Cross-experimental analysis of multi-laboratory and crowdsourced subjective studies," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 34, no. 8, pp. 2141–2153, 2016.

[24] J. Bruneau-Queyreix, M. Lacaud, and D. Négru, "A hybrid p2p/multi-server quality-adaptive live-streaming solution enhancing end-user's qoe," in *Proceedings of the 25th ACM International Conference on Multimedia*. ACM, 2017, pp. 1261–1262.

[25] F. Wang, C. Zhang, J. Liu, Y. Zhu, H. Pang, L. Sun *et al.*, "Intelligent edge-assisted crowdcast with deep reinforcement learning for personalized qoe," in *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2019, pp. 910–918.

[26] K. Miller, A.-K. Al-Tamimi, and A. Wolisz, "Qoe-based low-delay live streaming using throughput predictions," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, vol. 13, no. 1, p. 4, 2017.

[27] T. C. Thang, H. T. Le, A. T. Pham, and Y. M. Ro, "An evaluation of bitrate adaptation methods for http live streaming," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 32, no. 4, pp. 693–705, 2014.

[28] R. Huysegems, J. Van Der Hooft, T. Bostoen, P. Rondao Alface, S. Petrangeli, T. Wauters, and F. De Turck, "Http/2-based methods to improve the live experience of adaptive streaming," in *Proceedings of the 23rd ACM International Conference on Multimedia*. ACM, 2015, pp. 541–550.