Cross-layer Network Bandwidth Estimation for Low-latency Live ABR Streaming

Chinmaey Shende[1], Cheonjin Park[1], Subhabrata Sen[2], Bing Wang[1]
[1]University of Connecticut [2]AT&T Labs - Research

ABSTRACT

Low-latency live (LLL) adaptive bitrate (ABR) streaming relies critically on accurate bandwidth estimation to react to dynamic network conditions. While existing studies have proposed bandwidth estimation techniques for LLL streaming, these approaches are at the application level, and their accuracy is limited by the distorted timing information observed at the application level. In this paper, we propose a novel cross-layer approach that uses coarse-grained application-level semantics and fine-grained kernel-level packet capture to obtain accurate bandwidth estimation. We incorporate this technique in three popular open-source ABR players and show that it provides significantly more accurate bandwidth estimation than the state-of-the-art application-level approaches. In addition, the more accurate bandwidth estimation leads to better bandwidth prediction, which we show can lead to significantly better quality of experience (QoE) for end users.

CCS CONCEPTS

• **Information systems → Multimedia streaming**.

KEYWORDS

ABR streaming; Low-latency live streaming; Bandwidth estimation.

ACM Reference Format:
Chinmaey Shende, Cheonjin Park, Subhabrata Sen, and Bing Wang. 2023. Cross-layer Network Bandwidth Estimation for Low-latency Live ABR Streaming. In *Proceedings of the 14th ACM Multimedia Systems Conference (MMSys '23), June 7–10, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3587819.3590990

1 INTRODUCTION

Bandwidth estimation, i.e., estimating the rate of data transfer from a server to a client over a network setting, is important for many applications. One example is adaptive bitrate (ABR) streaming, where a client estimates the dynamic network bandwidth and adapts to it to maximize its QoE. In traditional ABR steaming, while bandwidth estimation is often necessary, accurate bandwidth estimation is not very critical—the client can buffer future content ahead of time, which can be tens of seconds for Video on Demand (VOD) and 10-15 seconds for live streaming. The buffer thus provides a cushion for the application, e.g., even if a bandwidth drop is not predicted accurately, the client can play back the content that is already prefetched in the buffer.

New applications, however, have drastically changed the above landscape. One example is low-latency live (LLL) streaming, which requires very low *target latency* (e.g., the playback cannot fall behind the live edge more than 5 seconds), for live sports and other live events such as online journalism, education, and virtual reality. To meet the low latency requirement, the state-of-the-art technique is using MPEG Common Media Application Format (CMAF) [8] coupled with HTTP/1.1 chunked transfer encoding (CTE) [23]. CMAF enhances the traditional media packaging by introducing very short decodable units, called *chunks*. Specifically, the origin server provides multiple *tracks* of the video that are encoded at different bitrate/quality levels, each divided into multiple *segments*, and each segment is further divided into chunks (e.g., a segment is a few seconds long, while a chunk is tens to hundreds of milliseconds long). The chunks in a segment can be transferred using CTE before the segment is fully encoded. Similarly, the client can decode and play back a chunk without waiting to receive the full segment.

Even with CMAF and CTE, high-quality LLL streaming is extremely challenging, evidenced by recent grand challenges organized by academia and industry [4, 64]. Specifically, assuming a target latency Δ (e.g., $\Delta = 1$ or 3 seconds), the amount of prefetched content is at best ahead of the playback by Δ, and hence the buffer becomes a much less effective cushioning mechanism. Instead, the application needs *accurate realtime* bandwidth estimation in order to react to the dynamic network conditions. The traditional bandwidth estimation method, which simply estimates the bandwidth as the number of bytes downloaded over a period of time divided by the duration of the time period, does not work for LLL streaming and can end up underestimatiing the available network bandwidth [12]. This is because the data transfer can be *source-constrained*: when the data transfer of one chunk is completed, the next chunk may still be in the process of being generated at the server, not ready to be transferred.

Existing studies have proposed bandwidth estimation techniques for LLL streaming (see §7). All these techniques are at the application level, by instrumenting the application, and use different heuristics to decide what data and time periods should be used to handle the idle periods in chunked-based data transfer. Such approaches are easy to implement since application-level information is easily accessible. However, we argue that they have fundamental limitations: timing information observed at the application level does not reflect the network bandwidth accurately, since it is subject to various end-system impacts (e.g., buffering, specific handling of TCP packets). Therefore, it is challenging to obtain accurate bandwidth estimation using an application-level approach.

In this paper, we propose a novel *cross-layer* approach that combines kernel-level packet capture and application semantics for

MMSys '23, June 7–10, 2023, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0148-1/23/06…$15.00
https://doi.org/10.1145/3587819.3590990

accurate bandwidth estimation. In addition to LLL streaming, this approach can be used for many other low-latency applications such as teleconferencing, cloud gaming, and augmented reality, that use mechanisms similar to ABR streaming to react to network dynamics. Our study makes the following main contributions:

• We propose a novel direction of cross-layer bandwidth estimation (§2) and develop one such technique for LLL streaming (§3). This technique, called *Cross-layer Bandwidth Estimation (CLBE)*, does not introduce any additional probing traffic to the network. It combines fine-grained kernel-level packet capture that provides high-fidelity network bandwidth information with coarse-grained application-level semantics to estimate network bandwidth accurately.

• We incorporate CLBE in three popular open-source ABR players, `dash.js` [17], ExoPlayer [27], and Shaka Player [28] that support LLL streaming (§4). Using a wide range of settings (§5), we demonstrate that CLBE provides significantly more accurate bandwidth estimation than state-of-the-art application-level approaches (§6). Even for highly dynamic cellular network bandwidth scenarios, 74%-78% of its bandwidth estimation errors are within 10%, while for the state-of-the-art application-level approaches, only 10-17% of the errors are within 10%. When coupling the accurate bandwidth estimation from CLBE with the bandwidth prediction modules in these three players, the prediction is much more accurate than that of the original players: 70%-73% of the prediction errors are within 20%, while even for the best prediction in the original players, only 24%-46% of the prediction errors are within 20%.

• We demonstrate that incorporating CLBE in the three players can lead to significantly better QoE than the original players (§6). The improvement includes less low-quality segments, and/or lower stall duration and live latency.

In this paper, we focus on improving bandwidth estimation, the first step of the logic flow that precedes bandwidth prediction and ABR logic, since it is very important—inaccurate information at the beginning of the flow will be hard to correct in the later stages. In addition, being the first step, a new bandwidth estimation technique can be easily incorporated in existing ABR players (see §4). While our results demonstrate that just the more accurate bandwidth estimates from CLBE already leads to QoE improvements, the overall QoE is a function of the overall end-to-end ABR streaming pipeline. Since bandwidth estimation by the player is only one (albeit important) component of the pipeline, further QoE gains beyond what we observe may be possible with suitable improvement to other components, e.g., bandwidth prediction and ABR adaptation logic.

We acknowledge that it may be difficult to capture packet timing information at the kernel level on some systems. Another question is how to make such kernel-level timing information available to multiple applications without incurring significant system overhead. In §6.4, we use CLBE as a concrete use case to advocate the need for the OS to expose kernel-level packet information as a service to the application, which can enable accurate bandwidth estimation that is needed by many applications.

2 CROSS-LAYER APPROACH

2.1 High-level Overview

Fig. 1 illustrates two scenarios of packet transmission from a server to a client. Fig. 1(a) shows a *network-constrained* scenario, where

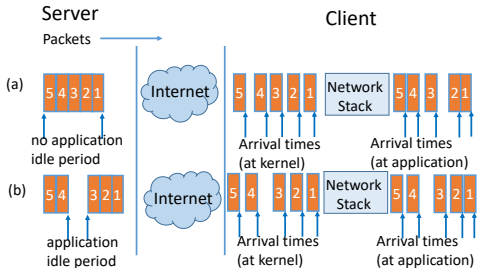


Figure 1: Kernel-level and application-level bandwidth measurement. (a) network-constrained. (b) source-constrained.

the network bandwidth constrains the rate of the data transfer. Specifically, the application-level content is divided into a sequence of back-to-back packets and sent from the server to the client. At the client, we illustrate the arrival times observed at the kernel and application levels. For the arrival times at the kernel level, the interval between two adjacent packets is affected by the available network bandwidth from the server to the client and can be used for bandwidth estimation. Specifically, for a packet of length L, if the time interval of its arrival and the next packet is d, then it provides a sample of the available network bandwidth as L/d. A sequence of n packets can provide up to $n-1$ samples for network bandwidth estimation, which can be further processed (e.g., smoothed, filtered) for more accurate and robust bandwidth estimation. In contrast, the arrival times observed at the application level can be *distorted* by various factors, e.g., buffering, TCP options, and TCP built-in mechanisms. The intermediate APIs (e.g., Fetch API [3]) that are used by the application can further distort the timing information. As a result, using the time interval between two adjacent packets observed at the application level will only provide *distorted* estimates of network bandwidth. On the other hand, a coarse-grained bandwidth estimation can be obtained as nL/T at the application level, where nL is the size of the downloaded content (i.e., n packets, each of length L) and T is the total downloading time.

Fig. 1(b) shows a *source-constrained* scenario, where in addition to network bandwidth, the source (or server) further constrains the data transfer rate. For instance, the data is generated on the fly, and hence there can be an application-level idle period between two sequences of back-to-back packets. In this case, at the client, even at the kernel level, one cannot simply use L/d as a bandwidth estimate since the inter-arrival time, d, can be affected by both the available network bandwidth and application idle period. Similarly, at the application level, one also needs to identify application idle periods and exclude them in bandwidth estimation to avoid underestimating the network bandwidth.

Summarizing the above two scenarios, we argue for a new *cross-layer* approach that combines kernel-level and application-level approaches for bandwidth estimation. (i) While the kernel-level approach by itself can obtain fine-grained bandwidth estimation in the network-constrained scenarios, in the source-constrained scenarios, it is challenging to identify the application idle periods solely based on kernel-level packet capture. (ii) The application-level approach by itself is limited by the distorted signals that are observed at the application layer, which are affected by many factors not related to the available network bandwidth. As a result,

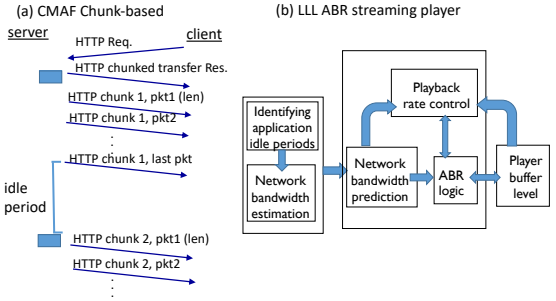


Figure 2: Data transmission patterns of CMAF chunk-based LLL streaming and player diagram.

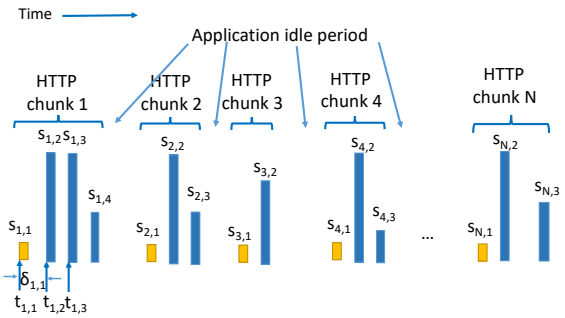


Figure 3: Illustration of the packets in HTTP chunks, which may or may not coincide with CMAF chunks. The small packets in yellow are length-packets and the packets in blue are CMAF data packets. The largest packets are of size MTU.

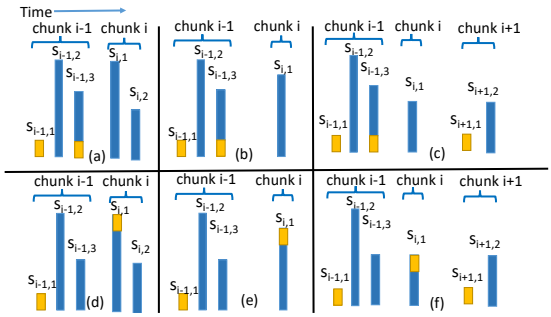


Figure 4: Top row: examples where the length-packet in HTTP chunk i is merged with the last packet in HTTP chunk $i-1$. Bottom row: examples where the length-packet in HTTP chunk i is merged with the next packet in the same chunk.

the bandwidth estimation can be coarse-grained and inaccurate, even if application idle periods are identified successfully. Combining application-level semantics and kernel-level information can provide a more reliable estimation methodology.

2.2 LLL ABR Streaming

As mentioned earlier, to meet the low-latency requirement in LLL streaming, the current approach uses CMAF chunk-based packaging and HTTP CTE. Fig. 2(a) illustrates the data transfer in such scenarios. Whenever a CMAF chunk is encoded, the server transmits the chunk to the client using HTTP CTE, without waiting for the entire segment to be encoded. As a result, there can be an application idle period between two CMAF chunks. In addition, in DASH LLL streaming, there is no application-level information that directly marks the beginning and end of the chunked transfer [33]. If we ignore the application idle period between two consecutive CMAF chunks, the estimated bandwidth will just be equal to the average encoding bitrate of the segment, which can significantly underestimate the network bandwidth. Accurate bandwidth estimation is, however, crucial for LLL streaming. Specifically, as shown in Fig. 2(b), the estimated historical network bandwidth is fed to the network bandwidth prediction module in a player to predict future bandwidths, which will be input to the ABR logic and may further affect the playback rate of the player. Overestimation of network bandwidth can lead to over prediction of future network bandwidth, leading to undesirable stalls and/or lower playback rate for the player to catch up with the target live latency. Conversely, underestimation can lead to lower video quality than what can be supported by the network. In settings with high network bandwidth variability, such as cellular networks, the adverse impact of inaccurate bandwidth estimation can be particularly acute.

Existing bandwidth estimation approaches for LLL streaming work at the application level, and use various heuristics to resolve the above challenges. For instance, the approaches in [10, 40, 62] parse the incoming data and use the CMAF format to identify CMAF boundaries. Other approaches identify the application idle periods as the periods where the estimated network bandwidth is lower than an average value (e.g., as in Shaka player, see §4.1). As we shall show later, due to the distorted signals and coarse-grained information at the application level, these existing approaches do not provide accurate bandwidth estimation.

In contrast to existing approaches, we develop a novel technique, *cross-layer bandwidth estimation (CLBE)*, for LLL streaming. Specifically, CLBE combines the undistorted signals captured at the kernel level with the application semantics for accurate network bandwidth estimation, as detailed below.

3 OUR APPROACH: CLBE

In this section, we describe our cross-layer approach, CLBE, for network bandwidth estimation in LLL streaming. Our description below considers source-constrained scenarios where data transfer uses HTTP CTE, which is the typical case for LLL streaming. If CTE is not used, which can be identified through the HTTP header lines, then the bandwidth estimation can just follow the traditional approach (see §2.1)[1].

3.1 Identifying Application Idle Periods

Consider the transfer of a segment that contains multiple CMAF chunks from the server to the client. As mentioned earlier, each CMAF chunk is transferred to the client using HTTP CTE when it is created, without waiting for the next CMAF chunk to be ready. Therefore, the transfer of a segment contains a number of *HTTP chunks*. Following the HTTP CTE specification [23], each HTTP chunk starts with a small *length-packet* (it is of a few bytes, including a hexadecimal number that represents the number bytes in the HTTP chunk), followed by $k \geq 0$ packets with the size of the

[1] CLBE can obtain accurate network bandwidth estimation for both CTE and non-CTE downloads, as confirmed by our measurement results.

maximum transfer unit (MTU), and ends with a packet with size no more than MTU. This pattern is because the server will push all the data that is ready to be sent to the client as fast as possible to the TCP socket (with the TCP PUSH flag on), and then to the client to satisfy the realtime requirement of LLL streaming. The data is divided into packets with sizes no more than the MTU imposed by the underlying link layer of the network.

Fig. 3 illustrates the above process. It shows N HTTP chunks in one video segment, where the packets belonging to one HTTP chunk are enclosed in a curly bracket in the figure, while the gap between two adjacent HTTP chunks represents an application idle period. In the figure, let $s_{i,j}$ and $t_{i,j}$ denote the size and arrival time of the jth packet in HTTP chunk i, and hence $\delta_{i,j} = t_{i,j+1} - t_{i,j}$ denotes the inter-arrival-time of packets j and $j+1$ for chunk i. Note that an HTTP chunk may contain one or multiple CMAF chunks (e.g., when multiple CMAF chunks are already generated at the server when being requested, they can be sent together in one HTTP chunk). Our bandwidth estimation approach only needs to consider HTTP chunks, and identify the boundary of the HTTP chunks—the time between two adjacent HTTP chunks is an application idle period, which will be excluded in bandwidth estimation. Unlike the approaches in [10, 40, 62], our approach uses HTTP-level information and does not rely on the flags in CMAF packaging (e.g., moof and mdat), and hence is generally applicable to scenarios with HTTP CTE, not limited to CMAF data.

In Fig. 3, for ease of illustration, each length-packet (in yellow) is marked as a standalone packet. In practice, due to various timing conditions, the length-packet in an HTTP chunk may not be standalone. Specifically, the length-packet of the ith HTTP chunk can be merged with the previous data packet, i.e., added to the end of the last packet in the $(i-1)$th HTTP chunk, which can happen when these two HTTP chunks are transferred close together in time. Conversely, the length-packet in an HTTP chunk can be merged with the next data packet in the same chunk. The above two scenarios are illustrated in the top and bottom rows of Fig. 4, respectively.

Identify HTTP chunk boundaries. We next present a method that uses the timing and size information of the packets captured at the kernel level to identify HTTP chunk boundaries. Consider the packets in a segment, where the beginning of a segment is identified as the first packet after an HTTP GET request, and the end of a segment is identified as the one before the next HTTP GET request. All the packets with payload of 0 are ignored. We identify the beginning of the first HTTP chunk in a segment as the one right after the HTTP GET request. Since this HTTP chunk contains the first frame of the segment (i.e., an I-frame), it is typically large, with one or multiple packets of size MTU, followed by one with size no more than MTU. We then wait until we see a packet size increase, i.e., when seeing a packet s_n that is larger than s_{n-1}. We then set s_n to be the beginning of the next chunk; if s_{n-1} is a not length-packet (identified by the length of the packet), we set s_{n-1} to be the end of the previous chunk, otherwise, we set s_{n-2} (if it exists) to be the end of the previous chunk.

As an example, in Fig. 3, we identify $s_{1,2}$ and $s_{1,4}$ as respectively the first and last packets in HTTP chunk 1, $s_{2,2}$ and $s_{2,3}$ as respectively the first and last packets in HTTP chunk 2, $s_{3,2}$ as both the first and last packets in HTTP chunk 3, and so on. (Here we slightly abuse notation and use $s_{i,j}$ to represent both a packet and the size of that packet.) The sequence of identified packets in an HTTP chunk will be used to estimate the network bandwidth in §3.2. Note that in the above, the length-packet in an HTTP chunk is not included in the sequence of identified packets, which is intentional, since small-size length-packets can lead to higher measurement noises in bandwidth estimation.

The above example in Fig. 3 is for the cases where length-packets are standalone packets (which happens to 72%-93% of the HTTP chunks in our experiments). We now consider the special cases where a length-packet is merged with the previous or next data packet, as illustrated in top and bottom rows of Fig. 4, respectively. Fig. 4(a)-(c) have the same setting for chunk $i-1$, and only differ in the setting for chunk i. In Fig. 4(a), both chunks $i-1$ and i are larger than MTU. We see packet size increases from $s_{i-1,1}$ to $s_{i-1,2}$, and then from $s_{i-1,3}$ to $s_{i,1}$, and hence $\{s_{i-1,2}, s_{i-1,3}\}$ is identified as a set of packets in chunk $i-1$, and $s_{i,1}$ is identified as the first packet in chunk i, which is correct. Fig. 4(b) differs from (a) in that chunk i is smaller than MTU. In this case, we still see an increase in packet size from $s_{i-1,3}$ to $s_{i,1}$, and again identify the packets in chunk $i-1$ correctly. Fig. 4(c) differs from (b) in that $s_{i,1} \le s_{i-1,3}$, and hence does not shows an increasing trend. In this case, we find an increasing trend from $s_{i+1,1}$ to $s_{i+1,2}$, and hence $s_{i,1}$ is identified as the last packet in chunk $i-1$, which is incorrect. This incorrect decision, however, will not lead to a large estimation error, since the reason why the length-packet in chunk i is merged with the last packet in chunk $i-1$ is that they are close to each other, and hence the application idle period between chunks $i-1$ and i is very short in this case anyway. In Fig. 4(a)-(c), the size of chunk $i-1$ is larger than MTU; the same results hold when the size of chunk $i-1$ is less than MTU as long as the size relationship between the last packet in chunk $i-1$ and the first packet in chunk i is consistent with those in Fig. 4(a)-(c); we omit the illustrating figures for clarity.

Fig. 4(d)-(f) only differ from Fig. 4(a)-(c) in that they show three cases where the length-packet of HTTP chunk i is merged with the next data packet. Similarly, we see that our method can identify the HTTP chunk boundaries for all but the last case in Fig. 4(f).

Even when the boundary of an HTTP chunk is not identified correctly, the impact will not last for more than a segment, since we run the above method per segment, with the segment boundaries identified using application semantics (i.e., HTTP messages). In addition, since the packet capture is fine-grained and we use the average value of multiple packets for bandwidth estimation, errors from one packet or missing samples do not lead to noticeable impact on the overall estimation error (see §6.1).

Alternative approach. Another approach for identifying HTTP chunk boundaries is determining the length of each HTTP chunk using the corresponding length-packet, and then keeping track of the packet sizes until reaching the HTTP chunk size. We do not use this approach in this paper because it requires parsing the packet content (to determine the individual HTTP chunks), and hence is more computationally intensive than the previous method. In addition, it needs to identify and exclude retransmitted packets (e.g., using the sequence numbers in the TCP packets), which will introduce more overhead. On the other hand, this method can be

potentially more accurate than the previous method. Further study of this approach is left as future work.

3.2 Bandwidth Estimation

Following the approach in §3.1, we identify a set of packets belonging to each HTTP chunk. Suppose k_i packets are identified in chunk i. Recall that $s_{i,j}$ and $t_{i,j}$ denote the size (including the application payload and the packet headers) and arrival time of packet j in chunk i, and $\delta_{ij} = t_{i,j+1} - t_{i,j}$ denotes the inter-arrival time of packet j and $j+1$ in chunk i. We refer to the first $k_i - 1$ packets in chunk i as *valid* packets for bandwidth estimation; the k_ith packet is excluded since it may be followed by an application idle period, which can skew the bandwidth estimation. In the case of $k_i = 1$, no valid packet is in that chunk.

With the above method for identifying valid packets, we now consider a time interval with $N \geq 1$ HTTP chunks (e.g., the chunks in a segment), identify all the valid packets for bandwidth estimation, and then obtain the average network bandwidth estimation for that interval, which is more robust to measurement noises than the estimation from a single packet. Specifically, consider the size and inter-arrival time pairs for all the valid packets in the N HTTP chunks, denoted as $\{(s_{i,1}, \delta_{i,1}), (s_{i,2}, \delta_{i,2}), \ldots, (s_{i,k_i-1}, \delta_{i,k_i-1})\}_{i=1}^{N}$. We use the following two methods to estimate the network bandwidth for the interval. The first method obtains a sample of network bandwidth estimation from each valid packet as $\widehat{c}_{i,j} = s_{i,j}/\delta_{i,j}$, where $i = 1, \ldots, N$, $j = 1, \ldots, k_i - 1$, and then uses their average as the estimated network bandwidth for the interval. The second method estimates the network bandwidth as the total size of the valid packets divided by the corresponding sum of the inter-arrival times. Specifically, these two methods are represented as

$$\frac{\sum_{i=1}^{N}\sum_{j=1}^{k_i-1}\widehat{c}_{i,j}}{\sum_{i=1}^{N}(k_i-1)} \quad \text{and} \quad \frac{\sum_{i=1}^{N}\sum_{j=1}^{k_i-1}s_{i,j}}{\sum_{i=1}^{N}\sum_{j=1}^{k_i-1}\delta_{i,j}}. \tag{1}$$

Considering that small packets can lead to more noises in bandwidth measurements, only the packets with sizes $\geq B$ are used, where B is a threshold value (we use $B = 50$ bytes for the rest of the paper). We find that these two methods lead to similar results; the results in the rest of the paper are obtained using the first method. For network bandwidth traces that change faster than what we use (i.e., faster than one second), one may develop other time-weighted method for more accurate bandwidth estimation.

4 INCORPORATING CLBE IN THREE PLAYERS

We implemented CLBE in three open-source players, `dash.js` (v4.0.0), ExoPlayer (v2.16.1), and Shaka (v3.3.1), that support LLL streaming with DASH. We chose these three players since they are popular state-of-the-art players: `dash.js` is the reference player maintained by the DASH Industry Forum [2]; ExoPlayer has been used by more than 140,000 apps in Google Play Store [26] for the Android platform; and Shaka player has been used by more than 1,600 websites [55]. In addition, these three players differ significantly in their bandwidth estimation and prediction, as well as rate adaptation and playback rate control, and hence represent different points in the design space. Since the focus of this work is on bandwidth estimation, our implementation only modifies the bandwidth estimation module in each player, with no change to the existing prediction algorithm, ABR logic and the playback rate control mechanism.

4.1 Bandwidth Estimation and Prediction

For all three players, we developed a new `CLBE` module at the client that estimates the network bandwidth combining live kernel-level packet capture and application semantics. The estimated network bandwidth values are then transmitted in real time using WebSocket protocol [7] to the bandwidth prediction module of each player. Specifically, we implemented the `CLBE` module on top of two alternative packet capture and processing tools, pyshark [5] and Scapy [6]. Our measurements show that Scapy provides lower and more consistent latency than pyshark. The results henceforth are obtained by building the `CLBE` module on top of Scapy.

`dash.js`. The existing bandwidth estimation in `dash.js` is based on the techniques proposed in (LoL [40] and LoL$^+$ [10]). It includes three steps. (i) It uses the Fetch API to track the download progress of a CMAF chunk, parses the chunk payload to identify the beginning and ending times of each chunk download based on the format of CMAF chunks. (ii) After that, it uses chunk filtering rules to decide whether to store the measurements for a CMAF chunk. (iii) At the end of a segment, it computes the segment throughput based on the chunks that passed the filtering process. The bandwidth prediction in `dash.js` uses a sliding window average of the bandwidth estimation of the past k segments, where k is dynamically determined based on the variability of the past network bandwidth. In addition, it times the average value by a factor (default as 0.9) to obtain a more conservative prediction value.

To incorporate CLBE, we modified ThroughputHistory.js to add a WebSocket to receive the estimated bandwidth values from our `CLBE` module and append these values to the sliding window buffer. We further modified LoLpRule.js to read the latest k bandwidth estimations from the buffer, and feed them to the sliding-window average algorithm to predict the bandwidth for the next segment.

ExoPlayer. In ExoPlayer, bandwidth is estimated on a per segment basis, as the amount of data downloaded divided by the duration of the downloading. To handle application idle periods, it uses a binary flag to indicate whether the network bandwidth is fully utilized or not, and the bandwidth estimate is only updated when this flag is true. For CTE-based transfer, this flag is set to false, causing no update to bandwidth estimation. The bandwidth prediction uses a SlidingPercentile algorithm [21]. It predicts the bandwidth based on a sliding window of past download rate observations, and then can calculate any percentile (the default is 0.5) over a sliding window of weighted values. To reduce the risk of rebuffering, it sets the predicted network bandwidth to be 70% of the predicted value.

We modified DefaultBandwidthMeter.java in ExoPlayer to add a WebSocket to receive the estimated bandwidth values from our `CLBE` module, and then push these values to the buffer for SlidingPercentile algorithm to predict network bandwidth for the next segment, regardless of the value of the binary flag described earlier.

Shaka. In Shaka Player, network bandwidth is estimated at short intervals (tens to hundreds of ms). It uses a discarding mechanism to exclude the application idle periods. Specifically, if the amount of data downloaded in the interval is below a pre-specified threshold (16K bytes), then the network bandwidth estimation for the

interval (i.e., the amount of data downloaded divided by the interval) is discarded. Otherwise, it is used to update the current network bandwidth estimation using Exponential Weighted Moving Average (EWMA). The EWMA-based bandwidth estimation is used directly as the predicted network bandwidth. We modified NetworkingEngine class in Player.js to add a WebSocket to receive the estimated bandwidth values from our `CLBE` module and append the values to the buffer to be used by the EWMA algorithm. Since the EWMA algorithm takes bandwidth estimates at short intervals, the `CLBE` module for Shaka also estimates network bandwidth at short intervals (approximately every 200 ms).

4.2 ABR Logic and Playback Rate Control

Although we did not modify the ABR logic and playback rate control mechanisms in the three players, we briefly describe them below since they affect the final QoE (§6.3). (i) **`dash.js`.** The ABR logic is a learning-based algorithm based on [10, 40]. The playback speed control algorithm determines a playback speed (e.g., 0.7 to 1.3× the normal speed). By default, segment replacement [56] is enabled in `dash.js`. We disabled it for LLL streaming to accommodate the low latency requirement. (ii) **ExoPlayer.** Its ABR logic is essentially a rate-based algorithm that selects the track with the bitrate below the predicted network bandwidth, while taking account of additional factors including buffer level and past track selection. The playback rate control module adjusts the playback rate from 0.97 to 1.03 by default. (iii) **Shaka Player.** Its ABR logic is also rate-based. In addition, Shaka does not incorporate a playback rate control mechanism to adjust the playback rate.

5 EVALUATION SETUP

Our evaluation is trace-driven, using real videos and network bandwidth traces that represent a wide range of network conditions.

5.1 Video and Encoding

We use two videos, Big Buck Bunny (BBB) [1] and a city sightseeing video (NYC), for the evaluation. Both videos are encoded as constant bitrate (CBR) encoding (often used in LLL streaming), using the H.264 codec, at 30 frames per second, segment length of 0.5 sec, and CMAF chunk duration is set to the frame duration (i.e., 33 ms). Both videos are encoded into 5 tracks, with the average encoding bitrate of 0.2, 0.4, 0.7, 1.0, and 2.1 Mbps for BBB, and 0.2, 0.4, 0.7, 1.0, and 2.6 Mbps for NYC. The resolutions of the tracks are 240p, 360p, 480p, 480p, and 720p, respectively. For both videos, the peak to average ratio of all the tracks measured per segment duration (i.e., 0.5 s) is 1.7, while the ratio at a coarser grain of 1 second is around 1.2-1.3 for BBB and 1.4 for NYC. BBB was used in Twitch grand challenge [4] and existing literature [29, 36, 40] with three tracks of bitrate 0.2, 0.6 and 1 Mbps; we use more tracks and a wider range of encoding bitrate to represent more realistic scenarios in practice. For each track, we encode the raw video using FFmpeg [22] with the following configuration settings: `vbvbuffer` set to half the average encoding btrate, zero-latency, no B frames, with ultra-fast preset and high profile to be suitable for LLL streaming.

5.2 Network Bandwidth Profiles

Synthetic traces. These include simple constant-bandwidth profiles, as well as Cascade and Intra-Cascade profiles from the Twitch grand challenge [4]. The Cascade profile is 150 seconds long, with 5 bandwidth values as 1.2, 0.8, 0.4, 0.8, and 1.2 Mbps, each lasting for 30 seconds. The Intra-Cascade profile is 135 seconds long, with 9 bandwidth values in the step of 0.2 Mbps as 1.0, 0.8, 0.6, 0.4, 0.2, 0.4, 0.6, 0.8, and 1.0 Mbps, each lasting for 15 seconds.

Real-world traces. These include one trace of average bandwidth 1.31 Mbps collected between a Twitch player and server every five seconds [10], as well as 10 cellular traces, taken randomly from 40 hours of traces that we collected from two commercial LTE networks (when stationary, walking, or driving). In the cellular traces, the network bandwidth was recorded every second from a well-provisioned server to a phone. Since the bitrate of the video tracks varies from 0.2 to 2.6 Mbps, we scale the bandwidths of these network traces so that the average bandwidth is 1.5 (in the middle of the track bitrates) or 4.0 Mbps (around 1.5× the top-track bitrate) to emulate heavily and lightly loaded network conditions, respectively. The coefficient of variation of the traces varies from 0.27 to 0.85.

5.3 Experimental Setup

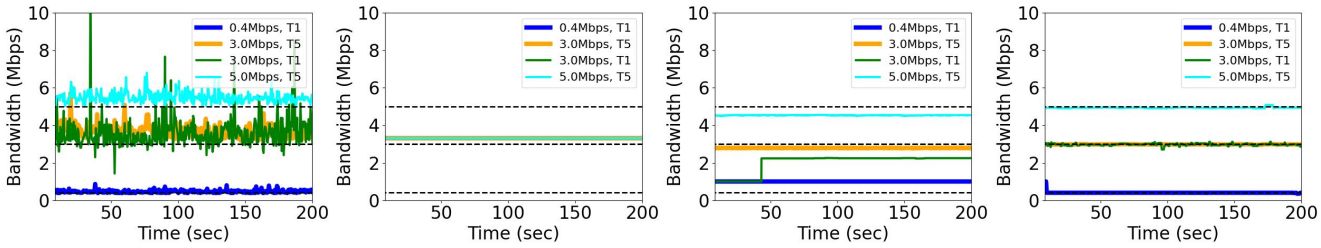
We set up a server on a desktop computer that does live encoding and packaging of the video stream, and sends the encoded content to a DASH live server. To ensure that the encoding has low latency, we follow the setup in [4] that emulates live encoding by pre-encoding the various tracks and sending the pre-encoded video to the DASH server in real time. The client is a laptop computer for `dash.js` and Shaka Player, and an Android phone for ExoPlayer. An intermediate computer is between the server and client, and uses Linux `tc` [41] to emulate the network bandwidth along the server-client path as per the specific bandwidth profile trace. Specifically, we use Hierarchical Token Bucket in `tc` and set two parameters `burst` and `cburst` to maintain the spacing between adjacent packets. The `CLBE` module runs on the client for the modified `dash.js` and Shaka Player; for the modified ExoPlayer, since the client is an Android phone, the `CLBE` module runs on the intermediate computer. We use Network Time Protocol (NTP) to synchronize the time of all the three computers to the server and ensure that the time difference is within 20 ms (shorter than 1 frame duration). At the client, we turn off Generic Receive Offload (GRO) [16] option to allow accurate timing capture at the packet level. Similarly, for the intermediate computer, since it is on the path from the server and client, we turn off GRO for its receiving network interface, and turn off TCP Segmentation Offload (TSO) [15] for its sending network interface.

6 EVALUATION RESULTS

We next present the evaluation results. For both `dash.js` and ExoPlayer, we set the target latency $\Delta = 1$ or 3 seconds; for Shaka, we use its default values for the low latency mode, which tries to make the live latency as small as possible. In the interest of space, we only present the results for the NYC video; the results when using the BBB video are consistent.

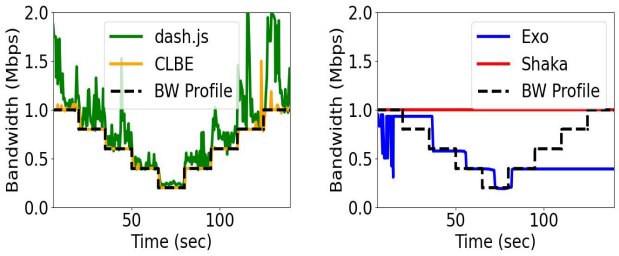
6.1 Bandwidth Estimation Results

Constant bandwidth. This is the simplest type of bandwidth profile. To further simplify the scenario, we configure the server to serve a single video track to the client, so no ABR rate adaptation is involved. We vary the track to explore the impact of track bitrate on bandwidth estimation errors. In addition, for a given track with bitrate b, we set the network bandwidth to be either slightly above



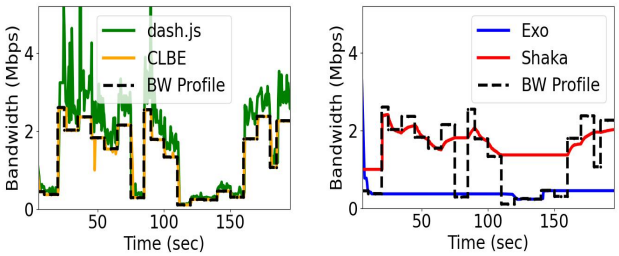
(a) dash.js, Δ = 1 s. (b) ExoPlayer, Δ = 1 s. (c) Shaka, default Δ. (d) CLBE, Δ = 1 s.

Figure 5: Bandwidth estimation for constant bandwidth profiles (represented by dashed black lines).



(a) dash.js and CLBE. (b) ExoPlayer and Shaka.

Figure 6: Bandwidth estimation for Intra-Cascade profile.



(a) dash.js and CLBE. (b) ExoPlayer and Shaka.

Figure 7: Bandwidth estimation for Twitch profile.

the track bitrate (as $b/0.7$) or significantly higher than b. Fig. 5 shows the results for two extreme cases: when using the lowest bitrate track (track T1, 0.2 Mbps) and the network bandwidth is 0.4 or 3 Mbps; and when using the highest bitrate track (track T5, 2.6 Mbps) and the network bandwidth is 3 or 5 Mbps; the results for other tracks show similar trend.

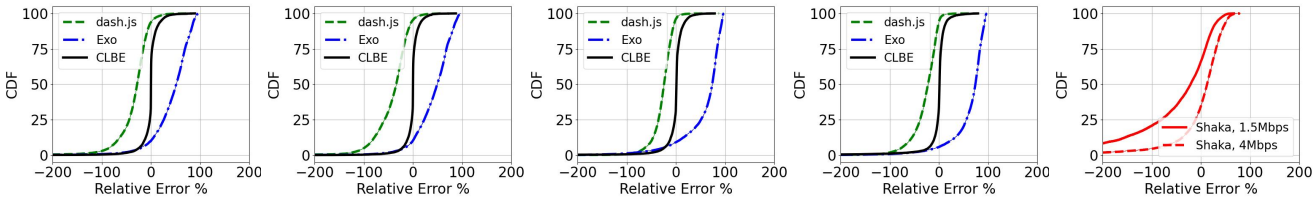
We see that dash.js overestimates the network bandwidth by 15% to 31%. ExoPlayer estimates the network bandwidth as the default value (3.3 Mbps, determined based on the connection type and country code), regardless of the the actual bandwidth. This is because the transmission uses HTTP CTE in most cases, and hence the bandwidth is deemed as not fully utilized and the bandwidth estimation is not updated. Shaka fails to obtain accurate bandwidth estimation under low bandwidth (0.4 Mbps), since the number of bytes downloaded in an interval is below the pre-specified threshold, and hence the estimation remains at the default value of 1 Mbps. When the network bandwidth is 3 Mbps, the bandwidth estimation is more accurate. However, for the low bitrate track (T1), the estimation stays at the default 1 Mbps, and only changes to close to 3 Mbps after around 50 sec. In contrast to these methods, CLBE leads to accurate bandwidth estimation in all the cases, demonstrating the advantage of our cross-layer approach.

Slow- and medium-varying traces. Fig. 6 shows the bandwidth estimation of the various approaches under the Intra-Cascade profile, with $\Delta = 1$ s, except for Shaka. While dash.js keeps track of the changes in the bandwidth profiles, there is significant overestimation at places. In contrast, CLBE closely tracks the changes in the network bandwidth, with low estimation errors. ExoPlayer keeps track of the bandwidth decrease at the beginning of the bandwidth profile, where ExoPlayer falls behind the target latency, leading to standard (instead of CTE) HTTP downloading. When the network bandwidth increases later on, the bandwidth is not fully utilized, and hence ExoPlayer does not update its bandwidth estimation, causing underestimation. Shaka has an almost constant bandwidth estimation, since the network bandwidth is low and the threshold for sufficient data downloading is not met.

Fig. 7 shows the results under the Twitch bandwidth profile. We again see that dash.js overestimates the bandwidth, while CLBE obtains accurate estimation. ExoPlayer leads to significant underestimation most of the time due to lack of bandwidth updates. Shaka keeps track of the bandwidth profile changes when the network bandwidth is high, while has significant overestimation when the network bandwidth is low (from 100 to 150 seconds).

We obtain relative estimation error as $(b_i - \widehat{b_i})/b_i$ for segment i, where b_i and $\widehat{b_i}$ are the ground-truth and estimated bandwidth for the duration of downloading segment i. For Cascade, Intra-Cascade and Twitch profiles, 97%-99% of the relative estimation errors obtained by CLBE are within ±10%. In contrast, only 5%-29% of the relative estimation errors by dash.js are within ±10%.

Fast-varying cellular traces. The first four subplots in Fig. 8 show the cumulative distribution function (CDFs) of the bandwidth estimation error of CLBE and those in the original dash.js and ExoPlayer, when the average network bandwidth is 1.5 or 4 Mbps, and the target latency $\Delta = 1$ or 3s. The last subplot in Fig. 8 is for the original Shaka Player, which does not allow explicit specification of target latency. Consistent with our earlier observations, dash.js tends to overestimate, ExoPlayer tends to underestimate, while Shaka has more accurate bandwidth estimation when the network bandwidth is high than when it is low. Although the bandwidth estimation in dash.js is significantly more accurate than that in ExoPlayer and Shaka, it still substantially lags behind CLBE. For CLBE, 74%-78% of its bandwidth estimation errors are within ±10% and 88%-91% of its errors are within ±20%, while for dash.js, only 10%-17% of the errors are within ±10% and 24%-46% of the errors



(a) Avg. bw 1.5 Mbps, Δ = 1 s. (b) Avg. bw 1.5 Mbps, Δ = 3 s. (c) Avg. bw 4 Mbps, Δ = 1 s. (d) Avg. bw 4 Mbps, Δ = 3 s. (e) Shaka, default target latency.

Figure 8: Bandwidth estimation errors for 10 cellular traces.

are within ±20%. Overall, CLBE achieves high accuracy even under highly dynamic cellular network conditions.

6.2 Bandwidth Prediction Results

We compare the accuracy of network bandwidth prediction in the original players and the modified players when they incorporate CLBE. The relative prediction error is calculated per segment, obtained as the actual bandwidth subtracted by the predicted bandwidth, and divided by the actual bandwidth.

In the following, we only present the bandwidth prediction results under the highly dynamic cellular traces, where accurate prediction is significantly more challenging than for other scenarios. The first four subplots in Fig. 9 compare the prediction errors in the original `dash.js` and ExoPlayer and those in the modified players with CLBE (we only show the results when running CLBE in the modified `dash.js` player; the results for the other two modified players are similar). The last subplot in Fig. 9 shows the prediction errors in the original Shaka Player. We see that the original `dash.js` player tends to overpredict and the original ExoPlayer tends to underpredict the bandwidth. The original Shaka player also has large prediction errors, particularly when the network bandwidth is low. The CLBE-based prediction is significantly better than that in the original players. Specifically, in the various settings, 70%-73% of the CLBE-based prediction errors are within ±20%, while even for the original `dash.js`, which provides more accurate prediction than the original ExoPlayer and Shaka Player, the accuracy is much lower (only 24%-46% of the errors are within ±20%).

6.3 QoE Results

We compare the QoE of the original and modified players. Specifically, we use the following commonly used objective metrics to measure QoE: (i) *quality of played back segments*, measured using a state-of-the-art perceptual quality metric, VMAF [38]. Since humans are sensitive to low-quality segments, we particularly quantify the percentage of low-quality segments, i.e., the segments with VMAF below 60 (considered as fair and below-fair quality). (ii) *rebuffering duration*, measured as the total amount of rebuffering/stall in a video session. (iii) *live latency*, i.e., the latency relative to the live edge for each segment, (iv) *playback rate*: we measure the percentage of the segments with playback rate deviating from the normal speed, and for them, the average deviation from the normal speed.

Under the synthetic traces, for all three players, the modified player with CLBE leads to track selection that matches the bandwidth profile much better than the original player. Fig. 10 shows an example for the Cascade profile. The top plot is the bandwidth profile. The three lower plots are track selection for the three players; each plot shows the track selections by the original and modified players. We next focus on the results under real-world network bandwidth profiles, i.e., the Twitch and cellular network profiles. Table 1 shows the various QoE metrics for the three players, where in each cell, the results for both the original and modified players are shown side-by-side.

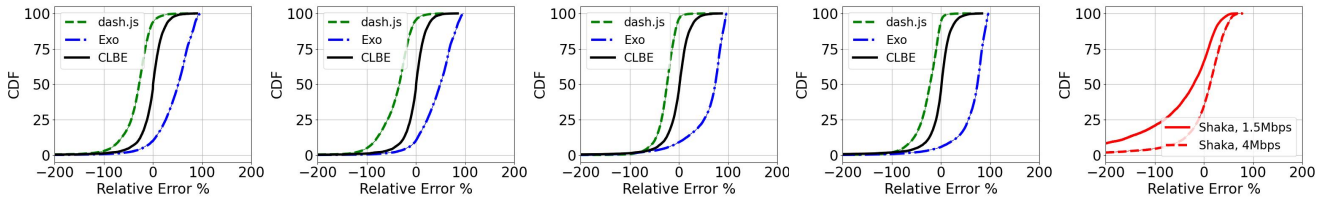
QoE for `dash.js`. For both the original and modified players, for the same network bandwidth profile, allowing a larger target latency leads to higher quality and lower stall duration. For the Twitch profile, when target latency Δ = 1 s, the modified player has much less low-quality segments compared to the original player (64% vs 86%), with slightly higher stall duration; when Δ = 3 s, the performance of the two players is similar. For both Δ values, the 75th percentile of the live latency of the two players is close to the target latency (not shown in the table). Both players have moderate amount of deviation from the normal playback rate.

For the fast varying cellular network profiles (bandwidth varying per second), compared to the original player, the modified player achieves higher quality with similar stall duration in all of the four bandwidth and target latency settings except for one setting (i.e., the average bandwidth is 1.5 Mbps and Δ = 1 s) where their performance is similar. The quality improvement is the most significant when the average bandwidth is 4 Mbps and Δ = 3 s (the percentage of low-quality segments is 12% lower, 21% vs 33%), with no stalls.

To gain further insights, we investigate the performance of an "oracle" player, which knows the ground-truth future network bandwidth when making track decisions. Specifically, when making a track decision at time t for the next segment, it knows the bandwidth for $[t, t+0.5]$, where 0.5 s is the segment duration. Figures 11 and 12 plot the VMAF and stall duration for the three player variants under the 10 cellular network traces. We see that the oracle player leads to visibly better quality than CLBE in one setting (average bandwidth 4 Mbps, Δ = 1 s) and slightly better quality in another setting, with lower or compatible stall duration. When the average bandwidth is 1.5 Mbps and Δ = 1 s, even the oracle player has significant rebuffering in two traces due to sudden bandwidth drops (even choosing the lowest track still causes rebuffering). When the average bandwidth is 1.5 Mbps and Δ = 3 s, the oracle player has slight rebuffering (0.45 s), again due to sudden drop in bandwidth.

Overall, our results show CLBE leads to benefits in QoE. The gap between CLBE-based player and the oracle player also indicates the room of further improvement by combining CLBE with more accurate bandwidth prediction techniques.

QoE for ExoPlayer. As shown in Table 1, for the Twitch bandwidth profile, the modified player achieves significantly better quality (the percentage of low-quality segments is 48% lower than that of the original player for both Δ = 1 and 3 s), at the cost of larger



(a) Avg. bw 1.5 Mbps, Δ = 1 s. (b) Avg. bw 1.5 Mbps, Δ = 3 s. (c) Avg. bw 4 Mbps, Δ = 1 s. (d) Avg. bw 4 Mbps, Δ = 3 s. (e) Shaka, default target delay.

Figure 9: Bandwidth prediction error for 10 cellular traces.

Table 1: QoE results. The two numbers in each cell represent the results from the original player and the modified player with CLBE (in green).

	Avg. network bw. (Mbps)	Target latency (s)	Low-qual. segs (≤ 60) (%)	Stall duration (s)	Live latency (90%ile) (s)	Playback rate deviation (%)	Avg. playback rate deviation
dash.js	**Twitch**	**1**	86, 64	2.5, 3.6	5.8, 5.7	31.1, 26.5	0.24, 0.26
	Twitch	**3**	54, 53	0.0, 0.0	7.6, 8.2	30.4, 26.8	0.28, 0.33
	Cellular (1.5)	**1**	78, 72	1.1, 1.0	1.0, 1.0	6.9, 6.3	0.17, 0.17
	Cellular (1.5)	**3**	69, 68	0.0, 0.0	3.1, 3.1	3.4, 1.4	0.18, 0.3
	Cellular (4.0)	**1**	49, 43	0.2, 0.3	1.0, 1.0	2.2, 2.2	0.14, 0.16
	Cellular (4.0)	**3**	33, 21	0.0, 0.0	3.1, 3.1	1.1, 0.4	0.2, 0.26
ExoPlayer	**Twitch**	**1**	99, 51	8.6, 13.9	9.6, 14.0	100.0, 88.9	0.03, 0.03
	Twitch	**3**	99, 51	2.7, 0.0	2.0, 13.2	19.6, 0.0	0.02, 0.03
	Cellular (1.5)	**1**	93, 76	1.7, 3.0	2.7, 3.9	33.9, 47.5	0.02, 0.02
	Cellular (1.5)	**3**	93, 76	1.6, 1.6	2.5, 2.0	36.1, 33.9	0.02, 0.01
	Cellular (4.0)	**1**	76, 57	0.7, 0.7	1.8, 1.5	18.4, 17.6	0.01, 0.01
	Cellular (4.0)	**3**	76, 54	0.7, 0.9	1.6, 1.6	19.7, 20.2	0.01, 0.01
Shaka	**Twitch**	**Default**	11, 44	28.4, 8.9	42.1, 16.3	0.0, 0.0	N/A, N/A
	Cellular (1.5)	**Default**	41, 52	36.4, 3.2	79.5, 8.5	0.0, 0.0	N/A, N/A
	Cellular (4.0)	**Default**	19, 16	3.7, 2.3	21.3, 7.2	0.0, 0.0	N/A, N/A

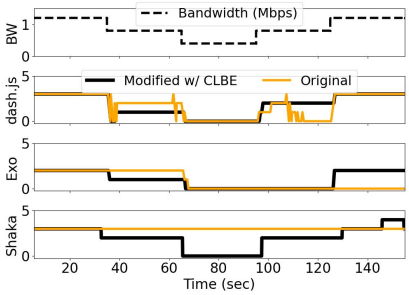


Figure 10: Selected tracks in the original and modified players for Cascade profile.

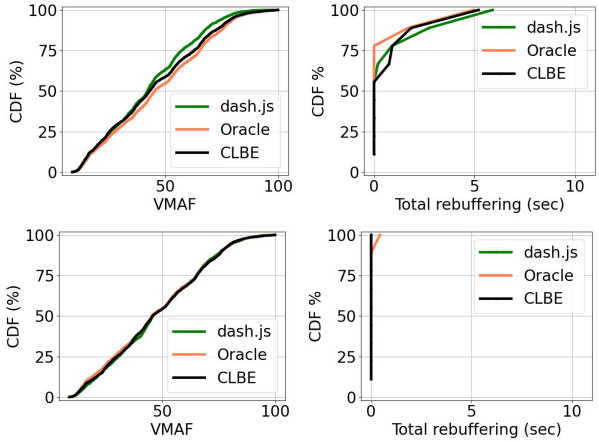


Figure 11: QoE of dash.js, cellular network with average bandwidth 1.5 Mbps, Δ = 1 and 3 s in the top and bottom rows, respectively.

rebuffering and/or live latency. For the cellular network traces, the modified player leads to 14% to 20% less low-quality segments across the four bandwidth and target latency settings, with comparable or larger stall duration and generally lower live latency. The percentage of playback rate deviation is high for both the original and modified players in several settings. On the other hand, the extent of deviation is low due to the tight rate limit (0.97 to 1.03) in ExoPlayer. We also observe that ExoPlayer does not take advantage of the larger target latency—the live latency when Δ = 3 s is only 1.3 or 1.4 s, significantly lower than the target.

QoE for Shaka. Shaka does not use playback rate control and does not catch up with the live edge even when the live latency is large. Table 1 shows that the original player has high live latency, particularly when the network bandwidth is low, while the modified player has significantly lower live latency. Specifically, in one setting, the 90th percentile live latency of the modified player is close to 10× lower than that of the original player. In addition, the modified player has 1.4 to 33.2 seconds less stall than the original player, at the cost of more low-quality segments under low bandwidth settings.

6.4 Summary and Discussion

We have shown that, compared to state-of-the-art application-level approaches, CLBE leads to significantly more accurate bandwidth

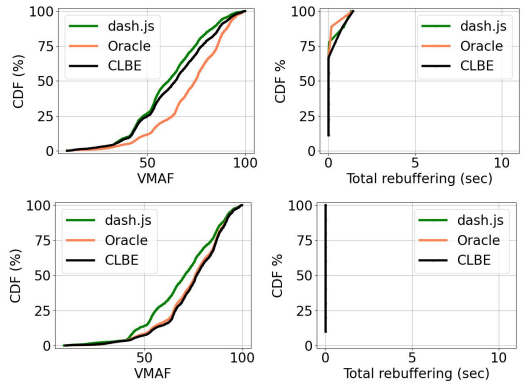


Figure 12: QoE for `dash.js`, cellular network with average bandwidth 4.0 Mbps, $\Delta = 1$ and 3 s in the top and bottom rows, respectively.

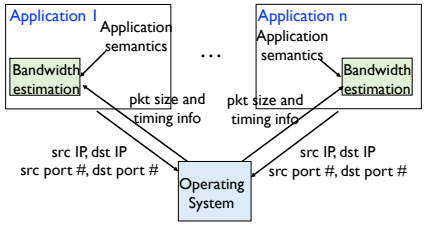


Figure 13: OS provides kernel-level packet size and timing information to applications as a service.

estimation and prediction, which can lead to improved QoE. Note that since the overall QoE is a function of the overall end-end ABR streaming pipeline (including bandwidth estimation, prediction and ABR logic), further QoE improvements beyond what we observe in our evaluations may be possible if these other important components were also suitably modified.

To support CLBE and other applications that need kernel-level packet capture, we advocate that the OS provides an API so that applications can obtain clean timing signals directly. Fig. 13 illustrates one way that the OS can provide such a service. An application sends a 4-tuple (source and destination IP addresses and port numbers) to the OS. The OS then returns a sequence of relevant kernel-level packet timing and size information to the application. Such an API eliminates the need for individual applications to develop kernel-level packet capture functionality. With such an API, we can directly incorporate CLBE in each player, without the need for either live packet capturing or using WebSocket to send the bandwidth estimation to each player. In addition, there would be no need to turn off aggregation mechanisms (GRO and TSO, see §5.3) at the client. The NEAT socket API [19, 61] and Socket Intent framework [20, 53] may be helpful in realizing the above API.

7 RELATED WORK

LLL ABR streaming. The work in [43] highlights the benefits of chunk-based CMAF for LLL streaming compared to the traditional segment-based approach. The first work that studies network bandwidth estimation with CMAF and HTTP CTE is ACTE [12, 13], which was improved in LoL [40] and LoL$^+$ [10]. The studies [62] and [47] further improve the bandwidth estimation in LoL and LoL$^+$ using more refined filtering rules. The above bandwidth estimation techniques were adopted in the `dash.js` player that we use in this paper. These techniques are at the application-level. Our work differs from them in that we use a cross-layer approach.

Other aspects of LLL ABR streaming includes bandwidth prediction, rate adaptation and playback rate control, which have been studied in ACTE [12, 13], LoL and LoL$^+$ [10, 40], QLive [62], TightRope [58], and the study in [47]. The study [60] presents an optimized delivery architecture for LLL streaming. LLL streaming is also studied in [45, 49], which however are not for CMAF-based content or use HTTP CTE. The studies in [31, 42, 46, 59, 63, 66] present bandwidth prediction techniques for ABR streaming, not targeting LLL streaming specifically. A recent study [11] presents a data-driven approach for bandwidth prediction, with a focus on LLL streaming. Our work focuses on accurate bandwidth estimation, which is important for later steps of bandwidth prediction and rate adaptation. We also show that due to the complex interplay between bandwidth prediction and ABR logic, even oracle bandwidth prediction sometimes does not provide better QoE, highlighting the importance of considering both components jointly.

Non-LLL streaming based rate adaptation has been studied extensively [9, 25, 30, 32, 34, 35, 39, 44, 51, 52, 57, 59, 63, 65], leading to buffer-based, rate-based, learning-based, or hybrid schemes. These schemes are not designed specifically for LLL streaming that has only up to a few seconds of target latency. Our study focuses on cross-layer bandwidth estimation and its benefits on LLL streaming. We use the existing rate adaptation techniques in three popular players, including both learning-based and rate-based techniques.

Cross-layer approach for streaming. Salsify [24] presents a cross-layer approach that combines the transport protocol's congestion control with the video codec's rate control into one algorithm for real-time streaming. It builds on UDP-based WebRTC, very different from our focus of LLL ABR streaming, where congestion control is by the underlying transport protocol (e.g., TCP) directly. In addition, Salsify is not for bandwidth estimation as in our study. The studies in [14, 37, 48, 54] present cross-layer approaches for streaming, which are not for bandwidth estimation.

Packet-level bandwidth estimation. Many techniques have been proposed for packet-level bandwidth estimation (see [18, 50] and the references within). These techniques typically use small probing packets to infer network bandwidth. Our study differs from them in that we use the TCP packets in the application and a cross-layer approach for bandwidth estimation for LLL streaming.

8 CONCLUSIONS

We proposed a cross-layer approach, CLBE, for network bandwidth estimation in LLL streaming, and showed that it provides significantly more accurate bandwidth estimation and prediction than the state-of-the-art application-level approaches, which leads to better QoE. Our work points to the importance of combining application semantics with packet-level timing information for accurate bandwidth estimation.

ACKNOWLEDGEMENT

We thank the anonymous reviewers who gave valuable feedback to improve this work. We also thank our shepherd, Thomas Zinner, for his insightful suggestions and guiding us through the revisions.

REFERENCES

[1] Big buck bunny. https://peach.blender.org/download/.
[2] DASH Industry Forum. https://dashif.org/.
[3] Fetch API. https://tinyurl.com/5ksttkas.
[4] Grand Challenge on Adaptation Algorithms for Near-Second Latency. https://2020.acmmmsys.org/lll_challenge.php.
[5] pyshark. https://github.com/KimiNewt/pyshark.
[6] Scapy. https://github.com/secdev/scapy.
[7] The WebSocket protocol. https://datatracker.ietf.org/doc/html/rfc6455.
[8] Information technology–Multimedia application format (MPEG-A)–Part19: Common media application format (CMAF) for segmented media. Standard ISO/IEC 23000-19:2018, International Organization for Standardization and International Electrotechnical Commission. https://www.iso.org/standard/71975.html, 2018.
[9] Z. Akhtar, Y. S. Nam, R. Govindan, S. Rao, J. Chen, E. Katz-Bassett, B. Ribeiro, J. Zhan, and H. Zhang. Oboe: Auto-tuning video abr algorithms to network conditions. In *SIGCOMM*, 2018.
[10] A. Bentaleb, M. N. Akcay, M. Lim, A. C. Begen, and R. Zimmermann. Catching the Moment with LoL+ in Twitch-Like Low-Latency Live Streaming Platforms. *IEEE Transactions on Multimedia*, 2021.
[11] A. Bentaleb, A. C. Begen, S. Harous, and R. Zimmermann. Data-driven bandwidth prediction models and automated model selection for low latency. *IEEE Transactions on Multimedia*, 23, 2020.
[12] A. Bentaleb, C. Timmerer, A. C. Begen, and R. Zimmermann. Bandwidth prediction in low-latency chunked streaming. In *Proc. of ACM NOSSDAV*, 2019.
[13] A. Bentaleb, C. Timmerer, A. C. Begen, and R. Zimmermann. Performance analysis of ACTE: a bandwidth prediction method for low latency chunked streaming. *ACM TOMM*, 2020.
[14] Y. Cho, C.-C. J. Kuo, R. Huang, and C. Lima. Cross-layer design for wireless video streaming. In *Proc. of IEEE GLOBECOM*, 2010.
[15] G. W. Connery, W. P. Sherer, G. Jaszewski, and J. S. Binder. Offload of TCP segmentation to a smart adapter, August 1999. US Patent 5,937,169.
[16] J. Corbet. JLS2009: Generic receive offload. https://lwn.net/Articles/358910/,2009.
[17] DASH Industry Forum. dash.js. https://goo.gl/XJcciV.
[18] C. Dovrolis, P. Ramanathan, and D. Moore. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions on Networking*, 12(6), 2004.
[19] T. Dreibholz. NEAT Sockets API. https://datatracker.ietf.org/doc/draft-dreibholz-taps-neat-socketapi/.
[20] T. Enghardt, T. Zinner, and A. Feldmann. Using informed access network selection to improve HTTP adaptive streaming performance. In *ACM MMSys*, 2020.
[21] ExoPlayer. Sliding percentile. https://goo.gl/FFtVr8, 2016.
[22] FFmpeg. FFmpeg Project. https://www.ffmpeg.org/, 2017.
[23] R. Fielding and J. Reschke. Hypertext Transfer Protocol – HTTP/1.1, RFC 7230. https://datatracker.ietf.org/doc/html/rfc7230, 2014.
[24] S. Fouladi, J. Emmons, E. Orbay, C. Wu, R. S. Wahby, and K. Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *Proc. of Networked Systems Design & Implementation (NSDI)*, 2018.
[25] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-Scale Control Plane for Video Quality Optimization. In *Proc. USENIX NSDI*, 2015.
[26] Google. ExoPlayer: Adaptive video streaming on Android - YouTube. https://tinyurl.com/58j8n3yb, 2014.
[27] Google. ExoPlayer. https://github.com/google/ExoPlayer, 2016.
[28] Google. Shaka Player. https://github.com/google/shaka-player, 2019.
[29] C. Gutterman, B. Fridman, T. Gilliland, Y. Hu, and G. Zussman. STALLION: video adaptation algorithm for low-latency video streaming. In *Proc. of ACM MMSys*, 2020.
[30] T. Hoßfeld, C. S. Michael Seufert, T. Zinner, and P. Tran-Gia. Identifying QoE optimal adaptation of HTTP adaptive streaming based on subjective studies. *Computer Networks*, 81, April 2015.
[31] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari. Confused, timid, and unstable: picking a video streaming rate is hard. In *ACM IMC*, 2012.
[32] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proc. of ACM SIGCOMM*, 2014.
[33] International Organization for Standardization. ISO/IEC DIS 23009-1.2 Dynamic adaptive streaming over HTTP (DASH), 2012.
[34] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang. CFA: A Practical Prediction System for Video QoE Optimization. In *Proc. USENIX NSDI*, 2016.
[35] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE. In *CoNEXT*, 2012.
[36] T. Karagkioules, R. Mekuria, D. Griffioen, and A. Wagenaar. Online learning for low-latency adaptive streaming. In *Proc. of ACM MMSys*, 2020.
[37] J.-L. Kuo, C.-H. Shih, C.-Y. Ho, and Y.-C. Chen. A cross-layer approach for real-time multimedia streaming on wireless peer-to-peer ad hoc network. *Elsevier Ad Hoc Networks*, 11(1), 2013.
[38] Z. Li, A. Aaron, I. Katsavounidis, A. Moorthy, and M. Manohara. Toward A Practical Perceptual Video Quality Metric. https://goo.gl/ptjrWv., 2016.
[39] Z. Li, A. Begen, J. Gahm, Y. Shan, B. Osler, and D. Oran. Streaming video over HTTP with consistent quality. In *ACM MMSys*, 2014.
[40] 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. of ACM MMSys*, 2020.
[41] Linux. tc. https://linux.die.net/man/8/tc, 2014.
[42] G. Lv, Q. Wu, W. Wang, Z. Li, and G. Xie. Lumos: towards Better Video Streaming QoE through Accurate Throughput Prediction. In *IEEE INFOCOM*, 2022.
[43] T. Lyko, M. Broadbent, N. Race, M. Nilsson, P. Farrow, and S. Appleby. Evaluation of CMAF in live streaming scenarios. In *Proc. of ACM NOSSDAV*, 2019.
[44] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with Pensieve. In *Proc. of ACM SIGCOMM*, 2017.
[45] K. Miller, A.-K. Al-Tamimi, and A. Wolisz. QoE-based low-delay live streaming using throughput predictions. *ACM TOMM*, 13(1), 2016.
[46] Y. S. Nam, J. Gao, C. Bothra, E. Ghabashneh, S. Rao, B. Ribeiro, J. Zhan, and H. Zhang. Xatu: Richer neural network based prediction for video streaming. In *Proc. of ACM SIGMETRICS*, 2022.
[47] I. M. Ozcelik and C. Ersoy. Low-Latency Live Streaming Over HTTP in Bandwidth-Limited Networks. *IEEE Communication Letters*, 25(2), 2021.
[48] H. B. Pasandi, T. Nadeem, H. Amirpour, and C. Timmerer. A cross-layer approach for supporting real-time multi-user video streaming over WLANs. In *Proc. of ACM MobiCom*, 2021.
[49] H. Peng, Y. Zhang, Y. Yang, and J. Yan. A hybrid control scheme for adaptive live streaming. In *ACM Multimedia*, 2019.
[50] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy. Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE Network*, 17(6), 2003.
[51] Y. Qin, S. Hao, K. R. Pattipati, F. Qian, S. Sen, B. Wang, and C. Yue. ABR streaming of VBR-encoded videos: characterization, challenges, and solutions. In *CoNext*. ACM, 2018.
[52] Y. Qin, R. Jin, S. Hao, K. R. Pattipati, F. Qian, S. Sen, B. Wang, and C. Yue. A control theoretic approach to ABR video streaming: A fresh look at PID-based rate adaptation. In *INFOCOM*, 2017.
[53] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann. Socket Intents: leveraging application awareness for multi-access connectivity. In *Proc. of ACM CoNext*, 2013.
[54] E. Setton, T. Yoo, X. Zhu, A. Goldsmith, and B. Girod. Cross-layer design of ad hoc networks for real-time video streaming. *IEEE Wireless Communications*, 12(4), 2005.
[55] SimilarTech Ltd. Facebook Video vs Shaka Player. urlhttps://www.similartech.com/compare/facebook-video-vs-shaka-player, 2019.
[56] K. Spiteri, R. Sitaraman, and D. Sparacio. From theory to practice: Improving bitrate adaptation in the dash reference player. In *MMSys*, 2018.
[57] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman. BOLA: near-optimal bitrate adaptation for online videos. In *INFOCOM*. IEEE, 2016.
[58] L. Sun, T. Zong, S. Wang, Y. Liu, and Y. Wang. Tightrope walking in low-latency live streaming: Optimal joint adaptation of video rate and playback speed. In *Proc. of ACM MMSys*, 2021.
[59] 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 *Proc. of ACM SIGCOMM*, 2016.
[60] F. Tashtarian, A. Bentaleb, A. Erfanian, H. Hellwagner, C. Timmerer, and R. Zimmermann. HxL3: Optimized Delivery Architecture for HTTP Low-Latency Live Streaming. *IEEE Transactions on Multimedia*, 2022.
[61] F. Weinrank, K. Grinnemo, Z. Bozakov, A. Brunström, T. Dreibholz, P. Hurtig, N. Khademi, and M. Tüxen. A NEAT Way to Browse the Web. In *Proc. of the ACM, IRTF and ISOC Applied Networking Research Workshop (ANRW)*, July 2017.
[62] P. K. Yadav, A. Bentaleb, M. Lim, J. Huang, W. T. Ooi, and R. Zimmermann. Playing chunk-transferred DASH segments at low latency with QLive. In *Proc. of ACM MMSys*, 2021.
[63] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein. Learning in situ: a randomized experiment in video streaming. In *Proc. USENIX NSDI*, 2020.
[64] G. Yi, D. Yang, A. Bentaleb, W. Li, Y. Li, K. Zheng, J. Liu, W. T. Ooi, and Y. Cui. The ACM Multimedia 2019 Live Video Streaming Grand Challenge. In *ACM Multimedia*, 2019.
[65] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *SIGCOMM*. ACM, 2015.
[66] X. K. Zou, J. Erman, V. Gopalakrishnan, E. Halepovic, R. Jana, X. Jin, J. Rexford, and R. K. Sinha. Can accurate predictions improve video streaming in cellular networks? In *HotMobile*, 2015.