BUFFEST: Predicting Buffer Conditions and Real-time Requirements of HTTP(S) Adaptive Streaming Clients

Vengatanathan Krishnamoorthi, Niklas Carlsson, Emir Halepovic and Eric Petajan

BUFFEST: Predicting Buffer Conditions and Real-time Requirements of HTTP(S) Adaptive Streaming Clients

Vengatanathan Krishnamoorthi
Linköping University, Sweden

Niklas Carlsson
Linköping University, Sweden

Emir Halepovic
AT&T Labs, USA

Eric Petajan
AT&T Labs, USA

ABSTRACT

Stalls during video playback are perhaps the most important indicator of a client's viewing experience. To provide the best possible service, a proactive network operator may therefore want to know the buffer conditions of streaming clients and use this information to help avoid stalls due to empty buffers. However, estimation of clients' buffer conditions is complicated by most streaming services being rate-adaptive, and many of them also encrypted. Rate adaptation reduces the correlation between network throughput and client buffer conditions. Usage of HTTPS prevents operators from observing information related to video chunk requests, such as indications of rate adaptation or other HTTP-level information.

This paper presents BUFFEST, a novel classification framework that can be used to classify and predict streaming clients' buffer conditions from both HTTP and HTTPS traffic. To illustrate the tradeoffs between prediction accuracy and the available information used by classifiers, we design and evaluate classifiers of different complexity. At the core of BUFFEST is an event-based buffer emulator module for detailed analysis of clients' buffer levels throughout a streaming session, as well as for automated training and evaluation of online packet-level classifiers. We then present example results using simple threshold-based classifiers and machine learning classifiers that only use TCP/IP packet-level information. Our results are encouraging and show that BUFFEST can distinguish streaming clients with low buffer conditions from clients with significant buffer margin during a session even when HTTPS is used.

CCS CONCEPTS

• **Information systems → Multimedia streaming**; • **Networks → Application layer protocols**;

KEYWORDS

HTTP-based adaptive streaming, HTTPS, Real-time requirements, Buffer condition estimation

MMSys'17, June 20-23, 2017, Taipei, Taiwan
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5002-0/17/06…$15.00
https://doi.org/10.1145/3083187.3083193

1 INTRODUCTION

To properly provision their networks and provide clients with the best possible service, operators need to understand the characteristics of the application traffic mix and how the users' Quality of Experience (QoE) may vary as data flows compete for bandwidth. The QoE can vary significantly as networks go through different utilization phases (e.g., due to diurnal traffic cycles), especially in constrained networks such as the wireless last mile. To provide users with high QoE when operating at moderate to high utilization, it is therefore important to understand user experience and real-time requirements associated with different network flows.

A new type of flow classification: In the past, various flow classification techniques have been applied that map flows[1] to the underlying services they provide. For example, by classifying flows into categories such as real-time streaming and peer-to-peer downloads, network providers have been able to prioritize real-time streaming services at times when the more elastic demands of peer-to-peer networks have used up much of the bandwidth [7, 18, 25]. These techniques are well explored. However, since video streaming is responsible for the majority of today's network traffic [2], classifying all video flows into a single class (without further differentiation within this class) would not help much.

Ideally, video flows should instead be continually and individually (re)classified based on their clients' current buffer conditions. Streaming clients often have highly heterogeneous real-time requirements, and these requirements typically change over the duration of a playback session. For example, streaming clients that have built up a large playback buffer may be highly tolerant to delays in receiving video data (e.g., compared to web clients that often expect immediate loading of websites), while clients with drained buffers may have tighter real-time requirements, in that they need additional video data sooner to avoid stalls (due to empty buffer events). In addition, the real-time requirements of a client may quickly change from critical to low priority, as the buffer builds up again. The importance of differentiating between these clients becomes particularly clear when considering that stalls (and their duration) is the factor that has the largest impact on clients' QoE [16, 27].

Problem formulation: This paper considers the problem of classifying video streaming flows based on the clients' current buffer conditions (i.e., their current real-time requirements). This is a challenging problem, which is further complicated by high usage of HTTPS combined with rate adaptation in almost all popular streaming services. First, with HTTP-based Adaptive Streaming (HAS), each video quality encoding is typically split into smaller

[1]A flow is typically defined as a sequence of a packets between a source IP-port pair and a destination IP-port pair.

chunks that can be independently downloaded and played. The use of multiple encodings allows efficient quality adaptation to be implemented on the clients. This helps clients adapt to network conditions and reduce the number of playback stalls, but also decreases correlation between packet-level throughput and buffer conditions (compared to players that do not perform quality adaptation).

Second, increasingly many video streaming services, including YouTube and Netflix, deliver all or most of their content using HTTPS. Usage of HTTPS prevents operators from observing HTTP requests for video chunks and associated metadata [9], restricting classifiers to TCP/IP packet-level information. Combined with the lack of correlation between packet-level throughput and buffer conditions observed for HAS clients, this restriction significantly complicates in-network estimation of clients' buffer conditions. As argued later in the paper, this challenge is further augmented in services such as YouTube, where different numbers of chunks may be requested simultaneously (e.g., using a single range request).

Contributions: Motivated by the need for real-time requirement classification based on streaming clients' current buffer conditions, we present a novel classification framework called BUFFEST [2] that can be used on both HTTP and HTTPS traffic, as well as validation and performance results for two classes of classifiers.

The primary technical contribution is the BUFFEST framework for estimating and predicting the buffer conditions and real-time requirements of HAS clients. BUFFEST provides tools for (i) detailed emulation of the clients' buffer conditions, in which we try to reconstruct the player's buffer conditions based on information and events from observed chunk downloads, (ii) automated training of online classifiers, and (iii) online classification of ongoing streaming sessions for HTTPS traffic. The ability to carefully estimate buffer conditions is important for characterizing and understanding the user experience of video streaming flows, whereas the ability to perform accurate online classification (and the signals it provides) is important for traffic and flow management.

The framework includes an event-based buffer emulator module that uses detailed HTTP and payload data to emulate the buffer conditions of the clients. For the HTTPS context, the emulator module uses a trusted proxy design for data extraction. The emulator module can be used both on its own (for detailed buffer analysis) or as a training tool for simpler online classifiers. The framework also includes training and evaluation modules for supervised and semi-supervised online classifiers. In contrast to the emulator module, these classifiers only require TCP/IP packet-level information, which can be collected in real-time, do not require any complementary data to be collected or extracted, and are applicable to encrypted HTTPS traffic. These properties allow the online classifiers to be effectively applied in real-time on ongoing HTTPS streams, providing us with in-session low-buffer warning signals.

The accuracy of our emulator is validated using two different services. First, the emulation of a YouTube player is validated against both the statistical reports sent by YouTube players and an instrumented YouTube client that logs its buffer conditions. The former is obtained from the trusted proxy, while the latter is obtained using YouTube's JavaScript interface. Second, we annotate videos of a commercial mobile streaming service with frame information and record a significant number of long duration sessions, for which we also collect proxy logs, providing us with a third type of ground truth comparison. Our emulator is shown to provide good estimations of the buffer occupancy, the number of stalls, and the overall stall duration. Furthermore, our preliminary characterization using the emulator on sample sessions shows that the rate-adaptive algorithms of both services are typically able to ensure "delay-tolerance" in the time until when the next chunk download needs to complete, as they seldom operate under low buffer conditions. Although we do not consider priority policies in this paper, delay-tolerance is an important property, as it suggests that accurate classification into a relatively small high-priority class can allow for effective flow management even when operating at high utilizations.

For online classification, we present and evaluate simple threshold-based and machine-learning-based classifiers based on TCP/IP packet-level information, allowing the classifiers to be applied to HTTPS streams in real-time. The classifiers are shown to effectively differentiate between flows that currently have low buffer conditions (and tight real-time requirements) from clients that have built up a significant playback buffer (and have slack). These results suggest that even simple classifiers can be used to identify low-buffer flows or provide "warning signs" that one or more users are at risk of experiencing reduced QoE, possibly allowing operators to take additional actions and remediation steps (e.g., power management of network elements, offloading, etc.).

Outline: Section 2 makes a case for passive buffer condition extraction, outlines the general challenges, and presents an overview of BUFFEST. Section 3 provides the necessary background of the YouTube streaming service. Sections 4 and 5 present the details and validation of our emulator module. Section 6 presents our online classifiers and the corresponding performance results. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2 BUFFER CONDITION ESTIMATION

Playback stalls are key indicators of user satisfaction and significantly impact video abandonment [16, 22]. Since stalls typically occur due to empty playback buffers, capturing the buffer occupancy of clients is important when trying to understand the clients' playback experience. Identification of clients with low buffer conditions can also be used to improve users' QoE. For example, at a coarse time granularity, an operator can use knowledge about overall streaming quality when performing capacity planning. At a finer time granularity, per-session knowledge or per-client knowledge every minute, for example, can be used to perform offloading and adapt resource allocation and availability (e.g., through power management). Finally, at an even finer granularity, of a few seconds, for example, clients with low buffer conditions can be helped so to reduce the risk of stalls.

2.1 Candidate approaches

To understand the playback quality of users, an operator can either run experiments with instrumented clients or try to extract the information from passive traffic monitoring. Client-side instrumentation is complicated by most popular streaming services using their own proprietary players. Furthermore, the few websites that

[2] Here, BUFFEST refers to the framework's ability to perform buffer condition estimation/prediction.

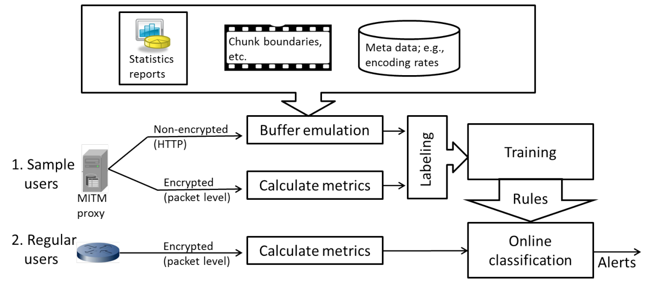


Figure 1: Overview of the BUFFEST framework.

provide player APIs typically significantly restrict the parameters that can be accessed. While additional plugins sometimes can be used to get around some limitations [30], client-side experiments typically introduce additional load on the system, and do not help capture the performance of non-instrumented clients.

In this paper, we instead focus on passive traffic monitoring, using an instrumented client only to validate our approach. This approach is more scalable and allows us to consider non-instrumented clients, but may miss some client-side processing, and therefore only provides an approximation of the clients' playback sessions.

With most video streaming services using HAS, a natural approach to estimate the buffer conditions during playback sessions is to extract HTTP information from traces. For example, HTTP requests during streaming sessions typically contain information about which chunks (or range of bytes) are being downloaded, and at what quality level or bitrate these chunks are downloaded. As we show here, given the right metadata (typically extracted elsewhere) regarding chunk boundaries and playback rates, for example, it is relatively easy to emulate what happens on the client side. However, with increasingly many services, including YouTube, using HTTPS for their delivery, extracting HTTP requests and the appropriate metadata information is becoming increasingly difficult. Future traffic monitoring-based approaches may therefore have to rely only on TCP/IP packet-level information. For this reason, we design a classification framework that can be used on both HTTP and HTTPS traffic, and evaluate different classifiers' relative ability to recognize when clients are experiencing low buffer conditions.

An alternative approach to detect low-buffer instances is for content providers or players to explicitly inform the "network" about potential buffer problems (e.g., by setting a packet header flag or sharing QoE reports). However, such approaches require collaboration between content providers and network operators, can be manipulated by clients wanting preferential treatment, and are today not implemented by any globally popular service. In the case such collaborative techniques become common, the techniques presented here could be valuable in distinguishing legitimate warning signals from signals generated by greedy clients.

2.2 Classification framework overview

There is a natural tradeoff between the classifiers' accuracy, complexity, and access to information. BUFFEST explores this tradeoff for the context of adaptive video streaming over HTTPS. At one end of the spectrum we present a careful buffer emulation module (Section 4) that uses as much application-layer and metadata information as possible and requires additional data acquisition and information extraction. At the other end of the spectrum we present simple online classifiers (Section 6) that make their decisions based on metrics easily calculated in real time from TCP/IP packet-level traces. To improve their accuracy, we also provide training and evaluation modules for supervised and semi-supervised learning.

Figure 1 summarizes the BUFFEST framework and its components, when used for online classification. Here, a buffer emulation module is used for automated labeling of sample flows. To extract HTTP information from HTTPS connections, the emulator module relies on a trusted proxy. By simultaneously calculating summary metrics on the corresponding packet-level data we can create labeled training datasets, which we use to extract online classification rules. The online classifiers are then applied with these rules on the encrypted packet-level data.

At a high level, we emulate a player that sits at the network interface card (NIC) of a client (or wherever the proxy is placed) and registers available HTTP-level, TCP/IP-level, and stream metadata. This data includes encoding rates, chunk boundaries, and other information that is typically contained in metadata files.

Although in most cases it may be unfeasible for an operator to acquire all information needed for such buffer emulation in real-time for all clients that it concurrently serves, the emulator provides a baseline for the potential accuracy that can be achieved with traffic monitoring-based approaches. Therefore, any client routed through the proxy can provide useful data for training of online classifiers. For these clients, we use the emulated buffer conditions to build an automatically labeled training dataset to be used for training of simpler online classifiers. The detailed chunk-level information available through the emulator module is also useful in gaining insights when designing simpler classifiers that can extract actionable information in real-time. For example, when evaluating online classifiers, the buffer emulator has proven a valuable tool for further investigation of cases of special interest.

Our online classifiers trade away some accuracy for faster processing. To achieve fast processing, we focus on features based on simple one-pass metrics that are calculated using only packet-level information. While our training and evaluation modules can easily be used for training of any advanced classifier using these metrics, for the purpose of this paper, we focus on simple threshold-based classifiers and basic machine learning classifiers. We have identified classifiers that perform the best after testing methods based on decision trees and Support Vector Machine (SVM) that are available in three public machine learning libraries (Waffles, LibSVM, and Microsoft Azure Machine Learning Studio).

Finally, even though our experiments are done with a trusted proxy, we note that our online classification techniques do not require the clients to go through trusted proxies. Instead, transparent proxies and middleboxes can be used[42]. In order to train these classifiers, the operator could use buffer emulations based on pilot experiments with a subset of clients (under their control) going through a trusted proxy (as done here) or through the use of an instrumented player, if available, for example.

3 YOUTUBE STREAMING SERVICE

To simplify our discussion and limit the amount of service-specific details, we present example classifiers and detailed analysis for the case of YouTube traffic. To validate the generality of the buffer emulation framework, we also present some complementary analysis using screenshot-based measurements from a commercial mobile streaming service. However, since the focus is on YouTube, we next present a brief overview of the YouTube service.

YouTube's streaming techniques are consistent with other HAS services. During a playback session, the client typically downloads video from one CDN server. In addition, YouTube clients typically communicate with a statistics server that collects client-side playback statistics and also with various advertisement servers.

YouTube supports playback in both Flash and HTML5 containers, with both video and audio streams generally being available in formats such as flv, mp4 and WebM. With HTML5 being the expected industry standard for Web streaming, we report experiments using HTML5 enabled clients that use WebM encoded videos.

With HAS services, each video quality encoding is typically split into smaller chunks with unique URLs that can be independently downloaded and played, allowing for efficient quality adaptation. With YouTube, however, each encoding of the video is given a separate identifier and range requests are instead used to download chunk sequences. This has the advantage that a single request can be used to request multiple chunks at a time, avoiding unnecessary on-off periods, for example, that may otherwise hurt client performance [3]. One disadvantage of long range-requests, however, is that clients may be less adaptive to bandwidth variations.

When a client initiates playback, a manifest file is first downloaded that contains information about the different encodings at which the video is available. As common with many services, the client also obtains additional metadata about the encodings and mappings between chunk byte offsets and their corresponding playtimes. This information is then used by the adaptation algorithms to make range-requests that typically map to one to six chunks (i.e., 5-30 seconds of data) at a time. Although the client receives this data linearly, the player requires a minimum amount of information before frames can be decoded. In our emulator we assume that a chunk must be fully downloaded before playback of that chunk.

4 BUFFER EMULATION MODULE

We have built an event-driven module that emulates the buffer conditions over entire playback sessions using HTTP and metadata extracted using a trusted proxy design.

4.1 Proxy measurements

We have setup an experimental testbed using a trusted proxy that splits the HTTPS end-to-end connection, which is a common method reported in literature [34, 40]. On the client side, we redirect the browser traffic to go through *mitmproxy* (v0.13)[3]. The proxy logs the application-level information for each HTTP request and response in clear text, before forwarding the unmodified (encrypted) requests/response to/from the server. Simultaneously, we collect TCP/IP packet-level information. In addition, we download the manifest file of each video and the video's metadata with chunk boundaries for each video quality encoding.

[3]https://mitmproxy.org/

For each video session, we then use the *mitmdump* proxy companion tool to extract information about the communication sequences. In particular, for the main video stream, we extract information about request initiation times, range requests, their encoding rates, and the ports over which these requests were delivered.

Due to limitations of *mitmproxy* v0.13, the proxy logs do not capture download completion times. To obtain these times for range requests and for the individual chunks that make up each range request, we first extract chunk byte boundaries from the metadata of each encoding (described next), and then count successfully delivered in-order payload bytes using the packet traces.

Due to variable bit-rate encoding, chunk sizes can vary significantly even within a specific video quality profile. To extract and identify chunk byte boundaries within a given encoding file and range request, we use *youtube-dl*[4]. The chunk boundaries are then associated with codec-level metadata to compute the mapping between playtime and bytes along the video. The *mkvinfo* tool is used to parse the metadata and to extract the location, playtime, and position in the video bit stream of every key frame.

YouTube specific optimization: In addition to the information about chunk transfers, we also extract information about all statistical reports, sent as separate HTTP requests to YouTube's statistics servers. The client-side information extracted from the URI of these reports include the timestamp of the request, the playpoint at that time, and the elapsed time since beginning playback.

For non-instrumented clients, these reports can be used as a coarse-grained ground truth for when stalls occurred and when playback was resumed. In this paper, we use the information from the statistical reports to (i) align the emulator's playback point with that of the emulations of the proprietary player, and (ii) as a type of ground truth in our evaluation for when playback was initiated and stalls took place. For the ground truth evaluations, we say that a stall has occurred between two statistical reports if there is a change in the relative time difference between the current video playtime and the time elapsed since beginning playback. The total change between these metrics is used to estimate the total stall duration of such events. It is, however, important to note that the frequency of statistical reports typically is only once every 20-30 seconds, and they therefore only provide limited time granularity.

4.2 Emulating the buffer at the NIC

The extracted information (described above) captures the data seen at the client's NIC. Using this data, our emulator module reconstructs the buffer conditions of a player, assuming it gets access to each chunk as soon as the chunk is fully downloaded. The emulator keeps track of the current state (i.e., "buffering", "playing", or "stalled") and the next event that can change the player's state, including chunk download completions and the buffer dropping to zero (causing stalls). To allow for post processing of player dynamics, we record logs with all emulated events and player states.

YouTube (and other HAS players) sometimes re-download chunks at a higher quality [28, 39]. In these cases, our emulator module (optimistically) assumes that the player always plays the chunk at the highest quality available at the player at the time the chunk is about to be played. Finally, we use statistical reports to determine

[4]https://github.com/rg3/youtube-dl/

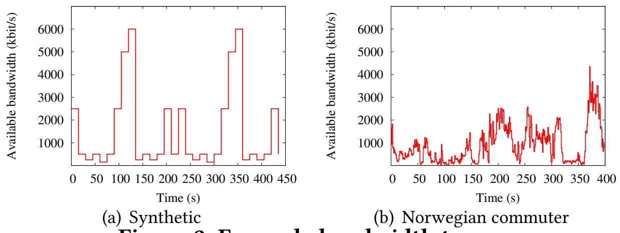


Figure 2: Example bandwidth traces.

Table 1: Summary of bandwidth traces.

Trace	Throughput (kbits/s)				Duration
	Min	Max	Mean	Std	(seconds)
Synthetic high	300	12,000	2,986	3,578	450
Synthetic low 1	150	6,000	1,493	1,789	450
Synthetic low 2	100	5,000	1,426	1,606	450
Synthetic low 3	150	6,000	1,493	1,789	450
Synthetic low 4	100	5,000	1,369	1,668	450
Synthetic low 5	150	6,000	1,493	1,789	450
Norway (ferry 1)	22	3,185	1,353	733	400
Norway (ferry 2)	114	3,594	1,376	776	400
Norway (tram 1)	11	4,354	915	806	400
Norway (tram 2)	11	2,999	983	578	400
Norway (tram 3)	11	2,003	609	367	400
Norway (Bus)	0	5,751	1,797	864	700

the time instances playback begins (within a granularity finer than an RTT) and to re-align the playpoint whenever a stall has occurred.

Thus far we have focused on the case when the user plays the video from start to finish. With stored on-demand videos, the client may also use interactive functionalities such as fast-forward, rewind, and pause. We have extended the framework to handle instances when the user forwards (or rewinds) to a location in the video that is beyond (or outside) the current buffer. In this case, our implementation notices a gap in the chunks downloaded and assumes that the player has moved to a new playback position. For pause operations, we note that our emulator will be conservative, as it will continue to drain the estimated buffer. Although being conservative under a pause might lead to cases where clients with large buffers are identified as otherwise (false positives), our approach would still avoid false negatives (low buffers identified as high). This is an important distinction, as a conservative classifier often is preferred over aggressive classifiers even if with higher precision.

Finally, we note that although user interactions and additional trick modes (e.g., playback at other play rates) are available, large user behavior studies have reported that sessions using these features account for a relatively small fraction [14]. We leave the design and evaluation of policies to detect trick play as future work.

5 EMULATOR VALIDATION

The accuracy of our emulator is validated using two different streaming services and multiple ground-truth datasets.

5.1 Videos and bandwidth profiles

For the YouTube validation we use five synthetic and five real-world bandwidth traces from a 3G network [35]. They are chosen to provide diverse and challenging conditions, and are used together with 50 YouTube videos (chosen to represent a diverse set of video categories): News/TV shows (7), Music videos (5), Professional User Generated Content (UGC) (11), Homemade UGC (10), Games/Sports (7), and Short movies/animations (10). All of our videos are 4-8 minutes long, with an average playtime of 347 seconds, and all videos are played for their full duration. Figure 2 shows two example traces and Table 1 summarizes key statistics for the bandwidth traces. As with other representative videos, some videos in this subset allow us to embed them in any player, particularly in our instrumented player, whereas others can only be played with the official YouTube player.

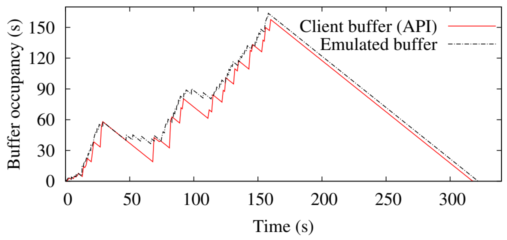


Figure 3: Example comparison of the buffer estimated at the NIC (using emulator) and observed at the player (API).

5.2 Player-instrumented validation

To validate the event-based emulator, we first use YouTube's JavaScript API to access parameters internal to the player and build a ground truth of the buffer conditions seen in the player. To access the YouTube player over the API, each video is embedded in a webpage to which we add JavaScript code that logs detailed client-level information. The player is then instrumented to make per-second logging of the Unix time, buffer occupancy, current play point, playback quality, and the true player state (i.e., if it is buffering, playing, or is stalled). By simultaneously logging HTTP and packet-level traces of the playback sessions (using our framework), we can emulate and compare the buffer levels and playback states obtained by our emulator module with those observed on the player.

While this data provides an excellent ground truth, a limitation of using the API (in our own custom page) to access the YouTube player is that we cannot use the videos for which the uploader has disabled embedding into other webpages or videos that require users to be logged in. This limits the API-based validation to the subset of 30 videos that can be played back with API-level access.

For our experiments, we use a Google Chrome browser configured to use a proxy that runs on the client machine. The machine runs Linux Mint v17 using Linux kernel (3.13.0-24) and is equipped with a Gigabit Ethernet interface, Intel i7 CPU, and 8 GB of RAM. We use *dummynet* [36] to control the available bandwidth at a per-second granularity. Due to the prevalence of CDNs, no additional delays are added to the RTTs from/to the YouTube edge servers.

Figure 3 shows an example comparison between the buffer occupancy reported by the API and the emulated buffer (observed at the NIC) during a streaming session of a 5.5 minute long video. In the example, the client has relatively good bandwidth conditions, allowing it to download chunks at a high quality for most of the session. As desired, the two buffer curves (for the emulated buffer and the actual buffer) nicely follow each other, showing that the emulator captures the general buffer dynamics. A closer look at the difference between the two curves shows that the emulated buffer

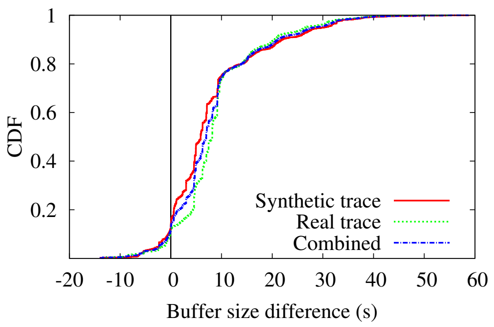


Figure 4: CDF of the differences in buffer sizes observed at the NIC (emulator) and the player (API).

size almost always is slightly larger in this scenario. The reasons for the slightly larger estimates are that in this example the time instances when playback starts are almost the same for the emulated player (whose startup instance partially can be adjusted with the help of statistical reports) and the real player, and the NIC always receives the chunks before the player (since the players experience additional operating system (OS) related delays, for example).

We next take a closer look at the difference in buffer sizes observed at the NIC (emulated) and the player (using API). Figure 4 shows the cumulative distribution function (CDF) of the difference between the two buffer sizes, measured at 1 second intervals during playback, over a large number of playback sessions. Here, we have used six bandwidth traces (3 synthetic and 3 real traces) combined with five different videos per trace. Although the differences typically are relatively small, we observe a few larger differences.

Most of the observed differences are due to differences in when chunks are seen on the NIC (emulated player) and by the API (real player). First, there is a delay between when a chunk is fully downloaded, as seen on the NIC, and when it is available at the player. This is in part due to OS-related delays, caused by having to pass TCP buffers and time varying CPU sharing between competing processes, for example. Second, a more subtle but highly noticeable difference occurs due to how and when the player receives consecutive chunks within a range request. Referring back to Figure 3, chunks often appear to be delivered to the real player in batches (indicated by sharp vertical spikes in the API curve). This is typically (but not always) due to multiple chunks associated with some range requests being delivered simultaneously, when a subset of chunks is fully downloaded. In contrast, the emulator always treats each constituent chunk of a range-request as available for playback as soon as it is fully downloaded. In these cases, the emulator is somewhat optimistic in when chunks are available to the player.[5]

While large differences due to the above reasons are not uncommon (e.g., 24% differ by more than two chunks), we have found that the lag causing these differences normally is temporary and the player typically quickly catches up. For example, the cases with more than 20 seconds difference (with an average difference of 27 seconds), the average difference for this subset (ignoring additionally downloaded chunks at the NIC) reduces to 9.5 seconds after 4 seconds and to 0.69 second after 8 seconds. This suggests that the

[5] This does not mean we provide a bound for the buffer size, since the startup delays may still differ (in both directions).

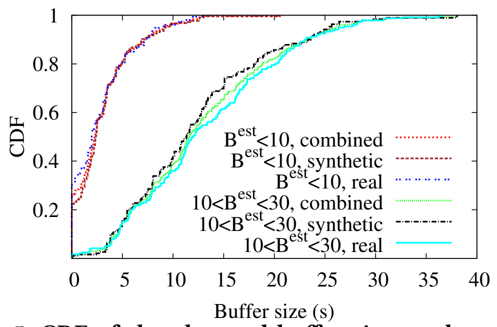


Figure 5: CDF of the observed buffer size at player (using API) when the emulated buffer B^{est} (at NIC) is low (<10s) and intermediate (10-30s).

OS-related delay, even when delivering multiple chunks at once, is less than 8 seconds.

As acknowledged and considered here, the above delays complicate predicting when chunks are needed by the clients and have implications for the design of the player-side algorithms themselves, as such algorithms also may need to take into account random delays introduced by the OS, not only the bandwidth conditions. Rather than modeling these delays (e.g., using a stochastic model), we acknowledge their existence and quantify their impact.

5.3 Coarse-grained classification

While the above OS and player internals make the exact buffer conditions impossible to capture using only network data, we have found that the framework can distinguish clients with low buffer conditions from other clients. To illustrate how this technique can achieve the goal, Figure 5 shows the actual buffer conditions for clients that the emulator estimates will have low buffer conditions (estimated by the emulator to have less than 10 seconds buffered) and intermediate buffer conditions (estimated to have buffered 10 to 30 seconds). Note that there is a clear separation between these two categories and most of the cases that are misclassified are due to overestimations. Furthermore, for the clients with low-buffer estimates ($B_{est} < 10$) the actual buffer is in fact less than 10 in 98% of the cases and it is very rare that the clients that we predict will have intermediate buffer conditions ($10 < B_{est} < 30$), drain their buffers down to zero. These results show that the emulator can be used as a good estimator of the coarse-grained buffer conditions of the player itself, despite the OS-related delays.

5.4 Startup delays and OS-related delays/inertia

In general, we have found that for most streaming sessions, especially those with good download speeds, playback begins when the first chunk is fully downloaded. This is illustrated in the scatter plot shown in Figure 6(a). Here, example results are shown for three traces (two synthetic and one real-world trace) and all 50 sample videos. Results for other traces are similar. Motivated by this observation, we calculate the time between the reported startup time and the time that the first chunk was fully downloaded. Figure 6(b) shows the CDF of this difference. Note that the time differences in most cases are between 1/8 and 1/4 of a second, suggesting that the OS-related delays for the first chunk typically are small, although there clearly are exceptions with significantly larger differences.

Table 2: Stall event summary for the emulator.

Metric	Synthetic low	Synthetic high	Bus trace
Actual stall events	111	6	8
Emulated stalls	107	7	10
Correct stall events	81	6	6
Videos with stall	41	6	5
Videos with emulated stall	41	6	8
Videos with correct stall	41	6	5
Videos with correct first stall	34	6	4
Overall false positives	0.5	0.02	0.08
Overall sensitivity	0.81	1.00	0.75
Overall specificity	0.99	0.99	0.99
Overall precision	0.75	0.85	0.63
Overall accuracy	0.98	0.99	0.99
Overall F1 score	0.78	0.92	0.66
Overall stall duration ratio	1.09	1.16	1.41

5.5 Stalls compared with statistical reports

For a provider-side comparison, we have also evaluated the accuracy of the NIC-based emulator against what YouTube may see based on the statistical reports that clients periodically send to their statistics servers. Since these reports do not provide information about buffer levels, we use stall and stall duration metrics for this evaluation.

Table 2 summarizes various accuracy metrics calculated across all stalls observed by the emulator, for the same traces and videos used in Section 5.4. Here, sensitivity (sometimes called recall) is the ratio of true positives to the sum of true positives and false negatives. Specificity is the ratio of true negatives to the sum of true negatives and false positives. Precision is the ratio of true positives to true positives and false positives. Accuracy is the ratio of the sum of true positives and true negatives to the sum of true positives, false positives, true negatives and false negatives. Finally, the F1-score is equal to the harmonic mean of the precision and the sensitivity. Since the statistical reports (submitted roughly every 20-30 seconds) only allow us to determine if there has been at least one stall between two reports, and the (combined) duration of any such stalls, not how many stalls there have been between the two reports, all reported statistics are calculated on the granularity of statistical reports. We call the interval between two reports a stall event if there was a stall between the two reports and we consider the emulated stall(s) as "correct" only if the stall(s) (i) occurs between the same two statistical reports as it is observed by YouTube, and (ii) the combined duration of the stall(s) between these two time instances differ by at most 50%. The overall stall duration ratio is calculated as the ratio of the stall duration reported by the emulator and the stall duration observed from the statistical reports.

Even with the restrictive interval definition, our emulator correctly emulates the time and duration for 93 of 125 stall events observed with the use of statistical reports. While it may appear that we have 32 false positives here, looking closer at the data, all these cases too correspond to actual stall events on the player. For these cases, either the timing or the stall duration do not (exactly) match those extracted based on the statistical reports. These differences are primarily due to the coarse granularity with which stalls are identified from the statistical reports (as they only reveal that a stall occurred between two stats reports, not when) combined with the lag between the NIC and the actual player. Similar observations hold for the other traces.

Another interesting observation is that we correctly identify all 52 sessions (out of 150 sessions) that contain at least one stall, while only having three false positives. Furthermore, for 44 of the videos the time instance and duration of the first stall was correct. The higher than average detection rate for the first stall (84.6%) compared to across all stalls (66.4%) is positive, since the first stall may be the most important to avoid for user satisfaction purposes. The higher accuracy can be explained by the initial startup instances being easier to estimate than those after stalls.

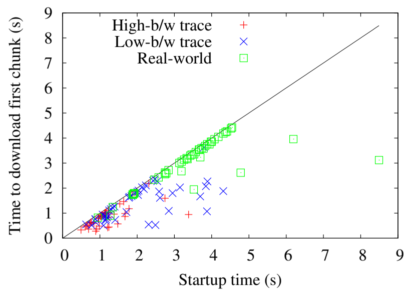
While the OS-related delays explain most stalls observed on the player that are not captured by the emulator, we have observed some interesting cases due to partial chunk replacement. In these cases, the client first downloads a sequence of chunks (say chunks 1-7) at a low quality, and then requests a sequence of chunks (say 5-7) at a higher rate, but does not obtain all chunks (e.g., chunk 6) by its playback deadline. In these cases, our emulator assumes that the client always plays at the highest quality for which it has a complete chunk, whereas it appears that the YouTube player in some cases does not fall back to the lower encoding after making a request to replace a set of chunks. This is probably because the player is implemented so that it cannot make use of the lower quality chunks as they may have been flushed from the buffer, for example, and there is overhead associated with switching back to the lower encoding. As these cases are rare and we expect future players to handle these situations better, we did not try to modify our emulator to match the YouTube player's current behavior.

We have also manually validated that any stall that the emulator identifies in fact is a stall on the player. This should always be the case whenever a statistical report has been used to synchronize the startup time of the emulator. Overall, our results suggest that emulating the buffer conditions at the NIC provides a reasonable estimation of the buffer conditions and stalls at the player.

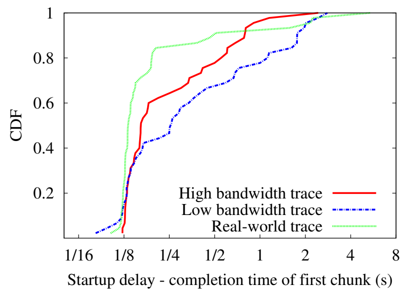
5.6 Fast-forward operations

Experiments have been performed to validate the effectiveness of the emulator under user interactive operations such as fast-forwards. Our approach applies to any interactive operation (fast-forward, rewind, etc.) leading the player to a play-point that has not been buffered. The results have been positive, with the approach discovering fast-forwards much faster than the statistical reports (which typically results in a 5-30 seconds observation delay). Our YouTube version of the emulator combines the two methods.

To illustrate the effectiveness of the approach, we summarize the results of 30 random experiments with both fast forwards and stall events. Out of these, 15 are based on synthetic traces and 15 use real traces. For each experiment, we initially play the video for 60 seconds, after which the playpoint is forwarded a random time-duration beyond the current buffer, causing an out-of-buffer forward. The video is then played until the end and the evaluation looks at the first stall event that occurs after the fast-forward. Out of the 30 experiments, 28 contained stalls after the fast-forward. The emulator was able to correctly predict the presence of a stall in 86% of the 28 stall-cases and did not make any false predictions. However, as before, the emulator in many cases (due to NIC placement) typically is somewhat ahead of the player and often has some data in its (emulated) buffer at the time that the stall occurred. Figure 7



(a) Scatter plot



(b) CDF

Figure 6: Differences between startup delays and the time that it takes to download the first chunk.

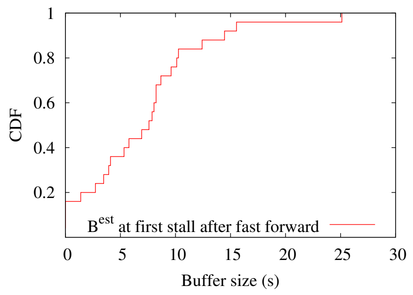


Figure 7: CDF of the emulated buffer conditions at time of stalls.

shows a CDF of the buffer at the emulator at the exact time of these stalls. In 40% of the cases the emulator sees less than one chunk in the buffer and in 84% of the cases it sees less than two chunks in its buffer. Most of the stalls with larger emulated buffer sizes are related to large range requests containing multiple chunks.

5.7 Third-party validation

While our focus in this paper is on YouTube, we have also validated our emulation framework using another video service. In particular, we have obtained simultaneous (i) HTTP transaction logs and (ii) time-synchronized screen captures of streaming sessions from a popular commercial service. The HTTP transaction logs are generated by a proxy that different mobile clients (Android and iPhones) are connected to. The time-synchronized screen captures are generated while playing a special video which displays the frame number and bitrate level in every frame. An optical character recognition (OCR) program reads the per-frame annotated information and logs the current bitrate level, playtime (frame number) and beginning and end of stalls and video playback.

Using these traces we have validated our emulator. While the time-synchronized screen captures do not provide any information about buffer levels, they do carry per-frame information that help us identify a ground truth with the exact time of stalls and their exact duration, as experienced by the user. To estimate the accuracy of our emulations, we therefore compare the emulated buffer levels (based on the information in the HTTP transaction logs) at the time instances when there was a stall. In roughly 50% of the stalls our emulator has less than a chunk and in 80% of the cases it has less than two chunks. The cases with larger buffer estimates can mostly be explained by a larger error in how we estimated when playback resumed. In particular, with the traces being collected over several hours and reported for each 30 minute period, we had to estimate playback starts based on playback resumptions after a preceding stall event. More specifically, we assumed that any data obtained after the time of the first stall occurrence corresponds to data needed after the stall event (estimated as the initial request) and that the playback resumed at the time instances that the player resumed playback. Naturally, this is not always the case, and our estimation of the startup instance can only be considered an approximation.

6 ONLINE CLASSIFICATION

With YouTube and other services that use HTTPS and/or require complementary data to be extracted (e.g., encoding rates and chunk boundary information), it is not possible to emulate the buffer conditions in real time. We next describe our online classification module for such contexts, and present preliminary results using both threshold-based and machine learning classifiers. While we focus on detecting low-buffer conditions, as such events are the most important indicators of viewer experience, our approach can also be extended to consider playback quality and other metrics.

6.1 Calculating metrics online

For the purpose of our proof-of-concept implementation we continually calculate the exponential weighted moving average (EWMA) for different window weights α for both the per-second throughput X_α and the inter-request times I_α. Here, throughput is calculated based on the packet payloads delivered from the server to the client, and inter-request times are estimated as the time between request packets (with payload) from the client to the server. These packets are larger than a regular ACK, and typically contain an HTTP range request to the server. Our validation (omitted) has shown that this is a highly accurate method to capture the timing of range requests.[6]

In parallel, we also calculate and keep track for how long time (T_α^X) that the weighted throughput metric X_α has been below a threshold X_α^*, and for how long time (T_α^I) that the weighted inter-request time metric I_α has been above a threshold I_α^*. As with the EWMA metrics, these metrics too can be effectively calculated online, using a single pass. Finally, our online classifiers are designed to make decisions based on these metrics, as calculated for different α values and thresholds values (X_α^* and I_α^*).

6.2 Classifiers

6.2.1 Threshold-based classifiers. First, for the training phase, we label each training trace based on the emulated buffer condition seen by the client, use a search to find the best threshold combination that provides the best F1 scores (harmonic mean of precision and sensitivity), where F1 scores are calculated based on how well the classifier (with selected thresholds) detects low buffer cases (as classified by the emulator). We consider the client to have low buffer conditions when the emulated buffer has less than B^* seconds of content. Note that training with $B^* = 0$ corresponds to only using

[6] Again, note that the number of chunks requested in each range request is still unknown. Otherwise, this information could be used to emulate the buffer of clients that do not use fast-forward pause, and other VoD functionalities. Within the current framework, it can also be used as a lower bound on the number of chunks obtained within a time window.

Table 3: Best classifier configuration and evaluation results for the threshold-based classifiers.

	Training				Evaluation		
	α	X^*_α(Kbit/s)	$T_\alpha^{X^*}$	F1 score	Sensitivity	Precision	F1 score
Synthetic trace with $B^* = 0$	0.15	400	5	0.49	0.49	0.59	0.49
Synthetic trace with $B^* = 5$	0.5	550	20	0.28	0.77	0.53	0.51
Synthetic trace with $B^* = 10$	0.3	600	25	0.40	0.7	0.66	0.57
Synthetic trace with $B^* = 20$	0.2	550	25	0.59	0.58	0.72	0.55
Synthetic trace with $B^* = 40$	0.25	300	10	0.71	0.62	0.73	0.58
Real trace with $B^* = 0$	0.45	800	25	0.37	0.37	0.66	0.40
Real trace with $B^* = 5$	0.15	600	5	0.16	0.63	0.46	0.48
Real trace with $B^* = 10$	0.05	900	10	0.35	0.72	0.73	0.67
Real trace with $B^* = 20$	0.1	850	20	0.61	0.53	0.72	0.55
Real trace with $B^* = 40$	0.15	900	20	0.70	0.62	0.85	0.65

stall instances for training. With this definition, a true positive is any instance where the classifier would indicate a buffer below B^* and the buffer actually is below B^*.

We have found that the best throughput-based classifiers almost always outperform the inter-request-based classifiers. For example, in our default scenario all inter-request-based classifiers obtain F1 scores less than 0.2. For this reason, we will focus primarily on the throughput-based classifiers[7].

To find a good threshold-based classifier we perform a fine-grained brute force search over all α values and threshold pairs X^*_α and $T_\alpha^{X^*}$ in which we use 20 different levels of α values and 105 different threshold pairs to identify the best combination of values which results in the highest F1 score. This configuration is then used for the evaluation on the evaluation set, different from the training set, for which we report results in Section 6.3.

6.2.2 Machine learning classifiers. Although threshold-based classifiers allow quick parameter selection and online reconfiguration, their predictive powers are generally considered limited when compared to machine learning techniques. In this work we tested the techniques based on decision trees and Support Vector Machine (SVM) implemented in three popular machine learning packages (Waffles[8], LibSVM[9], and Microsoft Azure Machine Learning Studio[10]). Here, we report results for the two-class boosted decision tree classifier. Among the classifiers we considered, this classifier provided the best scores both during training and evaluation.

Boosted decision trees [21] is a class of decision trees that adjusts (boosts) the weights of the trees at the end of every training step based on whether the previous tree classified the data correctly. In our context, the classification problem is based on whether a playback stall would occur or not, given the observed throughputs over different time periods. Boosted decision trees are particularly attractive when features are related (have low entropy) [10].

For the evaluation, the training data was generated by computing the average throughput per second observed over different time windows during playback. The window sizes that we consider are 5, 10, 20, 40, 80 and 160 seconds. By computing the average throughput over different time windows, we aim to capture short-term fluctuations with the smaller windows and long-term degradation with the larger windows. As before, both the training and evaluation datasets (different) are tagged with stall occurrences based on the emulated buffer. While these metrics are simple and easy to extract, it should be noted that they are correlated, again motivating the choice of boosted decision trees.

6.3 Prediction evaluation

For both threshold-based and machine learning classifiers, our evaluation was performed separately on the synthetic and the real traces. In all cases, we picked three bandwidth trace types for training and two different bandwidth trace types for evaluation. For each trace type, we run ten different experiments, with different randomly selected videos, giving us 2×30 training and 2×20 evaluation instances. Although we only have a limited set of bandwidth traces, this methodology allows us to ensure that there is no overlap in the bandwidth traces or in the videos between the two sets.

6.3.1 Threshold-based classifiers. The results of the threshold-based classifiers are summarized in Table 3. Here, we show the parameter selection from training (columns 2-4), the F1 score on the training dataset (column 5), and the results on the evaluation dataset (columns 6-8); broken down into sensitivity (column 6), precision (column 7) and F1 scores (column 8). For both the synthetic and real scenarios we show results with B^* equal to 0, 5, 10, 20, and 40 seconds. In general, a larger B^* value provides a larger window for detection. Referring to the parameter selection (columns 2-4), we note that our training framework allows us to adjust the parameters for each case. When interpreting the results it should be noted that the choice of B^* impacts the performance measures and the tests (and techniques) are designed to test how well low-buffer conditions (rather than stall events) can be identified.

Figure 8 shows the CDF of the buffer conditions as seen when the threshold-based online classifiers predicted low buffer conditions, and put them in contrast to the conditions as observed over all sessions. The substantial differences in the CDFs are encouraging as it shows that relatively simple classifiers can be useful in predicting low buffer conditions even when the traffic is encrypted.

While our results presented here are with relatively simple threshold classifiers, the generality of the framework allows automated labeling and training using a much richer set of classifiers. We next consider the machine learning classifiers.

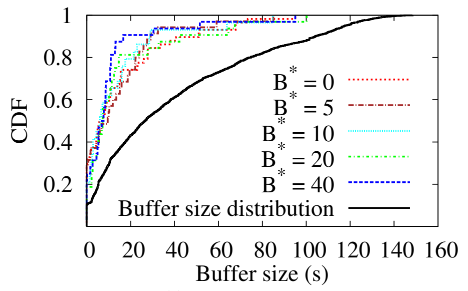
6.3.2 Machine learning classifiers. Table 4 shows the results of the boosted decision tree classifier available in Microsoft Azure Machine Learning Studio. We note that this classifier improves

[7] A combination of throughput-based and inter-request-based classifiers could also be used. While we leave this as future work, such techniques could be used to capture trick play modes (2x, 4x, etc.), where the inter-request-time could be used to estimate the playback rate.

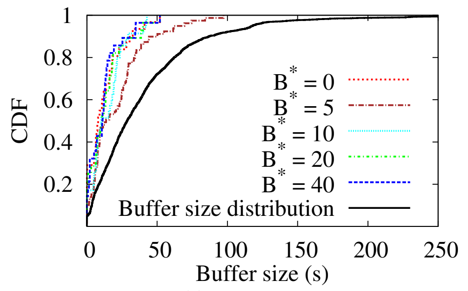
[8] http://waffles.sourceforge.net/

[9] https://www.csie.ntu.edu.tw/~cjlin/libsvm/

[10] https://studio.azureml.net/



(a) Synthetic trace



(b) Real trace

Figure 8: CDF of emulated buffer sizes and buffer conditions when using the threshold-based classifiers.

Table 4: Results for boosted decision tree classifier.

	Sensitivity	Precision	F1 score
Synthetic trace with $B^* = 0$	0.49	0.27	0.35
Synthetic trace with $B^* = 5$	0.43	0.52	0.47
Synthetic trace with $B^* = 10$	0.55	0.47	0.51
Synthetic trace with $B^* = 20$	0.75	0.63	0.69
Synthetic trace with $B^* = 40$	0.68	0.90	0.78
Real trace with $B^* = 0$	0.10	0.31	0.07
Real trace with $B^* = 5$	0.17	0.39	0.24
Real trace with $B^* = 10$	0.52	0.53	0.52
Real trace with $B^* = 20$	0.86	0.61	0.71
Real trace with $B^* = 40$	0.82	0.82	0.82

noticeably over the simple threshold-based classifiers for the cases when we use intermediate-to-large B^* values (e.g., 20 or 40), but performs much worse with small B^* values (e.g., when $B^* = 0$). One reason for the low accuracy when $B^* = 0$ is an imbalance between stall and non-stall instances. For example, with $B^* = 0$ the ratio of stall instances to playback instances was only 0.071 for the synthetic traces and 0.016 for the real traces. Furthermore, although there are several stall instances, when compared to the entire playback length, the duration of stalls is small.

Fortunately, as discussed above, the low-to-intermediate buffer cases (e.g., using $B^* = 20$) are likely of more interest for real-time optimization techniques. The better accuracy for these cases can be explained by richer and more balanced training data. For example, the ratio of instances where the buffer size was less than or equal to $B^* = 20$ was 0.441 for the synthetic trace and 0.358 for the real trace. With $B^* = 40$ the corresponding ratios were 0.694 and 0.826.

Finally, we look closer at the actual buffer conditions at the instances when the boosted decision tree classifier predict low buffer conditions. Figure 9 shows the CDF of buffer conditions when the boosted decision tree classifier uses different B^* values. Interestingly, although the classifier had a poor F1 score for the synthetic cases with $B^* = 0$, we note that a significant amount of the instances identified are cases where the buffer size is less than 20 seconds. This suggests that this classifier can be used to identify low buffer conditions even with $B^* = 0$. For other values of B^*, the classifier again performs better owing to the richer training data and more relaxed constraints.

Overall, these results show that the boosted decision tree classifier provides a good tool to predict instances with low buffer conditions. By careful selection of B^* we can also achieve a good tradeoff between the number of flagged low buffer instances and the accuracy with which these are reported. While the machine learning classifiers in general do not provide the same intuition as the threshold-based classifiers, we note that the boosted decision tree classifier typically has higher F1 scores (for intermediate-to-large B^* thresholds) and is easy to implement as a real-time classifier using existing software packages.

We have also evaluated other machine learning techniques, such as SVMs on our dataset. The boosted decision tree classifier outperforms the SVM classifier when looking across performance scores for different values of B^*, especially for $B^* = 0$, $B^* = 5$ and $B^* = 10$. For larger thresholds, the SVM classifier delivers very similar results, and in general, when compared to the boosted decision tree, has a slightly lower sensitivity and higher precision.

6.4 Discussion and limitations

While we only evaluate classifiers using two services, and acknowledge that the implementations and adaptation algorithms may change over time even for an individual service, we note that the general BUFFEST framework is easily extendable for other services and classifiers continually can be retrained. The ease of applying the framework to other services was demonstrated and validated when applying the emulation framework for the second commercial service. In this case, we simply changed the sources of data in the API. The retraining is simplified by the use of a separate training module and the use of the trusted proxy makes it applicable regardless of HTTP or HTTPS being used for the transfer.

For training purposes encoding rates and chunk boundaries need to be known. While the emulation module (used for training) currently works with decrypted manifest files, which were downloaded using *youtube-dl*, other services might not allow access to manifest files through external programs. However, this information can still be extracted from payload using the trusted proxy design.

The current experiments are done by collecting network traces on the client. Although the network can increase the variability and differences observed between the player and the emulator if located further away from the client, we expect these increases to be relatively small compared to the OS-related delays we have observed here. For example, most routers maintain reasonably-sized buffers (e.g., using the bandwidth-delay product rule) and the buffer bloat phenomenon is relatively rare in practice, typically resulting in fluctuating queues, rather than large-scale persistent queues [5]. It appears more important that both the packet-level traces and proxy-based HTTP traces are collected at the same location.

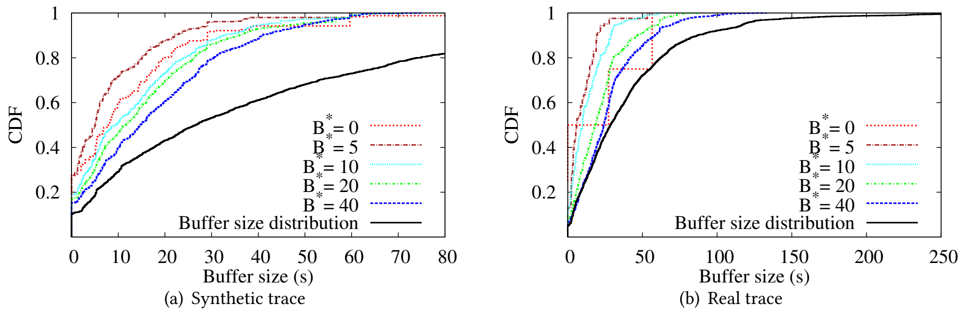


Figure 9: CDF of emulated buffer sizes and buffer conditions when using the boosted decision tree classifier.

7 RELATED WORK

Online flow classification has been used extensively in the past. While Deep-Packet Inspection (DPI) typically is considered too slow for online classification [43], efficient online performance has been demonstrated using supervised techniques based on Naïve Bayes [29], automated and semi-automated clustering techniques [7, 19, 45], blind traffic classification based on simple flow-based metrics [25], and statistical analysis based on specific properties [44]. To the best of our knowledge, our paper is the first to provide automated re-classification of encrypted streaming flows based on the expected buffer conditions and urgency of different clients.

Closest to ours is very recent work by Dimopoulos et al. [15] and Orsolic et al. [31]. Dimopoulos et al., present a framework to discern streaming video's QoE based on network traces. However, we differ significantly in our focus and ground truth evaluation. For example, they only consider per-session classification based on high-level statistics, do not capture the buffer dynamics, and some of their key indicators (e.g., sudden change in the requested quality) are only noticeable after a stall. In contrast, our framework identifies low buffer conditions in real-time, thereby facilitating the possibility of intervention so to avoid potential future stalls. We also differ significantly in how we collect and use ground truth measurements. In their case they rely on legacy HTTP traffic (which is diminishing), whereas we create a general training framework also for services relying fully on HTTPS and collect player-side ground truth measurements (e.g., using the JavaScript API, statistical reports, and screenshot measurements) for our validation.

Orsolic et al. [31] present a machine learning based approach to map playback sessions to QoE classes based on network traces. They use the YouTube player API to generate training and test datasets, however, critically differ in estimating only the QoE of the sessions and do not consider identifying low buffer conditions.

Others have designed stall monitoring tools or considered stall prediction, but only in the context of HTTP. For example, Casas et al. [13] design online monitoring of YouTube clients' stalls using DPI and packet-level monitoring, but do not take into account that YouTube has shifted to use HAS (with quality adaptation) and HTTPS. Similarly, Wu et al. [41] uses CDN logs and information shared from clients to create a machine-learning-based stall detection technique for non-encrypted HTTP traffic, which they evaluate on Apple HTTP Live Streaming video sessions obtained from a CDN and controlled lab experiments using Microsoft Smooth Streaming. Others have used packet-level and video information to reconstruct client-side buffer conditions [37], used information about the maximum buffer size, chunk size, and startup times to reconstruct VoD sessions [23]. Again, none of these works are applicable to HTTPS.

Other closely related works have also looked at measuring video Mean Opinion Scores (vMOS) [32] and estimating the encoding rate and playback duration of chunks that are being downloaded [17] by looking at HTTPS traces.

Schatz et al. [37] use YouTube's statistical reports to characterize client rebuffering and abandonment. However, similar to the work by Dimopoulos et al. [15], this approach facilitates identification of events such as stalls only after they have occurred and does not identify or facilitate low-buffer conditions in real-time. Several other works have characterized the YouTube service itself [11, 12, 20], including the quality adaptation and redundant downloads [28, 39].

Finally, it should be noted that many techniques have been proposed for client-driven or server-assisted quality adaptation [4, 24], and for network-assisted prioritization of flows [6, 26] using SDN-based technologies such as OpenFlow [33]. Others have proposed network-assisted quality selection for HAS clients based on network-based monitoring [8], or used measurements to model and characterize user satisfaction when using online services [38].

Standardization efforts have also focused on establishing frameworks for clients to directly report QoE metrics to network elements [1]. However, these approaches are not yet widely deployed, and place restrictions on using HTTPS and video formats that can be used. Complementary to these, our approach leverages the information encoded in the traffic to understand clients' buffer conditions through emulation and packet-level classification. While we do not consider prioritization here, future work could include the design of such optimization schemes that leverage BUFFEST to assess the urgency of different flows.

8 CONCLUSION

We have presented the BUFFEST classification framework that includes both an event-based buffer emulator module and training modules for online classifiers. Motivated by increasing usage of HAS over HTTPS, the emulator module leverages a trusted proxy method to extract required information about the video flows delivered to the clients, allowing us to identify chunk boundaries and track buffer conditions, as seen on the NIC of the proxy. We compare our solution against the player's ground truth using an instrumented YouTube client, synchronized screen captures of videos sessions using a different commercial streaming service, as well as YouTube's statistical reports. Although using an instrumented client is unfeasible in practice for real-time classification and screen captures only are possible in a lab environment, these validation results are useful to show that our emulator module relatively accurately captures and tracks the client's buffer conditions. Using the emulator for automated labeling of network traces provides an effective training framework for simpler online classifiers that only use TCP/IP packet-level information. While more advanced machine learning techniques easily can be used in the framework, our results show that relatively simple threshold-based and machine learning classifiers (e.g., the boosted decision tree classifier) can allow network operators to distinguish a significant fraction of low buffer instances even when the traffic is encrypted using HTTPS. The flexible design also allows the framework to be used for characterizing and understanding clients' QoE, and to identify early signs of stalls or insufficient QoE. Future work includes the design of flow management solutions based on these classifiers.

ACKNOWLEDGEMENTS

The authors are thankful to our shepherd Hermann Hellwagner and the anonymous reviewers for their feedback. This work was funded in part by the Swedish Research Council (VR) and the the Swedish National Graduate School in Computer Science (CUGS) at Linköping University.

REFERENCES

[1] 2012. ETSI TS 126 247 V10.1.0. Technical specification. European Telecommunications Standards Institute. (2012).

[2] 2015. Sandvine Global Internet Phenomena- Africa, Middle East and North America- Technical Report. (2015).

[3] S. Akhshabi, L. Anantakrishnan, A. C. Begen, and C. Dovrolis. 2012. What happens when HTTP adaptive streaming players compete for bandwidth?. In *Proc. ACM NOSSDAV.*

[4] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen. 2013. Server-based Traffic Shaping for Stabilizing Oscillating Adaptive Streaming Players. In *Proc. ACM NOSSDAV.*

[5] M. Allman. 2013. Comments on Bufferbloat. *ACM CCR* 43 (2013), 30–37. Issue 1.

[6] S. Benno, J. O. Esteban, and I. Rimac. 2011. Adaptive Streaming: The Network HAS to Help. *Bell Lab. Tech. Journal* (2011).

[7] L. Bernaille, R. Teixeira, and K. Salamatian. 2006. Early Application Identification. In *Proc. ACM CoNEXT.*

[8] N. Bouten, R. de Schmidt, J. Famaey, S. Latre, A. Pras, and F. De Turck. 2015. QoE-driven in-network optimization for Adaptive Video Streaming based on packet sampling measurements. *Computer Networks* 81 (2015), 96–115. Issue C.

[9] Z. Cao, G. Xiong, Y. Zhao, Z. Li, and L. Guo. 2014. A survey on encrypted traffic classification. In *Proc. ATIS.*

[10] R. Caruana and A. Niculescu-Mizil. 2006. An Empirical Comparison of Supervised Learning Algorithms Using Different Performance Metrics. In *Proc. ICML.*

[11] P. Casas, A. D'Alconzo, P. Fiadino, A. Bär, A. Finamore, and T. Zseby. 2014. When YouTube Does not Work: Analysis of QoE-Relevant Degradation in Google CDN Traffic. *IEEE Trans. Network and Service Management* 11 (2014), 441–457. Issue 4.

[12] P. Casas, P. Fiadino, A. Sackl, and A. D'Alconzo. 2014. YouTube in the move: Understanding the performance of YouTube in cellular networks. In *Proc. IFIP WD.*

[13] P. Casas, M. Seufert, and R. Schatz. 2013. YOUQMON: A System for On-line Monitoring of YouTube QoE in Operational 3G Networks. *SIGMETRICS Perform. Eval. Rev.* 41 (2013), 44–46. Issue 2.

[14] L. Chen, Y.P. Zhou, and D.M. Chiu. 2014. A Study of User Behavior in Online VoD services. *Computer Communications* (2014).

[15] G. Dimopoulos, I. Leontiadis, P. Barlet-Ros, and K. Papagiannaki. 2016. Measuring Video QoE from Encrypted Traffic. In *Proc. IMC.*

[16] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. 2011. Understanding the Impact of Video Quality on User Engagement. In *Proc. ACM SIGCOMM.*

[17] R. Dubin, A. Dvir, O. Pele, O. Hadar, I. Richman, and O. Trabelsi. 2016. Real Time Video Quality Representation Classification of Encrypted HTTP Adaptive Video Streaming - the Case of Safari. *arXiv preprint* (2016).

[18] J. Erman, A.Mahanti, M. Arlitt, and C. Williamson. 2007. Identifying and Discriminating Between Web and Peer-to-peer Traffic in the Network Core. In *Proc. ACM WWW.*

[19] J. Erman, M. Arlitt, and A. Mahanti. 2006. Traffic Classification Using Clustering Algorithms. In *Proc. ACM SIGCOMM Workshop on Mining Network Data.*

[20] A. Finamore, M. Mellia, M. Munafò, R. Torres, and S. Rao. 2011. YouTube Everywhere: Impact of Device and Infrastructure Synergies on User Experience. In *Proc. ACM IMC.*

[21] J. H. Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics* 29 (2001), 1189–1232. Issue 5.

[22] T. Hoßfeld, M. Seufert, M. Hirth, T. Zinner, P. Tran-Gia, and R. Schatz. 2011. Quantification of YouTube QoE via crowdsourcing. In *Proc. IEEE Symp. on Multimedia.*

[23] R. Huysegems, B. De Vleeschauwer, K. De Schepper, C. Hawinkel, T. Wu, K. Laevens, and W. Van Leekwijck. 2012. Session Reconstruction for HTTP Adaptive Streaming: Laying the Foundation for Network-based QoE Monitoring. In *Proc. IWQoS.*

[24] J. Jiang, V. Sekar, and H. Zhang. 2012. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proc. ACM CoNEXT.*

[25] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. 2005. BLINC: Multilevel Traffic Classification in the Dark. In *Proc. ACM SIGCOMM.*

[26] V. Krishnamoorthi, N. Carlsson, D. Eager, A. Mahanti, and N. Shahmehri. 2013. Helping hand or hidden hurdle: Proxy-assisted HTTP-based adaptive streaming performance. In *Proc. IEEE MASCOTS.*

[27] S. Krishnan and R. Sitaraman. 2012. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-experimental Designs. In *Proc. IMC.*

[28] A. Mansy, M. Ammar, J. Chandrashekar, and A. Sheth. 2013. Characterizing Client Behavior of Commercial Mobile Video Streaming Services. In *Proc. MoVID.*

[29] A. Moore and D. Zuev. 2005. Internet Traffic Classification Using Bayesian Analysis Techniques. In *Proc. ACM SIGMETRICS.*

[30] H. Nam, K. Kim, D. Calin, and H. Schulzrinne. 2014. YouSlow: A Performance Analysis Tool for Adaptive Bitrate Video Streaming. *ACM CCR* 44 (2014), 111–112.

[31] I. Orsolic, D. Pevec, M. Suznjevic, and L. Skorin-Kapov. 2016. YouTube QoE Estimation Based on the Analysis of Encrypted Network Traffic Using Machine Learning. In *Proc. IEEE Globecom Workshops.*

[32] W. Pan, G. Cheng, H. Wu, and Y. Tang. 2016. Towards QoE assessment of encrypted YouTube adaptive video streaming in mobile networks. In *Proc. IWQoS.*

[33] S. Petrangeli, T. Wauters, R. Huysegems, T. Bostoen, and F. De Turck. 2016. Software-defined network-based prioritization to avoid video freezes in HTTP adaptive streaming. *Int. Journal of Network Management* 26 (2016), 248–268.

[34] F. Qian, L. Ji, B. Han, and V. Gopalakrishnan. 2016. Optimizing 360 Video Delivery over Cellular Networks. In *Proc. ACM All Things Cellular.*

[35] H. Riiser, H. S. Bergsaker, P. Vigmostad, P. Halvorsen, and C. Griwodz. 2012. A comparison of quality scheduling in commercial adaptive HTTP streaming solutions on a 3G network.. In *Proc. MoVID.*

[36] Luigi Rizzo. 1997. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM CCR* 27 (1997), 31–41.

[37] R. Schatz, Hoßfeld T, and P. Casas. 2012. Passive YouTube QoE monitoring for ISPs. In *Proc. IEEE IMIS.*

[38] M. Shafiq, J. Erman, L. Ji, A. Liu, J. Pang, and J. Wang. 2014. Understanding the impact of network dynamics on mobile video user engagement. In *Proc. ACM SIGMETRICS.*

[39] C. Sieber, P. Heegaard, T. Hoßfeld, and W. Kellerer. 2016. Sacrificing Efficiency for Quality of Experience: YouTube's Redundant Traffic Behavior. In *Proc. IFIP/IEEE Networking.*

[40] F. Wamser, P. Casas, M. Seufert, C. Moldovan, P. Tran-Gia, and T. Hossfeld. 2016. Modeling the YouTube stack: From packets to quality of experience. *Computer Networks* 109 (2016), 211–224.

[41] T. Wu, S. Petrangeli, R. Huysegems, T. Bostoen, and F. De Turck. 2017. Network-based video freeze detection and prediction in HTTP adaptive streaming. *Comp. Comm.* 99 (2017), 37–47.

[42] X. Xu, J. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. 2015. Investigating transparent web proxies in cellular networks. In *Proc. PAM.*

[43] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz. 2006. Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection. In *Proc. ACM/IEEE ANCS.*

[44] S. Zander, T. Nguyen, and G. Armitage. 2005. Automated traffic classification and application identification using machine learning. In *Proc. IEEE LCN.*

[45] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu. 2015. Robust Network Traffic Classification. *IEEE/ACM Trans. Netw.* 23 (2015), 1257–1270. Issue 4.