Oboe: Auto-tuning Video ABR Algorithms to Network Conditions

Zahaib Akhtar*
University of Southern California

Yun Seong Nam*
Purdue University

Ramesh Govindan
University of Southern California

Sanjay Rao
Purdue University

Jessica Chen
University of Windsor

Ethan Katz-Bassett
Columbia University

Bruno Ribeiro
Purdue University

Jibin Zhan
Conviva

Hui Zhang
Conviva

ABSTRACT

Most content providers are interested in providing good video delivery QoE for all users, not just on average. State-of-the-art ABR algorithms like BOLA and MPC rely on parameters that are sensitive to network conditions, so may perform poorly for some users and/or videos. In this paper, we propose a technique called Oboe to auto-tune these parameters to different network conditions. Oboe *pre-computes*, for a given ABR algorithm, the best possible parameters for different network conditions, then *dynamically adapts* the parameters at run-time for the current network conditions. Using testbed experiments, we show that Oboe significantly improves BOLA, MPC, and a commercially deployed ABR. Oboe also betters a recently proposed reinforcement learning based ABR, Pensieve, by 24% on average on a composite QoE metric, in part because it is able to better specialize ABR behavior across different network states.

CCS CONCEPTS

• **Information systems** → **Information systems applications**; **Multimedia streaming**;

KEYWORDS

Video delivery, Adaptive bitrate algorithms

ACM Reference Format:
Zahaib Akhtar*, Yun Seong Nam*, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. 2018. Oboe: Auto-tuning Video ABR Algorithms to Network Conditions. In *SIGCOMM '18: ACM SIGCOMM 2018 Conference, August 20–25, 2018, Budapest, Hungary.* 15 pages. https://doi.org/10.1145/3230543.3230558

* Both authors contributed equally to this paper and can be contacted at following: zakhtar@usc.edu, nam21@purdue.edu

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230558

1 INTRODUCTION

Internet video forms a major fraction of Internet traffic today [13], and delivering high quality of experience (QoE) is critical since it correlates with user engagement and revenue [6, 23, 31]. To deliver high quality video across diverse network conditions, most Internet video delivery uses adaptive bitrate (ABR) algorithms [32, 48, 59], combined with HTTP chunk-based streaming protocols (*e.g.,* Apple's HTTP Live Streaming, Adobe's HTTP Dynamic Streaming). ABR algorithms (a) chop a video into *chunks*, each of which is encoded at a range of bitrates (or qualities); and (b) choose which bitrate level to fetch a chunk at based on conditions such as the amount of video the client has buffered and the recent throughput achieved by the client. Within this general framework, ABR algorithms differ in how bitrate level selection decisions are made, and these decisions impact metrics such as the average bitrate or the rebuffering ratio. We call these QoE metrics, because they have been shown to correlate well with QoE [23], but other perceptual video quality metrics [2] may also influence QoE.

ABR algorithm design remains an active research area because content providers continue to be interested in *improving the performance of video delivery*. Current ABR algorithms perform well on average, but some users can experience poor delivery performance as measured by the QoE metrics. These users suffer because ABR algorithms *have limited dynamic range*: they do not perform uniformly well across the range of network conditions seen in practice because their parameters are sensitive to throughput variability (§2).

Contributions. In this paper, we present the design of Oboe[1] (§3), a system that takes the first step towards overcoming these hurdles. Oboe improves the dynamic range of ABR algorithms by automatically tuning ABR behavior to

[1] In orchestras all instruments tune to the Oboe.

the current *network state* of a client connection, specifically to throughput and throughput variability.

Oboe's design is based on the observation made by prior work [17, 35, 38, 52, 60] that TCP connections are well-modeled as traversing a piecewise-stationary sequence of network states (§3.1): the connection consists of multiple non-overlapping segments where each segment is in a distinct stationary network state. For each possible network state, Oboe *pre-computes, offline, the best parameter configuration* for a given ABR algorithm (§3.2). It does this by subjecting the algorithm, for each state, to different parameter values, and picking the one that results in the best performance. Then, during video playback, Oboe continuously uses a change-point detection algorithm to detect changes in network state and selects the parameter identified by the offline analysis as best for the current state. Thus, if a video session encounters varying network state during its lifetime, Oboe automatically *specializes* the ABR parameter to each state (§3.3).

We have implemented Oboe and demonstrated several aspects of its performance through testbed experiments and trace driven simulations. First, Oboe significantly improves performance of QoE metrics for three qualitatively different ABR algorithms, one that makes bitrate switching decisions on buffer occupancy alone (BOLA) [48], another that uses both throughput and buffer occupancy (HYB, a widely deployed algorithm), and a third that also optimizes decisions across a finite lookahead horizon (RobustMPC) [59]. In each of these cases, Oboe results in significant improvement. For instance, Oboe reduces sessions with rebuffering from 33.3% to 5.3% relative to RobustMPC while also significantly improving a composite QoE metric.

Oboe, when applied to RobustMPC, also performs significantly better than a newly proposed approach called Pensieve that learns, from real traces (using reinforcement learning), how to adapt to a variety of network conditions. For nearly 80% of the sessions in our dataset, Oboe improves the same composite metric, with benefits exceeding 20% for 25% of the traces. Compared to Oboe, which can specialize parameters to individual network states, Pensieve is unable to specialize across the entire range of network throughputs. We have tried training specialized Pensieve models for different ranges of network throughputs and dynamically switching models based on estimated session throughput. This helps, but a significant gap between the two approaches still remains (§4.4).

While a variety of viable pathways exist to deploying Oboe, we focus on an architecture where Oboe and the entire ABR logic are deployed on the cloud which enables rapid evolution and fine-grain customizability. We show the viability of this architecture with results from a pilot deployment.

2 BACKGROUND AND MOTIVATION

The Internet video delivery ecosystem consists of hundreds of *content publishers* and hundreds of *client side applications* that stream video content to diverse user devices. Publishers, content delivery networks, and users all seek to improve user quality of experience (QoE). There are many factors that affect QoE including start up latency, the average bitrate for a video session, as well as the rebuffering ratio (the percentage of time playback is stalled because of drained buffer) [23]. Video players improve QoE using adaptive bitrate (ABR) algorithms which select bitrates for each chunk while (1) ensuring the bitrate seen by the user is as high as possible and (2) avoiding rebuffering events at the client. Some ABR algorithms may also try to minimize the number of bitrate switches to make the playback smooth.

Content publishers serve different types of content including VoD (Video on Demand) or Live broadcasts. They may also serve streams of different qualities ranging from HD (high definition) to SD (standard definition). These differences impact how they serve videos. For example, publishers who serve VoD content can use player buffers as large as 4 minutes [32], whereas publishers serving live content may have a time-to-live[2] requirement between 15-45 seconds. Similarly, based on the quality of streams they serve, publishers may use different bitrate levels or chunk sizes. Further, publishers may have different QoE objectives. For example, some may strictly prefer to minimize rebuffering and others may relax their tolerance for rebuffering to prioritize higher bitrates. We use the term *publisher specifications* to denote their choice of bitrate levels, chunk sizes, content type, and rebuffering tolerance.

2.1 Background on ABR Algorithms

ABR algorithms fall in two broad categories: (i) those that use both prediction of network throughput and buffer occupancy [34, 51, 59]; and (ii) those that are primarily based on buffer occupancy [32, 48]. Within the above two categories, ABR algorithms can be designed using approaches ranging from heuristics to stochastic optimization. In §4, we discuss a recently proposed ABR algorithm based on a qualitatively different approach, reinforcement learning [39].

MPC: Throughput prediction and buffer occupancy with look-ahead. Selects bitrate by solving an optimization problem. MPC [59] predicts throughput of future chunk downloads based on throughput samples of recently downloaded chunks, then uses this predicted throughput to select bitrates to optimize a given QoE function (§4) over a look-ahead window of 5 future chunks. The aggressive version of the algorithm (FastMPC) directly uses a throughput estimate obtained using a harmonic mean predictor. To compensate for

[2]For live content, the time between the event and its broadcast to users. This bounds the maximum buffer that a player streaming a live event can build.

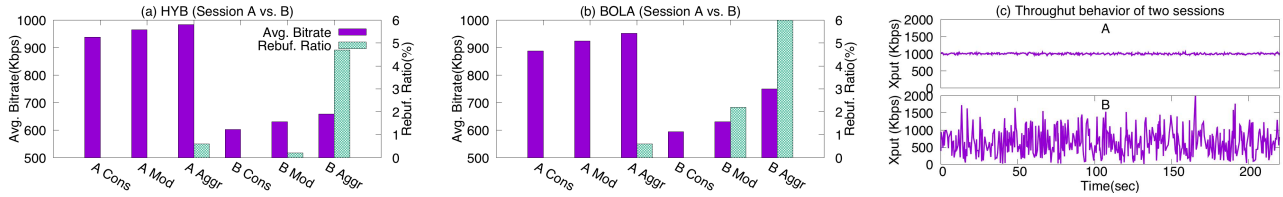

Figure 1—Performance of ABR algorithms using different configurations for two sessions with different throughput behaviors

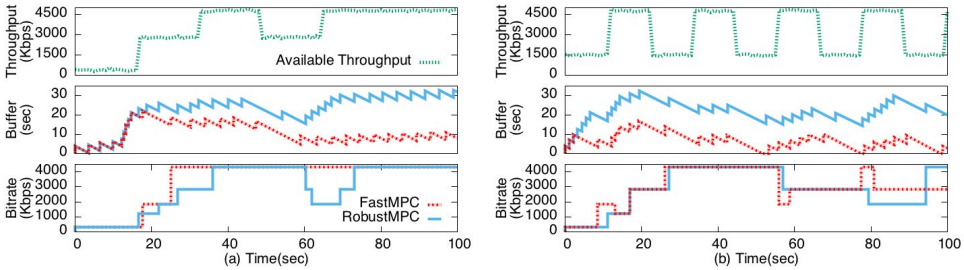

Figure 2—Illustrating how policy for setting discount factors in MPC impacts performance for different traces

throughput prediction errors, a more conservative version, RobustMPC, reduces predicted throughput by a *discount* factor $1+d$, where d is the maximum error in throughput predictions experienced in the last five chunk downloads.

BOLA: Buffer occupancy, selects bitrate by solving an optimization problem. BOLA is a buffer-based algorithm used in Dash.js [7], so it does not employ throughput prediction in making bitrate decisions [48]. It also models bitrate selection as an optimization problem which it solves for a given value of the buffer. It uses a parameter γ which is a ratio of (i) a minimum buffer threshold, below which it downloads the lowest bitrate and (ii) a target buffer threshold which it tries to maintain. Conceptually γ controls how strongly the ABR should avoid rebuffering [48]. Higher values of γ make the algorithm conservative.

HYB: Throughput prediction without lookahead. Selects bitrate using a simple heuristic. An algorithm widely used in production (§5), HYB considers both the predicted throughput and current buffer occupancy (HYB is short for hybrid). For each chunk, HYB picks the highest bitrate that can avoid rebuffering. Specifically, if $S_j(i)$ denotes the size of chunk j encoded at bitrate i, B is the predicted throughput based on past samples, and L the length of the buffer. HYB picks the largest bitrate i such that $\frac{S_j(i)}{B} < L \times \beta$. Here, β can have values between 0 and 1 (higher values represent aggressive ABR behavior). β can be tuned to offset prediction errors in throughput and to compensate for the greedy nature of the approach which may make it susceptible to future buffering events owing to unexpected throughput dips.

2.2 Ensuring High QoE for All Users

Despite widespread deployment, ABR algorithms continue to be an active area of research [32, 34, 39, 48, 51, 59]. This is because, while deployed ABR algorithms work well on average, they do not work uniformly well across all network conditions. A key reason for this is that ABR algorithms have parameters (which we henceforth refer to as *configurations*) that must be set in a manner sensitive to network conditions. ABR algorithms need to run on many different networks, ranging from cellular and WiFi networks at one end, to high-speed broadband connections at the other. Given this diversity, network conditions can vary significantly. Packet loss conditions can vary by an order of magnitude or more across the globe [25]. Network throughputs can also vary widely: for 90% of traces in a large dataset, the trace's maximum throughput is more than twice its average throughput. Yet, unfortunately, most ABR algorithms today either employ fixed configurations or simple heuristics to adapt these configurations (§2.1).

Figures 1(a) and 1(b) show how the choice of ABR configuration depends on network conditions. Figure 1(a) shows the bitrate and rebuffering ratio for two client sessions with the HYB algorithm for three different values of its β parameter, Cons (Conservative), Mod (Moderate), and Aggr (Aggressive). The throughput behavior of the two sessions is shown in Figure 1(c). If a publisher prefers to eliminate rebuffering, Mod is suitable for session A, but Cons is better for session B. Figure 1(b) shows that BOLA behaves similarly, with Mod being the preferred setting for session A and Cons for session B, to avoid rebuffering.

Figures 2(a) and 2(b) show the difficulty in setting the discount factor with MPC, by comparing the performance of FastMPC (no discount factor), and RobustMPC (discount factor set by local heuristic) for two throughput traces with different characteristics. In each figure, the top subgraphs show the available throughput (green curve) and the throughput estimate of FastMPC (red) and RobustMPC (blue). For

the left graph, although the throughput is generally good, the sudden variations force RobustMPC to make overly conservative bitrate decisions, as well as incur more bitrate switches. (bottom subgraph). In contrast, in Figure 2(b), the quicker and more frequent throughput changes (top subgraph) result in FastMPC experiencing rebuffering (middle subgraph), while RobustMPC does not. This is just one example illustrating the difficulty in picking parameters – in our evaluations (§4), we found that RobustMPC was itself too aggressive when selecting discount factors for some traces.

While this section uses synthetic traces for illustrative purposes, our evaluations with real traces (§4) more extensively demonstrate the limitations of current approaches with respect to selecting parameters and the benefits of automatically tuning ABR parameters to network conditions.

3 OBOE DESIGN

Oboe aims to ensure good QoE for all users by enabling ABR algorithms to perform better across a wide range of network conditions. The configurations of many ABR algorithms are sensitive to *network state*, specifically to the value and variability of the available throughput between the client and the video server. For example, β in HYB should be smaller when available throughput is highly variable, while γ in BOLA should be higher. This explains why the algorithms perform differently for different values of parameters on a given client trace (§2.2). However, a line of prior work [17, 35, 38, 52, 60] has observed that network connections are *piecewise stationary*: that is, connections can be in one of several distinct states (§3.1), where each state is distinguished by *stationarity* in the statistical sense (informally, a process is stationary if its statistical properties including mean and variance do not change over time - see [43] for a more formal definition).

Oboe leverages the piecewise stationarity of network connections to address the key challenge of sensitivity of configurations to network conditions. It does so using a two stage design: (a) an *offline* stage where it pre-computes the best configuration choice for each (stationary) network state (§3.2), and (b) and an *online* stage, where during a session, it detects changes in network state and applies the pre-computed best configuration for the current (stationary) state (§3.3). Oboe can also accommodate publisher specifications such as session type (live vs. video-on-demand, time-to-live requirements), bitrate levels or any explicit QoE tradeoffs (*e.g.*, preference between rebuffering and average bitrate) (§3.2), by using these to influence the selection of the best configuration for each (stationary) network state in the offline stage.

3.1 Representing Network State

Most ABR algorithms today adapt bitrates based on the throughput (more precisely, goodput) achieved by recently

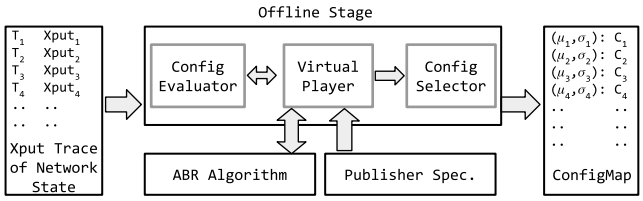

Figure 3—The logical diagram of the offline pipeline used by Oboe

downloaded chunks. This perceived throughput already accounts for network delays and loss-rates, as well as the dynamics of the underlying transport protocol.

The network throughput along a path is *not* necessarily a stationary process [17, 35, 38, 52, 60]: flows at the bottleneck along a path may change over time resulting in changes to available throughput, or the bottleneck itself may shift [35]. An analysis of the throughput traces used in our evaluations (§4) confirms the lack of stationarity when applied to the entire trace. We analyze throughput traces using the Augmented Dickey-Fuller (ADF [26]) test, a hypothesis test to check for stationarity in a time series. Our evaluations on a dataset of 15,000 video streaming throughput traces show that 59.5% were non-stationary (see §4.2 for details of the dataset), implying the presence of distinct mean and/or variance in different segments of the traces.

However, prior work [17, 35, 38, 52, 60] shows that TCP connection throughput *can* be modeled as a *piecewise* stationary process; the connection consists of multiple non-overlapping segments where each segment is stationary and often lasts for tens of seconds or minutes (*e.g.*, Figure 8). Moreover, Zhang et al. [60] show that the throughput in each segment may be modeled as an i.i.d. process.

Motivated by these observations, Oboe defines network state s by a tuple $< \mu_s, \sigma_s >$, where μ_s is the mean and σ_s the standard deviation of the client-perceived throughput in a (stationary) segment of the underlying TCP connection.

3.2 Offline Mapping of Network States

To map network states to their optimal ABR configurations, Oboe uses a pipeline (Figure 3) consisting of three components – the *ConfigEvaluator*, the *VirtualPlayer* and the *ConfigSelector*. The ConfigEvaluator takes a stationary throughput trace as input, which represents a particular network state, and drives the exploration of different ABR configurations over this trace. It does so by using the VirtualPlayer which models the dynamics of an actual video player. The VirtualPlayer interfaces with the ABR algorithm implementation and outputs the performance of different configurations of the ABR. Finally, the ConfigSelector compares the performance of different configurations and builds a *ConfigMap*, which maps a given network state to the best configuration.

Generating throughput traces for ConfigEvaluator. To explore configuration space of an ABR algorithm on each network state s, ConfigEvaluator needs a stationary throughput trace to represent s. To generate such a trace, we explored two different approaches. In one approach, we extracted stationary segments from real traces using offline change point detection ([10], described in §3.3). Change points capture points where the distribution changes. However, because we are not guaranteed *coverage* (i.e., not all states might be observable in real traces), we also explored a second approach which involved generating a *synthetic* trace for each s with s's mean and standard deviation, assuming a Gaussian distribution for the throughput samples. This was motivated by Dinda *et al.* [38] who showed that the throughput of TCP flows of the same size in a given stationary segment may be modeled as a Gaussian distribution (also see §3.1). More recent work also shows that TCP throughput is well modeled as a Markov process, each of whose states may be modeled as a Gaussian distribution [49]. We found that Oboe with synthetic traces performed comparably to stationary segments from real traces. So, ConfigEvaluator uses synthetic traces.

Specifically, ConfigEvaluator quantizes both mean and standard deviation of throughput using a quantum (in our experiments, of 50 Kbps), resulting in states (in our experiments, 10,000), spread over a two dimensional space (in our experiments, 0.05-10 Mbps) of throughput and standard deviation. For each state, we generate a synthetic stationary trace. We found that the benefits of finer quantization are marginal.

Estimating ABR performance with VirtualPlayer and publisher specifications. Oboe uses VirtualPlayer, a trace-based simulator that mimics the behavior of an actual video player without downloading or rendering actual videos. It takes as input a throughput trace and outputs the QoE performance metrics of a video session for a specified ABR algorithm. We have validated VirtualPlayer in §4.7. In designing VirtualPlayer, we have decoupled ABR logic (Figure 3), so the same implementation of the ABR logic can be used in Oboe's offline and online stage. Further, this design provides an interface to the ABR designer through which they can easily integrate their ABR algorithm with Oboe without having to know about Oboe's internals.

The VirtualPlayer also takes into account publisher specifications for bitrate levels, player buffer sizes (determined by time-to-live requirements) and chunk size. These specifications are used by VirtualPlayer when it executes ABR algorithms on the input traces, ensuring that the resulting ConfigMap meets the publisher specifications. Finally, Oboe also allows the publisher to optionally express an explicit QoE tradeoff such as maintaining the rebuffering under a desired threshold $x\%$. Oboe derives a ConfigMap that meets the rebuffering threshold in a best effort manner. We evaluate the efficacy of this flexibility in §4.7.

Building the ConfigMap using ConfigSelector. To build the ConfigMap, the ConfigEvaluator drives the exploration of different configurations for an ABR algorithm. For a given network state s, ConfigEvaluator sweeps through possible configurations of the ABR algorithm using the VirtualPlayer. For example, the β parameter in HYB can take values from 0 to 1, so ConfigEvaluator plays the trace for state s for multiple values of β (quantized for efficiency, see below) in this range.

For each parameter value c_i, VirtualPlayer outputs a *performance vector* $V_i =< v_1, v_2, \ldots v_m >$ where each v_k corresponds to the values achieved by c_i for a QoE metric (*e.g.*, bitrate, rebuffering ratio, and more generally join time and frequency of switching bitrates [23]). This set of performance vectors with the corresponding parameter values are then sent to ConfigSelector for picking the best configuration.

ConfigSelector takes the set of performance vectors and determines the best configuration from them using *vector dominance*. A configuration c_i is said to dominate c_j if V_i element-wise dominates V_j (*i.e.*, each element of V_i is better than or equal to the corresponding element of V_j). This step also takes into account any rebuffering tolerance, and ConfigSelector applies this tolerance to select the maximal performance vector. Deferring the selection of the maximal vector for a given rebuffering tolerance to this stage (instead of filtering vectors in the previous step) is beneficial: it minimizes recomputation by allowing Oboe to quickly compute a new maximal vector if the publisher changes the rebuffering tolerance. At the end of this stage, Oboe obtains the ConfigMap, a complete mapping of each network state to its corresponding optimal ABR configuration.

Two optimizations can be used to quicken the rate of exploration of the ConfigEvaluator. The first is to quantize the parameter sweep, so that configurations are evaluated at a coarser granularity. This trades off some performance for lower computational complexity. The second optimization is based on the observation that there is generally a monotonic relationship between parameter values and the performance. For instance, for HYB (§2.1), the rebuffering ratio and average bitrate are monotonically non-decreasing with the parameter β. Based on this observation, we can instead use an $O(\log n)$ binary search of the configuration space instead of doing a full $O(n)$ sweep of all configurations.

3.3 Online ABR Tuning

Oboe uses the ConfigMap generated offline, and live throughput measurements from the video player to dynamically change ABR configurations during a video playback. It does this by using an *online change point detection algorithm* [14]. This algorithm identifies, in an online fashion, if the distribution of the throughput samples has changed significantly, signaling a state transition. When a change point is detected, the algorithm also provides the new state

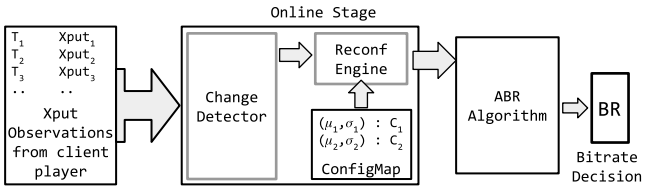

Figure 4—Logical diagram of Oboe's online pipeline

s's mean and standard deviation. Oboe's ChangeDetector (Figure 4) implements the change point detection algorithm, and the ReconfEngine is responsible for updating the ABR configuration based on a new network state and the ConfigMap.

Change point detection algorithms. Such algorithms analyze a time series and check if there are regions in the time series where the underlying distribution of the data changes to a different set of parameters. Offline change-point methods require the full time series to be available, whereas online methods work with a continuous stream of samples as they become available. We focus on online methods, since Oboe identifies change points for an in-progress session and dynamically changes configurations.

While several techniques exist for change point detection [22, 33, 36, 44, 54, 58], we focus on probabilistic methods [14, 18, 24, 57]. Further, we use a Bayesian online probabilistic change-point detector [14] for two reasons. First, in [14], a sequence of observations can be partitioned into non-overlapping states such that the observations are i.i.d. conditioned on a given network state s. This view aligns well with the way we have defined a network state (§3.1). Further, the algorithm is fast and requires no prior knowledge about the data stream, matching our scenario. We use the implementation provided in [10] and integrate it with the ChangeDetector.

Detecting changes in network state. During a video session, ChangeDetector is continually fed with a series of observations of the network throughput, which it uses to detect state changes. ChangeDetector calculates throughput and standard deviation by only considering those samples which belong to the current state. To generate inputs to ChangeDetector, one approach is to use each downloaded chunk to obtain a single throughput sample. However, this may be too coarse-grained, and prevent detection of changes in network state that occur during the chunk download. Instead, we use fine grained samples recorded at periodic intervals (tens of milliseconds) during the download of each chunk. Players such as Dash.js already periodically log intermediate throughput samples during a chunk download, so obtaining these samples does not incur any additional overhead. We only need to modify players to report these samples to Oboe. The set of samples are provided to ChangeDetector after the chunk download, and any change in state is only detected at the end of the chunk download. This is acceptable since any action that can be taken by the ABR algorithm (such as a bit rate switch) only impacts subsequent chunks. In the rarer case that an ABR algorithm abandons the download of a chunk that takes too long, the report is sent when the chunk download is abandoned. §4.8 evaluates the overheads of ChangeDetector.

An alternative approach to changing configurations is to use an exponentially weighted moving average (EWMA) of the mean and standard deviation of throughput samples and to lookup the corresponding configuration. We experimented with such an approach and found its performance unsatisfactory. The approach can result in continual and unnecessary reconfigurations, since throughput may vary across samples even when the network is (statistically) stationary. Damping these changes can result in slow reaction times when a reconfiguration is actually beneficial. In contrast, Oboe (i) models the underlying TCP connection as a sequence of states; (ii) does not make changes to the configuration within a given network state; and (iii) only reconfigures when a state change is observed.

Reconfiguring ABR Algorithm. When a change in the network state is detected, the ChangeDetector signals the change and the new network state s to the ReconfEngine. The ReconfEngine then searches a neighborhood of radius r in the ConfigMap to select the configuration to use for state s. Specifically, if state s is a point in a 2-dimensional space of average throughput and standard deviation of throughput, then it picks the most conservative ABR configuration within a search radius r around s. It does this for two reasons. First, because Oboe quantizes the network states, it might not have precomputed the best configuration for s. Second, the estimated new network state s may have some error, for example, due to inefficiencies in the client download stack [27]. Given these sources of uncertainty, Oboe chooses to be safe in its selection of the best configuration for s. Finally, ReconfEngine configures the ABR algorithm, and the reconfigured ABR algorithm is ready to compute the bitrate decision to be used for the next chunk at this point.

4 EVALUATION

In this section, we demonstrate Oboe's ability to auto-tune three existing algorithms: RobustMPC, BOLA and HYB. We also compare an Oboe-tuned RobustMPC to Pensieve [39].

4.1 Metrics

The performance of a video session depends on multiple factors. *Average bitrate* and *rebuffering ratio* were found to have the most impact on user quality of experience [23], though other factors such as changes in bitrates during a session can play a role [23]. There is no consensus on how to best capture a user's QoE. Consequently, ABR algorithms today are designed to optimize different metrics. For instance, HYB

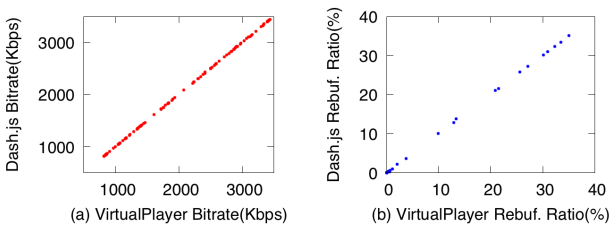

Figure 5—A scatter plot of average bitrate and rebuffering ratio between the VirtualPlayer and real Dash.js player

and BOLA primarily maximize average bitrate subject to low rebuffering. In contrast, other algorithms [39, 59] have been designed to optimize a QoE metric which is a linear combination of bitrate, rebuffering and bitrate changes (smoothness).

With Oboe, our primary evaluation goal is to demonstrate the extent to which it can improve the underlying metrics that an ABR algorithm is designed for. Thus, our evaluations with BOLA and HYB focus on average bitrate and rebuffering, while those with MPC+Oboe focus on the linear combination of QoE (which we refer to as `QoE-lin`, [59]), defined as follows. For a video with N chunks, let R_i be the bitrate chosen for chunk i. Then, the magnitude of bitrate changes M may be defined as $M = \sum_i^{N-1} |R_{i+1} - R_i|$. If the session experiences a total of T seconds of rebuffering, then, $\texttt{QoE-lin}(p, c) = \frac{1}{N} * \sum_i (R_i - pT - c * M)$, where p and c represent scaling penalties applied to rebuffering and changes in the session. This function may be viewed as the session QoE averaged over the number of chunks. For our videos that had a maximum bitrate of 4.3 Mbps, we use $p = 4.3$ and $c = 1$ as our default parameters (following previous work that set default rebuffering penalty equal to the maximum bitrate value [39, 59]).

Even when an algorithm optimizes a metric such as `QoE-lin`, *it is important to understand the distributions of underlying factors*. The underlying factors represent concrete application performance that publishers understand how to reason about. Moreover, a unified metric like `QoE-lin` can obscure important differences. For example, two sessions may have the same `QoE-lin` but different performance in underlying metrics, leading to varied user experience. So, we also present graphs of these metrics.

4.2 Methodology

Implementation. For RobustMPC, we used the implementation available at [11]. Our implementation of BOLA [@bola] is from the Dash.js player. The implementation of HYB is a variant of the algorithm used in a large-scale deployment. These ABR algorithms and Oboe's online stage (change point detection and ABR reconfiguration) run on the server in our experiments. Our client runs the Dash.js video player (version 1.2), a reference player implemented in JavaScript by the MPEG-DASH forum [7]. We modified Dash.js to send client player state information (*e.g.* buffer length, video play state and throughput measurements) to Oboe (§5). This player runs on the Google Chrome browser (version 61) in our experiments. In §5, we show that Oboe can also be run as a cloud service.

Testbed setup. Our evaluations measure ABR performance by delivering a video stream (the "EnvivioDash3" video from the MPEG-DASH reference videos [12]) from a video hosting server to a client, while varying network conditions using throughput traces from real user sessions. We use bitrates $\{300, 750, 1200, 1850, 2850, 4300\} kbps$ with a 4 second chunk duration and total length of 192 seconds. We focus on this video as it has been used in prior work [39], and we do not consider videos of longer duration because we only have throughput traces available for a video publisher that serves short music videos (as we discuss below). The video is hosted on an Apache server. Both the server and client software run on the same 8-core, 4 Ghz, Intel i7 commodity desktop with 12 GB RAM running Ubuntu 16.04. Between server and client, we emulate different network conditions using the Chrome DevTools API [9]. This allows us to control the upload/download throughput as well as latency using the Chrome-Remote-Interface based on throughput traces [5]. We use 571 throughput traces[3] from our dataset (discussed below) for this emulation. All our testbed experiments use a client buffer of 2 minutes.

Datasets. We use throughput traces from real user sessions collected over a three month period. Each trace contains the individual chunk sizes and their download times for on-demand video sessions from a publisher that serves short (4-6 minute) music videos. We derive throughput by dividing the chunk sizes by their download durations. The traces contain sessions that used desktops with wired connections and also sessions on mobile devices using WiFi or cellular connections. Like previous work [39, 59], we primarily focus on traces that have less than 6 Mbps average throughput, since this is the regime where bitrate switching decisions are likely to have QoE impact. We filtered out traces which were too short for playing our entire 192 second video, after which we obtained 5K traces from wired desktops and 4K sessions from WiFi or 3G/4G mobile devices. Our testbed experiments use a subset of 571 traces with roughly the same number of traces sampled from each of desktop and mobile clients.

VirtualPlayer setup. Recall that Oboe uses the VirtualPlayer to obtain a ConfigMap for any ABR algorithm. Since the majority of our results use an actual testbed with the Dash.js player, the benefits of Oboe in our evaluation results already arise despite any inaccuracies in building the ConfigMap on account of using the VirtualPlayer. That said, we have also verified that the VirtualPlayer does a good job

[3] Available at https://github.com/USC-NSL/Oboe

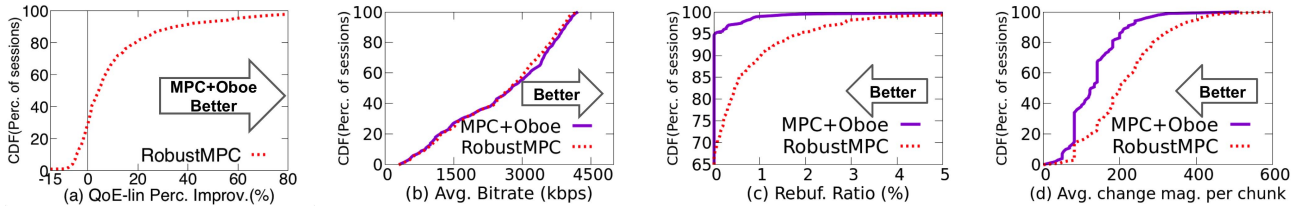

Figure 6—The percentage improvement in QoE-lin of MPC+Oboe over RobustMPC for the Testbed experiment. The distribution of average bitrate, rebuffering ratio and bitrate change magnitude for the schemes is also shown.

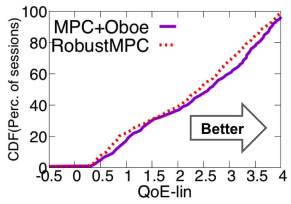

Figure 7—QoE-lin of MPC+Oboe compared to RobustMPC

of tracking the performance of the actual ABR algorithms. For instance, Figure 5(a) and 5(b) demonstrates this for the HYB algorithm. The figures shows the correlation for the average bitrate and rebuffering ratio for 100 throughput traces randomly sampled from our dataset using HYB on the VirtualPlayer compared to using an actual Dash.js player. For both metrics, the graph closely tracks the $y = x$ line indicating close correlation. Given these close correlations, we use the VirtualPlayer in §4.7 to explore Oboe's performance over a larger range of diverse settings and our entire set of traces.

4.3 Oboe with RobustMPC

We now demonstrate that Oboe can be used to auto-tune RobustMPC, the best performing variant of the MPC algorithms. The resulting MPC+Oboe uses the best value of the discount parameter d corresponding to the current network state, replacing RobustMPC's online adaptation based on throughput estimates obtained over the past 5 chunks (§2).

Figure 6(a) shows the CDF of the percentage improvement in QoE-lin of MPC+Oboe over RobustMPC.[4] MPC+Oboe improves QoE-lin for 71% of sessions, with an overall average QoE-lin improvement of 17.62% across all sessions. In particular, for 19% of the sessions, QoE-lin improves by more than 20%. For the sessions MPC+Oboe is unable to improve RobustMPC, its performance degradation is mostly under 8%. Figures 6(b), 6(c) and 6(d) show the constituent QoE metrics. While MPC+Oboe achieves distributionally similar bitrates as RobustMPC as shown in 6(b), it *significantly* reduces rebuffering across sessions: the number of sessions with rebuffering reduces from 33.2% to 5.3%. Further, it also achieves better playback smoothness by improving the median per chunk change magnitude by 38% (Figure 6(d)). Finally, Figure 7 shows the CDF of QoE-lin for MPC+Oboe and RobustMPC, and indicates MPC+Oboe distributionally performs better.

[4]The increase in QoE-lin over RobustMPC relative to the absolute QoE-lin value of RobustMPC expressed as a percentage.

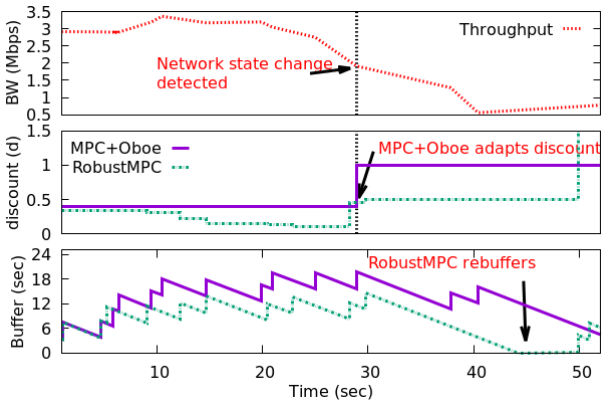

Figure 8—An example session showing how MPC+Oboe is able to outperform RobustMPC by reconfiguring the discount parameter when a network state change is detected.

Figure 8 illustrates, using a single session, why MPC+Oboe performs better than RobustMPC. The top graph shows throughput as a function of time, which includes an initial stable state followed by a drop in throughput. The middle graph shows how the discount factor d of both RobustMPC, and MPC+Oboe vary (the predicted throughput for each system is reduced by a factor of $\frac{1}{1+d}$, where d is shown on the y-axis). During the initial stable state, when prediction errors are low, RobustMPC steadily lowers its discount factor leading to more aggressive bitrate selections (not shown). This results in a *rebuffering event* 44 seconds into the session (lowest graph shows buffer occupancy with 0 indicating a rebuffering event). In contrast, MPC+Oboe does not incur a rebuffering event and maintains a fixed d during the initial stable state. At 29 sec, it detects a change in the network state and adapts its discount factor, leading to more conservative bitrate selections.

4.4 Oboe vs. Pensieve

Pensieve [39] uses deep reinforcement learning [41, 42], a combination of deep learning with reinforcement learning [50], and has been shown to outperform existing ABRs, including RobustMPC [39] in some settings. Since

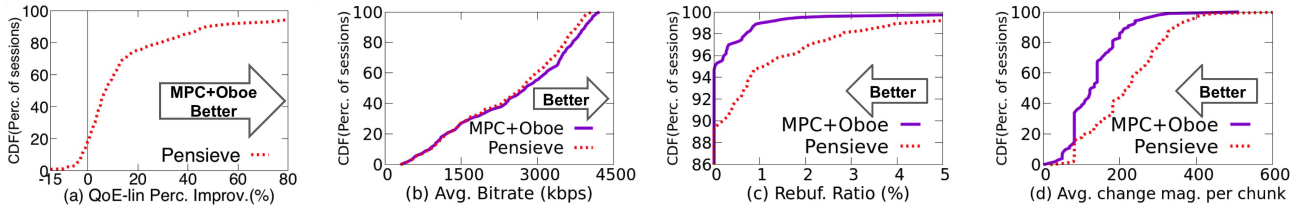

Figure 9—The percentage improvement in `QoE-lin` of MPC+Oboe over Pensieve for the 0-6 Mbps throughput region. The distribution of average bitrate, rebuffering ratio and bitrate change magnitude for the schemes is also shown.

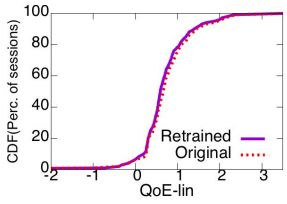

Figure 10—Validation of our training methodology for Pensieve.

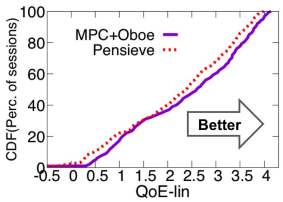

Figure 11—`QoE-lin` of MPC+Oboe compared to Pensieve

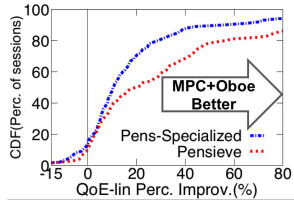

Figure 12—Benefits of specializing Pensieve models. Each curve shows the QoE improvement of MPC+Oboe relative to each Pensieve model.

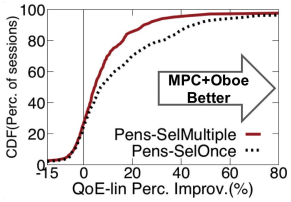

Figure 13—QoE improvement of MPC+Oboe over two ways of dynamically selecting from specialized Pensieve models.

MPC+Oboe outperforms RobustMPC as well, we explore how MPC+Oboe performs relative to Pensieve. Our experiments use the Pensieve implementation provided by the authors [11].

Pensieve Re-Training and Validation. Before evaluating Pensieve on our dataset, we retrain Pensieve using the source code on the trace dataset provided by the Pensieve authors [11]. This helps us validate our retraining given that deep reinforcement learning results are not easy to reproduce [29].

We experimented with five different initial entropy weights in the author suggested range of 1 to 5, and linearly reduced their values in a gradual fashion using plateaus, with five different decrease rates until the entropy weight eventually reached 0.1. This rate scheduler follows best-practice [55]. From the trained set of models, we then selected the best performing model (an initial entropy weight of 1 reduced every 800 iterations until it reaches 0.1 over 100K iterations) and compared its performance to the pre-trained Pensieve model provided by the authors. Figure 10 shows CDFs of `QoE-lin` for the pretrained (Original) model and the model trained by us (Retrained). The performance distribution of the two models are almost identical over the test traces provided by the Pensieve authors, thereby validating our retraining methodology.

Having validated our retraining methodology, we trained Pensieve on *our* dataset with the same complete strategy described above. For this, we pick 1600 traces randomly from our dataset with average throughput in the 0-6 Mbps range. The number of training traces, the number of iterations per trace, and the range of throughput are similar to [39]. We then compare Pensieve and MPC+Oboe over a separate test set of traces also in the range of 0-6 Mbps (§4.2).

Comparison with Pensieve. Figure 9(a) shows the CDF of the percentage improvement in `QoE-lin` for MPC+Oboe over Pensieve. MPC+Oboe outperforms Pensieve for 81% of the sessions, with a `QoE-lin` improvement of 23.9% in average across all sessions. 25% of the sessions achieve more than 20% `QoE-lin` improvement. For the sessions MPC+Oboe is unable to improve over Pensieve, the performance difference is mostly less than 5%. Figures 9(b), 9(c) and 9(d) show that MPC+Oboe distributionally outperforms Pensieve with respect to all underlying metrics. It reduces the number of sessions with rebuffering from 10.7% to 5.3%, reduces the median per chunk change magnitude by 43.9%, and improves median and 95th percentile average bitrate by 2.6% and 4.7% respectively. Finally, Figure 11 shows the CDF of `QoE-lin` for MPC+Oboe and Pensieve, and indicates MPC+Oboe performs distributionally better.

Analyzing Pensieve performance. To understand where these performance improvements were coming from, we examined the relative performance of these two schemes in the 0-3 Mbps range (*i.e.*, traces having an average throughput between 0-3 Mbps). In this more constrained range of network conditions, we found that MPC+Oboe achieves bigger gains over Pensieve (average `QoE-lin` improvement in 0-3 Mbps is 46.23%). We hypothesize that this performance difference stems from the fact that Pensieve builds a single model which does not *specialize* to different throughput ranges.

To test this, we trained a separate Pensieve model only with traces that have an average throughput between 0-3 Mbps range and compared it with MPC+Oboe. Figure 12 shows the per session `QoE-lin` improvement of MPC+Oboe compared to Pensieve models trained for 0-3 Mbps (which we refer to as Pens-Specialized) and for 0-6 Mbps. The median `QoE-lin` improvement with MPC+Oboe over Pens-Specialized is 10.49%, while the median improvement over

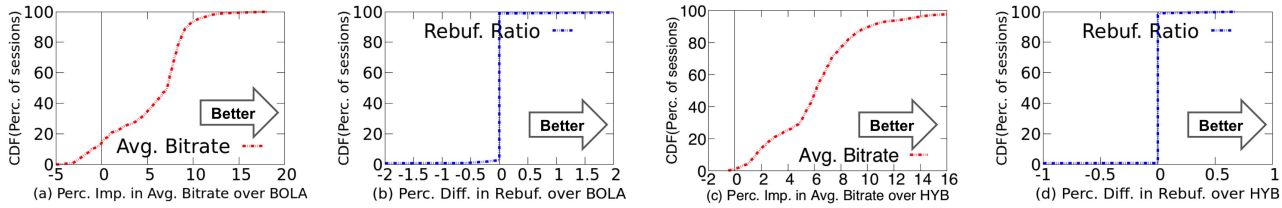

Figure 14—Percentage improvement in bitrate and rebuffering of BOLA+Oboe over BOLA (a),(b) and HYB+Oboe over HYB (c), (d)

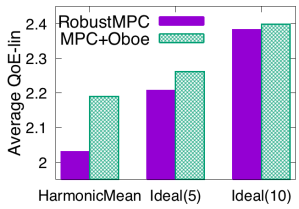

Figure 15—Average QoE-lin of MPC+Oboe with various throughput predictors

Pensieve is 19.9%. This indicates specializing the model does improve Pensieve's performance.

Thus, Pensieve's model is as yet unable to create specialized versions of itself based on the session characteristics. By contrast, Oboe specializes parameters for every network state and therefore performs better. We have also validated Pensieve's inability to specialize in several other ways: building a model for the 3-6 Mbps and showing that it performs better with test data in that range compared to a 0-6 Mbps model; checking that a 0-6 Mbps model performs better for data in that range compared to a 0-100 Mbps model; and ensuring that these results hold even when the training set is doubled. It is hard to pinpoint exactly why Pensieve is unable to learn to be more conservative in the 0-3 Mbps range; deep neural network models remain a black box despite efforts by the machine learning community to make these models more transparent [45], and obtaining such understanding may need further advances in interpretable deep learning models.

A model selector with Pensieve . One way to improve Pensieve's specialization might be to train different models for different throughput ranges and use the model more suited to the network conditions. To test the efficacy of this approach, we used two models (specialized for 0-3 Mbps and 3-6 Mbps), and tried two different *model selectors*. Pens-SelMultiple switches models throughout the session, using the average throughput of the past 5 chunks. Pens-SelOnce starts with the 0-6 Mbps model, selects either the 0-3 Mbps or 3-6 Mbps model based on the average throughput of the first 5 initial chunks, and does not switch thereafter.

Figure 13 shows CDFs of per-session QoE-lin improvement of MPC+Oboe over these selectors. MPC+Oboe is able to outperform both Pens-SelMultile and Pens-SelOnce, with average QoE-lin improvements of 14.2% and 24.32% respectively. Even though one of the model selection schemes offers some improvements over the 0-6 Mbps Pensieve model, the benefits are modest. We hypothesize that this behavior is due to the dynamic selection of distinct Pensieve models, which can interfere with reinforcement learning's decision choices, since, during training, the reinforcement learning algorithm assumes there is no such third party intervention.

4.5 Oboe with other ABR Algorithms

Oboe can also improve other existing ABR algorithms such as BOLA and HYB, which are designed to maximize average bitrate while minimizing rebuffering.

BOLA. BOLA+Oboe tunes γ (§2), which determines how much the algorithm strives to avoid rebuffering. BOLA, as implemented in Dash.js, uses a fixed default value of $\gamma = -10.28$. Figure 14(a) and 14(b) show CDFs of per session performance improvement over BOLA with respect to average bitrate and rebuffering ratio. BOLA+Oboe maintains the rebuffering ratio of BOLA while improving average bitrates for more than 83% of sessions with an overall increase of 7.2% in average across all sessions. For sessions where BOLA+Oboe does not outperform BOLA, its degradation is less than 3.1%.

HYB. The performance of HYB is sensitive to the choice of β parameter, which HYB+Oboe tunes. In production, HYB uses $\beta = 0.25$, determined using A/B tests in a large-scale deployment. Figure 14(c) and 14(d) show CDFs of per session performance improvement of average bitrate and rebuffering ratio over HYB. As with BOLA, HYB+Oboe maintains similar rebuffering ratios as shown in 14(d), but improves bitrates for 98% of sessions with an overall average bitrate improvement of 8.32% in average across all sessions.

4.6 Sensitivity experiments

Alternative throughput traces. To understand how Oboe works on throughput datasets beyond those discussed in §4.2, we evaluated Oboe on two other datasets, FCC [8] and HSDPA [46] that have been used in recent work [39, 59]. FCC is a broadband dataset, while HSDPA contains throughput traces collected from video streaming sessions over 3G networks in Norway using mobile devices. Our comparisons use the traces and a Pensieve model pre-trained for those traces available at [11]. We focus our evaluations on MPC+Oboe and Pensieve, given that Pensieve has been shown to outperform existing ABR schemes including RobustMPC on

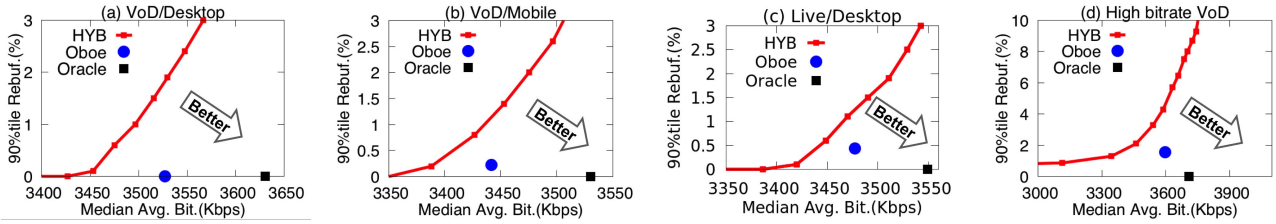

Figure 16—Comparing HYB with multiple fixed configurations and HYB+Oboe for various settings

these traces. Our results show that MPC+Oboe continues to perform better than RobustMPC on these traces. Further, relative to Pensieve, MPC+Oboe improves `QoE-lin` by an average of 6.94% across the FCC dataset and 10.92% across the HSDPA dataset. These improvements are more modest than those in Figure 9(a). The vast majority of traces in the FCC and HSDPA set have an average throughput under 3 Mbps (over 95% for FCC and 98% for HSDPA). The results corroborate Figure 12 which indicates that MPC+Oboe provides more modest gains over Pensieve when the latter is trained and evaluated on datasets with a narrow throughput range. MPC+Oboe provides larger gains in settings like the traces discussed in §4.2, where only 41% traces are under 3 Mbps and 59% are in the 3-6 Mbps range.

Alternative throughput prediction methods. Our experiments with RobustMPC rely on throughput prediction based on the harmonic mean of prior throughput samples (following earlier work [39, 59]), with Oboe tuning the configuration to compensate for prediction errors. We next consider if Oboe's benefits hold if RobustMPC were to have more accurate throughput predictions, potentially by using alternate prediction methods [49]. Rather than using a specific prediction technique, we consider an ideal (and unachievable) approach that we denote as Ideal(T), which can exactly predict the average throughput over the next T seconds. Our experiments were conducted in simulation, using the VirtualPlayer, and the testbed experiment traces (§4.2).

Figure 15 shows the average `QoE-lin` across the traces for RobustMPC and MPC+Oboe using both the default harmonic mean approach and Ideal(T) for different values of T. Although RobustMPC performs better with an ideal predictor, Oboe still provides benefits, achieving an average improvement in `QoE-lin` of 6.34% for Ideal(5) and of 1.8% for Ideal(10), compared to a 16.1% improvement with the harmonic mean estimator. While the magnitude of benefits is smaller with the ideal prediction approach, in practice Oboe will likely result in larger benefits, since even more sophisticated schemes [49] cannot achieve the ideal predictions, and the errors are likely to grow with larger T.

Oboe can improve performance over RobustMPC even when an Ideal(T) prediction method is used for two reasons. First, T may not match the duration of chunk downloads with RobustMPC, which depends on the exact sequence of bitrates chosen during the look-ahead window. The duration is not known a priori, since RobustMPC itself determines the bitrates based on a provided prediction. Second, the decisions made by RobustMPC are over a small look-ahead window, which may not guarantee optimality over the entire session duration.

4.7 Oboe Across Various Settings

In §4.5 we have shown that Oboe outperforms other ABR algorithms when compared to their default configurations. We now explore, for HYB, whether Oboe outperforms all parameter settings of HYB and whether it can tune ABRs based on content type and publisher specifications. For these experiments, we use the VirtualPlayer described in §3.2.

Comparison against all fixed configurations. To explore different fixed configurations, we run HYB with 10 different fixed βs and compare with HYB+Oboe. We summarize the performance for each configuration by considering the (i) median of the average bitrate and the (ii) 90th percentile of the rebuffering ratio across test traces. In this experiment, we also consider an *Oracle* which is the best fixed configuration for each throughput trace with respect to two metrics that HYB tries to optimize.

Figure 16(a) and 16(b) compare HYB, HYB+Oboe and Oracle over desktop, and mobile traces respectively. While Oracle and HYB+Oboe are depicted as single dots since their performance is uniquely determined, we present a frontier for HYB that shows its performance for different fixed configuration. Figure 16(a) shows that HYB+Oboe outperforms HYB in the sense that there is no fixed configuration for HYB that does better than HYB+Oboe performance. HYB+Oboe improves the average bitrates of the median session by 3.2%, while achieving similar rebuffering ratio. Alternately, it reduces the 90th percentile rebuffering ratio from 1.9% to 0%, while maintaining similar bitrates. A similar result holds for mobile traces (Figure 16(b)). Thus, even if publishers were to find the best fixed parameter choice for HYB, Oboe would outperform that choice because it dynamically adapts the parameters.

Comparison under different publisher specifications. Our results so far are for a VoD (video on demand) setting with a maximum buffer size of 2 minutes. Figure 16(c)

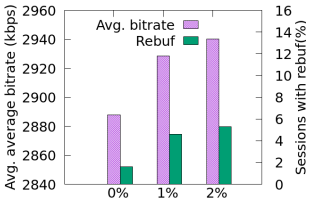

Figure 17—Avg. of avg. bitrate and fraction of sessions with rebuffering for HYB+Oboe and different publisher preferences

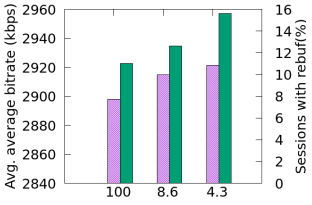

Figure 18—Avg. of avg. bitrate and fraction of sessions with rebuffering for RobustMPC and different publisher preferences

depicts performance for *live video* (which uses a maximum buffer size of 20 seconds to mimic live settings). HYB+Oboe outperforms HYB for this setting, though we note that the bitrate of both approaches degrades relative to the VoD setting since the baseline HYB switches to higher bitrates more conservatively owing to the smaller buffer sizes.

Finally, Figure 16(d) depicts performance for higher bitrate levels ($\{1002, 1434, 2738, 3585, 4661, 5886\} kbps$) and a chunk size of 5 seconds. Even for these choices, HYB+Oboe outperforms HYB, demonstrating its ability to adapt to different publisher specification.

Accommodating publisher's rebuffering tolerance. Oboe allows the publisher to optionally specify explicit rebuffering preferences (§3). ABR algorithms such as RobustMPC which use the `QoE-lin` function may permit this indirectly by adjusting `QoE-lin` weights (§2.1). Figure 17 shows the effectiveness of these approaches, showing the average of average bitrates, and the fraction of sessions with rebuffering for HYB+Oboe. As the publisher makes its rebuffering preference stricter (from 2%-0%), HYB+Oboe achieves lower rebuffering ratios close to the target rebuffering tolerance. In contrast, Figure 18 shows that RobustMPC is less effective at controlling rebuffering by adjusting its rebuffering penalty when the weight on the rebuffering term is varied between 100 (strictly avoid rebuffering) to 4.3.[5] We find that even with a very high rebuffering penalty of 100, RobustMPC causes rebuffering in 11% of the sessions. This shows the benefit of Oboe's approach which gives direct control over the underlying metrics.

4.8 Oboe Overhead

Computing the ConfigMap incurs a one-time cost, since the map can be reused across all clients once the it is built. Computing the best parameter configuration for one network state takes about 12 seconds on a single core. This task is perfectly parallelizable, so computing 10K network states (§3) will take approximately 3.5 hours to explore with two machines of 48 cores each. We have also analyzed the processing overhead incurred by the ChangeDetector module of Oboe. We measure the time taken by ChangeDetector for every decision

[5] We used a change penalty of 0 for fair comparison.

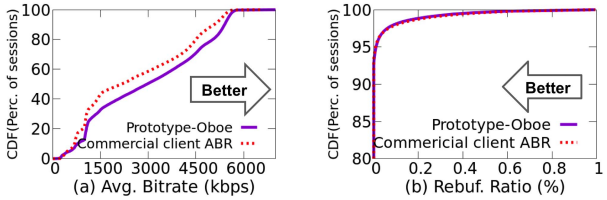

Figure 19—Comparing prototype Oboe with commercial client side ABR implementation in average bitrate and rebuffering ratio.

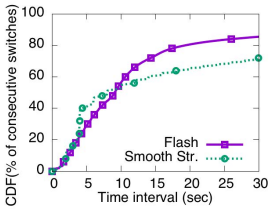

Figure 20—Time between consecutive bitrate switches for two commercial ABRs

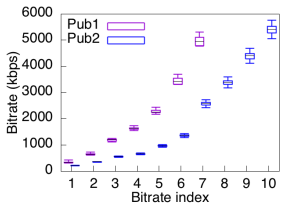

Figure 21—Variance in bitrate levels across videos from two content publishers.

cycle across our experiments, and the measurement indicates that the median processing time of ChangeDetector is around 14 ms. Since each decision is made at a chunk boundary and chunks are 4 seconds, ChangeDetector accounts for less than 0.35% overhead.

5 DEPLOYMENT CONSIDERATIONS

The offline stage of Oboe can be run on the cloud, but several choices exist for the online stage, ranging from embedding the online stage entirely in the client player, or moving some or all of the online stage to the cloud. In our implementation, Oboe's components run on the server side. This mimics a cloud implementation, which has the benefits of other cloud software: fast update deployment, device independence, etc. [4]. We leave a detailed comparison of these choices to future work, but explore, in this section, the *feasibility* of running the online stage on the cloud.

To this end, we have implemented a restricted version of HYB+Oboe on AWS. This limited version of Oboe implements HYB and incorporates tuning based on publisher specifications but not network state. In our implementation, a client player periodically reports player state (such as buffer length and current bitrate) and throughput samples to a Oboe cloud server and receives bitrate decisions in return. For 10 player features and two chunk downloads per second, the communication overhead is 6.4 Kbps, negligibly small compared to the size of video chunks. Figures 19(a) and 19(b) compare the performance of this implementation against a client player running HYB over 20K sessions collected during a two-week pilot deployment. Oboe is comparable in performance to the client side player and even improves bitrate slightly (because it was tuned to this publisher's specification).

We expected a cloud implementation would perform worse because of the latency induced by client-server communication. However, we found that most of the bitrate switching decisions occur on timescales much longer than the client-server latencies. Figure 20 shows the CDF of the time interval between consecutive bitrate switches for ABR algorithms in two widely used video players(Adobe's Flash [3] and Microsoft Smooth Streaming [1]). The figure shows that over 95% of switching decisions occur at intervals higher than 1 second for both players. This suggests that a cloud-based deployment is viable.

6 DISCUSSION AND FUTURE WORK

Performance improvements for all sessions. As our results (*e.g.*, Figure 6(a)) show, Oboe improves the performance for most but not all sessions relative to the ABR algorithm it tunes. For instance, after inspecting the results in §4.3, we have found that MPC+Oboe typically improves performance relative to RobustMPC by reducing rebuffering and/or the magnitude of bitrate changes, but at the expense of slightly lower bitrates. The resulting `QoE-lin` is improved for most sessions, indicating Oboe does a good job of properly balancing the various factors, but some sessions see lower `QoE-lin`. More generally, designing an ABR approach that can optimize the performance of all sessions is a hard problem that needs more research.

Sharing ConfigMap across videos. Oboe need not perform offline precomputation for each individual video, as it can use a single ConfigMap for a *class* of videos that follow a similar bitrate encoding scheme. Figure 21 shows that two popular video publishers use similar encoding schemes across two thousand videos each. Publisher 1 uses 7 distinct bitrate levels, and the coefficient of variance across bitrates within each level is only 0.13, while Publisher2, uses 10 distinct bitrate levels, and the coefficient of variance across bitrates within each level is only 0.067. This indicates the potential to share a single ConfigMap across videos.

Generality of Oboe. While we have shown that Oboe can tune a variety of configuration parameters across several ABR algorithms, whether Oboe can tune all algorithms and all parameters is an open question. It is unclear if Oboe can directly augment Pensieve, since a model learned by reinforcement learning may not interact well with intermediaries such as Oboe. However, combining the benefits of Oboe and Pensieve in other ways is an interesting avenue for future work.

7 RELATED WORK

Tuning ABR Algorithm Configurations. The BBA2 algorithm [32] tunes its lower reservoir based on buffer occupancy dynamics, while MPC [59] adapts its throughput discount factor based on past prediction errors (§2). In contrast to such ad-hoc heuristics, Oboe selects configuration parameters based on network state, and publisher specifications. The approach is generically applicable to many ABR algorithms. Newer congestion control protocols like BBR [19] estimate network throughput, which if exposed, could benefit Oboe.

Learning ABR Algorithms. Among ABR algorithms that use Reinforcement Learning and other machine learning techniques [20, 21, 39, 40, 53], Pensieve [39] has been shown to perform the best. While Pensieve does not specialize to different throughput regimes, Oboe performs better by specializing parameter values for each network state independently.

Other work in self-tuning. Beyond ABR algorithms, self-tuning approaches have been explored in other contexts. Winstein et al. [56] used simulations to determine TCP parameters for different settings, while Semke et al. [47] proposed tuning TCP socket buffers to ensure high throughput. More generally, Google Vizier [28] performs such black-box tuning as a service. While Vizier can potentially be used to implement the offline phase of Oboe, our work identifies underlying principles (such as the piecewise stationarity of available throughput) that forms the basis for the tuning.

Video QoE. Several researchers have pointed out that sub-optimal ABR performance can significantly impact user-engagement and hence revenue [16, 37]. Others have looked at quality issues that occur when multiple players start to compete for bandwidth [15, 30, 31, 34] In contrast, Oboe improves the QoE performance of several ABR algorithms across a range of different network conditions by automatically tuning their parameters.

8 CONCLUSION

Oboe is a system for automatically tuning ABR algorithms by adapting ABR configurations in realtime to match the current network state. Picking configurations in a manner informed by network state and publisher preferences distinguishes Oboe's approach from heuristics used today that do not consider these factors. Oboe significantly improves the performance of BOLA, HYB and RobustMPC; further, for nearly 80% of the sessions in our dataset, Oboe integrated with RobustMPC improves `QoE-lin` relative to Pensieve and the improvements exceed 20% for 25% of the sessions.

Acknowledgments. We thank our shepherd, Mohammad Alizadeh and the anonymous reviewers for their constructive feedback. We thank Oleg White, Yan Li and Shubo Liu for their assistance obtaining the throughput data and for helpful discussions. This work was funded in part by the National Science Foundation (NSF) Awards CNS-1618921, CNS-1564242, and CNS-1413978; and the ARO, under the U.S. Army Research Laboratory award W911NF-09-2-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ARO.

BIBLIOGRAPHY

[1] Microsoft Smooth Streaming. http://www.iis.net/downloads/microsoft/smooth-streaming.

[2] Toward A Practical Perceptual Video Quality Metric. https://medium.com/netflix-techblog/toward-a-practical-perceptual-video-quality-metric-653f208b9652.

[3] Adobe OSMF player. http://www.osmf.org.

[4] Oracle: 5 Reasons to Consider SaaS for Your Business Applications. http://www.oracle.com/us/solutions/cloud/saas-business-applications-1945540.pdf.

[5] Chrome Remote Interface. https://github.com/cyrus-and/chrome-remote-interface.

[6] Cisco: It Came to Me in a Stream. https://www.cisco.com/web/about/ac79/docs/sp/Online-Video-Consumption_Consumers.pdf.

[7] DASH Industry Forum. https://github.com/Dash-Industry-Forum/dash.js.

[8] Federal Communications Commission. Raw Data - Measuring Broadband America. www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-2016.

[9] Google-Chrome: Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/tot/Network/.

[10] Bayesian Changepoint Detection. https://github.com/hildensia/bayesian_changepoint_detection.

[11] Pensieve. https://github.com/hongzimao/pensieve.

[12] DASH Industry Forum. https://dash.akamaized.net/envivio/EnvivioDash3.

[13] Sandvine: Global Internet phenomena report . https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/2h-2014-global-internet-phenomena-report.pdf.

[14] Ryan Prescott Adams and David JC MacKay. Bayesian Online Changepoint Detection. In *arXiv:0710.3742v1*, 2007.

[15] Saamer Akhshabi, Lakshmi Anantakrishnan, Ali C Begen, and Constantine Dovrolis. What Happens when HTTP Adaptive Streaming Players Compete for Bandwidth? In *the International Workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV, 2012.

[16] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a Predictive Model of Quality of Experience for Internet Video. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2013.

[17] Hari Balakrishnan, Mark Stemm, Srinivasan Seshan, and Randy H Katz. Analyzing Stability in Wide-area Network Performance. *ACM SIGMETRICS Performance Evaluation Review*, 25:2–12, 1997.

[18] Daniel Barry and John A Hartigan. A Bayesian Analysis for Change Point Problems. *Journal of the American Statistical Society*, 88(421):309–319, 1993.

[19] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14: 20–53, 2016.

[20] Federico Chiariotti, Stefano D'Aronco, Laura Toni, and Pascal Frossard. Online Learning Adaptation Strategy for DASH Clients. In *Proceedings of the International Conference on Multimedia Systems*, MMSys, 2016.

[21] Maxim Claeys, Steven Latré, Jeroen Famaey, Tingyao Wu, Werner Van Leekwijck, and Filip De Turck. Design and Optimisation of a (FA)Q-learning-based HTTP Adaptive Streaming Client. *Connection Science*, 26(1):25–43, 2014.

[22] Frédéric Desobry, Manuel Davy, and Christian Doncarli. An Online Kernel Change Detection Algorithm. *IEEE Transactions on Signal Processing*, 53(8): 2961–2974, 2005.

[23] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the Impact of Video Quality on User Engagement. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2011.

[24] Paul Fernhead. Exact and Efficient Bayesian Inference for Multiple Changepoint Problems. *Statistics and Computing*, 16(2):203–213, 2006.

[25] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. An Internet-Wide Analysis of Traffic Policing. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2016.

[26] Wayne A Fuller. *Introduction to Statistical Time Series*. John Wiley and Sons, 1976.

[27] Mojgan Ghasemi, Partha Kanuparthy, Ahmed Mansy, Theophilus Benson, and Jennifer Rexford. Performance Characterization of a Commercial Video Streaming Service. In *Proceedings of the ACM Conference on Internet Measurement Conference*, IMC, 2016.

[28] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, SIGKDD, 2017.

[29] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning that Matters. In *Proceedings of the Association for Advancement of Artificial Intelligence*, AAAI, 2018.

[30] Rémi Houdaille and Stéphane Gouache. Shaping HTTP Adaptive Streams for a Better User Experience. In *Proceedings of the Multimedia Systems Conference*, MMSys, 2012.

[31] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard. In *Proceedings of the ACM Conference on Internet Measurement Conference*, IMC, 2012.

[32] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2014.

[33] Daniel R. Jeske, Veronica Montes De Oca, Wolfgang Bischoff, and Mazda Marvasti. Cusum Techniques for Timeslot Sequences with Applications to Network Surveillance. *Computational Statistics and Data Analysis*, 53:4332–4344, 2009.

[34] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proceedings of the ACM International Conference on Emerging Networking Experiments and Technologies*, CoNEXT, 2012.

[35] James Jobin, Michalis Faloutsos, Satish K Tripathi, and Srikanth V Krishnamurthy. Understanding the Effects of Hotspots in Wireless Cellular Networks. In *Proceedings of the Conference of the IEEE Computer and Communications Societies*, INFOCOM, 2004.

[36] Eamonn J. Keogh, Selina Chu, David Hart, and Michael J. Pazzani. An Online Algorithm for Segmenting Time Series. In *Proceedings of the IEEE International Conference on Data Mining*, ICDM, 2001.

[37] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-experimental Designs. In *Proceedings of the ACM Conference on Internet Measurement Conference*, IMC, 2012.

[38] Dong Lu, Yi Qiao, Peter A Dinda, and Fabian E Bustamante. Characterizing and Predicting TCP Throughput on the Wide Area Network. In *IEEE International Conference on Distributed Computing Systems*, ICDCS, 2005.

[39] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2017.

[40] Virginia Martín, Julián Cabrera, and Narciso García. Design, Optimization and Evaluation of a Q-learning HTTP Adaptive Streaming Client. *IEEE Transactions on Consumer Electronics*, 62(4):380–388, 2016.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level Control Through Deep Reinforcement Learning. *Nature*, 518(7540):529–533, 2015.

[42] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the International Conference on Machine Learning*, ICML, 2016.

[43] Hossein Pishro-Nik. *Introduction to Probability, Statistics and Random Processes*. Kappa Research, 2014.

[44] Thanawin Rakthanmanon, Eamonn J. Keogh, Stefano Lonardi, and Scott Evans. Time Series Epenthesis: Clustering Time Series Streams Requires Ignoring Some Data. In *Proceedings of the International Conference on Data Mining*, ICML, 2011.

[45] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why Should I Trust You?: Explaining the Predictions of Any Classifier. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, SIGKDD, 2016.

[46] Haakon Riiser, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. Commute Path Bandwidth Traces from 3G Networks: Analysis and Applications. In *Proceedings of the ACM Multimedia Systems Conference*, MMSys, 2013.

[47] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic TCP Buffer Tuning. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 1998.

[48] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. BOLA: Near-optimal Bitrate Adaptation for Online Videos. In *Proceedings of the IEEE International Conference on Computer Communications*, INFOCOM, 2016.

[49] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. CS2P: Improving Video Bitrate Selection and Adaptation with Data-Driven Throughput Prediction. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2016.

[50] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[51] Guibin Tian and Yong Liu. Towards Agile and Smooth Video Adaptation in Dynamic HTTP Streaming. In *the ACM International Conference on Emerging Networking Experiments and Technologies*, CoNEXT, 2012.

[52] Guillaume Urvoy-Keller. On the Stationarity of TCP Bulk Data Transfers. In *Proceedings of the Passive and Active Measurement Conference*, PAM, 2005.

[53] Jeroen van der Hooft, Stefano Petrangeli, Maxim Claeys, Jeroen Famaey, and Filip De Turck. A Learning-based Algorithm for Improved Bandwidth-awareness of Adaptive Streaming Clients. In *Symposium on Integrated Network Management*, IM, 2015.

[54] Li Wei and Eamonn Keogh. Semi-supervised Time Series Classification. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, SIGKDD, 2006.

[55] Ronald J Williams and Jing Peng. Function Optimization using Connectionist Reinforcement Learning Algorithms. *Connection Science*, 3(3):241–268, 1991.

[56] Keith Winstein and Hari Balakrishnan. TCP Ex Machina: Computer-generated Congestion Control. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2013.

[57] Xuan Xiang and Kevin Murphy. Modelling Changing Dependency Structure in Multivariate Time Series. In *Proceedings of the International Conference on Data Mining*, ICML, 2007.

[58] Kenji Yamanishi and Jun-ichi Takeuchi. A Unifying Framework for Detecting Outliers and Change Points from Non-stationary Time Series Data. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, SIGKDD, 2002.

[59] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2015.

[60] Yin Zhang and Nick Duffield. On the Constancy of Internet Path Properties. In *Proceedings of the ACM SIGCOMM Workshop on Internet Measurement*, 2001.