Learning *in situ*: a randomized experiment in video streaming

Francis Y. Yan and Hudson Ayers, *Stanford University;* Chenzhi Zhu, *Tsinghua University;* Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein, *Stanford University*

https://www.usenix.org/conference/nsdi20/presentation/yan

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Learning *in situ*: a randomized experiment in video streaming

Francis Y. Yan   Hudson Ayers   Chenzhi Zhu[†]   Sadjad Fouladi
James Hong   Keyi Zhang   Philip Levis   Keith Winstein

Stanford University, [†]Tsinghua University

Abstract

We describe the results of a randomized controlled trial of video-streaming algorithms for bitrate selection and network prediction. Over the last year, we have streamed 38.6 years of video to 63,508 users across the Internet. Sessions are randomized in blinded fashion among algorithms.

We found that in this real-world setting, it is difficult for sophisticated or machine-learned control schemes to outperform a "simple" scheme (buffer-based control), notwithstanding good performance in network emulators or simulators. We performed a statistical analysis and found that the heavy-tailed nature of network and user behavior, as well as the challenges of emulating diverse Internet paths during training, present obstacles for learned algorithms in this setting.

We then developed an ABR algorithm that robustly outperformed other schemes, by leveraging data from its deployment and limiting the scope of machine learning only to making predictions that can be checked soon after. The system uses supervised learning *in situ*, with data from the real deployment environment, to train a probabilistic predictor of upcoming chunk transmission times. This module then informs a classical control policy (model predictive control).

To support further investigation, we are publishing an archive of data and results each week, and will open our ongoing study to the community. We welcome other researchers to use this platform to develop and validate new algorithms for bitrate selection, network prediction, and congestion control.

1 Introduction

Video streaming is the predominant Internet application, making up almost three quarters of all traffic [41]. One key algorithmic question in video streaming is *adaptive bitrate selection*, or ABR, which decides the compression level selected for each "chunk," or segment, of the video. ABR algorithms optimize the user's quality of experience (QoE): more-compressed chunks reduce quality, but larger chunks may stall playback if the client cannot download them in time.

In the academic literature, many recent ABR algorithms use statistical and machine-learning methods [4, 25, 38–40, 46], which allow algorithms to consider many input signals and try to perform well for a wide variety of clients. An ABR decision can depend on recent throughput, client-side buffer occupancy, delay, the experience of clients on similar ISPs or types of connectivity, etc. Machine learning can find patterns in seas of data and is a natural fit for this problem domain.

However, it is a perennial lesson that the performance of learned algorithms depends on the data or environments used to train them. ML approaches to video streaming and other wide-area networking challenges are often hampered in their access to good and representative training data. The Internet is complex and diverse, individual nodes only observe a noisy sliver of the system dynamics, and behavior is often heavy-tailed and changes with time. Even with representative throughput traces, accurately simulating or emulating the diversity of Internet paths requires more than replaying such traces and is beyond current capabilities [15, 16, 31, 45].

As a result, the performance of algorithms in emulated environments may not generalize to the Internet [7]. For example, CS2P's gains were more modest over real networks than in simulation [40]. Measurements of Pensieve [25] saw narrower benefits on similar paths [11] and a large-scale streaming service [24]. Other learned algorithms, such as the Remy congestion-control schemes, have also seen inconsistent results on real networks, despite good results in simulation [45].

This paper seeks to answer: *what does it take to create a learned ABR algorithm that robustly performs well over the wild Internet?* We report the design and findings of Puffer[1], an ongoing research study that operates a video-streaming website open to the public. Over the past year, Puffer has streamed 38.6 years of video to 63,508 distinct users, while recording client telemetry for analysis (current load is about 60 stream-days of data per day). Puffer randomly assigns each session to one of a set of ABR algorithms; users are blinded to the assignment. We find:

[1]https://puffer.stanford.edu

In our real-world setting, sophisticated algorithms based on control theory [46] or reinforcement learning [25] did not outperform simple buffer-based control [18]. We found that more-sophisticated algorithms do not necessarily beat a simpler, older algorithm. The newer algorithms were developed and evaluated using throughput traces that may not have captured enough of the Internet's heavy tails and other dynamics when replayed in simulation or emulation. Training them on more-representative traces doesn't necessarily reverse this: we retrained one algorithm using throughput traces drawn from Puffer (instead of its original set of traces) and evaluated it also on Puffer, but the results were similar (§5.3).

Statistical margins of error in quantifying algorithm performance are considerable. Prior work on ABR algorithms has claimed benefits of 10–15% [46], 3.2–14% [40], or 12–25% [25], based on throughput traces or real-world experiments lasting hours or days. However, we found that the empirical variability and heavy tails of throughput evolution and rebuffering create statistical margins of uncertainty that make it challenging to detect real effects of this magnitude. Even with a *year* of experience per scheme, a 20% improvement in rebuffering ratio would be statistically indistinguishable, i.e., below the threshold of detection with 95% confidence. These uncertainties affect the design space of machine-learning approaches that can practically be deployed [13, 26].

It is possible to robustly outperform existing schemes by combining classical control with an ML predictor trained *in situ* on real data. We describe Fugu, a data-driven ABR algorithm that combines several techniques. Fugu is based on MPC (model predictive control) [46], a classical control policy, but replaces its throughput predictor with a deep neural network trained using supervised learning on data recorded *in situ* (in place), meaning from Fugu's actual deployment environment, Puffer. The predictor has some uncommon features: it predicts *transmission time* given a chunk's file size (vs. estimating throughput), it outputs a probability distribution (vs. a point estimate), and it considers low-level congestion-control statistics among its input signals. Ablation studies (§4.2) find each of these features to be necessary to Fugu's performance.

In a controlled experiment during most of 2019, Fugu outperformed existing techniques—including the simple algorithm—in stall ratio (with one exception), video quality, and the variability of video quality (Figure 1). The improvements were significant both statistically and, perhaps, practically: users who were randomly assigned to Fugu (in blinded fashion) chose to continue streaming for 5–9% longer, on average, than users assigned to the other ABR algorithms.[2]

Our results suggest that, as in other domains, good and representative training is the key challenge for robust performance of learned networking algorithms, a somewhat different point of view from the generalizability arguments in prior work [25, 37, 44]. One way to achieve representative training is to learn in place (*in situ*) on the actual deployment environment, assuming the scheme can be feasibly trained this way and the deployment is widely enough used to exercise a broad range of scenarios.[3] The approach we describe here is only a step in this direction, but we believe Puffer's results suggest that learned systems will benefit by addressing the challenge of "*how will we get enough representative scenarios for training—what is enough, and how do we keep them representative over time?*" as a first-class consideration.

We intend to operate Puffer as an "open research" project as long as feasible. We invite the research community to train and test new algorithms on randomized subsets of its traffic, gaining feedback on real-world performance with quantified uncertainty. Along with this paper, we are publishing an archive of data and results back to the beginning of 2019 on the Puffer website, with new data and results posted weekly.

In the next few sections, we discuss the background and related work on this problem (§2), the design of our blinded randomized experiment (§3) and the Fugu algorithm (§4), with experimental results in Section 5, and a discussion of results and limitations in Section 6. In the appendices, we provide a standardized diagram of the experimental flow for the primary analysis and describe the data we are releasing.

Results of primary experiment (Jan. 26–Aug. 7 & Aug. 30–Oct. 16, 2019)

Algorithm	Time stalled (lower is better)	Mean SSIM (higher is better)	SSIM variation (lower is better)	Mean duration (time on site)
Fugu	0.13%	16.64 dB	0.74 dB	33.6 min
MPC-HM [46]	0.22%	16.61 dB	0.79 dB	30.8 min
BBA [18]	0.19%	16.56 dB	1.11 dB	32.1 min
Pensieve [25]	0.17%	16.26 dB	1.05 dB	31.6 min
RobustMPC-HM	0.12%	16.01 dB	0.98 dB	31.0 min

Figure 1: In an eight-month randomized controlled trial with blinded assignment, the Fugu scheme outperformed other ABR algorithms. The primary analysis includes 637,189 streams played by 54,612 client IP addresses (13.1 client-years in total). Uncertainties are shown in Figures 9 and 11.

2 Background and related work

The basic problem of adaptive video streaming has been the subject of much academic work; for a good overview, we refer the reader to Yin et al. [46]. We briefly outline the problem here. A service wishes to serve a pre-recorded or live video stream to a broad array of clients over the Internet. Each client's connection has a different and unpredictable time-varying performance. Because there are many clients, it is not feasible for the service to adjust the encoder configuration in real time to accommodate any one client.

[2]This effect was driven solely by users streaming more than 3 hours of video; we do not fully understand it.

[3]Even collecting traces from a deployment environment and replaying them in a simulator or emulator to train a control policy—as is typically necessary in reinforcement learning—is not what we mean by "*in situ*."

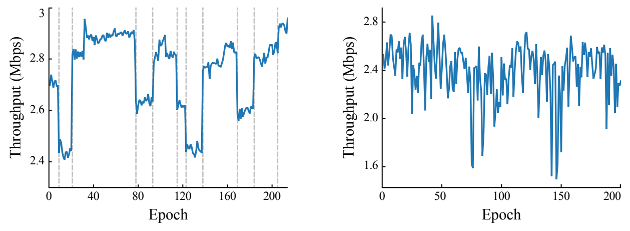
Instead, the service encodes the video into a handful of alternative compressed versions. Each represents the original video but at a different quality, target bitrate, or resolution. Client sessions choose from this limited menu. The service encodes the different versions in a way that allows clients to switch midstream as necessary: it divides the video into *chunks*, typically 2–6 seconds each, and encodes each version of each chunk independently, so it can be decoded without access to any other chunks. This gives clients the opportunity to switch between different versions at each chunk boundary. The different alternatives are generally referred to as different "bitrates," although streaming services today generally use "variable bitrate" (VBR) encoding [32], where within each alternative stream, the chunks vary in compressed size [47].

Choosing which chunks to fetch. Algorithms that select which alternative version of each chunk to fetch and play, given uncertain future throughput, are known as *adaptive bitrate* (ABR) schemes. These schemes fetch chunks, accumulating them in a playback buffer, while playing the video at the same time. The playhead advances and drains the buffer at a steady rate, 1 s/s, but chunks arrive at irregular intervals dictated by the varying network throughput and the compressed size of each chunk. If the buffer underflows, playback must stall while the client "rebuffers": fetching more chunks before resuming playback. The goal of an ABR algorithm is typically framed as choosing the optimal sequence of chunks to fetch or replace [38], given recent experience and guesses about the future, to minimize startup time and presence of stalls, maximize the quality of chunks played back, and minimize variation in quality over time (especially abrupt changes in quality). The importance tradeoff for these factors is captured in a QoE metric; several studies have calibrated QoE metrics against human behavior or opinion [6, 12, 21].

Adaptive bitrate selection. Researchers have produced a literature of ABR schemes, including "rate-based" approaches that focus on matching the video bitrate to the network throughput [20, 23, 27], "buffer-based" algorithms that steer the duration of the playback buffer [18, 38, 39], and control-theoretic schemes that try to maximize expected QoE over a receding horizon, given the upcoming chunk sizes and a prediction of the future throughput.

Model Predictive Control (MPC), FastMPC, and RobustMPC [46] fall into the last category. They comprise two modules: a *throughput predictor* that informs a predictive *model* of what will happen to the buffer occupancy and QoE in the near future, depending on which chunks it fetches, with what quality and sizes. MPC uses the model to plan a sequence of chunks over a limited horizon (e.g., the next 5–8 chunks) to maximize the expected QoE. We implemented MPC and RobustMPC for Puffer, using the same predictor as the paper: the harmonic mean of the last five throughput samples.

CS2P [40] and Oboe-tuned RobustMPC [4] are related to MPC; they constitute better throughput predictors that inform



(a) CS2P example session (Figure 4a from [40]) **(b)** Typical Puffer session with similar mean throughput

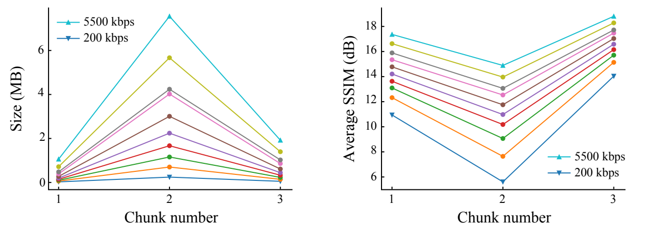
Figure 2: Puffer has not observed CS2P's discrete throughput states. (Epochs are 6 seconds in both plots.)

the same control strategy (MPC). These throughput predictors were trained on real datasets that recorded the evolution of throughput over time within a session. CS2P clusters users by similarity and models their evolving throughput as a Markovian process with a small number of discrete states; Oboe uses a similar model to detect when the network path has changed state. In our dataset, we have not observed CS2P and Oboe's observation of discrete throughput states (Figure 2).

Fugu fits in this same category of algorithms. It also uses MPC as the control strategy, informed by a network predictor trained on real data. This component, which we call the Transmission Time Predictor (TTP), incorporates a number of uncommon features, none of which can claim novelty on its own. The TTP explicitly predicts the transmission time of a chunk with given size and isn't a "throughput" predictor *per se*. A throughput predictor models the transmission time of a chunk as scaling linearly with size, but it is well known that observed throughput varies with file size [7, 32, 47], in part because of the effects of congestion control and because chunks of different sizes experience different time intervals of the path's varying capacity. To our knowledge, Fugu is the first to use this fact operationally as part of a control policy.

Fugu's predictor is also *probabilistic*: it outputs not a single predicted transmission time, but a probability distribution on possible outcomes. The use of uncertainty in model predictive control has a long history [36], but to our knowledge Fugu is the first to use stochastic MPC in this context. Finally, Fugu's predictor is a neural network, which lets it consider an array of diverse signals that relate to transmission time, including raw congestion-control statistics from the sender-side TCP implementation [17, 42]. We found that several of these signals (RTT, CWND, etc.) benefit ABR decisions (§5).

Pensieve [25] is an ABR scheme also based on a deep neural network. Unlike Fugu, Pensieve uses the neural network not simply to make predictions but to make *decisions* about which chunks to send. This affects the type of learning used to train the algorithm. While CS2P and Fugu's TTP can be trained with *supervised learning* (to predict chunk transmission times recorded from past data), it takes more than data to train a scheme that makes decisions; one needs training *envi-*



(a) VBR encoding lets chunk size vary within a stream [47].

(b) Picture quality also varies with VBR encoding [32].

Figure 3: Variations in picture quality and chunk size within each stream suggest a benefit from choosing chunks based on SSIM and size, rather than average bitrate (legend).

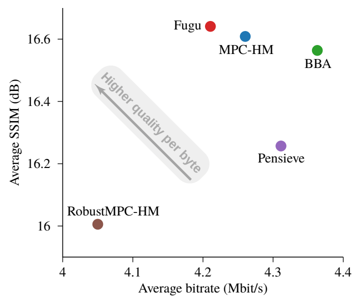


Figure 4: On Puffer, schemes that maximize average SSIM (MPC-HM, RobustMPC-HM, and Fugu) delivered higher quality video per byte sent, vs. those that maximize bitrate directly (Pensieve) or the SSIM of each chunk (BBA).

ronments that respond to a series of decisions and judge their consequences. This is known as reinforcement learning (RL). Generally speaking, RL techniques expect a set of training environments that can exercise a control policy through a range of situations and actions [3], and need to be able to observe a detectable difference in performance by slightly varying a control action. Systems that are challenging to simulate or that have too much noise present difficulties [13, 26].

3 Puffer: an ongoing live study of ABR

To understand the challenges of video streaming and measure the behavior of ABR schemes, we built Puffer, a free, publicly accessible website that live-streams six over-the-air commercial television channels. Puffer operates as a randomized controlled trial; sessions are randomly assigned to one of a set of ABR or congestion-control schemes. The study participants include any member of the public who wishes to participate. Users are blinded to algorithm assignment, and we record client telemetry on video quality and playback. A Stanford Institutional Review Board determined that Puffer does not constitute human subjects research.

Our reasoning for streaming live television was to collect data from enough participants and network paths to draw robust conclusions about the performance of algorithms for ABR control and network prediction. Live television is an evergreen source of popular content that had not been broadly available for free on the Internet. Our study benefits, in part, from a law that allows nonprofit organizations to retransmit over-the-air television signals without charge [1]. Here, we describe details of the system, experiment, and analysis.

3.1 Back-end: decoding, encoding, SSIM

Puffer receives six television channels using a VHF/UHF antenna and an ATSC demodulator, which outputs MPEG-2 transport streams in UDP. We wrote software to decode a stream to chunks of raw decoded video and audio, maintaining synchronization (by inserting black fields or silence) in the event of lost transport-stream packets on either substream. Video chunks are 2.002 seconds long, reflecting the 1/1001 factor for NTSC frame rates. Audio chunks are 4.8 seconds long. Video is de-interlaced with `ffmpeg` to produce a "canonical" 1080p60 or 720p60 source for compression.

Puffer encodes each video chunk in ten different H.264 versions, using `libx264` in `veryfast` mode. The encodings range from 240p60 video with constant rate factor (CRF) of 26 (about 200 kbps) to 1080p60 video with CRF of 20 (about 5,500 kbps). Audio chunks are encoded in the Opus format.

Puffer then uses `ffmpeg` to calculate each encoded chunk's SSIM [43], a measure of video quality, relative to the canonical source. This information is used by the objective function of BBA, MPC, RobustMPC, and Fugu, and for our evaluation. In practice, the relationship between bitrate and quality varies chunk-by-chunk (Figure 3), and users cannot perceive compressed chunk sizes directly—only what is shown on the screen. ABR schemes that maximize bitrate do not necessarily see a commensurate benefit in picture quality (Figure 4).

Encoding six channels in ten versions each (60 streams total) with `libx264` consumes about 48 cores of an Intel x86-64 2.7 GHz CPU in steady state. Calculating the SSIM of each encoded chunk consumes an additional 18 cores.

3.2 Serving chunks to the browser

To make it feasible to deploy and test arbitrary ABR schemes, Puffer uses a "dumb" player (using the HTML5 `<video>` tag and the JavaScript Media Source Extensions) on the client side, and places the ABR scheme at the server. We have a 48-core server with 10 Gbps Ethernet in a datacenter at Stanford. The browser opens a WebSocket (TLS/TCP) connection to a daemon on the server. Each daemon is configured with a different TCP congestion control (for the primary analysis, we used BBR [9]) and ABR scheme. Some schemes are more efficiently implemented than others; on average the CPU load from serving client traffic (including TLS, TCP, and ABR) is about 5% of an Intel x86-64 2.7 GHz core per stream.

Algorithm	Control	Predictor	Optimization goal	How trained
BBA	classical (linear control)	*n/a*	$+$SSIM *s.t.* bitrate $<$ limit	*n/a*
MPC-HM	classical (MPC)	classical (HM)	$+\overline{\text{SSIM}}$, $-$stalls, $-\Delta$SSIM	*n/a*
RobustMPC-HM	classical (robust MPC)	classical (HM)	$+\overline{\text{SSIM}}$, $-$stalls, $-\Delta$SSIM	*n/a*
Pensieve	learned (DNN)	*n/a*	$+\overline{\text{bitrate}}$, $-$stalls, $-\Delta$bitrate	reinforcement learning in simulation
Fugu	classical (MPC)	learned (DNN)	$+\overline{\text{SSIM}}$, $-$stalls, $-\Delta$SSIM	supervised learning *in situ*

Figure 5: Distinguishing features of algorithms used in the primary experiment. HM = harmonic mean of last five throughput samples. MPC = model predictive control. DNN = deep neural network.

Sessions are randomly assigned to serving daemons. Users can switch channels without breaking their TCP connection and may have many "streams" within each session.

Puffer is not a client-side DASH [28] (Dynamic Adaptive Streaming over HTTP) system. Like DASH, though, Puffer is an ABR system streaming chunked video over a TCP connection, and runs the same ABR algorithms that DASH systems can run. We don't expect this architecture to replace client-side ABR (which can be served by CDN edge nodes), but we expect its conclusions to translate to ABR schemes broadly. The Puffer website works in the Chrome, Firefox, Edge, and Opera browsers, including on Android phones, but does not play in the Safari browser or on iOS (which lack support for the Media Source Extensions or Opus audio).

3.3 Hosting arbitrary ABR schemes

We implemented buffer-based control (BBA), MPC, RobustMPC, and Fugu in back-end daemons that serve video chunks over the WebSocket. We use SSIM in the objective functions for each of these schemes. For BBA, we use the formula in the original paper [18] to decide the maximum chunk size, and subject to this constraint, the chunk with the highest SSIM is selected to stream. We also choose reservoir values consistent with our 15-second maximum buffer.

Deploying Pensieve for live streaming. We use the released Pensieve code (written in Python with TensorFlow) directly. When a client is assigned to Pensieve, Puffer spawns a Python subprocess running Pensieve's multi-video model.

We contacted the Pensieve authors to request advice on deploying the algorithm in a live, multi-video, real-world setting. The authors recommended that we use a longer-running training and that we tune the entropy parameter when training the multi-video neural network. We wrote an automated tool to train 6 different models with various entropy reduction schemes. We tested these manually over a few real networks, then selected the model with the best performance. We modified the Pensieve code (and confirmed with the authors) so that it does not expect the video to end before a user's session completes. We were not able to modify Pensieve to optimize SSIM; it considers the average bitrate of each Puffer stream. We adjusted the video chunk length to 2.002 seconds and the buffer threshold to 15 seconds to reflect our parameters. For training data, we used the authors' provided script to generate 1000 simulated videos as training videos, and a combination of the FCC and Norway traces linked to in the Pensieve codebase as training traces.

3.4 The Puffer experiment

To recruit participants, we purchased Google and Reddit ads for keywords such as "live tv" and "tv streaming" and paid people on Amazon Mechanical Turk to use Puffer. We were also featured in press articles. Popular programs (e.g. the 2019 and 2020 Super Bowls, the Oscars, World Cup, and "Bachelor in Paradise") brought large spikes ($> 20\times$) over baseline load. Our current average load is about 60 concurrent streams.

Between Jan. 26, 2019 and Feb. 2, 2020, we have streamed 38.6 years of video to 63,508 registered study participants using 111,231 unique IP addresses. About eight months of that period was spent on the "primary experiment," a randomized trial comparing Fugu with other algorithms: MPC, RobustMPC, Pensieve, and BBA (a summary of features is in Figure 5). This period saw a total of 314,577 streaming sessions, and 1,904,316 individual streams. An experimental-flow diagram in the standardized CONSORT format [35] is in the appendix (Figure A1).

We record client telemetry as time-series data, detailing the size and SSIM of every video chunk, the time to deliver each chunk to the client, the buffer size and rebuffering events at the client, the TCP statistics on the server, and the identity of the ABR and congestion-control schemes. A full description of the data is in Appendix B.

Metrics and statistical uncertainty. We group the time series by user stream to calculate a set of summary figures: the total time between the first and last recorded events of the stream, the startup time, the total watch time between the first and last successfully played portion of the stream, the total time the video is stalled for rebuffering, the average SSIM, and the chunk-by-chunk variation in SSIM. The ratio between "total time stalled" and "total watch time" is known as the rebuffering ratio or stall ratio, and is widely used to summarize the performance of streaming video systems [22].

We observe considerable heavy-tailed behavior in most of these statistics. Watch times are skewed (Figure 11), and while the *risk* of rebuffering is important to any ABR algorithm, actual rebuffering is a rare phenomenon. Of the 637,189 eligible streams considered for the primary analysis across all five ABR schemes, only 24,328 (4%) of those streams had *any* stalls, mirroring commercial services [22].

These skewed distributions create more room for the play of chance to corrupt the bottom-line statistics summarizing a scheme's performance—even two identical schemes will see considerable variation in average performance until a substantial amount of data is assembled. In this study, we worked to quantify the statistical uncertainty that can be attributed to the play of chance in assigning sessions to ABR algorithms. We calculate confidence intervals on rebuffering ratio with the bootstrap method [14], simulating streams drawn empirically from each scheme's observed distribution of rebuffering ratio as a function of stream duration. We calculate confidence intervals on average SSIM using the formula for weighted standard error, weighting each stream by its duration.

These practices result in substantial confidence intervals: with at least 2.5 years of data for each scheme, the width of the 95% confidence interval on a scheme's stall ratio is between $\pm 13\%$ and $\pm 21\%$ of the mean value. This is comparable to the magnitude of the total benefit reported by some academic work that used much shorter real-world experiments. Even a recent study of a Pensieve-like scheme on Facebook [24], encompassing 30 million streams, did not detect a change in rebuffering ratio outside the level of statistical noise.

We conclude that considerations of uncertainty in real-world learning and experimentation, especially given uncontrolled data from the Internet with real users, deserve further study. Strategies to import real-world data into repeatable emulators [45] or reduce their variance [26] will likely be helpful in producing robust learned networking algorithms.

4 Fugu: design and implementation

Fugu is a control algorithm for bitrate selection, designed to be feasibly trained in place (*in situ*) on a real deployment environment. It consists of a classical controller (model predictive control, the same as in MPC-HM), informed by a nonlinear predictor that can be trained with supervised learning.

Figure 6 shows Fugu's high-level design. Fugu runs on the server, making it easy to update its model and aggregate performance data across clients over time. Clients send necessary telemetry, such as buffer levels, to the server.

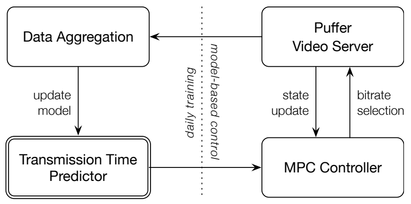


Figure 6: Overview of Fugu

The controller, described in Section 4.4, makes decisions by following a classical control algorithm to optimize an objective QoE function (§4.1) based on predictions for how long each chunk would take to transmit. These predictions are provided by the Transmission Time Predictor (TTP) (§4.2), a neural network that estimates a probability distribution for the transmission time of a proposed chunk with given size.

4.1 Objective function

For each video chunk K_i, Fugu has a selection of versions of this chunk to choose from, K_i^s, each with a different size s. As with prior approaches, Fugu quantifies the QoE of each chunk as a linear combination of video quality, video quality variation, and stall time [46]. Unlike some prior approaches, which use the average compressed bitrate of each encoding setting as a proxy for image quality, Fugu optimizes a perceptual measure of picture quality—in our case, SSIM. This has been shown to correlate with human opinions of QoE [12]. We emphasize that we use the exact same objective function in our version of MPC and RobustMPC as well.

Let $Q(K)$ be the video quality of a chunk K, $T(K)$ be the uncertain transmission time of K, and B_i be the current playback buffer size. Following [46], Fugu defines the QoE obtained by sending K_i^s (given the previously sent chunk K_{i-1}) as

$$\begin{aligned} QoE(K_i^s, K_{i-1}) = Q(K_i^s) - \lambda |Q(K_i^s) - Q(K_{i-1})| \\ - \mu \cdot \max\{T(K_i^s) - B_i, 0\}, \end{aligned} \tag{1}$$

where $\max\{T(K_i^s) - B_i, 0\}$ describes the stall time experienced by sending K_i^s, and λ and μ are configuration constants for how much to weight video quality variation and rebuffering. Fugu plans a trajectory of sizes s of the future H chunks to maximize their expected total QoE.

4.2 Transmission Time Predictor (TTP)

Once Fugu decides which chunk from K_i^s to send, two portions of the QoE become known: the video quality and video quality variation. The remaining uncertainty is the stall time. The server knows the current playback buffer size, so what it needs to know is the transmission time: how long will it take for the client to receive the chunk? Given an oracle that reports the transmission time of any chunk, the MPC controller can compute the optimal plan to maximize QoE.

Fugu uses a trained neural-network transmission-time predictor to approximate the oracle. For each chunk in the fixed H-step horizon, we train a separate predictor. E.g., if optimizing for the total QoE of the next five chunks, five neural networks are trained. This lets us parallelize training.

Each TTP network for the future step $h \in \{0, \dots, H-1\}$ takes as input a vector of:

1. sizes of past t chunks $K_{i-t}, \dots, K_{i-1}$,
2. actual transmission times of past t chunks: $T_{i-t}, \dots, T_{i-1}$,

3. internal TCP statistics (Linux `tcp_info` structure),
4. size s of a proposed chunk K^s_{i+h}.

The TCP statistics include the current congestion window size, the number of unacknowledged packets in flight, the smoothed RTT estimate, the minimum RTT, and the TCP estimated throughput (`tcpi_delivery_rate`).

Prior approaches have used Harmonic Mean (HM) [46] or a Hidden Markov Model (HMM) [40] to predict a single throughput for the entire lookahead horizon irrespective of the size of chunk to send. In contrast, the TTP acknowledges the fact that observed throughput varies with chunk size [7,32,47] by taking the size of proposed chunk K^s_{i+h} as an explicit input. In addition, it outputs a discretized probability distribution of predicted transmission time $\hat{T}(K^s_{i+h})$.

4.3 Training the TTP

We sample from the real usage data collected by *any* scheme running on Puffer and feed individual user streams to the TTP as training input. For the TTP network in the future step h, each user stream contains a chunk-by-chunk series of (a) the input 4-vector with the last element to be size of the actually sent chunk K_{i+h}, and, (b) the actual transmission time T_{i+h} of chunk K_{i+h} as desired output; the sequence is shuffled to remove correlation. It is worth noting that unlike prior work [25, 40] that learned from throughput traces, TTP is trained directly on real chunk-by-chunk data.

We train the TTP with standard supervised learning: the training minimizes the cross-entropy loss between the output probability distribution and the discretized actual transmission time using stochastic gradient descent.

We retrain the TTP every day, using training data collected over the prior 14 days, to avoid the effects of dataset shift and catastrophic forgetting [33,34]. Within the 14-day window, we weight more recent days more heavily. The weights from the previous day's model are loaded to warm-start the retraining.

4.4 Model-based controller

Our MPC controller (used for MPC-HM, RobustMPC-HM, and Fugu) is a stochastic optimal controller that maximizes the expected cumulative QoE in Equation 1 with replanning. It queries TTP for predictions of transmission time and outputs a plan $K^s_i, K^s_{i+1}, \ldots, K^s_{i+H-1}$ by value iteration [8]. After sending K^s_i, the controller observes and updates the input vector passed into TTP, and replans again for the next chunk.

Given the current playback buffer level B_i and the last sent chunk K_{i-1}, let $v^*_i(B_i, K_{i-1})$ denote the maximum expected sum of QoE that can be achieved in the H-step lookahead horizon. We have value iteration as follows:

$$v^*_i(B_i, K_{i-1}) = \max_{K^s_i} \Big\{ \sum_{t_i} \Pr[\hat{T}(K^s_i) = t_i] \cdot \\ (QoE(K^s_i, K_{i-1}) + v^*_{i+1}(B_{i+1}, K^s_i)) \Big\},$$

where $\Pr[\hat{T}(K^s_i) = t_i]$ is the probability predicted by TTP for the transmission time of K^s_i to be t_i, and B_{i+1} can be derived by system dynamics [46] given an enumerated (discretized) t_i. The controller computes the optimal trajectory by solving the above value iteration with dynamic programming (DP). To make the DP computational feasible, it also discretizes B_i into bins and uses forward recursion with memoization to only compute for relevant states.

4.5 Implementation

TTP takes as input the past $t = 8$ chunks, and outputs a probability distribution over 21 bins of transmission time: $[0, 0.25), [0.25, 0.75), [0.75, 1.25), \ldots, [9.75, \infty)$, with 0.5 seconds as the bin size except for the first and the last bins. TTP is a fully connected neural network, with two hidden layers with 64 neurons each. We tested different TTPs with various numbers of hidden layers and neurons, and found similar training losses across a range of conditions for each. We implemented TTP and the training in PyTorch, but we load the trained model in C++ when running on the production server for performance. A forward pass of TTP's neural network in C++ imposes minimal overhead per chunk (less than 0.3 ms on average on a recent x86-64 core). The MPC controller optimizes over $H = 5$ future steps (about 10 seconds). We set $\lambda = 1$ and $\mu = 100$ to balance the conflicting goals in QoE. Each retraining takes about 6 hours on a 48-core server.

4.6 Ablation study of TTP features

We performed an ablation study to assess the impact of the TTP's features (Figure 7). The prediction accuracy is measured using mean squared error (MSE) between the predicted transmission time and the actual (absolute, unbinned) value. For the TTP that outputs a probability distribution, we compute the expected transmission time by weighting the median value of each bin with the corresponding probability. Here are the more notable results:

Use of low-level congestion-control statistics. The TTP's nature as a DNN lets it consider a variety of noisy inputs, including low-level congestion-control statistics. We feed the kernel's `tcp_info` structure to the TTP, and find that several of these fields contribute positively to the TTP's accuracy, especially the RTT, CWND, and number of packets in flight (Figure 7). Although client-side ABR systems cannot typically access this structure directory because the statistics live on the sender, these results should motivate the communication of richer data to ABR algorithms wherever they live.

Transmission-time prediction. The TTP explicitly considers the size of a proposed chunk, rather than predicting throughput and then modeling transmission time as scaling linearly with chunk size [7, 32, 47]. We compared the TTP with an equivalent throughput predictor that is agnostic to the

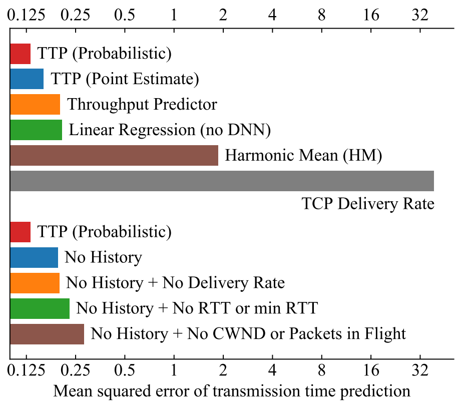


Figure 7: Ablation study of Fugu's Transmission Time Predictor (TTP). Removing any of the TTP's inputs reduced its ability to predict the transmission time of a video chunk. A non-probabilistic TTP ("Point Estimate") and one that predicts throughput without regard to chunk size ("Throughput Predictor") both performed markedly worse. TCP statistics (RTT, CWND, packets in flight) also proved helpful.

chunk's size (keeping everything else unchanged). The TTP's predictions were much more accurate (Figure 7).

Prediction with uncertainty. The TTP outputs a *probability distribution* of transmission times. This allows for better decision making compared with a single point estimate without uncertainty. We evaluated the expected accuracy of a probabilistic TTP vs. a point-estimate version that outputs the median value of the most-probable bin, and found an improvement in prediction accuracy with the former (Figure 7). To measure the end-to-end benefits of a probabilistic TTP, we deployed both versions on Puffer in August 2019 and collected 39 stream-days of data. It performed much worse than normal Fugu: the rebuffering ratio was 5× worse, without significant improvement in SSIM (data not shown).

Use of neural network. We found a significant benefit from using a deep neural network in this application, compared with a linear-regression model that was trained the same way. The latter model performed much worse on prediction accuracy (Figure 7). We also deployed it on Puffer and collected 448 stream-days of data in Aug.–Oct. 2019; its rebuffering ratio was 2.5× worse (data not shown).

Daily retraining. To evaluate our practice of retraining the TTP each day, we conducted a randomized comparison of several "out-of-date" versions of the TTP on Puffer between Aug. 7 and Aug. 30, 2019, and between Oct. 16, 2019 and Jan. 2, 2020. We compared vintages of the TTP that had been trained in February, March, April, and May 2019, alongside the TTP that is retrained each day. (We emphasize that the older TTP vintages were also learned *in situ* on two weeks of data from the actual deployment environment—they are simply earlier versions of the same predictor.) Somewhat to our

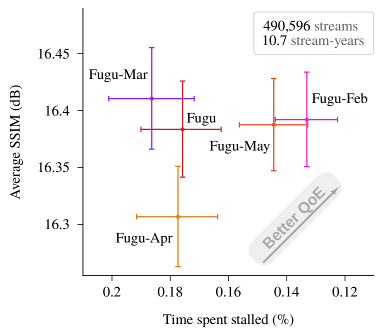


Figure 8: Fugu, which is retrained every day, did not outperform older versions of itself that were trained up to 11 months earlier. Our practice of daily retraining appears to be overkill.

surprise and disappointment, we were not able to document a benefit from daily retraining (Figure 8). This may reflect a lack of dynamism in the Puffer userbase, or the fact that once "enough" data is available to put the predictor through its paces, more-recent data is not necessarily beneficial, or some other reason. We suspect the older predictors might become stale at some point in the future, but for the moment, our practice of daily retraining appears to be overkill.

5 Experimental results

We now present findings from our experiments with the Puffer study, including the evaluation of Fugu. Our main results are shown in Figure 9. In summary, we conducted a parallel-group, blinded-assignment, randomized controlled trial of five ABR schemes between Jan. 26 and Aug. 7, and between Aug. 30 and Oct. 16, 2019. The data include 13.1 stream-years of data split across five algorithms, counting all streams that played at least 4 seconds of video. A standardized diagram of the experimental flow is available in the appendix (Figure A1).

We found that simple "buffer-based" control (BBA) performs surprisingly well, despite its status as a frequently outperformed research baseline. The only scheme to consistently outperform BBA in both stalls and quality was Fugu, but only when *all* features of the TTP were used. If we remove the probabilistic "fuzzy" nature of Fugu's predictions, *or* the "depth" of the neural network, *or* the prediction of transmission time as a function of chunk size (and not simply throughput), Fugu forfeits its advantage (§4.6). Fugu also outperformed other schemes in terms of SSIM variability (Figure 1). On a cold start to a new session, prior work [19, 40] suggested a need for session clustering to determine the quality of the first chunk. TTP provides an alternative approach: low-level TCP statistics are available as soon as the (HTTP/WebSocket, TLS, TCP) connection is established and allow Fugu to begin safely at a higher quality (Figure 10).

We conclude that robustly beating "simple" algorithms with machine learning may be surprisingly difficult, notwith-

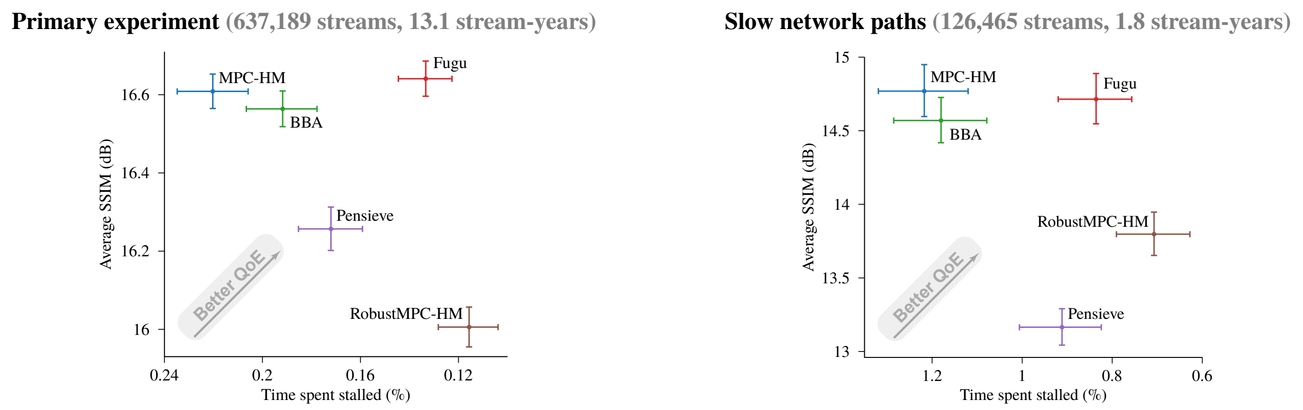


Figure 9: Main results. In a blinded randomized controlled trial that included 13.1 years of video streamed to 54,612 client IP addresses over an eight-month period, Fugu reduced the fraction of time spent stalled (except with respect to RobustMPC-HM), increased SSIM, and reduced SSIM variation within each stream (tabular data in Figure 1). "Slow" network paths have average throughput less than 6 Mbit/s; following prior work [25, 46], these paths are more likely to require nontrivial bitrate-adaptation logic. Such streams accounted for 14% of overall viewing time and 83% of stalls. Error bars show 95% confidence intervals.

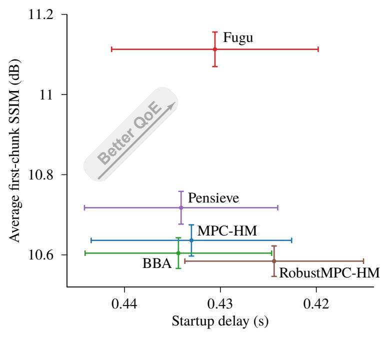


Figure 10: On a cold start, Fugu's ability to bootstrap ABR decisions from TCP statistics (e.g., RTT) boosts initial quality.

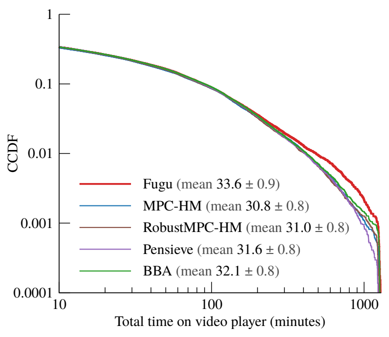


Figure 11: Users randomly assigned to Fugu chose to remain on the Puffer video player about 5%–9% longer, on average, than those assigned to other schemes. Users were blinded to the assignment. Legend shows 95% confidence intervals on the average time-on-site in minutes.

standing promising results in contained environments such as simulators and emulators. The gains that learned algorithms have in optimization or smarter decision making may come at a tradeoff in brittleness or sensitivity to heavy-tailed behavior.

5.1 Fugu users streamed for longer

We observed significant differences in the session durations of users across algorithms (Figure 11). Users whose sessions were assigned to Fugu chose to remain on the Puffer video player about 5–9% longer, on average, than those assigned to other schemes. Users were blinded to the assignment, and we believe the experiment was carefully executed not to "leak" details of the underlying scheme (MPC and Fugu even share most of their codebase). The average difference was driven solely by the upper 4% tail of viewership duration (sessions lasting more than 3 hours)—viewers assigned to Fugu are much more likely to keep streaming beyond this point, even as the distributions are nearly identical until then.

Time-on-site is a figure of merit in the video-streaming industry and might be increased by delivering better-quality video with fewer stalls, but we simply do not know enough about what is driving this phenomenon.

5.2 The benefits of learning *in situ*

Each of the ABR algorithms we deployed has been evaluated in emulation in prior work [25, 46]. Notably, the results in those works are qualitatively different from some of the real world results we have seen here—for example, buffer-based control matching or outperforming MPC-HM and Pensieve.

To investigate this further, we constructed an emulation environment similar to that used in [25]. This involved running the Puffer media server locally, and launching headless Chrome clients inside mahimahi [30] shells to connect to the server. Each mahimahi shell imposed a 40 ms end-to-end delay on traffic originating inside it and limited the downlink

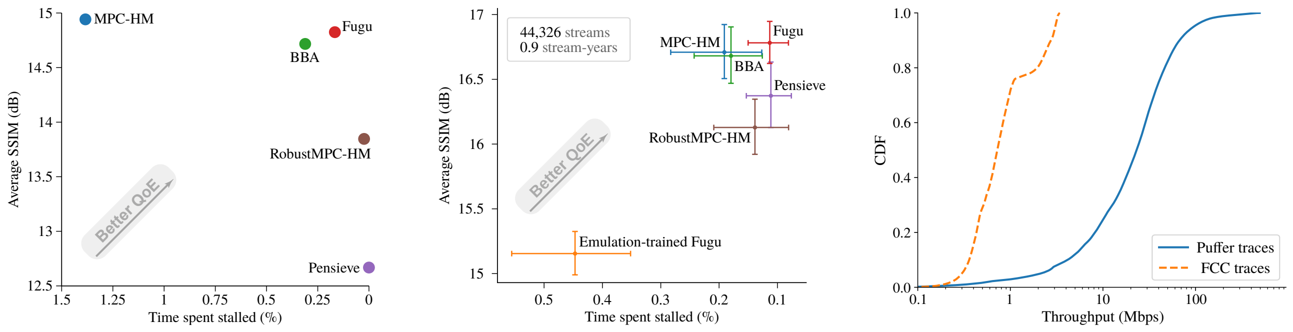


Figure 12: Left: performance in emulation, run in mahimahi [30] using the FCC traces [10], following the method of Pensieve [25]. **Middle:** During Jan. 26–Apr. 2, 2019, we randomized sessions to a set of algorithms including "emulation-trained Fugu." For Fugu, training in emulation did not generalize to the deployment environment. In addition, emulation results (left) are not indicative of real-world performance. **Right:** comparison of throughput distribution of FCC traces and of real Puffer sessions.

capacity over time to match the capacity recorded in a set of FCC broadband network traces [10]. As in the Pensieve evaluation, uplink speeds in all shells were capped at 12 Mbps. Within this test setup, we automated 12 clients to repeatedly connect to the media server, which would play a 10 minute clip recorded on NBC over each network trace in the dataset. Each client was assigned to a different ABR algorithm, and played the 10 minute video repeatedly over more than 15 hours of FCC traces. Results are shown in Figure 12.

We trained a version of Fugu in this emulation environment to evaluate its performance. Compared with the *in situ* Fugu—or with every other ABR scheme—the real-world performance of emulation-trained Fugu was horrible (Figure 12, middle panel). Looking at the other ABR schemes, almost each of them lies somewhere along the SSIM/stall frontier in emulation (left side of figure), with Pensieve rebuffering the least and MPC delivering the highest quality video. In the real experiment (middle of figure), we see a more muddled picture, with a different qualitative arrangement of schemes.

5.3 Remarks on Pensieve and RL for ABR

The original Pensieve paper [25] demonstrated that Pensieve outperformed MPC-HM, RobustMPC-HM, and BBA in both emulation-based tests and in video streaming tests on low and high-speed real-world networks. Our results differ; we believe the mismatch may have occurred for several reasons.

First, we have found that simulation-based training and testing do not capture the vagaries of the real-world paths seen in the Puffer study. Unlike real-world randomized trials, trace-based emulators and simulators allow experimenters to limit statistical uncertainty by running different algorithms on the same conditions, eliminating the effect of the play of chance in giving different algorithms a different distribution of watch times, network behaviors, etc. However, it is difficult to characterize the *systematic* uncertainty that comes from selecting a set of traces that may omit the variability or heavy-tailed nature of a real deployment experience (both network behaviors as well as user behaviors, such as watch duration).

Reinforcement learning (RL) schemes such as Pensieve may be at a particular disadvantage from this phenomenon. Unlike supervised learning schemes that can learn from training "data," RL typically requires a training *environment* to respond to a sequence of control decisions and decide on the appropriate consequences and reward. That environment could be real life instead of a simulator, but the level of statistical noise we observe would make this type of learning extremely slow or require an extremely broad deployment of algorithms in training. RL relies on being able to slightly vary a control action and detect a change in the resulting reward. By our calculations, the variability of inputs is such that it takes about 2 stream-years of data to reliably distinguish two ABR schemes whose innate "true" performance differs by 15%. To make RL practical, future work may need to explore techniques to reduce this variability [26] or construct more faithful simulators and emulators that model tail behaviors and capture additional dynamics of the real Internet that are not represented in throughput traces (e.g. varying RTT, cross traffic, interaction between throughput and chunk size [7]).

Second, most of the evaluation of Pensieve in the original paper focused on training and evaluating Pensieve using a single test video. As a result, the state space that model had to explore was inherently more limited. Evaluation of the Pensieve "multi-video model"—which we have to use for our experimental setting—was more limited. Our results are more consistent with a recent large-scale study of a Pensieve-multi-video-like scheme on 30 million streams at Facebook [24].

Third, the right side of Figure 12 shows that the distribution of throughputs in the FCC traces differs markedly from those on Puffer. This dataset shift could have harmed the performance of Pensieve, which was trained on the FCC traces. In response to reviewer feedback, we trained a version of Pensieve on throughput traces randomly sampled from real Puffer video sessions. This is essentially as close to a "learned *in situ*" version of Pensieve as we think we can achieve, but is

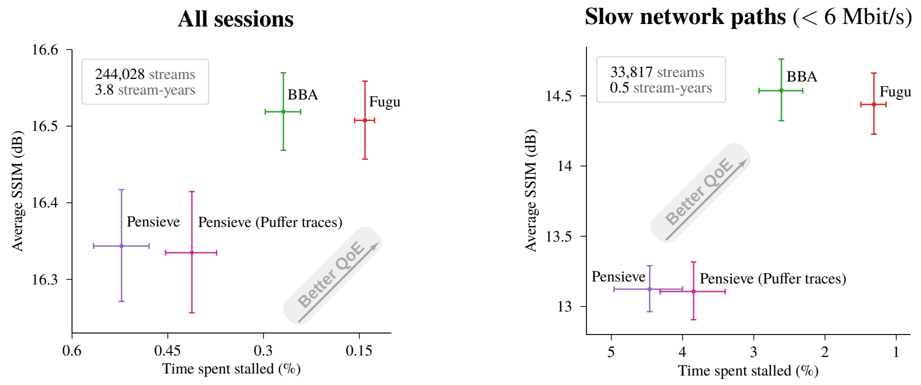


Figure 13: During Jan. 2–Feb. 2, 2020, we evaluated a version of Pensieve that was trained on a collection of network traces drawn randomly from actual Puffer sessions. This improved its performance compared with the original Pensieve, but the overall results were broadly similar.

not quite the same (§5.3). We compared "Pensieve on Puffer traces" with the original Pensieve, BBA, and Fugu between Jan. 2 and Feb. 2, 2020 (Figure 13). The results were broadly similar; the new Pensieve achieved better performance, but was still significantly worse than BBA and Fugu. The results deserve further study; they suggest that the representativeness of training data is not the end of the story when it comes to the real-world performance of RL schemes trained in simulation.

Finally, Pensieve optimizes a QoE metric centered around bitrate as a proxy for video quality. We did not alter this and leave the discussion to Section 6. Figure 4 shows that Pensieve was the #2 scheme in terms of bitrate (below BBA) in the primary analysis. We emphasize that our findings do not indicate that Pensieve cannot be a useful ABR algorithm, especially in a scenario where similar, pre-recorded video is played over a familiar set of known networks.

6 Limitations

The design of the Puffer experiment and the Fugu system are subject to important limitations that may affect their performance and generalizability.

6.1 Limitations of the experiments

Our randomized controlled trial represents a rigorous, but necessarily "black box," study of ABR algorithms for video streaming. We don't know the true distribution of network paths and throughput-generating processes; we don't know the participants or why the distribution in watch times differs by assigned algorithm; we don't know how to emulate these behaviors accurately in a controlled environment.

We have supplemented this black-box work with ablation analyses to relate the real-world performance of Fugu to the l^2 accuracy of its predictor, and have studied various ablated versions of Fugu in deployment. However, ultimately part of the reason for this paper is that we *cannot* replicate the experimental findings outside the real world—a real world whose behavior is noisy and takes lots of time to measure precisely. That may be an unsatisfying conclusion, and we doubt it will be the final word on this topic. Perhaps it will become possible to model enough of the vagaries of the real Internet "in silico" to enable the development of robust control strategies without extensive real-world experiments.

It is also unknown to what degree Puffer's results—which are about a single server in a university datacenter, sending to clients across our entire country over the wide-area Internet—generalize to a different server at a different institution, much less the more typical paths between a user on an access network and their nearest CDN edge node. We don't know for sure if the pre-trained Fugu model would work in a different location, or whether training a new Fugu based on data from that location would yield comparable results. Our results show that learning *in situ* works, but we don't know how specific the *situs* needs to be. And while we expect that Fugu could be implemented in the context of client-side ABR (especially if the server is willing to share its `tcp_info` statistics with the client), we haven't demonstrated this.

Although we believe that past research papers may have underestimated the uncertainties in real-world measurements with realistic Internet paths and users, we also may be guilty of underestimating our own uncertainties or emphasizing uncertainties that are only relevant to small or medium-sized academic studies, such as ours, and irrelevant to the industry. The current load on Puffer is about 60 concurrent streams on average, meaning we collect about 60 stream-days of data per day. Our primary analysis covers about 2.6 stream-years of data per scheme collected over an eight-month period, and was sufficient to measure its performance metrics to within about $\pm15\%$ (95% CI). By contrast, we understand YouTube has an average load of more than 60 *million* concurrent streams at any given time. We imagine the considerations of conducting data-driven experiments at this level may be completely different—perhaps less about statistical uncertainty, and more about systematic uncertainties and the difficulties of running experiments and accumulating so much data.

Some of Fugu's performance (and that of MPC, RobustMPC, and BBA) relative to Pensieve may be due to the fact that these four schemes received more information as they ran—namely, the SSIM of each possible version of each future chunk—than did Pensieve. It is possible that an "SSIM-aware" Pensieve might perform better. The load of calculating SSIM for each encoded chunk is not insignificant—about an extra 40% on top of encoding the video.

6.2 Limitations of Fugu

There is a sense that data-driven algorithms that more "heavily" squeeze out performance gains may also put themselves at risk to brittleness when a deployment environment drifts from one where the algorithm was trained. In that sense, it is hard to say whether Fugu's performance might decay catastrophically some day. We tried and failed to demonstrate a quantitative benefit from daily retraining over "out-of-date" vintages, but at the same time, we cannot be sure that some surprising detail tomorrow—e.g., a new user from an unfamiliar network—won't send Fugu into a tailspin before it can be retrained. A year of data on a growing userbase suggests, but doesn't guarantee, robustness to a changing environment.

Fugu does not consider several issues that other research has concerned itself with—e.g., being able to "replace" already-downloaded chunks in the buffer with higher quality versions [38], or optimizing the joint QoE of multiple clients who share a congestion bottleneck [29].

Fugu is not tied as tightly to the TCP or congestion control as it might be—for example, Fugu could wait to send a chunk until the TCP sender tells it that there is a sufficient congestion window for most of the chunk (or the whole chunk) to be sent immediately. Otherwise, it *might* choose to wait and make a better-informed decision later. Fugu does not schedule the transmission of chunks—it will always send the next chunk as long as the client has room in its playback buffer.

7 Conclusion

Machine-learned systems in computer networking sometimes describe themselves as achieving near-"optimal" performance, based on results in a contained or modeled version of the problem [25, 37, 39]. Such approaches are not limited to the academic community: in early 2020, a major video-streaming company announced a $5,000 prize for the best low-delay ABR scheme, in which candidates will be evaluated in a network simulator that follows a trace of varying throughput [2].

In this paper, we suggest that these efforts can benefit from considering a broader notion of performance and optimality. Good, or even near-optimal, performance in a simulator or emulator does not necessarily predict good performance over the wild Internet, with its variability and heavy-tailed distributions. It remains a challenging problem to gather the appropriate training data (or in the case of RL systems, training environments) to properly learn and validate such systems.

In this paper, we asked: *what does it take to create a learned ABR algorithm that robustly performs well over the wild Internet?* In effect, our best answer is to cheat: train the algorithm *in situ* on data from the real deployment environment, and use an algorithm whose structure is sophisticated enough (a neural network) and yet also simple enough (a predictor amenable to supervised learning on data, informing a classical controller) to benefit from that kind of training.

Over the last year, we have streamed 38.6 years of video to 63,508 users across the Internet. Sessions are randomized in blinded fashion among algorithms, and client telemetry is recorded for analysis. The Fugu algorithm robustly outperformed other schemes, both simple and sophisticated, on objective measures (SSIM, stall time, SSIM variability) and increased the duration that users chose to continue streaming.

We have found the Puffer approach a powerful tool for networking research—it is fulfilling to be able to "measure, then build" [5] to iterate rapidly on new ideas and gain feedback. Accordingly, we are opening Puffer as an "open research" platform. Along with this paper, we are publishing our full archive of data and results on the Puffer website. The system posts new data each week, along with a summary of results from the ongoing experiments, with confidence intervals similar to those in this paper. (The format is described in Appendix B.) We redacted some fields from the public archive to protect participants' privacy (e.g., IP address) but are willing to work with researchers on access to these fields in an aggregated fashion. Puffer and Fugu are also open-source software, as are the analysis tools used to prepare the results in this paper.

We plan to operate Puffer as long as feasible and invite researchers to train and validate new algorithms for ABR control, network and throughput prediction, and congestion control on its traffic. We are eager to collaborate with and learn from the community's ideas on how to design and deploy robust learned systems for the Internet.

Acknowledgments

We are greatly indebted to Emily Marx, who joined this project after the original submission of this paper, found and corrected bugs in our analysis tools, and performed the final data analysis. We thank our shepherd, Vyas Sekar, and the ACM SIGCOMM and USENIX NSDI reviewers for their helpful feedback. We are grateful for conversations with and feedback from Danfei Xu, T.Y. Huang, Hongzi Mao, Michael Schapira, and Nils Krahnstoever, and we thank the participants in the Puffer research study, without whom these experiments could not have been conducted. This work was supported by NSF grant CNS-1909212 and by Google, Huawei, VMware, Dropbox, Facebook, and the Stanford Platform Lab.

References

[1] Locast: Non-profit retransmission of broadcast television, June 2018. https://news.locast.org/app/uploads/2018/11/Locast-White-Paper.pdf.

[2] MMSys'20/Twitch Grand Challenge on Adaptation Algorithms for Near-Second Latency, January 2020. https://2020.acmmmsys.org/lll_challenge.php.

[3] Alekh Agarwal, Nan Jiang, and Sham M. Kakade. Lecture notes on the theory of reinforcement learning. 2019.

[4] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: Auto-tuning video ABR algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM SIGCOMM*, pages 44–58, 2018.

[5] Remzi Arpaci-Dusseau. Measure, then build (USENIX ATC 2019 keynote). Renton, WA, July 2019. USENIX Association.

[6] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for Internet video. *ACM SIGCOMM Computer Communication Review*, 43(4):339–350, 2013.

[7] Mihovil Bartulovic, Junchen Jiang, Sivaraman Balakrishnan, Vyas Sekar, and Bruno Sinopoli. Biases in data-driven networking, and what to do about them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 192–198, 2017.

[8] Richard Bellman. A Markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[9] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14(5):20–53, 2016.

[10] Federal Communications Commission. Measuring Broadband America. https://www.fcc.gov/general/measuring-broadband-america.

[11] Paul Crews and Hudson Ayers. CS 244 '18: Recreating and extending Pensieve, 2018. https://reproducingnetworkresearch.wordpress.com/2018/07/16/cs-244-18-recreating-and-extending-pensieve/.

[12] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154–166, 2016.

[13] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. In *ICML 2019 Workshop RL4RealLife*, 2019.

[14] Bradley Efron and Robert Tibshirani. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical science*, pages 54–75, 1986.

[15] Sally Floyd and Eddie Kohler. Internet research needs better models. *ACM SIGCOMM Computer Communication Review*, 33(1):29–34, 2003.

[16] Sally Floyd and Vern Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, 2001.

[17] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, 2018.

[18] 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 2014 Conference of the ACM SIGCOMM*, pages 187–198, 2014.

[19] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. CFA: A practical prediction system for video QoE optimization. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 137–150, 2016.

[20] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE. In *Proceedings of the 8th International Conference on emerging Networking EXperiments and Technologies*, pages 97–108, 2012.

[21] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking*, 21(6):2001–2014, 2013.

[22] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the 2017 Conference of the ACM SIGCOMM*, pages 183–196, 2017.

[23] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C. Begen, and David Oran. Probe and adapt: Rate adaptation for HTTP video streaming at scale.

IEEE Journal on Selected Areas in Communications, 32(4):719–733, 2014.

[24] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. Real-world video adaptation with reinforcement learning. In *ICML 2019 Workshop RL4RealLife*, 2019.

[25] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *Proceedings of the 2017 Conference of the ACM SIGCOMM*, pages 197–210. ACM, 2017.

[26] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. In *International Conference on Learning Representations*, 2019.

[27] Ricky K.P. Mok, Xiapu Luo, Edmond W.W. Chan, and Rocky K.C. Chang. QDASH: a QoE-aware DASH system. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 11–22, 2012.

[28] *Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats*, April 2012. ISO/IEC 23009-1 (http://standards.iso.org/ittf/PubliclyAvailableStandards).

[29] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 408–423, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[31] Vern Paxson and Sally Floyd. Why we don't know how to simulate the Internet. In *Proceedings of the 29th conference on Winter simulation*, pages 1037–1044, 1997.

[32] Yanyuan Qin, Shuai Hao, Krishna R. Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. ABR streaming of VBR-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 366–378. ACM, 2018.

[33] Anthony Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

[34] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 627–635, 2011.

[35] Kenneth F. Schulz, Douglas G. Altman, and David Moher. CONSORT 2010 statement: updated guidelines for reporting parallel group randomised trials. *BMC medicine*, 8(1):18, 2010.

[36] Alexander T. Schwarm and Michael Nikolaou. Chance-constrained model predictive control. *AIChE Journal*, 45(8):1743–1752, 1999.

[37] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. In *Proceedings of the 2014 Conference of the ACM SIGCOMM*, pages 479–490, 2014.

[38] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: Improving bitrate adaptation in the DASH reference player. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, pages 123–137, New York, NY, USA, 2018. ACM.

[39] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. BOLA: Near-optimal bitrate adaptation for online videos. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.

[40] 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 2016 Conference of the ACM SIGCOMM*, pages 272–285, 2016.

[41] Cisco Systems. Cisco Visual Networking Index: Forecast and trends, 2017–2022, November 2018. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf.

[42] Guibin Tian and Yong Liu. Towards agile and smooth video adaptation in dynamic HTTP streaming. In *Proceedings of the 8th International Conference on emerging Networking EXperiments and Technologies*, pages 109–120, 2012.

[43] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

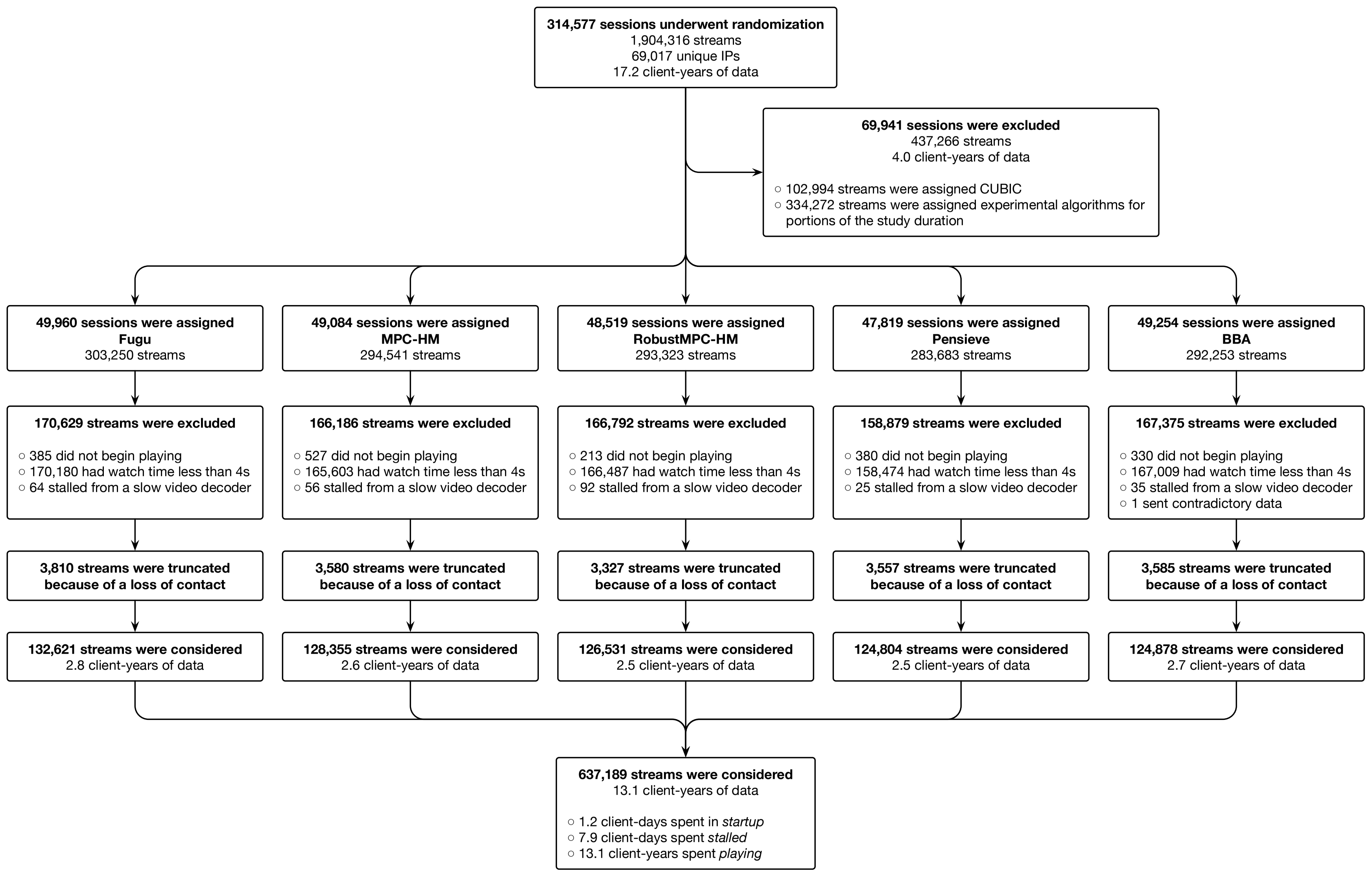
[44] Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-generated congestion control. *Proceedings of the 2013 Conference of the ACM SIGCOMM*, 43(4):123–134, 2013.

[45] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for Internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, Boston, MA, 2018. USENIX Association.

[46] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proceedings of the 2015 Conference of the ACM SIGCOMM*, pages 325–338, 2015.

[47] Tong Zhang, Fengyuan Ren, Wenxue Cheng, Xiaohui Luo, Ran Shu, and Xiaolan Liu. Modeling and analyzing the influence of chunk size variation on bitrate adaptation in DASH. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

A Randomized trial flow diagram



314,577 sessions underwent randomization
1,904,316 streams
69,017 unique IPs
17.2 client-years of data
69,941 sessions were excluded
437,266 streams
4.0 client-years of data
○ 102,994 streams were assigned CUBIC
○ 334,272 streams were assigned experimental algorithms for portions of the study duration
49,960 sessions were assigned Fugu
303,250 streams
49,084 sessions were assigned MPC-HM
294,541 streams
48,519 sessions were assigned RobustMPC-HM
293,323 streams
47,819 sessions were assigned Pensieve
283,683 streams
49,254 sessions were assigned BBA
292,253 streams
170,629 streams were excluded
○ 385 did not begin playing
○ 170,180 had watch time less than 4s
○ 64 stalled from a slow video decoder
166,186 streams were excluded
○ 527 did not begin playing
○ 165,603 had watch time less than 4s
○ 56 stalled from a slow video decoder
166,792 streams were excluded
○ 213 did not begin playing
○ 166,487 had watch time less than 4s
○ 92 stalled from a slow video decoder
158,879 streams were excluded
○ 380 did not begin playing
○ 158,474 had watch time less than 4s
○ 25 stalled from a slow video decoder
167,375 streams were excluded
○ 330 did not begin playing
○ 167,009 had watch time less than 4s
○ 35 stalled from a slow video decoder
○ 1 sent contradictory data
3,810 streams were truncated because of a loss of contact
3,580 streams were truncated because of a loss of contact
3,327 streams were truncated because of a loss of contact
3,557 streams were truncated because of a loss of contact
3,585 streams were truncated because of a loss of contact
132,621 streams were considered
2.8 client-years of data
128,355 streams were considered
2.6 client-years of data
126,531 streams were considered
2.5 client-years of data
124,804 streams were considered
2.5 client-years of data
124,878 streams were considered
2.7 client-years of data
637,189 streams were considered
13.1 client-years of data
○ 1.2 client-days spent in *startup*
○ 7.9 client-days spent *stalled*
○ 13.1 client-years spent *playing*

Figure A1: CONSORT-style diagram [35] of experimental flow for the primary results (Figures 1 and 9), obtained during the period Jan. 26–Aug. 7, 2019, and Aug. 30–Oct. 16, 2019. A "session" represents one visit to the Puffer video player and may contain many "streams." Reloading starts a new session, but changing channels only starts a new stream and does not change TCP connections or ABR algorithms.

B Description of open data

The open data we are releasing comprise different "measurements"—each measurement contains a different set of time-series data collected on Puffer servers. Below we highlight the format of interesting fields in three measurements that are essential for analysis: `video_sent`, `video_acked`, and `client_buffer`.

`video_sent` collects a data point every time a Puffer server sends a video chunk to a client. Each data point contains:

- `time`: timestamp when the chunk is sent
- `session_id`: unique ID for the video session
- `expt_id`: unique ID to identify the experimental group; `expt_id` can be used as a key to retrieve the experimental setting (e.g., ABR, congestion control) when sending the chunk, in another file we are providing.
- `channel`: TV channel name
- `video_ts`: unique presentation timestamp of the chunk
- `format`: encoding settings of the chunk, including resolution and constant rate factor (CRF)
- `size`: size of the chunk
- `ssim_index`: SSIM of the chunk
- `cwnd`: congestion window size (`tcpi_snd_cwnd`)
- `in_flight`: number of unacknowledged packets in flight (`tcpi_unacked - tcpi_sacked - tcpi_lost + tcpi_retrans`)
- `min_rtt`: minimum RTT (`tcpi_min_rtt`)
- `rtt`: smoothed RTT estimate (`tcpi_rtt`)
- `delivery_rate`: estimate of TCP throughput (`tcpi_delivery_rate`)

`video_acked` collects a data point every time a Puffer server receives a video chunk acknowledgement from a client. Each data point can be matched to a data point in `video_sent` using `video_ts` (if the chunk is ever acknowledged) and used to calculate the transmission time of the chunk—difference between the timestamps in the two data points. Specifically, each data point in `video_acked` contains:

- `time`: timestamp when the chunk is acknowledged
- `session_id`
- `expt_id`
- `channel`
- `video_ts`

`client_buffer` collects client-side information reported to Puffer servers on a regular interval and when certain events occur. Each data point contains:

- `time`: timestamp when the client message is received
- `session_id`
- `expt_id`
- `channel`
- `event`: event type, e.g., was this triggered by a regular report every quarter second, or because the client stalled or began playing.
- `buffer`: playback buffer size
- `cum_rebuf`: cumulative rebuffer time in the current stream

Between Jan. 26, 2019 and Feb. 2, 2020, we collected 675,839,652 data points in `video_sent`, 677,956,279 data points in `video_acked`, and 4,622,575,336 data points in `client_buffer`.