Sammy: smoothing video traffic to be a friendly internet neighbor

Bruce Spang
Stanford University

Shravya Kunamalla
Netflix

Renata Teixeira
Netflix

Te-Yuan Huang
Netflix

Grenville Armitage
Netflix

Ramesh Johari
Stanford University

Nick McKeown
Stanford University

ABSTRACT

On-demand streaming video traffic is managed by an adaptive bitrate (ABR) algorithm whose job is to optimize quality of experience (QoE) for a single video session. ABR algorithms leave the question of sharing network resources up to transport-layer algorithms. We observe that as the internet gets faster relative to video streaming rates, this delegation of responsibility gives video traffic a burstier on-off traffic pattern. In this paper, we show we can substantially smooth video traffic to improve its interactions with the rest of the internet, while maintaining the same or better QoE for streaming video. We smooth video traffic with two design principles: application-informed pacing, which allows ABR algorithms to set an upper limit on packet-by-packet throughput, and by designing ABR algorithms that work with pacing. We propose a joint ABR and rate-control scheme, called Sammy, which selects both video quality and pacing rates. We implement our scheme and evaluate it at a large video streaming service. Our approach smooths video, making it a more friendly neighbor to other internet applications. One surprising result is that being friendlier requires no compromise for the video traffic: in large scale, production experiments, Sammy improves video QoE over an existing, extensively tested and tuned production ABR algorithm.

CCS CONCEPTS

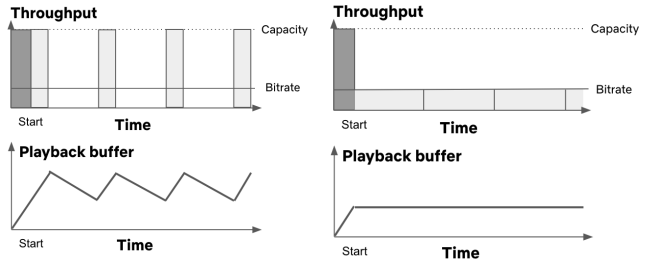
• **Networks** → **Cross-layer protocols**; **Network resources allocation**; • **Information systems** → **Multimedia streaming**;

KEYWORDS

Video streaming; Adaptive bitrate algorithms; Congestion control algorithms; Network friendliness

1 INTRODUCTION

On-demand streaming video traffic from services like Netflix and YouTube currently comprises 60-75% of internet traffic [57]. This fraction is likely even higher during peak viewing hours. With a single application having such a large volume of traffic, as a community we should make it as good a neighbor as possible. If we do so, we will improve the internet for all applications that share it.



(a) Video traffic today. **(b) Smoother, same QoE.**

Figure 1: A few seconds of a typical streaming video session. Today, video has on periods (light grey) where it sends data as fast as the network supports (a). But as shown in (b), we can smooth throughput and reduce congestion without impacting video QoE.

Streaming video traffic today has an on-off, bursty traffic pattern shown in Figure 1a: every few seconds, video switches between an on period of sending data as fast as possible and an off period of silence. This is a well known phenomenon [54], and arises from the standard architecture used to stream video. Videos are split into "chunks" of a few seconds each and stored on a server. Server-side congestion control algorithms send chunks to the video player as fast as the network allows, creating the on periods shown in Figure 1a. The video player then puts the chunks into a playback buffer, to be played back as needed. Each chunk is encoded at a number of different bitrates: from a higher quality, larger chunk-size, to a lower quality, smaller chunk-size. An adaptive bitrate (ABR) algorithm selects the bitrate of each chunk. When network bandwidth is higher than the bitrate of the chunk, chunks arrive at the client faster than they are played back and so the playback buffer grows. Client buffers can only store a limited number of chunks, so ABR algorithms periodically pause to make room for new chunks creating the off periods in Figure 1a. For more background, see Section 2.

Over the past decade, video traffic has been getting more bursty: on periods are getting shorter, and off periods are getting longer. A decade ago, home access speeds and video bitrates were both on the order of a few megabits per second [15, 38], and so video traffic

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604839

spent most of its time in on periods [54]. Since then, median last-mile network data-rates to the home have improved, worldwide, to tens or hundreds of megabits per second [16, 63]. Video encoding has also improved: today a 1080p video requires only a few megabits per second [38] and a 4k video requires typically less than twenty megabits per second [45]. Videos download faster, playback buffers fill up faster, and video traffic has shorter, faster on periods.[1]

Video traffic can be smoother. Figure 1b shows the exact same video streaming trace, smoothed out. We have reduced the *chunk throughput* (the throughput during on periods) to the level of the video bitrate—well below the network capacity. From the end user's perspective, the quality of experience (QoE) of the video is identical. Video QoE is measured by *video quality* (how good the video looks), *play delay* (how long the video takes to start playing), and *rebuffers* (times when the video playback is interrupted because data is not available). Both sessions in Figure 1 have the same QoE: quality is the same (the same number of bytes are downloaded in the same amount of time), the play delay is the same (the width of the dark gray box is the same), and there are no rebuffers (the buffer never goes to zero).

Smoothing video traffic as in Figure 1b has benefits to neighboring traffic sharing the same network. There are well-studied consequences to congestion control sending as fast as possible during on periods, including increased packet loss and queueing delay, bufferbloat [24], and unfairness between flows [1, 7, 8, 11, 12, 18, 32, 33, 35, 36, 42, 60, 68–71]. Conventional wisdom in queueing theory also suggests that burstiness increases router queues and network congestion [27], and so is detrimental to neighboring traffic.

By reducing chunk throughput *below* the capacity in Figure 1a, we avoid these issues completely. There will be *no* queueing delay or packet loss. A short HTTP request issued during an on period could complete faster with more available bandwidth and lower queueing delay. A longer-lived video conferencing flow would see more consistent throughput and delay. Reducing burstiness should benefit everyone.

The major challenge of making video traffic smoother at scale is doing so without making video traffic perform worse. Video QoE is important for the experience of the people watching the video [17, 40, 75] and we would not want to smooth video traffic—the majority use of the internet—by making its users suffer. There are two different aspects of this challenge.

First, there is a fundamental limit to how much traffic can be smoothed without impacting QoE. For example, we could smooth Figure 1b even more by reducing the throughput before playback starts to match the rest of the session. But this would increase play delay. No buffer will be built up, so the playback will rebuffer if throughput varies.

Second, ABR algorithms have historically had a core assumption that measurements of chunk throughput give them accurate estimates of the available bandwidth of the network [4, 31, 35, 59, 64–66, 72, 73]. Reducing chunk throughput breaks this assumption, and could cause an ABR algorithm to select lower qualities—artificially lowering QoE.

[1]Anecdotally, the median Netflix session today has an average throughput 13x higher than its average bitrate.

In this paper, we present a novel solution to this challenge and make video traffic smoother. Remarkably, our approach appears to benefit everyone: we are able to substantially smooth video traffic while slightly *improving* QoE relative to existing, finely-tuned production video streaming systems.

To smooth video traffic, we allow ABR algorithms to directly limit the packet-by-packet sending rate using a new technique called *application-informed pacing*. An ABR algorithm might ask for a chunk of video to be delivered at no more than one packet per millisecond, and a congestion control algorithm sends packets no faster than once every millisecond using TCP Pacing [1, 13, 26, 28, 47, 56, 71]. This allows the ABR algorithm to smooth out throughput across the full range of timescales: from the level of a few packets, to an entire chunk, to an entire video session. Application-informed pacing is described in more detail in Section 3.2.

We next propose *Sammy*,[2] an algorithm that selects both bitrates and pacing rates to achieve high video QoE and improve smoothness. Sammy is described in Section 4. Our key insight, described in Section 3.1, is that while ABR algorithms have historically *used* measurements of available bandwidth to make their decisions, they do not *need* accurate estimates to achieve good QoE.

We implement Sammy at Netflix and evaluate it with large scale, production experiments in Section 5. Sammy substantially smooths video traffic: reducing chunk throughput to roughly three times higher than the highest bitrate of a video. In production experiments, this lowers chunk throughput by 61% at the median. This improves congestion metrics: improving retransmissions by 35%, and RTTs by 14%. Surprisingly, Sammy actually *slightly improves* video QoE relative to production values despite this reduction in throughput: improving initial video quality by 0.2%, overall quality by 0.03%, and play delay by 1.3% while maintaining rebuffers.

We present illustrative lab experiments in Section 6 in which Sammy improves the performance of neighboring traffic by increasing its throughput and reducing queueing delay. Sammy improves delay for a neighboring UDP flow by 51%, improves throughput for a TCP flow by 28%, improves response times for HTTP traffic by 18%, and improves play delay for another video session by 4%.

Streaming video services are incentivized to deploy Sammy. First, neighboring traffic could easily be from the same video service. By improving performance for its neighbors, Sammy improves performance for the video service itself. Second, Sammy forces an ABR algorithm to be careful about its use of throughput estimates. This exercise gave us a slight QoE improvement, and could yield larger improvements for other ABR algorithms.

This work is a first step towards smoothing video traffic. We conclude in Section 7 by highlighting that there is still more work to be done. As a community, we have an opportunity to further smooth traffic, video and beyond. After all, a smoother internet benefits everyone.

To demonstrate the deployability of Sammy, we have released an open source prototype [61] which uses off the shelf components including an unmodified dash.js player and the Fastly CDN.

Ethical considerations: Our experiments involve live traffic running on a large video streaming service. Sammy makes video traffic friendlier to its neighbors while improving QoE, so we believe

[2]As of this writing, Sammy is the current reigning world's fastest snail [53].

our experiments are beneficial. Netflix regularly runs rigorous A/B testing for every change it makes to its service. Its customers have the ability to opt out of experiments, if they choose to.

2 BACKGROUND AND RELATED WORK

The burstiness of video traffic arises from the standard architecture of modern video streaming services. One of the core design principles of the internet is layering [14], conceptually separating applications from underlying transport protocols like TCP or QUIC. The internet community has kept a minimalist inter-layer interface, giving applications little ability to express their service goals. As a result, transport protocols optimize for transport-level goals: maximizing throughput, avoiding congestion, and splitting throughput fairly. Adaptive bitrate (ABR) algorithms select bitrates to ensure that viewers get the best quality of experience (QoE) possible, given whatever throughput is picked by the congestion control algorithm.

In this section, we will give an overview of existing work and describe why this architecture makes it difficult to achieve our goal of smoothing video traffic while achieving high QoE. In this section we focus on the most relevant papers. For a more complete overview, there are a number of wider surveys [41, 52, 58].

2.1 ABR algorithms

When a network has limited bandwidth, there is a tradeoff between the three major video QoE metrics: quality, startplay delay, and rebuffers. A video playback can have both high quality and no rebuffers if it incurs a high play delay by downloading the entire video before it starts. It can start quickly in a slow network by either picking a low quality or rebuffering after playback starts. The role of an ABR algorithm is to manage this tradeoff between the different QoE metrics, and ensure that the user gets the best possible QoE. Note that this does not necessarily mean picking the highest video quality—in a given network, a very low quality might significantly reduce rebuffers and have the best QoE.

Videos are split into chunks of a few seconds each. Each chunk is encoded in a ladder of different bitrates: from a small, low-quality version to a larger, high-quality version. A video provider will allow a particular device in a particular network to use some subset of this ladder based on the user's plan, device limitations, and other business policies. The ABR algorithm chooses a rung from this ladder for each chunk. The transport layer then splits that chunk into packets and sends each packet to the video client. When chunks are downloaded, they are added to a playback buffer in the video client. Even if the network is unavailable, the client can continue playing as long as there are chunks in the buffer.

When it takes too long to download a chunk, the buffer can shrink and it may be impossible to maintain high video quality without rebuffers. To deal with this, ABR algorithms can pick lower bitrates to grow the buffer and avoid rebuffers. There are two main types of ABR algorithms in use today:

Throughput-based: An ABR algorithm that takes explicit throughput measurements from the network, and uses them to select bitrates [4, 35, 49, 59, 64, 66, 72, 73]. Typical algorithms produce some estimate based on chunk throughput, and then use it to optimize the various QoE metrics. ABR algorithms can also use throughput in other ways, for example, Oboe [4] switches between several parameter settings based on throughput measurements. VOXEL makes modifications to the transport layer to drop video frames in challenging network conditions. It is generally understood [72, 73] that throughput-based algorithms will perform better the more accurately they are able to predict throughput of upcoming chunks.

Buffer-based: An ABR algorithm may select a bitrate based only on the buffer level [31, 65]. When the buffer is low, the algorithm will pick the lowest bitrate. When the buffer is high, it will pick the highest bitrate. Over time, these algorithms converge to an average bitrate close to the average chunk throughput. In effect, the buffer size encodes the past available bandwidth measurements from the network. In practice, buffer-based algorithms can also include a throughput-based component during startup [64].

Existing ABR algorithms rely on the available bandwidth measurements produced by congestion control algorithms. By decreasing chunk throughput, we could make existing ABR algorithms perform worse. We discuss how we address this in Section 3.2.

Until now, ABR algorithms have focused on maximizing the quality of experience (QoE) for a video streaming client given whatever throughput is chosen by congestion control algorithms. ABR algorithms do not make choices about throughput. In contrast, our goal is to design an algorithm that smooths video traffic and achieves high QoE while improving the internet for neighboring traffic.

2.2 Congestion control

Once an ABR algorithm has selected a chunk, it is the job of congestion control algorithms to decide how fast to send the packets of that chunk into the network. Congestion control algorithms balance competing goals: achieving high throughput, avoiding congestion, and fairly splitting network resources among users [50, Sec. 3.2].

There is a long line of research on congestion control, and we refer the reader to existing surveys for details [52]. It is challenging (if not impossible [74, 76]) to simultaneously achieve all the goals of congestion control, and there are examples of congestion control algorithms struggling with packet loss and queueing delay, bufferbloat [24], and unfairness [1, 7, 8, 11, 12, 18, 32, 33, 35, 36, 42, 60, 68–71]. By focusing on the needs of video QoE, Sammy gives up the goal of achieving high chunk throughput and so is able to improve congestion and leave more bandwidth available for other users of the network.

Our work uses the classic idea of TCP Pacing: a mechanism for adding delay between successive packet sends to reduce the size of bursts, and reduce packet drops and queueing delay [1, 13, 26, 28, 47, 56, 71]. Pacing gives packets a constant interarrival time, which theoretically minimizes queueing delay in general settings [27]. In practice, pacing reduces congestion [1, 47, 71]. Typically the time between successive packets is set to a value larger than the cwnd/RTT [1, 67], which reduces burstiness at the packet-level, but does not reduce chunk throughput. BBR [13] is a congestion control algorithm that directly adjusts the pace rate, but it aims to pace close to the bottleneck capacity while Sammy aims to pace significantly lower.

There has been prior work on congestion control to improve fairness among competing video clients. There are "scavenger" congestion control algorithms like LEDBAT [55] and PCC Proteus [46], that give up throughput when competing with a non-scavenger

Paper	Main Goal	Smoothing Mechanism	Eval.	Preserves QoE?	Smooths below avail. bandwidth?
Our work	Smoothness	ABR-informed Pacing	Prod.	✓	✓
TCP Pacing [1]	Packet bursts	Pacing	Prod.	✓	✗
Trickle baseline [25]	Wasted buffer	Token Bucket	?	?	✓
Trickle [25]	Packet bursts	CWND limit	Prod.	?	✓
[59]	Wasted buffer	Delays TTFB	Lab	✗	✗
SABRE #1 [44]	Bufferbloat	RWND limit	Lab	✗	✓
SABRE #2 [44]	Bufferbloat	RWND limit	Lab	✗	✓
[3]	Video fairness	Token Bucket	Lab	✗	✓
[9]	Video fairness	Token Bucket	Lab	✗	✗

Table 1: Related work on smoothing video traffic either does not preserve QoE or does not reduce throughput below the available network bandwidth.

congestion control algorithm. These were originally designed to reduce the impact of BitTorrent traffic on other internet traffic [55], but the PCC Proteus [46] authors describe a hybrid mode, in which video traffic switches from non-scavenger to scavenger mode when the throughput exceeds a threshold. The authors show that this improves performance when multiple video clients compete. Minerva [48] improves fairness between competing video sessions by sharing proportionally based on a measure of perceptual quality. These approaches will fully utilize the network when no neighboring traffic is present. In contrast, Sammy does not focus on fairness and instead consistently sends at a rate closer to the video bitrate. Surprisingly, we show that consistently smoothing (even when neighboring traffic is not present) achieves similar performance to fully utilizing the network.

2.3 Reducing Burstiness for Video Traffic

There has been prior work on reducing the burstiness of video traffic. The novelty of our work is that by designing a joint ABR and rate limiting algorithm, we are able to reduce chunk throughput below the available bandwidth of a network *while achieving comparable QoE to today's top ABR algorithms.*

Related work on reducing video burstiness is summarized in Table 1. There are a number of differences to our work. The baseline algorithm described in Trickle [25] reduces chunk throughput based on the video encoding rate with the goal of reducing wasted buffers when videos end early. The impact of this algorithm on QoE, smoothness, or neighboring traffic is not evaluated in the paper but since algorithm reduces buffer sizes it likely has some impact on QoE. In contrast, our work explores how to design an ABR algorithm to improve smoothness while achieving high QoE. Work on pacing like TCP Pacing [1] and Trickle (relative to their baseline) [25] focuses on reducing per-packet burstiness while maintaining congestion control-selected throughput. There is work [3, 44, 59] which reduces throughput below a congestion control algorithm's selected rate using mechanisms other than pacing, but this work does not preserve video QoE (typically because it treats ABR algorithms as a black box). Finally, there is some related work [9, 43] which delays the time to first byte (TTFB) of the HTTP response. This reduces buffer sizes, but does not improve smoothness over typical congestion control algorithms.

There has also been prior measurement work, observing that some video traffic reduces burstiness in practice using some of the mechanisms described above [5, 54].

Our work has some similarity to real-time video streaming systems (like FaceTime and Zoom), that are designed differently than on-demand systems (like Netflix and YouTube). These systems also need to pick bitrates and sending rates for videos. To pick bitrates, real-time systems pick an encoding bitrate for each frame, while on-demand systems pick bitrates from an offline-chosen ladder. Real-time systems are built on top of UDP and so need to decide when to send each packet, while on-demand systems have historically relied on congestion control algorithms. Because of these differences, real-time systems have less of a distinction between congestion control and bitrate adaptation schemes. For instance, the Google Congestion Control algorithm [29] for WebRTC picks sending rates using a delay-based algorithm and aims to match the encoding bitrate to the sending rate. Salsify [23] relies on packet-pair techniques, adjusts its sending rates to control queuing delay, and picks bitrates based on the chosen sending rates. In contrast, our work focuses on making *on-demand* systems friendlier while achieving comparable QoE to today's top on-demand systems.

2.3.1 The challenge of reducing burstiness with existing ABR algorithms. All prior work on reducing burstiness for video traffic falls short of achieving high video QoE because they treat ABR algorithms as a black box.

Today's ABR algorithms are designed to use either explicit measurements of available bandwidth (as in the case of throughput-based ABR algorithms) or implicit ones collected through the accumulation of a buffer (as with a buffer-based algorithm). If we reduce burstiness by reducing throughput, this changes available bandwidth and can easily cause ABR algorithms to select lower bitrates and reduce QoE.

As an example of how reducing chunk throughput can cause QoE issues, imagine a simple ABR algorithm which measures the minimum throughput x over the last few chunks and picks the highest video bitrate $< cx$ for some constant c.[3] Say we set $c = 0.5$, and picked a pacing rate of 1.5x the video bitrate. This would cause a downward spiral [30]: we would start with video bitrate B, pick a pacing rate of $1.5 \cdot B$, and measure a throughput of $x = 1.5 \cdot B$. We would then pick the highest video bitrate lower than $0.5 \cdot (1.5 \cdot B) =$

[3]This is the default dash.js algorithm when the buffer is low [64].

$0.75 \cdot B$, and would switch down. We would continue switching down until we reached the lowest bitrate. This example shows that we cannot treat ABR algorithms as a black box when reducing burstiness and maintain the same QoE.

As another example, imagine all chunks are one second long, and we pick a pace rate equal to the chunk bitrate. The chunk will download in exactly the same amount of time as it takes to play, one second. As a result, the playback buffer will not increase and remain at a near-zero level. If chunk throughput varied at all, there would be a rebuffer. For buffer-based algorithms, this also means that the buffer will not grow large enough to select a high bitrate chunk, reducing video quality. Note that picking a pace rate equal to or even lower than the chunk bitrate may be an effective approach. If the video buffer is relatively large, it may be desirable to give up some buffer growth to improve smoothness. By pacing at the chunk bitrate, the buffer can be kept at the same relatively high level. By pacing slightly below the chunk bitrate, video traffic can be made even smoother while only slightly shrinking the buffer.

3 DESIGN DISCUSSION

In this section, we will describe the key components of Sammy: the key idea behind designing an ABR algorithm for pacing, and the application-informed pacing mechanism Sammy uses to limit throughput. In Section 4, we will use these components when introducing Sammy.

3.1 ABR algorithms with paced throughput

Application-informed pacing reduces throughput below the available bandwidth of the network. With pacing, an ABR algorithm might *never* learn the available bandwidth of a network. As discussed in Section 2, existing ABR algorithms rely on measurements of available bandwidth. So how can an ABR algorithm optimize QoE with pacing?

The main idea behind our approach is that while ABR algorithms have historically *used* estimates of available bandwidth to make their decisions, they do not *need* accurate available bandwidth estimates to achieve good QoE. An ABR algorithm only needs to know whether the network can support the highest bitrate, or if it needs to reduce video quality to improve QoE.

For example, imagine streaming a video with a top bitrate of 10 Mbps. Once we have built up a small playback buffer, the ABR algorithm only needs to know whether the network throughput is high enough to sustain the top bitrate without rebuffering. If the available bandwidth is 100 Mbps or 1000 Mbps, a good algorithm would still pick the top bitrate.

Instead of relying on accurate estimates of available bandwidth, Sammy can use a pacing-informed ABR algorithm that instead solves a decision problem: *is the available bandwidth of the network enough to pick a bitrate, or not?* As we discuss in Section 4.2, many commonly used ABR algorithms implicitly rely on such a decision problem and do not need accurate estimates of available bandwidth.

An alternate approach would be to estimate available bandwidth despite pacing, for instance using packet pair techniques [37, 39], or not pacing some portion of requests. We did not pursue this, in favor of an approach that avoids exact throughput estimation in the first place.

3.2 Limiting throughput with application-informed pacing

Streaming video traffic is bursty because congestion control algorithms send as fast as the available network bandwidth, which is often higher than needed for good QoE. Our approach is to have the ABR algorithm reduce the server's sending rate with *application-informed pacing*: a new technique that allows applications to set an upper limit on the server's sending rate. By carefully limiting bursts, an ABR algorithm can reduce chunk throughput below the available network bandwidth while achieving high QoE.

In application-informed pacing, the ABR algorithm selects a pace rate and sends this rate to the server via an HTTP header. The server uses TCP Pacing as described in Section 2 and [1, 28, 71] to limit the sending rate at the server side. To achieve a desired rate of R packets per second, the server delays sending packets to ensure that there is a delay of at least $1/R$ seconds between the starts of successive packets.

Application-informed pacing runs in combination with a congestion control algorithm, and the pace rate is an upper limit on the sending rate. Congestion control algorithms can still limit the sending rate by reducing the congestion window or pace rate. If an application requests a pace rate higher than network bandwidth, congestion control algorithms will operate as normal and pick a lower sending rate. Because the resulting throughput is *at most* the requested pace rate, Application-informed pacing is TCP-fair to existing internet traffic.

Deployability. Application-informed pacing is readily deployable. TCP Pacing is already part of the Linux kernel [19], is used in production at Google [13, 47, 56], and is available in certain NICs [56]. In Linux, an HTTP server can implement application-informed pacing by setting the `SO_MAX_PACING_RATE` socket option [20] to an application-provided value.

There is CDN support for application-informed pacing. Akamai supports CMCD, a video standard that allows clients to limit server-side throughput using the `rtp` parameter [2, 6]. Fastly allows setting TCP pace rates based on the value of an HTTP header [22].

Application-informed pacing is an example of cross-layer design, and there are lots of other ways to limit a server's throughput. A system could use client receive windows to limit throughput [44], could limit a server's congestion window [25], or could use a server-side token bucket to reduce rates [3]. These techniques might be more bursty than application-informed pacing, but might be more easily deployable in certain settings.

4 SAMMY

We will now describe Sammy, our system that jointly selects bitrates and pace rates to smooth out video traffic while ensuring high QoE. In a significant shift from conventional video streaming systems, Sammy's primary mechanism for throughput selection is pace rate selection by the ABR algorithm, with congestion control acting as a backup to ensure TCP-fairness to existing systems. Sammy selects pace rates using information from the ABR algorithm, like buffer level and player state. To select video bitrates, Sammy relies on a pacing-aware ABR algorithm which ensures high QoE even without accurate estimates of available bandwidth. One of our

main contributions is showing that a wide range of existing ABR algorithms *already* do not require precise estimates of available bandwidth for high QoE.

Sammy is divided into two distinct algorithms: one for the initial phase (before playback starts), and one for the playing phase. This division follows naturally from the different QoE goals of each phase. During the initial phase, it is important to start playback quickly. After playback starts, there can be no change to play delay and the QoE goal shifts towards avoiding rebuffers with high quality.

4.1 Algorithm for the Initial Phase

The initial phase of a video is the time period between when a user initiates playback and the playback actually starts. During this phase, ABR algorithms download chunks to build up a small buffer before beginning playback. Sammy has four competing QoE goals in this phase:

(1) **High initial quality:** Sammy should pick high bitrates for the first few chunks of video playback.
(2) **Few rebuffers:** Sammy should build up a sufficiently large buffer before playback starts to avoid rebuffers.
(3) **Low play delay:** Sammy should begin playback as quickly as possible.
(4) **Smoothness:** Sammy should smooth out traffic by picking low pace rates.

In the initial phase, we will not aim to improve smoothness and will allow conventional congestion control algorithms to pick chunk throughput. If we reduce chunk throughput with the same initial quality and starting buffer size, we will be downloading the same number of bytes in a longer period of time and potentially increase play delay. The initial phase is a small fraction of traffic (typically a few seconds over a tens of minutes long session), so not pacing has a minor impact on overall smoothness.

The challenge in the initial phase is making bitrate selections with relatively few throughput measurements. ABR algorithms typically deal with this challenge by using historical throughput from the playing phase of previous sessions [34, 66]. But if Sammy reduces chunk throughput in the playing phase of previous sessions, this can change these estimates and result in lower initial quality (as shown in experiments in Section 5.5).

In the initial phase, Sammy requires an ABR algorithm whose initial bitrate selections are not affected by the throughput from the playing phase of other sessions. This can be accomplished in many ways. For an existing ABR algorithm that uses historical throughput estimates, we add separate *initial* throughput estimates and update these estimates only with throughput from the initial phase. For separate systems that predict initial throughput like CS2P [66], this can be done by supplying this system *only* with initial throughput measurements. Other ABR algorithm may need no modification, for instance an ABR algorithm which always selects the lowest quality for the first chunk, or Puffer [72] which uses statistics about the establishment of a TCP connection to estimate initial throughput.

In our experiments in Section 5, we record historical throughput measurements from *only* the initial phases of previous sessions on the same device and use them to select the initial bitrate. Sammy uses these estimates with Netflix's existing bitrate selection algorithm for the initial phase. This is described in pseudocode in Algorithm 1.

4.2 Algorithm for the Playing phase

During the playing phase, Sammy selects both a bitrate and a pace rate to balance three competing QoE goals:

(1) **High quality:** Sammy should pick high video bitrates.
(2) **Few rebuffers:** Sammy should avoid playback interruptions by keeping the buffer above zero.
(3) **Smoothness:** Sammy should smooth out traffic by picking low pace rates.

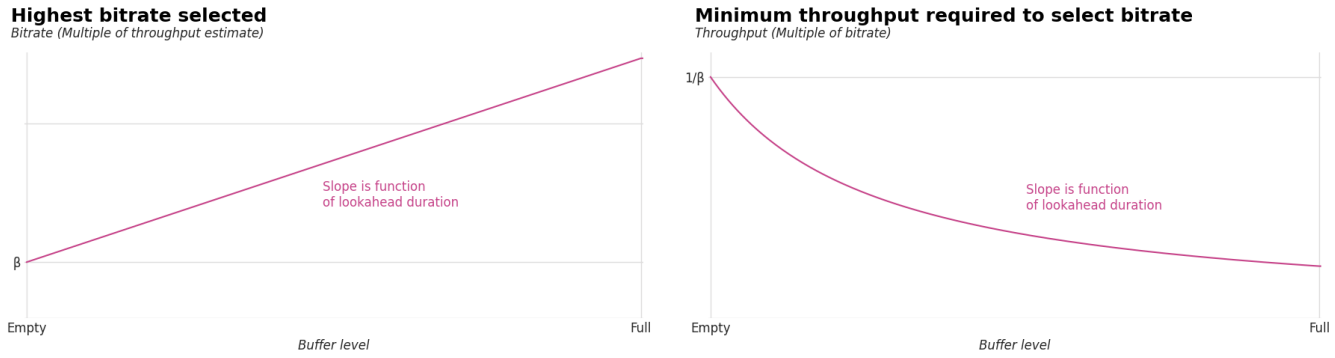
Picking a lower pace rate can affect all three goals: it improves smoothness, reduces throughput estimates (potentially impacting video quality), and causes buffers to grow more slowly (potentially impacting rebuffers).

4.2.1 Sammy's conceptual design. Sammy includes two main components in the playing phase: an ABR algorithm and a pace rate selection algorithm. Our overall strategy for the playing phase will be to take a given ABR algorithm and reduce chunk throughputs as much as possible without impacting bitrate selection. If an ABR algorithm picks the same sequence of bitrates with and without pacing, Sammy will achieve the same video quality with and without pacing. Achieving the same *QoE* with and without pacing is then just a matter of ensuring that the buffer is large enough to prevent rebuffers.

Instead of proposing a single new ABR algorithm, we will describe how to analyze a class of *pacing-aware* ABR algorithms to understand how much throughput can be reduced without impacting QoE. We then use a buffer-based algorithm [31] to pick high pace rates when the buffer is low (growing the buffer more quickly) and lower pace rates when the buffer is high (growing the buffer more slowly), while ensuring that the pace rates stay above the minimum required throughput from our analysis. This approach ensures that Sammy is easily deployable in existing, large-scale video streaming services. Surprisingly, we show that throughput can be significantly lower without impacting QoE for existing ABR algorithms.

Pacing-aware ABR algorithms: As discussed in Section 3.1, instead of relying on an ABR algorithm which *accurately estimates* available bandwidth, Sammy will use a pacing-aware ABR algorithm that relies on a decision problem: *is the available bandwidth high enough to pick a bitrate, or not?* This decision problem gives us a threshold throughput—a minimum value of the algorithm's throughput estimate that will cause it to pick the same bitrate. This threshold gives Sammy room to decrease throughput via pacing. As long as throughput estimates stay above this threshold, Sammy can decrease chunk throughput without changing bitrate decisions.

Fortunately, many existing ABR algorithms already implicitly use such a decision problem. As an example, consider a typical throughput-based algorithm: the HYB algorithm [4], modified to use lookahead (i.e. take upcoming chunk sizes into consideration). This analysis also applies to MPC algorithms [73] with appropriately chosen utility functions, ABR algorithms without lookahead, and so on. The HYB algorithm computes a throughput estimate from recent throughput measurements, and multiplies this estimate by



(a) HYB picks higher bitrates as the buffer grows.

(b) HYB has a minimum throughput needed to select a chunk.

Figure 2: By analyzing how an example throughput-based ABR algorithm (HYB) picks bitrates as a function of chunk throughput estimates (a), we can find a lower bound on pace rates to avoid impacting QoE (b). For example, to pick a bitrate with an empty buffer, HYB requires a throughput at least $1/\beta$ times higher than that bitrate.

a parameter $\beta \in [0, 1]$ to offset prediction errors. It then uses a standard buffer update equation [30] to predict how the buffer evolves over the lookahead duration. It picks the highest bitrate which keeps the buffer above zero.

To better understand the behavior of this algorithm, in Appendix A we analyze how the playback buffer evolves over time. Let D_T be the lookahead duration of the upcoming T chunks. We show that for a throughput x, bitrate r, and starting buffer size B_0, the buffer evolves according to

$$B_T = B_0 + D_T - D_T \frac{r}{\beta x}.$$

HYB picks the highest bitrate which keeps $B_T > 0$, which gives us the following constraint on the bitrate r:

$$r \leq \beta x \left(1 + \frac{B_0}{D_T}\right).$$

This function is shown in Figure 2a: as buffer size and throughput grows, HYB will pick higher bitrates[4].

As a corollary, this gives us a minimum throughput required to pick a bitrate r.

$$x \geq r\beta^{-1} \left(1 + \frac{B_0}{D_T}\right)^{-1}. \quad (1)$$

We graph this function in Figure 2b: when the buffer is empty, HYB needs an estimate of throughput equal to the bitrate divided by β. For example, if $\beta = 0.5$ and the buffer is empty, HYB will pick a bitrate provided the throughput is at least twice the bitrate. When the buffer is lower, HYB can select a bitrate with a lower throughput.

Equation 1 is the implicit function HYB uses to decide whether or not throughput is high enough to select a bitrate. In order to avoid impacting bitrate selection, we must pick a pace rate higher than this value. When the buffer is empty, we must pick a pace rate of at least $1/\beta$ times the top bitrate. When the buffer is larger, we can pick a lower pace rate without impacting bitrate selection.

[4]Previous research on ABR (e.g. [31]) has made a distinction between throughput-based and buffer-based algorithms. Interestingly, this analysis shows that while the description of HYB seems to be a classic throughput-based algorithm, implicitly it uses a buffer-based approach to select bitrates.

Sammy's pace-rate selection. Videos are encoded into a ladder of bitrates. Sammy takes the highest bitrate in this ladder, call this value r. When the buffer is empty, Sammy paces at a multiple of highest bitrate, e.g. at a rate of $c_0 \cdot r$ for some constant c_0. When the buffer is full, Sammy paces at a different multiple of the highest bitrate, e.g. at a rate of $c_1 \cdot r$ for some constant c_1. We set the parameters c_0, c_1 so that the resulting pace rate is always above the minimum throughput required to pick the highest bitrate given by Equation (1) and Figure 2. We can choose higher parameter values than this to tune the tradeoff between rebuffers and pace rates. Once Sammy selects a pace rate, it communicates this rate to the transport layer using application-informed pacing, as discussed in Section 3.2.

4.3 Sammy's implementation

In the first part of the section, we have presented a more generic version of Sammy that works with a variety of pacing-aware ABR algorithms. Here we describe the specific implementation of Sammy we use for experiments in Section 5. Sammy uses Netflix's production ABR algorithm, which is an MPC-style algorithm. This is a proprietary algorithm and we cannot describe it in detail.

During the playing phase, we use Netflix's production ABR algorithm without modification. During the initial phase, we record a separate set of historical initial throughput measurements and use these measurements in place of Netflix's existing historical throughput measurements. The distribution of initial throughput is slightly different than Netflix's existing measurements, and so accordingly we retune Netflix's initial bitrate selection logic to use these measurements without decreasing QoE. We present the results of experiments with just these changes in Section 5.7.

Algorithm 1 summarizes Sammy's implementation. To demonstrate the deployability of Sammy and show how its different components work in practice, we have released an open source prototype [61] of Sammy's playing phase. Our prototype uses off the shelf components including an unmodified dash.js player and the Fastly CDN. Our prototype likely decreases QoE relative to the production dash.js implementation (e.g. the parameters are untuned and we

make no modifications to dash.js's initial bitrate selection), and we leave achieving QoE parity as an exercise to the interested reader.

Algorithm 1 Sammy's bitrate and pace rate selection

Require: ABR algorithm, parameters $c_0, c_1 \geq 0$
  if ABR is in initial phase **then**
    bitrate $\leftarrow$ ABR select(initial throughput measurements)
    pace rate $\leftarrow$ no pacing
  else
    bitrate $\leftarrow$ ABR select(all throughput measurements)
    $B \leftarrow$ buffer/max buffer
    multiplier $\leftarrow c_1 \cdot B + c_0 \cdot (1 - B)$
    highest bitrate $\leftarrow \max_{\text{bitrate} \in \text{ladder}}$ bitrate
    pace rate $\leftarrow$ multiplier $\cdot$ highest bitrate
  end if
  return bitrate, pace rate

5 PRODUCTION EVALUATION

We implemented and evaluated Sammy on TV devices (TVs, set-top boxes, game consoles, etc...) in production at Netflix. To evaluate its performance, we ran a series of A/B tests [62, 72] to tune our algorithm and understand the tradeoffs between video QoE and congestion-related metrics. We compared Sammy to Netflix's existing extensively tested and finely-tuned production algorithm, to emphasize how Sammy can improve smoothness while maintaining or improving QoE.

Each A/B test consisted of a control group running Netflix's production algorithm, and twenty treatment groups with different settings of Sammy's parameters. We randomly picked a small fraction of Netflix's users and randomly assigned them to either control or one of the treatment groups. This resulted in a small fraction of Netflix's video sessions (< 1%). We ran the tests for about a week, and measured the values of video- and transport-level metrics for each session. Here we present the results of three experiments. Over all the sessions included in these experiment, Netflix's users watched on the order of thousands of *years* of video.

The experimental results show that Sammy significantly improves smoothness and reduces congestion-related metrics while *slightly improving* video QoE.

Parameter values: We will present results for a single set of parameters for Sammy throughout the rest of the paper, and we briefly discuss other parameter values in Section 5.3. Specifically, Sammy paces at 3.2x the maximum bitrate when the buffer is empty, and 2.8x the maximum bitrate when the buffer is full using the algorithm described in Section 4.2.

As discussed in Section 4.1, Sammy also includes changes to initial throughput estimation. We present the results of these initial changes (without pacing) in Section 5.4.

5.1 Sammy reduces congestion

We first show that Sammy significantly improves smoothness, retransmissions, and round-trip times of Netflix traffic compared to the existing production algorithm. Table 2 presents the percent changes between Sammy and Netflix's production algorithm with 95% confidence intervals.

Type	Metric	% Chg.	95% CI
Congestion	Chunk Throughput	-61.0	[-61.8, -60.2]
	% Retransmits	-35.5	[-37.8, -33.4]
	RTT	-13.7	[-16.4, -12.3]
QoE	Initial VMAF	0.14	[0.1, 0.2]
	VMAF	0.04	[0.0, 0.1]
	Play Delay	-1.29	[-2.0, -0.6]
	Rebuffers (% sess)	–	[-7.1, 4.0]
	Rebuffers (/ hr)	–	[-17.1, 3.6]

Table 2: A/B Test results for Sammy including the percentage change to control and confidence intervals. All statistically significant metric movements are improvements over Netflix's production algorithm.

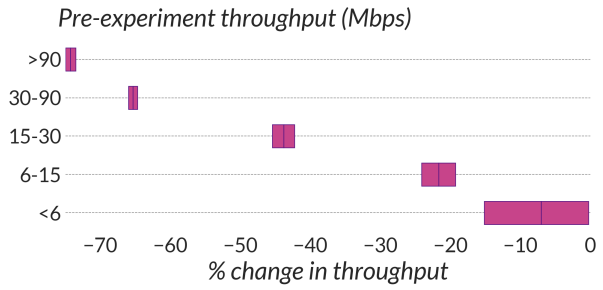


Figure 3: Reduction in chunk throughput (95% CI) split by each user's pre-experiment chunk throughput. Sammy reduces burstiness for users with pre-experiment throughput > 6 Mbps.

Improving smoothness: To measure how much Sammy smooths video traffic, we focus on the average *chunk throughput* (the throughput during "on" periods). Video clients report the average throughput for all chunk downloads in a session, weighted by download time as in Appendix A. We calculate the median of these per-session average chunk throughputs for both Netflix's production algorithm and Sammy. Sammy reduces chunk throughput by 61%. Sammy does not reduce quality, so reducing chunk throughput causes on periods to become longer, and increases the available bandwidth for neighboring traffic during on periods.

Sammy's ability to reduce throughput depends on how much higher network bandwidth is relative to maximum bitrates. This raises the question about how Sammy performs in slower networks. For all users in the A/B test, we computed their pre-experiment throughput by looking at the 95th percentile of their chunk throughput for the week before the test began. We grouped users by the range of pre-experiment throughput: <6 Mbps, 6-15 Mbps, 15-30 Mbps, 30-90 Mbps, and > 90 Mbps. We calculated average chunk throughput within each group of users, and compared Sammy's throughput of each group to that of the production algorithm. Figure 3 shows the percent change in throughput as a function of pre-experiment throughput. For users with pre-experiment throughput of more than 90 Mbps, Sammy reduces chunk throughput by 74%.

As pre-experiment throughput decreases, Sammy reduces chunk throughput less. But Sammy does significantly reduce throughput (improving smoothness) for all pre-experiment throughputs more than 6 Mbps.

Reducing network congestion: Intuitively, reducing chunk throughput and improving smoothness should translate into improvements in congestion-related metrics, specifically lower packet retransmission rates and round-trip times (RTTs). This is supported by our A/B test results.

We calculate the fraction of retransmitted bytes over all bytes sent by TCP for each session. Sammy improves the median fraction of retransmitted bytes over all sessions by 36%. We measure RTTs for each packet sent by TCP and store them for each TCP connection in a t-digest [21]. We merge the t-digests for all TCP connections in a session, and estimate the median RTT for the session. We measure the median of median RTTs over all sessions. Sammy improves RTTs by 14%.

Given Sammy reduces chunk throughput, improves smoothness, and reduces congestion for Netflix traffic, it is plausible that neighboring traffic sharing a bottleneck link with Netflix's traffic should see improvements as well. Section 6 shows in a lab setting that Sammy's improvements in congestion-related metrics can translate to QoE improvements for neighboring traffic.

5.2 Sammy improves QoE

It is not surprising that picking lower pace rates would improve smoothness and network congestion, but more surprisingly, we show this can be done at no cost to the video user experience. In our experiments, Sammy *slightly improves* QoE. These results are summarized in Table 2.

Improving quality and play delay: We measure video quality by Video Multi-method Assessment Fusion (VMAF) [10], a method for estimating a viewer's perception of a video's visual quality. We calculate a time-weighted average of VMAF to get a score for each session, and measure the median score over all sessions. Sammy *slightly* increases overall VMAF, which is driven primarily by an increase in initial VMAF (the VMAF during the first twenty seconds of video playback). In other words, Sammy's video quality is slightly higher than with Netflix's production algorithm.

We note that this is a very minor improvement to VMAF. It is a statistically significant improvement in our experiments, but it is a small and potentially imperceptible improvement. The more important point is that Sammy maintains a QoE that is at least on par to Netflix's existing, finely-tuned production algorithm, with more than 60% lower per-chunk throughput.

Sammy also slightly improves play delay by about 1.3%. This may seem surprising since we do not do any pacing in the initial part of the section. As we show in Section 5.4, the improvements to QoE (play delay included) come primarily from using *only* estimates of initial throughput during the initial phase.

Maintaining rebuffers: Sammy has no statistically significant impact on any other aspect of QoE. There is no significant change in rebuffers: both the fraction of sessions that have at least one rebuffer, and the number of rebuffers per hour streamed.

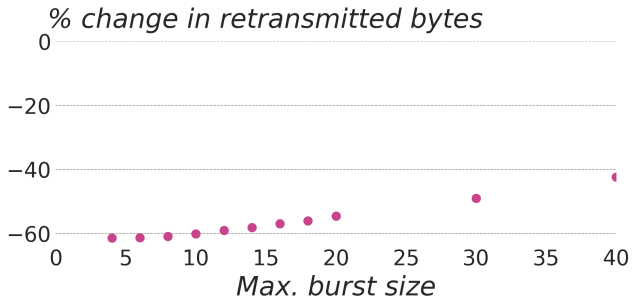


Figure 4: Change in retransmissions as a function of the pacing burst size in a production A/B test. Lower burst sizes improve retransmissions.

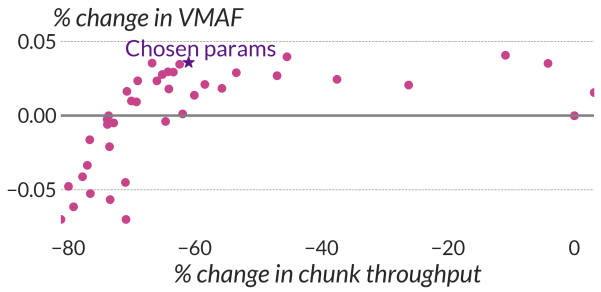


Figure 5: Tradeoff between video quality (VMAF) and chunk throughput for different choices of parameters.

5.3 Tradeoffs and parameter settings

Sammy has a number of parameters that can be tuned, including parameters for pace rate selection and for the chosen ABR algorithm. We used Ax [51] to search the parameter space and find a Pareto improvement to all metrics of interest across multiple rounds of A/B testing.

Different parameter settings allow us to trade off between chunk throughput and quality, as shown in Figure 5. Each point represents one treatment group in an A/B test, each with a different value of parameter settings. The x-axis is the % change in chunk throughput for that group, and the y-axis is the % change in VMAF. The parameters we selected reduced chunk throughput by 61% relative to control, while increasing VMAF by 0.04%. Other parameter settings give other points on this tradeoff. Eventually, decreasing throughput results in a decrease in VMAF.

5.4 QoE differences are primarily from initial phase

Sammy includes two sets of changes over Netflix's production ABR algorithm: reducing chunk throughput using pacing, and changes to the initial phase including using estimates of initial throughput and retuning Netflix's initial bitrate selection logic. Here we report on the results of an A/B test including only the changes to the initial phase, and not pacing.

Metric	% Chg.	95% CI
Initial VMAF	0.30	[0.28,0.35]
VMAF	–	[-1.8e-5, 1.7e-4]
Play Delay	-0.40	[-0.7, -0.1]
Rebuffers (% sess)	–	[-1.6,1.8]
Rebuffers (/ hr)	–	[-3.1,4.2]

Table 3: A/B Test results for Sammy's changes to initial throughput estimation and bitrate selection. All statistically significant metric movements are improvements over Netflix's production algorithm.

In this A/B test, there was a slight improvement in initial VMAF of about 0.3%, in play delay of about 0.4%, and in no other metrics. These results are run as a separate A/B test, and so shouldn't be directly compared to the results of Sammy in Table 2. But the direction of improvement and magnitude is similar in the two tests.

These results, plus our intuition that pacing late in the session should not impact the initial phase (including initial VMAF and play delay), suggests that Sammy's QoE improvements do not come from pacing. Instead, the results suggest that the improvements come primarily from the changes to initial bitrate selection.

5.5 A baseline approach reduces QoE

Sammy works hard to avoid reducing QoE with pacing, and a natural question is whether this work is necessary. Why not just pick a pace rate a bit higher than the maximum bitrate and call it a day? We ran an experiment which shows that this approach underperforms Sammy in all of our goals.

We ran an experiment with the production Netflix ABR algorithm in which we limited the pacing rate for each chunk (including in the initial phase) to 4x the maximum bitrate. We made no other changes. Pacing in this way reduced chunk throughput by 53% and we observed a degradation in most of the major components of video QoE: play delay increased by 6%, and VMAF decreased by 0.2%. The play delay increase was enough to reduce the overall level of streaming, causing the experiment to be automatically stopped by safety systems.

Sammy outperforms this approach in both congestion and QoE-related metrics. Sammy achieves a *higher* reduction in chunk throughput of 61% while *improving* QoE. If we instead chose parameters from Figure 5 which reduced QoE, Sammy would achieve a higher throughput reduction of more than 80% for a much lower VMAF reduction of 0.07%.

5.6 Effect of burst size

With pacing, there is an option of how large a burst to send at a time. To pace at 12 Mbps with 1500 byte MTU, we could send one packet every 1 ms, two packets every 2 ms, or 10 packets every 10ms, and so on. Intuitively, there is a tradeoff in picking the burst size: smaller bursts should improve congestion-related metrics, but also reduce opportunities for segmentation offload which can increase CPU usage.

Netflix's TCP implementation's default behaviour is to limit line-rate bursts to no more than 40 packets at a time. We ran an experiment where we paced at a constant 2x the maximum bitrate, and adjusted the per-packet bursts from 4 packets up to 40 packets. Figure 4 shows the results of this experiment.

Pacing with a burst size of 40 packets corresponds to only reducing chunk throughput, and not reducing the maximum possible size of per-packet bursts. This reduces retransmissions by 40% relative to not pacing. As the maximum burst size decreases, retransmits reduce by up to 60% relative to not pacing. But as the burst size decreases, there is no statistically difference in either chunk throughput or video QoE metrics.

This result shows why it is beneficial to use TCP Pacing instead of capping the congestion window as in prior work [25]. In our experiments, we use a burst size of 4 packets for CPU efficiency. By reducing the burst size from 40 (as it would be if we capped the congestion window) to a burst size of 4, we improve retransmissions by an additional 20%.

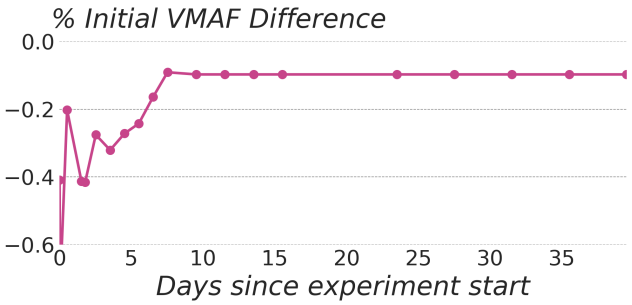


Figure 6: Initial quality difference over time during an A/B test. The treatment algorithm is missing historical data at the beginning of the experiment, and so performs worse over the entire experiment.

5.7 Effect of historical data

As described in Section 4.1, Sammy and prior work use historical throughput measurements for initial bitrate selection. Doing so creates a dependency between successive sessions: the throughput at the beginning of one session impacts the bitrate selection decisions at the beginning of the next. Using historical data improves performance, but the dependency creates challenges for evaluation.

As an example, we ran an experiment simulating introducing a new historical estimate. The treatment group started with no historical measurements, while the control group had historical measurements. Both groups updated historical throughput with the same estimates, and there were no other differences between the groups. Figure 6 shows the percent difference in initial quality over the course of the experiment. The treatment group started with much lower initial quality and surprisingly it stayed lower over the course of the experiment. It took a week for the initial quality of the treatment group to reach its closest point to the control group.

To deal with this challenge, we reset historical throughput information in both treatment and control groups in all experiments to enable an "apples-to-apples" comparison between the two.

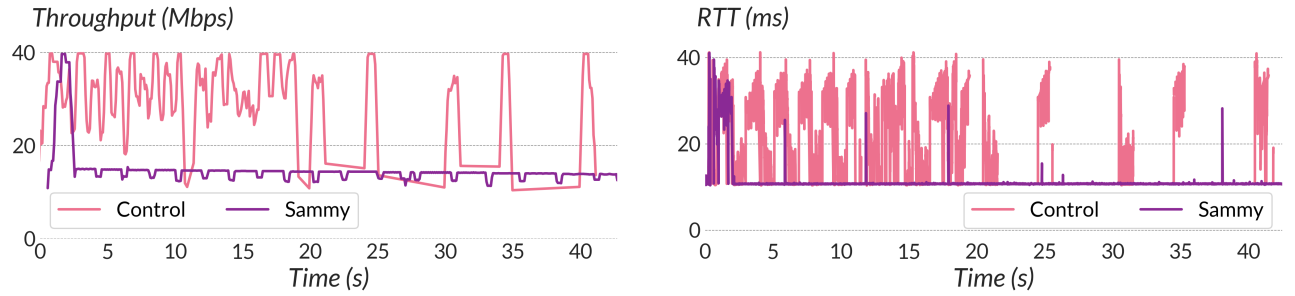
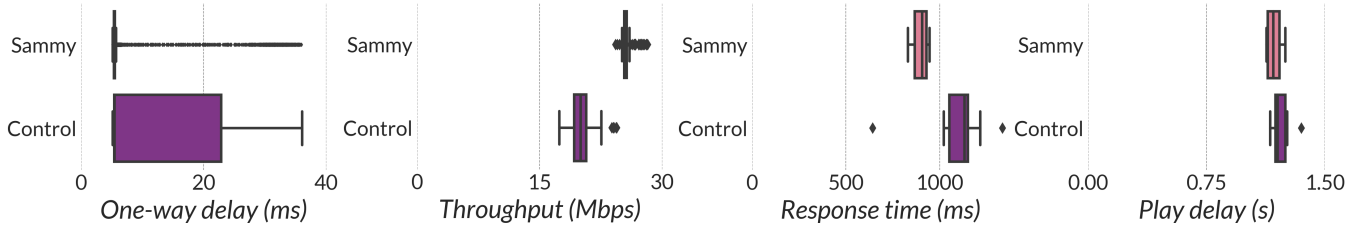


Figure 7: Throughput and RTT for a single Sammy flow running in a lab environment compared to Netflix's production algorithm. After 3 seconds, Sammy reduces chunk throughput enough to avoid congesting the link. Reducing chunk throughput helps it avoid on-off periods beginning for control traffic around 25 seconds.



(a) Neighboring UDP traffic **(b) Neighboring TCP traffic** **(c) Neighboring HTTP traffic** **(d) Neighboring video traffic**

Figure 8: In the lab, Sammy improves QoE for neighboring traffic relative to control for (a) UDP one-way delay, (b) TCP throughput, (c) HTTP response time, and (d) video play delay.

6 IMPROVING QOE OF NEIGHBORS

In this section, we will present lab experiments where we measure how Sammy improves the QoE of neighboring traffic. The previous section shows how Sammy reduces chunk throughput and congestion-related metrics at scale. Lab experiments with a single setting are clearly not representative of most traffic on the internet, so the goal of these experiments is to illustrate how Sammy can improve the QoE of a few neighbors that might share its bottleneck.

Without Sammy, the video traffic fully utilizes the link and fills up the queue, in turn impacting neighboring traffic. Sammy smooths out traffic, and chunk throughput drops to below half the network capacity. This behavior avoids congesting the link (reducing queueing delay for neighboring traffic), and gives neighboring traffic more bandwidth to use for itself.

Experiment setup. In all experiments, we use a 40 Mbps link with a 5ms RTT, and a queue size of 4 times the bandwidth-delay product. Sammy plays a video with a maximum bitrate of 3.3 Mbps. We run an experiment where a video session using Netflix's production algorithm runs at the same time as a neighboring application. We repeat the same experiment using Sammy and observe how the neighbor's QoE changes.

Sammy on its own. To understand how Sammy improves performance for neighboring traffic, we will first look at how it performs on its own. Figure 7 shows the throughput and RTT for a single Sammy flow, compared to a single control flow running Netflix's production ABR algorithm.

At the beginning of the session, both Sammy and control send as fast as possible during the initial phase: fully utilizing the network and filling up the queue. Playback starts after about three seconds, at which point Sammy begins pacing. The pace rate it picks is about 15 Mbps—low enough to avoid congesting the link, so queuing delay goes to zero and the RTT is the minimum of 5ms. Over the rest of the session, Sammy decreases the throughput to about 13 Mbps—below its TCP-fair share rate of 20 Mbps when it shares a link with neighboring traffic.

The change in metrics for this session is comparable to the overall change in metrics for the A/B test in Section 5. For this session Sammy reduces throughput by 53% (slightly less than in the A/B test) and RTTs by 47% (about four times more than the A/B test). Note that in the A/B test, Sammy shares networks with neighboring traffic running typical congestion control algorithms that keep queues full. It is possible that if the neighboring traffic instead used Sammy, the congestion reduction could be even larger [62].

When neighboring traffic shares this particular network with Sammy, it will experience an extra 5 Mbps of available bandwidth and no additional queueing delay. This leads to the following benefits (shown in Figure 8):

UDP: We first run an experiment where the neighboring traffic is a 5 Mbps paced UDP flow. The one-way delay measured for UDP packets is shown in Figure 8a. Sammy eliminates queueing delay for the UDP traffic, reducing the one-way delay by 51%. Without Sammy, video traffic keeps the queue full (see Figure 7) and the UDP traffic experiences queueing delay. Sammy sends no faster

than 15 Mbps during playback, so the queue stays empty even with 5 Mbps of UDP traffic.

TCP: We next run an experiment where the neighboring traffic is a standard, congestion window limited TCP Reno connection (the congestion control algorithm Netflix uses by default). The TCP connection begins 10 seconds after playback starts. Its throughput is shown in Figure 8b. Without Sammy, the TCP flow gets an average of 20 Mbps (its TCP-fair share of throughput). Sammy increases throughput for the TCP flow by 28% to an average of 25.7 Mbps. Any other congestion control algorithms that splits bandwidth equally without Sammy and fully utilizes the link with Sammy would have similar results.

HTTP: The next experiment demonstrates the benefits of Sammy to neighboring HTTP requests. We repeatedly issue 3MB HTTP requests during video playback. We measure the HTTP response time, the time between when the first byte of the request was issued and the last byte of the response was received. The results are shown in Figure 8c—Sammy improves average HTTP response times by 18%, reducing them from 1095ms to 898ms.

Streaming video: We run another experiment to measure the impact of Sammy on a neighboring video session. We start one Sammy session, and after a few seconds we start a neighboring session using Netflix's production algorithm. Figure 8d shows the play delay for the neighboring session. Over four trials, Sammy consistently improves the play delay of its neighbor by 4%—an average of 50ms. Whenever a streaming service shares a bottleneck with itself, Sammy can improve the service's own play delay. This result gives streaming services an incentive to deploy Sammy.

7 CONCLUSION

Our approach shows that ABR algorithms can dramatically reduce the burstiness of video traffic without reducing QoE. In our experiments run at scale at Netflix, Sammy is able to reduce the median chunk throughput by 61%, reducing retransmissions by 36% and RTTs by 14%. These improvements to network congestion came with no harm to video QoE. In fact, we observed a small improvement in video quality (both initially and overall) and play delay, and no statistically significant changes to rebuffers. Because Sammy does not aim to fully utilize the link, there is more bandwidth available for neighboring traffic during Sammy's on periods. Our lab experiments illustrate how this can improve performance for neighboring traffic: Sammy reduces delay for a neighboring UDP flow by 51%, increases throughput for a neighboring TCP flow by 28%, reduces response times for neighboring HTTP traffic by 18%, and even reduces play delay for neighboring video traffic by 4%. We leave deeper investigations of the impact on neighboring traffic to future work, and would be especially interested in experiments to measure the impact at scale.

In many ways, today's video streaming architecture is a *response to* the two control loops managed by ABR and congestion control. Congestion control algorithms learn and acquire their fair share of bandwidth on a packet-by-packet timescale; and in turn ABR algorithms adapt bitrates at chunk-by-chunk timescale. Given the steps taken in this paper, a compelling future path forward is to consider a *single control loop* to both determine the video bitrate and when to transmit each bit of the stream over the network. This algorithm could jointly optimize video QoE and transport-layer goals like congestion and fairness, and could avoid the pitfalls associated with two interacting control loops [30]. We leave that work for others. Here, we instead have the ABR algorithm limit the server's sending rate, so as to allow more rapid deployment with current video streaming services, and keeping with standard practice of sharing the internet using congestion control algorithms. One could also imagine a range of options between the two, where ABR algorithms share more and more information with the underlying transport layer. Broadly, the significant empirical results found in this paper suggest that such innovations have the potential for significant impact not only on video streaming services, but the internet at large.

We view our work as a starting point for using application-level logic to smooth out internet traffic. We have shown that video streaming does not always need the maximum throughput a network can achieve. The layering architecture of the internet encourages other applications to use a similar strategy of allowing congestion control algorithms to select the maximum throughput without application input. By using details about the behavior of other applications, we may be able to make other types of internet traffic into friendlier neighbors as well.

REFERENCES

[1] Amit Aggarwal, Stefan Savage, and Thomas E Anderson. 2000. Understanding the Performance of TCP Pacing. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, Vol. 3. IEEE, Tel Aviv, Israel, 1157–1165. https://doi.org/10.1109/INFCOM.2000.832483

[2] Akamai. 2023. Common Media Client Data & AMD. (Feb. 2023). https://techdocs.akamai.com/adaptive-media-delivery/docs/common-media-client-data-amd

[3] Saamer Akhshabi, Lakshmi Anantakrishnan, Constantine Dovrolis, and Ali C. Begen. 2013. Server-Based Traffic Shaping for Stabilizing Oscillating Adaptive Streaming Players. In *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '13*. ACM Press, Oslo, Norway, 19–24. https://doi.org/10.1145/2460782.2460786

[4] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, Budapest Hungary, 44–58. https://doi.org/10.1145/3230543.3230558

[5] João Taveira Araújo, Raúl Landa, Richard G. Clegg, George Pavlou, and Kensuke Fukuda. 2014. A Longitudinal Analysis of Internet Rate Limitations. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 1438–1446. https://doi.org/10.1109/INFCOM.2014.6848078

[6] Consumer Technology Association. 2020. *CTA Specification: Web Application Video Ecosystem - Common Media Client Data*. Technical Report CTA-5004. Consumer Technology Association. https://cdn.cta.tech/cta/media/media/resources/standards/pdfs/cta-5004-final.pdf

[7] Rukshani Athapathu, Ranysha Ware, Aditya Abraham Philip, Srinivasan Seshan, and Justine Sherry. 2020. Prudentia: Measuring Congestion Control Harm on the Internet. In *N2Women Workshop*. 2. http://www.justinesherry.com/papers/athapathu-n2women20.pdf

[8] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, Mark Stemm, and Randy H. Katz. 1998. TCP Behavior of a Busy Internet Server: Analysis and Improvements. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, Vol. 1. 252–262 vol.1. https://doi.org/10.1109/INFCOM.1998.659661

[9] Abdelhak Bentaleb, May Lim, Mehmet N. Akcay, Ali C. Begen, and Roger Zimmermann. 2021. Common Media Client Data (CMCD): Initial Findings. In *Proceedings of the 31st ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM, Istanbul Turkey, 25–33. https://doi.org/10.1145/3458306.3461444

[10] Netflix Technology Blog. 2017. Toward A Practical Perceptual Video Quality Metric. (April 2017). https://medium.com/netflix-techblog/toward-a-practical-perceptual-video-quality-metric-653f208b9652

[11] Bob Briscoe. 2007. Flow Rate Fairness: Dismantling a Religion. *ACM SIGCOMM Computer Communication Review* 37, 2 (March 2007), 63–74. https://doi.org/10.

1145/1232919.1232926

[12] Yi Cao, Arpit Jain, Kriti Sharma, Aruna Balasubramanian, and Anshul Gandhi. 2019. When to Use and When Not to Use BBR: An Empirical Analysis and Evaluation Study. In *Proceedings of the Internet Measurement Conference*. ACM, Amsterdam Netherlands, 130–136. https://doi.org/10.1145/3355369.3355579

[13] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-Based Congestion Control. *Commun. ACM* 60, 2 (Jan. 2017), 58–66. https://doi.org/10.1145/3009824

[14] David D Clark. 1988. The Design Philosophy of the DARPA Internet Protocols. *ACM SIGCOMM Computer Communication Review* 18, 4 (Aug. 1988), 106–114. https://doi.org/10.1145/52325.52336

[15] Federal Communications Commission. 2010. *Broadband Performance*. Technical Report OBI Technical Paper No. 4. Federal Communications Commission. https://transition.fcc.gov/national-broadband-plan/broadband-performance-paper.pdf

[16] Federal Communications Commission. 2021. *Measuring Fixed Broadband - Eleventh Report*. Technical Report. Federal Communications Commission. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-eleventh-report

[17] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. 2011. Understanding the Impact of Video Quality on User Engagement. *ACM SIGCOMM Computer Communication Review* 41, 4 (Aug. 2011), 362–373. https://doi.org/10.1145/2043164.2018478

[18] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, P Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *NSDI*. 15.

[19] Eric Dumazet. 2013. Pkt_sched: Fq: Fair Queue Packet Scheduler [LWN.Net]. (Aug. 2013). https://lwn.net/Articles/564825/

[20] Eric Dumazet. 2015. Tc-Fq(8) - Linux Manual Page. (Sept. 2015). https://man7.org/linux/man-pages/man8/tc-fq.8.html

[21] Ted Dunning. 2021. The T-Digest: Efficient Estimates of Distributions. *Software Impacts* 7 (Feb. 2021), 100049. https://doi.org/10.1016/j.simpa.2020.100049

[22] Fastly. 2023. Fastly Developer Hub. (Feb. 2023). https://developer.fastly.com/reference/vcl/variables/client-connection/client-socket-pace/

[23] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. 2018. Salsify: Low-Latency Network Video Through Tighter Integration Between a Video Codec and a Transport Protocol. In *NSDI*.

[24] Jim Gettys, Kathleen Nichols, and Kathleen Nichols. 2012. Bufferbloat: Dark Buffers in the Internet. *Commun. ACM* 55, 1 (2012), 15. https://doi.org/10.1145/2063176.2063196

[25] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. 2012. Trickle: Rate Limiting YouTube Video Streaming. In *USENIX ATC*. 6.

[26] Carlo Augusto Grazia, Martin Klapez, and Maurizio Casoni. 2021. The New TCP Modules on the Block: A Performance Evaluation of TCP Pacing and TCP Small Queues. *IEEE Access* 9 (2021), 129329–129336. https://doi.org/10.1109/ACCESS.2021.3113891

[27] Bruce Hajek. 1983. The Proof of a Folk Theorem on Queuing Delay with Applications to Routing in Networks. *J. ACM* 30, 4 (Oct. 1983), 834–851. https://doi.org/10.1145/2157.322409

[28] Janey C. (Janey Ching-Iu) Hoe. 1995. *Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes*. Thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/36971

[29] Stefan Holmer, Henrik Lundin, Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. 2015. *A Google Congestion Control Algorithm for Real-Time Communication*. Internet Draft draft-alvestrand-rmcat-congestion-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-alvestrand-rmcat-congestion-03

[30] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. 2012. Confused, Timid, and Unstable: Picking a Video Streaming Rate Is Hard. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference - IMC '12*. ACM Press, Boston, Massachusetts, USA, 225. https://doi.org/10.1145/2398776.2398800

[31] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. 2014. A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM - SIGCOMM '14*. ACM Press, Chicago, Illinois, USA, 187–198. https://doi.org/10.1145/2619239.2626296

[32] Per Hurtig, Habtegebreil Haile, Karl-Johan Grinnemo, Anna Brunstrom, Eneko Atxutegi, Fidel Liberal, and Ake Arvidsson. 2018. Impact of TCP BBR on CUBIC Traffic: A Mixed Workload Evaluation. In *2018 30th International Teletraffic Congress (ITC 30)*. IEEE, Vienna, 218–226. https://doi.org/10.1109/ITC30.2018.00040

[33] Geoff Huston. 2018. TCP and BBR. (May 2018). https://ripe76.ripe.net/presentations/10-2018-05-15-bbr.pdf

[34] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. 2016. CFA: A Practical Prediction System for Video QoE Optimization. In *NSDI*.

[35] Junchen Jiang, Vyas Sekar, and Hui Zhang. 2014. Improving Fairness, Efficiency, and Stability in HTTP-Based Adaptive Video Streaming With Festive. *IEEE/ACM Transactions on Networking* 22, 1 (Feb. 2014), 326–340. https://doi.org/10.1109/TNET.2013.2291681

[36] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols. In *Proceedings of the 2017 Internet Measurement Conference*. ACM, London United Kingdom, 290–303. https://doi.org/10.1145/3131365.3131368

[37] Seong-ryong Kang, Xiliang Liu, Min Dai, and Dmitri Loguinov. 2004. Packet-Pair Bandwidth Estimation - Stochastic Analysis of a Single Congested Node. *ICNP* (2004), 316–325. https://doi.org/10.1109/ICNP.2004.1348121

[38] Damian Karwowski, Tomasz Grajek, Krzysztof Klimaszewski, Olgierd Stankiewicz, Jakub Stankowski, and Krzysztof Wegner. 2017. 20 Years of Progress in Video Compression – from MPEG-1 to MPEG-H HEVC. General View on the Path of Video Coding Development. In *International Conference on Image Processing and Communications*, Vol. 525. 3–15. https://doi.org/10.1007/978-3-319-47274-4_1

[39] S Keshav. 1995. A Control-Theoretic Approach to Flow Control. *ACM SIGCOMM Computer Communication Review* (1995). http://dl.acm.org/citation.cfm?id=205463

[40] S. Shunmuga Krishnan and Ramesh K. Sitaraman. 2012. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-Experimental Designs. In *Proceedings of the 2012 Internet Measurement Conference (IMC '12)*. Association for Computing Machinery, New York, NY, USA, 211–224. https://doi.org/10.1145/2398776.2398799

[41] Jonathan Kua, Grenville Armitage, and Philip Branch. 2017. A Survey of Rate Adaptation Techniques for Dynamic Adaptive Streaming Over HTTP. *IEEE Communications Surveys & Tutorials* 19, 3 (2017), 1842–1866. https://doi.org/10.1109/COMST.2017.2685630

[42] Ike Kunze, Jan Ruth, and Oliver Hohlfeld. 2020. Congestion Control in the Wild—Investigating Content Provider Fairness. *IEEE Transactions on Network and Service Management* 17, 2 (June 2020), 1224–1238. https://doi.org/10.1109/TNSM.2019.2962607

[43] Will Law. 2022. Clever Monkeys Communicating Discreetly. (Oct. 2022). https://2022.demuxed.com/

[44] Ahmed Mansy, Bill Ver Steeg, and Mostafa Ammar. 2013. SABRE: A Client Based Technique for Mitigating the Buffer Bloat Effect of Adaptive Video Flows. In *Proceedings of the 4th ACM Multimedia Systems Conference on - MMSys '13*. ACM Press, Oslo, Norway, 214–225. https://doi.org/10.1145/2483977.2484004

[45] Aditya Mavlankar, Liwei Guo, Anush Moorthy, and Anne Aaron. 2020. Optimized Shot-Based Encodes for 4K: Now Streaming! (Aug. 2020). https://netflixtechblog.com/optimized-shot-based-encodes-for-4k-now-streaming-47b516b10bbb

[46] Tong Meng, Neta Rozen Schiff, P. Brighten Godfrey, and Michael Schapira. 2020. PCC Proteus: Scavenger Transport And Beyond. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 615–631. https://doi.org/10.1145/3387514.3405891

[47] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, London United Kingdom, 537–550. https://doi.org/10.1145/2785956.2787510

[48] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. 2019. End-to-End Transport for Video QoE Fairness. In *Proceedings of the ACM Special Interest Group on Data Communication - SIGCOMM '19*. ACM Press, Beijing, China, 408–423. https://doi.org/10.1145/3341302.3342077

[49] Mirko Palmer, Malte Appel, Kevin Spiteri, Balakrishnan Chandrasekaran, Anja Feldmann, and Ramesh K. Sitaraman. 2021. VOXEL: Cross-Layer Optimization for Video Streaming with Imperfect Transmission. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*. ACM, Virtual Event Germany, 359–374. https://doi.org/10.1145/3485983.3494864

[50] Larry Peterson, Lawrence Brakmo, and Bruce Davie. 2022. *TCP Congestion Control: A Systems Approach* (version 1.1-dev ed.). Self-published. https://tcpcc.systemsapproach.org/

[51] Meta Platforms. 2023. Ax: Adaptive Experimentation Platform. (Feb. 2023). https://ax.dev

[52] Michele Polese, Federico Chiariotti, Elia Bonetto, Filippo Rigotto, Andrea Zanella, and Michele Zorzi. 2019. A Survey on Recent Advances in Transport Layer Protocols. *IEEE Communications Surveys & Tutorials* 21, 4 (2019), 3584–3608. https://doi.org/10.1109/COMST.2019.2932905 arXiv:cs/1810.03884

[53] World Snail Racing. 2023. World Snail Racing Championships. (Feb. 2023). http://www.snailracing.net/

[54] Ashwin Rao, Arnaud Legout, Yeon-sup Lim, Don Towsley, Chadi Barakat, and Walid Dabbous. 2011. Network Characteristics of Video Streaming Traffic. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*. ACM, Tokyo Japan, 1–12. https://doi.org/10.1145/2079296.2079321

[55] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. 2016. LEDBAT: The New BitTorrent Congestion Control Protocol. In *2010 19th International Conference on Computer Communications and Networks (ICCCN 2010)*. IEEE, 1–6. https://doi.org/10.1109/ICCCN.2010.5560080

[56] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End Hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, Los Angeles CA USA, 404–417. https://doi.org/10.1145/3098822.3098852

[57] Sandvine. 2023. *Sandivne Global Internet Phenomena Report 2023*. Technical Report. Sandvine.

[58] Yusuf Sani, Andreas Mauthe, and Christopher Edwards. 2017. Adaptive Bitrate Selection: A Survey. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2985–3014. https://doi.org/10.1109/COMST.2017.2725241

[59] Kozo Satoda, Hiroshi Yoshida, Hironori Ito, and Kazunori Ozawa. 2012. Adaptive Video Pacing Method Based on the Prediction of Stochastic TCP Throughput. In *2012 IEEE Global Communications Conference (GLOBECOM)*. 1944–1950. https://doi.org/10.1109/GLOCOM.2012.6503400

[60] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. 2018. Towards a Deeper Understanding of TCP BBR Congestion Control. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, Zurich, Switzerland, 1–9. https://doi.org/10.23919/IFIPNetworking.2018.8696830

[61] Bruce Spang. 2023. Sammy: Making Video Traffic a Friendlier Internet Neighbor. (June 2023). https://sammy.brucespang.com

[62] Bruce Spang, Veronica Hannan, Shravya Kunamalla, Te-Yuan Huang, Nick McKeown, and Ramesh Johari. 2021. Unbiased Experiments in Congested Networks. In *Proceedings of the 21st ACM Internet Measurement Conference (IMC '21)*. Association for Computing Machinery, New York, NY, USA, 80–95. https://doi.org/10.1145/3487552.3487851

[63] Speedtest. 2023. Internet Speed around the World. (Feb. 2023). https://www.speedtest.net/global-index

[64] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. 2019. From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player. *ACM Transactions on Multimedia Computing, Communications, and Applications* 15, 2s (April 2019), 1–29. https://doi.org/10.1145/3336497

[65] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. 2020. BOLA: Near-Optimal Bitrate Adaptation for Online Videos. *IEEE/ACM Transactions on Networking* 28, 4 (Aug. 2020), 1698–1711. https://doi.org/10.1109/TNET.2020.2996964

[66] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. 2016. CS2P: Improving Video Bitrate Selection and Adaptation with Data-Driven Throughput Prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, Florianopolis Brazil, 272–285. https://doi.org/10.1145/2934872.2934898

[67] Linus Torvalds. 2021. Tcp_input.c - Linux (v5.11-Rc5). (Jan. 2021). https://github.com/torvalds/linux/blob/2ab38c17aac10bf55ab3efde4c4db3893d8691d2/net/ipv4/tcp_input.c#L873

[68] Belma Turkovic, Fernando A. Kuipers, and Steve Uhlig. 2019. Fifty Shades of Congestion Control: A Performance and Interactions Evaluation. *arXiv:1903.03852 [cs]* (March 2019). arXiv:cs/1903.03852 http://arxiv.org/abs/1903.03852

[69] Belma Turkovic, Fernando A. Kuipers, and Steve Uhlig. 2019. Interactions between Congestion Control Algorithms. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*. 161–168. https://doi.org/10.23919/TMA.2019.8784674

[70] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Modeling BBR's Interactions with Loss-Based Congestion Control. In *Proceedings of the Internet Measurement Conference*. ACM, Amsterdam Netherlands, 137–143. https://doi.org/10.1145/3355369.3355604

[71] David X. Wei, Pei Cao, and Steven H. Low. 2006. TCP Pacing Revisited. In *INFOCOM*, Vol. 2. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.2658&rep=rep1&type=pdf

[72] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in Situ: A Randomized Experiment in Video Streaming. In *NSDI*. Santa Clara, CA, USA, 16. https://www.usenix.org/system/files/nsdi20-paper-yan.pdf

[73] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. 2015. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 325–338. https://doi.org/10.1145/2785956.2787486

[74] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. 2017. An Axiomatic Approach to Congestion Control. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, Palo Alto CA USA, 115–121. https://doi.org/10.1145/3152434.3152445

[75] Xu Zhang, Yiyang Ou, Siddhartha Sen, and Junchen Jiang. 2021. SENSEI: Aligning Video Streaming Quality with Dynamic User Sensitivity. In *NSDI*.

[76] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. 2016. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. ACM, Irvine California USA, 313–327. https://doi.org/10.1145/2999572.2999593

A APPENDIX: RELATIONSHIP BETWEEN THROUGHPUT, BITRATE, AND BUFFERS

Appendices are supporting material that has not been peer-reviewed.

In this appendix, we will formalize the relationship between chunk throughput, bitrates, and buffer sizes.

There are a number of chunk selection opportunities, which occur at steps $t \in \{1, \ldots, T\}$. The chunk at time t has a duration d_t and size s_t which is selected by the ABR algorithm.

One step: Each time we select a chunk at time t, the buffer will evolve in some way. Let Δ_t be the time it takes to add chunk t to the buffer. For simplicity we will assume the buffer never becomes full and never becomes empty, but we could instead keep track of the amount of full and empty time after each chunk downloads.

The buffer evolution is given by the following standard equation [30]:

$$B_{t+1} = B_t + d_t - \Delta_t. \tag{2}$$

We will define the bitrate of chunk t as

$$r_t = \frac{s_t}{d_t}. \tag{3}$$

We will define x_t, the throughput of chunk t (e.g. in units of bits per second), as

$$x_t = \frac{s_t}{\Delta_t} = \frac{r_t d_t}{\Delta_t}. \tag{4}$$

Note that with these definitions,

$$B_{t+1} = B_t + d_t - d_t \frac{r_t}{x_t}. \tag{5}$$

Multiple steps: In addition to a single buffer step, we will also be interested in how the buffer evolves over T steps. Define the total duration D_T as

$$D_T = \sum_{t=1}^{T} d_t.$$

When playback starts, the buffer starts at some size B_0. If we expand (2), we have the following buffer size at time T+1. We could interpret this as being the buffer right after we finish downloading the last chunk.

$$B_{T+1} = B_0 + D_T - \sum_{t=1}^{T} \Delta_t. \tag{6}$$

We will define S_T to be the total size of chunks we download by time T

$$S_T = \sum_{t=1}^{T} s_t. \tag{7}$$

Define the time-average bitrate by

$$\bar{r} = \frac{\sum_{t=1}^{T} d_t r_t}{D_T} = \frac{S_T}{D_T}. \tag{8}$$

And finally we will define the time-average throughput as:

$$\bar{x} = \frac{\sum_{t=1}^{T} \Delta_t x_t}{\sum_{t=1}^{T} \Delta_t} = \frac{S_T}{\sum_{t=1}^{T} \Delta_t}. \tag{9}$$

With these definitions, the behavior of the buffer over T steps is the same as the behavior over one step, averaged. This is formalized

by the following theorem, which should be compared to the single step update equation in (5).

Theorem A.1. *In the above setting,*

$$B_{T+1} = B_0 + D_T - D_T \frac{\bar{r}}{\bar{x}}.$$

Proof. Given (6), all we need to show is that $D_T \frac{\bar{r}}{\bar{x}} = \sum_{t=1}^{T} \Delta_t$. Substituting the definition of $\bar{r}$, we have

$$D_T \frac{\bar{r}}{\bar{x}} = D_T \frac{S_T/D_T}{\bar{x}} = \frac{S_T}{\bar{x}}.$$

By the definition of $\bar{x}$, we have

$$D_T \frac{\bar{r}}{\bar{x}} = \frac{S_T}{S_T/\sum_{t=1}^{T} \Delta_t} = \sum_{t=1}^{T} \Delta_t.$$

□

A.1 Discussion

The main use of this theorem in our paper is to understand which bitrates our algorithm will pick, by understanding how a simulated buffer evolves as a function of bitrate and throughput. But this theorem is a much more general statement about how the playback buffer evolves. Note that the only critical assumption we have made is that (2) holds. Our definition of bitrates r_t and throughput x_t ensures that (4) follows from (2).

In this section, we will point out some of the consequences of the Theorem for ABR algorithms.

A.1.1 Cannot exceed average throughput without buffer help. Intuition tells us average bitrate cannot exceed average throughput. Theorem A.1 gives us a simple formalization.

Say that the buffer does not decrease, so $B_0 \leq B_{T+1}$. Then

$$1 - \frac{B_{T+1} - B_0}{D_T} \leq 1.$$

By Theorem A.1, $\bar{r} \leq \bar{x}$. That is, the bitrate cannot exceed average throughput.

However if we reduce the size of the buffer, we can exceed average throughput. Suppose $B_0 \geq B_{T+1}$. In this case,

$$1 - \frac{B_{T+1} - B_0}{D_T} \geq 1.$$

By Theorem A.1, $\bar{r} \geq \bar{x}$.

A.1.2 Building up a buffer comes at the expense of bitrate. Suppose we have built up a 5 minute buffer by the time we select the last chunk ($B_0 = 0$, $B_{T+1} = 300$), then rearranging Theorem A.1 gives:

$$\bar{r} = \bar{x}\left(1 - \frac{5}{D_T}\right).$$

Over a twenty minute session, this says that $\bar{r} = 0.75x$. Restating, if an ABR algorithm builds up a 5 minute buffer over a 20 minute session then it will get a bitrate which is 75% of average throughput.

A.1.3 Intermediate buffer values do not affect average bitrate. All the terms in Theorem A.1 are averages, the difference between ending and starting buffer, and the duration. From the perspective of the average bitrate we can achieve, it doesn't matter if the throughput is stable or wildly variable. The path of the buffer is also not important—the only terms that affect bitrate are the starting and ending buffer sizes. We can build up a large intermediate buffer by picking a lower bitrate than throughput, and then decrease the buffer to regain bitrate.

As an example, suppose we start with no buffer and build up a thirty second buffer during the first sixty seconds of playback. By Theorem A.1, over the first sixty seconds $\bar{r} = 0.5\bar{x}$. Suppose we make careful choices over the rest of the session, and keep the buffer at thirty seconds after twenty minutes of playback. By Theorem A.1, $\bar{r} = 0.975\bar{x}$. By controlling the size of the buffer over the course of the session, we don't suffer for our early choice to build up a large buffer. This effect is what allows buffer-based algorithms to achieve high bitrates.