Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE

Junchen Jiang
Carnegie Mellon University
junchenj@cs.cmu.edu

Vyas Sekar
Stony Brook University
vyas@cs.stonybrook.edu

Hui Zhang
Carnegie Mellon University/Conviva Inc.
hzhang@cs.cmu.edu

ABSTRACT

Many commercial video players rely on bitrate adaptation logic to adapt the bitrate in response to changing network conditions. Past measurement studies have identified issues with today's commercial players with respect to three key metrics—efficiency, fairness, and stability—when multiple bitrate-adaptive players share a bottleneck link. Unfortunately, our current understanding of why these effects occur and how they can be mitigated is quite limited.

In this paper, we present a principled understanding of bitrate adaptation and analyze several commercial players through the lens of an abstract player model. Through this framework, we identify the root causes of several undesirable interactions that arise as a consequence of overlaying the video bitrate adaptation over HTTP. Building on these insights, we develop a suite of techniques that can systematically guide the tradeoffs between stability, fairness and efficiency and thus lead to a general framework for robust video adaptation. We pick one concrete instance from this design space and show that it significantly outperforms today's commercial players on all three key metrics across a range of experimental scenarios.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed systems—*Distributed applications* ; C.4 [**Performance of Systems**]: [measurement techniques]

General Terms

Design, Performance, Measurement

Keywords

Video, HTTP, DASH, Adaptation

1. INTRODUCTION

Video traffic is becoming the dominant share of Internet traffic today [5]. This growth in video is accompanied, and in large part driven, by a key technology trend: the shift from customized connection-oriented video transport protocols (e.g., RTMP [10]) to HTTP-based adaptive streaming protocols (e.g., [1, 11, 13, 41]). With an HTTP-based adaptive streaming protocol, a video player can dynamically (at the granularity of seconds) adjust the video bitrate based on the available network bandwidth. As video traffic is expected to dominate Internet traffic [5], the design of robust adaptive HTTP streaming algorithms is important not only for the performance of video applications, but also the performance of the Internet as a whole. Drawing an analogy to the early days of the Internet, a robust TCP was critical to prevent "congestion collapse" [29]; we are potentially at a similar juncture today with respect to HTTP streaming protocols.

Building on this high-level analogy, it is evident that the design of a robust adaptive video algorithm must look beyond a single-player view to account for the interactions across *multiple* adaptive streaming players [14, 15, 22] that compete at bottleneck links. In this respect, there are three (potentially conflicting) goals that a robust adaptive video algorithm must strive to achieve:

- *Fairness:* Multiple competing players sharing a bottleneck link should be able to converge to an equitable allocation of the network resources.
- *Efficiency:* A group of players must choose the highest feasible set of bitrates to maximize the user experience.
- *Stability:* A player should avoid needless bitrate switches as this can adversely affect the user experience.

Recent measurements show that two widely used commercial solutions fail to achieve one or more of these properties when two players compete at a bottleneck link [14, 27]. We extend these experiments (Section 2) and confirm that the problems manifest across many state-of-art HTTP adaptive streaming protocols: Smooth-Streaming [12], Netflix [8], Adobe OSMF [2], and Akamai HD [3]. Furthermore, these problems worsen as the number of competing players increases.

While such measurements are valuable in identifying the shortcomings of today's players, our understanding of the root causes of these problems is limited. To this end, we systematically study these problems through the lens of an abstract video player that needs to implement three key components (see Section 2): (1) scheduling a specific video "chunk" to be downloaded, (2) selecting the bitrate for each chunk, and (3) estimating the bandwidth. At a high-level, the aforementioned problems arise as a result of *overlaying* the adaptation logic on top of several layers that may hide the true network state. Consequently, the feedback signal that the player receives from the network is not a true reflection of the network state. Furthermore, this feedback can also be biased by the decisions the player makes as well. Specifically, we observe that periodic chunk scheduling used in conjunction with stateless bitrate selection used by players today can lead to undesirable feedback loops with bandwidth estimation and cause unnecessary bitrate switches and unfairness in the choice of bitrates.

We leverage measurement-driven insights to design robust mechanisms for the three player components to overcome these biases. Our specific recommendations are (Section 3): (1) *randomized chunk scheduling* to avoid synchronization biases in sampling the network state, (2) a *stateful bitrate selection* that compensates for the biased interaction between bitrate and estimated bandwidth, (3) a *delayed update* approach to tradeoff stability and efficiency, and (4) a bandwidth estimator that uses the *harmonic mean* of download speed over recent chunks to be robust to outliers. Taken together, these approaches define a family of adaptation algorithms that vary in the tradeoff across fairness, efficiency, and stability. For example, we can consider player designs that choose the randomized scheduling with the stateful bitrate selection, without implementing the delayed update or the new bandwidth estimator.

As a concrete instance, we also show how to pick a sweet spot in this tradeoff space called the FESTIVE algorithm.[1] We implement FESTIVE using the Open Source Media Framework and show that our proposed logic is easy to implement and incurs low overhead. We evaluate FESTIVE against several real and emulated commercial players across a range of scenarios that vary the overall bandwidth and number of users. Compared to the closest alternative, FESTIVE improves fairness by 40%, stability by 50% and efficiency by at least 10%. Furthermore, FESTIVE is robust to the number of players sharing a bottleneck, increase in bandwidth variability, and the available set of bitrates.

In summary, this paper makes the following contributions:

- We systematically explore the design space of adaptive video algorithms with the goals of fairness, stability, and efficiency.
- We identify the main factors in bitrate selection and chunk scheduling employed in state-of-art players today that lead to undesirable feedback loops and instability.
- We design robust mechanisms for chunk scheduling, bandwidth estimation, and bitrate selection that inform the design of a suite of adaptation algorithms that vary in the tradeoff between stability, fairness and efficiency.
- We identify one concrete design from this family of algorithms as a reasonable point in this tradeoff space and show that it outperforms state-of-art players under most of the considered scenarios.

2. BACKGROUND AND MOTIVATION

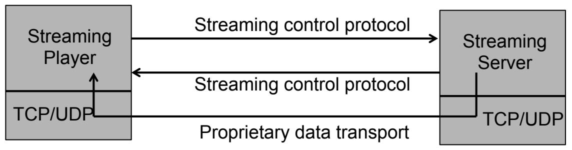
We begin with a high-level overview of how HTTP-based adaptive video streaming works and point out key differences w.r.t TCP-level control logic. Then, we formally define metrics to capture the three key requirements of fairness, efficiency, and stability, and evaluate how well today's state-of-art video players perform.

2.1 HTTP Adaptive Video Streaming

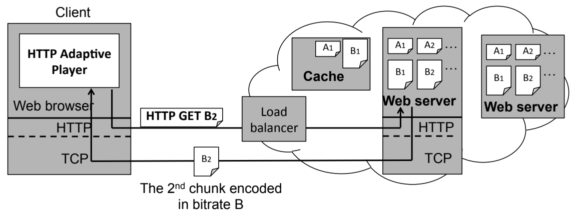
Early Internet video technologies (e.g., Apple QuickTime [4], Adobe Flash RTMP [10]) were based on connection-oriented video transport protocols. As shown in Figure 1(a), these protocols have a session abstraction between the client and the server, that both maintain per-session state and use a (proprietary) stateful control protocol to manage the data delivery. The new generation of Internet video technologies such as Microsoft SmoothStreaming [12], Apple's HLS [41], and Adobe's HDS [1], however, are HTTP-based adaptive streaming protocols.

In HTTP adaptive streaming, a video is encoded at multiple *discrete* bitrates. Each bitrate stream is broken into multiple 2-10 seconds segments or "chunks". The i^{th} chunk from one bitrate stream is aligned in the video time line to the i^{th} chunk from another bitrate stream so that a video player can smoothly switch to a different bitrate at each chunk boundary. As shown in Figure 1(b), HTTP-based adaptive streaming protocols differ from the traditional connection-oriented video transport protocols in several important aspects. First, clients use the standard HTTP protocol which provides more ubiquitous reach as this traffic can traverse NATs and firewalls [42]. Second, the servers are commodity Web servers or caches; this use of existing CDN and server technology has been a key driver for rapid growth and low costs. Third, the use of HTTP implies caches deployed by enterprise and service providers automatically improve the performance and reduce network load. Finally, a client fetches each chunk *independently* and maintains the playback session state while servers do not need to keep any state. This makes it possible for the client to receive chunks from multiple servers: enabling load-balancing and fault tolerance on both CDN side (across multiple servers) and client side (across multiple CDNs) [35, 36].

[1]The name FESTIVE refers to a **F**air, **E**fficient, and **S**table adap**TIVE** algorithm.


Streaming Player
TCP/UDP
Streaming control protocol
Streaming control protocol
Proprietary data transport
Streaming Server
TCP/UDP

(a) Connection-oriented streaming



(b) HTTP adaptive streaming

Figure 1: Difference between connection-oriented and HTTP adaptive streaming protocol.

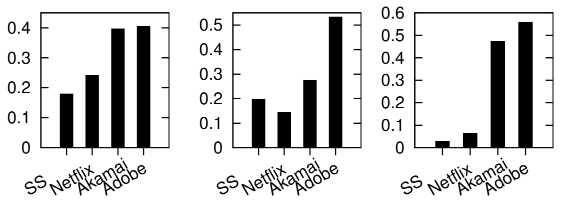
The client-side video player usually implements the adaptive logic in a constrained sandbox environment such as Flash or Silverlight. The adaptive part arises because the player uses the throughput observed for each chunk and the chunk size to estimate the available network bandwidth. These estimates are used to choose a suitable bitrate for the next chunk to be downloaded. The player tries to maintain an adequate video playback buffer to minimize rebuffering which can adversely impact user engagement [21].

2.2 Desired properties

We are specifically interested in a multi-player setting when multiple video players share a bottleneck link [14, 15, 27]. To formally define the metrics, we consider a setting with N players sharing a bottleneck link with bandwidth W, with each player x playing bitrate $b_{x,t}$ at time t.

- *Inefficiency:* The inefficiency at time t is $\frac{|\sum_x b_{x,t} - W|}{W}$. A value close to zero implies that the players in aggregate are using as high an average bitrate as possible to improve user experience [21].
- *Unfairness:* Now, some players could see a low bitrate while other players may see high quality. Akhshabi et al., use the difference between bitrates in a two-player setting to compute the unfairness [14]. We generalize this to multiple players as $\sqrt{1 - JainFair}$, where $JainFair$ is the Jain fairness index [43] of $b_{x,t}$ over all player x, because we want to quantify unfairness. A lower value of the metric implies a more fair allocation.

$J(b_{x,t}) = \text{JainFair}$



(a) Unfairness (b) Inefficiency (c) Instability

Figure 2: Performance of today's commercial players (SS stands for SmoothStreaming).

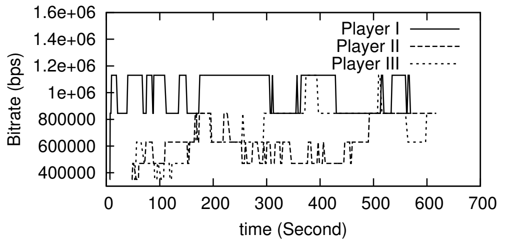


Figure 3: Visualizing unfairness and instability in SmoothStreaming

- *Instability:* Studies suggest users are likely to be sensitive to frequent and significant bitrate switches [18, 39]. We define the instability metric as $\frac{\sum_{d=0}^{k-1} |b_{x,t-d} - b_{x,t-d-1}| \cdot w(d)}{\sum_{d=1}^{k} b_{x,t-d} \cdot w(d)}$, which is the weighted sum of all switch steps observed within the last $k = 20$ seconds divided by the weighted sum of bitrates in the last $k = 20$ seconds. We use the weight function $w(d) = k - d$ to add linear penalty to more recent bitrate switch.

At first glance, these requirements for video adaptation seem analogous to traditional TCP. There are, however, key architectural differences between HTTP video adaptive streaming and TCP. First, the two control algorithms operate at different levels in the protocol stack. For example, video players can only access coarse information as they run in an application-level sandbox. Second, TCP is a connection-oriented protocol with control logic implemented at the sender-side while video adaptation is a connectionless protocol with receiver-side control. Third, the granularity of data and time are very different. TCP operates at the packet level ($\sim$1KB), has multiple packets in transit, and the control loop acts on the timescale of milliseconds. Video adaptation operates at the chunk level ($\sim$ hundreds of kilobytes), has only one chunk in transit, and the control loop runs at the timescale of seconds (i.e., chunk fetch delay). Last, due to the video-specific requirement that the buffer cannot be empty, the control actions are very different; a TCP sender delays packet transmission under congestion whereas the receiver in a video adaptation algorithm requests a lower bitrate chunk. Taken together, these factors mean that the rich literature and experience in designing TCP is not directly applicable here.

2.3 Performance of today's solutions

Given these formal metrics, we analyze the performance of existing commercial solutions—SmoothStreaming [11], Akamai HD [3], Netflix [8], and Adobe OSMF [2]. In doing so, we generalize the measurements from previous work that study 1 or 2 of these players in isolation and demonstrate that these problems are more widespread.

We consider a setup with three players sharing a bottleneck link with a stable bandwidth of 3 Mbps with default player settings. Each player runs in a separate Windows machine running on a

# players	BW (bps)	Unfairness	Inefficiency	Instability
5	5M	0.140	0.184	0.0537
11	11M	0.180	0.230	0.0648
19	19M	0.235	0.343	0.0909

Table 1: The performance of SmoothStreaming worsens as the number of players increases. We see similar trends with other players too (not shown for brevity).

2.8 Ghz desktop and accesses the respective demo website. Figure 2 shows the unfairness, inefficiency, and instability. We see that the Akamai and Adobe players are very unstable, while all of them are quite unfair. To give some context for what this unfairness index means, Figure 3 shows a time series of the bitrates of the three SmoothStreaming players which visually confirms that the allocation is quite unfair even for the best player in the above result. (In this case, the optimal allocation would be for all players to pick the same bitrate at all times.) Furthermore, Table 1 shows that the problems become worse as the number of players competing for the bottleneck link increases. Here, with a N player setup, we assume a stable bottleneck of $N \times 1$ Mbps. For brevity, we only show the result SmoothStreaming because this was the best overall player across all three metrics in our earlier experiment.

2.4 Design Space

Next, we describe the broader design space of how we can potentially address the above problems along three key dimensions:

- *What level in the protocol stack?*
  Video players can only access coarse information as they run in an application-level sandbox. To address this concern, we can re-architect the transport layer for video players; e.g., a new TCP variant or running it atop UDP and avoiding unnecessary interactions with the lower layer control loop. Alternatively, we can consider joint design of the video and transport layers. While this might be a better clean-slate solution, it does face deployment concerns—it may not be possible to upgrade the users' OS and new transport mechanisms may not work with middleboxes such as NATs and firewalls.
- *Where in the network?*
  There are three natural options: client-side, server-side, and in-network. For instance, TCP relies largely on sender-side control while video adaptation is a connectionless protocol with receiver-side control. Server-side solutions increase the requirements of CDN deployments beyond today's commodity web server designs. In-network solutions such as fair queuing or rate limiting in routers may offer more optimal solutions, but require a significant overhaul of the network infrastructure. Ultimately, the receiver is in the best situation to infer network bottlenecks and also take into account other user-side considerations (e.g., CPU load, bandwidth quotas) and thus we believe this is a pragmatic choice going forward.
- *Coordinated vs. Decentralized?*
  At one extreme, we can envision a logically centralized video controller that can have a global view of network conditions to assign bitrates to each user [36]. At the other extreme, we have purely decentralized solutions where the adaptation is purely receiver-driven. While this controller may be effective for coarse-grained decisions (e.g., choosing the CDN and bitrate at the start of a session), there are obvious challenges with respect to scale, visibility into client-side effects, and responsiveness for realizing such a controller in practice.

In this paper, we focus on a specific point in this design space—application-layer, receiver-driven, and decentralized adaptation. We cannot claim that this is the only or the ideal point in this space.

Rather, we pick this point as a pragmatic choice with a view toward a solution that is immediately deployable and that is backwards-compatible with today's video delivery ecosystem.

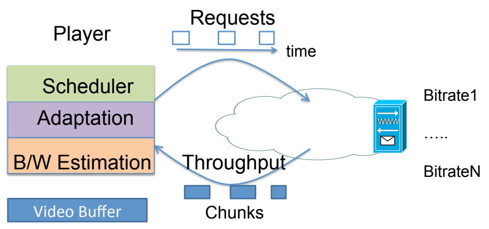


Figure 4: General framework of HTTP adaptive video streaming. The server supports multiple bitrate encodings, each a separate logically chunked file. The player issues GET requests for each chunk at a specific bitrate and adapts the bitrate based on the observed throughput.

3. DESIGN

As the previous section showed, today's state-of-art players do not satisfy the goals of fairness, efficiency, and stability. In this section, we describe how we design a adaptive streaming player that satisfies these properties. As the high-level model from Figure 4 shows, an adaptive streaming player involves three components:

1. Schedule when the next chunk will be downloaded.
2. Select a suitable bitrate for the next chunk.
3. Estimate the network bandwidth.

In designing each component, we make a conscious decision to be compatible with today's deployments and end-host stacks and do not require modifications to end-hosts' operating system stacks or CDN servers. For each component, we use measurement-driven insights to analyze problems with today's players to arrive at a suitable design. We validate each component in Section 6.2 and their interaction in Section 6.3.

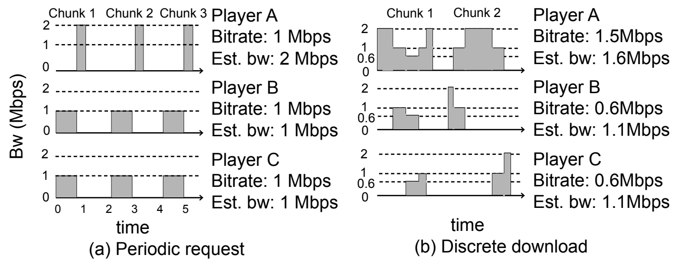


Figure 5: Two sources of bias with today's players: periodic request intervals lead to start-time biases and higher bitrates can cause players to estimate higher bandwidths.

3.1 Chunk Scheduling

The feedback that a player gets from the network is the observed throughput for each chunk. However, the discrete nature of the chunk download implies that the throughput a player observes is coupled to the time when the player "occupies" the link. This is in contrast to a long-running TCP flow that will observe its true share. Thus, we need a careful chunk scheduling approach to avoid biases in observing the network state.

We begin by considering two strawman options: (1) download the next chunk *immediately* after the previous chunk has been downloaded and (2) download chunks *periodically* so that the player buffer is sufficiently full. For example, SmoothStreaming uses the periodic strategy [14]. However, there are subtle issues with both approaches that we highlight next.

Immediate download: This greedily builds up the player buffer to avoid future buffering events. This approach, however, can be suboptimal for the following reasons. First, greedily downloading at the highest bitrate may needlessly increase the server's bandwidth costs, especially if users leave prematurely [23]. Second, greedily downloading low bitrate chunks may preclude the option of switching to a higher quality in case the network conditions improve. Furthermore, in the case of live content, future chunks may not even be available and thus this is not a viable option. While this greedy download option might be useful in the initial ramp-up phase for a player, the above reasons make it unsuitable in the steady state.

Periodic download: The periodic request strategy tries to maintain a *constant* playback buffer to minimize rebuffering [14]. This target buffer size is usually a fixed number of chunks; e.g., SmoothStreaming uses a 2-second chunk and a target playback buffer of 30 seconds (i.e., 15 chunks) [15]. This approach works as follows. Let t_i^{start} be the time when the i^{th} chunk is requested, t_i^{end} be the time that it is downloaded, and Δ denote the length of each chunk (in seconds). Suppose $buffer_i$ is length of the playback buffer (in seconds) at t_i^{end} and $targetbuf$ is the target buffer size (e.g., 30s). Then, the time to request the next chunk t_{i+1}^{start} can be written as:[2]

$$t_{i+1}^{start} = \begin{cases} t_i^{end}, \text{if } buffer_i < targetbuf \\ t_i^{end} + buffer_i - targetbuf, \text{otherwise.} \end{cases} \quad (1)$$

While this avoids wasting network bandwidth and prematurely committing to low quality, it suffers a different issue – players may see a biased view of the network state. Specifically, with the periodic download, the players' initial conditions may cause it to get stuck in suboptimal allocations. Figure 5(a) illustrates this problem. Suppose the players use a fixed request period of 2 seconds and the total bandwidth is 2 Mbps. Players A and B always request the next chunk at even seconds (i.e., 0,2,4,...), while player C requests it at odd seconds (i.e., 1,3,5,...). The throughput observed by A and B will be 1 Mbps (half the bandwidth) whereas C estimates it to be 2 Mbps (whole bandwidth). In other words, the initial conditions can lead to unfairness in bandwidth allocation.

Randomized scheduling: In order to avoid this bias induced by the initial conditions, we introduce a *randomized* scheduler that extends the periodic strategy. As before, we want to maintain a reasonable playback buffer. Instead of requiring a constant $targetbuf$, however, we treat it as an *expected* value. Specifically, for each chunk i we choose a target buffer size $randbuf_i$ uniformly at random from the range $(targetbuf - \delta, targetbuf + \delta]$. Specifically, we choose $\delta = \Delta$ which is driven by the analysis from Section 4. Then, the time to request the next chunk is:

$$t_{i+1}^{start} = \begin{cases} t_i^{end}, \text{if } buffer_i < randbuf_i \\ t_i^{end} + buffer_i - randbuf_i, \text{otherwise.} \end{cases} \quad (2)$$

At steady state, the chunks will be downloaded roughly periodically, but with some jitter as we randomize the target buffer size. We show via analysis in Section 4 and measurements in Section 6, that this strategy ensures that the time to request each chunk, and consequently the estimated bandwidth, is independent of the time at which a player arrives.

$\longrightarrow$ not biased by start time

[2]We can prove that this downloads one chunk every Δ seconds at steady state; we do not show this for brevity.

3.2 Bitrate Selection

Having chosen a chunk scheduling strategy that ensures that each player is not biased by its start time, we move to bitrate selection. Our high-level goal here is to ensure that the players will eventually *converge* to a fair allocation irrespective of their current bitrates.

Bias with stateless selection: A natural strategy is to choose the highest available bitrate lower than the estimated bandwidth. We refer to this as *stateless* approaches as it only considers the estimated bandwidth without considering the current bitrate or whether it is ramping up or ramping down its bitrate. For example, if the available bitrates are 400, 600, and 800 Kbps and the estimated bandwidth is 750 Kbps, the player chooses 600 Kbps.

While this stateless approach seems appealing, it can result in an unfair allocation of a bottleneck link. To understand why this happens, let us look at an example in Figure 5(b) with three players A, B and C sharing a bottleneck link with an available bandwidth of 2Mbps, using the randomized scheduler. There are three bitrates available: 600, 1200, and 1500Kbps. Suppose Player A is currently using a bitrate of 1500 Kbps and Player B and C are currently using bitrate 600 Kbps. As shown in Figure 5(b), because Player A uses a higher bitrate, its “wire occupancy” is higher than Player B and C. This implies that there are points in time where Player A is occupying the bottleneck link alone and thus Player A’s estimated bandwidth will be higher than Player B and C. In other words, the process of discretely downloading individual chunks naturally introduces a bias: *players currently using a higher bitrate observe a higher bandwidth*. We formally derive the relationship between estimated bandwidth and bitrates in Section 4.

Round	Bitrates (Kbps)	$\rightarrow$	Estimated bw. (Kbps) (network feedbacks)
1	[350,350,1520]	$\rightarrow$	[730,730,1356]
2	[470,470,1130]	$\rightarrow$	[717,717,1146]
3	[470,470,1130]	$\rightarrow$	[717,717,1146]
...	...	...	...

Table 2: Example of unfairness with stateless bitrate selection. The bitrate levels are {350,470,730,845,1130,1520}Kbps and the total bandwidth is 2Mbps.

Because there are only a discrete set of available bitrates (e.g., 4-5 encodings), players sharing a bottleneck link can often converge to an equilibrium state that is inherently unfair; e.g., in Figure 5(b), Player B and C will never increase their bitrate. This scenario is not merely hypothetical. For example, Table 2 shows an actual run using our setup (described in detail in Section 6), where the players converge to an equilibrium state that is inherently unfair.

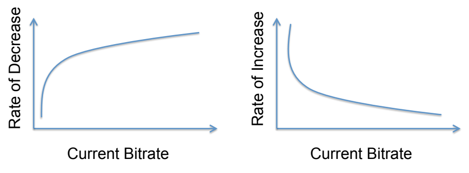


Figure 6: Intuition behind stateful selection: we want players with lower bitrate to ramp up aggressively or players with higher bitrate to ramp down aggressively.

Our approach: At a high-level, we need to compensate for the above bias so that the players can converge to a fair allocation irrespective of their current bitrates. We can achieve this in one of two ways as shown in Figure 6: (1) the rate of decrease is a monotonically increasing function of the bitrate or (2) the rate of increase is a monotonically decreasing function of the bitrate. Intuitively, we are making the player *stateful* by accounting for its current bitrate.[3] Our current design chooses option (2) and we simply keep the rate of decrease a constant function. In the example in Table 2, this approach causes the players starting at 350 Kbps to ramp up their bitrates more aggressively so that they will observe the true network state after 2-3 switches.

This stateful strategy can be realized either by allowing *multi-level* bitrate switches (e.g., from 350 to 1130 and skipping intermediate levels) or by altering the *rate* of switching the bitrates (e.g., once per chunk at 350 but once every 5 chunks at 1130). While we do not conclusively know if users are more sensitive to multi-level switches or the number of switches [18], recent work suggests that changing quality levels gradually is preferable [39]. Thus, we choose a *gradual* switching strategy where the player only switches to the next highest level and uses a lower rate of upward switches at higher bitrates. We discuss our specific approach in Section 3.5. We do note that the property achieved by a stateful approach is agnostic to how specific players implement the mechanism from Figure 6.

3.3 Delayed Update

While the previous discussion provides guidelines for choosing the bitrate to converge to a fair allocation, it does not consider the issue of *stability*—switching too frequently is likely to annoy users (e.g., [18]) and thus in this section, we focus on balancing these two potentially conflicting goals: efficiency and fairness on one hand vs. stability on the other.

To this end, we introduce a notion of *delayed update*. We treat the bitrate from the previous section only as a *reference* bitrate and defer the actual switch based on a measured tradeoff between efficiency/fairness and stability. Specifically, we compute how close to the efficient or stable allocation the current (b_{cur}) and the reference bitrate computed from the previous discussion (b_{ref}) are.

The *efficiency cost* for bitrate b is:

$$score_{efficiency}(b) = \left| \frac{b}{\min(w, b_{ref})} - 1 \right|$$

Here, w is the estimated bandwidth and b_{ref} is the reference bitrate from the previous section. Intuitively, the score is the best and equal to zero when $b = b_{ref}$. (The “min” in the denominator corrects for the fact that the reference bitrate may be underutilizing or overutilizing the bottleneck link.)

The *stability cost* for a given bitrate b is a function of the number of bitrate switches the player has undergone recently. Let n denote the number of bitrate switches in the last $k = 20$ seconds. Then the stability metric is,

$$score_{stability}(b) = \begin{cases} 2^n + 1 & \text{if } b = b_{ref} \\ 2^n & \text{if } b = b_{cur} \end{cases}$$

The reason to model the stability score using an exponential function of n is that $score_{stability}(b_{ref}) - score_{stability}(b_{cur})$ is monotonically increasing with n, which adds more penalty of adding a new switch if there have already been many switches in recent history.[4]

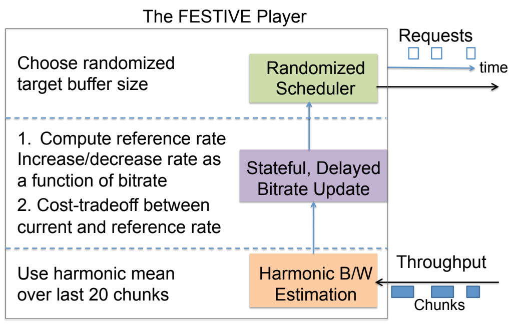
The combined score is simply the weighted average:

$$score_{stability}(b) + \alpha \times score_{efficiency}(b)$$

The player computes this combined score for both the current and reference bitrates, and picks the bitrate with the lower combined score. The factor α here provides a tunable knob to control

[3]We can show that this approach is sufficient; we do not claim that this is necessary.

[4]It is not necessary to model the stability score via an exponential function; this scheme has proven sufficient in our experiments.



The FESTIVE Player
Choose randomized target buffer size
Randomized Scheduler
Requests
time
1. Compute reference rate Increase/decrease rate as a function of bitrate
2. Cost-tradeoff between current and reference rate
Stateful, Delayed Bitrate Update
Use harmonic mean over last 20 chunks
Harmonic B/W Estimation
Throughput
Chunks

Figure 7: Overview of the FESTIVE adaptive video player.

the tradeoff between efficiency and fairness on one hand and stability on the other. We provide empirical guidelines on selecting a suitable value for α in Section 6.2.

3.4 Bandwidth Estimation

As we saw in the previous discussion, the throughput observed by a player for each chunk is not a reliable estimate of the available capacity. We suggest two guidelines to build a more robust bandwidth estimator. First, instead of using the instantaneous throughput, we use a *smoothed* value computed over the last several chunks. In our current prototype, we use the last 20 samples.[5] Second, we want this smoothing to be *robust* to outliers. For example, using the arithmetic *mean* is biased by outliers if one chunk sees a very high or low throughput. To this end, we use the *harmonic mean* over the last 20 samples. The reason for using this approach is two-fold. First, the harmonic mean is more appropriate when we want to compute the average of rates which is the case with throughput estimation. Second, it is also more robust to larger outliers [7]. This is especially relevant in the context of our randomized scheduler. With a randomized scheduler, if there are fewer competitors for a certain chunk, the estimated throughput will be larger. In such cases, the harmonic mean minimizes the impact of outliers. (If there are more competitors, then each player is more likely to observe a bandwidth close to its fair share.)

3.5 The FESTIVE algorithm

We now proceed to put the different design components together to describe the FESTIVE (Fair, Efficient, Stable, adaptIVE) algorithm. Figure 7 shows a high-level overview of FESTIVE. FESTIVE retains the same external-facing interface as today's HTTP video streaming players. That is, FESTIVE selects the bitrate for each chunk and decides when to schedule the request and the input to FESTIVE is the throughput observed per-chunk.

In describing FESTIVE, we focus on the steady-state behavior. The ramp up behavior of FESTIVE can be identical to today's players; e.g., aggressively download chunks (potentially at a low bitrate) to start playing the video as soon as possible. As discussed in the previous sections, FESTIVE has three key components:

1. The *harmonic bandwidth estimator* computes the harmonic mean of the last $k = 20$ throughput estimates. This provides *reliable* bandwidth estimates on which future bitrate update decisions can be made. In the initial phase before we have a sufficient number of samples, FESTIVE does not employ any rate switches because its bandwidth estimate will be unreliable.
2. The *stateful and delayed bitrate update* module receives throughput estimates from the bandwidth estimator and computes a reference bitrate. As a specific implementation of Figure 6, we use a gradual switching strategy; i.e., each switch is only to the next higher/lower level. Here, we increase the reference bitrate at bitrate level k only after k chunks, but decrease the bitrate level after every chunk if a decrease is necessary. This ensures that the bitrates eventually *converge* to a fair allocation despite the biased bitrate-to-bandwidth relationship. To decide if we need to decrease, we compare the current bitrate with $p = 0.85\times$ the estimated bandwidth. The parameter p helps tolerate the buffer fluctuation caused by variability in chunk sizes [15]. For the delayed update, we use a value of the tradeoff factor $\alpha = 12$ (see Section 6.2).
3. The *randomized scheduler* works as shown in Eq (2). It schedules the next chunk to be downloaded immediately if its playback buffer is less than the target buffer size. Otherwise, the next chunk is scheduled with a random delay by selecting a randomized target buffer size. This ensures there are no *start time biases*.

[5]We do not claim this is new; commercial players likely already implement some smoothing.

4. ANALYSIS OF FESTIVE

In this section, we show that:

- The randomized scheduler ensures that the request time of a player is independent of its start time.
- The stateful bitrate selection ensures that bitrates will eventually converge to a fair allocation.

Together, these ensure that the network state observed by competing players will not be biased by their arrival time or by the initial bitrates of other players.

Notation: We use i, k to denote chunk indices, j for a specific epoch, and x, y, z to denote players. Let n be the number of players and m be the number of chunks and let the bottleneck bandwidth be W. We use Δ to denote the length (in time) of each chunk.

Model: Our focus here is on the steady state behavior and not the initial ramp up phase. To make the analysis tractable, we make four simplifying assumptions. First, we assume the bottleneck bandwidth is stable. Second, this bandwidth is not saturated by the summation of bitrate, and each player's bitrate is less than its allocated bandwidth. As a result, for each chunk, a player will complete the download before the deadline, so the $buffer_i > randbuf_i$ will hold for most chunks. Third, if n players are simultaneously downloading over a bottleneck of bandwidth W, we assume that each player will get a bandwidth share of $\frac{W}{n}$. Last, we consider an *epoch-based* model, where players synchronously choose a new bitrate at the start of each epoch and estimate the bandwidth at the end of each epoch.[6]

4.1 Randomized scheduler

The goal of the randomized scheduler is to ensure the request time is independent of a player's start time. Formally, we want to show that:

Theorem 1. *If a player uses $randbuf_i$ drawn uniformly at random between $(targetbuf - \Delta, targetbuf + \Delta]$ and $buffer_i > randbuf_i$ for each chunk i ($i = 1, \ldots, m$), then the probability distribution of chunk request times does not depend on the start time t_0^{start}.*

PROOF. The buffer length at time t_{i-1}^{end}, when chunk $i-1$ has been downloaded is $buffer_{i-1} = (i-1)\Delta - (t_{i-1}^{end} - t_0^{start})$ where $(i-1)\Delta$ is the length of content downloaded so far and $t_{i-1}^{end} - t_0^{start}$ is the amount of video played. If $buffer_{i-1} > randbuf_{i-1}$, then by Eq (2), the time to request the next chunk:

[6]Each epoch can consist of multiple chunks.

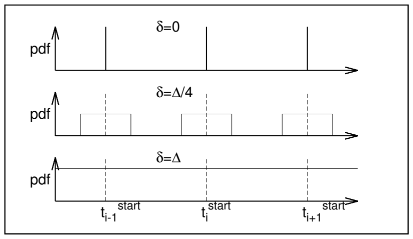


Figure 8: Intuition for Theorem 1.

$$t_i^{start} = t_{i-1}^{end} + \mathit{buffer}_{i-1} - \mathit{randbuf}_{i-1} =$$
$$t_{i-1}^{end} + (i-1)\Delta - (t_{i-1}^{end} - t_0^{start}) - \mathit{randbuf}_{i-1}$$
$$= t_0^{start} + (i-1)\Delta - \mathit{randbuf}_{i-1}$$

Because each $\mathit{randbuf}_{i-1}$ is a uniform random variable in the range $(\mathit{targetbuf} - \Delta, \mathit{targetbuf} + \Delta]$, this means that for a given i, t_i^{start} is a uniform random variable in the range $(t_0^{start} + (i-1)\Delta - \mathit{targetbuf} - \Delta, t_0^{start} + (i-1)\Delta - \mathit{targetbuf} + \Delta]$. Let T denote a random variable representing the request time. Then $T = t$ can occur for exactly two intervals i^* and $i^* + 1$, with $i^* = \frac{1}{\Delta}(t + \mathit{targetbuf} - t_0^{start})$ as shown in Figure 8. Thus, $f(T = t) = f(t_{i^*}^{start} = t \text{ or } t_{i^*+1}^{start} = t) = 2 * \frac{1}{2\Delta} = \frac{1}{\Delta}$ which is independent of t_0^{start}. $\square$

Notice that for other $\delta \neq \Delta$, if $\mathit{randbuf}_i$ is at random in range $(\mathit{targetbuf} - \delta, \mathit{targetbuf} + \delta]$, then the same argument of Theorem 1 does not hold. For example, if $\delta = \frac{1}{r}\Delta$ where $r > 2$, then the ranges of t_i^{start} for different i will not overlap (see Figure 8). Consequently, for any t, there will be at most one t_i^{start} whose range covers t. That is, $f(T = t)$ will be $\frac{1}{2\delta}$ for exactly one k such that $t_0^{start} + k\Delta - \mathit{targetbuf} - \delta < t \leq t_0^{start} + k\Delta - \mathit{targetbuf} + \delta$ and 0 otherwise. In other words, the request time distribution depends on the start time t_0^{start}. The periodic scheduler is an extreme case with $r \to \infty$.

4.2 Stateful bitrate selection

We begin by deriving the relationship between estimated bandwidth and bitrate in Lemma 1 which shows that a player with higher bitrate will see relatively higher bandwidth.

Lemma 1. *For two players, x and y, let w_x and w_y be the harmonic mean of the throughput seen by them and b_x, b_y be their bitrates. Then, $\frac{w_x}{w_y} = \frac{b_x+W}{b_y+W}$.*

PROOF. Since we are using the randomized scheduler, each player will join the link randomly. Let n_{ix} be the number of competitors when player x downloads chunk i, then the bandwidth allocation of chunk i is $\frac{W}{n_{ix}+1}$. Thus, the download time for chunk i is $d_{ix} = \frac{b_x\Delta(n_{ix}+1)}{W}$ where $b_x\Delta$ is the chunk size. The total download time is $\sum_{i=1}^{m} d_{ix}$, and the fraction of time when player x is downloading is:

$$q_x = \frac{1}{m\Delta}\sum_{i=1}^{m} d_{ix} = \frac{1}{m\Delta}\sum_{i=1}^{m}\frac{b_x\Delta(n_{ix}+1)}{W}$$
$$= \frac{b_x}{W}\sum_{i=1}^{m}\frac{(1+n_{ix})}{m} = \frac{b_x}{W}N_x$$

where $N_x = 1 + E(n_{ix})$ is the expected number of competitors for x. When each chunk length is small, the probability that player i is competing for the bandwidth is simply the fraction of time spent downloading $= q_x$. Thus, we have $N_x = 1 + E(n_{ix}) = 1 + \sum_{z\neq x} q_z$.[7] Thus, we have

$$\frac{q_x W}{b_x} + q_x = 1 + \sum_z q_z = \frac{q_y W}{b_y} + q_y \Rightarrow \frac{b_x + W}{b_y + W} = \frac{\frac{b_x}{q_x}}{\frac{b_y}{q_y}}$$

Now, the harmonic mean of bandwidth is simply:

$$w_x = \frac{m}{\sum_{i=1}^{m}\frac{1}{w_{ix}}} = \frac{W}{\frac{1}{m}\sum_{i=1}^{m}(1+n_{ix})} = \frac{W}{N_x} = \frac{b_x}{q_x}$$

Thus, we have $\frac{b_x+W}{b_y+W} = \frac{w_x}{w_y}$ $\square$

Notice that w_x is a harmonic mean, rather than expectation, of the bandwidth the player sees, which is consistent with how bandwidth is estimated in FESTIVE.

Based on this, we have the following theorem which proves bitrate convergence. Recall from Section 3.5 that if bitrate $b_x > pw_x$ where w_x is the harmonic mean of bandwidth of the epoch and p is a real value parameter, then the player x will decrease bitrate in the next epoch (i.e., stateless decrease). Otherwise, it will increase in a rate which depends on the bitrate level (i.e., stateful increase).

Theorem 2. *Let l_x^j and l_y^j be the bitrate levels of players x and y in j^{th} epoch with $l_y^j - l_x^j \geq 2$. Then the gap will eventually converge to be at most one level, i.e., $\exists j' > j$, where $|l_x^{j'} - l_y^{j'}| \leq 1$.*

PROOF. Given $l_y^j - l_x^j \geq 2$, we show that $l_y^j - l_x^j$ monotonically decreases as a function of j until $|l_y^j - l_x^j| = 1$. Let b_x^j, b_y^j denote the bitrates and w_x^j, w_y^j be the bandwidth in epoch j. By Lemma 1, there is no p for which $pw_x < b_x^j$ and $b_y^j < pw_y$. (Otherwise, $\frac{w_x^j}{w_y^j} < \frac{b_x^j}{b_y^j} < \frac{b_x^j+W}{b_y^j+W}$, which contradicts Lemma 1.) Therefore, there are only three cases for the estimated bandwidths w_x^j, w_y^j, (i) $pw_x^j > b_x^j, pw_y^j < b_y^j$, (ii) $pw_x^j < b_x^j, pw_y^j < b_y^j$, and (iii) $pw_x^j > b_x^j, pw_y^j > b_y^j$. For (i), b_y^j will decrease, and b_x^j will not decrease, therefore, $l_y^{j+1} - l_x^{j+1} \leq l_y^j - l_x^j - 1$. For (iii), before switching to (i) or (ii), x will increase earlier than y according to the stateful bitrate update (Figure 6), so $l_y^j - l_x^j$ will decrease. For (ii), since the two players cannot always decrease bitrate in (ii), so eventually, they will enter (iii) or (i). As a result, in each epoch, $l_x^j - l_y^j$ cannot increase and it will not always remain constant. $\square$

5. OSMF-BASED IMPLEMENTATION

We have implemented our FESTIVE algorithm in an open source Flash video player. The implementation builds on the Open Source Media Framework (OSMF) codebase (v2.0) and is written in ActionScript [9]. OSMF is an open-source framework developed by Adobe which provides most of the basic functionalities of a commercial video player.

OSMF provides a well-defined API for different player functions. There are two specific "hooks" that we leverage to implement the FESTIVE logic: (1) the function `play2(streamName)` that can be used to specify a target bitrate (via its URL prefix `streamName`), and (2) the function `bufferTime(t)` that can be used to specify target buffer length of `t`. One challenge, however, is that the function `play2` as currently implemented may not always change bitrate for downloading the next chunk. Thus, we extended `play2` by adding check points at the boundary of each chunk. At these points, our additional code receives the throughput of last chunk, and decides the target buffer length and the bitrate of the next chunk. We implement the specific FESTIVE logic in a separate class and set the bitrate (via extended `play2`) and target buffer length (via `bufferTime`). The additional code is $\approx$ 500

[7]This is by linearity of expectation.

lines, a small fraction compared to the full OSMF codebase (125K lines of code). We have run several microbenchmarks over a range of bitrates and confirmed that our FESTIVE code adds little or no CPU overhead to the existing player logic for downloading, buffering, decoding, and rendering (not shown for brevity).

6. EVALUATION

We divide our evaluation into four high-level sections:

1. We compare the performance of FESTIVE against (emulated) commercial players (Section 6.1).
2. We validate each component—randomized chunk scheduling, stateful and delayed bitrate selection, and harmonic bandwidth estimation (Section 6.2).
3. We evaluate how critical each component is to the overall performance of FESTIVE (Section 6.3).
4. Finally, we evaluate the robustness of FESTIVE as a function of bandwidth variability, number of players, and the set of available bitrates (Section 6.4).

Evaluation setup: We use our OSMF-based implementation and real commercial players wherever possible. However, it is difficult to run controlled experiments with the real commercial players due to the lack of access to their code and the difficulty in automating experiments with multiple players on different machines. Our goal is to evaluate the underlying *adaptation logic* of different adaptive players. However, the proprietary nature of the client/server code for these players makes it difficult to do a head-to-head comparison. Specifically, using the commercial players conflates external effects: network (e.g., wide-area bottlenecks) and server-side (e.g., CDN load) effects, issues w.r.t video encoding/decoding, and player plug-in performance.

Thus, in order to do a fair comparison, we augment these real player experiments with a custom emulation framework. Here, we heuristically create *emulated* clones that closely mimic each commercial player. In each case, we verified over a range of settings that our emulated clone is a *conservative* approximation of the commercial player; i.e., the unfairness, inefficiency, and instability with the emulated clone are *lower bounds* for the actual player. Our heuristic approach works as follows. We start with a basic algorithm that uses the periodic scheduler and the harmonic bandwidth estimation algorithms. Based on trace-driven analysis, we observed that most commercial players appear to employ a *stateless* bitrate selection algorithm that can be modeled as a linear function of the throughput estimated for the previous chunk(s). We use linear regression to find the best fit for each commercial player separately. For example, the SmoothStreaming player appears to pick the highest available bitrate below $0.85\times$ the estimated bandwidth [8]. We do not claim that these are the exact algorithms; our goal is to use these as conservative approximations of the players to do a fair comparison with FESTIVE.

We implemented a flexible framework that allows us to evaluate different algorithms for chunk scheduling, bitrate selection, and bandwidth estimation. Our setup consists of client players, video servers, and a bottleneck link. Both client and server side mechanisms are implemented as Java modules (about 1000 lines each) that run on different machines within a local network. The client player decides the bitrate for the next chunk and when to issue the request. Once the video server receives the request which explicitly encodes the bitrate, it generates a file with size dependent on the bitrate. The client downloads this chunk over a regular TCP socket. All traffic between clients and servers goes through the bottleneck which uses Dummynet [46] to control the total bandwidth and delay. Unless specified otherwise, we emulate a ten-minute long video with eight bitrate levels from 350Kbps to 2750Kbps and using 2 second chunks. (This is based on the parameters we observe in the demo website [11]). We use chunk sizes of an encoded video for each bitrate by analyzing real traces of commercial players from [11].

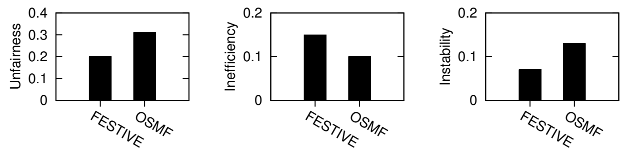


Figure 9: Comparison between real OSMF player and OSMF player using FESTIVE.

6.1 Comparison with Commercial Players

OSMF implementation: First, we use our real implementation atop the OSMF framework and compare it to the current OSMF adaptation logic using the video at [6]. Here, we consider a setup with three players that share a bottleneck link of 3 Mbps. Figure 9 shows the unfairness, inefficiency, and instability of FESTIVE vs. the OSMF logic. We observe that our implementation outperforms OSMF on two of three metrics. Our inefficiency, however, is slightly higher. We speculate that this is because our FESTIVE parameters are customized for the chunk sizes and bitrate levels seen in the SmoothStreaming demo as we discussed in Section 5.

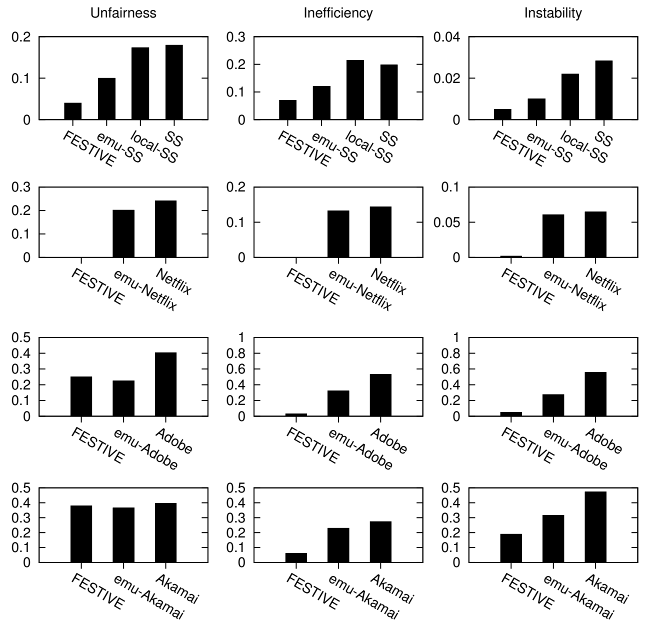


Figure 10: Comparison between FESTIVE, emulated commercial players, and the actual commercial players with 3 players sharing a bottleneck link of 3 Mbps. Here, SS stands for SmoothStreaming; "emu-X" stands for our conservative emulation of the "X" commercial player; and local-SS is running a local SmoothStreaming server.

Other players: As we discussed, the chunk lengths and bitrate levels vary across commercial players. Thus, in each result we use the corresponding bitrate levels and chunk lengths observed in the players' (demo) websites to compare FESTIVE, the emulated player, and real player. Our goal here is to compare FESTIVE to each commercial player independently; it is not meaning-

[8]Based on linear regression between selected bitrate and bandwidth estimated by update function $bw_{next} = 0.9bw_{prev} + 0.1bw_{cur}$

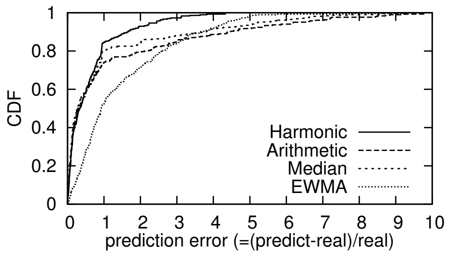


Figure 11: Prediction error in bandwidth estimation.

ful to draw any conclusions across players (e.g., is SmoothStreaming better than Netflix?) since the parameters such as chunk sizes and bitrate levels across the demo sites vary significantly. Figure 10 compares the performance of FESTIVE to the emulated commercial players, with the median value over 15 runs. In each case, a lower value of the performance metric is better. For reference, we also show the performance of the commercial players with an equivalent three player setup (using respective demo sites). For SmoothStreaming, we also have access to the server implementation. Thus, we also evaluate a local setup with the real players and server. For each commercial player, we confirm that the emulated version is a conservative approximation. We see that FESTIVE outperforms the next best solution, SmoothStreaming, by at least $2\times$ in all three metrics. We also observed that FESTIVE provides higher benefits as we increase the number of players (not shown). For example, our stability of 19 players (with 19Mbps bottleneck bandwidth) is still about $2\times$ higher than SmoothStreaming, but the gap between SmoothStreaming and us is $4\times$ larger than that of 3 players.

6.2 Component-wise Validation

Next, we examine whether each component achieves the properties outlined in Section 3. As a baseline point of reference, we use the emulated SmoothStreaming player and evaluate the effect of incrementally adding each component.

Bandwidth estimator: We begin by comparing the accuracy of four bandwidth estimation strategies: arithmetic mean, median, EWMA,[9] and harmonic mean. Each method computes the estimated bandwidth using the observed throughput of the $k = 20$ previous chunks. For this analysis, we extract the observed chunk throughputs from the real SmoothStreaming setup from Section 2 with 19 competing players and emulate each estimation algorithm. We report the CDF of the prediction error $\frac{|PredictedBW - ActualBW|}{ActualBW}$ in Figure 11. The result shows that the harmonic mean outperforms the other methods. (The large prediction errors in the tail appear because the observed bandwidth for each chunk depends on the number of competing players that chunk sees which is highly variable.) We also manually confirmed that the harmonic mean is effective when a new observed throughput is an outlier. Thus, for the rest of this section, we consider the baseline algorithm with a harmonic bandwidth estimator.

Chunk scheduling: Here, the baseline player uses stateless bitrate selection, instant update, harmonic bandwidth estimation, and the periodic chunk scheduling discussed in Section 3. We consider a modified baseline that uses the randomized scheduling instead but retains the other components. Figure 12 shows the perceived bandwidth for the three players over time for one run. (The results are consistent across runs, we do not show them for brevity.) We can visually confirm that the periodic scheduler leads to large bias in the estimated bandwidth, while the randomized scheduler ensures a more equitable bandwidth share. The result also shows the difference between maximum and minimum bitrate to confirm that this bias in observed bandwidth also translates into unfairness in bitrate selection.

[9]Using the update function $bw_{next} = 0.9bw_{prev} + 0.1bw_{cur}$

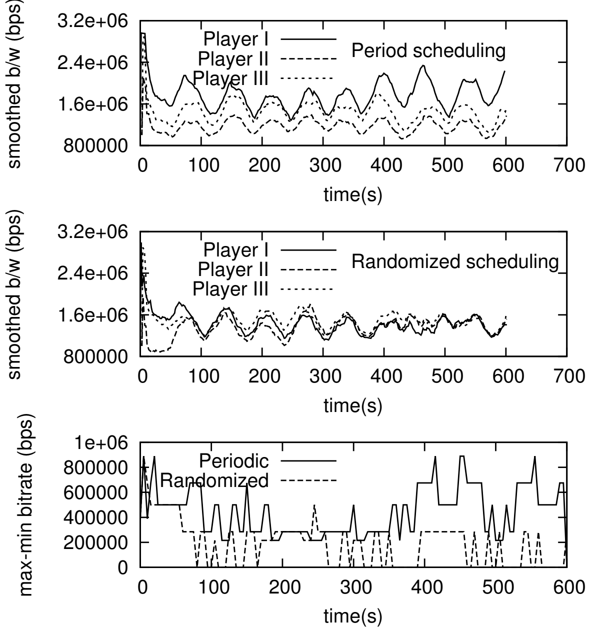


Figure 12: Randomized scheduling avoids start-time biases and ensures a fair allocation of bandwidth

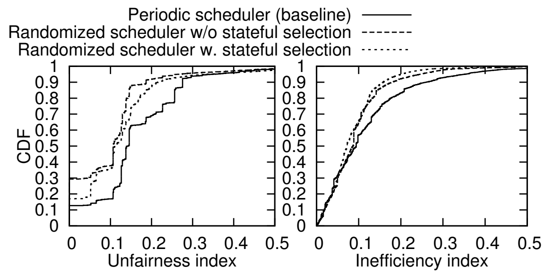


Figure 13: Stateful bitrate selection improves fairness with minimal impact on efficiency.

Stateful bitrate selection: The goal of the stateful bitrate selection approach is to ensure that different players will eventually converge to a fair allocation. To validate this, we consider ten players sharing a bottleneck link of 10 Mbps. Each player picks a start time uniformly at random in the interval of $[0, 30]$ seconds.

Figure 13 compares the efficiency and fairness achieved by three player settings: (1) fixed scheduler with stateless selection (baseline), (2) randomized scheduler with stateless selection, and (3) randomized scheduler with stateful selection. (We disable delayed update and use harmonic mean estimator for all three.) We see that stateful selection works well in conjunction with randomized scheduling and further improves the fairness. One concern with stateful bitrate selection is that players may increase/decrease bitrate synchronously and lead to over/under utilization (low efficiency). The result also shows that the efficiency is almost unaffected and may even be better than the stateless approach. The reason is that once the players converge to a fair allocation, all subsequent switches are only between two consecutive levels, which keeps the inefficiency small.

Delayed Update: The parameter α provides a way to tune the tradeoff between efficiency and stability. We examine this tradeoff with different number of players and bandwidth variability in Figure 14. (We discuss the exact variability model in Section 6.4).

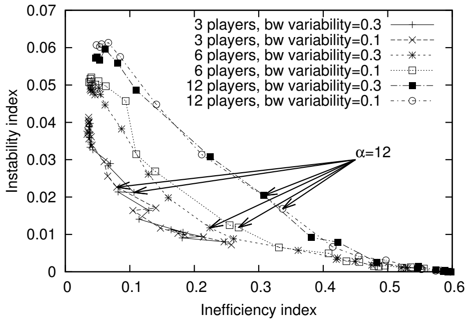


Figure 14: Tradeoff of delayed update between efficiency and stability: 'knee' points using $\alpha = 12$.

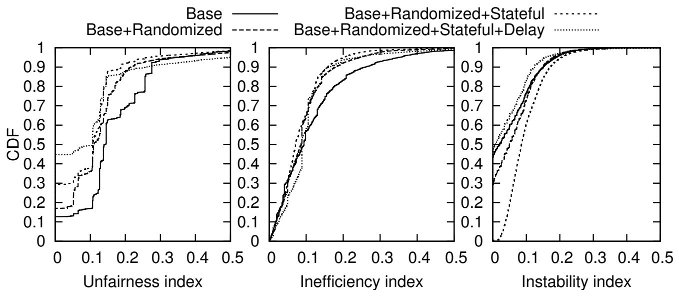


Figure 15: Break-down evaluation of FESTIVE.

From the bottom-right to top-left, α increases from 5 to 30; larger α provides higher efficiency at the cost of stability (Section 3). We suggest a guideline of picking the α that is close to the "knee" of the curve or the point closest to the origin. Across most scenarios, we find this roughly corresponds to $\alpha = 12$; we use this value for FESTIVE.

6.3 How critical is each component?

To see the effect of each component in FESTIVE, Figure 15 shows the effect of incrementally adding the randomized scheduler, stateful bitrate selection, delayed update to the baseline. For this result, we consider the scenario with 10 players competing for a 10 Mbps bottleneck link. First, we see that the randomized scheduler improves the fairness and efficiency over the baseline (by avoiding bias of starting time), and stateful bitrate selection further improves these (by avoiding bias of initial bitrates). However, these components are likely to increase the instability relative to the baseline. The delayed update then helps control this tradeoff between efficiency and stability; it reduces the efficiency slightly but improves stability significantly.

6.4 Robustness

Last, we investigate FESTIVE's performance in the presence of varying number of concurrent players, bandwidth variability and available bitrate sets.

Number of concurrent players: We fix the total bandwidth at 10Mbps and vary the number of concurrent players from 2 to 30. In each run, the players arrive randomly within the first 30 seconds after the first player starts. For each setting, we report the median and error bars over 15 runs for both baseline and FESTIVE in Figure 16. First, we see that FESTIVE outperforms the baseline across all settings and that the performance variability of FESTIVE is much smaller. Second, we see that unfairness and instability issues are lower when there are too few or too many players. In the former case, all player can sustain the highest bitrate and in the latter case the only feasible solution is for all players to choose the lowest bitrate (350 Kbps). Finally, we see an interesting effect where the metrics are not monotone in the number of players. Specifically, the case of 12 and 20 players are much better than their nearby points. This is essentially an effect of the discreteness of the bitrate levels. For example, when 12 players share a 10Mbps bottleneck, each player is very likely to stay at 845Kbps and saturate the link. However, at 10 players or 14 players, the player will try lower or higher bitrate because there is no optimal saturation bitrate.

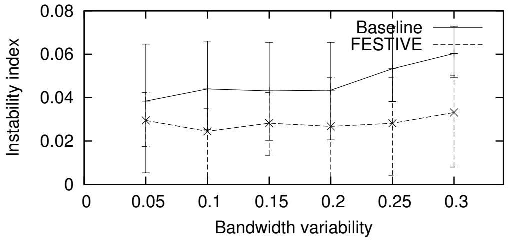


Figure 17: Instability vs. bandwidth variability when 10 players compete for a 10Mbps (expected) link

Bandwidth variability: We focus on the 10 player scenario with an *expected* bottleneck bandwidth of 10 Mbps. All players arrive within the first 30 seconds and we report the results from 15 runs per parameter. This bottleneck bandwidth is an expected value, because we vary the bandwidth every 20 second by picking a value uniformly at random $[BW \times (1-\epsilon), BW \times (1+\epsilon)]$. Figure 17 plots the performance of baseline and FESTIVE as a function of this parameter ϵ. We see that FESTIVE is more robust to the bandwidth variability (from $\epsilon = 0.05$ to $\epsilon = 0.3$) and in fact the improvement with FESTIVE increases with higher variability.

	Unfairness		Instability		Inefficiency	
g	Base.	FESTIVE	Base.	FESTIVE	Base.	FESTIVE
1.2	0.128	0.071	0.052	0.039	0.111	0.126
1.4	0.154	0.061	0.049	0.005	0.125	0.095
1.6	0.172	0.076	0.002	0.0	0.104	0.117
1.8	0.184	0.051	0.040	0.0	0.133	0.121

Table 3: Performance metrics vs. bitrate gaps when 10 players compete a bottleneck of 10Mbps

Available bitrates: Last, we test robustness to the set of available bitrate levels. We create a set of 10 available bitrate levels by $\{b_i = g^i \cdot 350Kbps\}_{i=0,\ldots 9}$, where g controls the gap between the bitrates, i.e., how discrete the bitrate levels are. A value of g close to 1 means that the gaps between consecutive levels are small and vice versa for larger g. Table 3 compares the performance of baseline and FESTIVE under g. FESTIVE consistently outperforms the baseline. The baseline becomes more unfair as g increases while FESTIVE works robust against higher g.

6.5 Summary of main results

In summary, our evaluation shows

- FESTIVE outperforms existing solutions in terms of fairness by $\geq$40%, stability by $\geq$50%, and efficiency by $\geq$10%.
- Each component of FESTIVE works as predicted by our analysis and is necessary as they complement each other.
- FESTIVE is robust against various number of players, bandwidth variability, and different available bitrate set.

7. DISCUSSION

We discuss three outstanding issues w.r.t the design of adaptive video players.

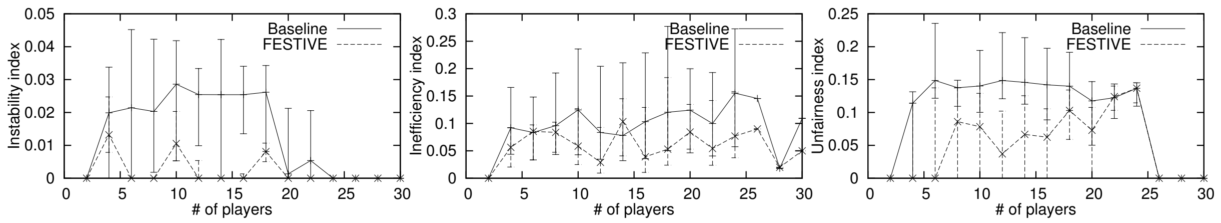


Figure 16: Performance of FESTIVE and the baseline player as a function of the number of concurrent players. Here, we assume the players are sharing a 10 Mbps bottleneck link.

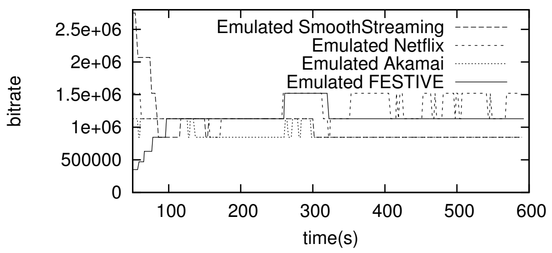


Figure 18: 8 emulated players including FESTIVE, SmoothStreaming, Netflix and Akamai HD with random start time share a bottleneck of 8Mbps, and 4 of them are presented. FESTIVE appears to be more stable and stays on an efficient bitrate.

Heterogeneous algorithms: With the diversity of video content providers, we expect that there will be heterogeneity in player designs. So far, we have considered a homogeneous settings where all players run FESTIVE or existing commercial algorithms. One natural concern here is the interaction between different players. As a preliminary result, we consider a mixed workload with 8 emulated players, 2 players each for FESTIVE, SmoothStreaming, Netflix and Akamai HD, sharing a 8Mbps bottleneck link. For a fair comparison, we use emulated players and let them use the same set of bitrates found on respective demo sites. Each player arrives at random in the first 20 seconds. We pick one player instance for each algorithm and show the time series of bitrate in Figure 18. We see that FESTIVE is more stable than the other players, and spends most of the time at an efficient bitrate (1130kbps). There are many other questions with heterogeneous players; e.g., Are there specific cases of unfairness or starvation when a particular combination of players compete? Can we incentivize players to be "good citizens" and avoid being greedy? Is there an analog to TCP-friendliness? Studying the interaction between multiple heterogeneous players is an interesting and rich direction of future work.

Interaction with non-video traffic: Another natural question is how video adaptation logic interacts with non-video traffic (e.g., short Web transfers) [32]. Because FESTIVE retains the single-connection HTTP-based interface, it retains TCP-level friendliness *per chunk*. We use an example scenario of Figure 19 to confirm the intuition. Here, 3 FESTIVE players share a bottleneck with available bandwidth of 3Mbps, and one short TCP session joins at the 30th second and two short TCP sessions join at the 150th second. The gray line gives the bitrate selected by one FESTIVE instance. The figure shows that when TCP sessions join, FESTIVE will still achieve a fair share of the bandwidth when downloading a chunk; FESTIVE trades efficiency for stability, so bitrate almost does not vary with the interference of short TCP sessions. At the same time, the TCP sessions are able to get a throughput that is not too much less than the fair share. This is only a preliminary result and we need to more systematically explore these effects.

Wide-area effects: Another interesting direction of future work is to see if and how the efficiency, fairness, and stability issues manifest in the wide area. For instance, wide area effects imply more background traffic, less synchronization but many more players, multiple bottlenecks, interaction with router buffer sizing, among a host of other factors.

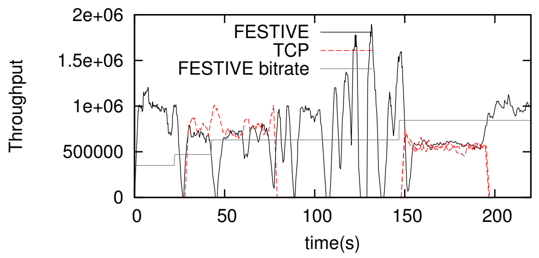


Figure 19: FESTIVE vs. short TCP sessions: short TCP sessions do not impact FESTIVE and vice versa.

8. RELATED WORK

Measurements of commercial players: Early studies focused on the bitrate switching behavior of a single player in response to bandwidth variation (e.g., [15, 19, 40, 44]). More recent work analyzes fairness, efficiency, and stability when two players share a bottleneck link [14, 27]). These have identified the periodic behavior as a potential problem similar to Section 3. Recent work has also identified some of the biased feedback loop effects we observe [28]. We confirm these problems on a broader set of commercial players and extend these beyond the two-player setting. More importantly, we provide a detailed understanding of the causes and present a concrete design to address these shortcomings.

Quality metrics: A key aspect in video delivery is the need to optimize user-perceived quality of experience. There is evidence that users are sensitive to frequent switches (e.g., [18]), sudden changes in bitrate (e.g., [39]), and buffering (e.g., [21]). The design of a good QoE metric (e.g., [47] [16]) is still an active area of research. As our understanding of video QoE matures, we can extend FESTIVE to be QoE-aware.

Player optimizations: The use of multiple connections or multipath solutions can improve throughput and reduce the bandwidth variability (e.g., [25, 26, 31, 33]). However, these require changes to the application stack and/or server-side support. Furthermore, they may not be friendly to background traffic. In contrast, FESTIVE retains the same single TCP connection interface and requires no modifications to the server infrastructure or the end-host stack. Other approaches use better bandwidth prediction and stability techniques (e.g., [34, 38, 40]). These proposals are largely complementary to the design of FESTIVE.

Server and network-level solutions: This includes the use of server-side bitrate switching (e.g., [32]), TCP changes to avoid bursts (e.g., [24]), and in-network bandwidth management and caching (e.g., [27, 40, 45]). Our focus is on client-side mechanisms without requiring changes to the network or servers. While these approaches will further improve the performance, we believe that a client-side solution is fundamentally necessary for two reasons. First, the client is in the best position to detect and respond to dynamics. Second, recent work suggests the need for cross-CDN optimizations that implies the need for keeping minimal state in the network or servers [35, 36].

Video Coding: Layered or multiple description coding offers more graceful degradation of video quality (e.g., [17]). However, they impose higher overhead on content providers and the delivery infrastructure and thus we do not consider this class of solutions.

9. CONCLUSIONS

With the growth of video traffic, we are revisiting classical networking problems w.r.t resource sharing and adaptation. These problems have a rich literature with solutions at the network (e.g., [20]), transport (e.g., [29]), and application layers (e.g., [17, 37]). However, there are several factors that make the problem unique and challenging in today's HTTP-based video delivery: the granularity of the control decisions, the timescales of adaptation, the nature of feedback from the network, and the interactions with other (indepedent) control loops in lower layers of the networking stack.

In this work, we have taken a pragmatic stance to work within the constraints that have spurred the growth of video traffic—using HTTP, no modifications to end-host stacks, and imposing no modification to the network and CDN server infrastructure. Within this context, we provide a principled understanding of problems that lead to inefficiency, unfairness, and instability when multiple players compete for a bottleneck link. Building on these insights, we provide guidelines on designing better scheduling and bitrate selection techniques to overcome these problems.

There are several open questions with respect to co-existence of video and non-video traffic, competition among heterogeneous players (e.g., FESTIVE vs. legacy players?), the interaction with management optimizations in other aspects of the video delivery system [36], and exploring "clean-slate" solutions that can redesign network and transport layers to support video traffic [30]. We hope that our work acts as a fillip to address these broad spectrum of issues as new standards for video transport emerge [13].

10. ACKNOWLEDGEMENTS

We are grateful to our shepherd Rittwik Jana and anonymous reviewers for their feedback. This research was supported in part by the National Science Foundation under awards CNS-1040757, CNS-1040800, and CNS-1040801.

11. REFERENCES

[1] Adobe http dynamic streaming. `www.adobe.com/products/hds-dynamic-streaming.html`.
[2] Adobe osmf player. `http://www.osmf.org`.
[3] Akamai hd adaptive streaming. `http://wwwns.akamai.com/hdnetwork/demo/index.html`.
[4] Apple quicktime. `www.apple.com/quicktime/download/`.
[5] Cisco forecast. `http://goo.gl/hHzW4`.
[6] Crystal-clear hd with adobe http dynamic streaming. `http://zeridemo-f.akamaihd.net/content/adobe/demo/1080p.f4m`.
[7] Harmonic mean. `http://en.wikipedia.org/wiki/Harmonic_mean`.
[8] Mail service costs Netflix 20 times more than streaming. `http://goo.gl/msuYK`.
[9] Osmf 2.0 release code. `http://sourceforge.net/projects/osmf.adobe/files/latest/download`.
[10] Real-time messaging protocol. `www.adobe.com/devnet/rtmp.html`.
[11] Smoothstreaming experience. `http://www.iis.net/media/experiencesmoothstreaming`.
[12] Smoothstreaming protocol. `http://go.microsoft.com/?linkid=9682896`.
[13] I. Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE Multimedia*, 2011.
[14] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen. What Happens when HTTP Adaptive Streaming Players Compete for Bandwidth? In *Proc. NOSSDAV*, 2012.
[15] S. Akhshabi, A. Begen, and C. Dovrolis. An Experimental Evaluation of Rate Adaptation Algorithms in Adaptive Streaming over HTTP. In *Proc. MMSys*, 2011.
[16] A. Balachandran, V. Sekar, A. Akella, S. Stoica, and H. Zhang. A quest for an internet video quality-of-experience metric. 2012.
[17] J. Byers, M. Luby, and M. Mitzenmacher. A digital fountain approach to asynchronous reliable multicast. *IEEE JSAC*, Oct. 2002.
[18] N. Cranley, P. Perry, and L. Murphy. User perception of adapting video quality. *International Journal of Human-Computer Studies*, 2006.
[19] L. De Cicco and S. Mascolo. An experimental investigation of the akamai adaptive video streaming. *HCI in Work and Learning, Life and Leisure*.
[20] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proc. SIGCOMM*, 1989.
[21] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. A. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proc. SIGCOMM*, 2011.
[22] J. Esteban, S. Benno, A. Beck, Y. Guo, V. Hilt, and I. Rimac. Interactions Between HTTP Adaptive Streaming and TCP. In *Proc. NOSSDAV*, 2012.
[23] A. Finamore, M. Mellia, M. Munafo, R. Torres, and S. G. Rao. Youtube everywhere: Impact of device and infrastructure synergies on user experience. In *Proc. IMC*, 2011.
[24] M. Ghobadi, Y. Cheng, A. Jain, and M. Mathis. Trickle: Rate Limiting YouTube Video Streaming. In *Proc. USENIX ATC*, 2012.
[25] S. Gouache, G. Bichot, A. Bsila, and C. Howson. Distributed and Adaptive HTTP Streaming. In *Proc. ICME*, 2011.
[26] D. Havey, R. Chertov, and K. Almeroth. Receiver driven rate adaptation for wireless multimedia applications. In *Proc. MMSys*, 2012.
[27] R. Houdaille and S. Gouache. Shaping http adaptive streams for a better user experience. In *Proc. MMSys*, 2012.
[28] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari. Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard. In *Proc. IMC*, 2012.
[29] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.
[30] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *Proc. CoNext*, 2009.
[31] R. Kuschnig, I. Kofler, and H. Hellwagner. Evaluation of http-based request-response streams for internet video streaming. *Multimedia Systems*, pages 245–256, 2011.
[32] L. De Cicco, S. Mascolo, and V. Palmisano. Feedback Control for Adaptive Live Video Streaming. In *Proc. of ACM Multimedia Systems Conference*, 2011.
[33] C. Liu, I. Bouazizi, and M. Gabbouj. Parallel Adaptive HTTP Media Streaming. In *Proc. ICCCN*, 2011.
[34] C. Liu, I. Bouazizi, and M. Gabbouj. Rate adaptation for adaptive http streaming. *Proc. ACM MMSys*, 2011.
[35] H. Liu, Y. Wang, Y. R. Yang, A. Tian, and H. Wang. Optimizing Cost and Performance for Content Multihoming. In *Proc. SIGCOMM*, 2012.
[36] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A Case for a Coordinated Internet Video Control Plane. In *SIGCOMM*, 2012.
[37] S. McCanne, M. Vetterli, and V. Jacobson. Low-complexity video coding for receiver-driven layered multicast. *IEEE JSAC*, Aug. 1997.
[38] K. Miller, E. Quacchio, G. Gennari, and A. Wolisz. Adaptation Algorithm for Adaptive Streaming over HTTP. In *Proc. Packet Video Workshop*, 2012.
[39] R. K. P. Mok, E. W. W. Chan, X. Luo, and R. K. C. Chang. Inferring the QoE of HTTP Video Streaming from User-Viewing Activities . In *SIGCOMM W-MUST*, 2011.
[40] R. K. P. Mok, X. Luo, E. W. W. Chan, and R. K. C. Chang. QDASH: A QoE-aware DASH system. In *Proc. MMSys*, 2012.
[41] R. Pantos. Http live streaming. 2011.
[42] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future internet. In *Proc. HotNets*, 2010.
[43] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer system. Technical Report, DEC, 1984.
[44] A. Rao, Y.-S. Lim, C. Barakat, A. Legout, D. Towsley, and W. Dabbous. Network Characteristics of Video Streaming Traffic. In *Proc. CoNext*, 2011.
[45] R. Rejaie and J. Kangasharju. Mocha: A quality adaptive multimedia proxy cache for internet streaming. In *Proc. NOSSDAV*, 2001.
[46] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.
[47] H. H. Song, Z. Ge, A. Mahimkar, J. Wang, J. Yates, Y. Zhang, A. Basso, and M. Chen. Q-score: Proactive Service Quality Assessment in a Large IPTV System. In *Proc. IMC*, 2011.