# *Presto:* Towards Fair and Efficient HTTP Adaptive Streaming from Multiple Servers

Shengkai Zhang[1], Bo Li[1], and Baochun Li[2]

[1]Department of Computer Science and Engineering, Hong Kong University of Science and Technology
[2]Department of Electrical and Computer Engineering, University of Toronto

***Abstract*—Though HTTP adaptive video streaming has been widely adopted in the industry, it has been shown that it suffers from lackluster performance with respect to a number of desirable properties: fairness, efficiency, and stability, especially when multiple players compete for a bottleneck link. Effective algorithms have been proposed for single-server HTTP adaptive streaming to mitigate these problems.**

**In this paper, we present an in-depth analysis on these desirable properties in the context of using HTTP adaptive streaming *from multiple servers*, which demonstrate that these properties will no longer hold with the existing algorithms. To address this challenge, we present *Presto*, a new protocol designed to improve the user experience by providing better fairness, efficiency and stability in the context of multi-server HTTP adaptive streaming. Our real-world experimental results indicate that *Presto* substantially outperforms existing protocols in the multiple server scenario.**

***Index Terms*—HTTP adaptive streaming, DASH, Fairness, Efficiency, Stability.**

## I. Introduction

Demand for Internet bandwidth has been increasingly dominated by video streaming traffic, fueled by a paradigm shift from traditional connection-oriented protocols (e.g., RTMP, RTSP) to stateless adaptive streaming protocols over HTTP (e.g., [1]). With HTTP adaptive streaming, a source video is encoded into *chunks*, each containing a few seconds of video, independent from one another. These chunks are then hosted on a typical web server, so that *players* on the client side are able to send download requests with a standard HTTP. A server stores multiple versions (with different resolutions) of the same video, each with their own bitrates, and a player is able to adapt to the varying availability of network bandwidth by selecting a particular bitrate when requesting chunks to download.

Unfortunately, it has been shown that the existing real-world systems of Internet video streaming providers using HTTP adaptive streaming suffered from lackluster performance when multiple players on the client side share the same bottleneck link. There are three desirable properties that existing systems failed to deliver: (1) *fairness*, in that multiple players should be able to fairly share the bandwidth at the same bottleneck link; (2) *efficiency*, in that a player should be able to choose the best possible bitrate to maximize user experience; and (3) *stability*, in that a player should maintain a stable bitrate to ensure a smooth playback and to minimize disruption. Effective algorithms have recently been proposed to mitigate these problems [2], [3] in the case of single-server HTTP adaptive streaming, where each player requests video chunks from a single web server.

Due to the insatiable appetite for bandwidth when playing high-definition video streams with high bitrates, it is naturally conceivable to extend single-server streaming to *multi-server streaming*, where a player requests video chunks from multiple web servers simultaneously. These web servers can be hosted by either traditional content distribution networks (CDNs), or multiple virtual machines in datacenters in the public cloud, such as Amazon EC2. Fig. 1 illustrates an example of such a multi-server streaming scenario, where player $A$ retrieves video streams simultaneously from three web servers over multiple TCP flows, and all three servers host the same video content from a particular content provider. Similar to the case of single-server streaming, multiple players may compete for bandwidth at a bottleneck link in the context of multi-server streaming. In this example, players $B$ and $C$ may share the same bottleneck link with $A$, even though each of them may receive its video stream from only one server.

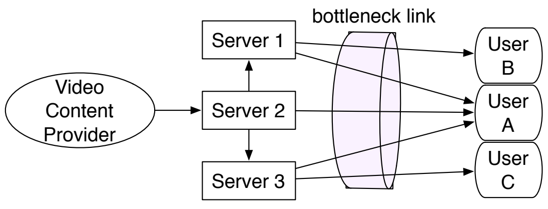


Fig. 1. HTTP adaptive streaming with multiple servers.

But how well do these properties — *fairness*, *efficiency*, and *stability* hold when existing algorithms are used in the context of multi-server HTTP adaptive streaming? To answer this question, we start with an in-depth analysis in this paper, and show that *these properties no longer hold*.

In this paper, we propose *Presto*[1], a new array of client-driven algorithms specifically designed to ensure *fairness*, *efficiency*, and *stability* in multi-server HTTP adaptive streaming. With *Presto*, the original contributions in this paper are two-fold. *First*, we design a new bitrate adaptation algorithm that achieves fairness *across players*, irrespective of the number of TCP flows each player establishes to multiple servers. *Second*,

[1]In music composition, *Presto* suggests that a musician plays the musical piece at a fast pace. The name is inspired by the acronym DASH, one of the HTTP adaptive streaming protocols that we use in our evaluation.

we propose a flow rescheduling algorithm to improve both efficiency and stability of multi-server HTTP adaptive streaming at the same time, by suspending and resuming TCP flows. Rather than using simulations, we evaluate *Presto* with its real-world implementation, which is built upon an open-source implementation of Dynamic Adaptive Streaming over HTTP (DASH) [4], one of the modern HTTP adaptive streaming protocols and an open MPEG standard, and demonstrate that *Presto* outperforms its closest alternative with respect to all the desirable properties — fairness, efficiency, and stability.

## II. Background

### A. HTTP adaptive streaming

HTTP adaptive bitrate streaming (HAS) is a technique used in streaming multimedia over computer networks. Early Internet video streaming technologies utilized connection-oriented streaming protocols (e.g., QuickTime, Adobe Flash). These protocols maintain transmission session states between client and server, and use stateful control protocol to deliver data. The new generation of video streaming technologies are designed to work efficiently over large distributed HTTP networks such as the Internet. Adaptive bitrate streaming provides users of streaming media with the best-possible experience by automatic adaptation of media server to any changes in each user's network conditions.

A web server encodes a single source video at multiple bitrates generating multiple resolution video copies. The streaming protocol detects available bandwidth in real-time and adjusts the quality of a video stream with respect to bitrates. The client is served by switching automatically among the different resolution copies. In detail, the protocol breaks each bitrate stream into multiple chunks with typical 2 – 15 seconds for each. Aligning the chunks from one bitrate stream to another in the video time line enables a smooth switch to a different bitrate at each chunk boundary. Since the rebuffering (video is frozen due to empty playback buffer) impairs user experience heavily [5], adaptive streaming player always tries to maintain an adequate video playback buffer while providing highest-possible streaming bitrates based on available bandwidth.

In a stable server-client connection, HAS does provide the best-possible user experience. However, researchers do not stop exploring more robust multimedia services to go beyond the single connection. Some solutions [6] exploit data centres and Content Delivery Networks (CDNs) for content distribution, streaming from multiple servers (multi-server HAS). Thus, when detecting a network congestion on a link that even HAS cannot save the streaming, it is possible to choose a candidate with better network condition to maintain user experience.

### B. Multi-server HTTP adaptive streaming

The primary advantage of multi-server HAS is to possibly rescue the multimedia streaming when the link is congested or broken, by resorting to alternative links. Basically, multi-server HAS has two modes: *mode-1*, dynamically switch the source from one server to another; *mode-2*, concurrent download contents from multiple servers. Apparently, there is always only one stream in *mode-1*. However, we are specially interested in *model-2* which is even better for maintaining streaming performance. Concurrent download prevents the estimation of bandwidth of links with other alternative sources for the switch so that it is more acute and straightforward to prevent the playback buffer from exhaustion by simultaneously fetching chunks from multiple sources.

*Mode-2* multi-server streaming only occurs in poor network conditions. In video streaming, ON-OFF cycle means the bit stream is interrupted due to the limited size of playback buffer. When the buffer is fully filled, video streaming protocol suspends the download until the player decodes at least one chunk so that the buffer has space. There are three video streaming strategies: no ON-OFF cycle, short ON-OFF cycle and long ON-OFF cycle. HAS protocols adopt the short ON-OFF cycle since it can minimize the downloaded but left unused bytes especially when many videos are quit early [2], [7].

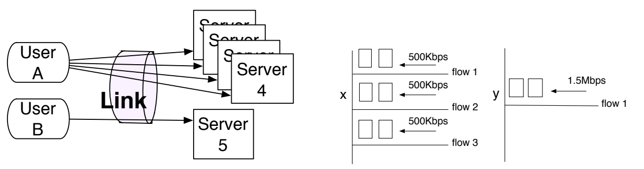
ON-OFF cycle state indicates that the available bandwidth of a link could support current bitrate. The player is periodically waiting a chunk is consumed by decoder. In this case, there is no *model-2* multi-server streaming. Despite we linked to multiple servers simultaneously, the ON-OFF cycle only allowed to download one chunk from one server at a time. However, when the network condition becomes very poor, or for high-definition (HD) video streams, we will have continuous chunk flows from multiple servers (*model-2*) for filling the playback buffer to be full as fast as possible. In the following text, we solve the problems arising from this scenario.

Recent works [2], [3] have identified three desired properties for HAS protocols: *fairness*, *efficiency* and *stability*. They also have proposed algorithms nicely holding these properties in the case of single-server HAS. We are specifically interested in a question: how well do the three properties hold in multi-server streaming? We next conduct an in-depth analysis and show that these properties no longer hold.

## III. Problem description

In this section, we examine the problems in the context of HTTP adaptive streaming *from multiple servers*.

### A. Fairness



(a) Unfairness (b) Inefficiency

Fig. 2. (a) The fairness problem in multi-server adaptive streaming. Player $A$ employs 4 TCP flows while player $B$ has only one flow. The allocated bandwidth of player $B$ will be only 20% of the bottleneck link. (b) The efficiency problem in multi-server streaming. Player $x$ streams from 3 video sources while player $y$ has only one source. They fairly share the 3Mbps bottleneck link.

The fairness problem only occurs when some users share a bottleneck link. Within the bottleneck link, we would like to know if each user could fairly be allocated available bandwidth. Although the state-of-the-art protocol (FESTIVE) [2] converges to a fair bitrate allocation in the context of single-server HTTP adaptive streaming, it does not work in the context of multi-server HTTP adaptive streaming as shown in Fig. 2(a). Player $A$ employs 4 TCP flows from different servers. Player $B$ has only one flow competing with $A$ at the bottleneck link. FESTIVE will allocate player $A$ $80\%$ of the bottleneck bandwidth, ensuring the fairness across different flows rather than players. Thus, the fairness property no longer holds.

## *B. Efficiency*

Efficiency requires that a player should be served with the best possible service, i.e., streaming the highest possible bitrate chunks that tries to saturate the available bandwidth. In Fig. 2(b), player $x$ and $y$ fairly share the 3Mbps bandwidth. Thus, the best possible bitrate should be close to $1.5$Mbps (available bandwidth) for $x$ and $y$. However, player $x$ was served very inefficient since it evenly assigned the bandwidth to three links of the simultaneous streaming, resulting $0.5$Mbps bitrate for each link. On the contrary, player $y$ with single-server enjoys streaming at the highest possible bitrate that fully utilizes the available bandwidth. A natural idea to solve this problem is that we simply either decrease the number of links or assign different bitrates to links. The former turns multi-server streaming into a single-server streaming, which eliminates the benefit of simultaneous streaming while the latter incurs stability problem that we elaborate in next subsection.

## *C. Stability*

Stability ensures a smooth playback and minimizes disruption. To cope with the change of network environment, a sensitive adaptation of bitrate will be more effective on efficiency and fairness. However, this will ruin the stability because of drastic changes on streaming bitrates. This is a tradeoff between fairness/efficiency and stability. In the context of single-server streaming, the delay-update strategy is proposed [2] for balancing the tradeoff.

The stability problem in multi-server streaming is more challenging since we have to additionally take care bitrates difference of concurrent links. Table I shows an instance that develops a big bitrate gap between two streams, causing video quality fluctuation. The setup simulates a real-world case that one link is heavily congested. Along with the concurrent download, the streaming bitrate in the congested link is suppressed while the other one keeps increasing because the bandwidth estimation signifies a higher bitrate for next chunk. At the 5th round, the gap reaches $500$ kbps and tends to expand further. This causes serious video quality fluctuation due to the resolution difference among chunks in playback buffer, which greatly impairs user engagement.

To solve the above problems, we propose a new protocol *Presto* that aims to achieve the fairness across players; streaming the best possible bitrate; guarantee a smooth playback. In next section, we will describe the protocol design in detail.

## IV. Protocol design

In this section, we present the design of *Presto*, our new suite of protocols addressing the problems discussed in the previous section. With an overview of multi-server HTTP adaptive streaming system, we describe the two components of *Presto*: (1) *Fair bitrate adaptation* that ensures the convergence of bitrate allocation among players; (2) *Flow rescheduling* that addresses the inefficiency and the instability. *Presto* is client-driven and is designed with the practicality of real-world implementations in mind, without the need to modify the operating system kernel.

| Round | Bitrates (kbps) | B/W estimation | Adaptation |
|---|---|---|---|
| 1 | 101.5<br>101.5 | 1085.5<br>229.7 | ↑<br>↑ |
| 2 | 201.4<br>201.4 | 1613.3<br>210.5 | ↑<br>– |
| 3 | 351.0<br>201.4 | 1254.6<br>227.0 | ↑<br>– |
| 4 | 501.1<br>201.4 | 1799.9<br>231.2 | ↑<br>– |
| 5 | 700.9<br>201.4 | 909.6<br>235.1 | ↑<br>– |

Note: "–" implies that the bitrate remains unchanged.

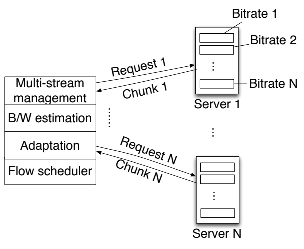
TABLE I
DEVELOPING A SUBSTANTIAL GAP OF BITRATES BETWEEN FLOWS: AN EXAMPLE.

## *A. System overview*

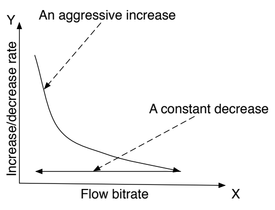
As Fig. 3(a) shows, a multi-server adaptive streaming player involves four components: (1) Multi-stream management coordinates the receiving order and the streaming chunk index for each flow; (2) B/W estimation estimates the available bandwidth; (3) Adaptation selects a suitable bitrate for next chunk; (4) Flow scheduler determines which flows should be suspended or resumed. The first two components are not the main contributions. We briefly introduce them next.

Multi-stream management is designed to solve three problems: how many links can be established, which chunk should be requested and how to guarantee the chunk order in playback buffer. The number of links is determined by the Media Presentation Description (MPD) file provided by servers. The player will contact a video streaming server. The server sends the MPD file first rather than video contents. By receiving and parsing the file, the player knows the URLs of available content resources. Apparently, requesting different chunks for each link is more efficient. We set a counter recording the latest *requested* chunk $ID$. The player will request chunk $ID + 1$ as long as a link is available. In addition, the player would receive out of order chunks by simultaneous download. To solve this problem, we set a swapping buffer with size $2 * N$, where $N$ denotes the number of links established. The swapping buffer will hand over chunks when they are in order. Otherwise, the buffer will keep storing the incoming chunks. It will signal multi-stream management to stop requesting chunks when the buffer stores over $N$ chunks for avoiding buffer overflow. We do not claim that $2 * N$ is optimal, but it is sufficient in our experiment.

B/W estimation takes advantage of the FESTIVE [2] method. We now walk into the key design of Presto: bitrate adaptation and flow rescheduling.



(a) System overview



(b) Algorithm intuition

Fig. 3. (a) is the framework of multi-server HTTP adaptive streaming. (b) is an intuition of our adaptation algorithm: an aggressive increase and a constant decrease.

## B. Bitrate adaptation

The key challenge is the convergence of a fair bitrate allocation among players. The different number of flows among users incurs the fairness problem. Players employing more flows are more powerful to seize bandwidth. Since the adaptation decision is mainly affected by the bandwidth estimation without considering its effect on other users' streaming quality, we believe that there is a bias like first-come first-served, i.e., *player using a higher aggregate bitrate of all flows observes a higher bandwidth*[2]. Here is an example of this bias. Suppose user $A$ has $m_A$ flows simultaneously streaming chunks. User $B$ has $m_B$ flows. $b_{Ai}$ ($b_{Bi}$) denotes the current bitrate of user $A$ ($B$)'s flow $i$. If $\sum_{i=1}^{m_A} b_{Ai} > \sum_{i=1}^{m_B} b_{Bi}$, user $A$ will always seize higher bandwidth, which is completely unfair.

To solve the problem, we propose an aggressive increase for the player employing fewer flows streaming at lower bitrates and throttle the increase rate for the player activates more flows streaming at higher bitrates. Considering user experience and stability issues, recent work [8] suggests that drastic bitrate increases do not annoy users, yet drastic bitrate decreases definitely disappoint them. A constant decrease rate is more reasonable. Therefore, our algorithm configures an aggressive increase and a constant decrease as shown in Fig. 3(b).

In HTTP adaptive streaming, the video content is encoded into several discrete bitrates. Suppose there are $L$ types of bitrate, $b_1 < b_2 < ... < b_L$, $b_i$ denotes the bitrate of type $i$, $i$ is the *type index*. In the following text, if the bitrate type index $i < j$, bitrate $b_i < b_j$. The adaptation approach decides the bitrate for next chunk. We model the aggressive increase by function $Y = \frac{\mathcal{K}_b}{mX}$, where $Y$ denotes the increase rate, $m$ denotes the number of flows for a player, $X$ is the current bitrate of a flow. $\mathcal{K}_b$ is a constant provided by the server, typically $\mathcal{K}_b = b_{\frac{L}{2}}$. We design a policy determining the increase rate as follows:

$$l_{\text{increase}} = \begin{cases} \frac{L}{2} & \text{if } \frac{\mathcal{K}_b}{mX} > \frac{L}{2} \\ 1 & \text{if } \frac{\mathcal{K}_b}{mX} < 1 \\ \frac{\mathcal{K}_b}{mX} & \text{otherwise} \end{cases} \tag{1}$$

This equation heavily penalizes player employing more flows, throttling the increase rate weighted with the number of flows $m_x$. For a player with few flows streaming low bitrates, $mX$ is small so that $Y$ can be very large. If $Y > \frac{L}{2}$, the streaming bitrate of a flow of the player will jump $\frac{L}{2}$ bitrate type indices, which is very aggressive. If the player has reached a high bitrate or employed many flows, $mX$ can be very large so that $Y$ is small. Thus, the increase rate is suppressed to 1. Regarding user experience [8], we adopt a gradual decrease strategy that the player only decreases the bitrate to the next lower bitrate type.

Besides the penalty of increase rate, we also penalize update frequency for players employing more flows. Refer to equation (1), $l_{\text{increase}} \geq 1$. If we allow a player with more flows to freely update the bitrate as long as a flow finishes transmitting a chunk, the player with fewer flows will never get a chance to reach a higher bitrate than median. To this end, we also determine the bitrate update frequency with respect to the number of a player's flows.

We maintain two vectors for a player with size $m$. One stores the current bitrate and the other stores the observed bandwidth. As long as a player completes a download, it adds the current bitrate and the bandwidth estimation into the vectors respectively. When the vectors are full, the stored data deduce the average bitrate $\bar{b} = \frac{1}{m}\sum_{i=1}^{m} q_{bi}$ and the average bandwidth estimation $\bar{\omega} = \frac{1}{m}\sum_{i=1}^{m} q_{\omega i}$, where $q_{bi}$ is the $i^{th}$ element in bitrate vector, $q_{\omega i}$ is the $i^{th}$ element in bandwidth estimation vector. Then we *empty* the two vectors and *choose* a flow to update its bitrate by the following rule:

$$l_{i+1} = \begin{cases} l_i + l_{\text{increase}} & \text{if } b_i < p \times \omega_i \text{ and } \bar{b}_i < p \times \bar{\omega}_i \\ l_i - 1 & \text{if } b_i \geq p \times \omega_i \text{ and } \bar{b}_i \geq p \times \bar{\omega}_i \\ l_i & \text{otherwise} \end{cases} \tag{2}$$

where $l_i$ denotes the bitrate type index for the $i^{th}$ chunk. The constant $p(= 0.85)$ provides a tolerance of buffer fluctuation due to the diversity of chunk sizes.

This strategy delays the update because it is triggered only when the two vectors are full. The player with more links takes longer time for filling the two vectors whose size is the number of links. Concerning the stability issue, we simply choose the link with the lowest bitrate for increase and the one with the highest bitrate for decrease (randomly choose one link if multiple links are ready to update with the same bitrate). After achieving fairness[2], we move forward to solve the rest of problems: inefficiency and instability.

## C. Flow rescheduling

We design a flow rescheduling algorithm that *suspends* and *resumes* flows to achieve efficiency and stability. First, we need to figure out when the inefficiency and the instability happen. If we find some events indicating the problems, we may design an algorithm to mitigate them. From our implementation and the experiment (describe in detail in Sec. V), we find the *local optimum* – the highest streaming bitrate cannot ramp up for a certain time – will cause inefficiency and the *large streaming bitrate gap* among flows will cause instability. Hence, we design the flow rescheduling algorithm as follows:

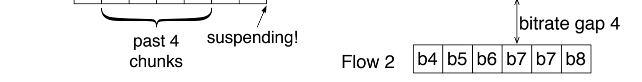
[2]Due to space constraints, we provide a technique report for the proofs including the bias and the correctness of our bitrate fair adaptation algorithm. https://www.cse.ust.hk/~szhangaj/techniqueReport/techniqueReportPresto.pdf

**Algorithm 1** Flow Rescheduling Algorithm

```
1:  For a player, at time $t$
2:  for $i = 1, ..., m$ do
3:      Collect recent $k$ samples of bitrate type index;
4:      if $\mathcal{G}_i \leq 1$ and $\mathcal{M}_i < L$ then
5:          $Stable_i =$ TRUE
6:      else
7:          $Stable_i =$ FALSE
8:      end if
9:  end for
10: if All $Stable_i ==$ TRUE, $i = 1, ..., m$ then
11:     Suspend the flow currently transmitting with the lowest bitrate
12: else
13:     Gap = ComputeGap($m$)
14:     if Gap $\geq \mathcal{H}$ then
15:         Suspend the flow currently transmitting with the lowest bitrate
16:     end if
17: end if
```

For a player with $m$ flows, each flow collects recent $k$ samples of the bitrate *type index* (line 3), where $k$ (typically equals 4) is a constant representing the sensitivity of our mechanism. $\mathcal{G}_i$ denote the maximum gap of the bitrate type index among the $k$ samples of flow $i$. $\mathcal{M}_i$ denote the maximum bitrate type index in the samples. $L$ is the highest bitrate type index provided by server. The condition of line 4 indicates flow $i$ is stable in the past $k$ chunks. When all flows are stable, the player suspends the flow currently transmitting the lowest bitrate (line 10-11). Fig. 4(a) shows an example of the stable state.

If the multi-server streaming state is not local optimum, the algorithm further investigates the large gap of streaming bitrates. Line 13 calculates the largest gap of bitrate type index among flows. When the gap reaches an empirical threshold $\mathcal{H}(=\frac{L}{2})$, the player suspends the flow currently transmitting the lowest bitrate. Fig. 4(b) shows an example with $L = 8$.



(a) Stable state (b) Large bitrate gap

Fig. 4. (a) illustrates the stable state suspension, where "!" denotes the stable signal and "X" denotes the cancellation of a signal. (b) is an instance of large bitrate gap triggering the suspension.

*Resume* is a complementary design to cope with the changing Internet environment. It resumes flows if necessary. The changing Internet environment may cause a *remarkable quality degradation*. We record two variables to identify such degradation: the highest bitrate type index before suspending any flow, $\mathcal{S}_s$ and the reached highest bitrate type index eventually, $\mathcal{S}_e$. Let $L_{\text{current}}$ denote the current maximum bitrate type index among active flows. Presto will resume all flows if $\mathcal{S}_e < L$ and $L_{\text{current}} < \frac{\mathcal{S}_s+\mathcal{S}_e}{2}$.

## V. Evaluation

In this section, we first describe the implementation and experiment setup. After defining the metrics for evaluating the fairness, efficiency and stability, we show the performance of *Presto*.

**System implementation**: Our system consists of three components: the player at client-side (Libdash [4]), the configurable link simulator (Dummynet [9]), and the distributed dataset (Distributed DASH [10]). The program runs on Visual Studio 2010.

**Experiment setup**: Players will download the media presentation description (MPD) file from Distributed DASH. Meanwhile, Dummynet is running for configuring the link capacity. We tweak the bottleneck link bandwidth as 6Mbps. Three participators (with 1, 3, 5 flow(s) respectively) compete the bottleneck link. The dataset targets 6 different sites at different locations. The servers provide 17 bitrate types ranging from 101Kbps to 5.94Mbps. The duration of each chunk is 2 seconds. The video consists of 2622 chunks in total. All contents in servers are synchronized.

**Evaluation metrics**: We define $1 - \text{Index}_{\text{Jain}}$ as a unfairness index to quantitatively evaluate fairness property, where $\text{Index}_{\text{Jain}}$ is Jain fairness index [11]. A lower value of the metric implies better fairness.

We define a new inefficiency metric: $\frac{|\sum_x \max_i\{b_{xi}\} - \sum_x M_{bx}|}{\sum_x M_{bx}}$, where $b_{xi}$ denotes the bitrate of flow $i$ of player $x$, $M_{bx}$ denotes the highest bitrate of chunks provided by server for player $x$. This metric highlights the maximum transmitting bitrate among flows of a player, which better reflects the user experience. A value close to zero implies that players on average choose to stream the highest possible bitrate chunks.

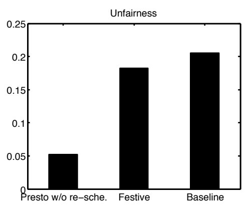
The instability metric is formally defined as $\frac{\sum_{d=0}^{m-1} \omega(d) \times \max\{0, b_{k-d-1} - b_{k-d}\}}{\sum_{d=1}^{m} \omega(d) b_{k-d}}$, where $b_{k-d}$ is the bitrate of $(k-d)^{th}$ chunk. It is a weighted sum of all the bitrate changes, observed within the last $m$ chunks, divided by the weighted sum of bitrates up to the last $m$ (typically $m = 10$) chunks. The weight $\omega(d) = m - d$ is linear penalty to more recent bitrate changes. $\max\{0, b_{k-d-1} - b_{k-d}\}$ indicates that we only take bitrate decreases into consideration, $\max\{0, b_{k-d-1} - b_{k-d}\} = 0$ when the player increases the bitrate. A larger value for this metric implies a larger quality fluctuation of video streaming.

We choose the state-of-the-art algorithm – FESTIVE [2] – as a benchmark to examine *Presto*. FESTIVE also concludes that most commercial players appear to employ a *stateless* bitrate adaptation method (Baseline). This method simply chooses the maximal bitrate type less than the estimated bandwidth without remembering any previous bitrate information.

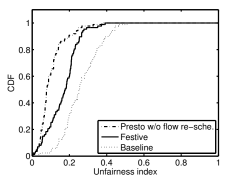
### A. Bitrate adaptation performance

To validate the fairness, we consider three players with 1, 3, and 5 flows sharing a bottleneck link of 6Mbps. We calculate the unfairness index in the time interval of 1 second during the video streaming for *Presto without flow rescheduling*[3], FESTIVE and Baseline. The arithmetic mean of the statistical data is outputted.

Fig. 5(a) demonstrates our algorithms converges to a fair bitrate allocation that the arithmetic mean of our unfairness index is only $25\%$ of the value from Festive. In Fig. 5(b), the unfairness index is lower than $0.15$ in $90\%$ of time while the others show a very serious unfairness problem that only $30\%$ of time, the unfairness index of Festive is lower than $0.15$. The baseline algorithm is worse.



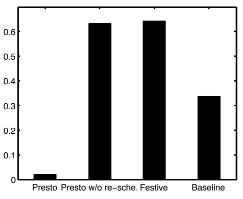
(a) Unfairness index



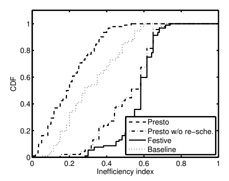
(b) CDF of unfairness

Fig. 5. Comparison between *Presto without flow rescheduling*, Festive, and stateless bitrate selection (Baseline).

## *B. Flow rescheduling performance*



(a) Inefficiency index



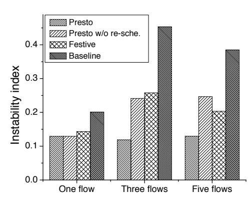
(b) CDF of inefficiency

Fig. 6. Comparison between *Presto*, Presto without flow rescheduling, Festive, and stateless bitrate selection (Baseline).

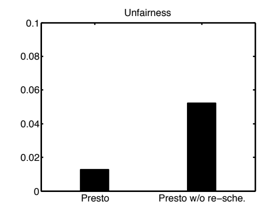
Fig. 6(a) shows that our bitrate adaptation algorithm (Presto without flow rescheduling) and Festive are highly inefficient. The players stream very low quality video chunks wasting their large bandwidth. However, the inefficiency index of Presto is close to zero, which implies the players stream the highest possible bitrate of chunks in average. Note that the baseline algorithm performs better than Festive and Presto without flow rescheduling. This is because the baseline has no rate adaptation. It trades the efficiency with the stability. Fig. 6(b) shows that over $80\%$ of the video streaming period, the inefficiency index is lower than $0.3$. It implies players stream bitrates higher than the median level. The slope of Presto CDF curve illustrates that the flow rescheduling needs some time to detect the stable state.

Fig. 7(a) illustrates the effectiveness of our large gap identification. When a player employs only one flow, Presto has a similar performance to Festive. In multi-server streaming, Presto keeps the instability index close to the single-server streaming. The instability indexes of Presto are $0.1291$, $0.1188$, and $0.1294$.

[3]This is for evaluating the bitrate adaptation algorithm without the effect of flow rescheduling.



(a) Instability index



(b) Unfairness index

Fig. 7. (a) is a comparison between *Presto*, Presto without flow rescheduling, Festive, and stateless bitrate selection (Baseline) for players employing different amounts of flows. (b) illustrates that the flow rescheduling has no adversely effect to the fairness property.

Finally, we would like to know whether the flow rescheduling adversely affects the fairness property. Fig. 7(b) shows that the rescheduling slightly improves the fairness because the algorithm suspends some flows, reducing the complexity of bitrate adaptation, leading a fine-grained fair allocation.

## VI. Conclusion

In this paper, we present an in-depth study on HTTP adaptive video streaming *from multiple servers*, which is conceived to be a natural extension from single-server HTTP streaming. We closely examine the key desirable properties including *fairness, efficiency*, and *stability*, and show that the existing algorithms fail to achieve any of such properties in the context of multi-server HTTP streaming. We then propose *Presto*, an array of client-side algorithms that not only provide high quality adaptive video streaming, but also hold the three desired properties.

The bitrate adaptation algorithm in *Presto* achieves the fairness by compensating the bias that player using a higher aggregate bitrate observes a higher bandwidth. Then we design a flow scheduling algorithm to overcome the inefficiency and instability problems and validate the effectiveness of Presto through real-world experiments.

## References

[1] I. Sodagar, "The MPEG-DASH standard for multimedia streaming over the internet," *MultiMedia, IEEE*, vol. 18, no. 4, pp. 62–67, 2011.

[2] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with festive," in *Proc. CoNext*, 2012.

[3] G. Tian and Y. Liu, "Towards agile and smooth video adaptation in dynamic HTTP streaming," in *Proc. CoNext*, 2012.

[4] "Libdash 2.1," http://www-itec.uni-klu.ac.at/dash/?p=1143.

[5] F. Dobrian, A. Awan, D. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang, "Understanding the impact of video quality on user engagement," in *Proc. SIGCOMM*, 2011.

[6] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang, "Unreeling netflix: Understanding and improving multi-cdn movie delivery," in *Proc. INFOCOM*, 2012.

[7] A. Finamore, M. Mellia, M. M. Munafò, R. Torres, and S. G. Rao, "Youtube everywhere: Impact of device and infrastructure synergies on user experience," in *Proc. IMC*, 2011.

[8] R. K. Mok, E. W. Chan, X. Luo, and R. K. Chang, "Inferring the QoE of HTTP video streaming from user-viewing activities," in *SIGCOMM W-MUST*, 2011.

[9] "Dummynet," http://info.iet.unipi.it/~luigi/dummynet/.

[10] S. Lederer, C. Mueller, C. Timmerer, C. Concolato, J. Le Feuvre, and K. Fliegel, "Distributed dash dataset," in *Proc. MMSys*, 2013.

[11] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Eastern Research Laboratory, Digital Equipment Corporation, 1984.