Towards Agile and Smooth Video Adaptation in Dynamic HTTP Streaming

Guibin Tian and Yong Liu
Department of Electrical and Computer Engineering
Polytechnic Institute of New York University
Brooklyn, NY, USA 11201
gtian01@students.poly.edu, yongliu@poly.edu

ABSTRACT

Dynamic Adaptive Streaming over HTTP (DASH) is widely deployed on the Internet for live and on-demand video streaming services. Video adaptation algorithms in existing DASH systems are either too sluggish to respond to congestion level shifts or too sensitive to short-term network bandwidth variations. Both degrade user video experience. In this paper, we formally study the responsiveness and smoothness trade-off in DASH through analysis and experiments. We show that client-side buffered video time is a good feedback signal to guide video adaptation. We then propose novel video rate control algorithms that balance the needs for video rate smoothness and high bandwidth utilization. We show that a small video rate margin can lead to much improved smoothness in video rate and buffer size. The proposed DASH designs are also extended to work with multiple CDN servers. We develop a fully-functional DASH system and evaluate its performance through extensive experiments on a network testbed and the Internet. We demonstrate that our DASH designs are highly efficient and robust in realistic network environment.

Categories and Subject Descriptors

H.5.1 [**Information Systems**]: Multimedia Information Systems—*Video(e.g., tape, disk, DVI)*

General Terms

Design

Keywords

Adaptation, DASH, Emulab, Multiple CDN, SVR

1. INTRODUCTION

Video traffic dominates the Internet. The recent trend in online video streaming is *Dynamic Adaptive Streaming over HTTP* (DASH) that provides uninterrupted video streaming service to users with dynamic network conditions and heterogeneous devices. Notably, Netflix's online video streaming service is implemented using DASH [1, 2]. In DASH, a video content is encoded into multiple versions at different rates. Each encoded video is further fragmented into small video chunks, each of which normally contains seconds or tens of seconds worth of video. Video chunks can be served to clients using standard HTTP servers in either live or on-demand fashion. Upon network condition changes, a client can dynamically switch video version for the chunks to be downloaded.

Different from the traditional video streaming algorithms, DASH does not directly control the video transmission rate. Transmission rate of a chunk is totally controlled by the TCP protocol, which reacts to network congestion along the server-client path. Intuitively, if TCP throughput is high, DASH should choose a high video rate to give user better video quality; if TCP throughput is low, DASH should switch to a low video rate to avoid playback freezes. To maximally utilize throughput achieved by TCP and avoid video freezes, DASH video adaptation should be *responsive* to network congestion level shifts. On the other hand, TCP congestion control incurs inherent rate fluctuations; and cross-traffic rate has both long-term and short-term variations. Adapting video rate to short-term TCP throughput fluctuations will significantly degrade user experience. It is therefore desirable to adapt video rate *smoothly*.

In this paper, we propose client-side video adaptation algorithms to strike the balance between the responsiveness and smoothness in DASH. Our algorithms use client-side *buffered video time* as feedback signal. We show that there is a fundamental conflict between buffer size smoothness and video rate smoothness, due to the inherent TCP throughput variations. We propose novel video rate adaptation algorithms that smoothly increase video rate as the available network bandwidth increases, and promptly reduce video rate in response to sudden congestion level shift-ups. We further show that imposing a buffer cap and reserving a small video rate margin can simultaneously decrease buffer size oscillations and video rate fluctuations. Adopting a machine-learning based TCP throughput prediction algorithm, we also extend our DASH designs to work with multiple CDN servers. Our contribution is four-fold.

1. We formally study the responsiveness and smoothness trade-off in DASH through analysis and experiments. We show that buffered video time is a good reference signal to guide video rate adaptation.

2. We propose novel rate adaptation algorithms that balance the needs for video rate smoothness and bandwidth utilization. We show that a small video rate margin can lead to much improved smoothness in video rate and buffer size.

3. We are the first to develop DASH designs that allow a client to work with multiple CDN servers. We show that machine-learning based TCP throughput estimation algorithms can ef-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CoNEXT'12, December 10–13, 2012, Nice, France.
Copyright 2012 ACM 978-1-4503-1775-7/12/12 ...$15.00.

fectively guide DASH server switching and achieve the multiplexing gain.

4. We implement the proposed algorithms into a fully-functional DASH system, which is evaluated through extensive experiments on network testbed and the Internet. We demonstrate that our DASH designs are highly efficient and robust in realistic network environment.

The rest of the paper is organized as follows. Section 2 describes the related work. DASH designs for single server are developed in Section 3. Extensions to the multiple-server case are presented in Section 4. In Section 5, we report the experimental results for single-server case and multi-server case on both Emulab testbed and the Internet. We conclude the paper with summary and future work in Section 6.

2. RELATED WORK

Although DASH is a relatively new application, due to its popularity, it has generated lots of research interests recently. In [3], Watson systematically introduced the DASH framework of Netflix, which is the largest DASH stream provider in the world. In [4], authors compared rate adaptation of three popular DASH clients: Netflix client, Microsoft Smooth Streaming [5], and Adobe OSMF [6]. They concluded that none of them is good enough. They are either too aggressive or too conservative. Some clients even just jump between the highest video rate and the lowest video rate. Also, all of them have relatively long response time under network congestion level shift. It was shown in [7] that dramatic video rate changes lead to inferior user quality-of-experience. They further proposed to gradually change video rate based on available bandwidth measurement. In [8], authors proposed a feedback control mechanism to control the sending buffer size on the server side. Our video adaptation is driven by buffered video time on the client-side, which has direct implication on client video playback. Our scheme does not require any change on the server side. There are also papers on DASH in wireless and mobile networks. In [9], several adaptive media players on the market were tested to see how they perform in challenging streaming scenarios in a mobile 3G network. In [10], Mueller et al implemented a DASH system and proved it works in vehicular network environments. In [11], a DASH-like algorithm was proposed for a server to regulate the video uploading from a mobile client. In DASH, it is important to predict TCP throughput and quickly detect congestion level shifts. One way is to monitor the path using network bandwidth measurement tools like pathload [12]. But measuring available bandwidth itself injects probing traffic into the path, and it may take long time to converge to an acceptable result. And the accuracy of such tools is not guaranteed. And in [13] and [14], authors presented history-based and machine-learning-based TCP throughput prediction. In DASH, video chunks are continuously transmitted from the server to the client. In this scenario, TCP throughput data can be collected in realtime. In our experiments, we found that even simple history-based TCP throughput prediction can achieve higher accuracy than those reported in [13] and [14]. For multi-server DASH, a client needs to continuously evaluate the throughput of a DASH server before switching to it. We implement the light-weight machine learning approach proposed in [14] for the TCP throughput predict of candidate DASH servers.

3. DASH WITH SINGLE SERVER

We start with DASH system where a client only downloads video chunks from a single server. We will extend our designs to the multiple server case in Section 4.

3.1 Buffered Video Time

To sustain continuous playback, a video streaming client normally maintains a video buffer to absorb temporary mismatch between video download rate and video playback rate. In conventional single-version video streaming, video buffer is measured by the size of buffered video, which can be easily mapped into buffered video playback time when divided by the average video playback rate. In DASH, different video versions have different video playback rates. Since a video buffer contains chunks from different versions, there is no longer direct mapping between buffered video size and buffered video time. To deal with multiple video versions, we use buffered video time to directly measure the length of video playback buffer.

Buffered video time process, denoted by $q(t)$, can be modeled as a single-server queue with constant service rate of 1, i.e., with continuous playback, in each unit of time, a piece of video with unit playback time is played and dequeued from the buffer. The enqueue process is driven by the video download rate and the downloaded video version. Specifically, for a video content, there are L different versions, with different playback rates $V_1 < V_2 < \cdots < V_L$. All versions of the video are partitioned into chunks, each of which has the same playback time of Δ. A video chunk of version i has a size of $V_i\Delta$. A client downloads video chunks sequentially, and for each chunk, he can choose one out of the L versions. Without loss of generality, a client starts to download chunk k from version i at time instant $t_k^{(s)}$. Then the video rate requested by the client for the k-th chunk is $v(k) = V_i$. Let $t_k^{(e)}$ be the time instant when chunk k is downloaded completely. In a “greedy" download mode, a client downloads chunk k right after chunk $k-1$ is completely downloaded, in other words, $t_{k-1}^{(e)} = t_k^{(s)}$. For the buffered video time evolution, we have:

$$q(t_k^{(e)}) = \Delta + \max\left(q(t_k^{(s)}) - (t_k^{(e)} - t_k^{(s)}), 0\right), \tag{1}$$

where the first term is the added video time upon the completion of the downloading of chunk k, the second term reflects the fact that the buffered video time is consumed linearly at rate 1 during the downloading of chunk k.

Using fluid approximation, we evenly distribute the added video time of Δ over the download interval $(t_k^{(s)}, t_k^{(e)}]$, then

$$\begin{aligned} \frac{dq(t)}{dt} &= \frac{\Delta}{t_k^{(e)} - t_k^{(s)}} - \mathbf{1}(q(t) > 0), & (2) \\ &= \frac{v(k)\Delta}{v(k)(t_k^{(e)} - t_k^{(s)})} - \mathbf{1}(q(t) > 0), & (3) \\ &= \frac{\bar{T}(k)}{v(k)} - \mathbf{1}(q(t) > 0), \quad t \in (t_k^{(s)}, t_k^{(e)}], & (4) \end{aligned}$$

where $\mathbf{1}(\cdot)$ is the indicator function, and $\bar{T}(k)$ is the average TCP throughput when downloading chunk k. The buffered video time remains constant when the requested video rate $v(k)$ exactly matches $\bar{T}(k)$ which is not practical. In practice, $v(k)$ can only assume one of the L predefined video rates. There will be unavoidable rate mismatches, thus buffer fluctuations.

3.2 Control Buffer Oscillations

From (4), if the requested video rate is higher than the actual TCP throughput, the buffered video time decreases, and video playback freezes whenever $q(t)$ goes down to zero; if the requested video rate is lower than the actual TCP throughput, the buffered video time ramps up, it suggests that user gets stuck at low video rate even though his connection supports higher rate. A responsive video

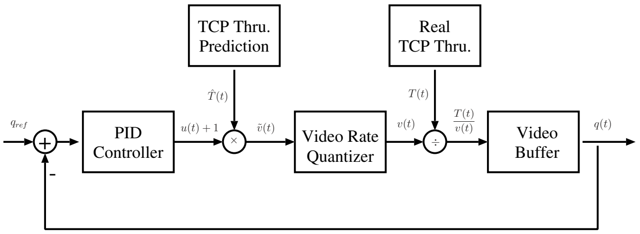


Figure 1: PID control oriented adaptive streaming

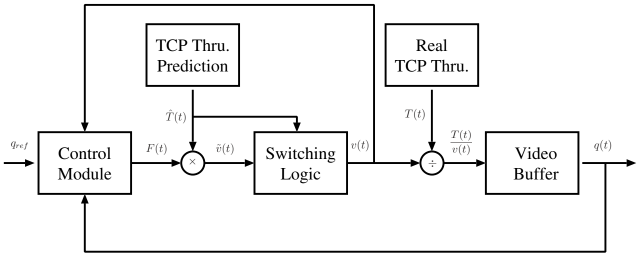


Figure 2: Buffer size oriented adaptive streaming

adaptation scheme should control video rate to closely track TCP throughput so that $q(t)$ stays within a bounded region. As a result, no video freeze happens, and the requested video rate matches the TCP throughput in long-run.

To maintain a stable queue length, one can employ a simple rate adaptation scheme:

$$v(k) = \underset{\{V_i, 1 \le i \le L\}}{\mathbf{argmin}} \quad |V_i - \hat{T}(k)|,$$

where $\hat{T}(k)$ is some estimate of TCP throughput before downloading chunk k. In other words, a client always downloads the version with a rate “closest" to the estimated TCP throughput. However, it is well-known that it is hard to accurately estimate TCP throughput. Such an open-loop design is not robust against TCP throughput estimation errors. To address this problem, we investigate a closed-loop feedback control design for video rate adaptation.

Instead of directly matching the requested video rate $v(k)$ with TCP throughput estimate, we use the evolution of buffered video time $q(t)$ as feedback signal to adjust $v(k)$. One straightforward way is to set up a reference queue length q_{ref}, i.e. the target video buffer time, and build a PID controller to regulate the requested video rate.

PID is the most commonly used feedback controller in industrial control systems. A PID controller calculates an "error" value as the difference between a measured process variable and a desired set point. The controller attempts to minimize the error by adjusting the process control inputs. The PID controller calculation (algorithm) involves three separate parameters: the proportional factor, the integral factor and derivative factor, denoted by K_P, K_I, and K_D respectively. Heuristically, these values can be interpreted in terms of time: K_P depends on the present error, K_I on the accumulation of past errors, and K_D is a prediction of future errors, based on current rate of change. The weighted sum of these three actions is used to adjust the process via the output of the controller. In practice, a PID controller doesn’t have to set all the three parameters. There are very common usage of variations of PID controller. In our research, because of inherent rate fluctuations of TCP transmission, we use PI controller instead of PID controller because the derivative factor may amplify the impact of TCP fluctuations.

Figure 1 illustrates the diagram of the control system. We adopt a Proportional-Integral (PI) controller, with control output driven by the deviation of buffered video time:

$$u(t) = K_p(q(t) - q_{ref}) + K_I \int_0^t (q(\tau) - q_{ref}) d\tau,$$

where K_p and K_I are the P and I control coefficients respectively. The target video rate for chunk k is

$$\tilde{v}(k) = (u(t_k^{(s)}) + 1)\hat{T}(t_k^{(s)}),$$

where $\hat{T}(t_k^{(s)})$ is the TCP throughput estimate right before downloading chunk k. Taking into account the finite discrete video rates, the actual requested video rate for chunk k is the highest video rate lower than $\tilde{v}(k)$,

$$v(k) = Q(\tilde{v}(k)) \triangleq \max_{\{V_i : V_i \le \tilde{v}(k)\}} V_i, \tag{5}$$

where $Q(\cdot)$ is the quantization function. When $q(t)$ oscillates around q_{ref} with small amplitude, the control signal $u(t)$ is small, the requested video rate is set close to the predicted TCP throughput. The throughput estimation error and video rate quantization error will be absorbed by the video buffer and closed-loop control.

3.3 Control Video Rate Fluctuations

To accurately control the buffer size, one has to constantly adapt the requested video rate to realtime TCP throughput. If the achieved TCP throughput is larger than the requested video rate, the buffer size will increase and the feedback control module will increase the video rate, which, according to (4), will slow down buffer increase, and vice versa. Since TCP throughput is by-nature time varying, the requested video rate will also incur constant fluctuations.

From control system point of view, there is a fundamental conflict between maintaining stable video rate and stable buffer size, due to the unavoidable network bandwidth variations. From end user point of view, video rate fluctuations are much more perceivable than buffer size oscillations. Recent study has shown that switching back-and-forth between different video versions will significantly degrade user video experience [7]. Meanwhile, buffer size variations don’t have direct impact on video streaming quality as long as the video buffer does not deplete. In this section, we revisit our video rate adaptation design with the following goals:

1. avoid video rate fluctuations triggered by short-term bandwidth variations and TCP throughput estimation errors;

2. increase video rate smoothly when the available network bandwidth is consistently higher than the current video rate;

3. quickly decrease video rate upon the congestion level shift-ups to avoid video playback freezes.

To simultaneously achieve the three goals, one has to strike the right balance between the responsiveness and smoothness of video rate adaptation upon network bandwidth increases and decreases. Classical feedback control, such as those presented in Figure 1, is no longer sufficient. We develop a new rate control system as shown in Figure 2.

3.3.1 Control Module

The control module still uses buffered video time $q(t)$ as feedback signal, since it directly reflects the mismatch between the video rate and realtime TCP throughput. Instead of controlling $q(t)$ to a target level q_{ref}, we only use $q(t)$ to guide the video rate selection. To determine video rate $v(k)$ for chunk k, we need TCP

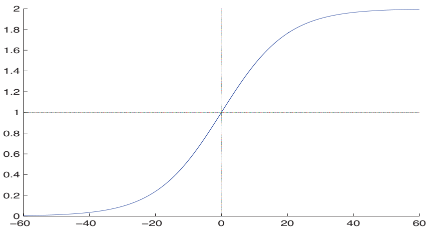


Figure 3: Adjustment Function for Buffer Size Deviation

throughput prediction $\hat{T}(k)$ and an adjustment factor $F(k)$, which is a function of the target buffer size, current buffer size, previous buffer size, and the current video rate:

$$F(k) = F_q(k) * F_t(k) * F_v(k), \tag{6}$$

with

$$F_q(k) = 2 * \frac{e^{p*(q(t_k^{(s)})-q_{ref})}}{1 + e^{p*(q(t_k^{(s)})-q_{ref})}} \tag{7}$$

$$F_t(k) = \frac{\Delta}{\Delta - (q(t_k^{(s)}) - q(t_{k-1}^{(s)}))} \tag{8}$$

$$F_v(k) = \frac{V_L}{v(k-1) + W} + \frac{W}{V_L + W} \tag{9}$$

In (6), the adjustment factor $F(k)$ is a product of three sub-factors: buffer size adjustment $F_q(k)$, buffer trend adjustment $F_t(k)$, and video chunk size adjustment $F_v(k)$, which we explain one-by-one in the following.

Buffer Size Adjustment $F_q(k)$ is an increasing function of buffer size deviation $q(t_k^{(s)}) - q_{ref}$ from the target buffer size in (7). Larger buffer size suggests one should be more aggressive in choosing higher video rate. As illustrated in Figure 3, when the buffer size matches the target q_{ref}, the adjustment is neutral (with value 1); when the deviation is small, the adjustment is approximately $1 + p(q(t_k^{(s)}) - q_{ref})$, mimicking a simple P-controller, with stationary output of 1 and $K_p = p$; when the deviation is large, the adjustment factor increases/decreases smoothly with upper bound of 2 and lower bound of 0. This is to avoid F_q overpowers the other two factors.

Buffer Trend Adjustment $F_t(k)$ is an increasing function of buffer size growth $q(t_k^{(s)}) - q(t_{k-1}^{(s)})$ since the downloading of the previous video chunk, as calculated in (8), where Δ is the video time contained in one chunk. If there is no buffer size growth, the adjustment is neutral (with value 1). If the buffer size grows fast, it suggests that the previous video rate is too conservative, one should increase the video rate; if the buffer size decreases fast, it suggests that the previous video rate is too aggressive, one should decrease the video rate. From equations (1) to (4), in a greedy download mode with $t_{k-1}^{(e)} = t_k^{(s)}$, it can be shown that with fluid approximation:

$$F_t(k) = \frac{\bar{T}(k-1)}{v(k-1)} = \frac{dq(t)}{dt} + \mathbf{1}(q(t) > 0). \tag{10}$$

In other words, $F_t(k)$ is the ratio between the actual download throughput and video rate for chunk $k-1$. F_t is essentially a *Derivative D-controller*, that responds fast to increase/decrease trend in buffered video time.

Video Chunk Size Adjustment $F_v(k)$ is a decreasing function of the previous video rate $v(k-1)$, calculated in (9), where W is a constant. If $v(k-1) = V_L$, the adjustment is neutral; if $v(k-1) < V_L$, $F_v(k) > 1$. This is because HTTP adaptive streaming uses TCP transmission. If a chunk is small, TCP has to go through slow-start process to open up its congestion window, leading to low TCP throughput even if the available bandwidth is much higher. This compensation enables fast rate increase when a DASH session starts or resumes from a low video rate. If the client connects to the server with persistent HTTP connection, there will be no such problem because only the first chunk will experience slow start. In such scenario, we can just set this adjustment factor to constant 1. Notice that, each adjustment factor assumes value of 1 when the system is at the equilibrium point, i.e., $q(t_k^{(s)}) = q_{ref}$, $q(t_k^{(s)}) = q(t_{k-1}^{(s)})$, $v(k-1) = V_L$. When the system operates within the neighborhood of the equilibrium point, each adjustment factor takes small positive or negative deviation from one. The total deviation of their product from one is approximately the summation of the individual deviations, similar to the PI controller in the previous section. Different from the PI controller, the product deviation changes smoothly within a bounded region when the system operates away from the equilibrium point.

3.3.2 Rate Switching Logic

After we get the final adjustment factor $F(k)$, similar to the buffer control case in Section 3.2, we can multiply it with the TCP throughput estimate and set a target video rate $\tilde{v}(k) = F(k)\hat{T}(t_k^{(s)})$, then use the quantization function $Q(\cdot)$ in (5) to convert it to a discrete video rate $v(k)$. If we adjust the video rate directly according to the quantized target video rate, there will be again frequent fluctuations. To resolve this, a rate switching logic module is added after the quantizer as shown in Figure 2. It controls video rate switch according to algorithm 1.

Algorithm 1 Smooth Video Adaptation Algorithm.

```
$\tilde{v}(k) = F(k)\hat{T}(t_k^{(s)})$;
if $q(t_k^{(s)}) < \frac{q_{ref}}{2}$ then
    $v(k) = Q(\bar{T}(k-1))$;
    return;
else if $\tilde{v}(k) > v(k-1)$ then
    $Counter++$
    if $Counter > m$ then
        $v(k) = Q(\hat{T}(t_k^{(s)}))$;
        $Counter = 0$;
        return
    end if
else if $\tilde{v}(k) < v(k-1)$ then
    $Counter = 0$
end if
$v(k) = v(k-1)$; return;
```

If the buffer size drops below half of the target size $\frac{q_{ref}}{2}$, it indicates that the current video rate is higher than the TCP throughput, and there is a danger of buffer depletion and playback freeze. We then immediately reduce the video rate to $v(k) = Q(\bar{T}(k-1))$, where $\bar{T}(k-1)$ is the actual TCP throughput of the previous chunk transmission. Due to the quantization, $v(k) < \bar{T}(k-1)$, if TCP throughput in the current round is close to the previous round, the buffer size is expected to increase until it goes back to above $\frac{q_{ref}}{2}$. If the buffer size is larger than $\frac{q_{ref}}{2}$, we consider it safe to keep the current rate or switch up to a higher rate. To avoid small timescale fluctuations, video rate is switched up only if the target video rate $\tilde{v}(k)$ calculated by the controller is larger than the current rate

$v(k-1)$ for m consecutive chunks. Whenever a switch-up is triggered, the video rate is set to match the TCP throughput estimate $\hat{T}(t_k^{(s)})$. Before the switch-up counter reaches m, if the target video rate calculated for one chunk is smaller than the current video rate, the switch-up counter will be reset and start over.

The parameter m controls the trade-off between the responsiveness and smoothness of rate adaptation. Larger m will definitely make the adaptation smoother, but sluggish. If the video rate is at low levels, the user will have to watch that video rate for a long time even if there is enough bandwidth to switch up. To address this problem, we dynamically adjust m according to the trend of buffer growth. More specifically, for chunk k, we calculate a switch-up threshold as a decreasing function of the recent buffer growth: $m(k) = f_m(q(t_k^{(s)}) - q(t_{k-1}^{(s)}))$, and video rate is switched up if the switch-up counter reaches $(m(k)+m(k-1)+m(k-2))/3$. The intuition behind this design is that fast buffer growth suggests TCP throughput is persistently larger than the current video rate, one should not wait for too long to switch up. Similar to (10), it can be shown that if buffer is non-empty,

$$q(t_k^{(s)}) - q(t_{k-1}^{(s)}) = \Delta\left(1 - \frac{v(k-1)}{\bar{T}(k-1)}\right).$$

One example dynamic-m function we use in our experiments is a piece-wise constant function which we got from empirical study:

$$m(k) = \begin{cases} 1 & \text{if } q(t_k^{(s)}) - q(t_{k-1}^{(s)}) \in [0.4\Delta, \Delta); \\ 5 & \text{if } q(t_k^{(s)}) - q(t_{k-1}^{(s)}) \in [0.2\Delta, 0.4\Delta); \\ 15 & \text{if } q(t_k^{(s)}) - q(t_{k-1}^{(s)}) \in [0, 0.2\Delta); \\ 20 & \text{otherwise.} \end{cases} \tag{11}$$

3.4 Control Buffer Overflow

So far we assume a “greedy" client mode, where a client continuously sends out “GET" requests to fully load TCP and download video chunks at the highest rate possible. In practice, this may not be plausible for the following reasons: 1) A DASH server normally handles a large number of clients. If all clients send out “GET" requests too frequently, the server will soon be overwhelmed; 2) If the requested video rate is consistently lower than TCP throughput, the buffered video time quickly ramps up, leading to *buffer overflow*. In Video-on-Demand (VoD), it means that the client pre-fetches way ahead of the its current playback point, which is normally not allowed by a VoD server. In live video streaming, pre-fetching is simply not possible for content not yet generated. The buffered video time is upper-bounded by the user tolerable video playback lag, which is normally in the order of seconds; 3) Finally, fully stressing TCP and the network without any margin comes with the risk of playback freezes, especially when the client doesn’t have large buffer size, like in the live streaming case.

To address these problems, we introduce a *milder* client download scheme. To avoid buffer overflow, we introduced a *buffer cap* q_{max}. Whenever the buffered video time goes over q_{max}, the client keeps idle for a certain timespan before sending out the request for the next chunk. Also, to mitigate the TCP and network stresses, we reserve a *video rate margin* of $0 \leq \rho_v < 1$. For any target video rate $\tilde{v}(k)$ calculated by the controllers in the previous sections, we only request a video rate of $v(k) = Q((1-\rho_v)\tilde{v}(k))$. With the video rate margin, the buffered video time will probably increase. When $q(t)$ goes over q_{max}, the client simply inserts an idle time of $q(t) - q_{max}$ before sending out the next download request. As will be shown in our experiments, even a small video rate margin can simultaneously reduce the buffer size oscillations and video rate fluctuations a lot.

4. DASH WITH MULTIPLE SERVERS

While most DASH services employ multiple servers hosting the same set of video contents, each client is only assigned to one server [1]. It is obviously more advantageous if a DASH client is allowed to dynamically switch from one server to another, or even better, simultaneously download from multiple servers. Our video adaptation algorithms in Section 3 can be easily extended to the case where a client can download from multiple servers. We consider two cases. In the first case, given n servers, a client always connects to the server which can provide the highest video download rate. In the second case, a client simultaneously connects to s out of n servers, and downloads different chunks from different servers then combine them in the order of the video.

4.1 TCP Throughput Prediction

In single-server study, since we keep downloading video chunks from the same server, we can use simple history-based TCP throughput estimation algorithm [13] to predict the TCP throughput for downloading new chunks. With multiple servers, it is necessary for a client to estimate its TCP throughput to a server even if it has not downloaded any chunk from that server. We adopt a light-weight TCP throughput prediction algorithm proposed in [14]. The authors showed that TCP throughput is mainly determined by packet loss, delay and the size of the file to be downloaded. They propose to use the Support Vector Regress (SVR) algorithm [15] to train a TCP throughput model $\hat{T}(p_l, p_d, f_s)$ out of training data consisting of samples of packet loss rate $p_l^{(i)}$, packet delay $p_d^{(i)}$, file size $f_s^{(i)}$ and the corresponding actual TCP throughput $T^{(i)}$. To predict the current TCP throughput, one just need to plug in the current measured packet loss, delay, and the download file size. For our purpose, we download each chunk as a separate file. Chunks from different video versions have different file sizes. In our SVR TCP model, we use video rate V_i in place of file size f_s.

4.2 Dynamic Server Selection

In the first case, we allow a client to dynamically switch to the server from which it can obtain the highest TCP throughput. While a client downloads from its current DASH server, it constantly monitors its throughput to other candidate servers by using the SVR TCP throughput estimation model. To accommodate the SVR throughput estimation errors, which was reported around 20% in [14], the client switches to a new DASH server only if the estimated throughput to that server is at least 20% higher than the achieved throughput with the current DASH server.

To avoid wrong server switch triggered by SVR estimate errors, we use trial-based transition. When a client decides to switch to a new server, it establishes TCP connections with the new server, and also keeps the connections with the current server. It sends out “GET" requests to both servers. After a few chunk transmissions, the client closes the connections with the server with smaller throughput and uses the one with larger throughput as its current DASH server.

4.3 Concurrent Download

In the second case, a clients is allowed to simultaneously download chunks from s out of n servers. The client-side video buffer is fed by s TCP connections from the chosen servers. The video rate adaptation algorithms in Section 3 work in a similar way, by just replacing TCP throughput estimate from the single server with the aggregate TCP throughput estimate from s servers.

A simple client video chunk download strategy is to direct different chunk download requests to different servers, with the number of chunks assigned to a server proportional to its TCP throughput

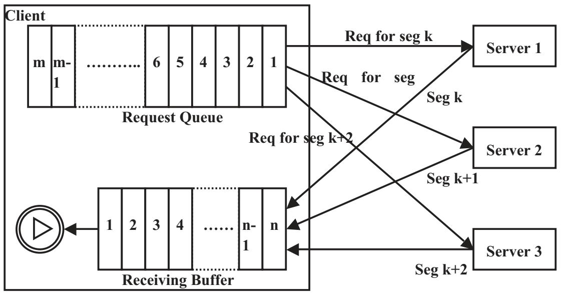


Figure 4: Self-adaptive Concurrent DASH Downloading.

estimate. But such a scheme is not robust against throughput estimation error. If the throughput to a server is overestimated, the server will be overloaded, and the chunks assigned to it cannot be downloaded in time, which will delay the overall video playback. To solve this problem, we introduce self-adaptive chunk assignment and timeout chunk retransmission.

Self-adaptive Chunk Assignment. We assume that all the DASH servers have the same copies of all the chunks of the same video. As illustrated in Figure 4, on the client side, all connections to chosen servers share a common chunk download request queue, which is sequentially injected with download requests for vide chunks at rates specified by the video adaptation algorithm. After a connection finishes downloading a chunk, it takes another chunk download request from the head of the download queue. This way, the distribution of chunks to each connection is automatically regulated by its TCP throughput, instead of the TCP throughput prediction. To avoid video buffer overflow, idle time is inserted to all connections.

Chunk Timeout Retransmission. Even with self-adaptive assignment, if the connection to a server suddenly incurs bandwidth deficiency, the chunk already assigned to that connection still cannot be downloaded in-time, which again will delay the video playback. To avoid this problem, we introduce timeout-based chunk retransmission mechanism. Based on the history-based throughput estimation and the chosen video rate, we can calculate the expected transmission time for each chunk on each connection. Every time a HTTP "GET" request for a chunk is sent to a server, one timer starts for the chunk. The timer will expire in twice of the expected chunk transmission time. If the chunk has not been completely downloaded before the timer expires, a download request for that chunk will be sent to another server.

4.4 Probabilistic Server Switching

If there are a large number of clients using the same DASH service, there is a possibility that many clients simultaneously detect and switch to the same lightly loaded server which will overload that server quickly, the client will then switch back. This will definitely causes oscillations in server selection to the whole system. To resolve this problem, probabilistic switching can be introduced into the system. In this solution, when a client detects that it should switch to a better server, it sorts all the servers that are better than its current server with their predicted throughput, and then switch to different servers with different probability. Assume there are n servers that are better than the current server, and T_i is the predicted throughput of the i^{th} server, T_c is the real-time throughput of the current server, then the probability to switch to the i^{th} server can be calculated as $T_i/(\sum_{k=1}^{n} T_k + T_c)$, also, the probability to stay at the current server is $T_c/(\sum_{k=1}^{n} T_k + T_c)$. This induces that a lighter loaded server has higher probability to be chosen and probabilistic switching reduces the possibility of synchronization between clients.

In this paper, we didn't implement probabilistic switching in our system because we only focus on single client case. In multiple clients scenarios, this mechanism should be implemented to avoid system oscillations.

5. PERFORMANCE EVALUATION

We implemented our video rate adaptation algorithms into a fully-functional DASH system, and extensively evaluated it using controlled experiments on a network testbed, the Emulab [16], as well as real Internet experiments, with both wireline and wireless clients.

5.1 System Implementation

Our DASH system runs on linux/unix platforms. It consists of a vanilla Apache HTTP server and a customized HTTP client. All the proposed video rate adaptation algorithms are implemented on the client side using C++. For multi-server experiments, we also implemented the SVR-based TCP throughput estimation algorithm proposed in [14]. A light-weight network measurement thread runs on the server and the client to periodically collect packet loss and delay statistics for TCP throughput estimation. To train the SVR TCP throughput model, the client sends HTTP "GET" requests to a server to get video chunk files. Since TCP throughput is affected by file size, the training data need to be collected for different video rates. Since we use a large number of video rates in our experiments, that would make the training process too long and too intrusive to the network. To reduce training experiments, we design a mechanism to collect TCP throughput data for all the video rates in one single HTTP "GET" request. On the server side, the file used for training is larger than the size of a video chunk at the highest video rate. After the client sends out a "GET" request to download the training file, it records the time lags when the total number of the received data bytes reaches the video chunk sizes at all possible video rates. TCP throughput to download a chunk at a video rate is calculated as the video chunk size divided by the recorded time lag for that rate. This way we can get the training data for all the video rates in one transmission. As mentioned above, we use non-persistent HTTP connection in this paper, so a separate HTTP connection is used for downloading each video chunk in the training. For persistent HTTP connection, we should use a persistent HTTP connection for downloading video chunks in training. It is expected that the chunk size has less impact on the obtained TCP throughput in that case.

5.2 Emulab Experiments Setup

Emulab network testbed allows us to conduct controlled and repeatable experiments to compare different designs. The structure of the our Emulab testbed is shown in Figure 5. It consists of five nodes, among which one node acts as the DASH client, the other four nodes are DASH servers. Each node runs the standard FreeBSD of version 8.1. The servers have Apache HTTP server of version 2.4.1 installed. Also, Ipfw and Dummynet are used on the servers to control the bandwidth between nodes.

In all our experiments, the server provides 51 different video rates, ranging from $100Kbps$ to $5.1Mbps$, with rate gap between two adjacent versions $100Kbps$. The link capacity between the client and server is set to be $5Mbps$. In this setting, even if there is no background traffic, the client still cannot sustain the highest video rate. To generate realistic network bandwidth variations with congestion level shift, we inject background TCP traffic between the servers and the client. Twenty TCP connections are established for each server-client pair. We control the rate at which data is

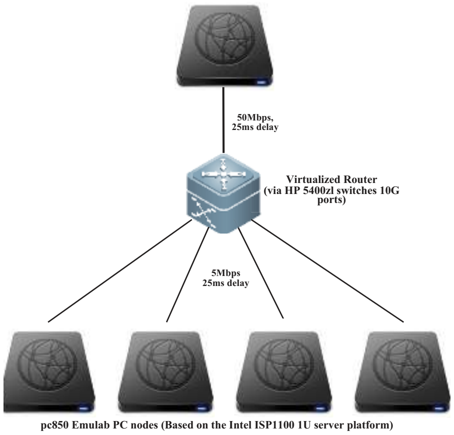

50Mbps,
25ms delay
Virtualized Router
(via HP 5400zl switches 10G
ports)
5Mbps
25ms delay
pc850 Emulab PC nodes (Based on the Intel ISP1100 1U server platform)

Figure 5: Emulab Testbed

injected to a background TCP connection to emulate network congestion level shift. The background TCP traffic we use for all single server experiments is shown in Figure 6. The aggregate background traffic rate jumps between three different levels, oscillations within the same level are due to TCP congestion control mechanisms.

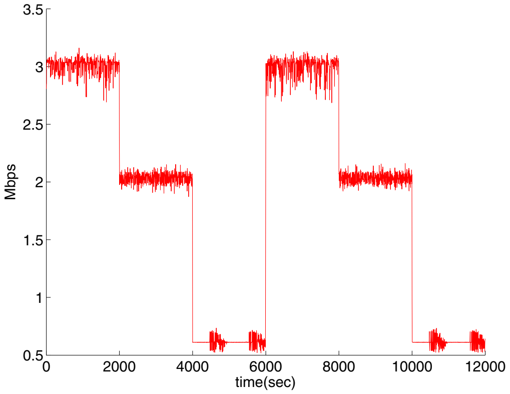
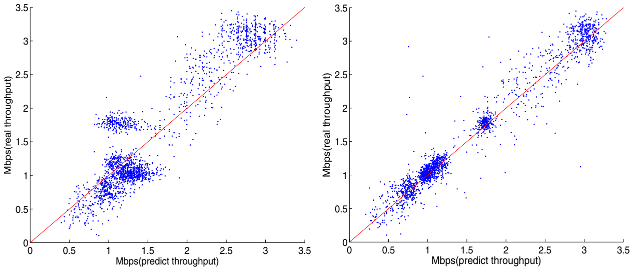


Figure 6: Background Traffic with Level-shift

5.3 TCP Throughput Prediction Accuracy

For DASH from single server, since a client continuously downloads video chunks from the same server which makes realtime TCP throughput data available, we use simple history-based TCP throughput prediction. The client measures the time it takes to download each chunk and calculates the average TCP throughput for each chunk. To predict TCP throughput for the next chunk downloading, it simply takes an average of TCP throughput for the previous W chunks, discarding the largest one and the smallest one. If W is too small, the estimate tends to be noisy, if W is too large, the estimate reacts slowly to sudden congestion level shift. In our experiments, we set $W = 10$. While some other history-based TCP throughput prediction algorithms have been proposed, e.g., [13], we found in our experiments that even the simple one works well in the context of DASH. For DASH with multiple servers, if a client has never downloaded from a server, there is no TCP throughput history to extrapolate on. We instead use SVR TCP throughput prediction [14] with initial offline training and light-weight online measurement. For our controlled experiments, single run of offline training is good enough. In practice, offline training can be done periodically to make the SVR model up-to-date.



(a) SVR: average error 25.7% (b) History: average error 9.6%

Figure 7: TCP Throughput Prediction Accuracy.

We now compare the accuracy of history-based and SVR TCP throughput prediction in our Emulab experiments. TCP background traffic follows the same trend as in Figure 6. To get more samples, we increase the duration of each congestion level to 160 minutes. During the experiments, the client downloads video chunks at all the 51 video rates. Figure 7 compares the accuracy of SVR prediction with history-based prediction. We can see that our history-based prediction is obviously better than SVR prediction. The average error for history-based prediction is 9.6%, while the error for SVR prediction is about 25.7%, which is consistent with the results reported in [14]. The high accuracy of history-based prediction in DASH is because chunks are downloaded by TCP continuously and TCP throughput for adjacent chunks are highly correlated. On the other hand, SVR prediction is light-weight, its accuracy is already good enough to guide the server selection in multi-server DASH. In our following experiments, the accuracy of history-based prediction is even lower than 5%. This is because the TCP throughput prediction accuracy at low video rates is worse than at high video rates. At low video rates, because the video chunk size is small, very small absolute estimation error can cause large relative error which doesn't happen for high video rates. In this accuracy experiment, we do the transmission for all 51 video rates from $100Kbps$ to $5.1Mbps$ while in real DASH experiments, for most of time, the video rate is higher than $1Mbps$.

5.4 Single-server DASH Experiments

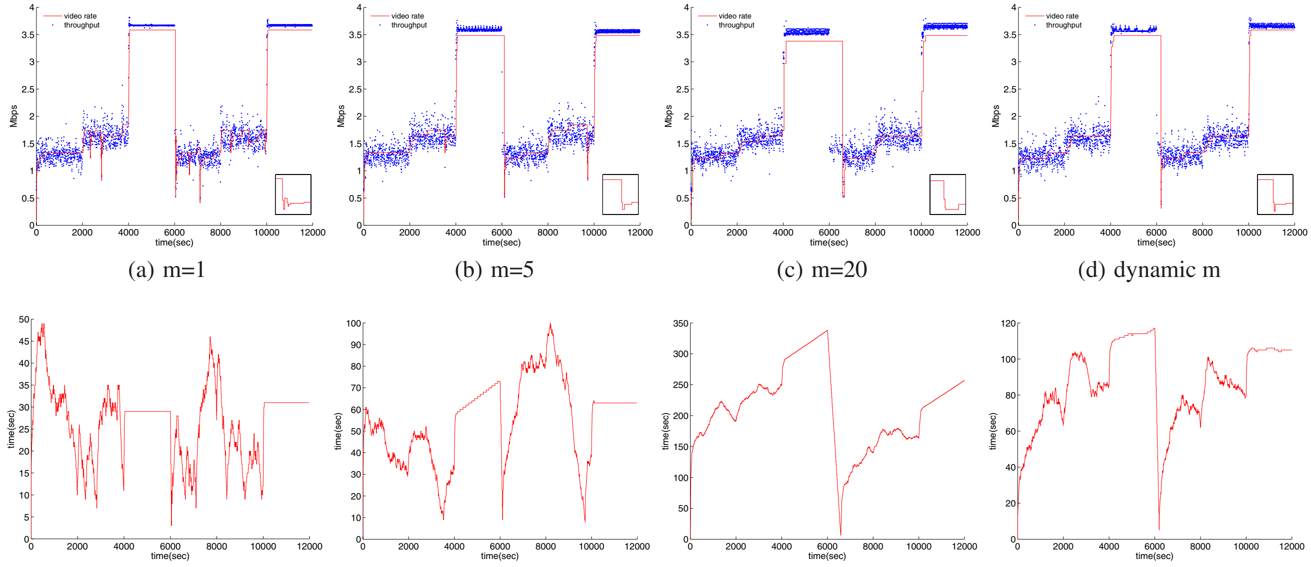
We present Emulab results for DASH from single-server.

5.4.1 *Buffer Size Control*

We first evaluate the performance of PID-based buffer control proposed in Section 3.2. We set $K_p = 0.1$, $K_I = 0.01$ and the reference buffered video time is $q_{ref} = 20$ seconds. The duration for each congestion level is scaled down to 500 seconds. In Figure 8(a), we can see that the buffer size can be controlled around 20 very well. In Figure 8(b), the requested video rate tracks the TCP throughput perfectly under all congestion level shifts. But the frequent video rate fluctuations are not acceptable to users. As stated in Section 3.3, instead of accurately controlling buffer size, one should smoothly reach high video rate under the throughput that can be achieved by TCP.

5.4.2 *Smooth Video Adaptation*

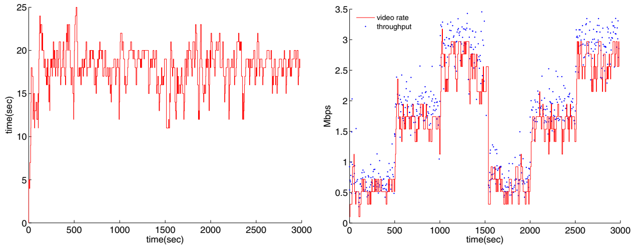
We conduct single-server DASH experiments using the smooth video adaptation algorithm proposed in Section 3.3. Figure 9 shows the results by setting the rate switch-up threshold m to 1, 5, 20, and dynamic value defined in (11) respectively.



(a) m=1 (b) m=5 (c) m=20 (d) dynamic m

(e) m=1 (f) m=5 (g) m=20 (h) dynamic m

Figure 9: Smooth Video Adaptation: (a)-(d): video rate and throughput; (e)-(h): buffer size evolution



(a) buffer size (b) video rate/throughput

Figure 8: PID-based Buffer Size Control.

Table 1 shows the average video rate and buffer size.

Table 1: average video rate and buffer size for single server

m	avg. video rate(Mbps)	avg. buffer size(second)
1	2.16	26.75
5	2.15	55.39
20	2.10	209.17
Dyn.	2.13	86.34

Compared with Figure 8(b), the video rate smoothness has significant improvement. Larger m has smoother video rate, but incurs much larger oscillations in buffer size. This again reflects the fundamental design trade-off between video rate smoothness and buffer size smoothness. The average video rate of all the cases are close, with larger m has slightly lower rate. This is because, with larger m, it takes longer to switch-up video rate after TCP throughput increases. The buffered video time keeps increasing, leading to large buffer size overshoot. Since the rate switch-down is triggered only after the buffer size goes down to half of the reference size, large buffer size overshoots with larger m also make it react slowly to the sudden congestion level leaps. In Figure 9(c), the video rate remains at high level way after the actual TCP throughput goes down around $6,000$ second, until the buffer size falls back to 10 seconds. The small windows at bottom-right corner of Figure 9 (a)-(d) are the zoom-in views of video rate within 200 seconds after the big rate switch-down around $6,000$ second. It is obvious that, with a larger m, it takes much longer time to switch up to a higher rate. In Figure 9(d) and Figure 9(h), by dynamically adjusting m based on the queue length increase trend, dynamic-m simultaneously achieves video rate smoothness of large m and switch responsiveness of small m.

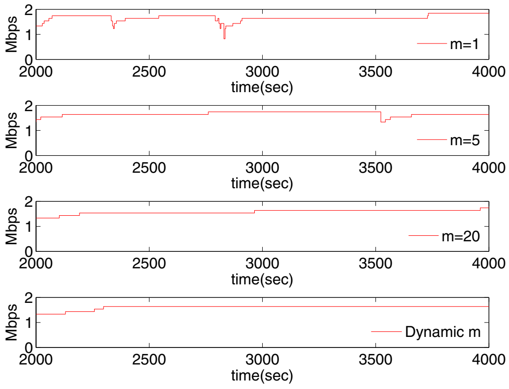
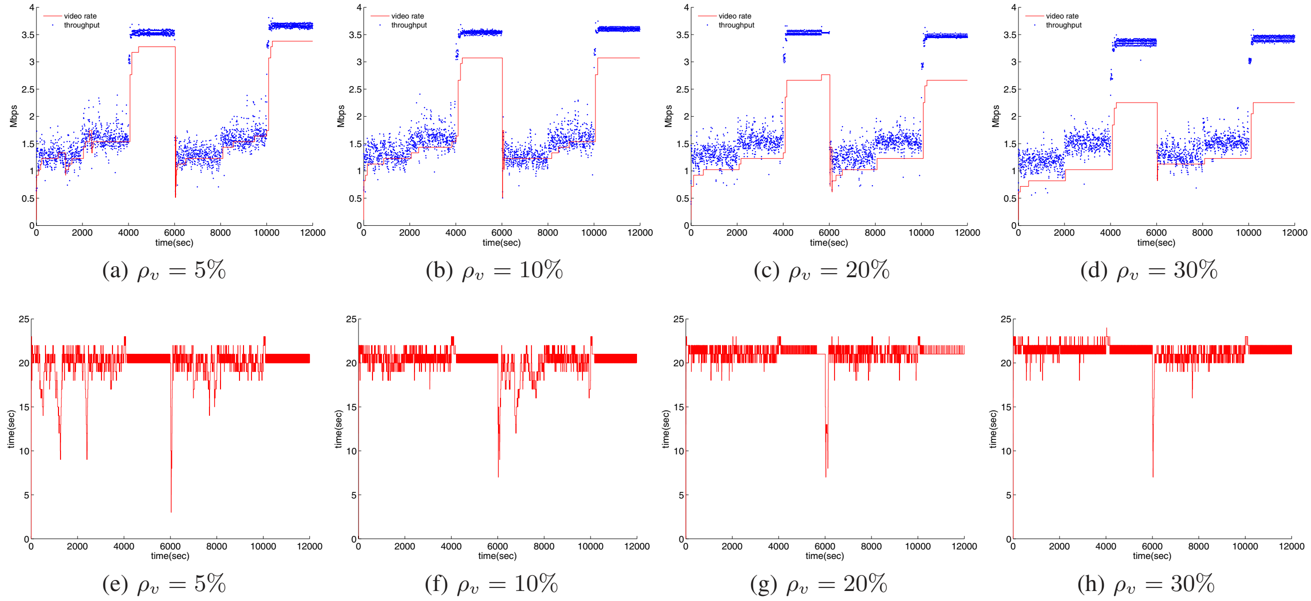


Figure 10: Video Smoothness of different m values

Figure 10 further shows the video rate trends between $2,000$ second and $4,000$ second for different m. For non-dynamic m algorithms, larger m leads to smoother video rate, but as shown in Figure 9, larger m results in slower responsiveness. As for dynamic m algorithm, it maintain good smoothness of the video rate trend while achieving fast responsiveness.

5.4.3 Adaptation with Buffer Cap and Rate Margin

As stated in Section 3.4, large buffer size is not possible in live streaming and not plausible in VoD; reserving a bandwidth margin can potentially reduce video rate fluctuations. In the first set



(a) $\rho_v = 5\%$ (b) $\rho_v = 10\%$ (c) $\rho_v = 20\%$ (d) $\rho_v = 30\%$

(e) $\rho_v = 5\%$ (f) $\rho_v = 10\%$ (g) $\rho_v = 20\%$ (h) $\rho_v = 30\%$

Figure 11: Video Adaptation under Buffer Cap of 20 Seconds and Different Video Rate Margins.

of experiments, we set the buffer cap q_{max} to 20 seconds and try different bandwidth margins. Figure 11 shows the results for different margins. Table 2 shows the average video rate, TCP throughput utilization (calculated as average video rate over average TCP throughput), and average buffer size for each case. With bandwidth margin and buffer cap, the buffer size oscillates within a small neighborhood of the buffer cap. Video rate adapts smoothly to congestion level shifts, and the TCP throughput utilization is close to the target value of $1 - \rho_v$.

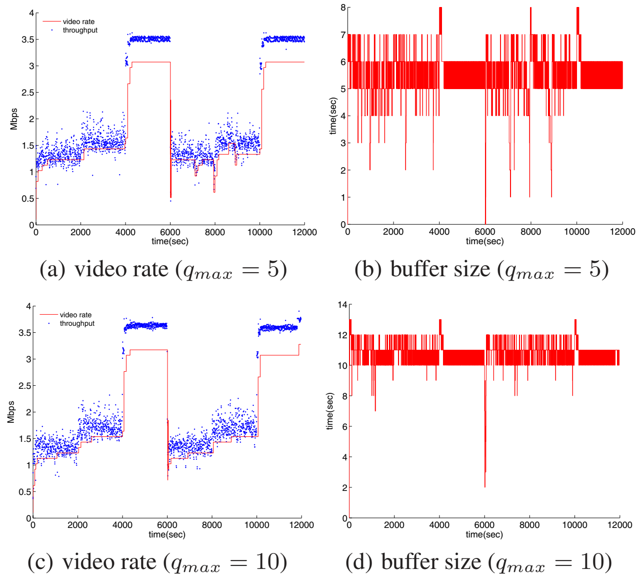
Table 2: Adaptation with Buffer Cap and Bandwidth Margin

margin ρ_v	average video rate(Mbps)	TCP throughput utilization (%)	average buffer size(second)
5%	1.99	91.98	19.79
10%	1.87	87.32	20.13
20%	1.6	76.43	20.98
30%	1.42	69.57	21.31

In live streaming, buffer cap q_{max} reflects the tolerable user playback lag. By setting q_{max} to a small value, we can simulate a live streaming session. Figure 12 shows the results for buffer cap of 5 seconds and 10 seconds. Table 3 shows the average video rate, utilization ratio and average buffer size. For both cases, the margin is set to 10%. We can see that when $q_{max} = 5$, there can be playback freezes when the background traffic suddenly jumps from $0.6Mbps$ to $3Mbps$ around $6,000$ second. Such a steep jump is a rare event in practice. When $q_{max} = 10$, the buffer never depletes.

Table 3: Live Streaming Simulation

buffer-cap second	avg. video rate(Mbps)	TCP throughput utilization (%)	avg. buffer size(second)
5	1.84	86.92	5.48
10	1.90	85.77	10.79



(a) video rate ($q_{max} = 5$) (b) buffer size ($q_{max} = 5$)

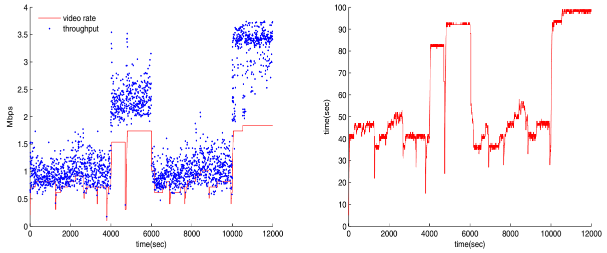
(c) video rate ($q_{max} = 10$) (d) buffer size ($q_{max} = 10$)

Figure 12: Simulation of live DASH streaming

5.4.4 Side-by-side Comparison

There have already been some client side video adaptation algorithms as introduced in Section 2. As a comparison, we implement the client side adaptation algorithm proposed in [17]. In [17], authors use $\mu = MSD/SFT$ as the transmission metrics to decide whether to switch up or switch down where MSD is the media segment duration and SFT is segment fetch time. If $\mu > 1 + \epsilon$, the chosen video rate will switch up one level, where $\epsilon = \max_i (b_{r_{i+1}} - b_{r_i})/b_{r_i}$ and b_{r_i} is the video rate of representation level i. If $\mu < \gamma_d$ where γ_d denotes switch down threshold, the chosen video rate switches down to the highest video rate level i that satisfies $b_{r_i} < \mu * b_c$ where b_c is the current video rate.

And also, to limit the maximum buffer size, idle time is added



(a) Video rate trend (b) Buffer size trend

Figure 13: Video rate and buffer size trend for [17]

between two consecutive chunk fetches. The idle time is calculated according to (12) where t_s, t_m and t_{min} denote the idle time, the buffered media time, the predefined minimum buffered media time respectively, b_c and b_{min} denote the current representation bitrate and the minimum representation bitrate respectively.

$$t_s = t_m - t_{min} - \frac{b_c}{b_{min}} MSD \tag{12}$$

We implemented the algorithm exactly as proposed in [17]. We did the same experiments as in our algorithm, the same curves are plotted to do side-by-side comparison with our algorithm. During the experiments, all configurations of the network including hardware, operating system and bandwidth settings between nodes are exactly the same as for our algorithm. Also, We added the same background traffic as shown in Figure 6.

The results are presented in Figure 13. By comparing with the results in Figure 9 and Figure 11, we can see that the average video rate in Figure 13 is much lower. This happens because the algorithm is too conservative and can't make full use of the available bandwidth. When the algorithm decides to switch up, it only switches up one level without considering the actual TCP throughput. Also, there are many video rate fluctuations because of the lack of smoothing mechanism in this algorithm. We can see that even one TCP throughput out-lier will cause the video rate to switch up or switch down.

To be more specific, Table 4 shows the detailed comparison of average video rate achieve by both algorithms at different congestion levels. During the whole experiments, background traffic consists of six stages with each stage of 2000 seconds, From Table 4, our algorithms achieve significantly higher video rates in a smoother fashion than the algorithm proposed in [17].

Table 4: Average Video Rates (Mbps) at different congestion levels

Algorithm Stage	m=1	m=5	m=20	Dyn m	[17]
0 ∼ 2000	1.30	1.28	1.17	1.24	0.75
2000 ∼ 4000	1.66	1.66	1.57	1.24	0.75
4000 ∼ 6000	3.56	3.44	3.32	3.44	1.62
6000 ∼ 8000	1.28	1.31	1.47	1.34	0.70
8000 ∼ 10000	1.64	1.72	1.60	1.61	0.84
10000 ∼ 12000	3.56	3.44	3.41	3.55	1.80

5.5 DASH with Multiple Servers

We set up four Emulab nodes as DASH servers, and conducted two types of multi-server DASH experiments.

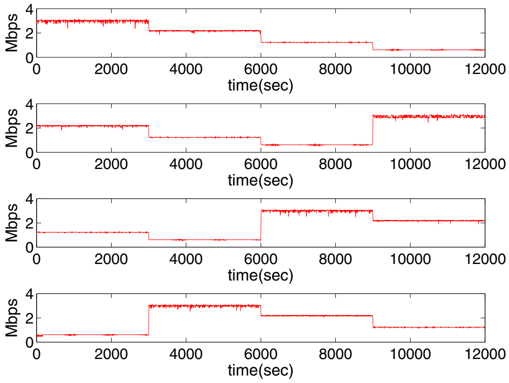


Figure 14: Background Traffic for Multiple Server DASH

5.5.1 *Dynamic Server Selection*

In the first experiment, we introduce background TCP traffic to each of the four servers. As illustrated in Figure 14, each server rotates between four background traffic levels, each level lasts for $3,000$ seconds. The whole experiment lasts for 200 minutes which is generally the length of an epic Hollywood movie. In the experiment, using SVR throughput prediction, the client is able to quickly detect traffic level shifts on all DASH servers, and dynamically switch to the server with the highest TCP throughput. As shown in Figure 15(a), the video rate is always maintained at a high level, independent of traffic level shifts on individual servers. There are several traffic rate dips. When we use a video rate margin of 10%, the rate smoothness is improved, as seen in Figure 15(b).

5.5.2 *Concurrent Download*

Figure 15(c) and Figure 15(d) show the experimental results when the client concurrently connects to two servers that offer the highest TCP throughput. To make a fair comparison with the single-connection case in the previous section, we reduce the link capacity and background traffic rate on all servers by roughly half. As shown in Figure 15(c), the client dynamically downloads chunks from the top-2 servers to sustain a high video rate throughout the experiments. The aggregate TCP throughput from top-2 servers has smaller variance than in the single dynamic server case, this leads to a much smoother video rate. The video rate can be further smoothed out when a bandwidth margin of 10% is added, as plotted in Figure 15(d). According to our experiment log, in both single connection and multiple connections cases, the switching of servers absolutely follows our prediction. This shows that SVR is accurate enough to monitor major congestion level shift and trigger DASH server switching.

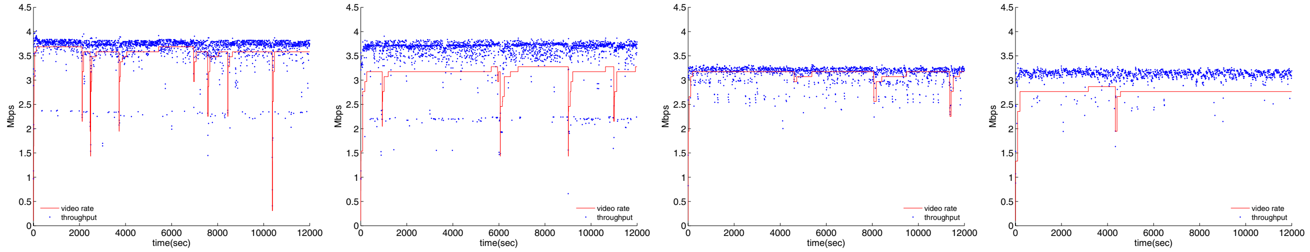
5.6 Internet Experiments

Finally, we test our DASH system on the Internet. We only do experiments for the single-server case.

5.6.1 *Client with Wireline Access*

We set up a Planetlab node in Shanghai, China ($planetlab-1.sjtu.edu.cn$) as the DASH server, and an Emulab node located in Utah, USA as the DASH client. We don't inject background traffic between the server and client.

Figure 16 and Table 5 show the results at different rate margins. Since different experiments are conducted sequentially, the TCP throughput patterns are not controllable. In all experiments, we do observed long-term shifts and short-term fluctuations of TCP throughput along the same Internet path. Our video adaptation al-



(a) video rate ($\rho_v = 0$) (b) video rate ($\rho_v = 10\%$) (c) video rate ($\rho_v = 0$) (d) video rate ($\rho_v = 10\%$)

Figure 15: Emulab Experiments with Multiple Servers: (a),(b): client only downloads from a single best server; (c),(d): client downloads concurrently from the top-2 servers.

gorithms can adapt well to the network condition changes, which is consistent with our Emulab results. In fact, the video rate is even smoother than our Emulab experiments. This is because the background traffic injected in Emulab experiments is more volatile than the real cross-traffic on the Internet path. Another difference from Emulab experiments is that when the rate margin is large e.g, $\rho_v = 30\%$, the achieved TCP throughput in Figure 16(d) is lower than with no or small margins. This is because in Emulab experiments, we control the total rate of background traffic. But in Internet experiments, background traffic is out of our control. When our DASH system is more conservative, the requested video chunks are smaller, which puts TCP at a disadvantage when competing with other cross-traffic.

Table 5: Wireline Internet Experiments

margin	average video rate(Mbps)	TCP throughput utilization (%)	average buffer size(second)
0%	3.34	97.80	32.99
5%	2.82	91.83	19.95
10%	2.55	86.32	20.50

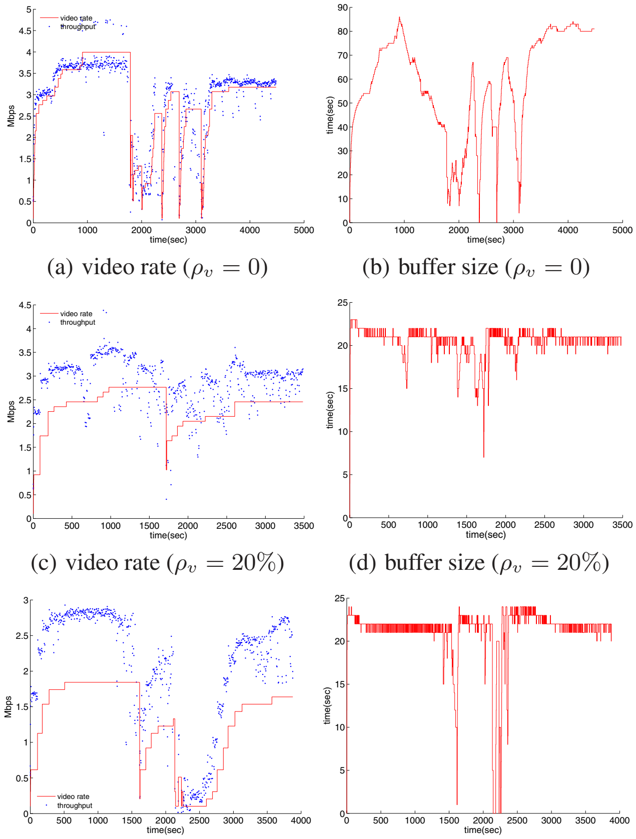
5.6.2 *Client with Wireless Access*

We also test our system on wireless clients. The server resides in Emulab, the client is a laptop in a residential network in New York City, which connects to Internet through a Wi-Fi router. we change the distance between the laptop and the router to create dynamic bandwidth while running the DASH client is running. Figure 17 and Table 6 show the results with different rate margins for the wireless client.

Table 6: Internet Experiments with Wireless Client

margin	average video rate(Mbps)	TCP throughput utilization (%)	average buffer size(second)
0%	2.80	97.24	57.83
20%	2.33	81.23	20.40
30%	1.27	63.44	21.11

When we take the laptop far away from the wireless router, the wireless connection is very unstable. TCP throughput varies over a wide range, and it is not repeatable across different experiments. Without a rate margin, the video rate also fluctuates a lot, and there are several video freezes. Increasing rate margin to 20% reduces video rate fluctuation and eliminates freezes. However, for the experiments with 30% margin in Figure 17(e) and Figure 17(f), the TCP throughput is extremely low from 1600 seconds to 2600 seconds, when the laptop was taken very far away from the router. The video rate is kept at $100Kbps$, and the video buffer depletes for many times. When the laptop is moved back closer to the router, the TCP throughput starts to increase, and the video rate switches up steadily.



(a) video rate ($\rho_v = 0$) (b) buffer size ($\rho_v = 0$)

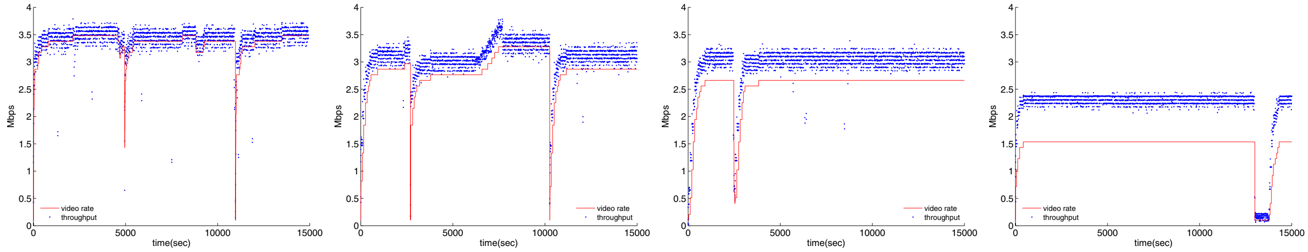
(c) video rate ($\rho_v = 20\%$) (d) buffer size ($\rho_v = 20\%$)

(e) video rate ($\rho_v = 30\%$) (f) buffer size ($\rho_v = 30\%$)

Figure 17: Internet Experiments with Wireless Client: comparison of different rate margins

6. CONCLUSIONS AND FUTURE WORK

In this paper, we developed video adaptation algorithms for DASH. Our algorithms use client-side buffered video time as feedback signal, and smoothly increase video rate as the available network band-



(a) video rate ($\rho_v = 0$)

(b) video rate ($\rho_v = 5\%$)

(c) video rate ($\rho_v = 10\%$)

(d) video rate ($\rho_v = 30\%$)

Figure 16: Internet Experiments with Wireline Client

width increases, and promptly reduce video rate in response to sudden congestion level shift-ups. By introducing buffer cap and small video rate margin, our algorithms simultaneously achieved stable video rate and buffer size. Using machine-learning based TCP throughput prediction, we extended our designs so that a DASH client simultaneously works with multiple servers to achieve the multiplexing gain among them. We conducted extensive controlled experiments on Emulab, as well as Internet experiments with wireline and wireless clients. We demonstrated that the proposed DASH system is highly responsive to congestion level shifts and can maintain stable video rate in face of short-term bandwidth variations.

Our wireless experiments demonstrated the unique challenges of delivering DASH in wireless networks. We are interested in refining our current adaptation algorithms for WiFi and 3G/4G cellular networks. In this paper, we mostly focus on the adaptation of single DASH client. When there are multiple DASH clients in the same service session, they compete with each other through the common TCP protocol at the transport layer and diverse video adaptation protocols at the application layer. We are interested in studying the fairness and stability of user video Quality-of-Experience among multiple competing DASH clients. Server-side DASH algorithms will be investigated to regulate the competition between them. Another direction is to investigate the application of DASH in P2P streaming systems. In P2P DASH, the video adaptation on a peer is not only driven by download throughput from the server, but also by upload/download throughput to/from other peers.

7. ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers and the Shepherd Dr. Yin Zhang for their valuable feedbacks and suggestions to improve the paper quality. The work is partially supported by USA NSF under contracts CNS-0953682, CNS-0916734, and CNS-1018032.

8. REFERENCES

[1] 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 *Proceedings of IEEE INFOCOM*, 2012.

[2] J. F. Kurose and K. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, 6th ed. Addison-Wesley, 2012.

[3] M. Watson, "Http adaptive streaming in practice," Netflix, Tech. Rep., 2011.

[4] S. Akhshabi, A. C. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http," in *ACM Multimedia Systems*, 2011.

[5] Microsoft, "IIS Smooth Streaming," http://www.iis.net/download/SmoothStreaming.

[6] Adobe, "Open Source Media Framework," http://www.osmf.org/.

[7] E. C. R. Mok, X. Luo and R. Chang, "Qdash: A qoe-aware dash system," in *ACM Multimedia Systems*, 2012.

[8] S. M. L. De Cicco and V. Palmisano, "Feedback control for adaptive live video streaming," in *ACM Multimedia Systems*, 2011.

[9] H. S. B. Haakon Riiser and P. Vigmostad, "A comparison of quality scheduling in commercial adaptive http streaming solutions on a 3g network," in *ACM Multimedia Systems*, 2012.

[10] S. L. Christopher Mueller and C. Timmerer, "An evaluation of dynamic adaptive streaming over http in vehicular environments," in *ACM Multimedia Systems*, 2012.

[11] W. C. B. Seo and R. Zimmermann, "Efficient video uploading from mobile devices in support of http streaming," in *ACM Multimedia Systems*, 2012.

[12] C. Dovrolis, M. Jain, and R. Prasad, "Measurement tools for the capacity and load of Internet paths," http://www.cc.gatech.edu/fac/Constantinos.Dovrolis/bw-est/.

[13] Q. He, C. Dovrolis, and M. Ammar, "On the predictability of large transfer tcp throughput," in *Proceedings of ACM SIGCOMM*, 2005.

[14] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A machine learning approach to tcp throughput prediction," in *ACM SIGMETRICS*, 2007.

[15] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, Aug. 2004.

[16] Emulab-Team, "Emulab - Network Emulation Testbed Home," http://www.emulab.net/.

[17] C. Liu, I. Bouazizi, and M. Gabbouj, "Rate adaptation for adaptive http streaming," in *ACM Multimedia Systems*, 2011.