Caching in HTTP Adaptive Streaming: Friend or Foe?

Danny H. Lee, Constantine Dovrolis
College of Computing
Georgia Institute of Technology
dannylee, constantine@gatech.edu

Ali C. Begen
Video and Content Platforms Research and Advanced Development
Cisco Systems
abegen@cisco.com

ABSTRACT

Video streaming is a major source of Internet traffic today and usage continues to grow at a rapid rate. To cope with this new and massive source of traffic, ISPs use methods such as caching to reduce the amount of traffic traversing their networks and serve customers better. However, the presence of a standard cache server in the video transfer path may result in bitrate oscillations and sudden rate changes for Dynamic Adaptive Streaming over HTTP (DASH) clients. In this paper, we investigate the interactions between a client and a cache that result in these problems, and propose an approach to solve it. By adaptively controlling the rate at which the client downloads video segments from the cache, we can ensure that clients will get smooth video. We verify our results using simulation and show that compared to a standard cache our approach (1) can reduce bitrate oscillations (2) prevents sudden rate changes, and compared to a no-cache scenario (3) provides traffic savings, and (4) improves the quality of experience of clients.

Keywords

Dynamic Adaptive Streaming over HTTP, MPEG DASH, Cache Servers, Traffic Shaping, Quality of Experience

General Terms

Performance, Evaluation, Algorithms

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

1. INTRODUCTION

Entertainment video is arguably the "killer application" of the Internet today, enabled by a combination of network infrastructure improvements, advances in processing power and modern video streaming protocols like Dynamic Adaptive Streaming over HTTP (DASH).

NOSSDAV '14 March 19-21 2014, Singapore, Singapore

ACM 978-1-4503-2706-0/14/03$15.00.
http://dx.doi.org/10.1145/2578260.2578270 .

DASH is an ISO standard that specifies manifest and media formats for adaptive streaming over the Internet. In DASH, the same media is encoded at different resolutions, bitrates and/or frame rates, called representations. Each representation is virtually or physically divided into chunks, called segments [5]. This allows high quality video to be served to clients over HTTP, using client logic to adaptively download segments based on available bandwidth. It has benefits like transparently crossing firewalls, a major problem that hampered adoption of prior UDP-based approaches. It also can be deployed on existing HTTP and CDN infrastructure, making it popular among content providers. For consumers, the proliferation of consumer electronic devices like game consoles, smart TVs and cellphones that support video streaming makes it easy to watch video content.

This popularity means video streaming is a huge source of traffic, and Cisco predicts that it will account for 69 percent of all consumer Internet traffic in 2017 [3]. Therefore, consumer access ISPs have an incentive to mitigate this load on their networks. One approach that ISPs may use is the deployment of cache servers within the access network to cache and store DASH segments. This allows clients to be served from the cache rather than retrieving every video segment from the origin server. Besides that, segments can be served to clients at higher speeds and with lower latency.

However, the client adaptation algorithm may overestimate the available path bandwidth when the requested segment is cached and served directly from the cache server. This overestimation may trigger the client to upshift to a higher bitrate representation. If the subsequent segment for the higher bitrate representation is not already cached, the cache will have to retrieve it from the upstream, potentially the origin server. The increased delivery delay will be observed by the client as a lower throughput. The client may then downshift to a lower bitrate representation in response to the reduction in the observed throughput. If the subsequent segment for the lower bitrate representation is cached, the observed throughput will increase leading to an overestimation again. This repeated cycle is known as *bitrate oscillation* and is highly undesirable, manifesting as switches between high and low bitrate video every few seconds. An extended period of oscillation depletes the playback buffer and causes the client algorithm to take drastic measures to refill it at the expense of video quality.

This paper aims to identify the conditions that can result in bitrate oscillations when a cache server exists in the path between the client and origin server. We show that oscillations are due to the erroneously high estimates introduced by a cache hit. We propose a cache-based solution that uses shaping to control the client download rate with an objective to avoid oscillations and keep the playback buffer full.

First, we describe how a typical DASH client operates, then we explain how the interactions with a cache server can cause oscillations and buffer drains. We then describe an algorithm to be implemented on a cache. We perform a discrete-time simulation comparing our solution with standard cache and no-cache configurations. We perform tests under both constant and fluctuating bandwidths, and show that our solution provides a better experience than both the standard and no-cache configurations.

The rest of this paper is organized as follows: In Section 2 we describe related work. Section 3 describes the problem and identifies the root cause. Section 4 describes our solution approach. Section 5 provides details about our simulation environment. Section 6 is an analysis of our results. Finally we have our conclusion and future work in Section 7.

2. RELATED WORK

The problematic interactions that occur when a cache server is introduced into a video streaming path is a relatively new topic, and the effects are still being explored. Mueller et al. [8] focus on the interactions of two or more clients through a proxy cache and aim to mitigate the negative effects of competition with a client-based solution. Our focus is on the simplest case where negative effects can occur with a single client and the presence of a cache. We aim to identify the most fundamental cause of oscillations, and solve it directly with a cache-based shaping solution.

Other authors have leveraged web distribution network infrastructure such as cache servers as a benefit instead of a problem source. Liu et al. [6] specifically use the distribution property of CDN servers to achieve higher video streaming rates by using parallel downloads from multiple different servers. In this case, they assume that the CDN network does the work of load balancing, ensuring that every request will result in a cache hit.

There has been work to modify cache servers to improve video scalability by co-locating video data near to clients [1], cache-based selection and prefetching of video streams based on popularity [10] and its application to RTSP [9]. Traffic shaping has been proposed to improve quality of experience by controlling requested bitrates from competing video streaming clients [4]. Although these papers were written in different contexts, they include several concepts that are fundamental to cache servers and shaping video streams that we would like to leverage in our work.

3. INTERACTIONS LEADING TO OSCILLATIONS

One DASH characteristic is that video segments are transferred over HTTP. This has the side effect that video segments are eligible to be stored in cache servers. Another property is that the quality of video playback is adaptive, based on a client algorithm. Our earlier study [2] explored the behavior of selection algorithms for various clients, and a common concept was identified. If the current transfer indicates the available bandwidth can support a higher representation bitrate, the client will use that higher bitrate when requesting the next segment. Similarly, when the bandwidth is insufficient, a lower representation bitrate will be used.

Cache servers are primarily deployed as a method of reducing upstream bandwidth. A typical cache server intercepts HTTP transfer requests coming from clients and checks if that file exists on the cache. If it exists and has not expired, it is served at the highest available speed, allowing lower latency and higher bandwidth than requests to the origin server. If the file does not exist or has expired, the cache server makes a request to the origin server and serves it to the client at a speed limited by the upstream bandwidth.

Now consider the scenario when a cache server exists in the path between the origin server and the client. We assume that the fair share available bandwidth between the origin server and the cache server C_s (**S**erver-cache) is lower than the bandwidth between the cache server and the client C_c (cache-**C**lient). This is a common situation, such as an access ISP cache server experiencing high traffic and multiple requests to a popular video content server, while the path to the customer watching the video is uncongested.

For a completely uncached video, clients should not experience any bitrate oscillations from the presence of the cache server. The cache server will act as a pass through, retrieving segments from the origin server through the bottleneck link. As playback progresses, the client algorithm will converge to download segments at the representation bitrate supported by C_s. By the end of playback, the cache will be populated with sequential segments belonging to that representation. Now let us examine the situation that results in

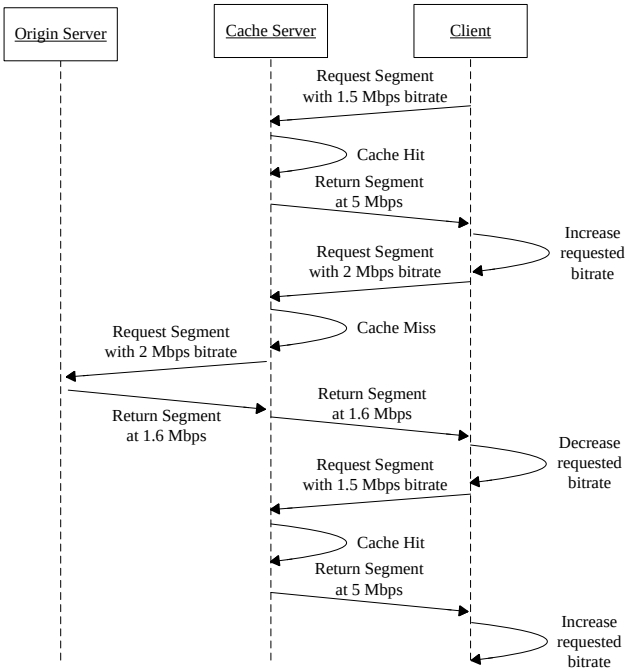


Figure 1: Bitrate Oscillation Sequence Diagram

bitrate oscillations. Let B_f be a representation bitrate of a segment, B_{f+1} be the next higher bitrate, B_{max} be the maximum bitrate. Consider following conditions:

1. Segments with representation bitrate B_f are cached
2. $B_f < B_{max}$
3. $B_f < C_s < B_{f+1} < C_c$

When the client requests segments at bitrate B_f, they are delivered from the cache at the maximum rate. Therefore the client will use the higher rate C_c to estimate the representation bitrate for the next segment and select B_{f+1}. However, since the higher bitrate segment is uncached, the request will be served at the lower rate C_s. As C_s is unsustainable for the representation bitrate B_{f+1}, the streaming algorithm will choose a lower bitrate B_f for the next segment. This request results in a cache hit and starts the cycle again. One oscillation is shown in Figure 1, where C_s = 1.6 Mbps, C_c = 5 Mbps, B_f = 1.5 Mbps, and B_{f+1} = 2 Mbps.

Another consequence of selecting a high bitrate segment based on an erroneous bandwidth reading is that transfers will be slower than real time, as $\frac{B_{f+1}}{C_s} > 1$. The client maintains a playback buffer to hold downloaded segments, and the download process needs to be at least as fast as the playback rate, or the playback buffer will eventually drain. In some clients, the selection algorithm will resort to a sudden reduction in bitrates so the download process can refill the buffer as soon as possible. In the worst case, playback can pause if the buffer is empty.

We believe that this oscillation scenario can occur independent of the cache replacement policy, as we are dealing with a single client and video. If we expand the number of clients and videos, cache replacement policy and disk sizes may become a factor, but for our current scenario, we assume that the cache server has sufficient disk capacity to hold all representations of a video.

4. THE VISIC CACHE

In this section we describe our approach to create a video-aware cache server, and implementation issues that need to be considered. We term our solution ViSIC, for **Vi**deo **S**haping **I**ntelligent **C**ache.

4.1 Monitoring Path Bandwidths

We collect path bandwidth measurements as inputs to our shaping algorithm, monitoring transfers to origin servers and clients. There are other methods of collecting available bandwidths in DASH transfers [7], but we wish to leverage the unique properties of cache servers for our measurements.

Cache servers typically run on powerful hardware and serve a large number of clients, and also service both video and standard HTTP requests. This means that bandwidth measurements can be derived from multiple sources, not just a single video stream. In addition to taking point measurements, we use an exponential moving average in our algorithm to help reduce the impact of fluctuations. Let $A(t)$ be the instantaneous rate of the segment at time t, and the averaged bandwidth $\hat{A}$ is calculated in (1).

$$\hat{A} = \begin{cases} (1-\alpha)\hat{A}(t-1) + \alpha A(t) & \text{if } t > 0 \\ A(t) & \text{if } t = 0 \end{cases} \quad (1)$$

This is calculated from the cache server to both clients and origin servers, producing $\hat{A}_s$ for the averaged server bandwidth and $\hat{A}_c$ for client bandwidth. We maintain the last T samples of these values, taken once every second, for identifying long duration changes in bandwidth. We set α to 0.1 to compute the weighted average to reduce the impact of temporary fluctuations and T to 15 seconds to have a long history of measurements to detect changes.

4.2 Shaping of Traffic from the Cache

As we identified in Section 3, the main problem of oscillations results from the false bandwidth readings from segments served from the cache. To prevent this, we propose a cache-based algorithm that uses traffic shaping to ensure clients will not request segments that exceed the path bandwidth.

Our secondary objective is to provide a high quality of experience with as few bitrate transitions as possible. This means the shaping algorithm should be resistant to temporary bandwidth spikes and dips. When the bottleneck lies between the origin server and cache server, our algorithm can compensate for both bandwidth spikes and dips, but if the bottleneck is between the cache server and client, shaping can only compensate for bandwidth spikes.

The first part of the algorithm is to find a target bitrate with index ϕ_{target} that we want the client to use as a result of shaping. Let the set of representation bitrates be B sorted in ascending order, with each element $B_i \in B$ having an index ϕ. Let B_{req} be the bitrate requested by the client with index ϕ_{req}. We calculate candidate bitrates based on the averaged bandwidths: B_{s1} at index ϕ_{s1} is the maximum bitrate supported by the instantaneous transfer rate A_s, B_{s2} at index ϕ_{s2} is the maximum bitrate supported by $\hat{A}_s$. B_{c1} at index ϕ_{c1} and B_{c2} at index ϕ_{c2} are defined similarly for client transfer rates A_c and $\hat{A}_c$, respectively. This can be expressed using the formulae in (2).

$$\begin{aligned} \phi_{s1} &= \underset{i}{\operatorname{argmax}}\{B_i < A_s\} \\ \phi_{s2} &= \underset{i}{\operatorname{argmax}}\{B_i < \hat{A}_s\} \end{aligned} \quad (2)$$

We define three conditions for detecting long duration bandwidth changes: $increaseBW_s$, $increaseBW_c$ and $decreaseBW_s$. $increaseBW_s$ is true if (1) $\phi_{s1} > \phi_{req}$ and (2) the last T samples of $\hat{A}_s$ are greater than B_{req}. This indicates a bandwidth increase if the average bandwidth over the last T seconds is consistently higher than the requested bitrate. The condition $increaseBW_c$ is defined similarly using the client values ϕ_{c1} and $\hat{A}_c$. $decreaseBW_s$ is true if (1) $\phi_{s1} < \phi_{req}$ and (2) the last T samples of $\hat{A}_s$ are smaller than B_{req}, indicating that the client is requesting a bitrate that is too high. We set T to 15 seconds, to limit the number of bitrate changes per minute to four.

Using the above information, we implement the shaping as described in Algorithm 1. The main case we want to handle is when $\hat{A}_s \leq \hat{A}_c$ and a segment is cached (lines 3 to 7): we choose between ϕ_{req} or ϕ_{s1} for the target rate. We use ϕ_{s1} if A_s can support a higher bitrate, otherwise we keep the client at the same bitrate. Note that a bandwidth decrease is treated the same as steady bandwidth, as we want to leverage the chance that the next segment is also cached. In the case where the requested segment is not cached, we smooth out temporary fluctuations by only reacting to long duration increases and decreases (lines 9 to 13).

If $\hat{A}_s > \hat{A}_c$, we can only compensate for transient bandwidth increases (lines 16 to 20), and use the candidate index derived from client measurements. Note that the available bandwidth C_c may be lower than the shaped rate, and may be an external limiting factor.

Once ϕ_{target} is decided, simply using the corresponding bitrate B_{target} directly for shaping is not ideal, as it does not maximize a high bandwidth path from cache server to client. Therefore in lines 22 and 23 we derive a target shaping rate R_{target}, which is the next higher representation bitrate multiplied by β, a constant less than 1. The objective is to serve segments at a higher rate than B_{target}, but lower than a rate that will cause the client to upshift. For the simulation, we set β to 0.9 to maximize the bandwidth.

4.3 Determining Representation Bitrates

The key to our approach is for the cache server to recognize client requests for segments with specific representation bitrates, and shape eligible transfers. This information can be directly read from the Media Presentation Description (MPD) file as part of the setup phase of a DASH stream, or by inference methods. A ViSIC implementation can use the following ways to access this information depending on the situation:

1. Directly reading the MPD file:
   If the content provider uses a plaintext MPD file sent over an unencrypted channel, the file can be read directly by the cache server before it is sent to the client.
2. Duplicating the MPD request:
   If the client requests the MPD file over an encrypted channel like HTTPS, the cache server can have additional logic to identify this event and duplicate the request. One approach

Algorithm 1 ViSIC shaping algorithm

if $\hat{A}_s \leq \hat{A}_c$ **then**
  if $isCached(Segment) = True$ **then**
    if $increaseBW_s$ **then**
      $\phi_{target} \leftarrow \phi_{s1}$
    else
      $\phi_{target} \leftarrow \phi_{req}$
    end if
  else
    if $increaseBW_s$ or $decreaseBW_s$ **then**
      $\phi_{target} \leftarrow \phi_{s1}$
    else
      $\phi_{target} \leftarrow \phi_{req}$
    end if
  end if
else
  if $increaseBW_c$ **then**
    $\phi_{target} \leftarrow \phi_{c1}$
  else
    $\phi_{target} \leftarrow \phi_{req}$
  end if
end if
$j \leftarrow \phi_{target} + 1$ {Choose the next higher bitrate}
$R_{target} \leftarrow \beta * B_j$ {Reduce the rate slightly for shaping}

is by masquerading as a client and requesting a new stream to get the MPD.

3. HTTP transfer inference:
   DASH transfers follow certain characteristics, such as requesting successive segment files at regular intervals in steady state. This knowledge can be used to infer information about video segments. E.g., file names can be directly read from HTTP request headers, and patterns in the naming of files can be used to get the segment sequence. The bitrate of each segment can be estimated by dividing file size by segment duration.

5. SIMULATION

We created a discrete-time simulator written in python to validate our results. This simulator includes clients, servers and caches, and we describe their behavior in this section.

5.1 Simple Client

The simplified player consists of a download process, a playback process and a playback buffer. The download process fills the buffer with segments and the playback process consumes segments in the buffer. Each segment contains τ seconds of video, and the buffer can hold up to 30 seconds of segments. Let $buffer(t)$ be the amount of buffered seconds at time t, and θ be the low buffer threshold. The playback process starts when the playback buffer is full, and consumes segments in real time. Playback pauses when the buffer is empty and resumes when the buffer exceeds θ. We set τ to two and θ to 10 seconds.

The download process operates in two states: buffering state and steady state. In buffering state, it downloads segments one after the other without delay. In steady state it retrieves a segment once every τ seconds. The client starts in buffering state, and switches to steady state when the playback buffer is full. For each segment, an algorithm is used to select representation bitrates based on throughput measurements and current buffer size. Throughput measurements use the exponential moving average $\hat{A}$ based on instantaneous throughput $A(t)$, where t is the current time. This is shown in (3). For our simulation, δ is set to 0.2 to compute the weighted average towards historical values.

$$\hat{A} = \begin{cases} (1-\delta)\hat{A}(t-1) + \delta A(t) & \text{if } t > 0 \\ A(t) & \text{if } t = 0 \end{cases} \quad (3)$$

Let B_i be an element in the set of representation bitrates sorted in ascending order, where i is the index in the set. The client determines two candidate bitrate indices ϕ_1 and ϕ_2 using the formulae in (4). We use $c = 0.9$ to leave a buffer so small fluctuations in bandwidth do not cause the client to switch bitrates.

$$\begin{aligned} \phi_1 &= \underset{i}{\text{argmax}}\{B_i < c * A\} \\ \phi_2 &= \underset{i}{\text{argmax}}\{B_i < c * \hat{A}\} \end{aligned} \quad (4)$$

Let B_f be the current representation bitrate, with index ϕ_f. At the start of the simulation, B_f is initialized to B_0. During runtime, if $buffer(t) < \theta$ and $\phi_1 < \phi_f$, 'Panic Mode' is triggered. The client goes into buffering state and sets B_f to B_0, aiming to refill the buffer as soon as possible. Algorithm 2 describes the bitrate selection process.

Algorithm 2 Client segment bitrate selection

if $buffer(t) > \theta$ **then**
  if $\phi_2 < \phi_f$ and $\phi_1 < \phi_f$ and $\phi_f > 0$ **then**
    Decrease B_f to the next lower bitrate
  else if $\phi_2 > \phi_f$ and $\phi_1 > \phi_f$ and $\phi_f < \phi_{max}$ **then**
    Increase B_f to the next higher bitrate
  end if
else if $buffer(t) \leq \theta$ and $\phi_1 < \phi_f$ {'Panic Mode'} **then**
  $B_f \leftarrow B_0$
  $bufferingMode \leftarrow True$
end if

5.2 Origin Server

The origin server holds segments that can be requested by the client, stored internally as a list of file names. When a request is made to the server, it will check if the file exists before accepting the transfer. For our simulation, we assume that the origin server holds all the possible representations and their segments specified in the MPD.

5.3 Standard Cache

In our simulation we model the cache servers as cut-through, that is, uncached requests do not need to be fully downloaded by the cache server before they are served to clients. The standard cache operates as follows: it intercepts requests and checks if the file exists on disk. If it exists, the file will be served out at the cache-client bandwidth C_c. If not, a new transfer from the origin server will be started and the file will be served to the client with an increased delay accounting for the path between the cache and origin server. If the bottleneck link lies in the path to the origin server, the overall effective transfer rate will be C_s.

6. EVALUATION

6.1 Test Setup

For our evaluation of ViSIC, we implemented the network depicted in Figure 2 with a single origin server, cache server and client connected by links with variable bandwidths. The cache servers that we evaluated are a standard cache and a ViSIC cache.

Figure 2: Simulation Topology

We also evaluated the no-cache scenario. For all simulations, we used five representation bitrates: 256 Kbps, 768 Kbps, 1.5 Mbps, 2.8 Mbps, 4.5 Mbps. The cache servers were populated with 1.5 Mbps segments. We varied the simulation bandwidths for each test to compare the performance (1) under constant bandwidth and (2) under varying bandwidth. All experiments were performed on a Windows 7 Home Premium 64-bit host running on an Intel®Core™i5-3320M 2.6 GHz processor with 16 GB of RAM. The python version used was 2.7.3.

6.2 Constant Bandwidth

In this test, we wanted to verify that clients would experience bitrate oscillation under the situation described in Section 3, and that ViSIC could avoid that. We set the bottleneck bandwidth C_s to a constant 2 Mbps and C_c to 5 Mbps. In Figure 3, we can see

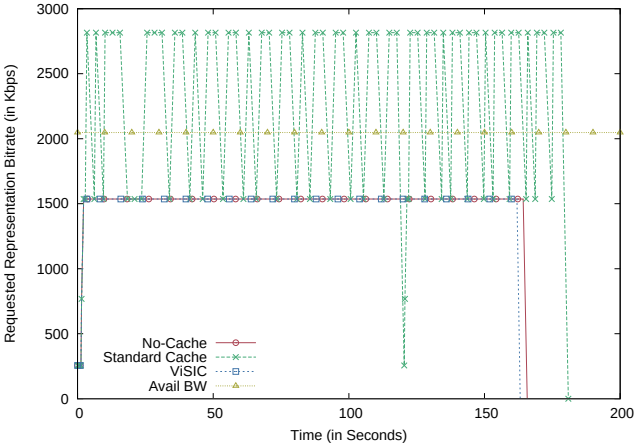


Figure 3: Bitrates Requested by the Client

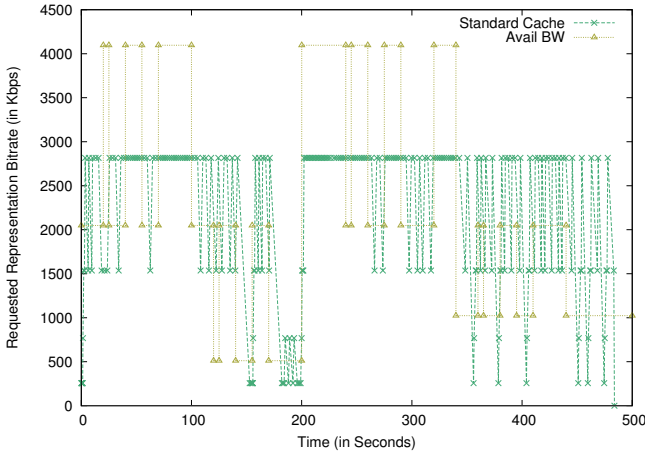
that for a standard cache with the 1.5 Mbps segment cached, the download occurs at a high rate causing the client to switch up to use the 2.8 Mbps bitrate for the next segment, triggering a cache miss. This repeats multiple times, resulting in almost constant bitrate oscillations.

The problem of buffer draining manifests in the standard cache test at 120 s: There is a sudden drop from the 2.8 Mbps to the 256 Kbps bitrate due to the client going into 'Panic Mode'. The no-cache and ViSIC cache avoid this disruptive behavior completely and stay at a constant 1.5 Mbps representation.

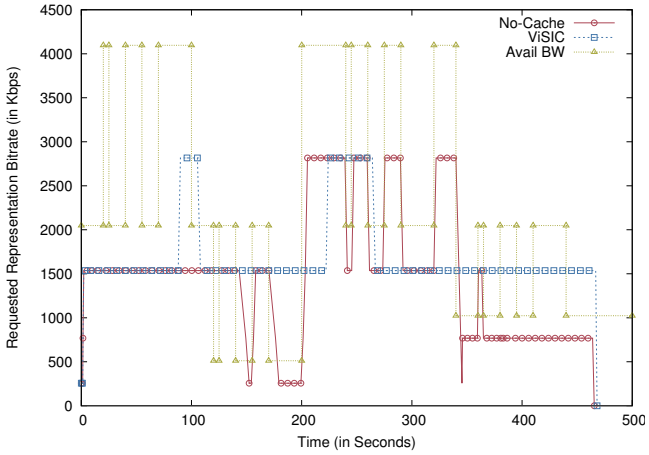
6.3 Variable Bandwidth

In this test, we wanted to evaluate the performance of ViSIC against the no-cache and standard cache under varying bandwidths. Keeping C_c constant at 5 Mbps, we varied C_s to produce the graphs shown in Figure 4. From 0 to 200 s we test the effect of bandwidth spikes and dips when the client retrieves cached segments, with varying durations of 5, 15 and 30 seconds. ViSIC causes the client to increase bitrate only if the bandwidth increase is longer than 15 seconds, and ignores all bandwidth dips.

From 200 to 320 s we test how bandwidth decreases affect non-cached segments by raising the available bandwidth to 4 Mbps and then reducing it to 2 Mbps to trigger a cache hit. ViSIC



(a) Avail BW and Standard Cache



(b) Avail BW, No-Cache and ViSIC

Figure 4: Available BW and Bitrates Requested

shaping causes the client to request uncached segments if the available bandwidth is high, but reverts to shaping to cached segments when the bandwidth decrease is longer than 15 seconds.

The last section from 320 s onwards shows the behavior of clients requesting uncached segments in the presence of bandwidth spikes. ViSIC's algorithm causes the client to retrieve cached segments even if C_s is very low.

As can be seen from the graphs in Figure 4(a), standard cache causes the client to start oscillating when it serves cached segments, and reacts quickly to changes in bandwidth, resulting in sudden bitrate changes. In Figure 4(b), we can see that in the no-cache scenario, the client also reacts quickly to drops in bandwidth. We discuss this reaction in the following section using metrics.

6.3.1 Bitrate Stability

We define the instability metric as the ratio of consecutive bitrate changes over the number of segments retrieved, and calculated this for every five segments in the video. A higher instability metric indicates a poorer quality of experience for the viewer, manifesting as sudden bitrate increases or drops.

The results are shown in Figure 5. The average instability for ViSIC is lower than the no-cache and standard cache scenarios. ViSIC has a maximum instability of 0.4 at the start of the simulation, and can be explained by the client successively increasing bitrates during initial buffering. Following that, there

are no other major bitrate changes, only single level transitions of metric 0.2. The no-cache scenario performs poorly in reaction to sudden changes in available bandwidth, and the standard cache constantly oscillates, resulting in higher instability.

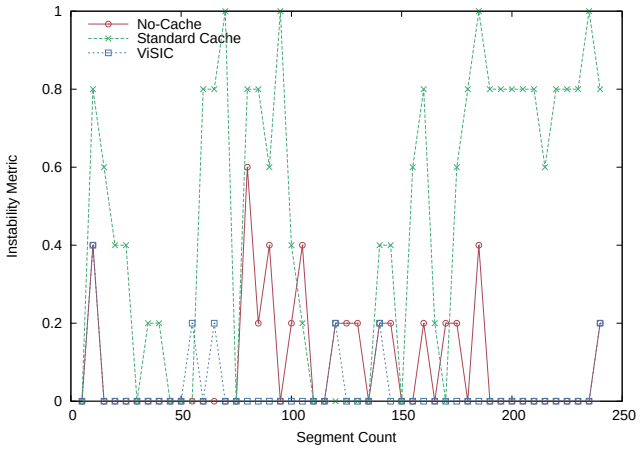


Figure 5: Instability Metrics

6.3.2 *Buffer Fullness*

Another quality of experience measurement is how the buffer level varies with time. Ideally, the initial buffering stage of video playback should be as short as possible, and the buffer should be close to full over the course of playback. We examine the buffer levels in Figure 6: ViSIC fills the playback buffer and starts playback at 11.9 s, compared to no-cache at 14.2 s and standard cache at 15.6 s.

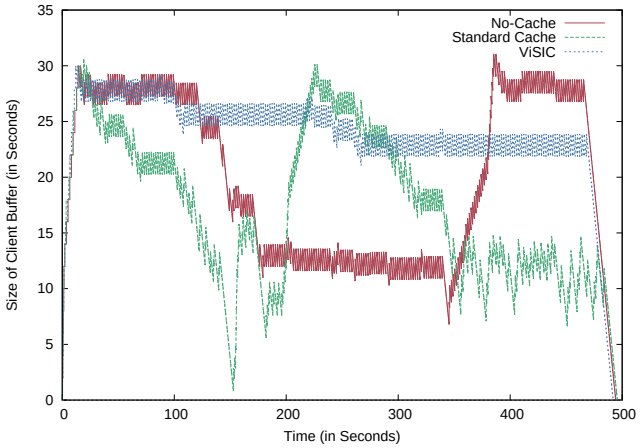


Figure 6: Playback Buffer Levels[1]

Besides preventing 'Panic Mode' in clients, a full playback buffer also allows the viewer to seek forward or backward in the buffered range. Throughout the simulation, ViSIC keeps the playback buffer near full regardless of bandwidth changes. The no-cache case drains the buffer in the presence of drops in available bandwidth at 180 and 320 s. The standard cache performs the worst, going into 'Panic Mode' at 120, 180 and five more times between 330 and 480 s.

[1]Due to data density, markers have been omitted. Better in color.

7. CONCLUSIONS AND FUTURE WORK

In this paper we described how the introduction of a cache server can result in bitrate oscillations in DASH systems. We identified that cache hits cause erroneous bandwidth readings in clients, which may then request uncached higher quality segments and result in oscillations. We also found that oscillations drain the playback buffer, causing playback disruptions as clients attempt to refill the buffer. We introduced ViSIC, a cache-based approach that uses shaping to eliminate oscillations. We used simulations to prove ViSIC has higher stability and buffer fullness than standard cache and no-cache configurations under different bandwidth scenarios. ViSIC retains cache server benefits like reducing upstream traffic and serving segments at high bandwidth.

For future work, we are in the process of conducting experiments on a testbed setup using real clients and modified Squid cache server software. We also plan to study the application of the shaping algorithm when multiple clients with different C_c values watch the same video through a cache server. Scalability is another concern, as the algorithm needs to be able to scale to monitor servers in the order of hundreds and clients in thousands. Another area for future work is to compare DASH-aware cache replacement policies in relation to cache server throughput and latency.

8. ACKNOWLEDGMENTS

The authors are grateful to Ashok Narayanan and Saamer Akhshabi. Ashok described the instability problem at the Adaptive Media Transport Workshop, organized by Cisco, in June 2012. Saamer provided guidance and contributed to discussions that led to this paper.

9. REFERENCES

[1] S. Acharya and B. Smith. Middleman: A video caching proxy server. In *Proceedings of ACM NOSSDAV*, 2000.

[2] S. Akhshabi, A. C. Begen, and C. Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of ACM MMSys*, 2011.

[3] Cisco Systems Inc. Cisco Visual Networking Index: Forecast and Methodology, 2012 - 2017. Technical report, May 2013.

[4] R. Houdaille and S. Gouache. Shaping HTTP adaptive streams for a better user experience. In *Proceedings of ACM MMSys*, 2012.

[5] ISO Standard: Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats, 2012.

[6] C. Liu, I. Bouazizi, M. M. Hannuksela, and M. Gabbouj. Rate adaptation for dynamic adaptive streaming over HTTP in content distribution network. *Signal Processing: Image Communication*, 27(4):288–311, 2012.

[7] R. K. P. Mok, X. Luo, E. W. W. Chan, and R. K. C. Chang. QDASH: A QoE-aware DASH system. In *Proceedings of ACM MMSys*, 2012.

[8] C. Mueller, S. Lederer, and C. Timmerer. A proxy effect analysis and fair adaptation algorithm for multiple competing dynamic adaptive streaming over HTTP clients. In *Proceedings of VCIP*, 2012.

[9] R. Rejaie and J. Kangasharju. Mocha: A quality adaptive multimedia proxy cache for internet streaming. In *Proceedings of ACM NOSSDAV*, 2001.

[10] R. Rejaie, H. Yu, M. Handley, and D. Estrin. Multimedia proxy caching mechanism for quality adaptive streaming applications in the Internet. In *Proceedings of IEEE INFOCOM*, 2000.