An Experimental Investigation of the Akamai Adaptive Video Streaming

Luca De Cicco and Saverio Mascolo
ldecicco@gmail.com, mascolo@poliba.it

Dipatimento di Elettrotecnica ed Elettronica, Politecnico di Bari,
Via Orabona n.4, Bari, Italy

Abstract. Akamai offers the largest Content Delivery Network (CDN) service in the world. Building upon its CDN, it recently started to offer High Definition (HD) video distribution using HTTP-based adaptive video streaming. In this paper we experimentally investigate the performance of this new Akamai service aiming at measuring how fast the video quality tracks the Internet available bandwidth and to what extent the service is able to ensure continuous video distribution in the presence of abrupt changes of available bandwidth. Moreover, we provide details on the client-server protocol employed by Akamai to implement the quality adaptation algorithm. Main results are: 1) any video is encoded at five different bit rates and each level is stored at the server; 2) the video client computes the available bandwidth and sends a feedback signal to the server that selects the video at the bitrate that matches the available bandwidth; 3) the video bitrate matches the available bandwidth in roughly 150 seconds; 4) a feedback control law is employed to ensure that the player buffer length tracks a desired buffer length; 5) when an abrupt variation of the available bandwidth occurs, the suitable video level is selected after roughly 14 seconds and the video reproduction is affected by short interruptions.

1 Introduction and related works

Nowadays the Internet, that was originally designed to transport delay-insensitive data traffic, is becoming the most important platform to deliver audio/video delay-sensitive traffic. Important applications that feed this trend are YouTube, which delivers user-generated video content, and Skype audio/video conference over IP. In this paper we focus on adaptive (live) streaming that represents an advancement *wrt* classic progressive download streaming such as the one employed by YouTube. With download streaming, the video is a static file that is delivered as any data file using greedy TCP connections. The receiver employs a player buffer that allows the file to be stored in advance *wrt* the playing time in order to mitigate video interruptions. With adaptive streaming, the video source is adapted on-the-fly to the network available bandwidth. This represents a key advancement *wrt* classic download streaming for the following reasons: 1) live video content can be delivered in real-time; 2) the video quality can be continuously adapted to the network available bandwidth so that users can watch

videos at the maximum bit rate that is allowed by the time-varying available bandwidth.

In [8] the authors develop analytic performance models to assess the performance of TCP when used to transport video streaming. The results suggest that in order to achieve good performance, TCP requires a network bandwidth that is two times the video bit rate. This bandwidth over provisioning would systematically waste half of the available bandwidth.

In a recent paper [6] the authors provide an evaluation of TCP streaming using an adaptive encoding based on H.264/SVC. In particular, the authors propose to throttle the GOP length in order to adapt the bitrate of the encoder to the network available bandwidth. Three different rate-control algorithms for adaptive video encoding are investigated. The results indicate that the considered algorithms perform well in terms of video quality and timely delivery both in the case of under-provisioned links and in the case of competing TCP flows.

In this paper we investigate the adaptive streaming service provided by Akamai, which is the worldwide leading Content Delivery Network (CDN). The service is called High Definition Video Streaming and aims at delivering HD videos over Internet connections using the Akamai CDN. The Akamai system is based on the *stream-switching* technique: the server encodes the video content at different bit rates and it switches from one video version to another based on client feedbacks such as the measured available bandwidth. It can be said that the Akamai approach is the leading commercial one since, as we will see shortly, it is employed by the Apple HTTP-based streaming, the Microsoft IIS server, the Adobe Dynamic Streaming, and Move Networks. By encoding the same video at different bitrates it is possible to overcome the scalability issues due to the processing resources required to perform multiple on-the-fly encoding at the price of increasing storage resources. HTTP-based streaming is cheaper to deploy since it employs standard HTTP servers and does not require specialized servers at each node.

In the following we summarize the main features of the leading adaptive streaming commercial products available in the market.

IIS Smooth Streaming [9] is a live adaptive streaming service provided by Microsoft. The streaming technology is offered as a web-based solution requiring the installation of a plug-in that is available for Windows and iPhone OS 3.0. The streaming technology is codec agnostic. IIS Smooth Streaming employs stream-switching approach with different versions encoded with configurable bitrates and video resolutions up to 1080p. In the default configuration IIS Smooth Streaming encodes the video stream in seven layers that range from 300 kbps up to 2.4 Mbps.

Adobe Dynamic Streaming [4] is a web-based adaptive streaming service developed by Adobe that is available to all devices running a browser with Adobe Flash plug-in. The server stores different streams of varying quality and size and switches among them during the playback, adapting to user bandwidth and CPU. The service is provided using the RTMP streaming protocol [5]. The supported video codecs are H.264 and VP6 which are included in the Adobe Flash

plug-in. The advantage of Adobe's solution is represented by the wide availability of Adobe Flash plug-in at the client side.

Apple has recently released a client-side *HTTP Adaptive Live Streaming* solution [7]. The server segments the video content into several pieces with configurable duration and video quality. The server exposes a playlist (.m3u8) containing all the available video segments. The client downloads consecutive video segments and it dynamically chooses the video quality employing an undisclosed proprietary algorithm. Apple HTTP Live Streaming employs H.264 codec using a MPEG-2 TS container and it is available on any device running iPhone OS 3.0 or later (including iPad), or any computer with QuickTime X or later installed.

Move Networks provides live adaptive streaming service [2] to several TV networks such as ABC, FOX, Televisa, ESPN and others. A plug-in, available for the most used web browsers for Windows and Mac OS X, has to be installed to access the service. Move Networks employs VP7, a video codec developed by On2, a company that has been recently acquired by Google. Adaptivity to available bandwidth is provided using the stream-switching approach. Five different versions of the same video are available at the server with bitrates ranging from 100 kbps up to 2200 kbps.

The rest of the paper is organized as follows: Section 2 describes the testbed employed in the experimental evaluation; in Section 3 we show the main features of the client-server protocol used by Akamai in order to implement the quality adaptation algorithm; Section 4 provides a discussion of the obtained results along with an investigation of the dynamics of the quality adaptation algorithm; finally, Section 5 draws the conclusions of the paper.

2 Testbed and experimental scenarios

The experimental evaluation of Akamai HD video server has been carried out by employing the testbed shown in Figure 1. Akamai HD Video Server provides a number of videos made available through a demo website [1]. In the experiments we have employed the video sequence "*Elephant's Dream*" since its duration is long enough for a careful experimental evaluation. The receiving host is an Ubuntu Linux machine running 2.6.32 kernel equipped with NetEm, a kernel module that, along with the traffic control tools available on Linux kernels, allows downlink channel bandwidth and delays to be set. In order to perform traffic shaping on the downlink we used the Intermediate Functional Block pseudo-device IFB[1].

The receiving host was connected to the Internet through our campus wired connection. It is worth to notice that before each experiment we carefully checked that the available bandwidth was well above 5 Mbps that is the maximum value of the bandwidth we set in the traffic shaper. The measured RTT between our client and the Akamai server is of the order of 10 ms. All the measurements we report in the paper have been performed after the traffic shaper (as shown in

[1] http://www.linuxfoundation.org/collaborate/workgroups/networking/ifb

Figure 1) and collected by dumping the traffic on the receiving host employing `tcpdump`. The dump files have been post-processed and parsed using a Python script in order to obtain the figures shown in Section 4.

The receiving host runs an `iperf` server (TCP Receiver) in order to receive TCP greedy flows sent by an `iperf` client (TCP Sender).

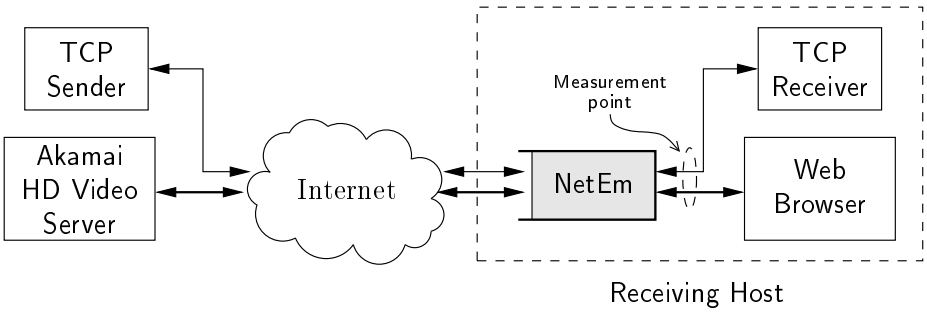


Fig. 1: Testbed employed in the experimental evaluation

Three different scenarios have been considered in order to investigate the dynamic behaviour of Akamai quality adaptation algorithm:

1. Akamai video flow over a bottleneck link whose bandwidth capacity changes following a step function with minimum value 400 kbps and maximum value 4000 kbps;
2. Akamai video flow over a bottleneck link whose bandwidth capacity varies as a square wave with a period of 200 s, a minimum value of 400 kbps and a maximum value of 4000 kbps;
3. Akamai video flow sharing a bottleneck, whose capacity is fixed to 4000 kbps, with one concurrent TCP flow.

In scenarios 1 and 2 abrupt variations of the available bandwidth occur: even though we acknowledge that such abrupt variations are not frequent in real-world scenarios, we stress that step-like variations of the input signal are often employed in control theory to evaluate the key features of a dynamic system response to an external input [3]. The third scenario is a common use-case designed to evaluate the dynamic behaviour of an Akamai video flow when it shares the bottleneck with a greedy TCP flow, such as in the case of a file download. In particular, we are interested in assessing if Akamai is able to grab the fair share in such scenarios.

3 Client-server Quality Adaptation Protocol

Before discussing the dynamic behaviour of the quality adaptation algorithm employed by Akamai, we focus on the client-server protocol used in order to implement this algorithm.

To this purpose, we analyzed the dump file captured with `tcpdump` and we observed two main facts: 1) The Akamai server employs TCP in order to transport the video flows and 2) a number of HTTP requests are sent from the client to the server throughout all the duration of the video streaming. Figure 2 shows the time sequence graph of the HTTP requests sent from the client to the Akamai server reconstructed from the dump file.

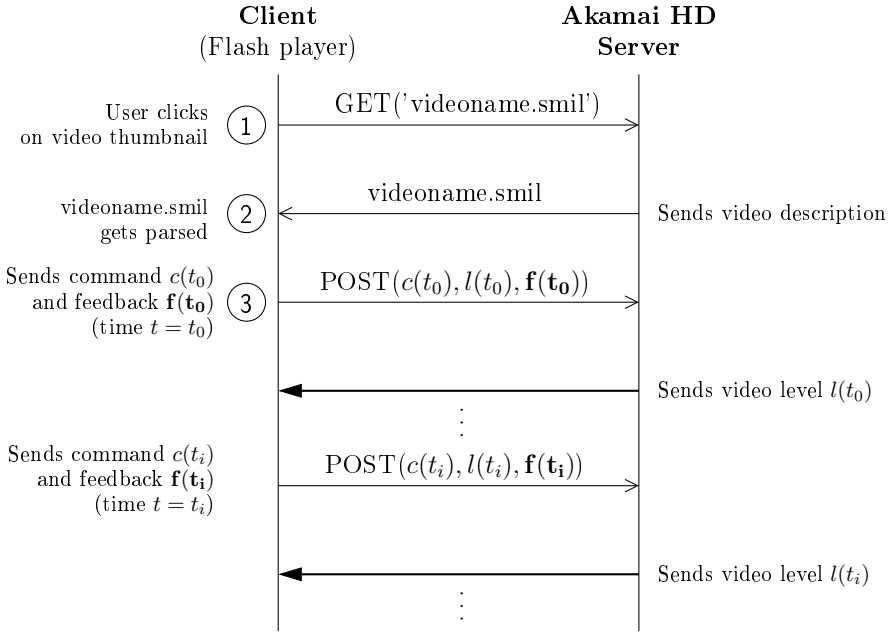


Fig. 2: Client-server time sequence graph: thick lines represent video data transfer, thin lines represent HTTP requests sent from client to server

At first, the client connects to the server [1], then a Flash application is loaded and a number of videos are made available. When the user clicks on the thumbnail (1) of the video he is willing to play, a GET HTTP request is sent to the server pointing to a Synchronized Multimedia Integration Language 2.0 (SMIL) compliant file[2]. The SMIL file provides the base URL of the video (`httpBase`), the available levels, and the corresponding encoding bit-rates. An excerpt of this file is shown in Figure 3.

Then, the client parses the SMIL file (2) so that it can easily reconstruct the complete URLs of the available video levels and it can select the corresponding video level based on the quality adaptation algorithm. All the videos available on the demo website are encoded at five different bitrates (see Figure 3). In particular, the video level bitrate $l(t)$ can assume values in the *set of available*

[2] http://www.w3.org/TR/2005/REC-SMIL2-20050107/

```
<head>
 <meta name="title" content="Elephants Dream" />
 <meta name="httpBase"
       content="http://efvod-hdnetwork.akamai.com.edgesuite.net/"/>
 <meta name="rtmpAuthBase" content="" />
</head>
<body>
 <switch id="Elephants Dream">
  <video src="ElephantsDream2_h264_3500@14411" system-bitrate="3500000"/>
  <video src="ElephantsDream2_h264_2500@14411" system-bitrate="2500000"/>
  <video src="ElephantsDream2_h264_1500@14411" system-bitrate="1500000"/>
  <video src="ElephantsDream2_h264_700@14411" system-bitrate="700000"/>
  <video src="ElephantsDream2_h264_300@14411" system-bitrate="300000"/>
 </switch>
</body>
```

Fig. 3: Excerpt of the SMIL file

video levels $L = \{l_0, \ldots, l_4\}$ at any given time instant t. Video levels are encoded at 30 frames per second (fps) using H.264 codec with a group of picture (GOP) of length 12. The audio is encoded with Advanced Audio Coding (AAC) at 128 kbps bitrate.

Table 1: Video levels details

Video level	**Bitrate (kbps)**	**Resolution (width×height)**
l_0	300	320x180
l_1	700	640x360
l_2	1500	640x360
l_3	2500	1280x720
l_4	3500	1280x720

Table 1 shows the video resolution for each of the five video levels l_i, that ranges from 320×180 up to high definition 1280×720.

It is worth to notice that each video level can be downloaded individually issuing a HTTP GET request using the information available in the SMIL file. This suggests that the server does not segment the video as in the case of the Apple HTTP adaptive streaming, but it encodes the original raw video source into N different files, one for each available level.

After the SMIL file gets parsed, at time $t = t_0$ (3), the client issues the first POST request specifying five parameters, two of which will be discussed in detail here[3]. The first POST parameter is `cmd` and, as its name suggests, it specifies a

[3] The remaining three parameters are not of particular importance. The parameter `v` reports the HDCore Library of the client, the parameter `g` is fixed throughout all

command the client issues on the server. The second parameter is `lvl1` and it specifies a number of feedback variables that we will discuss later.

At time $t = t_0$, the quality adaptation algorithm starts. For a generic time instant $t_i > t_0$ the client issues commands via HTTP POST requests to the server in order to select the suitable video level. It is worth to notice that the commands are issued on a separate TCP connection that is established at time $t = t_0$. We will focus on the dynamics of the quality adaptation algorithm employed by Akamai in the next section.

3.1 The `cmd` parameter

Let us now focus on the commands the client issues to the server via the `cmd` parameter.

Table 2: Commands issued by the client to the streaming server via `cmd` parameter

Command	Number of arguments	Occurrence (%)
c_1 `throttle`	1	~80%
c_2 `rtt-test`	0	~15%
c_3 `SWITCH_UP`	5	~2%
c_4 `BUFFER_FAILURE`	7	~2%
c_5 `log`	2	~1%

Table 2 reports the values that the `cmd` parameter can assume along with the number of command arguments and the occurrence percentage.

We describe now the basic tasks of each command, and leave a more detailed discussion to Section 4.

The first two commands, i.e. `throttle` and `rtt-test`, are issued periodically, whereas the other three commands are issued when a particular event occurs. The periodicity of throttle and `rtt-test` commands can be inferred by looking at Figure 4 that shows the cumulative distribution functions of the interdeparture times of two consecutive `throttle` or `rtt-test` commands. The Figure shows that `throttle` commands are issued with a median interdeparture time of about 2 seconds, whereas `rtt-test` commands are issued with a median interdeparture time of about 11 seconds.

`throttle` is the most frequently issued command and it specifies a single argument, i.e. the *throttle percentage* $T(t)$. In the next Section we will show that:

$$T(t) = \frac{\overline{r}(t)}{l(t)} 100 \tag{1}$$

the connection, whereas the parameter `r` is a variable 5 letters string that seems to be encrypted.

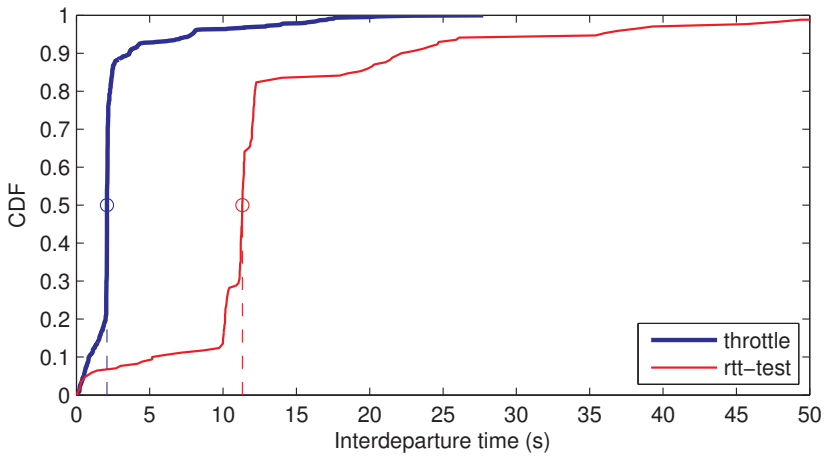


Fig. 4: Cumulative distribution functions of interdeparture time between two consecutive `throttle` or `rtt-test` commands

where $\overline{r}(t)$ is the maximum sending rate at which the server can send the video level $l(t)$. Thus, when $T(t) > 100\%$ the server is sending the video at a rate that is greater than the video level encoding rate $l(t)$. It is important to stress that in the case of live streaming it is not possible for the server to supply a video at a rate that is above the encoding bitrate since the video source is not pre-encoded. A traffic shaping algorithm can be employed to bound the sending rate to $\overline{r}(t)$. We will see in detail in Section 4 that this command plays a fundamental role in controlling the receiver buffer length.

The `rtt-test` command is issued to ask the server to send data in greedy mode. We conjecture that this command is periodically issued in order to actively estimate the end-to-end available bandwidth.

The `SWITCH_UP` command is issued to ask the server to switch from the current video level l_j to a video level l_k characterized with an higher encoding bitrate, i.e. $k > j$. We were able to identify four out of the five parameters supplied to the server: 1) the estimated bandwidth $b(t)$; 2) the bitrate l_k of the video level the client wants to switch up; 3) the video level identifier k the client wants to switch up; 4) the filename of the video level l_j that is currently playing.

The `BUFFER_FAIL` command is issued to ask the server to switch from the current video level l_j to a video level l_k with a lower encoding bitrate, i.e. $k < j$. We identified four out of the seven parameters supplied with this command: 1) the video level identifier k the client wants to switch down; 2) the bitrate l_k of the video level the client wants to switch down; 3) the estimated bandwidth $b(t)$; 4) the filename of the video level l_j that is currently playing.

The last command is `log` and it takes two arguments. Since this command is rarely issued, we are not able to explain its function.

3.2 The `lvl1` parameter

The `lvl1` parameter is a string made of 12 feedback variables separated by commas. We have identified 8 out of the 12 variables as follows:

1. **Receiver Buffer size** $q(t)$: it represents the number of seconds stored in the client buffer. A key goal of the quality adaptation algorithm is to ensure that this buffer never gets empty.
2. **Receiver buffer target** $q_T(t)$: it represents the desired size of the receiver buffer size measured in seconds. As we will see in the next Section the value of this parameter is in the range $[7, 20]$s.
3. unidentified parameter
4. **Received video frame rate** $f(t)$: it is the frame rate, measured in frames per second, at which the receiver decodes the video stream.
5. unidentified parameter
6. unidentified parameter
7. **Estimated bandwidth** $b(t)$: it is measured in kilobits per second.
8. **Received goodput** $r(t)$: it is the received rate measured at the client, in kilobits per second.
9. **Current video level identifier**: it represents the identifier of the video level that is currently received by the client. This variable assumes values in the set $\{0, 1, 2, 3, 4\}$.
10. **Current video level bitrate** $l(t)$: it is the video level bitrate measured in kilobits per second that is currently received by the client. This variable assumes values in the set $L = \{l_0, l_1, l_2, l_3, l_4\}$ (see Table 1).
11. unidentified parameter
12. **Timestamp** t_i: it represents the Unix timestamp of the client.

4 The quality adaptation algorithm

In this Section we discuss the results obtained in each of the considered scenarios. Goodput measured at the receiver and several feedback variables specified in the `lvl1` parameters will be reported. It is worth to notice that we do not employ any particular video quality metric (such as PSNR or other QoE indices). The evaluation of the QoE can be directly inferred by the instantaneous video level received by the client. In particular, the higher the received video level $l(t)$ the higher the quality perceived by the user. For this reason we employ the received video level $l(t)$ as the key performance index of the system.

In order to assess the efficiency η of the quality adaptation algorithm we propose to use the following metric:

$$\eta = \frac{\hat{l}}{l_{max}} \tag{2}$$

where $\hat{l}$ is the average value of the received video level and $l_{max} \in L$ is the maximum video level that is below the bottleneck capacity. The index is 1 when

the average value of the received video level is equal to l_{max}, i.e. when the video quality is the best possible with the given bottleneck capacity.

An important index to assess the Quality of Control (QoC) of the adaptation algorithm is the transient time required for the video level $l(t)$ to match the available bandwidth $b(t)$.

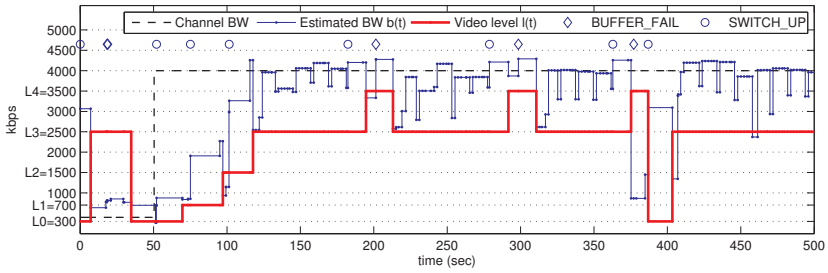
In the following we will investigate the quality adaptation control law employed by Akamai HD network in order to adapt the video level to the available bandwidth variations.

4.1 The case of a step-like change of the bottleneck capacity

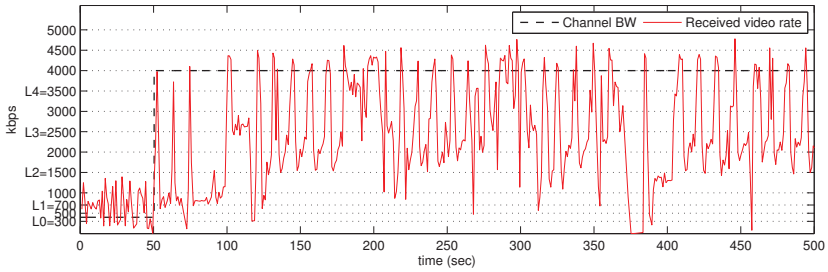
We start by investigating the dynamic behaviour of the quality adaptation algorithm when the bottleneck bandwidth capacity increases at time $t = 50$ s from a value of $A_m = 500$ kbps to a value of $A_M = 4000$ kbps. It is worth to notice that $A_m > l_0$ and that $A_M > l_4$ so that we should be able to test the complete dynamics of the $l(t)$ signal. Since for $t > 50$s the available bandwidth is well above the encoding bitrate of the maximum video level l_4 we expect the steady state video level $l(t)$ to be equal to l_4. The aim of this experiment is to investigate the features of the quality adaptation control. In particular we are interested in the dynamics of the received video level $l(t)$ and of the receiver buffer length $q(t)$. Moreover, we are interested to validate the command features described in the previous Section.

Figure 5 shows the results of this experiment. Let us focus on Figure 5 (a) that shows the dynamics of the video level $l(t)$ and the estimated bandwidth $b(t)$ reported by the `lvl1` parameter. In order to show their effect on the dynamics of $l(t)$, Figure 5 (a) reports also the time instants at which `BUFFER_FAIL` and `SWITCH_UP` commands are issued.

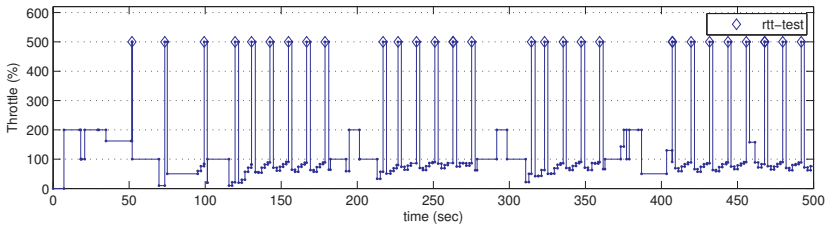
The video level is initialized at l_0 that is the lowest available version of the video. Nevertheless, at time $t = 0$ the estimated bandwidth is erroneously overestimated to a value above 3000 kbps. Thus, a `SWITCH_UP` command is sent to the server. The effect of this command occurs after a delay of 7.16 s when the channel level is increased to $l_3 = 2500$ kbps that is video level closest to the estimated bandwidth initialized at $t = 0$. By setting the video level to l_3, which is above the channel bandwidth $A_m = 500$ kbps, the received buffer length $q(t)$ starts to decrease and it eventually goes to zero at $t = 17.5$ s. Figure 5 (e) shows that the playback frame rate is zero, meaning that the video is paused, in the time interval $[17.5, 20.8]$ s. At time $t = 18.32$ s, a `BUFFER_FAIL` command is finally sent to the server. After a delay of about 16 s the server switches the video level to $l_0 = 300$ kbps that is below the available bandwidth A_m. We carefully examined each `BUFFER_FAIL` and `SWITCH_UP` command and we have found that to each `BUFFER_FAIL` command corresponds a decrease in the video level $l(t)$. On the other hand, when a `SWITCH_UP` command is issued the video level is increased. Moreover, we evaluated the delays incurring each time such commands are issued. We found that the average value of the delay for `SWITCH_UP` is $\tau_{su} \simeq 14$ s, whereas for what concerns the `BUFFER_FAIL` command



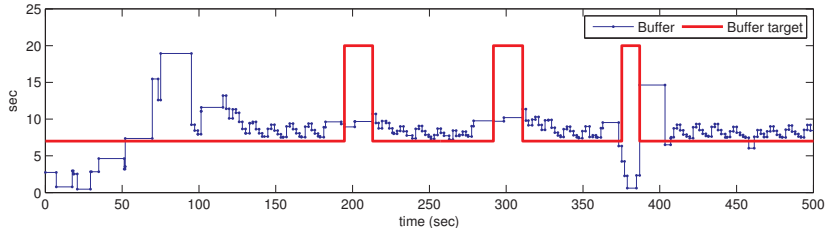
(a) Estimated BW, video level , BUFFER_FAIL, and SWITCH_UP events



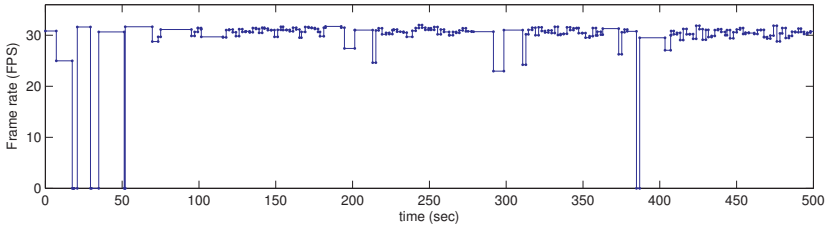
(b) Measured received video rate



(c) Throttle



(d) Receiver buffer length $q(t)$ and target buffer length $q_T(t)$



(e) Frame rate $f(t)$

Fig. 5: Akamai adaptive video streaming response to a step change of available bandwidth at $t = 50$s

the average value is $\tau_{sd} \simeq 7$ s. These delays pose a remarkable limitation to the responsiveness of the quality adaptation algorithm.

By considering the dynamics of the received frame rate, shown in Figure 5 (e), we can infer that the quality adaptation algorithm does not throttle the frame rate to shrink the video sending rate. We can conclude that the video level $l(t)$ is the only variable used to adapt the video content to the network available bandwidth.

Let us now focus on the dynamics of the estimated bandwidth $b(t)$. When the bottleneck capacity increases to $A_M = 4000$ kbps, $b(t)$ slowly increases and, after a transient time of 75 s, it correctly estimates the bottleneck capacity A_M. Figure 5 (a) shows that `SWITCH_UP` commands are sent to select level l_i when the estimated bandwidth $b(t)$ becomes sufficiently greater than l_i. Due to the large transient time of $b(t)$, and to the delay τ_{su}, the transient time required for $l(t)$ to reach the maximum video level l_4 is around 150 s. Even though we are not able to identify the algorithm that Akamai employs to adapt $l(t)$, it is clear that, as we expected, the dynamics of the estimated bandwidth plays a key role in controlling $l(t)$. Finally, to assess the performance of the quality adaptation algorithm, we evaluated the efficiency η by using equation (2), finding a value of 0.676 and the average absolute error $|q_T(t) - q(t)|$ that is equal to 3.4 s.

Another important feature of Akamai streaming server can be inferred by looking at Figure 5 (c) that shows the throttle signal $T(t)$ and time instants at which `rtt-test` commands are issued. The figure clearly shows that each time a `rtt-test` command is sent, the throttle signal is set to 500%. By comparing Figure 5 (b) and Figure 5 (c) we can infer that when a `rtt-test` command is sent the received video rate shows a peak which is close to the channel capacity, in agreement with (1). Thus, we can state that when the throttle signal is 500% the video flow acts as a greedy TCP flow. For this reason, we conjecture that the purpose of such commands is to actively probe for the available bandwidth.

In order to validate equation (1), Figure 6 compares the measured received video rate with the maximum sending rate that can be evaluated as $\overline{r}(t) = \frac{T(t)}{100} l(t)$. The figure shows that equation (1) is able to model quite accurately the maximum rate at which the server can send the video. Nevertheless, it is important to stress that the measured received rate is bounded by the available bandwidth and its dynamics depends on the TCP congestion control algorithm.

The last feature we investigate in this scenario is the way the throttle signal $T(t)$ is controlled. In first instance, we conjecture that $T(t)$ is the output of a feedback control law whose goal is to make the difference between the target buffer length $q_T(t)$ and the buffer length $q(t)$ as small as possible. Based on the experiments we run, we conjecture the following control law:

$$T(t) = \max\left((1 + \frac{q_T(t) - q(t)}{q_T(t)})100, 10\right) \tag{3}$$

The throttle signal is 100%, meaning that $\overline{r}(t) = l(t)$, when the buffer length matches the buffer length target, i.e. when $q_T(t) = q(t)$. When the error $q_T(t) - q(t)$ increases, $T(t)$ increases accordingly in order to allow the maximum sending rate $\overline{r}(t)$ to increase so that the buffer can be filled.

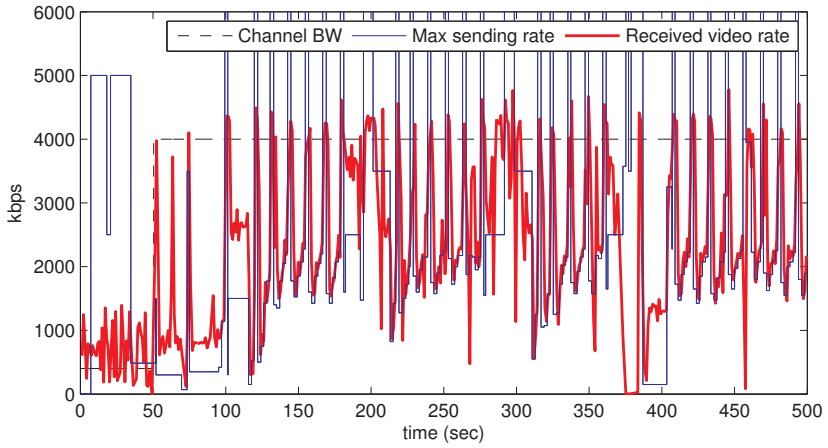


Fig. 6: Maximum sending rate $\bar{r}(t)$ and received video rate dynamics when the available bandwidth varies as a step function

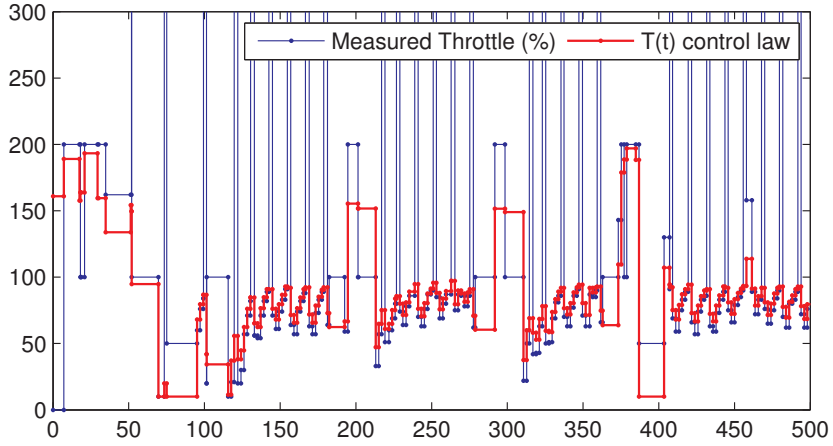


Fig. 7: Measured throttle signal $T(t)$ compared to the conjectured control law, eq. (3)

Figure 7 compares the measured throttle signal with the dynamics obtained by using the conjectured control law (3). Apart from the behaviour of the throttle signal in correspondence of `the rtt-test` commands that we have already commented above, equation (3) recovers with a small error the measured throttle signal.

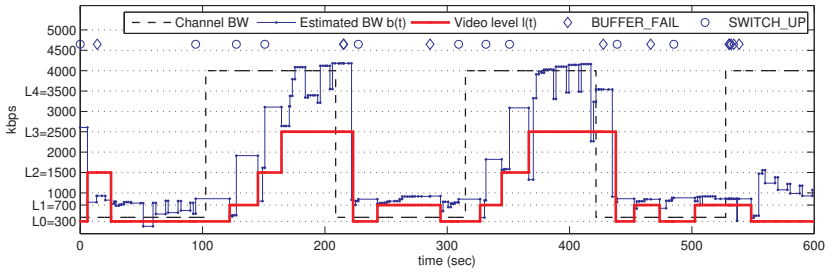
To summarize, the main results of this experiment are the following: 1) the only variable used to adapt the video source to the available bandwidth is the video level $l(t)$; 2) the video level $l(t)$ takes around 150 s to match the available bandwidth; 3) when a `BUFFER_FAIL` command is sent to switch the video level down, the server takes $\tau_{sd} \simeq 7$ s to actuate this command; 4) when a `SWITCH_UP` command is sent to switch the video level up, the server takes $\tau_{su} \simeq 14$ s to actuate the command; 5) when a `rtt-test` command is issued the throttle signal is set to 500% allowing the video flow to act as a greedy TCP flow to actively probe for the available bandwidth; 6) a feedback control law is employed to ensure that the player buffer length $q(t)$ tracks the desired buffer length $q_T(t)$.

4.2 The case of a square-wave varying bottleneck capacity

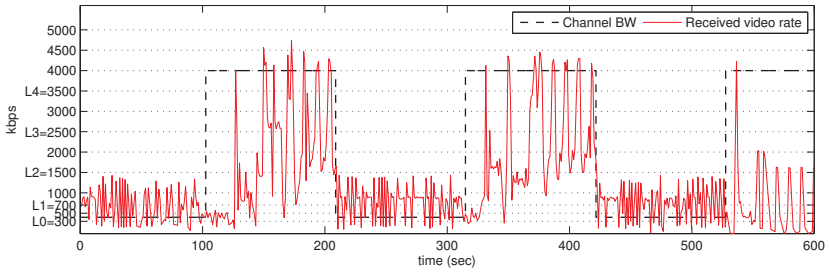
In this experiment we show how the quality adaptation algorithm reacts in response to abrupt drops/increases of the bottleneck capacity. Towards this end, we let the bottleneck capacity to vary as a square-wave with a period of 200 s, a minimum value $A_m = 400$ kbps and a maximum value $A_M = 4000$ kbps. The aim of this experiment is to assess if Akamai adaptive video streaming is able to quickly shrink the video level when an abrupt drop of the bottleneck capacity occurs in order to guarantee continuous reproduction of the video content.

Figure 8 shows the results of this experiment. Let us first focus on Figure 8 (a): when the first bandwidth drop occurs at time $t \simeq 208$ s, a `BUFFER_FAIL` is sent to the server after a delay of roughly 7 s in order to switch down the video level from l_3 to l_0. After that, a switch-down delay τ_{sd} of 7 s occurs and the video level $l(t)$ is finally switched to l_0. Thus, the total delay spent to correctly set the video level $l(t)$ to match the new value of the available bandwidth is 14 s. Because of this large delay an interruption in the video reproduction occurs 13 s after the bandwidth drop as it can be inferred by looking at Figure 8 (d) and Figure 8 (e). The same situation occurs when the second bandwidth drop occurs. In this case, the total delay spent to correctly set the video level is 16 s. Again, 13 s after the second bandwidth drop, an interruption in the video reproduction occurs. We found an efficiency $\eta = 1$ when the bandwidth is $A_m = 400$ kbps, i.e. the quality adaptation algorithm delivers the best possible quality to the client. On the contrary, during the time intervals with bandwidth $A_M = 4000$ kbps, the efficiency is roughly 0.5. Finally, in this scenario the average absolute error $|q_T(t) - q(t)|$ is equal to 3.87 s.

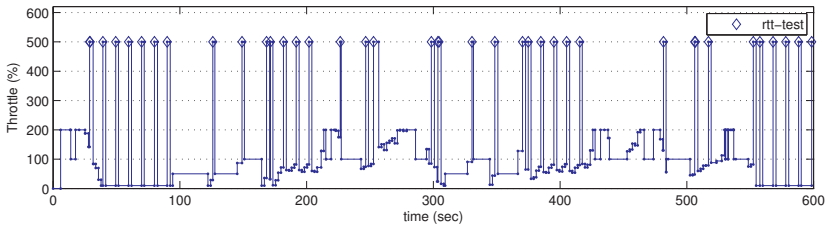
To summarize, this experiment has shown that short interruptions affect the video reproduction when abrupt changes in the available bandwidth occur. The main cause of this issue is that the video level is switched down with a delay of roughly 14 s after the bandwidth drop occurs.



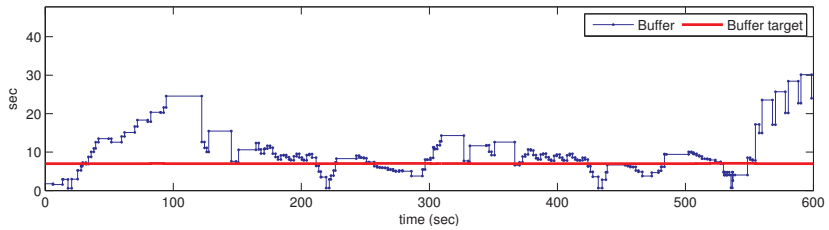
(a) Estimated BW, video level , `BUFFER_FAIL`, and `SWITCH_UP` events



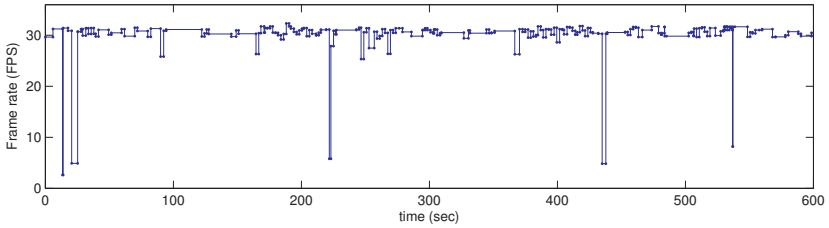
(b) Measured received video rate



(c) Throttle



(d) Receiver buffer length $q(t)$ and target buffer length $q_T(t)$



(e) Frame rate $f(t)$

Fig. 8: Akamai adaptive video streaming response to a square-wave available bandwidth with period 200 s

4.3 The case of one concurrent greedy TCP flow

This experiment investigates the quality adaptation algorithm dynamics when one Akamai video streaming flow shares the bottleneck capacity with one greedy TCP flow. The bottleneck capacity has been set to 4000 kbps, a video streaming session has been started at $t = 0$ and a greedy TCP flow has been injected at time $t = 150$ s and stopped at time $t = 360$ s.

Figure 9 (a) shows the video level dynamics $l(t)$ and the estimated bandwidth $b(t)$. Vertical dashed lines divide the experiment in three parts.

During the first part of the experiment, i.e. for $t < 150$ s, apart from a short time interval $[6.18, 21.93]$ s during which $l(t)$ is equal to $l_4 = 3500$ kbps, the video level is set to $l_3 = 2500$ kbps. The efficiency η in this part of the experiment is 0.74.

When the second part of the experiment begins ($t = 150$ s), the TCP flow joins the bottleneck grabbing the fair bandwidth share of 2000 kbps. Nevertheless, the estimated bandwidth $b(t)$ decreases to the correct value after 9 s. After an additional delay of 8 s, at $t = 167$ s, a `BUFFER_FAIL` command is sent (see Figure 9 (a)). The video level is shrunk to the suitable value $l_2 = 1500$ kbps after a total delay of 24 s. In this case, this actuation delay does not affect the video reproduction as we can see by looking at the frame rate dynamics shown in Figure 9 (e). At time $t = 182$ s a second `BUFFER_FAIL` command is set and the video level is shrunk after the usual delay $\tau_{sd} \simeq 7$ s at time $t = 189$ s. At time $t = 193$ s an `rtt-test` command is issued so that for a short amount of time the video flow becomes greedy (see Subsection 4.1). At time $t = 196$ s the bandwidth is estimated to 2200 kbps so that a `SWITCH_UP` command is sent and at $t = 212$ s the video level is switched up to the suitable value of $l_2 = 1500$ kbps. The efficiency η in this part of the experiment is 1, i.e. the best video quality has been provided.

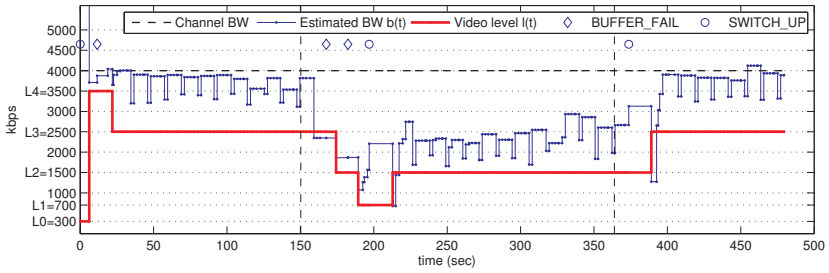
Finally, when the TCP flow leaves the bottleneck at time $t = 360$ s, the level is switched up to $l_3 = 2500$ kbps with a delay of 26 s. In this part of the experiment the efficiency is 0.69.

To summarize, this experiment has shown that the Akamai video streaming flow correctly adapt the video level when sharing the bottleneck with a greedy TCP flow.

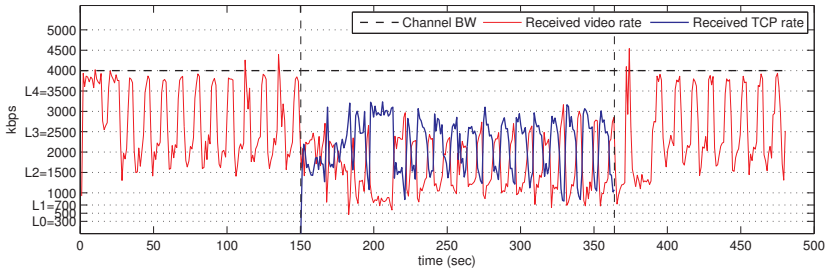
5 Conclusions

In this paper we have shown the results of an experimental evaluation of Akamai adaptive streaming. The contribution of this paper is twofold: firstly, we have analyzed the client-server protocol employed in order to actuate the quality adaptation algorithm; secondly, we have evaluated the dynamics of the quality adaptation algorithm in three different scenarios.

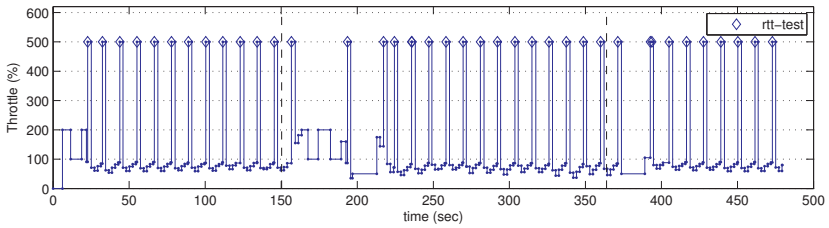
For what concerns the first issue, we have identified the POST messages that the client sends to the server to manage the quality adaptation. We have shown that each video is encoded in five versions at different bitrates and stored in



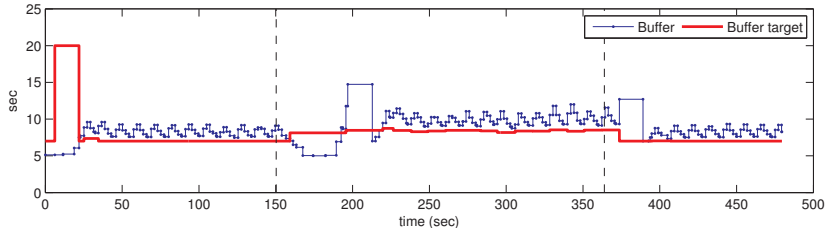
(a) Estimated BW, video level , `BUFFER_FAIL`, and `SWITCH_UP` events



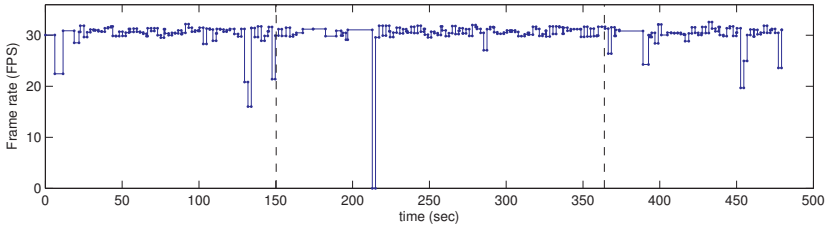
(b) Measured received video rate



(c) Throttle



(d) Receiver buffer length $q(t)$ and target buffer length $q_T(t)$



(e) Frame rate $f(t)$

Fig. 9: Akamai adaptive video streaming when sharing the bottleneck with a greedy TCP flow

separate files. Moreover, we identified the feedback variables sent from the client to the server by parsing the parameters of the POST messages. We have found that the client sends commands to the server with an average interdeparture time of about 2 s, i.e. the control algorithm is executed on average each 2 seconds.

Regarding the second issue, the experiments carried out in the three considered scenarios let us conclude that Akamai uses only the video level to adapt the video source to the available bandwidth, whereas the frame rate of the video is kept constant. Moreover, we have shown that when a sudden increase of the available bandwidth occurs, the transient time to match the new bandwidth is roughly 150 seconds. Furthermore, when a sudden drop in the available bandwidth occurs, short interruptions of the video playback can occur due to the a large actuation delay. Finally, when sharing the bottleneck with a TCP flow, no particular issues have been found and the video level is correctly set to match the fair bandwidth share.

References

1. Akamai hd network demo. http://wwwns.akamai.com/hdnetwork/demo/flash/default.html.
2. Move networks hd adaptive video streaming. http://www.movenetworkshd.com.
3. L. De Cicco and S. Mascolo. A Mathematical Model of the Skype VoIP Congestion Control Algorithm. *IEEE Trans. on Automatic Control*, 55(3):790–795, March 2010.
4. David Hassoun. Dynamic streaming in flash media server 3.5. Available online: http://www.adobe.com/devnet/flashmediaserver/articles/dynstream_advanced_pt1.html, 2009.
5. Adobe Systems Inc. Real-Time Messaging Protocol (RTMP) Specification. 2009.
6. R. Kuschnig, I. Kofler, and H. Hellwagner. An evaluation of TCP-based rate-control algorithms for adaptive internet streaming of H. 264/SVC. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 157–168. ACM, 2010.
7. R. Pantos and W. May. HTTP Live Streaming. *IETF Draft*, June 2010.
8. B. Wang, J. Kurose, P. Shenoy, and D. Towsley. Multimedia streaming via TCP: An analytic performance study. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 4(2):1–22, 2008.
9. A. Zambelli. IIS smooth streaming technical overview. *Microsoft Corporation*, 2009.