Tracker-assisted rate adaptation for MPEG DASH live streaming

Andrea Detti, Bruno Ricci, Nicola Blefari-Melazzi
CNIT, Electronic Engineering Dept., University of Rome "Tor Vergata", Italy
andrea.detti@uniroma2.it, bruno.ricci@uniroma2.it, blefari@uniroma2.it

***Abstract*—MPEG DASH is a widely used standard for adaptive video streaming over HTTP. The conceptual architecture for DASH includes a web server and clients, which download media segments from the server. Clients select the resolution of video segments by using an Adaptive Bit-Rate (ABR) strategy; in particular, a throughput-based ABR is used in the case of live video applications. However, recent papers show that these strategies may suffer from the presence of proxies/caches in the network, which are instrumental in streaming video on a large scale. To face this issue, we propose to extend the MPEG DASH architecture with a *Tracker* functionality, enabling client-to-client sharing of control information. This extension paves the way to a novel family of Tracker-assisted strategies that allow a greater design flexibility, while solving the specific issue caused by proxies/caches; in addition, its utility goes beyond the problem at hand, as it can be used by other applications as well, e.g. for peer-to-peer streaming.**

I. Introduction

Nowadays, video contents over the Internet are mainly streamed using HTTP. Typically, a video content is segmented in parts and the streaming session is a sequence of HTTP GETs controlled by the client. Recent solutions are adaptive, offering different representations of the same video in terms of resolution, coding rate, etc. This variety allows the client to dynamically select the representation more suitable to its current context: e.g. device type, available bandwidth, energy constraints. HTTP adaptive video streaming systems support both video on-demand and live video applications. In the case of video on-demand, the entire media is made available and each user can choose the playback time (e.g. YouTube). In case of live video, the media is made available at a given time and possibly removed at another time, and the playback of all users is synchronized (e.g. IPTV).

Since 2010 a growing consortium of companies is designing an international standard for HTTP adaptive video streaming, the MPEG Dynamic Adaptive Streaming over HTTP (DASH) [1]. An important organization of the MPEG DASH ecosystem is the MPEG DASH Industry Forum [2], which is developing the so called dash.js reference client [3], an open-source javascript DASH player working with HTML5 browsers.

An MPEG DASH system has very good scalability properties versus the number of clients, both in terms of server bandwidth and processing loads. The bandwidth load can be scaled down by using HTTP proxy/cache infrastructures, such as those provided by Content Delivery Networks [4]. The processing load for the adaptive selection of the video representation is completely distributed on clients.

A client uses an Adaptive Bit-Rate (ABR) strategy to select a video representation. Literature papers and practical implementations propose several ABR strategies, roughly classifiable as: throughput-based, buffer-based or a combination of them [3] [5] [6]. A throughput-based ABR strategy estimates the available network bandwidth from past observations and selects the video representation with the highest coding rate lower than the estimated bandwidth. A buffer-based strategy modifies the current video coding rate depending on the occupation of the playout buffer.

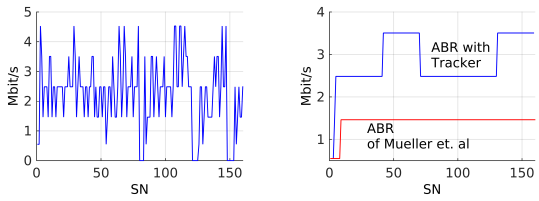
Throughput-based strategies are subject to throughput estimation errors and suffer from highly variable throughput environments [6]; however, they are more reactive than buffer-based ones and do not need large playout buffers. Buffer-based strategies are more stable in presence of highly variable throughput environments, but require larger buffer capacity.

In case of video on-demand, both throughput-based and buffer-based ABR strategies can be adopted; the playout delay is not a relevant performance parameter, and thus the playout buffer capacity can be quite large (few minutes). In case of live video applications, throughput-based ABR strategies are often the only possible option, since these applications require short playout delay and thus small playout buffer (few seconds).

In [7], C. Mueller, S. Lederer and C. Timmerer observed, through video on-demand laboratory experiments, that the presence of a cache may hamper the effectiveness of throughput-based ABR strategies. Segments retrieved from caches can alter the estimation of the actual client-server available network capacity; wrong capacity estimations lead to wide coding rate oscillations, segment losses and re-buffering events. Thus, cache infrastructures, supposed to be a friend of MPEG DASH, may actually be a foe [8].

We argue that such a caching issue [7] can also take place in case of live video applications since, in a realistic Internet scenario, the not perfect synchronization among video clients may yield to cache-hit events. In this context, the problem is even more critical with respect to on-demand scenarios, since throughput-based ABR strategies are the only sensible option. Fig. 1(a) provides an evidence of our argumentation. It reports the video coding rate versus the video segment

This research was partly funded by the EU H2020 Bonvoyage project



(a) Basic throughput-based ABR (b) Optimized throughput-based ABRs

Fig. 1. Video coding rate of a client in case of 10 clients, Web server and Cache/Proxy on the real Internet. Server output bandwidth limited to 8 Mbit/s

number achieved by a client that uses the basic throughput-based ABR strategy earlier described. Ten clients are behind a proxy, which is connected to an HTTP server providing the MPEG DASH media. All processes run on real Internet nodes. We observe very unpleasant oscillations of the video coding rate and segment losses (i.e., coding rate = 0), implying re-buffering events.

Motivated by such important practical consequences of the caching issue, we aim to make HTTP caches an advantage rather than a disadvantage also for MPEG DASH. To the best of our knowledge, this is the first paper focused on the live video use-case, when clients are (mildly) synchronized among each other.

We propose to extend the MPEG DASH system architecture with a *Tracker* functionality. The Tracker enables clients to share control information about their status (e.g. selected video coding rate, measured throughput), and then to make more informed decisions. This extension does not affect the scalability properties of MPEG DASH and can be implemented on the server side only, by using a javascript for the player implementation. The extension can be exploited to design novel and more flexible Tracker-assisted ABR algorithms, but can also be used by other applications (e.g. P2P streaming [9], real-time user commenting, etc.).

As a proof of concept, we devise a simple Tracker-assisted throughput-based ABR strategy for improving live video quality in presence of network caches. We use the AVC video coding scheme [10] and assume the presence of a single cache on the client-server path, leaving hierarchical cache scenarios for future works.

II. Related Work

MPEG DASH live video applications

An MPEG DASH system is composed of an HTTP server and a number of DASH clients. The HTTP server provides a Media Presentation Description (MPD) and segments. The MPD describes the representations available for a media content (coding rate, resolution, codec type, etc.), the list of segment URLs, the segment duration, and so forth. Segments contain chunks of actual multimedia bitstreams, whose duration is a design choice. For instance, a chunk can be 4 seconds long, or even the whole content.

A client first fetches and parses the MPD file and then starts the playback by downloading segments through HTTP GETs. The representation of a segment is locally decided by an ABR algorithm, and the time scheduling of HTTP GETs depends on the streaming application: live or on-demand. In case of live video, the publication of new segments by the server and the related GETs from clients are almost concurrent events. A client downloads only one segment at a time, and when a download is completed the client waits for the next segment to be made available by the server. Such a mild client-server synchronization is achieved by preliminary aligning client and server clocks, e.g. through NTP means. Before starting the playback, a client stores a given amount of segments in a playout buffer. In case of buffer depletion, the playback stops, the client re-syncs the segment GET process to the current published sequence number, refills the playout buffer (aka rebuffering) and then restarts the playback.

The MPEG DASH standard defines formats for MPD and segments in a very flexible way. It makes possible to restrict the applied formats by means of the definition of Profiles; external organization may further restrict a Profile by means of the specification of Interoperability Points and Extensions. Currently, the MPEG DASH Industry Forum has released the DASH-AVC/264 Interoperability Guidelines [11] in which the video codec is the H.264 AVC.

ABR strategies

Literature papers and implementations suggest several ABR strategies (e.g. see [12] for an experimental comparison). However, very few papers face the issue of coding rate oscillation in case of throughput-based ABR strategies due to caches; and we experimentally observed that strategies used in the dash.js (v1.3.0) reference player suffer from this problem too.

In [7] Authors propose to mistrust bandwidth estimations made during segment downloads, and instead to estimate the server-to-client bandwidth by means of non-cacheable HTTP GETs, called *probes*. A probe may be an HTTP GET of any segment with a byte-range option, since proxy implementations usually do not cache range requests. In our real Internet experiments we found that this solution is actually effective in reducing coding rate oscillation. It has a simple deployment, since it only impacts the ABR logic, which can be pushed by the server, e.g. by a javascript implementation. However, the ABR logic uses only local knowledge, whereas our Tracker-assisted solution allows client-to-client sharing of control information achieving better performance.

In [8] Authors propose to solve the coding rate oscillation by shaping the output bandwidth from the proxy/cache to the client; indeed, rate oscillation shows up when the proxy-client bandwidth is greater than the available server-proxy bandwidth. This solution requires changing existing proxy infrastructures. Conversely, as in the case of [7], our solution only requires server side changes.

In [13] Authors propose the CF-DASH system, in which clients and proxy share information about a profile limit, i.e., a limitation on the highest coding rate that a client can request among the ones available on the server. In doing so, ABR selections of high speed clients will saturate on the profile

limit, thus increasing cache-hit events. The paper shows the benefit of having a profile limit, albeit the computation of this value is not addressed. Our solution does not impose such a global limit for all clients, but each client can select its own best coding rate, according to dynamic network conditions.

Finally, we believe that it is useful to report the interesting findings of [14], even though they are not related to a scenario with caches. The Authors observe that the client-server traffic pattern is mainly an ON-OFF process, in which the duration of the ON and OFF periods depends on the selected coding rate. Consequently, clients that make throughput measurements during the OFF periods of other clients wrongly overestimate the available bandwidth. This error leads the clients to temporarily scale up the coding rate and then scale it down again, since the higher rate is actually not sustainable. Authors propose a server-based solution to reduce such phenomenon, in which the server carries out a per-client traffic shaping. In case of live video applications, clients' requests are almost synchronized, and we experimentally observed that such an ON-OFF issue, although present, is less critical than in the case of video on-demand applications, as throughput measurements are less falsified. Moreover, our solution does not require the server to make per-client operations, which may hamper the processing scalability of an MPEG DASH system.

HTTP caching

An HTTP caching infrastructure is formed by HTTP proxy servers, such as Squid, Apache Traffic Server, Varnish, NGiNX, etc. There are two kinds of proxy deployments, namely forward and reverse proxy. A forward proxy forwards HTTP requests directed to any server and is usually deployed close to the end users, to speed up the service response time through caching. Proxy and servers are connected through Internet, thus they can suffer from bandwidth limitations, which trigger the coding rate oscillation phenomenon [8].

A reverse HTTP proxy is deployed in front of a server farm and handles requests only for HTTP servers of the server farm. It can be used to provide content-based services such as load balancing, caching, access control, etc. The connections among the proxy and the servers are very fast (e.g. Gigabit Lan), thus likely not able to trigger coding rate oscillation.

Proxy implementations can support both deployments, (e.g. Squid, Apache Traffic Server), or only one of them, (e.g. Varnish and NGiNX are reverse only proxies). Any recent release of these implementations support the "read-while-write" (aka collapsed forwarding) feature, with which concurrent multiple requests for the same URL are processed by only one request to the HTTP server. The read-while-write feature is fundamental for live video applications to reduce server-proxy bandwidth demand, since requests arrive to the cache concurrently. Without this feature, each request would be forwarded to the server, thus eliminating any caching benefit.

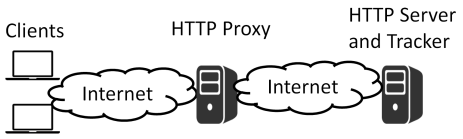


Fig. 2. Reference Tracker Scenario

III. Tracker-assisted MPEG DASH architecture

Basic observation

In this section we present the proposed MPEG DASH architectural extension. We observe that clients behind a same proxy use common network resources, e.g. the server-proxy Internet bandwidth. In this case, a shared knowledge about the status of such clients would make possible a finer prediction about the consequences of an ABR choice, with respect to a prediction achieved using only local knowledge. For instance, let us assume that a client A wishes to upscale to a video representation that a client B is already playing back. If client A is aware of B (shared knowledge), it knows that such an upscale will not demand additional server-proxy bandwidth, since the new stream towards A will be completely served by the proxy. Conversely, if the client A is not aware of B (local knowledge), it should safely, nevertheless wrongly, predict that the upscaling will activate a new server-proxy stream, with a consequent increase of server-proxy bandwidth demand.

Architectural proposal

Fascinated by the possibility to improve ABR performances by means of client-to-client knowledge sharing, we propose to deploy a Tracker functionality, which enables clients to exchange information about their status (selected video coding-rate, measured throughput, etc.). Fig. 2 reports the related architectural scenario.

Obviously, any extension of the MPEG DASH streaming system should not hamper its scalability properties and our Tracker complies with this requirement. The Tracker is merely a relay of client-to-client information; the ABR processing load completely remains on clients, and a proper design can limit client-tracker communications, possibly also leveraging on intermediate caches. Last but not least, the introduction of a Tracker does not require any modification of existing in-network proxies and can be implemented on the server side only, by using a javascript for the player implementation, and co-locating the Tracker with the HTTP server, e.g. using a basic Apache2/Tomcat web server configuration.

The results of Fig. 1(b) give a preview of the achievable improvements; we see that our Tracker-assisted ABR strategy (described later on) achieves better performance than the one proposed in [7]. This comparison, as the following ones, should not be understood as functional to show shortcomings of this or that ABR strategy. Instead, it should be considered as a demonstration that it is possible to devise better strategies by using a Tracker. The message that we want to convey with this paper is the usefulness of our architectural extension, rather than the performance of a specific ABR scheme.

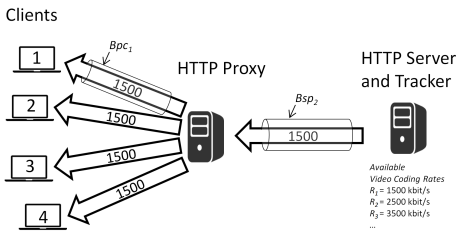


Fig. 3. Initial condition

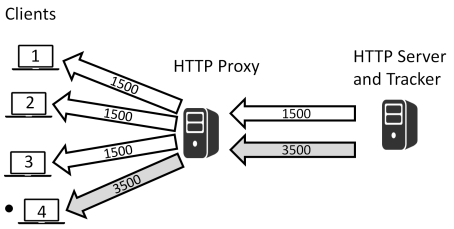


Fig. 4. Clique creation

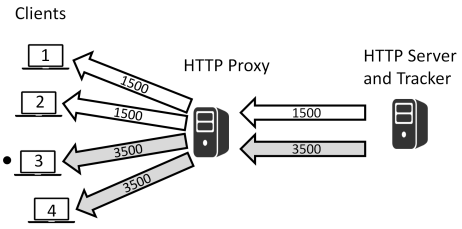


Fig. 5. Clique join

Tracker

The Tracker maintains the status of a list of clients behind the same HTTP proxy. We call this group of clients *swarm*, the whole set of status information as *swarm-status* and the status information of a single client as *client-status*. A client-status is the tuple <client-id, rep-id, bw>. The client-id is a random number uniquely identifying the client. The rep-id is the index of the current representation selected by the client. The bw is a measurement of the average download rate of the client.

A client gets swarm-status information by using HTTP GETs and exploits caching to reduce the Tracker traffic load. In addition, a client pushes updates about its status by using HTTP POSTs. At the POST reception, the Tracker inserts the client information in the swarm identified by the IP source address of the received message. In case of plain proxies this address is equal to the IP address of the proxy, thus all clients behind the same proxy belong to the same swarm [1].

IV. Tracker-assisted ABR strategy

In this section we describe a proof-of-concept, throughput-based ABR strategy that exploits the Tracker.

To devise the strategy we first studied and understood the impact of coding rate scaling on network bandwidth demands. Then we derived analytic conditions for the sustainability of a rate scaling, which take as input download rate measurements made by clients. Finally, we used the analytic findings to design a Tracker-assisted ABR algorithm.

Consequences of coding rate scaling

We describe the consequences of a rate scaling on network bandwidth demands through an example concerning a sequence of coding rate upscalings. The same reasoning can be repeated for a sequence of downscalings.

In Fig. 3 we have four clients, all playing video representation n. 2, whose coding rate is $R_2 = 1500$ kbit/s. We define kth *clique* the group of clients that is playing the video representation n. k. In this case, clients 1,2,3 and 4 form the clique n. 2. For each segment requested by the clients of a clique, the proxy downloads from the server one copy of the segment and relays the downloaded bits to each client. Consequently, the number of video streams transferred on the server-proxy network path is equal to the number of cliques. In this case, only one stream at 1500 kbit/s on the server-proxy network path.

In Fig. 4 client 4 upscales the coding rate to 3500 kbit/s. The client leaves the clique working at 1500 kbit/s and forms a new clique at 3500 kbit/s. A new stream is activated on the server-proxy path at 3500 kbit/s and an increase of the whole consumed bandwidth from the proxy to clients, from 6000 kbit/s to 8000 kbit/s. We call this event *clique-creation*.

In Fig. 5 the client 3 upscales the coding rate to 3500 kbit/s. The client leaves the clique at 1500 kbit/s and joins the clique at 3500 kbit/s. The involved cliques remain those at 1500 kbit/s and 3500 kbit/s, thus the server-proxy bandwidth demand is not affected by this upscale. Conversely, the whole consumed bandwidth from the proxy to the clients increases to 10 Mbit/s. We call this type of event *clique-join*.

In Fig. 6 all clients of the clique at 1500 kbit/s scale the coding rate to 2500 kbit/s. On the server-proxy path the old stream at 1500 kbit/s is switched off and a new stream at 2500 kbit/s is switched on. We call this type of event as *clique-switching*.

In Fig. 7 all clients of the clique at 2500 kbit/s scales the coding rate to 3500 kbit/s. No new stream is activated on the server-proxy path and the stream at 2500 kbit/s is switched off. We call this event as *clique-merging*.

Understanding the download rate measurement

In any throughput-based strategy, the triggers of the rate scaling are the measurements of the download rates (D) made by clients. Accordingly, it is fundamental to understand the relationship between the download rate and the bandwidth available on the involved network paths (Fig. 3): from server to proxy (Bsp) and from proxy to client (Bpc).

Let us assume that the ith client plays back the k representation, whose coding rate is R_k. The client measures

[1]It would be necessary to look for the presence of XFF (x-forwarder-for) HTTP headers to discriminate Proxy from NAT. However, accurate techniques to determine if a client is behind a proxy are out of the scope of this paper.

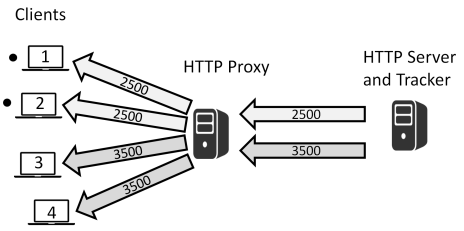


Fig. 6. Clique switching

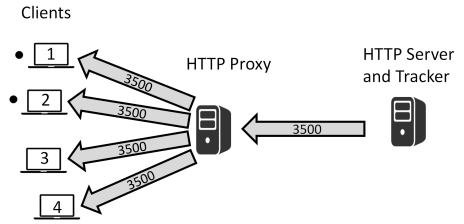


Fig. 7. Clique merging

an average download rate of D_i, which is the average ratio between segment length and segment download time.

In the case of a cache-hit, the rate D_i is an estimation of the average bandwidth Bpc_i that the client perceives on the proxy-client path. In the case of a cache-miss, the rate D_i is an estimation of the minimum between the average bandwidth Bpc_i, and the average bandwidth Bsp_k that the clique of the client has available on the server-proxy path.

Since clients are almost synchronized, a client usually experiences a *partial* cache-hit: finding only part of a segment stored in the cache, since another client has started the segment download before and the download is not completed yet.

In the case of a partial cache-hit, cached bits are downloaded at Bpc_i, while the download rate of the remaining missing bits is, theoretically, $\min(Bpc_i, Bsp_k)$. However, we experimentally observed that the latter rate depends on the implementation of read-while-write proxy functionality and it can be lower than the theoretical value. For instance, in the case of Squid proxy (v3.5.4) we observed that the download rate provided to clients concurrently downloading the same set of bits is bound by the download rate of the slowest client [2]. Conversely, the Apache Traffic Server proxy follows the expected theoretical behavior. To sum up, eq. (1) shows the relationship between the download rate and the network bandwidths.

$$\min\left(\left(\min_{j \in \text{ clique } k} Bpc_j\right), Bsp_k\right) \leq D_i \leq Bpc_i \quad (1)$$

Sustainability of rate scaling

A client i can carry out a rate scaling from coding rate R_k to R_h if the consequent clique event (creation, join, switching, merging) is "sustainable" by the bandwidths available on the involved network paths. Tab. I summarize scalability conditions hereafter discussed.

TABLE I
SUSTAINABILITY OF CLIENT i RATE SCALING FROM R_k TO R_h

Clique event	Sustainable when
creation	succesful attempt
join	$D_i \geq R_h$
switching and merging	$\min_{j \in \text{ clique } k} D_j \geq R_h$

Sustainability for clique-creation: a client i can create a new clique h if the server-proxy bandwidth Bsp_h available for the new clique and the proxy-client bandwidth Bpc_i are both greater than the coding rate R_h. In formulas:

$$Bsp_h \geq R_h \quad (2)$$

$$Bpc_i \geq R_h \quad (3)$$

Let us discuss now how to practically verify these inequalities. The download rate measurement D_i can not be used for this purpose, since it is related to the server-proxy bandwidth available for the clique k rather than the bandwidth Bsp_h that *will* be available to a new connection sustaining the new clique h. Consequently, we resort to the following *attempting approach* to verify the clique-creation sustainability.

The client starts to download the segment at coding rate R_h. In doing so, the client is temporarily activating the clique h. Assuming that only client i is doing such an attempt, a cache-miss occurs, and the measured download rate is $\min(Bpc_i, Bsp_h)$. This rate is frequently monitored (e.g. every 1s) during the segment download. If any measurement goes below R_h, one of the two inequalities fail and the rate R_h is not sustainable. Else, if the download of the segment at rate R_h ends, the coding rate is sustainable.

Sustainability for clique-join: a clique-join does not request additional bandwidth on the server-proxy path and a client i can join an existing clique h, if the available proxy-client bandwidth Bpc_i is greater than the coding rate R_h, i.e. inequality (3). As a consequence of (1), the inequality (3) is verified and clique-join is sustainable if the download rate $D_i \geq R_h$ [3].

Sustainability for clique-switching and merging: a client i can perform a clique-switching from clique k to clique h, if the server-proxy bandwidth Bsp_k available for the current clique can also sustain the coding rate R_h, and if the available proxy-client bandwidths of all clients of clique k are greater than R_h. In formulas:

$$Bsp_k \geq R_h \quad (4)$$

$$Bpc_j \geq R_h \qquad \forall j \in \text{clique } k \quad (5)$$

[2]We argue that this is due to the limited size of the single read-ahead-buffer used by the proxy to relay downloaded data to all served clients

[3]We note that the download rate D_i may be lower than Bpc_i (see 1). Therefore, there could be cases for which a possible clique-join is not recognized using $D_i \geq R_h$. However, for the sake of simplicity we used such a sufficient condition in the ABR strategy

We observe that both conditions are verified and clique-switching is sustainable if the minimum value among the download rates observed by any jth client of the clique k is greater or equal than the coding rate R_h, i.e.

$$\min_{j \in \text{ clique } k} D_j \geq R_h \tag{6}$$

The demonstration is easy. Due to (1), the inequality (6) verifies the condition (5). Regarding inequality (4), we split the demonstration in two complementary cases.

If any proxy-client bandwidth Bpc_j is greater than the server-proxy bandwidth Bsp_k, the first client f that downloads a segment experiences a cache-miss and measures a download rate D_f equal to the server-proxy available bandwidth Bsp_k. All other next clients experience a partial cache-hit and measure greater download rates. Therefore, $\min(D_j) = D_f = Bsp_k$, and the condition (6) verifies inequality (4).

In the remaining cases of some/all proxy-client bandwidths Bpc_j lower than Bsp_k, these clients measure a download rate D_j lower than Bsp_k. Consequently, $\min(D_j) < Bsp_k$, and condition (6) verifies inequality (4).

In the case of clique-merging, only inequality (5) is necessary. Therefore, also in this case, if condition (6) is true then clique-merging is sustainable.

The strategy

We designed and implemented an open-source live video client using the Python language [15]. The client fetches and consumes segments without actually reproducing them. For space limitations, client operations are cursorily described, while we discuss with more details the ABR strategy and scalability aspects of our solution. The client works as follows. At each publishing time, the client gets the swarm-status from the Tracker. The client then computes the sequence number of the next segment to download, and uses the ABR algorithm 1 to select the coding rate and fetch the segment. After that, the client updates the statistic of the download rate (exponential moving average); if necessary, posts client-status information to the Tracker; and sleeps until the publishing time of the next segment to download.

The ABR strategy reported in the algorithm 1 initially checks the need of downscaling the coding rate. When client i of clique k measures a download rate D_i lower than R_k, it must downscale to avoid playout buffer depletion. For the sake of simplicity, we devised only procedures to handle single node downscaling events, whose consequences are clique-join or clique-creation. Starting from the coding rate immediately lower than D_i and going down, the strategy checks the sustainability of the related clique event and, if verified, selects that rate and fetches the segment. Obviously, this procedure also handles the case in which more clients of a clique must concurrently downscale towards the same rate, but in this case the downscale might be too aggressive, albeit effective.

If downscaling is not necessary, the logic verifies the possibility of upscaling, giving preference to upscales that imply: clique-merging/switching, clique-join, clique-creation. In the latter case, the client attempts to upscale to the immediately higher coding rate, when granted by a traditional backoff procedure. The backoff limits the concurrence of attempting procedures, that could falsify the measurement of the available server-proxy bandwidth. If no upscale is sustainable, the client maintains the current quality.

To realize a scalable system versus the number of clients, we limit the number of HTTP GET and POST messages exchanged between clients and Tracker as follows.

The URL used by clients to download the swarm-status

Algorithm 1 ABR with segment download

```
i = client index
lastSn = last downloaded segment number
serverSn = last segment number published by server
x = sequence number of the segment to download
k = coding rate index of previous segment
0..M = coding rate index range
D_i = average download rate of client i
BO = upscaling attempt backoff value
BO = BO − 1
Downscaling
if D_i < R_k then
    g = (max coding rate index ≤ D_i)
    for j ∈ g..1 do
        if clique j is not null then                    ▷ join
            Download and return segment at rate R_j
        else if succ. attempt for rate R_j then         ▷ creation
            Return attempted segment at rate R_j
        end if
    end for
    Download and return segment at rate R_0
end if
Upscaling
if k < M then
    if min(D_j) ≥ R_{k+1} then                          ▷ switching/merging
        g = (max coding rate index ≤ min(D_j))
        Download and return segment at rate R_g
    end if
    if D_i ≥ R_{k+1} then                               ▷ join
        g = (max coding rate index ≤ D_i)
        for j ∈ g..(k + 1) do
            if clique j is not null then
                Download and return segment at rate R_j
            end if
        end for
    end if
    if BO == 0 then                                     ▷ creation
        extract new random BO value
        if successful attempt for rate R_{k+1} then
            Return attempted segment at rate R_{k+1}
        end if
    end if
end if
Maintain quality
Download and return segment at rate R_k
```

contains a Nonce equal to the number of the last segment published by the source. In doing so, at each publishing time all clients concurrently send out an HTTP GET for the same URL, and the proxy processes these requests by using only one request to the HTTP server.

A client reports to the Tracker a "quantized" measurement (`bw`) of its average download rate. An HTTP POST occurs only if the quantized value is different from the one already present at the Tracker, or if a refresh timeout (e.g. 50 sec) is elapsed. The greater the quantization step, the lower the probability of needing an update and thus the number of HTTP POSTs. However, a too big quantization step may prevent the exploitation of all available coding rates. As a trade-off we used a quantization step equal to the minimum difference among available video coding rates: 1 Mbit/s in our case.

V. Experimental assessment

We carried out an experimental campaign in a laboratory environment and on the real Internet, reproducing the configuration of Fig. 2. The laboratory environment is formed by virtual machines hosting: video clients, a Squid proxy, an Apache HTTP server, an Apache Tomcat running the Tracker [15]. HTTP server and Tomcat are on the same node. In the real Internet case we used the PlanetLab infrastructure.

We streamed in live mode the Big Buck Bunny movie, encoded with six representations, whose coding rates are 0.55, 1.5, 2.5, 3.5, 4.5, 8.6 Mbit/s. Segment duration is 4s, the number of segments is 160. We implemented five Python-based clients with the following ABR strategies:

- Buffer-based (BB): a single step upscale occurs when the playout buffer reaches 3/4 of its capacity and a single step downscale occurs when the buffer goes below 1/3 of its capacity.
- Throughput-based (TB): the coding rate is selected as the highest one lower than the average download rate.
- DASHJS-based (DJ): we used the ABR rules of the dash.js v1.3.0 [3] reference player, namely: ThroughputRule (TR), BufferOccupancyRule (BR), InsufficentBufferRule (IBR). These rules are concurrently used. The latter two have priority with respect to the former. TR is like the previous TB one. As a consequence of BR, the coding rate switches to the maximum one when the buffer is greater than a RICH-threshold (3/4 of playout buffer). Due to IBR, the coding rate switches to the lowest one when the playout buffer is empty[4].
- PROBE-based (PB): it is the ABR logic of [7]. It is a TB strategy, but before performing an upscale to the TB coding rate, the client probes the server-client bandwidth to verify the actual availability of such a rate.
- Tracker-assisted (TKR): it is our proposed strategy, described in sec. IV.

[4]DASHIF is now working on the new version v.1.4.0 where these strategies will change. However, v1.4.0 was not stable at the time of writing

A. Laboratory evaluation

Basic throughput-based strategies (e.g. TB) yields quality oscillation when the server-proxy path is the network bottleneck (Fig. 2), and when there is a non-perfect synchronization among live video clients, as it may occur in a realistic Internet environment. We prove this fact and show the behavior of ABR strategies in these conditions.

We force the server-proxy bandwidth to be the network bottleneck by using Linux Traffic Control tools, and we reproduce a "desync" condition among clients by differentiating their clocks within a desync range $[0, MD]$ ms. In case of V clients, the clock difference between client 1 and client v is equal to $(v-1)MD/(V-1)$, thus the last client gets the max desync value MD ms.

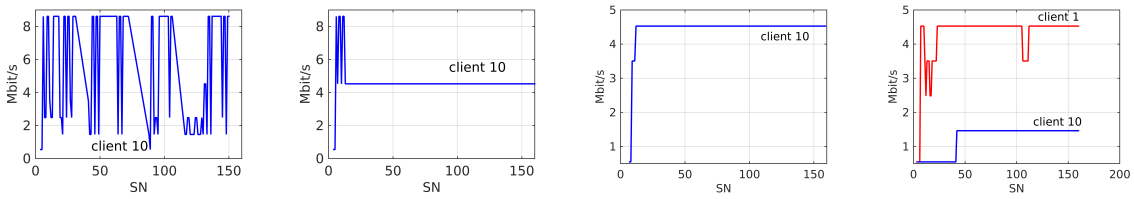
We analyzed performance versus server-proxy bandwidth, proxy-client bandwidth, number of clients and desync range. We set the playout buffer time to 4 segments. For lack of space we only report a subset of results.

Fig. 8 shows the coding rate oscillation phenomenon in the case of TB strategy and how instead it is solved by our TKR strategy and by the PB strategy [7]. The server-proxy bandwidth is limited to 8 Mbit/s. Fig. 8(a) reports the coding rate of (last) client n. 10 in the case of the TB strategy and max desync = 500ms. Coding rate oscillation occurs and it is rather wide in terms of bandwidth variation. Fig. 8(b) shows the coding rate of the same client in absence of desync. After an initial transitory phase, the coding rate reaches the stable optimal value of 4.5 Mbit/s. Indeed, the highest coding rate of 8.6 Mbit/s is not sustainable by the server bandwidth. We point out that the other clients, not reported in the figure, follow the same behavior. Therefore, in absence of desync, oscillations do not show up and even the simple TB strategy provides good and fair performance.

Fig. 8(c) shows that our TKR strategy avoids oscillations also in presence of desync. Albeit not reported, all other clients have the same behavior; indeed, during the test, the clients concurrently upscaled by performing clique-switchings. Fig. 8(d) shows that also the PB strategy solves the oscillation issue. However, since clients do not share status information as in our TKR, they are not always able to reach a fair behavior. Indeed, client 1 and 10 have both unlimited access bandwidth but stabilize their coding rate to rather different values.

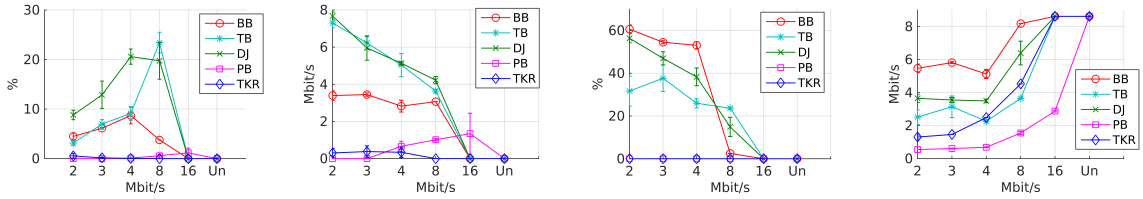
The motivation behind this *unfair deadlock* of the PB strategy is the following. During the PB test, we observed two streams on the server-proxy path, whose coding rate were 1.5 and 4.5 Mbit/s, respectively. When a client performs a probe, there are three connections (two for video streams and one for probe) and TCP fair share bandwidth is close to 8/3=2.66 Mbit/s for each of them, including network headers. But the application layer bandwidth measured by the probing procedure of PB is lower than 2.5 Mbit/s and prevents any upscale from 1.5 Mbit/s.

Fig. 9 reports performances by varying the server-proxy bandwidth, in case of 10 clients with unlimited access bandwidth and max desync = 500 ms. Fig. 9(a) reports the percentage of quality switching, measured as the ratio between



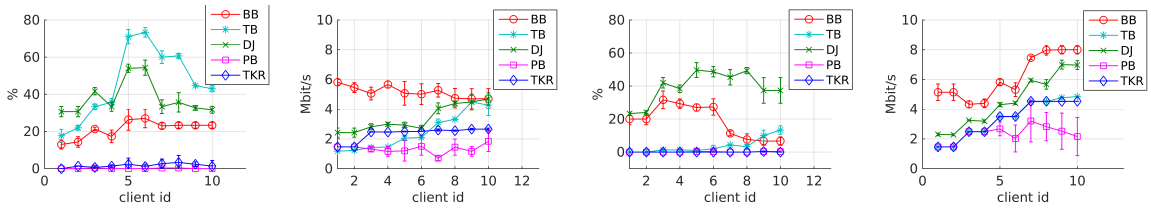
(a) TB, max desync 500ms (b) TB, no de-sync (c) TKR, max desync 500ms (d) PB, max desync 500ms

Fig. 8. Coding rate vs segment number, server-proxy bandwidth 8 Mbit/s, proxy-client unlimited, 10 clients



(a) % quality switching (b) quality switch rate variation (c) % segment loss (d) coding rate

Fig. 9. Perf. vs server-proxy bandwidth, 10 clients, max desync = 500 ms, proxy-client bandwidth unlimited, 90% conf. int.



(a) % quality switching (b) coding rate (c) % quality switching (d) coding rate

Fig. 10. Perf. vs client id, 10 clients, max desync = 500 ms, server-proxy bandwidth = 8 Mbit/s (a,b) and 16 Mbit/s (c,d), proxy-client bandwidth heterogeneous: clients 1,2 = 2 Mbit/s, clients 3,4 = 3 Mbit/s, clients 5,6 = 4 Mbit/s, clients 7,8 = 8 Mbit/s, clients 9,10 = 16 Mbit/s; 90% conf. int.

the average number of quality switching per-client and the total number of segments. Fig. 9(b) shows the average coding rate variation per quality-switch. Fig. 9(c) reports the average percentage of lost segments per client, and Fig. 9(d) the average coding rate per client. As expected, in the case of unlimited server-proxy bandwidth (Un), independently from the desync value, all strategies get the same, optimal, result: no switching, no loss, and maximum coding rate. Problems show up for lower server bandwidths. The BB, TB and DJ strategies show quality switching, with wide bandwidth oscillation per switch and very high segment loss. These problems are practically avoided by the PB and TKR strategies, but the TKR strategy achieves a coding rate about equal to two times the one achieved by PB.

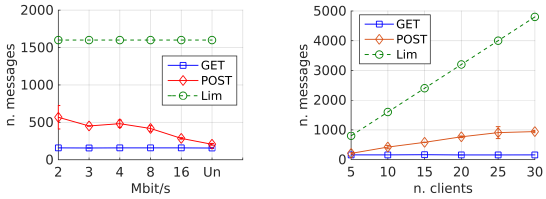
We also consider more realistic cases of clients with limited and *heterogeneous* proxy-client bandwidths. In each test we deployed five set of clients, with the following per-set rates: 2,3,4,8,16 Mbit/s. The number of clients per set is uniform.

We repeated the measurements of Fig. 9 and achieved similar performance, even though the coding rate gain that TKR achieves over PB is lower, due to the constraints on client bandwidths. For lack of space we do not report these results, but prefer to show in Fig. 10 a per-client analysis in the case of server-proxy bandwidth of 8 Mbit/s and 16 Mbit/s.

In terms of quality switching and segment loss (not reported) our TKB strategy achieves fair values among clients, close to zero. Regarding the coding rate, we observe that TKR gets a max-min fairness condition. In the case of server-proxy bandwidth equal to 8 Mbit/s (Fig. 10(b)), the first two clients select a coding rate of 1.5 Mbit/s and the other clients a coding rate of 2.5 Mbit/s. Higher rates (e.g. 3.5 Mbit/s) would be possible for clients 5..10, but not sustainable on the server-proxy bandwidth, considering protocol overheads. When the server-proxy bandwidth increases to 16 Mbit/s (Fig. 10(d)), clients 5,6 reach a coding rate of 3.5 Mbit/s, and the remaining clients select 4.5 Mbit/s. The highest coding rate 8.6 Mbit/s would be possible for clients 9,10, but not sustainable on the server-proxy bandwidth.

With respect to TKR, some other strategies achieve higher coding rates but, in these cases, with the drawbacks of an higher number of quality switching and segment loss. In addition, performance are unfair among clients.

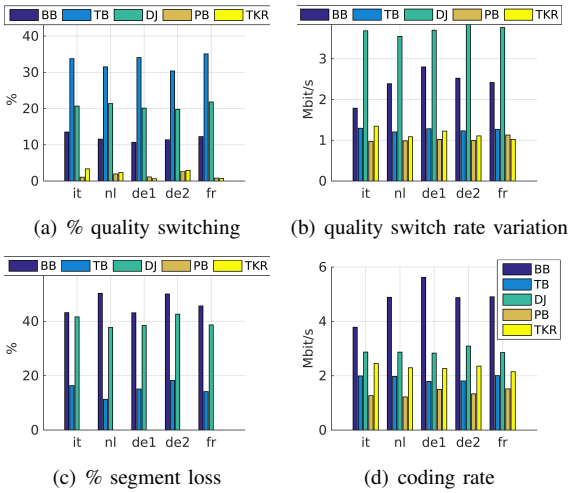
Fig. 11 shows a scalability performance of the TKR strategy, measured as the number of GET and POST messages received and handled by the TRACKER. We vary the server access bandwidth (Fig. 11(a)) and the number of clients (Fig. 11(b)), with a heterogeneous proxy-client bandwidth configuration. Segment-by-segment clients GET the swarm status and, if



(a) vs server-proxy bandwidth and 10 clients (b) vs n.clients and server-proxy bandwidth = 8000 kbit/s

Fig. 11. Tracker messages for heterogeneous proxy-client bandwidth, 90% conf. int.

necessary, POST their client status. Consequently, without any optimization, the number of GET/POST procedures is equal to the number of segments (160) multiplied by the number of clients. This is the 'Lim' curve in Fig. 11. We observe that, thanks to the exploitation of proxy/cache also for Tracker related interactions, the number of GET messages remains unchanged and equal to 160 in both analysis. As regards the number of POSTs, when increasing the server-proxy bandwidth there is a light decrease of the number of POSTs, due to the fact that more and more stable measurements of download rate reduce the number of necessary updates. If we vary the number of clients we have an increase of POSTs, but the related value is rather below the Lim one.



(a) % quality switching (b) quality switch rate variation

(c) % segment loss (d) coding rate

Fig. 12. Perf. vs PlanetLab Web Server, 10 clients, proxy-client bandwidth heterogeneously limited, server output bandwidth limited to 8 Mbit/s

B. Performance evaluation in the Internet

Fig. 12 reports measurements done by using by 15 PlanetLab nodes. Ten nodes are used as clients, while five other nodes are alternatively used as Web servers (X axis indicates node country). The Proxy is located in our University. To reproduce bandwidth limitations that Internet users may have on the access section (e.g. ADSL, mobile, etc.) we limit the proxy-client bandwidth as in the previous heterogeneous configuration. To reproduce lack of Web Server bandwidth (e.g. server serving several proxies, different videos, etc.) we limit the Web Server output bandwidth to 8 Mbit/s. Clearly, both bounds are effective when the actual Internet bandwidth is greater than them. We do not force any desync, since it is naturally produced by the Internet environment.

Also in these tests the PB and TKR strategies achieve better performances than other strategies. The number of switching is very limited. When a switch occurs, it is almost always between nearby representations (i.e., rate variation of about 1 Mbit/s), segment loss never shows up and, as in the laboratory tests, TKR provides a better coding rate than PB.

VI. Conclusions

In this paper we proposed to add a Tracker functionality in the MPEG DASH architecture, and showed how it is possible to exploit this functionality to improve live video streaming performance in presence of proxy/cache. This is only one of the possible applications of the Tracker, other examples including a P2P functionality to scale down server load in absence of proxy/cache [9].

The integration of a Tracker in the MPEG DASH architecture can be done without modifying the MPEG DASH standard. Tracker information (e.g. IP address:port) and protocol used by clients to interact with it can be embedded in the player, which in turn can be pushed by the server as a javascript. Thus, our proposal only needs server-side software. However, to support heterogeneity of players, new standard MPD metadata should include tracker information and dedicated specifications should describe Tracker protocols.

References

[1] I. Sodagar, "The MPEG-DASH standard for Multimedia Streaming over the Internet," *IEEE MultiMedia*, no. 4, pp. 62–67, 2011.

[2] DASH Industry Forum. [Online]. Available: http://dashif.org/

[3] dash.js reference client implementation. [Online]. Available: http://dashif.org/reference/players/javascript

[4] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai network: a platform for high-performance Internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[5] K. Miller, E. Quacchio, G. Gennari, and A. Wolisz, "Adaptation algorithm for adaptive streaming over http," in *Packet Video Workshop (PV), 2012 19th International*. IEEE, 2012, pp. 173–178.

[6] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, "A buffer-based approach to rate adaptation: Evidence from a large video streaming service," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 187–198.

[7] C. Mueller, S. Lederer, and C. Timmerer, "A proxy effect analyis and fair adatpation algorithm for multiple competing dynamic adaptive streaming over http clients," in *Visual Communications and Image Processing (VCIP), 2012 IEEE*. IEEE, 2012, pp. 1–6.

[8] D. H. Lee, C. Dovrolis, and A. C. Begen, "Caching in http adaptive streaming: Friend or foe?" in *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*. ACM, 2014, p. 31.

[9] Y. Zhang and N. Zong, "Problem statement and requirements of the peer-to-peer streaming protocol (ppsp)," 2013.

[10] T. Wiegand, G. J. Sullivan, G. Bjøntegaard, and A. Luthra, "Overview of the h. 264/avc video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 560–576, 2003.

[11] DASH-IF INTEROPERABILITY POINTS AND EXTENSIONS. [Online]. Available: http://dashif.org/guidelines

[12] S. Akhshabi, A. C. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http," in *Proceedings of the second annual ACM conference on Multimedia systems*. ACM, 2011, pp. 157–168.

[13] Z. Aouini, M. T. Diallo, A. Gouta, A.-M. Kermarrec, and Y. Lelouedec, "Improving caching efficiency and quality of experience with cf-dash," in *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*. ACM, 2014, p. 61.

[14] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen, "Server-based traffic shaping for stabilizing oscillating adaptive streaming players," in *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM, 2013, pp. 19–24.

[15] Source code of players and trackers. [Online]. Available: http://netgroup.uniroma2.it/Andrea_Detti/DASHTracker/infocom2016.tar.gz