TAPAS: a Tool for rApid Prototyping of Adaptive Streaming algorithms

Luca De Cicco
Politecnico di Bari, Italy
luca.decicco@poliba.it

Vito Caldaralo
Quavlive srl, Italy
vito.caldaralo@quavlive.com

Vittorio Palmisano
Quavlive srl, Italy
vpalmisano@quavlive.com

Saverio Mascolo
Politecnico di Bari, Italy
mascolo@poliba.it

ABSTRACT

The central component of any adaptive video streaming system is the stream-switching *controller*. This paper introduces TAPAS, an open-source Tool for rApid Prototyping of Adaptive Streaming control algorithms. TAPAS is a flexible and extensible video streaming client written in python that allows to easily design and carry out experimental performance evaluations of adaptive streaming controllers without needing to write the code to download video segments, parse manifest files, and decode the video. TAPAS currently supports DASH and HLS and has been designed to minimize the CPU and memory footprint so that experiments involving a large number of concurrent video flows can be carried out using a single client machine. An adaptive streaming controller is implemented to illustrate the simplicity of the tool along with a performance evaluation which validates the tool.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: [General]

Keywords

Adaptive streaming, DASH, HLS, rapid prototyping

1. INTRODUCTION

Today video streaming applications generate more than half of the global Internet traffic [3]. The choice of using the *progressive download streaming* (PDS) approach made by YouTube in 2005 has been the main driver of this growth. With the PDS technique the video is encoded at a given bitrate and it is sent to the user through a HTTP connection using a TCP socket and played using a web browser.

An evolution of PDS is the *adaptive streaming*. With this technique the video content bitrate and resolution can be dynamically varied to match the network available bandwidth and user screen resolution. The two leading standardization efforts related to adaptive streaming are the *HTTP Live Streaming* (HLS) proposed by Apple[1] and the ISO standard *Dynamic Adaptive Streaming over HTTP* (MPEG-DASH) [10]. Both the standards require the video content to be encoded at different bitrates and resolutions, the *video levels* or *representations*. Then, each video level is logically or physically divided into *segments*, or *chunks*, of fixed durations. A manifest file[2] is stored at the server and is used to associate to each (chunk,video level) pair its corresponding URL. The client downloads and *parses* the manifest file and constructs a data structure that is used to request consecutive video segments of possibly different representations. A *controller* dynamically decides, for each video segment, the video level to be streamed to achieve the best possible quality given the available bandwidth with the constraint of avoiding video interruptions. Both HLS and MPEG-DASH require the controller to be implemented at the client and to use HTTP requests to fetch the video segments from standard HTTP servers.

The controller is a key component of any video streaming system having the overall goal of maximizing the user QoE. Towards this end, a careful design phase is required which typically follows an iterative and incremental development method made of two phases which form a cycle: 1) design and implementation of the controller; 2) experimental performance evaluation to check whether the specifics are met. In order to carry out the experimental evaluation, a testbed has to be set-up and a complete adaptive video streaming player, incorporating the controller, has to be implemented. Developing an adaptive video streaming player is a time consuming process even for the experienced programmer and requires several modules to be designed and implemented.

In this paper we propose TAPAS, a *Tool for rApid Prototyping Adaptive Streaming* controllers, that allows to only focus on controller design rather than having to spend time on the complex task of implementing a complete adaptive video player. After the design and experimentation phase is completed, the controller can be implemented and integrated in a production adaptive video streaming player. Our over-

VideoNext'14, December 02-05 2014 Sydney, Australia.

http://dx.doi.org/10.1145/2676652.2676654 .

[1]R. Pantos and W. May. HTTP Live Streaming. IETF Draft, June 2012.

[2]In the case of HLS a set of *extended M3U8* playlists is used. In this paper we will use the terms manifest or playlist interchangeably.

all goal is to make TAPAS a reference tool for the research community in order to foster experiment reproducibility.

The tool has been designed to allow a large number of concurrent video streaming sessions to be launched on the same machine. To make this possible, TAPAS allows to disable decoding of the video so that both CPU load and RAM usage can be contained. Another important feature is that, being TAPAS written in Python, the design of a new controller requires only to extend a base controller class without having to compile the code. TAPAS has been tested to be compliant with the video samples provided in the DASH dataset [7] and the YouTube DASH dataset [1].

The DASH Industry Forum[3] (DASH-IF) initiative collects several reference web players supporting DASH for both HTML5 and Flash. Even though these players are complete, they all have to be run in web browsers, making experimentation of the control algorithm difficult. In particular, experimenting with web based video players limits the number of concurrent videos that can be run from the same machine due to memory and CPU requirements. Moreover, instrumentation of the experiment and results logging may be a complex task. The most closely related work to ours is [9] where authors present a plugin written in C++ to enable DASH on the popular VLC[4] media player. The plugin is written to allow the implementation of controllers by extending a base class. However, since the video stream has to be decoded, the tool proposed in [9] cannot be used to run experiments in which a large number of video streaming in parallel is involved due to excessively high CPU and RAM requirements.

2. ARCHITECTURE

The stream-switching controller is the key component of any adaptive video streaming system; for each video segment s_i it chooses the video level l_j in the discrete *set of available video levels* $\mathscr{L}$ with the overall goal of maximizing the user QoE in terms of: 1) *video playout continuity*: interruptions due to buffer underruns should be avoided and video level switching frequency should be minimized; 2) *video quality*: the video level should be maximized; 3) *latency*: the start-up delay, i.e. the time elapsed to start the video stream playback, should be minimized.

After the control algorithm has been designed, a careful experimental evaluation phase is required to verify that the design requirements are met. To the purpose, a full adaptive video streaming player has to be implemented comprising at least the following modules: 1) a *parser* module that parses the manifest file; 2) a *downloader* that fetches the video segments from the HTTP server and feeds them to the player playout buffer; 3) a *media engine* that plays the video stored in the playout buffer; the media engine drains the playout buffer and feeds the video segments to a decoder; 4) the *controller* that chooses the video level and communicates to the downloader module the video segment to download.

TAPAS[5] is an open source tool written in *python*[6] language conceived to simplify both the design and experimentation of adaptive streaming algorithms. TAPAS employs the `GStreamer`[7] multimedia framework for decoding and playing the received video. In particular, it provides a full adaptive video streaming player that is able to play video streams using both DASH and HLS standards. The unique feature of TAPAS is its flexibility: new control logics and manifest parsers can be implemented by inheriting base classes as it will be shown in the following. In this manner, the researcher can focus only on the design of the controller by leveraging all the other implemented components.

[3] `http://dashif.org`
[4] `http://www.videolan.org/vlc/`
[5] `http://c3lab.poliba.it/index.php/Tapas`
[6] `http://www.python.org`
[7] `http://gstreamer.freedesktop.org/`

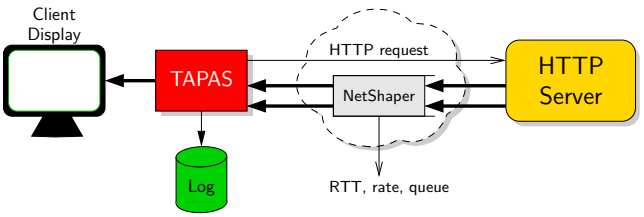

Figure 1: A typical controlled testbed

Algorithm 1 Tapas player

```
1  c = Controller(ctrl_options)
2  p = Parser(url_playlist)
3  m = MediaEngine(media_options)
4  player = TapasPlayer(controller=c, parser=p
5                       media=m, other_options)
6  player.play()
```

Figure 1 shows a typical controlled testbed that can be used to carry out an experimental evaluation of a stream-switching control algorithm. The testbed can be implemented using only two hosts: the first machine runs TAPAS that requests the video chunks to a standard HTTP server placed on the second host. On the second host a bandwidth shaper similar to *dummynet* can be employed to emulate a WAN scenario in which the end-to-end bandwidth, the base RTT of the link, and the bottleneck queue can be freely adjusted. It is important to notice that TAPAS can be used in any other scenario provided that the videos are made available through a HTTP server. We have tested TAPAS to be compliant with many webTVs employing HLS or DASH.

`TapasPlayer` is implemented by *aggregation* of three interacting components: 1) the `Controller` that selects the video level of the next segment to be downloaded; 2) the `Parser`, that parses the video manifest file; 3) the `MediaEngine` that stores the downloaded video in the *playout buffer* and plays the video. Each of these three components can be extended individually by inheriting the corresponding base class (see Section 3 for further details). Moreover, it is worth to mention that `TapasPlayer` includes an extensible module that periodically logs in a file the variables of interest.

Algorithm 1 shows how an adaptive video player can be implemented with TAPAS. The first step (lines 1-3) is to create the three aforementioned components. Then the components are passed to the `TapasPlayer` constructor (line 4) and the player is created. The playback of the video is started by issuing the `play()` method of `TapasPlayer` (line 6).

Figure 2 shows the workflow of `TapasPlayer`. When the `play()` method is issued the `Parser` downloads the manifest and populates two lists of dictionaries, i.e. `playlists` and `levels`: the first one maintains information about segments (or chunks), the second one holds video levels (or representations) information. The two lists of dictionaries

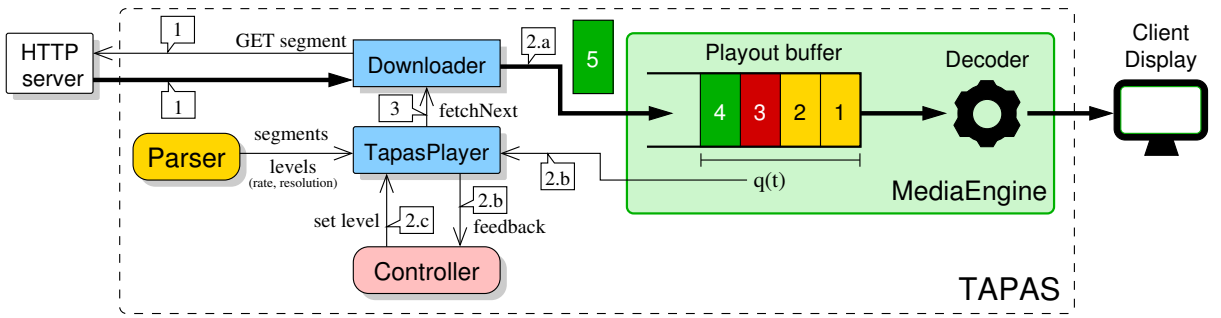

Figure 2: TAPAS workflow

are then passed to the player. At this point two concurrent threads are started: 1) a thread that fills the playout buffer by fetching the video segments from the HTTP server and 2) a thread that drains the playout buffer to play the video stream.

Let us now focus on the thread filling the buffer. The following operations are executed in a loop until the last video segment has been played:

1. The `Downloader` fetches from the HTTP server the current segment at the selected video level.
2. When the download is completed the following operations are performed:
   (a) The segment is enqueued in the playout buffer handled by the `MediaEngine` component.
   (b) The player gets from the `MediaEngine` the *queue length* and other feedbacks and builds the *player feedback* dictionary with this information. Then *player feedback* is passed to the `Controller`.
   (c) The `Controller` computes two values: 1) the *control action*, i.e. the video level rate of the next segment to be downloaded; 2) the *idle duration*, possibly equal to zero, that is the time interval that has to elapse before the next video segment can be fetched.
3. a timer of duration *idle duration* is started. When the timer expires the loop repeats from step 1.

Finally, the thread draining the playout buffer is handled by the `MediaEngine` that decodes the compressed video frames, and plays the raw video.

3. COMPONENTS

In the following we describe the essential features of the three components of TAPAS.

3.1 Parser

In this section we describe the main fields (Section 3.1.1) and methods (Section 3.1.2) that a parser has to implement by inheriting the base class `BaseParser`. The main task of a parser is to populate and update two data structures, `playlists` and `levels`, which are used by `TapasPlayer`. In particular, in the case of *live streaming* the `TapasPlayer` calls the parser to continuously update the two data structures, whereas in the case of *video on demand* (VoD) the two data structures are populated only once.

	VoD	Live	Containers	Supported dataset
DASH	Yes	No	MP4	ITEC [7], YouTube [1]
HLS	Yes	Yes	MPEG-TS	Any *m3u8* playlist

Table 1: Implemented parsers

Field	Description
`url`	the video level base URL
`is_live`	true if the video is a live stream
`segments`	a list of dictionaries. Each dictionary contains the segment URL (`segment_url`), its duration (`segment_duration`) and the byte range[8]
`start_index`	index of the first chunk to be downloaded by the `Downloader`
`end_index`	index of the last chunk of the current playlist
`duration`	the duration (in seconds) of the playlist

Table 2: `playlists` fields

Table 1 reports the parsers implemented in TAPAS at the time of writing along with their main features.

3.1.1 Fields

In the following we give the essential details about the data structures of `playlists` and `levels`.

The `playlists` data structure is a list of dictionaries, one for each available video level $l_i \in \mathscr{L}$. Table 2 reports the main fields of such dictionaries along with their description. In particular, the field `end_index` has different meanings depending on the type of video stream: 1) in the case of live streaming (`is_live` is true) it is the index of the last chunk of the current playlist; in this case `end_index` increases when a new chunk is made available by the the HTTP server; 2) in the case of video on demand (`is_live` is false) it represents the index of the last chunk of the video. Moreover, the field `duration` represents the duration of the video stream in the case of VoD, whereas in the case of live video it represents the duration of the available video segments.

The `levels` data structure is a list of dictionaries, one for each available video level. The dictionary has to include the following keys: 1) `rate`: is the encoding rate of the video

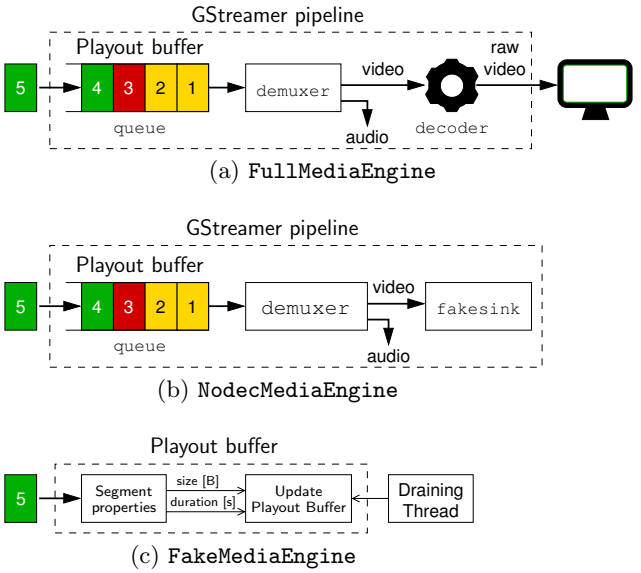

Figure 3: Implemented media engines

level measured in bytes/s; 2) `resolution`: is the video level resolution.

3.1.2 Methods

A parser must implement two methods: 1) the `start` method, which fetches and parses the manifest to populate the `playlists` and `levels` data structures, and the `updateLevelSegmentsList` method, which updates the `playlists` structure.

The `TapasPlayer` calls the parser `start` method when its `play` method is issued. For each available video level $l_i \in \mathscr{L}$, this method has to use `updateLevelSegmentsList(`l_i`)` to populate the `segments` list of the `playlists` data structure. The `updateLevelSegmentsList` method is also used by `TapasPlayer` to fetch and update the `segments` list at run time in the case of a live video streaming session.

Finally, it is important to notice that the implementation of the logic to fetch and parse the playlist is specific to the parser and thus is not implemented in `BaseParser`.

3.2 Media Engine

The `MediaEngine` provides methods to fill/drain its playout buffer and to decode and play the video stream. A `MediaEngine` must extend the base class `BaseMediaEngine` and implement at least the following methods: `start`, `stop`, `pushData`, and `getQueuedTime`.

The `start` and `stop` methods are used to respectively initialize the `MediaEngine` and stop the video playback. The `MediaEngine` starts the playback of the video when the playout buffer length reaches a minimum threshold `min_queue_time` specified in the `MediaEngine` constructor.

The `pushData` method is called by `TapasPlayer` to fill the playout buffer on the completion of a video segment download.

The `getQueuedTime` method must return the playout buffer length measured in seconds. This method can be used by the `Controller` to compute a control action based on the playout buffer length.

The logic for draining the buffer, decode and play the video stream depends on the particular media engine. TAPAS already includes three media engines that allow to give different levels of detail to the experimental evaluation: the `FullMediaEngine` is a complete player that decodes and renders the raw video to the screen; the `NodecMediaEngine` is a player that only demuxes the video stream without decoding and rendering the video; the `FakeMediaEngine` only keeps track of the playout buffer length, but does not demux, decode, and render the video. In particular the `NodecMediaEngine` and `FakeMediaEngine` do not decode the video to keep the CPU and memory utilization low while ensuring that the playout buffer dynamics is identical to that of `FullMediaEngine`. This is a fundamental feature since the main dynamics of an adaptive video streaming system is that of the playout buffer: firstly, several controllers calculate the control action based on the playout buffer queue length [4, 11], secondly, the playout buffer dynamics also determines the re-buffering events which are known to be a key factor impairing the QoE [5]. Section 5 provides an experimental validation that proves this important feature.

In the following we give more details about the three media engines.

3.2.1 FullMediaEngine

Figure 3 (a) shows the block diagram of the `FullMediaEngine` which leverages a GStreamer pipeline made of three elements: the `queue`, the `demuxer`, and the `decoder`. In particular, the `TapasPlayer` enqueues the downloaded video segments in a GStreamer `queue` element by calling the media engine `pushData` method. The `demuxer` is a GStreamer element that demuxes the segment in video and audio buffers. The video buffers are fed to the `decoder` that decodes the compressed video and renders the raw video to the screen. This media engine implements the `getQueuedTime` and the `getQueuedBytes` methods by using the GSstreamer `queue` element properties. Finally, the implementation of the `start` and `stop` methods respectively start and stop the GStreamer pipeline. At the time of writing, `FullMediaEngine` supports the H.264 codec and the MP4 and MPEG-TS containers.

Since this media engine requires demuxing an decoding the video stream, its CPU and memory requirements are similar to those of a media player such as VLC. For this reason this media engine cannot be used to produce a large number of concurrent flow using a single machine.

3.2.2 NodecMediaEngine

Figure 3 (b) shows the block diagram of the `NodecMediaEngine`. This media engine is similar to `FullMediaEngine` in that it uses a GStreamer pipeline to maintain the playout buffer (`queue` element) and demux the segments (`demuxer` element). This media engine is designed to drastically reduce the CPU and memory requirements, since it does not decode the video stream. Towards this end, the demuxed video is simply discarded by using a GStreamer element called `fakesink`. It is important to notice that the playout buffer is drained exactly as in the case of `FullMediaEngine`, since the `fakesink` element exactly emulates the behaviour of a video player. The methods of this media engine are exactly the same of those implemented by `FullMediaEngine`.

3.2.3 FakeMediaEngine

Figure 3 (c) shows the block diagram of the `FakeMediaEngine`. In this case the playout buffer does not store the video segments and it is implemented as a simple data struc-

ture which holds two fields `seconds` and `bytes`, maintaining respectively the playout buffer length measured in seconds and in bytes. In order to obtain the dynamics of the playout buffer two dynamics have to be emulated: the filling and the draining of the buffer. The buffer is filled when a new segment has been downloaded, whereas it is continuously drained when the video is playing [4].

Let us analyze how the buffer is filled. When a segment is enqueued using the `pushData` method, the "Segment properties" function returns the segment size, measured in bytes, and duration, measured in seconds. This information is available by leveraging on the data structures provided by the `Parser` component. Then, this information is passed to an "update playout buffer" module that adds to `seconds` and `bytes` respectively the segment size measured in seconds and in bytes.

In order to emulate the buffer draining dynamics, we employ a thread that every T seconds decreases the playout buffer length of T seconds[9]. A similar procedure is used to decrease the `bytes` field.

This media engine has two advantages: 1) it can be used regardless of the employed video container and codec provided that the `Parser` is able to parse the manifest file; 2) similarly to `NodecMediaEngine` it significantly decrease the CPU and memory requirements.

3.3 Controller

The controller is the central component of the adaptive video streaming system. Its goal is to decide the video level, among those advertised in the manifest files, based on *feedbacks,* such as the estimated bandwidth or the playout buffer length, and the player *state*. Typically, an adaptive video streaming controller can be in two different states: *buffering* or *steady state.* When in buffering, the client requests a new segment right after the previous has been downloaded in order to quickly build up the player queue; on the other hand, during the steady state an *idle period* has to elapse to request a new video segment after the last segment download has been completed [2, 8, 6].

In order to implement a controller, the `BaseController` class has to be inherited and two methods must be implemented: 1) `calcControlAction`, computing the control action, and 2) `isBuffering`, which decides – according to the implemented logic – whether the state of the system is *buffering* or *steady state*. The two methods are described in the following.

The `calcControlAction` must implement the control logic which returns a rate in bytes/s. Towards this end, the controller employs the `feedback` field that is a dictionary storing several feedback information such as the playout buffer length and the estimated bandwidth. In particular, when the download of a segment is completed, the `TapasPlayer` updates this dictionary and calls `calcControlAction` to get the video level to be used for the next segment download. Moreover, in `calcControlAction` the `setIdleDuration` method must be used to set the idle time between the download of two consecutive segments. The output of `calcControlAction` is passed to `quantizeRate`, an overloadable method that selects the video level index according to the computed control action. In its default implementation the quantizer selects the highest video level below the rate computed by `calcControlAction`.

[9]By default $T = 0.1$s.

Algorithm 2 `ConventionalController` implementation

```
1  class ConventionalController(BaseController):
2    def calcControlAction(self):
3      T = self.feedback['last_download_time']
4      cur = self.feedback['cur_rate']
5      tau = self.feedback['fragment_duration']
6      x = cur * tau / T
7      y = self.ewma_filter(x)
8      self.setIdleDuration(tau-T)
9      return y
10   def isBuffering(self):
11     return self.feedback['queued_time'] < self.Q
12   def quantizeRate(self,rate):
13     ...
14     return level
15   def ewma_filter(self, rate):
16     ...
17     return filtered_rate
```

Finally, the `isBuffering` method must implement a logic that returns `True` if the system is considered in buffering state. Recall that in the buffering state the idle period is equal to zero. A typical implementation of this method returns `True` if the playout buffer length is below a given threshold.

4. RAPID PROTOTYPING WITH TAPAS

This section provides an example showing how an adaptive streaming controller can be implemented. To the purpose we consider a simple controller, named *conventional*, that is described in details in [8].

In a nutshell, the k-th controller output is equal to the a filtered version y_k of the bandwidth samples. The k-th bandwidth sample x_k is computed as $x_k = \tau r_{k-1}/T_{k-1}$ where r_{k-1} is the video level rate of the last downloaded segment, τ is the segment duration in seconds, and T_{k-1} is the time spent to download the last segment. The bandwidth samples x_k are then filtered with an exponential weighted moving average (EWMA) giving the filtered bandwidth samples y_k computed as $y_k = y_{k-1} - T_{k-1}\alpha(y_{k-1} - x_k)$ where $\alpha > 0$ is the filter parameter. The video level index of the next video segment to be downloaded is the output of a quantization function $Q(\cdot)$ (see [8] for more details). Finally, when the system is in *steady state* the controller sets the *idle period* as $\max(\tau - T_{k-1}, 0)$ (see Section 3.3).

Algorithm 2 shows the implementation of `ConventionalController` which extends `BaseController`. Due to space constraint we are not able to show the implementation of all the methods, but we focus on `calcControlAction` and `isBuffering`. Lines 3-5 get T, r and τ from the `feedback` structure. Then, in line 6 the bandwidth estimate `x` is computed and it is filtered in line 7 by the `ewma_filter` method. Line 8 sets the idle period using the method `setIdleDuration`. Finally, the `isBuffering` method is a boolean condition that is true if the playout buffer length is less than a threshold `Q`.

5. EXPERIMENTAL EVALUATION

This section validates the three media engines described in Section 3.2. In particular, we want to show that the dynamics of the playout buffer queue and of the control action is identical for the three considered media engines. The experiments were carried out by using the testbed scenario shown in Figure 1, where the client and the server are connected using through a 100 Mbps ethernet connection. TAPAS runs

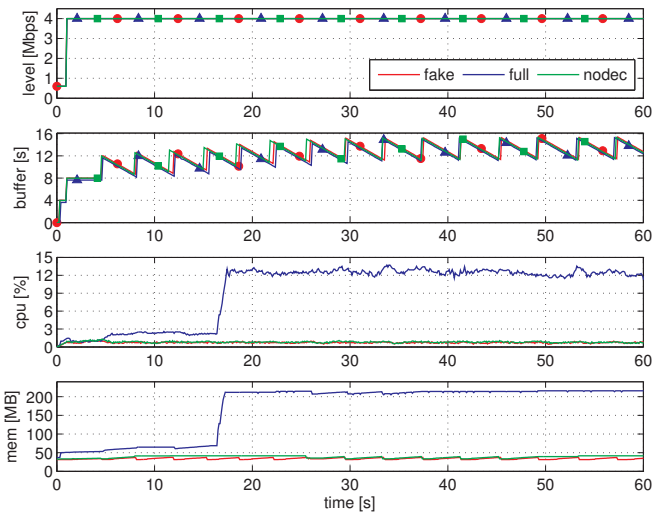

Figure 4: Dynamics of a single video flow

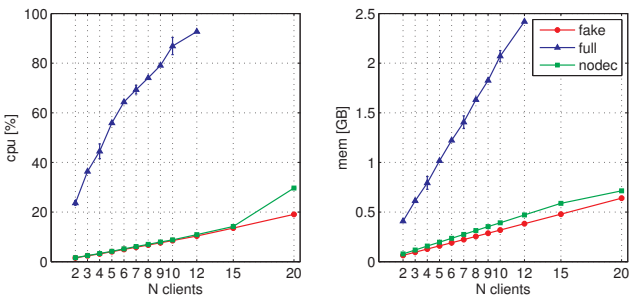

Figure 5: CPU and Memory load with an increasing number of client

on a DELL precision T1650 workstation with 16GB of RAM and an Intel Xeon E3-1270 3.50GHz processor.

Figure 4 compares the dynamics of the video level, the playout buffer length, the CPU load and the memory occupation, for each of the three considered media engines. The figure clearly shows that the three media engines produce exactly the same playout buffer length and video level dynamics, validating the `NodecMediaEngine` and `FakeMediaEngine` implementations. The figure also shows that the CPU load and the memory occupation in the case of `FullMediaEngine` is much higher *wrt* those provided by the other two media engines. In particular, the CPU and memory utilizations are comparable until around $t = 17$s where the player starts to decode the video level 4 that is a 1080p video. For $t > 17$s the CPU utilization in the case of `FullMediaEngine` increases to 12% whereas in the case of the two other media engines it always keeps less than 1%. Similarly, the memory occupation increases when the player starts to decode the 1080p video segments. In fact, the video player has to temporarily store in the RAM the decoded raw video before being rendered by the CPU. It is worth to notice that the CPU and memory utilization obtained using TAPAS with `FullMediaEngine` are similar to those obtained when using a VLC player.

The `NodecMediaEngine` and `FakeMediaEngine` provide similar CPU and memory utilizations, the latter requiring slightly less RAM due to the fact that it does not store the compressed video segments.

To conclude this section, we measure the CPU and memory requirements as a function of the number of concurrent video sessions running on the same machine. Figure 5 shows that in the case of `FullMediaEngine` the CPU gets overloaded already at 12 concurrent videos. On the other hand, the other two media engines are able to stream 20 concurrent video sessions with around a 20% CPU utilization. Similarly, the RAM occupation are much higher in the case of `FullMediaEngine`, that in the case of 12 concurrent video flow already requires 2.5GB, whereas the other two media engines require less than 650MB.

6. CONCLUSIONS

In this paper we have presented TAPAS, an open-source extensible tool written in python for rapid prototyping of adaptive streaming algorithms. We have described its architecture and the essential features of all its modules. TAPAS has been designed to allow experimental evaluations also involving a large number of video streaming sessions. A simple controller has been implemented to show the simplicity of designing an adaptive streaming control algorithm with TAPAS. An experimental evaluation has shown that TAPAS is able to significantly reduce CPU and memory requirements *wrt* standard players, while producing the same playout buffer and video level dynamics.

7. ACKNOWLEDGMENTS

This project has been made possible in part by the gift CG #574954 from the Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation. This work has been also partially supported by the Italian Ministry of Education, Universities and Research (MIUR) through the MAIVISTO project (PAC02L1_00061). Any opinions, findings, conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect the views of the funding agencies.

8. REFERENCES

[1] Mpeg-dash media source demo. http://dash-mse-test.appspot.com/media.html.

[2] S. Akhshabi et al. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. *Proc. of ACM MMSys 2011*, pages 157–168, 2011.

[3] Cisco. Cisco Visual Networking Index:Forecast and Methodology 2013-2018. *White Paper*, June 2014.

[4] L. De Cicco et al. ELASTIC: a Client-side Controller for Dynamic Adaptive Streaming over HTTP (DASH). In *Packet Video Workshop '13*, pages 1–8, 2013.

[5] F. Dobrian et al. Understanding the impact of video quality on user engagement. In *Proc. of ACM SIGCOMM 2011*, pages 362–373, 2011.

[6] J. Jiang et al. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proc. of CoNEXT '12*, pages 97–108, 2012.

[7] S. Lederer et al. Dynamic adaptive streaming over http dataset. In *Proc. of ACM MMSYS '12*, pages 89–94, 2012.

[8] Z. Li et al. Probe and Adapt: Rate Adaptation for HTTP Video Streaming At Scale. *IEEE J. on Selected Areas in Communications*, 32(4):719–733, April 2014.

[9] C. Müller and C. Timmerer. A VLC media player plugin enabling dynamic adaptive streaming over HTTP. In *Proc. of ACM Multimedia*, pages 723–726, 2011.

[10] I. Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE MultiMedia*, 18(4):62–67, 2011.

[11] G. Tian and Y. Liu. Towards agile and smooth video adaptation in dynamic HTTP streaming. In *Proc. of ACM CoNeXT '12*, pages 109–120. ACM, 2012.