Multimedia Systems manuscript No.
(will be inserted by the editor)

Performance Evaluation of TcpHas: TCP for HTTP Adaptive Streaming

Chiheb Ben Ameur · **Emmanuel Mory** · **Bernard Cousin** · **Eugen Dedu**

Received: date / Accepted: date

Abstract HTTP Adaptive Streaming (HAS) is a widely used video streaming technology that suffers from a degradation of user's Quality of Experience (QoE) and network's Quality of Service (QoS) when many HAS players are sharing the same bottleneck link and competing for bandwidth. The two major factors of this degradation are: the large OFF period of HAS, which causes false bandwidth estimations, and the TCP congestion control, which is not suitable for HAS given that it does not consider the different video encoding bitrates of HAS.

This paper proposes a HAS-based TCP congestion control, TcpHas, that minimizes the impact of the two aforementioned issues. It does this by using traffic shaping on the server. Simulations indicate that TcpHas improves both QoE, mainly by reducing instability and convergence speed, and QoS, mainly by reducing queuing delay and packet drop rate.

Keywords HTTP Adaptive Streaming; TCP Congestion Control; Cross-layer Optimization; Traffic Shaping; Quality of Experience; Quality of Service.

1 Introduction

Video streaming is a widely used service. According to 2016 Sandvine report [32], in North America, video and audio streaming in fixed access networks accounts for over 70% of the downstream bandwidth in evening hours. Given this high usage, it is of extreme importance to optimize its use. This is usually done by adapting the video to the available bandwidth. Numerous adaptation methods have been proposed in the literature and by major companies, and their differences mainly rely on the entity that does the adaptation (client or server), the variable used for adaptation (the network or sender or client buffers), and the protocols used; the major companies having finally opted for HTTP [16].

HTTP Adaptive Streaming (HAS) is a streaming technology where video contents are encoded and stored at different qualities at the server and where players (clients) can choose

Authors' addresses: C. Ben Ameur, (Current address) INNES, ZAC Atalante Champeaux, 5A rue Pierre Joseph Colin, 35000 Rennes, France; E. Mory, Orange Labs, 4 rue du Clos Courtel, 35510 Cesson Sévigné, France; B. Cousin, IRISA, University of Rennes 1, 263 av. Général Leclerc, 35000 Rennes, France; E. Dedu, FEMTO-ST Institute, Numérica, cours Leprince-Ringuet, 25200 Montbéliard, France.

periodically the quality according to the available resources. Popular HAS-based methods are Microsoft Smooth Streaming, Apple HTTP Live Streaming, and MPEG DASH (Dynamic Adaptive Streaming over HTTP). Still, *this technology is not optimal for video streaming*, mainly because its HTTP data is transported using the TCP protocol. Indeed, video data is encoded at distinct bitrates, and TCP does not increase the throughput sufficiently quickly when the bitrate changes. TCP variants (such as Cubic, Illinois, and Westwood+) specific to high bandwidth-delay product networks achieve high bandwidth more quickly and seem to give better performance for HAS service than classical TCP variants such as NewReno and Vegas [9], but the improvement is limited.

Another reason for this suboptimality is the highly periodic ON–OFF activity pattern specific to HAS [2]. Currently, a HAS player estimates the available bandwidth by computing the download bitrate for each chunk at the end of the download (for the majority of players, this estimation is done by dividing the chunk size by its download duration). As such, it is impossible for a player to estimate the available bandwidth when no data is being received, i.e. during OFF periods. Moreover, when several HAS stream compete in the same home network, bandwidth estimation becomes more difficult. For example, if the ON period of a player coincides with the OFF period of a second player, the first player will overestimate its available bandwidth, and makes it select for the next chunk a quality level higher than in reality. This, in turn, could lead to a congestion event if the sum of the downloading bitrates of the two players exceeds the available bandwidth of the bottleneck. An example is given in [9] (table 4): the congestion rate for two competing HAS clients is considerably reduced when using a traffic shaping. Finally, unstable quality levels are harmful to user's Quality of Experience (QoE) [33]. Traffic shaping, which expands the ON periods and shrinks the OFF periods, can considerably limit the drawbacks mentioned above [1, 21, 7, 35, 3, 8].

One method to reduce occurrences of ON–OFF patterns is to use server-based shaping at application layer [3]. This approach is cross-layer because it interacts with the TCP layer and its parameters such as the congestion window, *cwnd*, and the round-trip time estimation, *RTT*. Hence, implementing HAS traffic shaping at the TCP level is naturally more practical and easier to manage; in addition, this should offer better bandwidth share among HAS streams, reduce congestion events and improve the QoE of HAS users.

Despite the advantages of using a transport layer-based method for HAS, and in contrast with other types of streaming, where methods at the transport layer have already been proposed (RTP, Real-time Transport Protocol, and TFRC, TCP Friendly Rate Control [17]), to the best of our knowledge, *there is no proposition at the transport level specifically designed for HAS*. For commercial video providers YouTube, Dailymotion, Vimeo and Netflix, according to [19], "The quality switching algorithms are implemented in the client players. A player estimates the bandwidth continuously and transitions to a lower or to a higher quality stream if the bandwidth permits." The streaming depends on many parameters, such as player, video quality, device and video service provider etc., and uses various techniques such as several TCP connections, variable chunk sizes, different processing for audio and video flows, different throttling factors etc. To conclude, all these providers use numerous techniques, all of them based on client.

Therefore, in this paper, we extend our previous work [10] by proposing a HAS-oriented TCP congestion control variant, TcpHas, that aims to minimize the aforementioned issues (TCP throughput insufficient increase and ON-OFF pattern) and to unify all the techniques given in the previous paragraph. It uses four sub-modules: bandwidth estimator, optimal quality level estimator, *ssthresh* adjusting, and *cwnd* adjusting to the shaping rate. Simulation results show that TcpHas considerably improves both QoS (queuing delay, packet drop

rate) and QoE (stability, convergence speed), performs well with several concurrent clients, and does not cause stalling events.

The remainder of this paper is organized as follows: Section 2 presents server-based shaping methods and describes possible optimizations at TCP level. Then, Section 3 describes TcpHas congestion control and Section 4 evaluates it. Section 5 concludes the article.

2 Background and related works

Our article aims to increase QoE and QoS by fixing the ON–OFF pattern. Many server-based shaping methods have been proposed in the literature to improve QoE and QoS of HAS. Their functioning is usually separated into two modules:

1. Estimation of the optimal quality level, based on network conditions, such as bandwidth, delay, and/or history of selected quality levels, and available encoding bitrates of the video.
2. The shaping function of the sending rate, which should be suitable to the encoding bitrate of the estimated optimal quality level.

The next two subsections describe constraints and proposed solutions for each module. The last subsection presents some possible ways of optimization, which provides the basis for the TcpHas design.

2.1 Optimal Quality Level Estimation

A major constraint of optimal quality level estimation is that the server has no visibility on the set of flows that share the bottleneck link.

Ramadan et al. [31] propose an algorithm to reduce the oscillations of quality during video adaptation. During streaming, it marks each quality as unsuccessful or successful, depending on whether it has led to lost packets or not. A successfulness value is thus attached to each quality, and is updated regularly using an EWMA (Exponential Weighted Moving Average) algorithm. The next quality increase is allowed if and only if its successfulness value does not exceed some threshold. We note that, to discover the available bandwidth, this method increases throughput and pushes to packet drop, which is different from our proposed method, where the available bandwidth is computed using an algorithm.

Akhshabi et al. [3] propose a server-based shaping method that aims to stabilize the quality level sent by the server by detecting oscillation events. The shaping function is activated only when oscillations are detected. The optimal quality level is based on the history of quality level oscillations. Then, the server shapes its sending rate based on the encoding bitrate of the estimated optimal quality level. However, when the end-to-end available bandwidth increases, the HAS player cannot increase its quality level when the shaper is activated. This is because the sending bitrate is limited on the server side and when the end-to-end available bandwidth increases, the player is still stable on the same quality level that matches the shaping rate. To cope with that, the method deactivates the shaping function for some chunks and uses two TCP parameters, *RTT* and *cwnd*, to compute the connection throughput that corresponds to the end-to-end available bandwidth ($\frac{cwnd}{RTT}$). If the estimated bandwidth is higher than the shaping rate, the optimal quality level is increased to the next higher quality level and the shaping rate is increased to follow its encoding bitrate. We note

that this method is implemented in the application layer. It takes as inputs the encoding bitrates of delivered chunks and two TCP parameters (*RTT* and *cwnd*). The authors indicate that their method stabilizes the competing players inside the same home network without significant bandwidth utilization loss.

Accordingly, the optimal quality estimation process is based on two different techniques: quality level oscillation detection and bandwidth estimation using the throughput measurement. The former is based on the application layer information (i.e., the encoding bitrate of the actual and previous sent chunks) and is sufficient to activate the shaping function (i.e., the shaper). However, to verify whether the optimal quality level has been increased or not, the server is obliged to deactivate the shaper to let the TCP congestion control algorithm occupy the remaining capacity available for the HAS stream.

Although this proposed method offers performance improvements on both QoE and QoS, the concept of activating and deactivating the shaper is not sufficiently solid, especially against unstable network conditions, and raises a number of open questions about the duration of deactivation of the traffic shaping and its impact on increasing the OFF period duration. In addition, this method is not proactive and the shaper is activated only in the case of quality level oscillation.

What is missing in this proposed method is a good estimation of the available bandwidth for the HAS flow. This method relies on the throughput measurement during non-shaped phases. If a bandwidth estimation less dependent on *cwnd* could be given, we could keep the shaper activated during the whole HAS stream and adapt the estimation of optimal quality level to the estimation of available bandwidth.

2.2 Traffic Shaping Methods

Ghobadi et al. propose a shaping method on the server side called *Trickle* [18]. It was proposed for YouTube in 2011, when it adopted progressive download technology. Its key idea is to place a dynamic upper bound on *cwnd* such that TCP itself limits the overall data rate. The server application periodically computes the *cwnd* bound from the product between the round-trip time (*RTT*) and the target streaming bitrate. Then it uses a socket option to apply it to the TCP socket. Their results show that Trickle reduces the average *RTT* by up to 28% and the average TCP loss rate by up to 43%. However, HAS differs from progressive download by the change of encoding bitrate during streaming. Nevertheless, Trickle can also be used with HAS by adapting the *cwnd* bound to the encoding bitrate of each chunk.

We note that the selection of the shaping rate by the server-based shaping methods does not mean that the player will automatically start requesting that next higher quality level [3]. The transition to another shaping rate may take place several chunks later, depending on the player's bitrate controller and the server-based shaping method efficiency.

Furthermore, it was reported [9] that *ssthresh* has a predominant effect on the convergence speed of the HAS client to select the desired optimal quality level. Indeed, when *ssthresh* is set higher than the product of shaping rate and *RTT*, the server becomes aggressive and causes congestions and a reduction of quality level selection on the player side. In contrast, when *ssthresh* is set lower than this product, *cwnd* takes several *RTT*s to reach the value of this product, because in the congestion avoidance phase the increase of *cwnd* is relatively slow (one *MSS*, Maximum Segment Size, each *RTT*). Consequently, the server becomes conservative and needs a long time to occupy its selected shaping rate. Hence, the player would have difficulties reaching its optimal quality level.

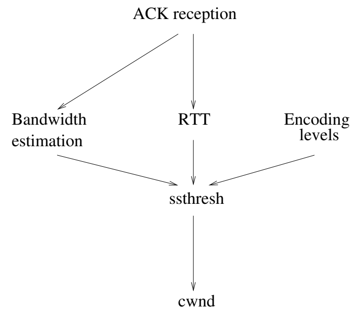


Fig. 1 TCP parameter setting process for quality level selection.

Accordingly, shaping the sending rate by limiting *cwnd*, as described in Trickle, has a good effect on improving the QoE of HAS. However, it is still insufficient to increase the reactivity of the HAS player and consequently to accelerate the convergence speed. Hence, to improve the performance of the shaper, *ssthresh* needs to be modified too. The value of *ssthresh* should be set at the right value that allows the server to quickly reach the desired shaping rate.

2.3 Optimization of Current Solutions

What can be noted from the different proposed methods for estimating the optimal quality level is that an efficient end-to-end estimator of available bandwidth can improve their performance, as shown in Subsection 2.1. In addition, the only parameter from the application layer needed for shaping the HAS traffic is the encoding bitrate of each available quality level of the corresponding HAS stream. As explained in Subsection 2.2, the remaining parameters are found in the TCP layer: the congestion window *cwnd*, the slow-start threshold *ssthresh*, and the round-trip time *RTT*. We are particularly interested in adjusting *ssthresh* to accelerate the convergence speed. This is summed up in figure 1.

Naturally, what is missing here is an efficient TCP-based method for end-to-end bandwidth estimation. We also need a mechanism that adjusts *ssthresh* based on the output of the bandwidth estimator scheme. Both this mechanism and estimation schemes used by various TCP variants are introduced in the following.

2.3.1 Adaptive Decrease Mechanism

In the literature, we found a specific category of TCP variants that set *ssthresh* using bandwidth estimation. Even if the estimation is updated over time, TCP uses it only when a congestion event is detected. The usefulness of this mechanism, known as *adaptive decrease mechanism*, is described in [27] as follows: "*the adaptive window setting provides a congestion window that is decreased more in the presence of heavy congestion and less in the presence of light congestion or losses that are not due to congestion, such as in the case of losses due to unreliable links*". This low frequency of *ssthresh* updating (only after congestion detection) is justified in [12] by the fact that, in contrast, a frequent updating of *ssthresh* tends to force TCP into congestion avoidance phase, preventing it from following the variations in the available bandwidth.

Hence, the unique difference of this category from the classical TCP congestion variant is only the adaptive decrease mechanism when detecting a congestion, i.e., when receiving three duplicated ACKs or when the retransmission timeout expires. This mechanism is described in Algorithm 1.

Algorithm 1 TCP adaptive decrease mechanism.

```
1: if 3 duplicate ACKs are received then
2:     ssthresh = Bwe^ × RTT_min
3:     if cwnd > ssthresh then
4:         cwnd = ssthresh
5:     end if
6: end if
7: if retransmission timeout expires then
8:     ssthresh = Bwe^ × RTT_min
9:     cwnd = initial_cwnd
10: end if
```

▷ where $\widehat{Bwe}$ is the estimated bandwidth and RTT_{min} is the lowest RTT measurement

We remark that the algorithm uses the estimated bandwidth, $\widehat{Bwe}$, multiplied by RTT_{min} to update the *ssthresh* value. The use of RTT_{min} instead of the actual *RTT* is justified by the fact that RTT_{min} can be considered as an estimation of *RTT* of the connection when the network is not congested.

2.3.2 Bandwidth Estimation Schemes

The most common TCP variant that uses bandwidth estimation to set *ssthresh* is Westwood. Other newer variants have been proposed, such as Westwood+ and TIBET (Time Intervals-based Bandwidth Estimation Technique). The only difference between them is the bandwidth estimation scheme used. In the following, we introduce the different schemes and describe their performance.

Westwood estimation scheme [28]: The key idea of Westwood is that the source performs an end-to-end estimation of the bandwidth available along a TCP connection by measuring the rate of returning acknowledgments [28]. It consists of estimating this bandwidth by properly filtering the flow of returning ACKs. A sample of available bandwidth Bwe_k is computed each time t_k the sender receives an ACK:

$$Bwe_k = \frac{d_k}{t_k - t_{k-1}} \tag{1}$$

where d_k is the amount of data acknowledged by the ACK that is received at time t_k. d_k is determined by an accurate counting procedure by taking into consideration delayed ACKs, duplicate ACKs and selective ACKs. Then, the bandwidth samples Bwe_k are low-pass filtered by using a discrete-time low-pass filter to obtain the bandwidth estimation $\widehat{Bwe_k}$. The low-pass filter employed is generally the exponentially-weighted moving average function:

$$\widehat{Bwe_k} = \gamma \times \widehat{Bwe_{k-1}} + (1-\gamma) \times Bwe_k \tag{2}$$

where $0 \leq \gamma \leq 1$. Low-pass filtering is necessary because congestion is due to low-frequency components of the available bandwidth, and because of the delayed ACK option [24, 26].

However, this estimation scheme is affected by the *ACK compression* phenomenon. This phenomenon occurs when the time spacing between the received ACKs is altered by the congestion of the routers on the return path [39]. In fact, when ACKs pass through one congested router, which generates additional queuing delay, they lose their original time spacing because during forwarding they are spaced by the short ACK transmission time [12]. The result is ACK compression that can lead to bandwidth overestimation when computing the bandwidth sample Bwe_k. Moreover, the low-pass filtering process is also affected by ACK compression because it cannot filter bandwidth samples that contain a high-frequency component [26]. Accordingly, the ACK compression causes a systematic bandwidth overestimation when using the Westwood bandwidth estimation scheme. ACK compression is commonly observed in real network operation [30] and thus should not be neglected in the estimation scheme.

Another phenomenon that distorts the Westwood estimation scheme is *clustering*: As already noted [12, 39], the packets belonging to different TCP connections that share the same link do not intermingle. As a consequence, many consecutive packets of the same connection can be observed on a single channel. This means that each connection uses the full bandwidth of the link for the time needed to transmit its cluster of packets. Hence, a problem of fairness between TCP connections is experienced when the estimation scheme does not take the clustering phenomenon into consideration and continues to estimate the bandwidth of the whole shared bottleneck link instead of their available bandwidth.

Westwood+ estimation scheme [29]: To estimate correctly the bandwidth and alleviate the effect of ACK compression and clustering, a TCP source should observe its own link utilization for a time longer than the time needed for entire cluster transmission. For this purpose, Westwood+ modifies the bandwidth estimation (*Bwe*) mechanism to perform the sampling every *RTT* instead of every ACK reception as follows:

$$Bwe = \frac{d_{RTT}}{RTT} \tag{3}$$

where d_{RTT} is the amount of data acknowledged during one RTT. As indicated in [29], the result is a more accurate bandwidth measurement that ensures better performance when compared with NewReno and it is still fair when sharing the network with other TCP connections. *Bwe* is updated once per RTT. The bandwidth estimation samples are low-pass filtered to give a better smoothed estimation of $\widehat{Bwe}$.

However, the amount of acknowledged data during one RTT (d_{RTT}) is bounded by the sender's window size, $\min(cwnd, rwnd)$, which is defined by the congestion control algorithm. In fact, $\min(cwnd, rwnd)$ defines the maximum amount of data to be transmitted during one RTT. Consequently, the bandwidth estimation of Westwood+, given by each sample *Bwe*, is still always lower than the sender sending rate ($Bwe \leq \frac{\min(cwnd,rwnd)}{RTT}$).

Hence, although the Westwood+ estimation scheme reduces the side effects of ACK compression and clustering, it is still dependent on the sender sending rate rather than the available bandwidth of the corresponding TCP connection.

TIBET estimation scheme [12, 11]: TIBET (Time Interval-based Bandwidth Estimation Technique) is another technique that gives a good estimation of bandwidth even in the presence of packet clustering and ACK compression.

The basic idea of TIBET is to perform a run-time sender-side estimate of the average packet length and the average inter-arrival separately. The bandwidth estimation scheme is applied to the stream of the received ACKs and is described in Algorithm 2 [12], where *acked* is the number of segments acknowledged by the last ACK, *packet_size* is the average

segment size in bytes, *now* is the current time and $last_ack_time$ is the time of the previous ACK reception. $Average_packet_length$ and $Average_interval$ are the low-pass filtered measures of the packet length and the interval between sending times.

Algorithm 2 Bandwidth estimation scheme.

1: **if** *ACK is received* **then**
2: $sample_length = acked \times packet_size \times 8$
3: $sample_interval = now - last_ack_time$
4: $Average_packet_length = alpha \times Average_packet_length + (1 - alpha) \times sample_length$
5: $Average_interval = alpha \times Average_interval + (1 - alpha) \times sample_interval$
6: $Bwe = Average_packet_length / Average_interval$
7: **end if**

$Alpha$ ($0 \leq alpha \leq 1$) is the pole of the two low-pass filters. The value of $alpha$ is critical to TIBET performance: If $alpha$ is set to a low value, TIBET is highly responsive to changes in the available bandwidth, but the oscillations of *Bwe* are quite large. In contrast, if $alpha$ approaches 1, TIBET produces more stable estimates, but is less responsive to network changes. Here, we note that if $alpha$ is set to zero we have the Westwood bandwidth estimation scheme, where the sample *Bwe* varies between 0 and the bottleneck bandwidth.

TIBET estimation scheme uses a second low-pass filtering, with parameter γ, on the estimated available bandwidth *Bwe* to give a better smoothed estimation $\widehat{Bwe}$, as described in Equation 2. γ is a variable parameter, equal to e^{-T_k}, where $T_k = t_k - t_{k-1}$ is the time interval between the two last received ACKs. This means that bandwidth estimation samples *Bwe* with high T_k values are given more importance than those with low T_k values.

Simulations [12] indicate that TIBET gives bandwidth estimations very close to the correct values, even in the presence of other UDP flows with variable rates or other TCP flows.

3 TcpHas Description

As shown in the previous section, a protocol specific to HAS needs to modify several TCP parameters and consist of several algorithms. Our HAS-based TCP congestion control, TcpHas, is based on the two modules of server-based shaping solution: optimal quality level estimation and sending traffic shaping itself, both with two submodules. The first module uses a bandwidth estimator submodule inspired by the TIBET scheme and adapted to HAS context, and an optimal quality level estimator submodule to define the quality level, $\widehat{QLevel}$, based on the estimated bandwidth. The second module uses $\widehat{QLevel}$ in two submodules that update respectively the values of *ssthresh* and *cwnd* over time.

This section progressively presents TcpHas by describing the four submodules, i.e., the bandwidth estimator, the optimal quality level estimator, *ssthresh* updating process, and *cwnd* value adaptation to the shaping rate.

3.1 Bandwidth Estimator of TcpHas

As described in Section 2, TIBET performs better than other proposed schemes. It reduces the effect of ACK compression and packet clustering and is less dependent on the congestion window than Westwood+.

The parameter γ used by TIBET to smooth *Bwe* estimations (see Equation 2) is variable and equal to e^{-T_k}. However, this variability is not suited to HAS. Indeed, when the HAS stream has a large OFF period, the HTTP GET request packet sent from client to server to ask for a new chunk is considered by the server as a new ACK. As a consequence, the new bandwidth estimation sample, *Bwe*, will have an underestimated value and γ will be reduced. Hence, this filter gives higher importance to an underestimated value to the detriment of the previous better estimations. For example, if the OFF period is equal to 1 second, γ will be equal to 0.36, which means that a factor of 0.64 is given to the new underestimated value in the filtering process. Consequently, the smoothed bandwidth estimation, $\widehat{Bwe}$, will be reduced at each high OFF period. However, the objective is rather to maintain a good estimation of available bandwidth, even in the presence of large OFF periods. For this purpose, we propose to make parameter γ constant.

Hence, the bandwidth estimator of TcpHas is the same as in the TIBET bandwidth estimation scheme, except for the low-pass filtering process: we use a constant value of γ instead of e^{-T_k} as defined by TIBET.

3.2 Optimal Quality Level Estimator of TcpHas

TcpHas' optimal quality level estimator is based on the estimated bandwidth, $\widehat{Bwe}$, described in Subsection 3.1. This estimator is a function that adapts HAS features to TCP congestion control and replaces $\widehat{Bwe}$ value by the encoding bitrate of the estimated optimal quality level $\widehat{QLevel}$. One piece of information from the application layer is needed: the available video encoding bitrates, which are specified in the index file of the HAS stream. In TcpHas they are specified, in ascending order, in the *EncodingRate* vector. TcpHas' estimator is defined by the function *QLevelEstimator*, described in Algorithm 3, which selects the highest quality level whose encoding bitrate is equal to or lower than the estimated bandwidth, $\widehat{Bwe}$.

Algorithm 3 QLevelEstimator function.

1: **for** $i = length(EncodingRate) - 1$ **downto** 0 **do**
2:     **if** $EncodingRate[i] \leq \widehat{Bwe}$ **then**
3:         $\widehat{QLevel} = i$
4:         **return**
5:     **end if**
6: **end for**
7: $\widehat{QLevel} = 0$

$\widehat{QLevel}$ parameter is updated only by this function. However, the time and frequency of its updating is a delicate issue:

- We need to use the adaptive decrease mechanism (see Algorithm 1), because when a congestion occurs $\widehat{QLevel}$ needs to be updated to the new network conditions. Hence, this function is called after each congestion detection.
- Given that TcpHas performs a shaping rate that reduces $\widehat{OFF}$ occupancy, when TcpHas detects an $\widehat{OFF}$ period, it may mean that some network conditions have changed (e.g. an incorrect increase of the shaping rate). Accordingly, to better estimate the optimal quality level, this function is called after each $\widehat{OFF}$ period.

The *EncodingRate* vector is also used by TcpHas during application initialization to differentiate between a HAS application and a normal one: when the application returns an empty vector, it is a normal application, and TcpHas just makes this application be processed by classical TCP, without being involved at all.

3.3 Ssthresh Modification of TcpHas

The TCP variants that use the TCP decrease mechanism use RTT_{min} multiplied by the estimated bandwidth, $\widehat{Bwe}$, to update *ssthresh*. However, given that the value of *ssthresh* affects the convergence speed, it should correspond to the desired shaping rate instead of $\widehat{Bwe}$. Also, the shaping rate is defined in Trickle [18] to be 20% higher than the encoding bitrate, which allows the server to deal better with transient network congestion.

Hence, for TcpHas we decided to replace $\widehat{Bwe}$ by $EncodingRate[\widehat{QLevel}] \times 1.2$, which represents its shaping rate:

$$ssthresh = EncodingRate[\widehat{QLevel}] \times RTT_{min} \times 1.2 \quad (4)$$

The timing of *ssthresh* updating is the same as that of $\widehat{QLevel}$: when detecting a congestion event and just after an idle $\widehat{OFF}$ period. Moreover, the initial value of *ssthresh* should be modified to correspond to the context of HAS. These three points are presented in the following.

3.3.1 Congestion Events

Inspired by Algorithm 1, the TcpHas algorithm when detecting a congestion event is described in Algorithm 4. It includes the two cases of congestion events: three duplicated ACKs, and retransmission timeout. In both cases, $\widehat{Qlevel}$ is updated from $\widehat{Bwe}$ using the *QLevelEstimator* function. Then, *ssthresh* is updated according to Equation 4. The update of *cwnd* is as in Algorithm 1.

Algorithm 4 TcpHas algorithm when congestion occurs.

```
1: if 3 duplicate ACKs are received then
2:     QLevel^ = QLevelEstimator(Bwe^)
3:     ssthresh = EncodingRate[QLevel^] × RTT_min × 1.2
4:     if cwnd > ssthresh then
5:         cwnd = ssthresh
6:     end if
7: end if
8: if retransmission timeout expires then
9:     QLevel^ = QLevelEstimator(Bwe^)
10:    ssthresh = EncodingRate[QLevel^] × RTT_min × 1.2
11:    cwnd = initial_cwnd
12: end if
```

3.3.2 Idle Periods

As explained in [4, 8, 9], the congestion window is reduced when the idle period exceeds the retransmission timeout *RTO*, and *ssthresh* is updated to $\max(ssthresh, 3/4 \times cwnd)$. In HAS context, the idle period coincides with the OFF period. In addition, we denote by $\widehat{OFF}$ the OFF period whose duration exceeds *RTO*. Accordingly, reducing *cwnd* after an $\widehat{OFF}$ period will force *cwnd* to switch to slow-start phase although the server is asked to deliver the video content with the optimal shaping rate.

To avoid this, we propose to remove the *cwnd* reduction after the $\widehat{OFF}$ period. Instead, as presented in Algorithm 5, TcpHas updates $\widehat{Qlevel}$ and *ssthresh*, then sets *cwnd* to *ssthresh*. This modification is very useful in the context of HAS. On the one hand, it eliminates the sending rate reduction after each $\widehat{OFF}$ period, which adds additional delay to deliver the next chunk and may cause a reduction of quality level selection on the player side. On the other hand, the update of *ssthresh* each $\widehat{OFF}$ period allows the server to adjust its sending rate more correctly, especially when the client generates a high $\widehat{OFF}$ period between two consecutive chunks.

Algorithm 5 TcpHas algorithm after an $\widehat{OFF}$ period.

```
1: if idle > RTO then                                   ▷ OFF^ period detected
2:     QLevel^ = QLevelEstimator(Bwe^)
3:     ssthresh = EncodingRate[QLevel^] × RTT_min × 1.2
4:     cwnd = ssthresh
5: end if
```

3.3.3 Initialization

By default, TCP congestion control uses an initial value of *ssthresh*, *initial_ssthresh*, of 65535 bytes. The justification comes from the TCP classical goal to occupy quickly (exponentially) the whole available end-to-end bandwidth.

However, in HAS context, *initial_ssthresh* is better to match an encoding bitrate. We decided to set it to the highest quality level at the beginning of streaming for two reasons: 1) to give a similar initial aggressiveness as classical TCP and 2) to avoid setting it higher than the highest encoding bitrate to maintain the HAS traffic shaping concept.

This initialization should be done in conformity with Equation 4, hence the computation of *RTT* is needed. Consequently, TcpHas just updates the *ssthresh* when the first *RTT* is computed. In this case, our updated *ssthresh* serves the same purpose as *initial_ssthresh*. TcpHas initialization is presented in Algorithm 6.

3.4 Cwnd Modification of TcpHas for Traffic Shaping

As shown in Subsection 2.2, Trickle does traffic shaping on the server-side by setting a maximum threshold for *cwnd*, equal to the shaping rate multiplied by the current *RTT*. However, during congestion avoidance phase (i.e., when $cwnd > ssthresh$), *cwnd* is increased very slowly by one MSS each *RTT*. Consequently, when *cwnd* is lower than this threshold, it takes several *RTTs* to reach it, i.e. a slow reactivity.

Algorithm 6 TcpHas initialization.

```
1: QLevel^ = length(EncodingRate) − 1                          ▷ the highest quality level
2: cwnd = initial_cwnd
3: ssthresh = initial_ssthresh                                  ▷ i.e. 65535 bytes
4: RTT = 0
5: if newACK is received then
6:     if RTT ≠ 0 then                                         ▷ i.e. when the first RTT is computed
7:         ssthresh = EncodingRate[QLevel^] × RTT × 1.2
8:     end if
9: end if
```

To increase its reactivity, we modify TCP congestion avoidance algorithm by directly tuning *cwnd* to match the shaping rate. Given that TcpHas does not change its shaping rate ($EncodingRate[\widehat{QLevel}] \times 1.2$) during the congestion avoidance phase (see Subsection 3.2), we update *cwnd* according to the *RTT* variation. However, in this case, we are faced to the following dilemma related to *RTT* variation:

- On the one hand, the increase of *RTT* means that queuing delay increases and could cause congestion when the congestion window is still increasing. Worse, if the standard deviation of *RTT* is important (e.g., in the case of a wireless home network, or unstable network conditions), an important jitter of *RTT* would force *cwnd* to increase suddenly and cause heavy congestion.
- On the other hand, the increase of *RTT* over time should be taken into account by the server in its *cwnd* updating process. In fact, during the ON period of a HAS stream, the *RTT* value is increasing [25]. Consequently, using a constant value of RTT (such as RTT_{min}) does not take into consideration this increase of *RTT* and may result in a shaping rate lower than the desirable rate.

One way to mitigate *RTT* fluctuation and to take into account the increase of *RTT* during the ON period is to use smoothed *RTT* computations. We propose to employ a low-pass filter for this purpose. The smoothed *RTT* that is updated at each ACK reception is:

$$\widehat{RTT}_k = \psi \times \widehat{RTT}_{k-1} + (1-\psi) \times RTT_k \tag{5}$$

where $0 \leq \psi \leq 1$. TcpHas algorithm during the congestion avoidance phase is described in Algorithm 7, where $EncodingRate[\widehat{QLevel}] \times 1.2$ is the shaping rate.

Algorithm 7 TcpHas algorithm in congestion avoidance phase.

```
1: if newACK is received and cwnd ≥ ssthresh then
2:     cwnd = EncodingRate[QLevel^] × RTT^ × 1.2
3: end if
```

To sum up, TcpHas is a congestion control optimized for video streaming of type HAS. It is implemented in the server, at transport layer, no other modifications are needed. TcpHas needs only one information from the application layer: the encoding bitrates of the selected video level. It coexists gracefully with TCP on server, the transport layer simply checking whether the *EncodingRate* vector returned by application is empty or not, as explained in section 3.2. It is compatible with all TCP clients.

There is no direct interaction between the client and the server to make the adaptation decision. When TcpHas performs bandwidth estimation (at the server), it is independent of the estimation made in the HAS player on the client side. The only objective of the bandwidth estimation of the server is to shape the sending bitrate in order to prevent the HAS player to select a quality level higher than the optimal one. Hence, the bandwidth estimation in the server provides a proactive optimization: it limits the sending bitrate before the client may select an inappropriate quality level.

4 TcpHas Evaluation

The final goal of our work is to implement our idea in real software. However, at this stage of the work, we preferred instead to use a simulated player because of the classical advantages of simulation over experimentation, such as reproducibility of results, and measurement of individual parameters for better parameter tuning. More precisely, we preferred to use a simulated player instead of a commercial player for the following main reasons:

- First, the commercial players have complex implementation with many parameters. Besides, the bitrate controller is different among commercial players and even between different versions of the same player. Accordingly, using our own well-controlled player allows better evaluations than a "black box" commercial player that could give incomprehensible behaviors.
- Second, some image-oriented perceptual factors used in a real player (e.g. video spatial resolution, frame rate or type of video codec) are of no interest for HAS evaluation.
- Third, objective metrics are increasingly employed for HAS evaluation. In reliable flows, such as those using TCP, objective metrics lead to the same results no matter the file content. Hence, with a fully controlled simulated player we can easily get the variation of its parameters during time and use them for objective QoE metric computation.
- Fourth, for our simulations, we need to automatize events such as triggering the beginning and the end of HAS flows at precise moments. Using a simulated player offers easier manipulation than a real player, especially when many players need to be launched simultaneously.

In this section, we evaluate TcpHas using the classical ns-3 simulator, version 3.17. In our scenario, several identical HAS players share the same bottleneck link and compete for bandwidth inside a home network. We first describe the network setup used in all of the simulations. Then, we describe the parameter settings of TcpHas. Afterwards, we show the behavior of TcpHas compared to the other methods. Finally, after describing the QoE and QoS metrics used, we analyze results for 1 to 9 competing HAS flows in the home network and with background traffic.

4.1 Simulation setup

Fig. 2 presents the architecture we used, which is compliant with the fixed broadband access network architecture used by Cisco to present its products [15]. The HAS clients are located inside the home network, a local network with 100 Mbps bandwidth. The Home Gateway (HG) is connected to the DSLAM. The bottleneck link is located between HG and DSLAM and has 8 Mbps. The queue of the DSLAM uses Drop Tail discipline with a length that corresponds to the bandwidth-delay product. Nodes BNG (Broadband Network Gateway)

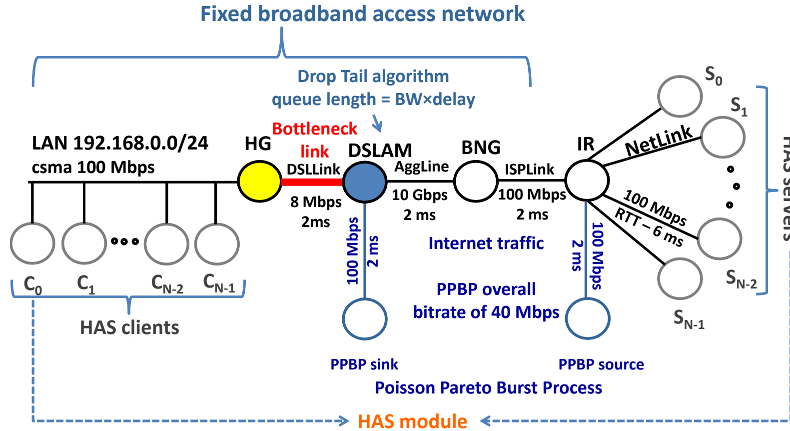


Fig. 2 Network architecture used in ns-3 for the evaluation.

and IR (Internet Router), and links AggLine (that simulates the aggregate line), ISPLink (that simulates the Internet Service Provider core network) and NetLink (that simulates the route between the IR and the HAS server) are configured so that their queues are large enough (1000 packets) to support a large bandwidth of 100 Mbps and high delay of 100 ms without causing significant packet losses.

We generate Internet traffic that crosses ISPLink and AggLine, because the two simulated links are supposed to support a heavy traffic from ISP networks. For Internet traffic, we use the *Poisson Pareto Burst Process* (PPBP) model [40], considered as a simple and accurate traffic model that matches statistical properties of real-life IP networks (such as their bursty behavior). PPBP is a process based on the overlapping of multiple bursts with heavy-tailed distributed lengths. Events in this process represent points of time at which one of an infinite population of users begins or stops transmitting a traffic burst. PPBP is closely related to the $M/G/\infty$ queue model [40]. We use the PPBP implementation in ns-3 [5, 6]. In our configuration, the overall rate of PPBP traffic is 40 Mbps, which corresponds to 40% of ISPLink capacity.

The ns-3 simulated TcpHas player we use is similar to the emulated player described in [9], with a chunk duration of 2 seconds and a playback buffer of 30 seconds (maximum video size the buffer can hold). Note that [9] compares four TCP variants and two router-based traffic shaping methods, whereas the current article proposes a new congestion control to be executed on server. Our player is classified as *Rate and Buffer based* (RBB) player, following classification proposed in [38, 37]. Using buffer occupancy information is increasingly proposed and used due to its advantages for reducing stalling events. In addition, the bandwidth estimator we used consists in dividing the size of received chunk by its download duration. The buffer occupancy information is used only to define an aggressiveness level of the player, which allows the player to ask a quality level higher than the estimated bandwidth. The player uses HTTP GET requests to ask for each chunk. It has two phases: buffering and steady state. During buffering phase it fills up its playback buffer by asking for chunks of the lowest video quality level, each chunk immediately after the other. When the playback buffer fills up, the player switches to the steady state phase. In this phase, it asks for the next chunk of the estimated quality level each time the playback buffer occupancy drops for more than 2 seconds (i.e. it remains less than 28 seconds of video in the buffer). When the playback buffer is empty, the player re-enters the buffering phase.

Table 1 Available encoding bitrates for the video file used in simulations.

Video quality level L_{C-S}	0	1	2	3	4
Encoding bitrate (kbps)	248	456	928	1632	4256

All tests use five video quality levels with constant encoding rates presented in Table 1, and correspond to the quality levels usually used by many video service providers. Since objective metrics are used, cf. section 4, and given that TCP use ensures that all packets arrive to the destination, the exact video type used does not influence results (in our case we used random bits for the video file).

We also use the HTTP traffic generator module given in [13, 14]. This module allows communication between two nodes using HTTP protocol, and includes all features that generate and control HTTP GET Request and HTTP response messages. We wrote additional code into this HTTP module by integrating the simulated HAS player. We call this implementation the HAS module, as presented in Fig. 2. Streaming is done from S_f to C_f, where $0 \leq f < N$. The round-trip propagation delay between S_f and C_f is 100 ms.

We show results for TcpHas, Westwood+, TIBET and Trickle. We do not consider Westwood because Westwood+ is supposed to replace Westwood since it performs better in case of ACK compression and clustering. Concerning Trickle, it is a traffic shaping method that was proposed in the context of progressive download, as described in SubSection 2.2. In order to adapt it to HAS, we added to it the estimator of optimal quality level of TcpHas, the adaptive decrease mechanism of Westwood+ (the same as TIBET), and applied the Trickle traffic shaping based on the estimated optimal quality level. This HAS adaptation of Trickle is simply denoted by "Trickle" in the reminder of this article.

For all evaluations, we use competing players that are playing simultaneously during K seconds. We set $K = 180$ seconds, which allows the HAS players to reach stationary behavior when they are competing for bandwidth [21].

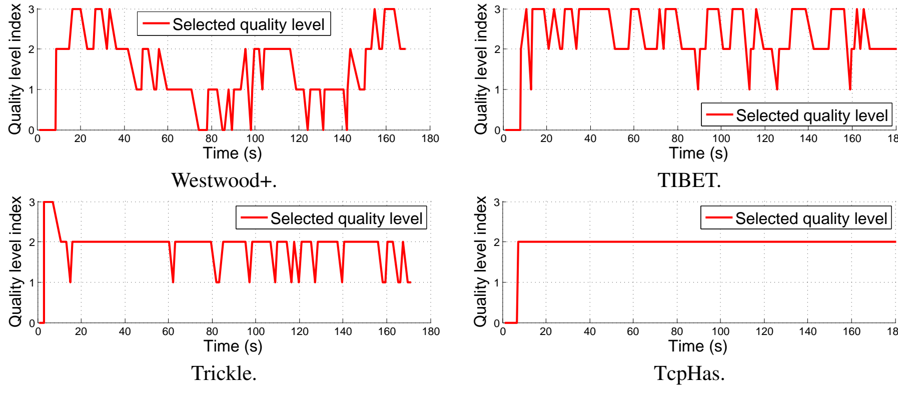
4.2 TcpHas Parameter Settings

The parameter γ of the $\widehat{Bwe}$ low-pass filter is constant, in conformity with Subsection 3.1. We set $\gamma = 0.99$ to reduce the oscillation of bandwidth estimations, $\widehat{Bwe}$, over time. We set $initial_cwnd = 2 \times MSS$.

The initial bandwidth estimation value of $\widehat{Bwe}$ is set to the highest encoding bitrate. If the first value was set to zero or to the first estimation sample, the low-pass filtering process with parameter $\gamma = 0.99$ would be too slow to reach the correct estimation. In addition, we want TcpHas to quickly reach the highest quality level at the beginning of the stream, as explained in Subsection 3.3.

The parameter $alpha$ of the TIBET estimation scheme (see Algorithm 2) is chosen empirically in our simulations and is set to 0.8. A higher value produces more stable estimations but is less responsive to network changes, whereas a lower value makes TcpHas more aggressive with a tendency to select the quality level that corresponds to the whole bandwidth (unfairness).

The parameter ψ used for low-pass filtering the RTT measurements in Subsection 3.4 is set to 0.99. The justification for this value is that it reduces better the RTT fluctuations, and consequently reduces $cwnd$ fluctuation during the congestion avoidance phase.



Westwood+. TIBET. Trickle. TcpHas.

Fig. 3 Quality level selection over time for the four methods compared.

4.3 TcpHas Behavior Compared to the Other Methods

We present results for a scenario with 8 competing identical clients.

Figure 3 shows the quality level selection over time of one of the competing players for the above methods. The optimal quality level that should be selected by the competing players is n° 2. During the buffering phase, all players select the lowest quality level, as allows by slow start phase. However, during the steady-phase the results diverge: Westwood+ player frequently changes the quality level between n° 0 and n° 3, which means that not only the player produces an unstable HAS stream, but also a high risk of generating stalling events. TIBET player is more stable and presents less risk of stalling events. Trickle player has an improved performance and becomes more stable around the optimal quality level n° 2, with some oscillations between quality levels n° 1 and n° 3. In contrast, TcpHas player is stable at the optimal quality level during the steady-state phase, hence it performs the best among all the methods.

Given that the congestion control algorithms of these four methods use bandwidth estimation $\widehat{Bwe}$ to set *ssthresh*, it is interesting to present $\widehat{Bwe}$ variation over time, shown in Figure 4. The optimal $\widehat{Bwe}$ estimation should be equal to the bottleneck capacity (8 Mbps) divided by the number of competing HAS clients (8), i.e., 1 Mbps. For Westwood+, $\widehat{Bwe}$ varies between 500 kbps and 2 Mbps. For TIBET, $\widehat{Bwe}$ is more stable but varies between 1.5 Mbps and 2 Mbps, which is greater than the average of 1 Mbps; this means that an unfairness in bandwidth sharing occurred because this player is more aggressive than the other 7 competing players. For TcpHas, $\widehat{Bwe}$ begins by the initial estimation that corresponds to the encoding bitrate of the highest quality level (4256 kbps), as described in Algorithm 6, then $\widehat{Bwe}$ converges rapidly to the optimal estimation value of 1 Mbps. Both Trickle and TcpHas present a similar $\widehat{Bwe}$ shape because they use the same bandwidth estimator.

The dissimilarity between the four algorithms is more visible in Figure 5, which presents the variation of *cwnd* and *ssthresh*. Westwood+ and TIBET yield unstable *ssthresh* and even more unstable *cwnd*. In contrast, Trickle and TcpHas provide stable *ssthresh* values. For Trickle, *cwnd* is able to increase to high values during the congestion avoidance phase because Trickle limits the congestion window by setting an upper bound in order to have a sending bitrate close to the encoding bitrate of the optimal quality level. For TcpHas, *ssthresh* is stable at around 14 kB, which corresponds to the result of Equation 4 when $\widehat{QLevel} = 2$ and $RTT_{min} = 100\,\text{ms}$. Besides, *cwnd* is almost in the same range as *ssthresh*

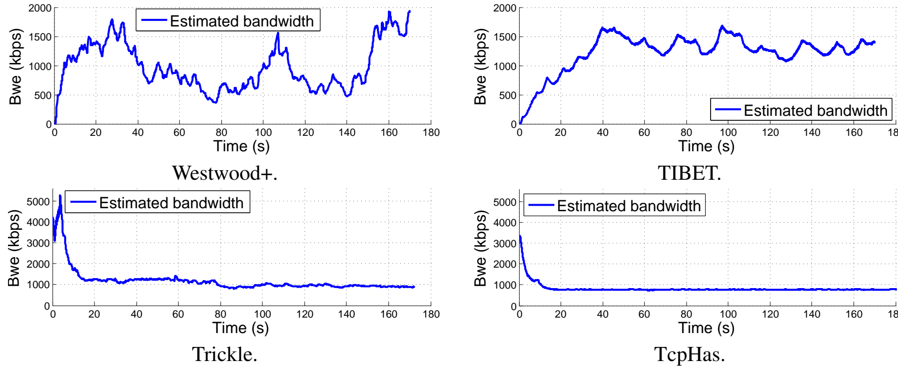


Fig. 4 Estimated bandwidth $\widehat{Bwe}$ over time for the four methods compared.

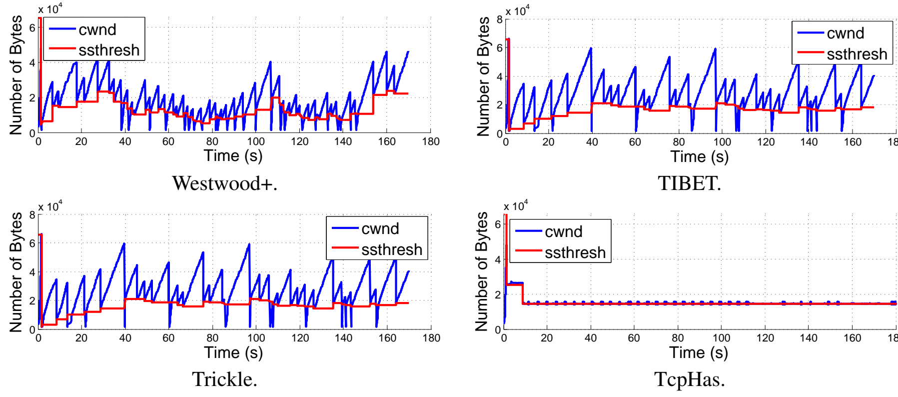


Fig. 5 *cwnd* and *ssthresh* values over time for the four methods compared.

and increases during ON periods because it takes into acocount the increase of *RTT* as presented in Algorithm 7.

4.4 QoE and QoS Metrics

This subsection describes the specific QoE and QoS metrics we selected to evaluate objectively TcpHas, and justify their importance in our evaluation.

4.4.1 QoE Metrics

We use three QoE metrics described by formulas in [8, 9]: the *instability of video quality level* [8, 9, 22] (0% means the same quality, 100% means the quality changes *each* period), the *infidelity to the optimal quality level* [8, 9] (percentage in seconds where the optimal quality is used), and the *convergence speed to the optimal quality level* [8, 9, 21] (time to stabilize on the optimal quality and be stable over at least 1 minute). The optimal quality level $L_{C-S,opt}$ used in our evaluation is given by the highest encoding bitrate among the quality levels which is lower than or equal to the ratio between the bottleneck bandwidth,

avail_bw, and the number of competing HAS players, *N*, as follows:

$$L_{C-S,opt} = \left\{ \max_{0 \leq L_{C-S} \leq 4} (L_{C-S}) \middle| EncodingRate(L_{C-S}) \times 1.2 \leq \frac{avail_bw}{N} \right\} \quad (6)$$

This formula applies to all the flows, i.e., we attach the same optimal quality level to all flows; this is because our focus is on fairness among flows. We acknowledge that this does not use the maximum achievable bandwidth in some cases, for example for six clients sharing an 8 Mbps bottleneck link, the above formula gives 928 kbps for each client, and not 928 kbps for five clients and 1632 kbps for the sixth client (see Table 1 for the bitrates). We however noticed that TcpHas does maximize the bandwidth use in some cases, as presented in the next section.

The fourth metric is the *initial delay*, metric adopted by many authors [23, 34], which accounts for the fact that the user dislikes to wait a long time before the beginning of video display.

The fifth metric is the *stalling event rate*; the user is highly disturbed when the video display is interrupted while concentrating on watching [20]. We define the stalling event rate, $StallingRate(K)$, as the number of stalling events during a *K*-second test duration, divided by *K* and multiplied by 100:

$$StallingRate(K) = \frac{number\,of\,stalling\,events\,during\,K\,seconds}{K} \times 100 \quad (7)$$

The greater the $StallingRate(K)$, the greater the dissatisfaction of the user. A streaming technology must try as much as possible to have a zero stalling event rate.

4.4.2 QoS Metrics

We use four QoS metrics, described in the following.

The first metric is the *frequency of* $\widehat{OFF}$ *periods* [8, 9]. $\widehat{OFF}$ is an OFF period whose duration exceeds TCP retransmission timeout duration (*RTO*); such periods lead to a reduction of bitrate and potentially to a degradation of performance [8, 9]. This metric is defined as the total number of $\widehat{OFF}$ periods divided by the total number of downloaded chunks of one HAS flow:

$$fr_{\widehat{OFF}} = \frac{number\,of\,\widehat{OFF}\,periods}{number\,of\,chunks} \quad (8)$$

A high queuing delay is harmful to HAS and for real-time applications [36]. We noticed in our tests that this delay could vary considerably, and so the RTT of the HAS flow, so we use as the second metric the *average queuing delay*, defined as:

$$Delay_{C-S}(K) = RTT_{C-S,mean}(K) - RTT^0_{C-S} \quad (9)$$

where $RTT_{C-S,mean}(K)$ is the average among all RTT_{C-S} samples of the whole HAS session between client *C* and server *S* for a *K*-second test duration, and RTT^0_{C-S} is the initial round-trip propagation delay between the client *C* and the server *S*.

The congestion detection events greatly influence both QoS and QoE of HAS because the server decreases its sending rate at each such event. This event is always accompanied by a *ssthresh* reduction. Hence, we use a third metric, *congestion rate*, which we define as the rate of congestion events detected on the server side:

$$CNG_{C-S}(K) = \frac{D^{ssthresh}_{C-S}(K)}{K} \times 100 \quad (10)$$

where $D_{C-S}^{ssthresh}(K)$ is the number of times the *ssthresh* has been decreased for the *C–S* HAS session during the *K*-second test duration.

The fourth metric we use is the *average packet drop rate*. The rationale is that the number of dropped packets at the bottleneck gives an idea of the congestion severity of the bottleneck link. We define this metric as the average packet drop rate at the bottleneck during a *K*-second test duration:

$$DropPkt(K) = \frac{number\,of\,dropped\,packets\,during\,K\,seconds}{K} \times 100 \tag{11}$$

Note that this metric is different from the congestion rate described above, because the TCP protocol at the server could detect a congestion event whereas there is no packet loss.

4.5 Performance Evaluation

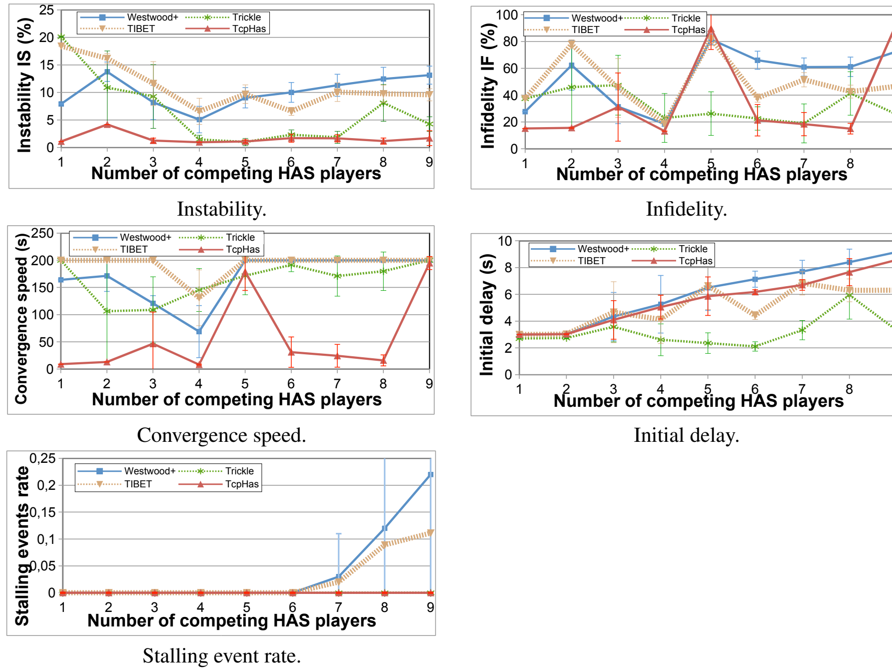
In this subsection we evaluate objectively the performance of TcpHas compared to Westwood+, TIBET and Trickle. For this, we give and comment results of evaluation in two scenarios: when increasing the number of competing HAS flows in the home network and when increasing the background traffic in access network.

We use 16 runs for each simulation. We present the mean value for QoE and QoS among the competing players and among the number of runs for each simulation. We present the performance unfairness measurements among HAS clients with vertical error bars. We chose 16 runs because the relative difference between mean value of instability and infidelity of 16 and 64 runs is less than 4%.

4.5.1 Effect of Increasing the Number of HAS Flows

Here, we vary the number of competing players from 1 to 9. We select a maximum of 9 competing HAS clients because in practice the number of users inside a home network does not exceed 9.

QoE results are given in Figure 6. In this Figure, the lowest instability rate is that of TcpHas (less than 4%), with a negligible instability unfairness between players. Trickle shows a similar instability rate when the number of competing players is between 4 and 7, but for the other cases it has a high instability rate, whose cause is that Trickle does not take into consideration the reduction of *cwnd* during $\widehat{OFF}$ periods which causes a low sending rate after each $\widehat{OFF}$ period. Hence, Trickle is sensitive to $\widehat{OFF}$: we can see in the Figure a correlation between instability and frequency of $\widehat{OFF}$ period. In contrast, the instability of Westwood+ and TIBET is much greater and increases with the number of competing players. The infidelity and convergence speed of TcpHas are satisfactory, as presented in the Figure: the infidelity rate is less than 30% and convergence speed is smaller than 50 seconds in all but two cases. When there are 5 or 9 competing HAS clients, TcpHas selects a quality level higher than the optimal quality level that we defined (equation 6); TcpHas is thus able to select a higher quality level, converge to it, and be stable on it for the whole duration of the simulation. This result is rather positive, because TcpHas is able to maximize the occupancy of the home bandwidth to almost 100% in these two particular cases. In contrast, Westwood+ and TIBET present high infidelity to the optimal quality level and have difficulties to converge to it. For Trickle, due to its traffic shaping algorithm, the infidelity rate is lower than 45%, and lower than 25% when the $\widehat{OFF}$ frequency (hence the instability rate) is low.



Instability.

Infidelity.

Convergence speed.

Initial delay.

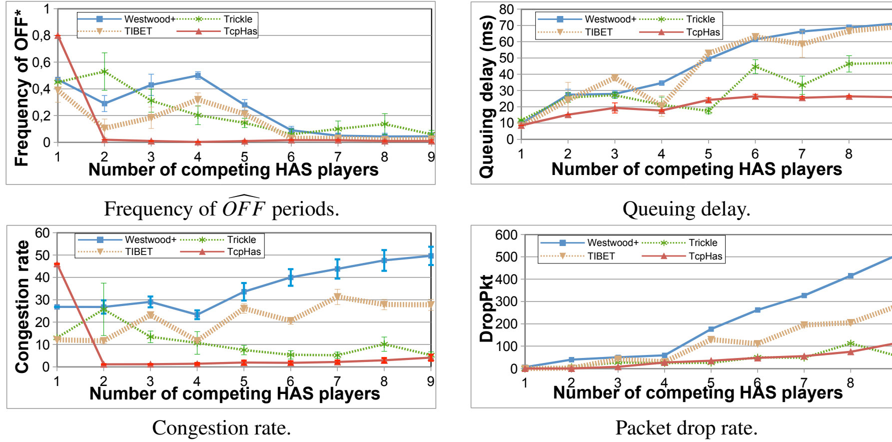
Stalling event rate.

Fig. 6 Values of QoE metrics when increasing the number of competing HAS clients for the four methods compared.

The initial delay of the four methods increases with the number of competing HAS clients, as presented in Figure 6. The reason is that during the buffering phase the HAS player asks for chunks successively, and when the number of competing HAS clients increases, the bandwidth share for each flow decreases, thus generating additional delay. We also notice that Westwood+, TIBET and TcpHas present initial delay in the same range of values. However, Trickle has a lower delay; the reason is that, as shown in figures 4 and 5, during buffering state Trickle is able to maintain the initial bandwidth estimation that corresponds to the encoding bitrate of the highest quality level and does not provoke congestions. In other words, Trickle is able to send video chunks with a high sending bitrate without causing congestions. This leads to the reduction of the initial delay.

Finally, Figure 6 shows that TcpHas and Trickle generate no stalling events, whereas Westwood+ and TIBET do starting from 7 competing HAS clients. The result for TcpHas and Trickle comes from their high stability rate even for a high number of competing HAS clients.

QoS results are given in Figure 7. It shows that the $\widehat{OFF}$ period frequency of TcpHas is kept near to zero and much lower than Westwood+, TIBET and Trickle, except in the case when the home network has only one HAS client. In this case, the optimal quality level is $n°4$ whose encoding rate is 4.256 Mbps. Hence, the chunk size is equal to 8.512 Mbits. Consequently, when TcpHas shapes the sending rate according to this encoding rate while delivering chunks with large sizes, it would be difficult to reduce OFF periods below the retransmission timeout duration, RTO. Note that we have taken this case into account when proposing TcpHas by eliminating the initialization of $cwnd$ after idle periods, as explained in Algorithm 5, to preserve high QoE.



Frequency of $\widehat{OFF}$ periods.

Queuing delay.

Congestion rate.

Packet drop rate.

Fig. 7 Values of QoE metrics when increasing the number of competing HAS clients for the four methods compared.

As presented in Figure 7, although the queuing delay of the four methods increases with the number of competing HAS clients, TcpHas and Trickle present a lower queuing delay than Westwood+ and TIBET. The reason is that both TcpHas and Trickle shape the HAS flows by reducing the sending rate of the server which reduces queue overflow in the bottleneck. Additionally, we observe that TcpHas reduces better the queuing delay than Trickle; TcpHas has roughly half the queuing delay of Westwood+ and TIBET. Besides, TcpHas does not increase its queuing delay more than 25 ms even for 9 competing players, while Trickle increases it to about 50 ms. This result is mainly due to the high stability of the HAS quality level generated by TcpHas which offers better fluidity of HAS flows inside the bottleneck. The same reason applies for the very low congestion detection rate and packet drop rate at the bottleneck of TcpHas, given in Figure 7. Furthermore, the congestion rate of Trickle is correlated to its frequency of $\widehat{OFF}$ periods; this means that *ssthresh* reduction of Trickle is principally caused by the detection of $\widehat{OFF}$ periods. In addition, due to its corresponding traffic shaping method that reduces the sending rate of HAS server, the packet drop rate of Trickle is quite similar to that of TcpHas, as shown in Figure 7.

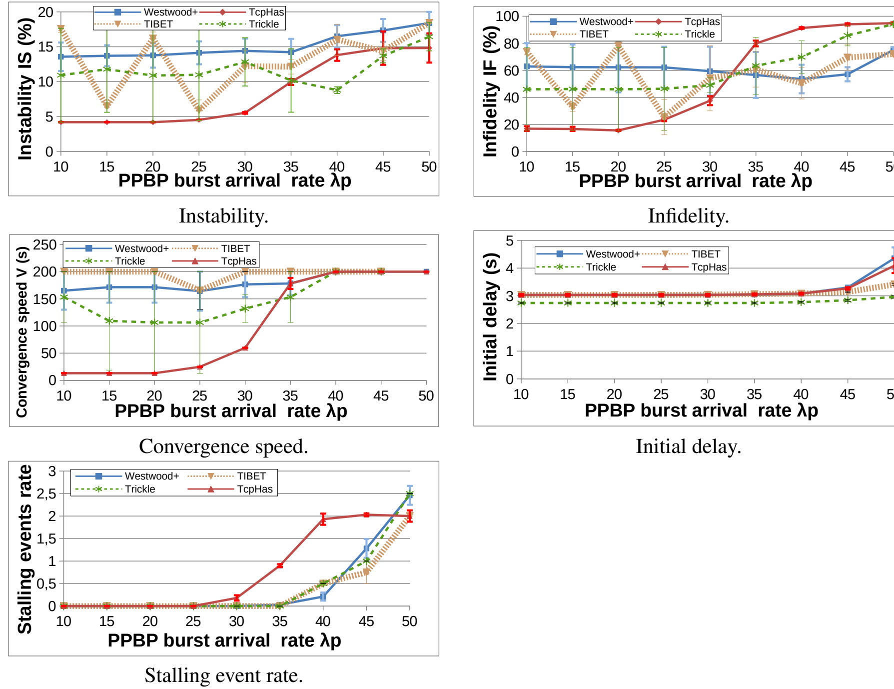
To summarize, TcpHas is not affected by the increase of the competing HAS clients in the same home network. From QoE point of view, it preserves high stability and high fidelity to optimal quality level, and has a tendency to increase the occupancy of the home bandwidth. From QoS point of view, it maintains a low $\widehat{OFF}$ period duration, low queuing delay, and low packet drop rate.

4.5.2 Background Traffic Effect

Here we vary the burst arrival rate λ_p of the Poisson Pareto Burst Process (PPBP) that simulates the traffic that crosses Internet Router (IR) and DSLAM from 10 to 50. Table 2 shows the percentage of occupancy of the 100 Mbps ISP network (ISPLink in our network) for each selected λ_p value (without counting HAS traffic). Hence, we simulate a background traffic in the ISP network ranging from 20% to 100% of network capacity. In our simulations, we used two competing HAS clients inside the same home network.

Table 2 ISP network load when varying the burst arrival rate λ_p.

λ_p	10	15	20	25	30	35	40	45	50
ISP network load (%)	20	30	40	50	60	70	80	90	100



Instability.

Infidelity.

Convergence speed.

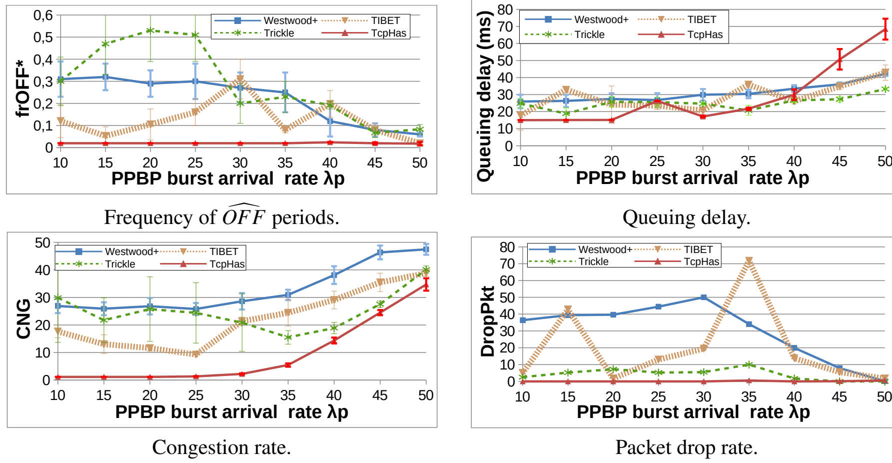
Initial delay.

Stalling event rate.

Fig. 8 Values of QoE metrics when increasing the burst arrival rate for the four methods compared.

QoE results are given in Figure 8. It shows that the instability, infidelity, convergence speed and stalling event rate curves of TcpHas present two different regimes. When $\lambda_p < 35$, TcpHas keeps practically the same satisfying measurements much better than Westwood+, TIBET and Trickle. However, when $\lambda_p > 35$, the four measurements degrade suddenly and stabilize around same high values; even for infidelity and stalling rate, TcpHas yields worse values than Weswood+, TIBET and Trickle. We deduce that TcpHas is sensitive to additional load of ISP network, and could be more harmful than the three other methods. Trickle presents relatively better performance than Westwood+ and TIBET in terms of average values. However, it presents higher unfainess between clients, as shown by its big vertical error bars. In addition, we observe that TcpHas presents the same initial delay as the other methods, which is around 3 seconds, and does not exceed 5 seconds and does not disturb the user's QoE.

QoS results are presented in Figure 9. Westwood+, TIBET and Trickle present high frequency of $\widehat{OFF}$ periods, which decreases when increasing the ISP network load, whereas TcpHas presents low $\widehat{OFF}$ frequency. The average queuing delay generated by TcpHas is lower than that of Westwood+, TIBET and Trickle for $\lambda_p < 40$. The reason for this is explained in the Figure: the congestion detection rate increases with λ_p (especially above 40), while the packet drop rate at the bottleneck is still null for TcpHas. Hence, we deduce



Frequency of $\widehat{OFF}$ periods.

Queuing delay.

Congestion rate.

Packet drop rate.

Fig. 9 Values of QoS metrics when increasing the burst arrival rate for the four methods compared.

that the bottleneck is no more located in the link between DSLAM and IR, but is rather transposed inside the loaded ISP link.

To summarize, TcpHas yields very good QoE and QoS results when the ISP link is not too loaded. Beyond 70% load of the ISP link, the congestion rate increases, which degrades the QoS and forces TcpHas to frequently update its estimated optimal quality level, which in turn degrades the QoE.

5 Conclusion

This paper presents and analyses server-based shaping methods that aim to stabilize the video quality level and improve the QoE of HAS users.

Based on this analysis, we propose and describe TcpHas, a HAS-based TCP congestion control that acts like a server-based HAS traffic shaping method. It is inspired by the TCP adaptive decrease mechanism and uses the end-to-end bandwidth estimation of TIBET to estimate the optimal quality level. Then, it shapes the sending rate to match the encoding bitrate of the estimated optimal quality level. The traffic shaping process is based on updating *ssthresh* when detecting a congestion event or after an idle period, and on modifying *cwnd* during the congestion avoidance phase.

We evaluate TcpHas in the case of HAS clients that share the bottleneck link and are competing for the same home network under various conditions. Simulation results indicate that TcpHas considerably improves both HAS QoE and network QoS. Concerning QoE, it offers a high stability, high fidelity to optimal quality level, a rapid convergence speed, and an acceptable initial delay. Concerning QoS, it reduces the frequency of large OFF periods that exceed TCP retransmission timeout, reduces queuing delay, and reduces considerably the packet drop rate in the shared bottleneck queue. TcpHas performs well when increasing the number of competing HAS clients and does not cause stalling events. It shows excellent performance for small and medium loaded ISP network.

As future work, we plan to implement TcpHas in real DASH servers of a video content provider, and to offer a large-scale evaluation during a long duration of tests in real and

variable network conditions when hundreds of DASH players located in different access networks are asking for video content.

References

1. Abdallah A, Meddour DE, Ahmed T, Boutaba R (2010) Cross layer optimization architecture for video streaming in WiMAX networks. In: 2010 IEEE Symposium on Computers and Communications (ISCC), IEEE, pp 8–13
2. Akhshabi S, Anantakrishnan L, Begen AC, Dovrolis C (2012) What happens when HTTP adaptive streaming players compete for bandwidth? In: Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video, ACM, pp 9–14
3. Akhshabi S, Anantakrishnan L, Dovrolis C, Begen AC (2013) Server-based traffic shaping for stabilizing oscillating adaptive streaming players. In: 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video, ACM, pp 19–24
4. Allman M, Paxson V, Blanton E (2009) TCP congestion control. RFC 5681
5. Ammar D (2016) PPBP in ns-3. https://codereview.appspot.com/4997043
6. Ammar D, Begin T, Guerin-Lassous I (2011) A new tool for generating realistic internet traffic in ns-3. In: 4th International ICST Conference on Simulation Tools and Techniques, pp 81–83
7. Ben Ameur C, Mory E, Cousin B (2014) Shaping HTTP adaptive streams using receive window tuning method in home gateway. In: IEEE International Conference on Performance Computing and Communications (IPCCC), pp 1–2
8. Ben Ameur C, Mory E, Cousin B (2015) Evaluation of gateway-based shaping methods for HTTP adaptive streaming. In: Quality of Experience-based Management for Future Internet Applications and Services (QoE-FI) Workshop, IEEE International Conference on Communications (ICC), London, UK, pp 1–6
9. Ben Ameur C, Mory E, Cousin B (2016) Combining traffic shaping methods with congestion control variants for HTTP adaptive streaming. Multimedia Systems pp 1–18
10. Ben Ameur C, Mory E, Cousin B, Dedu E (2017) TcpHas: TCP for HTTP adaptive streaming. In: IEEE International Conference on Communications (ICC), IEEE, Paris, France, pp 1–7
11. Capone A, Martignon F, Palazzo S (2001) Bandwidth estimates in the TCP congestion control scheme. In: Thyrrhenian International Workshop on Digital Communications: Evolutionary Trends of the Internet (IWDC), Springer, pp 614–626
12. Capone A, Fratta L, Martignon F (2004) Bandwidth estimation schemes for TCP over wireless networks. IEEE Transactions on Mobile Computing 3(2):129–143
13. Cheng Y (2016) HTTP traffic generator. https://codereview.appspot.com/4940041
14. Cheng Y, Çetinkaya EK, Sterbenz JP (2013) Transactional traffic generator implementation in ns-3. In: 6th International ICST Conference on Simulation Tools and Techniques, pp 182–189
15. Cisco (2013) Broadband network gateway overview. http://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k_r4-3/bng/configuration/guide/b_bng_cg43xasr9k/b_bng_cg43asr9k_chapter_01.html
16. Dedu E, Ramadan W, Bourgeois J (2015) A taxonomy of the parameters used by decision methods for adaptive video transmission. Multimedia Tools and Applications 74(9):2963–2989

17. Floyd S, Handley M, Padhye J, Widmer J (2008) TCP Friendly Rate Control (TFRC): Protocol specification. RFC 5348
18. Ghobadi M, Cheng Y, Jain A, Mathis M (2012) Trickle: Rate limiting youtube video streaming. In: Usenix Annual Technical Conference, Boston, MA, USA, pp 191–196
19. Hoquea MA, Siekkinena M, Nurminena JK, Aaltob M, Tarkoma S (2015) Mobile multimedia streaming techniques: QoE and energy saving perspective. Pervasive and Mobile Computing 16, Part A:96–114
20. Hoßfeld T, Egger S, Schatz R, Fiedler M, Masuch K, Lorentzen C (2012) Initial delay vs. interruptions: Between the devil and the deep blue sea. In: 4th International Workshop on Quality of Multimedia Experience (QoMEX), IEEE, Melbourne, Australia, pp 1–6
21. Houdaille R, Gouache S (2012) Shaping HTTP adaptive streams for a better user experience. In: 3rd Multimedia Systems Conference, ACM, Chapel Hill, NC, USA, pp 1–9
22. Jiang J, Sekar V, Zhang H (2012) Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with festive. In: 8th international conference on Emerging networking experiments and technologies, ACM, pp 97–108
23. Krogfoss B, Agrawal A, Sofman L (2012) Analytical method for objective scoring of HTTP adaptive streaming (HAS). In: IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), IEEE, pp 1–6
24. Li SQ, Chong S, Hwang CL (1995) Link capacity allocation and network control by filtered input rate in high-speed networks. IEEE/ACM Transactions on Networking (TON) 3(1):10–25
25. Mansy A, Ver Steeg B, Ammar M (2013) Sabre: A client based technique for mitigating the buffer bloat effect of adaptive video flows. In: 4th ACM Multimedia Systems Conference, ACM, pp 214–225
26. Mascolo S, Grieco LA (2003) Additive increase early adaptive decrease mechanism for TCP congestion control. In: 10th International Conference on Telecommunications (ICT), IEEE, vol 1, pp 818–825
27. Mascolo S, Racanelli G (2005) Testing TCP Westwood+ over transatlantic links at 10 gigabit/second rate. In: Protocols for Fast Long-distance Networks (PFLDnet) Workshop, Lyon, France, pp 1–6
28. Mascolo S, Casetti C, Gerla M, Sanadidi MY, Wang R (2001) TCP Westwood: Bandwidth estimation for enhanced transport over wireless links. In: 7th annual international conference on Mobile computing and networking, ACM, pp 287–297
29. Mascolo S, Grieco LA, Ferorelli R, Camarda P, Piscitelli G (2004) Performance evaluation of Westwood+ TCP congestion control. Performance Evaluation 55(1):93–111
30. Mogul JC (1992) Observing TCP dynamics in real networks. ACM SIGCOMM Computer Communication Review 22(4)
31. Ramadan W, Dedu E, Bourgeois J (2011) Avoiding quality oscillations during adaptive streaming of video. International Journal of Digital Information and Wireless Communications (IJDIWC) 1(1):126–145
32. Sandvine (2016) Global internet phenomena report. `https://www.sandvine.com`
33. Seufert M, Egger S, Slanina M, Zinner T, Hobfeld T, Tran-Gia P (2014) A survey on quality of experience of HTTP adaptive streaming. Communications Surveys & Tutorials, IEEE 17(1):469–492
34. Shuai Y, Petrovic G, Herfet T (2015) Olac: An open-loop controller for low-latency adaptive video streaming. In: IEEE International Conference on Communications (ICC), IEEE, London, UK, pp 6874–6879

35. Villa BJ, Heegaard PE (2013) Group based traffic shaping for adaptive HTTP video streaming by segment duration control. In: 27th IEEE International Conference on Advanced Information Networking and Applications (AINA), IEEE, pp 830–837
36. Yang H, Chen X, Yang Z, Zhu X, Chen Y (2014) Opportunities and challenges of HTTP adaptive streaming. International Journal of Future Generation Communication and Networking 7(6):165–180
37. Yin X, Sekar V, Sinopoli B (2014) Toward a principled framework to design dynamic adaptive streaming algorithms over HTTP. In: 13th ACM Workshop on Hot Topics in Networks, ACM, Los Angeles, CA, USA, pp 1–9
38. Yin X, Jindal A, Sekar V, Sinopoli B (2015) A control-theoretic approach for dynamic adaptive video streaming over HTTP. ACM SIGCOMM Computer Communication Review 45(4):325–338
39. Zhang L, Shenker S, Clark DD (1991) Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. ACM SIGCOMM Computer Communication Review 21(4):133–147
40. Zukerman M, Neame TD, Addie RG (2003) Internet traffic modeling and future technology implications. In: 22nd Annual Joint Conference of the IEEE Computer and Communications (INFOCOM), IEEE, vol 1, pp 587–596