Client-side Bandwidth Estimation Technique for Adaptive Streaming of a Browser Based Free-Viewpoint Application

Tilak Varisetty
Institute of Communications Technology
Leibniz Universität Hannover, Germany
tilak.varisetty@ikt.uni-hannover.de

David Dietrich
Institute of Communications Technology
Leibniz Universität Hannover, Germany
david.dietrich@ikt.uni-hannover.de

ABSTRACT

High bandwidth demands in interactive streaming applications pose challenges in efficiently utilizing the available bandwidth. Well known standards like MPEG-DASH and Apple HTTP streaming use buffer control mechanisms at the client for bandwidth estimation and segmentation at the server, which can add latency and complexity. We present a prototype of a real-time interactive free-viewpoint rendered application, that can be applied as an alternative to the existing buffering and segmentation-based approaches. We employ a novel approach in passive bandwidth estimation and implemented a weighed bit rate algorithm in JavaScript and HTML5 that utilizes browser statistics. The video quality fluctuations at the client are signaled to the server via WebSocket. Using open source technologies and client-side bandwidth estimation techniques, the feasibility of the proposed algorithm is demonstrated. The evaluation results show that the prototype quickly responds to fluctuations in the available bandwidth under varying network conditions.

KEYWORDS

Free-viewpoint streaming; HTML5; bandwidth estimation

ACM Reference Format:
Tilak Varisetty and David Dietrich. 2019. Client-side Bandwidth Estimation Technique for Adaptive Streaming of a Browser Based Free-Viewpoint Application. In *2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges (HotEdgeVideo'19), October 21, 2019, Los Cabos, Mexico.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3349614.3356021

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotEdgeVideo'19, October 21, 2019, Los Cabos, Mexico
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6928-2/19/10…$15.00
https://doi.org/10.1145/3349614.3356021

1 INTRODUCTION

Interactive streaming has come into importance with the technologies like 3D rendering, multi-view, and free-viewpoint video (FVV) streaming, where the user can switch the views based on the multiple views available at the server or client-side [17]. In the case of multi-view streaming, view synthesis can be achieved by server or client-side rendering, or with multiple camera arrays, which demand high network bandwidth and CPU capacities. In our current work, we demonstrate a FVV streaming application where the requested view is rendered by the server with synthesis and reference depth images instead of multiple cameras. The rendered 3D model is encoded and transferred to the client through HTTP streaming. Fluctuations in the available bandwidth (ABW) due to the varying network conditions have an effect on the video delivery, and adaptive video streaming comes into play to cope with the varying network conditions. The key phases of adaptive streaming are bandwidth estimation, signaling to the server changes in the ABW and transferring the video stream that matches with the client bandwidth requirement [9]. Changes in the ABW can be estimated using passive or active methods. We focus on the passive method, which does not require probing additional data [8].

ABW can be estimated by modeling the client buffer, which is not ideal in the case of interactive streaming applications due to the lower expected response times from the client. As obtaining the lower layer statistics (packets per sec) is not yet standardized in Javascript API, it is challenging to implement a passive bandwidth estimation technique based on the information available at the HTML5 client browser without any plugins. In case of MPEG-DASH adaptive streaming method, at least one segment is already fetched to estimate the ABW, that leads to buffering delay of one segmented chunk [1]. Considering a delay-sensitive interactive FVV application, reacting to the current changes in the bandwidth maybe delayed until the completion of processing and buffering duration of last segment, that can lead to a delay in the order of seconds. We address this problem by passive bandwidth

estimation without basing on the segmentation into chunks, but instead on the open source browser statistics collected instantaneously from the HTML5 browser. We presented a DASH implementation using the open source Shaka-Player with JavaScript and evaluated the delay performance and overhead with our measurement framework.

Our contribution is twofold: (i) enabling passive bandwidth estimation support for a delay-sensitive interactive free-viewpoint application with lower response times, which is independent of segmented or buffered approach, and (ii) extracting the parameters being embedded into HTML5 documents, based on WHATWG version specification [3], that are used as an input to the bandwidth estimation algorithm, which reacts to the bandwidth fluctuations.

2 RELATED WORK

The literature on adaptive free-viewpoint streaming is multi-folded with focus on how to synthesize the views by reducing the redundancy, and adapt the switching view requested by the client. Authors in [4] discussed about the complexity and view switching latency of choosing the views spread on the server based on the given view position. However, the work does not illustrate a practical implementation on how the client can passively estimate the bandwidth and report it to the server. Work from [2] introduces the feasibility of using a HTML5 browser supporting the DASH segments and suggested a number of changes to the HTML5 video tag. The motivation from above work lead to further investigating on browser statistics to develop a prototype for adaptive video streaming of a FVV application.

Considering the work in client-side bandwidth estimation techniques, the download time of a segment can be used to compute a rough approximation of the ABW [6] by monitoring the buffer length and dropped frames [13] for an on-demand streaming application. In [11], a smoothed HTTP/TCP bandwidth estimation on the application layer is presented using the segment fetch time, which can be inaccurate due to the inconsistencies in time synchronizations between the server and the client. Findings from [12], [10] show that the delay components of adaptive video streaming systems comprises of segmentation on the server, fetching of media segments, download time, client buffering, decoding, and playback, which have an affect on the QoE of adaptive streaming applications [15]. From the previous works, using the segmentation and buffered approach to adapt with the current network conditions on a delay-sensitive FVV application can lead to delay in the order of seconds for the adaptation.

We propose a novel approach for an exclusive delay-sensitive FVV [19] application, that makes use of the non-segmented video delivery over HTML5 without using the existing DASH

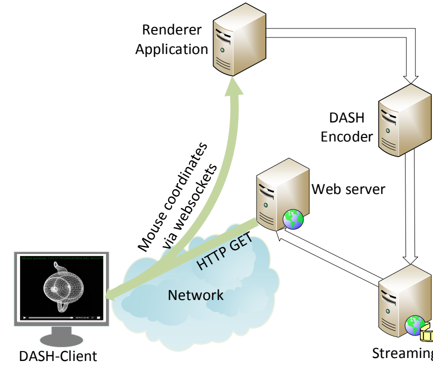

Figure 1: Architecture of the Free-Viewpoint application with the FFmpeg DASH encoding.

systems. We make an effort to demonstrate the feasibility of using open-source browser statistics with the help of WHATWG, direct the trend towards non-segmented approaches, and show the applicability under varying network conditions. The prototype is valid not only for FVV systems but also for on-demand and live streaming on HTML5.

3 SYSTEM OVERVIEW

This work demonstrates the use case of free-viewpoint streaming and rendering, where the user can control the view with different mouse positions on the browser [19]. The architecture from Fig. 1 consists of the DASH client-browser, renderer application, the DASH encoder, the streaming server, and the web server on the server-side. The client-server interaction including the feedback channel is shown in Fig. 2. In the following, system and implementation details with MP are described.The server-side rendering application is programmed using OpenGL libraries in C++ and can generate images up to 50 frames per second that enables thin clients using HTML5 to view the rendered images. The images created by the application are written to a named pipe, which are read and encoded by the FFmpeg encoder, streamed with FFserver, and decoded by HTML5 without plugins and third-party software [20]. Constant bit rate (1 Mbps) with HTTP/TCP streaming has been used to avoid firewall. The client sends an HTTP GET request to receive the HTML5 stream from the web server, and sends the desired mouse coordinates to the server along with the ABW changes through the websocket channel. The bandwidth adaptation algorithm is implemented on the client-side browser (google-chrome).

3.1 DASH implementation

Webm-DASH encoding is used to enable segmentation of the video stream to several chunks, and Shaka-Player is used as the client to decode the webm chunks. FFmpeg runs two processes: (i) encoding the images in real-time to webmchunks and (ii) creating the manifest file with the chunk identifier and the necessary time stamps. The encoded webm chunks

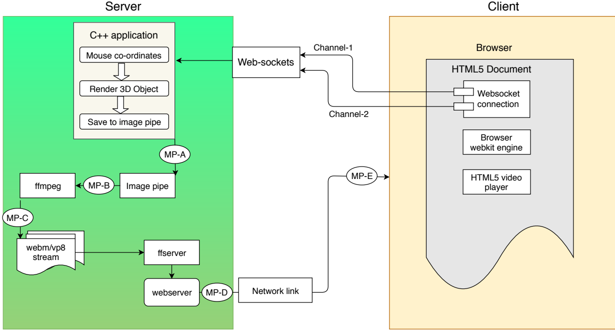

Figure 2: Free-viewpoint rendering application marked with Measurement Points (MP).

are of length equal to the Group of Pictures (GoP), and every chunk should start with a key frame. Shaka-Player uses JavaScript player library and loads the manifest file with the defined chunk identifier. The chunks are received with the HTTP GET request from the webserver and the media attribute from the Shaka-Player plays the downloaded chunks. The *suggestedPresentationDelay* is set to zero seconds.

3.2 Measurement Points and Delays

The latency in DASH implementation is observed and experimentally verified with the logging framework from FFmpeg source code. MP-A, MP-B (Fig. 2) denote the time-stamp after the video frame is rendered by the render software and fetched from the pipe with FFmpeg. The chunks created at MP-C in the webm container format are written to the tcp socket at MP-D. MP-E is the time-stamp when the first data segment is written to the socket. In our evaluation, we show the delay, overhead due to the segmentation i.e., between MP-C $\rightarrow$ MP-E. We evaluate three parameters: (i) initial delay (IDL) (ii) overhead and (iii) delay-offset from the live edge:

Initial delay (IDL). When the client requests for a specific chunk, the requested chunk is encoded by FFmpeg at MP-C and the first segment is received at MP-F. The IDL is dependent on the chunk size, which corresponds to one GoP. In our evaluation, we have shown the IDL for the first chunk.

Overhead. The interaction between Shakaplayer client requests to the server encoding software (FFmpeg) influences the delay experience and the overhead. Every time, should the client request for a new chunk, a HTTP GET request of 388 Bytes is sent across the network to the server.

Delay-offset. By default, Shaka-Player cannot download chunks faster than a GoP of one second. To elaborate on this, if the frame rate is 30 fps, a GoP of less than 30 leads to a delay offset; where the encoder (FFmpeg) is creating more chunks than the player can actually download at that instant of time. In the case of live or interactive streaming, this can lead to delays in the order of seconds. An experimental verification and analysis of the above effects is provided in Sec. 6

4 BANDWIDTH ADAPTATION

This section describes the bandwidth adaptation procedure. It describes the way performance metrics are being generated and utilized in our bandwidth adaptation algorithm (BWA).

4.1 Mining Video Quality Statistics

We rely on Webkit supported web browser engines to retrieve statistics on the received and decoded bytes and frames. In particular, the client-side Webkit engine exposes both metrics over JavaScript-based interfaces named $webkitDecodedByteCount$ and $webkitDecodedFrameCount$, respectively. They represent the cumulative count of bytes and frames, which have been decoded since the video started. Derivation over time (ever t= 1s) then yields the current bit rate sample, frame rate sample.

$$b = \frac{d(webkitDecodedByteCount)}{dt} \quad (1)$$

$$f = \frac{d(webkitDecodedFrameCount)}{dt} \quad (2)$$

4.2 Bit Rate Estimation

Our bit rate estimation is based on the implementation of the Shaka-Player, but is applied to a non-segmented video delivery. Bit rate estimation $\hat{b}$ depends, besides the currently sampled bit rate, on the history of past bit rate samples. Considering the current (n) and the previous ($n-1$) step in bit rate estimation, we compute the current $\hat{b}_{(n)}$ as follows.

$$\hat{b}_{(n)} = (1-\alpha) \cdot b_{(n)} + \alpha \cdot \hat{b}_{(n-1)} \quad (3)$$

The weight of these components can be adjusted by α. If α decreases then the weighted bit rate converges towards the current bit rate. Vice versa, an increasing α yields estimates more depending on the past bit rates.

$$\alpha = e^{\frac{log(\frac{1}{2})}{\eta}} \quad (4)$$

The additional parameter η defines the half-life of the estimation. The Shaka-Player can be run in a fast-adaptive mode, where the half-life is set to 3 seconds and a slow-adaptive mode, with 10 seconds half-life. In our implementation, a constant $\eta = 3s$ is used.

4.3 Desired Bit Rate

We define the desired reference bit rate b_r as:

$$b_r = \frac{f_r}{f} \cdot \hat{b} \quad (5)$$

The parameter f_r is a constant set on the client representing the desired frames per second (fps) for smooth video playback with decent quality while f is the actual frame rate (Eqn. (2)). f_r is equal the ideal frame rate on the client when the ABW is greater than the current instantaneous bit rate. In our scenario, f_r is set to 20 fps. In above equation, f_r/f represents a factor to adjust the sender's bit rate in dependence of the

estimated bit rate at the receiver. If $f_r/f = 1$ then the sender's bit rate is adequate. Otherwise, if this factor is greater than 1 then also b_r is greater than $\hat{b}$. That means the sender's bit rate must be increased accordingly. The algorithm calculates a reference bit rate value after the initial 10 seconds of video playback using Eqn. (5). The hypothesis is that the ABW is higher than the bit rate for the first 10 seconds, and the video stream has a smooth playback.

4.4 Switch Cases

Case 1: High quality to lower. ($\hat{b} < \beta \cdot b_r$)

Switch is triggered from higher to lower video quality if the weighted bit rate drops below $\beta \cdot b_r$. If the available bandwidth is less than the instantaneous bit rate of the video stream, cumulative *webkitDecodedByteCount* is reduced.

Case 2: Low quality to higher. ($\hat{b} > \beta \cdot b_r$)

The switch signal is sent if the weighted bit rate increases above b_r/β, and when the cross traffic does not influence the video quality. Hence, the ABW is more than the instantaneous bit rate of the application.

Algorithm 1 Bandwidth Adaptation Algorithm

Require: β (Sec. 4.4), τ (Sec. 4.4)

Main procedure.

1: **repeat**
2: every τ seconds:
3: Recompute b_r acc. to Eqn. (5)
4: Recompute $\hat{b}$ acc. to. Eqn. (3)
5: Call **Signalling of Bit Rate Switch** (b_r, $\hat{b}$, β) (Alg. 2)
6: **until** Service stopped
7: **return**

Algorithm 2 Signalling of Bit Rate Switch

Require: $b_r, \hat{b}, \beta$

Decision making process to trigger a WebSocket signal.

1: **if** $\hat{b} < \beta \cdot b_r$ **then**
2: WebSocket Signal (0, $\hat{b}$)
3: **else if** $\hat{b} > \frac{b_r}{\beta}$ **then**
4: WebSocket Signal (1, $\hat{b}$)
5: **end if**
6: **return**

4.5 Signalling via WebSockets

Changes in estimated $\hat{b}$ from the case 1 or case 2 (Sec. 4.4) are sent through the WebSocket channel, that listens for both the mouse coordinates (Fig. 2). First of the two parameters is a boolean flag (0 or 1) and the second one is $\hat{b}$. The server logic for adaption can be implemented based on either this flag or $\hat{b}$. The requested encoding rate is received by the application, and the proposed $\hat{b}$ can be used to notify the encoding bit rate, until there is a further change in the network conditions.

Table 1: Notations.

b	bit rate sample
b_r	desired bit rate
$\hat{b}$	estimated bit rate
f	frame rate sample
f_r	desired frame rate
α	balancing between current and past samples during bit rate estimation
β	threshold on bit rate gap used for signalling
η	half-life of bit rate estimation
τ	time interval for switching the sender's bit rate

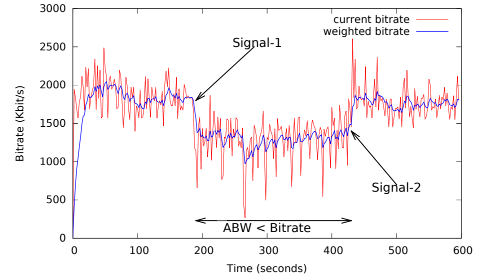

Figure 3: Bandwidth adaptation algorithm reacts to the fluctuations with signal-1 and signal-2.

With little overhead, server has the possibility to change the bit rate of the encoder to $\hat{b}$.

5 EXPERIMENTAL SETUP

We have conducted the experiments in our controlled research networking testbed environment based on Emulab [16] using a Dumbbell topology. The nodes representing servers, routers and clients are dedicated physical machines that are configured and connected using an Emulab configuration script. The nodes are connected to a central physical switch with 1 Gbps links. The switch creates virtual LANs (VLANs) based on the topology and assigns the requested bandwidth to the links. Network parameters like bandwidth, delay, packet loss can be configured on a traffic shaping delay node introduced between the bottleneck routers using the ipfw utility [18]. This creates a dynamic real-time networking scenario for testing the prototype. Changing the bottleneck link attributes between the routers influence the end-to-end capacity of the link from video server to the client. The cross traffic (CT) sender, receiver send and receive User Datagram Protocol (UDP) CT with Iperf [14]. During a particular flow, the instantaneous ABW is influenced by the magnitude of CT that goes through the bottleneck link between the routers.

6 RESULTS AND ANALYSIS

6.1 Video Analysis with BWA

For video quality anaylsis, we collected the HTML5 video statistics on the client. The application was run for 10 minutes

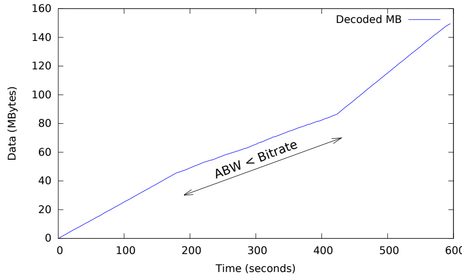

Figure 4: Cumulative data decoded by the client.

and UDP traffic was sent during the entire run that saturates the link to effect the ABW. The impact of network impairments on the received video throughput is shown with the help of mining the video specific attributes as described in 4.1 CT of 46 Mbps is sent over 50 Mbps capacity link for the first and last 3 minutes, that has no influence on $\hat{b}$, as the ABW is larger than the bit rate of the application (1.5 Mbps). CT between 180 to 420 sec. (Fig. 3) is increased to 49 Mbps, changing the ABW to 1 Mbps. If the ABW is lower than the $\hat{b}$ required for the smooth playback of the video, the application performance at the client is affected, and hence, the number of decoded bytes is reduced. This effect is shown in the Fig. 4. Thus, $\hat{b}$ falls to a value less than the factor of β and b_r, which is handled by case 1 in Sec. 4.4. The label Signal-1 in Fig. 3 points to this case and a WebSocket signal containing $\hat{b}$ is sent with the flag 0, i.e., message (0, $\hat{b}$). It is also noticed that the $\hat{b}$ smooths the effect of current bit rate fluctuations and the smoothing factor depends on the half-life η. After 420 sec., CT is reduced to 46 Mbps and the ABW is larger than the application bit rate. The $\hat{b}$ is increased by a quotient of b_r by β. This triggers a switch in the bitrate and the algorithm detects a change in the network conditions and the ABW by sending Signal-2 (Fig. 3) with the message (1, $\hat{b}$), where the server can adapt to the desired bitrate.

6.2 DASH Evaluation

The delays, overhead, and the delay-offset are shown in the Tables 2 and 3 respectively. The average delay from ten runs in table 2 is marginally larger than 1 sec. in the case of DASH implementation as it takes at least 1 sec. to create the next chunk with 1 sec. GOP and 30 fps. Hence, in the case of smaller GoP, the IDL is nearly equal to the network delay, i.e., delay between MP-E → MP-F. In case of NSS, the average delay for the 10 runs is 113 msec. The overhead from Tab. 3 increases with the increasing GoP, due to the Shaka-Player interaction and the frequency in sending the HTTP GET requests of the respective chunk identifier (CI). Overhead for 1000 video frames (33 chunks) has been measured with 30 fps. As the next requested chunks are already available at the encoding server before the next GET request, GoP-8 has the minimum overhead, which is approximately equal

Table 2: Average initial delay between MP-C → MP-F.

Streaming Type	Delay (s)
DASH	1.14926
NSS	0.113797

Table 3: Average overhead observed for 30 fps.

Group of Pictures	Overhead(KByte)	Delay offset (s)
Gop-8	13.58	7.83
Gop-16	19.01	2.34
GoP-30	26.38	1.14

to one GET request per chunk. Hence, there are no duplicate or multiple GET requests for the same CI. For GoP-16, there are some repeated GET requests observed for the same CI during the beginning of the session, where the encoder is slower in creating the chunks than the expected rate of GET requests. In the last case of GoP-30, there are duplicate and multiple GET requests for each CI, that causes additional overhead from the client. This is due to the corresponding chunks that are not available at the encoding server at the time of HTTP GET requests. In a lossy network and varying RTT, the overhead can increase, however for real-time interactive streaming application, a GoP of 1 sec. (30 fps) is recommended.

Considering 1 sec. as the highest frequency Shaka-Player can request the chunks from the encoding server, if the encoding software creates chunks smaller than 1 sec. GoP, the trade-off in delay (Tab. 3) shows that GoP-8 has the largest offset delay (OSD). As the Shaka-Player client is unable to fetch the segments or send HTTP GET requests at a rate approximately equal to 0.25 sec. A lag of 7.8 sec. is seen, which implies that if the encoder creates 50 chunks from the live stream, the client is downloading/requesting the chunk identifier with OSD of 7.8 sec. OSD is reduced to 0.5 sec. in GoP-16 due to the increase in the GoP.

7 CONCLUSIONS AND FUTURE WORK

We presented an adaptive streaming prototype for interactive FVV application without using DASH segmentation method, and introduced a client-side passive bandwidth estimation technique with open source technologies like HTML5 and WebSockets, which detects changes in the ABW. It is tested on a real networking testbed with varying cross traffic. The implications from current work can also be applied in one of the emerging fields, such as video analytics. Examples are provided in the work of [7] and [5], that addresses the issue with network bandwidth for improving the scalability, overhead associated with large video sets, the application of lightweight video analytics using WebSockets for adaptive streaming of 3D interactive video and camera arrays.

REFERENCES

[1] N. Bouzakaria, C. Concolato, and J. Le Feuvre. 2014. Overhead and performance of low latency live streaming using MPEG-DASH. In *IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications*. 92–97.

[2] Cyril Concolato, Jean Le Feuvre, and Romain Bouqueau. 2011. Usages of DASH for Rich Media Services. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*. ACM, New York, NY, USA, 265–270.

[3] Web Hypertext Application Technology Working Group. [n.d.]. *Video Metrics*. https://wiki.whatwg.org/wiki/Video_Metrics

[4] Ahmed Hamza and Mohamed Hefeeda. 2016. Adaptive Streaming of Interactive Free Viewpoint Videos to Heterogeneous Clients. In *Proceedings of the 7th International Conference on Multimedia Systems (MMSys '16)*. ACM, New York, NY, USA, Article 10, 12 pages.

[5] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodík, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. 2018. VideoEdge: Processing Camera Streams using Hierarchical Clusters. *2018 IEEE/ACM Symposium on Edge Computing (SEC)* (2018), 115–131.

[6] J. M. Jeong and J. D. Kim. 2015. Effective bandwidth measurement for Dynamic Adaptive Streaming over HTTP. In *2015 International Conference on Information Networking (ICOIN)*. 375–378.

[7] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 253–266.

[8] Sukhpreet Kaur Khangura and Markus Fidler. 2017. Available Bandwidth Estimation from Passive TCP Measurements using the Probe Gap Model. In *Proceedings of IFIP Networking*. IEEE.

[9] Doowon Kim, Jinsuk Baek, and Paul S. Fisher. 2014. Adaptive Video Streaming over HTTP. In *Proceedings of the 2014 ACM Southeast Regional Conference (ACM SE '14)*. ACM, New York, NY, USA, Article 26, 3 pages.

[10] L. S. Lam, J. Y. B. Lee, S. C. Liew, and W. Wang. 2004. A transparent rate adaptation algorithm for streaming video over the Internet. In *18th International Conference on Advanced Information Networking and Applications, 2004. AINA 2004.*, Vol. 1. 346–351 Vol.1.

[11] Chenghao Liu, Imed Bouazizi, and Moncef Gabbouj. 2011. Rate Adaptation for Adaptive HTTP Streaming. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems (MMSys '11)*. ACM, New York, NY, USA, 169–174.

[12] T. Lohmar, T. Einarsson, P. Fröjdh, F. Gabin, and M. Kampmann. 2011. Dynamic adaptive HTTP streaming of live content. In *2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*. 1–8.

[13] M. S. Mushtaq, B. Augustin, and A. Mellouk. 2014. HTTP rate adaptive algorithm with high bandwidth utilization. In *10th International Conference on Network and Service Management (CNSM) and Workshop*. 372–375.

[14] O. Olvera-Irigoyen, A. Kortebi, and L. Toutain. 2012. Available Bandwidth Probing for path selection in heterogeneous home Networks. In *2012 IEEE Globecom Workshops*. 492–497.

[15] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia. 2015. A Survey on Quality of Experience of HTTP Adaptive Streaming. *IEEE Communications Surveys Tutorials* 17, 1 (Firstquarter 2015).

[16] C. Siaterlis, A. P. Garcia, and B. Genge. 2013. On the Use of Emulab Testbeds for Scientifically Rigorous Experiments. *IEEE Communications Surveys Tutorials* 15, 2 (Second 2013), 929–942.

[17] Aljoscha Smolic. 2011. 3D video and free viewpoint video from capture to display. *Pattern Recognition* 44, 9 (Sep. 2011), 1958–1968.

[18] FreeBSD sys admin group. [n.d.]. *FreeBSD Man Pages*. https://www.freebsd.org/cgi/man.cgi?ipfw(8)

[19] Matthias Ueberheide, Felix Klose, Tilak Varisetty, Markus Fidler, and Marcus Magnor. 2015. Web-based Interactive Free-Viewpoint Streaming: A Framework for High Quality Interactive Free Viewpoint Navigation. In *Proceedings of the 23rd ACM International Conference on Multimedia*. 1031–1034.

[20] G. Zhu, F. Zhang, W. Zhu, and Y. Zheng. 2012. HTML5 based media player for real-time video surveillance. In *2012 5th International Congress on Image and Signal Processing*. 245–248.