Web-native Video Live Streaming

Luigi Lo Iacono
Cologne University of Applied Sciences
Cologne, Germany
Email: luigi.lo_iacono@fh-koeln.de

Silvia Santano Guillén
G&L Geißendörfer & Leschinsky GmbH
Cologne, Germany
Email: silvia.santano@gl-systemhaus.de

***Abstract*—The usage of the Web has experienced a vertiginous growth in the last few years. Watching video online has been one major driving force for this growth lately. Until the appearance of the HTML5 agglomerate of (still draft) specifications, the access and consumption of multimedia content in the Web has not been standardized. Hence, the use of proprietary Web browser plugins flourished as intermediate solution.**
With the introduction of the HTML5 video element, Web browser plugins are replaced with a standardized alternative. Still, HTML5 video is currently limited in many respects, including the access to only file-based media. This paper investigates on approaches to develop video live streaming solutions based on available Web standards. Besides a pull-based design based on HTTP, a push-based architecture is introduced, making use of the WebSocket protocol being part of the HTML5 standards family as well. The evaluation results of both conceptual principles emphasize, that push-based approaches have a higher potential of providing resource and cost efficient solutions as their pull-based counterparts.

Keywords-HTML5, Video, Live Streaming, DASH, WebSockets

I. INTRODUCTION

In the early days, video content has been delivered in the Internet by specific streaming protocols such as Real-Time Protocol (RTP) [1] or Real-Time Streaming Protocol (RTSP) [2] in conjunction with specialized server-side software to handle the stream. These protocols break up the streams – it can be more than one, such as a video and multiple audio channels – into very small chunks and send them from the server to the client. This method is also denoted as *push-based delivery*.

Such streaming protocols suffered, however, from unfavorable firewall configurations restricting in many cases the access to media data. HTTP progressive download has been developed partially to overcome this issue and to get multimedia streams past firewalls. The basic concept behind HTTP progressive download is to play back the media content while the resource is being downloaded from the Web server. This approach is also known as *pull-based delivery*, since the file containing the media data needs to be pulled from the server by a client's request.

While capable of finding the path from a requesting client to responding Web server, the HTTP progressive download still did not offer true streaming capabilities. This lack motivated the introduction of methods for adaptive streaming over HTTP. To provide a streaming behaviour, adaptive streaming over HTTP segments the media stream into small, easy-to-download chunks. The adaptiveness is realized by encoding the media content at multiple distinct bitrates and resolutions, creating different chunks of different qualities and sizes. The available encodings enable the client to choose between various bitrates and resolutions and then adapt to larger or smaller chunks automatically as network conditions keep changing. In order to inform the client about the offered video quality levels and the corresponding names of the resources, a meta file containing this information is provided by the server. The client then chooses a suitable quality level and starts requesting the small chunks in the order given in the meta file. This pull of media data needs to be performed by the client in a continuous manner in order to construct an enduring stream out of the obtained chunks. In an equivalent fashion an updated version of the meta file needs to be requested as well, so that the client retrieves information on upcoming chunks to request.

The arena of technologies for adaptive streaming over HTTP has been dominated by proprietary vendor-proposed solutions, as will be discussed in the subsequent Section II. To harmonize the scattered picture a standardized approach known as MPEG Dynamic Adaptive Streaming over HTTP (DASH) has been ratified in December 2011 and published by the International Organization for Standards (ISO) in April 2012 [3].

Although, adpative streaming over HTTP has been standardized and largely build upon Web standards, the play back still requires propietary extensions to be included into the Web browsers. Thus, from a perspective of a live video streaming that is truly Web-native, the following set of requirements need to be met:

- *Live content support*
  Delivering live content by the concept of chunk-based distribution.
- *Web-native*
  Building solely upon Web standards, so that no additional components are needed to develop and use the streaming services (e.g., by being HTML5-compliant on the client-side).
- *Minimal meta data exchange*
  Avoiding of extra message exchanges required for media stream control, e.g., by the adoption of communication patterns following the push model instead of the pull model.

 ISBN: 978-1-61208-333-9

- *Low protocol and processing overhead*
  Reducing overheads introduced by communication and processing means.

In the following section, available technologies will be described and analysed in the light of these requirements. After that, in Section III, the proposed approach will be introduced in terms of an architecture. Section IV then introduces an implementation of such architecture which serves as foundation for building various evaluation testbeds as described in Section V. Finally, a detailed discussion of the evaluation results obtained from performed test runs will conclude the contribution of the present paper.

II. Related work

Microsoft Smooth Streaming (MSS) [4] has been one of the first adaptive media streaming over HTTP announced in October 2008 as part of the Silverlight [5] architecture. MSS is an extension for the Microsoft HTTP server IIS (Internet Information Server) [6] that enables HTTP media streaming of H.264 [7] video and AAC [8] audio to Silverlight and other clients. Smooth Streaming has all of the typical characteristics of adaptive streaming. The video content is segmented into small chunks that are delivered over HTTP. As transport format of the chunks MSS makes use of fragmented ISO MPEG-4 [7] files. To address the unique chunks Smooth Streaming uses time codes in the requests and thus the client does not have to repeatedly download a meta file containing the file names of the chunks. This minimizes the number of meta file downloads that in turn allows to have small chunk durations of five seconds and less. This approach introduces, however, additional processing costs on the server-side for translating URL requests into byte-range offsets within the MPEG-4 file.

Apple's HTTP Live Streaming (HLS) [9] came next as a proposed standard to the Internet Engineering Task Force (IETF). As MSS it enables adaptive media streaming of H.264 video and AAC audio. At the beginning of a session, the HLS client downloads a play list containing the meta data for the available media streams, which use MPEG-2 TS (Transport Stream) [10] as wire format. This document will be repeatedly downloaded, every time a chunk is played back. The content is embedded into a Web page using the HTML5 video element, whose source is the m3u8 manifest file [11], so that both the parsing of the manifest and the download of the chunks are handled by the browser. Due to the periodic retrieval of the manifest file, there exists a lower bound for the minimal duration of the chunks, which is commonly about ten seconds.

With the announcement of HTTP Dynamic Streaming (HDS) [12] Adobe entered the adaptive streaming arena in late 2009. Like MSS and HLS, HDS breaks up video content into small chunks and delivers them over HTTP. The client downloads a manifest file in binary format – called bootstrap information (F4M) [13] – at the beginning of the session and periodically during its life time. As in MSS, segments are encoded as fragmented MP4 files that contain both audio and video information in one file. It, however, differs from MSS with respect to the use a single aggregate file from which the MPEG file container fragments are extracted and then delivered. In this respect, HDS follows the principle used in HLS instead, which requests and transmits individual chunks via an unique name.

These three major adaptive streaming protocols have much in common. Most importantly, all three streaming platforms use HTTP streaming for their underlying delivery method, relying on standard HTTP Web servers instead of special streaming servers. They all use a combination of encoded media files and manifest files that identify the main and alternative streams and their respective URLs for the player. And their respective players all monitor either buffer status or CPU utilization and switch streams as necessary, locating the alternative streams from the URLs specified in the manifest. The overriding problem with MSS, HLS and HDS is that these three different streaming protocols, while quite similar to each other in many ways, are different enough that they are not technically compatible. Indeed, each of the three proprietary commercial platforms is a closed system with its own type of manifest format, content formats, encryption methods and streaming protocols, making it impossible for them to work together.

Seeing the need for a universal standard for the delivery of adaptive streaming media over HTTP, MPEG decided to step into. MPEG DASH (Dynamic Adaptive Streaming over HTTP) [3] is an international standard for HTTP streaming of multimedia content that allows standard-based clients to retrieve content from any standard-based server. It offers the advantage that it can be deployed using standard Web servers. Its principle is to provide formats that enable efficient and high-quality delivery of streaming services over the Internet to provide very high user-experience (low start-up, no rebuffering, trick modes). To accomplish this, it proposes the reuse of existing technologies (containers, codecs, DRM, etc.) and the deployment on top of HTTP-CDNs (Web Infrastructures, caching). It specifies the use of either MPEG-4 or MPEG-2 TS chunks and an XML manifest file, the MPD (media presentation description), that is repeatedly downloaded to the client making it aware of which chunks are available.

	Support Live Streaming	Use HTML5 video element	Push delivery	Low overhead
HDS	✓	✗	✗	✗
HLS	✓	✓	✗	✗
MSS	✓	✗	✗	✗
DASH	✓	✓	✗	✗

Figure 1: Characteristics of HTTP-based adaptive live streaming platforms

Although the DASH standard may become the format of choice in the future, there is a lack of native Web browser integration. The DASH-JS [14] project from the University of Klagenfurt introduces an approach to overcome this gap. It proposes a seamless integration of the DASH standard into Web browsers using the HTML5 video element and the media source extensions [15]. The media source extensions are at the moment the only possibility to access the HTML5 video element via JavaScript, enabling a seamless playback of a chunk-based stream. The media source extensions are still a W3C working draft and they are currently only supported by the Chrome browser. As the segments are downloaded, this sequence is played back by feeding it chunk-wise into

 ISBN: 978-1-61208-333-9

a HTML5 video element and using media source extensions.

The characteristics of the discussed adaptive live streaming platforms over HTTP are summarized in the light of the requirements defined for a Web-native live video streaming (see Figure 1). As can be observed, none of the currently available platforms covers all of these characteristics, which motivates this implementation and research.

III. ARCHITECTURE

The basic idea of the proposed architecture is to ground the live streaming approach on a distinct communication protocol other than HTTP, which is still native to the Web but allows for a different communications design.

The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011 [16]. It has been designed for Web applications, and is at the moment supported by all major browsers such as Chrome, Internet Explorer, Firefox, Safari and Opera in their desktop as well as mobile occurrence. The protocol operates on top of a standard TCP socket and offers a bidirectional communication channel between a Web browser and a WebSocket server. The WebSocket is established by a HTTP-based opening handshake commonly operated on port 80 which preserves firewall-friendliness.

The code running on the browser side acts as client while there must be a server program running awaiting for connections, usually installed on a web server.

Figure 2 illustrates the architecture of the developed system, where the two different communication protocols used are represented, as well as a sample of the message exchange.

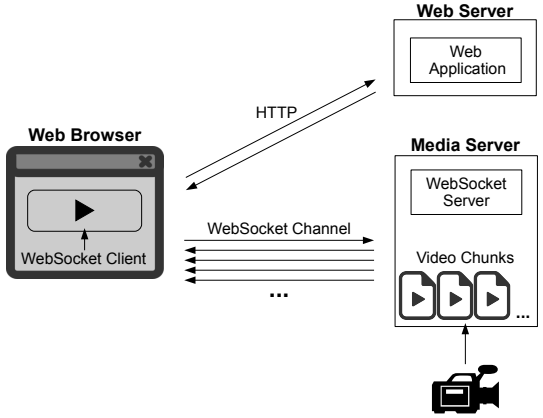

Figure 2: Proposed push-based architecture

The communication between the Web browser and the Web server will be the first to be executed, as for every website, via HTTP. After the web browser has downloaded the website, the JavaScript code on the Web Appication will attempt to start the communication via WebSocket with the Media Server.

The communication between client and media server starts with a two-way handshake, as can be seen in Figure 2, before the actual data transmission. The way the data transmission between the two parts takes place, facilitates its use for live content and real-time applications. This is achieved by enabling the server to send content without the need of the client asking first for it, creating a real bidirectional connection that remains open for both parts to send data at anytime.

The fact of being able to follow a push model is a great advantage for the purposes of this implementation, where a lot of real-time data needs to be sent, and will be sent from the server periodically, as soon as it is available instead of using a request-response procedure.

The second greatest advantage over implementing the same application over HTTP is the significant reduction in the length of the headers, which normally introduces an overhead of about 400 bytes for the request and about the same amount for the response on HTTP, while a regular header on WebSocket, as specified on RFC 6455, introduces an overhead of as low as two bytes, or maximumly up to 8 bytes, for an extended payload length.

IV. IMPLEMENTATION

A prototype implementation of the proposed pushed-based architecture has been developed. The technologies and components used for developing the prototype are depicted in Figure 3.

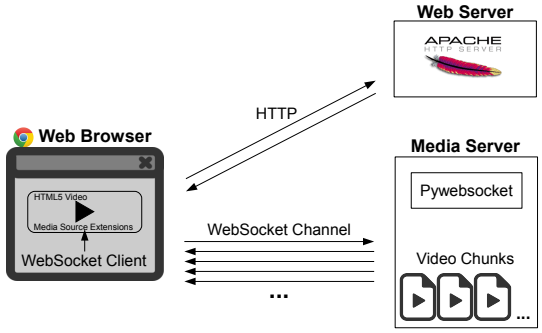

Figure 3: Implementation of push-based live video streaming prototype

The initial Web page is delivered by any HTTP server containing a JavaScript program, which gets downloaded by the browser. While being executed, the code creates an HTMLVideoElement object and a MediaSource object and connects them using the API. This API allows the construction of media stream objects for the HTML5 video element through which the media segments can be passed to the HTMLMediaElement for play back. Thus, the decoding and rendering parts will be natively handled by the browser.

In what follows, the client needs to create the WebSocket connection and to assign the according event listeners to specific functions waiting for the next content chunks to arrive so that they can be added to the corresponding media source buffer. This will be performed until the end of the session, which is reached either when the server has no more content to deliver or when the user decides to stop watching.

The WebSocket server application is implemented in Python language, using Pywebsocket [17], an extension for the Apache HTTP Server. This API makes possible to develop a server for the test, which resulted consuming very low RAM memory even for a large amount of clients connected, which is actually translated to a large amount of threads for the operative system. Just like most server applications, it does not

 ISBN: 978-1-61208-333-9

start connections by itself but waits for connection requests. After the establishment, the client applications emit a starting signal, with which the video session begins and remains open as long as there is more content available.

V. EVALUATION

To evaluate the proposed approach two distinct testbeds have been implemented. One (browser-based, in JavaScript) is targeting the amount of meta data, i.e., data not part of the video, required to be exchanged between client and server. The second (not browser-based, in Python) is concerned with the processing overhead on the server-side and the number of simultaneous clients servable from one server instance. These testbeds have been realized for both a DASH-like HTTP transfer and the proposed WebSocket-based approach.

To perform the first evaluation, two different browser-based clients have been developed. The version over HTTP avails itself of Apache HTTP server and the one over WebSocket, after establishing the connection, of our WebSocket server application.

To perform the second evaluation, for the client-side the programs have been implemented using the Python modules websocket-client [18] and httplib [19], respectively. The server-side of the HTTP approach is programmed on top of the HTTP protocol implementations provided by the Python modules BaseHTTPServer [20] and SocketServer [21]. Based on these components, the implementation of a multi-threaded HTTP and WebSocket server has been undertaken. The server-side of the WebSocket approach is the same described on previous section.

The video used to perform the evaluation is the open source movie *Big Buck Bunny* [22], which has been produced by the Blender Foundation and has been released under Creative Commons License Attribution 3.0 [23]. The AVC codec is used in an MP4 container. The test video's bitrate is 100 kbps, the duration is 9' 56" and the total file size is 6.7 MB (6,656,763 bytes).

To simulate a live stream, the movie has been chunked into separate segment files according to the MP4 standard. These segements contain each a short portion of two seconds of duration and are stored in the media server. Since the chunk length is approximately two seconds, the number of chunks is 300.

A. Communication Overhead

To gather the overhead introduced by each one of the two investigated communication alternatives, the network traffic has been captured, analysed and contrasted with theoretical thoughts. The network packets exchanged in both scenarios have been captured using Wireshark [24].

Each layer of the TCP/IP model introduces its own meta data in form of a header and in some cases even a trailer, but since Ethernet, IP and TCP are common to both compared approaches, only the protocol elements of the application-level are taken into account, which are the HTTP messages and the WebSocket frames respectively.

Figure 4 shows the typical size of an HTTP GET request for retrieving the next video chunk which has in this particular case a size of 440 bytes.

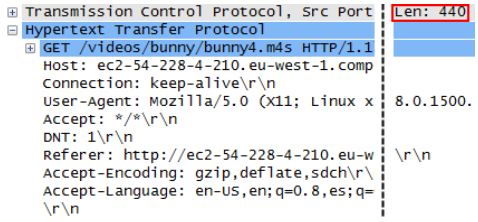

Figure 4: Captured HTTP request asking for video chunk #4

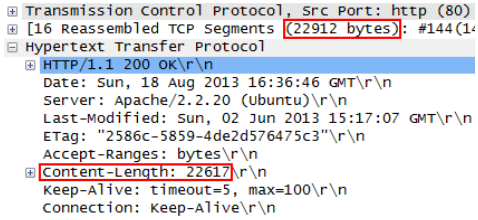

Figure 5: Captured HTTP response containing video chunk #4

Figure 5 presents the size of an corresponding HTTP response packet. The upper-most mark in the figure shows that a total of 22,912 bytes have been transmitted in the HTTP response. From the HTTP content-length header the amount of video bytes contained in this chunk can be retrieved, which is 22,617 bytes. With these two values, the size of the HTTP response header can be calculated (300 bytes). This makes a final amount of metadata of 740 bytes per chunk (440 bytes for the whole request and 300 bytes for the response header). This again sums up to an overall overhead of 222,000 bytes when considering all of the 300 chunks. For Transmitting the test video of the size of 6,656,763 bytes, this method introduces an overhead of 3.3% in relation to the media content.

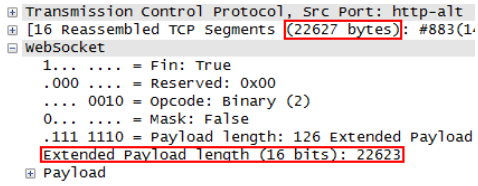

Figure 6: Captured WebSocket frame containing video chunk #4

The WebSocket protocol specification defines the header as a variable size structure ranging from a size of at least two bytes to a maximum of 8 bytes. This mainly depends on the size of the payload carried by the WebSocket packet, since this is encoded in a length field in the header which grows depending on the actual content size. In case of a minimal two bytes header, the payload of the WebSocket frame can contain a maximum of 125 bytes. Since all of the 300 two seconds video segments are in any case larger than this mark, the resulting WebSocket packets do all have a header of four bytes, as can be observed from the captured WebSocket frame shown in Figure 6. This is due to a required extended payload length header field, which introduces additional two bytes. With this two byte extended payload length header field a maximum of 65,662 bytes of payload can be specified, which is large enough for all of the 300 video chunks.

Since there are no requests required to retrieve a next video chunk, this communication overhead from the DASH-like approach is not inherent to the proposed WebSocket-based transmission. Thus, the total amount of meta data introduced per chunk is four bytes (zero bytes for the request since it

 ISBN: 978-1-61208-333-9

does not exist and four bytes for the header in the WebSocket frame). For all of the 300 chunks this sums up to a total of 1,200 bytes for transferring the video from the server to the web client. This represents an overhead of around 0.02% in relation to the plain multimedia content of 6,656,763 bytes.

When observing carefully the numbers given in the Figures 5 and 6 it appears that the sizes of the payloads found in the HTTP response and the WebSocket frame differ by six bytes. This constant six byte offset can be found in any WebSocket frame in comparison to the corresponding HTTP response. This is due to additional meta data added by the WebSocket implementation used in this testbed (binaryjs [25]). Thus, the concrete WebSocket framework and libraries used for development need to be examined whether they add additional meta data to the payload, since this has an influence on the overall efficiency. In this particular case, the exchanged meta data sums up to a total of 3,000 bytes, which represents an overhead of around 0.05% in relation to the plain multimedia content of 6,656,763 bytes.

B. Processing Overhead

To further examine the potential benefits of the proposed approach of using WebSockets as communication means for video live streaming in the Web, an additional testbed has been developed and operated, aiming at finding out the total quantity of clients that one server is able to handle simultaneously. Again, two equivalent instantiations of the testbed have been deployed for the DASH-like and for the WebSocket-based live video streaming.

The machine used for this evaluation is an Amazon EC2 small instance server composed of one 64 bit ECU (EC2 Compute Unit) which provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor and 1.7 GB of RAM [26]. To simulate a large number of clients a set of 15 distinct and distributed EC2 micro instances have been deployed. An EC2 micro instance is equipped with up to 2 ECUs for short periodic bursts and 613 MB of RAM. The developed components described in Section IV have been installed on these systems in order to setup and operate the testbeds. When building such a large scale testbed, the OS settings for the maximum number of open files per user, the maximum number of threads and the maximum number of TCP connections need to be modified accordingly.

The clients are all set up at the same time. At the moment the last of them connects to the server, all of them start being simultaneously served with the test video. After each client instance has received all content, it measures its own duration time, measured from the moment it started receiving content, to calculate the bitrate as follows:

Bitrate [bps] = Video size [bits] / Transfer time [s].

As mentioned previously, the video encoding bitrate is around 100 kbps. Hence, as long as the receiving bitrate is higher than the video bitrate, the user will be able to watch the video without encoutering any disturbance. The moment in time when the number of clients is so big that theythe majority of them can not be served anymore at the required minimum bitrate will be considered as the inflexion point. The expected theoretical results of these tests are shown in Figure 7, with a red dot simbolizing the defined inflexion point.

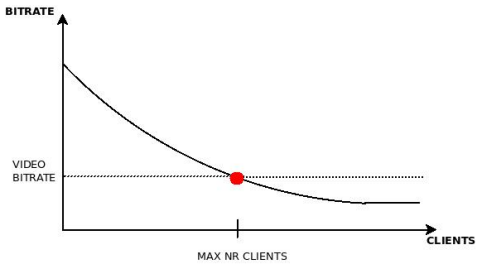

Figure 7: Expected curve of transmission bitrate

The number of clients has been increased stepwise starting from 100 clients. On each run, all clients have been equally distributed on 15 separate machine instances. Each run has been repeated 10 times to obtain a mean value. In each additional run, the server is restarted and the number of concurrent clients is increased by 100, until reaching 2,000 clients in the final run.

Figure 8 shows the results obtained from the DASH-like live streaming testbed. It can be observed that the graph for HTTP transmission bitrate shows a corresponding shape as theoretically expected and depicted in Figure 7.

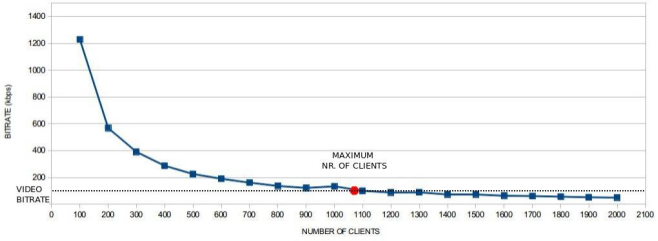

Figure 8: Average transmission bitrate for DASH-like streaming

The bitrate decreases from an average of 1,228 kbps, when there are 100 simultaneous clients to an average of 49 kbps, when the number of connected clients increases to 2,000. The red point indicates the inflexion point, which lies between 1,000 and 1,100 active clients. This denotes the largest quantity of simultaneous clients for this server, so that the minimum required video bitrate can still be served to the connected clients.

Figure 9 summarizes the results obtained from the WebSocket-based live streaming testbed. The bitrate decreases from an average of 4,067 kbps, when there are 100 simultaneous clients to an average of 170 kbps, when the number of active clients increases to 2,000. Thus, the WebSocket-based video server can still handle as much as 2,000 simultaneous clients and provide each with a video stream that comes with a bitrate still above the required encoding bitrate of 100 kbps.

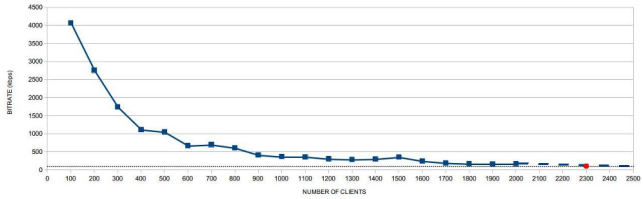

Figure 9: Average transmission bitrate for WebSocket-based streaming

The tests runs have been performed in both cases until

2,000 concurrent clients have been reached. Further measurements in the WebSocket-based testbed have not been performed. When extrapolating the obtained results, then the inflexion point will be located at around 2,300 clients (see Figure 9).

From these experiments it can be deduced, that besides the communication overhead advantages, the proposed WebSocket-based live streaming approach has additional benefits in terms of processing costs. These efficiency advantages result in a larger user base being servable with the same amount of infrastructure ressources.

VI. Conclusions

The access of video content in the web is evolving rapidly. Currently, file-based video content is dominating whereas the consumption of live streams is on the raise. The available standards and technologies for enjoying live video content in a web-native manner are, however, still in their infancy. The HTTP-based DASH is a first step in this direction.

The adoption of HTTP for video distribution in the web has its pros and cons. For the on-demand access of file-based videos the comprehensive and pervasive HTTP guarantees a broad usage of the content. This approach also fits well with the current deployment and usages of CDNs (Content Distribution Networks), ensuring the necessary scaling of such an approach.

Things change, however, if it comes to live streaming of video content. First, CDNs can not exploit their strength, since the feeding of the content to the distributed cache servers does not adhere to the real-time character of live video streams. The idempotence of the HTTP GET method is henceforth less relevant for live casts and brings other drawbacks of HTTP back in focus. The client-initiated request-response communication pattern is one major source of issues when push-based communications need to be implemented as it is the case for the transmission of media content.

This paper examined the possiblity of developing a live video streaming solution in a web-native manner by means of standards belonging to the HTML5 standards family. Such an approach has been realized based on the HTML5 video element and WebSockets as real-time communication means. The performed evaluation of the developed video streaming solution demonstrates that this approach is much more efficient compared to methods relying on HTTP. Both, the communication as well as the processing overheads can be significantly reduced by the proposed WebSocket-based solution in comparison to HTTP-relying methods such as DASH.

Future research activities will focus on the relation of CDNs and connection-oriented protocols such as the WebSocket protocol. The lack of the idempotence property and the real-time nature of such content stream pose new requirements and challenges to such caching and load distribution systems.

References

[1] Audio-Video Transport Working Group, “Rtp: A transport protocol for real-time applications,” IETF, RFC 1889, 1996, online available at www.ietf.org/rfc/rfc1889 (last accessed: Jan 2014).

[2] H. Schulzrinne, A. Rao, and R. Lanphier, “Real time streaming protocol (rtsp),” IETF, RFC 2326, 1998, online available at www.tools.ietf.org/html/rfc2326 (last accessed: Jan 2014).

[3] ISO/IEC Moving Picture Experts Group (MPEG), “Dynamic adaptive streaming over http,” ISO/IEC, Tech. Rep., 2013, online available at www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57623 (last accessed: Jan 2014).

[4] Microsoft, “Microsoft smooth streaming,” www.iis.net/downloads/microsoft/smooth-streaming (last accessed: Jan 2014).

[5] Microsoft Corporation, “Silverlight 5.1,” 2013, online available at www.microsoft.com/silverlight (last accessed: Jan 2014).

[6] Microsoft, “Internet information services,” www.iis.net (last accessed: Jan 2014).

[7] ISO/IEC Moving Picture Experts Group (MPEG), “Iso mpeg-4,” ISO/IEC, International Standard, May 2012, online available at www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=61490 (last accessed: Jan 2014).

[8] ISO/IEC Moving Picture Experts Group (MPEG), “Advanced audio coding,” ISO/IEC, International Standard, 2004, online available at www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=62074 (last accessed: Jan 2014).

[9] Apple Inc., “Http live streaming,” IETF, Internet-Draft, 2013, online available at www.tools.ietf.org/html/draft-pantos-http-live-streaming-12 (last accessed: Jan 2014).

[10] ISO/IEC Moving Picture Experts Group (MPEG), “Iso mpeg-ts,” ISO/IEC, International Standard, 2013, online available at www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=62074 (last accessed: Jan 2014).

[11] Apple Inc., “Http live streaming - section 3: Playlist file,” IETF, Internet-Draft, 2013, www.tools.ietf.org/html/draft-pantos-http-live-streaming-12 - Section 3 (last accessed: Jan 2014).

[12] Adobe, “Adobe http dynamic streaming,” www.adobe.com/products/hds-dynamic-streaming.html (last accessed: Jan 2014).

[13] Adobe, “Flash media manifest file format specification 1.01,” 2010, online available at osmf.org/dev/osmf/specpdfs/FlashMediaManifestFileFormat\-Specification.pdf base(last accessed: Jan 2014).

[14] B. Rainer, S. Lederer, C. Muller, and C. Timmerer, “A seamless web integration of adaptive http streaming,” in Signal Processing Conference (EUSIPCO), 2012 Proceedings of the 20th European, 2012, pp. 1519–1523.

[15] A. Colwell, A. Bateman, and M. Watson, “Media source extensions,” W3C, Last Call Working Draft, 2013, online available at www.dvcs.w3.org/hg/html-media/raw-file/tip/media-source/media-source.html (last accessed: Jan 2014).

[16] I. Fette and A. Melnikov, “The websocket protocol,” IETF, RFC 6455, 2011, online available at www.tools.ietf.org/html/rfc6455 (last accessed: Jan 2014).

[17] T. Yoshino, “Pywebsocket,” www.pypi.python.org/pypi/mod_pywebsocket (last accessed: Jan 2014).

[18] H. Ohtani, “websocket-client,” www.pypi.python.org/pypi/websocket-client/0.7.0 (last accessed: Jan 2014).

[19] Python Software Foundation, “Http protocol client,” www.docs.python.org/2/library/httplib.html (last accessed: Jan 2014).

[20] Python Software Foundation, “Basehttpserver,” www.docs.python.org/2/library/basehttpserver.html (last accessed: Jan 2014).

[21] Python Software Foundation, “Socketserver,” www.docs.python.org/2/library/socketserver.html (last accessed: Jan 2014).

[22] Blender Foundation, “Big buck bunny,” 2008, www.bigbuckbunny.org (last accessed: Jan 2014).

[23] Creative Commons, “Creative commons license attribution,” 2007, ”www.creativecommons.org/licenses/by/3.0/us/legalcode” (last accessed: Jan 2014).

[24] Open Source, “Wireshark 1.10.2,” 2013, www.wireshark.org (last accessed: Jan 2014).

[25] E. Zhang, “Binaryjs,” 2013, www.binaryjs.com.

[26] Amazon Web Services, “Amazon elastic compute cloud (ec2),” www.aws.amazon.com/en/ec2 (last accessed: Jan 2014).

 ISBN: 978-1-61208-333-9