Efficient and Adaptive Web-native Live Video Streaming

Luigi Lo Iacono* and Silvia Santano Guillén†
*Cologne University of Applied Sciences, Cologne, Germany
Email: luigi.lo_iacono@fh-koeln.de
†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 VideoElement, 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. In addition, initial approaches to instrument the proposed push-based architecture with adaptiveness to network conditions have been developed.**

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

I. Introduction

The access of video content in the Web is evolving rapidly, as the internet traffic increases, with live video streaming becoming web-native as well [1]. According to the Cisco Visual Networking Index Global Forecast and Service Adoption for 2013 to 2018 [2], consumer Internet traffic has increased enormously on the last years and the expectations are that this trend continues growing due to more users and devices, faster broadband speeds and more video viewing. Already today, monthly web traffic is at 62 exabytes a month, meaning a 62% of the whole traffic. The forecast includes concrete numbers on Internet traffic predictions, indicating that the annual global IP traffic will surpass the zettabyte (1000 exabytes) threshold in 2016. In other words, it will reach 91.3 exabytes (one billion gigabytes) per month in 2016, going up to 131.6 exabytes per month in 2018. The latter figure means that it would take an individual over 5 million years to watch the amount of video that will be crossing global IP Networks in one month in 2018.

The increase in Internet usage is mainly driven by online video consumption, for which the expected proportion of all consumer Internet traffic will account 79% in 2018, up from 66% in 2013. Moreover, adding related forms of video distribution to this calculation such as video on demand and video exchanged through peer-to-peer file sharing would mean between 80 to 90% of global consumer traffic. Regarding to the number of users online video will then be the fastest growing Internet service with a Compound Annual Growth Rate (CAGR) of 10 percent from 2013 to 2018, growing from 1.2 billion users to 1.9 billion users by 2018, as shown in Figure 1.

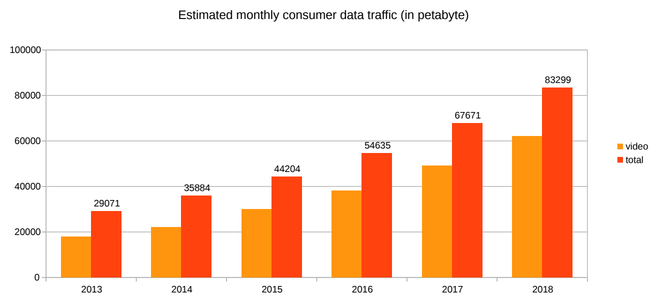


Figure 1: Estimated monthly consumer video traffic and total data traffic on the Internet [2]

In the early days of the Internet, video content has been delivered by specific streaming protocols such as Real-Time Protocol (RTP) [3] or Real-Time Streaming Protocol (RTSP) [4] 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 unfavourable firewall configurations restricting in many cases the access to media data. HTTP progressive download [5] 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 reliably finding the path from a requesting client to a responding Web server, HTTP progressive download still not offers 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 chunks of the same media fragment in different qualities and sizes. The available encodings enable the client to choose between the provided quality levels and then adapt to lower or higher definition 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 corresponding chunks in order and with the file named specified 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 periodically 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 [6]. Although, adaptive streaming over HTTP has been standardized lately 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 native to the Web, the following set of requirements can be defined as necessary foundation:

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

In the following section, available technologies will be reviewed in the light of these requirements. After that, in Section III, the proposed approach of a Web-native live Video Streaming will be introduced in terms of an architecture. Section IV then introduces an implementation of the introduced architecture, which serves as foundation for building various evaluation test beds 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) [7] has been one of the first adaptive media streaming over HTTP announced in October 2008 as part of the Silverlight [8] architecture. MSS is an extension for the Microsoft HTTP server IIS (Internet Information Server) [9] that enables HTTP media streaming of H.264 [10] video and AAC [11] audio to Silverlight and other clients. Smooth Streaming has all 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 uses fragmented ISO MPEG-4 [10] 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) [12] 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) [13] as wire format. This meta data document will be repeatedly downloaded, every time a chunk is played back. The media content is embedded into a Web page using the HTML5 VideoElement [14], whose source is the m3u8 manifest file [15], 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. A drawback of HLS is the current lack of client platforms support, with an availability at the moment mostly restricted to iOS devices and desktop computers with Mac OS X [16]. At the moment, the support for Android Operative System is only available on a few Android devices running Android 4.x and above, although still presenting several inconsistencies and difficulties. Moreover, for desktop no other web browser than Safari has native support for HLS and specific player plugins are therefore necessary in other browsers to play the streams back [17].

With the announcement of HTTP Dynamic Streaming (HDS) [18] 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, the Flash Media Manifest (F4M) [19], 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 metadata file from which the MPEG file container fragments are determined and then delivered. In this respect, HDS follows the principle used in HLS instead, which requests and transmits individual chunks via a 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 specialized 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 overarching 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 not to be 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.

As introduced in Section I, it is a well-known fact that the consumption of video on the Web is growing every day and, moreover, consumers are moving from desktop computers to smartphones, tablets, and other mobile devices to watch video. All these devices present huge differences in compatibility. Despite that, the same experience is expected on all of them, maintaining the high quality and availability. To enable the delivery of video to any platform, a number of streaming protocols and different applications have to be supported. The situation would change greatly if it was possible to have a single distribution method and a single cross-platform client application. On the other hand, removing the requirement of installing plugins on the client side removes a significant obstacle for many users. Furthermore, for cross-platform compatibility, security and stability, many browser vendors have already decided they are not supporting plugins in the future. All those reasons have as a consequence the intention of avoiding solutions that involve plugins and opt for a Web-browser-native approach.

Recognizing this need for a universal standard for the delivery of adaptive streaming media over HTTP, the MPEG standardisation group decided to step into. MPEG DASH (Dynamic Adaptive Streaming over HTTP) [6] 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 Content Distribution Networks (CDN). It specifies the use of either MPEG-4 or MPEG-2 TS chunks and an XML manifest file, the so-called Media Presentation Description (MPD), 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 2: 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. Initial steps towards a browser-native integration of MPEG-DASH using the HTML5 VideoElement have been undertaken. In these implementations, the browser is in charge of parsing, decoding and rendering the media data while, traditionally, applications like Adobe Flash or Microsoft Silverlight have been used for these features, in form of plugins. With JavaScript most of them can be achieved but still remains as a pitfall how to feed media data to the HTML5 VideoElement, a core issue to enable live streaming and adaptive streaming, where the source to play is a sequence of chunks and therefore, the 'src' parameter needs to be updated accordingly.

On this research several different solutions have been investigated in order to overcome the problem of the integration of DASH with HTML5 VideoElement, described in detail in Section IV.

The DASH-JS [20] project from the University of Klagenfurt introduces an approach to overcome this lack of native Web browser support. It proposes a seamless integration of the DASH standard into Web browsers using the HTML5 VideoElement and the MediaSource extensions [21]. The MediaSource extensions enable a seamless playback of a chunk-based stream, by defining a JavaScript API, which allows media streams to be constructed dynamically. They are still a W3C working draft and are currently supported by some of the major browsers: Chrome, Firefox Internet Explorer, Safari, Windows Phone 8.1 and Chrome for Android. This API solves the problem simplifying the process, taking care of the playing as the segments are downloaded and creating a sequence that is played back by feeding it chunk-wise into the HTML5 VideoElement. The media segments are downloaded and appended to a MediaSource buffer and this sequence will then be used as source for the HTML5 VideoElement. The first of the chunks consists of initialization data, which has to be appended to the buffer on the first place. After this initialization data has been loaded, the media segments will be retrieved and played back in the required sequence. Listing 1 shows the major parts of the MediaSource API and their usage in order to construct a steady media stream constructed out of the media fragments downloaded from the Web server via continuous requests issued by the XMLHttpRequest (XHR) [22] API. The second part is repeated as long as the session is open.

Figure 2 summarizes the characteristics of the discussed adaptive live streaming platforms over HTTP in the light of the requirements defined for a Web-native live video streaming. As can be observed, none of the currently available platforms covers all of these characteristics, leaving space for further research and development.

Listing 1: Usage of the MediaSource API for fragmented media access.

```
//URL of next chunk
var url = (...);

var mediaSource = new MediaSource();
var video = document.querySelector('video');
video.src =
   window.URL.createObjectURL(mediaSource);

(...)

var sourceBuffer =
   mediaSource.addSourceBuffer(
   'video/mp4; codecs="avc1.42c00d"');
var xhr = new XMLHttpRequest();
xhr.open('GET', url, true);
xhr.responseType = 'arraybuffer';
xhr.onload = function(e)
{
   data = new Uint8Array(this.response);
   if (data == "false")
      mediaSource.endOfStream();
   else
      videoSource.appendBuffer(data);
};

xhr.send();
```

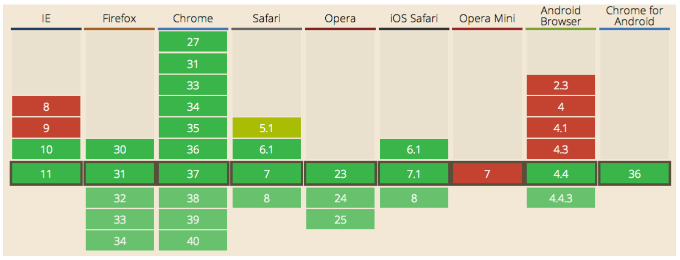
III. Architecture

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 [23]. As a flanking W3C standard, the WebSocket JavaScript API [24] provides an entirely event-driven interface for browser applications to use the WebSocket protocol. WebSockets are supported by all major browsers such as Chrome, Internet Explorer, Firefox, Safari and Opera in their desktop as well as mobile occurrence, as shown in Figure 3.

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 waiting for connections, usually installed on a Web server.



IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Chrome for Android
		27						
		31						
		33					2.3	
8		34					4	
9		35	5.1				4.1	
10	30	36	6.1		6.1		4.3	
11	31	37	7	23	7.1	7	4.4	36
	32	38	8	24	8		4.4.3	
	33	39		25				
	34	40						

Figure 3: WebSocket browser support [25]

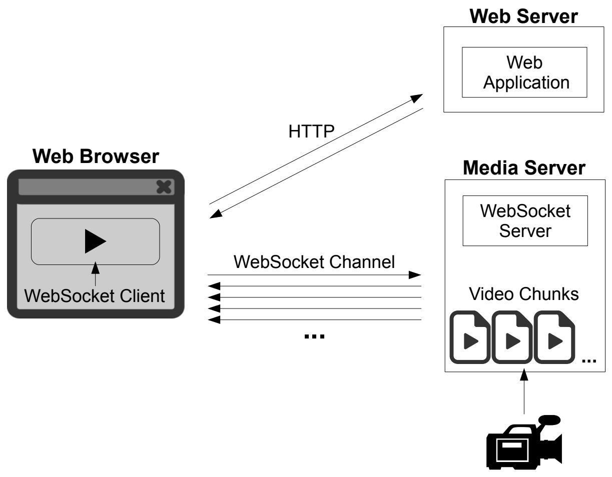


Figure 4: Proposed push-based Web-native live Video Streaming architecture

Figure 4 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.

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 Application 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 4, 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 any time.

The fact of being able to follow a push model is the core principle of this architecture, 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.

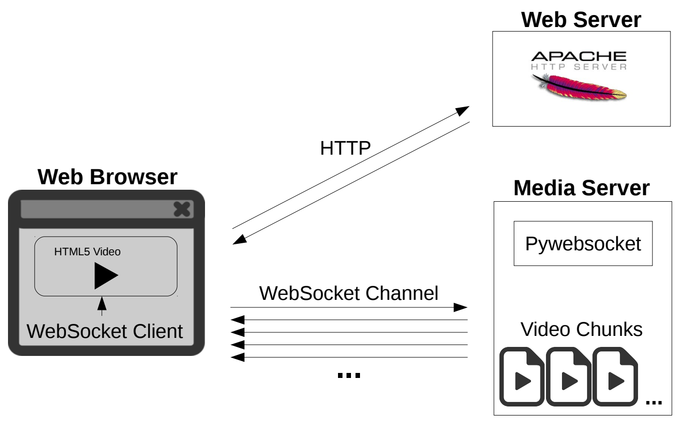


Figure 5: Prototype implementation of the proposed push-based Web-native live Video Streaming architecture

IV. Implementation

A prototype implementation of the proposed pushed-based architecture has been developed as a foundation for analysing the properties of the introduced approach. The technologies and components used for developing the prototype are depicted in Figure 5.

The initial Web page is delivered by an HTTP server and contains a JavaScript program, which gets downloaded by the browser. The browser will also be in charge of rendering the video while it is being downloaded.

As introduced at the beginning of this paper, for this research several different solutions have been investigated in order to overcome the problem of the integration of DASH with the HTML5 VideoElement. The trivial approach of updating the source of the VideoElement when the previous chunk has come to an end gives as a result quite a noticeable gap between sources for the user watching. Another approach, from which a better seamless switch could be expected, utilizes two VideoElements at the same position. This can be understood as one on top of the other in a certain way, letting at every moment just one of them to be visible. In this case, the next chunk to play would always be loaded on the hidden element in advance, carrying out the switch at the exact end of the playing chunk. The behaviour this technique produces is in fact very similar to the previous approach, showing gaps between the chunks to play for a short time during the switch.

Furthermore, the use of some publicly available APIs to reproduce a list of files using the VideoElement, such as SeamlessLoop 2.0 for JavaScript [26] and an own version of it replacing the audio by a VideoElement has also been considered, without success solving the problem.

As a consequence, from all options investigated the only remaining possibility to overcome this issue at the moment of implementing this system is to use the already mentioned MediaSource API for the implementation.

Afterwards, the JavaScript code, which will be executed on the client after downloading, creates an HTML5 VideoElement object and a MediaSource object and connects them using the API. This API allows the construction of media stream objects for the HTML5 VideoElement through which the media segments can be passed to the HTML5 VideoElement 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 MediaSource 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 [27], 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 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 test beds have been implemented. Test bed A (browser-based, in JavaScript) is targeting the amount of metadata, i.e., data not part of the video, required to be exchanged between client and server. Test bed A (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 test beds have been realized for both a DASH-like HTTP transfer and the proposed WebSocket-based approach.

To perform the first evaluation (Test bed A), two browser-based clients have been developed: a version over HTTP, which avails itself of Apache HTTP server and another one over WebSocket, which, after establishing the connection, connects to our WebSocket server application.

To perform the second evaluation (Test bed B), all components have been implemented in Python language. For the clients, the modules used are websocket-client [28] and httplib [29], respectively. The server-side of the HTTP approach is programmed on top of the HTTP protocol implementations provided by the Python modules BaseHTTPServer [30] and SocketServer [31]. 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 in previous section.

The video used to perform the evaluation is the open source movie *Big Buck Bunny* [32], which was produced by the Blender Foundation and has been released under Creative

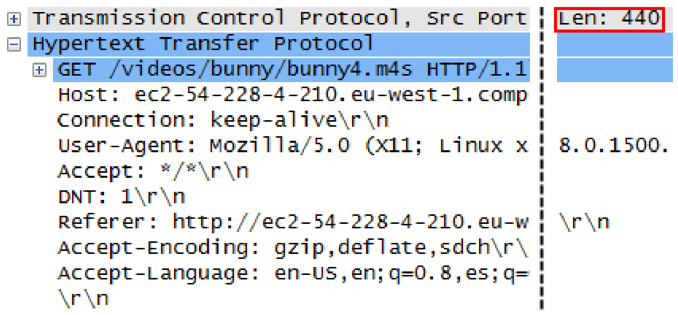
Commons License Attribution 3.0 [33]. The AVC codec is used in an MP4 container. The test video's bitrate is 100 kbps, the duration is 9' 56" and the total file size is 6.7 MB (6,656,763 bytes).

To simulate a live stream, the movie has been chunked into separate segment files according to the MP4 standard. These segments 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 [34].

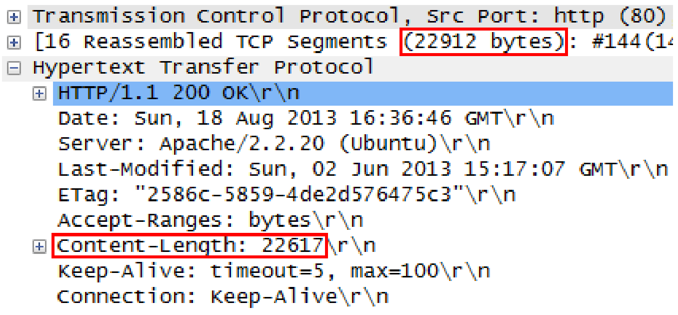
Each layer of the TCP/IP model introduces its own metadata in form of a header and in some cases even a trailer, but since Ethernet, IP and TCP are common to both compared approaches, only the protocol elements of the application-level are taken into account, which are the HTTP messages and the WebSocket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



```
⊞ Transmission Control Protocol, Src Port ¦ Len: 440
⊟ Hypertext Transfer Protocol
  ⊞ GET /videos/bunny/bunny4.m4s HTTP/1.1
    Host: ec2-54-228-4-210.eu-west-1.comp
    Connection: keep-alive\r\n
    User-Agent: Mozilla/5.0 (X11; Linux x ¦ 8.0.1500.
    Accept: */*\r\n
    DNT: 1\r\n
    Referer: http://ec2-54-228-4-210.eu-w ¦ \r\n
    Accept-Encoding: gzip,deflate,sdch\r\
    Accept-Language: en-US,en;q=0.8,es;q=
    \r\n
```

Figure 6: Captured HTTP request asking for video chunk #4

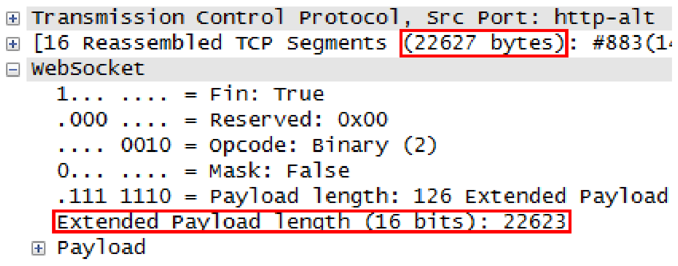
Figure 6 shows the typical size of an HTTP GET request for retrieving the next video chunk, which has in this particular case a size of 440 bytes.



```
⊞ Transmission Control Protocol, Src Port: http (80)
⊞ [16 Reassembled TCP Segments (22912 bytes): #144(1
⊟ Hypertext Transfer Protocol
  ⊞ HTTP/1.1 200 OK\r\n
    Date: Sun, 18 Aug 2013 16:36:46 GMT\r\n
    Server: Apache/2.2.20 (Ubuntu)\r\n
    Last-Modified: Sun, 02 Jun 2013 15:17:07 GMT\r\n
    ETag: "2586c-5859-4de2d576475c3"\r\n
    Accept-Ranges: bytes\r\n
  ⊞ Content-Length: 22617\r\n
    Keep-Alive: timeout=5, max=100\r\n
    Connection: Keep-Alive\r\n
```

Figure 7: Captured HTTP response containing video chunk #4

Figure 7 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.



```
⊞ Transmission Control Protocol, Src Port: http-alt
⊞ [16 Reassembled TCP Segments (22627 bytes): #883(1
⊟ WebSocket
    1... .... = Fin: True
    .000 .... = Reserved: 0x00
    .... 0010 = Opcode: Binary (2)
    0... .... = Mask: False
    .111 1110 = Payload length: 126 Extended Payload
    Extended Payload length (16 bits): 22623
  ⊞ Payload
```

Figure 8: 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 8. 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 metadata introduced per chunk is four bytes (zero bytes for the request since it 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 7 and 8 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 test bed (binaryjs [35]). Thus, the concrete WebSocket framework and libraries used for development need to be examined whether they add additional metadata to the payload, since this has an influence on the overall efficiency. In this particular case, the exchanged metadata 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 test bed 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 test bed have been deployed for the DASH-like and for the WebSocket-based live video streaming.

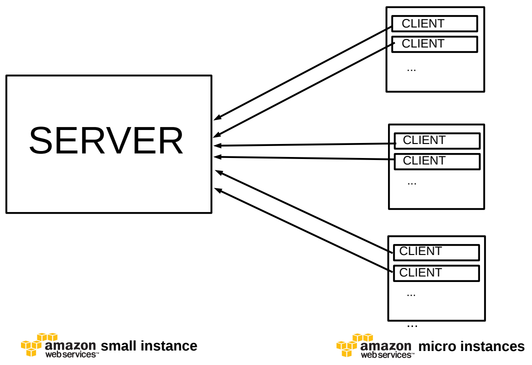

SERVER
CLIENT
CLIENT
...
CLIENT
CLIENT
...
CLIENT
CLIENT
...
...
amazon webservices small instance
amazon webservices micro instances

Figure 9: Architecture of the processing overhead test bed.

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 [36]. 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 architecture of this test bed can be observed on Figure 9. The developed components described in Section IV have been installed on these systems in order to setup and operate the test beds. When building such a large scale test bed, 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 bit rate is higher than the video bitrate, the user will be able to watch the video without encountering any disturbance. The moment in time when the number of clients is so big that the majority of them cannot 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 10, with a red dot symbolizing the defined inflexion point.

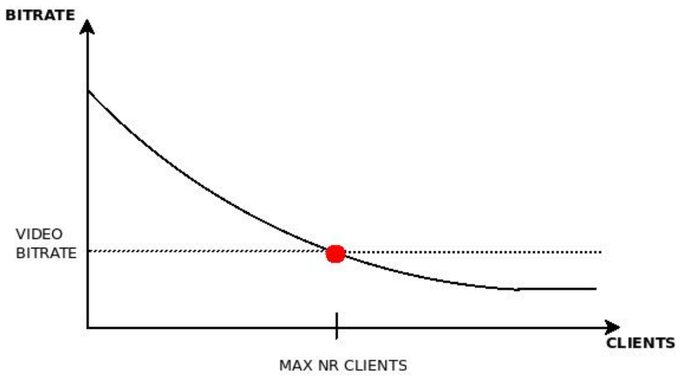


Figure 10: Expected curve of transmission bitrate

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 11 shows the results obtained from the DASH-like live streaming test bed. It can be observed that the graph for HTTP transmission bitrate shows a corresponding shape as theoretically expected and depicted in Figure 10.

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 12 summarizes the results obtained from the WebSocket-based live streaming test bed. 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.

The tests runs have been performed in both cases until 2,000 concurrent clients have been reached. Further measurements in the WebSocket-based test bed have not been performed. When extrapolating the obtained results, then the inflexion point will be located at around 2,300 clients (see Figure 12).

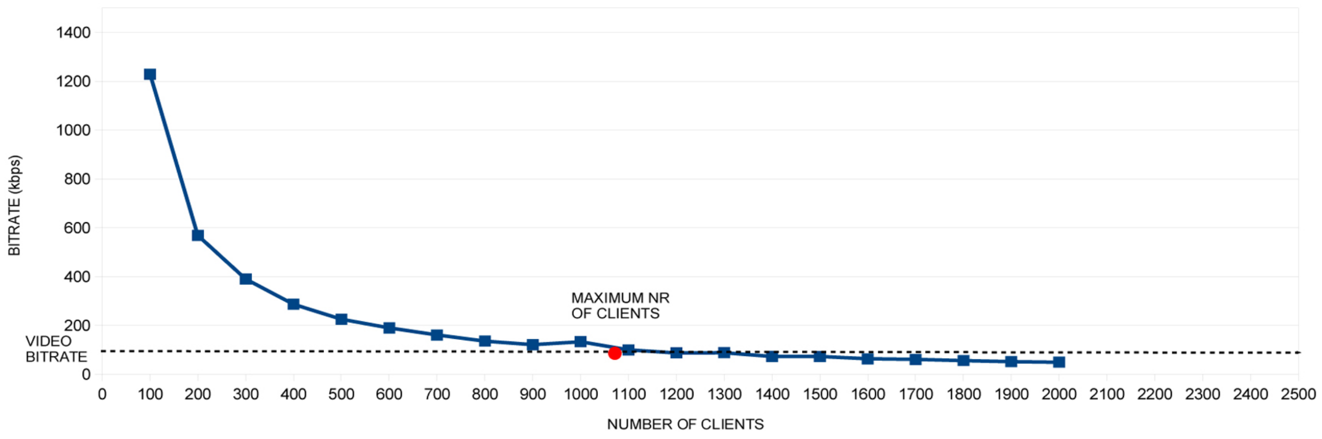


Figure 11: Average transmission bitrate for DASH-like streaming

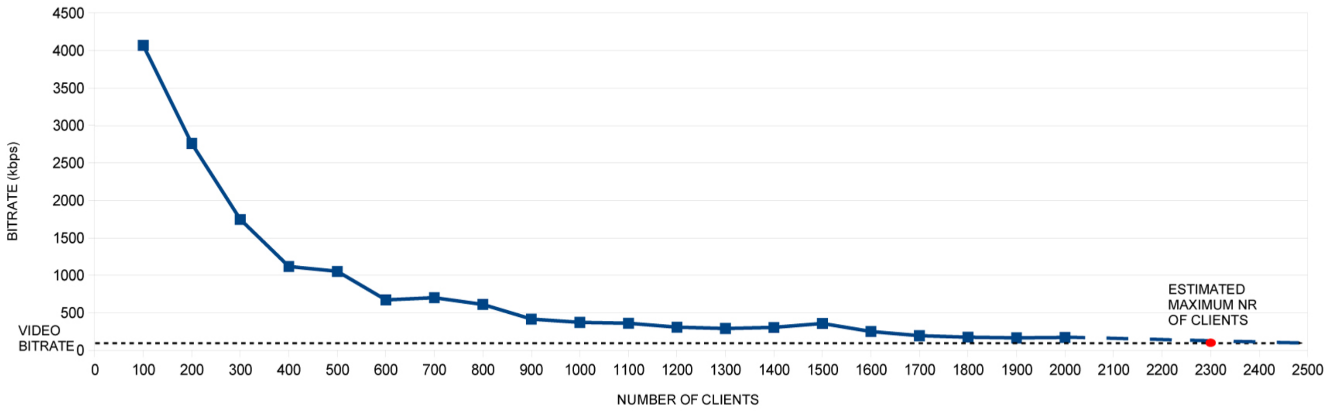


Figure 12: Average transmission bitrate for WebSocket-based streaming

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 resources.

To facilitate the task of evaluating the performance of this implementation, a video encoded at very low quality has been chosen. However, the obtained results are also to be applied for different video qualities.

VI. Further Considerations

Future research activities in respect to the proposed push-based web-native live streaming approach need to focus on further aspects. One is concerned with the adaptiveness to the underlying network conditions, in order to provide a better user experience in the presence of changing network properties. Another aspect to focus on in further research and development activities is the relation of Content Delivery Networks (CDN) and connection-oriented protocols, such as the WebSocket protocol.

A. Adaptiveness to the network conditions

Although adaptive bitrate streaming solutions are significantly more complex than constant bitrate streaming technologies, still, a method that determines the data throughput capabilities of each user in real time and controls the quality of the media stream accordingly, provides consumers with the best experience available, depending on their specific network and playback conditions. This can be achieved for the proposed architecture by including an encoder, which generates multiple distinct video encodings out of a a single source. The switch between the different video qualities occurs when necessary, attending to the network conditions. This can be achieved in two different ways. The first one is to monitor the video playback on the client-side and have the client notify the server when it should reduce or increase the quality. The second one monitors on the server-side and takes the decision whether to change to another video quality level directly there. Although the first approach is currently used by most of the adaptive video streaming systems, it introduces an additional overhead due to the exchange of control messages between the client and the server. Therefore, it would not be reasonable to adopt such an approach for this architecture, where the focus has been to reduce overhead coming from periodical control requests in the first place. The second option, instead, is more reasonable in the light of the proposed push-based approach. The status of the WebSocket connection can be leveraged to facilitate this. More concretely, the TCP channel underlying the WebSocket can be used to monitor the current network conditions in terms of the available data throughput rate and to determine the most appropriate video delivery rate.

A first implementation of this concept has been developed, with three video qualities:

- Low definition video (320x240, 100 kbps)
- Standard definition video (1280x720, 1,200 kbps)
- High definition video (1920x1080, 5,000 kbps)

It provides the adaptively by checking whether the fill level of the output buffer increases, remains static or is empty during a certain observation time. In the first case the server stops sending data to the client until the buffer is drained and then continues sending with a lower video quality, if available. The second case does not require any specific action, since the current streaming settings fit to the current network conditions. In the last case, however, it seems that the client might be able to consume a higher quality level, which will be delivered, if available.

To verify this concept, a test environment has been implemented. The adaptive streaming server has been based on the Vert.x [37] application platform and concretely its modules for WebSockets and Flow Control. To simulate different network states a delay is introduced on every packet sent using the tool Netem [38]. It is controlled by the command 'tc', part of the iproute2 package of tools. The command to add a fixed amount of delay of n ms to the outgoing packages is:

```
tc qdisc add dev eth0 root netem delay n ms
```

It has been observed that the server adapts to the current state and selects the source video accordingly. For a case where the client is connected through a high-bandwidth connection with no delay, the client receives the High Definition video on the browser. When there is a delay higher than n ms, the client receives the standard definition quality and for any delay higher than m ms the low definition video.

With this prototype implementation the server-side data throughput monitoring and control has been proofed as a low-overhead and seamless extension for the push-based live video streaming system proposed in this paper. However, future work should continue in this direction to provide a fully adaptive implementation, which does not only reduce the quality for a low-bandwidth or high-latency connection but also applies more precise algorithms taking into account all parameters taking part on the data throughput and delivery to select the higher quality available for each user.

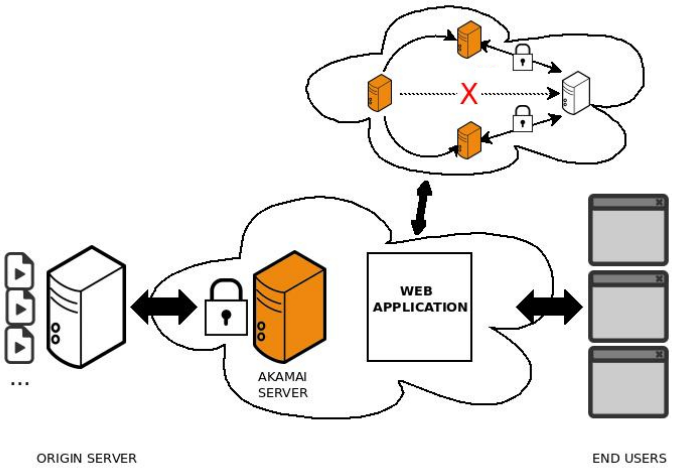


Figure 13: Akamai CDN architecture [39]

B. CDNs and push-based live video streaming

Currently, Content Distribution Networks (CDN) play a significant role when referring to web-based video streaming. The term refers to a large geographically distributed system of servers deployed in multiple data centres across the Internet, which has as a goal to serve content to end-users with high availability and high performance. This is accomplished by transparently mirroring content from customer servers, replicated all over the world. Thus, users receive the content from a server part of the CDN, which will be automatically picked depending on the kind of content and the user's location. The architecture of Akamai, one of the current top CDNs is depicted in Figure 13.

Content providers, and in particularly media companies, require these services for delivering their content to end-users. Referring back to the Cisco Visual Networking Index Global Forecast and Service Adoption for 2013 to 2018 mentioned in Section I, CDNs carried over 36% of the Internet traffic in 2013.

Therefore, the main issue they solve is the latency, the amount of time it takes the server to receive, process, and deliver a resource for a request by this mechanism, which leads to low download-times, enabling that a live event can be transmitted to every part of the world in real time as it is being consumed, as well as to decreasing the vulnerability to network congestion.

However, when proposing such a switch from HTTP to WebSocket protocol for live video streaming, the real-time nature of such content stream poses impediments concerning caching and load balancing systems, the main advantages of CDNs.

Web caching is used to store content for a certain amount of time. A situation where this is extremely useful is one where a file will be repeatedly request by a big amount of users. CDNs are equipped with a cache of static resources like static images, CSS files, JavaScripts, as well as bigger files like video and audio media to reduce latency and network traffic. Certainly, most objects in the cache do not stay there permanently but expire so that new content can be served. How long the resources stay in the cache can vary very much depending on the content from some minutes to years. This mechanism is however not applicable to live content whichthat needs to be consumed in real-time and for which this does not provide an advantage. On the other hand, load balancing systems distribute all requests over multiple servers in order to avoid that a single server becomes overwhelmed and provide the maximum availability. This target is also difficult to accomplish with this mechanism when the media content to be delivered is being produced in real-time and the protocol used for delivering is WebSockets instead of the usual HTTP.

Further research should focus in finding options to take advantage of the infrastructure of CDNs and investigate if some changes would need to be made to use them for such a live streaming implementation over WebSockets as the one presented on this paper.

VII. Conclusion and Future Work

Video content distribution in the Web is evolving greatly. The adoption of HTTP for video streaming in the Web has its pros and cons.

For the on-demand retrieval of file-based videos the comprehensive and pervasive HTTP guarantees a broad accessibility of the content. This approach also fits well with the current deployment and usages of CDNs, ensuring the necessary global scaling of such an approach. However, these advantages do not apply to live streaming of video content. First, CDNs cannot 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.

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

This paper examined the possibility 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.

First steps towards an adaptive live streaming architecture have been undertaken, proposing a mechanism situated on the server-side to introduce adaptiveness to the underlying network conditions to the proposed push-based web-native approach, which has to be further developed in future research activities, to enable a better user experience in the presence of changing network properties.

Another significant issue for future research relies on the relation of connection-oriented protocols, such as WebSocket and CDNs, in order to investigate options to capitalize on the infrastructure of CDNs for their use in a live streaming implementation over WebSockets as the proposed.

References

[1] L. Lo Iacono and S. Santano Guillén, “Web-native video live streaming,” in WEB 2014 - The Second International Conference on Building and Exploring Web Based Environments, ISSN: 2308-4421, ISBN: 978-1-61208-333-9, 2014, pp. 14–19.

[2] Cisco, “Cisco visual networking index: Forecast and methodology, 20132018,” White Paper, 2014, online available at www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf (last accessed: Nov. 2014).

[3] 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: Nov. 2014).

[4] 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: Nov. 2014).

[5] J. Follansbee, Get Streaming!: Quick Steps to Delivering Audio and Video Online. Elsevier, 2004.

[6] 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: Nov. 2014).

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

[8] Microsoft Corporation, “Silverlight 5.1,” 2013, www.microsoft.com/silverlight (last accessed: Nov. 2014).

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

[10] 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: Nov. 2014).

[11] 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: Nov. 2014).

[12] 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: Nov. 2014).

[13] 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: Nov. 2014).

[14] R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O’Connor, S. Pfeiffer, and I. Hickson, Tech. Rep.

[15] 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: Nov. 2014).

[16] D. Minoli, Linear and Non-Linear Video and TV Applications: Using IPv6 and IPv6 Multicast (7.1.2). John Wiley & Sons, 2012.

[17] Encoding.com, "The definitive guide to hls," July 2014, "http://www.encoding.com/http-live-streaming-hls/" (last accessed: Nov. 2014).

[18] Adobe, "Adobe http dynamic streaming," www.adobe.com/products/hds-dynamic-streaming.html (last accessed: Nov. 2014).

[19] Adobe, "Flash media manifest file format specification 1.01," 2010, online available at osmf.org/dev/osmf/specpdfs/FlashMediaManifestFileFormatSpecification.pdf base(last accessed: Nov. 2014).

[20] B. Rainer, S. Lederer, C. Müller, 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.

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

[22] A. van Kesteren, J. Aubourg, J. Song, and H. R. M. Steen, Working Draft, 2014, online available at http://www.w3.org/TR/XMLHttpRequest (last accessed: Nov. 2014).

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

[24] I. Hickson, "The web sockets api," W3C, Candidate Recommendation, 2012, online available at www.w3.org/TR/websockets (last accessed: Nov. 2014).

[25] A. Deveria, "Can i use web sockets," April 2005, "http://caniuse.com/websockets" (last accessed: Nov. 2014).

[26] D. T. Rico, "Seamlessloop," 2007, "github.com/Hivenfour/SeamlessLoop" (last accessed: Nov. 2014).

[27] T. Yoshino, "Pywebsocket," pypi.python.org/pypi/mod_pywebsocket (last accessed: Nov. 2014).

[28] H. Ohtani, "websocket-client," pypi.python.org/pypi/websocket-client/0.7.0 (last accessed: Nov. 2014).

[29] Python Software Foundation, "Http protocol client," docs.python.org/2/library/httplib.html (last accessed: Nov. 2014).

[30] Python Software Foundation, "Basehttpserver," docs.python.org/2/library/basehttpserver.html (last accessed: Nov. 2014).

[31] Python Software Foundation, "Socketserver," docs.python.org/2/library/socketserver.html (last accessed: Nov. 2014).

[32] Blender Foundation, "Big buck bunny," 2008, peach.blender.org (last accessed: Nov. 2014).

[33] Creative Commons, "Creative commons license attribution," 2007, "www.creativecommons.org/licenses/by/3.0/us/legalcode" (last accessed: Nov. 2014).

[34] Open Source, "Wireshark 1.10.2," 2013, www.wireshark.org (last accessed: Nov. 2014).

[35] E. Zhang, "Binaryjs," 2013, www.binaryjs.com (last accessed: Nov. 2014).

[36] Amazon Web Services, "Amazon elastic compute cloud (ec2)," www.aws.amazon.com/en/ec2 (last accessed: Nov. 2014).

[37] Eclipse Foundation, "Vert.x," 2013, "http://vertx.io/" (last accessed: Nov. 2014).

[38] Linux Foundation, "Netem," April 2005.

[39] E. Nygren, R. Sitaraman, and J. Sun, "The akamai network: A platform for high-performance internet applications," in ACM SIGOPS Operating Systems Review, Vol. 44, No.3, 2010, online available at http://www.akamai.com/dl/technical_publications/network_overview_osr.pdf (last accessed: Nov. 2014).