# Evaluating HTTP/2 for the efficient delivery of digital media


**Stylianou George**

**This dissertation is submitted for the degree of Master of Science in**

**Computer Science**

**August 2016**

**School of Computing and Communications**


George Stylianou

Evaluating HTTP/2 for the efficient delivery of digital media

MSc. Computer Science

26/08/2016

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted version of this submitted work, I consent to this being stored electronically and copied for assessment purposes, including the Department's use of plagiarism detection systems in order to check the integrity of assessed work. I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Date: 26/08/2016

Signed: George Stylianou

# Acknowledgements

I would like to express my gratitude to my supervisor Dr. Nicholas Race for his guidance that has been much valued and appreciated. I would also like to express my appreciation to the researches of the School of Computing and Communications Jamie Trench-Jellicoe and Matthew Broadbent for their time and support. Lastly, I would like to express my gratefulness to my parents, friends and family for their encouragement.



Word Template for Postgrad Dissertation

# Abstract

The Hypertext Transfer Protocol (HTTP) is without any doubt the most popular web protocol in the Internet world. It has been used for a wide variety of tasks and media streaming could not be an exception. Nowadays, HTTP is considered as the key protocol for video distribution over the Internet. Furthermore, adaptive bitrate streaming has been widely adopted and deployed as a flexible technology to deliver media content over the web. This is based on the fact that in an adaptive bitrate streaming scenario a media content is encoded into multiple representations and the client can select the most appropriate video quality in order to deal with network fluctuations. Recently, the new HTTP/2 protocol has been introduced and it provides a wide variety of new features aiming to solve the various issues of its predecessors. A very interesting feature of HTTP/2 and the main focus in this project, is the server push feature which allows the server to send resources to the client without requiring an explicit request for those resources. In this project, the impact of the HTTP/2 protocol and the server push feature on media delivery is investigated in detail. To achieve that, various server push strategies have been developed and evaluated. Moreover, the project aims to facilitate the way that pushed resources are configured in the Apache web server by developing a web interface to automate the overall process. To the best of our knowledge this is the first study which attempts an evaluation of the server push feature on the most popular web server available.

TABLE OF CONTENTS





Word Template for Freshmen @ Aalto-yliopisto 2014

# List of Tables





Word Template for Thesis at Mälardalen University

# List of Figures

Word Template for Freshmen & Master 2014

# 1 Introduction

The HTTP protocol has been the most dominant web protocol since its invention. It has been used for nearly every task on the Internet (Stenberg, 2014) and media streaming could not be an exception. In recent years, HTTP streaming has become one of the most popular approaches to deliver media content over the Internet (Sodagar, 2011). The main reasons behind its wide adoption are derived from the significant advantages of the HTTP protocol. More specifically, the scalability, ease of deployment, cost efficiency and wide support of HTTP are the most important reasons which led to the adoption of this streaming technique by the most popular commercial deployments. Following the development and adoption of HTTP streaming, the need for a flexible

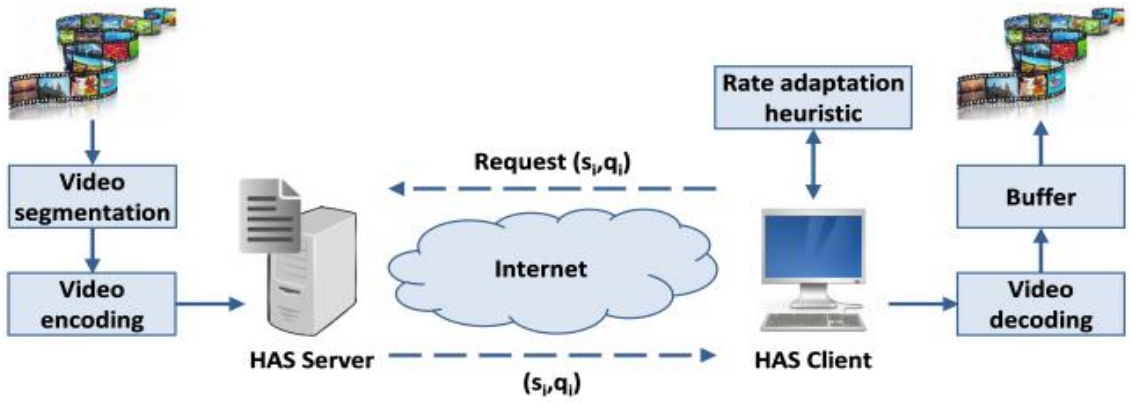


**Figure 1-1: A simple representation of the concept of HTTP adaptive streaming (Huysegems et al., 2015).**

video distribution had arisen. As a result, the HTTP Adaptive Streaming was introduced and its behaviour can be seen in Figure 1-1.

In an adaptive streaming scenario, a media content is encoded into different quality representations each own identified by its corresponding bit rate (Huysegems et al., 2015). Each representation is split into various segments of a predefined segment length. Information and details about the various representations and segments are stored in a manifest file which needs to be downloaded by a client each time a new session is initiated. Using the information provided in the manifest file and by monitoring the network conditions, the client decides and downloads the most appropriate quality level segment. The ability of adaptive streaming to adapt the video quality according to network fluctuations is the most important advantage of this technique. In this way, video freezes and poor video quality, which are common characteristics of progressive download, can be optimised and even eliminated. Despite the several advantages of HTTP adaptive streaming, the fact that HTTP protocol was not originally invented for media streaming introduces several issues which require to be addressed (Wei & Swaminathan, 2014). Some of these inefficiencies include:

- The segmentation mechanism of HTTP adaptive streaming requires the request of each individual segment using a HTTP request. Therefore, the total number of requests needed for the entire video playback is dependent on the video duration and the segment size. Moreover, the fact that the video and audio content is not multiplexed results in doubled number of requests compared to multiplexed media streams (Wei & Swaminathan, 2014).
- Low average quality and therefore low link utilisation is another major issue of adaptive streaming. Taking into consideration Conviva Releases 2014 Viewer Experience Report - Conviva (2014), more than 40% of the streaming sessions faced low resolution video delivery. This is equivalent to two poor video views every five streaming sessions.
- Various experiments showed that frequent video-quality changes can also have a negative impact on the Quality of Experience (Huysegems et al., 2015).

- The amount of waiting time for the viewer when interacting with a client can also be considered a major issue. The total delay should be kept as small as possible (Huysegems et al., 2015).

The main goal of this study is to perform an extensive evaluation of delivering media content over the newly developed HTTP/2 protocol. The new version of HTTP was published in May 2015 and aims to solve the numerous issues of its predecessors. (Belshe, Peon, & Thomson, 2015). One of the most interesting new features of HTTP/2 and the main focus of this study, is the server push feature. Server push gives the ability to the server to send resources to the client without requiring an explicit request for those resources (Belshe et al., 2015). The pushed assets are resources which the server anticipates that the client will need in the near future. This new feature of HTTP/2 was originally designed for web page browsing where resources, such as stylesheets or images, can be pushed to the client in order to reduce latencies. However, it is believed that the server push feature can also have a significant impact on adaptive streaming scenarios.

This study is a part of a long term project which aims to implement HTTP/2 into the Vision software, a TV and radio player available on Lancaster University campus (Mu, Trench-Jellicoe & Race, 2014). According to its authors, many social TV platforms have been created in recent years in order to facilitate the demand for sharing of television content and communication between viewers. Vision was created with the aim of becoming a comprehensive social TV platform for the faculty and students of Lancaster University, a platform where they can access high quality video and audio media from both web-based and mobile device application formats. Moreover, this study is limited into an extensive research about the feasibility of its long term project and does not require any changes to the Vision software itself. The project focuses on the server push feature of HTTP/2 and investigates its impact on adaptive streaming scenarios. To

achieve that, several server push strategies were implemented and evaluated in terms of various evaluation metrics. Despite the evaluation focus of the project, a web interface was also implemented in order to facilitate the configuration of pushed resources in the Apache web server.

To summarise, the key contributions of this study include:

- The development of various server push strategies which are evaluated to identify their performance characteristics regarding several evaluation metrics. These strategies aim to eliminate various issues of adaptive streaming.
- The evaluation was performed on the popular Apache web server. Most of the existing literature work evaluated HTTP and the server push feature in different web servers but none of them performed the evaluation on Apache. To the best of our knowledge, this is the first attempt leveraging and evaluating the server push feature of HTTP/2 on the most widely used web server.
- The implemented web interface aims to simplify and speed up the process of configuring pushed resources on the Apache web server. To the best of our knowledge, this is the first implementation which attempts to do that.

The remainder of the paper is organised as follows. Initially, a detailed literature review about the various concepts of the project and any related work is provided. Additionally, the paper describes the design of the experimental setup and the design of the web interface. Following that, the paper highlights the implementation of the various server push strategies and also gives details about the implemented software. Moreover, in the Experimental Analysis section all the experiments conducted along with their results are reported in great detail. Finally, the paper summarises the findings and present some possible future work.

# 2 Literature Review

In this section, the various concepts utilised in the project are explained in detail. Special focus is given on the HTTP protocol, its history, and the various versions introduced throughout the years. Also, the HTTP/2 protocol is examined in detail along with its numerous new features. Special focus is given to the server push feature, which is a key part of this project. Finally, the section highlights the advantages of HTTP streaming and explains the behaviour of the MPEG – DASH standard.

## 2.1 History of HTTP

As discussed in Grigorik (2013), the HTTP protocol is the most popular application protocol in the web and is defined as the common language that clients and servers share between them to enable the modern web. It was introduced by Tim Berners Lee to further support his idea about the World Wide Web and has been around for more than 20 years. Nowadays, the HTTP protocol is not only limited to web browsing but has also become the standard application protocol for every Internet-connected application. It operates by exchanging request and response messages between the servers and the clients. The majority of the HTTP communications take place over TCP/IP connections and are initiated by a user agent software (common example of a user agent is a web browser) which sends a request to an origin server about a specific resource (Network Working Group, 1999). In its simplest form, such a communication can be accomplished by a direct connection between the client and the server while more complicated connections involve the presence of one or more intermediaries. It is important to note that HTTP does not require to be

implemented on top of TCP/IP, it can also be implemented on top of any other protocol as long as a reliable transport is guaranteed.

Based on the above, a brief history about HTTP will be beneficial in order to examine the progress and the development of the protocol and also to understand how far this protocol has come. The very first version of HTTP was the extremely simple HTTP 0.9 which introduced to facilitate the idea of Tim Berners-Lee about the World Wide Web (Grigorik, 2013). Using HTTP 0.9, a request was comprised of a GET method and the path of the requested resource. Following the simplicity of the request, the response consisted of a single hypertext document without any metadata information. Moreover, after every request the connection between the two endpoints was terminated. It is important to note that the HTTP 0.9 protocol is still supported by some web servers due to its simplicity. However, the requirements of the web led to the exposure of the numerous limitations of HTTP 0.9 (Grigorik, 2013). Some of these limitations included the serving of just HTML documents and the absence of metadata information about requests and responses. Based on the aforementioned requirements, a huge number of experimental HTTP/1.0 implementations were deployed by various developers and in May 1996 a document was published by HTTP Working Group which documented the most popular of these implementations. It is important to note that HTTP/1.0 is not a formal Internet standard. Having said that, HTTP/1.0 introduced the option for the response to be of any type (HTML, plain text, image, etc.) and also proposed header fields in both requests and responses. Other interesting features of HTTP/1.0 included content encoding, authorisation and caching. Despite the fact that the informal HTTP/1.0 resolved a lot of the issues of HTTP 0.9, the goal of turning HTTP into an official Internet standard was still a work in progress.

The first HTTP/1.1 document was released in January 1997 and its further improved version, which included various improvements and updates, was released in June of 1999 (Grigorik, 2013).

HTTP/1.1 managed to address a lot of ambiguities of the protocol, while at the same time introduced several critical features to further optimise the overall performance of the protocol. The most significant difference of HTTP/1.1 was the introduction of keep alive connections, a feature which allows multiple requests to reuse an existing established TCP connection. As a result, the end-user experience was optimised significantly. It is important to note that HTTP/1.1 uses keep alive connections by default, meaning that every connection to a server is kept open unless told otherwise. Additionally, HTTP/1.1 introduced multiple negotiable capabilities which can be included on each request, such as caching directives, client cookies and language negotiation.

The main feature of HTTP which enabled its rapid growth and adoption is the overall simplicity of the protocol. It is currently used by billions of devices and has been evolving throughout the years into a protocol which can be used for almost any situation. However, the user requirements keep increasing and the performance demands are now higher than ever. Moreover, after more than a decade since the publication of HTTP/1.1, web browsers have evolved dramatically, the mobile Web has become an essential part of our daily life and the complexity of web applications has seen a huge growth. According to Stenberg (2014), a modern website is consisted of more than one hundred individual resources and this number is expected to increase further. Moreover, the total transfer size and the total number of average requests on the most popular web sites in the world have seen an enormous growth in the last four years. HTTP/1.1 is currently an outdated protocol and the way in which it operates has caused a significant degradation of the web performance. In particular, HTTP/1.1 aimed to improve the performance of HTTP/1.0 by introducing request pipelining, a technique where multiple requests can be sent at the same TCP connection without the need to receive a response for every previous request. However, request pipelining introduced head-of-line blocking, a performance-limiting phenomenon with huge latency delays for the client. The head-of-line-blocking occurs when a request takes a very long time to be processed on the server and due to this delay all the subsequent requests are blocked

and delayed. Therefore, HTTP/1.1 clients need to establish multiple TCP connections to the server in order to be able to send multiple requests to the server at the same time and minimise latency (Belshe et al., 2015). Another significant limitation of HTTP/1.1 is the repetitive HTTP header fields which result in excessive latencies and network traffic. HTTP/1.1 is described by Stenberg (2014) as a latency sensitive protocol which can no longer keep up with the rapid evolution and the near real-time requirements of the web. Therefore, the new version of HTTP protocol needed to be designed.

## 2.2 SPDY and HTTP/2

Back in 2009, the first attempt to address some of the most popular performance limitations of HTTP/1.1 was initiated by Google with the development of a new protocol called SPDY (Grigorik, 2013). SPDY was an experimental application-layer protocol designed specifically to minimize the loading times of web pages. Moreover, the SPDY project aimed at a 50% reduction of page loading time with no need for changes in the existing network infrastructures or in the contents of a website. It is important to note that SPDY uses TCP as the transport - layer protocol in order to minimize the deployment complexity. Furthermore, SPDY was meant to be designed in partnership with the open - source community and therefore the source code of the project was made available soon after its announcement. After some years of experimentation, SPDY proved that it was a working concept and that it could significantly improve the web performance (Grigorik, 2013).

Following the success of SPDY, the HTTP Working Group started the development on a new HTTP protocol, the HTTP/2. During the period of 2012 - 2015, SPDY was used as an evaluation platform for a lot of the experimental features of HTTP/2. After a lot of intermediate drafts, the official HTTP/2 standard was published in May 2015 (Belshe et al., 2015). The new protocol was aiming to enhance the efficiency of various aspects of the HTTP performance and improve the

user-perceived latency. Moreover, its main goal was to address the significant performance limitations caused by the head-of-line blocking problem in HTTP/1.1 and improve the use of TCP. This improvement of TCP could be achieved by eliminating the need to establish multiple TCP connections to the server and thus enabling parallelism and minimising latency (Grigorik, 2013). To deliver these goals, HTTP/2 needed to make no changes to the core concepts of HTTP and simultaneously continue to support all the main features of HTTP/1.1.

### 2.2.1 Request and Response Multiplexing

One of the main improvements introduced by HTTP/2 is the full request and response multiplexing feature which is closely related to the new binary framing layer used by this new protocol (Grigorik, 2013). More specifically, a HTTP/2 communication between a server and a client can be executed into a single TCP connection which can be used to send any number of streams. A stream is defined as a bidirectional flow of bytes which can be distinguished by a unique integer identifier and a priority value. Moreover, each stream communicates in messages which have the ability to split into multiple frames and then reassemble at the other end using the stream identifier which is included in each frame. At this point it is important to define what a message and a frame represent in HTTP/2. A frame is defined as the smallest unit of communication in HTTP/2, carries a specific type of data and belongs to a particular stream. Some of the most common types of frames used in HTTP/2 are:

- DATA frames: used to send message bodies.
- HEADERS frames: used to transport header fields of a stream.
- RST_STREAM: used to pronounce termination of a stream.
- SETTINGS: used to configure the communication details between two endpoints.
- PUSH_PROMISE: used to reserve a stream id in order to send a preload resource (server push feature).

Each frame needs to identify the stream which it belongs to by including the stream identifier into the frame header. On the other hand, a message is defined as a collection of frames that maps to an HTTP request or response. Taking into consideration Grigorik (2013), the ability of HTTP/2 to split a message into various frames, multiplex frames of different streams, send them into a single TCP connection, consider the predefined priority and finally reassemble them at the other end, is the most important enhancement introduced by HTTP/2. Multiplexing of two (or more) individual streams simply means that the frames of each stream are mixed together, sent over a single connection and split up again at the other end. An example of various interleaved frames from multiple streams, can be seen in Figure 2-1.

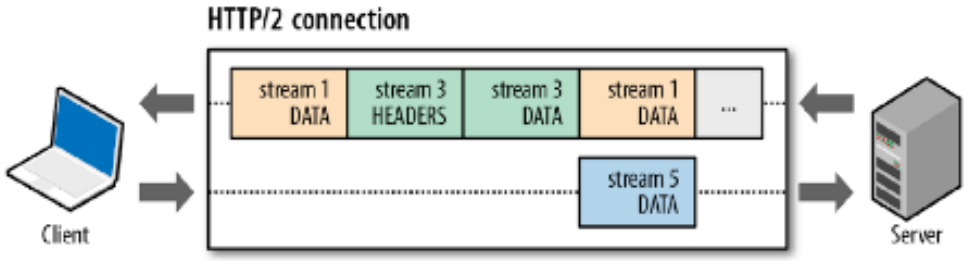


**Figure 2-1: An example of the request and response multiplexing feature of HTTP/2 (Grigorik, 2013, p. 218).**

Despite the fact that this feature was already implemented by other protocols and cannot be considered as an “original” HTTP/2 feature, it provides the means to solve a wide range of issues of HTTP/1.1. The most notable benefits of this approach are (Grigorik, 2013):

- A single TCP connection is being used for multiple requests and responses in parallel.
- Interleave requests and responses without blocking if a request takes time to be processed (the head-of-line blocking problem is eliminated).
- Reduces latency and improve page-load times.

### 2.2.2 Stream Prioritisation and Dependencies

The splitting of a stream into multiple frames allows a client to easily assign stream priority by defining prioritization information in the HEADERS frame of the stream (Belshe et al., 2015). By using stream prioritisation, the delivery of the various frames can be optimised and the overall performance of an application can be further improved. For example, when a web page is about to be displayed in a browser, the main HTML document which contains the structure of the page, the CSS documents which define the style-sheet rules, and the JavaScript required to alter the displayed contents are going to be served with the highest priority. Following these, the remaining resources, such as images, can be served with lower priority (Grigorik, 2013). The purpose of stream prioritisation is to allow the client to express to the server which streams need to be considered most important. This is really useful in cases where there is limited capacity for sending or when resource constraints force the server to send resources in a different order (Stenberg, 2014). Furthermore, the priority of a stream can be changed dynamically at run time using the PRIORITY frame. It is really important to note that specifying the priority of a stream doesn't mean that an endpoint can force a peer to send streams in a particular order. Therefore, we can define priority as a suggestion and not as a rule (Belshe et al., 2015).

HTTP/2 can also define stream prioritisation by defining dependencies. A stream can be dependent on another stream and therefore called dependent stream, while the stream upon which the stream is dependent is defined as the parent stream. By using dependencies one can express a suggestion to prioritise the parent stream rather than the dependent stream.

### 2.2.3 Flow Control

The ability of HTTP/2 for request and response multiplexing introduces contention on how the bandwidth resources of the TCP connection are going to be allocated between the various multiplexed streams (Belshe et al., 2015). This conflict results in blocked streams and therefore

undesired latencies. Despite the fact that stream prioritisation can provide the means for the server to determine the most appropriate way to send data, it is not an efficient way to control the allocation of bandwidth resources. To address the issue of blocked streams, HTTP/2 provides a flow control mechanism for both individual streams and the whole TCP connection which is used for the communication between two endpoints. As described in Belshe et al. (2015), the most important characteristics of the flow-control scheme are:

- Both flow control types (streams and connection) are considered in every single hop and not between end-to-end.
- The flow control scheme is based on WINDOW_UPDATE frames. Receivers advertise the amount of bytes (DATA frames) which are willing to accept for each individual stream and also for the entire connection.
- For every hop, the receiver of the connection has the overall control of the flow control scheme. The receiver can choose the window size it desires for every individual stream and for the entire connection. It is important to note that the sender of each connection needs to respect the window sizes set by the receiver.
- The flow control mechanism used by HTTP/2 applies only to DATA frames and therefore all the other frame types sent between a connection cannot be blocked by flow control and are not subject to the advertised window size.
- Flow control cannot be completely disabled. However, it can be set to the maximum available size (2^31-1) and can maintain this window size.

The main purpose of the flow control mechanism is to protect two endpoints which are operating under limited resources. Nevertheless, HTTP/2 only provides the essential tools for the implementation of that algorithm and different implementations can adjust the mechanism based on their specific needs.

### 2.2.4 Header Compression

One of the most important limitations of HTTP/1.1 is the large amount of repetitive header fields containing redundant data (Belshe et al., 2015). More precisely, each HTTP request carries a set of headers which includes metadata information used to describe the transferred resource. In HTTP/1.x versions of the protocol, this information was sent as plaintext and typically added more than 500 bytes of additional overhead in each request (Grigorik, 2013). For example, when a web page is loading and a client sends a large series of requests to the server, it often ends up with a series of almost identical responses (Stenberg, 2014). Undoubtedly, this can have a negative impact on the overall performance and introduce significant latencies. To address the issue of redundant header information and minimize the overhead caused by these, HTTP/2 compresses header frames. To achieve that, HTTP/2 creates header tables on each endpoint and keeps these for the entire HTTP/2 connection. As a result, both ends are able to store and reuse any header key-value pairs sent previously. New header key-value pairs can be appended to the table and existing pairs can be replaced by new values. In the ideal case where headers remain the same between requests, all headers' metadata is retrieved from the first occurrence of that particular request. The great advantage of the header compression feature of HTTP/2 is the significant impact on request sizes by eliminating any redundant and repetitive information (Belshe et al., 2015).

### 2.2.5 Server Push

Having discussed all the different features of HTTP/2, it is important to examine in depth a feature proposed by the new HTTP protocol, which is the main focus of this project. Typically, a modern web application constitutes of a large amount of resources which are discovered by the client after inspecting the document delivered by the server. Once the requests have been discovered, the client requests each individual resource by sending multiple HTTP requests to the server. Apparently, the procedure of sending a distinct individual request for each resource takes precious time to be

completed and also introduces significant latencies. These latencies could be easily eliminated by allowing the server to push these resources to the client ahead of time, without waiting explicit requests for specific resources. For example, when a web page is requested from a server, the client initially sends a request for the main file. The server already knows that additional requests will be shortly received for the stylesheet of the page and for any images that the page may contain. The server can decide to push the resources which knows that the client will definitely require, just after the reception of the first request, as shown in Figure 2-2. That's server push (Grigorik, 2013). This mechanism can be really useful in cases where the server knows that the client needs some additional resources in order to fully process a previous initiated request. Despite the fact that server push is a new feature firstly presented in HTTP/2 protocol, many of the modern web applications are already using it. This is achieved using inlining. Inlining is a technique used in HTTP/1.1 to minimize latencies and to avoid sending any unnecessary requests for additional resources. By inlining an asset using a data Uniform Resource Identifier (URI), the server is pushing that asset without waiting for the client to request it. Although the behaviours of the two mechanisms are identical, the major difference of server push and inlining is the fact that server push is an embedded feature of HTTP/2 while inlining needs to be defined by the application. This offers some significant advantages such as cacheable resources, decline of a pushed resource by the client, reusability and prioritisation (Grigorik, 2013).

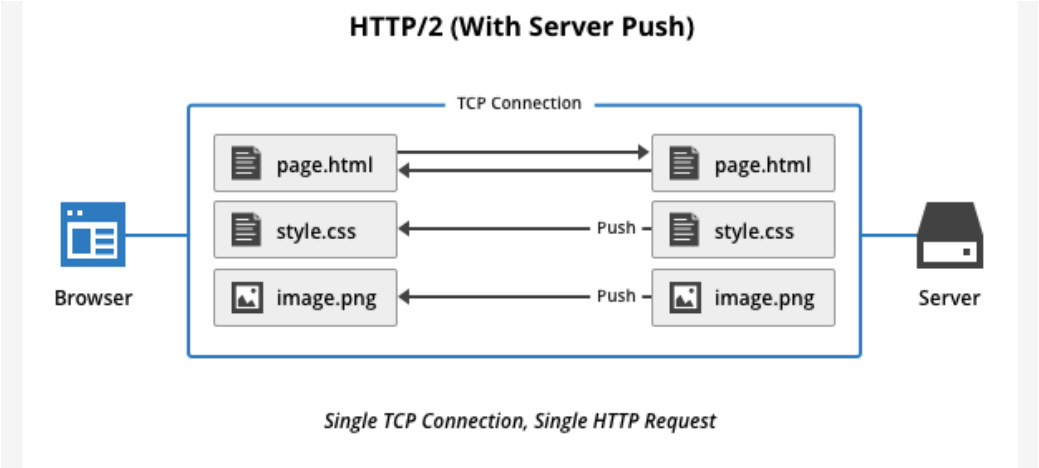


**Figure 2-2: An HTTP/2 connection with Server Push feature (Announcing Support for HTTP/2 Server Push, 2016).**

To push a resource to the client, the server needs to inform the client about the resource that is intended to be pushed. This is done using a PUSH_PROMISE frame (Belshe et al., 2015). PUSH_PROMISE frames can also be defined as a server request to the client in order to accept a promise for a resource which the server is intended to push. PUSH_PROMISE frames contain a header block with a complete set of request header fields and always need to be associated with a client-initiated request. The most important information in the header of a PUSH_PROMISE frame is the *Promised Stream ID* field which is used to identify the stream that is reserved by that frame. It is important to note that all the PUSH_PROMISE frames sent by a server are sent on the explicit stream of the initial request. Furthermore, in order to ensure that the client would not send any requests for any pushed resources, it is important for the server to send the PUSH_PROMISE frames prior to sending the actual push content. In that way, the client is able to see that a resource is intended to be pushed before discovering any embedded links and issue additional requests (Belshe et al., 2015). By sending a PUSH_PROMISE frame, a new stream is created and placed into the reserved state for both endpoints. Upon receiving a PUSH_PROMISE frame, the client can decide whether it wants to accept or cancel the request of the server. In the case where the client decides to accept the request, the server begins the sending of the pushed response on the stream which has been reserved for that particular asset. Moreover, the client also has the option to specify a SETTINGS_MAX_CONCURRENT_STREAMS value in order to limit the number of resources which can be pushed by a server at the same time (Belshe et al., 2015). However, the client may also decide to refuse a pushed resource by sending an RST_STREAM frame as a response to the PUSH_PROMISE frame received. In the RST_STREAM frame, the pushed stream identifier needs to be referenced.

### 2.2.6 Related Work

Prior to the development and adoption of HTTP/2, one of the first evaluations comparing SPDY and HTTP performance in terms of page loading times was carried out by Padhye and NIelsen

(2009). The main goal of the study was to understand how the different features of the two protocols affect performance using various websites. The tested websites included a simple web page consisting of several images and stylesheets files and also popular websites copied using the WinHTTrack software (HTTrack Website Copier - Free Software Offline Browser (GNU GPL), 2016). Moreover, the main performance metric in the evaluation was the page loading time obtained by Chromedriver, an open source standalone server which can be used for automated testing of web applications (ChromeDriver - WebDriver for Chrome, 2016). Using the simple website and various round trip times (RTTs), results showed that SPDY outperformed HTTP/1.1 and that in some cases SPDY managed to reduce loading times by up to 39%. Furthermore, the authors showed that for lower bandwidths, SPDY's performance improvement is significantly smaller. Using real websites fetched with the WinHTTrack software, SPDY fared better than HTTP mostly because of the fact that SPDY only establishes a single connection compared to the multiple connections of HTTP. Moreover, another important factor regarding the difference in performance between the two protocols is the fact that SPDY transfers less data than HTTP as a result of the header compression feature. Finally, the paper evaluated the impact of SSL on both protocols and it was concluded that using SSL, the page loading times are increased by a factor of 20%.

Following the development of HTTP/2, several studies aimed to evaluate the performance of HTTP/1.1 compared to HTTP/2. One of them was conducted in order to rate the performance of HTTPS, SPDY and HTTP/2 protocols in terms of size of request and response headers, number of TCP connections required during the loading of a web page and overall page loading times (A Simple Performance Comparison of HTTPS, SPDY and HTTP/2 | HttpWatch BlogHttpWatch Blog, 2016). Regarding header sizes, SPDY and HTTP/2 outperformed HTTPS due to the header compression feature. However, based on the HPACK algorithm of HTTP/2 which was designed specifically to compress headers, the headers' sizes of HTTP/2 are almost three times smaller than

SDPY. Furthermore, in terms of number of TCP connections, SPDY and HTTP/2 both outperformed HTTPS and their results were almost identical. This is due to the multiplexing support of these two protocols which reduces the total number of TCP connections established to the server. Finally, HTTP/2 did a greater job compared to HTTPS and SPDY in terms of page loading time. It is important to note that page loading time is defined as the time needed for the page to be downloaded and presented to the user. According to the authors, header compression and the small number of TCP connections were the main factors of the success of HTTP/2. This study also concluded that the advantages of HTTP/2 should be even more visible in more complex pages.

Despite the fact that previous work showed that HTTP/2 and SPDY have a great impact in the overall performance of a web page, other studies showed that this might not be the case. A study conducted by Kim et al. (2015), tested HTTP/2 on popular websites and concluded that it does not provide any improvements in overall performance. The study attempted to simulate a multi domain environment by building a web server with virtual host configurations. The test websites were cloned and evaluated - both their PC and mobile versions- on varying network conditions. The main performance metric of the study was the loading time of the main page. Initially, the authors evaluated the impact of HTTP/2 in a high speed mobile network environment (LTE network) and it was concluded that the use of HTTP/2 led to an increase of the page loading time by an average of 28%. Then, HTTP/2 was evaluated in a low-speed environment (3G network) and the results demonstrated a performance degradation of 8% on average. Therefore, the study inferred that HTTP/2 performs better for sites with fewer domains under low-speed network environments. An explanation for that might be based on the small number of domains which can reduce the overall time needed to establish the multiple connections. Last but not least, the paper examined the impact of HTTP/2 on high-speed wired networks using the PC versions of the web sites. In that case, HTTP/2 showed similar performance behaviour to the high speed mobile environment. It was

concluded that HTTP/2 degrades page loading time severely in high-speed environments. However, it is important to note that the authors highlighted various limitations of the study and argued that it cannot be concluded that HTTP/2 has a negative impact on page loading time in general.

## 2.3 Media Streaming and MPEG – DASH

Recent evidence suggests that IP video traffic will reach 82% of all global Internet traffic by 2020, while in 2015 the video traffic was no more than 70% (Index, 2016). The same study also demonstrated that smartphone traffic will surpass PC traffic by 2020 since smartphones will hold 30% of the total Internet traffic in comparison to the 29% of PC traffic. Moreover, internet traffic of wireless and mobile devices is estimated to account for more than 65% of IP traffic. As the majority of the Internet traffic is shifting towards mobile devices and video content, the expectations of the mobile users in terms of overall video performance will be further increased.

However, the IP traffic results suggest that network congestion will grow and this could definitely have a negative impact on the quality of service (QoS) for end users (Monchamp, 2013). Nowadays, several techniques have been developed to improve the overall performance of delivering video content over the Internet. The one which has the most significant market adoption is the HTTP streaming.

### 2.3.1 HTTP Streaming

The first attempts to deliver video content over the Internet were reported in the 1990s. To address the challenges of timely delivery and consumption of large amounts of data, the Real-Time Transport Protocol (RTP) was designed (Sodagar, 2011). RTP's main function was to define packet formats for both audio and video content and also manage the different sessions. However, the adoption of content delivery networks (CDN) in today's Internet world exposed different issues

regarding RTP, which works really well only in managed networks. The most important limitation of RTP is its resource intensive nature in large-scale networks since it requires the creation of a separate streaming session for each client. Moreover, RTP streaming is not supported by a wide range of CDNs and RTP packets are often blocked in firewalls (Sodagar, 2011). The tremendous growth of the World Wide Web has led to the efficient delivery of media content using HTTP protocol. The adoption of HTTP as the delivery protocol for streaming services has various benefits and some of these are:

- HTTP is widely adopted and supported by the Internet infrastructure and therefore NAT and firewall traversal issues are avoided. Furthermore, CDNs support local caches and therefore the performance of long distance traffic can be further improved (Sodagar, 2011).
- Using HTTP, the streaming session is exclusively managed by the client and the server doesn't need to maintain a session state for each client. Therefore, additional cost on server resources is eliminated (Sodagar, 2011).
- The deployment of HTTP media streaming is relatively simple due to the wide adoption of both HTTP and its underlying TCP protocol (Stockhammer, 2011).

For all of the above reasons, HTTP streaming has been widely adopted and deployed. However, the need for a flexible video distribution specification which would utilize HTTP as the underlying delivery method had arisen. As a result, adaptive bitrate streaming over HTTP was introduced.

### 2.3.2 Adaptive Bitrate Streaming

Prior to the development of adaptive bitrate streaming, CDNs used to host media content and stream them at a constant bit rate, without taking into consideration the available bandwidth of the client and the network congestion (Monchamp, 2013). In the case where the bandwidth of the client was not sufficient to maintain a continuous playback of a streamed video, the user was forced to wait until enough chunks of the video were downloaded in order to resume the playback from

where it left off. This could be a really frustrating experience for users, especially during the peak hours of the day. The adaptive bitrate streaming technique was adopted in order to eliminate freezes of a video playback during streaming. Nowadays, it has become the accepted standard for delivering media content online in a wide range of devices (Adaptive Bitrate Streaming, 2016). Adaptive bitrate streaming can be established by using a simple HTTP server and a browser or a media player client. It can be described as a combination of both endpoints' software, since it detects the available bandwidth of the client and adjusts the quality of video accordingly. Media content can be available in different bit rates and dimensions, and all of them are managed by a manifest file which contains metadata information about the different segments. As a result, media content which is delivered online using adaptive bitrate streaming can be adjusted in order for its quality to be as good or as bad as the available bandwidth of the client (Adaptive Bitrate Streaming, 2016). Furthermore, the fact that adaptive bitrate streaming is delivered over HTTP makes it both an efficient and a cost effective solution. Various protocols were developed for this type of adaptive behaviour and the most widely used were the HTTP Live Streaming, the Microsoft Smooth Streaming and the HTTP Dynamic Streaming. However, each protocol supported different methods and formats and therefore a client needed to deploy all of the available protocols to receive any media content from any server (Dynamic Adaptive Streaming over HTTP (MPEG-DASH), 2016). The lack of a widely adopted standard was too important to be ignored. An international adaptive bitrate streaming standard would allow a client to stream any media content, without the need to deploy multiple protocols. Having considered the need for an HTTP streaming standard utilising the flexible behaviour of adaptive bitrate streaming, Moving Picture Experts Group (MPEG) proposed the development of a widely acceptable standard in April 2009. In the next two years, MPEG in collaboration with various organizations and companies developed the MPEG-DASH standard (Dynamic Adaptive Streaming over HTTP (MPEG-DASH), 2016).

### 2.3.3 MPEG – DASH

MPEG-DASH is an open source technology which has been defined as the international standard for HTTP streaming since 2011. Its behaviour is similar with the early adaptive streaming protocols in that information about different media qualities and various representations is stored in a manifest file, called Media Presentation Description (MPD). The client still needs to request the manifest file in order to get all the different properties of the stream (Monchamp, 2013). But how does an interaction between an HTTP server and a DASH client work? A simple example of an interaction between an HTTP server and a DASH client can be seen in Figure 2-3. Taking into consideration Sodagar (2011), the media content is stored in the HTTP server in two different formats:

- The actual segments of the media content are stored in various bit rates and resolutions. It is important to note that segments are represented in the form of chunks and can be placed either in a single file or each individual segment can be placed in a separate file.
- A manifest file (MPD) which contains metadata information about the media content.

The information contained in the MPD file is required by the DASH client in order to be able to play the media content. Therefore, the manifest file is the first thing that the client needs to request from the HTTP server. The client can get the manifest file from the server using various methods but the most common one is using HTTP (Sodagar, 2011). The MPD file provides the client with information regarding media bit rates, resolutions, URL addresses of segments, minimum and maximum bandwidth, alternatives for the media components, accessibility restrictions, Digital Rights Management (DRM) information and other characteristics (Sodagar, 2011). Having obtained all the above metadata information, the client needs to decide the most appropriate media bit rate and quality resolution based on its network available bandwidth and request the first segment of the content. The subsequent segments will also be requested while monitoring

variations in the available bandwidth. During that procedure, the client may decide to fetch segments of different bit rates, lower or higher (Sodagar, 2011).

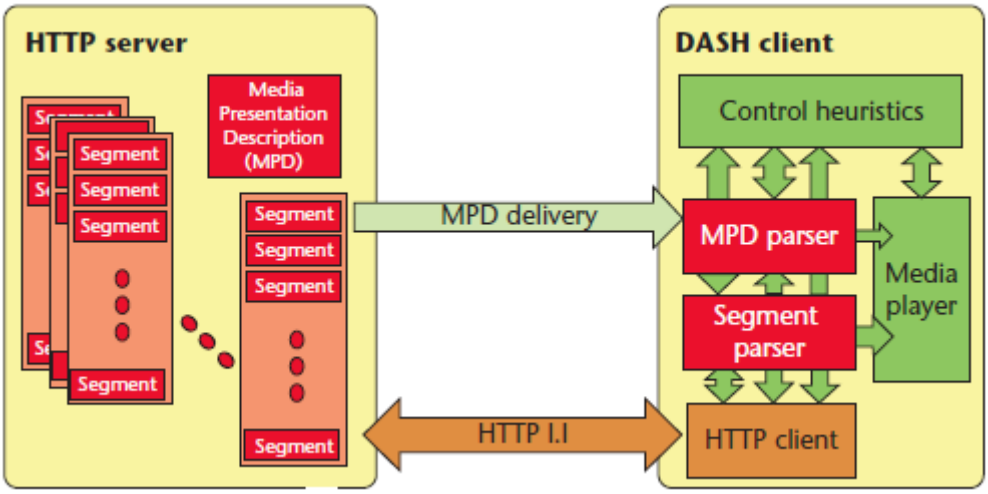


**Figure 2-3: The scope of MPEG - DASH standard (Sodagar, 2011, p. 64).**

Based on the above, it can be easily understood that a key component of the overall functionality of MPEG-DASH is the manifest file. The MPD file of a media content is an Extensive Mark-up Language (XML) file containing multiple program intervals, defined as periods (Sodagar, 2011). Periods are comprised of a starting point and a duration. Each period contains information about the different media components available (audio, video and text) and their alternatives, defined inside adaptation sets (Sodagar, 2011). For instance, one adaptation set might contain information about the different video bit rates while another one might contain information about the different bit rates of an audio component. A version of a media content encoded with a particular bit rate is called a representation and each representation is comprised of various segments. Segments are stored on the server and can be downloaded using HTTP GET requests by specifying a segment's URI.

The MPEG-DASH standard eliminated a lot of the issues of adaptive bitrate streaming and definitely provided numerous benefits (Sodagar, 2011). The fact that several companies were involved in its development, eliminated any technical issues and reduced transcoding costs.

Furthermore, the combination and support of several protocols and standards has achieved seamless streaming on different machines. A single media content can be encoded to be compatible on a range of devices and systems, such as mobile, desktop and HTML5. As a result, MPEG-DASH ensures that compatibility is not an issue and also ensures that all media content can be played on every device.

## 2.4 Related Work

In recent years, there has been an increasing amount of literature work examining the performance of DASH over HTTP/2. One of the first studies which attempted to do that was conducted by Mueller, Lederer, Timmerer and Hellwagner (2013). In that study, the performances of HTTP/1.0, HTTP/1.1 and SPDY were evaluated under various restrictions in terms of overhead, bandwidth utilisation and streaming performance. The experimental setup of the study, apart from the client and the server, also included bandwidth shaping and network emulation nodes, in order to restrict bandwidth and configure various RTTs. For the overhead evaluation of the three protocols, the quality level was fixed for each experiment and therefore the adaptive behaviour of DASH did not affect the experiment's results. Based on the results, HTTP/1.1 was proved to be the most efficient protocol in terms of overhead for the effective media bit rate. The poor performance of SPDY was mostly based on its framing layer. SPDY surpassed HTTP/1.1 only in the smallest quality level (100 kbps) because of the header compression feature and the small payload. SPDY with SSL was the least efficient protocol and its overhead was almost always higher than that of HTTP/1.0. Following the overhead evaluation, the authors examined the link utilisation of the different protocols under varying bandwidths and RTTs. It is important to note that link utilisation is defined as the ratio of the media throughput to the available bandwidth. In the case of HTTP/1.0, link utilisation was very low especially with high RTTs and high bandwidths. This is related to TCP's slow start and to the fact that HTTP 1.0 opens a different TCP connection for each request. When the HTTP/1.1 and SPDY protocols were evaluated, the difference between very low and very high

RTTs was much smaller than HTTP/1.0. This was mostly based on the keep alive connections and pipelining features of HTTP/1.1, and the stream multiplexing feature of SPDY which led to the establishment of a single TCP connection for the entire connection. Overall, authors concluded that the performance of HTTP/1.1 and SPDY was very similar. However, the link utilisation of SPDY with SSL was smaller compared to that of HTTP/1.1 and SPDY without SSL. Finally, a test scenario showed that streaming performance of HTTP/1.1, SPDY and SPDY with SSL was equally satisfying and stable under varying network conditions. The main conclusion of the paper was that SPDY, and therefore HTTP/2, could definitely improve streaming performance compared to HTTP/1.0 and could also eliminate the most important problem of HTTP/1.1, the head-of-line blocking problem.

Despite the positive results of HTTP/2 on adaptive media streaming as presented by Mueller et al. (2013), the findings reported by an internal study conducted at a later time by BBC (Performance Testing Results of Adaptive Media Streaming over HTTP/2 - BBC R&D, 2015) were not that optimistic. The main goal of the study was to examine the performance of HTTP/2 for media streaming applications, particularly concerning server-side performance, and to compare that with HTTP/1.1. The performance of the server was tested under high load conditions and both clear text and encrypted versions of each protocol were examined. In addition, the main performance metric used for the comparison was the time interval between receiving the first byte of the request, processing the request, sending the response and then writing the log line. All experiments were based on MPEG-DASH media segments. Following the evaluation, authors deduced that encrypted versions of both protocols perform worse than their clear text version and that HTTP/1.1 performs better than HTTP/2 for both versions.

Several recent studies investigating the performance of HTTP/2 for adaptive media streaming have been carried out and a really interesting one was conducted by Timmerer and Bertoni (2016).

Assuming that MPEG-DASH does not require a specific transport protocol and supports flexible deployment, the study attempted to evaluate HTTP/2 and Google's Quick UDP Internet Connections (QUIC) protocol for MPEG-DASH streams. Moreover, the evaluation was based on protocol overhead, link utilisation and adaptation behaviour of HTTP/1.1 and HTTP/2 over both TCP and QUIC. Results reported that protocol overhead was relatively low for all combinations, below 10%, with a slightly higher overhead for the QUIC protocol. Following protocol overhead, the link utilisation was evaluated for different protocol combinations under varying network conditions of different RTTs. In general, link utilisation was over 85% for all cases. However, the use of QUIC protocol provided a more unstable link utilisation compared to TCP/SSL for higher RTTs. Furthermore, the results revealed that adaptation performance was very similar for all combinations without any significant difference. The authors confirmed the results of Mueller et al. (2013) and concluded that QUIC does not have a significant contribution to the overall streaming performance.

Having discussed the potential benefits of the server push feature of HTTP/2, it was essential to explore any practical work which attempted to evaluate the performance of server push in media streaming scenarios. The first systematic study which aimed to leverage the server push feature for low latency live video streaming scenarios was reported by Wei and Swaminathan (2014). The motivation of the study was the request explosion problem that occurs in live video streaming when the segments' duration is set to be very small (1 second) in order to achieve low latencies. The study considered the All – Push strategy, where all the video and audio segments of the media content are pushed by the server when the first request is received and also the K-Push strategy where the client issues a request for every K segment. The experiments were performed using the Jetty web server (Jetty - Servlet Engine and Http Server, 2016) which supports the HTTP/2 protocol and the server push feature. Furthermore, the main evaluation metric of the study was the time difference between the actual event and the playback of a video frame, defined as the live

latency. The authors also considered the total number of requests required for the playback of the live event. The evaluation of the All – Push mechanism revealed that the huge savings observed in the number of requests come with a significant overhead in the adaptive bitrate switching. This occurs because the client sends only one request for the entire live event and therefore a change in the bit rate can occur only if the client cancels the push stream and issues another request for the new bit rate. As a result, it was concluded that the All Push strategy is not ideal for the adaptive streaming of a live event. Using the K - Push approach, the live latency was reduced by almost 2 segment durations compared to the no-push case, while the total number of requests was identical. Moreover, the client was able to issue a new request every 5 seconds and therefore the limited switching ability of the All – Push strategy was eliminated.

Following their work on live video streaming, Wei and Swaminathan (2014) performed a similar series of experiments to reduce the request overhead in on-demand video streaming. Apart from the optimisation of the request overhead, the authors aimed to propose a mechanism which would not violate the adaptive media bitrate switching and also minimise the overhead introduced by server push. The push based approaches of the study were sending audio and video segments which they were stored temporarily on a local client cache. The client was forced to first check its local cache before requesting any segments from the server. In case that the required segment has been already pushed by the server, the client retrieves and uses the segment without sending a request. The server push strategies of the study included the no-push approach which was considered a baseline for comparison, the audio only approach where the server pushes the accompanying audio segment of the video segment request that it receives and finally the K-push approach where K-1 pairs of audio and video segments are being pushed. Regarding the request overhead, the findings reported that the audio push strategy is able to reduce the number of requests by around 50% while the requests saved by the K-push strategy are increased as long as the k parameter keeps increasing. Having mentioned the savings in terms of number of requests, authors further evaluated the server

push strategies in terms of overhead in bit rate switching using switching delay and unclaimed pushes. The audio push strategy increased switching delay by a very small margin, while the switching delay in the k-push strategy was significantly higher, due to the consumption of the bandwidth by the pushed segments. Using the K-push strategy, a delay of around 2 segment durations was observed, independent of the K value. Considering all of the above, the study inferred that the audio push strategy is the best approach for low overhead adaptive streaming and at the same time provides significant improvements in the number of requests.

Recently, a number of authors have considered the effects of the server push feature on adaptive bitrate streaming (Huysegems et al., 2015; Petrangeli et al., 2016; Nguyen, Le, Nam, Pham & Thang, 2016). The study conducted by Huysegems et al. (2015), presented multiple methods to improve the quality of experience of HTTP adaptive streaming and also designed and evaluated a server push strategy which utilised segments in the order of milliseconds. Having performed several experiments, the authors initially showed that super-short segments can have a positive impact on the total time required to send and start playing a segment on the client but it can also degrade the average video quality severely. However, the use of server push combined with super-short segments can eliminate the issue of pool average quality. Furthermore, the time required to play a segment on the client and the start-up delay can be reduced significantly. In another study, Petrangeli et al. (2016) performed a similar set of experiments also focusing on the server push feature of HTTP/2 and HTTP adaptive streaming. Their findings confirm that server push can reduce live latency and start-up delay in combination with super-short segments when high RTTs exist. They also proposed an OpenFlow controller for network-based prioritisation which can effectively improve video freezes observed during a video streaming as a result of poor network performance. The study carried out by Nguyen et al. (2016) leveraged the server push feature of HTTP/2 in order to propose an adaptation method for adaptive streaming. More notably, the proposed approach takes into consideration the network and buffer conditions and smartly changes

the number of pushed segments in different requests. The results of the study showed that adaptive pushed segments can improve the balance between the number of requests and buffer stability, compared to the use of a fixed number of pushed segments across every request. Moreover, further research conducted by Wei, Swaminathan and Xiao (2015), proposed a video streaming mechanism which aimed to improve battery power savings on mobiles devices over cellular networks. Using a server push strategy and an analytical model, the authors were able to quantify power consumption and their proposed mechanism achieved battery power savings of up to 17%.

# 3 Design

Having considered and described the relevant literature for this study, this section focuses on the design of the project. The experimental setup will be explained in detail focusing on the media content and the client and server side implementations. Moreover, the different components and tools used will be described. The section also includes details about the implemented software and the script used to automate the data collection procedure.

This study is a part of a long term project which aims to implement and utilise the HTTP/2 protocol into the Vision software, a free TV & radio player available on Lancaster University Campus. However, the scope of the study is limited into an extensive research about the feasibility of the long term project and the contribution to the current state of the art regarding the Apache web server and the new HTTP/2 protocol. To achieve that, the project was divided into two parts: the evaluation of the performance of HTTP/2 on media streaming and a software implementation. In order to evaluate the performance of HTTP/2 and examine the impact of the server push feature on media content delivery, an HTTP adaptive streaming scheme was developed. The overall architecture is comprised of a client and a server component. The client component is running on a Windows machine utilising the Chrome Canary browser while the server component is running on a Linux machine. To perform the evaluation, a media content was stored on the server and the streaming was achieved using the MPEG-DASH standard, on the Chrome Canary browser. The server is based on the widely adopted Apache web server. The media was encoded into various bit rates and quality representations, with two different segment durations. Furthermore, a script was implemented to automate the collection of the results and speed up the overall process. The script

was used to store the results on the server and therefore the results could be collected and analyse way easier. The implementation part of the project focused on the development of a web interface which aims to address the configuration complexity of the server push feature on the Apache web server. Based on the fact that the configuration of pushed resources on Apache requires full access to the server, the developed web interface aims to automate and facilitate the overall procedure.

## 3.1 Experimental Setup

The media content used for the evaluation was an advertisement of the Google Chrome browser with a total duration of 91 seconds. The video was encoded in 3 different bit rates with different resolutions and segment durations. The selected bit rates were 400, 800 and 1500 kbps and the selected resolutions were 360p, 540p and 720p respectively. The audio of the media content was encoded at 128 kbps across all the different resolutions, since it would not have any positive impact if the audio content was encoded at the same bit rate as the video content. The different video representations provide a good mix for the goals of this project. It is important to note that the video was encoded in the H.264 format which is considered as one of the most popular and commonly used formats for video compression and distribution. In Table 3-1 the details of the different representations of the media content are summarised. In addition, two different versions of the media content were provided, one with segment duration of 1 second and the other with segment length of 10 seconds. These are the two most common segment sizes adopted by a wide variety of deployments.

**Table 3-1: Details of the media content.**

| Media Content - Google Chrome Advertisement | | | |
|---|---|---|---|
| Video content | | | Audio content |
| No. | Bit Rate | Resolution | Bit Rate |
| 1 | 400 kbps | 360p | 128 kbps |
| 2 | 800 kbps | 540p | 128 kbps |
| 3 | 1500 kbps | 720p | 128 kbps |

## 3.2 Server – side

The server component of the architecture setup is comprised of an Apache web server (version 2.4.20) running on an Ubuntu Linux 14.04 machine, able to accept HTTP/1.1 and HTTP/2 connections. The Apache web server project offers an open-source, secure, extensible and efficient HTTP server able to provide HTTP services (Group, 2016). It is currently the most popular web server available in the Internet and it is being used by a wide range of deployments. The Apache HTTP Server was firstly released in 1995 and support for HTTP/2 was introduced in October of 2015, just a couple of months after the official release of the protocol. Support for HTTP/2 in Apache is provided via the *mod_http2* module (mod_http2 - Apache HTTP Server Version 2.4, 2016). The server can easily be configured to accept HTTP/2 connections by updating the *Protocols* directive in the main configuration file. The *Protocols* directive is used to list all the protocols supported by the server. The Apache web server supports only the HTTP/1.1 protocol by default. In this project, the *Protocols* directive was configured to support, apart from HTTP/1.1, the HTTP/2 protocol's both encrypted and clear text versions. This was achieved by specifying the *Protocols* directive as follows: *Protocols h2 h2c http/1.1*. What's more, the server push feature can be easily enabled on Apache server by specifying the link of the desired resource as a link

header to a response. Link headers can be configured either by the application or via the *mod_headers* module of Apache. Several link headers can be configured for a particular application and therefore there is no limit to the amount of triggered pushes. An example of the configuration of multiple link headers using the *mod_headers* module can be seen in Figure 3-1.

```
<Location /index.html>
    Header add Link "</css/site.css>;rel=preload"
    Header add Link "</images/logo.jpg>;rel=preload"
</Location>
```

**Figure 3-1: Configuration of multiple link headers into a response (mod_http2 - Apache HTTP Server Version 2.4, 2016).**

A key part of this project was the restriction of the available bandwidth and also the configuration of various RTTs. To achieve bandwidth restriction, the Linux Traffic control (tc) and the Hierarchical Token Bucket (htb) tools were used. Taking into consideration Hubert (2002), the tc command can be used to shape outgoing traffic and apply bandwidth allocation for different ports and services. It is important to note that using the tc command, only data transmitted can be controlled and one has no control for incoming traffic to the server. Moreover, the htb approach is used to divide the available bandwidth for different tasks and for different configurations. This approach is based on the Token Bucket (TB) algorithm which is a widely used technique to slow down traffic on a particular interface (Hubert, 2002). The algorithm is consisted of a virtual bucket of a predefined size, continuously filled with data, called tokens. Furthermore, an important parameter of the algorithm is the token rate which is defined as the rate at which tokens arrive in the bucket. Based on the token rate, incoming data may pass without any delay or may cause the algorithm to throttle and packets may be dropped. The bandwidth was limited in each experiment in order to get a more consistent view and to eliminate huge variances to the results due to network fluctuations. Furthermore, the bandwidth was limited according to the type of the experiment. For DASH purposes, it was always limited to a sufficient value to provide the content at the highest

quality level. Following the use of Linux Traffic control, the netem tool of Linux was used to emulate the properties of wide area networks (networking:netem [Linux Foundation Wiki], 2016). The netem tool can be used to apply delay, packet loss, duplications and re-ordering of packets. In this project the netem command was used to apply packet delay and adjust RTTs for evaluation purposes. The various RTTs were configured in the range of 0 to 300 ms. The evaluation metrics were reported for different RTTs in the specified range in order to observe the impact of packet delay in the overall performance.

## 3.3 Client – side

The client component of the architecture setup is running on a Windows machine mainly because of the fact that the Chrome Canary browser (Chrome Browser, 2016) is supported only by Windows and OS X operating systems. The Chrome Canary browser is defined by Google as "the bleeding edge of the Web" and it supports all the latest development features of Chrome. It is installed as a separate browser and does not interfere with the Chrome browser. The Chrome Canary browser was selected in this project mainly because of its clear indication of which resources are pushed using the server push feature of HTTP/2. Each pushed asset in Canary is identified in the Initiator column as *Push/Other.* As a result, the user is able to easily separate the resources pushed by the server from all the other resources which have been requested by the client independently. As can be seen in Figure 3-2, the *Push/Other* indicator improves readability significantly.

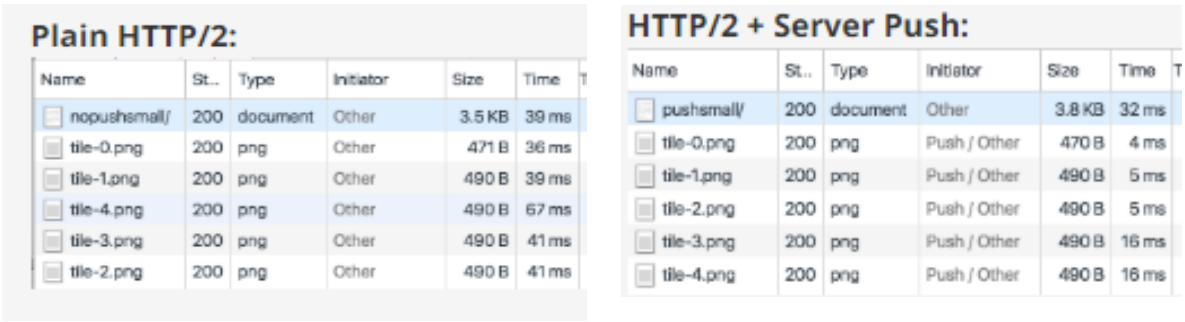


**Figure 3-2: An example of how Chrome Canary indicates pushed resources (Announcing Support for HTTP/2 Server Push, 2016).**

In order to playback MPEG-DASH streams, the dash.js framework (Dash-Industry-Forum/dash.js, 2016) was used. Dash.js is an open source JavaScript library which was initiated by the DASH Industry Forum. It was chosen for this project mainly because of its best performing adaption algorithms compared to the other DASH clients, open - source nature, agnostic behaviour in terms of codec and browser, support of a wide range of features and ease of use.

Following the selection of the DASH client, an important consideration for this project was the deployment of an automated mechanism to collect the results of the various experimental procedures. To achieve that, the desired values were calculated using the Navigation Timing API (Wang & Jain, 2013), which is a JavaScript mechanism used to measure the performance of a website in different stages, and they were stored in a text file using a simple PHP script. In this way, each time a web page was loaded the various performance results were stored in the text file and the collection of the data became much easier. Further details about the implementation of the automated script will be given in a subsequent section.

## 3.4 Implemented Software

Despite the fact that this project is mostly focused on evaluating the performance of the server push feature of HTTP/2 in media delivery, a significant attention was also given to the implementation of something new which could have a significant impact on the new HTTP protocol and the Apache web server. Therefore, this project examined and identified potential gaps or aspects which may need improvements. The way that resources are configured to be pushed by the Apache web server can definitely be considered as a gap to the overall implementation of the server and the support of HTTP/2. Currently, Apache supports the server push feature by specifying the link of a resource as a link header to a response. To do that, the configuration file of the server or the configuration file of a virtual host needs to be edited. Following that, the server

needs to be restarted or reloaded for the changes to be applied. As a result, a user who wants to apply changes and configure resources to be pushed needs to have full access to the server. Obviously, this is not always the case and in many cases is not possible. To address this issue, a web interface was developed to automate the overall procedure of configuring assets to be pushed. The web interface allows the user to select a file, based on the available files located on the server, and presents to the user the resources which can be configured to be pushed. Then, the user selects the desired resources and the update of the configuration file and the reload of the server are done automatically. It is important to note that the web interface is completely based on PHP. It is believed that the described implementation can facilitate the configuration of pushed resources on the Apache web server. A snapshot of the implemented web interface can be seen in Figure 3-3.

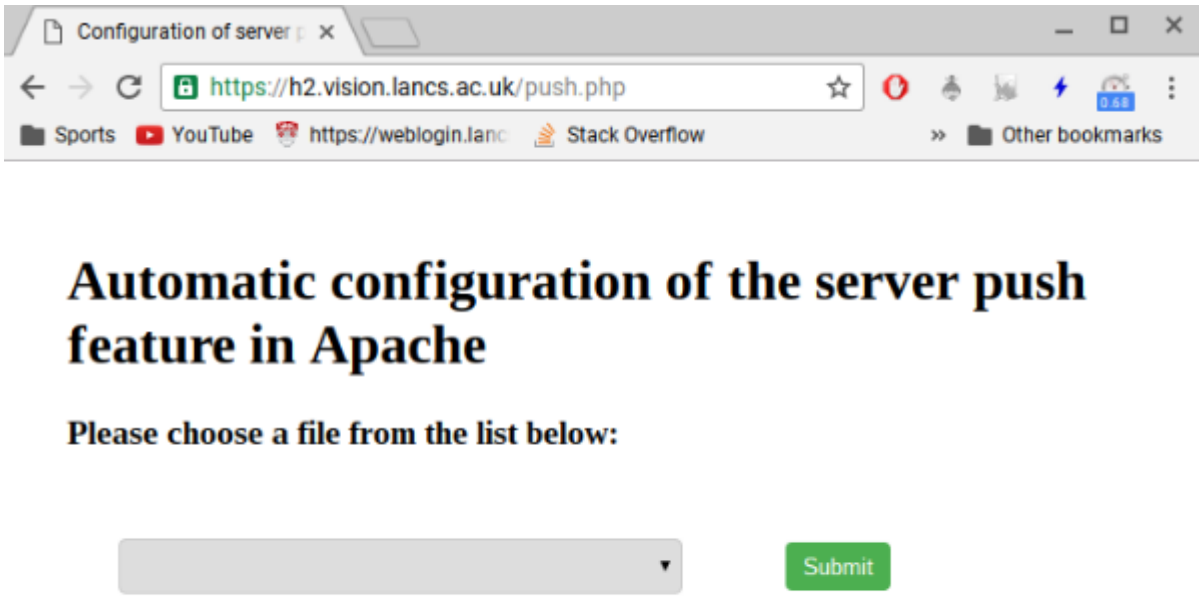


**Figure 3-3: A snapshot of the web interface.**

# 4 Implementation

This section describes the implemented server push strategies along with the system design. Details are given about the encoding and the preparation of the media content using the FFmpeg software and the MP4Box multimedia packager respectively. Moreover, this section gives special attention to the implemented web interface which aims to facilitate the configuration of pushed resources in the Apache web server. The implementation is explained in detail.

## 4.1 Server Push Strategies

Undoubtedly, the most interesting feature of HTTP/2 is the ability of the server to pre-emptively push resources to the client without receiving an explicit request for them. The pushed resources are entirely based on a previous initiated request from the client to the server. This ability of HTTP/2 protocol is called server push. Based on the identified literature work, server push can have a significant impact on the overall performance of HTTP streaming. Moreover, several experiments have shown that server push has several advantages in the context of adaptive media streaming, especially MPEG-DASH. Without server push, the client needs to send a separate request for each individual segment of the media content. Furthermore, the ability to push subsequent segments back-to-back, results in a significant reduction of the number of requests needed to playback a media content and also improves page loading times since multiple RTT cycles are saved. However, the most important aspect of the server push mechanism is the applied server push strategy, since not all the strategies are beneficial to the performance of adaptive streaming. A server push strategy defines which assets to push and when to push them. Furthermore, the fact that HTTP/2 does not define any specific strategy enhances customisability

and allows for multiple strategies to be implemented and adopted, based on the situation in which they will be applied. Particularly, in this project several server push strategies were considered and they are all discussed in detail below:

- No - Push Strategy: It is considered as the baseline for comparison and it is equivalent to the traditional HTTP/2 scheme. The client needs to send a request for each segment, both video and audio, and the server responds with the requested segment without pushing anything to the client. This is an extreme case where the total number of requests is the highest possible.
- All-Push Strategy: The client issues only a single request and all the subsequent segments are pushed by the server. When the client decides to switch to a new bit rate, it needs to cancel the push stream and issue another request for the segments of the new bit rate. This is considered as another extreme case since the client issues only a single request for the entire video playback.
- Initialise - Push Strategy: As soon as the server receives a request for the main file, it pushes the initialisation files of the DASH stream. The initialisation files include the script required for the use of dash.js, the manifest file (MPD) which links to all the available video and audio segments, the audio initialisation file and the initialisation file of the available video representations. These files are considered essential for the DASH stream and they are the first files which are requested by the client in a traditional scheme.
- Initialise + first audio/video segment - Push Strategy: A similar approach to the previous strategy, with two additional pushes. Besides the four initialisation files, the first segment pair of the medium quality representation is also pushed. Having performed some prior testing of the MPEG-DASH standard, it was observed that most of the times the medium quality representation is the first video segment which is requested by the client.
- Audio - Push Strategy: Upon receiving the first request, the server pushes the script required for the use of dash.js, the manifest file and the initialisation file of the available

video representations. When the server receives a request for the audio initialisation file, all the audio segments are pushed to the client. This mechanism saves a large number of requests.

The behaviour of the two extreme cases, the No – Push and the All – Push strategies, can be seen in Figure 4-1.

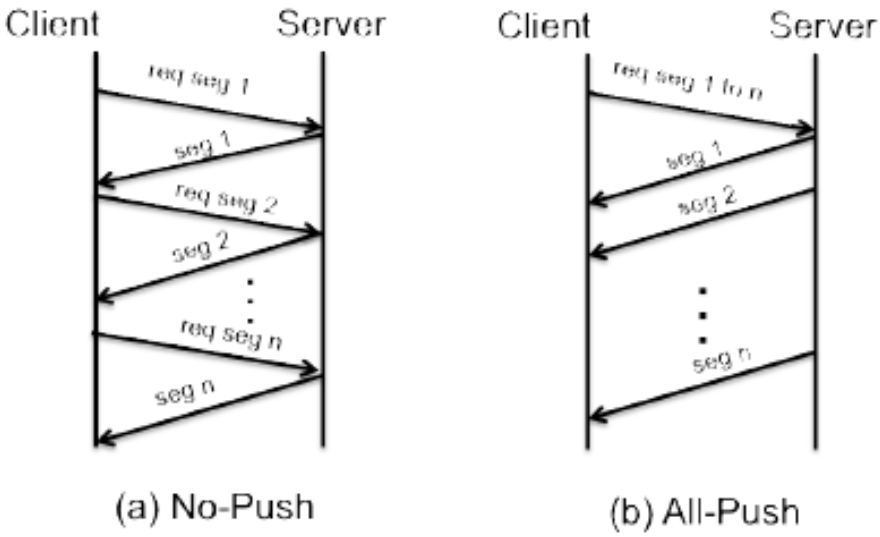


**Figure 4-1: A representation of the two extreme cases (Wei & Swaminathan, 2014).**

It is important to note that some of the server push strategies were implemented only for the 10 seconds segments since the smaller number of segments makes the configuration easier. This is based on the fact that the Apache web server configures pushed assets manually.

## 4.2 Encoding of the media content

In order to achieve the encoding of the file into different quality representations, the well-known tool, FFmpeg (FFmpeg, 2016) was used. FFmpeg is considered the leading multimedia framework able to perform a wide variety of tasks in a really simple way. The main functions of this tool is

the conversion of media files between different formats (ffmpeg), the streaming of audio and video using a streaming server (ffserver), the playback of media files using a simple media player and also the gathering of information about media streams and their presence in human and machine readable format (ffprobe). It has been used in this project to encode the media content into the different desired bit rates and resolutions. To do that, several parameters were configured and an example of a command used for a single representation (720p) is given in Figure 4-2.

```
ffmpeg -y -i inputfile -c:a libfdk_aac -ac 2 -ab 128k -c:v libx264 -x264opts 'keyint=24:min-keyint=24:no-scenecut' -b:v 1500k -maxrate 1500k -bufsize 1500k -vf "scale=-1:720" outputfile.mp4
```

**Figure 4-2: An example command for the encoding of the 720p representation.**

Initially, the command overwrites any files with the same name and specifies the input file to be encoded. Then, the highest quality Advanced Audio Coding (AAC) encoder available with FFmpeg, called *libfdk_aac*, is selected to be used for all the audio streams (Encode/AAC – FFmpeg, 2016). This specific encoding format is commonly used within an MP4 container. Moreover, the audio of the file is mixed down to two channels and the bit rate is set to 128 kbps. It is important to note that the audio parameters of all the representations are exactly the same. Following the configuration of the audio streams, the video streams are set to be encoded using *libx264*, which encodes all the video streams in the H.264/MPEG-4 AVC compression format. Despite the fact that H.264 format is the most commonly supported format in today's browsers, big attention in video encoding should be given in the so called I-frames. I-frames are frames that are comprised of information about the pixels in each image and they have the special characteristic of being able to be reconstructed without any reference to other frames of the video (Beavers, 2014). Each video segment needs to be started with an I-frame, otherwise the I-frames will not be aligned between the different video representations and quality switching will be

impossible. Obviously, users need to be able to switch between the different representations without any issue and as a result I-frames needed to be forced. Using FFmpeg, this is achieved by overwriting the default settings of the *libx264* library, setting the maximum and minimum number of frames between two I-Frames, called Group of Pictures (GOP) size, and removing key-frames on scenecuts. Furthermore, the bit rate of the video stream along with the buffer size needed to be set. It is important to note that in this project the buffer size is always equivalent with the bit rate of the corresponding segment. For the 720p representation the bit rate and the buffer size were set to 1500 kbps, for the 540p they were set to 800 kbps and for the 360p the bit rate and the buffer size were set to 400 kbps. Finally, the command specifies the scale filter, which sets the resolution of the representation and the name of the encoded output file.

## 4.3 Preparation of the media content for adaptive streaming

Having encoded the media file in the desired representations, the next step in the process was the creation of the various segments and the generation of the manifest file which are essential for MPEG-DASH streaming. This process was achieved using the MP4Box multimedia packager for two different segment durations (MP4Box | GPAC, 2016). The MP4Box is part of an open source multimedia framework, called GPAC which is widely adopted in research and academic purposes. The framework has various capabilities and offers a multimedia player, a multimedia packager and some server tools. The MP4Box can be used for a wide variety of tasks with most notable being:

- The manipulation of files including adding, removing and multiplexing audio, video and data.
- The encoding and decoding of media into/from binary formats.
- Stream encryption.
- Preparation of media content for adaptive streaming.

The open source nature of MP4Box, its widespread use and the ease in which it's operated were the main characteristics that lead to the adoption of MP4Box in this project. The command used for the preparation of the 1 second segments is given in Figure 4-3.

```
MP4Box -dash 1000 -segment-name mysegs_%s -out final.mpd outputfile720p.mp4#video outputfile720p.mp4#audio outputfile540p.mp4#video outputfile360p.mp4#video
```

**Figure 4-3: The command used for the creation of the 1 second segments.**

The way that the segmentation of the media content and the creation of the manifest file are achieved is really simple and this is one of the main advantages of MP4Box. Initially, the *-dash* parameter is used to create the segments as parts of the specified files and allow for individual HTTP download. Following that, the length of each segment needs to be configured in the range of milliseconds. In the above example, the duration of the segments is set to 1000 milliseconds or 1 second. Moreover, the names of the segments and the name of the manifest file are set using the *segment-name* and the *-out* parameters respectively. Finally, the files which are meant to be segmented and included in the adaptive process are specified. It is important to note that the video files of every representation are included along with a single audio file. A single audio file is sufficient to enable DASH streaming since the audio element of every representation is encoded to the same bit rate. A similar command was also executed but with the segment duration set to 10000 milliseconds (10 seconds).

## 4.4 Implementation of the Dash client and the automated script

Having created the different quality representations and having also prepared the content for adaptive streaming behaviour, the implementation of the DASH client was essential in order to examine and evaluate its behaviour. A video element needed to be created inside the body of an HTML file using the *video* tag. Following the creation of the video element, the necessary files for the implementation of DASH, which are located inside the dash.all.min.js file, were specified. Additionally, a JavaScript function was created in order to create and initialize the DASH media player, using the already created video element. Inside the function, the URI of the manifest file was specified and a DASH compatible player was created using the *MediaPlayer* class. Finally, the function initialised the media player using the video element and the URI of the manifest file. A snapshot of the streaming was captured and can be seen in Figure 4-4.

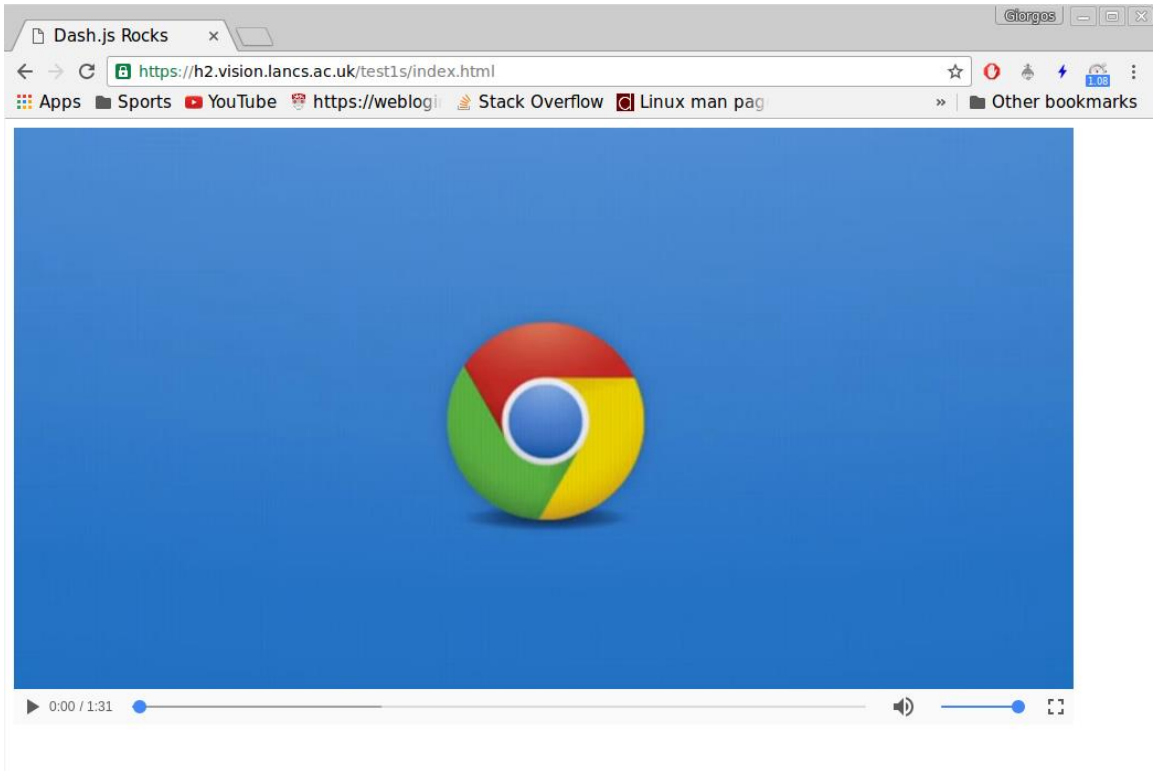


**Figure 4-4: A snapshot of the media streaming, using the DASH client.**

Another interesting part of the implementation was the script which was responsible to automate the process of acquiring and storing experimental results each time a page under consideration was loaded. To achieve that, the *Navigation Timing API* played a crucial role. Using that JavaScript mechanism, the user perceived page loading time and the Document Object Model (DOM) loading time were recorded. At this point, it is important to note that the DOM is defined as an interface providing a structure representation of the page in a way that can be accessed and updated from applications and programs (W3C Document Object Model, 2009). The loading time of the page as perceived by the user was calculated by subtracting the time immediately after the browser finished prompting to unload the previous page, from the value returned by the *Date()* class (number of milliseconds since 01/01/1970). On the other hand, the DOM loading time was calculated by subtracting the time before the browser starts loading the page from the time immediately after the loading has completed. These values were passed to a PHP script which was responsible for storing the results on the server. The transfer of the data was achieved using the *XMLHttpRequest API* which provides the ability to transfer data between two endpoints (XMLHttpRequest, 2016). The special feature of this JavaScript library is that it doesn't require a full page refresh and therefore it provides a seamless data transfer without disrupting the user. It is mostly used in Asynchronous JavaScript and XML (AJAX) programming. Furthermore, the process of calculating the required values and transferring the data to the server, was performed upon the loading of the main body of the page. The file which was responsible for storing the results on to the server is nothing more than a simple PHP script which utilises the *file_put_contents()* function and simply writes the transferred data to a specified text file.

## 4.5 Implementation of the software

The developed web interface, which has as its primary goal the automation of the configuration of server push assets, is purely based on PHP. Two scripts were used for the implementation. The

first one is responsible for all the functionalities and the behaviour of the web interface while the other one is used to automate the procedure of updating the Apache configuration file. The initial display of the web interface provides a drop-down list to the user with all the available HTML files stored on the server. In order to retrieve all the available HTML files from the server, the *glob()* function is utilised. This function is able to perform an extensive search to find any matches of a predefined pattern. In this case the HTML extension is being used as the matching pattern. Each identified HTML file is added to the drop-down list. It is important to note that the drop-down list has been created using the *select* element of HTML. Then, the user needs to select a file from the list and update that specific file's pushed resources. Upon submitting a request for a file, the user is presented with a list of all the available files which can be configured as assets to be pushed. To do that, the selected file needs to be compressed and examined for any *href* or *src* tags. These are the tags which are used by HTML to specify link destinations and therefore they can be used to identify any resources which can be pushed. To find these tags, the *preg_match_all()* function of PHP has been used, which is able to find all matches of a regular expression and store them in an array. The identified resources are presented to the user in the form of checkboxes. Any resource which already exists in the Apache configuration file as a pushed asset, is presented with a pre-selected checkbox. In case that there are no available files which can be pushed or the user doesn't specify a file, the appropriate message is displayed. Upon selecting the desired resources, the user submits the request and a new configuration file is created with the name of the file and the selected assets, formatted in the same way as in the Apache configuration file. Based on the type of the resources, the link headers are added accordingly. In the case that the user doesn't select any file to be pushed, the new configuration file is still created with the name of the file and a message used to identify that there are no resources to be pushed. This message is used to distinguish a file with no pushes during the update of the main configuration file. The update is performed every minute and is handled by the second script which comprises this implementation. To improve readability and the user experience, a JavaScript-based timer is used to calculate the

time remaining for the next update. When the update has been performed, the appropriate message is displayed to the user. Therefore, the user is aware of the time needed for his changes to be applied.

As described above, the update of the configuration file is handled by another PHP script which is executed every minute. The script is aware of the new configuration file and checks if the file exists every time it is executed. This is done using the *file_exists()* function of PHP. Having discovered the new configuration file, the script adds its contents to the Apache configuration file and rearranges the file accordingly. Also, any unnecessary spaces or empty lines are removed. At the end, the new configuration file is deleted and the Apache server is reloaded. It is important to note that the script is configured to be executed every minute using the *crontab* job scheduler of Linux.

# 5 Experimental Analysis

This section presents the experimental results as reported by the evaluation. Initially, the various evaluation metrics are presented and explained in detail and then the section focuses on the numerous experiments conducted. It is important to note that all the server push strategies were evaluated against all the evaluation metrics. The section also considers the impact of the HTTP/2 protocol and the server push feature in standard webpages.

## 5.1 Evaluation Metrics

To demonstrate the possible gains of the HTTP/2 protocol and the server push feature, various test web pages and push strategies were implemented and evaluated in terms of multiple evaluation metrics. In this project, the following metrics were considered:

- **User - Perceived page loading time** which is defined as the time difference between the time immediately after the browser finishes prompting to unload the previous page and the value returned by the *Date()* class (number of milliseconds since 01/01/1970). It can be considered as a reasonable measure of the latency as seen by the user.
- **DOM - Content loading time** which is defined as the time difference between the time before the browser starts loading the page and the time immediately after the loading of the page has completed.
- **Number of requests** which is defined as the total number of requests required by the client for the entire video playback.
- **Link utilisation** which is defined as the ratio of the media throughput and the available bandwidth which is restricted in every experiment using the Linux tc command tool.

- **Initial start-up delay** which is defined as the time required to start playing the first video segment on the DASH client.
- **Bandwidth Overhead in terms of unclaimed pushes** which is defined as the overhead introduced by the pushed resources which have been configured to be pushed but remained unused by the application.

To evaluate the impact of the segment duration and the different network conditions, several experiments were conducted for both segment durations (1 second and 10 seconds) for various RTTs in the realistic range of 0 ms to 300 ms. Moreover, bandwidth was constrained according to the type of the content (DASH or simple webpage) to get more consistent results. In this project, each server push strategy was evaluated against each evaluation metric and the protocols under consideration were: HTTP/1.1, HTTP/2 without server push and HTTP/2 with server push. To eliminate any outliers, each one of the experiments was repeated multiple times and therefore the results report the average values.

## 5.2 Web server Evaluation

Despite the fact that Apache is the most popular web server available, the literature reported that numerous web servers were also used to examine the benefits of HTTP/2 protocol. One of them is the Java-based web server, called Jetty (Jetty - Servlet Engine and Http Server, 2016), which is hosted by the Eclipse Foundation and has been used by many major studies (Wei et al., 2015; Huysegems et al., 2015; Wei & Swaminathan, 2014). It has the ability to serve both static and dynamic content and it is adopted by commercial and open source products. Additionally, it supports HTTP/2 and the server push feature. The Jetty web server was also considered as an option for this project and so a simple performance comparison was performed between the two. To achieve that, both of the servers were implemented and evaluated in terms of page loading times, using HTTP/1.1 and HTTP/2, by serving a standard web page (multiple images) and a

DASH video streaming. Having performed the evaluation, the Apache server was able to load the standard web page twice as fast compared to Jetty for both HTTP/1.1 and HTTP/2. This was also the case in the DASH streaming scenario. Moreover, based on the fact that the server push feature occupies a big part of this project, the behaviour of both servers regarding server push was also examined. Both of the servers support the server push feature but in a different way. The Apache server enables HTTP/2 push mechanism by default, from version 2.4.17 and later, and configures new resources to be pushed by adding link headers in responses. All of these are done by editing the configuration file of the server. On the other hand, Jetty doesn't have server push enabled by default but it can be easily enabled by configuring a *PushCacheFilter* in the *web.xml* file of an application. Furthermore, the *PushCacheFilter* has the ability to analyse and organise the various resources which are needed by an application into primary and secondary resources. The secondary resources which have been requested in a predefined time window, are automatically pushed. Jetty gives the ability to update that time window and also the number of secondary resources related to a primary resource. However, the major performance advantages of the Apache server compared to Jetty could not be ignored and therefore Apache was selected as the server for this project. Apart from that, the fact that this was the first performance evaluation of the Apache web server using the server push mechanism of HTTP/2, was also a very important motivation.

## 5.3 Evaluation of user – perceived and DOM – content loading times

Having selected the web server and prior of making any evaluation using the server push feature, the first experiment was conducted in order to get a better idea about the advantages of HTTP/2 over HTTP/1.1, on a standard web page and on a DASH video streaming scenario.

### 5.3.1 Standard Webpage

The standard web page was consisted of 20 PNG images and 1 stylesheet file. The bandwidth for this experiment was restricted to 1 Mbit and the results of the evaluation of both protocols under

varying RTTs in terms of DOM loading time and page loading time, can be seen in Figures 5-1 and 5-2.

**Table 5-1: Average values of the evaluation for the standard web page.**

| Standard Web page | | | | |
|---|---|---|---|---|
| RTT (ms) | HTTP/1.1 | | HTTP/2 | |
| | Dom page loading time (ms) | User - perceived page loading time (ms) | Dom page loading time (ms) | User - perceived page loading time (ms) |
| 0 | 152.4 | 169.1 | 117.6 | 142 |
| 100 | 636.5 | 957.3 | 308.6 | 629.9 |
| 200 | 1042.1 | 1662.3 | 536.6 | 1150.6 |
| 300 | 1590 | 2505.8 | 811.2 | 1721.6 |

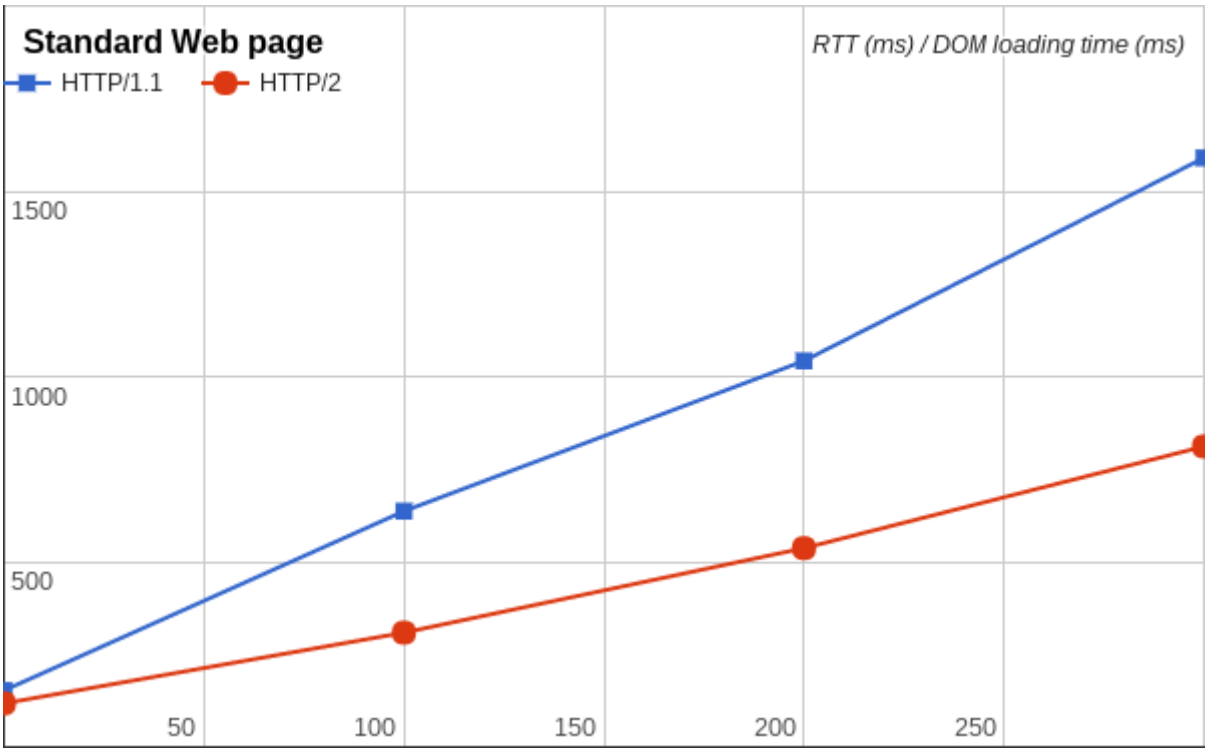


**Figure 5-1: Results of DOM content loading times for a standard web page.**

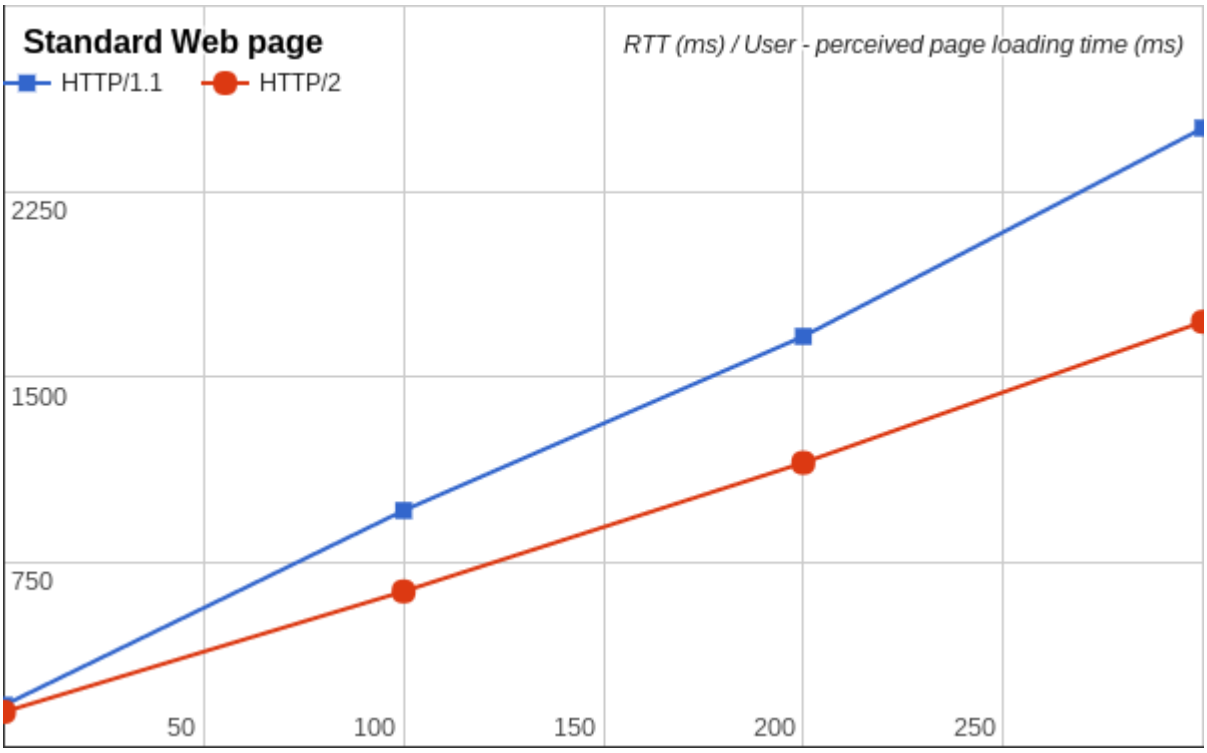


**Figure 5-2: Results of user - perceived page loading times for a standard web page.**

The horizontal axis of the graphs represents the RTT and the vertical axis represents the DOM loading time and the user - perceived loading time respectively. It can be observed that HTTP/2 protocol improves the DOM loading time and the user latency significantly. More specifically, the HTTP/1.1 requires double the time required by the HTTP/2 protocol to load the DOM object, while HTTP/2 is almost ⅓ faster than HTTP/1.1 regarding user - perceived loading time. These performance improvements can be mainly attributed to the following reasons:

- Header Compression: Due to the header compression feature of HTTP/2, duplicate header frames are avoided and a single header frame is sent for the entire connection. On the other hand, HTTP/1.1 sends a separate header frame for each individual request and response.
- TCP connections: HTTP/2 protocol establishes a single TCP connection for the entire communication between the client and the server and therefore every resource is sent in the same connection. This is not the case in HTTP/1.1, where multiple connections are established on the server.

The average values of the evaluation for the standard webpage are shown in Table 5-1.

### 5.3.2 DASH video streaming

Following the evaluation of the two protocols for the standard web page, the DASH streaming was evaluated under similar conditions for segment durations of 1 and 10 seconds. The bandwidth for the video streaming was limited to 2 Mbits. The performance of both segment durations regarding DOM loading time and user perceived page loading time can be seen in Figures 5-3 and 5-4.

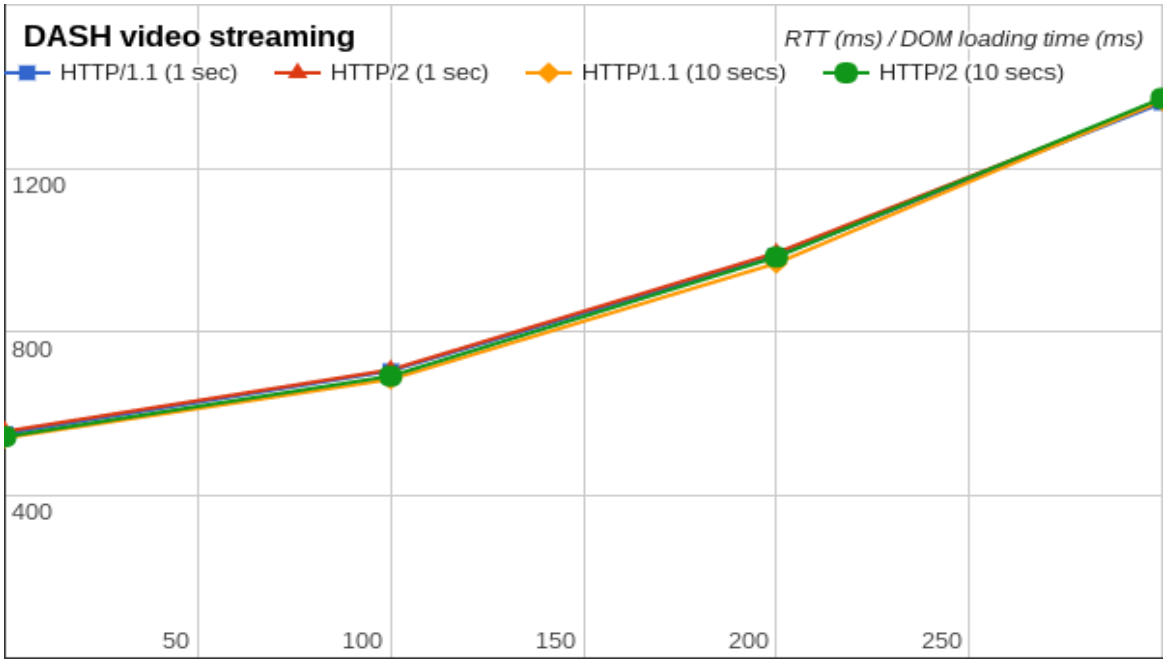


**Figure 5-3: Results of DOM content loading times for both segment durations.**

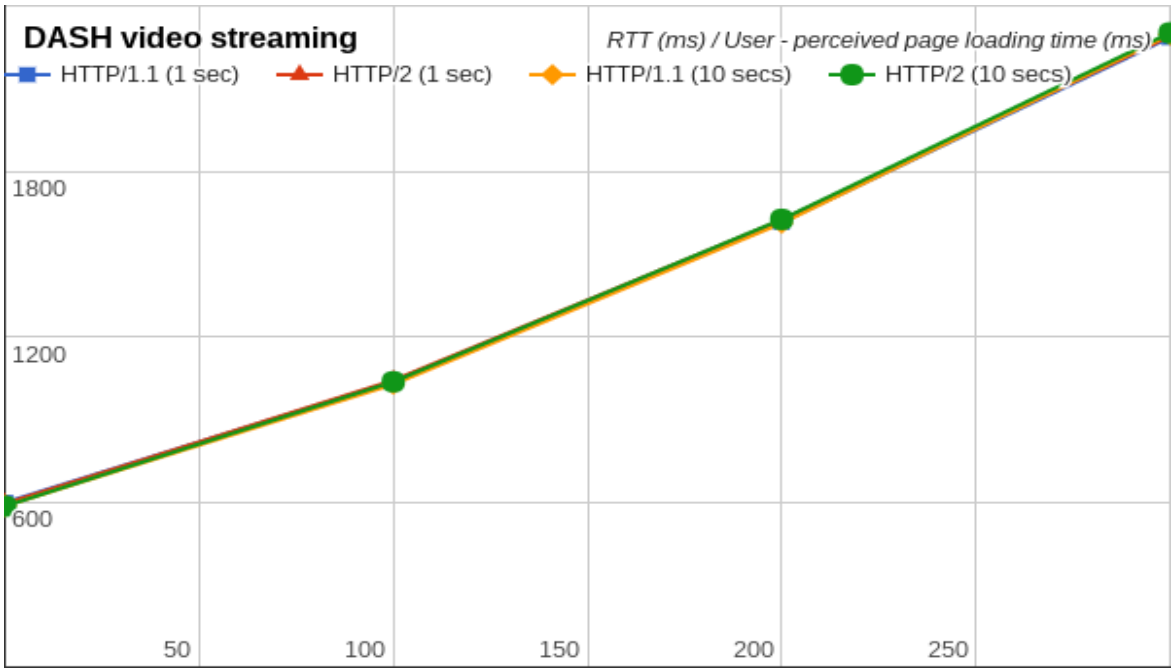


**Figure 5-4: Results of user - perceived page loading times for both segment durations.**

Based on the evaluation results, it can be concluded that the performance of HTTP/1.1 and HTTP/2 for both segment durations is comparable, with almost identical results. However, it is important to note that the media content with a segment duration of 10 seconds achieved slightly better results for both evaluation metrics. It can be concluded that raw HTTP/2 doesn't seem to improve DASH video streaming in terms of user perceived latency and DOM loading time. Further evaluation was conducted utilising the server push feature and proved that this is not the case. The average values of the evaluation for the DASH scenarios (both segment durations) for HTTP/1.1 and HTTP/2 can be seen in Tables 5-2 and 5-3.

**Table 5-2: Average values of the evaluation for the 1 second segments.**

| DASH (1 sec segments) | | | | |
|---|---|---|---|---|
| RTT (ms) | HTTP/1.1 | | HTTP/2 | |
| | Dom page loading time (ms) | User - perceived page loading time (ms) | Dom page loading time (ms) | User - perceived page loading time (ms) |
| 0 | 549.4 | 597.4 | 555.2 | 593.2 |
| 100 | 703.6 | 1036.5 | 706.2 | 1041 |
| 200 | 985 | 1614.7 | 992.4 | 1622.7 |
| 300 | 1359.5 | 2285.3 | 1364.9 | 2290 |

**Table 5-3: Average values of the evaluation for the 10 second segments**

| DASH (10 secs segments) | | | | |
|---|---|---|---|---|
| RTT (ms) | HTTP/1.1 | | HTTP/2 | |
| | Dom page loading time (ms) | User - perceived page loading time (ms) | Dom page loading time (ms) | User - perceived page loading time (ms) |
| 0 | 539.3 | 584.4 | 542.3 | 584.1 |
| 100 | 682.8 | 1026.2 | 690.6 | 1035.6 |
| 200 | 966.9 | 1610.9 | 982.4 | 1625 |
| 300 | 1365.8 | 2296 | 1371.1 | 2301.9 |

## 5.3.3 Server Push

### 5.3.3.1 Standard Webpage

Initially, the impact of the server push feature was examined on the standard web page. All the resources of the page (20 images and 1 stylesheet file) were configured to be pushed upon the request of the main file and therefore a single request was needed to load the whole page. As can be seen in Figures 5-5 and 5-6, and in the Table 5-4, the push mechanism was able to optimise the DOM loading time by up to 37% for a RTT of 0ms and by more than 60% for a RTT of 200ms, compared to the results obtained when HTTP/1.1 was used. Compared to HTTP/2, the server push mechanism optimised user-perceived page loading time by more than 11% for high RTT networks and by more than 16% for low RTT networks. Based on the results reported in Table 5-4, the highest improvements of the server push features were recorded when the RTT was set to 100ms.

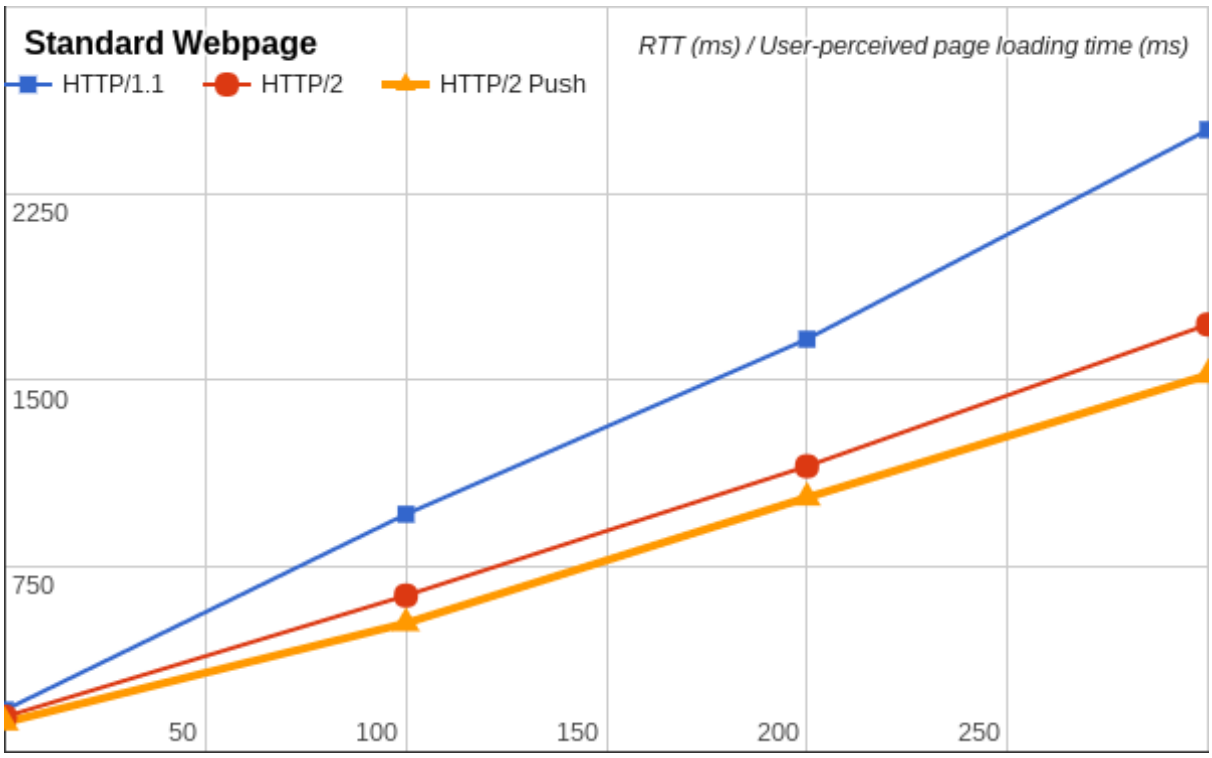


**Figure 5-6: The impact of the server push feature on the user - perceived page loading times of a standard web page.**

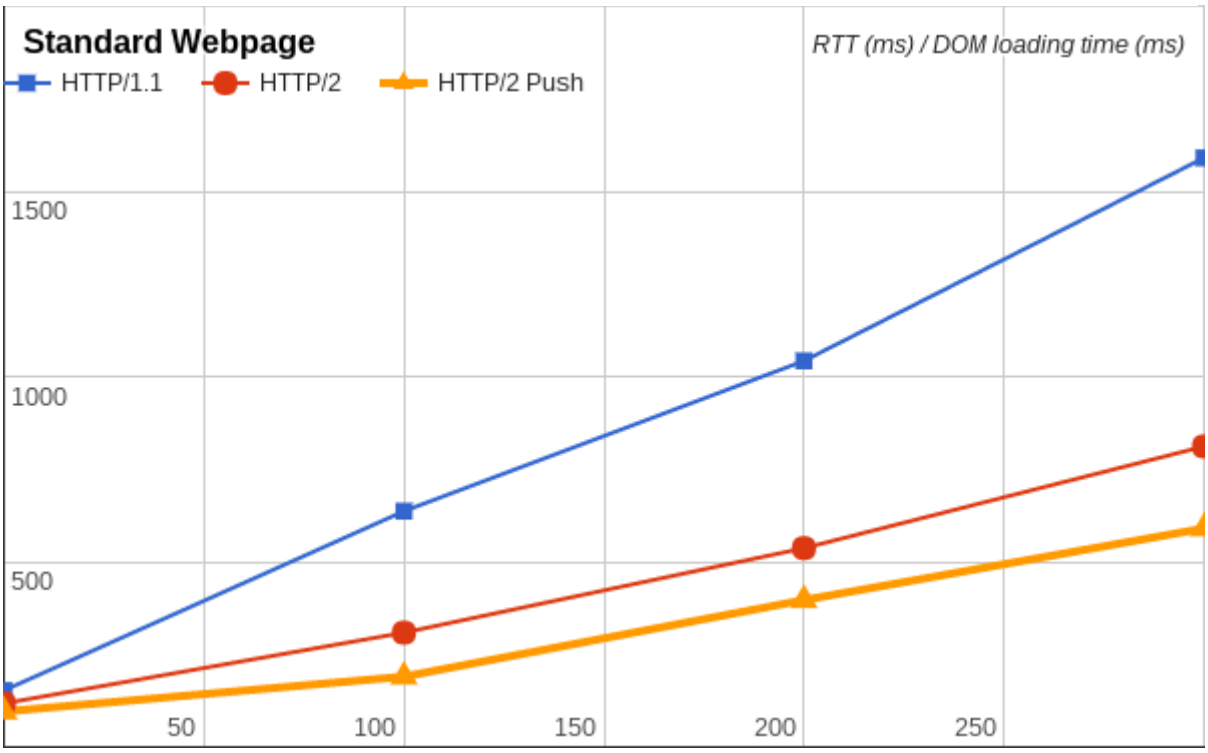


**Figure 5-5: The impact of the server push feature on the DOM content loading times of a standard web page.**

**Table 5-4: The average results of the server push feature and the reported improvements compared to HTTP/1.1 and HTTP/2 for the standard webpage.**

| Standard Web page - Server Push | | | | | | |
|---|---|---|---|---|---|---|
| RTT (ms) | HTTP/2 - Server Push | | Improvement Compared to HTTP/1.1 | | Improvement Compared to HTTP/2 | |
| | Dom page loading time (ms) | User - perceived page loading time (ms) | Dom page loading time | User - perceived page loading time | Dom page loading time | User - perceived page loading time |
| 0 | 95.6 | 118.7 | 37.27 % | 29.80 % | 18.70 % | 16.41 % |
| 100 | 189.9 | 518.7 | 70.16 % | 45.82 % | 38.46 % | 17.65 % |
| 200 | 396.6 | 1024.4 | 61.94 % | 38.37 % | 26.09 % | 10.97 % |
| 300 | 589.9 | 1518.3 | 62.90 % | 39.41 % | 27.28 % | 11.81 % |

The main reason behind the significant gains of the server push mechanism is the fact that no additional requests are being issued in order to get the multiple resources of the page. The initial request of the HTML file is all that is needed to download all the assets of the page. As a result, multiple RTTs are being saved and the DOM and user perceived loading times are further reduced, optimising the overall performance of the application. Combining the gains in RTT cycles with the headers compression feature and the establishment of a single TCP connection for the entire session, has led to some dramatic improvements compared to HTTP/1.1. Based on the results recorded by the web page evaluation, it was concluded that the server push mechanism is able to offer some significant advantages in DOM loading time and user perceived latency.

#### 5.3.3.2 DASH video streaming

However, a DASH streaming scenario is much more complex than a simple web page and the selected push strategy is not an easy procedure. Multiple approaches could be adopted and therefore multiple strategies were implemented and evaluated. To begin with, a DASH video streaming for both segment durations was evaluated using the Initialise - Push Strategy, where the 4 initialisation files are set to be pushed upon the request of the main file

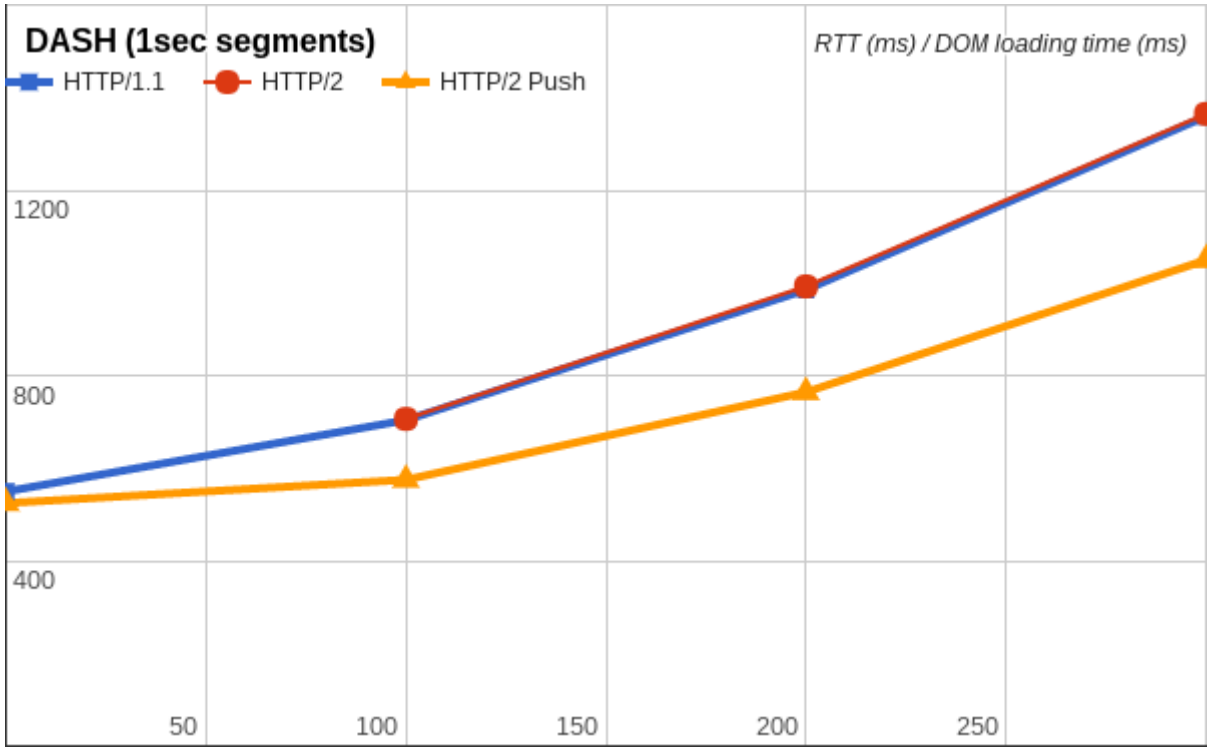


**Figure 5-7: The impact of the server push feature on the DOM content loading times of a DASH video streaming for a segment duration of 1 second.**

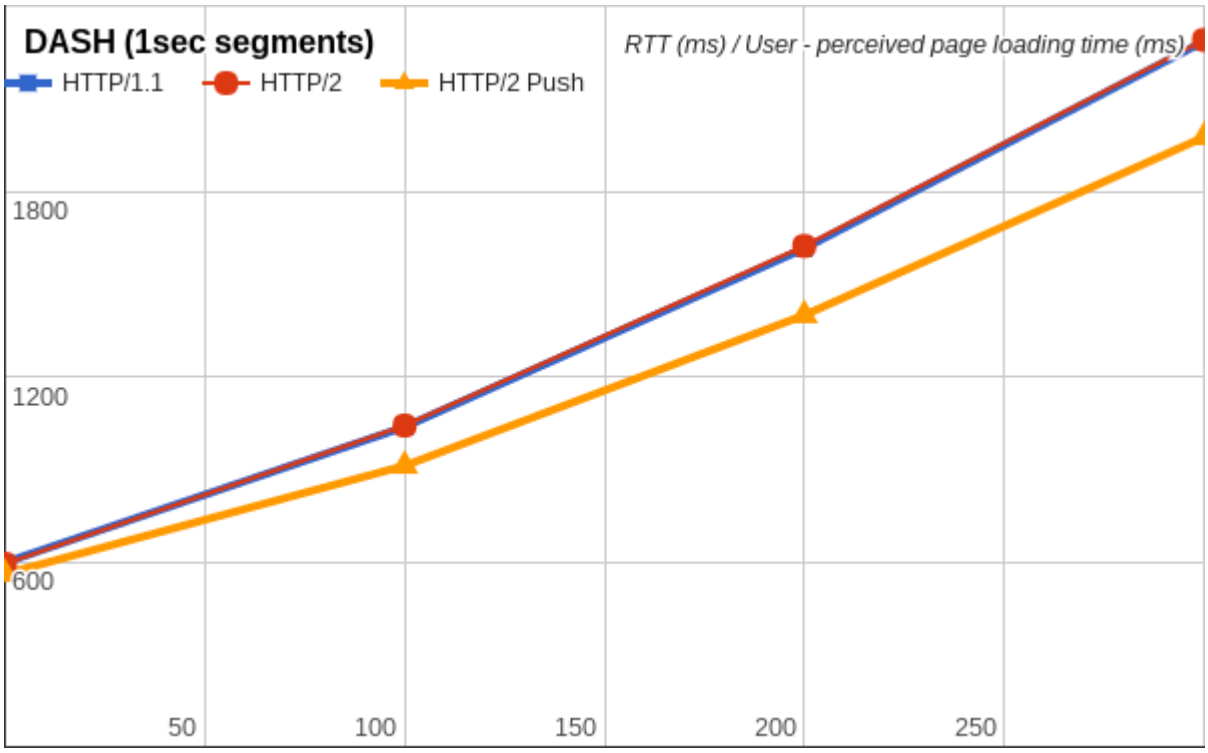


**Figure 5-8: The impact of the server push feature on the user - perceived page loading times of a DASH video streaming for a segment duration of 1 second.**

In Figures 5-7 and 5-8 the results of the DOM loading times and the user-perceived latencies for a DASH video streaming with a segment duration of 1 second, are shown. Based on the results, the server push feature can decrease both evaluation metrics by a significant margin. It can also be observed that the improvement in loading times is further increased for higher RTT networks. Moreover, it is expected that the performance of the streaming would be further improved for even higher RTTs. Similar with the applied server push mechanism on the standard web page, the main reason of that improvement is the gains of multiple RTTs. The four initialisation files are pushed as soon as a request is received for the main HTML file and therefore there is no need for the client to issue requests for these resources. Compared to HTTP/1.1, the applied push strategy is able to achieve improvements of up to 22 .82% and 13.57% for DOM and user - perceived loading times respectively. Compared to HTTP/2, improvements of more than 23% and 13% are achieved. A summary of the results and the improvements achieved can be seen in Table 5-5.

**Table 5-5: The average results of the server push feature and the reported improvements compared to HTTP/1.1 and HTTP/2 for a DASH video streaming (1 sec).**

| DASH video streaming (1 second segments) - Initialise Push strategy | | | | | | |
|---|---|---|---|---|---|---|
| RTT (ms) | HTTP/2 - Server Push | | Improvement Compared to HTTP/1.1 | | Improvement Compared to HTTP/2 | |
| | Dom page loading time (ms) | User - perceived page loading time (ms) | Dom page loading time | User - perceived page loading time | Dom page loading time | User - perceived page loading time |
| 0 | 524.3 | 560.4 | 4.57 % | 6.19 % | 5.57 % | 5.53 % |
| 100 | 574.8 | 910.9 | 18.31 % | 12.12 % | 18.61 % | 12.50 % |
| 200 | 764 | 1398.6 | 22.44 % | 13.38 % | 23.01 % | 13.81 % |
| 300 | 1049.3 | 1975.2 | 22.82 % | 13.57 % | 23.12 % | 13.75 % |

Following the evaluation of the server push feature on the 1 second segments, a similar evaluation was performed for a segment duration of 10 seconds in order to examine the impact of the segment length when the server push feature is applied. In Figures 5-9 and 5-10, the results of the DOM loading times and the user-perceived latencies for a DASH video streaming with segment duration of 10 seconds, are shown. As with the 1 second segments, it was also observed that server push can improve both evaluation metrics significantly and can achieve a performance improvement of more than 13%. More specifically, the evaluation reported an increase of more than 23% compared to both protocols, regarding DOM loading time. On the other hand, the improvement of the user perceived latency surpassed 13 % for both protocols. Once again, the performance improvements for both protocols were comparable.

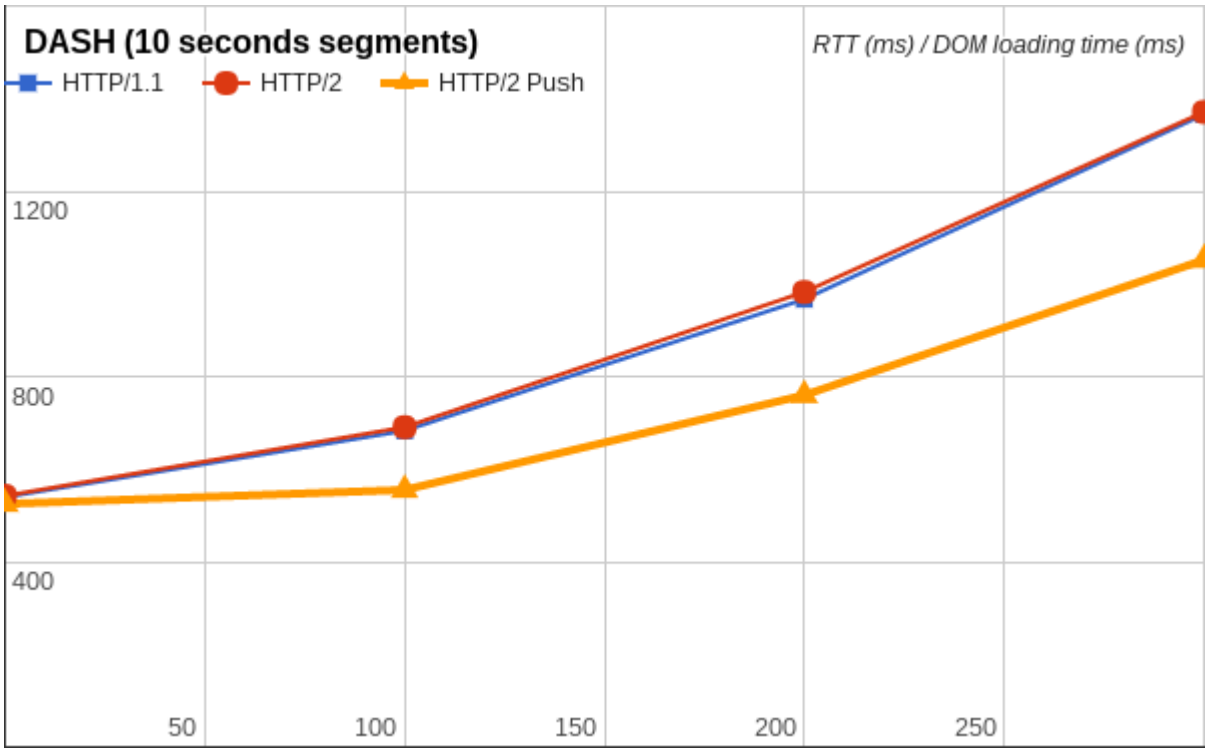


**Figure 5-9: The impact of the server push feature on the DOM content loading times of a DASH video streaming for a segment duration of 10 seconds.**

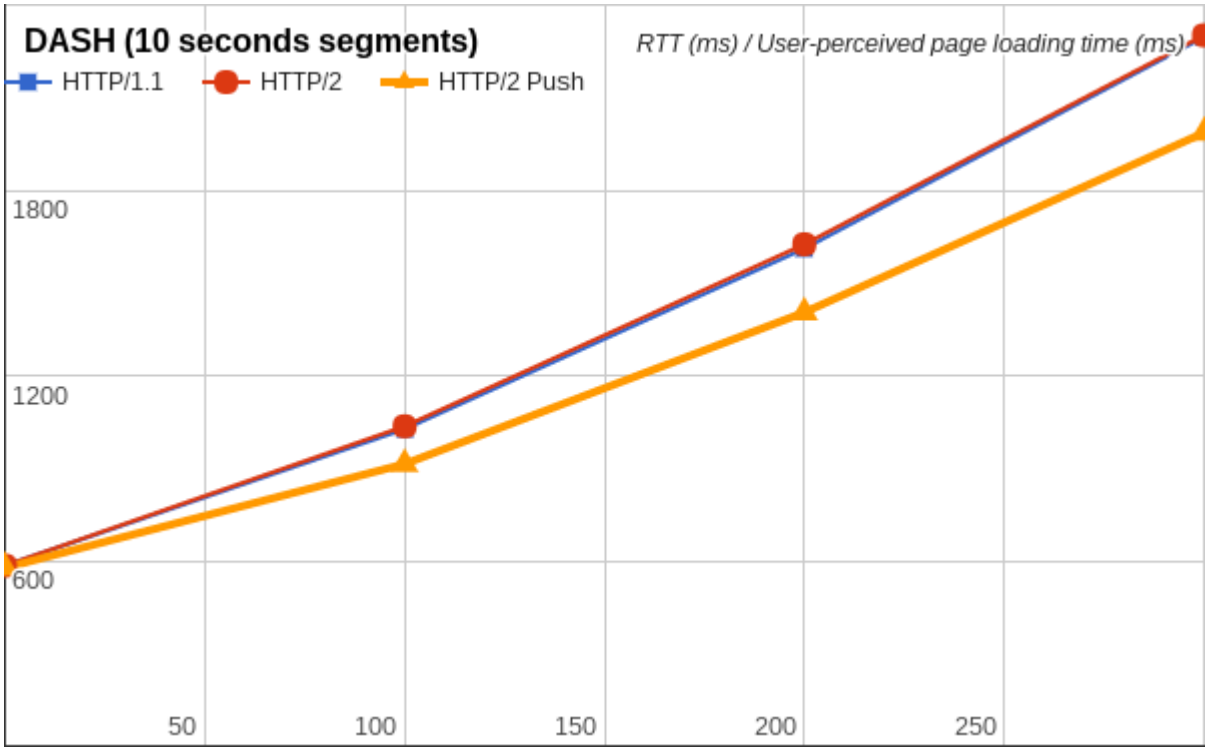


**Figure 5-10: The impact of the server push feature on the user - perceived page loading times of a DASH video streaming for a segment duration of 10 seconds.**

Based on the detailed results reported in Table 5-6, it can be concluded that the DOM and user-perceived page loading times for both segment durations are comparable. Both segment durations achieved similar results but when the media content was split into 10 seconds segments, slightly better DOM loading times were recorded. However, the evaluation of the 1 seconds segments reported better loading times for the latency as perceived by the user. The most straightforward reason behind the better user-perceived loading times of the 1 second segments, is the smaller size of segments which can have a significant impact on latency.

**Table 5-6: The average results of the server push feature and the reported improvements compared to HTTP/1.1 and HTTP/2 for a DASH video streaming (10 secs).**

| DASH video streaming (10 seconds segments) - Initialise Push strategy | | | | | | |
|---|---|---|---|---|---|---|
| RTT (ms) | HTTP/2 - Server Push | | Improvement Compared to HTTP/1.1 | | Improvement Compared to HTTP/2 | |
| | Dom page loading time (ms) | User - perceived page loading time (ms) | Dom page loading time | User - perceived page loading time | Dom page loading time | User - perceived page loading time |
| 0 | 524.8 | 577.2 | 2.69 % | 1.23 % | 3.23 % | 1.18 % |
| 100 | 555.1 | 913.3 | 18.7 % | 11 % | 19.62 % | 11.81 % |
| 200 | 759.7 | 1404.3 | 21.43 % | 12.83 % | 22.67 % | 13.58 % |
| 300 | 1051.6 | 1986.3 | 23 % | 13.49 % | 23.3 % | 13.71 % |

At this point it is important to note that the other push strategies were also evaluated in terms of DOM and user-perceived page loading times. However, the reported results were almost identical to the Initialise - Push strategy and therefore they are not presented here. Finally, it can be concluded that server push has a significant impact on DOM and user perceived page loading times in DASH streaming scenarios.

## 5.4 Evaluation of the start-up delay

In order to further evaluate the impact of segments duration in adaptive bitrate streaming, the initial start-up delay was considered. The start-up delay is defined as the time needed for the first video segment to start playing on the client. Moreover, the impact of the different server push strategies in the initial start-up delay was also considered. The results for the initial start-up delay were collected by observing the value of the "Native video element playing" event. Initially, the two different segment durations were evaluated under HTTP/2 without the server push mechanism. The results are summarised in Table 5-7.

**Table 5-7: Results of the initial start-up delays for both segments durations.**

| | DASH 1 second segments (ms) | DASH 10 seconds segments (ms) |
|---|---|---|
| Initial start - up Delay | 595.4 | 4664.8 |

The above results report the average values for each experiment. Based on the results, it can be observed that there is a huge difference in the initial start - up delay between the two segment durations. The DASH streaming, with the 1 second segments, achieved a start - up delay of just 595 ms while the streaming with 10 seconds segments reported results of more than 4.5 seconds. The reason behind this huge difference is the size of each segment. In the case of the 1 second segments, each segment has a size of no more than 16 kb and therefore it can be downloaded much faster than a 10 seconds segment with a size of 160 kb. Following that, the different push strategies were also evaluated to examine if they have any impact on the initial start-up delay. The Initialise and the Initialise + first audio/video segment push strategies both managed to reduce the start-up delay. However, the latter reported slightly better results. The reason behind the optimised results of these two strategies is based on the pushed initialisation files. To start playing the first video segment in a DASH streaming scenario, a client needs to have access to the initialisation files. Based on the fact that these files are pushed by the server in both techniques, the overall process

is optimised since they become available much faster than when they are individually requested. More specifically, when the two techniques were applied for the case of the 10 seconds segments, the start-up delay was improved by 1 % in the Initialise - Push strategy and by 3.68 % in the Initialise - Push + first audio/video segment strategy. However, this was not the case when the other two strategies were applied. Again for the 10 seconds segments, the Audio - Push strategy increased the start-up delay by more than 5 seconds which resulted to a total delay of more than 10 seconds in order to start playing the first video segment. In the case of the All - Push strategy, the total delay was even higher. It is believed that the main reason behind the huge delay of these two strategies is the total number of resources that are pushed to the client upon the request of the main file. Many of these files are not immediately required by the client but despite that, all the files need to be pushed in order to start the video streaming. This process may take significant time and therefore the start-up delay is increased. It can be concluded that the server push feature can improve the start-up delay but this is not the case for all the strategies.

## 5.5 Evaluation of the total number of requests

Another significant evaluation metric of this project was the number of requests required to playback the entire media content. Undoubtedly, a huge number of requests can lead to a request explosion and therefore undesired user perceived latencies. In order to address the request explosion problem and minimize the overall latency, the server push feature of HTTP/2 can be

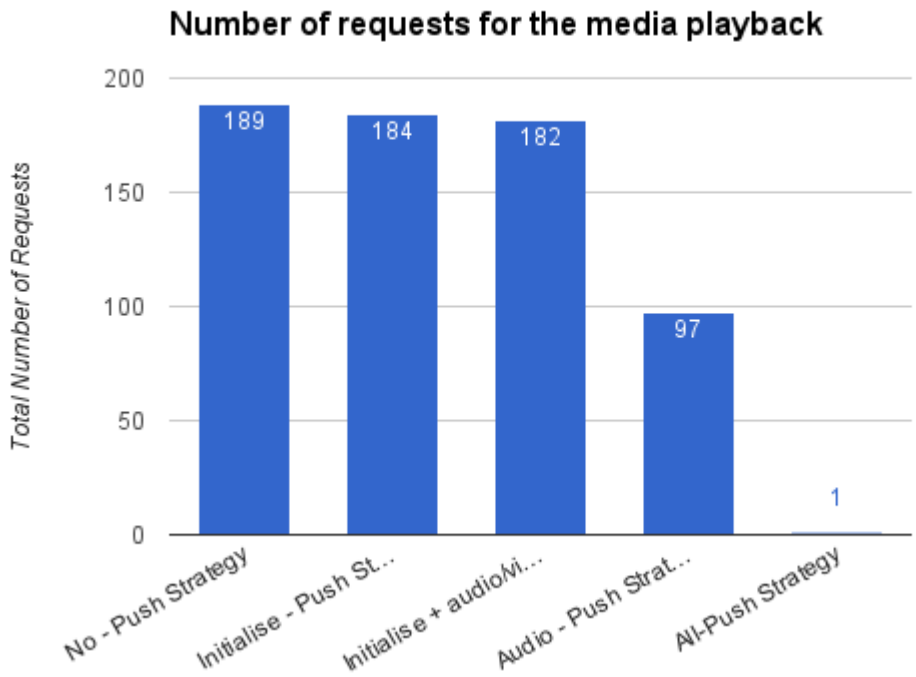


**Figure 5-11: Total number of requests for each server push strategy.**

leveraged. The different server push strategies were evaluated and the results can be seen in Figure 5-11.

The above results report the total number of requests required by each different server push strategy in order to playback the entire video content. The vertical and the horizontal axes represent the total requests and the different push strategies respectively. When the HTTP/2 protocol is used without the server push feature, each resource has to be individually requested and this equals to 189 total requests. This number includes the 92 video segments, the 92 audio segments, the 4 initialisation files and the main HTML file. The No-Push strategy represents the HTTP/2 protocol without the server push mechanism and therefore it requires 189 total requests for the entire

playback. Having said that, each of the other server push strategies managed to minimise the total number of requests. When the Initialise - Push strategy was applied, the 4 initialisation files required for the streaming were pushed upon the request of the main HTML file and therefore 4 requests were saved. In addition, the Initialise + first audio/video segment - Push Strategy managed to save two more requests compared to the Initialise - Push strategy and therefore it required a total of 184 requests for the entire media playback. Despite the fact that these two strategies managed to save 4 and 6 requests respectively, the total number of requests required is not far from the highest available. This is not the case for the remaining two strategies. After evaluating the Audio - Push strategy, it was reported that 97 total requests were required for the media playback, 92 requests less than the No - Push strategy. The total number of requests of the Audio - Push strategy was reduced significantly since each individual audio segment was pushed from the server upon the request of the audio initialisation file. As a result, no request was issued from the client for any audio segment. Finally, the All - Push strategy which represents the extreme case where only a single request is required for the video playback, was evaluated. Upon the request of the main HTML file, all the audio and video segments along with the initialisation files are pushed by the server. As a result, the All - Push strategy saved 188 requests. Having said that, it can be concluded that every push strategy was able reduce the total number of requests but the two latter strategies were able to achieve the most significant improvements. Specifically, the Audio - Push strategy saved around 50% and the All - Push strategy saved more than 99% of the request overhead. However, the significant gains of these two strategies in the number of requests come with other consequences which are going to be described in subsequent sections. It is important to note that the reported results are attributed to the 1 second segments duration, but the similar results are applied to any segment duration.

## 5.6 Evaluation of Link Utilisation

In this project, the link utilisation is defined as the ratio of the media throughput to the available bandwidth and the media throughput is defined as the average video quality of the entire playback. The available bandwidth was restricted using Linux Traffic control, to a level sufficient to provide the highest available quality level. A DASH streaming scenario with both segment durations was first evaluated for different RTT configurations, under both HTTP/1.1 and HTTP/2. It is important to note that the bandwidth was restricted to 2Mbits and the highest available quality was 1500 Kbits. Therefore, the highest possible score for the link utilisation is 75 %. The results of the evaluation can be seen in Figure 5-12.

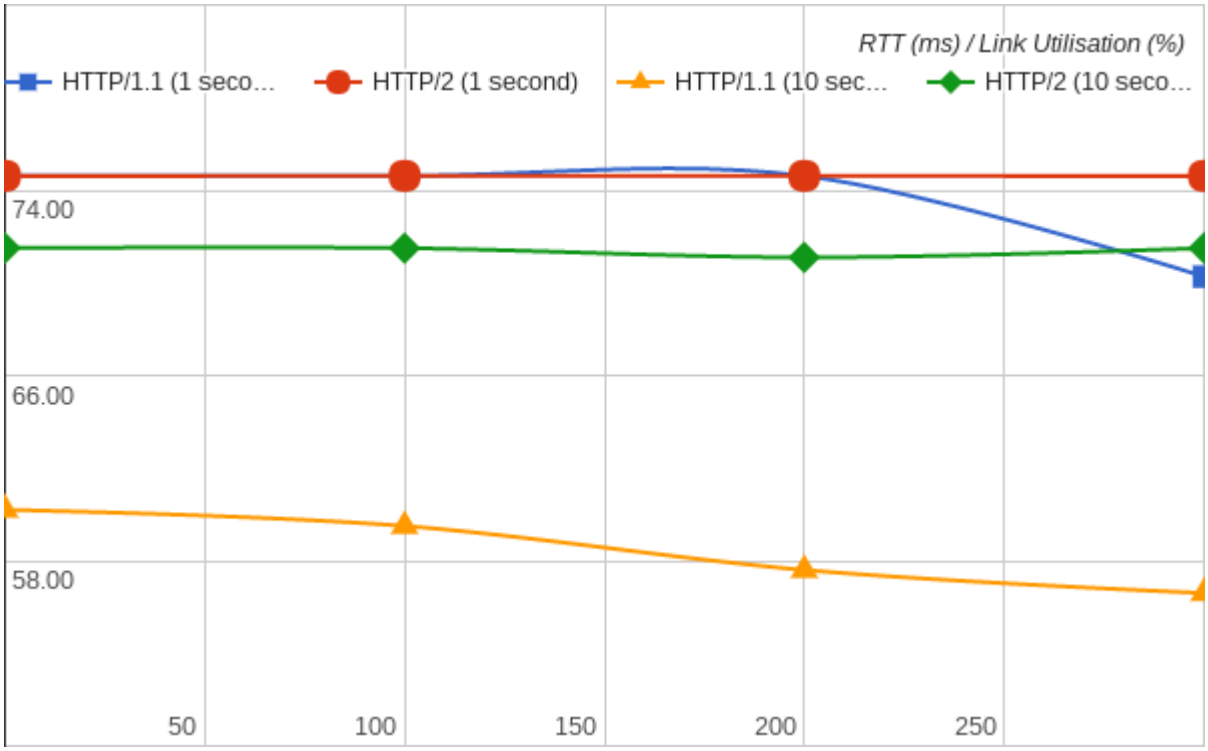


**Figure 5-12: The link utilisation for both HTTP/1.1 and HTTP/2 in a DASH streaming scenario.**

The results above indicate the link utilisation, expressed as a percentage, in terms of HTTP/1.1 and HTTP/2 for segment lengths of 1 and 10 seconds. Based on the reported results, it can be concluded that the segment duration and the underlying protocol definitely impact the media throughput and therefore the link utilisation. As can be seen in the above graph, the link utilisation

is getting worse when the segment size is increased for both protocols. This is due to the long duration of each segment preventing the client to adjust as fast and as flexible as with the 1 second segments. Therefore, the overall bit rate and the media throughput gets worse. Apart from the segment duration, results reported that the underlying protocol also impacts the link utilisation, with HTTP/2 outperforming HTTP/1 in both segments' length. A possible explanation for that is the multiple TCP connections required by HTTP/1.1 to be established on the server. The slow start of TCP prevents the maximum utilisation of the available bandwidth.

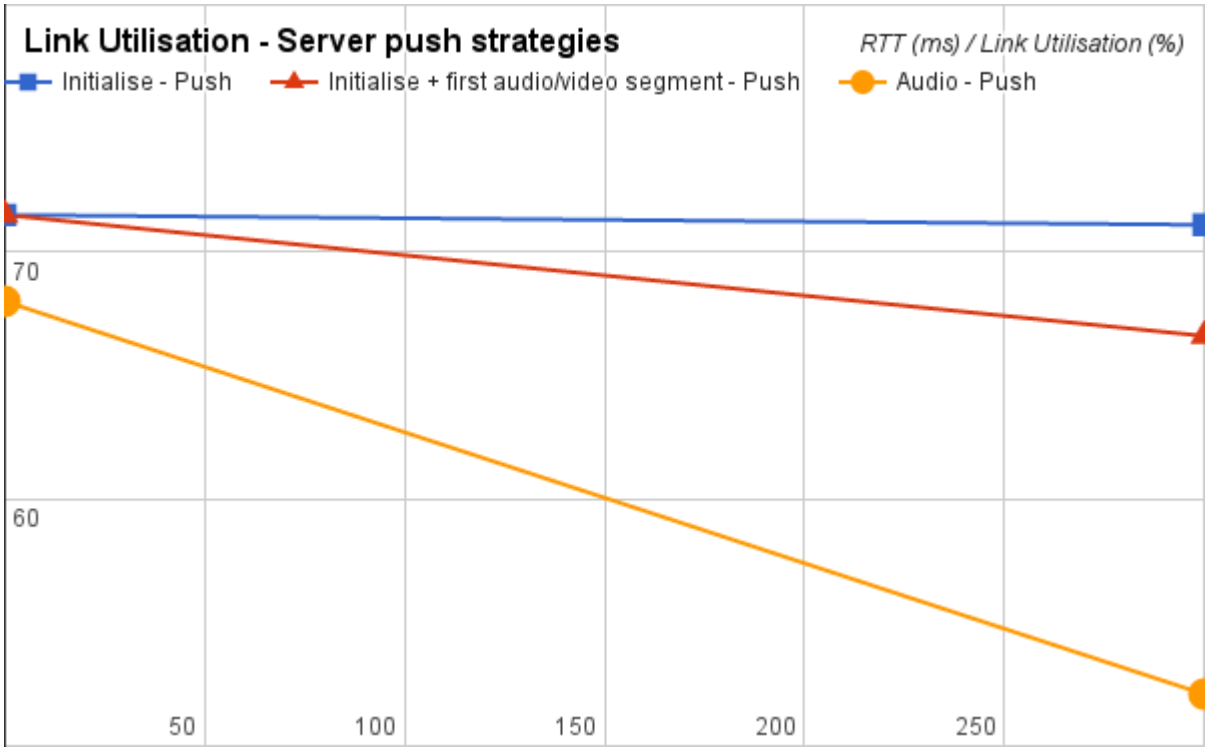


**Figure 5-13: The impact of server push on link utilisation.**

To examine the impact of the server push feature in link utilisation, several strategies were evaluated and their results can be seen in Figure 5-13. The reported results highlight the performance of the different server push strategies with a segment duration of 10 seconds for RTTs of 0 and 300 ms. Due to the small differences, the intermediate RTTs have been omitted. As it can be seen, the Initialise - Push strategy is the only one which can be compared with the traditional HTTP/2 scheme. It is believed that the large number of the pushed resources of the Audio - Push

strategy, is the main reason behind its poor performance, especially for higher RTTs. The large number of pushed assets utilises a big part of the available bandwidth and as a result the client is forced to request segments of lower quality. Based on the poor performance of the Audio - Push, the All - Push strategy was not evaluated in this experiment. A summary of the results for a segment duration of 10 seconds is shown in Table 5-8.

**Table 5-8: Results of the server push strategies on the link utilisation.**

| Link Utilisation - Media Throughput | | | | | |
|---|---|---|---|---|---|
| RTT (ms) | HTTP/1.1 | HTTP/2 | Initialise – Push | Initialise + first audio/video segment - Push | Audio - Push |
| 0 | 60.20 | 71.50 | 71.50 | 71.50 | 68.00 |
| 300 | 56.60 | 71.50 | 71.10 | 66.60 | 52.10 |

## 5.7 Evaluation of bandwidth overhead

Last but not least, the bandwidth overhead introduced by each push strategy was also considered. In this project, it is defined as the overhead generated by all the resources which have been pushed by the server but remain unused. Obviously, the No - Push strategy doesn't introduce any bandwidth overhead since nothing is pushed by the server and therefore the client requests only what it needs. This is also the case for the Initialise - Push strategy. In this approach, the four initialisation files which are essential for the DASH video streaming are pushed by the server. There is no chance that these files are not going to be used by the. Another approach which does not introduce any bandwidth overhead is the Audio - Push strategy. The audio content of the video is encoded into a single bit rate for every representation. As a result, all the segments need to be requested by the client, independent of the video quality. Following these straightforward approaches, the Initialise + first audio/video segment - Push strategy is a little more complex. This is because there are cases where the strategy may introduce some bandwidth overhead but there

are also cases where it may not. More specifically, this approach pushes the first video segment of the medium quality representation along with the first audio segment. Therefore, if the medium quality segment is the one which is about to be used, no additional overhead is introduced. However, if for any reason a higher or lower quality segment is about to be used, the pushed resource remains unused and therefore bandwidth overhead is introduced. The audio segment doesn't introduce any extra bandwidth since it is always needed by the client. Finally, the All - Push strategy is the strategy where all the available resources are pushed by the server as soon as the initial request is made. Based on the adaptive behaviour of DASH, this strategy introduces a significant overhead. Having pushed all the segments of a predefined quality level, the only option for the client to switch to a different bit rate is to issue a new request for the new quality level and as a result obtain all the segments of that quality. It can easily be concluded that this approach is likely to introduce huge bandwidth overheads which are not desirable in an adaptive streaming scenario.

# 6 Conclusion

In this paper, an extensive evaluation was conducted in order to examine the performance of the server push feature of HTTP/2 on DASH streaming scenarios. This project utilised Apache, the most popular web server available, and to the best of our knowledge this is the first study which attempted to evaluate the server push feature on Apache. To achieve that, numerous server push strategies were implemented and each one of them was evaluated in terms of DOM and user-perceived loading times, total number of requests required to playback the entire media content, link utilisation, initial start-up delay and bandwidth overhead. The evaluation also highlighted the impact of segments duration in the performance of a DASH video streaming. Based on the various experiments and the reported results, a summary of the performance of each server push strategy is given below:

- No - Push Strategy: An extreme case which was used as a baseline for comparison. It is equivalent to the traditional HTTP scheme. Regarding DOM and user - perceived page loading times, its performance was really similar with HTTP/1.1 and was outperformed by all the applied server push strategies. The segment duration didn't have any particular impact on the loading times since both durations achieved similar results. On the other hand, the initial start - up delay was highly dependent on the segment duration since the segments of 1 second managed to start playing the first video segment of the media content 4 seconds faster than the 10 second segments. Furthermore, based on the fact that this strategy individually requests each video and audio segment, the total number of requests needed is equal to the highest possible. Moreover, the No - Push strategy managed to score the highest link utilisation, compared to all the applied server push strategies, when the

segment duration was set to 1 second. Finally, it does not introduce any bandwidth overhead.

- All-Push Strategy: Using this strategy, a single request is all that is needed in order to playback the entire media content. Every audio and video segment, along with the initialisation files of the streaming, are pushed to the client at the beginning of the session. However, the huge optimisation of the number of requests and the maintaining of comparable page loading times, comes with additional cost. The introduced overhead in the adaptive bitrate switching is a major problem of this strategy since a client can switch to a different bit rate only by issuing a new request for the segments with new bit rates. Moreover, evaluation indicated that the large amount of pushed resources result in poor start-up delay and low link utilisation. Finally, based on the fact that the adaptive behaviour is the main characteristic of HTTP streaming, it was concluded that this strategy is not efficient for adaptive streaming scenarios.
- Initialise - Push Strategy: Managed to improve DOM and user - perceived page loading times significantly compared to HTTP/1.1 and raw HTTP/2, for both segment durations. It is important to note that all the applied server push strategies reported similar performance regarding loading times. The Initialise – Push strategy was also able to minimise the start-up delay by 1% and save a total of 4 requests. Furthermore, it was the only server push strategy which was comparable with the raw HTTP/2 in terms of link utilisation. Last but not least, no additional overhead was introduced by this strategy since it is configured to push all the initialisation files of DASH which are essential for the streaming.
- Initialise + first audio/video segment - Push Strategy: In this approach, apart from the initialisation files, the first video segment of the medium quality representation along with the corresponding audio segment are also pushed. Based on the results, this strategy achieved the lowest start up delay compared to all the server push strategies. Additionally, it saved a total of 6 requests and its link utilisation was slightly lower than the Initialise -

Push approach, but still relatively high. However, it may introduce additional overhead in the case where the medium quality representation is not selected for the first video segment.

- Audio - Push Strategy: Similar to the All - Push strategy, the large amount of pushed resources has a negative impact on link utilisation and start-up delay. However, this approach managed to save around 50 % of the total number of requests, due to the fact that every audio segment is pushed at the beginning of the connection. Moreover, there is no additional bandwidth overhead introduced since each pushed resource is essential for the streaming session.

The various experiments indicated that the newly developed HTTP/2 protocol can have a positive impact on the delivery of media content. Furthermore, by leveraging the server push feature, the performance of adaptive streaming scenarios can be further improved. More specifically, the evaluation of this study reported significant gains in terms of start-up delay, link utilisation, DOM content loading time and user - perceived latency. The various server push mechanisms indicated that the selection of the most appropriate strategy is crucial to a DASH application. As a future work, additional push mechanisms will be implemented and evaluated in order to find the most optimal one. Moreover, the expansion of the experimental setup and the addition of extra nodes for the bandwidth restrictions and RTT configurations may lead to more optimal results. Furthermore, the impact of segment duration will be further evaluated with the addition of different segment's lengths. Finally, the project will be further expanded to include an evaluation of the server push strategies on top of the experimental QUIC protocol.

Apart from the extensive evaluation, this project also aimed to contribute to the improvement of the current state of the art, regarding the Apache web server and the new HTTP/2 protocol. To achieve that, a web interface was implemented in order to facilitate the way that Apache web server configures any new pushed resources. Currently, full access to the Apache server is required in

order for the user to be able to update the pushed assets of a response. The implemented web interface managed to address this issue by automating the overall procedure. It is believed that this is the first software implementation which attempts to address this issue. The future work will include the support of DASH streaming by allowing the user to select specific segments to be pushed.

# 7 References

*A Simple Performance Comparison of HTTPS, SPDY and HTTP/2 | HttpWatch BlogHttpWatch Blog*. (2016). *Blog.httpwatch.com*. Retrieved 25 August 2016, from http://blog.httpwatch.com/2015/01/16/a-simple-performance-comparison-of-https-spdy-and-http2/comment-page-1/.

*Adaptive Bitrate Streaming*. (2016). *Encoding.com*. Retrieved 25 August 2016, from http://www.encoding.com/packaging/.

*Announcing Support for HTTP/2 Server Push*. (2016). *CloudFlare*. Retrieved 25 August 2016, from https://blog.cloudflare.com/announcing-support-for-http-2-server-push-2/.

Belshe, M., & Peon, R. *M. Thomson, Ed.," Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540, DOI 10.17487/RFC7540, May 2015,< http://www. rfc-editor. org/info/rfc7540.

*Chrome Browser*. (2016). *Google.com*. Retrieved 25 August 2016, from https://www.google.com/chrome/browser/canary.html.

*ChromeDriver - WebDriver for Chrome*. (2016). *Sites.google.com*. Retrieved 25 August 2016, from https://sites.google.com/a/chromium.org/chromedriver/.

*Conviva Releases 2014 Viewer Experience Report - Conviva*. (2014). *Conviva*. Retrieved 25 August 2016, from http://www.conviva.com/conviva-releases-2014-viewer-experience-report/.

*Dash-Industry-Forum/dash.js*. (2016). *GitHub*. Retrieved 25 August 2016, from https://github.com/Dash-Industry-Forum/dash.js/wiki.

*Dynamic Adaptive Streaming over HTTP (MPEG-DASH)*. (2016). *Encoding.com*. Retrieved 25 August 2016, from http://www.encoding.com/mpeg-dash/.

*Encode/AAC – FFmpeg*. (2016). *Trac.ffmpeg.org*. Retrieved 25 August 2016, from https://trac.ffmpeg.org/wiki/Encode/AAC.

*FFmpeg*. (2016). *Ffmpeg.org*. Retrieved 25 August 2016, from https://ffmpeg.org/.

Grigorik, I. (2013). *High Performance Browser Networking: What every web developer should know about networking and web performance*. " O'Reilly Media, Inc.".

Grigorik, I. (2013). Making the web faster with HTTP 2.0. *Communications of the ACM*, *56*(12), 42-49.

Group, D. (2016). *Welcome! - The Apache HTTP Server Project*. *Httpd.apache.org*. Retrieved 25 August 2016, from https://httpd.apache.org/.

*How to encode Multi-bitrate videos in MPEG-DASH for MSE players (1/2)*. (2014). *Streamroot Blog*. Retrieved 25 August 2016, from https://blog.streamroot.io/encode-multi-bitrate-videos-mpeg-dash-mse-based-media-players/.

*HTTrack Website Copier - Free Software Offline Browser (GNU GPL)*. (2016). *Httrack.com*. Retrieved 25 August 2016, from https://www.httrack.com/page/1/en/index.html.

Hubert, B. (2002). Linux advanced routing & traffic control HOWTO. *setembro de*.

Huysegems, R., van der Hooft, J., Bostoen, T., Rondao Alface, P., Petrangeli, S., Wauters, T., & De Turck, F. (2015, October). Http/2-based methods to improve the live experience of adaptive streaming. In *Proceedings of the 23rd ACM international conference on Multimedia* (pp. 541-550). ACM.

Index, C. V. N. (2016). Forecast and methodology, 2015-2020 white paper.*Retrieved 30th July*.

*Jetty - Servlet Engine and Http Server*. (2016). *Eclipse.org*. Retrieved 25 August 2016, from http://www.eclipse.org/jetty/.

Kim, H., Lee, J., Park, I., Kim, H., Yi, D. H., & Hur, T. (2015, August). The upcoming new standard HTTP/2 and its impact on multi-domain websites. In*Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific* (pp. 530-533). IEEE.

*mod_http2 - Apache HTTP Server Version 2.4*. (2016). *Httpd.apache.org*. Retrieved 25 August 2016, from https://httpd.apache.org/docs/2.4/mod/mod_http2.html.

Monchamp, T. D. (2013). Adaptive bit-rate streaming. *InSight: Rivier Academic Journal*, *9*(2).

*MP4Box | GPAC*. (2016). *Gpac.wp.mines-telecom.fr*. Retrieved 25 August 2016, from https://gpac.wp.mines-telecom.fr/mp4box/.

Mu, M., Trench-Jellicoe, J., & Race, N. (2014). Vision social TV: towards personalised media experience and community atmosphere.

Mueller, C., Lederer, S., Timmerer, C., & Hellwagner, H. (2013, July). Dynamic adaptive streaming over HTTP/2.0. In *2013 IEEE International Conference on Multimedia and Expo (ICME)* (pp. 1-6). IEEE.

Network Working Group. (1999). RFC 2616: Hypertext Transfer Protocol--HTTP/1.1. *R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee*.

*networking:netem [Linux Foundation Wiki]*. (2016). *Wiki.linuxfoundation.org*. Retrieved 25 August 2016, from https://wiki.linuxfoundation.org/networking/netem.

Nguyen, D. V., Le, H. T., Nam, P. N., Pham, A. T., & Thang, T. C. (2016, January). Request adaptation for adaptive streaming over HTTP/2. In *2016 IEEE International Conference on Consumer Electronics (ICCE)* (pp. 189-191). IEEE.

Padhye, J., & Nielsen, H. F. (2012). A comparison of SPDY and HTTP performance. *Microsoft Res*.

*Performance Testing Results of Adaptive Media Streaming over HTTP/2 - BBC R&D*. (2016). *Bbc.co.uk*. Retrieved 25 August 2016, from http://www.bbc.co.uk/rd/blog/2015-07-performance-testing-results-of-adaptive-media-streaming-over-http.

Petrangeli, S., van der Hooft, J., Wauters, T., Huysegems, R., Alface, P. R., Bostoen, T., & De Turck, F. (2016, May). Live streaming of 4K ultra-high definition video over the internet. In *Proceedings of the 7th International Conference on Multimedia Systems* (p. 27). ACM.

Sodagar, I. (2011). Industry and standards.

Stenberg, D. (2014). HTTP2 explained. *Computer Communication Review*,*44*(3), 120-128.

Stockhammer, T. (2011, February). Dynamic adaptive streaming over HTTP--: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems* (pp. 133-144). ACM.

Timmerer, C., & Bertoni, A. (2016). Advanced Transport Options for the Dynamic Adaptive Streaming over HTTP. *arXiv preprint arXiv:1606.00264*.

*W3C Document Object Model*. (2009). *W3.org*. Retrieved 25 August 2016, from https://www.w3.org/DOM/.

Wang, Z., & Jain, A. (2013). Navigation timing.

Wei, S., & Swaminathan, V. (2014, March). Low latency live video streaming over http 2.0. In *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop* (p. 37). ACM.

Wei, S., & Swaminathan, V. (2014, September). Cost effective video streaming using server push over HTTP 2.0. In *Multimedia Signal Processing (MMSP), 2014 IEEE 16th International Workshop on* (pp. 1-5). IEEE.

Wei, S., Swaminathan, V., & Xiao, M. (2015, October). Power efficient mobile video streaming using HTTP/2 server push. In *Multimedia Signal Processing (MMSP), 2015 IEEE 17th International Workshop on* (pp. 1-6). IEEE.

*XMLHttpRequest*. (2016). *Mozilla Developer Network*. Retrieved 25 August 2016, from https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.