UNIVERSITY OF AMSTERDAM

Impacts of the HTTP/2 protocol for large scale web environments

Martin Leucht, James Gratchoff
Master SNE
University of Amsterdam

March 31, 2015

Abstract

This paper compares version 1.1 and version 2 of the HTTP protocol. Both versions have been tested with TLS enabled. The main purpose is to show differences in terms of latency, header size, bandwidth utilisation and server performance between both protocol versions. In order to measure and compare the characteristics of the protocols, a benchmark setup has been developed. On server side different webserver implementaions, were deployed in order to serve both protocol versions. HTTP clients are located around the world to represent different RTTs. The tool used for the protocol benchmarking is h2load. A script that calls h2load and is capable to provide our measurement requirements was developed. The results identify a clear improvement of the HTTP/2 protocol in terms of latency times and decreasing header sizes. Moreover the server performance will benefit from improvements that have been introduced in HTTP/2.

Contents

1 Introduction

Specified by the IETF in 1999 in multiple RFCs(7230-7235[1]), HTTP/1.1 is a standard protocol for web-browsing. HTTP is the essence of the web and most of the users are using this protocol on a daily basis. After more than 15 years of use it was time for a change. All the workarounds not to slow the HTTP/1.1 protocol have to be forgotten as on February the 18th 2015, the specification [2] for the new HTTP protocol HTTP/2 (and HPACK[3]), has been formally approved by the IESG and is on the way to become an RFC standard.
The main focus during the development period of the successor of HTTP/1.1 was to improve the performance, and thus provide a better user experience. The performance improvement is based on how the packets are sent over the wire within a HTTP/2 session. HTTP/2 data is sent in binary format and instead of creating a single TCP session per element retrieved, HTTP/2 use TCP streams that are multiplexed and can be prioritized. The purpose of this technique is to reduce to one the number of TCP sessions created every time a user access a web page. Concerning security, HTTP/2 will not make the use of TLS mandatory. However, leading browsers firms, such as Mozilla Firefox and Google Chrome have already mentioned that HTTP/2 will only be implemented over TLS. Thus ensuring that the data will be encrypted between two end parties. HTTP/2 introduces several other new features that will be presented in this paper.
There are already web client and server implementations[17] that support the final HTTP/2 specification. This research is intending to show what are the impacts of implementing HTTP/2 with TLS compared to HTTPS in a large environment. A benchmark will be performed on a physical web server that will be tested with a high number of virtual clients located at different places around the world to outline the impact of the round trip time (RTT) on both version of the protocol and to see which performs the best. The research will draw conclusions on what impacts does the new protocol can have on large web service providers infrastructure.

2 Research Questions

To conduct the research the following research questions have been formulated:

How do the new features of the HTTP/2 protocol improve the performance for frequently visited web-pages/webserver?

That question can be narrowed down to some sub questions:

- What are possible drawbacks that can occur for large web service providers when switching from HTTP/1.1 to HTTP/2 ?
- What is the predictable impact that could be related to changes in the infrastructure of Web service providers?
- What is the difference in bandwidth utilization between HTTP/1.1 and HTTP/2 considering the size ratio of header and data for multiple concurrent sessions on a server?
- How much can HTTP/2 decrease the amount of data when considering header compression and using multiple streams within one TCP Session compared to HTTP/1.1 on a server ?
- What is the impact regarding the response time experienced by a client when the number of concurrent requests / clients increases seen from three different geographical locations?
- What is the impact to the load and CPU utilization to the server when conducting the benchmark tests?

3 Literature review

The creation of the HTTP/2 protocol have been discussed for many years. This section discusses the drawbacks of the previous version and presents the reasons why to move to a new protocol. The features that will improve the performance of the HTTP/2 protocol are also introduced. And finally the related work section will present the benchmarks that have been done for the SPDY and HTTP/2 protocol. Information has been derived from the HTTP/2 draft[2] and from Daniel Stenberg paper explaining HTTP/2[4].

3.1 HTTP/1.1 drawbacks

The HTTP/1.1 protocol uses a client-server model where the clients is most of the time a browser that is getting data from a server located in a different location. Since the early beginning of the protocol, users complained about the time that takes a page to load. In the early days one major reason of this problem was not due to the protocol but to the speed and the reliability of the Internet. However the web has changed and more and more users have a high speed and reliable connection to the Internet. This and the growth of clients on the web has led to the increase of web pages size and the number of elements on a web page. HTTP was not designed for this use as indeed it is inadequately using the TCP protocol by retrieving one element from a web page using one TCP connection. It is thus not taking full advantage of the high performance TCP protocol as indeed the current use of TCP in HTTP/1.1 protocol can introduce problems that are leading to a slower loading time. These problems are for example the head of line blocking or the repetition of headers between multiple TCP sessions. The loading time is also highly dependent on the Round Trip Time (RTT) and some efforts have been done on improving this aspects over the years by adding geographical redundancy however the problem is still present for low cost organisation that are only able to develop server in a single location. This latency directly affect the clients and can be critical for any website (e.g. users leaving the page as it takes too long to load). So over the years, web developers have tried to reduce this critical factor that is latency. In order to do so they have tried to adapt themselves to the protocol with workarounds that reduce the number of TCP connection. The most known workarounds are:

- Spriting: The fact to create an image containing many pictures destined to be present on the website and let the browsers display (via CSS or Javascript) the picture wanted by cropping/cutting out a single image.
- Sharding: In order to overcome the problem of increasing TCP connections per domain, developers started to create several domains, holding different part of the website. This decreases the page loading time by reducing the number of connection per domain. Thus leading to a better performance of the HTTP/1.1 protocol. This method can also be used to allow more TCP sessions as each domains are bounded to a certain number.
- Concatenation: In order to reduce the number of TCP connections, developers started to concatenate files (e.g. javascript) into one big files.
- Inlining: By embedding the data straight in the CSS in base64 format it avoids to send picture and thus creating new TCP connections.

All these workarounds were needed as the page loading time and the number of requests were increasing so much that the web was slowing down even though more and more people had access to a reliable connection. That is why the IETF created a working group named HTTPbis[5] that started working on a new protocol. Beforehand Google started working on a new protocol, called SPDY[6], that was aiming to encounters the problems from the HTTP/1.1 protocol. The HTTPbis group started working from a working concept of protocol in the name of SPDY/3 (draft). And this was the start of HTTP/2.

3.2 HTTP/2 improvements

This section is subdivided into several ones describing the major improvements made from its predecessor HTTP/1.1 and emphasis on the reasons to move to the new HTTP/2 protocol for large scale environments.

3.2.1 Binary format

HTTP/1.1 is based on a text/ascii format which is an advantage for humans to read and thus to debug the protocol. It is described by Raymond as "easy for human beings to read, write, and edit without specialized tools"[7]. However for computers such as clients and server, ascii is not their mother tongue. Indeed computers are using binary as a format for exchange. HTTP/2 is using this format. "HTTP/2 also enables more efficient processing of messages through use of binary message framing."[2] Binary is known to be much more efficient for binary structures. It is indeed hard to define the start and the end of a field in text based protocols. However with binary format it is much more natural. Binary will then improve the structure of the protocol and thus the efficiency of the protocol. In order to overcome the difficulty of debugging the binary protocol, tools such as curl[8] have added support for HTTP/2 and Wireshark have created extensions[9] to decode the network streams.

3.2.2 Multiplexing and priority

The use of streams is a major enhancement of the HTTP/2 protocol. A stream is described in the specification as "an independent, bi-directional sequence of frames exchanged between the client and server within an HTTP/2 connection."[2] The stream identifier, present in the header format as an integer, will associate each frame belonging to the same stream. One HTTP/2 connection can contain several concurrently open stream, that can each be closed by the client or the server. Streams are multiplexed which can mean that they do not arrive at the same order as they have been sent. It is the role of the client to put this streams back together in a correct order to process the data. Each stream has a priority that can be set by the client in the HEADERS frame that opens the stream. This priority can be changed to re-prioritize a specific stream. This can permit to allow an endpoint to express how it would prefer to retrieve data when managing multiple concurrent streams. A stream can also be dependent on other streams by setting the stream dependency parameter.

3.2.3 Header compression

One of the problem of HTTP/1.1 described earlier is that more and more elements are retrieved per web page. If too elements are close in type and location the headers will be very similar and "thus the redundant header fields in these requests unnecessarily consume bandwidth, measurably increasing latency. "[3] That is why header compression is a good solution to this problem. However HTTPS and the SPDY compression mechanism have been vulnerable to the BREACH[10] and CRIME[11] attacks. That is why the HTTPbis group has created a compression format HPACK[3]. It is used to represents efficiently HTTP header fields, to be used in HTTP/2. HPACK is described as "simple and inflexible."[3] It eliminates redundant header fields and prevent security issues.

3.2.4 Flow control

The flow control is implemented in HTTP/2 by assigning an integer value to a window that will define how many octets of data the sender (server) is able to transmit. In that way the server is taking in consideration the buffering capacity of the receiver. This has enormous advantages as receivers on the web can be of different nature (e.g.Smartphones, laptop) with different connection. Flow control control can either operate on the entire connection or on each individual stream. The default value is 65535 octets. In order to reassign this value the receiver (client) can send a SETTINGS frame with his flow control integer value.

3.2.5 Server push

HTTP/2 allows the server to send extra information with a requested information required by a client. This functionality permits to increased the overall loading time by anticipating what the client will need before it asks for it. In that way the client will already possess the information when asking for it. This is called the server push mechanism. This mechanism is not required but can improve the user experience. The client must allow the server to do so. This allow the client to stay in control and indeed the client is able to terminate that pushed stream by sending a RST_STREAM.

3.3 Related work

The new HTTP2 protocol has been based on the SPDY protocol developed mainly by Google. As shown by Google [12], the SPDY protocol meets its expectations by reducing the loading time of web pages by 55%. Other people have tried to look into the protocol and one of the most interesting analysis has been done by Servy[13]. Servy evaluated the performance of the web servers implementing the SPDY protocol comparing it to HTTP/1.1 and HTTPS. The load testing tool used for this benchmark was the NeoLoad 4.1.2. His results showed that the implementation of SPDY increases by a factor of 6 the number concurrent of users possible before errors start showing up in comparison to HTTP and HTTPS. A contradictory study showing some boundaries of implementing SPDY has been done by Podjarny[14]. He shows that most of the websites use different domains and as SPDY works on a per-domain basis it does not necessarily help it to be faster. Finally, Wang et al.[15] have investigated the performance of SPDY for the improvements of the protocol compared to HTTP/1.1. This study highlights that SPDY is much faster since its benefits from the single TCP connection mechanism. However, they also mention that SPDY degrades under high packet loss compared to HTTP. Concerning the new standard HTTP/2 a few benchmarks have been performed by the creators of different client/server platforms. They reach the same conclusion for SPDY.
However, studies on comparisons between HTTP/2 (draft-ietf-httpbis-http2-14 [18]) and HTTP/1.1 with regards to concurrent clients and increasing amount of requests in different geographical locations and different page sizes or amount of elements a web page contains, have not been conducted yet.

3.4 Conclusion

To conclude, this section presented the reasons that have lead to the design of a new version of the HTTP protocol. The TCP protocol have been implemented in the HTTP protocol in the early days of the web however due to the evolution of the web the protocol started lacking and the principal reason was that the use of the TCP protocol was not suiting the web usage any more. HTTP/2 address these issues by exploiting in a better way the TCP protocol. It also creates new features that will redesign the way to implement large infrastructure such as flow control, stream priority or server push. The new performance of the protocol have been tested by the designers of the implementation of the protocol but not by other parties. That is why this research has been conducted.

4 Scope

The scope of this research is to conduct benchmarks of the HTTP/1.1 and HTTP/2 protocol, using different static and dynamic parameters and compare them among each other. Static parameters are the distance between client/server (RTT) and the page size, whereas dynamic parameters are the number of requests and the number of concurrent clients. Therefore we will implement two different HTML pages with different sizes (number of URLs included) and benchmark them from three different geographical locations. The tests will be implemented using the secured version of the protocols with TLS enabled. We do not focus on implementing the new features of the HTTP/2 protocol like server push capabilities.

5 Approach and Methods

5.1 Measurement

5.1.1 Measurement Parameter

To perform various measurements that reflect realistic scenarios a large web content provider will encounter, a topology consisting of several components was composed. As HTTP client machines, Amazon AWS microinstance VMs [16] located at different geographical locations (Europe, North America and Asia) are used. Measurements taken from clients located in different geographical areas ensure to have different magnitudes of Round Trip Times towards the destination web server located in Amsterdam. On server side two TLS enabled webserver instances are running on one physical machnine in order to conduct measurements for both protocols - HTTP/1.1 and HTTP/2. Two reference HTML pages of different sizes and different numbers of statically linked resources have been created to be compared. These pages reflect the sizes that can be encountered in most common websites and has been derived from top 100 websites statistics [23]. The appropriate web page sizes including the number of referenced resources (URLs) are listed in Table 1.

Web Page	Size (kB)	Number of URLs included
small.html	20	2
large.html	1600	54

Table 1: different Web Pages

So far, we have defined different static parameters that will be applied while performing the measurement. These parameters include different Round Trip Times (RTTs) and different web page sizes. To simulate varying and realistic load scenarios on the server, the number of clients and thereby the number of requests towards the web servers is incremented during each test starting from one client upto maximal 750 clients in parallel.

5.1.2 Measurement Tools

It is essential to choose and implement the right tool and method to measure the response/request times characteristics of the HTTP/1.1 and HTTP/2 protocol and to be able to compare them with each other. Thus, the benchmarking tool was chosen carefully and with respect to retrieve accurate measurement data that can be used to compare both HTTP versions with each other. H2load [21] was elected as the best candidate for that purpose. It has been developed to run benchmarking tests against HTTP/2 enabled web servers. It can also be used to conduct measurements against HTTP/1.1 enabled web servers if, like in that case, a HTTP/2 - HTTP/1.1 reverse proxy is used that is capable to translate HTTP/2 requests into HTTP/1.1 requests on server side. It is possible to start h2load with a file option, that includes a list of URIs that each measurement will request in parallel. H2load returns after each measurement round accurate and valuable data, like the retrieved amount of header and body data in bytes per request. Furthermore it returns the minimum, maximum and mean round trip time for each web page request and the actual performance in requests/second.

5.1.3 Measurement Methods

In order to collect statistical analysable data, all measurements are performed with twenty repetitions. A wrapper script was created that performs the measurements using h2load, does mean calculations on the data and writes them into a file. The data is later processed for visualizing round trip times and other measurements. The wrapper script can be found at the end of that report as part of the appendices 8.
It is important to notice that the main difference between HTTP/1.1 and HTTP/2 sessions is based on

the number of created TCP sessions on server side. A HTTP/1.1 client (e.g. web browser) opens for each HTML link that is present in a HTML web page a separate TCP session to the web server. In contrast to HTTP/1.1, the new HTTP/2 protocol opens one single TCP session towards the server and multiplexes all requested data over multiple streams within a single TCP session. Thus, the maximum number of concurrent streams for each HTTP/2 session is set equal to the amount of requested URIs in all measurements. Figure 1 shows HTTP/1.1 GET requests in order to get the entire set of resources the reference web page (large.html) includes.

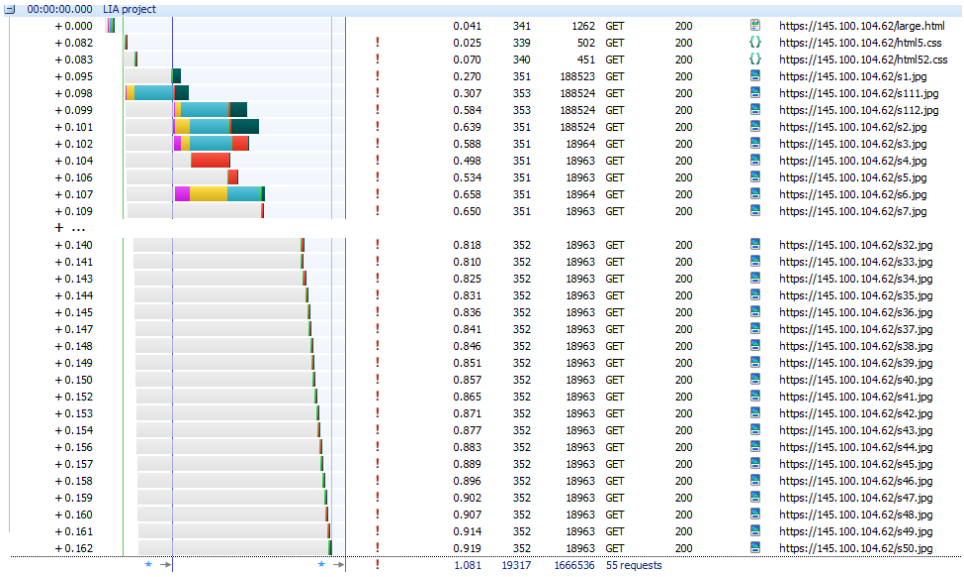

Figure 1: HTTP/1.1 GET requests for large.html

HTTP/2 handles all GET requests in a single TCP connection by using multiple streams s, thus the expected number of TCP connections t will be equal to the number of concurrent simulated clients n. For HTTP/1.1 we expect the number of TCP connection t as a function of the product of concurrent clients n by the requested number of HTML links l. Thus we can describe the number of TCP sessions l for HTTP/1.1 as:

$$t = f(n) = n * l \qquad \text{and} \qquad l \geq 1 \tag{1}$$

and for HTTP/2 as:

$$t = f(n) = n \qquad \text{and} \qquad s = l; l \geq 1; s \in n \tag{2}$$

That results in Table 2 which shows the expected numbers of TCP connections towards the web server during some stages in the measurements with an assumed number of l=*55* HTML links per page request.

clients/requests n	**t (HTTP/2)**	**t (HTTP/1.1)**
1	1	55
10	10	550
50	50	2750
150	150	8250
300	300	16500
500	500	27500
750	750	41250

Table 2: Number of TCP connections HTTP/1.1 and HTTP/2

5.2 HTTP Server and Clients

HTTP clients that conduct measurements from long distances are deployed using Amazon t2.micro instances. T2 instances provide a baseline level of CPU performance that is comparable to 2 vCPUs with the ability to burst above the baseline level [25]. For local tests within the OS3 network, XEN VM are used. Each client got at least 2 vCPUs and 2 Gigabyte RAM assigned. The clients are located in different geographical areas, resulting in different RTT times between clients and server. On each client h2load was compiled and the wrapper script uploaded. Table 3 shows all clients and their corresponding average round trip times to the server.

Location	**RTT in ms**
Tokyo/Asia	280
North Carolina/North America	150
Frankfurt am Main/Europe	7
Amsterdam/Europe (local)	0.3

Table 3: RTT (in ms) per location

The server that provides access for the HTTP clients is not virtualized. It has 8 Intel Xeon CPUs (1,87GHz) and 8GB RAM installed. The web server has a public IPv4 address and is connected to the public internet via a 1 Gbit Ethernet interface. As HTTP/2 server nghttpd [19] is used which listens on TCP port 8881. For HTTP/1.1 requests Apache2 [20] in combination with nghttpx [26] is used. Nghttpx is a reverse proxy and accepts HTTP/2, SPDY and HTTP/1.1 over SSL/TLS on TCP port 8443 via its front-end. The protocol to the back-end is HTTP/1.1. The usage of an reverse proxy enables us to use our measurement tools without modification, although native HTTP/1.1 requests instead of using a reverse proxy would probably result in more accurate measurements. Since our time was very limited, we decided us to go for the reverse proxy option.

6 Results analysis

In that section the measurement results are presented and analysed. First the HTTP header sizes for HTTP/2 and HTTP/1.1 are compared against each other. Second, the results of the RTT measurements and the actual performed rate of requests per second with a growing number of concurrent clients are presented and discussed for each measurement separately. Finally an overview of the server utilization during the measurements is given for network bandwidth, CPU utilization and TCP sockets.

6.1 HTTP header size

HTTP/2 uses HPACK [3] as a header compression mechanism in order to decrease the amount of data for repetitive HTTP header information. Figure 2 shows the difference between HTTP/2 and HTTP/1.1 regarding the amount of header depending on the amount of clients.

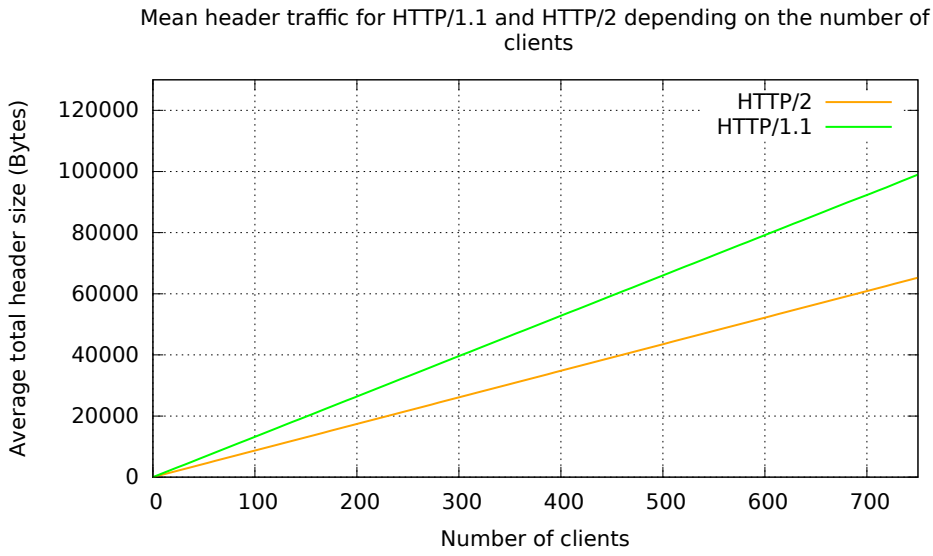

Figure 2: header traffic in bytes for HTTP1.1 and HTTP/2

The result is a linear graph for both versions with a higher rate of growth for HTTP/1.1. It turns out that in our measurement set-up the amount of traffic produced to transmit HTTP headers is approximately 30% lower for HTTP/2 compared to HTTP/1.1.

6.2 HTTP Round Trip Times and Request Rates

In that section the RTT time seen from different locations with a growing number of clients is presented. Furthermore the performance in requests per second is shown. The measurements are described by location ordered by a decreasing distance to the server

6.2.1 Tokyo/Asia

The measurements in Asia are performed with the highest base RTT of 280 ms. The graph in Figure 3 represents the data for 4 measurements by using for each HTTP version two different pages (20kB, 1600kB). The number of clients is incremented by one from 1 upto 750, while performing the measurements.

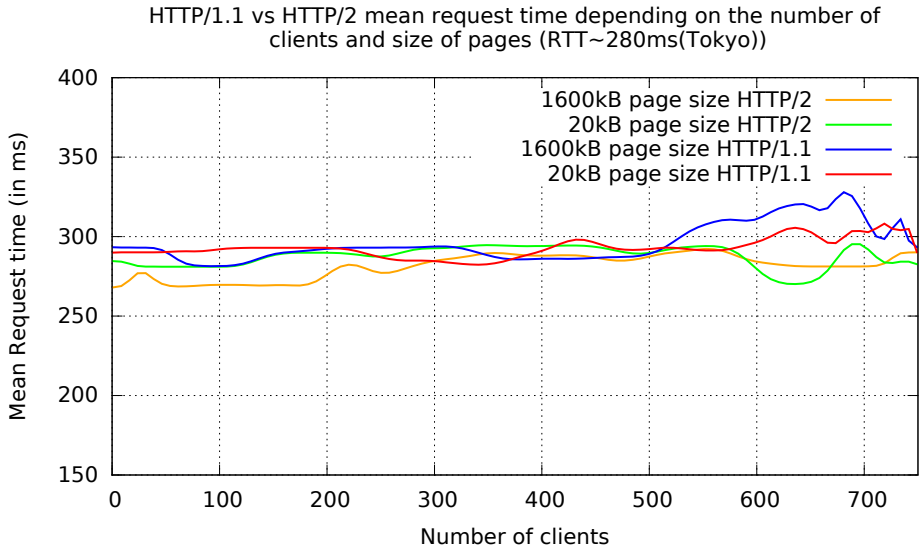

Figure 3: Mean RTT measurements for Tokyo/Asia

A significant difference between both HTTP versions cannot be identified, although HTTP/2 is a few milliseconds faster during the measurements. The graph shows nearly a constant response time regardless if the number of clients increases. That can be explained by considering the rate of requests per seconds seen from the client in Tokyo. As more clients are added as more requests are being executed in approximately the same time. The graph in Figure 4 shows that dependency.

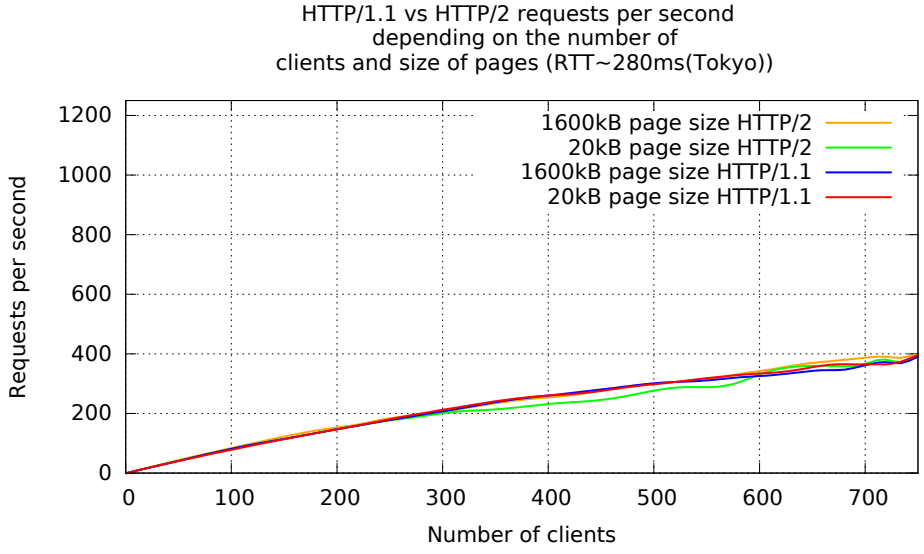

Figure 4: Mean Requests per second for Tokyo/Asia

6.2.2 North Carolina/North America

The next measurements were taken from a client in North America/North Carolina with a measured base average RTT of 150ms and is shown in Figure 5.

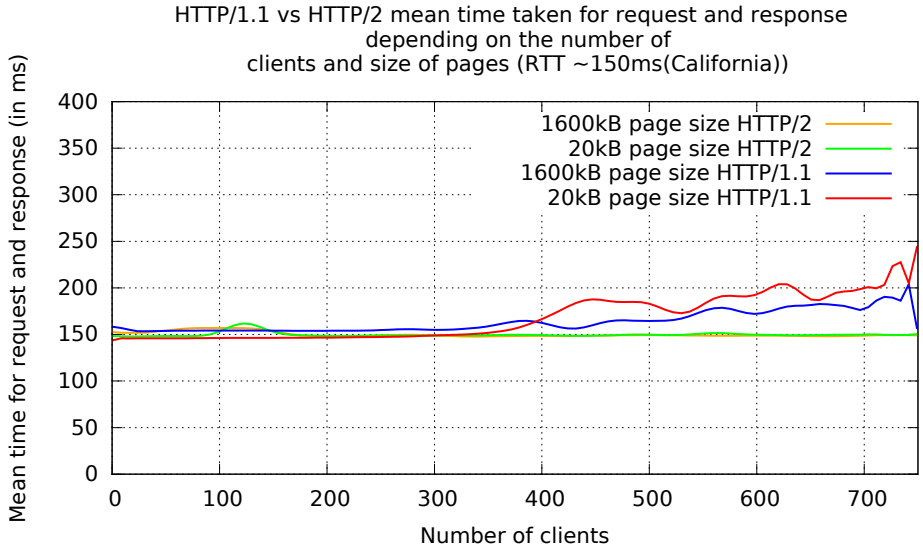

Figure 5: Mean RTT measurements for North Carolina/US

The graph shows for HTTP/2 measurements, that the HTTP RTT is nearly almost equal to the measured base RTT. That means there is almost no jitter measurable and the RTT is constant for HTTP/2 requests regardless if the number of clients is raised or not. Moreover we see a growing HTTP/1.1 RTT starting from approximately 400 concurrent clients upwards. Figure 6 shows the corresponding request per second rate graph.

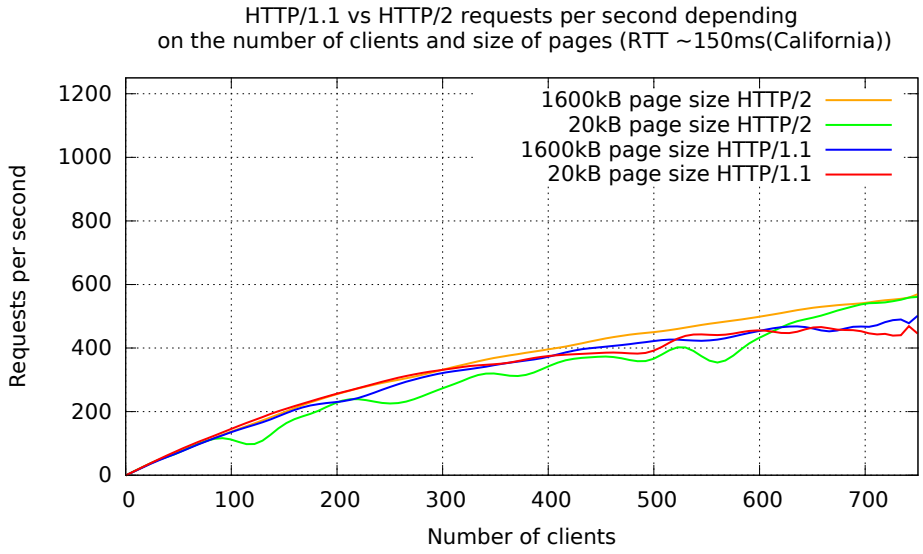

Figure 6: Mean Requests per second for North Carolina/US

A slightly better request performance rate for HTTP/2 is the result of that mesurement.

6.2.3 Frankfurt am Main/Europe

In contrast to the previous measurements, the measurements taken from a client in Frankfurt am Main/Europe are representing a low latency link with approximately 7ms of base RTT. The graph for the HTTP RTTs is shown in Figure 7.

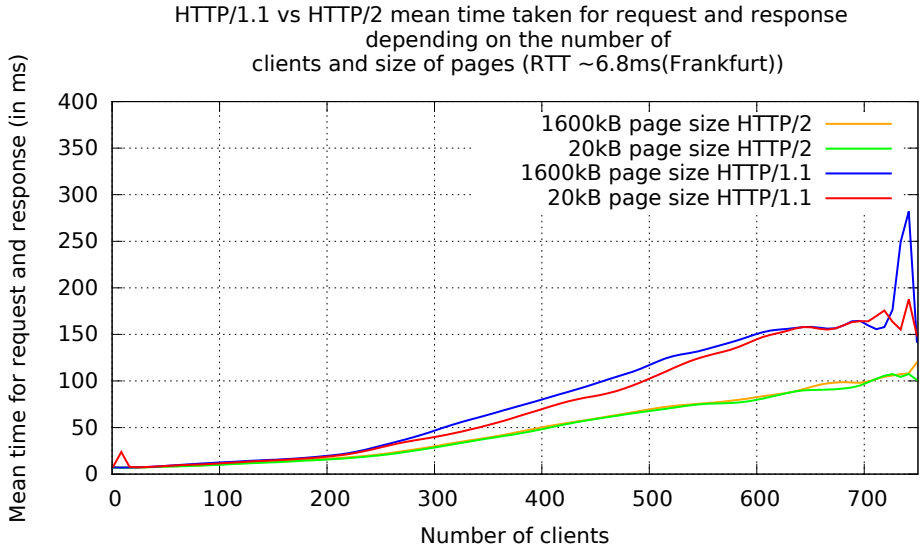

Figure 7: Mean RTT measurements for Frankfurt am Main/Europe

Compared to the previous measurements, the RTT graph looks different. Depending on the number of clients the RTT times grow nearly proportional. It is also a clear difference between HTTP/1.1 and HTTP/2 visible. The performed HTTP/2 measurements show significanltly less growing upto approximately 100ms for 750 clients, whereas the HTTP/1.1 RTT is for all measurements around 30% slower ending up in a maximum of around 150ms. The request rate graph shown in Figure 8 shows a different characteristic compared to the previous measurements.

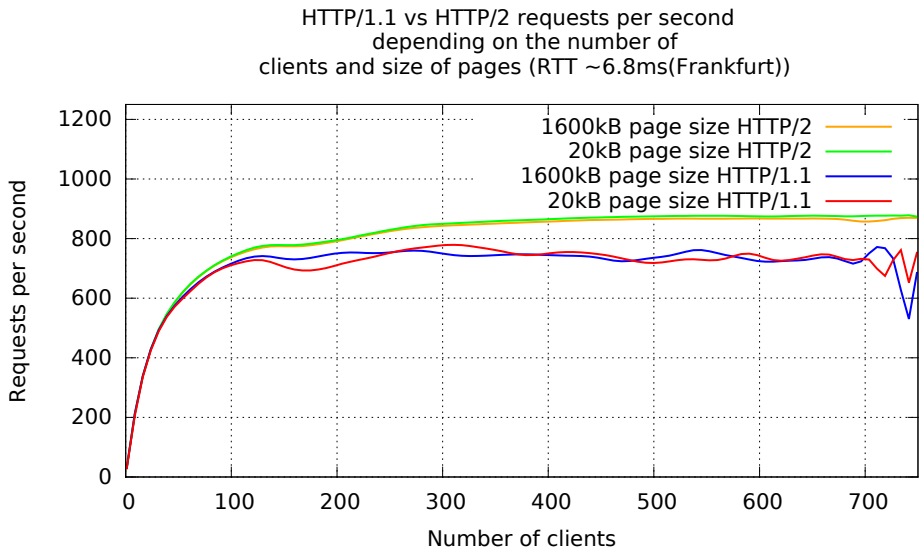

Figure 8: Mean Requests per second for Frankfurt/Germany

It turns out that the request rate is identical to a asymptotic curve for all measurements. The server simply cannot handle more than approximately 800 concurrent connections. That is also the reason that after 100 concurrent concurrent clients the request rate remains nearly static. It is again visible that HTTP/2 performs better compared to HTTP/1.1. The fact that the base RTT for that measurement is significantly less than in the previous measurement, leads to much more connections at the same time at the server and different performance graphs.

6.2.4 Amsterdam/Europe (local)

Finally we performed measurements within the OS3 network (RTT 0.3 ms). We conducted those tests in two different ways. First we did separate measurements for each protocol version and then we conducted parallel tests in order to simulate workloads on the server for HTTP/1.1 and HTTP/2 requests simultaneous. Figure 9 and Figure 10 show the graphs for the separate measurements.

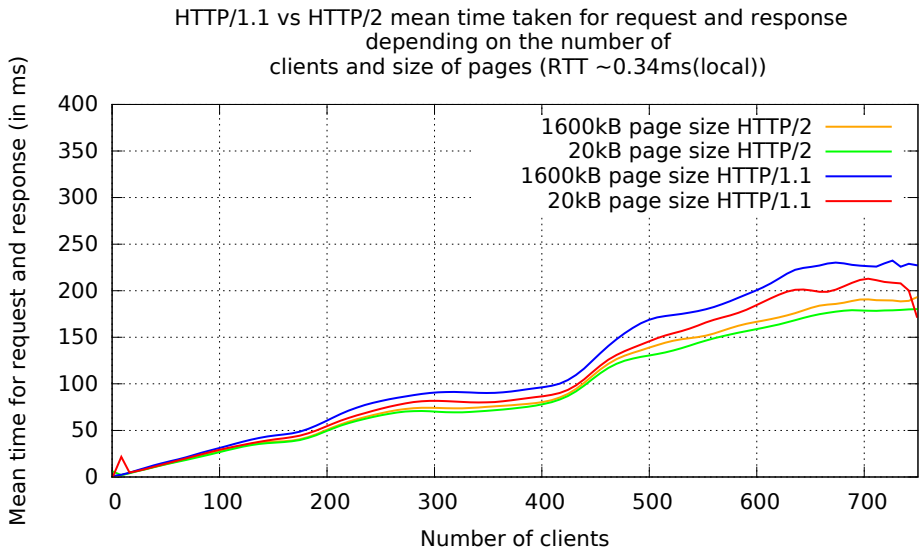

Figure 9: Mean RTT measurements for local/OS3 network

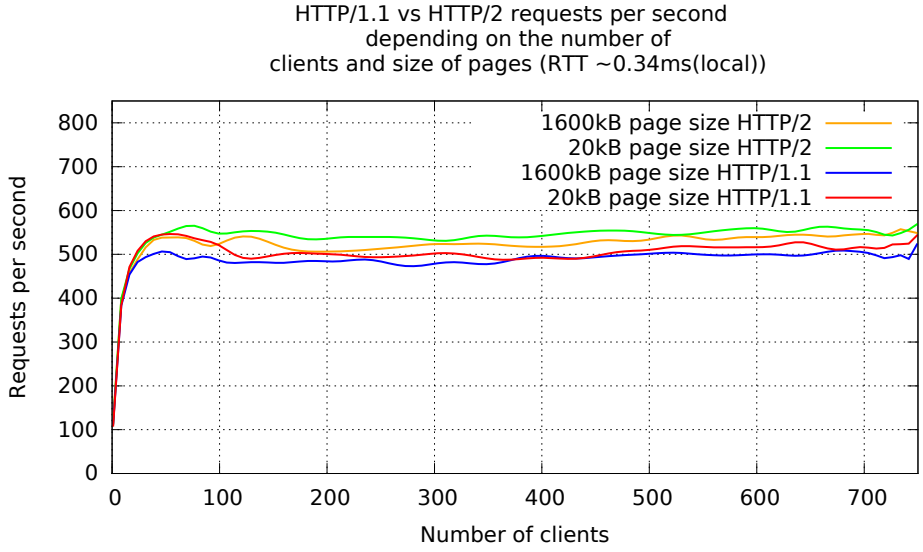

Figure 10: Mean Requests per second for local/OS3 network

The graph characteristics is similar to the low latency link in Frankfurt. The difference between both measurements is the number of client needed to reach the server limit in terms or requests per second. The local measurement already reaches that limit at approximately 40 clients. That can be explained by the lower latency within the OS3 network resulting in a much faster connection establishment.

As a last measurement we conducted parallel tests for HTTP/1.1 and HTTP/2. The related graphs are shown in Figure 11 and Figure 12

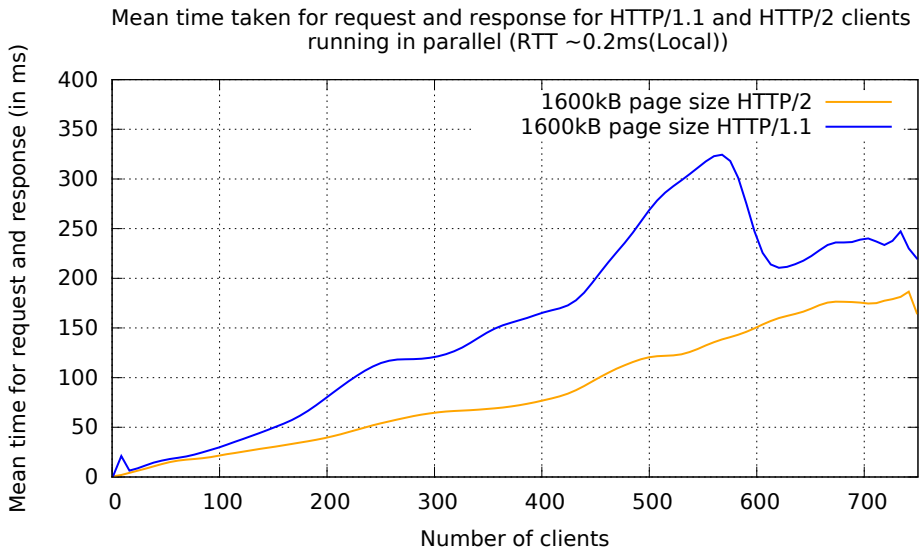

Figure 11: Mean RTT measurements for local parallel/OS3 network

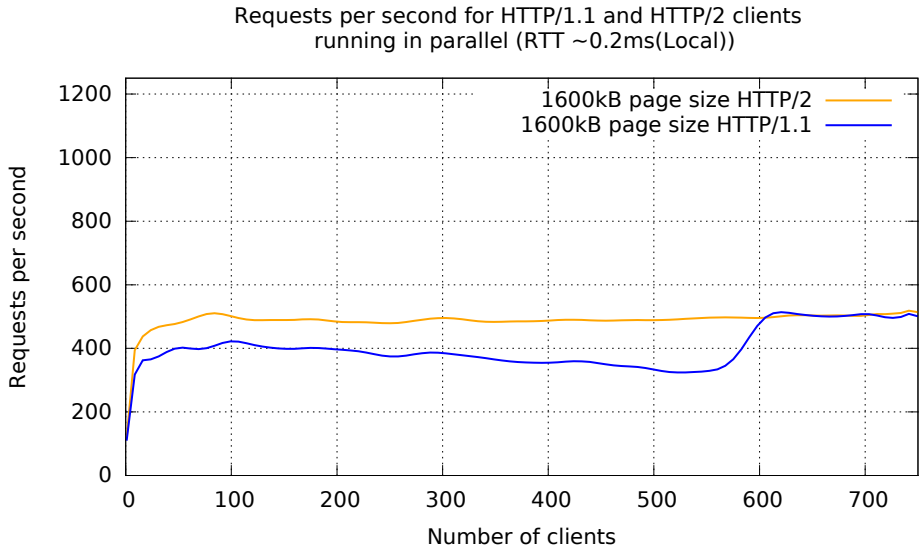

Figure 12: Mean Requests per second for local parallel/OS3 network

The results of the local parallel measurements is showing again a significant difference between HTTP RTT times of both versions. HTTP/2 is clearly faster than HTTP/1.1. Figure 11 shows a peak RTT for HTTP/1.1 at around 550 clients. At that moment the last HTTP/2 measurement was finished and afterwards the server was only busy with responding to HTTP/1.1 request. That is the reason for the decreasing RTT for HTTP/1.1 and the increasing number of requests per second starting from that point.

6.2.5 Mesurement analysis

We can conclude that for almost all measurements the RTT time for HTTP/2 is significantly faster than for HTTP/1.1. That result is independent of the requested page size and also from the distance between client and server. For high latency links we discovered different characteristics regarding RTT and request rate compared to low latency links (Europe/Local). On low latency links (<10ms) the number of requests per seconds grows much faster, which results in a quick saturation of the server to its maximum number of parallel requests. For high latency (Asia/North America) links a constant increase of RTT and request rate was measured for an increasing number of clients. In Figure 13 all measured differences between HTTP/1.1 and HTTP/2 for all locations is shown. Lines above zero (y-axis) show a faster RTT for HTTP/2.

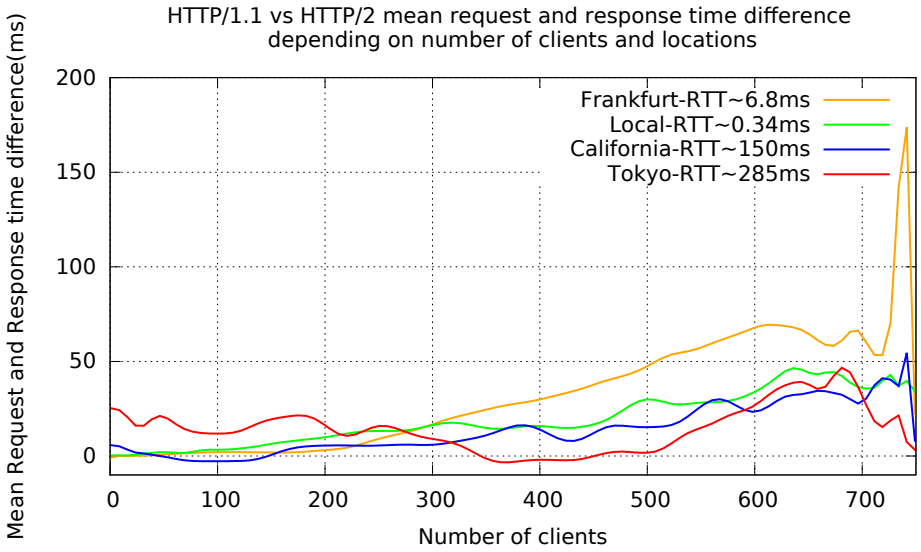

Figure 13: Mean RTT differences HTTP/1.1 - HTTP/2

6.3 Server Utilization

During all measurements, CPU and network utilization on the server has been monitored and measured. The different results can be categorized into low latency and high latency measurements. For that reason only the server utilization for the local (low latency) and North America (high latency) measurements will be presented. The performance graphs for the server CPU utilization is shown in Figure 14.

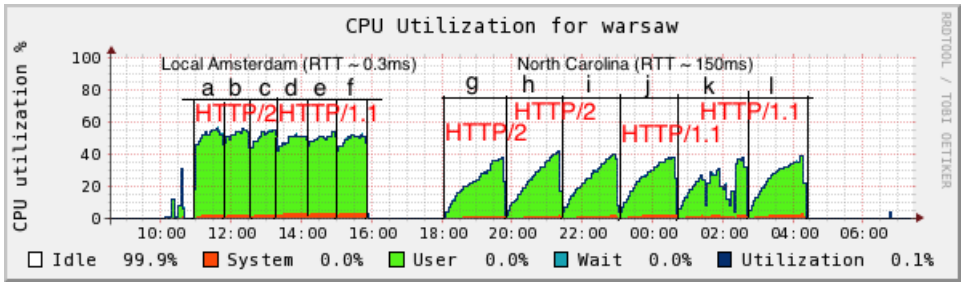

Figure 14: Server CPU utilization for Local and North America measurements

Table 4 describes the coreresponding annotations that have been made to point out which measurement belongs to which measurement parameters. That table applies also for the next performance graphs since we always show exactly the same time range in all following graphs.

Index	Protocol	Web Page Size
a	HTTP/2	small.html (20kB)
b	HTTP/2	medium.html (600kB)
c	HTTP/2	large.html (1600kB)
d	HTTP/1.1	small.html (20kB)
e	HTTP/1.1	medium.html (600kB)
f	HTTP/1.1	large.html (1600kB)
g	HTTP/2	small.html (20kB)
h	HTTP/2	medium.html (600kB)
i	HTTP/2	large.html (1600kB)
j	HTTP/1.1	small.html (20kB)
k	HTTP/1.1	medium.html (600kB)
l	HTTP/1.1	large.html (1600kB)

Table 4: Different Measurement parameters

The attentive reader will now notice that a medium.html web page shows up in Table 4. Indeed we created three different types of web pages. During the analysis of the data we realised that there is almost no significant difference in all measurements between the small size page and the medium size pages. For that reason we did not consider the measurements for the medium.html page and only focussed on the other two categories (large.html/medium.html). However, in the performance graphs of the server the measurement for the medium size web pages are included.
The left part of the graph shows the CPU utilization for local measurements (low latency) on the server and is annotated with the letters a to f. A sudden increase up to 60% from the beginning on staying steady until the measurements are finished is remarkable. That correlates to the request rates graph for local measurements and nicely presents a high CPU utilization caused by the high number of requests towards the server. The measurements from a to c show HTTP/1.1 measurements and from d to f HTTP/2 measurements. A significant difference regarding CPU utilization is not noticeable. Starting from 6pm until 4am (annotated by g to l) measurements over a high latency link (North America) were performed. We see a constant growing

of the CPU graph until the maximum number of clients of 750 for each test is reached. A nearly identical characteristic shows the bandwidth utilization graph. That correlates to the request per second graph as well and reflects a growing load caused by constantly increasing number of requests. A significant difference between HTTP/1.1 and HTTP/2 is also not noticeable for the high latency measurements. Furthermore a different load characteristic for changing sizes of web pages is not measurable for both protocol variants. The traffic graph (Figure 15 that represents the bandwidth utilization of the external interface of the server shows a similar characteristic compared to the CPU utilization graph (Figure 14). One difference is the increased amount of traffic that can be explained by the increased amount of data for larger web pages.

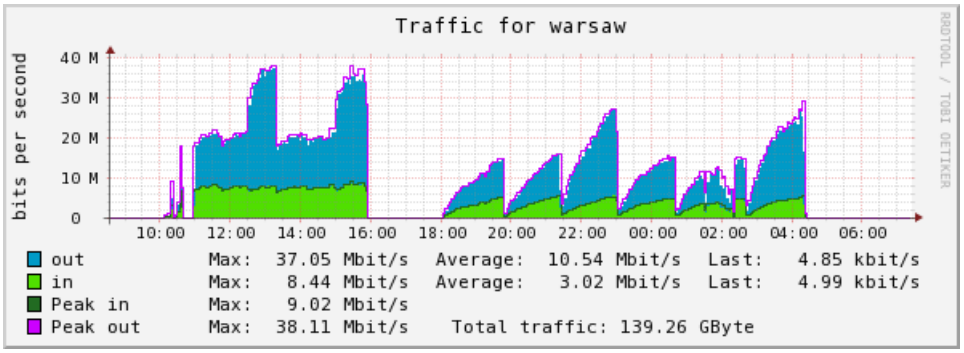

Figure 15: Server Bandwidth utilization for Local and North America measurements

It is worth to mention that for all HTTP/1.1 measurements a lot of TCP TIME WAIT connections were discovered on the server. TCP TIME WAITS occur when the endpoint (server) blocks a current connection before it can close it due to some missing packets for that particular session. That happens because the server has much more TCP connections to manage for HTTP/1.1 compared to HTTP/2 and shows clearly the benefit of using the HTTP/2 protocol. The graph representing the TCP socket states on the server is shown in Figure 16.

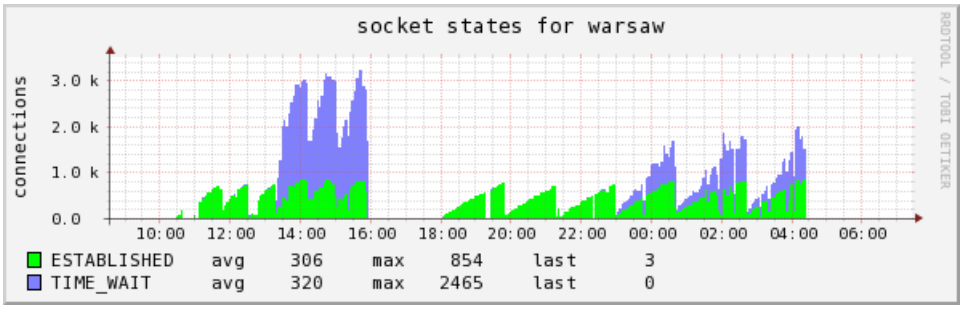

Figure 16: Server sockets for Local and North America measurements

7 Conclusions

The research has identified differences between the HTTP/1.1 and HTTP/2 in terms of response and request time, amount of header data produced, possible request rates and impacts on the server utilization. A full benchmark setup has been developed consisting of several HTTP clients in different geographically locations and a server instance that delivers responses for both HTTP protocol versions. Moreover webpages of different sizes have been created to simulate "real-world" scenarios. In order to have accurate measurement results a benchmarking script 8 that makes use of h2load [21] was deployed.
The measurements reflect the performance improvements a clients will experience in the near future by using the new HTTP/2 protocol. Those improvements are mainly based on header compression mechanisms and more effective usage of TCP connections. Particulary high latency connections will benefit from the multiplexing mechanism that was introduced in HTTP/2. HTTP/2 will result in significant less TCP connection establishments (TCP three-way handshakes) on server side and thus it will save a lot of costly packet round trip times that would be required to fetch the same web page by using HTTP/1.1.
A significant difference between HTTP/1.1 and HTTP/2 in terms of network load or CPU utilization was not detectable on the server. The main difference that has been discovered on server side is the TCP connection handling. It is faster and more effective when HTTP/2 is used, especially significantly less TCP TIME WAITS sockets were discovered on the server while performing HTTP/2 measurements.
Large web service providers need to consider some technical implementation details when switching to the new protocol. First, so called deep inspection packet filters need to be adjusted in order to let HTTP/2 packets to pass through. Those devices cannot longer inspect HTTP/2 traffic on application level, like it was used to be with encrypted or unencrypted plain text HTTP/1.1 connections. The protocol is now binary and a simple telnet connection to a web server in order to conduct an HTTP GET request manually will not be possible with HTTP/2 anymore. Furhermore "hacks", like Sharding or Sprinting which were introduced in HTTP/1.1 to solve performance issues of the protocol are no longer necessary. In fact those HTTP/1.1 workarounds will lead to performance loss rather than to performance gain, if combined with HTTP/2.

8 Future Work

As it is a recent specification new implementations of the protocol will show up. Moreover the big players of server implementations such as NGINX[24] and apache [20] have not implemented the protocol yet. But this is just a question of the time and probably they will wait until the protocol is specified in an RFC. In the meantime new implementations could be tested in comparison to see which one perform the best. At the time of writing the interesting server implementation of the protocol that are available are nghttp, H20, node-http2, openLiteSpeed. The list of all the implementations are available on the HTTP/2 github page.[17] Further work could be done by testing every new features of the protocol in order to show where are the improvement/drawbacks of this protocol in details per feature. HTTP/2 is still able to support HTTP/1.1 request with a proxy for legacy purposes. An interesting research subject could be how the HTTP/1.1 legacy in HTTP/2 can slow down the adoption of the new version of protocol. This has been the case for IPv6 and by showing the improvements in performance for HTTP/2 it would be a pity to see legacy hold improvements in technology back again.

References

[1] RFC7230 Fielding, R., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014, Available at:http://www.rfc-editor.org/info/rfc7230 [Accessed 26 Mar. 2015].
RFC7231 Fielding, R., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014, Available at: http://www.rfc-editor.org/info/rfc7231 [Accessed 26 Mar. 2015].
RFC7232 Fielding, R., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, June 2014, Available at: http://www.rfc-editor.org/info/rfc7232 [Accessed 26 Mar. 2015].
RFC7233 Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, June 2014, Available at: http://www.rfc-editor.org/info/rfc7233 [Accessed 26 Mar. 2015].
RFC7234 Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, June 2014, Available at: http://www.rfc-editor.org/info/rfc7234 [Accessed 26 Mar. 2015].
RFC7235 Fielding, R., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, June 2014, Available at: http://www.rfc-editor.org/info/rfc7235 [Accessed 26 Mar. 2015].

[2] Hypertext Transfer Protocol version 2 (draft-ietf-httpbis-http2-17) (2015). M. Belshe, Twist R. Peon [online] IETF, Available at: https://datatracker.ietf.org/doc/draft-ietf-httpbis-http2/?include_text=1 [Accessed 26 Mar. 2015].

[3] HPACK - Header Compression for HTTP/2 draft-ietf-httpbis-header-compression-07 (2015). R. Peon, H. Ruellan [online] IETF, Available at: http://tools.ietf.org/html/draft-ietf-httpbis-header-compression-07 [Accessed 26 Mar. 2015].

[4] Stenberg, D. (2015). http2 explained - The HTTP/2 book. [online] Daniel.haxx.se. Available at: http://daniel.haxx.se/http2/ [Accessed 27 Mar. 2015].

[5] Tools.ietf.org, (2015). Httpbis Status Pages. [online] Available at: https://tools.ietf.org/wg/httpbis/ [Accessed 27 Mar. 2015].

[6] Google Developers, (2015). SPDY. [online] Available at: https://developers.google.com/speed/spdy/ [Accessed 27 Mar. 2015].

[7] Raymond, E. (2015). The Importance of Being Textual. [online] Catb.org. Available at: http://www.catb.org/esr/writings/taoup/html/ch05s01.html [Accessed 27 Mar. 2015].

[8] Curl.haxx.se, (2015). cURL - README.http2. [online] Available at: http://curl.haxx.se/dev/readme-http2.html [Accessed 27 Mar. 2015].

[9] Wiki.wireshark.org, (2015). HTTP2 - The Wireshark Wiki. [online] Available at: https://wiki.wireshark.org/HTTP2 [Accessed 27 Mar. 2015].

[10] Breachattack.com, (2015). BREACH ATTACK. [online] Available at: http://breachattack.com/ [Accessed 27 Mar. 2015].

[11] Blackhat, (2013). i A Perfect CRIME?. [online] Available at: https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf [Accessed 27 Mar. 2015].

[12] A 2x Faster Web. (2009). [online] Chromium Blog. Available at: http://blog.chromium.org/2009/11/2x-faster-web.html [Accessed 20 Feb. 2015].

[13] Servy, H. (2015). Evaluating the Performance of SPDY-enabled Web Servers. [online] Neotys.com. Available at: http://www.neotys.com/blog/performance-of-spdy-enabled-web-servers/ [Accessed 20 Feb. 2015].

[14] Podiatry, G. (2015). Guy's Pod Blog Archive Not as SPDY as You Thought. [online] Guypo.com. Available at: http://www.guypo.com/not-as-spdy-as-you-thought/ [Accessed 20 Feb. 2015].

[15] Wang et al. (2014). How Speedy is SPDY?, 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). [online] usenix.org. Available at: https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-wang_xiao_sophia.pdf [Accessed 20 Feb. 2015].

[16] Amazon Elastic Compute Cloud (Amazon EC2) (2015). Available at: http://aws.amazon.com/ec2/ [Accessed 20 Feb. 2015].

[17] HTTP/2 client/server implemenations (2015). Available at: https://github.com/http2/http2-spec/wiki/Implementations [Accessed 20 Feb. 2015].

[18] Hypertext Transfer Protocol version 2 draft-ietf-httpbis-http2-14 (2014). Available at: https://tools.ietf.org/html/draft-ietf-httpbis-http2-14 [Accessed 4. March 2015]

[19] HTTP/2 experimental server (2015). Available at: https://nghttp2.org/documentation/nghttpd.1.html [Accessed 4. March 2015]

[20] The Apache HTTP Server Project (2015). Available at: http://httpd.apache.org/ [Accessed 4. March 2015]

[21] Benchmarking tool for HTTP/2 and SPDY server (2015). Available at: https://nghttp2.org/documentation/h2load.1.html [Accessed 4. March 2015]

[22] Apache HTTP server benchmarking tool (2015). Available at http://httpd.apache.org/docs/2.2/programs/ab.html [Accessed 4. March 2015]

[23] Httparchive.org, (2015). HTTP Archive - Trends. [online] Available at: http://httparchive.org/trends.php [Accessed 11 Mar. 2015].

[24] Garrett, O. (2015). How NGINX Plans to Support HTTP/2 - NGINX. [online] NGINX. Available at: http://nginx.com/blog/how-nginx-plans-to-support-http2/ [Accessed 27 Mar. 2015].

[25] amazon.com, (2015). Amazon - T2 Instances. [online] Available at: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html [Accessed 11 Mar. 2015].

[26] nghtt2.org, HTTP/1.1 HTTP/2 reverse proxy. [online] Available at: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html [Accessed 11 Mar. 2015].

Appendices

```perl
#!/usr/bin/perl
# Author Martin Leucht 2015 <martin.leucht@os3.nl>
#
# Benchmarking tool for HTTP/2 servers/reverse proxies
# based on h2load
# https://nghttp2.org/documentation/h2load.1.html
#
# Usage: ./measure.pl [URL-list-file]
# Output is written to an CSV file

use List::Util qw(sum);
use strict;
use warnings;

my $file_urls = $ARGV[0];
my $runs=10;
my $stepsize_clients=2;
my $parallel_clients = 750;
my $bin_v2 = '/usr/local/bin/h2load';
my $start_datestring = gmtime();
my $time = time;
my $filename = "report_http_${file_urls}_${time}.csv";

sub check{
    my %units = ('1000' => 's','0.001' => 'us','1' => 'ms',);
    my $value = shift;
    my $unit = shift;

    foreach my $key (keys %units) {
        if ($units{$key} eq $unit) {
            our $ret = $key * $value;
            }
        }
return our $ret;
}

sub average {
    my $size = @_;

    if ($size == 0) {
        print "NA";
        }
    else {return sum(@_)/@_;}
}


open(my $fh, '>', $filename) or die "Could not open file '$filename' $!";
print $fh "$start_datestring\n";
print $fh "number_of_requests,total_time_test,req_per_sec_v2,speed,traffic_total,traffic_header,
           traffic_data,min_request,max_request,mean_request,sd_request,sd_percent_request,
           succeeded_requests,succeeded_failed\n";
close $fh;


```

```
55
56  for (my $n=1; $n <= $parallel_clients; $n=$n+$stepsize_clients) {
57      my @finished_total_time_v2= ();
58      my @finished_req_per_sec_v2 = ();
59      my @finished_speed_v2 = ();
60      my @finished_traffic_total_v2= ();
61      my @finished_traffic_header_v2 = ();
62      my @finished_traffic_data_v2 =();
63      my @finished_min_request_v2 =();
64      my @finished_max_request_v2 =();
65      my @finished_mean_request_v2 =();
66      my @finished_sd_request_v2 =();
67      my @finished_sd_percent_request_v2 =();
68      my @finished_succeeded =();
69      my @finished_failed =();
70
71
72
73      for (my $i=1; $i <= $runs; $i++) {
74
75          my @command = `$bin_v2 -n $n -c $n --input-file=$file_urls --max-concurrent-streams=auto `;
76
77          foreach my $line (@command) {
78
79              if ( $line =~ m/^finished.*?([0-9]*\.[0-9]*)(\w{1,2}).*?(\d+)\s.*?(\d+\.\d+).*/ ) {
80
81                  my $val = check($1,$2);
82                  push(@finished_total_time_v2, $val);
83                  push(@finished_req_per_sec_v2, $3);
84                  push(@finished_speed_v2, $4);
85                  }
86              elsif ( $line =~ m/^traffic.*?(\d+).*?(\d+).*?(\d+).*/ ) {
87                  push(@finished_traffic_total_v2, $1);
88                  push(@finished_traffic_header_v2, $2);
89                  push(@finished_traffic_data_v2, $3);
90                  }
91
92              elsif ( $line =~ m/^requests.*?(\d+)\s(?=succeeded).*?(\d+)\s(?=failed)/ ) {
93                  push(@finished_succeeded, $1);
94                  push(@finished_failed, $2);
95                  }
96
97              elsif ( $line =~ m/^time.*?(\d+\.*\d*)(\w{1,2}).*?(\d+\.*\d*)(\w{1,2}).*?(\d+\.*\d*)(\
98                  my $val1 = check($1,$2);
99                  my $val2 = check($3,$4);
100                 my $val3 = check($5,$6);
101                 my $val4 = check($7,$8);
102
103                 push(@finished_min_request_v2, $val1);
104                 push(@finished_max_request_v2, $val2);
105                 push(@finished_mean_request_v2, $val3);
106                 push(@finished_sd_request_v2, $val4);
107                 push(@finished_sd_percent_request_v2, $9);
108                 }
109             else {print "";}
110         }
```

```
111     }
112
113
114     my $mean_finished_total_time_v2 = sprintf("%.4f", average(@finished_total_time_v2));
115     my $mean_finished_req_per_sec_v2 = sprintf("%.2f",average(@finished_req_per_sec_v2));
116     my $mean_finished_speed_v2 = sprintf("%.2f",average(@finished_speed_v2));
117     my $mean_finished_traffic_total_v2 = sprintf("%.0f",average(@finished_traffic_total_v2));
118     my $mean_finished_traffic_header_v2 = sprintf("%.0f",average(@finished_traffic_header_v2));
119     my $mean_finished_traffic_data_v2 = sprintf("%.0f",average(@finished_traffic_data_v2));
120     my $mean_finished_min_request_v2 = sprintf("%.4f",average(@finished_min_request_v2));
121     my $mean_finished_max_request_v2 = sprintf("%.4f",average(@finished_max_request_v2));
122     my $mean_finished_mean_request_v2 = sprintf("%.4f",average(@finished_mean_request_v2));
123     my $mean_finished_sd_request_v2 = sprintf("%.4f",average(@finished_sd_request_v2));
124     my $mean_finished_sd_percent_request_v2 = sprintf("%.2f",average(@finished_sd_percent_request_
125     my $mean_finished_succeeded_v2 = sprintf("%.0f",average(@finished_succeeded));
126     my $mean_finished_failed_v2 = sprintf("%.0f",average(@finished_failed));
127
128     open(my $fh, '>>', $filename) or die "Could not open file '$filename' $!";
129     print $fh "$n,$mean_finished_total_time_v2,$mean_finished_req_per_sec_v2,
130                 $mean_finished_speed_v2,$mean_finished_traffic_total_v2,$mean_finished_traffic_hea
131                 $mean_finished_traffic_data_v2,$mean_finished_min_request_v2,$mean_finished_max_re
132                 $mean_finished_mean_request_v2,$mean_finished_sd_request_v2,$mean_finished_sd_perc
133                 $mean_finished_succeeded_v2,$mean_finished_failed_v2\n";
134     close $fh;
135
136 }
137
138 my $end_datestring = gmtime();
139 open(my $fh, '>>', $filename) or die "Could not open file '$filename' $!";
140 print $fh "$end_datestring\n";
141 close $fh;
```