TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Rechnernetze

Thesis [Diplomarbeit]

HTTP/2 – POSSIBILITIES FOR AND EXTENSIONS TO THE PROXY PATTERN

Timo Lutz
Matriculation number: 3444118

Supervised by:
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
and:
Dr. Ing. Tenshi Hara

Submitted on 31th March 2017

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur Professur Rechnernetze

Aufgabenstellung für die Diplomarbeit

THEMA: HTTP 2 – Möglichkeiten und Erweiterungen für das Proxy-Pattern

Name:	Lutz, Timo	Studiengang:	Dipl. Medien-Inf. (PO 2004)
Matrikel-Nummer:	3444118	Projekt/Fokus:	Mobile and Ubiquitous Comp.
verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill		
involvierte Mitarbeiter:	Dr.-Ing. Tenshi Hara, Dr.-Ing. Thomas Springer		
Beginn am:	28.07.2016	einzureichen bis:	27.01.2017

ZIELSTELLUNG

Mit der Veröffentlichung des Standards für HTTP 2 haben sich diverse Verbesserungen und Neuerungen für die Kommunikation im WWW ergeben. Dies betrifft nicht nur klassische Browser-Anwendungen, sondern alle auf der klassischen Client-Server-Architektur aufbauenden Anwendungen.

Aus diversen Gründen ist oftmals eine Implementierung des Proxy-Patterns notwendig. Dies kann seine Ursache beispielsweise in der dynamischen Adaption von Web-Inhalten für Mobilgeräte, oder aber auch in Anonymisierungsbegehren haben. Am Lehrstuhl Rechnernetze wurde beispielsweise ein Crowdsourcing-Proxy implementiert, der eine zentrale Nutzer- und Submission-Verwaltung bereitstellt, um Crowdsourcing-Beitragende gegenüber Crowdsourcing-Betreibenden zu anonymisieren.

Ziele dieser Diplomarbeit sind ersten die systematische Untersuchung der Neuerungen und Möglichkeiten von HTTP 2 im allgemeinen Bezug auf die Client-Server-Architektur, und zweitens die Betrachtung und ggf. Konzeption einer Erweiterung der Mechanismen auf das Proxy-Pattern. Insbesondere soll dabei Rücksicht auf Ende-zu-Ende-HTTP-2-Umsetzung genommen werden. Bereits existierende Ansätze sollen dabei auf ihre Erweiterbarkeit untersucht werden.

Die Ergebnisse sollen in einem Konzept zusammengeführt werden und anschließend sinnvoll evaluiert werden. Dazu ist die Definition von Evaluationskriterien zwingend notwendig.

Bei Bedarf kann zur Untersuchung der Konzepte und für die notwendige Evaluation der oben erwähnte Corwdsourcing-Proxy verwendet werden.

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
(verantwortlicher Hochschullehrer)

SCHWERPUNKTE

- Untersuchung verwandter Arbeiten aus aktuellem Stand in Forschung und Technik,
- Definieren von Anforderungen,
- Definieren von Bewertungskriterien für die Anforderungserfüllung,
- Konzeption einer Evaluationsmethodik,
- Proof-of-Concept-Implementierung, und
- Evaluation und Auswertung der Ergebnisse.

TECHNISCHE UNIVERSITÄT DRESDEN

EINGEGANGEN
17. JAN. 2017

Fakultät Informatik Prüfungsamt

Antrag auf Verlängerung der Bearbeitungszeit der Diplomarbeit (PO 2004)

Name: Lutz	Vorname: Timo
E-Mail-Adresse: timo.lutz@mailbox.tu-dresden.de	Mat.-Nummer: 3444118
Studiengang: Dipl.-Medien-Inf. (PO 2004)	Imma.-Jahrgang: 2007
Betreuender HSL: Prof. Dr. A. Schill	
Beginn: 28. Juli 2016	Abgabe: 27.01.2017, wegen Krankheit 24.02.2017 ✓
Dauer der Verlängerung ***(max. 3 Monate)***: 4 Wochen	
Neuer Abgabetermin: 24. März 2017	

Begründung der Verlängerung:

Die Vielzahl zu berücksichtigender und zu untersuchender Ansätze hat eine sehr komplexe Konzeptionierungsphase nach sich gezogen. Die Unterbrechungen durch Krankheit haben diesen Prozess weiterhin erschwert.

Um die Qualität der sich aus der Implementierung ergebenden Ergebnisse nicht zu gefährden, sollten keine Abstriche in der Evaluationszeit gemacht werden. Daher beantrage ich eine Verlängerung um 4 Wochen.

Das Vorgehen wurde mit dem Betreuer abgestimmt.

16.01.2017	i. A. Braun
Datum, Unterschrift Studierende/r	Zustimmung betreuender HSL

Dieser Antrag ist mit einer Kopie der Anmeldung zur Diplomarbeit fristgerecht im Prüfungsamt vorzulegen.

17. Jan. 2017	
Datum	Unterschrift Prüfungsamt

Entscheidung des Prüfungsausschusses:

Dem Antrag auf Verlängerung der Diplomarbeit wird stattgegeben / nicht stattgegeben.

25. Jan. 2017	Prof. Dr. [illegible] Vorsitzender des Prüfungsausschusses Studiengang Medieninformatik Technische Universität Dresden Fakultät Informatik
Datum	Unterschrift Prüfungsausschuss

ABSTRACT

HTTP/2 is the next evolutionary step of the Internet's most used and adopted protocol. It accommodates the changed demands of on the one hand yearly growing content in number of resources and size and on the other hand altered way of accessing this content: In 2015, for the first time in the history of the internet, more traffic was generated from mobile devices than from stationary ones and this trend is expected to last. Moreover has the bandwidth available to each user grown exponentially while the latency, mostly depending on the distance to span, only improved marginally. In order to resolve this issue, a paradigm shift from simplicity towards performance has been put into effect by using a binary framing layer instead of being a pure text based protocol as its predecessor HTTP 1.1. The effects of this combined with header compression and a new set of features like *Server Push* shall be evaluated in general and with respect to the proxy pattern using the example of the SANE proxy at the Technische Universität Dresden.

CONTENTS

1 INTRODUCTION

“

Dimidium facti, qui coepit, habet: sapere aude, incipe.

Horaz

”

Since the introduction of the Hypertext Transfer Protocol the Internet or better the *Web* underwent many changes: Was the first version only intended to serve *hypertext* as the name suggests, it nowadays is used to transport a great variety of file types a thousandfold in size. As the usage pattern changed, the protocol, aside from smaller optimizations, remained the same. To better cope with the changed demands, a new version of this jack of all trades protocol has been launched in 2015: HTTP2.0 or HTTP/2 or just H2. The increase of the major version and the abandonment of a minor version reflects the extensive changes made under the hood, as will be examined carefully in the course of this thesis.

1.1 OBJECTIVE

The objectives of the work at hand are, firstly, to systematically evaluate the impact of the improvements on client-server architecture in HTTP/2 in a general way. Secondly, to relate these improvements to the proxy pattern and, in case they turn out advantageous, implement them with special consideration of end-to-end architectures and possibly pre-existing solutions. The outcome of the conducted improvements will finally be measured on the basis of previously defined evaluation criteria.

1.2 MOTIVATION

Since the introduction of HTTP1.1 in the year 1991, the *Web* underwent massive changes, although with the increase in size of websites also the bandwidth increased to a comparable extent. Alongside sheer bandwidth, the second factor significant for the experienced speed only improved marginally: Latency, which is mainly a function of the peers' distance, as the time a packet needs to travel a certain distance is delimited by the speed of light and the wire's refractive index respectively. As those last two factors can be considered constant for the case at hand, other ways of reducing the load time have to be found, as latency influences the necessary time to load data on multiple ways: Due to the increased complexity of web sites and -applications, the amount of requests to download an average website also increased, leading to a lower perceived speed, as the individual latencies add up with every additional round-trip needed until the whole website has arrived at their particular destination. Moreover, is the latency of a connection also determining the maximum data rate as a factor in the *Bandwidth-Delay Product*. As an application layer protocol HTTP/2 has in fact minimal impact on this issue of lower layers, but bears functionality to send data *earlier* and more efficiently, combine certain steps to require less round-trips or to recognize whether something has to be sent at all.

"No bit is faster than a bit not sent." [Gri13a]

The effect of the Bandwith-Delay product becomes even stronger in mobile networks, where the transmission of packets naturally takes even more time. This issue will be examined more detailed in 2.4.3. With HTTP/2, the subject of this work, most improvements concern an overall faster perceived transmission alongside a richer feature set, which in turn entails other implications/effects, as will be shown in the following.

This work's purpose is to take a closer look at the improvements both in general and with relation to an already established project at the TU Dresden, yet still being under development, called *the SANE*. It is a generic platform, providing an infrastructure for arbitrary crowdsourcing services, freeing the crowdsourcer from recurring tasks like user management or targeting of certain user groups. Another aspect of this platform is a proxy service, mainly to provide anonymization of the crowdsourcing participants to the crowdsourcing conductor. Examining the improvements of HTTP/2 regarding reasonability and feasibility, the implementation of reasonable improvements and rating their impact by means of the evaluation criteria is the main focus of this work.

1.3 STRUCTURE

First of all, this work will introduce the networking basics necessary to understand HTTP's high level mechanics in the Preliminaries chapter. More precisely, the basics of TCP and TLS will be treated. Next, the evolution of HTTP will be retraced from the first basic draft to HTTP in version 1.1, which is still in use nowadays. In the following is pointed out how and where this version falls short and how SPDY, a blueprint for HTTP/2, already tried to solve its issues. Having presented the precursors of HTTP/2 comprehensively, the proxy pattern and - based upon that - the SANE project at the TU Dresden are introduced in short terms to close this chapter. The following chapter, Related Work, treats relevant works of other authors. At first, works that summarize the development of HTTP/2 are given, then works are presented that evaluate HTTP/2 in comparison to its predecessors based on various quantification criteria, to finally give a summary of these works. After that, the Concept chapter examines which of HTTP/2 amendments can be reasonably put to use for the SANE and how they can be reconciled with the special requirements of SANE's architecture. The following Proof of Concept chapter treats the practical limitations and the implementation of the functionality found advantageous. Based upon the evaluation criteria prepared in the Concept chapter, the now following Evaluation chapter assesses new SANE implementations, the principle functional demonstration and verifies whether HTTP/2's advancements work as intended. The last chapter finally sums up the entire work and gives an overview of upcoming trends of HTTP/2's wider context as well as the next steps recommended for the SANE's further development.

2 PRELIMINARIES

“

The extreme sophistication of modern technology - wonderful though its benefits are - is, ironically, an impediment to engaging young people with basics: with learning how things work.

Martin Rees

”

The purpose of this chapter is to introduce the basic knowledge crucial to understanding why and where HTTP 1.1 falls short and how HTTP/2 deals with its predecessors' issues.

2.1 TCP BASICS

In order to understand where HTTP 1.1 falls short and possible solutions it is vital to be familiar with the fundamentals of the TCP layer underneath. Especially as some issues of this underlying layer reoccur in HTTP and this protocol in turn reaches out into this lower layer as can be seen in the following.

Three Way Handshake

Every TCP connection starts with a *Three Way Handshake*, as can be seen in 2.1, via SYN (synchronize) packet by the peer inducing the connection. It is responded to with a combined SYN and ACK (acknowledge) packet. To finalize this process the reception of the SYN ACK packet is again acknowledged via ACK packet. During the handshake client and server mainly agree upon sequence numbers to be used for further communication. These packet sequence numbers are randomly appointed by the sender of the SYN packet on either side; they are from this point on used to mark a packet as an answer to a previously received packet by incrementing its sequence number by 1 and to determine the correct order of the packets, as they can take different routes from sender to receiver and are therefore not necessarily received in the order sent. Furthermore, the peers propagate their initial receive window size, which is relevant to Flow Control, as further elaborated below. As the name *Three Way Handshake* suggests, it embraces three packets in total to be sent among the peers, meaning establishing a TCP connection takes at least three times the duration one packet takes to be transmitted among them, commonly referred to as *latency*.

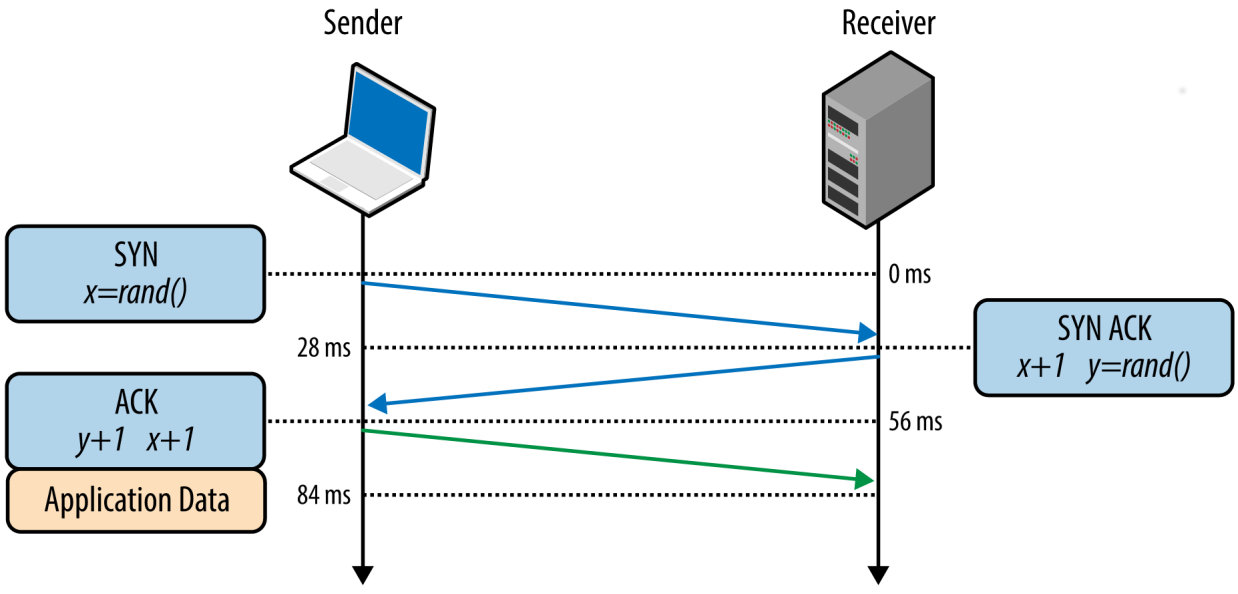

Figure 2.1: Three Way Handshake with a packet latency of 28 ms

Flow Control

Flow Control denotes the technique utilized to avoid overloading the receiver with packets from the sender it is currently unable to digest for various possible reasons. On establishment of a connection, each side initializes the size of their receive window (rwnd) with system default settings, corresponding to the buffer size allocated for incoming packets, and advertising them in their acknowledgment packets, ACK and SYN ACK, respectively. If the receive window reaches zero, no more packets are to be sent until another sent packet, carrying another rwnd value, gets acknowledged.

Congestion Control, TCP slow start and TCP fast open

The internet protocol suite embraces many strategies to avoid or take control of congestions, to number all of them would exceed the scope of this thesis. Nevertheless, one of them is crucial as will be seen later: *TCP Slow Start*.

In order to avoid overloading not only the receiver, as Flow Control does, but also the network with TCP packets, the capacity of the the medium, expressed as amount of packets en route between the peers has to be determined by the use of this exact mechanic: After the above mentioned initial Three Way Handshake, in which sender and receiver exchange their respective *receive window* (rwnd), the server starts sending packets in the amount of the minimum of the receive window and the congestion window size of 10 segments (since the most recent RFC 6928 in 2013 [Chu+13]). For every packet acknowledged by the receiver, the sender doubles the size of the congestion window (cwnd), what bears comparison with an exponential growth, until lost packets are recognized by not receiving an acknowledgment (ACK) packet for them from the receiver. This procedure is known as *congestion control*. At this very point of packet loss, *congestion avoidance* takes over: The sender reduces the congestion window size to the last known working cwnd value and increases it in smaller steps of the size of the initial window size (10 segments) with every round trip. Thus, the sender has to wait for a whole server round-trip to increase the window size, making latency again a crucial value. Also, for small files, the transfer has often already been finished before reaching the maximum window size, again making the the time for a full server round-trip the limiting factor for the connection.

With *TCP fast open* it is now possible to not only abbreviate the latency afflicted Three Point Handshake, but also the even more costly TCP slow start by reusing the parameters of an already existing connection. To achieve this, with the establishment of the first connection between peers, a cryptographic cookie is included by the server. This common secret is used later on to authenticate to the server as a client already known and proceed with the last known working connection settings.

2.2 TLS BASICS

As TLS is important for HTTP/2, it will be introduced here briefly. The abbreviations stands for **T**ransport **L**ayer **S**ecurity and, as the name suggests, it provides security features and is residing between the above mentioned TCP transport layer and the application layer. The most recent version and the version used by HTTP/2 is 1.2. It shall be mentioned in advance that HTTP/2 does not require TLS to be used, but all major browser developers made clear that their products will only support HTTP/2 with activated HTTPS, just as its blueprint SPDY as will be further elaborated in 2.3.4 TLS basically provides three services: Encryption of transmitted data using symmetric encryption, mutual authentication of the peers and measures to ensure the integrity of the data transmitted. With the optimizations *TLS false start* and *TLS session resumption*, it was possible to reduce the additional round-trips for TLS to only 1, both for new connections (with TLS false start) and for pre-existing connections (with TLS session resumption). Within this round-trip also the negotiation of sender and receiver regarding the HTTP protocol version to use takes place under application of the **A**pplication **L**ayer **P**rotocol **N**egotiation.

2.3 EVOLUTION OF HTTP

In this chapter, first the common characteristics of all revisions of the HTTP protocol are outlined, as each one is designed to be fully backward compatible. Then, a brief look at the history of the HTTP protocol and its evolutionary steps is given.

2.3.1 Commonalities among all revisions

As the **H**yper**T**ext **T**ransfer **P**rotocol is an application protocol on layer 4 in the DOD reference model, it operates on top of the transport layer. Typically for this purpose TCP is used, but also the use of the **U**ser **D**atagram **P**rotocol, a connection-less protocol operating on the same layer, is alternatively possible. Additionally TLS/SSL can be used to establish a **s**ecure connection, what contributes the **S** in **HTTPS**. HTTP was intentionally designed for transmitting *hypertext*, a term formed by its inventor, Tim Berners Lee in 1991, but is not necessarily limited to that. Moreover, it is able to transmit any kind of data in its body, but also in its header by the use of Base64 encoding, a method of encoding binary data into a set of characters that are not to be misinterpreted by intermediaries on the way to the recipient. It is designed to be stateless, meaning that the server does typically not retain any state or further data about the user. This means everything to fulfill the client's request has to be conveyed with every request. HTTP is a typical representative of a request-response protocol, meaning that every transmission from the server to the client (i.e. the response) results from a request from the client to the server, regarding a resource specified via **U**nified **R**essource **I**dentifier.

2.3.2 History of HTTP

In the following, the evolutionary steps of the individual revisions and its historic background are outlined.

HTTP 0.9

Already in 1991, the first proposal for a protocol to transfer so-called *hypertext* was written by the appointed father of the World Wide Web, Tim Berners Lee, himself. Nowadays this proposal is commonly known as *HTTP 0.9*; it was designed with simplicity in mind, what resulted in a also very straightforward, Telnet-friendly protocol. It supported only the *GET* method which is used to retrieve a document specified by file name and optionally the path relative to the web servers root directory. It completely lacked support for headers, version numbers and MIME typing of multimedia content. Nevertheless it built the base for any further development. [Ber91] [Gri13a] [GT02]

HTTP 1.0

The next iteration of HTTP, Version 1.0, already accommodated the newly emerged class of software named *Web Browsers* by being able to transmit meta-data about the client in the request and perform content negotiation to transmit documents other than those in plain text like HTML. The document that embraces HTTP 1.0 [BFF] as we know it today, should not be understood as a specification or an internet standard. Instead it is more an aggregation of practical improvements of its predecessor HTTP 0.9, that have already been consistently

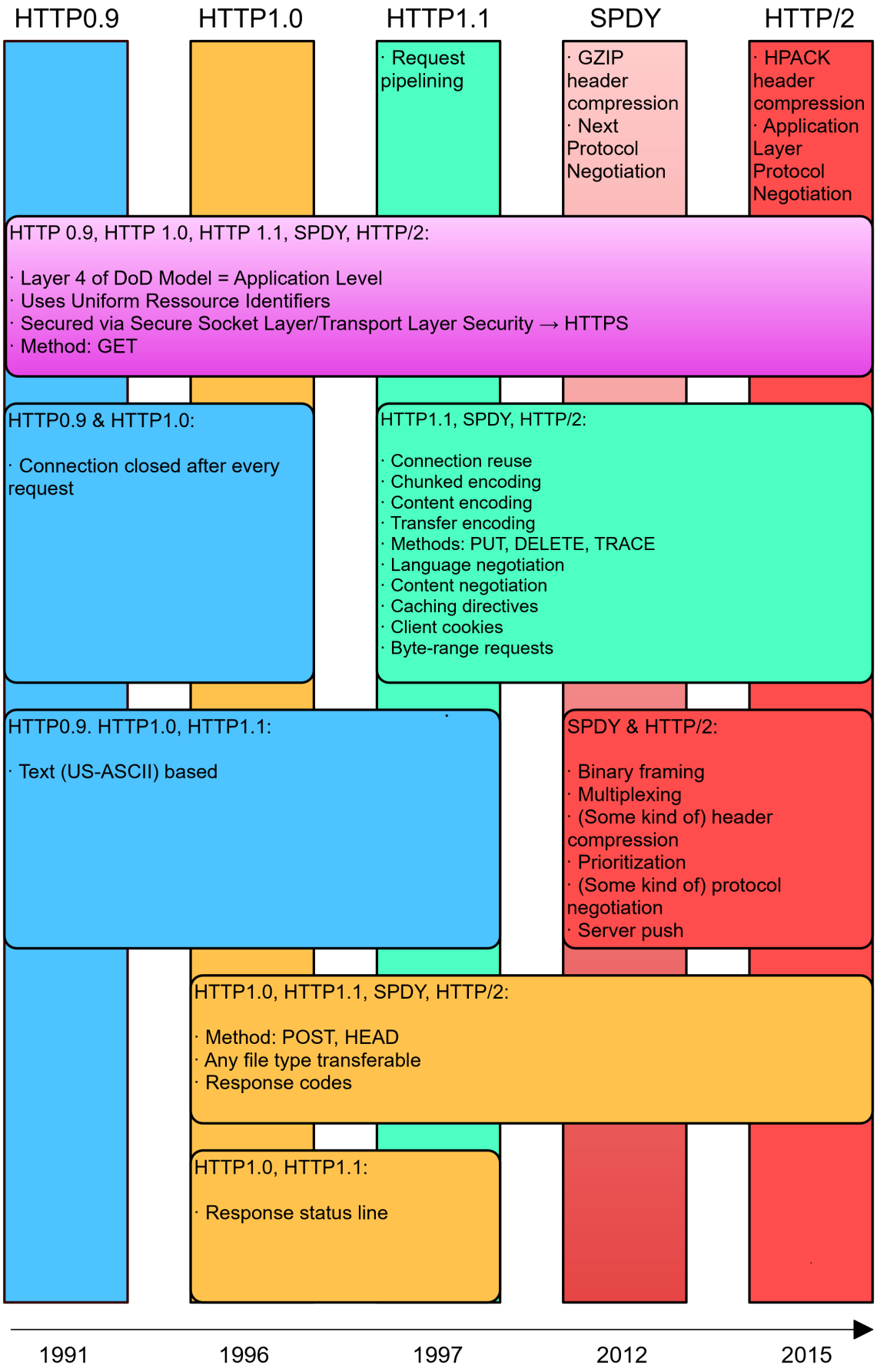

Figure 2.2: Functionality of individual revisions of HTTP and related protocol SPDY

implemented by the developers of the most widespread web servers at that time. Its methods were extended beyond the already existing *GET* by the two new methods *POST* and *HEAD*. The former one is intended to be used "to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line" [BFF] i.e. to add data to a specific resource. The latter is identical to the *GET* method with the only difference that the server must not return any entity body in the response. Its only intended use was to obtain meta-information about a specific resource. The other improvements comprise changes to the structure of request-/response objects as follows: A request may consist of multiple header fields, both type of objects had US-ASCII encoded headers, the response object is ought to be prefixed by a status line, has its own headers and was not limited to consist of only hypertext anymore. With this version, the connection was still closed after every request/response. [Gri13a] Besides those features implemented in the most common web servers, some existed that were only available to some web servers and are therefore only listed under *additional features* in [BFF], like the ability to reuse connections for multiple requests, additional methods and additional header fields for specifying the used character set and language. This feature set between HTTP 1.0 and HTTP 1.1 is often referred to as HTTP1.0+ [GT02]

HTTP 1.1

These just mentioned features were not officially standardized until the release of HTTP 1.1 in June 1999 as RFC2616 in [Gro+99] and got finally rid of protocol ambiguities. It was specified first in RFC2068 as a work in progress in January 1997.

The following focuses on those improvements with a certain relevance for optimizations in HTTP/2. From today's point of view, the most important improvement was *connection keep-alive*, what allowed the use of an already established connection for more than just one single request as with HTTP 1.0. Certainly, the repeated use of a connection made other extensions necessary in order to enable the peers to perceive the end of one request and the beginning of another: In HTTP 1.0 the closing of a connection also meant the end of a transmission. As connections are reused, another way of delimiting had to be introduced. One way of doing so is to specify the length of the subsequent content by the use of a *content-length* field in the header. Another way of doing so is using *chunked encoding*, sending a series of pre-specified chunks of a fixed length. This is especially of use if the content is generated dynamically, when the server does not know the resulting size at this point in time. It is the only allowed transfer encoding specified in this version of HTTP. [GT02]

Other than transfer encoding applying on the whole message, the newly introduced content encoding applies only on the body of the message, enabling it to be encoded in order to minimize its size, in other words to be compressed. The common algorithms available comprise gzip, zlib and the standard unix file compression, of which gzip is the most commonly used and the most effective one. More content encoding algorithms can be added as extension encodings. These can be specified by the client upon request by adding its supported ones in an *accept-encoding*-field in the request header.

The previously defined methods were extended by *PUT*, being essentially the same as POST but with the intention to consider data transferred this way as an update to existing data, *DELETE*, to delete resources from the server and the *TRACE* method, intended to verify the received data by prompting the server to send it back to the client. Nevertheless, only shortly after the release of RFC2616 it turned out that these newly introduced methods were neither sufficient for realizing Tim Berners Lee originally proclaimed goal of an editable world wide web, nor comprehensively implemented by web server developers. For this reason *WebDAV*, an extension of HTTP to allow the deployment and versioning of a set of files at once, was introduced, already disclosing first shortcomings of HTTP 1.1 only shortly after its introduction.

A further, at least intended, performance optimization is the ability of HTTP 1.1 to do *request pipelining* over persistent connections, meaning "(...) multiple requests can be enqueued before the responses arrive. While the first request is streaming across the network to a server on the other side of the globe, the second and third requests can get underway." [GT02] Anyway, this considered improvement leads to other issues, namely *head-of-line blocking*, which will be discussed further in the following chapter. Due to these issues, request pipelining has never been practically used and has hence been removed in all major browsers.

Also with this version, client cookies were introduced. With this feature it was for the first time possible to identify users and keep persistent sessions, i.e. extending the HTTP protocol, which is stateless by design, by tracking a non-volatile state. On the downside, these informations to identify the user sessions have to be transmitted with every request, bloating the request header, leading to new problems as will also be discussed in the following chapter. Anyway, this solution was *cheaper* than having to include session-held informations in every request.

It should be mentioned that there are many more improvements with this version beyond those mentioned above, like byte-range requests, new caching mechanism and -directives, language- and content negotiation which have no direct relevance for the work at hand.

2.3.3 Shortcomings of HTTP 1.1

Since HTTP has been introduced, its improvements have had a major stake in the success of the Web as we know it today, being the protocol responsible for the lion's share of the Internet traffic volume. As the size of websites and hence the traffic volume grow year by year the deficiencies of HTTP1.1 and their workarounds become more and more evident. The advancements of HTTP/2 address these very issues. For this reason they are named in the following.

According to [Ste14] and [Tes] the average size of a website has risen from 725 KB in 2011 to above 1.9 MB in 2015, split over a hundred individual resources. Transferring all of these objects over a single persistent connection turned out being too slow. For this reason browsers nowadays utilize four to eight TCP connections per target domain in order to transfer those resources.

Since this was still not enough to ensure short page load times, web developers adopted several strategies to increase page loading speed. An indirect one is to circumvent these hard-coded connection limits by the use of *domain sharding*. With this technique, the resources are distributed over several domains or sub domains, effectively circumventing the connection amount limits of the browsers. On the downside, with every request to a different domain come all negative side effects like the effects of TCP slow start, an additional DNS lookup and naturally the increased complexity of deploying and delivering a partitioned website. Other strategies,to increase page loading speed, in contrast, avoid the overhead and additional time needed for establishing new TCP connections by reducing the amount of objects to be transferred:

Spriting and *concatenation* are combination techniques to minimize the total amount of requests and therefore the protocol overhead every newly opened connection implies. The former combines many small images into a big one, which are then selectively *cut out* on the client side; the latter combines multiple text-based files like JavaScript or Cascading Style Sheets into a single one. [Gri13b] calls this *application layer pipelining*, as the result is the same as if HTTP pipelining was available: The data is transferred closely packed without suffering from the overhead and latency of having additional requests. *Inlining*, however, denotes embedding all kinds of resources directly into *parent* files like HTML and CSS with the same goal of less outbound requests. Certainly, these techniques also entail their very own disadvantages, as unneeded parts of that files are transferred. Moreover, this goes to the expense of cache granularity as it is only possible to cache the entire file or cache nothing at all and to memory usage on the client side, as all files have to be kept in there; image files even as a memory

intensive bitmap. Last but not least, also the initial page load time is affected, as the page is not rendered until all necessary resources, the HTML file itself and according CSS and JavaScript, are available to the client. From the developer's view, these techniques make the application more complex, as they imply pre-processing the files, need extra client-side code and also its deployment over various hosts with domain sharding has to be taken into consideration.

Instead, these workarounds would not be necessary, if it was possible to request all required resources at once, having the server deliver them in the order of their request over a single connection, called *pipelining*, which was introduced as aforementioned, with HTTP 1.1. Unfortunately, this entails another problem known as *Head of Line-Blocking*: If one of the resources in the server's response queue can not be processed or takes very long, all following responses queued are not processed, thus blocked. For this reason, this feature is disabled by default or even has been completely removed in almost all common web browsers.

Besides the growing payload from server to client, also the other way around - from client to server - increased in size, mainly due to larger request headers and cookies as a part of them, which have to be sent with every request. They even got that large, that they often exceed the initial TCP window, as pointed out in 2.1. In this case, an entire additional round-trip is needed in order to deliver the request.

Other issues of HTTP1.1 result from it being text-based and therefore not only slower to process than binary, but also more error-prone due to its ambiguity, as the standard defines four different ways to parse a single message, depending on type of the message, or more specifically, whether it is allowed to include a message body; the type of transfer encoding; the length of its content and the used media type.

2.3.4 SPDY

The SPDY protocol, pronounced *speedy*, served as a blueprint for the HTTP/2 protocol. It was mainly developed by Google engineers from mid 2009, although its development remained open to suggestions from the open-source community. Its declared objectives embraced a reduction in page load time of 50% while still maintaining content compatibility and of the underlying network infrastructure. Still during its development, after first results were published that showed the success of the measures undertaken by the protocol developers, the HTTPBis group started the development of HTTP/2, who incurred the SPDY/3 draft into what became http2 draft-00 [Gri13b] [Ste14]. At that point, the SPDY draft already contained the idea of a Binary Framing Layer, flow control on the application layer, Server Push capabilities and header compression, although at that time based on dictionary- and zlib compression. [PB]

2.4 HTTP/2

Based on the secondary literature of Stenberg [Ste14] and Grigorik [Gri13b] alongside the official HTTP/2 [BPT15] and HPACK [PR15] RFCs the following section sums up HTTP/2's advancements.

2.4.1 Preserving downward compatibility

As the HTTP/2 standard is an **extension**, **not a replacement**, the high level API remains almost the same. This basically comprises methods, status codes, header fields and URI schemes. The only differences regarding the API are

- Header keys are lowercase
- No more customized response messages, only error codes
- An additional error status code *421*

The additional status code 421 signals a *misdirect request*, meaning no response can be generated to the client's request, suggesting the wrong server was addressed.

According to [Gri13b], the experience from protocol updates in the past suggests that HTTP 1.1 will be in use by legacy clients for at least another decade. This means clients and servers have to maintain both standards and have the ability to upgrade from HTTP1.1 to HTTP/2 if supported. This protocol negotiation is done via the newly introduced ***A****pplication* ***L****ayer* ***N****egotiation* ***P****rotocol*, piggybacked onto the mandatory TLS handshake, without adding latency for an extra server round-trip.

If HTTP/2 without TLS (h2c) is to be used, this upgrade mechanism also has to be used to negotiate the protocol version as both versions use the same server port 80. This detour over HTTP1.1 thus introduces another round-trip that can not be avoided. Although it is possible to use unencrypted HTTP/2 in principle, all major browsers enforce the use of TLS by not establishing a connection to a server offering only h2c.

2.4.2 Advancement of HTTP/2

In the course of this section the advancements of HTTP/2 are introduced and put into relation.

Binary Framing Layer and implications

The most comprehensive advancement concerns the way HTTP messages are transferred. In contrast to HTTP 1.1, which transfers messages encoded in US-ASCII or a subset of it, they are binary encoded in HTTP/2. This means nothing less than a paradigm shift from simplicity to performance.

This *binary framing layer* builds the base for many other improvements as follows and has some other implications that make the use of HTTP/2 much more efficient, as you can see in 3.2.2.

The term *binary framing layer* constitutes the encapsulation and transfer of the HTTP messages between client and server in binary coded *frames*, which are the smallest communication unit in HTTP/2.

Frame

Every frame begins with a fixed header of nine octets as its frame header, followed by a variable length payload, only limited by the receiver's maximum frame size (SETTINGS_MAX_FRAME_SIZE)

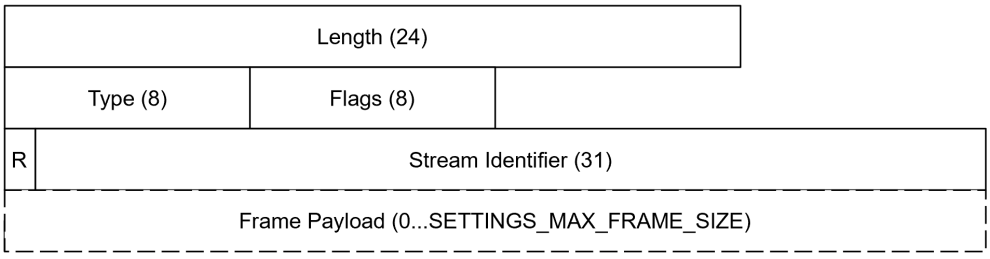

Figure 2.3: Frame Layout

Header Field	Explanation
Length(24)	An unsigned 24 bit integer expressing the payload size Maximum value=2^{24} octets
Type(8)	Pre-specified frame type
Flags(8)	Frame-type specific Boolean flags
R(1)	A reserved 1-bit field, undefined yet
Stream Identifier(31)	An unsigned 31-bit stream identifier
Frame payload	Up to size of *length* field in octets, 2^{14} at minimum Can take up to $2^{24} - 1$ octets, if advertised accordingly by receiver

Table 2.1: Header fields

The frame space of 8 bit theoretically allows 256 different types of frames. As values between 0xf0 and 0xff are reserved for experimental use and the value 0 is transparent for check sum operations and therefore not used, leaves 240 values for frame types, of which 10 are defined in the official RFC [BPT15] as follows.

Frame Type	Code
DATA	0x0
HEADERS	0x1
PRIORITY	0x2
RST_STREAM	0x3
SETTINGS	0x4
PUSH_PROMISE	0x5
PING	0x6
GOAWAY	0x7
WINDOW_UPDATE	0x8
CONTINUATION	0x9

Table 2.2: Frame types

Each *message*, what conforms to a request to the server or response from it, is split up into frames, interleaved with frames from other streams and reassembled on the receiver's side by using the *stream identifier* in its frame header.

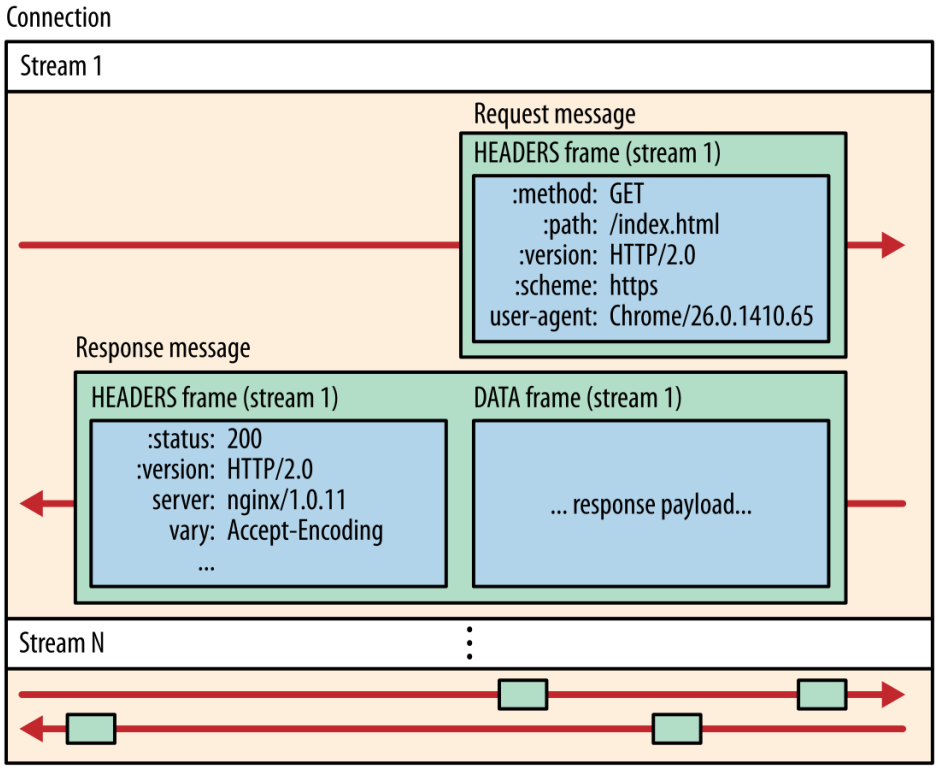

Figure 2.4: The relation of streams, messages and frames

Those messages, in turn, belong to a *stream*, whose amount is only limited by the size of the *stream identifier*, an unsigned integer of 31 bits, meaning $2^{31} - 1$ (2147483647) streams can coexist on a single TCP connection.

This technique is called *stream multiplexing*. By means of this, HTTP/2 is able to transfer multiple objects over one single connection, as depicted in figure 2.4, instead of having to open a new connection for each resource to transmit in parallel as you have to with HTTP 1.1, even if the costs for these kind of operations are contained thanks to TCP fast open.

HPACK header compression

Completely detached from the binary framing layer, both regarding its specification and scope, is a technique to compress message headers, named *HPACK*. As pointed out, besides the increasing payload over the recent years, also the size of the header massively increased, mainly due to additional fields in the header. Those additional header fields are added for various different reasons, mostly though for cache optimization or security reasons. If above all cookies are used, a request quickly exceeds the size of the initial TCP window (as described in 2.1), slowing down the whole process as a complete round-trip is needed to fulfill the whole request.

As HTTP is a stateless protocol, all header values necessary for the website to operate correctly have to be transmitted with each and every request. For this reason, HPACK makes use of a dictionary which is maintained on both sides, meaning it has identical content. If a header name and/or value that is known to be inside this dictionary needs to be encoded, merely its index needs to be transferred. Depending on the size of the dictionary (and hence the amount of entries), an index is usually represented by 1 byte (256 entries) or 2 bytes (65535 entries). Yet, theoretically the amount of entries is only limited by the maximum table size, which houses all keys and values.

The first 61 entries, however, are a static list, defined in the specification [PR15], of the most commonly used header fields, while the rest of the table is extended in a dynamic manner with each request and response with keys and/or values that do not yet reside inside it. Those string keys and values, representing the lion's share in size and occurrence, which travel over the wire for the first time or are marked not to be indexed are *at least* entropy encoded using a static Huffman code. The code used is able to achieve a maximum compression ratio of 8:5 or 37.5% still [Kra].

Although the HTTP/2 headers are smaller than HTTP1.1's or SPDY's for first time requests, HPACK shows its full potentially not until consecutive requests, when - as usually - mostly the same header data (keys and values) are transmitted. For instance, a 2 kB cookie can then be represented by only a 1 byte index, what equals to a compression rate of 2048%. However, also first time requests and responds benefit from the dictionary, as most of the header keys and even their values in a typical request are covered by the static table.

For more details on the operating mode of HPACK please consult the work of Pu [Pu16] or the official HPACK RFC [PR15].

Stream Reset

If a request or response is sent via HTTP 1.1, there is, aside from interrupting the connection and having to reestablish it in a costly manner, no way of stopping a once started transfer. HTTP2 introduces a *reset stream*-frame *RST_STREAM*, a new type of frame for doing exactly what the name suggests: To reset the stream and start anew instead of wasting resources for a pointless operation.

Flow Control

As already introduced in 2.1, flow control is not only used on the transport layer, it is essential on the application layer to control both the flow of streams inside the multiplexed HTTP2 connection and of the connection itself. This is necessary, as the use of multiple streams within one connection introduces contention over use which may result in blocked streams. The current receive window is advertised via the aforementioned *WINDOW_UPDATE* frame. The standard itself defines only a set of rules, as following, and leaves the detailed implementation open, letting the implementor "select any algorithm that suits their needs." [BPT15]

- Each receiver may choose its own appropriate receive window size
- Flow control is credit based: When a packet is sent, the remaining size is decremented by the packet size
- Flow control is mandatory, it can not be disabled
- Flow control works hop-based, not end-to-end
- Flow control only affects *DATA* frames in order to ensure the reception of control frames

Stream Prioritization

By assigning and an integer weight to a stream and a dependency of one stream to another, HTTP/2 allows the client to define in which order it prefers the requested resources to be delivered by the server. The server *can* use this priority and dependency information to assign its system resources and bandwidth accordingly. The desired priority of a single stream is expressed by an 8 bit integer, allowing to set a value between 0 and 255. Due to a 1 bit offset this assignment results in an effective weight between 1 and 256. Combined with the dependency to another stream, the client is able to construct a *prioritization tree*, as every child inherits its parent's priority value. However, the desired weight and dependency is not obligatory for the server. It may hence choose to process and deliver the resources in a different order as requested.

Server Push

The only real innovation that was not introduced to cope with design flaws, from today's point of view, is *Server Push*. This term denotes the ability to send additional resources to the client in response to a single request. To do so, the resource is announced via *PUSH_PROMISE* frame containing the URL of the resource alongside other header entries and the stream id to use prior to the parent's response. Its intended use is to push data to the client it will most likely need, for instance dependent resources of a requested website. The client remains in full control, meaning a stream can be rejected by the client via *RST_STREAM* frame in case the client has the resource to push already cashed, be prioritized and flow controlled. Above that the client can also completely deactivate or reactivate this feature at any time via *SETTINGS* frame.

2.4.3 HTTP/2 compared to HTTP 1.1

This section's purpose is to show the intended use of the improvements of HTTP/2 to cope with the insufficencies of HTTP1.1 that were outlined in 2.3.3.

Several improvements derive from the nature of a binary protocol instead of a text based one. This embraces an overall faster and less error prone processing by avoiding the ambiguity of a text format, which needs to be parsed, while protocol and framing parts with HTTP/2 are completely separated and also smaller in size by omitting white spaces. Above that, does the binary framing build the base for *multiplexing* and thus having an almost arbitrary amount of streams within a connection, which then can be be controlled individually as will be depicted in the following.

Apart from a better network utilization by avoiding the overhead of multiple connections, *sharding* becomes futile, as it is now possible to exploit as much parallel streams as necessary to retrieve the referenced resources. This effectively bypasses the Head-of-Line blocking problem on application level, as the download of one resource does not have to wait for another one to complete. Additionally, single streams can now be prioritized according to their importance. By avoiding the costs for establishing new TCP connections including the impact of TCP slow start for every new host and reusing connections instead, the network latency decreases while the throughput increases. Admittedly, in case of employing a ***c****ontent* ***d****elivery* ***n****etwork*, an additional connection to its servers is indeed mandatory and the web applications currently running on HTTP1.1 have to be adapted to make use of as few hosts as possible in order to benefit from multiplexing. Also the head of line blocking problem may of course still occur on the TCP layer below due to retransmission of lost packets, what seems to be the only Achilles heel of HTTP/2, as packet loss naturally affects a single connection more than if only one of multiple connections is bothered.

Anyway, the need for optimizations like *Inlining*, *Concatenation* and *Spriting* are already diminished by the omission of the connection limit, as all files can be requested in parallel and the transmission in binary is more efficient. Moreover, the requests for resources, now unbundled and uninlined, can be prioritized in the order they are needed to render the page or even be pushed to the client preemptively alongside the response to the initial request for the Web page referencing them. Also, having the resources *atomic* make them easier to cache, as a cache invalidation only concerns individual files, not the combined one. For all of the above mentioned techniques, this is expected to lead to a reduced page load time as only required resources are transferred and the page is not rendered until at least the structure in HTML form is loaded in its entirety. Omitting Inlining even leads to smaller files, as the embedded data does not need to be inefficiently base64 encoded. For website creators this entails a lighter build process as the Inlining can be omitted while the same website has a smaller memory footprint as the *prefetched* data does not have to be kept enitirely in memory.

As pointed out in 2.3.3, also the typical request massively increased in size since the introduction of HTTP 1.1, mainly due to a larger amount of header attributes and the extensive use of cookies, often exceeding the initial request window and hence making an additional round-trip necessary in order to deliver it to the server in its entirety. For this reason, HTTP/2 makes use of HPACK to compress the size of the header, making it once again fit into the initial window and thereby avoiding an additional server round trip. Also the protocol efficiency and latency in form of a reduced transmission delay is improved as overhead is minimized by HPACK, merely having to transmit table indexes for recurring entries or at least Huffman entropy encoded content for new entries.

2.5 THE PROXY PATTERN

A proxy is, generally speaking, an entity acting on behalf of something or someone else. This pattern is also widely applied in various disciplines of computer science, like in software architecture and networking. The latter constitutes the context this work relates to: The proxy server. It serves, as the name suggests, as an entity between the subject (the accessing part) and the object (the accessed part) in order to fulfill one of the following goals: Protection of either part, adding or hiding functionality to or of and from either side, caching or transforming content, logging, filtering or bypassing filters, protocol adaptation, load balancing and, last but not least, anonymization.

2.6 SANE

The name of the platform that serves in this work for the proof of concept to demonstrate the progress of HTTP/2 with regard to the proxy pattern is an acronym for **S**erver **A**ccess **N**etwork **E**ntity. It represents a platform for providing arbitrary crowdsourcing services and can moreover be extended with additional, project related methods to satisfy customization needs. It has been developed by Tenshi Hara in the context of his thesis [Har12] with regard to the MapBiquitous project, an integrated system for location based services, but is not limited to it. Instead it was designed with a general application for crowdsourcing in mind. Its main purpose is to free the crowdsourcer from recurring tasks like user management. Beyond that it serves as a proxy between the users and the crowdsourcing server to provide anonymization in the first place, but also all the other advantages of using a proxy, such as improved scalability and load balancing if required.

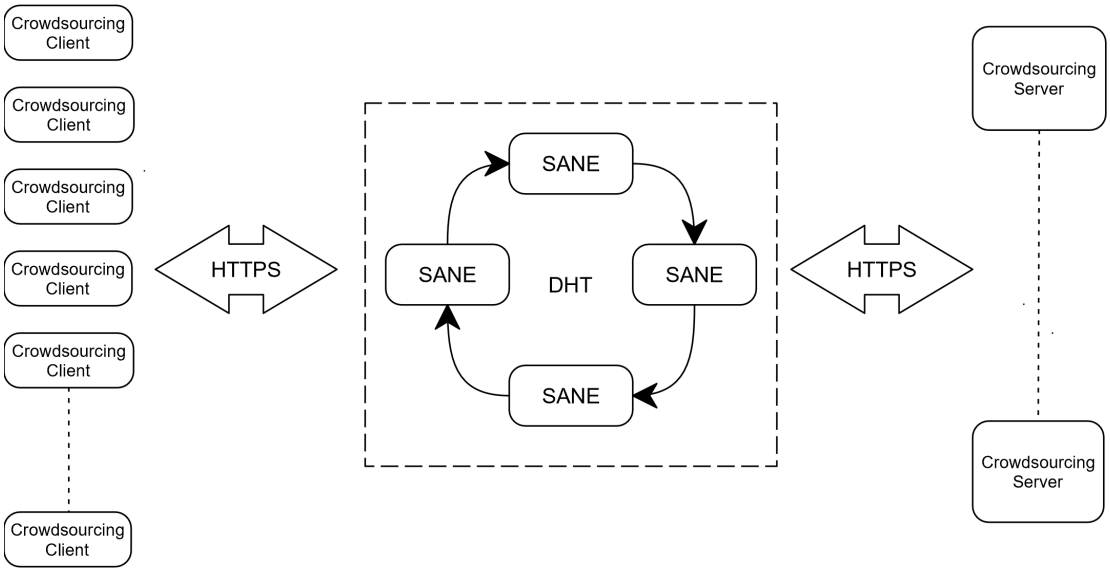

Figure 2.5: Crowdsourcing platform architecture (following [Pu16])

In order to provide anonymization, all user contributions are forwarded to the crowdsourcing server by using a unique submitter id, which is only known to SANE. To ensure scalability and fault tolerance it is designed to work with arbitrary amount of instances inside a self organizing **D**istributed **H**ash **T**able, with every instance being responsible for a equally distributed area of a 256 bit SHA1 hash representing a submitter. For the sake of completeness it shall be mentioned that the data is kept in a MySQL database. The frontend interface is a REST API communicating solely via HTTPS.

3 RELATED WORK

“

The statistics about reading are particularly discouraging: The average software developer, for example, doesn’t own a single book on the subject of his or her work, and hasn’t ever read one.

Tom DeMarco

”

This chapter takes a look around the academic works regarding HTTP/2. Although, due to its relative novelty, the amount of scientific papers with this topic is relatively low, only those which either serve as a basis to build on or treat the same subject to narrow down the scope are consulted in the following.

3.1 HTTP/2 IN A NUTSHELL

The following secondary literature alongside the official RFC standard documents of HTTP/2 itself [BPT15] and the HPACK RFC [PR15] have been used to compose the overview over HTTP/2 in 2.4, which is required to comprehend the related work and this thesis itself presented in the following.

The probably most comprehensive assessment of HTTP/2 [Ste14] comes from the Mozilla employee and developer of curl and libcurl Daniel Stenberg. As he has actively been involved in the specification of HTTP/2 in the HTTPBis group within the *I*nternet *E*ngineering *T*ask *F*orce, he gives a very broad overview of HTTP as it is used today, the specification process of HTTP/2, its inherent concepts, its expected impact, its implementation in the widely used webbrowsers Firefox and Chrome and naturally also the implementation in his own child curl.

From Ilya Grigorik, self-titled *Internet plumber*, co-chair of W3C and Google employee who also participated in the development of SPDY, comes a quite extensive book about *high performance browser networking* [Gri13a] in which he devotes one chapter to HTTP/2, especially addressing its common base with the SPDY protocol, the binary framing layer, stream multiplexing and how it is realized in detail, Flow Control, Server Push, Header Compression and the upgrade mechanism from HTTP 1.1 to HTTP/2. [Gri13b]

Another assessment with a more practical approach and target group comes from developers behind the web server and reverse-proxy NGINX. This white paper gives an overview of the differences of HTTP/2 compared to HTTP 1.1 and how the optimizations, that were necessary to cope with the drawbacks of HTTP 1.1, have to be treated in order to unlock the full potential of HTTP/2. Namely these are *domain sharding*, the use of *image sprites*, the *concatenation of client code files* and *Inlining* as mentioned in 2.3.3 [NGI15]

3.2 QUANTIFIABLE ASPECTS

The following works address measurable aspects of the use of HTTP/2 compared to HTTP 1.1.

3.2.1 Current adoption across the Web

In [Var+16] Varvello et al. report about their measurement platform that monitors HTTP/2 adoption across the top one million websites on a daily basis according to the web traffic analysis service Alexa. They found that 68,000 domains *report* to use HTTP/2 via ALPN of which only about 10,000 domains actually do. Those *true* numbers, as entitled by the authors, are referred to in the following. Since the date of publication in October 2015, the amount of websites supporting HTTP/2 increased dramatically as the website [Var+] Varvello et al. created during the implementation of their study indicates. The amount of websites using HTTP/2 rose up to 148,612 websites, equaling 14.9% of all websites in the scope, until the evaluation has obviously been discontinued in November 2016, as since then no more statistics have been released. Interestingly, the steepest increase can be observed in mid December 2015, where

the amount of websites actually supporting HTTP/2 climbed from about 25,000 websites to almost 75,000 websites in a matter of only days. This sudden spike, however, can be traced back to updates of their measurement methodology that have been conducted at that time. With the first update also websites that redirect clients from top level domains via HTTP 1.1 to their *www* sub domain, actually serving HTTP/2, are included. With the second update the measurement software makes use of the *Server Name Indication* TLS extension when probing websites. Without the use of this extension "(...) some HTTP/2 sites (e.g., those using virtual hosts) do not indicate HTTP/2 support in their ALPN responses." [Var+]

Their findings indicate that most of the websites that make use of HTTP/2 still use practices introduced to overcome the drawbacks of HTTP 1.1, like the before mentioned domain sharding and the use of image sprites. According to the authors however, these workarounds make HTTP/2 more resilient to packet losses and jitters. Altogether, they find that 80% of the websites effectively supporting HTTP/2 experience a reduction of the page load time, compared to its predecessor HTTP 1.1.

Beside the study above from an academic origin, Web companies create their own statistics about HTTP/2 adoption across the Web. Due to the discontinuation of the Website belonging to the study cited above, additional sources from [W3T] are consulted, which refer to the top 10 million websites listed in Alexa. Due to the different scope and potentially different evaluation method, these numbers are unfortunately not directly comparable to those from the study above. According to [W3T], 6.7% at the beginning of their evaluation in March 2016 almost linearly, exceeding 10% in September 2016 over 10.3% in November 2016 to 12.9% to date make use of HTTP/2. If anyhow put into relation to the numbers from the study above, the numbers suggest that the *bigger* sites, expressed by their Alexa ranking, tend to implement HTTP/2 sooner, what coincides with the findings of [Var+16].

According to recently released statistics of the Content Delivery Network KeyCDN [Jac] their HTTP/2 traffic, reflecting client usage, rose from 51% mid October 2015 to 68% in mid April 2016. Although this means that most of the HTTP traffic meanwhile is HTTP/2, this suggests that still many browsers are not up to date, as if otherwise they would make use of HTTP/2.

3.2.2 Performance

In [SOC15] de Saxcé et al. try to answer the question *Is HTTP/2 Really Faster Than HTTP 1.1* asked in its title by conducting three kind of tests, of which all are using page load time as metric.

The first test measures the time it takes to download a photograph of about 1 MB in size, which is split dynamically into a certain number of parts of equal dimensions according to the number of used streams; the first run retrieves the picture in a single request, while in the last run it is split into 100 parts. The total amount of time to request the image is measured for each configuration.With HTTP 1.1 the page load time almost doubles over the 100 requests, where it almost remains constant with H2. This result indicates that the Head-of-Line blocking issue mentioned in 2.3.3 is responsible for the prolonged requests of HTTP 1.1 when compared to HTTP/2.

In the second test, which is conducted on the Internet instead of a local environment, the top 15 websites, again according to Alexa, are tested in terms of page load time. From this point of view, the authors state that simple websites do not gain much from HTTP/2. Bigger websites, containing lots of pictures, however, benefit greatly as indicated by reduced page load times of 20% in average, spiking at 48%. "This decrease is mainly due to the multiplexing as we noticed with the dummy web page", as the authors state. Recapitulatory can be said that the page load time has been tested being constantly lower with HTTP/2 than with HTTP1.1. This experiment has been repeated employing a 3G mobile network, where the average round-trip time was

around 400 ms. The page load time decreased - compared to HTTP 1.1 - also by 20% on average. Due to a unusually low packet loss rate, the performance was better than expected.

The third test was conducted, in order to examine the influence of latency and packet loss, again in local area networks. That way, it is possible to keep these elevating screws under control and eliminate other possible distorting network parameters. They found out than increasing latency *widens* the difference in page load time between HTTP 1.1 and HTTP/2, meaning it is able to cope better with high latency. This, in turn, suggests that HTTP/2 might perform better than HTTP 1.1 in mobile network use cases, where latency is naturally higher than in cable-based networks.

The other part of this third test was to examine the page load times - again with mobile networks in mind - of HTTP/2 regarding packet loss. This benchmark was conducted with fixed latency and a varying packet loss rate where it indicated a completely different result than the previous: The higher that packet loss was, the smaller were the benefits from HTTP/2. Moreover, the page load time ratio between HTTP 1.1 and HTTP/2 exceeded 1, meaning it actually took even longer than using HTTP 1.1. This was ascribed to the behavior of HTTP/2 of using just one single connection: If this single connection suffers from packet loss, every multiplexed stream suffered in proportion to its connection load, whereas with multiple TCP connections between client and server, the latter was simply able to mitigate the packet loss issue.

The developers behind the "ultimate in-browser HTTP sniffer" HttpWatch conducted and published their own performance comparison of HTTPS, SPDY and HTTP/2. Although pre-final standards of SPDY (3.1) and HTTP/2 (draft 14) were used for the test, the benchmarks give an idea how the individual protocols perform, especially as the the final version and RFC respectively only differ regarding Flow Control, Server Push and security considerations being virtually irrelevant to performance. [HTT]

According to them, HTTP/2 outperforms both SPDY and HTTP1.1 (using TLS) in terms of request and response header size. This measurement is conducted by performing a request to a Google beacon, returning an answer with only a header and without any content. The compression ratio of the specifically designed HPACK algorithm put to use is higher than the *deflate* algorithm of SPDY and, naturally, better than no compression at all with HTTP 1.1. Adding text to the request, however, turns the tide: For pure text the deflate algorithm of SPDY performs better, returning smaller responses. The authors attribute this to padding bytes added to HTTP/2 *DATA* frames for security reasons. These padding frames are necessary to obscure the size of the content within, mitigating *BREACH* attacks to which SPDY is vulnerable to. For image data however, where no such padding data is added, HTTP/2 again outperforms both SPDY and HTTP1.1. Another aspect [HTT] took a look at is the number of required TCP connections and TLS handshakes during page load. In this very discipline, SPDY and HTTP/2 perform equally due to multiplexing, which both protocols use, while HTTP1.1 tries to use multiple connections to improve concurrency. In their fourth test, which regards page load time, again HTTP/2 wins, followed by SPDY and HTTP1.1.

To put it into a nutshell, HTTP/2 outperforms all other protocols in terms of page load time, latency, header- and content compression, especially when taking the additional security measures into account.

3.2.3 Energy Efficiency

The topic energy efficiency has become more and more important over the recent years. This is less rooted in the also quite recently emerged idea of generally avoiding dissipation of any kind, than the shift from desktop computer usage to the use of devices for *ubiquitous computing* in the form of smartphones nowadays and possibly other devices in the coming *Internet of Things* era. This rise of importance does not only apply on the client side, which is increasingly incurred by the mentioned mobile clients, where energy efficiency directly affects the usage time

between recharging, but also on server side to minimize energy consumption in data centers, which multiplies with regard to cooling, as every additionally used Watt for computing entails in average two Watts for cooling. The former is addressed by [CSH15] in which Chowdhury et al. try to answer the homonymous question *Is HTTP/2 more energy efficient than HTTP 1.1 for mobile users* by measuring the exact energy consumption of (four) Galaxy Nexus Smartphones running scripted workloads on Mozilla Firefox Nightly builds for Android that represent typical real world scenarios using either HTTP 1.1 or HTTP/2. These workloads comprise the following scenarios: **(a1)** Downloading images from the static photo gallery generator *fgallery* running on a H2O web server, **(b1)** downloading 180 tiled images with added artificial latency from a web server written in the popular open-source language *Go* and **(c1)** accessing the public mobile websites of Google and Twitter, which use HTTP/2 to its full extent. Wherever feasible, the authors conducted every test of HTTP 1.1 with enabled TLS, HTTP 1.1 without TLS and HTTP/2 with TLS enabled by default in order to supply comparable results. As *Gopher*, the Go-based server, does not support HTTP 1.1 with TLS and the publicly available websites do not support HTTP 1.1 anymore at all, these tests had to be omitted. The test bed used here is called "*Green Miner* - a continuous testing framework similar to a continuous integration framework but with a focus on energy consumption testing" [CSH15]

As long as there were no latency differences, HTTP 1.1 without TLS clearly outperforms both HTTP 1.1 with TLS and HTTP/2 with TLS enabled by default. The latter two performed almost equally in terms of energy efficiency. This can be explained with the at least one additional round-trip necessary to establish a secure connection and the higher CPU usage due to encryption. The picture changes as soon as there are packet latencies involved: Already above a threshold of 30 ms, the difference in energy usage decreases. With a latency of 200 ms and above, HTTP 1.1 already consumes more energy than HTTP/2, especially with a higher number of TCP connections and a high number of objects to retrieve. The energy consumptions increases even more the higher the latency becomes. Summing it up, Chowdhury et al. state that HTTP/2 never performs worse than HTTP1.1 and will help to save energy particularly in mobile networks, which tend to suffer from a higher latency than wired networks. [CSH15] Altogether, they answer the question *if HTTP/2 saves energy* with a clear "Yes, when round trip times are above 30 ms and TLS is being used (...)" [CSH15] and HTTP 1.1 becomes expensive for a large number of TCP connections, leading to the overall conclusion that HTTP/2 will be more mobile user friendly and HTTP/2 should be adopted in order to save energy"

In the latter, in [SH16] two co-authors of the just discussed paper examine the impact of the used protocol version on energy consumption of web servers. To do so, they also use the Green Miner framework to monitor the power consumption, but instead of the Galaxy Smartphones as clients a Raspberry Pi is used for running the web server instances under assessment. As a client either another Raspberry Pi or a typical business laptop with four CPU cores are used. For the test Sapra et al. employ a similar, server side software setup as in **(a1)** in the work above, but extend the test of *fgallery* to the Java-based web server Jetty. Their stated objective is to emulate the client-server communication as close as possible to real-world scenarios. They ensure this by relying on recent studies that determined the average size and amount of individual objects to be fetched during a typical round-trip on a multimedia website nowadays. Again, three different experiment sets are conducted: **(a2)** Energy performance evaluation under HTTP 1.1 / HTTP 1.1 with TLS enabled / HTTP2 (with TLS enabled by default), **(b2)** The energy consumption behavior of HTTP/2 with stream multiplexing enabled and **(c2)** the server energy consumption under HTTP 1.1 and HTTP/2 with varying latencies over the network. When HTTP/2 is to be tested, the authors employ the newly introduced Server Push feature in order to push resource files like images and cascading style sheets to the client referenced in a HTML page they belong to. The underlying idea of doing so is to preemptively push resources that are required anyway to the client to stint server round-trips. Again, the results show that HTTP 1.1 with TLS enabled performs the worst in terms of energy efficiency. The results of this paper draw a similar picture to the previous one: In the absence of latency, HTTP 1.1 needs less energy for the same tasks as HTTP/2. Again, the tables turn as soon as even a small latency is involved.

Concluding both of the studies above, it is safe to say that HTTP/2 performs significantly better in terms of energy consumption, especially with high latencies, as they are very common in mobile networks. [SH16]

3.3 SERVER PUSH

[HHQ15] also concerns the use of HTTP/2 in mobile applications but with a slightly different emphasis in the first place: To reduce the data usage and thereby indirectly the amount of energy used for information transfer. The shorter or less frequently the mobile's radio is used, the less energy is consumed. Besides the intended, straightforward use of Server Push to provide the client with all the information it potentially needs and the use of server hints, where the client is informed of a URL where resources are located it may need in the future, the authors of [HHQ15] propose a technique they named *MetaPush* with which they "address a key challenge of minimizing PLT while avoiding unnecessary data transfers" [HHQ15] To achieve that goal, they employ Server Push to transmit *meta files* to the client, which contain resource URLs (i.e. hints), which are to be requested later by the client. In the first *fetch phase* as the authors call it, the clients conduct a request for some or all resources that are specified within the *meta file*. These requests are combined with the request of the page itself in order to download the whole content needed to display that page at once in only one round-trip. According to the authors, this decouples the network transfer and the local computation in order to break the *load-parse-load* dependencies among the objects, what leads to a reduced overall page load time. This way the authors were able to achieve a reduction of page load time by up to 45% while reducing energy consumption by up to 37%. Although this concerns a specifically optimized and specialized case, it shows the potential of applications of HTTP/2's Server Push.

Beyond using the HTTP/2 Server Push mechanism as intended to preemptively deliver resources associated with Web pages to speed up page load and save energy due to using the network and therefore its hardware components more efficiently, several studies examined diverting the Server Push feature from its intended use to extend it for streaming video content: To name the most important, Wei et al. examined Server Push's general potential for video streaming to reduce latency [WS14b], eliminate redundant requests [WS14a] and also to save energy in mobile applications [WSX15]. All of those studies found out Server Push to be promising to achieve the mentioned goals with only minor exceptions for edge cases. Based on those findings Xiao et al. designed an *adaptive push mechanism*, which is put to use to handle those edge cases to avoid increasing the data volume again due to unnecessary pushes, what they call *Over-Pushing* [Xia+16]. The insight from this studies, to relate it to the the case at hand, is that the Server Push feature can be used outside the comparatively static scope for which it was designed in the first place to improve the overall performance. The same holds for the use of Server Push in this work, as will be further described in the following.

3.4 SANE AND HTTP/2

Another thesis, recently written at the TU Dresden [Pu16], also treats the usage of HTTP/2 in proxy settings, yet with a focus on data compression proxies using the newly introduced header compression HPACK. He implements data compression on SANE as a Proof of Concept following two different approaches. On the one hand he follows a more general approach using the HTTP/2 proxy *nghttpx*, on the other hand he uses the native web server capabilities (of Apache and NGINX) and employs curl to implement header compression between SANE and the destination crowdsourcing server. In the following evaluation, Pu measures the gain via the metrics of *header compression ratio* and response time pertaining to the SANE. Both metrics show an improvement by the employment of HTTP/2. The results of this work will be discussed more detailed during the evaluation chapter, as they premise a certain understanding of the SANE's inherent concepts introduced in the next chapter.

As this thesis already treats header compression and HTTP/2's general performance, these metrics are not re-measured. However, they are re-evaluated due to new discoveries during this thesis. Above that the terminology and some findings are taken on in the the following.

3.5 SUMMARY

Almost two years after its introduction, HTTP/2 is already well-adopted among the bigger sites and is still increasing. However, many of the sites already using HTTP/2 still employ HTTP 1.1 optimization techniques. The performance of HTTP/2 regarding page load time, latency, header- and content compression was found to be better than the of HTTP 1.1 and SPDY in every case. Also in matters of energy efficiency HTTP/2 outperformed all other protocols, especially with high latencies, which usually occur in mobile networks. HTTP1.1 only beats HTTP/2 in terms of energy efficiency without TLS encryption, which is virtually obligatory for HTTP/2. By actively optimizing network utilization under use of Server Push energy efficiency can even further enhanced while shortening page load times. Above that, related works show that this feature can be employed aside from its intended purpose, even for time-critical applications.

Generalized Impact of HTTP/2

Generally speaking, based on the preliminary pre-assessment in 2.4 and findings of related work, the biggest impact derives from the introduction of binary framing and the ability to multiplex streams within only one connection. This not only solves the head of line blocking problem elegantly and implicitly, it also allows more resources to be loaded in parallel, even compared to HTTP1.1 with sharding, and bears less protocol overhead. In combination with HPACK header compression this results into less data to be sent over the wire. Less data naturally requires less time to complete the transfer and also leads to a lower network load for the exactly same result as with HTTP1.1. Moreover entails the binary framing the omittance of parsing and thereby also the ambiguity it brings along and a lower CPU load.

This effect, however, may be outweighed by the employment of the, in terms of CPU load, more expensive header compression and TLS encryption, which is enforced by browser developers. Virtually all browsers only support HTTP/2 combined with enabled TLS encryption. The positive aspect of this constraint is the increasing spread of encryption due to mandatory TLS usage, what boosts security for every Internet user on the long run.

Certainly, to really benefit from the advantages, the websites have to be optimized for the use of HTTP/2 by their developers and administrators. Effectively, they have to reverse the optimizations that were conducted for HTTP1.1 in order for the above mentioned arrangements to work, as Inlining and Spriting promise no performance gain anymore and Sharding will likely even be detrimental, as it increases the total amount of connections used instead of decreasing them by the use of multiplexing. At least the *data providers* have to adapt their development processes for future projects that are about to use HTTP/2. As a result their development processes are going to be simpler, as the former optimizations require additional steps during development, like merging the code and images for Inlining/Spriting and deploying the resources over several domains for Sharding. Above that, the new *Server Push* feature can be put to use either to accelerate page delivery for static content or optimize connection utilization for dynamic content, as streams do not have to be kept in an open or half-closed state for the duration of pending operations.

To put it in a nutshell, if developers adapt to HTTP/2, the user's overall web experience is expected to be faster and safer, network load will decrease and the developers' boilerplate tasks are thinned out due to a more straightforward deployment.

4 CONCEPT

“

Computer system analysis is like child-rearing; you can do grievous damage, but you cannot ensure success.

Tom DeMarco

”

This chapter shall evaluate the individual advancements of HTTP/2, which have been introduced and pre-assessed in 2.4.2 and further backed up with related work in chapter 3 whether they are promising, have no or negligible impact or may even bear disadvantages over the use of HTTP 1.1 for the proxy setting using the example of the SANE. In the following the necessary changes to the available software and criteria to measure their gain are being carved out.

4.1 SANE BASICS AND PRINCIPLES

To be able to decide which of the improvements of HTTP/2 are advantageous for SANE and how to implement them, it is necessary to understand the structure and principles it is making use of.

4.1.1 Architecture configurations

As new technology takes time to establish, especially when based on a standard, what usually leads to a inhomogeneous software landscape, it is not guaranteed to function as intended. This is either due to configurations errors, as could be seen in 3.2.1 where web servers announced to support HTTP/2 but in fact did not, or simply outdated software. In case at hand this may apply on web browsers or diverted client software updates just as outdated web servers or libraries, why a fallback solution has to be held available: So, if one of the involved peers Client, Proxy and Server in the chain does not support HTTP/2, the preceding protocol HTTP1.1 is used. This leads to the following four possible architecture configurations, where the terminology of [Pu16], which treats a quite similar issue, is incorporated. As the proxy server terminates connections to it and employs yet another connection to forward the just received and altered data to the respective server, the link between Client and Proxy ($\mathcal{C} - \mathcal{P}$ in the following) as well as the link between Proxy and Server ($\mathcal{P} - \mathcal{S}$ in the following) can be considered autonomous, which differ in requirements and characteristic as will be pointed out below.

Status quo

Currently, the entire communication is based on HTTP1.1, both for $\mathcal{C} - \mathcal{P}$ and $\mathcal{P} - \mathcal{S}$. This configuration serves as point of origin and fallback if both the Client and the Server do not support HTTP/2.

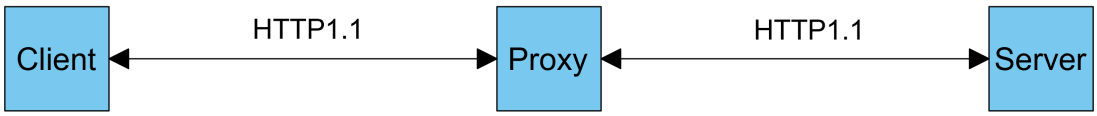

Figure 4.1: Status quo

Upgrade Proxy

This configuration is called *upgrade proxy*, because it *upgrades* an incoming HTTP1.1 connection to outgoing HTTP/2. This configuration comes into effect if the client does not support HTTP/2. This might be the case if the used client library does not yet support HTTP/2 or it has not yet been *switched on*. For the implementation of the proxy, this means it has to be able to support incoming HTTP1.1 connections still and *convert* them into HTTP2 requests it transmits to the server on behalf of the client.

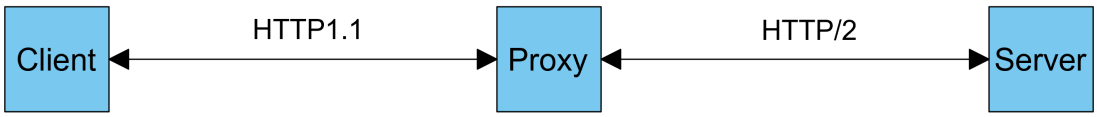

Figure 4.2: Upgrade Proxy

For this case, the partial link $\mathcal{P} - \mathcal{S}$ has to be implemented using HTTP/2.

Downgrade Proxy

This configuration is called *downgrade proxy* because it downgrades incoming HTTP/2 to HTTP1.1. It takes effect if the client is able to perform HTTP/2 requests, but the crowdsourcing server is unable to digest them, because it has not yet been updated or is running legacy software, whose development has ceased.

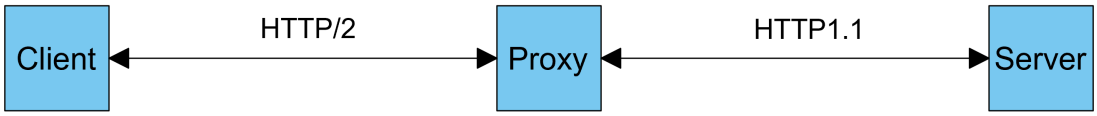

Figure 4.3: Downgrade Proxy

For this case, the partial link $\mathcal{C} - \mathcal{P}$ has to be implemented using HTTP/2.

Straightforward Proxy

This configuration is the ideal case, as both the incoming request and the request the proxy transmits to the server are both HTTP/2, what means all the improvements of HTTP/2 can be put to account.

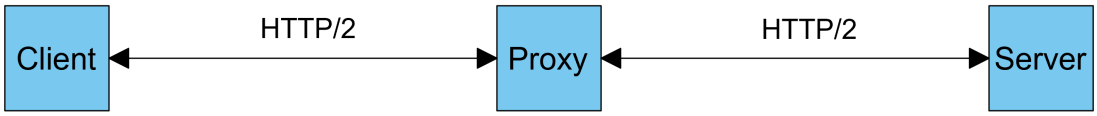

Figure 4.4: Straightforward Proxy

For this case, the partial links $\mathcal{C} - \mathcal{P}$ and $\mathcal{P} - \mathcal{S}$ have to be implemented in order for the entire link to use HTTP/2.

4.1.2 SANE method range

The methods of the SANE can be divided in management methods, which are processed only on the SANE proxy and crowdsourcing methods, which usually get processed on the proxy and trigger another request to the crowdsourcing server. Its response is in turn again processed on the SANE proxy before and triggering another response to the client.

Management methods

As pointed out, those management methods have only a local effect. They do not employ the partial link between proxy and server ($\mathcal{P} - \mathcal{S}$). These methods are not specific to campaigns. Instead they are used to manage general user- or campaign data or the SANE itself.

Crowdsourcing methods

Unlike the SANE management methods, crowdsourcing methods both have a proxy-local effect, for instance registering the user's submissions and an effect on the crowdsourcing server. A typical crowdsourcing request is first processed by SANE and *forwarded* to the crowdsourcing server. This forwarding is essentially another, new request to the destination server; its result gets usually again processed on the SANE proxy after receipt and its outcome is then returned to the client. This communication across partial link borders is referred to as *Cross-Link Propagation* in the following. Thus, crowdsourcing methods make use of the entire link between client over the proxy to the crowdsourcing server and vice versa ($\mathcal{C} - \mathcal{P} - \mathcal{S}$ link).

4.1.3 SANE: An application layer proxy

The proxy acts, as already pointed out, as a server to the client and as a client to the crowdsourcing server. The individual connections on the partial link between Client and Proxy are mostly handled by the web server running the SANE proxy itself or the scripting runtime environment on top in special cases, as will be shown below, while the link between Proxy and Server is handled by the script runtime environment using client functions in its entirety. Thus, for the improvements on the $\mathcal{P} - \mathcal{S}$ link, additional implementation is mandatory.

4.2 EXPECTED IMPACT OF IMPROVEMENTS ON THE SANE

As pointed out in previous sections, there are several improvements in HTTP/2 of which some bear the chance of having an impact on the SANE, which serves as guinea pig for the proof of this concept. Whether the impact is positive or even negative shall be evaluated in this section.

The above mentioned requirements, regarding the partial links, depend entirely on the amount of separate data *communication entities* (connections in HTTP1.1 and streams on the single HTTP/2 connection). For this reason, the following will first treat the improvements of HTTP/2 that apply on both link types $\mathcal{C} - \mathcal{P}$ and $\mathcal{P} - \mathcal{S}$ and secondly the improvements that are specific to its link type.

4.2.1 Implicit improvements

Some of those improvements work *out of the box* if a HTTP/2 connection is put to use with no need for additional implementation on the SANE, as they can be attributed to the use of the binary framing layer. This comprises the ability of having an almost arbitrary amount of parallel streams to a host and solving the Head-of-Line Blocking problem implicitly, as pointed out in 2.4.3, just as the better utilization of the TCP layer by reusing a once-established connection, instead of having to establish new ones for every new request or response. Hence, both lead in theory to lower latency data transmissions.

4.2.2 Link-type independent improvements

The following improvements of HTTP/2 apply in any case and are fully independent of the link type, as they do not require multiple streams multiplexed onto one connection to work. They hence are *link-type independent*, meaning they apply on both the $\mathcal{C} - \mathcal{P}$ and the $\mathcal{P} - \mathcal{S}$ partial links.

Header compression

There is nothing that challenges the reasonability of header compression, as it is an essential component of HTTP/2, even though it can be switched off in theory.

As a RESTful API the SANE does not make use of cookies, every possible request, even if SANE's method to upload XML files with a maximum size of 16,384 Bytes resulting in a total size of 21,487 Bytes on the wire is taken as a basis, fits in the smallest receiver's initial window (rwnd) of 29,200 Bytes [1], which is 20 times the maximum segment size (MSS) of 1460 Bytes for Ethernet connections. It is calculated by the kernel itself taking the system wide setting for *net.core.rmem_default*, which is set to 212,992 Bytes, into account. Nevertheless, as the used kernel is commonly put to use in Linux web servers, this value can be considered authoritative. For servers deploying the Windows operating system, the initial receive window is, with 64 kB, even bigger, so that it is safe to say that also on these systems no additional round trip is necessary for any SANE operation. Due to a lack of a MapBiquitous server an exact value of bytes on wire could neither be determined experimentally nor calculated because of a varying size due to header compression. However, it is safe to suppose that the same, every request fitting in the server's initial receive window, also applies for operations on the $\mathcal{P} - \mathcal{S}$ partial link with a considerably smaller maximum payload of 3,243 Bytes.

Hence no gain regarding latency improvements can be expected by avoiding additional round trips, that were necessary if the initial requests total data size exceeded rcwd window size. Yet indeed is the transmission of the smaller payload finished sooner, what could have a minimal influence on the latency - at least in theory.

The other advantage of HPACK is, as described previously, the usage of a static table for the most common fields and a dynamic table for those header fields and optionally their content, which are not covered by the static table. Even if, as can be seen from the excerpts from the network packet analyzer Wireshark (see C.1 and C.2) showing the request- and response headers of an exemplary invocation of a SANE method, almost all used fields are already mapped by the static table, the according values also reoccur with every subsequent request. As they are held server- and client side in a mutually maintained table, merely the value's index has to be transmitted. This applies the connection between client and proxy ($\mathcal{C} - \mathcal{P}$) and to the connection between proxy and crowdsourcing server ($\mathcal{P} - \mathcal{S}$), what renders header compression advantageous in any case.

Stream Reset

The ability to reset ongoing operations and transfers from and to the proxy may come in handy at that point in time when it turns out that the currently running operation or its outcome respectively has become obsolete since its invocation. In this case, the recipient can stop the processing and forwarding to avoid wasting CPU time and bandwidth. Naturally, this only makes sense for operations that potentially exceed the average time for a full round-trip (t_{rtt}) and the time of the request to time out $t_{timeout}$, depicted by the formula:

[1] On the system used for development and testing, please see B for more information on the test setup.

$$Thresh_{SR} = t_{rtt} + t_{timeout}$$

The duration t_{op} of operations exceeding this threshold are *worthwhile* to be canceled. This condition results in the following conditional expression:

$$B_{SR}(op) = \begin{cases} true, & \text{if } t_{op} \geq t_{rtt} + t_{timeout} \\ false, & \text{otherwise} \end{cases}$$

Otherwise the operation would already have completed until the *RST_STREAM* frame reaches its designated recipient.

Server Push

Similar to the methods that are worthwhile to use *Stream Reset*, long-running methods that require maintaining an open connection with HTTP1.1 in order to deliver their results can be modified to exploit the Server Push feature. It can be applied to notify the client or the proxy respectively, when the invoked operation completes.

With HTTP1.1, both links have to maintain an open connection for the entire operation's run-time in order to deliver the result to the requesting entity, as can be seen from 4.5.

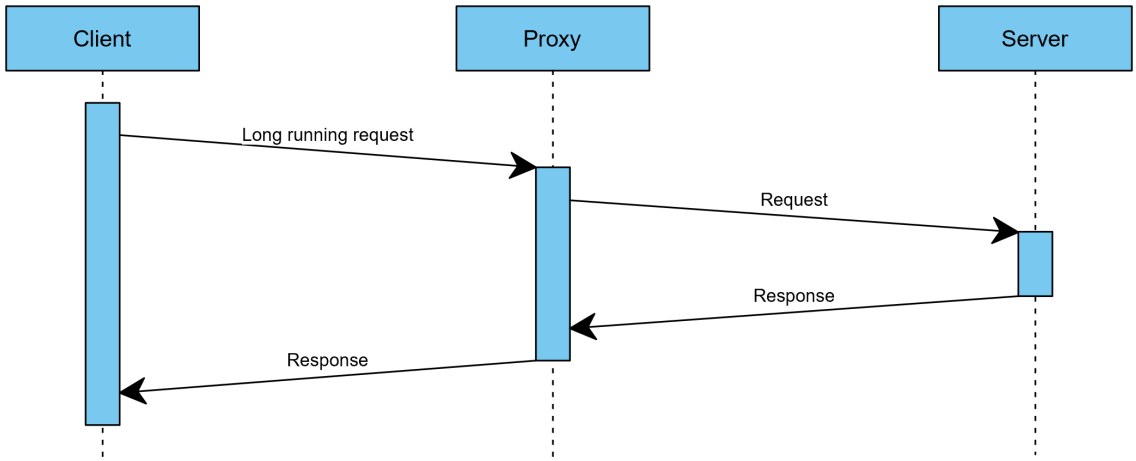

Figure 4.5: Synchronous delivery of results

With HTTP/2 and Server Push, on the contrary, the requesting stream does not have to be kept *alive* for the entire runtime of the operation invoked. Instead, being able to use open streams only when actually transferring data, network resource usage is made more effective.

The implementation of the *Publish-Subscribe* pattern, which is closely related to the *Observer* design pattern of the *Gang of Four*, in contrast, is not possible, as it would require the emission of another *PUSH_PROMISE* frame piggybacked onto the promised stream. This stream, however, is induced server-side, what the HTTP/2 RFC explicitly forbids: "*PUSH_PROMISE* frames MUST only be sent on a peer-initiated stream that is in either the *open* or *half-closed (remote)* state." [BPT15] Due to this relatively vague description this will anyway be further examined in 6.3.1 in the Evaluation chapter.

4.2.3 Link-type specific improvements

As pointed out previously, the application of *Flow Control* and *Stream Prioritization* basically depends on the amount of data transmissions that take place in parallel on a single connection. They hence are *link-type specific*. Whether they are beneficial for the respective link-type or not is evaluated in the following.

Client-Proxy ($\mathcal{C} - \mathcal{P}$) link

Due to the fact that the SANE proxy, as an API, only makes use of one stream inside a connection limits the use of *Flow Control* to the case in which the respective receiver of a transmission has a limited amount of memory for the receive buffer or a limited bandwidth. Then, it can set the receive window accordingly to limit the amount of data sent by the client.

The use of *Stream Prioritization* naturally is superfluous, as there is no need for prioritization if there is only one potentially affected stream, as it does not take effect on other streams outside the connection in question.

Proxy-Server ($\mathcal{P} - \mathcal{S}$) link

In difference to the SANE proxy having many incoming connections from clients, it needs to communicate with only a small amount of other SANE instances or crowdsourcing servers. Here, the SANE proxy combines all incoming request that are relayed to one specific crowdsourcing server onto a single connection. Depending on the user and/or the kind of data transmission and processing to and on the server, it might be reasonable for the proxy to prioritize one stream over another or to limit the incoming data depending on system- or connection load. Hence, the employment of *Stream Prioritization* and *Flow Control* is found to be reasonable for the $\mathcal{P} - \mathcal{S}$ link.

4.3 CLASSIFICATION AND REALIZATION OF HTTP/2 ADVANCEMENTS

The advancements of HTTP/2 that have been found advantageous share certain properties. These commonalities facilitate a classification, which allows a common implementation approach.

4.3.1 Link-type independent improvements

Header compression can be considered a class of its own, as both partial links are completely independent of each other, as the proxy serves as an endpoint for both link types. This involves for the header data to be completely unpacked before further treatment.

As header compression is fully dependent on the used web server for the connection between client and proxy no additional steps have to be undertaken in order to support this feature on this link type.

On the proxy side, where it acts as a client to the server, the header data is potentially amended, repacked and sent to the server. The response from the server then again gets processed on the proxy, while the headers are again extracted, repacked and sent to the client.

4.3.2 Cross propagation/runtime dependent features

The formula introduced in 4.2.2 applies in theory and can be used based on existing statistical data in the future, without realistic usage data, however, assumptions about their runtimes have to made based on their workload.

Unlike Header Compression, Server Push and Stream Reset both depend on the expected runtime of a method and have to take propagation over link borders into account.

The methods, which are expected to be *Long-Running*, are basically those with non-deterministic runtime behavior. This embraces methods with the following characteristics:

- Make use of the Dynamic Hash Table (**DHT**)
- Return an arbitrary large amount of data sets (**LA**)
- Require further processing on the crowdsourcing server (**CS**)

Due to the fact that neither for peers organized in a dynamic hash table, nor for the crowdsourcing server guarantees regarding runtime can be granted, it has to be assumed that methods that make use of outbound network connections are potentially long-running. This also applies to methods that make use of the database and may return an arbitrary large amount of data sets.

This is expected to apply on the managements methods of SANE in table 4.1, regarding only the link between Client and Proxy ($\mathcal{C} - \mathcal{P}$).

Method	Qualifier
findClosestInsanes	DHT
findData	DHT
getCampaigns	LA
getDHTNeighbours	DHT
getMySane	DHT
getMySubmissions	LA
getSane	DHT
storeData	DHT

Table 4.1: Potentially long-running management methods

Above that, this is expected to apply on the already existing crowdsourcing methods of MapBiquitous in table 4.2, spanning the whole virtual $\mathcal{C} - \mathcal{P} - \mathcal{S}$ link, what entails also having to take the according propagation *down the line* into consideration:

Method	Qualifier
correctWLANFingerprintingPosition	CS
createGSMFingerprint	CS
createWLANFingerprint	CS
getMyGSMFingerprintingSubmissions	LA
getMyWLANFingerprintingPositionCorrectionSubmissions	LA
getMyWLANFingerprintingSubmissions	LA
getWFSDataFromBS	CS
getWLANFingerprintingPositionFromBS	CS

Table 4.2: Potentially long-running crowdsourcing methods

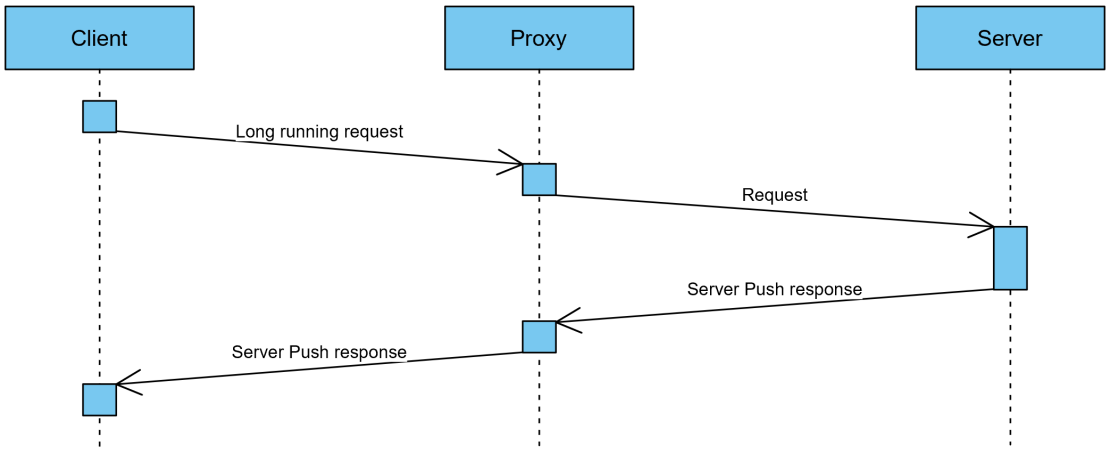

Figure 4.6: Server Push sequence diagram (Straightforward configuration)

Stream Reset

In case of *Stream Reset* this even accounts for any direction of data flow: If an operation on the crowdsourcing-server gets interrupted due to an error, the stream on the link between proxy and crowdsourcing server gets reset, which may in turn reset the connection to the client. The other way around is also possible: If a stream to the proxy is reset by the client, for instance in the case that the outcome of that particular operation became obsolete, also the stream to the proxy and thereby the stream to the crowdsourcing server may be reset in order to terminate the ongoing operation. In addition, also the proxy may interrupt an operation due to an error and in consequence reset both the stream to the client and to the crowdsourcing server, if already invoked.

Server Push

In case of *Server Push* in a Straightforward Proxy configuration, if a request that has been forwarded to the crowdsourcing-server terminates, the result has to be pushed to the proxy, which in turn has to trigger a subsequent push to the client, as can be seen in 4.6, which represents the optimal case of all participants being able of using HTTP/2.

In the case that one of the partial links only operates in fallback mode, meaning under the use of HTTP 1.1 as Upgrade- or Downgrade proxy, the particular connection has to be kept open instead to ensure a proper propagation, as can be seen in figure 4.7 and figure 4.8 respectively.

4.3.3 $\mathcal{C} - \mathcal{P}$ link features for load control

As stated in the previous section, the only reasonable HTTP/2 improvement applicable for controlling the load is stream or connection based *Flow Control* when the affected receiver experiences a high system or network load. Then, it can issue an update of its receive window (rcwd) via *WINDOW_UPDATE* frame to notify the sender, either client or proxy, of how much data he may send.

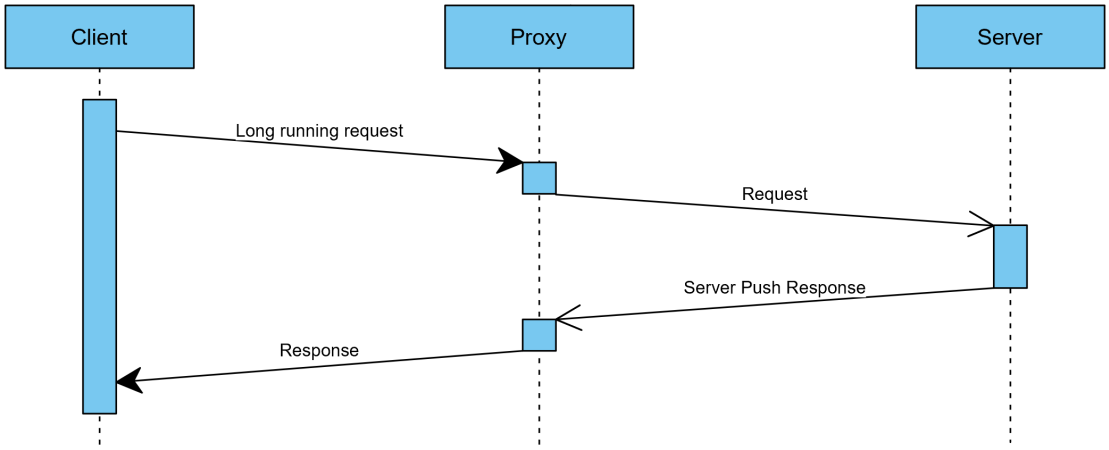

Figure 4.7: Server Push sequence diagram (Upgrade configuration)

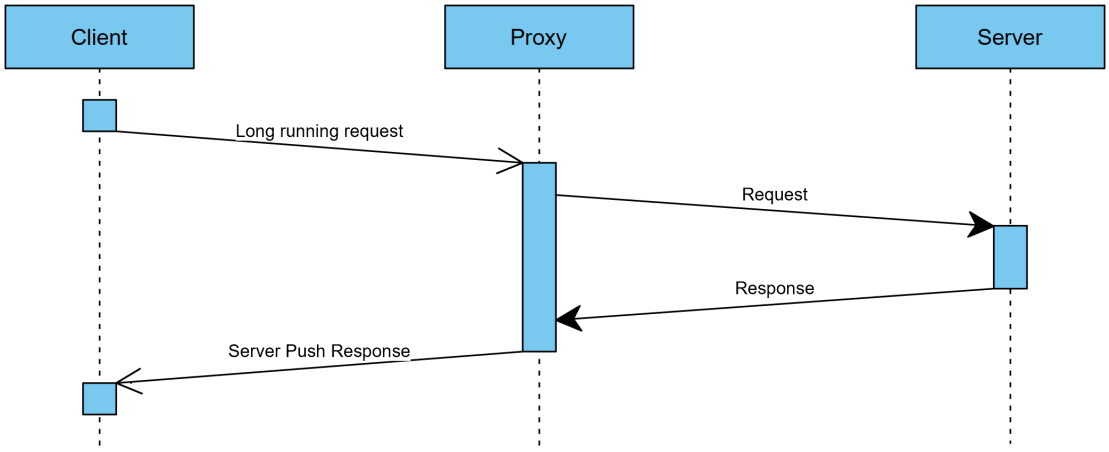

Figure 4.8: Server Push sequence diagram (Downgrade configuration)

4.3.4 $\mathcal{P} - \mathcal{S}$ link features for load control

Those features, that only concern the link between the SANE proxy and the crowdsourcing server, as only they use multiple streams within one connection, are basically targeting the ability of one respective side of the link to take control over the load it has to deal with.

As stated above, *Flow Control* enables the receiving side to control the amount of data its counterpart is allowed to send regarding one particular stream conforming to one particular operation by settings its current receive window (rcwd) via *WINDOW_UPDATE* frame. This can, similar to Flow Control regarding the $\mathcal{C} - \mathcal{P}$ link, be used in cases of high system or network load to signal the crowdsourcing server to limit the amount of data to send.

Stream Prioritization, in contrast, enables the Proxy to assign a priority to a stream and thereby to one particular operation, that is to be executed on the crowdsourcing server. The server can then use this information to prioritize the processing of individual streams by allocating system and bandwidth resources accordingly.

This concludes into the idea that every operation the proxy invokes on destination crowdsourcing servers has to have a priority assigned that puts it into relation of other ongoing operations, either to prefer the processing of one over another on the proxy or on the destination server. It should be noted that a triggered Server Push inherits the weight assigned to its parent stream and gets prioritized correspondingly.

For the proxy to be capable of defining these priorities, there has to exist some sort of basis of decision-making, which operation to prefer.

Priority determination

For being able to assign a weight to streams, corresponding to particular operations, the following deterministic approach is proposed as an example. This component, however, needs to be exchangeable to easily replace it with a component employing a more specifically designed algorithm if needed.

The following characteristics are taken into account:

- The crowdsourcing derivate the operation is part of, e.g. MapBiquitous
- The operation of the crowdsourcing derivate itself
- The user requesting this particular operation

This accommodates the fact that one user, one particular operation of a crowdsourcing derivate or the crowdsourcing derivate itself and its outcome respectively may be considered more important than others. Above that, it may be reasonable to also take the target crowdsourcing server itself into account in future implementations.

According to the HTTP/2 standard [BPT15], a stream's priority has to be specified on the interval $[1, 256]$. As the weight is proportional, as explained in 2.4.2, this interval does not necessarily have to be used to full capacity.

Taking the possibility of ultimately having 4 different characteristics that affect the resulting stream weight, the 255 possible values are distributed equally among them, leaving every characteristic with 63 *points* to assign. For every crowdsourcing derivate (including the SANE itself) (w_{cs}), every method (w_{op}) and every user (w_{user}) a value on the interval $[0, 63]$ has to be

assigned. If no weight is defined for a characteristic, the *maximum allowed value*$W_{max} - 1$ is used instead.

The resulting stream weight of an operation is now calculated as follows:

$$w_{opTotal} = w_{user} + w_{op} + w_{cs}$$

For the case that every characteristic is rated with 0, which would lead to the resulting value being 0, the stream weight is set to the lowest value allowed of 1.

For the sake of simplicity, it is also applied for Flow Control, but in contrast to the Stream Priority, where the particular weights are constructed into a prioritization tree, with flow control they form a total order. This means, for instance, an operation with the weight of 254 gets always preferred over an operation with a weight of *only* 253.

Weights put to use

Having an exact weight between 1 and 255 for every operation that is to be executed on the server, it can now be put to use for *Stream Priority* and *Flow Control*. The former is the case with every forwarded call to the crowdsourcing server, where the stream, which is used to execute the according operation, gets assigned this very priority. The crowdsourcing server is then able to use this priority information to allocate resources accordingly to ensure that the pending requests get processed as intended.

Flow Control, however, is only put to use in case of a system at load limit. When this is detected by the SANE proxy, it issues *WINDOW_UPDATE* frames to the crowdsourcing server corresponding to the operation with the **lowest** score until the load has normalized to allow for a regular operation. If this is again the case, the proxy tries to reverse this reduction by sending *WINDOW_UPDATE* frames again in the reverse order - thus starting with the operation with the highest priority whose stream's receive window has been reduced in the first place.

4.4 EVALUATION CRITERIA

In order to be in a position to give an objective evaluation of the implemented improvements on the SANE, certain criteria have to be defined upon which the evaluation is based on. Again, it is reasonable to distinguish between those that assess HTTP/2's general performance and the performance of the individual improvements.

4.4.1 General criteria

Latency

Improving packet latency has been defined an explicit goal of HTTP/2's advancements, why it serves perfectly as a metric, where it reflects the effectivity of all advancements like its binary nature, header compression and optionally Stream Prioritization, Flow Control and the Server Push feature, if used.

Processing speed

The processing speed or the time a defined operation needs to complete since reception, in comparison to using the same operation with HTTP1.1, indicates protocol effectiveness. As HTTP/2 is binary and therefore does not have to be parsed, operations are expected to finish earlier.

Resource usage

In addition to a faster processing speed, it is expected to consume less resources. These resources can furthermore be differentiated in CPU time, memory usage and bytes transferred.

4.4.2 Advancement specific criteria

In addition to the metrics above that are used to rate the overall performance, the following metrics will be used in order to assess particular advancements.

Header compression

Following Pu in his thesis [Pu16], the *header compression ratio* can be used to rate the effectiveness of HPACK for SANE.

Stream Reset

Issuing a Stream Reset can save resources by canceling an ongoing operation it would otherwise have wasted. The effectiveness of canceling an ongoing operation can hence be expressed by saved CPU time and bandwidth.

Above that, also the latency of an RST_STREAM frame, which equals to the time passing between the reception of this kind of packet until the closing of a stream comes into effect, should be taken into account. Closely related is the *overhead*, thus the amount of data being sent *pointlessly* during the just mentioned period between the intent to close a stream and the closing coming to effect.

Server Push

Using Server Push for SANE effectively avoids having to maintain open connections/streams for the entire runtime of an operation. Instead the server opens a new one on completion or on ulterior change of data, implementing the Publish-Subscribe pattern. For this reason, the total duration of connections or streams in an open state for executing an operation is an objective metric for Server Push.

Another way of measuring the effectivity of Server Push are the amount of octets sent in order to maintain the open stream compared to the additional amount of octets necessary to induce a subsequent push from the server.

Stream Prioritization

Measuring the effectivity of Stream Prioritization can be conducted by putting the time to complete a prioritized operation into relation with the time the same operation would take if it was invoked with base- or another priority.

Flow Control

Flow Control, however, can not be evaluated in reasonable manner in the environment at hand, as all the traffic on SANE is generated locally and does therefore not reflect a comparable usage pattern. Above that, its employment only makes sense where bandwidth or buffers for incoming transfers are limited, thus at the upper load boundary of the receiving instance.

4.5 CONCLUSION AND FURTHER PROCEEDING

By switching the connection type from HTTP 1.1 to HTTP/2, the payload is no longer embedded in a text stream but strictly separated into binary encoded protocol and payload. By the use of binary framing, the advantages it brings along come for *free*, meaning no further implementation work has to be done in order to solve the Head-of-Line Blocking problem of HTTP 1.1 and its inefficient and ambiguous nature. The other improvements of HTTP/2 over HTTP1.1 have been valuated as follows.

HPACK header compression

As header compression is realized by the used web server, it only has to be implemented where the SANE proxy acts as a client on the $\mathcal{P} - \mathcal{S}$ partial link. For any possible case this improvement is expected to be advantageous, as it reduces the size of the data to transmit, especially with subsequent requests, although it may increase CPU load minimally.

Stream Reset

This advancement of HTTP/2 has been found advantageous for certain operations that are expected to exceed a certain runtime. If all involved peers support HTTP/2, the Steam Reset has to be propagated across *link borders* accordingly, which has to be done on application level.

Server Push

In order to support informing the client of the termination of long-running methods or state changes and updates of underlying data, it has been decided to implement Server Push both on the link client-proxy ($\mathcal{C} - \mathcal{P}$) and proxy-server ($\mathcal{P} - \mathcal{S}$), also in order to allow informing the client on server-side updates while maintaining anonymity. Similar to Stream Reset, if all involved peers support HTTP/2, a Server Push from the server to the proxy has to be propagated *down the line*, from the proxy to the client, accordingly.

Flow Control

As the application of Flow Control is only reasonable if there is more than one stream within one connection, this improvement of HTTP/2 is only recommended to be implemented on the link between SANE proxy and crowdsourcing server ($\mathcal{P} - \mathcal{S}$). Here it makes use of a weighting system, where it serves as a way to prioritize the sending of responses from the server to the client according to the prementioned score of a stream, which represents one certain operation. This new feature can here be considered as a way of load controlling on the proxy.

Stream Prioritization

Similar to Flow Control, also Stream Prioritization is only reasonable if there is potentially more than one stream within a connection. For this reason, the use of this improvement is only plausible on the $\mathcal{P} - \mathcal{S}$ link, where it also employs the weighting system to instruct the server which streams and therefore operations to favor over others.

5 PROOF OF CONCEPT

“A proof is a proof. What kind of a proof? It’s a proof. A proof is a proof. And when you have a good proof, it’s because it’s proven.

Jean Chrétien”

5.1 FEASIBILITY

This section's purpose is to assess whether the implementations of the improvements expected to have a positive impact on SANE from 4.2 are feasible or not. A constraint for any of the following features is, naturally, that HTTP/2 is used for the partial link in question.

5.1.1 Client-Proxy ($\mathcal{C} - \mathcal{P}$) link

For this partial link, the feasibility of the improvements depend entirely on the web server's support for HTTP/2. As the Apache httpd with mod_http2, used for the test system (cf. B), fully supports HTTP/2, all improvements found advantageous are also found to be feasible, thus the following advancements are concerned:

- Header Compression
- Stream Reset
- Server Push

Flow Control however can currently not be implemented for this partial link, as it is not possible to actively set the receive window size using PHP. The only way is setting it statically during the web server's setup procedure, leaving no way of using it in a dynamic manner yet.

5.1.2 Proxy-Server ($\mathcal{P} - \mathcal{S}$) link

To implement reasonable HTTP/2 features, the SANE proxy has to behave as a client supporting HTTP/2. As it is neither possible nor reasonable to implement the protocol given the limited time available oneself, a program library, in which this functionality is contained, shall be employed. For this reason, available HTTP/2 libraries to use with PHP, were taken into account. There exist a few of those libraries for HTTP1.1, but only two with support for HTTP/2: Libcurl and guzzle, of which the latter also makes use of libcurl to deal with HTTP/2, providing an easier to use object oriented interface. Although it might be easier to use a fully-fledged HTTP client like guzzle, for the sake of not introducing additional dependencies as it has already been used by Pu in [Pu16], the use of libcurl as a *pure* HTTP2 handler is preferred.

Which HTTP/2 features are supported on the $\mathcal{P} - \mathcal{S}$ link depends mostly on the feature set supported by libcurl and its existing bindings to PHP. Those bindings determine which of the features implemented by libcurl are actually usable. The server part again merely runs on a web server for which is assumed that it fully supports HTTP/2, just as the web server used for this thesis. In the following will be examined which of the HTTP/2 advancements hence are supported by the given runtime environment.

Header Compression

As [Pu16] has found in his thesis, Header Compression is fully supported by libcurl for the version used. Above that, in his thesis Pu also provided the implementation of header compression for SANE. For this reason the following will focus on the advancements not yet implemented.

Stream Reset

According to [Stea] libcurl "will attempt to re-use existing HTTP/2 connections and just add a new stream over that when doing subsequent parallel requests", if configured appropriately. Specifically that is, if CURLMOPT_PIPELINING is set to the constant CURLMOPT_MULTIPLEX and every additional stream hooks into the curl multi interface. If then one of the single streams, represented by a single curl handle, is closed a *RST_STREAM* frame should be sent. Hence, Stream Reset as a feature is found to be feasible with the software and its particular version at hand.

Server Push

Due to the fact that PHP 5.6, which the SANE was targeted to run on, is lacking the necessary bindings for libcurl, it is not possible to employ the Server Push feature for SANE on the $\mathcal{P} - \mathcal{S}$ partial link under usage of this particular version.

For the proxy to be able to react on pushes from the server, the essential precondition naturally is that a crowdsourcing server exists, that has this feature implemented. Furthermore, it has to be evaluated whether or not the SANE proxy is able to run on the brand new PHP version 7.1, as there are no bindings to make use of the necessary callback handler with PHP 5.6. According to the release notes of the new PHP versions 7.0 [PHP15] and 7.1 [PHP16a], they underwent massive changes that can make it behave quite differently than PHP 5.6. Even though most changes concern the object oriented part, also basic syntax and data types just as high level functions for JSON processing are affected, just to name a few. As there are also no up-to-date unit tests available for regression testing it can not be guaranteed that the SANE proxy is operating as intended with PHP 7.1. Nevertheless, a proof of concept omitting an implementation of Server Push for the partial link $\mathcal{P} - \mathcal{S}$ for SANE is conducted either ways to demonstrate the principle operational capability.

Flow Control

Flow Control, however, is currently neither supported by PHP 7.1 nor by the underlying libcurl library. For this reason, the implementation of flow control, also for this partial link, is omitted.

Stream Prioritization

Although libcurl itself supports setting the stream priority on single handles, representing a stream within a connection, it is currently not supported by PHP in its most recent version according to [PHP16b] at this juncture. For this reason, unfortunately also the use of Stream Prioritization has to be omitted.

5.2 FUNCTIONAL DEMONSTRATION

This section's purpose is to prove the operational capability of the HTTP/2 improvements that were found to be advantageous, feasible and not already functional, either implicitly by design or by previous works. Above that the following demonstrations should give an understanding of the underlying principles and dynamics.

All tests were performed using the test setup described in B. During the tests, client- and server-side caching was completely switched off. The proof itself is given using a recording of the HTTP/2 traffic using the *Wireshark* packet analyzer. In order to enable Wireshark to decrypt the TLS traffic on the $\mathcal{C} - \mathcal{P}$ link, the environment variable *SSLKEYLOGFILE* was set which Chromium uses to log TLS session keys. Wireshark in turn uses the keys from this file to decrypt the TLS encrypted traffic.

In order to analyze the traffic on the $\mathcal{P} - \mathcal{S}$ link, the Apache web server was configured to use HTTP/2 over TCP (h2c) on the default port 80, instead of HTTP/2 over TLS (h2). This was necessary given the inability of libcurl to export Pre-Master-Keys as the webbrowsers Chromium and Firefox are able to, as described above. As long as not explicitly stated otherwise, PHP 5.6 is used for the following functional demonstrations, as the SANE has been developed for this version.

5.2.1 $\mathcal{C} - \mathcal{P}$ Stream Reset

To prove the functionality of stream resetting, the following test was conducted. The purpose of this test is, on the one hand, to ensure that closing or canceling a data stream results in closing a single stream within a connection, not closing the entire connection itself and, on the other hand, that indeed a *RST_STREAM* frame is used. The communication was recorded via Wireshark and can be found under C in a shortened form, in which *SETTINGS*, *WINDOW_UPDATE* and recurring *DATA* frames were, just as unnecessary details of frames, omitted.

The PHP script (*dl-closetest.php*) sends the necessary headers to initiate a download, as can be seen from 5.1, and puts out a lorem ipsum example text, until it is canceled. The entire source code can be found in appendix A, specifically under *$\mathcal{C} - \mathcal{P}$ Stream Reset test source code*.

```
1 ignore_user_abort(true);
2 header("Content-Description: File Transfer");
3 header("Content-Type: text/plain");
4 header("Content-Disposition: attachment; filename=lorem.txt");
5 header("Expires: 0");
6 header("Cache-Control: no-cache");
7 header('Transfer-Encoding: chunked');
```

Listing 5.1: HTTP header to initiate download of *endless* lorem ipsum file (*dl-closetest.php*)

The test protocol can be found under *$\mathcal{C} - \mathcal{P}$ Stream Reset test* in appendix C. Due to its length irrelevant frames and belonging irrelevant content have been removed. It begins with a *HEADERS* frame being sent from the client to server, requesting *closetest.php*, as can be seen in the *:path* entry on line 12. The following frame 40, beginning in line 14, is the web server's response to that request, as line 19 indicates with having 443 as source port, what conforms to the web server's source port for TLS encrypted data. The new stream (stream id=5) then gets opened by the client via frame 45, requesting *dl-closetest.php* as can be seen in in the *:path* header entry on line 37. The download itself is then initiated on this very stream in frame 46, as the lines 47 and 54 state, where the defined headers from 5.1 reappear in the *HEADERS* frame (from line 49 to 53) just before the first content is attached in a *DATA* frame, still belonging to the same Ethernet frame.

After about 2.3 seconds and several *DATA* frames later the download got canceled manually. The cancellation results in a *RST_STREAM* frame being sent, as can be seen from 5.2, effectively closing the stream with id 5 within the connection. ■

```
Frame 165: 128 bytes on wire (1024 bits), 128 bytes captured (1024 bits
    ) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
    _00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 50618 (50618), Dst Port: 443
    (443), Seq: 1010, Ack: 271413, Len: 42
Secure Sockets Layer
HyperText Transfer Protocol 2
    Stream: RST_STREAM, Stream ID: 5, Length 4
        Length: 4
        Type: RST_STREAM (3)
        Flags: 0x00
            0000 0000 = Unused: 0x00
        0... .... .... .... .... .... .... .... = Reserved: 0x00000000
        .000 0000 0000 0000 0000 0000 0000 0101 = Stream Identifier: 5
        Error: CANCEL (8)
```

Listing 5.2: Wireshark capture of *RST_STREAM* frame signaling a cancellation of the download

This demonstration thus shows that resetting a stream already works without additional implementation if the SANE runs on a HTTP/2 enabled web server. Furthermore the behavior of the SANE, in case of a Stream Reset, is identical to its momentary behavior in case of an unexpected close of the entire connection. Nevertheless, it should be considered to react accordingly to the abortion of an operation in order to ensure data consistency. This topic will be again be picked up and treated more thoroughly in 5.4.1.

5.2.2 $\mathcal{C} - \mathcal{P}$ Server Push

The following tests were conducted to evaluate the operational capability of the method to be used for the implementation of Server Push for SANE on the link between client and proxy.

```
header("Connection: Keep-Alive");
header("Expires: 0");
header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
header("Link: <https://localhost/poc/filestream.php>; rel=preload; as=
    document", false);
```

Listing 5.3: Server Push inducing headers

GET Request

In order to induce the pushing of data to the client from the web server running PHP, a *link header* with a reference to the source of the data to push (*filestream.php*) has to be sent to the client, what is done by the part of the *pushtest.php* script that can be seen in listing 5.3. The source code of the referenced script *filestream.php* is attached in *$\mathcal{C} - \mathcal{P}$ Server Push test source code* in the appendix A. For easier trace- and readability, Apache's DEFLATE module (mod_deflate) has been switched off, what effectively deactivates the entire server-side content encoding.

When accessed directly via Chromium web browser, what effectively conforms to a GET request, the script referenced in the *link header* pushes a *lorem ipsum* text of 296 bytes and

adds, also for readability, a number in brackets before and a newline character (\n) after, resulting in 300 byte payload in total to push.

The Wireshark protocol attached in $\mathcal{C} - \mathcal{P}$ *Server Push test* in appendix C first shows the request itself, after successfully establishing communication and setting connection parameters, in Frame 37. This request is responded to in Frame 39 with a *PUSH_PROMISE* frame (lines 13 and 21), which defines stream id 2 (line 22) to use for pushing the data. This stream is then actually put to use to push *HEADERS* and *DATA* frames to the client, as line 40 shows for the *HEADERS* frame and line 42 shows for the following *DATA* frame. The length of the *DATA* frame (also line 42) equals exactly the length of the used *lorem ipsum* snippet.■

POST Request

However, if the script in listing 5.3 is invoked via POST request, no *PUSH_PROMISE* frame is sent to the client, hence no server push is induced in that case, even though the HTTP/2 specification does neither explicitly prohibit a *Push after POST* nor does it even mention this case.

According to the HTTP/2 specification it is just "(...) not possible to push a response to a request that includes a request body" [BPT15], which was not the case in the test scenario, as can be seen from the protocol also attached to $\mathcal{C} - \mathcal{P}$ *Server Push test* in appendix C.

Line 1 shows the beginning of the client's request in frame 23 solely containing the request's *HEADERS* frame, while the next frame 25 (line 13) already shows the server's answer in form of a regular response on the same stream id 1 (lines 21 and 24). Hence, no *DATA* frame, where the request body would reside, has been sent between those, which would disallow the pushing of the requested content according to the standard cited above. The only difference of this request to the one above is it being a POST request (line 10) instead of a GET-, what obviously prevents a *PUSH_PROMISE* frame being sent to induce the Server Push. ■

The test of Server Push on the $\mathcal{P} - \mathcal{S}$ connection in 5.2.4, employing libcurl, also using a POST request instead of a GET as the only difference, arrives at the exactly same result.

To the date the SANE completely relies on POST requests, both for the incoming communication on the $\mathcal{C} - \mathcal{P}$ and the outgoing communication on the $\mathcal{P} - \mathcal{S}$ link. However, there is - at least at this juncture - no solution for implementing Server Push for SANE using POST requests. As HTTP/2's GET request and its associated payload reside inside a *HEADERS* frame, it can be considered justifiable from a security related point of view, to use GET- instead of POST requests as long as a TLS encrypted HTTP/2 connection is employed.

5.2.3 $\mathcal{P} - \mathcal{S}$ Stream Reset

In order to evaluate if the Stream Reset advancement of HTTP/2 is usable for resetting streams inside a single connection under usage of libcurl on the $\mathcal{P} - \mathcal{S}$ link, the following test was implemented. For PHP to being able to use HTTP/2 in the role of a client, as previously stated, libcurl is employed.

The pivotal point of this test is the script *ps-closetest-timedabort.php*, which can among the other all the scripts in this proof be found under $\mathcal{P} - \mathcal{S}$ *Stream Reset test source code* in appendix A. This script uses a curl multi handle, adds two single handles of which the first accesses *dl-closetest2.php*, a script quite similar to *dl-closetest.php*, which is also used to generate an endless stream of *lorem ipsum* snippets in 5.2.1. The second one accesses *idlestream.php*, which, as the name suggests, does not much but returning a number in

brackets that increments with every loop execution at the same frequency only for ensuring that the multi handle is kept open when the first single handle is removed. After 3 seconds of execution, the first stream is closed by removing the single- from the multi handle by using *curl_multi_remove_handle(resource $mh, resource $ch)*. According to the PHP manual "Removing the ch handle while being used, will effectively halt the transfer in progress involving that handle" (sic). Then, after an additional 2 seconds since execution start, the entire multi handle processing is stopped.

To reset a single stream, as described in 2.4.2 a *RST_STREAM* frame has to be sent by either client or server on the stream in question. To evaluate if such a frame is being sent, again Wireshark is put to use to provide an in-depth view of the network traffic to verify a *RST_STREAM* frame is actually being issued.

After requesting *ps-closetest-timedabort.php* via web browser, the upper Wireshark recording in $\mathcal{P}-\mathcal{S}$ *Stream Reset test* in appendix C shows two HTTP1.1 requests being sent machine-local via IPv4 in frames 33 and 38 (lines 13 and 21) to the web server on port 80, requesting *dl-closetest2.php* and *idlestream.php* on two separate ports (51726 and 51728). The use of IPv4 signals that it is actually curl doing the request as the browser uses IPv6. Due to the fact that each request uses their own port this indicates that for each request an own connection is being used. This sustains even after the connection, or better connections, being upgraded in frame 40 and 41 (lines 31 and 37). The web server's response to the request to *ps-closetest-timedabort.php* starts in Ethernet frame 52, containing, alongside the *HEADERS* frame and a first *DATA* frame, also two *SETTINGS* frames and a *WINDOW_UPDATE* frame. The destination port number in line 50 tells that this Ethernet frame belongs to the request as stated. The entire connection is finally closed via *GOAWAY* frame 69, starting on line 96, while the other connection is closed, also under usage of a *GOAWAY* frame, in line 107. However, no *RST_STREAM* frame is sent. As two separate connections containing only one stream each are used, there is no point in sending an *RST_STREAM* frame at all, as the entire connection has to be closed anyway.

When the exactly same test is repeated under usage of PHP 7.1, depicted by the lower protocol in $\mathcal{P}-\mathcal{S}$ *Stream Reset test* in appendix C it paints a completely different picture: The first request is, as with PHP 5.6, also sent as a HTTP1.1 request, as can be seen in frame 32 (line 13), and upgraded to HTTP/2 in frame 34 (line 21). The second request, however, is sent using HTTP/2 inside the *HEADERS* frame in Ethernet frame 39 (line 31), naturally using the just upgraded connection as the only one instead of two as with PHP 5.6. Consequently, when the download is canceled by removing the single handle from the multi handle, only this particular stream is reset by a *RST_STREAM* frame being sent in frame 52 (line 61), affecting stream 1, which is automatically assigned for upgraded connections, as line 68 states. The second stream is closed implicitly as the now idle connection is closed via *GOAWAY* frame 58 beginning in line 83, referencing explicitly the contained stream 3 in line 91.

From these results it follows that it is not viable to use Stream Resetting with PHP 5.6 as this version is not yet enabled to use libcurl's stream multiplexing. This finding weighs even heavier as this also entails that the synergistic effects of combining multiple requests onto one connection would have to be omitted as long as the SANE is not verified to to operate on PHP 7.1.

5.2.4 $\mathcal{P}-\mathcal{S}$ Server Push

Due to the fact that it is not feasible to implement Server Push on the $\mathcal{P}-\mathcal{S}$ partial link with PHP 5.6 and it is currently not testable, let alone provable that the SANE is able to operate on PHP 7.1, it will not be implemented in the course of this thesis for the SANE. Instead, at least a proof of the principle concept shall be presented here.

The pivotal point of this test is the *ps-serverpushtest.php* script, which is available under $\mathcal{P}-\mathcal{S}$ *Server Push test source code* in the appendix A. For communication with the server (in the role of a client) it also makes use of libcurl. On invocation it creates a libcurl multi handle, registers a callback function for handling incoming push responses, then an easy handle is added, which calls the script *pushtest_insecure.php*, similar to the script already known from 5.2.2, with the only difference that it references a *http://* URL instead of *https://* in its *link header* in order to use an unencrypted connection, making it possible to record the traffic via Wireshark, as described previously.

The referenced script, in turn, sends a *link header* that references another script (*filestream.php*), which generates the actual data stream that is pushed to the libcurl instance in the context of *ps-serverpushtest.php*.

For the sake of confirmability this push response is finally printed out to the invoking client.

GET Request

The first protocol that can be found under $\mathcal{P}-\mathcal{S}$ *Server Push test* in the appendix C shows a Wireshark capture of the relevant communication including details, which the following proof relates to.

Line 1 and following shows the GET request issued by libcurl; due to the fact that unencrypted HTTP/2 is used in order for Wireshark to be capable of recording the otherwise TLS encrypted traffic and the only way of doing this is issuing an upgradable HTTP1.1 request, the initial request is HTTP1.1, using port 80. According to line 5, the communication happens entirely on the *localhost*, indicated by the IP address 127.0.0.1. The communication between web browser and the web server, in contrast, uses an IPv6 address and port 443.

After the upgrade to HTTP/2 (frame 33, line 14), an Ethernet frame containing a *PUSH_PROMISE* HTTP/2 frame is sent from the server to the libcurl client in frame 40 beginning with line 23 alongside a *HEADERS* frame and a *DATA* frame containing the regular server answer. Line 31 shows the *Promised-Stream-ID: 2* inside the *PUSH_PROMISE* frame, i.e. the stream id for the upcoming data transfer. Line 36 then shows the resource whose content is about to be pushed. The next Ethernet frame, number 42, comprises two HTTP/2 frames, again *HEADERS* and *DATA*, but this time on a differing Stream ID 2 (lines 47 and 49), which has been promised to be used for the push transmission as stated above. ■

POST Request

Is the script containing the libcurl invocation, however, invoked to do a POST instead of a GET request using the curl setting shown in listing 5.4, the result is, as already foreclosed in 5.2.2, that no *PUSH_PROMISE* frame is sent by the server and hence no Server Push is being induced.

```
1 curl_setopt($chRequest, CURLOPT_POST, 1);
```

Listing 5.4: Curl setting for POST instead of default GET

The second protocol of $\mathcal{P}-\mathcal{S}$ *Server Push test* in appendix C shows a Wireshark capture of the relevant communication including details, which the following proof relates to.

Line 1 shows that instead of the GET request a POST request is sent from the libcurl instance (that uses the IPv4 address and port 80 as source and target, as can be also seen in line 1).

The Ethernet frame number 38, beginning in line 20, the server's answer, now in HTTP/2 after the upgrade in frame 33, then shows no sign of a *PUSH_PROMISE* frame as an answer to the POST request, as was the case when a GET request (see above) was used for the otherwise identical request. ■

Due to the fact that the only difference between those two executions is the setting to do a POST request, instead of a GET request, this leaves no other conclusion as that it is at this juncture not possible to implement Server Push for POST requests. Certainly, it is highly likely that the workaround using GET instead of POST requests, that was used in the functional demonstration of Server Push on the $\mathcal{C}-\mathcal{P}$ partial link, can also be applied on the $\mathcal{P}-\mathcal{S}$ link, as soon at is is guaranteed that SANE is capable of running on PHP 7.1, which has the necessary libcurl bindings to make use of the callback function that is invoked when a push response arrives. The principle of using GET- instead of POST requests for the entire virtual $\mathcal{P}-\mathcal{S}$ is demonstrated in 5.2.6.

5.2.5 $\mathcal{C}-\mathcal{P}-\mathcal{S}$ Stream Reset

The previous demonstrations 5.2.1 and 5.2.3 prove that particular operations on the partial links $\mathcal{C}-\mathcal{P}$ and $\mathcal{P}-\mathcal{S}$ can indeed be canceled by an HTTP/2 *RST_STREAM* frame, though for the $\mathcal{P}-\mathcal{S}$ link this is only reasonable under employment of PHP 7.1. Otherwise, libcurl is not able to do connection multiplexing, meaning that instead of streams within a connection, an own connection is employed for every request and response, what partially foils the advantages of HTTP/2 and Stream Reset respectively.

The following test now combines the demonstrations mentioned above to reset an ongoing operation on the proxy and subsequently on the server from the client's perspective.

This demonstration is constructed as follows: The client issues a request to the proxy (specifically to *proxy-index.php* in the *cps-reset* sub directory) , which in turn issues a request to the server (specifically to *server-index.php*) under usage of libcurl. The invoked operations continue to run until the reception of a sign that the link to the respective counterpart is interrupted, ideally in form of a *RST_STREAM* frame, what shall be proven in this test. As previously described, to recognize the closing of a stream or a connection, each of the scripts has to check for this event actively using PHP's *connection_aborted()* function after data has been sent. If checked without sending data, PHP will not recognize a link being closed since. If the proxy now recognizes such a link being closed, it in turn closes the link to the server by removing the single handle from the multi handle, while keeping another stream open to avoid the entire connection being closed. The source code of the scripts in this test is attached under $\mathcal{C}-\mathcal{P}-\mathcal{S}$ *Stream Reset test source code* in appendix A.

The following test is again conducted using Wireshark to show that actually a *RST_STREAM* frame is sent from the client to the proxy as well as from the proxy to the server. The protocol can again also be found in the appendix C, in $\mathcal{C}-\mathcal{P}-\mathcal{S}$ *Stream Reset test* specifically, due to its length, although it contains details only in the required depth and has already been stripped of HTTP/2 frames not relevant to the proof like *Magic*, *SETTINGS*, *WINDOW_UPDATE* and redundant *DATA* frames.

The Wireshark protocol begins with the request from web browser to the proxy in line 1, followed by a HTTP 1.1 GET request, issued via libcurl by the proxy to the server instance in the

Ethernet frame 31, beginning in line 11. Alike the previous tests, libcurl uses IPv4, as can also be seen in line 11 by means of the IPv4 address or line 15. This distinguishes it from requests issued by the used Chromium web browser, which uses IPv6. The following frame 33 upgrades the connection from HTTP 1.1 to HTTP/2, just as the according message "HTTP/1.1 101 Switching Protocols" states. Ethernet frame 38, beginning in line 27, represents the libcurl client's request on the aforementioned second stream that is merely required only to have a second stream on the $\mathcal{P} - \mathcal{S}$ link to avoid that the entire connection is closed. Frame 40, beginning in line 36, is the first response from the server to the proxy's libcurl client, as can again be seen from the IPv4 addresses and ports used. It contains both a *HEADERS* and already a *DATA* frame. As the source IP address and port is on the left side and the destination IP address and port is located on the right side in the protocol, this frame travels from server (HTTP standard port 80) to the client (client port 58672). The used stream has id 1, which is assigned by default to the request leading to an upgrade from HTTP 1.1. The next recorded Ethernet frame 46 (line 57 and following), in contrast, only contains a single *DATA* frame from proxy to the web browser client, just as the now following Ethernet frame from server to the proxy also contains a *DATA* frame. This pattern repeats several times until the stream is reset by the client, triggering a *RST_STREAM* frame in Ethernet frame 74 (line 95 and following). The fact that it again uses an IPv6 address shows that this is indeed the *RST_FRAME* issued by the client. In Ethernet frame 80 (line 123 an following) then also the stream between proxy and server is reset, as the usage of IPv4 tells. A cancellation of the client's request thus leads to a *RST_STREAM* frame being sent from the client to the proxy what in turn triggers another *RST_STREAM* frame being sent on the link between proxy and server. ■

Interestingly do the first responses to the second request on stream id 3 not appear in the protocol until after the actual request, being subject to this test, is canceled via the *RST_STREAM* frame. This suggests that both requests belonging to the multi handle are executed on the same web server thread. Moreover is the second request hence not executed until the first has terminated, making multiple requests within a multi handle in fact being executed sequentially instead of parallelly as one would expect. This circumstance will be further examined in the following evaluation chapter. Above that it has to be evaluated if this behavior is exclusive to Apache httpd using an internal module for executing PHP scripts or if this also applies on scripts executed using FastCGI, as other web servers do.

5.2.6 $\mathcal{C} - \mathcal{P} - \mathcal{S}$ Server Push

The previous demonstrations 5.2.2 and 5.2.4 prove that it is feasible to implement Server Push under usage of the HTTP GET method with parameters held in the HTTP/2 header. For the $\mathcal{C} - \mathcal{P}$ link with PHP 5.6 and PHP 7.1, for the $\mathcal{P} - \mathcal{S}$ link only with PHP 7.1 due to limitations of the PHP bindings to libcurl.

The following test now combines these findings to demonstrate, that it is possible to introduce Server Push on the entire virtual $\mathcal{C} - \mathcal{P} - \mathcal{S}$ link under usage of PHP 7.1. It intentionally resembles the communication that takes place in case of a *Straightforward Configuration* of SANE as proposed in 4.3.2, depicted in the respective sequence diagram in figure 4.6.

To do so, the client in form of a web browser again transmits parameters inside the HTTP/2 header to the Proxy (*proxy-index.php*), which extracts these headers from the request and issues another requesting using libcurl to the server (*server-index.php*). The server in turn again extracts these HTTP/2 headers, passes them to a Memcached instance while using a SHA256 hash of the content as a key and induces a Server Push using the aforementioned link header to *server-pusher.php*, attaching this very key as a GET value. The *server-pusher.php* script fetches the content in Memcached by means of the passed hash key and returns it, now on a newly opened stream to the proxy. The proxy recognizes an incoming Server Push as the push callback is called, puts the response again into Memcached under usage of a SHA256 hash of the response as a key and induces a Server Push of *proxy-pusher.php* to the client, again attaching the SHA256 key to the URL as a GET parameter. The *proxy-pusher.php* script finally extracts the content by means of this key from Memcached and returns it, as a new stream to the client. To ensure that Proxy and Server put different values into Memcached, the returned content is altered by adding the script's name and a line break to the beginning of the incoming response, which also allows for tracing the call order in the response finally pushed to the client. This communication is depicted in figure 5.1. The source code of this test can again be found under *$\mathcal{C} - \mathcal{P} - \mathcal{S}$ Server Push test source code* in appendix A.

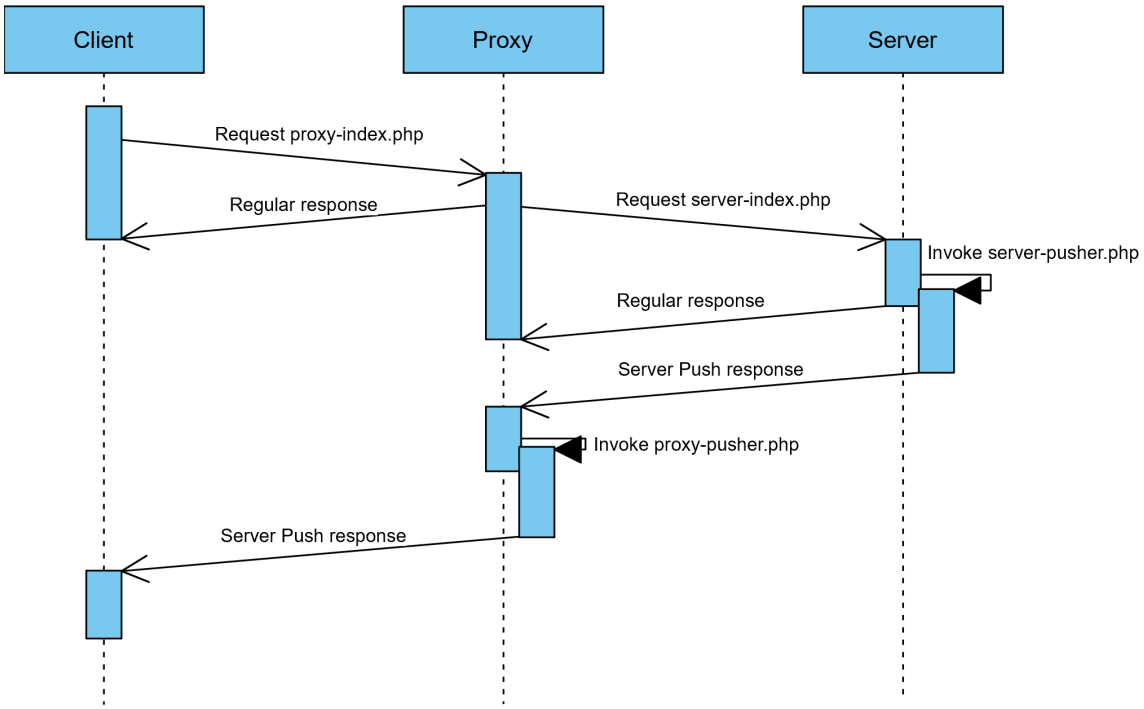

Figure 5.1: $\mathcal{C} - \mathcal{P} - \mathcal{S}$ Server Push demo sequence diagram

The following proof is again issued using Wireshark and the protocol can again be found under $\mathcal{C}-\mathcal{P}-\mathcal{S}$ *Server Push test* in the appendix C due to its length, although it contains details only in the required depth and has already been stripped of frames and belonging content not relevant to the proof like *Magic*, *SETTINGS*, *WINDOW_UPDATE* and redundant *DATA* frames. The request from the client to the proxy starts with frame 26 to *proxy-index.php*, as can be seen from line 11. The client's request again uses IPv6, as the lines 1 and 5 tell. This triggers another request, now from the libcurl client to the server using IPv4 and HTTP 1.1 in clear text, as can be seen on line 14 or the lines 26 and 27 respectively. The target of this request now is *server-index.php*, as also line 14 states. The following reply, starting with frame 33 in line 24 upgrades the connection to TLS secured HTTP/2, followed by an Ethernet frame number 51 (starting with line 32) containing an HTTP/2 *PUSH_PROMISE* frame, a *HEADERS* frame and a *DATA* frame. Line 34 again tells, using IPv4, that it indeed depicts the communication between the server and the libcurl client. According to line 38, the *PUSH_PROMISE* frame *promises* to use *Promised-Stream-ID: 2*. The target to push can be seen via *:path:* header entry on line 39, still inside the *PUSH_PROMISE* frame, disclosing the content of *server-pusher.php* to be pushed with an id attached. The now following Ethernet frame 63, starting with line 48, contains this content. The lines 53 and 55 state that this actually happens via Stream 2. Ethernet frame 79, starting with line 68, contains another *PUSH_PROMISE* frame, now however from the proxy to the client, as line 72 indicates as again IPv6 is used. Line 77 shows to use Stream ID 2 on the connection between client and proxy, while line 78 shows that *proxy-pusher.php* is the target of this Server Push. Ethernet frame 92, starting with line 86, then finally shows that the data indeed is pushed, which can be identified by the fact that stream id 2 is used for the *HEADERS* (line 94) and the *DATA* frame (line 96) itself. ■

5.3 $\mathcal{C} - \mathcal{P}$ SERVER PUSH IMPLEMENTATION

5.3.1 GET instead of POST

As described in 5.2.2, it is not feasible to implement Server Push employing HTTP POST for incoming requests, while it is under usage of the GET method. As the name suggests is the GET method usually used for retrieving data from the server. Modifying data, however, using the GET method is generally considered a bad practice, as GET methods are considered to be safe and idempotent, meaning they can be cached by intermediaries and arbitrary often resent without modifying underlying data. Above that, with GET parameters are embedded within the URL, what entails them being logged on the server side if an access log is used and several symbols are reserved and have to be escaped and *percent-encoded* respectively [BFM05]. Besides, even though there is no hard-coded limit for the URL length of GET requests, neither defined in the URL RFC cited before nor in in the HTTP/2 standard [BPT15], it is in fact either limited by the browser or the maximum length of the *:path* field of the HTTP/2 header, where the entire URL eventually resides.

5.3.2 Parameters in the header

For this reason, when using GET for SANE, the parameters are about to be put into the request header, what solves the problems stated above: As each individual value is put into an own header entry, its size limit concerns a single value, not all values combined; the possibly logged URL does not contain the parameters and the parameter values do not need to be escaped and percent-encoded. Above that it has to be ensured that the responses are not cached, the requirement to be safe naturally can not be fulfilled, as the GET method is here used to perform operations that were prior to this solution implemented by using HTTP POST. Their idempotency, however, is ensured on application level by the usage of signatures over the parameter content.

Nevertheless, using a HTTP/2 *HEADERS* frame to contain the parameters limits the maximum size of a parameter. According to [PR15], the length of a newly added header, thus the combined length of key and value prior to a possible Huffman encoding, may not exceed the current setting for the current maximum table size. If this happens anyway, the current table is "(...) emptied of all existing entries and results in an empty table.". The current maximum can, however, be extended by a dynamic table size update, which can occur multiple times "(...) between the transmission of two header blocks". The initial size is (for HPACK with HTTP/2) 4,096 octets or 4kB, what is hence also the limit for the first value to be added to the dynamic table, before it is getting resized. For all operations exceeding this limit, it has to be made sure to check the maximum table size before adding, otherwise the table will be emptied, as described before. In this case a value smaller than the current maximum table size is to be added. If only values exceeding the current maximum size are left to be added, they need to be gradually split into chunks smaller than the particular maximum table size, identifying the split parts using a serial number following a - as separator. This algorithm is depicted in Algorithm 1 (Recursive chunking algorithm).

On the recipient's side, the chunked headers simply have to be concatenated using the sequence number being attached to the header's key. Due to the simplicity, an explicit description is omitted.

Algorithm 1 Recursive chunking algorithm

```
Require: paramList: list of value lists sorted by value length
 1: function ADDCHUNKEDPARAM(paramList, chunkCount=0, headers=new List())
 2:     for i = 0 to paramList.size − 1 do
 3:         for j = 0 to paramList.get(i).size − 1 do
 4:             currentMts =GETMAXTABLESIZE
 5:             if paramList.get(i).get(j).value.size ≤ currentMts then
 6:                 headers.add(paramList.get(i).key+"−"+chunkCount, paramList.get(i).get(j).value)
 7:                 chunkCount + +
 8:             elseADDCHUNKEDPARAM( SPLITPARAM(paramList.get(i).key,paramList.get(i).get(j)],
    currentMts),chunkCount,headers)
 9:             end if
10:         end for
11:         chunkCount ← 0
12:     end for
        return headers
13: end function
14: function SPLITPARAM(key, value, maxTableSize)
15:     chunkedEntry=SUBSTRING(0, maxTableSize, value)
16:     rest=SUBSTRING(maxTableSize+1, value.length, value)
17:     chunkedParamList= new List()
18:     chunkedParamList.get(0).key = key
19:     chunkedParamList.get(0).value.add(chunkedEntry)
20:     chunkedParamList.get(0).value.add(rest)
        return chunkedParamList
21: end function
```

5.3.3 SANE control flow

In order to being able to comprehend the modifications conducted, it is necessary to understand the internal architecture and control flow during the processing of a SANE request. As the SANE uses procedural- instead of object oriented code, it employs *include*-commands to *invoke* operations dependent on certain preconditions, which effectively combines PHP scripts into one that is then being executed.

The pivotal point of all requests is the *index.php*, which first includes either the *method_includer.php* for SANE's own methods or the *cs_includer.php* for crowdsourcing-related methods, based on the passed parameter *method*. These includers in turn, include the php files containing the logic of the particular methods, which then do the actual processing and response generation. This control flow is depicted in figure 5.2.

5.3.4 Modifications for Server Push

In order to establish the Server Push functionality for the SANE while maintaining compatibility to the currently employed procedure under the use of HTTP POST without pushing functionality, a new component *h2push_includer* is introduced. This component is used by both of the above mentioned includers to separate the validation and pre-processing of the request parameters from the actual processing and response generation. Instead of directly including the according methods, *method_includer.php* and *cs_includer.php* generate the previously introduced *link header*, referencing the *h2push_includer.php*, which in turn includes and therefore invokes the actual SANE or crowdsourcing method, based on the passed parameters. The result is then pushed to the client. In case of not fulfilling one of the preconditions or an error, the pre-existing execution path returns the result directly, using the stream established during the client's request. Those preconditions are

- Method marked as *pushable*
- Request parameter *s-useserverpush* in HTTP/2 header set to *1* or *true*

Marked as *pushable* are those methods that have been found to be runtime dependent in 4.3.2. These criteria should also be applied on methods added in the future. To mark the methods accordingly, their respective method property information has to be amended by the monotonous attribute *pushable* as key and a Boolean as value. In addition to the criteria above, the configuration file *config.inc.php* contains a variable $_CONFIG['EnforceServerPush'], which works as an override for testing purposes. If set to *true* each and every method is tried to be pushed, irrespective of the method's and client's settings.

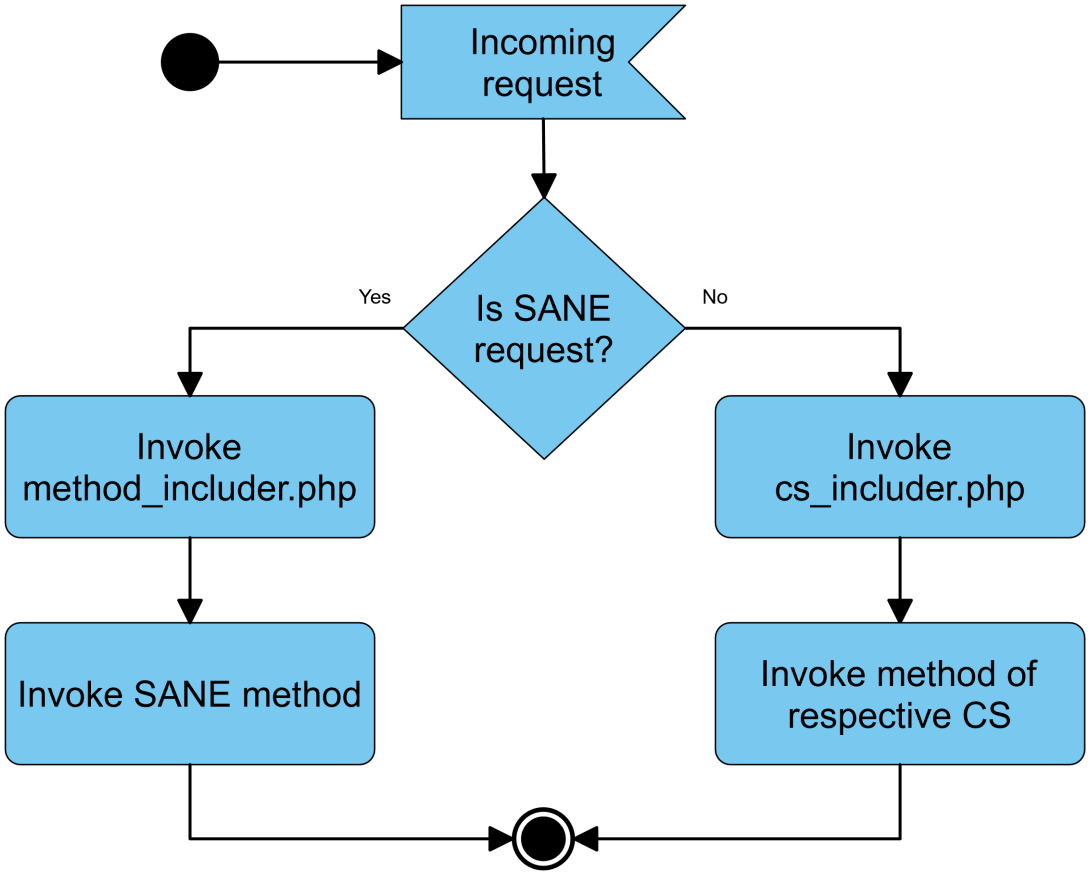

Figure 5.2: SANE regular request processing

Instead of having only one web server thread handling the entire processing, with Server Push enabled, two separate threads are used. This entails also having to pass the parameters from the first (*method_includer.php* and *cs_includer.php*) to the second process (*h2push_includer.php*), including the actual method to be executed. As for this implicit GET request no header can be added, meaning parameters can only be transmitted inside the URL, another way of transmitting them among the execution contexts had to be found.

The most obvious solution is to employ PHP shared memory to transfer values. As this is naturally only possible among threads, sharing a common address space and it can not be ensured that the SANE is employed in a multi-threading environment this solution is

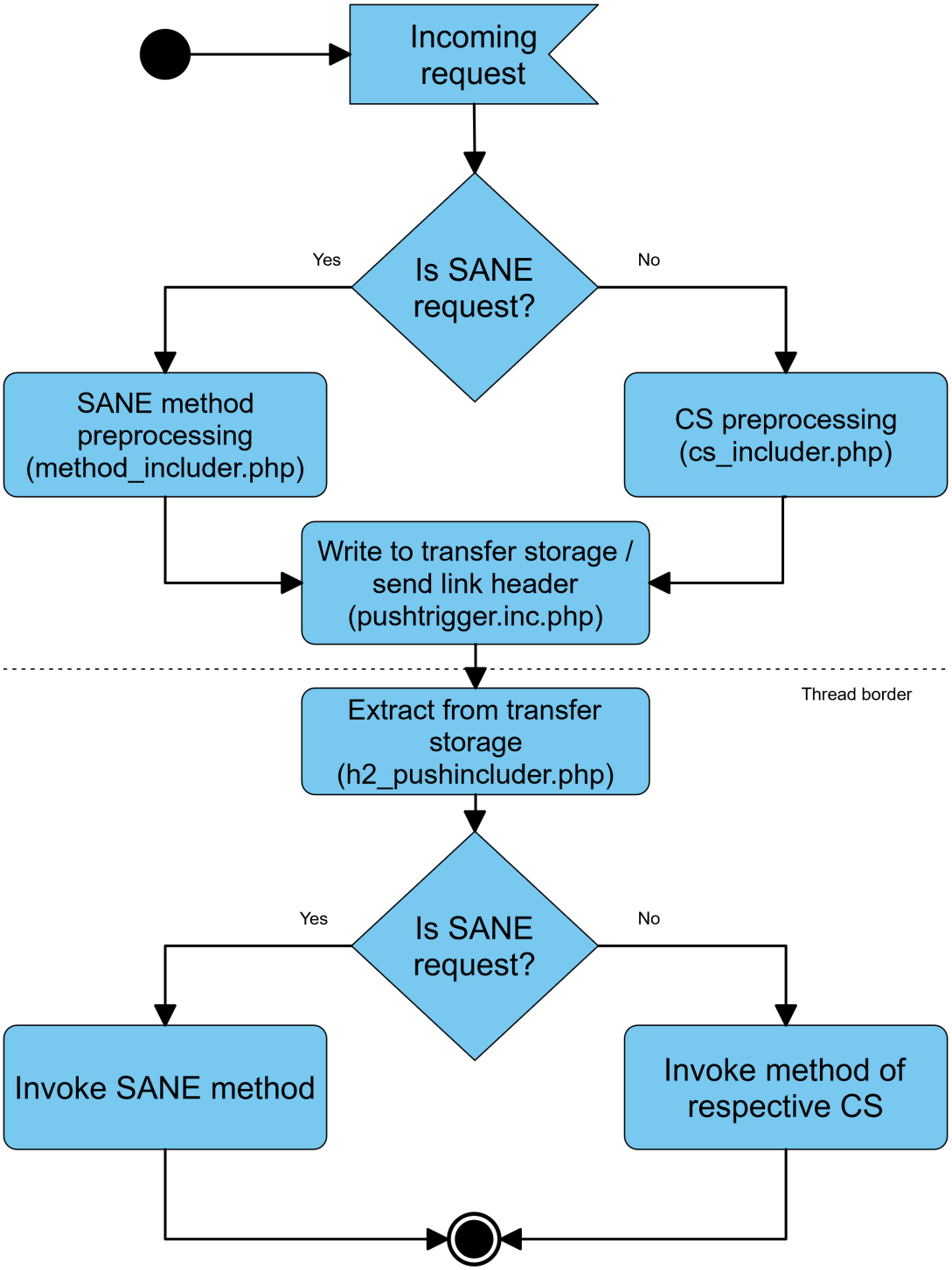

Figure 5.3: SANE server push request processing

depreciated. Above that, the shared memory function of PHP has no built-in memory management or locking functionality to avoid race conditions or memory corruption, what makes it prone to errors even if self-implemented. For these reasons it has been decided to put up with the introduction of an additional dependency in form of *Memcached* to use it is a *transfer storage*, which is predestined for storing and accessing temporary data with low latency. As a key/value store, Memcached can store any kind of data, which is identifiable by a unique key. The unique key is, in this case, generated by hashing the JSON encoded parameters that have to be transmitted from the process answering the request in the first place and the process whose response is then pushed to the client via SHA256. This very key is the only value transmitted via the *link header* as a GET parameter.

Alternatively, if the introduction of new dependencies shall be avoided at any costs, the database server can diverted from its intended use to be used as transfer storage. As this is expected to entail additional processing latency, which should be avoided, this solution is not further pursued.

5.3.5 Detailed implementation

The order of the modifications to the individual components follows the control flow of the request processing in the following description.

index.php

The *index.php*, which at first receives every request, has been modified to only allow GET requests, if both HTTP/2 and TLS is used. This accommodates the fact that is possible to use HTTP/2 without TLS encryption, what would lead to the sensitive parameters being transferred unencrypted. If HTTP1.1 is used, there is no point in using a GET request, as the parameters cannot be transferred anyway.

```
1
2  //Allow GET ONLY if Protocol is HTTP/2 and HTTPS is used (upgrade
       allowed)
3  if ($_SERVER["SERVER_PROTOCOL"] !== "HTTP/2.0" || ($_SERVER["
       SERVER_PROTOCOL"] === "HTTP/2.0" && !isset($_SERVER["HTTPS"]))) {
4          if (count($_GET) > 0) {
5                  header("HTTP/1.1 405 Method Not Allowed");
6                  header("Allow: POST");
7                  header("Content-Type: text/plain");
8                  print("Only HTTP POST requests are allowed!");
9                  print("GET only allowed for TLS secured HTTP/2
                       connections");
10                 exit(0);
11         }
12 }
```

Listing 5.5: Allowing GET requests only for TLS enabled HTTP/2

As already mentioned in 5.2.2, the parameters reside inside the header, from where they are extracted and inserted into the $_POST array. From this point on, their processing does not differ from the values transmitted via POST request. The extraction also comprises the removal of the prefix *s-* to mark them as belonging to SANE, as can be seen from 5.6

```
1
2  # extract data from http/2 headers if available when key starts with "S
       -" (for SANE ;)
3  #
4  $headerString="";
5  //if (function_exists("getallheaders")) {
6          foreach (getallheaders() as $key => $value) {
7                  $headerString.=$key.":".$value.";";
8                  if (substr($key,0,2) === "S-" || substr($key,0,2) === "
                       s-") {
9                          $seperatorPos = strpos($key,"-");
10                         $targetKey=lcfirst(substr($key,$seperatorPos+1)
                               );
11                         $_POST[$targetKey] = $value;
12                 }
```

Listing 5.6: Parameter header extraction

Apart from that, the definition of certain constants was moved to *config.inc.php* as the execution of the components, whose content is to be pushed, would otherwise be lacking this definitions, as in their context the *index.php* is not executed beforehand.

method_includer.php, cs_includer.php and Submission_Checker.php

The next components down the line *method_includer.php* and *cs_includer.php*, which show a similar structure, were slightly modified to extract and evaluate the setting that indicates whether the requested method is pushable or not alongside the other description from their according files and the user's desire, expressed by the $_POST parameter *useserverpush*. At the end, these files then include the new component *pushtrigger.inc.php*, which triggers the actual Server Push if all of the preconditions are fulfilled. If not, the request is answered regularly.

As the keys of *HEADERS* frames are limited to lower key characters, a slight modification of the component that validates the parameters (*Submission_Checker.php*, which is used by both the *method_includer.php* and the *cs_includer.php* was necessary. The parameter keys can anyway be set in uppercase, but are implicitly converted to lowercase already on the client side.

pushtrigger.inc.php

The component triggering the push, viewable in 5.7, first encodes the method values as JSON (line 4), hashes the resulting string (line 5) and stores them in Memcached using the hash value as key (line 7). This key is finally attached to the URL that is sent via link header in order to induce a Server Push (line 17) as a typical parameter of GET requests.

```
1  if($_SERVER["SERVER_PROTOCOL"] === "HTTP/2.0" && $useServerPush) {
2      $mcd = new Memcached();
3      $mcd->addServer($_CONFIG["memcachedServer"], $_CONFIG["
           memcachedPort"]);
4      $jsonMethodValues = json_encode($_POST);
5      $jmvId = hash("sha256", $jsonMethodValues);
6      //If setting memcached entry fails continue with normal execution
```

```
7       if($mcd->set($jmvId, $jsonMethodValues, $_CONFIG["memcachedTimeout"
            ])) {
8           $actualBaseUrl =($_CONFIG["ssl"] ? "https://" : "http://").
                $_SERVER["HTTP_HOST"] . substr($_SERVER["REQUEST_URI"], 0,
                strrpos($_SERVER["REQUEST_URI"], "/"));
9           header("Connection: Keep-Alive");
10          header("Expires: 0");
11          header("Cache-Control: must-revalidate, post-check=0, pre-check
                =0");
12          header("Link: <" . $actualBaseUrl . "/h2push_includer.php/?
                jmvid=" . $jmvId . ">; rel=preload; as=document", false);
13          exit(0);
14      }
15  }
```

Listing 5.7: Component triggering the Server Push itself (debug mode lines omitted)

h2push_includer.php

The *h2_pushincluder.php*, now on the other web server thread whose result is then about to be pushed, extracts the method parameters using the transmitted key and includes the method by its name, expressed in the parameter *method*.

```
1   include_once(getcwd()."/config.inc.php");
2   $jmvId = $_GET["jmvid"];
3   //restore method values containing post params from transfer storage
4   $mcd = new Memcached();
5   $mcd->addServer($_CONFIG["memcachedServer"],$_CONFIG["memcachedPort"]);
6   $method_values_json = $mcd->get($jmvId);
7   $method_values = json_decode($method_values_json,true);
8   //"invoke" according method on according crowdsourcing
9   if ($method_values['cs'] === "SANE") {
10      include_once (getcwd() . "/methods/" . $method_values["method"] . "
            .php");
11  }
12  else {
13      include_once(getcwd()."/cs_deploy/". $method_values["cs"] ."/
            methods/". $method_values["method"] .".php");
14  }
```

Listing 5.8: Component *h2_pushincluder.php* that extracts values and delegates processing to the according method

5.3.6 Proof of functional capability

In order to proof the functional capability of Server Push on the partial link between client and proxy both execution paths have to be evaluated. For this reason, to proof Server Push functionality for SANE's own methods, simply another method has been added, while to proof the crowdsourcing methods, a new crowdsourcing derivate with the following method has been added. Apart from the different file location, both have the same content why only one of them are shown in listing 5.9.

```php
<?php
header("Content-Type: text/plain");
var_dump($method_values);
?>
```

Listing 5.9: returnMethodValuesServerPushProofCS

These methods merely return an array dump of the transmitted parameters to the client. The proof itself is, similar to the principal proof of functionality in 5.2.2, shown by in-depth analyzing the transmitted packets via Wireshark. The payload size is based on the existing method with the largest payload when JSON encoded *correctWLANFingerprintingPosition*. As also the method's name is part of the payload, with *returnMethodValuesServerPushProof* method name with equal length of 33 characters has been chosen.

To address the test methods the *index.php* is called with the header entries from 6.1, which are common to both execution paths.

HEADERS key	HEADERS value
s-useserverpush	1
s-largeparameter1	<2048 byte randomized string>
s-mediumparam1	<512 byte randomized string>
s-mediumparam2	<512 byte randomized string>
s-deviceid	<64 byte randomized string>
s-username	<64 byte randomized string>
s-password	<64 byte randomized string>
s-server	<32 byte randomized string>
s-signature	<4096 byte randomized string>

Table 5.1: Common test method headers

SANE methods

The values to be set for executing the SANE method path can be seen in table 5.2.

HEADERS key	HEADERS value
s-method	returnMethodValuesServerPushProof
s-cs	SANE

Table 5.2: SANE specific method headers

The index.php is accessed via the Chromium browser with the above mentioned header values set. After establishing the connection and setting its parameters, the Wireshark protocol in *C – P Server Push proof SANE path* in appendix C shows the request itself in frame 26 (line 1 and following) in form of a HTTP/2 *HEADERS* frame, containing the above stated settings amongst other header fields set by the browser which have been omitted. The *:path* header (line 120) indicates that indeed SANE's front controller *index.php* is requested, while the header entries *s-method* (line 13) and *s-cs* (line 14) conform to the settings in table 5.2. This request is then responded to in the next Ethernet frame 28, which contains amongst frames belonging to the regular answer a *PUSH_PROMISE* frame (line 32 and following), where the stream id to use is advertised (line 34). The *:path* header entry (line 35) shows the source of the content to push, including the hash value used as key for the transfer storage, which is used later on to extract the intermediately stored method value parameters. In frame 29 (line 45 and following), the push itself is introduced by a *HEADERS* frame on the preassigned stream id 2 (line 53 and 55),

followed by *DATA* frames inside the subsequent Ethernet frame containing the actual data, also on stream id 2 as can be seen from the lines 67 and 70. This proofs that indeed a Server Push is used executing the newly introduced SANE method *returnMethodValuesServerPushProof*, which serves as an example for all SANE methods, using the implementation of 5.3.5. ■

Crowdsourcing methods

As we want this test to follow the crowdsourcing path the parameters are set as shown in table 5.3.

HEADERS key	HEADERS value
s-method	returnMethodValuesServerPushProofCS
s-cs	ServerPushTest

Table 5.3: Crowdsourcing specific method headers

The *s-method* parameter must not be the same as in SANE or it will be removed by validation logic and hence not executed.

This proof follows the exact same pattern as the previous proof 5.3.6: The index.php is, again, accessed via the Chromium browser with the above mentioned header values set. After successful establishment of the connection and setting its parameters, the Wireshark protocol in $\mathcal{C} - \mathcal{P}$ *Server Push proof CS path* shows the request itself in frame 29 (line 1 and following) in form of a HTTP/2 *HEADERS* frame, containing the above stated settings amongst other header fields set by the browser which have been omitted due to irrelevance for this proof. The *:path* header (line 12) proofs that indeed the *index.php* is requested, while the *s-method* (line 14) and *s-cs* (line 15) conform to the settings in table 5.3. This request is then responded to in the next Ethernet frame 31, which contains a *PUSH_PROMISE* frame (line 32 and following) where the stream id to use is advertised (line 34), alongside a *HEADERS* and a *DATA* frame. The *:path* header entry (line 35) shows the source of the content to push, including the hash value used as key for the transfer storage, which is used later on to extract the required method value parameters. In frame 32 (line 45 and following), the push itself is introduced by a *HEADERS* frame on the reserved stream with id 2 (line 53), followed by *DATA* frames containing the actual data, also on stream id 2 as can be seen from the lines 67 and 70. This proofs that indeed a Server Push is used for the newly introduced crowdsourcing method *returnMethodValuesServerPushProofCS* on the *ServerPushTest* derivate, representing all CS methods, using the implementation of 5.3.5. ■

5.4 REMARKS TO UPCOMING IMPLEMENTATIONS FOR SANE

For Stream Reset on the partial link $\mathcal{C} - \mathcal{P}$ and HTTP/2 and Header Compression for the $\mathcal{P} - \mathcal{S}$ link to work as intended, the following has to be taken into account.

Apart from that shall findings from the next chapter 6 already be foreclosed here for the sake of completeness of this section.

5.4.1 $\mathcal{C} - \mathcal{P}$ Stream Reset

The test in 5.2.1 shows that stream resetting already works for the $\mathcal{C} - \mathcal{P}$ partial link. The first line of the according code snippet sets via *ignore_user_abort(<Boolean>)*, how the PHP execution environment reacts on the reception of a *RST_STREAM* frame. In the default setting *false*, PHP stops the script execution immediately, meaning it is not able to react on this event, as this would require further execution. If set to *true*, to recognize this, the currently running script has to actively check for the stream still being active by using *connection_aborted()* as, at this juncture, no way of event handling exists for this case. Furthermore, for this function to notice a stream being closed, it has to have sent data right before the invocation of this function, as can be seen in the source code of *dl-closetest.php* under *$\mathcal{C} - \mathcal{P}$ Stream Reset test source code* in the appendix A. Unfortunately, the callback function defined with *register_shutdown_function(<callbackFunction>)* unexpectedly is not invoked in this case.

The fact that the closing of a stream, or even a connection with HTTP1.1, results in an immediate stop of execution may lead to data corruption, as operations altering data are only partially executed. If Stream Reset shall actually be implemented for SANE, hence a new component should be introduced that handles the actual response transmission while checking for an abortion of an operation. In this case it could delegate the handling back to the invoked SANE method itself, as only it knows how to handle an execution abort in a graceful manner, for example by rolling back the effects of non-idempotent operations or setting a *dirty* state.

5.4.2 $\mathcal{P} - \mathcal{S}$ Multiplexing

Due to the limitations of PHP 5.6, as found in 5.2.3, every transmission, even though already using HTTP/2, still employs an own connection. With PHP 7.1 however, when it is guaranteed that the SANE works as intended, for the HTTP/2 functions that have already been implemented by Pu in the course of his thesis to unfold its full potential, outgoing requests have to be multiplexed onto the same connection. Otherwise for each request also a new connection would be used, bringing all side effects for establishing a secure TLS connection along. As the proxy only deals with a limited amount of destination servers, it is highly likely that in case of a forwarded request already an open connection to the destination server exists that can be reused.

As the description of the multi interface of libcurl by Daniel Stenberg in [Ste15] allows the interpretation that libcurl is able to do *automatic smart multiplexing* over separate multi handles and even though it is not expected to work it is worth to look into as this would simplify multiplexing massively since no thread- or process-spanning component had to be developed that does this explicitly. For that reason an additional test to evaluate this has been conducted in 6.3.1 with a negative outcome. Hence, to multiplex requests and their responses onto the same connection a central *connection broker* component is required that handles the dynamic adding and removing of single handles from/to a single SANE-wide multi handle.

5.4.3 Employing FastCGI

As will be shown in the next chapter or more specifically in 6.3.2 and 6.3.3 should FastCGI or better the PHP FastCGI Process Manager be employed to allow libcurl to perform requests via *multi handle* in parallel. Apart from a missing *getallheaders()* function, which has been re-implemented for the case it is not supported, there is no difference in execution regarding the newly implemented features for the SANE when it employs FastCGI instead of the internal Apache httpd PHP module.

For further information please consult the above referenced parts of this document.

5.5 SUMMARY

Of all the advancements of HTTP/2 that were found to be advantageous in the Concept, only *Header Compression*, *Stream Reset* and *Server Push* for both partial links were found to be feasible according to official documentation of PHP and libcurl. Due to Header Compression being already implemented in the course of [Pu16] and limitations of libcurl that became obvious not until the implementation of the functional demonstrations, the only currently actually feasible advancement under usage of PHP 5.6 is Server Push on the $\mathcal{C} - \mathcal{P}$ partial link. This feature was then also implemented under usage of a workaround that allowed putting the parameters into the HPACK header, in turn making it possible to transmit requests with heavy payload using GET requests, as POST requests were found not to be able to trigger a Server Push. Nevertheless, it is still possible to use the SANE in a *traditional* manner employing POST requests - even though of course without Server Push. The SANE, hence, remains fully downward compatible.

All of the other improvements, however, presume PHP 7.1 to be used. This especially applies also for the use of Multiplexing, alongside Server Push and Header Compression probably the biggest improvement of HTTP/2. Above that, Multiplexing is naturally a necessary requirement for using the Stream Reset feature, as only if multiple stream reside on a single connection, there is a point in resetting a stream within.

For these reasons it is highly recommended to ensure that the SANE is fully compatible to this version of PHP, proven by carefully evaluating constraints and implementing unit tests for regression testing and debugging. Then, Server Push and Stream Reset can be implemented similar to the demonstrations $\mathcal{C} - \mathcal{P} - \mathcal{S}$ Server Push and $\mathcal{C} - \mathcal{P} - \mathcal{S}$ Stream Reset.

6 EVALUATION

“

Es gibt für uns Physiker nur noch die Kapitulation vor der Wirklichkeit.

Friedrich Dürrenmatt in *Die Physiker*

”

During the previous chapter, the solutions that have been found beneficial for SANE have been implemented or proven to work in principle if the employed software framework did not permit a direct implementation on the SANE yet. The purpose of this chapter now is to quantify the effect these implementations have or might have in the future, based on the evaluation criteria defined in 4.4.

First Pu's results regarding SANE's general performance under use of HTTP/2 are discussed once more. Secondly, the advancements that were implemented on the SANE or could be implemented in the future, as soon as it is ensured that the SANE is capable of working on PHP 7.1, will be evaluated in the following. Thirdly, additional tests will be conducted that evaluate so far untreated questions that could limit the advancements of HTTP/2 that have been conceptually found to be beneficial to the SANE.

6.1 GENERAL PERFORMANCE ASSESSMENT

As the previous work of Pu [Pu16] has already evaluated the overall performance of HTTP/2 including Header Compression, the measurements are not repeated here. His results, however, are picked up and discussed once more to provide a comprehensive evaluation of HTTP/2's performance. Above that it is necessary to re-discuss the results as they were produced under the assumption that the HTTP/2 connections make use of multiplexing, what they in fact did not.

In his work Pu makes use of a crowdsourcing method called *viaSANE* to test the entire virtual $\mathcal{C} - \mathcal{P} - \mathcal{S}$ link with the various possible configurations introduced in 4.1.1. The used method inserts a transmitted *message* parameter with a length between 0 and 64 characters into the proxy-local database and forwards a SHA256 hash of the transferred data using libcurl, yet without multiplexing, to a crowdsourcing server instance. This test has been conducted using a both local and remote crowdsourcing server. The tests were carried out with 10 clients using 100 concurrent connections (n100/c10), 5 clients with 50 concurrent connections (n50/c5) and 1 client with only 1 (n1/c1) connection. As the SANE constraints every connection to be TLS secured, this is expected to apply on every connection in this evaluation, although not explicitly stated.

As Pu uses a different denomination in the diagrams the following table should help to clarify any ambiguities.

Diagram name	Concept name	$\mathcal{C} - \mathcal{P}$	$\mathcal{P} - \mathcal{S}$
H2/H2	straightforward proxy	HTTP/2	HTTP/2
H2/H1	downgrade proxy	HTTP/2	HTTP1.1
H1/H2	upgrade proxy	HTTP1.1	HTTP/2
H2/H1	status quo	HTTP1.1	HTTP1.1

Table 6.1: SANE Architecture configuration denomination overview

The test run on the local crowdsourcing server showed an almost equal total runtime for both the n1/c1 and the n50/c5 configuration. The results can be seen in figure **??** As expected was HTTP/2 employed on both $\mathcal{C} - \mathcal{P}$ and $\mathcal{P} - \mathcal{S}$ partial links the fastest, while HTTP 1.1 on both partial links was the slowest. With n100/c10 also HTTP/2 was the fastest with 732 ms total runtime. Interestingly, all other configurations took significantly longer for the downgrade proxy (921 ms), the upgrade proxy (1293 ms) and the configuration named *status quo* in 4.1.1 where HTTP 1.1 is used for both partial links (1876 ms). These results indicate that HTTP/2 scales much better under high load. Furthermore seems the $\mathcal{C} - \mathcal{P}$ link to be much more susceptible to high load than the $\mathcal{P} - \mathcal{S}$ link. This behavior may however be rooted in the absence of multiplexing by

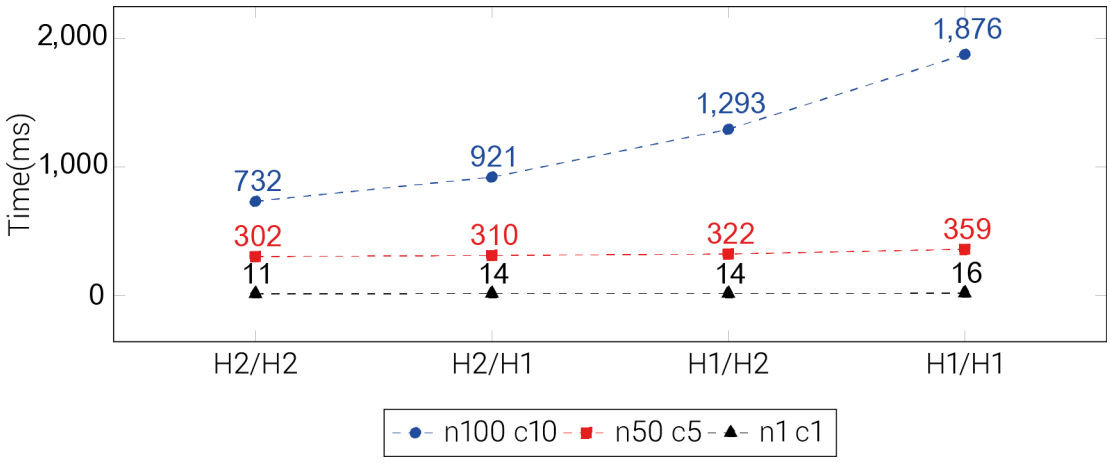

Figure 6.1: Local SANE HTTP/2 performance (from [Pu16])

the used implementation without a central instance managing outgoing connections. Compared to diagram 6.2 the timings of the upgrade proxy and the *status quo* configuration were even longer than for the remote execution, suggesting that in this cases the usage of HTTP 1.1 pushed the system to its load limit what would in turn indicate a generally lower system load due to not having to parse the text based data of HTTP1.1 with HTTP/2.

Taking the mean time consumption with 10 clients and 100 connections into account shows that the straightforward configuration (17.80 ms) is outperformed by both the upgrade (14.84 ms) and the downgrade configuration (13.33 ms). Only the *status quo* configuration with HTTP 1.1 on the entire $\mathcal{C} - \mathcal{P} - \mathcal{S}$ is insignificantly slower. The almost equal performance during connection establishment can be explained by the employed test tool *h2load*'s behavior of using cleartext HTTP/2 (h2c) by default, which takes a detour over HTTP 1.1 for the first request which is then upgraded to HTTP/2. When actual data is transmitted during the response, the *straightforward* configuration employing HTTP/2 performs significantly better with a mean response time of 68.34 ms, compared to 87.15 ms of the downgrade proxy configuration, 115.65 ms of the upgrade proxy configuration and 168.49 ms for the *status quo* configuration, showing the higher processing speed of HTTP/2 using header compression and binary encoding. As the tests were conducted locally, the resulting size of the data to transfer can be neglected as factor.

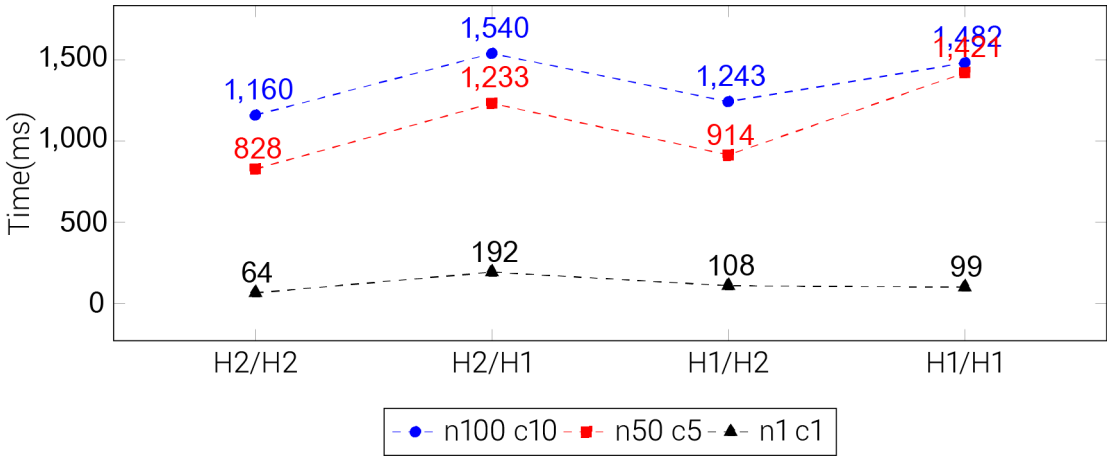

Figure 6.2: Remote SANE HTTP/2 performance (from [Pu16])

However, the table is turned with a remote crowdsourcing server, as can be seen in figure 6.2. Again the straightforward configuration is the fastest while interestingly the downgrade

configuration is yet faster with n50/c5 and n1/c1, but slower than the *status quo* configuration using only HTTP1.1 for both partial links under high load (n100/c5). The upgrade configuration, however, is in all cases faster. This indicates that HTTP/2 can play out its strengths over distant, latency afflicted connections and even under high load.

Certainly, these results are expected to be much better using multiplexed connections for the $\mathcal{P} - \mathcal{S}$ partial link, as they already are on the $\mathcal{C} - \mathcal{P}$ link, simply by employing a HTTP/2 compliant web server.

6.2 ADVANCEMENT SPECIFIC ASSESSMENT

In this section, the HTTP/2 advancements that have been implemented on the SANE or whose functional capability has been proven are about to be evaluated. To minimize side effects induced by the network, all tests are performed on the local machine.

6.2.1 Stream Reset

To evaluate Stream Reset, the latency of the Apache httpd web server's reaction to the reception of a *RST_STREAM* frame to rate the compliance to the official HTTP/2 standard [BPT15] was assessed. According to the standard, the web server must not send further *DATA* packets after a Stream Reset frame has been received. For this assessment the protocols of the previously conducted functional tests with this topic 5.2.1, 5.2.3 (with PHP 7.1) and 5.2.5 were evaluated. In none of these protocols, further *DATA* frames were emitted after reception of a *RST_STREAM* frame. Hence, the used web server complies with the standard. The amount of data sent unnecessarily, the overhead in a manner of speaking, equals to zero.

Nevertheless, in practice, under employment of a network, the packet latency is larger than 0 ms, meaning that it is highly likely that during the time the *RST_STREAM* frame travels from its sender to the receiver, the receiver emits another *DATA* packet. This case is also provided in the HTTP/2 RFC, which also urges the involved peers to treat that case gracefully. However, it is safe to state that a reference implementation of HTTP/2 should stop sending additional *DATA* frames immediately, just as the Apache httpd does.

Besides the packet latency, also the load of the web server probably affects this proper reaction as it influences the processing of all kind of frames.

In order to make a well-grounded statement about the saved resources due to a reset stream on obsolescence, the evaluation has to take network latency and system- and network load under realistic usage patterns and with real data into account. Regrettably, this is at the at this juncture not feasible, as this implementation does not yet exist on the entire virtual $\mathcal{C} - \mathcal{P} - \mathcal{S}$ link due to unconfirmed support of PHP 7.1 by the SANE. If SANE supported PHP 7.1 and this feature was implemented for the entire virtual link, either realistic usage pattern had to be modeled or statistic evaluation had to be done over real usage data.

However, the insight that web servers react to the reception of a *RST_STREAM* frame immediately further encourages its projected use for the SANE to abort operations that have become obsolete since its invocation.

6.2.2 Server Push

To assess the Server Push implementation, on the one hand saved resources and the variation of total execution time and on the other hand the reaction to unexpected termination of streams or entire connections have to be taken into account, as will be in the following.

Performance

Open and *half-closed* streams both reserve memory for the TCP buffer and "count toward the maximum number of streams that an a endpoint is permitted to open" [BPT15]. Moreover must, for a stream in an open state, the according window size be maintained, what requires additional resources by means of CPU cycles and memory. These characteristics, however, do not apply on streams in a *reserved* state, to which streams promised via *PUSH_PROMISE* count. They remain in this state until actual data is transmitted. For this reason the effectivity of Server Push can be expressed by the total time all streams remain in an non-closed or non-reserved state for a given operation. Above that, naturally, also the total time from issuing a request to the reception of the response has to be taken into account, as the overall goal of HTTP/2 and one of the reasons for incorporating HTTP/2 for SANE is the reduction of this period.

At this juncture, no automated test tool to issue a request and receive a Server Push response, let alone evaluate the overall time a stream remains in an open state, exists, why the evaluation had to be conducted manually using Wireshark. Conducting this tests manually unfortunately only allowed for a limited amount of repetitions.

For this evaluation, a newly introduced SANE method *returnMethodValuesServerPushDelayedCS* from listing 6.1 is used, which resembles the long-running methods from 4.3.2 with unpredictable runtime, by returning the results only after a delay, which can be set via *delay* parameter on request. The delay of those methods result from the methods either having to retrieve an arbitrarily large data set or having to employ another network connection to retrieve the data. Apart from that, this method is based upon *returnMethodValuesServerPushProofCS.php*, which was already employed for the functional proof of the SANE's Server Push implementation in 5.3. This method is then either called *traditionally* via POST, without using Server Push, or via GET with the parameter *s-useserverpush* with the value *1* set in order to enable the use of Server Push.

```
1 <?php
2 //delay server answer according to delay parameter
3 sleep($method_values['delay']);
4 var_dump($method_values);
5 ?>
```

Listing 6.1: returnMethodValuesServerPushDelayedCS

The timings are extracted from the Wireshark protocol, which can again be found under *Evaluation SANE Server Push performance* in the appendix C, as follows. The execution time is defined as the time from the first Ethernet frame to establish the connection, which is not part of the protocol, to the time the last *DATA* frame is being emitted. As this first Ethernet frame triggers the recording, the timings of the individual frames in the protocol are correct. The *stream open timings* reflect the time passed from the timing of the *HEADERS* frame, which opens a stream to the time of the last *DATA* frame on that stream, indicated by the flag 0x01, which signals that no more frames are sent using this stream (*END_STREAM*), according to [BPT15], the official HTTP/2 RFC. For reasons of relative briefness of the test protocols and simplicity, the *DATA* frame was chosen as an end-mark instead of the according TCP ACK-packet

sent from the client to acknowledge the reception. As the tests were all conducted on the local machine, those two timings only differ minimally.

The evaluation runs where executed using PHP 5.6, just as the SANE implementation itself.

Delay 0	Without Server Push	With Server Push
Execution time	0.036997181	0.026483089
Time stream 1 open	0.025637935	0.018024038
Time stream 2 open		0.000033343
Total stream open time	0.025637935	0.018057381

Table 6.2: Timings of *returnMethodValuesServerPushDelayedCS.php* without delay

Delay 1	Without Server Push	With Server Push
Execution time	1.038798693	1.046927065
Time stream 1 open	1.021375243	0.025733594
Time stream 2 open		0.000031963
Total stream open time	1.021375243	0.025765557

Table 6.3: Timings of *returnMethodValuesServerPushDelayedCS.php* with 1 second delay

The figures 6.3 and 6.4 depict the the values from the tables 6.2 and 6.3. The columns show the overall execution time of the operation, while the light blue part shows the fraction of the execution time the involved streams remained in an open state.

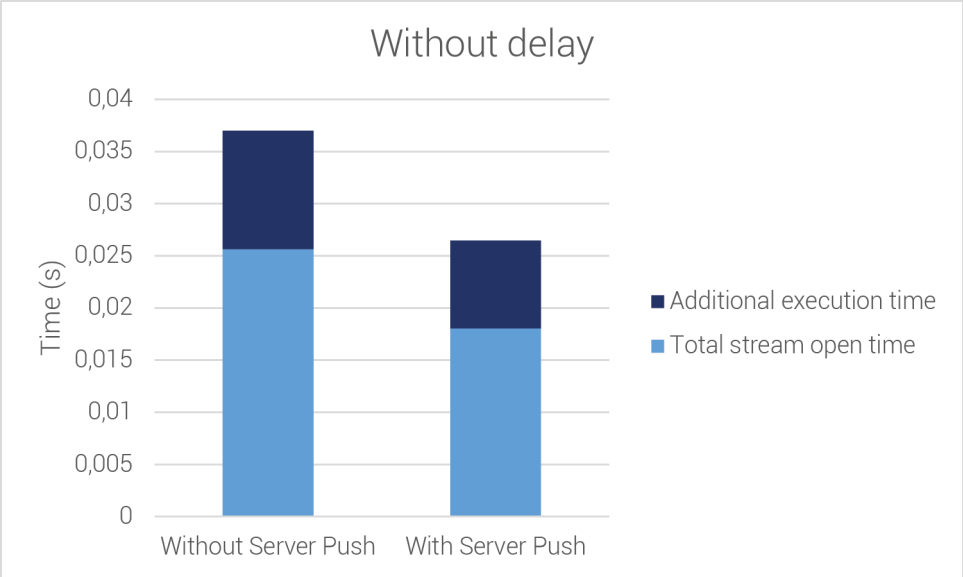

Figure 6.3: Timings from table 6.2 (without delay)

Although the tests were only run one time for each delay and configuration and are therefore not statistically significant, the results show that the additional effort of delegating the processing to another thread via link header and having to use additional hashing and a transfer storage do not introduce a delay itself regarding the total execution time. Instead, the total amount of time streams had to be kept open are significantly reduced under employment of Server Push. This means, the more time the long-running methods take, the more reasonable the employment of Server Push gets.

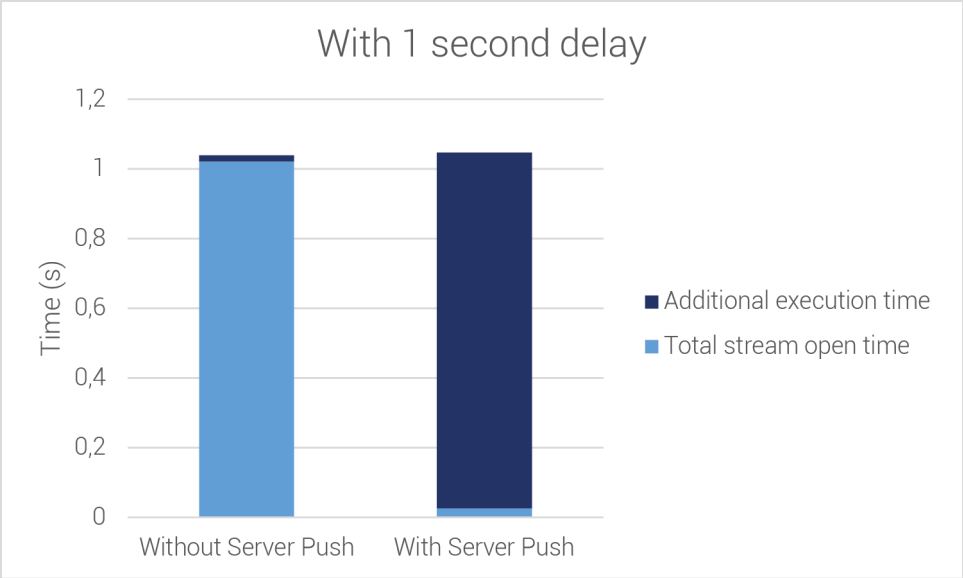

Figure 6.4: Timings from table 6.3 (1 second delay)

In practice, depending on system- and connection load, the results of course may deviate significantly. Also due to lacking realistic usage data or load testing tools supporting Server Push it is regrettably currently not feasible to make a well-grounded statement regarding execution time under load. For this reason, this evaluation has to be postponed until the Server Push implementation is deployed to production. Then the same metrics can be used to rate the effectivity of Server Push.

Publish/subscribe pattern

Due to the relatively vague description in the official HTTP/2 RFC regarding the sending of additional *PUSH_PROMISE* frames over a pushed stream, this will be examined more closely in the following. To do this, the above used *returnMethodValuesServerPushDelayedCS.php* is extended to issue another link-header to promise another stream. The source code of *returnMethodValuesServerPushDelayedNewPush* can be seen in Listing 6.2, while the according Wireshark protocol can be found under *SANE Server Push Publish/Subscribe evaluation* in appendix C. It has again been stripped of HTTP/2 frames irrelevant to this evaluation.

```
1 <?php
2 //delay server answer according to delay parameter
3 sleep($method_values['delay']);
4 header("Connection: Keep-Alive");
5 header("Expires: 0");
6 header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
7 header("Link: <https://localhost/poc/filestream.php>; rel=preload; as=
      document", false);
8 var_dump($method_values);
9 ?>
```

Listing 6.2: returnMethodValuesServerPushDelayedNewPush

After the reception of the request in frame 28, the following Ethernet frame 55 (line 26 and following) contains the ordinary answer consisting of *HEADERS* and *DATA* frame alongside the

PUSH_PROMISE frame, promising stream id 2 (line 36) for the Server Push. Ethernet frame 79, beginning in line 47, then contains the Server Push's *HEADERS* frame, indicated by the used stream id 2 (line 55). Although it also contains a link header (line 58) referencing the well-known *filestream.php* script, no additional *PUSH_PROMISE* frame is sent for this resource. Hence, the expected behavior, to not being able to induce further Server Pushes on an already pushed stream, has been confirmed. ■

Fault Tolerance

In order to encounter possible problems at an early stage and to be able to handle them gracefully, it has to be examined if the SANE reacts differently to connection errors under usage of the newly introduced Server Push feature and if this reaction could possibly even lead to data corruption. More specifically is to be examined what happens in case of an operation being interrupted after a *PUSH_PROMISE* has already been sent, thus the belonging stream has been put into *reserved* state, but no data has yet been sent.

For this evaluation, the script of the above conducted performance evaluation is reused without any change to its code. As the only difference the time between invocation of the according SANE method *returnMethodValuesServerPushDelayedCS.php* and the output of data has been prolonged by setting the*delay* value passed by the client to *10* seconds. This allows for easily canceling the request that is sent via Chromium browser manually or closing the connection unexpected by simply *killing* the Chromium browser process, not allowing it to gracefully end the communication to the server.

In the former case the client emits a *RST_STREAM* frame, as can be seen in frame 30 (line 28) of the first Wireshark protocol under *SANE Server Push fault tolerance evaluation* in appendix C. After the reception of this frame, neither any further *DATA* packet is sent nor any output to the Apache error log is being written.

Regarding the latter case and protocol in *SANE Server Push fault tolerance evaluation*, if the client Chromium browser is terminated immediately via *pkill -SIGKILL chromium-browser*, which conforms to a termination the target process cannot *block*, the web server ends the connection by issuing a *GOAWAY* HTTP/2 frame to end the connection in line 47, after the TCP connection has been finalized by the preceding [FIN] TCP packet in line 33, sent by the client's network stack, which recognized the unexpected termination of the browser's connection. As expected also no further output is written to the error log, hence the script execution is stopped likewise.

The SANE under usage of Server Push hence behaves exactly as without using the new Server Push implementation regarding interrupted requests.■

These findings coincide with the findings of 5.2.1 and as already remarked in 5.4.1: If the script execution is supposed to continue, user abortion should be ignored by setting *ignore_user_abort(true)* for each script concerned. Moreover, to avoid possible data corruption it is strongly recommended to either continue execution and handle these cases manually or employ some kind of persistence component that rolls back the changes made in case of an unexpected abortion i.e. makes use of transactions for non-atomic operations on the database.

6.3 ADDITIONAL ASPECTS

Aside from proving that the advancements work in principle, what has been done in chapter 5 and quantifying them in the previous sections of this chapter, additional constraints have to be fulfilled in order to unlock the full potential of HTTP/2's advancements. This evaluation shall be done in the following.

6.3.1 Automated Smart Multiplexing

In addition to the ability of libcurl to combine several requests onto the same connection by adding a single handle explicitly to a multi handle, it would be necessary to have them multiplexed onto the same connection even if not explicitly stated, as it is the case with SANE if distinct requests are performed on the same destination server. According to Stenberg's libcurl blog [Ste15], libcurl should be capable of doing so with the setting in listing 6.3, where it says "If you use the multi interface and enable pipelining, libcurl will try to re-use established connections and just add streams over them rather than creating new connections". As this statement leaves it open whether or not this reusing of connections happens implicitly, this has to be evaluated.

```
1 curl_multi_setopt($mh, CURLMOPT_PIPELINING, CURLPIPE_MULTIPLEX);
```

Listing 6.3: Libcurl setting to enable automated multiplexing

For this test two separate requests to a server are posed under usage of libcurl, comparable to other tests on the $\mathcal{P} - \mathcal{S}$ link. In order to verify that streams opened by different web server threads are multiplexed onto the same connection, each of this requests is realized using its own multi handle in which a single handle hooks in. If libcurl behaves as intended, only one connection should be used for the two requests to the same origin. The proof itself is again conducted via Wireshark analysis of the traffic between libcurl client and server. The script containing the libcurl code to perform the requests to multiplex is called by not only two different browser instances, but even two different browsers. The target of the requests issues via libcurl is again *idlestream.php* that has already been used in previous proofs. It merely returns *0* every 100 ms until aborted.

The Wireshark protocol, which can be found under *Evaluation of Smart Multiplexing on $\mathcal{P} - \mathcal{S}$ link using libcurl* in appendix C, shows the communication after successfully establishing the first connection. The establishment itself has been, just as in previous proofs and tests, for the sake of relative briefness, been stripped of all connection related frames and recurring *DATA* frames. Also does this protocol only contain the communication of the libcurl client by only letting through HTTP1.1 and HTTP/2 frames on port 80, as again cleartext HTTP/2 (h2c) is used in order for Wireshark to be able to record the otherwise TLS secured communication.

Frame 1 in line 1 shows the first request to *idlestream.php* from the libcurl client before the upgrade to HTTP/2. The request is answered in frame 6 (line 9) with a *HTTP/1.1 101 Switching Protocols* message by the server, signaling an upgrade to HTTP/2. The following Ethernet frame then contains a HTTP/2 *HEADERS* frame and also a *DATA* frame, while the Ethernet frame 14 after that only contains a *DATA* frame. Frame 37, from line 33 on, shows then the second request to *idlestream.php*, where the same procedure as above gets repeated. The Ethernet frame 53, containing again one *HEADERS* and one *DATA* HTTP/2 frame, then is the first frame in which the multiplexed communication should occur. The client port number 45732 in column 6, however, differs from the client port used by the first established communication, port 45728. It hence uses a different connection, rather than only a different stream. This only allows making the conclusion that multiplexing is not used in this case.

A plausible explanation for multiplexing acting up, against the assertion in [Ste15] by curls creator Daniel Stenberg, could be the usage of cleartext HTTP/2, as it takes a detour over HTTP 1.1. This leads to a predicament: Employing HTTP/2 with TLS encryption thus could make libcurl combine streams automatically onto a single connection, but without libcurl being able to log the pre-master keys necessary to decrypt the TLS encryption, it is not possible to observe this using Wireshark. Hence, from the Wireshark protocols can only be inferred that with cleartext HTTP/2 requests via libcurl are not multiplexed automatically.

In order to confirm or rebut this finding, the option *CURLOPT_VERBOSE* for the single handle attached to the multi handle in the *multipair.php* was set to enable libcurl to return a more verbose connection log. The test then was repeated under employment of HTTP/2 with TLS encryption (h2) instead of cleartext HTTP/2 (h2c). Regrettably, the libcurl log, which can be also be found in appendix C, shows two connection establishments using stream id 1 (line 21 and line 54) with different easy handle ids in the same lines. The belonging date declaration in lines 28 and 61 with a difference of merely 4 seconds indicate that these two requests indeed belong to the same test run. This means that also under use of h2, two separate connections are being used, hence no automated connection multiplexing is done. ■

Curiously, the curl log shows for both connections the use of of HTTP1.1 GET in line 22 and 55, even though the use of HTTP/2 has already been confirmed previously (in lines 11 and 18) for the first connection and (in lines 44 and 51 for) the second connection.

Tests that are comparable to those above repeated under use of PHP over FastCGI instead of running as an Apache httpd web server module arrived at the exactly same result: No multiplexing has been done *automatically*. The reason for rechecking this under usage of FastCGI was the possible requirement of a multi threaded execution for the libcurl multi handle to do automated multiplexing, as it indeed is a requirement for parallel execution as will be shown in the following section. The exact results are omitted due to its identical outcome.

However, these results strongly suggest that for SANE, to fully support connection multiplexing, it is necessary to develop an own *connection broker* component, which handles outgoing requests via one central multi handle to which it adds and removes single handles in a dynamic way. Admittedly, there is also a chance that the multi interface is used incorrectly above due to the still sparse documentation of libcurl with HTTP/2. As soon as more details are announced and usage examples exist, this should be reviewed.

6.3.2 Parallel execution of libcurl requests

During the demonstration of 5.2.5 in the previous chapter has emerged that HTTP requests issued via libcurl multi handle were executed sequentially instead of parallelly, as expected. It stands to reason that this behavior is rooted in the multi-process but single-threaded nature of the Apache httpd module used for executing PHP code. To verify this, Apache httpd was reconfigured to use the *PHP FastCGI Process Manager* instead of its internal PHP 7.1 module in combination with the *mpm_worker* module, which allows real multi threading, in contrast to the *mpm_prefork* module, which spawns an own process for every script execution. As libcurl naturally has do the requests in parallel, the obvious conclusion is that multiple threads have to be employed internally. The libcurl documentation however states that one objective of the multi handles is to "Enable multiple simultaneous transfers in the same thread without making it complicated for the application" [Steb], stating that multiple requests can indeed be done using just one thread. Hence, the issue of not being able to do parallel requests using the internal php module is likely to be rooted in some other reason, which will however not be further evaluated in the course of this thesis.

Compared to the internal PHP module of Apache httpd, the external processing via PHP-FPM leads to a slightly different behavior regarding the web server's responses. Instead of continuously flushing the current results out, the external module returns the result as a whole and only if the operation is not canceled on the way. For this reason, the test script used in in the 5.2.5 demonstration is inapt for this one, as it continues until canceled, what would not return any result at all. Instead, the existing script that does two requests within a multi-handle was slightly modified to request another script, which terminates autonomously after about 25 seconds and has also previously been used: *Filestream.php*, which can be found in appendix A attached to where it was first used, in $\mathcal{C}-\mathcal{P}$ *Server Push Test*.

The proofs are once again conducted via Wireshark; the protocols can again be found under *Evaluation of parallel execution of libcurl requests* in appendix C, they have again been stripped of HTTP/2 frames not essentially necessary for the proof, what embraces *SETTINGS*, *WINDOW_UPDATE* and recurring *DATA* frames. Above that, they also only contain the traffic issued by libcurl, not the traffic that was generated by accessing the script containing the libcurl code by the client.

The test using FastCGI starts with a HTTP1.1 GET request to */poc/filestream.php* in frame 38, which is upgraded in the following frame 40 to HTTP/2 with a *HTTP/1.1 101 Switching Protocols* message. After the omitted *SETTINGS* frame, a *HEADERS* frame with number 46 can be seen starting from line 36. With this frame the other request to */poc/filestream.php* begins, as the *:path:* header entry in line 36 states. Exactly 25 seconds later, as can be seen from the second column from every Ethernet frame summary at the beginning, the Ethernet frame 58 is transmitted from the web server on port 80 to the client on port 59472 (line 44) containing not one but two *HEADERS* and *DATA* frames. The first *HEADERS* and *DATA* frames use thereby stream id 3 as the lines 46 and 52 tell, while the latter frames of the same type use stream id 1 (as lines 49 and 54 state), hence belonging to the first request, which was upgraded from HTTP1.1. After a total 25 seconds of execution time, the connection is ended via *GOAWAY* frame, starting in line 57. As request and response were transmitted simultaneously and took only 25 seconds, they were executed in parallel. ■

To make sure this behavior is due to the Apache PHP module, the same test was repeated with the Apache web server again configured to use the internal PHP module. The protocol is quite similar to the one above until the reception of the first response from the server in line 39 and following: Instead of an Ethernet frame containing two frames of each type *HEADERS* and *DATA*, there is only one with stream id 1 (lines 46 and 49), belonging to the first request, that has been upgraded to HTTP/2. After the reception of a total of 5 *DATA* frames on stream 1, another Ethernet frame containing a *HEADERS* and a *DATA* frame is transmitted, but this time using stream id 3, as can be seen from lines 69 and 72. After another 4 additional frames and a total execution time of 50 seconds, the connection is closed via *GOAWAY* frame by the server from line 161. The fact that the first chunk of data is transmitted via stream id 1, while the second is transmitted via stream id 3 and the total execution time takes 50 seconds instead of 25 above proves clearly that the single requests are executed sequentially instead of parallelly. ■

This means, it is absolutely necessary to use libcurl in the field with PHP over FastCGI.

6.3.3 External script processing

As in the previous section has just been found out does libcurl in order to issue parallelized request, PHP over FastCGI has to be used. To verify that the newly introduced code for the SANE is compatible to this requirement, the previous methods to test both execution paths introduced in 5.3.6 with the same parameters are used to verify an error free execution.

During the execution of both the methods using both the PHP FastCGI Process Manager (php-fpm) services in PHP version 5.6 and 7.1 the necessary function to extract request headers and therefore the SANE parameters was missing. Prior to using this function it was known that this function was first introduced by Apache under the name *apache_request_headers()* and was originally only available when PHP was run as an internal module, as the PHP manual for this function [PHP16c] states. This documentation however further states, that this function was introduced to the FastCGI Process Manager versions from PHP 5.4, why its availability in any case was taken for granted. However, in case the SANE runs on a platform, which does not provide this function, its functionality has been reproduced and integrated in *config.inc.php* to ensure the SANE's ability to not only run but also use the newly implemented Server Push on the $\mathcal{C} - \mathcal{P}$ partial link, irrespective of the employed php runtime environment.

After fixing this issue, both execution paths returned the same results as those from the original test to prove the SANE's functionality in 5.3.6.

To ensure also the findings of 6.2.2 hold for PHP over FastCGI, also this fault tolerance test was repeated for PHP version 5.6 and 7.1. Both tests showed the exactly same results as in the original one. ■

6.4 SUMMARY

For the general assessment of HTTP/2's performance the results from another recent work at the Technische Universität Dresden were re-used. These results confirmed the anticipated effects on the SANE from the concept and the findings from related work, cited in the according chapter: With HTTP/2, data transmission is going to be faster and less resource intensive. However, the test results did not take multiplexing on the $\mathcal{P} - \mathcal{S}$ link into account. For this reason the results are expected to be even better when this partial link is involved if a central *connection broker* that facilitates multiplexing is put to use.

Not only during proofing the concept, also while evaluating, the relative youth of HTTP/2 became obvious, as at this juncture of this thesis not only the most advanced client library for HTTP/2 *libcurl* has not yet implemented many more sophisticated features of HTTP/2, also automated test tools that necessarily also have to be tailored to the new features, are lacking the ability to test *Stream Reset* and *Server Push*. Those are the only advancements for which either a direct implementation for SANE, $\mathcal{C} - \mathcal{P}$ Server Push, or at least a functional demonstration, for *Stream Resetting* on all partial links and *Server Push* on the missing $\mathcal{P} - \mathcal{S}$ and $\mathcal{C} - \mathcal{P} - \mathcal{S}$ link implementations could be carried out.

Due to lacking automated test tools and realistic usage- and user data, tests under load have been omitted. Instead, the web server's behavior was evaluated on reception of a *RST_STREAM* frame, which immediately stops every further processing.

The Server Push advancement was evaluated regarding its ability to avoid unnecessarily open connections and therefore saving resources, its ability to implement the publish/subscribe pattern and its fault tolerance in the case of an unexpected connection abort after a *PUSH_PROMISE* frame had already been sent.

The results of the first evaluation show that by using Server Push for long-running operations streams, that otherwise had to be kept open, can be closed sooner, while the stream to carry the data can remain in a reserved state. This saves resources on both ends of the link, while not prolonging the operation's total execution time. Admittedly, this finding can deviate under high system or connection load and should therefore be reevaluated when used in practice.

Examining the possible use of the Server Push feature for implementing the Publish/Subscribe pattern regrettably showed that the HTTP/2 standard does not permit sending *PUSH_PROMISE* frames over an already promised stream. As this is necessary to implement Publish/Subscribe, it can not be implemented using Server Push.

The fault tolerance evaluation of Server Push, however, shows that the SANE behaves exactly alike without Server Push by not outputting any more data on the promised stream if the connection is ended, either intentionally or unintentionally.

Beside the advancement specific evaluation, several other aspects that derive from observations that were made over the course of this thesis had to be taken into account:

According to Daniel Stenberg, in his capacity as developer of libcurl, it is capable of recognizing if new data transfers are destined for a recipient, to whom already an open connection exists. In this case, instead of establishing a new connection, merely a new HTTP/2 stream is added. The purpose of the evaluation now was to verify this proclaimed feature. Unfortunately, it was not possible to reproduce this behavior taking different versions of PHP and different environment configurations into account. From this it follows that in order to utilize connection multiplexing for the SANE, the implementation of a central component is necessary to combine outgoing requests onto one connection explicitly.

Another aspect that resulted from observations made during proving the feasibility of HTTP/2 advancements for the SANE is that requests using the libcurl multi handle, which should per definition allow parallel requests, were processed sequentially instead. The assumption that this behavior could be rooted in the Apache's internal php module could be confirmed. Yet unclear is if the Apache module is incapable of doing so, due to its multi-process but single-threaded execution of PHP code, as the libcurl multi handle should be able to issue parallel requests using only one thread, according to the official documentation. This finding constraints the SANE to be used in a server environment solely employing PHP via FastCGI.

Making the employment of FastCGI a constraint for the SANE entails having to verify the functional capability after the changes were made in in the course of the Server Push implementation. It turned out that PHP-FPM was missing a function that should be available according to the official PHP documentation. However, it was possible to reproduce its functionality. In the following could be verified that the SANE is now working as expected.

7 CONCLUSION

“

It is quite amazing how hard the subconscious works when it is made to understand that this life is not a rehearsal, there is no safety net and no assurance of any final closure. It is also quite appalling to realize how catatonic the imagination can become when we hedge our bets, opt for the safer direction at every fork in the path.

John Burdett

”

When HTTP came up in the year 1991, no one could possibly have anticipated its success story. Meanwhile, it has become the most used, in terms of request amount and most familiar protocol to even the average user of all protocols in the Internet Protocol Suite. It also builds the backbone for the lion's share of data exchange over the Internet.

24 years after its introduction a successor to the to the protocol developed by the father of the *Web* himself *Tim Berners-Lee* has been ratified that gets rid of legacy while still maintaining its high-level API. It is much more appropriate to the changed demands with special regard to a cross-linked *Internet of Things* world that is imminent.

As the cited studies point out in chapter 3 do the improvements of HTTP/2 allow faster data transmission while consuming less energy due to less packet latency, less server round trips and more straightforward processing. Moreover bears the lower power demand and extended feature set the chance of new applications. On the opposite site has to be stated that due to multiplexing HTTP/2 becomes more prone to packet loss, what however seems to be the only real downside.

7.1 SUMMARY

This section once more recapitulates the essentials from concept, its proof and the evaluation.

7.1.1 Concept

The subject of this thesis was to rate the improvements of HTTP/2 compared to HTTP 1.1 in a general way and with special regards to the proxy pattern using the example of the SANE, a crowdsourcing management software that facilitates a proxy component mainly for anonymizing the origin of crowdsourcing submissions. It is based on the previous work of Pu [Pu16], who mainly treated the header compression feature beside the implicit advantages of HTTP/2. The remaining advancements were related to the SANE incorporating his classification of link types in the partial links for Client-Proxy $\mathcal{C} - \mathcal{P}$ and Proxy-Server $\mathcal{P} - \mathcal{S}$ communication, which together constitute the entire *virtual link* between Client and Server. As the proxy acts as a server to the client and as a client to the server, different demands and ways to satisfy them had to be taken into account.

On the $\mathcal{C} - \mathcal{P}$ partial link only the employment of *Server Push*, the possibility of pushing information to the client on a previously established connection, and *Stream Reset*, the ability to reset a stream and therefore a pending operations before completion, have been found advantageous. May *Flow Control* have a use for signaling the sending instance to reduce the amount of sent data in cases of high network or system load if not already done by the TCP layer below is *Stream Prioritization* naturally not reasonable for single streams on a connection.

As the $\mathcal{P} - \mathcal{S}$ partial link, however, combines or should combine the transmissions to the same server onto one connection using multiple streams, prioritizing one over another or choking single ones is reasonable. The ability for the server to push informations to the proxy and for the proxy or the server to invalidate an operation using *Stream Reset* is naturally also advantageous on this partial link, at the very least to extend its use onto the entire virtual $\mathcal{C} - \mathcal{P} - \mathcal{S}$ link.

7.1.2 Proof of Concept

While most of these demands on the $\mathcal{C} - \mathcal{P}$ virtual link are covered by native web server functionality, for the $\mathcal{P} - \mathcal{S}$, link a HTTP/2 supporting client library has to be put to use. In the course of this thesis it turned out that libcurl, even though it is the most sophisticated HTTP/2 client currently available, does not support all newly introduced HTTP/2 advancements: *Stream Prioritization* indeed is supported by libcurl, but not yet even by the newest version of PHP 7.1, *Flow Control* is not even supported by libcurl itself, while the use of *Server Push* requires PHP 7.1 to provide the necessary callback handlers. As the SANE is developed for using PHP 5.6 and the version 7.1 breaks downward compatibility, it is not safe to assume that the SANE is capable of operating error-free on this version. Hence, none of the features found to be advantageous for the $\mathcal{P} - \mathcal{S}$ link could be implemented on the SANE. Instead, a principal proof of functionality has been conducted for the features supported by libcurl and PHP 7.1 *Server Push* and *Stream Reset* on the $\mathcal{C} - \mathcal{P}$, $\mathcal{P} - \mathcal{S}$ and on the entire virtual $\mathcal{C} - \mathcal{P} - \mathcal{S}$ link.

Nevertheless, *Server Push* was also implemented for the SANE on the $\mathcal{C} - \mathcal{P}$ link, allowing the result of both SANE management and crowdsourcing methods to be pushed to the requesting client, if certain constraints are met as follows. As it is by HTTP/2's design not possible to push responses to POST requests, a HTTP GET request with the necessary parameters moved into the HTTP header has to be used. In addition to the parameters, which keys need to have a prefixed *s-* to identify them as SANE parameters, an entry named *s-useserverpush* as with a string or integer encoded Boolean value has to be added to signalize the clients desire for the response to be pushed. Furthermore must the invoked method be marked as *pushable* in its description as long as the SANE is not instructed to push every response via override in its configuration. Finally, this Server Push functionality has been verified for both above mentioned execution paths executing methods that resemble the largest currently existing method regarding its parameter size. In case the parameters' size exceeds the size limit for a single entry, a chunking algorithm was designed which allows adding parameters with arbitrary size for future implementation.

7.1.3 Evaluation

For the general evaluation of SANE's performance using HTTP/2 the results from [Pu16] have been reused, as the involved SANE code is still identical for the default execution path. Only when the SANE is able to do connection multiplexing on the $\mathcal{P} - \mathcal{S}$ partial link, this evaluation should be renewed. The results show that, only due to header compression and the implicit improvements of HTTP/2, using the SANE is faster and probably uses up less resources.

Apart from the general performance, the improvements Server Push and Stream Reset, which were either implemented on the SANE or shown to work in principle, are subject to evaluation. Due to the relative youth of HTTP/2 no free to use load test environments, which take the features to test into account, are available at this juncture. As also the newly implemented features are not yet in practical use by a representative amount of users no realistic performance evaluation can currently be conducted regarding them.

The evaluation of Stream Reset is therefore confined to verify the latency of the reaction to the reception of a *RST_STREAM* frame based on the Wireshark protocols of previously ran tests, which were evaluated *by hand*. According to them, under zero latency, the reception of an *RST_STREAM* frame entails an immediate stop of processing, what complies with the standard. As a packet latency may delay the reception of such a frame, the HTTP/2 RFC [BPT15] encourages the implementation to treat belated *DATA* frames after reception of an *RST_STREAM* frame gracefully. Errors in processing due to the employment of stream resetting are therefore not expected.

Due to the absence of statistically significant usage data, the effectivity of Server Push has been rated based on single evaluation runs of a newly introduced crowdsourcing method for this purpose that returns the input parameters after a delay with- and without using Server Push. By manually evaluating Wireshark protocols of the communication, the timings for the entire operation and the overall timings of open streams were extracted. This shows, while having a comparable overall execution time, that using Server Push streams only had to be kept open for the time an actual data transmission took place, not for the entire runtime of an operation. This saves system resources, as streams in the reserved state almost do not use any.

Above using Server Push to save system and network resources, it was evaluated whether this technique can be used to implement the *Publish/Subscribe* pattern to allow clients to get notified when subscribed to a specific service. Due to the fact that this would require sending yet another *PUSH_PROMISE* frame over an already promised stream, which is not viable as demonstrated, HTTP/2 Server Push can obviously not be used for this purpose. The preceding evaluation of Server Push took a closer look at its behavior in case the entire connection is closed client-side after a stream has been already promised to be pushed via *PUSH_PROMISE* frame. It turned out that in all cases the script execution stopped immediately. Hence, the pushed stream's behavior on cancellation is identical to the behavior of the stream delivering the requested content on the regular way.

Beside the evaluation of implemented improvements of HTTP/2, other aspects essential to the intended use of HTTP/2 were evaluated. Due to the relatively vague description of the libcurl's multi handle function range, there was a chance that libcurl could do some kind of *automatic smart multiplexing*, which would have massively simplified multiplexing spanning over thread- or process borders. As expected, this test turned out that no automatic multiplexing takes place and consequently a central component is required which handles outgoing connections dynamically.

Another evaluation was conducted as a consequence to the libcurl multi handle acting up with parallel processing of added multi handles with the internal Apache PHP module. As it turned out, the internal PHP module was unable to execute multiple single handles added to a multi handle in parallel - unlike the FastCGI Process Manager. The reason for this is yet unclear and has not been further investigated, as the SANE should anyway not be bound to one specific web server and therefore rather be able to operate via PHP-FPM, which is also used by other web servers.

Whether this constraint actually applies is checked in the last evaluation, which verified that the execution paths touched in the course of this thesis remain functional. As it turned out, the used versions of PHP are lacking one function that - according to the official PHP documentation - should definitely be available from version 5.4 on. As a matter of fact, it was not - neither in version 5.6 nor in version 7.1 of PHP-FPM. Luckily, the missing function could easily be reproduced and included in the SANE. Hence it is guaranteed for the concerned execution path to run on any version of PHP over FastCGI.

7.2 PERSPECTIVE AND FUTURE WORK

7.2.1 In General

In order for the Internet users and content providers to profit from employing HTTP/2, many previously employed optimization techniques have to be reversed, as they either bear no more advantages or can even be detrimental, as pointed out. Then, HTTP/2 promises an overall *better* Internet experience.

However will it even improve further, as new protocols are already waiting in the wings that pursue the same goal:

The **Q**uick **U**DP **I**nternet **C**onnections protocol, another Google development, aims to compete with TCP to provide a reliable network abstraction over an unreliable channel. In difference to TCP it is UDP based and therefore connection-less, what cuts down the number of required round-trips for handshake, encryption setup and initial request to 0 (in the optimal case) prior to transmitting actual data while still offering TCP-like congestion control and recovery of lost packets. It was developed especially with multiplexing in mind as, with out-of-order delivery, a single lost packet will also stall only one stream within the QUIC connection, not the entire connection like TCP.

Also the upcoming TLS in version 1.3 promises, aside from being more secure, to further reduce the required round-trips for resuming previously established connection from 1 round-trip (1-RTT) with TLS 1.2 to 0 round-trips (0-RTT), what further reduces the influence of latency for these cases.

7.2.2 For the SANE

The findings of this thesis suggest a lot more work has to be put into the SANE to make it *HTTP/2 proof*. First of all it has to be ensured that the SANE is able to run on PHP 7.1 as only this version and probably versions above allow all of the HTTP/2 features already implemented in libcurl, for the $\mathcal{P} - \mathcal{S}$ link between proxy and server, to actually to be used. To ensure this, the SANE should be carefully evaluated and unit tests should be designed and tested against the current target version 5.6 of PHP. This allows *regression testing* against the SANE running on PHP 7.1 to verify it operates correctly.

When the SANE's operation capability on PHP 7.1 is ensured, the first step should be to implement the aforementioned *connection broker* component, which centrally manages outgoing requests and their responses to multiplex them onto one common connection to a server. In the current state of the software, the use of HTTP/2 for outgoing requests, developed by Pu, has not yet consequently been integrated, what could be done along the way. Moreover, a central connection management instance facilitates the use of Stream Reset, as it naturally only makes sense for multiplexed connections. To extend the Server Push feature, which has been implemented on the $\mathcal{C} - \mathcal{P}$ link in this thesis, to the server and thereby to the entire virtual $\mathcal{C} - \mathcal{P} - \mathcal{S}$ link, the incoming Server Pushes have to be handled and processed accordingly, what also requires PHP 7.1, as previous versions do not provide the necessary bindings to libcurl.

As soon as future versions of PHP 7 also support Stream Prioritization, like libcurl already does, it can be implemented as proposed to assign priorities to streams representing a particular SANE operation. Finally, if both libcurl and PHP also support Flow Control on application level it may be used for load controlling individual streams within a connection in addition to prioritizing them.

Naturally, all those improvements also need to be supported by the SANE's respective peers Client and Server. As Server Push has already been implemented on the SANE in the course of this thesis, one could begin with also implementing this feature for the MapBiquitous client, according to the specifications in this thesis, to transmit parameter values in the header via GET request and handle the resulting incoming push responses. The crowdsourcing servers however need, aside from the Server Push implementation, only an upgrade of its web servers to support HTTP/2.

A CODE SNIPPETS

$\mathcal{C}-\mathcal{P}$ Stream Reset test source code

Listing A.1: dl-closetest.php

```php
<?php
$run = true;
$lorem = "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. ";
ignore_user_abort(true);
header("Content-Description: File Transfer");
header("Content-Type: text/plain");
header("Content-Disposition: attachment; filename=lorem.txt");
header("Expires: 0");
header("Cache-Control: no-cache");
header('Transfer-Encoding: chunked');
ob_flush();
flush();

while($run) {
        echo "0";
        ob_flush();
        flush();
        if(connection_aborted()) {
                $run = false;
                endPacket();
                exit();
        }
        usleep(10000);
        echo $lorem . "\n";
        ob_flush();
        flush();
}

function endPacket() {
        echo "0\r\n\r\n";
        ob_flush();
        flush();
}
?>
```

$\mathcal{C}-\mathcal{P}$ Server Push test source code

Listing A.2: filestream.php

```php
<?php
header("Connection: Keep-Alive");
header("Expires: 0");
header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
header("Content-Type: text/plain");
header('Transfer-Encoding: chunked');
```

```php
ob_flush();
flush();

$run = true;
$lorem="Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
    diam nonumy eirmod tempor invidunt ut labore et dolore magna
    aliquyam erat, sed diam voluptua. At vero eos et accusam et justo
    duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata
     sanctus est Lorem ipsum dolor sit amet. ";

$i=0;
while($i < 5) {
        sleep(5);
        $i++;
        echo "(".$i.")".$lorem."\n" ;
        ob_flush();
        flush();
}
exit();
?>
```

$\mathcal{P}-\mathcal{S}$ Stream Reset test source code

Listing A.3: ps-closetest-timedabort.php

```php
<?php
$timeToWaitBeforeExecution = 500000; //0.5 seconds
$timeToWaitBeforeCancelationSingleHandle = 3000000; // 3 seconds
$timeToWaitBeforeCancelationMultiHandle = 5000000; // 5 seconds
$urlTest = "http://localhost/poc/dl-closetest2.php";
$urlIdle = "http://localhost/poc/idlestream.php";
$ch = array();
//Initialize connection using the libcurl multi interface
$mh = curl_multi_init();
//use multiplexing and wait for first connection to have settled in
    order to use same connection for future requests
curl_multi_setopt($mh, CURLMOPT_PIPELINING, CURLPIPE_MULTIPLEX);
$ch[0] = curl_init($urlTest);
$ch[1] = curl_init($urlIdle);
foreach($ch as $clienthandle) {
        curl_setopt($clienthandle, CURLOPT_HEADER, 0);
        curl_setopt($clienthandle, CURLOPT_HTTP_VERSION,
            CURL_HTTP_VERSION_2_0);
        curl_setopt($clienthandle, CURLOPT_SSL_VERIFYHOST, 0);
        curl_setopt($clienthandle, CURLOPT_SSL_VERIFYPEER, 0);
        curl_setopt($clienthandle, CURLOPT_PIPEWAIT, 1);
        curl_setopt($clienthandle, CURLOPT_RETURNTRANSFER, 1);
        curl_multi_add_handle($mh,$clienthandle);
}
usleep($timeToWaitBeforeExecution);
$isActive = null;
//clear out curl buffer
do{
```

```
27          $mrc = curl_multi_exec($mh,$isActive);
28  } while($mrc == CURLM_CALL_MULTI_PERFORM);
29
30  $startTime = time();
31  //execution
32  $ch0Removed=false;
33  while($isActive && $mrc==CURLM_OK) {
34          //wait until network is ready
35          if(curl_multi_select($mh) != -1) {
36                  do {
37                          $mrc = curl_multi_exec($mh, $isActive);
38                          $isActive=(time()-$startTime <
                                $timeToWaitBeforeCancelationMultiHandle /
                                1000000);
39                          if(time()-$startTime >=
                                $timeToWaitBeforeCancelationSingleHandle /
                                1000000 && !$ch0Removed) {
40                                  curl_multi_remove_handle($mh,$ch[0]);
41                                  $ch0Removed = true;
42                                  }
43                  } while ($mrc == CURLM_CALL_MULTI_PERFORM && isActive);
44          }
45  }
46  if($mrc != CURLM_OK) {
47  curl_close($ch[0]);
48  curl_multi_remove_handle($mh, $ch[1]);
49  curl_close($ch[1]);
50  curl_multi_close($mh);
51  if($response) print_r($response);
52  ?>
```

Listing A.4: dl-closetest2.php

```
1   <?php
2   ignore_user_abort(true);
3   $run = true;
4   $lorem = "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
        diam nonumy eirmod tempor invidunt ut labore et dolore magna
        aliquyam erat, sed diam voluptua. At vero eos et accusam et justo
        duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata
         sanctus est Lorem ipsum dolor sit amet. ";
5   header("Content-Description: File Transfer");
6   header("Content-Type: text/plain");
7   header("Content-Disposition: attachment; filename=lorem.txt");
8   header("Expires: 0");
9   header("Cache-Control: no-cache");
10  ob_flush();
11  flush();
12
13  while($run) {
14          echo $lorem . "\n";
15          ob_flush();
16          flush();
17          if(connection_aborted()) {
18                  $run = false;
```

```
                endPacket();
                exit();
        }
        sleep(1);
}

function endPacket() {
        echo "0\r\n\r\n";
        ob_flush();
        flush();
}
?>
```

Listing A.5: idlestream.php

```
<?php
header("Connection: Keep-Alive");
header("Expires: 0");
header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
header("Content-Type: text/plain");
header('Transfer-Encoding: chunked');
ob_flush();
flush();
$run = true;
$i = 0;
//output every 100 ms to avoid stream being closed
while($run) {
        usleep(100000);
        echo "(" . ++$i . ")\n";
        ob_flush();
        flush();
        if(connection_aborted()) {
                $run = false;
                endPacket();
        }
}
exit();

function endPacket() {
        echo "0\r\n\r\n";
        ob_flush();
        flush();
}
?>
```

$\mathcal{P} - \mathcal{S}$ Server Push test source code

Listing A.6: ps-serverpushtest.php

```php
<?php
$transfers = 1;
$server_push_callback = function($parent_ch, $pushed_ch, array $headers)use(&$transfers) {
        error_log("server push callback called");
        $transfers ++;
        return CURL_PUSH_OK;
};

//Initialize connection using  the libcurl multi interface
$mh = curl_multi_init();
//use multiplexing and wait for first connection to have settled in order to use same connection for future requests
curl_multi_setopt($mh, CURLMOPT_PIPELINING, CURLPIPE_MULTIPLEX);
curl_multi_setopt($mh, CURLMOPT_PUSHFUNCTION, $server_push_callback);
$chRequest = curl_init("http://localhost/poc/pushtest_insec.php");
curl_setopt($chRequest, CURLOPT_HEADER, 0);
curl_setopt($chRequest, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_2_0);
curl_setopt($chRequest, CURLOPT_SSL_VERIFYHOST, 0);
curl_setopt($chRequest, CURLOPT_SSL_VERIFYPEER, 0);
curl_setopt($chRequest, CURLOPT_PIPEWAIT, 1);
curl_setopt($chRequest, CURLOPT_RETURNTRANSFER, 1);
curl_multi_add_handle($mh, $chRequest);

$isActive = null;
do {
        $status = curl_multi_exec($mh, $isActive);
        do {
                $info = curl_multi_info_read($mh);
                if($info !== false && $info['msg'] == CURLMSG_DONE) {
                        $handle = $info['handle'];
                        if($handle !== null) {
                                $transfers --;
                                $response = curl_multi_getcontent($info['handle']);
                                curl_multi_remove_handle($mh, $handle);
                                curl_close($handle);
                        }
                }
        } while($info);
} while($transfers);
curl_multi_close($mh);
print("Server Push response body:\n" .$response);
?>
```

Listing A.7: pushtest_insecure.php

```php
<?php
header("Connection: Keep-Alive");
header("Expires: 0");
header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
```

```php
header("Link: <http://localhost/poc/filestream.php>; rel=preload; as=
    document", false);
?>
```

$\mathcal{C}-\mathcal{P}-\mathcal{S}$ Stream Reset test source code

Listing A.8: cps-reset/server-index.php

```php
<?php
header("Content-Description: File Transfer");
header("Content-Type: text/plain");
header("Connection: Keep-Alive");
header("Expires: 0");
header("Cache-Control: no-cache");
ob_flush();
flush();

$run = true;
while($run) {
        echo "0";
        ob_flush();
        flush();
        if(connection_aborted()) {
                $run=false;
                endPacket();
                exit();
        }
        usleep(100000);
}

function endPacket() {
        echo "0\r\n\r\n";
        ob_flush();
        flush();
}
?>
```

Listing A.9: cps-reset/proxy-index.php

```php
<?php
$urlTest = "http://localhost/poc/cps-reset/server-index.php";
$urlIdle = "http://localhost/poc/idlestream.php";
ignore_user_abort(true);
$ch = array();
//Initialize connection using the libcurl multi interface
$mh = curl_multi_init();
//use multiplexing and wait for first connection to have settled in
    order to use same connection for future requests
curl_multi_setopt($mh, CURLMOPT_PIPELINING, CURLPIPE_MULTIPLEX);
$ch = array();
$ch[0] = curl_init($urlTest);
$ch[1] = curl_init($urlIdle);
```

```
13 foreach($ch as $clienthandle) {
14         curl_setopt($clienthandle, CURLOPT_HEADER, 0);
15         curl_setopt($clienthandle, CURLOPT_HTTP_VERSION,
               CURL_HTTP_VERSION_2_0);
16         curl_setopt($clienthandle, CURLOPT_SSL_VERIFYHOST, 0);
17         curl_setopt($clienthandle, CURLOPT_SSL_VERIFYPEER, 0);
18         curl_setopt($clienthandle, CURLOPT_PIPEWAIT, 1);
19         curl_setopt($clienthandle, CURLOPT_RETURNTRANSFER, 1);
20         curl_multi_add_handle($mh,$clienthandle);
21 }
22 $isActive = null;
23 //clear out curl buffer
24 do{
25         $mrc = curl_multi_exec($mh,$isActive);
26 } while($mrc == CURLM_CALL_MULTI_PERFORM);
27
28 $stopTime = null;
29 //execution
30 while($isActive && $mrc==CURLM_OK) {
31         //wait until network is ready
32         if(curl_multi_select($mh) != -1) {
33                 do {
34                         $mrc = curl_multi_exec($mh, $isActive);
35                                 if ($stopTime != null) $isActive = ((
                                       time()-$stopTime) < 1); //give curl
                                        time to settle, execution stopped
                                       1 seconds after connection abort
36                                 echo "0";
37                                 ob_flush();
38                                 flush();
39                                 if (connection_aborted() && ($stopTime
                                       ==null)) {
40                                         curl_multi_remove_handle($mh,
                                               $ch[0]);
41                                         $stopTime = time();
42                                 }
43                                 usleep(100000);
44                 } while ($mrc == CURLM_CALL_MULTI_PERFORM && $isActive)
                       ;
45         }
46 }
47 sleep(1);
48 curl_close($ch[0]);
49 curl_multi_remove_handle($mh, $ch[1]);
50 curl_close($ch[1]);
51 curl_multi_close($mh);
52 ?>
```

$\mathcal{C}-\mathcal{P}-\mathcal{S}$ Server Push test source code

Listing A.10: cps-push/server-index.php

```php
<?php
$headerString="";
foreach (getallheaders() as $name => $value) {
        $headerString.= $name.":".$value."; ";
}

$headerStringHash = hash("sha256",$headerString);
$mcd = new Memcached();
$mcd->addServer("127.0.0.1", 11211);
$mcd->set($headerStringHash,$headerString);

header("Connection: Keep-Alive");
header("Expires: 0");
header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
header("Link: <http://localhost/poc/cps-push/server-pusher.php/?hid=".
    $headerStringHash.">; rel=preload; as=document", false);
?>
```

Listing A.11: cps-push/proxy-index.php

```php
<?php
$transfers = 1;
$server_push_callback = function($parent_ch, $pushed_ch, array $headers
    )use(&$transfers) {
        error_log("server push callback called");
        $transfers ++;
        return CURL_PUSH_OK;
};

//extract headers into array
$headerArray = array();
$i = 0;
foreach (getallheaders() as $key => $value) {
        $headerArray[$i++]=$key.": ".$value;
}

//Initialize connection using  the libcurl multi interface
$mh = curl_multi_init();
//use multiplexing and wait for first connection to have settled in
    order to use same connection for future requests
curl_multi_setopt($mh, CURLMOPT_PIPELINING, CURLPIPE_MULTIPLEX);
curl_multi_setopt($mh, CURLMOPT_PUSHFUNCTION, $server_push_callback);
$chRequest = curl_init("http://localhost/poc/cps-push/server-index.php"
    );
curl_setopt($chRequest, CURLOPT_HEADER, 0);
curl_setopt($chRequest, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_2_0);
curl_setopt($chRequest, CURLOPT_SSL_VERIFYHOST, 0);
curl_setopt($chRequest, CURLOPT_SSL_VERIFYPEER, 0);
curl_setopt($chRequest, CURLOPT_PIPEWAIT, 1);
curl_setopt($chRequest, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($chRequest, CURLOPT_HTTPHEADER, $headerArray);
```

```
curl_multi_add_handle($mh, $chRequest);

$isActive = null;
do {
        $status = curl_multi_exec($mh, $isActive);
        do {
                $info = curl_multi_info_read($mh);
                if($info !== false && $info['msg'] == CURLMSG_DONE) {
                        $handle = $info['handle'];
                        if($handle !== null) {
                                $transfers --;
                                $response = curl_multi_getcontent($info
                                    ['handle']);
                                curl_multi_remove_handle($mh, $handle);
                                curl_close($handle);
                        }
                }
        } while($info);
} while($transfers);
curl_multi_close($mh);

$responseHash = hash("SHA256", $response);
$mcd = new Memcached();
$mcd->addServer("127.0.0.1", 11211);
$mcd->set($responseHash, $response);

//in callback: trigger push to client
header("Connection: Keep-Alive");
header("Expires: 0");
header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
header("Link: <https://localhost/poc/cps-push/proxy-pusher.php/?rid=".
    $responseHash.">; rel=preload; as=document", false);
?>
```

Listing A.12: cps-push/proxy-pusher.php

```
<?php
$mcd = new Memcached();
$mcd->addServer("127.0.0.1", 11211);
$payload = $mcd->get($_GET["rid"]);
print($payload);
?>
```

Listing A.13: cps-push/server-pusher.php

```
<?php
$mcd = new Memcached();
$mcd->addServer("127.0.0.1", 11211);
$payload = $mcd->get($_GET["hid"]);
print($payload);
?>
```

B TEST SETUP

Hardware - Dell Inspiron 7537

CPU	Intel Core i7-4500U
RAM	8 GB DDR3L-1600
Storage Device	Samsung SSD 840 EVO 256 GB
GPU	NVidia GeForce GT 750M

Software - LAMP Stack

OS	Linux Mint 18 Sarah
Kernel	4.4.0-45 generic
Webserver	Apache/2.4.25 (Ubuntu)
Database	MySQL 14.14 Distrib 5.7.17 (x86_64)
PHP	5.6.30-1+deb.sury.org xenial+1
	Apache 2.0 handler
Webbrowser	Chromium 55.0.2883.87
	Wireshark 2.0.2

C COMMUNICATION PROTOCOLS

Headers example

Listing C.1: Request headers example

```
HyperText Transfer Protocol 2
    Stream: HEADERS, Stream ID: 1, Length 332
        [Header Length: 632]
        [Header Count: 15]
        Header: :method: POST
            Name Length: 7
            Name: :method
            Value Length: 4
            Value: POST
            Representation: Indexed Header Field
            Index: 3
        Header: :authority: localhost
            Name Length: 10
            Name: :authority
            Value Length: 9
            Value: localhost
            Representation: Literal Header Field with Incremental
               Indexing — Indexed Name
            Index: 1
        Header: :scheme: https
            Name Length: 7
            Name: :scheme
            Value Length: 5
            Value: https
            Representation: Indexed Header Field
            Index: 7
        Header: :path: /SANE/
            Name Length: 5
            Name: :path
            Value Length: 6
            Value: /SANE/
            Representation: Literal Header Field without Indexing — New
                Name
        Header: content-length: 30
            Name Length: 14
            Name: content-length
            Value Length: 2
            Value: 30
            Representation: Literal Header Field with Incremental
               Indexing — Indexed Name
            Index: 28
        Header: cache-control: max-age=0
            Name Length: 13
            Name: cache-control
            Value Length: 9
            Value: max-age=0
            Representation: Literal Header Field with Incremental
               Indexing — Indexed Name
            Index: 24
        Header: origin: https://localhost
            Name Length: 6
```

```
48              Name: origin
49              Value Length: 17
50              Value: https://localhost
51              Representation: Literal Header Field with Incremental
                    Indexing - New Name
52          Header: upgrade-insecure-requests: 1
53              Name Length: 25
54              Name: upgrade-insecure-requests
55              Value Length: 1
56              Value: 1
57              Representation: Literal Header Field with Incremental
                    Indexing - New Name
58          Header: user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit
                /537.36 (KHTML, like Gecko) Ubuntu Chromium/55.0.2883.87
                Chrome/55.0.2883.87 Safari/537.36
59              Name Length: 10
60              Name: user-agent
61              Value Length: 133
62              Value: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (
                    KHTML, like Gecko) Ubuntu Chromium/55.0.2883.87 Chrome
                    /55.0.2883.87 Safari/537.36
63              Representation: Literal Header Field with Incremental
                    Indexing - Indexed Name
64              Index: 58
65          Header: content-type: application/x-www-form-urlencoded
66              Name Length: 12
67              Name: content-type
68              Value Length: 33
69              Value: application/x-www-form-urlencoded
70              Representation: Literal Header Field with Incremental
                    Indexing - Indexed Name
71              Index: 31
72          Header: accept: text/html,application/xhtml+xml,application/xml
                ;q=0.9,image/webp,*/*;q=0.8
73              Name Length: 6
74              Name: accept
75              Value Length: 74
76              Value: text/html,application/xhtml+xml,application/xml;q
                    =0.9,image/webp,*/*;q=0.8
77              Representation: Literal Header Field with Incremental
                    Indexing - Indexed Name
78              Index: 19
79          Header: dnt: 1
80              Name Length: 3
81              Name: dnt
82              Value Length: 1
83              Value: 1
84              Representation: Literal Header Field with Incremental
                    Indexing - New Name
85          Header: referer: https://localhost/SANE/
86              Name Length: 7
87              Name: referer
88              Value Length: 23
89              Value: https://localhost/SANE/
90              Representation: Literal Header Field with Incremental
                    Indexing - Indexed Name
```

```
            Index: 51
        Header: accept-encoding: gzip, deflate, br
            Name Length: 15
            Name: accept-encoding
            Value Length: 17
            Value: gzip, deflate, br
            Representation: Literal Header Field with Incremental
                Indexing - Indexed Name
            Index: 16
        Header: accept-language: en-US,en;q=0.8,de;q=0.6
            Name Length: 15
            Name: accept-language
            Value Length: 23
            Value: en-US,en;q=0.8,de;q=0.6
            Representation: Literal Header Field with Incremental
                Indexing - Indexed Name
            Index: 17
```

Listing C.2: Response headers example

```
HyperText Transfer Protocol 2
    Stream: HEADERS, Stream ID: 1, Length 77
        [Header Length: 164]
        [Header Count: 6]
        Header table size update
            Header table size: 4096
        Header: :status: 409
            Name Length: 7
            Name: :status
            Value Length: 3
            Value: 409
            Representation: Literal Header Field with Incremental
                Indexing - Indexed Name
            Index: 8
        Header: date: Wed, 25 Jan 2017 20:29:19 GMT
            Name Length: 4
            Name: date
            Value Length: 29
            Value: Wed, 25 Jan 2017 20:29:19 GMT
            Representation: Literal Header Field with Incremental
                Indexing - Indexed Name
            Index: 33
        Header: server: Apache/2.4.25 (Ubuntu)
            Name Length: 6
            Name: server
            Value Length: 22
            Value: Apache/2.4.25 (Ubuntu)
            Representation: Literal Header Field with Incremental
                Indexing - Indexed Name
            Index: 54
        Header: content-length: 498
            Name Length: 14
            Name: content-length
            Value Length: 3
            Value: 498
```

```
33              Representation: Literal Header Field without Indexing —
                    Indexed Name
34              Index: 28
35          Header: content-type: text/plain;charset=UTF-8
36              Name Length: 12
37              Name: content-type
38              Value Length: 24
39              Value: text/plain;charset=UTF-8
40              Representation: Literal Header Field with Incremental
                    Indexing — Indexed Name
41              Index: 31
```

$\mathcal{C} - \mathcal{P}$ Stream Reset test

Listing C.3: $\mathcal{C} - \mathcal{P}$ Stream Reset test protocol

```
1       34 0.039048314    50618  443          HTTP2    389    HEADERS
2
3  Frame 34: 389 bytes on wire (3112 bits), 389 bytes captured (3112 bits)
       on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 6, Src: ::1, Dst: ::1
6  Transmission Control Protocol, Src Port: 50618 (50618), Dst Port: 443
       (443), Seq: 502, Ack: 1501, Len: 303
7  Secure Sockets Layer
8  HyperText Transfer Protocol 2
9      Stream: HEADERS, Stream ID: 1, Length 265
10         Header: :method: GET
11         Header: :scheme: https
12         Header: :path: /closeTest.php
13
14      40 0.040109307    443    50618        HTTP2    488    HEADERS, DATA
15
16 Frame 40: 488 bytes on wire (3904 bits), 488 bytes captured (3904 bits)
       on interface 0
17 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
18 Internet Protocol Version 6, Src: ::1, Dst: ::1
19 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 50618
       (50618), Seq: 1630, Ack: 843, Len: 402
20 Secure Sockets Layer
21 HyperText Transfer Protocol 2
22     Stream: HEADERS, Stream ID: 1, Length 226
23         Header: :status: 200
24     Stream: DATA, Stream ID: 1, Length 129
25
26      45 2.362650979    50618  443          HTTP2    158    HEADERS
27
28 Frame 45: 158 bytes on wire (1264 bits), 158 bytes captured (1264 bits)
       on interface 0
29 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
```

```
30 Internet Protocol Version 6, Src: ::1, Dst: ::1
31 Transmission Control Protocol, Src Port: 50618 (50618), Dst Port: 443
      (443), Seq: 938, Ack: 2347, Len: 72
32 Secure Sockets Layer
33 HyperText Transfer Protocol 2
34     Stream: HEADERS, Stream ID: 5, Length 34
35         Header: :method: GET
36         Header: :authority: localhost
37         Header: :path: /dl-closeTest.php
38
39      46 2.398878243    443    50618       HTTP2    4065   HEADERS, DATA
40
41 Frame 46: 4065 bytes on wire (32520 bits), 4065 bytes captured (32520
      bits) on interface 0
42 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
43 Internet Protocol Version 6, Src: ::1, Dst: ::1
44 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 50618
      (50618), Seq: 2347, Ack: 1010, Len: 3979
45 Secure Sockets Layer
46 HyperText Transfer Protocol 2
47     Stream: HEADERS, Stream ID: 5, Length 285
48         Header: :status: 200
49         Header: content-description: File Transfer
50         Header: content-disposition: attachment; filename=lorem.txt
51         Header: expires: 0
52         Header: cache-control: must-revalidate, post-check=0, pre-check
              =0
53         Header: pragma: public
54     Stream: DATA, Stream ID: 5, Length 3647
55
56      48 2.429836609    443    50618       HTTP2    4058   DATA
57
58 Frame 48: 4058 bytes on wire (32464 bits), 4058 bytes captured (32464
      bits) on interface 0
59 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
60 Internet Protocol Version 6, Src: ::1, Dst: ::1
61 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 50618
      (50618), Seq: 6326, Ack: 1010, Len: 3972
62 Secure Sockets Layer
63 HyperText Transfer Protocol 2
64     Stream: DATA, Stream ID: 5, Length 3934
65
66     164 4.698210043    443    50618       HTTP2    4058   DATA
67
68 Frame 164: 4058 bytes on wire (32464 bits), 4058 bytes captured (32464
      bits) on interface 0
69 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
70 Internet Protocol Version 6, Src: ::1, Dst: ::1
71 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 50618
      (50618), Seq: 267441, Ack: 1010, Len: 3972
72 Secure Sockets Layer
73 HyperText Transfer Protocol 2
74     Stream: DATA, Stream ID: 5, Length 3934
```

```
75
76      165 4.716882024     50618   443          HTTP2     128     RST_STREAM
77
78  Frame 165: 128 bytes on wire (1024 bits), 128 bytes captured (1024 bits
        ) on interface 0
79  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
80  Internet Protocol Version 6, Src: ::1, Dst: ::1
81  Transmission Control Protocol, Src Port: 50618 (50618), Dst Port: 443
        (443), Seq: 1010, Ack: 271413, Len: 42
82  Secure Sockets Layer
83  HyperText Transfer Protocol 2
84      Stream: RST_STREAM, Stream ID: 5, Length 4
85          Error: CANCEL (8)
```

$\mathcal{C} - \mathcal{P}$ Server Push test

Listing C.4: $\mathcal{C} - \mathcal{P}$ Server Push test protocol via GET request

```
1       37 0.016653157     ::1                   40320        ::1
                              443                HTTP2     410     HEADERS
2
3   Frame 37: 410 bytes on wire (3280 bits), 410 bytes captured (3280 bits)
        on interface 0
4   Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
        _00:00:00 (00:00:00:00:00:00)
5   Internet Protocol Version 6, Src: ::1, Dst: ::1
6   Transmission Control Protocol, Src Port: 40320 (40320), Dst Port: 443
        (443), Seq: 758, Ack: 242, Len: 324
7   Secure Sockets Layer
8   HyperText Transfer Protocol 2
9       Stream: HEADERS, Stream ID: 1, Length 286
10          Header: :method: GET
11          Header: :path: /pushtest.php
12
13      39 0.017931286     ::1                   443          ::1
                              40320              HTTP2     896
            PUSH_PROMISE, HEADERS, DATA
14
15  Frame 39: 896 bytes on wire (7168 bits), 896 bytes captured (7168 bits)
        on interface 0
16  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
        _00:00:00 (00:00:00:00:00:00)
17  Internet Protocol Version 6, Src: ::1, Dst: ::1
18  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 40320
        (40320), Seq: 242, Ack: 1082, Len: 810
19  Secure Sockets Layer
20  HyperText Transfer Protocol 2
21      Stream: PUSH_PROMISE, Stream ID: 1, Length 249
22          .000 0000 0000 0000 0000 0000 0000 0010 = Promised-Stream-ID: 2
23          Header: :path: /filestream.php
24          Header: :method: GET
25      Stream: HEADERS, Stream ID: 1, Length 156
```

```
26          Header: :status: 200
27          Header: cache-control: must-revalidate, post-check=0, pre-check
                =0
28          Header: link: </filestream.php>; rel=preload; as=document
29          Header: push-policy: default
30      Stream: DATA, Stream ID: 1, Length 349
31
32       41 5.018174901     ::1                   443          ::1
                            40320             HTTP2     458     HEADERS,
            DATA
33
34  Frame 41: 458 bytes on wire (3664 bits), 458 bytes captured (3664 bits)
        on interface 0
35  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
        _00:00:00 (00:00:00:00:00:00)
36  Internet Protocol Version 6, Src: ::1, Dst: ::1
37  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 40320
        (40320), Seq: 1052, Ack: 1082, Len: 372
38  Secure Sockets Layer
39  HyperText Transfer Protocol 2
40      Stream: HEADERS, Stream ID: 2, Length 25
41          Header: :status: 200
42      Stream: DATA, Stream ID: 2, Length 300
43
44       43 10.018259171    ::1                   443          ::1
                            40320             HTTP2     424     DATA
45
46  Frame 43: 424 bytes on wire (3392 bits), 424 bytes captured (3392 bits)
        on interface 0
47  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
        _00:00:00 (00:00:00:00:00:00)
48  Internet Protocol Version 6, Src: ::1, Dst: ::1
49  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 40320
        (40320), Seq: 1424, Ack: 1082, Len: 338
50  Secure Sockets Layer
51  HyperText Transfer Protocol 2
52      Stream: DATA, Stream ID: 2, Length 300
```

Listing C.5: $\mathcal{C} - \mathcal{P}$ Server Push test protocol via POST request

```
1        23 0.003171020     ::1                   50072        ::1
                            443               HTTP2     472     HEADERS
2
3   Frame 23: 472 bytes on wire (3776 bits), 472 bytes captured (3776 bits)
        on interface 0
4   Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
5   Internet Protocol Version 6, Src: ::1, Dst: ::1
6   Transmission Control Protocol, Src Port: 50072 (50072), Dst Port: 443
        (443), Seq: 720, Ack: 147, Len: 386
7   Secure Sockets Layer
8   HyperText Transfer Protocol 2
9       Stream: HEADERS, Stream ID: 1, Length 348
10          Header: :method: POST
11          Header: :path: /poc/pushtest.php
```

```
12
13      25 0.003890712     ::1                    443          ::1
                          50072             HTTP2     699     SETTINGS,
        SETTINGS, WINDOW_UPDATE, HEADERS, DATA
14
15  Frame 25: 699 bytes on wire (5592 bits), 699 bytes captured (5592 bits)
      on interface 0
16  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
17  Internet Protocol Version 6, Src: ::1, Dst: ::1
18  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 50072
      (50072), Seq: 147, Ack: 1106, Len: 613
19  Secure Sockets Layer
20  HyperText Transfer Protocol 2
21      Stream: HEADERS, Stream ID: 1, Length 158
22          Header: :status: 200
23          Header: link: <https://localhost/poc/filestream.php>; rel=
              preload; as=document
24      Stream: DATA, Stream ID: 1, Length 371
```

$\mathcal{P} - \mathcal{S}$ Stream Reset test

Listing C.6: Wireshark protocol under usage of PHP 5.6

```
1       26 0.006446702     ::1                    34174        ::1
                            443               HTTP2     6388    HEADERS
2
3   Frame 26: 6388 bytes on wire (51104 bits), 6388 bytes captured (51104
      bits) on interface 0
4   Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
5   Internet Protocol Version 6, Src: ::1, Dst: ::1
6   Transmission Control Protocol, Src Port: 34174 (34174), Dst Port: 443
      (443), Seq: 795, Ack: 1781, Len: 6302
7   Secure Sockets Layer
8   HyperText Transfer Protocol 2
9       Stream: HEADERS, Stream ID: 1, Length 6264
10          Header: :method: GET
11          Header: :path: /poc/ps-closetest-timedabort.php
12
13      33 0.508560214     127.0.0.1              51726        127.0.0.1
                        80                HTTP      220     GET /poc/dl-
          closetest2.php HTTP/1.1
14
15  Frame 33: 220 bytes on wire (1760 bits), 220 bytes captured (1760 bits)
      on interface 0
16  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
17  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
18  Transmission Control Protocol, Src Port: 51726 (51726), Dst Port: 80
      (80), Seq: 1, Ack: 1, Len: 154
19  Hypertext Transfer Protocol
20
```

```
21      38 0.508640210    127.0.0.1             51728        127.0.0.1
                    80               HTTP     217    GET /poc/
        idlestream.php HTTP/1.1
22
23  Frame 38: 217 bytes on wire (1736 bits), 217 bytes captured (1736 bits)
        on interface 0
24  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
25  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
26  Transmission Control Protocol, Src Port: 51728 (51728), Dst Port: 80
        (80), Seq: 1, Ack: 1, Len: 151
27  Hypertext Transfer Protocol
28
29      40 0.508889763    127.0.0.1             80           127.0.0.1
                    51728            HTTP     137    HTTP/1.1 101
        Switching Protocols
30
31  Frame 40: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
        on interface 0
32  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
33  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
34  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51728
        (51728), Seq: 1, Ack: 152, Len: 71
35  Hypertext Transfer Protocol
36
37      41 0.508889778    127.0.0.1             80           127.0.0.1
                    51726            HTTP     137    HTTP/1.1 101
        Switching Protocols
38
39  Frame 41: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
        on interface 0
40  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
41  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
42  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51726
        (51726), Seq: 1, Ack: 155, Len: 71
43  Hypertext Transfer Protocol
44
45      52 0.509709085    127.0.0.1             80           127.0.0.1
                    51726            HTTP2    542    SETTINGS,
        SETTINGS, WINDOW_UPDATE, HEADERS, DATA
46
47  Frame 52: 542 bytes on wire (4336 bits), 542 bytes captured (4336 bits)
        on interface 0
48  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
49  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
50  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51726
        (51726), Seq: 72, Ack: 200, Len: 476
51  HyperText Transfer Protocol 2
52      Stream: HEADERS, Stream ID: 1, Length 124
53          Header: :status: 200
54          Header: content-description: File Transfer
55          Header: content-disposition: attachment; filename=lorem.txt
56      Stream: DATA, Stream ID: 1, Length 297
```

```
57
58      56 1.509870601    127.0.0.1             80          127.0.0.1
                      51726            HTTP2    372    DATA
59
60  Frame 56: 372 bytes on wire (2976 bits), 372 bytes captured (2976 bits)
       on interface 0
61  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
62  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
63  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51726
       (51726), Seq: 548, Ack: 209, Len: 306
64  HyperText Transfer Protocol 2
65      Stream: DATA, Stream ID: 1, Length 297
66
67      57 1.509905348    127.0.0.1             80          127.0.0.1
                      51728            HTTP2    187    HEADERS, DATA
68
69  Frame 57: 187 bytes on wire (1496 bits), 187 bytes captured (1496 bits)
       on interface 0
70  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
71  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
72  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51728
       (51728), Seq: 109, Ack: 206, Len: 121
73  HyperText Transfer Protocol 2
74      Stream: HEADERS, Stream ID: 1, Length 99
75          Header: :status: 200
76      Stream: DATA, Stream ID: 1, Length 4
77
78      60 2.510027674    127.0.0.1             80          127.0.0.1
                      51726            HTTP2    372    DATA
79
80  Frame 60: 372 bytes on wire (2976 bits), 372 bytes captured (2976 bits)
       on interface 0
81  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
82  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
83  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51726
       (51726), Seq: 854, Ack: 209, Len: 306
84  HyperText Transfer Protocol 2
85      Stream: DATA, Stream ID: 1, Length 297
86
87      61 2.510027639    127.0.0.1             80          127.0.0.1
                      51728            HTTP2    79     DATA
88
89  Frame 61: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on
       interface 0
90  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
91  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
92  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51728
       (51728), Seq: 230, Ack: 206, Len: 13
93  HyperText Transfer Protocol 2
94      Stream: DATA, Stream ID: 1, Length 4
95
```

```
96      69 3.510434931    127.0.0.1             80          127.0.0.1
                    51726            HTTP2    83     GOAWAY
97
98  Frame 69: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on
        interface 0
99  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
100 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
101 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51726
        (51726), Seq: 1466, Ack: 210, Len: 17
102 HyperText Transfer Protocol 2
103     Stream: GOAWAY, Stream ID: 0, Length 8
104         .000 0000 0000 0000 0000 0000 0000 0001 = Promised-Stream-ID: 1
105         Error: NO_ERROR (0)
106
107     76 5.510905214    127.0.0.1             80          127.0.0.1
                    51728            HTTP2    83     GOAWAY
108
109 Frame 76: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on
        interface 0
110 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
111 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
112 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51728
        (51728), Seq: 282, Ack: 207, Len: 17
113 HyperText Transfer Protocol 2
114     Stream: GOAWAY, Stream ID: 0, Length 8
115         .000 0000 0000 0000 0000 0000 0000 0001 = Promised-Stream-ID: 1
116         Error: NO_ERROR (0)
```

Listing C.7: Wireshark protocol under usage of PHP 7.1

```
1       25 0.009840310     ::1                   34242       ::1
                       443              HTTP2    6388   HEADERS
2
3   Frame 25: 6388 bytes on wire (51104 bits), 6388 bytes captured (51104
        bits) on interface 0
4   Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
5   Internet Protocol Version 6, Src: ::1, Dst: ::1
6   Transmission Control Protocol, Src Port: 34242 (34242), Dst Port: 443
        (443), Seq: 795, Ack: 1781, Len: 6302
7   Secure Sockets Layer
8   HyperText Transfer Protocol 2
9       Stream: HEADERS, Stream ID: 1, Length 6264
10          Header: :method: GET
11          Header: :path: /poc/ps-closetest-timedabort.php
12
13      32 0.511728063    127.0.0.1             51794       127.0.0.1
                    80               HTTP     220    GET /poc/dl-
            closetest2.php HTTP/1.1
14
15  Frame 32: 220 bytes on wire (1760 bits), 220 bytes captured (1760 bits)
        on interface 0
```

```
16 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
17 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
18 Transmission Control Protocol, Src Port: 51794 (51794), Dst Port: 80
       (80), Seq: 1, Ack: 1, Len: 154
19 Hypertext Transfer Protocol
20
21      34 0.511945094    127.0.0.1             80          127.0.0.1
                      51794            HTTP     137    HTTP/1.1 101
           Switching Protocols
22
23 Frame 34: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
       on interface 0
24 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
25 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
26 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51794
       (51794), Seq: 1, Ack: 155, Len: 71
27 Hypertext Transfer Protocol
28
29      39 0.512056031    127.0.0.1             51794       127.0.0.1
                      80               HTTP2    106    HEADERS
30
31 Frame 39: 106 bytes on wire (848 bits), 106 bytes captured (848 bits)
       on interface 0
32 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
33 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
34 Transmission Control Protocol, Src Port: 51794 (51794), Dst Port: 80
       (80), Seq: 200, Ack: 72, Len: 40
35 HyperText Transfer Protocol 2
36      Stream: HEADERS, Stream ID: 3, Length 31
37          Header: :method: GET
38          Header: :path: /poc/idlestream.php
39
40      42 0.513120096    127.0.0.1             80          127.0.0.1
                      51794            HTTP2    505    HEADERS, DATA
41
42 Frame 42: 505 bytes on wire (4040 bits), 505 bytes captured (4040 bits)
       on interface 0
43 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
44 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
45 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51794
       (51794), Seq: 109, Ack: 249, Len: 439
46 HyperText Transfer Protocol 2
47      Stream: HEADERS, Stream ID: 1, Length 124
48          Header: content-description: File Transfer
49          Header: content-disposition: attachment; filename=lorem.txt
50      Stream: DATA, Stream ID: 1, Length 297
51
52      44 1.513249870    127.0.0.1             80          127.0.0.1
                      51794            HTTP2    372    DATA
53
54 Frame 44: 372 bytes on wire (2976 bits), 372 bytes captured (2976 bits)
       on interface 0
```

```
55 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
56 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
57 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51794
   (51794), Seq: 548, Ack: 249, Len: 306
58 HyperText Transfer Protocol 2
59     Stream: DATA, Stream ID: 1, Length 297
60
61      52 4.513657678    127.0.0.1             51794        127.0.0.1
                    80               HTTP2     79      RST_STREAM
62
63 Frame 52: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on
   interface 0
64 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
65 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
66 Transmission Control Protocol, Src Port: 51794 (51794), Dst Port: 80
   (80), Seq: 249, Ack: 1772, Len: 13
67 HyperText Transfer Protocol 2
68     Stream: RST_STREAM, Stream ID: 1, Length 4
69         Error: STREAM_CLOSED (5)
70
71      55 5.515169020    ::1                   443          ::1
                         34242             HTTP2     1663    HEADERS,
         DATA
72
73 Frame 55: 1663 bytes on wire (13304 bits), 1663 bytes captured (13304
   bits) on interface 0
74 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
75 Internet Protocol Version 6, Src: ::1, Dst: ::1
76 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 34242
   (34242), Seq: 1781, Ack: 7135, Len: 1577
77 Secure Sockets Layer
78 HyperText Transfer Protocol 2
79     Stream: HEADERS, Stream ID: 1, Length 73
80         Header: :status: 200
81     Stream: DATA, Stream ID: 1, Length 1457
82
83      58 5.843502547    127.0.0.1             80           127.0.0.1
                    51794             HTTP2     83      GOAWAY
84
85 Frame 58: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on
   interface 0
86 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
87 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
88 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51794
   (51794), Seq: 1772, Ack: 263, Len: 17
89 HyperText Transfer Protocol 2
90     Stream: GOAWAY, Stream ID: 0, Length 8
91         .000 0000 0000 0000 0000 0000 0000 0011 = Promised-Stream-ID: 3
92         Error: NO_ERROR (0)
```

$\mathcal{P} - \mathcal{S}$ Server Push test

Listing C.8: $\mathcal{P} - \mathcal{S}$ Server Push test via GET request

```
1     31 0.009273999    127.0.0.1             33668        127.0.0.1
                  80               HTTP     215    GET /poc/pushtest
        _insec.php HTTP/1.1
2
3  Frame 31: 215 bytes on wire (1720 bits), 215 bytes captured (1720 bits)
     on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
6  Transmission Control Protocol, Src Port: 33668 (33668), Dst Port: 80
    (80), Seq: 1, Ack: 1, Len: 149
7  Hypertext Transfer Protocol
8      Connection: Upgrade, HTTP2-Settings\r\n
9      Upgrade: h2c\r\n
10     [Response in frame: 33]
11
12    33 0.009538381    127.0.0.1             80           127.0.0.1
                  33668            HTTP     137    HTTP/1.1 101
        Switching Protocols
13
14 Frame 33: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
     on interface 0
15 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
16 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
17 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 33668
    (33668), Seq: 1, Ack: 150, Len: 71
18 Hypertext Transfer Protocol
19     HTTP/1.1 101 Switching Protocols\r\n
20     Upgrade: h2c\r\n
21     [Request in frame: 31]
22
23    40 0.010818401    127.0.0.1             80           127.0.0.1
                  33668            HTTP2    660    PUSH_PROMISE,
        HEADERS, DATA
24
25 Frame 40: 660 bytes on wire (5280 bits), 660 bytes captured (5280 bits)
     on interface 0
26 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
27 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
28 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 33668
    (33668), Seq: 109, Ack: 204, Len: 594
29 HyperText Transfer Protocol 2
30     Stream: PUSH_PROMISE, Stream ID: 1, Length 43
31         .000 0000 0000 0000 0000 0000 0000 0010 = Promised-Stream-ID: 2
32         Header: :path: /poc/filestream.php
33         Header: :method: GET
34     Stream: HEADERS, Stream ID: 1, Length 170
35         Header: :status: 200
```

```
36          Header: link: <http://localhost/poc/filestream.php>; rel=
                preload; as=document
37          Header: push-policy: default
38      Stream: DATA, Stream ID: 1, Length 354
39
40       42 5.017117693     127.0.0.1              80          127.0.0.1
                       33668             HTTP2     432     HEADERS, DATA
41
42  Frame 42: 432 bytes on wire (3456 bits), 432 bytes captured (3456 bits)
        on interface 0
43  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
44  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
45  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 33668
        (33668), Seq: 703, Ack: 204, Len: 366
46  HyperText Transfer Protocol 2
47      Stream: HEADERS, Stream ID: 2, Length 48
48          Header: :status: 200
49      Stream: DATA, Stream ID: 2, Length 300
```

Listing C.9: $\mathcal{P} - \mathcal{S}$ Server Push test via POST request

```
 1  31 0.005586456     127.0.0.1              52908        127.0.0.1
                  80                HTTP      293     POST /poc/pushtest_
        insec.php HTTP/1.1
 2
 3  Frame 31: 293 bytes on wire (2344 bits), 293 bytes captured (2344 bits)
        on interface 0
 4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
 5  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
 6  Transmission Control Protocol, Src Port: 52908 (52908), Dst Port: 80
        (80), Seq: 1, Ack: 1, Len: 227
 7
 8  33 0.007771016     127.0.0.1              80          127.0.0.1
                  52908             HTTP      137     HTTP/1.1 101 Switching
         Protocols
 9
10  Frame 33: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
        on interface 0
11  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
12  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
13  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 52908
        (52908), Seq: 1, Ack: 228, Len: 71
14  Hypertext Transfer Protocol
15      HTTP/1.1 101 Switching Protocols\r\n
16      Upgrade: h2c\r\n
17      Connection: Upgrade\r\n
18      [Request in frame: 31]
19
20  38 0.008161286     127.0.0.1              80          127.0.0.1
                  52908             HTTP2     645     SETTINGS, SETTINGS,
        WINDOW_UPDATE, HEADERS, DATA
21
```

```
22 Frame 38: 645 bytes on wire (5160 bits), 645 bytes captured (5160 bits)
       on interface 0
23 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
24 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
25 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 52908
       (52908), Seq: 72, Ack: 273, Len: 579
26 HyperText Transfer Protocol 2
27     Stream: SETTINGS, Stream ID: 0, Length 6
28     Stream: SETTINGS, Stream ID: 0, Length 0
29     Stream: WINDOW_UPDATE, Stream ID: 0, Length 4
30     Stream: HEADERS, Stream ID: 1, Length 154
31     Stream: DATA, Stream ID: 1, Length 370
```

$\mathcal{C}-\mathcal{P}-\mathcal{S}$ Stream Reset test

Listing C.10: $\mathcal{C}-\mathcal{P}-\mathcal{S}$ Stream Reset test protocol

```
1      26 0.002412532     ::1                    41120        ::1
                              443                HTTP2     421     HEADERS
2
3  Frame 26: 421 bytes on wire (3368 bits), 421 bytes captured (3368 bits)
       on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 6, Src: ::1, Dst: ::1
6  Transmission Control Protocol, Src Port: 41120 (41120), Dst Port: 443
       (443), Seq: 758, Ack: 242, Len: 335
7  Secure Sockets Layer
8  HyperText Transfer Protocol 2
9      Stream: HEADERS, Stream ID: 1, Length 297
10
11     31 0.003854788     127.0.0.1              58672        127.0.0.1
                      80                HTTP      229     GET /poc/cps-
           reset/server-index.php HTTP/1.1
12
13 Frame 31: 229 bytes on wire (1832 bits), 229 bytes captured (1832 bits)
       on interface 0
14 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
15 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
16 Transmission Control Protocol, Src Port: 58672 (58672), Dst Port: 80
       (80), Seq: 1, Ack: 1, Len: 163
17 Hypertext Transfer Protocol
18
19     33 0.003994874     127.0.0.1              80           127.0.0.1
                      58672             HTTP      137     HTTP/1.1 101
           Switching Protocols
20
21 Frame 33: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
       on interface 0
22 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
```

```
23 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
24 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 58672
      (58672), Seq: 1, Ack: 164, Len: 71
25 Hypertext Transfer Protocol
26
27       38 0.004122385    127.0.0.1             58672       127.0.0.1
                        80               HTTP2    106    HEADERS
28
29 Frame 38: 106 bytes on wire (848 bits), 106 bytes captured (848 bits)
      on interface 0
30 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
31 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
32 Transmission Control Protocol, Src Port: 58672 (58672), Dst Port: 80
      (80), Seq: 209, Ack: 72, Len: 40
33 HyperText Transfer Protocol 2
34       Stream: HEADERS, Stream ID: 3, Length 31
35
36       40 0.004377003    127.0.0.1             80          127.0.0.1
                        58672            HTTP2    186    HEADERS, DATA
37
38 Frame 40: 186 bytes on wire (1488 bits), 186 bytes captured (1488 bits)
       on interface 0
39 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
40 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
41 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 58672
      (58672), Seq: 109, Ack: 249, Len: 120
42 HyperText Transfer Protocol 2
43       Stream: HEADERS, Stream ID: 1, Length 101
44       Stream: DATA, Stream ID: 1, Length 1
45
46       41 0.004417134    ::1                   443         ::1
                           41120            HTTP2    201    HEADERS,
          DATA
47
48 Frame 41: 201 bytes on wire (1608 bits), 201 bytes captured (1608 bits)
       on interface 0
49 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
50 Internet Protocol Version 6, Src: ::1, Dst: ::1
51 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 41120
      (41120), Seq: 242, Ack: 1093, Len: 115
52 Secure Sockets Layer
53 HyperText Transfer Protocol 2
54       Stream: HEADERS, Stream ID: 1, Length 67
55       Stream: DATA, Stream ID: 1, Length 1
56
57       46 0.104480206    ::1                   443         ::1
                           41120            HTTP2    125    DATA
58
59 Frame 46: 125 bytes on wire (1000 bits), 125 bytes captured (1000 bits)
       on interface 0
60 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
61 Internet Protocol Version 6, Src: ::1, Dst: ::1
```

```
62  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 41120
        (41120), Seq: 357, Ack: 1093, Len: 39
63  Secure Sockets Layer
64  HyperText Transfer Protocol 2
65      Stream: DATA, Stream ID: 1, Length 1
66
67       48 0.104500095    127.0.0.1             80          127.0.0.1
                       58672            HTTP2    76     DATA
68
69  Frame 48: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on
        interface 0
70  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
71  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
72  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 58672
        (58672), Seq: 229, Ack: 258, Len: 10
73  HyperText Transfer Protocol 2
74      Stream: DATA, Stream ID: 1, Length 1
75
76       50 0.204613737    127.0.0.1             80          127.0.0.1
                       58672            HTTP2    76     DATA
77
78  Frame 50: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on
        interface 0
79  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
80  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
81  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 58672
        (58672), Seq: 239, Ack: 258, Len: 10
82  HyperText Transfer Protocol 2
83      Stream: DATA, Stream ID: 1, Length 1
84
85       52 0.204627191    ::1                   443         ::1
                          41120            HTTP2    125    DATA
86
87  Frame 52: 125 bytes on wire (1000 bits), 125 bytes captured (1000 bits)
        on interface 0
88  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
89  Internet Protocol Version 6, Src: ::1, Dst: ::1
90  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 41120
        (41120), Seq: 396, Ack: 1093, Len: 39
91  Secure Sockets Layer
92  HyperText Transfer Protocol 2
93      Stream: DATA, Stream ID: 1, Length 1
94
95       74 0.737457288    ::1                   41120       ::1
                          443              HTTP2    128    RST_STREAM
96
97  Frame 74: 128 bytes on wire (1024 bits), 128 bytes captured (1024 bits)
        on interface 0
98  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
99  Internet Protocol Version 6, Src: ::1, Dst: ::1
100 Transmission Control Protocol, Src Port: 41120 (41120), Dst Port: 443
        (443), Seq: 1093, Ack: 630, Len: 42
```

```
Secure Sockets Layer
HyperText Transfer Protocol 2
    Stream: RST_STREAM, Stream ID: 1, Length 4

     76 0.809046026    127.0.0.1             80          127.0.0.1
                    58672            HTTP2    76     DATA

Frame 76: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on
    interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 58672
    (58672), Seq: 299, Ack: 258, Len: 10
HyperText Transfer Protocol 2
    Stream: DATA, Stream ID: 1, Length 1

     78 0.906231535    127.0.0.1             80          127.0.0.1
                    58672            HTTP2    76     DATA

Frame 78: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on
    interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 58672
    (58672), Seq: 309, Ack: 258, Len: 10
HyperText Transfer Protocol 2
    Stream: DATA, Stream ID: 1, Length 1

     80 0.916144346    127.0.0.1             58672       127.0.0.1
                    80               HTTP2    79     RST_STREAM

Frame 80: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on
    interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 58672 (58672), Dst Port: 80
    (80), Seq: 258, Ack: 319, Len: 13
HyperText Transfer Protocol 2
    Stream: RST_STREAM, Stream ID: 1, Length 4

     82 1.106958223    127.0.0.1             80          127.0.0.1
                    58672            HTTP2    148    HEADERS, DATA

Frame 82: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits)
    on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 58672
    (58672), Seq: 319, Ack: 271, Len: 82
HyperText Transfer Protocol 2
    Stream: HEADERS, Stream ID: 3, Length 60
    Stream: DATA, Stream ID: 3, Length 4
```

```
141
142      84 1.207096997     127.0.0.1              80           127.0.0.1
                       58672             HTTP2     79      DATA
143
144 Frame 84: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on
       interface 0
145 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
146 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
147 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 58672
       (58672), Seq: 401, Ack: 271, Len: 13
148 HyperText Transfer Protocol 2
149      Stream: DATA, Stream ID: 3, Length 4
```

$\mathcal{C}-\mathcal{P}-\mathcal{S}$ Server Push test

Listing C.11: $\mathcal{C}-\mathcal{P}-\mathcal{S}$ Server Push test protocol

```
1       26 0.003440877     ::1                    39370        ::1
                              443               HTTP2     443     HEADERS
2
3  Frame 26: 443 bytes on wire (3544 bits), 443 bytes captured (3544 bits)
       on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 6, Src: ::1, Dst: ::1
6  Transmission Control Protocol, Src Port: 39370 (39370), Dst Port: 443
       (443), Seq: 758, Ack: 242, Len: 357
7  Secure Sockets Layer
8  HyperText Transfer Protocol 2
9       Stream: HEADERS, Stream ID: 1, Length 319
10          Header: :method: GET
11          Header: :path: /poc/cps-push/proxy-index.php
12          Header: proof: C-P-S Server Push
13
14       31 0.003920199     127.0.0.1              56922        127.0.0.1
                       80                HTTP      637     GET /poc/cps-push
           /server-index.php HTTP/1.1
15
16 Frame 31: 637 bytes on wire (5096 bits), 637 bytes captured (5096 bits)
       on interface 0
17 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
18 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
19 Transmission Control Protocol, Src Port: 56922 (56922), Dst Port: 80
       (80), Seq: 1, Ack: 1, Len: 571
20 Hypertext Transfer Protocol
21
22       33 0.004038798     127.0.0.1              80           127.0.0.1
                       56922             HTTP      137     HTTP/1.1 101
           Switching Protocols
23
```

```
Frame 33: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
    on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 56922
   (56922), Seq: 1, Ack: 572, Len: 71
Hypertext Transfer Protocol

     51 0.005106234    127.0.0.1             80          127.0.0.1
                    56922            HTTP2    626    PUSH_PROMISE,
        HEADERS, DATA

Frame 51: 626 bytes on wire (5008 bits), 626 bytes captured (5008 bits)
    on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 56922
   (56922), Seq: 109, Ack: 626, Len: 560
HyperText Transfer Protocol 2
    Stream: PUSH_PROMISE, Stream ID: 1, Length 307
        .000 0000 0000 0000 0000 0000 0000 0010 = Promised-Stream-ID: 2
        Header: :path: /poc/cps-push/server-pusher.php/?hid=3c5c5d6a
           3633b3e61cc1e2183369a75ea8a7c13bdfc7a50aaa7dfb907ce2841a
        Header: :method: GET
    Stream: HEADERS, Stream ID: 1, Length 226
        Header: link: <http://localhost/poc/cps-push/server-pusher.php
           /?hid=3c5c5d6a3633b3e61cc1e2183369a75ea8a7c13bdfc7a50aaa7
           dfb907ce2841a>; rel=preload; as=document
        Header: push-policy: default
    Stream: DATA, Stream ID: 1, Length 0

     63 0.005727866    127.0.0.1             80          127.0.0.1
                    56922            HTTP2    592    HEADERS, DATA

Frame 63: 592 bytes on wire (4736 bits), 592 bytes captured (4736 bits)
    on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 56922
   (56922), Seq: 669, Ack: 626, Len: 526
HyperText Transfer Protocol 2
    Stream: HEADERS, Stream ID: 2, Length 33
        Header: :status: 200
    Stream: DATA, Stream ID: 2, Length 475

     66 0.005782561    127.0.0.1             80          127.0.0.1
                    56922            HTTP2    83     GOAWAY

Frame 66: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on
   interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
```

```
62 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 56922
      (56922), Seq: 1195, Ack: 627, Len: 17
63 HyperText Transfer Protocol 2
64     Stream: GOAWAY, Stream ID: 0, Length 8
65         .000 0000 0000 0000 0000 0000 0000 0001 = Promised-Stream-ID: 1
66         Error: NO_ERROR (0)
67
68      79 0.006130407     ::1                   443          ::1
                              39370             HTTP2     1304    PUSH_
           PROMISE, HEADERS, DATA
69
70 Frame 79: 1304 bytes on wire (10432 bits), 1304 bytes captured (10432
      bits) on interface 0
71 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
72 Internet Protocol Version 6, Src: ::1, Dst: ::1
73 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 39370
      (39370), Seq: 242, Ack: 1115, Len: 1218
74 Secure Sockets Layer
75 HyperText Transfer Protocol 2
76     Stream: PUSH_PROMISE, Stream ID: 1, Length 310
77         .000 0000 0000 0000 0000 0000 0000 0010 = Promised-Stream-ID: 2
78         Header: :path: /poc/cps-push/proxy-pusher.php/?rid=a657dcc7a5d0
              d7fc160ebd1e42e93b3a5d207fb9e3fd3844331e5c38de889590
79         Header: :method: GET
80     Stream: HEADERS, Stream ID: 1, Length 229
81         Header: :status: 200
82         Header: link: <https://localhost/poc/cps-push/proxy-pusher.php
              /?rid=a657dcc7a5d0d7fc160ebd1e42e93b3a5d207fb9e3fd3844331e5
              c38de889590>; rel=preload; as=document
83         Header: push-policy: default
84     Stream: DATA, Stream ID: 1, Length 623
85
86      92 0.006440762     ::1                   443          ::1
                              39370             HTTP2     662     HEADERS,
           DATA
87
88 Frame 92: 662 bytes on wire (5296 bits), 662 bytes captured (5296 bits)
      on interface 0
89 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
90 Internet Protocol Version 6, Src: ::1, Dst: ::1
91 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 39370
      (39370), Seq: 1460, Ack: 1115, Len: 576
92 Secure Sockets Layer
93 HyperText Transfer Protocol 2
94     Stream: HEADERS, Stream ID: 2, Length 9
95         Header: :status: 200
96     Stream: DATA, Stream ID: 2, Length 520
```

$\mathcal{C} - \mathcal{P}$ Server Push proof SANE path

Listing C.12: $\mathcal{C} - \mathcal{P}$ Server Push on SANE execution path

```
1     26 0.003519025     ::1                    54074        ::1
                            443                HTTP2     6369    HEADERS
2
3  Frame 26: 6369 bytes on wire (50952 bits), 6369 bytes captured (50952
       bits) on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 6, Src: ::1, Dst: ::1
6  Transmission Control Protocol, Src Port: 54074 (54074), Dst Port: 443
       (443), Seq: 758, Ack: 242, Len: 6283
7  Secure Sockets Layer
8  HyperText Transfer Protocol 2
9      Stream: HEADERS, Stream ID: 1, Length 6245
10         Header: :method: GET
11         Header: :path: /SANE/index.php
12         Header: s-useserverpush: 1
13         Header: s-method: returnMethodValuesServerPushProof
14         Header: s-cs: SANE
15         Header: s-largeparameter1: EnV1ZoqwfxvTQkvN1ys6OIx0mSPakERAI1
               iCrw5QxfO3CkrjkqbvykhnwxDJT70PN74WW8YW1yIRuMCSKtThI0JHTM8
               rgVJTPHi1c7WcnwkJxDixiNrgUhO49TcpNa1hECsecgYNOvYSY3DiM4
               vhiZOKTHfJLQUAg4scBjNcEkvbILUXiIjOfJeqRLqUmwI8YetHm9T6
               aSCMIjaRWtt6
16         Header: s-mediumparam1: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
               HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
               nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
               QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
               zlPpca1A6RrcRS
17         Header: s-deviceid: DBD74A516D994F2D88664402C9BF6BC9
18         Header: s-password: OJhiRxHQgcVsK7XkvrnchHy75PvJ0OSLw3
               MjcinGReWkQQuimkMgZKmJq4c5EjF6
19         Header: s-server: fbddb61d7a6fd2304d4a1504a8516eb7
20         Header: s-signature: B4zKFymkxXlavxapHS37u7b0f17NSTbMCsbbvGwXx8
               tshaWD3UQRHhxlAHIKh1C4J4ubCFsEyOWqNzejlrKgzaB4JD37ntVJLkel4
               w9eTYxyHkJtUkjslzZoBYMqTCGOulpAcOyq60PgIX0Qxyck5fGcOK1p7
               kRYJtOk2cOVNuFE1cMVrlPuM1pHQAgSxYr1XbJSIOri58oEB16
               VzJHMXRapHp
21         Header: s-mediumparam2: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
               HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
               nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
               QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
               zlPpca1A6RrcRS
22         Padding: <MISSING>
23
24    28 0.006621587     ::1                    443          ::1
                            54074              HTTP2     920     PUSH_
           PROMISE, HEADERS, DATA
25
26 Frame 28: 920 bytes on wire (7360 bits), 920 bytes captured (7360 bits)
       on interface 0
```

```
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 443 (443), Dst Port: 54074
    (54074), Seq: 242, Ack: 7041, Len: 834
Secure Sockets Layer
HyperText Transfer Protocol 2
    Stream: PUSH_PROMISE, Stream ID: 1, Length 310
        Flags: 0x04
        .000 0000 0000 0000 0000 0000 0000 0010 = Promised-Stream-ID: 2
        Header: :path: /SANE/h2push_includer.php/?jmvid=b97c20f443df1
            aaf5eb6ee1e69ceaa8465a69b28b9545e8155f9d89f3ce3ff14
        Header: :method: GET
    Stream: HEADERS, Stream ID: 1, Length 226
        Flags: 0x04
        Header: :status: 200
        Header: link: <https://localhost/SANE/h2push_includer.php/?
            jmvid=b97c20f443df1aaf5eb6ee1e69ceaa8465a69b28b9545e8155f9d
            89f3ce3ff14>; rel=preload; as=document
        Header: push-policy: default
    Stream: DATA, Stream ID: 1, Length 242
        Flags: 0x01

     29 0.007311120    ::1                   443          ::1
                           54074             SSL      5402   [SSL
        segment of a reassembled PDU]

Frame 29: 5402 bytes on wire (43216 bits), 5402 bytes captured (43216
    bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 443 (443), Dst Port: 54074
    (54074), Seq: 1076, Ack: 7041, Len: 5316
Secure Sockets Layer
HyperText Transfer Protocol 2
    Stream: HEADERS, Stream ID: 2, Length 29
        Flags: 0x04
        Header: :status: 200

     31 0.007326724    ::1                   443          ::1
                           54074             HTTP2    2645   DATA

Frame 31: 2645 bytes on wire (21160 bits), 2645 bytes captured (21160
    bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 443 (443), Dst Port: 54074
    (54074), Seq: 6392, Ack: 7041, Len: 2559
Secure Sockets Layer
[6 Reassembled SSL segments (7663 bytes): #29(1262), #29(1300),
    #29(1300), #29(1300), #31(1300), #31(1201)]
[6 Reassembled SSL segments (7663 bytes): #29(1262), #29(1300),
    #29(1300), #29(1300), #31(1300), #31(1201)]
HyperText Transfer Protocol 2
```

```
67      Stream: DATA, Stream ID: 2, Length 7654
68          Flags: 0x01
69  HyperText Transfer Protocol 2
70      Stream: DATA, Stream ID: 2, Length 7654
71          Flags: 0x01
```

$\mathcal{C} - \mathcal{P}$ Server Push proof CS path

Listing C.13: $\mathcal{C} - \mathcal{P}$ Server Push on CS execution path

```
1       29 0.011955687     ::1                    54034        ::1
                              443                HTTP2     6369    HEADERS
2
3   Frame 29: 6369 bytes on wire (50952 bits), 6369 bytes captured (50952
        bits) on interface 0
4   Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
5   Internet Protocol Version 6, Src: ::1, Dst: ::1
6   Transmission Control Protocol, Src Port: 54034 (54034), Dst Port: 443
        (443), Seq: 527, Ack: 1810, Len: 6283
7   Secure Sockets Layer
8   HyperText Transfer Protocol 2
9       Stream: HEADERS, Stream ID: 1, Length 6245
10          Flags: 0x25
11          Header: :method: GET
12          Header: :path: /SANE/index.php
13          Header: s-useserverpush: 1
14          Header: s-method: returnMethodValuesServerPushProofCS
15          Header: s-cs: ServerPushTest
16          Header: s-largeparameter1: EnV1ZoqwfxvTQkvN1ys6OIx0mSPakERAI1
                iCrw5QxfO3CkrjkqbvykhnwxDJT70PN74WW8YW1yIRuMCSKtThI0JHTM8
                rgVJTPHi1c7WcnwkJxDixiNrgUhO49TcpNa1hECsecgYNOvYSY3DiM4
                vhiZOKTHfJLQUAg4scBjNcEkvbILUXiIjOfJeqRLqUmwI8YetHm9T6
                aSCMIjaRWtt6
17          Header: s-mediumparam1: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
                HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
                nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
                QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
                zIPpca1A6RrcRS
18          Header: s-deviceid: DBD74A516D994F2D88664402C9BF6BC9
19          Header: s-password: OJhiRxHQgcVsK7XkvrnchHy75PvJ0OSLw3
                MjcinGReWkQQuimkMgZKmJq4c5EjF6
20          Header: s-server: fbddb61d7a6fd2304d4a1504a8516eb7
21          Header: s-signature: B4zKFymkxXlavxapHS37u7b0f17NSTbMCsbbvGwXx8
                tshaWD3UQRHhxlAHIKh1C4J4ubCFsEyOWqNzejlrKgzaB4JD37ntVJLkel4
                w9eTYxyHkJtUkjslzZoBYMqTCGOulpAcOyq60PgIX0Qxyck5fGcOK1p7
                kRYJtOk2cOVNuFE1cMVrlPuM1pHQAgSxYr1XbJSIOri58oEB16
                VzJHMXRapHp
22          Header: s-mediumparam2: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
                HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
                nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
                QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
                zIPpca1A6RrcRS
```

```
23
24      31 0.015090726     ::1                       443          ::1
                           54034             HTTP2     911     PUSH_
        PROMISE, HEADERS, DATA
25
26  Frame 31: 911 bytes on wire (7288 bits), 911 bytes captured (7288 bits)
     on interface 0
27  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
28  Internet Protocol Version 6, Src: ::1, Dst: ::1
29  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 54034
    (54034), Seq: 1810, Ack: 6810, Len: 825
30  Secure Sockets Layer
31  HyperText Transfer Protocol 2
32      Stream: PUSH_PROMISE, Stream ID: 1, Length 301
33          Flags: 0x04
34          .000 0000 0000 0000 0000 0000 0000 0010 = Promised-Stream-ID: 2
35          Header: :path: /SANE/h2push_includer.php/?jmvid=2f24415a570dde
               86fce15e7799f2ea376b59bb07bf0a35530468c5de38d3c9fb
36          Header: :method: GET
37      Stream: HEADERS, Stream ID: 1, Length 226
38          Flags: 0x04
39          Header: :status: 200
40          Header: link: <https://localhost/SANE/h2push_includer.php/?
               jmvid=2f24415a570dde86fce15e7799f2ea376b59bb07bf0a35530468c
               5de38d3c9fb>; rel=preload; as=document
41          Header: push-policy: default
42      Stream: DATA, Stream ID: 1, Length 242
43          Flags: 0x01
44
45      32 0.015962450     ::1                       443          ::1
                           54034             SSL       5402    [SSL
        segment of a reassembled PDU]
46
47  Frame 32: 5402 bytes on wire (43216 bits), 5402 bytes captured (43216
    bits) on interface 0
48  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
49  Internet Protocol Version 6, Src: ::1, Dst: ::1
50  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 54034
    (54034), Seq: 2635, Ack: 6810, Len: 5316
51  Secure Sockets Layer
52  HyperText Transfer Protocol 2
53      Stream: HEADERS, Stream ID: 2, Length 29
54          Flags: 0x04
55          Header: :status: 200
56
57      34 0.015986420     ::1                       443          ::1
                           54034             HTTP2     2658    DATA
58
59  Frame 34: 2658 bytes on wire (21264 bits), 2658 bytes captured (21264
    bits) on interface 0
60  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
61  Internet Protocol Version 6, Src: ::1, Dst: ::1
```

```
62 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 54034
       (54034), Seq: 7951, Ack: 6810, Len: 2572
63 Secure Sockets Layer
64 [6 Reassembled SSL segments (7676 bytes): #32(1262), #32(1300),
       #32(1300), #32(1300), #34(1300), #34(1214)]
65 [6 Reassembled SSL segments (7676 bytes): #32(1262), #32(1300),
       #32(1300), #32(1300), #34(1300), #34(1214)]
66 HyperText Transfer Protocol 2
67     Stream: DATA, Stream ID: 2, Length 7667
68         Flags: 0x01
69 HyperText Transfer Protocol 2
70     Stream: DATA, Stream ID: 2, Length 7667
71         Flags: 0x01
```

Evaluation SANE Server Push performance

Listing C.14: SANE performance evaluation protocol without delay (regular response)

```
1    24 0.011844654    ::1                   45482       ::1
                           443              HTTP2    475    HEADERS
2
3  Frame 24: 475 bytes on wire (3800 bits), 475 bytes captured (3800 bits)
       on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 6, Src: ::1, Dst: ::1
6  Transmission Control Protocol, Src Port: 45482 (45482), Dst Port: 443
       (443), Seq: 795, Ack: 1715, Len: 389
7  Secure Sockets Layer
8  HyperText Transfer Protocol 2
9      Stream: HEADERS, Stream ID: 1, Length 351
10         Flags: 0x24
11         Header: :method: POST
12         Header: :scheme: https
13         Header: :path: /SANE/index.php
14
15   25 0.011879799    ::1                   45482       ::1
                           443              HTTP2    2976   DATA
16
17 Frame 25: 2976 bytes on wire (23808 bits), 2976 bytes captured (23808
       bits) on interface 0
18 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
19 Internet Protocol Version 6, Src: ::1, Dst: ::1
20 Transmission Control Protocol, Src Port: 45482 (45482), Dst Port: 443
       (443), Seq: 1184, Ack: 1715, Len: 2890
21 Secure Sockets Layer
22 HyperText Transfer Protocol 2
23     Stream: DATA, Stream ID: 1, Length 2852
24         Flags: 0x00
25
26   27 0.011905543    ::1                   45482       ::1
                           443              HTTP2    2976   DATA
```

```
27
28 Frame 27: 2976 bytes on wire (23808 bits), 2976 bytes captured (23808
       bits) on interface 0
29 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
30 Internet Protocol Version 6, Src: ::1, Dst: ::1
31 Transmission Control Protocol, Src Port: 45482 (45482), Dst Port: 443
       (443), Seq: 4074, Ack: 1715, Len: 2890
32 Secure Sockets Layer
33 HyperText Transfer Protocol 2
34     Stream: DATA, Stream ID: 1, Length 2852
35         Flags: 0x00
36
37      28 0.011923030    ::1                   45482        ::1
                              443               HTTP2    1902    DATA
38
39 Frame 28: 1902 bytes on wire (15216 bits), 1902 bytes captured (15216
       bits) on interface 0
40 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
41 Internet Protocol Version 6, Src: ::1, Dst: ::1
42 Transmission Control Protocol, Src Port: 45482 (45482), Dst Port: 443
       (443), Seq: 6964, Ack: 1715, Len: 1816
43 Secure Sockets Layer
44 HyperText Transfer Protocol 2
45     Stream: DATA, Stream ID: 1, Length 1778
46         Type: DATA (0)
47         Flags: 0x01
48
49      32 0.036968433    ::1                   443          ::1
                              45482             SSL      5402    [SSL
           segment of a reassembled PDU]
50
51 Frame 32: 5402 bytes on wire (43216 bits), 5402 bytes captured (43216
       bits) on interface 0
52 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
53 Internet Protocol Version 6, Src: ::1, Dst: ::1
54 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45482
       (45482), Seq: 1781, Ack: 8818, Len: 5316
55 Secure Sockets Layer
56 HyperText Transfer Protocol 2
57     Stream: HEADERS, Stream ID: 1, Length 73
58         Flags: 0x04
59         Header: :status: 200
60
61      33 0.036997181    ::1                   443          ::1
                              45482             HTTP2    2796    DATA
62
63 Frame 33: 2796 bytes on wire (22368 bits), 2796 bytes captured (22368
       bits) on interface 0
64 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
65 Internet Protocol Version 6, Src: ::1, Dst: ::1
66 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45482
       (45482), Seq: 7097, Ack: 8818, Len: 2710
```

```
67 Secure Sockets Layer
68 [7 Reassembled SSL segments (7741 bytes): #32(1218), #32(1300),
       #32(1300), #32(1300), #33(1300), #33(1300), #33(23)]
69 [7 Reassembled SSL segments (7741 bytes): #32(1218), #32(1300),
       #32(1300), #32(1300), #33(1300), #33(1300), #33(23)]
70 [7 Reassembled SSL segments (7741 bytes): #32(1218), #32(1300),
       #32(1300), #32(1300), #33(1300), #33(1300), #33(23)]
71 HyperText Transfer Protocol 2
72     Stream: DATA, Stream ID: 1, Length 7732
73         Flags: 0x01
74 HyperText Transfer Protocol 2
75     Stream: DATA, Stream ID: 1, Length 7732
76         Flags: 0x01
77 HyperText Transfer Protocol 2
78     Stream: DATA, Stream ID: 1, Length 7732
79         Flags: 0x01
```

Listing C.15: SANE performance evaluation protocol without delay (Server Push response)

```
1      25 0.007172834     ::1                    45502        ::1
                            443                HTTP2     6389    HEADERS
2
3  Frame 25: 6389 bytes on wire (51112 bits), 6389 bytes captured (51112
       bits) on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 6, Src: ::1, Dst: ::1
6  Transmission Control Protocol, Src Port: 45502 (45502), Dst Port: 443
       (443), Seq: 795, Ack: 1781, Len: 6303
7  Secure Sockets Layer
8  HyperText Transfer Protocol 2
9      Stream: HEADERS, Stream ID: 1, Length 6265
10         Flags: 0x25
11         Header: :method: GET
12         Header: :scheme: https
13         Header: :path: /SANE/index.php
14         Header: s-useserverpush: 1
15         Header: s-method: returnMethodValuesServerPushDelayedCS
16         Header: s-cs: ServerPushTest
17         Header: s-largeparameter1: EnV1ZoqwfxvTQkvN1ys6OIx0mSPakERAI1
               iCrw5QxfO3CkrjkqbvykhnwxDJT70PN74WW8YW1yIRuMCSKtThI0JHTM8
               rgVJTPHi1c7WcnwkJxDixiNrgUhO49TcpNa1hECsecgYNOvYSY3DiM4
               vhiZOKTHfJLQUAg4scBjNcEkvbILUXiIjOfJeqRLqUmwI8YetHm9T6
               aSCMIjaRWtt6
18         Header: s-mediumparam1: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
               HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
               nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
               QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
               zIPpca1A6RrcRS
19         Header: s-deviceid: DBD74A516D994F2D88664402C9BF6BC9
20         Header: s-password: OJhiRxHQgcVsK7XkvrnchHy75PvJ0OSLw3
               MjcinGReWkQQuimkMgZKmJq4c5EjF6
21         Header: s-server: fbddb61d7a6fd2304d4a1504a8516eb7
22         Header: s-signature: B4zKFymkxXlavxapHS37u7b0f17NSTbMCsbbvGwXx8
               tshaWD3UQRHhxlAHIKh1C4J4ubCFsEyOWqNzejIrKgzaB4JD37ntVJLkel4
```

```
            w9eTYxyHkJtUkjsIzZoBYMqTCGOulpAcOyq60PgIX0Qxyck5fGcOK1p7
            kRYJtOk2cOVNuFE1cMVrlPuM1pHQAgSxYr1XbJSIOri58oEB16
            VzJHMXRapHp
23          Header: s-mediumparam2: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
            HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
            nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
            QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
            zlPpca1A6RrcRS
24          Header: s-delay: 0
25
26       28 0.025196872     ::1                       443          ::1
                            45502             HTTP2     918     PUSH_
            PROMISE, HEADERS, DATA
27
28  Frame 28: 918 bytes on wire (7344 bits), 918 bytes captured (7344 bits)
        on interface 0
29  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
30  Internet Protocol Version 6, Src: ::1, Dst: ::1
31  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45502
       (45502), Seq: 1781, Ack: 7136, Len: 832
32  Secure Sockets Layer
33  HyperText Transfer Protocol 2
34      Stream: PUSH_PROMISE, Stream ID: 1, Length 309
35          Flags: 0x04
36          .000 0000 0000 0000 0000 0000 0000 0010 = Promised-Stream-ID: 2
37          Header: :scheme: https
38          Header: :path: /SANE/h2push_includer.php/?jmvid=c3c2689ea0def
               842e8064955ea0f1348b27046cce0b720fae8bb05e3b8168a6f
39          Header: :method: GET
40      Stream: HEADERS, Stream ID: 1, Length 225
41          Flags: 0x04
42          Header: :status: 200
43          Header: link: <https://localhost/SANE/h2push_includer.php/?
               jmvid=c3c2689ea0def842e8064955ea0f1348b27046cce0b720fae8bb
               05e3b8168a6f>; rel=preload; as=document
44      Stream: DATA, Stream ID: 1, Length 242
45          Flags: 0x01
46
47       29 0.026449746     ::1                       443          ::1
                            45502             SSL       5402    [SSL
            segment of a reassembled PDU]
48
49  Frame 29: 5402 bytes on wire (43216 bits), 5402 bytes captured (43216
       bits) on interface 0
50  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
51  Internet Protocol Version 6, Src: ::1, Dst: ::1
52  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45502
       (45502), Seq: 2613, Ack: 7136, Len: 5316
53  Secure Sockets Layer
54  HyperText Transfer Protocol 2
55      Stream: HEADERS, Stream ID: 2, Length 29
56          Flags: 0x04
57          Header: :status: 200
58
```

```
59      31 0.026483089     ::1                   443          ::1
                           45502                 HTTP2     2691    DATA
60
61  Frame 31: 2691 bytes on wire (21528 bits), 2691 bytes captured (21528
        bits) on interface 0
62  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
63  Internet Protocol Version 6, Src: ::1, Dst: ::1
64  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45502
        (45502), Seq: 7929, Ack: 7136, Len: 2605
65  Secure Sockets Layer
66  [6 Reassembled SSL segments (7709 bytes): #29(1262), #29(1300),
        #29(1300), #29(1300), #31(1300), #31(1247)]
67  [6 Reassembled SSL segments (7709 bytes): #29(1262), #29(1300),
        #29(1300), #29(1300), #31(1300), #31(1247)]
68  HyperText Transfer Protocol 2
69      Stream: DATA, Stream ID: 2, Length 7700
70          Flags: 0x01
71  HyperText Transfer Protocol 2
72      Stream: DATA, Stream ID: 2, Length 7700
73          Flags: 0x01
```

Listing C.16: SANE performance evaluation protocol without 1 second delay (regular response)

```
1       25 0.017498282     ::1                   45496        ::1
                           443                   HTTP2     475     HEADERS
2
3   Frame 25: 475 bytes on wire (3800 bits), 475 bytes captured (3800 bits)
        on interface 0
4   Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
5   Internet Protocol Version 6, Src: ::1, Dst: ::1
6   Transmission Control Protocol, Src Port: 45496 (45496), Dst Port: 443
        (443), Seq: 795, Ack: 1781, Len: 389
7   Secure Sockets Layer
8   HyperText Transfer Protocol 2
9       Stream: HEADERS, Stream ID: 1, Length 351
10          Flags: 0x24
11          Header: :method: POST
12          Header: :scheme: https
13          Header: :path: /SANE/index.php
14
15      26 0.017539757     ::1                   45496        ::1
                           443                   HTTP2     2976    DATA
16
17  Frame 26: 2976 bytes on wire (23808 bits), 2976 bytes captured (23808
        bits) on interface 0
18  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
19  Internet Protocol Version 6, Src: ::1, Dst: ::1
20  Transmission Control Protocol, Src Port: 45496 (45496), Dst Port: 443
        (443), Seq: 1184, Ack: 1781, Len: 2890
21  Secure Sockets Layer
22  HyperText Transfer Protocol 2
23      Stream: DATA, Stream ID: 1, Length 2852
```

```
24          Flags: 0x00
25
26       28 0.017568787     ::1                   45496        ::1
                              443                HTTP2     2976    DATA
27
28  Frame 28: 2976 bytes on wire (23808 bits), 2976 bytes captured (23808
       bits) on interface 0
29  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
30  Internet Protocol Version 6, Src: ::1, Dst: ::1
31  Transmission Control Protocol, Src Port: 45496 (45496), Dst Port: 443
       (443), Seq: 4074, Ack: 1781, Len: 2890
32  Secure Sockets Layer
33  HyperText Transfer Protocol 2
34       Stream: DATA, Stream ID: 1, Length 2852
35          Flags: 0x00
36
37       29 0.017591775     ::1                   45496        ::1
                              443                HTTP2     1902    DATA
38
39  Frame 29: 1902 bytes on wire (15216 bits), 1902 bytes captured (15216
       bits) on interface 0
40  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
41  Internet Protocol Version 6, Src: ::1, Dst: ::1
42  Transmission Control Protocol, Src Port: 45496 (45496), Dst Port: 443
       (443), Seq: 6964, Ack: 1781, Len: 1816
43  Secure Sockets Layer
44  HyperText Transfer Protocol 2
45       Stream: DATA, Stream ID: 1, Length 1778
46          Flags: 0x01
47
48       33 1.038771856     ::1                   443          ::1
                              45496              SSL       5402    [SSL
           segment of a reassembled PDU]
49
50  Frame 33: 5402 bytes on wire (43216 bits), 5402 bytes captured (43216
       bits) on interface 0
51  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
52  Internet Protocol Version 6, Src: ::1, Dst: ::1
53  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45496
       (45496), Seq: 1781, Ack: 8818, Len: 5316
54  Secure Sockets Layer
55  HyperText Transfer Protocol 2
56       Stream: HEADERS, Stream ID: 1, Length 73
57          Flags: 0x04
58          Header: :status: 200
59
60       34 1.038798693     ::1                   443          ::1
                              45496              HTTP2     2796    DATA
61
62  Frame 34: 2796 bytes on wire (22368 bits), 2796 bytes captured (22368
       bits) on interface 0
63  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
```

```
64 Internet Protocol Version 6, Src: ::1, Dst: ::1
65 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45496
       (45496), Seq: 7097, Ack: 8818, Len: 2710
66 Secure Sockets Layer
67 [7 Reassembled SSL segments (7741 bytes): #33(1218), #33(1300),
       #33(1300), #33(1300), #34(1300), #34(1300), #34(23)]
68 [7 Reassembled SSL segments (7741 bytes): #33(1218), #33(1300),
       #33(1300), #33(1300), #34(1300), #34(1300), #34(23)]
69 [7 Reassembled SSL segments (7741 bytes): #33(1218), #33(1300),
       #33(1300), #33(1300), #34(1300), #34(1300), #34(23)]
70 HyperText Transfer Protocol 2
71     Stream: DATA, Stream ID: 1, Length 7732
72         Flags: 0x01
73 HyperText Transfer Protocol 2
74     Stream: DATA, Stream ID: 1, Length 7732
75         Flags: 0x01
76 HyperText Transfer Protocol 2
77     Stream: DATA, Stream ID: 1, Length 7732
78         Flags: 0x01
```

Listing C.17: SANE performance evaluation protocol with 1 second delay (Server Push response)

```
1      29 0.019889927     ::1                   45512         ::1
                            443                 HTTP2     6389    HEADERS
2
3  Frame 29: 6389 bytes on wire (51112 bits), 6389 bytes captured (51112
       bits) on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 6, Src: ::1, Dst: ::1
6  Transmission Control Protocol, Src Port: 45512 (45512), Dst Port: 443
       (443), Seq: 833, Ack: 1810, Len: 6303
7  Secure Sockets Layer
8  HyperText Transfer Protocol 2
9      Stream: HEADERS, Stream ID: 1, Length 6265
10         Flags: 0x25
11         Header: :method: GET
12         Header: :scheme: https
13         Header: :path: /SANE/index.php
14         Header: s-useserverpush: 1
15         Header: s-method: returnMethodValuesServerPushDelayedCS
16         Header: s-cs: ServerPushTest
17         Header: s-largeparameter1: EnV1ZoqwfxvTQkvN1ys6OIx0mSPakERAI1
               iCrw5QxfO3CkrjkqbvykhnwxDJT70PN74WW8YW1yIRuMCSKtThI0JHTM8
               rgVJTPHi1c7WcnwkJxDixiNrgUhO49TcpNa1hECsecgYNOvYSY3DiM4
               vhiZOKTHfJLQUAg4scBjNcEkvbILUXiIjOfJeqRLqUmwI8YetHm9T6
               aSCMIjaRWtt6
18         Header: s-mediumparam1: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
               HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
               nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
               QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
               zlPpca1A6RrcRS
19         Header: s-deviceid: DBD74A516D994F2D88664402C9BF6BC9
20         Header: s-password: OJhiRxHQgcVsK7XkvrnchHy75PvJ0OSLw3
               MjcinGReWkQQuimkMgZKmJq4c5EjF6
```

```
21          Header: s-server: fbddb61d7a6fd2304d4a1504a8516eb7
22          Header: s-signature: B4zKFymkxXlavxapHS37u7b0f17NSTbMCsbbvGwXx8
                tshaWD3UQRHhxlAHIKh1C4J4ubCFsEyOWqNzejlrKgzaB4JD37ntVJLkel4
                w9eTYxyHkJtUkjslzZoBYMqTCGOulpAcOyq60PgIX0Qxyck5fGcOK1p7
                kRYJtOk2cOVNuFE1cMVrlPuM1pHQAgSxYr1XbJSIOri58oEB16
                VzJHMXRapHp
23          Header: s-mediumparam2: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
                HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
                nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
                QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
                zlPpca1A6RrcRS
24          Header: s-delay: 1
25
26       31 0.045623521     ::1                       443          ::1
                              45512             HTTP2     918     PUSH_
            PROMISE, HEADERS, DATA
27
28  Frame 31: 918 bytes on wire (7344 bits), 918 bytes captured (7344 bits)
        on interface 0
29  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
30  Internet Protocol Version 6, Src: ::1, Dst: ::1
31  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45512
       (45512), Seq: 1810, Ack: 7136, Len: 832
32  Secure Sockets Layer
33  HyperText Transfer Protocol 2
34      Stream: PUSH_PROMISE, Stream ID: 1, Length 309
35          Flags: 0x04
36          .000 0000 0000 0000 0000 0000 0000 0010 = Promised-Stream-ID: 2
37          Header: :scheme: https
38          Header: :path: /SANE/h2push_includer.php/?jmvid=6237be49b06e
                6165200207478f80038738422f0f3e824da924d3f22e93bf2b01
39          Header: :method: GET
40      Stream: HEADERS, Stream ID: 1, Length 225
41          Flags: 0x04
42          Header: :status: 200
43          Header: link: <https://localhost/SANE/h2push_includer.php/?
                jmvid=6237be49b06e6165200207478f80038738422f0f3e824da924d3f
                22e93bf2b01>; rel=preload; as=document
44          Header: push-policy: default
45      Stream: DATA, Stream ID: 1, Length 242
46          Flags: 0x01
47
48       33 1.046895102     ::1                       443          ::1
                              45512             SSL       5402    [SSL
            segment of a reassembled PDU]
49
50  Frame 33: 5402 bytes on wire (43216 bits), 5402 bytes captured (43216
       bits) on interface 0
51  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
52  Internet Protocol Version 6, Src: ::1, Dst: ::1
53  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45512
       (45512), Seq: 2642, Ack: 7136, Len: 5316
54  Secure Sockets Layer
55  HyperText Transfer Protocol 2
```

```
56      Stream: HEADERS, Stream ID: 2, Length 29
57          Flags: 0x04
58          Header: :status: 200
59
60       35 1.046927065     ::1                   443          ::1
                              45512             HTTP2     2691    DATA
61
62  Frame 35: 2691 bytes on wire (21528 bits), 2691 bytes captured (21528
        bits) on interface 0
63  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
64  Internet Protocol Version 6, Src: ::1, Dst: ::1
65  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 45512
        (45512), Seq: 7958, Ack: 7136, Len: 2605
66  Secure Sockets Layer
67  [6 Reassembled SSL segments (7709 bytes): #33(1262), #33(1300),
        #33(1300), #33(1300), #35(1300), #35(1247)]
68  [6 Reassembled SSL segments (7709 bytes): #33(1262), #33(1300),
        #33(1300), #33(1300), #35(1300), #35(1247)]
69  HyperText Transfer Protocol 2
70      Stream: DATA, Stream ID: 2, Length 7700
71          Flags: 0x01
72  HyperText Transfer Protocol 2
73      Stream: DATA, Stream ID: 2, Length 7700
74          Flags: 0x01
```

SANE Server Push Publish/Subscribe evaluation

Listing C.18: SANE Server Push Publish/Subscribe evaluation

```
1        28 0.024375459     ::1                   41224        ::1
                              443               HTTP2     6405    HEADERS
2
3   Frame 28: 6405 bytes on wire (51240 bits), 6405 bytes captured (51240
        bits) on interface 0
4   Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
5   Internet Protocol Version 6, Src: ::1, Dst: ::1
6   Transmission Control Protocol, Src Port: 41224 (41224), Dst Port: 443
        (443), Seq: 833, Ack: 1810, Len: 6319
7   Secure Sockets Layer
8   HyperText Transfer Protocol 2
9       Stream: HEADERS, Stream ID: 1, Length 6281
10          Length: 6281
11          Type: HEADERS (1)
12          Flags: 0x25
13              .... ...1 = End Stream: True
14              .... .1.. = End Headers: True
15              .... 0... = Padded: False
16              ..1. .... = Priority: True
17              00.0 ..0. = Unused: 0x00
18          0... .... .... .... .... .... .... .... = Reserved: 0x00000000
19          .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
```

```
20          [Pad Length: 0]
21          1... .... .... .... .... .... .... .... = Exclusive: True
22          .000 0000 0000 0000 0000 0000 0000 0000 = Stream Dependency: 0
23          Weight: 255
24          [Weight real: 256]
25          Header Block Fragment: 824186a0e41d139d09870084b958d33f8c63743a
                70306aa4...
26          [Header Length: 8081]
27          [Header Count: 23]
28          Header: :method: GET
29          Header: :authority: localhost
30          Header: :scheme: https
31          Header: :path: /SANE/index.php
32          Header: pragma: no-cache
33          Header: cache-control: no-cache
34          Header: upgrade-insecure-requests: 1
35          Header: user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit
                /537.36 (KHTML, like Gecko) Ubuntu Chromium/55.0.2883.87
                Chrome/55.0.2883.87 Safari/537.36
36          Header: accept: text/html,application/xhtml+xml,application/xml
                ;q=0.9,image/webp,*/*;q=0.8
37          Header: dnt: 1
38          Header: accept-encoding: gzip, deflate, sdch, br
39          Header: accept-language: en-US,en;q=0.8,de;q=0.6
40          Header: s-useserverpush: 1
41          Header: s-method: returnMethodValuesServerPushDelayedNewPush
42          Header: s-cs: ServerPushTest
43          Header: s-largeparameter1: EnV1ZoqwfxvTQkvN1ys6OIx0mSPakERAI1
                iCrw5QxfO3CkrjkqbvykhnwxDJT70PN74WW8YW1yIRuMCSKtThI0JHTM8
                rgVJTPHi1c7WcnwkJxDixiNrgUhO49TcpNa1hECsecgYNOvYSY3DiM4
                vhiZOKTHfJLQUAg4scBjNcEkvbILUXiIjOfJeqRLqUmwI8YetHm9T6
                aSCMIjaRWtt6
44          Header: s-mediumparam1: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
                HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
                nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
                QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
                zlPpca1A6RrcRS
45          Header: s-deviceid: DBD74A516D994F2D88664402C9BF6BC9
46          Header: s-password: OJhiRxHQgcVsK7XkvrnchHy75PvJ0OSLw3
                MjcinGReWkQQuimkMgZKmJq4c5EjF6
47          Header: s-server: fbddb61d7a6fd2304d4a1504a8516eb7
48          Header: s-signature: B4zKFymkxXlavxapHS37u7b0f17NSTbMCsbbvGwXx8
                tshaWD3UQRHhxlAHIKh1C4J4ubCFsEyOWqNzejlrKgzaB4JD37ntVJLkel4
                w9eTYxyHkJtUkjslzZoBYMqTCGOulpAcOyq60PgIX0Qxyck5fGcOK1p7
                kRYJtOk2cOVNuFE1cMVrlPuM1pHQAgSxYr1XbJSIOri58oEB16
                VzJHMXRapHp
49          Header: s-mediumparam2: 1UqbixOFoHRfxff3guhCqGABramsrZEYM35
                HWLmpb50rPYqfmbz9kPz7Nei6QA2jGFHWSHz7oO7i1oOCMFkNxb3m83
                nKyMfsg7N983fDXiMrSPyifo80oBOLOtykXbnK00vFwSgIsr9HE57pHv1
                QchGwGtw949KZvV1epK9TVYU5AqeXrAY59kbjv5sQIH6Nrv1AUQgG4
                zlPpca1A6RrcRS
50          Header: s-delay: 5
51          Padding: <MISSING>
52
```

```
53      55 0.028035135     ::1                   443           ::1
                          41224              HTTP2     912     PUSH_
        PROMISE, HEADERS, DATA
54
55  Frame 55: 912 bytes on wire (7296 bits), 912 bytes captured (7296 bits)
        on interface 0
56  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
57  Internet Protocol Version 6, Src: ::1, Dst: ::1
58  Transmission Control Protocol, Src Port: 443 (443), Dst Port: 41224
        (41224), Seq: 1810, Ack: 7152, Len: 826
59  Secure Sockets Layer
60  HyperText Transfer Protocol 2
61      Stream: PUSH_PROMISE, Stream ID: 1, Length 308
62          Length: 308
63          Type: PUSH_PROMISE (5)
64          Flags: 0x04
65              .... .1.. = End Headers: True
66              .... 0... = Padded: False
67              0000 ..00 = Unused: 0x00
68          0... .... .... .... .... .... .... .... = Reserved: 0x00000000
69          .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
70          [Pad Length: 0]
71          0... .... .... .... .... .... .... .... = Reserved: 0x00000000
72          .000 0000 0000 0000 0000 0000 0000 0010 = Promised-Stream-ID: 2
73          Header [truncated]: ?\357\277\275\037\357\277\275A
                \357\277\275\357\277\275\357\277\275\035\023\357\277\275\t
                \004\357\277\275ct:p
                18\357\277\275\357\277\275\023\357\277\275j\357\277\275E
                \357\277\275\026\357\277\275\357\277\275\357\277\275\35
74          [Header Length: 556]
75          [Header Count: 11]
76          Header table size update
77          Header: :scheme: https
78          Header: :authority: localhost
79          Header: :path: /SANE/h2push_includer.php/?jmvid=5f037e8af7293
                cbbd4dbb2d1387302c7533cd06644213abe1c4e62ed22119cb8
80          Header: :method: GET
81          Header: cache-control: no-cache
82          Header: user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit
                /537.36 (KHTML, like Gecko) Ubuntu Chromium/55.0.2883.87
                Chrome/55.0.2883.87 Safari/537.36
83          Header: accept: text/html,application/xhtml+xml,application/xml
                ;q=0.9,image/webp,*/*;q=0.8
84          Header: accept-encoding: gzip, deflate, sdch, br
85          Header: accept-language: en-US,en;q=0.8,de;q=0.6
86          Header: host: localhost
87          Padding: <MISSING>
88      Stream: HEADERS, Stream ID: 1, Length 220
89          Length: 220
90          Type: HEADERS (1)
91          Flags: 0x04
92              .... ...0 = End Stream: False
93              .... .1.. = End Headers: True
94              .... 0... = Padded: False
95              ..0. .... = Priority: False
```

```
            00.0 ..0. = Unused: 0x00
        0... .... .... .... .... .... .... .... = Reserved: 0x00000000
        .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
        [Pad Length: 0]
        Header Block Fragment: 886196c361be940baa681d8a08017540b5700fdc
           0054c5a3...
        [Header Length: 397]
        [Header Count: 8]
        Header: :status: 200
        Header: date: Fri, 17 Mar 2017 14:09:01 GMT
        Header: server: Apache/2.4.25 (Ubuntu)
        Header: expires: 0
        Header: cache-control: must-revalidate, post-check=0, pre-check
           =0
        Header: link: <https://localhost/SANE/h2push_includer.php/?
           jmvid=5f037e8af7293cbbd4dbb2d1387302c7533cd06644213abe1c4e
           62ed22119cb8>; rel=preload; as=document
        Header: content-type: text/html;charset=UTF-8
        Header: push-policy: default
        Padding: <MISSING>
    Stream: DATA, Stream ID: 1, Length 242
        Length: 242
        Type: DATA (0)
        Flags: 0x01
            .... ...1 = End Stream: True
            .... 0... = Padded: False
            0000 .00. = Unused: 0x00
        0... .... .... .... .... .... .... .... = Reserved: 0x00000000
        .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
        [Pad Length: 0]
        Data: 3c68746d6c3e3c686561643e3c2f686561643e3c626f6479...
        Padding: <MISSING>

     79 5.029188589    ::1                   443          ::1
                          41224           SSL      5402   [SSL
        segment of a reassembled PDU]

Frame 79: 5402 bytes on wire (43216 bits), 5402 bytes captured (43216
   bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 443 (443), Dst Port: 41224
   (41224), Seq: 2636, Ack: 7152, Len: 5316
Secure Sockets Layer
HyperText Transfer Protocol 2
    Stream: HEADERS, Stream ID: 2, Length 75
        Length: 75
        Type: HEADERS (1)
        Flags: 0x04
            .... ...0 = End Stream: False
            .... .1.. = End Headers: True
            .... 0... = Padded: False
            ..0. .... = Priority: False
            00.0 ..0. = Unused: 0x00
        0... .... .... .... .... .... .... .... = Reserved: 0x00000000
```

```
143         .000 0000 0000 0000 0000 0000 0000 0010 = Stream Identifier: 2
144         [Pad Length: 0]
145         Header Block Fragment: 88c4c3c2c16db0fff93a535a2e30c50720e89ce
               84b159c8c...
146         [Header Length: 294]
147         [Header Count: 7]
148         Header: :status: 200
149         Header: date: Fri, 17 Mar 2017 14:09:01 GMT
150         Header: server: Apache/2.4.25 (Ubuntu)
151         Header: expires: 0
152         Header: cache-control: must-revalidate, post-check=0, pre-check
               =0
153         Header: link: <https://localhost/poc/filestream.php>; rel=
               preload; as=document
154         Header: content-type: text/html; charset=UTF-8
155         Padding: <MISSING>
156
157      81 5.029262892     ::1                   443          ::1
                              41224             HTTP2     2742   DATA
158
159 Frame 81: 2742 bytes on wire (21936 bits), 2742 bytes captured (21936
       bits) on interface 0
160 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
161 Internet Protocol Version 6, Src: ::1, Dst: ::1
162 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 41224
       (41224), Seq: 7952, Ack: 7152, Len: 2656
163 Secure Sockets Layer
164 [6 Reassembled SSL segments (7714 bytes): #79(1216), #79(1300),
       #79(1300), #79(1300), #81(1300), #81(1298)]
165 [6 Reassembled SSL segments (7714 bytes): #79(1216), #79(1300),
       #79(1300), #79(1300), #81(1300), #81(1298)]
166 HyperText Transfer Protocol 2
167     Stream: DATA, Stream ID: 2, Length 7705
168         Length: 7705
169         Type: DATA (0)
170         Flags: 0x01
171             .... ...1 = End Stream: True
172             .... 0... = Padded: False
173             0000 .00. = Unused: 0x00
174         0... .... .... .... .... .... .... .... = Reserved: 0x00000000
175         .000 0000 0000 0000 0000 0000 0000 0010 = Stream Identifier: 2
176         [Pad Length: 0]
177         Data: 617272617928313029207b0a20205b2264656c6179225d3d...
178         Padding: <MISSING>
179 HyperText Transfer Protocol 2
180     Stream: DATA, Stream ID: 2, Length 7705
181         Length: 7705
182         Type: DATA (0)
183         Flags: 0x01
184             .... ...1 = End Stream: True
185             .... 0... = Padded: False
186             0000 .00. = Unused: 0x00
187         0... .... .... .... .... .... .... .... = Reserved: 0x00000000
188         .000 0000 0000 0000 0000 0000 0000 0010 = Stream Identifier: 2
189         [Pad Length: 0]
```

```
190         Data: 617272617928313029207b0a20205b2264656c6179225d3d...
191         Padding: <MISSING>
```

SANE Server Push fault tolerance evaluation

Listing C.19: SANE Server Push fault tolerance evaluation with graceful termination of client

```
1     25 0.014244468    ::1                   51574       ::1
                          443              HTTP2    6381   HEADERS
2
3  Frame 25: 6381 bytes on wire (51048 bits), 6381 bytes captured (51048
       bits) on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 6, Src: ::1, Dst: ::1
6  Transmission Control Protocol, Src Port: 51574 (51574), Dst Port: 443
       (443), Seq: 489, Ack: 1781, Len: 6295
7  Secure Sockets Layer
8  HyperText Transfer Protocol 2
9
10    27 0.014326382    ::1                   51574       ::1
                          443              HTTP2    124    SETTINGS
11
12 Frame 27: 124 bytes on wire (992 bits), 124 bytes captured (992 bits)
       on interface 0
13 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
14 Internet Protocol Version 6, Src: ::1, Dst: ::1
15 Transmission Control Protocol, Src Port: 51574 (51574), Dst Port: 443
       (443), Seq: 6784, Ack: 1781, Len: 38
16 Secure Sockets Layer
17 HyperText Transfer Protocol 2
18
19    28 0.016908533    ::1                   443         ::1
                          51574            HTTP2    911    PUSH_
           PROMISE, HEADERS, DATA
20
21 Frame 28: 911 bytes on wire (7288 bits), 911 bytes captured (7288 bits)
       on interface 0
22 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
       00:00:00_00:00:00 (00:00:00:00:00:00)
23 Internet Protocol Version 6, Src: ::1, Dst: ::1
24 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 51574
       (51574), Seq: 1781, Ack: 6822, Len: 825
25 Secure Sockets Layer
26 HyperText Transfer Protocol 2
27
28    30 3.647867670    ::1                   51574       ::1
                          443              HTTP2    128    RST_STREAM
29
30 Frame 30: 128 bytes on wire (1024 bits), 128 bytes captured (1024 bits)
       on interface 0
```

```
31 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
      00:00:00_00:00:00 (00:00:00:00:00:00)
32 Internet Protocol Version 6, Src: ::1, Dst: ::1
33 Transmission Control Protocol, Src Port: 51574 (51574), Dst Port: 443
      (443), Seq: 6822, Ack: 2606, Len: 42
34 Secure Sockets Layer
35 HyperText Transfer Protocol 2
```

Listing C.20: SANE Server Push fault tolerance evaluation ungraceful termination of client

```
1       31 0.029009351     ::1                       52512        ::1
                              443                HTTP2     6381    HEADERS
2
3  Frame 31: 6381 bytes on wire (51048 bits), 6381 bytes captured (51048
      bits) on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
      _00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 6, Src: ::1, Dst: ::1
6  Transmission Control Protocol, Src Port: 52512 (52512), Dst Port: 443
      (443), Seq: 527, Ack: 1810, Len: 6295
7  Secure Sockets Layer
8  HyperText Transfer Protocol 2
9
10      32 0.029040836     ::1                       443          ::1
                              52512              TCP       86      443 -->
          52512 [ACK] Seq=1810 Ack=6822 Win=175744 Len=0 TSval=22175909
          TSecr=22175909
11
12 Frame 32: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on
      interface 0
13 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
      _00:00:00 (00:00:00:00:00:00)
14 Internet Protocol Version 6, Src: ::1, Dst: ::1
15 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 52512
      (52512), Seq: 1810, Ack: 6822, Len: 0
16
17      33 0.031763892     ::1                       443          ::1
                              52512              HTTP2     911
          PUSH_PROMISE, HEADERS, DATA
18
19 Frame 33: 911 bytes on wire (7288 bits), 911 bytes captured (7288 bits)
       on interface 0
20 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
      _00:00:00 (00:00:00:00:00:00)
21 Internet Protocol Version 6, Src: ::1, Dst: ::1
22 Transmission Control Protocol, Src Port: 443 (443), Dst Port: 52512
      (52512), Seq: 1810, Ack: 6822, Len: 825
23 Secure Sockets Layer
24 HyperText Transfer Protocol 2
25
26      34 0.069408921     ::1                       52512        ::1
                              443                TCP       86      52512 -->
          443 [ACK] Seq=6822 Ack=2635 Win=180480 Len=0 TSval=22175920
          TSecr=22175910
27
```

```
Frame 34: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on
    interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
    _00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 52512 (52512), Dst Port: 443
    (443), Seq: 6822, Ack: 2635, Len: 0

     35 2.432698924    ::1                   52512       ::1
                          443              TCP      86     52512 -->
        443 [FIN, ACK] Seq=6822 Ack=2635 Win=180480 Len=0 TSval
        =22176510 TSecr=22175910

Frame 35: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on
    interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
    _00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 52512 (52512), Dst Port: 443
    (443), Seq: 6822, Ack: 2635, Len: 0

     36 2.469401131    ::1                   443         ::1
                          52512            TCP      86     443 -->
        52512 [ACK] Seq=2635 Ack=6823 Win=175744 Len=0 TSval=22176520
        TSecr=22176510

Frame 36: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on
    interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
    _00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 443 (443), Dst Port: 52512
    (52512), Seq: 2635, Ack: 6823, Len: 0

     37 2.766235092    ::1                   443         ::1
                          52512            HTTP2    132    GOAWAY

Frame 37: 132 bytes on wire (1056 bits), 132 bytes captured (1056 bits)
     on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00
    _00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 443 (443), Dst Port: 52512
    (52512), Seq: 2635, Ack: 6823, Len: 46
Secure Sockets Layer
HyperText Transfer Protocol 2
```

Evaluation of *Smart Multiplexing* on $\mathcal{P} - \mathcal{S}$ link using libcurl

Listing C.21: Automatic smart multiplexing evaluation

```
1       4 0.000035471    127.0.0.1             45728       127.0.0.1
                    80               HTTP     217    GET /poc/
          idlestream.php HTTP/1.1
2
3  Frame 4: 217 bytes on wire (1736 bits), 217 bytes captured (1736 bits)
     on interface 0
4  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
     00:00:00_00:00:00 (00:00:00:00:00:00)
5  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
6  Transmission Control Protocol, Src Port: 45728 (45728), Dst Port: 80
     (80), Seq: 1, Ack: 1, Len: 151
7  Hypertext Transfer Protocol
8
9       6 0.000169542    127.0.0.1             80          127.0.0.1
                    45728            HTTP     137    HTTP/1.1 101
          Switching Protocols
10
11 Frame 6: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
     on interface 0
12 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
     00:00:00_00:00:00 (00:00:00:00:00:00)
13 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
14 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45728
     (45728), Seq: 1, Ack: 152, Len: 71
15 Hypertext Transfer Protocol
16
17      14 0.100700176    127.0.0.1             80          127.0.0.1
                   45728            HTTP2    187    HEADERS, DATA
18
19 Frame 14: 187 bytes on wire (1496 bits), 187 bytes captured (1496 bits)
      on interface 0
20 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
     00:00:00_00:00:00 (00:00:00:00:00:00)
21 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
22 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45728
     (45728), Seq: 109, Ack: 206, Len: 121
23 HyperText Transfer Protocol 2
24
25      16 0.200816332    127.0.0.1             80          127.0.0.1
                   45728            HTTP2    79     DATA
26
27 Frame 16: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on
     interface 0
28 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
     00:00:00_00:00:00 (00:00:00:00:00:00)
29 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
30 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45728
     (45728), Seq: 230, Ack: 206, Len: 13
31 HyperText Transfer Protocol 2
32
```

```
33      37 1.104043469    127.0.0.1             45732        127.0.0.1
                    80               HTTP     217    GET /poc/
        idlestream.php HTTP/1.1
34
35  Frame 37: 217 bytes on wire (1736 bits), 217 bytes captured (1736 bits)
     on interface 0
36  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
37  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
38  Transmission Control Protocol, Src Port: 45732 (45732), Dst Port: 80
    (80), Seq: 1, Ack: 1, Len: 151
39  Hypertext Transfer Protocol
40
41      39 1.104135783    127.0.0.1             80           127.0.0.1
                    45728            HTTP2    80     DATA
42
43  Frame 39: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on
    interface 0
44  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
45  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
46  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45728
    (45728), Seq: 348, Ack: 206, Len: 14
47  HyperText Transfer Protocol 2
48
49      41 1.104161868    127.0.0.1             80           127.0.0.1
                    45732            HTTP     137    HTTP/1.1 101
        Switching Protocols
50
51  Frame 41: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
     on interface 0
52  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
53  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
54  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45732
    (45732), Seq: 1, Ack: 152, Len: 71
55  Hypertext Transfer Protocol
56
57      51 1.204269728    127.0.0.1             80           127.0.0.1
                    45728            HTTP2    80     DATA
58
59  Frame 51: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on
    interface 0
60  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
61  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
62  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45728
    (45728), Seq: 362, Ack: 206, Len: 14
63  HyperText Transfer Protocol 2
64
65      53 1.204855549    127.0.0.1             80           127.0.0.1
                    45732            HTTP2    187    HEADERS, DATA
66
67  Frame 53: 187 bytes on wire (1496 bits), 187 bytes captured (1496 bits)
     on interface 0
```

```
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45732
    (45732), Seq: 109, Ack: 206, Len: 121
HyperText Transfer Protocol 2

     55 1.304427922    127.0.0.1             80          127.0.0.1
                    45728            HTTP2    80     DATA

Frame 55: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on
    interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45728
    (45728), Seq: 376, Ack: 206, Len: 14
HyperText Transfer Protocol 2

     57 1.304935946    127.0.0.1             80          127.0.0.1
                    45732            HTTP2    79     DATA

Frame 57: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on
    interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45732
    (45732), Seq: 230, Ack: 206, Len: 13
HyperText Transfer Protocol 2


    133 3.209556740    127.0.0.1             80          127.0.0.1
                    45728            HTTP2    80     DATA

Frame 133: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on
     interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45728
    (45728), Seq: 642, Ack: 206, Len: 14
HyperText Transfer Protocol 2

    136 3.209904940    127.0.0.1             80          127.0.0.1
                    45728            HTTP2    83     GOAWAY

Frame 136: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on
     interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45728
    (45728), Seq: 656, Ack: 207, Len: 17
HyperText Transfer Protocol 2

```

```
106     138 3.308755228    127.0.0.1             80          127.0.0.1
                    45732            HTTP2    80     DATA
107
108 Frame 138: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on
      interface 0
109 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
110 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
111 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45732
    (45732), Seq: 502, Ack: 206, Len: 14
112 HyperText Transfer Protocol 2
113
114     171 4.812610393    127.0.0.1             80          127.0.0.1
                    45732            HTTP2    83     GOAWAY
115
116 Frame 171: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on
      interface 0
117 Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
    00:00:00_00:00:00 (00:00:00:00:00:00)
118 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
119 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 45732
    (45732), Seq: 726, Ack: 207, Len: 17
120 HyperText Transfer Protocol 2
```

Listing C.22: curl.log

```
1  *   Trying 127.0.0.1...
2  * TCP_NODELAY set
3  * Connected to localhost (127.0.0.1) port 443 (#0)
4  * ALPN, offering h2
5  * ALPN, offering http/1.1
6  * Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:
     @STRENGTH
7  * successfully set certificate verify locations:
8  *   CAfile: /etc/ssl/certs/ca-certificates.crt
9    CApath: none
10 * SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
11 * ALPN, server accepted to use h2
12 * Server certificate:
13 *  subject: C=DE; ST=Saxony; L=Dresden; O=TU Dresden; CN=127.0.0.1;
     emailAddress=timo.lutz@mailbox.tu-dresden.de
14 *  start date: Jul 27 17:13:44 2016 GMT
15 *  expire date: Jul 27 17:13:44 2017 GMT
16 *  issuer: C=DE; ST=Saxony; L=Dresden; O=TU Dresden; CN=127.0.0.1;
     emailAddress=timo.lutz@mailbox.tu-dresden.de
17 *  SSL certificate verify result: self signed certificate (18),
     continuing anyway.
18 * Using HTTP2, server supports multi-use
19 * Connection state changed (HTTP/2 confirmed)
20 * Copying HTTP/2 data in stream buffer to connection buffer after
     upgrade: len=0
21 * Using Stream ID: 1 (easy handle 0x7fea1c052850)
22 > GET /poc/idlestream.php HTTP/1.1
23 Host: localhost
24 Accept: */*
```

```
25
26 * Connection state changed (MAX_CONCURRENT_STREAMS updated)!
27 < HTTP/2 200
28 < date: Thu, 09 Mar 2017 20:58:48 GMT
29 < server: Apache/2.4.25 (Ubuntu)
30 < expires: 0
31 < cache-control: must-revalidate, post-check=0, pre-check=0
32 < content-type: text/plain;charset=UTF-8
33 <
34 *   Trying 127.0.0.1...
35 * TCP_NODELAY set
36 * Connected to localhost (127.0.0.1) port 443 (#0)
37 * ALPN, offering h2
38 * ALPN, offering http/1.1
39 * Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:
     @STRENGTH
40 * successfully set certificate verify locations:
41 *   CAfile: /etc/ssl/certs/ca-certificates.crt
42   CApath: none
43 * SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
44 * ALPN, server accepted to use h2
45 * Server certificate:
46 *  subject: C=DE; ST=Saxony; L=Dresden; O=TU Dresden; CN=127.0.0.1;
     emailAddress=timo.lutz@mailbox.tu-dresden.de
47 *  start date: Jul 27 17:13:44 2016 GMT
48 *  expire date: Jul 27 17:13:44 2017 GMT
49 *  issuer: C=DE; ST=Saxony; L=Dresden; O=TU Dresden; CN=127.0.0.1;
     emailAddress=timo.lutz@mailbox.tu-dresden.de
50 *  SSL certificate verify result: self signed certificate (18),
     continuing anyway.
51 * Using HTTP2, server supports multi-use
52 * Connection state changed (HTTP/2 confirmed)
53 * Copying HTTP/2 data in stream buffer to connection buffer after
     upgrade: len=0
54 * Using Stream ID: 1 (easy handle 0x7fea1c044960)
55 > GET /poc/idlestream.php HTTP/1.1
56 Host: localhost
57 Accept: */*
58
59 * Connection state changed (MAX_CONCURRENT_STREAMS updated)!
60 < HTTP/2 200
61 < date: Thu, 09 Mar 2017 20:58:52 GMT
62 < server: Apache/2.4.25 (Ubuntu)
63 < expires: 0
64 < cache-control: must-revalidate, post-check=0, pre-check=0
65 < content-type: text/plain;charset=UTF-8
66 <
```

Evaluation of parallel execution of libcurl requests

Listing C.23: Parallel execution via libcurl multi handle over FastCGI

```
     38 9.761004273    127.0.0.1             59472       127.0.0.1
                    80               HTTP     217    GET /poc/
        filestream.php HTTP/1.1

Frame 38: 217 bytes on wire (1736 bits), 217 bytes captured (1736 bits)
    on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 59472 (59472), Dst Port: 80
   (80), Seq: 1, Ack: 1, Len: 151
Hypertext Transfer Protocol
    GET /poc/filestream.php HTTP/1.1\r\n
    Connection: Upgrade, HTTP2-Settings\r\n
    Upgrade: h2c\r\n
    [Full request URI: http://localhost/poc/filestream.php]
    [Response in frame: 40]

     40 9.761150957    127.0.0.1             80          127.0.0.1
                    59472            HTTP     137    HTTP/1.1 101
        Switching Protocols

Frame 40: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
    on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 59472
   (59472), Seq: 1, Ack: 152, Len: 71
Hypertext Transfer Protocol
    HTTP/1.1 101 Switching Protocols\r\n
    Upgrade: h2c\r\n
    Connection: Upgrade\r\n
    [Request in frame: 38]

     46 9.761249673    127.0.0.1             59472       127.0.0.1
                    80               HTTP2    106    HEADERS

Frame 46: 106 bytes on wire (848 bits), 106 bytes captured (848 bits)
   on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 59472 (59472), Dst Port: 80
   (80), Seq: 197, Ack: 72, Len: 40
HyperText Transfer Protocol 2
    Stream: HEADERS, Stream ID: 3, Length 31
        Flags: 0x05
        Header: :method: GET
        Header: :path: /poc/filestream.php
        Header: :scheme: http
```

```
38
39      58 34.762157615    127.0.0.1              80           127.0.0.1
                    59472             HTTP2     3230    HEADERS, HEADERS,
          DATA, DATA
40
41  Frame 58: 3230 bytes on wire (25840 bits), 3230 bytes captured (25840
     bits) on interface 0
42  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
     00:00:00_00:00:00 (00:00:00:00:00:00)
43  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
44  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 59472
     (59472), Seq: 109, Ack: 246, Len: 3164
45  HyperText Transfer Protocol 2
46      Stream: HEADERS, Stream ID: 3, Length 99
47          Flags: 0x04
48          Header: :status: 200
49      Stream: HEADERS, Stream ID: 1, Length 29
50          Flags: 0x04
51          Header: :status: 200
52      Stream: DATA, Stream ID: 3, Length 1500
53          Flags: 0x01
54      Stream: DATA, Stream ID: 1, Length 1500
55          Flags: 0x01
56
57      61 35.763544553    127.0.0.1              80           127.0.0.1
                    59472             HTTP2     83      GOAWAY
58
59  Frame 61: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on
     interface 0
60  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
     00:00:00_00:00:00 (00:00:00:00:00:00)
61  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
62  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 59472
     (59472), Seq: 3273, Ack: 247, Len: 17
63  HyperText Transfer Protocol 2
64      Stream: GOAWAY, Stream ID: 0, Length 8
65          .000 0000 0000 0000 0000 0000 0000 0011 = Promised-Stream-ID: 3
66          Error: NO_ERROR (0)
```

Listing C.24: Parallel exeuction via libcurl multi handle over Apache PHP module

```
1       31 0.006640730     127.0.0.1              59574        127.0.0.1
                    80                HTTP      217     GET /poc/
          filestream.php HTTP/1.1
2
3   Frame 31: 217 bytes on wire (1736 bits), 217 bytes captured (1736 bits)
      on interface 0
4   Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
     00:00:00_00:00:00 (00:00:00:00:00:00)
5   Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
6   Transmission Control Protocol, Src Port: 59574 (59574), Dst Port: 80
     (80), Seq: 1, Ack: 1, Len: 151
7   Hypertext Transfer Protocol
8       GET /poc/filestream.php HTTP/1.1\r\n
9       Connection: Upgrade, HTTP2-Settings\r\n
```

```
    Upgrade: h2c\r\n
    [Full request URI: http://localhost/poc/filestream.php]
    [Response in frame: 33]

     33 0.006860252    127.0.0.1             80          127.0.0.1
                    59574            HTTP     137    HTTP/1.1 101
        Switching Protocols

Frame 33: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
    on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 59574
   (59574), Seq: 1, Ack: 152, Len: 71
Hypertext Transfer Protocol
    HTTP/1.1 101 Switching Protocols\r\n
    Upgrade: h2c\r\n
    Connection: Upgrade\r\n
    [Request in frame: 31]

     38 0.006989115    127.0.0.1             59574       127.0.0.1
                    80               HTTP2    106    HEADERS

Frame 38: 106 bytes on wire (848 bits), 106 bytes captured (848 bits)
   on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 59574 (59574), Dst Port: 80
   (80), Seq: 197, Ack: 72, Len: 40
HyperText Transfer Protocol 2
    Stream: HEADERS, Stream ID: 3, Length 31
        Flags: 0x05
        Header: :method: GET
        Header: :path: /poc/filestream.php
        Header: :scheme: http

     42 5.008246115    127.0.0.1             80          127.0.0.1
                    59574            HTTP2    483    HEADERS, DATA

Frame 42: 483 bytes on wire (3864 bits), 483 bytes captured (3864 bits)
    on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
   00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 59574
   (59574), Seq: 109, Ack: 246, Len: 417
HyperText Transfer Protocol 2
    Stream: HEADERS, Stream ID: 1, Length 99
        Flags: 0x04
        Header: :status: 200
    Stream: DATA, Stream ID: 1, Length 300
        Flags: 0x00

```

```
52      44 10.008333653    127.0.0.1             80          127.0.0.1
                      59574            HTTP2    375    DATA
53
54  Frame 44: 375 bytes on wire (3000 bits), 375 bytes captured (3000 bits)
        on interface 0
55  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
56  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
57  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 59574
        (59574), Seq: 526, Ack: 246, Len: 309
58  HyperText Transfer Protocol 2
59      Stream: DATA, Stream ID: 1, Length 300
60          Flags: 0x00
61
62      57 30.009430923    127.0.0.1             80          127.0.0.1
                      59574            HTTP2    413    HEADERS, DATA
63
64  Frame 57: 413 bytes on wire (3304 bits), 413 bytes captured (3304 bits)
        on interface 0
65  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
66  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
67  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 59574
        (59574), Seq: 1762, Ack: 246, Len: 347
68  HyperText Transfer Protocol 2
69      Stream: HEADERS, Stream ID: 3, Length 29
70          Flags: 0x04
71          Header: :status: 200
72      Stream: DATA, Stream ID: 3, Length 300
73          Flags: 0x00
74
75      59 35.009514709    127.0.0.1             80          127.0.0.1
                      59574            HTTP2    375    DATA
76
77  Frame 59: 375 bytes on wire (3000 bits), 375 bytes captured (3000 bits)
        on interface 0
78  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
79  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
80  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 59574
        (59574), Seq: 2109, Ack: 246, Len: 309
81  HyperText Transfer Protocol 2
82      Stream: DATA, Stream ID: 3, Length 300
83          Flags: 0x00
84
85      72 50.010417389    127.0.0.1             80          127.0.0.1
                      59574            HTTP2    83     GOAWAY
86
87  Frame 72: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on
        interface 0
88  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
        00:00:00_00:00:00 (00:00:00:00:00:00)
89  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
90  Transmission Control Protocol, Src Port: 80 (80), Dst Port: 59574
        (59574), Seq: 3345, Ack: 247, Len: 17
91  HyperText Transfer Protocol 2
```

```
92      Stream: GOAWAY, Stream ID: 0, Length 8
93          Flags: 0x00
94          .000 0000 0000 0000 0000 0000 0000 0011 = Promised-Stream-ID: 3
95          Error: NO_ERROR (0)
```

D COPYRIGHT PERMISSIONS

The CC-BY-NC-ND 4.0 license applies to the following figures

- Figure 2.1
  Title: Three-way handshake
  Creator: Ilya Grigorik
  Source: https://hpbn.co/building-blocks-of-tcp/
- Figure 2.3
  Title: HTTP/2 streams, messages and frames
  Creator: Ilya Grigorik
  Source: https://hpbn.co/http2/

E INDEX

LIST OF FIGURES

LIST OF TABLES

BIBLIOGRAPHY

[Ber91] Tim Berners-Lee. "The Original HTTP as defined in 1991". In: *World Wide Web Consortium (W3C)* (1991) (cit. on p. 19).

[BFF] Tim Berners-Lee, R Fielding, and H Frystyk. *Hypertext transfer protocol– HTTP/1.0. RFC1945, May 1996* (cit. on pp. 19, 21).

[BFM05] Tim Berners-Lee, R Fielding, and Larry Masinter. "RFC 3986". In: *Uniform Resource Identifier (URI): Generic Syntax* (2005) (cit. on p. 68).

[BPT15] Mike Belshe, Roberto Peon, and M Thomson. "RFC 7540: hypertext transfer protocol version 2 (HTTP/2)". In: *Internet Engineering Task Force (IETF)//BitGo, Google Inc.//May* (2015) (cit. on pp. 23, 25, 27, 33, 45, 50, 61, 68, 83, 84, 96).

[Chu+13] Jerry Chu et al. *Increasing TCP's initial window*. Tech. rep. 2013 (cit. on p. 18).

[CSH15] Shaiful Alam Chowdhury, Varun Sapra, and Abram Hindle. "Is HTTP/2 more energy efficient than HTTP/1.1 for mobile users?" In: *PeerJ PrePrints* 3 (2015), e1571 (cit. on p. 36).

[Gri13a] Ilya Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. " O'Reilly Media, Inc.", 2013 (cit. on pp. 13, 19, 21, 33).

[Gri13b] Ilya Grigorik. "Making the Web Faster with HTTP 2.0". In: *Commun. ACM* 56.12 (Dec. 2013), pp. 42–49. ISSN: 0001-0782. DOI: 10.1145/2534706.2534721. URL: http://doi.acm.org/10.1145/2534706.2534721 (cit. on pp. 22–24, 33).

[Gro+99] Network Working Group et al. "RFC 2616: Hypertext Transfer Protocol–HTTP/1.1". In: *R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee* (1999) (cit. on p. 21).

[GT02] David Gourley and Brian Totty. *HTTP: the definitive guide*. " O'Reilly Media, Inc.", 2002 (cit. on pp. 19, 21, 22).

[Har12] Tenshi Hara. "Towards a reliable Architecture for Crowdsourcing in the Context of the MapBiquitous Project". Diplomarbeit. Technische Universität Dresden, Oct. 2012 (cit. on p. 29).

[HHQ15] Bo Han, Shuai Hao, and Feng Qian. "MetaPush: Cellular-Friendly Server Push For HTTP/2". In: *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*. ACM. 2015, pp. 57–62 (cit. on p. 37).

[HTT] HTTPWatch. *A simple performance comparison of HTTPS, SPDY and HTTP/2*. URL: http://blog.httpwatch.com/2015/01/16/a-simple-performance-comparison-of-https-spdy-and-http2/ (visited on 11/30/2016) (cit. on p. 35).

[Jac] Brian Jackson. *HTTP/2 statistics - KeyCDN report on HTTP/2 distribution*. (Visited on 03/28/2017) (cit. on p. 34).

[Kra] Vlad Krasnov. *HPACK: the silent killer (feature) of HTTP/2*. URL: https://blog.cloudflare.com/hpack-the-silent-killerfeature-of-http-2/ (visited on 02/14/2017) (cit. on p. 27).

[NGI15] NGINX. *HTTP/2 for Web Application Developers*. Tech. rep. 4. NGINX Inc., Sept. 2015 (cit. on p. 33).

[PB] Roberto Peon and Mike Belshe. *SPDY Protocol - Draft 3*. URL: https://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3 (visited on 09/30/2016) (cit. on p. 23).

[PHP15] PHP.net. *PHP 7.0 backward incompatible changes*. 2015. URL: http://php.net/manual/en/migration70.incompatible.php (visited on 03/16/2017) (cit. on p. 58).

[PHP16a] PHP.net. *PHP 7.1 backward incompatible changes*. 2016. URL: http://php.net/manual/en/migration71.incompatible.php (visited on 03/16/2017) (cit. on p. 58).

[PHP16b] PHP.net. *PHP manual - getallheaders()*. 2016. URL: http://php.net/manual/en/function.curl-setopt.php (visited on 03/28/2017) (cit. on p. 58).

[PHP16c] PHP.net. *PHP manual - getallheaders()*. 2016. URL: http://php.net/manual/en/function.getallheaders.php (visited on 03/13/2017) (cit. on p. 90).

[PR15] Roberto Peon and Herve Ruellan. *HPACK: Header Compression for HTTP/2*. Tech. rep. 2015 (cit. on pp. 23, 27, 33, 68).

[Pu16] Junyu Pu. "Adapting Server Frameworks to Support HTTP/2 in Proxy Settings". MA thesis. Technische Universität Dresden, 2016 (cit. on pp. 27, 30, 37, 41, 52, 57, 78, 81, 82, 95, 96).

[SH16] Varun Sapra and Abram Hindle. "Web servers energy efficiency under HTTP/2". In: *PeerJ Preprints* 4 (2016), e2027v1 (cit. on p. 36).

[SOC15] Hugues de Saxcé, Iuniana Oprescu, and Yiping Chen. "Is HTTP/2 really faster than HTTP/1.1?" In: *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2015, pp. 293–299 (cit. on p. 34).

[Stea] Daniel Stenberg. *HTTP/2 with curl*. URL: https://curl.haxx.se/docs/http2.html (visited on 01/31/2017) (cit. on p. 58).

[Steb] Daniel Stenberg. *Multi interface overview*. (Visited on 03/18/2017) (cit. on p. 89).

[Ste14] Daniel Stenberg. "HTTP2 explained." In: *Computer Communication Review* 44.3 (2014), pp. 120–128 (cit. on pp. 22, 23, 33).

[Ste15] Daniel Stenberg. *HTTP/2 in CURL, status update*. 2015. URL: https://daniel.haxx.se/blog/2015/05/04/http2-in-curl-status-update/ (visited on 03/09/2017) (cit. on pp. 77, 88).

[Tes] WebPage Test. *http archive*. URL: http://httparchive.org/trends.php (visited on 09/19/2016) (cit. on p. 22).

[Var+] Matteo Varvello et al. *HTTP/2 Dashboard*. URL: http://isthewebhttp2yet.com/measurements/adoption.html (visited on 03/26/2017) (cit. on pp. 33, 34).

[Var+16] Matteo Varvello et al. "Is the Web HTTP/2 Yet?" In: *International Conference on Passive and Active Network Measurement*. Springer. 2016, pp. 218–232 (cit. on pp. 33, 34).

[W3T] W3Techs. *Usage of HTTP/2 for websites*. (Visited on 03/28/2017) (cit. on p. 34).

[WS14a] Sheng Wei and Viswanathan Swaminathan. "Cost effective video streaming using server push over HTTP 2.0". In: *Multimedia Signal Processing (MMSP), 2014 IEEE 16th International Workshop on*. IEEE. 2014, pp. 1–5 (cit. on p. 37).

[WS14b] Sheng Wei and Viswanathan Swaminathan. "Low latency live video streaming over HTTP 2.0". In: *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*. ACM. 2014, p. 37 (cit. on p. 37).

[WSX15] Sheng Wei, Viswanathan Swaminathan, and Mengbai Xiao. "Power efficient mobile video streaming using HTTP/2 server push". In: *Multimedia Signal Processing (MMSP), 2015 IEEE 17th International Workshop on*. IEEE. 2015, pp. 1–6 (cit. on p. 37).

[Xia+16] Mengbai Xiao et al. "Evaluating and improving push based video streaming with HTTP/2". In: *Proceedings of the 26th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM. 2016, p. 3 (cit. on p. 37).

F DECLARATION OF AUTHORSHIP

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, 31th March 2017