HTTP/2

Michael Conrads
Betreuer: Benjamin Hof
Seminar Future Internet SS2015
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: conrads@in.tum.de

ABSTRACT

The new version of HTTP/1.1, HTTP/2, has been approved by the IETF's steering group for publication as an RFC. The number of assets per web page has increased over the last ten years. Optimization techniques like sharding, CSS data: inlining or image sprites are used to address speed issues inherent to the HTTP/1.X protocols. HTTP/2 removes the need for these optimizations by introducing binary frames, transmitted over streams on a single TCP/IP connection. It enables the client to indicate priorities for streams to the server. Servers are able to push data to clients anticipating the clients need for resources like JavaScript or CSS files referenced in requested HTML. HTTP/2 transmits headers using the HPACK format, using redundancy in the header key-value pairs to reduce the transmitted header size. The HTTPbis working group reached its goal in creating an performance optimized HTTP protocol, but it is questionable if advanced features of HTTP/2 will be fully supported.

Keywords

SPDY, HTTP/2, HPACK

1. INTRODUCTION

HTTP is one of the most commonly used protocols on the web. It was first defined in 1991 [6] and was published as a Request For Comment (RFC) in 1996 (RFC 1954, Hypertext Transfer Protocol – HTTP/1.0) [10]. It was widely adopted and the next version HTTP/1.1 followed in 1999 [9]. In addition to functionality updates, speed issues were addressed by adding *pipelining* [9, section 8.1.2.2], which is explained in section 3. Furthermore, HTTP/1.1 made persistent TCP connections the default setting for HTTP [9, section 8.1.2]. The increasing number of images, CSS and JavaScript files per page require an increasing number of HTTP request, which made new drawbacks of HTTP/1.1 became apparent. Figure 1 shows the average web page transfer size as well as number of requests, based on the Alexa top 1 million pages between Jan 2011 and Mar 2015 [12]. As can be seen, the number of requests per page increased from 78 to 96 and the total transmission size increased from 724kB to 1977kB in the last 4 years.

HTTP/2 tries to address these problems and adapt HTTP to the requirements of the modern web.

2. A SHORT INTRODUCTION TO HTTP

HTTP/1.X (HTTP/1.0 and HTTP/1.1) are text-based protocols used for transmitting content between a client and a server in request-response form. A well known use case is the transmission of web pages. HTTP is built on top of the TCP/IP stack, guaranteeing reliable transfer between two parties. HTTP/1.X messages consists of a header and a body, separated by two carriage-return linefeeds (`CR LF CR LF`, `CR` and `LF` are control characters denoting a new line).

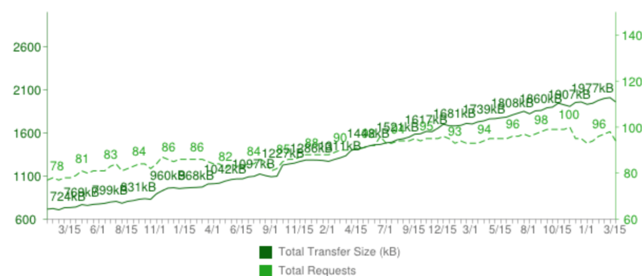

Figure 1: Average transfer size and number of requests required to load a web page

Listing 1 shows an exemplary HTTP/1.1 response from a server to a request from the client. Every header value is separated by one `CR LF`. The last header value '`Server: Apache`' is separated from the content of the response '`<!DOCTYPE html`' by `CR LF CR LF`, visible as the empty line after the header. The header of a HTTP message contains meta-information regarding the request or response, e.g. the method used for the request (GET, POST etc.) or the status code for the response as well as additional information like the type of the content or information about the server. The body of the HTTP response contains the requested content, in this example a page containing HTML.

Listing 1: Example HTTP Response from a Server

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: Apache

<!DOCTYPE html ...>
<html>
content of the web page here
</html>
```

3. PROBLEMS WITH HTTP/1.X

When requesting resources over HTTP/1.0, a new TCP connection is established for every resource transmitted. After the transmission of the resource, the TCP connection is closed again. With the number of resources per web page growing, clients need an increasing number of requests to load them [12]. The TCP *slow-start* mechanism was designed to decrease network load and find the optimal transmission rate for an established connection [21]. *Slow-start* gradually increases the number of transmitted bytes, based on the amount of transmission acknowledgments received. As a result, short lived TCP connections never reach their optimal transmission capacity.

HTTP/1.1 addresses these issues by allowing established TCP connections to be reused by default (`keep-alive`) [9, Section 8.1]. Nevertheless, the popular Apache webserver closes a connection if no request is sent over a period of time, which defaults to 5 seconds [22]. Even if the timeout on the server side is increased, clients maintain their own timeout setting e.g. in Firefox version 36 persistent connections timeouts after 115 seconds. To reuse an established TCP connection, the client has to wait for a response to a request before issuing a new one. HTTP/1.1 circumvents this problem by introducing Pipelining.

Pipelining enables the client to issue multiple requests to a server without having to wait for a response to the first request to issue the second one. The order of the requests is significant, as it is required that the server responds to these requests in the order they were issued. As a result, if the response to the first request is delayed (e.g. because of large file size), all other requests are delayed as well. This problem is called *head of line blocking*. As can be seen, by fixing one problem, HTTP/1.1 creates a new one.

4. HTTP/1.1 WORKAROUNDS

Several workarounds have been found to mitigate the issues described in section 3 and increase web page load speed:

CSS data: inline By including base64 encoded images in CSS stylesheets, multiple images can be loaded with one CSS request, reducing the number of requests to the server.

Spriting Multiple images are combined into one larger image, called a sprite. The individual images are displayed by specifying the offset X and Y coordinates in combination with the target image size [15]. This workaround aims at reducing the number of requests.

Sharding HTTP/1.1 states, a maximum of 2 parallel HTTP connections should be open simultaneously between a client and a server. [9, 8.1.4] Modern browsers don't conform to this standard. Firefox version 36.0.1 allows 6 persistent connection. Sharding refers to the technique of distributing assets of a page on multiple subdomains. These are considered seperate servers, thus resetting the *number-of-connections* limit. Figure 2 show a screenshot of a HTTP-request trace when visiting `www.gmx.de`. GMX distributes images via multiple domains to increase the number of connections between client and server.

Concatenation Another way of decreasing the number of HTTP requests is the concatenation of text based assets. Similar to the spriting technique for images, multiple JavaScript or CSS files are concatenated into larger files, reducing the number of requests sent to the server.

Domain	Type	Method	Scheme	Status
i0.gmx.net	Image	GET	HTTP	200
i0.gmx.net	Image	GET	HTTP	200
i2.gmx.net	Image	GET	HTTP	200
i0.gmx.net	Image	GET	HTTP	200
i0.gmx.net	Image	GET	HTTP	200
i2.gmx.net	Image	GET	HTTP	200
i1.gmx.net	Image	GET	HTTP	200
i1.gmx.net	Image	GET	HTTP	200
i2.gmx.net	Image	GET	HTTP	200
i0.gmx.net	Image	GET	HTTP	200

Figure 2: Sharding used by www.gmx.de

The existence of these workarounds points to a shift in usage of the HTTP/1.1 protocol in regard to its original purpose when it was created.

5. THE HTTP/2 WORKING GROUP

The Working Group (WG) charged with maintaining the HTTP specifications is part of the Internet Engineering Task Force (IETF) [13]. Its charter states the goals they want to achieve with the HTTP/2 standard [14]:

- "[...] us[e] draft-mbelshe-httpbis-spy-00 as starting point."
- "Substantially and measurably improve end-user perceived latency in most cases, over HTTP/1.1 using TCP."
- "Address the 'head of line blocking' problem in HTTP."
- "Not require multiple connections to a server to enable parallelism, thus improving its use of TCP, especially regarding congestion control."
- "Retain the semantics of HTTP/1.1, [...] including [...] HTTP methods, status codes, URIs, and [...] header fields." This goal aims at compability with websites accessing HTTP/1.1 header fields.
- "Clearly define how HTTP/2.0 interacts with HTTP/1.x" As HTTP/2 will be gradually rolled, dynamic upgrade mechanism from HTTP/1.1 connections to HTTP/2 are important to ensure a smooth transition which is not noticed by the user (e.g. no popup asking the user if he wants to use the HTTP/2 protocol).
- "Clearly identify any new extensibility points and policy for their appropriate use."

The fulfillment of these goals will be evaluated in section 9.

5.1 The SPDY Protocol

As stated in the WG charter, `draft-mbelshe-httpbis-spy-00` [4] of the SPDY protocol was used in 2012 as the starting point for HTTP/2 [16]. The SPDY protocol was developed by Google, which made the existence of the project known to the public in 2009. It was meant as a replacement for HTTP and "optimize the way browsers and servers communicate" [3]. SPDY features a binary format, stream multiplexing as well as header compression. These features were

retained in the HTTP/2 protocol and are described in the following sections. The SPDY Protocol is supported on the client side by Google Chrome since version 6 (2010), Firefox since version 13 (2012) and Safari since version 8 (2014) [8]. On the server side, SPDY support was added as a mod to Apache version 2.2 (2012) [20] and nginx since version 1.3 (2013) [1].

Google will retire the SPDY protocol in "early 2016", as "[...] HTTP/2 is well on the road to standardization." [2].

6. THE HTTP/2 PROTOCOL

This section gives a summary of the core features of the HTTP/2 protocol. Contrary to the HTTP/1.1 text based protocol, HTTP/2 is a binary protocol. The main difference between text based and binary protocols is the way in which they represent values. Text based protocols use ASCII key-value pairs like 'Content-Length: 42' (which is at least 16 bytes long) whereas binary protocols transmit values in designated bit positions (e.g. the first 3 bytes in an HTTP/2 message contain the length of the message). This space-efficient value representation reduces the size of transmitted messages.

HTTP/1.1 defines `CR LF` as the separator between header fields [9, section 4.1]. The header parsing is not clearly defined in some points, e.g. HTTP/1.1 defines four different ways to specify message length based on the context of the message or the existence of header fields [9, section 4.4]. The RFC even states an example of wrong implementations producing extra `CR LF` sequences [9, section 4.1]. HTTP/2 transmits data in frames, which are logical grouped bytes. The concept of frames and their different types are explained in section 6.2.

As opposed to HTTP/1.1, HTTP/2 defines a clear separation between header information and content. Header fields are transmitted using `HEADERS` frames, content is transmitted using `DATA` frames. In HTTP/1.1 header and content are transmitted together, separated by `CR LF CR LF`.

Another advantage of binary protocols is the reduction of protocol overhead in regard to number of bytes transmitted, as described at the top of this section. Common criticism of binary protocols is their illegibility when displayed during text based debugging. While this is true, TLS encrypted HTTP/1.1 messages are not human readable as well. In addition, the popular network packet analyzer Wireshark features HTTP/2 support[1].

Frames and Streams

Frames and streams are the core concepts of HTTP/2. Frames are the basic unit of data transmission between clients and servers. Similar to TCP or IP frames, they start with meta information about their purpose and payload and carry their content in the payload field. Frames are described in section 6.2. Frames are transmitted via streams. Streams are theoretical groups of frames which are transmitted over one TCP connection. The TCP connection is not aware of the existence of streams. Every frame carries a `stream identifier`, marking their assigned stream. Streams can be prioritized in regard to each other and allow multiplexing of frames on one TCP connection. Streams are described in section 6.3.

The following sections shows how an HTTP/2 connection is established and data transmitted between clients and servers.

6.1 Initiating a HTTP/2 connection

There are multiple ways to initialize a HTTP/2 connection, depending on the usage of TLS or cleartext transmission.

HTTPS over TLS

To establish a secure connection to a HTTPS URI, the client uses the application layer protocol negotiation (ALPN, [11]), an extension to the TLS protocol. As part of the TLS handshake, the client sends a list of supported application layer protocols (represented by protocol identifiers) to the server in the *ClientHello* message. The server responds with a selected protocol in the *ServerHello* message [11, section 3.1]. The protocol identifier used for HTTP/2 is `h2` [5, section 3.3].

After the completion of the TLS handshake, client and server have to send a connection preface [5, Section 3.4]. The *Connection Preface* is composed of the string '`PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n`' base 64 encoded and a `Settings` frame. `PRI` is a new HTTP method, used exclusively for the HTTP/2 connection establishment [5, section 11.6].

Cleartext HTTP

If the client is not aware of server HTTP/2 capabilities, it uses the HTTP/1.1 upgrade mechanism, issuing a HTTP/1.1 request to the server. This initial request contains a HTTP `Upgrade` header field with the value `h2c` (HTTP/2 cleartext) as well as a base64 encoded HTTP/2 `SETTINGS` frame as the value of the new header field `HTTP2-Settings` [5, section 3.2].

The server acknowledges the protocol change by sending a `101 switching protocols` HTTP/1.1 message. If the server does not support HTTP/2, it ignores the `Upgrade` and `HTTP2-Settings` header fields and respond with a HTTP/1.1 `200 OK` response.

If the client is aware of HTTP/2 capabilities of the server (e.g. from the `Alt-Svc`, see section 8 or previous connections) and wants to establish a cleartext TCP connection, it can send the *Connection Preface* and immediately start sending HTTP/2 frames.

Note the `\r\n\r\n` (`CR LF CR LF`) in the *Connection Preface*. When receiving the *Connection Preface*, the server starts scanning for a HTTP/1.1 header, which is ended by `CR LF CR LF`. In cases where the client assumes the server supports HTTP/2 from prior knowledge, but the server does not (e.g. server has been replaced), the server does not recognize the *Connection Preface* and does not attempt to scan the following HTTP/2 frames.

[1] `https://wiki.wireshark.org/HTTP2`

doi: 10.2313/NET-2015-09-1_02

6.2 Frames

Frames are the smallest unit of communication between client and server. Figure 3 shows the format of a HTTP/2 frame. The number in braces specifies the number of bits per field. Every frame starts with a `Length` field, containing the number of bytes in the `Payload` field. `Type` and `Flags` are discussed in the following section. `R` is a reserved bit. The `Stream Identifier` (stream ID) indicates the stream this frame is assigned to, see section 6.3. The `Frame Payload` field contains the actual payload of the frame.

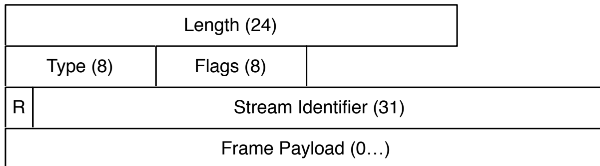

Figure 3: HTTP/2 Frame format

Frame Types and Flags

There exist several frame types, the most important ones are listed here, other frame types are mentioned where appropriate. The data transmitted by these frames is carried in the payload field. Unknown frame types must be discarded [5, section 4.1].

DATA A frame with type `DATA` transmits request or response payloads like HTML, CSS, images or other resources.

HEADERS The `HEADERS` frame is used to open a new stream and transmit *header fragments*, compressed via the compression format HPACK. If the HTTP header in its compressed form is too large the fit in one `HEADERS` frame, it is split up into *header fragments* where the first one is transmitted in the `HEADERS` frame and the following fragments are transmitted in `CONTINUATION` frames. If the `END_HEADERS` flag is set in one frame, it signals to the receiver the completed transmission of all *header fragments* and the receiver is able to assemble them to their decompressed HTTP header form. The HPACK compression and *header fragments* are explained in section 7.

An optional use of the `HEADERS` frame is to specify an initial stream dependency, explained in section 6.3.

SETTINGS The `SETTINGS` frame is used to control the entire connection, not just a single stream. The `Stream Identifier` must therefore be set to `0x00`. The `SETTINGS` frame must be sent from both endpoints to negotiate the connection at startup.
The `SETTINGS` frame is used to control the size of the dynamic header compression table (see section 7). It allows the clients to enable/disable `PUSH_PROMISE` frames and specify a flow control window size (`SETTINGS_INITIAL_WINDOW_SIZE`, see section 6.4) as well as a maximum frame size (`SETTINGS_MAX_FRAME_SIZE`).

PRIORITY The `PRIORITY` frame is used to indicate a dependency between two streams. It contains the identifier of the stream the current stream depends on, as well as a weight which is assigned to the siblings of the current stream. Stream dependencies and weights are explained in section 6.3.

PUSH_PROMISE The `PUSH_PROMISE` frame payload contains a promised stream ID and *header fragments*. The `PUSH_PROMISE` must only be sent from the server and indicates to the client the intent to push resources which are not (yet) requested. An example would be the request to a web page with an embedded image A. The server knows, that the client will request the image as soon as it receives the HTML with the embedded link. The server can send a `PUSH_PROMISE` frame *before* it sends the HTML in a `DATA` frame. The `PUSH_PROMISE` has to be sent *beforehand* to avoid race conditions (e.g. the client requests resources while the server is already sending them). The client receives the `PUSH_PROMISE` frame which contains a request header for the not yet requested image A. The client reserves a stream with the transmitted `Stream Identifier` and knows (from the header) that the server will push the image A on this stream. When the client parses the HTML, it can match the link to image A to the request header for image A sent in the `PUSH_PROMISE` from the server, thereby saving the request to the server for the image. In this regard, `PUSH_PROMISE` works similar to the HTTP/1.1 *Data: inline* technique with the possibility of client opt-out: If the client does not want to receive server pushes, it can disable them with the `SETTINGS_ENABLE_PUSH` flag in the `SETTINGS` frame. If the client wants to receive server pushes in general, but wants to cancel a specific `PUSH_PROMISE`, it can send a `RST_STREAM` frame.

RST_STREAM The `RST_STREAM` frame (Reset Stream) is used to cancel a stream. In HTTP/1.1, peers had no way of aborting an ongoing transmission of data without having to close the TCP connection. The `RST_STREAM` frame can be sent at any time to tell the peer that all transmissions on a specific stream should be canceled.

WINDOW_UPDATE The `WINDOW_UPDATE` frame is used to *increase* the flow control window. Flow control can apply to individual streams as well as the whole connection. If a peer wants to reduce the flow control window, it has to send a new `SETTINGS` frame, containing the new value for the setting `SETTINGS_INITIAL_WINDOW_SIZE`. Flow control is described in section 6.4.

6.3 Streams

A stream is a concept for bi-directional transmission of frames between client and host, not a physical existing connection. One TCP connection can contain multiple streams. Every frame is assigned to a stream. As streams are logical concepts, there is no overhead in establishing new streams, as opposed to establishing a new TCP connection. Streams are referenced by IDs. To circumvent a possible ID collision, streams opened by the server are assigned even numbered IDs, streams opened by the client odd numbered IDs [5, section 5.1.1].

Streams are opened using a `HEADERS` frame, identifying the resource that will be transmitted over the stream. If the `PRIORITY` flag is set, the `HEADERS` frame includes an initial stream priority and dependency. As opposed to HTTP/1.1 requests/responses, HTTP/2 stream communication is stateful. Both `HEADERS` and `DATA` frames are needed to transmit resources and their order is significant for their reassembly on the receiver side.

Priority and Dependency
Streams can be dynamically (re)prioritized using a `Priority` frame, which assigns priorities as weights. Weights are values between 1 and up to 255. The priority of a stream tells the server which requests it should answer first. If a server receives requests from a high-priority stream and a low priority stream, it should assign more resources (e.g. bandwidth) to answering the high-priority stream than the low-priority stream.

Streams can be dependent on each other. This creates a dependency tree on the server. The stream with ID 0 is not used and is inserted on the server to form the root of the tree. The HTTP/2 specification does not enforce strict compliance to the priority and dependency model [5, section 5.3.1 and 5.3.2].

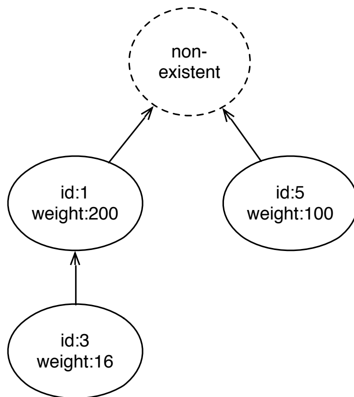

Figure 4: Stream Dependency example

Figure 4 shows an example for stream dependency. Streams 1 and 5 do not have a parent stream (the `Stream Dependency` field contained `0x00`) but different weights. This prompts the server to assign more resources to stream 1 than to stream 5. Stream 3 is depending on stream 1, which means requests on stream 1 should be handled before requests on stream 3. A weight of 16 is assigned as the default value, if no priority is specified.

Consider this exemplary transmission of a web page using the dependency in figure 4:
A client requests a web page, which contains an image and a JavaScript file. As the HTML contains the links to all needed resources, it should be transmitted first and will be requested over stream 1. The JavaScript file is transmitted over stream 3, as the JavaScript has to wait for the browser to build the DOM-tree before it is able to apply DOM manipulations. The image can be requested over stream 5, as soon as the HTML `DATA` frame containing the image link is parsed.

6.4 Data Transmission and Flow Control

Persistent connections introduced in HTTP/1.1 in combination with *pipelining* allow multiple HTTP requests in parallel. The order in which these requests are answered is used to match the responses to the requests and is therefore significant. Consider two exemplary *pipelined* HTTP/1.1 requests to a server, requesting a database entry and a static html page. *Pipelining* requires these requests to be answered in the same order, e.g. the response from the database followed by the static html page. As the response from the database takes longer than the static html page, the first response blocks the seconds one. Streams allow HTTP/2 frames to be multiplexed over one TCP connection. The order of frames is significant per stream, not per TCP connection. This ensures requests to slower resources (e.g. images or database entries) do not block faster responses, regardless of the order in which they are sent. Figure 5 visualizes a simplified HTTP/1.1 transmission without the usage of persistent connections to demonstrate the TCP and HTTP handshake overhead. The *index.html* response carries the payload as well as the HTTP headers response.

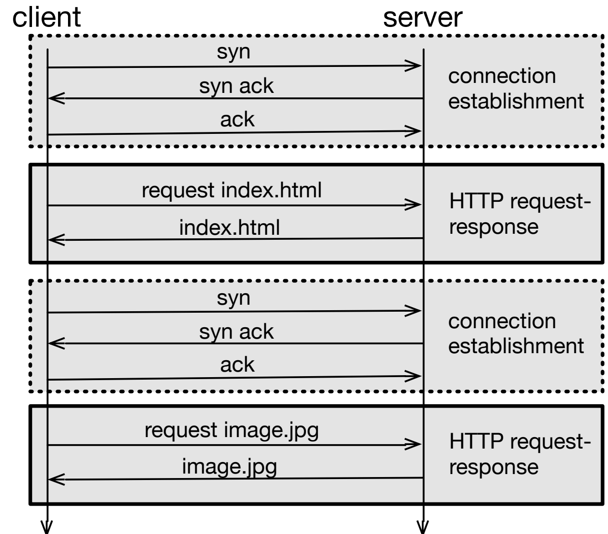

Figure 5: HTTP/1.1 transmission without keep-alive, simplified

Figure 6 shows the request and transmission of HTML as well as several images: `HEADERS` frames from the client open new streams and request resources. No TCP overhead is generated if a new stream is opened. The server responds by sending a `HEADERS` frame with the specified stream ID to transmit the HTTP response headers, followed by `DATA` frames carrying the resources. If the server transmitted a resource, the last frame on the stream carries the `END_STREAM` flag (`ES` in figure 6), signaling the end of transmission on the stream.

HTTP/2 provides a flow control mechanism for each TCP connection. Flow control applies only to `DATA` frames, all other frames are not affected. TCP flow control is meant to protect the client from receiving data faster than it can process. As multiple streams are transmitted over the same TCP connection, TCP flow control cannot apply to single streams. HTTP/2 flow control can restrict or increase the throughput on individual streams by transmitting `SETTINGS` or `WINDOW_UPDATE` frames. Every peer maintains a flow control window, which is initially set to 65,535 Bytes [5, section

doi: 10.2313/NET-2015-09-1_02

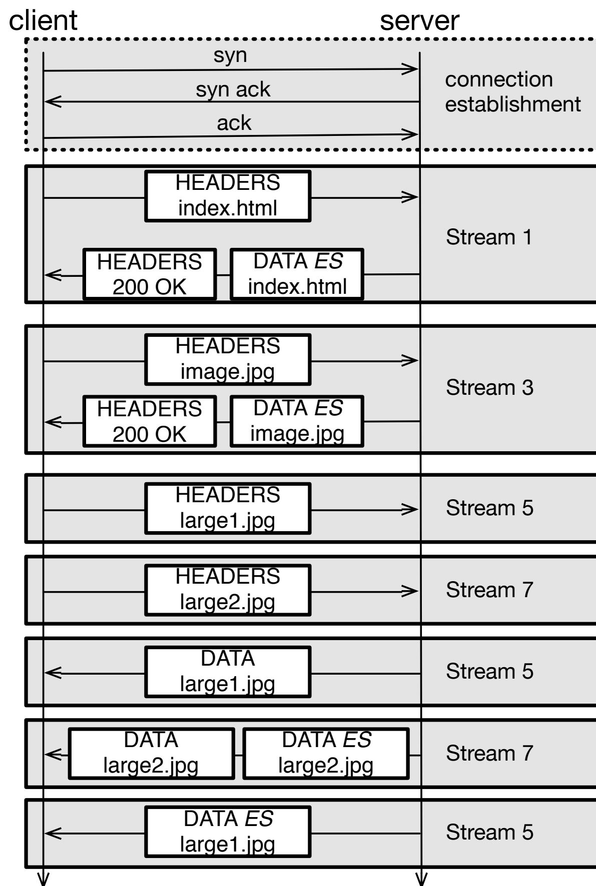

Figure 6: HTTP/2 transmission, simplified. ES denotes the END_STREAM flag

5.2.1]. While stream priorities are relative values regarding the processing of requests on the server side, flow control can set hard limits on individual streams.

7. HPACK HEADER COMPRESSION

One of the HTTP/2 goals is to "Retain the semantics of HTTP/1.1 [...] including [...] header fields." [14]. HTTP/1.1 header fields are very repetitive, with minimal changes between requests for resources. The header compression format `HPACK` [19] uses this repetition to substantially reduce transmitted header size.

`HPACK` allows header fields to be split into multiple *header fragments* if they are too large to fit into one frame. `HEADERS` and `CONTINUATION` frames are used to transmit *header fragments*. The client collects the fragments and assembles the complete header [19, section 1.1]. The format of the *header fragments* in the `HEADERS` frame is identical to the one in the `CONTINUATION` frame. For convenience, only the `HEADERS` frame is mentioned.

The `HPACK` compression uses two tables for decoding, a static table and a dynamic table.

Static Table

The static table contains fixed values for the most common header fields. The index 2 for example contains the HTTP method `GET`. Index 5 contains path `/index.html`. A transmission of the index 2 and the index 5 results in `GET /index.html`. The content of the static table can be found in [19, Appendix A.].

Dynamic Table

The dynamic table is empty at the start of the connection. Both client and server maintain the same dynamic table. `HEADERS` frames transmit headers in multiple key-value tuples inside *header fragments*, in combination with bit flags if the entry in the key or value field in each tuple is an index or a literal.

If a peer wants to send a header, it iterates over each field in the original HTTP header (consisting of key and value), looking for a representation in the static table. If an entry is found, the appropriate field in the `HEADERS` frame is set. Combinations of an indexed key and a literal value or vice versa are indicated by the bit flags for each tuple. If a representation cannot be found in the static table, the dynamic table is searched. If no entry can be found as well, the key or value is added to the dynamic table and added as a literal to the `HEADERS` frame. When decoding the `HEADERS` payload, the receiver adds the transmitted literals to its dynamic table as well, recreating the same dynamic table as the sender. If the same key or value is sent again in another request or response, its index from the dynamic table is used.

Hence, `HPACK` compression is not stateless, requiring every peer to maintain an identical encoding context in form of the dynamic table. To further reduce size, the `header fragments` can be *Huffman* encoded, which is indicated by a bit in the transmitting frame.

The `HPACK` specification describes only the decoding of headers. The encoder is required to transmit the header fragments in a way, such that the client is able to decode them [19, Section 2.]. Individual implementations (on client and server side) must come up on their own with efficient algorithms for table traversal and storage. The specification therefore does not provide an implementation proposal.

8. EXTENSIONS

HTTP/2 is a very strict protocol. The specification states, that all unknown frames have to be ignored by the receiver [5, Section 5.5]. Extension points exist to increase the flexibility of the protocol. HTTP/2 extensions are not part of the main specification, which states only how extensions must be registered.

Alternative Services

The *Alternative Services Extension* (Alt-Svc) [17] is an additional header field that can be transmitted using the `Alt-Svc` frame in a HTTP/2 connection or as a regular header field in HTTP/1.1. Servers can use the *Alt-Svc* frame to signal the client a resource is available in another location, too. It can be used to switch protocols, e.g. the server tells the client during a HTTP/1.1 connection the resource is also available via HTTP/2 on another port: `Alt-Svc:`

h2c=":8080". It can reroute the client to another endpoint as well: client visits www.domain.com/endpoint and receives Alt-Svc: http="www2.domain.com/endpoint". The client is now able to transition to the www2 server without having to notify the user. This is not to be confused with a HTTP redirect response (a 3XX status code). Alt-Svc allows the client to decide, if he wants to use the alternative service [17, 2.4]. In addition, Alt-Svc can offer multiple alternative locations for a resource; the client is free to choose any. The HTTP redirect status does not allow such fine control as it forces a hard redirect on the client. The client can response with a header field Alt-Used to notify the server of a successful transition. Using Alt-Svc, the server can transition a client slowly to another location until all requests are sent to the new location (in case a full transition is desired).

Opportunistic TLS

Opportunistic TLS [18] is an experimental extension based on Alt-Svc. It aims at mitigating pervasive monitoring attacks (RFC 7258). Servers advertise using Alt-Svc if their content is available over TLS as well. Clients are now able to choose between staying on the current cleartext connection, risking e.g. http traffic monitoring, or switch to a new TLS connection.

9. FULFILLMENT OF WG GOALS

The WG aimed at improving end-user perceived latency. Akamai, a large cloud service provider, released a HTTP/2 test page (https://http2.akamai.com/demo), demonstrating the speed improvements gained by using HTTP/2 over HTTP/1.1.

Another goal was to address the TCP *slow-start* performance problem. HTTP/1.1 tried to improve performance by introducing persistent connections, with led to *head of line blocking* problems. HTTP/2 mitigates the slow-start by reusing the same TCP connection. As opposed to HTTP/1.1, HTTP/2 stream multiplexing and stream priorities circumvent the *head of line blocking* issues.

HTTP/2 does not change HTTP/1.1 semantics. As soon as the client has reassembled all *header fragments*, the HTTP/2 header is identical to the HTTP/1.1 header. Therefore, migration to HTTP/2 does not break working HTTP/1.1 websites which access header fields (e.g. for cookies).

As the upgrade to HTTP/2 is not mandatory, the HTTP/2 specification provides methods for discovering server HTTP/2 capabilities during an existing HTTP/1.1 cleartext connection or a TLS handshake using ALPN without breaking existing HTTP/1.1 implementations.

The strength of HTTP/2 is the reduction of number of round trips, as less TCP connections need to be established and the transmitted header size is reduced. If the RTT is already low because of good network conditions, the impact of HTTP/2 for the end user will not be felt as strongly.

10. CONCLUSION

The success of HTTP/2 can be measured on its rate of adoption. All major browsers have added HTTP/2 and/or SPDY support. Due to the HTTP/1.1 header compatibility and the HTTP/2 support in Apache and nginx, gradual roll out of the protocol should be straightforward for most servers, eliminating the need for the HTTP/1.1 workarounds mentioned section 4. According to Daniel Stenberg, a member of the HTTP/2 WG, Google reported 5% of their global traffic uses HTTP/2 by the end of January 2015 [7]. While this early adoption rate sounds promising, it is difficult to validate this number and therefore should be treated with caution. In the opinion of the author, HTTP/2 coverage will never reach 100%, as many small or "abandoned" private webservers will never be updated (following the principle: 'if it ain't broken, don't fix it').

Several speed comparisons between HTTP/1.1 and HTTP/2 exist, claiming varying speed improvements using HTTP/2 over HTTP/1.1[2]. These comparisons are often lacking in information regarding the test setup and are dubious in their expressiveness. HTTP/2 core features like stream multiplexing or header compression aim at increasing responsiveness for web pages using a large number of assets. To objectively measure HTTP/2 performance, a web page operator has to measure HTTP/1.1 web page load times with current HTTP/1.1 workarounds in place and compare the results to the same page using HTTP/2 without using the mentioned workarounds. This test requires a full HTTP/2 implementation (using features like promise and stream priorities) on server and client side as well as a web page with actual content. While HTTP/2 test pages show promising results, the use case of requesting 200 images at once seems questionable. Therefore it is very difficult to provide actual HTTP/2 test results using real web page content and would go beyond the scope of this paper.

Nevertheless, the author is certain that the impact of HTTP/2 for the end user will be felt on mobile devices due to the efficient usage of TCP. HTTP/2 offers advantages for web pages with large numbers of assets as well, removing the need for many HTTP/1.1 optimizations. The webserver support for advanced features of HTTP/2 like stream dependency, priorities or push promises seems questionable though, as the HTTP/2 specification does not strictly require the server to adhere to them or use them at all.

[2]http2.golang.org/gophertiles or blog.httpwatch.com/2015/01/16/a-simple-performance-comparison-of-https-spdy-and-http2

11. REFERENCES

[1] Bartenev, V. Announcing SPDY draft 2 implementation in nginx. http://mailman.nginx.org/pipermail/nginx-devel/2012-June/002343.html, June 2012.

[2] Beky, B. and Bentzel, C. Hello HTTP/2, Goodbye SPDY. http://blog.chromium.org/2015/02/hello-http2-goodbye-spdy-http-is_9.html, Feb. 2015.

[3] Belshe, M. and Peon, R. A 2x Faster Web. http://googleresearch.blogspot.de/2009/11/2x-faster-web.html, Nov. 2009.

[4] Belshe, M. and Peon, R. SPDY Protocol. http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00, Feb. 2012. Internet-Draft.

[5] Belshe, M. and Peon, R. and Thomson, M. Hypertext Transfer Protocol version 2, Draft 17. https://tools.ietf.org/html/draft-ietf-httpbis-http2-17, Feb. 2015. Internet-Draft.

[6] Tim Berners-Lee. The Original HTTP as defined in 1991. http://www.w3.org/Protocols/HTTP/AsImplemented.html.

[7] Daniel Stenberg. HTTP/2 is at 5%. http://daniel.haxx.se/blog/2015/02/10/http2-is-at-5/, Feb. 2015.

[8] Deveria, A. HTTP/2 protocol / SPDY. http://caniuse.com/#feat=spdy, Feb. 2015. dynamically generated.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol-HTTP/1.1, June 1999. RFC 2616.

[10] R Fielding, U C Irvine, and H Frystyk. HTTP1.0, May 1996. RFC 1945.

[11] Friedl, S. and Popov, A. and Langley, A. and Stephan, E. Transport Layer Security (TLS), Application-Layer Protocol Negotiation Extension, July 2014. RFC 7301.

[12] httparchive.org. Trends. http://httparchive.org/trends.php?s=All&minlabel=Jan+20+2011&maxlabel=Mar+15+2015, Mar. 2015. dynamically generated.

[13] IETF HTTP Working Group. About Us. http://httpwg.github.io/about.

[14] IETF HTTP Working Group. Charter for Working Group. http://datatracker.ietf.org/wg/httpbis/charter/, Oct. 2012.

[15] Mozilla Developer Network. CSS Image Sprites. https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/CSS_Image_Sprites, 2015.

[16] Nottingham, M. Rechartering HTTPbis. https://lists.w3.org/Archives/Public/ietf-http-wg/2012JanMar/0098.html, Feb. 2012. HTTPbis Mailing List.

[17] Nottingham, M. and McManus, P. and Reschke, J. HTTP Alternative Services, Draft 6. http://tools.ietf.org/html/draft-ietf-httpbis-alt-svc-06, Feb. 2015. Internet-Draft.

[18] Nottingham, M. and Thomson, M. Opportunistic Security for HTTP. https://tools.ietf.org/html/draft-ietf-httpbis-http2-encryption-01, Dec. 2014.

[19] R Peon and H. Ruellan. HPACK - Header Compression for HTTP/2. http://tools.ietf.org/html/draft-ietf-httpbis-header-compression-12, Feb. 2015. Internet-Draft.

[20] Steele, M. mod_spdy is now an Apache project. http://googledevelopers.blogspot.de/2014/06/modspdy-is-now-apache-project.html, June 2014.

[21] Stevens, W. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, Jan. 1997. RFC 2001.

[22] The Apache Software Foundation. Apache-Kernfunktionen. https://httpd.apache.org/docs/2.4/mod/core.html#keepalivetimeout, 2015.

All online sources were last accessed on 28. April 2015.