HTTP/2

Péter Jeszenszky
Faculty of Informatics, University of Debrecen
jeszenszky.peter@inf.unideb.hu

Last modified: December 8, 2024

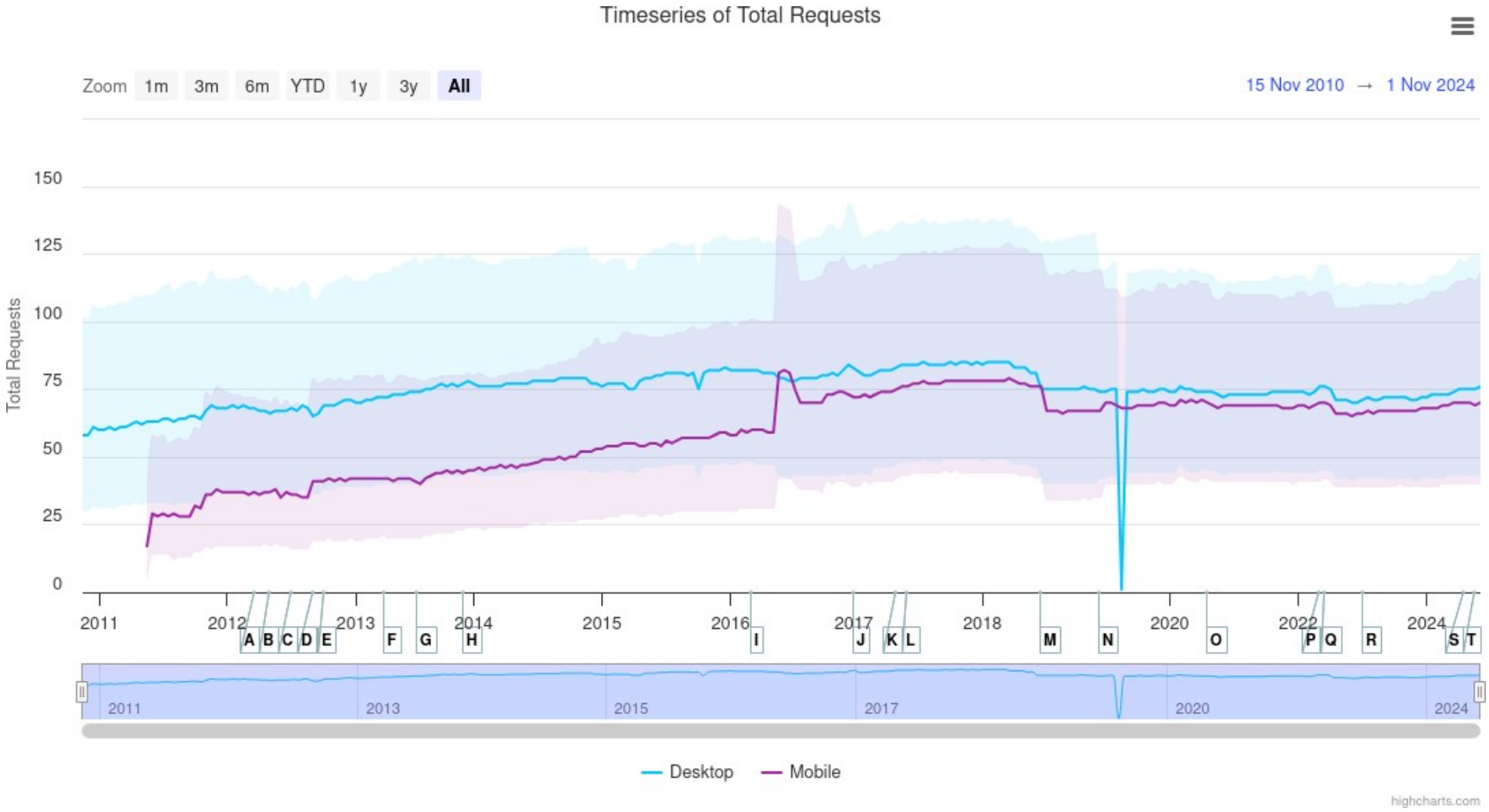
HTTP/1.*x* Characteristics that Affect Performance Negatively (1)

- HTTP/1.0 allowed only one request to be outstanding at a time on a given TCP connection.

HTTP/1.*x* Characteristics that Affect Performance Negatively (2)

- HTTP/1.1 added request pipelining, but this only partially addressed request concurrency and still suffers from head-of-line blocking.
  - Therefore, HTTP clients that need to make many requests use multiple connections to a server to achieve concurrency and thereby reduce latency.
- Furthermore, HTTP header fields are often repetitive and verbose, causing unnecessary network traffic as well as causing the initial TCP congestion window to quickly fill.

Statistics (1)



Total Requests: The number of resources requested by the page (November 15, 2010–October 1, 2023). Source: https://httparchive.org/reports/state-of-the-web

Statistics (2)

- About the mysterious gap between December 2016 and February 2017, see:
  - *Why is there gap between Dec 2016 and Feb 2017 trends?* https://discuss.httparchive.org/t/why-is-there-gap-between-dec-2016-and-feb-2017-trends/919

Best Practices for Reducing Latency (1)

- **Inlining**: embedding images directly into CSS stylesheets.
  - ```
    {
      background:
    url(
        ⟨Base64-encoded data⟩) no-repeat;
    }
    ```
  - See: Chris Coyier, *Data URIs*, March 25, 2010. https://css-tricks.com/data-uris/
    ```

Best Practices for Reducing Latency (2)

- **Spriting**: combining multiple images into a single image file.
  - See: Chris Coyier, CSS Sprites: *What They Are, Why They're Cool, and How To Use Them*, October 24, 2009. https://css-tricks.com/css-sprites/
  - Example: *MDN Web Docs* https://web.archive.org/web/20170701113221/https://developer.mozilla.org/en-US/
    - https://web.archive.org//web/20170701211407im_/https://developer.cdn.mozilla.net/static/img/logo_sprite.7d36c4a1422b.svg

Best Practices for Reducing Latency (3)

- **Sharding**: distributing content among multiple web servers.
  - For example, it may be worth serving images from a separate web server that does not use any cookies.
  - Related concept: Content Delivery Network (CDN)
    - CDN providers:
      - Azure Content Delivery Network https://azure.microsoft.com/en-us/products/cdn
      - cdnjs https://cdnjs.com/
      - Cloudflare https://www.cloudflare.com/
      - Google https://cloud.google.com/cdn
      - jsDelivr https://www.jsdelivr.com/
      - …
    - See: https://developer.mozilla.org/en-US/docs/Glossary/CDN

Best Practices for Reducing Latency (4)

- **Concatenation**: concatenating multiple CSS stylesheets and JavaScript files.
- **Minification**: removing unnecessary characters from CSS stylesheets and JavaScript files without altering processing semantics.
  - For example, removing comments and whitespace characters.

Best Practices for Reducing Latency (5)

- **CSS minifier tools**:
  - cssnano (written in: PostCSS; license: MIT License) https://cssnano.github.io/cssnano/ https://github.com/cssnano/cssnano
    - Online: https://cssnano.github.io/cssnano/playground/
  - CSSO (CSS Optimizer) (written in: JavaScript; license: MIT License) https://github.com/css/csso
    - Online: https://css.github.io/csso/csso.html
- **JavaScript minifier tools**:
  - Closure Compiler (written in: Java, JavaScript; license: Apache License 2.0) https://developers.google.com/closure/compiler/ https://github.com/google/closure-compiler
  - UglifyJS (written in: JavaScript; license: Simplified BSD License) http://lisperator.net/uglifyjs/ https://github.com/mishoo/UglifyJS2
    - Online: JSCompress https://jscompress.com/
- **Complex minifier solutions**:
  - PageSpeed Modules (Apache PageSpeed) (written in: C++; license: Apache License 2.0) https://www.modpagespeed.com/ https://developers.google.com/speed/pagespeed/module/ https://github.com/we-amp/mod_pagespeed https://github.com/we-amp/ngx_pagespeed
    - Apache HTTP Server and nginx modules.

What is HTTP/2?

- An optimized expression of the semantics of HTTP.
  - Its goal is to enable a more efficient use of network resources and a reduced perception of latency.
  - A major goal is to allow clients to require only a single TCP connection to be maintained to a server.
- Uses the same methods, status codes, header fields, and URI schemes used by HTTP/1.1.
  - Messages are formatted and transmitted differently.

Development

- Is developed by the HTTP Working Group of IETF (httpbis).
  - Web page: https://http2.github.io/

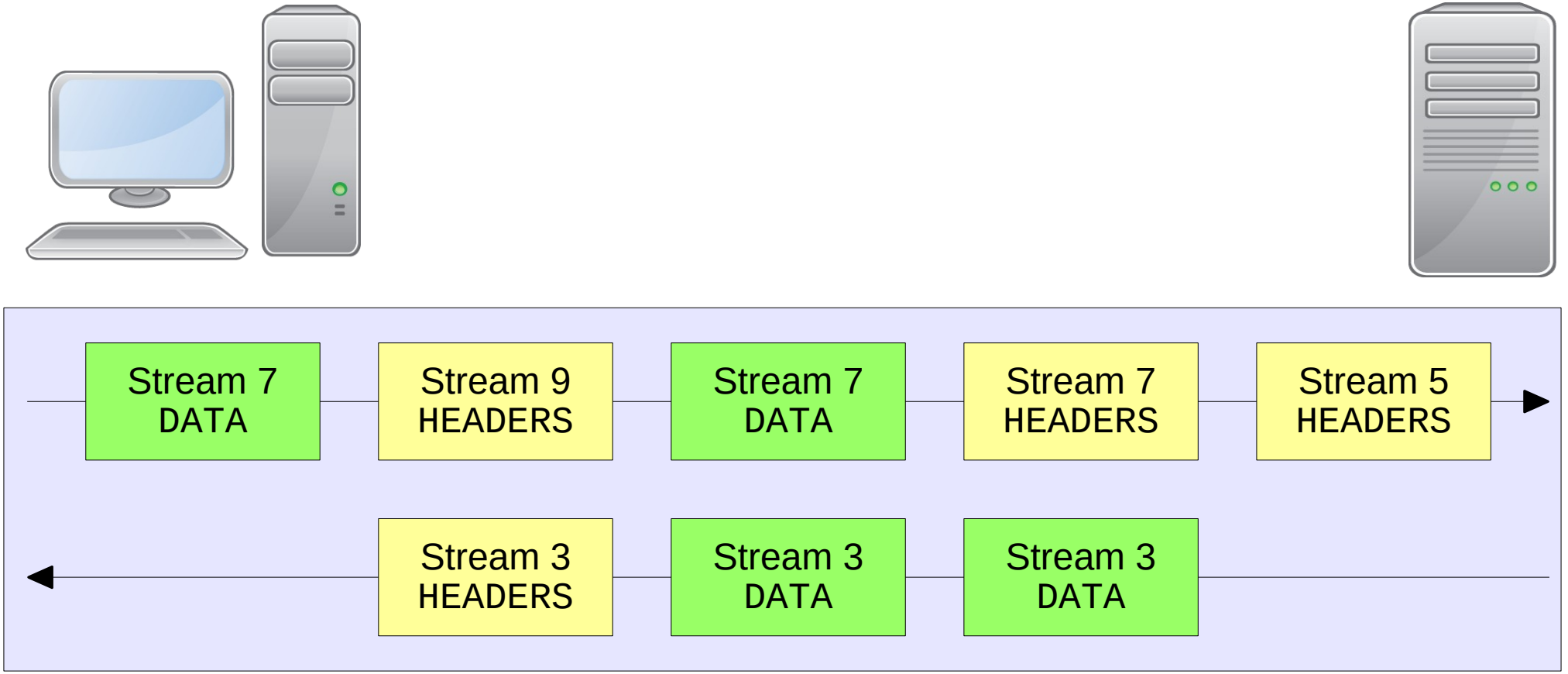
Specifications

- Martin Thomson (ed.), Cory Benfield (ed.). *RFC 9113: HTTP/2.* June 2022. https://www.rfc-editor.org/rfc/rfc9113
- Roberto Peon, Herve Ruellan. *RFC 7541: HPACK: Header Compression for HTTP/2.* May 2015. https://www.rfc-editor.org/rfc/rfc7541

New Features of HTTP/2

- **Multiplexing**: is achieved by using streams.
  - Streams are largely independent of each other, so a blocked or stalled request or response does not prevent progress on other streams.
- **Flow control and prioritization**: ensure that it is possible to efficiently use multiplexed streams.
  - Flow control helps to ensure that only data that can be used by a receiver is transmitted.
  - Prioritization ensures that limited resources can be directed to the most important streams first.
- **Server push**: allows a server to speculatively send data to a client that the server anticipates the client will need.
- **Binary protocol**: enables more efficient processing of messages through use of binary message framing.
- **Header compression** (HPACK)

Message Multiplexing



Preliminaries

- HTTP/2 is based on the SPDY protocol developed by Google. https://www.chromium.org/spdy/
  - Objectives: *SPDY: An experimental protocol for a faster web* https://www.chromium.org/spdy/spdy-whitepaper/
    - Objectives included targeting a 50% reduction in page load time.
  - SPDY/2 served as the basis for the HTTP/2 specification.

Adoption (1)

- HTTP/2-enabled websites:
  - Dropbox https://www.dropbox.com/
  - Facebook https://www.facebook.com/
  - Flickr https://www.flickr.com/
  - Google https://www.google.com/
  - Twitter https://twitter.com/
  - Wikipedia https://www.wikipedia.org/
  - W3C https://www.w3.org/
  - Yahoo https://www.yahoo.com/
  - …

Adoption (2)

- HTTP2.Pro https://http2.pro/
  - Online tool to check server HTTP/2 support.

Adoption (2)

- Adoption statistics:
  - *HTTP Archive – State of the Web – HTTP/2 Requests*
    https://httparchive.org/reports/state-of-the-web#h2
  - *Usage of HTTP/2 for websites*
    https://w3techs.com/technologies/details/ce-http2/all/all

Implementations (1)

- A list of implementations: https://github.com/http2/http2-spec/wiki/Implementations

Implementations (2)

- Servers:
  - Apache HTTP Server (written in: C; license: Apache License 2.0) https://httpd.apache.org/
    - The `mod_http2` module introduced in version 2.4.17 provides HTTP/2 support: *Overview of new features in Apache HTTP Server 2.4* https://httpd.apache.org/docs/trunk/new_features_2_4.html
  - Apache Tomcat (written in: Java; license: Apache License 2.0) https://tomcat.apache.org/
    - HTTP/2 support is available in versions 8.5.*x* and 9.*x*, see: https://tomcat.apache.org/whichversion.html
  - Jetty (written in: Java; license: Apache License 2.0/Eclipse Public License v1.0) https://jetty.org/ https://github.com/eclipse/jetty.project
    - HTTP/2 is supported from version `9.3.0.M2` available from the Maven Central Repository.
  - Microsoft IIS (written in: C++; license: non-free) https://www.iis.net/
    - HTTP/2 support was introduced in IIS 10.0: https://learn.microsoft.com/en-us/iis/get-started/whats-new-in-iis-10/http2-on-iis
  - nginx (written in: C; license: Simplified BSD License) https://nginx.org/
    - *Faisal Memon, NGINX Open Source 1.9.5 Released with HTTP/2 Support*, September 22, 2015. https://blog.nginx.org/blog/nginx-1-9-5
  - Undertow (written in: Java; license: GPLv2.1) https://undertow.io/ https://github.com/undertow-io/undertow
    - Is the default web server in the Wildfly Application Server (formerly JBoss Application Server). https://www.wildfly.org/

Implementations (3)

- Web browsers: any of the browsers mentioned below supports HTTP/2 over TLS only (i.e., for `https` URIs only).
  - **Firefox**:
    - See the `network.http.spdy.enabled.http2` option (`about:config`).
  - **Chromium/Google Chrome/Opera**:
    - See: https://chromestatus.com/feature/5125863656652 80
- See: https://caniuse.com/http2
- In web browsers, the protocol version used to obtain a resource can be seen on the Network tab of DevTools.

Implementations (4)

- Other clients:
  - curl (written in: C; license: X11 License) https://curl.se/ https://github.com/curl/curl
  - hyper-h2 (written in: Python; license: MIT License) http://python-hyper.org/projects/h2/en/stable/ https://github.com/python-hyper/h2
  - h2i (written in: Go; license: New BSD License) https://github.com/golang/net/tree/master/http2/h2i
  - Netty (written in: Java; license: Apache License 2.0) https://netty.io/ https://github.com/netty/netty
  - nghttp2 (written in: C; license: MIT License) https://nghttp2.org/ https://github.com/nghttp2/nghttp2
  - OkHttp (written in: Java; license: Apache License 2.0) https://square.github.io/okhttp/ https://github.com/square/okhttp

Browser Add-ons

- Browser add-ons indicating the HTTP version used in the address bar:
  - **Firefox**: *HTTP Version Indicator*
    https://addons.mozilla.org/hu/firefox/addon/http2-indicator/
    https://github.com/bsiegel/http-version-indicator
  - **Chromium**, **Google Chrome**: *HTTP Indicator*
    https://chrome.google.com/webstore/detail/http-indicator/hgcomhbcacfkpffiphlmnlhpppcjgmbl
    https://github.com/pd4d10/http-indicator

curl

- HTTP/2 support is implemented using the nghttp2 library.
  - See: *HTTP/2 with curl* https://curl.se/docs/http2.html
- Example of use:
  - `curl --http2 --head https://www.w3.org/`
- Since 7.47.0, HTTP/2 is enabled by default for HTTPS connections.

Terminology

- **Client**: the endpoint that initiates an HTTP/2 connection, clients send HTTP requests and receive HTTP responses.
- **Server**: the endpoint that accepts an HTTP/2 connection, servers receive HTTP requests and send HTTP responses.
- **Endpoint**: either the client or server of a connection.
- **Connection**: a transport-layer connection between two endpoints.
- **Frame**: the smallest unit of communication within an HTTP/2 connection.
- **Stream**: an independent, bidirectional sequence of frames exchanged between the client and server within an HTTP/2 connection.

Establishing a HTTP/2 Connection (1)

- HTTP/2 uses the same `http` and `https` URI schemes used by HTTP/1.1.

Establishing a HTTP/2 Connection (2)

- The client initiates the TCP connection.
- The client is required to first discover whether the server supports HTTP/2. How support for HTTP/2 is determined is different for `http` and `https` URIs.
  - For an `http` URI, the client makes an HTTP/1.1 request that includes an `Upgrade` header field.
  - For an `https` URI, protocol negotiation is performed using ALPN.

Establishing a HTTP/2 Connection (3)

- Example:
  - `curl --http2 -v http://nghttp2.org/`

```
> GET / HTTP/1.1
> Host: nghttp2.org
> User-Agent: curl/8.10.1
> Accept: */*
> Connection: Upgrade, HTTP2-Settings
> Upgrade: h2c
> HTTP2-Settings: AAMAAABkAAQAAQAAAAIAAAAA
>
< HTTP/1.1 101 Switching Protocols
< Connection: Upgrade
< Upgrade: h2c
<
< HTTP/2.0 200
...
```

Establishing a HTTP/2 Connection (4)

- Example:
  - `curl --head https://www.facebook.com/`

Establishing a HTTP/2 Connection (5)

- Each endpoint is required to send a **connection preface** as a final confirmation of the protocol in use and to establish the initial settings for the HTTP/2 connection.
  - The client connection preface starts with the `"PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n"` sequence of octets that must be followed by a `SETTINGS` frame, which MAY be empty.
    - See: Matthew Kerwin, *Painting Sheds*, 2013-12-09. https://matthew.kerwin.net.au/blog/20131209_painting_sheds
  - The server connection preface consists of a potentially empty `SETTINGS` frame.
- The client sends the client connection preface immediately upon receipt of a `101` (`Switching Protocols`) response (indicating a successful upgrade) or as the first application data octets of a TLS connection.
- If starting an HTTP/2 connection with prior knowledge of server support for the protocol, the client connection preface is sent upon connection establishment.

ALPN (1)

- A Transport Layer Security (TLS) extension for application-layer protocol negotiation:
  - Stephan Friedl, Andrei Popov, Adam Langley, Emile Stephan. *RFC 7301: Transport Layer Security (TLS) – Application-Layer Protocol Negotiation Extension.* July 2014. https://www.rfc-editor.org/rfc/rfc7301
- IANA maintains the registry of protocol identifiers.
  - See: *Application-Layer Protocol Negotiation (ALPN) Protocol Ids* https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#alpn-protocol-ids

ALPN (2)

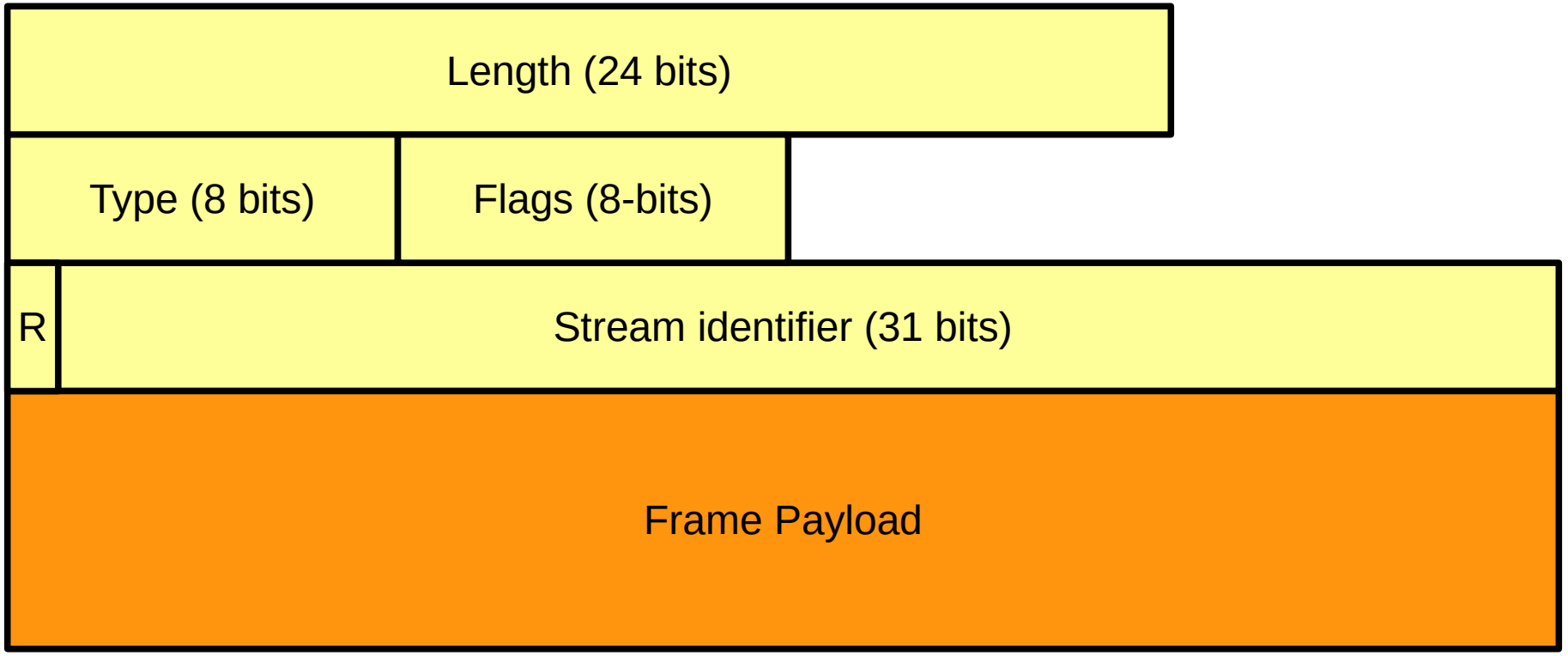
- Protocol negotiation:
  - The client sends the list of supported application protocols as part of the TLS `ClientHello` message.
  - The server chooses a protocol and sends the selected protocol as part of the TLS `ServerHello` message.
- The application protocol negotiation can thus be accomplished within the TLS handshake.

ALPN (3)

- The protocol identifiers used by HTTP/2:
  - `"h2"`: HTTP/2 over TLS
  - `"h2c"`: HTTP/2 that does not use TLS (the letter `'c'` stands for cleartext)

Frame Format (1)

- All frames begin with a fixed 9-octet header followed by a variable-length **payload**.



Frame Format (2)

- The fields of the frame header:
  - **Length**: the length of the frame payload expressed as an unsigned 24-bit integer.
  - **Type**: the 8-bit type of the frame that determines the format and semantics of the frame.
  - **Flags**: an 8-bit field reserved for boolean flags specific to the frame type.
  - **R**: a reserved 1-bit field set to 0 whose semantics is undefined.
  - **Stream identifier**: a stream identifier expressed as an unsigned 31-bit integer.
- The structure and content of the frame payload is dependent entirely on the frame type.

Frame Types (1)

Code	Type	Function
0x0	DATA	Conveys message payload.
0x1	HEADERS	Opens a stream and additionally carries a header block fragment.
0x2	PRIORITY	Specifies the priority of a stream.
0x3	RST_STREAM	Immediately terminates a stream.
0x4	SETTINGS	Conveys configuration parameters and also acknowledges the receipt of them.

Frame Types (2)

Kód	Típus	Funkció
0x5	PUSH_PROMISE	Implements server push, notifies the peer endpoint in advance of streams the sender intends to initiate.
0x6	PING	A mechanism for measuring a minimal round-trip time (RRT) from the sender, as well as determining whether an idle connection is still functional.
0x7	GOAWAY	Initiates shutdown of a connection or signals a connection error.
0x8	WINDOW_UPDATE	Implements flow control.
0x9	CONTINUATION	Continues a sequence of header block fragments.

Stream Characteristics

- A single HTTP/2 connection can contain multiple concurrently open streams.
- Streams can be established and used unilaterally or shared by either the client or server.
- Streams can be closed by either endpoint.
- The order in which frames are sent on a stream is significant.
  - Recipients process frames in the order they are received.
- Streams are identified by an unsigned integer.

Stream Identifiers (1)

- Streams are identified with an unsigned 31-bit integer that is assigned to a stream by the initiating endpoint.
  - Streams initiated by a client must use odd-numbered stream identifiers.
  - Streams initiated by the server must use even-numbered stream identifiers.
  - A stream identifier of zero is used for connection control messages.
- The identifier of a newly established stream must be numerically greater than all streams that the initiating endpoint has opened or reserved.

Stream Identifiers (2)

- Stream identifiers cannot be reused.
- Long-lived connections can result in an endpoint exhausting the available range of stream identifiers.
  - A client that is unable to establish a new stream identifier can establish a new connection for new streams.

HTTP Request/Response Exchange

- A client sends each HTTP request to a server on a new stream, using a previously unused stream identifier.
- A server sends an HTTP response on the same stream as the request.
- An HTTP request/response exchange fully consumes a single stream.
- The last frame of a response closes the stream.

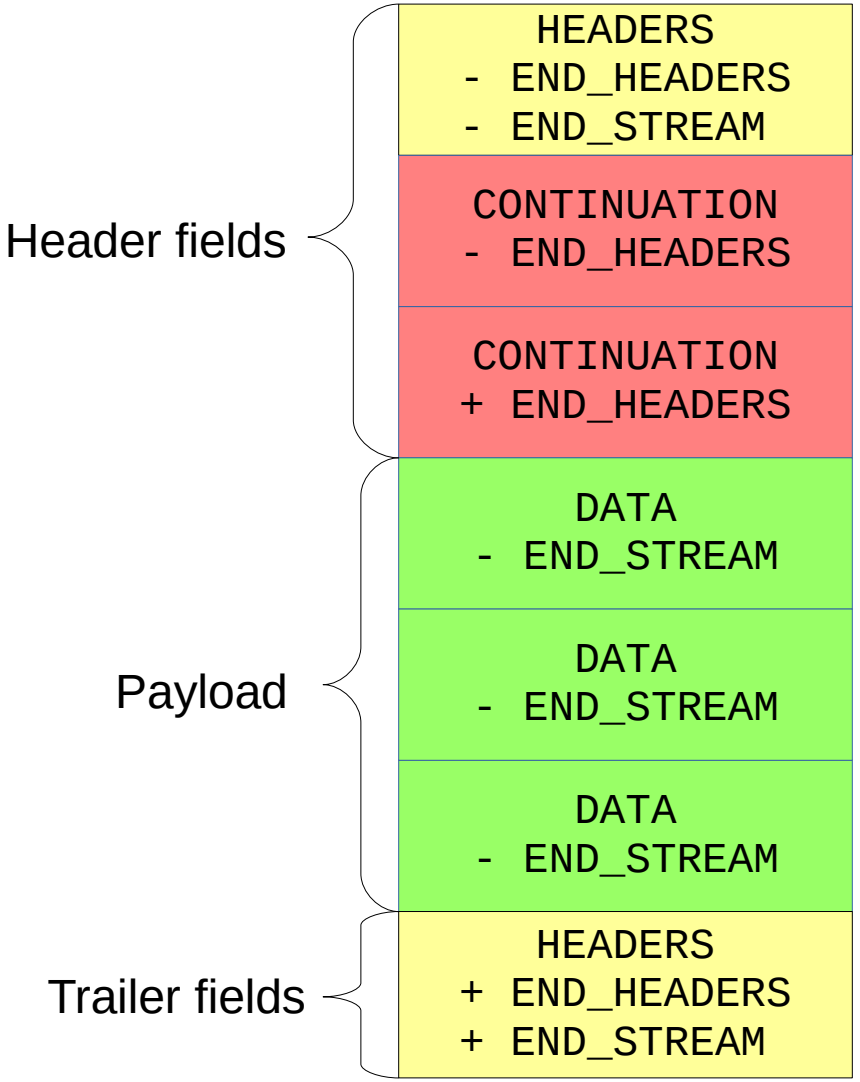
Structure of HTTP/2 Messages (1)

- An HTTP message (request or response) consists of:
  - For a response only, zero or more `HEADERS` frames (each followed by zero or more `CONTINUATION` frames) containing the message headers of informational (1*xx*) HTTP responses.
  - One `HEADERS` frame (followed by zero or more `CONTINUATION` frames) containing the message headers.
  - Zero or more `DATA` frames containing the payload body.
  - Optionally, one `HEADERS` frame, followed by zero or more `CONTINUATION` frames containing the trailer part, if present.

Structure of HTTP/2 Messages (2)

- The last frame in the sequence has the `END_STREAM` flag set.
- Other frames (from any stream) must not occur between the `HEADERS` frame and any `CONTINUATION` frames that might follow.
- The `chunked` transfer encoding defined must not be used in HTTP/2.
  - HTTP/2 uses `DATA` frames to carry message payloads, thus, the payload can be transmitted in chunks.

Structure of HTTP/2 Messages (3)



Header Fields

- Header field names must be converted to lowercase prior to their encoding in HTTP/2.

Pseudo-Header Fields (1)

- HTTP/2 uses special pseudo-header fields beginning with a `':'` character to represent information provided in the start-line of HTTP/1.*x* messages.
- Pseudo-header fields are not HTTP header fields.

Pseudo-Header Fields (2)

- Request pseudo-header fields:
  - **`:method`**: includes the HTTP method.
  - **`:scheme`**: includes the scheme portion of the target URI.
  - **`:authority`**: includes the authority portion of the target URI, corresponds to the `Host` header field.
    - Clients that generate HTTP/2 requests directly should use the `:authority` pseudo-header field instead of the `Host` header field.
  - **`:path`**: includes the path and query parts of the target URI.
    - Its value is `'*'` for server-wide `OPTIONS` requests.
- Response pseudo-header fields:
  - **`:status`**: carries the HTTP status code field, must be included in all responses.

Pseudo-Header Fields (3)

- All pseudo-header fields must appear before regular header fields.
- All HTTP/2 requests must include exactly one valid value for the `:method`, `:scheme`, and `:path` pseudo-header fields, unless it is a `CONNECT` request.
- HTTP/2 does not define a way to carry:
  - The protocol version identifier in a request.
  - The protocol version identifier and the reason phrase in a response.

Example (1)

```
> GET /index.html HTTP/1.1
> User-Agent: curl/8.10.1
> Host: eg.com
> Accept: */*
>
```

```
HEADERS (stream_id=1)
  + END_STREAM
  + END_HEADERS
  :method = GET
  :path = /index.html
  :scheme = https
  :authority = eg.com
  user-agent = curl/8.10.1
  accept = */*
```

```
< HTTP/1.1 200 OK
< ETag: "54ef4a17"
< Content-Length: 8192
< Content-Type: text/html
<
< ⟨data⟩
```

```
HEADERS (stream_id=1)
  - END_STREAM
  + END_HEADERS
  :status = 200
  etag = "54ef4a17"
  content-length = 8192
  content-type = text/html

DATA (stream_id=1)
  + END_STREAM
  ⟨data⟩
```

Example (2)

```
> GET /index.html HTTP/1.1
> User-Agent: curl/8.4.0
> Host: eg.com
> Accept: */*
> If-None-Match: "54ef4a17"
>
```

```
HEADERS (stream_id=1)
  + END_STREAM
  + END_HEADERS
  :method = GET
  :path = /index.html
  :scheme = https
  :authority = eg.com
  user-agent = curl/8.4.0
  accept = */*
  if-none-match = "54ef4a17"
```

```
< HTTP/1.1 304 Not Modified
< ETag: "54ef4a17"
<
```

```
HEADERS (stream_id=1)
  + END_STREAM
  + END_HEADERS
  :status = 304
  etag = "54ef4a17"
```

Example (3)

```
> PUT /image HTTP/1.1
> User-Agent: curl/8.4.0
> Host: eg.com
> Accept: */*
> Content-Length: 8192
> Content-Type: image/png
>
> ⟨data⟩
```

```
HEADERS (stream_id=1)
  - END_STREAM
  + END_HEADERS
  :method = PUT
  :path = /image
  :scheme = https
  :authority = eg.com
  user-agent = curl/8.4.0
  accept = */*
  content-length = 8192
  content-type = image/png

DATA (stream_id=1)
  + END_STREAM
  ⟨data⟩
```

Server Push (1)

- Allows a server to pre-emptively send (or “push”) responses (along with corresponding “promised” requests) to a client in association with a previous client-initiated request.
  - This can be useful when the server knows the client will need to have those responses available to fully process the response to the original request.
- The server does this by synthesizing a request, which it sends as a `PUSH_PROMISE` frame. The server is then able to send a response to the synthetic request on a separate stream.

Server Push (2)

- The sender uses a `PUSH_PROMISE` frame to notify the recipient that it intends to create a stream for sending an unsolicited response.
- The `PUSH_PROMISE` frame can be followed by one or more `CONTINUATION` frames.
- The `PUSH_PROMISE` and subsequent `CONTINUATION` frames together contain a complete set of request header fields that the server attributes to the synthetic request.
- The `PUSH_PROMISE` frame also includes an identifier of the stream the endpoint intends to create.

Server Push (3)

- Pushed responses are always associated with an explicit request from the client. The PUSH_PROMISE frames sent by the server are sent on that explicit request's stream.
- The server should send PUSH_PROMISE frames before sending any frames that reference the promised responses.
- After sending the PUSH_PROMISE frame, the server can begin delivering the pushed response as a response to a server-initiated stream that uses the promised stream identifier.

Server Push (4)

- A client can request that server push be disabled.
  - The SETTINGS_ENABLE_PUSH setting can be set to 0 to indicate that server push is disabled.
    - The initial value is 1, which indicates that server push is permitted.
  - An endpoint must not send a PUSH_PROMISE frame if it receives this parameter set to a value of 0.
- Recipients of PUSH_PROMISE frames can choose to reject promised streams by returning a RST_STREAM frame.

Server Push (5)

- Example:

```
HEADERS (stream_id=1)
  + END_STREAM
  + END_HEADERS
  :method = GET
  :path = /index.html
  :scheme = https
  :authority = example.com
  accept = */*
```

```
PUSH_PROMISE (stream_id=1,
  promised stream id=2)
  + END_HEADERS
  :method = GET
  :path = /style.css
  :scheme = https
  :authority = example.com
  accept = */*

HEADERS (stream_id=1)
  - END_STREAM
  + END_HEADERS
  :status = 200
  content-length = 8192
  content-type = text/html

DATA (stream_id=1)
  + END_STREAM
  ⟨data⟩
```

Server Push (6)

- Example (continued):

```
HEADERS (stream_id=2)
  - END_STREAM
  + END_HEADERS
  :status = 200
  content-length = 1024
  content-type = text/css

DATA (stream_id=2)
  + END_STREAM
  ⟨data⟩
```

Server Push (7)

- Server-side support:
  - *Apache HTTP Server*: yes
    - See: *Apache Module mod_http2 – H2Push Directive* https://httpd.apache.org/docs/current/mod/mod_http2.html#h2push
  - *Apache Tomcat*: yes
    - See: *Apache Tomcat 10 – Changelog* https://tomcat.apache.org/tomcat-10.1-doc/changelog.html
  - *Jetty*: yes
    - See: *HTTP/2 Push of Resources* https://jetty.org/docs/jetty/12/programming-guide/server/http2.html#push
  - *nginx*: yes
    - See: *Introducing HTTP/2 Server Push with NGINX 1.13.9* https://www.f5.com/company/blog/nginx/nginx-1-13-9-http2-server-push
  - *Undertow*: yes
    - See: https://undertow.io/javadoc/2.1.x/io/undertow/UndertowOptions.html#HTTP2_SETTINGS_ENABLE_PUSH

Server Push (8)

- Client-side support:
  - curl: yes
    - See: https://github.com/curl/curl/blob/master/docs/FEATURES.md
  - Hyper: yes
  - Netty: yes

Server Push (9)

- Browser support:
  - **Firefox**:
    - See the `network.http.http2.allow-push` option (`about:config`).
  - **Chromium/Google Chrome/Opera**:
    - See: Barry Pollard. *Removing HTTP/2 Server Push from Chrome*. August 18, 2022. https://developer.chrome.com/blog/removing-push/

Server Push (10)

- Practical implementation:
  - ***Apache HTTP Server***:
    - *Apache HTTP Server Version 2.4 – HTTP/2 guide – Server Push*
      https://httpd.apache.org/docs/current/howto/http2.html#push
  - ***nginx***:
    - Owen Garrett, *Introducing HTTP/2 Server Push with NGINX 1.13.9*, February 20, 2018.
      https://www.f5.com/company/blog/nginx/nginx-1-13-9-http2-server-push

Server Push (11)

- Adoption statistics:
  - HTTP Archive. *Web Almanac*. 2022.
    - *Part IV Chapter 23: HTTP—HTTP/2 Push*
      https://almanac.httparchive.org/en/2022/http#http2-push

Connection Management

- HTTP/2 connections are persistent.
- Clients should not open more than one HTTP/2 connection to a given host and port pair.

HPACK

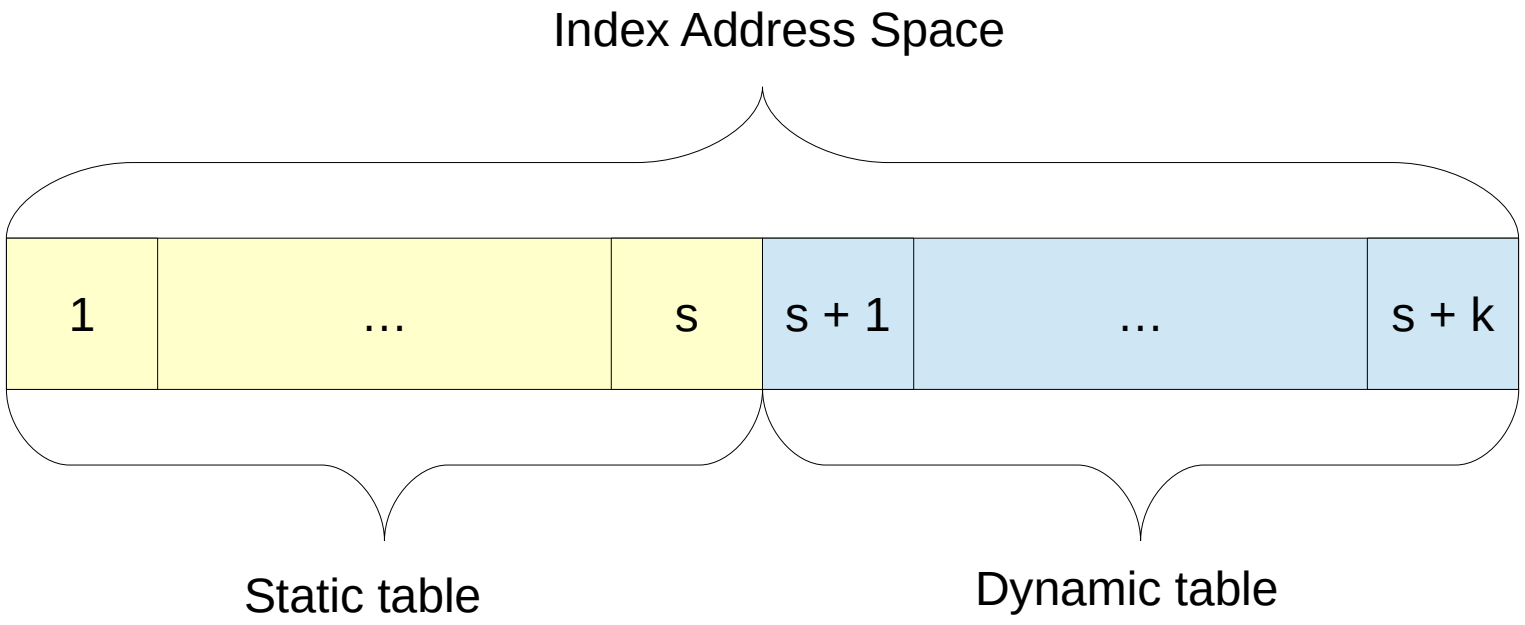
- A compression format for efficiently representing HTTP header fields to be used in HTTP/2.

Indexing Tables (1)

- Header fields are encoded using two indexing tables that associate header fields with indexes.
  - **Static table**: a read-only table predefined by the specification that statically associates header fields that occur frequently with index values.
  - **Dynamic table**: an initially empty table maintained by the encoder/decoder to index repeated header fields not present in the static table.
    - Is managed in a FIFO (first-in, first-out) manner.
    - Its maximum size can be constrained.
    - The encoding and decoding dynamic tables maintained by an endpoint are completely independent, i.e., the request and response dynamic tables are separate.

Indexing Tables (2)

- The static and dynamic tables are combined into a single address space for defining index values.



The Static Indexing Table

Index	Name	Value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
9	:status	204
…	…	…
58	user-agent	
59	vary	
60	via	
61	www-authenticate	

Header Field Representation

- A header field can be represented in two ways:
  - **As an index**: identifying an entry in either the static table or the dynamic table.
  - **Literally**: providing the name and the value of the header field.
    - The header field name is provided either as a literal or by reference to an existing table entry, either from the static table or the dynamic table.
    - The header field value is provided as a string literal.
- Literal values (i.e., field names and field values) are either encoded directly or use a static Huffman code.

Huffman Encoding

- RFC 7541 specifies a static Huffman code.

Decimal Value	Character	Code	Length (bits)
…	…	…	…
32	' '	|010100	6
33	'!'	|11111110|00	10
34	'"'	|11111110|01	10
35	'#'	|11111111|1010	12
36	'$'	|11111111|11001	13
37	'%'	|010101	6
38	'&'	|11111000	8
39	'''	|11111111|010	11
…	…	…	…

Example (1)

Index	Name	Value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
9	:status	204
…	…	…
58	user-agent	
59	vary	
60	via	
61	www-authenticate	

:method	GET
:scheme	https
:path	/index.html
:authority	www.example.com
user-agent	my-user-agent
accept	*/*

2	
7	
5	
1	Huffman("www.example.com")
58	Huffman("my-user-agent")
19	Huffman("*/*")

Example (2)

- Size of HTTP/1.1 request: 91 octets
- Size of equivalent HTTP/2 request: 9 + 6 + 34 = 49 octets

```
> GET /index.html HTTP/1.1
> Host: www.example.com
> User-Agent: my-user-agent
> Accept: */*
>
```

```
HEADERS
  + END_STREAM
  + END_HEADERS
  :method = GET
  :scheme = http
  :path = /index.html
  :authority = example.com
  user-agent: my-user-agent
  accept = */*
```

Example (3)

```
< HTTP/1.1 200 OK
< Date: Wed, 03 Oct 2018 11:45:09 GMT
< Last-Modified: Wed, 03 Oct 2018 11:45:09 GMT
< Content-Length: 4096
< Cache-Control: public, max-age=300
< Expires: Wed, 03 Oct 2018 11:50:09 GMT
< Content-Type: text/html
<
```

```
HEADERS
  - END_STREAM
  + END_HEADERS
  :status = 200
  date: Wed, 03 Oct 2018 11:45:09 GMT
  last-modified: Wed, 03 Oct 2018 11:45:09 GMT
  content-length: 4096
  cache-control: public, max-age=300
  expires: Wed, 03 Oct 2018 11:50:09 GMT
  content-type: text/html
```

- Size of status line and header fields: 225 octets
- Size of equivalent HEADERS frame: 9 + 6 + 103 = 118 octets

Performance

- Examples:
  - *HTTP/2 Technology Demo* http://www.http2demo.io/

Further Recommended Reading

- Daniel Stenberg. *http2 explained*. https://daniel.haxx.se/http2/ https://github.com/bagder/http2-explained
- Ilya Grigorik. *High Performance Browser Networking*. O'Reilly, 2013. https://hpbn.co/
- Jeremy Wagner. *A Comprehensive Guide To HTTP/2 Server Push.* April 10, 2017. https://www.smashingmagazine.com/2017/04/guide-http2-server-push/