http2 explained

Daniel Stenberg

Table of Contents

http2 explained

This is a detailed document describing HTTP/2 (RFC 7540), the background, concepts, protocol and something about existing implementations and what the future might hold.

See http://daniel.haxx.se/http2/ for the canonical home for this project.

See https://github.com/bagder/http2-explained for the source code of all book contents.

CONTRIBUTING

I encourage and welcome help and contributions from anyone who may have improvements to offer. We accept pull requests, but you can also just file issues or send email to daniel-http2@haxx.se with your suggestions!

/ Daniel Stenberg

1. Background

This document describes http2 from a technical and protocol level. It started out as a presentation Daniel did in Stockholm in April 2014 that was subsequently converted and extended into a full-blown document with all details and proper explanations.

RFC 7540 is the official name of the final http2 specification and it was published on May 15th 2015: http://www.rfc-editor.org/rfc/rfc7540.txt

All and any errors in this document are my own and the results of my shortcomings. Please point them out and they will be fixed in updated versions.

In this document I've tried to consistently use the word "http2" to describe the new protocol while in pure technical terms, the proper name is HTTP/2. I made this choice for the sake of readability and to get a better flow in the language.

1.1 Author

My name is Daniel Stenberg and I work for Mozilla. I've been working with open source and networking for over twenty years in numerous projects. Possibly I'm best known for being the lead developer of curl and libcurl. I've been involved in the IETF HTTPbis working group for several years and there I've kept up-to-date with the refreshed HTTP 1.1 work as well as being involved in the http2 standardization work.

Email: daniel@haxx.se

Twitter: @bagder

Web: daniel.haxx.se

Blog: daniel.haxx.se/blog

1.2 Help!

If you find mistakes, omissions, errors or blatant lies in this document, please send me a refreshed version of the affected paragraph and I'll make amended versions. I will give proper credits to everyone who helps out! I hope to make this document better over time.

This document is available at http://daniel.haxx.se/http2

1.3 License

This document is licensed under the Creative Commons Attribution 4.0 license: http://creativecommons.org/licenses/by/4.0/

1.4 Document history

The first version of this document was published on April 25th 2014. Here follows the largest changes in the most recent document versions.

Version 1.13

- Converted the master version of this document to Markdown syntax
- 13: Mention more resources, updated links and descriptions
- 12: Updated the QUIC description with reference to draft
- 8.5: Refreshed with current numbers
- 3.4: The average is now 40 TCP connections
- 6.4: Updated to reflect what the spec says

Version 1.12

- 1.1: HTTP/2 is now in an official RFC
- 6.5.1: Link to the HPACK RFC
- 9.1: Mention the Firefox 36+ config switch for http2
- 12.1: Added section about QUIC

Version 1.11

- Lots of language improvements mostly pointed out by friendly contributors
- 8.3.1: Mention nginx and Apache httpd specific acitivities

Version 1.10

- 1: The protocol has been “okayed”
- 4.1: Refreshed the wording since 2014 is last year
- Front: Added image and call it “http2 explained” there, fixed link
- 1.4: Added document history section
- Many spelling and grammar mistakes corrected
- 14: Added thanks to bug reporters
- 2.4: Better labels for the HTTP growth graph
- 6.3: Corrected the wagon order in the multiplexed train
- 6.5.1: HPACK draft-12

Version 1.9

- Updated to HTTP/2 draft-17 and HPACK draft-11
- Added section "10. http2 in Chromium" (== one page longer now)
- Lots of spell fixes
- At 30 implementations now
- 8.5: Added some current usage numbers
- 8.3: Mention internet explorer too
- 8.3.1 Added "missing implementations"
- 8.4.3: Mention that TLS also increases success rate

2. HTTP today

HTTP 1.1 has turned into a protocol used for virtually everything on the Internet. Huge investments have been made in protocols and infrastructure that take advantage of this, to the extent that it is often easier today to make things run on top of HTTP rather than building something new on its own.

2.1 HTTP 1.1 is huge

When HTTP was created and thrown out into the world it was probably perceived as a rather simple and straightforward protocol, but time has proved that to be false. HTTP 1.0 in RFC 1945 is a 60-page specification released in 1996. RFC 2616 that describes HTTP 1.1 was released only three years later in 1999 and had grown significantly to 176 pages. Yet, when we within IETF worked on the update to that spec, it was split up and converted into six documents, with a much larger page count in total (resulting in RFC 7230 and family). By any count, HTTP 1.1 is big and includes a myriad of details, subtleties and not the least a lot of optional parts.

2.2 A world of options

HTTP 1.1's nature of having lots of tiny details and options available for later extensions has grown a software ecosystem where almost no implementation ever implements everything – and it isn't even really possible to exactly tell what “everything” is. This has led to a situation where features that were initially little used saw very few implementations and those who did implement the features then saw very little use of them.

Later on, this caused an interoperability problem when clients and servers started to increase the use of such features. HTTP Pipelining is a primary example of such a feature.

2.3 Inadequate use of TCP

HTTP 1.1 has a hard time really taking full advantage of all the power and performance that TCP offers. HTTP clients and browsers have to be very creative to find solutions that decrease page load times.

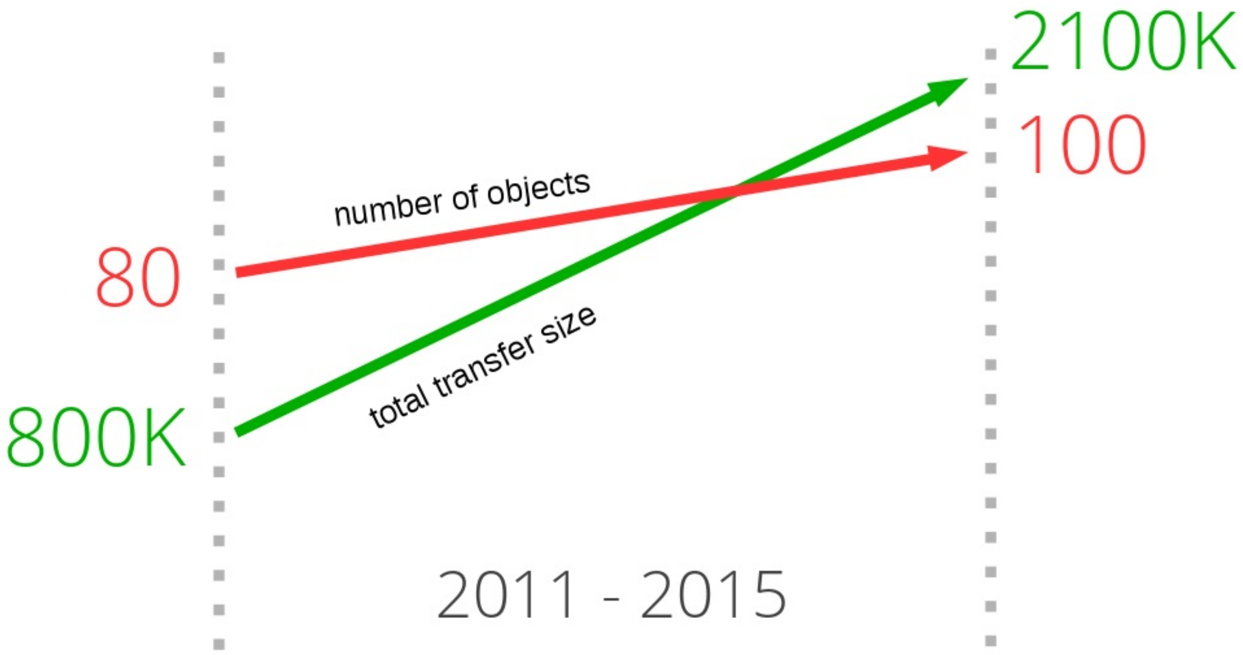
Other attempts that have been going on in parallel over the years have also confirmed that TCP is not that easy to replace and thus we keep working on improving both TCP and the protocols on top of it.

Simply put, TCP can be utilized better to avoid pauses or wasted intervals that could have been used to send or receive more data. The following sections will highlight some of these shortcomings.

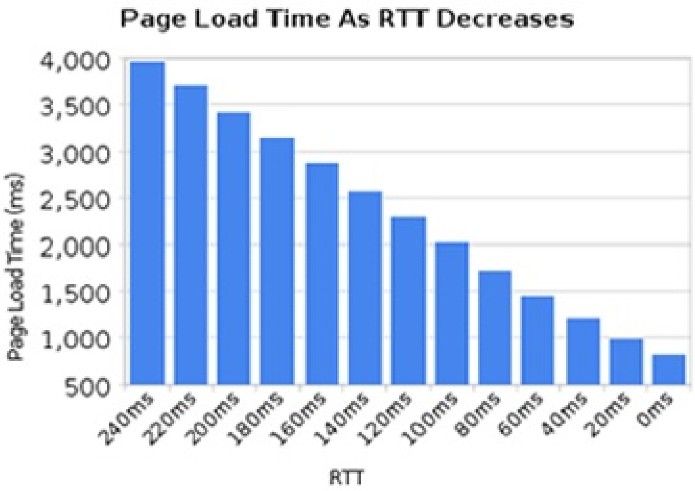
2.4 Transfer sizes and number of objects

When looking at the trend for some of the most popular sites on the web today and what it takes to download their front pages, a clear pattern emerges. Over the years the amount of data that needs to be retrieved has gradually risen up to and above 1.9MB . What is more important in this context is that on average over a hundred individual resources are required to display each page.

As the graph below shows, the trend has been going on for a while and there is little to no indication that it will change anytime soon. It shows the growth of the total transfer size (in green) and the total number of requests used on average (in red) to serve the most popular web sites in the world, and how they have changed over the last four years.



2.5 Latency kills



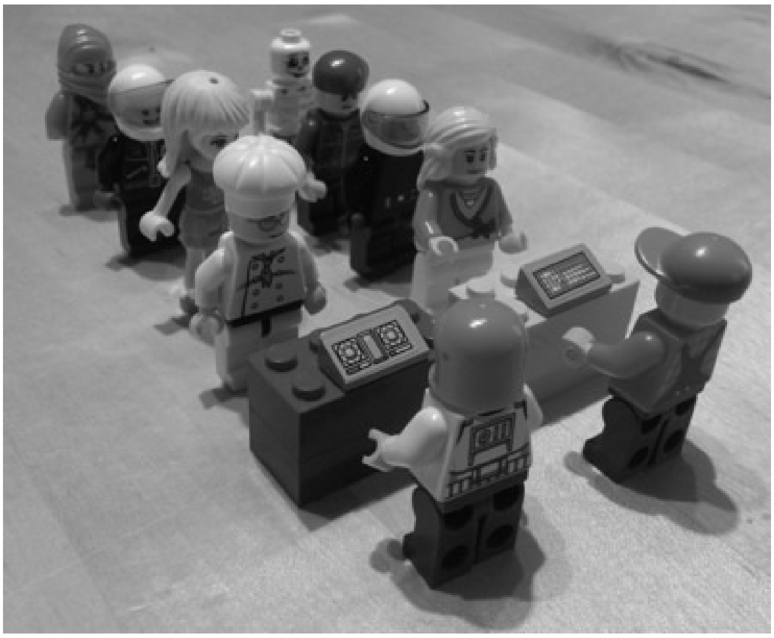
HTTP 1.1 is very latency sensitive, partly because HTTP Pipelining is still riddled with enough problems to remain switched off to a large percentage of users.

While we've seen a great increase in available bandwidth to people over the last few years, we have not seen the same level of improvements in reducing latency. High latency links, like many of the current mobile technologies, make it really hard to get a good and fast web experience even if you have a really high bandwidth connection.

Another use case that really needs low latency is certain kinds of video, like video conferencing, gaming and similar where there's not just a pre-generated stream to send out.

2.6. Head of line blocking

HTTP Pipelining is a way to send another request while waiting for the response to a previous request. It is very similar to queuing at a counter at the bank or in a super market. You just don't know if the person in front of you is a quick customer or that annoying one that will take forever before he/she is done: head of line blocking.



Sure you can be careful about line picking so that you pick the one you really believe is the correct one, and at times you can even start a new line of your own but in the end you can't avoid making a decision and once it is made you cannot switch lines.

Creating a new line is also associated with a performance and resource penalty so that's not scalable beyond a smaller number of lines. There's just no perfect solution to this.

Even today, 2015, most desktop web browsers ship with HTTP pipelining disabled by default.

Additional reading on this subject can be found for example in the Firefox bugzilla entry 264354.

3. Things done to overcome latency pains

As always when faced with problems, people gather to find workarounds. Some of the workarounds are clever and useful, some of them are just awful kludges.

3.1 Spriting

Spriting is the term often used to describe when you put a lot of small images together into a single large image. Then you use javascript or CSS to “cut out” pieces of that big image to show smaller individual ones.

A site would use this trick for speed. Getting a single big image is much faster in HTTP 1.1 than getting a 100 smaller individual ones.

Of course this has its downsides for the pages of the site that only want to show one or two of the small pictures and similar. It also makes all pictures get evicted from the cache at the same time instead of possibly letting the most commonly used ones remain.

3.2 Inlining

Inlining is another trick to avoid sending individual images, and this is done by using data: URLs embedded in the CSS file. This has similar benefits and drawbacks as the spriting case.

```
.icon1 {
    background: url(data:image/png;base64,<data>) no-repeat;
}

.icon2 {
    background: url(data:image/png;base64,<data>) no-repeat;
}
```

3.3 Concatenation

A big site can end up with a lot of different javascript files. Front-end tools will help developers merge every one of them into a single huge lump so that the browser will get a single big one instead of dozens of smaller files. Too much data is sent when only little is needed. Too much data needs to be reloaded when a change is needed.

This practice is of course mostly an inconvenience to the developers involved.

3.4 Sharding

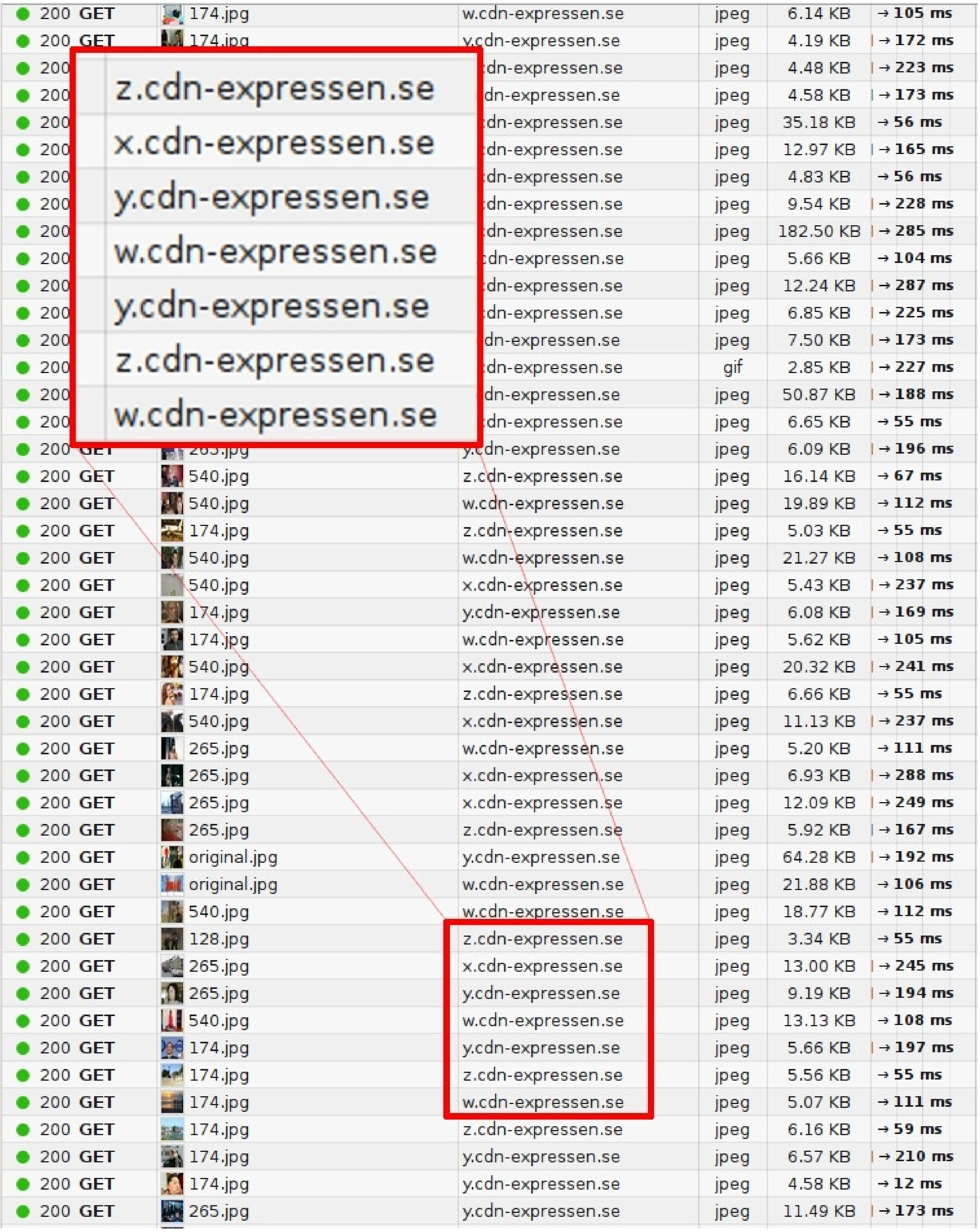
The final performance trick I'll mention is often referred to as “sharding”. It basically means serving aspects of your service on as many different hosts as possible. At first glance this seems strange but there is sound reasoning behind it.

Initially the HTTP 1.1 specification stated that a client was allowed to use maximum of two TCP connections for each host. So, in order to not violate the spec clever sites simply invented new host names and – voilá - you could get more connections to your site and decreased page load times.

Over time, that limitation was removed and today clients easily use 6-8 connections per host name but they still have a limit so sites continue to use this technique to bump the number of connections. As the number of objects are ever increasing – as I showed before – the large number of connections are then used just to make sure HTTP performs well and makes your site fast. It is not unusual for sites to use well over 50 or even up to 100 or more connections now for a single site using this technique. Recent stats from httparchive.org show that the top 300K URLs in the world need on average 40(!) TCP connections to display the site, and the trend says this is still increasing slowly over time.

Another reason is also to put images or similar resources on a separate host name that doesn't use any cookies, as the size of cookies these days can be quite significant. By using cookie-free image hosts you can sometimes increase performance simply by allowing much smaller HTTP requests!

The picture below shows how a packet trace looks like when browsing one of Sweden's top web sites and how requests are distributed over several host names.



z.cdn-expressen.se
x.cdn-expressen.se
y.cdn-expressen.se
w.cdn-expressen.se
y.cdn-expressen.se
z.cdn-expressen.se
w.cdn-expressen.se

● 200	GET	174.jpg	w.cdn-expressen.se	jpeg	6.14 KB	→ 105 ms
● 200	GET	174.jpg	y.cdn-expressen.se	jpeg	4.19 KB	→ 172 ms
● 200			dn-expressen.se	jpeg	4.48 KB	→ 223 ms
● 200			dn-expressen.se	jpeg	4.58 KB	→ 173 ms
● 200			dn-expressen.se	jpeg	35.18 KB	→ 56 ms
● 200			dn-expressen.se	jpeg	12.97 KB	→ 165 ms
● 200			dn-expressen.se	jpeg	4.83 KB	→ 56 ms
● 200			dn-expressen.se	jpeg	9.54 KB	→ 228 ms
● 200			dn-expressen.se	jpeg	182.50 KB	→ 285 ms
● 200			dn-expressen.se	jpeg	5.66 KB	→ 104 ms
● 200			dn-expressen.se	jpeg	12.24 KB	→ 287 ms
● 200			dn-expressen.se	jpeg	6.85 KB	→ 225 ms
● 200			dn-expressen.se	jpeg	7.50 KB	→ 173 ms
● 200			dn-expressen.se	gif	2.85 KB	→ 227 ms
● 200			dn-expressen.se	jpeg	50.87 KB	→ 188 ms
● 200			dn-expressen.se	jpeg	6.65 KB	→ 55 ms
● 200	GET	265.jpg	y.cdn-expressen.se	jpeg	6.09 KB	→ 196 ms
● 200	GET	540.jpg	z.cdn-expressen.se	jpeg	16.14 KB	→ 67 ms
● 200	GET	540.jpg	w.cdn-expressen.se	jpeg	19.89 KB	→ 112 ms
● 200	GET	174.jpg	z.cdn-expressen.se	jpeg	5.03 KB	→ 55 ms
● 200	GET	540.jpg	w.cdn-expressen.se	jpeg	21.27 KB	→ 108 ms
● 200	GET	540.jpg	x.cdn-expressen.se	jpeg	5.43 KB	→ 237 ms
● 200	GET	174.jpg	y.cdn-expressen.se	jpeg	6.08 KB	→ 169 ms
● 200	GET	174.jpg	w.cdn-expressen.se	jpeg	5.62 KB	→ 105 ms
● 200	GET	540.jpg	x.cdn-expressen.se	jpeg	20.32 KB	→ 241 ms
● 200	GET	174.jpg	z.cdn-expressen.se	jpeg	6.66 KB	→ 55 ms
● 200	GET	540.jpg	x.cdn-expressen.se	jpeg	11.13 KB	→ 237 ms
● 200	GET	265.jpg	w.cdn-expressen.se	jpeg	5.20 KB	→ 111 ms
● 200	GET	265.jpg	x.cdn-expressen.se	jpeg	6.93 KB	→ 288 ms
● 200	GET	265.jpg	x.cdn-expressen.se	jpeg	12.09 KB	→ 249 ms
● 200	GET	265.jpg	z.cdn-expressen.se	jpeg	5.92 KB	→ 167 ms
● 200	GET	original.jpg	y.cdn-expressen.se	jpeg	64.28 KB	→ 192 ms
● 200	GET	original.jpg	w.cdn-expressen.se	jpeg	21.88 KB	→ 106 ms
● 200	GET	540.jpg	w.cdn-expressen.se	jpeg	18.77 KB	→ 112 ms
● 200	GET	128.jpg	z.cdn-expressen.se	jpeg	3.34 KB	→ 55 ms
● 200	GET	265.jpg	x.cdn-expressen.se	jpeg	13.00 KB	→ 245 ms
● 200	GET	265.jpg	y.cdn-expressen.se	jpeg	9.19 KB	→ 194 ms
● 200	GET	540.jpg	w.cdn-expressen.se	jpeg	13.13 KB	→ 108 ms
● 200	GET	174.jpg	y.cdn-expressen.se	jpeg	5.66 KB	→ 197 ms
● 200	GET	174.jpg	z.cdn-expressen.se	jpeg	5.56 KB	→ 55 ms
● 200	GET	174.jpg	w.cdn-expressen.se	jpeg	5.07 KB	→ 111 ms
● 200	GET	174.jpg	z.cdn-expressen.se	jpeg	6.16 KB	→ 59 ms
● 200	GET	174.jpg	y.cdn-expressen.se	jpeg	6.57 KB	→ 210 ms
● 200	GET	174.jpg	y.cdn-expressen.se	jpeg	4.58 KB	→ 12 ms
● 200	GET	265.jpg	y.cdn-expressen.se	jpeg	11.49 KB	→ 173 ms

4. Updating HTTP

Wouldn't it be nice to make an improved protocol? It would...

1. Be less latency sensitive
2. Fix pipelining and the head of line blocking problem
3. Eliminate the need to keep increasing the number of connections to each host
4. Keep all existing interfaces, all content, the URI formats and schemes
5. Be made within the IETF's HTTPbis working group

4.1. IETF and the HTTPbis working group

The Internet Engineering Task Force (IETF) is an organization that develops and promotes internet standards, mostly on the protocol level. They're widely known for the RFC series of memos documenting everything from TCP, DNS, and FTP, to best practices, HTTP, and numerous protocol variants that never went anywhere.

Within IETF, dedicated “working groups” are formed with a limited scope to work toward a goal. They establish a “charter” with some set guidelines and limitations for what they should produce. Everyone and anyone is allowed to join in the discussions and development. Everyone who attends and says something has the same weight and chance to affect the outcome and everyone is counted as an individual, with little regard to which company they work for.

The HTTPbis working group (see later for an explanation of the name) was formed during the summer of 2007 and tasked with creating an update of the HTTP 1.1 specification. Within this group the discussions about a next-version HTTP really started during late 2012. The HTTP 1.1 updating work was completed early 2014 and resulted in the RFC 7230 series.

The final inter-op meeting for the HTTPbis WG was held in New York City in the beginning of June 2014. The remaining discussions and the IETF procedures performed to actually get the official RFC out continued into the following year.

Some of the bigger players in the HTTP field have been missing from the working group discussions and meetings. I don't want to mention any particular company or product names here, but clearly some actors on the Internet today seem to be confident that IETF will do good without these companies being involved...

4.1.1. The "bis" part of the name

The group is named HTTPbis where the "bis" part comes from the Latin adverb for two. Bis is commonly used as a suffix or part of the name within the IETF for an update or the second take on a spec; in this case, the update to HTTP 1.1.

4.2. http2 started from SPDY

SPDY is a protocol that was developed and spearheaded by Google. They certainly developed it in the open and invited everyone to participate but it was obvious that they benefited by being in control over both a popular browser implementation and a significant server population with well-used services.

When the HTTPbis group decided it was time to start working on http2, SPDY had already proven that it was a working concept. It had shown it was possible to deploy on the Internet and there were published numbers that proved how well it performed. The http2 work began with the SPDY/3 draft that was basically made into the http2 draft-00 with a little search and replace.

5. http2 concepts

So what does http2 accomplish? Where are the boundaries for what the HTTPbis group set out to do?

The boundaries were actually quite strict and put many restraints on the team's ability to innovate:

- http2 has to maintain HTTP paradigms. It is still a protocol where the client sends requests to the server over TCP.
- http:// and https:// URLs cannot be changed. There can be no new scheme for this. The amount of content using such URLs is too big to expect them to change.
- HTTP1 servers and clients will be around for decades, we need to be able to proxy them to http2 servers.
- Subsequently, proxies must be able to map http2 features to HTTP 1.1 clients one-to-one.
- Remove or reduce optional parts from the protocol. This wasn't really a requirement but more a mantra coming from SPDY and the Google team. By making sure everything is mandatory there's no way you can not implement anything now and fall into a trap later on.
- No more minor version. It was decided that clients and servers are either compatible with http2 or they are not. If a need arises to extend the protocol or modify things, then http3 will be born. There are no more minor versions in http2.

5.1. http2 for existing URI schemes

As mentioned already, the existing URI schemes cannot be modified, so http2 must use the existing ones. Since they are used for HTTP 1.x today, we obviously need a way to upgrade the protocol to http2, or otherwise ask the server to use http2 instead of older protocols.

HTTP 1.1 has a defined way to do this, namely the Upgrade: header, which allows the server to send back a response using the new protocol when getting such a request over the old protocol, at the cost of an additional round-trip.

That round-trip penalty was not something the SPDY team would accept, and since they only implemented SPDY over TLS, they developed a new TLS extension which shortcuts the negotiation significantly. Using this extension, called NPN for Next Protocol Negotiation, the server tells the client which protocols it knows and the client can then use the protocol it prefers.

5.2. http2 for https://

A lot of focus of http2 has been to make it behave properly over TLS. SPDY requires TLS and there's been a strong push for making TLS mandatory for http2, but it didn't get consensus so http2 shipped with TLS as optional. However, two prominent implementers have stated clearly that they will only implement http2 over TLS: the Mozilla Firefox lead and the Google Chrome lead, two of today's leading web browsers.

Reasons for choosing TLS-only include respect for user's privacy and early measurements showing that the new protocols have a higher success rate when done with TLS. This is because of the widespread assumption that anything that goes over port 80 is HTTP 1.1, which makes some middle-boxes interfere with or destroy traffic when any other protocols are used on that port.

The subject of mandatory TLS has caused much hand-wringing and agitated voices in mailing lists and meetings – is it good or is it evil? It is a highly controversial topic – be aware of this when you throw this question in the face of an HTTPbis participant!

Similarly, there's been a fierce and long-running debate about whether http2 should dictate a list of ciphers that should be mandatory when using TLS, or if it should perhaps blacklist a set, or if it shouldn't require anything at all from the TLS “layer” but leave that to the TLS working group. The spec ended up specifying that TLS should be at least version 1.2 and there are cipher suite restrictions.

5.3. http2 negotiation over TLS

Next Protocol Negotiation (NPN) is the protocol used to negotiate SPDY with TLS servers. As it wasn't a proper standard, it was taken through the IETF and the result was ALPN: Application Layer Protocol Negotiation. ALPN is being promoted for use by http2, while SPDY clients and servers still use NPN.

The fact that NPN existed first and ALPN has taken a while to go through standardization has led to many early http2 clients and http2 servers implementing and using both these extensions when negotiating http2. Also, NPN is what's used for SPDY and many servers offer both SPDY and http2, so supporting both NPN and ALPN on those servers makes perfect sense.

ALPN differs from NPN primarily in who decides what protocol to speak. With ALPN, the client gives the server a list of protocols in its order of preference and the server picks the one it wants, while with NPN the client makes the final choice.

5.4. http2 for http://

As previously mentioned, for plain-text HTTP 1.1 the way to negotiate http2 is by presenting the server with an Upgrade: header. If the server speaks http2 it responds with a “101 Switching” status and from then on it speaks http2 on that connection. Of course this upgrade procedure costs a full network round-trip, but the upside is that it's generally possible to keep an http2 connection alive much longer and re-use it more than a typical HTTP1 connection.

While some browser's spokespersons have stated they will not implement this means of speaking http2, the Internet Explorer team has expressed that they will, and curl already supports this.

6. The http2 protocol

Enough about the background, the history and politics behind what got us here. Let's dive into the specifics of the protocol: the bits and the concepts that make up http2.

6.1. Binary

http2 is a binary protocol.

Just let that sink in for a minute. If you've been involved in internet protocols before, chances are that you will now be instinctively reacting against this choice, marshaling your arguments that spell out how protocols based on text/ascii are superior because humans can handcraft requests over telnet and so on...

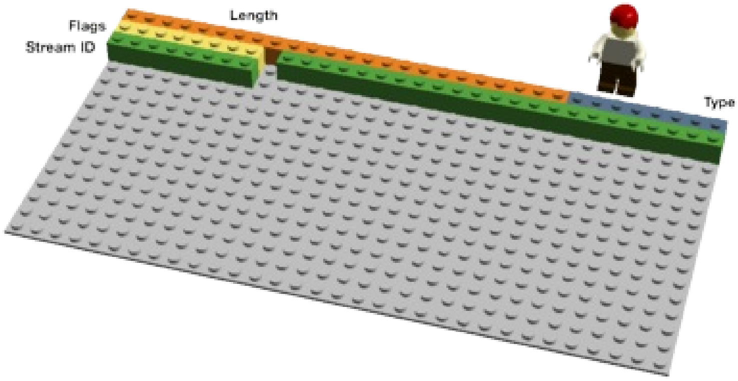
http2 is binary to make the framing much easier. Figuring out the start and the end of frames is one of the really complicated things in HTTP 1.1 and, actually, in text-based protocols in general. By moving away from optional white space and different ways to write the same thing, implementation becomes simpler.

Also, it makes it much easier to separate the actual protocol parts from the framing - which in HTTP1 is confusingly intermixed.

The fact that the protocol features compression and will often run over TLS also diminishes the value of text, since you won't see text over the wire anyway. We simply have to get used to the idea of using something like a Wireshark inspector to figure out exactly what's going on at the protocol level in http2.

Debugging this protocol will probably have to be done with tools like curl, or by analyzing the network stream with Wireshark's http2 dissector and similar.

6.2. The binary format



http2 sends binary frames. There are different frame types that can be sent and they all have the same setup: Length, Type, Flags, Stream Identifier, and frame payload.

There are ten different frame types defined in the http2 spec and perhaps the two most fundamental ones that map to HTTP 1.1 features are DATA and HEADERS. I'll describe some of the frames in more detail further on.

6.3. Multiplexed streams

The Stream Identifier mentioned in the previous section associates each frame sent over http2 with a “stream”. A stream is an independent, bi-directional sequence of frames exchanged between the client and server within an http2 connection.

A single http2 connection can contain multiple concurrently-open streams, with either endpoint interleaving frames from multiple streams. Streams can be established and used unilaterally or shared by either the client or server and they can be closed by either endpoint. The order in which frames are sent within a stream is significant. Recipients process frames in the order they are received.

Multiplexing the streams means that packages from many streams are mixed over the same connection. Two (or more) individual trains of data are made into a single one and then split up again on the other side. Here are two trains:

The two trains multiplexed over the same connection:

6.4. Priorities and dependencies

Each stream also has a priority (also known as “weight”), which is used to tell the peer which streams to consider most important, in case there are resource restraints that force the server to select which streams to send first.

Using the PRIORITY frame, a client can also tell the server which other stream this stream depends on. It allows a client to build a priority “tree” where several “child streams” may depend on the completion of “parent streams”.

The priority weights and dependencies can be changed dynamically at run-time, which should enable browsers to make sure that when users scroll down a page full of images, the browser can specify which images are most important, or if you switch tabs it can prioritize a new set of streams that suddenly come into focus.

6.5. Header compression

HTTP is a stateless protocol. In short, this means that every request needs to bring with it as much detail as the server needs to serve that request, without the server having to store a lot of info and meta-data from previous requests. Since http2 doesn't change this paradigm, it has to work the same way.

This makes HTTP repetitive. When a client asks for many resources from the same server, like images from a web page, there will be a large series of requests that all look almost identical. A series of almost identical somethings begs for compression.

While the number of objects per web page has increased (as mentioned earlier), the use of cookies and the size of the requests have also kept growing over time. Cookies also need to be included in all requests, often the same ones in multiple requests.

The HTTP 1.1 request sizes have actually gotten so large that they sometimes end up larger than the initial TCP window, which makes them very slow to send as they need a full round-trip to get an ACK back from the server before the full request has been sent. This is another argument for compression.

6.5.1. Compression is a tricky subject

HTTPS and SPDY compression were found to be vulnerable to the BREACH and CRIME attacks. By inserting known text into the stream and figuring out how that changes the output, an attacker can figure out what's being sent in an encrypted payload.

Doing compression on dynamic content for a protocol - without becoming vulnerable to one of these attacks - requires some thought and careful consideration. This is what the HTTPbis team tried to do.

Enter HPACK, Header Compression for HTTP/2, which – as the name suggests - is a compression format especially crafted for http2 headers, and it is being specified in a separate internet draft. The new format, together with other counter-measures (such as a bit that asks intermediaries to not compress a specific header and optional padding of frames), should make it harder to exploit compression.

In the words of Roberto Peon (one of the creators of HPACK):

> “HPACK was designed to make it difficult for a conforming implementation to leak information, to make encoding and decoding very fast/cheap, to provide for receiver control over compression context size, to allow for proxy re-indexing (i.e., shared state between frontend and backend within a proxy), and for quick comparisons of Huffman-encoded strings”.

6.6. Reset - change your mind

One of the drawbacks with HTTP 1.1 is that when an HTTP message has been sent off with a Content-Length of a certain size, you can't easily just stop it. Sure, you can often (but not always) disconnect the TCP connection, but that comes at the cost of having to negotiate a new TCP handshake again.

A better solution would be to just stop the message and start a new. This can be done with http2's RST_STREAM frame which will help prevent wasted bandwidth and the need to tear down connections.

6.7. Server push

This is the feature also known as “cache push”. The idea is that if the client asks for resource X, the server may know that the client will probably want resource Z as well, and sends it to the client without being asked. It helps the client by putting Z into its cache so that it will be there when it wants it.

Server push is something a client must explicitly allow the server to do. Even then, the client can swiftly terminate a pushed stream at any time with RST_STREAM should it not want a particular resource.

6.8. Flow Control

Each individual http2 stream has its own advertised flow window that the other end is allowed to send data for. If you happen to know how SSH works, this is very similar in style and spirit.

For every stream, both ends have to tell the peer that it has enough room to handle incoming data, and the other end is only allowed to send that much data until the window is extended. Only DATA frames are flow controlled.

7. Extensions

The http2 protocol mandates that a receiver must read and ignore all unknown frames (those with an unknown frame type). Two parties can negotiate the use of new frame types on a hop-by-hop basis, but those frames aren't allowed to change state and they will not be flow controlled.

The subject of whether http2 should allow extensions at all was debated at length during the protocol's development with opinions swinging for and against. After draft-12 the pendulum swung back one last time and extensions were ultimately allowed.

Extensions are not part of the actual protocol but will be documented outside of the core protocol spec. There are already two frame types that have been discussed for inclusion in the protocol that will probably be the first frames sent as extensions. I'll describe them here because of their popularity and previous state as “native” frames:

7.1. Alternative Services

With the adoption of http2, there are reasons to suspect that TCP connections will be much lengthier and be kept alive much longer than HTTP 1.x connections have been. A client should be able to do a lot of what it wants with a single connection to each host/site, and that connection could potentially be open for quite some time.

This will affect how HTTP load balancers work and there may arise situations when a site wants to suggest that the client connect to another host. It could be for performance reasons, or if a site is being taken down for maintenance, etc.

The server will send the Alt-Svc: header (or ALTSVC frame with http2) telling the client about an alternative service: another route to the same content, using another service, host, and port number.

A client should then attempt to connect to that service asynchronously and only use the alternative if the new connection succeeds.

7.1.1. Opportunistic TLS

The Alt-Svc header allows a server that provides content over http:// to inform the client that the same content is also available over a TLS connection.

This is a somewhat debatable feature. Such a connection would do unauthenticated TLS and wouldn't be advertized as “secure” anywhere, wouldn't use any padlock in the UI, and in fact there is no way to tell the user that it isn't plain old HTTP, but this is still opportunistic TLS and some people are very firmly against this concept.

7.2. Blocked

A frame of this type is meant to be sent exactly once by an http2 party when it has data to send off but flow control forbids it to send any data. The idea is that if your implementation receives this frame you know you have messed up something and/or you're getting less than perfect transfer speeds.

A quote from draft-12, before this frame was moved out to become an extension:

> “The BLOCKED frame is included in this draft version to facilitate experimentation. If the results of the experiment do not provide positive feedback, it could be removed”

8. An http2 world

So what will things look like when http2 gets adopted? Will it get adopted?

8.1. How will http2 affect ordinary humans?

http2 is not yet widely deployed nor used. We can't tell for sure exactly how things will turn out. We have seen how SPDY has been used and we can make some guesses and calculations based on that and other past and current experiments.

http2 reduces the number of necessary network round-trips and it avoids the head of line blocking dilemma completely with multiplexing and fast discarding of unwanted streams.

It allows a large amount of parallel streams that go way over even the most sharding sites of today.

With priorities used properly on the streams, chances are much better that clients will actually get the important data before the less important data. All this taken together, I'd say that the chances are very good that this will lead to faster page loads and to more responsive web sites. Shortly put: a better web experience.

How much faster and how much improvements we will see, I don't think we can say yet. First, the technology is still very early and then we haven't even started to see clients and servers trim implementations to really take advantage of all the powers this new protocol offers.

8.2. How will http2 affect web development?

Over the years web developers and web development environments have gathered a full toolbox of tricks and tools to work around problems with HTTP 1.1, recall that I outlined some of them in the beginning of this document as a justification for http2.

Lots of those workarounds that tools and developers now use by default and without thinking, will probably hurt http2 performance or at least not really take advantage of http2's new super powers. Spriting and inlining should most likely not be done with http2. Sharding will probably be detrimental to http2 as it will probably benefit from using less connections.

A problem here is of course that web sites and web developers need to develop and deploy for a world that in the short term at least, will have both HTTP1.1 and http2 clients as users and to get maximum performance for all users can be challenging without having to offer two different front-ends.

For these reasons alone, I suspect there will be some time before we will see the full potential of http2 being reached.

8.3. http2 implementations

Trying to document specific implementations in a document such as this is of course completely futile and doomed to fail and only feel outdated within a really short period of time. Instead I'll explain the situation in broader terms and refer readers to the list of implementations on the http2 web site.

There was a large amount of implementations already early on, and the amount has increased over time during the http2 work. At the time of writing this there are over 40 implementations listed, and most of them implement the final version.

8.3.1 Browsers

Firefox has been the browser that's been on top of the bleeding edge drafts, Twitter has kept up and offered its services over http2. Google started during April 2014 to offer http2 support on a few test servers running their services and since May 2014 they offer http2 support in their development versions of Chrome. Microsoft has shown a tech preview with http2 support for their next Internet Explorer version. Safari (with iOS 9 and Mac OS X El Capitan) and Opera have both said they will support http2.

8.3.2 Servers

There are already many server implementations of http2.

The popular Nginx server offers http2 support with since 1.9.5 released on September 22, 2015 (where it replaces the SPDY module, so they cannot both run in the same server instance).

Apache's httpd server has a http2 module mod_http2 since 2.4.17 which was released on October 9, 2015.

H2O, Apache Traffic Server, nghttp2, Caddy and LiteSpeed have all released http2 capable servers.

8.3.3 Others

curl and libcurl support insecure http2 as well as the TLS based one using one out of several different TLS libraries.

Wireshark supports http2. The perfect tool for analyzing http2 network traffic.

8.4. Common critiques of http2

During the development of this protocol the debate has been going back and forth and of course there is a certain amount of people who believe this protocol ended up completely wrong. I wanted to mention a few of the more common complaints and mention the arguments against them:

8.4.1. “The protocol is designed or made by Google”

It also has variations implying that the world gets even further dependent or controlled by Google by this. This isn't true. The protocol was developed within the IETF in the same manner that protocols have been developed for over 30 years. However, we all recognize and acknowledge Google's impressive work with SPDY that not only proved that it is possible to deploy a new protocol this way but also provided numbers illustrating what gains could be made.

Google has publicly announced that they will remove support for SPDY and NPN in Chrome in 2016 and they urge servers to migrate to HTTP/2 instead.

8.4.2. “The protocol is only useful for browsers”

This is sort of true. One of the primary drivers behind the http2 development is the fixing of HTTP pipelining. If your use case originally didn't have any need for pipelining then chances are http2 won't do a lot of good for you. It certainly isn't the only improvement in the protocol but a big one.

As soon as services start realizing the full power and abilities the multiplexed streams over a single connection brings, I suspect we will see more application use of http2.

Small REST APIs and simpler programmatic uses of HTTP 1.x may not find the step to http2 to offer very big benefits. But also, there should be very few downsides with http2 for most users.

8.4.3. “The protocol is only useful for big sites”

Not at all. The multiplexing capabilities will greatly help to improve the experience for high latency connections that smaller sites without wide geographical distributions often offer. Large sites are already very often faster and more distributed with shorter round-trip times to users.

8.4.4. “Its use of TLS makes it slower”

This can be true to some extent. The TLS handshake does add a little extra, but there are existing and ongoing efforts on reducing the necessary round-trips even more for TLS. The overhead for doing TLS over the wire instead of plain-text is not insignificant and clearly notable so more CPU and power will be spent on the same traffic pattern as a non-secure protocol. How much and what impact it will have is a subject of opinions and measurements. See for example istlsfastyet.com for one source of info.

Telecom and other network operators, for example in the ATIS Open Web Alliance, say that they need unencrypted traffic to offer caching, compression and other techniques necessary to provide a fast web experience over satellite, in airplanes and similar. http2 does not make TLS use mandatory so we shouldn't conflate the terms.

Many Internet users have expressed a preference for TLS to be used more widely and we should help to protect users' privacy.

Experiments have also shown that by using TLS, there is a higher degree of success than when implementing new plain-text protocols over port 80 as there are just too many middle boxes out in the world that interfere with what they would think is HTTP 1.1 if it goes over port 80 and might look like HTTP at times.

Finally, thanks to http2's multiplexed streams over a single connection, normal browser use cases still could end up doing substantially fewer TLS handshakes and thus perform faster than HTTPS would when still using HTTP 1.1.

8.4.5. “Not being ASCII is a deal-breaker”

Yes, we like being able to see protocols in the clear since it makes debugging and tracing easier. But text based protocols are also more error prone and open up for much more parsing and parsing problems.

If you really can't take a binary protocol, then you couldn't handle TLS and compression in HTTP 1.x either and its been there and used for a very long time.

8.4.6. “It isn't any faster than HTTP/1.1”

This is of course subject to debate and discussions on how to measure what faster means, but already in the SPDY days many tests were made that proved faster browser page loads (like "How Speedy is SPDY?" by people at University of Washington and "Evaluating the Performance of SPDY-enabled Web Servers" by Hervé Servy) and such experiments have been repeated with http2 as well. I'm looking forward to seeing more such tests and experiments getting published. A basic first test made by httpwatch.com might imply that HTTP/2 holds its promises.

8.4.7. “It has layering violations”

Seriously, that's your argument? Layers are not holy untouchable pillars of a global religion and if we've crossed into a few gray areas when making http2 it has been in the interest of making a good and effective protocol within the given constraints.

8.4.8. “It doesn't fix several HTTP/1.1 shortcomings”

That's true. With the specific goal of maintaining HTTP/1.1 paradigms there were several old HTTP features that had to remain. Such as the common headers that also include the often dreaded cookies, authorization headers and more. But by the upside of maintaining these paradigms is that we got a protocol that is possible to deploy without an inconceivable amount of upgrade work that requires fundamental parts to be completely replaced or rewritten. Http2 is basically just a new framing layer.

8.5. Will http2 become widely deployed?

It is too early to tell for sure, but I can still guess and estimate and that's what I'll do here.

The naysayers will say “look at how good IPv6 has done” as an example of a new protocol that's taken decades to just start to get widely deployed. http2 is not an IPv6 though. This is a protocol on top of TCP using the ordinary HTTP update mechanisms and port numbers and TLS etc. It will not require most routers or firewalls to change at all.

Google proved to the world with their SPDY work that a new protocol like this can be deployed and used by browsers and services with multiple implementations in a fairly short amount of time. While the amount of servers on the Internet that offer SPDY today is in the 1% range, the amount of data those servers deal with is much larger. Some of the absolutely most popular web sites today offer SPDY.

http2, based on the same basic paradigms as SPDY, I would say is likely to be deployed even more since it is an IETF protocol. SPDY deployment was always held back a bit by the “it is a Google protocol” stigma.

There are several big browsers behind the roll-out. Representatives from Firefox, Chrome, Safari, Internet Explorer and Opera have expressed they will ship http2 capable browsers and they have shown working implementations.

There are several big server operators that are likely to offer http2 soon, including Google, Twitter and Facebook and we hope to see http2 support soon get added to popular server implementations such as the Apache HTTP Server and nginx. H2o is a new blazingly fast HTTP server with http2 support that shows potential.

Some of the biggest proxy vendors, including HAProxy, Squid and Varnish have expressed their intentions to support http2.

All through-out 2015, the amount of traffic that is http2 has been increasing. In early September, Firefox 40 usage was at 13% out of all HTTP traffic and 27% out of all HTTPS traffic, while Google see roughly 18% incoming HTTP/2. It should be noted that Google runs other new protocol experiments as well (see QUIC in 12.1) which makes the http2 usage levels lower than it could otherwise be.

9. http2 in Firefox

Firefox has been tracking the drafts very closely and has provided http2 test implementations for many months. During the development of the http2 protocol, clients and servers have to agree on what draft version of the protocol they implement which makes it slightly annoying to run tests. Just be aware so that your client and server agree on what protocol draft they implement.

9.1. First, make sure it is enabled

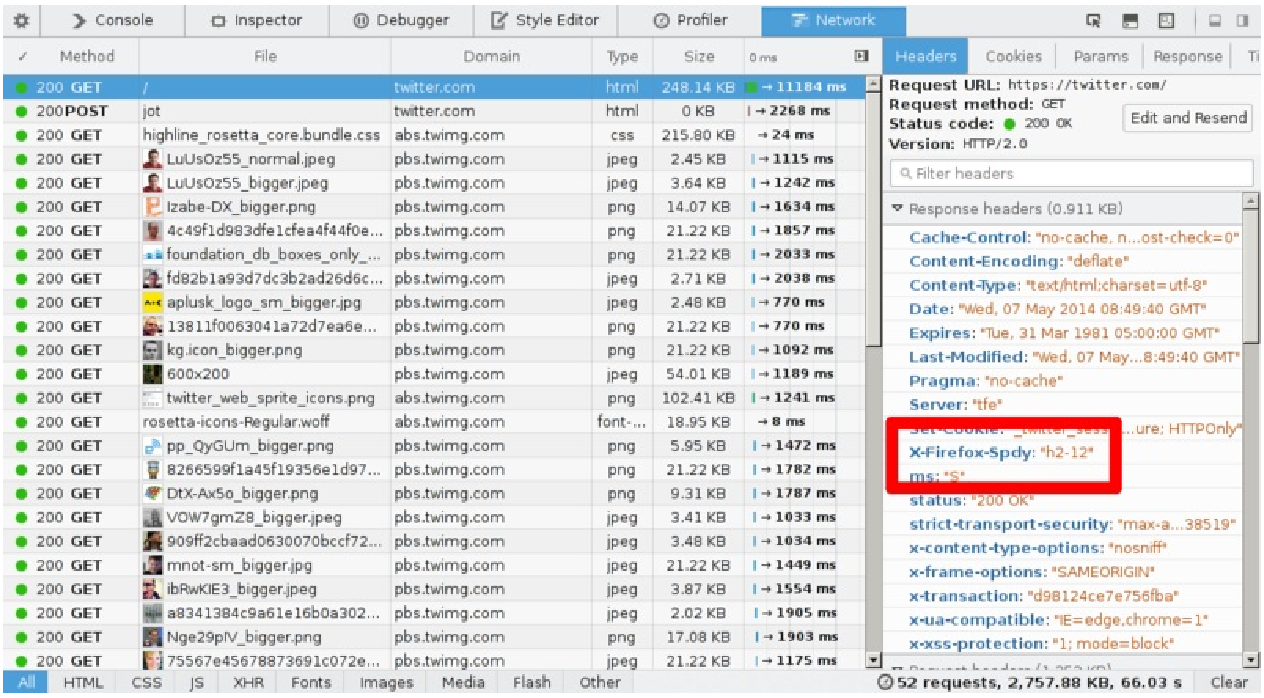
In all Firefox versions since version 35, released January 13th 2015, http2 support is enabled by default.

Enter 'about:config' in the address bar and search for the option named “network.http.spdy.enabled.http2draft”. Make sure it is set to *true*. Firefox 36 added another config switch named “network.http.spdy.enabled.http2” which is set *true* by default. The latter one controls the “plain” http2 version while the first one enables and disables negotiation of http2-draft versions. Both are true by default since Firefox 36.

9.2. TLS-only

Remember that Firefox only implements http2 over TLS. You will only ever see http2 in action with Firefox when going to https:// sites that offer http2 support.

9.3. Transparent!



✓	Method	File	Domain	Type	Size	0 ms
●	200 GET	/	twitter.com	html	248.14 KB	→ 11184 ms
●	200 POST	jot	twitter.com	html	0 KB	→ 2268 ms
●	200 GET	highline_rosetta_core.bundle.css	abs.twimg.com	css	215.80 KB	→ 24 ms
●	200 GET	LuUsOz55_normal.jpeg	pbs.twimg.com	jpeg	2.45 KB	→ 1115 ms
●	200 GET	LuUsOz55_bigger.jpeg	pbs.twimg.com	jpeg	3.64 KB	→ 1242 ms
●	200 GET	Izabe-DX_bigger.png	pbs.twimg.com	png	14.07 KB	→ 1634 ms
●	200 GET	4c49f1d983dfe1cfea4f44f0e...	pbs.twimg.com	png	21.22 KB	→ 1857 ms
●	200 GET	foundation_db_boxes_only_...	pbs.twimg.com	png	21.22 KB	→ 2033 ms
●	200 GET	fd82b1a93d7dc3b2ad26d6c...	pbs.twimg.com	jpeg	2.71 KB	→ 2038 ms
●	200 GET	aplusk_logo_sm_bigger.jpg	pbs.twimg.com	jpeg	2.48 KB	→ 770 ms
●	200 GET	13811f0063041a72d7ea6e...	pbs.twimg.com	png	21.22 KB	→ 770 ms
●	200 GET	kg.icon_bigger.png	pbs.twimg.com	png	21.22 KB	→ 1092 ms
●	200 GET	600x200	pbs.twimg.com	jpeg	54.01 KB	→ 1189 ms
●	200 GET	twitter_web_sprite_icons.png	abs.twimg.com	png	102.41 KB	→ 1241 ms
●	200 GET	rosetta-icons-Regular.woff	abs.twimg.com	font-...	18.95 KB	→ 8 ms
●	200 GET	pp_QyGUm_bigger.png	pbs.twimg.com	png	5.95 KB	→ 1472 ms
●	200 GET	8266599f1a45f19356e1d97...	pbs.twimg.com	png	21.22 KB	→ 1782 ms
●	200 GET	DtX-Ax5o_bigger.png	pbs.twimg.com	png	9.31 KB	→ 1787 ms
●	200 GET	VOW7gmZ8_bigger.jpeg	pbs.twimg.com	jpeg	3.41 KB	→ 1033 ms
●	200 GET	909ff2cbaad0630070bccf72...	pbs.twimg.com	jpeg	3.48 KB	→ 1034 ms
●	200 GET	mnot-sm_bigger.jpg	pbs.twimg.com	jpeg	21.22 KB	→ 1449 ms
●	200 GET	ibRwKIE3_bigger.jpeg	pbs.twimg.com	jpeg	3.87 KB	→ 1554 ms
●	200 GET	a8341384c9a61e16b0a302...	pbs.twimg.com	jpeg	2.02 KB	→ 1905 ms
●	200 GET	Nge29plV_bigger.png	pbs.twimg.com	png	17.08 KB	→ 1903 ms
●	200 GET	75567e45678873691c072e...	pbs.twimg.com	jpeg	21.22 KB	→ 1175 ms

```
Request URL: https://twitter.com/
Request method: GET
Status code: ● 200 OK
Version: HTTP/2.0

▾ Response headers (0.911 KB)
    Cache-Control: "no-cache, n...ost-check=0"
    Content-Encoding: "deflate"
    Content-Type: "text/html;charset=utf-8"
    Date: "Wed, 07 May 2014 08:49:40 GMT"
    Expires: "Tue, 31 Mar 1981 05:00:00 GMT"
    Last-Modified: "Wed, 07 May...8:49:40 GMT"
    Pragma: "no-cache"
    Server: "tfe"
    Set-Cookie: "_twitter_sess...ure; HTTPOnly"
    X-Firefox-Spdy: "h2-12"
    ms: "S"
    status: "200 OK"
    strict-transport-security: "max-a...38519"
    x-content-type-options: "nosniff"
    x-frame-options: "SAMEORIGIN"
    x-transaction: "d98124ce7e756fba"
    x-ua-compatible: "IE=edge,chrome=1"
    x-xss-protection: "1; mode=block"

52 requests, 2,757.88 KB, 66.03 s
```

There is no UI element anywhere that tells that you're talking http2. You just can't tell that easily. One way to figure it out, is to enable “Web developer->Network” and check the response headers and see what you got back from the server. The response is then “HTTP/2.0” something and Firefox inserts its own header called “X-Firefox-Spdy:” as shown in the screenshot above.

The headers you see in the Network tool when talking http2 have been converted from http2's binary format into the old-style HTTP 1.x look-alike headers.

9.4. Visualize http2 use

There are Firefox plugins available that help visualize if a site is using http2. One of them is “HTTP/2 and SPDY Indicator”.

10. http2 in Chromium

The Chromium team has implemented http2 and provided support for it in the dev and beta channel for a long time. Starting with Chrome 40, released on January 27th 2015, http2 is enabled by default for a certain amount of users. The amount started off really small and then increased gradually over time.

SPDY support will eventually be removed. In a blog post, the project announced in February 2015:

> “Chrome has supported SPDY since Chrome 6, but since most of the benefits are present in HTTP/2, it’s time to say goodbye. We plan to remove support for SPDY in early 2016”

10.1. First, make sure it is enabled

Enter “chrome://flags/#enable-spdy4" in your browser's address bar and click “enable” if it isn't already showing it as enabled.

10.2. TLS-only

Remember that Chrome only implements http2 over TLS. You will only ever see http2 in action with Chrome when going to https:// sites that offer http2 support.

10.3. Visualize HTTP/2 use

There are Chrome plugins available that helps visualize if a site is using HTTP/2. One of them is “HTTP/2 and SPDY Indicator”.

10.4. QUIC

Chrome's current experiments with QUIC (see section 12.1) dilute the HTTP/2 numbers somewhat.

11. http2 in curl

The curl project has been providing experimental http2 support since September 2013.

In the spirit of curl, we intend to support just about every aspect of http2 that we possibly can. curl is often used as a test tool and tinkerer's way to poke on web sites and we intend to keep that up for http2 as well.

curl uses the separate library nghttp2 for the http2 frame layer functionality. curl requires nghttp2 1.0 or later.

Note that currently on linux curl and libcurl are not always delivered with HTTP/2 protocol support enabled.

11.1. HTTP 1.x look-alike

Internally, curl will convert incoming http2 headers to HTTP 1.x style headers and provide them to the user, so that they will appear very similar to existing HTTP. This allows for an easier transition for whatever is using curl and HTTP today. Similarly curl will convert outgoing headers in the same style. Give them to curl in HTTP 1.x style and it will convert them on the fly when talking to http2 servers. This also allows users to not have to bother or care very much with which particular HTTP version that is actually used on the wire.

11.2. Plain text, insecure

curl supports http2 over standard TCP via the Upgrade: header. If you do an HTTP request and ask for HTTP 2, curl will ask the server to update the connection to http2 if possible.

11.3. TLS and what libraries

curl supports a wide range of different TLS libraries for its TLS back-end, and that is still valid for http2 support. The challenge with TLS for http2's sake is the APLN support and to some extent NPN support.

Build curl against modern versions of OpenSSL or NSS to get both ALPN and NPN support. Using GnuTLS or PolarSSL you will get ALPN support but not NPN.

11.4. Command line use

To tell curl to use http2, either plain text or over TLS, you use the `--http2` option (that is “dash dash http2”). curl still defaults to HTTP/1.1 so the extra option is necessary when you want http2.

11.5. libcurl options

11.5.1 Enable HTTP/2

Your application would use https:// or http:// URLs like normal, but you set curl_easy_setopt's `CURLOPT_HTTP_VERSION` option to `CURL_HTTP_VERSION_2` to make libcurl attempt to use http2. It will then do a best effort and do http2 if it can, but otherwise continue to operate with HTTP 1.1.

11.5.2 Multiplexing

As libcurl tries to maintain existing behaviors to a far extent, you need to enable HTTP/2 multiplexing for your application with the CURLMOPT_PIPELINING option. Otherwise it will continue using one request at a time per connection.

Another little detail to keep in mind is that if you ask for several transfers at once with libcurl, using its multi interface, an applicaton can very well start any number of transfers at once and if you then rather have libcurl wait a little to add them all over the same connection rather than opening new connections for all of them at once, you use the CURLOPT_PIPEWAIT option for each individual transfer you rather wait.

11.5.3 Server push

libcurl 7.44.0 and later supports HTTP/2 server push. You can take advantage of this feature by setting up a push callback with the CURLMOPT_PUSHFUNCTION option. If the push is accepted by the application, it'll create a new transfer as an CURL easy handle and deliver content on it, just like any other transfer.

12. After http2

A lot of tough decisions and compromises have been made for http2. With http2 getting deployed there is an established way to upgrade into other protocol versions that work which lays the foundation for doing more protocol revisions ahead. It also brings a notion and an infrastructure that can handle multiple different versions in parallel. Maybe we don't need to phase out the old entirely when we introduce new?

http2 still has a lot of HTTP 1 “legacy” brought with it into the future because of the desire to keep it possible to proxy traffic back and forth between HTTP 1 and http2. Some of that legacy hampers further development and inventions. Perhaps http3 can drop some of them?

What do you think is still lacking in http?

12.1. QUIC

Google's QUIC (Quick UDP Internet Connections) protocol is an interesting experiment, performed much in the same style and spirit as they did with SPDY. QUIC is a TCP + TLS + HTTP/2 replacement implemented using UDP.

QUIC allows the creation of connections with much less latency, it solves packet loss to only block individual streams instead of all of them like it does for HTTP/2 and it makes connections possible to be done over different network interfaces easily - thus also covering areas MPTCP is meant to solve.

QUIC is so far only implemented by Google in Chrome and their server ends and that code is not easily re-used elsewhere, even if there's a libquic effort trying exactly that. The protocol has been brought as a draft to the IETF transport working group.

13. Further reading

If you think this document was a bit light on content or technical details, here are additional resources to help you satisfy your curiosity:

- The HTTPbis mailing list and its archives: http://lists.w3.org/Archives/Public/ietf-http-wg/
- The actual http2 specification in a HTMLified version: https://httpwg.github.io/specs/rfc7540.html
- Firefox http2 networking details: https://wiki.mozilla.org/Networking/http2
- curl http2 implementation details: http://curl.haxx.se/docs/http2.html
- The http2 web site: http://http2.github.io/ and perhaps in particular the FAQ: http://http2.github.io/faq/
- Ilya Grigorik's HTTP/2 chapter in his book “High Performance Browser Networking”: http://chimera.labs.oreilly.com/books/1230000000545/ch12.html

14. Thanks

Inspiration and the package format Lego image from Mark Nottingham.

HTTP trend data comes from http://httparchive.org.

The RTT graph comes from presentations done by Mike Belshe.

My kids Agnes and Rex for letting me borrow their Lego figures for the head of line picture.

Thanks to the following friends for reviews and feedback: Kjell Ericson, Bjorn Reese, Linus Swälas and Anthony Bryan. Your help is greatly appreciated and has really improved the document!

During the various iterations, the following friendly people have provided bug reports and improvements to the document: Mikael Olsson, Remi Gacogne, Benjamin Kircher, saivlis, florin-andrei-tp, Brett Anthoine, Nick Parlante, Matthew King, Nicolas Peels, Jon Forrest, sbrickey, Marcin Olak, Gary Rowe, Ben Frain, Mats Linander, Raul Siles, Alex Lee, Richard Moore