http2 explained

Background, the protocol, the implementations and the future

http://daniel.haxx.se/http2/

by Daniel Stenberg

Table of Contents

1.Background

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

The final http2 specification has just been completed and was okayed by the IESG on February 18th 2015. The current version is called draft-17[1] and this is the protocol format that will become the final RFC. All and any errors in the document are my own and the results of my shortcomings. Please point them out to me and I might do updates with corrections.

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.

This is document version 1.12, published on March 19, 2015.

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   http://tools.ietf.org/html/draft-ietf-httpbis-http2-17

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 1.12:

- 9.1: mention the Firefox 36+ config switch for http2

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: February 11, 2015

- 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 "missing implementations" added
- 8.4.3: mention that TLS also increases success rate

Version 1.8: January 15th, 2015

- Compressed the images better, leading to a *much* smaller PDF
- Updated to draft-16 and hpack-10
- Replaced several images
- Linkified many URLs
- Added a few questions in 8.4
- Mentions IETF Last Call

2.HTTP today

HTTP 1.1 has turned into a protocol used for virtually everything on the Internet. Huge investments have been done on protocols and infrastructure that take advantage of this. This is taken 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 have grown a software ecosystem where almost no implementations ever implement everything – and it isn't even really possible to exactly tell what "everything" is. This has lead 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.

Then later on, it 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 moments in time 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'll 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.
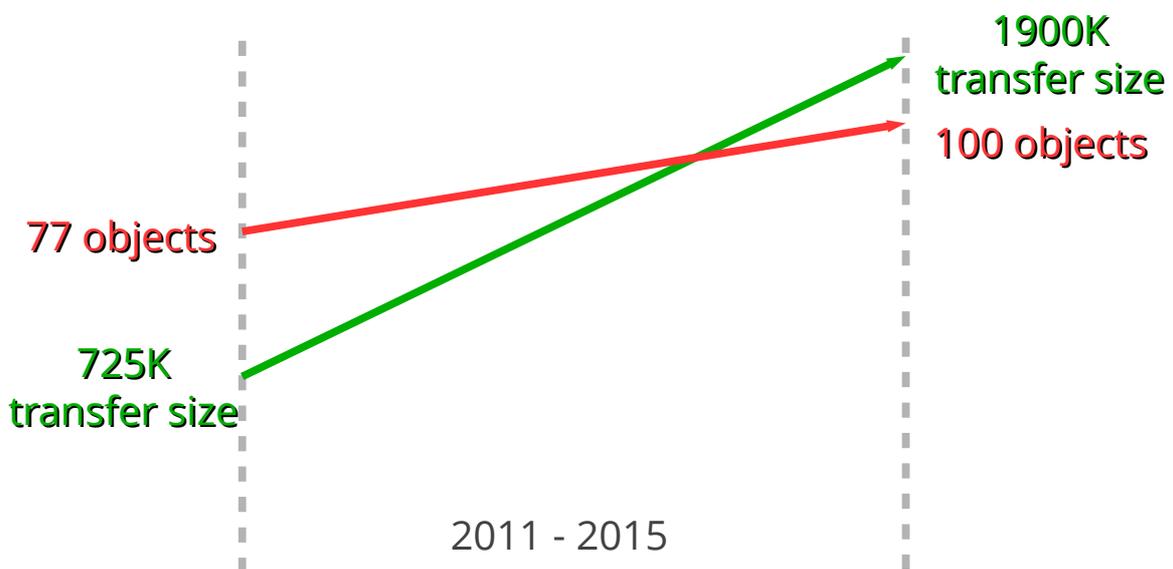
data from httparchive.org

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
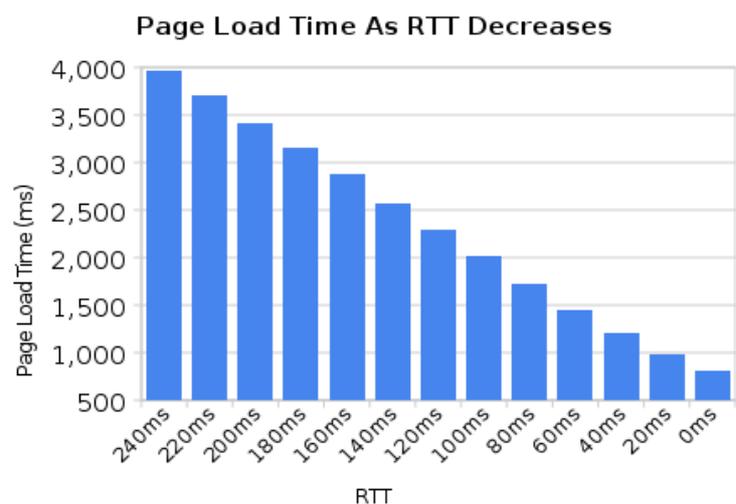
Drawing 1: RTT, Round Trip Time affects Page Load Time

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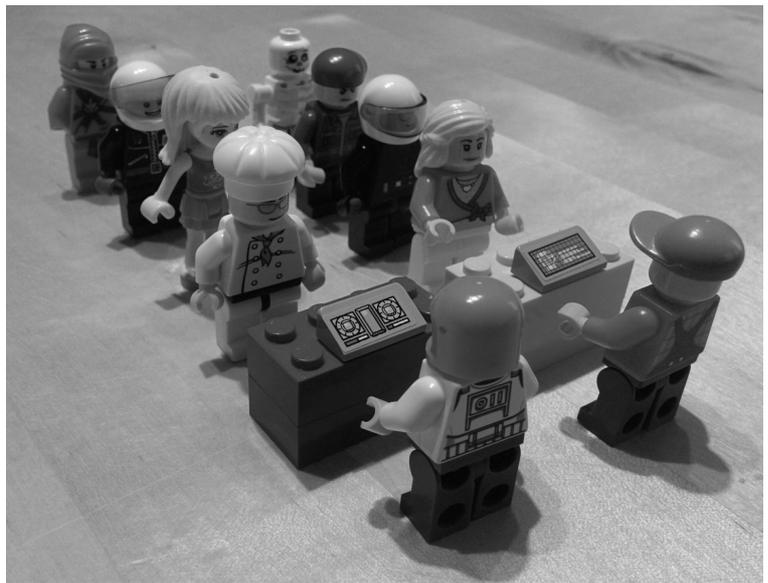
Even today, 2015, most desktop web browsers ship with HTTP pipelining disabled by default.

Additional reading on this subject can be found for example in the Firefox bugzilla entry 264354[2].

---

2   https://bugzilla.mozilla.org/show_bug.cgi?id=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.

```css
.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 everyone 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 and beyond 100 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 38(!) 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.

●	GET	/	www.aftonbladet.se	html	205.71 KB	◑ → 32 ms
●	GET	general.css?57785785	www.aftonbladet.se	css	193.22 KB	◑ → 10 ms
●	GET	d-head.js?57785785	www.aftonbladet.se	js	190.40 KB	◑ → 20 ms
●	GET	aftonbladet.gif	gfx.aftonbladet-cdn.se	gif	6.33 KB	○ → 44 ms
●	GET	sanna_lundell_white.jpg	gfx.aftonbladet-cdn.se	jpeg	3.01 KB	○ → 44 ms
●	GET	pappa01.jpg	gfx2.aftonbladet-cdn.se	jpeg	18.66 KB	◑ → 228 ms
●	GET	forskare.jpg	gfx.aftonbladet-cdn.se	jpeg	6.88 KB	○ → 44 ms
●	GET	story26jan.jpg	gfx.aftonbladet-cdn.se	jpeg	2.66 KB	○ → 45 ms
●	GET	osbalk.jpg	gfx2.aftonbladet-cdn.se	jpeg	4.11 KB	○ → 238 ms
●	GET	Bank.jpg	gfx2.aftonbladet-cdn.se	jpeg	16.09 KB	◑ → 247 ms
●	GET	Tatu.JPG	gfx.aftonbladet-cdn.se	jpeg	7.38 KB	○ → 44 ms
●	GET	sve8.jpg	gfx.aftonbladet-cdn.se	jpeg	7.39 KB	○ → 45 ms
●	GET	otto-tvaa.jpg	gfx.aftonbladet-cdn.se	jpeg	6.71 KB	○ → 46 ms
●	GET	AStridTT.JPG	gfx.aftonbladet-cdn.se	jpeg	6.52 KB	○ → 66 ms
●	GET	nimoy.jpg	gfx.aftonbladet-cdn.se	jpeg	7.87 KB	○ → 67 ms
●	GET	PSYK0§.jpg	gfx2.aftonbladet-cdn.se	jpeg	18.28 KB	◑ → 257 ms
●	GET	parislgh.jpg	gfx2.aftonbladet-cdn.se	jpeg	21.89 KB	◑ → 267 ms
●	GET	ProjectRunwayvinnare.jpg	gfx2.aftonbladet-cdn.se	jpeg	7.83 KB	○ → 276 ms
●	GET	par_strand-opt.jpg	gfx2.aftonbladet-cdn.se	jpeg	5.72 KB	○ → 286 ms
●	GET	mack.jpg	gfx2.aftonbladet-cdn.se	jpeg	16.27 KB	○ → 299 ms
●	GET	skildaVpuffmitt.jpg	gfx1.aftonbladet-cdn.se	jpeg	4.49 KB	○ → 144 ms
●	GET	socker_ny.jpg	gfx1.aftonbladet-cdn.se	jpeg	3.79 KB	○ → 153 ms
●	GET	andreetta.jpg	gfx1.aftonbladet-cdn.se	jpeg	5.53 KB	○ → 163 ms
●	GET	skidor_50.gif	gfx.aftonbladet-cdn.se	gif	3.20 KB	○ → 70 ms

4. Updating HTTP

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

1. Make a protocol that's less RTT sensitive

2. Fix pipelining and the head of line blocking problem

3. Stop the need for and desire to keep increasing the number of connections to each host

4. Keep all existing interfaces, all content, the URI formats and schemes

5. This would 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 documents documenting everything from TCP, DNS, 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 humans and individuals, little care is given to which companies the individuals 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 7320 series.

The supposedly 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 to actually get an official RFC out would prove to continue over 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"[3]. 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. Like in this case for HTTP 1.1.

---

3   http://en.wiktionary.org/wiki/bis#Latin

4.2. http2 started from SPDY

SPDY[4] 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 numbers published that proved how it performed. The http2 work then subsequently started off from the SPDY/3 draft that was basically made into the http2 draft-00 with a little search and replace.

---

4   http://en.wikipedia.org/wiki/SPDY

5. http2 concepts

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

They were actually quite strict and put quite a few restraints on the team's ability to innovate.

✔ It 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 1:1.

✔ Remove or reduce optional parts from the protocol. This wasn't really a requirement but more a mantra coming over 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 there comes a need 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 has to be done using the existing ones. Since they are used for HTTP 1.x today, we obviously need to have 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 how 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 a cost of a round-trip.

That round-trip penalty was not something the SPDY team would accept, and as they also only implemented SPDY over TLS they developed a new TLS extension which is used to shortcut the negotiation quite significantly. Using this extension, called NPN

for Next Protocol Negotiation, the server tells the client which protocols it knows and the client can then proceed and 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 is only done over TLS and there's been a strong push for making TLS mandatory for http2 but it didn't get consensus and http2 will ship 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 the leading web browsers of today.

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

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

Similarly, there's been a fierce and long-going debate on whether http2 should dictate a list of ciphers that should be mandatory when using TLS, or if it perhaps should blacklist a set or if it shouldn't require anything at all from the TLS "layer" but leave that to the TLS WG.

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 ALPN came out of that: *Application Layer Protocol Negotiation*. ALPN is what is being promoted to be used for 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 lead to many early http2 clients and http2 servers implementing and using both these extensions when negotiating http2. Also, as NPN is what's used for SPDY and many servers offer both SPDY and http2 so supporting both NPN and ALPN on those servers make perfect sense.

ALPN primarily differs from NPN in who decides what protocol to speak. With ALPN the client tells 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 that final choice.

5.4. http2 for http://

As mentioned briefly previously, for plain-text HTTP 1.1 the way to negotiate http2 is by asking 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. You of course realize that this upgrade procedure costs a full network round-trip, but the upside is that a http2 connection should be possible to keep alive and re-use to a larger extent than HTTP1 connections generally are.

While some browsers' 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 took us here. Let's dive into the specifics of the protocol. The bits and the concepts that create http2.

6.1. Binary

http2 is a binary protocol.

Just let that sink in for a minute. If you're a person who has been involved in internet protocols before, chances are that you will now be instinctively reacting against this choice and marshaling your arguments that spell out how protocols that are made based on text/ascii are superior because humans can handcraft request 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 spaces and different ways to write the same thing, implementations become simpler.

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

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

Debugging of this protocol will instead 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:

Type, Length, Flags, Stream Identifier and frame payload.

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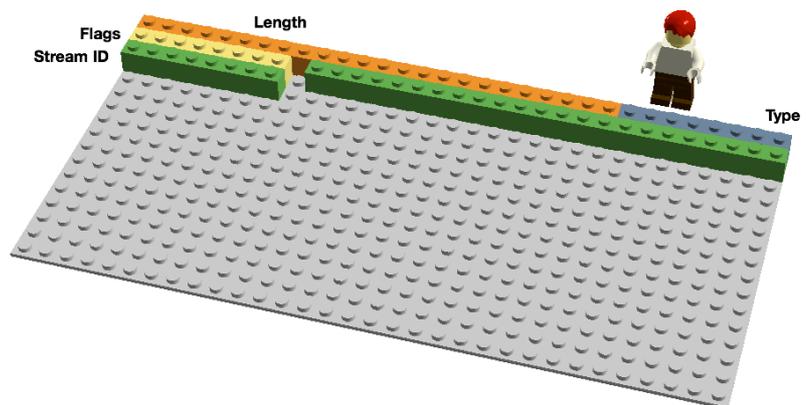
6.3. Multiplexed streams

The Stream Identifier mentioned in the previous section describing the binary frame format, makes each frame sent over http2 get associated with a "stream". A stream is a logical association. 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:

So they're crammed into and over the same connection in a multiplexed manner:

In http2 we will see tens and hundreds of simultaneous streams. The cost of creating a new one is very low.

6.4. Priorities and dependencies

Each stream also has a priority, which is used to tell the peer which streams to consider most important.

The exact details on how priorities work in the protocols have changed several times and are still being debated. The point is however that a client can specify which streams that are most important and there's a dependency parameter so that one stream can be made dependent on another.

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

6.5. Header compression

HTTP is a state-less protocol. In short that means that every request needs to bring with it as much details 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 any such paradigms, it too has to do this.

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 something begs for compression.

While the number of objects per web page increases as I've 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, mostly the same over many requests.

The HTTP 1.1 request sizes have actually gotten so large over time so they sometimes even 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. Another argument for compression.

6.5.1. Compression is a tricky subject

HTTPS and SPDY compressions were found to be vulnerable to the BREACH[5] and CRIME attacks[6]. By inserting known text into the stream and figuring out how that changes the output, an attacker can figure out what's being sent.

Doing compression on dynamic content for a protocol without then becoming vulnerable for one of these attacks requires some thoughts and careful

---

5   http://en.wikipedia.org/wiki/BREACH_%28security_exploit%29
6   http://en.wikipedia.org/wiki/CRIME

considerations. This is what the HTTPbis team tries to do.

Enter HPACK[7], *Header Compression for HTTP/2*, which – as the name suitably suggests - is a compression format especially crafted for http2 headers and it is strictly speaking 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 this 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 a 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 then comes as the price of having to negotiate a new TCP handshake again.

A better solution would be to just stop the message and start anew. This can be done with http2's RST_STREAM frame which will help in preventing wasted bandwidth and avoid having to tear down any connection.

6.7. Server push

This is the feature also known as "cache push". The idea here is that if the client asks for resource X the server may know that the client then also most likely want resource Z and sends that to the client without it asking for it. It helps the client to put Z into its cache so that it'll be there when it wants it.

Server push is something a client explicitly must allow the server to do and even if a client does that, it can at its own choice swiftly terminate a pushed stream with RST_STREAM should it not want a particular one.

6.8. Flow Control

Each individual stream over http2 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 more room to fit incoming data in, and the other end is only allowed to send that much data until the window is extended. Only DATA frames are flow controlled.

---

7   http://tools.ietf.org/html/draft-ietf-httpbis-header-compression-12

7.Extensions

The protocol mandates that a receiver must read and ignore all unknown frames using unknown frame types. Two parties can thus negotiate use of new frame types on a hop-by-hop basis, and 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 time the protocol was developed with opinions swinging for and against. After draft-12 the pendulum swept back one last time and extensions were allowed again.

Extensions are then not part of the actual protocol but will be documented outside of the core protocol spec. Already at this point, there are two frame types that have been discussed for inclusion in the protocol that probably will be the first frames sent as extensions. I'll still describe them here just because of their popularity and previous state as "native" frames:

7.1. Alternative Services

With http2 getting adopted, 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 single one could then potentially be open for quite some time.

This will affect how HTTP load balancers work and there may come situations when a site wants to advertise and suggest that the client connects to another host. It could be for performance reasons but also if a site is being taken down for maintance and similar.

The server will then send the Alt-Svc: header[8] (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 is then meant to attempt to connect to that service asynchronously and only use the alternative if that works fine.

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 debated feature. Such a connection would do unauthenticated TLS and wouldn't be advertized as "secure" anywhere, wouldn't use any padlock in the UI or in fact in no way 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.

---

8   http://tools.ietf.org/html/draft-ietf-httpbis-alt-svc-01

7.2. Blocked

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

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.A 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[9] on the http2 web site.
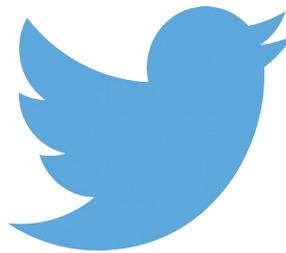
There was a large amount of implementations already early on, and the amount has increased over time during the http2 work. At the time of writing this there are over 30 implementations listed, and most of them implement the latest drafts.

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.

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

While draft-17 is the current draft version, it is binary compatible back to draft-14 and the -14 version is the last one that is marked as an implementation / interop draft. But while binary compatible over the wire, the language in the spec drafts have changed somewhat so they are not identical behavior wise.

8.3.1.   Missing implementations

The list of existing implementations still lacks some reputable brands. We have not heard anything official from Apple about support for http2 in Safari. The two immensely popular servers Apache HTTPD and Nginx both offer SPDY support but none of them have yet shown official http2 support.

Nginx says "we plan to release versions of nginx and NGINX Plus by the end of 2015 that will include support for HTTP/2"[10]. There's an early version of a HTTP/2 module

---

9   https://github.com/http2/http2-spec/wiki/Implementations
10   http://nginx.com/blog/how-nginx-plans-to-support-http2/

for Apache called mod_h2[11] claimed to be "Very alpha"

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[12] 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

---

11 https://icing.github.io/mod_h2/
12 http://blog.chromium.org/2015/02/hello-http2-goodbye-spdy-http-is_9.html

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[13] 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?"[14] by people at University of Washington and "Evaluating the Performance of SPDY-enabled Web Servers"[15] 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[16] might imply that HTTP/2 holds its promises.

http2 is demonstrably faster in some scenarios, especially with high latency connections involving many objects and as shown in previous sections, the trend is heading towards even more objects and more data per site.

---

13   http://www.atis.org/openweballiance/docs/OWAKickoffSlides051414.pdf (page 26)
14   https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-wang_xiao_sophia.pdf and
     https://www.usenix.org/sites/default/files/conference/protected-files/nsdi14_slides_wang.pdf
15   http://www.neotys.com/blog/performance-of-spdy-enabled-web-servers/
16   http://blog.httpwatch.com/2015/01/16/a-simple-performance-comparison-of-https-spdy-and-http2/

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 and Internet Explorer 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[17] 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.

---

17 https://github.com/h2o/h2o

I think there is likely to be even more implementations popping up once the spec becomes a ratified RFC.

In late January 2015, after Firefox 35 had shipped with HTTP/2 enabled by default and Chrome 40 had it enabled for 2% of its users, Google reported early and not statistically safe numbers to me. They then saw HTTP/2 in roughly 5% of their global traffic. At the same time, Firefox 35 responses recorded HTTP/2 in about 9% of all responses.

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.
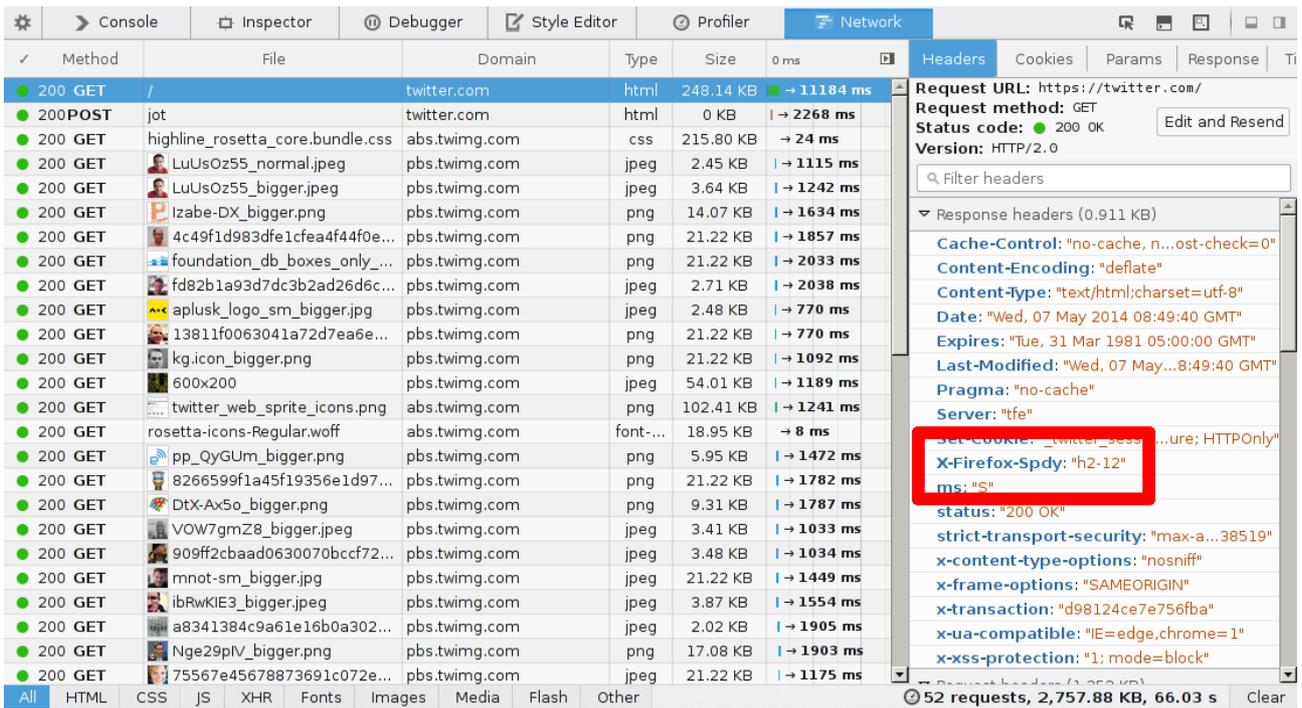
9.3. Transparent!

Illustration 1: Screenshot showing http2 draft-12 being used by Firefox

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 HTTP/2 use

There are Firefox plugins available that help visualize if a site is using HTTP/2. One of them is "SPDY Indicator"[18].

---

18 https://addons.mozilla.org/en-US/firefox/addon/spdy-indicator/

10. http2 in Chromium

The Chromium team has implemented HTTP/2 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 exact amount started off really small but is planned to increase gradually over time.

SPDY support will be removed. In a blog post, the project announced in February 2015[19]:

> *"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 "SPDY Indicator"[20].

---

19 http://blog.chromium.org/2015/02/hello-http2-goodbye-spdy-http-is_9.html
20 https://chrome.google.com/webstore/detail/spdy-indicator/mpbpobfflnpcgagjijhmgnchggcjblin

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[21] for the http2 frame layer functionality.

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 a 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

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,

---

21   https://nghttp2.org/

but otherwise continue to operate with HTTP 1.1.

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?

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 drafts and associated documents from the HTTPbis group: http://datatracker.ietf.org/wg/httpbis/

- Firefox http2 networking details: https://wiki.mozilla.org/Networking/http2

- curl http2 implementation details: http://curl.haxx.se/dev/readme-http2.html

- The http2 web site: http://http2.github.io/ and perhaps in particular the FAQ: http://http2.github.io/faq/

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