HTTP/2 Frequently Asked Questions

http2.github.io/faq/

HTTP/2

- IETF HTTP WG

These are Frequently Asked Questions about HTTP/2.

General Questions

Why revise HTTP?

HTTP/1.1 has served the Web well for more than fifteen years, but its age is starting to show.

Loading a Web page is more resource intensive than ever (see the HTTP Archive's page size statistics), and loading all of those assets efficiently is difficult, because HTTP practically only allows one outstanding request per TCP connection.

In the past, browsers have used multiple TCP connections to issue parallel requests. However, there are limits to this; if too many connections are used, it's both counter-productive (TCP congestion control is effectively negated, leading to congestion events that hurt performance and the network), and it's fundamentally unfair (because browsers are taking more than their share of network resources).

At the same time, the large number of requests means a lot of duplicated data "on the wire".

Both of these factors means that HTTP/1.1 requests have a lot of overhead associated with them; if too many requests are made, it hurts performance.

This has led the industry to a place where it's considered Best Practice to do things like spriting, data: inlining, domain sharding and concatenation. These hacks are indications of underlying problems in the protocol itself, and cause a number of problems on their own when used.

Who made HTTP/2?

HTTP/2 was developed by the IETF's HTTP Working Group, which maintains the HTTP protocol. It's made up of a number of HTTP implementers, users, network operators and HTTP experts.

Note that while our mailing list is hosted on the W3C site, this is *not* a W3C effort. Tim Berners-Lee and the W3C TAG are kept up-to-date with the WG's progress, however.

A large number of people have contributed to the effort, but the most active participants include engineers from "big" projects like Firefox, Chrome, Twitter, Microsoft's HTTP stack, Curl and Akamai, as well as a number of HTTP implementers in languages like Python, Ruby and NodeJS.

To learn more about participating in the IETF, see the Tao of the IETF; you can also get a sense of who's contributing to the specification on Github's contributor graph, and who's implementing on our implementation list.

What's the relationship with SPDY?

HTTP/2 was first discussed when it became apparent that SPDY was gaining traction with implementers (like Mozilla and nginx), and was showing significant improvements over HTTP/1.x.

After a call for proposals and a selection process, SPDY/2 was chosen as the basis for HTTP/2. Since then,

there have been a number of changes, based on discussion in the Working Group and feedback from implementers.

Throughout the process, the core developers of SPDY have been involved in the development of HTTP/2, including both Mike Belshe and Roberto Peon.

In February 2015, Google announced its plans to remove support for SPDY in favor of HTTP/2.

Is it HTTP/2.0 or HTTP/2?

The Working Group decided to drop the minor version (".0") because it has caused a lot of confusion in HTTP/1.x.

In other words, the HTTP version *only* indicates wire compatibility, not feature sets or "marketing."

What are the key differences to HTTP/1.x?

At a high level, HTTP/2:

- is binary, instead of textual
- is fully multiplexed, instead of ordered and blocking
- can therefore use one connection for parallelism
- uses header compression to reduce overhead
- allows servers to "push" responses proactively into client caches

Why is HTTP/2 binary?

Binary protocols are more efficient to parse, more compact "on the wire", and most importantly, they are much less error-prone, compared to textual protocols like HTTP/1.x, because they often have a number of affordances to "help" with things like whitespace handling, capitalization, line endings, blank lines and so on.

For example, HTTP/1.1 defines four different ways to parse a message; in HTTP/2, there's just one code path.

It's true that HTTP/2 isn't usable through telnet, but we already have some tool support, such as a Wireshark plugin.

Why is HTTP/2 multiplexed?

HTTP/1.x has a problem called "head-of-line blocking," where effectively only one request can be outstanding on a connection at a time.

HTTP/1.1 tried to fix this with pipelining, but it didn't completely address the problem (a large or slow response can still block others behind it). Additionally, pipelining has been found very difficult to deploy, because many intermediaries and servers don't process it correctly.

This forces clients to use a number of heuristics (often guessing) to determine what requests to put on which connection to the origin when; since it's common for a page to load 10 times (or more) the number of available connections, this can severely impact performance, often resulting in a "waterfall" of blocked requests.

Multiplexing addresses these problems by allowing multiple request and response messages to be in flight at the same time; it's even possible to intermingle parts of one message with another on the wire.

This, in turn, allows a client to use just one connection per origin to load a page.

Why just one TCP connection?

With HTTP/1, browsers open between four and eight connections per origin. Since many sites use multiple origins, this could mean that a single page load opens more than thirty connections.

One application opening so many connections simultaneously breaks a lot of the assumptions that TCP was built upon; since each connection will start a flood of data in the response, there's a real risk that buffers in the intervening network will overflow, causing a congestion event and retransmits.

Additionally, using so many connections unfairly monopolizes network resources, "stealing" them from other, better-behaved applications (e.g., VoIP).

What's the benefit of Server Push?

When a browser requests a page, the server sends the HTML in the response, and then needs to wait for the browser to parse the HTML and issue requests for all of the embedded assets before it can start sending the JavaScript, images and CSS.

Server Push potentially allows the server to avoid this round trip of delay by "pushing" the responses it thinks the client will need into its cache.

However, Pushing responses is not "magical" – if used incorrectly, it can harm performance. Correct use of Server Push is an ongoing area of experimentation and research.

Why do we need header compression?

Patrick McManus from Mozilla showed this vividly by calculating the effect of headers for an average page load.

If you assume that a page has about 80 assets (which is conservative in today's Web), and each request has 1400 bytes of headers (again, not uncommon, thanks to Cookies, Referer, etc.), it takes at least 7-8 round trips to get the headers out "on the wire." That's not counting response time - that's just to get them out of the client.

This is because of TCP's Slow Start mechanism, which paces packets out on new connections based on how many packets have been acknowledged – effectively limiting the number of packets that can be sent for the first few round trips.

In comparison, even mild compression on headers allows those requests to get onto the wire within one roundtrip – perhaps even one packet.

This overhead is considerable, especially when you consider the impact upon mobile clients, which typically see round-trip latency of several hundred milliseconds, even under good conditions.

Why HPACK?

SPDY/2 proposed using a single GZIP context in each direction for header compression, which was simple to implement as well as efficient.

Since then, a major attack has been documented against the use of stream compression (like GZIP) inside of encryption; CRIME.

With CRIME, it's possible for an attacker who has the ability to inject data into the encrypted stream to "probe" the plaintext and recover it. Since this is the Web, JavaScript makes this possible, and there were demonstrations of recovery of cookies and authentication tokens using CRIME for TLS-protected HTTP resources.

As a result, we could not use GZIP compression. Finding no other algorithms that were suitable for this use case as well as safe to use, we created a new, header-specific compression scheme that operates at a coarse granularity; since HTTP headers often don't change between messages, this still gives reasonable compression efficiency, and is much safer.

Can HTTP/2 make cookies (or other headers) better?

This effort was chartered to work on a revision of the wire protocol – i.e., how HTTP headers, methods, etc. are put “onto the wire”, not change HTTP’s semantics.

That’s because HTTP is so widely used. If we used this version of HTTP to introduce a new state mechanism (one example that’s been discussed) or change the core methods (thankfully, this hasn’t yet been proposed), it would mean that the new protocol was incompatible with the existing Web.

In particular, we want to be able to translate from HTTP/1 to HTTP/2 and back with no loss of information. If we started “cleaning up” the headers (and most will agree that HTTP headers are pretty messy), we’d have interoperability problems with much of the existing Web.

Doing that would just create friction against the adoption of the new protocol.

All of that said, the HTTP Working Group is responsible for all of HTTP, not just HTTP/2. As such, we can work on new mechanisms that are version-independent, as long as they’re backwards-compatible with the existing Web.

What about non-browser users of HTTP?

Non-browser applications should be able to use HTTP/2 as well, if they’re already using HTTP.

Early feedback has been that HTTP/2 has good performance characteristics for HTTP “APIs”, because the APIs don’t need to consider things like request overhead in their design.

Having said that, the main focus of the improvements we’re considering is the typical browsing use cases, since this is the core use case for the protocol.

Our charter says this about it:

```
The resulting specification(s) are expected to meet these goals for common
existing deployments of HTTP; in particular, Web browsing (desktop and
mobile), non-browsers ("HTTP APIs"), Web serving (at a variety of scales),
and
intermediation (by proxies, corporate firewalls, "reverse" proxies and
Content
Delivery Networks). Likewise, current and future semantic extensions to
HTTP/1.x (e.g., headers, methods, status codes, cache directives) should be
supported in the new protocol.

Note that this does not include uses of HTTP where non-specified behaviours
are relied upon (e.g., connection state such as timeouts or client affinity,
and "interception" proxies); these uses may or may not be enabled by the
final
product.
```

Does HTTP/2 require encryption?

No. After extensive discussion, the Working Group did not have consensus to require the use of encryption (e.g., TLS) for the new protocol.

However, some implementations have stated that they will only support HTTP/2 when it is used over an encrypted connection, and currently no browser supports HTTP/2 unencrypted.

What does HTTP/2 do to improve security?

HTTP/2 defines a profile of TLS that is required; this includes the version, a ciphersuite blacklist, and extensions used.

See the spec for details.

There is also discussion of additional mechanisms, such as using TLS for HTTP:// URLs (so-called “opportunistic encryption”); see the relevant draft.

Can I use HTTP/2 now?

In browsers, HTTP/2 is supported by the most current releases of Edge, Safari, Firefox and Chrome. Other browsers based upon Blink will also support HTTP/2 (e.g., Opera and Yandex Browser). See the caniuse for more details.

There are also several servers available (including beta support from Akamai, Google and Twitter’s main sites), and a number of Open Source implementations that you can deploy and test.

See the implementations list for more details.

Will HTTP/2 replace HTTP/1.x?

The goal of the Working Group is that typical uses of HTTP/1.x *can* use HTTP/2 and see some benefit. Having said that, we can’t force the world to migrate, and because of the way that people deploy proxies and servers, HTTP/1.x is likely to still be in use for quite some time.

Will there be a HTTP/3?

If the negotiation mechanism introduced by HTTP/2 works well, it should be possible to support new versions of HTTP much more easily than in the past.

Implementation Questions

Why the rules around Continuation on HEADERS frames?

Continuation exists since a single value (e.g. Set-Cookie) could exceed 16KiB - 1, which means it couldn’t fit into a single frame. It was decided that the least error-prone way to deal with this was to require that all of the headers data come in back-to-back frames, which made decoding and buffer management easier.

What is the minimum or maximum HPACK state size?

The receiver always controls the amount of memory used in HPACK, and can set it to zero at a minimum, with a maximum related to the maximum representable integer in a SETTINGS frame, currently 2^32 - 1.

How can I avoid keeping HPACK state?

Send a SETTINGS frame setting state size (SETTINGS_HEADER_TABLE_SIZE) to zero, then RST all streams until a SETTINGS frame with the ACK bit set has been received.

Why is there a single compression/flow-control context?

Simplicity.

The original proposals had stream groups, which would share context, flow control, etc. While that would benefit proxies (and the experience of users going through them), doing so added a fair bit of complexity. It was decided that we’d go with the simple thing to begin with, see how painful it was, and address the pain (if any) in a future protocol revision.

Why is there an EOS symbol in HPACK?

HPACK's huffman encoding, for reasons of CPU efficiency and security, pads out huffman-encoded strings to the next byte boundary; there may be between 0-7 bits of padding needed for any particular string.

If one considers huffman decoding in isolation, any symbol that is longer than the required padding would work; however, HPACK's design allows for bytewise comparison of huffman-encoded strings. By requiring that the bits of the EOS symbol are used for padding, we ensure that users can do bytewise comparison of huffman-encoded strings to determine equality. This in turn means that many headers can be interpreted without being huffman decoded.

Can I implement HTTP/2 without implementing HTTP/1.1?

Yes, mostly.

For HTTP/2 over TLS (`h2`), if you do not implement the `http1.1` ALPN identifier, then you will not need to support any HTTP/1.1 features.

For HTTP/2 over TCP (`h2c`), you need to implement the initial upgrade request.

`h2c`-only clients will need to generate an OPTIONS request for "*" or a HEAD request for "/", which are fairly safe and easy to construct. Clients looking to implement HTTP/2 only will need to treat HTTP/1.1 responses without a 101 status code as errors.

`h2c`-only servers can accept a request containing the Upgrade header field with a fixed 101 response. Requests without the `h2c` upgrade token can be rejected with a 505 (HTTP Version Not Supported) status code that contains the Upgrade header field. Servers that don't wish to process the HTTP/1.1 response should reject stream 1 with a REFUSED_STREAM error code immediately after sending the connection preface to encourage the client to retry the request over the upgraded HTTP/2 connection.

Is the priority example in Section 5.3.2 incorrect?

No. Stream B has weight 4, stream C has weight 12. To determine the proportion of the available resources that each of these streams receive, sum all the weights (16) and divide each stream weight by the total weight. Stream B therefore receives one-quarter of the available resources and stream C receives three-quarters. Consequently, as the specification states: stream B ideally receives one-third of the resources allocated to stream C.

Will I need TCP_NODELAY for my HTTP/2 connections?

Yes, probably. Even for a client-side implementation that only downloads a lot of data using a single stream, some packets will still be necessary to send back in the opposite direction to achieve maximum transfer speeds. Without TCP_NODELAY set (still allowing the Nagle algorithm), the outgoing packets may be held for a while in order to allow them to merge with a subsequent one.

In cases where such a packet, for example, is a packet telling the peer that there is more window available to send data, delaying its sending for multiple milliseconds (or more) can have a severe impact on high speed connections.

Deployment Questions

How do I debug HTTP/2 if it's encrypted?

There are many ways to get access to the application data, but the easiest is to use NSS keylogging in combination with the Wireshark plugin (included in recent development releases). This works with both Firefox and Chrome.

How can I use HTTP/2 server push?

HTTP/2 server push allows a server to provide content to clients without waiting for a request. This can improve the time to retrieve a resource, particularly for connections with a large bandwidth-delay product where the network round trip time comprises most of the time spent on a resource.

Pushing resources that vary based on the contents of a request could be unwise. Currently, browsers only use pushed requests if they would otherwise make a matching request (see Section 4 of RFC 7234).

Some caches don't respect variations in all request header fields, even if they are listed in the `Vary` header field. To maximize the likelihood that a pushed resource will be accepted, content negotiation is best avoided. Content negotiation based on the `accept-encoding` header field is widely respected by caches, but other header fields might not be as well supported.