acmqueue

Making the Web Faster with HTTP 2.0

HTTP continues to evolve

Ilya Grigorik

HTTP (Hypertext Transfer Protocol) is one of the most widely used application protocols on the Internet. Since its publication, RFC 2616 (HTTP 1.1) has served as a foundation for the unprecedented growth of the Internet: billions of devices of all shapes and sizes, from desktop computers to the tiny Web devices in our pockets, speak HTTP every day to deliver news, video, and millions of other Web applications we have all come to depend on in our everyday lives.

What began as a simple one-line protocol for retrieving hypertext (i.e., `"GET /resource"`—Telnet to google.com on port 80, take HTTP 1.0 for a spin) quickly evolved into a generic hypermedia transport protocol. Now a decade later it is used to power just about any use case imaginable.

Under the weight of its own success, however, and as more and more everyday interactions continue to migrate to the Web—social, e-mail, news and video, and, increasingly, our personal and job workspaces—HTTP has begun to show signs of stress. Users and developers alike are now demanding near-realtime responsiveness and protocol performance from HTTP 1.1, which it simply cannot provide without some modifications.

To meet these new challenges, HTTP must continue to evolve, which is where HTTP 2.0 enters the picture. HTTP 2.0 will make applications faster, simpler, and more robust by enabling efficient multiplexing and low-latency delivery over a single connection and allowing Web developers to undo many of the application "hacks" used today to work around the limitations of HTTP 1.1.

PERFORMANCE CHALLENGES OF MODERN WEB APPLICATIONS

A lot has changed in the decade since the HTTP 1.1 RFC was published: browsers have evolved at an accelerating rate, user connectivity profiles have changed, with the mobile Web now at an inflection point, and Web applications have grown in their scope, ambition, and complexity. Some of these factors help performance while others hinder. On balance, Web performance remains a large and unsolved problem.

TABLE 1 **Global broadband adoption**

Rank	Country	Average Mbps	Year over Year change
-	Global	3.1	17%
1	South Korea	14.2	-10%
2	Japan	11.7	6.8%
3	Hong Kong	10.9	16%
4	Switzerland	10.1	24%
5	Netherlands	9.9	12%
...			
9	United States	8.6	27%

First, the good news: modern browsers have put significant effort into performance. JavaScript execution speed continues its steady climb (e.g., the launch of the Chrome browser in 2008 delivered a 20x improvement, and in 2012 alone, the performance was further improved by more than 50 percent on mobile[11]). And it's not just JavaScript where the improvement is occurring; modern browsers also leverage GPU acceleration for drawing and animation (e.g., CSS3 animations and WebGL), provide direct access to native device APIs, and leverage numerous speculative optimization techniques[5] to help hide and reduce various sources of network latency.

Similarly, broadband adoption (table 1) has continued its steady climb over the past decade. According to Akamai, while the global average is now at 3.1 Mbps, many users have access to far higher throughput, especially with the rollout of residential fiber solutions.[1] Bandwidth is only half the equation, however. Latency is the oft-forgotten factor, and unfortunately, it is now often *the limiting factor* when it comes to browsing the Web.[4]

In practice, once the user has more than 5 Mbps of bandwidth, further improvements deliver minimal increase in the loading speed of the average Web application: streaming HD video from the Web is bandwidth-bound; loading the page hosting the HD video, with all of its assets, is latency-bound.

A modern Web application looks significantly different from a decade ago. According to HTTP Archive,[6] an average Web application is now composed of more than 90 resources, which are fetched from more than 15 distinct hosts, totaling more than 1,300 KB of (compressed) transferred data. As a result, a large fraction of HTTP data flows consist of small (less than 15 KB), bursty data transfers over dozens of distinct TCP connections. Therein lies the problem. TCP is optimized for long-lived connections and bulk data transfers. Network RTT (round-trip time) is the limiting factor in throughput of new TCP connections (a result of TCP congestion control), and consequently, latency is also the performance bottleneck for most Web applications.

How do you address this mismatch? First, you could try to reduce the round-trip latency by positioning the servers and bits closer to the user, as well as using lower-latency links. Unfortunately, while these are necessary optimizations—there is now an entire CDN (content delivery network) industry focused on exactly this problem—they are not sufficient. As an example, the global average RTT to google.com, which employs all of these techniques, was approximately 100 milliseconds in 2012, and unfortunately this number has not budged in the past few years.

Many existing links are already within a small constant factor (1.2~1.5) of the speed-of-light limit, and while there is still room for improvement, especially with respect to "last-mile latency," the relative gains are modest. Worse, with the rise of mobile networks, the impact of the latency bottleneck has only gotten worse. While the latest 4G mobile networks are specifically targeting low-latency data delivery, the advertised and real-world performance is still often measured in hundreds of milliseconds of overhead (see table 2 for advertised latencies in the AT&T core radio networks[2,3]).

TABLE 2 **Advertised latencies**

	LTE	HSPA+	HSPA	EDGE	GPRS
Latency	40-50 ms	100-200 ms	150-400 ms	600-750 ms	600-750 ms

If you can't get the performance-step function needed from improving the underlying links—if anything, with the rise of mobile traffic, there is a regression—then you must turn your attention to how you construct applications and tune the performance of the underlying transport protocols responsible for their delivery.

PERFORMANCE LIMITATIONS OF HTTP 1.1

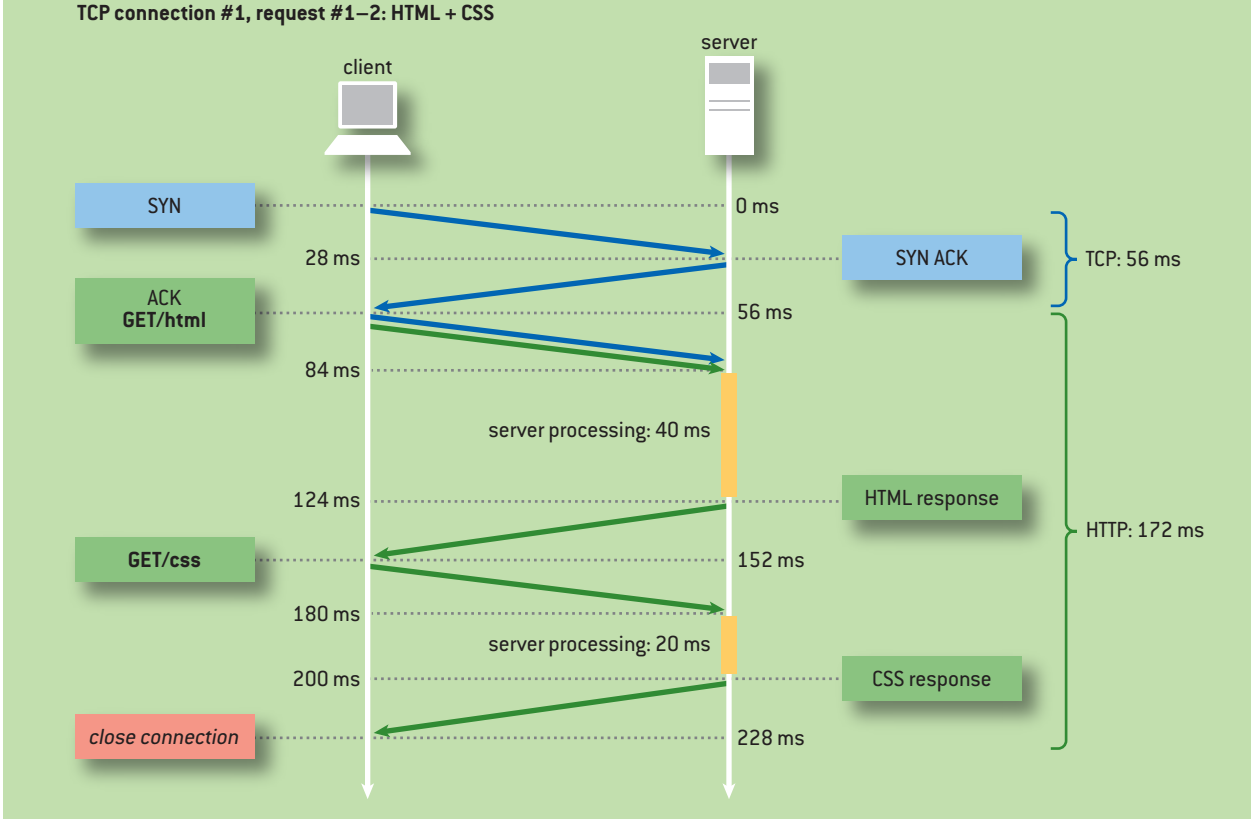
Improving the performance of HTTP was one of the key design goals for the HTTP 1.1 Working Group, and the standard introduced many critical performance enhancements. A few of the best-known include:

- Persistent connections to allow connection reuse.
- Chunked transfer encoding to allow response streaming.
- Request pipelining to allow parallel request processing.
- Byte serving to allow range-based resource requests.
- Improved and much better-specified caching mechanisms.

Unfortunately, some HTTP 1.1 features such as request pipelining have effectively failed due to lack of support and deployment challenges; while some browsers today support pipelining as an optional feature, few if any have it enabled by default. As a result, HTTP 1.1 forces strict request queuing on the client (figure 1): the client dispatches the request and must wait until the response

FIGURE 1

HTTP 1.1 Forces Strict Request Queuing on the Client



is returned by the server, which means that a single large transfer or a slow dynamic resource can block the entire connection. Worse, the browser has no way of reliably predicting this behavior and, as a result, is often forced to rely on heuristics to guess whether it should wait and attempt to reuse the existing connection or open another one.

In light of the limitations of HTTP 1.1, the Web developer community—always an inventive lot—has created and popularized a number of homebrew application workarounds (calling them *optimizations* would give them too much credit):

• Modern browsers allow up to six parallel connections per origin, which effectively allows up to six parallel resource transfers. Not satisfied with the limit of six connections, many developers decided to apply *domain sharding*, which splits site resources across different origins, thereby allowing more TCP connections. Recall that an average page now talks to 15 distinct hosts, each of which might use as many as six TCP connections.

• Small files with the same file type are often concatenated together, creating larger bundles to minimize HTTP request overhead. In effect, this is a form of multiplexing, but it is applied at the application layer—for example, CSS (Cascading Style Sheets) and JavaScript files are combined into larger bundles, small images are merged into image sprites, and so on.

• Some files are inlined directly into the HTML document to avoid the HTTP request entirely.

For many Web developers all of these are matter-of-fact optimizations—familiar, necessary, and universally accepted. Each of these workarounds, however, also often carries many negative implications for both the complexity and the performance of applications:

• Aggressive sharding often causes network congestion and is counterproductive, leading to: additional and unnecessary DNS (Domain Name Service) lookups and TCP handshakes; higher resource load caused by more sockets on client, server, and intermediaries; more network contention between parallel streams; and so on.

• Concatenation breaks the modularity of application code and has a negative impact on caching (e.g., a common practice is to concatenate all JavaScript or CSS files into large bundles, which forces download and invalidation of the entire bundle on a single byte change). Similarly, JavaScript and CSS files are parsed and executed only when the entire file is downloaded, which adds processing delays; large image sprites also occupy more memory on the client and require more resources to decode and process.

• Inlined assets cannot be cached individually and inflate the parent document. A common practice of inlining small images also inflates their size by more than 30 percent via base64 encoding and breaks request prioritization in the browser—typically, images are fetched with lower priority by the browser to accelerate page construction.

In short, many of the workarounds have serious negative performance implications. Web developers shouldn't have to worry about concatenating files, spriting images, inlining assets, or domain sharding. All of these techniques are stopgap workarounds for limitations of the HTTP 1.1 protocol. Hence, HTTP 2.0.

HTTP 2.0 DESIGN AND TECHNICAL GOALS

Developing a major revision of a protocol underlying all Web communication is a nontrivial task requiring a lot of careful thought, experimentation, and coordination. As such, it is important to define a clear technical charter and, arguably even more important, to define the boundaries of the

project. The intent is not to overhaul every detail of the protocol but to make meaningful though incremental progress to improve Web performance.

With that, the HTTPbis Working Group charter[7,8] for HTTP 2.0 is scoped as follows:

- Substantially and measurably improve end-user-perceived latency in most cases over HTTP 1.1 using TCP.
- Address the HOL (head-of-line) blocking problem in HTTP.
- Do not require multiple connections to a server to enable parallelism, thus improving its use of TCP, especially regarding congestion control.
- Retain the semantics of HTTP 1.1, leveraging existing documentation, including (but not limited to) HTTP methods, status codes, URIs, and where appropriate, header fields.
- Clearly define how HTTP 2.0 interacts with HTTP 1.x, especially in intermediaries.
- Clearly identify any new extensibility points and policy for their appropriate use.

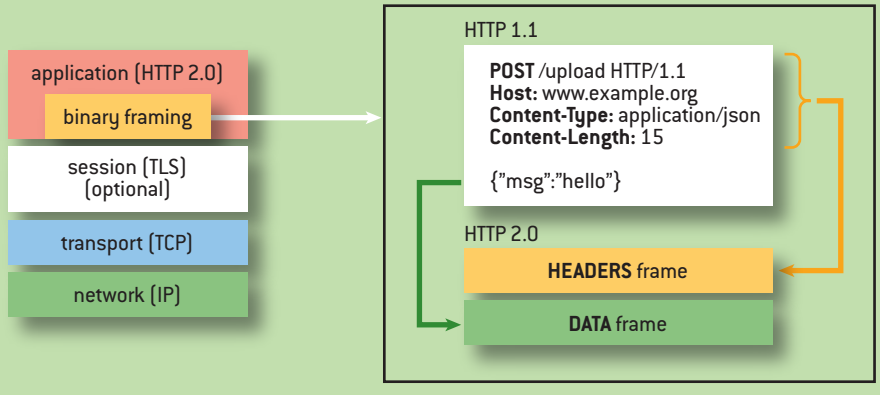
To deliver on these goals HTTP 2.0 introduces a new layering mechanism onto TCP, which addresses the well-known performance limitations of HTTP 1.x. The application semantics of HTTP remain untouched, and no changes are being made to the core concepts such as HTTP methods, status codes, URIs, and header fields—these changes are explicitly out of scope. With that in mind, let's now take a look "under the hood" of HTTP 2.0.

REQUEST AND RESPONSE MULTIPLEXING

At the core of all HTTP 2.0's performance enhancements is the new binary framing layer (figure 2), which dictates how HTTP messages are encapsulated and transferred between the client and server. HTTP semantics such as verbs, methods, and headers are unaffected, but the way they are encoded while in transit is different.

With HTTP 1.x, if the client wants to make multiple parallel requests to improve performance, multiple TCP connections are required. This behavior is a direct consequence of the newline-delimited plaintext HTTP 1.x protocol, which ensures that only one response at a time can be delivered per connection—worse, this also results in HOL blocking and inefficient use of the

FIGURE 2 **HTTP 2.0 Binary Framing**



underlying TCP connection.

The new binary framing layer in HTTP 2.0 removes these limitations and enables full request and response multiplexing. The following HTTP 2.0 terminology will help in understanding this process:

• *Stream*—a bidirectional flow of bytes, or a virtual channel, within a connection. Each stream has a relative priority value and a unique integer identifier.
• *Message*—a complete sequence of frames that maps to a logical message such as an HTTP request or a response.
• *Frame*—the smallest unit of communication in HTTP 2.0, each containing a consistent frame header, which at minimum identifies the stream to which the frame belongs, and carries a specific type of data (e.g., HTTP headers, payload, and so on).

All HTTP 2.0 communication can be performed within a single connection that can carry any number of bidirectional *streams*. In turn, each stream communicates in *messages*, which consist of one or multiple *frames*, each of which may be interleaved (figure 3) and then reassembled via the embedded stream identifier in the header of each individual frame.

The ability to break down an HTTP message into independent frames, prioritize and interleave them within a shared connection, and then reassemble them on the other end is the single most important enhancement of HTTP 2.0. By itself, this change is entirely unremarkable, since many protocols below HTTP already implement similar mechanisms. This "small" change, however, introduces a ripple effect of numerous performance benefits across the entire stack of all Web technologies, allowing developers to do the following:

• Interleave multiple requests in parallel without blocking on any one.
• Interleave multiple responses in parallel without blocking on any one.
• Use a single connection to deliver many requests and responses in parallel.
• Reduce page-load times by eliminating unnecessary latency.
• Remove unnecessary HTTP 1.x workarounds from application code.
• And much more…

BINARY FRAMING

HTTP 2.0 uses a binary, length-prefixed framing layer, which offers more compact representation than the newline-delimited plaintext HTTP 1.x protocol and is both easier and more efficient to

FIGURE 3 **Interleaved Frames from Multiple In-Flight HTTP 2.0 Streams Within a Single Connection**

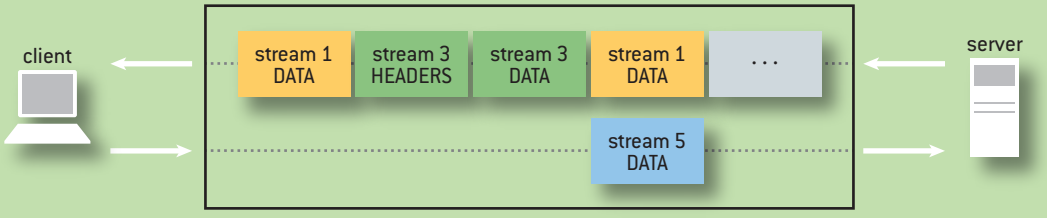
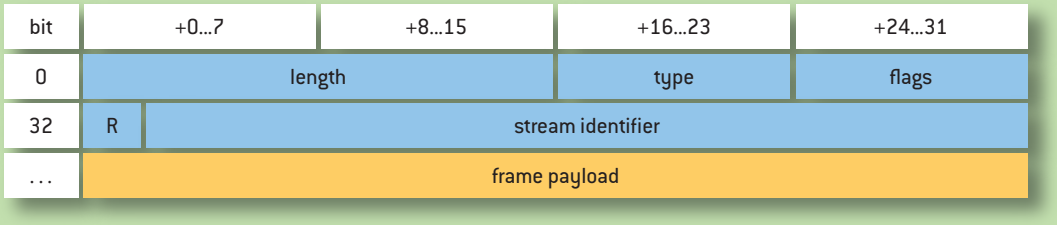


FIGURE 4 **Common Eight-Byte Frame Header**



process. All HTTP 2.0 frames share a common eight-byte header (figure 4), which contains the length of the frame, its type, a bit field for flags, and a 31-bit stream identifier.

- The 16-bit length prefix reveals that a single frame can carry 2^{16}–1 bytes of data - ~64 KB, which excludes the eight-byte header size.
- The eight-bit type field determines how the rest of the frame is interpreted.
- The eight-bit flags field allows different frame types to define frame-specific messaging flags.
- A one-bit reserved field is always set to 0.
- The 31-bit stream identifier uniquely identifies the HTTP 2.0 stream.

Given this knowledge of the shared HTTP 2.0 frame header, you can write a simple parser that can examine any HTTP 2.0 bytestream, identify different frame types, and report their flags and the length of each by examining the first eight bytes of every frame. Further, because each frame is length prefixed, the parser can skip ahead to the beginning of the next frame quickly and efficiently. This is a big performance improvement over HTTP 1.x.

Once the frame type is known, the parser can interpret the remainder of the frame. The HTTP 2.0 standard defines the types listed in table 3.

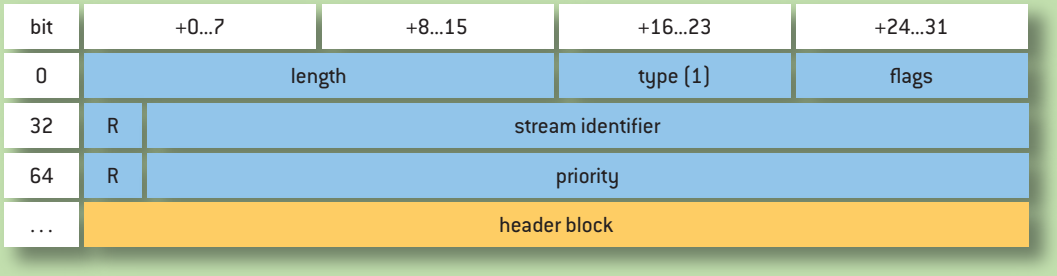
Full analysis of this taxonomy of frames is outside the scope of this discussion—after all, that's

TABLE 3 **HTTP 2.0 frame types**

DATA	used to transport HTTP message bodies
HEADERS	used to communicate additional header fields for a stream
PRIORITY	used to assign, or reassign, priority of referenced resource
RST_STREAM	used to signal abnormal termination of a stream
SETTINGS	used to signal configuration data about how two endpoints may communicate
PUSH_PROMISE	used to signal a promise to create a stream and serve referenced resource
PING	used to measure the round-trip time and perform "liveness" checks
GOAWAY	used to inform the peer to stop creating streams for current connection
WINDOW_UPDATE	used to implement flow control on per-stream or per-connection basis
CONTINUATION	used to continue a sequence of header block fragments

FIGURE 5

Headers Frame with Stream Priority and Header Payload



what the spec[7] is for (and it does a great job!). Having said that, let's go just one step further and look at the two most common workflows: initiating a new stream and exchanging application data.

INITIATING NEW HTTP 2.0 STREAMS

Before any application data can be sent, a new stream must be created and the appropriate metadata such as HTTP headers must be sent. That's what the HEADERS frame (figure 5) is for.

Notice that in addition to the common header, an optional 31-bit stream priority has been added. As a result, whenever the client initiates a new stream, it can signal to the server the relative priority of that request, and even reprioritize it later by sending another PRIORITY frame.

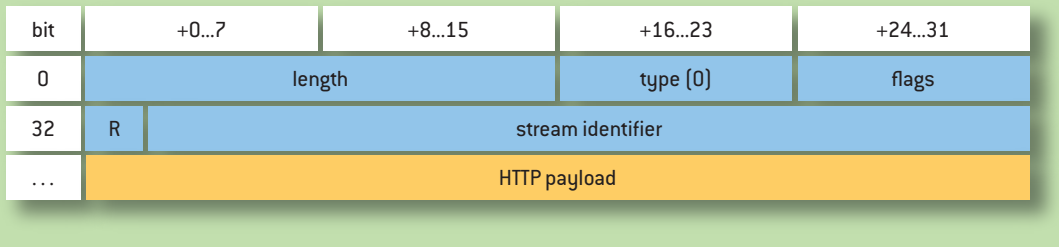
How do the client and server negotiate the unique stream IDs? They don't. Client-initiated streams have even-numbered stream IDs and server-initiated streams have odd-numbered stream IDs. This offset eliminates collisions in stream IDs between the client and server.

Finally, the HTTP header key-value pairs are encoded via a custom header compression algorithm (more on this later) to minimize the size of the payload, and they are appended to the end of the frame.

Notice that the HEADERS frames are used to communicate only the metadata about each stream. The actual application payload is delivered independently within the DATA frames (figure 6) that follow them (i.e., there is a separation between "data" and "control" messaging).

FIGURE 6

DATA Frame



The DATA frame is trivial: it's the common eight-byte header followed by the actual payload. To reduce HOL blocking, the HTTP 2.0 standard requires that each DATA frame not exceed 2^{14}-1 (16,383) bytes, which means that larger messages have to be broken up into smaller chunks. The last message in a sequence sets the END_STREAM flag to mark the end of data transfer.

There are a few more implementation details, but this information is enough to build a very basic HTTP 2.0 parser—emphasis on *very basic*. Features such as stream prioritization, server push, header compression, and flow control (not yet mentioned) warrant a bit more discussion, as they are critical to getting the best performance out of HTTP 2.0.

STREAM PRIORITIZATION

Once an HTTP message can be split into many individual frames, the exact order in which the frames are interleaved and delivered within a connection can be optimized to improve the performance of an application further. Hence, the optional 31-bit priority value: 0 represents the highest-priority stream; 2^{31}–1 represents the lowest-priority stream.

Not all resources have equal priority when rendering a page in the browser: the HTML document is, of course, critical, as it contains the structure and references to other resources; CSS is required to create the visual rendering tree (you can't paint pixels until you have the style-sheet rules); increasingly, JavaScript is also required to bootstrap the page; remaining resources such as images can be fetched with lower priority.

The good news is that all modern browsers already perform this sort of internal optimization by prioritizing different resource requests based on type of asset, their location on the page, and even learned priority from previous visits[4] (e.g., if the rendering was blocked on a certain asset in a previous visit, the same asset may be given a higher priority in the future).

With the introduction of explicit stream prioritization in HTTP 2.0, the browser can communicate these inferred priorities to the server to improve performance: the server can prioritize stream processing by controlling the allocation of resources (CPU, memory, bandwidth); and once the response data is available, the server can prioritize delivery of high-priority frames to the client. Even better, the client is now able to dispatch all of the requests as soon as they are discovered (i.e., eliminate client-side request queueing latency) instead of relying on request prioritization heuristics in light of the limited parallelism provided by HTTP 1.x.

SERVER PUSH

A powerful new feature of HTTP 2.0 is the ability of the server to send multiple replies for a single client request—that is, in addition to the response to the original request, the server can push additional resources to the client without having the client explicitly request each one.

Why would such a mechanism be needed? A typical Web application consists of dozens of resources, all of which the client discovers by examining the document provided by the server. As a result, why not eliminate the extra latency and let the server push the associated resources to the client ahead of time? The server already knows which resources the client will require—that's *server push*.

In fact, while support for server push as an HTTP protocol feature is new, many Web applications are already using it, just under a different name: *inlining*. Whenever the developer inlines an asset—CSS, JavaScript, or any other asset via a data URI—they are, in effect, pushing that resource to the

client instead of waiting for the client to request it. The only difference with HTTP 2.0 is that this workflow can now move out of the application and into the HTTP protocol itself, which offers important benefits: pushed resources can be cached by the client, declined by the client, reused across different pages, and prioritized by the server.

In effect, server push makes obsolete most of the cases where inlining is used in HTTP 1.x.

HEADER COMPRESSION

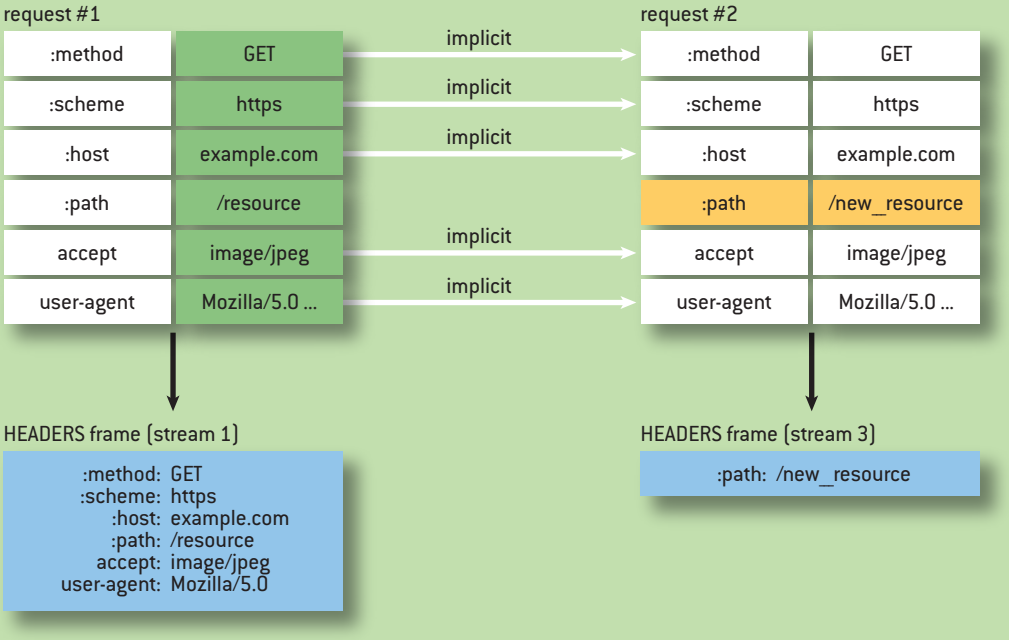
Each HTTP transfer carries a set of headers that are used to describe the transferred resource. In HTTP 1.x, this metadata is always sent as plaintext and typically adds anywhere from 500 to 800 bytes of overhead per request, and often much more if HTTP cookies are required. To reduce this overhead and improve performance, HTTP 2.0 compresses header metadata[9]:

- Instead of retransmitting the same data on each request and response, HTTP 2.0 uses header tables on both the client and server to track and store previously sent header key-value pairs.
- Header tables persist for the entire HTTP 2.0 connection and are incrementally updated by both the client and the server.
- Each new header key-value pair is either appended to the existing table or replaces a previous value in the table.

As a result, both sides of the HTTP 2.0 connection know which headers have been sent, and their previous values, which allows a new set of headers to be coded as a simple difference (figure 7) from the previous set.

Common key-value pairs that rarely change throughout the lifetime of a connection (e.g., user-

FIGURE 7 **Differential Encoding of HTTP 2.0 Headers**



agent, accept header, and so on), need to be transmitted only once. In fact, if no headers change between requests (e.g., a polling request for the same resource), then the header-encoding overhead is zero bytes—all headers are automatically inherited from the previous request.

FLOW CONTROL

Multiplexing multiple streams over the same TCP connection introduces contention for shared bandwidth resources. Stream priorities can help determine the relative order of delivery, but priorities alone are insufficient to control how resources are allocated between multiple streams. To address this, HTTP 2.0 provides a simple mechanism for stream and connection flow control:

- Flow control is hop-by-hop, not end-to-end.
- Flow control is based on window update frames: the receiver advertises how many bytes of DATA-frame payload it is prepared to receive on a stream and for the entire connection.
- Flow-control window size is updated via a WINDOW_UPDATE frame that specifies the stream ID and the window increment value.
- Flow control is directional—the receiver may choose to set any window size it desires for each stream and for the entire connection.
- Flow control can be disabled by a receiver.

As experience with TCP shows, flow control is both an art and a science. Research on better algorithms and implementation improvements are continuing to this day. With that in mind, HTTP 2.0 does not mandate any specific approach. Instead, it simply provides the necessary tools to implement such an algorithm—a great area for further research and optimization.

EFFICIENT HTTP 2.0 UPGRADE AND DISCOVERY

Though there are a lot more technical and implementation details, this whirlwind tour of HTTP 2.0 has covered the highlights: binary framing, multiplexing, prioritization, server push, header compression, and flow control. Combined, these features will deliver significant performance improvements on both the client and server.

Having said that, there is one more minor detail: how does one deploy a major revision of the HTTP protocol? The switch to HTTP 2.0 cannot happen overnight. Millions of servers must be updated to use the new binary framing protocol, and billions of clients must similarly update their browsers and networking libraries.

The good news is that most modern browsers use efficient background update mechanisms, which will enable HTTP 2.0 support quickly and with minimal intervention for a large portion of existing users. Despite this, some users will be stuck with older browsers, and servers and intermediaries will also have to be updated to support HTTP 2.0, which is a much longer labor- and capital-intensive process.

HTTP 1.x will be around for at least another decade, and most servers and clients will have to support both 1.x and 2.0 standards. As a result, an HTTP 2.0-capable client must be able to discover whether the server—and any and all intermediaries—support the HTTP 2.0 protocol when initiating a new HTTP session. There are two cases to consider:

- Initiating a new (secure) HTTPS connection via TLS.
- Initiating a new (unencrypted) HTTP connection.

In the case of a secure HTTPS connection, the new ALPN (Application Layer Protocol

FIGURE 8

HTTP Upgrade mechanism

```
GET /page HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: HTTP/2.0
HTTP2-Settings: (SETTINGS payload)

HTTP/1.1 200 OK
Content-length: 243
Content-type: text/html

(... HTTP 1.1 response ...)

          (or)

HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: HTTP/2.0

(... HTTP 2.0 response ...)
```

Negotiation[10]) extension to the TLS protocol allows users to negotiate HTTP 2.0 support as part of the regular TLS handshake: the client sends the list of protocols it supports (e.g., http/2.0); the server selects one of the advertised protocols and confirms its choice by sending the protocol name back to the client as part of the regular TLS handshake.

Establishing an HTTP 2.0 connection over a regular, nonencrypted channel requires a bit more work. Because both HTTP 1.0 and HTTP 2.0 run on the same port (80), in the absence of any other information about the server's support for HTTP 2.0, the client will have to use the HTTP Upgrade mechanism to negotiate the appropriate protocol, as shown in figure 8.

Using the Upgrade flow, if the server does not support HTTP 2.0, then it can immediately respond to the request with an HTTP 1.1 response. Alternatively, it can confirm the HTTP 2.0 upgrade by returning the "101 Switching Protocols" response in HTTP 1.1 format, and then immediately switch to HTTP 2.0 and return the response using the new binary framing protocol. In either case, no extra round-trips are incurred.

CRYSTAL GAZING

Developing a major revision of a protocol underlying all Web communication is a nontrivial task requiring a lot of careful thought, experimentation, and coordination. As such, crystal gazing for HTTP 2.0 timelines is a dangerous business—it will be ready when it's ready. Having said that, the HTTP Working Group is making rapid progress. Its past and projected milestones are as follows:

• November 2009—SPDY protocol announced by Google.
• March 2012—call for proposals for HTTP 2.0.
• September 2012—first draft of HTTP 2.0.
• July 2013—first implementation draft of HTTP 2.0.
• April 2014—Working Group last call for HTTP 2.0.
• November 2014—submit HTTP 2.0 to IESG (Internet Engineering Steering Group) as a Proposed Standard.

SPDY was an experimental protocol developed at Google and announced in mid-2009, which later formed the basis of early HTTP 2.0 drafts. Many revisions and improvements later, as of late 2013, there is now an implementation draft of the protocol, and interoperability work is in full swing—recent Interop events featured client and server implementations from Microsoft Open Technologies, Mozilla, Google, Akamai, and other contributors. In short, all signs indicate that the projected schedule is (for once) on track: 2014 should be the year for HTTP 2.0.

MAKING THE WEB (EVEN) FASTER

With HTTP 2.0 deployed far and wide, can we kick back and declare victory? The Web will be fast, right? Well, as with any performance optimization, the moment one bottleneck is removed, the next one is unlocked. There is plenty of room for further optimization:

• HTTP 2.0 eliminates HOL blocking at the application layer, but it still exists at the transport (TCP) layer. Further, now that all of the streams can be multiplexed over a single connection, tuning congestion control, mitigating bufferbloat, and all other TCP optimizations become even more critical.
• TLS is a critical and largely unoptimized frontier: we need to reduce the number of handshake round-trips, upgrade outdated clients to get wider adoption, and improve client and server performance in general.
• HTTP 2.0 opens up a new world of research opportunities for optimal implementations of header-compression strategies, prioritization, and flow-control logic both on the client and on the server, as well as the use of server push.
• All existing Web applications will continue to work over HTTP 2.0—the servers will have to be upgraded, but otherwise the transport switch is transparent. That is not to say, however, that existing and new applications can't be tuned to perform better over HTTP 2.0 by leveraging new functionality such as server push, prioritization, and so on. Web developers will have to develop new best practices, and revert and unlearn the numerous HTTP 1.1 workarounds they are using today.

In short, there is a lot more work to be done. HTTP 2.0 is a significant milestone that will help make the Web faster, but it is not the end of the journey.

REFERENCES

1. Akamai. 2013. State of the Internet; http://www.akamai.com/stateoftheinternet/.
2. AT&T. 2013. Average speeds for AT&T LaptopConnect Devices; http://www.att.com/esupport/article.jsp?sid=64785&cv=820&_requestid=733221#fbid=vttq9CyA2iG.
3. AT&T. 2012. Best Practices for 3G and 4G App Development; http://developer.att.com/home/develop/referencesandtutorials/whitepapers/BestPracticesFor3Gand4GAppDevelopment.pdf.

4. Belshe, M. 2010. More bandwidth doesn't matter (much); https://docs.google.com/a/chromium.org/viewer?a=v&pid=sites&srcid=Y2hyb21pdW0ub3JnfGRldnxneDoxMzcyOWI1N2I4YzI3NzE2.
5. Grigorik, I. 2013. High-performance networking in Google Chrome; http://www.igvita.com/posa/high-performance-networking-in-google-chrome/.
6. HTTP Archive; http://www.httparchive.org/.
7. IETF HTTPbis Working Group. 2012. Charter; http://datatracker.ietf.org/doc/charter-ietf-httpbis/.
8. IETF HTTPbis Working Group. 2013. HTTP 2.0 specifications; http://tools.ietf.org/html/draft-ietf-httpbis-http2.
9. IETF HTTPbis Working Group. 2013. HPACK-Header Compression for HTTP/2.0; http://tools.ietf.org/html/draft-ietf-httpbis-header-compression.
10. IETF Network Working Group. 2013. Transport Layer Security (TLS) Application Layer Protocol Negotiation Extension; http://tools.ietf.org/html/draft-friedl-tls-applayerprotoneg.
11. Upson, L. 2013. Google I/O 2013 keynote address; http://www.youtube.com/watch?v=9pmPa_KxsAM.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ILYA GRIGORIK is a web performance engineer and developer advocate at Google, where he works to make the Web faster by building and driving adoption of performance best practices at Google and beyond.