HTTP/2

HTTP, CHAPTER 12

HTTP/2 will make our applications faster, simpler, and more robust—a rare combination—by allowing us to undo many of the HTTP/1.1 workarounds previously done within our applications and address these concerns within the transport layer itself. Even better, it also opens up a number of entirely new opportunities to optimize our applications and improve performance!

The primary goals for HTTP/2 are to reduce latency by enabling full request and response multiplexing, minimize protocol overhead via efficient compression of HTTP header fields, and add support for request prioritization and server push. To implement these requirements, there is a large supporting cast of other protocol enhancements, such as new flow control, error handling, and upgrade mechanisms, but these are the most important features that every web developer should understand and leverage in their applications.

HTTP/2 does not modify the application semantics of HTTP in any way. All the core concepts, such as HTTP methods, status codes, URIs, and header fields, remain in place. Instead, HTTP/2 modifies how the data is formatted (framed) and transported between the client and server, both of whom manage the entire process, and hides all the complexity from our applications within the new framing layer. As a result, all existing applications can be delivered without modification. That's the good news.

However, we are not just interested in delivering a working application; our goal is to deliver the best performance! HTTP/2 enables a number of new optimizations our applications can leverage, which were previously not possible, and our job is to make the best of them. Let's take a closer look under the hood.

Why not HTTP/1.2? §

To achieve the performance goals set by the HTTP Working Group, HTTP/2 introduces a new binary framing layer that is not backward compatible with previous HTTP/1.x servers and clients—hence the major protocol version increment to HTTP/2.

That said, unless you are implementing a web server (or a custom client) by working with raw TCP sockets, then you won't see any difference: all the new, low-level framing is performed by the client and server on your behalf. The only observable differences will be improved performance and availability of new capabilities like request prioritization, flow control, and server push!

Brief History of SPDY and HTTP/2 §

primary goal was to try to reduce the load latency of web pages by addressing some of the well-known performance limitations of HTTP/1.1. Specifically, the outlined project goals were set as follows:

- Target a 50% reduction in page load time (PLT).
- Avoid the need for any changes to content by website authors.
- Minimize deployment complexity, avoid changes in network infrastructure.
- Develop this new protocol in partnership with the open-source community.
- Gather real performance data to (in)validate the experimental protocol.

Note

To achieve the 50% PLT improvement, SPDY aimed to make more efficient use of the underlying TCP connection by introducing a new binary framing layer to enable request and response multiplexing, prioritization, and header compression; see Latency as a Performance Bottleneck.

Not long after the initial announcement, Mike Belshe and Roberto Peon, both software engineers at Google, shared their first results, documentation, and source code for the experimental implementation of the new SPDY protocol:

> *So far we have only tested SPDY in lab conditions. The initial results are very encouraging: when we download the top 25 websites over simulated home network connections, we see a significant improvement in performance—pages loaded up to 55% faster.*
>
> *A 2x Faster Web, Chromium Blog*

Fast-forward to 2012 and the new experimental protocol was supported in Chrome, Firefox, and Opera, and a rapidly growing number of sites, both large (e.g., Google, Twitter, Facebook) and small, were deploying SPDY within their infrastructure. In effect, SPDY was on track to become a de facto standard through growing industry adoption.

Observing the above trend, the HTTP Working Group (HTTP-WG) kicked off a new effort to take the lessons learned from SPDY, build and improve on them, and deliver an official "HTTP/2" standard: a new charter was drafted, an open call for HTTP/2 proposals was made, and after a lot of discussion within the working group, the SPDY specification was adopted as a starting point for the new HTTP/2 protocol.

Over the next few years SPDY and HTTP/2 would continue to coevolve in parallel, with SPDY acting as an experimental branch that was used to test new features and proposals for the HTTP/2 standard: what looks good on paper may not work in practice, and vice versa, and SPDY

the end, this process spanned three years and resulted in a over a dozen intermediate drafts:

- March 2012: Call for proposals for HTTP/2
- November 2012: First draft of HTTP/2 (based on SPDY)
- August 2014: HTTP/2 draft-17 and HPACK draft-12 are published
- August 2014: Working Group last call for HTTP/2
- February 2015: IESG approved HTTP/2 and HPACK drafts
- May 2015: RFC 7540 (HTTP/2) and RFC 7541 (HPACK) are published

In early 2015 the IESG reviewed and approved the new HTTP/2 standard for publication. Shortly after that, the Google Chrome team announced their schedule to deprecate SPDY and NPN extension for TLS:

> *HTTP/2's primary changes from HTTP/1.1 focus on improved performance. Some key features such as multiplexing, header compression, prioritization and protocol negotiation evolved from work done in an earlier open, but non-standard protocol named SPDY. 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, and to also remove support for the TLS extension named NPN in favor of ALPN in Chrome at the same time. Server developers are strongly encouraged to move to HTTP/2 and ALPN.*
>
> *We're happy to have contributed to the open standards process that led to HTTP/2, and hope to see wide adoption given the broad industry engagement on standardization and implementation.*
>
> *Hello HTTP/2, Goodbye SPDY, Chromium Blog*

The coevolution of SPDY and HTTP/2 enabled server, browser, and site developers to gain real-world experience with the new protocol as it was being developed. As a result, the HTTP/2 standard is one of the best and most extensively tested standards right out of the gate. By the time HTTP/2 was approved by the IESG, there were dozens of thoroughly tested and production-ready client and server implementations. In fact, just weeks after the final protocol was approved, many users were already enjoying its benefits as several popular browsers (and many sites) deployed full HTTP/2 support.

Design and Technical Goals §

First versions of the HTTP protocol were intentionally designed for simplicity of implementation: HTTP/0.9 was a one-line protocol to bootstrap the World Wide Web; HTTP/1.0 documented the popular extensions to HTTP/0.9 in an informational standard; HTTP/1.1 introduced an official IETF standard; see Brief History of HTTP. As such, HTTP/0.9-1.x delivered exactly what it set out to do: HTTP is one of the most ubiquitous and widely adopted application protocols on the Internet.

HTTP/1.x clients need to use multiple connections to achieve concurrency and reduce latency; HTTP/1.x does not compress request and response headers, causing unnecessary network traffic; HTTP/1.x does not allow effective resource prioritization, resulting in poor use of the underlying TCP connection; and so on.

These limitations were not fatal, but as the web applications continued to grow in their scope, complexity, and importance in our everyday lives, they imposed a growing burden on both the developers and users of the web, which is the exact gap that HTTP/2 was designed to address:

> *HTTP/2 enables a more efficient use of network resources and a reduced perception of latency by introducing header field compression and allowing multiple concurrent exchanges on the same connection… Specifically, it allows interleaving of request and response messages on the same connection and uses an efficient coding for HTTP header fields. It also allows prioritization of requests, letting more important requests complete more quickly, further improving performance.*
>
> *The resulting protocol is more friendly to the network, because fewer TCP connections can be used in comparison to HTTP/1.x. This means less competition with other flows, and longer-lived connections, which in turn leads to better utilization of available network capacity. Finally, HTTP/2 also enables more efficient processing of messages through use of binary message framing.*
>
> *Hypertext Transfer Protocol version 2, Draft 17*

It is important to note that HTTP/2 is extending, not replacing, the previous HTTP standards. The application semantics of HTTP are the same, and no changes were made to the offered functionality or core concepts such as HTTP methods, status codes, URIs, and header fields—these changes were explicitly out of scope for the HTTP/2 effort. That said, while the high-level API remains the same, it is important to understand how the low-level changes address the performance limitations of the previous protocols. Let's take a brief tour of the binary framing layer and its features.

Binary Framing Layer §

At the core of all performance enhancements of HTTP/2 is the new *binary framing layer* (Figure 12-1), which dictates how the HTTP messages are encapsulated and transferred between the client and server.

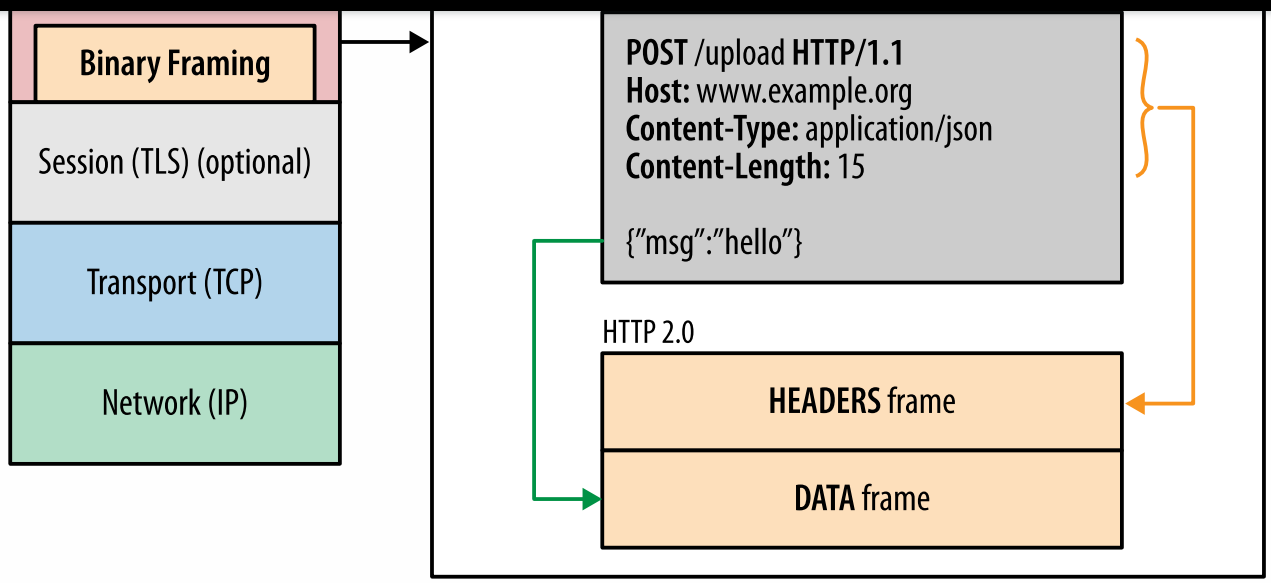

Figure 12-1. HTTP/2 binary framing layer

The "layer" refers to a design choice to introduce a new optimized encoding mechanism between the socket interface and the higher HTTP API exposed to our applications: the HTTP semantics, such as verbs, methods, and headers, are unaffected, but the way they are encoded while in transit is what's different. Unlike the newline delimited plaintext HTTP/1.x protocol, all HTTP/2 communication is split into smaller messages and frames, each of which is encoded in binary format.

As a result, both client and server must use the new binary encoding mechanism to understand each other: an HTTP/1.x client won't understand an HTTP/2 only server, and vice versa. Thankfully, our applications remain blissfully unaware of all these changes, as the client and server perform all the necessary framing work on our behalf.

The Pros and Cons of Binary Protocols §

ASCII protocols are easy to inspect and get started with. However, they are not very efficient and are typically harder to implement correctly: optional whitespace, varying termination sequences, and other quirks make it hard to distinguish the protocol from the payload and lead to parsing and security errors. By contrast, while binary protocols may take more effort to get started with, they tend to lead to more performant, robust, and provably correct implementations.

HTTP/2 uses binary framing. As a result, you will need a tool that understands it to inspect and debug the protocol—e.g., Wireshark or equivalent. In practice, this is less of an issue than it seems, since you would have to use the same tools to inspect the encrypted TLS flows—which also rely on binary framing (see TLS Record Protocol)—carrying HTTP/1.x and HTTP/2 data.

Streams, Messages, and Frames §

(Figure 12-2) between the client and server. To describe this process, let's familiarize ourselves with the HTTP/2 terminology:

Stream

A bidirectional flow of bytes within an established connection, which may carry one or more messages.

Message

A complete sequence of frames that map to a logical request or response message.

Frame

The smallest unit of communication in HTTP/2, each containing a frame header, which at a minimum identifies the stream to which the frame belongs.

- All *communication* is performed over a single TCP connection that can carry any number of bidirectional streams.
- Each *stream* has a unique identifier and optional priority information that is used to carry bidirectional messages.
- Each *message* is a logical HTTP message, such as a request, or response, which consists of one or more frames.
- The *frame* is the smallest unit of communication that carries a specific type of data—e.g., HTTP headers, message payload, and so on. Frames from different streams may be interleaved and then reassembled via the embedded stream identifier in the header of each frame.

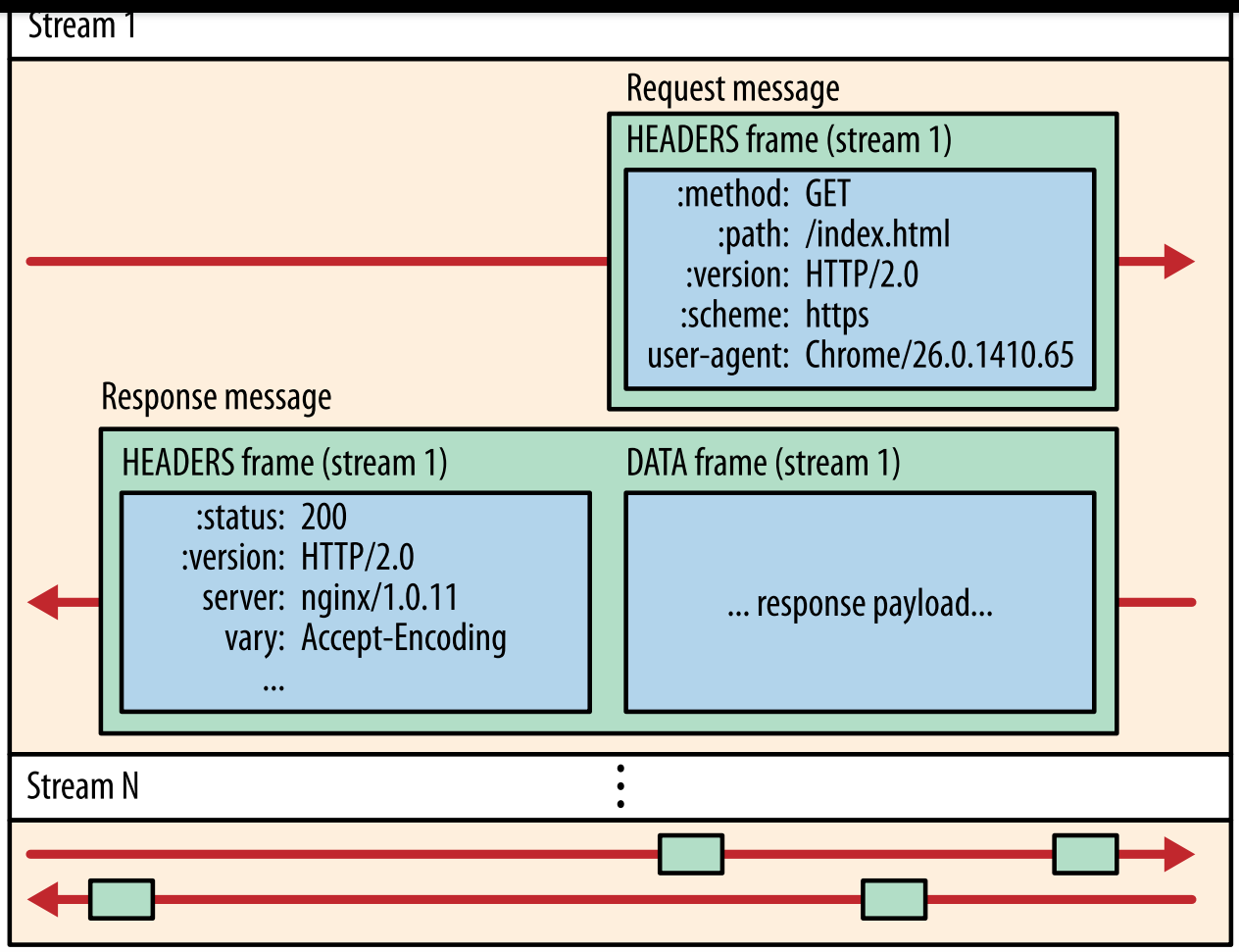

Figure 12-2. HTTP/2 streams, messages, and frames

In short, HTTP/2 breaks down the HTTP protocol communication into an exchange of binary-encoded frames, which are then mapped to messages that belong to a particular stream, and all of which are multiplexed within a single TCP connection. This is the foundation that enables all other features and performance optimizations provided by the HTTP/2 protocol.

Request and Response Multiplexing §

With HTTP/1.x, if the client wants to make multiple parallel requests to improve performance, then multiple TCP connections must be used; see Using Multiple TCP Connections. This behavior is a direct consequence of the HTTP/1.x delivery model, which ensures that only one response can be delivered at a time (response queuing) per connection. Worse, this also results in head-of-line blocking and inefficient use of the underlying TCP connection.

The new binary framing layer in HTTP/2 removes these limitations, and enables full request and response multiplexing, by allowing the client and server to break down an HTTP message into independent frames (Figure 12-3), interleave them, and then reassemble them on the other end.

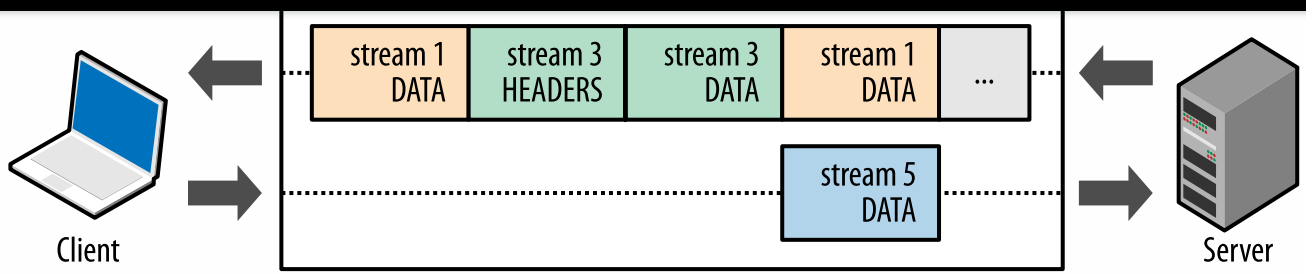

Figure 12-3. HTTP/2 request and response multiplexing within a shared connection

The snapshot in Figure 12-3 captures multiple streams in flight within the same connection: the client is transmitting a DATA frame (stream 5) to the server, while the server is transmitting an interleaved sequence of frames to the client for streams 1 and 3. As a result, there are three parallel streams in flight!

The ability to break down an HTTP message into independent frames, interleave them, and then reassemble them on the other end is the single most important enhancement of HTTP/2. In fact, it introduces a ripple effect of numerous performance benefits across the entire stack of all web technologies, enabling us to:

- 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 multiple requests and responses in parallel
- Remove unnecessary HTTP/1.x workarounds (see Optimizing for HTTP/1.x), such as concatenated files, image sprites, and domain sharding
- Deliver lower page load times by eliminating unnecessary latency and improving utilization of available network capacity
- *And much more…*

The new binary framing layer in HTTP/2 resolves the head-of-line blocking problem found in HTTP/1.x and eliminates the need for multiple connections to enable parallel processing and delivery of requests and responses. As a result, this makes our applications faster, simpler, and cheaper to deploy.

Stream Prioritization §

Once an HTTP message can be split into many individual frames, and we allow for frames from multiple streams to be multiplexed, the order in which the frames are interleaved and delivered both by the client and server becomes a critical performance consideration. To facilitate this, the HTTP/2 standard allows each stream to have an associated weight and dependency:

- Each stream may be assigned an integer weight between 1 and 256

The combination of stream dependencies and weights allows the client to construct and communicate a "prioritization tree" (Figure 12-4) that expresses how it would prefer to receive the responses. In turn, the server can use this information to prioritize stream processing by controlling the allocation of CPU, memory, and other resources, and once the response data is available, allocation of bandwidth to ensure optimal delivery of high-priority responses to the client.

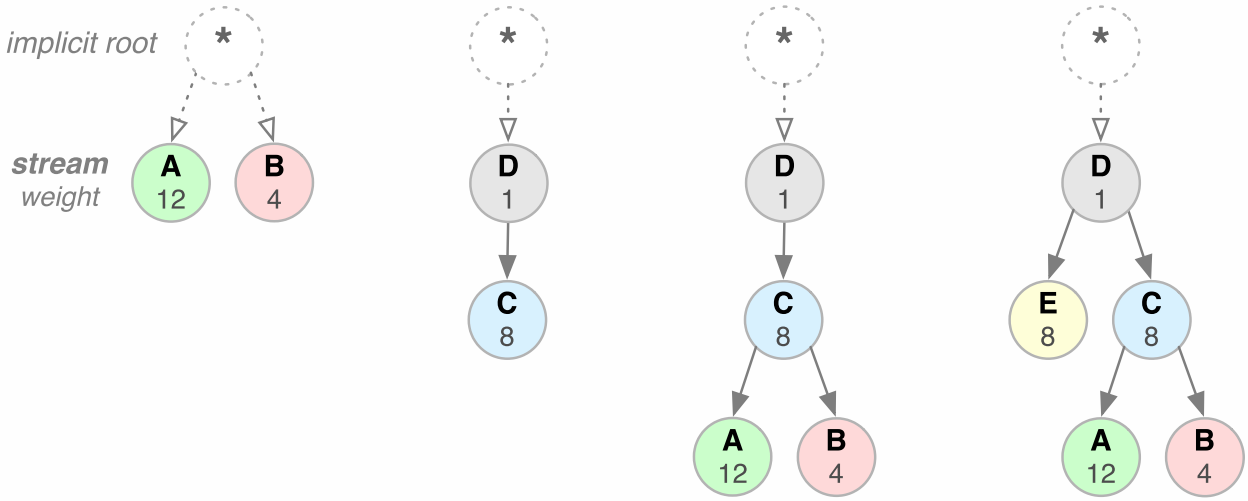

Figure 12-4. HTTP/2 stream dependencies and weights

A stream dependency within HTTP/2 is declared by referencing the unique identifier of another stream as its parent; if omitted the stream is said to be dependent on the "root stream". Declaring a stream dependency indicates that, if possible, the parent stream should be allocated resources ahead of its dependencies—e.g., please process and deliver response D before response C.

Streams that share the same parent (i.e., sibling streams) should be allocated resources in proportion to their weight. For example, if stream A has a weight of 12 and its one sibling B has a weight of 4, then to determine the proportion of the resources that each of these streams should receive:

1. Sum all the weights: $4 + 12 = 16$
2. Divide each stream weight by the total weight: $A = 12/16$, $B = 4/16$

Thus, stream A should receive three-quarters and stream B should receive one-quarter of available resources; stream B should receive one-third of the resources allocated to stream A. Let's work through a few more hands-on examples in Figure 12-4. From left to right:

1. Neither stream A nor B specify a parent dependency and are said to be dependent on the implicit "root stream"; A has a weight of 12, and B has a weight of 4. Thus, based on proportional weights: stream B should receive one-third of the resources allocated to stream A.

allocation of resources ahead of C. The weights are inconsequential because C's dependency communicates a stronger preference.

3. D should receive full allocation of resources ahead of C; C should receive full allocation of resources ahead of A and B; stream B should receive one-third of the resources allocated to stream A.
4. D should receive full allocation of resources ahead of E and C; E and C should receive equal allocation ahead of A and B; A and B should receive proportional allocation based on their weights.

As the above examples illustrate, the combination of stream dependencies and weights provides an expressive language for resource prioritization, which is a critical feature for improving browsing performance where we have many resource types with different dependencies and weights. Even better, the HTTP/2 protocol also allows the client to update these preferences at any point, which enables further optimizations in the browser—e.g., we can change dependencies and reallocate weights in response to user interaction and other signals.

Note

Stream dependencies and weights express a transport preference, not a requirement, and as such do not guarantee a particular processing or transmission order. That is, the client cannot force the server to process the stream in particular order using stream prioritization. While this may seem counterintuitive, it is in fact the desired behavior: we do not want to block the server from making progress on a lower priority resource if a higher priority resource is blocked.

Browser Request Prioritization and HTTP/2 §

Not all resources have equal priority when rendering a page in the browser: the HTML document itself is critical to construct the DOM; the CSS is required to construct the CSSOM; both DOM and CSSOM construction can be blocked on JavaScript resources (see DOM, CSSOM, and JavaScript); and remaining resources, such as images, are often fetched with lower priority.

To accelerate the load time of the page, all modern browsers prioritize requests based on type of asset, their location on the page, and even learned priority from previous visits—e.g., if the rendering was blocked on a certain asset in a previous visit, then the same asset may be prioritized higher in the future.

With HTTP/1.x, the browser has limited ability to leverage above priority data: the protocol does not support multiplexing, and there is no way to communicate request priority to the server. Instead, it must rely on the use of parallel connections, which enables limited parallelism of up to six requests per origin. As a result, requests are queued on the client until a connection is available, which adds unnecessary network latency. In theory, HTTP Pipelining tried to partially address this problem, but in practice it has failed to gain adoption.

can communicate its stream prioritization preference via stream dependencies and weights, allowing the server to further optimize response delivery.

One Connection Per Origin §

With the new binary framing mechanism in place, HTTP/2 no longer needs multiple TCP connections to multiplex streams in parallel; each stream is split into many frames, which can be interleaved and prioritized. As a result, all HTTP/2 connections are persistent, and only one connection per origin is required, which offers numerous performance benefits.

> *For both SPDY and HTTP/2 the killer feature is arbitrary multiplexing on a single well congestion controlled channel. It amazes me how important this is and how well it works. One great metric around that which I enjoy is the fraction of connections created that carry just a single HTTP transaction (and thus make that transaction bear all the overhead). For HTTP/1 74% of our active connections carry just a single transaction—persistent connections just aren't as helpful as we all want. But in HTTP/2 that number plummets to 25%. That's a huge win for overhead reduction.*
>
> *HTTP/2 is Live in Firefox, Patrick McManus*

Most HTTP transfers are short and bursty, whereas TCP is optimized for long-lived, bulk data transfers. By reusing the same connection HTTP/2 is able to both make more efficient use of each TCP connection, and also significantly reduce the overall protocol overhead. Further, the use of fewer connections reduces the memory and processing footprint along the full connection path (i.e., client, intermediaries, and origin servers), which reduces the overall operational costs and improves network utilization and capacity. As a result, the move to HTTP/2 should not only reduce the network latency, but also help improve throughput and reduce the operational costs.

Note

Reduced number of connections is a particularly important feature for improving performance of HTTPS deployments: this translates to fewer expensive TLS handshakes, better session reuse, and an overall reduction in required client and server resources.

Packet Loss, High-RTT Links, and HTTP/2 Performance §

Wait, I hear you say, we listed the benefits of using one TCP connection per origin but aren't there some potential downsides? Yes, there are.

- We have eliminated head-of-line blocking from HTTP, but there is still head-of-line blocking at the TCP level (see Head-of-Line Blocking).

- When packet loss occurs, the TCP congestion window size is reduced (see Congestion Avoidance), which reduces the maximum throughput of the entire connection.

Each of the items in this list may adversely affect both the throughput and latency performance of an HTTP/2 connection. However, despite these limitations, the move to multiple connections would result in its own performance tradeoffs:

- Less effective header compression due to distinct compression contexts
- Less effective request prioritization due to distinct TCP streams
- Less effective utilization of each TCP stream and higher likelihood of congestion due to more competing flows
- Increased resource overhead due to more TCP flows

The above pros and cons are not an exhaustive list, and it is always possible to construct specific scenarios where either one or more connections may prove to be beneficial. However, the experimental evidence of deploying HTTP/2 in the wild showed that a single connection is the preferred deployment strategy:

> *In tests so far, the negative effects of head-of-line blocking (especially in the presence of packet loss) is outweighed by the benefits of compression and prioritization.*
>
> *Hypertext Transfer Protocol version 2, Draft 2*

As with all performance optimization processes, the moment you remove one performance bottleneck, you unlock the next one. In the case of HTTP/2, TCP may be it. Which is why, once again, a well-tuned TCP stack on the server is such a critical optimization criteria for HTTP/2.

There is ongoing research to address these concerns and to improve TCP performance in general: TCP Fast Open, Proportional Rate Reduction, increased initial congestion window, and more. Having said that, it is important to acknowledge that HTTP/2, like its predecessors, does not mandate the use of TCP. Other transports, such as UDP, are not outside the realm of possibility as we look into the future.

Flow Control §

Flow control is a mechanism to prevent the sender from overwhelming the receiver with data it may not want or be able to process: the receiver may be busy, under heavy load, or may only be willing to allocate a fixed amount of resources for a particular stream. For example, the client may have requested a large video stream with high priority, but the user has paused the video and the client now wants to pause or throttle its delivery from the server to avoid fetching and buffering unnecessary data. Alternatively, a proxy server may have a fast downstream and slow upstream connections and similarly wants to regulate how quickly the downstream delivers data to match the speed of upstream to control its resource usage; and so on.

effectively identical—see Flow Control. However, because the HTTP/2 streams are multiplexed within a single TCP connection, TCP flow control is both not granular enough, and does not provide the necessary application-level APIs to regulate the delivery of individual streams. To address this, HTTP/2 provides a set of simple building blocks that allow the client and server to implement their own stream- and connection-level flow control:

- Flow control is directional. Each receiver may choose to set any window size that it desires for each stream and the entire connection.
- Flow control is credit-based. Each receiver advertises its initial connection and stream flow control window (in bytes), which is reduced whenever the sender emits a `DATA` frame and incremented via a `WINDOW_UPDATE` frame sent by the receiver.
- Flow control cannot be disabled. When the HTTP/2 connection is established the client and server exchange `SETTINGS` frames, which set the flow control window sizes in both directions. The default value of the flow control window is set to 65,535 bytes, but the receiver can set a large maximum window size ($2^{31} - 1$ bytes) and maintain it by sending a `WINDOW_UPDATE` frame whenever any data is received.
- Flow control is hop-by-hop, not end-to-end. That is, an intermediary can use it to control resource use and implement resource allocation mechanisms based on own criteria and heuristics.

HTTP/2 does not specify any particular algorithm for implementing flow control. Instead, it provides the simple building blocks and defers the implementation to the client and server, which can use it to implement custom strategies to regulate resource use and allocation, as well as implement new delivery capabilities that may help improve both the real and perceived performance (see Speed, Performance, and Human Perception) of our web applications.

For example, application-layer flow control allows the browser to fetch only a part of a particular resource, put the fetch on hold by reducing the stream flow control window down to zero, and then resume it later—e.g., fetch a preview or first scan of an image, display it and allow other high priority fetches to proceed, and resume the fetch once more critical resources have finished loading.

Server Push §

Another powerful new feature of HTTP/2 is the ability of the server to send multiple responses 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 (Figure 12-5), without the client having to request each one explicitly!

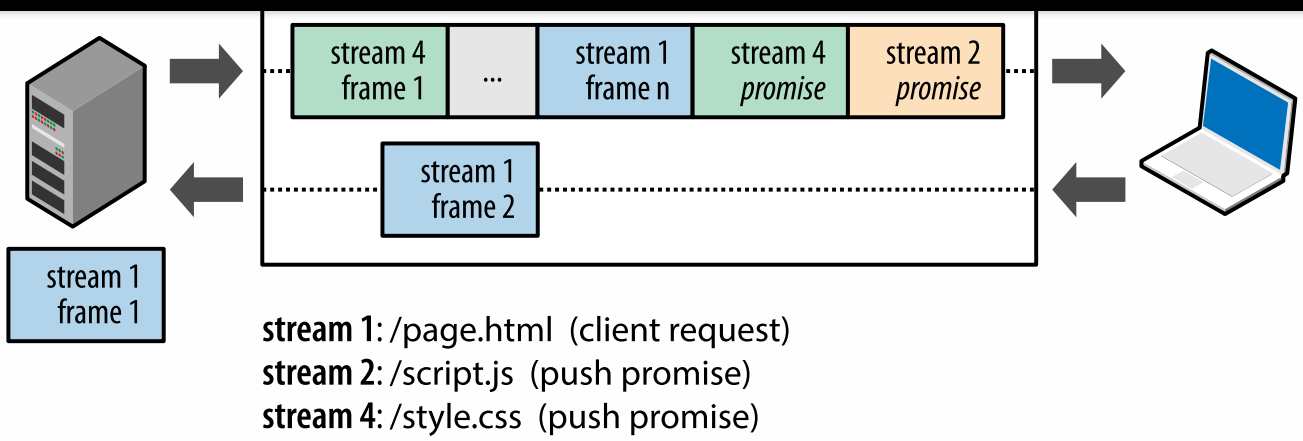

Figure 12-5. Server initiates new streams (promises) for push resources

Note

HTTP/2 breaks away from the strict request-response semantics and enables one-to-many and server-initiated push workflows that open up a world of new interaction possibilities both within and outside the browser. This is an enabling feature that will have important long-term consequences both for how we think about the protocol, and where and how it is used.

Why would we need such a mechanism in a browser? A typical web application consists of dozens of resources, all of which are discovered by the client 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 ahead of time? The server already knows which resources the client will require; that's server push.

In fact, if you have ever inlined a CSS, JavaScript, or any other asset via a data URI (see Resource Inlining), then you already have hands-on experience with server push! By manually inlining the resource into the document, we are, in effect, pushing that resource to the client, without waiting for the client to request it. With HTTP/2 we can achieve the same results, but with additional performance benefits:

- Pushed resources can be cached by the client
- Pushed resources can be reused across different pages
- Pushed resources can be multiplexed alongside other resources
- Pushed resources can be prioritized by the server
- Pushed resources can be declined by the client

Each pushed resource is a stream that, unlike an inlined resource, allows it to be individually multiplexed, prioritized, and processed by the client. The only security restriction, as enforced by the browser, is that pushed resources must obey the same-origin policy: the server must be authoritative for the provided content.

All server push streams are initiated via PUSH_PROMISE frames, which signal the server's intent to push the described resources to the client and need to be delivered ahead of the response data that requests the pushed resources. This delivery order is critical: the client needs to know which resources the server intends to push to avoid creating own and duplicate requests for these resources. The simplest strategy to satisfy this requirement is to send all PUSH_PROMISE frames, which contain just the HTTP headers of the promised resource, ahead of the parent's response (i.e., DATA frames).

Once the client receives a PUSH_PROMISE frame it has the option to decline the stream (via a RST_STREAM frame) if it wants to (e.g., the resource is already in cache), which is an important improvement over HTTP/1.x. By contrast, the use of resource inlining, which is a popular "optimization" for HTTP/1.x, is equivalent to a "forced push": the client cannot opt-out, cancel it, or process the inlined resource individually.

With HTTP/2 the client remains in full control of how server push is used. The client can limit the number of concurrently pushed streams; adjust the initial flow control window to control how much data is pushed when the stream is first opened; disable server push entirely. These preferences are communicated via the SETTINGS frames at the beginning of the HTTP/2 connection and may be updated at any time.

Header Compression §

Each HTTP transfer carries a set of headers that describe the transferred resource and its properties. In HTTP/1.x, this metadata is always sent as plain text and adds anywhere from 500–800 bytes of overhead per transfer, and sometimes kilobytes more if HTTP cookies are being used; see Measuring and Controlling Protocol Overhead. To reduce this overhead and improve performance, HTTP/2 compresses request and response header metadata using the HPACK compression format that uses two simple but powerful techniques:

1. It allows the transmitted header fields to be encoded via a static Huffman code, which reduces their individual transfer size.
2. It requires that both the client and server maintain and update an indexed list of previously seen header fields (i.e., establishes a shared compression context), which is then used as a reference to efficiently encode previously transmitted values.

Huffman coding allows the individual values to be compressed when transferred, and the indexed list of previously transferred values allows us to encode duplicate values (Figure 12-6) by transferring index values that can be used to efficiently look up and reconstruct the full header keys and values.

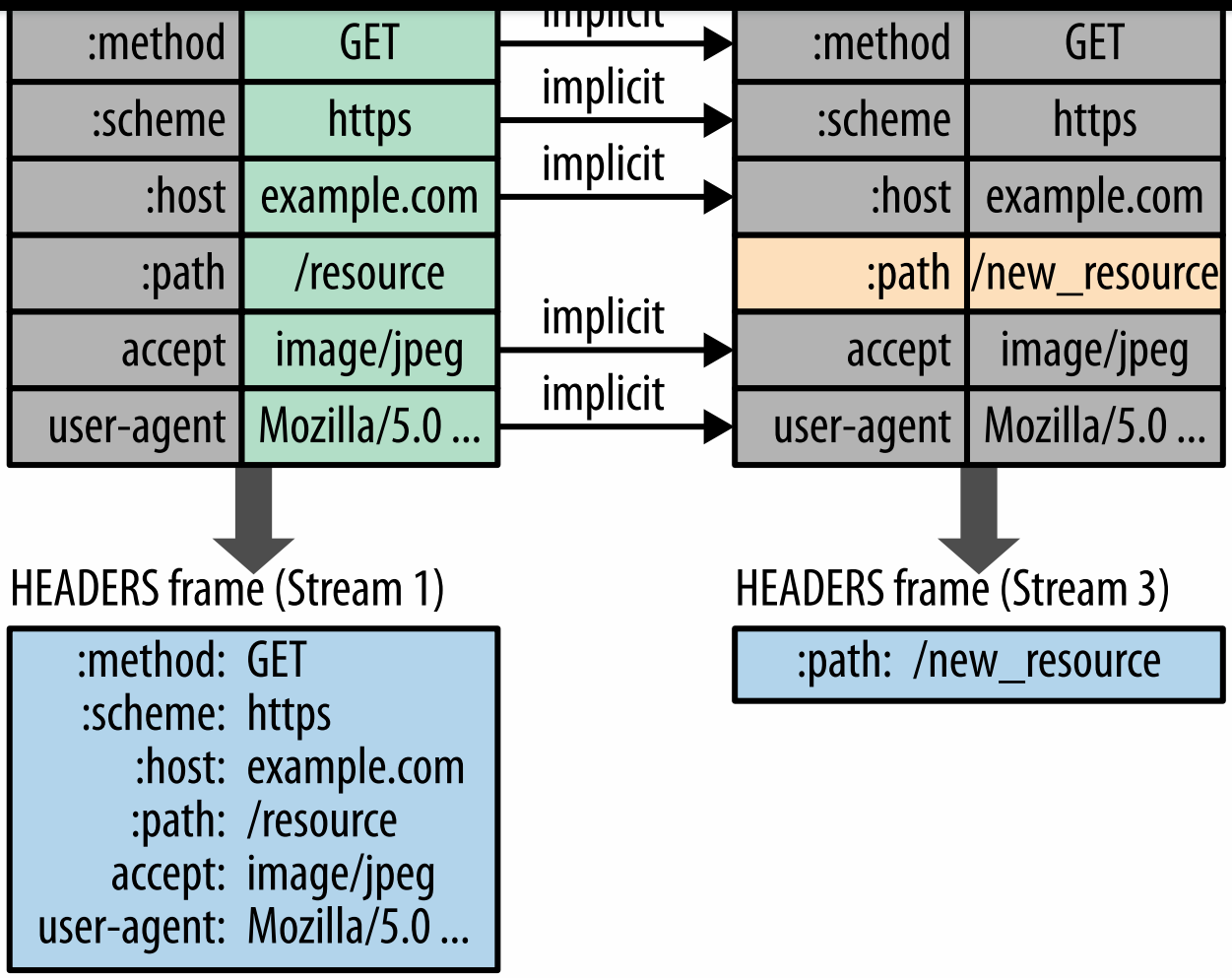

Figure 12-6. HPACK: Header Compression for HTTP/2

As one further optimization, the HPACK compression context consists of a static and dynamic tables: the static table is defined in the specification and provides a list of common HTTP header fields that all connections are likely to use (e.g., valid header names); the dynamic table is initially empty and is updated based on exchanged values within a particular connection. As a result, the size of each request is reduced by using static Huffman coding for values that haven't been seen before, and substitution of indexes for values that are already present in the static or dynamic tables on each side.

Note

The definitions of the request and response header fields in HTTP/2 remain unchanged, with a few minor exceptions: all header field names are lowercase, and the request line is now split into individual `:method`, `:scheme`, `:authority`, and `:path` pseudo-header fields.

Security and Performance of HPACK §

significant improvement in page load time latency:

> *On the lower-bandwidth DSL link, in which the upload link is only 375 Kbps, request header compression in particular, led to significant page load time improvements for certain sites (i.e., those that issued large number of resource requests). We found a reduction of 45–1142 ms in page load time simply due to header compression.*
>
> *SPDY whitepaper, chromium.org*

However, in the summer of 2012, a "CRIME" security attack was published against TLS and SPDY compression algorithms, which could result in session hijacking. As a result, the zlib compression algorithm was replaced by HPACK, which was specifically designed to: address the discovered security issues, be efficient and simple to implement correctly, and of course, enable good compression of HTTP header metadata.

For full details of the HPACK compression algorithm, see *https://tools.ietf.org/html/draft-ietf-httpbis-header-compression* ⊘ .

Upgrading to HTTP/2 §

The switch to HTTP/2 cannot happen overnight: millions of servers must be updated to use the new binary framing, and billions of clients must similarly update their networking libraries, browsers, and other applications.

The good news is, all modern browsers have committed to supporting HTTP/2, and most modern browsers use efficient background update mechanisms, which have already enabled HTTP/2 support with minimal intervention for a large proportion of existing users. That said, some users will be stuck on legacy browsers, and servers and intermediaries will also have to be updated to support HTTP/2, which is a much longer (and 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 HTTP/1.x and HTTP/2 standards. As a result, an HTTP/2 client and server must be able to discover and negotiate which protocol will be used prior to exchanging application data. To address this, the HTTP/2 protocol defines the following mechanisms:

1. Negotiating HTTP/2 via a secure connection with TLS and ALPN
2. Upgrading a plaintext connection to HTTP/2 without prior knowledge
3. Initiating a plaintext HTTP/2 connection with prior knowledge

The HTTP/2 standard does not require use of TLS, but in practice it is the most reliable way to deploy a new protocol in the presence of large number of existing intermediaries; see Proxies, Intermediaries, TLS, and New Protocols on the Web. As a result, the use of TLS and ALPN is the recommended mechanism to deploy and negotiate HTTP/2: the client and server negotiate the desired protocol as part of the TLS handshake without adding any extra latency or roundtrips;

constraint, while all popular browsers have committed to supporting HTTP/2 over TLS, some have also indicated that they will *only* enable HTTP/2 over TLS—e.g., Firefox and Google Chrome. As a result, TLS with ALPN negotiation is a de facto requirement for enabling HTTP/2 in the browser.

Establishing an HTTP/2 connection over a regular, non-encrypted channel is still possible, albeit perhaps not with a popular browser, and with some additional complexity. Because both HTTP/1.x and HTTP/2 run on the same port (80), in absence of any other information about server support for HTTP/2, the client has to use the *HTTP Upgrade* mechanism to negotiate the appropriate protocol:

```
GET /page HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c (1)
HTTP2-Settings: (SETTINGS payload) (2)

HTTP/1.1 200 OK (3)
Content-length: 243
Content-type: text/html

(... HTTP/1.1 response ...)

          (or)

HTTP/1.1 101 Switching Protocols (4)
Connection: Upgrade
Upgrade: h2c

(... HTTP/2 response ...)
```

1. Initial HTTP/1.1 request with HTTP/2 upgrade header
2. Base64 URL encoding of HTTP/2 SETTINGS payload
3. Server declines upgrade, returns response via HTTP/1.1
4. Server accepts HTTP/2 upgrade, switches to new framing

Using the preceding `Upgrade` flow, if the server does not support HTTP/2, then it can immediately respond to the request with HTTP/1.1 response. Alternatively, it can confirm the HTTP/2 upgrade by returning the `101 Switching Protocols` response in HTTP/1.1 format and then immediately switch to HTTP/2 and return the response using the new binary framing protocol. In either case, no extra roundtrips are incurred.

Finally, if the client chooses to, it may also remember or obtain the information about HTTP/2 support through some other means—e.g., DNS record, manual configuration, and so on—instead of having to rely on the `Upgrade` workflow. Armed with this knowledge, it may choose to send HTTP/2 frames right from the start, over an unencrypted channel, and hope for the best. In the

a TLS tunnel with ALPN negotiation.

Note

Secure communication between client and server, server to server, and all other permutations, is a security best practice: all in-transit data should be encrypted, authenticated, and checked against tampering. In short, use TLS with ALPN negotiation to deploy HTTP/2.

Brief Introduction to Binary Framing §

At the core of all HTTP/2 improvements is the new binary, length-prefixed framing layer. Compared with the newline-delimited plaintext HTTP/1.x protocol, binary framing offers more compact representation that is both more efficient to process and easier to implement correctly.

Once an HTTP/2 connection is established, the client and server communicate by exchanging *frames*, which serve as the smallest unit of communication within the protocol. All frames share a common 9-byte header (Figure 12-7), which contains the length of the frame, its type, a bit field for flags, and a 31-bit stream identifier.

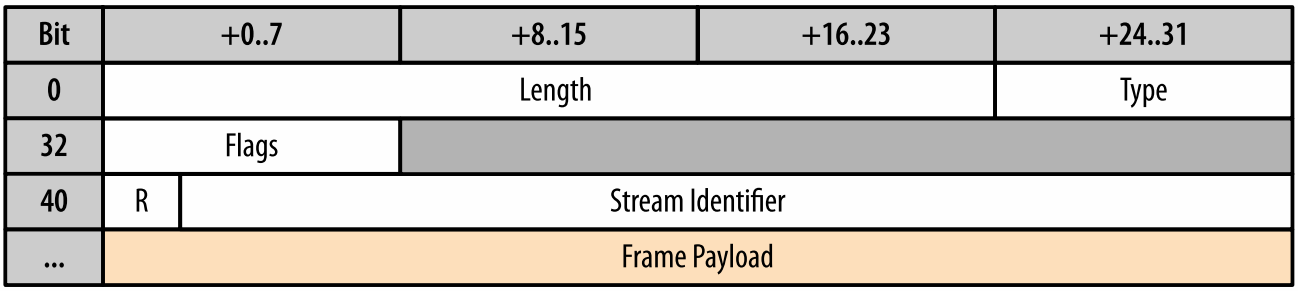

Figure 12-7. Common 9-byte frame header

- The 24-bit length field allows a single frame to carry up to 2^{24} bytes of data.
- The 8-bit type field determines the format and semantics of the frame.
- The 8-bit flags field communicates frame-type specific boolean flags.
- The 1-bit reserved field is always set to 0.
- The 31-bit stream identifier uniquely identifies the HTTP/2 stream.

Note

Technically, the `Length` field allows payloads of up to 2^{24} bytes (~16MB) per frame. However, the HTTP/2 standard sets the default maximum payload size of DATA frames to 2^{14} bytes (~16KB) per frame and allows the client and server to negotiate the higher value. Bigger is not always better: smaller frame size enables efficient multiplexing and minimizes head-of-line blocking.

can examine any HTTP/2 bytestream and identify different frame types, report their flags, and report the length of each by examining the first nine bytes of every frame. Further, because each frame is length-prefixed, the parser can skip ahead to the beginning of the next frame both quickly and efficiently—a big performance improvement over HTTP/1.x.

Once the frame type is known, the remainder of the frame can be interpreted by the parser. The HTTP/2 standard defines the following types:

DATA
: Used to transport HTTP message bodies

HEADERS
: Used to communicate header fields for a stream

PRIORITY
: Used to communicate sender-advised priority of a stream

RST_STREAM
: Used to signal termination of a stream

SETTINGS
: Used to communicate configuration parameters for the connection

PUSH_PROMISE
: Used to signal a promise to serve the referenced resource

PING
: Used to measure the roundtrip time and perform "liveness" checks

GOAWAY
: Used to inform the peer to stop creating streams for current connection

WINDOW_UPDATE
: Used to implement flow stream and connection flow control

CONTINUATION
: Used to continue a sequence of header block fragments

Note

You will need some tooling to inspect the low-level HTTP/2 frame exchange. Your favorite hex viewer is, of course, an option. Or, for a more human-friendly representation, you can use a tool like Wireshark, which understands the HTTP/2 protocol and can capture, decode, and analyze the exchange.

relevant to server and client implementers, who will need to worry about the semantics of flow control, error handling, connection termination, and other details. The application layer features and semantics of the HTTP protocol remain unchanged: the client and server take care of the framing, multiplexing, and other details, while the application can enjoy the benefits of faster and more efficient delivery.

Having said that, even though the framing layer is hidden from our applications, it is useful for us to go just one step further and look at the two most common workflows: initiating a new stream and exchanging application data. Having an intuition for how a request, or a response, is translated into individual frames will give you the necessary knowledge to debug and optimize your HTTP/2 deployments. Let's dig a little deeper.

Fixed vs. Variable Length Fields and HTTP/2 §

HTTP/2 uses fixed-length fields exclusively. The overhead of an HTTP/2 frame is low (9-byte header for a data frame), and variable-length encoding savings do not offset the required complexity for the parsers, nor do they have a significant impact on the used bandwidth or latency of the exchange.

For example, if variable length encoding could reduce the overhead by 50%, for a 1,400-byte network packet, this would amount to just 4 saved bytes (0.3%) for a single frame.

Initiating a New Stream §

Before any application data can be sent, a new stream must be created and the appropriate request metadata must be sent: optional stream dependency and weight, optional flags, and the HPACK-encoded HTTP request headers describing the request. The client initiates this process by sending a HEADERS frame (Figure 12-8) with all of the above.

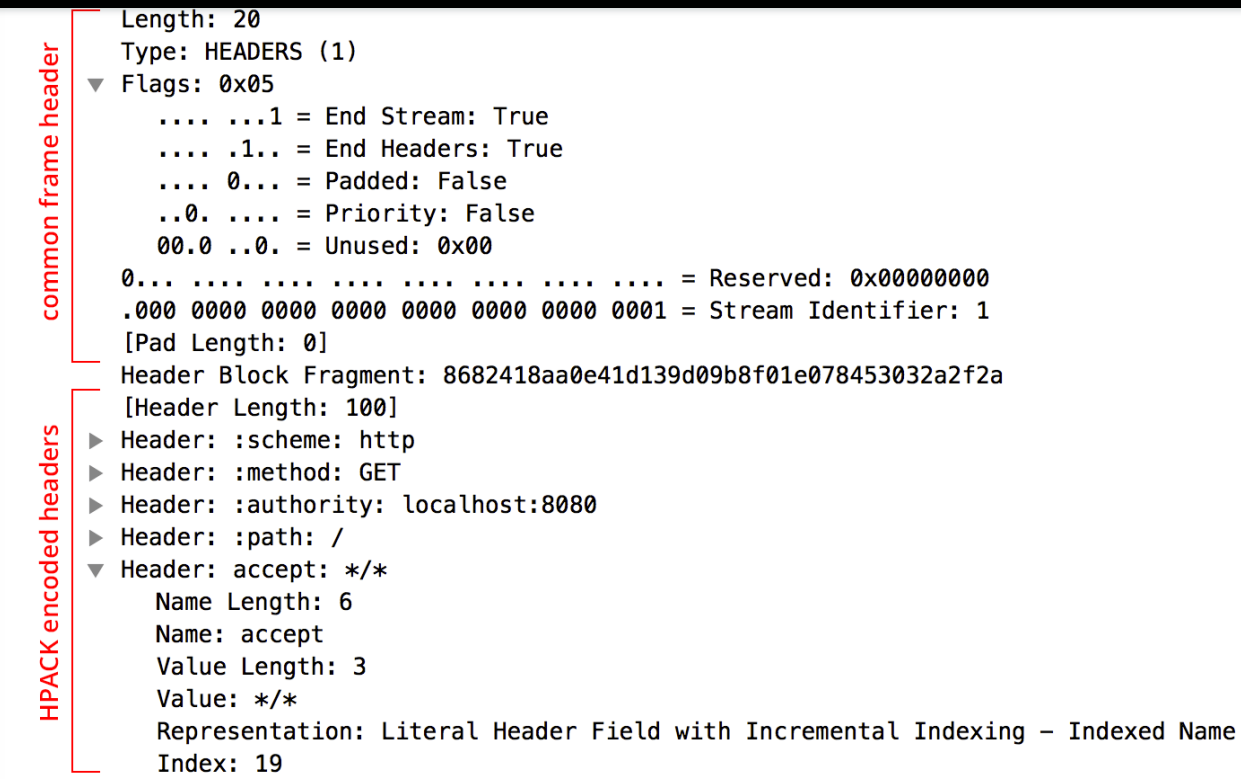

Figure 12-8. Decoded HEADERS frame in Wireshark

Note

Wireshark decodes and displays the frame fields in the same order as encoded on the wire—e.g., compare the fields in the common frame header to the frame layout in Figure 12-7.

The HEADERS frame is used to declare and communicate metadata about the new request. The application payload, if available, is delivered independently within the DATA frames. This separation allows the protocol to separate processing of "control traffic" from delivery of application data—e.g., flow control is applied only to DATA frames, and non-DATA frames are always processed with high priority.

Server-Initiated Streams via PUSH_PROMISE §

HTTP/2 allows both client and server to initiate new streams. In the case of a server-initiated stream, a PUSH_PROMISE frame is used to declare the promise and communicate the HPACK-encoded response headers. The format of the frame is similar to HEADERS, except that it omits the optional stream dependency and weight, since the server is in full control of how the promised data is delivered.

To eliminate stream ID collisions between client- and server-initiated streams, the counters are offset: client-initiated streams have odd-numbered stream IDs, and server-initiated streams have even-numbered stream IDs. As a result, because the stream ID in Figure 12-8 is set to "1", we can infer that it is a client-initiated stream.

Once a new stream is created, and the HTTP headers are sent, DATA frames (Figure 12-9) are used to send the application payload if one is present. The payload can be split between multiple DATA frames, with the last frame indicating the end of the message by toggling the END_STREAM flag in the header of the frame.

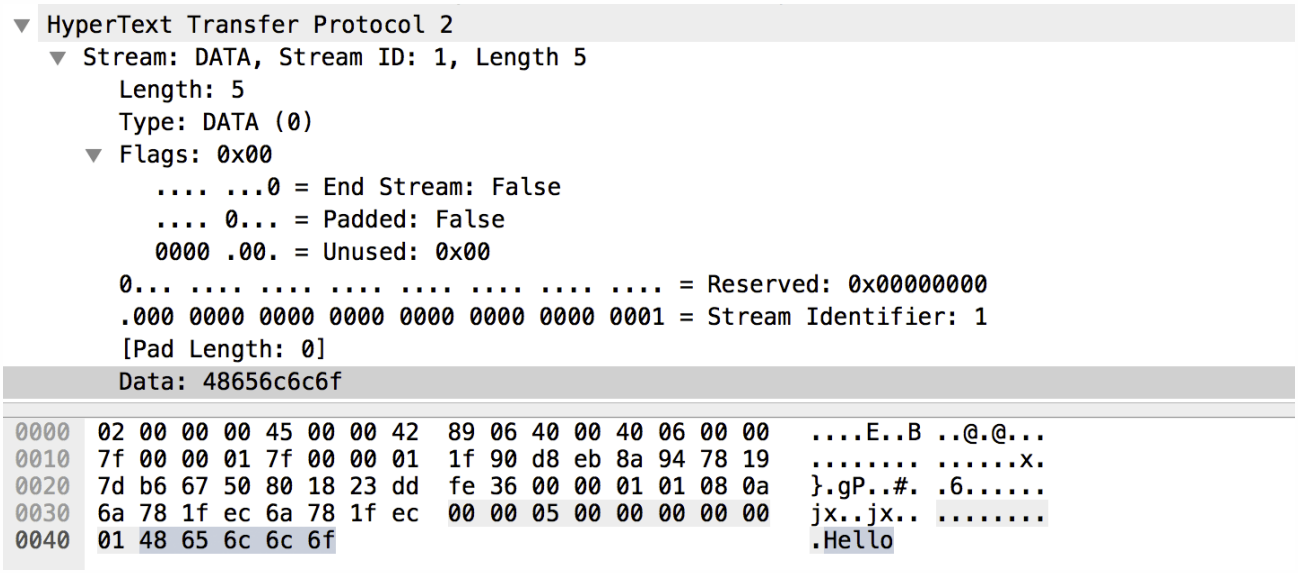

Figure 12-9. DATA frame

Note

The "End Stream" flag is set to "false" in Figure 12-9, indicating that the client has not finished transmitting the application payload; more DATA frames are coming.

Aside from the length and flags fields, there really isn't much more to say about the DATA frame. The application payload may be split between multiple DATA frames to enable efficient multiplexing, but otherwise it is delivered exactly as provided by the application—i.e., the choice of the encoding mechanism (plain text, gzip, or other encoding formats) is deferred to the application.

Analyzing HTTP/2 Frame Data Flow §

Armed with knowledge of the different frame types, we can now revisit the diagram (Figure 12-10) we encountered earlier in Request and Response Multiplexing and analyze the HTTP/2 exchange:

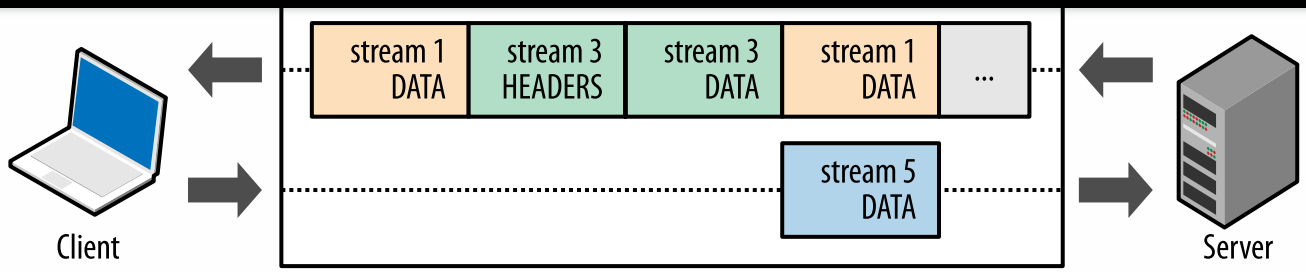

Figure 12-10. HTTP/2 request and response multiplexing within a shared connection

- There are three streams, with IDs set to 1, 3, and 5.
- All three stream IDs are odd; all three are client-initiated streams.
- There are no server-initiated ("push") streams in this exchange.
- The server is sending interleaved DATA frames for stream 1, which carry the application response to the client's earlier request.
- The server has interleaved the HEADERS and DATA frames for stream 3 between the DATA frames for stream 1—response multiplexing in action!
- The client is transferring a DATA frame for stream 5, which indicates that a HEADERS frame was transferred earlier.

The above analysis is, of course, based on a simplified representation of an actual HTTP/2 exchange, but it still illustrates many of the strengths and features of the new protocol. By this point, you should have the necessary knowledge to successfully record and analyze a real-world HTTP/2 trace—give it a try!

« Back to the Table of Contents