HTTP/1.1 vs HTTP/2: What's the Difference?

Introduction

The Hypertext Transfer Protocol, or HTTP, is an application protocol that has been the de facto standard for communication on the World Wide Web since its invention in 1989. From the release of HTTP/1.1 in 1997 until recently, there have been few revisions to the protocol. But in 2015, a reimagined version called HTTP/2 came into use, which offered several methods to decrease latency, especially when dealing with mobile platforms and server-intensive graphics and videos. HTTP/2 has since become increasingly popular, with some estimates suggesting that around a third of all websites in the world support it. In this changing landscape, web developers can benefit from understanding the technical differences between HTTP/1.1 and HTTP/2, allowing them to make informed and efficient decisions about evolving best practices.

After reading this article, you will understand the main differences between HTTP/1.1 and HTTP/2, concentrating on the technical changes HTTP/2 has adopted to achieve a more efficient Web protocol.

Background

To contextualize the specific changes that HTTP/2 made to HTTP/1.1, let's first take a high-level look at the historical development and basic workings of each.

HTTP/1.1

Developed by Timothy Berners-Lee in 1989 as a communication standard for the World Wide Web, HTTP is a top-level application protocol that exchanges information between a client computer and a local or remote web server. In this process, a client sends a text-based request to a server by calling a *method* like `GET` or `POST`. In response, the server sends a resource like an HTML page back to the client.

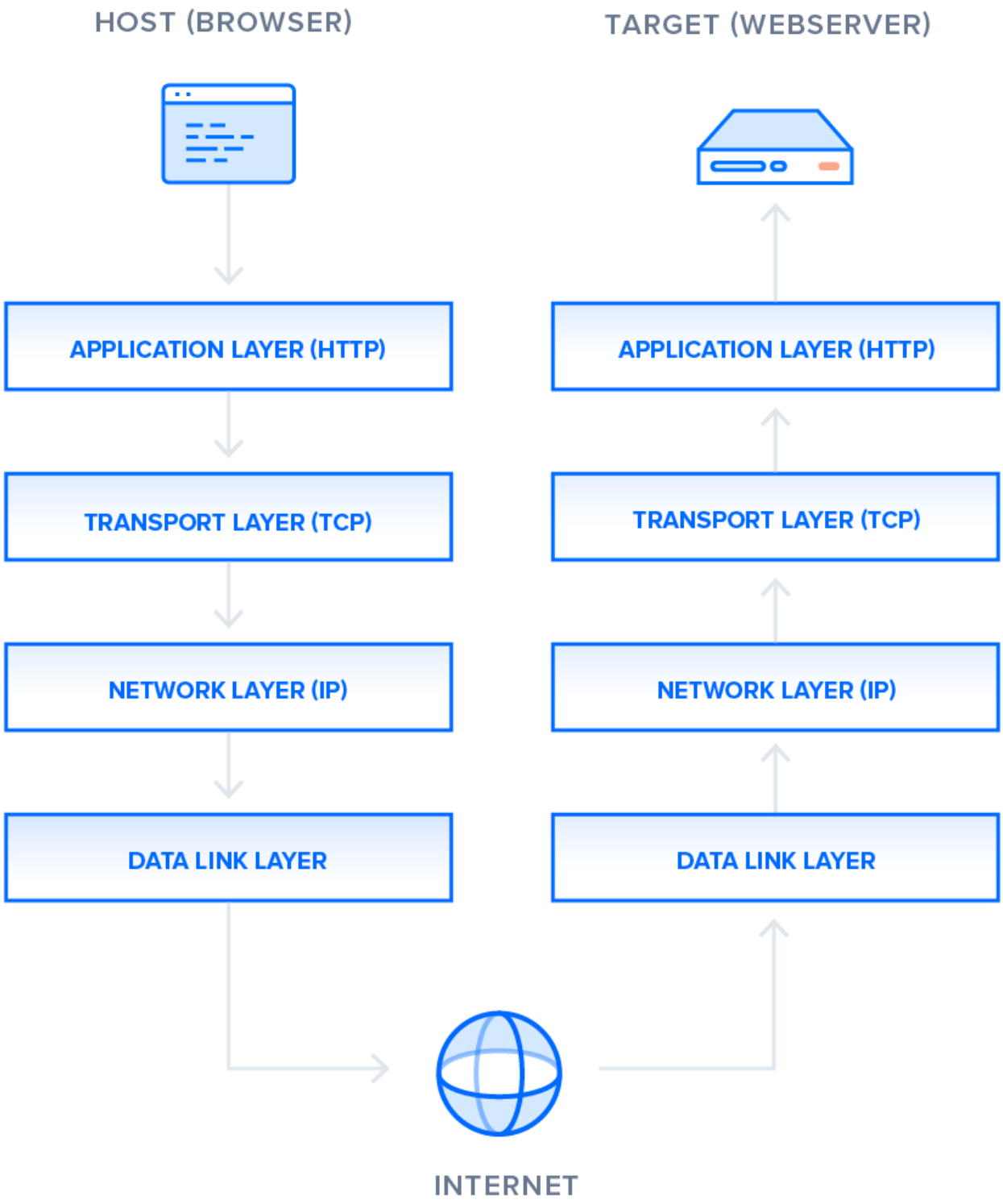
For example, let's say you are visiting a website at the domain `www.example.com`. When you navigate to this URL, the web browser on your computer sends an HTTP request in the form of a text-based message, similar to the one shown here:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

This request uses the `GET` method, which asks for data from the host server listed after `Host:`. In response to this request, the `example.com` web server returns an HTML page to the requesting

client, in addition to any images, stylesheets, or other resources called for in the HTML. Note that not all of the resources are returned to the client in the first call for data. The requests and responses will go back and forth between the server and client until the web browser has received all the resources necessary to render the contents of the HTML page on your screen.

You can think of this exchange of requests and responses as a single *application layer* of the internet protocol stack, sitting on top of the *transfer layer* (usually using the Transmission Control Protocol, or TCP) and *networking layers* (using the Internet Protocol, or IP):


HOST (BROWSER)
TARGET (WEBSERVER)
APPLICATION LAYER (HTTP)
TRANSPORT LAYER (TCP)
NETWORK LAYER (IP)
DATA LINK LAYER
APPLICATION LAYER (HTTP)
TRANSPORT LAYER (TCP)
NETWORK LAYER (IP)
DATA LINK LAYER
INTERNET

There is much to discuss about the lower levels of this stack, but in order to gain a high-level understanding of HTTP/2, you only need to know this abstracted layer model and where HTTP figures into it.

With this basic overview of HTTP/1.1 out of the way, we can now move on to recounting the early development of HTTP/2.

HTTP/2

HTTP/2 began as the SPDY protocol, developed primarily at Google with the intention of reducing web page load latency by using techniques such as compression, multiplexing, and prioritization. This protocol served as a template for HTTP/2 when the Hypertext Transfer Protocol working group httpbis of the IETF (Internet Engineering Task Force) put the standard together, culminating in the publication of HTTP/2 in May 2015. From the beginning, many browsers supported this standardization effort, including Chrome, Opera, Internet Explorer, and Safari. Due in part to this browser support, there has been a significant adoption rate of the protocol since 2015, with especially high rates among new sites.

From a technical point of view, one of the most significant features that distinguishes HTTP/1.1 and HTTP/2 is the binary framing layer, which can be thought of as a part of the application layer in the internet protocol stack. As opposed to HTTP/1.1, which keeps all requests and responses in plain text format, HTTP/2 uses the binary framing layer to encapsulate all messages in binary format, while still maintaining HTTP semantics, such as verbs, methods, and headers. An application level API would still create messages in the conventional HTTP formats, but the underlying layer would then convert these messages into binary. This ensures that web applications created before HTTP/2 can continue functioning as normal when interacting with the new protocol.

The conversion of messages into binary allows HTTP/2 to try new approaches to data delivery not available in HTTP/1.1, a contrast that is at the root of the practical differences between the two protocols. The next section will take a look at the delivery model of HTTP/1.1, followed by what new models are made possible by HTTP/2.

Delivery Models

As mentioned in the previous section, HTTP/1.1 and HTTP/2 share semantics, ensuring that the requests and responses traveling between the server and client in both protocols reach their destinations as traditionally formatted messages with headers and bodies, using familiar methods like `GET` and `POST`. But while HTTP/1.1 transfers these in plain-text messages, HTTP/2 encodes these into binary, allowing for significantly different delivery model possibilities. In this section, we will first briefly examine how HTTP/1.1 tries to optimize efficiency with its delivery model and the problems that come up from this, followed by the advantages of the binary framing layer of HTTP/2 and a description of how it prioritizes requests.

HTTP/1.1 — Pipelining and Head-of-Line Blocking

The first response that a client receives on an HTTP `GET` request is often not the fully rendered page. Instead, it contains links to additional resources needed by the requested page. The client discovers that the full rendering of the page requires these additional resources from the server only after it downloads the page. Because of this, the client will have to make additional requests to retrieve these resources. In HTTP/1.0, the client had to break and remake the TCP connection with every new request, a costly affair in terms of both time and resources.

HTTP/1.1 takes care of this problem by introducing persistent connections and pipelining. With persistent connections, HTTP/1.1 assumes that a TCP connection should be kept open unless directly told to close. This allows the client to send multiple requests along the same connection without waiting for a response to each, greatly improving the performance of HTTP/1.1 over HTTP/1.0.

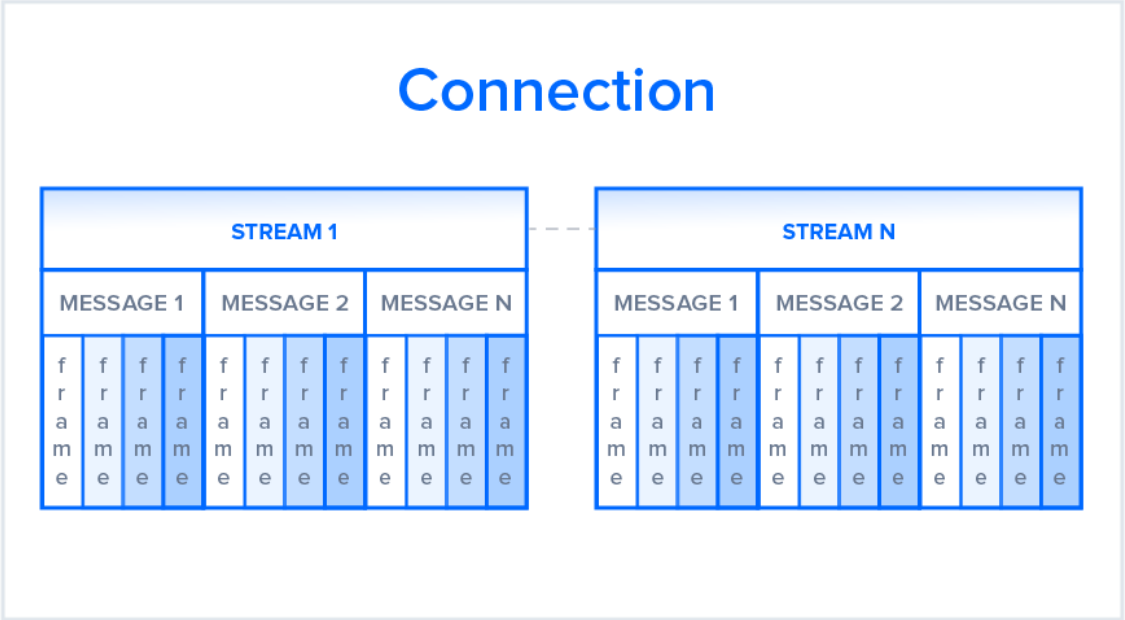
Unfortunately, there is a natural bottleneck to this optimization strategy. Since multiple data packets cannot pass each other when traveling to the same destination, there are situations in which a request at the head of the queue that cannot retrieve its required resource will block all the requests behind it. This is known as *head-of-line (HOL) blocking*, and is a significant problem with optimizing connection efficiency in HTTP/1.1. Adding separate, parallel TCP connections could alleviate this issue, but there are limits to the number of concurrent TCP connections possible between a client and server, and each new connection requires significant resources.

These problems were at the forefront of the minds of HTTP/2 developers, who proposed to use the aforementioned binary framing layer to fix these issues, a topic you will learn more about in the next section.

HTTP/2 — Advantages of the Binary Framing Layer

In HTTP/2, the binary framing layer encodes requests/responses and cuts them up into smaller packets of information, greatly increasing the flexibility of data transfer.

Let’s take a closer look at how this works. As opposed to HTTP/1.1, which must make use of multiple TCP connections to lessen the effect of HOL blocking, HTTP/2 establishes a single connection object between the two machines. Within this connection there are multiple *streams* of data. Each stream consists of multiple messages in the familiar request/response format. Finally, each of these messages split into smaller units called *frames*:



At the most granular level, the communication channel consists of a bunch of binary-encoded frames, each tagged to a particular stream. The identifying tags allow the connection to interleave these frames during transfer and reassemble them at the other end. The interleaved requests and responses can run in parallel without blocking the messages behind them, a process called *multiplexing*. Multiplexing resolves the head-of-line blocking issue in HTTP/1.1 by ensuring that no message has to wait for another to finish. This also means that servers and clients can send concurrent requests and responses, allowing for greater control and more efficient connection management.

Since multiplexing allows the client to construct multiple streams in parallel, these streams only need to make use of a single TCP connection. Having a single persistent connection per origin improves upon HTTP/1.1 by reducing the memory and processing footprint throughout the network. This results in better network and bandwidth utilization and thus decreases the overall operational cost.

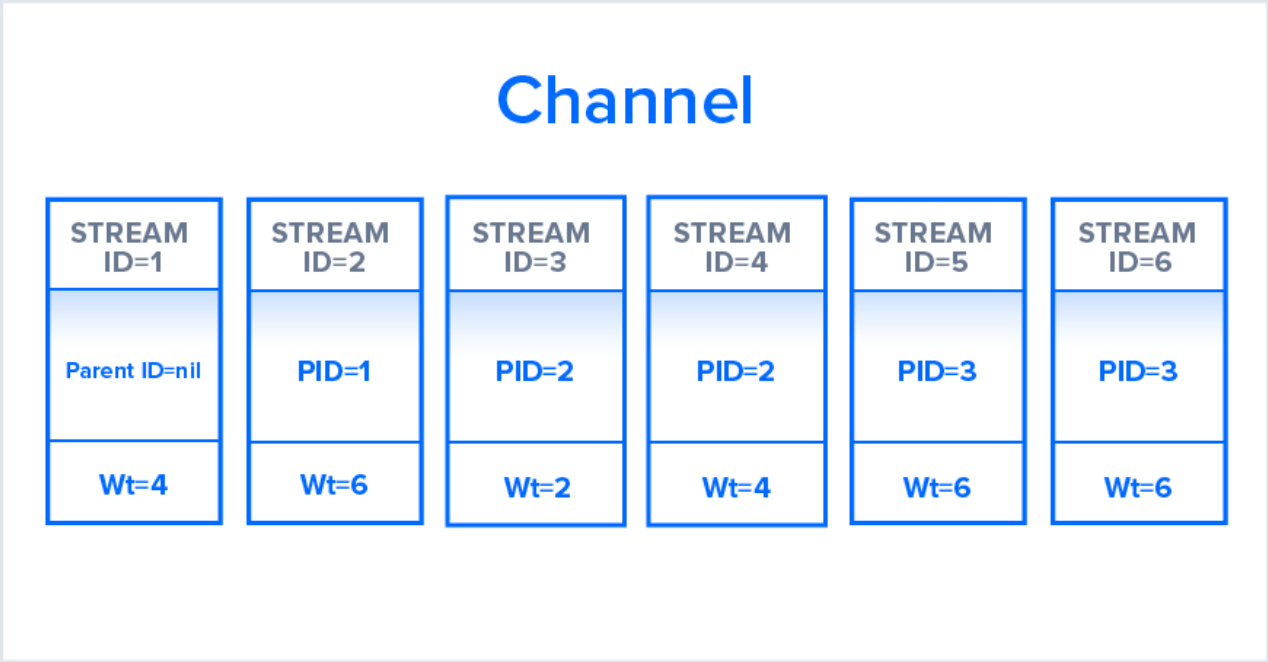
A single TCP connection also improves the performance of the HTTPS protocol, since the client and server can reuse the same secured session for multiple requests/responses. In HTTPS, during the TLS or SSL handshake, both parties agree on the use of a single key throughout the session. If the connection breaks, a new session starts, requiring a newly generated key for further communication. Thus, maintaining a single connection can greatly reduce the resources required for HTTPS performance. Note that, though HTTP/2 specifications do not make it mandatory to use the TLS layer, many major browsers only support HTTP/2 with HTTPS.

Although the multiplexing inherent in the binary framing layer solves certain issues of HTTP/1.1, multiple streams awaiting the same resource can still cause performance issues. The design of HTTP/2 takes this into account, however, by using stream prioritization, a topic we will discuss in the next section.

HTTP/2 — Stream Prioritization

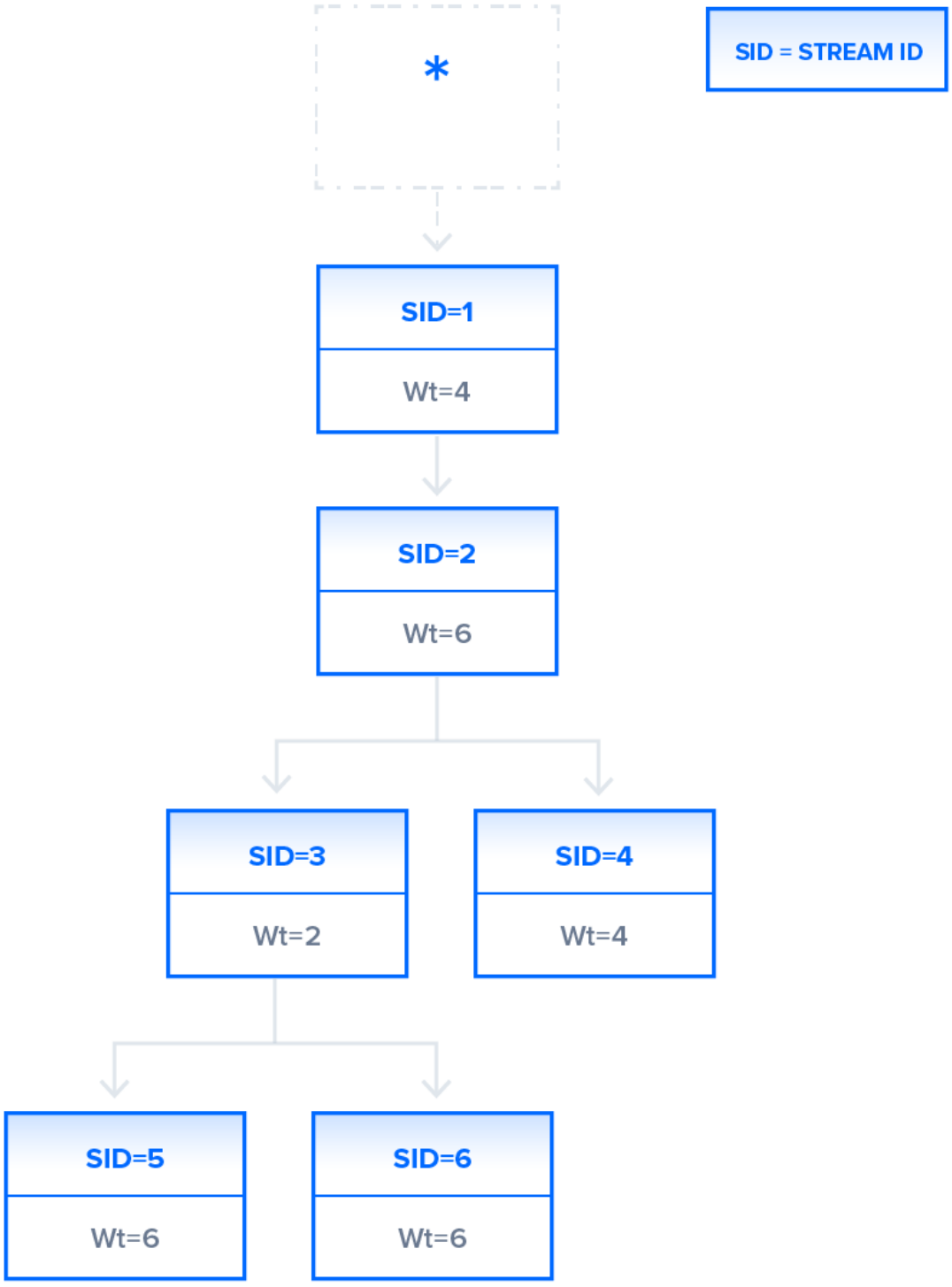
Stream prioritization not only solves the possible issue of requests competing for the same resource, but also allows developers to customize the relative weight of requests to better optimize application performance. In this section, we will break down the process of this prioritization in order to provide better insight into how you can leverage this feature of HTTP/2.

As you know now, the binary framing layer organizes messages into parallel streams of data. When a client sends concurrent requests to a server, it can prioritize the responses it is requesting by assigning a weight between 1 and 256 to each stream. The higher number indicates higher priority. In addition to this, the client also states each stream's dependency on another stream by specifying the ID of the stream on which it depends. If the parent identifier is omitted, the stream is considered to be dependent on the root stream. This is illustrated in the following figure:



In the illustration, the channel contains six streams, each with a unique ID and associated with a specific weight. Stream 1 does not have a parent ID associated with it and is by default associated with the root node. All other streams have some parent ID marked. The resource allocation for each stream will be based on the weight that they hold and the dependencies they require. Streams 5 and 6 for example, which in the figure have been assigned the same weight and same parent stream, will have the same prioritization for resource allocation.

The server uses this information to create a dependency tree, which allows the server to determine the order in which the requests will retrieve their data. Based on the streams in the preceding figure, the dependency tree will be as follows:



In this dependency tree, stream 1 is dependent on the root stream and there is no other stream derived from the root, so all the available resources will allocate to stream 1 ahead of the other

streams. Since the tree indicates that stream 2 depends on the completion of stream 1, stream 2 will not proceed until the stream 1 task is completed. Now, let us look at streams 3 and 4. Both these streams depend on stream 2. As in the case of stream 1, stream 2 will get all the available resources ahead of streams 3 and 4. After stream 2 completes its task, streams 3 and 4 will get the resources; these are split in the ratio of 2:4 as indicated by their weights, resulting in a higher chunk of the resources for stream 4. Finally, when stream 3 finishes, streams 5 and 6 will get the available resources in equal parts. This can happen before stream 4 has finished its task, even though stream 4 receives a higher chunk of resources; streams at a lower level are allowed to start as soon as the dependent streams on an upper level have finished.

As an application developer, you can set the weights in your requests based on your needs. For example, you may assign a lower priority for loading an image with high resolution after providing a thumbnail image on the web page. By providing this facility of weight assignment, HTTP/2 enables developers to gain better control over web page rendering. The protocol also allows the client to change dependencies and reallocate weights at runtime in response to user interaction. It is important to note, however, that a server may change assigned priorities on its own if a certain stream is blocked from accessing a specific resource.

Buffer Overflow

In any TCP connection between two machines, both the client and the server have a certain amount of buffer space available to hold incoming requests that have not yet been processed. These buffers offer flexibility to account for numerous or particularly large requests, in addition to uneven speeds of downstream and upstream connections.

There are situations, however, in which a buffer is not enough. For example, the server may be pushing a large amount of data at a pace that the client application is not able to cope with due to a limited buffer size or a lower bandwidth. Likewise, when a client uploads a huge image or a video to a server, the server buffer may overflow, causing some additional packets to be lost.

In order to avoid buffer overflow, a flow control mechanism must prevent the sender from overwhelming the receiver with data. This section will provide an overview of how HTTP/1.1 and HTTP/2 use different versions of this mechanism to deal with flow control according to their different delivery models.

HTTP/1.1

In HTTP/1.1, flow control relies on the underlying TCP connection. When this connection initiates, both client and server establish their buffer sizes using their system default settings. If the receiver's buffer is partially filled with data, it will tell the sender its *receive window*, i.e., the amount of available space that remains in its buffer. This receive window is advertised in a signal known as an *ACK packet*, which is the data packet that the receiver sends to acknowledge that it received the opening signal. If this advertised receive window size is zero, the sender will send no more data until the client clears its internal buffer and then requests to resume data transmission. It is important to note here that using receive windows based on the underlying TCP connection can only implement flow control on either end of the connection.

Because HTTP/1.1 relies on the transport layer to avoid buffer overflow, each new TCP connection requires a separate flow control mechanism. HTTP/2, however, multiplexes streams within a single TCP connection, and will have to implement flow control in a different manner.

HTTP/2

HTTP/2 multiplexes streams of data within a single TCP connection. As a result, receive windows on the level of the TCP connection are not sufficient to regulate the delivery of individual streams. HTTP/2 solves this problem by allowing the client and server to implement their own flow controls, rather than relying on the transport layer. The application layer communicates the available buffer space, allowing the client and server to set the receive window on the level of the multiplexed streams. This fine-scale flow control can be modified or maintained after the initial connection via a `WINDOW_UPDATE` frame.

Since this method controls data flow on the level of the application layer, the flow control mechanism does not have to wait for a signal to reach its ultimate destination before adjusting the receive window. Intermediary nodes can use the flow control settings information to determine their own resource allocations and modify accordingly. In this way, each intermediary server can implement its own custom resource strategy, allowing for greater connection efficiency.

This flexibility in flow control can be advantageous when creating appropriate resource strategies. For example, the client may fetch the first scan of an image, display it to the user, and allow the user to preview it while fetching more critical resources. Once the client fetches these critical resources, the browser will resume the retrieval of the remaining part of the image. Deferring the implementation of flow control to the client and server can thus improve the perceived performance of web applications.

In terms of flow control and the stream prioritization mentioned in an earlier section, HTTP/2 provides a more detailed level of control that opens up the possibility of greater optimization. The next section will explain another method unique to the protocol that can enhance a connection in a similar way: predicting resource requests with *server push*.

Predicting Resource Requests

In a typical web application, the client will send a `GET` request and receive a page in HTML, usually the index page of the site. While examining the index page contents, the client may discover that it needs to fetch additional resources, such as CSS and JavaScript files, in order to fully render the page. The client determines that it needs these additional resources only after receiving the response from its initial `GET` request, and thus must make additional requests to fetch these resources and complete putting the page together. These additional requests ultimately increase the connection load time.

There are solutions to this problem, however: since the server knows in advance that the client will require additional files, the server can save the client time by sending these resources to the

client before it asks for them. HTTP/1.1 and HTTP/2 have different strategies of accomplishing this, each of which will be described in the next section.

HTTP/1.1 — Resource Inlining

In HTTP/1.1, if the developer knows in advance which additional resources the client machine will need to render the page, they can use a technique called *resource inlining* to include the required resource directly within the HTML document that the server sends in response to the initial `GET` request. For example, if a client needs a specific CSS file to render a page, inlining that CSS file will provide the client with the needed resource before it asks for it, reducing the total number of requests that the client must send.

But there are a few problems with resource inlining. Including the resource in the HTML document is a viable solution for smaller, text-based resources, but larger files in non-text formats can greatly increase the size of the HTML document, which can ultimately decrease the connection speed and nullify the original advantage gained from using this technique. Also, since the inlined resources are no longer separate from the HTML document, there is no mechanism for the client to decline resources that it already has, or to place a resource in its cache. If multiple pages require the resource, each new HTML document will have the same resource inlined in its code, leading to larger HTML documents and longer load times than if the resource were simply cached in the beginning.

A major drawback of resource inlining, then, is that the client cannot separate the resource and the document. A finer level of control is needed to optimize the connection, a need that HTTP/2 seeks to meet with server push.

HTTP/2 — Server Push

Since HTTP/2 enables multiple concurrent responses to a client's initial `GET` request, a server can send a resource to a client along with the requested HTML page, providing the resource before the client asks for it. This process is called *server push*. In this way, an HTTP/2 connection can accomplish the same goal of resource inlining while maintaining the separation between the pushed resource and the document. This means that the client can decide to cache or decline the pushed resource separate from the main HTML document, fixing the major drawback of resource inlining.

In HTTP/2, this process begins when the server sends a `PUSH_PROMISE` frame to inform the client that it is going to push a resource. This frame includes only the header of the message, and allows the client to know ahead of time which resource the server will push. If it already has the resource cached, the client can decline the push by sending a `RST_STREAM` frame in response. The `PUSH_PROMISE` frame also saves the client from sending a duplicate request to the server, since it knows which resources the server is going to push.

It is important to note here that the emphasis of server push is client control. If a client needed to adjust the priority of server push, or even disable it, it could at any time send a `SETTINGS` frame to modify this HTTP/2 feature.

Although this feature has a lot of potential, server push is not always the answer to optimizing your web application. For example, some web browsers cannot always cancel pushed requests, even if the client already has the resource cached. If the client mistakenly allows the server to send a duplicate resource, the server push can use up the connection unnecessarily. In the end, server push should be used at the discretion of the developer. For more on how to strategically use server push and optimize web applications, check out the PRPL pattern developed by Google. To learn more about the possible issues with server push, see Jake Archibald's blog post HTTP/2 push is tougher than I thought.

Compression

A common method of optimizing web applications is to use compression algorithms to reduce the size of HTTP messages that travel between the client and the server. HTTP/1.1 and HTTP/2 both use this strategy, but there are implementation problems in the former that prohibit compressing the entire message. The following section will discuss why this is the case, and how HTTP/2 can provide a solution.

HTTP/1.1

Programs like gzip have long been used to compress the data sent in HTTP messages, especially to decrease the size of CSS and JavaScript files. The header component of a message, however, is always sent as plain text. Although each header is quite small, the burden of this uncompressed data weighs heavier and heavier on the connection as more requests are made, particularly penalizing complicated, API-heavy web applications that require many different resources and thus many different resource requests. Additionally, the use of cookies can sometimes make headers much larger, increasing the need for some kind of compression.

In order to solve this bottleneck, HTTP/2 uses HPACK compression to shrink the size of headers, a topic discussed further in the next section.

HTTP/2

One of the themes that has come up again and again in HTTP/2 is its ability to use the binary framing layer to exhibit greater control over finer detail. The same is true when it comes to header compression. HTTP/2 can split headers from their data, resulting in a header frame and a data frame. The HTTP/2-specific compression program HPACK can then compress this header frame. This algorithm can encode the header metadata using Huffman coding, thereby greatly decreasing its size. Additionally, HPACK can keep track of previously conveyed metadata fields and further compress them according to a dynamically altered index shared between the client and the server. For example, take the following two requests:

Request #1

```
method:     GET
scheme:     https
host:       example.com
path:       /academy
accept:     /image/jpeg
```

```
user-agent: Mozilla/5.0 ...
```

Request #2

```
method:     GET
scheme:     https
host:       example.com
path:       /academy/images
accept:     /image/jpeg
user-agent: Mozilla/5.0 ...
```

The various fields in these requests, such as `method`, `scheme`, `host`, `accept`, and `user-agent`, have the same values; only the `path` field uses a different value. As a result, when sending `Request #2`, the client can use HPACK to send only the indexed values needed to reconstruct these common fields and newly encode the `path` field. The resulting header frames will be as follows:

Header Frame for Request #1

```
method:     GET
scheme:     https
host:       example.com
path:       /academy
accept:     /image/jpeg
user-agent: Mozilla/5.0 ...
```

Header Frame for Request #2

```
path:       /academy/images
```

Using HPACK and other compression methods, HTTP/2 provides one more feature that can reduce client-server latency.

Conclusion

As you can see from this point-by-point analysis, HTTP/2 differs from HTTP/1.1 in many ways, with some features providing greater levels of control that can be used to better optimize web application performance and other features simply improving upon the previous protocol. Now that you have gained a high-level perspective on the variations between the two protocols, you can consider how such factors as multiplexing, stream prioritization, flow control, server push, and compression in HTTP/2 will affect the changing landscape of web development.

If you would like to see a performance comparison between HTTP/1.1 and HTTP/2, check out this Google demo that compares the protocols for different latencies. Note that when you run the test on your computer, page load times may vary depending on several factors such as bandwidth, client and server resources available at the time of testing, and so on. If you'd like to study the results of more exhaustive testing, take a look at the article HTTP/2 – A Real-World Performance Test and Analysis. Finally, if you would like to explore how to build a modern web application, you could follow our How To Build a Modern Web Application to Manage Customer Information with Django and React on Ubuntu 18.04 tutorial, or set up your own HTTP/2 server with our How To Set Up Nginx with HTTP/2 Support on Ubuntu 16.04 tutorial.