NGINX

HTTP/2 for Web Application Developers

Published
September 16th, 2015

Table of Contents

Introduction

As you may already know, HTTP/2 is the new version of HyperText Transport Protocol (HTTP), which was released as an IETF standard in early 2015. HTTP/2 support is now available in some web servers, including NGINX, and in recent versions of most web browsers.

HTTP/2 uses a single, multiplexed connection, replacing the multiple connections per domain that browsers opened up in HTTP/1.x. HTTP/2 compresses header data and sends it in a concise, binary format, rather than the plain text format used previously. By using a single connection, HTTP/2 reduces the need for several popular HTTP/1.x optimizations, making your web applications simpler.

HTTP/2 is closely tied to SSL. While SSL is not required by the HTTP/2 specification, all web browsers released so far will only run HTTP/2 if a website also uses SSL. HTTP/2 speeds up SSL-enabled websites and makes web applications simpler.

We encourage you to test HTTP/2 in your web applications, with a view to moving onto the new protocol. NGINX played a leading role in the adoption of SPDY, the precursor of HTTP/2, and NGINX software is often used for SSL termination, among other roles. So we expect NGINX to play a significant role in supporting implementation of HTTP/2.

This white paper describes the obstacles to web performance that are inherent in HTTP/1.x, the improvements found in HTTP/2, how to implement HTTP/2 with NGINX, and how to unwind HTTP/1.x optimizations to maximize HTTP/2 performance.

Pro Tip: If you want to enable HTTP/2 and start testing the new protocol immediately, you can jump to the section, Testing HTTP/2 on Your Site.

HTTP/1.x So Far

HTTP was introduced alongside HTML in the early 1990s, when the web first appeared. An HTTP connection could only handle one message at a time and could not be reused. Early web pages were simple; text, headers, and perhaps a few images, so this connection model was sufficient.

As web pages grew to include image files, CSS, and JavaScript, a single connection was no longer sufficient. Many web pages today have more than 100 separate elements. To reduce page load times, browsers created multiple connections at once - usually six to eight - to allow for some parallelism in file transfer.

HTTP/1.1 was introduced in 1999 to address some issues with the original HTTP. This update to the standard formalized the practice of using keepalive connections - connections that can be re-used. However, keepalive connections are still subject to head-of-line blocking; a request for a large file holds up many subsequent requests for smaller files.

Connection usage in HTTP/1.x also works against widespread adoption of SSL. SSL needs extra cycles to establish a new connection, because each connection has to be authenticated. The use of multiple connections means more wait time for initial handshaking to complete.

Developers use several optimizations to get the most out of the available connections with HTTP/1.x:

- **Domain sharding** - Files are parceled out to multiple subdomains, with the browser opening six to eight connections for each one. This increases parallelism in file transfer but adds more connection overhead.
- **Image sprites** - Images are combined in one large file for efficient downloading, and each individual image is retrieved by lookup. Users must wait until the combined file arrives before they can see any images from it. The large files undermine caching and tie up RAM.
- **Concatenating code files** - All code of a given type (JavaScript, CSS, etc.) goes into a single file. This saves on connections but undermines caching, makes the user wait for all the code before any of it can run, and consumes extra RAM.
- **Inlining files** - CSS code, JavaScript code, and even images are inserted directly into the HTML file. This reduces connection usage, but takes up extra RAM, and initial page rendering is delayed until the enlarged HTML file finishes downloading.

These optimizations also add extra steps to development and deployment workflows. And finally, extra connections opened by browsers - multiplied by the use of domain sharding - put strain on networks. Thankfully, these connection-related tricks are rendered unnecessary by HTTP/2 and can be eliminated under the new standard.

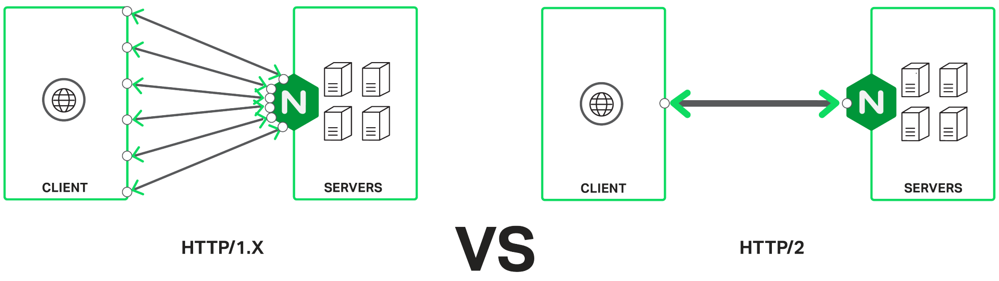
Presenting HTTP/2

HTTP/2 allows for faster, more efficient, more secure data transfer in most applications. HTTP/2 is based on SPDY, a fast, non-standard web transport protocol that was introduced by Google in 2009.

HTTP/2 retains the same semantics as HTTP/1.1. This includes HTTP methods such as GET and POST, status codes such as 404 (page not found), URLs, and how header fields are defined and used.

While retaining these characteristics, HTTP/2 adds five key features:

- Single, Persistent Connection - Only one connection is used for each web page, as shown in the figure. The same connection is used as long as the web page is open.
- Multiplexing - Requests and replies are prioritized and multiplexed onto separate streams within the single connection. When the connection is stable, “head of line blocking” - making every transfer wait for all previous transfers to complete - is eliminated.
- Header Compression and Binary Encoding - Headers are compressed using a new, separate, secure standard, HPACK compression, which reduces the amount of data crossing the network. Header information is sent in compact, binary format, not as plain text.
- Prioritization - Requests are assigned levels of dependency and requests at the same level are prioritized. The server uses this information to order and assign resources to fulfilling requests.
- SSL Encryption - HTTP/2 allows you to add SSL support with, in some cases, no performance penalty, making your site more secure.



How does HTTP/2 overcome the performance overhead imposed by SSL on HTTP/1.x? There are four key techniques:

1. Having a single connection minimizes SSL handshaking.
2. Headers are compressed, reducing the time needed to send them.
3. Multiplexing means file transfers don’t wait on other requests.
4. Files don’t need to be in-lined, concatenated, or sprited, so caching can work optimally.

When compared to a non-SSL implementation, there is still SSL performance overhead for authenticating the single connection and for encrypting and decrypting data, but this remaining overhead should be more or less offset by the performance improvements in HTTP/2.

You can begin using HTTP/2 for your websites and web applications without understanding its internals, but this information can help you know what to expect from HTTP/2 in terms of performance with your web content.

Note. The HTTP/2 specification also includes server push, in which the server sends files listed in a page's HTML before they're requested. Server push is not supported in NGINX's current implementation of HTTP/2.

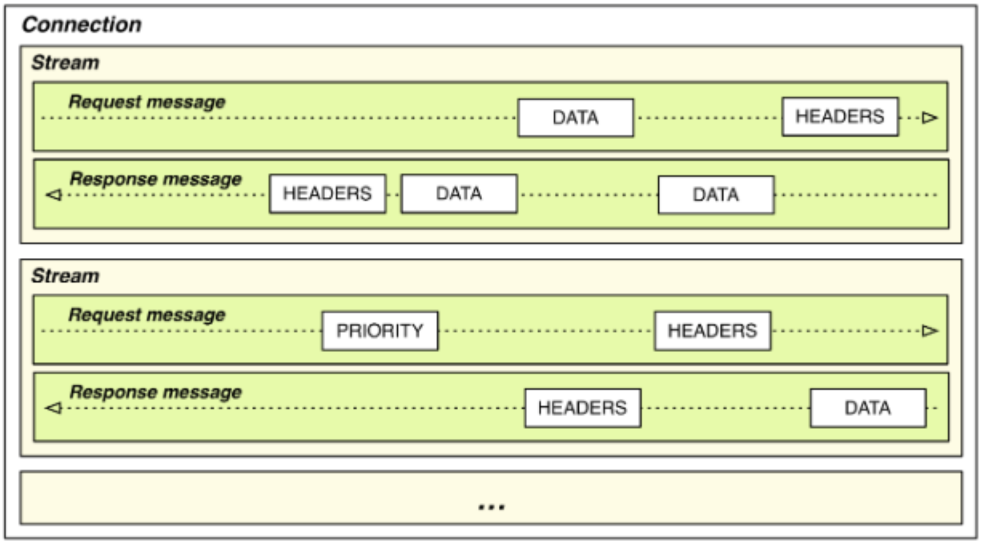
Single, Persistent Connection

Like driving a car down a narrow alleyway, an HTTP/1.1 can only carry one file at a time. An HTTP/2 connection is multiplexed, allowing different requests and responses to share the same connection. This difference is the basis for many of HTTP/2's benefits.

An HTTP/2 connection has three elements:

- **Messages** - A request or response.
- **Streams** - A call and response channel that carries messages in two directions simultaneously. A connection can have as many streams as needed.
- **Frames** - Messages are broken down into frames for sending. Each frame has a header that identifies its message and stream, so frames from different messages can be intermixed in the request or response direction of a stream.

The figure shows the traffic flow over an HTTP/2 connection. Two streams, each with a request and a response channel, carry multiplexed requests and responses. Each request and response is independent of the others.



Multiplexing

The single connection used by HTTP/2 supports multiplexing, allowing it to be shared flexibly by pending requests and responses.

The table shows the typical web browser’s page-loading process with HTTP/1.1 vs. HTTP/2. Bold text in the HTTP/1.1 column highlights how HTTP/1.1 requires more time, more steps, and more data to be transferred.

HTTP/1.1 Page Load	**HTTP/2 Page Load**
1. Create six to eight connections.	1. Create a single connection.
2. Request HTML page.	2. Request HTML page.
3. Receive HTML page.	3. Receive HTML page.
4. Decode HTML page.	4. Decode HTML page.
5. Request first six to eight files included in the HTML page, **no priorities or dependencies**. **(Requests have uncompressed, plain-text headers.)**	5. Request all files included in the HTML page, with priorities and dependencies. (Requests have compressed, binary headers.)
6. **On each connection, wait for requested file to arrive.**	(Files returned, multiplexed on single connection, as ready.)
7. **Request next file on now-open connection.**	--
8. **Repeat 6-7 for each remaining file.**	--
9. **Close six connections.**	8. Close single connection.

The steps that are faster and more efficient for HTTP/2 are:

1. **Creating and closing connections** - HTTP/2 creates a single connection; HTTP/1.1 creates six to eight, or many more if domain sharding is in use.
2. **Waiting to make requests** - HTTP/2 makes all requests together, with multiplexing and prioritization to deliver high-priority resources as quickly as possible. HTTP/1.1 makes six to eight requests, then waits to receive responses before making additional requests, one at a time on each connection as it becomes available.
3. **Sending less header data** - Unlike HTTP/1.1, header data for HTTP/2 is compressed, putting less data on the wire.

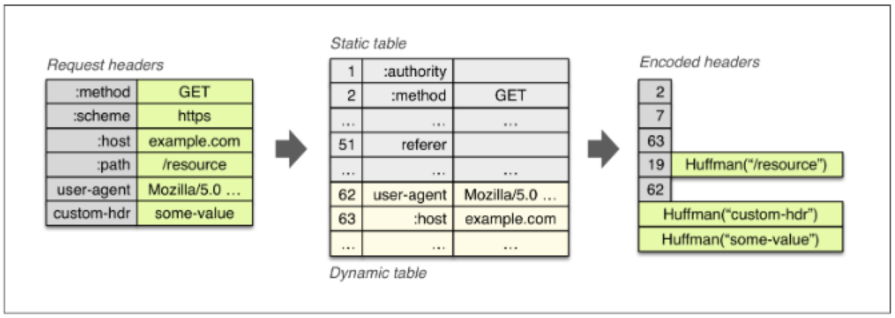
Header Compression and Binary Encoding

In HTTP/1.x, header data is sent as plain text. In HTTP/2, header data for the single, multiplexed connection is compressed according to the new HPACK standard, which uses Huffman coding. Header data is also sent in binary format, further reducing file size.

Header compression is not always a big positive. Compressing header data adds CPU overhead. In cases where the uncompressed request header fits in a single TCP packet, compression doesn’t help. For responses, header data is often much smaller than the accompanying data, so the benefit of header compression is small as well. Both compression and binary encoding make for obscurity in debugging, compared to plain text. Telnet can no longer be used for debugging, but Wireshark supports HTTP/2.

In HPACK compression, a static table contains known header fields and common values, each assigned an index number. As header transmission starts, values in the static table replace text strings in the header. As header transmission continues, a dynamic table is built up as well, assigning codes to new header-value pairs. The dynamic table is used as fields repeat within the session. Testing shows compression of greater than 50% on headers sent by the client and nearly 90% on headers sent by the server, probably due to greater use of the dynamic table as the connection is used.

The figure shows HPACK in use. As the transfer starts, the static table (middle column, top) already has header-value pairs such as `:method GET` (index 2). As the actual request headers are processed, a header-in pair that is not in the static table, `user-agent Mozilla/5.0`, is added to the dynamic table (middle column, bottom, index 62). The encoded headers then include the index values `2` and `62`, rather than the much longer header-value pairs that they represent.



Request headers

:method	GET
:scheme	https
:host	example.com
:path	/resource
user-agent	Mozilla/5.0 ...
custom-hdr	some-value

Static table

1	:authority	
2	:method	GET
...	...	...
51	referer	
...	...	...
62	user-agent	Mozilla/5.0 ...
63	:host	example.com
...	...	...

Dynamic table

Encoded headers

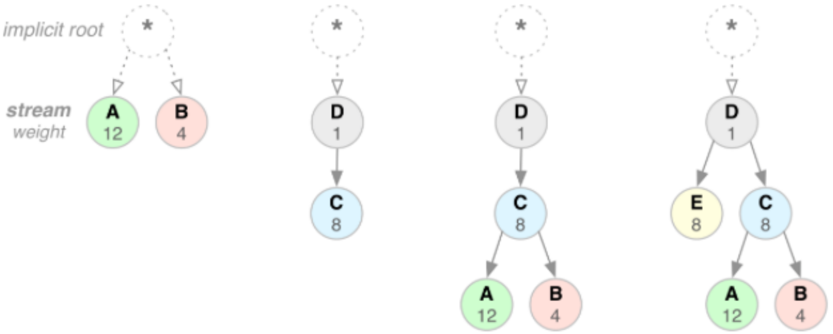
2	
7	
63	
19	Huffman("/resource")
62	
Huffman("custom-hdr")	
Huffman("some-value")	

Prioritization

Having a single, multiplexed connection means that the order in which responses are received makes a big difference in performance. So HTTP/2 streams have a prioritization weight, which is an integer between 1 (lowest priority) and 256 (highest), and any stream may be made dependent on another. The prioritization scheme is used to determine the order in which files are requested and in which the server retrieves data.

In an optimal implementation, for instance, CSS and JavaScript files are given a higher priority than image files. This allows code files to be download and run quickly, preventing delays.

The combination of dependencies and weights can be expressed as a "prioritization tree", as shown in the figure. The server can allocate CPU, memory, and bandwidth to support the prioritization expressed by the client.



In the figure, the last example is the most complex. First, stream D gets all available resources; stream E and stream C then share the available resources equally; and stream A and B then share the available resources, with stream A getting three-fourths of the available resources. Dependencies and weights can be changed by the client on the fly, in response to user interaction and other new information.

Multiplexing and prioritization operate to greatest effect when there are multiple requests operating in parallel. When evaluating performance, you are likely to find that advantages of HTTP/2 are less, and the performance slowdown due to SSL greater, for very large files and streaming media files such as audio and video. There may be cases where HTTP/1.x without SSL is required for acceptable performance.

Pro Tip: Consider testing performance and user success on various site tasks - articles viewed, file downloads, purchases made - before and after HTTP/2 implementation. See if site response time improves, as it should with HTTP/2, and if so, whether users complete more tasks on the faster site.

SSL Encryption

SSL is not a requirement of HTTP/2 per se, but currently shipping browsers only support HTTP/2 if server support for HTTP/2 is enabled and SSL is in use. Future browser support for HTTP/2 is also likely to require SSL support.

According to the Trustworthy Internet Movement, nearly 25% of the most popular websites implement SSL site-wide, whereas only 10% did so in 2012. SSL sites include Google Search, Gmail, and Google Docs, as well as Facebook and Yahoo!.

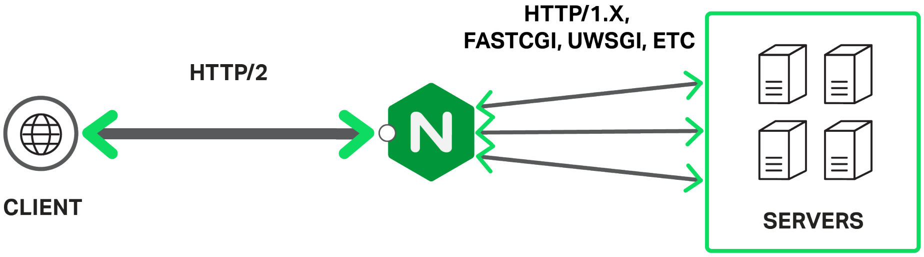
New connections are expensive with SSL; each requires a computationally intensive exchange of keys – typically, an RSA 2k public/private key pair. But, under HTTP/2, only one initial connection is needed, instead of six to eight under HTTP/1.x (more with domain sharding).

If you already use SSL then moving to HTTP/2, with its single, persistent connection, will cut SSL overhead. If, not using SSL already, you add it in a move to HTTP/2, you will likely see the slowdown from SSL more or less offset by the performance benefit from HTTP/2.

For more details on secure connections, and quotes from Facebook, Google, Twitter and others as to the manageability of the associated overhead, see the website, Is TLS Fast Yet?.

Testing HTTP/2 on Your Site

You can easily implement HTTP/2 in a web application that already uses NGINX or NGINX Plus. In most cases, NGINX will be in front of other web servers, providing capabilities such as load balancing, content caching, and SSL termination. Turning on HTTP/2 on the NGINX server enables HTTP/2 for client communications, but doesn't affect the other servers, since NGINX uses HTTP/1.x or other appropriate protocol to communicate with them, as shown in the figure.



Many of the infrastructure changes required to add HTTP/2 support to your website will be automatic from the point of view of both users and website operators. Once you've applied the beta patch to NGINX, there are just a few things you need to do to use HTTP/2 at your site:

1. SSL-enable your website.
2. Install up-to-date SSL software.
3. Turn on HTTP/2 support in NGINX.
4. Unwind HTTP/1.1 optimizations that reduce performance for HTTP/2.

As soon as the first three steps are completed, browsers that support HTTP/2 will use it when requesting web content. Then, as convenient, unwind any performance optimizations that you've implemented for HTTP/1.1, as described below. Making these changes will simplify your web applications and improve the performance of your website under HTTP/2.

Note. For fast performance today in production environments, consider using SPDY/3.1 on NGINX or NGINX Plus. SPDY is currently supported by more web browser versions than HTTP/2 and works with earlier versions of SSL.

Caveats

A few notes about HTTP/2 with NGINX and NGINX Plus:

- **Server push** - The HTTP/2 server push feature is not supported in this patch and will not be supported in the first production-ready release with HTTP/2 support. Server push may appear in future releases.

- **HTTP/1 traffic** - Even after you enable HTTP/2, traffic from older browser versions that do not support HTTP/2 will continue to communicate with your web application using HTTP/1 (currently about 50% each). Expect to test against HTTP/1.x and support it, even as your HTTP/2 share grows, for several years to come.
- **SPDY removed** – NGINX versions with HTTP/2 enabled by default, as described in Step 3 below, no longer support SPDY. SPDY is being deprecated by Google as of early 2016, so there is no need to support both.

1. SSL-Enable Your Website

The HTTP/2 specification does not require that SSL be in force, but current browsers do. So SSL-enable your website to get it ready for HTTP/2.

Follow these steps:

1. Get an SSL certificate. New providers, some free, have recently come online.
2. Update embedded links from `http` to `https`, or have NGINX redirect all your users to the SSL-encrypted site with the code snippet below.

```
server {
   listen 80;
   location / {
       return 301 https://$host$request_uri;
   }
}
```

With SSL enabled, when a user accesses your site using an HTTP/2-aware web browser, the browser will attempt to connect to your web server using HTTP/2.

Note: You will need to assess performance before and after moving to SSL and before and after moving to HTTP/2. This includes response time and peak capacity. Be sure that your host server can still handle the same load as previously, both in terms of response time and CPU usage.

2. Install Up-to-Date SSL Software and Link to It

The NGINX implementation of HTTP/2 uses the Application Layer Protocol Negotiation (ALPN) extension to TLS. Prior to enabling HTTP/2, you must install OpenSSL 1.0.2, which includes ALPN support, or a later version, on your NGINX servers. Then link to OpenSSL, as described in Step 5 of this NGINX blog post.

Note. Currently, the Firefox (Firefox version 39) and Chrome (Chromium version 44) browsers also support HTTP/2 negotiation using Next Protocol Negotiation (NPN), which is available in OpenSSL 1.0.1 and later. This support is deprecated and is due to be removed from the browsers in spring 2016. However, for now, you can run on OpenSSL 1.0.1 using NPN if needed.

3. Turn On HTTP/2 in Your Server

In order to finish the move to HTTP/2, your web server needs to be ready to receive HTTP/2 requests. You will also want to add an HTTP/1.x-to-HTTP/2 redirector.

NGINX Plus users simply need to use the HTTP/2-enabled package `nginx-plus-http2`. For NGINX open source, use Version 1.9.6 or later.

For any HTTP/2-enabled version of NGINX, do the following:

1. Add the `http2` parameter and, if not already in place, the `ssl` parameter, to existing listen directives - but first, add a URL redirector for HTTP/1.x:

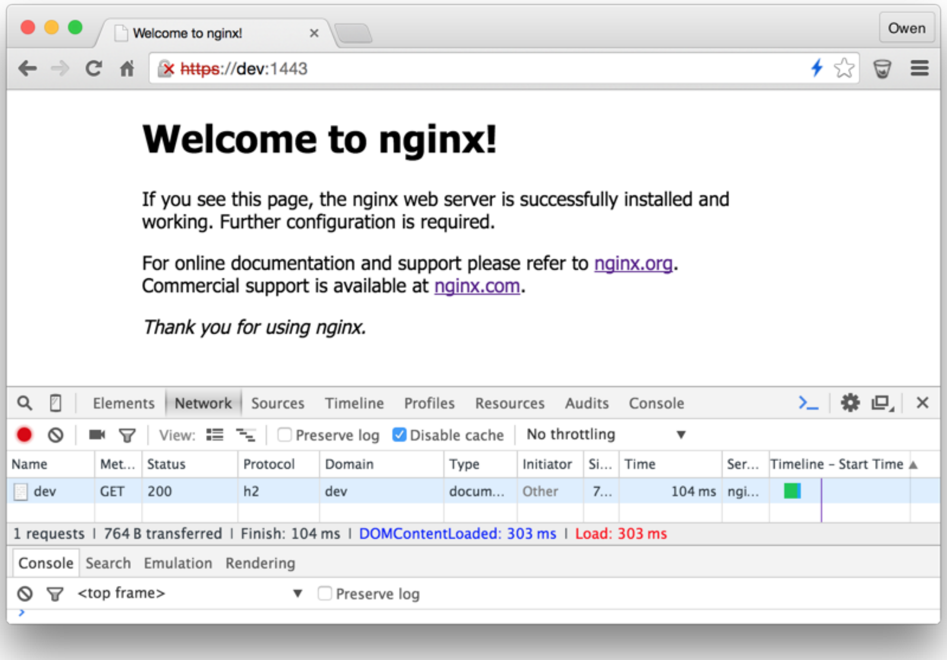
```
server {
    listen              80;
    server_name         nginx.com;
    return              301 https://www.nginx.com$request_uri;
}

server {
    listen              443 ssl http2 default_server;
    ssl_certificate     server.crt;
    ssl_certificate_key server.key;
…
}
```

2. Restart NGINX:

```
 # nginx -s reload
```

3. Verify that HTTP/2 is indeed being used. One good way to do this is to use the HTTP/2 and SPDY indicator plugin for Google Chrome. The figure shows the plugin in use; a blue lightning bolt in the browser address bar shows that HTTP/2 is active. The web page shows the protocol as "h2", short for HTTP/2.



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

□ Preserve log ☑ Disable cache

Name	Met...	Status	Protocol	Domain	Type	Initiator	Si...	Time	Ser...	Timeline – Start Time
dev	GET	200	h2	dev	docum...	Other	7...	104 ms	ngi...	🟩

1 requests | 764 B transferred | Finish: 104 ms | DOMContentLoaded: 303 ms | Load: 303 ms

□ Preserve log

If you have any proxy devices, such as load balancers, Content Delivery Networks (CDNs), and Web Application Firewalls (WAFs), they should be moved behind the NGINX server, so that the client communicates with NGINX first.

Note: If you want to stay with HTTP/1 for server-to-server communications, you need to deploy an HTTP/2-to-HTTP/1 proxy. You can put an NGINX server in front of your other servers as an HTTP/2 gateway and for SSL termination, as NGINX supports HTTP/2 connections from web clients and automatically uses HTTP/1 for communication with the servers it is proxying.

4. Review HTTP/1.1 Optimizations

Under HTTP/2, HTTP/1.x optimizations are likely to hurt performance, so you should consider removing them. Here's a brief listing of the changes to consider and the potential benefits:

- **Domain sharding**. Host files within a single domain to satisfy all requests. This cuts connection overhead and simplifies website management. (If some or all of the URLs that you currently use can be resolved to a single domain, and you have a wildcard SSL certificate that covers all of the subdomains in use, HTTP/2 will treat them as one, effectively overriding sharding.)
- **Image spriting, concatenated files, inlined files**. Consider removing or reducing spriting of images, concatenation of code files, and inlining of data into HTML files. Breaking up combined files can cut the initial HTML file download time; reduce long initial waits for larger files to download; cuts the memory footprint of your web page, as you don't need to keep big image and code files in memory; and allows caching to work properly at all levels, from the browser on up. Keeping files separate also simplifies workflow for site creation and maintenance.

You may need to experiment to find out what practices yield optimal performance under HTTP/2. Accessing many small files may slow performance even under HTTP/2. Also, larger files might compress better than smaller files, due to a larger compression context.

Consider testing to identify instances where limited use of image spriting, file concatenation, and file inlining might still benefit performance, even under HTTP/2. (For instance, you may find it worthwhile to sprite a group of images which make up a page navigation block.) A limited implementation of sharding may also offer modest performance advantages, even in the HTTP/2 context. If you use a CDN, you will need to evaluate whether it still delivers improved performance under HTTP/2.

When you do implement HTTP/2, you have the opportunity for a natural experiment - a chance to do comparison testing in a real-world environment. Many of your users will be using browser versions that don't support HTTP/2; other users will be using browser versions that do. You can compare and contrast performance on the two protocols to help you optimize your code for HTTP/2 performance without affecting performance for the remaining HTTP/1/x users too much.

Pro Tip. Be ready for increased work in testing, as you will need to test new releases on both HTTP/1.x, for browser versions that only support that, and HTTP/2 for up-to-date browser versions.

Conclusion

HTTP/2 is an exciting new option for web applications. It provides strong support for more secure, simpler, faster sites.

In addition to taking the specific steps above, you may need to consider other implementation approaches. To implement HTTP/2 in a mixed server environment, you can use NGINX as an HTTP/2 gateway, with SSL termination included.

Simply put an NGINX server in front of other servers and enable HTTP/2, as described above. The NGINX server will speak HTTP/1.x or HTTP/2 to browsers, as required, and HTTP/1.x and other protocols to proxied servers.

You may also need to review other implementation strategies. For instance, under HTTP/2, CDNs may lose some of their advantage; the benefits of using faster servers, or servers one or two steps closer to the user, may be offset by the extra overhead of opening up additional connections to access the CDN servers. (According to the HTTP Archive, about 40% of the top 1000 sites use a CDN, so this has big implications for site architecture.)

With HTTP/2, the domain sharding that's easily implemented through CDNs is no longer an advantage, while the extra handshaking needed to access separate servers will hurt performance, with less offsetting advantage due to multiple connections for file transfer.

You and others may find new and different performance techniques that help your web application under HTTP/2. Expect to find lively online discussions about the best ways to use the new protocol.

We hope this white paper is a useful early step in your journey toward gaining the simplicity, site performance improvements, and security offered by HTTP/2 for your web applications.

Appendix: The Story of SPDY

SPDY is the precursor to HTTP/2. SPDY was initially developed by Google and introduced in 2009 as an open, experimental, non-standard protocol for speeding up the web. In many ways it is the foundation for HTTP/2, and is in use across roughly 5% of websites as of August 2015.

SPDY was designed to speed up websites while requiring minimal changes in the infrastructure of the Internet. The key techniques used are the same that we now see in HTTP/2:

- Use a single Internet connection, with requests and responses multiplexed on it.
- Use binary format, rather than plain text, for headers.
- Compress headers.
- Prioritize requests and responses within the single connection.
- Require SSL on all sites that support the new protocol.

SPDY achieved significant performance improvements for most websites, with some pages loading more than 50% faster.

For SPDY to be useful, both web servers and web clients need to support it. NGINX is by far the most popular web server that supports SPDY; as of this writing, about 77% of sites that use SPDY use NGINX.

Currently, most browser versions in active use support SPDY. Google, Facebook, and Twitter use SPDY, and WordPress sites have been the largest generators of SPDY traffic. (WordPress runs on NGINX.)

HTTP/2 is based on SPDY, with a few changes. The major changes are:

- Compression is by HPACK instead of ZLIB.
- HTTP/2 makes the prioritization scheme more complex and more powerful.

During the years in which HTTP/2 was being developed, SPDY was used to test alternatives for use within HTTP/2. When HTTP/2 was standardized in mid-2015, SPDY support was deprecated, and SPDY usage is now expected to decrease steadily.

Additional Resources

Following are a few key resources for understanding HTTP/2:

- SPDY whitepaper - The Google Chrome team introduced SPDY with a whitepaper.
- SPDY blog post - Google's Ilya Grigorik explains some SPDY details.
- HTTP/2 Wikipedia entry - Includes differences with HTTP/1.x.
- HTTP/2 talk and slide deck - Ilya explains the benefits of HTTP/2.
- HTTP/2 RFC and FAQ - The official specification for HTTP/2, published as a Request for Comment (RFC) on Github, plus Frequently Asked Questions (FAQ).
- HPACK definition - The RFC for HPACK compression.
- Overclocking SSL - Write-up of a talk on SSL/TLS handshake performance costs, including handshaking.
- HTTP/2 chapter - The chapter on HTTP/2 from Ilya's book, High Performance Browser Networking.
- HTTP/2 Tradeoffs - Google's William Chan carefully analyzes HTTP/2 pluses and minuses.
- NGINX Plans for Supporting HTTP/2 - Owen Garrett of NGINX describes the company's support for SPDY and plans to support HTTP/2.