*What Every Web Developer Should Know About Networking and Browser Performance*

Compliments of NGINX

**SPECIAL EDITION: SELECTED CONTENT**

*High Performance*

# Browser Networking

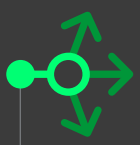
O'REILLY®

*Ilya Grigorik*

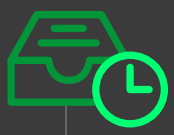
# Building a great app is just the beginning

**NGINX Plus is a complete application delivery platform for fast, flawless delivery**

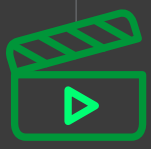
**Load Balancer**
Optimize the availability of apps, APIs, and services



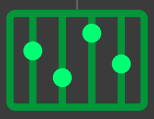
**Content Caching**
Accelerate local origin servers and create edge servers



**Web Server**
Deliver assets with speed and efficiency



**Streaming Media**
Stream high-quality video on demand to any device



**Monitoring & Management**
Ensure health, availability, and performance of apps with devops-friendly tools

See why the world's most innovative developers choose NGINX to deliver their apps – from Airbnb to Netflix to Uber.

Download your free trial **NGINX.com**

NGINX

# High Performance Browser Networking

*Ilya Grigorik*

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

**High Performance Browser Networking**

by Ilya Grigorik



Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Editor:** Courtney Nash
**Production Editor:** Melanie Yarbrough
**Proofreader:** Julie Van Keuren
**Indexer:** WordCo Indexing Services

**Cover Designer:** Randy Comer
**Interior Designer:** David Futato
**Illustrator:** Kara Ebrahim

September 2013: First Edition

**Revision History for the First Edition:**

2013-09-09: First release

2014-05-23: Second release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449344764* for release details.



ISBN: 978-1-449-34476-4

[LSI]

This Special Edition of *High Performance Browser Networking* contains Chapters 9, 10, 12, and 13. The full book with the latest updates is available on oreilly.com.

# Table of Contents

# Foreword

"Good developers know how things work. Great developers know why things work."

We all resonate with this adage. We want to be that person who understands and can explain the underpinning of the systems we depend on. And yet, if you're a web developer, you might be moving in the opposite direction.

Web development is becoming more and more specialized. What kind of web developer are you? Frontend? Backend? Ops? Big data analytics? UI/UX? Storage? Video? Messaging? I would add "Performance Engineer" making that list of possible specializations even longer.

It's hard to balance studying the foundations of the technology stack with the need to keep up with the latest innovations. And yet, if we don't understand the foundation our knowledge is hollow, shallow. Knowing how to use the topmost layers of the technology stack isn't enough. When the complex problems need to be solved, when the inexplicable happens, the person who understands the foundation leads the way.

That's why *High Performance Browser Networking* is an important book. If you're a web developer, the foundation of your technology stack is the Web and the myriad of networking protocols it rides on: TCP, TLS, UDP, HTTP, and many others. Each of these protocols has its own performance characteristics and optimizations, and to build high performance applications you need to understand why the network behaves the way it does.

Thank goodness you've found your way to this book. I wish I had this book when I started web programming. I was able to move forward by listening to people who understood the why of networking and read specifications to fill in the gaps. High Performance Browser Networking combines the expertise of a networking guru, Ilya Grigorik, with the necessary information from the many relevant specifications, all woven together in one place.

In *High Performance Browser Networking*, Ilya explains many whys of networking: Why latency is the performance bottleneck. Why TCP isn't always the best transport mechanism and UDP might be your better choice. Why reusing connections is a critical optimization. He then goes even further by providing specific actions for improving networking performance. Want to reduce latency? Terminate sessions at a server closer to the client. Want to increase connection reuse? Enable connection keep-alive. The combination of understanding what to do and why it matters turns this knowledge into action.

Ilya explains the foundation of networking and builds on that to introduce the latest advances in protocols and browsers. The benefits of HTTP/2 are explained. XHR is reviewed and its limitations motivate the introduction of Cross-Origin Resource Sharing. Server-Sent Events, WebSockets, and WebRTC are also covered, bringing us up to date on the latest in browser networking.

Viewing the foundation and latest advances in networking from the perspective of performance is what ties the book together. Performance is the context that helps us see the why of networking and translate that into how it affects our website and our users. It transforms abstract specifications into tools that we can wield to optimize our websites and create the best user experience possible. That's important. That's why you should read this book.

—Steve Souders, Head Performance Engineer, Google, 2013

# Preface

The web browser is the most widespread deployment platform available to developers today: it is installed on every smartphone, tablet, laptop, desktop, and every other form factor in between. In fact, current cumulative industry growth projections put us on track for 20 billion connected devices by 2020—each with a browser, and at the very least, WiFi or a cellular connection. The type of platform, manufacturer of the device, or the version of the operating system do not matter—each and every device will have a web browser, which by itself is getting more feature rich each day.

The browser of yesterday looks nothing like what we now have access to, thanks to all the recent innovations: HTML and CSS form the presentation layer, JavaScript is the new assembly language of the Web, and new HTML5 APIs are continuing to improve and expose new platform capabilities for delivering engaging, high-performance applications. There is simply no other technology, or platform, that has ever had the reach or the distribution that is made available to us today when we develop for the browser. And where there is big opportunity, innovation always follows.

In fact, there is no better example of the rapid progress and innovation than the networking infrastructure within the browser. Historically, we have been restricted to simple HTTP request-response interactions, and today we have mechanisms for efficient streaming, bidirectional and real-time communication, ability to deliver custom application protocols, and even peer-to-peer videoconferencing and data delivery directly between the peers—all with a few dozen lines of JavaScript.

The net result? Billions of connected devices, a swelling userbase for existing and new online services, and high demand for high-performance web applications. Speed is a feature, and in fact, for some applications it is *the feature*, and delivering a high-performance web application requires a solid foundation in how the browser and the network interact. That is the subject of this book.

## About This Book

Our goal is to cover what every developer should know about the network: what protocols are being used and their inherent limitations, how to best optimize your applications for the underlying network, and what networking capabilities the browser offers and when to use them.

In the process, we will look at the internals of TCP, UDP, and TLS protocols, and how to optimize our applications and infrastructure for each one. Then we'll take a deep dive into how the wireless and mobile networks work under the hood—this radio thing, it's very different—and discuss its implications for how we design and architect our applications. Finally, we will dissect how the HTTP protocol works under the hood and investigate the many new and exciting networking capabilities in the browser:

- Upcoming HTTP/2 improvements
- New XHR features and capabilities
- Data streaming with Server-Sent Events
- Bidirectional communication with WebSocket
- Peer-to-peer video and audio communication with WebRTC
- Peer-to-peer data exchange with DataChannel

Understanding how the individual bits are delivered, and the properties of each transport and protocol in use are essential knowledge for delivering high-performance applications. After all, if our applications are blocked waiting on the network, then no amount of rendering, JavaScript, or any other form of optimization will help! Our goal is to eliminate this wait time by getting the best possible performance from the network.

*High-Performance Browser Networking* will be of interest to anyone interested in optimizing the delivery and performance of her applications, and more generally, curious minds that are not satisfied with a simple checklist but want to know how the browser and the underlying protocols actually work under the hood. The "how" and the "why" go hand in hand: we'll cover practical advice about configuration and architecture, and we'll also explore the trade-offs and the underlying reasons for each optimization.

Our primary focus is on the protocols and their properties with respect to applications running in the browser. However, all the discussions on TCP, UDP, TLS, HTTP, and just about every other protocol we will cover are also directly applicable to native applications, regardless of the platform.

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**`Constant width bold`**
: Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

## Safari® Books Online


*Safari Books Online* is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/high-performance-browser*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

## Content Updates

### May 23, 2014

1. Added new section on using TLS False Start to optimize TLS handshake.
2. Added new section on benefits of TLS Forward Secrecy.
3. Updated TLS record size optimization with recommendation to use dynamic record sizing.

4. Updated WebRTC code examples to use latest authentication and callback syntax.
5. Updated SPDY roadmap reference to 2014 timelines.
6. Fixed odd/even stream ID references in Chapter 3.
7. Fixed spelling mistakes in text and diagrams.

# PART I
# HTTP

CHAPTER 1

# Brief History of HTTP

The Hypertext Transfer Protocol (HTTP) is one of the most ubiquitous and widely adopted application protocols on the Internet: it is the common language between clients and servers, enabling the modern web. From its simple beginnings as a single keyword and document path, it has become the protocol of choice not just for browsers, but for virtually every Internet-connected software and hardware application.

In this chapter, we will take a brief historical tour of the evolution of the HTTP protocol. A full discussion of the varying HTTP semantics is outside the scope of this book, but an understanding of the key design changes of HTTP, and the motivations behind each, will give us the necessary background for our discussions on HTTP performance, especially in the context of the many upcoming improvements in HTTP/2.

## HTTP 0.9: The One-Line Protocol

The original HTTP proposal by Tim Berners-Lee was designed with *simplicity in mind* as to help with the adoption of his other nascent idea: the World Wide Web. The strategy appears to have worked: aspiring protocol designers, take note.

In 1991, Berners-Lee outlined the motivation for the new protocol and listed several high-level design goals: file transfer functionality, ability to request an index search of a hypertext archive, format negotiation, and an ability to refer the client to another server. To prove the theory in action, a simple prototype was built, which implemented a small subset of the proposed functionality:

- Client request is a single ASCII character string.
- Client request is terminated by a carriage return (CRLF).
- Server response is an ASCII character stream.

- Server response is a hypertext markup language (HTML).
- Connection is terminated after the document transfer is complete.

However, even that sounds a lot more complicated than it really is. What these rules enable is an extremely simple, Telnet-friendly protocol, which some web servers support to this very day:

```
$> telnet google.com 80

Connected to 74.125.xxx.xxx

GET /about/

(hypertext response)
(connection closed)
```

The request consists of a single line: `GET` method and the path of the requested document. The response is a single hypertext document—no headers or any other metadata, just the HTML. It really couldn't get any simpler. Further, since the previous interaction is a subset of the intended protocol, it unofficially acquired the HTTP 0.9 label. The rest, as they say, is history.

From these humble beginnings in 1991, HTTP took on a life of its own and evolved rapidly over the coming years. Let us quickly recap the features of HTTP 0.9:

- Client-server, request-response protocol.
- ASCII protocol, running over a TCP/IP link.
- Designed to transfer hypertext documents (HTML).
- The connection between server and client is closed after every request.

Popular web servers, such as Apache and Nginx, still support the HTTP 0.9 protocol—in part, because there is not much to it! If you are curious, open up a Telnet session and try accessing google.com, or your own favorite site, via HTTP 0.9 and inspect the behavior and the limitations of this early protocol.

## HTTP/1.0: Rapid Growth and Informational RFC

The period from 1991 to 1995 is one of rapid coevolution of the HTML specification, a new breed of software known as a "web browser," and the emergence and quick growth of the consumer-oriented public Internet infrastructure.

### The Perfect Storm: Internet Boom of the Early 1990s

Building on Tim Berner-Lee's initial browser prototype, a team at the National Center of Supercomputing Applications (NCSA) decided to implement their own version. With that, the first popular browser was born: NCSA Mosaic. One of the programmers on the NCSA team, Marc Andreessen, partnered with Jim Clark to found Mosaic Communications in October 1994. The company was later renamed Netscape, and it shipped Netscape Navigator 1.0 in December 1994. By this point, it was already clear that the World Wide Web was bound to be *much more* than just an academic curiosity.

In fact, that same year the first World Wide Web conference was organized in Geneva, Switzerland, which led to the creation of the World Wide Web Consortium (W3C) to help guide the evolution of HTML. Similarly, a parallel HTTP Working Group (HTTP-WG) was established within the IETF to focus on improving the HTTP protocol. Both of these groups continue to be instrumental to the evolution of the Web.

Finally, to create the perfect storm, CompuServe, AOL, and Prodigy began providing dial-up Internet access to the public within the same 1994–1995 time frame. Riding on this wave of rapid adoption, Netscape made history with a wildly successful IPO on August 9, 1995—the Internet boom had arrived, and everyone wanted a piece of it!

The growing list of desired capabilities of the nascent Web and their use cases on the public Web quickly exposed many of the fundamental limitations of HTTP 0.9: we needed a protocol that could serve more than just hypertext documents, provide richer metadata about the request and the response, enable content negotiation, and more. In turn, the nascent community of web developers responded by producing a large number of experimental HTTP server and client implementations through an ad hoc process: implement, deploy, and see if other people adopt it.

From this period of rapid experimentation, a set of best practices and common patterns began to emerge, and in May 1996 the HTTP Working Group (HTTP-WG) published RFC 1945, which documented the "common usage" of the many HTTP/1.0 implementations found in the wild. Note that this was only an informational RFC: HTTP/1.0 as we know it is not a formal specification or an Internet standard!

Having said that, an example HTTP/1.0 request should look very familiar:

```
$> telnet website.org 80

Connected to xxx.xxx.xxx.xxx

GET /rfc/rfc1945.txt HTTP/1.0 ❶
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
Accept: */*

HTTP/1.0 200 OK ❷
Content-Type: text/plain
Content-Length: 137582
Expires: Thu, 01 Dec 1997 16:00:00 GMT
Last-Modified: Wed, 1 May 1996 12:45:26 GMT
Server: Apache 0.84

(plain-text response)
(connection closed)
```

❶ Request line with HTTP version number, followed by request headers

❷ Response status, followed by response headers

The preceding exchange is not an exhaustive list of HTTP/1.0 capabilities, but it does illustrate some of the key protocol changes:

- Request may consist of multiple newline separated header fields.
- Response object is prefixed with a response status line.
- Response object has its own set of newline separated header fields.
- Response object is not limited to hypertext.
- The connection between server and client is closed after every request.

Both the request and response headers were kept as ASCII encoded, but the response object itself could be of any type: an HTML file, a plain text file, an image, or any other content type. Hence, the "hypertext transfer" part of HTTP became a misnomer not long after its introduction. In reality, HTTP has quickly evolved to become a *hypermedia transport*, but the original name stuck.

In addition to media type negotiation, the RFC also documented a number of other commonly implemented capabilities: content encoding, character set support, multi-part types, authorization, caching, proxy behaviors, date formats, and more.

Almost every server on the Web today can and will still speak HTTP/1.0. Except that, by now, you should know better! Requiring a new TCP connection per request imposes a significant performance penalty on HTTP/1.0; see ???, followed by ???.

## HTTP/1.1: Internet Standard

The work on turning HTTP into an official IETF Internet standard proceeded in parallel with the documentation effort around HTTP/1.0 and happened over a period of roughly four years: between 1995 and 1999. In fact, the first official HTTP/1.1 standard is defined in RFC 2068, which was officially released in January 1997, roughly six months after the publication of HTTP/1.0. Then, two and a half years later, in June of 1999, a number of improvements and updates were incorporated into the standard and were released as RFC 2616.

The HTTP/1.1 standard resolved a lot of the protocol ambiguities found in earlier versions and introduced a number of critical performance optimizations: keepalive connections, chunked encoding transfers, byte-range requests, additional caching mechanisms, transfer encodings, and request pipelining.

With these capabilities in place, we can now inspect a typical HTTP/1.1 session as performed by any modern HTTP browser and client:

```
$> telnet website.org 80
Connected to xxx.xxx.xxx.xxx

GET /index.html HTTP/1.1 ❶
Host: website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)

HTTP/1.1 200 OK ❷
Server: nginx/1.0.11
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Via: HTTP/1.1 GWA
Date: Wed, 25 Jul 2012 20:23:35 GMT
Expires: Wed, 25 Jul 2012 20:23:35 GMT
Cache-Control: max-age=0, no-cache
Transfer-Encoding: chunked

100 ❸
<!doctype html>
(snip)
```

```
100
(snip)

0 ❹

GET /favicon.ico HTTP/1.1 ❺
Host: www.website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: */*
Referer: http://website.org/
Connection: close ❻
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)

HTTP/1.1 200 OK ❼
Server: nginx/1.0.11
Content-Type: image/x-icon
Content-Length: 3638
Connection: close
Last-Modified: Thu, 19 Jul 2012 17:51:44 GMT
Cache-Control: max-age=315360000
Accept-Ranges: bytes
Via: HTTP/1.1 GWA
Date: Sat, 21 Jul 2012 21:35:22 GMT
Expires: Thu, 31 Dec 2037 23:55:55 GMT
Etag: W/PSA-GAu26oXbDi

(icon data)
(connection closed)
```

❶ Request for HTML file, with encoding, charset, and cookie metadata

❷ Chunked response for original HTML request

❸ Number of octets in the chunk expressed as an ASCII hexadecimal number (256 bytes)

❹ End of chunked stream response

❺ Request for an icon file made on same TCP connection

❻ Inform server that the connection will not be reused

❼ Icon response, followed by connection close

Phew, there is a lot going on in there! The first and most obvious difference is that we have two object requests, one for an HTML page and one for an image, both delivered over a single connection. This is connection keepalive in action, which allows us to reuse the existing TCP connection for multiple requests to the same host and deliver a much faster end-user experience; see ???.

To terminate the persistent connection, notice that the second client request sends an explicit `close` token to the server via the `Connection` header. Similarly, the server can notify the client of the intent to close the current TCP connection once the response is transferred. Technically, either side can terminate the TCP connection without such signal at any point, but clients and servers should provide it whenever possible to enable better connection reuse strategies on both sides.

HTTP/1.1 changed the semantics of the HTTP protocol to use connection keepalive by default. Meaning, unless told otherwise (via `Connection: close` header), the server should keep the connection open by default.

However, this same functionality was also backported to HTTP/1.0 and enabled via the `Connection: Keep-Alive` header. Hence, if you are using HTTP/1.1, technically you don't need the `Connection: Keep-Alive` header, but many clients choose to provide it nonetheless.

Additionally, the HTTP/1.1 protocol added content, encoding, character set, and even language negotiation, transfer encoding, caching directives, client cookies, plus a dozen other capabilities that can be negotiated on each request.

We are not going to dwell on the semantics of every HTTP/1.1 feature. This is a subject for a dedicated book, and many great ones have been written already. Instead, the previous example serves as a good illustration of both the quick progress and evolution of HTTP, as well as the intricate and complicated dance of every client-server exchange. There is a lot going on in there!

For a good reference on all the inner workings of the HTTP protocol, check out O'Reilly's *HTTP: The Definitive Guide* by David Gourley and Brian Totty.

## HTTP/2: Improving Transport Performance

Since its publication, RFC 2616 has served as a foundation for the unprecedented growth of the Internet: billions of devices of all shapes and sizes, from desktop computers to the tiny web devices in our pockets, speak HTTP every day to deliver news, video, and millions of other web applications we have all come to depend on in our lives.

What began as a simple, one-line protocol for retrieving hypertext quickly evolved into a generic hypermedia transport, and now a decade later can be used to power just about any use case you can imagine. Both the ubiquity of servers that can speak the protocol

and the wide availability of clients to consume it means that many applications are now designed and deployed exclusively on top of HTTP.

Need a protocol to control your coffee pot? RFC 2324 has you covered with the Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)—originally an April Fools' Day joke by IETF, and increasingly anything but a joke in our new hyper-connected world.

> The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol that can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.
>
> — RFC 2616: HTTP/1.1
> *June 1999*

The simplicity of the HTTP protocol is what enabled its original adoption and rapid growth. In fact, it is now not unusual to find embedded devices—sensors, actuators, and coffee pots alike—using HTTP as their primary control and data protocols. But under the weight of its own success and as we increasingly continue to migrate our everyday interactions to the Web—social, email, news, and video, and increasingly our entire personal and job workspaces—it has also begun to show signs of stress. Users and web developers alike are now demanding near real-time responsiveness and protocol performance from HTTP/1.1, which it simply cannot meet without some modifications.

To meet these new challenges, HTTP must continue to evolve, and hence the HTTPbis working group announced a new initiative for HTTP/2 in early 2012:

> There is emerging implementation experience and interest in a protocol that retains the semantics of HTTP without the legacy of HTTP/1.x message framing and syntax, which have been identified as hampering performance and encouraging misuse of the underlying transport.
>
> The working group will produce a specification of a new expression of HTTP's current semantics in ordered, bi-directional streams. As with HTTP/1.x, the primary target transport is TCP, but it should be possible to use other transports.
>
> — HTTP/2 charter
> *January 2012*

The primary focus of HTTP/2 is on improving transport performance and enabling both lower latency and higher throughput. The major version increment sounds like a big step, which it is and will be as far as performance is concerned, but it is important to note that none of the high-level protocol semantics are affected: all HTTP headers, values, and use cases are the same.

Any existing website or application can and will be delivered over HTTP/2 without modification: you do not need to modify your application markup to take advantage of HTTP/2. The HTTP servers will have to speak HTTP/2, but that should be a transparent

upgrade for the majority of users. The only difference if the working group meets its goal, should be that our applications are delivered with lower latency and better utilization of the network link!

Having said that, let's not get ahead of ourselves. Before we get to the new HTTP/2 protocol features, it is worth taking a step back and examining our existing deployment and performance best practices for HTTP/1.1. The HTTP/2 working group is making fast progress on the new specification, but even if the final standard was already done and ready, we would still have to support older HTTP/1.1 clients for the foreseeable future—realistically, a decade or more.

# CHAPTER 2
# Primer on Web Performance

In any complex system, a large part of the performance optimization process is the untangling of the interactions between the many distinct and separate layers of the system, each with its own set of constraints and limitations. So far, we have examined a number of individual networking components in close detail—different physical delivery methods and transport protocols—and now we can turn our attention to the larger, end-to-end picture of web performance optimization:

- Impact of latency and bandwidth on web performance
- Transport protocol (TCP) constraints imposed on HTTP
- Features and shortcomings of the HTTP protocol itself
- Web application trends and performance requirements
- Browser constraints and optimizations

Optimizing the interaction among all the different layers is not unlike solving a family of equations, each dependent on the others, but nonetheless yielding many possible solutions. There is no one fixed set of recommendations or best practices, and the individual components continue to evolve: browsers are getting faster, user connectivity profiles change, and web applications continue to grow in their scope, ambition, and complexity.

Hence, before we dive into enumerating and analyzing individual performance best practices, it is important to step back and define what the problem really is: what a modern web application is, what tools we have at our disposal, how we measure web-performance, and which parts of the system are helping and hindering our progress.

## Hypertext, Web Pages, and Web Applications

The evolution of the Web over the course of the last few decades has given us at least three different classes of experience: the hypertext document, rich media web page, and interactive web application. Admittedly, the line between the latter two may at times be blurry to the user, but from a performance point of view, each requires a very different approach to our conversation, metrics, and the definition of performance.

*Hypertext document*
: Hypertext documents were the genesis of the World Wide Web, the plain text version with some basic formatting and support for hyperlinks. This may not sound exciting by modern standards, but it proved the premise, vision, and the great utility of the World Wide Web.

*Web page*
: The HTML working group and the early browser vendors extended the definition of hypertext to support additional hypermedia resources, such as images and audio, and added many other primitives for richer layouts. The era of the *web page* has arrived, allowing us to produce rich visual layouts with various media types: visually beautiful but mostly non-interactive, not unlike a printed page.

*Web application*
: Addition of JavaScript and later revolutions of Dynamic HTML (DHTML) and AJAX shook things up once more and transformed the simple web page into an interactive *web application*, which allowed it to respond to the user directly within the browser. This paved the way for the first full-fledged browser applications, such as Outlook Web Access (originator of XMLHTTP support in IE5), ushering in a new era of complex dependency graphs of scripts, stylesheets, and markup.

An HTTP 0.9 session consisted of a single document request, which was perfectly sufficient for delivery of hypertext: single document, one TCP connection, followed by connection close. Consequently, tuning for performance was as simple as optimizing for a single HTTP request over a short-lived TCP connection.

The advent of the *web page* changed the formula from delivery of a single document to the document plus its dependent resources. Consequently, HTTP/1.0 introduced the notion of HTTP metadata (headers), and HTTP/1.1 enhanced it with a variety of performance-oriented primitives, such as well-defined caching, keepalive, and more. Hence, multiple TCP connections are now potentially at play, and the key performance metric has shifted from *document load time* to *page load time*, which is commonly abbreviated as PLT.

The simplest definition of PLT is "the time until the loading spinner stops spinning in the browser." A more technical definition is time to onload event in the browser, which is an event fired by the browser once the document and all of its dependent resources (JavaScript, images, etc.) have finished loading.

Finally, the web application transformed the simple web page, which used media as an enhancement to the primary content in the markup, into a complex dependency graph: markup defines the basic structure, stylesheets define the layout, and scripts build up the resulting interactive application and respond to user input, potentially modifying both styles and markup in the process.

Consequently, page load time, which has been the de facto metric of the web performance world, is also an increasingly insufficient performance benchmark: we are no longer building pages, we are building dynamic and interactive web applications. Instead of, or in addition to, measuring the time to load each and every resource (PLT), we are now interested in answering application-specific questions:

- What are the milestones in the loading progress of the application?
- What are the times to first interaction by the user?
- What are the interactions the user should engage in?
- What are the engagement and conversion rates for each user?

The success of your performance and optimization strategy is directly correlated to your ability to define and iterate on application-specific benchmarks and criteria. Nothing beats application-specific knowledge and measurements, especially when linked to bottom-line goals and metrics of your business.

### DOM, CSSOM, and JavaScript

What exactly do we mean by "complex dependency graph of scripts, stylesheets, and markup" found in a modern web application? To answer this question, we need to take a quick detour into browser architecture and investigate how the parsing, layout, and scripting pipelines have to come together to paint the pixels to the screen.

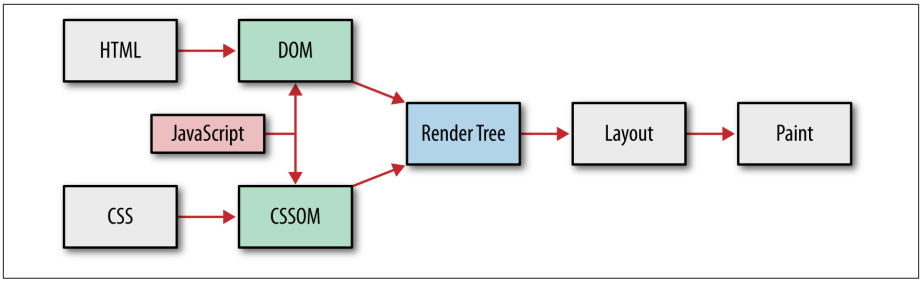



*Figure 2-1. Browser processing pipeline: HTML, CSS, and JavaScript*

The parsing of the HTML document is what constructs the Document Object Model (DOM). In parallel, there is an oft-forgotten cousin, the CSS Object Model (CSSOM), which is constructed from the specified stylesheet rules and resources. The two are then combined to create the "render tree," at which point the browser has enough information to perform a layout and paint something to the screen. So far, so good.

However, this is where we must, unfortunately, introduce our favorite friend and foe: JavaScript. Script execution can issue a synchronous `doc.write` and block DOM parsing and construction. Similarly, scripts can query for a computed style of any object, which means that JavaScript can also block on CSS. Consequently, the construction of DOM and CSSOM objects is frequently intertwined: DOM construction cannot proceed until JavaScript is executed, and JavaScript execution cannot proceed until CSSOM is available.

The performance of your application, especially the first load and the "time to render" depends directly on how this dependency graph between markup, stylesheets, and JavaScript is resolved. Incidentally, recall the popular "styles at the top, scripts at the bottom" best practice? Now you know why! Rendering and script execution are blocked on stylesheets; get the CSS down to the user as quickly as you can.

## Anatomy of a Modern Web Application

What does a modern web application look like after all? HTTP Archive can help us answer this question. The project tracks how the Web is built by periodically crawling

the most popular sites (300,000+ from Alexa Top 1M) and recording and aggregating analytics on the number of used resources, content types, headers, and other metadata for each individual destination.

An average web application, as of early 2013, is composed of the following:

- 90 requests, fetched from 15 hosts, with 1,311 KB total transfer size
  - — HTML: 10 requests, 52 KB
  - — Images: 55 requests, 812 KB
  - — JavaScript: 15 requests, 216 KB
  - — CSS: 5 requests, 36 KB
  - — Other: 5 requests, 195 KB

By the time you read this, the preceding numbers have already changed and have grown even larger (Figure 2-2); the upward climb has been a stable and reliable trend with no signs of stopping. However, exact request and kilobyte count aside, it is the order of magnitude of these individual components that warrants some careful contemplation: an average web application is now well over 1 MB in size and is composed of roughly 100 sub-resources delivered from over 15 different hosts!

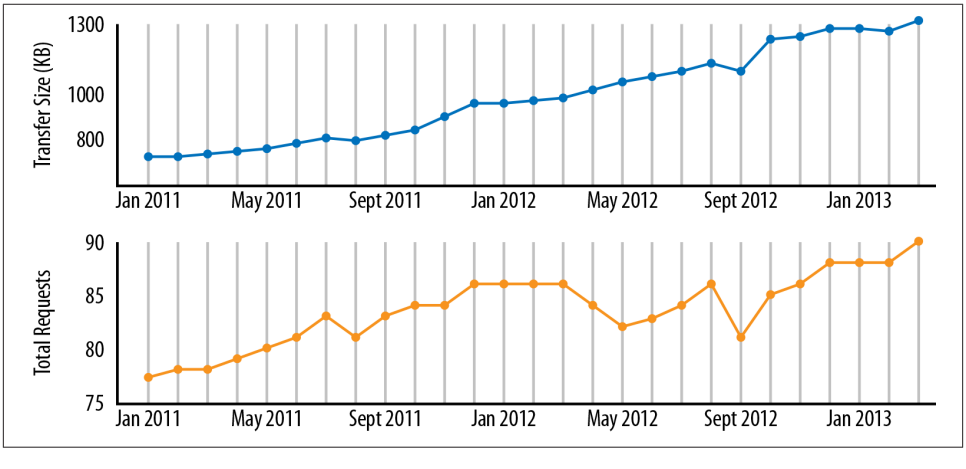



*Figure 2-2. Average transfer size and number of requests (HTTP Archive)*

Unlike their desktop counterparts, web applications do not require a separate installation process: type in the URL, hit Enter, and we are up and running! However, desktop applications pay the installation cost just once, whereas web applications are running the "installation process" on each and every visit—resource downloads, DOM and CSSOM construction, and JavaScript execution. No wonder web performance is such a fast-growing field and a hot topic of discussion! Hundreds of resources, megabytes of

data, dozens of different hosts, all of which must come together in hundreds of milliseconds to facilitate the desired instant web experience.

## Speed, Performance, and Human Perception

Speed and performance are relative terms. Each application dictates its own set of requirements based on business criteria, context, user expectations, and the complexity of the task that must be performed. Having said that, if the application must react and respond to a user, then we must plan and design for specific, *user-centric perceptual processing time constants*. Despite the ever-accelerating pace of life, or at least the feeling of it, our reaction times remain constant (Table 2-1), regardless of type of application (online or offline), or medium (laptop, desktop, or mobile device).

*Table 2-1. Time and user perception*

| Delay | User perception |
|---|---|
| 0–100 ms | Instant |
| 100–300 ms | Small perceptible delay |
| 300–1000 ms | Machine is working |
| 1,000+ ms | Likely mental context switch |
| 10,000+ ms | Task is abandoned |

The preceding table helps explain the unofficial rule of thumb in the web performance community: render pages, or at the very least provide visual feedback, in under 250 milliseconds to keep the user engaged!

For an application to feel instant, a perceptible response to user input must be provided within hundreds of milliseconds. After a second or more, the user's flow and engagement with the initiated task is broken, and after 10 seconds have passed, unless progress feedback is provided, the task is frequently abandoned.

Now, add up the network latency of a DNS lookup, followed by a TCP handshake, and another few roundtrips for a typical web page request, and much, if not all, of our 100–1,000 millisecond latency budget can be easily spent on just the networking overhead; see ???. No wonder so many users, especially when on a mobile or a wireless network, are demanding faster web browsing performance!

Jakob Nielsen's *Usability Engineering* and Steven Seow's *Designing and Engineering Time* are both excellent resources that every developer and designer should read! Time is measured objectively but perceived subjectively, and *experiences can be engineered* to improve perceived performance.

### Translating Web Performance to Dollars and Cents

Speed is a feature, and it is not simply speed for speed's sake. Well-publicized studies from Google, Microsoft, and Amazon all show that web performance translates directly to dollars and cents—e.g., a 2,000 ms delay on Bing search pages decreased per-user revenue by 4.3%!

Similarly, an Aberdeen study of over 160 organizations determined that an extra *one-second* delay in page load times led to 7% loss in conversions, 11% fewer page views, and a 16% decrease in customer satisfaction!

Faster sites yield more page views, higher engagement, and higher conversion rates. However, don't just take our word for it, or put your faith into well-cited industry benchmarks: measure the impact of web performance on your own site, and against your own conversion metrics. If you're wondering how, then keep reading, or skip ahead to "Synthetic and Real-User Performance Measurement" on page 27.

## Analyzing the Resource Waterfall

No discussion on web performance is complete without a mention of the resource waterfall. In fact, the resource waterfall is likely the single most insightful network performance and diagnostics tool at our disposal. Every browser provides some instrumentation to see the resource waterfall, and there are great online tools, such as WebPageTest, which can render it online for a wide variety of different browsers.

WebPageTest.org is an open-source project and a free web service that provides a system for testing the performance of web pages from multiple locations around the world: the browser runs within a virtual machine and can be configured and scripted with a variety of connection and browser-oriented settings. Following the test, the results are then available through a web interface, which makes WebPageTest an indispensable power tool in your web performance toolkit.

To start, it is important to recognize that every HTTP request is composed of a number of separate stages (Figure 2-3): DNS resolution, TCP connection handshake, TLS negotiation (if required), dispatch of the HTTP request, followed by content download. The visual display of these individual stages may differ slightly within each browser, but to keep things simple, we will use the WebPageTest version in this chapter. Make sure to familiarize yourself with the meaning of each color in your favorite browser.

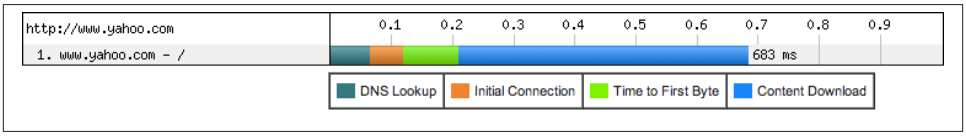



*Figure 2-3. Components of an HTTP request (WebPageTest)*

Close analysis of Figure 2-3 shows that the Yahoo! homepage took 683 ms to download, and over 200 ms of that time was spent waiting on the network, which amounts to 30% of total latency of the request! However, the document request is only the beginning since, as we know, a modern web application also needs a wide variety of resources (Figure 2-4) to produce the final output. To be exact, to load the Yahoo! homepage, the browser will require 52 resources, fetched from 30 different hosts, all adding up to 486 KB in total.

The resource waterfall reveals a number of important insights about the structure of the page and the browser processing pipeline. First off, notice that while the content of the *www.yahoo.com* document is being fetched, new HTTP requests are being dispatched: HTML parsing is performed incrementally, allowing the browser to discover required resources early and dispatch the necessary requests in parallel. Hence, the scheduling of when the resource is fetched is in large part determined by the structure of the markup. The browser may reprioritize some requests, but the incremental discovery of each resource in the document is what creates the distinct resource "waterfall effect."

Second, notice that the "Start Render" (green vertical line) occurs well before all the resources are fully loaded, allowing the user to begin interacting with the page while the page is being built. In fact, the "Document Complete" event (blue vertical line), also fires early and well before the remaining assets are loaded. In other words, the browser spinner has stopped spinning, the user is able to continue with his task, but the Yahoo! homepage is progressively filling in additional content, such as advertising and social widgets, in the background.

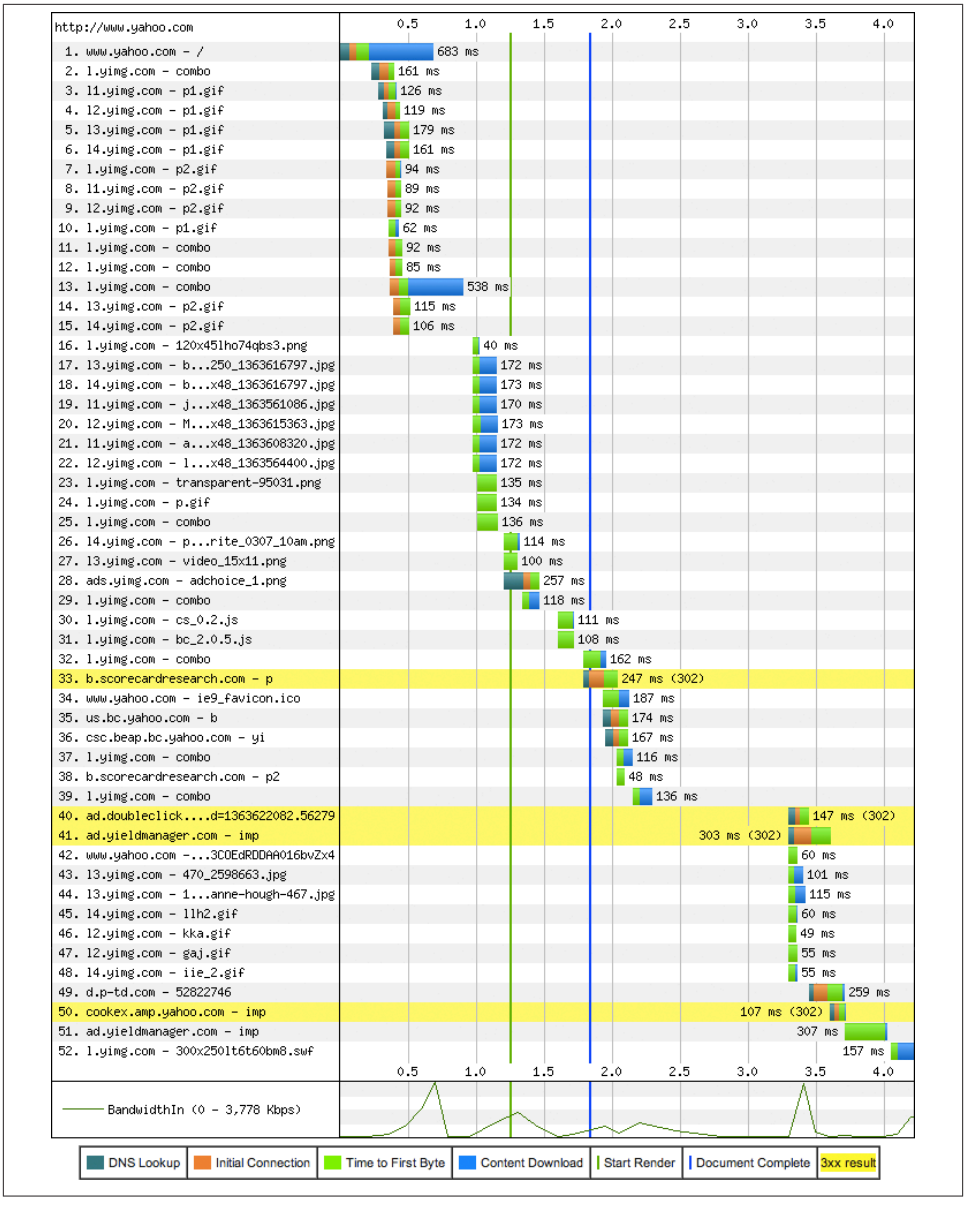



*Figure 2-4. Yahoo.com resource waterfall (WebPageTest, March 2013)*

The difference between the first render time, document complete, and the time to finish fetching the last resource in the preceding example is a great illustration of the necessary context when discussing different web performance metrics. Which of those three metrics is the right one to track? There is no one single answer; each application is different! Yahoo! engineers have chosen to optimize the page to take advantage of incremental loading to allow the user to begin consuming the important content earlier, and in doing so they had to apply application-specific knowledge about which content is critical and which can be filled in later.

Different browsers implement different logic for when, and in which order, the individual resource requests are dispatched. As a result, the performance of the application will vary from browser to browser.

Tip: WebPageTest allows you to select both the location and the make and version of the browser when running the test!

The network waterfall is a power tool that can help reveal the chosen optimizations, or lack thereof, for any page or application. The previous process of analyzing and optimizing the resource waterfall is often referred to as *front-end performance* analysis and optimization. However, the name may be an unfortunate choice, as it misleads many to believe that all performance bottlenecks are now on the client. In reality, while JavaScript, CSS, and rendering pipelines are critical and resource-intensive steps, the server response times and network latency ("back-end performance") are no less critical for optimizing the resource waterfall. After all, you can't parse or execute a resource that is blocked on the network!

To illustrate this in action, we only have to switch from the *resource waterfall* to the *connection view* (Figure 2-5) provided by WebPageTest.

Unlike the resource waterfall, where each record represents an individual HTTP request, the connection view shows the life of each TCP connection—all 30 of them in this case—used to fetch the resources for the Yahoo! homepage. Does anything stand out? Notice that the download time, indicated in blue, is but a small fraction of the total latency of each connection: there are 15 DNS lookups, 30 TCP handshakes, and a lot of network latency (indicated in green) while waiting to receive the first byte of each response.

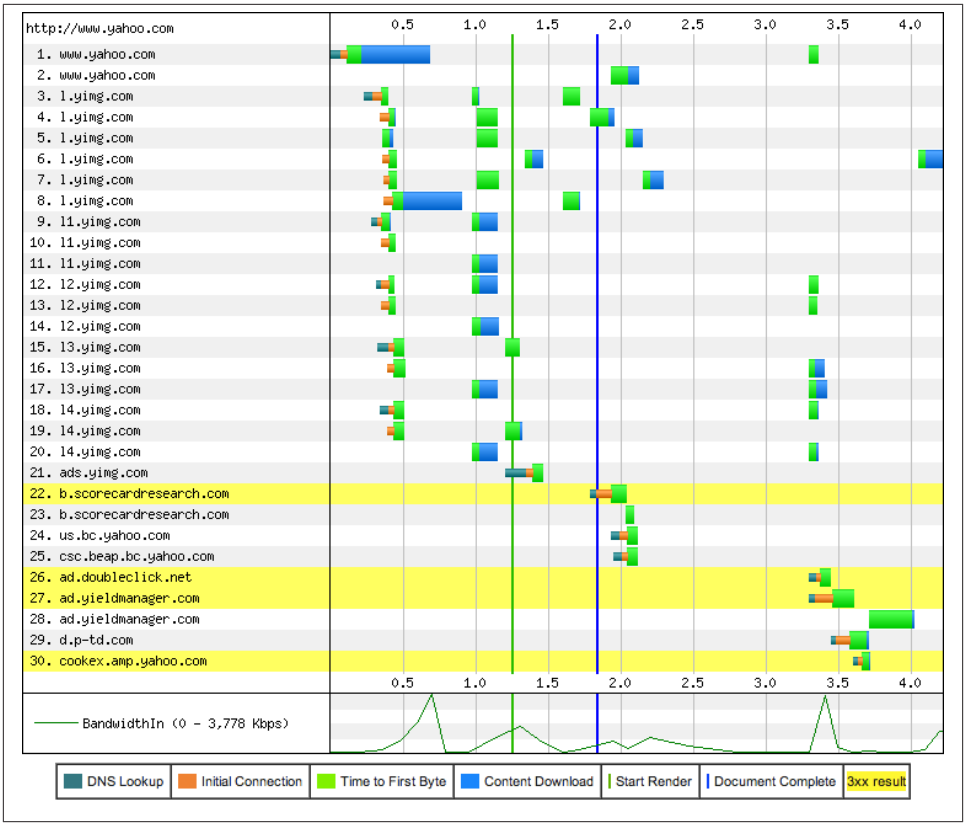



*Figure 2-5. Yahoo.com connection view (WebPageTest, March 2013)*

Wondering why some requests are showing the green bar (time to first byte) only? Many responses are very small, and consequently the download time does not register on the diagram. In fact, for many requests, response times are often dominated by the roundtrip latency and server processing times.

Finally, we have saved the best for last. The *real* surprise to many is found at the bottom of the connection view: examine the bandwidth utilization chart in Figure 2-5. With the exception of a few short data bursts, the utilization of the available connection is very low—it appears that we are not limited by bandwidth of our connection! Is this an anomaly, or worse, a browser bug? Unfortunately, it is neither. Turns out, bandwidth is not the limiting performance factor for most web applications. Instead, the bottleneck is the network roundtrip latency between the client and the server.

# Performance Pillars: Computing, Rendering, Networking

The execution of a web program primarily involves three tasks: fetching resources, page layout and rendering, and JavaScript execution. The rendering and scripting steps follow a single-threaded, interleaved model of execution; it is not possible to perform concurrent modifications of the resulting Document Object Model (DOM). Hence, optimizing how the rendering and script execution runtimes work together, as we saw in "DOM, CSSOM, and JavaScript" on page 16, is of critical importance.

However, optimizing JavaScript execution and rendering pipelines also won't do much good if the browser is blocked on the network, waiting for the resources to arrive. Fast and efficient delivery of network resources is the performance keystone of each and every application running in the browser.

But, one might ask, Internet speeds are getting faster by the day, so won't this problem solve itself? Yes, our applications are growing larger, but if the global average speed is already at 3.1 Mbps (???) and growing, as evidenced by ubiquitous advertising by every ISP and mobile carrier, why bother, right? Unfortunately, as you might intuit, and as the Yahoo! example shows, if that were the case then you wouldn't be reading this book. Let's take a closer look.

For a detailed discussion of the trends and interplay of bandwidth and latency, refer back to the "Primer on Latency and Bandwidth" in ???.

## More Bandwidth Doesn't Matter (Much)

Hold your horses; of course bandwidth matters! After all, every commercial by our local ISP and mobile carrier continues to remind us of its many benefits: faster downloads, uploads, and streaming, all at up to speeds of *[insert latest number here]* Mbps!

Access to higher bandwidth data rates is always good, especially for cases that involve bulk data transfers: video and audio streaming or any other type of large data transfer. However, when it comes to everyday web browsing, which requires fetching hundreds of relatively small resources from dozens of different hosts, roundtrip latency is the limiting factor:

- Streaming an HD video from the Yahoo! homepage is bandwidth limited.
- Loading and rendering the Yahoo! homepage is latency limited.

Depending on the quality and the encoding of the video you are trying to stream, you may need anywhere from a few hundred Kbps to several Mbps in bandwidth capacity —e.g., 3+ Mbps for an HD 1080p video stream. This data rate is now within reach for

many users, which is evidenced by the growing popularity of streaming video services such as Netflix. Why, then, would downloading a much, much smaller web application be such a challenge for a connection capable of streaming an HD movie?

## Latency as a Performance Bottleneck

We have already covered all the necessary topics in preceding chapters to make a good qualitative theory as to why latency may be the limiting factor for everyday web browsing. However, a picture is worth a thousand words, so let's examine the results of a quantitative study performed by Mike Belshe (Figure 2-6), one of the creators of the SPDY protocol, on the impact of varying bandwidth vs. latency on the page load times of some of the most popular destinations on the Web.

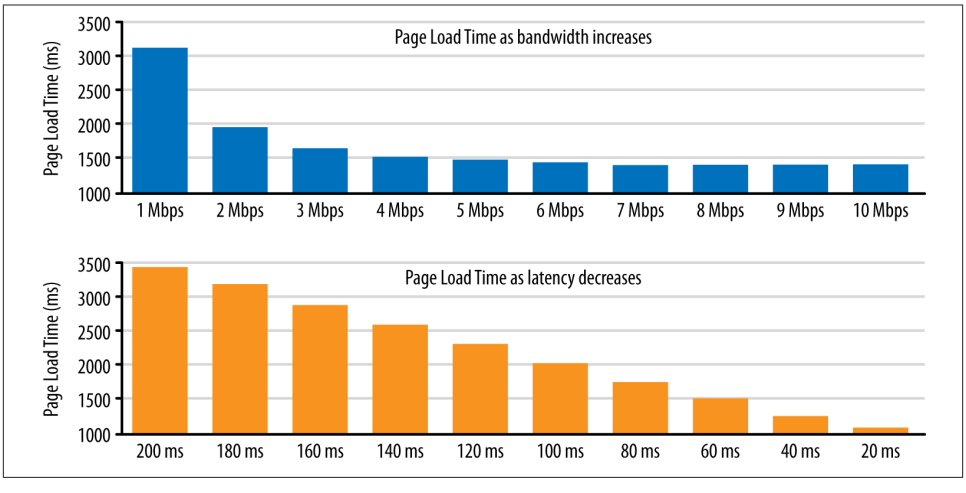



*Figure 2-6. Page load time vs. bandwidth and latency*

This study by Mike Belshe served as a launching point for the development of the SPDY protocol at Google, which later became the foundation of the HTTP/2 protocol.

In the first test, the connection latency is held fixed, and the connection bandwidth is incrementally increased from 1 Mbps up to 10 Mbps. Notice that at first, upgrading the connection from 1 to 2 Mbps nearly halves the page loading time—exactly the result we want to see. However, following that, each incremental improvement in bandwidth yields diminishing returns. By the time the available bandwidth exceeds 5 Mbps, we are looking at single-digit percent improvements, and upgrading from 5 Mbps to 10 Mbps results in a mere 5% improvement in page loading times!

Akamai's broadband speed report (???) shows that an average consumer in the United States is already accessing the Web with 5 Mbps+ of available bandwidth—a number that many other countries are quickly approaching or have surpassed already. Ergo, we are led to conclude that an average consumer in the United States would not benefit *much* from upgrading the available bandwidth of her connection if she is interested in improving her web browsing speeds. She may be able to stream or upload larger media files more quickly, but the pages containing those files will not load noticeably faster: *bandwidth doesn't matter, much*.

However, the latency experiment tells an entirely different story: for every 20 millisecond improvement in latency, we have a linear improvement in page loading times! Perhaps it is latency we should be optimizing for when deciding on an ISP, and not just bandwidth?

> To speed up the Internet at large, we should look for more ways to bring down RTT. What if we could reduce cross-atlantic RTTs from 150 ms to 100 ms? This would have a larger effect on the speed of the internet than increasing a user's bandwidth from 3.9 Mbps to 10 Mbps or even 1 Gbps.
>
> Another approach to reducing page load times would be to reduce the number of round trips required per page load. Today, web pages require a certain amount of back and forth between the client and server. The number of round trips is largely due to the handshakes to start communicating between client and server (e.g. DNS, TCP, HTTP), and also round trips induced by the communication protocols (e.g. TCP slow start). If we can improve protocols to transfer this data with fewer round trips, we should also be able to improve page load times. This is one of the goals of SPDY.
>
> — Mike Belshe
> *More Bandwidth Doesn't Matter (Much)*

The previous results are a surprise to many, but they really should not be, as they are a direct consequence of the performance characteristics of the underlying protocols: TCP handshakes, flow and congestion control, and head-of-line blocking due to packet loss. Most of the HTTP data flows consist of small, bursty data transfers, whereas TCP is optimized for long-lived connections and bulk data transfers. Network roundtrip time is the limiting factor in TCP throughput and performance in most cases; see ???. Consequently, latency is also the performance bottleneck for HTTP and most web applications delivered over it.

If latency is the limiting performance factor for most wired connections then, as you might intuit, it is an even more important performance bottleneck for wireless clients: wireless latencies are significantly higher, making networking optimization a critical priority for the mobile web.

## Synthetic and Real-User Performance Measurement

If we can measure it, we can improve it. The question is, are we measuring the right criteria, and is the process sound? As we noted earlier, measuring the performance of a modern web application is a nontrivial challenge: there is no one single metric that holds true for every application, which means that we must carefully define custom metrics in each case. Then, once the criteria are established, we must gather the performance data, which should be done through a combination of synthetic and real-user performance measurement.

Broadly speaking, synthetic testing refers to any process with a controlled measurement environment: a local build process running through a performance suite, load testing against staging infrastructure, or a set of geo-distributed monitoring servers that periodically perform a set of scripted actions and log the outcomes. Each and every one of these tests may test a different piece of the infrastructure (e.g., application server throughput, database performance, DNS timing, and so on), and serves as a stable baseline to help detect regressions or narrow in on a specific component of the system.

When configured well, synthetic testing provides a controlled and reproducible performance testing environment, which makes it a great fit for identifying and fixing performance regressions before they reach the user. Tip: identify your key performance metrics and set a "budget" for each one as part of your synthetic testing. If the budget is exceeded, raise an alarm!

However, synthetic testing is not sufficient to identify all performance bottlenecks. Specifically, the problem is that the gathered measurements are not representative of the wide diversity of the real-world factors that will determine the final user experience with the application. Some contributing factors to this gap include the following:

- Scenario and page selection: replicating real user navigation patterns is hard.
- Browser cache: performance may vary widely based on the state of the user's cache.
- Intermediaries: performance may vary based on intermediate proxies and caches.
- Diversity of hardware: wide range of CPU, GPU, and memory performance.
- Diversity of browsers: wide range of browser versions, both old and new.
- Connectivity: continuously changing bandwidth and latency of real connections.

The combination of these and similar factors means that in addition to synthetic testing, we must augment our performance strategy with real-user measurement (RUM) to capture actual performance of our application as experienced by the user. The good news is the W3C Web Performance Working Group has made this part of our data-gathering process a simple one by introducing the Navigation Timing API

(Figure 2-7), which is now supported across many of the modern desktop and mobile browsers.

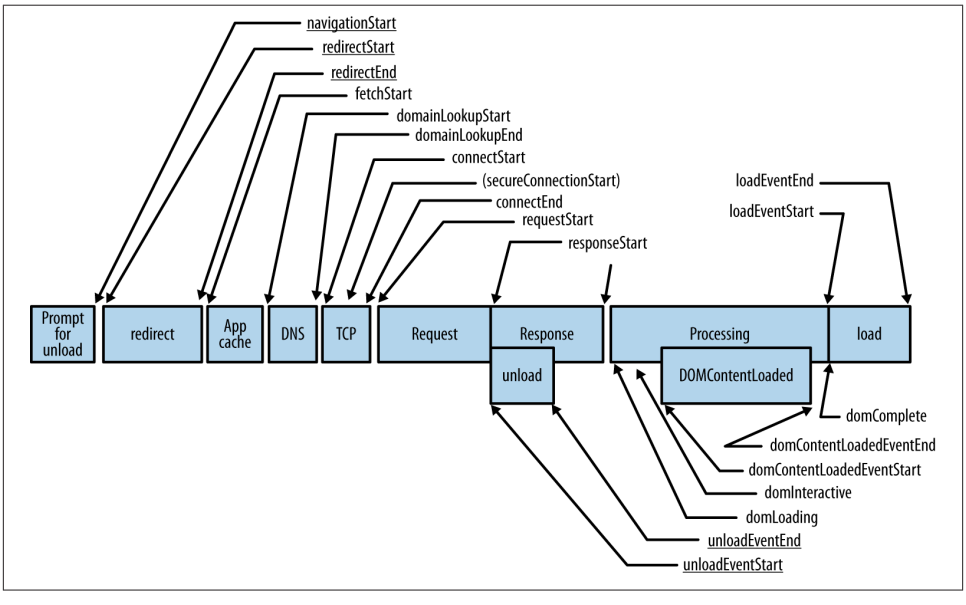



*Figure 2-7. User-specific performance timers exposed by Navigation Timing*

As of early 2013, Navigation Timing is supported by IE9+, Chrome 6+, and Firefox 7+ across desktop and mobile platforms. The notable omissions are the Safari and Opera browsers. For the latest status, see caniuse.com/nav-timing.

The real benefit of Navigation Timing is that it exposes a lot of previously inaccessible data, such as DNS and TCP connect times, with high precision (microsecond timestamps), via a standardized `performance.timing` object in each browser. Hence, the data gathering process is very simple: load the page, grab the timing object from the user's browser, and beacon it back to your analytics servers! By capturing this data, we can observe real-world performance of our applications as seen by real users, on real hardware, and across a wide variety of different networks.

## Analyzing Real User Measurement Data

When analyzing performance data, always look at the underlying distribution of the data: throw away the averages and focus on the histograms, medians, and quantiles. Averages lead to meaningless metrics when analyzing skewed and multimodal distributions. Figure 2-8 shows a hands-on, real-world example of both types of distributions on a single site: skewed distribution for the page load time and a multimodal distribution for the server response time (the two modes are due to cached vs. uncached page generation time by the application server).

| Page Load Time Bucket (sec) | Page Load Sample | Percentage of total |
|---|---|---|
| 0 - 1 | 22 | 5.35% |
| 1 - 3 | 116 | 28.22% |
| 3 - 7 | 148 | 36.01% |
| 7 - 13 | 66 | 16.06% |
| 13 - 21 | 22 | 5.35% |
| 21 - 35 | 14 | 3.41% |
| 35 - 60 | 10 | 2.43% |

| Server Response Time Bucket (sec) | Response Sample | Percentage of total |
|---|---|---|
| 0 - 0.01 | 18 | 4.40% |
| 0.01 - 0.10 | 33 | 8.07% |
| 0.10 - 0.50 | 168 | 41.08% |
| 0.50 - 1 | 22 | 5.38% |
| 1 - 2 | 124 | 30.32% |
| 2 - 5 | 38 | 9.29% |
| 5+ | 6 | 1.47% |

*Figure 2-8. Page load time (skewed) and response time (multimodal) distributions for igvita.com*

Ensure that your analytics tool can provide the right statistical metrics for your performance data. The preceding data was taken from Google Analytics, which provides a histogram view within the standard Site Speed reports. Google Analytics automatically gathers Navigation Timing data when the analytics tracker is installed. Similarly, there are a wide variety of other analytics vendors who offer Navigation Timing data gathering and reporting.

Finally, in addition to Navigation Timing, the W3C Performance Group also standardized two other APIs: User Timing and Resource Timing. Whereas Navigation Timing provides performance timers for root documents only, Resource Timing provides similar performance data for each resource on the page, allowing us to gather the full performance profile of the page. Similarly, User Timing provides a simple JavaScript API to mark and measure application-specific performance metrics with the help of the same high-resolution timers:

```
function init() {
  performance.mark("startTask1"); ❶
  applicationCode1(); ❷
  performance.mark("endTask1");

  logPerformance();
}

function logPerformance() {
  var perfEntries = performance.getEntriesByType("mark");
  for (var i = 0; i < perfEntries.length; i++) { ❸
    console.log("Name: " + perfEntries[i].name +
                " Entry Type: " + perfEntries[i].entryType +
                " Start Time: " + perfEntries[i].startTime +
                " Duration: "   + perfEntries[i].duration  + "\n");
  }
  console.log(performance.timing); ❹
}
```

❶ Store (mark) timestamp with associated name (startTask1).

❷ Execute application code.

❸ Iterate and log user timing data.

❹ Log Navigation Timing object for current page.

The combination of Navigation, Resource, and User timing APIs provides all the necessary tools to instrument and conduct real-user performance measurement for every web application; there is no longer any excuse not to do it right. We optimize what we measure, and RUM and synthetic testing are complementary approaches to help you identify regressions and real-world bottlenecks in the performance and the user experience of your applications.

Custom and application-specific metrics are the key to establishing a sound performance strategy. There is no generic way to measure or define the quality of user experience. Instead, we must define and instrument specific milestones and events in each application, a process that requires collaboration between all the stakeholders in the project: business owners, designers, and developers.

## Browser Optimization

We would be remiss if we didn't mention that a modern browser is much more than a simple network socket manager. Performance is one of the primary competitive features for each browser vendor, and given that the networking performance is such a critical criteria, it should not surprise you that the browsers are getting smarter every day: pre-resolving likely DNS lookups, pre-connecting to likely destinations, pre-fetching and prioritizing critical resources on the page, and more.

The exact list of performed optimizations will differ by browser vendor, but at their core the optimizations can be grouped into two broad classes:

*Document-aware optimization*
: The networking stack is integrated with the document, CSS, and JavaScript parsing pipelines to help identify and prioritize critical network assets, dispatch them early, and get the page to an interactive state as soon as possible. This is often done via resource priority assignments, lookahead parsing, and similar techniques.

*Speculative optimization*
: The browser may learn user navigation patterns over time and perform speculative optimizations in an attempt to predict the likely user actions by pre-resolving DNS names, pre-connecting to likely hostnames, and so on.

The good news is all of these optimizations are done automatically on our behalf and often lead to hundreds of milliseconds of saved network latency. Having said that, it is important to understand how and why these optimizations work under the hood, because we *can* assist the browser and help it do an even better job at accelerating our applications. There are four techniques employed by most browsers:

*Resource pre-fetching and prioritization*
: Document, CSS, and JavaScript parsers may communicate extra information to the network stack to indicate the relative priority of each resource: blocking resources required for first rendering are given high priority, while low-priority requests may be temporarily held back in a queue.

*DNS pre-resolve*
: Likely hostnames are pre-resolved ahead of time to avoid DNS latency on a future HTTP request. A pre-resolve may be triggered through learned navigation history, a user action such as hovering over a link, or other signals on the page.

*TCP pre-connect*
: Following a DNS resolution, the browser may speculatively open the TCP connection in an anticipation of an HTTP request. If it guesses right, it can eliminate another full roundtrip (TCP handshake) of network latency.

*Page pre-rendering*
: Some browsers allow you to hint the likely next destination and can pre-render the entire page in a hidden tab, such that it can be instantly swapped in when the user initiates the navigation.

For a deep dive into how these and other networking optimizations are implemented in Google Chrome, see High Performance Networking in Google Chrome.

From the outside, a modern browser network stack presents itself as simple resource-fetching mechanism, but from the inside, it is an elaborate and a fascinating case study for how to optimize for web performance. So how can we assist the browser in this quest? To start, pay close attention to the structure and the delivery of each page:

- Critical resources such as CSS and JavaScript should be discoverable as early as possible in the document.
- CSS should be delivered as early as possible to unblock rendering and JavaScript execution.
- Noncritical JavaScript should be deferred to avoid blocking DOM and CSSOM construction.
- The HTML document is parsed incrementally by the parser; hence the document should be periodically flushed for best performance.

Further, aside from optimizing the structure of the page, we can also embed additional hints into the document itself to tip off the browser about additional optimizations it can perform on our behalf:

```
<link rel="dns-prefetch" href="//hostname_to_resolve.com"> ❶
<link rel="subresource"  href="/javascript/myapp.js"> ❷
<link rel="prefetch"     href="/images/big.jpeg"> ❸
<link rel="prerender"    href="//example.org/next_page.html"> ❹
```

❶ Pre-resolve specified hostname.

❷ Prefetch critical resource found later on this page.

❸ Prefetch resource for this or future navigation.

❹ Prerender specified page in anticipation of next user destination.

Each of these is a hint for a speculative optimization. The browser does not guarantee that it will act on it, but it may use the hint to optimize its loading strategy. Unfortunately, not all browsers support all hints (Table 2-2), but if they don't, then the hint is treated as a no-op and is harmless; make use of each of the techniques just shown where possible.

*Table 2-2. Speculative browser optimization hints*

| Browser | dns-prefetch | subresource | prefetch | prerender |
|---|---|---|---|---|
| Firefox | 3.5+ | n/a | 3.5+ | n/a |
| Chrome | 1.0+ | 1.0+ | 1.0+ | 13+ |
| Safari | 5.01+ | n/a | n/a | n/a |
| IE | 9+ (prefetch) | n/a | 10+ | 11+ |

Internet Explorer 9 supports DNS pre-fetching, but calls it *prefetch*. In Internet Explorer 10+, *dns-prefetch* and *prefetch* are equivalent, resulting in a DNS pre-fetch in both cases.

To most users and even web developers, the DNS, TCP, and SSL delays are entirely transparent and are negotiated at network layers to which few of us descend. And yet each of these steps is critical to the overall user experience, since each extra network roundtrip can add tens or hundreds of milliseconds of network latency. By helping the browser anticipate these roundtrips, we can remove these bottlenecks and deliver much faster and better web applications.

### Optimizing Time to First Byte (TTFB) for Google Search

The HTML document is parsed incrementally by the browser, which means that the server can and should flush available document markup as frequently as possible. This enables the client to discover and begin fetching critical resources as soon as possible.

Google Search offers one of the best examples of the benefits of this technique: when a search request arrives, the server immediately flushes the static header of the search page prior to even analyzing the query. After all, why should it wait, the header is the same for every search page! Then, while the client is parsing the header markup, the search query is dispatched to the search index, and the remainder of the document, which includes the search results, is delivered to the user once the results are ready. At this point, the dynamic parts of the header, such as the name of the logged-in user, are filled in via JavaScript.

# CHAPTER 3
# HTTP/2

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

SPDY was an experimental protocol, developed at Google and announced in mid-2009, whose 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.

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" on page 25.

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 offered a route to test and evaluate each proposal before its inclusion in the HTTP/2 standard. In the end, this process spanned three years and resulted in a over a dozen intermediate drafts:

- Mar, 2012: Call for proposals for HTTP/2
- Nov, 2012: First draft of HTTP/2 (based on SPDY)
- Aug, 2014: HTTP/2 draft-17 and HPACK draft-12 are published
- Aug, 2014: Working Group last call for HTTP/2
- Feb, 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*'

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 Chapter 1. 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.

Unfortunately, implementation simplicity also came at a cost of application performance: 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 3-1), which dictates how the HTTP messages are encapsulated and transferred between the client and server.

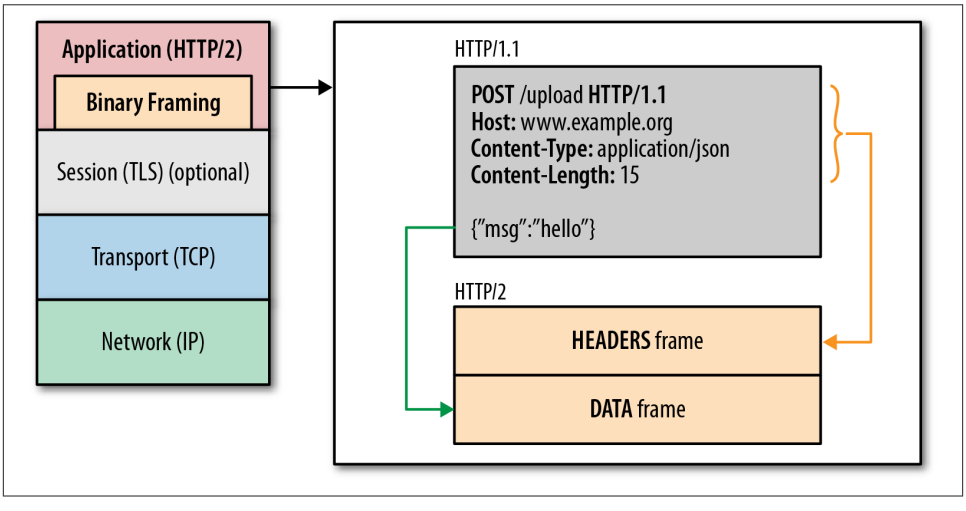



*Figure 3-1. HTTP/2 binary framing layer*

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 as efficient and 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 ???)—carrying HTTP/1.x and HTTP/2 data.

## Streams, Messages, and Frames

The introduction of the new binary framing mechanism changes how the data is exchanged (Figure 3-2) 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.

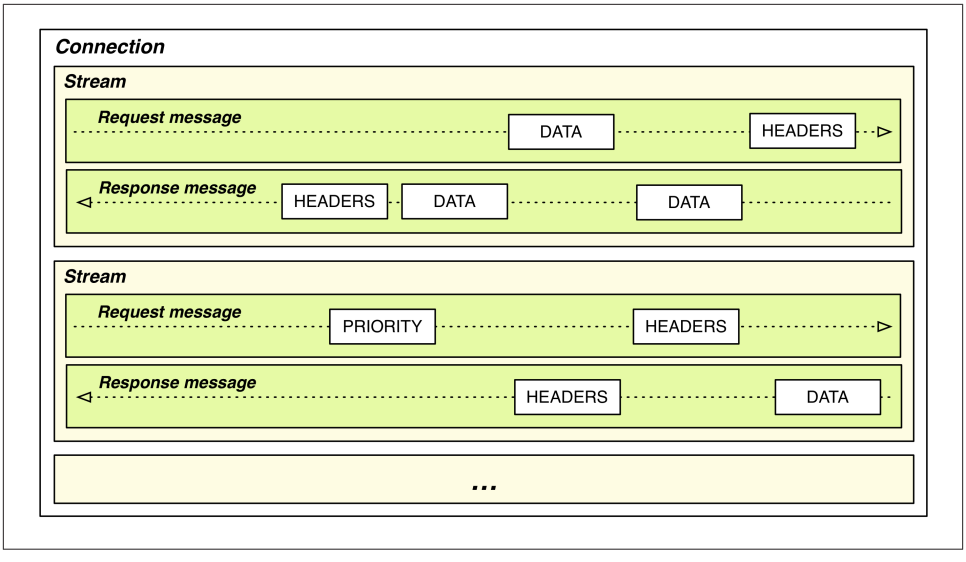



*Figure 3-2. HTTP/2 streams, messages, and frames*

- 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.

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 ???. 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 3-3), interleave them, and then reassemble them on the other end.

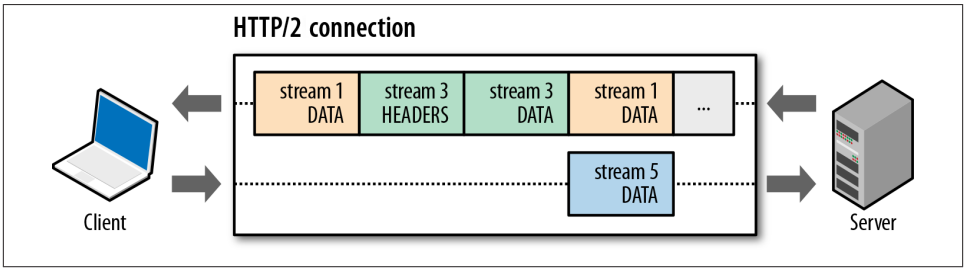



*Figure 3-3. HTTP/2 request and response multiplexing within a shared connection*

The snapshot in Figure 3-3 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" on page 66), 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
- Each stream may be given an explicit dependency on another stream

The combination of stream dependencies and weights allows the client to construct and communicate a "prioritization tree" (Figure 3-4) 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.

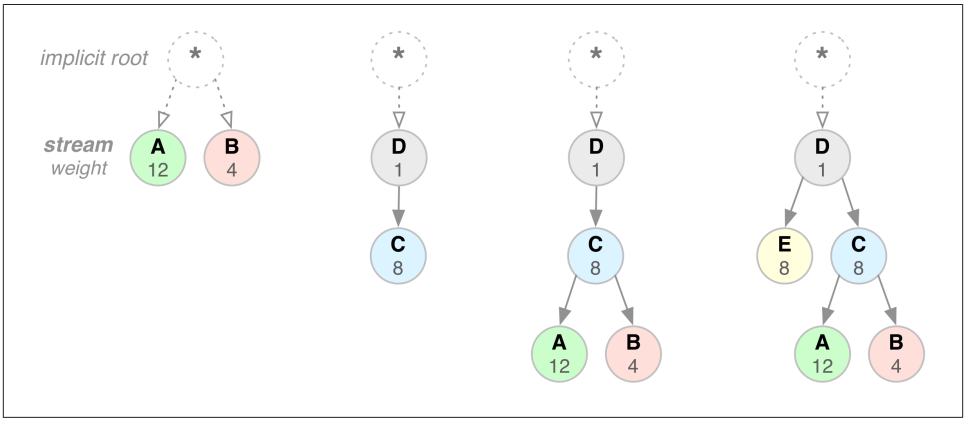



*Figure 3-4. HTTP/2 stream dependencies and weights*

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 3-4. 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.
2. D is dependent on the root stream; C is dependent on D. Thus, D should receive full 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.

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 counter-intuitive, 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" on page 16); 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, ??? tried to partially address this problem, but in practice it has failed to gain adoption.

HTTP/2 resolves these inefficiencies: request queuing and head-of-line blocking is eliminated because the browser can dispatch all requests the moment they are discovered, and the browser 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.

Reduced number of connections is a particularly important feature for improving performance of HTTPS deployments: this translates to fewer expensive TLS handshakes, better session re-use, 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 ???).
- Effects of bandwidth-delay product may limit connection throughput if TCP window scaling is disabled.
- When packet loss occurs, the TCP congestion window size is reduced (see ???), 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

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.

Do the above requirements remind you of TCP flow control? They should, as the problem is effectively identical—see ???. 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 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" on page 18) 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 3-5), without the client having to request each one explicitly!

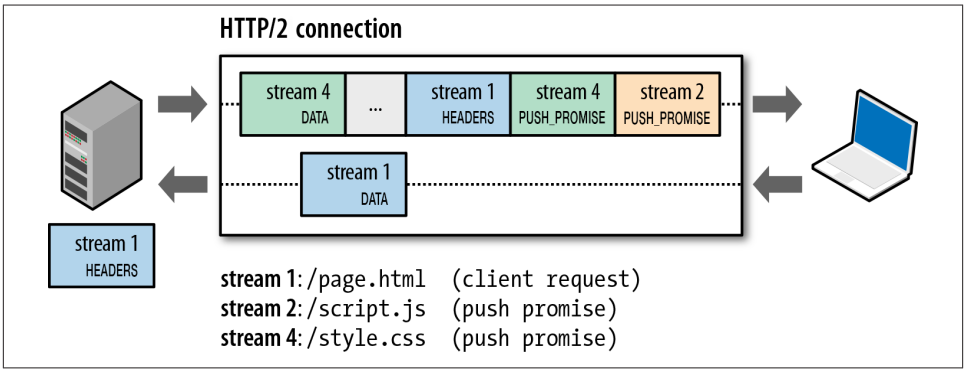



*Figure 3-5. Server initiates new streams (promises) for push resources*

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 ???), 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.

## PUSH_PROMISE 101

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 ???. 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 3-6) by transferring index values that can be used to efficiently look up and reconstruct the full header keys and values.

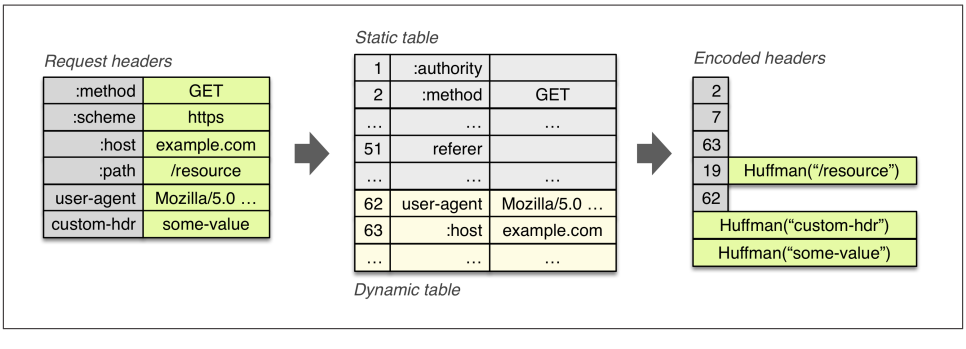



*Figure 3-6. HPACK: Header Compression for HTTP/2*

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.

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

Early versions of HTTP/2 and SPDY used zlib, with a custom dictionary, to compress all HTTP headers, which delivered 85%–88% reduction in the size of the transferred header data, and a 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 ???. 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; see ??? and ???. Further, as an additional 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 ❶
HTTP2-Settings: (SETTINGS payload) ❷

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

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

          (or)

HTTP/1.1 101 Switching Protocols ❹
Connection: Upgrade
Upgrade: h2c

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

❶ Initial HTTP/1.1 request with HTTP/2 upgrade header

❷ Base64 URL encoding of HTTP/2 SETTINGS payload

❸ Server declines upgrade, returns response via HTTP/1.1

❹ 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 worst case, the connection will fail, and the client will fall back to `Upgrade` workflow or switch to a TLS tunnel with ALPN negotiation.

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 3-7), which contains the length of the frame, its type, a bit field for flags, and a 31-bit stream identifier.

| Bit | +0..7 | +8..15 | +16..23 | +24..31 |
|---|---|---|---|---|
| 0 | Length | | | Type |
| 32 | Flags | | | |
| 40 | R | Stream Identifier | | |
| ... | *Frame Payload* | | | |

*Figure 3-7. 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.

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.

Given this knowledge of the shared HTTP/2 frame header, we can now write a simple parser that 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

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.

The good news is that the exact semantics of the preceding taxonomy of frames is mostly only 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 3-8) with all of the above.

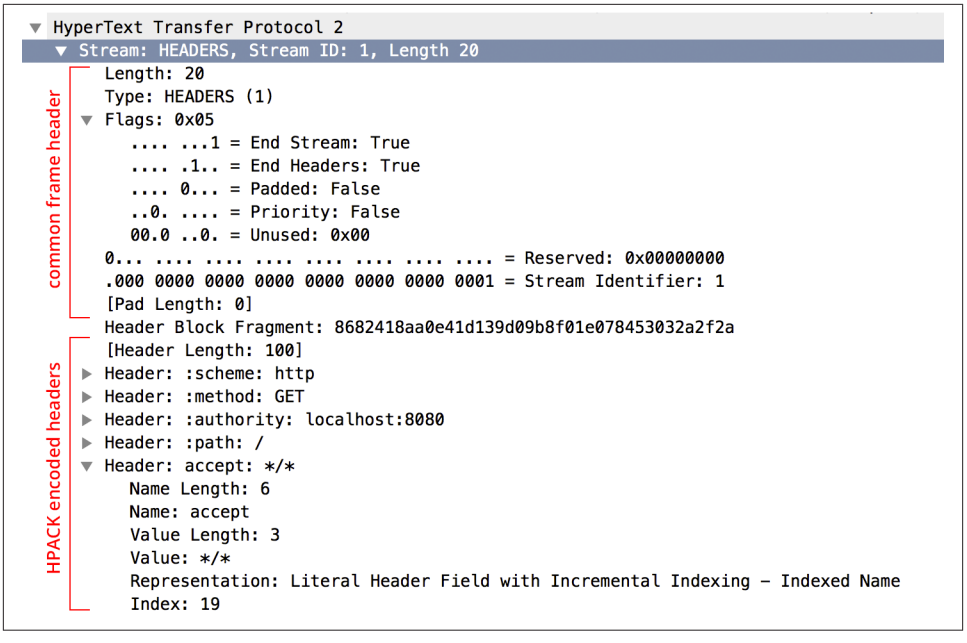



*Figure 3-8. Decoded HEADERS frame in Wireshark*

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 3-7.

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 3-8 is set to "1", we can infer that it is a client-initiated stream.

## Sending Application Data

Once a new stream is created, and the HTTP headers are sent, DATA frames (Figure 3-9) 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.

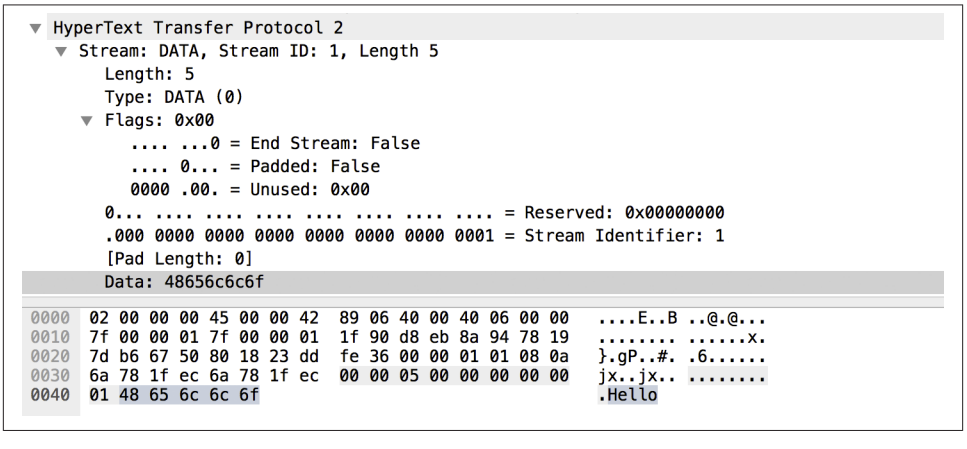



*Figure 3-9. DATA frame*

The "End Stream" flag is set to "false" in Figure 3-9, 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 3-10) we encountered earlier in "Request and Response Multiplexing" on page 41 and analyze the HTTP/2 exchange:

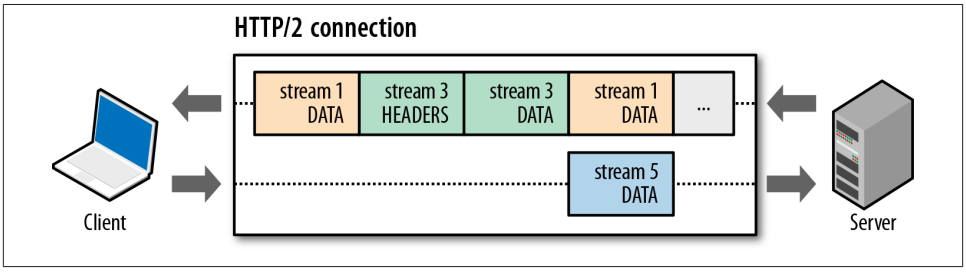



*Figure 3-10. HTTP/2 request and response multiplexing within a shared connection*

- 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!

CHAPTER 4

# Optimizing Application Delivery

High-performance browser networking relies on a host of networking technologies (Figure 4-1), and the overall performance of our applications is the sum total of each of their parts.

We cannot control the network weather between the client and server, nor the client hardware or the configuration of their device, but the rest is in our hands: TCP and TLS optimizations on the server, and dozens of application optimizations to account for the peculiarities of the different physical layers, versions of HTTP protocol in use, as well as general application best practices. Granted, getting it all right is not an easy task, but it is a rewarding one! Let's pull it all together.

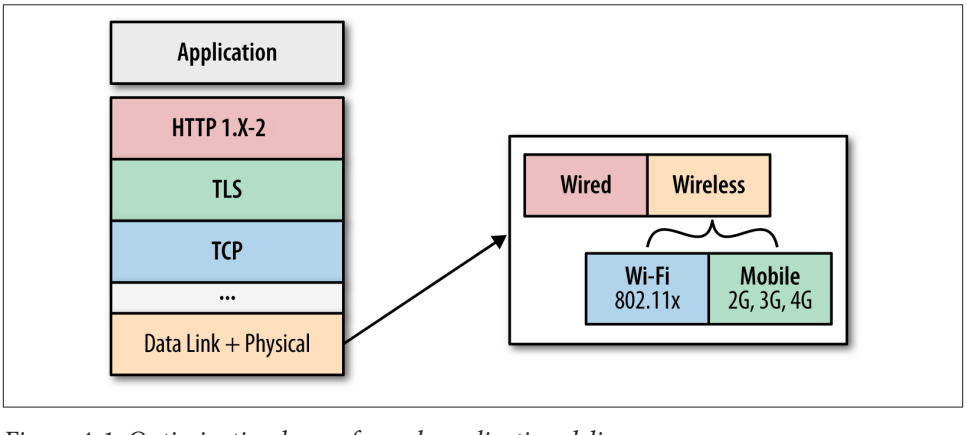



*Figure 4-1. Optimization layers for web application delivery*

## Optimizing Physical and Transport Layers

The physical properties of the communication channel set hard performance limits on every application: speed of light and distance between client and server dictate the propagation latency, and the choice of medium (wired vs. wireless) determines the processing, transmission, queuing, and other delays incurred by each data packet. In fact, the performance of most web applications is limited by latency, not bandwidth, and while bandwidth speeds will continue to increase, unfortunately the same can't be said for latency:

- ???
- ???
- "Latency as a Performance Bottleneck" on page 25

As a result, while we cannot make the bits travel any faster, it is crucial that we apply all the possible optimizations at the transport and application layers to eliminate unnecessary roundtrips, requests, and minimize the distance traveled by each packet—i.e., position the servers closer to the client.

Every application can benefit from optimizing for the unique properties of the physical layer in wireless networks, where latencies are high, and bandwidth is always at a premium. At the API layer, the differences between the wired and wireless networks are entirely transparent, but ignoring them is a recipe for poor performance. Simple optimizations in how and when we schedule resource downloads, beacons, and the rest can translate to significant impact on the experienced latency, battery life, and overall user experience of our applications:

- ???
- "Optimizing for Mobile Networks"

Moving up the stack from the physical layer, we must ensure that each and every server is configured to use the latest TCP and TLS best practices. Optimizing the underlying protocols ensures that each client can get the best performance—high throughput and low latency—when communicating with the server:

- ???
- ???

Finally, we arrive at the application layer. By all accounts and measures, HTTP is an incredibly successful protocol. After all, it is the common language between billions of clients and servers, enabling the modern Web. However, it is also an imperfect protocol, which means that we must take special care in how we architect our applications:

- We must work around the limitations of HTTP/1.x.
- We must leverage new performance capabilities of HTTP/2.
- We must be vigilant about applying the evergeen performance best practices.

The secret to a successful web performance strategy is simple: invest into monitoring and measurement tools to identify problems and regressions (see "Synthetic and Real-User Performance Measurement" on page 27), link business goals to performance metrics, and optimize from there - i.e. treat performance as a feature.

## Evergreen Performance Best Practices

Regardless of the type of network or the type or version of the networking protocols in use, all applications should always seek to eliminate or reduce unnecessary network latency and minimize the number of transferred bytes. These two simple rules are the foundation for all of the evergreen performance best practices:

*Reduce DNS lookups*
: Every hostname resolution requires a network roundtrip, imposing latency on the request and blocking the request while the lookup is in progress.

*Reuse TCP connections*
: Leverage connection keepalive whenever possible to eliminate the TCP handshake and slow-start latency overhead; see ???.

*Minimize number of HTTP redirects*
: HTTP redirects impose high latency overhead - e.g., a single redirect to a different origin can result in DNS, TCP, TLS, and request-response roundtrips that can add hundreds to thousands of milliseconds of delay. The optimal number of redirects is zero.

*Reduce roundtrip times*
: Locating servers closer to the user improves protocol performance by reducing roundtrip times (e.g. faster TCP and TLS handshakes), and improves the transfer throughput of static and dynamic content; see ???.

*Eliminate unnecessary resources*
: No request is faster than a request not made. Be vigilant about auditing and removing unnecessary resources.

By this point, all of these recommendations should require no explanation: latency is the bottleneck, and the fastest byte is a byte not sent. However, HTTP provides some

additional mechanisms, such as caching and compression, as well as its set of version-specific performance quirks:

*Cache resources on the client*
: Application resources should be cached to avoid re-requesting the same bytes each time the resources are required.

*Compress assets during transfer*
: Application resources should be transferred with the minimum number of bytes: always apply the best compression method for each transferred asset.

*Eliminate unnecessary request bytes*
: Reducing the transferred HTTP header data (e.g., HTTP cookies) can save entire roundtrips of network latency.

*Parallelize request and response processing*
: Request and response queuing latency, both on the client and server, often goes unnoticed, but contributes significant and unnecessary latency delays.

*Apply protocol-specific optimizations*
: HTTP/1.x offers limited parallelism, which requires that we bundle resources, split delivery across domains, and more. By contrast, HTTP/2 performs best when a single connection is used, and HTTP/1.x specific optimizations are removed.

Each of these warrants closer examination. Let's dive in.

## Cache Resources on the Client

The fastest network request is a request not made. Maintaining a cache of previously downloaded data allows the client to use a local copy of the resource, thereby eliminating the request. For resources delivered over HTTP, make sure the appropriate cache headers are in place:

- `Cache-Control` header can specify the cache lifetime (max-age) of the resource.
- `Last-Modified` and `ETag` headers provide validation mechanisms.

Whenever possible, you should specify an explicit cache lifetime for each resource, which allows the client to use a local copy, instead of re-requesting the same object all the time. Similarly, specify a validation mechanism to allow the client to check if the expired resource has been updated: if the resource has not changed, we can eliminate the data transfer.

Finally, note that you need to specify both the cache lifetime and the validation method! A common mistake is to provide only one of the two, which results in either redundant transfers of resources that have not changed (i.e., missing validation), or redundant

validation checks each time the resource is used (i.e., missing or unnecessarily short cache lifetime).

For hands-on advice on optimizing your caching strategy, see the "HTTP caching" section on Google's Web Fundamentals: *http://hpbn.co/wf-caching*.

### Web Caching on Smartphones: Ideal vs. Reality

Caching of HTTP resources has been one of the top performance optimizations ever since the very early versions of the HTTP protocol. However, while seemingly everyone is aware of its benefits, real-world studies continue to discover that it is nonetheless an often-omitted optimization! A recent joint study by AT&T Labs Research and University of Michigan reports:

> Our findings suggest that redundant transfers contribute 18% and 20% of the total HTTP traffic volume in the two datasets. Also they are responsible for 17% of the bytes, 7% of the radio energy consumption, 6% of the signaling load, and 9% of the radio resource utilization of all cellular data traffic in the second dataset. Most of such redundant transfers are caused by the smartphone web caching implementation that does not fully support or strictly follow the protocol specification, or by developers not fully utilizing the caching support provided by the libraries.
>
> — Web Caching on Smartphones
> *MobiSys 2012*

Is your application fetching unnecessary resources over and over again? As evidence shows, that's not a rhetorical question. Double-check your application and, even better, add some tests to catch any regressions in the future.

## Compress Transferred Data

Leveraging a local cache allows the client to avoid fetching duplicate content on each request. However, if and when the resource must be fetched, either because it has expired, it is new, or it cannot be cached, then it should be transferred with the minimum number of bytes. Always apply the best compression method for each asset.

The size of text-based assets, such as HTML, CSS, and JavaScript, can be reduced by 60%–80% on average when compressed with Gzip. Images, on the other hand, require more nuanced consideration:

- Images often carry a lot of metadata that can be stripped - e.g. EXIF.
- Images should be sized to their display width to minimize transferred bytes.

- Images can be compressed with different lossy and lossless formats.

Images account for over half of the transferred bytes of an average page, which makes them a high-value optimization target: the simple choice of an optimal image format can yield dramatically improved compression ratios; lossy compression methods can reduce transfer sizes by orders of magnitude; sizing the image to its display width will reduce both the transfer and memory footprints (see ???) on the client. Invest into tools and automation to optimize image delivery on your site.

For hands-on advice on reducing the transfer size of text, image, webfont, and other resources, see the "Optimizing Content Efficiency" section on Google's Web Fundamentals: *http://hpbn.co/wf-compression*.

## Eliminate Unnecessary Request Bytes

HTTP is a stateless protocol, which means that the server is not required to retain any information about the client between different requests. However, many applications require state for session management, personalization, analytics, and more. To enable this functionality, the HTTP State Management Mechanism (RFC 2965) extension allows any website to associate and update "cookie" metadata for its origin: the provided data is saved by the browser and is then automatically appended onto every request to the origin within the `Cookie` header.

The standard does not specify a maximum limit on the size of a cookie, but in practice most browsers enforce a 4 KB limit. However, the standard also allows the site to associate many cookies per origin. As a result, it is possible to associate tens to hundreds of kilobytes of arbitrary metadata, split across multiple cookies, for each origin!

Needless to say, this can have significant performance implications for your application. Associated cookie data is automatically sent by the browser on each request, which, in the worst case can add entire roundtrips of network latency by exceeding the initial TCP congestion window, regardless of whether HTTP/1.x or HTTP/2 is used:

- In HTTP/1.x, all HTTP headers, including cookies, are transferred uncompressed on each request.
- In HTTP/2, headers are compressed with HPACK, but at a minimum the cookie value is transferred on the first request, which will affect the performance of your initial page load.

Cookie size should be monitored judiciously: transfer the minimum amount of required data, such as a secure session token, and leverage a shared session cache on the server to look up other metadata. And even better, eliminate cookies entirely wherever possible

—chances are, you do not need client-specific metadata when requesting static assets, such as images, scripts, and stylesheets.

When using HTTP/1.x, a common best practice is to designate a dedicated "cookie-free" origin, which can be used to deliver responses that do not need client-specific optimization.

## Parallelize Request and Response Processing

To achieve the fastest response times within your application, all resource requests should be dispatched as soon as possible. However, another important point to consider is how these requests will be processed on the server. After all, if all of our requests are then serially queued by the server, then we are once again incurring unnecessary latency. Here's how to get the best performance:

- Re-use TCP connections by optimizing connection keepalive timeouts.
- Use multiple HTTP/1.1 connections where necessary for parallel downloads.
- Upgrade to HTTP/2 to enable multiplexing and best performance.
- Allocate sufficient server resources to process requests in parallel.

Without connection keepalive, a new TCP connection is required for each HTTP request, which incurs significant overhead due to the TCP handshake and slow-start. Make sure to identify and optimize your server and proxy connection timeouts to avoid closing the connection prematurely. With that in place, and to get the best performance, use HTTP/2 to allow the client and server to re-use the same connection for all requests. If HTTP/2 is not an option, use multiple TCP connections to achieve request parallelism with HTTP/1.x.

Identifying the sources of unnecessary client and server latency is both an art and science: examine the client resource waterfall (see "Analyzing the Resource Waterfall" on page 19), as well as your server logs. Common pitfalls often include the following:

- Blocking resources on the client forcing delayed resource fetches; see "DOM, CSSOM, and JavaScript" on page 16.
- Underprovisioned proxy and load balancer capacity, forcing delayed delivery of the requests (queuing latency) to the application servers.
- Underprovisioned servers, forcing slow execution and other processing delays.

### Optimizing Resource Loading in the Browser

The browser will automatically determine the optimal loading order for each resource in the document, and we can both assist and hinder the browser in this process:

- We can provide hints to assist the browser; see "Browser Optimization" on page 31.
- We can hinder by hiding resources from the browser.

Modern browsers are designed to scan the contents of HTML and CSS files as efficiently and as soon as possible. However, the document parser is also frequently blocked while waiting for a script or other blocking resources to download before it can proceed. During this time, the browser uses a "preload scanner," which speculatively looks ahead in the source for resource downloads that could be dispatched early to reduce overall latency.

Note that the use of the preload scanner is a speculative optimization, and it is used only when the document parser is blocked. However, in practice, it yields significant benefits: based on experimental data with Google Chrome, it offers a ~20% improvement in page loading times and rendering speeds!

Unfortunately, these optimizations do not apply for resources that are scheduled via JavaScript; the preload scanner cannot speculatively execute scripts. As a result, moving resource scheduling logic into scripts may offer the benefit of more granular control to the application, but in doing so, it will hide the resource from the preload scanner, a trade-off that warrants close examination.

## Optimizing for HTTP/1.x

The order in which we optimize HTTP/1.x deployments is important: configure servers to deliver the best possible TCP and TLS performance, and then carefully review and apply mobile and evergreen application best practices: measure, iterate.

With the evergreen optimizations in place, and with good performance instrumentation within the application, evaluate whether the application can benefit from applying HTTP/1.x specific optimizations (read, *protocol workarounds*):

*Leverage HTTP pipelining*
: If your application controls both the client and the server, then pipelining can help eliminate unnecessary network latency; see ???.

*Apply domain sharding*
: If your application performance is limited by the default six connections per origin limit, consider splitting resources across multiple origins; see ???.

*Bundle resources to reduce HTTP requests*
: Techniques such as concatenation and spriting can both help minimize the protocol overhead and deliver pipelining-like performance benefits; see ???.

*Inline small resources*
: Consider embedding small resources directly into the parent document to minimize the number of requests; see ???.

Pipelining has limited support, and each of the remaining optimizations comes with its set of benefits and trade-offs. In fact, it is often overlooked that each of these techniques can hurt performance when applied aggressively, or incorrectly; review ??? for an in-depth discussion.

HTTP/2 eliminates the need for all of the above HTTP/1.x workarounds, making our applications both simpler and more performant. Which is to say, the best optimization for HTTP/1.x is to deploy HTTP/2.

# Optimizing for HTTP/2

HTTP/2 enables more efficient use of network resources and reduced latency by enabling request and response multiplexing, header compression, prioritization, and more - see "Design and Technical Goals" on page 38. Getting the best performance out of HTTP/2, especially in light of the one-connection-per-origin model, requires a well-tuned server network stack. Review ??? and ??? for an in-depth discussion and optimization checklists.

Next up—surprise—apply the evergreen application best practices: send fewer bytes, eliminate requests, and adapt resource scheduling for wireless networks. Reducing the amount of data transferred and eliminating unnecessary network latency are the best optimizations for any application, web or native, regardless of the version or type of the application and transport protocols in use.

Finally, undo and unlearn the bad habits of domain sharding, concatenation, and image spriting. With HTTP/2 we are no longer constrained by the limited parallelism: requests are cheap, and both requests and responses can be multiplexed efficiently. These workarounds are no longer necessary and omitting them can improve performance.

## Eliminate Domain Sharding

HTTP/2 achieves the best performance by multiplexing requests over the same TCP connection, which enables effective request and response prioritization, flow control, and header compression. As a result, the optimal number of connections is exactly one and domain sharding is an anti-pattern.

HTTP/2 also provides a TLS connection-coalescing mechanism that allows the client to coalesce requests from different origins and dispatch them over the same connection when the following conditions are satisfied:

- The origins are covered by the same TLS certificate - e.g. a wildcard certificate, or a certificate with matching "Subject Alternative Names".
- The origins resolve to the same server IP address.

For example, if `example.com` provides a wildcard TLS certificate that is valid for all of its subdomains (i.e., `*.example.com`) and references an asset on `static.example.com` that resolves to the same server IP address as `example.com`, then the HTTP/2 client is allowed to reuse the same TCP connection to fetch resources from `example.com` and `static.example.com`.

An interesting side-effect of HTTP/2 connection coalescing is that it enables an HTTP/1.x friendly deployment model: some assets can be served from alternate origins, which enables higher parallelism for HTTP/1 clients, and if those same origins satisfy the above criteria then the HTTP/2 clients can coalesce requests and re-use the same connection. Alternatively, the application can be more hands-on and inspect the negotiated protocol and deliver alternate resources for each client: with sharded asset references for HTTP/1.x clients and with same-origin asset references for HTTP/2 clients.

Depending on the architecture of your application you may be able to rely on connection coalescing, you may need to serve alternate markup, or use both techniques as necessary to provide the optimal HTTP/1.x and HTTP/2 experience. Alternatively, you may consider focusing on optimizing HTTP/2 performance only; the client adoption is growing rapidly, and the extra complexity of optimizing for both protocols may be unnecessary.

Due to third-party dependencies it may not be possible to fetch all the resources via the same TCP connection - that's OK. Seek to minimize the number of origins regardless of the protocol and eliminate sharding when HTTP/2 is in use to get the best performance.

## Minimize Concatenation and Image Spriting

Bundling multiple assets into a single response was a critical optimization for HTTP/1.x where limited parallelism and high protocol overhead typically outweighed all other concerns - see ???. However, with HTTP/2, multiplexing is no longer an issue, and header compression dramatically reduces the metadata overhead of each HTTP request. As a result, we need to reconsider the use of concatenation and spriting in light of its new pros and cons:

- Bundled resources may result in unnecessary data transfers: the user might not need all the assets on a particular page, or at all.
- Bundled resources may result in expensive cache invalidations: a single updated byte in one component forces a full fetch of the entire bundle.
- Bundled resources may delay execution: many content-types cannot be processed and applied until the entire response is transferred.
- Bundled resources may require additional infrastructure at build or delivery time to generate the associated bundle.
- Bundled resources may provide better compression if the resources contain similar content.

In practice, while HTTP/1.x provides the mechanisms for granular cache management of each resource, the limited parallelism forced us to bundle resources together. The latency penalty of delayed fetches outweighed the costs of decreased effectiveness of caching, more frequent and more expensive invalidations, and delayed execution.

HTTP/2 removes this unfortunate trade-off by providing support for request and response multiplexing, which means that we can now optimize our applications by delivering more granular resources: each resource can have an optimized caching policy (expiry time and revalidation token) and be individually updated without invalidating other resources in the bundle. In short, HTTP/2 enables our applications to make better use of the HTTP cache.

That said, HTTP/2 does not eliminate the utility of concatenation and spriting entirely. A few additional considerations to keep in mind:

- Files that contain similar data may achieve better compression when bundled.
- Each resource request carries some overhead, both when reading from cache (I/O requests), and from the network (I/O requests, on-the-wire metadata, and server processing).

There is no single optimal strategy for all applications: delivering a single large bundle is unlikely to yield best results, and issuing hundreds of requests for small resources may not be the optimal strategy either. The right trade-off will depend on the type of content, frequency of updates, access patterns, and other criteria. To get the best results, gather measurement data for your own application and optimize accordingly.

## Eliminate Roundtrips with Server Push

Server push is a powerful new feature of HTTP/2 that enables the server to send multiple responses for a single client request. That said, recall that the use of resource inlining (e.g. embedding an image into an HTML document via a data URI) is, in fact, a form

of application-layer server push. As such, while this is not an entirely new capability for web developers, the use of HTTP/2 server push offers many performance benefits over inlining: pushed resources can be cached individually, reused across pages, canceled by the client, and more—see "Server Push" on page 48.

With HTTP/2 there is no longer a reason to inline resources just because they are small; we're no longer constrained by the lack of parallelism and request overhead is very low. As a result, server push acts as a latency optimization that removes a full request-response roundtrip between the client and server - e.g. if, after sending a particular response, we know that the client will always come back and request a specific subresource, we can eliminate the roundtrip by pushing the subresource to the client.

If the client does not support, or disables the use of server push, it will initiate the request for the same resource on its own - i.e. server push is a safe and transparent latency optimization.

Critical resources that block page construction and rendering (see "DOM, CSSOM, and JavaScript" on page 16) are prime candidates for the use of server push, as they are often known or can be specified upfront. Eliminating a full roundtrip from the critical path can yield savings of tens to hundreds of milliseconds, especially for users on mobile networks where latencies are often both high and highly variable.

- Server push, as its name indicates, is initiated by the server. However, the client can control how and where it is used by indicating to the server the maximum number of pushed streams that can be initiated in parallel by the server, as well as the amount of data that can be sent on each stream before it is acknowledged by the client. This allows the client to limit, or outright disable, the use of server push—e.g. if the user is on an expensive network and wants to minimize the number of transferred bytes, they may be willing to disable the latency optimization in favor of explicit control over what is fetched.
- Server push is subject to same-origin restrictions; the server initiating the push must be authoritative for the content and is not allowed to push arbitrary third-party content to the client. Consolidate your resources under the same origin (i.e. eliminate domain sharding) to enable more opportunities to leverage server push.
- Server push responses are processed in the same way as responses received in reply to a browser-initiated requests—i.e. they can be cached and reused across multiple pages and navigations! Leverage this to avoid having to duplicate the same content across different pages and navigations.

Note that even the most naive server push strategy that opts to push assets regardless of their caching policy is, in effect, equivalent to inlining: the resource is duplicated on

each page and transferred each time the parent resource is requested. However, even there, server push offers important performance benefits: the pushed response can be prioritized more effectively, it affords more control to the client, and it provides an upgrade path towards implementing much smarter strategies that leverage caching and other mechanisms that can eliminate redundant transfers. In short, if your application is using inlining, then you should consider replacing it with server push.

### Automating performance optimization via Server Push

How does the server determine which resources should be delivered via server push? The HTTP/2 standard does not specify any particular algorithm, and the server is free to implement own and custom strategies for each application.

For example, server-side application code can specify which resources should be pushed and when. This strategy requires explicit configuration but provides full control to the application developer. Alternatively, the server can learn the associated resources based on observed traffic patterns (e.g. by observing `Referrer` headers) and automatically initiate server push for related resources; use some mechanism to track or guess client's cache state and initiate push for missing resources; and so on.

Server push enables many new and previously not possible optimization opportunities. Check the documentation of your HTTP/2 server for how to enable, configure, and deploy the use of server push for your application.

## Test HTTP/2 Server Quality

A naive implementation of an HTTP/2 server, or proxy, may "speak" the protocol, but without well implemented support for features such as flow control and request prioritization it can easily yield less that optimal performance—e.g., saturate user's bandwidth by sending large low priority resources, such as images, while the browser is blocked from rendering the page until it receives higher priority resources, such as HTML, CSS, or JavaScript.

With HTTP/2 the client places a lot of trust on the server. To get the best performance, an HTTP/2 client has to be "optimistic": it annotates requests with priority information (see "Stream Prioritization" on page 43) and dispatches them to the server as soon as possible; it relies on the server to use the communicated dependencies and weights to optimize delivery of each response. A well-optimized HTTP server has always been important, but with HTTP/2 the server takes on additional and critical responsibilities that, previously, were out of scope.

Do your due diligence when testing and deploying your HTTP/2 infrastructure. Common benchmarks measuring server throughput and requests per second do not capture

these new requirements and may not be representative of the actual experience as seen by your users when loading your application.

### Optimizing response delivery with request prioritization

The purpose of request prioritization is to allow the client to express how it would prefer the server to deliver responses when there is limited capacity - e.g. the server may be ready to send multiple responses, but due to limited bandwidth it should prioritize sending some resources ahead of others.

- What if the server disregards all priority information?
- Should higher-priority streams always take precedence?
- Are there cases where different priority streams should be interleaved?

If the server disregards all priority information, then it runs the risk of causing unnecessary processing delays for the client—e.g., block the browser from rendering the page by sending images ahead of more critical CSS and JavaScript resources. However, delivering streams in a strict dependency order can also yield suboptimal performance as it may reintroduce the head-of-line blocking problem where a high priority but slow response may unnecessarily block delivery of other resources. As a result, a well-implemented server should give precedence to high priority streams, but it should also interleave lower priority streams if all higher priority streams are blocked.

## About the Author

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

## Colophon

The animal on the cover of *High Performance Browser Networking* is a Madagascar harrier (*Circus macrosceles*). The harrier is primarily found on the Comoro Islands and Madagascar, though due to various threats, including habitat loss and degradation, populations are declining. Recently found to be rarer than previously thought, this bird's broad distribution occurs at low densities with a total population estimated in the range of 250–500 mature individuals.

Associated with the wetlands of Madagascar, the harrier's favored hunting grounds are primarily vegetation-lined lakes, marshes, coastal wetlands, and rice paddies. The harrier hunts small invertebrates and insects, including small birds, snakes, lizards, rodents, and domestic chickens. Its appetite for domestic chickens (accounting for only 1% of the species' prey) is cause for persecution of the species by the local people.

During the dry season—late August and September—the harrier begins its mating season. By the start of the rainy season, incubation (~32–34 days) has passed and nestlings fledge at around 42–45 days. However, the harrier reproduction rates remain low, averaging at 0.9 young fledged per breeding attempt and a success rate of three-quarter of nests. This poor nesting success—owing partly to egg-hunting and nest destruction by local people—can also be attributed to regular and comprehensive burning of grasslands and marshes for the purposes of fresh grazing and land clearing, which often coincides with the species' breeding season. Populations continue to dwindle as interests conflict: the harrier requiring undisturbed and unaltered savannah, and increasing human land-use activities in many areas of Madagascar.

Several conservation actions proposed include performing further surveys to confirm the size of the total population; studying the population's dynamics; obtaining more accurate information regarding nesting success; reducing burning at key sites, especially during breeding season; and identifying and establishing protected areas of key nesting sites.

The cover image is from *Histoire Naturelle, Ornithologie, Bernard Direxit*. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.