O'REILLY®

Early Release

RAW & UNEDITED

# Getting Started with Web Performance

OPTIMIZING THE USER EXPERIENCE

Daniel Austin

# Getting Started with Web Performance

*Daniel Austin*

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

**Getting Started with Web Performance**

by Daniel Austin



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*.

**Editors:** Courtney Nash and Brian Anderson
**Production Editor:** FIX ME!
**Copyeditor:** FIX ME!
**Proofreader:** FIX ME!

**Indexer:** FIX ME!
**Cover Designer:** Karen Montgomery
**Interior Designer:** David Futato
**Illustrator:** Rebecca Demarest

December 2014: First Edition

**Revision History for the First Edition:**

2014-07-30: Early release revision 1

See *http://oreilly.com/catalog/errata.csp?isbn=9781491945063* for release details.



ISBN: 978-1-491-94506-3

[?]

# Table of Contents

# Preface

This book is a work in progress – new chapters will be added as they are written. We welcome feedback – if you spot any errors or would like to suggest improvements, please email *perfbook@danielaustin.org*.

## Overture: Who Should Read This Book (and Why)

If you want to help make the Web faster and smarter, read this book.

If you're involved in designing, creating, developing, or running a Web site, this book for you. This book is entirely focused on helping you improve your user's experience by making it faster.

This is a book about Web performance – about understanding how the Web works under the covers from a performance point of view, and why some sites on the Web are faster than others.

How to make the slower ones faster, and the fast ones faster still, is the most important goal.

The original title of this book was *End-to-End Web Performance* and while the title and content have evolved over time, we're going to cover the entire HTTP request/response cycle, examining it in detail to find out what really happens when you load a Web page, tweet your latest achievement or play an online game (and why it takes so long). Along the way I'll introduce a simple model for reasoning about end-to-end response times, and talk about how to use it to think about performance in a new way.

Using our simple model, we'll see how to identify performance problems easily, prioritize them, analyze them, and then optimize them. Perhaps most importantly, I want to help you to prevent performance problems *before* they occur, and that will be one of the big themes of this book.

Web Performance is a complex topic, and there's a lot to learn. I want to help readers understand the big picture on performance, and how it relates to *systems thinking* – another theme of the book.

We'll be looking at a lot of data on how long it takes Web sites to load in the course of this book, and I'll focus on learning how to gather and examine performance data in every chapter (after this one). Testing things for yourself, and gaining a hands-on, intuitive feel for the numbers, is the best way to become an amazing performance expert. I also want to inject a sense for how things work in the so-called 'real world', using real data from real users, rather than idealized performance numbers that are never experienced by ordinary users.

## Why You Should Read This Book

This book is written specifically for people working on Web experiences. Since performance is a *design-time activity* I want especially to reach the people who are involved in the design and development process. But a good Web experience –on a desktop, a mobile device or some other Web client – involves a lot more than just good design. Once the design is ready, then it's the developer's turn, and then after your Web experience goes live, it's the operational aspects of the system that drive the quality of the user experience, and I want to write my book so that it's useful to everyone involved. If you are into building cool stuff on the Web, and want to make it faster, please read this book.

Here's a partial list of roles performed by people involved in performance work that I'd like to reach with my book.

- Web Designers – who want to create faster Web experiences with performance 'baked in up front', that is, part of the initial design.
- Developers – who want to build faster services and applications, avoid common problems and write software that really helps users
- Systems engineers and operations staff - who face the difficult task of making sure the Web is fast, all the time, and quickly resolving problems when they come up.
- Performance Engineers and Analysts – who are involved in identifying ways to make the Web faster and resolve performance issues on behalf of the end user.
- Project and People Managers – who need to understand Web Performance as part of their area of responsibility and make wise decisions about performance as part of their overall strategy.

## What We'll Cover

This book is intended to get you started thinking about Web performance, and provide you with the knowledge and skills required to tackle performance problems and optimize your Web experiences. So I'll start from the beginning, and try to give you a real overview of the performance domain. I've tried to organize the book to provide a good mixture of both theory and practice, only introducing enough math to support our analysis and trying to focus on what's really happening.

One of the big challenges in writing this book has been to decide what to leave out, rather than what to include, and there's any number of topics that might have been included, but were not, in the name of staying focused. I haven't covered, for example, every possible edge case, and as a friend said to me in the course of writing this book, "There's an exception to everything on the Web". It's true, but here we'll try to cover the most likely scenarios, under the sorts of circumstance that ordinary users face in their lives.

Another challenge is introducing a lot of technical words and ideas, some of which may be unfamiliar to readers, in a few short pages. I've made a point of making sure that everything is explained as we go along, and not to make any assumptions. My goal is to make sure that you are as interested and excited about performance as I am, and that I'm communicating what are often complex ideas so that everyone has a shared understanding.

'Test it yourself' is another big theme and as we go through the book I've added several hands-on sections and suggested tests for you to try for yourself. I've found that people grasp and retain information far more easily if they are able to actually perform the work themselves. There's an element of *personal knowledge* involved in Web performance work, and engaging with real numbers and data is the best way to get a feel for performance problem solving. I hope you enjoy these exercises as much as I enjoyed putting them together.

I want to start off by explaining what Web performance is, in Chapter 1, and explain how performance fits into the bigger picture of the entire system, the systemic qualities, and the trade-offs involved from a high-level. Then we'll zoom in on performance and look at some of the factors involved in the End-to-End response time (E2E).

One important topic we'll cover in Chapter 1 is "How fast is fast enough?" This will introduce another important idea - that *users* are the most important component on the Web, and *fast enough* depends on them, not on clients, networks, or servers.

In chapters 2-7 we'll be talking specifically about the response time of the HTTP request/response cycle, and covering it in detail will lead us to the *Multiple Parallel Persistent Connections* (MPPC) model, which gives us a framework for understanding what happens under the covers when we surf the Web, starting with an overview in Chapter 2.

I'll go over each of the MPPC time segments (more on this later) in chapters 3-6, and describing what's actually happening in the client, network, and server(s), as well as how to identify problems and optimize the different contributions to the E2E. Along the way we'll find opportunities to test and measure various aspects of Web response time in the hands-on exercises, intended to give you a concrete sense of what's happening. Chapter 7 will tie it all together, bringing all the different aspects of our model into focus.

Can we use the MPPC model to predict Web response times? In general, we often can, up to a point. Understanding the E2E is more than the sum of the individual response times however, and we'll talk about the user side of things in more detail in Chapter 7 as well.

There are many tools in Web performance science, and we'll talk about them in chapter 8, and how to use them wisely. Some of these tools are better than others at specific kinds of tests, and understanding how to pick the right tool for the job is an important skill.

Designing an effective test methodology goes hand-in-hand with our understanding of the tools available, and we'll spend quite a bit of time on this often contentious aspect of performance work in chapter 9.

Strategies for making Web sites faster (beyond just fixing immediate problems) are the topic for chapter 10. I'll share some ideas on how not to have performance problems to begin with, and what not to do performance-wise.

HTML5 is the next generation of the Web's native language and it's significantly different from previous generations. The discussion here in chapter 11 will focus on some of the more complete and widely implemented aspects of this exciting new revision of HTML, including the relationship between HTML5 and Web applications on different platforms and devices.

Performance for mobile devices is a huge topic. My approach in this book has been two-fold: first, to consider mobile as a first-class platform throughout the book, taking its particular constraints and advantages into account equally on par with the desktop experience, and also to provide a specific chapter later in the book (chapter 12) devoted to mobile-related issues, and spending some time explaining things like 4G networking challenges in more depth. But mobile performance deserves a book all by itself, and this is not that book. Other forthcoming titles from O'Reilly will examine mobile performance in more depth.

In closing, I'm going to add a few thoughts of my own on Web performance challenges in the future in chapter 13, and what sorts of new technologies might play a role in achieving our end goal: making the Web smarter and faster for everyone.

## What You Need to Know to Get Started

I often teach a one-day boot camp session on Web performance, and when I'm asked what you need to know, I always say 'a laptop and an open mind'. This book is a much longer and more detailed explanation of the material in that one-day session, but my answer still applies here. It's helpful to have some background in basic statistics, HTML, and JavaScript, but everything you need to know will be explained as we go along, and I don't expect readers to be experts on any of the topics discussed here. (At least not until you've read the book of course!)

## How to Read This Book

This book is written as a single narrative or story, and later chapters will build on earlier ones, especially in the beginning chapters. The glossary in the back will help if you want to skip around in the book and learn about a specific topic, and I've tried to keep concepts and their definitions together so that you can quickly refer back to an earlier point or figure easily.

I personally find that diagrams and pictures are amazingly helpful in learning new material, so I've tried to include illustrations for every important concept.

I also encourage you to follow along with the hands-on exercises in the book, and see for yourself how these ideas work in practice.

## Software Tools

In order to be able to run the exercises you'll need access to a machine connected to the Web. You'll need to have the Firefox and Chrome browsers installed in some cases, and it's always a good idea to have multiple browsers on hand. In some cases you'll need to use some UNIX (or UNIX-like) command-line tools. For Windows users (like myself) I prefer the Cygwin package of POSIX-compliant tools, and in cases where the Windows tools have differences from standard Unix commands I'll call them out. For Mac users, the underlying Unix operating system provides all of the command-line tools discussed here and the same is of course true for users of other Unix-like systems.

One tool you'll use heavily in this book is cURL, a command-line HTTP client. You'll want to have cURL installed on your system, and also on your mobie device, where versions are available for major mobile platforms. You can download cURL here:

*http://curl.haxx.se/*

It'll be helpful to have a tool for crunching data, and I'll use the powerful R language throughout this book for examples. However there's nothing here that couldn't equally be done with Microsoft Excel. You can download The R command line tools here:

*http://www.r-project.org/*

and a useful GUI called RStudio here:

*http://www.rstudio.com/*

Both of these tools are freely available.

We'll use a number of online and desktop tools to measure performance, and I'll spend some time explaining them as we go. In cases where we're talking about commercial software, I'll be sure to note how free services can (in many cases) fill the gaps.

### About the Companion Site for This Book

In the course of the book we'll be creating some of our own tools, from simple scripts to Web performance testing nodes. All of this will be available online at the book's companion site on github.

I'll try to credit everyone when I'm using their tools, and make all of the data used here public for everyone to view and analyze for themselves.

### Notes on Terminology and References

There are a lot of technical terms used in this book, and I'll try to make a point of defining each of them when they're first used. I'll also provide a glossary at the back of the book, and an index to make it easier to find terms and their definitions, as well as references to Web standards.

## Themes of the Work

Throughout the book, several key themes will appear over and over again, and deserve to be called out specifically.

- *Systems thinking*
- *Performance is a design-time activity*
- *Test it for yourself*
- *Personal knowledge*
- *Performance is fundamentally about respect*

With that in mind, let's get started!

## 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.

## Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

## 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://www.oreilly.com/catalog/<catalog page>*.

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*

## Acknowledgments

Every book has a cast of thousands, and this one is no exception. Among the people I'd like to especially thank are Doug Crockford, Courtney Nash, Brian Anderson, Uli Romahn, Jem Day, Petter Graff, and Vladimir Bacvanski. They've all encouraged this work, and equally important, pointed it out when I was wrong - the best kinds of friends to have. ANd especially Ambrin Ali, who made it all readable.

CHAPTER 1

# What Is Web Performance?

This chapter of the book is intended to provide a big-picture overview of Web performance. I want to try to explain what Web performance is, where it fits into the larger scheme of things, and why it's important. A large part of my own view is informed by thinking about systems at scale, often very large or high-volume systems, and so I'll be making the case that performance is just one of many systemic qualities that need to be considered. We'll discuss the role of the user in Web performance, and what *fast* (and *slow*) really mean in the next section. Then we'll take a whirlwind tour through many of the different factors involved in the end-to-end performance of the Web experience.

## Scope

"Performance" means different things to different audiences, or even to the same audience based on the context. We use it to describe everything from Gatorade and medicines to corporate profits. Even in an IT context, we can be talking about performance in any number of different ways, from spinning disks in Winchester drives to the number of queries per second that can be completed by a database system or Website.

For our purposes in this book, we'll be talking about Web performance exclusively, rather than databases or disks. This means that first we're talking about using HTML, the HTTP protocol, and the URI addressing scheme, which define the Web. All of the examples and discussions in this book will use HTTP, with a couple of notable exceptions, when we talk about HTML5 WebSockets and Web RTC. That doesn't mean, however that we're limited to talking about the typical desktop browser scenarios – much of the information in this book applies equally well to any applications that use HTTP to transport information – commonly called *clients* or *User Agents*.

This includes smartphone and tablet apps (which commonly use a native interface to make HTTP requests and render the response), Web Services APIs that make *head-*

*less* requests and act on responses from multiple sources, and other modern devices that use HTTP such as POS systems, TVs and game consoles.

Beyond just looking at HTTP requests and response times, of course, we'll be talking about all of the other factors that affect Web performance (and there are a lot of them).

### Performance is Response Time

Throughout this book I'll use consistent terminology (and units) about performance measurements. For our purposes, *performance is response time*. Sometimes you'll hear words like 'latency' used for performance measurements. Latency, when used here, is a technical term for TCP packet delay. Packets traveling on the network experience latencies. Users, out there on the Web with a mouse or mobile phone in their hands, experience response times – how long it takes to do what they want to do. Network latency may be some part of the overall response time the user experiences, but it's not the entire story if we want to understand the end-to-end performance of the system, in this case the Web.

Since it's a big theme, I want to stress once again, *performance is response time*.

## An Art and a Science

### Doing Science Experiments on the Web

Web Performance is both and Art and a Science, in the sense that there is both something to be done (Art) and something to know (Science). This may come as a surprise to many readers of this book, who might have thought initially that the subject was a branch of Engineering!

The reason I'm insisting that this is science instead is that the nature of the activities involved are scientific. Basically we're going to perform an experiment on the Web and analyze the results. Many times, in real world contexts, we'll be gathering and analyzing the results of a large number of experiments, and we'll need some statistical background to properly understand what the experimental results are telling us. We'll then use the results of our analysis to identify problems and optimizations, make inferences that solve our problems quickly, and then we'll run the next experiments to confirm.

It would be nice if the results of our experiments on the Web were easily predicted and that our testing was just a matter of confirming previous results. Not so on the Web, where the underlying technologies and the industry standards are continuou ly in a transitional state, and new capabilities and features (and new and unexpected ways of using existing ones) are being added all the time. This is what makes Web performance a science – we are not just confirming existing theories or optimizing existing systems, but dealing with new technologies and new methods continuously. The Web is, as the

saying goes, under construction, and finding new ways to make it faster is surely a big part of that.

### Beyond Analysis

When we test performance, and share our results, in books, blogs, and other media, we can provide direct feedback to the community about what the problems and pain points are. This can contribute directly to changes, for example, in how browsers work, or how standards are written. Testing and measuring Web performance (and publishing the information!) is a way of participating in making the Web better for everyone.

This is also part of what I mean when I say that Web Performance is an Art. We have to be very inventive and creative to keep up with both the users and the developers and the browser manufacturers as well. Coming up with new methods and tools for measuring performance is an important part of this, but even more important are the techniques we've learned about how to write our Web sites (and applications) to be faster. YouTube is a good example of this. They are great at analyzing their user's data and identifying the problems associated with streaming video on the Web.

## Systems Thinking: Where Does Performance Fit In?

One of the big themes of this book is *systems thinking*. If you look in the dictionary, or on Wikipedia, you'll find several different definitions of what a system is. Each of them is correct in its own way, and I'll adapt one of them for this book. We'll define a *system* as *a dynamic and complex whole, made up of multiple components, interacting as a structured functional unit*.

We're mostly concerned with computer systems, and in particular the Web (or some specific part of it, such as a Web page) and its related technologies. Such systems invariably have three primary types of components: *hardware*, *software*, and *peopleware*. All of these components are necessary to make the system as a whole work properly. Each of them needs to be taken into account when the system is being designed, and can only be ignored at peril.

There's one more component to our notion of system that often goes unmentioned, but is really quite important: the standards that describe how the other three components work together. The standards for HTTP, HTML, and URIs are the most important for the Web, but the standards that describe how USB devices work or how audio files are encoded are equally important for the system. This underlies a big idea in systems thinking: the rules (standards) around the way the parts work together are a crucial part of the system.

## Systemic Qualities and the Big Picture on Performance

So where does Web Performance fit into all of this big-picture thinking? Performance is a *systemic quality*. These are sometimes called 'ilities' or even worse, 'non-functional requirements'. Let's dig into the idea of systemic qualities. These are the properties of the system itself, rather than any particular part of it (though bottlenecks can certainly occur at specific places). One good way to think about the systemic qualities is as 'anything you can say about a black box – from the outside'.

If we think about the users of the system, we can see that there are essentially four groups of stakeholders in every system. They look at the system from different viewpoints and play different roles, and so the properties of the system that they care about are often very different as well. This gives us a way to organize the systemic qualities around each group of system stakeholders in a reasonable way. The four groups are end users, developers, operators, and planners, and they each have a group of systemic qualities associated with them, as shown in Figure 1-1 below.

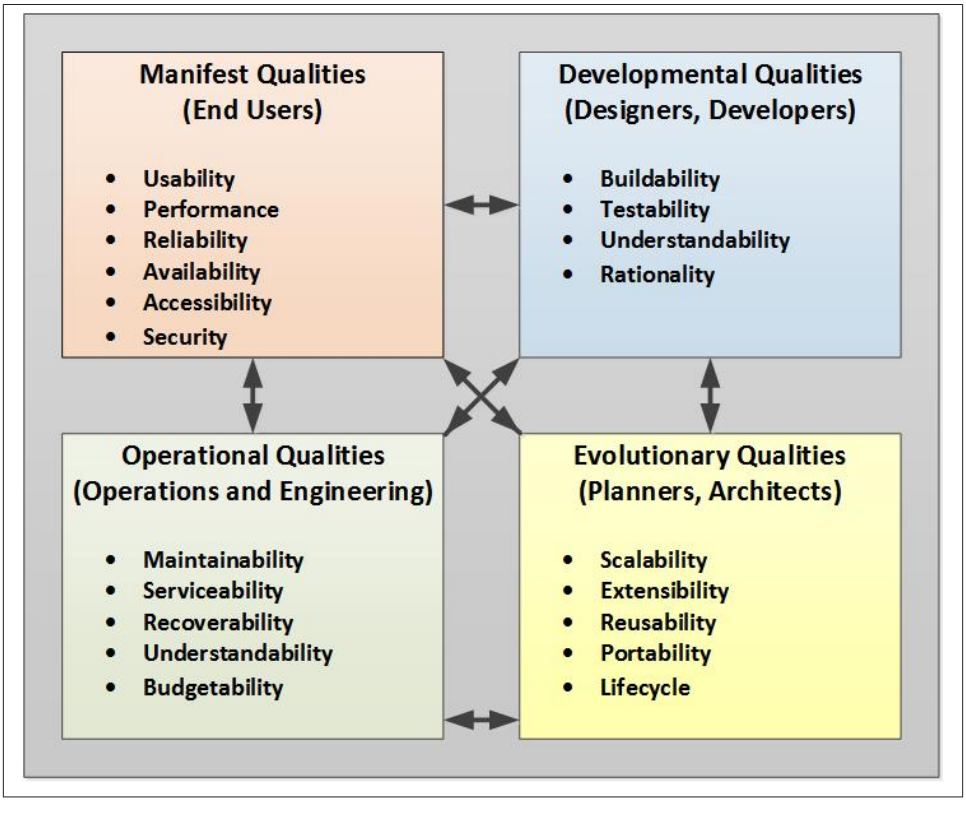


*Figure 1-1. The systemic qualities*

## Manifest Qualities

The manifest qualities are those that are important to the end user at runtime. These are the most evident qualities of the system in many ways, and certainly the ones we are most concerned with, in this book.

- *Usability* is the ease with which users can accomplish their goals
- *Performance* is the response time of the system, and reflects how much time users must wait for actions to complete
- *Reliability* measures how often the system fails
- *Availability* measures uptime vs. downtime
- *Accessibility* measures the system's ability to serve users regardless of location, device, network or physical condition (including I18N and L10N)
- *Security/Privacy* indicates the level of trust the user has in the integrity of the system

I'm not going to go into a lot of detail around the other groups of systemic qualities - I'll talk more about them as they come up in the book. We're primarily focused around performance and the manifest systemic qualities.

## Developmental Qualities

The next group of stakeholders in the system is the development staff – all of the people involved in creating software and hardware involved in the system. In addition to the other systemic qualities, they have their own concerns:

- *Buildability*
- *Budgetability/Manageability*
- *Understandability*
- *Balance*

## Operational Qualities

The operators of the system, the people who manage the networks, the Web servers, the software updates, and any other tasks involved in keeping the system going, are primarily concerned with their own aspects of the system:

- *Maintainability*
- *Serviceability*
- *Recoverability*
- *Budgetability*

- *Manageability*

## Evolutionary Qualities

Every system goes through some lifecycle – it is conceived, developed, operated, updated, and eventually, ended. How the lifecycle of the system is planned and managed is determined by the evolutionary systemic qualities, which are of primary concern to another group of stakeholders in the system: the planners. In every case, Web systems have planners who are tasked with managing the system lifcycle. These stakeholders care especially about how the system will develop, and how successful it will be. We can identify the evolutionary systemic qualities as those concerned with how the system changes over time:

- *Scalability*
- *Maintainability*
- *Extensibility*
- *Reusability*
- *Portability*

When we look at a system in this way, we can easily see that no specific set of qualities is more important than any other, and the same is true for each group of system stakeholders. Rather, every system has to meet very high standards in each of these properties in order to work well at all. Likewise, each group of stakeholders is concerned with other systemic qualities as well as their primary focus.

Another important point about the systemic qualities is that they are not really independent. They are all interrelated and can't be described as independent variables in some grand architectural 'Theory of Everything".

# Handling Performance Tradeoffs

When you're designing a system, you have to consider all of the systemic qualities, and how they relate to each other – and which ones are the most important for this particular system. Which systemic qualities have a higher priority attached to them? In a way, this is the fundamental question of architecture, and it brings us to the problems associated with making tradeoffs between among the systemic qualities.

## How to Manage Tradeoffs in Design

In the design of every system, some tradeoffs are necessary. These compromises are due to any number of factors, and amount to reconciling the weight of the interests of the various groups of stakeholders in the system. Since computer systems are generally

designed on behalf of the end user, we often put the highest priority on the manifest qualities. But many of the most common tradeoffs we make in our daily design activities center around how to manage them. Here are some of the most common cases we see in the world of Web performance.

### Security v. Performance

This is one of the most common tradeoffs we're called on to make, and it's almost invariably, and rightly, called on the side of security. This is not a hard decision to make most of the time, but it should be made in an 'eyes-wide-open' way, with a solid quantitative understanding of the performance costs involved.

A good example of this is when you are faced with deciding which parts of your site need to be secured with TLS connections. Web connections are faster without TLS than with it, and it's usually fairly easy to test the real response time difference between the two cases. This gives you a quantitative sense of what's involved.

### Budgetability v. Performance

Another common tradeoff is between the response times we desire for our end users and the costs associated with achieving it. This often comes up when big companies are deciding where to put their data centers, or when smaller companies are deciding how many servers to put into production. Some performance optimizations may simply be too costly to implement, just as some optimizations may make the code less portable. In these cases the decision often turns on how much performance the organization is willing to afford.

### Market Velocity v. Performance

This is a common and often costly trade-off to make, which can lead to poor quality products being rushed into the hands of users without testing their performance. This is a first-class mistake or performance antipattern. Never launch a product without testing its performance in the real world!

We've spent a fair amount of time describing the ideas of systems, systemic qualities, and the role of performance. This gives us a mental model or framework to guide us and give us perspective on what we are about. Understanding where performance fits into the big picture is important because it places things in perspective and shows us that while we, as performance professionals, are focused on our specific mission of making the Web faster, we are just one set of actors in a larger system, and not necessarily always the most important. Performance, alas, isn't everything.

# How Fast is Fast Enough?

Now that we can see how performance fits into the overall scheme of things, and have some idea of the kinds of trade-offs involved, let's ask a different question: what are the upper and lower limits of response time? Can we ever say that a site or service is *fast enough* and we can move on to something else? The answer is definitely yes.

A good analogy here is with television. One rarely hears anyone complaining that their TV show is playing slowly. Television generally provides a high-fidelity, high-bandwidth platform, at a rate of 30 frames per second or more. If you have a cable Internet connection as well as your TV service, you won't achieve anything like the same performance from your Web surfing as you do from your television, even though they come into your home over the same physical connection.

So if TV is *fast enough*, what about the Web? What makes the Web slow, or more precisely, a user's *perception* of the Web to be slow, compared to TV (or even the plain old-fashioned telephone service, where no one ever complained about network latency, even for the longest of long-distance calls).

The answer, it turns out, is you, and your own personal bandwidth.

## Role of the Human in Performance

The psychology of how users perceive Web performance in their everyday lives is a key topic of this book, and I'll touch on various aspects of this fascinating topic in later chapters. Users, unsurprisingly, have little interest in how many milliseconds it takes a site to load and a lot of interest in not having to wait for tasks to complete. This is the key point, *not having to wait*.

One popular college textbook, "Engineering Psychology and Human Performance", talks about *human performance* in terms very similar to the ones we use to describe our other performance measures: response time, accuracy, and attentional demand. In our case, it takes time to view, comprehend, and respond to the results of a Web request. This *user think time* is an important part of the E2E, and this leads us to an important idea that I call the *Layer 8 Hypothesis* (for reasons that will become clear when we talk about the OSI stack):

**"Humans are nodes on the Web and play a role in the overall E2E response time."**

Humans are, in fact, the most important part of the Web (connected, usually, via an optical link), and we have our own response times, the time it takes us to absorb the information coming in, accurately, within some reasonable level of attentional demand, and to respond appropriately. If our attentional demand goes to zero, we're waiting. If the information comes in too fast, it gets discarded; too slowly, and we're bored. Is it even possible for a Web site to be too fast?

## What do Fast and Slow Really Mean?

I'm going to state upfront the answer to my earlier question about how fast is too fast, and that will lead us to a better understanding of what fast and slow mean to users.

*A Website is fast enough when the information is being presented to the user as fast or faster than they can accurately respond.*

Think about it; this means that the user's own response time, not that of the Web server or network or client, is now the bottleneck in the system. If the information is coming in faster than we can respond, we say a site is fast, and if that's not the case, we say it's slow. This is the goal of your performance optimization efforts, make sure that all the work we do doesn't leave the user waiting.

This idea also explains why there are no complaints about television being slow; increasing the frame rate won't improve your perception of the show, because the information is already coming in at close to the rate that you can comprehend it - think about fast-forwarding a video - and so you don't perceive it as being slow. It's all about the wait, and the user's perception of the system's response time.

# Dimensions of Web Performance

There are a very large number of factors that affect end-user response times, and together they make up the user experience. Many of these factors are ourside the control of the user and sometimes of the developers and operators as well. This section provides an overview of some of the most important dimensions of performance from the end user's viewpoint.

## The Stochastic Web

There are a lot of different variables that go into the overall response time for HTTP requests and responses. Not all of them are known or understood, and in addition, the underlying Internet systems that use TCP/IP are inherently stochastic in nature. That means they are non-deterministic, and require statistical analysis to make sense of them. Stochastic systems like the Internet are subject to unpredictable transient behavior and failure modes that are not necessarily reproducible. This shows up in our measurements in the form of unexplainable outliers or failures that aren't correlated with any specific changes in the system.

### The "Contraptionary" Nature of the Web

Edward Tufte, famous for his graphic design sensibility, once called the Web *contraptionary*, a word I've always admired. What he meant is that, in addition to being stochastic at the network level, the Web is also, because of its perpetually-under-construction state and heterogenous network environment, always subject to unexpec-

ted failures and delays. It doesn't always work, or work as intended, for every user in every time and place, and you can't predict every circumstance.

## Factors Affecting Web Response Times

Even recognizing that there are many variables involved and that they are related in complicated ways we don't always understand, we can pretty easily list the most important factors that affect Web response times. Once again, it's difficult to say which of these factors is the most important for any individual Web application without testing it and analyzing the results. The Multiple Parallel Persistent Connections (MPPC) model of Web Performance described in this book will give you an easy way to determine which one of these factors is the 'long pole' in your performance tent.

### Geography

The geographic location of the user and the Web server is one of the first things usually mentioned when we talk about Web performance. It's clearly important, but it's not nearly as simple as distance measured in miles. In addition, the geographical location (what you see on a map of the earth's surface) and the user's network location, is also complex. In cities for instance, response times tend to be lower than in remote areas, regardless of physical distance. This is due to the higher level of network access points and better infrastructure, both hardware and software. Geography of course determines where the cities are – you can see this if you lot response times for, say, India, where you can see that response times are lower near the rivers – because that's where the cities are located.

### Network Location

The user's location on the network, related to their geographical location, is determined by their Internet services provider. It's usually closely related to the user's physical location, though not necessarily. The network location is determined by the length of the data paths between the user and the server responding to the user's request, in units of time. I say paths, plural because it's not the case, with TCP/IP networks, that any particular bit of data will follow any particular path at any time. TCP/IP was designed for flexibility and resilience. Data may find its way from server to client over multiple paths. However, the system is fairly stable under reasonable loads and most users see similar response times in a particular network location.

### Bandwidth and Throughput

Bandwidth is a significant factor in Web performance, and is also determined by the user's service provider. Later in the book we'll see that bandwidth is a limiting factor in response times only up to a certain point, and that increasing bandwidth reaches a point of diminishing returns thereafter.

We should also note that it's the effective bandwidth at the time we're actually performing a measurement that matters, and not the number that is cited by the ISP when the user establishes an account. This number is called the Committed Data Rate (CDR), and it's only a composite number based on capacity, average usage, and data size and type. We can think of the CDR as a sort of upper limit on the range of the user's bandwidth, but it's not a hard limit and can usually be exceeded for short periods of time. This often works out well for Web traffic, which is often 'bursty' in nature (meaning it can go through large changes in bandwidth utilization very quickly in a transient way).

The user's effective bandwidth can also be changed by other factors, such as network congestion in their network neighborhood. The user's home network setup can have a huge effect as well. If the user is on a laptop for example, and moves to another room in their home, or to another network context entirely, such as a coffee shop, their effective bandwidth will also change, and the user's experience will show the effects.

Throughout this book I'll try to use words precisely, and so I'm avoiding using the term *throughput* in this discussion. Bandwidth has a precise definition, whereas throughput is subject to interpretation. For our purposes, we'll stick with bandwidth, recognizing that this is basically synonymous with thoughput in any case where it matters, while avoiding unnecessary complications.

## Connection Type

How the user connects to the network plays a large role in their Web experience. We're certainly aware, as users, that there's a huge difference in how Web applications perform on the desktop versus on a mobile device or wireless network. These are all common means of connecting to the Internet, and they each have their own characteristics when it comes to Web performance. Here we'll focus on three basic transport types: directly connection via Ethernet (LAN), wireless connections via 802.11 (WiFi), and cellular connections, usually using 3G or 4G/LTE technologies. In each case, the underlying network protocols are TCP/IP, but the connection type adds an additional layer of complexity (and response time increase) on top of the underlying networks.

### Direct Connections (LAN)

This is the desktop or laptop experience that most users are familiar with, and represents the baseline against which the other connection types are compared for performance purposes.LAN connections are much faster than either WiFi or cellular. Usually these connections are made from the user device, frequently a desktop computer, via Ethernet cables to the home's local routing device. The local router is often a cable, or an aDSL connection, with direct fiber connections becoming more common in urban areas. Residential services are usually sold based on their CDR on a subscription basis.

At work, many people are connected via the company's high-grade commercial ISP services, which often have much better performance characteristics than home services, in order to support the associated business activities.

However you're connected to the Internet, a direct connection via cable or other method is often the best user experience, with higher bandwidth and lower latency than either wireless or cellular. Being untethered is great, but it comes at some costs in terms of response time.

### 802.11 Wireless (WiFi)

Wireless connections, often called 'WiFi', are based on the IEEE 802.11 family of standards. WiFi first appeared in 1997, but it wasn't until the 802.11b-1999 version came out that it began to be widely adopted by consumers. More updates and versions have followed, and today we have 802.11ac & 802.11ad devices being sold in stores, designed for different frequencies and ever-higher bandwidth (and reduced connection/wake up times). Today's Wifi is several times better than it was only 5 years ago, and continues to improve.

All of these different flavors of WiFi are similar in their performance characteristics after accounting for bandwidth. The one exception to this is for the 802.11ac & ad versions, which use the 5 MHz frequency band and may be subject to less interference.

WiFi connections suffer from limitations around both latency and bandwidth. A recent report from the US Department of Commerce showed that WiFi connections may see 30% or more bandwidth reduction compared to directly connected devices, even when connecting to the same access point.

### Cellular Networks

Cell phones are ubiquitous, far outnumbering landlines and desktop computers of all kinds, and their use is growing seemingly exponentially. The technologies used by cellular networks are specific to the provider and region, and have significant performance effects. Cellular connections are typically much slower than either direct or wireless connections, and have other performance issues including extended connection times. We'll cover cellular networking effects later.

## Hands-on: Check Your Bandwidth

One of the important themes of this book is personal knowledge – the knowledge gained by doing something for yourself and becoming familiar with the tools, methods and ideas you use to gather data to solve performance problems and optimize the user experience. All of science contains an element of personal knowledge, despite the efforts made at 'objectivity'. Throughout this book I'm going to add hands-on exercise sections that will give you opportunity to measure Web performance in a number of different

ways and get some exposure to real-world data. In some cases we'll require some mathematics, especially statistics. I'll explain as we go along.

## Task

In this first chands-on exercise, we're going to check our effective bandwidth right now, with as many different devices and/or connection types as we have available. I'll demonstrate using my desktop (LAN), and my phone using both Wifi and LTE networks, and then we'll look at the results.

### Using the SpeedTest Tools

We want to try to conduct our experiments on the Web keeping all of the factors listed above the same, while only changing a single variable. This is the right approach for apples-to-apples comparisons, but not for qualitatively different experiences. In our case, we'll keep the geographic location the same for all of our tests, and to some extent we'll use the same measurement tool.

For the LAN tests, we'll visit:

*http://www.speedtest.net/*

and run the test three times for each trial, noting the test server location carefully so we can check that it's the same when we test using the mobile app.

First open your favorite browser on your LAN-coonected machine and open your favorite browser, it doesn't matter which one for this test. Navigate the URL above, and the Sppedtest app will load and determine your location and choose the best test server for you, usually the closest one geographically. When you choose "Begin Test", the application will perform three types of tests, a ping test, a upload bandwidth test, and a download bandwidth test. The results will be three numbers for each test, and we want to perform the test three times and take the average of the three samples. Table 1-1 below shows my results, with the average values. You should end up with a similar table from your own experiments.

For mobile devices, there's an app from the company behind speedtest.net (called SpeedTest, surprisingly) that uses the same methodology, just on a different device. We can test both Wifi and cellular networks on the phone, and compare them apples-to-apples, but on the LAN side we have to be careful; not only the connection type but the user agent is different, as well as other factors.

On your mobile device, open the speedtest app and the app will identify your location and perform the saame three tests described above. You should first turn off your wifi connection and perform the test three times (and record the results and averages), and then perform the same steps with the WiFi turned on, so you can compare them. Try to perform the tests altogether in the same place and time. Your final result should be

a set of 9 numbers, the averages of the three measurements (ping, upload, and download), usuing each of the connection types.

Once you've completed these tests, what can we learn? let's look at the results I got when I performed this test. Your numbers will be different, depending on your network context.

*Table 1-1. Typical bandwidth test results (These figures are averages over three individual tests.)*

| Connection Type | Ping (ms) | Download (mbps) | Upload (mbps) |
|---|---|---|---|
| LAN | 9 | 120.25 | 23.05 |
| Wifi | 18 | 72.78 | 22.84 |
| LTE | 51 | 7.98 | 1.74 |

Notice that upload speeds are always less than download speeds, due to the asymmetric bandwidth model used by most North American ISPs.

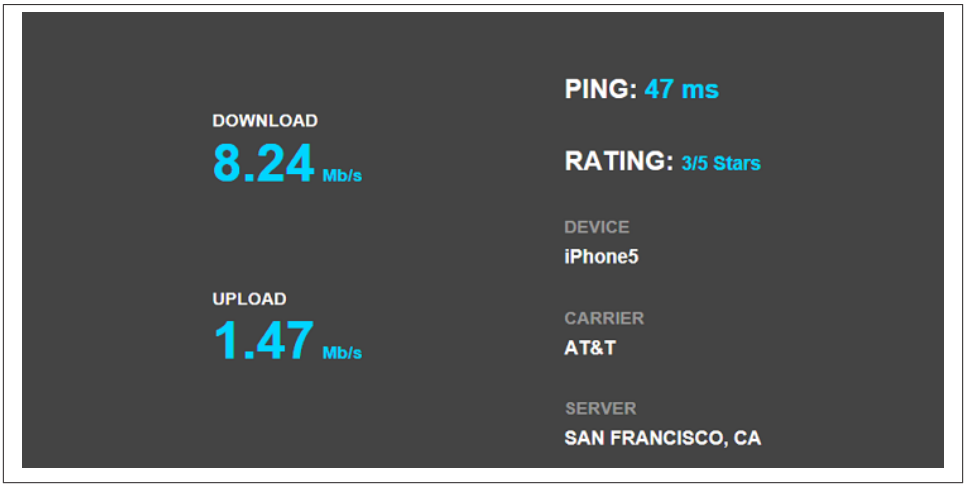


*Figure 1-2. Example test from SpeedTest.net, your results should look similar.*

## What Did We Learn?

It's striking how great the differences are across the different dimensions of response times, even on a fast connection like this one. The results on the cellular device are an order of magnitude less than for Wifi, which is hardly unusual, and bandwidth reduction from the LAN v. Wifi connection was around 40%, also not surprising. The large disparities across connection types suggest that the user experience is qualitatively different for each type as well. We'll see more evidence to support this as we go forward.

Q: Did the general order of the your test results, i.e. LAN >> Wifi >> LTE prove true in your tests? Do you think this is always the case?

A: While this is the general order, it's not always the case. In particular, some Wifi networks may provide lower bandwidth than local celluar networks, for technical reasons.

Q: Since we're testing bandwidth, not reponse time, there are no real *user agents* involved, in the sense we use the term here. Can you think of other factors beyone connection type that vary between our three test runs?

Q: Speedtest.net provided *ping* times as well as bandwidth. The ping time is defined as the time it takes a single packet to travel to the destination, and to receive a single packet in return (also called the RTT or round-trip time). This is the meaning of *latency*. TCP/IP network performance is determined by some (nonlinear) relationship between the point-to-point latency and bandwidth as a function of time. We'll examine this relationship in detail later (when it will seem straightforward, even if nonlinear). For now, it's enough to notice that ping times and bandwidth seem roughly correlated. If you perform the tests several more times, you may see significant variation between ping times, even for consecutive tests under the same conditions in a short period of time. What do you think this variance means for your Web site? Can you think of any factors that will affect ping times?

Q: If you choose a different test server using the Speedtest.net tool, what effect does this have on response times? Is it always the case the the farther away your test server, the lower the bandwidth? Test again, using a far away server, someplace geographically distant, and compare the results to your earlier experiments. Were you able to successfully predict the result?

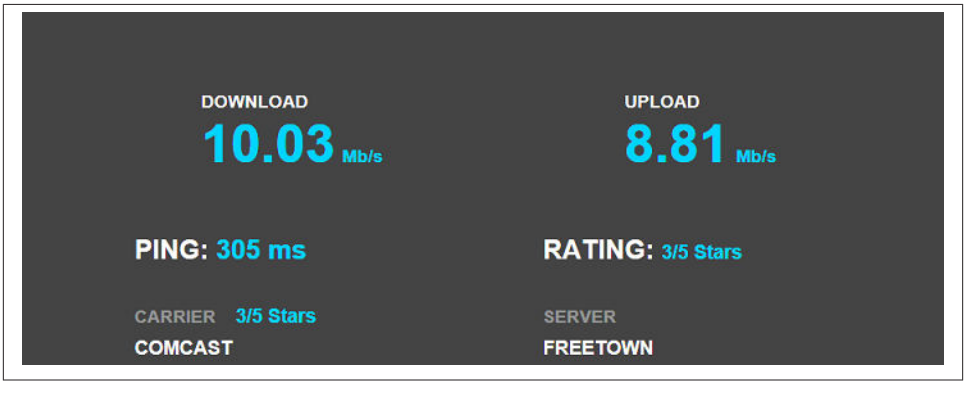


*Figure 1-3. Bandwith testing using a remote server.*

Q: Here's an example result (figure 3) using a LAN connection, to Freetown, Sierra Leone. Note the long ping times - do you think this is the limiting factor in network performance?

Testing your bandwidth everywhere you go can be addictive, so be careful! From this exercise we can see that not only is bandwidth a key factor in performance, but that it varies quite a lot depending on what kind of connection you have, as well as your location. While none if this suprising in itself, we can start to see how the user experience is shaped by the network and client device, and gain some insight into how your own We activities are impacted by your network context.

## User Agents

In the HTTP request/response cycle, the User Agent is responsible for making HTTP requests for resources, and also for managing the presentation of the information inthe response to the user. The first HTTP User Agents were text-based clients with very limited capabilities. Today's user agents are highly sophisticated software programs that have evolved along with the Web, sometimes driving its evolution and sometimes being driven by it. User agents can be of almost any kind on today's Web, and this can be a big factor in user response times.

### Browsers

The makers of the most common desktop browsers such as Internet Explorer, Firefox and Chrome make a point of competing with each other based, in part, on performance. This is good for the industry and good for end-users. Modern 'evergreen' browser techniques that allow for fast, incremental updates also help keep a continuous focus on performance as each of these talented software teams seeks ways of improving the user experience.

How much difference does the User Agent you choose make from a performance point of view? In general, a lot. While changing versions and benchmarks make it hard to distinguish exact numbers, I've encountered cases in my own experience where the choice of browser made a 50% difference in response time. Again, we don't want to generalize from any single measurement, but it's fair to say that some significant differences are not unlikely. It's also the case that even among the major browsers, each of them has some things it does well, such as executing JavaScript code, while another may be faster at parsing HTML, or performing visual layout tasks.

### Mobile Applications as User Agents

More mobile phone connections are made to the Web than for calling and talking to people, and our phones have become sophisticated multimodal Internet 'tricorders'. Having HTTP applications isn't new, many of the earliest PDAs and phones had Web apps. With the arrival of the smartphone revolution, and the app store, everything

changed. Now, a large part of the mobile experience involves apps, especially those built on the Android and iOS systems. In many cases, these apps are also HTTP User Agents, what I call *Directed Agents* or user agents designed for a specific purpose or service. Good examples are mapping apps, which use HTTP, HTML, and Web technologies such as JavaScript under the covers, as well as games and media players. All of these are directed agents, essentially Web clients optimized for a specific purpose or application. One of the primary purposes of building a cutomized native app is to obtain better performance than would be possible by using a generalized user agent on the mobile device, such as Safari or Chrome. This performance difference plays a role in many discussions about the relative benefit of developing native apps as opposed to developing Web-based applications designed for the user's general-purpose browser.

## Traffic Types on the Web

Web traffic analysis is a key skill for understanding Web performance, and understanding the differences in traffic patterns across multiple dimensions like device, user agent, and transport type plays a big role.

When we're trying to understand the performance of a Web service or site, there's a lot we'll need to learn about the traffic patterns the users follow, a process called 'workload characterization'. Some of this can be done by analyzing HTTP log files or other log data. There are also commercial and homebrew systems that can help provide this sort of analysis.

Users typically make different requests based on their own circumstances. Factors mentioned above such as network context, user device and agent, and even the time of day can make a big difference in user requests and traffic patterns.

### API Traffic

Loading a Web page or Web site can be a complicated affair. When we are looking at traffic analysis, one thing we want to know is how much of our traffic involves API calls to remote and/or third party services. If the page calls for a map from a third-party service, for example, the response time of that page will depend in part on the response time of the API calls involved in getting that map on the page.

### Advertising

Advertising makes a huge difference in Web response times, and needs to be taken into account whenever you are looking at traffic distributions. Since many ad exchanges deliver their content based on the (perceived) character of the user, they can take a long time to load. The ads may consist of multiple resources, such as images, CSS, and JavaScript, from multiple domains and locations. Cases where the page weight (number of bytes to be transferred to load the page) varies by 50% due to advertising are not uncommon. Ads are a well-known source of performance issues on the Web, as well as a

significant source of errors. The impact of advertising on the user experience is a huge topic, but in this book we'll only talk about its performance-related aspects.

#### Streaming Media Traffic

Streaming audio and video traffic makes us a significant proportion of the Web today. Reports suggest that streaming video service Netflix makes up more than 20% of Web traffic (in bytes) on some nights in North America. That's a lot of traffic, certainly enough to impact the Web's overall response times in some areas. Measuring and testing streaming media is an art in itself. We'll have more to say on this later in the book.

# Hands On: Our First Look at Web Response Times

But first, let's look at some real-world response times, in this case for Twitter's base HTML page. We'll gather a number of samples taken in a short period of time, and analyze and plot them to learn something about gathering performance data and analyzing it quickly.

## Tasks

For this hands-on exercise we're going to carry out multiple measurements of response time for a single URL, and analyze the results using a *5-number report*. This is a handy way to make some immediate sense out of a groups of sample measurements. Once we understand 5- number reports, we'll be able to characterize the distribution of our sample measurements and see some response time data.

We'll use the Unix command-line tool *curl* to make 10 HTTP requests for the URL *https://www.twitter.com/* to measure response times.

## Gathering Data

The cURL tool is a common Unix client for making HTTP requests. You can think of it as a simple kind of browser. cURL has a lot of sophisticated functionality for making requests, but one thing is doesn't have is any way to render the result to the screen once it receives the response, other than displaying the raw data on the screen. This makes it very useful for quickly checking a URL to see if it's working. For our purposes, we want to use curl to tell us how long it took to receive the entire response.

First open a terminal window on your system, and type the command below. We want to perform this test 10 times, and I've provided a script to do this

The command looks like this:

```
$ curl -o /dev/null -s -w %{time_total}\\n https://www.twitter.com/

0.462
```

The output will be a single floating point number, the total response time in seconds (here 0.462 seconds).

We need to gather several sample measurements to have a good idea of the overall response time. Let's run the cURL command 10 times and gather the data. We'll use a shell script to do this. Our script will stop for 1000 milliseconds between tests, based on the assumption that the request won't take that long! The result should be a list with 10 numbers in it. The script on the companion site will also take care of converting the data values to milliseconds, which is the most common unit of measure for Web response times. If you dont want to use the script, you can just repeat the command ten times manually.

## Generating the Report

Once we've completed our measurements, we want to use the R language to calculate our 5-number report. R is available for all platforms, but you can also use excel, mathematica, or any other number-crunching tool. A 5-number report is, obviously, a set of 5 numbers that we calculate from our data that will help us understand how long it takes twitter to load using cURL. The 5 numbers are: *Minimum, 1st quartile, 2nd quartile, 3rd quartile, and Maximum*. The second quartile is commonly called the *median.* Why do we care about these particular numbers? Once we know them, we can easily draw the distribution of the samples and see what our data looks like. By knowing the minimum and maximum values, we know the range of the data; knowing the 1st, 2nd, and 3rd quartiles gives us a sound basis for understanding overall 'shape of the curve'. The quartiles are simple to understand; they basically divide the dataset into 4 parts (quartiles), each of which contains 25% of the samples. The median therefore divides the lower and upper 50% of the data points, so it's in the middle. The result of our script will be the 5 numbers we are looking for. Let's look at the R script for this, it's very simple:

```
R># Define the response time vector with 10 values for https://www.twitter.com/

R>responseTime<- c(480,516,625,531,465,409,582,440,611,460,419)

R>fivenum(responseTime)

[1] 409.0 450.0 480.0 556.5 625.0
```

Our 5-number report is now complete – let's see what we can learn. First, looking at the range of the data (MAX – MIN = 625 - 409 = 216 ms) we can see that twitter.com's home page loaded reasonably consistently, with a median value of 480 ms by looking at these results.

Notice that our report doesn't include the mean or average value (503.5 ms in this case). Why not? In many cases, the mean value of a set of measurements can be distorted by outliers or extreme values in the data, and the median represents a better estimator.

We'll generally use the median in this book for performance data, and when we use the mean I'll carefully call it out so you can be sure.

## Plotting the Data

One thing I find useful is to plot the data from a 5-number report by hand. Try it – it's pretty easy. Of course you can cheat and use R to make a fancy histogram of the data like this:

```
# Create a histogram for response time

R>hist(responseTime, main="cURL Tests for https://www.twitter.com/",
xlab="Response Time (ms)", border="blue", density=c(10,20,30,40,50))
```

Here's the result from my own tests. Yours should look similar. You can find all of the scripts used here on the book's companion Website.

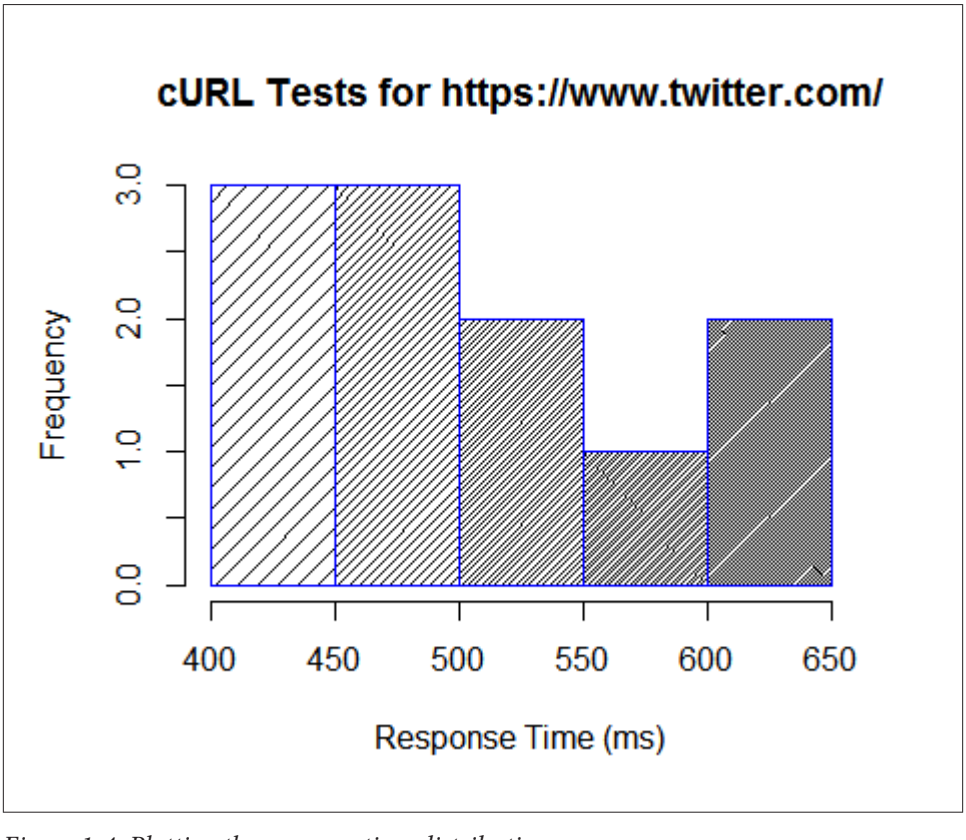


*Figure 1-4. Plotting the response time distribution*

## What We Learned

Working with the data hands-on is fun and helps you build your intuition. Here are some questions to ask yourself about your data, now that you've gone through the analysis.

Q: What does the shape of the distribution of response times tell you about the performance of Twitter's home page?

Q: If you repeated the same test, but on a computer in Norway, how do you think that would change the shape of the distribution?

Q: What if you did the same tests on a mobile device?

Extra credit: cURL is available for most major mobile platforms. Download the app and perform the same tests using your cellular connection (not WiFi). You'll need to enter the data manually into R. Why do your mobile results look so different?

CHAPTER 2

# Introducing the MPPC Model of Web Performance

In this chapter I'll describe a model for response times on the Web called the *MPPC Model*. MPPC stands for *Multiple Parallel Persistent Connections*, because it's based on how browsers and other user agents connect and download Web resources using the HTTP 1.1 protocol. The MPPC model includes the entire end-to-end response time including all of the many factors that have some effect on how fast resources load. The model doesn't attempt to account for each and every factor individually, but divides them up based on their sequence in time.

I originally created the MPPC model in 2005, and it's undergone a lot of revision and improvements since then. This book is the most recent, and best, written description of the model available, so you're getting the most up-to-date information here.

The MPPC model is a mathematical construct, but it doesn't require any sophisticated math to use it. Using the model to solve your everyday problems usually doesn't involve anything more than descriptive statistics and algebra; the mathematics is largely there to provide a means of testing and calibrating the model, and making sure we understand what's happening.

Why do we want to have a model at all, much less a fancy MPPC model? There are several reasons, and several ways you can use the ideas presented here to make your site faster:

1. The model will help you quickly identify performance problems and issues, and give you some idea of the magnitude and source of the problem, in a very simple way.
2. The model helps validate our understanding of what is actually happening, as well as our limitations and what we don't know, and how confident we are in our results.

3. You can use the model to predict trends and understand changes, and spot problems quickly when they occur.
4. You can use the model to help understand your Web traffic and where to focus your efforts on making performance better, or to compensate for performance problems elsewhere in the end-to-end response time.

Having a reasonably rigorous model of the Web's performance is a good way to approach your own performance goals. You can use your understanding of how things work from a performance point of view to make sure that you design your site to be fast by design when it launches (most important!), and quickly address any issues that do occur later.

## Model Overview

Our model is based on three simple ideas:

- The reasons any individual HTTP request/response cycle takes as long as it does varies considerably, depending on what kind of object it is and what role it plays in the content delivered. By analyzing what causes the delays in each different time segment of the overall cycle, we can quickly determine the nature and magnitude of performance problems for individual objects, or identify opportunities for optimization.
- The dynamics of Web performance using the HTTP 1.1 protocol are described by the multi-step, multi-source MPPC loading model, and depends on the page composition and runtime delivery.
- The response time for a Web page depends on the bandwidth efficiency achieved over the course of the E2E, taking into account what happens when many objects are loading at the same time, and understanding how the complete cycle that is actually used in real browsers changes the user experience of the page.

In order to fully describe the MPPC model, I need to first draw a picture of the entire HTTP request/response cycle for a single object. From there we'll work our way up to multiple objects, adding the ability to use (multiple) parallel persistent connections. Along the way we'll introduce a cast of characters for the Web, and an understanding of how all of the pieces fit together. We'll end up with a set of tools – a model – of how things work and how long they take. We'll add all of this up to find the E2E.

It's important to note what our model can't do for us as well as what it can. First, we can't use this model (or any other model) to predict the overall response time for any particular HTTP request/response cycle exactly. There are simply too many factors involved in a single request, and too much variance from stochastic (random) factors, to make tight predictions. The response time for a user's experience of your site is subject to many sources of delay that can't easily be predicted at design-time. This requires our

model to be statistical in nature; our understanding is based on conducting a lot of 'science experiments' on the Web, and acting on the results.

Also, we can't expect that our model will produce good results in cases where the user's environment is subject to transient changes in the network; if the user's machine and network are saturated, this is obviously going to have a huge impact on their experience of your site, and there's not a lot you can do about it. Since these conditions often change over short periods of time, this often introduces a lot of variance in the client-side time segments of our model. Nor can we expect that our model will produce good data without good sampling techniques. How to solve sampling issues when you're measuring your own user's experiences is a topic that we'll address throughout the book.

We'll start by looking at what it takes to load a single object on the Web, and build from there to take into account what happens when we load more complex (and more realistic) pages on today's Web.

## The User Narrative Viewpoint

In this book, we're going to discuss Web performance entirely from the viewpoint of the end user of the Web site or application. I call this the *user narrative* point of view. As we'll see as we go along, there are many aspects of performance that the user cannot see, and can't be easily tested from the user location in time and space. A good example is the time it takes for the HTTP request initiated by the user to actually reach the server. This time segment is invisible to the client, and is absorbed in our model into the overall T2 time. The information is simply not available to the user and has to be estimated. Similarly with the response time of any particular database query taking place behind the scenes, the end user cannot see how long it takes the database to respond (and doesn't care). He or she can only see what happens when an HTTP request is made and the response received.

## The HTTP Request / Response Cycle

Every Web system has three (or more) components in the user narrative. There's a user, using a device running software called a *user agent*. The user agent is responsible for making HTTP requests and receiving and interpreting responses on behalf of the user. There's a Web *server*, or more likely a set of Web servers, which respond to those requests. Finally, there's a *network* that connects them. Sometimes we'll talk about *browsers*, which are special kinds of user agents.)

The basic theme here is the request/response pattern. All HTTP traffic begins with an HTTP request, made by a user agent, and followed by an HTTP response from a server. All of the analysis and thinking in this book will be concerned with how fast the request/response cycles involved in creating the user's experience can be completed.

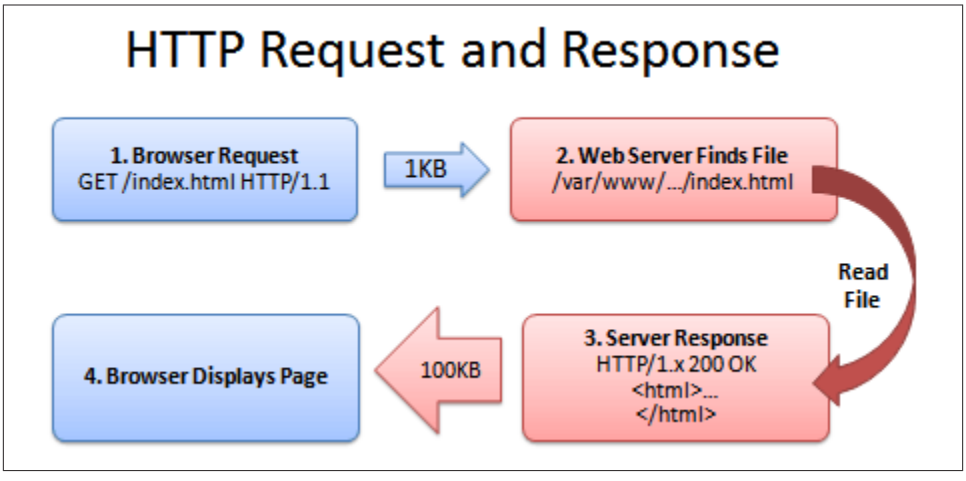


*Figure 2-1. HTTP request response cycle*

### HTTP Requests

The HTTP protocol [RFC 7320] describes requests in terms of 'verbs' or actions to be taken when the user requests them. The most common verbs are GET, HEAD, and POST. GET requests do precisely what the name implies, namely get the most recent copy of a Web resource from the server based on the URL. HEAD and GET will be covered later, but in general, HTTP requests are smaller than responses for GET requests, and mostly consist of technical metadata about the request and the form of any possible response in the form of text-based HTTP headers.

### Response Composition

The response to an HTTP request can be anything that can be transported over the Web, from basic HTML to WebGL-based game information in real-time or a streaming audio or video file. Response times for Web traffic are predictably related to the composition of the response. In the most common case, where the user experience is implemented around the Web page/online application concept, responses may be for resources in one of the Web's three standard languages: HTML, CSS, or JavaScript. How the User Agent handles these responses can be a significant part of the overall response time of the page, as we'll see in later chapters.

## Loading a Single Object: The Simplest Case

Let's take a look at the request and response cycle for a single object on the Web. The diagram below (Fig. 2-2) shows the single-object cycle and its division into time seg-

ments. When a user first requests an object, let's say an image file to keep things simple, we can see the sequence of steps involved.

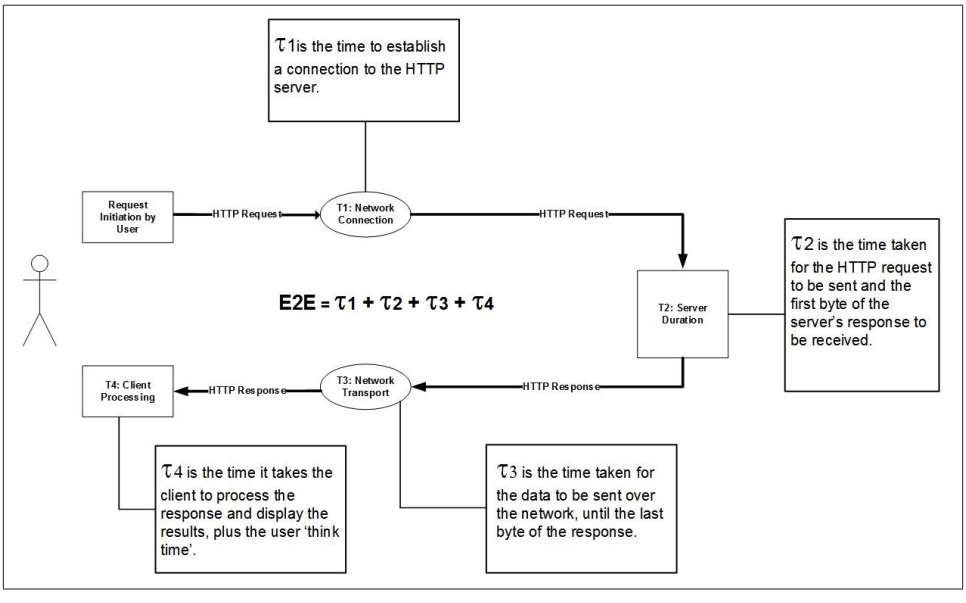


*Figure 2-2. Loading a single object over HTTP*

First, the user clicks (say) on a hypertext link to the image in question. The URL in the link might be *http://somedomain.com/images/picture.gif*. When the user clicks the link, the user agent, in this case a common desktop browser, generates an event that leads to the display of the image on the user's screen.

As you can see, we've divided the end-to-end response time into four sequential time steps, which we'll simply call T1, T2, T3 and T4. For a single object, the first 3 steps happen in a sequential and linearly independent way (but not T4), and our strategy will be to add them up to get the total time:

$$T_{E2E} = T_1 + T_2 + T_3 + T_4$$

We'll use this equation (sometimes converting the values to percentages) throughout this book.

## T1 - The Connection Time

The first time step, T1, is the *network connection time*. It's the sum of the time taken for all of the work required to establish a TCP connection to the host machine that will respond to our request. This includes all of the steps involved in setting up our connection, including DNS, TCP connection time, and TLS handshake/setup time (if any).

Once T1 is complete, we'll be ready to send an HTTP request to the server in question, and to deal with the response when it arrives.

## T2 - The Server Duration

The second time step, T2, is the *server duration*. It includes all of the time required to actually send the HTTP request to the server, until the client receives the first byte of the response. T2 encapsulates the time required for the server to do whatever processing needs to occur in order to send a response, including any database calls, service requests, or any other work that needs to be done behind the scenes. For a single image, as in our example, the work necessary is minimal, but for real-world Web pages back-end processing can take up the significant part of the total E2E.

T2 is often the first target for performance optimization. Many developers think that if their site is slow it must be due to their server being slow. This is often not the case. One of the advantages of the MPPC model is being able to understand how much of a role your server (and the associated processing) play in your end-to-end performance.

## T3 - The Network Transport Time

The third time step, T3, is the *network transport time*. It covers the time from the receipt of the first byte of the response to the last, and takes into account how long it takes to actually deliver the content to the user agent. This is where the underlying TCP/IP network plays the largest role, and understanding T3 will require some understanding of how bytes are transported over HTTP and its underlying TCP/IP protocol.

## T4 - The Client Processing Time

T4 is the *client processing time (+user think time)*. It includes all of the processing that the user agent must perform on the response content in order to complete the user's action. In our case, this means displaying a picture on the user's screen. When exactly T4 completes can be a difficult question in some real-world situations, but for simple processes like loading an image file it's pretty straightforward.

Here's the full E2E equation, including all of the gory details:

$$T_{E2E} = T_1 + T_2 + T_3 + T_4$$

$$T_{E2E} = (T_{DNS} + T_{conn} + T_{TLS} + T_{redirect}) + (T_{req}+T_{server}) + (T_{TCP}) + (T_{client}+T_{user})$$

Together, all of these steps add up to a complete cycle, and the time taken is the total response time for that object. We'll go over all of these contributions to the E2E as we go along.

This is a very simple model for what is actually happening when we request an object over HTTP, and it gets complicated once we begin to take multiple objects and persistent

connections into account. It's also not entirely the case that in the real world, all of these steps happen in precise sequential order – we know from experience that a large image file might begin to render on the screen even while it is still downloading, so that T3 and T4 overlap in time. In fact T4 almost always overlaps with T3 and can't be considered linearly independent in any rigorous sense, even in this simplest case. This is one of the limitations of our model, and we'll find that T4 is often more difficult to deal with than the other time segments.

Another point regarding T4 is that it also includes the user's *think time,* the time it takes the user themselves to receive the information (often via a high-bandwidth optical connection, the Mark I eyeball!), process the information in the context of whatever task they initially set out to perform, and respond accordingly. This becomes important when the user undertakes a serial sequence of events such as checking out of an online store, accessing email, or other activities that involve well-defined sequences of user events.

Our simple single-object model allows us to break the total response time down into several steps and analyze them separately. In the next few sections, we'll discuss each of these steps in detail, and then put it all together at the end.

## Hands On: A Single-Object Example

In order to increase our personal knowledge and get some experience with this simple single-object model, let's work through an example. We'll use a variation on the same cURL script we used before, and get the individual values for T1, T2, and T3. We'll ignore T4 for the moment, since cURL doesn't do any rendering or processing, other than dumping text to the screen, and that time is not included in cURL's output.

The script is freely available (along with many others) at this book's companion website.

In this case, the script does four things:

1. Clears the local machine's DNS cache (so we don't see cached results)
2. Calls cURL with options to prevent HTTP caching and write the response times to a csv file
3. Runs the awk utility to process the output from cURL into a useful format.
4. Runs R to produce a graphical image of the output for easy visualization.

The script is called 'mppc-single-object.sh' and the results look like this:

```
MPPC response times using cURL

================================================================================

URL                 timestamp (UTC)     T1  T2  T3  Status Total (bytes)

================================================================================
```

```
https://twitter.com/ 06/25/2014 03:05:19 325 26 233 585 200 50837

https://www.ebay.com/ 06/28/2014 22:33:57 308 108 194 610 200  228448
```

I've modified the earlier cURL script to display the MPPC values (in milliseconds) as output.

We can immediately see that, at least as far as retrieving the base page for *www.twitter.com* is concerned, T1 > T3 >> T2, with T2 being very small compared to the connection and transport times. If you worked at Twitter, where would you spend your time optimizing? Do any of these numbers seem unusually large or very different from what you expected? If you guessed that T1 would be largest, you were right!

I've also included a small script to use R to produce a simple bar chart showing our results as shown in Fig. 2-3 below. This is very helpful because we can immediately see visually the relative proportions of the different time segments involved in loading this object. Here's the R command:

```
R> barplot(t(test2),legend = colnames(mppc),col=c("blue","red", "green"), horiz=1, xlab="Response
```

assuming you've loaded your data into an array called "mppc". As always, you can use any other tool that suits you for plotting data.

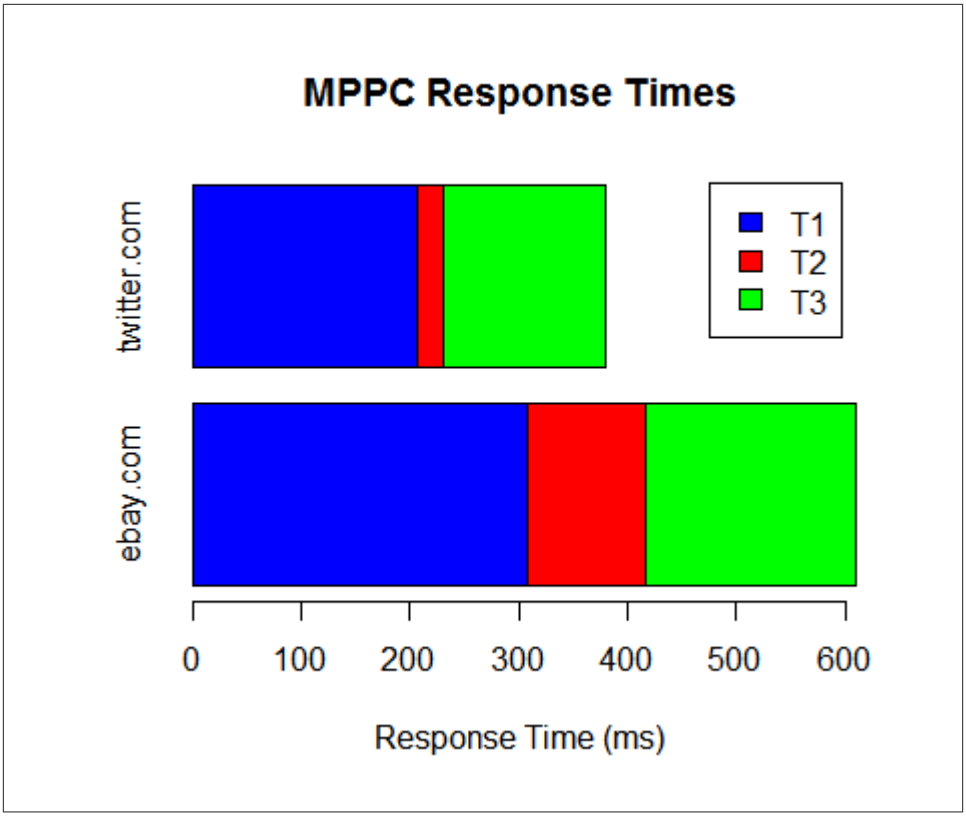


*Figure 2-3. Example MPPC response times for a single object*

One things we can see immediately is that when we look at different Websites we see different relative proportions for different time segments. In this case, eBay's server, unsurprisingly, spent a lot more time composing the page than twitter's server did; such differences reflect the differences in what these sites do, as much as how their data is delivered. The eBay base page is roughly 4 times larger in bytes as well (but that doesn't mean it takes 4 times as long to download – see chapter 5).

# Working with Multiple Objects

## The Multi-step MPPC Request/Response Pattern

So far, our model only describes what happens when we load a single object. In the real world, Web pages usually consist of many objects, which may be sourced from many servers in different places. How do we account for this in our model?

In order to see this, we need to understand the multi-step MPPC loading cycle that a browser uses to load a Web page, shown in FIgure 2-4 below. When the user first initiates an HTTP request, perhaps by clicking on a hyperlink, the browser takes a number of steps. (Here I'm using a typical browser/Web page example, but this pattern applies generally for the HTTP 1.1 protocol.

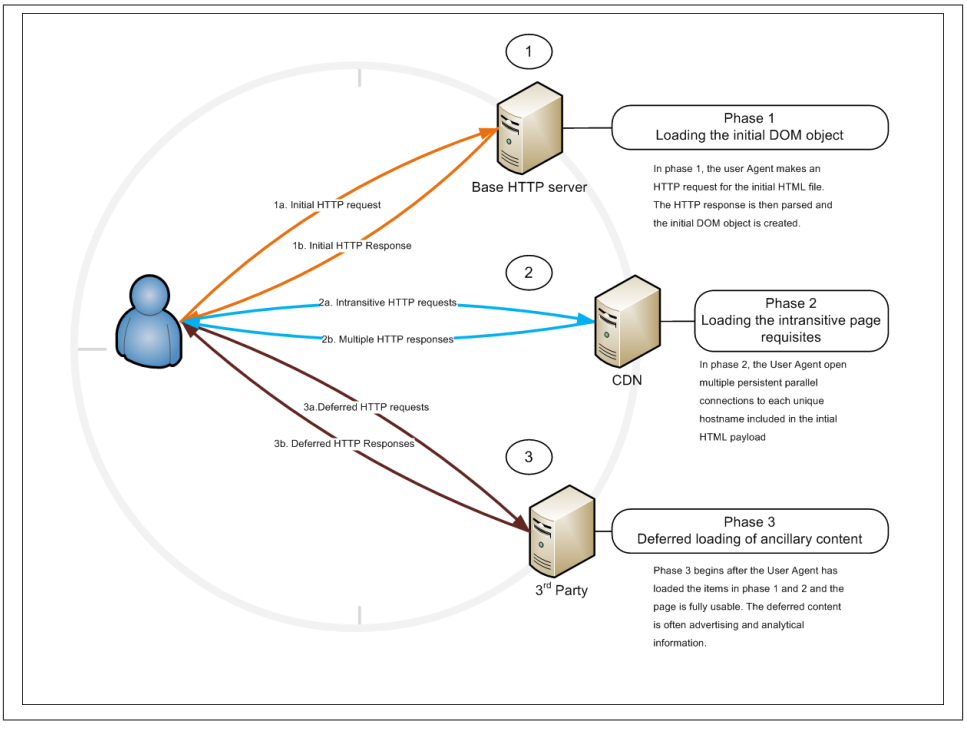


*Figure 2-4. The Multi-step Loading Process. The delays are different in each step.*

## Step One: Loading the Initial DOM Object

The first step is to retrieve the HTML text file addressed by the URL that formed the target of the initial hyperlink. We'll call this the 'DOM object' or sometimes the 'base page'. It's typically an HTML file, and contains a formatted description of the Web page, its structure, function, and hypertext links to associated objects, such as images, scripts, and style sheets. 'DOM' here refers to the HTML Document Object Model, a W3C standard. (REF)

## What is the DOM?

The DOM, as we usually refer to it here, is a data structure that's described by the HTML tags in a document. In this case, we're talking about the initial HTML document for a

website, but every HTML document describes a DOM or a DOM fragment of some kind. (This file is named 'index.html' by default, and when we make a request for *https://www.twiter.com/* we're really requesting *https://www.twitter.com/index.html*.) The DOM itself describes a hierarchical tree structure, where individual 'nodes' are connected to others either by parent-child relationships (a list item is a child of a list node) or by sibling relationships (a list item may have several sibling list items. The parent-child relationship is a 'vertical' relationship and sibling relations are 'horizontal' in some sense, so we are able to construct DOM trees with both height and depth as desired.

Figure 2-5 shows an example of a simple DOM structure:

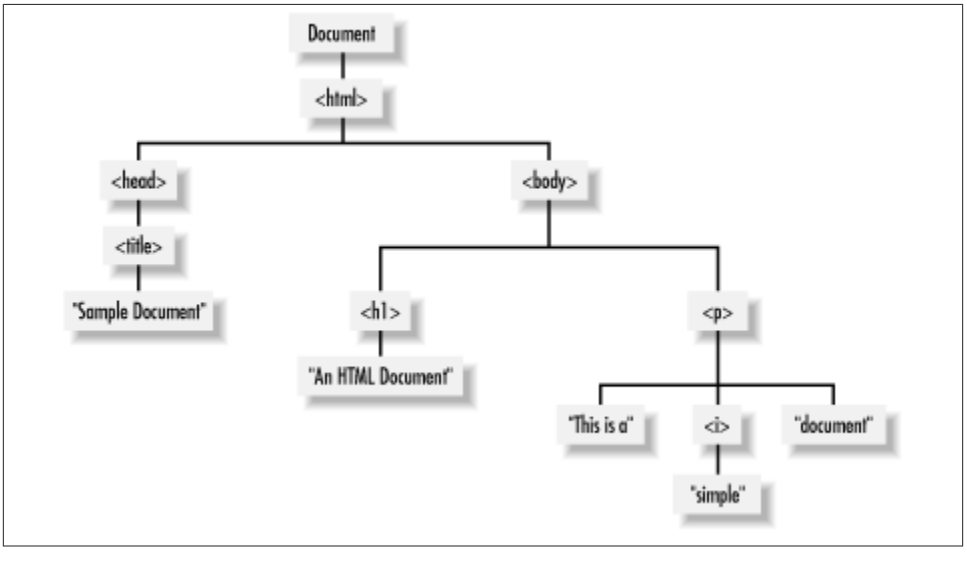


*Figure 2-5. An example of a DOM object and the HTML document that describes it. (Taken from JavaScript the Definitive Guide 4th edition, O'Reilly.)*

And here's what the HTML document itself looks like:

```
<html>
    <head>
        <title>Sample Document</title>
    </head>
    <body>
        <h1>An HTML Document</h1>
        <p>This is a <i>simple</i> document.</p>
    </body>
</html>
```

The DOM as delivered by the HTTP server in the initial HTML document is a static structure, in the sense that it is fixed until it's received by the client; then it becomes a dynamic structure in the sense that its structure can be changed at runtime, either by

using JavaScript to modify the DOM directly, by adding or removing elements or 'subtrees' of elements, or by making an inline HTTP request for content that's to be used to modify the DOM. These inline requests are usually accomplished via the XMLHTTPRequest (XHR) object, once again using JavaScript. We'll describe the performance aspects of these inline XHR calls as we go along.

Later in the book we'll talk about 'states' rather than documents or pages, and how to take into account changes to the DOM and inline HTTP requests to improve performance by analyzing the time it takes the user to complete a task rather than loading a document. This method of performance management is better suited for the dynamic Web-based applications that we encounter in the real world, and allows us to solve performance issues at the level where it really matters to task-driven users and sites.

The initial DOM object, or base page for an HTTP request may be very simple, with just some basic text (often saying 'under construction'!) or it may be amazingly complex (look at the HTML source for the Yahoo! Home page). Some pages may look simple but are not, such as Google's search page. In each case, the browser will initiate a request for the base page, and (hopefully) receive an HTML response. The browser will then *parse* the HTML file, creating the initial DOM tree, and identifying all of the other URLs embedded in the text file, immediately queuing them up for retrieval as well. This is the second step of the process – loading the necessary page objects after parsing the base HTML file into a DOM object.

This first step in the process is qualitatively different in performance terms than any of the following steps. This is one of the key ideas of our model.

## Step Two: Loading Intransitive Page Objects

Once the initial DOM object has loaded (or even before), the user agent extracts all of the embedded (intransitive) URLs from the DOM and begins retrieving them as fast as possible.

In step two, the primary activity is retrieving multiple files automatically, using parallel persistent connections, to a server that is (by intention) considerably closer to the user's physical location (and therefore hopefully faster) than the primary HTTP server in step one. How many connections can be opened in parallel is determined by number and size of the objects retrieved, as well as the browser connection constant *K*, as we'll discuss in chapter 5 on network transport. For now, it's enough to say that given some basic information about the page and the response times of the individual servers involved, we can predict the overall response time for step two within reasonable confidence levels.

## Kinds of Hypertext Links

This is a good time to make an important distinction about hyperlinks on the Web. Hypertext itself is a rich and complex set of ideas, and the Web currently only supports a limited subset of hypertext functionality. Specifically, we have two kinds of hyperlinks, *transitive* and *intransitive*.

*Transitive hyperlinks* are those that require users to explicitly actuate them, usually by clicking on a bit of underlined blue text, and so transitive hyperlinks are often associated with an <A> (anchor) tag in HTML.

*Intransitive hyperlinks* are those associated with page objects, and do not require a user to actuate them. Intransitive hyperlinks are actuated by the browser on the user's behalf, as part of this second step in the MPPC process; the browser actuates these hypertext links 'automagically'. Intransitive hyperlinks may resolve to images, scripts, style sheets or other objects using HTML tags that carry the 'src' or 'link' attribute. The <IMG> tag is the most common use for intransitive hyperlinks.

The majority of hypertext links on the Web are intransitive; no user ever clicks on them! Transitive links almost always refer to an HTML file with a DOM description; intransitive links primarily lead to 'dumb objects', often in binary formats.

I've made this description a bit simpler in some respects than things are; many things are happening concurrently during the page loading sequence, including the page layout, tasks associated with client code (usually JavaScript) loaded by the page, and other factors. It's also important to note that the browser doesn't necessarily wait for the entire base page to load in its entirety before starting to load the secondary resources; the browser will usually try to do whatever it can to optimize the loading process, including reading ahead, making as many concurrent calls for resources as possible.

Modern Web sites take advantage of technologies like XHR, CSS, and (a lot of) dynamic scripting to build their presentation, and the industry is (slowly) moving past the 'onload' model we're describing. We'll extend our model to account for these more complicated scenarios along the way. None of these 'Web 2.0' technologies changes the fundamentals of the MPPC model, which is based on the dynamics of HTTP 1.1.

This shift toward a more dynamic Web has multiple effects on performance; one of the primary ones being to shift some of the processing formerly done on the server (in T2) to the client (T4). This allows sites to be more dynamic, and reduces network traffic, a much better (and faster) way of doing things. It clashes somewhat with another countervailing trend toward less capable mobile devices however. This will improve rapidly as the devices themselves improve.

The HTTP requests made in step two are typically made to a Content Distribution Network (CDN). CDNs are caching proxies for the Web, essentially farms of special-purpose HTTP servers located in data centers close to the end user's physical location,

or at least closer than the base server in step 1. CDNs and how to use them to make your site faster are a big topic in chapter 5, and later in the book we'll talk about how to set up your own CDN, either using commercial services or building it yourself.

The types of files delivered in step 2 are often very different from step 1. While step 1 included a single HTML file, step 2 is largely made up of non-HTML content, usually either binary content such as images, video and audio files, presentational content such as CSS style sheets, or client code such as JavaScript or Java applets. The order in which these content types are loaded, and how they impact the user experience, will be the subject of Chapter 6.

How do we know when step 2 is done? In this case, 'done' means that the page is fully loaded and operable, and the user can carry out their tasks. The user agent signals this by emitting a *load* event for the top-level DOM object. The *load* event is part of the DOM standard and at least in simple cases, we call 'time' when this event is fired. This however doesn't mean we're done yet!

## Step Three: Deferred Loading

Deferred loading is a performance improvement strategy often used to improve user-perceived response times, especially on sites hosting display advertising. Instead of loading everything on the page in step 2, the deferred loading strategy identifies some parts of the presentation that are of secondary importance to the user's task at hand, like ads, and intentionally defers loading this ancillary content until the main functionality of the page is available to the user. This is a standard technique borrowed from the video-game industry, extended and applied now to the Web.

Once step 2 of the loading process is completed and the *load* event has been fired, the user can freely interact with the page in a normal way. If we're measuring the response time of the page, this is the point where we can say the page is done loading. Sites that use deferred loading, however, don't stop loading data after the *load* event fires, but continue to load content and run scripts in a third 'deferred loading step'. Often the *load* event triggers the start of the deferred content cycle.

Typically, deferred loading is used for advertising and page analytics, both of which often require some client code execution as well as rendering images to the display, and so require additional client processing time.

Deferred loading is also used as a means of updating the display with fresh content, such as scrolling stock tickers or news stories, and so step 3 may continue for the length of the user's session. Measuring the performance of these active services can be difficult, but usually these activities will not directly affect the user or their behavior (unless they click on an ad).

Strictly speaking, for a given page, the deferred loading cycle ends when the user agent fires another event, called *unload*. Just as the load event is fired when the DOM is

completely loaded, the *unload* event fires when the DOM is unloaded, which usually means the user has gone to another page. At that time, all tickers, ads, updates and other deferred content is unloaded and new content is loaded to replace it. This cycle of *load* and *unload* events punctuates the MPPC loading process, and the user's session can be conceived as a set of successive *load* and *unload* events, with the MPPC process happening between them.

## Pitfalls of Deferred Loading

Deferred loading is a very good way to improve the E2E, but it does have some drawbacks in some circumstances. While the deferred loading cycle in step 3 usually doesn't have any direct impact on response times as measured using the *load* event, they may still impact the user experience by using scarce system resources and network bandwidth on the user's client and in some cases, cause the following page to load more slowly as the *unload* event takes some time to stop any existing deferred loading processes. Some processes may also block others, a notorious problem examined in detail later, and this can affect the overall E2E.

## What Does MPPC Mean?

Now that we've reviewed the multi-step process used to load resources on the Web, we can see what 'multiple parallel persistent connections' means for Web performance. In step 1, a single object, the initial DOM, is loaded. But in step 2, the user agent can open more than one connection at the same time, and keep those connections open (avoiding the T1 time segment altogether) until all of the intransitive page objects have been loaded. In step 3, the deferred content is also loaded across using MPPC. So in reality, we really only use MPPC (and HTTP 1.1) in steps 2 and 3.

Prior to HTTP 1.1, connections could not be made persistent, and so each object required a complete connection/disconnection cycle. User agents were free to create multiple parallel connections for resources, but the connections had to be opened and closed for each object. So the MPPC model is specific to HTTP 1.1. We'll see a hands-on example of how this works in practice in Chapter 4.

## Summary: MPPC Model Overview

This chapter is intended to provide a brief summary of the MPPC model, just enough to get us started on the next few chapters, where we'll cover all of this in more detail. There's a lot more to learn, but we can already begin to use these ideas to identify where the delays are in the user experience. Once we understand how objects are loaded by the user agent, and how the MPPC loading process plays out, we have all the information we need to identify performance problems, and up to a point, their causes.

Let's look at some important points to be made about the MPPC cycle and how it affects performance. Since step 1 involves a single object, and must needs be done before the rest of the page can load, the response times for each MPPC time segment in step 1 are crucial. Problems in step 1 have a magnified effect on the user's perceptions of response times, much more so than in step 2. The connection in step 1 is made to the 'base' server in some sense, as its domain is usually associated with the site's provider i.e. www.twitter.com. This initial connection may be, and in fact often is, the only connection to this domain that occurs when the page loads. Many, sometimes all, of the items loaded in step 2 are typically stored on a different set of (CDN) servers with a different domain name. The items loaded in step 3 once again typically call yet another group of third party servers, for analytics, advertising, and content updates.

This complicated multi-step conversation involves a lot of actors, and each of them can have a huge impact on response times. In step 1, the T2 server duration is very important; this is where the main functionality of the page is composed by the server, and so might be expected to take up some significant amount of time compared to T1 and T3. This is where the majority of the server duration for the entire loading cycle happens.

In step 2, the server duration for each object is likely to be far less, since the server is only transmitting files, and step 2 response times can reasonably be expected to scale linearly under load. Also, since we're using persistent connections in step 2, we might expect that connection times (T1) would be less significant (since connections only need be made once) and the transport time (T3) to be much larger, as a percentage, than in step 1, since the step 2 content often includes images and other byte-heavy content, where the (plain text) base HTML page delivered in step 1 is usually much smaller.

At each step, the user's client requests a different *kind* of information from a different set of servers located in different places! This means that the response times and causes of delay are also quite different. If your users are experiencing server delays during step 2, the causes and solutions to the problem are likely to be quite different than if they are experiencing delays in step 1. The user's perception of these delays will also be quite different as well.

The table below (Figure 2-6) shows what we might expect to happen to response times when a page loads, using our MPPC model:

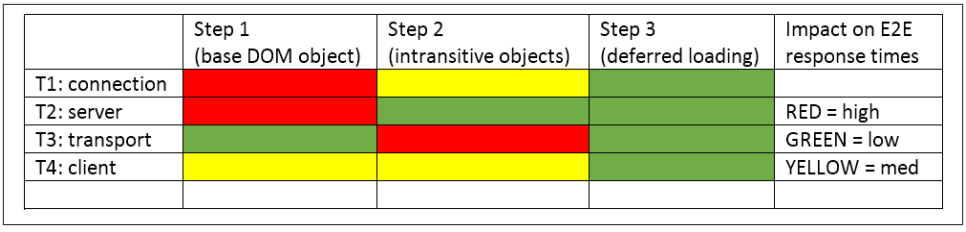


*Figure 2-6. Relative levels of impact on E2E response times in each step of the MPPC process*

In the table I've tried to indicate the relative impact of each part of the overall cycle. I've marked all the deferred loading steps as green, or low impact (since they don't impact the E2E directly). In step 1, the transport time is usually small, for transporting the base HTML page, but is much larger in step 2, as described previously.

There are more qualitative differences between the steps in the MPPC model, as shown in Table 3.

*Table 2-1. Qualitative differences among the steps in the MPPC cycle.*

| | Step 1 (base DOM object) | Step 2 (intransitive objects) | Step 3 (deferred loading) |
|---|---|---|---|
| File Types | text/HTML | Media files, style sheets, scripts | All file types |
| Client Caching | No | Yes | Partial |
| File Compression | Yes | Partial (limited) | Partial (limited) |
| Connection | Single | Multiple parallel | Multiple parallel |
| Link Types | Transitive | Intransitive | Intransitive |
| HTTP servers | Base domain server | CDN | CDN/3rd party |

There are two key ideas here, the segmented HTTP response time cycle and the multi-step loading pattern that defines the MPPC process.

Our model is connection-oriented; it consists of the response time breakdown for a single HTTP request/response cycle into T1, T2, T3, and T4 time segments, which when combined with the multi-step loading model provided by HTTP 1.1 and some algebra, will give us a dynamic picture of what happens when a page loads. These are powerful ideas, and once grasped and applied they'll form the basis for our approach to understanding how the web works and why it takes so long.

Just for review, the different time segments in the HTTP request response cycle are:

- T1 - Connection time
- T2 - Server Duration
- T3 - Network Transport Time

- T4 - Client Processing Time

By looking at some examples, we were able to measure response times for a single object using cURL and R to visualize the overall performance metrics. (We also noted that these spot measurements taken from our desktops are not good data samples in a well-organized test regime, and so they aren't terribly useful except for quick-look diagnostics, because our model is *statistical* in nature.)

The three steps in the MPPC loading cycle are:

- Step 1: load the 'base page' or initial DOM object
- Step 2: load the secondary page objects via intransitive hyperlinks
- Step 3: load ancillary content via 'deferred loading'

We've also learned that *transitive* hyperlinks are those the user actuates of their own accord, and *intransitive* hyperlinks are those actuated on behalf of the user automatically by the user agent. Transitive hyperlinks and the request/response associated with them make up step 1 of the MPPC loading process. Intransitive hyperlinks make up step 2 and 3 as well. Step 3, being designed **not** to affect the user's response times, is not included in the E2E in most cases.

We've noted that each of the 3 steps involves requests to different sets of servers used for different purposes and response times, and that this adds to the complexity of measuring response times, which can vary considerably between one page loading cycle and the next. Response times are not constant, but depend on a number of factors, some of them adding stochastic noise to the data.

I've left out a lot of the mechanics of the model at this point, and issues such as validation, calibration, and congruence of the model aren't discussed at all. This doesn't mean our model is any less rigorous; we can certainly validate each part of the model and identify the (many) sources of approximation and errors. But for our purposes in this book we're going to stick to using the model to improve response times.

## Hands On: Looking at Waterfall Diagrams

In this hands-on exercise we're going to have a look at our first *waterfall diagram*. Waterfall diagrams are a type of hanging bar chart commonly used to display response times for HTTP clients. Waterfall diagrams can be shown in many different formats, depending on the tools involved. A waterfall diagram shows the complete MPPC loading cycle, instead of just step 1 as in our earlier exercise.

*Learning to read waterfall diagrams is a key skill for Web Performance work*. Once you get the hang of it, you will be able to easily identify issues and problems, learn to recognize transient problems from chronic ones, and begin to gain some personal, hands-

on knowledge of the site you are looking at. With enough practice you will be able to immediately understand the performance of a Web page or request just by looking at the waterfall diagram, even from across the room!

The MPPC model described here show us that waterfall diagrams should all essentially have the same pattern, as shown in the figure below. There's a single bar for the download of the base page, followed by many connections to retrieve the page assets in step 2, and then some additional entries that all begin at (roughly) the time most of the previous entries in step were complete. Nearly every Website will follow a loading pattern like this; the waterfall diagram neatly displays the MPPC model, and the MPPC model in turn encapsulates the dynamics of the HTTP 1.1 protocol. Every XHR call, for example, strictly follows the MPPC model, or rather the model for a single object (which may result in further intransitive calls, if the object is an HTML fragment).

## Task

Use common desktop tools to view the waterfall diagram for www.twitter.com/

## Tools

Internet Explorer and Chrome both provide built-in developer tools for analyzing Web pages. Many other tools, both free and commercial ones also provide a waterfall view of performance data.

For the screenshot used in this example I'm using IE 11, but you should use the browser that works best for you.

## Running a Test

Open your favorite browser from the list above and press F12. You should see the developer tools window open. You may need to adjust the window size. Choose the 'network icon on the left and at the top you'll see a 'record' button – a small green arrow pointing to the right. Press to start capturing network traffic for our waterfall diagram. After the recorder starts, type: *http://www.twitter.com* into the address box of the browser. As the browser begins to make HTTP requests (and receive responses), you can watch the elements loading and see the waterfall diagram form. Your diagram should look similar to this, although not exactly the same.

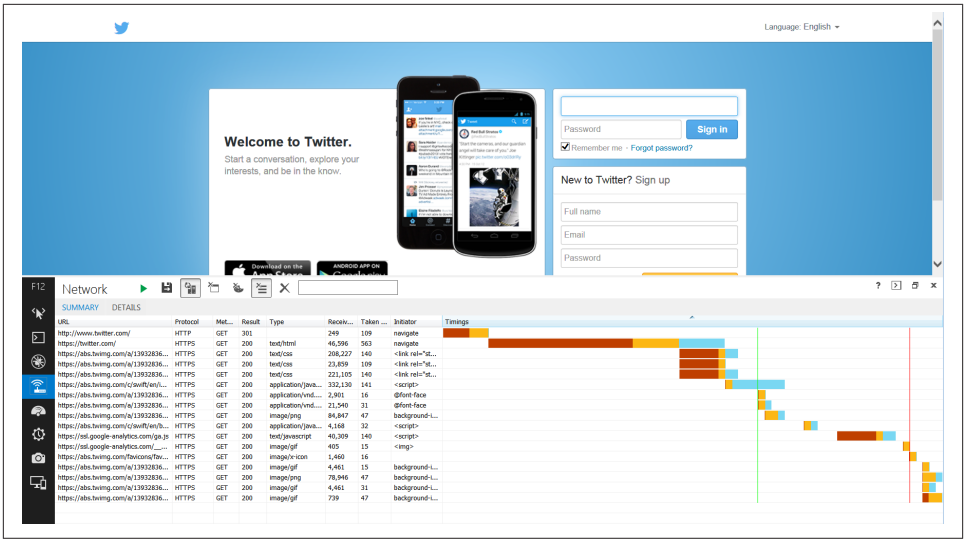


*Figure 2-7. Example waterfall diagram for http://www.twitter.com (needs to be marked up for steps and colors)*

When looking at the image in Figure 2-7 for the first time, it can be a bit confusing. If you're already familiar with waterfall diagrams, you'll recognize the items displayed here. Each entry in the table on the left represents an item on the page – an object of some kind that is loaded and (possibly) displayed on the screen. The colored bar beside each entry shows how long each item took to load, and how the time was spent. We can see immediately that some objects take a great deal longer than others, for example. There's a wealth of information here, some of it useful and some it less so. We can immediately see what happened as the browser loaded the page, and understand the response times of the objects and their relationships. Once again, this is a single test, done from a single computer located in a specific place (Silicon Valley) using a single browser, at a given time of day. It's important to remember that we can't draw conclusions based on a single sample. To convince yourself of this, run the test again. You'll see that the numbers and spacing of the objects are now different, for any of a number of reasons. You can usually see a strong pattern in the waterfall for a given site.

Waterfall diagrams are a good example of a *diagnostic* test; a test conducted in order to see if the HTTP server(s) are correctly delivering the content to the client. If you want to see if your page loads correctly after you've made a change, this is good way to do it. If you want to know if your page loads rapidly in Norway, this test is useless. Understanding how to identify and conduct the right tests ('science experiments') is the subject of a later chapter.

## What We Learned

Understanding all of the information presented in the waterfall diagram will require reading the rest of this book! Or at least a few further chapters. But we can ask some simple questions and learn a lot, both about what we do know, and what we don't.

Q: How long did it take this page to load?

Q: Which object took longest to load? Why?

Q: If you perform the same test again, in the middle of the night when fewer users are on the network, what do you think the results will be? What if you are on a wifi connection? A mobile phone? What other variables can you think of that can make a difference in the waterfall diagram for a specific URL?

Q: Can you identify the steps in the MPPC process in your waterfall? (Hint: I've marked them up in the image above.)

Q: How much time was spent in step 1? In step 2? In step 3? What did you expect?

Extra Credit: use another tool (perhaps a different browser) and generate the waterfall diagram for the same URL. They're likely to look somewhat different. Can you identify the same basic loading pattern across these tools?

CHAPTER 3

# T1 - The Connection Time

## Introduction to T1 - The Connection Time

The T1 time segment is defined as the time required to establish a TCP (or TLS) connection to the server to which the HTTP request will be sent, and from which the (initial) response will be received. All of the T1 time occurs before the first HTTP request is even sent. The entire T1 sequence occurs at the network level, and no HTTP request is actually sent until the beginning of T2. As we go through this chapter, you'll see that setting up a connection for an HTTP request/response cycle involves a lot of work, and can take quite a bit of time, even though nothing's happened yet! This is especially true for secure HTTPS connections.

From a performance point of view, T1 is important because:

- It affects the user's perception of how long a request takes to receive a response,
- By minimizing the proportion of the overall end-to-end response time (E2E) you can reserve more of your overall response time budget for issues like content and features,
- For many secure sites, T1 response times dominate the entire E2E, at least for the initial (and often most important) connection to the base HTML document
- Mobile users often experience significantly greater T1 response times, for a number of reasons.

Optimizing T1 can be a difficult challenge; T1 involves complex network issues that may require careful work just to identify, much less fix. Sometimes 'fixing' a problem in the T1 time segment involves working with network service providers rather than your own internal staff and colleagues. However, T1 issues, because they happen at a lower level of the stack, commonly affect many if not all of the users of a particular

service, so fixing them is both necessary (because so many users are impacted) and very worthwhile, for the same reason.

The T1 time segment is the sum of the individual response times required to establish a TCP connection to the host. It includes the time required to look up the IP address for the host, establish a TCP connection, and then establish a securely encrypted connection using TLS (if needed):

$$\mathbf{T_1 = T_{DNS} + T_{connect} + T_{TLS} + T_{redirect}}$$

We'll look at each of these steps in detail, along with ways and means for measuring, testing, fixing, and optimizing the overall response times involved, with the exception of the redirect time segment, which is part of T1 but requires some background on HTTP first. I'll explain redirects at the beginning of the next chapter.

## Tools Used

T1 is purely network time, and we'll need some special tools to measure it. All are freely available. You'll want to become familiar with many of these basic tools for network testing:

- Ping – a common UNIX command line utility for testing the round-trip times (RTT) between two servers.
- cURL – another UNIX command-line utility, cURL acts as an HTTP client, connecting to a server, establishing connections, and downloading resources. cURL has some limitations as an HTTP client; it doesn't provide any display to the user! This is actually useful when you are trying to understand the effects of T2 on overall response times. cURL supports persistent connections, but doesn't try to retrieve intransitive hyperlinks. It does a good job of measuring overall response times for a single HTTP request however.
- Webpagetest.org (WPTO) – this is a free online tool for measuring Web performance. WPTO is a great tool and we'll use it often throughout this book. It's very similar to many of the larger, more expensive, commercial systems.

In order to support this book, and generate some of the data we'll be looking at later, I created a cloud-based performance testing system using WPTO, and I'll share that experience here too as well.

- For TLS testing, I used a freely available toolkit and library called OpenSSL. This tool will help you examine the internals of secure connections and test their performance. OpenSSL is widely used by many popular sites for their secure connections.

*Please make sure you are using the latest version of OpenSSL to avoid serious security risks!*

# DNS: Internet Names and Addresses

DNS is the *Domain Name System*, first deployed in 1983 [RFC 882 & 883], and it provides a means of finding the *IP address* associated with a specific *domain name*. You can think of DNS as an Internet phone book, and in fact one of the first implementations of DNS, developed at Sun Microsystems, was called "yp" for yellow pages. The correspondence between looking up a domain's IP address and looking up a telephone number in a phone book is often used but is somewhat oversimplified; DNS can be more complex than it first appears. After all, the phone book doesn't require real-time updates across millions of computers all over the Earth.

Computers on the Internet use specific numbers for addresses, based, naturally enough, on the Internet Protocol (IP) addressing scheme described in RFC 791. Every device that is connected to the Internet via the IP protocol has an IP address, including phones, tablets, and even some cars and light bulbs. An IP address is a 32-bit binary number, slowly upgrading to 128- bits as the industry moves from IPv4 to IPv6. Only IPv4 will be used in this book.

An IP address has a human readable notation, usually 4 groups of 3 digits separated by dots, like this: 172.16.254.1

(This isn't a real IP address.)

A domain name on the other hand is a human-readable string we use to give names to computers, such as www.yahoo.com. It's the translation between the names and the addresses that's the tricky part.

In the RFC there's a famous line:

"A **name** indicates what we seek. An **address** indicates where it is."

Let's imagine a user clicking on a link in her browser. The URL associated with the (transitive) hypertext link our user clicked on contains a domain name, in this case anysite.com (along with other information about the requested resource). We'll need the IP address (location) for anysite.com (what we seek) before we can send any request.

## How DNS Works

DNS doesn't use TCP/IP to send lookup requests. Instead it uses UDP, an alternative way of sending packets on the Internet. UDP has its own set of characteristics regarding performance and reliability, but for our purposes here we'll focus almost entirely on TCP/IP. DNS is (almost) the only exception.

The sequence diagram below (Figure 3-1) shows the steps involved in the DNS lookup and setting up the TCP connection, in the simplest case.

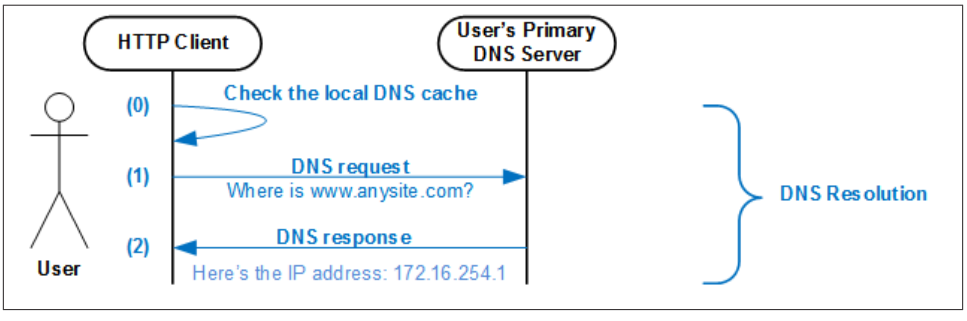


*Figure 3-1. A simple DNS query sequence.*

The first step (0) in looking up ('resolving') an IP address is to check the local machine's DNS cache. All modern operating systems (including mobile ones) perform some type of host-level DNS caching, storing the IP addresses for the most commonly accessed domain names locally at the operating system level, for a certain period of time, so no external DNS query need be made. This has a huge impact when you are loading many pages with URLs that share a domain name, but it doesn't entirely eliminate the need for DNS lookups. In addition to the DNS caching that happens at the host level, almost every HTTP client provides DNS caching services as well. Some browsers may provide advanced DNS services such as *speculative lookups* or *precaching* of DNS queries.

Many different approaches have been tried to reduce the amount of time spent on DNS resolution, often trying to resolve DNS queries in advance, or to parallelize them. These optimizations indicate the importance of DNS in the request/response cycle.

If the browser finds the IP address for a request in its cache, it will immediately go on to the next step. If it does not find it there, it will attempt to make a DNS query. Usually this triggers another lookup on the local machine, this time in the network-level DNS cache. If the IP address is cached locally then DNS response times are minimal, often 5ms or less.

Let's take a look at the case where the IP address isn't cached locally. From the sequence diagram in Figure 1, the client (a desktop browser in this case) makes a DNS request (using UDP) to its primary DNS server (1-2). This is the primary 'phone book' assigned to this machine for looking up IP addresses. It's usually the closest DNS resolver (host server) operated by the user's ISP but it could be your own or your company's DNS server, or a public DNS system. Some organizations, such as OpenDNS and Google, provide public DNS services for free as an alternative to commercial providers.

DNS is a hierarchical addressing system; if the primary DNS server does not have the current IP address that you requested on hand, a secondary request can then be made all the way up through a hierarchy ending in one of the 14 *root servers* for the DNS system (steps 2-7 in Figure 3-2). These aren't individual hardware nodes but are complex groups of servers distributed all over the world.

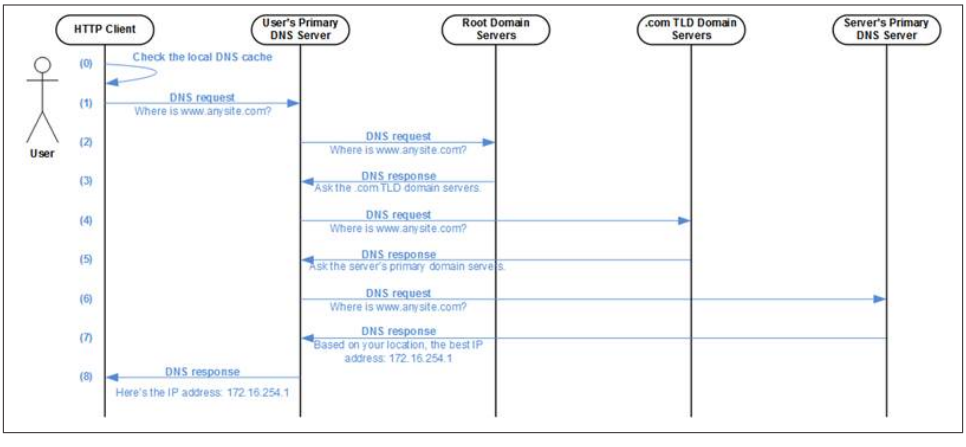


*Figure 3-2. An authoritative DNS request with full recursion.*

In cases where the IP address isn't cached nearby, extra response time will be spent doing a recursive 'authoritative' lookup as we can see in Figure 3-2. However this will affect only one user, since the IP address will be cached for everyone else using that primary DNS server in the near future. DNS servers generally use frequency-based caching, meaning that the more frequently requested IP address are also most likely to be cached. Domain names for hosts accessed only rarely may see extra DNS response times due to recursive lookups.

These authoritative lookups are made by your primary DNS server, not by your client. In the user narrative, authoritative lookups appear just as in Figure 1, they just take longer while your DNS resolver is getting an answer on your behalf.

## DNS and Performance

Like every other aspect of the network, the user's location – really their distance from the responding server, whether DNS or HTTP, is a key variable in response times experienced in the real world. For a given user in a given network context, the IP address will be a constant value in most cases. However, different users may receive different results for DNS lookups for a number of reasons, including their geographic or network location, server load, or other factors. Large-scale websites may change IP addresses fairly often, making things yet more complicated.

Figure 3-3 shows some example DNS response times for both commercial and free DNS providers in North America for the domains associated with the top 250 websites as ranked on alexa.com. These are only intended as representative numbers; now that we understand how DNS works we know that users themselves don't query these servers, and have no obvious way even of knowing which DNS provider a particular Website uses. It's all down to the user's ISP.

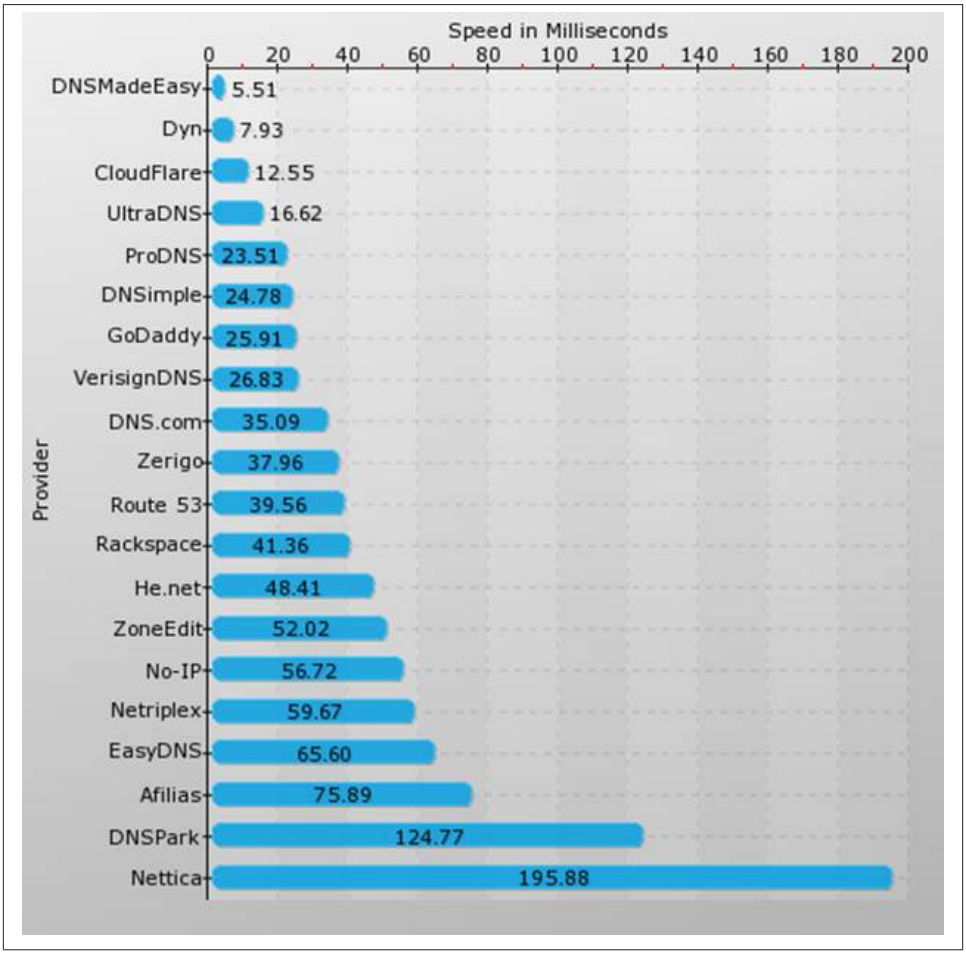


*Figure 3-3. Average DNS response times in North America. (Source: solvedns.com)*

Once a DNS lookup has been performed it's now in the local cache and future lookups will avoid DNS costs altogether, until the cached DNS record expires, then the whole process will be need to be repeated. This limits the impact of DNS lookups on the overall response time. In general, we require one DNS lookup per unique hostname on the

page (two each for TLS connections), plus one for the initial URL. The number of unique hostnames, called the *host factor*, is a key value for Web performance, which will appear many times throughout the book.

## DNS and Global Server Load Balancing

At the network level there are three main techniques for reducing response times:

1) Reduce the amount of data *packets* going over the wire

2) Reduce the number of *messages* exchanged between the client and server

3) Move the content to a server closer to the client

Almost all of the optimizations described in this chapter, and indeed throughout this book, will amount to implementing one or more of these techniques.

So what does this have to do with DNS? One of the key techniques for reducing overall response times is to reduce the physical distance between the user and the responding server. The RTT (round-trip time, the time it takes one packet to be sent and another to return) between client andserver is basically irreducible by any other means. In practice this means that users far from the server are at a great disadvantage, performance-wise, and your site may be nearly unusable if the delays are too long. If you have users distributed all over the World, you're going to need to have your content distributed as well, in order to have reasonable response times.

Typically, sites with globally-distributed users will utilize the DNS system to direct their users to the nearest server; users in Asia are given the IP address of a relatively nearby data center, while users in the EU are sent, again via DNS, to another location closer to them. This is called *Global Server Load Balancing (GSLB),* and while it's not often mentioned in this context, it's the number one means of reducing response times on the Web. It does require some time and effort on your part to implement, or at least in choosing a good DNS provider to handle this for you.

Nothing you can do to optimize your site will be as important as this one; other optimizations are secondary to reducing the RTT. Unfortunately, this can be very hard to do, and costly as well.

It's interesting to note, from a systems thinking point of view, the DNS system is highly reliable without being predictable, in the sense that I can't predict the IP address associated with a domain name more than a short time ahead, but the system still functions correctly. The DNS system, based on simple flat files, was the first large-scale 'NoSQL' data system in production.

Another important note about DNS response times (and network performance in general) is called out in the Google Public DNS FAQ:

*"…In addition to the ping [network latency] time, you also need to consider the average time to resolve a name. For example, if your ISP has a ping time of 20 ms, but a mean name resolution time of 500 ms, the overall average response time is 520ms. If Google Public DNS has a ping time of 300 ms, but resolves many names in 1 ms, the overall average response time is 301ms.*

This points out not only the importance of the breakdown of the overall E2E but also the client think time and the server duration. For our purposes we are concerned about DNS response times and how they affect the user experience. DNS typically only takes a few hundred milliseconds at most, so why should we care? One good reason is that the end user is very sensitive to DNS lookup times. This is because the DNS lookup has to occur before any other steps can be taken, and because user agents may appear to be non-responsive during the lookup period TDNS. In cases where the DNS lookup takes an unusually long time (more than say, 300ms) users will often abandon their request. DNS response times usually make up only a small part of the overall E2E response time, but they have an outsized impact on users' patience!

Mobile device users will experience longer DNS resolution times as well, approximately twice as long roughly on average.

## Hands-on: The User Impact of DNS

### Tools Required

To really understand the impact of DNS on the end user, I've found the best exercise is to simply demonstrate what the Web would be like without any DNS at all. We'll do this by modifying a system file stored on your computer called *hosts*. This exercise can be done by Windows and Mac users as well as UNIX users. (The hosts file exists on most mobile phone operating systems as well, but it can't be easily edited.) This is a temporary test; you'll want to reverse these changes later. No computers will be harmed in this experiment! As always, be careful when modifying server files; make backup copies of all the files you change.

### Performing the Test

The hosts file is used by the system as an internal DNS phone book – it stores domain names and IP addresses locally so no DNS call ever need be made. This reduces $\mathbf{T}_{\mathbf{DNS}}$ to the minimum value. We're going to add the domain names and IP addresses for three of your favorite sites to your hosts file. Then, when you visit those sites, you'll be able to perceive the difference easily.

On UNIX and Mac machines, the file location is /etc/hosts. On Windows systems, it's in Windows\System32\Drivers\etc\hosts. The file is very simple; a set of domain names and IP addresses. Here's what mine looked like when I set this up:

```
# sample etc/hosts file

www.yahoo.co.uk 68.180.206.184

www.google.com 74.125.239.148

www.wikipedia.org 198.35.26.96
```

The domain names and IP addresses have to be separated by at least one space, and comments start with a '#' symbol. Of course, due to GSLB and other reasons, the IP addresses in your file won't be the same as in mine – please don't use these numbers. So where can you find these IP addresses for your hosts file? There are a number of ways but here we'll use a common command-line tool called dig. Dig is a DNS analysis tool with a lot of options and features. Here we'll just ask it for the IP address:

```
$ dig www.yahoo.com +short | tail -n1

206.190.36.105
```

This tells dig to perform a DNS lookup on www.yahoo.com and write a short version of the response. (The tail command is just for formatting.) You can use this method to get the IP addresses for your hosts file.

## What We Learned

Once you've edited the hosts file, it will immediately take effect. Try surfing to the sites you've added and see if you notice a difference in response time. (This might be very useful for the IP address of your browser's home page.)

Once you've convinced yourself that DNS matters, a lot, you can revert these changes and restore the hosts file to its original state. Why not just leave the hostnames and IP addresses in the file? Because commercial IP addresses change frequently. The ones in your hosts file will go stale and you'll start to see errors. Be sure and restore the hosts file to its original state.

You can also use dig to test DNS response times. Here's the command:

```
$ dig www.yahoo.com |grep "Query\ time" | cut -d" " -f2,3,4,5

Query time: 11 msec
```

This is about as fast as we can expect a DNS lookup to go. However, if we do the same test with a domain name that is unlikely to be locally cached, we can immediately see the impact:

```
$ dig www.yahoo.ru |grep "Query\ time" | cut -d" " -f2,3,4,5

Query time: 323 msec
```

This probably indicates that looking up the IP address required an authoritative lookup, which required an order of magnitude longer to resolve.

## Finding and Solving DNS Performance Problems

DNS performance problems are the responsibility of the server's DNS provider, or the user's. If you suspect you might have DNS problems, you'll have to go through the problem solving process to identify and fix the problem. First you'll need to use the MPPC model to identify the problem – this involves gathering response time data for your domain(s), perform a simple T-time analysis to find T1-4. If T1 is large compared to the overall E2E, then you'll want to drill down to determine if the problem lies with DNS, TCP, or (optionally) TLS response times.

Once you've narrowed down the problem area to one of T-time segments you'll know where to focus your time and resources for improving E2E response times, which is the whole point.

If you've determined that your problem lies with $\mathbf{T}_{\mathbf{DNS}}$, then you'll want to determine which users are affected; all of them, or only those in a specific region or using a specific connection type or ISP? If only some users are affected and others are not, the problem is very likely the ISP's DNS services to the local user, rather than your own configuration.

There are cases where DNS records that you provide to your DNS provider (or create yourself if you run your own servers) may have errors in them that can cause long DNS response times. These are relatively rare and will be easy to find.

Problems with DNS resolution by the user's ISP can be very difficult to solve. Savvy users can change their DNS resolver to a free service such as Google Public DNS, but ordinary users aren't likely to be able to do this (and shouldn't have to).

Mobile devices typically use the DNS service provided by their mobile provider, and have few options to change this. The good news is that DNS resolution times have gotten a lot better over time. Choosing the right DNS provider is the best way to avoid DNS problems before they occur.

## Optimizing Your DNS Response Times

Unless you run your own DNS servers, the options for optimization all boil down to working with your DNS provider to make sure that there are no problems. There's little that can be done directly to optimize DNS beyond fixing problems; generally only one RTT is required, and the server involved (the user's ISP's DNS resolver) is not under your control. The graph in Figure 3 only indicates the response times from the server's DNS provider, not the client's. Once you've optimized your DNS response times to values typical of the results in Figure 3, you can probably accept it as tolerable and go on to focus on the rest of the E2E.

# SYN SYN ACK - Setting Up a TCP Connection

Setting up a TCP connection is a very simple process compared to resolving an IP address or establishing a secure connection using TLS. The TCP connection sequence involves the exchange of only 4 TCP packets, requiring only a single RTT as shown below. This famous sequence is sometimes called 'SYN SYN ACK" by network engineers (and savvy performance experts) because those are the types of the packets exchanged as shown in Figure 3-4.

First, the client sends a SYN packet asking the server to make a connection by synchronizing packet sequence numbers (3). The server responds with another SYN + ACK message indicating the beginning of the sequence (4). Then the client then sends a final ACK packet to let the server know all's well (5).

Once this is completed, the next step is for the client to either immediately send the HTTP request or, if needed, first establish a secure session.

Again, it should be noted that from the user narrative point of view, we can only know the time when the first SYN packet was sent, and when the corresponding SYN + ACK from the server was received. We cannot know the how long the final ACK took to reach the server or even if it arrived. The client automatically assumes that the connection is good as soon as the server's SYN arrives and sends the request. There are some things that can't be seen from the user's point of view. Fortunately for our model, none of these things make a material difference in user's response times.

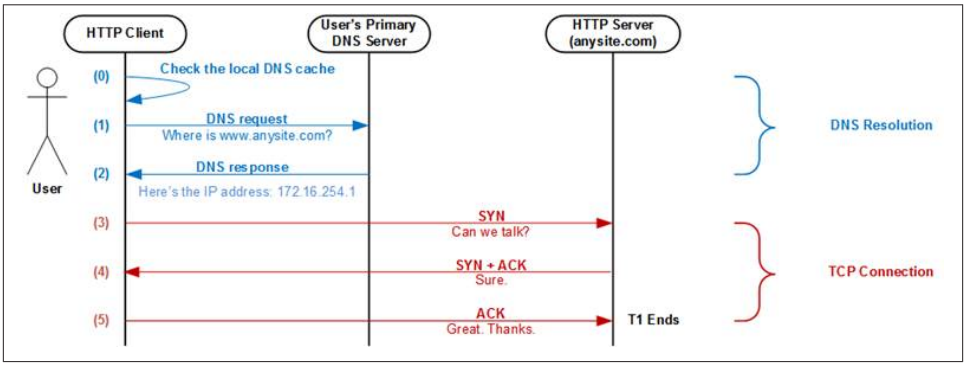


*Figure 3-4. Setting Up a TCP Connection*

## TCP Connections and Response Times

The time to set up a TCP connection should be small relative to the overall E2E. Creating and destroying TCP connections is a costly operation, and the rule is to do it as infrequently as possible. Reducing the overall costs associated with TCP connection times

is one of the problems that advanced protocols like *SPDY* and *HTTP/2* are attempting to solve.

In the last chapter we talked about the multi-step query pattern for HTTP requests and responses. The initial step, obtaining the base HTML payload, is blocked while the initial connection to the server is established, just as it is for the DNS lookup. In the second step this rarely matters from a practical point of view, since the connections for the intransitive page requisites are usually established using (multiple parallel) persistent connections.

Table 3-1 shows some values for comparing TCP connection times on the Web.

*Table 3-1. Representative TCP connection times*

| Connection | RTT (ms) | TCP connection time |
|---|---|---|
| Local direct cable | 15 | 22.5 |
| US or EU broadband | 80 | 120 |
| Inter-continental | 150 | 225 |
| Satellite | 500 | 750 |
| LTA Mobile | 400 | 650 |

Even rough, back-of-the-envelope figures such as these illustrate the impact of location and connection type on Web performance.

## Finding and Solving TCP Connection problems

Diagnosing TCP connection problems usually involves determining if the problem is with the value of RTT or instead with the server's response time for connection requests.

TCP connection times directly depend on the user's location, like everything else on the network. But since only single packets are exchanged, the actual time taken to send the SYN SYN ACK messages back and forth is minimal, and can't usually be made much faster. We also usually assume that the value of the RTT for a given connection is constant throughout its lifecycle. From the user narrative point of view, the TCP connection cycle takes one RTT, although some references (incorrectly) use 1.5 RTT.

The other main factor that can cause poor TCP connection times is the load on the HTTP server's host (or possibly on any load balancers, proxies or other intermediaries that lie between, but let's keep things simple for the moment).

Almost every site you visit will be *load-balanced* in some way, meaning that www.google.com will actually resolve to an IP address of a piece of networking hardware whose job is to distribute the incoming traffic across many individual machines. It's the load balancer's job to act as a kind of 'traffic cop', directing the incoming requests so that the traffic is evenly distributed, or at least that no single machine is completely busy while another sits idle. If you're experiencing unusually long TCP connection times, it's pos-

sible that this load balancing mechanism isn't working. (Here we're talking about *local* load-balancing across multiple servers located in the same physical location, rather than GSLB services provided through DNS.)

If the HTTP server machine is heavily loaded, meaning that it's serving many concurrent TCP connections, TCP connection response times are likely to increase. Basically the time between the first two SYN packets will increase as the server takes extra time to respond to the client's request for another additional connection. There are of course other reasons for the machine to be busy than having too many TCP connections. Still, it's probably the first thing you want to look for if you are experiencing long TCP response times.

Packets can also get lost on the network, and a certain amount of stochastic packet loss is normal. TCP has built-in mechanisms for making sure that lost packets are re-sent if they do get lost (this is called 'guaranteed delivery'). However, if the lost packets are either of the two initial SYN packets that make up part of the TCP connection process, then this mechanism won't work and the connection request will either not be received by the server or never seen to be acknowledged by the server, and the connection will fail, probably in a hard-to-detect way.

Packet loss is an important factor in E2E response times, especially on some kinds of mobile connections. The TCP protocol's built-in congestion-control mechanism limits how much data can be transmitted based on packet loss in the network.

In any event, typical TCP connection times are usually reasonably stable over time, and are not often a source of much trouble. It's usually not hard to diagnose the problem once you recognize it. But fixing the problem may incur high costs in both money and time if it involves adding capacity, and will only work up to a point.

## Optimizing TCP Connection Times

If your TCP connection times are within your own levels of tolerance, perhaps because it's a small part of the T1 time and even less of the E2E, you may choose not to spend your time and resources optimizing for it. But if you do choose to improve your response times, you'll need to go through the same diagnostic process of measuring your RTT as outlined earlier for DNS problems, and determine where the problem lies.

If your problem lies with the overall RTT, you'll want to break the problem down by location and connection. Does the problem only occur for users in specific parts of the world, or to users on specific devices or in specific situations or times of day?

Reducing RTT times is difficult and usually involves moving the servers closer to the users via GSLB. (Moving the users closer to the servers is an often-suggested joke in the industry, but isn't really practical.)

In cases where the RTT seems reasonable but the connection times are still unusually long, the cause is very likely server delay and increasing server capacity is probably the best method of addressing the issue.

### Secular Cycles and Response Times

One commonly seen performance effect involves *secular cycles* in response times – cycles that vary according to the time of day and the day of the week, and sometimes also for monthly and yearly cycles. Typically we see longer response times in a particular area at midday and shorter ones at midnight. This cycle is present across all time segments of the E2E (indicating it's a property of the network) and is simply due to more people using the network during those times. Just as we see peaks in highway traffic during the morning commute, we see network congestion at time of greatest use, and for exactly the same reason.

One strategy for improving the overall response times for your site is to measure and identify these secular cycles and find ways to increase server capacity at peak congestion times. This won't affect network response times directly; but by adding capacity you can at last make sure that your servers aren't adding to the delays users experience once they finally get to your site. It's the equivalent of making sure that users have plenty of parking once they get off the highway.

Knowing about the secular cycles that affect your user's response times is an important aspect of performance management and capacity planning.

## TLS, SSL, and Secure Connections

The TCP/IP protocols weren't designed with explicit encryption-based security in mind; security was intended to be security of access rather than of data. It's a straightforward process to 'sniff' TCP packets and read their contents on the wire. The majority of traffic on the Internet is sent 'in the clear' and can be read by anyone who cares to take the time and trouble to do so.

This is changing however, as Web users become more security and privacy-conscious. Many large Web sites such as Google and Twitter have moved toward requiring secure connections, and the forthcoming HTTP/2 standard will require that all connections be secure. This change will have a significant impact on performance, regardless of its impact on security and privacy.

When data needs to be secure, we use an additional layer of transport-level encryption called the '*Transport Layer Security'* protocol or TLS. TLS uses a public-key encryption mechanism to establish the validity of a X.509 certificate, providing assurance that the certificate holder is who they claim to be. While both client and server can be authenticated, it's usually the case that the server authenticates itself to the user agent, so that users can be sure that they are really shopping at Macy's and not badguy.com.

Client authentication is usually used in cases where access to the server is restricted for information security purposes, for example corporate firewalls or proxy servers reserved for emergency services providers. For this discussion we'll assume it's the server that's presenting a certificate and being authenticated.

Security in this context means addresses three important aspects of the communication between client and server:

· *Authentication* – by examining the server's response and verifying its certificate, we can be reasonably sure that the site we are visiting isn't misrepresenting itself.

· *Confidentiality* – encrypting the data sent over the wire prevents it from being observed during the transport process.

· *Integrity* – because the entire session is encrypted, we can be sure that no one has tampered with the message contents. This is why we don't just encrypt passwords and other 'sensitive' data.

TLS is complex and presents a number of performance challenges for users. As the saying goes, 'nobody ever made anything faster by adding another layer to the architecture'. Understanding the impact of TLS on performance is easier once we recognize that it falls naturally into two parts, the initial TLS 'handshake' required to establish the secure connection, and the subsequent secure date transport mechanism itself. Conveniently, this not only makes good sense in the MPPC model, but also follows the natural division of the TLS process itself into two parts. While the initial handshake uses an (asymmetric) public key exchange to authenticate, once the server's certificate has been validated, a symmetric key is then used for all further communication. Since asymmetric public-key encryption is often considerably slower than symmetric encryption, the performance characteristics of the TLS handshake (part of T1) will likely be (very) different than the encrypted bulk data transport, which is part of the T3 time segment.

## TLS, SSL, and All That

First we should try to clear up some terminology around encryption standards for the Web. *Secure Socket Layer* (SSL) is the name of the original technology, developed by Netscape Communications in 1994. As SSL became more widely adopted, especially for e-commerce, the Internet Engineering Task Force (IETF), one of the primary standards organizations for Internet technologies, began the standardization process for SSL, and along the way the formal standard was renamed *Transport Layer Security* or TLS. This acknowledged the contributions of the rest of the community beyond Netscape and also the changes that occurred during the standards process. TLS is defined in RFC 5246.

Today the TLS standard is on version 1.2. The first version of TLS and the last version of SSL (3.0) were nearly the same; the standard has moved on since then, improving overall security and fixing exploits that have come to light in the meantime.

When the browser and server begin the TLS handshake process, they have to identify the version of TLS that they are using. Many servers still support "TLSv1/SSLv3', which is the first version, although you should upgrade for security reasons.

For the purposes of this book, I'm going to use 'TLS' to refer to this family of technologies, for purposes of communication. When I discuss a specific version of the standard, I'll call that out directly. Still, there are numberous cases where SSL is still used (for example for the "OpenSSL" software package) and these terms are often used interchangeably.

## How TLS Works

In this section I want to describe the basic steps involved in setting up an TLS connection. The diagram on the next page (Figure 3-5) shows the blow-by-blow sequence of events.

### HTTP vs. HTTPS

The first question to answer is how the user's HTTP client knows that this is a secure connection, that is, how does the client know to initiate a secure session before sending the HTTP request? This is signaled to the client via the *protocol scheme* used by the URL, the address of the thing the user seeks. Secure URLs use the *https* scheme, instead of *http*, which is used for unsecure connections. So a secure connection to Google would use the URL:

*https://www.google.com/*

And an unsecured connection would use:

*http://www.google.com/*

The https part of the URL tells the client to use TLS to secure this connection prior to making the HTTP request.

### The Basic TLS Handshake

Once the TCP handshake is completed, and the client has sent the final ACK to the server, it immediately initiates a secure session with the server. It's important to note that TLS, unlike TCP/IP, is a session-based protocol, meaning that it has a well-defined beginning and end, and that the state of the session is managed for the entire duration.

Shown below is the simplest possible version of this exchange, and it rarely happens this way for public sites. I've left out the entire certificate revocation check process until the next section. For now, keep in mind that this is still an idealized version of events. What really happens takes longer and costs more, as we'll see.

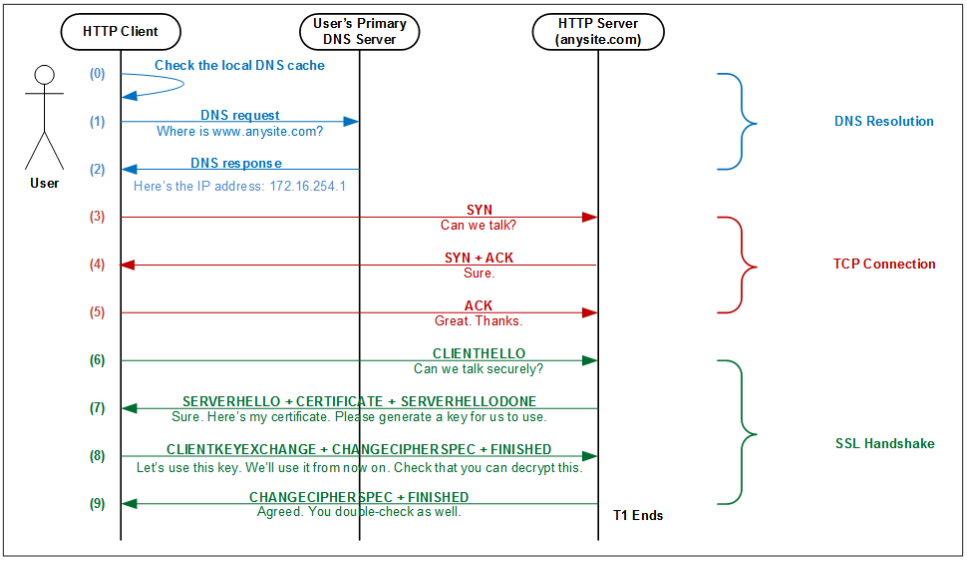


*Figure 3-5. Basic TLS handshake message exchange (without certificate verification)*

In this description, (Figure 3-5) we'll assume the server will authenticate itself to the client, offering reassurance that the server is truly responding on behalf of the organization indicated in the target URL. There are four basic steps in the process:

1. Protocol Negotiation

   The client sends a CLIENTHELLO message to the server, including a list of supported protocol versions and encryption methods (ciphersuites) (6). The server responds with a SERVERHELLO response indicating the server's choice of protocol and ciphersuite (7). The server will choose the highest common protocol version (TLS 1.2 over 1.1 say). At this point both server and client have agreed on the protocol and ciphersuite.

   A *ciphersuite* in TLS terminology indicates a combination of three cryptographic algorithms that will be used in the course of the TLS session. The first algorithm is an asymmetric encryption algorithm that will be used for authentication during the initial handshake. The second is the symmetric key encryption method that will be used for the bulk data transport. The third algorithm specifies the message digest format, which is used to ensure the integrity of the messages.

   Here's an example of a ciphersuite supported by many servers and clients:

   ```
   ECDHE-RSA-AES256-SHA
   ```

   In this case, the asymmetric algorithm being used to secure the initial connection is the "Elliptic Curve Diffie-Hellman" algorithm, using keys signed by RSA, a well-known security company. The AES 256-bit symmetric encryption algorithm will

then be used for bulk transport of the data. Later, when the client performs an integrity check on the incoming messages, the SHA checksum algorithm will be used.

All of these issues have performance implications for TLS connections. Elliptic curve algorithms are typically faster than other algorithms, for example. Some bulk encryption algorithms such as RC4, can perform streaming operations, which can improve overall throughput, but may also incur additional security risk or be less secure (require less effort and time to break) than others.

2. Certificate Exchange

   The server immediately follows the SERVERHELLO message with a CERTIFICATE message that contains the server's certificate information (in binary DER format). (If both client and server are authenticating, the server will send a CLIENTCERTIFICATEREQUEST message as well.) Then the server, having offered its credentials to the client, sends a SERVERHELLODONE message (7).

   At this point in the TLS handshake, we have simply exchanged (unencrypted) TCP messages, and relatively simple ones at that. The response time of this part of the process is likely to be little more than the time required for the messages to traverse the network, and the only message of significant size is the CERTIFICATE message.

3. Certificate Verification

   Once the client receives the server's certificate, it has to take a number of steps to ensure that the certificate is valid, a process called *verification*. Verification involves checking the certificate's digital signature, its IP address and URL, its activation and expiration dates, and most importantly for our purposes, the revocation status. Revocation checking is commonly done using two methods, OCSP and CRLs. Both of these can be time-consuming processes and we'll have a lot more to say about certification verification in the next section.

   In this basic handshake we'll omit the certificate revocation steps and go on to the next step in the process. Certificate revocation checking isn't always required, and sites with low or medium security may choose to use certificates that don't require these steps. This is becoming less frequent however, as security levels increase across the Web.

4. Client Key Exchange and Confirmation

   Finally, provided that all's well with the certificate's verification status, the client is satisfied concerning the server's identity. It's the client's job now to create a *pre-master secret* for this session. This secret will only be used for this session, and is essentially a random number generated on the client. The client then extracts the server's public key from the certificate sent earlier and uses it to encrypt the pre-master secret, so that only the server can decrypt it. Once this CLIENTKEYEXCHANGE message is sent, the client immediately follows up with another message,

CHANGECIPHERSPEC, telling the server that all future messages will now be encrypted using the 'master key' that can be derived algorithmically from the pre-master secret (8). The server will generate this same master key, using the pre-master secret as input, in exactly the same way. The goal of this entire process is to generate a master key that only both sides know. There's considerable client think time and server duration required on both sides in this step.

The client will immediately send a FINISHED message to the server, encrypted with the master key derived as described above, now switching the symmetric cipher chosen in step 1 (8). This message is composed of a digest of all the previous messages concatenated together and then encrypted using the agreed-upon symmetric algorithm. The point here is that if the server is able to decrypt the FINISHED message (and compare the results with its own internal records) successfully, then it will be able to decrypt the HTTP data as well.

The server now performs the same steps in reverse; it sends a CHANGECIPHERSPEC message to the client to let it know all future discussion will be encrypted in the agreed way, and then sends a FINISHED message based on all previous messages (9). If the client can decrypt this message successfully, the TLS handshake is now complete, and all future communication in this session will be encrypted as negotiated. Receipt of the server's FINISHED message also implicitly indicates that the server successfully read the client's FINISHED message as well.

So far, looking at Figure 3-5 we see that the entire T1 sequence has required only four roundtrips, one for the DNS lookup, one for the TCP connection, and two for the TLS handshake. Again, a lot of work needs to be done to establish an TLS connection. Now let's look at what happens in the real world.

### Hands-On: The TLS Handshake in Action

In this hands-on exercise we're going to try to get a concrete feel for what happens during the TLS handshake, by watching it happen in real-time on the screen. Actually, it happens pretty quickly and you might have to scroll! Performing this simple test is often an eye-opener for people understanding TLS for the first time.

## Task

We'll use cURL to reveal the steps involved in establishing a connection to Facebook. This is very simple, here's the command and the results I got performing the test:

```
$ curl -v -o /dev/null https://www.facebook.com/

Hostname was NOT found in DNS cache
*   Trying 31.13.75.1...
* STATE: CONNECT => WAITCONNECT handle 0x8001f188; line 1076 (connection #0)
 Connected to www.facebook.com (31.13.75.1) port 443 (#0)
* successfully set certificate verify locations:
```

```
*   CAfile: /usr/ssl/certs/ca-bundle.crt
  CApath: none
* SSLv3, TLS handshake, Client hello (1):
} [data not shown]
* STATE: WAITCONNECT => PROTOCONNECT handle 0x8001f188; line 1189 (connection #0)
* SSLv3, TLS handshake, Server hello (2):
{ [data not shown]
* SSLv3, TLS handshake, CERT (11):
{ [data not shown]
* SSLv3, TLS handshake, Server key exchange (12):
{ [data not shown]
* SSLv3, TLS handshake, Server finished (14):
{ [data not shown]
* SSLv3, TLS handshake, Client key exchange (16):
} [data not shown]
* SSLv3, TLS change cipher, Client hello (1):
} [data not shown]
* SSLv3, TLS handshake, Finished (20):
} [data not shown]
* SSLv3, TLS change cipher, Client hello (1):
{ [data not shown]
* SSLv3, TLS handshake, Finished (20):
{ [data not shown]
* SSL connection using TLSv1.2 / ECDHE-ECDSA-AES128-GCM-SHA256
* Server certificate:
*        subject: C=US; ST=CA; L=Menlo Park; O=Facebook, Inc.; CN=*.facebook.com
*        start date: 2013-09-20 00:00:00 GMT
*        expire date: 2014-09-25 12:00:00 GMT
*        subjectAltName: www.facebook.com matched
*        issuer: C=US; O=DigiCert Inc; OU=www.digicert.com; CN=DigiCert High Assurance CA-3
*        SSL certificate verify ok.
* STATE: PROTOCONNECT => DO handle 0x8001f188; line 1208 (connection #0)
```

This command tells cURL to output verbose troubleshooting information to the screen.

## What We Learned

Looking at the output from cURL, you can see all of the steps involved in T1, from the DNS lookup, to the TCP connection, and then the sequence of messages required to establish the TLS connection. (Notice that cURL still uses SSL instead of TLS in many places. The connection actually used TLS 1.2.) The actual certificate presented to us by Facebook is shown, followed by a verification step described next. Finally, the connection is established! The top-level certificates for the CA's known to cURL are in the file /usr/ssl/certs/ca-bundle.crt.

Q: Does seeing the TLS handshake happen increase your intuition about how TLS works under the covers?

Q: In the handhake above, cURL doesn't show the associated response times. Do you think establishing a TLS connection is slow? Fast? Why?

Q: If you are in Norway, do you think you will get the same certificate? Will the process follow the same steps?

Q: What could cuase a TLS handshake to be slower than normal? What might be an approach to resolving the issue?

#### TLS: Certificates & Verification

Adding the final step in the TLS handshake process to our sequence diagram requires some explanation of how TLS certificate verification works.

## Certificate Verification

Verifying that the certificate sent by the server requires multiple steps on the client. First the client will check the expiration date on the server's certificate, and the URL and IP address of the connection to make sure everything looks good. Then the server's public key is checked to make sure it's the right key for this organization. Once all of this checks out, the client must now determine if the server's certificate has been revoked by the certificate authority (CA) who issued it.

TLS uses *public-key* encryption for its initial handshake, using the x.509 certificate standard format. This requires a large-scale *public-key infrastructure* (PKI) system to support it. Certificate Authorities are needed to issue certificates, sign them, and in some cases revoke them. They also are required to run a certificate revocation system that allows the client to determine if a certificate is valid or not.

TLS certificates participate in a chain of trust, and each certificate in the chain must be signed by yet another higher-level certificate, until we reach the CA's own root certificate, which terminates the chain.

During the verification process, each certificate in the chain must be verified, with the exception of the root certificate. The root certificates for each CA must be stored in the browser's trusted store, in order to complete the chain as we saw earlier.

In the end, you have to trust someone. In this case, the CA itself provides the assurance that the certificate is valid, as shown in Figure 3-6.

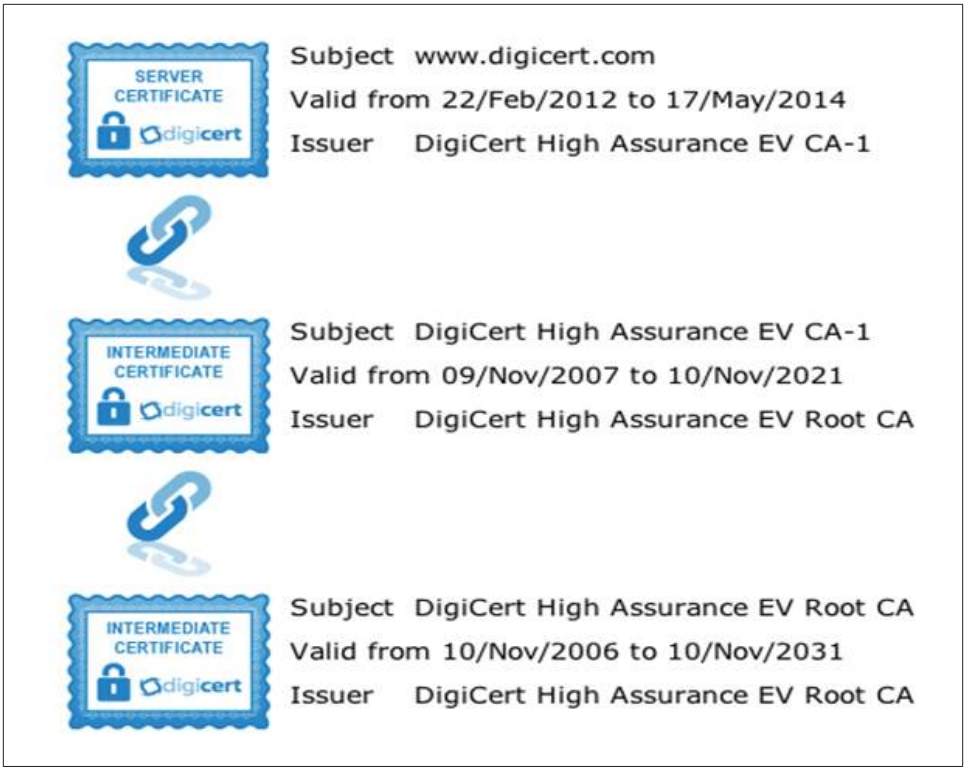


*Figure 3-6. The chain of trust for an TLS session (from www.digicert.com)*

How do you decide who to trust? And who makes that decision? The *CA/Browser Forum*, an organization made up of certificate authorities and browser manufacturers, makes these decisions based on consensus within the industry. The CA/Browser forum doesn't make rules about how TLS itself works (IETF does), but it does make decisions about how HTTP clients handle secure connections, including rules about what kinds of certificates are acceptable, and how browsers display the security levels of different kinds of secure connections to end users.

The CERTIFICATE message sent in the TLS handshake will contain at least one, and usually at least two, certificates that need to be verified. In many cases the verification steps need to be performed serially, one after another, rather than in parallel.

Clearly the length of the certificate chain for your site will have an impact on the handshake time, with each certificate in the chain adding some additional time to the overall T1. Minimizing the certificate chain for your TLS certificates is a good way to reduce your handshake times. The minimum length of the certificate chain is two, for various reasons, and in most cases, good certificate management practices can keep the number

of intermediate certificates down to this level. If your site's certificate chain is longer than two, this may offer you a relatively cheap optimization opportunity.

## Key Length, Security, & Performance

The process of encrypting and decrypting the TLS handshake messages depends to some extent on the relative *encryption strength* of the encryption used, both in terms of the cipher chosen for the work but also in terms of the length of the key being used. A message encrypted with a longer key is harder to break; using a longer key can offer higher levels of security. It also takes longer to encrypt data using a longer key (and remember that this is an asymmetric key). TLS key lengths are typically given in powers of two, e.g. 128-bit encryption vs. 256-bit.

Until recently, most sites were using 1024-bit keys for TLS. However, the ongoing arms race between computing power and encryption strength means that it's getting considerably easier to break 1024-bit encryption. For this reason, the National Institute of Standards (NIST) has recommended that no new 1024-bit certificates be issued after Dec 31, 2013.

NIST itself cannot mandate this change. However, the CA/Browser Forum has mandated that the NIST guidelines be implemented in this case. This isn't unusual; in the past, shorter key lengths, such as 576- and 768-bit, have been deprecated in this fashion. Eventually, the current 2048-bit guideline will have to be changed to 4096-bit, or some other means will need to be found. Many large commercial sites are already using 2048-bit encryption, especially financial institutions.

*(Note that the correlation between 'strength' of encryption and key length is weak at best; the most you can say is that for a specific encryption session, more resources will be required to decrypt a message originally encrypted with a longer key. Different key lengths can provide the same level of security when used in a symmetric vs. asymmetric session as well. RSA claims on their website that a 1024-bit asymmetric key is only as strong as an 80-bit symmetric one (using the same algorithm and block size).)*

Performance-wise, using TLS certificates with 2048-bit key lengths imposes some response time costs. Typically, a 2048-bit RSA key will take 4x to 8x the amount of processor operations that a 1024-bit key will require on the same machine. This doesn't mean that it will take 4x as long to process! The relationship between the server's response time and the symmetric key generation isn't linear; as we'll see below, sites using 2048-bit keys generally have longer response times for TLS handshakes, by a factor of 1.5 or 2 at most.

## Extended Validation

The CA/Browser Forum has also defined how user agents display information about connections to secure sites. In particular, the Forum has created the concept of Extended

Validation certificates. In order to obtain an EV certificate, the organization that will present the certificate is required to undergo some level of scrutiny regarding good business practices, and have a verifiable public presence. In addition, an EV certificate requires certificate verification at connection time and only sites with EV certificates will show a 'green bar' visual effect in your browser. TLS connections that do not use EV certificates are indicated with a yellow 'lock' symbol, but no green bar. EV certificates are (in 2014) always 2048 bits or more in length, and certificate authorities have phased out issuing EV certificates with smaller key lengths.

The purpose of Extended Validation is to give the user confidence that they are connecting to a legitimate server. The EV constraints themselves do very little to increase security however. It's also important to realize that this behavior is user agent-specific, and is basically targeted at the desktop user with a traditional Web browser. It's not always clear in other scenarios, such as mobile devices. Also, many user agents are not 'browsers' and don't display a location bar. API calls and other intransitive and *headless* HTTPS requests will not display the green bar either.

#### Revocation Checks: OCSP and CRL

There are basically two methods used to perform certificate revocation checks, called *Online Certificate Status Protocol* (OCSP) and *Certificate Revocation List protocol* (CRL). The CRL method is often the faster method overall, but has some security-related problems compared to OCSP.

The CA/Browser Forum guidelines for TLS certificates do not require that certificate revocation checking be performed in all cases. However, EV certificates do require revocation checks at runtime. It's also an important security practice. Organizations should not accept any TLS certificate without verification, unless they generate the certificate themselves.

Support for certificate revocation checks is universally supported in all modern browsers, although how revocation checks are done has varied over time. Different versions of on popular browser, for instance, have turned off revocation checking altogether, reverted to using CRLs, and then phased out CRLs in favor of OCSP, all in a short period of time.

How the revocation checking is implemented matters as well. For instance, there are several proposed optimizations for OCSP checks that are implemented in very different ways, e.g. OCSP stapling, which can result in significant changes in response times if they are implemented (properly) in that particular browser version.

## OCSP

OCSP [RFC 6960] is a runtime protocol for checking the revocation status of an TLS certificate. It involves making an HTTP request to an OCSP server (called a 'responder'

because it responds to HTTP requests) and checking the response, as shown in Figure 3-7. This is usually done serially, so that if the certificate chain length is two, then two OCSP lookups will be required. OCSP lookups are not encrypted.

Since OCSP uses HTTP, which is layered on top of TCP/IP, performing an OCSP lookup requires one full HTTP request/response cycle, including a DNS lookup for the OCSP service, TCP connection time, and all of the other parts of the E2E, except for the TLS handshake of course. This inevitably takes some time. If the second OCSP lookup is made to the same OCSP server as the first, then the DNS lookup and TCP connection steps can be skipped for the second query, since the IP address for this server will be cached locally. This not always the case however, and secondary OCSP lookups often incur the entire E2E cost. The secondary server may also be in a different physical location and have a completely different E2E response time for this lookup. These 'embedded' lookups are often hidden (for security reasons) from common tools used to test Web performance, and sometimes you will need to use a different browser, such as Firefox, or a proxy such as Fiddler, to examine OCSP response times.

Figure 3-7 shows the waterfall diagram for retrieving a single object over a secure HTTP connection.

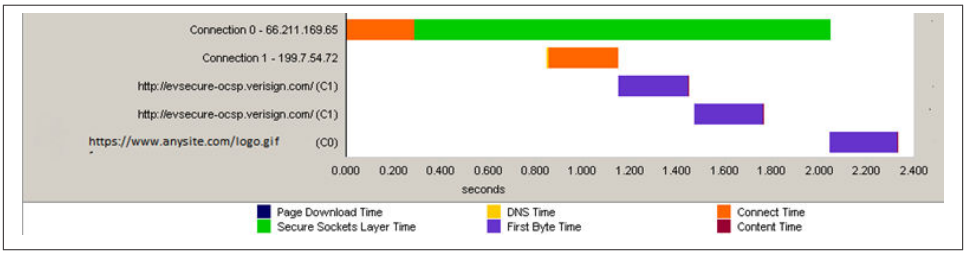


*Figure 3-7. Waterfall chart for a single object over TLS showing the OCSP lookups (Source: Compuware)*

The first entry is the connection to anysite.com, and you can see that the IP address was cached for this connection. Then the TCP connection is made (orange) and the TLS handshake begins. It takes some time before the first OCSP lookup is made, in the second entry in the diagram. The DNS resolution query for the OCSP responder at 199.7.54.72 took very little time (compare Figure 3-8 steps 8-9 on the next page), but a bit more for the TCP connection (steps 10-12). On the next line you can see the OCSP request being made and the content returned (steps 13-14). The T3 time here was also very small (blue) because the objects being returned were small. The second OCSP lookup doesn't require a new DNS lookup or TCP connect but immediately begins as soon as the client can obtain the URL from the first certificate (steps 15-16). The next steps require processing on the client; steps 17-18 are represented by the gap in the waterfall diagram

after the OCSP lookups are complete, taking roughly 200ms. After this, on the fifth line, we can finally see the HTTP response for the original object being sent back to the client.

In Figure 3-8 below, you can see the full-on sequence diagram for T1 including the OCSP lookups. Compare this to the previous waterfall diagram which shows the same sequence. (The last line in the waterfall diagram, showing the HTTP request, is not shown in the sequence diagram.) You can see from the waterfall that the OCSP lookups took considerable time.

Counting the total number of steps in the process, we can see that there are no less than eight (!) roundtrips in the sequence diagram for a secure HTTP connection, and two without it, to at least two and possibly three different servers (the initial DNS server, the HTTP server, and the OCSP server). The initial connection contains these embedded sub-connections, which have their own associated DNS lookups and TCP connection times.

Additionally, we need to consider that the RTT for each of these servers is likely to be rather different. The RTT for your ISP's DNS server should be very short, and the OCSP lookups might also be expected to have shorter RTT times than the TLS connection, because both DNS and OCSP are designed to be very fast real-time lookup or 'responder' systems, widely distributed, just to minimize these overall response times and not hold up the real show, which is the connection to the HTTP server.

The performance of the DNS and OCSP systems are decoupled from the performance of your server; their response times are more closely coupled to the client context (e.g. local caching of DNS and OCSP responses). It's also a good point that both $\mathbf{T}_{\mathbf{DNS}}$ and $\mathbf{T}_{\mathbf{OCSP}}$ are likely to be relatively constant, regardless of the Website you are visiting. This is because your primary DNS server's response time is more or less consistent: either the domain's IP address is cached or not, and so users basically see two DNS response times, 'short' and 'long'. Short (cached) DNS response times should all be very similar. Authoritative queries will not only take longer, but response times will be more variable.

OCSP query response times should also be constant; it shouldn't take the CA's OCSP server longer to respond on behalf of one Website than another. OCSP lookups to the same CA should be more or less constant in their response times, no matter the URL, for a given user in a specific browsing context.

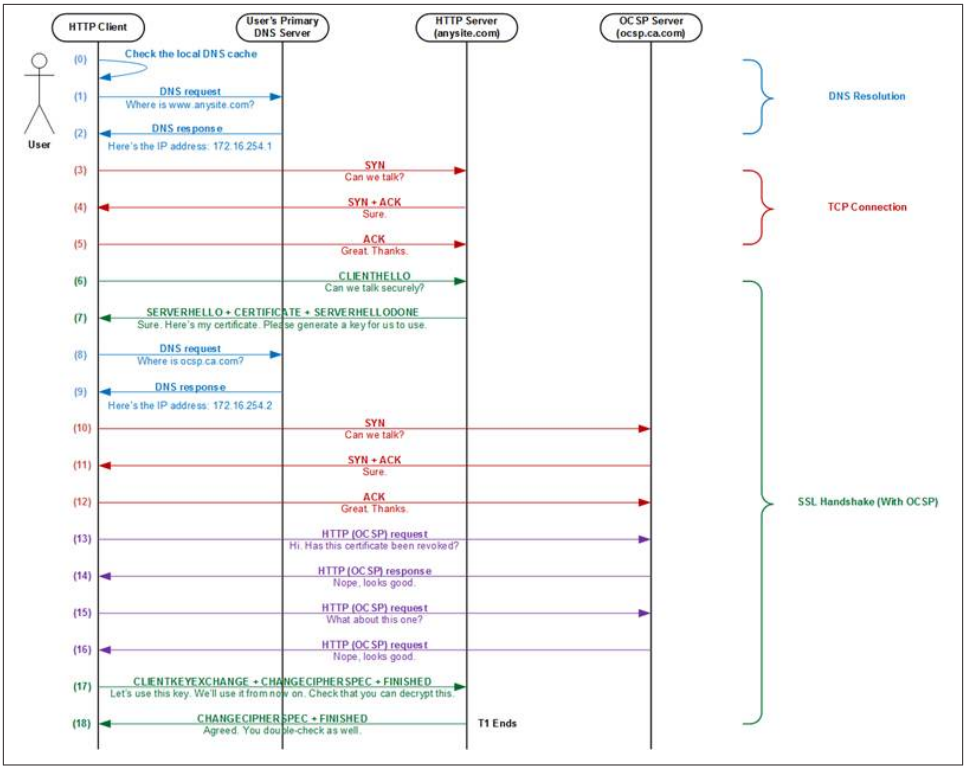


*Figure 3-8. Complete T1 sequence with TLS and OCSP lookups*

This is what really happens, most of the time, when a user goes to a secure Website.

### How Long Does an OCSP Lookup Take?

The response time taken for OCSP lookups can be highly variable, largely depending on the response time of the server (T2), which largely depends on the load on the server. If the server is heavily loaded, response times will increase.

In Figure 3-9, taken from x509labs.com, we can see that average OCSP response times vary considerably from one vendor to another, by as much as 400 ms!

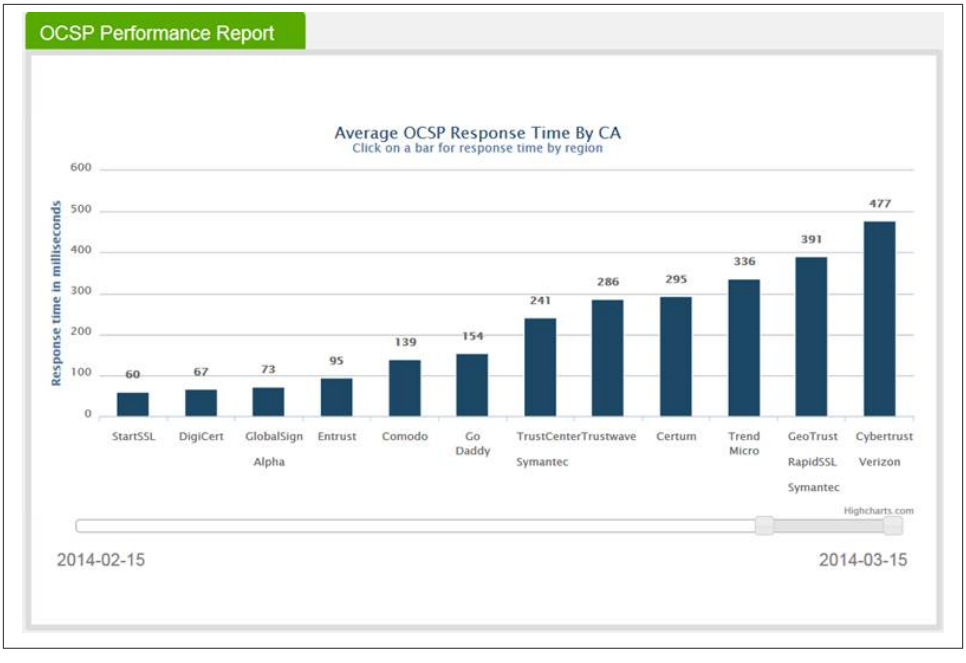


*Figure 3-9. Average OCSP response times by certificate authority.*

Keep in mind that these are averages. Recall that server location is a big part of OCSP response times, and you can see that it's not just the average that's important but the regional breakdown. Let's drill down into OCSP response times by region for a single CA in Figure 3-10 below.

We can see that the minimum average response time, on the East coast of the US, is around 181 ms! If we need to perform two such lookups, we have added up to 362ms to our overall T1 time, which is a very large amount of time to make users wait. With an average OCSP response time of around 850 ms, users of sites in Singapore might wait as long as 1782ms, just for the certificate revocation steps in the TLS handshake, which seems unreasonable.

OCSP lookups are slow, and add response time at the very worst time in the overall E2E, from the user's viewpoint. Since the user agent will be unresponsive for some time after the initial TLS handshake is complete, the user may be faced with a blank screen for some period of time, as discussed in the section on DNS.

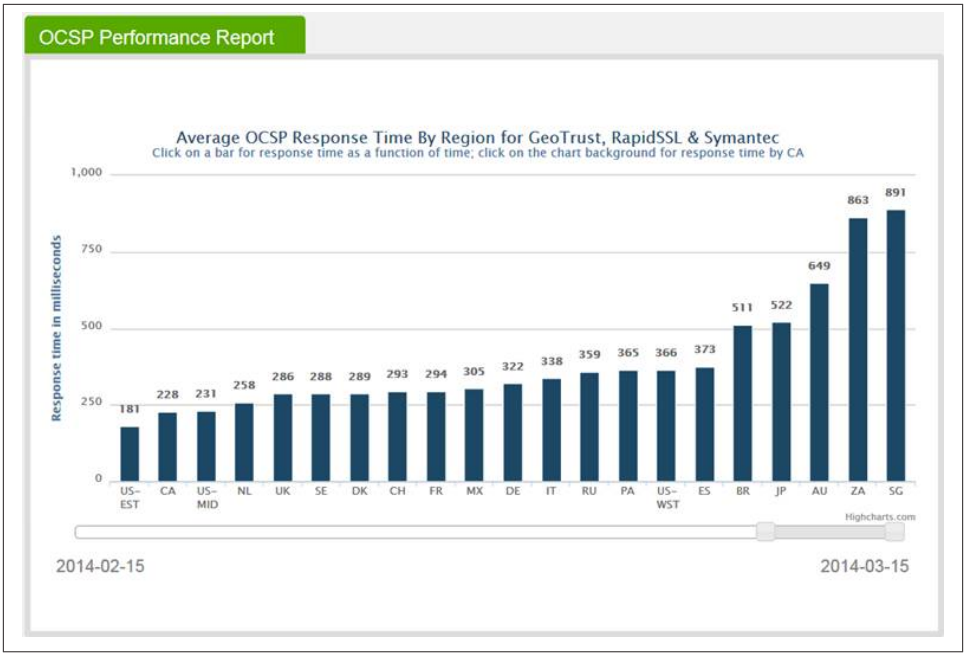


*Figure 3-10. OCSP response times by region, expanding on the previous diagram.*

In figure 3-11 below, we can see that even the most popular and technically sophisticated sites can suffer from long OCSP response times. This waterfall diagram shows the loading of the Twitter home page, *https://www.twitter.com/*, and you can see that the first two entries are for OCSP lookups. The total time taken for the initial OCSP lookups was more than 200ms, and the test was performed over a high-bandwidth connection in Los Angeles, under more-or-less best case conditions.

At the bottom of the waterfall you can see two additional OCSP entries, which correspond to the twitter page calling Google Analytics, a real-time measurement platform. Those OCSP lookups took more than 272 ms as well. This had little effect on the overall initial E2E, because this event did not take place until the rest of the page had already loaded. This is the 3rd step in the MPPC loading process ('deferred loading'). The waterfall diagram clearly illustrates the 3-step process used in our model.

Of course, it can't be stressed enough that these are individual spot tests, under specific circumstances, and we can't reason based on any single measurement. These diagrams are examples, intended to show what's happening, and the response times involved in any particular measurement may vary considerably from one test to another. The average OCSP response times shown above in Figures 3-9 & 3-10 are a better guide as to what we should expect for certificate revocation checks.

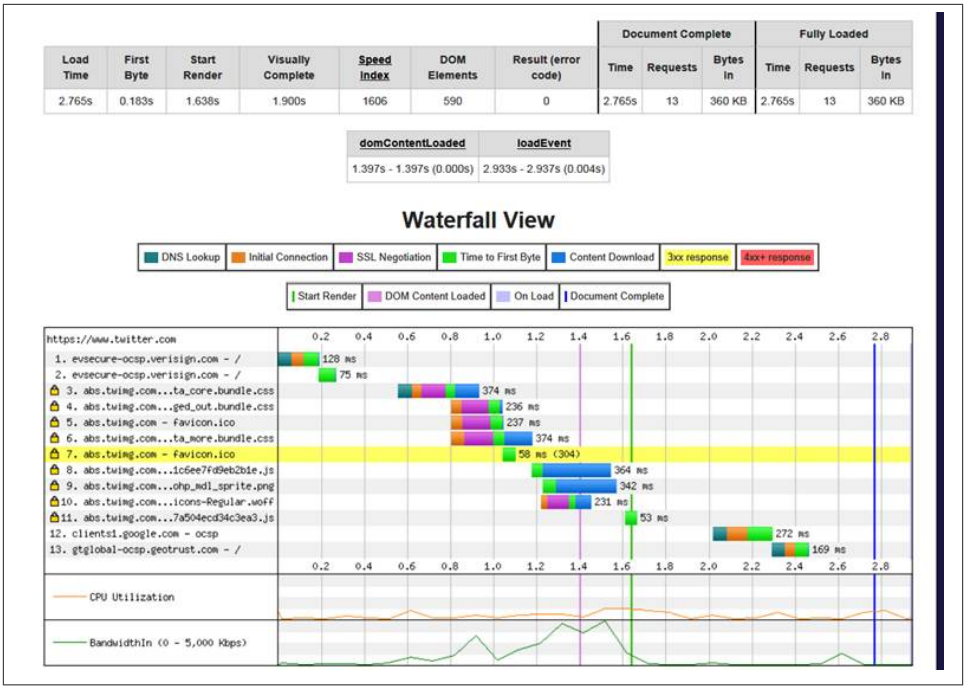


*Figure 3-11. Waterfall diagram for https://www.twitter.com, exposing the OCSP lookup response times*

In this diagram, I'm using a different tool than I used to create the waterfall diagram in Chapter 1 and for Figure 3-7 above. It looks a bit different, and there's some additional information shown. I used webpagetest.org to create this waterfall diagram. We'll take a closer look at this tool, and get a better understanding of the data displayed, in chapters to come.

For now we are interested in the OCSP lookups, which are the first two lines of the waterfall. It's worth noting again that the second lookup has no DNS or TCP connection response times, as the IP address was cached and the connection was persistent, so the second lookup was considerably shorter than the first, being made to the same server.

Since the OCSP server response body itself is just a status code ('0' for 'good' usually), the actual payloads across the TCP channel are each a single packet.

Looking at this data, both the global and regional averages for CAs and the detailed example of how OCSP works in practice, we see that it is not unlikely that each OCSP lookup adds as much as 200-300ms or more to the overall E2E, which is by no means insignificant. OCSP time is a big part of T1, and in some cases, can be the biggest single factor in page load time. TLS is slow in part because OCSP lookups are slow.

*(It's definitely not my intention here or elsewhere in this book to criticize Twitter or its performance. I like to use their homepage as a good example of how to do things right.)*

OCSP also has other, security-related issues. Since OCSP happens over an unsecured connection, it may be subject to interception, allowing for surreptitious tracking, or replay attacks.

### OCSP Caching

One obvious optimization in this process would be for the user agent to cache OCSP responses locally, for some period of time, to avoid repeated lookups. This is exactly what happens, or at least what's supposed to happen. While this technique doesn't improve the response time for the initial connection, it can greatly reduce the time to establish an TLS connection to that domain for all future connections within this session (or for some specific time-to-live), just like DNS caching. This is a very useful optimization that is implemented unevenly across user agents and versions, and it can be very difficult to determine if OCSP responses are being cached for some Fuser agents. It's useful to mention here because you will often encounter it in waterfall diagrams and perhaps wonder why no OCSP lookup is associated with an TLS connection.

This also brings up another point, which is that while we've been focused here on a single connection, in the context of loading a full page, TCP connections will likely be made to multiple domain names. As we'll see later, the number of (parallel persistent) connections that can be made to a given hostname in the page is fixed by the client, OCSP caching means that we will only have to do certificate revocation checks once for each unique hostname on the page, no matter how many objects are downloaded from that domain. This is important when many objects such as scripts, images and style sheets are downloaded from a single domain, such as a CDN. Making sure your OCSP responses are cached on the client is one of the most effective ways to reduce TLS handshake times for a particular TLS session.

Earlier I said that OCSP responses were very small, and usually a '0' is returned. Just to be clear, the OCSP protocol uses *ASN.1* binary syntax to define responses, and there are many different kinds of OCSP messages of different lengths. The '0' returned here is just the shortest possible message, indicating that the certificate has not been revoked.

## CRL

The alternative to OCSP is to use the Certificate Revocation List protocol [RFC 5280], which works rather differently. CRLs preceded OCSP, but the browser industry is moving more toward using OCSP, largely due to the use of EV certificates, as well as offering real-time revocation status.

For CRL revocation checking, each certificate authority publishes a list of the serial numbers of revoked certificates, usually on a periodic basis, typically 24 hours. The user

agent is responsible for obtaining the CRL lists and updating itself accordingly. This is considerably faster than performing OCSP lookups for each certificate, however, since the CRL for each CA need be updated only once for each revocation period, essentially caching a blacklist of revoked certificates.

CRLs require significantly more effort on the part of the user agent, which may be a limiting factor for mobile devices. If the user agent does not have a current CRL from the CA, it will need to make an HTTP call (usually) to get one, in much the same way that an OCSP lookup is performed. The chart below (Figure 3-12) shows response times for CRLs, just as we saw previously for OCSP. As you can see, actually making a CRL lookup call is not very different in response time than for OCSP lookups. The numbers all look rather similar, though not exactly the same.


*Figure 3-12. CRL response times by certificate authority. Compare to figure 3-9 above.*

CRLs can reduce the number of roundtrip times required for an TLS handshake to two, which is a big improvement over OCSP. It does have problems of its own, including being unable to determine if a certificate has been revoked in the last revocation period (CRL doesn't include any notification mechanism). The CRL itself becomes larger over time as well, as more certificates are revoked.

Another thing to take into account for OCSP and CRL is that client-side processing for CRLs can require more horsepower, because of the need to manage, update and parse

these lists. With OCSP the client processing is much simpler on a per-request basis, but it still requires some sort of cache on the client. This may be a consideration for user agents on mobile and device platforms.

OCSP and CRL mechanisms are generally supported, give or take a few optimizations for performance, across almost all different platforms, user agents, and versions. As of version 28, the Firefox browser will cease to support CRLs and move to using OCSP exclusively. Both of these systems have their flaws and advantages.

I've spent a lot of time explaining the verification process during the TLS handshake because the significant effect it can have on Web performance is largely unknown to many workers and the existing tools don't make it easy to learn what's happening. But to be fair, I want to point out that running a PKI (Public Key Infrastructure) business is a serious task, and I have some sympathy for those tasked with building and managing these large-scale OCSP systems. As the number of TLS connections increase, so do OCSP lookups, and because of the real-time nature of OCSP, these systems have to perform well all the time.

If, as many people are suggesting, we begin to use TLS for all Web connections (and I'm not advocating for or against this), then the OCSP providers will become an even more necessary part of the infrastructure, adding hundreds of milliseconds to each Web connection that requires certificate verification.

Not every TLS connection requires certificate verification, and therefore OCSP or CRLs. This depends on the user agent, and some browsers don't perform verification unless the session is using an EV certificate. Other clients may choose not to perform verification for valid reasons. Basically everyone who uses an EV certificate for their site depends heavily on OCSP response times.

## What Makes TLS Slow?

It's controversial to say that TLS is slow in the Web performance community. In the last few years we've heard from conferences, books, and other sources that TLS is no longer a performance problem. As we've seen, this conclusion is not borne out by the data. In fact, the response times for T1 using TLS and without it are significantly different, and in many cases the TLS handshake time dwarfs the other T1 components.

In the Figure 3-13 below, we can see comparative $\mathbf{T}_{\mathbf{TLS}}$ and E2E times for many of the Alexa Top 50 websites, illustrating the differences in handshake times. TLS connections typically add several hundred milliseconds to the overall T1 (and to the E2E) response time. There are also significant differences depending on whether the user is using a 2048-bit EV certificate or not. Sites using EV certificates have longer response times than other sites.

It's important to understand what it means to say that TLS is slow. One commonly quoted statement is "*SSL is not computationally intensive anymore*." But 'computationally in-

tensive' is not the same as 'takes a long time', and from the user's perspective, TLS (and SSL) is slow, especially on mobile devices. The time required to send the various messages back and forth, to verify the certificates, and the computational time required to process all of these steps and perform the encryption (which *is* 'computationally intensive') must all be taken into account when we are trying to optimize our TLS handshake times.

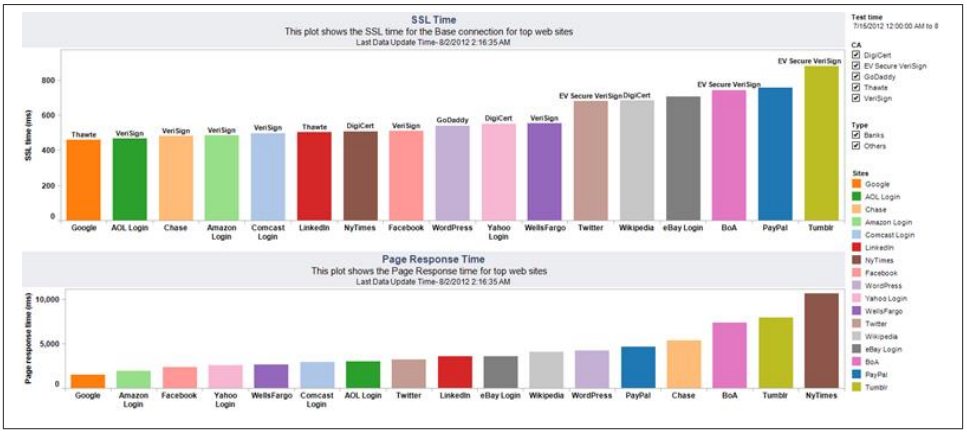


*Figure 3-13. Comparing TLS response times across the Alexa Top 50 sites.*

It's interesting that the time taken for TLS handshakes with 2048-bit keys are all very similar, with a few exceptions, and so are the times taken for 1024-bit TLS connections. Given that the organizations whose data is shown here are all very different, with very different hardware and software setups, we might expect more variation among the processing times for TLS handshakes. That they are pretty clearly grouped into two distinct sets of response times is a good indication that the dominating factor in their TLS handshake response times is not the number of servers or site architecture but lies with the network and the certificates themselves.

Part of this is the way TLS key generation is handled on the machine itself, and part of it is that some steps in the handshake process take the same amount of time regardless of key length or server load. Certificate revocation checking, for example, takes the same amount of time for all connections.

While the very largest companies have the resources to invest in heavily optimizing their TLS traffic, most other sites are left behind. The bad news is that TLS **is** slow. The good news is that there is a lot you can do about it.

Looking back at the sequence diagram in figure 3-8, we can see that in addition to the TCP transport times associated with sending the TLS handshake messages back and forth, there are several time segments that include the time required for certificate ver-

ification and the work that needs to be done by both client and server in the process of encrypting and decrypting the data exchanged in the process. There are several important points to be made on this.

The first is about asymmetric vs. symmetric keys. The TLS handshake uses asymmetric encryption, which is significantly slower than the symmetric encryption used for bulk TLS download. This imposes a higher cost at setup time, with benefits later when data is being transmitted. The second point is about asymmetric vs. symmetric processors. Servers are normally high-powered machines designed to handle a lot of traffic. In fact one of the key factors measured in TLS hardware benchmarking is the handshake capacity. Clients however, especially mobile devices, usually have much less horsepower. Since basically the same processing steps are required on both sides, clients may struggle to do what a larger server does with ease.

I haven't talked much about the size of the certificate message, but this can be an important issue, since larger objects will increase the TCP transport time (T3) of the CERTIFICATE message. We'll talk more about this in the next section, when we talk about optimization.

## Optimizing Your TLS Response Times

The discussion above describes the 'canonical' TLS handshake, before any optimizations are applied. The TLS handshake's complexity make it a good target for optimization and there are many proposed methods for doing so, most of them focused on reducing the number of round trips required to establish an TLS connection, one way or another.

The unfortunate part of the problem is that it's very difficult to take advantage of all of these optimizations for any given connection. Some of them require that a feature be implemented on both client and server; others only apply to recent or dropped connections.

Given sufficient time and resources, you can probably find several ways to reduce the response times for TLS connections to your site, or at least make sure that your site isn't making any obvious errors. Unfortunately, even with all of the optimizations in place, there is an irreducible number of message exchanges, encryption processing cycles, and certificate verification steps required to complete the handshake, and once you've ensured that you've minimized these steps there's not a lot more that you can do beyond that point. As we've seen, even highly optimized sites with a lot of resources can suffer from slow response times for TLS handshakes.

Following up on our original question "Why is TLS slow?" we've established that these are the three main components of the handshake response time as listed above: TLS message exchange, certificate verification, and cryptographic processing.

Cryptographic processing for TLS handshakes depends on the ciphersuite chosen, specifically the asymmetric cipher used, and the machinery (hardware + software) used to do the processing.

Choosing ciphersuites based on performance characteristics alone is hazardous. Remember when we discussed earlier about making tradeoffs among the systemic qualities of performance and security in chapter 1? This provides a good example. Some of the asymmetric ciphers commonly available are very secure, but add precious milliseconds to the processing time, while others may be faster but less secure. How you make these choices will have an impact on your user's experience.

Adding more hardware, the second ingredient of cryptographic processing for TLS handshakes, works, but only up to a point of diminishing returns. Even sites that are not by any means lacking in hardware resources experience hundreds of milliseconds of TLS handshake delays. 'Throwing hardware at it' is the most common response to any performance problem, but it's a very limited solution. This is why the following TLS handshake optimizations are focused on reducing the number of messages exchanged, and the time taken for certificate verification (which are the same thing in some cases).

### Early Termination

Early termination simply means 'terminating' the connection at a server closer to the user, or one with more or better hardware and software for processing. This is basically an TLS proxy. Many CDN providers offer this as a service. In many cases, this is the best way to reduce your TLS handshake response times, especially for sites with a global user base.

This technique is also sometimes used by large sites, which will often terminate at a load balancer equipped with specialized hardware to handle TLS connections. In this case, the proxy server is not necessarily closer geographically to the user, it's just more powerful. Because there's no overall reduction in the RTT, this method is less effective.

Terminating TLS connections by distributing your site's hosts (and applications and content!) is always a good idea, but may not always be practical for smaller sites or other situations.

For sites using early termination, the termination proxy usually establishes persistent connections to the site on our behalf, and this 'backhaul' step is what saves time. But from the user narrative point of view, this is all invisible; the connection is still (seemingly) made to the responding server. Users can't tell if early termination is being used, except for reduced overall response times. But they will be very glad if you are able to implement it!

### Session Resumption

"SSL Session Resumption" [RFC 5070] is a method for reducing reconnection times for clients that have connected and disconnected cleanly within a specified timeout period. If the client sends a new request to the same server, and includes a session ID (and if the server is configured properly), the server will recognize that session ID and can resume the connection with an abbreviated handshake, involving fewer messages sent and received. In particular, the CERTIFICATE, SERVERDONE, and CLIENTKEYEXCHANGE messages are all omitted, which reduces not only the roundtrip time required to send these messages across the Network, but also the processing that needs to be done on either end, basically very little cryptographic processing needs to be done for resumed connections. The session flow for the resumed handshake is also slightly different.

Instead of sending the CERTIFICATE, the server immediately sends the CHANGECIPHERSPEC message and FINISHED messages. The client returns the same messages, after making sure it can decrypt the server's FINISHED message. After the server checks to make sure it can decrypt the client's FINISHED message as well, the connection can be resumed. This 'abbreviated handshake' reduces the message exchange by one full roundtrip, and it reduces the server duration and the client think time as well.

The scheme outlined here is technically called 'SSL session caching'. Another method of accomplishing similar ends is called 'SSL stateless resumption' and is defined in RFC 5077. Session tickets were invented to address some of the problems inherent in managing stateful sessions for many concurrent clients. The flow for stateless resumption is slightly different, but doesn't include any extra roundtrips or significant increases in processing. From a performance point of view they are very similar.

You should be sure that both of these features are configured on your servers. Of course not every server (or ISP!) will support these features, and not every user will see the entire benefit, but it's important for those users making multiple or repeated connections to your site. All modern browsers support the abbreviated handshake by one or the other or both of these methods.

### OCSP Stapling

One still poorly supported but promising method for reducing TLS handshake response times is OCSP Stapling. Formally known as the *TLS Certificate Status Request* extension, OCSP stapling [RFC 6066] is a method of shifting one (and only one) OCSP lookup on the server itself; the server makes regular requests for the OCSP response to its own certificate and caches it. When a request comes in, the OCSP response is 'stapled' to the HTTP response, reducing the verification step by one roundtrip. The second (full) OCSP lookup is performed on the intermediate certificate.

This makes the server responsible for presenting not only its certificate but also a valid revocation check as well. This has the added benefit (in addition to removing one

roundtrip from the TLS handshake) of reducing the load on the OCSP provider, since the response is cached and shared across many sessions. One other benefit is that it reduces the risk somewhat, since only one OCSP lookup need be done.

So OCSP stapling works out well for everyone; clients get shorter handshakes, OCSP providers see reduced load, especially on popular sites, and overall security is increased as well. The bad news is that OCSP stapling is only partially implemented in many clients. Since both server and client must support OCSP stapling for it to work, it will only provide benefits to some users, some of the time. Still, for those that can support it, it's a significant response time improvement. Many commonly used HTTP servers such as Apache and support OCSP stapling, and more browsers are beginning to support it as well.

All of these optimizations are helpful to at least some users in some situations, and usually cost little to implement, since they mostly involve configuration of the terminating server, or working with your ISP to make sure their servers are configured properly. By implementing OCSP stapling and session resumption (using both IDs and session tickets) you'll have covered most of the bases on configuration. Choosing the order in which ciphersuites are presented can possibly shave a little more time off. Early termination works well but isn't for everyone, but all servers terminating TLS connections should support stapling and session resumption.

### Certificate Chain Length

One other area of optimization for TLS is to make sure that your certificate's trust chain is as short as possible, optimally of length three, and therefore requiring only two OCSP (or CRL) lookups. (The third certificate is the root certificate for the CA itself and is already installed in your browser.)

There are certainly legitimate circumstances where the certificate chain may be longer than this, but they are relatively rare.

There are two points to be made about the chain length. The first is to reduce your certificate chain to the minimum value, optimally two, and the second is more of an error to avoid. When your server sends the CERTIFICATE message, make sure that all (both) of the certificates are included, i.e. the domain certificate and the CA intermediary certificate are both sent to the client. If the intermediate certificate is not sent as part of the CERTIFICATE message, the browser will have to fetch it (and then verify it) which will add hundreds of milliseconds to your response time, or in some cases will cause the connection to fail.

However *don't* send the CA's top-level certificate; it's not necessary, since the user agent will already have these certificates in its trust store. Sending the extra data won't trigger an extra OCSP lookup, but it does add extra data on the wire, which is always to be avoided.

## TCP-Level Optimizations

So far the optimization techniques mentioned here have all been at the TLS protocol level, and they've been designed to reduce the number of TLS handshake messages exchanged. Now let's have a look at some of the optimizations available by at the TCP protocol layer. These techniques usually involve trying to send fewer packets, rather than fewer messages, and fewer packets usually means reduced response time. None of these techniques can eliminate any message exchange steps entirely, but they can reduce the amount of data sent over the network.

### Certificate (and Record) Size

It's a general rule in performance work that you should always minimize the number of bytes transferred across the network, for performance, efficiency, and simplicity. TLS handshake messages are no different. Making sure that you are not sending the CA's root certificate so that the CERTIFICATE message fits into the minimum number of packets as mentioned earlier is one approach. (Make sure you do include all of the intermediate certificates regardless of size – multiple packets will cost less than a certificate fetch.) If messages are segmented across multiple packets, the server (or client) must wait until all of the packets in the message have arrived before it can begin processing, which is exactly what you don't want. TCP and how packets are transferred is the topic of the chapter on T3 later in the book.

There's a lot of advice on the Web about TLS message sizes and how to optimize them for various server configurations, depending on your site's needs. I'm not going to go into this in detail, as the information is easily available and really only applies to a limited number of sites. You should know however that these more sophisticated optimization techniques exist and can be applied when needed.

### TLS Compression

TLS provides support for compressing message payloads, similar to using gzip to compress Web content. Compression has some security and performance issues however, and is generally turned off by default. You should avoid TLS compression schemes as a matter of best practice.

### TLS False Start

Of all of the TCP-level optimizations discussed here, the most promising in terms of really making a dent in TLS handshake times is TCP False Start. False Start is simple in concept but not always easy to implement, and is only inconsistently supported at this time.

TLS False Start is a network technique that doesn't involve any changes to the TLS protocol or message exchange patterns. Looking back at Figure 3-8, where we described the TLS handshake process in full, you can see that the client waits to hear from the

server that it has successfully decrypted the client's FINISHED message and receives the server's own FINISHED message before it begins sending any application data. Using TLS False Start, the client simply doesn't wait – it starts sending application data as soon as it's sent its FINISHED message. This cuts out on half of a roundtrip from the total handshake, which is always good news. False Start is another one of those TLS optimizations that's great when it works, but simply won't work for everyone. To configure your server to support TLS False Start, first you'll need a server that supports it (and most of the major servers do) and you'll need to configure it so that it works across different browsers, many of which have their own individual requirements for supporting it, including supporting specific ciphersuites. TLS False Start is well worth supporting, and a good addition to your toolbox of TLS optimizations.

### A Word on Packet Loss

Earlier it was noted that if a packet in a TCP stream is lost, the entire process must begin again. This is true for TLS handshake as for DNS lookups and TCP connections. Any packets lost (or broken) during T1 will prove very costly for that user. Since packet losses on many mobile networks approach 1% or even more, there's a significant chance that any particular connection will encounter a delay.

There's not a lot that you can do about packet loss, unfortunately; it's a network effect. However, being aware of how it affects your E2E is important, especially knowing which scenarios (mobile users, say, on LTE connections) are more subject to packet loss than others, so that you can plan (and set expectations for users) accordingly. Packet loss also puts an upper limit on TCP bandwidth, as we'll discover shortly.

### Finding and Fixing TLS-related problems

If you suspect that TLS handshakes are causing performance-related problems for your users, there are a number of steps you can take.

First you need to identify your site's performance problems, and determine if the problem really resides with the TLS handshake, or with some other issue. You'll also want to convince yourself that the problem is real by comparing your response times to previous data before the problem occurred, or by looking at other similar sites. TLS handshakes are very similar across sites, and if your handshakes are taking longer than other, similar sites then this may indicate a problem.

Once you've determined what the problem is (and this process is the same for T2, T3, or any other aspect of the E2E) then you'll want to quantify it and determine just how big a problem it really is.

This diagnostic process will quickly help you find out where the problems lie in your site's response times. Once you've identified they problem, then you can consider the range of possible solutions. If there's no ready solution to your problem, can you compensate elsewhere in the E2E? If you know for instance that users in a particular area

suffer from long T1 times, and your analysis indicates that there's little you can do about it directly, it may be possible to reduce the overall response time elsewhere in the E2E. Users don't care about the individual steps required to load pages!

### How Much Does TLS Matter for Performance?

The value of $\mathbf{T}_{TLS}$ is often a large contributor the overall E2E for the initial payload for your site, and less of a problem for the secondary intransitive hyperlinks, largely because caching and the use of persistent parallel connections reduces the overall number of lookups, and also because the user can (often) visually see that the page-loading process is proceeding and this may convince them to wait a little longer.

As you drill down into the problem, you'll want to try to identify if the problem resides on your servers, on your ISP's or CDN's servers, with the network connection, or with the user's client software. In most cases, TLS problems are usually either due to configuration problems (missing intermediaries, termination problems) or with hardware capacity. Computationally expensive or not, TLS handshakes require more processing cycles, especially floating point operations, than unsecured connections, and you'll want to take this into account so as not to leave your users waiting.

Many browsers don't expose the certificate verification steps in their developer tools, so many developers don't even know that OCSP lookups are happening. Of course this doesn't make debugging problems any easier. My own experience is that Firefox is the best browser for working on TLS-related problems, because it does expose what's happening and provides data that other browsers do not.

For some sites, the TLS handshake may make the larger part of the total E2E. Despite all of the optimizations done by the best people in the performance business, the E2E for www.google.com is dominated by TLS handshake times. This isn't for lack of trying on their part! There's a certain irreducible level of delay incurred by secure connections that unsecured connections don't face.

The honest answer is that TLS matters a lot. If you are using TLS connections, then you are paying a performance price for the additional security, and it's likely a very significant price, as we noted in Figure 3-13 above, where we saw response times for many popular sites, many of which are dominated by their TLS handshake times.

## Summing Up on T1

Whew! All of that work and we haven't yet sent our user's original HTTP request! The work involved in setting up a connection, especially a secure one, can be pretty daunting, and identifying problems and avenues for optimization requires some knowledge, familiarity with some specialized tools, and as usual with performance issues, patience.

The T1 time segment often receives little attention from performance experts, and most optimizations are targeted at the other time segments. Part of my goal in this book is to direct some attention back on to T1 and its contribution to the E2E. I firmly believe that improving T1 times, especially for the initial connection for the base HTML page, is extremely significant for user's perceptions of your site. Remember that the T1 time for the base page often occurs while displaying exactly nothing – other than a hung screen – to the users. This can be excruciating on mobile devices especially, where users are often focused on completing some specific time-sensitive task.

This is true even if you are not using a traditional browser or delivering traditional Web content. If your REST-based APIs have slow connections, users will know it, even if the cause isn't immediately clear to them. Since the initial connection's T1 time directly involves your servers, rather than say, those of your CDN provider, minimizing that crucial time segment is your responsibility.

This discussion of the T1 response times has largely focused around the TLS handshake, both because it's a huge component of the E2E for those connections that use it, but also because it's one aspect of T1 that you as a site host may have the ability to address directly.

Problems with DNS and TCP times may involve working with ISPs or other network providers and may not be easy or even cost-effective to fix in many cases. Unlike problems with the T2 server duration time, T1 occurs entirely at the network level and requires network-level fixes. For globally distributed sites, there may be cases where differential support for various kinds of optimizations and network conditions place significant constraints on what you can effectively do to improve T1. In these cases, you will have to find other ways to compensate in the overall E2E cycle, in order to achieve reasonable response times.

For example, in Table 1, response times for TCP connections over an LTE mobile network are given as a blood-curdling 650ms. There's virtually nothing that you, as a provider of content and services on the Web, can do to improve this; it will improve over time as the wireless devices and network infrastructure improve. However, you can recognize that your users will face these long connection times, and design your sites so that overall response times don't explode. Knowing is always better than not knowing; given accurate information about the causes of delays in your user experience, you can plan around them even if they can't be addressed directly.

The best problems are always the ones you don't have, and this applies more to T1 problems than others, because it's often difficult and expensive to make fixes at the network level. It may be difficult to get attention on the problem once it's 'in production' as well. The overall T1 time budget is usually only a few hundred milliseconds and it can be a difficult decision to spend time and money reducing what seems like a quite small time already.

Make sure, as you build your site and grow your user base, that you've anticipated all of the possible T1 problems and made sure they won't occur. Smart choices about network providers and certificate authorities are among the best preventive medicine for T1, as well having the tools in place to measure your user's experience, so that you can quickly find and fix problems, and also so you (and your users) can have realistic expectations, based on real data.

Now, onward to the HTTP request!